POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Flexible RISC-Based Hardware Accelerator for Random Forests

Supervisors

Prof. Daniele JAHIER PAGLIARI

Dr. Chen XIE

Dr. Alessio BURRELLO

Candidate

Yali KANG

Academic Year 2023-2024

Abstract

Random Forests (RFs) have emerged as a powerful machine learning(ML) technique, offering high accuracy for a variety of tasks. Hardware accelerators are crucial for the growing demand for high accuracy, more efficient processing capabilities in RFs applications, particularly involving big data and real-time processing. This thesis introduces an innovative RISC-based accelerator design that prioritizes flexibility, addressing a significant gap in current designs that struggle with frequent RFs model updates.

In this thesis, a pipelined RISC-based hardware accelerator and a parallel RISC-based hardware accelerator have been developed. The design of each accelerator unfolds in several stages, beginning with adopting RISC principles, and transforming the RFs model into different types of instruction sets, then dynamic mapping instructions in Field-Programmable Gate Arrays (FPGAs) memory. The mapping method enables efficient RFs classification without the need for extensive hardware reconfiguration, significantly reducing both time and costs. A specific hardware architecture consisting of multiple processing elements (PEs) is used to efficiently support the inference of RFs, by encoding and executing the generated instructions. The final design has been deployed on the PYNQ Z2 FPGA board to validate the function and evaluate the performance of the implemented hardware accelerator.

Experimental results showed speedup compared with an ARM Cortex-A9 processor, analyzed resource utilization, and power consumption by supporting the various RFs models with different depths and numbers of trees. In particular, the initial architecture employs a pipelined approach. By configuring the FPGA's Programmable Logic (PL) section to leverage 30 Block RAMs (BRAMs), this hardware accelerator demonstrates remarkable flexibility, efficiently evaluating all tested RFs models. It achieves a maximum speedup of about 34.69x. Notably, this setup maintains consistent latency and throughput across all test RF models, regardless of variations in tree depth and number of trees, at 45 ns and 22.22 MS/s (Million samples per second), respectively. The power consumption of the final design is 0.171 watts. The results show a notable enhancement in the acceleration performance as the complexity of the RFs model increases. The architecture incurs significant occupation of BRAM resources and Look-Up Table (LUT). The second accelerator implementation uses a parallel architecture, when employing the identical RFs models as utilized in the pipeline results, the parallel accelerator achieves a maximum acceleration ratio of 16.99x. At its optimal speed-up, the parallel accelerator exhibits a latency of 76 ns and a throughput of 13.16 MS/s. The power consumption is 0.149 watts. Although the parallel accelerator exhibits a slightly lower speedup ratio compared to the pipelined accelerator, it vastly surpasses the latter in terms of resource efficiency. The BRAM resources required by the pipelined accelerator are 6 times that of the parallel accelerator, and this figure exponentially increases with the complexity of the RFs models, represented by tree depth and the number of trees. Therefore, by using the same number of BRAM, the parallel hardware accelerator supports a larger number of trees for classification compared to the pipelined structure.

Acknowledgements

Thanks to my dear families. My dear mother, my father, and my younger brother. I have gotten a lot of support, encouragement, and love from them. They will always be my love.

Table of Contents

List of Tables

List of Algorithms

List of Figures

Acronyms

RFs

Random Forests

CPU

Central Processing Unit

GPU

Graphic Processing Unit

FPGAs

Field-Programmable Gate Arrays

ASIC

Application-Specific Integrated Circuit

RISC

Reduced Instruction Set Computer

NOInst

Node Operation Instruction

FLInst

First Layer Instruction

CTInst

Control Instruction

MSB

Most Significant Bit

CU	Control Unit
PE	Processing Element
NodeRA	Node Relative Address
HDL	Hardware Description Language
IP	Intellectual Property
PS	Processing System
PL	Programmable Logic
NCU	Node Computing Unit
IOT	Internet of Things
SOC	System on Chip
ML	Machine Learning
IF	Instruction Fetch
ID	Instruction Decode

BRAM

Block RAM

LUT

Look-Up Table

MS/s

Million Samples per Second

Chapter 1

Introduction

1.1 Motivation

Random Forests (RFs) [1] is a popular and powerful ensemble learning method, offering robust performance for classification and regression problems. The algorithm combines multiple decision trees and outputs the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. RFs showed great performance in the number of variables much larger than the number of observations [2].

Due to its simplicity, ease of parallelization, and high accuracy, RFs has been widely used in various practical applications, including, but not limited to medical diagnosis, environmental modeling, financial forecasting, [3][4][5][6] etc. For instance, in the domain of remote sensing, RFs have been extensively utilized to extract information from multispectral, radar, and thermal remote sensing imagery, showing its ability to handle high data dimensionality and multicollinearity effectively [3]. Another significant application of RFs is in the medical field, where they have been employed for disease diagnosis and prognosis, including predicting diabetes and its complications, demonstrating the capability of algorithms to deal with complex biological data [4]. Moreover, in the financial sector, RFs have been applied for credit scoring and fraud detection, benefitting from their ability to model the nonlinear relationships often present in financial data [5]. The versatility of RFs also extends to environmental science, where they have been used to predict water quality and model species distribution, highlighting their effectiveness in ecological modeling and conservation planning [6].

However, in all of these scenarios, as the volume of data and feature dimensions increase, achieving high accuracy with RFs becomes computationally intensive, demanding substantial resources for both training and inference. This computational demand poses challenges to deploying RFs on memory and energy-constrained Internet of Things (IOT) edge nodes, significantly limiting application scenarios [7].

A hardware accelerator is a specialized processor designed to perform specific functions more efficiently to overcome this limitation. Hardware accelerators tailored for specific tasks have become a promising paradigm, to significantly improve the inference efficiency of RFs than the original inference execution on a general-purpose Central Processing Unit (CPU). By optimizing the computational tasks running on the specific hardware, these accelerators can perform the necessary calculations with lower energy consumption and higher speed. This is crucial for sustainable computing and for applications where power availability is limited, such as embedded systems.

The platform to implement hardware accelerators can be categorized into the Graphics Processing Unit (GPU), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuit(ASIC). They provide significantly speed up the training and inference of RFs.

ASICs typically provide the highest performance and energy efficiency. However, compared to FPGAs, their logic is fixed after manufacturing and cannot be reprogrammed. Therefore, the trade-off between flexibility and efficiency can be explored by making the design partially programmable or configurable. In the case of RFs, this trade-off translates into the one between an accelerator capable of executing a single, specific, RF model, and one able to support a variety of models through programmable instructions. The former could achieve extreme efficiency, but would not be able to support changes in the model structure (e.g. number or depth of the trees in the ensemble), while the latter would.

This thesis aims to design a high-flexibility RFs hardware accelerator, belonging to the latter group (i.e., partially programmable). Although our final target is an ASIC deployment, we prototype our accelerator architectures on FPGA, due to their low cost and fast reprogrammability.

Currently, much research focuses on how to train random forest models quickly or speed up the inference of RFs. However, the flexibility of hardware accelerators for RFs has been largely underestimated. Addressing this gap in research, this thesis proposes to implement two variants of RISC-based hardware accelerators for RFs. In particular, one implementation involves fully pipelined stages, while another achieves inference through highly parallel processing. The key to the implemented accelerators is to achieve highly flexible solutions that can accommodate modifications to random forest models without requiring hardware adjustments, thereby reducing the time and financial costs associated with hardware reconfiguration.

1.2 Thesis Structure

This thesis is organized as follows.

- **Chapter 1** provides a brief introduction to RISC-based flexible hardware accelerators for RFs, and the corresponding motivation and context.
- **Chapter 2** presents the background respect to RFs algorithms, FPGAs hardware platform, and RISC architecture.
- **Chapter 3** provides a detailed summary of the state-of-the-art FPGAs-based hardware accelerator for RFs and the limitations and challenges in existing solutions
- **Chapter 4** illustrates the first of the two proposed RF accelerator architectures.
- **Chapter 5** presents a second variant of the implemented hardware accelerator, based on a highly parallel processing strategy.
- **Chapter 6** shows the experimental results of the evaluation of the two variants of implemented hardware accelerators and provides a detailed analysis and discussion.
- **Chapter 7** concludes the main contributions of the thesis and suggestions for future work.

Chapter 2

Scientific Background

2.1 Random Forests

Ensemble learning is a machine learning (ML) technique which in multiple models, often called `classifiers` are combined to solve a particular problem [8][9]. This ML method is widely used thanks to its superior classification and regression performance on target tasks.[10]. By pooling the predictions of multiple models, ensemble methods can compensate for the limitations of a single model, often resulting in better performance on varied datasets.

Random Forests (RFs) stand out as a prominent ensemble learning method for classification. RFs build on the concept of decision trees, which are simple yet powerful for handling complex decision-making tasks. However, a single decision tree can be prone to overfitting, where it performs well on training data but poorly on unseen data. RFs addresses this by creating a `forest` of decision trees, each trained on a random subset of the training data and using a random subset of features at each split [1]. The aggregation of predictions from multiple trees through majority voting further mitigates the risk of overfitting, making RF-based solutions popular for classification tasks across various domains.

In RFs, each decision tree makes a a classification prediction by independently processing the input data through its hierarchical structure, based on the features selected during training. Each tree produces a vote for a particular class based on its learned patterns. For classification tasks, the collective decision is often derived via a consensus mechanism, where the output class is determined by garnering the majority of votes from the ensemble of trees.

Algorithm 1 proposed an example for RFs classification. In line 1, the function for random forest classification is defined. Line 2 introduces a variable named *votes* for storing classification outcomes, which is then initialized to 0. Starting from line 3, the algorithm iterates over all trees within the random forest. The detailed classification process is delineated from lines 4 to 11. Beginning with the root node of a tree, if it is not a leaf node, the input feature value is compared with the node's threshold to determine the direction of the next node, either left or right. This process is repeated for the subsequent node until a leaf node is encountered. The class label of the leaf node is stored in the votes array, as demonstrated in lines 12 and 13. After obtaining the results from all trees, a majority voting is conducted in line 15 to output the final classification result.

Algorithm 1 Random Forest Classification

```
 1: function RANDOM FOREST CLASSIFICATION($forest$, $instance$)
 2:     $votes \leftarrow$ array for class votes initialized to 0
 3:     for $tree$ in $forest$ do
 4:         $n \leftarrow \text{Root}(tree)$
 5:         while $n$ not in Leaves($tree$) do
 6:             if $instance[\text{Feature}(n)] > \text{Threshold}(n)$ then
 7:                 $n \leftarrow \text{RightChild}(n)$
 8:             else
 9:                 $n \leftarrow \text{LeftChild}(n)$
10:             end if
11:         end while
12:         $P \leftarrow \text{Prediction}(n)$
13:         $votes[P] \leftarrow votes[P] + 1$
14:     end for
15:     $final\ prediction \leftarrow$ class with maximum votes in $votes$
16:     return $final\ prediction$
17: end function
```

2.2 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are semiconductor devices that are based on a matrix of configurable logic blocks (CLBs) connected via programmable interconnects [11]. This allows FPGAs to be programmed to perform a wide array of digital computations, making them highly versatile and adaptable to various applications.

The development of FPGAs involves using Hardware Description Languages (HDLs), such as VHDL or Verilog, which describe the logic and behavior of the digital circuit to be implemented on the FPGAs. Modern FPGAs design tools also provide high-level synthesis options, allowing developers to work with higher-level programming languages, thereby simplifying the design process and making it more accessible.

Unlike CPUs that execute instructions sequentially, FPGAs can process multiple operations in parallel. By decomposing the target algorithms into a set of concurrent tasks, FPGAs can achieve significant computation acceleration for specific applications, thanks to its parallel processing capability.

One of the typical features of FPGAs is their reconfigurability. They are reprogrammable to execute different logical functions or tasks. This allows the same FPGAs device to be used in different applications or stages of a project, offering a flexible and cost-effective solution for digital design [12].

FPGAs are utilized in a wide range of applications, from signal processing, data analytics, and financial modeling to embedded systems, telecommunications, and high-performance computing [13]. Their adaptability also makes them ideal for prototyping and research, where design flexibility and iterative development are critical. Considering these distinct advantages of FPGAs, this work is based on an FPGAs platform, to develop a flexible hardware accelerator for RFs. FPGAs offer unparalleled flexibility and reconfigurability, enabling the hardware to be precisely tailored to fit the requirements.

2.3 Reduced Instruction Set Computer

Reduced Instruction Set Computer, or RISC, is a CPU design philosophy that emphasizes efficiency through a simplified set of instructions. This approach contrasts with Complex Instruction Set Computer (CISC), which utilizes a comparatively larger set of instructions [14]. The RISC philosophy is predicated on the observation that a simplified instruction set can perform operations more quickly and efficiently, as it allows faster execution times and lower power consumption. The key principles and innovations of the RISC architecture include [15] [16]:
`Simplified Instruction Set:` RISC architectures streamline the number of instructions by focusing on a smaller set of simple and general instructions. This simplification enables each instruction to be executed within a single clock cycle, improving the processor's speed.

`Load/Store Architecture:` In RISC systems, memory operations are limited to specific instructions (load to bring data from memory to CPU registers and store to write data from registers back to memory). This separation of memory and arithmetic/logical instructions simplifies the executions and allows for more efficient pipelining.

`Pipelining:` Pipelining is a fundamental feature of RISC processors, where multiple instructions are executed concurrently by overlapping their execution stages. A RISC processor divides the processing of instructions into separate stages, allowing each stage to handle a different instruction simultaneously. This process increases the throughput of instructions, enhancing overall performance.

`Uniform Instruction Format:` RISC architectures employ a fixed-length instruction format, which simplifies instruction decoding and execution. This uniformity allows for simpler hardware design and enhances the efficiency of the stages including Instruction Fetch (IF) and Instruction Decode (ID).

In this thesis, the implemented hardware accelerator is inspired by the characteristics of RISC architectures, including uniform instruction format, load architecture, and pipelining, which will be instrumental. These features are fundamental to improving the efficiency and performance of design in this thesis. The detailed process and implementation of these RISC features in the accelerators will be thoroughly elucidated in Chapter 4.

Chapter 3

Related Works

This chapter reviews existing literature on the acceleration of RFs classifiers and decision tree-based models on different hardware platforms, including FPGA, GPU, and multi-core processors.

3.1 GPU based Acceleration

GPUs are well-known for their high throughput in parallel computations, making them ideal for accelerating complex ML algorithms. Recent research has explored the potential of GPUs in speeding up RFs classification and decision tree-based models, achieving considerable reductions in execution time compared to CPU-based implementations.

The paper [17] proposes accelerating RFs classification on GPUs by introducing a hierarchical memory layout, which balances the efficiency of compressed sparse row (CSR) format with more regular array-based representations. The approach divides each decision tree into subtrees, optimizing for GPU's memory hierarchy to reduce irregular memory accesses and improve data locality. Three variants of tree traversal code were developed to explore different parallelization approaches and memory layouts. This method significantly enhances performance over the CSR baseline, demonstrating speed-ups from 5x to 9x over CSR and up to 2x over Nvidia's cuML library on an Nvidia Xp GPU.

3.2 FPGA based Acceleration

FPGAs have become popular due to their reconfigurability, which enables fine-tuning of hardware resources to meet specific algorithmic requirements. This feature is particularly beneficial for the implementation of an RFs classifier, where the parallelism of decision trees can be exploited to enhance performance and efficiency.

Several previous works have targeted hardware acceleration to RFs. But all of these works are less flexible due to the constrain of BRAM depth or BRAM number in FPGA. Mingyu Zhu et al.[18] proposed an FPGA-based hardware accelerator for Deep Forest models, aiming to achieve high speed up, and lower power consumption. Their approach involves designing an efficient Node Computing Unit (NCU) for rapid logic calculations and an optimized storage scheme for effective tree storage within limited memory resources. Furthermore, the proposed architecture incorporates specialized hardware and adaptive dataflow to manage node computing imbalances during classification, thereby achieving high accuracy and low power consumption. This paper focuses on minimizing the amount of node information stored in memory to reduce the usage of BRAM resources. However, due to the depth limitations of BRAM, once the depth of the tree exceeds the limit, it requires a reconfiguration of the FPGA. Xiang Lin et al. [19] explored three FPGA architectures for RFs in their paper, aiming at multiple applications. They investigated memory-centric, comparator-centric, and synthesis-centric architectures, focusing on the trade-offs between area reconfiguration and context switch time. Among these three architectures, The memory-centric architecture uses the most FPGA resources but has the lowest context switch time. The flexibility of the memory-centric method is limited by BRAM depth and resource utilization is also a crucial problem for resource-limited application

Studies have demonstrated the capability of FPGA to achieve significant speedups in model inference times while maintaining high accuracy. For instance, Adrián Alcolea and Javier Resano [20] propose an FPGA accelerator for Gradient Boosting Decision Trees (GBDT) aimed at embedded systems to optimize execution while reducing energy consumption. They focus on a case study involving pixel classification of hyperspectral images. This accelerator achieved 2X speed up and 72X less energy compared with a high-performance processor running optimized software. Compared to an embedded processor, it is 30 times faster and consumes 23 times less energy. This design is similar to the first related work demonstrated. Use a method to minimize the node information to 32-bit to reduce memory usage. In contrast, this limitation makes it more sensitive to model changes resulting in more FPGA reconfiguration work.

Shuang Zhao et al. [21] proposed a novel flexible random forest accelerator based on FPGA. This work targets the accelerator suit RFs models that frequently change, aiming to reduce the need for reconfiguration on FPGAs and enhance flexibility. It proposes an optimization based on the memory C approach, organizing random trees in levels where each BRAM stores information for a single level, significantly boosting flexibility. However, this method can greatly consume resources, leading to waste in scenarios where models have small depths but are numerous.

Given the focus on flexibility, this thesis addresses a critical gap identified in various approaches to hardware acceleration for RFs models. Existing research primarily concentrates on speedup performance, resource, and energy consumption on platforms like FPGAs and GPUs. The exploration of flexibility, particularly in accommodating model modifications without hardware adjustments, while maintaining high acceleration ratios and optimizing resource usage, is still not thoroughly addressed. While this flexibility is not critical for FPGAs, which can be reconfigured easily and at runtime, it would be fundamental if one would like to realize a dedicated RF accelerator in ASIC technology. This thesis introduces two distinct accelerator architectures designed for deployment across various random forest tree models. It seeks to find a balance between flexibility, resource utilization, and acceleration ratios suitable for different applications of Random Forests (RFs) models, which vary in tree depth and the number of trees. This thesis's emphasis on flexibility could lead to innovative solutions that offer not only high performance but also the adaptability needed for RFs hardware accelerators.

Chapter 4

Pipelined Hardware Accelerator

4.1 Overall Flow

This thesis does not focus on the training aspect of RFs models. Instead, it focuses on the inference acceleration of trained RFs. The first accelerator architecture implemented uses a pipelined architecture allowing simultaneous memory access can significantly accelerate the inference of RFs. This accelerator is strongly inspired by Shuang Zhao et al. [21]. Generally, the design process can be divided into the following parts as shown in Figure 4.1.

Step 1: Define a specialized instruction set that can support the efficient inference for RFs.

Step 2: In the software platform, use a specific script to encode the RFs model into a set of instructions in the format defined in step 1..

Step 3: Develop the hardware to support the instruction set. This hardware accelerator is written by the SystemVerilog hardware description language. After the accelerator top file is simulated correctly, package it and add it as a part of a block design in the Xilinx Vivado tool. Then synthesize and implement this block design, generate bitstream, and export the hardware. Next, use the Xilinx Vitis tool to load (mapping) the instruction set containing the efficient RFs model information into the memory of FPGAs and execute them to infer the classification.

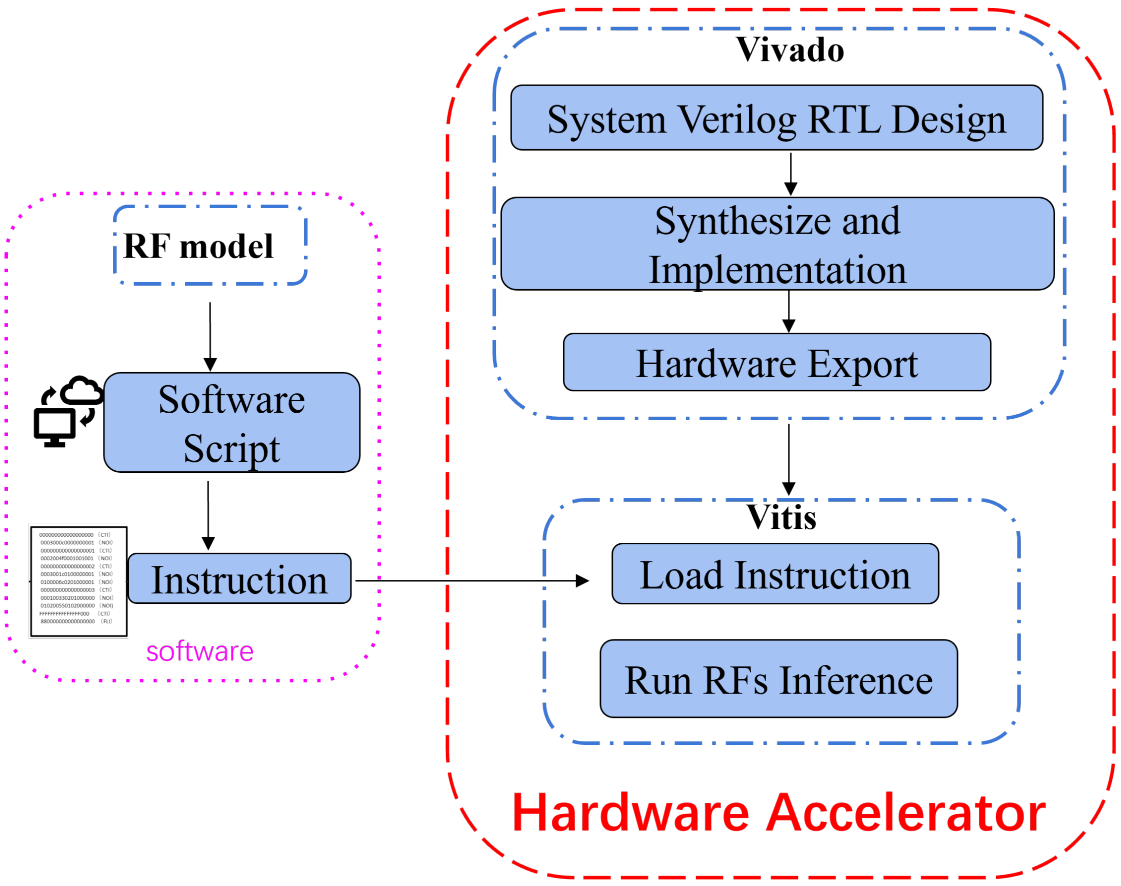


Figure 4.1: Overall flow of the RFs accelerator

4.2 Instruction Set

Building on the understanding of the prediction process within random forest algorithms, a RISC-oriented instruction set to encode the RFs model efficiently has been designed. Embracing the RISC principles, this instruction set comprises three distinct types of instructions, each precisely 72 bits in length. The simplicity and uniformity of this instruction architecture facilitate streamlined decoding and execution, aligning with the RISC philosophy of optimizing performance through a reduced set of simple instructions. This design choice simplifies hardware implementation and RFs inference process, leading to potential gains in both speed and energy efficiency.

The first type is the Node Operation Instruction (NOInst). Other types of instruction are First-layer instruction (FLInst), and Control Instruction (CTInst). The information of the node is stored mainly inside NOInst. FLInst is used to indicate the root node of each tree. Only these two types of instructions participate in the prediction process. CTInst only used to control load the NOInst into BRAM and FLInst into a register.

4.2.1 Node Operation Instruction

As shown in Figure 4.2, Each NOInst stores information for one internal node. The information of the leaf nodes is not stored specially in a NOInst but is stored as additional information for the internal nodes. NOInst can be divided into 7

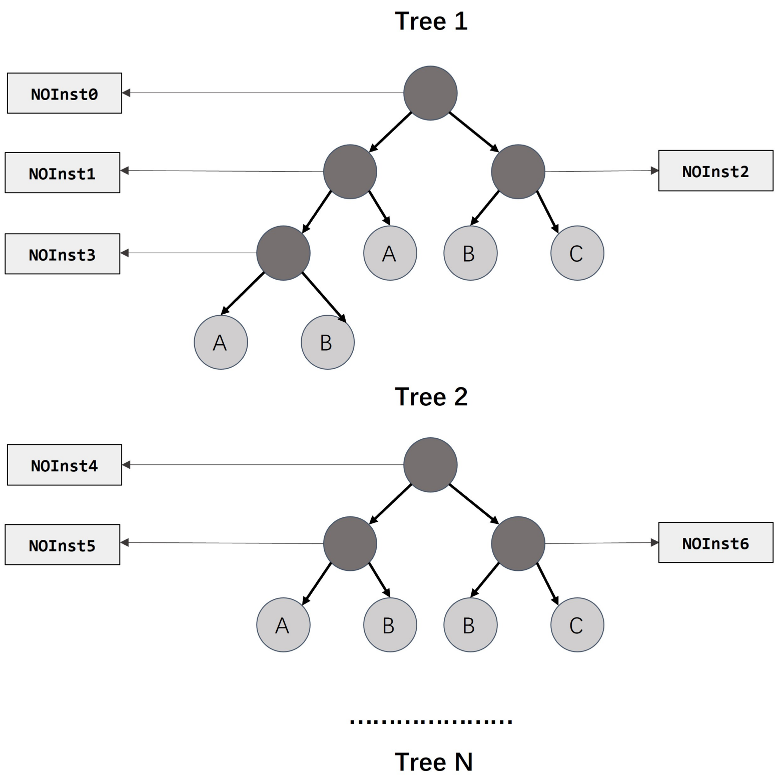


Figure 4.2: NOInst for internal nodes

segments as in Figure 4.3. All of them will be detailed and illustrated as follows.

Node Relative Address.	Feature_ID. (Attribute_ID.)	Threshold.	Left Child Node Info.	Right Child Node Info.	Left Child Node Type	Right Child Node Type.
71-64(8bit).	63-56(8bit).	55-40(16bit).	39-32(8bit).	31-24(8bit).	23-12(12bit).	11-0(12bit).

Figure 4.3: Node operation instruction

Node Relative Address

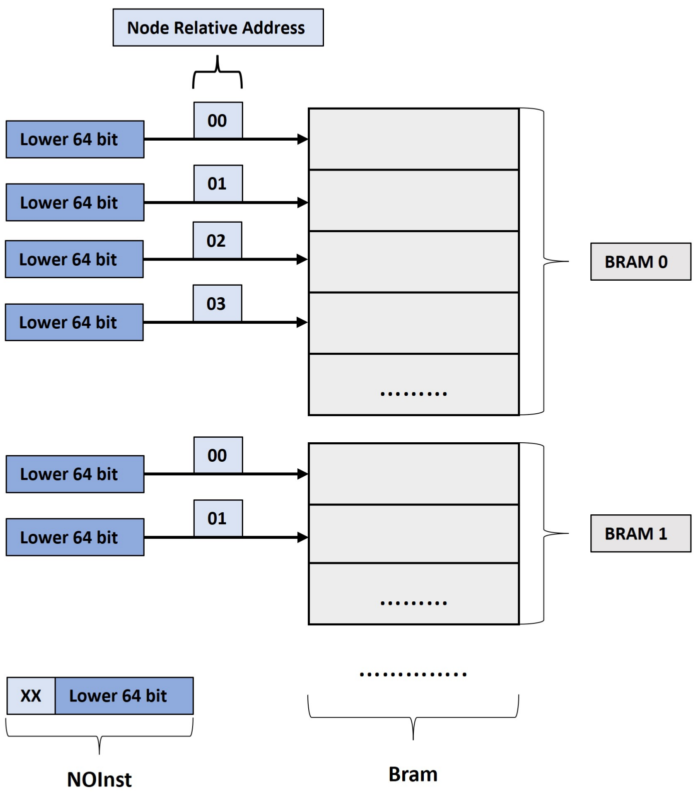


Figure 4.4: Node Relative Addres

Node relative address is used to do the mapping instructions into BRAM of FPGAs. It is the highest 8 bits in NOInst. As demonstrated before, NOInst will be loaded into BRAM, but not all 72 bits will be stored, only the lower 64-bit which is used to perform inference will be stored in memory. The NOInst will be loaded into different BRAM, this process will be controlled by CTInst. This mapping method will be detailed later. Unlike CTInst, the relative address of the node helps in mapping the instruction in a specific BRAM. It denotes the relative address where the lower 64-bit segment is allocated storage within a solitary BRAM. As shown in Figure 4.4.

Threshold

The threshold is the constant value of each node in a trained model RFs that performs compare and branch operations. It accounts for 12 bits in the NOInst format. During the inference, from the root node of the tree, compare a specific feature value with the threshold in that node to decide the next branch. This process will be iterated until it reaches a leaf node that contains a class label.

Feature ID

Feature ID is an 8-bit information. It is used to indicate which feature value will be used. The samples that needed to be classified contained several features. In each internal node, only one feature value will be used to compare with the threshold of that node. The Feature ID decides which feature is needed to do a comparison in the node. This information will be used as an address in hardware parts. On the hardware side, the sample information will be stored in a register. When performing the prediction process, it will be decoded to address that register to get the correct feature value that is needed.

Child Node Information

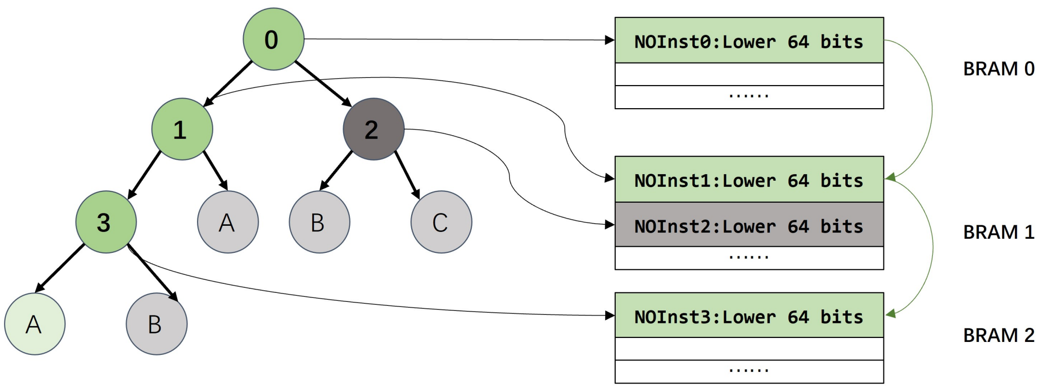


Figure 4.5: Prediction in a tree and corresponding instructions

If the current internal node is not a leaf node, child node information stores the node-relative information as demonstrated before.

Like the instruction executed in a RISC system, in this hardware system, first, it requires addressing the instructions needed. This corresponds to the prediction process in an individual tree of the RFs model shown in Figure 4.5. The system first addresses the NOInst0, which contains the root node information. After decoding the instruction and executing the comparison operation, it decides to execute node 1 in the next step. Then address the BRAM to get the NOInst1 that contains the information of node 1. The branch direction is determined by the comparison result of the feature and threshold in each internal node. There exists plenty of BRAM to store instructions. If already know which BRAM stores the next instruction, it still needs the relative address in the specific BRAM to address the instruction since one BRAM has a lot of address information. That address information is in the child node information.

There is another possibility that the children node is not an internal node but a leaf node. In this scenario, the child node information stores the class label of the leaf nodes.

- **Left Child Node Information** stores 8-bit child node information of the left child node.
- **Right Child Node Information** stores 8-bit child node information of the right child node.

Child Node Type

Similar to the information of the child nodes, the information of the type of child presents two different meanings depending on whether the child node of the current internal node is a leaf node or not. If the next child node is not a leaf node, then it implies the target BRAM for next NOInst. As it can be seen in Figure 4.5, NOInsts have been stored in different BRAM. When the address process, the child node type will help to find the relative address from the current BRAM to the target BRAM. A noticeable thing is that this relative address can be 0, which means that the next instruction will always be executed in a different BRAM and never be in the current BRAM.

If the child node of the current internal node is a leaf node, then 0 will be stored in this segment. This will be used to indicate whether a tree has finished the inference.

- **Left Child Node Type** stores 12-bit child node type information of the left child node.
- **Right Child Node Type** stores 12-bit child node type information of the right child node.

The NOInst will be used in different phases. First, during the mapping process, CTInst and **node relative address** will both be used to store the lower 64 bits of NOInst in BRAM. Then in the inference process, an instruction will be addressed. In this instruction, **feature ID** will be used as an address to obtain a feature value, then compared with the **threshold** in this instruction. If the feature is larger than the threshold, the next part of the execution will be based on **left child node information** and **left child node type**. Otherwise, it will be based on **right child node information** and **right child node type**. For instance, assume that the feature value is greater than the threshold. If the left child node type is not equal to 0, then it will be used to find the target BRAM. The information from the left child node will be used as the relative address of that specific BRAM. If the left child node type is 0, then it means that a leaf node is reached and the predication label is in the left child node information.

4.2.2 First Layer Instruction

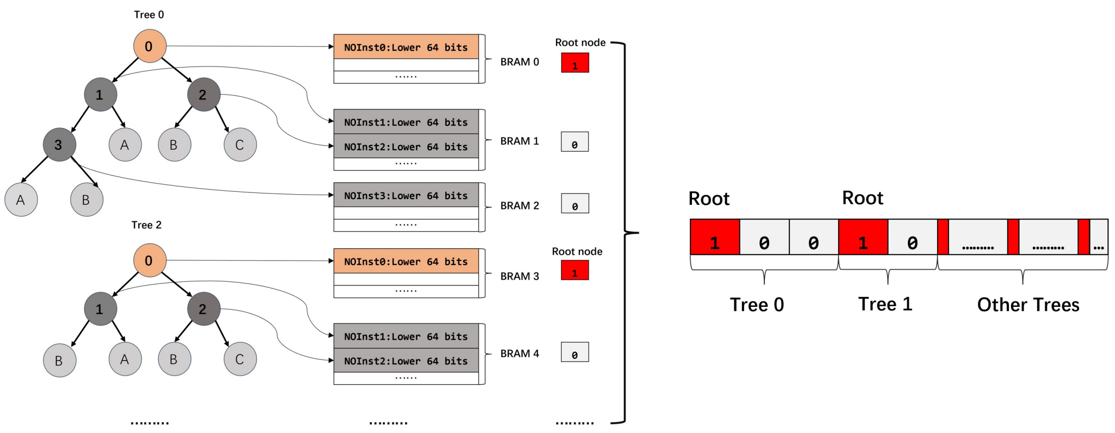


Figure 4.6: FLInst Format

FLInst is 72 bits and it will be stored in a specific register. Each bit of FLInst indicates whether the BRAM has a root node of one tree. Figurev4.6 clearly showed the meaning of it. Internal nodes loaded into different BRAM. If the BRAM stores a root node, then a bit of FLInst will be set to 1, otherwise, it will be set to 0. The most significant bit (MSB) indicates the BRAM 0 that stores the root node of tree 0.

The FLInst will be used to infer the whole structure of the RFs model in hardware. For example, in Figure 4.6, it can be seen that 72 BRAMs were used to store 5 trees. Tree 0 uses 3 BRAMs, and tree 1 uses two BRAMs. The whole structure can be obtained if all FLInst were analyzed.

4.2.3 Control Instruction

As discussed before, CTInst is used to store NOInsts and FLinsts in the memory. CTInst has two segments as in Figure 4.7, the higher 60 bits is the instruction type.

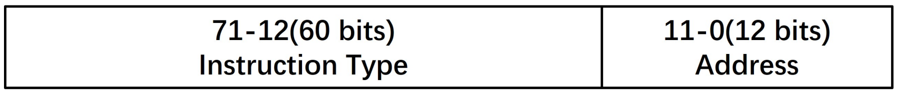


Figure 4.7: CTInst format

For the NOInst, this part will be set as 0, and for the FLinst, all 60 bits will be set as 1. The lower bits indicate which BRAM the instruction should be loaded for NOInst or the relative address of a specific register for FLInst.

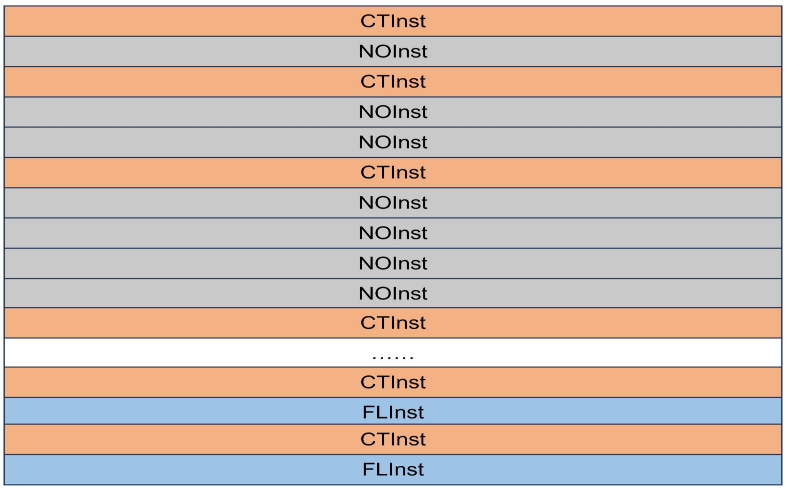


Figure 4.8: An example of whole instruction for an RFs model

A noticeable thing is in the load instruction phase, a CTInst means a group load operation for NOInst but only means one load operation for FLInst. This is because after converting an RFs model into a set of instructions, a CTInst will follow more than one NOInst, but only one FLInst.

4.3 Instructions Mapping Method

After generating instructions, these instructions need to be mapped to BRAM to support further operations. Figure 4.9 shows the working method of this step. It assumed that each BRAM can store 3 64-bit instructions. All BRAMs are organized adjacently.

This approach can be seen from the Figure 4.9.

Step 1: Divide trees in RFs into layers: Start by dividing each tree within the RFs model into layers. This ensures that the NOInsts can be processed layer by layer,

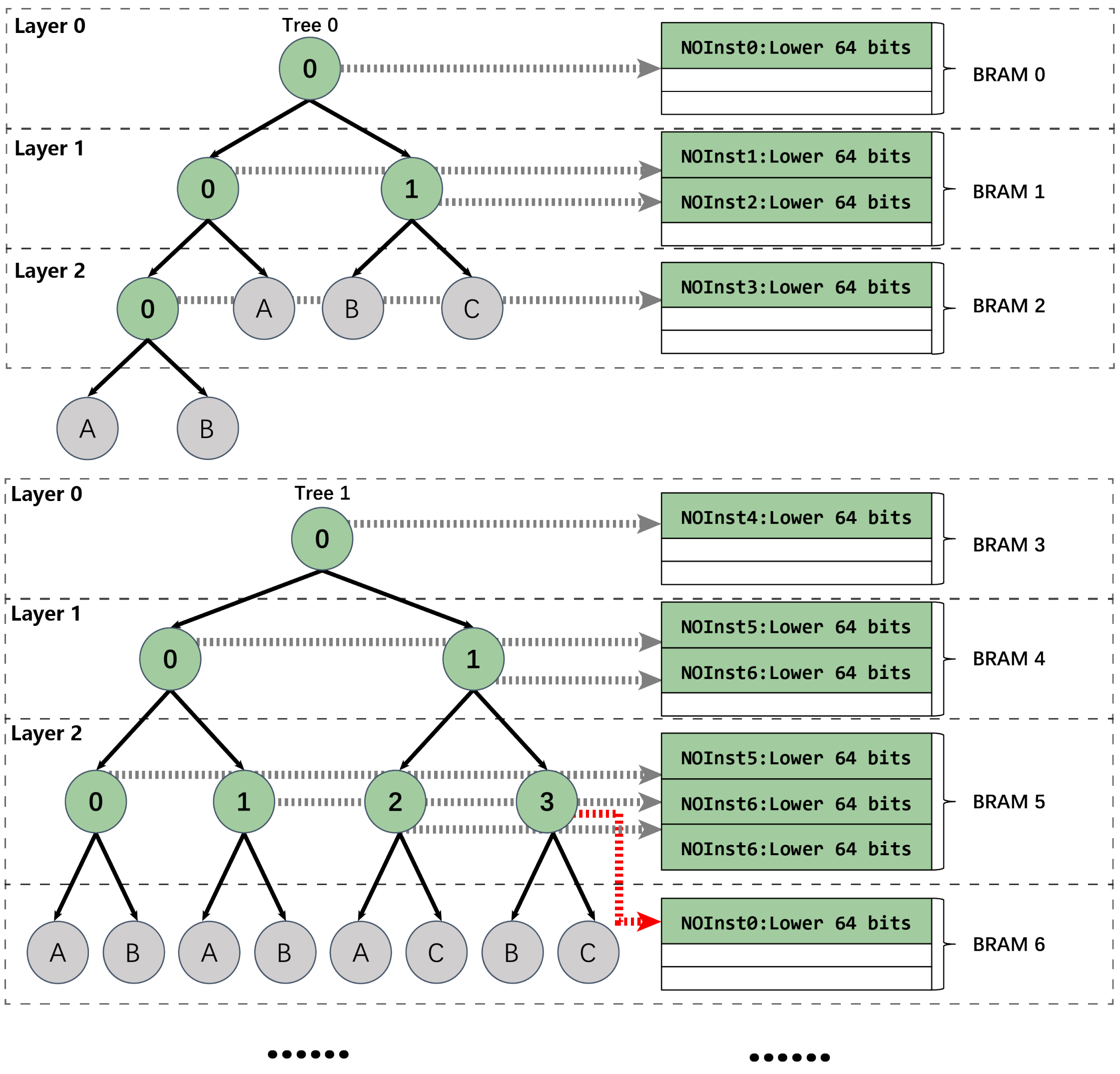


Figure 4.9: Mapping Method of Pipelined Hardware Accelerator

from the top layer down to the bottom.

Step 2: Store NOInsts layer by layer, from left to right: For each layer, start storing nodes from the leftmost node to a BRAM. If the nodes of a layer are too many to fit in a single BRAM, then store the remaining nodes in the adjacent next BRAM. It is important that each BRAM only stores nodes from the same layer to maintain the integrity of the layer structure.

Step 3: Follow storage principles for cross-tree node storage: After storing all layers of one tree, begin storing the nodes of the next tree in the same manner (from top to bottom, from left to right). During this process, it's necessary to adhere to the previously mentioned storage principles.

Step 4: Sequentially store nodes of all trees: Start with the 0th tree and proceed to store the nodes of each tree following the steps mentioned above, until all nodes of the last tree (the n-th tree) have been stored. This method ensures that all nodes of the entire random forest are organized and stored in a clear hierarchical and sequential order, facilitating subsequent data access and processing.

For more details, Figure 4.9 shows one example, in particular, store the root NOInst of tree 0 in BRAM 0 at first. Then store node 0 of layer 1 of tree 0 at the address 0 of BRAM 1. Next, still load node 1 of layer 1 to the second address 1 of BRAM 1. Then begin to store other layers. It should be noticed that for layer 2 of tree 1, BRAM 5 is not enough to store all NOInst, so BRAM 6 is used to store node 3 in this layer. NOInsts from other layers will never stored in this BRAM since one BRAM is only allowed to store NOInst of one layer.

4.4 Instruction generation

In this section, a discussion of the process to generate instructions for a trained RF model [22] is presented.

4.4.1 Function of NOInst and CLInst Generation

As mentioned above, CLInst will be used to map both NOInst and FLInst. In this part, the generation process of NOInst and CLInst used for NOInst is detailed. The steps are shown below.

Get Node Depth

This part uses a function to obtain the depth of each node in a tree as show in Algorithm 2.

CTInst Generation

This part is to iterate each tree of the RF model and organize all trees by layers. Then for each layer, assign the BRAM ID to it. The process is like the following Algorithm 3.

Sets up various counters and lists to track the allocation of nodes across BRAMs, the instruction sets, and other details from line 3 to line 9. Then iterates over each

Algorithm 2 Node Depth Calculation

```
 1: function GetNodeDepths($Tree$, $NodeID = 0$, $Depth = 0$)
 2:     Initialize $NodeDepths$ as a dictionary with $NodeID$ as the key and $Depth$ as the value
 3:     $LeftChildID \leftarrow Tree.children\_left[NodeID]$
 4:     $RightChildID \leftarrow Tree.children\_right[NodeID]$
 5:     if $LeftChildID \neq -1$ then
 6:         $LeftChildDepths \leftarrow$ GetNodeDepths($Tree$, $LeftChildID$, $Depth + 1$)
 7:         Update $NodeDepths$ with $LeftChildDepths$
 8:     end if
 9:     if $RightChildID \neq -1$ then
10:         $RightChildDepths \leftarrow$ GetNodeDepths($Tree$, $RightChildID$, $Depth + 1$)
11:         Update $NodeDepths$ with $RightChildDepths$
12:     end if
13:     return $NodeDepths$
14: end function
```

tree in the classifier. Marks the root BRAM ID of each tree in line 11 for FLInst generation. Calculate the depth of each node to organize nodes by layer in line 16 to line 18. Determines how many BRAMs are needed for each layer based on a single BRAM depth, then generates CTInst from line 19 to line 27. From line 38 to line 40, it assigns each node to a BRAM and calculates its address within that BRAM to get NOInst in the next step.

NOInst Generation

Each internal node (nodes with children) generates an instruction that includes details about the node's feature ID, threshold, child node information, child node type, and node relative address). Algorithm 4 shows the process.

Identify the internal node from all nodes in lines 2-4. Generate the child node type and child node information for each internal node based on the child node from line 6 to line 7. Construct the NOInst for each internal node in lines 8 to line 11. Finally return NOInst, CLinst, node index of each layer from left to right, and maker of root node BRAM ID.

4.4.2 Function of FLInst and CLInst Generation

Algorithm 5 introduces a function to generate FLInst and CTInst based on the makers of root nodes.

Algorithm 3 BRAM ID and Node Relative Address Calculation

```
 1: function GETOINST($clf$, $MaxNodesPerBRAM$, $M$)
 2:                                                         ▷ Initialize variables
 3:     $CurrentBRAMId \leftarrow 0$
 4:     $TotalLayer \leftarrow 0$
 5:     $NoInst \leftarrow array$
 6:     $ClInst \leftarrow array$
 7:     $NodeIndexInLayer \leftarrow array$
 8:     $NtreeNlayer \leftarrow array$
 9:     $V \leftarrow [0] * M$                              ▷ Tracks if a BRAM is used for root node
10:     for each $tree$ in $clf.estimators$ do
11:         $V[CurrentBRAMId] \leftarrow 1$                ▷ Mark the current BRAM as used
12:                          ▷ Initialize lists for BRAM ID and address for each node
13:         $RamId \leftarrow [0] * tree.node\_number$
14:         $Ra \leftarrow [0] * tree.node\_number$
15:                                         ▷ Retrieve nodes and calculate depths
16:         $Nodes \leftarrow tree.nodes$
17:         $NodeDepths \leftarrow$ GETNODEDEPTHS($tree$)
18:         $MaxDepth \leftarrow tree.max\_depth$
19:         for $Layer$ from 0 to $MaxDepth$ do
20:                                    ▷ Identify internal nodes in the current layer
21:             $LayerNodes \leftarrow$ [index for index, node in $Nodes$ if $NodeDepths$[index] == $Layer$
    and not leaf node
22:             if $LayerNodes$ then
23:                 Generate layer instruction ($ClInst$) and calculate needed BRAMs
24:                 $ClAddr \leftarrow CurrentBRAMId$
25:                 $ClIns \leftarrow (0 << 12 | ClAddr \ \& \ 0xFFF)$
26:                 Append $ClIns$ to $ClInst$
27:                 $NeededBRAM \leftarrow (\text{len}(LayerNodes)//MaxNodesPerBRAM) + 1$
28:                 $Index \leftarrow 0$
29:                 $TotalLayer \leftarrow TotalLayer + 1$
30:                 for each $j$ in $LayerNodes$ do
31:                     $RamId[j] \leftarrow CurrentBRAMId + (Index//MaxNodesPerBRAM)$
32:                     $Ra[j] \leftarrow Index \% MaxNodesPerBRAM$
33:                     $Index \leftarrow Index + 1$
34:                 end for
35:                 $CurrentBRAMId \leftarrow CurrentBRAMId + NeededBRAM$
36:                 Append $LayerNodes$ to $NodeIndexInLayer$
37:             end if
38:         end for
39:         Append $TotalLayer$ to $NtreeNlayer$
40:     end for
41: end function
```

Calculate the number of FLInst needed based on the total number of BRAM and the length of an instruction in line 2. Obtain the root node indicators for each tree from the GETOInst function in line 3. Get the address of the register that stores

Algorithm 4 NOInst Generation

```
 1:                                                      ▷ Generate NOInst for each internal node
 2: for each $k$ in range(0, $tree.node\_node$) do
 3:     Identify internal nodes
 4:     $NInternalNode \leftarrow [k$ for $k$, $node$ in enumerate($Nodes$) if $tree.children\_left[k] \neq -1$ and $tree.children\_right[k] \neq -1]$
 5:     $NodeNoInst \leftarrow$ Initialize list of node_count elements to 0
 6:     for each $k$ in $NInternalNode$ do
 7:         Determine node type and node information of left and right children
 8:         Construct NOInst for the node
 9:         Update $NodeNoInst$ for $k$
10:     end for
11:     Append $NodeNoInst$ to $NoInst$
12: end for
13:                                                      ▷ Check if the allocated BRAMs are sufficient
14: if $CurrentBRAMId > M$ then
15:     Print "The number of BRAMs allocated is not enough!"
16: end if
17: return $NoInst$, $ClInst$, $NtreeNlayer$, $NodeIndexInLayer$, $V$
```

Algorithm 5 FLInst Generation

```
 1:                                                      ▷ Calculate the number of instruction lines needed
 2: $NFLins \leftarrow M \div$ instruction_length + 1
 3: $Node\_indicator \leftarrow$ GET_OINST($clf, max\_nodes\_per\_BRAM, M$)[4]
 4: function GETFLINST($NodeIndicator$, $M$, $NFLins$, $InstructionLength$)
 5:                                                      ▷ Initialize CTInst and FLInst arrays with zeros
 6:     Initialize $CTInst$ as an array of size $NFLins$ with all elements set to 0
 7:     Initialize $FLInst$ as an array of size $NFLins$ with all elements set to 0
 8:     for $i$ from 0 to $M - 1$ do
 9:         Calculate the register ID ($RegID$) based on the current BRAM index and instruction length
10:         $RegID \leftarrow i \div InstructionLength$
11:         Calculate the reverse address within the instruction line ($FlRa$)
12:         $FlRa \leftarrow InstructionLength - (i \mod InstructionLength) - 1$
13:         Set the control instruction for the current register ID
14:         $CTInst[RegID] \leftarrow (0xFFFFFFFFFFFFFFFF \ll 12) \vee (RegID \ \& \ 0xFFF)$
15:         Update the $FLInst$ at this register ID with the node indicator
16:         $FLInst[RegID] \vee= (NodeIndicator[i] \ll FlRa)$
17:     end for
18:     return $FLInst$, $CTInst$
19: end function
```

FLInsts and use it to generate CTInst from line 8 to line 14. Generate FLInsts at last.

4.5 Hardware Design

4.5.1 Overview of Hardware Architecture

As shown in Figure 4.10, the hardware system is divided into two parts, majority voting belongs to datapath, datapath is responsible for the classification inference, while the controller for generating control signals for the accelerator.Generally, the operational status is as follows.

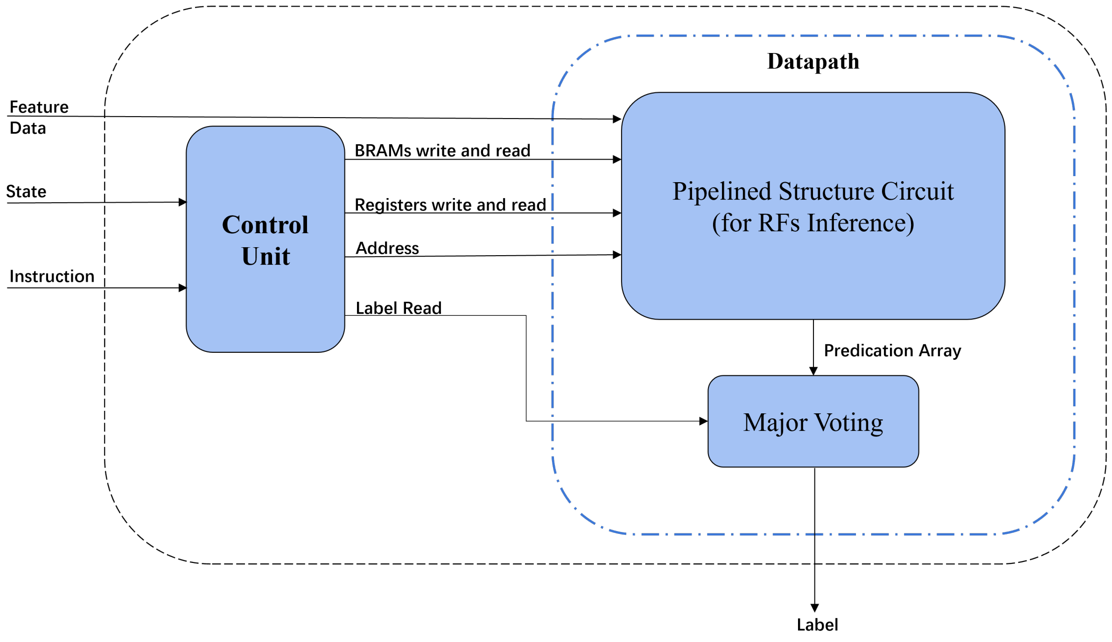


Figure 4.10: Overall framework of hardware accelerator

In the state of *load*, commands are loaded into the FPGAs for storage in the FLinst the NOInst. When the state is *run*, samples are fed into the accelerator, followed by the execution of the predictions. In the *idle* state, no operations are performed, and the system waits. The function of each unit is:

The `control unit` receives all bits of instruction and controls the reading, writing, and address decoding operations of devices within the datapath and the major voting unit, based on the type and content of instructions.

The `datapath` contains a pipelined structure circuit and a major voting unit. The pipelined structure circuit is a particularly complex segment responsible for most of the classification process. The major voting unit determines the final class label based on the frequency of occurrence of predicted labels from various trees.

4.5.2 Datapath

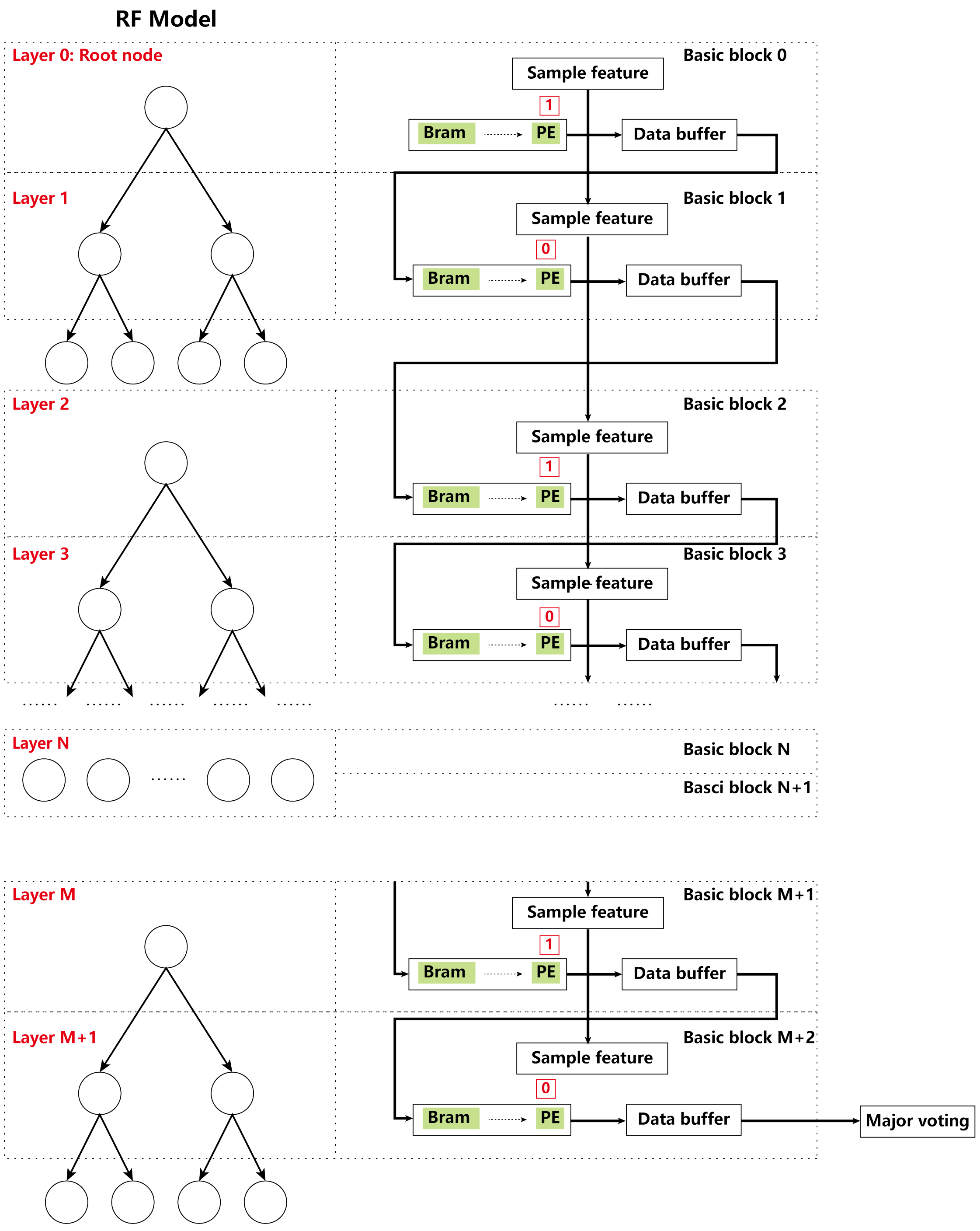


Figure 4.11: The Main Architecture of Datapath

In Figure 4.11, BRAMs are structured as interconnected units, serving the purpose of storing the lower 64 bits of NOInsts. Each BRAM is used to store only one layer. In instances where a single BRAM's capacity is insufficient for the NOInsts of a specific layer, these instructions can be sequentially allocated across multiple BRAMs, as illustrated by Layer N in the figure. Adjacent to the PE, each red rectangle is a single bit of all FLInst indicating the root node. The presence of a *1* indicates that the BRAM is allocated to store a root node. The dynamic mapping of NOInsts to the BRAMs requires that each BRAM is equipped with a Processing Element (PE) to facilitate pipelined comparison operations. Additionally, the architecture incorporates buffers designed to retain intermediate data, which include both the relative BRAM ID and Node relative Address of the next NOInst to be executed and their corresponding prediction outcomes. This arrangement ensures a more efficient handling and retrieval of data, facilitating the smooth operation of the system.

4.5.3 Processing Element

Let's first direct our attention towards the PE's input ports in Figure 4.12.

As previously mentioned, the `BRAM_ID` is utilized to retrieve the relative location of the BRAM containing the next instruction to be executed. Within the PE, the BRAM ID undergoes a subtraction operation. When the output of this operation equals zero, it indicates that the required NOInst is located within the BRAM corresponding to this PE. Consequently, it can fetch this instruction to perform the comparison operation.

The `NodeRA` (Node Relative Address) also plays a crucial role in fetching the next NOInst, originating from the output of the previous Basic Block. When the BRAM within this PE does not store the root node, the NodeRA is employed to address the BRAM to retrieve the corresponding instruction.

The port `attribute` refers to the value of a feature that is used for comparison with a threshold. This attribute is utilized exclusively during the comparison operations performed by the Processing Element (PE).

The `data_in` parameter represents the lower 64 bits of the NOInst. During load operations within the accelerator, `data_in` is written into the BRAM. This process is integral to the accelerator's functionality, facilitating the efficient handling and storage of instruction data necessary for subsequent processing and execution tasks.

The `is_root_node` flag originates from the FLInst (First Level Instruction), serving as an indicator of the node's status within the structure. When set to 1, it means that the node is the root node. Under such circumstances, the NodeRA becomes invalid and the addressing mechanism for the BRAM automatically defaults to an address of 0. This mechanism ensures efficient instruction retrieval in scenarios where the PE's BRAM stores the root node.

The `Prediction` port within the figure is an array designated for storing the class label outcomes of individual trees within the RFs model. This array is derived from the output of the preceding basic block. The internal data of this prediction array undergoes modification exclusively upon the traversal of a tree to its leaf node. This mechanism ensures that the prediction outcomes remain static until the leaf node of a tree is reached, thereby reflecting the final classified results of the RFs model with precision and reliability.

During the Run step of the hardware accelerator, `read_enable` for BRAM will be set as 1.

In the load step of the hardware accelerator, `write_enable` for BRAM will be set as 1.

The `write_address` is also employed during the load phase of operations. It is derived from the upper 8 bits of the NOInst, the Node Relative Address section. This allocation of bits for the write address facilitates the identification and selection of the precise location within the BRAM where NOInst should be written during the load operations.

`PE_state` serves as an indicator of the specific task to be executed by the PE. It can be in one of three states, each signifying a different operational mode:

- Value 00: This state indicates that the next NOInst to be executed is not located in this BRAM. Consequently, no computation is performed during this state.
- Value 01: When the PE state is 01, it means that the NOInst required for execution resides within the current BRAM. Under this condition, the PE proceeds to fetch the instruction based on the input and performs the comparison operation, moving forward with the processing workflow.
- Value 10: This state indicates that the tree traversal has culminated in obtaining a class label. In this state, no comparison operations are conducted. But if this BRAM contains a root node NOInst, at that juncture, regardless of the `PE_state` value, comparison operations resume to facilitate further decision-making processes within the Random Forest model.

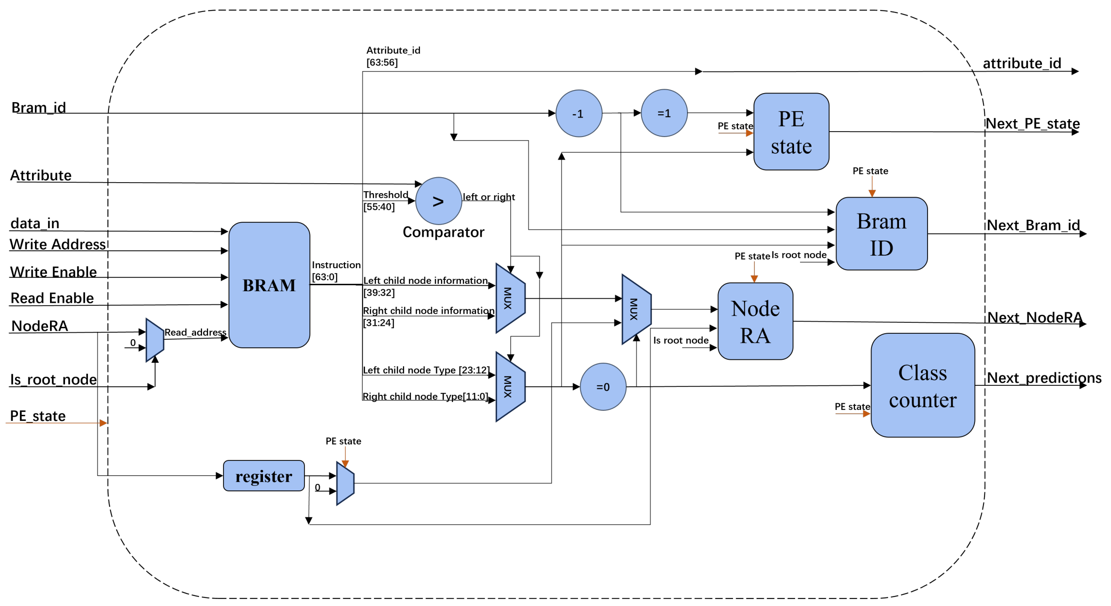


Figure 4.12: Processing Element Circuit

The next part discusses output ports.

The `Next_BRAM_ID` port is strategically connected to the input of the subsequent basic block. This connection is pivotal for the flow of information and instructions through the accelerator's architecture. The value assigned to the `Next_BRAM_ID` depends critically on the current state of the Processing Element (PE) as follows:

- When the PE state is 00: the value of the `Next_BRAM_ID` is set to the current "BRAM id" minus one. This adjustment facilitates the sequential navigation through the BRAMs to locate the required NOInst.
- When the PE state is 01: The value for the `Next_BRAM_ID` is then derived from the fetched NOInst's Child Node Type.

The port `Next_NodeRA` is linked to the current state of the PE

- When the PE state is 00: In such a scenario, `Next_NodeRA` is set to mirror the input `NodeRA` albeit with a one-clock cycle delay. This delay accounts for the time taken to pipeline the NodeRA to the next stage of operation, ensuring that the data flow remains consistent and uninterrupted.
- When the PE state is 01: The value of the `Next_NodeRA` is derived directly from the NOInst's Child Node Information.

`Attribute_ID` is routed to the input address of the register array associated with input samples, serving the purpose of addressing the appropriate feature value.

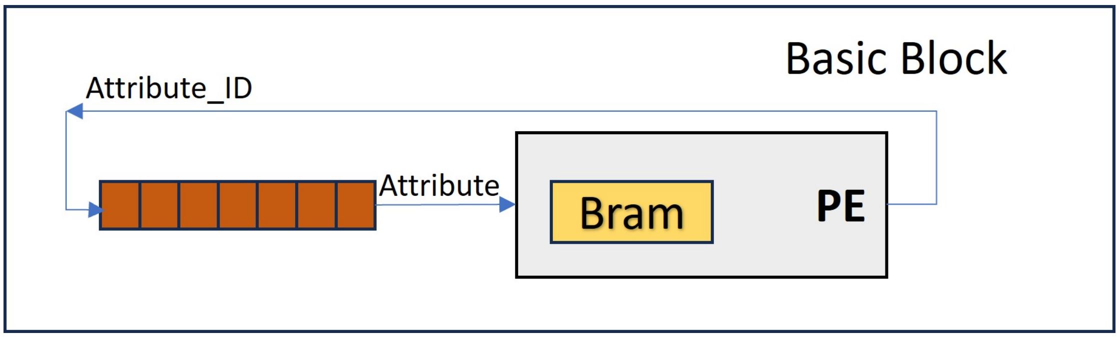


Figure 4.13: Basic Block

Subsequently, this feature value is provided as input to the PE unit. The value of the Attribute_ID exclusively originates from the NOInst housed within the current BRAM. It remains internal to the Basic Block, with no transmission to the subsequent Basic Block as in Figure 4.13.

Both `Next_PE_state` and `Next_prediction` ports serve as connections to the input of the following basic block. The roles and implications of these signals have previously been outlined in the input ports section.

4.5.4 Prediction Process

This section will elaborate in detail on the classification process based on the hardware accelerator.

Step 1: Initialize the input variables `NodeRA`, `BRAM_ID`, `Prediction Array` to 0.

Step 2: Loop through all available BRAMs to find the relevant NOInst for classification.

Step 3: In each BRAM, fetch the NOinst located at the address specified by `BRAM_ID` and `NodeRA`. This instruction contains the `Attribute_ID` and threshold for decision-making.

Step 4: Use `Attribute_ID` to address the samples register array and get the feature value.

Step 5: Determine the direction (left or right) for the next node based on the current attribute's value compared to the threshold in the fetched instruction.

Step 6: Handle the fetched node based on its type. If the node is a leaf node, update the `Prediction Array` by incrementing the count for the predicted class. If the node is not a leaf, adjust `BRAM_ID` and `NodeRA` to point to the next node's location based on the direction determined earlier. The `BRAM_ID` is adjusted by adding the child node type, and `NodeRA` is set to child node information.

Step 7: After iterating through all Basic Blocks and updating the `Prediction array`, output the label corresponding to the maximum value in the result. This label is the predicted class for the input samples.

It is imperative to note that despite the entire process being pipelined, for the prediction of the first sample, a waiting period of M clock cycles is necessitated. Here, M represents the total number of BRAMs utilized in the operation. This reflects the latency inherent to the initialization of the pipelined process.

Chapter 5

Parallel Hardware Accelerator

5.1 Overall Flow

The primary distinction between the parallel hardware accelerator and the pipelined hardware accelerator lies in their design aspects, yet the overarching workflow remains consistent across both types. Consequently, this section will not delve extensively into the general operational process shared by both. Instead, the subsequent discussion will focus predominantly on elucidating the differences between the two, highlighting the unique characteristics of the parallel hardware accelerator.

5.2 Instruction Set

In this section, a notable modification is the reduction of bit width from 72 bits to 64 bits. This adjustment primarily stems from changes in the accelerator's architecture, which obviated the need for the Node Relative Address (NodeRA) segment within the NOInst. Furthermore, the categorization of instruction types has been simplified to two, eliminating the necessity for First Level Instruction (FLInst). The reasons behind these changes will be elaborately discussed in the subsequent sections.

5.2.1 Node Operation Instruction

As depicted in Figure 5.1, the format of the NOInst has changed. Compared to Figure 4.3, the Node Register Address (NodeRA) is no longer utilized, while the rest of the components, including their content and bit width, remain unchanged.

Feature_ID. (Attribute_ID.)	Threshold.	Left Child Node Info.	Right Child Node Info.	Left Child Node Type	Right Child Node Type.
63-56(8bit).	55-40(16bit).	39-32(8bit).	31-24(8bit).	23-12(12bit).	11-0(12bit).

Figure 5.1: New Node Relative Instruction

This change is attributed to the change from the previous approach where each BRAM only stored the internal nodes of one layer of a tree. In contrast, like Figure 5.2, now each tree is allocated one Bram, which stores all of the internal nodes of the tree. This will significantly reduce the demand for resources. Meanwhile, such a modification eliminates the need to ascertain the relative position of each internal node within the layers. In other words, it obviates the need for NodeRA. During the software phase, in the process of generating the NOInst, it suffices to traverse the nodes within the tree sequentially, generating instructions for the internal nodes in order. This allows for the sequential loading of NOInst into the designated Bram during the load phase.

5.2.2 Disappearance of First Layer Instruction

When transitioning from a pipelined architecture to a parallel one, a significant change is that there is no longer a need to specify which Bram stores the root node. As mentioned previously, each Bram now stores the Next Operation Instruction (NOInst) for an entire tree, leaving no doubt that each Bram contains the root node.

5.2.3 Control Instruction

For the CTInst, although currently, only the NOInst type of instruction requires loading into the FPGA's memory, the necessity of utilizing the Instruction Type field to identify the instruction type remains significant. This is because this data segment is transmitted to the Control Unit section. When this segment is entirely zeros or ones, the CU will recognize the instruction as a CTInst. Consequently, the instruction format will same as shown in Figure 4.7.

5.3 Instructions Mapping Method

The change mentioned above will result in a transformation of the Mapping Method, with specific details as follows.

- Internal nodes are stored in sequence according to the order of the tree nodes' indices

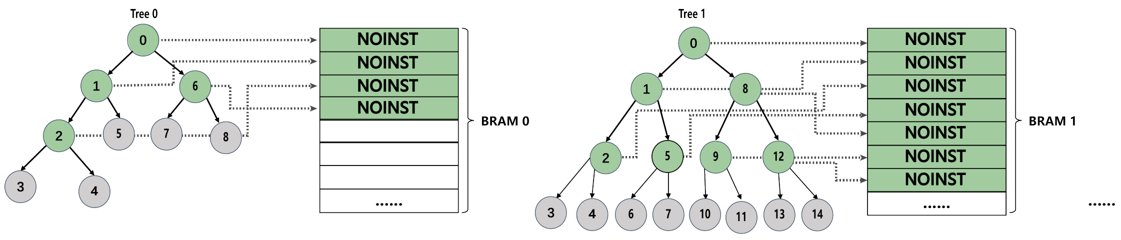


Figure 5.2: Mapping Method for parallel accelerator

- Store the NOInst of all trees from tree 0 to tree n.
- Each BRAM stores NOInsts of one tree.

As shown in Figure 5.2, storing the NOInst of node 0 of tree 0 in Bram 0 at address 0, node 1's NOInst at the position corresponding to address 1, and node 2's NOInst at the position corresponding to address 2. Notably, node 6's NOInst is stored at the position corresponding to address 3. This is different from previous mapping principles of pipelined structure. The same logic is applied to other trees as well.

5.4 Software Design

With minor adjustments to the existing code base, we can generate suitable instructions for parallel hardware accelerators.

5.4.1 Function of NOInst and CLInst Generation

The `Get The Depth` function is no longer necessary. Previously, this function was employed to assign layers to the tree. However, now that we no longer store the Next Operation Instructions (NOInst) by layer, its utility has been rendered obsolete.

CTInst and NOInst Generation Algorithm

The given Algorithm 6 is designed to compute CTInst and NOInst for decision trees within a Random Forests classifier. The algorithm manages memory allocation for these instructions in Brams, considering the hardware constraints like the maximum number of nodes per BRAM and the total number of BRAMs available. The operation of the algorithm is as follows:

Lines 2-4: Initialization. The algorithm begins by initializing the `current_bram_id` to zero and creating two lists: `no_inst` for Node Operation Instructions and

Algorithm 6 Generation of CLInst and NOInst for Each Internal Node

```
1: function GetInstructions($clf$, $max\_nodes\_per\_bram$, $M$)
2:     $current\_bram\_id \leftarrow 0$
3:     $no\_inst \leftarrow array$                              ▷ Operation instructions
4:     $cl\_inst \leftarrow array$                                ▷ Control instructions
5:     $Internal\_node\_index \leftarrow array$                 ▷ Indexes of internal nodes
6:     for $i$, $tree$ in $clf.estimators\_$ do
7:         $current\_bram\_id \leftarrow i$
8:         $ra \leftarrow$ [0] * node number
9:         $Internal\_nodes \leftarrow$ [index if the node has children]
10:        if $Internal\_nodes$ then
11:            Calculate $cl\_addr$ and $cl\_ins$ for current tree, append to $cl\_inst$
12:            Check if $max\_nodes\_per\_bram$ is sufficient for $Internal\_nodes$
13:            for $j$ in $Internal\_nodes$ do
14:                $ra[j] \leftarrow index \ \% \ max\_nodes\_per\_bram$
15:            end for
16:            Append $Internal\_nodes$ to $Internal\_node\_index$
17:        end if
18:        for $k$ in range(0 $to\ node\ bumber$) do
19:            if $Internal\_nodes$ then
20:                Determine node type/info of each child node for $k$
21:                Calculate operation instruction for $k$, append to $no\_inst$
22:            end if
23:        end for
24:    end for
25:    if $current\_bram\_id > M$ then
26:        print("Not enough BRAMs allocated.")
27:    end if
28:    return $no\_inst$, $cl\_inst$, $Internal\_node\_index$
29: end function
```

`cl_inst` for Control Instructions. Additionally, `Internal_node_index` keeps track of internal node indices.

Lines 5-21: Tree Processing Loop. Each tree in the classifier's ensemble is processed individually. This includes initializing a tracking list for node allocation within BRAMs and identifying all internal nodes.

Lines 10-14: Control Instruction Generation. For trees with internal nodes, control instructions are calculated and appended to the `cl_inst` list, with a check for sufficient BRAM depth.

Lines 15-18: Node Address Assignment. Addresses within the BRAM are assigned to each internal node, ensuring proper mapping to physical memory locations.

Lines 22-27: Operation Instruction Calculation. The algorithm calculates Node Operation Instructions for all internal nodes, determining the type and information of each child node.

Lines 28-30: BRAM Utilization Check. It concludes with a verification step to ensure that BRAM usage does not exceed the available amount, issuing a warning if necessary.

Line 31: Output. Finally, lists containing the Node Operation and Control Instructions, along with internal node index, are returned for further processing.

The primary differences between the current approach and the previous algorithm can be summarized in two main points:

Firstly, regarding the address in the CLInst, complex calculations are no longer required. Instead, the address is directly obtained based on the iterative index of the trees within the RFs (Random Forests) model.

Secondly, the calculation of the `Ra` as listed in line 14 of Algorithm 4 is no longer necessary, due to the instruction structure no longer requiring NodeRA.

5.5 Hardware Design

The hardware system is similar to the pipelined hardware accelerator. It consists of a CU, a specific datapath that contains the parallel structure circuit for RFs inference, and the major voting unit.

5.5.1 Datapath

As illustrated in Figure 5.3, the structure has significantly diverged from the previous serial pipeline configuration to a parallel architecture. Each tree's internal nodes are now stored within a single Bram. Each basic block is equipped with a dedicated register array to store the input sample's feature values, facilitating simultaneous access to specific feature values by all Processing Elements (PEs) during the classification process. A PE is assigned to each Bram for the task of classification; after performing a comparison operation with a fetched Next

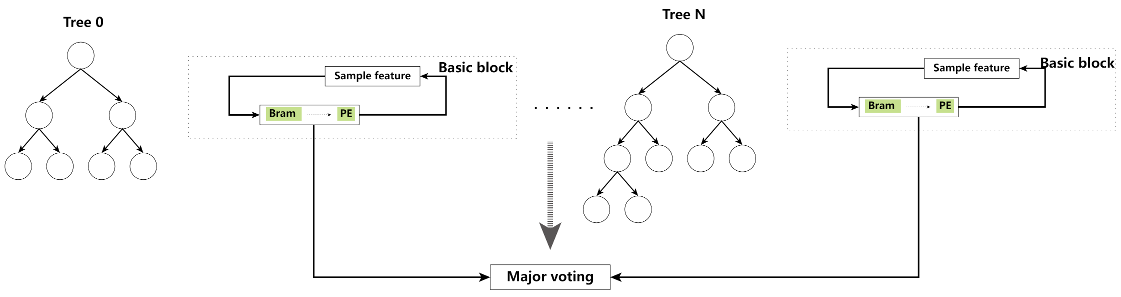


Figure 5.3: Datapath of Parallel Acceletator

Operation Instruction (NOInst) from the Bram, it continues to fetch the next instruction from the same Bram until the tree's classification process is complete and the correct class label is determined.

Once all the trees have completed their classification tasks, the output results from all trees are sent to the majority voting unit to conduct voting and determine the final outcome.

5.5.2 Processing Element

Let's focus on the port differences from the previous approach. In Figure 5.4.

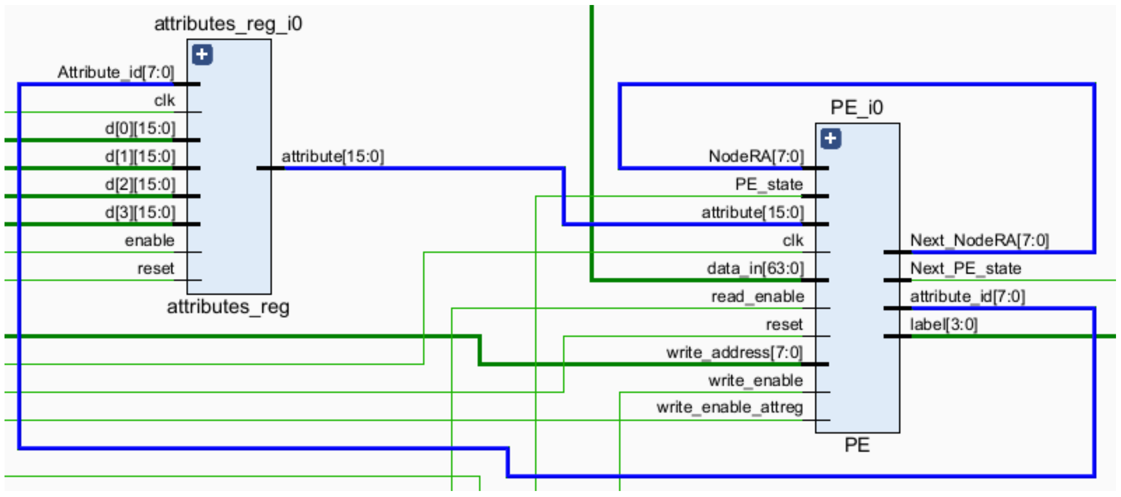


Figure 5.4: New PE Circuit

Firstly, the `Bram_ID` is no longer required because the next instruction to be fetched and then executed is always located within this Bram.

The input value for `NodeRA` no longer originates from other basic blocks but instead comes from the output of the `Next_NodeRA` within the same Processing Element (PE).

Similay, the `is_root_node` flag is not needed anymore.

The `Prediction Array` port is unnecessary. With the discontinuation of the pipeline architecture, it becomes superfluous to store the class label array following each execution of NOInst. Instead, it suffices to output a single class label upon completion of the prediction process.

The ports `attribute`, `data_in`,`read_enable`, `write_enable` and `write_address` are same with previous one.

The significance of the `PE_state` has also undergone modification, now encompassing two distinct states, as described below.

- Value 0: It indicates that the tree has not yet obtained a class label.
- Value 1: It signifies that the tree has completed its prediction and acquired a class label.

Subsequently, the focus is shifted towards the output port. In the output port, the only aspect that merits attention is the replacement of the original `Next_prediction` output with the `label` out port. The rationale for this change has been sufficiently discussed previously and will not be elaborated upon here.

5.5.3 Prediction Process

The prediction process will shift from a serial pipeline to a parallel configuration, with the specific details outlined below:

Step 1: Initialize the input variables `NodeRA` to 0.

Step 2: Simultaneously traverse the basic blocks that have stored the trees.

Step 3: In each BRAM, fetch the NOinst located at the address specified by `NodeRA`.

Step 4: Use `Attribute_ID` to address the samples register array.

Step 5: Within this basic block, at every clock cycle, continue performing comparison operations until its label is outputted.

Step 6: Once all basic blocks have produced their results, the outcomes are outputted to the majority voting unit.

Chapter 6

Experimental Results

6.1 Hardware Implementation

In this section, a comprehensive description of the hardware acceleration implementation process is provided. Initially, hardware design is performed using SystemVerilog within the Vivado software environment, culminating in the generation of a bitstream file. After that, the next step is exporting the so-called hardware platform file. This file is then integrated into the Vitis software framework, where the software driver, designated to operate on an ARM CPU, is developed. Subsequently, the Random Forest algorithm is accelerated in the programmable logic segment of the FPGA. Detailed implementation is delineated in the following sections.

6.1.1 PYNQ Z2

This section of the thesis presents the details of the PYNQ Z2 FPGA board. The PYNQ Z2 board is selected for its flexibility, integration capabilities, and ease of use. The key specifications of the PYNQ Z2 board are as follows:

Processor: Xilinx Zynq-7020 SoC, featuring a dual-core ARM Cortex-A9 processor with a clock speed of up to 650MHz.

Programmable logic:

- 13300 logic slices, each with four 6-input LUTs and 8 flip-flops
- 630 KB of fast block RAM
- 220 DSP slices

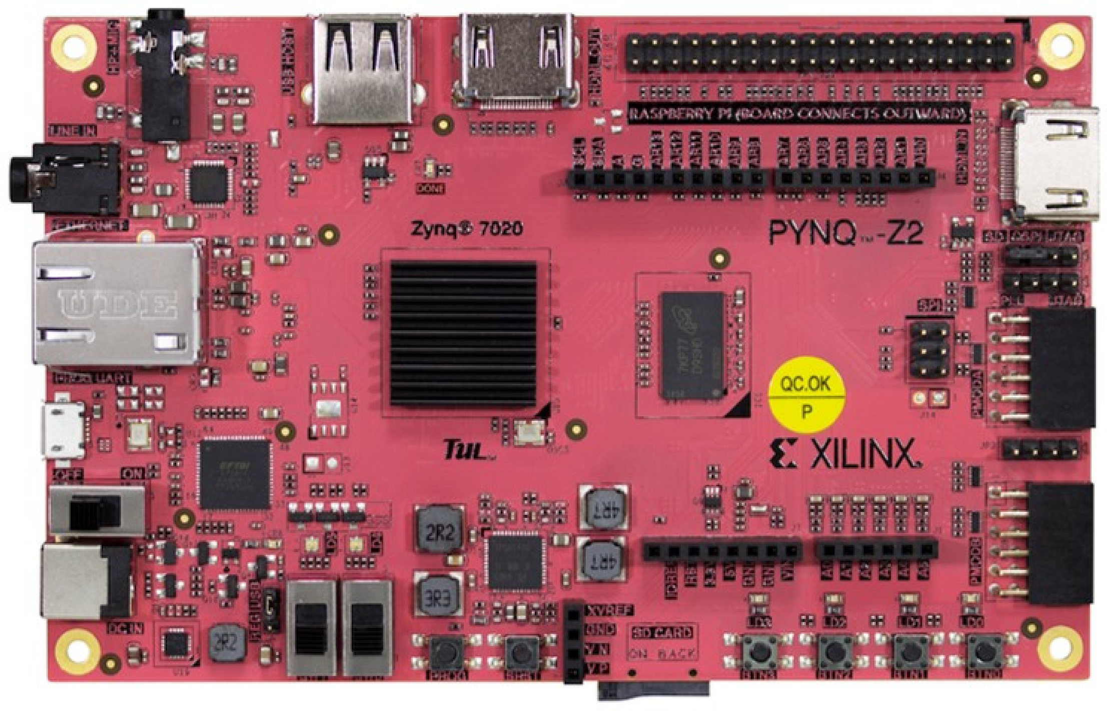


Figure 6.1: PYNQ Z2 [23]

Memory: 512MB DDR3 RAM with a data rate of up to 1050Mbps, complemented by a MicroSD slot for external storage.

Power: Powered from USB or 7V-15V external power source.

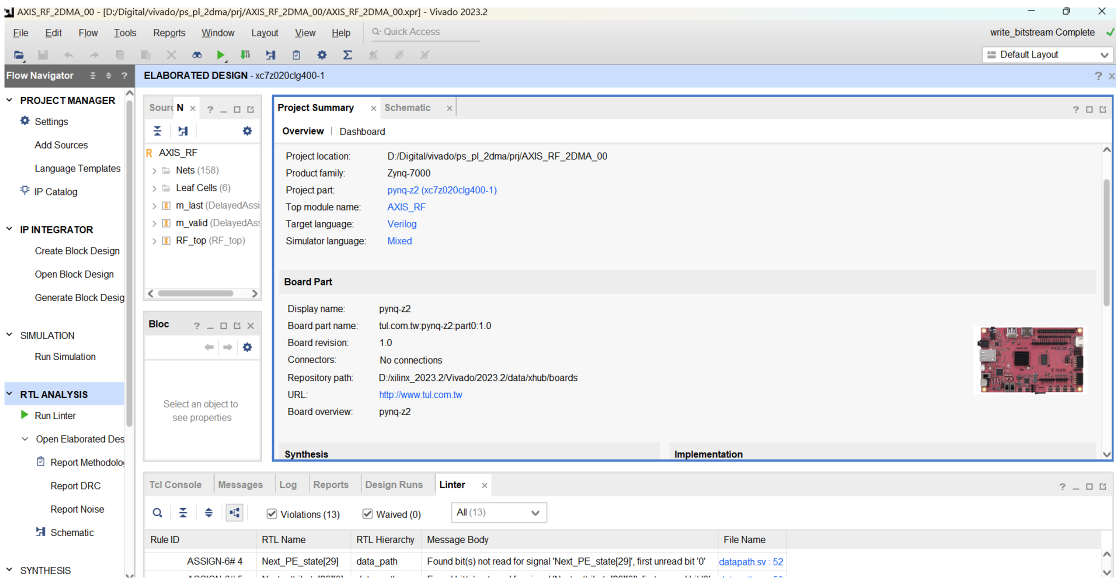
Programmable Mode JTAG, Quad-SPI flash, and MicroSD card

Connectivity: Includes Gigabit Ethernet for network connections and USB ports for programming and UART communication.

In the implementation, an external power source will be used for the power supply. Communication will be facilitated through UART, and the programming mode will be set to JTAG.

6.1.2 Design Process Using Vivado

Figure 6.2 displays the overall interface of Vivado. The design flow in Vivado encompasses several key stages, each critical for optimizing the performance and


AXIS_RF_2DMA_00 - [D:/Digital/vivado/ps_pl_2dma/prj/AXIS_RF_2DMA_00/AXIS_RF_2DMA_00.xpr] - Vivado 2023.2

write_bitstream Complete

ELABORATED DESIGN - xc7z020clg400-1

Project location: D:/Digital/vivado/ps_pl_2dma/prj/AXIS_RF_2DMA_00
Product family: Zynq-7000
Project part: pynq-z2 (xc7z020clg400-1)
Top module name: AXIS_RF
Target language: Verilog
Simulator language: Mixed

Board Part

Display name: pynq-z2
Board part name: tul.com.tw:pynq-z2:part0:1.0
Board revision: 1.0
Connectors: No connections
Repository path: D:/xilinx_2023.2/Vivado/2023.2/data/xhub/boards
URL: http://www.tul.com.tw
Board overview: pynq-z2

Synthesis
Implementation

Violations (13) Waived (0) All (13)

Rule ID	RTL Name	RTL Hierarchy	Message Body	File Name
ASSIGN-6# 4	Next_PE_state[29]	data_path	Found bit(s) not read for signal 'Next_PE_state[29]', first unread bit '0'	datapath.sv : 52

Figure 6.2: The Vivado Tool

resource utilization of the FPGA. These stages are:

Project Creation and Environment Setup: Launching Vivado and creating a new project specific to the PYNQ Z2 board by selecting the appropriate board definition files. Then Configure the Vivado environment to match the project requirements, setting up the design language (Verilog).

Design Entry and Simulation Writing the Hardware Description Language (HDL) code (SystemVerilog) that defines the hardware accelerator's functionality. Running simulations to validate the design under various conditions and inputs, ensuring that correctness and performance targets are met. Utilizing Vivado's Block Design for visually constructing the design using IP (Intellectual Property) cores and custom modules. In this section, two IPs provided by Xilinx are utilized. The first is the ZYNQ7 Processing System, which is employed to configure various parameters for both the PS (Processing System) and PL (Programmable Logic) sides. The second is the AXI Direct Memory Access, which facilitates communication between the PS and PL sides. Figure 6.3 represents the final Block Design of the Pipelined Hardware Accelerator, and Figure 6.4 depicts the final Block Design of the Parallel Hardware Accelerator.

Synthesis: Translating the HDL code into a gate-level representation that maps onto the FPGA's resources, optimizing for speed, area, and power consumption.

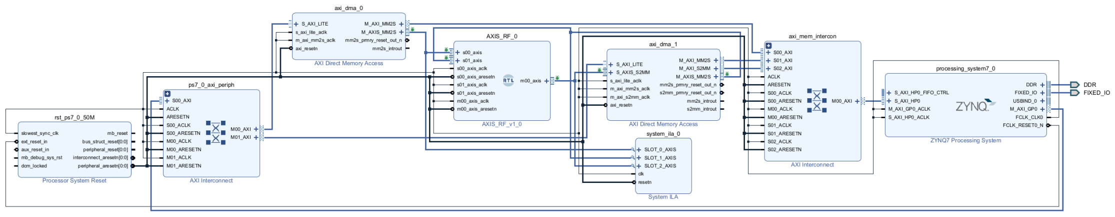


Figure 6.3: Block Design of Pipelined Hardware Accelerator

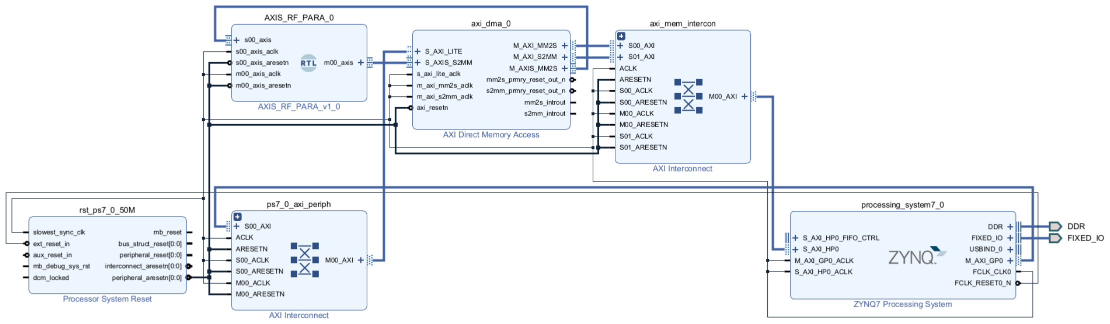


Figure 6.4: Block Design of Parallel Hardware Accelerator

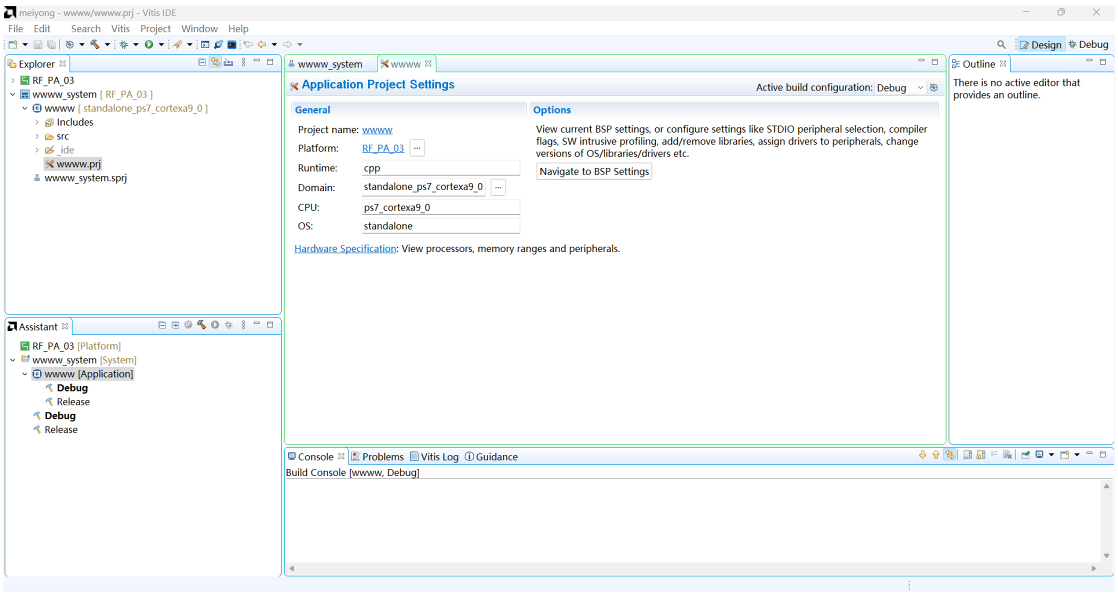
Bitstream Creation: Generating the bitstream file, which is the binary file that will be loaded onto the FPGA to configure its logic blocks and interconnects according to the design.

6.1.3 Driver Development Using Vitis

This section details the development of custom drivers for the hardware accelerator, utilizing the Vitis software platform. The drivers facilitate communication between the hardware accelerator and the application software, ensuring data transfer and command execution. The interface of Vitis is shown in Figure 6.5.

Initial Setup: Launching Vitis, select a workspace and create an application project using the XSA file export from Vivado.

Driver Development Process: Writing a DMA-based data transfer and accelerator execution driver.



Application Project Settings

Active build configuration: Debug

General

Project name: wwww
Platform: RF_PA_03
Runtime: cpp
Domain: standalone_ps7_cortexa9_0
CPU: ps7_cortexa9_0
OS: standalone

Hardware Specification: View processors, memory ranges and peripherals.

Options

View current BSP settings, or configure settings like STDIO peripheral selection, compiler flags, SW intrusive profiling, add/remove libraries, assign drivers to peripherals, change versions of OS/libraries/drivers etc.

Navigate to BSP Settings

```
Build Console [wwww, Debug]
```

Figure 6.5: The Vitis Tool

Run: Running the driver, this step performs the execution of inference based on the hardware accelerator.

6.2 Results Evaluation

This section presents a comprehensive evaluation of the pipelined hardware accelerator and the parallel hardware accelerators implemented on the PYNQ Z2 FPGA board. The objective is to assess their performance in terms of speed up compared with CPU, resource utilization, power consumption, and flexibility in various operational scenarios.

6.2.1 Speed up

In this part, the inference time used is compared between the hardware accelerator and CPU of PYNQ-Z2. The C codes of RFs that run in the CPU are obtained from the **Eden** library [24].

This library use methodology put forth by Daghero et al.[25] hinges on the concept of early stopping, which is a set of policies determining when the execution of decision trees within an ensemble can be halted without significant loss of accuracy. Notably, these policies adaptively halt the inference process, conserving energy by avoiding redundant computations for easy-to-predict inputs. The researchers meticulously designed data structures and memory allocation strategies that are optimized for the constrained computing environment of IoT devices.

The authors underscore the practicality of their approach by conducting extensive benchmarking across three IoT-relevant datasets, encompassing applications such as hand gesture recognition from electromyography signals, hard drive failure prediction, and human activity recognition using accelerometer data. These applications reflect diverse challenges in the IoT domain, including varying input signal types, data dimensionalities, and class imbalances.

The toolchain developed auto-converts a Python ensemble into optimized C code, efficiently mapping the ensemble model to the device's multicore architecture and memory hierarchy. This optimized C code achieved speed-ups ranging from 3.15× to 7.92× for tree execution from 1 core to 8 cores for the SOC GAP8 platform.

In this thesis, the acceleration ratio is calculated by comparing the inference time of the hardware accelerator against that of a dual-core ARM Cortex-A9 processor.

Speed up of Pipelined Hardware Accelerators

Table 6.1: Speed-up Performance of Pipelined Acceletor

Test Case	SD_C(ns)	T_C(ns)	N_bram	T_HD(ns)	Speed-up
N_tree2_Depth2	61.83	550	4/30	45	12.22x
N_tree2_Depth3	95.96	599	6/30	45	13.31x
N_tree2_Depth4	114.87	621	8/30	45	13.80x
N_tree2_Depth5	100	607	9/30	45	13.49x
N_tree3_Depth2	70.69	796	6/30	45	17.68x
N_tree3_Depth3	125.95	874	9/30	45	19.42x
N_tree3_Depth4	132.15	881	12/30	45	19.58x
N_tree3_Depth5	151.49	898	14/30	45	19.96x
N_tree4_Depth2	73.48	1063	8/30	45	23.62x
N_tree4_Depth3	135.63	1156	12/30	45	25.69x
N_tree4_Depth4	126.67	1176	16/30	45	26.13x
N_tree4_Depth5	175.38	1227	19/30	45	27.27x
N_tree5_Depth2	112.77	1291	10/30	45	28.69x
N_tree5_Depth3	190.13	1407	15/30	45	31.27x
N_tree5_Depth4	290.51	1561	20/30	45	34.69x
N_tree5_Depth5	247.87	1524	23/30	45	33.87x

In Table 6.1, different configurations of RFs models are tested, denoting the number of trees and their maximum depth, with experiments conducted using 240 input samples. `N_tree2_Depth2` indicating a model with two trees, each with a max

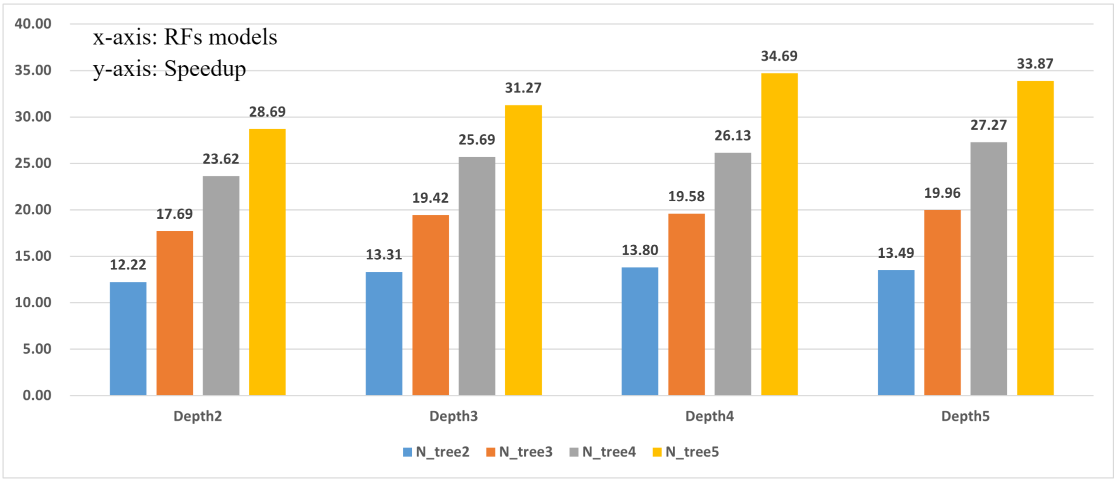


Figure 6.6: Speed Up

depth of two. `T_C` represents the average time to classify a sample using a C language implementation of the RF running on the Cortex-A9, with this code derived from the Eden library. `N_bram` signifies the number of BRAMs needed for a specific test case. `T_HD` denotes the classification time using a pipelined hardware accelerator. `Speed-up` is the ratio of prediction time needed in the CPU and time needed in the accelerator. `SD_C` is the standard deviation of the C code's prediction time.

As depicted in Figure 6.6, a marked escalation in speed-up correlates with the augmentation in both the number of trees and the increment of their depth within the RFs models. It is observed that, with a fixed number of trees, an enhancement in tree depth yields a marginal improvement in speed-up. Conversely, maintaining a constant tree depth while amplifying the number of trees results in a substantial increase in speed-up. This trend is attributable to the fact that augmenting the number of trees considerably elevates the complexity of the RF model to a greater extent than an increase in tree depth does. Consequently, the larger and more intricate the RF model becomes, the more pronounced the acceleration ratio realized by the hardware accelerator.

Table 6.2: Prediction Time for Different Bram Configuration

	N_Bram_used=10	**N_Bram_used=30**
T_HC (ns)	45	45

In this test, setting the FPGA's PL part to utilize 30 BRAMs resulted in a maximum acceleration ratio of approximately 34.69x. However, this is not the limit of acceleration. Increasing the complexity of the RFs model, such as the number of trees, will further enhance the acceleration ratio. This is because increasing complexity extends the CPU's execution time, whereas the hardware accelerator's time remains unchanged due to its pipelined structure, depending solely on the PL's clock cycles as shown in Table 6.2. The only drawback is the need for more BRAMs than the 30 initially configured, necessitating FPGA reconfiguration.

It is pertinent to note that the test clock cycle was set at 50 MHz, which translates to a theoretical throughput of 50 Million samples per second (MS/s). However, the actual observed acceleration latency was approximately 45 ns, and the corresponding throughout was 22.22 MS/s. The deviation from the theoretical time can be attributed to the fact that the measured inference time of the accelerator encompasses not only the intrinsic processing time of the accelerator itself but also the time taken to transfer the input feature data from the PS memory to the accelerator in the PL part via the AXI stream DMA, as well as the time required to relay the output results back from the PL part to the PS. In this context, the transfer time of the DMA constitutes a significant portion of the time expenditure.

Table 6.3: Inference time for different quantities of samples utilizing DMA

30 samples	**60 samples**	**120 samples**	**240 samples**
185ns	105ns	65ns	45ns

Table 6.3 elucidates the impact on the results, reflecting the accelerator inference time per sample when transmitting different quantities of samples utilizing DMA. The pipelined architecture of the accelerator ensures consistent results across various RFs model test cases, thus the table does not distinguish between different models. It is evident from the table that the greater the number of transmitted samples, the shorter the inference time per sample, indicating a more effective acceleration. In fact, the intrinsic inference time for the accelerator on the PL side is a constant 20ns per sample, regardless of the number of samples processed. The additional time expenditure is attributable to DMA transfer overheads. While increasing the amount of data for transmission does incur more time, this increase is negligible for the DMA itself, given the substantial inherent transfer time of the DMA IP. As such, it is a wiser choice to transmit a larger batch of samples, as this fixed time cost is then amortized over each sample. The results presented in Table 6.1 and the following speed-up of the parallel hardware accelerators are based on tests of 240 sample transmissions.

Speed up of Parallel Hardware Accelerators

Table 6.4: Speed-up Performance of Parallel Acceletor

Test Case	T_C(ns)	N_bram	T_HD0(ns)	Speed-up	T_HD1(ns)	Drawback
N_tree2_Depth2	550	2/5	70	7.86x	45	1.56x
N_tree2_Depth3	599	2/5	77	7.78x	45	1.71x
N_tree2_Depth4	621	2/5	85	7.31x	45	1.89x
N_tree2_Depth5	607	2/5	86	7.06x	45	1.91x
N_tree3_Depth2	796	3/5	70	11.37x	45	1.56x
N_tree3_Depth3	874	3/5	82	10.66x	45	1.82x
N_tree3_Depth4	881	3/5	86	10.24x	45	1.91x
N_tree3_Depth5	898	3/5	87	10.32x	45	1.93x
N_tree4_Depth2	1063	4/5	77	13.81x	45	1.71x
N_tree4_Depth3	1156	4/5	89	12.99x	45	1.98x
N_tree4_Depth4	1176	4/5	93	12.65x	45	2.07x
N_tree4_Depth5	1227	4/5	109	11.26x	45	2.42x
N_tree5_Depth2	1291	5/5	76	16.99x	45	1.69x
N_tree5_Depth3	1407	5/5	91	15.46x	45	2.02x
N_tree5_Depth4	1561	5/5	101	15.46x	45	2.24x
N_tree5_Depth5	1524	5/5	109	13.98x	45	2.42x

In Table 6.4, the term `Test Case` and the notation `T_C` retain their meanings consistent with prior usage. `N_bram` denotes the number of BRAMs required for different test cases within the parallel accelerator, illustrating the hardware resource allocation. `T_HD0` represents the inference time of the parallel accelerator, while `Speed-up` quantifies the acceleration ratio of the parallel accelerator in comparison to the CPU for random tree inference tasks. `T_HD1` indicates the inference time under identical configurations for the pipelined accelerator. The `drawback` metric compares the latency of the parallel accelerator with that of the pipelined accelerator, thereby illustrating the extent to which the parallel accelerator's inference time is slower compared to the pipelined accelerator.

Table 6.4 displays the speed-up results of the parallel pipelined accelerator compared to the CPU. Under identical settings with the pipelined accelerator, a maximum speed-up of 16.99x was achieved. This occurs in the scenario where the RFs consist of five trees with a tree depth of 2. At this juncture, the average inference delay for a single sample is 76 ns, and the throughput reaches its peak at 13.16 MS/s. Theoretically, for parallel accelerators, the inference time primarily depends on the tree depth and is less affected by the number of trees. This is because the inference process for each tree is independent and non-interfering, making the inference time largely dependent on tree depth. However, as observed in the table, even with the same tree depth, cases with fewer trees tend to have shorter inference times. For instance, the delay for test case N_tree5_depth5 is 109ns, while that for N_tree2_depth5 is 86ns. This discrepancy can be attributed to the fact that, one is that the models used in this thesis are optimized through early stopping strategies via the Eden library; and the other is that for a given input sample, it has a higher likelihood of reaching a deeper depth in models with a larger number of trees.

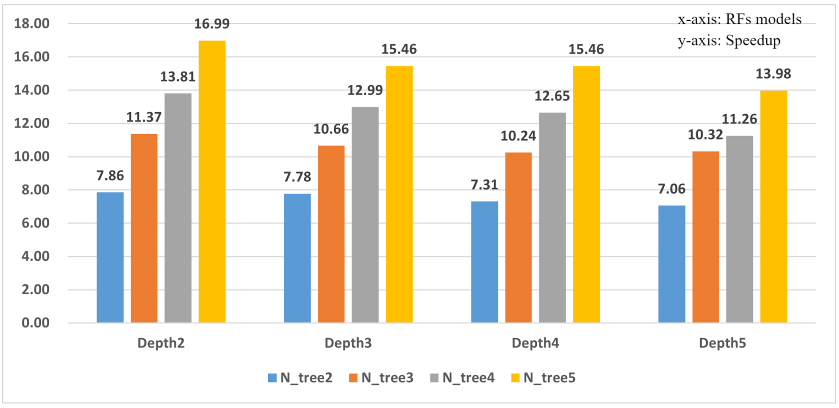


Figure 6.7: Speed-up for Varying Numbers of Trees at a Fixed Tree Depth

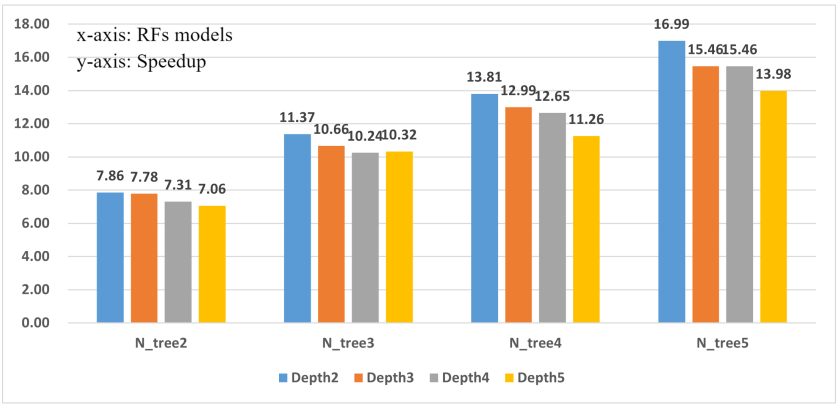


Figure 6.8: Speed-up for Varying Tree Depth at Fixed Numbers of Trees

In Figure 6.7, it is observed that with a constant tree depth, the acceleration ratio significantly increases as the number of trees grows. This phenomenon occurs because, under conditions where the tree depth remains unchanged while the number of trees increases, the inference time on the CPU surges drastically, whereas the parallel accelerator incurs only a marginally higher time cost. Furthermore, due to the parallel accelerator's inference time being influenced by tree depth, Figure 6.8 reveals that keeping the number of trees constant while increasing the depth results in a decreased acceleration ratio. These findings validate the design of the parallel accelerator, demonstrating its suitability for applications with a large number of trees but fixed tree depths.

6.2.2 Resource Utilization

Resource Utilization of Pipelined Hardware Accelerators

Table 6.5: Resource Utilization of the Standalone Pipelined Accelerator IP

Resource	**Utilization**	**Available**	**Utilization %**
LUT	4299	53200	8.08
LUTRAM	2	17400	0.01
FF	5502	106400	5.17
BRAM	30	140	21.43

Table 6.6: Resource Utilization of the Pipelined Complete System

Resource	**Utilization**	**Available**	**Utilization %**
LUT	8515	53200	16.01
LUTRAM	315	17400	1.81
FF	11028	106400	10.36
BRAM	35	140	25.00

The final resource utilization is detailed in Table 6.5, showing that the customized accelerator module used 30 BRAMs. Incorporating other IPs and adding units to support AXI stream transmission resulted in the use of a total of 35 BRAMs. Table 6.5 presents the resource utilization of the Standalone Pipelined Accelerator, revealing that 30 BRAMs are employed. Table 6.6 shows the final resource utilization after integrating the necessary DMA IP and other connectivity IPs for communication with the ZYNQ 7000 processor. Comparing the data in these tables, it becomes evident that these peripheral circuits account for a significant portion of the final resource usage, consuming nearly half of the LUT and FF resources, 14.39% of the BRAM resources, and 99% LUTRAM resources.

Resource Utilization of Parallel Hardware Accelerators

Tables 6.7 and 6.8 respectively illustrate the resource utilization of the parallel accelerator on its own and after the inclusion of circuits for connectivity with the processor. Given that the resource consumption by the accelerator itself is significantly lower compared to a pipelined architecture supporting the same test cases, with the values being considerably minimal, the relative proportion

Table 6.7: Resource Utilization of the Standalone Parallel Accelerator

Resource	**Utilization**	**Available**	**Utilization %**
LUT	667	53200	1.25
LUTRAM	0	17400	0
FF	567	106400	0.53
BRAM	5	140	3.57

Table 6.8: Parallel Complete System Resource

Resource	**Utilization**	**Available**	**Utilization %**
LUT	3021	53200	5.68
LUTRAM	190	17400	1.09
FF	3773	106400	3.55
BRAM	7.5	140	5.36

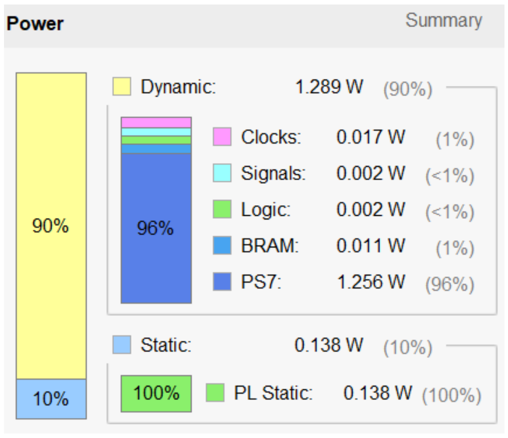
of resources utilized by the circuits external to the accelerator increases within the overall resource usage. Within the configuration involving DMA and other peripheral connectivity circuits, the external circuit represents 77.77% LUT resource usage of the total resources utilized. LUTRAM utilization stands at 100%, Flip-Flops (FF) consumption contributes to 84.97% of all FF consumption, and BRAM usage constitutes 35.71% of the overall resource used.

6.2.3 Power Consumption

Figure 6.9 and Figure 6.10 respectively display the power consumption of the pipelined and parallel accelerator. This thesis focuses on the power consumption of the accelerators, thus the power consumption originating from the PS side, the ZYNQ 7000 processor, is not considered within this study. The reason why the measured power consumption includes the PS side is due to the implementation requirement of incorporating the PS IP to address the compatibility issues of IO ports caused by the wide bit-width inputs and outputs in this design. Ultimately, the power consumption of the pipelined accelerator is 0.171 watts, while the power consumption of the parallel accelerator is slightly less than that of the pipelined accelerator, at 0.149 watts, attributable to its utilization of fewer resources.

6.2.4 Comparative Analysis of Pipelined and Parallel Accelerators

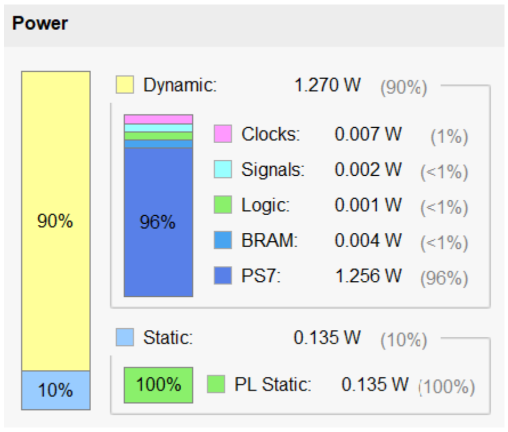
Within Table 6.9, *Ratio* refers to the comparative value of each metric for the pipeline accelerator relative to those of the parallel accelerator. It is immediately



Power Summary

Dynamic:	1.289 W	(90%)	
	Clocks:	0.017 W	(1%)
	Signals:	0.002 W	(<1%)
	Logic:	0.002 W	(<1%)
	BRAM:	0.011 W	(1%)
	PS7:	1.256 W	(96%)
Static:	0.138 W	(10%)	
	PL Static:	0.138 W	(100%)

Figure 6.9: Power Consumption of the Pipelined Hardware Accelerator



Power

Dynamic:	1.270 W	(90%)	
	Clocks:	0.007 W	(1%)
	Signals:	0.002 W	(<1%)
	Logic:	0.001 W	(<1%)
	BRAM:	0.004 W	(<1%)
	PS7:	1.256 W	(96%)
Static:	0.135 W	(10%)	
	PL Static:	0.135 W	(100%)

Figure 6.10: Power Consumption of Parallel Hardware Accelerators

apparent that, based on all test cases previously mentioned, the pipelined accelerator is capable of achieving a maximum speedup that is twice that of the parallel accelerator. At its optimal speed-up, the parallel accelerator exhibits a latency of 76 ns and a throughput of 13.16 MS/s. However, it is noteworthy, as Table 6.4 reveals, that the throughput at the maximum speedup does not correspond to its optimal throughput of 14.29 MS/s with corresponding latency 70 ns, despite identical tree depths in both models. This discrepancy arises because the acceleration ratio is influenced by both the inference time of the CPU and that of the accelerator, with the latter being affected not only by tree depth but also by the number of trees. Meanwhile, the pipelined accelerator's latency remains constant at 45ns, a figure previously mentioned and consistent across all test cases.

Although the parallel accelerator exhibits a slightly lower speedup ratio compared

Table 6.9: Pipeline VS. Parallel

Metric	**Pipelined**	**Parallel**	**Ratio**
Throughput at Max Speed-up			
Maximum Speed-up	34.69x	16.99x	2.04
Throughput (MS/s)	22.22	13.16	1.69
Resource Utilization of Standalone Accelerator			
LUT Utilization (%)	8.08	1.25	6.46
FF Utilization (%)	5.17	0.53	9.75
BRAM Utilization (%)	21.43	3.57	6.00
Resource Utilization with Additional Circuits			
LUT Utilization	16.01	5.68	2.82
FF Utilization	10.36	3.55	2.92
BRAM Utilization	25	5.36	4.66
Power Consumption (watts)	1.427	1.405	1.02
PS power Consumption (watts)	1.256	1.256	1
PL Power Consumption (watts)	0.171	0.149	1.15

to the pipelined accelerator, it vastly surpasses the latter in terms of resource efficiency. Without the addition of external circuits, the pipelined accelerator's utilization of LUT resources is 6.46 times that of the parallel accelerator, its FF resource consumption is 9.75 times higher, and the BRAM utilization is sixfold. After incorporating peripheral connectivity circuits, these ratios decrease due to the substantial resource consumption by the external circuits. Nonetheless, the parallel accelerator maintains superior resource utilization compared to the pipelined accelerator.

The pipelined accelerator achieves high flexibility and a superior speedup at the cost of considerable resource utilization. When ample memory resources such as BRAM are available, it is well-suited for scenarios with large data input volumes and extensive numbers of deep trees, albeit with significantly higher power consumption compared to parallel accelerators. Although not evident in current tests, the disparity in power efficiency between the two accelerators is expected to widen with increasing model complexity. Additionally, the latency required to fully utilize the pipeline increases with more BRAM usage. Under fixed resource conditions, the pipelined accelerator is advantageous for scenarios with very deep trees and a set number of trees.

Conversely, the parallel accelerator makes certain trade-offs in speedup to achieve reduced resource usage and improved parallelism, resulting in better energy efficiency. It maintains excellent flexibility, surpassing the pipelined accelerator, especially when tree depth is fixed and BRAM allocation is similar. For instance, in previous tests, only 5 BRAMs were necessary to meet requirements. While both pipelined and parallel accelerators are compatible with numerous random tree models, allocating the same 30 BRAMs to the pipelined accelerator allows testing additional cases, like increasing the number of trees—without reconfiguration, as long as the tree depth does not exceed the maximum allowed. This level of adaptability is unattainable for the pipelined accelerator. Therefore, the parallel accelerator is particularly well-suited for applications with a vast number of trees at a fixed depth.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

This thesis presented a detailed exploration and implementation of hardware accelerators for Random Forests (RFs) utilizing a RISC-based architecture. The work began with an analysis of the limitations of traditional software implementations of Random Forests, highlighting the need for accelerated processing to meet the demands of big data applications. Then delve into the current landscape of hardware accelerators, pinpointing a notable gap in the flexibility of existing RFs hardware accelerators. This thesis has then addressed the critical shortfall, and introduced an innovative accelerator architecture designed to enhance the adaptability of RFs model deployment on hardware platforms. The proposed design stands out by enabling dynamic adjustments to model parameters without requiring extensive hardware reconfigurations, significantly reducing the time and resource costs traditionally associated with hardware adaptations to evolving RFs models.

Key contributions of this thesis include the following.

- The development of a pipelined and a parallel hardware accelerator architecture that optimizes the RFs inference acceleration for high flexibility.
- A comparative analysis of pipelined versus parallel approaches, reveals the trade-offs between resource utilization, power consumption, classification speed, and flexibility.
- Implementation of the proposed hardware accelerators on a PYNQ Z2 FPGA board, showcasing the practical feasibility and scalability of the design.

Experimental results validated the hypothesis that hardware acceleration of Random Forests could achieve substantial speedups over traditional CPU-based implementations. Specifically, with great flexibility, both pipelined and parallel hardware accelerators showed remarkable efficiency in handling multiple trees and deeper depths, proving their effectiveness for complex models.

7.2 Future Works

While this research marks a significant step toward efficient hardware-accelerated Random Forests, several avenues for future work have been identified:

`Cross-Platform Adaptability:` Exploring the adaptability of the proposed architecture across different hardware platforms, including emerging technologies in ASICs and SoCs, to validate and potentially enhance its flexibility and performance benefits across a broader spectrum of computational environments.

`Energy Efficiency Considerations:` While focusing on flexibility, it is imperative to concurrently evaluate and improve the energy efficiency of the proposed architecture to ensure that increased adaptability does not come at the expense of higher power consumption.

`Optimization and Scalability:` Further optimization of the proposed accelerator architecture for scalability, ensuring that it can efficiently handle increasingly complex RFs models, as well as other decision tree algorithms.

Bibliography

[1] Leo Breiman. «Random Forests». In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324 (cit. on pp. 1, 4).

[2] Gérard Biau and Erwan Scornet. «A random forest guided tour». In: *TEST* 25.2 (2016), pp. 197–227. ISSN: 1863-8260. DOI: 10.1007/s11749-016-0481-7. URL: https://doi.org/10.1007/s11749-016-0481-7 (cit. on p. 1).

[3] Mariana Belgiu and Lucian Drăguţ. «Random forest in remote sensing: A review of applications and future directions». In: *ISPRS Journal of Photogrammetry and Remote Sensing* 114 (2016), pp. 24–31. ISSN: 0924-2716. DOI: https://doi.org/10.1016/j.isprsjprs.2016.01.011. URL: https://www.sciencedirect.com/science/article/pii/S0924271616000265 (cit. on p. 1).

[4] Sofia Benbelkacem and Baghdad Atmani. «Random Forests for Diabetes Diagnosis». In: *2019 International Conference on Computer and Information Sciences (ICCIS)*. 2019, pp. 1–4. DOI: 10.1109/ICCISci.2019.8716405 (cit. on p. 1).

[5] Matthias Schonlau and Rosie Yuyan Zou. «The random forest algorithm for statistical learning». In: *The Stata Journal* 20.1 (2020), pp. 3–29. DOI: 10.1177/1536867X20909688. URL: https://doi.org/10.1177/1536867X20909688 (cit. on p. 1).

[6] Hristos Tyralis, Georgia Papacharalampous, and Andreas Langousis. «A Brief Review of Random Forests for Water Scientists and Practitioners and Their Recent History in Water Resources». In: *Water* 11.5 (2019). ISSN: 2073-4441. DOI: 10.3390/w11050910. URL: https://www.mdpi.com/2073-4441/11/5/910 (cit. on p. 1).

[7] Li Du and Yuan Du. «Hardware Accelerator Design for Machine Learning». In: *Machine Learning*. Ed. by Hamed Farhadi. Rijeka: IntechOpen, 2017. Chap. 1. DOI: 10.5772/intechopen.72845. URL: https://doi.org/10.5772/intechopen.72845 (cit. on p. 2).

[8] Zhi-Hua Zhou. «Ensemble Learning». In: *Encyclopedia of Biometrics*. Ed. by Stan Z. Li and Anil Jain. Boston, MA: Springer US, 2009, pp. 270–273. ISBN: 978-0-387-73003-5. DOI: 10.1007/978-0-387-73003-5_293. URL: https://doi.org/10.1007/978-0-387-73003-5_293 (cit. on p. 4).

[9] Gavin Brown. «Ensemble Learning». In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 312–320. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_252. URL: https://doi.org/10.1007/978-0-387-30164-8_252 (cit. on p. 4).

[10] Gautam Kunapuli. *Ensemble Methods for Machine Learning*. Simon and Schuster, 2023 (cit. on p. 4).

[11] Xilinx. *What is an FPGA?* https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. Accessed: February 21, 2024 (cit. on p. 5).

[12] Eduardo Sanchez. «Field programmable gate array (FPGA) circuits». In: *Towards Evolvable Hardware*. Ed. by Eduardo Sanchez and Marco Tomassini. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–18. ISBN: 978-3-540-49947-3 (cit. on p. 6).

[13] Xilinx. *Programming an FPGA: An Introduction to How It Works*. https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html. Accessed: 2024-03-27 (cit. on p. 6).

[14] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. «Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures». In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 1–12. DOI: 10.1109/HPCA.2013.6522302 (cit. on p. 6).

[15] Michael J. Flynn. «Computer Architecture: Pipelined and Parallel Processor Design». In: 1995. URL: https://api.semanticscholar.org/CorpusID:60782999 (cit. on p. 6).

[16] Bernard Goossens. «The RISC-V Architecture». In: *Guide to Computer Processor Architecture: A RISC-V Approach, with High-Level Synthesis*. Cham: Springer International Publishing, 2023, pp. 105–122. ISBN: 978-3-031-18023-1. DOI: 10.1007/978-3-031-18023-1_4. URL: https://doi.org/10.1007/978-3-031-18023-1_4 (cit. on p. 6).

[17] Milan Shah, Reece Neff, Hancheng Wu, Marco Minutoli, Antonino Tumeo, and Michela Becchi. «Accelerating Random Forest Classification on GPU and FPGA». In: *Proceedings of the 51st International Conference on Parallel Processing*. ICPP '22. Bordeaux, France: Association for Computing Machinery, 2023. ISBN: 9781450397339. DOI: 10.1145/3545008.3545067. URL: https://doi.org/10.1145/3545008.3545067 (cit. on p. 8).

[18] Mingyu Zhu, Jiapeng Luo, Wendong Mao, and Zhongfeng Wang. «An Efficient FPGA-based Accelerator for Deep Forest». In: *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2022, pp. 3334–3338. DOI: 10.1109/ISCAS48785.2022.9937620 (cit. on p. 9).

[19] Xiang Lin, R.D. Shawn Blanton, and Donald E. Thomas. «Random Forest Architectures on FPGA for Multiple Applications». In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. GLSVLSI '17. Banff, Alberta, Canada: Association for Computing Machinery, 2017, pp. 415–418. ISBN: 9781450349727. DOI: 10.1145/3060403.3060416. URL: https://doi.org/10.1145/3060403.3060416 (cit. on p. 9).

[20] Adrián Alcolea and Javier Resano. «FPGA Accelerator for Gradient Boosting Decision Trees». In: *Electronics* 10.3 (2021). ISSN: 2079-9292. DOI: 10.3390/electronics10030314. URL: https://www.mdpi.com/2079-9292/10/3/314 (cit. on p. 9).

[21] Shuang Zhao, Shuhui Chen, Hui Yang, Fei Wang, and Ziling Wei. «RF-RISA: A novel flexible random forest accelerator based on FPGA». In: *Journal of Parallel and Distributed Computing* 157 (2021), pp. 220–232. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2021.07.001. URL: https://www.sciencedirect.com/science/article/pii/S0743731521001477 (cit. on pp. 9, 11).

[22] F. Pedregosa et al. «Scikit-learn: Machine Learning in Python». In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 20).

[23] PYNQ Project. *PYNQ - Python Productivity for Zynq*. https://www.pynq.io/. Accessed: 2024-03-27. 2023 (cit. on p. 40).

[24] eml-eda. *EDEN: EDA Environment*. https://github.com/eml-eda/eden. Accessed: date. 2023 (cit. on p. 43).

[25] Francesco Daghero, Alessio Burrello, Enrico Macii, Paolo Montuschi, Massimo Poncino, and Daniele Jahier Pagliari. «Dynamic Decision Tree Ensembles for Energy-Efficient Inference on IoT Edge Nodes». In: *IEEE Internet of Things Journal* (2023) (cit. on p. 43).