VLSI IMPLEMENTATION OF TRAINING ACCELERATORS FOR DECISION TREE ALGORITHM

RITUPARNA CHOUDHURY

VLSI Implementation of Training Accelerators for Decision Tree Algorithm

A

Thesis submitted

for the award of the degree of

DOCTOR OF PHILOSOPHY

By

RITUPARNA CHOUDHURY

DEPARTMENT OF ELECTRONICS AND ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

GUWAHATI - 781 039, ASSAM, INDIA

MARCH 2023

Certificate

This is to certify that the thesis entitled "**VLSI Implementation of Training Accelerators for Decision Tree Algorithm**", submitted by **RITUPARNA CHOUDHURY** (176102101), a research scholar in the *Department of Electronics and Electrical Engineering, Indian Institute of Technology Guwahati*, for the award of the degree of **Doctor of Philosophy**, is a record of an original research work carried out by her under my supervision and guidance. The thesis has fulfilled all requirements as per the regulations of the institute and in my opinion, has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

Prof. S. R. Ahamed
Professor
Dept. of Electronics and Electrical Engg.
Indian Institute of Technology Guwahati
Guwahati - 781 039, Assam, India.

Dr. Prithwijit Guha
Associate Professor
Dept. of Electronics and Electrical Engg.
Indian Institute of Technology Guwahati
Guwahati - 781 039, Assam, India.

To

The Almighty

&

My dear parents

Ranajit Choudhury and Pampa Das Gupta

for their love and support

Acknowledgements

I am obliged to the Almighty for His divine guidance and blessings. I attribute this achievement to my parents for their constant blessings, support, and silent prayers for my success, and moreover, for making me stand in this position. This thesis would not have been possible without the immense help and support of several people in various measures. I would like to convey my acknowledgment to all of them. First and foremost, I express my sincere gratitude to my research supervisors, Prof. S. R. Ahamed and Dr. Prithwijit Guha for providing me with an opportunity to work under their guidance. It would be completely impossible for me to bring the research as well as the thesis to this form without the immense facilities provided by them in the VLSI Lab-II Laboratory and the freedom of work they have given to me. I am thankful to my doctoral committee members Prof. R. Sinha, Dr. Gaurav Trivedi and Dr. Aryabartta Sahu for their encouragement and valuable suggestions on my work. I would like to thank all faculty members and the office staff of the Department of Electronics and Electrical Engineering, IIT Guwahati, for their help in carrying out this research work. I am thankful to Jinti Hazarika, Riyazuddin Khan, Raktim Acharjee, Aikendrajit Ningthoujam, and all other members of the VLSI Lab-II. I would like to thank my seniors, Dr. Tasleem Khan and Dr. Satyajit Bora.

Rituparna Choudhury

Abstract

This thesis presents the hardware realization of three DT algorithms namely, Two Means Decision Tree (TMDT), Hybrid Decision Tree (HDT), and Perceptron Decision Tree (PDT). The TMDT algorithm classifies the data based on two-means clustering in each node. This reduces the computation to a great extent and thus, results in efficient hardware. First, the offline implementation of TMDT training is performed where the entire data is loaded to on-chip memory on Field Programmable Gate Array (FPGA). In this work, the TMDT algorithm is implemented in serial and mixed mode for binary classification only. However, the serial processing of blocks limits the speed of execution. So, next, a mixed implementation of TMDT training is proposed on FPGA. In this work, some blocks are implemented in pipelined manner and some blocks in parallel to minimize the latency. The memory access latency increased to a great extent due to large on-chip memory. Also, the huge memory consumption limited the training data size. So, the TMDT is modified and a batch-mode training process is implemented for multi-class classification on FPGA in the next chapter. In this implementation, the training data is divided into batches and one batch at a time is loaded into the chip memory to train the DT. This batch-mode implementation removed any constraint on the training data size. The accuracy of TMDT can be enhanced by implementing hybrid nodes. So, next, HDT and its training implementation on FPGA are proposed. This DT is a hybrid combination of split nodes hosting mean-dependent and axis-aligned split decision functions. The mean-dependent nodes are similar to the TMDT nodes and the axis-aligned split function parameters are learned by reducing the number of impurity computations. The HDT is observed to perform better than TMDT. Although there is a little increase in training latency, the performance measures (accuracy and F1-score) were observed to improve. However, the HDT performance was still inferior for small-sized datasets. To further improve the accuracy of the small-sized datasets, PDT training is implemented on FPGA. In this DT, each split node hosts a single output perceptron (no hidden layer). For implementing a perceptron efficiently in hardware, the perceptron architecture consisting of Offset Binary Coding (OBC) and Co-Ordinate Rotation-axis Digital Computer (CORDIC) is proposed and implemented on both FPGA and Application Specific Integrated Circuits (ASICs). The OBC architecture has been used in literature for inner product calculation in filters where multiplication is implemented using shifters and adders. Also, training hardware for PDT is proposed on FPGA. Next, classification hardware for PDT is proposed for bio-medical applications on both FPGA and ASIC. The DT algorithms pre-

sented in this thesis have lower complexity compared to CART, C4.5 or ID3 algorithms. The training of these DT algorithms is implemented on FPGA and found to be both resource-efficient than the existing training accelerators. The serial architecture consumes fewer resources and runs at least 10$\times$ faster than the software implementation. The serial implementation of TMDT optimizes the resource consumption by at-least 8$\times$ as compared to the existing training accelerator for CART. The mixed implementation is found to be at least 14$\times$ faster than the software implementation. The batch-mode implementation is found to speed up the training by at least 27$\times$ as compared to software implementation. It halves the LUT consumption as compared to previous designs. It also exhibits 10$\times$ reduction in BRAM usage as compared to the training accelerator for CART. The PDT training is accelerated by 34$\times$ for the worst-case scenario (largest dataset) as compared to software implementation. This design almost halves the resource utilization as compared to previous designs and has 6$\times$ saving in BRAM consumption as compared to CART training accelerator. This hardware achieves a speed-up by a factor of 2 as compared to the software.

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
ML	Machine Learning
NN	Neural Network
SVM	Support Vector Machine
DT	Decision Tree
CART	Classification and Regression Tree
TMDT	Two Means Decision Tree
HDT	Hybrid Decision Tree
PDT	Perceptron Decision Tree
OBC	Offset Binary Coding
CORDIC	Co-Ordinate Rotation-axis Digital Computer
KNN	K Nearest Neighbour
NT	Neural Tree
VLSI	Very Large Scale Integration
GPU	Graphical Processing Unit
ANN	Artificial Neural Network
DNN	Deep Neural Network
CLB	Configurable Logic Block
I/O	Input/Output
LUT	Look-Up Table
RTL	Register Transfer level
HDL	Hardware Description Language
CPU	Central Processing Unit

DA	Distributed Arithmetic
MAC	Multiply-and-Accumulate
BRAM	Block Random Access Memory
DRAM	Distributed Random Access Memory
FF	Flip Flop
DSP	Digital Signal Processor
MLP	Multi-Layer Perceptron
RLC	Rotational Co-Ordinate Rotation-axis Digital Computer
VLC	Vector Co-Ordinate Rotation-axis Digital Computer
FSM	Finite State Machine

List of Symbols

q	Node number
n_q	Number of data vectors entering node q
C	Total number of classes in data matrix
b	Batch-size of data
N	Number of epochs
H	Total number of batches
$\mathbf{X}_q$	Data-matrix of node q
$\mathbf{X}_{qL}$	Data-matrix of left child of node q
$\mathbf{X}_{qR}$	Data-matrix of right child of node q
$\mathbf{x}^k$	k^{th} data-vector belonging to data-matrix $\mathbf{X}_q$
m	Dimension of data vector $\mathbf{x}^k$
$\mathbb{R}^m$	Data-space consisting of m-dimensional real numbers
$\mathbf{x}^k[l]$	l^{th} dimension of data-vector $\mathbf{x}^k$
$y(\mathbf{x}^k)$	Actual class of data vector $\mathbf{x}^k$
$\hat{y}_k$	Predicted class of data vector $\mathbf{x}^k$
μ_{qL}	Left mean vector of node q
μ_{qR}	Right mean vector of node q
d^k_{qL}	Distance of $\mathbf{x}^k$ with μ_{qL}
d^k_{qR}	Distance of $\mathbf{x}^k$ with μ_{qR}
ζ_{max}	Maximum depth of DT
η_p	Minimum purity to declare a node as leaf node
ρ_c	Minimum data-vectors required to declare a node as split node
α	Mean-update factor

$\mathcal{T}_H$	DT trained on data batch H
$\mathbf{w}$	Weight vector
γ	Weight update factor
N_{DT}	Number of DTs used for feature selection

1

Introduction

Contents

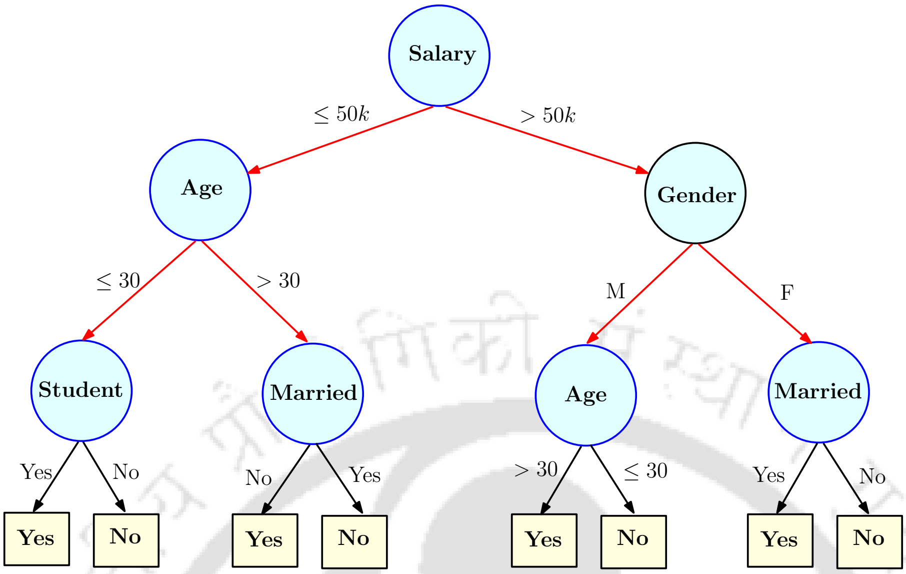

Figure 1.1: The flow chart of Decision Tree algorithm.

1.1 The Decision Tree (DT) Algorithm

Machine Learning (ML) algorithms are used to train a model from a dataset for performing classification or regression tasks. Supervised ML algorithms use labeled training data for model learning. Artificial Neural Networks (ANN) [1], Support Vector Machines (SVM) [2] and Decision Trees (DT) [3] are widely used as supervised ML algorithms. Unsupervised ML algorithms use unlabeled training data and group them into clusters. K-means [4], hierarchical agglomerative clustering [5] and DB-SCAN [6] are a few examples of unsupervised learning algorithms.

Due to simplicity, DTs are more suited for low-power and low-cost applications [7]. The DT is a supervised algorithm that can be used for both classification and regression tasks. Classification involves identifying and assigning a category label to the data by evaluating its features. While regression is predicting the value of a particular data feature by evaluating the feature information. The parameters of the DT algorithm are first tuned using a set of training datasets in the training phase. The performance of the DT algorithm is then measured by using it to classify the test dataset.

The basic structure of DT algorithm comprises of a tree with split or decision nodes and leaf or label nodes. A binary DT structure is shown in Fig. 1.1. It partitions the input space in a hierarchical top-down approach. The data is first input to the root node and then continuously routed by decision nodes and sent to the appropriate child node till it arrives at a leaf node where it is assigned a label. The DT can function with a hybrid structure. That is, different split nodes in a DT can host decision functions of different forms. The round nodes in blue represent split nodes and the square nodes in yellow represent the leaf nodes. Data partitioning at split nodes is represented with a red line and leaf node labeling is represented with a black line. In a DT, the split node stores the split function parameters. The input data is passed from parent to child split node (at consecutive depth) while checking different feature information till it reaches the terminal leaf node where it is classified with a category label. In this figure, the DT determines whether a person's loan can be approved (labeled Yes) or not (labeled No). As shown, the first node compares the salary feature of the person. If the salary is greater than 50k, then, the gender is checked. Otherwise, it is sent to the left node where age is checked. If the age is greater than 30 and the person is married then the loan is not approved otherwise, the loan is approved. A DT algorithm has two phases, training and inference. In the training phase, the DT parameters are tuned according to the training dataset [3]. To enable a DT to perform autonomous classification or inference for a particular application, it is important to train the DT with data from that application. The dataset is divided into two disjoint parts – training and test dataset. A training dataset consists of a majority of data instances that are used to train the DT while the test set consists of fewer data instances used to validate the efficacy of the training algorithm. The training of DT involves tuning the algorithm parameters using this training dataset so as to efficiently perform classification or inference on unseen input data. During training, the growth of the DT is controlled by termination conditions. Then, pruning is applied to the full-grown tree to avoid over-fitting or memorization of data. Over-fitting results in poor generalization and thus affects tree performance for unseen data. There are two types of pruning – pre-pruning and post-pruning. In pre-pruning, a split node is checked for pruning conditions before splitting. If it satisfies the condition then, it is set as a leaf node and it is not split further. In post-pruning, the DT is grown to full depth, and then the pruning conditions are checked. Thus, post-pruning is more suitable for efficient hardware implementation as hardware resources are allotted at the very beginning. This also allows for efficient pipelining in hardware which assumes that no node is missing in the middle (as it happens in the case of pre-pruning). The DT algorithms hierarchically partition the input space in a top-down manner till specific termination conditions are satisfied. This hierarchical partitioning is achieved by routing data through a decision function hosted at the split nodes of the DT. The DT flow terminates at leaf nodes. These leaf nodes correspond to almost pure regions of input data space containing a majority of the instances with the same label. The model complexity and training time of DT depend on decision functions hosted at splitting nodes. This ranges from simple axis-aligned rules (C4.5) [8], oblique splits [9] to complex networks [10]. Most practical applications use software platforms to train the DT. Model parameters obtained by such training are incorporated in hardware chips for operations involving embedded systems. However, re-training parameters on these chips become difficult for adapting to different applications. Besides, several applications prefer the local availability of data to embedded processors [11]. Also, an embedded device capable of the model update has the advantage of use in various applications. Graphical Processing Units (GPUs) are often deployed for ML applications involving heavy computations which also lead to heavy power consumption. Comparatively, Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) are convenient low-power alternatives to accelerate training. In contrast to FPGAs, ASICs do not offer reconfigurability. However, ASICs offer higher speed and low power advantages over FPGAs [12].

1.2 Existing works on DT Algorithm

Several DT training algorithms have been reported in the literature [13–18]. The Deep NT proposed in [13]consists of neural networks. These networks are trained using gradient descent rather than greedy splitting. This tree also implements self-pruning at both feature and split levels. A modified J48 DT algorithm is used in [14] for energy security. This intrusion detection system handles interruption and interference attacks in grids. The RES-OT proposed in [15] uses oblique splits for Parkinson's tremor detection. It uses l2 regularisation of weights instead of l1 regularisation for better accuracy. The Eager DT proposed in [16] trains the DT to include results for unknown attribute values. This enables the trained DT to handle data with missing attribute values. The DT training in [17] considers re-training of nodes based on the importance criterion. The node having higher importance is chosen for re-training first. The DT proposed in [18] implements a two class support vector machine (SVM) in each node. Here, the root node data is divided into almost homogeneous clusters and given as input to the children nodes. These children nodes consist of the SVM which performs the splitting. These highly complex DT algorithms are implemented using software that results in higher run-time. The hardware implementation can speed-up the execution by at-least $10\times$ for the most basic hardware architecture [19].

1.2.1 Challenges in DT training and need for training accelerators

The DT training consumes greater amount of time as compared to classification. So, speeding up the training phase is very much important for efficient implementation of DT training algorithms. Moreover, the software executes the code sequentially which leads to longer training time. Hardware implementation is a good solution to speed up the execution due to provision of higher parallelism which reduces the execution time. The hardware implementation of DT training has increased the speed and thus, resulted in a much faster training [20]. Furthermore, training on hardware would offer possibilities of on-device training and reduce the security risk of data leak which happens in case of training on cloud [21]. The Classification and Regression Tree (CART) architecture reported in [20] implements training for CART algorithm on multi-FPGA system in High Capacity server. This design achieves a speed-up of two orders of magnitude as compared to software solutions at the cost of increased complexities. The ASIC implementation of training accelerator is proposed in [22]. It implements training using both Two Means DT (TMDT) and axis-aligned DT. The DT giving the best result is chosen. It supports training for only upto 10 kilobyte of training data. The higher computational complexities involved in training conventional DT results in higher resource consumption which ultimately results in a rise in hardware costs involved in the design [23]. Therefore, implementing training for a low-complexity algorithm in hardware will increase resource usage efficiency.

1.2.2 Limitations of DT and necessity of NT

In conventional DTs, the split is axis-aligned. Thus, the decision boundary is highly affected by noise in training data. Therefore, the DT algorithm was modified to include oblique nodes as split nodes instead of axis-aligned nodes. These oblique nodes consist of either a single perceptron or a fully-connected network of perceptrons like Artificial NN (ANN) [24], Deep NN (DNN) [21], etc. These hybrid structures were named as NT. The performance of NT algorithms were found to be comparable to NNs while the complexity is much lower [15].

1.3 Suitability of FPGA for training accelerator implementation

FPGAs are re-configurable chips consisting of Configurable Logic Blocks (CLBs), switching matrices, digital signal processing (DSP) cores for performing multiplications or additions, DDR4 memories and Input/Output (I/O) pads [25]. The circuit is implemented on FPGA using CLBs which consist of

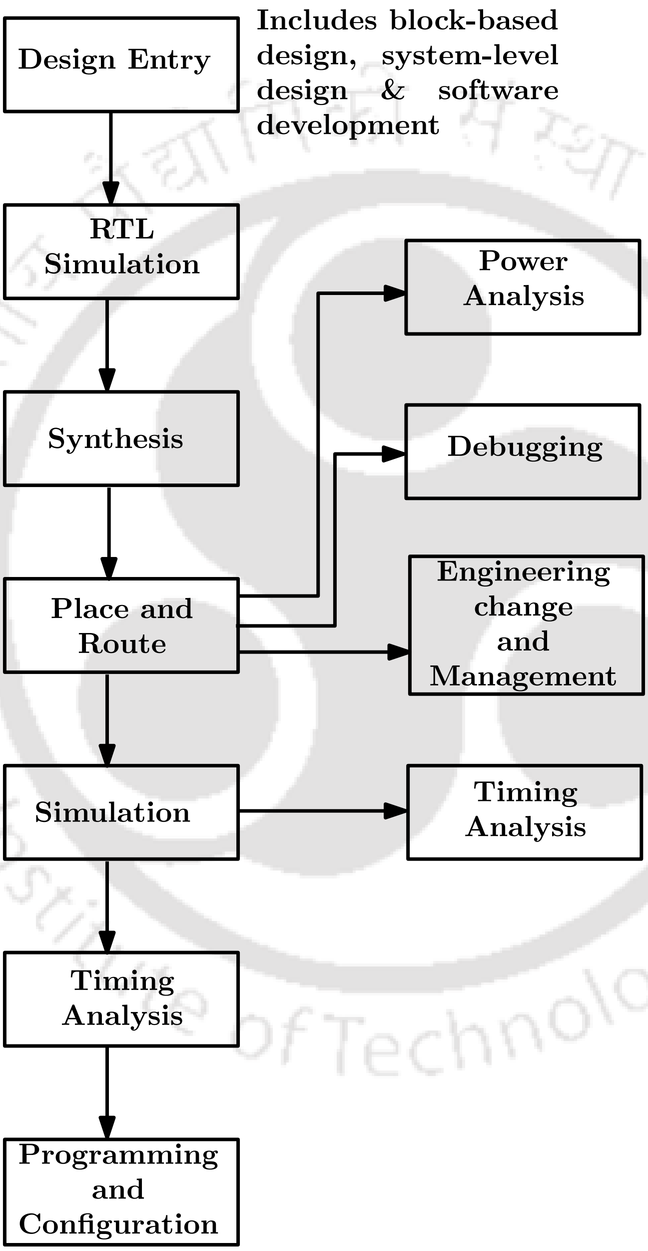

Figure 1.2: The steps involved in implementing a circuit on FPGA

Look-Up Tables (LUTs). The connection between the CLBs are done using switching matrices. The switching matrices are connected according to the circuit to be designed. Therefore, the connection of the CLBs are re-configurable. Thus, the circuit implemented on FPGA can be easily re-designed. The steps involved in implementing a circuit on FPGA is shown in Fig. 1.2. The flow starts with designing the circuit and then, the Register Transfer Level (RTL) simulation is checked to validate its correct functioning. Next, the circuit is synthesized, i.e., transformed into CLBs. Afterwards, during final implementation, this synthesized circuit is placed on the board and routing is done. Finally, the post-implementation simulation is checked for the proper functioning of the circuit. Lastly, the board is programmed and configured with the implemented circuit. For re-configuring the hardware, the changes are made in the design and the above steps are repeated. GPUs have been used to implement many in-memory architectures for NN classification but GPUs tend to have higher power consumption [26]. ASIC have been used to improve speed and reduce power but they incur high cost due to full-custom design and cannot be re-configured [27]. So, semi-custom designed FPGA platforms can be used to implement training. The re-configurability features make it ideal for implementing training of DT as FPGAs can be easily re-trained with new data to suit a different application. Thus, re-configurable training architectures designed using FPGA can be used for various applications [19]. The serial architecture can be used to design resource-efficient low-speed hardware [27]. The parallel architectures result in high speed and high resource consumption. While the pipelined architectures can be designed to optimize resource consumption and obtain high speed [12]. Pipelining optimizes resource utilization by sharing the computations in two successive iterations.

1.4 Existing classification hardwares

1.4.1 DT classification hardwares

In order to speed-up the classification, several hardware architectures implementing DT classification has been proposed in [12, 27–31]. The architecture proposed in [28] is used to filter the sensor data. Here, a serial architecture implemented on both FPGA and Von-Neumann Central Processing Unit (CPU) is compared. The FPGA architecture is found to be less power-hungry. While the software counterpart has higher throughput and power usage. Saqib *et al.* in [12] proposed a pipelined architecture to achieve faster classification. Tong *et al.* proposed one architecture that provided balanced classification in a shorter time and another architecture produced an optimized tree with less

Table 1.1: Table recording the key features and limitations of existing works

Existing hardwares	Key features	Limitations
[27]	Energy efficient bio-medical hardware	Only classification
[12]	Pipelined classification	Only classification
[28]	Power efficient design with sacrificed speed	Only classification
[29]	Balanced DT for fast execution (DQ) and low complexity DT (ODT) for resource efficiency	Only classification
[30]	Hardware for sea-state classification	Only classification
[31]	Power-area efficient spike classification for neural implants	Only classification
[32]	Multi-core k-means clustering	Only classification
[33]	k-means clustering for image processing	Only classification
[20]	CART training accelerator	Require multiple FPGAs
[22]	TMDT+Axis-aligned DT training on ASIC	High resource usage

area in [29]. The resource-constrained architecture for seizure detection proposed by Shoaran *et al.* in [27] reduced the resource consumption. This resource efficiency resulted in low cost hardware. The hardware architecture for texture sea-state classification proposed in [30] performs automatic texture recognition of sea-states. The hardware proposed by Yang *et al.* in [31] proposes a spike classification System-on-Chip to reduce the data rate of a brain-machine interface. The classification was realized using a DT-based classification which acquired good accuracy. Zhu *et al.* proposed a hardware model for epilepsy, Parkinson's and finger movement classification using resource-efficient oblique trees in [15]. This work used oblique splits in the split nodes to achieve higher accuracy with only 1 tree and 7 split nodes. The work proposed power-efficient regularisation of weights and compared it with conventional l1-regularisation. The model was found to have superior performance as compared to a random forest consisting of a large number of DTs.

1.4.2 Other classification hardwares

Data instances from similar categories are found to form single or multiple clusters in space. Such clusters of different categories may also overlap. Thus, a clustering algorithm can be employed to perform classification. As clustering does not require sorting so, the complexity is reduced. K-means is a well-known clustering technique where instance labels are assigned based on their nearest cluster centroid [4]. It is a very popular and less complex clustering algorithm where data points are assigned to the nearest cluster. The design proposed in [32] implemented k-means algorithm classification parallely on multiple FPGA to achieve $10\times$ speed-up as compared to software implementation. The

Euclidean distance calculation in the k-means algorithm was replaced by Manhattan and Max distance in [4]. This enabled the distance calculation implementation without multipliers. The k-means tree implementation in [33] was realized using a kd-tree pruning on the search space. This pruning leads to almost 5× lesser computation as compared to conventional k-means classification.

1.4.3 Perceptron hardware

The last three decades have witnessed a steady rise in the innovations and applications of ANN in different fields. A rise in hardware implementations of ANN is also witnessed on account of higher speed [21] and field deployability. The hardware implementations on FPGA and ASIC are found to be less power-hungry and more area-efficient as compared to the implementations on GPU [10]. A perceptron (or single neuron) forms the basic building block of an ANN. Different variants of ANNs are realized as connectionist architectures of perceptrons in parallel and cascade [24]. In most practical applications, ANNs contain a large number of perceptrons. Thus, due to high network complexity, the hardware implementations of ANN containing many perceptrons tend to consume more power. The perceptron involves an inner product computation unit followed by an activation unit. The existing perceptron hardware realized on FPGA uses a conventional multiplier for the inner product and a non-linear function memorizer for sigmoid evaluation [34]. This resulted in high resource usage and error percentage. Distributed Arithmetic (DA) is an efficient multiplier-less architecture that replaces the Multiply-and-Accumulate (MAC) unit with LUTs, shifters, and adders. This allows the calculation of the inner product using lesser resources and time [35]. But, the size of LUT increases exponentially with an increase in data dimension. The Offset Binary Coding (OBC) architecture reduces the LUT size to half as compared to DA by modifying the addressing scheme of LUTs [36]. The sigmoid unit implemented using Co-Ordinate Rotation Axis Digital Computer (CORDIC) algorithm [37,38] can reduce the error while minimizing the resource consumption. There are many works in literature that implemented the sigmoid function using multiplier-less CORDIC architecture [37, 38]. The implementation of CORDIC enables more accurate computing of sigmoid function as compared to approximation [34].

1.5 Motivation and problem formulation

As discussed in Section 1.4, the classification hardwares proposed in the literature are proven to run faster than the software implementation. However, these hardwares are restricted to a particular application. For classifying data from a particular application, the DT should be trained using the dataset from that application. To reuse the hardware for a different application, the hardware needs to be re-trained using a different dataset. So, training hardware designed on FPGA can be used for many applications. Thus, the reconfigurability feature of FPGA allows easy re-training. This makes FPGA a suitable platform for training hardware implementation. Therefore, the reconfigurability and higher speed of execution make FPGA a suitable platform for DT training implementation. However, conventional DT algorithms like C4.5, CART, or ID3 require highly complex computations like sorting. The CART training accelerator requires a set-up consisting of 4 FPGAs [20]. Thus, the cost of hardware implementation is directly proportional to the computational complexity. The training implementation for these DT algorithms in hardware incurs huge cost and slower execution. So, the calculation complexity needs to be reduced for resource-efficient hardware implementation. The resource efficiency is very important for portable and bio-medical applications. While, the higher speed is required for real-time processing applications, like activity recognition and some bio-medical applications like Parkinson's tremor detection.

Hence, in this thesis, low-complexity DT algorithms have been designed. The hardware architecture for these algorithms, designed and implemented on FPGA, is also proposed. These hardware are found to speed up the training as compared to software platforms. First, two new DT algorithms and their training architectures are proposed. Next, a neuron hardware implementation on FPGA and ASIC is proposed. Similarly, a training architecture for DT implemented using neurons is proposed. Also, the classification hardware with offline training for this DT implemented on FPGA and ASIC is proposed. The contributions of each chapter of the thesis are discussed next.

1.6 Scope of the thesis

The Two Means Decision Tree (TMDT) algorithm and its serial and mixed parallel and pipelined hardware implementation are proposed in Chapter 2. This algorithm is similar to the hierarchical two-means algorithm and employs a divisive clustering-based approach. It stores two mean vectors (left and right) in the split nodes. The data instances are routed toward the left (or right) child node based on their proximity to the left (or right) mean vector. Two FPGA hardwares for training the TMDT are proposed. First, a serial architecture is proposed for efficient resource usage. Next, a mixed parallel and pipelined hardware architecture is also proposed. The execution at nodes is pipelined to increase the throughput while minimizing resource consumption. The memory access and distance calculation are parallelized for all dimensions to reduce latency. These hardwares can be re-trained by using reset signal which makes it adaptable to changes in training data. The inferencing performance of all ML models (and, hence DT) increases with exposure to larger training datasets. However, these architectures were not able (constrained by on-chip memory requirements) to use larger datasets and were designed for binary classification problems.

A batch-mode training of TMDT for multi-class applications is proposed in Chapter 3. The memory limitation of FPGA put a constraint on the DT training dataset size in the previous hardware (proposed in Chapter 2). Here, the large training dataset (e.g. 100k 4-dimensional vectors) is partitioned into batches (small-size data subsets of 512 instances). The DT parameters are updated using each batch. One epoch implies the DT parameter learning using all the batches. In each epoch, the batches are presented in a different order. An inferencing performance improvement is observed with the proposed training scheme involving multiple epochs and batch reshuffling. Thus, unlike offline implementations proposed in Chapter 2), the batch-mode implementation permits training on a larger dataset. This overcomes the constraint of on-chip memory consumption. The proposed architecture implements pipelining of depths for efficient resource usage as it requires a data instance to be loaded only once at each depth. Here, one pipeline consists of depths 1-2 and the second pipeline consists of depths 3-4. The implementation of batch-mode training using large datasets results in better accuracy. In order to optimize the clock cycle consumption, a pipelined architecture is proposed to implement multiplication involved in the distance computation block. The accuracy of the TMDT algorithm can be further improved by implementing hybrid nodes.

A Hybrid Decision Tree (HDT) algorithm and its FPGA implementation are proposed in Chapter 4. The HDT speeds up execution as compared to axis-aligned DT (C4.5) while maintaining good detection accuracy. The HDT structure has the following three types of nodes – mean-dependent split nodes, axis-aligned split nodes, and terminal leaf nodes. The tree starts with a mean-dependent root node. The input dataset is split through mean-dependent nodes as in TMDT till a certain depth of ζ_{md}. From depth $\zeta_{md} + 1$ onward till $\zeta_{max} - 1$ (where ζ_{max} is the maximum depth till which the DT is grown), the data are processed through axis-aligned split nodes. The DT terminates with all leaf nodes at the depth ζ_{max}. The axis-aligned nodes store two node parameters – threshold value (θ) and dimension (l). During classification, when data enters the node, the l^{th} dimension value is checked.

If the value is less than θ, then the data is routed to the left child node and to the right child node, otherwise. The FPGA implementation is observed to speed up the training while reducing the hardware cost. The performance of classification can be further improved by hosting neurons in the DT split nodes. This exhibits good performance while maintaining low complexity and is important for hardware implementation.

In Chapter 5, a resource-efficient and high-speed perceptron hardware is proposed. In this hardware, the inner product calculation is realized using OBC and the sigmoid activation function calculation is done using CORDIC architecture. This also improves the accuracy as compared to the existing architectures. Next, a training hardware for Perceptron Decision Tree (PDT) is proposed. Here, a perceptron based decision function is used by each split node of DT. The input data features are filtered and m_s best features are selected to achieve better classification accuracy while having lower resource usage. The inner product unit was implemented using shift and add and the sigmoid block was designed using the CORDIC algorithm. The DT is found to be more resource-efficient. Next, classification hardware implemented on both FPGA and ASIC using PDT is proposed. In this design, the training was performed offline and the trained DT is loaded into hardware for performing classification. This hardware implemented Parkinson's detection and three types of activity recognition (lying/sitting, lying/walking, and walking/standing). A mode number is used to select the application and accordingly, the hardware can switch between these four applications. The proposed hardware sacrificed the speed to achieve resource efficiency which is important for biomedical and wearable applications.

2

Two Means Decision Tree: Offline and binary class implementation

- R. Choudhury, S. R. Ahamed, and P. Guha, "Efficient Hardware Implementation of Decision Tree Training Accelerator", in SN Computer Science 2, 360, 2021.
- R. Choudhury, S. R. Ahamed, and P. Guha, "Training Accelerator for Two Means Decision Tree", in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 7, pp. 1465-1469, July 2021.
- R. Choudhury, S. R. Ahamed, and P. Guha, "Efficient Hardware Implementation of Decision Tree Training Accelerator", in IEEE International Symposium on Smart Electronic Systems (IEEE-iSES), 2020.

Contents

Overview

The Decision Tree (DT) training implementation on hardware is highly resource intensive. The resource usage can be optimized by implementing a low-complexity DT algorithm. So, this chapter first proposes a low-complexity algorithm that realizes a Two Means Decision Tree (TMDT). Then, a serial architecture of TMDT implemented on FPGA is proposed. The serial hardware is observed to attain a speed-up of 10$\times$ as compared to the software implementation. Next, a mixed parallel and pipelined architecture for the TMDT is proposed. This mixed architecture is observed to attain a speed-up of 14$\times$ as compared to the software platforms. Further, these architectures are observed to be more resource efficient as compared to the existing k-means classification hardware. Thus, implementing training for a low-complexity algorithm (TMDT) is found to be more resource efficient than the high-complexity algorithm (k-means) classification hardware.

2.1 Problem formulation

In this chapter, we first propose a TMDT algorithm for binary class data. This algorithm exploits the fact that similar data are generally found in clusters. Thus, the data found in the same clusters can be grouped together and assigned a common class label. Then, the mean of each cluster can be used to classify the data belonging to the same cluster. In contrast to conventional axis-aligned DT algorithms (C4.5), TMDT does not require sorting which reduces the complexity. Thus, TMDT is resource-efficient and suitable for hardware realization. The reduced complexity of the TMDT algorithm as compared to the conventional axis-aligned DT algorithm enables it to execute at a higher speed. In this chapter, two architectures for the implementation of TMDT are proposed. These architectures are compatible with 32-bit integer binary class training data. These designs have been tested on binary datasets of variable size and dimension. Thus, the proposed hardware are compatible with a wide range of training datasets. Moreover, the proposed architectures can be easily re-trained for the next set of data using a single RESET signal. This on-the-go training makes the hardware versatile for any kind of application. The proposed architectures are implemented on an FPGA platform. First, a resource-efficient serial architecture for the acceleration and implementation of DT training is proposed. Second, a trade-off between speed and resource usage is obtained by designing a mixed parallel and pipelined architecture for TMDT training. This architecture uses a combination of parallel execution for training time reduction and pipelined execution to minimize resource consumption.

Algorithm 2.1: TMDT Algorithm

while $k \leq n_q$ **do**
- Load node data;
- Calculate initial mean μ^0_{qL} and μ^0_{qR};

while $k \leq n_q$ **do**
- Calculate *distance*$(\boldsymbol{\mu}^{(\mathbf{k-1})}_{\mathbf{qL}}, \mathbf{x^k})$ & *distance*$(\boldsymbol{\mu}^{(\mathbf{k-1})}_{\mathbf{qR}}, \mathbf{x^k})$;
- **if** *distance*$(\boldsymbol{\mu}^{(\mathbf{k-1})}_{\mathbf{qL}}, \mathbf{x^k}) \leq$ *distance*$(\boldsymbol{\mu}^{(\mathbf{k-1})}_{\mathbf{qR}}, \mathbf{x^k})$ **then**
  - Update $\boldsymbol{\mu}^{(\mathbf{k-1})}_{\mathbf{qL}}$;
  - $(1-\alpha)\mu^{(k-1)}_{qL} + \alpha\mathbf{x}^k$;
- **else**
  - Update $\boldsymbol{\mu}^{(\mathbf{k-1})}_{\mathbf{qR}}$;
  - $(1-\alpha)\mu^{(k-1)}_{qR} + \alpha\mathbf{x}^k$

while $k \leq n_q$ **do**
- Calculate *distance*$(\boldsymbol{\mu}_{\mathbf{qL}}, \mathbf{x^k})$ & *distance*$(\boldsymbol{\mu}_{\mathbf{qR}}, \mathbf{x^k})$;
- **if** *distance*$(\boldsymbol{\mu}_{\mathbf{qL}}, \mathbf{x^k}) \leq$ *distance*$(\boldsymbol{\mu}_{\mathbf{qR}}, \mathbf{x^k})$ **then**
  - Update left child node;
- **else**
  - Update right child node;

while $NodeDepth \leq \zeta_{max}$ **do**
- **if** *(purity(Node)* $\geq \eta_p$*) or (NodeData* $< \rho_c$*)* **then**
  - Set current node as label node;
  - Set label as max(class1,class2);
- **else**
  - Preserve the node structure;

2.2 Proposed TMDT algorithm

The TMDT algorithm is motivated by the Competitive Means DT approach proposed in [39]. The TMDT split nodes host two mean vectors that decide the data routing along the tree as shown in Fig. 2.1. Let $\mathbf{X}_q = [\mathbf{X}_{q0}, \mathbf{X}_{q1}]$ be the training dataset for node q. Here, $\mathbf{X}_{q0} = \{\mathbf{x}^i : y(\mathbf{x}^i) = 0; \mathbf{x}^i \in \mathbb{R}^m;$ $i = 0, \dots (n_{q0}-1)\}$ and $\mathbf{X}_{q1} = \{\mathbf{x}^j : y(\mathbf{x}^j) = 1; \mathbf{x}^j \in \mathbb{R}^m; j = 0, \dots (n_{q1}-1)\}$. Here, $y(\mathbf{x}^i)$ refers to the class label of $\mathbf{x}^i$. The mean vectors (split parameters) μ_{qL} and μ_{qR} are learned from $\mathbf{X}_q$. The TMDT algorithm is described in Algorithm 2.1. The TMDT split node parameters are learned in two phases. First, the mean vectors μ^0_{qL} and μ^0_{qR} are approximated as averages of data instances from $\mathbf{X}_{q0}$ and $\mathbf{X}_{q1}$, respectively. In the second phase, data instances of $\mathbf{X}_q$ are used to incrementally update the mean vectors. An instance $\mathbf{x}^k \in \mathbf{X}_q$ $(k = 0, \dots (n_q-1);\ n_q = n_{q0}+n_{q1})$ is tested for proximity against

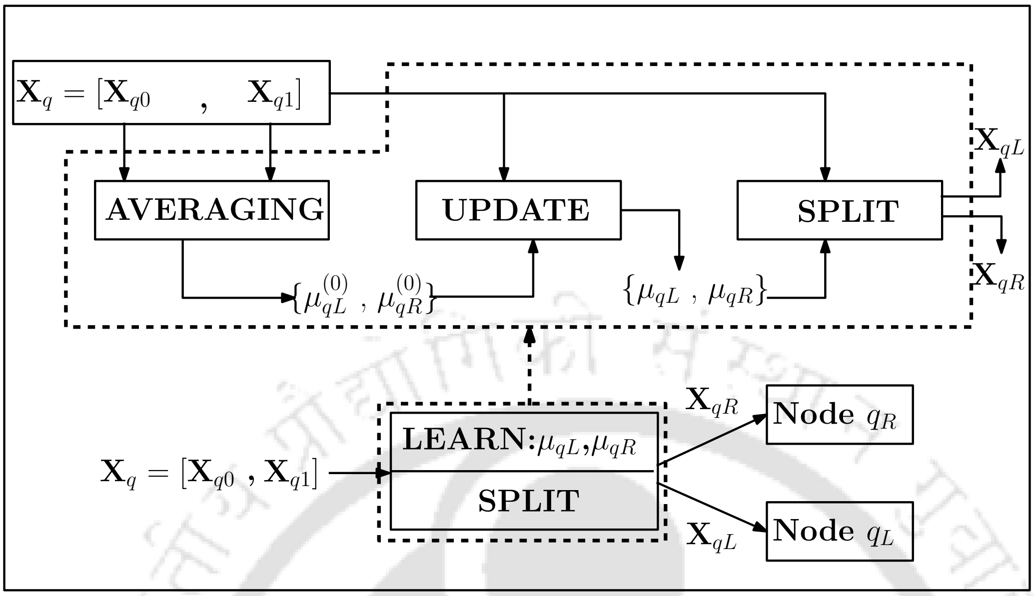

Figure 2.1: The flow diagram of TMDT.

the two means. Let d_{qL}^k and d_{qR}^k be the respective Euclidean distances of $\mathbf{x}^k$ from $\mu_{qL}^{(k-1)}$ and $\mu_{qR}^{(k-1)}$. The nearest mean vector from $(k-1)^{th}$ iteration is updated with $\mathbf{x}^k$ (2.1 and 2.2). The accuracy values for different values of $\alpha \in (0,1)$ are shown in Fig. 2.2. Here, the α values are taken such that it is realizable by the sum of power of two (SOPT), i.e., $0.03125, 0.0625, 0.125, 0.25$, and so on. Among these values, the maximum accuracy is exhibited for $\alpha = 0.875$ for all datasets (marked in black in the graph). This $\alpha = 0.875$ can be represented by 3 shift operations in hardware without incurring any accuracy loss.

$$\mu_{qL}^{(k)} = \begin{cases} (1-\alpha)\mu_{qL}^{(k-1)} + \alpha\mathbf{x}^k; & d_{qL}^k < d_{qR}^k \\ \mu_{qL}^{(k-1)}; & \text{Otherwise} \end{cases} \tag{2.1}$$

$$\mu_{qR}^{(k)} = \begin{cases} (1-\alpha)\mu_{qR}^{(k-1)} + \alpha\mathbf{x}^k; & d_{qL}^k \geq d_{qR}^k \\ \mu_{qR}^{(k-1)}; & \text{Otherwise} \end{cases} \tag{2.2}$$

The mean vectors μ_{qL} and μ_{qR} obtained at the end of the second phase are used to split $\mathbf{X}_q$ to children datasets $\mathbf{X}_{qL}$ and $\mathbf{X}_{qR}$. Here, $\mathbf{X}_{qL}$ and $\mathbf{X}_{qR}$ consists of instances that are respectively proximal to μ_{qL} and μ_{qR}. The split node q is initialised with two mean vectors μ_{qL}^0 and μ_{qR}^0. $\mathbf{X_q}$ first updates the two mean vectors then it is routed by these two mean vectors. The input is routed to the left child if it is closer to μ_{qL} and to the right, otherwise. Thus $\mathbf{X_q}$ is split into two sets $\mathbf{X_{qL}}$ and $\mathbf{X_{qR}}$

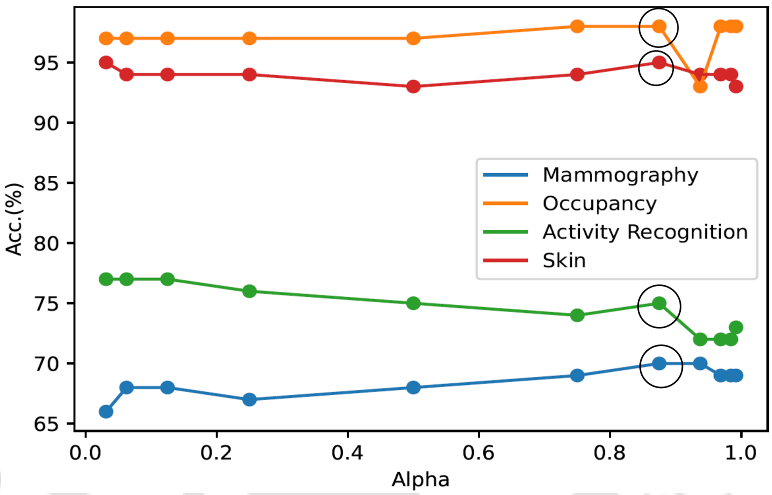

Figure 2.2: Plot showing variation of accuracy with Alpha (α) for all the 4 datasets [40].

which are assigned to the two child nodes of q, i.e., (qL) and (qR). These datasets are used further for learning the split node parameters for left and right child nodes of parent node q. This process is recursively continued until a fully grown tree is constructed with a certain maximum depth ζ_{max}. The fully grown TMDT is subjected to post-pruning. This post-pruning allows efficient pipelining of hardware architecture. A TMDT node is declared as a leaf node if one of the following conditions is satisfied [3].

- Based on the **purity criterion**, if the proportion of the majority class $\dfrac{max(n_{q0}, n_{q1})}{n_{q0} + n_{q1}}$ exceeds a purity threshold η_p.

$$\frac{max(n_{q0}, n_{q1})}{n_{q0} + n_{q1}} < \eta_p \quad : \text{Split Node} \tag{2.3}$$

$$\frac{max(n_{q0}, n_{q1})}{n_{q0} + n_{q1}} \geq \eta_p \quad : \text{Leaf Node} \tag{2.4}$$

- Using the **cardinality** criterion, if $n_q = n_{q0} + n_{q1}$ falls below a certain threshold ρ_c.

$$n_q < \rho_c \quad : \text{Leaf Node} \tag{2.5}$$

$$n_q \geq \rho_c \quad : \text{Split Node} \tag{2.6}$$

Here, ρ_c and η_p are the respective node input deficiency and node input purity thresholds. The value of η_p is generally taken as 0.95 (or 95%) for standard cases [3]. The value of $\rho_c = 20$ is obtained for maximum accuracy from Fig. 2.3 across all datasets. The accuracy remains unaffected for *Occupancy*

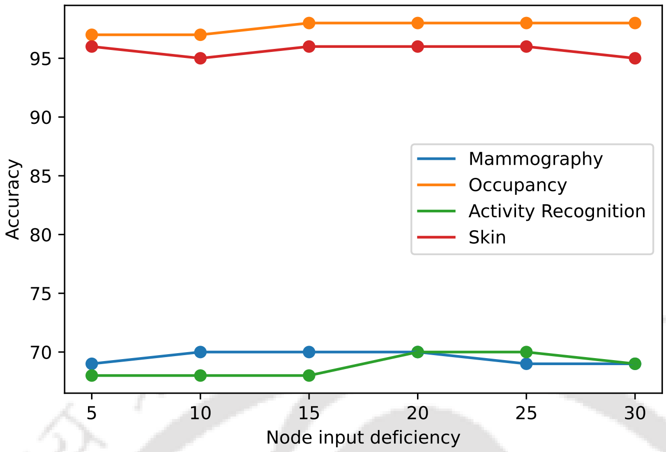

Figure 2.3: Plot showing variation of accuracy with node input deficiency (ρ_c) for $\alpha = 0.875$ [40].

dataset for $\rho_c > 20$. However, the accuracy decreases with further increase in node input deficiency for mammography, activity recognition and skin dataset. So, the optimal value is found to be $\rho_c = 20$. This may be explained by the fact that increasing node input deficiency decreases the number of split nodes which may decision parameter. In the post-pruning stage, if a split node is declared as a leaf node, then all its children nodes are removed. Also, a leaf node is assigned its majority class label as its parameter. In multi-class problems, the instances will be similarly clustered into two groups irrespective of their class labels. The leaf nodes will be tagged with the label of the dominant class in its input dataset.

During classification, an input data $\mathbf{x}^k$ is routed through the split nodes by checking its proximity to either of the two means. This routing terminates at one of the leaf nodes whose label is declared as the class label of $\mathbf{x}^k$.

A C4.5 decision tree splitting node would learn an axis-aligned rule of the form $\mathbf{x}[l] \geq \theta$. The optimal split parameters (l, θ) are obtained by sorting data in each dimension while maximizing the impurity drop from parent to children nodes. For example, in the case of $\mathbf{X}_q$, a total of $n_q \times m \times log(n_q)$ sorting operations and $n_q \times m$ impurity computations will be required. The TMDT, on the other hand, performs a divisive two-means clustering of input data. The TMDT node q would require only $4 \times n_q$ comparisons (two comparisons each for updating and splitting) and n_q update operations for each split. Thus, the latency of TMDT is much lower as compared to C4.5. The sort operations involve a large number of comparisons. The comparisons involved in sorting consume longer time as compared to additions and shift operations involved in update operations. The complexity is further reduced as the impurity of each TMDT node is computed only once at the post-pruning stage. The hardware implementation of TMDT learning is described in the subsequent section.

2.3 Proposed Hardware Architectures

2.3.1 Serial architecture

2.3.1.1 Overall architecture

In this section, we discussed the mapping of the TMDT algorithm onto a novel 32-bit hardware architecture as shown in Fig. 2.4. In order to implement the proposed TMDT algorithm discussed in Section 2.2, this architecture is designed to process each node q serially in order to optimize resource consumption and minimize power. In this architecture, the first block is the mean initialization module where, for each iteration, the data is loaded from the data memory and added to $\boldsymbol{\mu}_{qL}^{0}$ or $\boldsymbol{\mu}_{qR}^{0}$ obtained from previous iteration and result is forwarded to node memory after processing complete data stored in data memory. If the data label is 0, then it is added to left mean $\boldsymbol{\mu}_{qL}^{0}$ otherwise, to $\boldsymbol{\mu}_{qR}^{0}$. The data number counter is compared to the total node data. When the data number equals the total node data, then the final values of $\boldsymbol{\mu}_{qL}^{0}$ and $\boldsymbol{\mu}_{qR}^{0}$ are obtained by dividing $\boldsymbol{\mu}_{qL}^{0}$ and $\boldsymbol{\mu}_{qR}^{0}$ with the total number of data in that node belonging to class 0 and 1 respectively. This division operation is executed using shifters to speed up the division. To divide a number X by 2^{G}, X is shifted by G bits. If the divisor cannot be decomposed into a power of 2, then it is approximated to the nearest value of 2^{G}. In this way, two means for each node are calculated and stored as initial means in the node memory which constitutes the mean approximation module. This module consists of a multiplexer to select the appropriate mean according to the class label of data retrieved from Data Memory. The module is activated by the output of the comparator. The input to this comparator is total node data and the current data number under consideration (Data Num Counter). The output of the comparator activates the mean approximation module if the current data number is less than total node data otherwise, the final mean is calculated using a shifter and stored in Node Memory. In the subsequent phase, the mean update and split module are executed for the node q. In the mean update module, either $\boldsymbol{\mu}_{qL}^{0}$ or $\boldsymbol{\mu}_{qR}^{0}$ is updated by the data-instance. The mean closer to data, depending on the distance is updated according to (2.1) and (2.2). The details of the distance calculator architecture are presented in the next subsection. The multiplexer (MUX) selects the mean closer to the data.

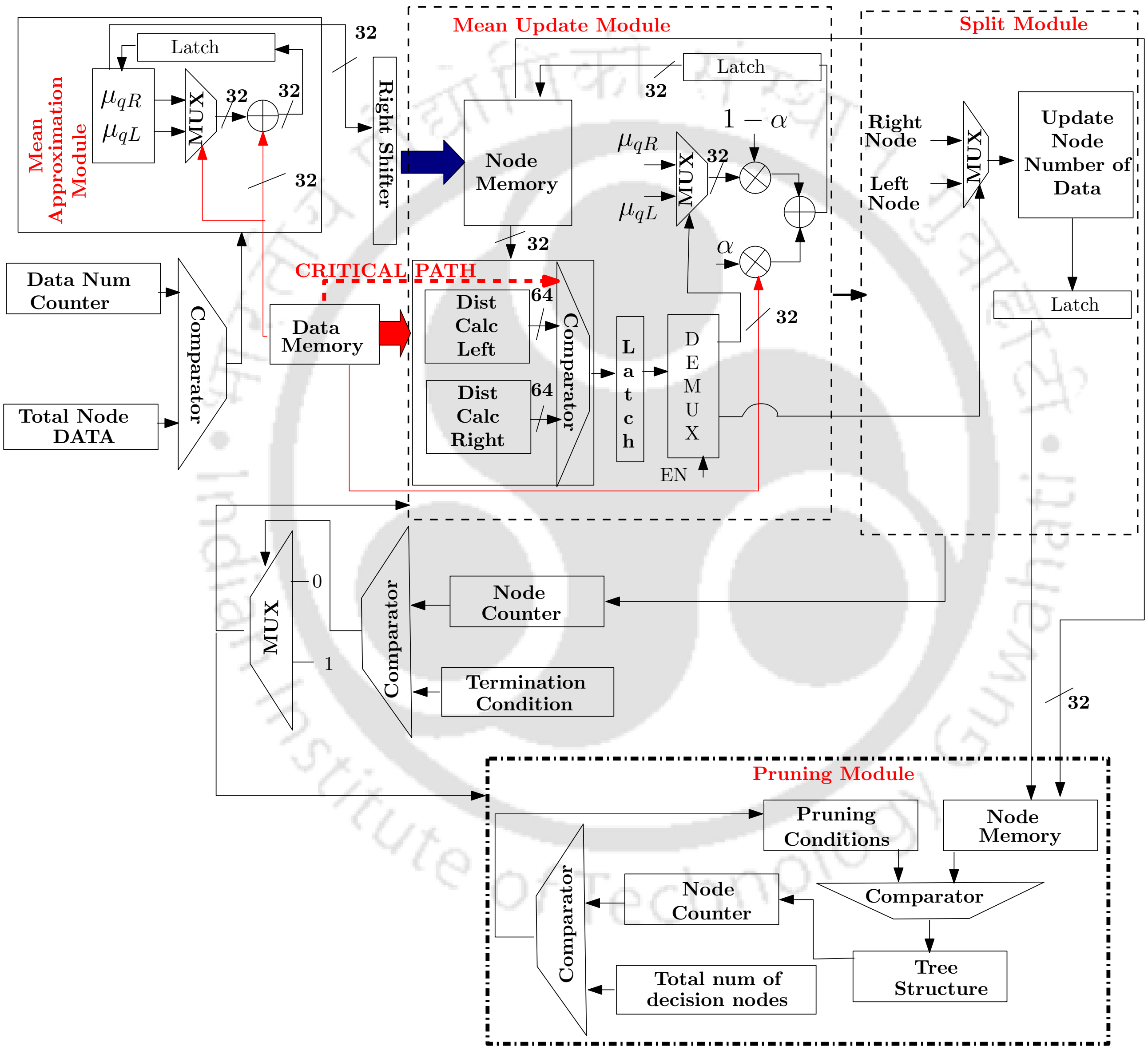

Figure 2.4: Proposed serial architecture for TMDT algorithm

The select line of this multiplexer is connected to the output of the comparator comparing left (d_{qL}^k) and right mean distance (d_{qR}^k) with k^{th} data-instance (Dist Calc Left and Right respectively). Once the mean update is done then the updated mean is stored back to the Node Memory. Once all the data belonging to that node is passed for the mean update, then the same distance calculators are re-used to calculate the distance between the updated mean stored in node memory and the node data in the split module using a switch. This switch is implemented using a de-multiplexer whose control signal is derived from Split module. Once splitting is complete then the output of comparator is connected to update module. Otherwise, the output is connected to split module. This minimizes resource utilization as the distance calculator uses a huge number of multiply and addition operations.

In the split module, depending on whether the data is closer to $\boldsymbol{\mu}_{qL}$ or $\boldsymbol{\mu}_{qR}$, the data is either sent to the left child or right child node selected by the multiplexer. The select line of this multiplexer is connected to the same distance comparator used in mean update module. Then the node number of data and node memory is updated accordingly with the child node number to which the data is sent in the next depth. Once the mean update and splitting are done, then the Node Memory and Data Memory are updated and the new initial means of the child nodes (qR and qL) of node q are calculated in the mean initialization module. The node counter is then incremented and compared with the termination condition. If the termination condition is satisfied (i.e., maximum depth is reached), then the pruning module starts execution. Otherwise, the next node is loaded in the mean update module. Then, split module and mean initialization module are executed for the node and its child nodes respectively. This process is repeated for all the nodes till the maximum depth is reached. In the pruning module, discussed later, the two post-pruning conditions as discussed in Section 2.2 are tested and nodes that satisfy the pruning conditions are set as leaf nodes, and their children nodes (if any) are pruned or removed from Node Memory.

2.3.1.2 Distance calculator

The parallel hardware architecture for the distance calculator module for calculating distance of data-instance $\mathbf{x}^k$ with left mean $\boldsymbol{\mu}_{qL}$ (d_{qL}^k) is shown in Fig. 2.5. The distance calculator is parallelized to speed up the execution. For m dimensional data, this module uses m subtractors and m multipliers to compute $(\mu_{qL}[l] - x^k[l])^2$ where $l = 0, ..(m-1)$, for m dimensions in parallel. Then, the time taken to compute the final distance is only $t_{sub} + t_{mul} + t_{add}$ units of time. The serial operation would result in use of only one subtractor and one multiplier. While, the time complexity will increase to

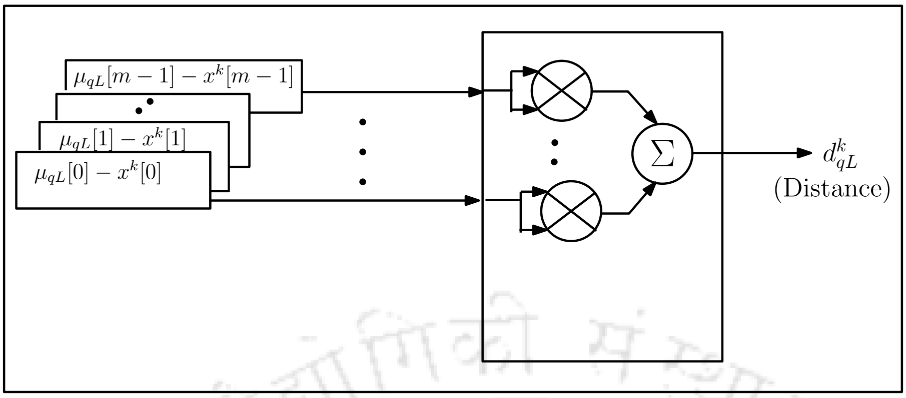

Figure 2.5: Hardware architecture for distance calculator

$m \times (t_{sub} + t_{mul}) + t_{add}$. Thus, this parallel architecture avoids the additional time needed to compute m dimensions sequentially. Similar hardware is used for calculating the distance of data-instance with the right mean in parallel.

2.3.1.3 Pruning module

As shown in Fig. 2.6, in the pruning module, two comparators are used parallelly to compare the purity of corresponding node data with purity threshold η_p and the total node data n_q with data threshold ρ_c. These thresholds are set based on simulation studies discussed in Section 2.4. The output of both comparator is 1 only if the node satisfies the condition to be declared as leaf node according to (5.11), otherwise, the output is 0. The output of the comparator is then fed into the OR gate. Thus, if at least one condition is satisfied, the node qualifies as leaf node. The output of the OR gate is connected to a multiplexer which sets the node as either a split node (if output of OR gate is 0) or leaf node (if output of OR gate is 1) in tree memory and accordingly updates the children node.

After all the nodes are processed and the termination condition is satisfied then the pruning module is executed to check the split condition for decision nodes. The decision nodes which do not satisfy the split condition are labeled as a leaf node and their child nodes are deleted from node memory. Then, the tree structure is updated and stored in the memory. Once this split condition check is completed for all decision nodes, then the training is complete and the hardware is ready for classification.

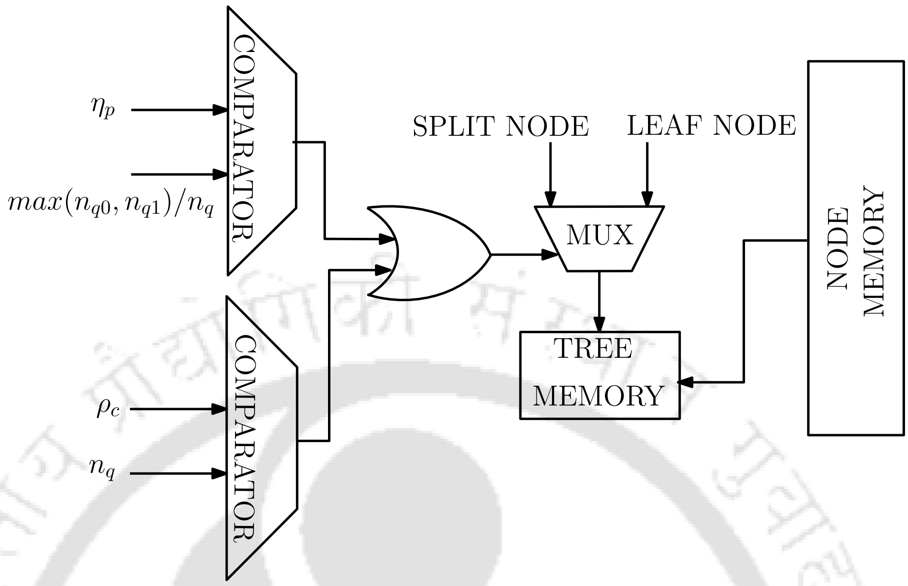

Figure 2.6: Hardware architecture for pruning module

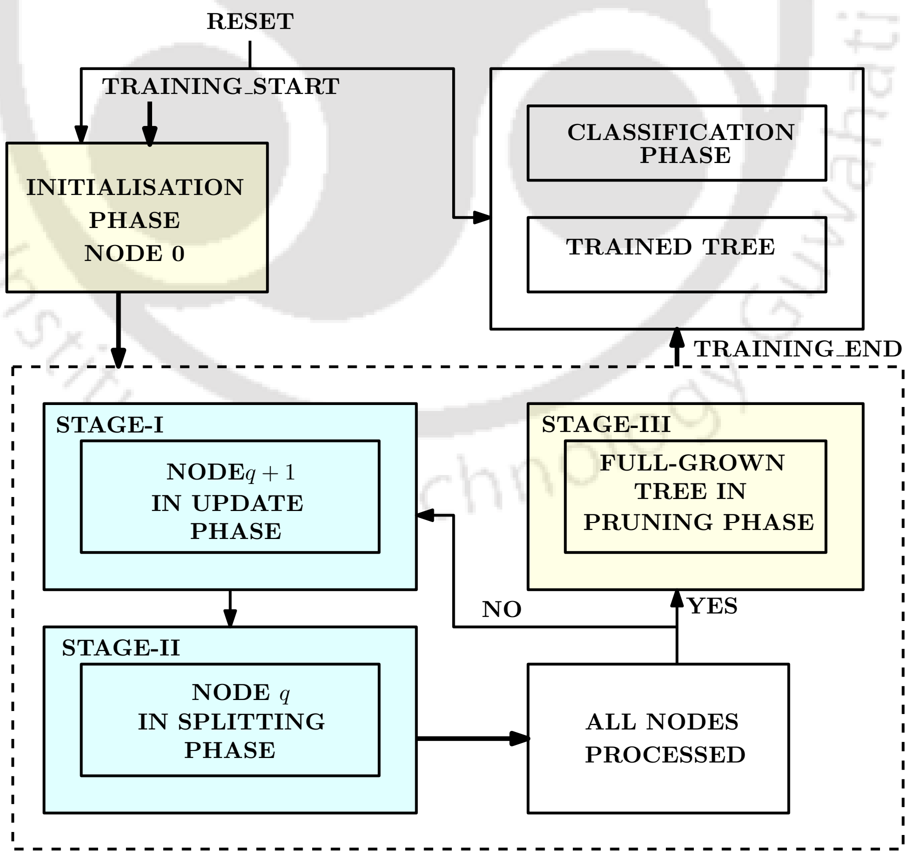

Figure 2.7: Block diagram showing the flow of operations involved in the proposed mixed pipeline and parallel architecture for TMDT

2.3.2 Mixed pipeline and parallel architecture

2.3.2.1 Overall architecture

The training process flow of the TMDT algorithm with pipelining of nodes is illustrated in Fig. 2.7. The design passes through three states – standby, training, and classification respectively. In standby mode, the power supply to the FPGA board is turned ON and an active high RESET signal is asserted 1, which initializes the hardware registers to 0. Once the RESET is set to 0, the TRAINING_START signal is set to 1 and the training is enabled. The completion of training sets the TRAINING_END signal to 0 which activates the classification. The board performs classification as long as the RESET signal is 0. Whenever the RESET is set to 1, it erases all previous data and makes the hardware register contents 0. After that, the training for new data starts by again asserting the TRAINING_START signal to 1 and RESET to 0. The classification starts after the completion of the training process for new data. Thus, this hardware can be re-trained on new data by switching the RESET signal between 0 (training or classification) and 1 (standby). So, hardware obtained by mapping this flow diagram into architecture is very cost-effective as it can be re-trained anytime to suit different applications. It is more convenient to implement such hardware on FPGA due to its reconfigurability.

Once initialization is complete, then node 0 is executed in stage-I and then enters stage-II through pipelined registers. When node 0 is processed in stage-II, the initial mean calculation of its child nodes, i.e., nodes 1 and 2 takes place in parallel while splitting the data of node 0. After the root node is executed, node 1 enters stage-I and then goes to stage-II. As node 1 enters stage-II, stage-I resources are released. So, node 2 is fed to stage-I while node 1 is executed in stage-II. While node 1 is in stage-II, the initial means of child nodes of node 1, i.e., nodes 3 and 4 are calculated in parallel and stored in the node register. So, when node 2 reaches stage-II, node 3 is fed to stage-I for the mean update as the initial mean of node 3 is already available in the node register. So, the initial mean of $(q+1)^{th}$ node is already available while q^{th} node is in stage-II where $q = 1, ..14$. Therefore, this pipelining of the mean update phase in stage-I and the splitting phase in stage-II is done to optimize resource utilization while increasing the throughput. Once the tree is grown to the maximum depth, then stage-III is executed to prune the tree. In this stage, the impurity and total data entering a node stored in the node register in the splitting phase is compared with the pruning condition. Once all the split nodes are checked then the TRAINING_END signal is reset to 0 which indicates the completion of training. The hardware enters the classification phase. Here, the data to be classified is assigned a

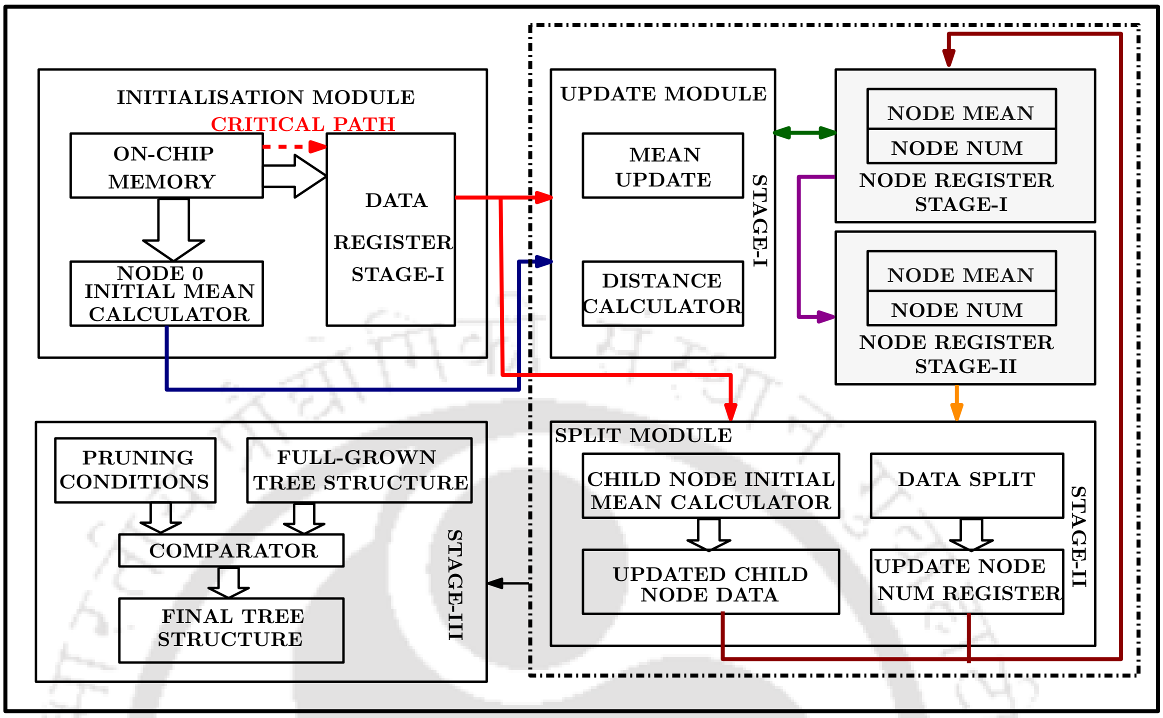

Figure 2.8: Outline of proposed mixed pipeline and parallel architecture for TMDT Algorithm

label after classification is done according to the decision rule derived from the trained DT.

2.3.2.2 Description of hardware modules

The hardware structure of the TMDT algorithm obtained by mapping the flow diagram in Fig. 2.7 onto architecture is shown in Fig. 2.8. The details of several modules involved in Fig. 2.8 are explained below.

Initialisation module: In the initialization module, the data is first loaded into the data register (DATA REGISTER STAGE-I) from the on-chip memory which avoids the time delay in accessing the memory. The initial mean of the root node is calculated in parallel while the same data set is loaded into the data register array from on-chip DRAM. This initialization module is executed only once. Then, the data register array and the initial mean calculated are used to update the root node mean in the update module.

Update module: The update module consists of mean update and distance calculator as the left and right means are updated based on the shortest distance from the instances of the data matrix. After every iteration, the node mean register is accessed to store the updated mean calculated in pipeline stage-I. Once all data instances of the node are processed in the update module, then the node is sent to the split module.

Split module: In the split module, the node data is split into two parts depending on the distance from the left and right mean as discussed in Section 2.2. In the split module, whenever a data vector is assigned to a particular child node, the node number of that data vector is updated in NODE NUM register and the vector is added to the initial mean of that child node. This parallel execution saves the time that would have been wasted otherwise in calculating the initial mean of the child node separately. Once the splitting is over for all data and the node register array is updated, then the next node starts processing.

After all nodes are processed in stage-I and II, then stage-III is executed. In this stage, all the split nodes of the fully grown tree obtained from the previous stage are tested for split condition as shown in Fig. 2.6. The split nodes which do not satisfy the splitting condition are declared as leaf nodes and their child nodes are removed from memory. Then, once all the nodes are checked, the pruned tree structure is stored in the memory.

2.3.2.3 Hardware architectures of stage-I and II

The detailed architecture for stage-I and stage-II is demonstrated in Fig. 2.9. This architecture uses two distance calculator units running in parallel to calculate the distance between the data and the left and right mean in stage-I. The two distances are then compared and the mean with smaller distance from the data is selected by the mean selector. The selected mean is then updated and stored in the node register. In this way, all the data belonging to that node is processed in stage-I. Then the final updated mean and node number from the node register of stage-I is sent to the node register of stage-II through a latch and stage-I starts processing the next node. The clock period of the latch is set a little higher than the latency of stage II so that the data is not overwritten. In stage-II, again two distance calculators are used to calculate the distance between μ_{qL}, μ_{qR}, and $\mathbf{x}^k$ stored in the updated node. The data is designated with the node number of the child node whose mean is closest to the data. This node number is stored in the NODE NUM register. In this way, all data instances in the node are split into two parts and sent to one of the two child nodes. At the same time, the data going to the child node is summed up to calculate the initial mean of that child node which is stored in the updated initial mean register. Once all the nodes have been processed in stage-I and II, then stage-III is executed.

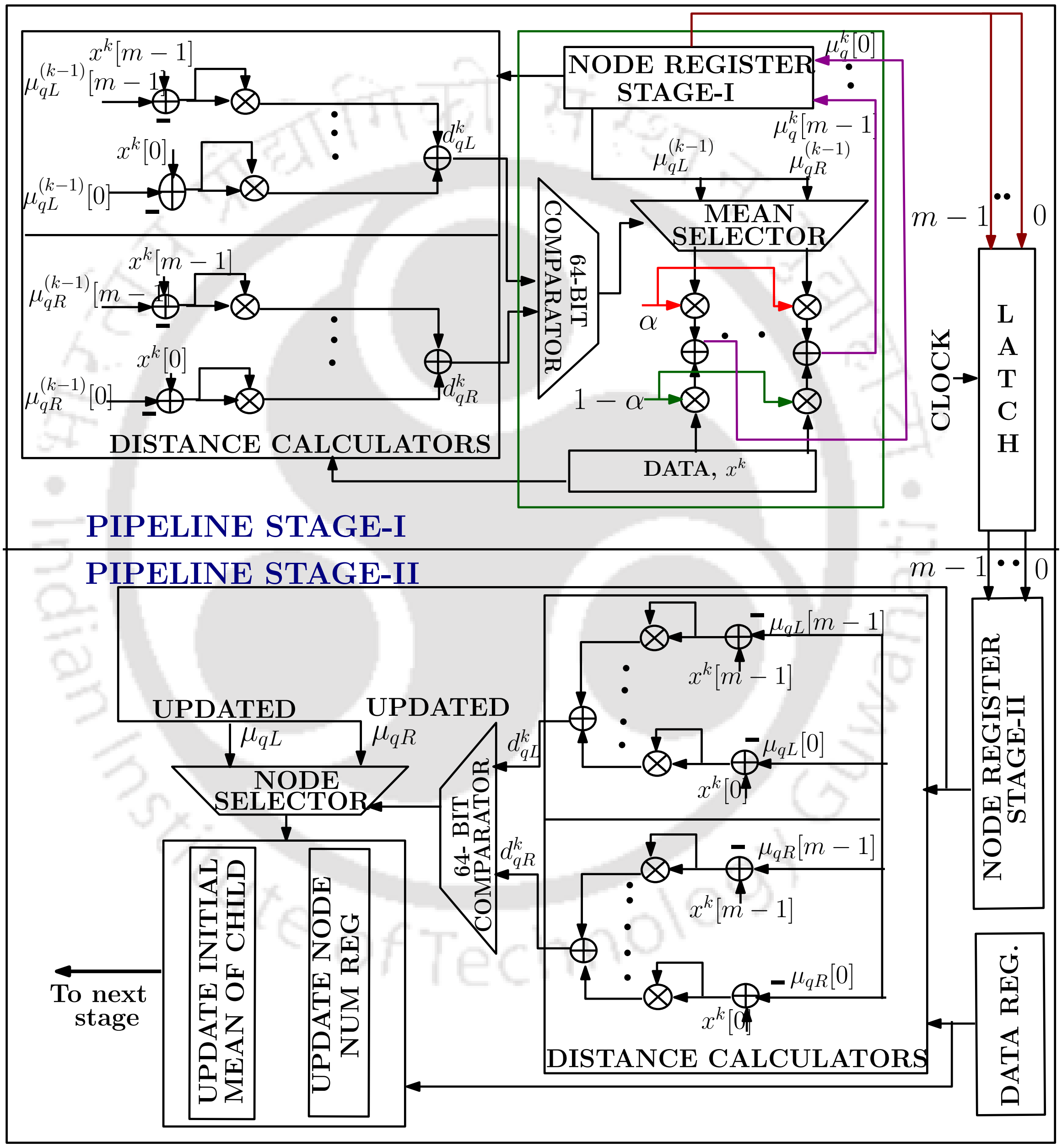

Figure 2.9: Detailed architecture for pipelined stage-I and II

TH-3177_176102101

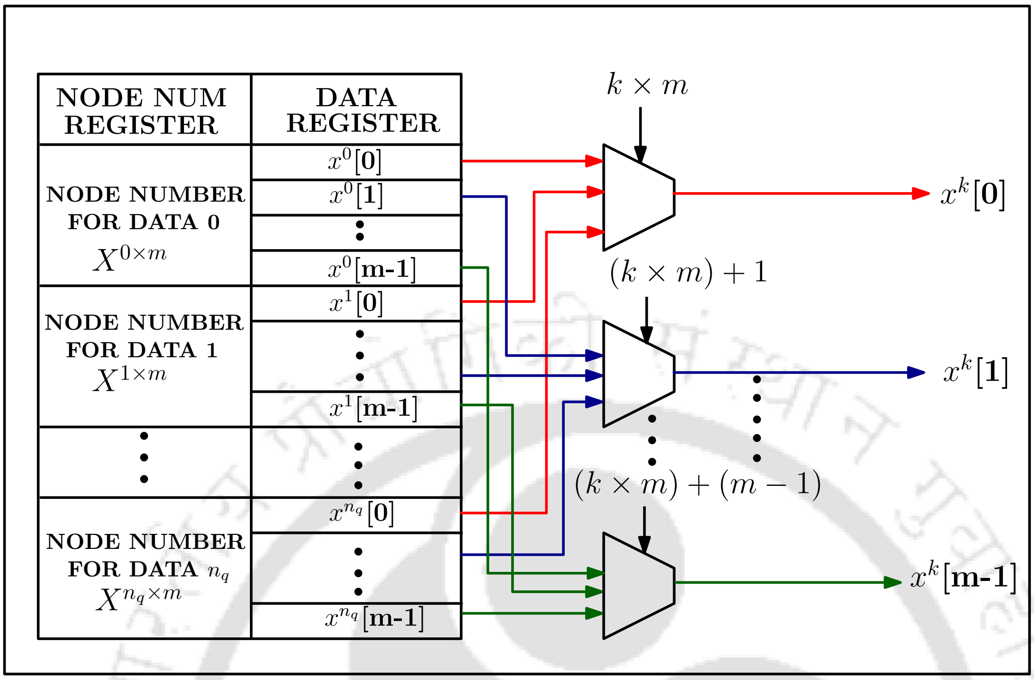

Figure 2.10: Hardware architecture to access data memory in parallel

2.3.2.4 Parallel data memory access

The data register access is also parallelized to further speed up the execution as shown in Fig. 2.10. The register named as NODE NUM register stores the node number of all the data in order to identify the node to which the data belongs. If the node number of k^{th} data does not match with the current node number then, the node number of $(k+1)^{th}$ data is checked otherwise, k^{th} data is selected. This way all the data of the current node is processed. To access all the m dimensions of data in parallel, m number of multiplexers are used. The current data location $k \times m$ is used as the select line to identify 0^{th} dimension of k^{th} data. Similarly, $(k \times m) + (m-1)$ selects m^{th} dimension of k^{th} data. This allows to access m-dimensional data in a single clock cycle thus saving $(m-1)$ clock cycles. Each m-dimensional data uses m memory locations of the data register array. So for $n_q \times m$ data, $n_q \times m$ locations of Node Num register and Data Register are used to store the node number and data respectively, where each location is 32-bit. As data is stored sequentially, if one data is at location 1 then the next data and its node number are at location $1 + m$ for m dimensional data.

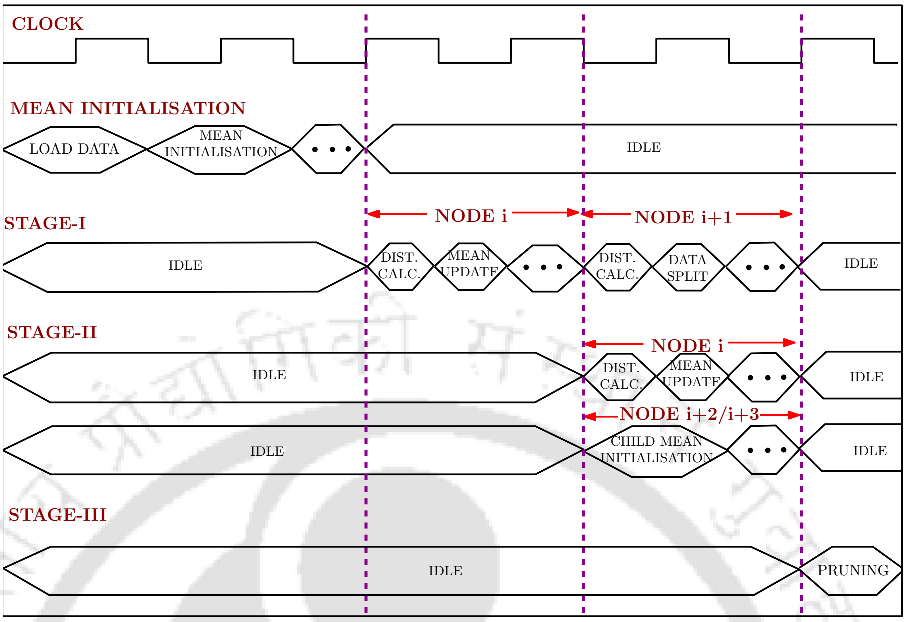

Figure 2.11: Timing diagram showing the clock cycle consumption for different stages

2.3.2.5 Timing diagram and clock cycle consumption

The timing diagram for the three stages including the mean initialisation module of mixed design is shown in Fig. 2.11. At first, the mean initialization (2 clock cycles for each data-instance) for the entire data is done (only for node 0). Then, the data is processed in Stage-I, where mean update according to the shortest distance from mean is done which consumes 1 clock cycle. After mean update, node i is sent for data split in Stage-II (1 clock cycle for each data-instance) and node $i+1$ is processed in Stage-I. The mean initialisation for children nodes of node i (node $i+2$ and $i+3$) is executed in parallel (node 3 onwards). This enables the processing of node $i+2$ in Stage-I once node $i+1$ is sent to Stage-II. Once processing of Stage-I and II are done for all the nodes, then Stage-III (pruning) is executed. Thus, the critical path of this design involves the loading of data from RAM.

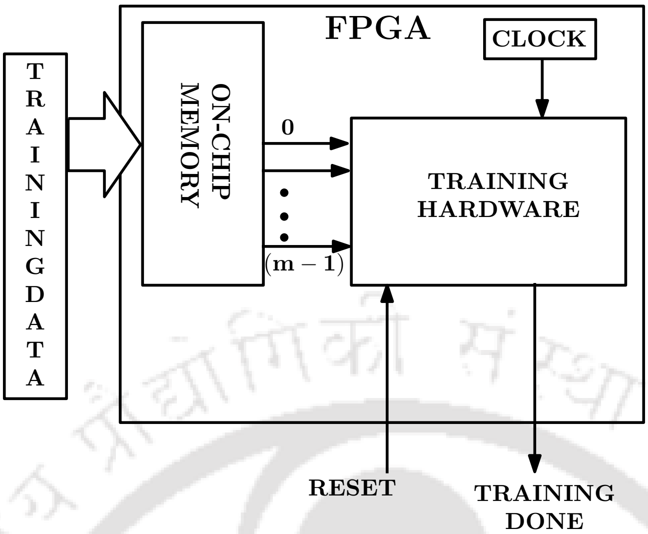

Figure 2.12: Block diagram showing the experimental arrangement

2.4 Results

2.4.1 Experimental set-up and dataset description

The proposed architectures are implemented on Virtex Ultrascale+ XCVU13P-FLGA2577-3-E FPGA board which uses 16 nm technology. The design was implemented using Verilog Hardware Description Language (HDL) in Vivado 2017 and no high-level synthesis tool was used. The TMDT algorithm was implemented on both FPGA and CPU platforms for comparison. The algorithm was implemented on an Intel core i5 processor for software comparison which runs at 3.2 GHz. The maximum operating frequency of serial architectures is found to be 62 MHz due to the critical path of 16 ns (which involves the distance computation). While, the maximum operating frequency of mixed architectures is found to be 71 MHz due to the critical path of 14 ns. This critical path is mainly due to Block Random Access Memory (BRAM) access operation. This hardware is able to support 32-bit training data. This board has 500 megabyte (MB) of on-chip memory allowing a considerably large amount of 32-bit training datasets of almost 125 million data points. This large on-chip memory allows efficient latency data access. All hardware results are post-implementation results. The block diagram for the experimental setup is shown in Fig. 2.12. The data is copied into this on-chip memory of the FPGA board from the computer prior to the start of the training process. This enabled independent training by connecting the FPGA board directly to the power supply once the design is implemented on board. The maximum depth was set as $\zeta_{max} = 4$ and thus, maximum number of

Table 2.1: Table recording the performance (F1-score and accuracy) and comparison of training latency between software (Python and C) and the two proposed hardware architectures for all datasets

Datasets	**Data-size**	**F1-score**	**Acc.(%)**	**Latency**(in ms)			
				Python	**C**	**Serial**	**Mixed parallel & pipelined**
Mamm.	793×6	0.62	67	156	8.76	0.494	0.29
Occ.	10152×6	0.69	87	1111	100.93	6.4	3.76
Skin	38000×4	0.93	93	6000	224.5	24	14.47

nodes was $2^{\zeta_{max}} - 1 = (2^4 - 1) = 15$ (considering the root node or node 0 at depth 1). As discussed in Section 2.2, for these datasets after experimentation, the update factor was fixed at $\alpha = 0.875$ and the cardinality threshold was set as $\rho_c = 20$. The impurity threshold was set as $\eta_p = 0.95$.

The balanced binary datasets, viz., skin, occupancy, and mammography were used for testing [40]. The skin dataset has $38k$ data points each having 3 features ($m = 3$), i.e., R, G, B pixel and a binary value corresponding to skin or non-skin data. The occupancy has $12k$ data points each having 5 features ($m = 5$), CO_2, relative humidity, temperature, light, humidity ratio, and whether the room is occupied. The mammography dataset consists of 961 data points each having 5 attributes ($m = 5$), size, shape, margin, density, severity, and whether the tumor is benign or malignant. In each case, the training dataset is constructed using 80% of total data and the remaining 20% is used for testing. Thus, these hardwares are capable of supporting up to 5 features per data instance. The reported BRAM utilization is for a maximum of 38k training data instances which is extendable subject to the maximum BRAM available on the FPGA board used.

2.4.2 Experimental results

The distribution of data for the skin dataset from the parent node to the child node till depth 3 is shown in Fig. 2.13. As shown in the graph, the purity of the node increases with an increase in depth and results in almost pure nodes in depth 4 (having leaf nodes). The bar plot of comparison of accuracy and time consumption for TMDT and C4.5, when implemented in the Python platform for the 3 datasets, is shown in Fig. 2.14. It is observed that TMDT has much lower latency as compared to C4.5 for large-sized datasets. While the accuracy of TMDT is slightly lower as compared to C4.5 for small-sized datasets, it becomes almost comparable for large-sized datasets. Thus, TMDT gives comparable performance while running much faster than C4.5 for large-sized datasets which is the usual scenario in ML algorithms.

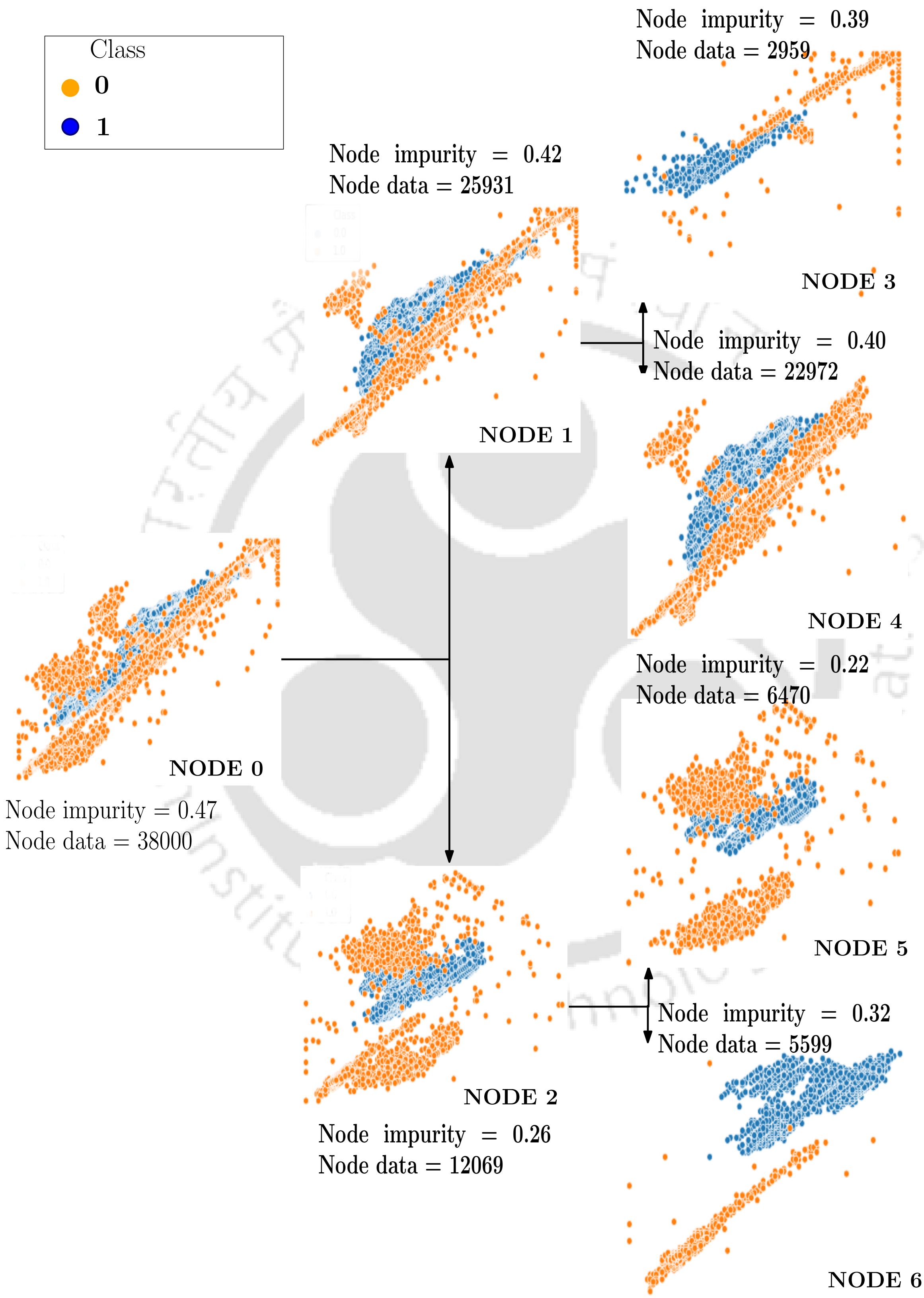

Figure 2.13: Node data separation from the parent node to children nodes in TMDT is visualized above for the Skin dataset for the split nodes. The data overlap is gradually reduced from the parent node to the children nodes. The node impurity as indicated in the figure is observed to drop gradually with an increase in depth

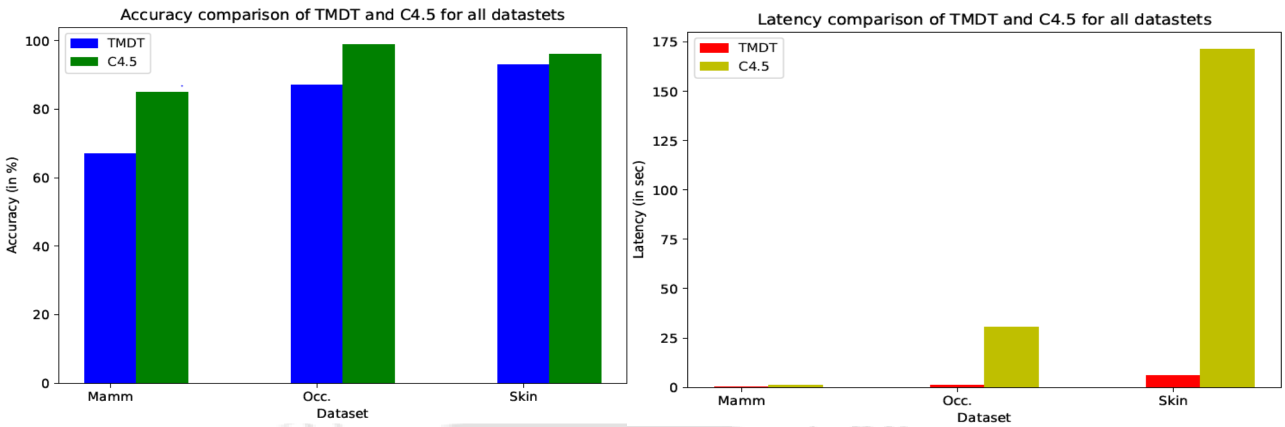

Figure 2.14: Bar plots showing the comparison of latency and accuracy for all datasets for C4.5 and TMDT algorithm

Table 2.2: FPGA training latency comparison of proposed training hardwares with existing training hardware

Design	Latency for 1k data (in ms)	Latency for 10k data (in ms)	Training Algorithm	No. of FPGAs	No. of Features	Supported Classes
[20]	260	360	CART	4	32	16
TMDT serial	0.49	6.40	TMDT	1	5	2
TMDT mixed	0.29	3.76	TMDT	1	5	2

The efficiency of ML algorithms is measured by F1-score and accuracy. The F1-score is computed using equation (2.7) [15]. Here, TP and TN are the number of correctly classified data instances with label 1 and 0 respectively. FP indicates the number of negative (label 0) instances classified as positive (label 1). Similarly, FN indicates the number of positive (label 1) instances classified as negative (label 0). The binary confusion matrix is shown in Fig. 2.15. The training latency of proposed TMDT training accelerators are compared with existing CART training accelerator implemented on HC server comprising of 5 FPGAs in Table 2.2. It is observed that the proposed hardwares are faster than the multiple FPGA configuration. This speed-up is achieved due to reduced complexity of TMDT as compared to CART. Though, the proposed hardware supports lesser number of features as compared to the existing hardware, the number of features will not affect the execution time as all features are processed in parallel. The proposed hardwares can be modified to accommodate higher number of features which will increase resource consumption.

$$F1 - score = \frac{2rp}{r+p}; \qquad r = \frac{TP}{TP+FP}; \qquad p = \frac{TP}{TP+FN}; \tag{2.7}$$

		Predicted Class	
True Class		P	N
	P	TP	FN
	N	FP	TN

Figure 2.15: Confusion matrix used for calculation of F1-score is shown.

F1-score, accuracy, and latency of C, Python, and both hardware implementations for all three data sets have been listed in Table 2.1. The F1-score and accuracy for the largest dataset skin are found to be 0.93 and 93% respectively. Thus, it is observed that the efficiency of the design increases with the dataset size. The software implementations are found to have higher training latency as compared to the hardware implementation. The fastest implementation on C took 224 ms whereas serial and mixed implementation took only about 24 and 16.54 ms, respectively. Thus, for the worst-case scenario, the serial hardware is found to be 250× and 10× faster than Python and C implementations, respectively. While, the mixed implementation is found to be 400× and 14× faster than Python and C implementations, respectively. Therefore, it is observed that the hardware implementation accelerates the training by a huge factor. The hardware utilization comparison of the proposed training architecture with conventional and optimized k-means classification architecture is discussed in Table 2.3. The serial training accelerator utilizes only 3.4k flip-flop (FF) and 48 BRAM for 38k data instances as compared to the optimized k-means classification hardware proposed in [33] which utilized 24k FF and 0.24k BRAM for only 16.384k data-instances. However, the LUT and Digital Signal Processor (DSP) consumption is a little higher in this design due to higher computational complexities as compared to classification. Thus, dynamic power consumption for this design is 1.5W only. The break up of power consumption by different components for the serial implementation is shown in Fig. 2.16. The proposed mixed design is shown to reduce the BRAM consumption for 38k data instances as compared to [33] for only 16.384k data instances. The mixed architecture consumes 1.3 W of dynamic power. This improvement in power consumption is achieved due to the use of efficient arithmetic units (distance calculators and comparators) instead of using DSPs. The power

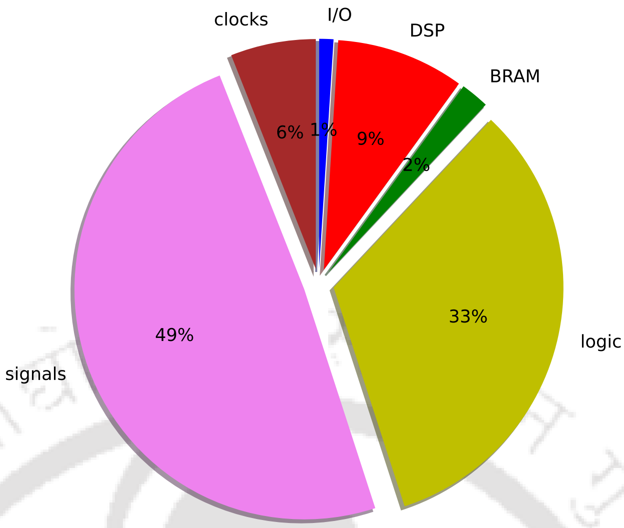

Figure 2.16: Pie-chart showing the power consumption for the TMDT serial architecture.

Table 2.3: Table recording resource utilization comparison of the proposed hardwares with existing designs

Design	**LUT** (in K)	**FF** (in K)	**BRAM** (in K)	**DSP** (in K)	**Data -points** (in K)	**FPGA Board used**	**Algorithm implemented**
k-means	108	54	0.100	1.062	16.384	Virtex 7	Conventional k-means classification
Optimised k-means [33]	14	24	0.240	0.186	16.384	Virtex 7	Optimised k-means classification
Proposed serial arch.	297.91	3.42	0.048	0.20	38	Virtex Ultrascale+	TMDT training
Proposed mixed parallel & pipelined arch.	298.96	3.57	0.187	0.06	38	Virtex Ultrascale+	TMDT training

consumption by different components for mixed architecture is shown in Fig. 2.17.

2.5 Discussions

In this chapter, we proposed a low complexity and medium accuracy TMDT algorithm. The FPGA implementation of TMDT in both serial and mixed parallel and the pipelined modes was also proposed. The maximum working frequency for both designs was found to be 62 MHz. The proposed hardwares have been tested on three binary data sets each having variable dimensions and size. So, it can be used for a wide variety of applications by re-training the hardware on the target application dataset. The TMDT algorithm runs at least 28× faster due to reduced complexity as compared to the widely used conventional C4.5 algorithm. The classification accuracy is observed to increase with the increase in training dataset size for TMDT. From FPGA implementation results, the serial FPGA

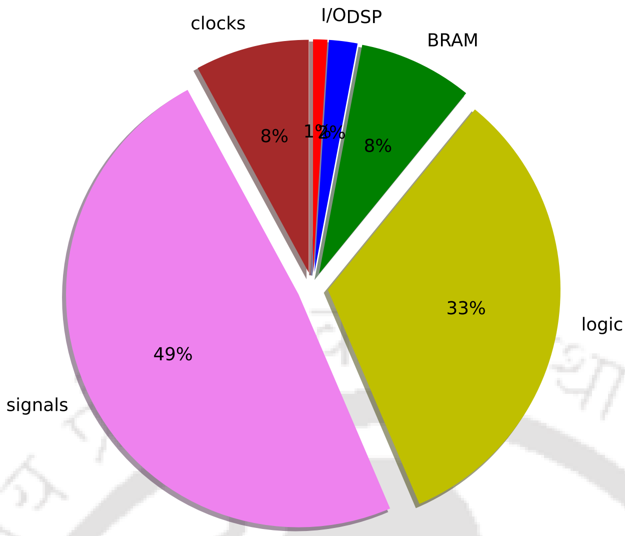

Figure 2.17: Pie-chart showing the power consumption for the TMDT mixed architecture.

implementation is shown to achieve at least 10× speed-up as compared to software implementations for the same datasets. This hardware is observed to have FF and BRAM consumption even less than the existing optimized classification accelerator due to the use of a low-complexity algorithm. In the second hardware, a mixed parallel and pipelined architecture for TMDT training were designed. Here, node level pipelining was used. This training accelerator is found to speed up the training process by at least 14× as compared to software platforms. The training in FPGA was completed in 16.54×10^{-3}s whereas software training required 224×10^{-3}s for the same number of data instances. It reduced the hardware usage even further as compared to the existing classification accelerator.

In these designs, while processing a node, all the training data instances were loaded in the same sequence as stored in the memory. From depth 2 onward, the node number associated with each data instance was matched to the node currently being updated. The data instances whose node number did not match with the node currently being executed were not processed further. This led to clock cycle wastage and hence, resulted in increased latency. While splitting the data in a node, it is assigned to one node in the next depth. So, to address this clock cycle wastage, the data can be processed depth-wise. At a certain tree depth, if the node module is selected according to the node number of current data, then a data instance will be loaded only once in each depth and clock cycle wastage can be prevented. In the designs proposed in this chapter, the training data size was limited by the size of the on-chip memory. This problem can be addressed by breaking the data matrix into batches which will reduce the memory size. Also, the TMDT can be modified to include multi-class classification instead of only the binary classification proposed in this chapter. These issues are addressed in the next chapter.

3

Two Means Decision Tree: Batch-mode and multi-class implementation

- R. Choudhury, S. R. Ahamed, and P. Guha, "FPGA Implementation of Batch-mode Depth-pipelined Two Means Decision Tree", in IEEE Embedded Systems Letters, 2022.

Contents

Overview

This chapter proposes the depth pipelined batch-mode multi-class implementation of Two Means Decision Tree (TMDT) on FPGA. This design divides the training data into batches to minimize the memory requirement. These batches are loaded one-by-one into the memory to train the Decision Tree (DT) which reduces the memory size. Further, the architecture implements depth pipelining. Here, two batches are processed simultaneously in two pipelines where each pipeline handles nodes from two consecutive tree depths. It was observed that the hardware implementation on FPGA accelerated the training by at least 27× as compared to the software counterparts. This training accelerator is also observed to be faster than the existing training accelerator. Moreover, this hardware has less resource consumption as compared to existing classification and training architectures.

3.1 Problem formulation

The improved performance of TMDT is observed with an increase in training dataset size. So, in this chapter, to accommodate large-size datasets and hence, to increase accuracy, batch-mode implementation of TMDT is proposed. In the proposed design, the training data is divided into small batches and one batch at a time is loaded into the chip memory. Here, the multi-class training of TMDT is implemented on FPGA. In this implementation of TMDT, depth-pipelining is used in place of node-pipelining in order to further improve resource usage and optimize clock cycle consumption. The hardware is divided into two pipelines and each pipeline consists of two depths. This hardware implements a DT of maximum depth 5 where depth 5 consists of only leaf nodes. So, split nodes till depth 4 are divided into two pipelines each consisting of two depths where the root node (node 0) is present at depth 1. The multi-class implementation makes it suitable for day-to-day ML applications where most of the scenarios are multi-class. Moreover, the batch-mode implementation makes it more resource-efficient as the memory size is reduced to a great extent which in turn reduces the memory access latency thus, accelerating the hardware further. It also removes the limitation on on-chip memory thus, removing the hard line on the size of training data. Further, due to depth-wise processing entire dataset is loaded only once in each depth. Then, the node number is used to select the node in that depth to which the data belongs. This reduces the latency of the design.

3.2 Batch-mode TMDT algorithm for multi-class data

The TMDT performs divisive clustering of the input data. A TMDT of maximum depth ζ_{max} consists of split nodes till depth $(\zeta_{max} - 1)$ and leaf nodes at depth ζ_{max}. Each split node of TMDT performs two-means clustering on its input data and stores the left (μ_{qL}) and right mean (μ_{qR}) as the decision parameters. An input data instance closer to μ_{qL} is routed to the left child node and to the right one, otherwise.

This work trains the TMDT in batch mode as shown in Fig. 3.1. The training dataset $\mathbf{X} = \{\mathbf{x}^k : y(\mathbf{x}^k) = c; \mathbf{x}^k \in \mathbb{R}^m; \}$ where $c = 0 \dots (C-1)$ and $k = 0, \dots (n-1)$ containing n instances (with C different category labels) is divided into H disjoint batches. Each batch $\mathbf{X}_i$ $(\mathbf{X} = \bigcup_{i=1}^{H} \mathbf{X}_i)$ contains b training instances and $H = \left\lfloor \frac{n}{b} \right\rfloor$ (where H is total number of batches and b is the batch-size). The first batch $\mathbf{X}_1$ is used to initialise the fully grown TMDT $\mathcal{T}_1$ of maximum depth ζ_{max}. The split and leaf node parameters of the TMDT $\mathcal{T}_{i-1}$ are updated with the instances from $\mathbf{X}_i$ to obtain $\mathcal{T}_i$. The TMDT $\mathcal{T}_H$ is considered to be the one induced on the entire dataset $\mathbf{X}$.

For the first batch $\mathbf{X}_1$, the split node parameters are learned in two stages. First, the data-subset $\mathbf{X}_1^{(q)}$ input to node q is randomly divided into two halves and their centroids initialize the left (μ_{qL}) and right (μ_{qR}) mean vectors. Second, data instances $\mathbf{x}^k \in \mathbf{X}_1^{(q)}$ are used to update the relatively closer mean vector. Thus, μ_{qL} is updated according to (3.1) if $||\mathbf{x}^k - \mu_{qL}||_2 < ||\mathbf{x}^k - \mu_{qR}||_2$ else μ_{qR}. The update equation (2.1) discussed in the previous chapter has been modified to reduce the number of multiplications as follows.

$$\mu_{qL} \leftarrow \mu_{qL} + \alpha(\mathbf{x}^k - \mu_{qL}) \tag{3.1}$$

Similarly, μ_{qR} is updated by instances closer to it. For training batches $\mathbf{X}_i$ $(i > 1)$, the split node parameters of TMDT $\mathcal{T}_i$ $(i > 1)$ are only updated using (3.1).

The mean update process (3.1) is susceptible to bias induced by the order of presentation of input data instances. Thus, the data instances are randomly selected to form H number of different batches. Each epoch process all the H batches. So, to improve accuracy, the contents of these batches are again shuffled before presenting them to the next epoch. The contents of all batches are shuffled for N times to form N epochs. Therefore, a total of $N \times H$ different batches are used to update the TMDT.

The batches of input instances $\mathbf{X}_i$ $(i = 1, \dots H)$ are again passed through $\mathcal{T}$ for post-pruning of the fully grown TMDT. For each batch, an instance count is maintained for each split node and leaf node. Let, $n_c^{(qS)}$ and $n_c^{(qT)}$ be the respective number of instances of category c $(c = 0, \dots C-1)$ in

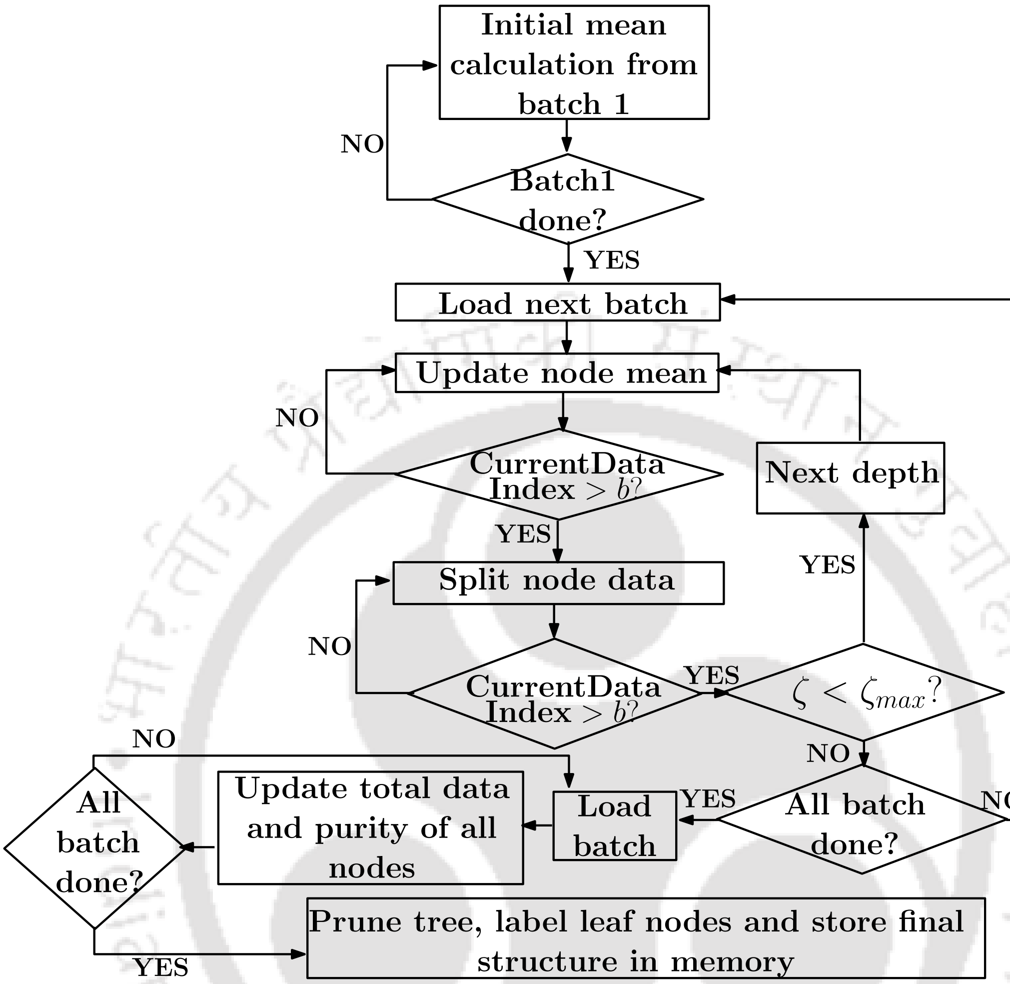

Figure 3.1: The flow diagram of batch-mode TMDT algorithm.

split node qS and leaf node qT. The node qS is set to a leaf node if either one or both the following conditions (3.2) are satisfied.

$$\begin{aligned} &\text{Node Input Deficiency} : \sum_{c=0}^{C-1} n_c^{(qS)} \ < \ \rho_c \\ &\text{Node Input Purity} : max_c \left\{ \frac{n_c^{(qS)}}{\sum_{v=0}^{C-1} n_v^{(qS)}} \right\} \ \geq \ \eta_p \end{aligned} \tag{3.2}$$

The children nodes of such split nodes are removed resulting in the pruning of TMDT $\mathcal{T}$. After pruning, the instance counts of all leaf nodes are used to finalize their category labels. The category label C_{qT} of leaf node qT is set as follows.

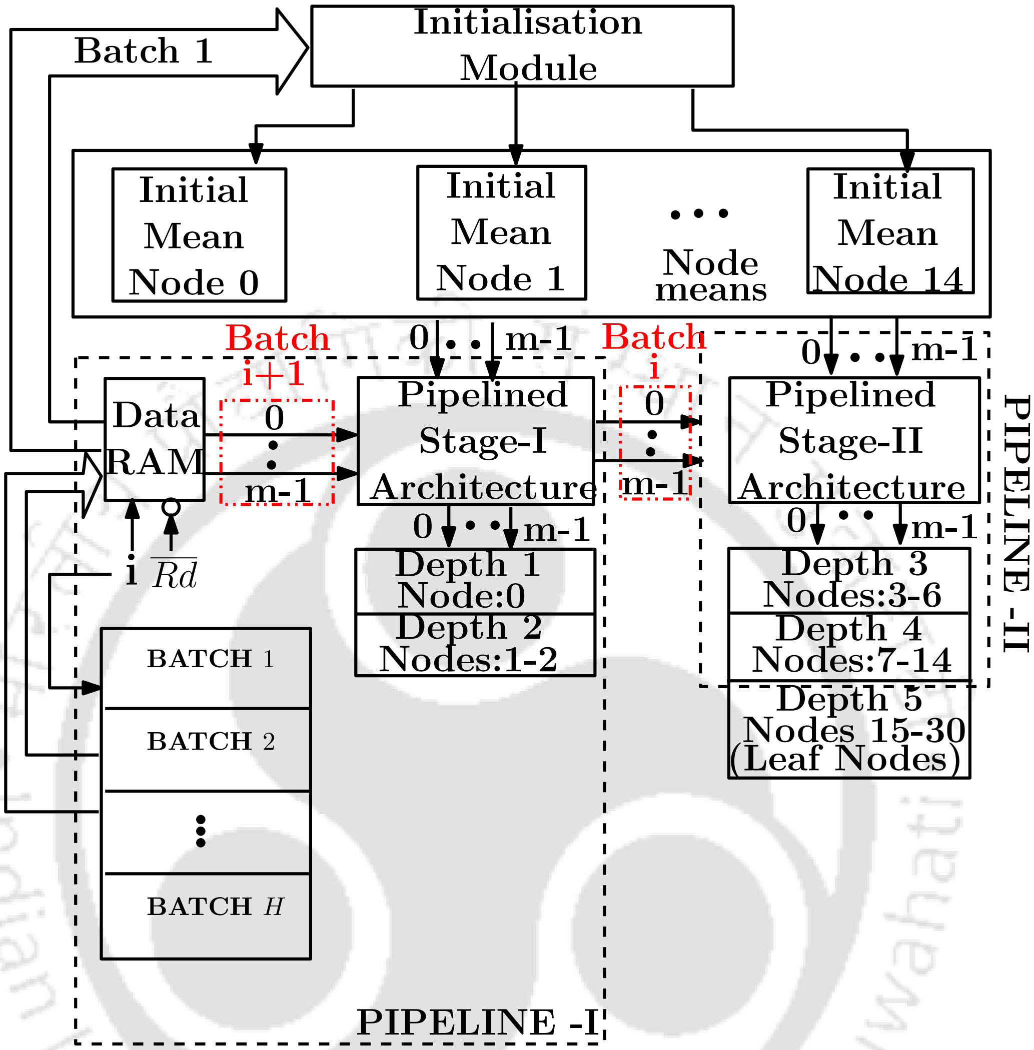

Figure 3.2: Outline of hardware showing flow of data batches between Pipeline-I and II. After initialisation is complete, pipeline stage-I and II are executed, simultaneously.

$$C_{qT} = \underset{c=0,\ldots C-1}{argmax}\, n_c^{(qT)}. \tag{3.3}$$

3.3 Proposed hardware architecture

3.3.1 Flow of pipeline

The flow of batches between the pipelined stages is shown in Fig. 3.2. The initial means of all split node are calculated from the first batch of data where b is the batch-size. This b is chosen in the power of 2 so that the averaging in mean initialisation is replaced with shift (without affecting accuracy). The update of RAM data is done by updating the address. To load a data from different batch, a variable i has been used which is incremented to load the next batch file from the memory. The memory access latency decreases with decrease in memory-size. So, breaking data into mini batches reduces the memory size. This in turn decreases latency. Further, while a data instance is processed in mean update phase of node 0, the next data instance is loaded into the RAM which saves the time that would otherwise be wasted in loading the data. Let, i^{th} batch of m-dimensional data is fed into the pipeline stage-I hardware through RAM. After this batch is processed in stage-I, it is forwarded to stage-II and next batch, i.e., $(i+1)^{th}$ batch is fed into stage-I. The operations involved in pipeline stage-I architecture are loading of data from RAM and execution of depth $1-2$ nodes. The nodes of depth $3-4$ is executed in stage-II. As data is processed depth-wise, so the maximum number of data-instances processed at a certain depth ζ is fixed at b. The entire batch traverses the tree from depth 1 to ζ_{max} due to post-pruning. During pruning, as explained in the previous section, all the data in a batch travels through the nodes from root node to leaf node and updates the data count, purity and label of all nodes. This process is repeated for all batches. The architectural details of pipelined stages-I and II are described in subsequent sections.

3.3.2 Pipeline stage-I architecture

The detailed architecture of pipeline stage-I starts with node 0 as shown in Fig. 3.3. A data-instance $\mathbf{x}^k$ is sent to the update module from RAM to update the node mean. The node architecture starts with comparing the distance of data with both left and right mean. This DIST_COMP architecture implements euclidean distance computation and comparison. The squaring operation involved in distance computation is realised using a general purpose multiplier whose both inputs are tied to the same register. The proposed multiplier is discussed below.

A general purpose 32-bit multiplier for computing $Y = A \times B$ (where $A = a_{31}a_{30}\ldots a_0$, $B = b_{31}b_{30}\ldots b_0$ and $Y = y_{63}y_{62}\ldots y_0$) is proposed. The architecture is realized using multiplexers as shown in Fig. 3.4. Initially, a 2:1 multiplexer is used to select the output according to the bit value. For example, if b_0 is 0 then 0 is selected otherwise, a_0 bit is selected. Likewise, for 32×32 bit multiplication, 64 bit output $\mathbf{Y}$ is generated. The bit addition to generate y_w (where $w = 0, \ldots, 31$) is again divided into 5 levels of pipeline as shown in Fig. 3.5. In level-I, 16 (32/2) adders are used each of which adds the two partial products obtained from previous stage and 16 sum outputs are generated. In this way, these outputs are propagated to level-II where again 8 adders are used to calculate the sum of these 16 outputs. This process is repeated till it reaches 5^{th} level (level-V) where

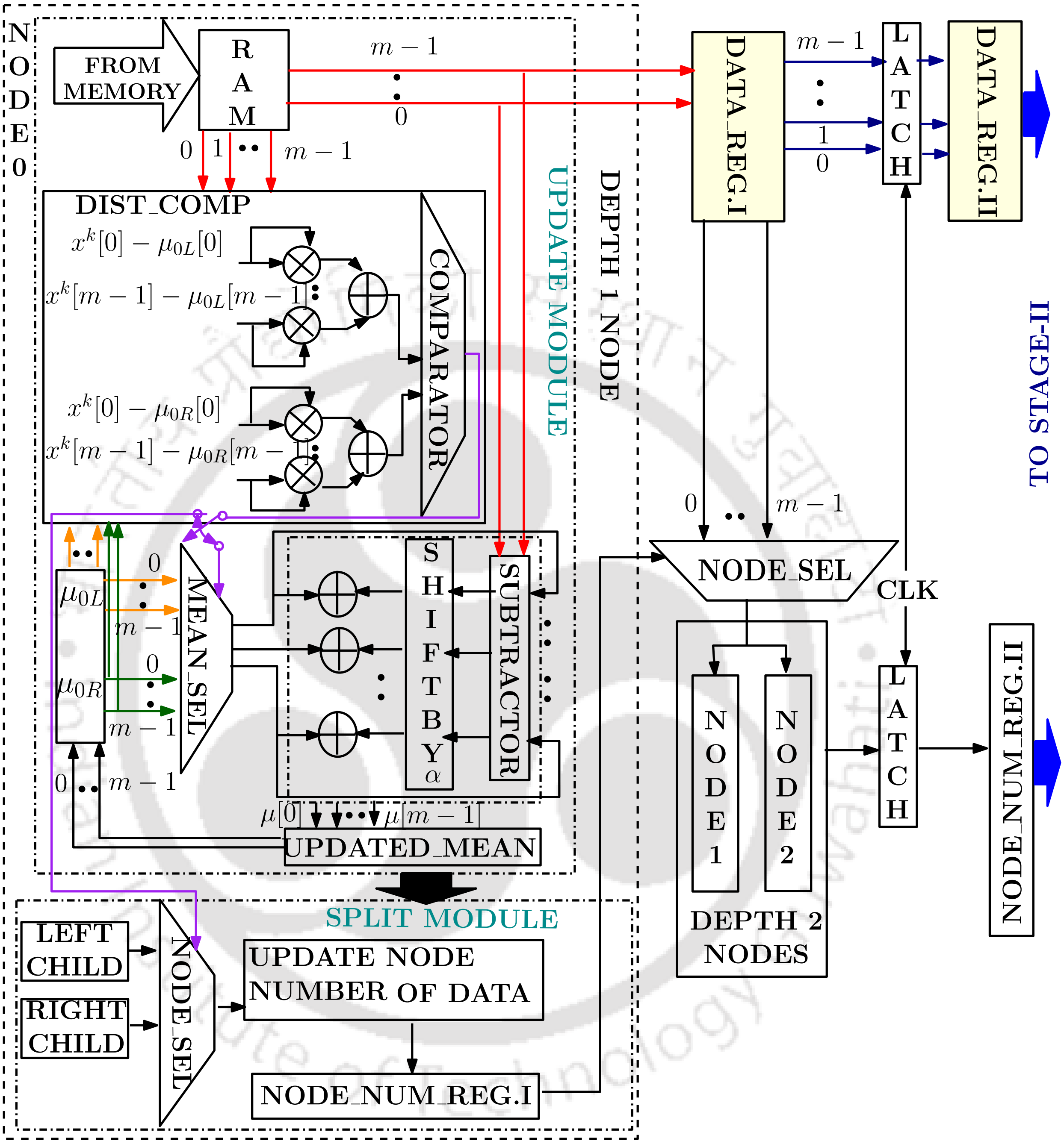

Figure 3.3: Hardware architecture of pipeline stage-I.

one adder calculates and stores the final product in a 64-bit register. So, for 32-bit multiplier, 5 levels of pipelining is required ($log32 = 2^5$). For distance calculation, squaring is required which is a special case of multiplication. So, in this case, both inputs to this multiplier are same, i.e., $A = B$. For m-dimensional data, m such multipliers are used in parallel and their outputs are summed to obtain the final distance. Thus, two distance calculators run in parallel to calculate distance of data with left and right mean. Finally, a 64-bit comparator compares these two distances and outputs a binary number depending on whether $\mathbf{x}^k$ is closer to left (1) or right mean (0). Then, output of distance comparator (DIST_COMP) is used as the select signal of MEAN_SEL to select the closest mean. Then, the data and the selected mean is sent to the UPDATE module and the selected mean is updated by the data according to (3.1). The multiply operation in the update equation is replaced by shifter. As update factor α is constant, so it is represented as SOPT and to multiply a number by α, it is shifted and added $(2^{s_1} + 2^{s_2} + 2^{s_3})$. For this design, $\alpha = 0.875$, so $s_1 = -1, s_2 = -2$ and $s_3 = -3$. This results in optimized resource and clock cycle consumption. For each dimension update, one shifter and one adder module are used. So, for updating m-dimensional mean in parallel, m such modules are used and the final updated mean is stored in UPDATED_MEAN register. The m-dimensional data is accessed from memory in parallel using m multiplexers whose select lines are set as the index of current data as discussed in previous chapter. Likewise, update for entire training data in node 0 is done.

Once update is over, then same data batch is sent to the split module where same DIST_COMP module is re-used for comparing distance of $\mathbf{x}^k$ with μ_{qL} and μ_{qR}. The output of DIST_COMP is used as the select line of NODE_SEL to select the nearest child node. Then, the data-instances are assigned with left or right child node numbers present in depth 2. Accordingly, the NODE_NUM_REG.I is updated with the child node number associated with each data. The entire training data is also stored parallely in DATA_REG.I to be accessed during execution of depth 2 (node 1 & 2) which saves the memory access time. The node module 0 architecture is applicable to all split node modules. Once splitting is complete for depth 1 (node 0) then, for updating the mean of depth 2 nodes, the data stored in DATA_REG.I is used. The node number of the data-instance stored in NODE_NUM_REG.I is fed to the select line of NODE_SEL which chooses the node to be updated. This procedure is repeated for all the data and accordingly, depth 2 node modules (1 or 2) are executed as discussed above. In the selected node module, as described above, MEAN_SELECTOR output is used to select the node mean (μ_{qL} or μ_{qR}) based on shortest distance between the two means and the data currently being executed. Accordingly, the means are updated.

After update is complete for a batch, then that batch is sent to split module. Similarly, entire batch of data is split by assigning it to the closest child node (present in next depth) of current node.

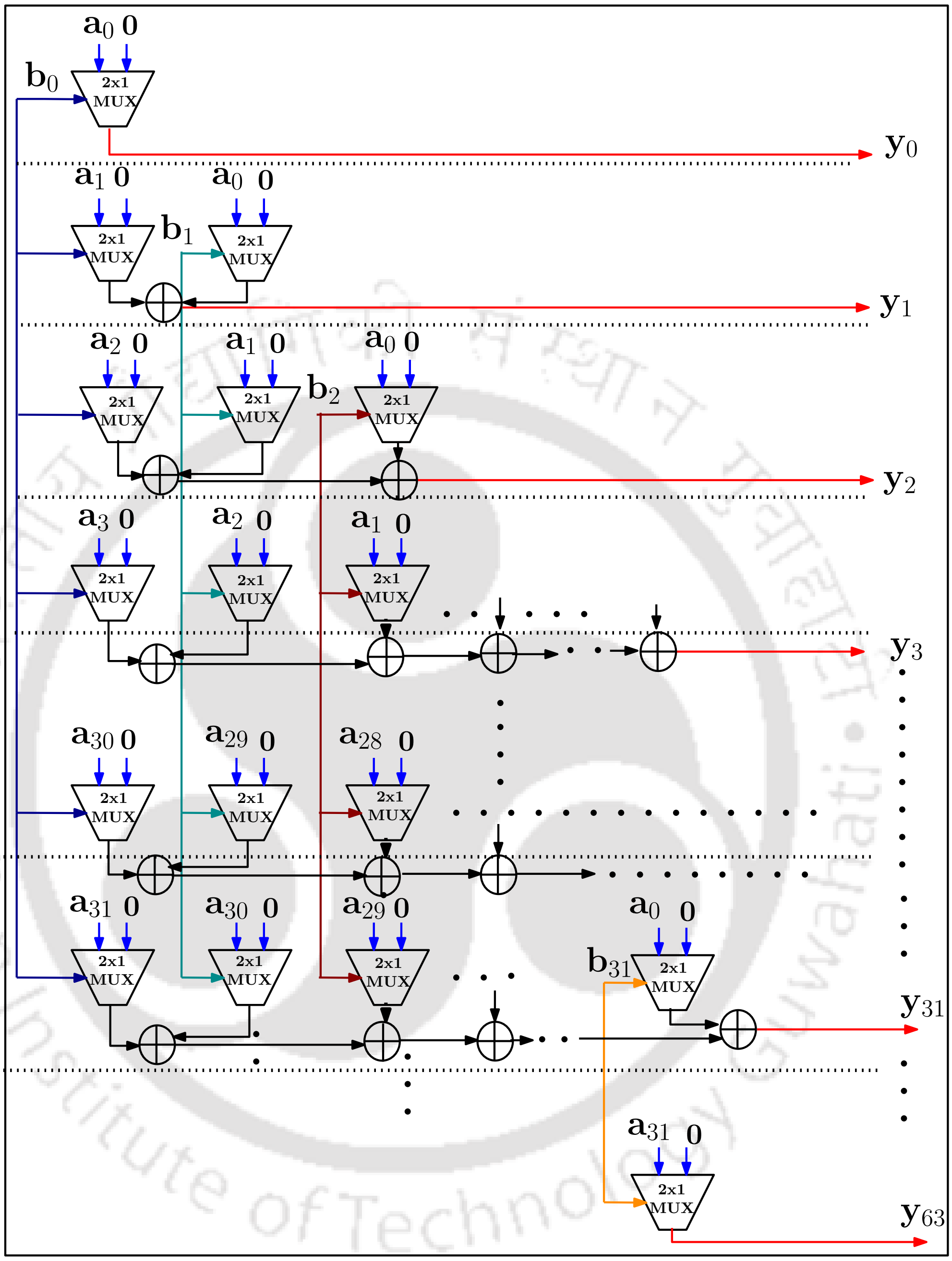

Figure 3.4: Hardware architecture of general-purpose multiplier used in distance calculation.

Accordingly, the node numbers of each data is stored in the NODE_NUM_REG.II. While the splitting is going on, the batch of data is transferred from DATA_REG.I to DATA_REG.II parallely through

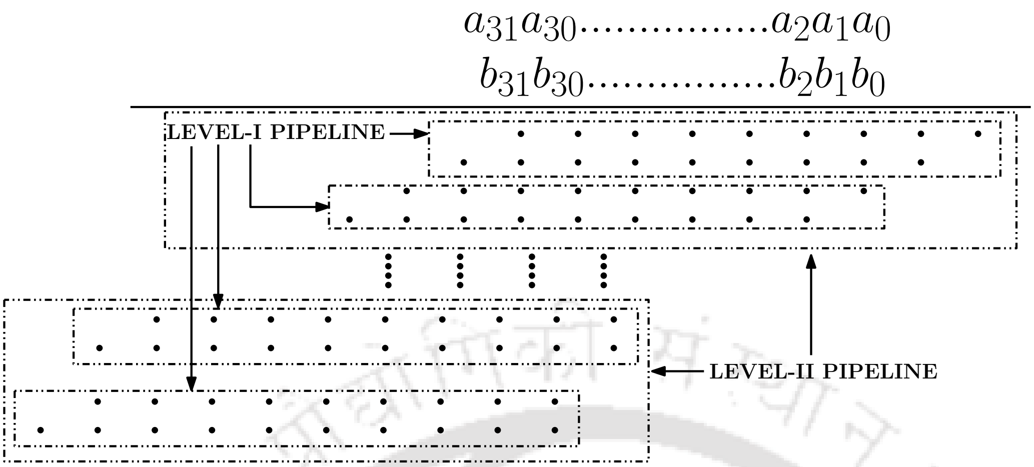

Figure 3.5: Pipelining of addition in the multiplier.

a latch. The clock period of the latch is chosen a little more than the latency of stage-II to avoid erroneous data overwriting. Thus, in stage-I and II, two different registers are used for storing node number (NODE_NUM_REG.I and II) and data-instances (DATA_REG.I and II) to allow pipelined execution. After the entire data batch and their respective node numbers are transferred to the stage-II registers, next batch of data is loaded into stage-I registers (DATA_REG.I).

3.3.3 Pipeline stage-II architecture

In stage-II, first depth 3 and then depth 4 node modules are executed as shown in Fig. 3.6. The data stored in DATA_REG.II is used in the processing of pipeline stage-II. The node number of data is obtained from NODE_NUM_REG.II. Similar to stage-I, a NODE_SEL with select line connected to NODE_NUM_REG.II selects the node to which data belongs and updates the mean accordingly. Once update is complete for a batch in that depth, then the split module is executed as discussed above. Each data-instance in that batch, belonging to a node, is assigned to either left or right child of that node during splitting as discussed above. Likewise, splitting is done for entire batch and the batch of data is sent to next depth. The updated mean and node number is stored in NODE_MEAN register and NODE_NUM_REG.II, respectively. Once all the data of this batch is processed in depth 3 and 4 then, the next batch of data is transferred from stage-I registers.

Once execution of all batches in stage-I and II is completed, then pruning stage is executed. Here, the same batches of data are passed again to update the total data and the purity of each node. Once the passing is complete, then the status of split nodes is checked as per (3.2) (whether pruned or not),

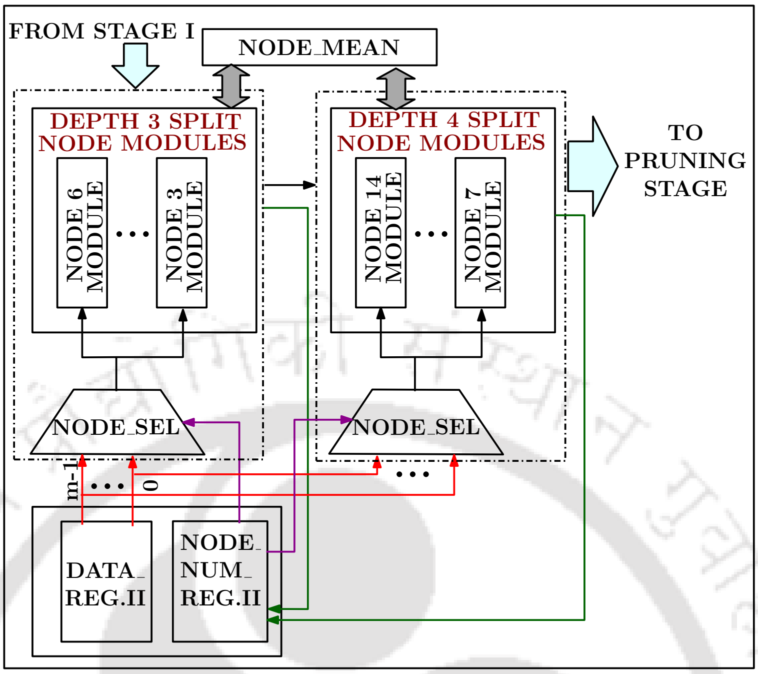

Figure 3.6: Hardware architecture of pipeline stage-II.

leaf node labels are assigned and accordingly, tree structure is stored in the memory which completes the training.

3.4 Results

3.4.1 Experimental set-up and dataset description

The FPGA implementation of the proposed hardware was performed on Virtex Ultrascale+ board XCVU13P-FLGA2577-3-E and operates at a maximum frequency of 62 MHz. This frequency is limited by the critical path of 16 ns required for data load and batch number update. The hardware was implemented using Verilog HDL in Vivado 2017. For software comparison, the batch-mode design was implemented in C and Python on an Intel i7 processor running at 3.2 GHz. It has been tested on 4 balanced datasets of varying size [40]. The *mammography* dataset consists of 5 feature ($m = 5$) input (size, shape, margin, density, severity) and class label corresponding to benign or malignant. The *occupancy* dataset also has $m = 5$ (CO_2, relative humidity, temperature, light, humidity ratio) and class label corresponding to room occupancy. The multi-class *activity recognition* dataset has $m = 7$ consisting of 7 means of reading from sensors and a label indicating whether the person is lying, standing or walking. Lastly, the skin dataset has $m = 3$ corresponding to R, G, B pixel values

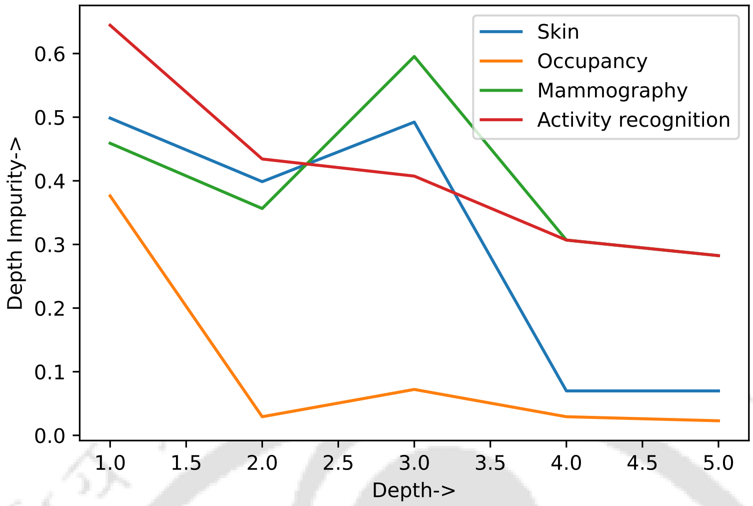

Figure 3.7: Plot showing variation of depth impurity with maximum depth (ζ_{max}) of TMDT for all datasets.

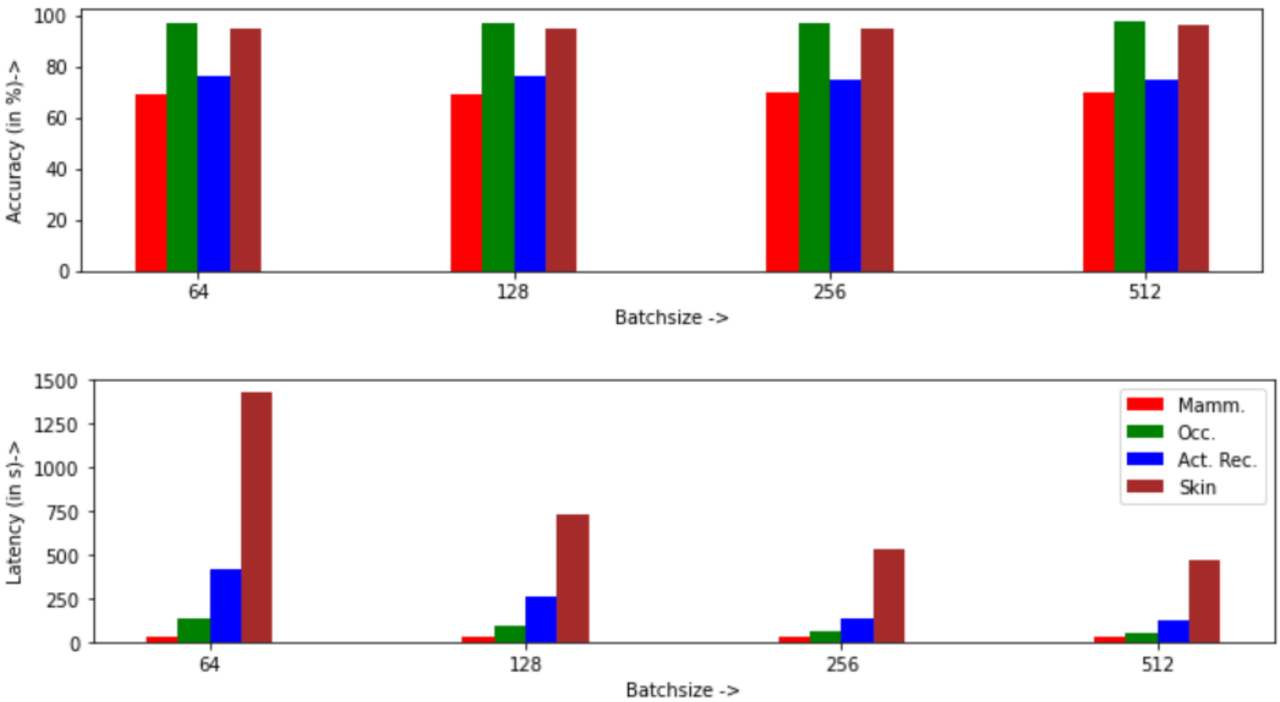

Figure 3.8: Bar-plot showing accuracy and latency variation for different batch sizes (b) of all datasets.

Table 3.1: Table recording the number of batches (H), accuracy and training latency on software (Python and C) and hardware platforms (FPGA) of all datasets for $N = 10$

Data-set	Mammography	Occupancy	Activity recog.	Skin
Python (in s)	4.52	49	120	473
C (in s)	0.09	1.47	2.58	7.34
FPGA (in s)	3.07×10^{-3}	26.54×10^{-3}	58.31×10^{-3}	270×10^{-3}
H	2	19	42	195
Accuracy(%)	70	98	75	96

Table 3.2: Table recording the accuracy and training latency comparison of proposed hardware with existing hardwares for varying training dataset size

Design	**1k**(Mammography) (in ms)	**10k**(Occupancy) (in ms)	**100k**(Skin) (in ms)	**Accuracy** (%)	**No.of FPGAs**
CART [20]	260	360	1790	-	4
TMDT Serial [19]	0.49	6.40	-	67	1
TMDT Mixed [41]	0.29	3.76	-	67	1
Proposed Design	0.99	6.85	67.42	70	1

Table 3.3: Table recording the resource utilisation comparison of proposed hardware with existing hardwares

Design	**LUT** (in K)	**FF** (in K)	**BRAM**	**DSP**	**Data** (in K)	**Max. Freq** (in MHz)	**Class**	**Task**
Optimised k-means [33]	14.16	24.49	240	186	16	200	Multi	Inference
TMDT Serial [19]	297.91	3.42	48	200	38	62	Binary	Training
TMDT Mixed [41]	298.96	3.57	187	60	38	71	Binary	Training
Proposed Design	154.65	5.44	24.5	0	0.512 (batch-size)	62	Multi	Training

and a binary label indicating skin colored pixel. Therefore, this hardware is designed to support a maximum of 7 features without any boundary on number of training data-instances. It is observed from Fig. 3.7 that the impurity of the node inputs (dataset) almost becomes constant at depth of $\zeta = 5$, so this work uses $\zeta_{max} = 5$. Using simulations on software platform, the values of update factor α, node input deficiency threshold ρ_c and node input purity η_p are empirically set to $\alpha = 0.875$, $\rho_c = 20$ and $\eta_p = 0.95$, respectively. The variation of accuracy and training latency with batch-size b for all datasets is shown in Fig. 3.8. The smallest dataset *Mammography* has only 769 training instances so, the value of b is taken such that $b \in [64, 512]$. The smallest value of b is 64 to allow sufficient data in the nodes as the DT has a depth of 5. The batch-size b is taken in power of 2 to allow efficient division in initialisation phase. From Fig. 3.8, it is observed that $b = 512$ has minimum latency and highest accuracy (except for Activity Recog. dataset which has a negligible drop in accuracy for $b = 512$). The training latency is lowest for $b = 512$ as the frequency of memory access is less. As the accuracy is maximum and latency is minimum so, for hardware implementation, training data is loaded in batch size of $b = 512$.

3.4.2 Experimental results

To increase the accuracy, same batches are passed through the DT in N epochs, where each epoch is formed by shuffling and re-ordering the data in the batches. This shuffling allowed fine-tuning of the DT parameters and hence improved the accuracy. Here, $N = 10$ epochs were used as further increase in epoch would result in increased latency and no improvement in accuracy. The comparison of training latency between hardware and software implementations and accuracy for each dataset for $N = 10$ is listed in Table 3.1. The latency of this hardware is independent of data features (m) as all features are executed in parallel. Although, the accuracy of medium-sized multi-class dataset is only 75% but medium-size binary dataset accuracy is found to be 98% as accuracy increases with increase in data-instances from each class. However, the accuracy for largest dataset *Skin* is observed to be only 96% due to highly overlapping data as shown in the data plot in previous chapter. Thus, the accuracy is observed to improve by a great extent as compared to the offline implementation as discussed in previous chapter. Further, it is observed that the hardware implementation is at least $27\times$ and $10^3\times$ faster as compared to corresponding C and Python implementations, respectively. The speed-up and accuracy of the proposed single FPGA hardware as compared to existing multiple [20] and single FPGA training accelerators proposed in previous chapter for varying n and for one epoch ($N = 1$) is shown in Table 3.2. The proposed architecture was found to be at least $26\times$ faster than [20] though latency is slightly more as compared to training accelerators proposed in previous chapter due to multiple memory access required for batch upload but, has improved accuracy. Also, the batch-mode implementation is able to perform training for comparatively larger training dataset consisting of almost $100k$ data-instances which was not possible in the offline implementations due to memory constraint. The hardware consumption (independent of board used) is compared with existing hardwares in Table 3.3. The memory (BRAM) usage is reduced (minimizes power) due to batch-mode training as compared to the classification hardware proposed in [33]. The DSP usage is also observed to be reduced (minimizes power) due to efficient multiplier and comparator designs as compared to [33]. This hardware, while accommodating larger training data-set, minimizes both LUT and BRAM usage as compared to training hardwares proposed in previous chapter. This hardware is implemented for batch-size of 512. However, there is no limitation on training dataset size as there is no upper limit on number of batches that can be processed by the hardware. This hardware consumes 1.79W dynamic power. The break-up of power consumption is shown in Fig. 3.9. Further, the proposed realisation is resource-efficient as compared to training architectures, proposed for $38k$ data-points.

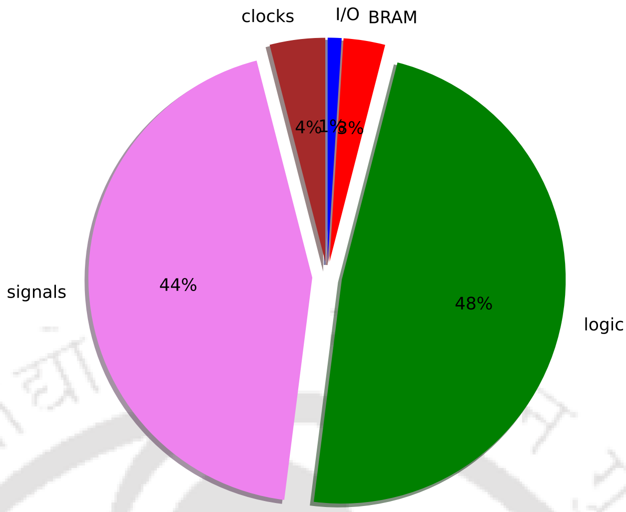

Figure 3.9: Pie-chart showing the break-up of power consumption.

3.5 Discussions

The proposed design implemented TMDT in batch-mode for multi-class classification. The hardware was implemented on state-of-the-art FPGA and supports a maximum frequency of 62 MHz. The FPGA implementation results were compared with both C and Python. It was found that the speed-up on FPGA was superior to both these platforms for similar application. The FPGA implementation executes at least 27× faster than C based implementation. Due to increase in training data-size, this design was found to have higher accuracy as compared to our previous designs. The depth pipelining also optimized the clock cycle usage and hence, further reduced the training latency. Thus, this implementation was found to be 26× faster than the existing training hardware [20] and our previous designs [19, 41]. The number of data processed is divided into batches and the hardware is re-used for each batch. Thus, this design also reduced the hardware resource usage. Hence, it is proved that this implementation is both faster and more resource-efficient than existing training hardware.

The proposed TMDT realizes a hierarchical two means clustering. It constructs a binary tree in an unsupervised framework. Finally, the terminal node data are inspected to assign class labels to leaf nodes. The splits learned in the TMDT are not optimal ones. These splits are not learned with objective of label impurity reduction at each node. Such axis-aligned split function learning is observed in CART and its variants. However, such split learning is also associated with additional split parameter search complexity (on account of sorting). To this end, we propose a hybrid tree consisting of both TMDT and axis-aligned split nodes. This proposal is described in the next chapter.

4

HDT: Hybrid Decision Tree

- R. Choudhury, S. R. Ahamed, and P. Guha, “FPGA Implementation of Low Complexity Hybrid Decision Tree Training Accelerator”, in IEEE Mid-West Symposium on Circuits and Systems (IEEE-MWSCAS), 2021.

Contents

Overview

The classification accuracy of TMDT can be improved by replacing mean-dependent nodes with axis-aligned nodes after a certain depth. In this chapter, a Hybrid Decision Tree (HDT) is proposed where *mean-dependent* nodes similar to TMDT are grown till a certain depth followed by nodes hosting *axis-aligned* split functions. The later set of nodes learn the split function parameters by minimizing the label impurity drop. This proposal reduces search complexity of axis-aligned split function parameters by avoiding the sorting operations with a limited number of impurity calculations. The hardware implementation is faster by 34× than the software program for the same HDT. While it is 84× faster than FPGA implementation of CART training. It is because CART algorithm has more complexity than HDT. So, CART algorithm is much slower than HDT algorithm. It was also found to be more resource-efficient as compared to the existing classification or training hardware.

4.1 Need for HDT

The split nodes have saturated node purity after a certain depth. The accuracy can be improved by further splitting the nodes if sufficient data instances are available in the node. So, in this chapter, HDT algorithm and its implementation on FPGA are proposed. It consists of hybrid nodes. First, mean-dependent nodes split the data into two clusters till a certain depth. Then, these clusters are subjected to further splitting using axis-aligned nodes. Unlike axis-aligned nodes used in conventional DT algorithms, the axis-aligned nodes in HDT perform a search on a reduced set of impurity parameters. Thus, HDT reduces the complexity to a great extent and hence, accelerates the training as compared to the conventional DT. The accuracy is also higher than the TMDT. The parallel processing on FPGA platform further enables the hardware to accelerate the training process. The reconfigurability of FPGA allows the hardware to be used for multiple applications by re-training with desired application dataset.

4.2 Proposed HDT algorithm

The HDT structure has three types of nodes. First, *mean-dependent* split nodes (N_{md}) where the node stores two means and the data is split depending on the distance from the means. Second, *axis-aligned* split nodes (N_{aa}) where the node stores a threshold value and the data is split depending on whether the data value at a certain dimension is greater (or lesser) than a threshold value. Third, leaf nodes (N_{ln}) that store the class label based on the majority category of the dataset at the terminal node. The tree starts with a mean-dependent root node. The input dataset is split through mean-dependent nodes till a depth of ζ_{md} as shown in Fig. 4.1. From depth $\zeta_{md}+1$ onward till $\zeta_{max}-1$, the data is processed through axis-aligned split nodes. The DT terminates with all leaf nodes at the depth ζ_{max}. A post pruning strategy is applied to identify leaf nodes between root node and depth $\zeta_{max}-1$.

The HDT algorithm is discussed in Algorithm 4.1. Let the training dataset for mean-dependent node a be $\mathbf{X}_a = \left\{\mathbf{x}^k : y(\mathbf{x}^k) = c; \mathbf{x}^k \in \mathbb{R}^m; k = 0, \ldots (n_a - 1)\right\}$ where, $c = 0, \ldots (C-1)$ and C is the number of class labels in the training dataset. Processing for a mean-dependent node has three phases v.i.z. initialization, update and split. In the initialization phase, the dataset $\mathbf{X}_a$ is randomly divided into two groups. The means of these two groups are estimated as initial values for left $\mu_{aL}^{(0)}$ and right mean $\mu_{aR}^{(0)}$. During update phase, an instance $\mathbf{x}^k$ is tested for proximity from μ_{aL}^{k-1} and μ_{aR}^{k-1}. If $\mathbf{x}^k$ is closer to μ_{aR}^{k-1}, then the later is updated. Otherwise, μ_{aL}^{k-1} is updated. This is performed using an update rate α (discussed in previous chapters) using (4.1).

$$\mu_a^k = \mu_a^{k-1} + \alpha(\mathbf{x}^k - \mu_a^{k-1}) \tag{4.1}$$

The instances of $\mathbf{X}_a$ are again passed through node a for dataset splitting. Instances closer to μ_{aL} (or μ_{aR}) are sent to aL (or aR), the left (or right) child of a. These instances form the data subset $\mathbf{X}_{aL}$ (or $\mathbf{X}_{aR}$). These data subsets are similarly processed in the children nodes. This process is continued recursively till the depth ζ_{md} is reached. This work opts for post-pruning of HDT to implement efficient pipelining of nodes. The nodes in the depth range $[2, \zeta_{md}]$ are set as leaf nodes, if the input dataset size is lesser than a threshold ($n_a < \rho_c$). The value of ρ_c is set as 128. A leaf node has no child node and is tagged with the label of the majority class of its input dataset.

The dataset split by mean-dependent nodes at depth ζ_{md} are further processed by axis-aligned split nodes from depth $\zeta_{md}+1$ onward. Let, $\mathbf{X}_b = \left\{\mathbf{x}^k : y(\mathbf{x}^k) = \mathbf{c}; \mathbf{x}^k \in \mathbb{R}^m; k = 0, \ldots (n_b - 1)\right\}$ be the input to the axis-aligned split node b. The input data ranges $R[l] = R_{max}[l] - R_{min}[l]$ along each dimension ($l = 0, \ldots (m-1)$) are estimated first. This is performed through a linear search along each dimension. Then, for each l^{th} dimension, f threshold values are computed. This range information is used to construct a $f \times m$ threshold matrix $\mathbf{\Theta}$ according to (4.3) containing thresholds

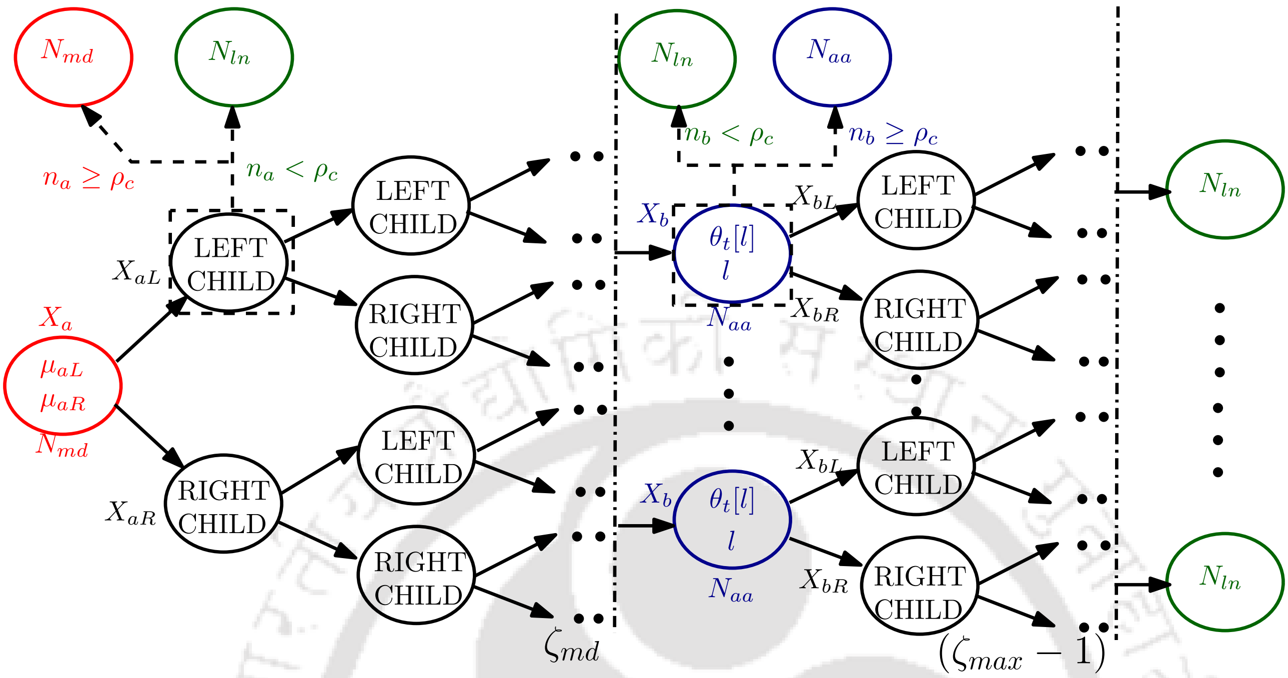

Figure 4.1: The flow of HDT algorithm. For each node a (or b), possible node types depending on data-size n_a (or n_b) are shown by dashed arrows and node splits are shown by solid arrows. The mean-dependent nodes are indicated in red, axis-aligned in blue and leaf nodes in green.

$\theta_t[l]$ $(t = 1, \ldots f; l = 0, \ldots (m-1))$.

$$\Delta[l] = \frac{(R_{max}[l] - R_{min}[l])}{2f} \tag{4.2}$$

$$\theta_t[l] = 0.25(R_{max}[l] - R_{min}[l]) + (\Delta[l] \times t) \tag{4.3}$$

This leads to $f \times m$ possible axis-aligned split functions of the form $\mathcal{H}_b(l, \theta_t[l])$ for the node b. The split function $\mathcal{H}_b(l, \theta_t[l])$ can be used to split the input dataset $\mathbf{X}_b$ into the disjoint subsets $\mathbf{X}_{bL}(l, \theta_t[l]) = \{\mathbf{x^k} : \mathbf{x^k}[l] \leq \theta_t[l]; \mathbf{x^k} \in \mathbf{X}_b\}$ and $\mathbf{X}_{bR}(l, \theta_t[l]) = \{\mathbf{x^k} : \mathbf{x^k}[l] > \theta_t[l]; \mathbf{x^k} \in \mathbf{X}_b\}$. The impurity drop $\mathbf{\Delta Im}(\mathbf{X}_b, l, \theta_t[l])$ corresponding to split parameters $(l, \theta_t[l])$ is estimated using (4.4).

$$\mathbf{\Delta Im}\left(\mathbf{X}_b, l, \theta_t[l]\right) = \mathbf{Im}\left(\mathbf{X}_b\right) - \mathbf{Im}\left(\mathbf{X}_{bL}, \mathbf{X}_{bR}, l, \theta_t[l]\right) \tag{4.4}$$

$$\mathbf{Im}\left(\mathbf{X}_{bL}, \mathbf{X}_{bR}, l, \theta_t[l]\right) = \frac{|\mathbf{X}_{bL}(l,\theta_t[l])|}{|\mathbf{X}_b|}\mathbf{Im}(\mathbf{X}_{bL}(l, \theta_t[l])) + \frac{|\mathbf{X}_{bR}(l,\theta_t[l])|}{|\mathbf{X}_b|}\mathbf{Im}(\mathbf{X}_{bR}(l, \theta_t[l])) \tag{4.5}$$

Here, $\mathbf{Im}(\mathbf{X}_b)$ refers to the impurity of a dataset $\mathbf{X}_b = \left\{\mathbf{x}^k : y(\mathbf{x}^k) = \mathbf{c}; \mathbf{x}^k \in \mathcal{R}^m; k = 0, \ldots (n_b - 1)\right\}$ containing instances from C different classes input to the node b hosting the decision function $\mathcal{H}_b(l, \theta_t[l])$: $[\mathbf{x}[l] > \theta_t]$. The expression $\mathbf{Im}\left(\mathbf{X}_{bL}, \mathbf{X}_{bR}, l, \theta_t[l]\right)$ refers to the joint impurity of the data sets input to left (bL) and right (bR) children nodes of b. The optimal split function parameters $(l_b^\star, \theta_{t^\star}[l_b^\star])$ for node b are chosen to maximize the (parent-to-children) impurity drop $\mathbf{\Delta Im}\left(\mathbf{X}_b, l, \theta_t[l]\right)$ or equivalently, to minimize $\mathbf{Im}\left(\mathbf{X}_{bL}, \mathbf{X}_{bR}, l, \theta_t[l]\right)$. The misclassification impurity of the dataset $\mathbf{X}_b$ is computed as follows.

$$q_b(c_r) = \sum_{k=0}^{n_b-1} \delta[y(\mathbf{x}^k) - c_r] \tag{4.6}$$

$$P_b(c_r) = \frac{q_b(c_r)}{\mid \mathbf{X}_b \mid} \tag{4.7}$$

$$\mathbf{Im}(\mathbf{X}_b) = 1 - \max_{c=0,\ldots C-1} P_b(c) \tag{4.8}$$

Here, $q_b(c_r)$ $(\sum_{c=0}^{C-1} q_b(c) = \mid \mathbf{X}_b \mid)$ is the number of data vectors in node b which belongs to class c_r. Here, $\delta[y(\mathbf{x}^k) - c_r]$[1] is used to calculate the number of data instances $\mathbf{x}^k$ having class label $y(\mathbf{x}^k) = c_r$. In (4.7), $P_b(c_r)$ refer to the fraction of data-vectors belonging to the class/category c_r in dataset $\mathbf{X}_b$ of node b. Using (4.5) - (4.8) the joint children impurity can be expressed as follows.

$$\begin{aligned}\mathbf{Im}\left(\mathbf{X}_{bL}, \mathbf{X}_{bR}, l, \theta_t[l]\right) &= \frac{\mid \mathbf{X}_{bL}(l, \theta_t[l]) \mid}{\mathbf{X}_b}\mathbf{Im}(\mathbf{X}_{bL}(l, \theta_t[l])) + \frac{\mid \mathbf{X}_{bR}(l, \theta_t[l]) \mid}{\mathbf{X}_b}\mathbf{Im}(\mathbf{X}_{bR}(l, \theta_t[l])) \\ &= \frac{\mid \mathbf{X}_{bL}(l, \theta_t[l]) \mid \{1 - \max_{c=0,\ldots C-1} q_{bL}(c)\}}{\mid \mathbf{X}_b \mid} + \frac{\mid \mathbf{X}_{bR}(l, \theta_t[l]) \mid \{1 - \max_{c_r=0,\ldots C-1} q_{bR}(c_r)\}}{\mid \mathbf{X}_b \mid}\end{aligned} \tag{4.9}$$

After performing the algebraic manipulations in (4.9) and using $\mid \mathbf{X}_{Lb}(l, \theta_t[l]) \mid + \mid \mathbf{X}_{Rb}(l, \theta_t[l]) \mid = \mid \mathbf{X}_b \mid$ $(\forall l, \theta_t[l])$ provides the following simplified expression.

$$\mathbf{Im}\left(\mathbf{X}_{bL}, \mathbf{X}_{bR}, l, \theta_t[l]\right) = \frac{\mid \mathbf{X}_{bL}(l, \theta_t[l]) \mid \{1 - \max_{c=0,\ldots C-1} q_{bL}(c)\} + \mid \mathbf{X}_{bR}(l, \theta_t[l]) \mid \{1 - \max_{c_r=0,\ldots C-1} q_{bR}(c_r)\}}{\mid \mathbf{X}_b \mid} \tag{4.10}$$

Note that the denominator in (4.10) is invariant with respect to $(l, \theta_t[l])$. Thus, by using (4.10), the joint children impurity values can be estimated by varying split function parameters while avoiding the division by dataset sizes. Avoiding divisions in searching for the optimal parameters greatly reduces the computation cost. This process is continued recursively till the depth of $\zeta_{max} - 1$. The nodes in depth range $[\zeta_{md} + 1, \zeta_{max} - 1]$ are also subjected to post-pruning for identifying potential leaf nodes tagged with their majority class labels. Finally, the HDT terminates at depth ζ_{max} where all nodes are set to leaf nodes. These leaf nodes are tagged with the class label of the dominant category of their respective input datasets. During classification, an input data instance $\mathbf{x}^k$ is first routed through

[1] $\delta[i - j]$ is the Kronecker Delta function where $\delta[i - j] = 1$ for $i = j$ and $\delta[i - j] = 0$, otherwise

Algorithm 4.1: HDT Algorithm

```
while a= 0 to 2^{ζmd} − 1 do
    Load X_a; Compute μ_{aL}^{(0)} and μ_{aR}^{(0)};
while ζ=1 to ζmd do
    while a= nodes in depth d do
        while k=0 to (n_a − 1) do
            Load x^k, μ_{aL}^{k−1}, μ_{aR}^{k−1};
            Compute d_{aL}(x^k) =|| μ_{aL}^{k−1} − x^k ||_2;
            Compute d_{aR}(x^k) =|| μ_{aR}^{k−1} − x^k ||_2;
            if d_{aL}(x^k) ≤ d_{aR}(x^k) then
                μ_{aL}^k = μ_{aL}^{k−1} + β(x^k − μ_{aL}^{k−1});
            else
                μ_{aR}^k = μ_{aR}^{k−1} + β(x^k − μ_{aR}^{k−1});
        μ_{aL} = μ_{aL}^{(n_a−1)}; μ_{aR} = μ_{aR}^{(n_a−1)};
while ζ=1 to ζmd do
    while a= nodes in depth ζ do
        while k=0 to (n_a − 1) do
            Split:[X_a = X_{aL} ∪ X_{aR}] ∧ [X_{aL} ∩ X_{aR} = φ];
            X_{aL} = {x : d_{aL}(x) < d_{aR}(x); x ∈ X_a};
            X_{aR} = {x : d_{aL}(x) ≥ d_{aR}(x); x ∈ X_a}
while ζ = (ζmd + 1) to (ζmax − 1) do
    while b = nodes in depth ζ do
        Load X_b;
        Evaluate R_min, R_max, Θ;
        Θ = {θ_t[l]}; t ∈ [1, f]; l ∈ [0, (m − 1)];
        (l⋆, θ_{t⋆}[l⋆]) = arg max_{(t,l)} ΔIm (X_b, l, θ_t[l]);
        Split:[X_b = X_{bL} ∪ X_{bR}] ∧ [X_{bL} ∩ X_{bR} = φ];
        X_{bL} = {x : x[l⋆] ≤ θ_{t⋆}[l⋆]; x ∈ X_b};
        X_{bR} = {x : x[l⋆] > θ_{t⋆}[l⋆]; x ∈ X_b}
while ζ = 2 to (ζmax − 1) do
    while α = nodes in depth ζ do
        if n_α < ρ_c then
            Set α as leaf node ;
            ClassLabel(α) = arg max_{c_r} q_α(c_r)
        else
            Preserve node α;
while λ = nodes in depth ζmax do
    Set λ as leaf node ;
    ClassLabel(λ) = arg max_c q_λ(c)
```

Table 4.1: Table comparing HDT with conventional DT (C4.5) and TMDT

	Conv. DT	**TMDT** [19]	**HDT**
Node type	Complex axis-aligned	Mean-dependent	Mean-dependent + Simple axis-aligned
Node comp.	$O_{CDT} = mnlog(n)O_{cp} + m(n-1)O_{imp}$	$O_{TMDT} = n(4T_{dc} + 2O_{cp} + O_u) + O_{imp}$	$O_{HDT} = O_{TMDT}$ or $n(2m + k \times m + 1)O_{cp} + (k \times m)(O_\theta + O_{imp})$
Time comp.	T_{CDT}	$T_{TMDT} < T_{CDT}$	$T_{TMDT} < T_{HDT} < T_{CDT}$

the mean-dependent nodes till the depth ζ_{md}. Here, $\mathbf{x}^k$ is sent to left (or right) child node by checking its proximity to left (or right) mean stored at the mean-dependent node. From depth $\zeta_{md}+1$ onward, $\mathbf{x}^k$ is routed through axis-aligned nodes. Here, $\mathbf{x}^k$ is sent to right (or left, otherwise) according to the learned split function ($[x[l^\star] > \theta^\star]$) of that node. Such routing through mean-dependent and axis-aligned nodes terminates at a certain leaf node. The instance $\mathbf{x}^k$ is finally classified with the category label associated with the terminal leaf node. The comparison of HDT with conventional DT and TMDT [19] w.r.t type of node in DT, calculation complexity (O) of each node (node complexity), and timing complexity (T) is recorded in Table 4.1. The conventional DT consisting of complex axis-aligned node has $nlog(n)$ sorting (O_{cp}) and $(n-1)$ impurity calculations (O_{imp}) for j^{th} dimension where $j = 0, \ldots, (m-1)$ and n is the total number of data vector in the node. Similarly, for each data-vector in a node, TMDT has four distance calculations (O_{dc}) and four comparisons (O_{cp}) of data-vector with two means (two during mean update and two during splitting), one update operation (O_u) and one impurity calculation (O_{imp}). The HDT consists of both mean-dependent nodes (similar to TMDT) and simple axis-aligned nodes. So, the complexity of HDT is equal to TMDT for mean-dependent node a. For an axis-aligned node b, the complexity is two comparisons for determining the range ($R[j]$) for each dimension j for each data vector $\mathbf{x}^{(i)}$. It also involves $k \times m$ calculations to compute θ (O_θ) as per (4.2). It also has $k \times m$ impurity calculations (O_{imp}) (4.9). Then, $k \times m$ comparisons are performed for each data-vector to select the best split value $(j_b^\star, \theta_{t^\star}[j_b^\star])$ for node b. Finally, one comparison for each data-vector is done to split the node data into left ($\mathbf{X}_{Lb}$) and right child data($\mathbf{X}_{Rb}$). As $k << n$ and $O_{cp} < O_s$ (as sorting data involves multiple comparisons) so, $O_{CDT} > O_{HDT} > O_{TMDT}$ and $T_{CDT} > T_{HDT} > T_{TMDT}$. Moreover, in this design, HDT consists of mean-dependent nodes till depth 2 and axis-aligned nodes in depth 3 only, which reduces the overall complexity of HDT.

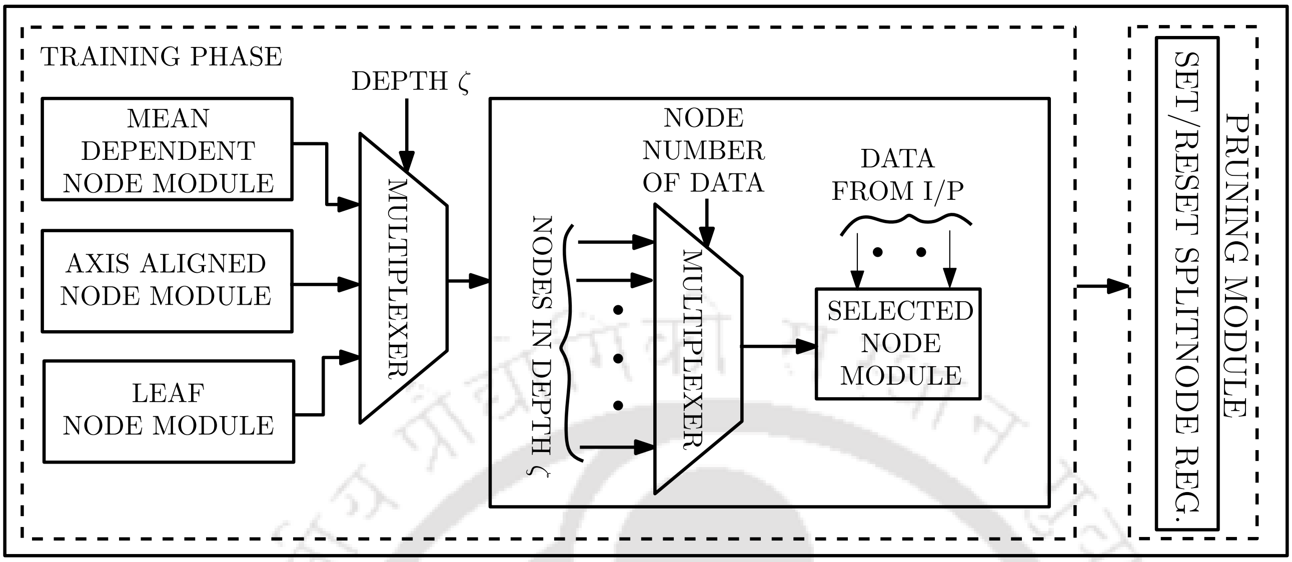

Figure 4.2: Outline of hardware architecture for HDT training. The depth variable chooses the type of node to be executed. Then, the node number of data is used to select the node to be executed.

4.3 Proposed hardware architecture

The HDT algorithm have been optimized as described above to reduce the complexity. The architecture has been implemented to maximize resource utilization and minimize latency. So, the mean-dependent and axis-aligned node architecture are parallelized for all features to reduce latency. This will require $m\times$ more resources. However, the latency will also be reduced by a factor of m. Also, the comparator has been pipelined to reduce critical path, maximize resource utilization and increase frequency to overall latency. The HDT algorithm as described in Algorithm 4.1 is implemented in a modular fashion as shown in Fig. 4.2. It consists of mean-dependent, axis-aligned and leaf node modules. The module to be executed is chosen according to the current depth.

4.3.1 State diagram for the control unit

The control unit used to control the execution of the hardware is shown in Fig. 4.3. It starts with the initialization of all registers storing the initial values. Then, once the reset is 0, the mean update for a node starts and the node to which the data-instance belongs is updated (node 1 onwards as all data is processed in node 0). Once all the data are processed, the data split is done. Then, the depth is incremented and if depth $\leq\zeta_{md}$, then the data is similarly processed in next depth. When the depth $> \zeta_{md}$, then the data is used to update the range (R_{max}, R_{min}) of that node. Once the range update is complete for all data in that depth, then *theta* and impurity for a node in that depth are calculated. Next, the best-split values (θ and dimension) are selected and stored as node data. Similarly, all the nodes in that depth are processed and their best-split values are stored. Then, the depth is incremented. If it is less than ζ_{max}, then the data is similarly processed. Otherwise, the leaf node modules are executed. Once all leaf nodes are updated, the training is complete and the hardware can be used for classification.

4.3.2 Overall architecture

As discussed in previous section, HDT comprises of three types of node: mean-dependent (N_{md}), axis-aligned node (N_{aa}) and leaf node (N_{ln}). The leaf node module only consists of updating the label according to the maximum number of data-instances that particular node is exposed to. The algorithmic details of mean-dependent and axis-aligned node operations presented in previous section are mapped onto hardware architecture modules. For reducing the latency and resource utilization, HDT architecture processes data in a depth-wise manner. The node module is re-used in each depth as only one node is processed at a time in depth ζ. The HDT execution starts with the initialization phase. In the initialization phase, the pre-calculated initial mean of mean-dependent nodes are stored into the node memory as discussed in Section 4.2. The flow of execution of training phase and pruning module after initialization is shown in Fig. 4.2. In training phase, depth is used as the select line of the multiplexer which selects the type of node in which the current instance of data $\mathbf{x}^k$ will be executed. Once the node type is selected, then the corresponding node number of data is used to select the desired node module to be executed. First, the entire data set in that depth is passed in update phase of corresponding node modules. Then, the same data set is similarly passed through the split phase. In split phase, the node number register is updated with the node number (i.e., left/right child node of current node) in which the data is to be executed in the next depth. After entire tree is learnt, then the tree is pruned. In pruning module, an g-bit (where g is the total nodes in HDT) SPLITNODE REG. is updated. The a^{th}-bit position in SPLITNODE REG. indicates whether node a is a split (SPLITNODE$[a] = 0$) or leaf node (SPLITNODE$[a] = 1$) where $a = 0, \ldots, g$. In the pruning phase, if the node (at $\zeta \leq (\zeta_{max} - 1)$) satisfies the pruning condition, then the corresponding bit is set as 1 (leaf node).

In the update phase, the initial means of mean-dependent nodes (starting from node 0), stored during initialization, are updated according to (4.1). After mean-update is done with the data in that depth, then the data is split according to the split rule of mean-dependent node and the next depth

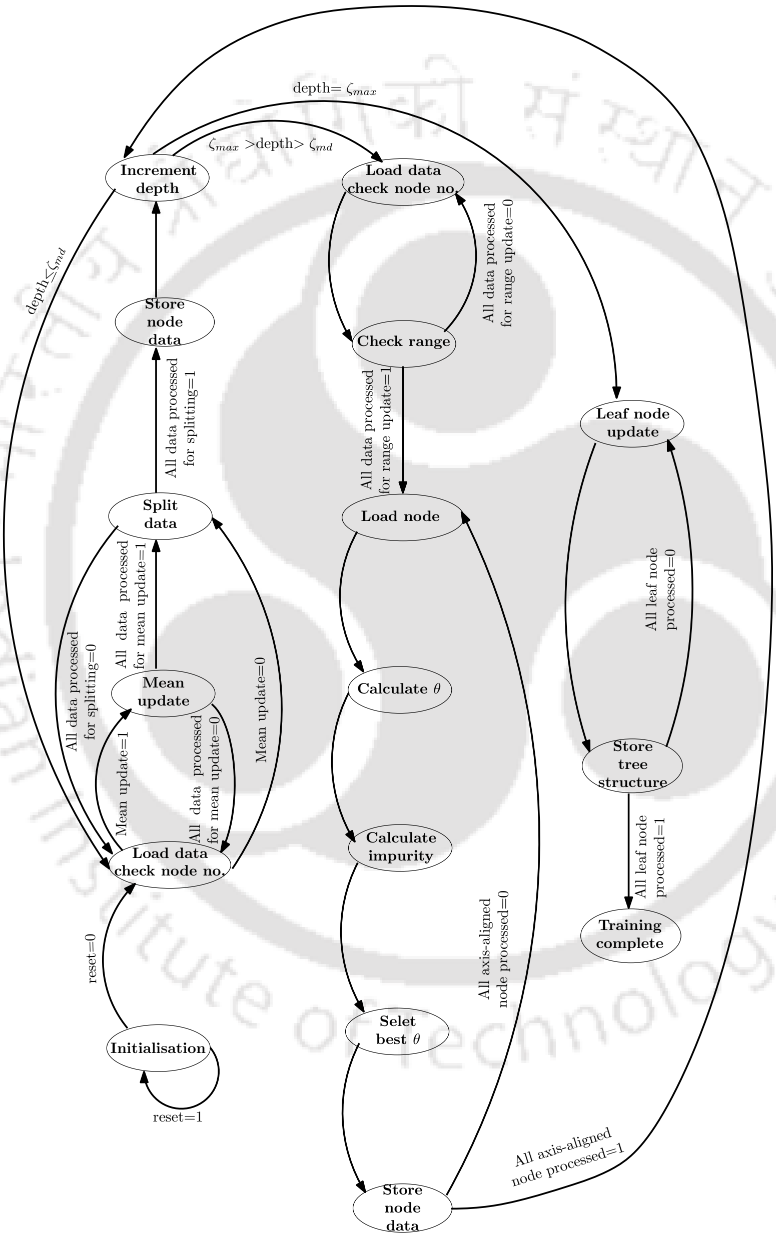

Figure 4.3: State diagram showing the states involved in the hardware.

is executed similarly. Once depth $\zeta_{md}+1$ is reached, then the axis-aligned node module is executed. First, the data in depth ζ (for $\zeta_{max} > \zeta \geq \zeta_{md}+1$) is processed in update phase of axis-aligned node, i.e., range of the node is updated. After determining the range from the node data in range update phase, then the split module of axis-aligned node is executed.

The training starts with the processing of root node 0 (in depth 1) in update module of mean-dependent node, where the training data $\mathbf{x}^k$ is taken sequentially from BRAM. For m dimensions, m instances of BRAMs are used to reduce latency. Once update is complete for all data-instances, then in split module each $\mathbf{x}^k$ is assigned to either left or right child present in next depth. In next depth, the node number associated with each $\mathbf{x}^k$ is used to select the corresponding node parameters as shown in Fig. 4.4. The $data_{in}$ is the updated node parameters obtained from the update phase which is stored in the memory during write cycle. The $\overline{RD}/WR$ signal is used to control the read and write operation. When $\overline{RD}/WR = 0$, the data is read from the memory and if $\overline{RD}/WR = 1$, the data is written into the memory. A $\zeta_{max} \times (2^{\zeta_{max}} - 1)$ address decoder circuitry is used to select the node data based on the node number of data. Further, a dual port RAM is designed to allow two memory accesses at the same time. Thus, the two node parameters, left and right mean in case of mean-dependent node ($\theta^\star$ and $l^\star$ in case of axis-aligned node) stored in consecutive memory locations is accessed in same cycle as shown in data storage circuitry. Then, while processing the same dataset in the next depth, each $\mathbf{x}^k$ is sent to respective node (in the same sequence as previous depth), assigned during splitting phase in previous depth.

Similarly, for depth $\zeta_{md}+1$ onward, the axis-aligned node module is used. In this node, there are two types of modules, range update and split module. The range update module examines l^{th} dimension of the incoming training data instance $x^k[l]$ whether it falls within the range $[R_{min}[l], R_{max}[l]]$. If the value is not within the range then, the range is modified otherwise, it is kept as in previous iteration and accordingly $\theta_t[l]$ values are calculated (4.3). Once all the data in that depth are checked in range update module, then they are executed in split module of their respective nodes. In the split module, all dimensions of the training data-instance run in parallel which enables to speed-up the training.

This process is repeated for all nodes in that depth. After entire dataset is processed in that depth then it is processed similarly in the next depth. This process continues till maximum depth ζ_{max} (consisting of leaf nodes only) is reached. The leaf node module have data-size less than ρ_c and

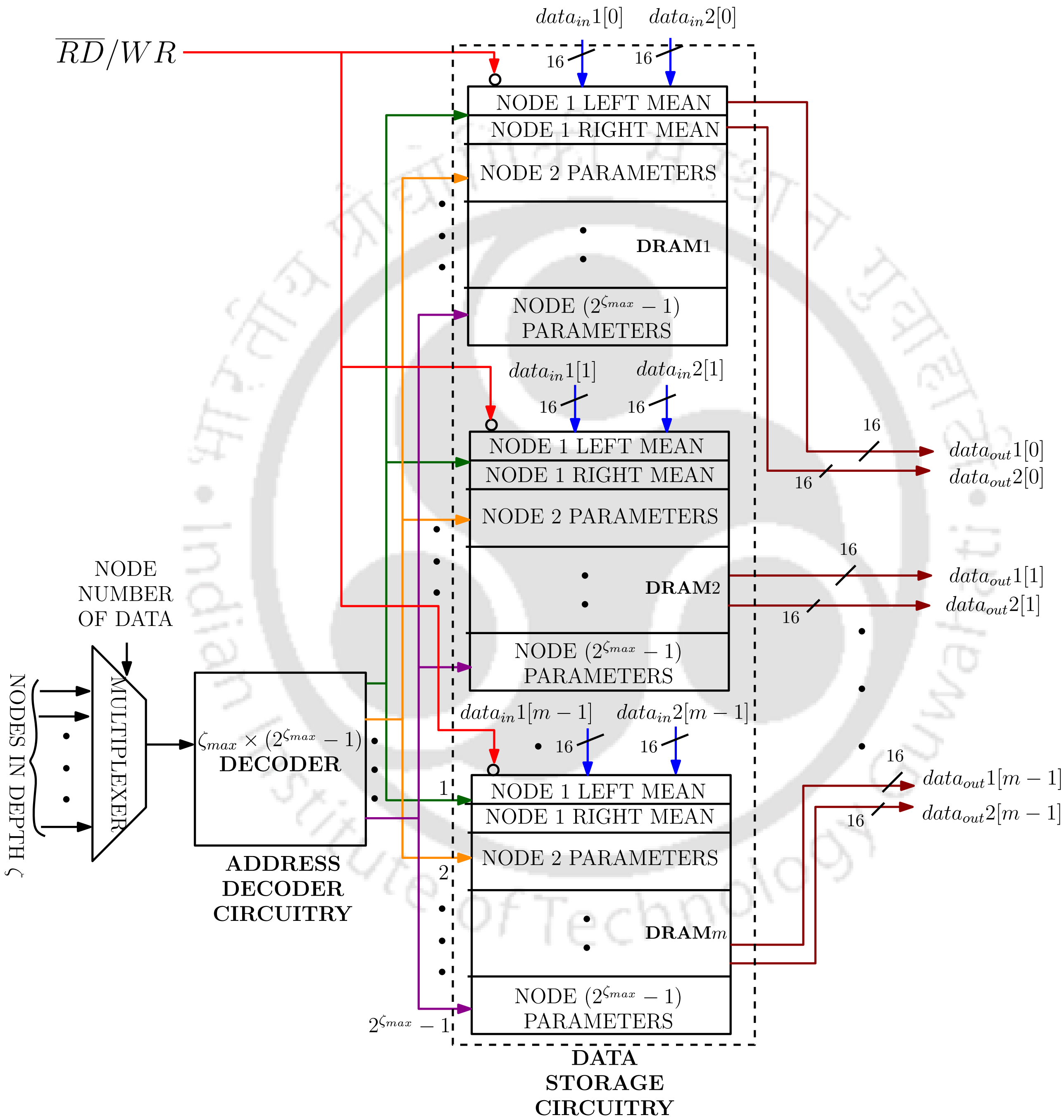

Figure 4.4: Hardware architecture for NODE MEMORY. The blue lines show the input data and brown lines show the output data.

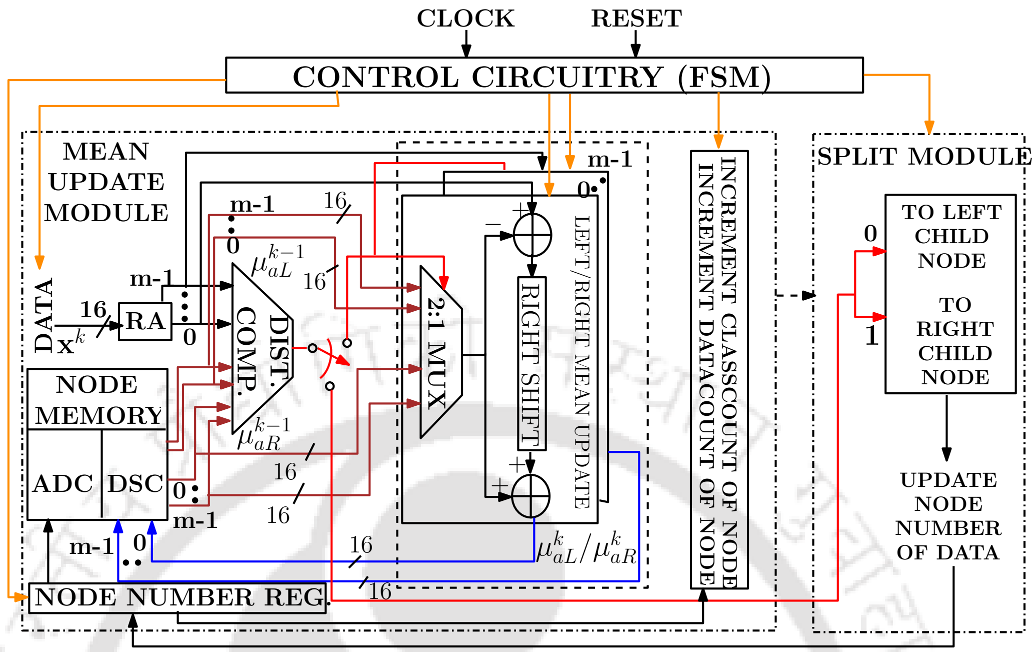

Figure 4.5: Hardware architecture for mean-dependent node module in HDT. Here, RA denotes Register Array, ADC denotes Address Decoder Circuitry and DSC denotes Data Storage Circuitry.

consists of label update operation only as described in Section 4.2. Then in pruning phase, the total data of each node above depth ζ_{max} is checked. If the total data is less than the required length for the node to be split node, then the node is set as leaf node and its child nodes are removed. In this way, all the nodes till depth $\zeta_{max} - 1$ is checked which concludes the training.

4.3.3 Mean-dependent node architecture

The mean-dependent node architecture is shown in Fig. 4.5. This architecture consists of update and split module. In the update module, the node means are retrieved from the NODE MEMORY according to the node number of data. Then, their distance with $\mathbf{x}^k$ is calculated. The distance of $\mathbf{x}^k$ with both left and right mean is then calculated and compared in parallel. The closest mean is updated according to (4.1) for all dimensions in parallel in the mean update module. For k^{th} iteration, it has three external inputs, the selected means of a^{th} node to which $\mathbf{x}^k$ belongs (from NODE MEMORY) and the data-instance $\mathbf{x}^k$. A 2:1 multiplexer is used to select the closer mean whose select line is connected to the output of DIST. COMP. module discussed in Section 4.3.5. This mean is subtracted from $\mathbf{x}^k$ as per (4.1). Next, the multiplication operation is realised using shifters as α is a constant and its value is approximated using sum of power of 2 (right shift is used as $\alpha < 1$) as discussed in previous chapter. This product is added to the selected mean to get the final desired output. This output of mean-update module is the updated mean of node a. For m-dimensional training data, m such modules are run in parallel to update all dimensions of the mean in parallel. The Register Arrays (RAs) storing the class-count (no. of data from each class the node is exposed to) and datacount, i.e., total number of data-instances travelling to node under consideration (required for pruning) are also updated in this module.

This process is repeated for entire data in that depth. Once update is done then, the split module is executed. Similarly, the split module compares the proximity of data with both left and right means (stored in NODE MEMORY) of the corresponding node. The data is assigned left child node number if it is closer to left mean otherwise, it is assigned the right child node number. The distance comparator (DIST COMP.) from update module is re-used in the split module (through a switch controlled by completion signal of update phase) to determine the child node number closer to the data. As discussed earlier, the parallel computation of distance of data with both left and right means optimizes the clock cycle consumption during splitting also. To further optimize the throughput, pipelined comparators are used as discussed below. The data is assigned to the child node present in the next depth and accordingly, the node number associated with it is updated. Similarly, entire training data in a particular depth is processed in their respective mean-dependent nodes till it reaches depth $\zeta_{md} + 1$.

4.3.4 Axis-aligned node architecture

The axis-aligned node architecture is implemented in parallel as shown in Fig. 4.6. This architecture consists of two modules, range update module and split module. For a certain depth ζ ($\zeta_{max} > \zeta \geq \zeta_{md} + 1$), data-instance $\mathbf{x}^k$ is routed to its assigned node where it is first processed in range update module and then, splitted according to split rule as described in Section 4.2. In range update module, for each dimension l, $x^k[l]$ is compared to $R_{max}[l]$ and $R_{min}[l]$ using two comparators in parallel to reduce clock cycle. The output of these comparators are fed as input to OR gate and the output of OR gate is used as the control signal for θ UPDATE module. This control signal activates the update module if OR-gate output is 1, else it remains deactivated. The θ UPDATE module updates the $\theta_t[l]$ value whenever $x^k[l]$ is outside the range $[R_{min}[l], R_{max}[l]]$. The output of OR gate is 1 if the output of one or both comparators are 1, i.e., if $x^k[l] > R_{max}[l]$ and $R_{min}[l] > x^k[l]$. This output is used to select the input of the two multiplexers whose one input is connected to $x^k[l]$ and the other input to $R_{max}[l]$ and $R_{min}[l]$. The range for l^{th} dimension is calculated by determining the difference between

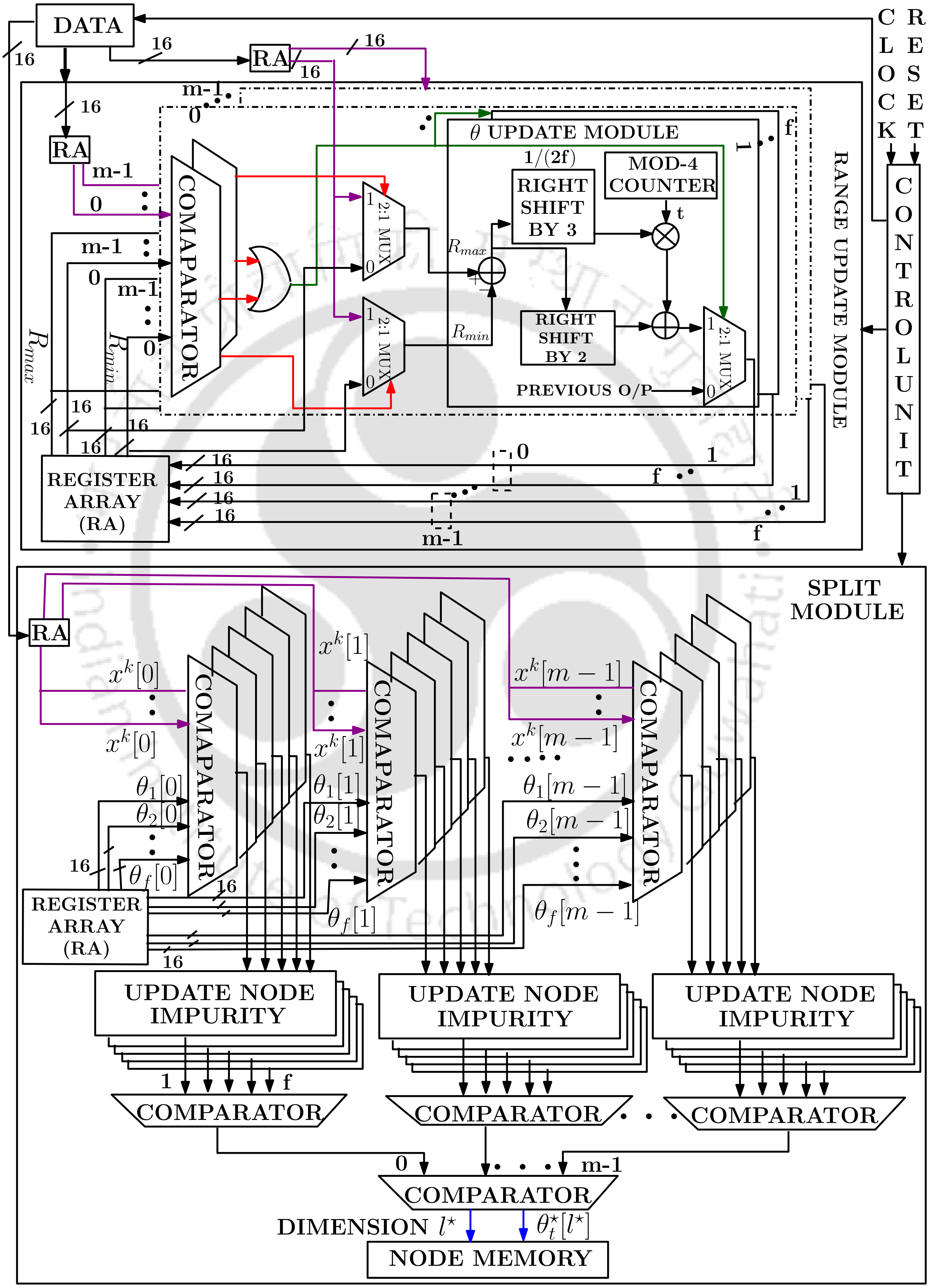

Figure 4.6: Hardware architecture for axis-aligned node module in HDT.

$R_{max}[l]$ and $R_{min}[l]$. Then, for calculating $\Delta[l]$, this difference is multiplied by $(1/2f)$ realised using 3 bit right shift (as $f = 4$) according to (4.2). The multiplication of range by 0.25 is realised using shifters as 0.25 can be represented as 1/4. Finally, $\theta_t[l]$ is calculated by multiplying Δ with t and adding it with the output of shifter according to (4.3). A MOD-4 counter with initial state set to 1 (as $t = 1, \ldots, f$) is used to generate the values of t. The final output is selected by the multiplexer whose select line is connected to the output of OR gate. The output of comparator is 1, if $a > b$ else, 0 for $a \leq b$. Thus the output of comparator is 1 in two conditions: if $x^k[l] > R_{max}[l]$ or if $R_{min}[l] > x^k[l]$. If any one or both of these conditions is true then, $R_{max}[l]$ or/and $R_{min}[l]$ changes as the data is out of range. The calculation of these ranges are done in parallel to optimize clock cycle consumption. Thus, if OR gate output is 1 then, $\theta_t[l]$ is updated with current value. Otherwise, if OR gate output is 0 then there is no update and previous value of $\theta_t[l]$ is retained. For l^{th} dimension, f θ UPDATE modules are run in parallel to calculate f number of θ values corresponding to range value of l^{th} dimension. For m-dimensional data, m θ UPDATE modules are used to calculate the $\theta_t[l]$ values for all dimensions in parallel where $t = 1 \ldots f$ and $l = 0 \ldots (m-1)$. Therefore, for m-dimensional data, $f \times m$ such modules are run in parallel to calculate $f \times m$ values of $\mathbf{\Theta}$ matrix. This results in reduced clock cycle consumption. Thus, in total, range update module calculates threshold matrix $\mathbf{\Theta}$ values of dimension $(f \times m)$ in parallel using shift and add operations according to (4.3) and stores these values in the RA. Likewise, entire data in that depth is executed in range update module of their respective nodes.

Once the range and $\theta_t[l]$ values are determined then, the split module of axis-aligned node is executed. In split module, data corresponding to a node is split into left and right child data by assigning a node number (present in next depth). Then, this node number is used in next depth to determine the node to which this data belongs. The split module has four levels. The first level of split module consists of f comparators for each dimension to compare the f number of $\theta_t[l]$ values with the $x^k[l]$ value in parallel where l is the dimension of $\mathbf{x}^k$. Again, for each value of f, m number of comparators are designed to compare all dimensions in parallel. Thus, this level of split module consists of $f \times m$ comparators running in parallel which splits the data according to threshold matrix $\mathbf{\Theta}$ values for that node. Then, in the second level, the impurity value associated to each $\theta_t[l]$ is calculated according to (4.10). Then in the third level, again these f impurity values for each dimension are compared to select the $\theta_t[l]$ value giving minimum impurity for l^{th} dimension. For this comparison, each dimension needs one f-input comparator to select the best $\theta_t[l]$ in l^{th} dimension. Then, in the

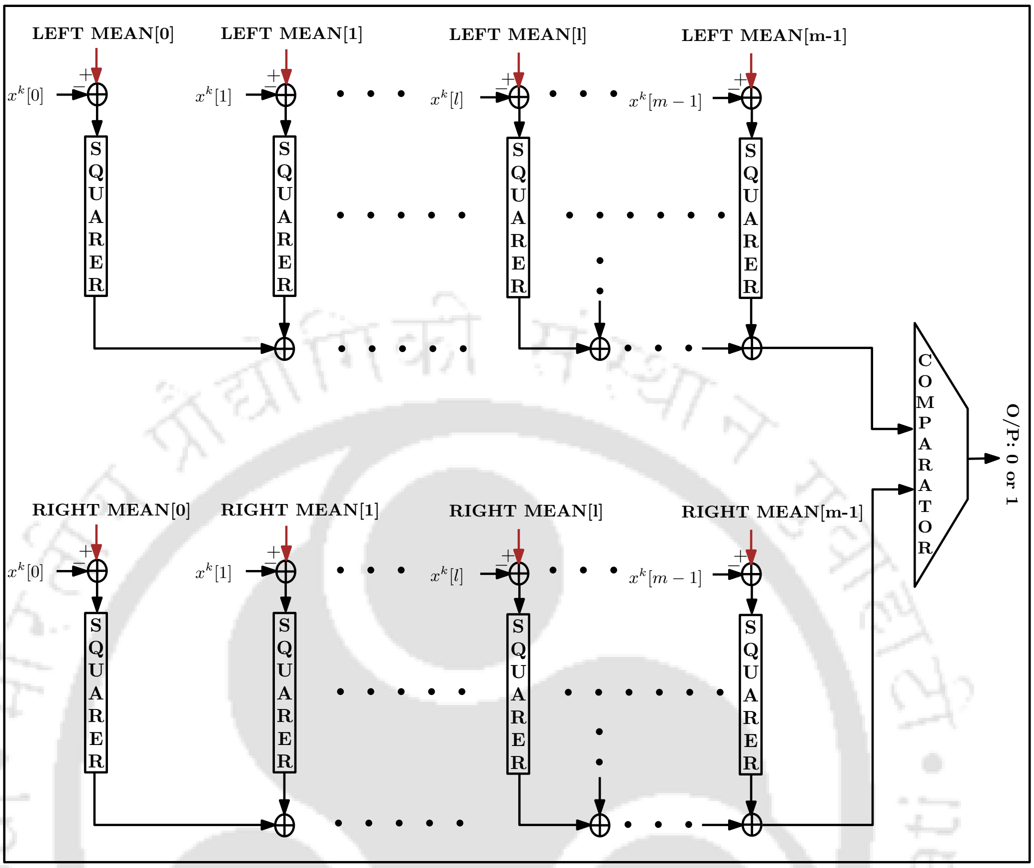

Figure 4.7: Hardware architecture for distance computation (DIST. COMP.) used in mean-dependent node module.

last level, finally one value of $\theta_{t^\star}[l^\star]$ is selected from the m number of $\theta_t[l]$ values. This selected $\theta_{t^\star}[l^\star]$ value is set as the best threshold value and the best l value corresponding to that $\theta_{t^\star}[l^\star]$ is chosen as the best dimension $(l^\star)$ giving global minimum impurity among $f \times m$ values as the split parameter for that node. For selecting these parameters, one m-input comparator is used in the last level. Similarly, data is processed for all nodes in that depth. This process continues till maximum depth of $(\zeta_{max} - 1)$ is reached. Then the leaf nodes present at depth ζ_{max} are updated with labels of the majority data the leaf node is exposed to. This completes the training of HDT.

4.3.5 Distance computation architecture

The mean-dependent node architecture uses euclidean distance computation to compute distance between left and right means and the data-instance $\mathbf{x}^k$ as shown in Fig. 4.5. The details of this DIST. COMP. architecture is shown in Fig. 4.7. The euclidean distance (represented by 32 bits to maximize accuracy) between the two means and $\mathbf{x}^k$ is computed in parallel and then the distances are compared using a comparator. The output of comparator is 1-bit, i.e., 0 (left mean is closer) or 1 (right mean is closer). This output of comparator decides whether left mean or right mean is closer to the data-instance. The hardware for squarer used in distance computation is shown in Fig. 3.4 as discussed in previous chapter.

4.3.6 Three stage pipelined distance comparator architecture

The architecture of DIST. COMP. includes distance computation of data instance $\mathbf{x}^k$ with left and right mean (as discussed in previous chapters) and the comparison of these two distances. The 32-bit distance comparison using 3-stage pipelined comparator is shown in Fig. 4.8 for two 32-bit inputs a and b. The pipeline registers are placed along points where the cut-sets intersect the nets. The comparator operation is divided into three stages. At first, each comparator input is divided into group of 8, each containing 4-bits for 32-bit input which is given as input to a 4-bit comparator. The pipeline stage-II starts by comparing the MSB first. The group of 4-bit MSB ($a_{31} - a_{28}$ and $b_{31} - b_{28}$) is checked first. The first multiplexer in stage-III checks the four MSB bit group. If all bits are equal, then the group of next 4-bits are checked. If these are also equal, then the next group is checked. Otherwise, if all bits are equal then next group is checked and it continues till the LSB bits. If the XNOR of all bits are equal then, output is 1 which indicates that both numbers are equal. Therefore, the output of the comparator is either 1 for $a_i \leq b_i$ or 0 if $a_i > b_i$. First stage comprises of 32 XNOR gates whose input is connected to i^{th} bit of a and b where $i = 0, ..31$. The next stage pipeline comprises of four 4-bit comparators and AND gates. The output of four XNOR gates are fed to the input of AND gate. The output of AND gate is 1 if all these 4-bits are equal, else it is 0. Each 32-bit number is divided into 8 groups of 4-bit numbers and each group of 4-bits are fed as input to each 4-bit comparator. As shown in figure, each 4-bit comparator consists of two levels of multiplexers (MUX). The level 1 consists of four multiplexers. The select line of each multiplexer is connected to an AND gate. The input to AND gate is $\overline{a_i}$ and b_i. The output of this AND gate is 1 if $a_i < b_i$ and 0 if $a_i > b_i$. If $a_i = b_i$, then the next bit is checked by the level 2 MUX. For 4-bit LSB group, the input of first multiplexer in level 1 is connected to the MSB in that group i.e., a_3 and b_3. If $a_3 > b_3$, then 0 is selected else, 1 is selected if $a_3 < b_3$. Otherwise, if $a_3 = b_3$ then, the first MUX in level 2 selects the output of second MUX in level 2 as shown in Fig. 4.8. Likewise, level 2 multiplexers selects 0 or 1 for all the 4-bits. In level 1, there are three multiplexers whose select line is connected to three AND gates as discussed above. If $a_3 = b_3$, then a_2 and b_2 are checked, else the output of level 1 multiplexer whose input is

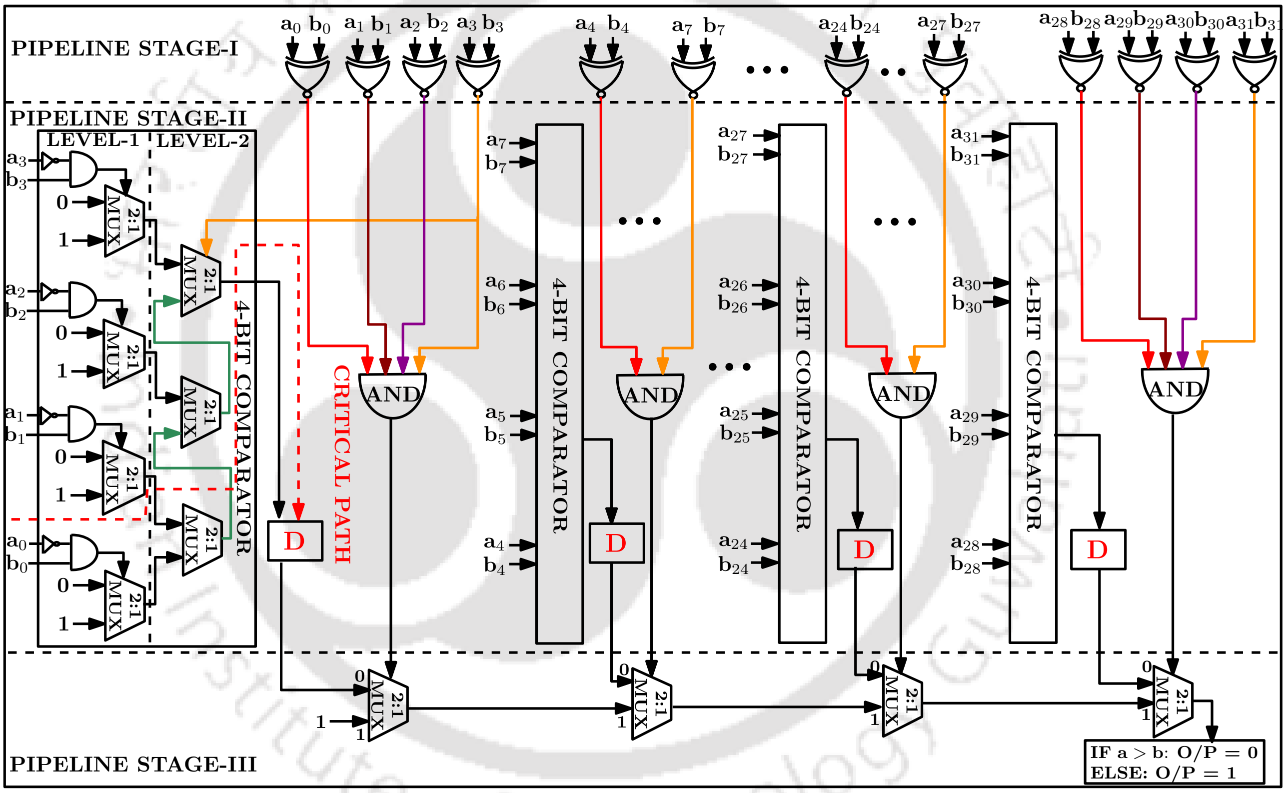

Figure 4.8: Hardware architecture for 32-bit pipelined comparator used in DIST. COMP. The cut-set is along the delay registers placed after each 4-bit comparator module.

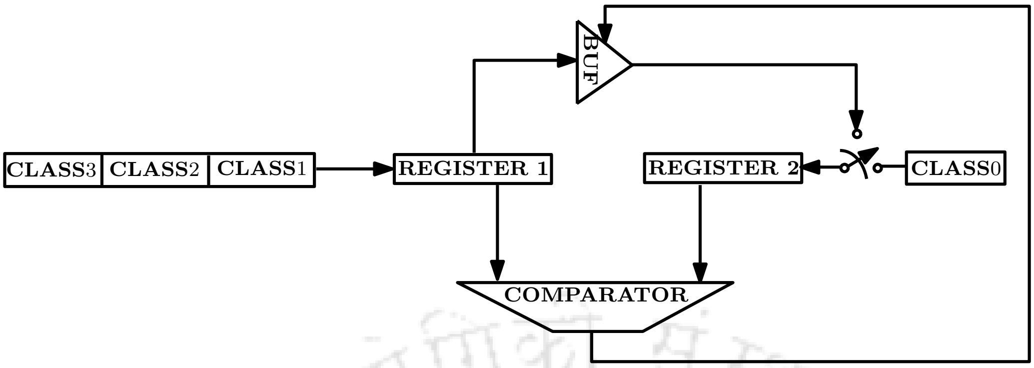

Figure 4.9: Hardware architecture for node label update

connected to a_3 and b_3 is selected. Similarly, if $a_3 = b_3$ and $a_2 = b_2$, then a_1 and b_1 is tested. Finally, if $a_1 = b_1$, then a_0 and b_0 is checked. If $a_0 > b_0$, 0 is sent to the output else, 1 is sent to the output of the comparator. The third stage consists of four multiplexers. The select line of each multiplexer is connected to the output of AND gate. If the AND gate output is 1, i.e., all 4-bits are equal, then the next 4-bits are checked. Otherwise, if AND gate output is 0, then the 4-bit comparator output is selected. Thus, the output of comparator is 0 if $a > b$ else, it is 1. The 4-bit comparator is not pipelined. The critical path is limited by the memory read delay, so pipelining the 4-bit comparator will not contribute to critical path delay reduction.

4.3.7 Hardware architecture of node label detector

The hardware architecture to update the label of leaf nodes is shown in Fig. 4.9. The number of data in a node belonging to class 0, 1, 2 and 3 is stored in CLASS 0, CLASS 1, CLASS 2 and CLASS 3 registers, respectively. Then, a comparator is used to compare two of the class data (stored in REGISTER 1 and 2) at a time. The output of comparator is used to control the tri-state buffer (BUF). If REGISTER 2 content is smaller than REGISTER 1 then, REGISTER 1 content is transferred to REGISTER 2, else REGISTER 2 content remains same. In first cycle, CLASS 0 register content is stored in REGISTER 2 and CLASS 1 content is stored in REGISTER 1. Then, once these CLASS 0 and 1 are compared then, CLASS 2 content is transferred to REGISTER 1. In this way, all the four class data are compared and the largest of four is stored in REGISTER 2. Accordingly, the label of that node is stored as 0, 1, 2 or 3 (as this hardware supports 4 classes) with respect to which ever class has highest data instances in that node.

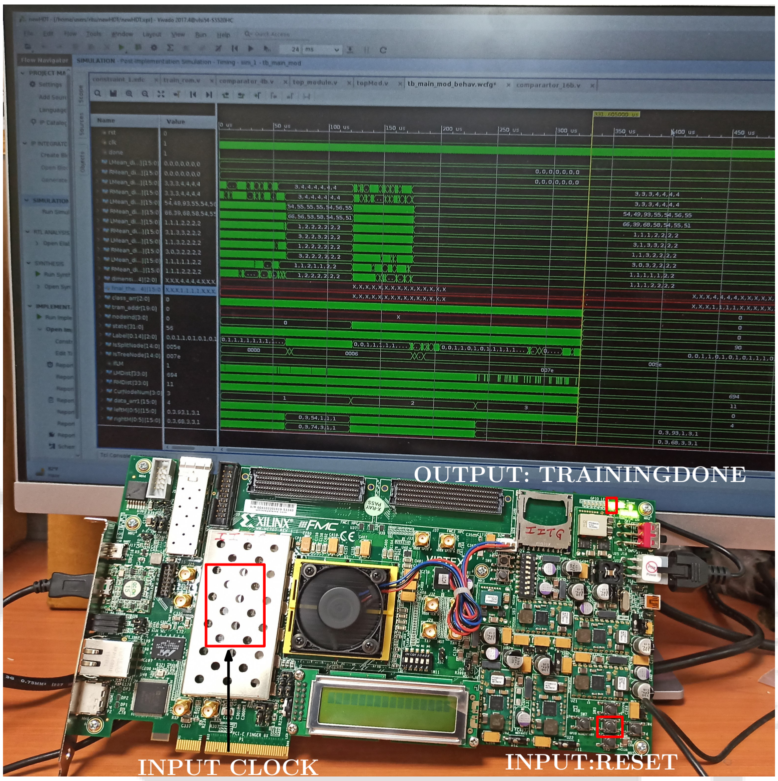

Figure 4.10: Experimental set-up for FPGA implementation.

4.4 Results

4.4.1 Experimental set-up and dataset description

The hardware implementation of this design was done on Virtex Ultrascale + board XCVU13P-FLGA2577-3-E using Verilog HDL in Vivado 2017 and no high-level synthesis tool was used. From implementation results, the critical path for proposed architecture is found to be 5 ns limited by the 4-bit comparator. Hence, FPGA can operate at a maximum frequency of 200 MHz (speed grade -3). The experimental set-up for on-chip implementation of HDT on FPGA is shown in Fig. 4.10. The clock input is connected to the system clock. The reset input is connected to the push button as shown in the figure. The output indicating the training completion is connected to the LED. The LED glows once the training is complete. The training data consisting of feature matrix is loaded into the on-chip memory (BRAM) from computer as binary file. During training, the data-instances are loaded into the training module from on-chip memory. The input RESET signal is given through the DIP switch present in FPGA board. If RESET is 1, the training hardware is initialised and training begins when RESET is 0. The training completion signal TRAINING DONE is connected

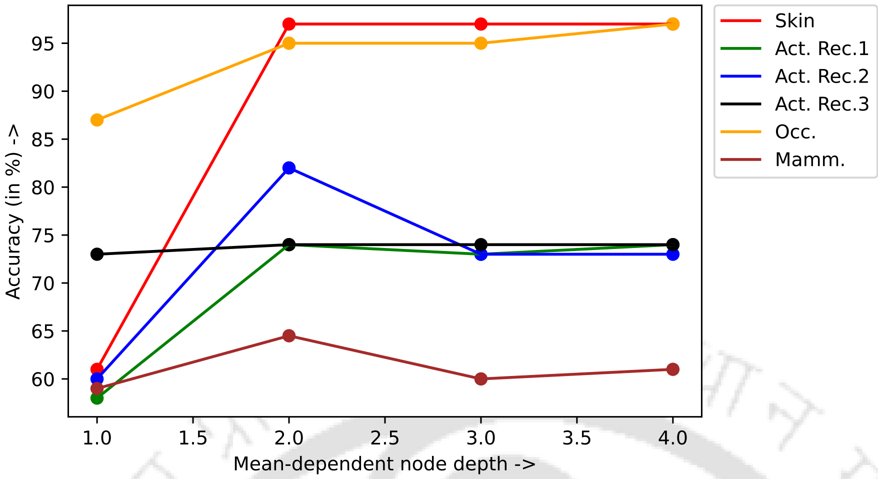

Figure 4.11: Plot showing the variation of accuracy with mean-dependent nodes depth (ζ_{md}).

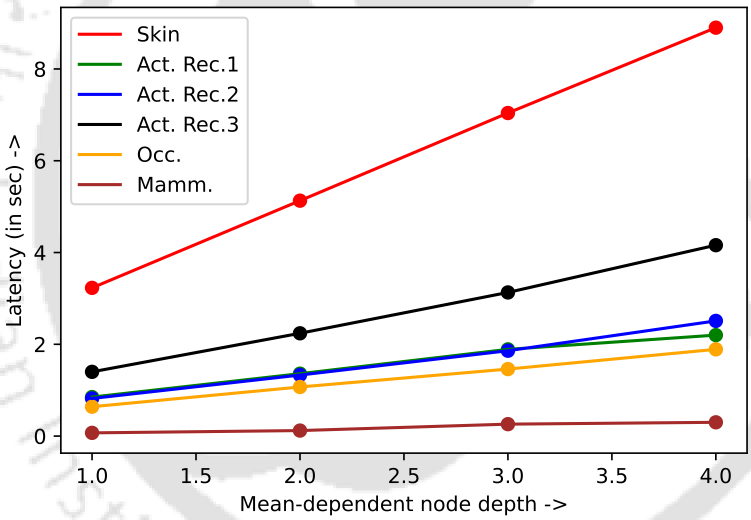

Figure 4.12: Plot showing the variation of training latency with mean-dependent nodes depth (ζ_{md}).

to an LED. Once the training is complete, TRAINING DONE signal output is asserted high and the LED glows. For software comparison, the design was implemented on both C and Python on Intel i7 processor running at 3.2 GHz having 6 cores. The design has been tested on 9 datasets, viz.,heart attack detetction, wine variety, dry beans variety, mammography, occupancy, activity recognition 1, activity recognition 2, activity recognition 3 and skin [40]. The training and testing data consisted of 80% and 20% of total data respectively. The heart attach detection is a binary dataset consisting of 13 features. The wine and dry beans variety datasets are multi-class consisting of 14 and 17 features, respectively. The *Mammography* or breast cancer dataset has 961 instances and 5 attributes, viz.,

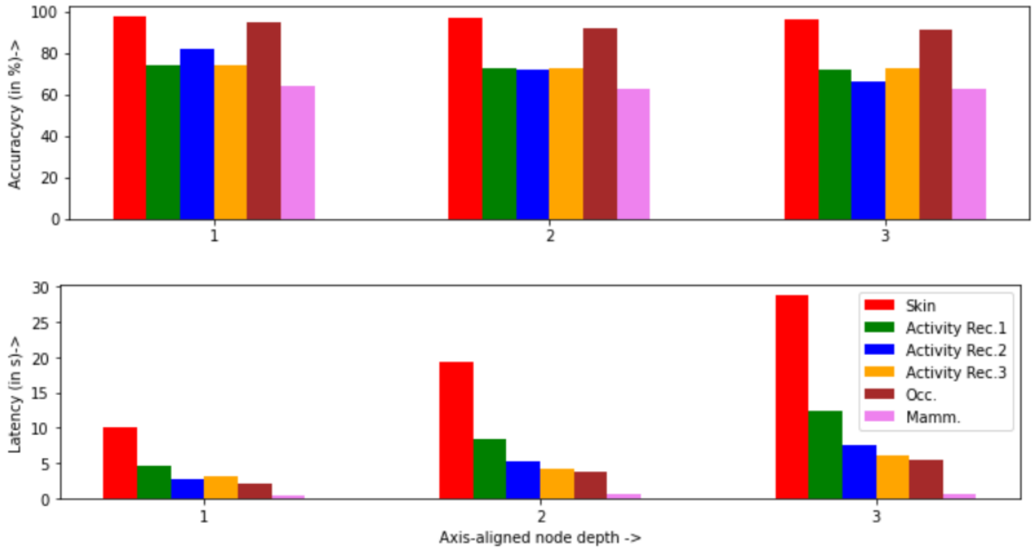

Figure 4.13: Bar-plot showing the variation of accuracy and training latency with axis-aligned nodes depth.

age, shape, margin, density, severity and a binary class number having value 0 for benign and 1 if it is malignant. The *Occupancy* has 12690 data-instances and each instance has 5 attributes, viz., age, shape, margin, density, severity and a binary label indicating whether the room is occupied or not. Next, the *Activity recognition1* has 6 attributes for each instance where each attribute contains the 6 sensor readings according to various movements of the body and a label for identifying the task performed. It consists of 25919 data instances. This dataset distinguishes between whether a person is lying, walking or standing. Similarly, *Activity recognition2* and *Activity recognition3* also has these 6 attributes. *Activity recognition2* containing 15497 data instances distinguishes between cycling and two types of bending. While *Activity recognition3* is a binary dataset having 17278 instances that distinguish between sitting and lying. Finally, the *Skin* dataset has 3 attributes related to each instance, viz., R,G,B pixel values with a binary label indicating whether it is a skin pixel. The maximum data-instance value in these datasets can be represented well with 16 bit representation. The largest dataset (*Skin*) consists of 62000 (50000 training data) data instances and the highest number of features ($m = 6$) is present in *Activity Recognition 1, 2, 3* datasets. So, this hardware supports maximum of 50000 data instances with 6 ($m = 6$) attributes and supported bit-size is 16. The number of features and bit-size can be extended based on the requirement of dataset without affecting the training latency.

Table 4.2: Table recording the training latency and accuracy comparison of conventional DT with HDT constructed using mean-dependent nodes and hybrid nodes

Dataset	**Conventional DT**		**HDT**			
	Axis-aligned nodes only		**Mean-dependent nodes only**		**Proposed hybrid nodes**	
	Time (in s)	Acc. (in %)	Time (in s)	Acc. (in %)	Time (in s)	Acc. (in %)
Mammography	0.92	85	0.15	67	0.83	64.50
Occupancy	30	99	1.11	87	5	95
Activity recognition 1	63	80	2.50	62	10	74
Activity recognition 2	35	91	2.40	58	7.42	82
Activity recognition 3	33	82	2.33	71	5.78	74
Skin	180	94	7	93	15.93	98

4.4.2 Experimental results

The performance of HDT is evaluated in terms of accuracy and latency. Fig. 4.11 shows the plot for variation in accuracy with varying depth of mean-dependent nodes (ζ_{md}) for all datasets. It is observed that the accuracy either remained constant or dropped after a depth of 2. Similarly, the latency variation with an increase in depth of mean-dependent node is plotted in Fig. 4.12. As can be observed from the plot, the latency increases with an increase in depth. So, $\zeta_{md} = 2$ is found to be the optimum value for these datasets. Similarly, performance variation is evaluated for various values of axis-aligned node depths. The accuracy and latency plot for various depths of axis-aligned nodes are shown in Fig. 4.13. It is observed that there is no accuracy improvement with increase in depth of mean-dependent nodes. While accuracy is almost constant for all datasets, there is a significant increase in latency with increase in depth. So, the optimum depth for axis-aligned nodes is found to be 1. The mean-dependent node depth is set as 2 and axis-aligned node depth is set as 1. So, the optimum value of $\zeta_{max} = 4$, as depth 4 contains leaf node (root node being at $\zeta = 1$). As discussed in previous chapters, the trial-and-error based software simulations shows that the maximum accuracy is obtained for $\alpha = 0.875$, $\rho_c = 128$ and $f = 4$. The training latency and accuracy comparison of conventional DT (having only axis-aligned nodes) with HDT constructed with only mean-dependent as well as hybrid nodes (both mean-dependent and axis aligned) is presented in Table 4.2. It is observed that HDT is faster as compared to conventional DT for all medium and large-sized datasets. Similarly, accuracy difference of HDT and conventional DT for small-sized binary dataset is higher than medium-sized binary dataset. The accuracy is lowest in case of smallest binary *Mammography*

Table 4.3: Data-size description and F1-score of all datasets. The binary classes do not have F1-score (class 2).

Dataset	Training instances	No. of features	F1-score (class0)	F1-score (class1)	F1-score (class2)
HeartAttack	216	13	0.61	0.5	-
Mamm.	769	5	0.70	0.61	-
Occ.	9727	5	0.96	0.93	-
Activity recog.3	14398	6	0.77	0.70	-
Skin	50000	3	0.98	0.98	-
Wine	116	14	0.84	0.71	0.44
Activity recog.1	21599	6	0.76	0.78	0.82
Activity recog.2	12959	6	0.86	0.50	0.86
Dry beans	3981	17	0.97	0.96	0.71

Table 4.4: Table recording the training latency of software (Python and C) and hardware (FPGA) platforms for all datasets with number of classes.

Dataset	Python (in s)	C (in s)	FPGA (in s)	No. of class
Mamm.	0.83	0.009	0.00016	2
Occ.	5	0.180	0.00209	2
Activity recognition1	10	0.310	0.00464	3
Activity recognition2	7.42	0.200	0.00279	3
Activity recognition3	5.78	0.240	0.00309	2
Skin	15.93	0.650	0.01075	2

dataset. The accuracy is also observed to be higher for binary dataset as compared to multi-class for similar-sized dataset. While HDT has accuracy of 98% for the largest binary *Skin* dataset. This proves that the accuracy of HDT improves with increase in instances from each class.

The dataset size and the F1-scores of each dataset is recorded in Table 4.3. HDT shows lowest F1-score of 0.44 for multi-class small-size dataset, i.e. wine as number of instances belonging to class 2 was less. The F1-score is 0.5 for class 1 of medium-sized Activity recognition2 as the number of instances belonging to class 1 is less than class 0 and 2 for this dataset. The *Skin* dataset shows the highest F1-score of 0.98 among all datasets. So, F1-score of HDT also improves with increase in the dataset size.

The latency comparison of proposed FPGA implementation with corresponding software implementations based on C and Python is presented in Table 4.4. Table 4.4 also provides information on number of classes required for each dataset. The FPGA implementation was found to be at-least 27× faster than C and 2500× faster than the Python implementation. The hardware training for medium-sized binary dataset *Activity recognition 3* was found to be 47× and 1150× faster than C and Python implementations, respectively. The speed-up for medium sized multi-class datasets *Activity recognition 1 and 2* was found to be at-least $30-37\times$ compared to C and $1100-1200\times$ compared to Python-based implementation. The FPGA implementation for largest binary *Skin* dataset was found to be 34× faster then the C implementation and almost $10^3\times$ faster than the Python implementation without incurring any performance degradation. Next, the latency of proposed hardware is compared with our previous works and the multi-FPGA implementation proposed in [20] which uses five FPGAs. The multi-FPGA implementation is found to be slower than the proposed single FPGA-based implementation due to involvement of complex operations such as sorting and division. As listed in Table 4.5, it is observed that the proposed single-FPGA implementation for 6 features and 4 possible value of class is found to be at-least 80× faster as compared to multi-FPGA implementation proposed in [20] for $10k$ data-instances having 32 features and 16 possible class values. This design processes all dimensions of a data-instance in parallel, so the training latency of the proposed design is independent of number of features and classes. Although, the resource usage will increase with increase in number of feature or class. The proposed hardware has at least 1.5 ms less latency than our previous works proposed in [19, 42] for $10k$ data-instances. This design has reduced clock cycle consumption due to parallel implementation of some blocks which has led to overall latency improvement. Also, this design supports four classes. This design also has same delay as pipelined implementation of TMDT proposed in [41] for $1k$ data-instances, the latency is reduced by 200μs for $10k$ data-instances. Thus, this hardware performs better as dataset size increases. The resource comparison of proposed hardware with existing classification hardware as well as training hardwares is shown in Table 4.6. This resource consumption is independent of board. The proposed training accelerator consumes lesser LUT and FF as compared to conventional k-means classification architecture [43]. It is also found that the proposed training hardware consumes very less DSP as compared to the optimized classification hardware reported in [33] implemented on Xilinx Virtex 7 FPGA running at 200 MHz. In spite of using more BRAMs, the proposed design has lower LUT and FF usage as compared to TMDT training hardware proposed in [19]. The proposed hardware is also resource-efficient compared to pipelined training hardware reported in [41] and batch-mode hardware proposed in [42]. Though, other resource

Table 4.5: Table recording the training latency comparison of proposed hardware with existing hardwares along with number of FPGAs used, number of features in each dataset and number of supported classes.

Design	Latency for 1k data (in ms)	Latency for 10k data (in ms)	Training algorithm	No. of FPGAs	No. of features	No. of supported classes
CART [20]	260	360	CART	4	32	16
TMDT Serial [19]	0.49	6.40	TMDT	1	5	2
TMDT Mixed [41]	0.29	3.76	TMDT	1	5	2
TMDT Batch-mode [42]	0.99	6.85	TMDT	1	7	4
Proposed Design	0.16	2.79	HDT	1	6	4

Table 4.6: Table recording the resource utilization comparison of proposed hardware with existing hardwares

Design	LUT (in K)	FF (in K)	BRAM	DSP	Max. freq. (in MHz)	Task
Conventional k-means	108	54	100	1062	200	Inference
Optimised k-means [33]	14.16	24.49	240	186	200	Inference
TMDT Serial [19]	297.91	3.42	48	200	62	Training
TMDT Mixed [41]	298.96	3.57	187	60	71	Training
TMDT Batch-mode [42]	154.65	5.44	24.5	0	62	Training
Proposed Design	5	3.03	30.5	0	200	Training

usage is less compared to the existing hardwares, the BRAM usage for proposed design is higher due to increase in dataset size. In this design, the intermediate register arrays have been implemented using BRAMs to free-up the important LUTs and FFs for use by classification block. This has led to increase in BRAM consumption. Also, to retrieve all features in parallel multiple instances of BRAM has been used which has lead to further increase in BRAM usage.

4.5 Discussions

In this chapter, HDT algorithm was proposed which was found to be faster than the conventional DT algorithm as tested on the Python platform. Moreover, the proposed architecture implemented on FPGA operates at a maximum frequency of 125 MHz. The proposed FPGA implementation also shows a speed-up of at-least 34× as compared to software implementation on C. Hence, the FPGA implementation executes faster than the software implementation for the same number of data instances. Further, it was found to be at least 80× faster than the existing FPGA DT training accelerator. It also has less memory and computational resource consumption as compared to optimized classification hardware. The proposed design also shows a manifold reduction in hardware usage as compared to existing training accelerators. Thus, the training hardware for HDT was found to execute faster than the existing training hardware. It was also found to be more resource-efficient as compared to prediction or classification hardware. This speed-up and resource efficiency were achieved due to a reduction in complexity as compared to the DT algorithms used in the existing classification or training hardware.

This thesis has explored the implementation of DTs with only mean-dependent split nodes and their combination with axis-aligned split nodes. We propose to further explore the hardware realization of neural trees. In neural trees, the split nodes host oblique split functions. Such trees are often found to provide greater accuracy with a lesser number of nodes. The simplest neural tree uses a perceptron at each split node. The next chapter describes the hardware based training and evaluation of neural trees.

5

PDT: Perceptron Decision Tree

- R. Choudhury, S. R. Ahamed, and P. Guha, "Hardware Implementation of Low Complexity High-speed Perceptron Block", in IEEE International Symposium on Circuits and Systems (IEEE-ISCAS), 2022.

Contents

Overview

The Neural Trees (NTs) are known for providing higher accuracy with a lesser number of split nodes. In such trees, the split nodes generally host a neural network. This work realizes a neural tree with single output perceptrons at each split node. The perceptron evaluation involves an inner product followed by a non-linear activation function (here, sigmoid) computation. In this chapter, a resource-efficient perceptron hardware is proposed. The inner product computation is implemented using OBC which uses simple adders and shifters. The sigmoid calculation has been implemented using the CORDIC algorithm for efficient resource usage and improved accuracy. Next, the Perceptron Decision Tree (PDT) algorithm is presented where each DT split node hosts a perceptron. A training hardware for PDT on FPGA is proposed. The training hardware is found to be more resource-efficient than our previous works. However, the accuracy of training hardware is found to be less than the software platform. Thus, a classification hardware implemented on both FPGA and ASIC is proposed where PDT trained on the software platform is loaded into hardware for classification. This hardware is found to be resource-efficient than the existing DT classification hardwares.

5.1 Problem formulation

The perceptrons are the main building block of Convolutional NNs or DNNs, etc. So, this chapter first proposes a resource-efficient and fast hardware for perceptron. This perceptron hardware implements the inner product computation unit and activation function unit using OBC and CORDIC, respectively. The OBC implements the inner-product computation using shifters and adders instead of multipliers for lower complexity. Next, this perceptron hardware is used to build a low-power and resource-efficient Multi-Layer Perceptron (MLP) network. Then, the PDT algorithm is proposed. A split node in PDT consists of a single order neuron. Due to the presence of perceptron or neuron in each node, the DT is able to achieve better accuracy at smaller depths as compared to TMDT and HDT and thus, results in an efficient hardware implementation. This chapter presents a training accelerator for PDT. The training here is done in batches. Next, classification hardware for PDT (trained offline) is also proposed.

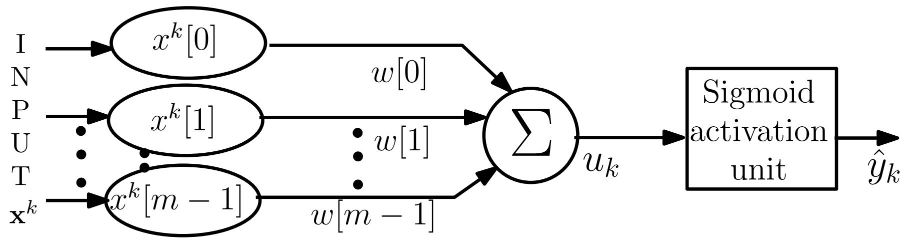

Figure 5.1: Structure of a perceptron showing the operations for k^{th} instance of data matrix $\mathbf{X}$ ($\mathbf{x}^k$)

5.2 Perceptron evaluation

5.2.1 Perceptron algorithm

A perceptron processes the input $\mathbf{x}^k \in \mathbb{R}^m$ in two steps. In the first step, the intrinsic weight parameters $\mathbf{w} = [w[0], \ldots w[l], \ldots w[m-1]]^T$ are point-wise multiplied with the input to compute the inner product u_k according to (5.1). In the second step, the inner-product value u_k is input to the activation function $g_a(\bullet)$ to obtain the perceptron output $\hat{y}_k$ as shown in (5.2). The computation diagram of the perceptron is shown in Fig. 5.1. Here, the perceptron bias is set to 0 for optimizing hardware consumption while achieving a trade-off between performance and resource utilization.

$$u_k = \mathbf{w}^T \mathbf{x}^k = \sum_{l=0}^{m-1} w[l] x^k[l] \tag{5.1}$$

$$\hat{y}_k = g_a(u_k) = \frac{1}{1 + e^{-u_k}} \tag{5.2}$$

Consider the training dataset $\mathbf{X} = \left\{\mathbf{x}^k : y(\mathbf{x}^k) = c; \mathbf{x}^k \in \mathbb{R}^m; k = 0, \ldots (n-1)\right\}$ where class label $c \in \{0, 1\}$. The perceptron parameters are learned from the training dataset $\mathbf{X}$. This is achieved by minimizing total error $E_{\mathbf{X}} = \sum_{k=0}^{n-1} \left\{y(\mathbf{x}^k) - \hat{y}_k\right\}^2$ while tuning $\mathbf{w}$.

5.2.2 Proposed hardware architecture

5.2.2.1 Overall architecture

The perceptron unit described above is the main building block of NN. As the NN consists of huge number of single perceptron units, so the hardware implementation of these network results in heavy resource usage and thus, increased power consumption. The challenge in the hardware implementation of neuron is designing both resource-efficient and high-speed inner product calculation unit. So, in this design, the inner product calculation is designed using OBC unit. The conventional piece-wise linear and non-linear approximation techniques used for calculation of sigmoid function introduced high error and higher critical path delay due to frequent memory access. Thus, in the proposed hardware, sigmoid unit was designed using CORDIC algorithm which reduced the error and the critical path delay as shown in Fig. 5.2(a) where the CONTROL UNIT, formed by Finite-State Machine (FSM), controls the execution of OBC and sigmoid modules. To further reduce the critical path, the OBC and sigmoid modules are pipelined. The mathematical formulation of inner product calculation using OBC is described in the following section.

5.2.2.2 OBC Architecture

The l^{th} dimension of L-bit data-instance $x^k[l]$ to OBC unit is expressed according to (5.3) where $-x^k_{L-1}[l]$ represents signed bit of $x^k[l]$ and $\overline{x}^k_{L-1}[l]$ represents signed bit of $-x^k[l]$ (2's complement of $x^k[l]$).

$$\begin{aligned} x^k[l] =& 1/2(x^k[l] - (-x^k[l])) \\ =& 1/2(-(x^k_{L-1}[l] - \overline{x}^k_{L-1}[l]) + \\ & \sum_{i=1}^{L-1}(x^k_{L-1-i}[l] - \overline{x}^k_{L-1-i}[l])2^{-i} - 2^{(L-1)}) \\ =& 1/2(d_{L-1}[l] + \sum_{i=1}^{L-1} d_{L-1-i}[l]2^{-i} - 2^{-(L-1)}) \end{aligned} \tag{5.3}$$

Therefore the inner product calculation of u for m-dimensional data-vector $\mathbf{x}^k$ can be represented as (5.4) where $D_{extra} = \sum_{l=0}^{m-1}(w[l]/2)2^{-(L-1)}$ is a constant.

$$\begin{aligned} u = \sum_{l=0}^{m-1} w[l]x^k[l] =& (\sum_{l=0}^{m-1} d_{L-1}[l]w[l]/2) + \\ & (\sum_{i=1}^{L-1}(\sum_{l=0}^{m-1} d_{L-1-i}[l]w[l]/2)2^{-i}) \\ & - (\sum_{l=0}^{m-1}(w[l]/2)2^{-(L-1)}) \\ =& \sum_{i=0}^{L-1} D_{L-1-i}2^{-i} + D_{extra}2^{-(L-1)} \end{aligned} \tag{5.4}$$

Thus v^{th} $(v = L-1-i)$ bit of y, D_v can be expressed as (5.5) where $v = 0, \ldots L-2$ while the MSB

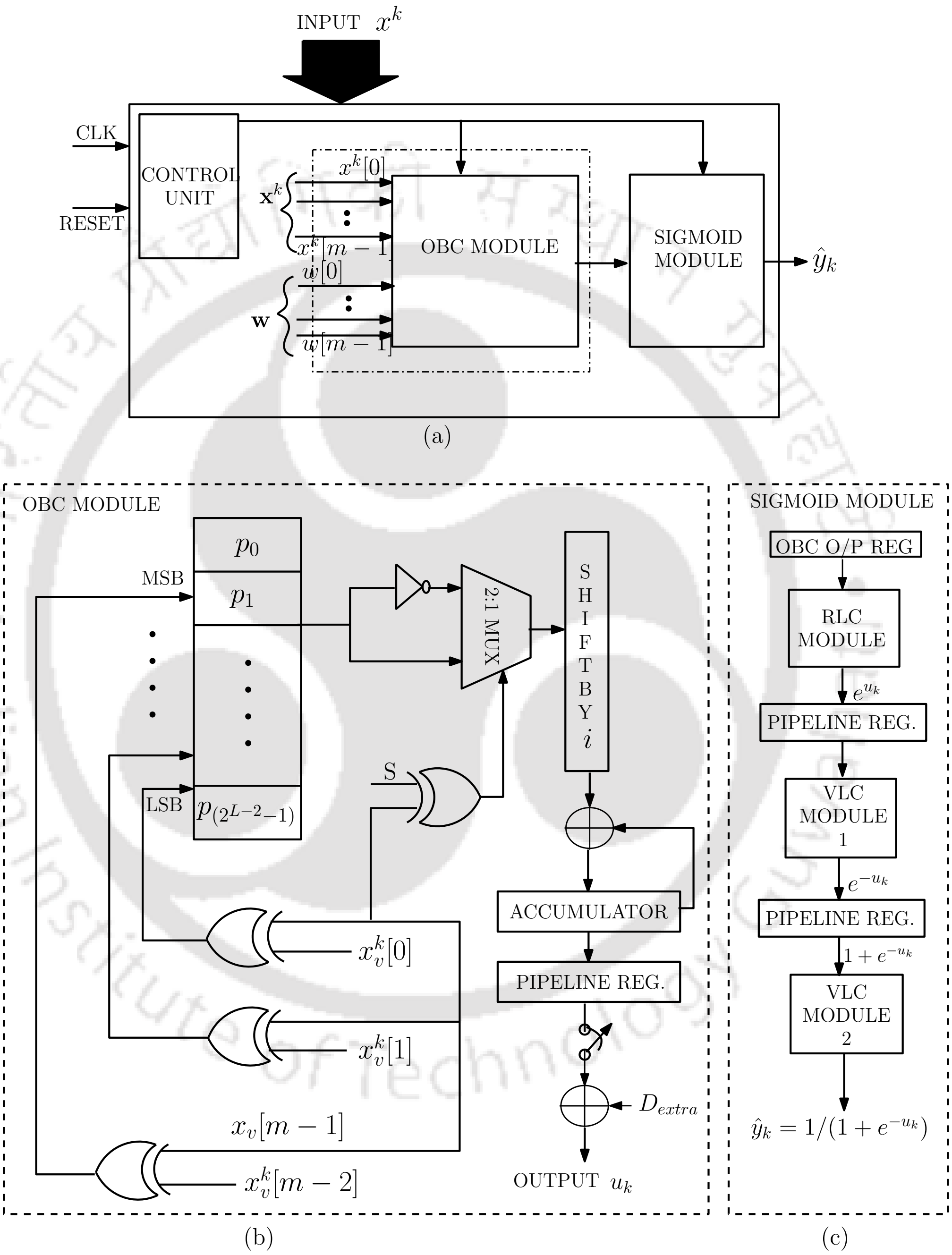

Figure 5.2: (a)Hardware architecture of perceptron block consisting of OBC and sigmoid module, (b) OBC module architecture, (c) CORDIC unit architecture.

bit D_{L-1} bit can be represented by replacing $[x_v^k[l] - \overline{x}_v^k[l]]$ with $[\overline{x}_v^k[l] - x_v^k[l]]$ in (5.5).

$$D_v = \sum_{l=0}^{m-1} [x_v^k[l] - \overline{x^k}_v[l]]w[l]/2$$
$$=(w[0]/2)[x_v^k[0] - \overline{x^k}_v[0]] + .. + (w[m-1]/2)[x_v^k[m-1] - \overline{x^k}_v[m-1]] \tag{5.5}$$

So, from (5.5) it can be observed that the sign of $w[l]/2$ corresponding to $x_v^k[l]$ is negative if $x_v^k[l] = 0$ or positive if $x_v^k[l] = 1$ which is reverse for D_{L-1} where $l = 0 \ldots m-1$. Therefore, D_v can be expressed as sum of weights $w[l]$ where the addition or subtraction of $w[l]$ is decided by the v^{th} bit $x_v^k[l]$. Thus, sum of weight values for every possible combination of $x_v^k[l]$ for $l = 0 \ldots m-2$ can be stored in an LUT. For v^{th} bit D_v, the $(m-1)$-bit address value A of LUT is $A = x_v^k[m-2] \ldots x_v^k[0]$ if $x_v^k[m-1] = 0$ while for $x_v^k[m-1] = 1$, the address is $A = \overline{x}_v^k[m-2] \ldots \overline{x}_v^k[0]$. This addressing reduces the memory-size of LUT by half. For $v = L-1$, i.e., D_{L-1}, a negative sign is inserted before LUT value if $x_{L-1}^k[m-1] = 1$, otherwise it is positive. For instance, for 7-dimensional input $\mathbf{x}^k$ $(m = 7)$, if $x_0^k[m-1] = 0$, the LUT value for D_0 is accessed at memory-location 000000 while the value corresponding to D_0, if $x_0^k[m-1] = 1$, is accessed at memory-location 111111. Further, for D_0, if $x_0^k[m-1] = 0$, the LUT values are complemented. Similarly, for D_{L-1}, if $x_{L-1}^k[m-1] = 1$, the LUT value is complemented and else it is positive. Thus, for determining D_v the address of LUT is realised using XOR gates whose one input is $x_v^k[m-1]$ to complement the address bits if $x_v^k[m-1] = 1$ as shown in Fig. 5.2(b). A multiplexer (2:1 MUX) is used to select the complemented LUT values or the original values. The select signal for the multiplexer is connected to XOR of S and $x_v^k[m-1]$, where S= 1 for $v = L-1$, i.e., D_{L-1} and 0, otherwise. Once the selected LUT values are obtained, the multiplexer output is shifted by i bits and added to the PIPELINE REG. content according to (5.4). The content of this register is updated after each clock cycle. An example showing the LUT values stored to determine the D_v bit for a 3 dimensional input $\mathbf{x}^k$, i.e., $L = 3$ is shown in Table 5.1 where $v = 0, \ldots L-2$ as for $v = L-1$, the sign is reversed. It can be observed that $p_3 = -p_4, p_2 = -p_5, p_1 = -p_6$ and $p_0 = -p_7$. Thus, the memory-size is halved by storing $p_0 \ldots p_3$ in the memory and $p_4 \ldots p_7$ are generated by taking 2's complement where the XOR of S and MSB bit of address,i.e., $x_v[2]$ (for $m = 3$) is used as the select signal. This process is repeated for $v = 0, \ldots 2$ (where $L = 3$) and added to previous PIPELINE REG. content by shifting the obtained LUT value by i bits. After all bits $D_0 \ldots D_{L-1}$ are obtained, then the register value is added to D_{extra} (controlled by a switch which closes after every L clock cycles) for L-bit input-vector $\mathbf{x}^k$.

Table 5.1: Table recording the LUT contents for 3 dimensional input vector $\mathbf{x}^k$ ($L = 3$) where the chosen LUT value is used to calculate D_v

LUT Address			**LUT stored value**
$x_v^k[2]$	$x_v^k[1]$	$x_v^k[0]$	D_v
0	0	0	$p_0 = (-1/2)(w[2] + w[1] + w[0])$
0	0	1	$p_1 = (-1/2)(w[2] + w[1] - w[0])$
0	1	0	$p_2 = (-1/2)(w[2] - w[1] + w[0])$
0	1	1	$p_3 = (-1/2)(w[2] - w[1] - w[0])$
1	0	0	$p_4 = (-1/2)(-w[2] + w[1] + w[0])$
1	0	1	$p_5 = (-1/2)(-w[2] + w[1] - w[0])$
1	1	0	$p_6 = (-1/2)(-w[2] - w[1] + w[0])$
1	1	1	$p_7 = (-1/2)(-w[2] - w[1] - w[0])$

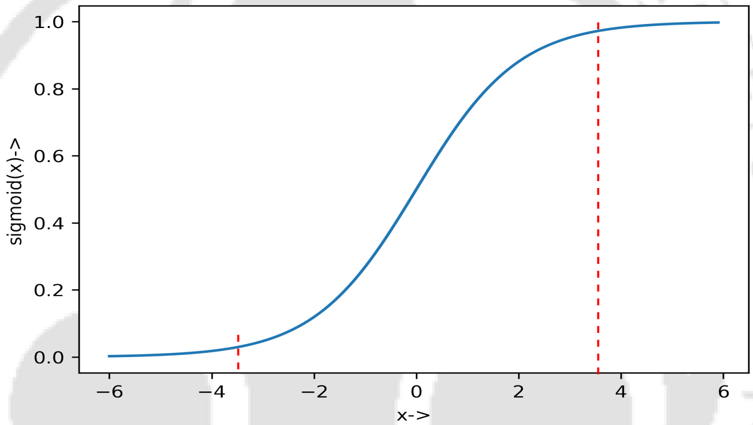

Figure 5.3: Plot showing the variation of sigmoid output with input **x**

As shown in Fig. 5.2(b), PIPELINE REG. is used to store the intermediate values before adding it to D_{extra} bit. The switch connecting the PIPELINE REG. content to the adder is connected after every L clock cycles to generate the final output u.

5.2.2.3 CORDIC architecture

The variation of the sigmoid graph is limited for a certain range of input as shown in Fig. 5.3. The maximum variation is observed in the range $[-3.5, 3.5]$. So, in this design, the input range of sigmoid hardware was divided into four sections as the output for sigmoid is almost 0 for input range $(-\infty, -3.5]$ and 1 for $[3.5, \infty)$. For input $\mathbf{w}^T\mathbf{x}^k = 0$, the output of sigmoid unit was set as 0.5, else for $\mathbf{w}^T\mathbf{x}^{\mathbf{k}} > 3.5$, output is set as 1 or if $\mathbf{w}^T\mathbf{x}^k < -3.5$ then output is set as 0. Otherwise, for $-3.5 < \mathbf{w}^T\mathbf{x}^k < 3.5$, the CORDIC unit is executed to get the output $\hat{y}_k$. The sigmoid activation function used in the perceptron unit can be implemented by using Rotation CORDIC (RLC) and Vector CORDIC (VLC) [38]. In this hardware, the RLC unit implements the exponential part ($e^{\mathbf{w}^T\mathbf{x}^k}$), and two VLC units are used to implement ($e^{-\mathbf{w}^T\mathbf{x}^k}$) and ($1/(1+e^{-\mathbf{w}^T\mathbf{x}^k})$) part as shown in Fig. 5.4. The sigmoid hardware error for input range $[-3.5, 3.5]$ was limited to 10^{-6}. The critical path delay due to CORDIC unit has been optimized by placing delay registers between the RLC and VLC 1 modules and between VLC 1 and 2 modules.

The RLC unit comprises of multiplexer and shifters for computation of exponential function. The input to this unit is $x_0 = 1/\Gamma$, $y_0 = 0$ and $z_0 = \mathbf{w}^T\mathbf{x}^k$. While the output is the value of $x_g + y_g$ taken after g iterations which corresponds to $e^{\mathbf{w}^T\mathbf{x}^k}$ where $\Gamma = \Pi_{s=1}^{g}\sqrt{1-2^{-2s}}$ and $s = -r, \ldots 0, \ldots g$. For $s = -1$ and $s = 0$, the values of Γ and the expressions in (5.6) changes as 2^{-s} is replaced by $1 - 2^{-2^{-s+1}}$. The equation for computing x_{s+1}, y_{s+1} and z_{s+1} in s^{th} iteration in RLC is described in (5.6) where $tanh^{-1}$ is pre-calculated and stored in LUT.

$$
\begin{aligned}
x_{s+1} &= x_s + sign(z_s)(2^{-s}y_s) \\
y_{s+1} &= y_s + sign(z_s)(2^{-s}x_s) \\
z_{s+1} &= z_s - sign(z_s)tanh^{-1}(2^{-s})
\end{aligned} \tag{5.6}
$$

The VLC unit also implements the complex inverse operation using simple LUTs, shifters and adders which makes it resource-efficient and fast. To compute $e^{-\mathbf{w}^T\mathbf{x}^k}$, the inputs to the VLC unit is set as $x_0 = e^{\mathbf{w}^T\mathbf{x}^k}$, $y_0 = 1$ and $z_0 = 0$. The value of z_t after p iterations is the required value. The computation of the values in t^{th} iteration is realised using (5.7) where $t = 1, \ldots, p$.

$$
\begin{aligned}
x_{t+1} &= x_t \\
y_{t+1} &= y_t - sign(y_t)(2^{-t}x_t) \\
z_{t+1} &= z_t + sign(y_t)(2^{-t})
\end{aligned} \tag{5.7}
$$

Finally, the output of VLC unit 2 is the predicted label $\hat{y}_k$ corresponding to input vector $\mathbf{x}^k$. The sigmoid module is pipelined to reduce the critical path delay. The output of RLC module and VLC unit 1 are stored in two pipeline registers as shown in Fig. 5.2(c). The highest accuracy is obtained for values $g = 15, r = -1$ and $p = 18$.

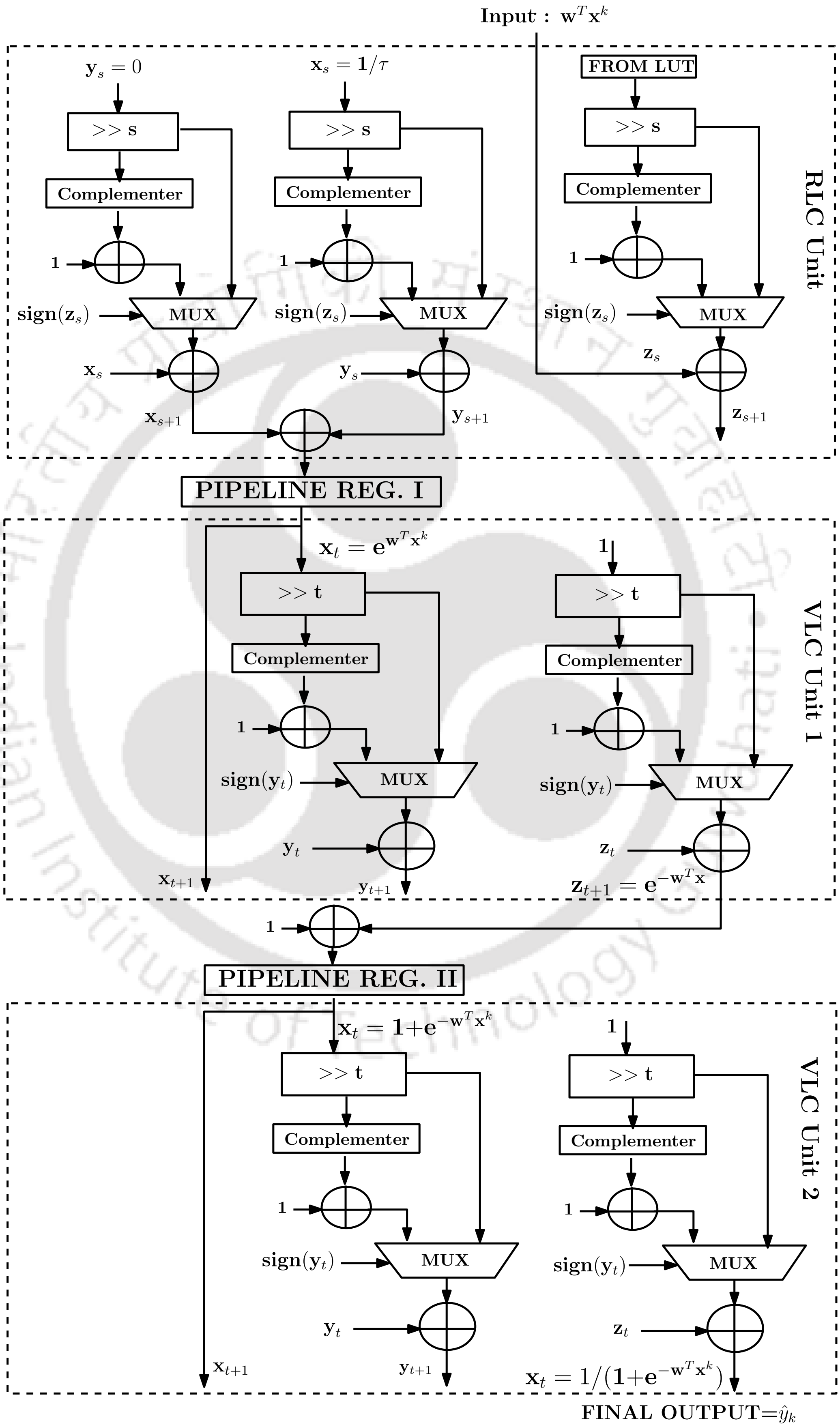

Figure 5.4: Hardware architecture of CORDIC

TH-3177_176102101

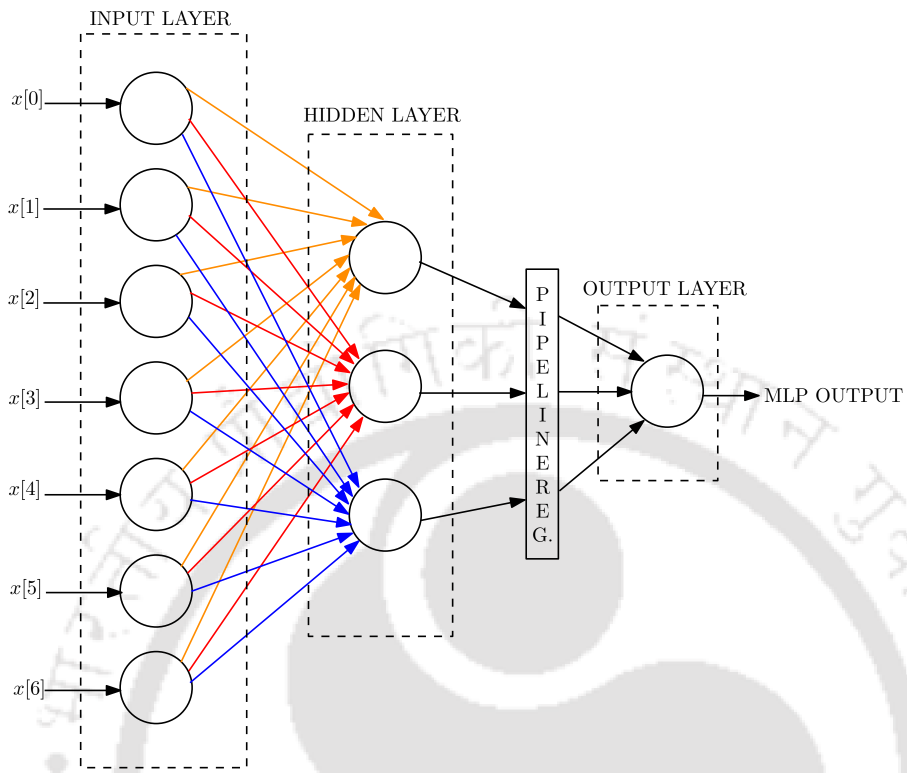

Figure 5.5: Hardware architecture for feed-forward MLP module constructed using proposed perceptron hardware.

5.3 Feed-forward MLP architecture designed using the proposed perceptron hardware

A feed-forward MLP network architecture was designed using this single perceptron module. The network had one input layer, one hidden layer and one output layer as shown in Fig. 5.5. The input layer had seven inputs (for 7-dimensional data), hidden layer had 3 perceptron units running in parallel, and one perceptron unit in the output layer. The output of perceptron in hidden layer is truncated to 16 bits before feeding into the next layer perceptron. To increase the speed, one pipelined register was placed between hidden layer and output layer. This resulted in increase in frequency thereby increasing the speed.

5.4 The PDT algorithm

5.4.1 Feature selection and data pre-processing

In this design, m_s features are selected from the feature set (of m dimensions) available in the training dataset $\mathbf{X}$ using Extra-Trees classifier [44]. This is a randomized approach, consisting of

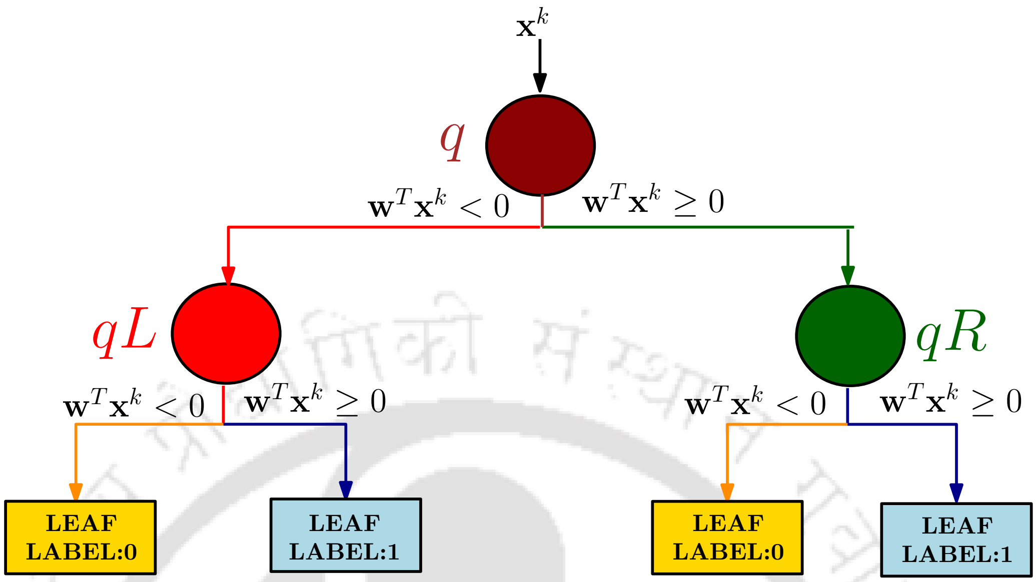

Figure 5.6: Structure of fully grown PDT till depth 3 is shown along with split conditions for data vector $\mathbf{x}^k$. The circles signify split nodes while the squares represent leaf nodes.

an ensemble of N_{DT} DTs to select the m_s best features. The input dataset to a DT $\mathcal{T}_F$ (where $F = 1, \ldots, N_{DT}$) consists of randomly selected feature subsets from a set of m features. For a DT $\mathcal{T}_F$, a split *score* (impurity criteria selected by user) is computed for all the features. The feature giving the best *score* is selected. This is repeated for all N_{DT} DTs. Finally, m_s features having top scores are selected.

Consider the weight vector $\mathbf{w} = [w[0], w[1], \ldots, w[m-1]]$. The input data-matrix $\mathbf{X} = \{\mathbf{x}^k : y(\mathbf{x}^k) = c; \mathbf{x}^k \in \mathbb{R}^{m_s}; \}$ where $c \in \{0, 1\}$ and $k = 0, \ldots, (n-1)$ has n instances (with binary category labels). In this algorithm, each data vector $\mathbf{x}^k$ is normalized using zero mean unity variance normalization. This normalization results in an increase in accuracy and thus improves the algorithm's performance.

5.4.2 Perceptron training

The training of perceptron involves node weight correction to tune the weights according to the training data space. First, the inner product $(\mathbf{w}^T\mathbf{x}^k)$ is calculated. Then, the sigmoid function $(\sigma(\mathbf{w}^T\mathbf{x}^k))$ is calculated for all data-vectors $\mathbf{x}^k$. The output of the sigmoid block is the predicted class $(\hat{y}_k)$ corresponding to $\mathbf{x}^k$. Then, $\delta w[l])$ corresponding to l^{th}-dimension of data-vector $\mathbf{x}^k[l]$ is calculated according to (5.8) where $l = 0, \ldots, (m_s - 1)$ and $n_0^q + n_1^q$ is the total data from class 0 and 1, respectively, which visited node q. In this way, the vector $\delta\mathbf{w} = [\delta w[0], \delta w[1], \ldots, \delta w[l], \ldots, \delta w[m_s - 1]]$

corresponding to $\mathbf{x}^k$ is calculated.

$$\delta w[l] = \frac{\sum_{k=0}^{b-1} \hat{y}_k\{1-\hat{y}_k\}\{y(\mathbf{x}^k)-\hat{y}_k\}\mathbf{x}^k[l]}{n_0^q + n_1^q} \tag{5.8}$$

The perceptron is trained in batch mode and multiple epochs are executed to improve its accuracy. An epoch is formed by reshuffling the data instances present in the previous epoch. This design uses $N = 10$ epochs which are observed to maximize the accuracy. The training data-matrix $\mathbf{X}$ is decomposed into H batches ($H = \left\lfloor \frac{n}{b} \right\rfloor$, where b is the batch-size) in each epoch. Each batch $\mathbf{X}_h$ ($\mathbf{X} = \bigcup_{h=1}^{H} \mathbf{X}_h$) contains b training instances ($\mathbf{X}_1 \cup \mathbf{X}_2 \cup \mathbf{X}_3 \ldots \mathbf{X}_H \in \mathbf{X}$). For batch-mode training, the error is averaged over the entire batch h and the node weight is updated at the end of execution of each batch h according to (5.9) where $\delta\mathbf{w} = [\delta w[0], \delta w[1], \ldots, \delta w[l], \ldots, \delta w[m_s - 1]]$ and γ is the weight update factor. The value of γ is set as 0.1 after trial and error giving maximum accuracy. Here, bias is set as 0 to optimize the hardware consumption and to achieve a trade-off between performance and resource utilization.

$$\mathbf{w}^h = \mathbf{w}^{(h-1)} - \gamma\delta\mathbf{w} \tag{5.9}$$

5.4.3 PDT training flow

A PDT induces oblique splits in the input space in contrast to axis-aligned ones induced by algorithms like CART, C4.5 or ID3. To perform classification, the PDT needs to be trained first. The structure of a fully grown PDT is shown in Fig. 5.6. Each node of the PDT consists of a perceptron. Here, each batch of data instances ($\mathbf{X}_h$, $h = 1, \ldots H$) is passed through the perceptron embedded in a node. The processing of $\mathbf{X}_h$ starts with the root node. At root node, $\mathbf{X}_h$ is divided into datasets $\mathbf{X}_h^{qL}$ and $\mathbf{X}_h^{qR}$. Next, $\mathbf{X}_h^{qL}$ and $\mathbf{X}_h^{qR}$ are sent to the child nodes qL (node 1) and qR (node 2), respectively. The perceptrons in nodes qL and qR are trained accordingly. Then, $\mathbf{X}_h^{qL}$ and $\mathbf{X}_h^{qR}$ are again split in nodes qL and qR respectively and sent to their respective child nodes. Thus, the training data batch $\mathbf{X}_h$ enters through the root node and its partitions traverse the entire PDT till the last depth.

Let $\mathbf{X}_h^{(q)} \subset \mathbf{X}_h$ be the dataset input to the q^{th} node of the PDT. The weight vector $\mathbf{w}_q$ associated with node q is tuned using $\mathbf{X}_h^{(q)}$ according to (5.9). After tuning, $\mathbf{X}_h^{(q)}$ is split following (5.10) and each data-instance $\mathbf{x}^k \in \mathbf{X}_h^{(q)}$ is sent either to left (qL) or right child node (qR) of q.

$$\text{Left child data} : \mathbf{X}_h^{qL} = \left\{ \mathbf{x}^k : \mathbf{w}^T\mathbf{x}^k < 0 \right\}$$
$$\text{Right child data} : \mathbf{X}_h^{qR} = \left\{ \mathbf{x}^k : \mathbf{w}^T\mathbf{x}^k \geq 0 \right\} \tag{5.10}$$

Similarly, the perceptrons of all nodes are trained. This continues till depth $(\zeta_{max} - 1)$ as leaf nodes in depth ζ_{max} only consist of node labels. Once batch $\mathbf{X}_h$ finishes training of all nodes till depth $(\zeta_{max} - 1)$, then, the next batch $\mathbf{X}_{h+1}$ is loaded. Then, similarly, all split node weights are tuned till depth $(\zeta_{max} - 1)$. This continues for all batches and epochs. The pruning is performed after the completion of training with all batches. For a split node q, the node data comprises of a number of data vectors having label 0 (n_0^q) and label 1 (n_1^q) which entered node q. In pruning, the same batches are again passed to update the node data. In the post-pruning phase, the node status is updated. This node data is used for setting node q either as leaf node (if pruning conditions as given in (5.11) are satisfied) or as a split node where ρ_c is the data-length threshold and η_p is the node purity threshold.

$$\text{Total Node data} : n_0^q + n_1^q < \rho_c$$
$$\text{Purity of node} : \left\{ \frac{max(n_0^q, n_1^q)}{n_0^q + n_1^q} \right\} > \eta_p \tag{5.11}$$

This process is repeated for all epochs and the final tree structure after pruning is stored to be used during classification.

5.4.4 PDT classification flow

In the classification phase, the input data is routed through the root node to the leaf node by the perceptron embedded in the split node according to (5.10). Then, the class label is assigned in the leaf node. When a data instance $\mathbf{x}^k$ enters the node, the perceptron first performs inner product calculation between $\mathbf{x}^k$ and the node weight vector $\mathbf{w}$. Next, if $\mathbf{w}^T\mathbf{x}^k < 0$, then $\mathbf{x}^k$ is sent to the left child, otherwise to the right child. This is continued till the data vector reaches a leaf node where it is assigned a label (0 or 1). This design implements a binary PDT.

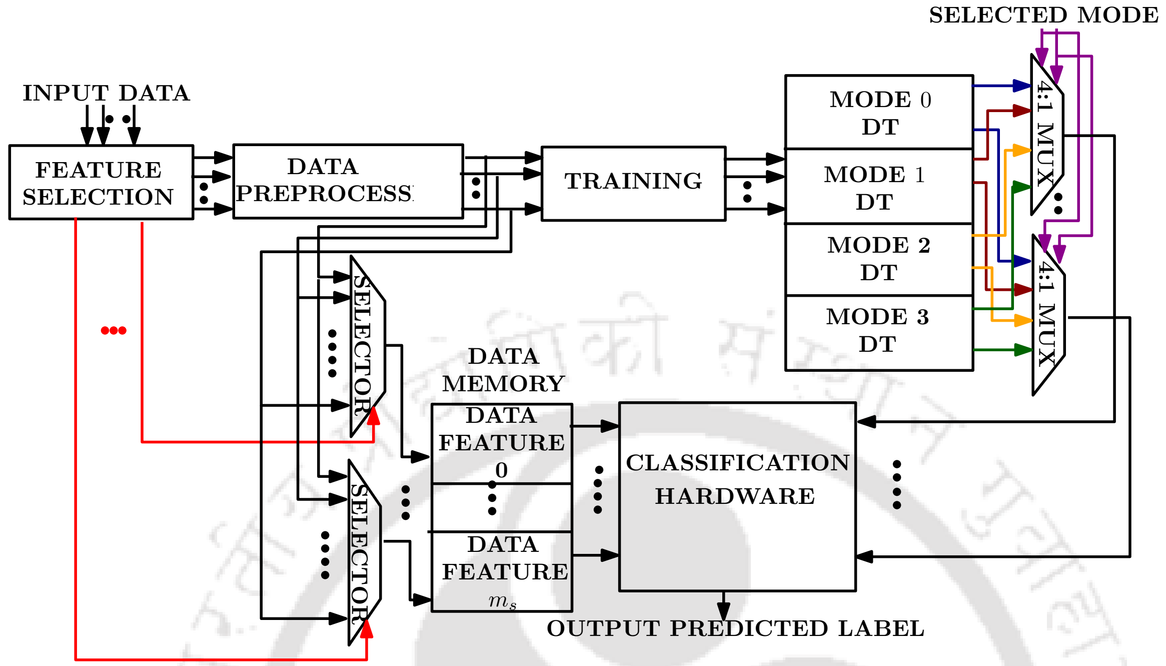

Figure 5.7: Block diagram showing the flow of steps starting from feeding the input raw data till detection in PDT. The DT structure output corresponding to different modes is shown in different colors.

5.5 Proposed PDT hardware architecture

5.5.1 Proposed PDT training hardware

5.5.1.1 Overall architecture

A resource-efficient binary training hardware is proposed. The block diagram showing the operations involved in the entire process starting from feeding the input data to getting the predicted class is shown in Fig. 5.7. In this architecture, simplified calculation of weight update and inner product units requires only adders and shifters. So, implementing these modules in parallel will not incur a high cost. However, it will save the time otherwise required for serially processing the features. The resource consumption is already found to be very less as compared to complex hardware like multipliers. So, the architecture is implemented in parallel to achieve a trade-off between resource utilization and latency. The flow starts with the feeding of input data to the feature selection block which selects the m_s best features and their indices are sent to the on-chip hardware to be used during classification. The selected data-feature matrix is then given as input to the data pre-processing block. The output of this block is used as input data for training as well as classification. After completion of the training process, the updated weight vectors of all nodes are stored in register arrays. The proposed training hardware for PDT is shown in Fig. 5.8. The perceptron embedded in each node of PDT is trained with

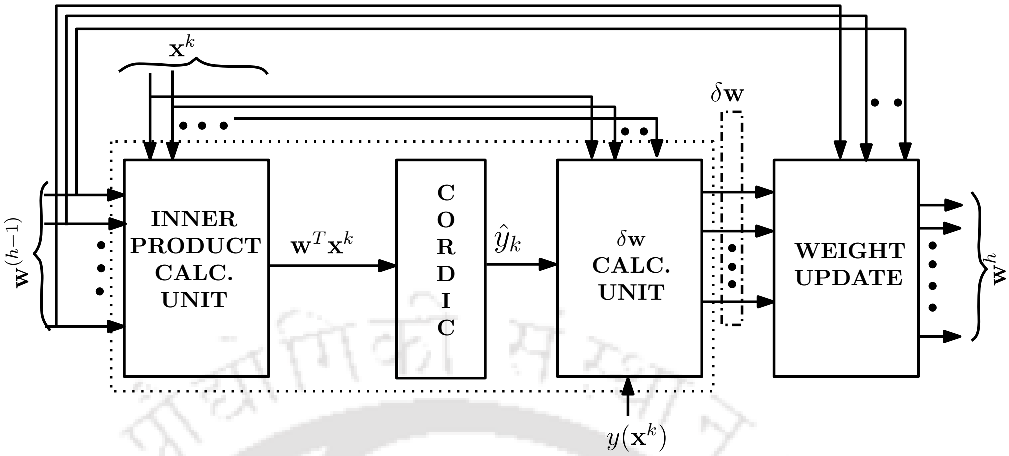

Figure 5.8: Hardware architecture for PDT training

data batch $\mathbf{X}_h$. The training starts from node 0 (root node). For h^{th} batch of data, the weight $\mathbf{w}^{(h-1)}$ obtained from previous batch is tuned with data-instances from $\mathbf{X}_h$. The training of perceptron in a node q starts with inner product calculation between the data-vector $\mathbf{x}^k$ and weight vector $\mathbf{w}^{(h-1)}$. This product is given to the CORDIC unit. The output of CORDIC unit is the predicted class $\hat{y}_k$. This predicted class ($\hat{y}_k$) and actual class ($y(\mathbf{x}^k)$) are then used to calculate $\delta\mathbf{w}$ for $\mathbf{x}^k$. Similarly, $\delta\mathbf{w}$ calculated over entire batch h is summed up and stored in a m_s-dimensional register. After entire batch is passed, then, the weight vector $\mathbf{w}^{(h-1)}$ is updated to obtain $\mathbf{w}^h$. In this way, all the weight vectors of split nodes till depth $(\zeta_{max} - 1)$ are updated. Once the update for batch h is complete then batch $(h + 1)$ is loaded and a similar procedure is followed. Once training of weight vector $\mathbf{w}$ is completed for all batches and epochs, then the DT is pruned. The PDT training hardware is trained using small batches of data (batch-size is 32) which has resulted in a resource-efficient design. Also, the hardware is designed using only shifters and adders which has led to a further reduction in hardware usage.

5.5.1.2 Inner product calculation hardware architecture

The inner product calculation unit in a perceptron is the most resource-consuming unit as it involves multiplication. In this design, we have used normalized data for training as well as classification. The normalization converts the input data of the four applications used in this design into integers in the range $[-7, 7]$. Thus, the multiplication of the weight vector with the input data vector can be realized by using shifters and adders as demonstrated in Fig. 5.9. The input data range can be represented by 4 bits (1 sign bit and 3 magnitude bits) while the weights are represented as fixed-point

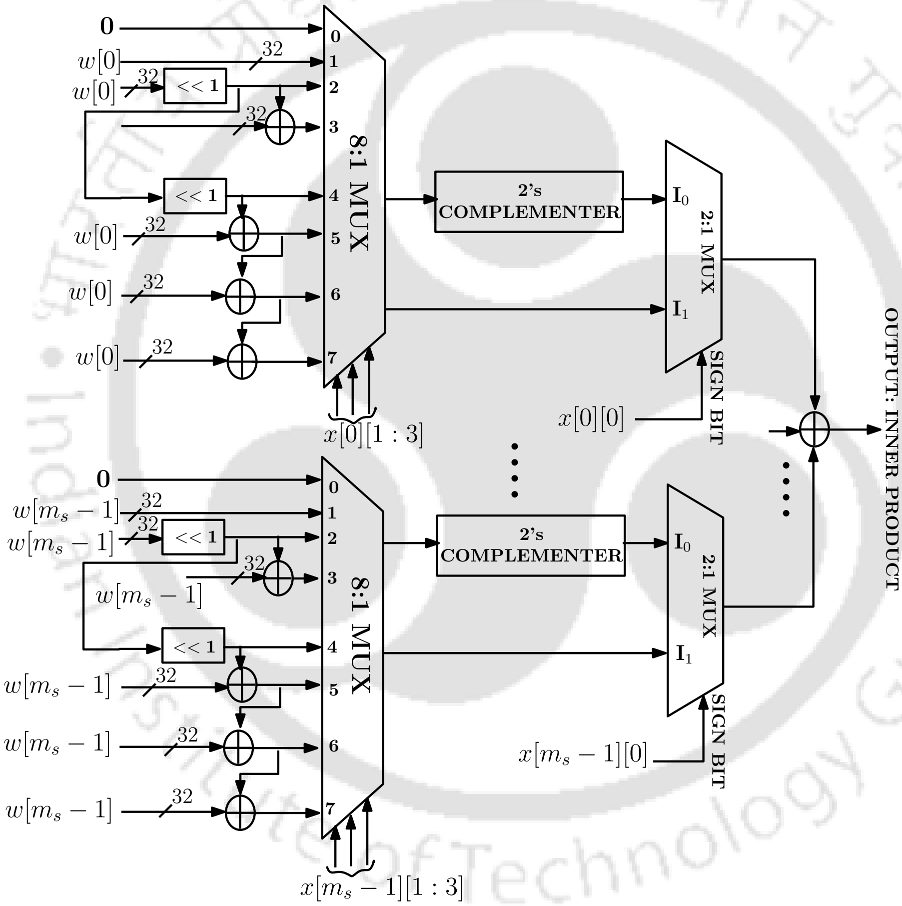

Figure 5.9: Hardware architecture of simplified inner product calculation unit

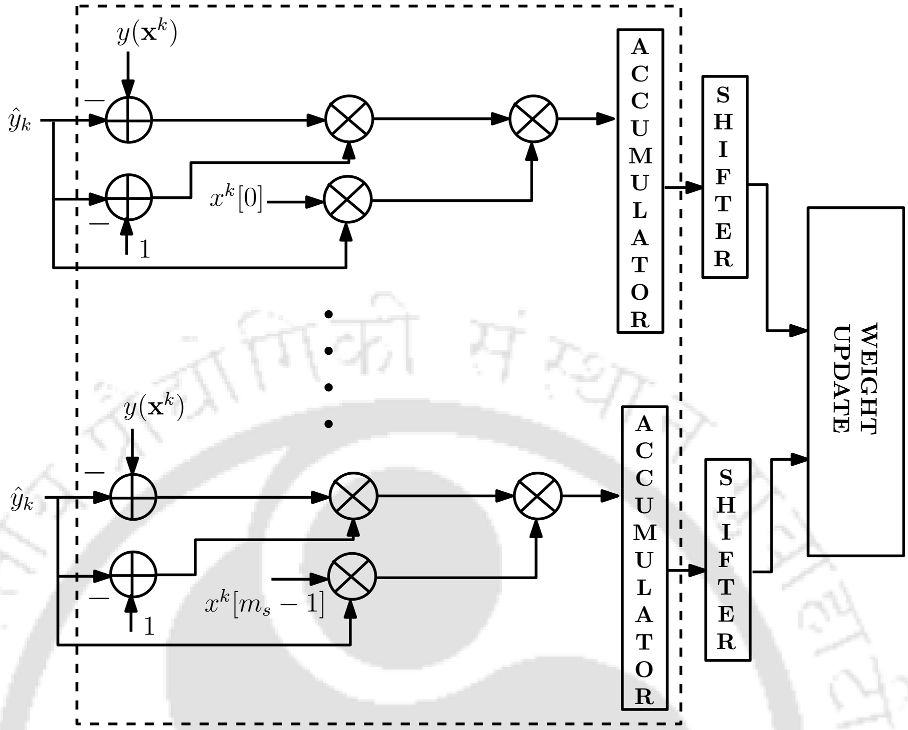

Figure 5.10: Hardware architecture for weight update.

numbers with 1 sign bit and 31 fractional bits. So, the 3 data bits ($x[l][1:3]$ where $l = 0, \ldots, (m_s - 1)$) are used as the select line for the $8:1$ multiplexer which selects the appropriate output. Then, this output is sent to the next multiplexer whose one input is the twos complement form of the output and another input is the output itself. So, the select line of this $2:1$ multiplexer is connected to the sign bit of the input data ($x[l][0]$). It selects the 2's complemented form if the sign bit is 1, i.e., input data is negative otherwise, it selects the original output. Let us consider, the input data is -2, then input data bits will be 1010. The input will be shifted by 1 unit and the final output will be the two's complement of the shifted data. Further, the number of shifters and adders has been reduced by enabling hardware re-use. For instance, multiplication by 2 and 4 requires 1 and 2 shifts, respectively. So, this hardware connects the 1-bit shifter used for multiplication by 2 to a 1-bit shifter to obtain the multiplication by 4 result. Thus, this helps in reducing hardware usage to a great extent. This calculation unit is used for all features. Therefore, m_s such units run in parallel, and their results are added to finally obtain the inner product at the output. This module is a simple circuit and running m_s such modules in parallel does not incur huge resource consumption.

5.5.1.3 Weight update hardware architecture

The weight update unit used in training hardware is explained in detail in Fig. 5.10. For a data vector $\mathbf{x}^k$, $\delta\mathbf{w}$ is calculated and stored in the accumulator as per (5.8). This involves multiplication and addition of the predicted class, actual class, and the data vector. For a node q, the final $\delta\mathbf{w}$ for batch h is calculated by dividing the accumulator output by the number of data vectors entering the node q, i.e., $(n_0^q + n_1^q)$. This division is realized by shifters. Then, the weight $\mathbf{w}$ of node q is updated at the end of each batch h as per (5.9).

5.5.2 Proposed PDT classification hardware

5.5.2.1 Overall architecture

In this section, classification hardware is proposed where training is done on software and the trained weights are stored in Distributed RAM (DRAM) available on-chip. The feature selection, data pre-processing, and training blocks are implemented on the software platform. The on-chip hardware implementation starts with the SELECTOR which selects the data features from the DATA MEMORY using the feature indices obtained from the feature selection block as shown in Fig. 5.7. These selected feature indices are fed as the select line to the SELECTOR block. To reduce latency, m_s such SELECTOR blocks are operated in parallel in this hardware which selects and stores the selected features in DATA MEMORY. This classification hardware supports four different applications. So, m_s number of $4:1$ multiplexers are used to select the DT structure according to the selected application mode. First, the m_s-dimensional weight vector of the root node corresponding to the selected DT is loaded. After the root node weight vector is processed, then the next node weight vectors are loaded. Similarly, these multiplexers are used to obtain the weight vectors for all the split nodes. This weight vector and the input data vector are then fed to the classification block which gives the predicted class label as the output.

5.5.2.2 Hardware architecture of memory storing the DT structures for all the modes

The memory architecture which stores the DT structure for all four operational modes is shown in Fig. 5.11. This memory is constructed from DRAM. The DRAM is constructed from register arrays. The advantage of using DRAM is that multiple locations can be accessed in parallel. Moreover, the memory access latency is also less than the conventional BRAM. The only drawback is that DRAM uses the LUTs available on-chip instead of the dedicated memory resources. In the proposed architecture,

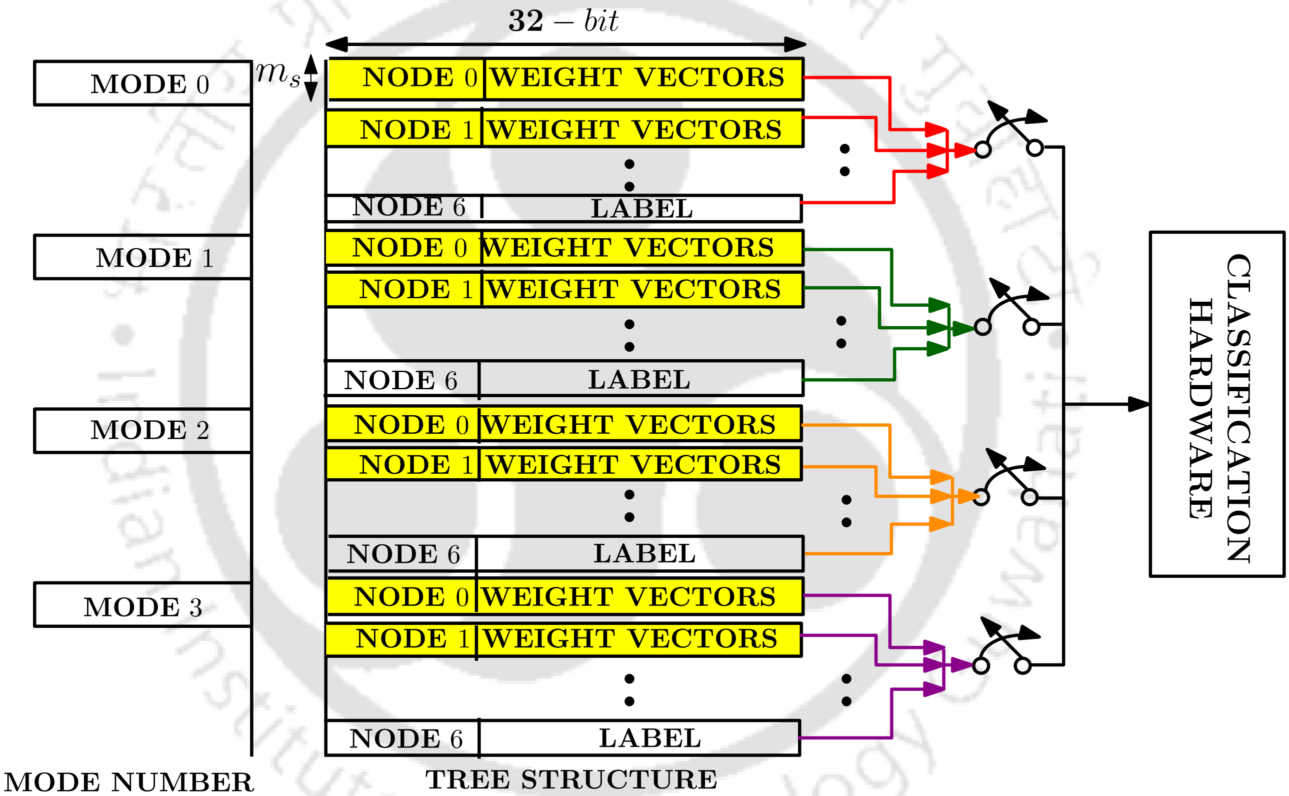

Figure 5.11: Hardware architecture of DRAM storing the DT structures for the four application modes. The width of memory is 32-bits as each weight is a 32-bit. The memory locations in yellow denote a depth of m_s as a split node weight is a m_s-dimensional vector.

due to resource-efficient design, a large number of LUTs are left unused. So, in this architecture DRAM has been used for reduced latency and efficient use of resources. This RAM has a width of 32 as each weight value is 32-bit. For each mode, the locations storing the split node (node $0-2$) weights have a depth of m_s (shown in yellow in Fig. 5.11) as the weight vector is m_s-dimensional. For leaf nodes, only the node labels are stored and so they have a depth of 1. The maximum depth of DT in this design is set to be $\zeta_{max} = 3$ (root node being at depth 1) so the maximum number of nodes in this DT is 7 ($2^{\zeta}_{max} - 1$). For each application mode, the entire DT structure is stored in consecutive locations. Each location consists of a node weight value for split nodes (till depth 2 for this design) or node labels for leaf nodes (in depth 3). Due to the use of DRAM, multiple memory locations of the node memory can be accessed at the same time. So, the m_s-dimensional node weight vector for a particular mode is loaded into the CLASSIFICATION HARDWARE in parallel. As shown in Fig. 5.11, four switches are used to load the DT structure into the CLASSIFICATION HARDWARE. These switches are controlled by the 4 : 1 MUX whose select line is connected to the input mode number (SELECTED MODE) as shown in Fig. 5.7. The switch corresponding to the selected mode is closed and the DT structure corresponding to the mode number is loaded into the REGISTER ARRAY as shown in Fig. 5.12.

5.5.2.3 Detailed implementation of classification hardware

The detailed architecture of the classification block is illustrated in Fig. 5.12. This classification hardware classifies the input data and assigns a label that is given at the output of the block. The classification hardware consists of a CONTROL UNIT which comprises an FSM controlled by CLOCK and RESET signal. This unit controls the counter and the functioning of the classification modules. Due to the top-down nature of DT, the node execution starts with the root node (node 0). So, initially, the NODE MEMORY address is set to 0. The node 0 weights and the input data vector stored in DATA MEMORY are sent to the INNER PRODUCT CALCULATION UNIT. The INNER PRODUCT CALCULATION UNIT calculates the inner product between the data-vector $\mathbf{x}^k$ and the weight vector $\mathbf{w}$ as discussed above. Then, the output of this unit is sent to the comparator to check if it is less than or greater than 0 as discussed previously. The data is sent to the LEFT CHILD node or the RIGHT CHILD node according to (5.10). Accordingly, the node number is updated in the NODE NUMBER register which is initialized to 0 (root node). Then, the depth is incremented by 1. This updated depth number is compared with the maximum depth ζ_{max}. The output of this

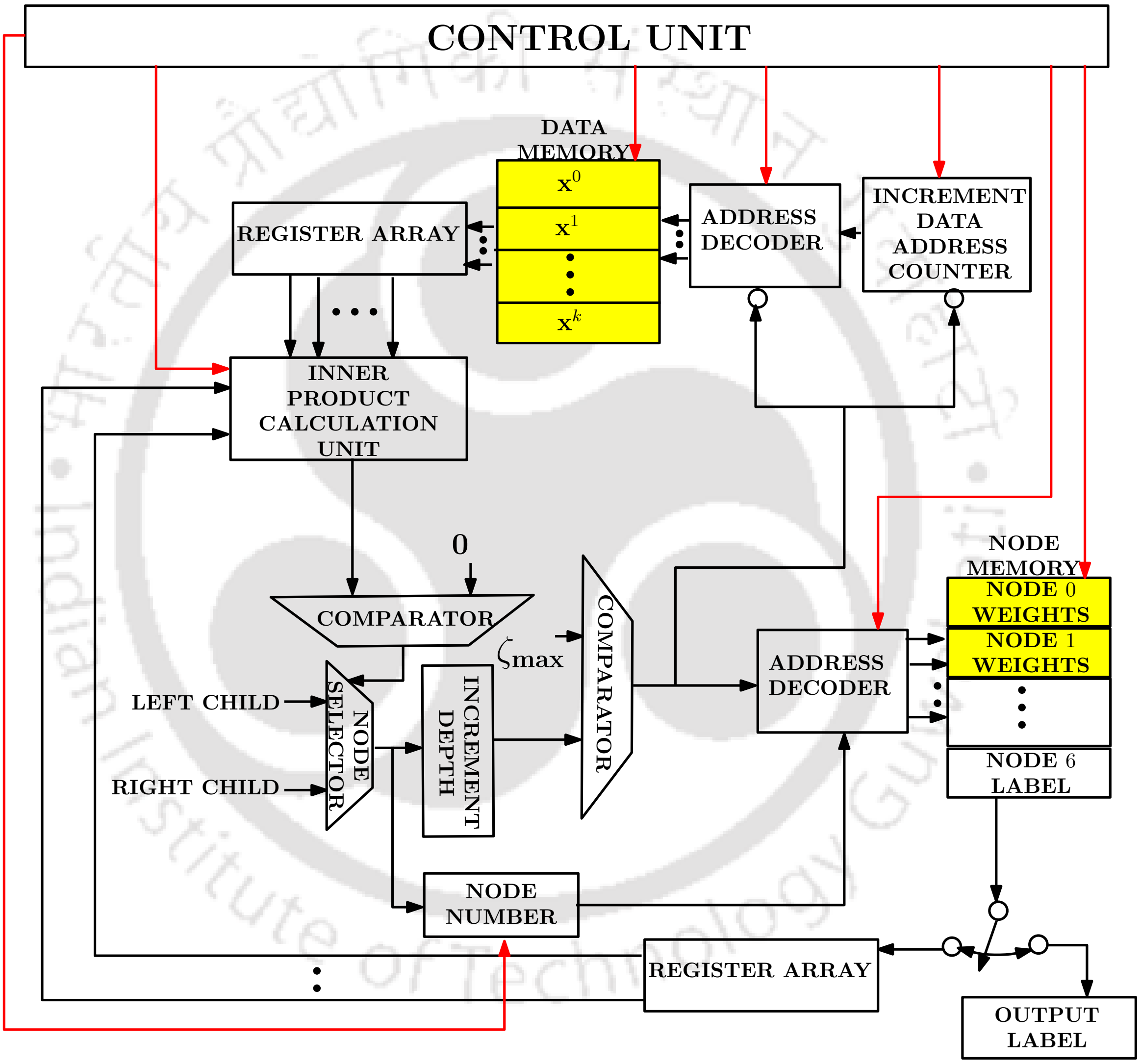

Figure 5.12: Detailed hardware architecture of proposed classification hardware. The red lines indicate the CONTROL UNIT signals. The memory locations in yellow denote a depth of m_s as a split node weight is a m_s-dimensional vector.

Table 5.2: Table recording the latency comparison on Python and FPGA for proposed perceptron hardware

Batch-size	Latency(in ms)		Speed-up
	Python	FPGA	
128	9.59	0.162	55×
256	10.11	0.322	31×
512	14.7	0.643	21×
1024	29.64	1.283	22×

comparator is 0 if the depth is greater than ζ_{max}, else, it is 1. If the maximum depth is less than ζ_{max}, then, the address decoder connected to NODE MEMORY is activated. The switch connecting the NODE MEMORY to the REGISTER ARRAY is closed. The node weights of the next nodes are loaded into REGISTER ARRAY and sent to the INNER PRODUCT CALCULATION UNIT. Otherwise, if the maximum depth is reached, i.e., depth is equal to ζ_{max}, then the switch connecting the NODE MEMORY to OUTPUT LABEL is closed. In this way, the detected class of input data is sent to the output. In case the depth is greater than ζ_{max}, the address decoder connected to DATA MEMORY is activated and the node address is set to 0. The address of the next data is loaded from the counter DATA ADDRESS COUNTER and the desired data is sent to the REGISTER ARRAY for classification and the same process is followed. So, for classifying a data vector it is passed through the entire tree, starting from the root node till the leaf nodes. The label corresponding to the selected leaf node is the predicted class of the input data.

5.6 Results

5.6.1 Perceptron and MLP hardware

5.6.1.1 Experimental set-up

The hardware implementation of this design is done on Virtex Ultrascale+ board XCVU9P-FLGA2104-2LV-E-ES1 using Verilog HDL in Vivado 2017 and no high-level synthesis tool is used. For software comparison, the design is implemented in Python on Intel i7 processor running at 3.2 GHz and having 6 cores. The hardware is implemented for 16-bit fixed-point number, i.e., $L = 16$ with 1 signed bit and 15-bits for the fractional part as the input data to the OBC unit is normalized in the range $(-1, 1)$. The input data dimension is $m = 7$ and the root mean square error in the output is limited to around 0.11×10^{-6}. The total area and power consumption for the hardware are found to be $0.085662mm^2$ and 2.28 mW, respectively, for $65nm$ CMOS technology @$200MHz$. The latency comparison of proposed hardware on FPGA with the similar implementation on Python is reported

Table 5.3: Table recording the resource utilization comparison of proposed perceptron hardware realized using OBC with perceptron hardware realized using conventional multipliers.

Resources	**Multiplier architecture**	
	Conventional mult.	**OBC**
LUT	2402	1945
FF	1811	976
DSP	14	0

Table 5.4: Table recording the resource utilization comparison of proposed perceptron hardware with existing hardware

Design	**Logic gates**	**Frequency** (in MHz)	**I/P bits**	**Multiplier design**
[34]	6666	115	8	Conventional
Proposed	2921	200	16	OBC and CORDIC

in Table 5.2 for different batch sizes (b) of input. It is observed that the proposed hardware is able to achieve at least 21× speed-up as compared to software.

5.6.1.2 Experimental results

The perceptron hardware realized by the conventional multiplier module is compared with the proposed multiplier-less perceptron hardware realized using OBC architecture as shown in Table 5.3. It is found that the resource consumption of perceptron hardware designed using OBC unit is almost 50% less than the perceptron hardware realized using a conventional multiplier. The resource consumption and frequency of the proposed hardware design are compared with only available conventional hardware implementation in Table 5.4. It is found that the proposed hardware reduced the resource consumption to almost half of that of conventional implementation while the input bits are doubled. The frequency is also found to increase. This was achieved due to the use of OBC instead of conventional multipliers. Further, the MLP network hardware using the proposed perceptron hardware is implemented. This hardware consumed 5051 logic gates and the power consumption is 2.338 mW as implemented on FPGA. Thus, this hardware is suitable for low-power high-speed applications.

Table 5.5: Table recording the performance comparison of CORDIC with other existing techniques.

Design	**Tech.** (in nm)	**Area** (in mm^2)	**Freq.** (in MHz)	**Throughput** (in MCPS)	**Avg. err.**	**Bits**	**Algo.**
[45]	90	4.8	300	75	0.02	8	NL approx.
[46]	120	0.405	50	0.00005	0.0101	6	Linear approx.
Proposed	65	0.0649	200	0.00018	0.0059	16	CORDIC

The variation of sigmoid output in the portion of maximum variation $(-1, 1)$ is shown in Fig. 5.13 for clarity. The variation in sigmoid value calculated using CORDIC and LUTs from the actual value computed using Python is shown. The resolution is set as 0.1 for sigmoid calculation using LUTs. For example, $\sigma(x) = 0.729088$ for $x \in (0.90, 0.99]$. It can be observed that the sigmoid function implemented using CORDIC has lesser deviation as compared to the sigmoid function implemented using LUTs. Though, the latency of the CORDIC unit is slightly more ($\approx 0.3\mu$s) as compared to LUT-based realization ($\approx$ 30ns). So, CORDIC is suitable for applications where comparable accuracy is required such as bio-medical signal processing. The hardware performance of sigmoid implemented using CORDIC is compared with existing works implementing sigmoid using piece-wise linear approximation and second-order non-linear approximation in Table 5.5. It is observed that CORDIC unit has the lowest average error as compared to the other two techniques. Though the CORDIC has lower throughput it also occupies the lowest area for an increased number of bits.

5.6.2 PDT training hardware

5.6.2.1 Experimental set-up

The hardware implementation of this design is done on Virtex Ultrascale+ board XCVU13P-FLGA2577-3-E using Verilog HDL in Vivado 2017 and no high-level synthesis tool is used. For software comparison, the PDT is implemented in Python on an Intel i7 processor running at 3.2 GHz and having 6 cores. The values used for different variables have been listed in Table 5.6. These values have been estimated through trial and error based on software simulations in Python. The performance of the algorithm is tested for a varying number of features for all 9 datasets from different application areas like biomedical (Parkinson's detection, cryotherapy, heart attack detection, mammography) and other miscellaneous applications (3 types of activity recognition and occupancy). The training data matrix comprises 80% and the test data matrix comprised of remaining 20% of total instances n. The feature selection is only used for heart attack detection and Parkinson's detection which showed maximum accuracy when trained with 7 best-selected features (m_s) using an ensemble consisting of 100 ($N_{DT} = 100$) DTs. Further increase in DT number did not exhibit any improvement in accuracy as shown in Fig. 5.14 for Parkinson's dataset. Similarly, no accuracy improvement is observed with an increase in the number of features. The accuracy is tested for varying maximum depth (ζ_{max}) and batch size (b). The best value is obtained for a maximum depth $\zeta_{max} = 3$. A similar trend is observed for all other datasets. For small-sized datasets like cryotherapy, Parkinson's detection, heart attack

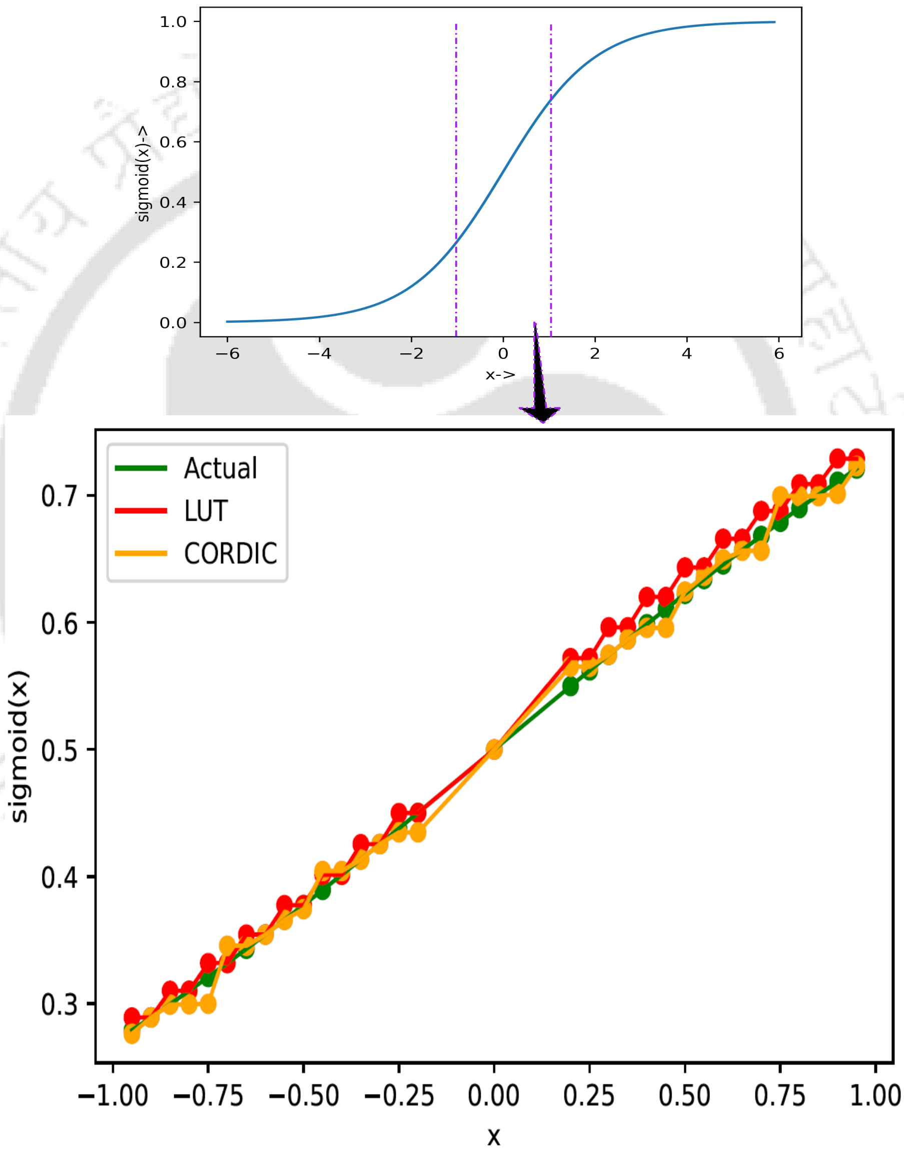

Figure 5.13: Plot showing deviation of the $\sigma(x)$ calculated using LUTs and CORDIC.

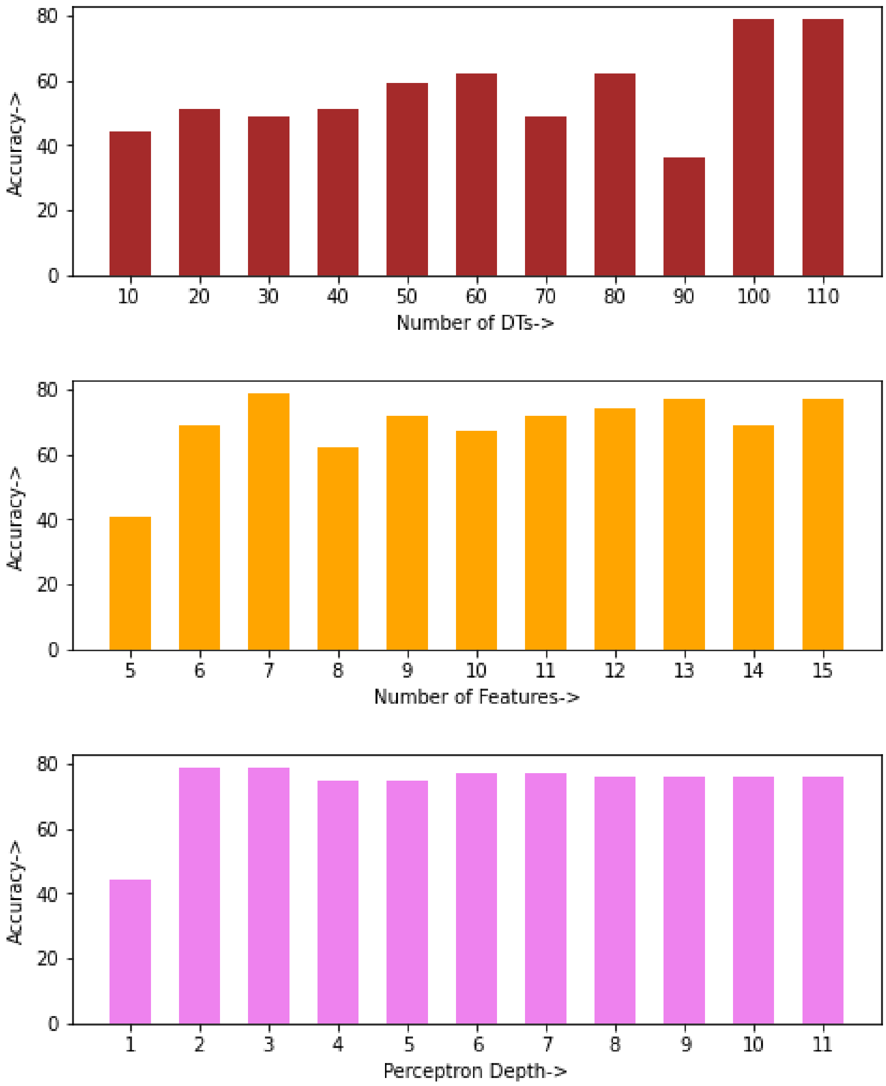

Figure 5.14: Bar-plot showing accuracy variation with number of DT used in feature selection, number of selected features, and depth of PDT

Table 5.6: The variables and their values as used in the hardware

Variable	Purpose	Value
N_{DT}	No. of DTs used for feature selection	100
m_s	No. of selected features	7
b	No. of data vectors in each batch	32
ζ_{max}	Maximum depth of PDT	3
ρ_c	Min. no. of data-vectors required to declare a node as split node	5
η_p	Node purity threshold	0.95
γ	Weight update factor	0.1

Table 5.7: Table recording the performance of PDT for variable-sized datasets on Python

Dataset	Acc.	Spec.	Sens.	Avg. F1 score	Latency (in s)	No. of instances
Cryotherapy	0.68	0.72	0.63	0.68	0.16	89
Parkinson'sDet.	0.74	0.71	0.75	0.66	0.13	195
HeartAttackDet.	0.81	0.83	0.79	0.81	0.35	270
Mammography	0.81	0.79	0.84	0.81	1.25	961
Occupancy	0.97	0.96	1.00	0.97	8.05	12690
Lie/walk	0.72	0.60	0.84	0.71	11.56	17278
Lie/sit	0.78	0.74	0.82	0.78	7.08	17278
Stand/walk	0.92	0.90	0.94	0.92	11.30	17278

detection, and mammography, the maximum accuracy is obtained for batch-size $b = 32$. While for medium (occupancy, lying/standing, lying/walking, sitting/lying), the optimum value for b is found to be 128.

5.6.2.2 Experimental results

To evaluate the performance of PDT, the accuracy, specificity, sensitivity, average F1-score, latency, and the number of instances (n) for those datasets implemented on the Python platform is recorded in Table 5.7. The training hardware for PDT is implemented on Virtex Ultrascale + board XCVU13P-FLGA2577-3-E using Verilog HDL in Vivado 2017 and no high-level synthesis tool is used. All reported hardware results are post-place and route. The critical path for the proposed architecture is found

Table 5.8: Table recording the accuracy and training latency of PDT on Python and FPGA.

Training data	Accuracy (%)		Latency(in s)		Batches (H)
	Python	FPGA	Python	FPGA	
Heart Attack	0.81	0.76	0.64	0.0005	19
Parkinsons	0.74	0.81	0.13	0.0006	25
Mammography	0.81	0.72	19.74	0.0007	30
Epilepsy	0.63	0.68	9.38	0.0008	31

Table 5.9: Table recording the resource utilization comparison of proposed PDT training hardware with training hardware proposed in previous chapters as implemented on Virtex Ultrascale+

Design	LUT (in K)	FF (in K)	BRAM	DSP	Data (in K)
TMDT Serial [19]	297.91	3.42	48	200	38
TMDT Mixed [41]	298.96	3.57	187	60	38
TMDT Batchmode [42]	154.65	5.44	24.5	0	512 (batch size)
HDT	5	3.03	30.5	0	50
Proposed Design	6.95	4.96	-	32	32 (batch size)

to be 6 ns. Hence, FPGA can operate at a maximum frequency of 165 MHz (speed grade -3). This hardware is trained on four applications, viz., epilepsy detection, heart attack detection, Parkinson's tremor detection [47] and to distinguish between a skin or non-skin pixel in image processing. The data is loaded into the training hardware in batch size of $b = 32$. The performance of PDT training on Python is compared to the FPGA implementation in Table 5.8. It is observed that the training implemented on FPGA is highly accelerated as compared to the Python implementation. Although, the accuracy of FPGA implementation is less as compared to Python implementation. The main reason behind this degradation is the error propagation caused by the CORDIC approximation in the FPGA platform. The resource consumption of PDT training hardware with the previous training accelerators is shown in Table 5.9. This training hardware has no data-point limitation as the training is performed in batches. This training accelerator is found to be more resource efficient than the existing training accelerators. The dynamic power consumption of this design is found to be 0.195 W.

5.6.3 PDT classification hardware

5.6.3.1 Experimental set-up

The classification hardware for PDT is implemented on Virtex Ultrascale+ board XCVU13P-FLGA2577-3-E using Verilog HDL in Vivado 2017 and no high-level synthesis tool is used. All reported hardware results are post-place and route. The critical path for the proposed architecture is found to be 5 ns. Hence, FPGA can operate at a maximum frequency of 200 MHz (speed grade -3). This hardware can perform binary classification for 4 tasks, viz., Parkinson's tremor detection [47] and distinguish between whether a person is walking or standing, lying or sitting, and lying or walking. The architecture is implemented to support 4 different data sizes for different applications. If the hardware is restricted to use for only one application, the cost of implementation is very high as

Table 5.10: Table recording the performance of PDT classification hardware for all four application modes on Python and FPGA as implemented on Virtex US+

Dataset	Spec.		Sens.		Avg. F1 score		Latency (in ms)	
	Python	**FPGA**	**Python**	**FPGA**	**Python**	**FPGA**	**Python**	**FPGA**
Lie/Sit	0.74	0.74	0.82	0.82	0.78	0.78	82.81	0.40
Lie/walk	0.60	0.75	0.84	0.53	0.71	0.63	85.03	0.40
Walk/Stand	0.90	0.90	0.94	0.93	0.92	0.92	68	0.31
ParkinsonsDet.	0.71	1	0.75	0.53	0.66	0.55	3.00	0.006

hardware resources are costly. So, apart from using it only for Parkinson detection, the design can be used for activity recognition also which increases the design re-usability and thus reduces the cost. This hardware takes the mode of operation and data vector as input and gives the predicted class at the output (as either 0 or 1) as described in the previous section.

5.6.3.2 Experimental results

The performance of the classifier, i.e., specificity, sensitivity, avg. F1 score and latency, as implemented on FPGA compared to the software implementation on Python for all four applications are reported in Table 5.10. It can be seen that classification implemented on FPGA is much faster than Python. For small datasets like Parkinson's tremor detection, the performance on FPGA is less than Python implementation. For the other three classification tasks, other performance parameters are almost the same with a large improvement in detection latency. The FPGA implementation is observed to be much faster than the Python implementation. The proposed hardware consumes 0.06 W of dynamic power. The power consumption distribution by various hardware components is shown in Fig. 5.15. It can be observed that only 31% of power is consumed by logic and a huge percentage of around 30% is consumed by seven BRAMs used to store the test input features. In a real-life application, these BRAMs will not be required as input can be fed directly through the input port which will require much lesser power. Thus, the power consumption will be further reduced in the case of real-life applications.

The resource and power consumption and throughput of proposed hardware are compared with existing classification hardware in Table 5.11. Buschjager *et al.* has compared the classification performance using 3 types of DT implementations, viz., if-else, naive implementation, and DNF implementation. Tong *et al.* also proposed two approaches for DT classification: ODT and DQ. The proposed hardware consumes 5× lesser LUT and also has lower FF and power consumption as com-

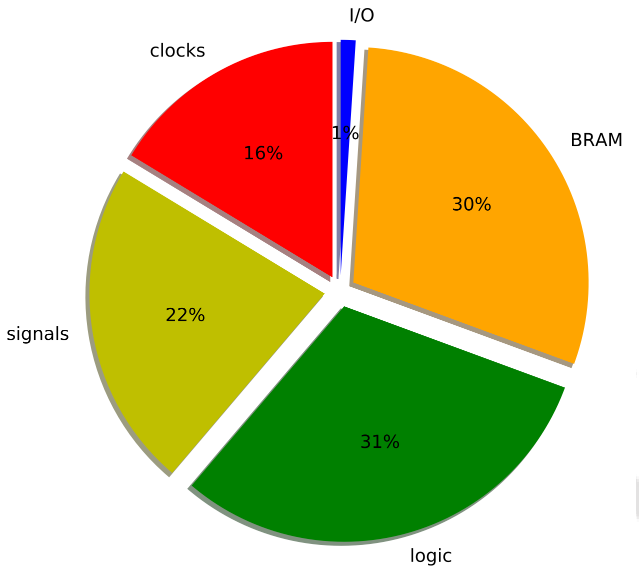

Figure 5.15: Pie-plot showing the dynamic on-chip power distribution proposed PDT classification hardware.

Table 5.11: Table recording the resource utilization comparison of proposed PDT classification hardware with existing classification hardware on FPGA.

Design	LUT (in K)	FF (in K)	Throughput (in MCPS)	Power (in W)	Board used
[28]-if-else	10.24	1.31	1.40	0.07	Artix-7
[28]-naive	0.03	0.06	1.17	0.01	Artix-7
[28]-DNF	9.61	3.18	1.10	0.02	Artix-7
[29]-ODT	1750	1000	800	-	Virtex US
[29]-DQ	1500	4000	400	-	Virtex US
Our Design	1.13	0.51	14.3	0.019	Virtex US
Our Design	1.31	0.57	1.27	0.032	Artix 7

pared to if-else and DNF implementations in [28]. The throughput is higher for proposed hardware than the naive implementation in [28] with an increase in resource consumption. As compared to DQ and ODT architectures proposed in [29] for high throughput applications, the proposed design has at least 1000× reduction in resource and area usage with lower throughput. Thus, the area is reduced at the cost of speed for the bio-medical applications. Thus, the proposed architecture achieved comparable performance with only 3 split nodes due to the use of oblique nodes. The replacement of multiplication by shift and add in the inner product calculation unit further reduced the resource and therefore reduced the power consumption while maintaining the speed.

The proposed design is synthesized in Synopsys DC compiler using UMC 65 nm technology @167MHz for ASIC implementation. The resource utilization details of the proposed design are compared with existing ASIC-based implementations of bio-medical systems in Table 5.12. As there

Table 5.12: Table recording the performance comparison of proposed PDT classification hardware with existing classification hardware on ASIC.

Design	**Tech.** (in nm)	**Classifier**	**Energy eff. /class** (in nJ)	**Area** (in mm^2)	**Logic-size**	**Application**
[27]	65	XGB	41.2	1	330k	Epilepsy
[48]	90	GTCA+SVM	970	0.13	-	Epilepsy
[49]	28	SOUL	1.5	0.10	-	Epilepsy
Proposed	65	PDT	0.008	0.12	132k	Multi

are no existing ASIC implementations for Parkinson's detection or activity recognition, the proposed design utilization is compared with existing seizure detection designs. The proposed design is found to be more energy-efficient as compared to existing seizure detection hardware. Further, this hardware is designed for use in four different applications. While the hardware in [27, 48, 49] is specific to seizure detection only.

5.7 Discussions

In this chapter, a perceptron hardware realized using OBC unit (for inner product computation) and CORDIC unit (for sigmoid function evaluation) was proposed. It is found to be both resource-efficient (consumes low power) and high-speed as compared to software-based implementation as well as the existing perceptron hardware [34]. This design was found to be at least 21× faster than its corresponding software implementation. It was also found to minimize resource consumption to almost half of the existing hardware. This hardware was used to construct an MLP network and found to be resource-efficient. This perceptron block can be used to construct low-power and high-speed training hardware for MLPs and NNs. Next, training hardware was proposed for PDT algorithm. The nodes in PDT implement oblique splits. This implementation of oblique nodes enables the DT to achieve comparable performance with a lesser number of nodes. The training hardware was observed to have reduced latency and accuracy when implemented on FPGA as compared to Python. The design has reduced resource usage as compared to the training hardware proposed in previous chapters.

The classification hardware was proposed next that performs classification using the PDT algorithm. The trained DT was loaded on FPGA and was used in classification. The hardware is capable of performing classification for four different applications, v.i.z. Parkinson's disease detection, lie/sit, lie/walk and walk/stand recognition according to the selected mode of operation. The inner product calculation was designed to be resource efficient. This reduction in resource consumption leads to reduced power consumption as compared to existing classification architectures [12, 28, 29].

6 Conclusion

Contents

6.1 Summary

Decision Tree (DT) training is computation intensive and requires a substantial amount of resources for hardware-based training implementation. This thesis proposes low-complexity DT training algorithms. The training of these DTs were implemented on FPGA platform. These algorithms required lesser hardware resources while providing almost similar performance like conventional DTs. The classification hardware implemented on both ASIC and FPGA was also proposed. A summary of the contributions of this thesis is presented next.

A 32-bit serial architecture running at 62 MHz was proposed in Chapter 2 for implementing the training of the Two Means Decision Tree (TMDT) algorithm on FPGA. This provided reconfigurability thereby allowing dynamic training. Here, 2 different architectures were proposed for offline TMDT implementation. As the proposed hardware is capable of testing 5 binary data sets each having different dimensions and sizes, it can be used to train a wide variety of datasets. The proposed TMDT algorithm runs at least 28× faster and has lower complexity than the widely used conventional C4.5 algorithm. The serial architecture has been shown to achieve at least 10× speed-up as compared to software implementations for the same datasets by introducing some extent of parallelism. In this chapter, a mixed pipeline and parallel training accelerator for the TMDT algorithm is also proposed. The training accelerator is found to speed up the training process by at least 14×. The training in FPGA was completed in 16.54×10^{-3} sec whereas software training required a minimum of 224×10^{-3} sec. These two training hardwares were tested using 32-bit training data.

A multi-class training accelerator for TMDT learning in batch mode was proposed in Chapter 3. This proposal is a depth pipelined architecture, enabling the execution of two depths at a time, and is implemented in batch mode. This design was reported to execute at least 27× faster than a C-based software implementation and 26× faster than existing hardware [20]. Also, it is much more resource efficient as compared to existing classification as well as training architectures. In contrast to the training accelerators proposed in Chapter 2, training with large datasets resulted in higher accuracy. Also, the implementation of depth-pipelining resulted in efficient clock cycle usage.

The hardware realization of a Hybrid Decision Tree (HDT) algorithm was proposed in Chapter 4. This proposal was 8× faster than the C4.5 DT algorithms. Moreover, the proposed architecture allows the FPGA to operate at 125 MHz for the training of HDT. The proposed hardware implementation shows a speed-up of at least 20× as compared to the software implementation. It has 30× less memory and 180× computation resource consumption as compared to optimized classification hardware. Also, it achieved a reduction in hardware usage as compared to existing training accelerators. Moreover, it was found to be at least 60× faster than the existing FPGA-based training accelerator for CART algorithm [20]. This training hardware was found to be more resource efficient as compared to the existing classification hardware proposed in [33] and training hardwares proposed in Chapters 2 and 3. The HDT was found to have higher accuracy as compared to the proposals presented in Chapters 2 and 3.

Chapter 5 describes the hardware realization of a single output Perceptron, a Multi-layered Perceptron (MLP, with single hidden layer), and the training of a Perceptron Decision Tree (PDT). The single output perceptron hardware is realized using an OBC and a CORDIC unit. This proposal was found to be resource-efficient (consumes low power) and high-speed compared to software implementation and existing perceptron hardware [34]. The proposed hardware reduced the resource consumption by 50% as compared to the proposal presented in [34]. This proposal is found to be at least 21× faster than its corresponding software implementation. This perceptron hardware was used further to construct a resource-efficient MLP network. This perceptron block can be used to construct low-power and high-speed training hardware for MLPs and other artificial neural network architectures. This perceptron block was also used to construct a PDT. Accordingly, training and classification hardware for PDT were proposed. The PDTs implemented on both FPGA and ASIC were applied to applications involving Parkinson's disease detection and activity recognition. The implementation of oblique split decision functions enabled the PDT to achieve good performance with a lesser number of nodes. This design showed good performance with reduced resource consumption. This resulted in lower resource consumption thereby leading to reduced power consumption as compared to existing classification architecture proposed in [33].

Thus, this thesis proposed three low-complexity DT algorithms. The TMDT algorithm was observed to be low complexity and low accuracy algorithm. Then, the performance of TMDT was improved by implementing in batch-mode and training on large sized dataset. HDT was observed to have better accuracy than TMDT. PDT showed further improvement in accuracy, especially for small-sized datasets used in bio-medical applications. These algorithms were then trained on FPGA and ASIC. Due to a reduction in algorithmic complexity, these algorithms were observed to run faster than the existing training hardware for CART DT training. Also, the training latency on hardware was improved to a great extent as compared to C and Python platforms. The architectural optimizations further resulted in improved latency and resource utilization.

6.2 Future Research Directions

This thesis proposed hardware realization of low-complexity DT algorithms and their implementations on FPGA. The present work achieved competitive performance with low hardware resources for three different decision tree algorithms (TMDT, HDT and PDT). Based on the observations of the present work, the following research directions can be explored for future work

- The HDT architecture can be depth pipelined to allow mean-dependent and axis-aligned node execution, simultaneously. This would reduce the execution time while efficiently utilizing the resources. The depth-parallel hardware can be designed and implemented on FPGA. This would increase the speed.
- A hybrid tree can be realized with split nodes having all three categories of split decision functions, v.i.z. mean-dependent (from TMDT), axis-aligned, and oblique (perceptron based). These split nodes will be employed at different sets of depths. A depth-pipelined architecture may be implemented for realizing the training of such a hybrid tree.
- The decision forest (or random forest) is widely used by the ML community. It has exhibited superior performance in several applications. This work can be extended by training multiple low-complexity DTs on data subsets or feature sub-spaces. The DTs can be modified to provide class probabilities in place of discrete labels. A perceptron block can be trained to combine the inferences of multiple trees in the ensemble framework.

Bibliography

[1] E. Alpaydin, *Introduction to machine learning*. MIT press, 2020.

[2] V. Cherkassky and Y. Ma, "Practical selection of SVM parameters and noise estimation for SVM regression," *Neural networks*, vol. 17, no. 1, pp. 113–126, 2004.

[3] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[4] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware," in *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, 2001, pp. 103–110.

[5] A. Lukasová, "Hierarchical agglomerative clustering procedure," *Pattern Recognition*, vol. 11, no. 5-6, pp. 365–381, 1979.

[6] J. Shen, X. Hao, Z. Liang, Y. Liu, W. Wang, and L. Shao, "Real-time superpixel segmentation by DBSCAN clustering algorithm," *IEEE transactions on image processing*, vol. 25, no. 12, pp. 5933–5942, 2016.

[7] M. W. Ahmad, M. Mourshed, and Y. Rezgui, "Trees vs neurons: Comparison between random forest and ANN for high-resolution prediction of building energy consumption," *Energy and Buildings*, vol. 147, pp. 77–89, 2017.

[8] S. Ruggieri, "Efficient C4.5 classification algorithm," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 2, pp. 438–444, 2002.

[9] A. Godbole, S. Bhat, and P. Guha, "Progressively balanced multi-class neural trees," in *2018 Twenty Fourth National Conference on Communications (NCC)*, 2018, pp. 1–6.

[10] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji, "High-performance FPGA-based CNN accelerator with block-floating-point arithmetic," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[11] J. Wang, J. Lin, and Z. Wang, "Efficient hardware architectures for deep convolutional neural network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 6, pp. 1941–1953, June 2018.

[12] F. Saqib, A. Dutta, J. Plusquellic, P. Ortiz, and M. S. Pattichis, "Pipelined decision tree classification accelerator implementation in FPGA (DT-CAIF)," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 280–285, 2013.

[13] Y. Yang, I. G. Morillo, and T. M. Hospedales, "Deep neural decision trees," *arXiv preprint arXiv:1806.06988*, 2018.

[14] K. Ramya, Y. Teekaraman, and K. R. Kumar, "Fuzzy-based energy management system with decision tree algorithm for power security system," *International Journal of Computational Intelligence Systems*, vol. 12, no. 2, pp. 1173–1178, 2019.

[15] B. Zhu, M. Farivar, and M. Shoaran, "ResOT: Resource-efficient oblique trees for neural signal classification," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 14, no. 4, pp. 692–704, 2020.

[16] S. S. Gavankar and S. D. Sawarkar, "Eager decision tree," in *2017 2nd International Conference for Convergence in Technology (I2CT)*, 2017, pp. 837–840.

[17] S. Mitrofanov and E. Semenkin, "An approach to training decision trees with the relearning of nodes," in *2021 International Conference on Information Technologies (InfoTech)*, 2021, pp. 1–5.

[18] H. Elaidi, Y. Elhaddar, Z. Benabbou, and H. Abbar, "An idea of a clustering algorithm using support vector machines based on binary decision tree," in *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, 2018, pp. 1–5.

[19] R. Choudhury, S. R. Ahamed, and P. Guha, "Efficient hardware implementation of decision tree training accelerator," *SN Computer Science*, vol. 2, no. 5, pp. 1–10, 2021.

[20] G. Chrysos, P. Dagritzikos, I. Papaefstathiou, and A. Dollas, "HC-CART: A parallel system implementation of data mining classification and regression tree (CART) algorithm on a multi-FPGA system," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, pp. 1–25, 2013.

[21] J. Wang, J. Lin, and Z. Wang, "Efficient hardware architectures for deep convolutional neural network," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 6, pp. 1941–1953, 2017.

[22] X.-Y. Shih, Y. Chiu, and H.-E. Wu, "Design and implementation of decision-tree (DT) online training hardware using divider-free GI calculation and speeding-up double-root classifier," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2022.

[23] G. L. Foresti and T. Dolso, "An adaptive high-order neural tree for pattern recognition," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 34, no. 2, pp. 988–996, 2004.

[24] M. Khalil Alsmadi, K. B. Omar, S. A. Noah, and I. Almarashdah, "Performance comparison of multi-layer perceptron (back propagation, delta rule and perceptron) algorithms in neural networks," in *2009 IEEE International Advance Computing Conference*. IEEE, 2009, pp. 296–299.

[25] Xilinx.

[26] F. Schuiki, M. Schaffner, F. K. Gürkaynak, and L. Benini, "A scalable near-memory architecture for training deep neural networks on large in-memory datasets," *IEEE Transactions on Computers*, vol. 68, no. 4, pp. 484–497, 2018.

[27] M. Shoaran, B. A. Haghi, M. Taghavi, M. Farivar, and A. Emami-Neyestanak, "Energy-efficient classification for resource-constrained biomedical applications," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 8, no. 4, pp. 693–707, 2018.

[28] S. Buschjäger and K. Morik, "Decision tree and random forest implementations for fast filtering of sensor data," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 209–222, 2017.

[29] D. Tong, Y. R. Qu, and V. K. Prasanna, "Accelerating decision tree based traffic classification on FPGA and multicore platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 11, pp. 3046–3059, 2017.

[30] S. Lopez-Estrada and R. Cumplido, "Decision tree based FPGA-architecture for texture sea state classification," in *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*. IEEE, 2006, pp. 1–7.

[31] Y. Yang, C. S. Boling, and A. J. Mason, "Power-area efficient VLSI implementation of decision tree based spike classification for neural recording implants," in *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*. IEEE, 2014, pp. 380–383.

[32] J. Canilho, M. Véstias, and H. Neto, "Multi-core for k-means clustering on FPGA," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[33] F. Winterstein, S. Bayliss, and G. A. Constantinides, "FPGA-based K-means clustering using tree-based data structures," in *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE, 2013, pp. 1–6.

[34] W. Qinruo, Y. Bo, X. Yun, and L. Bingru, "The hardware structure design of perceptron with FPGA implementation," in *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme - System Security and Assurance (Cat. No.03CH37483)*, vol. 1, 2003, pp. 762–767 vol.1.

[35] M. S. Prakash and R. A. Shaik, "Low-area and high-throughput architecture for an adaptive filter using distributed arithmetic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 11, pp. 781–785, 2013.

[36] M. T. Khan and R. A. Shaik, "An energy efficient VLSI architecture of decision feedback equalizer for 5G communication system," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 7, no. 4, pp. 569–581, 2017.

[37] M. Qian, "Application of CORDIC algorithm to neural networks VLSI design," in *The Proceedings of the Multiconference on" Computational Engineering in Systems Applications"*, vol. 1. IEEE, 2006, pp. 504–508.

[38] H. Chen, L. Jiang, Y. Luo, Z. Lu, Y. Fu, L. Li, and Z. Yu, "A CORDIC-based architecture with adjustable precision and flexible scalability to implement sigmoid and tanh functions," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2020, pp. 1–5.

[39] S. Behnke and N. B. Karayiannis, "Competitive neural trees for pattern classification," *IEEE Transactions on Neural Networks*, vol. 9, no. 6, pp. 1352–1369, 1998.

[40] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[41] R. Choudhury, S. R. Ahamed, and P. Guha, "Training accelerator for two means decision tree," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 7, pp. 1465–1469, July 2021.

[42] R. Choudhury, S. R. Ahamed, and P. Guha, "FPGA implementation of batch-mode depth-pipelined two means decision tree," *IEEE Embedded Systems Letters*, pp. 1–1, 2022.

[43] A. P. Muniyandi, R. Rajeswari, and R. Rajaram, "Network anomaly detection by cascading K-means clustering and C4.5 decision tree algorithm," *Procedia Engineering*, vol. 30, pp. 174–182, 2012, international Conference on Communication Technology and System Design 2011.

[44] P. Geurts, D. Ernst., and L. Wehenkel, "Extremely randomized trees," *Machine Learning*, vol. 1, no. 63, pp. 3–42, 2006.

[45] C.-H. Tsai, Y.-T. Chih, W. H. Wong, and C.-Y. Lee, "A hardware-efficient sigmoid function with adjustable precision for a neural network system," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 11, pp. 1073–1077, 2015.

[46] M. Al-Nsour and H. Abdel-Aty-Zohdy, "Implementation of programmable digital sigmoid function circuit for neuro-computing," in *1998 Midwest Symposium on Circuits and Systems (Cat. No. 98CB36268)*, 1998, pp. 571–574.

[47] M. Little, P. McSharry, and S. Roberts, "Exploiting nonlinear recurrence and fractal scaling properties for voice disorder detection," *BioMed Eng OnLine*, vol. 6, no. 23, 2007.

[48] M. Zhang, L. Zhang, C.-W. Tsai, and J. Yoo, "A patient-specific closed-loop epilepsy management SoC with one-shot learning and online tuning," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 4, pp. 1049–1060, 2022.

[49] A. Chua, M. I. Jordan, and R. Muller, "Soul: An energy-efficient unsupervised online learning seizure detection classifier," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 8, pp. 2532–2544, 2022.

List of Publications

Journal Publications

1. R. Choudhury, S. R. Ahamed and P. Guha, "Training Accelerator for Two Means Decision Tree", in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 7, pp. 1465-1469, July 2021.
2. R. Choudhury, S. R. Ahamed, and P. Guha, "Efficient Hardware Implementation of Decision Tree Training Accelerator", in Springer Nature Computer Science 2, 360, 2021.
3. R. Choudhury, S. R. Ahamed, and P. Guha, "FPGA Implementation of Batch-mode Depth-pipelined Two Means Decision Tree", in IEEE Embedded Systems Letters, 2022.

Conference Publications

1. R. Choudhury, S. R. Ahamed, and P. Guha, "Hardware Implementation of Low Complexity High-speed Perceptron Block", in IEEE International Symposium on Circuits and Systems (IEEE-ISCAS), 2022.
2. R. Choudhury, S. R. Ahamed, and P. Guha, "FPGA Implementation of Low Complexity Hybrid Decision Tree Training Accelerator", in IEEE Mid-West Symposium on Circuits and Systems (IEEE-MWSCAS), 2021.
3. R. Choudhury, S. R. Ahamed, and P. Guha, "Efficient Hardware Implementation of Decision Tree Training Accelerator", in IEEE International Symposium on Smart Electronic Systems (IEEE-iSES), 2020.

Manuscripts submitted

1. R. Choudhury, S. R. Ahamed, and P. Guha, "Pipelined training accelerator for portable devices", in IEEE Transactions on Circuits and Systems-II.

Manuscripts under preparation

1. R. Choudhury, S. R. Ahamed, and P. Guha, "Resource-efficient training hardware for Perceptron Decision Tree".