POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Logic-in-Memory implementation of Random Forest Algorithm

Relatori:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Ph.D. Giovanna Turvani

Candidata:
Debora Uccellatore

Anno Accademico 2017-2018

Table of contents

List of tables

List of figures

Introduction

Data processing in modern systems, based on a Von Neumann architecture, is usually performed by passing information between the processing unit (CPU) and the storage unit (MEMORY) that are connected via buses. One of the main problems of this system is the speed of access to memory that becomes the bottleneck of the structure. In this thesis it is implemented a machine learning algorithm in which the amount of data to be analyzed is very high, therefore, in order to overcome the limits inherent to memory and consequently to increase the efficiency of data processing, the search has been oriented on the implementation of a Logic-in-Memory architecture. This structure has been created to reduce the exchange of information between the memory and the processing unit because memory and logic are mixed in the same structure. The LIM implementation can be considered as a memory that not only stores the data but, at the same time, processes them within its structure, exceeding the limits of access to memory.

Chapter 1

State of Art

1.1 Machine Learning Description

Machine Learning [1] [2] is one of the most interesting and most used recent technologies. In fact, whenever you search the web, you use social apps to digitize photos and when you read e-mails, machine learning algorithms are used.

It is a branch of computer science that deals with creating systems and algorithms that can learn, based on input data. Moreover, it can also be seen as mathematics applied to data processing by means of data mining as shown in the figure 1.1.

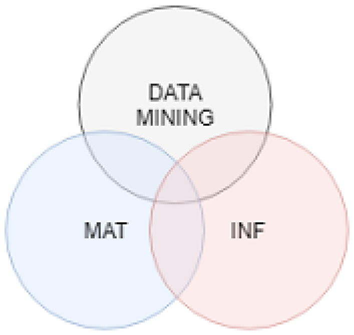


Figure 1.1: Machine learning creation

Machine Learning is one of the fundamental areas of artificial intelligence.

One of the main objectives of research in this area is to learn how to recognize complex models automatically and make smart decisions based on incoming data.

" A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance as tasks in T, as measured by P improves with experience E" [1]

[1]Mitchell(1997) [22]

Machine Learning can be categorized [3] into three types:

- Supervised Learning: it learns the relation between an input and an output target.
- Unsupervised Learning: it learns from the input without using an output target.
- Reinforced Learning: it is a process that simulates human judgment.

These systems use the knowledge learned from a previous event as a basis, and then process input with similar characteristics and classify them. In Machine Learning, the term CLASSIFIER means the operation that determines the belonging of an observation to a class. It is interesting to be able to build a system that ”learns” to correctly classify a series of samples, in different application contexts, deducing the parameters that characterize them.

- In the medical field, a system that, obtained a sufficient number of clinical data, is able to find out whether a patient is suffering from a disease, or what the severity of this.
- In the field of computer security, a system that, given a set of emails, is able to classify new emails and decide whether they are spam or not.
- In the graphic field a system that is able to recognize the objects represented by data points belonging to an image.

Classification methods are numerous. In fact, it is possible to combine various types of classifiers with each other.

The classifiers can be grouped as follows:

- Statistic: memorizes the parameters of the various probability distributions. To classify a generic object you can estimate the probabilities of belonging to the various classes.
- Mathematics: through the examples training creates a mathematical classification function that models the data. This function is used on new data to identify the correct label.

- Logical: the classification function is expressed by logical conditions extracted from the data.
- Based on the examples: they memorize all the examples of the training set and assign an object to the relative class by evaluating the similarity with the memorized examples.

In supervised learning the idea is that we are going to teach the computer how to do something, whereas in unsupervised learning we are going to let it learns by itself. In the following section will be described in general way these different kinds of learning will be described.

1.1.1 Supervised Learning

Supervised learning technique trains the model starting from some labeled data set. This then it will produce predicted output using new data samples as input as shown in figure 1.2. This type of ML algorithm generalizes the response to all possible input based on the training data providing an output solution.

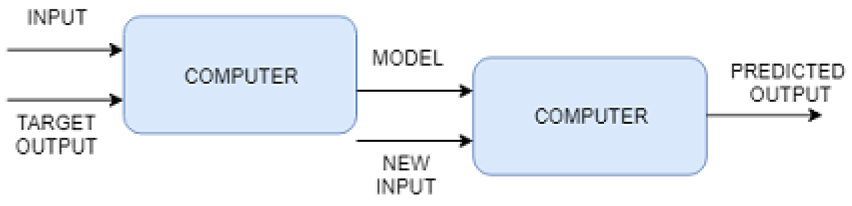


Figure 1.2: Supervised Learning Representation

Supervised Learning starts from a set N that contains a large number of vectors that describe the object under examination. This set is called training set and it is used to set the adaptive model parameters, which is based on the use of a classifier. The categories of membership of the objects in the training set are known in advance, typically through a labeling process, which associates a label to each vector. We can express the object category using a vector t target, which represents its identity. The result of a Machine Learning algorithm can be expressed as a function g that takes an object x as input and generates a vector y in the output, coded in the same way as t. The form of the y function is determined during the training phase, also known as the learning phase, based on the training data. The purpose of the training phase is to approximate as much as possible an ideal function f, which performs the same function as g, but with an accuracy equal to 100% as shown in figure 1.3.

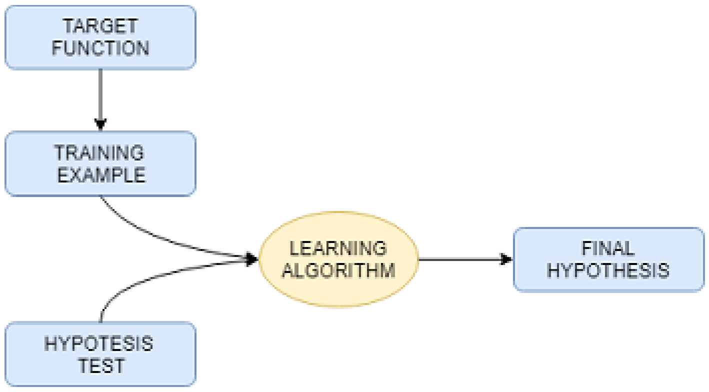


Figure 1.3: General outline of Supervised Learning

In addition, the training is performed by minimizing a cost function, the loss function, which represents the error of the output provided by the network with respect to the desired output. The choice of the loss function to be minimized is important. Furthermore, it is good practice not to use the whole set of training to train the network, because the network tends to bind too specific to the examples of the training set and when the input varies, there is the risk of having an incorrect prediction, an overfitting. A possible solution is to divide the training set into two parts one for training and one for a check.

1.1.2 Unsupervised Learning

Unsupervised learning is a paradigm used for machine learning. This algorithm provides the learning model with a series of "unlabeled" inputs and allows the model itself to reclassify and organize inputs based on common characteristics, making forecasts on subsequent inputs as shown in figure 1.4.

In fact, this technique proceeds by comparing the data and looking for similarities and differences.

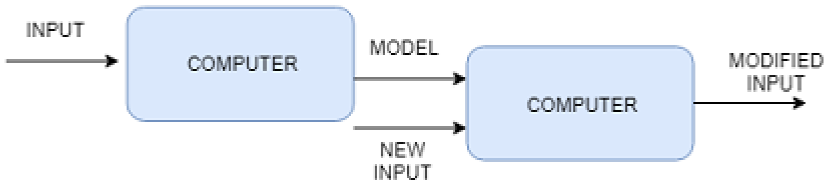


Figure 1.4: Unsupervised Learning Representation

1.1.3 Reinforced Learning

Reinforcement learning is a type of Machine Learning algorithms which allows software agents and machines to determine automatically the ideal behavior within a specific context, to maximize its performance as shown in figure 1.5.

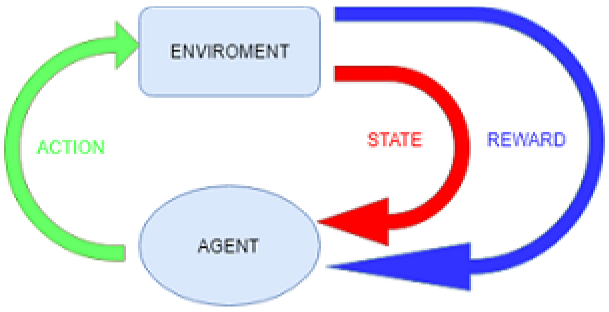


Figure 1.5: Reinforced Learning Representation

This type of algorithm learns to react to an environment.

1.2 Machine Learning Algorithms

In this section, the starting is point analyzing the Machine learning algorithms. We analyze four different types of algorithms [4], in particular:

- The supervised learning algorithm K-NN.
- The unsupervised learning algorithms K-means and K-modes.
- The ensemble learning algorithm Random Forest.

The last one is the chosen algorithm to be implemented within the project using a Logic-in-Memory architecture.

1.2.1 K-NN (K-Nearest Neighbor)

The K-Nearest Neighbor algorithm is part of the instance-based classifiers. This technique bases the forecasting procedure directly on the already classified data, comparing the characteristics of the observations. Given a metric in multidimensional space, the k-nearest-neighbor classifier classifies the k-patterns x_k with the same class as the element x'_k closest to it in the training set (TS). KNN is the simplest supervised learning algorithm. Any object is classified according to the majority of the votes of the k its neighbors where k is a positive integer, the choice of which depends on the characteristics of the data. The algorithm has two main phases:

1. Training phase: in which the space is partitioned according to the position and characteristics of the objects.
2. Classification phase: in which an object is associated to a class if the latter is the most frequent among the k-examples closest to the object.

Furthermore, this algorithm has two main steps for each test instance:

1. Computes distances between the testing instance and reference one.
2. It finds the K-nearest reference instance to the testing instance and assigns the testing instance the most frequent label.

In this type of algorithm the most time-consuming operations are computing distances between instances. Considering the following figure 1.6, it is clear that by considering $k = 5$, then 5-NN, the object x taken into consideration will be part of the Black class since the latter is in greater number than the Red class.

1.2.2 K-Means

K-means is an unsupervised learning algorithm. It is a clustering algorithm that allows you to subdivide a set of objects into k groups based on their attributes. Each cluster is identified by a centroid or a midpoint. This algorithm follows an iterative procedure by performing the following steps:

1. Select initial K-means for K-cluster.

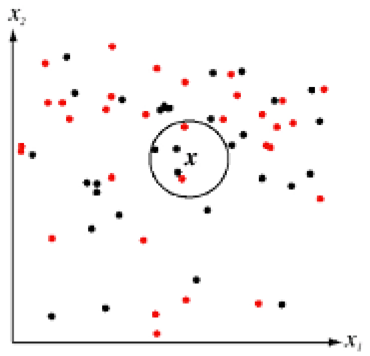


Figure 1.6: K-Nearest Neighbor algorithm example

2. Calculate the dissimilarity between an object to the cluster.

3. Allocate an object to the cluster whose mean is nearest to the object.

4. Re-calculate the mean of a cluster from the objects allocated to it so that the intra cluster dissimilarity is minimize.

This algorithm is efficient in processing large data sets. Furthermore, computational complexity is much faster than hierarchical clustering. However, in this algorithm it is difficult to specify the number of clusters. It works only on numeric values.

1.2.3 K-Modes

To remove the limitation due to only working on numerical value used into K-means, it is possible to extend the use on categorical domains. This algorithm follows an iterative procedure by performing the following steps:

1. Select k initial modes, one of each cluster

2. Allocate an object to the cluster whose mode is the nearest to it according to dissimilarity. Update the mode of the cluster after each allocation.

3. After all objects have been allocated to clusters retest the dissimilarity of objects against the current modes.

4. Repeat the step 3 until no object has changed clusters after a full cycle test of the whole set of data.

1.2.4 Random Forest

Random Forest is an Ensemble learning method [5] [6] [7]. This type of training uses several classifiers combined together to maximize performance. At the base there are weak classifiers, which added in a particular way, define a strong classifier that performs the classification. The aim is, therefore, to train M classifiers with different training sets and to combine the results as shown in figure 1.7.

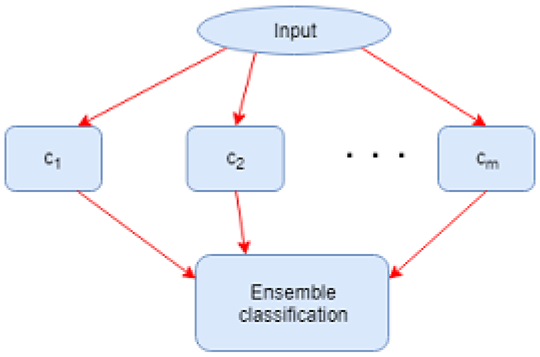


Figure 1.7: Ensemble learning method scheme

The Random Forest classifier is made up of many decision trees, each of which gives out the corresponding class. Finally, each class is put to a vote to get the final result as shown in figure 1.8.

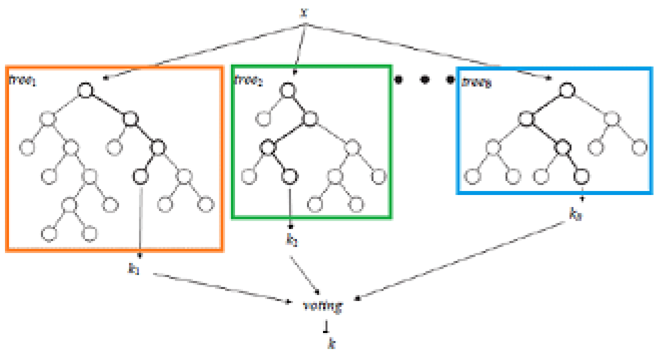


Figure 1.8: [8] Random Forest scheme

The Random Forest method is a variant of bagging that uses decision trees as basic classifiers and random vectors to generate trees. This randomness is a factor used to increase diversity in the construction of classifiers. The idea of Bagging is to train each tree within a forest on a different subset of the training set, randomly sampling the same labeled database, repeatedly sampling the input training set, with uniform probability distribution. This technique avoids the specialization of the selected parameters to a single training set, increasing the generalization, moreover the training phase is faster, than using the entire set labeled. This procedure follows the following steps:

1. Run k bootstrap sample from the dataset.
2. From each, create a classifier.
3. To classify an object never before seen:
    - Pick up the predictions of each classifier.
    - Take the most popular one for good.

Each tree processes data obtained by random sub-sampling (RSS) the input (X) using a pattern vector. The construction of Random Forest is described as in the following steps:

1. Draw m-tree bootstrap samples from the original data.
2. Grow a tree for each bootstrap data set.
3. Aggregate information from the m-tree for new data prediction.
4. Compute an out of bag (OOB) error rate by using the data not in the bootstrap sample.

So considering a tree that performs object recognition, this could have a long list of features (features). Each node of the tree is assigned a random subset of these characteristics to best determine the division of data. Each subsequent node, consequently, obtains a new subset of characteristics, still chosen randomly, on which to divide the data again.

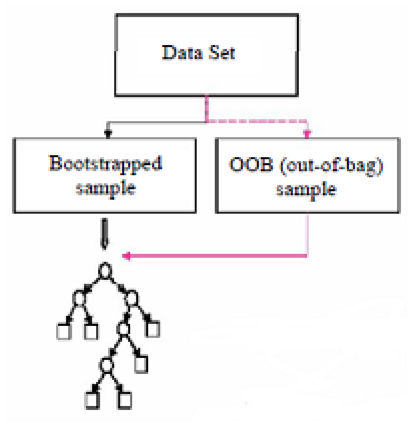


Figure 1.9: [8] Flow of Random Forest algorithm

1.3 Hardware Machine Learning Accelerator

Machine Learning algorithms require the processing of large volumes of data. Thus, the energy efficiency of the hardware is limited by the energy cost of memory accesses. Being able to reuse data leads to a reduction in energy consumption. This process is carried out through the use of Hardware accelerators that are used to support Machine Learning techniques.

1.3.1 DianNao: A Small-Footrint High-Throughput Accelerator

DianNao [9] is an Hardware accelerator that is used for image classification by implementing a series of layers. These layers are executed in sequence so they can be considered independently. Each layer contains several sub-layer called feature maps. We can distinguished three kinds of layers, as shown in figure 1.10:

1. Convolutional layer
2. Pooling layers
3. Classifier layers

In the convolutional layer, a pixel is associated with an element in the array. Subsequently the polling phase it is used to aggregate the information of the following

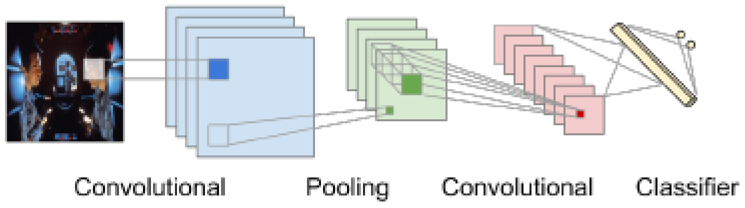


Figure 1.10: [9] Neural Network with convolutional, pooling and classifier layers

layers and reduce the size of the feature maps. Then there is the classification phase where the image is classified according to its characteristics. To analyze this type of accelerator we implement a neural network. This consists of an entry layer, a series of hidden layers and an exit layer as shown in figure 1.11.

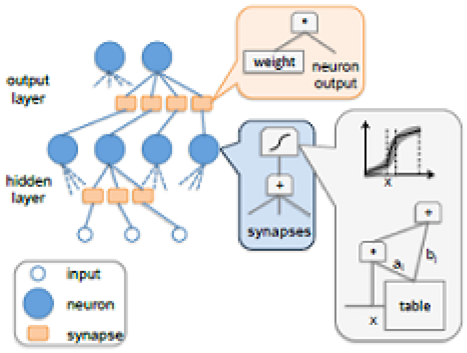


Figure 1.11: [9] Full hardware implementation of Neural Network

To implement a silicon neural network, the layout must be filled with neurons and synapses that must be implemented in hardware.

The neurons are implemented through a logic circuit and the synapses are implemented as latches or RAM memory. The delay, the area and the energy increase with the number of neurons as shown in figure 1.12, so it was decided to implement only some neurons and synapses in hardware.

The principle was to share the various physical neurons and use a RAM on chip to store the synapses and values of the intermediate neurons of the hidden layers. If the neural networks are small all the intermediate values could be stored in RAM but if the network is wide, an accelerator is inserted which becomes an interaction

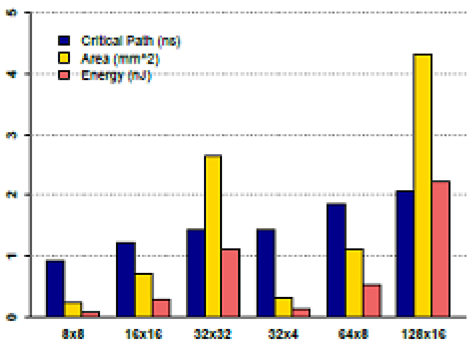


Figure 1.12: [9] Energy, critical path and area of full hardware layers

between the convolutional layer and the hierarchical memory. The main components of the accelerator are the following as shown in figure 1.13:

- An input buffer for input neurons (NBin)
- An output buffer for output neurons (NBout)
- A third buffer for synaptic weights (SB)
- The Neural Functional Unit (NFU)
- The control logic (CP)

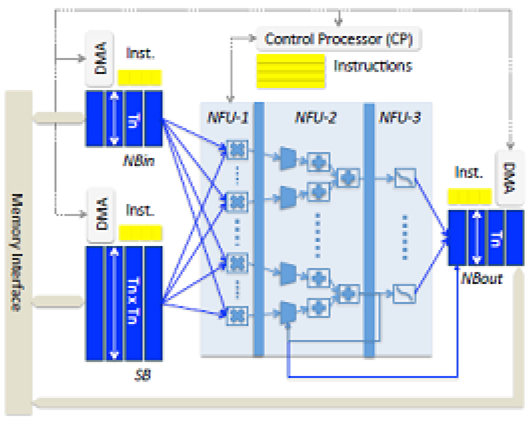
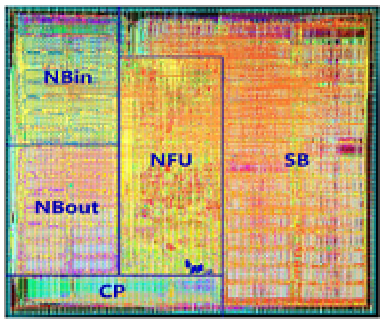
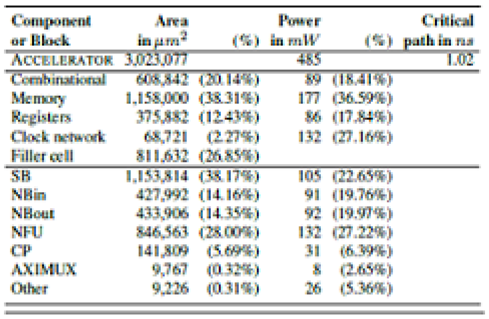


Figure 1.13: [9] Accelerator

The NFU block has the goal to reflect the decomposition of a computational layer.

Furthermore, this structure is pipelined and each layer performs different operations. The CP block describes the accelerator control and guides the execution of the DMA, the three buffers and the NFU. The calculation of each layer is decomposed into different stages and the instructions are stored in the associated SRAM Through the use of accelerator for Machine Learning we can achieve high performance by implementing it in a very small area. In fact, as shown in figure 1.14 we get to get:





Component or Block	Area in μm^2	(%)	Power in mW	(%)	Critical path in ns
ACCELERATOR	3,023,077		485		1.02
Combinational	608,842	(20.14%)	89	(18.41%)	
Memory	1,158,000	(38.31%)	177	(36.59%)	
Registers	375,882	(12.43%)	86	(17.84%)	
Clock network	68,721	(2.27%)	132	(27.16%)	
Filler cell	811,632	(26.85%)			
SB	1,153,814	(38.17%)	105	(22.65%)	
NBin	427,992	(14.16%)	91	(19.76%)	
NBout	433,906	(14.35%)	92	(19.97%)	
NFU	846,563	(28.00%)	132	(27.22%)	
CP	141,809	(5.69%)	31	(6.39%)	
AXIMUX	9,767	(0.32%)	8	(2.65%)	
Other	9,226	(0.31%)	26	(5.36%)	

Figure 1.14: [9] Layout and Characteristics of accelerator

- A speed of 117.87x
- An energy reduction of 21.08x with a frequency of 2GHz at 128 bit with a normal hierarchical cache with a 65nm design.

1.3.2 PuDianNao: A Polyvalent Machine-Learning Accelerator

PuDianNao [10] is a hardware accelerator supporting several Machine Learning techniques. This accelerator consists of:

1. Several Functional Units (FUs). It is the basic unit of execution. Each FU consists of two parts:
   - Machine-Learning Unit (MLU) that supports basic calculations that occur in Machine Learning techniques

  - Arithmetic Logic Unit (ALU) that is a unit of calculation that carries out operations not supported by the MLU

2. Three data buffers
   - HotBuf
   - ColdBuf
   - OutputBuf
3. An instruction buffer (InstBuf)
4. A control module
5. A Direct Memory Access (DMA).

as shown in the figure 1.15.

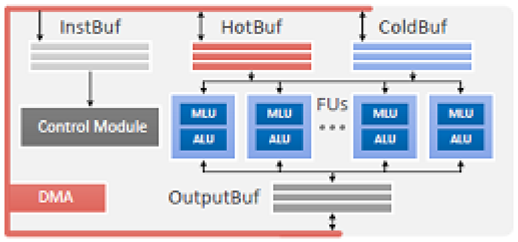


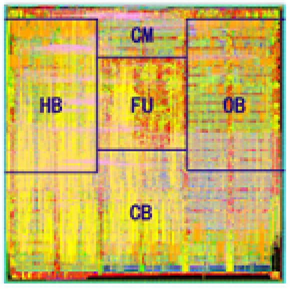
Figure 1.15: [10] Accelerator architecture of PuDianNao

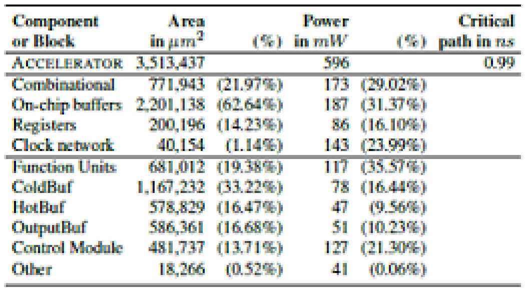
Considering the current version of PuDianNao with 16 MLUs, we have:

- Total $Area = 3{,}55mm^2$
- Total $PowerConsumption = 596mW$
- $CriticalPathDelay = 0{,}99ns$

as shown in figure 1.16.
However, comparing the performance of PuDianNao with GPU baseline, we have:

- The average speedup of PuDianNao over the GPU is $1{,}20x$
- PuDianNao reduces the energy consumption of the GPU by $128.41x$





Component or Block	Area in μm^2	(%)	Power in mW	(%)	Critical path in ns
ACCELERATOR	3,513,437		596		0.99
Combinational	771,943	(21.97%)	173	(29.02%)	
On-chip buffers	2,201,138	(62.64%)	187	(31.37%)	
Registers	200,196	(14.23%)	86	(16.10%)	
Clock network	40,154	(1.14%)	143	(23.99%)	
Function Units	681,012	(19.38%)	117	(35.57%)	
ColdBuf	1,167,232	(33.22%)	78	(16.44%)	
HotBuf	578,829	(16.47%)	47	(9.56%)	
OutputBuf	586,361	(16.68%)	51	(10.23%)	
Control Module	481,737	(13.71%)	127	(21.30%)	
Other	18,266	(0.52%)	41	(0.06%)	

Figure 1.16: [10] Layout and Characteristics of accelerator

1.3.3 Eyeriss: An Energy-Effivient Reconfigurable Accelerator

Eyeriss [11] is an implemented and fabricated CNN accelerator that can support high throughput, optimizing for the energy efficiency of the all system, including the accelerator chip and off-chip DRAM. Furthermore, it is also reconfigurable to handle different CNN shapes, considering different filters shapes.

The Eyeriss accelerator as seen in figure 1.17 is composed as follows:

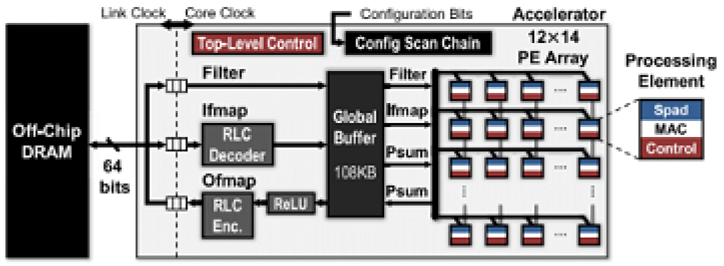


Figure 1.17: [11] Eyeriss system architecture

1. A spatial architecture composed of a processing element matrix.

2. A Row Stationary (RS), that is a CNN dataflow, that is used to map the computation of a given CNN shape optimizing the energy efficiency.

3. A network-on-chip (NoC) architecture that supports the RS dataflow. It is composed of:

   - Global Input Network (GIN).

- Global Output Network (GON).
- Local Network.

4. Run-length compression (RLC) and PE data gating are used to increase energy efficiency.

This architecture has two clock domains that run independently:

1. The core clock domain for processing is composed of:

    - A spatial array of PEs.
    - A Global Buffer (GLB). It can communicate with the DRAM through an asynchronous interface and with the PE through the NoC. This structure stores all types of data.
    - An RLC CODEC.
    - An ReLU module.

2. The link clock domain for communication with the off-chip DRAM through a bidirectional data bus.

The Eyeriss chip was implemented in 65-nm CMOS and had been integrated into the Caffe framework as shown in figure 1.18.

Eyeriss's performance, including both the chip's energy efficiency and access to DRAM. The DRAM is addressed using two different CNNs see in the following tables 1.19 and 1.20:

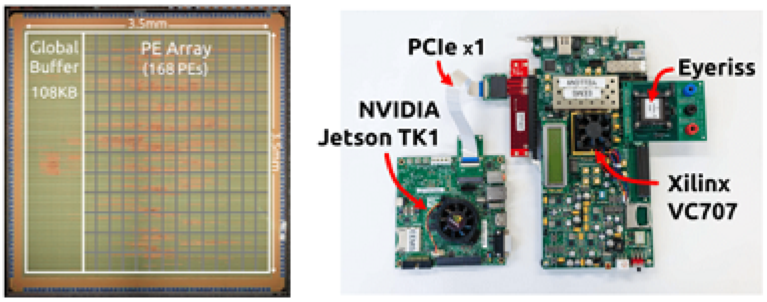
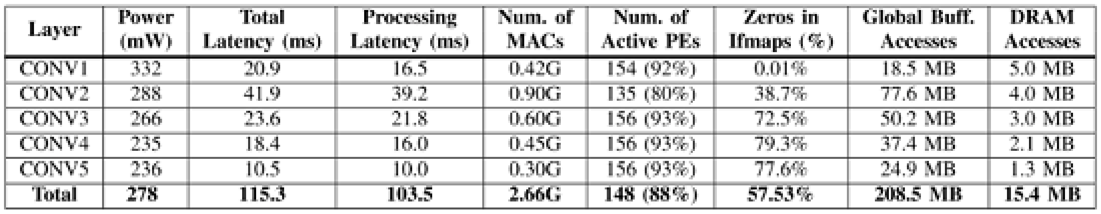


Figure 1.18: [11] Layout and evaluation board

- AlexNet: the power consumption of the chip decreases gradually through the depth of the layer.
  - Reaches a frame rate of $34.7 frames/s$
  - The power of the chip is $278mW$
  - Energy efficiency is $83.1GMACS/W$ compared to the maximum equal to $122.8GMACS/W$ at $0.82V$.
  - The current throughput is less than the maximum throughput of $45frame/s$ at $1.17V$.
- VGG-16: the performance depends on both the calculation and the shape of the configuration.
  - The chip operates at $0.7frame/s$
  - Power consumption is equal to $236mW$



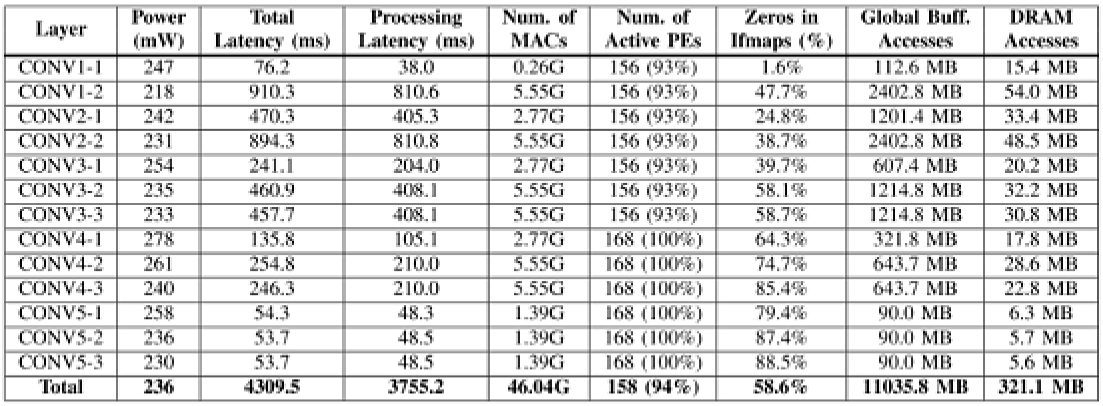
Layer	Power (mW)	Total Latency (ms)	Processing Latency (ms)	Num. of MACs	Num. of Active PEs	Zeros in Ifmaps (%)	Global Buff. Accesses	DRAM Accesses
CONV1	332	20.9	16.5	0.42G	154 (92%)	0.01%	18.5 MB	5.0 MB
CONV2	288	41.9	39.2	0.90G	135 (80%)	38.7%	77.6 MB	4.0 MB
CONV3	266	23.6	21.8	0.60G	156 (93%)	72.5%	50.2 MB	3.0 MB
CONV4	235	18.4	16.0	0.45G	156 (93%)	79.3%	37.4 MB	2.1 MB
CONV5	236	10.5	10.0	0.30G	156 (93%)	77.6%	24.9 MB	1.3 MB
Total	**278**	**115.3**	**103.5**	**2.66G**	**148 (88%)**	**57.53%**	**208.5 MB**	**15.4 MB**

Figure 1.19: [11] AlexNet results

1.3.4 Intelligence Bosting Engine (IBE)

IBE (Intelligence Boost Engine) [12] is a hardware accelerator to process machine learning algorithms for SLH200 sensor hub SoC. The internal architecture shown in figure 1.21 consists of:

- Control unit (CU): which controls the operations of IBE and executes them. It also stores information about the mode of operation and the size of the input data.
- DMA: which loads the incoming data and stores the results independently.



Layer	Power (mW)	Total Latency (ms)	Processing Latency (ms)	Num. of MACs	Num. of Active PEs	Zeros in Ifmaps (%)	Global Buff. Accesses	DRAM Accesses
CONV1-1	247	76.2	38.0	0.26G	156 (93%)	1.6%	112.6 MB	15.4 MB
CONV1-2	218	910.3	810.6	5.55G	156 (93%)	47.7%	2402.8 MB	54.0 MB
CONV2-1	242	470.3	405.3	2.77G	156 (93%)	24.8%	1201.4 MB	33.4 MB
CONV2-2	231	894.3	810.8	5.55G	156 (93%)	38.7%	2402.8 MB	48.5 MB
CONV3-1	254	241.1	204.0	2.77G	156 (93%)	39.7%	607.4 MB	20.2 MB
CONV3-2	235	460.9	408.1	5.55G	156 (93%)	58.1%	1214.8 MB	32.2 MB
CONV3-3	233	457.7	408.1	5.55G	156 (93%)	58.7%	1214.8 MB	30.8 MB
CONV4-1	278	135.8	105.1	2.77G	168 (100%)	64.3%	321.8 MB	17.8 MB
CONV4-2	261	254.8	210.0	5.55G	168 (100%)	74.7%	643.7 MB	28.6 MB
CONV4-3	240	246.3	210.0	5.55G	168 (100%)	85.4%	643.7 MB	22.8 MB
CONV5-1	258	54.3	48.3	1.39G	168 (100%)	79.4%	90.0 MB	6.3 MB
CONV5-2	236	53.7	48.5	1.39G	168 (100%)	87.4%	90.0 MB	5.7 MB
CONV5-3	230	53.7	48.5	1.39G	168 (100%)	88.5%	90.0 MB	5.6 MB
Total	**236**	**4309.5**	**3755.2**	**46.04G**	**158 (94%)**	**58.6%**	**11035.8 MB**	**321.1 MB**

Figure 1.20: [11] VGG-16 results

- Internal Buffer (IB).
- Processing Element (PE): This architecture takes data from the sensor and creates a matrix. The data enter the structure that is able to process both matrices and vectors. In this case, each PE contains 12 MACs (multiplier and accumulator).

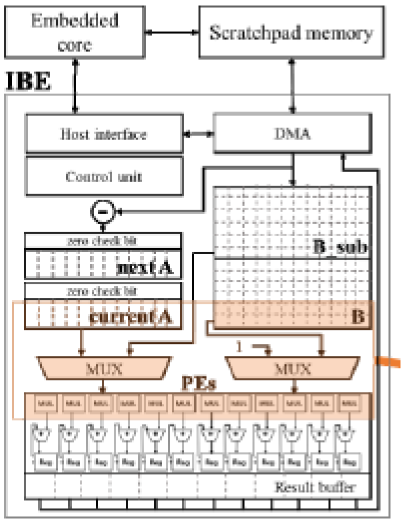


Figure 1.21: [12] IBE accelerator

The SLH200 sensor hub SoC with IBE was fabricated with 55 nm process.

The performance estimation was performed in the evaluation board having SLH200. The results obtained, as shown in figure 1.22, are:

- IBE occupies about 10% of the chip area.
- IBE can reduce the energy consumption from 45% to 75% for target applications.

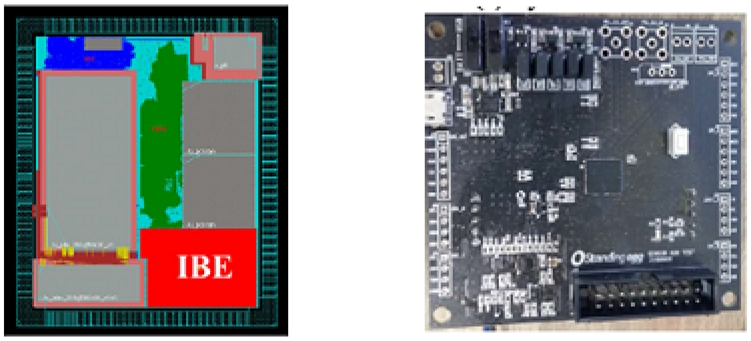


Figure 1.22: [12] SLH200 layout and evaluation board

1.4 Machine Learning Architectures

The machine learning algorithms are susceptible to acceleration due to a high degree of parallelism of calculation.

Let's now analyze a series of architectures that use hardware accelerators to implement Machine Learning algorithms.

1.4.1 DaDianNao: A Machine-Learning Supercomputer

We define the architecture DaDianNao [13] as "supercomputer" because its aim is to achieve high machine-learning performance,using a Hardware Accelerator DianNao [9] with a multi-chip system, beyond single-GPU performance. Furthermore, each node is cheaper than a single-GPU even if exhibiting a same or higher compute density. The main issue is the memory storage and bandwidth requirements of the synapses. We solve this issue by adopting the following design steps:

1. To save both time and energy, we create an architecture fully distributed fully where synapses are stored near to the neurons which will use them.In this way, we must not use a main memory.

2. We create an asymmetric architecture where each node is direct toward storage than computations.

3. To limit the external bandwidth, we transfer neurons values instead synapses values because the first are fewer than the seconds in the layer.

4. We authorize high internal bandwidth by dividing the local storage into many tiles.

The general architecture seen in figure 1.23 is a set of same nodes, one per chip, that are inserted in a mesh topology. Each node contains:

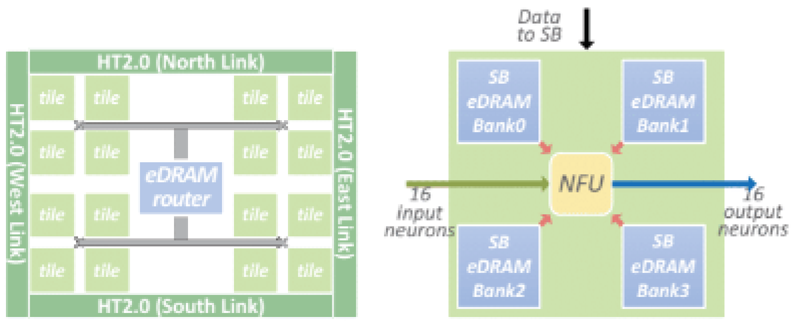


Figure 1.23: [13] The based organized of a node and tile architecture

- 16 tiles
- 2 central eDRAM banks
- Tree interconnect

While, each tile contains:

- Neural Functional Unit (NFU)
- 4 eDRAM banks
- Input/Output interfaces to/from the central eDRAM banks

A tile implements a neural functional unit (NFU) that has parallel digital arithmetic units; these units are supplied with data coming from memory banks eDRAM via buffer. The dominant data structures in CNN and DNN are the synaptic weight matrices that define neuronal states. These are distributed across multiple tiles in different eDRAM memory banks. The calculations involving these weights are then performed on adjacent NFUs, thus obtaining the processing of neighboring data. This requires moving the outputs of the previous neuron layer to the relevant tiles in order to obtain the outputs of the current level. These outputs are then routed to the eDRAM banks to be used as input for the next level. Most of the chip area is, however used to store synaptic weights in eDRAM. Each NFU as in figure 1.24, implemented through a DianNao accelerator, is composed of:

- Adder block
- Multiplier block
- Transfer bus

This block works with two modes of operation:

- Inference mode: where the network processes a model.
- Training mode: where the inputs of the various layers are analyzed and the weights are modified to train the network.

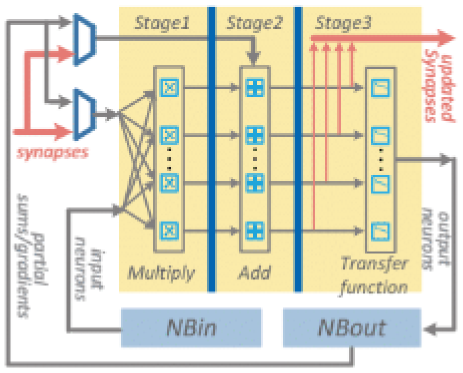
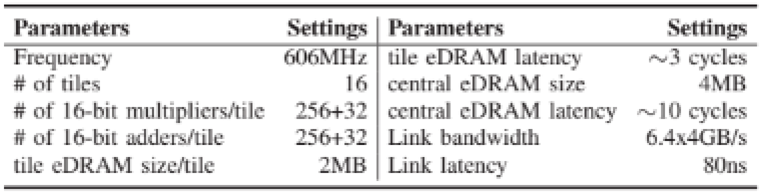


Figure 1.24: [13] The different operations of an NFU

Each NFU processes 16 inputs and outputs 16 outputs. In fact, for each set of input neurons transmitted to all the tiles, different output neurons are calculated on the same hardware neuron. The intermediate values of these neurons are stored locally in the eDRAM tile. When the calculation of an output neuron is terminated, the value is sent through the tree in the corresponding central eDRAM. The number of NFU is much smaller than the number of neurons in a level. Therefore, the NFUs are shared by multiple neurons in the multiplexed mode. A single CNN layer is processed at a time through parallel processing on all NFUs in the system and the outputs are collected in eDRAM banks. Once a level has been completely processed, DaDianNao goes to the next level, again parallel to all the NFUs in the system. The architecture characteristics are summarized in Table 1.25 where we have implemented a structure with 16 tiles per node with 4 eDRAM banks. The

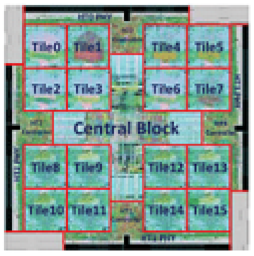


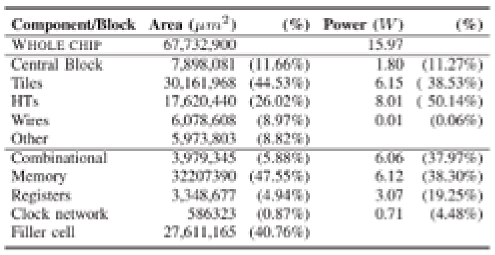
Parameters	Settings	Parameters	Settings
Frequency	606MHz	tile eDRAM latency	~3 cycles
# of tiles	16	central eDRAM size	4MB
# of 16-bit multipliers/tile	256+32	central eDRAM latency	~10 cycles
# of 16-bit adders/tile	256+32	Link bandwidth	6.4x4GB/s
tile eDRAM size/tile	2MB	Link latency	80ns

Figure 1.25: [13] Architecture characteristics

whole chip as in figure 1.26 has:

- $TotalArea = 67{,}732{,}900 \mu m^2$
- $TotalPowerConsumption = 15{,}97W$





Component/Block	Area (μm^2)	(%)	Power (W)	(%)
WHOLE CHIP	67,732,900		15.97	
Central Block	7,898,081	(11.66%)	1.80	(11.27%)
Tiles	30,161,968	(44.53%)	6.15	(38.53%)
HTs	17,620,440	(26.02%)	8.01	(50.14%)
Wires	6,078,608	(8.97%)	0.01	(0.06%)
Other	5,973,803	(8.82%)		
Combinational	3,979,345	(5.88%)	6.06	(37.97%)
Memory	32207390	(47.55%)	6.12	(38.30%)
Registers	3,348,677	(4.94%)	3.07	(19.25%)
Clock network	586323	(0.87%)	0.71	(4.48%)
Filler cell	27,611,165	(40.76%)		

Figure 1.26: [13] Layout and node characteristic

where the area contribution is:

- 44.53% is used for the 16 chip tiles
- 26.02% is used for the four HT IPs
- 11.66% is used for the central block 8.97% is used for connections between the central block and the tiles

while the power contribution is:

- 38.30% is consumed by the memory cells
- 37.97% is consumed by the combinatorial logic
- 19.25% is consumed by the registers.

1.4.2 ISAAC: A Convolutionam Neural Network Accelerator

The ISAAC chip structure [14] is very similar to DaDianNao one [13] organized in nodes/tiles with crossbar arrays used for the calculation at the center of each tile. The high-level structure in figure 1.27 consists of a number of tiles (labeled T), attached with an on-chip mesh (C-mesh).

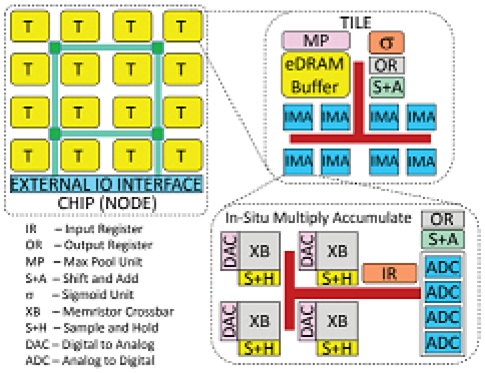


Figure 1.27: [14] ISAAC architecture hierarchy

Each tile consists of:

- eDRAM buffer to save input values

- a number of In-situ Multiply-Accumulated units (IMA)
- output registers to add the results via a shared bus.
- shift-and-add units
- sigmoid unit
- max-pool unit

Each IMA is made up of:

- crossbar and ADC arrays connected with a shared bus.
- input/output logs
- shift-and-add unit.

Let's analyze the crossbar structure shown in figure 1.28:

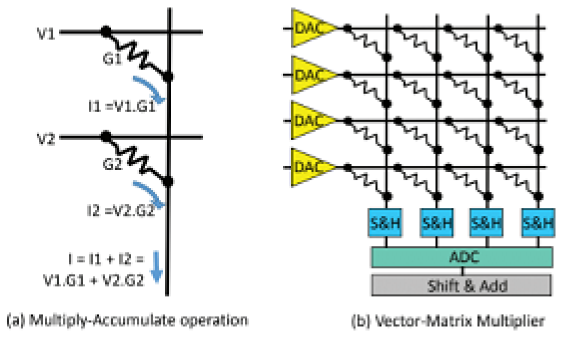


Figure 1.28: [14] Crossbar structure

Each BL is connected to the WL through resistive memory cells. Assuming that the cells are programmed for resistances R, and conductivity G, applying a voltage V to all the cells, each column will produce a current. The sum of these currents represents the value of the product between vectors. Input voltages are applied to the lines and the current flowing in each BL represents the output of a neuron. Each neuron is powered by the same input but with different synaptic weights. This model reaches high levels of parallelism. In this structure we have the presence of:

- DAC which converts the digital input into a voltage value to be applied to the line.
- Sample-and-Hold ($S\&H$) which receives the current from the BL
- ADC that converts currents to digital before being passed to digital circuits for the sum.

The resistive memory elements are implemented through a 1T1R structure they are characterized by the series of resistor and transistor as shown in figure 1.29.

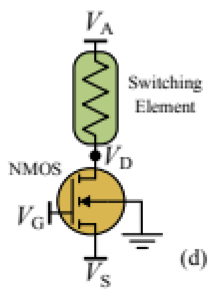


Figure 1.29: [14] Resistive memory element 1T1R structure

ISAAC uses memristor arrays to store not only synaptic weights, but also to perform calculations on them. Initially there is a training phase where the machine learns the model, in fact after the training has determined the weights for each neuron, the weights are loaded inside the memristors with a programming phase. During prediction, inputs are provided to ISAAC by an I/O interface and addressed to the tiles that make up the first layer of CNN. A finite state machine sends the inputs to the respective IMAs. Scalar product operations are performed on crossbar arrays, which results are sent to the ADCs and aggregated in the output registers. The result is sent via the sigmoid operator and stored in the eDRAM banks of the tiles that process the next level. The process continues until the final level generates an output that is sent to the I/O interface also used to communicate with other ISAAC chips. Therefore, unlike DaDianNao, ISAAC tiles/IMAs must be partitioned between the different CNN levels. ISAAC's behavior depends of many parameters:

- The size of the memristor crossbar array

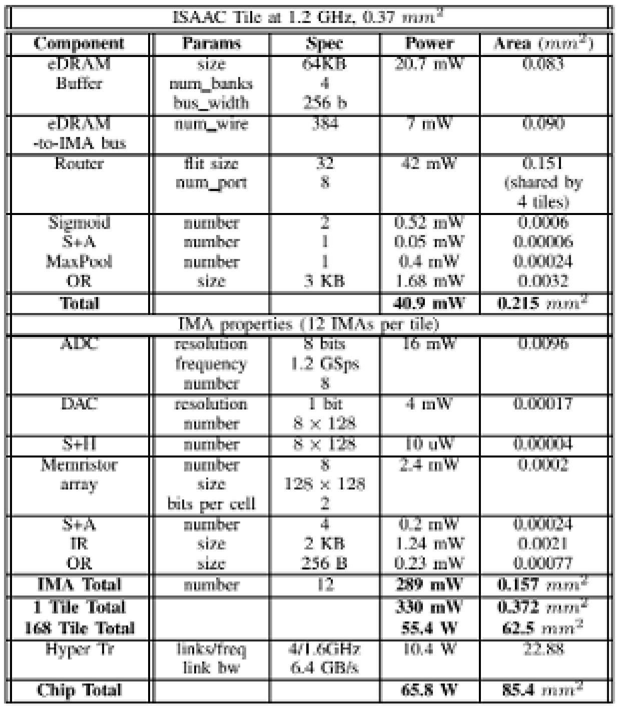
- The number of IMAs in a tile.
- The number of crossbars in an IMA
- The number of ADCs in an IMA

The following table 1.30 shows the characteristic parameters of the ISAAC architecture. It should be noted that with regard to the area:

- ADC occupies 31% of the tile area
- eDRAM buffer and eDRAM IMA bus occupy 47%.

for the power:

- ADC consumes 58% of the power of the tiles.



ISAAC Tile at 1.2 GHz, 0.37 mm^2				
Component	**Params**	**Spec**	**Power**	**Area (mm^2)**
eDRAM Buffer	size num_banks bus_width	64KB 4 256 b	20.7 mW	0.083
eDRAM -to-IMA bus	num_wire	384	7 mW	0.090
Router	flit size num_port	32 8	42 mW	0.151 (shared by 4 tiles)
Sigmoid S+A MaxPool OR	number number number size	2 1 1 3 KB	0.52 mW 0.05 mW 0.4 mW 1.68 mW	0.0006 0.00006 0.00024 0.0032
Total			**40.9 mW**	**0.215** mm^2
IMA properties (12 IMAs per tile)				
ADC	resolution frequency number	8 bits 1.2 GSps 8	16 mW	0.0096
DAC	resolution number	1 bit 8×128	4 mW	0.00017
S+H	number	8×128	10 uW	0.00004
Memristor array	number size bits per cell	8 128×128 2	2.4 mW	0.0002
S+A IR OR	number size size	4 2 KB 256 B	0.2 mW 1.24 mW 0.23 mW	0.00024 0.0021 0.00077
IMA Total	number	12	**289 mW**	**0.157** mm^2
1 Tile Total **168 Tile Total**			**330 mW** **55.4 W**	**0.372** mm^2 **62.5** mm^2
Hyper Tr	links/freq link bw	4/1.6GHz 6.4 GB/s	10.4 W	22.88
Chip Total			**65.8 W**	**85.4** mm^2

Figure 1.30: [14] ISAAC parameters

Furthermore this architecture can achieve an improvement over the DaDianNao of:

- 14.8x throughput
- 5.5x for energy
- 7.5x for calculation density

using a resistive memory to perform many calculations in parallel.

1.4.3 LookNN: Neural Network with No Moltiplication

A new type of architecture is the LookNN [15] through which we implement a method to place multiplications with look up tables. For each neuron, LooKNN stores all possible input combinations and corresponding outputs in a look up table. During execution, each neuron searches the look-up table for the result of the multiplication previously stored. This architecture seen in figure 1.31 is divided into three sections:

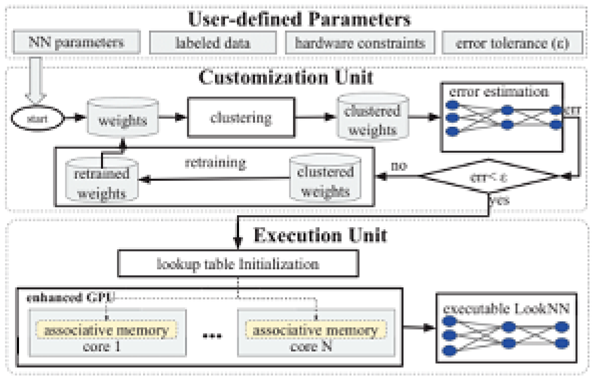


Figure 1.31: [15] Global flow of LookNN

1. Definition of parameters by User:
    - Error tolerance
    - Hardware constrain
    - Training date
    - Baseline NN.

2. Customization Unit: modifying the NN mapping it into the LooKNN. In this unit, three blocks are defined iteratively to reduce the error:
    - Weight clustering: where each row of the array is partitioned into N clusters using a k-means algorithm for clustering.
    - Error estimation: where the classification error is calculated.
    - Weight retraining: to compensate these errors the NN is reorganized reducing clustering errors.
3. Execution unit: where the GPU is used to implement the LOOKNN. Each GPU core has access to an associated memory whose energy and execution time depend on the size of the look up table.

The calculation of each neuron is associated with one of the GPU cores. We use the crossbar memristor to implement both the TCAM and the crossbar memory. The memory shown in figure 1.32 is characterized by:

- TCAM: where the values are stored on cells based on NVM resistors. Low resistance identifies the logical state 1 while high resistances denotes logic state 0. Before searching for operands, all lines are preloaded to V_{dd} voltage. During the search, the buffer distributes the inputs across the line. The sense amplifier (SA) samples and at each clock identifies a hit.
- Multi-Stage TCAM: where it reduces its switching activity splitting it in multiple stages reducing energy but increasing the delay.
- Memory crossbar: which consumes less energy occupying a negligible area
- Associative memory configuration.

Before execution, the first TCAM is initialized with N_q quantized values while the second TCAM is initialized with $N_{cluster}$ shared weights. The crossbar memory contains the outputs of the multiplications. During execution, for two inputs, the two TCAMs are analyzed in parallel, the address decoder generates the correct address and the multiplication result is taken from the crossbar memory. To evaluate the following architecture, 4 different applications were analyzed:

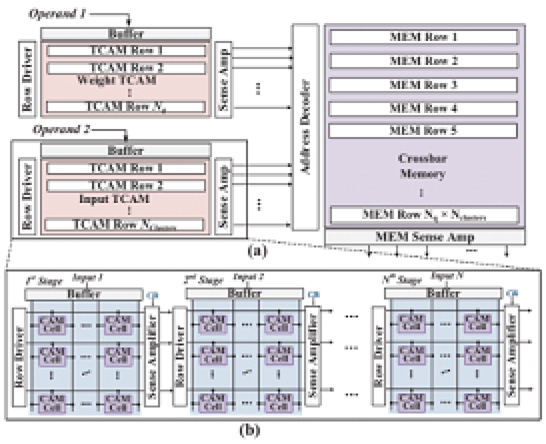


Figure 1.32: [15] LookNN hardware for memory-based computation

- Voice Recognition
- Hyperspectral Imaging
- Human activity Recognition
- MNIST

For each dataset we compare the baseline with the LookNN architecture that uses associative memory. We define the additional error as the difference between the lookNN error and the baseline error. We note that increasing the quantization number N_q and cluster $N_{cluster}$ the error decreases, also in some configurations the error is negative implying that the LookNN architecture can reach a lower error-rate. Furthermore, the contribution of energy and time of execution is modified leading to an improvement:

- 2.2x on energy
- 2.5x on the speed with an additional zero error.

But you can still increase the performance up

- 3x for energy
- 2.6x for speed with an additional error of less than 0.2

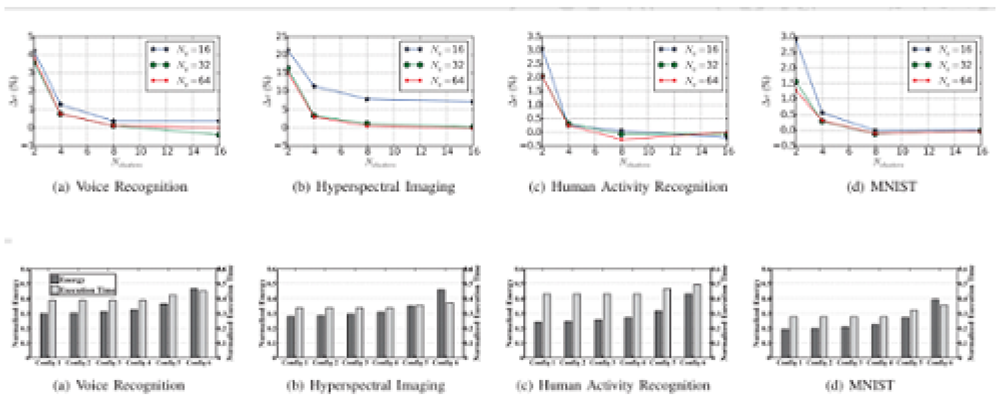


Figure 1.33: [15] Results of LookNN in different configuration of NN

1.4.4 A ML Classifier Implementated in a Standard 6T SRAM Array

The architecture seen in the figure 1.35 shows an in-memory-classifier architecture [16]. We analyze a Machine Learning classifier where storage and calculations are done in a standard 6T SRAM. The calculation is performed on site by the memory cells, avoiding high-energy accesses. It presents limitations because the 6T SRAM structure limits the possible computations and the non-linearity of the circuit degrades the quality of the output.

To solve these problems, an ensemble learning is used, ie a series of methods are used to obtain a better predictive performance. At the base are the Weak classifier that can classify only 50% + 1 of the samples and Strong classifier that are well correlated with the real classification as shown in figure 1.34.

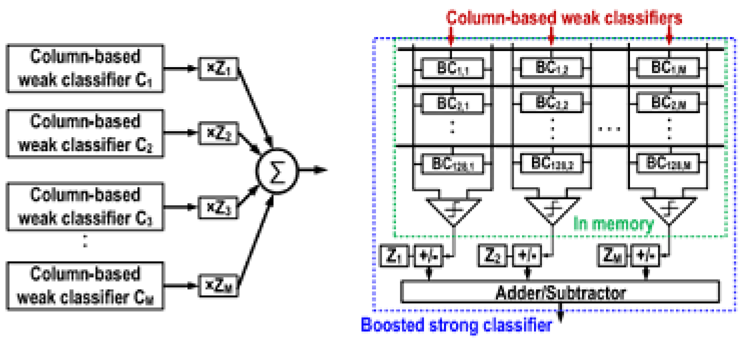

Column-based weak classifier C₁
Column-based weak classifier C₂
Column-based weak classifier C₃
Column-based weak classifier C_M
×Z₁
×Z₂
×Z₃
×Z_M
Σ
Column-based weak classifiers
BC₁,₁
BC₁,₂
BC₁,M
BC₂,₁
BC₂,₂
BC₂,M
BC₁₂₈,₁
BC₁₂₈,₂
BC₁₂₈,M
In memory
Z₁
Z₂
Z_M
+/-
Adder/Subtractor
Boosted strong classifier

Figure 1.34: [16] Ensemble learning structure

The aim is to train M classifier with different training sets and combine the results. The system consists of a standard 6T bit-cell array that works two operating modes:

1. In SRAM mode, the operation is the reading/writing of digital data, where machine-learning models derived from training are saved in bit-cells.
2. In Classify mode, where each SRAM column forms a weak classifier and all wordlines (WL) are driven to analogue voltages. To create a strong classifier from multiple base weak classifier we use Boosting Algorithm.

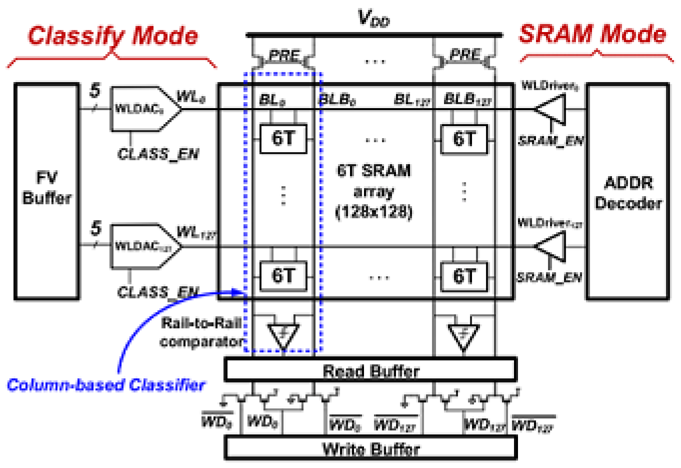


Figure 1.35: [16] Architecture of in-memory classifier

Let's analyze the classify mode starting from the weak column-base classifier inside the memory classifier architecture shown in figure 1.36.

The BL/BLB pair is preloaded. The WLs are supplied with an analog voltage corresponding to the input value x leading to the I_{BC} currents. Each current is applied to both BL and BLB depending on the data stored in the bit-cell. Considering both the bit-lines (BL/BLB) as differential signals, the stored data are evaluated as the product of the value of the current I_{BC} for the weight -1 or +1. Finally, all the bit-cell currents are added together to make the line and a sign of the threshold voltage is defined through a comparator.

A training algorithm is needed to overcome the non-idealities of weak classifiers. In this specific case, the Boosting algorithm (AdaBoost) is used when the outputs

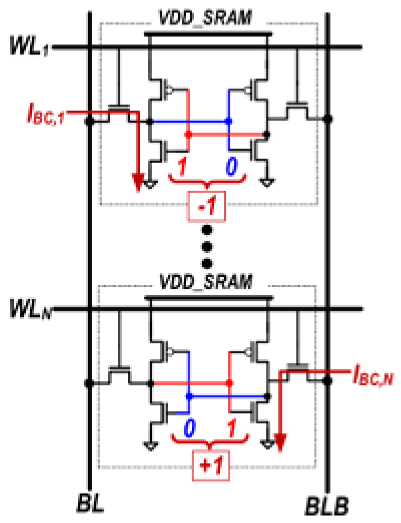


Figure 1.36: [16] Column-based weak classifier of in-memory classifier architecture

of the weak classifiers are combined into a weighted sum that represents the final output of the enhanced classifier. We also use an extension of the latter Error Adaptive Classifier Boosting (EACB) to correct non-linearity and assembly errors. Once all the basic weak classifiers are trained, the strong classifier is constructed by weighted voting against the decisions of weak classifiers as shown in figure 1.34. An important component is the DAC which is used to convert digital data to analog one before entering in memory. The bits identify the values of the input to the DAC and are formed by a weighted binary source of pmos. The current I_{DAC} is converted into voltage WL which enables a bit-cell replica. The driver transistor $M_{D,R}$ receives a gate voltage V_{DD} by enabling the $Class\_EN$ signal when the structure is in classifier mode. The transistor $M_{A,R}$ is self-polarized to generate a voltage corresponding to the current I_{DAC}. As a result, I_{DAC} is mirrored by creating an $I_{B,C}$. The design is quite independent of how the number of cells in the column increases, since nominally the ratio of aggregate current to capacity remains constant.

We also have an array of cells where the data are stored and n SA that is used to compare the results of each column.

A prototype SRAM array is implemented in 130nm CMOS shown in figure 1.39 for an MNIST dataset. This prototype operates:

- In SRAM-mode up to 300MHz

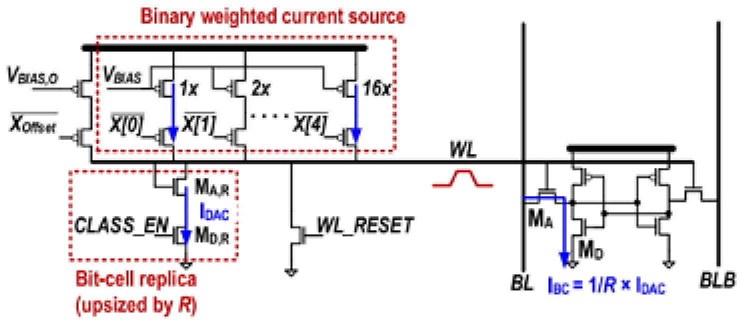


Figure 1.37: [16] WLDAC circuit

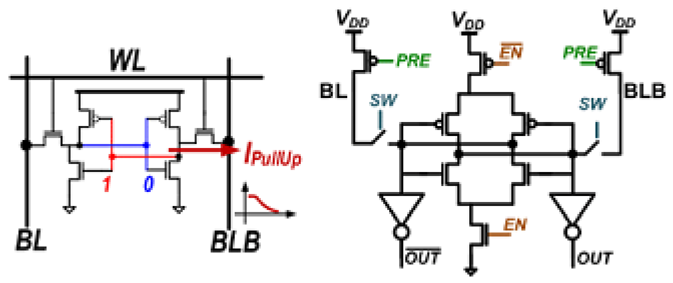
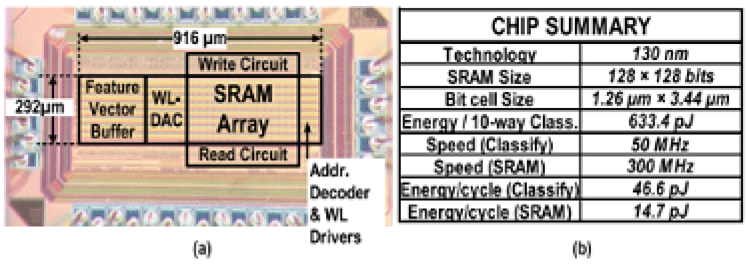


Figure 1.38: [16] Bit-cell and Sense amplifier circuits

- In Classify-mode up to $50MHz$ generating a classification every cycle.

In addition, the prototype achieves improvements from an energy point of view compared to the discrete system of the traditional training algorithm.


916 µm
292µm
Feature Vector Buffer
WL-DAC
Write Circuit
SRAM Array
Read Circuit
Addr. Decoder & WL Drivers
(a)
(b)

CHIP SUMMARY	
Technology	130 nm
SRAM Size	128 × 128 bits
Bit cell Size	1.26 µm × 3.44 µm
Energy / 10-way Class.	633.4 pJ
Speed (Classify)	50 MHz
Speed (SRAM)	300 MHz
Energy/cycle (Classify)	46.6 pJ
Energy/cycle (SRAM)	14.7 pJ

Figure 1.39: [16] Prototype die and measurement-summary table

1.4.5 PRIME: A Novel PIM architecture for Neural Network

PRIME, is an architecture that aims to accelerate the neural networks exploiting the computational capacity of ReRAM and PIM architectures [18]. PRIME acts just like a real in-memory architecture as shown in figure 1.40, as it does not add logic components, but uses the same memory array for calculations, which leads to area savings. The additions of PRIME hardware concern changes to memory device circuits.

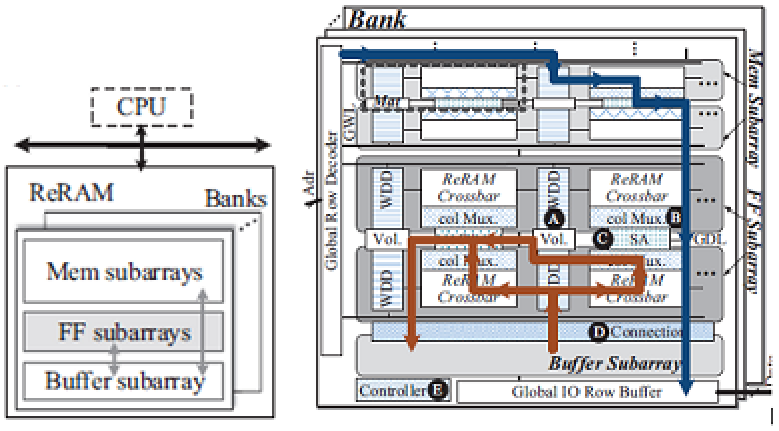


Figure 1.40: [18] PRIME architecture

The ReRAM counters are divided into:

- Memory subarray (Mem) used as storage
- Full Function subarrays (FF) that can work both in computation mode and in memory mode
- Buffer subarrays, which act as both buffers and storage, if it is as storage a memory is needed.

PRIME controller is the main part of the architecture. A important role of the controller is to configure the FF subarray both in memory and computation modes. It decodes instructions and gives control signals to all the peripheral circuits.

PRIME is simulated and compared with CPU-only, NPU co-processor and NPU-PIM processor solutions. Among the all solutions, PRIME architecture as shown in figure 1.41.

- Achieves the highest speed up over CPU-only solution and about 4.1x of pNPU.
- Shows superior energy-efficiency than to the other solutions.
- Reduces all the parts of energy consumption: computation energy, buffer energy, and memory energy.
- Only incurs 5.76% area overhead.

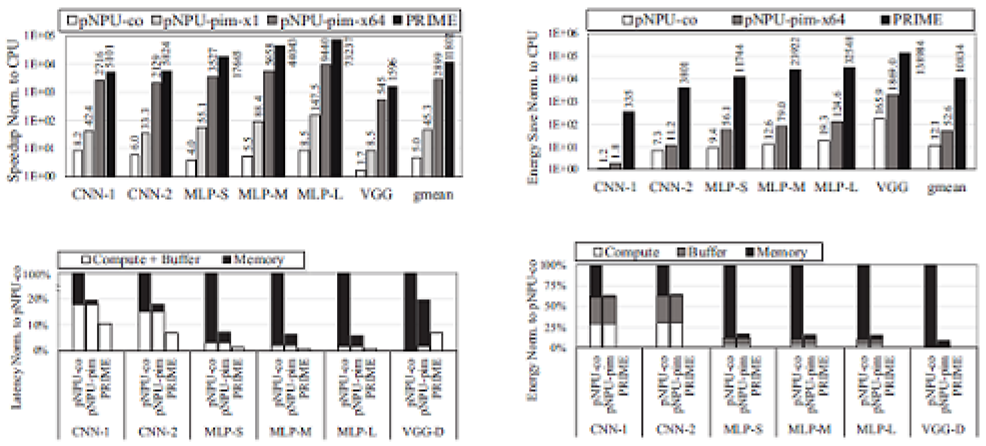


Figure 1.41: [18] Results of time, speed-up and energy of PRIME structure

1.5 Hardware implementation of Decision Tree Classifier

The decision tree classifier is created using an axis-parallel decision tree implementation [23]. In order to implement this classifier a threshold network is made through a series of combinational logic circuits.
In detail, it implements a binary decision tree classifier as shown in figure 1.42.

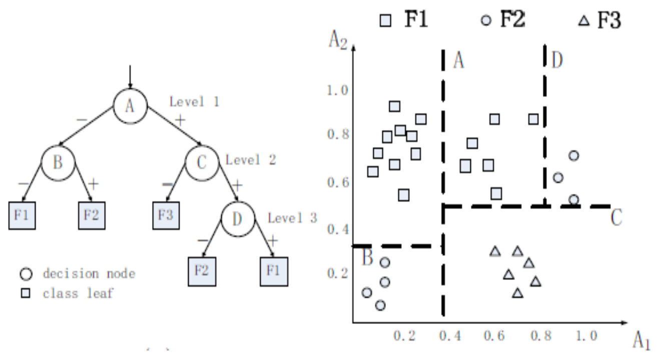


Figure 1.42: [23] Axis-parallel Decision Tree classifier structure

The classification starts at the root node and ends at the leaf node through different paths using two different branches. The axis-parallel implementation implements each single tree using different modules. This implementation is characterized by an high latency. In order, to overcome this problem, each node is identified as an universal node of the decision tree architecture and different nodes are separated by one pipe stage for each level.
A second approach is called Oblique Decision Tree classifier [24] and it is created checking the similarity between tree structure and the threshold network.
This classification is shown in figure 1.43.
In details, each threshold block is implemented by two-dimensional array architecture made up using a processing element which stores the threshold value.

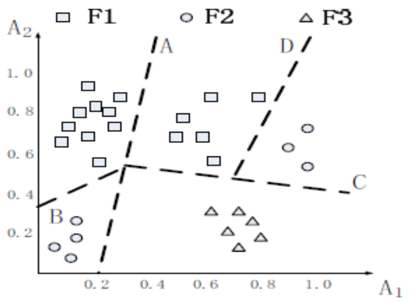


Figure 1.43: [23] Oblique Decision Tree classifier classification

The proposed architecture [23] is shown in figure 1.44.
The classifier is composed of tree blocks:

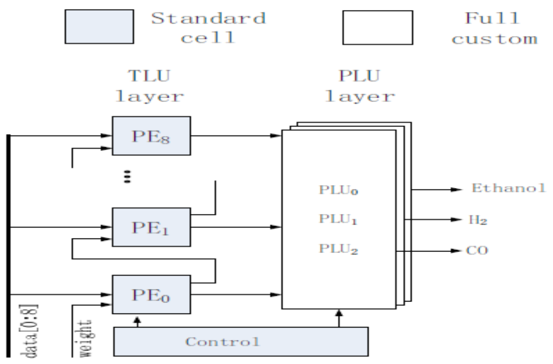


Figure 1.44: [23] Axis-Parallel Decision Tree architecture

- Threshold Unit: a processing element(PE) where the weights are compared with the threshold value.
- The Programmable Unit (PLU): the structure is divided in several parts:
  - Decoder
  - Transistor switches
  - Comparator
  - Latches
- Control Unit: it creates a synchronous signal to generate the control

In order to test this implementation is used a 180nm technology and the layout is shown in figure 1.45.

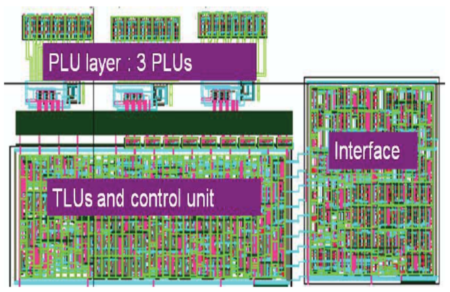
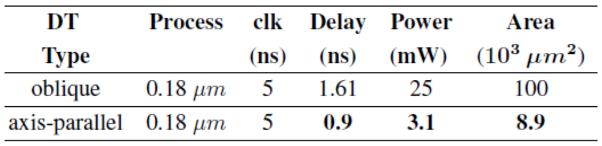


Figure 1.45: [23] Layout of Axis-Parallel Decision Tree implementation

These two implementation are compared and the results in terms of timing, area and power are shown in figure 1.46.



DT Type	Process	clk (ns)	Delay (ns)	Power (mW)	Area (10^3 μm^2)
oblique	0.18 μm	5	1.61	25	100
axis-parallel	0.18 μm	5	**0.9**	**3.1**	**8.9**

Figure 1.46: [23] Comparison results of axis-parallel and oblique implementations

Finally, comparing the results, the axis-parallel classifier is characterized by better performances in term of power and it consumes $3.1mW$ at the frequency equal to $100MHz$.

Chapter 2

Random Forest implementation

2.1 Algorithm Description

In recent years, research fields on machine learning have brought to the development of new classification methods. The most common method is the Ensemble one. This technology is made up of a combination of multiple classifiers, called weak classifiers, in a single model to generate greater precision and accuracy in forecasting. To obtain a unique value from multiple classifiers it is necessary to differentiate the classifiers within the model. To get this differentiation the options are:

- Divide the data set into several parts by modeling a classifier for each of the subgroups obtained. However, this could lead to tree generation based on small portions of the original dataset which would result in poor performance.
- Bootstrap sampling, a statistical technique that assumes, given a data set D, that a subset of D1 data is created from random selection with repetition of D. instances. Using this technique, M training sets are created, each of which induces a model for each sample, in order to obtain M classifiers that will compose the general model.

Random Forests are part of the so-called Ensemble methods. Specifically, Random Forest is a learning model made up of several decision trees. These are used both for classification analysis and for regression ones. In the analyzed case it is considered the classification analysis. In particular, it is given by the majority voter of the trees predictions classes within it. The decision tree classification has always been one of the most used method in data mining thanks to its characteristics of robustness and adaptation. A decision tree consists of a set of nodes connected each other through the branches creating so graph oriented in descending direction that starts from a single root node and ends in a series of leaf nodes. Starting from the root node, depending on the input data, the observations are attributed to the following nodes. In order to grow a tree it is necessary to identify the field that best separates the attributes in groups where a single class predominates. The initial segmentation produces two nodes, each of which in turn is segmented. If all the results of the node segmentation are equal, the result is identified as a leaf node. Otherwise the algorithm checks all input fields to choose which one has to be divided. In order to

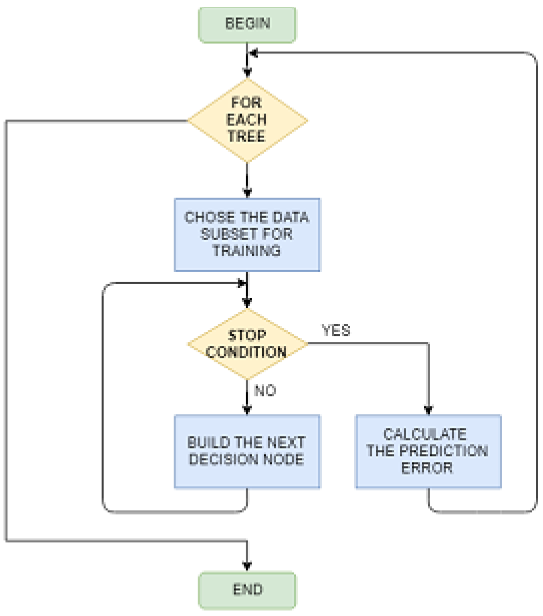

BEGIN
FOR EACH TREE
CHOSE THE DATA SUBSET FOR TRAINING
STOP CONDITION
YES
NO
BUILD THE NEXT DECISION NODE
CALCULATE THE PREDICTION ERROR
END

Figure 2.1: Flow to create Random Forest Algorithm

need avoiding the phenomena of over-fitting, ie the over-processing of the model to the training set. To overcome this problem, it is possible to use the Random forest algorithm. Its aim is to reduce variance through a common decision-making process between different deep trees, induced on different parts of the training set. The used algorithm for forest algorithm applies the bootstrap sampling technique. This technique foresees that, during the construction of the trees, only random selection of dataset are taken than the all features. Typically, for classification problems the number of features to select is equal to the square root of n, where n is the total number of dataset features. Once the training phase is over, the prediction of new instances is performed by calculating, for each classification, the class that is the most present in the predictions of each tree within the forest. The accuracy of the Random Forest depends on the strength of the classification of the individual trees and on the measurement of mutual OOB dependence with which it is possible to estimate not only the error committed, but also the relevance of each attribute for the purpose of classification.

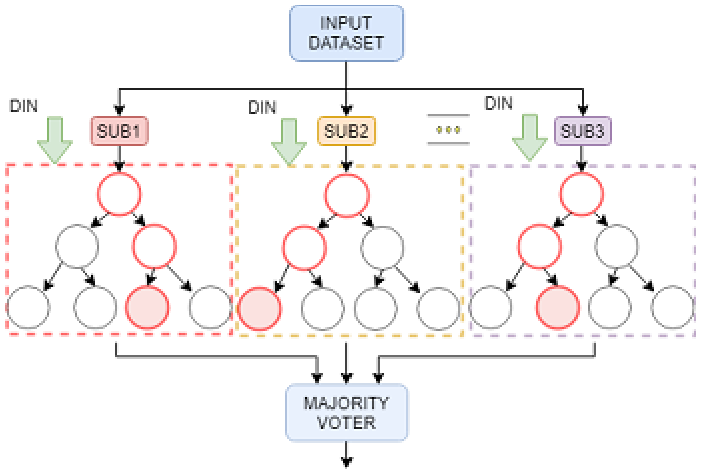


Figure 2.2: General structure of Random Forest Algorithm

2.2 Python Implementation

Machine Learning algorithms are described, developed and tested using a high level PYTHON language. This type of language is used because it is easy to write and modify, it has a rapid prototyping and it is able to implement many models. It has among its main objectives:

- Dynamism
- Simplicity
- Flexibility

It is possible to define an automatic learning algorithm and it is necessary to follow several steps:

1. Starting from the input data, a coefficient vector is generated.
2. Subsequently this vector is modulated obtaining the outputs.

3. It is possible to get the probability of the output value. For this it is applied the SOFTMAX operation.

4. To define the accuracy of the approximation we measure the error between the calculated value and the real value.

The Machine Learning then implements a predictive model as shown in figure 2.3:

- The first part is used for training that is made to minimize the error.

- The second part is used for the forecast.

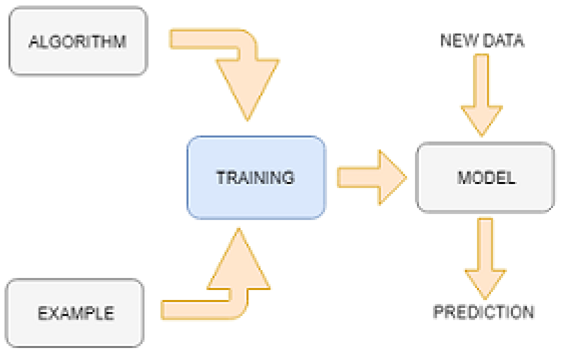


Figure 2.3: Predictive model Representation

In order to analyze a Machine Learning algorithm and to create an architecture that can support it is important to follow a series of steps as shown in figure 2.4:

1. Initially, the data is acquired, processed and grouped according to the ML algorithm used.

2. At this point we start with learning and it is considered a recursive algorithm divided into layers:

   - The first layer learns to predict using forecasts and by using Ensemble methods the algorithms are aggregated, capturing the linearity.
   - The second uses non-linear models and captures non-linearity.
   - The supervisor analyzes the data coming from the various layers and calculates the error making predictions on all the processes.

3. The next phase is the calibration phase, the cross-validation, where the error is compared.

    - If this error is greater than a certain threshold, the grouping done previously is repeated starting from all learning steps.
    - If instead the error is within the limits it is necessary to proceed with the development of the model where the forecasts going to evaluate a new input of data.

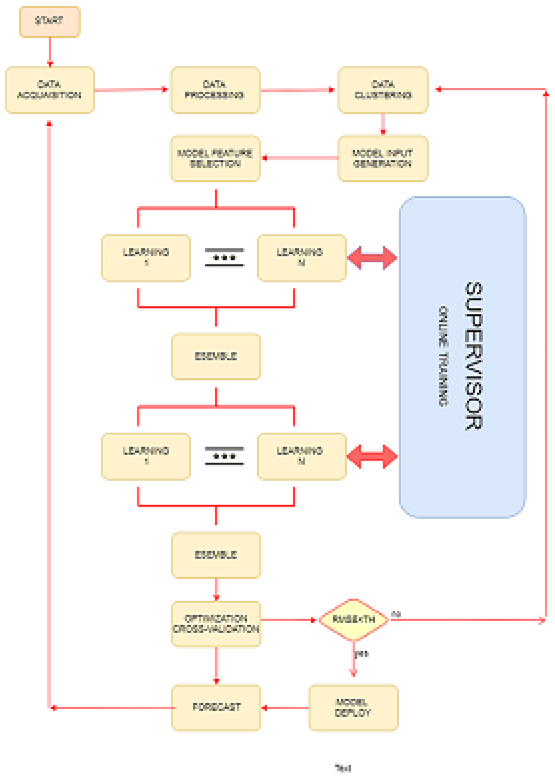


Figure 2.4: Flow of operations of a Machine Learning algorithm

Python is a general high level programming language.

python™

Its structure is very interesting for the development of different applications. Its simple syntax puts the accent on readability and also reduces the cost of the program which supports modules and packages emphasizing the modularity and the reuse of the code. SciPy is an ensemble of Python libraries for machine learning. It includes different modules used to machine learning:

- NumPy: it allows an efficient work with data in arrays
- Pandas: it allows to organize and analyze the data

- Matplotlib: it allows the creation of plots from data

In this thesis project, it's used the Scikit-learn library to develop and practice machine learning in Python. The main part of the library is machine learning algorithms for classification, regression and clustering. First of all, before implementing a machine learning algorithm is required to load the data. The most common format of the data for machine learning is CSV file where the values are separated using the comma (,) character.

There are a different ways to load a file in Python using:

- Python Standard Library
- NumPy
- Pandas

In this case, the file is loaded using Pandas through the *pandas.read_csv*().

There are different ways to evaluate the performance of an algorithm:

- Making predictions for new data of which they already know each other the answers.
- The use of resampling methods to make accurate estimates of the algorithm efficiency on new data.

For the problem of over-fitting the machine learning algorithm cannot use the same dataset for both prediction and the evaluation. In order to have high efficiency the classification must be corrected analyzing datas not used to train the algorithm. For this reason the dataset must be divided in two parts, one of training and the other one of testing. Then it is evaluated the predictions of the expected results. This method is ideal for all datasets and also when the algorithm is slow to train.

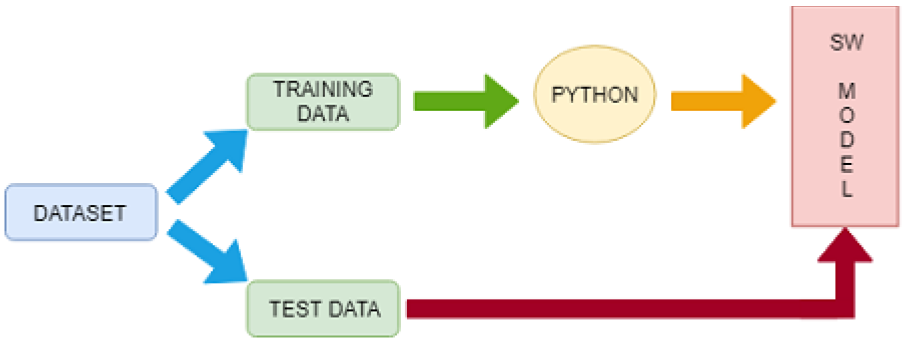


Figure 2.5: Flow of input

To evaluate predictions of model in the script, three parameters are used:

- Classification Accuracy
- Confusion Matrix
- Classification Report

In detail the *Classification Accuracy* is the number of correct predictions compared to all predictions made, the *Confusion Matrix* is a presentation of the accuracy that gives predictions on the x-axis and accuracy results on the y-axis. The *Classification Report* provides a quick idea of the accuracy of a model using a number of defined input. In Python, to improve the performances and the precision on dataset, it is useful to use Ensemble methods. In order to obtain the Ensemble methods it's necessary to combine the predictions. In fact to reach the goal there are different models:

- Bagging: builds multiple models using different sub-samples of the training dataset
- Boosting: builds multiple models where the single model learns to fix the prediction errors of the previous
- Voting: builds several different models by evaluating the prediction using a simple statistic

The implemented algorithm is the Random Forest that is an extension of bagged decision trees. Each tree is created to reduce the correlation of individual classifiers.

In python this type of algorithm can be made using the $RandomForestClassifier$ function.

2.3 Hardware Implementation

The hardware implementation of Random Forests consists of multiple decision tree blocks. Each decision tree is mapped into its own hardware block and it produces an output. The hardware implementation of each tree in the Random Forests is done using the Logic-in-Memory implementation for each base elements. In Random Forests, the same input data is given to all the different decision trees and multiple classification results are obtained. The outputs from all the different blocks are then routed into the majority voter block. According to the majority, the final classification result is produced and stored in the final output register which provides the final classification result. Because the Random Forest is composed of multiple decision trees and also because all their information are stored inside the memory, the Random Forest uses more memory intensive. After the creation of the algorithm in software, each tree is stored inside the memory. The overall structure that implements the Random Forest algorithm is shown in Figure 2.6. This is composed of several components:

- Register File

- Memory structure
- Majority voter
- Control Unit

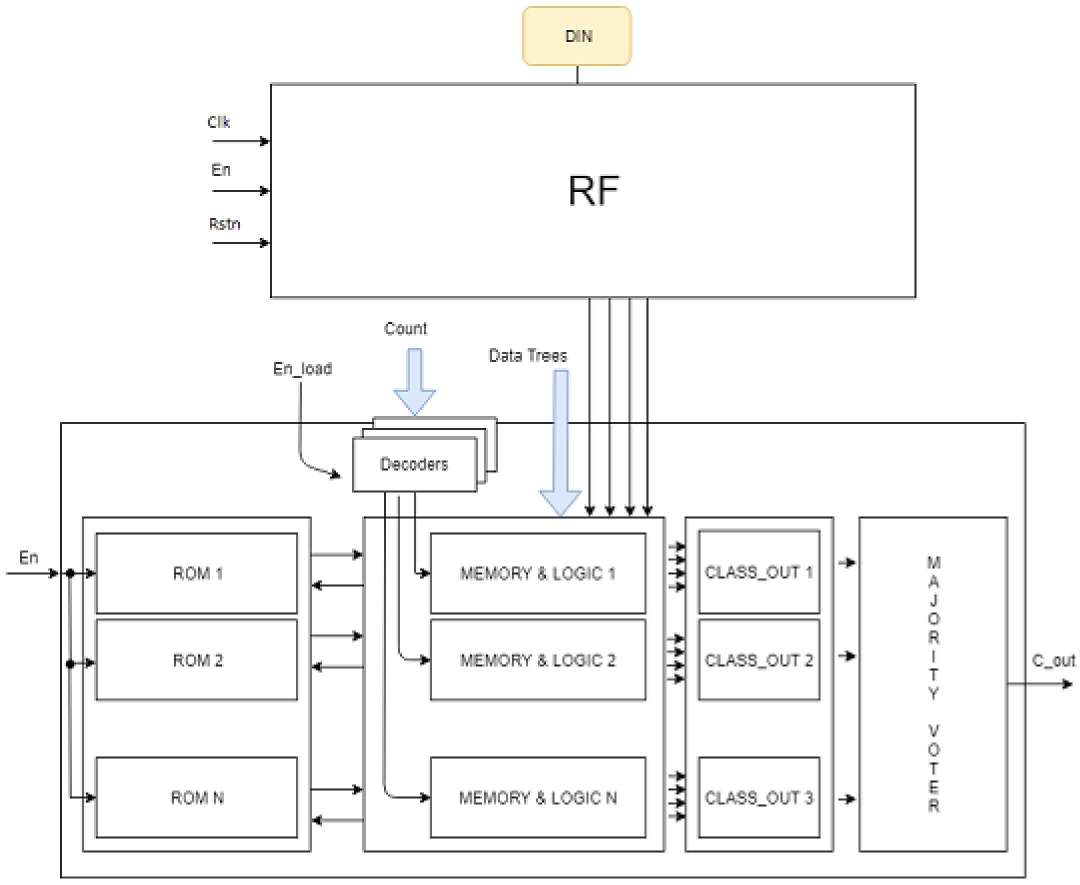


Figure 2.6: General Structure

2.3.1 Register File

According to the analysis carried out on the memory support it is reasonable to use a set of registers equal to the number of characteristics inside the dataset. The input data is inserted into a text file and converted according to the type of data. In this case, a dataset is used with numerical decimal data which are converted into integers by means of software multiplication. The register file shown in the figure

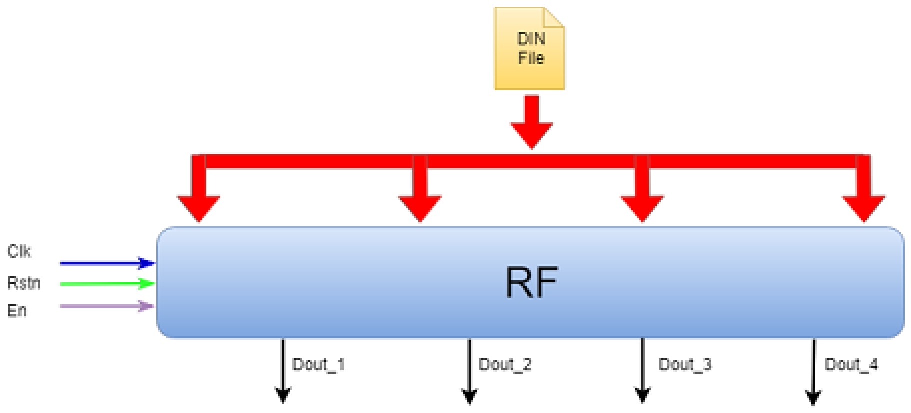


Figure 2.7: General view of Register File

2.7 provides a series of registers, as represented in figure 2.8, that are enabled by a write enable signal (*En*), by a Clock (*Clk*) and a Reset (*Rstn*) signal. The signal of *En* is driven by the filling of the memory. In fact, it is activated only when all the threshold values have been stored in the corresponding memory location.

The data is initially stored in the registers where, in absence of the enable signal, it remains without an output. When the threshold values of each decision node have been memorized, then the enable signal is enabled and from the Register File all datas are output at the same time.

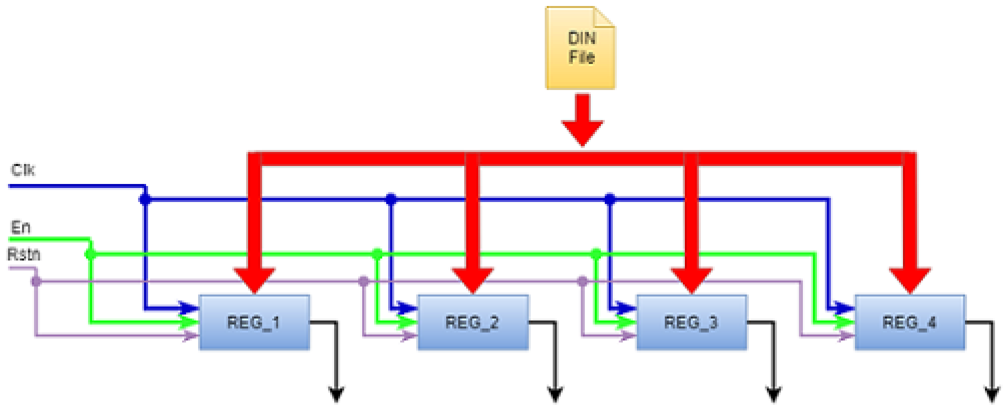


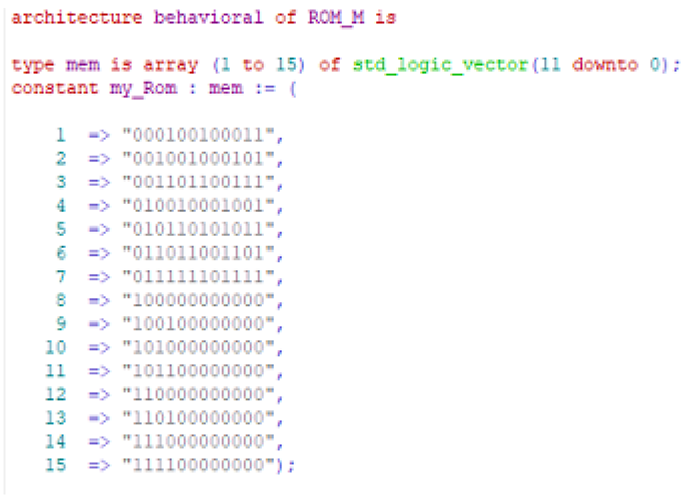
Figure 2.8: Details view of Register File

2.3.2 Memory Structure

The memory structure consists of two main parts: a constant, denominated ROM and a Logic-in-Memory. In fact, the memory doubles in a constant array in which the indexes of the nodes are stored, the current one, the right one and the left one and another containing the threshold values of each decision node and the part of logic that performs the comparison.

Constant Memory

The Rom is the memory in which the information relating to each node is stored as shown in the figure 2.9.



```vhdl
architecture behavioral of ROM_M is

type mem is array (1 to 15) of std_logic_vector(11 downto 0);
constant my_Rom : mem := (

     1  => "000100100011",
     2  => "001001000101",
     3  => "001101100111",
     4  => "010010001001",
     5  => "010110101011",
     6  => "011011001101",
     7  => "011111101111",
     8  => "100000000000",
     9  => "100100000000",
    10  => "101000000000",
    11  => "101100000000",
    12  => "110000000000",
    13  => "110100000000",
    14  => "111000000000",
    15  => "111100000000");
```

Figure 2.9: Constant Array Memory

Each row of the constant memory is divided as shown in figure 2.10 into:

- N bits that identify the current node
- N bits that identify the left node that corresponds to the true decision of the comparison
- N bits that identify the right node that corresponds to the false decision of the comparison.

In addition to the constant array, this part of architecture is composed by:

- Registers: one every two rows that stores the value of address of the following nodes.
- XNOR: it compares the output of the registers with the value of the current node of the row.
- Multiplexer: that choices the LSB of the following node.

In details, the constant memory is composed of $(N-1)/2$ registers used to identify the next node as a function of the comparison result.

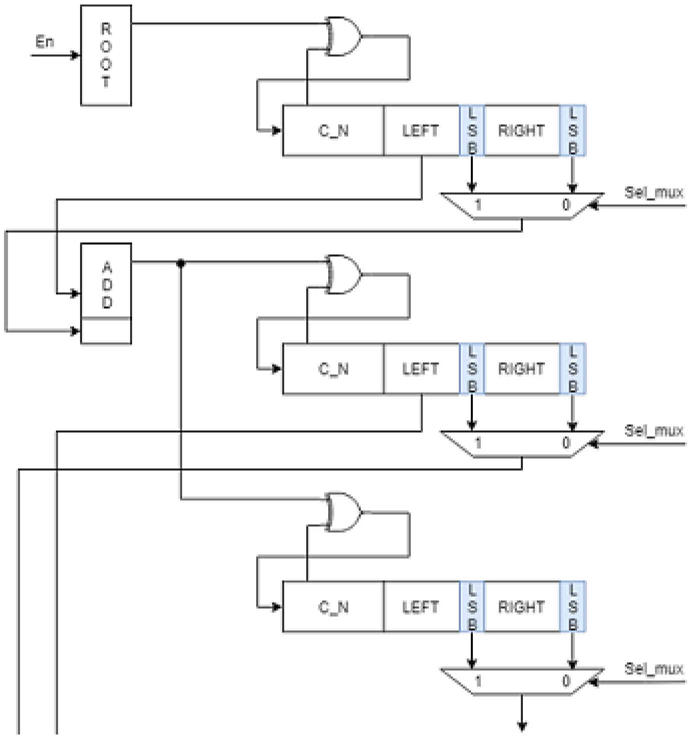


Figure 2.10: General view of Constant Memory

In fact this register consists of $(N-1)$ bits corresponding to the address of the next node minus the LSB. This value of the LSB is taken from a multiplexer inserted in memory to select LSB of the right or left node. Once the complete address has been obtained, a comparison is made with the current addresses of the two adjacent nodes through an XNOR port. If the addresses are the same, the corresponding node of the XNOR output port is equal to 1 and the row is activated and the relative comparison is enabled. If the values are different the output is 0 and its node is not enable. The comparison is enabled through a flag called *En_comp* that actives the logic memory row corresponding. This process continues until a classification is reached.

Logic Memory

After the implementation of Random Forest through Python, each tree is stored in a text file. This file is used to write in memory, for each node, the value of the selector, the relative threshold value and the corresponding class. Each decision tree node is implemented as a row of the memory array. The decision, ie the comparison between the stored value and the value to be analyzed is implemented by means of a logic block consisting of a series of logic gates. In detail, as shown in figure 2.11 a basic memory line is characterized by:

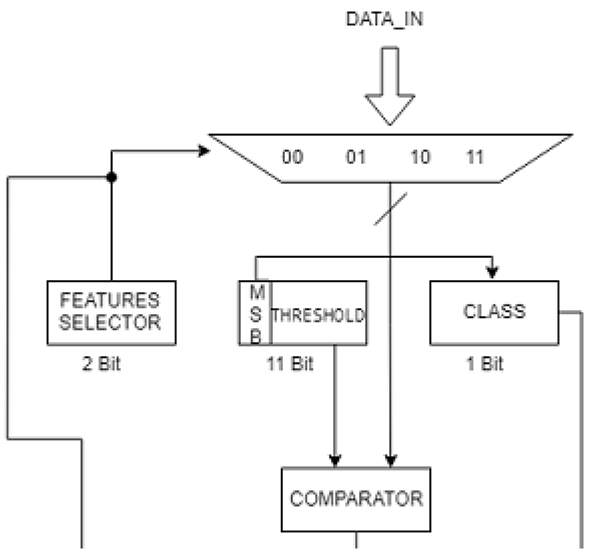


Figure 2.11: Row logic Memory

- $N-$bit register that stores the value of the selector which enables a multiplexer that selects the value corresponding to the characteristic.
- Multiplexer with number of inputs equal to the number of registers present in the Register File

- $1-$bit register in which the MSB which corresponds to the sign of the stored threshold value
- M $1-$bit registers in which the value of the threshold value is stored
- $N-$bit register that stores the value of the class
- Logic block that performs the comparison through a series of logic gates.

Now, it is analyzed the single components of the Logic-in-Memory row. First of all, the MSB block is composed as shown in figure 2.12 by:

- Register
- AND port
- NOR port

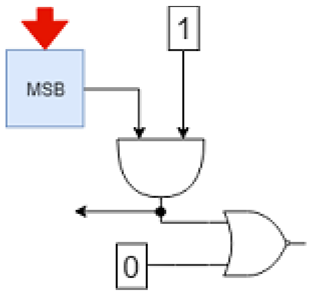


Figure 2.12: MSB logic block

The value stored in the register is equivalent to the value of the MSB of the threshold value. If this is 0, the number is positive and the comparison can be made. In this case the logic network output is positive and acts as an enable for the register containing the first bit to be compared. Instead if this bit is 1, the stored number is negative, which identifies that node is a leaf node. In this case the output of the logic network is negative and the comparison is not calculated because the class register is directly enabled.

The second block is make of:

- Register for the bit of tree threshold value

- Register for the bit of the input data
- Comparator block
- Register for the bit that represents the result of comparison.

The comparator element is composed by a logic network as shown in figure 2.13.

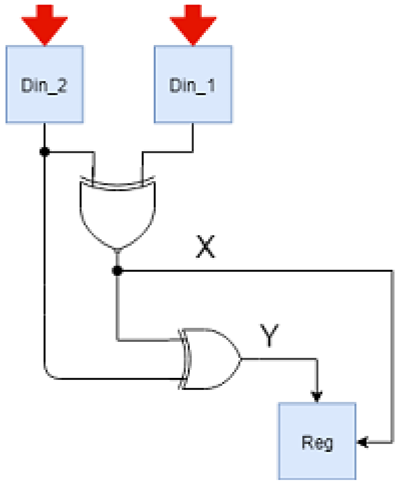


Figure 2.13: Comparator logic block

When the values are the same, the first output of the logic network, called *equal* signal, is equal to 1 and acts as enable for the following logic block. Instead if the values are different, the *equal* signal is 0 and the second output, called *less* signal, is equal to 1 or 0 if the Din_1 is less than Din_2 or if the Din_1 is higher than Din_2 respectively. This value is stored inside the register and its outputs is used to select the value 0 or 1 of the multiplexer inserted in the Rom memory block.

The total logic row structure is shown in the figure 2.14

Class Block

The Class block is the one responsible for checking if all the data relating to each decision node have been correctly memorized.

The Class block is composed in turn by:

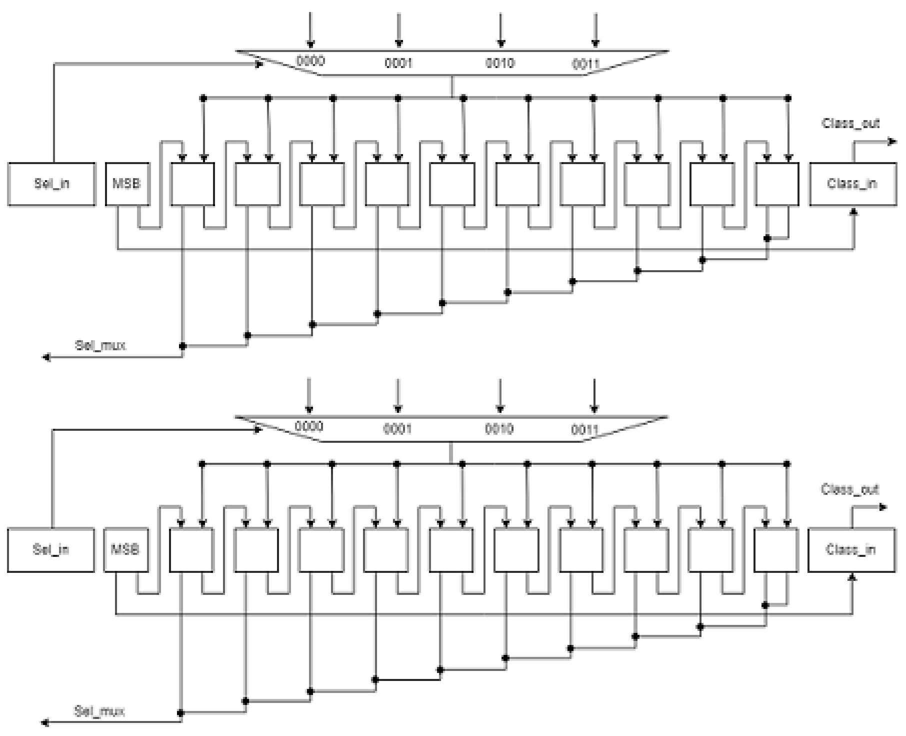


Figure 2.14: General view of Logic Memory

- **Store Flag block** used to check, that within all registers in a line are filled correctly by a flag. Each register enable a flag when it is filled. This block is characterized by a vector of N elements, where N is equal to the number of bits used to store the values of each node.

- **Store Load block** which checks all the nodes of the decision tree to be completely memorized. At the output of this block it is possible to have an enable signal that enables the root of the tree. It is composed by M vectors of N elements, where M represents the number of nodes that made up the tree.

- **Class out block** that has in input the classes of each decision node, which are enabled or not depending on the data to be analyzed. The output of this block therefore represents the class relative to the corresponding tree.

Each element of the vector corresponds to a flag that is enabled when the corresponding value has been stored in the register. When all the trees are correctly memorized in output to the Class Out, a flag is activated enabling the register file and consequently the start of the classification.

The global memory structure is shown in figure 2.15

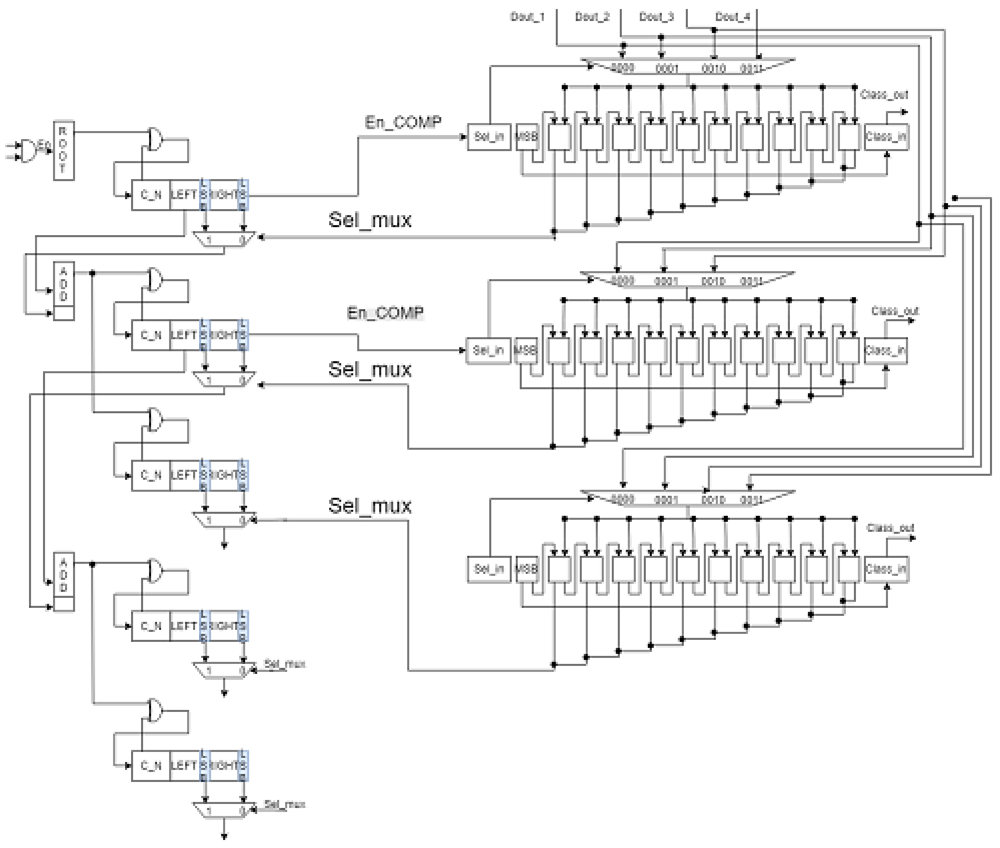


Figure 2.15: General view of Memory Structure

2.3.3 Majority Voter

The majority voter is a block outside the memory that performs an analysis on all the classes leaving each tree in the memory. Each tree structure has an output class that depends on the classification path. The majority voter receives N inputs, where N is the number of trees in the structure and its output is a single class that is the real classification result. It is composed by series of AND and OR ports. First of all, a inputs combination are the inputs of a chain of 2-input AND port, the result is the input of the OR port with the result of the second combination of all input and so on. The final output result is the class that present the largest number of times is highlighted as shown in the figure 2.16.

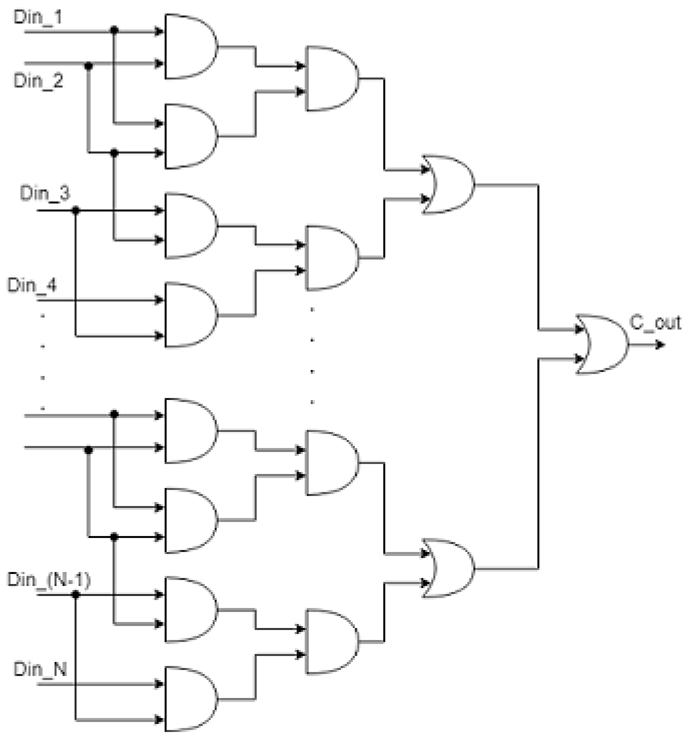


Figure 2.16: General view of Majority Voter

2.3.4 Control Unit Structure

Starting from the datapath, the flow of operations useful to achieve the correct classification and the relative FSM has been defined.

This FSM has the following states:

- **IDLE**: occurs when the machine is reset and remains in this state until the *S_start* signal arrives.
  If the *S_start* is asserted, it goes into the START state.

- **START**: the decoder enable signal is enabled and the data relating to the tree are loaded into memory. The machine remains in this state until the loading of all the nodes has been completed. When the data has been correctly loaded, the *class_store* signal is enabled, allowing the change to the LOAD status.

- **LOAD**: in this state the loading of the data to be analyzed by the Register file is enabled.
  The machine remains in this state a clock stroke.
  Subsequently, the status of CLASSIFIER is passed.

- **CLASSIFIER**: in this state the actual classification takes place. The data are taken from the Register file and analyzed using the trees. Subsequently the result of classes from each tree are processed by the majority voter and the final result is obtained.
  Subsequently, the LOAD state is returned until the data file to be analyzed ends and the *end_file* signal becomes 1.

The flow of the state machine is shown in the figure 2.17 and the internal signals for each state are also listed in the table 2.1

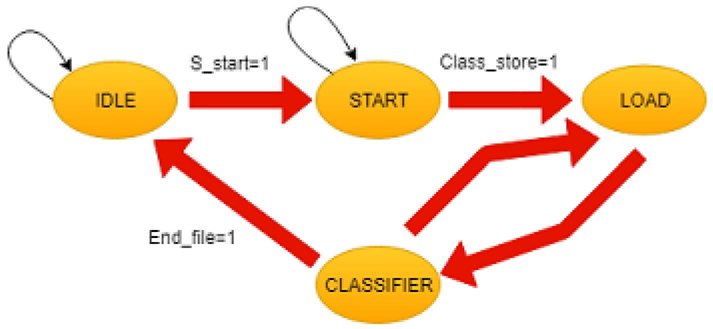


Figure 2.17: Data Flow Graph of FSM structure

The final overall architecture of Random Forest implementation is shown in the figure 2.18. This picture shows, not only different components of implementation,

	Start_1	*En*	*Rstn*	*En_dec*	*Sel*
IDLE	0	0	0	0	0
START	0	0	0	1	0
LOAD	1	1	1	0	0
CLASSIFIER	1	1	1	1	0

Table 2.1: FSM Signals

but also the external signals which are the inputs of the test-bench and the internal signals that link the different parts of architecture.

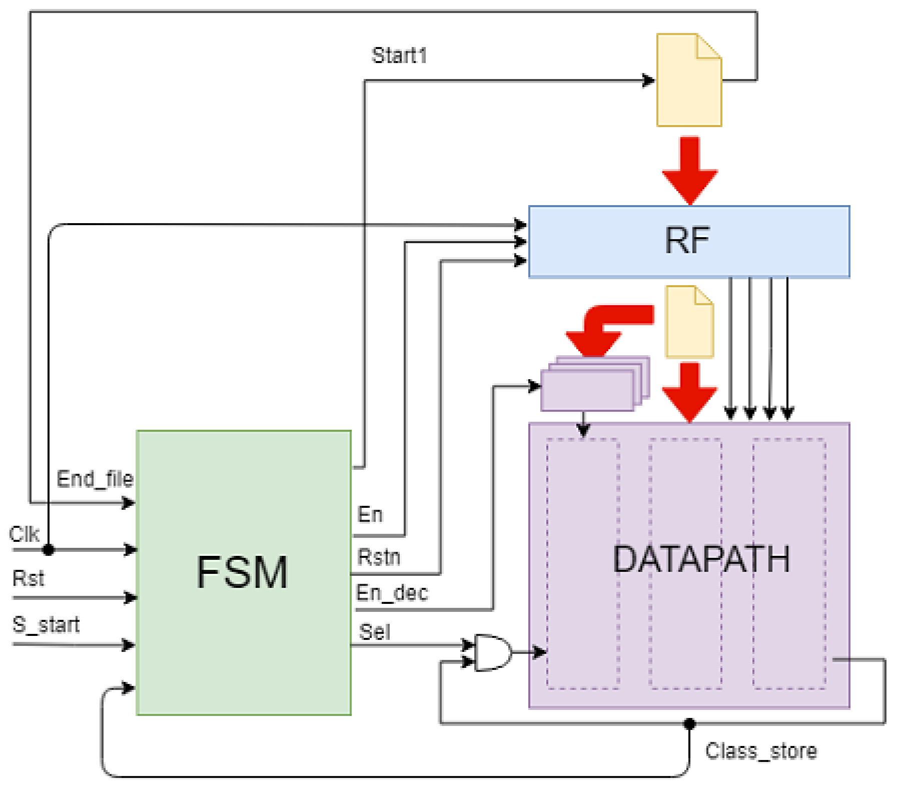


Figure 2.18: General Architecture of Random Forest Implementation

Chapter 3

Test and Simulation

The described system has been simulated with *Modelsim*. The results are obtained testing the architecture with the VHDL Test-bench. In order to have a better understanding of the behavior of process, in *Modelsim* simulation only main signals are shown:

- *Clk*
- *Rst*
- *Din_1*: the input data for classification.
- *Din_2*: the threshold value of decision node
- *Class_in*: the input class of each node
- *Sel_in*: the index of selection that enables the characteristic that must be analyzed
- *Count*: a flag that indicates the end of the input file
- *S_start*: a signal that indicates the starting point of classification
- *End_file*: a flag that indicates the end of the input file
- *C_out*: the output of Majority voter. It is the Final Classification value.

The simulation can be divided in different steps:

1. Initially, the dataset is divided into two parts by Python as shown in figure 3.1:
    - the part of training from which the trees are created. The trees are plotted and are inserted in the relative files.

- the part of the test useful for creating the file containing the data to be classified.

```
## Loading the dataset
dataset = pd.read_csv("C:/Users/asus/Desktop/Esempi_python/esempio/IRIS.csv")
X = dataset.drop('Class', axis=1)
y = dataset['Class']

## Spliting the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y , random_state = 1)
```

Figure 3.1: Python script to divide the dataset

2. Sequentially, the memory must be filled. The files are read line by line and are inserted in the appropriate position. This operation causes an overhead time as shown in figure 3.2.

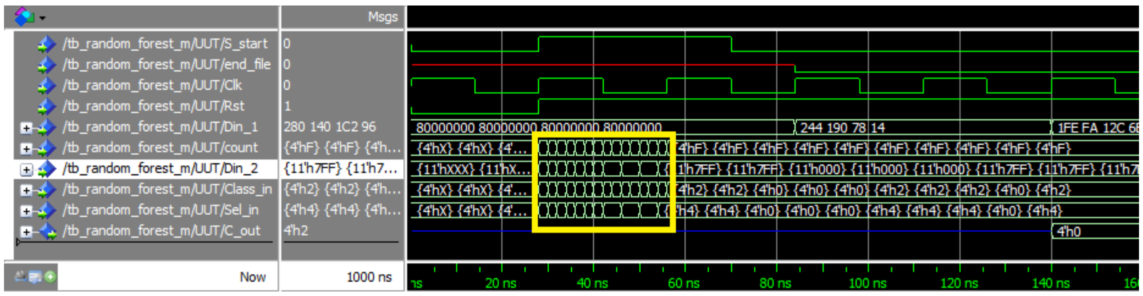

Msgs
/tb_random_forest_m/UUT/S_start 0
/tb_random_forest_m/UUT/end_file 0
/tb_random_forest_m/UUT/Clk 0
/tb_random_forest_m/UUT/Rst 1
/tb_random_forest_m/UUT/Din_1 280 140 1C2 96
/tb_random_forest_m/UUT/count {4'hF} {4'hF} {4'h...
/tb_random_forest_m/UUT/Din_2 {11'h7FF} {11'h7...
/tb_random_forest_m/UUT/Class_in {4'h2} {4'h2} {4'h...
/tb_random_forest_m/UUT/Sel_in {4'h4} {4'h4} {4'h...
/tb_random_forest_m/UUT/C_out 4'h2
80000000 80000000 80000000 80000000
244 190 78 14
1FE FA 12C 6
4'h0
Now 1000 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns

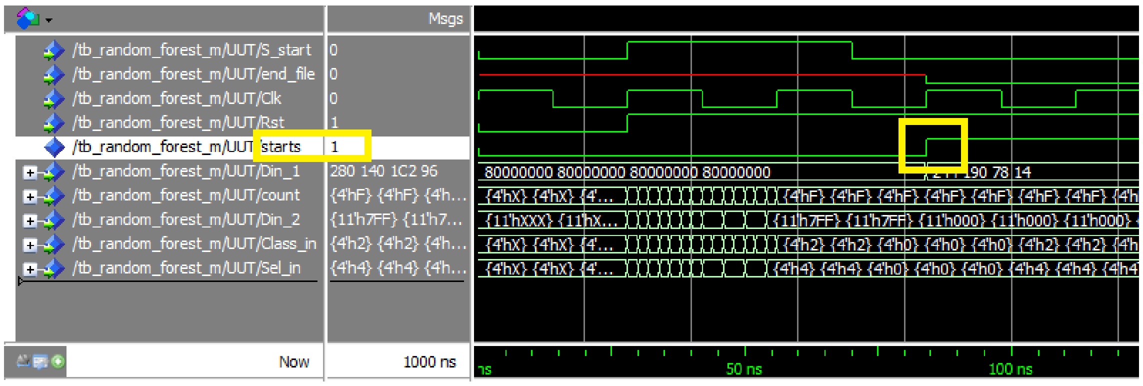
Figure 3.2: Overhead to load trees in memory

3. Once the trees have been stored, the *starts* signal is activated at the following rising edge and processing begins as shown in figure 3.3.

4. The data to be analyzed are stored in the *"data1.txt"* file. This file is the input of register file where each element is stored inside the apposite register as shown in figure 3.4. The register file outputs become the memory inputs.

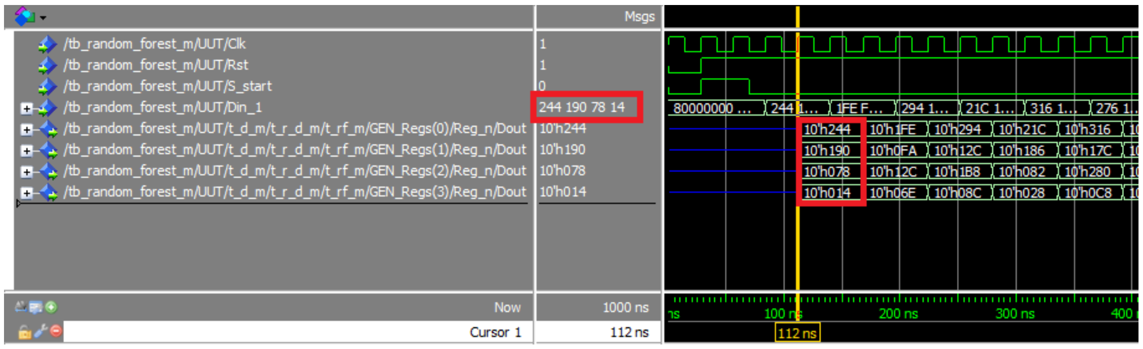
5. Each tree carries out its classification as shown in figure 3.5.

6. The classes leaving each tree are the inputs of the majority voter that performs the final classification as shown in figure 3.6



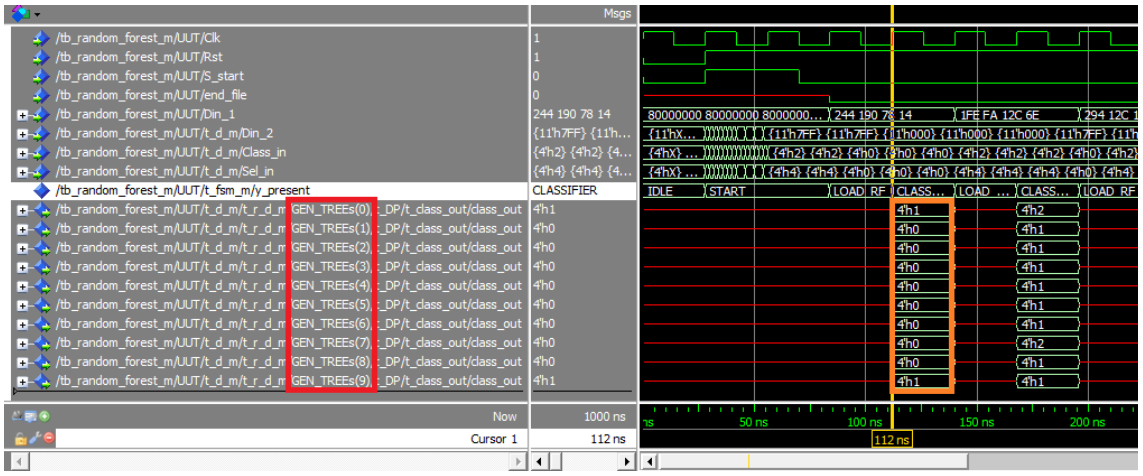
```
                                   Msgs
/tb_random_forest_m/UUT/S_start    0
/tb_random_forest_m/UUT/end_file   0
/tb_random_forest_m/UUT/Clk        0
/tb_random_forest_m/UUT/Rst        1
/tb_random_forest_m/UUT/starts     1
/tb_random_forest_m/UUT/Din_1      280 140 1C2 96
/tb_random_forest_m/UUT/count      {4'hF} {4'hF} {4'h...
/tb_random_forest_m/UUT/Din_2      {11'h7FF} {11'h7...
/tb_random_forest_m/UUT/Class_in   {4'h2} {4'h2} {4'h...
/tb_random_forest_m/UUT/Sel_in     {4'h4} {4'h4} {4'h...
Now                                1000 ns
```

Figure 3.3: Starting process simulation



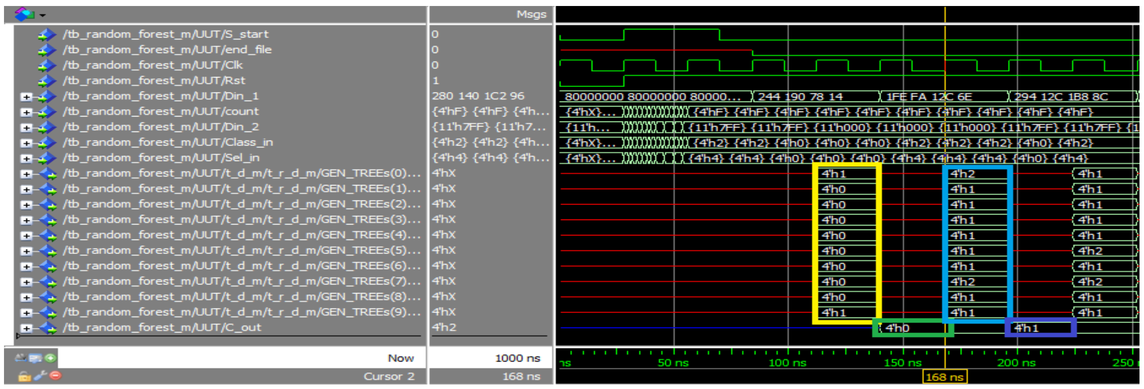
```
                                                                          Msgs
/tb_random_forest_m/UUT/Clk                                               1
/tb_random_forest_m/UUT/Rst                                               1
/tb_random_forest_m/UUT/S_start                                           0
/tb_random_forest_m/UUT/Din_1                                             244 190 78 14
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/t_rf_m/GEN_Regs(0)/Reg_n/Dout      10'h244
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/t_rf_m/GEN_Regs(1)/Reg_n/Dout      10'h190
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/t_rf_m/GEN_Regs(2)/Reg_n/Dout      10'h078
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/t_rf_m/GEN_Regs(3)/Reg_n/Dout      10'h014
Now                                                                       1000 ns
Cursor 1                                                                  112 ns
```

Figure 3.4: Loading of Register File



```
                                                                          Msgs
/tb_random_forest_m/UUT/Clk                                               1
/tb_random_forest_m/UUT/Rst                                               1
/tb_random_forest_m/UUT/S_start                                           0
/tb_random_forest_m/UUT/end_file                                          0
/tb_random_forest_m/UUT/Din_1                                             244 190 78 14
/tb_random_forest_m/UUT/t_d_m/Din_2                                       {11'h7FF} {11'h...
/tb_random_forest_m/UUT/t_d_m/Class_in                                    {4'h2} {4'h2} {4...
/tb_random_forest_m/UUT/t_d_m/Sel_in                                      {4'h4} {4'h4} {4...
/tb_random_forest_m/UUT/t_fsm_m/y_present                                 CLASSIFIER
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(0) DP/t_class_out/class_out  4'h1
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(1) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(2) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(3) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(4) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(5) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(6) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(7) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(8) DP/t_class_out/class_out  4'h0
/tb_random_forest_m/UUT/t_d_m/t_r_d_m GEN_TREEs(9) DP/t_class_out/class_out  4'h1
Now                                                                       1000 ns
Cursor 1                                                                  112 ns
```

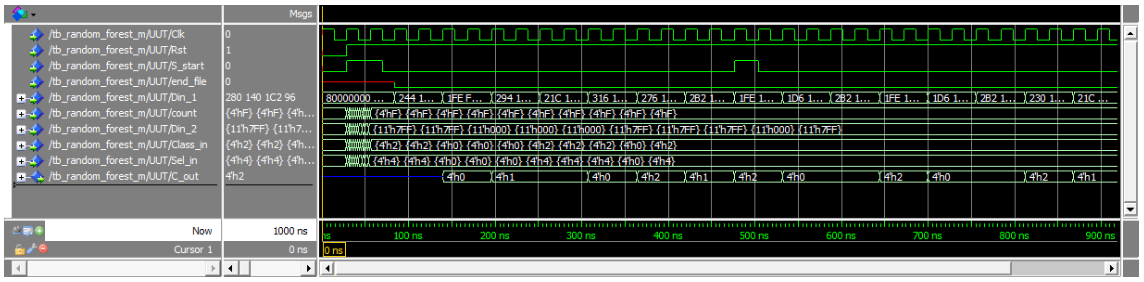
Figure 3.5: Tree Classifications



```
                                                              Msgs
/tb_random_forest_m/UUT/S_start                               0
/tb_random_forest_m/UUT/end_file                              0
/tb_random_forest_m/UUT/Clk                                   0
/tb_random_forest_m/UUT/Rst                                   1
/tb_random_forest_m/UUT/Din_1                                 280 140 1C2 96
/tb_random_forest_m/UUT/count                                 {4'hF} {4'hF} {4'h...
/tb_random_forest_m/UUT/Din_2                                 {11'h7FF} {11'h7...
/tb_random_forest_m/UUT/Class_in                              {4'h2} {4'h2} {4'h...
/tb_random_forest_m/UUT/Sel_in                                {4'h4} {4'h4} {4'h...
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(0)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(1)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(2)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(3)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(4)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(5)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(6)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(7)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(8)...         4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(9)...         4'hX
/tb_random_forest_m/UUT/C_out                                 4'h2

                                                      Now     1000 ns
                                                 Cursor 2     168 ns
```

Figure 3.6: Maority voter result



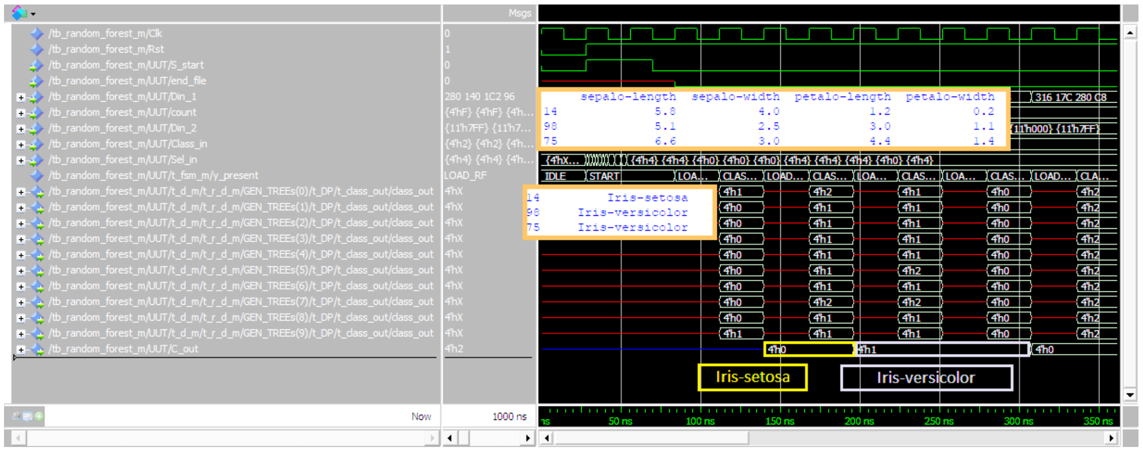
```
                                                Msgs
/tb_random_forest_m/UUT/Clk                     0
/tb_random_forest_m/UUT/Rst                     1
/tb_random_forest_m/UUT/S_start                 0
/tb_random_forest_m/UUT/end_file                0
/tb_random_forest_m/UUT/Din_1                   280 140 1C2 96
/tb_random_forest_m/UUT/count                   {4'hF} {4'hF} {4'h...
/tb_random_forest_m/UUT/Din_2                   {11'h7FF} {11'h7...
/tb_random_forest_m/UUT/Class_in                {4'h2} {4'h2} {4'h...
/tb_random_forest_m/UUT/Sel_in                  {4'h4} {4'h4} {4'h...
/tb_random_forest_m/UUT/C_out                   4'h2

                                        Now     1000 ns
                                   Cursor 1     0 ns
```

Figure 3.7: Final Simulation results

After all steps, the overall simulation shows the output of classification in the figure 3.7. Finally the results produced by VHDL are comparing with the ones obtained with Python program. It is possible to see that the results are the same. The comparison between output of classification and the Python results are shown in the figure 3.8.

The final result of classification of the Hardware implementation are inserted inside a file called *"out.txt"*. Subsequently, this file is compared with the output file that is generated with the Software implementation through Python script as shown in figure 3.9 The result shows the Classification Error and the Classification Accuracy in both of them implementations, Software and Hardware. This script results are shown in figure 3.10.

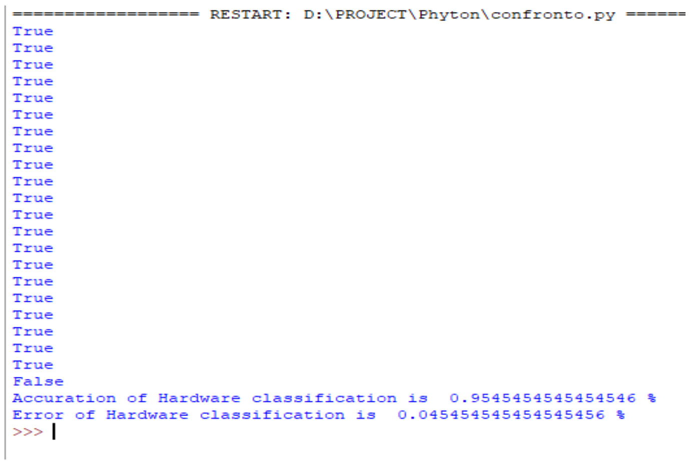


```
/tb_random_forest_m/Clk                                                   0
/tb_random_forest_m/Rst                                                   1
/tb_random_forest_m/UUT/S_start                                           0
/tb_random_forest_m/UUT/end_file                                          0
/tb_random_forest_m/UUT/Din_1                                             280 140 1C2 96
/tb_random_forest_m/UUT/count                                             {4'hF} {4'hF} {4'h...
/tb_random_forest_m/UUT/Din_2                                             {11'h7FF} {11'h7...
/tb_random_forest_m/UUT/Class_in                                          {4'h2} {4'h2} {4'h...
/tb_random_forest_m/UUT/Sel_in                                            {4'h4} {4'h4} {4'h...
/tb_random_forest_m/UUT/t_fsm_m/y_present                                 LOAD_RF
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(0)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(1)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(2)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(3)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(4)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(5)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(6)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(7)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(8)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/t_d_m/t_r_d_m/GEN_TREEs(9)/t_DP/t_class_out/class_out   4'hX
/tb_random_forest_m/UUT/C_out                                             4'h2
Now                                                                       1000 ns
```

```
    sepalo-length  sepalo-width  petalo-length  petalo-width
14            5.8           4.0            1.2           0.2
98            5.1           2.5            3.0           1.1
75            6.6           3.0            4.4           1.4
```

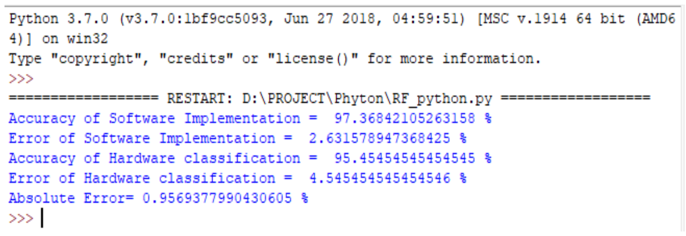
```
14         Iris-setosa
98     Iris-versicolor
75     Iris-versicolor
```

Figure 3.8: Comparison with Software and Hardware Classification



```
=================== RESTART: D:\PROJECT\Phyton\confronto.py ======
True
True
True
True
True
True
True
True
True
True
True
True
True
True
True
True
True
True
True
True
True
False
Accuration of Hardware classification is  0.9545454545454546 %
Error of Hardware classification is  0.045454545454545456 %
>>> |
```

Figure 3.9: Result of comparison between Hardware and Software implementation



```
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 64 bit (AMD6
4)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
================== RESTART: D:\PROJECT\Phyton\RF_python.py ==================
Accuracy of Software Implementation =  97.36842105263158 %
Error of Software Implementation =  2.631578947368425 %
Accuracy of Hardware classification =  95.45454545454545 %
Error of Hardware classification =  4.545454545454546 %
Absolute Error= 0.9569377990430605 %
>>> |
```

Figure 3.10: Accuracy and Error of Hardware and Software implementation

Chapter 4

Synthesis

The architecture has been synthesized with *Synopsis Design Compiler* in order to find maximum clock frequency that design achieves and the corresponding area.

The logical synthesis process can be divided into different steps:

- Define a library and read VHDL source files
- Define and apply constraints
- Start the synthesis of the design
- Save the results of the report

The operations that must be executed are shown in figure 4.1

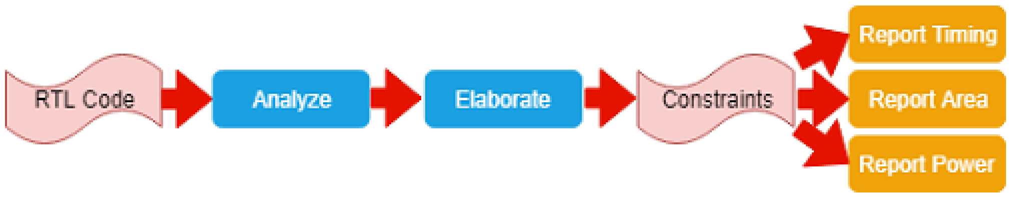


Figure 4.1: General Flow of Synthesis operations

First of all, the first command is *analyze*. With this command the architecture is checked within all components inside it. In the specific case, the general architecture is the Random Forest implementation that is composed by different components as:

- N Decision tree
- Register File

- Majority voter

Each of them, is consequently the architecture is composed by other components and so on. The second operation is *elaborate* command which requires the parameters analyzed in the top entity of the synthesized architecture. In this step the compiler maps the VHLD file in a series of logic gates. The following step is to apply the constraints that depend on the considered top entity. In this synthesis ii is possible to insert a clock and load constraints:

- *create_clock -name CLK -period 11.175 CLK*, to create a clock signal with a certain period.
- *set_dont_touch_network CLK*, to not change the value of clock signal.
- *set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]*
- *set_load $OLOAD [all_outputs]* to set the load of each output design.

After the constraints generation, the architecture can be compiled using the *compile* command. Now the design is synthesized and the results can be stored inside the text file through the report command.

4.1 Synthesis of Hardware implementation

First of all, to synthesize this architecture it is used a $45nm$ technology, starting from synthesized structure without constraints. In the following table 4.1 are inserted the results of timing, power and area of the different components into the global architecture of Random Forest implementation.

The first four synthesized components are composed by only combination logic. Of course, inserting constraints these reports don't change. The last components and the global architecture, change with the constraints. First of all, inserting a clock of 0, it is defined the maximum timing delay through the negative SLACK. After, an optimal result of timing delay is obtained replacing the synthesis with a new clock. The results are reported in the following table 4.2.

	AREA $[\mu m^2]$	**POWER** $[\mu W]$	**TIMING** $[ns]$
Comparator	7.448	3.203	0.14
Logic Cell	15.96	31.29	0.38
Row Logic	269.1920	45.3780	3.77
Tree	4227.5380	367.1344	13.07
Memory Structure	4306.008	366.9151	13.38
Datapath	43854.6226	$1.9352e^{+03}$	13.55
Random Forest	43875.6366	$1.0709e^{+03}$	13.68

Table 4.1: 45nm Synthesis Results without constraint

	AREA $[\mu m^2]$	**POWER** $[\mu W]$	**TIMING** $[ns]$	**CLK** $[ns]$
Datapath	44273.3066	$1.055e^{+03}$	10.74	10.79
Random Forest	45955.4906	$1.6627e^{+03}$	11.13	11.175

Table 4.2: 45nm Synthesis Results with clock and load constraints

Subsequently, the architecture is synthesized using a 28nm technology and the results without constraint are shown in the table 4.3 and the results of Random Forest implementation using constraint is shown in the table 4.4

	AREA $[\mu m^2]$	**POWER** $[\mu W]$	**TIMING** $[ns]$
Tree	1589.894382	$7.4209e^{-02}$	1.81
Memory Structure	1641.955182	$7.5203e^{-02}$	2.03
Random Forest	21327.628691	0.2636	2.37

Table 4.3: 28nm Synthesis Results without constraints

4.2 Comparison with other Hardware implementation

The architecture is synthesized and it is compared with two different architectures present in the State of art. These two architecture implement a single tree of oblique type and an axis parallel tree respectively. In order to compare these architectures with the proposed hardware implementation, a single tree is synthesized. The results of area, power and timing are shown in the table 4.5. In details, in the case of

	AREA $[\mu m^2]$	**POWER** $[\mu W]$	**TIMING** $[ns]$	**CLK** $[ns]$
Random Forest	16986.182236	$7.4775e^{-02}$	1.96	2.0

Table 4.4: $28nm$ Synthesis Results with clock and load constraints

	AREA $[\mu m^2]$	**POWER**	**TIMING**
DT oblique	$100{\cdot}10^3$	$25\ mW$	$10\ ns$
DT axis parallel	$8.9\ {\cdot}10^3$	$3.1\ mW$	$10\ ns$
LIM implementation of TREE	5301.38	$175.54\ \mu W$	$9.80\ ns$

Table 4.5: Comparison Results

implemented hardware architecture, the frequency is higher than both solutions considered. In fact, in this case the frequency is equal about $102MHz$, and moreover, improvements are also achieved in area and power.

Chapter 5

Conclusions and future works

This thesis work has been carried out in detailed way the hardware implementation of machine learning algorithm in order to perform classification operations. The implementation is developed creating a Logic-in-memory architecture.
First of all, the elaboration phase of datas is implemented in software and subsequently the classifier is created in hardware.
The obtained results about both implementations are compared to each other. The implemented model is called Random Forest which is an accurate classifier with high percentage of accuracy equal to 97% in Software and 95.4% in Hardware meanwhile the error percentages are computed in both implementations around 3% and 4.6% respectively.
The device speed reaches a value equal to $89.5MHz$ for $45nm$ technology and $500MHz$ for $28nm$ technology. Moreover, considering the single tree in the full implemented hardware architecture, it is possible to obtain the frequency about 102 MHz, higher than two decision tree hardware implementation analyzed in state of art, obtaining improvements are also achieved in area and power At the end, it is possible to highlight that using a Logic-in-memory implementation, the computational and elaboration speed increases. In this case the absolute error percentage of both implementations is less than 1%.
The future developments can be grouped in two possibilities:

- The first one is about the training part of model, not only the classification. It is possible to insert a co-processor used to implement the operations of creation and training of the algorithm.
- The second solution is about the insertion of pipe stages to improve the elaboration speed and to optimize furthermore the data management.

Appendix A

Python codes

A.1 Random Forest Script

```python
## Random forest classifier  - iris dataset
import pandas as pd
import numpy as np
import timeit
import matplotlib.pyplot as plt
import os
import sys
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

##Import tools needed for visualization
from sklearn.tree import export_graphviz
from sklearn.metrics import classification_report,confusion_matrix
import pydot
import csv

## Loading the dataset
dataset = pd.read_csv("D:/PROJECT/Phyton/IRIS.csv")
X = dataset.drop('Class', axis=1)
y = dataset['Class']


## Spliting the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y , random_state = 1)

## Building the model
clf = RandomForestClassifier(max_depth=3, n_estimators=10)
clf.fit(X_train, y_train)

## Pull out one tree from the forest
tree = clf.estimators_[0]
tree1 = clf.estimators_[1]
tree2 = clf.estimators_[2]
tree3 = clf.estimators_[3]
tree4 = clf.estimators_[4]
```

```
tree5 = clf.estimators_[5]
tree6 = clf.estimators_[6]
tree7 = clf.estimators_[7]
tree8 = clf.estimators_[8]
tree9 = clf.estimators_[9]

## Export the image to a txt file
## Precision indicates the maximum of precision bit
export_graphviz(tree, out_file = 'tree0_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree1, out_file = 'tree1_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree2, out_file = 'tree2_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree3, out_file = 'tree3_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree4, out_file = 'tree4_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree5, out_file = 'tree5_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
```

```
export_graphviz(tree6, out_file = 'tree6_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree7, out_file = 'tree7_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree8, out_file = 'tree8_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)
export_graphviz(tree9, out_file = 'tree9_p.txt', max_depth=3,
      feature_names=['sepalo-length','sepalo-width','petalo-length','petalo-width'],
                 class_names=['Iris-setosa','Iris-versicolor','Iris-virginica'],
                 label='all', filled=True, leaves_parallel=False, impurity=True,
                       node_ids=True ,
                 proportion=False, rotate=False, rounded=True,
                       special_characters=False, precision=2)

## Use dot file to create a graph
(graph, ) = pydot.graph_from_dot_file('tree0_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree1_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree2_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree3_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree4_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree5_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree6_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree7_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree8_p.txt')
(graph, ) = pydot.graph_from_dot_file('tree9_p.txt')

## Predicting the Species
predicted = clf.predict(X_test)
print('Accuracy of Software Implementation = ',100*accuracy_score(predicted,
      y_test),"%")
print('Error of Software Implementation = ',(100 - 100*accuracy_score(predicted,
      y_test)),"%")

#open a new file to store the test values
out_file_test=open("file_test.txt","w")
```

```
out_file_test.write(str(X_test))
out_file_test.close()

out_file_test1 = open("file_test.txt","r")
out_file_test2=open("file_test1.csv","w")
for line in out_file_test1:
    out_file_test2.write(line[15:18]+","+line[29:32]+","+line[44:47]+","+line[58:63])
out_file_test1.close()
out_file_test2.close()


with open("file_test1.csv", "r") as infile:
    lines = infile.readlines()

with open("file_test1.csv", "w") as outfile:
    for pos, line in enumerate(lines):
        if pos != 0:
            outfile.write(line)
out_file = open("file2.txt","w")
out_file.write(str(y_test))
##    out_file.write("\n")
out_file.close()
##
with open("./file_test1.csv") as filecsv:
    lettore=csv.reader(filecsv,delimiter=",")
    dati=[(riga[0],riga[1],riga[2],riga[3]) for riga in lettore]
    dati1 = [list(riga) for riga in dati]

#creo un file dati1.txt in scrittura
    out_file6 = open("dati1.txt","w")
    out_file6 = open("dati1.txt","a")

#riempio il file andando ad eliminare le parentesi e la punteggiatura
    for i in range(len(dati1)):
        c = dati1[i]
        #print(c)
        f=float(c[0])*100
        f1=float(c[1])*100
        f2=float(c[2])*100
        f3=float(c[3])*100
        out_file6.write(str(int(f))+" " +str(int(f1))+" " +str(int(f2))+" "
              +str(int(f3)))
        out_file6.write("\n")
    out_file6.close()

out_file = open("file2.txt","r")
outfile1 = open('output.txt', 'w')

for line in out_file:
    if(line[11:22]=='Iris-setosa'):
        outfile1.write(line.replace('Iris-setosa', '0'))
    if(line[7:22]=='Iris-versicolor'):
        outfile1.write(line.replace('Iris-versicolor', '1'))
    if(line[8:22]=='Iris-virginica'):
        outfile1.write(line.replace('Iris-virginica', '2'))
out_file.close()
outfile1.close()
outfile1 = open('output.txt', 'r')
outfile2 = open('output_final.txt', 'w')
for line in outfile1:
    if(line[11:12]=='0'):
        outfile2.write(line[11:22])
    if(line[7:8]=='1'):
        outfile2.write(line[7:22])
    if(line[8:9]=='2'):
        outfile2.write(line[8:22])
outfile1.close()
outfile2.close()
```

A.2 Input file Script

```
## This file reads the dataset and it creates an input file of RF
## hardware implementation.
## All datas are inserted inside the file that is called "dati1.txt"

import csv

## The script reads the database and it analizes only the row that respects
## the requested conditions
with open("./IRIS.csv", newline="") as filecsv:
    lettore=csv.reader(filecsv,delimiter=",")
    dati=[(riga[0],riga[1],riga[2],riga[3]) for riga in lettore]
    dati1 = [list(riga) for riga in dati]

## The script creates the file
    out_file = open("dati1.txt","w")
    out_file = open("dati1.txt","a")

## The value are multiplied for a x value and they are stored inside the new file
    for i in range(11):
        c = dati1[i]
        f=float(c[0])*100
        f1=float(c[1])*100
        f2=float(c[2])*100
        f3=float(c[3])*100
        out_file.write(str(int(f))+" " +str(int(f1))+" " +str(int(f2))+" "
              +str(int(f3)))
        out_file.write("\n")
    out_file.close()
```

A.3 Comparison Script

```python
## This script implements the comparison between two different file.
## The first file is the output of the software implementation.
## The second file is the output of the hardware implementation.

filename1 = "D:/PROJECT/Phyton/output_final.txt"
filename2 = "D:/PROJECT/Phyton/out.txt"
a=0
b=0
c=0
d=0
## Open file
with open(filename1) as f1:
    with open(filename2) as f2:
        file1list = f1.read().splitlines()
        file2list = f2.read().splitlines()
        list2length = len(file2list)
        list1length = len(file1list)
        for index in range(list2length):
            print(file1list[index] == file2list[index])
            if(file1list[index] == file2list[index]):
                c=c+1
            if(file1list[index] != file2list[index]):
                a=a+1
        b=a/list2length
        d=c/list2length
        print("Accuration of Hardware classification is ", d ,"%")
        print("Error of Hardware classification is ", b,"%")
```

Bibliography

[1] James G, Witten D, Hastie T, Tibshirani R, *"An Introduction to Statistical Learning"*, Springer, New York, 2013.

[2] J.H. Friedman, T. Hastie, R. Tibshirani, *"The Elements of Statistical Learning: Data Mining, Inference and Prediction"*, SpringerVerlag, Heidelberg, 2001.

[3] Muhammad Shafiq, Xiangzhan Yu, Asif Ali Laghari, Lu Yao, Nabin Kumar Karn, Foudil Abdessamia, *"Network Traffic Classification techniques and comparative analysis using Machine Learning algorithms"*, 2nd IEEE International Conference on Computer and Communications (ICCC), Oct. 2016

[4] Huang Z, *"A fast clustering algorithm to cluster very large categorical data sets in data mining"*, Res Issues Data Min Knowl Discov, 1997.

[5] M. Kang, S. K. Gonugondla, and N. R. Shanbhag, *"A 19.4 nJ/decision 364K decisions/s in-memory random forest classifier in 6T SRAM array"*, in Proc. IEEE Eur. Solid-State Circuits Conf. (ESSCIRC),Sep. 2017, pp. 263-266.

[6] M. Kang, S. K. Gonugondla, and N. R. Shanbhag, *"A 19.4-nJ/Decision, 364-K Decisions/s, In-Memory Random Forest Multi-Class Inference Accelerator"*, in Proc. IEEE Eur. Solid-State Circuits Conf. (ESSCIRC),Sep. 2017, pp. 263-266.

[7] Bernard S, Adam S, Heutte L (2007) *"Using random forests for handwritten digit recognition. Ninth International Conference on Document Analysis and Recognition"* (IEEE Computer Society, Los Alamitos, CA), pp 1043-1047.

[8] Nguyen C., Wang Y., Nguyen H.N., *Random forest classifier combined with feature selection for breast cancer diagnosis and prognostic.*, J. Biomed. Sci. Eng. 2013, 6, 551-560.

[9] T. Chen et al.*"DianNao: A Small-footprint High-throughput Accelerator For Ubiquitous Machine-learning"*, ASPLOS, 2014.

[10] D. Liu et al., *"Pudiannao: A polyvalent machine learning accelerator"*, in Proc. ASPLOS, 2015.

[11] Y.-H. Chen et al., *"Eyeriss: A Spatial Architecture For Energy-Efficient Dataflow For Convolutional Neural Networks"*, ISCA, 2016.

[12] Minkwan Kee, Seung-jin Lee; Hyun-su Seon; Jongsung Lee; Gi-Ho Park, *"Intelligence Boosting Engine (IBE): A hardware accelerator for processing sensor fusion and machine learning algorithm for a sensor hub SoC"*, in IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS), 2017

[13] Y. Chen et al., *"DaDianNao: A machine-learning supercomputer"*, in Proc. MICRO, 2014.

[14] A. Shafiee et al., *"ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic In Crossbars"*, ISCA, 2016.,

[15] M. Samragh et al., *"Looknn: Neural network with no multiplication"*, in IEEE/ACM DATE, 2017.

[16] Jintao Zhang, Zhuo Wang, Naveen Verma, *"In-Memory Computation of a Machine-Learning Classifier in a Standard 6T SRAM Array"*, in IEEE Journal of Solid-State Circuits, 2017

[17] Jintao Zhang, Zhuo Wang, Naveen Verma, *"A machine-learning classifier implemented in a standard 6T SRAM array"*, in IEEE Symposium on VLSI Circuits (VLSI-Circuits), 2016

[18] P. Chi et al., *"PRIME: A Novel Processing-in-Memory Architecture For Neural Network Computation In ReRAM-Based Main Memory"*, ISCA, 2016.

[19] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, and et. al, *"Tensorflow: A system for large-scale machine learning"*, in 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016, pp. 265-283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, and et. al, *"Tensorflow: Large-scale machine learning on heterogeneous distributed systems"*, 2015. [Online]. Available: http://download. tensorflow.org/paper/whitepaper2015.pdf

[21] S. Liu et al., *"Cambricon: An instruction set architecture for neural networks"*, in Proc. ISCA, 2016.

[22] Kwihoon Kim, Yong-Geun Hong- Youn-Hee Han, *"General labelled data generator framework for network machine learning"*, IEEE 20th International Conference on Advanced Communication Technology (ICACT)Feb. 2018.

[23] Q. Li and A. Berma, *"A Low-Power Hardware-Friendly Binary Decision Tree*

Classifier for Gas Identification", J. Low Power Electron. Appl., vol. 1, pp. 45-58, 2011

[24] Bermak, A.; Martinez, D. *"A Compact 3D VLSI Classifier using Threshold Network Ensembles"* IEEE Trans. Neural Networks 2003, 14, 1097-1109.