Anomaly detection via low-power on chip machine learning

Pouya Houshmand
Matr. 252814

Thesis submitted for the degree of
Master of Science in
Electrical Engineering, option
Electronics and Integrated Circuits

Thesis supervisor:
Prof. dr. ir. Maurizio Zamboni,
Prof. dr. ir. Marian Verhelst

Assessor:
Ir. Jaro De Roose

Mentor:
Ir. Jaro De Roose

Academic year 2018 – 2019

© Copyright Politecnico di Torino

La tesi di laurea, in quanto dotata di carattere creativo, costituisce opera dell'ingegno tutelata dalla legge sul diritto d'autore secondo la Legge 22 aprile 1941 n. 633 (in particolare artt. 1 e 6). La data di deposito, nonché quella della discussione orale dell'opera, non dovrebbero lasciare adito a dubbi circa l'originalità della tesi di laurea rispetto ad eventuali utilizzi dell'elaborato successivi, anche parziali, senza l'assenso del laureando/ato autore. Secondo le interpretazioni di Legge più recenti, la sfera dei diritti sulla tesi è di pertinenza dello studente che redige e concreta l'idea (anche) del suo relatore in una forma tangibile, in una creazione originale. La tesi di laurea non può essere consultata né utilizzata da eventuali interessati senza il consenso del laureando/ato, titolare dei relativi diritti di autore morali e patrimoniali. Questi ultimi permangono in capo all'autore anche se una copia del testo viene ceduta alla Facoltà (al riguardo, va sottolineato che nessun regolamento di Ateneo può prevaricare i contenuti di legge) o spedita ad una casa editrice perché, secondo l'art. 109, comma primo, Legge 633/1941, la cessione di uno o più esemplari dell'opera non importa, salvo patto contrario, la trasmissione dei diritti di utilizzazione.

Preface

This work would not have been possible without the unwavering support and encouragement of my parents, to whom I express my deepest gratitude. Thank you for being by my side in all occasions.
I am extremely grateful to my mentor and the committee, who provided me patient advice, invaluable help and guided me through my research process. I am humbled by the support I received from you.

Pouya Houshmand

Contents

Preface i

Contents ii

Abstract iii

1 Introduction 1
- 1.1 Introduction 1
- 1.2 Anomaly types and detection settings 2
- 1.3 Machine learning basics 4
- 1.4 Supervised and unsupervised learning for anomaly detection 5
- 1.5 Detection techniques 6
- 1.6 Data representation and system regimes 8
- 1.7 Power consumption issue 10

2 Algorithm 13
- 2.1 Naive Bayes 14
- 2.2 Hidden Markov Model 15
- 2.3 Dynamic Naive Bayes Classifier 18

3 Digital implementation 21
- 3.1 Forward algorithm block 29
- 3.2 Feature extraction 36
- 3.3 Probability extraction 39
- 3.4 Floating point operation 44

4 Performance analysis 55
- 4.1 Simulations results 55
- 4.2 Power consumption estimation 60

5 Future work 65

Bibliography 67

Abstract

In an ever-growing interconnected world, the amount of data produced is increasing fast every day and keeping every source of information under control has become a hard and non-trivial task. Anomalies may arise in any instant in this humongous flow of information and human capabilities are simply not good enough in this scenario to achieve the detection in an efficient way. The automation of the task is necessary and special purpose devices must be designed to deal with the problem: the functionality of the service must be guaranteed at all times, especially with critical tasks.
The purpose of this thesis is to find the right algorithm which serves the purpose, its design with hardware description language and the estimation of the performance indexes of the synthesized implementation. Special focus is placed on the low-power design of the architecture, since power affects the reliability, cost and performances of modern chips.

Chapter 1

Introduction

1.1 Introduction

The definition of anomaly is something different, abnormal, peculiar, or not easily classified, which deviates from the common rule. Anomaly detection refers to the identification of these abnormal values which lie outside the usual distribution of data.
The task is challenging: the outlier is an unknown or unwanted condition of a system of which sole source of information are external sensor acquisitions. Moreover the detection must be carried on with the lowest latency possible, so as to provide time for intervention and prevent damages.
Raw incoming data from the sensors can not be used for the purpose since it does not convey enough information for the anomaly detection purpose.
To make an example, if we compare the single incoming sensor acquisition to words in a speech, we can not say if the single word belongs to the context in which it had been said or if it is gramatically correct: the single words of "*the cat is on the table*" or "*the is is is table*" or "*the on table is the cat*" are all known words that may not arise suspicion if taken by themselves, while are considered anomalies if considered the context in which they have been said.
The operation of extracting information from the raw data so as to be knowledgeable of the context in which the acquisition has been made is called *feature extraction*.
Once the features are extracted and enough information has been gathered regarding the state of the system, the computed knowledge can be deployed to find the anomaly. Machine learning techniques often prove to be useful to use such data to tackle the detection problem with optimal results.
Machine learning is one of the many fields of artificial intelligence which focuses on the development of algorithms that have the ability to *learn* autonomously the parameters of a model and improve their performances without the need of being externally programmed. The model is chosen by the designer so as to best fit the task that needs to be done.
Given the anomaly detection task, different models have been examined and the best one in terms of computational effort, latency and efficiency has been chosen as the right candidate for the job.

Finding the right algorithm is not sufficient: it must be implemented and shown to be employable in real case scenarios. In order to carry on the calculations it is not recommended to use a general purpose processor. A processor consumes too much power, takes too much time and therefore does not guarantee very small latency times that are needed for the purpose; what is needed instead is a device which can be embedded in small scales, that does not consume too much power and has a long autonomy and that it is optimized for the detection of anomalies with the lowest latency possible. To make an analogy we do not need a tank to go to the grocery store to buy a can of milk but a bike is *often* (depending on where you live) sufficient.
Hence the need for special purpose architectures that comply with the requirements.
The purpose of this thesis is to come up with an efficient and fast architecture that carries on all the tasks described in the previous paragraph, starting from the selection of the right features to the final implementation of the system.

1.2 Anomaly types and detection settings

Anomalies can be classified into three main categories: point, contextual and collective anomalies. The basic requirement for an anomaly detection system is to be able to recognize them.

1.2.1 Point anomaly

A *point anomaly* is a point in the time-series anomalous with respect to all data. This means that, referring to the statistical distributions of all the known values, the item does not belong to any previously known. To make an example, given that the daytime temperature varies from $-10°C$ to $+35°C$ throughout the year, a day with $+50°C$ is a *point anomaly* in the time series given by the sequence of the days.

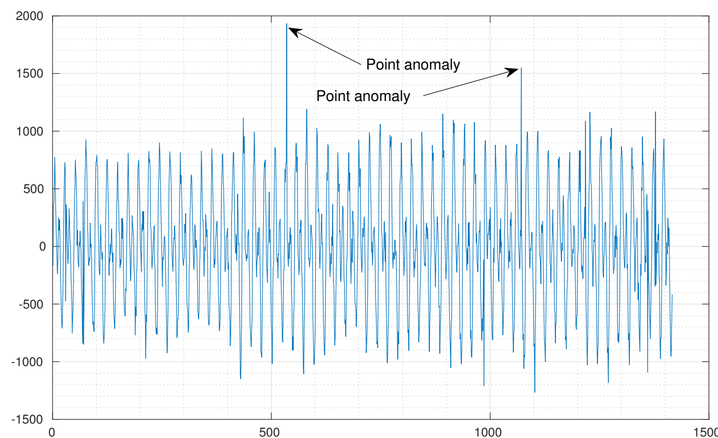


FIGURE 1.1: Point anomaly

1.2.2 Contextual anomaly

An item of the time-series is a *contextual anomaly* when, given the temporal context in which the value is obtained, it does not belong to it. This means that the anomalous value *belongs* to one of the known statistical distributions, in contrast to the point anomaly, but it is an anomaly since it is not characterized by similar features of its predecessors. By referring to the regimes of the system, a contextual anomaly is detected when a system is working in a certain regime and suddenly a value is detected that is highly probable to belong to another regime. In order to be detected, such an anomaly needs to take into account the probability of such a switch between regimes(or better *transition*) happening. An analogy of a contextual anomaly would be a baby with a long grey beard: even though a long grey beard might not be an anomaly on a human being at a certain stage of life, on the face of a baby would definitely be an unusual sight; in the same way the value of the data stream indicated in the figure belongs to the general distribution of data and would not be considered an anomaly if taken by itself, but is a clear spike with respect to the context in which it has been obtained.

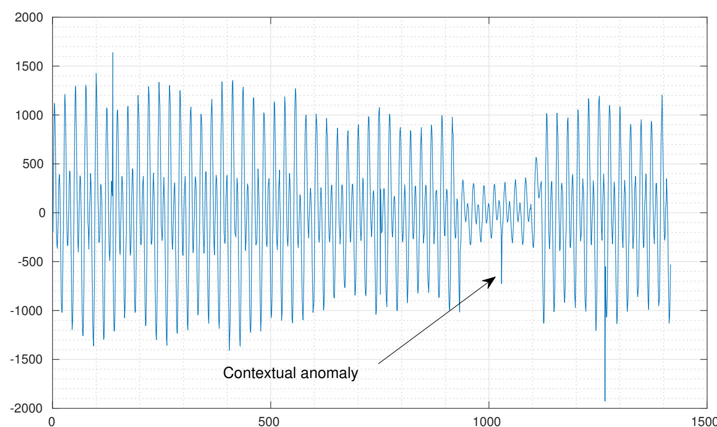


FIGURE 1.2: Contextual anomaly

1.2.3 Collective anomaly

A *collective anomaly* is defined as a series of items in the data set that are deemed as anomalies. If taken singularly they are not distinguished as point anomalies, nor as contextual anomalies. This means that they belong to the known statistical distributions and that they belong to the context in which they are observed. However they are detected when observed in sequence. An example would be to observe a fixed value for an undefined amount of time: while a pause of a couple of seconds between sentences in a speech is normal, a ten minute prolonged silence would probably cause some concerns in the audience regarding the health of the speaker.

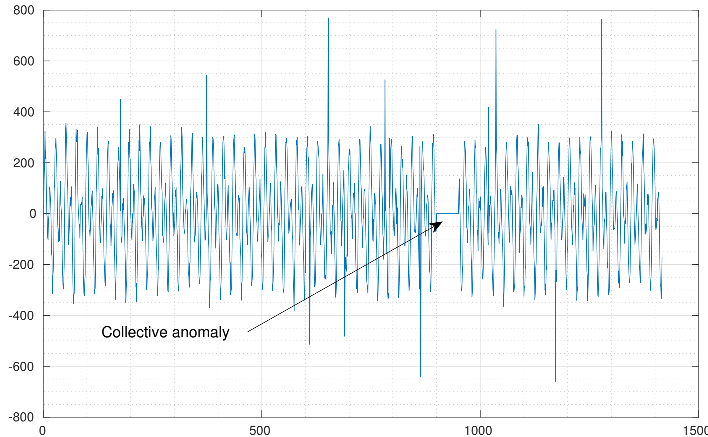


FIGURE 1.3: Collective anomaly

1.3 Machine learning basics

Generally speaking a machine learning system is a function L that given some inputs x belonging to an input space X and some parameter settings p which belong to a space P, are able to generate an output $o \in O$ through a learning process, which defines the parameters of the model.
The input and output data are organized in *data sets*, in which a set of *items* are contained. The items contained in the dataset belong to an instance space characterized by the *set of attributes* related to the items. As an example a time-series data set is characterized by a series of items that are characterized by a set of attributes such as the sampling time, the value of the sampling and/or other features. Depending on the knowledge represented in the dataset, different types of learning processes can be carried on; those processes which may fit our purpose are predictive or descriptive types. In the predictive type the model learns in a supervised way to predict, for a given item, the value of one of its attributes from the values of other attributes. Depending on the type of attribute to be predicted (if continuous or discrete) the prediction can be considered a regression or a classification. On the other hand the descriptive model is able to learn in an unsupervised way new attributes of the items contained in the dataset by mapping the items in the dataset into clusters depending on an index of similarity.
For our purpose, predicting might be considered the right choice, since given a set of attributes we might be interested in predicting how similar is the observed value to the predicted one and classify the observed value as an anomaly if it differs from the predicted value beyond a certain threshold. Nowadays multiple techniques exist for carrying on the prediction task on time-series data but those who have the best performances, such as recurrent long short-term memory neural networks, often involve long and time consuming operations. Clustering on the other hand implicates less operations, since for each item just the index of similarity has to be computed with respect to the various clusters, but

does not provide optimal performances in terms of precision and recall. More detailed insights are provided in the next section.

1.4 Supervised and unsupervised learning for anomaly detection

When it comes to machine learning techniques, the correct definition of the task to be carried on is crucial to choose properly the model so as to have the best performance. Depending on the task, the model has to be trained accordingly. Training a model is the process of providing the model with examples and evaluating the improvement of its performance indexes. The model is characterized by a training algorithm which upon arrival of new examples adapts its set of parameters so as to best fit the type of incoming data; the algorithm and the type of examples provided heavily influence the outcome of the training process.
The learning process is usually classified in two categories: supervised or unsupervised. Supervised learning is usually associated with prediction and classification tasks, while unsupervised is related to clustering tasks.
In the supervised case *labeled* data is fed to the system, which learns to recognize those cases which have previously been denoted as anomalies; the model is trained to identify a function f that maps the values from an input space X to an output space Y, with the Y values, which in our case are the anomalies, labeled as such and known beforehand. This obviously poses some limitation to the detection of the anomalies: they are by definition something that is unpredicted and does not conform to the normal behaviour of the system. Teaching a model in a supervised manner how to detect anomalies is a rather challenging task: all kinds of anomalies must be in the first stance foreseen and they must be present in the training set. So supervised learning techniques are limited in performance since they have rather high precision but poor recall and they require high computational effort and are time consuming.
Unsupervised learning algorithms on the other hand do not perform the same kind task of a model which is supervised, since the type of data they are fed with are different: the model learns a function f which maps instances from X to Y but in this case Y values are **not** defined during the learning process and the model characterizes autonomously the training set.
The task of the model is the recognition of the type of anomalies described before. Knowing that anomalies are by definition unpredictable, providing the model with a training set with labeled anomalies seems not to be a viable solution. So unsupervised learning might be considered the solution to the problem, but it may yield numerous false positive outcomes: since the model does not know the underlying state of the system it might not be able to differentiate between an anomalous value or a context anomaly. A feedback system is required so as to discover false positive or false negatives and online learning techniques are needed so as to correct the model on the go. This kind of learning may yield better recall factor since most of the anomalies are detected (even though often they are not) as the time goes by and the system develops and the model optimizes itself and this is done at the expense of the precision, which suffers from the big number false

positives of anomalies detected while the system optimizes itself.
It has to be taken into consideration however that it is much better to have a high number of false positives (which can be ignored by the final user and do not pose a threat) than having false negatives, which may go undetected and may cause severe damage. In the supervised method the number of false negatives is obviously higher than in the unsupervised case, given that the known anomalies are limited to those present in the training set.

1.5 Detection techniques

The type of items that make up the dataset that are considered in this work are time-series values, which are a collection of *ordered* items with respect to the moment in time they have been added to the set. In order to detect anomalies with such kind of data, the detection techniques to be adopted must be able to extract the underlying trends, cyclicities and/or seasonalities so as to be able to spot those values that do not belong to the context in which they have been acquired [Chandola et al., 2007]. To execute the task different techniques exist, which can be classified in window-based techniques, prediction based techniques and those based on Markovian models.

1.5.1 Window based

The time series is divided in a series of fixed sized windows, and the task of the anomaly detection algorithm is to find the outlier in each window separately.
The time-series set S is subdivided in p windows, where each window is obtained by sliding an initial window by m hops. The number of windows is equal to $|S| + m - 1$. This procedure is done for both the training sets and for the test sets. An anomaly score is given to each test window based on its grade of similarity with the training windows, with k-Nearest Neighbour or one class Support Vector Machines models.
The size of the hopping step affects the performace of the algorithm: a too small step may be computationally expensive since the number of windows to be analyzed would be rather big, while on the other hand a rather big step may cause loss of information. The major drawback of this technique is the high number of operation that have to be made and the fact that storing of all data in the windows is needed: as a matter of fact every pair of test and training windows have to be compared. This aspect goes against the low-power requirements that are today ever more needed.

1.5.2 Prediction based

The idea behind these methodologies is to detect anomalies as time-series values that differ from the predicted ones. In these methodologies the anomaly is usually a point anomaly which lays outside the normal behaviour learnt by the system. A very basic predictive model learns the behaviour of the model from m samples and predicts the value of the $m+1$ sample and afterwards, after the effective sampling of the $m+1$ sample, provides an anomaly score as a function of the distance between the predicted value and the sampled one, taking also into account some features of the data that may affect the

observed values (such as its variance). The various techniques that involve prediction differ in the kind of filter they implement to compute the prediction and can be classified in moving average (MA), autoregressive (AR), autoregressive moving average (ARMA), Kalman filters and many more.

Moving average (MA)

The filter is basically a FIR filter with as many b_i coefficients as the length of the history that is taken into account.

$$y(t) = \Sigma_{i=0}^{m} b(i)x(t-i)$$

In the case the true mean of the values is mean to be predicted then $b_i = \frac{1}{m}$ with m the length of the history of samples

Autoregression (AR)

Autoregression employs a recursive filter that makes use of the last m values such that

$$y(t) = \Sigma_{i=1}^{m} a(i)y(t-i)$$

Linear and non-linear ARMA

ARMA and NARMA models implemented with recurrent neural network are inherently made to filter out outliers. In the case of an anomaly in the time series, when compared to the output of the network, it will exhibit striking differences and be deemed as an anomaly.

Kalman filters

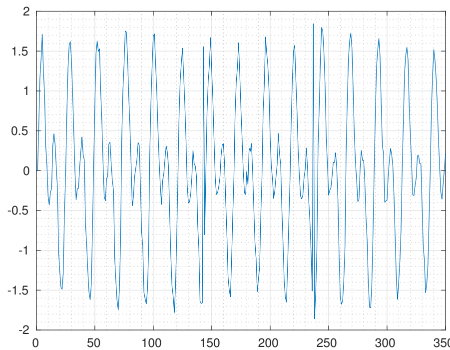
A Kalman filter is a recursive filter that infers parameters of interest from indirect, inaccurate and uncertain observations. Is an optimal estimator in the sense that, assumed that all observations are affected by Gaussian noise, it minimizes the mean square error of the estimated parameters. The estimated parameter is subsequently compared with the observed one and any eventual anomaly detected.

1.5.3 Hidden Markov Model based

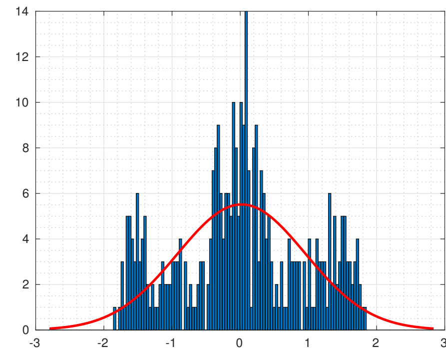
A Hidden Markov Model (HMM) characterizes the internal state of the system given its observable parameters. This means that the time-series used for training and testing is a series of observations that are expression of a certain underlying set of internal states of the system and that these states are linked to each other through a markovian process. By following Bayesian probability, the probability of being in a certain state is computed given all the observations. The anomaly is detected when the probability of being in a certain state given the observations falls below a certain threshold.
The biggest advantage is that they enable detection of all type of anomalies, be it point anomalies, contextual or collective anomalies, and this can be done on-line, without the need for batching data and analyze it offline. The premise, as it has been said, is that the sequence of inner states of the system follows a markovian model.

1.6 Data representation and system regimes

Every single sampling acquisition corresponds to an item in the time-series set; every item is characterized by a set of attributes and these attributes characterize the item in its context. In order to obtain a deeper understanding of the state of the observed system we can not rely simply on the absolute value of the sampling acquisition but we must resort as well to the features of the incoming data; these features provide a frame in which the single acquisition takes place. The features that we refer to are the **average**, the **variance** and the **slope**.
Selecting a proper set of features is crucial in identifying the regime in which the system is working and detecting the anomalies: those which have been chosen concur to the identification of the three categories of anomalies.
Incoming data and its features are usually characterized by a certain probabilistic distribution. In our implementation we assumed that all distributions are gaussian: uniform distribution is usually associated with white noise and its features are useless to our purpose; beside that it is assumed that the algorithms that preprocess the data filter out useless noise. By considering only 1 variable therefore in a 1-dimensional attribute space, incoming data can be characterized by a certain mean and by a certain standard deviation in a given time frame in which the system works in the same regime.



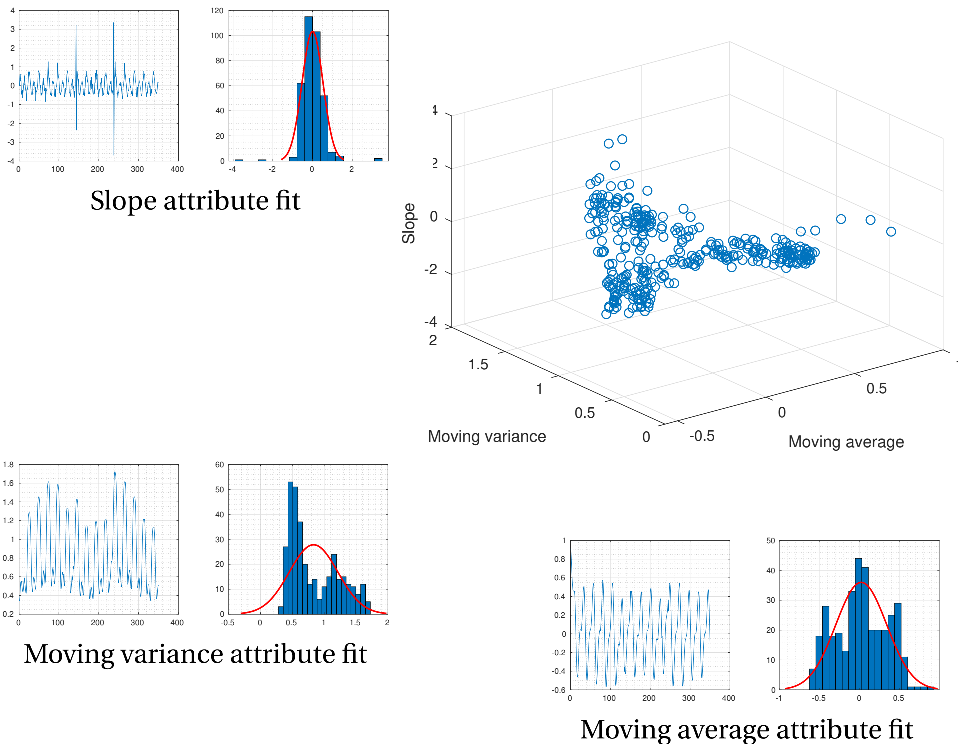
Data stream



Data stream Gaussian distribution fit

Consequently if we observe data in a multi-dimensional space, in which we consider all attributes of the items, we do observe some sort of clustering in the input space. A very interesting point is that for every regime of the system, a particular cluster is observed. What can be therefore assumed is that if the system works in a non-anomalous state, incoming data and its features belong to one of the known clusters.
By referring to the data stream of the previous figure, the attributes (slope, average and variance) can be extracted and fitted to their relative gaussian distribution. These values

can be represented in a 3-dimensional attribute space as in the following figure

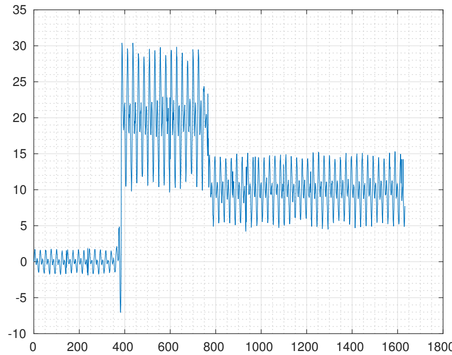


Slope attribute fit

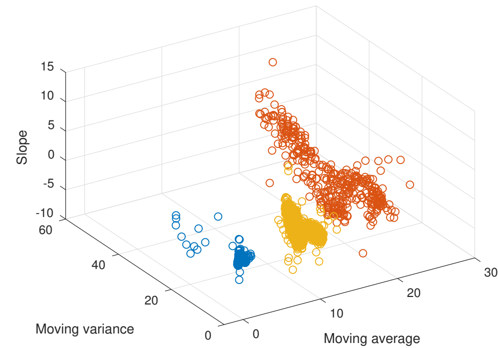
Moving variance attribute fit

Moving average attribute fit

However, as it has been noted before, this considerations are useful if we work with a limited number of dimensions and that the attributes that we consider truly are representative of the state of the system: if these requirements do not hold what we would observe if we were to use any similarity measure to evaluate the clusters would be a dominance of irrelevant attributes with respect to relevant ones. Moreover a high dimensional space is harder to represent than one with less dimensions in terms of computational effort. The previous problem is often referred in machine learning as *curse of dimensionality*



Data stream with different regimes



Regime attributes scattering plot

In the example displayed, the items in the dataset are defined by three attributes, which correspond to the three dimensions. The different clusters should be easily distinguished from one another, as in this case.
The clustering of data in the multi-dimensional space provides a very insightful perspective on the state of the system. It is useful to remember that nothing is known on the internal state of the system under investigation and all the information must be retrieved from the data. Since it is also known that the system works under different regimes, and it is assumed that under different regimes the incoming data varies its features, it can be concluded that every single cluster observed in the multi-dimensional attribute space corresponds to one particular regime of the system. This consideration will come handy when looking for the right algorithm for the anomaly detection.

1.7 Power consumption issue

When it comes to choosing the right algorithm, beside taking into account the performance of the algorithm in terms of accuracy, precision and recall, the computational cost must be considered as well. The number of operations needed to perform the algorithm, which determine the computational cost, determine the feasibility of the model in a real-case scenario. It is not enough to have an algorithm that detects any anomaly in any context with extreme accuracy, it must also be capable of delivering the detection in a reasonable amount of time without needing considerable amounts of power to carry on the computations.
For the aim of this work, the anomaly detection task must be carried on ideally with zero latency and with minimum power consumption. In a real-case scenario the device must be able to perform its task integrated in the system it is analyzing and without needing frequent battery charge or change.
Nowadays mobile and wireless devices are ever more used for consumer and industrial application in a massively connected world and the anomaly detection task must be

integrated in these kind of system. These devices require as less power as possible so as to need less battery charge or change, which is a costly task. Battery change involves man-power, which means money and time cost, and some application might need pervasive and/or risky operations to carry on the task, such as in implanted devices or devices kept in hazardous areas. Therefore to reduce the recurrence of this operations would bring significant benefits.

1.7.1 Sources of power consumption

To execute the anomaly detection task a processor is required and the basic elements which make up the processor are the transistors. These elements are the cause of two main sources of power consumption: static and dynamic. The sum of the two components determines the overall consumption.

Static power consumption

Static power consumption is related to the leakage currents that flow in the transistor. It has to be remarked that real transistor are non-ideal switches: even in non-switching state their resistance R_{off} is not infinite and there are multiples paths between the supply and the ground in which current flows. This generates power consumption, given by $P_{leak} = I_{leak}R_{off}$ + others.
Given the fact that the leakage is dependent on the OFF resistance of the transistors and not on the switching, leakage power is correlated to the area occupied by the chip. The higher the area, the higher the static power consumption

Dynamic power consumption

Dynamic power consumption is related to the switching of the transistors. Given that transistor are not ideal and their switching time is non-zero, power is consumed to charge the gate and internal capacitances of the transistors as well as during switching time, in which a short-circuit path is formed between supply and ground.
The dynamic power consumption is expressed as a function of the capacitances, the voltage supply, the operational frequency and the switching activity.

$$P_{dyn} \propto C_L V_{DD}^2 f E_{sw}$$

Dynamic power consumption reduction is the main concern in our design and we seek to achieve it by working at different levels of the design hierarchy.
The design hierarchy is made of the following :

1. Algorithm level: is the highest level of abstraction and it is defined by the kind of operations to be carried on. An algorithm that requires less computations and/or less memory storage and serves the same purpose reduces drastically the power consumption

2. Behavioural level: is the level in which the steps of the algorithm are characterized and the sequence of operations is defined. Power optimizations can be achieved by

parallelizing, pipelining, loop unrolling, retiming and substition or rescheduling of the operations.

3. RTL level: at register transfer level (RTL) the hardware blocks needed to carry on the computations are determined, described by their behaviour rather by their logic structure. The major components which define the power consumption at this level are the interconnections, the number of executing blocks and memory cells needed: the optimization can be done by multiple techniques, such as adopting certain coding schemes, precomputing architectures and/or clock gating.

4. Circuit level: this level corresponds to the netlist in which the RTL blocks are synthesized. Power savings can be achieved by means of reducing the glitching activity on the single nodes by working on the circuit function, technology and topology.

5. Gate level: Is the lowest level of abstraction and refers to the gates which compose the nodes in the netlist. The power savings are dependent on the technology used and the design of standard cells.

Unless different technology nodes are used in the same design, the higher the level of abstraction the greater are the savings: those which were interested by optimization in this work were the algorithm, the behavioural and the RTL level.

Chapter 2

Algorithm

The approach taken to detect anomalies in the system under observation is one of *identification* of the state of system (in which regime is working) and afterwards *detection* of the eventual anomaly given the state identified as most probable.
In order to carry on with this task a probabilistic approach is the most suited since it provides the probability of being in a defined state which is compared to a threshold that establishes the boundary between ordinary and anomalous behaviour.
It is crucial to keep power saving in mind when searching for the suitable algorithm: the higher the level of abstraction optimized the higher the power-savings, as explained in the previous section. An important factor to keep in consideration is the amount of computation needed and the memory storage needed to carry on the detection. The number of operation has to be minimized so as to reduce the amount of cycles needed (and therefore lower the operating frequency) and the switching activity. If the algorithm enables re-utilizitation of variables across different cycles is even better.
The reasoning which led to the identification of the algorithm is the following:

1. Given that the inputs that we can exploit for our purpose are the time-series sampling acquisitions coming from the system under analysis and their extracted features, we can assume that their probabilistic distribution is a Gaussian one (or a mixture of Gaussians)

2. Given this probabilistic distributions, we may use them to provide a probabilistic value for the different states of the system considering the history of other acquisitions as well by using a classifier.

3. Once the classification of the current regime is done it is a matter of comparing the probability of being in the most probable state in a given instant with a threshold associated with the state.

Provided these considerations a Dynamic Naive Bayes Classifier (DNBC) has been found to be the optimal choice.
The DNBC is the combination of two algorithms: the Naive Bayes classifier and the Hidden Markov Model; before going into detail about the DNBC an explanation of the Naive Bayes classifier and the Hidden Markov model is given.

2.1 Naive Bayes

Naive Bayes is a simple probabilistic classifier, which provides the most probable classification given the the likelihoods of a set of predictors.
The computation of the posterior probability of the class involves the heavy assumption of independence between predictors and therefore its outcome must not be regarded as the precise probabilistic value of the classification. This assumption is not realistic: predictors do have some degree of dependency among themselves. However, to fulfill only the classification task, the assumption may be held true and the class for which the posterior probability is maximized effectively corresponds to the most probable class. The Naive Bayes classifier, even though it does not provide accurate results given its assumption of independence between predictors, it does provide surprisingly well classifications; we can conclude therefore that a correct classification can be carried on without an exact estimation of the probability of the classification.
Optimal prediction is given by

$$y^* = \underset{y \in Y}{arg\,max}\, p(y|x_1, ..., x_n)$$

from which we can derive by applying Bayes

$$\begin{aligned} p(y|x_1, ..., x_n) &= p(x_1, ..., x_n, y) / p(x_1, ..., x_n) \\ & p(x_1, ..., x_n|y) p(y) / p(x_1, ..., x_n) \\ & p(x_1|y) p(x_2|y) ... p(x_n|y) p(y) / p(x_1, ..., x_n) \end{aligned}$$

and therefore

$$y^* = \underset{y \in Y}{arg\,max}\, p(x_1|y) p(x_2|y) ... p(x_n|y) p(y)$$

the denominator $p(x_1, ..., x_n)$ can be dropped since it does not affect y.

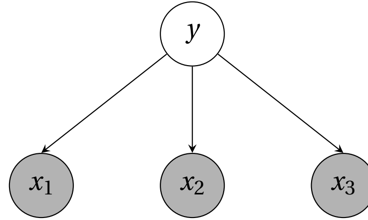


FIGURE 2.1: Naive Bayes classifier with 3 predictors

$$y = \underset{y \in Y}{arg\,max}\, P(y_j) \prod_{x \in X} P(x|y)$$

The inference process does not involve heavy computation since is just a series of multiplications. This feature makes it suitable for our low power requirements: less multiplications means less switching activity of the transistor which translates in less dynamic power consumption.

2.2 Hidden Markov Model

The Hidden Markov Model (HMM), as explained in [Rabiner, 1989] is a model that can be applied to systems which are characterized by a set of states, linked by a discrete Markov process, so as to identify the sequence of the hidden states of the system given a sequence of observations related to the system. Before detailing the features of the Hidden Markov Model, the characteristics of a Discrete Markov process are briefly summarized

2.2.1 Discrete Markov Process

A system can be modeled by a Discete Markov process when it can be described at any moment by one of a set of S distinct states $S_1, ..., S_S$. The system changes its state in time according to a set of probabilities related to a transition probability matrix T.
The sequence of instants in time is indicated as $t = 1, 2, ...$ and the state at time t is indicated as q_t. The probability of being in a particular state given the sequence of its predecessing states, defined as

$$P(q_t = S_j | q_{t-1} = S_i, q_{t-2} = S_k, ...)$$

can be truncated to just the current and the first predecessing state as

$$P(q_t = S_j | q_{t-1} = S_i)$$

and this special case is called *first order* Markov chain.
All the combination of sequence of states is the set of state transition probabilities, contained in the transition matrix T, which elements are defined as

$$T_{ij} = P(q_t = S_i | q_{t-1} = S_j)$$

with $T_{ij} \geq 0$ and $\sum_{j=1}^{S} T_{ij} = 1$ because of standard stochastic constraints.

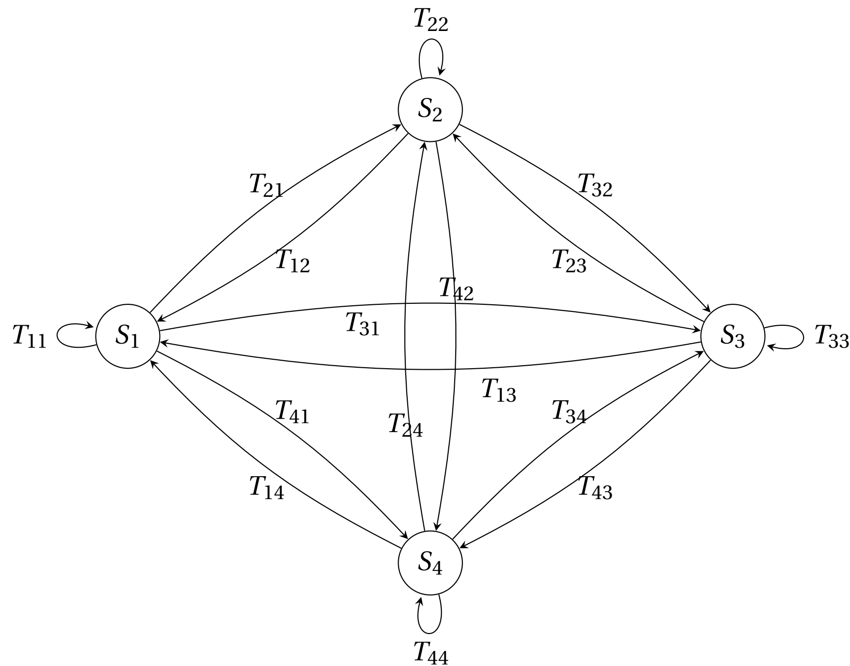


FIGURE 2.2: Markovian process

The Markov process assumption can be effectively applied to real devices, which possess a certain set of possible regimes from which they transit with a certain probability. If the gears of a car are taken as an example, the transitions from one gear to the next or previous one are linked by a certain probability if the gear is correctly operated. An abnormal transition, such as going from the sixth gear to reverse gear would be characterized by a very low transition probability, while a normal transition, such as going from second gear to third gear, would possess a much higher transition probability.

2.2.2 Extension to Hidden Markov models

In the case of Discrete Markov processes the state of the system in the current instant is assumed to be known with probability = 1. However, for real applications, the *observable events* do not necessarily mirror the state of the system.
A very straightforward example is the following: given that today is rainy and the temperatures are around $10°C$ can we automatically assume that we are in autumn? In this example the observable is the weather and the hidden state of the system is the season. The Discrete Markov process can not therefore be applied in this case: a more appropriate approach is to assign to each observable event a probability related to the system it may belong. The state of the system will therefore be considered not certain a priori but *hidden*.

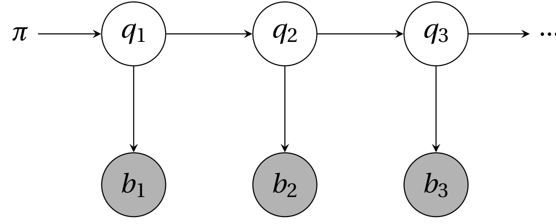


FIGURE 2.3: Hidden Markov Model sequence of states

A Hidden Markov model is defined by the following elements:

- S hidden states, which are a set of states $S = S_1, ..., S_S$, and each state a time t is define as q_t
- The matrix of transition probabilities $T = T_{ij}$ defined as

$$T_{ij} = P(q_t = S_i | q_{t-1} = S_j)$$

- The matrix of the probability distributions of M observables for each state $B = b_j(k)$ with $1 \leq j \leq S$ and $1 \leq k \leq M$
- The priori distribution probability of states $\pi = \pi_i$

$$\pi_i = P(q_1 = S_i) \qquad \text{with} \qquad 1 \leq i \leq S$$

A complete parameter set of an HMM is defined by $\lambda = (T, B, \pi)$

2.2.3 The main tasks of a Hidden Markov Model

The HMM can be used in real life applications for some particular tasks: *evaluation*, *decoding* and *training*. Following is a brief explanation of each.

- **Evaluation**: evaluates the probability of a certain sequence of states being produced by the model given the observations. The task is carried on by the *forward algorithm*.
- **Decoding**: given a certain sequence of observations, it parses the most probable sequence of hidden states. Is based on the *Viterbi decoding algorithm*
- **Training**: find the optimal model parameters $\lambda = (T, B, \pi)$. To carry on this task 3 algorithms can de employed: MLE (maximum likelihood estimation), Viterbi training, Baum-Welch (also known as forward-backward algorithm).

The task which is more fitting for our purpose is the evaluation one, by means of the forward algorithm. Evaluating at each step which are the probabilities of being in any state is more useful because the computed values can be compared with a given threshold and anomalies easily detected.
Decoding is mainly used for speech or handwriting recognition since it provides the most probable sequence of states, by recursively analyzing previous steps, and would cancel out any outlier states.

2.3 Dynamic Naive Bayes Classifier

The Dynamic Naive Bayes classifier combines the classification feature of the Naive Bayes with the evaluating ability of the HMM so as to find the most probable sequence of states given a set of observables per time instant.
At each single time-step, or whenever samples are available to be processed, the description space X should be translated into a target space Y. In our case the description space X is the set of observations and features extracted from the observations that concur to the determination of the hidden state of the system, while Y is the target set of hidden states that have to be decoded.
Every single attribute in the description space is defined by a probability distribution: in our case we assumed that all attributes are characterized by a Gaussian distribution with a certain mean and standard deviation *in each state*. This has been observed by doing histograms of the distribution of values in the different states in ordinary regime of the system, so without anomalies, as explained in section 1.6. Nevertheless it is worth noting that to have a proper and more faithful representation of the distribution of the values in the description space a single Gaussian is often not enough and multiple Gaussian mixtures may provide an improved probability estimation. In the mixture of Gaussians each Gaussian is characterized by a certain weight and the overall probability is given by the sum of all Gaussians.
Proper care must be taken when considering a mixture of Gaussians since the model may **overfit** and not provide accurate results. Given a certain distribution of data upon which train the model, if the number of Gaussian mixtures is chosen exactly so as to fit the training data it may be detrimental when the model is simulated with the validation set. In the designed model just a single Gaussian has been considered so as to not risk overfitting.
A mixture of Gaussians implies heavier computation as well and hence more power consumption. The choice for a single mixture gaussian is therefore dictated by the low-power requirements and by the risk of overfitting the model.
Another option, promptly discarded, would have been to use tabular format to represent the values, by quantizing the values in intervals in an offline stage and assigning to each interval its corresponding probability given its context. However the tabular format, even though much more power efficient, does not provide with enough information so as to detect eventual anomalies: since we are dealing with continuous values, quantization would mean loss of information.
Given that the anomaly is defined as an item outside the boundaries defined by the clusters of the normal behaviour, we should define what we mean by outside and how this is quantized. Since we assume that the distribution of the values of the attributes in the single clusters follows a gaussian distribution (or a mixture of many,for our explicative purpose will consider only one Gaussian) we can assign to every single item in the multidimensional attribute space a particular probability of belonging to each of the clusters. The DNBC can be represented schematically as

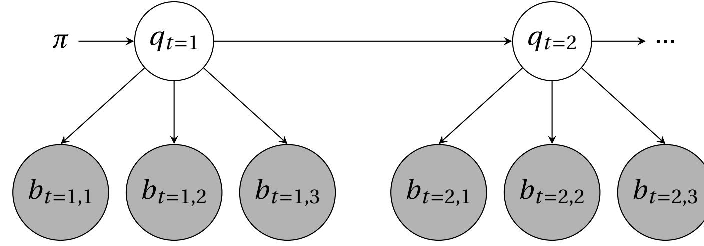


FIGURE 2.4: Dynamic Naive Bayes Classifier sequence of states

Fig. 2.4 shows the dependencies of each node: the probability of the hidden state q_t at time t depends only on the probability of q_{t-1}, while the probabilities of the observations at t $(O_{t,1},...,O_{t,V})$ depend on the discrete hidden state at t.
As a quick summary, the system under analysis is characterized by a set of regimes in which it works. The transitioning between one regime and another is defined by a certain probability and is a Markovian process. The underlying current regime in which the system works is not known a priori with certainty but it must be extracted by evaluating what can be observed from the system. This classification task is done by considering a set of features which concur to the identification of the underlying regime of the system. A classification task based upon the features observed it is not enough to define the state of the system given that the state of the system also depends on the sequence of states that preceded it. A Dynamic Naive Bayes classifier is a suitable model for the task since it expresses all the dependencies described and necessitates very few operations to carry on its task.

2.3.1 Forward algorithm

To compute the value of the probability of the hidden state q_t^i, which will be indicate with α_t^i, with $1 \le i \le N$, the forward algorithm is employed.
Given a model $\lambda = (\pi, T, B)$ of the Hidden Markov chain and given the probability of the observation given the state (equal to $b_j(O) = \prod_{i=1}^{M} P(o_i|S_j)$) the steps are the following:

1. **Initialization**: $\alpha_1^i = \pi_i b_i(O_1)$ with $1 \le i \le N$
   Given the priori probabilities, it computes the α probabilities given the first set of observations O_1 at time $t = 1$.

2. **Induction**: $\alpha_{t+1}^i = [\sum_{j=1}^{N} \alpha_t^j T_{ji}] b_i(O_{t+1})$ with $1 \le t \le T-1$ and $1 \le i \le N$
   Computes recursively the α probabilities over all possible states, given the observations.

3. **Termination**: $P(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i)$

The trellis figure helps to summarize the idea behind the forward algorithm:

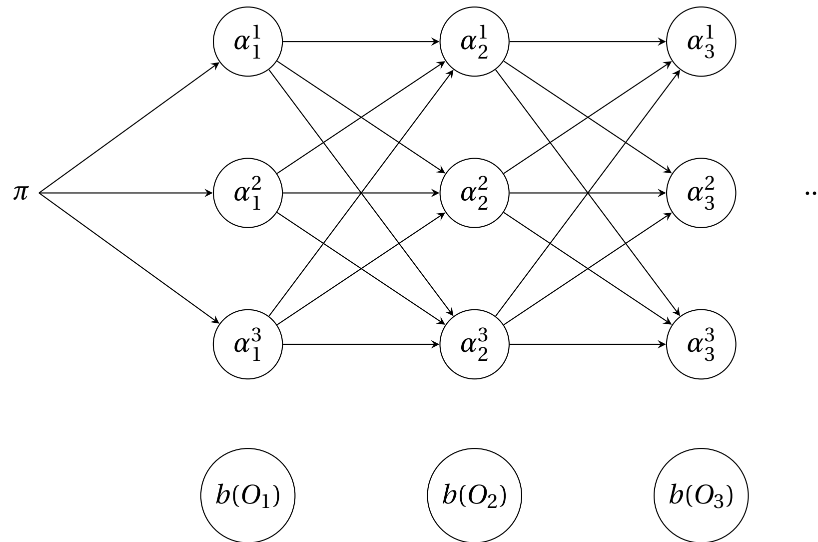


FIGURE 2.5: DNBC Trellis

Every arrow depicted in the figure describes the probability of going from a certain state s_x at time time t to state s_y at time $t+1$. The value of each arrow is computed as $\alpha_t^{s_x} T_{yx} b_y(O_{t+1})$. The overall probability of being in a certain state is the sum of all arrows that point to the given state. The value of $b(O_t)$ is the product of the probabilities of observing the features in the timestep t, and has a different value for each state.

Chapter 3

Digital implementation

The architecture consists of three main blocks which execute the operations, a set of register files which hold the required parameters for the correct operation of the system and a control unit that controls the flow of instructions in the system.

3.0.1 Execution blocks

The three main blocks implemented are the feature extraction block, the probability extraction block and the forward alogrithm block.

The incoming data stream is fed to the feature extraction block, which provides N features outputs, which are subsequently sent to the probability extraction blocks, which given the parameters of the probability distribution of the features computes the $b(O_y)$ values. Finally these values concur to the execution of the forward algorithm, which outputs the most probable state α_{MAX} probability, to be compared to the relative threshold in order to detect anomalies.

Feature extraction

As explained in the previous chapters, features have to be extracted from the incoming stream of data so as to have a deeper understanding of the state of the system under exam. These features consist of the **average**, the **variance** and the **first derivative** of the data; therefore $N = 3$ in this scenario. The feature extraction block carries on this task by storing the input data and outputting the required information in 4 clock cycles. Since in our system we consider a single stream of input data, a single block for feature extraction data is needed, placed at the beginning of the structure. The output of the said block is stored in three separate registers so as to guarantee synchronization and avoid switching activity

of blocks further down the architecture.

Probability extraction

After the extraction of the features the related probabilites have to be computed. In order to compute the probabilities of the obtained values, the set of parameters related to the probability distributions (the mean and the standard deviation of the gaussian distribution) must be available and readily retrieved. The calculation of this set of parameters should be carried on during the training phase. Once computed, these values have to be stored in the designated register files via the right instructions (LOAD MU and LOAD STD). As soon as they are available the execution of the task can be carried on by the block.
The probability extraction module is made up of two separate sub-blocks: the *Z-score computation block* and the *standard normal probability extraction block*. In order to avoid computing the Gaussian distribution by directly considering the generic formula, which implicates a long and power consuming series of operations, the values are firstly fitted in the standard normal distribution by computing their Z-score and afterwards the standard probability related to the Z-score is computed. The figure represents the fitting of a Gaussian distribution with mean $\mu = 25$ and standard deviation $\sigma = 4$ to a standard distribution with $\mu = 0$ and $\sigma = 1$.

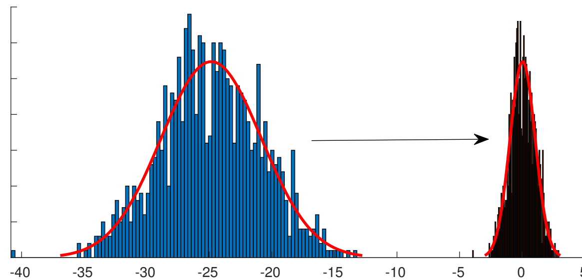


FIGURE 3.1: Standardization of a Gaussian distribution

The probability extraction operation is carried on in two cycles for each feature, in a pipelined fashion: first the Z-score is computed and stored in a register bank and afterwards its related probability extracted.
The number of probability extraction operations to be done depends on the number of features and the number of states of the system. In order to carry on all the computations two different architectures can be deployed: a pipelined structure or a parallelized structure. Given N features and S states, the two architectures are compared

- In a pipelined structure, the area consumption is reduced since the two sub-blocks are sufficient do the job, but this is traded with a greater number of clock cycles needed to terminate all operations. The total number of clock-cycles required

increases proportionally with respect to the number of features N and with the number of states S.

- In a parallelized structure, the area consumption is greater with respect to the pipelined structure but the total number of cycles is greatly reduced. In order to optimize the cycle reduction, N sub-blocks would be placed in parallel and each of them would carry on the probability extraction, so that $S+1$ cycles are required.

Given that the number of cycles needed greatly impacts the minimum operating frequency, which in turn impacts the dynamic power consumption of the structure, the parallelized solution is chosen so as to minimize the number of cycles required.
The first value, $b(O_y|s_1)$ is obtained after 2 cycles and the joint probability of observing the given features is computed by multiplying $\prod_{k=1}^{S} P(O_y|s_k)$ so as to have $b(O_y)$.
After $S+1$ cycles all $b(O_y)$ are computed and fed to the next block, the forward algorithm block.

Forward algorithm block

The block carries on the sequence of operations explained in section X. Since the algorithm does not need the $b(O_y)$ up until the partial sums $(\sum_{i}^{S} \alpha^i T_{ji})$ are ready, the block starts its operations at the beginning, right when the feature extraction is doing his task: while the features and the probabilities are being extracted, the forward algorithm block is proceeding in its operations up until the point it needs the $P(O_y)$ values. At that point stalls and waits for the probabilities to be ready and proceeds in its task. This greatly reduces the number of cycles needed to provide a final output: instead of having $N \times S + S^2 + 4$ in a purely pipelined architecture, the output α_{MAX} is ready after $S^2 + 3$ cycles.

3.0.2 Register files

In order to store parameters and partial values some internal registers are needed. To extract the probabilities the μ mean and σ standard deviation of all the gaussian distributions related to all states and features have to be stored in dedicated files; given N features and S states, $2 \times N \times S$ are required for this purpose.
Moreover, given that the forward algorithm carries on its operation in parallel with respect to the rest of the system and the system stalls until the forward algorithm has reached the state in which it needs the features joint probability, a register file (sized S) is needed, in addition to the previous ones, to store the $P(O_y)$ probabilities.

Execution unit overview

The feature extraction block (F.E.) provides the inputs for the Z-score computation and probability extraction blocks, which are parallelized as explained in the previous section to reduce the overall number of cycles needed. Once the values, computed in parallel, are ready, their product is computed and sent to the forward algorithm (F.A.) block. In

between each step a register bank is placed so as to guarantee the synchronization of the data.

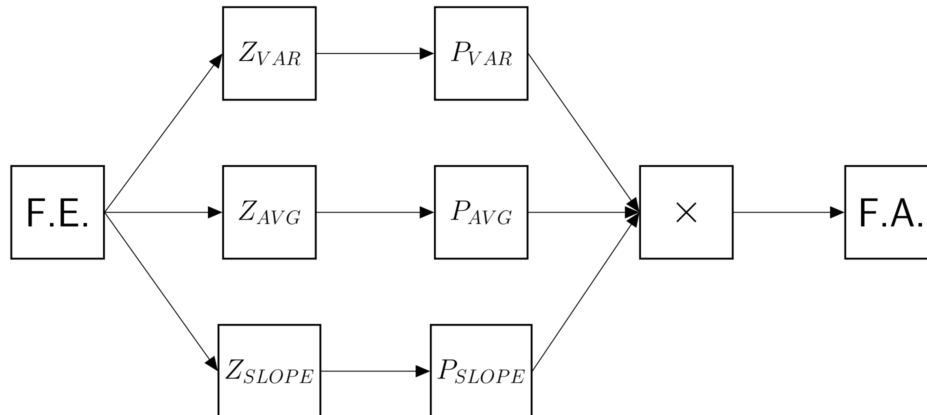


FIGURE 3.2: Microarchitecture overview

3.0.3 Control unit

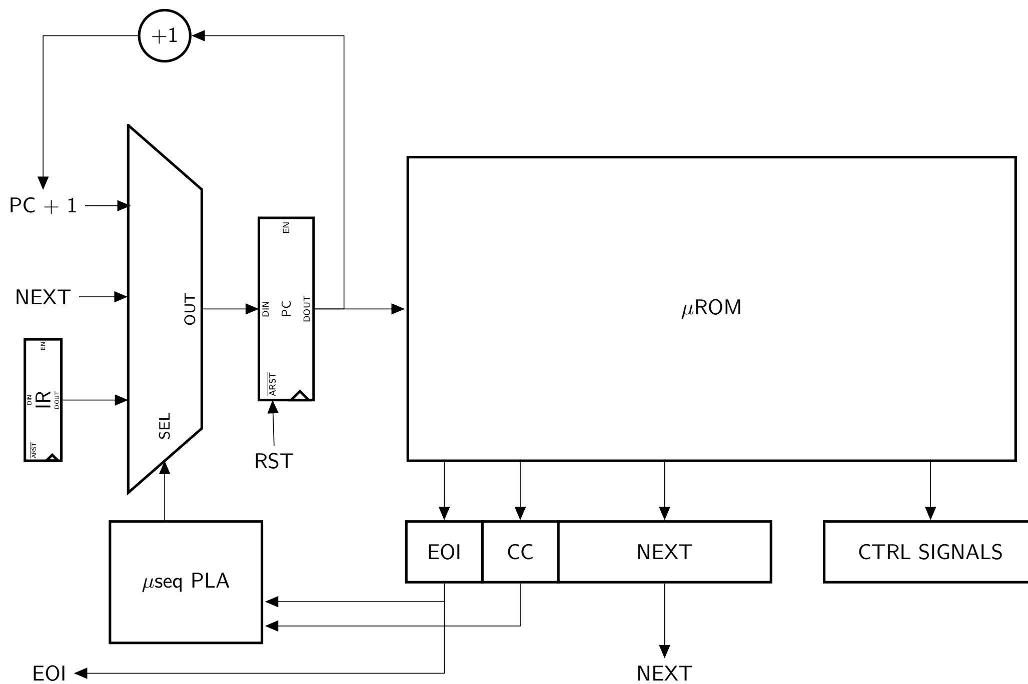


FIGURE 3.3: Micro control unit

The control unit is the block responsible for sequencing the instructions to be sent to the execution unit.

The instruction decoding and the elaboration of the next instruction occurr in a single clock cycle.
The PC register holds the current instruction: its value is sent as an input the combinatorial block which decodes it and generates the next instruction as well as the control signals to be sent to the execution unit. The combinatorial block can thus be partitioned in two sections: a microsequencer and a command generator. The microsequencer is, as the name suggest, the part related to generating the next instruction based on the current instruction, while the command generator gives out the control signals for the execution unit.
The output of the standard microsequencer is not just the next address value but contains other fields as well:

- **NEXT field**: contains the next instruction to be executed in case of non-sequential instructions
- **EOI field**: End-Of-Instruction field contains a single bit which indicates the termination of the task and the availability for fetching new instructions from outside. This bit is set to 0 while the algorithm is working and therefore does not accept new instructions, while is set to 1 when the in IDLE state or when α, T_{mat} or parameter values are being loaded in the system. The detailed explanation of the instructions is in the section (microsequencer)
- **CC field**: given that multiple status signal come from the execution unit, the CC bits indicate which combination of those to consider when elaboration the next instruction in the μsequencer selection block.
- **μSEQ field**: indicates whether there is a need for the μsequencer selector block or if it can remain inactive.
- **CTRL SEL field**: has the same purpose of the CC field, applied this time to the control signals to be sent to the execution unit.

In this implementation, in order to reduce the size of the combinatorial block, some fields have been excluded given that some could have been merged and others were useless: the μSEQ and the EOI field were merged since they had the same values in the same conditions while the CTRL SEL field was excluded given that no optimization was possible.
Since no JUMP conditions would ever happen in this implementation, there is no need for a Return-to-Address-Register (RAR) either.
Two situations can occurr in the flow of instructions:

1. Sequential flow: the PC is updated with its values increased by 1
2. Non-sequential flow: The PC is updated with the value contained in the NEXT field or in the IR register depending on the status signals coming from the execution unit or whether the EOI is active.

In order to carry on with the correct sequence of instructions the μSEQ selector block outputs the selection bits for the multiplexer set before the PC.

Microsequencer

STATE	ENCODING	NEXT STATE	EOI	CC
FE	0010	Z	0	0
Z	0011	ZP	0	0
ZP	0100	ZPM	0	0
ZPM	0101	ZPM	0	1
PM	0110	M	0	0
M	0111	WAIT	0	0
WAIT	1000	WAIT	0	1
M0	1001	M0	0	1
W1	1010	W2	0	0
W2	1011	FE	0	0
LOAD MU	1100	IDLE	1	0
LOAD SIGMA	1101	IDLE	1	0
IDLE	0001	IDLE	1	0
RESET	0000	IDLE	1	0

The instruction flow has been optimized so as to reduce latency as much as possible. The only instructions which can be used by the external user are the start instruction (FE), the LOAD SIGMA and LOAD MU ones (which define the parameters for the extraction of the probabilities) and LOAD CNT, asserted to define some control parameter which ensure the correct sequence of instructions.

FE :The feature extraction task is done in one clock cycle, the forward algorithm starts its cycle.

Z : The Z-score of a given feature for a given state is carried on

ZP : The Z-score and the probability extraction blocks are working in parallel in the same clock cycle. They work with different given values.

ZPM : The Z-score computation block, the probability extraction block and the joint probability of the computed probabilities are carried on in parallel with different values.

PM : The probability extraction block and the joint probability are computed.

M : The joint probability of the different features is computed

WAIT : No block is active except for the forward algorithm module, waiting for the forward-algorithm to carry on its computation and output the status signal necessary to carry on

M0 : The forward algorithm accepts the observation probabilities O_y computed so as to finish the computation of the α factors

W1 : Only the forward algorithm module is active, the counter values are cleared so as to prepare for a new loop.

W2 : The threshold is ready to be evaluated at the output.

LOAD MU : Load the mean of the guassian distribution of the indicated feature at the indicated state.

LOAD SIGMA : Load the standard deviation of the guassian distribution of the indicated feature at the indicated state.

μsequencer selection control

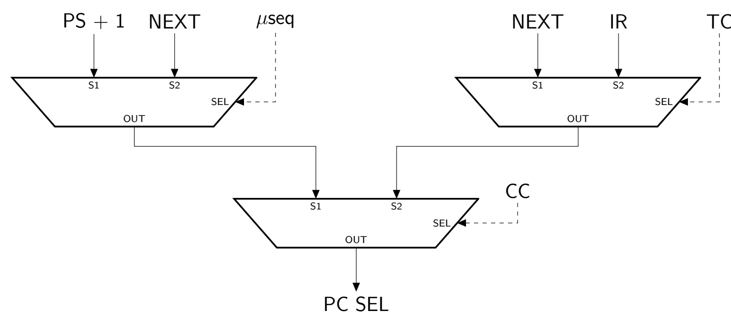
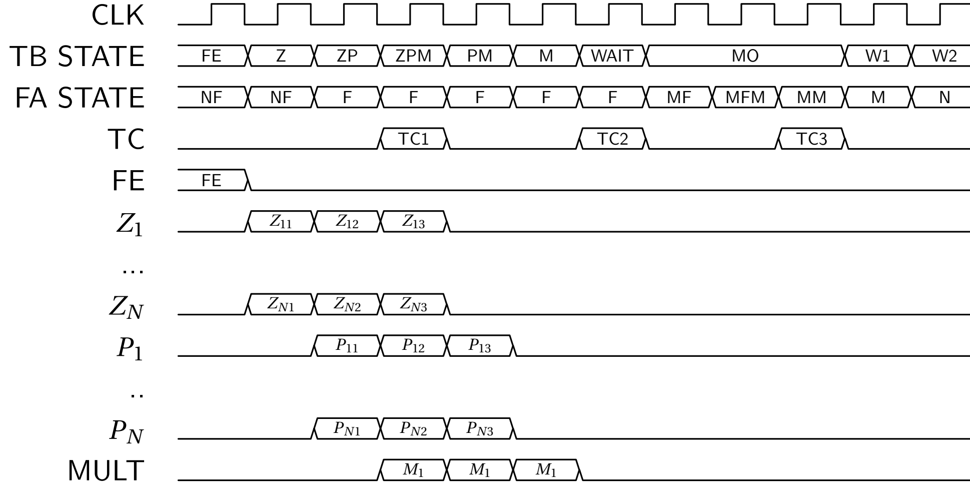


FIGURE 3.4: Microsequencer logic

Depending on the parameters contained in the instruction decoded (defined by the labels CC and μseq) and depending on a status signal coming from the execution unit (TC signal), the μsequencer chooses the correct selection signal to be sent to the program counter multiplexer so as to prepare the next instruction. Depending on the previously described signals, the program counter will store the incoming IR instruction, the instruction contained in the NEXT field of the output of the μROM in case of a jump or the incremented value of the present instruction, in case of sequential instruction.

3.0.4 Timing diagram



The forward algorithm instruction and the probability extraction of the features are carried on at the same time in order to reduce the overall number of cycles. The sequence of instructions has been chosen so as to never have data hazards; the probability extraction and the forward algorithm are interlaced and overlapped. The timing diagram represents a state with 3 states ($S = 3$) and N features.
The instruction flow of the forward algorithm has been described in section [SECTION] so the focus will be on the operation of the blocks outside the forward algorithm module. The fundamental requirement is that the $b(O_y)$ probabilities get computed *before* the F.A. block computes its partial sums.
In the first cycle the features are extracted, with the FE instruction which triggers the start of the cited task. In the next cycle the observables are available and the Z-score can be extracted. The counter, correctly set with the LOAD CNT instruction, provides the addresses where to look for the μ and σ values in the register files. The Z-score related to the first state are extracted.
In the next cycle the Z-scores of the first α computed can be used by the probability extraction block while the Z-score block computes the next α; the subsequent clock cycle the probabilities of the first α are extracted and the joint probability $\prod_{k=1}^{S} P(O_y|s_k)$ can be computed (M_1). Starting from the Z-score, the probability extraction follows and subsequently the joint probability is computed in a pipelined manner. Each block evaluates one state per cycle: the $b(O_y)$ values are therefore ready after $S+3$ cycles, with S the number of states. The F.A. block requires the first $b(O_1)$ after $S^2 - S + 1$ cycles, therefore in any case with $S \geq 3$ the system works fine.
Once the $b(O_y)$ have been computed and stored in the dedicated register file, the TopBlock must wait for the forward algorithm module to compute all its partial sums and enters in the WAIT state.
When the partial sums are computed, the system enters the M0 state and the forward algorithm starts evaluating the $b(O_y)$ computed. The forward algorithm block carries on with its max detection and outputs the threshold in the W1 state.

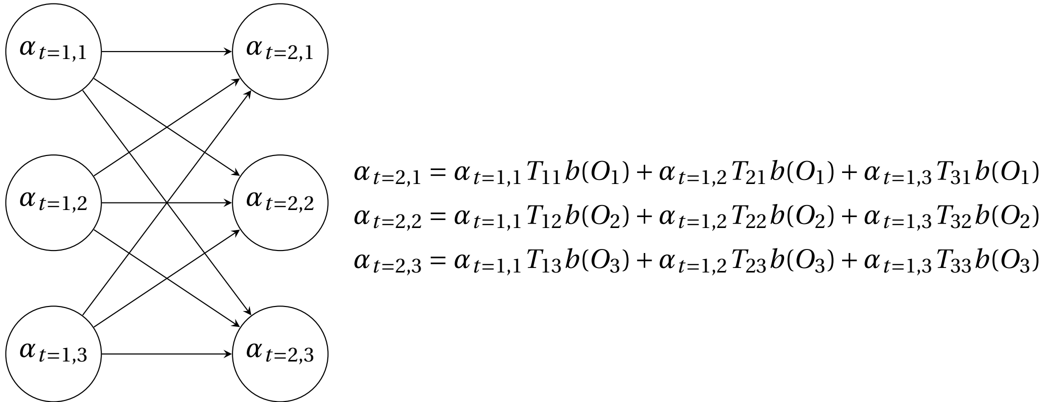
3.1 Forward algorithm block

3.1.1 Forward algorithm operations

In order to compute the probabilites of the states (denoted by α) we have to compute

$$\alpha_{t,i} = \sum_{k=1}^{S} \alpha_{t-1,k} T_{k,i} b(O_i)$$

The α value for each state at each timestep is the sum of products of the overall probabilities of the α values of the previous timestep (α_{t-1}), multiplied by the probability of transitioning to the previous state to the next state ($T_{k,i}$) multiplied by the probability of having the observations of the current time step for the interested state $b(O_i)$.
In other words for each state we compute the joint probability of transitioning from state k to state i, the probability of having the kind of observations seen at the current timestep in the state i and the probability of being in state k in the previous timestep.



In order to compute the α factors of the current timestep we would have to retrieve from memory $3 * N^2$ with N the number of states of the system.
Since collecting data from memory is an expensive operation in terms of power consumption and delay, the re-utilization of the values is fundamental. Whenever possible the stationarity of the values across multiple cycles must be exploited.
By observing the algorithm, data stationarity can be exploited with the α factors of the previous timestep, by computing the first product ($\alpha_{t=1,1} T_{12} b(O_2)$ across all states and storing the result in a dedicated register file. The dedicated register file, which is the *partial sum* register file, holds N values. By doing so we reduce the number of retrieval from memory of the α's from N^2 to N.
The observation probability values $b(O_y$ present stationarity as well and the algorithm can be rewritten so as to necessitate them only N times instead of N^2: this can be done by

collecting the $b(O_y)$ for each state and executing the multiplication just once.

$$\alpha_{t=2,1} = (\alpha_{t=1,1}T_{11} + \alpha_{t=1,2}T_{21} + \alpha_{t=1,3}T_{31})b(O_1)$$
$$\downarrow \quad \downarrow \quad \downarrow$$
$$\alpha_{t=2,2} = (\alpha_{t=1,1}T_{12} + \alpha_{t=1,2}T_{22} + \alpha_{t=1,3}T_{32})b(O_2)$$
$$\downarrow \quad \downarrow \quad \downarrow$$
$$\alpha_{t=2,3} = (\alpha_{t=1,1}T_{13} + \alpha_{t=1,2}T_{23} + \alpha_{t=1,3}T_{33})b(O_3)$$

In this example $S = 3$; the first three cycles of the instruction flow the three products which contain the red αs are computed, followed by three instructions in which the blue αs are computed and so on. The products computed can not be stored in a multiply-and-accumulate structure since they belong to different equations, and therefore have to be temporarily stored in the register file previously described.
Once the α's are computed, the biggest α will be considered as the most probable state and its value compared to the threshold related to the state so as to detect an anomalous state. If the probability of the most probable state falls below the threshold then an anomaly is observed.
Before starting again the elaboration of the α's for the next timestep, those of the current timestep need to be normalized in order not to run into underflow. Underflow happens when the values to be deployed become smaller than the smallest representable value: the probability of it occurring is rather high since we deal with probabilistic values (which are comprised between 0 and 1) and at each timestep the previous α's, which are probabilities, are used again as an input for multiplications. This generates a positive feedback phenomenon in which at each timestep the α values become smaller and smaller until underflow is observed.
The pseudo-code for the algorithm will be the following:

```
1   for x = 1:S
2     for y = 1:S
3       PS[y] = PS[y] + a_t[x]*T[x][y];
4     end
5   end
6
7   for y = 1:S
8     a_t1[y] = O[y]*PS[y];
9   end
10
11  max_val = max(a_t1);
12
13  for x = 1:S
14    a_t[x] = a_t1[x]/max_val;
15  end
```

3.1.2 Microarchitecture overview

The control unit is the block which manages the sequence of instructions to be executed. It handles as input a set of instructions from upper hierarchical levels which include the loading of parameters in the block as well as start/reset instructions. It generates control

signals for the execution unit and for the register files and status signals to be sent to upper levels to notify about the end of an instruction. It has not been implemented as hardwired but as a canonical microsequencing control unit, with a μsequencer and a command generator.
The execution unit handles the execution of instructions. It retrieves values from the register file and receives parameters as input and provides values to be stored in the register file, as well as the threshold value for the anomaly detection. It is made of a series of dedicated sub-blocks which optimize a set of operations:

- **MAC + Adder**: executes a multiplication and a sum in a single cycle, two multiplication and a sum in two cycles or a single sum in a clock cycle.
- **Divider**: carries on a floating point division
- **Max detector**: given a series of values it stores the maximum value received.
- **Counters**: provide the addressing for the register file block and control signals for the control unit.

3.1.3 Register files

The register file is made up of four main files and its size depends on the number of states: the T_{mat} file holds the values of the transition matrix and its size is $S \times S$, the PS one holds the partial sums and holds S values, the α holds the normalized values of the α values of timestep $t-1$ while α' holds and they both store S values.
The number of maximum register files available depends on the maximum number of states possible set at design time; if there are less states needed, given the number of states with the correct instruction during the setting time, clock-gates the unused registers so as to reduce power consumption.

3.1.4 Control unit

The architecture of the control unit is equal to the one described in Sec. 3.0.3.

Microsequencer

STATE	ENCODING	NEXT STATE	EOI	CC
IDLE	0001	IDLE	1	0
NF	0010	NF	0	1
F	0011	F	0	1
MF	0100	MFM	0	0
MFM	0101	MFM	0	1
MM	0110	M	0	0
M	0111	N	0	0
N	1000	NF	0	0
RST	0000	IDLE	1	0
LD CNT	1001	IDLE	1	0
LD TMAT	1010	IDLE	1	0
LD ALFA	1011	IDLE	1	0

The instruction flow follows the sequence of operations described in Sec. 3.1.1.

NF : Is the start instruction, once it is given the loop of operations begins and is stopped only in the case of a reset. The MA block executes the multiplication $\alpha_x T_{xy}$ and adds the partial sum PS contained in a register inside the execution unit. The max detector block normalizes the given α

F : The MA block executes the multiplication $\alpha_x T_{xy}$ and adds the partial sum PS contained in a register inside the execution unit

MF : The MA block executes the same operation of F and computes the multiplication $PS \times b(O_y)$.

MFM : Executes the same operation of MF and the MAX DETECTOR block checks for the maximum value of the α given

MM : The MA block executes the multiplication $PS \times b(O_y)$ and the MAX DETECTOR block checks for the maximum value of the α given.

M : the MAX DETECTOR block checks for the maximum value of the α given.

N : Normalization of the computed α values so as to prepare them for the next timestep

LD TMAT : This instruction must be provided by the user before the start instruction (NF). In the same clock cycle the instruction is given, the value of the transition matrix must be asserted on the DATA input port and the address of the value asserted on the ADDRESS input port.

LD ALFA : This instruction must be provided by the user before the start instruction (NF). It serves the purpose of specifying the prior probabilities and as such they must be set before the start of the loop. When asserted, the prior probability of the wanted state must be asserted on the DATA input port while the ADDRESS input port holds the number of the state interested.

LD CNT : This instruction must be provided by the user before the start instruction (NF) and is fundamental for the proper functioning of the system. Depending on the number of states different parameters must be set on the counters so that to activate the control signals with the right synchronization.

μsequencer selection control

The μsequencer has the same characteristic of the one described in Sec. 3.0.3.

3.1.5 Execution unit

The execution unit is made of the multiply-accumulate, the divider and the max-detector. The divider is explained in Sec. 3.4.3 while the first two are described in the following sections.

Multiply Accumulate

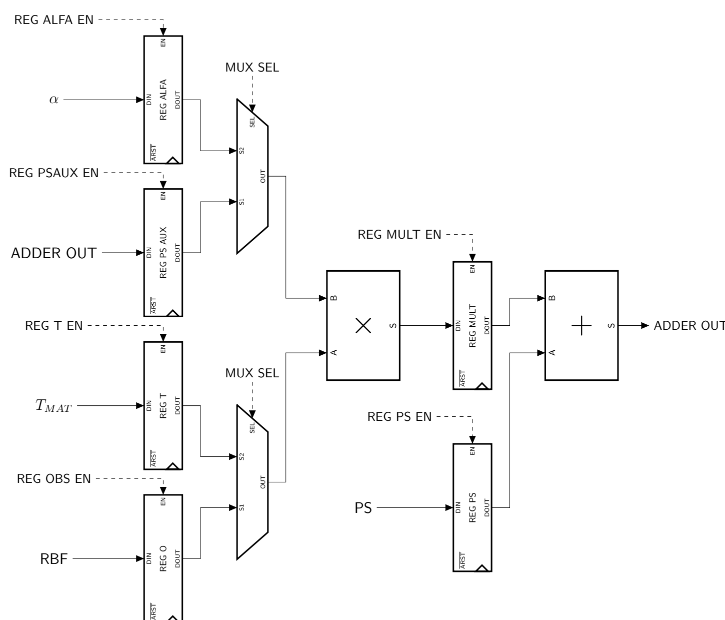


FIGURE 3.5: Multiply-accumulate datapath

The sub-block in the execution is able to do a multiply-accumulate, a multiplication, a sum or a multiply-accumulate and sum, depending on the control signals. When the

$\alpha_x T_{yx}$ products have to be computed the α_x value is stored in the REG ALFA and the T_{xy} is stored in REG T and the result in REG MULT. The $\alpha_x T_{yx}$ product is followed by its sum with the partial sum related to its α_x, as explained in the section 3.2.1.
When the product between the partial sums and the features joint probability O_y has to be computed the last partial sum is stored in PS AUX REG while O_y is stored in RBF REG.

STATE	CTRL SIGNALS				
	MUX SEL	LD 1	LD 2	OBS EN	PS AUX EN
IDLE	1	0	0	0	0
SF1	1	1	0	0	0
SF2	1	0	1	0	0
SF3	1	0	0	0	1

TABLE 3.1: Sequence of states for computing $\alpha_x T_{yx} + \text{PS}$

STATE	CTRL SIGNALS				
	MUX SEL	LD 1	LD 2	OBS EN	PS AUX EN
IDLE	1	0	0	0	0
SFM1	1	1	0	1	0
SFM2	1	0	1	0	0
SFM3	0	0	1	0	1

TABLE 3.2: Sequence of states for computing $\alpha_x T_{yx} + \text{PS}$ and the product PS × $P_(O_y)$

STATE	CTRL SIGNALS				
	MUX SEL	LD 1	LD 2	OBS EN	PS AUX EN
IDLE	1	0	0	0	0
SM1	0	0	0	1	0
SM2	0	0	1	0	0

TABLE 3.3: Sequence of states for computing the product PS × $P_(O_y)$

Max detector

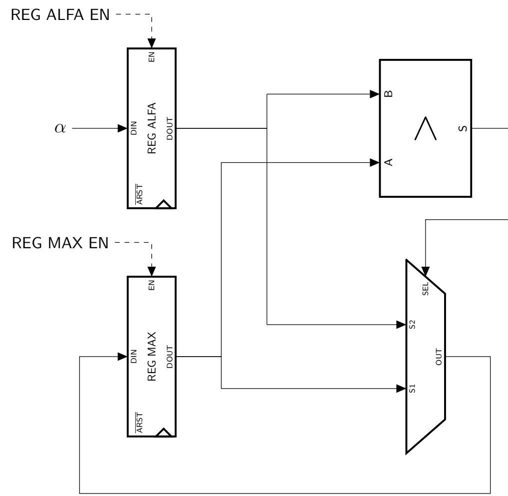


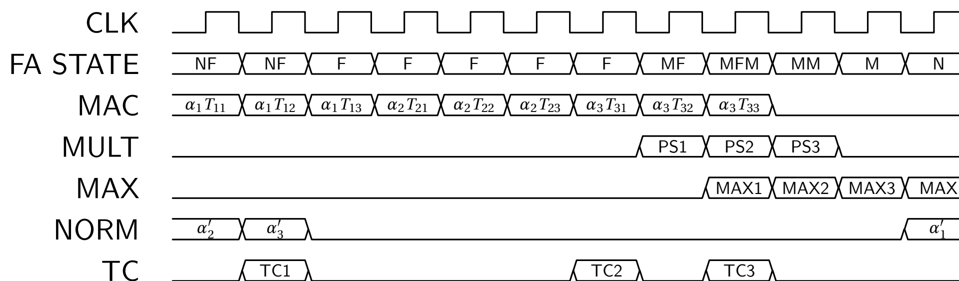
FIGURE 3.6: Max α detector datapath

The incoming α is compared to the value stored in the MAX register. If the value stored in the MAX register is lower, the MAX register is updated with the new value. Both the MAX register and ALPHA register are clock gated by means of the REG ALFA EN and REG MAX EN signals so as to reduce the power consumption. The control unit for this block is hardwired.

3.1.6 Register files

Partial values and parameters must be stored in register files within the microarchitecture. The transition matrix T values are stored in a $S \times S$ matrix, with S states: its values have to be stored before the FE command is issued with the LOAD TMAT command.
Since α_t values are normalized in separate steps and are not normalized as soon as they are ready, the α_t and α_{t-1} values have to be stored as well in S sized matrices.

3.1.7 Timing diagram



The scenario considered for this timing diagram is a system with 3 states ($S = 3$).
The number of clock cycles needed to carry on the computations must be minimized to save power. Given a certain input frequency, if the system is able to provide an output in less clock cycles is also able to operate at a lower frequency which means lower power consumption.
Because of this, whenever there were no risk of data hazards, parallelism of instructions was deployed.
By referring to the timing diagram, the first operation to be carried out are the $\alpha_x T_{xy}$ products while the normalization of the α's of the previous timestep is being done. A point of concern would be the risk of data hazard in such scenario: α values are used both for the normalization step and for calculating the partial products. However, considering the data stationarity of the α values there's no risk of such thing since the normalized α_1, computed in the last clock cycle of the previous cycle of operations is stationary enough cycles so as to let the system normalize the α_2 before is needed. The subsequent α's are no matter of concern.
After the normalization is done the execution unit only carries on the sums of the partial products $\alpha_x T_{xy}$ until the conditions are met as to start with the multiplication with O_y.
For example, if there are 3 states, in order to compute α'_1 of the current timestep, the partial products $\alpha_1 T_{11}$,$\alpha_2 T_{21}$ and $\alpha_3 T_{31}$ have to computed and summed and only after the last sum multiplied by O_1.
As soon as the first α_1 factor of the current timestep is ready it is sent to the max detector block; in the next cycle the next α_2 factor is ready and sent to the block and so on.
The normalization step begins as soon as all the α have been computed and the maximum value been found: in the final clock cycle of the task the normalization of the first α_1 is done so as to have the value ready for the next cycle of the algorithm.
The microprocessing unit carries on the operation in a loop unless a reset signal is detected by the control unit, in which case the loop is halted and all the registers which held parameters or termporal values are cleared.

3.2 Feature extraction

The feature extraction is the first sub-block of the system and its task is to extract the underlying features of the data, which have been selected to be the **variance**, the **average** and the **slope**. These features provide enough information to detect the three different type of anomalies, beside being informative enough to classify the regimes of the system.

3.2.1 Average extraction

The system does not deal with batched data but with online data and the correct definition of the feature is *moving average.*
A window of N values stores the previous N inputs and computes the average with respect to the last stored values. At each new input stored the window shifts in a FIFO manner and computes the moving average.

Its definition is similar to the average one

$$M.A. = \frac{\sum_{i=N}^{0} X_{t-i}}{N}$$

In order to compute the $M.A.$ the following steps are done in loop

1. Compute $PS_t = X_t + PS_{t-1} - X_{t-N}$, with $PS_{t-1} = \sum_{i=N}^{1} X_{t-i}$, which is the sum of the previous N inputs.

2. Divide by N, store the new value PS_t, shift the FIFO with the new value X_t

3.2.2 Variance extraction

Here as well we do not compute the variance of a batch of values but the *moving variance* of the incoming data.
A window of N values stores the previous N inputs and computes the variance with respect to the last stored values. At each new input stored the window shifts in a FIFO manner and computes the moving average.
Its definition is similar to the average one

$$M.V. = \frac{\sum_{i=N}^{0} X_{t-i}^2}{N} - M.A.^2$$

In order to compute the $M.V.$ the following steps are done in loop

1. After having done the power of two of X_t, $PS_t^2 = X_t^2 + PS_{t-1}^2 - X_{t-N}^2$ is computed, with $PS_{t-1}^2 = \sum_{i=N}^{1} X_{t-i}^2$ the sum of the previous N squared inputs.

2. Divide by N, store the new value PS_t^2, shift the FIFO with the new value X_t^2

3.2.3 Slope extraction

The slope extraction is done by subtracting the current input with the previous value $X_t' = X_t - X_{t-1}$. A register is needed to store the previous value and it has to be updated every time a new value is present at the input.

3.2.4 Datapath schematic

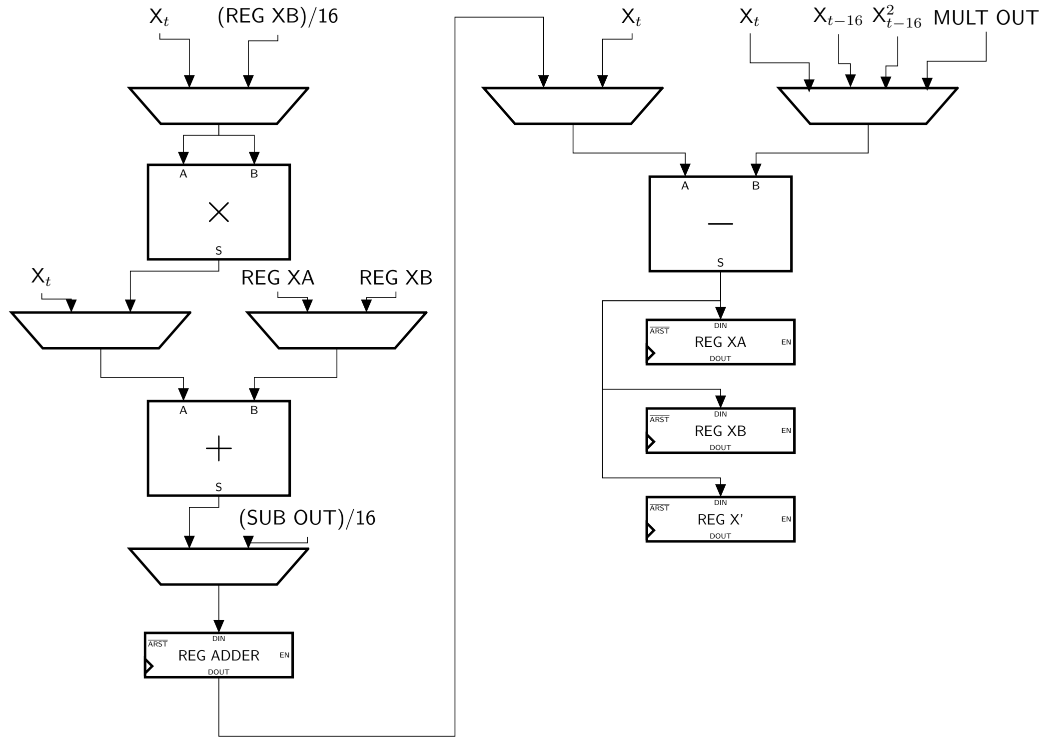
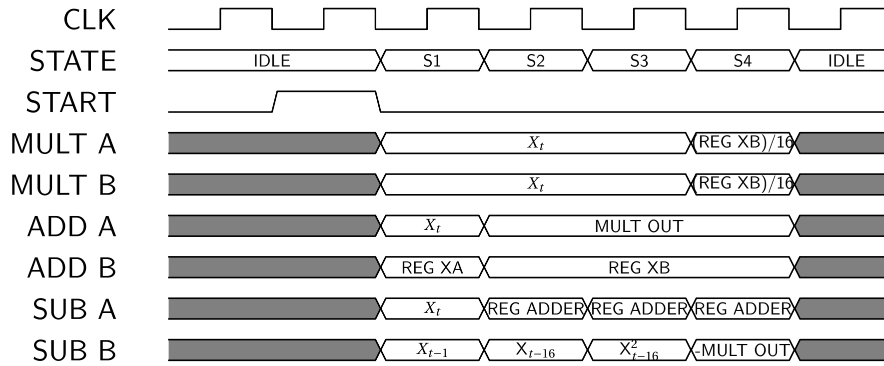


FIGURE 3.7: Feature extraction datapath

3.2.5 Timing diagram



The input of the current timestep is X_t. In the first clock cycle X_t^2 is computed by the multiplier, as well as the slope $X_t - X_{t-1}$; the adder adds X_t to the value stored in REG XA which corresponds to $PS_{t-1} = \sum_{i=16}^{1} X_{t-i}$. In the following clock cycle the adder adds

the previously computed X_t^2 to the value stored in REG XB which corresponds to $PS_{t-1}^2 = \sum_{i=16}^{1} X_{t-i}^2$, while the subtractor computes the difference between the output of the adder of the previous cycle, stored in REG ADDER, and the input that arrived 16 cycles before stored in the shift register; this corresponds to doing $(X_t + PS_{t-1} = \sum_{i=N}^{1} X_{t-i}) - X_{t-16}$. The result of this subtraction is PS_t. This value divided by 16 (an operation which consists of subtracting 4 from the exponent of PS_t) provides the mean feature μ_x. The same operation, but applied to the squared inputs, is carried on in the following cycle, so as to obtain μ_{x^2}. In the last cycle the mean μ_x value computed two clock cycles before is squared and subtracted from μ_{x^2} so as to compute the variance $\sigma^2 = \mu_{x^2} - \mu_x^2$.

3.3 Probability extraction

In order to provide the probabilities of the observations for the forward algorithm, the probabilities of the extracted features have to be computed.
As explained in Sec. 1.6 the features (from now on called observations) are assumed to be characterized by a certain Gaussian distribution, defined by a standard deviation and mean. The correct and most precise way to compute the probability of a given observation x would be to compute the normal distribution expression for the probability density in that point given the standard deviation σ and the mean μ.

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

To evaluate the expression the Taylor expansion would be needed and this would involve long and power consuming series of computations.
Since this is not a viable solution an alternative way has been implemented, less precise but estimated to be less power consuming.
The procedure involves the translation of the observed value to its Z-score and afterwards the evaluation of the standard normal distribution as a function of the Z-score.
The Z-score is the value assigned to the single observation which indicates how many standard deviations from the mean of the probabilistic Gaussian distribution the value of the observation under analysis is.
The standard normal distribution is the special case of the Gaussian distribution in which the mean is equal to zero ($\mu = 0$) and the variance is equal to 1 ($\sigma = 1$), expressed as

$$f(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}z^2}$$

in which z is the Z-score, computed as

$$z = \frac{x-\mu}{\sigma}$$

In this case, since the values of the standard normal distribution are expressed as a function of a single variable and since we are interested in an interval of its values, the

extraction of the probabilites is based on a combinatorial block, in which the probabilities are stored. Given a certain Z-score it will be a matter of translating its value to the corresponding probability.
The resolution of the output values and the interval in which the values are translated are fundamental parameters that determine the size, and consequently the power consumption, of the block.

3.3.1 Z-score evaluation

To evaluate the value of the Z-score the following steps are taken

1. Compute the difference between $x - \mu$, then multiply per $1/\sigma$. The division operation would have been a waste of time and power consumption; multiplying the difference times the inverse of the standard deviation, which is known and stored in a register file, is faster and more straightforward.

2. Compare the result of the multiplication with the maximum value accepted by the combinatorial block, which has been set at 16. If the result exceeds the maximum value known, the maximum value will be sent at the output instead of the result obtained. This is done to avoid having unknown inputs to the combinatorial block.

Given the simplicity of the block the control unit is hardwired, consisting of 2 states

Datapath and control unit

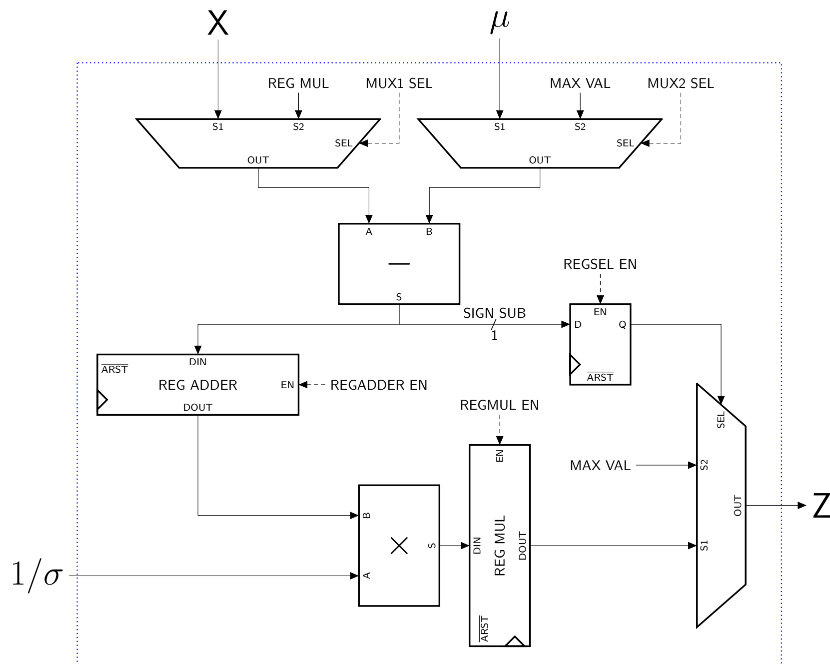


FIGURE 3.8: Z-score computation datapath

STATE	CTRL SIGNALS			
	MUX SEL	D ADDER EN	R ADDER EN	R MULT EN
IDLE	0	0	0	0
S1	1	1	0	0
S2	0	0	0	1
S3	0	0	1	0

TABLE 3.4: Z-score evaluation sequence of states

3.3.2 Probability extraction

The Z-score evaluated in the previous cycle is evaluated in the standard probability extraction sub-block. In order to precisely evaluate the result it would be needed to compute the value of the normal guassian distribution as

$$P(z) = \frac{1}{\sqrt{2\pi}} e^{\frac{z^2}{2}}$$

which involves long and power consuming operations. An easier, faster and less power consuming implementation is a combinatorial block which, given a certain input corresponding to the Z-score, evaluates the probability.
The most straightforward solution would be to realize this element by considering all possible combination at the inputs and synthesize the combinatorial block with a case-select structure. A floating point representation with 16 mantissa bits and 10 exponent bits would require many millions input combinations and the realization effort, let alone the power consumption due to leakage, would be unfeasable. Therefore the input space must be quantized so as to consider less values and reduce the input combinations. This procedure comes at a cost: quantization introduces an error on the accuracy of the output which may cause evaluation errors in the next steps of the algorithm. To overcome this problem a first order interpolation is computed on the extracted probability.
The Z-score interval considered is between 0 and 16. Beyond $z = 16$ the probability is saturated at $P(16) = 1.0262 \times 10^{-56}$.
The interval between 0 and 16 has to be quantized so as to guarantee enough accuracy at the output and at the same time do not generate numerous input combinations.
Quantizing the input interval with a quantization step of 2^{-8} guaranteed very high accuracy of the output but 1280 input combinations, which caused elevated power consumption for each retrieval operation (around 60 μW at 1 MHz). On the other hand quantizing the input with a quantization step equal to 2^{-6} reduced the input space to 320 combinations but heavily reduced the accuracy.

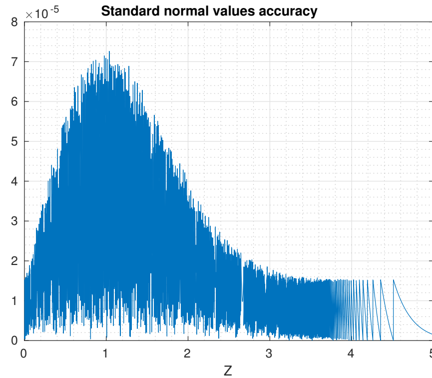


FIGURE 3.9: 2^{-8} step quantization relative error

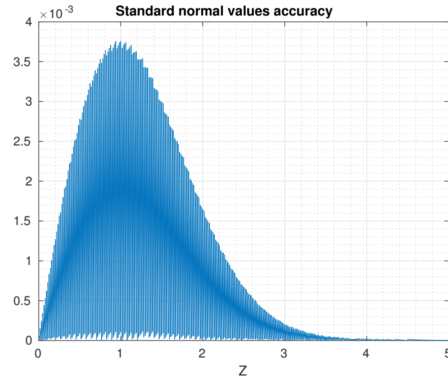


FIGURE 3.10: 2^{-6} step quantization relative error

The compromise which led to this solution was to divide the interval in a series of sub-intervals with different quantization steps for each interval.
The tail of the distribution represents less probable values which may not impact the outcome: if an observation is very improbable in a certain distribution then it will probably minimize the relative α factor in the forward algorithm step: in the regime classification step, when the maximum α is sought after, the relative α will get discarded. Therefore the quantization step in the tail (between $z = 8$ and $z = 16$) can be set rather large (2^{-2}). Moreover at the tail of the gaussian between one quantization step and the next one the probability value gets exponentially smaller and since a floating point representation is deployed there's no risk of underflow with this large steps.
The quantization step is gradually decreased across different intervals until $z = 2^{-5}$. Between this point and 2^{-8} a quantization step of 2^{-11} is used. The probability distribution has higher values, which means that it is more probable to observe values in this interval, and a higher accuracy is needed. Decreasing even more the quantization step is useless: the slope of the distribution in this interval is less than 2^{-16}, which is the lowest resolution of the floating point: a lower quantization step would provide identical outcomes for many input combination and would correspond to a waste of area and power in the combinatorial logic.
Between 2^{-8} and 0 the slope is even more flat and a quantization step of 2^{-11} is too large and would provide identical results in a floating point representation with 16 bit resolution: the quantization step in this last interval was set to 2^{-10}.

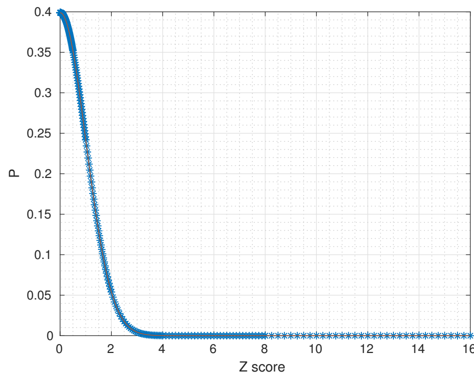


FIGURE 3.11: LUT output characteristic

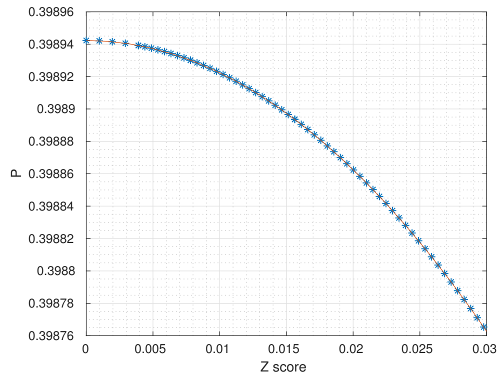


FIGURE 3.12: LUT output characteristic zoom

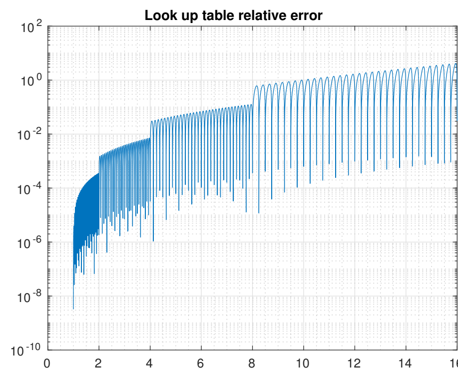


FIGURE 3.13: Relative error of the final implementation

The figure shows how in the interval quantized with 2^{-6} quantization steps the error reaches 450% relative error while for smaller quantization steps the error becomes negligible, in the order of 10^{-7}.

Datapath

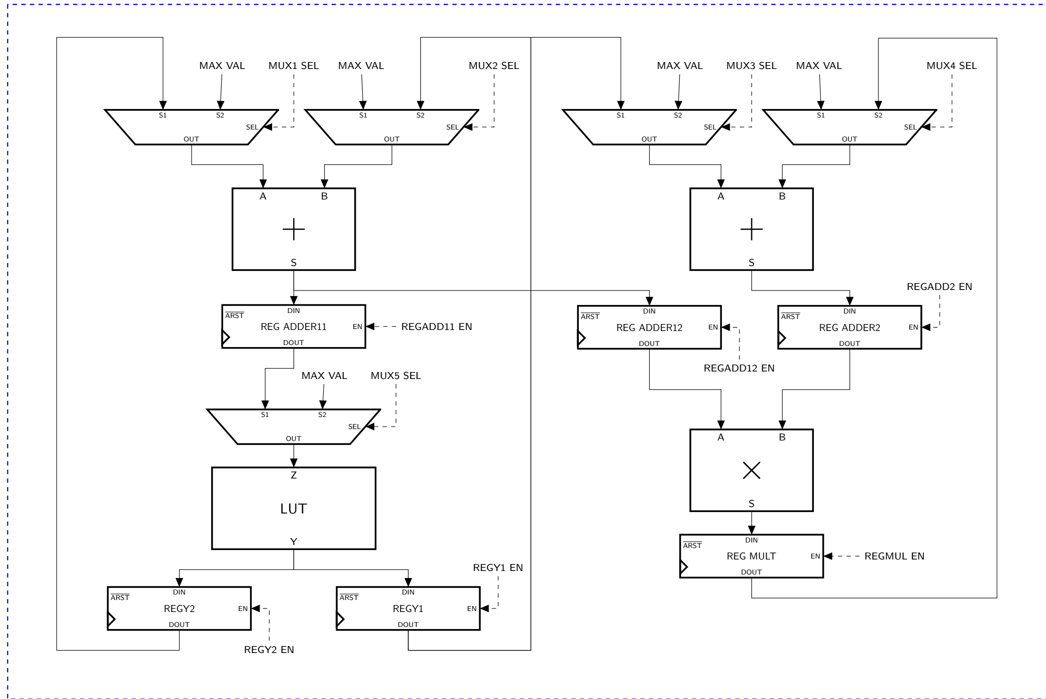


FIGURE 3.14: Probability extraction datapath

STATE	CTRL SIGNALS							
	ADD2 EN	Y1 EN	Y2 EN	ADD11 EN	ADD12 EN	MUL EN	MUX1 SEL	MUX2 SEL	MUX5 SEL
IDLE	0	0	0	0	0	0	0	0	0
S1	0	1	0	1	0	0	1	0	1
S2	0	0	1	0	0	0	1	1	0
S3	1	0	0	0	1	0	0	1	0
S4	0	0	0	0	0	1	0	1	0

TABLE 3.5: Probability extraction module sequence of states

3.4 Floating point operation

In order to represent numerical values two representations are available: fixed point and floating point.
In the fixed point notation the bits used to represent the value possess a fixed value depending on their position: according to the specification proposed by the designer, there exist a fixed number of fractional bits and a fixed number of integer bits.
Given a $Qa.b$ representation, the step between adjacent numbers will always be constant

and equal to the step (2^{-b}). The range of the values which can be represented is determined by the number of bits deployed and is **fixed**. In the signed case the range is equal to $-2^a \leq Qa.b \leq 2^a - 2^{-b}$.
On the other hand the floating point representation is characterized by a *dynamic range*. In this notation the position of the decimal point *varies* and this enables the representation of very large numbers as well as very small ones. While the fixed point representation suffers from the trade-off between range and precision in the sense that high precision can be achieved at the expense of small range, floating point offers both a wide dynamic and at the same time acceptable precision.

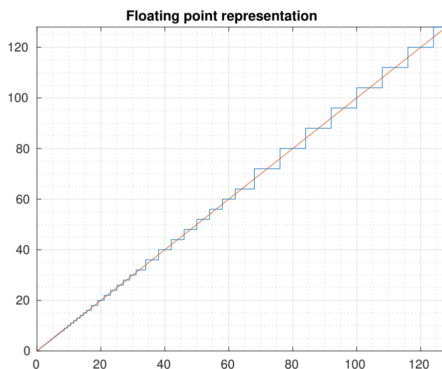


FIGURE 3.15: Floating point representation with 4-bit mantissa

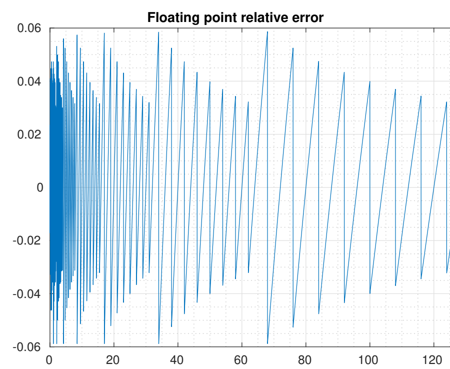


FIGURE 3.16: Floating point relative error with 4-bit mantissa

Fig. 3.15 and Fig. 3.16 show how a floating point notation with 4 mantissa bits represents the values in the 0-128 interval. The greater the value of the number the greater the absolute error: this is due to the fact that in each power of 2 interval the absolute error corresponds to the value of the least significant bit LSB (2^{-4} in this case) times the power of 2 interval the value belongs to. The maximum relative error however stays the same since it only depends on what is the value of the LSB, which is $2^{-4} = 0.0625$ in the case considered.
The floating point notation is defined as

S	*M* MANTISSA BITS	*E* EXPONENT BITS

$$N = (-1)^S \times M \times 2^E$$

When it comes to choosing which representation to implement, the fixed point notation is much more advantageous with respect to floating point. If we have a fixed, known range of values and a certain required precision the former is much easier to implement, is less power consuming and is considerably faster.

The reason why the floating point notation has been chosen over the fixed point is that the system must be able to handle a very wide range of values be it the incoming data or the probabilistic values.
The feature extraction block, the first one the incoming data is sent to, must be able to extract the indicated features with a proper precision in order for the next block to be able to evaluate the related probabilities.
The fact that the range is application specific, it is unknown during the design phase of the system. In order to be safe, so as to have a wide range of acceptable values and a certain minimum precision, floating point notation appears to be more suited for the application.
The successive blocks after the feature extraction all deal with probabilistic values. Given that the algorithm chosen requires to cover a very wide range of values (down to 10^{-100} or even more) and given the fact that, as it will be described later on, in order to extract the normalized gaussian probabilities we will need to have a certain resolution, floating point is essential to carry on the computations.
The main disadvantages of the floating point become rather negligible for our purpose:

- **Complex implementation** Given the topic of the thesis and the time given, floating point operation blocks were implemented with success

- **Speed** The system must be low-power and therefore no high operating frequencies are needed. The setup times are respected by a large margin

- **Power consumption** The two most important and frequent operations in the algorithm are addition and multiplication. Provided that values down to 10^{-100} have to be represented, more than 350 bits would be needed in fixed point representation. The power consumption of an array multiplier in fixed point would then be even more than the power consumption of the floating point multiplier.
  Beside that, storing 300+ bits is extremely power consuming, while storing around 25-30 bits in floating point, even though still quite power consuming, is orders of magnitude inferior.

3.4.1 Floating point adder

The floating point adder is the most complex floating point operation to be implemented. The sequence of operations to be made are the following:

1. **Unpacking**: The incoming values have to be separated in their main components: the sign s, the mantissa m and the exponent e. An addition between a and b is therefore represented as

$$(\pm m_a \times 2^{e_a}) + (\pm m_b \times 2^{e_b}) = \pm m \times 2^e$$

2. **Pre-shift**: The bigger of the exponents has to be computed (between e_a and e_b). Once computed the bigger value will be denoted by g and the smaller by s. Therefore $e_g \geq e_s$. In order to carry on with the addition, the two mantissas have to be aligned.

The difference between e_g and e_s will be the number of bits the mantissa m_g will have to be shifted right.

$$\pm m_s \times 2^{e_s} = \frac{\pm m_s}{2^{e_g - e_s}} \times 2^{e_g}$$

3. **Mantissa addition**: Once the preshifting step is done in order to compute the mantissa magnitude the two mantissa have to be added taking into account the sign of the addends. Proper care must be taken to avoid and detect overflow. The addition is performed as follows

$$(\pm s_g \times 2^{e_g}) + (\frac{\pm m_s}{2^{e_g - e_s}} \times 2^{e_g}) = (\pm m_g \pm \frac{\pm m_s}{2^{e_g - e_s}}) \times 2^{e_g} = \pm m \times 2^{e}$$

4. **Post-shift**: Once the mantissa is computed a normalization step is needed so as to be able to represent the value in floating point notation: in order to go back to floating point the module of the mantissa is first computed and then it is shifted so as to have the MSB equal to 1. After the normalization the exponent has to be corrected as well, by adding to it the number of bits the mantissa had been shifted in the normalization step.

5. **Sign selection**: If opposite signs are added we may observe sign flip, and this has to be detected.

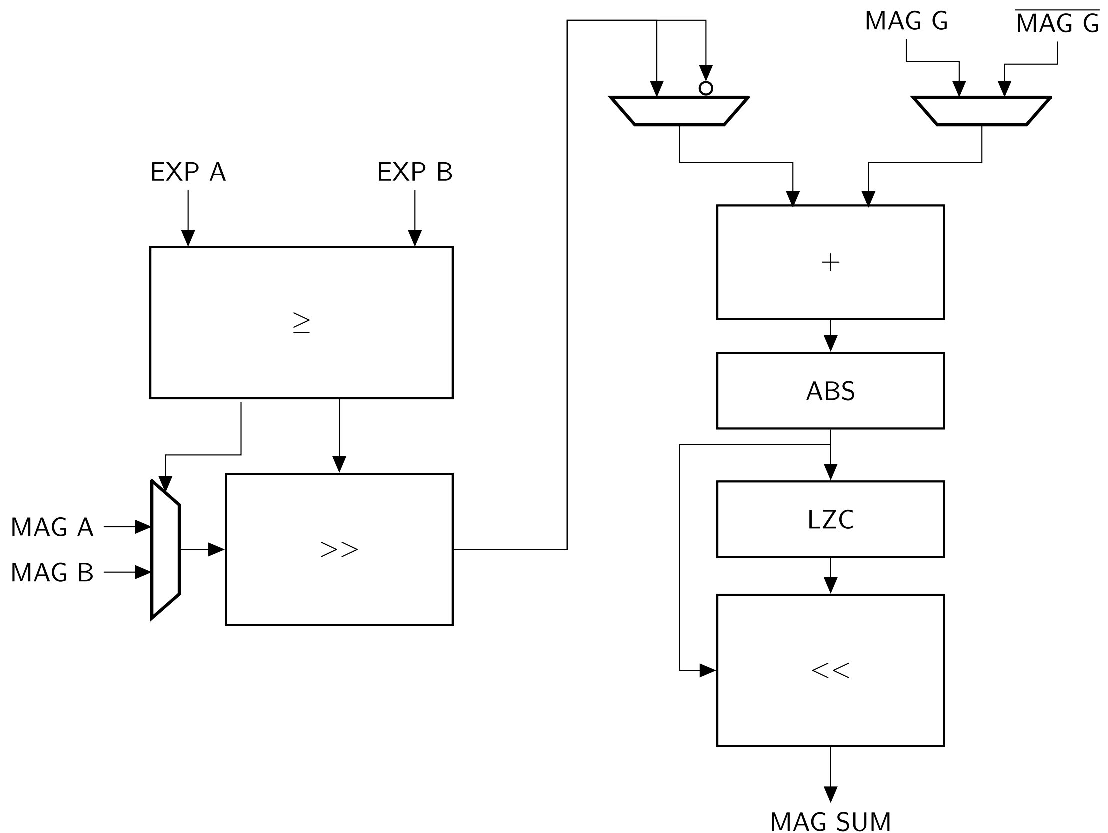


FIGURE 3.17: Floating point adder datapath

Fig. 3.17 depicts the RTL blocks for computing the mantissa of the result between value *a* and *b*. In the first place the two exponents of the addends have to be compared so as to find the smallest addend. The smaller addend is selected depending on the sign of the comparison and shifted right by the difference between the two exponents EXP A - EXP B so as to align the two mantissas. Once this is done the two mantissa are summed/subtracted together (MAG G represents the mantissa of the greatest between the two mantissa). The values which are summed together are their normal or their inverse depending on whether a subtraction or a sum has to take place. Since a subtraction might flip the sign of the mantissa (which is computed by a standard adder) after the sum the absolute value has to be computed. The absolute value of the sum/subtraction is subsequently re-normalized to a floating point representation of the mantissa which requires the MSB to be equal to 1: because of this the obtained value is shifted to the left by the number of bits detected by the leading zero counter block.
In order to have a result correct to the LSB a rounding block is needed; however in this implementation it has been avoided. Given that perfect precision is not a priority in the chosen algorithm, in order to limit the power consumption the rounding block has not

been realized. Given that the infinity and not-a-number (NaN) would never happen under normal operating conditions, they were not implemented in the design.

Adder typology selection

Various adder architectures are described in the literature, each of which presents advantages and disadvantages in terms of delay, power consumption and area. For our purpose, the area and the delay factors are not prioritized, while the power consumption is the determining factor.
Different architectures have been analyzed (the ripple carry adder, the carry look-ahead, the carry-save and the carry skip adder) and the conclusions that have been reached are similar to those exposed by [Ramadass et al., 2012].
The ripple carry adder (RCA) presents the best performances in terms of power consumption, even though is a bit slower than the other architectures.
The power estimations contained in the following table for different 8-bit adder topologies have been tested using 120 nm technology, 1.2 V voltage suppy and temperature 27 °C.

Adder Topology	Power dissipation [nW]
Ripple carry adder	0.206
Carry save adder	1.082
Carry look-ahead adder	0.312
Carry select adder	1.109
Carry bypass adder	0.459

TABLE 3.6: Different 8-bit adder topologies power consumption at 500 kHz

Shifter implementation

The shifter block handles the pre-shift and the post-shift of the mantissa part. It can be implemented with a barrel shifter or alternatively with a sequence of multiplexer. The barrel shifter solution was the less efficient in terms of power consumption and because of that it was disregarded.
The multiplexer solution involves a sequence of multiplexers which each shift the input by a power of two amount of bits depending on the position in the sequence. The shifting at the n-th multiplexer is done by assigning to one input the non-shifted value and at the other input the shifted by 2^n bits value
The first multiplexer shifts the input by 2^0 (=1) bits, the second one by 2^1 bits and so on. Given the length of the mantissa M there will be $\log_2(M)$ multiplexer stages needed.

Pipelined implementation

The structure involves many combinatorial blocks interconnected between each other. In order to reduce the dynamic power consumption caused by the glitches that propagate in the combinatorial structure, pipelining can be done. By inserting register banks in the middle of the combinatorial block the structure is cut in separate parts, and the glitching

does not propagate beyond the registers. Theoretically, in an extreme limit case, we could insert registers after every single logic gate, so as to cancel any glitching activity.
A pipelined structure was realized and what was observed was an actual increase of the dynamic power consumption: the flip flop cells used to realize the pipelining consumed more power than what it was saved by reducing the glitching activity. Therefore the pipelined structure was discarded.

Leading zero counter implementation

Given an input $A[N-1:0]$ the number of leading zeros is computed as following

1. **Find first 1 position**: To find the position of the first 1 the S values are computed as

$$S[i] = \sum_{i}^{N-1} A[i]$$

   The S vector obtained will be a sequence of zeros up until the first 1 position and after that it will be a sequence of ones, such as

$$000001001110110 \quad \rightarrow \quad 00000111111111$$

2. **Isolate first 1**: Once the position of the first zero is found we isolate it so as to have a one-hot encoding of the first zero position. The one hot encoded vector L is computed as follows

$$L[i] = S[i] \oplus S[i+1]$$

   The S vector is transformed to the L vector

$$00000111111111 \quad \rightarrow \quad 0000100000000$$

3. **One-hot decoding**: The L vector is decoded to the unsigned output value by means of a decoder.

3.4.2 Floating point multiplier

The floating point multiplier has a more compact architecure compared to the adder.
The mantissas of the input values are multiplied by means of a multiplier and subsequently normalized, with a procedure similar to the one of the adder.
The exponents are summed and later adjusted depending on the result of the multiplication of the mantissas. Depending on the position of the leading zero of the mantissa, an adjusting factor has to be added to the already computed sum of exponents.
The sign of the result is easily computed as the XOR of the signs of the inputs.
Given that the LSB precision is not a hard constraint, the rounding block has been omitted to save on power consumption.

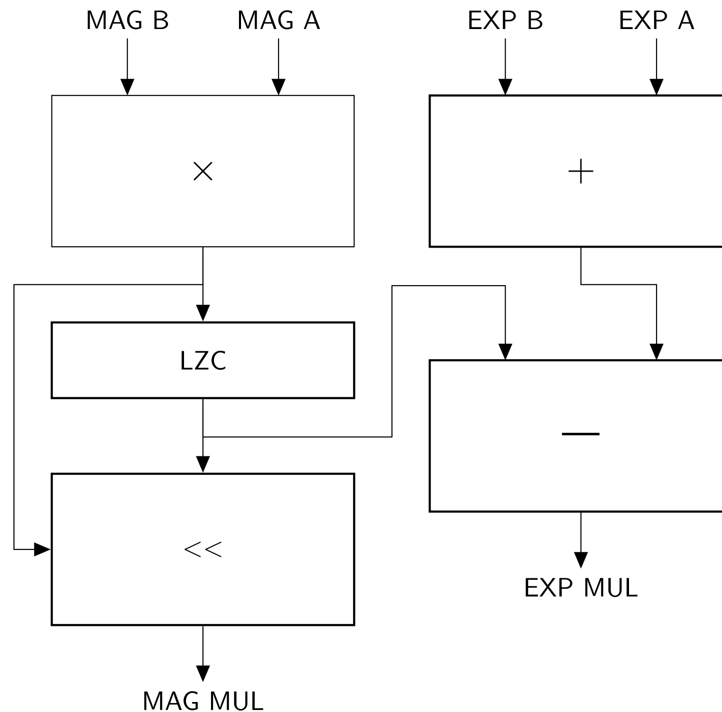


FIGURE 3.18: Floating point multiplier datapath

Multiplier architectures

The multiplier block which operates on the mantissas is the most power consuming element and a proper architecture has to be chosen so as to minimize its power.
Two architectures have been analyzed: the array multiplier and the Dadda multiplier.

Array multiplier

Given two values A and B, represented respectively with m and n bits, the result of the multiplication is represented as a sum of partial products

$$P(m+n) = A(m)B(n) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} a_i b_j 2^{i+j}$$

The computation of this value implies $(m-1) \times n$ adders ordered in an array, which, unless pipelining is deployed, causes a non-negligible amount of glitching in the structure. The glitching generated heavily affects the dynamic power consumption. Because of this the structure has not been adopted.

Dadda multiplier

The Dadda multiplier belongs to the tree multiplier category and minimizes the number of partial products necessary to carry on the computation. It is faster than the array multiplier and requires far less gates, hence the reduced power consumption.

3.4.3 Floating point divider

The floating point division is represented as

$$\pm m_z \times 2^{e_z} \div \pm m_d \times 2^{e_d} = \pm m_q \times 2^{e_q}$$

where the z indices refer to the divisor,the d indices to the dividend and the q indices to the quotient.
The floating point divider inherently needs multiple number of cycles to carry on with the computation.
The scheme adopted in the implementation in the digit-recurrence one, in which one digit of the quotient is computed every cycle, starting from the MSB. The number of cycles needed to calculate the result is therefore equal to the number of bits in the mantissa.

Quotient mantissa computation

The mantissa of the result is calculated as a sequence of cycles in which the divisor is compared each time to the partial reminder plus the dividend shifted by one bit. The sequence of operations is the following:

1. In the first step the mantissa of the divisor m_z is subtracted from the mantissa of the dividend m_d and provides the first reminder $m_z - m_d = m_r$.
   If the reminder is a positive value then the first digit of the quotient will be 1. This means that $m_z - m_d \times 2^0 \geq 0$ and therefore $m_d \times 2^0$ is contained in m_z. If it is instead less than zero the dividend must be divided at least by two (therefore shifted at least by one digit) in order to be contained in m_z. The MSB of the quotient will in this scenario be equal to 0.
   In the first case the partial remainder m_{pr} will be stored in a register; in the latter case the partial remainder will not be updated and will stay at zero.

2. In the following step the eventual partial remainder is added to the dividend shifted by one digit and the difference with respect to m_z is computed again. In a similar fashion if the result of the subtraction is a positive value then $m_z - (m_d \times 2^{-1} + m_{pr}) \geq 0$ and the following digit (MSB - 1) will be equal to 1. In this case the partial remainder will be stored for the next cycle. Otherwise a zero will be assigned to the (MSB - 1) digit of the quotient and the partial reminder will not be updated.

3. The subsequent cycles follow the same procedure and the values are assigned to the corresponding digit of the quotient. After N cycles the mantissa of the quotient will be calculated

4. If the MSB digit of the mantissa of the quotient is zero then the obtained value will have to be shifted by one position in order to be compliant with the floating point representation which requires a 1 at the MSB digit. If this situation occurs then the exponent will have to be updated by subtracting one to its final value.

Quotient exponent and sign computation

The exponent of the quotient is the result of the difference of exponents between the divisor and the dividend $e_z - e_d = e_q$. The quotient exponent has to be adjusted by subtracting one in the case the MSB bit of the mantissa of the quotient m_q is equal to zero. The sign of the quotient is equal the sign of the xor of the dividend and the divisor.
The pseudo-code that describes the behavior of the divider is the following

```
1 for i = 0:1:N-1
2    if ((res_aux+res_aux2) <= Q_Ar)
3       res_final = res_final + 2^(-i);
4       res_aux2 = res_aux2 + res_aux;
5    end
6    Q_B = [0 Q_B([1:N-1])];
7 end
```

Chapter 4

Performance analysis

4.1 Simulations results

The A4Benchmark-TS44 of the Numenta benchmark dataset collection [Lavin and Ahmad, 2015] for anomaly detection is considered, in which 3 states are distinguishable, as is shown in Fig. 4.1. The anomalies are indicated by the red star; in this dataset point and contextual anomalies are present.
The model is trained by

1. Separating the time series in its three states and cleaning out its anomalies

2. Extracting the features (moving average, moving variance, slope) out of the three states identified

3. Fitting the Gaussian distribution relative to each feature in each state, so as to have a mean and a standard deviation for each feature and each state.

4. Computing the transition probabilities from one state to the other. Since the dataset was not lengthy enough so as to have an accurate representation of the transition, it was assumed that the probability of transitioning from one state to another was 0.1% and the probability of staying in the same state in the next time step equal to 99.9%.

The obtained values are loaded in the system by issuing the LOAD MU, LOAD STD and LOAD TMAT as explained in section Sec. 3.0.3 and Sec. 3.1.4. The priori probability is assumed to be equal for all the states.

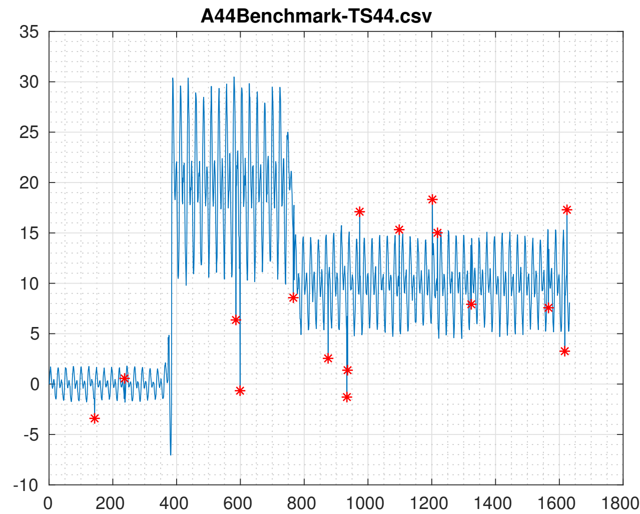


FIGURE 4.1: Time series simulated

4.1.1 Probability extraction

The probabilities of features extracted are represented in Fig. 4.2 for each state. As it can be seen the synthesized probabilities are saturated at 1.0262e-56, which corresponds the lowest value the LUT is able to generate. The syntesized probabilities do not correspond exactly to the modeled ones using Matlab, but this does not affect the anomaly detection. As described in the section Sec. 2.1 where the Naive Bayes classifier is explained, an accurate probability extraction is not of vital importance given that during the classification step the maximal probable state is classified as such, regardless of its accuracy. Nevertheless this does not justify a large loss of accuracy since misclassification can occurr if the error is so large as to affect the next steps of the algorithm.

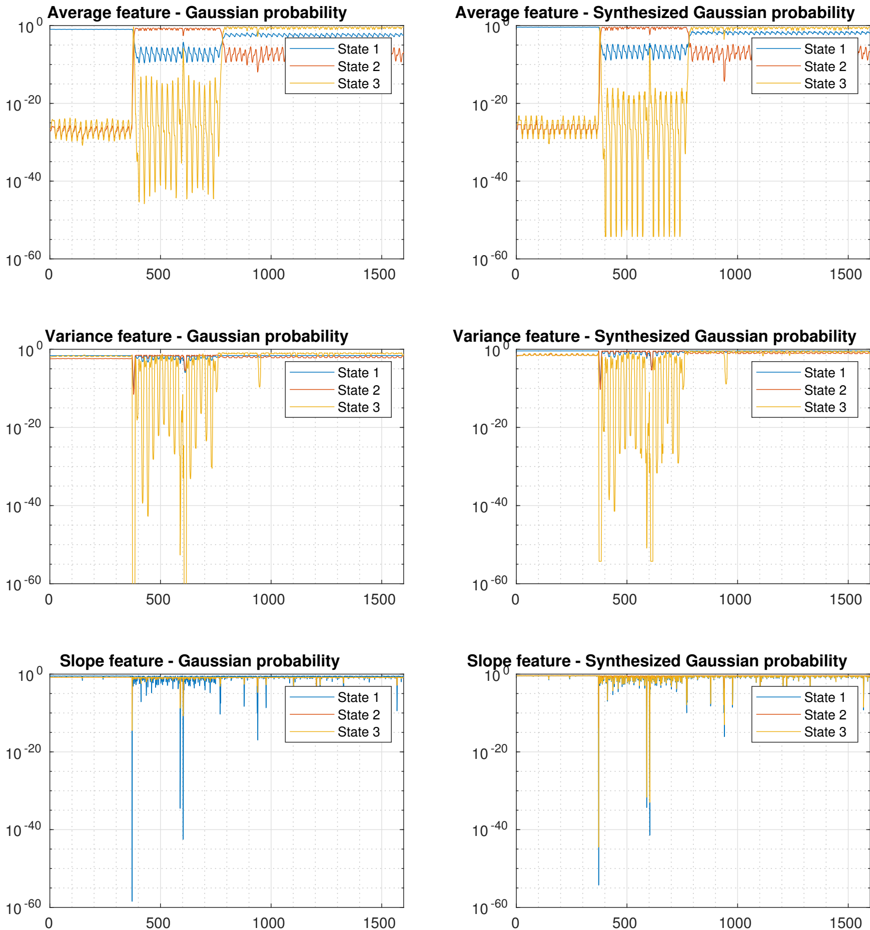


FIGURE 4.2: Feature probability extraction

The joint probability of the features, which corresponds to the $b(O)$ factors, are the ones depicted in Fig. 4.3, in which the ones obtained with the Matlab simulation are compared to the ones obtained with the synthesized model. In the Matlab simulations the joint probability may happen to be a not-a-number for some intervals. This is due to the fact that the probability values in those intervals overflow and they go beyond the range of values representable in the IEEE-754 standard for floating points. In the synthesized model however this is not observed since, as explained in the Sec. 3.3.2, the smallest value represented is 1.0262e-56, which therefore limits the accuracy but mantains the probability values in the representable range.
By adopting a simple Naive Bayes Classifier and not a DNBC the anomaly detection would

stop here. Given that the distributions in this example are distanced from each other, the contextual anomalies are detected even though the forward algorithm is not applied yet. In the case the state probabilistic characteristics were very similar, a Naive Bayes classifier alone would not be able to differentiate between the two with enough precision. By applying the forward algorithm and therefore introducing the α and T_{ij} values, the former scenario would be solved.

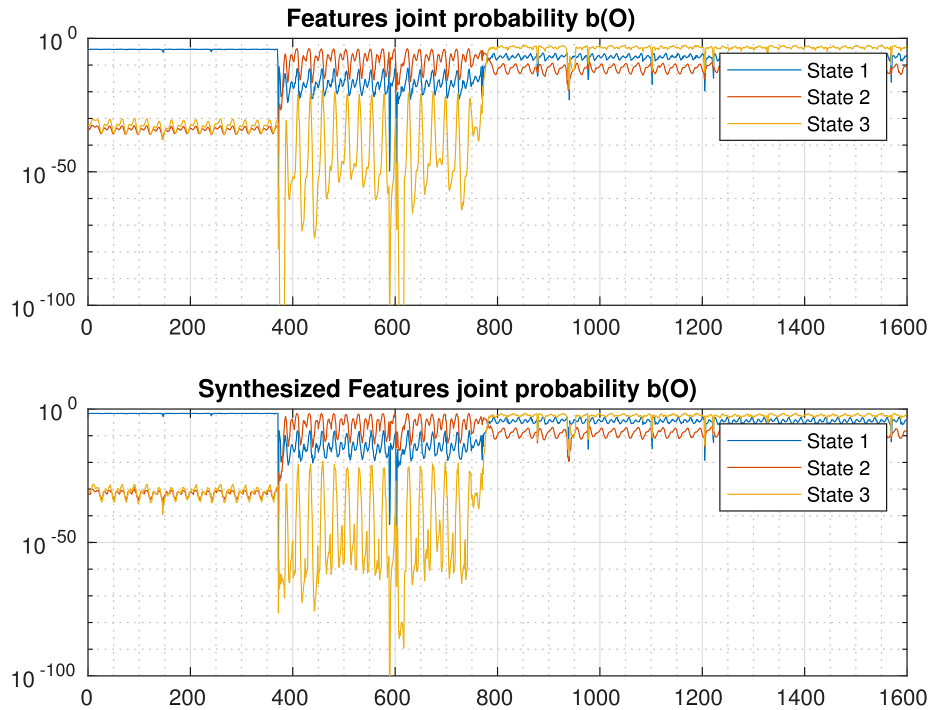


FIGURE 4.3: Joint feature probability

4.1.2 Alfa factors

The α values are depicted in Fig. 4.4. The accuracy of the synthesized model is worse with respect to the ones modeled with Matlab, due to the fact that

1. Matlab uses 21 bits for the mantissa to carry on its computations, while the floating point notation synthesized only uses 16 bits.
2. Matlab computes the Gaussian probability by doing all the calculations, while the synthesized model resorts to the LUT, with its related inaccuracies shown in Sec. 3.3.2.

These differences do not affect the outcome of the classification in this simulation. The classification of the regimes is based upon the α factor with the highest probability per

each timestep, which defines the state of the system. In the three regimes the most probable states are correctly classified in both models and the peaks related to the anomalies are detectable by examining the profile of the curves.

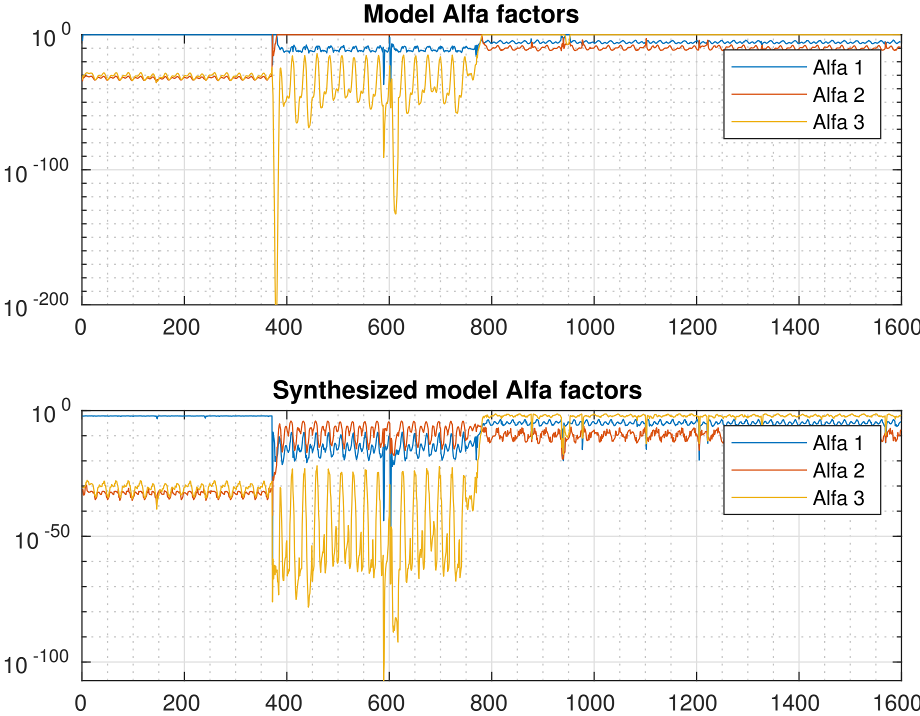


FIGURE 4.4: α factors for the 3 states

4.1.3 Threshold check

The probability of the most probable state is compared to its related threshold and an anomaly is detected if the probability falls below the given threshold. Fig. 4.5 represents the output of the forward algorithm and the related threshold, with the expected and the detected anomalies.

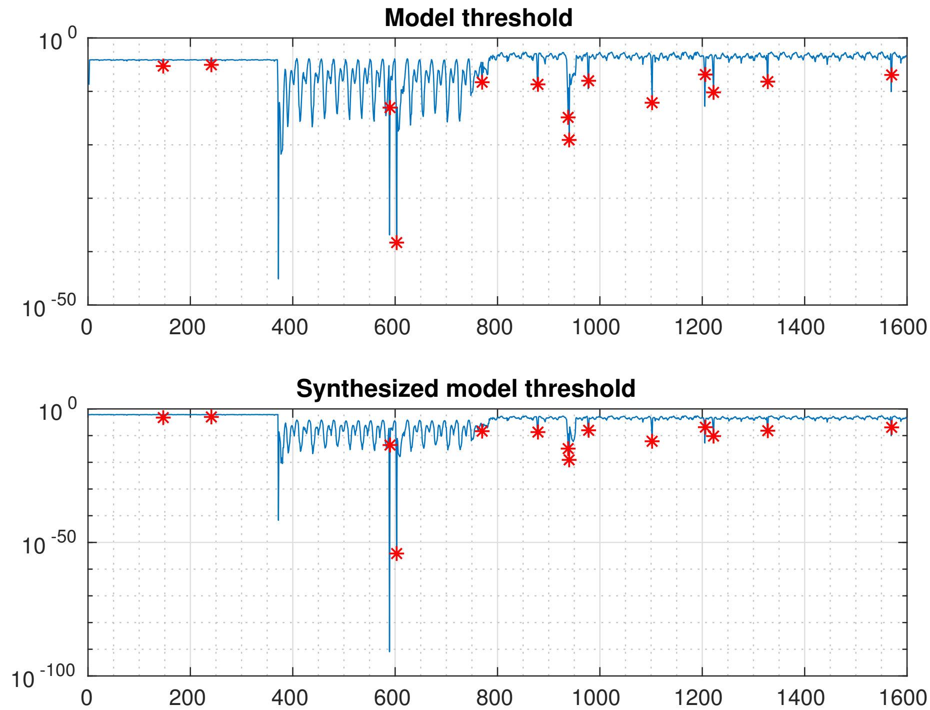


FIGURE 4.5: Theshold check

4.2 Power consumption estimation

The design is synthesized with 90nm UMC cells and the operating conditions in which the power estimations have been made are 25°C temperature and 1 V voltage supply. The sub-blocks are evaluated in test conditions, so as to mimic the power consumption in a real-case scenario. The input

4.2.1 Floating point adder

Frequency [Hz]	Leakage Power [nW]	Dynamic Power [nW]	Energy/op
500 kHz	3.74	2299.5	4.4 pJ/op
1 MHz	3.74	4492.5	

TABLE 4.1: Floating point adder power consumption estimation

The adder tested used a floating representation with 16 mantissa bits and 12 exponent bits. Dynamic power is in all cases the dominant power consumption component. From

the estimations, the block which consumes most power are the adders of the mantissa and the exponent, followed by the barrel shifters, which consume 1200 nW each at 1 MHz.

4.2.2 Floating point multiplier

Frequency [Hz]	Leakage Power [nW]	Dynamic Power [nW]	Energy/op
500 kHz	11.72	3478.6	6.9 pJ/op
1 MHz	11.72	6932.9	

TABLE 4.2: Floating point multiplier power consumption estimation

The multiplier implemented used a floating point representation with 16 mantissa bits and 12 The main contribution of the power consumption comes from the multiplier, which accounts for 4174 nW. During design phase a low power multiplier (the Dadda multiplier) was preferred over a standard array multiplier, which would have caused a 50% increase in the power consumption, since it consumed around 6000 nW in the same conditions. The barrel shifters used for normalizing the mantissa are the second most power consuming block in the architecture, as they account for 1270 nW at 1 MHz. The high leakage power consumption is mainly due to the multiplier, which uses 1169 cells.

4.2.3 Floating point divider

Frequency [Hz]	Leakage Power [nW]	Dynamic Power [nW]	Energy/op
500 kHz	3.63	450	1.1 pJ/op
1 MHz	3.63	1001.5	

TABLE 4.3: Floating point divider power consumption estimation

At 1 MHz the floating point divider implemented with 16 bits for the mantissa and 10 for the exponent consumes 1005 nW, of which 1001 nW from the dynamic power contribution. This value may seem low but it has to be taken into account that the divider takes M cycles, with M the size of the mantissa, to compute the result and therefore, unless pipelining is implemented to guarantee the same throughput, the clock cycle must be 16 times higher to have a result in the same period of time the other two operators provide a value. At 16 MHz, based on the estimation, the component consumes 17 μW.

4.2.4 Feature extraction

Frequency [Hz]	Leakage Power [nW]	Dynamic Power [nW]	Energy/op
500 kHz	49.74	13521.6	27.8 pJ/op
1 MHz	49.82	26844.7	

TABLE 4.4: Feature extraction power consumption estimation

The feature extraction block comprises two floating point adders, a multiplier and 4 registers and 2 shift registers, which are active during all cycles of the process. The power estimation extracted at 1 MHz corresponds to 26844 nW, which is close to the value expected if the power consumption of the single components where summed together. The two shift registers cause around 2500 nW dynamic power consumption and are the most power consuming blocks after the floating point operator blocks. The control unit of the module consumes 1 μW. Half of the leakage power, even though negligible with respect to the dynamic power, is caused by the shift registers.

4.2.5 Z-score evaluation

Frequency [Hz]	Leakage Power [nW]	Dynamic Power [nW]	Energy/op
500 kHz	17.1	9354.8	18.6 pJ/op
1 MHz	17.1	18630	

TABLE 4.5: Z-score computation power consumption estimation

The Z-score is evaluated with a floating point multiplier, an adder and 2 registers. The power consumption of the datapath of the module is 17.5 μW at 1 MHz, while the control unit accounts for 500 nW.

4.2.6 Probability extraction

Frequency [Hz]	Leakage Power [nW]	Dynamic Power [nW]	Energy/op
500 kHz	29.44	12788.8	25.6 pJ/op
1 MHz	29.44	25682.2	

TABLE 4.6: Probability extraction power consumption estimation

The module responsible for retrieving the probabilities from the LUT consumes 25711 nW at 1 MHz, mainly due to the activity of the floating point operators. The LUT, which was expected to be a bottleneck for the power consumption, resulted in very limited dynamic power dissipation (2305 nW at 1 MHz) and accounts for 15% of the leakage power of the whole module (4.7 nW)

4.2.7 Forward algorithm and overall power consumption

The power estimations for the forward algorithm, computed with 3 states and in 12 clock cycles, resulted in an energy consumption of 175.5 pJ/op. The power consumption of the system, excluding the forward algorithm block, consists of 181.4 pJ/op in a system with 3 features. The registers which hold the parameters dissipate 42 μW of dynamic power at 1 MHz accross the 12 cycles needed for carrying on all the computations.
The energy consumption of the whole system across the 12 cycles is 398.8 J.

Eq. 4.1 provides an estimation for the power consumption for a system with S states and N features.

$$F.E. + 44.2 \times N + 11.3 \times S^2 + 31.6 \times S \qquad [pJ/op] \tag{4.1}$$

The F.E. extraction power may vary given that for different systems under examination different features may be extracted, the probability extraction and forward algorithm power dissipation figures can be estimated independently on the system based only upon the number of states and features. The $44.2 \times N$ term is dependent on the probability extraction of the features and the computation of their joint probability. The $11.3 \times S^2$ term is related to the calculation of the partial sums $[\sum_{j=1}^{N} \alpha_{t-1}^{j} T_{ji}]$, while the following term $31.6 \times S$ expresses the power dissipated for the calculation of the α values and the normalization step.

Chapter 5

Future work

The model has been implemented and proven to be working in simulated conditions but some optimizations can still be achieved. At the algorithmic level, in order to have a more faithful representation of the data, their statistical distribution can be expressed as a mixture of Gaussians instead of a single Gaussian representation. As it has been reported in Sec. 4.2.6 the retrieval of the standard gaussian probabilities from the LUT is not as power consuming in comparison to the other types of operation; the trade-off between the accuracy of a model with mixture of Gaussians and the increase of power consumption shall be studied.
The forward algorithm only evaluates the α factors of the previous timestep and does not go beyond that point in time; alternative model which consider also older timesteps (also called *higher order models*) should be considered and the improvement in accuracy studied.
The digital implementation suffers from a relatively high power consumption; as it has been shown in Sec. 4.2 the main factor which causes power consumption are the floating point operations. A decrease in the number of mantissa bits may yield lower power dissipations at the price of a decrease the accuracy of the operations. Alternative floating point with fewer mantissa bits representations shall be compared and the relative decrease in power consumption estimated.
Finally the power consumption of the implemented system shall be compared to the power consumption of processors already present in the market, considering in particular those possessing a floating point unit, such as the ARM Cortex-M4F and M7F.

Bibliography

[Chandola et al., 2007] Chandola, V., Banerjee, A., and Kumar, V. (2007). Anomaly detection: A survey.

[Lavin and Ahmad, 2015] Lavin, A. and Ahmad, S. (2015). Evaluating real-time anomaly detection algorithms - the numenta anomaly benchmark. *CoRR*, abs/1510.03336.

[Rabiner, 1989] Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. In *PROCEEDINGS OF THE IEEE*, pages 257–286.

[Ramadass et al., 2012] Ramadass, U., Vijayan, V., Mohanapriya, M., and Paul, S. (2012). Area, delay and power comparison of adder topologies. *International Journal of VLSI Design Communication Systems*, 3.

Master's thesis filing card

Student: Pouya Houshmand

Title: Anomaly detection via low-power on chip machine learning

UDC: 621.3

Abstract:
In an ever-growing interconnected world, the amount of data produced is increasing fast every day and keeping every source of information under control has become a hard and non-trivial task. Anomalies may arise in any instant in this humongous flow of information and human capabilities are simply not good enough in this scenario to achieve the detection in an efficient way. The automation of the task is necessary and special purpose devices must be designed to deal with the problem: the functionality of the service must be guaranteed at all times, especially with critical tasks.
The purpose of this thesis is to find the right algorithm which serves the purpose, its design with hardware description language and the estimation of the performance indexes of the synthesized implementation. Special focus is placed on the low-power design of the architecture, since power affects the reliability, cost and performances of modern chips.

Thesis submitted for the degree of Master of Science in Electrical Engineering, option Electronics and Integrated Circuits

Thesis supervisor: Prof. dr. ir. Maurizio Zamboni,
Prof. dr. ir. Marian Verhelst

Assessor: Ir. Jaro De Roose

Mentor: Ir. Jaro De Roose