MSE | MASTER OF SCIENCE IN ENGINEERING

Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Industrial Technologies (INT), Mechatronics

Model-Based Predictive Maintenance on FPGA

Author:
Sami Föry

Under the direction of:
Prof. Roland Scherwey
TIN/HEIA-FR

Internal expert:
Frédéric Schenker
ROSAS Center

Internal expert:
Jonathan James Hendriks
ROSAS Center

External expert:
Kim Brugnetti
KOORD Sàrl

External expert:
Niall McCullough
KOORD Sàrl

Lausanne, HES-SO//Master, July 20, 2023

Information about this report

Contact information

Author: Sami Föry
MSE Student
HES-SO//Master
Switzerland
Email: *sami.fory@hes-so.ch*

Declaration of honor

I, undersigned, Sami Föry, here by declare that the work submitted is the result of a personal work. I certify that I have not resorted to plagiarism or other forms of fraud. All sources of information used and the author quotes were clearly mentioned.

Place, date: Lausanne, 11.02.2022

Signature: 

Validation

Accepted by the HES-SO//Master (Switzerland, Lausanne) on a proposal from:

Prof. Scherwey Roland, Thesis project advisor

Frédéric Schenker, ROSAS Center, Expert

Jonathan James Hendriks, ROSAS Center, Expert

Place, date: Lausanne, 11.02.2022

Prof. Guido Frosio	M. Philippe Walther
Advisor	Head of MSE, HES-SO//Master

Contents

List of Figures

List of Appendices

Acknowledgments

I would like to thank all the people who have contributed to the success of my Master thesis project in the field of predictive maintenance. First of all, I would like to thank my project manager, Mr. SCHERWEY Roland, electrical engineering professor at the HEIA-FR, for his supervision, his availability and especially his judicious advices which contributed to feed my reflection. I would also like to thank Mr. SCHENKER Frédéric and Mr. HENDRIKS Jonathan James, system engineers at ROSAS Center Fribourg, who actively participated in the realization of this work by advising me and bringing me the support and the necessary steps when I needed it. I would like to thank Mr. BRUGNETTI Kim and Mr. MCCULLOUGH Niall from KOORD Sàrl who accepted to entrust me with a concrete industrial project with a lot of challenges. This one allowed me to acquire an experience and competences not negligible for the beginning of my professional career. I also want to thank them for their interest which stimulated me, their investment and their advices. I would also like to thank Mr. PECLAT Jonathan, software developer at ROSAS Center Fribourg, for providing me with some useful documentations and for having taken the time to help me with some of my reflections.

Finally, I would like to thank all the entity ROSAS Center Fribourg for its reception and its steps so that I can realize my project in their premises and in the best conditions.

Text indications

Abbreviations

API	→	Application Programming Interface
ASIC	→	Application Specific Integrated Circuits
AXI	→	Advanced Extensible Interface
BIF	→	Boot Image File
BSP	→	Board Support Package
CLB	→	Configurable Logic Block
COCOTB	→	Coroutine Cosimulation Test Bench
CPHA	→	Clock Phase
CPOL	→	Clock Polarity
CPU	→	Central Processing Unit
CS	→	Chip Select
DMA	→	Direct Memory Access
CNN1D	→	Convolutional Neural Network 1 Dimension
DDR	→	Double Data Rate
DL	→	Deep Learning
DUT	→	Device Under Test
FD	→	File Descriptor
FN	→	False Negative
FPGA	→	Field-Programmable Gate Array
FP	→	False Positive
GPIO	→	General Purpose Input/Output
GPU	→	Graphics Processing Unit
HEIA-FR	→	Haute école d'ingénierie et d'architecture de Fribourg
HDL	→	Hardware Description Language
HLS	→	High Level Synthesis
HLS4ML	→	High Level Synthesis For Machine Learning
HES-SO	→	Haute école spécialisée de suisse occidentale
I2C or IIC	→	Inter-Integrated Circuit
ILA	→	Integrated Logic Analyzer
iSIS	→	Institut des systèmes intelligents et sécurisés
LSB	→	Least Significant Bit
LSTM	→	Long Short Time Memory

MAE	→	Mean Absolute Error
MISO	→	Master Input Slave Output
MM2S	→	Memory Mapped to Slave
MOSI	→	Master Output Slave Input
MSB	→	More Significant Bit
MSE	→	Master of Science in Engineering
MSE	→	Mean Square Error
PL	→	Programmable Logic
PS	→	Processing System
RAM	→	Random-Access Memory
RMSE	→	Root Mean Square Error
ROSAS	→	Robust and Safe System
RNN	→	Recurrent Neural Network
RT	→	Real-Time
RTL	→	Register Transfer Level
SBC	→	Single Board Computer
SCLK	→	Serial Clock
SoC	→	System on Chip
SPI	→	Serial Peripheral Interface
SS	→	Slave Select
SVM	→	Support Vector Machine
S2MM	→	Slave to Memory Mapped
TCF	→	Target Connection Framework
TM	→	Thèse de Master
TP	→	True Positive
UIO	→	Userspace I/O
VHDL	→	VHSIC Hardware Description Language
V&V	→	Verification & Validation
XIL	→	X-In-the-Loop

Legends

Italic	→	file name/format
Symbolic	→	Paths
Underlined Roman	→	Hypertext links

```
Command / Part of code
```

Abstract

Predictive maintenance is a current field where a lot of efforts are applied in order to allow an early detection of system defects. Anomalies are treated directly to avoid the need for corrective maintenance, which can be very costly in many cases. Predictive maintenance is mainly implemented with the help of data analysis tools that allow the detection of anomalies on measured signals.

Vibration analysis is the most widely used technique to perform predictive maintenance on industrial machinery. This data is easily accessible in large quantities with the use of sensors. With a good detection model, the results can be accurate and reliable. This type of data is very representative of the health of a system.

Some anomaly detection methods are difficult to implement, such as classification methods. It is necessary to use clean data and data containing anomalies, which is relatively difficult to obtain knowing that anomalies are rare and that there are often not enough of them for a model to retain them. For this reason, the use of analysis techniques based on the comparison of real data with healthy behavior data is beneficial, as it is easy to obtain. In addition, when marketing a predictive maintenance tool, it is difficult to tell the customer that the tool only works if it provides data that contains anomalies.

The use of field programmable gate arrays (FPGA) to implement machine learning networks is an efficient and flexible solution, especially for real-time applications. With an optimized and accurate machine learning model, it is possible to obtain relatively low latencies and therefore fast results that allow for more accurate analysis. There are many types of FPGAs that can support large resources, but the costs are also proportional. It is therefore important to find a compromise between latency, accuracy and resources.

This paper presents the development of a predictive maintenance tool based on vibration data analysis. The use of a deep learning model implemented in an FPGA allows an application and a detection of anomalies in real time. The type of network implemented is the "Long-Short Term Memory" (LSTM) which allows to predict the behavior of temporal series over long periods. The FPGA is used to accelerate the operation of the trained model on an SoC and to reduce the latency of the processing system. A prototype version of HLS4ML allowing to generate RTL equivalence of LSTM models is used in this project.

Key words: Anomaly Detection, Deep Learning, Embedded Linux, Embedded Systems, FPGA, Hardware Engineering, HLS, HLS4ML, Keras, LSTM, Machine Learning, MBE, Prediction, Predictive Maintenance, Python, Real-time, SoC, Software Development, SPI, Time series, V&V, Vivado

1 Introduction

Contents

1.1 Context

1.1.1 General context

Predictive maintenance is a field that is becoming more and more important in the industry in general. It is an interesting way to detect defects on an equipment before they occur and therefore to finance a modification only when it is necessary, contrary to preventive maintenance. The implementation of a predictive maintenance program on high value-added activities can lead to a very important return on investment. According to a study conducted by the U.S. Department of Energy [8], it is possible to obtain financial savings of more than 30 to 40% for companies that opt for predictive maintenance programs. Some systems are critical to the operation of a business and a shutdown of these systems can cost the company a lot of money. Depending on the size of the company, the average cost of equipment downtime is $260,000 per hour. Knowing that the average downtime is four hours, this represents a financial loss of approximately $1,040,000 according to [11]. Despite the fact that the solutions are generally expensive, this is a field with a lot of potential for the industry.

Nowadays, systems are becoming more and more consistent and complicated. The use of models is therefore an aspect of a design in general that is very important and where much effort is applied. A model must be verified and validated before the system can be implemented. It is therefore important to use efficient and reliable verification and validation methods in order to save time in this phase and to ensure that the model works according to the initial requirements. The verification and validation part of a design takes up a significant amount of time, about 50% of the total time. For this reason, a lot of effort is also put into innovating and finding new and more efficient methods.

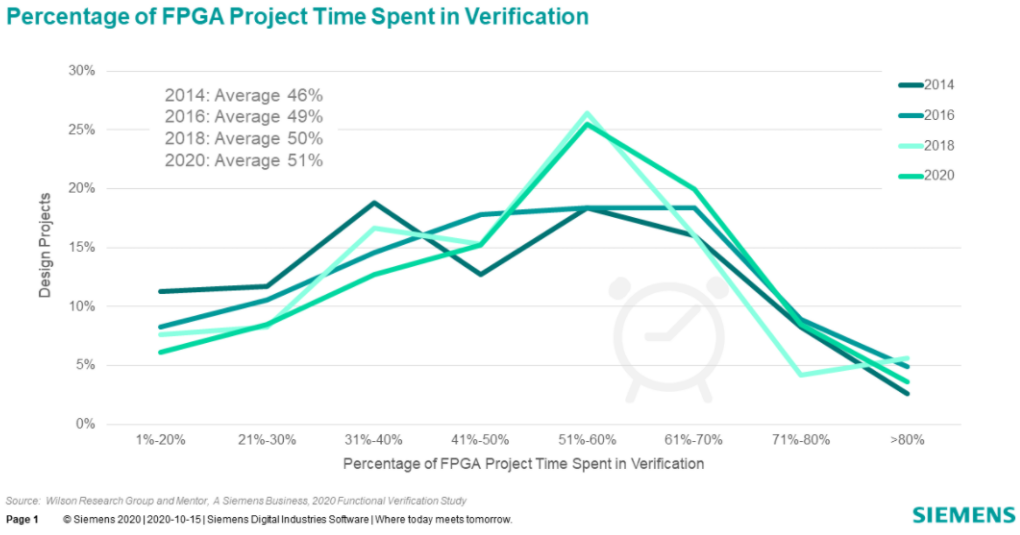


Figure 1.1 Percentage of spending time for verification step[a]

[a] Extract from [10]

A model can also allow a system to control itself by using the model as a reference. If the model is verified and validated, it will be possible to compare its behavior with the real behavior of a system. If the latter diverges too much from the model's behavior, an alert will be triggered. Obviously, for this comparison to be reliable, it will be mandatory to put the model and the physical system in the same conditions of use.

1.1.2 Specific context

KOORD Sàrl is a company specialized in the mechatronic systems. Important efforts are put in the development of robust, reliable and low cost tools to obtain quality systems. Koord works with major clients. His experience with them led him to think that a predictive maintenance solution could be interesting. For this reason, the company has developed a test tool with an algorithm implemented in an Octavo OSD3358-512M-BSM processor. This tool allows to test the lifetime of mechatronic systems on the basis of a fault detection which is currently done "offline". The data are acquired with an accelerometer and the processing of the data is done on a computer, once all the data have been acquired. This tool works well, but is relatively resource intensive in terms of processing time and infrastructure. The idea of using an FPGA in this context would allow to acquire the data and to simultaneously process them in the FPGA chip before displaying them. The control would be in real time. It is this aspect that motivated the proposal of this master thesis topic and to implement the concept on an FPGA. This project is interesting for industrial applications and would allow a predictive maintenance of mechatronic systems in order to increase their life span.

This master's thesis is interesting and falls under different fields of engineering, including computer science and electronics. These are two fields that are often linked and where the demand is important. Acquiring experience in these fields at the end of this project will be extremely beneficial. The focus is on a modeling phase at the software level and implementation at the hardware level through testing phases.

1.1.3 Project context

The project who's will be treated is a Master's Thesis and it takes an important part in the context of study of the "Master of Science HES-SO in Engineering". This master thesis concerns the mechatronics orientation. It focuses on the aspects of model-based simulations for system verification and validation, artificial intelligence with deep learning algorithm and physical implementation on FPGA. The deep learning algorithm is based on the time series techniques. It aims at applying a predictive maintenance on the systems used by industries at the level of vibratory defects.

The steps included in the scope of this project are described below :

- First, it will be important to organize the course of this project with a detailed planning of its phases.
- Then, it is necessary to deepen the various topics addressed in this project. In particular, the predictive maintenance on the basis of vibratory data, the implementation of a deep learning algorithm on FPGA and the real-time application of the tool.

- The modeling of the algorithm in an HDL language in order to be able to simulate and synthesize it will take an important part of this project.

- It will be necessary to carry out a verification and validation phase of the model in order to verify that the outputs correspond to the functional requirements of the tool. To do this, it is planned to use the "cocotb" method which allows to test the system with a test bench written in Python. The method may change as the project evolves.

- Once the system is validated in an "offline" way, it will be necessary to choose a SoC FPGA board to realize the necessary configurations on the Vivado environment and to physically implement the model.

- Once the algorithm is physically usable, it will be possible to test the predictive maintenance tool in real time, with the requirements that go with it. Before that, it will be necessary to establish and design the software process of the system. That is to say, the acquisition of data, the processing/transformation of these and the display of the results.

- In addition to the work to be done, it will be necessary to establish a complete report of the project as well as weekly summaries for its follow-up.

The initial scope of the project was defined according to the representation in appendix B.

1.2 Aim of Study

Based on vibration data representing the complete life of a ball bearing acquired with an accelerometer, the goal of this end of master work is to develop a predictive maintenance tool directly for Koord Sàrl and to bring additional skills to Rosas Center Fribourg, especially in predictive maintenance, which is an area related to safety. This tool must be able to detect anomalies accurately and in real time. The goal is to develop a trained machine learning model in order to implement it in an FPGA to perform the analysis in real time. The model will serve as an anomaly detector by running in parallel with the real system.

The different main requirements provided by Koord are listed in the table below.

Master Thesis Project Requirements

Functional requirements

N°	Function	Criterion	Requirement	Flexibility
1	Analysis of the results	Data acquisition time Data processing time Display results time	10 minutes max.	F1
2	Algorithm training	Acceptable time before analysis phase	1 hour	F3
3	Real-time analysis	The tool must allow an online test	Available	F0

Non-fonctional requirements

N°	Function	Criterion	Requirement	Flexibility
1	Hardware composition	Maximal price of the tool	30.-	F1
2	User utilisation	Safety utilisation	Data security	F0
		Complexity of utilisation	Userfriendly	F0

Flexibility level	
F0	No flexibility
F1	Low flexibility
F2	Good flexibility
F3	Strong flexibility

Figure 1.2 Project requirements

This tool must be marketable afterwards, so it is necessary to find a solution that uses licenses allowing this. Moreover, it is necessary to provide a solution with correct implementation costs.

Currently, Koord uses an LSTM network to perform predictive maintenance offline. The initial idea of this project is to use the same type of network to obtain equivalent results.

This project will go through several important steps to achieve the final goal. These steps are the following.

1. A neural network modeling phase at the Python level that will allow to detect anomalies when it is compared to abnormal data.

2. The conversion of the model into a synthetizable RTL project.

3. Offline verification and validation of the model at different levels.

4. Hardware configuration and implementation of the model in an adapted FPGA.

5. Online verification and validation of the predictive maintenance tool.

At the end of this work, it would be interesting to obtain a single board computer (SBC). That is, a complete and autonomous system at the level of use. External peripherals, such as a screen, keyboard and mouse could be connected directly to the tool in order to monitor it. All this would allow to have a tool with edge operations, which is close to an interesting marketing perspective.

2 Related Theory and Literature Review

This part of the report serves to deepen the different concepts and tools that will be used during this project. That is to say, from the development of the neural network, to the hardware implementation on FPGA, through the phases of automatic code generation, offline and online verification and validation, and FPGA configuration.

Contents

2.1 Predictive maintenance of vibratory defects

Anomaly detection is a current field where a lot of efforts are put in many areas, such as health, finance, predictive maintenance or cyber security. For this work, the targeted domain is anomaly detection based on data analysis tools for predictive maintenance. The goal is to detect as soon as possible, i.e. at the first signs of anomalies, a vibratory defect of a system. By proceeding in this way, it will be possible to repair a system in an early stage in order to avoid its shutdown, which can be quite costly.

2.2 Neural Networks

Artificial intelligence is a modern technology, which has largely proven its efficiency. It is used in many fields and has allowed the technologies used to evolve in a significant way. We find artificial intelligence in our phones, our cars, in medical equipments, in finance, etc.

Neural networks are inspired by the functioning of the human brain and allow to recognize numerical patterns from large amounts of data and make decisions in the place of the human. When we talk about several hidden layers, we talk about deep learning. Deep learning networks can be implemented with labeled and unlabeled data.

2.3 Supervised and unsupervised learning

Supervised learning is an approach that is based on the use of labeled data. Supervised learning is a method that allows to classify data or to accurately predict time series. For example, in the case of time series prediction, the model will retrieve input data and output data. It will only learn in an iterative way to reproduce the output data according to the input data.

Unsupervised learning is an approach that analyze and cluster unlabeled data sets. These algorithms discover hidden patterns in data without the need for human intervention. Hence the name "unsupervised".

A supervised model will learn to obtain a desired and known result. An unsupervised model will learn by itself, from a large data set, the different links that exist between the data. Thus, the model will be able to automatically extract certain relevant results.

2.4 LSTM Networks

The "Long Short-Time Memory (LSTM)" layers are the basis of our modeling. It is therefore important to understand how they work and why they are relevant for our use. LSTM networks are derived from RNN networks. First, it is interesting to know how Recurrent Neural Networks (RNN) work and what are their limitations that led to the use of LSTM networks.

2.4.1 Recurrent Neural Networks

We can compare neural networks with the human brain. Humans do not start thinking from scratch at every moment. For example, when we communicate, we base each new piece of information on previous information.

Classical networks cannot do this. They cannot take into account past information to make choices about future events for example. This is where RNNs come in. They are made up of loops which allow to maintain the information.

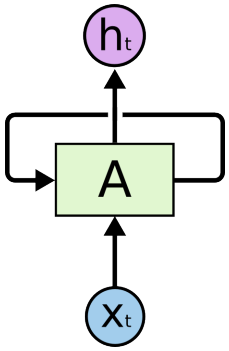


Figure 2.1 Rolled Recurrent Neural Network[a]

[a]Extract from [3]

This representation is equivalent to a network that transmits its information to an identical following network.

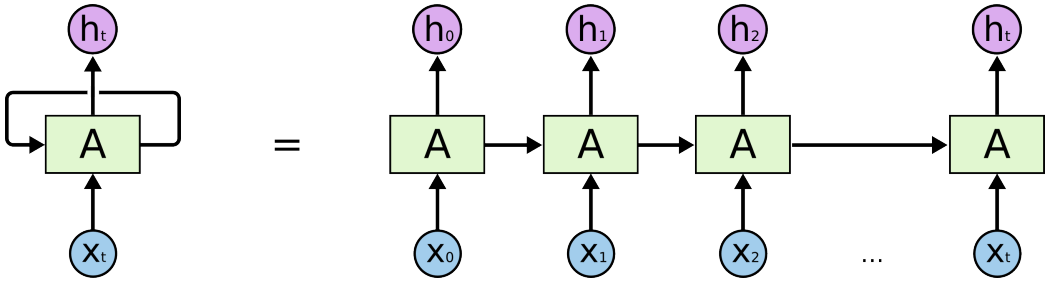


Figure 2.2 Unrolled Recurrent Neural Network[a]

[a]Extract from [3]

It is relatively intuitive to see that this type of network has the right architecture to handle sequential data naturally.

2.4.2 Long-Term Dependencies

The main limitation of RNN networks is the length of the dependencies. When the information required at time t is too far away in time, RNNs cannot do their tasks correctly. We can take for example the prediction of certain words in the time of a conversation. If a person says "I was born in England, so I speak fluently ...", the RNN model will be able to recognize that the next word is "English", because the length of the dependency between the time when the person speaks about where he was born and the word he is looking for is relatively small.

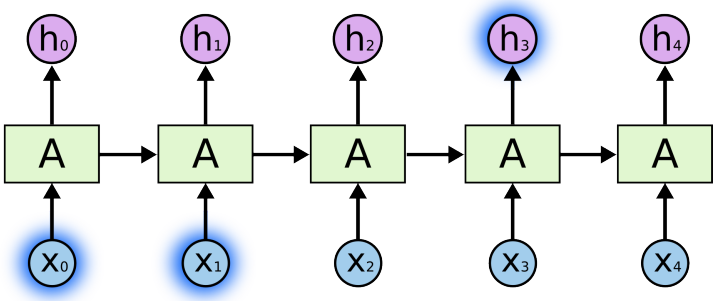


Figure 2.3 *RNN short term dependencies*[a]

[a]Extract from [3]

In the case where the person says "I was born in England, I work as a freelance developer, ... *(long time after)*, and I speak fluently ...". it will be much more difficult for an RNN model to recognize the next word, because the dependency will be much greater and the model does not retain information from the past in the long run.

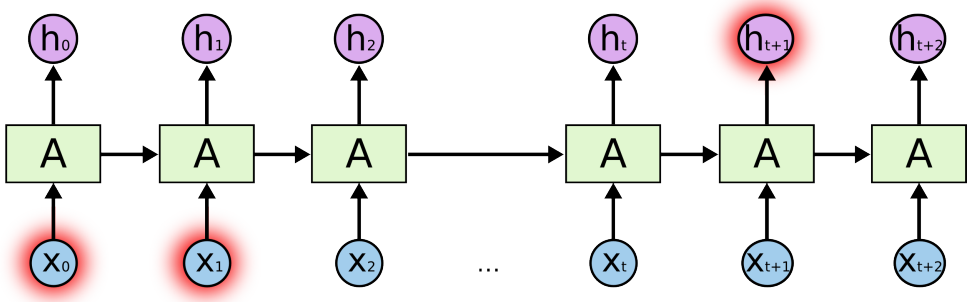


Figure 2.4 *RNN long term dependencies*[a]

[a]Extract from [3]

The LSTM networks were created to counter this problem.

2.4.3 Long Short-Term Memory (LSTM)

LSTM networks are a type of network that is part of RNN networks. They have been designed to avoid dependency problems in the long term, as we have seen with the previous example. As for all RNN networks, the modules in an LSTM network are repeated.

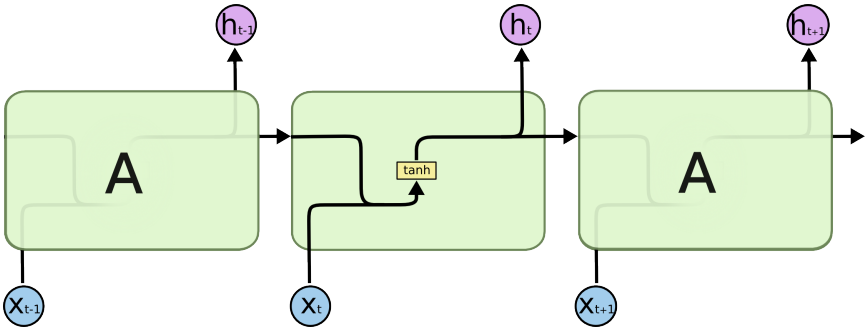


Figure 2.5 *RNN chain description*[a]

[a]Extract from [**colah**]

The difference is that in a simple RNN network, there is a single layer function inside the modules, where for LSTM networks there are four.

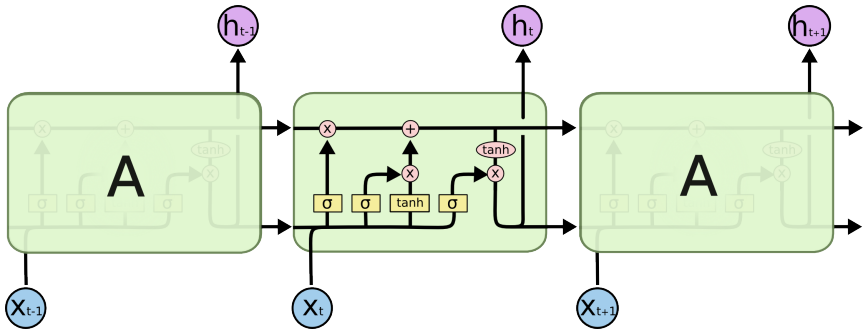


Figure 2.6 *LSTM chain description*[a]

[a]Extract from [3]

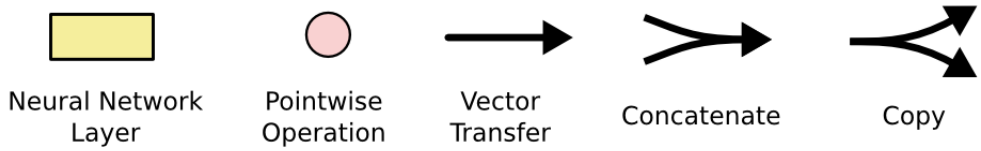


Figure 2.7 *Chain legends*[a]

[a]Extract from [3]

The layers interact in a very precise way in order to keep the information of the different states in the long term. Obviously, this leads to a higher complexity. The calculations are more consequent and the LSTM networks require more important resources.

2.5 Field-Programmable Gate Array (FPGA)

FPGAs (Field Programmable Gate Array) are semiconductors that are based on a matrix configuration of logic blocks (CLBs). FPGAs can be reconfigured as needed after they are manufactured. This feature makes this device very flexible and adaptable. We can make a quick comparison with ASICs which are designed in a hardware way specifically for a certain need, the circuits are etched in hard, unlike FPGAs. It is therefore not possible to reconfigure the device after manufacturing. Logic blocks can be configured to perform complex combinatorial functions or simple logic gates, such as AND or OR gates. FPGAs, like ASICs, are programmed with specific languages, which are mainly Verilog and VHDL. They are grouped in a category of languages called Hardware Description Language (HDL). These languages allow a low-level description of the behavior of a system based on FPGA. They are relatively complex languages and that is why some alternatives exist, such as the generation of code from a system described in HLS (High Level Synthesis) language. The flexibility and freedom is reduced compared to a direct description in HDL, but the passage through the HLS level can save time during the development of the system.

Thanks to its programmable nature, the FPGA is adapted and used in several fields. For example, in automotive for driver assistance, in medical, for diagnosis and monitoring of certain tasks, in ASIC prototyping or when high performance calculations are required.

There are different manufacturers and suppliers of FPGAs, the best known are Xilinx and Intel. They offer FPGAs that can perform edge computing and others cloud computing.

2.5.1 Edge and cloud computing

There are several possibilities for implementing embedded systems. Cloud computing is used when the processes that enable a system to function go through servers in order to perform some or all of them. We talk about cloud computing for example when calculations are performed outside the physical system or when data is transferred to the cloud to be stored there. On the other hand, we talk about edge computing when everything is done directly on the physical board thanks to the different components it uses.

There are different advantages and disadvantages on both sides, the most important ones are for example the security of the information. Data that remains local is not likely to be intercepted, unlike data that passes through servers. On the other hand, if a part of the processes is carried out outside the physical equipment, it is obvious that much more resources can be used in terms of calculations, storage, etc.

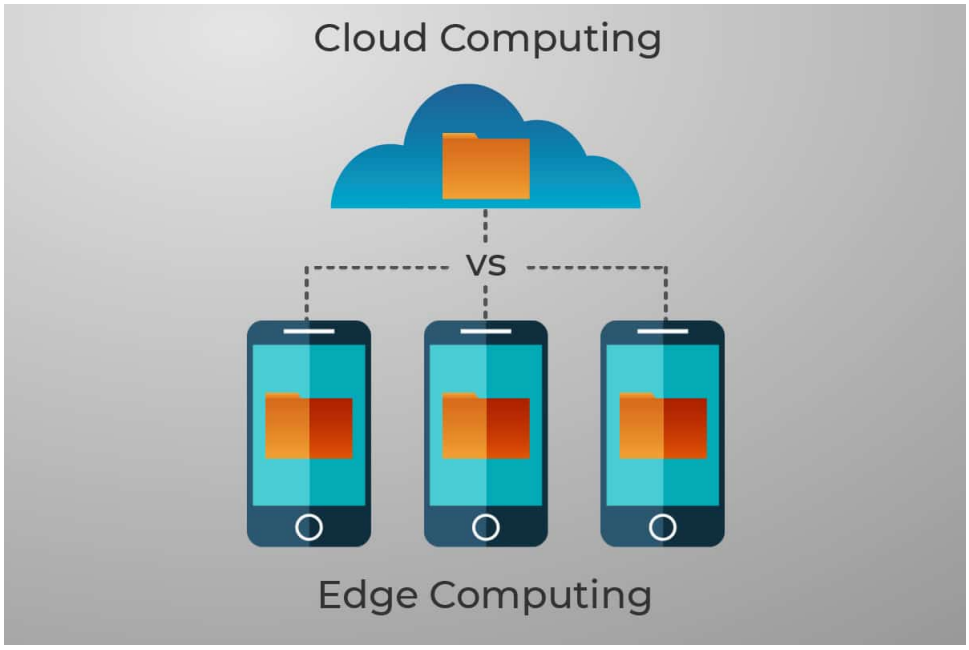


Figure 2.8 *Cloud vs edge computing representation*[a]

[a]Extract from [16]

2.6 System on Chip (SoC)

A System on Chip or SoC, is an integrated circuit or chip that includes different components. Most of the time, there is at least a CPU, a memory and inputs/outputs. There can also be components, such as GPUs or FPGAs. The advantage of having all components in the same chip is the low impedance due to the communication length. The communication between the components of a chip is done with a very low latency, which is very advantageous in some cases where the response times must be very low.

2.7 LSTM networks implementation on FPGA

Implementing neural networks on embedded systems is very important nowadays, mainly due to the evolution of the complexity of different systems. We can for example find neural networks in our cell phones or in some cars. Being able to embed neural networks allows to bring intelligence to the systems which will be able in some cases to make decisions or to propose optimal solutions to their users according to the observed data.

Obviously, an embedded system contains fewer available resources than a computer. This is why some of the more resource-intensive networks are not easily embedded. LSTM networks are relatively resource-intensive. The fact of keeping the states of a time series on the long term leads to relatively consequent networks. Moreover, the use of an unsupervised learning strategy requires a parameterization of the network which will increase the necessary resources. It is mainly for these networks that LSTM networks are difficult to embed.

Implementing a neural network in an embedded system will make it easier to communicate with other devices in real time, such as a data sensor. The ecosystem of these is more suitable with I/Os that can be easily used for different types of communication. Moreover, when it comes to a product to be commercialized, it is necessary that it can be adapted to one or a few specific needs in order to decrease the cost price.

Part of the work presented is based on code conversion in order to go from a high level representation, which is Python, to another, low level and synthesizable, which is register transfer level. This is done through a conversion at HLS level. It is therefore important to become familiar with these two levels of representation.

2.7.1 High Level Synthesis

High Level Synthesis (HLS) is a technology that facilitates the development of hardware behavior at the RTL level in order to implement it in an FPGA or ASIC. HLS specifications can be written in C, C++ or SystemC in a relatively specific format in order to be able to convert it with the available tools into an RTL model.

HLS Benefits

Designing directly in an HDL language is relatively difficult and time consuming. This is why this technology exists. It allows to save time in the development of the behavior of a numerical system by going through a high level of abstraction with languages much more familiar to software developers.

By developing algorithms on the C-level, it is possible to test them on the C-level with a suitable test bench. This means that a gain in productivity is also possible during the verification and validation phases of a system. The same test bench written in C can be used to validate a C-level model and an RTL-level model with C/RTL cosimulation.

Nevertheless, the use of this technology logically leads to a reduction in development flexibility, but in the majority of cases, this is negligible and the benefit of quality over time is favorable.

The following representation describes the development steps of a system through an HLS abstraction level.

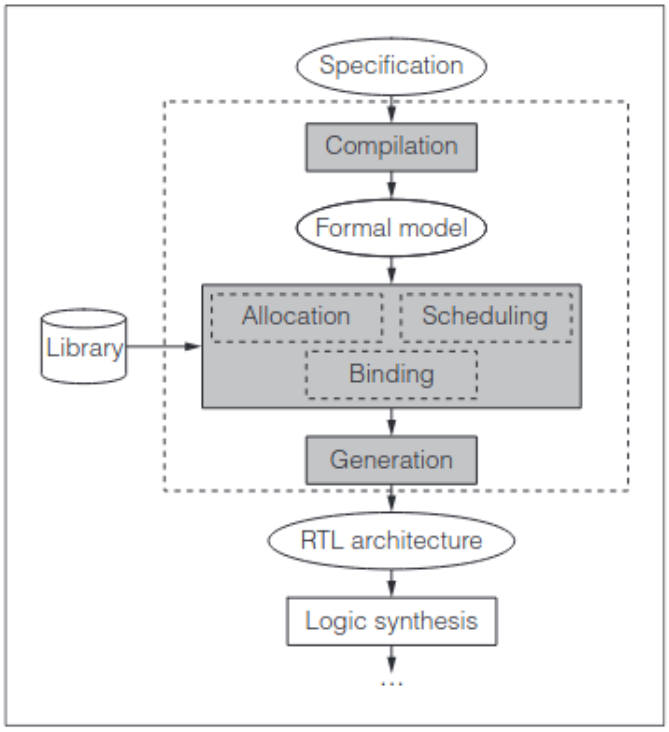


Figure 2.9 High Level Synthesis (HLS) design steps[a]

[a] Extract from [5]

1. Compiles the specification
2. Allocates hardware resources
3. Schedules the tasks
4. Binds the tasks to functional units and variables to storage elements
5. Generates the RTL design

2.7.2 Register Transfer Level

In digital system design, register transfer level (RTL) is a way of describing the architecture of a system. In RTL design, the operation of a circuit is defined in terms of the sending of signals or the transfer of data between registers. It is also defined in terms of logical operations performed on these signals. RTL is used in hardware description languages (HDL), such as VHDL or Verilog, which allow to describe the behavior of a system at a higher level of abstraction.

2.8 Anomaly detection in time series

Anomaly detection consists in detecting observations, points or patterns that do not correspond to normal behavior. As far as time series are concerned, they are made of multiple values representing points evolving in time. This means that to characterize a time series, at least two dimensions are needed. The timestamp and the value. An anomaly in a time series of values is one or more points in time that deviate from normal behavior, in terms of raw value, trend, etc. To perform effective anomaly detection, it is not enough to rely solely on each point to be analyzed. It is also necessary to work with past values in order to retain information over the long term. For this purpose, an anomaly detector based on an LSTM network is a good choice.

2.8.1 Unsupervised learning LSTM-based Autoencoder

The unsupervised learning LSTM-based Autoencoder is an anomaly detection method based on the use of deep learning. It combines the different principles seen previously to obtain a powerful concept in the quality of detection. This concept allows the use of raw data not labeled for the learning of the model. To do this, it is necessary to use a large amount of data that do not contain any anomalies. The model will therefore be trained only on the basis of clean data. The advantage is that, in general, it is not difficult to obtain data that represent the normal behavior of a system. It is much more difficult to obtain a large amount of anomalous data required by some other detection methods.

The principle of the autoencoder is based on the reconstruction of input data. The prediction of temporal data consists in providing a set of input data and another set of output data to the model so that it learns, on the basis of the inputs, to reproduce the output data. In the case of the autoencoder, the input data will be the same as the output data. By iteration, as for prediction, the model will try to reconstruct as well as possible the input data provided. It is also important to correctly represent the encoding and decoding parts of the network. To do so, it is necessary to create a bottleneck by reducing the learning capacity of the model on the hidden layers. The representation below illustrates this principle of auto-encoding at the neural network level.

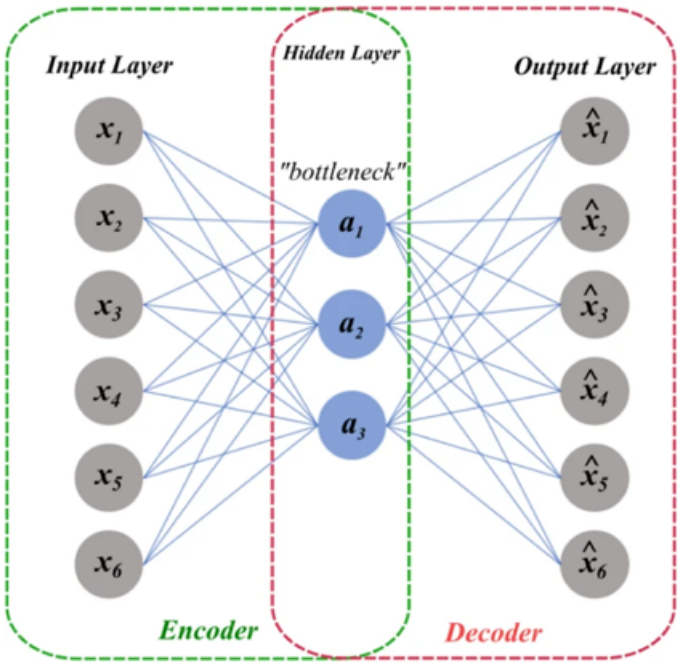


Figure 2.10 General autoencoder representation[a]

[a] Extract from [31]

The use of this concept to perform anomaly detection is based on the error in the reconstruction of input data. If the model encounters data that diverges in one or more aspects from what it has learned, it will fail to reconstruct the data correctly and the error will be large and visible. The point in question will be classified as an anomaly if it exceeds a certain defined threshold.

2.8.2 Statistics-based anomaly analysis

- **Gaussian distribution for threshold definition**

  To detect an anomaly, it is necessary to define a certain threshold that marks the boundary between points representing anomalies and points considered normal. For this purpose, the Gaussian distribution based on the reconstruction errors of the training data is a good reference. The threshold is defined by taking the rightmost point of the distribution. This principle works well, in particular because the training is done only on the basis of seine or considered health data.

- **Mahalanobis distance for rarity measurement**

  Once the anomalies are detected, we obtain data labeled as "anomaly" and "non-anomaly". It is therefore important to evaluate the importance of each anomaly and to do this, an interesting method is the calculation of the Mahalanobis distance that can be considered as a reference of severity. The higher the calculated distance, the more important the anomaly will be. The mahalanobis distance is calculated with the following formula.

$$a(e) = (e - \hat{\mu})^\top \hat{\sigma}^{-1} (e - \hat{\mu})$$

In the case of one-dimensional data, it will be as follow. $\hat{\sigma}^2$ is the sample variance.

$$a(e) = \frac{(e - \hat{\mu})^2}{\hat{\sigma}^2}$$

The Mahalanobis distance is often used for the detection of outliers in a dataset. This technique minimizes the influence of the noise of the noisiest components (those with the largest variance).

2.9 Conclusion

In this section, we have been able to highlight different main concepts that have been used to carry out this work. These explanations have been structured to describe the general concepts at first, until arriving at the more specific concepts used in this project. Being able to familiarize oneself with these theories and tools will make it easier to understand the choices and methods used later. Obviously, new concepts and additional details will be discussed throughout the report. The different aspects of this section are based on several articles, documentations, tutorials, etc. These will be referenced at the end of the report.

To summarize, we have provided some details about the predictive maintenance domain, then we have discussed several aspects about neural networks which will be the core of this project. We then provided details about the physical implementation part by describing what is an FPGA with possible usage methods (edge and cloud computing) as well as details about the implementation of an LSTM network in an FPGA by going through different levels of abstraction. We ended by detailing the concept of anomaly detection on time series. This step will be important for the validation of our tool, during the modeling phase, but also during the development phase of the main system application.

3 Environment Definition

It is interesting to detail the environment in which the project will be developed. That is to say, to go through the different tools and technologies adopted to carry it out. In this section, we will do this in a structured way by going through each major step of the project.

Contents

3.1 Modeling

The modeling phase of the project consists in obtaining a functional deep learning algorithm allowing to reproduce the behavior of a new system at the vibratory level. This phase goes from the acquisition of real data, to the generation of the project at the RTL level.

3.1.1 Data acquisition

To model the behavior of a system, it is necessary to have data that represents it. The data we are interested in are vibratory data from a 3-axis accelerometer. These data are initially provided by the company Koord Sàrl in order to realize the offline phase of the project. For the real time part, it will be necessary to take into account this stage of acquisition in a more detailed way, because it is it which will be with the source of all the process of real time analysis.

The useful data for the project represents the entire life of a bearing. This data is raw and stored in CSV files. There are 255 files with approximately 65'000 samples per file.

3.1.2 Python Keras

The deep learning model was realized in Python using the standard machine learning library, Keras. This part includes the phases of data structuring, model training and results analysis.

3.1.3 Model training

To train large models with a lot of parallelism, it is best to configure a GPU to handle the computations in order to increase the parallel computations and decrease the training time. If the model is relatively small with little parallelism in the computations, the use of a CPU is sufficient and sometimes faster due to its high frequency.

In the case of using Tensorflow for the training phase of an algorithm, it is necessary to use a CUDA-compatible Nvidia graphics card and to install various adapted drivers. It is necessary to take care to use the good compatible versions between Tensorflow, CUDA, Python and the cuDNN library for a functional and optimal configuration. The configuration process for using Tensorflow with a Nvidia GPU is described in the Tensorflow documentation in [36].

3.2 HLS/RTL conversion

The model conversion phase is based on the use of an open source tool which is HLS4ML in order to go through a HLS to RTL conversion with Vivado HLS.

3.2.1 HLS4ML

The conversion of the Python model into an HLS model was done using a prototype version of HLS4ML from a specific GitHub repository named "Keras-RNN" [4]. This repository brings the necessary addition to HLS4ML to support LSTM based networks. Other libraries than Keras, like PyTorch or Tensorflow are normally also supported. The utilization of Linux is necessary with HLS4ML.

3.2.2 Vivado HLS

The HLS project is imported into the Vivado HLS 2019.2 software to be able to simulate the model at the C-level, synthesize the model, and cosimulate it at the C/RTL level. It is also possible to export the project in IP format. It is important to note that HLS4ML is not fully compatible with all versions of Vivado HLS. Versions 2018.2 through 2020.1 are recommended. You have to be careful that the Ubuntu version used is compatible with the Vivado HLS version chosen. In our case, Ubuntu 20.04 was used with Vivado HLS 2019.2, despite the fact that these two versions are not necessarily recommended. The use of Vivado HLS did not pose any problem at this level.

3.3 Verification & Validation

The verification and validation phase of the developed model requires the use of efficient tools in order to measure the efficiency of its anomaly detection. The verification and validation had to be done at several levels. The main model developed at the Python level must be validated at the same level. The evaluation of the other levels, i.e. C and RTL was carried out on the basis of a comparison of the results obtained with the same inputs at all levels.

3.3.1 Numenta Anomaly Benchmark

The verification and validation phase at the Python level will have a considerable impact on the final results. This is why it is necessary to use an adapted verification method that allows to obtain quantitative results rather than qualitative ones. Numenta Anomaly Benchmark (NAB) is an open source solution that has been designed to test detectors under test (DUT). A scoring mechanism as well as datasets containing known anomalies are available for this phase.

3.3.2 Python verification script and C testbench

The conversion of the Python model into a model written in C/C++ will lead to a modification of the precision, depending on the one chosen during the conversion configuration. For this reason it is necessary to evaluate this difference in accuracy and to judge whether the detector is still able to detect anomalies efficiently. To do this, a Python script for graphical comparison was realized in appendix N. This comparison is made between the values reconstructed in Python and the values obtained from the C simulation.

3.3.3 C Test Bench for C simulation and C/RTL cosimulation

A test bench written in C/C++ is provided with the HLS4ML directory and was used to perform not only the C simulation, but also the RTL simulation (C/RTL cosimulation). By providing the test bench with a list of input values and the results obtained at the Python level in *.dat* format. It is possible to obtain in output of the C simulation, the results of the reconstruction at the C level with the same values as at the Python level and the same for what concerns the RTL level. To perform these simulations, the Vivado simulator was used.

3.4 Hardware Implementation

The hardware implementation phase requires the use of different adapted tools proposed by the Xilinx suite. We had to develop the global hardware strategy, the main application of our tool and we also embedded a Linux OS in the processing system part of the SoC in order to use the Linux ecosystem to manage the different processes.

3.4.1 Vivado

After developing, verifying and validating the model at all levels, we can export the model in a reusable IP format to other Xilinx platforms. First, the generated IP is used with Vivado 2019.2 for the hardware block design. Using a more recent version than the one used for Vivado HLS will lead to errors. It is therefore preferable to use the same versions for Vivado HLS and Vivado. In our case, both versions are 2019.2.

3.4.2 PetaLinux

In this project, we will implement our solution using the Linux environment. PetaLinux provides a set of tools to embed a Linux OS in a Xilinx device. These tools allow us to create, configure and build the PetaLinux project in order to boot a Linux image in our target board. We can customize and optimize the OS to fit the need. PetaLinux is based on the Yocto project which provides open-source tools and processes to develop projects with embedded Linux.

The representation below shows different possibilities to implement a Linux image in a Xilinx target.

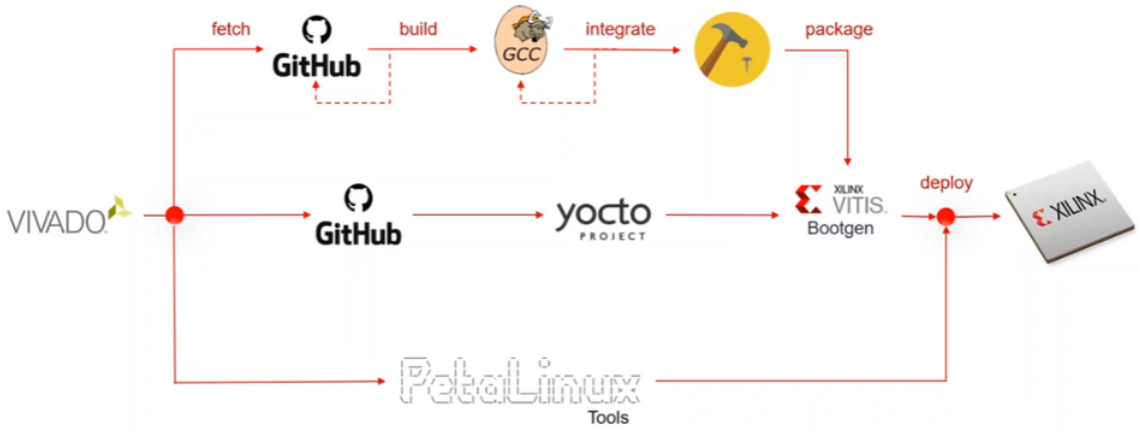


Figure 3.1 Embedded Linux implementation possibilities[a]

[a] Extract from [51]

3.4.3 Vitis IDE

Vitis IDE is a software development platform provided by Xilinx and is adapted to other tools, such as Vivado or PetaLinux. It is a software that, on a hardware base developed from Vivado, allows to develop, compile and debug the main application to be integrated in the processing system. To avoid problems, we have used the 2019.2 version of Vitis in order to keep compatibility with the Vivado version used. It is possible to develop and debug applications written in C or C++ to be integrated in a bare metal or embedded Linux environment.

3.4.4 SoC Zynq Ultrascale+ with FPGA ZCU104

On the hardware side, this project is based on the use of a Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. It is a board that gives us a certain flexibility for the implementation of our project and to prove the global concept of it. Obviously, in a marketing context, a board like this one will not necessarily be mandatory, because we have to take into account the fact that it is a relatively expensive board.

4 | Deep Neural Network Modeling

This section is dedicated to the development of the deep learning model that will be the basis of the developed anomaly detector. We will go through different explanations of general and specific concepts that concern neural networks in general and in our use in particular. The steps of realization of the deep learning model go from the structuring of the acquired data, until the test of this one. All this is done using Python and its Keras library for the Deep Learning side.

Contents

4.1 Unsupervised LSTM Autoencoder Architecture

The strategy adopted to realize a predictive maintenance tool is the use of a model based on LSTM network. These LSTM layers will allow a maintenance of the states in the time to be able to reconstruct in an optimal way the entries of the network. The model is trained on "clean" data, i.e. without anomalies. This reconstruction is based on the principle of autoencoder which will be described later. The reconstruction errors of the model inputs will serve as an anomaly detector.

4.1.1 Unsupervised Learning

Let's go back to what was described earlier. Let's first consider supervised learning. It is based on providing the model with a set of input data and a set of output data. On this basis, the model will simply learn to best classify or predict the output data according to the input data. It will do this many times, until it gets a minimal prediction error. To do this, it is not necessary to have a huge amount of data, unlike unsupervised learning, as we will see. It is also important to know that supervised learning is rather done when using labeled data. Supervised learning can be separated in two types of problems. Classification and regression.

For unsupervised learning, it uses mainly unlabeled datasets. This is what we will allow the model to use all the necessary information for discover hidden patterns in the data and this, without an human intervention. This type of training requires a large amount of data so that the model can correctly analyze the existing patterns. In our case, we will give the model a large amount of data so that it can analyze them and train itself to reconstruct the input data taking into account the different hidden patterns. Thanks to this, the model will be able to detect not only aberrant anomalies, but also those that would be more subtle and not necessarily visible.

4.1.2 LSTM-Based Network

There are many types of networks and layers that can allow us to run a project based on machine learning. However, some are more suitable than others for certain types of problems. Some by nature are more powerful, but require more resources to implement.

Here, we only work with time series data. That is to say data acquired and based on time. Many networks are relatively well adapted to perform image processing, it is especially in this type of project that machine learning is applied. Nevertheless, there are RNN type networks, which are very well adapted for time series processing. The details and drawbacks have been explained previously for this type of network. The LSTM networks are a sub-category of the RNN networks allowing to avoid the explosion of the gradient on the long term.

The LSTM networks are the basis of this project, because it is the type of network, the most adapted to provide an accurate anomaly detection on the long term. Indeed, we will see in the results that an anomaly detection can also be done with a series of dense layers. This is a very basic type of layer and widely used in all types of models. The dense layers are connected with all the neurons that precede them, as well as with all those that follow them. Information is shared in this way, with background matrix-vector multiplications using parameters that can be trained and updated with the

help of backpropagation. Obviously, a network based on a series of dense layers will not provide the advantage that LSTM layers bring to keep in memory, the past information on the long term. A series of LSTM layers, with correctly defined parameters, will allow us to obtain accurate anomaly detection, taking into account the changes in trends over time. An LSTM network based on unsupervised learning will be able to learn in a relatively accurate way, all the existing correlations between the points of the time series. In this way, it will be more likely to be able to correctly reconstruct the input data when it is abnormal.

4.1.3 LSTM Autoencoder

It is possible to use LSTM networks to predict time series, but it is also possible to use them to reconstruct input values, based on an *encoder-decoder* operation, as represented previously. It is on the basis of this method that the tool developed in this project works. The architecture and the parameterization of the developed LSTM network allows to compress the input information before reconstructing it at the output of the network. The hidden layers have been parameterized in order to obtain a *"bottleneck"* in the center of the hidden layers. The first half of the network represents the encoder and the second half the decoder with the output of the input values reconstructed as accurately as possible.

It was possible to use this concept to teach the model to accurately reconstruct healthy input data, representing the vibrations behavior of a non-defective system. To do this, it was necessary to define in a complete data set, which part represents a healthy behavior. Obviously, it is not difficult to obtain a large amount of normal data from the complete life of a machine for example. Once the model has been trained, it is possible to send it any kind of data, healthy or not. The model will naturally reconstruct normal data correctly and will have more difficulty in reconstructing abnormal points. These abnormal values are not representative of the patterns learned by the model. On this basis, it is logically possible to use the absolute reconstruction error as an anomaly detector.

4.2 Python Keras Model

The realization of the deep learning model was done with the Keras library. Keras is a high level API that allows to build deep learning models in a fast and efficient way. In our use case, it is not necessary to use a lower level library. Moreover, using a high level library reduces the risk of encountering problems when converting the model. Nevertheless, Keras is linked to Tensorflow, in the case of more specific aspects, Keras can be combined with it in order to use a lower level of description.

We will see in detail later in this report that we use an updated prototype of HLS4ML [4] based on a CERN thesis [29] for the conversion of the Python model into an HLS model. This prototype was used on the basis of Keras and this is also one of the reasons why we decided to use this API.

4.2.1 Datasets

To carry out this modeling phase, it was necessary to work with real vibration data. Initially, the work was based on the data set provided by the contractor of this project. It is a complete dataset representing the life of a stimulated and prematurely aging bearing. This data set is represented graphically below.

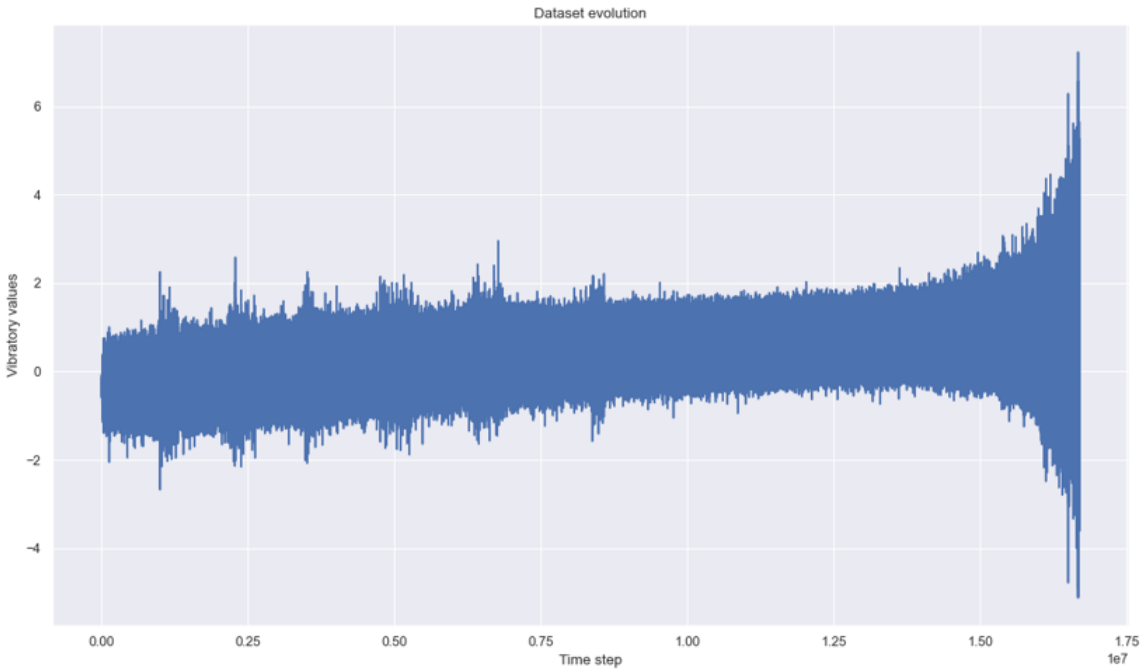


Figure 4.1 Bearing full dataset

These data were acquired with a 3-axis accelerometer ADcmXL3021BLMZ with a fixed acquisition frequency of 220kHz. To represent the above graph and to define our model, we used only the data from the X axis. We observe a data set with a large amount of data available. The evolution of these data is normally done with normal data going towards abnormal to aberrant data. We can also observe some noise going up to about half of the dataset. The origin of these noises is not known, but may come from different factors, such as work or door slamming, close to the acquisition. Nevertheless, these noises are probably not numerous enough for the model to retain them correctly. The model will naturally filter out these and retain mostly the actual behavior of the bearing.

One piece of information is missing with this dataset to complete the final model development. This is only raw data and we do not know where the actual anomalies are. It is possible to roughly estimate when the bearing starts to fail, but this is not enough to provide a quality model. It is necessary to use a dataset with known anomalies in order to evaluate the detection quality of the model. The use of this dataset was beneficial at first to see in a rough way if the principle is consistent and can work, but for the final adjustments, it was necessary to use another dataset.

The second dataset used is provided by Numenta Anomaly Benchmark (NAB). It is an open-source repository providing data and scripts to evaluate the quality of an anomaly detection algorithm. The details of NAB will be discussed in the verification and validation chapter. NAB offers multiple datasets, but we used one to observe and evaluate in detail the performance of the developed model in order to adjust it. It is a dataset representing the acquisition from a temperature sensor measuring a component widely used in industrial machines. The graph below represents the data set in question with the real abnormalities identified.

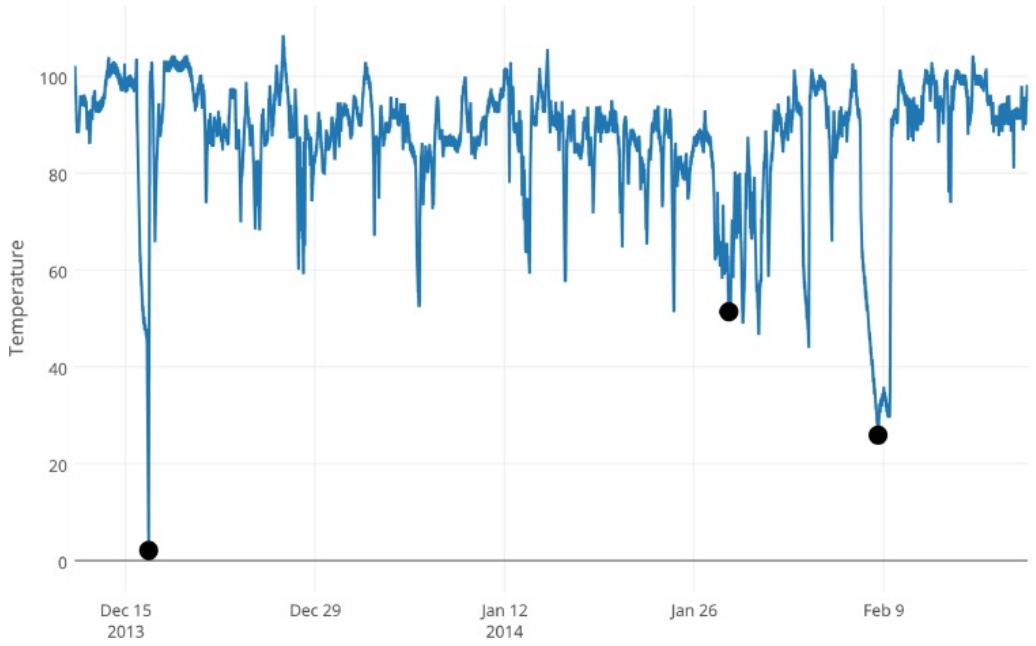


Figure 4.2 Numenta's machine temperature dataset[a]

[a] Extract from [1].

The X axis shows time stamps, and the Y axis measures temperature. The first anomaly represents a planned shutdown of the machine. The second anomaly is difficult to detect and is directly related to the third anomaly which represents a serious machine failure. There are a total of 22'695 points acquired from the temperature sensor. These points were acquired over 5 minute intervals.

Using this dataset with classified values, it was possible to observe whether our model can accurately detect these anomalies or not.

4.2.2 Data structure

The first step of the modeling phase, taking into account that real data are available, is to structure the data in such a way that they can be directly used afterwards for training the model as well as for its analysis and testing.

The realization of this step follows the process described by the following representation.

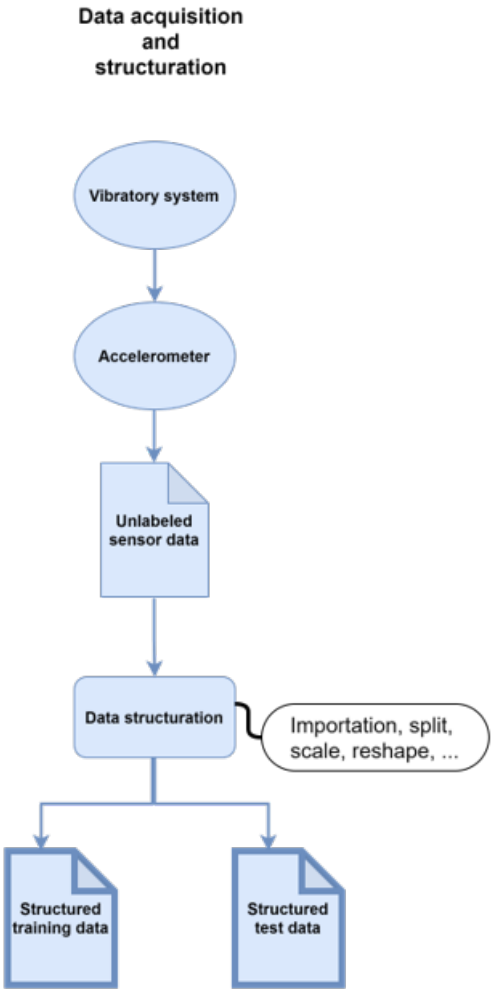


Figure 4.3 Data acquisition and structuration workflow representation

As an input to this step, we use the data acquired with the accelerometer on a vibrating machine. At the end of the data structuring phase, we obtain two data sets. This is a common operation in machine learning. The first set of data is defined for use in training. Our model will use this unlabeled dataset to learn from each input as best as possible. This dataset must be breast and therefore must not contain any anomalies. The second dataset will be used for the test phase of our model. It will be used to see how our model reacts when the data is clean and especially when data contains anomalies. It is therefore necessary to use a test set that contains known anomalies in order to evaluate the efficiency of our model in reconstructing the input data when it encounters anomalies.

4.2.3 Network architecture and configuration

As previously stated, the anomaly detector developed is based on the principle of autoencoder, it is therefore necessary to develop an architecture that goes in this direction. To achieve this, it is necessary to reduce the number of neurons on the encoder part in order to compress the input vector and to remove some information.

On the decoder part, it is necessary to increase again the number of neurons in order to be able to recover the information of the encoder and to reproduce as well as possible the input vector which is provided to it.

In our case, the hidden layers of the figure 2.10 are made of serial LSTM layers. The errors of reconstruction will be used as detector of anomalies. The quality of reconstruction and anomaly detection will be related to the configuration of the model parameters. If the model is properly trained, without being overfed and with enough data, it will succeed in reconstructing the inputs easily with small errors. If the model is poorly configured, it will detect anomalies where there may not be any and it will not detect some true anomalies. The model configuration is therefore a crucial step to obtain a quality model with an acceptable detection accuracy.

There are no existing methods or rules, which allow to get the optimal parameters for the need of a machine learning model based solution the first time. For this reason, multiple trials must be performed in order to choose the most satisfactory parameters. It is with this same way of doing that it is possible to obtain an architecture that corresponds to the needs.

Several important hyperparameters must be considered and tuned correctly.

Number of hidden layers

The hidden layers are the layers between the input and output layers. The number of hidden layers represents the number of layers used in the complete model network, between the input vector and the output layer. The more hidden layers there are, the better the model will learn during training. On the other hand, each layer will require additional resources, which can be problematic when we are limited at this level. For example, in the case of an embedded model.

Number of neurons (hidden units)

The neurons or hidden units represent the learning capacity of the model. Each layer has a number of neurons containing information that it will share with other neurons on other layers. The more neurons the network uses, the better it will learn. The combination of the number of layers and neurons is essential to define the learning capacity of the model.

Number of epochs

The epochs is a parameter that defines the number of times the model will be trained with a passage of information through the whole network. To define a sufficient number of epochs, it is possible to use the results of the cost function to minimize. When this function approaches 0 and becomes stable, it is no longer necessary to continue training the network. The cost function will be described below.

Batch size

The batch size defines the number of samples that will pass one by one in the network, during the learning process. A batch size of 1, defines that each input value will pass through the network in turn. This can be very time consuming when there is a large amount of input data. The number of batches in an epoch is defined by the number of input values divided by the batch size.

Again, a high batch size leads to many inputs passing in parallel through the network. This saves time, but also risks degrading the final quality of the model. Again, an optimal value must be defined for this parameter.

Dropout

The dropout is a parameter that prevents the model from overfitting. This parameter accepts values between 0 and 1 with a default value of 0. The more you increase this parameter, the more the model will randomly ignore a certain number of neurons (units). The other neurons will have to learn more to provide the expected results and in this way, this parameter allows to avoid learning too many neurons during the training. It is important to note that this parameter is to be defined for each layer. Moreover, an overfit can be detected when visualizing the evolution of the cost function during training. When the cost function decreases and then increases, this is a sign of model overfit. Another sign of overfit is simply that if we increase the learning capabilities of the model by increasing for example the number of hidden layers or neurons and the model has performances that continue to degrade.

The figure below 4.4 represents the cost function of an overfitted model. The orange curve represents the data used to validate this cost function and it is this one which is used as reference.

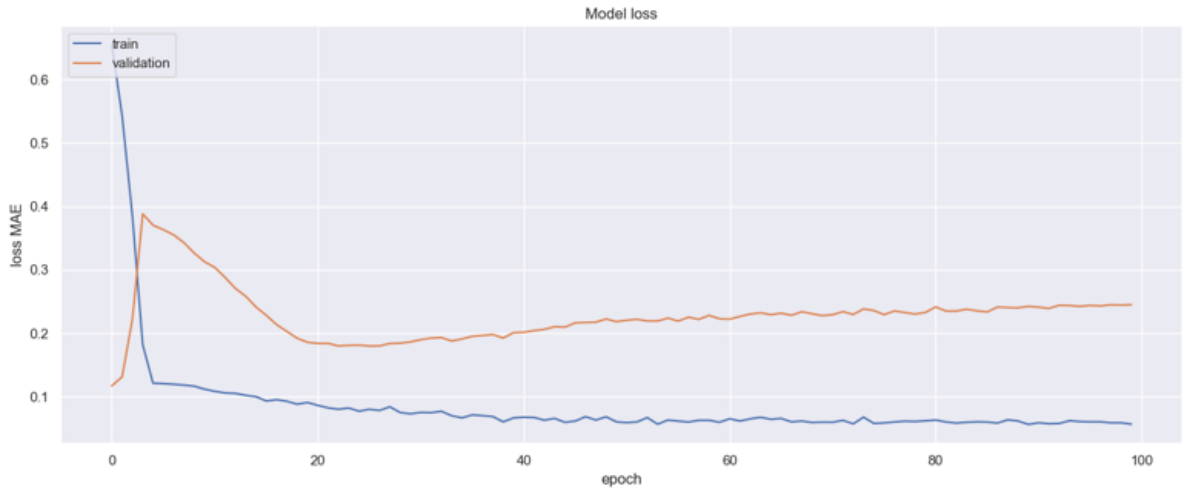


Figure 4.4 Loss MAE function example

Return sequences

This parameter can be either enabled or disabled. By default it is disabled. When enabled, it allows each hidden unit (neuron) to share its hidden state with all other hidden units. In this way the information is exploited at all levels in order to obtain a satisfactory result, especially when it comes to unsupervised learning. The patterns will be better taken into account.

Activation function

An activation function is a mathematical expression that allows to activate or not a neuron. Neurons whose inputs have no or little importance for the global process will be deleted for the sake of the model performance. In our case, we use the default activation function of the LSTM layers which is the *tanh* function.

Loss function

The cost function of a deep learninng network or loss function is the one we want to minimize during training. The cost function plays an important role when designing a deep learning model. It will be used to distil all the aspects of the model into a single value, which must represent the quality of the model. Again, the choice of the most suitable cost function is done by testing. Possible cost functions are for example the Mean Square Error (MSE), the Mean Absolute Error (MAE) or the Root Mean Square Error (RMSE). In the specific case of this project, the Mean Absolute Error (MAE) was a good reference.

There are many additional parameters, which allow you to customize a model as much as possible in order to achieve the desired performance. However, in most cases it is not necessary to change the default values.

4.2.4 Training

After defining the architecture and the configuration of the model's hyperparameters, it is now possible to train the model to reconstruct the input data with low errors. The model uses unlabeled data and thus performs unsupervised learning. It will highlight the important patterns between the data so that it can learn to reconstruct the input data in the best way on this basis. This technique generally requires a large amount of data to be trained.

The objective of this model is that it can accurately reconstruct the data used for training as well as the "normal" test data. On the other hand, it must have difficulty reconstructing specific points or windows of points that have anomalies. As a result, the model will be able to show a more important error that will stand out from the others and that can be considered as anomalies.

The precise description of the final model used for the rest of the project is shown below. Several architectures and configurations were tested before arriving at a final model that was validated in terms of resources and detection capacity in the section concerning the verification and validation phase. The most relevant of them are listed in the appendix O

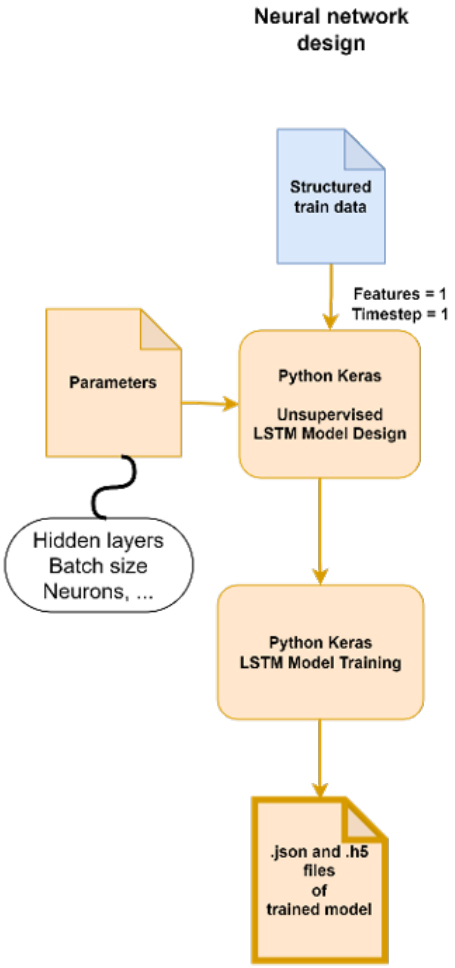


Figure 4.5 Training workflow representation

Once the model is trained, it is automatically saved in *.json* and *.h5* files, respectively for storing model structure and weights. The saved model will be used for conversion by the HSL4ML tool, but we will also be able to reload the model each time we need it for Python level analysis.

4.2.5 Train analysis

The training phase is a source of interesting information for the quality of the model and consequently the quality of its detection. We have seen in a first step that some information could be useful to determine some parameters, like the number of epochs or if it is necessary to add parameters against the overfit of the model. To do this, it is possible to visualize the evolution of the cost function to be minimized during the training.

Loss function analysis

With Keras, the basic method is to record the results of the fitting method in a plotter variable at the end of the training, like this with the variable *hist*.

```
        hist = model.fit()
```

Another way to do this is to use callbacks, which is an object that allows you to perform different actions at different times during the training phase. This can be very useful when the whole training takes a long time. With this method, it is possible to stop the training beforehand when the results are not satisfactory. It is possible to record specific logs that can be used with TensorBoard which is a visualization tool for machine learning. It is possible to configure the type of logs to be recorded in order to visualize the results of the training in different ways. To do this, it is necessary to use the callbacks library with the TensorBoard class. It is then possible to create an object to be used in the fitting parameters of the model, like this.

```
tensorboard_callback = TensorBoard(log_dir=log_dir,
                                   histogram_freq=1)

model.fit(callbacks=[tensorboard_callback])
```

With these two methods, it is possible to visualize the quality of the training of our model in order to get useful information. The first method is less flexible and precise than the second, but can be sufficient.

Reconstruction on training data

Another source of useful information can be applied on the basis of the data that were used to perform the training. We know that the training data are supposed to be healthy data, without anomalies. Therefore, our trained model is supposed to reconstruct correctly these data, except if there is some noise in small quantities that the model could not necessarily retain. Nevertheless, it is possible to reconstruct the training data by our trained model in order to calculate the overall error and plot the results. In addition to the analysis of the cost function, this is a good reference to know if the model has been correctly trained or not. In addition, we will be able to use the reconstruction errors of the training data, as a reference to define the threshold we will use for anomaly detection. This threshold will be used to classify the points considered as normal and abnormal. To do this, a good method is to plot the Gaussian distribution of the reconstruction errors. The rightmost value of the distribution is considered as a good reference for defining the threshold. The graph below represents an example of the distribution of training errors with the choice of the threshold value.

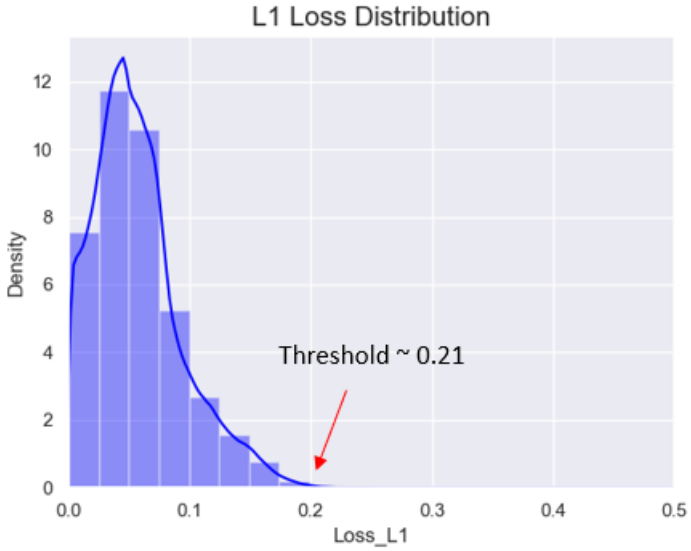


Figure 4.6 Loss distribution from training data reconstruction

We will see later that this method is relatively reliable. Obviously, it is not forbidden to adjust this threshold slightly after experience of using it.

4.2.6 Anomaly detection from reconstruction error

The reconstruction step is the equivalent of the prediction phase for a standard model trained to predict time series. This step consists in loading the trained model and testing it on the test data set. The goal is to reconstruct the test data and to evaluate the evolution of the reconstruction errors. It is interesting to first reconstruct the training data in order to analyze the errors. These are a good starting reference for the test data reconstruction phase.

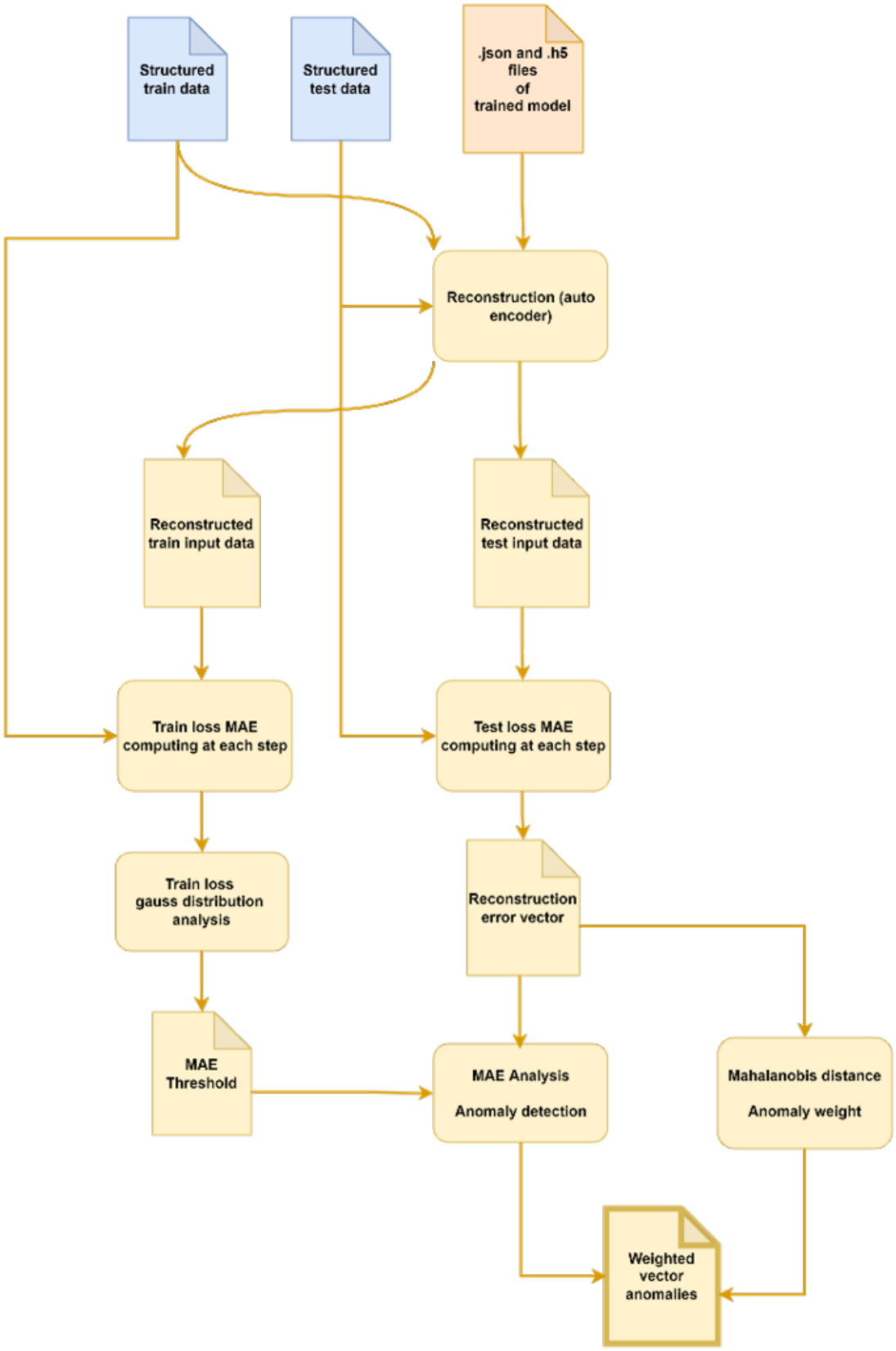


Figure 4.7 Anomaly detection workflow representation

Anomaly detection is based on the principle of classification. We have stated above that we work with a model based on unlabeled data. For anomaly detection, we will label each data over time into two categories, which are "anomaly" and "non-anomaly". To do this, it is necessary to define a detection threshold. As explained in the previous subsection, this threshold is based on the errors obtained during the reconstruction of the training data. If an error exceeds the threshold, the corresponding time point will be considered as an anomaly and if the error is below the threshold, the point will be considered as normal. This classification method is relatively simple and reliable since we compare real values to be evaluated with reconstructed values based on a model representing normal behavior. The figure below shows an example of anomaly detection on a time series from a public dataset that will be presented later in this report.

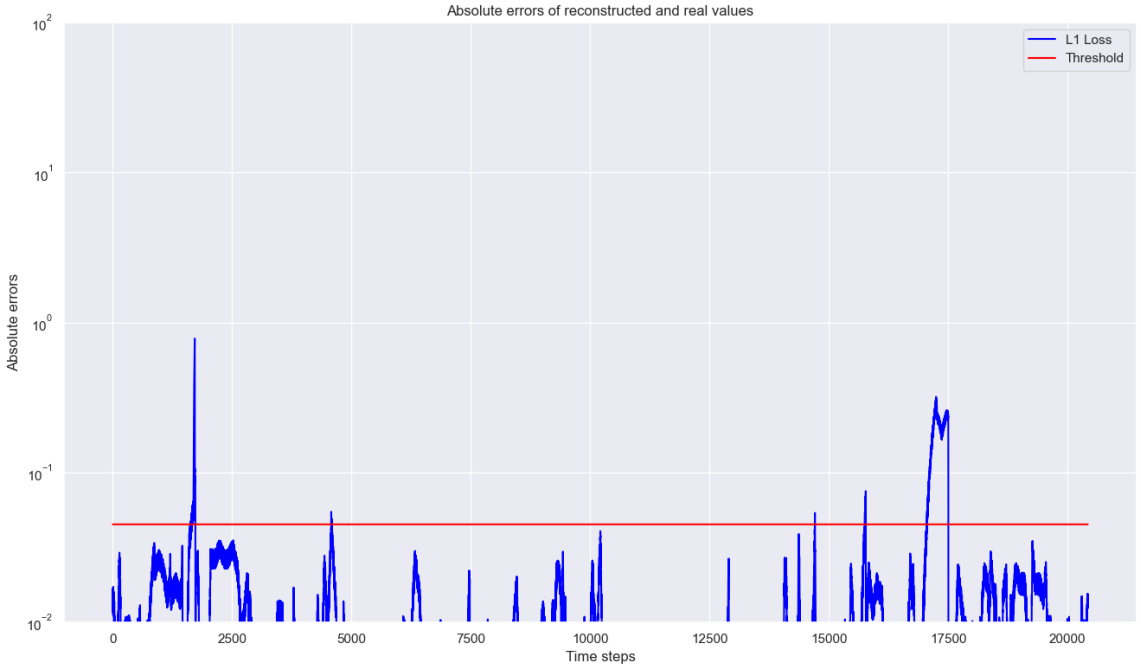


Figure 4.8 Graph of the absolute error on the test data of model n°28

The graph below represents the actual test data with the anomalies detected by the calculated errors and the defined threshold.

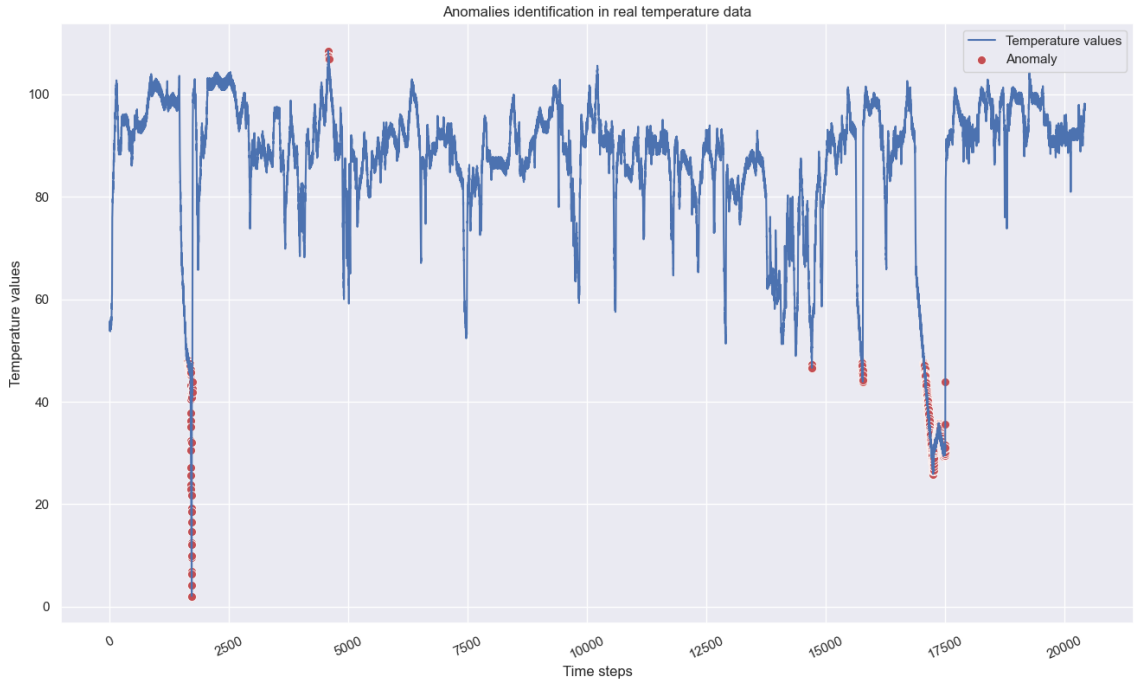


Figure 4.9 Graph of the test data of model n°28 with detected anomalies

As can be seen, we obtain a classification of data considered as anomalies (the red points) and non-anomalies. In order to perform a relevant and useful analysis for the future and for a concrete use, it is interesting to implement a technique that allows to define how important the anomaly is. We have therefore performed an additional analysis based on a statistical method called the mahalanobis distance, the concept of which was explained in detail earlier. This distance calculated for each point of the time series takes into account the correlation between the points. This means that if an anomaly is important, it will be represented by a high mahalanobis distance. Conversely, if the anomaly is just above the defined threshold, the mahalanobis distance will be small, because the correlation with the other points will still be important. The graph below represents the mahalanobis distance calculated on the basis of the anomalies detected previously.

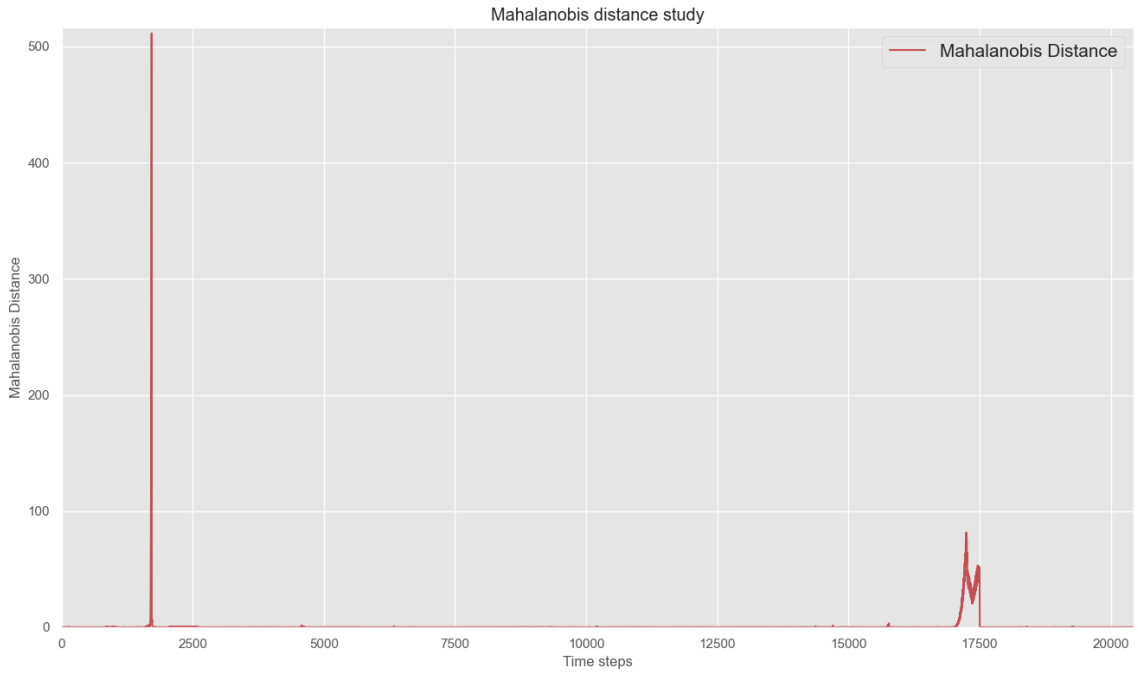


Figure 4.10 Plot of the mahalanobis distance of the model 28

With this method, we can weight the detected anomalies and add additional useful information for the analysis of a system. For example, anomalies with a low weighting can be neglected and those with a high weighting will be reviewed in more detail. In this example, we can notice that there are important anomalies that stand out at two places in the time series. The other anomalies exist, but are less important.

The model used as an example above is the final model that was chosen and defined as the most effective. This is the model that will be used for the next steps of the project. The results of this model will therefore be presented in detail in the following section.

4.3 Results

As for the results obtained in this modeling phase, it was possible to obtain a model that meets the different needs of the project. To achieve this, it was necessary to go through about thirty tests. Most of these tests led to unsatisfactory results in terms of detection or in terms of resources required. There was therefore no need to go further with them. Other models proved to be relatively interesting and it was necessary to decide between them during the verification phase with a scoring method which allows to give a quantitative note on the detection quality of a detector.

In this section, we will go into detail about the model that was finally chosen for the physical implementation of the project. The results presented below will be based on the use of the temperature machine failure dataset provided by NAB.

4.3.1 Global results

The table below shows some of the models trained with different configurations.

Trained models configuration										
Model	Hidden layers	Neurons	Epochs	Batch size	Dropout	Return sequences	Cost function	Activation function	NAB score (STD, FP, FN)	Remark
24	2 LSTM	8\16	200	64	0	True	MAE	tanh	50/50/67	Good results in NAB scoring
25	2 LSTM	8\16	100	64	0.1	True	MAE	tanh	45/37/63	
25.1	2 LSTM	8\16	200	64	0.1	True	MAE	tanh	44/37/63	
26	2 LSTM	8\16	200	64	0	True	MSE	tanh	34/17/56	
27	4 Dense	32/16	200	64	0	True	MAE	Sigmoid	48/46/65	
28	3 LSTM	16\4\16	200	64	0	True	MAE	tanh	43/47/65	Best results in NAB score using multiple datasets

Figure 4.11 Summary table of training with different configurations

The column that groups the NAB results is based on the use of a single dataset. We will see in the verification phase that by calculating the score on the basis of 3 datasets, it is model 28 that performs best and is the most reliable on different data. Moreover, we will see that there is not much difference in terms of required resources between model 24 and 28, despite the fact that model 28 has an additional layer. The final network configuration chosen is therefore model number 28. This one has the configuration shown below.

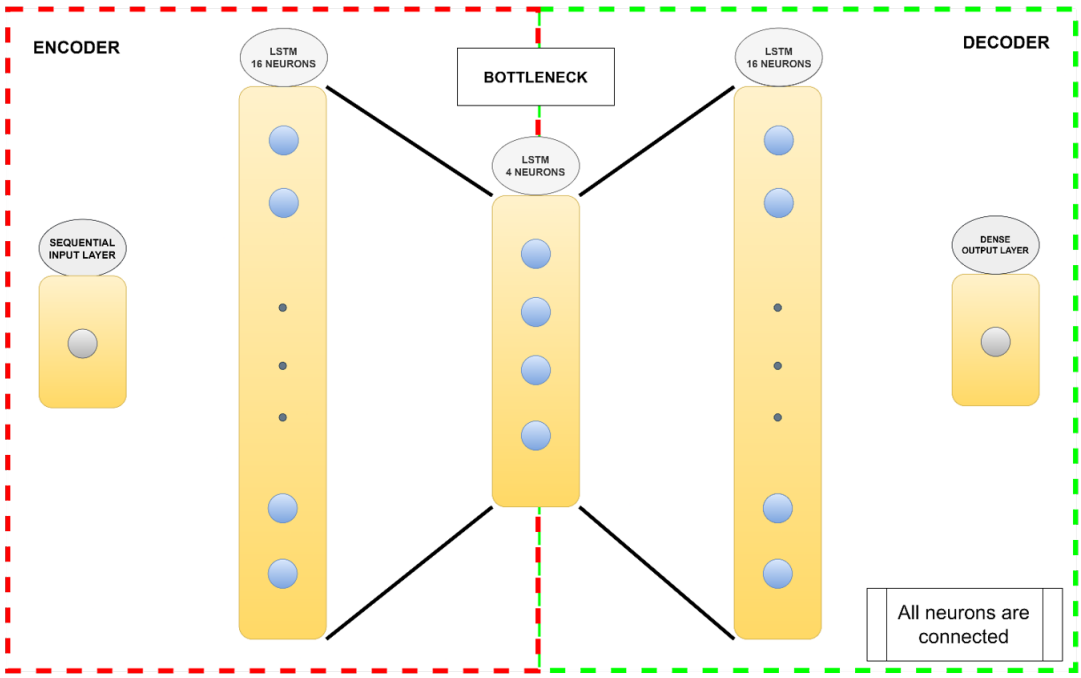


Figure 4.12 Model 28 architecture

The network has an input layer, 3 hidden layers, which are serial LSTM layers and an output layer, represented by a dense layer. On the 3 hidden layers, the first and the second represent the encoding part of the autoencoder. The second and the third represent the decoder part. By setting the middle layer with less neurons than the two others. This allowed to create a *"bottleneck"*, as we have seen before with the autoencoder representations. 16 neurons were defined on the outer layers and 4 neurons on the inner layer. For the fit of the model, 200 epochs have been defined, as it is enough to train the model. The batch size was set to 64, because it is possible that the tool is really trained with more data than the NAB data. If 64 leads to good results here, it will also be the case with more data and it will decrease the training time. The dropout of each layer remains at the default value of 0, as there is no overfit observed in the results and in the cost function graph that we will be able to visualize soon. The return sequences are activated in order to obtain better performance, as described previously. The compilation is based on a loss function of type Mean Absolute Error (MAE). Finally, the activation function is the default one for LSTM layers, i.e. the *tanh* function.

4.3.2 Detailed results for the selected model

The following steps demonstrate the results obtained by model 28 from the configuration presented above.

To begin with, during the data transformation phase, we had to break the data into two parts.

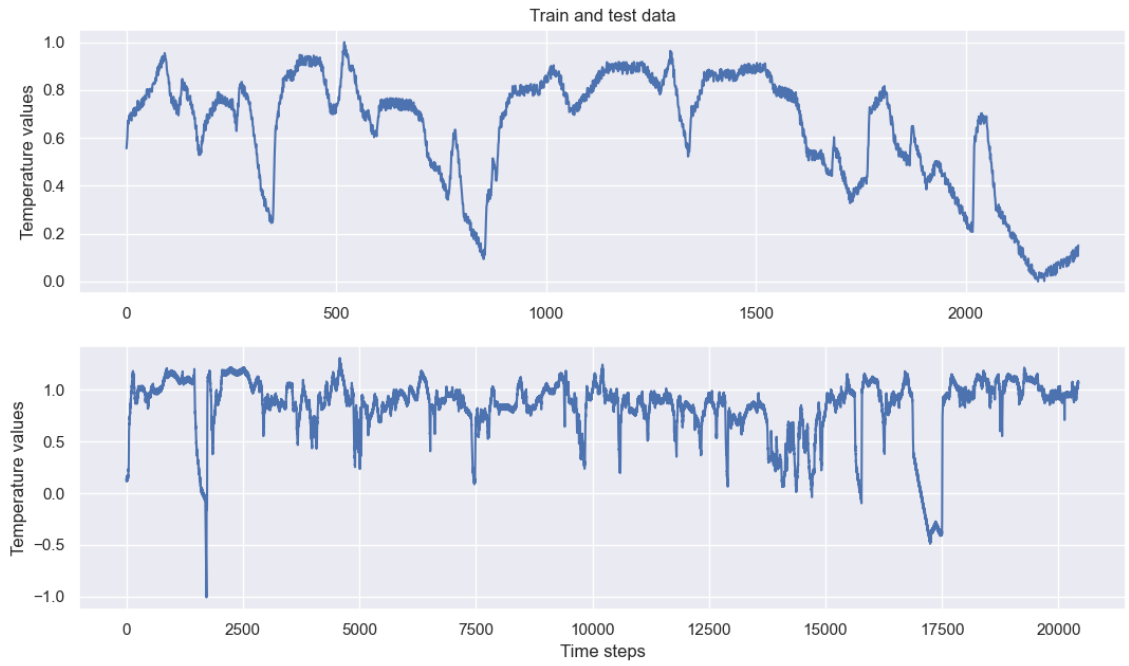


Figure 4.13 Split the dataset into training and test sets

It was defined to use the first 10 percent of the data points as reference data, i.e. without anomalies for use in training. With this dataset, we really know that this part has no anomalies since the three identified anomalies are located further in the time series. The complete dataset was therefore separated into two subsets. The top graph represents the training data and the bottom graph represents the test data. From this, it was possible to run the training of our model with the right configuration.

Once the training is finished, the analysis phase of the training can take place with the analysis of the cost function and the reconstruction analysis of the training data.

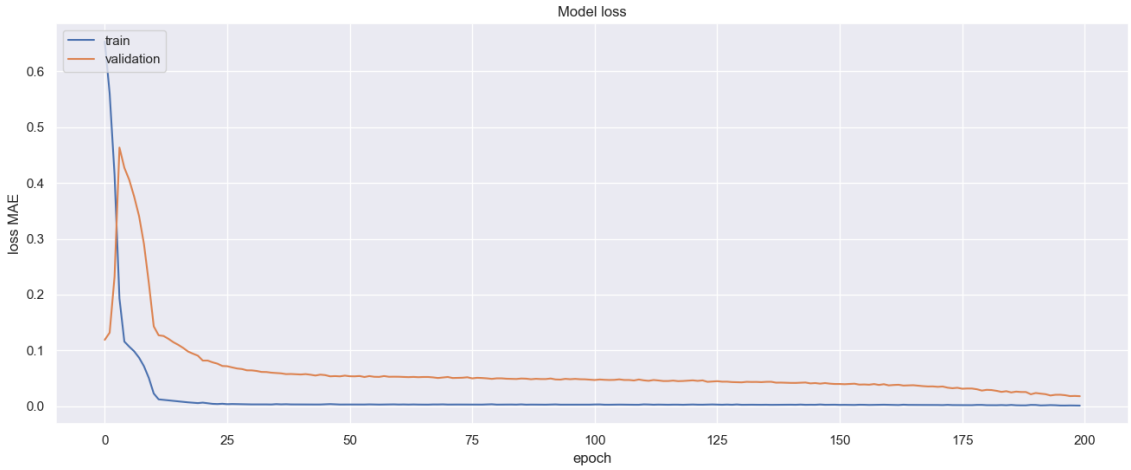


Figure 4.14 Loss function of the model 28

Looking at the validation data, in orange, it is possible to observe a stabilization of the function to minimize during a certain number of epochs. Around 150 epochs, the curve goes down again. In our situation, we stopped at 200 epochs where we can see a satisfactory result with a validation curve close to 0. Nevertheless, it is likely that by adding a few epochs, we arrive at something even closer to 0. However, 200 epochs is enough in our case. There is no need to add training time for little benefit. It is difficult to say how exactly the training behaves, because many factors come into play. Therefore, it is also difficult to provide the reason for stabilization before the curve continues to drop with a steeper slope. Nevertheless, we can also say that our model is not overfitted, because there is no deterioration of the function to be minimized. It only goes downwards.

The second analysis allows to define the threshold of detection of anomalies. To do this, the Gaussian distribution method of error values was used.

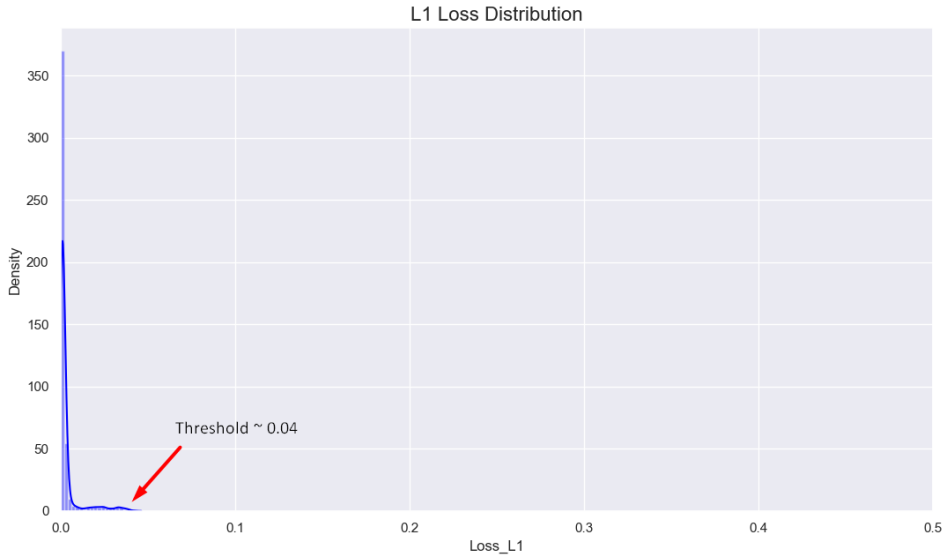


Figure 4.15 Loss distribution of the train data for the model 28

The reconstruction error of the training data is low. This was expected, since the reconstruction is done on the same data used for training. It was therefore possible to define approximately the threshold value for anomaly detection that will be used when reconstructing the test data.

Now that we have all the necessary training data available, we can proceed to the analysis on the test data. The reconstruction of the test data gives the following result.

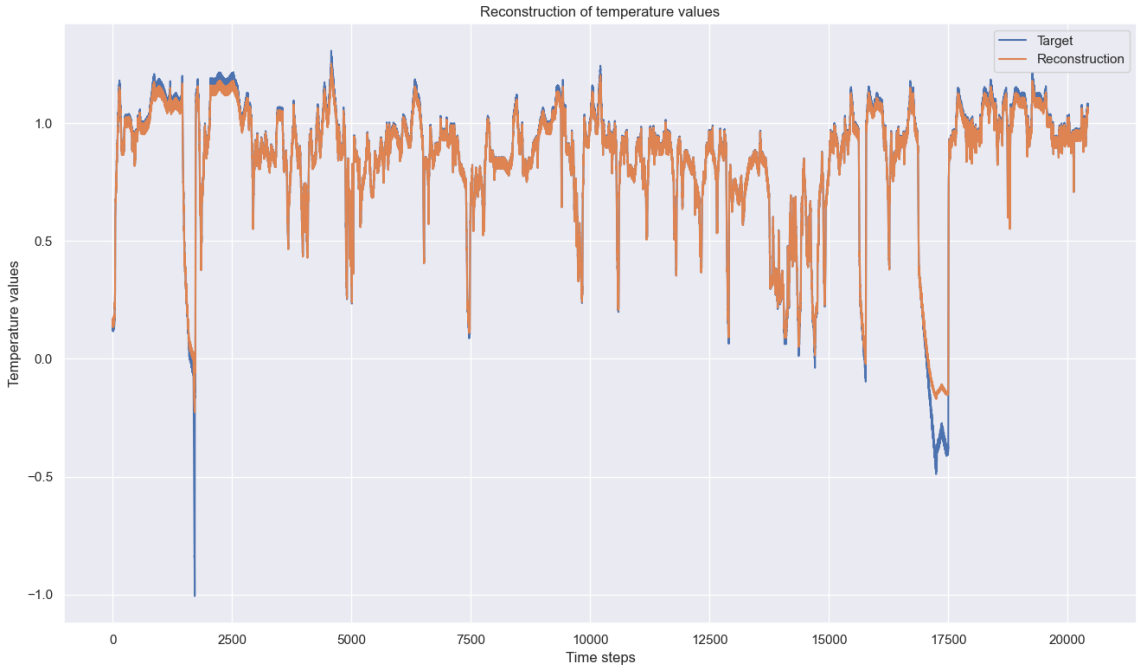


Figure 4.16 Reconstructed vs real values of the test data for the model 28

It is possible to notice visually that the model fails to reconstruct correctly the outlier anomalies. That is to say the first known anomaly and the last one, as indicated in the section presenting this NAB dataset. We know that there is a third anomaly between the two and that this one is relatively difficult to detect, as it turns out to be quite subtle.

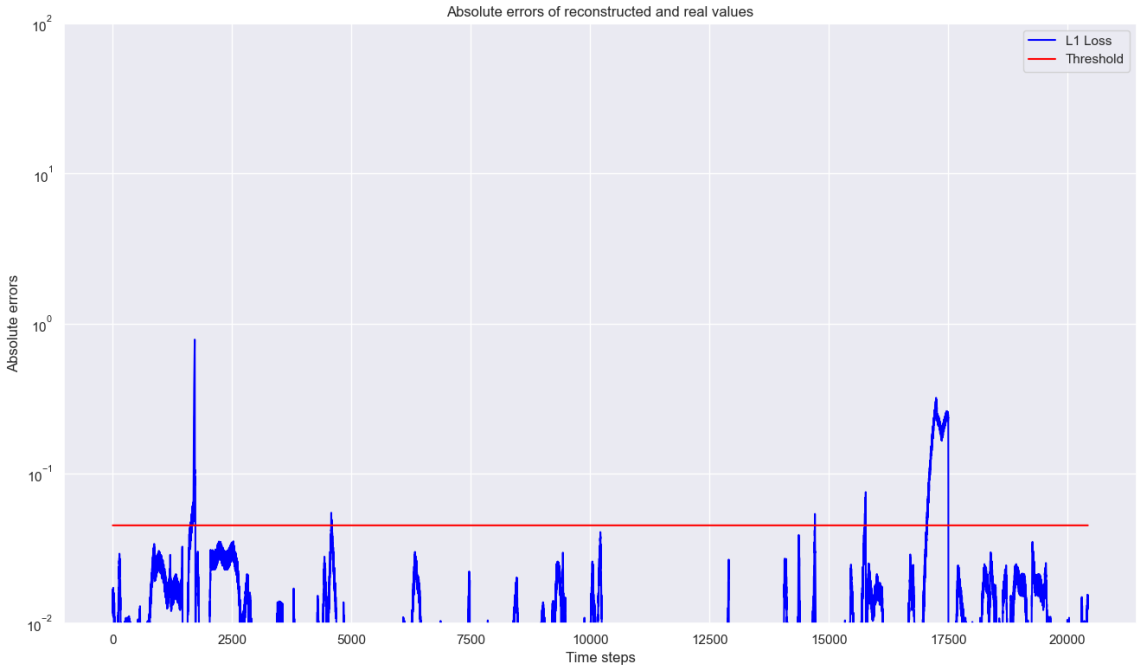


Figure 4.17 Absolute errors using test data in model 28

The graph above shows the evolution of the error between the real data and the reconstructed data. We can also notice, in red, the threshold defined previously, thanks to the reconstruction on the training data. All the errors that exceed this threshold will be classified as anomalies. We can already notice that the two most aberrant anomalies are detected. Nevertheless, we can also notice that there will be some false positives that will be detected at first. We know that the second anomaly, which is a bit more subtle, is the direct consequence of the last one. We can already notice that some errors exceed the threshold, before the last abnormality. It is therefore likely that this one has been detected.

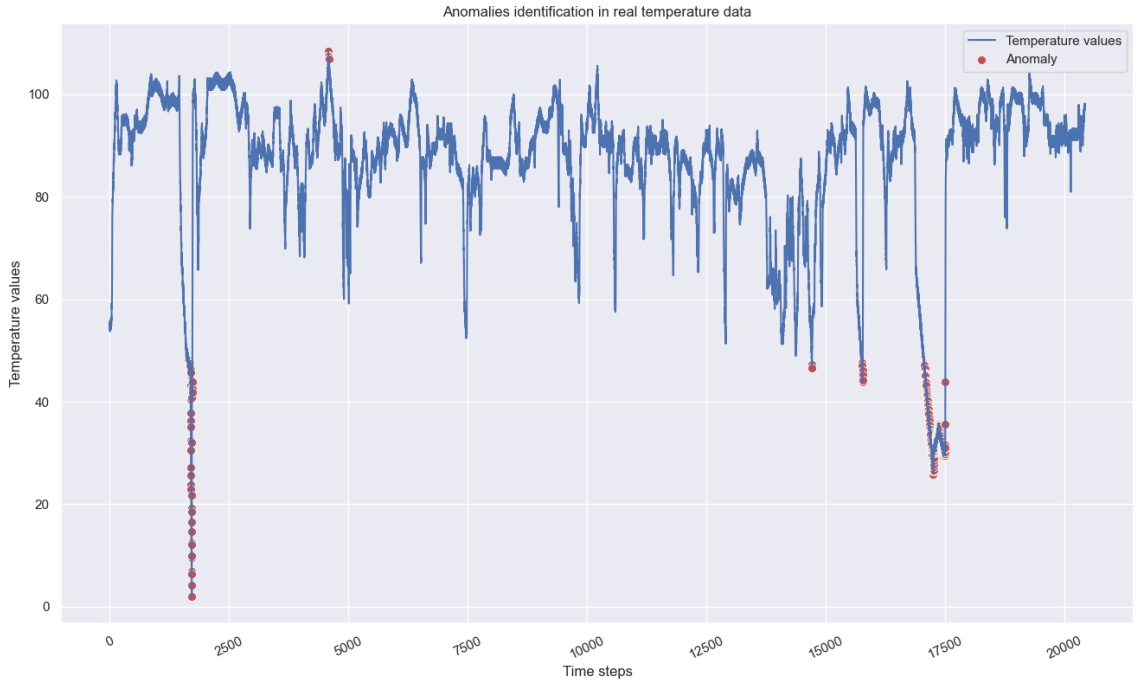


Figure 4.18 Graph of the test data of model n°28 with detected anomalies

This representation is more telling, because we can observe all the anomalies detected on the graph of the real data. In comparison with the graph composed of the real anomalies in figure 4.2, we can see that the detector we have developed, detects the two outliers. That is to say the first and the third anomaly. It is also possible to observe that the detector does not detect precisely the second anomaly, more subtle. However, it detects an abnormality close to this one. It can therefore represent a reliable sign of a future more serious anomaly, like the one we have last. We also notice that some false positives have been detected. It is not necessarily serious to detect some false positives if it is possible to weight the importance of each abnormality. If these false positives are detected with low weights, they will not be extremely significant in the decision of a predictive maintenance action. We will be able to weight the anomalies thanks to the mahalanobis distance.

From now on, data that were initially non-labeled are now labeled. In effect, we have points considered as anomalies and others as non-anomalies. An additional step to take into account the importance of an anomaly is the evaluation of the mahalanobis distance. This statistical study will attenuate the noise of the absolute error calculated previously.

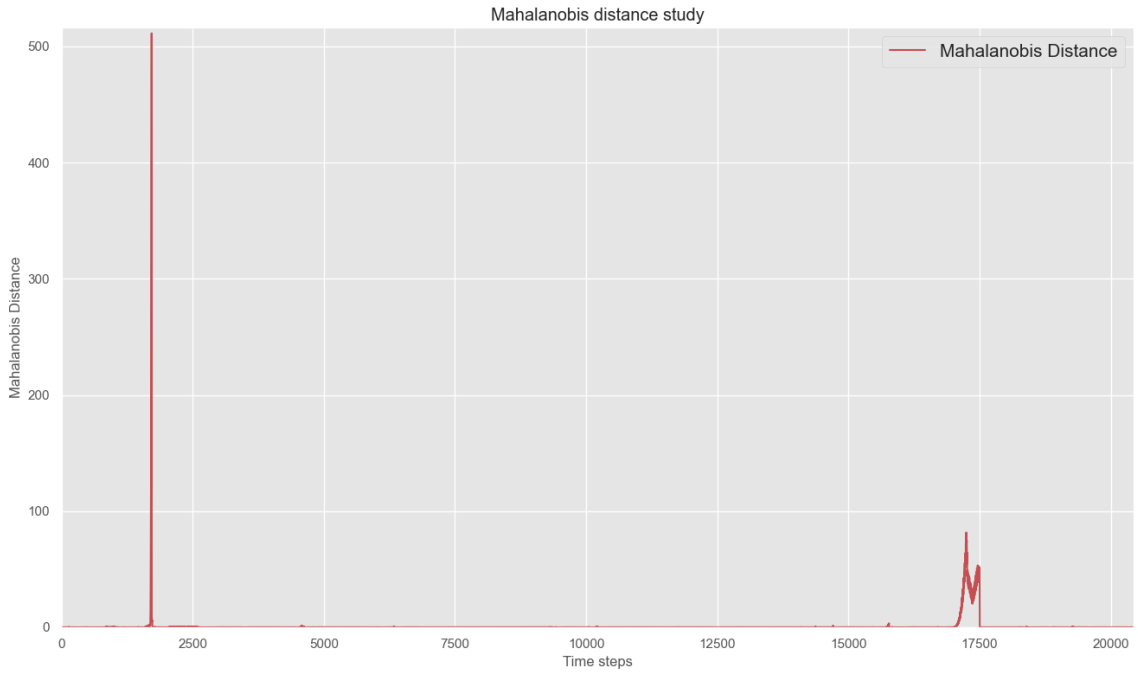


Figure 4.19 Plot of the mahalanobis distance of the model 28

It can be observed that the first and last anomalies, which are the most aberrant, stand out strongly on the mahalanobis distance plot. The other points are attenuated. The distance calculated for the previously detected anomalies represents their importance. The false positives do not stand out here and neither does the second anomaly. From now on, it is possible for the user, with the help of his experience, to judge which anomalies are to be considered and which ones can be neglected.

Finally, we have an anomaly detection model, based on an LSTM network, which works and detects real anomalies quite accurately. It is of course possible to adjust the detection threshold according to the experience gained with the tool developed in order to be even more accurate and reliable.

4.4 Conclusion

The use of a model that can be trained only on clean data is a significant advantage, especially when developing a tool to be marketed. It is not possible to require the customer to use data with anomalies in order to use the tool. Moreover, even if this were an option, it is very difficult to obtain anomalous signals in large enough quantities for the model to learn them correctly. This is the main reason why this strategy is suitable for our use.

There is no ready-made technique that will provide an optimal configuration with ideal results. That is why many different configurations have been tested on the basis of a dataset where real anomalies are known. We will see with the verification and validation step, by which means we could evaluate the quality of our models and why we chose one model over another. Obviously, in order to obtain a model that is close to perfection, we would have needed a long project only dedicated to the development of a machine learning model. So we had to define a satisfactory model and continue with it for the rest of the project.

The different aspects discussed in this section describe how the deep learning model has been developed to serve as an anomaly detector on time series. Details on the parameters to be configured and on the global strategy of the tool have also been discussed. Statistical concepts have been used to detect anomalies and to weight them. We have also described the datasets used for this procedure. It should be taken into account that it is difficult to evaluate the quality of a detector in a qualitative way by visualizing graphs. For this reason we will see in the next chapter that an open-source scoring method exists and allows to quantify the detection quality of an anomaly detector in a reliable way.

5 High Level Synthesis & Register Transfer Level conversion

This section is dedicated to the conversion / generation of code from a model written with the Python machine learning libraries, up to the generation of the RTL synthetizable model. Explanations about the steps taken before using the final solution which is HLS4ML are also available.

Contents

5.1 Machine Learning in RTL

To carry out this project, it is necessary to go through a relatively delicate and critical step in order to achieve it. This one is to obtain a Deep Learning model described in an HDL language so that it can be supported by a real time operation on FPGA. It is difficult and unsuitable to write a deep learning algorithm directly in a low-level hardware language, such as VHDL or Verilog. That's why different alternatives exist based on automatic code generation.

Tools such as MATLAB/Simulink, Vitis AI or PYNQ provide packages or processes that allow to switch from a high level language such as Python or Matlab with libraries adapted to Machine Learning to an HDL language. However, it is important to take into account that some layers are not supported at all or are supported only by some specific devices. For example, MATLAB provides a toolbox called "Matlab Deep Learning HDL Toolbox" which allows to quickly design deep learning models before creating a bitstream compatible with Vivado tools for example. On the other hand, Matlab allows to do this only with some deep learning layers and the LSTM layers are not part of them. This is also the case for the PYNQ solution. Vitis AI allows to integrate LSTM networks only in some boards that are part of the Alveo family. These boards are based on cloud computing and that is not what we are interested in here.

The incompatibility of these different tools, specifically with LSTM networks, is mainly due to the fact that it is a relatively complicated type of network to embed. As explained above, it is a type of network that allows to keep the past states to evaluate those at time t. This principle leads to a consequent increase in the resources required to run this algorithm. Obviously, in embedded systems, we are largely limited in terms of resources compared to computers for example. A lot of optimization work is necessary in some cases and the tools mentioned above have not found it necessary for the moment to make their tools compatible with LSTM networks. These networks are not as popular as CNNs, which are compatible with almost all solutions.

In this situation, we notice that the solutions are very limited and that it will be difficult to use an approved and reliable tool to develop an anomaly detector on time series based on an LSTM network. A solution could have been to use another type of network to achieve this. The study performed a comparison of different types of networks in terms of accuracy and resource consumption. The results are presented below.

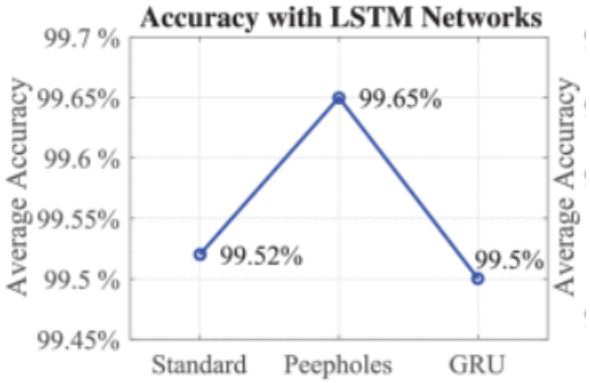


Figure 5.1 Accuracy graphics for LSTM networks on the Xilinx XCVU9P FPGA[a]

[a]Extract from [23]

Three types of LSTM networks have been evaluated, we see that they all have good results in terms of accuracy.

Neural Network	Accuracy	Time Consumption (μs)
RNN	99.23%	49.13
CNN1D	99.64%	122.72
LIBSVM	98.67%	1307718.02

Figure 5.2 Comparison of different neural networks accuracy on the Xilinx XCVU9P FPGA[a]

[a]Extract from [23]

The comparison is made with other types of networks, the RNN, CNN1D and SVM. Still in terms of accuracy, SVM is less accurate than the others. Only the CNN1D allows to obtain a satisfactory accuracy in comparison with the LSTM networks.

NN Type	BRAM	DSP	FF	LUT
Standard LSTM	84(1%)	3195(46%)	467834(19%)	521559(44%)
LSTM with Peepholes	84(1%)	3219(47%)	467693(19%)	521618(44%)
GRU	44(1%)	2290(33%)	325738(13%)	359103(30%)
RNN	84(1%)	1626(23%)	320651(13%)	316723(26%)
CNN1D	169(3%)	826(12%)	992523(41%)	1146756(96%)
LIBSVM	8(0%)	29(0%)	1453775(61%)	1181086(99%)

Figure 5.3 Comparison of LSTM and others neural networks resources utilization on the Xilinx XCVU9P FPGA[a]

[a]Extract from [23]

Then, in terms of resources, we see that the SVM and the CNN1D are very costly.

Finally, SVM networks are not adapted for this kind of problem, this one requires a large resource and gives the least good result in terms of accuracy. As for the standard RNN network, it cannot reach the accuracy of an LSTM. There is the famous problem of the gradient which does not follow when it comes to long-term estimation. CNN1D networks could be an alternative in terms of accuracy, but they will require a lot of resources.

During the realization of the model, we also tried to implement a network based on dense layers with the parameters found in model 27 of table in appendix O. It turned out that a more resource intensive network was needed to achieve the performance of an LSTM network, as can be seen by comparing the results below with the estimates for the Dense network and those presented below with the figure 5.10 for the final LSTM model.

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
20	20	0.100 us	0.100 us	1	1	function

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6	-
FIFO	-	-	-	-	-
Instance	48	779	8224	57125	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	36	-
Register	-	-	3114	-	-
Total	48	779	11338	57167	0
Available	624	1728	460800	230400	96
Utilization (%)	7	45	2	24	0

Figure 5.4 Model based on Dense layers estimation from Vivado HLS

A network based on LSTM layers is the best solution for our situation. For this reason it was advantageous to persist in finding a functional and satisfactory conversion solution. We will see below that the open source solution HLS4ML was the final solution chosen for the conversion of a deep learning model at the Python level into an RTL model, via a conversion at the HLS level.

5.2 HLS4ML framework

HLS4ML is an open source tool that gathers packages that allow to use a trained model for the inference phase of a machine learning algorithm on an FPGA through a conversion at HLS. This tool allows the use of standard machine learning libraries, such as Tensorflow, Keras or PyTorch for the design of the algorithm to be embedded afterwards. HLS4ML works according to the following representation.

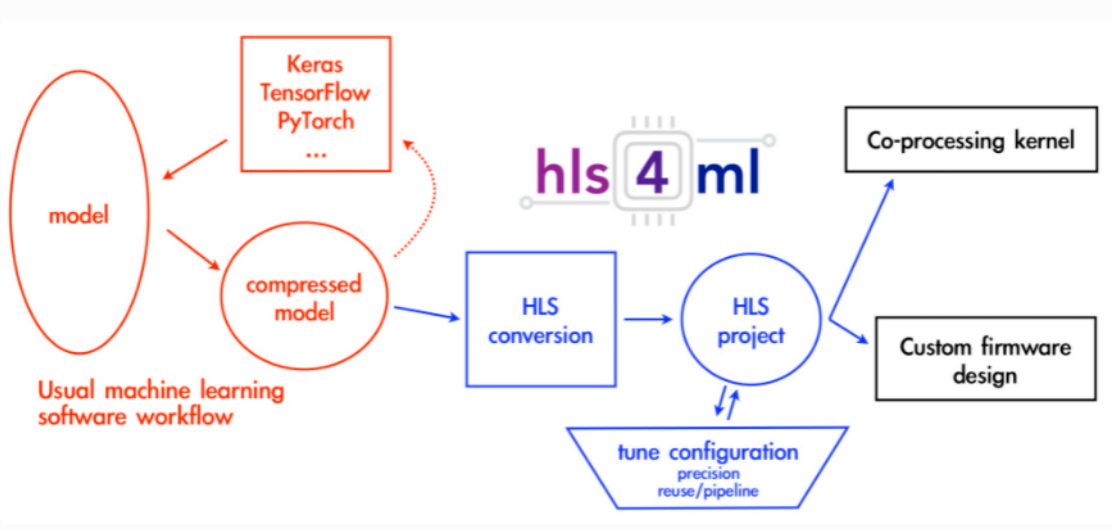


Figure 5.5 HLS4ML workflow[a]

[a] Extract from [20]

Based on a trained model and a configuration file, the tool will convert the Python model into an HLS model written in C/C++. It goes through compression and tuning steps for performance and resources, which can be customized. The goal is to obtain an HLS project that can be used with HLS to synthetizable RTL conversion tools, like Vivado HLS.

As for most of the tools mentioned above, HLS4ML does not officially support neural networks based on LSTM layers. However, since it is an open-source tool, it is a bit easier to find alternatives or versions that are not yet official and that support the conversion of LSTM networks. This is what was done for the realization of this project. We based our work on a GitHub directory [4] from HLS4ML, on which work is still in progress. The use of this directory allows us to obtain a relatively accurate conversion of an LSTM model written in Python Keras into an HLS model.

As this is an unofficial version of HLS4ML, you should not download the library via the pip or conda managers, as these will automatically use the official version and when it is necessary to convert it, incompatibility errors will appear. It is therefore necessary to clone the unofficial directory in question and work entirely on it. It is necessary to indicate where the HLS4ML directory that you want to use is located in `scripts` » `hls4ml`, by adding the following lines at the top of the code.

```
import sys
sys.path(<path-to-hls4ml-repo>/hls4ml)
```

Once this is done, there are no other adaptations to do to use the tool, except that instead of using directly the commands provided in the HLS4ML documentation with the prefix "hls4ml", we will indicate the path of the hls4ml file which is in */scripts/hls4ml* and use it as prefix instead. This way, we will be able to use all the commands provided by the official HLS4ML with the unofficial cloned directory.

To work efficiently, it was useful to create a new workspace directory in the hls4ml directory. It is from this directory that we have carried out the conversion steps of our model. This workspace directory contains :

- A second directory "keras" which contains the files *.json* and *.h5* generated at the end of the training of the model. It is also possible to provide in this same directory two files which will be used during the phases of C and RTL simulation for the stage of V&V. These files are a *.dat* file containing a column of input data and another *.dat* file containing the results obtained from our deep learning model at the Python level. Both files must logically contain the same number of lines.

- A file *keras-config.yml* which gathers the personalized parameters which will be used for the following stages. This file will be described in detail in the following section.

This workspace directory allows you to have a place to work to perform the conversion with HLS4ML. It is in this directory that the HLS projects to be used later in Vivado HLS will be created.

5.3 Configuration

After having designed the deep learning model, it is necessary to configure various parameters related to the conversion of this one in an HLS format. This configuration step is very important, because it is directly responsible for adjusting the compromise between performance, in terms of accuracy and latency, and the resources needed for the model to run on an FPGA. We will see that different parameters can be adjusted at this level. This configuration file in *.yml* format is filled in the following form.

```
KerasJson: keras/model.json
KerasH5:   keras/model.h5
InputData: keras/input_data.dat
OutputPredictions: keras/reconstruction.dat
OutputDir: kerasHLS
ProjectName: kerasHLS
XilinxPart: xczu7ev-ffvc1156-2-e
ClockPeriod: 5
Backend: Vivado

MaxLoop: 20

#options: io_serial/io_parallel
IOType: io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,8>
    ReuseFactor: 2
    Strategy: Resource
```

```
    Compression: True
  LayerType:
    Dense:
      ReuseFactor: 2
      Strategy: Resource
      Compression: True
    LSTM:
      ReuseFactor: 2
      Strategy: Resource
      Compression: True
```

The first part of this configuration manages the loading paths, project name, hardware reference and clock period. The second part is dedicated to the configuration of the customizable elements, which allow to define a certain quality of results for a certain amount of necessary resources. It is possible to define a specific configuration for the whole model and to customize the configuration of a particular type of layer for example. This configuration includes settings for precision, parallelization, the strategy to be favored, between latency and resource or even compression (pruning).

5.3.1 Optimization

Once the deep learning agorithm has been modeled and provides satisfactory results, it is still possible to perform some additional optimization steps that allow us to obtain a less expensive model in terms of resources, while keeping the performance obtained. Again, this becomes interesting when the resources available to run a model are limited, as is the case for this project. There are two main methods that can be used with standard machine learning libraries.

Quantization

The quantization method allows to reduce the precision of the calculations, i.e. by reducing the input values, the weights and the biases. By doing this, it is possible to considerably reduce the resources needed to perform the various calculations in the FPGA.

There are two main ways to apply quantization. The first is simply to reduce the accuracy after training the model. This is the method that was used to modify the computational accuracy of the model in this project. As for most embedded systems, FPGAs support fixed point values. For this reason, HLS4ML allows to define the desired precision in the configuration file in order to transform the values in fixed point form and with the defined precision. The representation below shows the syntax to adopt to define a conversion to a fixed point type with a certain precision.

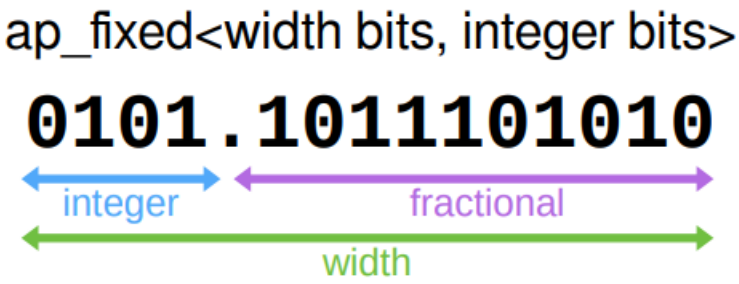


Figure 5.6 Precision configuration[a]

[a]Extract from [12]

The second method is to perform the quantization directly at the training of the model. In this way, we can directly obtain the quantified results from the trained model. This avoids validating the model twice, once with full accuracy and the other after reducing it. There is a library called QKeras which integrates very well with the standard Keras library in order to directly train a quantized model. Instead of using LSTM layers, QKeras provides the possibility to use QLSTM layers, with some different parameters, but the development principle remains the same. So it is a simple and very useful method. However, it was not possible to use the layers proposed by QKeras for our project. Indeed, since we use an unofficial version of HLS4ML, these specific layers are not supported. We were therefore restricted to using post-training quantization, as detailed above.

Compression

Another method of optimizing a neural network is pruning. It is an interesting method, which allows to reduce the size of a network with the resources necessary to use it without reducing its performance. Pruning is based on the deletion of synapses and neurons with weights close to 0 and therefore without much impact.

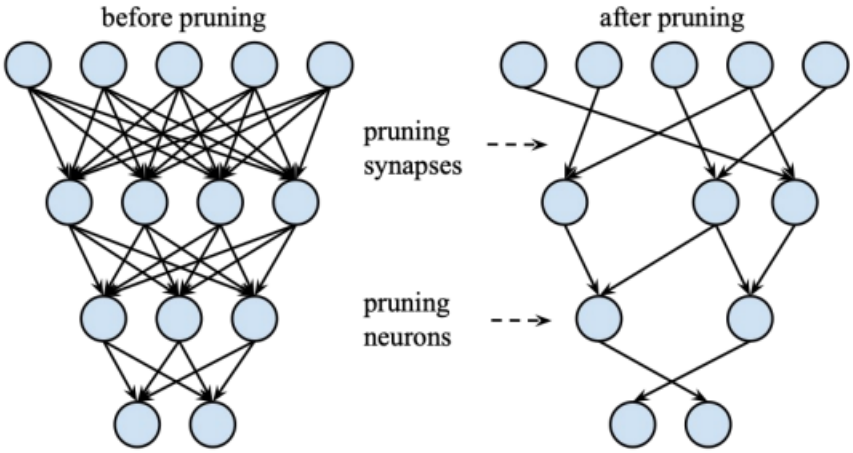


Figure 5.7 Pruning method representation[a]

[a]Extract from [12]

This way of optimizing a neural network can be implemented directly with tools like Tensorflow for example. However, since in our case, we are using a conversion using the HLS4ML directory, it will be possible to take advantage of this tool which is able to directly compress a model that we will provide it. To do so, you just have to activate the "compression" parameter in the yaml configuration file.

Parallelization

In the HLS4ML configuration file, it is also possible to parameterize the level of parallelization of the calculations in each layer of the network with the "reuse factor" parameter. Each layer performs matrix vector multiplications from its neurons. The reuse factor indicates how many times we want to reuse a multiplier to perform these calculations. So, this parameter allows to indicate if we want to have a minimal latency, with the calculations being done in parallel and with a higher consumption of resources or the opposite, if the latency is not very important, it is possible to increase the calculations being done in a sequential way in order to decrease the resources necessary to carry out them.

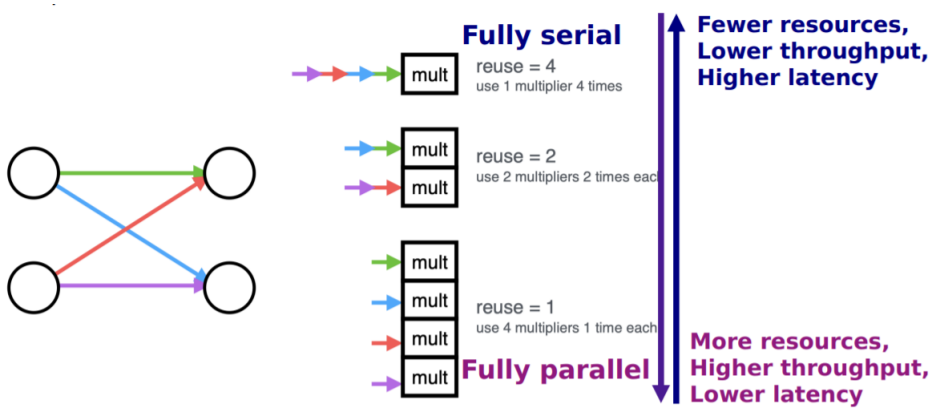


Figure 5.8 Reuse factor representation[a]

[a] Extract from [12]

The higher the value of the reuse factor parameter, the more the calculations will be performed in series. Conversely, if the reuse factor is low, with a minimum value of 1. Obviously, it is necessary that the "strategy" parameter is in agreement with the choice of the reuse factor. It is not possible to set a latency on strategy parameter with a high reuse factor. It is mandatory to set the reuse factor to 1, if you want a latency strategy.

5.3.2 HLS4ML C Test Bench

The HLS4ML directory also provides a test bench written in C that can be used during the model conversion verification phase. It is directly part of the HLS4ML directory and to use it, you just have to provide the location of two *.dat* files. One file lists the input values used by the model to be tested and the other file lists the reconstruction results obtained at the Python level. If these files are not provided, HLS4ML will create them automatically so as not to block the conversion process. Obviously, in this case, the

files created will be empty and will not be used to test the conversion. To automatically generate the *.dat* files required by HLS4ML, a script in appendix M has been designed during this project. You just have to call it with the following command.

```
python save_data_tb.py
```

We will see in the verification and validation step, that this same test bench written in C will allow to verify if the conversion of the model at C level gives the same results as the conversion of the model at RTL level.

5.4 Synthetizable RTL code generation

This section, details the aspect of converting the Keras model into an HLS model. It is then possible to build this model into an HLS project that can be supported by Vivado HLS. At the end of this step, we obtain a sythetizable RTL project ready to be exported into a reusable IP by Vivado. We will also get an approximation of the resources that will be needed to use this design as well as the estimated latency. If we reference an FPGA with too few resources in the configuration file, Vivado HLS will point out that the FPGA is not suitable for the design.

5.4.1 Keras to HLS conversion

Once all the configurations are done, it is possible to launch the conversion of the Python model into an HLS model written in C/C++. This conversion is done thanks to templates established by HLS4ML. The operations specific to each type of network are pointed to files grouped in the directory hls4ml » templates » vivado » nnet_utils. The one that allows the use of LSTM layers, in the unofficial directory, is the file *recurrent.h*. Despite the fact that we are using an unofficial version of HLS4ML, it is possible to use the options available in the official documentation. As mentioned before, instead of using the prefix *hls4ml* after a normal download of the library, you should target the file scripts » hls4ml in the cloned directory as prefix.

To convert a python template into an HLS project, just use the following command from the *workspace* directory.

```
python ../scripts/hls4ml convert -c keras-config.yml
```

This command will launch the conversion by flagging the configuration file *keras-config.yml*, which groups the configurations presented previously. At the end of this, we obtain a new directory in *workspace*, with the name we gave it in the configuration file. This directory contains the different HLS files established in relation to the python model provided, the *nnet_utils* files, the test bench, etc. It also contains the tcl instructions necessary to build the HLS project in order to use it directly in Vivado HLS.

The new generated directory has the following structure.

```
|- my-hls-test
   |- firmware
   |  |- weights
   |  |  |- All weights and biases per layer
   |  |- nnet_utils
   |  |  |- All relevant algorithm header files
   |  |- parameters.h
   |  |- myproject.h
   |  |- myproject.cpp
   |  |- defines.h
   |- vivado_synth.tcl
   |- myproject_test.cpp
   |- hls4ml_config.yml
   |- build_prj.tcl
   |- tb_data
      |- Input data can be given to the testbench from here
```

Figure 5.9 Output directory structure[a]

[a] Extract from [29]

5.4.2 Vivado HLS project

To build the HLS project, the command *vivado_hls* must first be recognized by the bash or dash terminal and to do this, you must source the file *settings64.sh*, which is found in the Vivado HLS installation file. In order for the command to be available, even after the computer has been restarted, simply add this line to the *.bashrc* file.

```
source ~/<path-to-vivado-hls>/2019.2/settings64.sh
```

When the terminal is restarted or reloaded with the following command

```
source .bashrc
```

It will be possible to use the command *vivado_hls* from the terminal. This will allow us to build the HLS project from the tcl instructions with the following command.

```
vivado_hls -f build_prj.tcl
```

Written like this, this command will simulate at C level, synthesize, co-simulate at C/RTL level and export the HLS project with all default values. To be able to control all these steps in more detail directly in Vivado HLS, just add the following options to the command.

```
vivado_hls -f build_prj.tcl "csim=0 synth=0 cosim=0 export=0"
```

Only the HLS project compatible with Vivado HLS will be created and will be able to be opened from Vivado HLS in order to perform the simulations, the systhesis and the export with the interface and the options offered by the software.

To control only certain operations from within Vivado HLS, it is possible to manipulate these options in a custom way.

```
vivado_hls -f build_prj.tcl "csim=1 synth=1 cosim=1 export=0"
```

In this project, we regularly used the above configuration. The C simulation, the synthesis and the C/RTL cosimulation will be done automatically with the default settings. The export will not be done automatically, so it is possible to choose the format of the export from Vivado HLS, this one is *IP catalog* in order to obtain a reusable IP format in Vivado. For the synthesis, it is done by default with the Verilog language. It is possible to use the option *synth=0* and to carry out the synthesis directly from Vivado HLS in order to choose the target hardware language, like VHDL. It should be noted that if the automatic synthesis is not activated, the cosimulation will not be possible.

5.5 Results

This section concerns the results obtained during the conversion step of the Keras model chosen and described in the previous chapter. The configuration file, the estimated performances and resources after the conversion as well as an analysis of the accuracy obtained at the C and RTL level are presented.

5.5.1 Configuration

The specific configuration for the chosen model is as follows.

```
KerasJson: keras/model.json
KerasH5:   keras/model.h5
InputData: keras/input_data.dat
OutputPredictions: keras/reconstruction.dat
OutputDir: kerasHLS
ProjectName: kerasHLS
XilinxPart: xczu7ev-ffvc1156-2-e
ClockPeriod: 5
Backend: Vivado

MaxLoop: 20

#options: io_serial/io_parallel
IOType: io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,8>
    ReuseFactor: 1
```

We targeted the reference of the FPGA integrated in the Zynq UltraScale+ MPSoC ZCU104. The clock period was left with the default proposal, which is 5 ns. Latency has been favored with a reuse factor of 1 and thus a default setting *strategy* in "latency" mode. In terms of resources, we will see that we are already very low compared to what our equipment can support. Therefore, our model has a margin of maneuver which allows us to further decrease the resource consumption to integrate our model in a cheaper FPGA. Obviously, the current latency will be impacted, but since it is already

very low and with the need we have with our tool, this impact will not be problematic. Regarding the chosen precision of *<16, 8>*, we will see in the subsection concerning precision that it is the precision that generates low and stable errors. In our case, the precision is important for an acceptable detection quality. Therefore, it is not possible to favor resource consumption over precision in the development of this tool. It is necessary to obtain an accuracy that is as close as possible to what we had in Python.

5.5.2 Performances and resources

At the end of the synthesis phase of the HLS model into an RTL model, Vivado HLS automatically proposes an estimate of the model's performance in terms of latency. More importantly, it also provides an estimate of the resources required to run the model. It compares this with the resources available in the target SoC, indicating percentages of usage as well as an indication when it is not suitable to support the generated design.

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
32	32	0.160 us	0.160 us	1	1	function

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	8	-	-	-
Expression	-	-	0	6	-
FIFO	-	-	-	-	-
Instance	90	337	8647	44745	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	36	-
Register	0	-	1721	128	-
Total	90	345	10368	44915	0
Available	624	1728	460800	230400	96
Utilization (%)	14	19	2	19	0

Figure 5.10 Vivado HLS estimations for model 28

The above estimates are from the final model presented earlier and the hardware reference used is the XCZU7EV-2FFVC1156 MPSoC implemented on the Zynq US+ ZCU104 used.

The configuration that leads to these results is the one presented in the previous subsection. It is possible to observe that the equipment used is largely over-dimensioned in relation to the needs of this project. However, the aim was not to use an optimal equipment, but to prove the concept with a sufficient hardware equipment to avoid some saturation problems. The estimated values in terms of resources are provided by the Block RAM (BRAM), DSP, Flip Flop (FF) and Look Up Table (LUT). This is the means used to define the resources of an FPGA in general. These different estimates would allow the model to be integrated into a cheaper SoC, such as the Zynq 7015. It would still be necessary to lower the share at the DSP level. In our case, it would be possible to do it by increasing the reuse factor, which will not affect the performance of our tool, because as it is possible to observe, the latency estimated at a few hundred nano seconds, which is very low.

5.6 Conclusion

As a result of this step, we were able to obtain a deep learning model translated into a HDL that can be synthesized and reused by Vivado for its physical integration. To do so, we first translated the Keras model into an HLS model written in C/C++. This allowed us to build the HLS project usable in Vivado HLS with a series of tcl instructions. The HLS project could then be used in Vivado HLS in order to parameterize the different steps, until the export of the project in IP format. The results obtained for the chosen Keras model could be presented. After the conversion, it was possible to obtain an estimation of the resource consumption of the RTL model, as well as an estimation of the performance in terms of latency. We have seen that these were largely satisfactory for the chosen target FPGA. It would be possible to integrate this same model in an FPGA with much less resources available and therefore cheaper. We also analyzed the accuracy obtained after the quantization of the Python model and it appeared that this one does not deteriorate or very little the quality of reconstruction obtained at the Python level.

6 | Verification & Validation

The verification phase of the deep learning model is an important step for the quality of the project. This step was carried out in three phases.

The first one consists in efficiently evaluating the performance of the anomaly detector at the Python level. Once the reconstruction results have been validated at the initial level, it is important to make sure that the conversion at the HLS level is done correctly and that the results are close to those obtained during the verification at the Python level. The third step is identical to the second one, but this time we check the quality of the conversion from the HLS model to the RTL model. We will use the same input set for each level.

Contents

6.1 Python level

The verification and validation of the final model at the Python level is the main step of this part of the project. If the Python model is satisfactory and meets the requirements of the project, it will simply verify the quality of the conversion up to the RTL model which is the lowest level. Therefore, it is important to use a reliable and quantitative method that clearly represents the performance of an anomaly detector. To do this, the NAB (Numenta Anomaly Benchmark) scoring mechanism was used.

6.1.1 Numenta Anomaly Benchmark

Numenta Anomaly Benchmark (NAB) is an open-source tool available under github that allows to evaluate the quality of an anomaly detection algorithm. The tool provides many datasets where anomalies are labeled. Thus, the scoring mechanism can evaluate the detection quality of the detector under test (DUT).

The mechanism works on the principle of true positive (TP), true negative (TN), false positive (FP) and false negative (FN) classification. Anomalies labeled as true positive by the detector increase the NAB score. Decreases in the score are caused by false detection (FP and FN).

The tool will also score the detection quality based on windows considered as abnormal, i.e. containing one or more anomalies. These windows allow to give the benefit of the doubt to the detector under test. If the detector detects real anomalies in these windows, the mechanism will offer more or less points depending on the accuracy compared to the real anomaly point.

We used this analysis tool in the following way.

1. **Training and reconstruction from NAB dataset**

   In a first step, we used a provided dataset where the anomalies are known. This set represents the temperature evolution of a machine acquired by a sensor. The type of signal of this data set is quite representative of a vibratory signal, which allows to start on a good reference. We have trained several of our models on 10% of the dataset and we have generated the results of the reconstruction.

2. **Generation of results in a specific CSV format**

   An automated Python program has been developed in appendix L to output the CSV file in the correct format.

   The NAB format is built on the basis of 4 columns.

   (a) timestamp

   (b) value

   (c) anomaly_score

(d) label

The *timestamp* column is identical to the one in the CSV files provided by NAB. The *value* column represents the results obtained during the reconstruction. The *anomaly_score* column contains binary data where 0 represents a value that is not considered as anomaly and 1 represents a value labeled as anomaly. Anomalies are detected based on absolute reconstruction error values exceeding the defined threshold. The last *label* column also contains binary values representing this time anomaly windows. These windows are defined by the json file provided by NAB and we have transferred them to the final CSV format.

3. **Create a new detector structure in the NAB repository**

   To use the open-source tool NAB, it is necessary to clone their directory locally. From the main directory, it is possible to run the following command.

   ```
   python scripts/create\_new\_detector.py --detector
   LSTMdetector
   ```

   This allows you to obtain all the instances and files necessary for the new LSTM detector to be tested.

4. **Launch the NAB scoring mechanism**

   Once the LSTM detector has reconstructed the input data provided by NAB and once the new detector has been created in the NAB directory, it is possible to run the NAB scoring mechanism to obtain quantitative results that represent the detection capability of the detector under test. To do this, it is first necessary to place the result file(s) in CSV format in their respective folders under results > LSTMdetector

   ```
   python run.py -d LSTMdetector --score --normalize}
   ```

The overall representation of using Numenta Anomaly Benchmark to verify and validate the Python model looks like the figure 6.1.

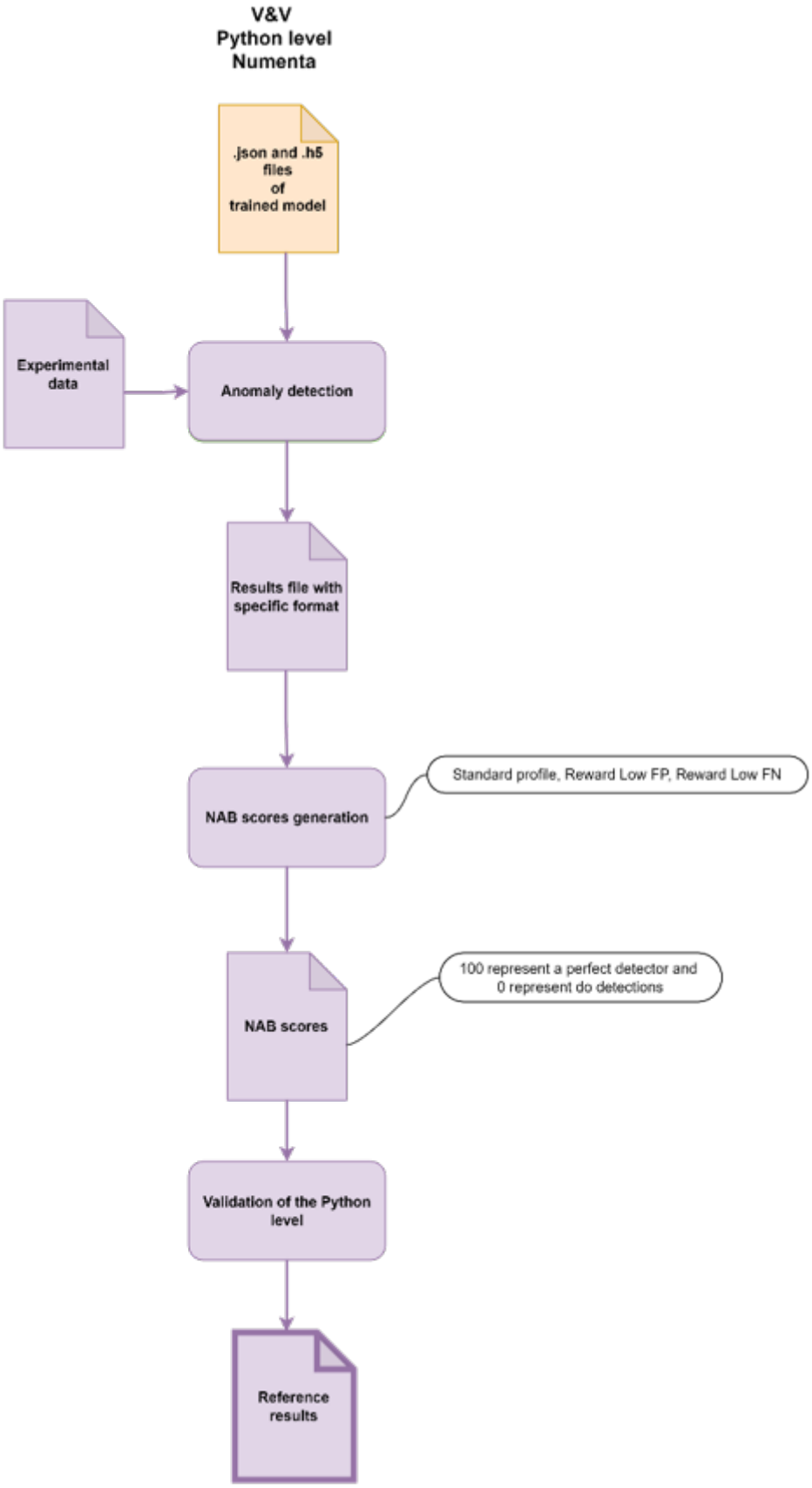


Figure 6.1 NAB process flow representation

After using this method, it is possible to quantify the detection quality of our model on the basis of 3 different scenarios to be favored, i.e. a score that rewards a minimum of false negatives, another one that rewards a minimum of false positives and another one that does not weight anything and puts everything on the same level. For example, in this way, if when designing our detector, we know that detecting false positives is not serious, we can accept a lower score for the weighting concerned. Obviously, the ideal is to get a score that is close to 100 on all 3 scenarios.

6.1.2 Results

It was possible to use this NAB tool in this project. It was a precious help for this phase of verification and validation of the model and thus in the choice of the most appropriate detector. Indeed, the evaluation of the model at the Python level was essential and we had to find a way to quantify this to validate a model to be used during the rest of the project. For this step, we also tested several models with NAB in order to choose the most appropriate. However, in this section, only the results of the final model chosen will be presented. That is to say the results of model 28.

It should be noted that in a first step, the different models were evaluated with NAB only on the basis of a data set. This allowed us to sort out and to highlight a more interesting model than the others. Then, this model was tested with 3 datasets in order to confirm its result and its robustness when used on several types of data.

Before presenting the NAB scores obtained for the Model 28, here is a classification table available in the NAB directory. It groups different types of anomaly detectors that have been tested with the NAB method. The ranking is based on the scores obtained for each detector. This will give us an idea of where our detector would rank in this table and therefore, if the method and configuration used are effective. It should be noted that we are using a detector based on the principle of the autoencoder using an LSTM network, but any type of detector can be evaluated with NAB. This table gathers detectors based on different statistical methods and not necessarily on deep learning algorithms.

Detector	Standard Profile	Reward Low FP	Reward Low FN
Perfect	100.0	100.0	100.0
Numenta HTM*	70.5-69.7	62.6-61.7	75.2-74.2
CAD OSE†	69.9	67.0	73.2
earthgecko Skyline	58.2	46.2	63.9
KNN CAD†	58.0	43.4	64.8
Relative Entropy	54.6	47.6	58.8
Random Cut Forest ****	51.7	38.4	59.7
Twitter ADVec v1.0.0	47.1	33.6	53.5
Windowed Gaussian	39.6	20.9	47.4
Etsy Skyline	35.7	27.1	44.5
Bayesian Changepoint**	17.7	3.2	32.2
EXPoSE	16.4	3.2	26.9
Random***	11.0	1.2	19.5
Null	0.0	0.0	0.0

Figure 6.2 NAB scoreboard[a]

[a] Extract from [25]

As specified in the usage steps of the previous subsection, we trained our model with the 10 first percent of data from three sets provided by NAB. More precisely, we used the following data sets.

- machine_temperature_system_failure
- ambient_temperature_system_failure
- cpu_utilization_asg_misconfiguration

Then, we defined the detection threshold based on the reconstruction of the training data, and we reconstructed the test data in order to detect anomalies in it and for each of the data sets. As previously stated, a Python script was developed to automate this process and automatically save the required results in NAB format after the test data reconstruction phase. The generated csv result files have the NAB format, which looks like this.

	timestamp	value	anomaly_score	label
2	2014-04-01 00:00:00	20.0	0	0
3	2014-04-01 00:05:00	20.0	0	0
4	2014-04-01 00:10:00	20.0	0	0
5	2014-04-01 00:15:00	20.0	0	0
6	2014-04-01 00:20:00	20.0	0	0
7	2014-04-01 00:25:00	20.0	0	0

Figure 6.3 NAB results file format[a]

[a]Extract from [25]

These files are then to be placed in the directory generated for our detector NAB > results > LSTM28 > realKnownCause. In this way NAB will be able to search for these files and calculate the results according to the mechanism described above. At the end of the generation of the NAB score, the results are listed in the file *final_results.json* under NAB > results. All tested detectors are listed there. This is a way to archive the test results. The results are presented like this.


```
"reward_low_FN_rate": 65.15911811040229,
"reward_low_FP_rate": 47.23375230889405,
"standard": 48.60074613112066
```


Figure 6.4 NAB score from 1 dataset for model 28

These results represent the NAB scores obtained only with the use of one dataset, which is the machine failure temperature set. We already observe that these are good results, if we refer to the table of scores presented previously. Nevertheless, it is useful to use the other data sets mentioned, in order to confirm the stability of the results in other use cases.


```
"reward_low_FN_rate": 64.44751590703164,
"reward_low_FP_rate": 43.98445921038503,
"standard": 47.533342826064704
```


Figure 6.5 NAB score from 3 datasets for model 28

It can be observed that the scores are decreasing very slightly, but this still proves that our model is relatively robust and reliable with use on several types of data. By analyzing the scores, we can deduce that our LSTM 28 model detects few false negatives. Regarding the weighting rewarding low false positive detection, we obtain a satisfactory score. Moreover, in our situation, it is less serious to detect a few false positives than false negatives. It is better to receive false alerts, rather than no alerts when there should be one. In addition, it is important to note that the mahalanobis distance analysis can also help to identify false positives, as these often appear with a slight exceedance of the detection threshold. The weighting will therefore be relatively small and negligible. As far as standard profile is concerned, similarly, this is a correct result. In general, if we base ourselves on the NAB score table, the position of the scores obtained for our LSTM 28 detector would be approximately at the 6th place. If we favor one weighting, rather than another, such as the profile *reward low FN* for example, we would move to the 4th position. Nevertheless, with additional testing, it is probably possible to further increase the score of our detector. In the context of this project, the scores obtained are very satisfactory and meet the needs of the project.

If a model is satisfactory at the Python level, the only thing that remains to be verified and validated during the verification and validation phase is its conversion quality. Thus, it will be possible to affirm that our model will meet the requirements of the project, once it is physically implemented in the FPGA.

6.2 C level

Once the model is validated at the Python level, it is necessary to check and validate the conversion of the model to the C level. If the conversion is correct, the results obtained at the Python level must be close to those we obtain at the C level. That is, the overall error between the two results should be minimal. To ensure that the conversion has been done correctly, two factors must be taken into account. The first is the reliability of the converter and the second is the accuracy chosen for the conversion. Is the latter sufficient to reproduce as closely as possible the results obtained at the Python level.

6.2.1 C simulation

To verify and validate the model at the C level, we will use a test bench also written in C, which is provided by the HLS4ML solution. This test bench allows us to retrieve a vector of input values, which is the same one we used to validate the Python model. These values will be used as a stimulator for the model described in C during the simulation at the C level so that it produces an output vector to be compared with the results obtained at the output of the Python model. At the end of the results of the simulation in C, the test bench will not provide any information concerning the quality of the conversion since it is normal not to obtain identical results to those obtained at the Python level. For this reason, an additional Python script available in appendix N has been developed to analyze, mainly visually, this comparison between the results of the two models. We will see in the next section that the C test bench is also used to verify and validate the conversion at RTL level with a C/RTL cosmulation.

6.2.2 Python script verification

A Python script N has been developed to retrieve the results obtained after the simulation at the C level. This one allows to calculate the global error between the two models and also to plot the results obtained under different angles. On the same graph, we will have the plot of the real values, the plot of the reconstructed values at the Python level and the plot of the reconstructed values at the C level. In this way, we will be able to observe visually and at the same scale if the results obtained at the C level are satisfactory. Moreover, this script allows to plot two error graphs. The first one represents the conversion errors, i.e. between the values reconstructed at the Python level and at the C level. The second error graph represents the reconstruction errors, as we know it at Python level, but this time at C level. It is therefore the errors between the real values and the reconstructed values at the C level.

The different graphs discussed are presented in the results section below with different accuracies tested for the LSTM 28 model conversion.

6.2.3 Results

As detailed previously, HLS4ML converts a Python model into an HLS model based on a precision to be integrated into the configuration file. In our case, it was necessary to experiment with several precisions in order to choose the one that leads to a minimal error. Obviously, the higher the desired precision, the higher the resource consumption. We have chosen a precision that leads to the best results with a minimal consumption.

We will go over the different results obtained with several different accuracies. We do not address the question of resource consumption here, but simply the quality of the results in output from the C model. The details of the resources used by the model with the chosen final precision can be found in the results of the previous chapter in the figure 5.10.

The definition of precision looks like this.

```
<Width, Integer>
```

This is the form that allows conversion to the fixed point format, as detailed above.

The following figures are the result of the use of the script in the appendix N with a different precision configured for each.

For all the following tested precision, values plotted are normalized temperatures. For the figure 6.6, the first graph represents 3 sets of values. The blue plot represents the target values, i.e. the real input values of the model. The orange plot represents the values reconstructed by the Python model. Finally the green plot represents the values reconstructed by the C model. Ideally, the orange and green plots should overlap at most. The second graph represents the conversion error, i.e. the error between the orange and green plots of the first graph. The last graph represents the error of reconstruction by the C model. That is to say the error between the blue and green plots.

- **<16, 6>**

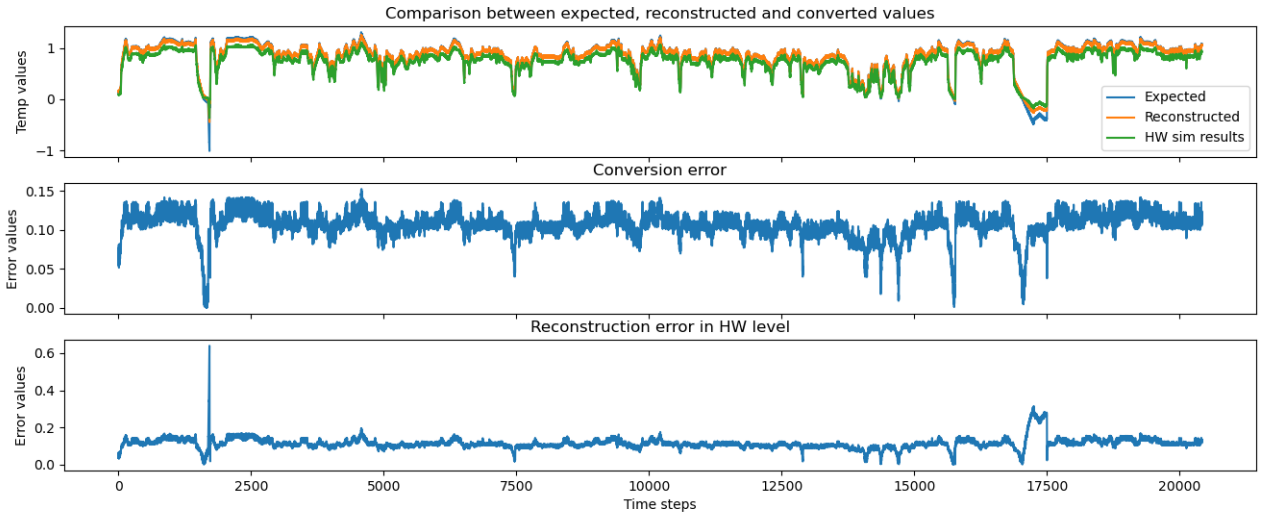


Figure 6.6 Conversion results for precision 16,6

This accuracy is quite satisfactory visually. We observe relatively small errors, but quite fluctuating over time. We can see in the last graph that the reconstruction error resembles what we had seen previously at the Python level. The average conversion error is 0.107.

- **<16, 8>**

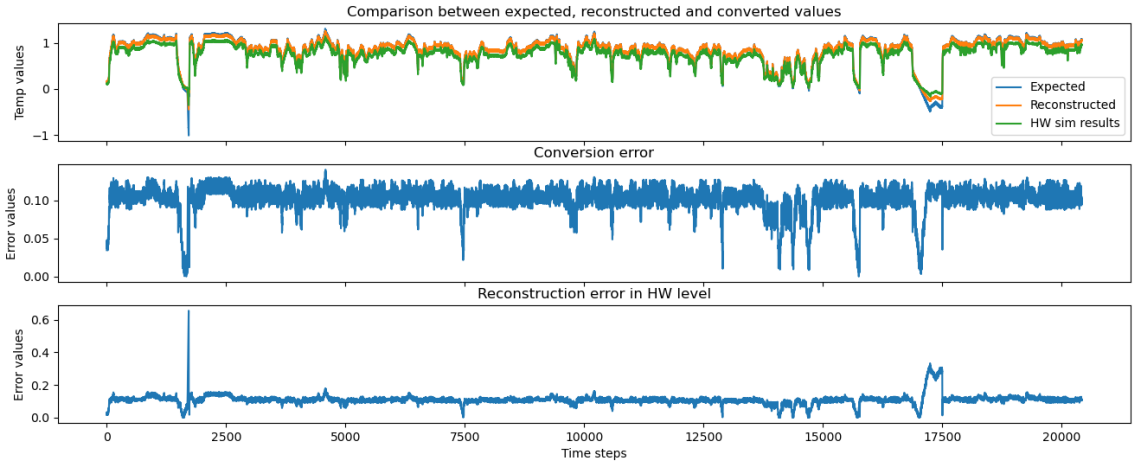


Figure 6.7 Conversion results for precision 16,8

The results of this conversion are very similar to the previous conversion. We just see that the errors of the second graph are more regular and still as low. The average conversion error is 0.102

- **<16, 10>**

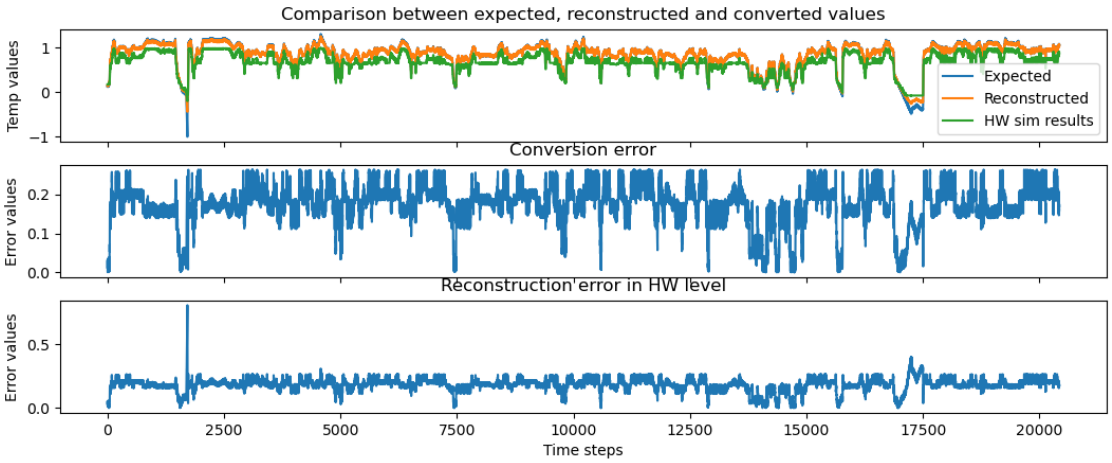


Figure 6.8 Conversion results for precision 16,10

With a further increase in the integer share over 16 bits in width, we see an increase in the conversion error on the second graph. On the last graph, we see that the model has mitigated the last abnormality. The average conversion error is 0.173.

- **<32, 10>**

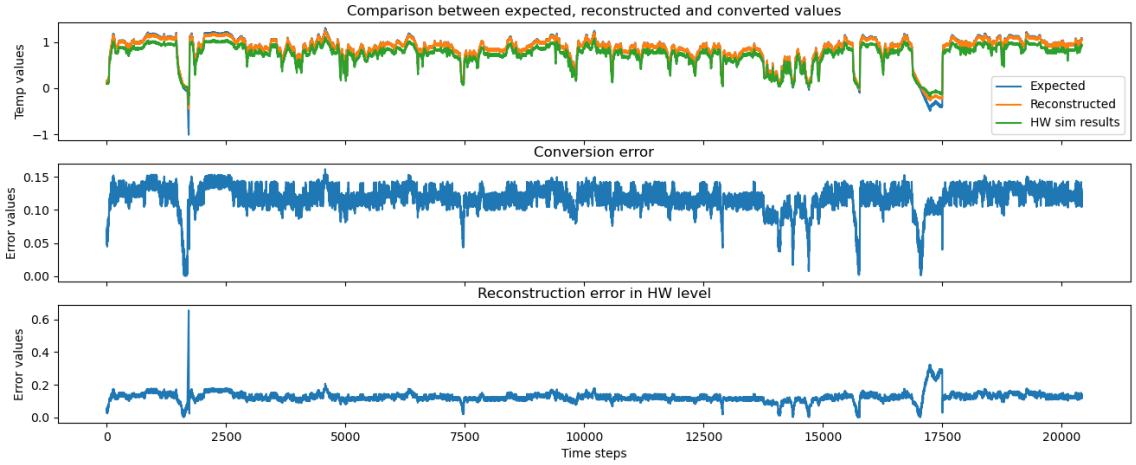


Figure 6.9 Conversion results for precision 32,10

To make up for the previous deterioration, we tried to keep the same full share with an increase of the total width to 32 bits. Logically, the conversion error has decreased again and we find a model that provides correct results, but not better than the first two conversions with a bit width twice as small. The average conversion error is 0.118.

- **<8, 4>**

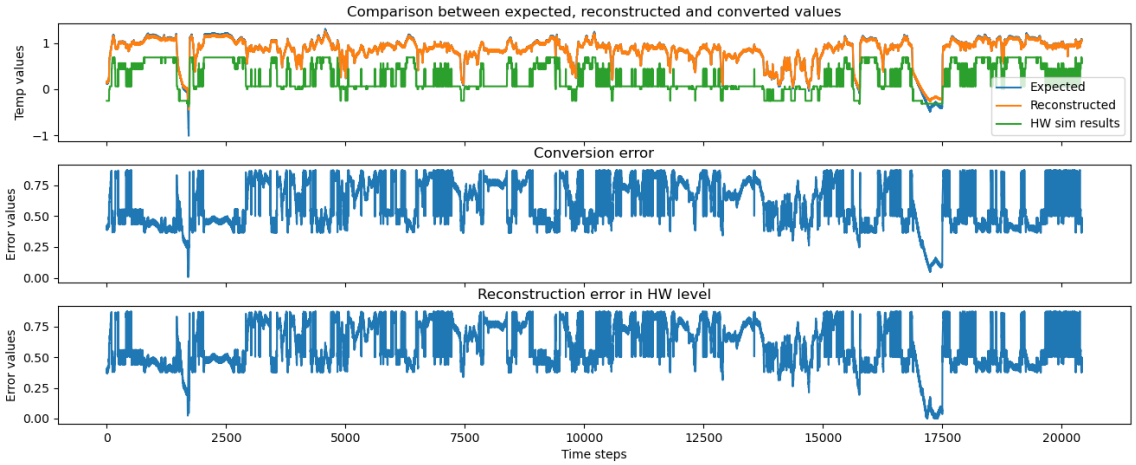


Figure 6.10 Conversion results for precision 8,4

Finally, we wanted to try a precision setting that only favors resource consumption with a width of 8 bits, 4 of which are dedicated to the integer part. We simply observe a model that is no longer able to detect the slightest anomaly. The average conversion error is 0.598.

At the end of these results, we could observe different configurations of the precision, which allowed us to get an idea on the acceptable limits. The best result, which we will keep for the rest of the project, is the second conversion with a precision of <16, 8>. This one has a low conversion error and as we see, it is largely possible to detect the two main anomalies in the dataset. It will probably be more difficult to detect the more subtle anomaly, which is before the last one, but as we have seen, it was already relatively difficult to do so at the Python level.

6.3 Register Transfer Level

Our project requires a second conversion, this time from the C level, in order to obtain a model written in a synthetizable hardware language that can be implemented on an FPGA. It is therefore also necessary to validate whether this conversion has been carried out correctly. Unlike the conversion from Python to C model, which does not provide identical results and where it is necessary to evaluate the level of error, this conversion from C model to RTL model must provide identical results. Indeed, between the model written in C and the RTL model, there is no difference in accuracy. The desired final accuracy is already configured and used by the HLS model. For this reason, the C test bench provided by HLS4ML compares the output results of the HLS model and the RTL model. The test passes only if the two generated result files have identical values.

6.3.1 C/RTL simulation

The advantage of using Vivado HLS to develop a system on FPGA is that there is no need to create a second test bench in SystemVerilog or in Python with the initially planned method which is cocotb. It is possible to reuse the test bench written in C and used previously for the C simulation in order to verify the functioning of the model. For the verification of conversion at RTL level, the test bench is used in the same way. That is to say that it will take the input vector and use it as inputs to the RTL model. However, this time, the test bench will compare the results generated at the C level and those generated at the RTL level in order to confirm or not whether they are identical.

The simulation at C/RTL level also allows to analyze wavesforms generated from Vivado. These are especially useful to understand the functioning of the interfaces that can be used at the level of the IP block that will be exported. The waveforms will be detailed in the next chapter for the explanations concerning the use of the IP block. In the context of the verification and validation of the algorithm's operation, waveforms are not very relevant. If the output results are identical to those of the C model, it is sufficient to validate the operation of the algorithm at the RTL level.

6.3.2 Results

The results obtained for the verification of the HLS model to the RTL model are simple. At the end of the simulation, the terminal will provide information if the test has passed or failed, depending on the comparison of the result file of the RTL model with that of the HLS model. In our case, it was possible to observe the following result.

```
+----------+----------+-----------------------------------------+-----------------------------------------+
|          |          |                 Latency                 |                 Interval                |
+   RTL    +  Status  +-----------------------------------------+-----------------------------------------+
|          |          |      min      |      avg      |      max      |      min      |      avg      |      max      |
+----------+----------+-----------------------------------------+-----------------------------------------+
|      VHDL|        NA|             NA|             NA|             NA|             NA|             NA|             NA|
|   Verilog|      Pass|             22|             22|             22|              1|              1|              1|
+----------+----------+-----------------------------------------+-----------------------------------------+

***** C/RTL SIMULATION COMPLETED IN 0h1m27s *****
***** C/RTL VALIDATION *****
INFO: Test PASSED
```

Figure 6.11 C/RTL simulation results

The HLS and RTL models generate identical results when given identical input values as well. In this case, the simulation used the default hardware reference language which is Verilog, but the same thing could have been done using VHDL as the reference language. It would simply have been necessary to specify this in Vivado HLS.

6.4 Conclusion

In order to carry out this verification and validation phase, it was first necessary to evaluate the detection quality of the LSTM model at the Python level. To do this, a quantitative approach was required and this was achieved with the NAB tool. It was also necessary to use datasets where anomalies are known in order to prove the efficiency of the detector under test. Many data sets are also available with the NAB tool. At the end of this first phase of V&V and on the basis of the scores generated by the NAB mechanism, we were able to validate the model 28 as a reliable and robust model for different datasets.

Once the model was validated at the Python level, it was only necessary to prove that the conversion at the HLS level was done correctly. This was done by providing a dataset to the C testbench made available by HLS4ML so that it could stimulate the C model during the C simulation. At the end of the simulation, it was possible to compare the results obtained at the C level with those obtained at the Python level, thanks to a graphical evaluation of the errors. The precision chosen for the conversion plays an important role in this level of error, this is why several configurations were tested and the precision with 16 bits of width, including 8 for the integer part was favored.

Finally, it was also necessary to validate the conversion at the RTL level. To do so, it was possible to reuse the C test bench to provide the same dataset to the model and to record the results of the model. A second role of the C testbench was to check whether the results obtained at the output of the HLS model are identical to those obtained at the output of the RTL model. Since the accuracy is the same, the results are not expected to vary. The test was passed as part of the DUT that we verified.

At the end of this phase of verification and validation of the functioning of the deep learning algorithm at different levels of abstraction, it is now possible to say that it was a success. This was an extremely critical step in this project. Since an unofficial version of HLS4ML was used to convert a neural network based on LSTM layers described in Python, there was a chance that it had some flaws that could have been critical to the conversion and therefore to the evolution of this project. Finally, it turns out that this was not the case and that the results obtained from the conversions correspond to the initial expectations. It was observed, in particular in the graphs presented in the verification part at C level, that there are no flagrant inconsistencies between the results after conversion and those of the Python model. The slight errors encountered are known and depend mainly on the definition of the precision of the algorithm's calculations.

7 Hardware Implementation

This section gathers the details concerning the hardware implementation part of the project. this is a very important part of the project. We will detail the board used for the physical development of the tool. We will also detail the steps that allowed us to use the deep learning model in the FPGA part of the SoC. The majority of the work involved in this step is the management of the global process, that is to say the acquisition of the data, the pre-processing of these before passing them through the deep learning model, and then the post-processing, which allows us to display the same values as those obtained during the simulations. The steps of the hardware implementation are described in the following overview 7.1.

Contents

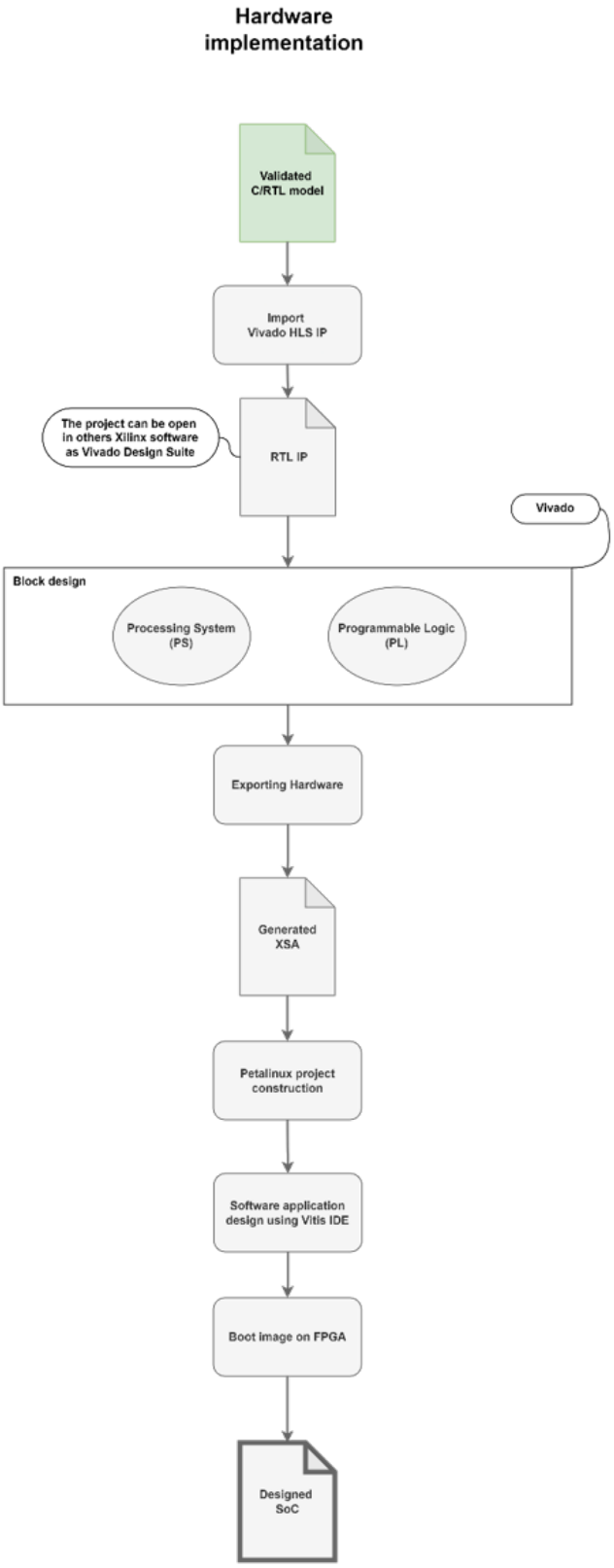


Figure 7.1 Hardware implementation process flow

7.1 Selected Board

To carry out the physical integration phase of the project, we used a relatively expensive board in order not to be limited in terms of resources, available components to handle, etc. The use of this board allowed us to carry out the proof of concept of the project without any real limitation on the hardware level. Nevertheless, it is obvious that a hardware solution like the one presented below is probably not the best option for a commercialization of the tool. For this reason, other alternatives could be tested in order to choose the optimal option, especially in terms of costs. An evaluation of the possible solutions has been made at the end of this report, based on research, but also on the results and feelings gained from this project.

7.1.1 Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit

The equipment used for this project is the Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit. It is composed of various components that allow communication with external equipment, such as displays via the HDMI and display ports, SD cards or a main computer with the USB/JTAG UART port. The board used is based on a System on Chip (SoC). The Zynq UltraScale+ MPSoC PS block has three major processing units. According [53], the processing system is composed by:

- Cortex-A53 application processing unit (APU)-Arm v8 architecture-based 64-bit quad-core multiprocessing CPU.
- Cortex-R5 real-time processing unit (RPU)-Arm v7 architecture-based 32-bit dual real-time processing unit with dedicated tightly coupled memory (TCM).
- Mali-400 graphics processing unit (GPU)-graphics processing unit with pixel and geometry processor and 64 KB L2 cache.

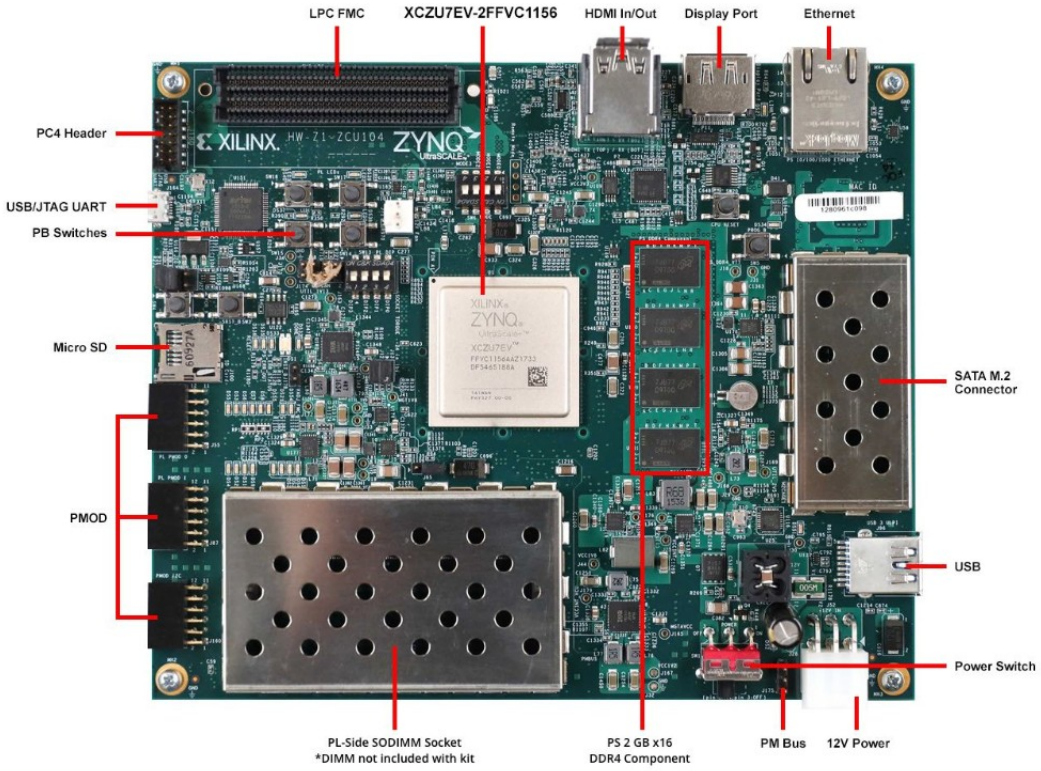


Figure 7.2 Overview of the Zynq UltraScale+ XCZU7EV-2FFVC1156 MPSoC[a]

[a] Extract from [53]

7.2 RTL IP generation

The hardware export from Vivado HLS creates a packaged IP representing the RTL design of the Deep Learning model. We can reuse this IP in the different tools offered by Xilinx to continue the development of our system. It is possible to generate the IP on a base coded in VHDL or Verilog, the behavior is identical in both cases.

7.2.1 Vivado HLS Export

The export of hardware can be done in two ways. One is based on the graphical interface of Vivado HLS and the other on terminal commands.

It is possible to easily export the hardware design without opening Vivado HLS. Once the HLS project is created from HLS4ML, it is possible to build it with the command

```
vivado_hls -f build_prj
```

which is equivalent to

```
vivado_hls -f -build_prj "csim=1 synth=1 cosim=1 export=1"
```

If the export option is set to 1, we ask that the export is done automatically directly after the RTL design is cosimulated.

The second option is the one that has been used regularly and consists in reusing the previous command with the automatic export disabled, i.e.

```
vivado_hls -f -build_prj "csim=1 synth=1 cosim=1 export=0"
```

This way, the HLS project can be built in a complete way and the export can be done by opening Vivado HLS in order to have the possibility to choose which type of export we will use. In our case, we will simply generate an IP catalog format that we can import into the Vivado hardware design block.

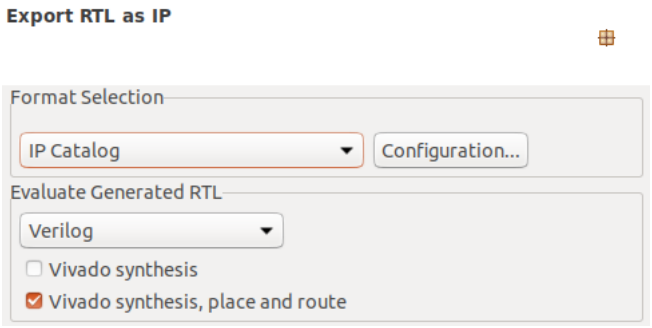


Figure 7.3 Export RTL configuration

7.2.2 Generated Vivado HLS IP Block

The hardware accelerator was generated from the design obtained at the HLS. It represents the deep learning model at hardware level. It is not a standard block proposed by Vivado, so it is relatively relevant to go through the different ports available to drive this block.

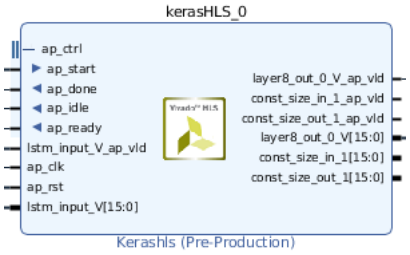


Figure 7.4 Generated Vivado HLS IP

It is important to know that Vivado HLS automatically generates control ports grouped in *ap_ctrl*, which allow to control the process of the generated RTL block with *ap_start*, *ap_ready*, etc. The details of the operation of these control inputs/outputs are available in the Vivado HLS documentation [49].

It is possible to visualize through the waveforms of the RTL simulation, how these ports must be managed in order to execute the acceleration process correctly.

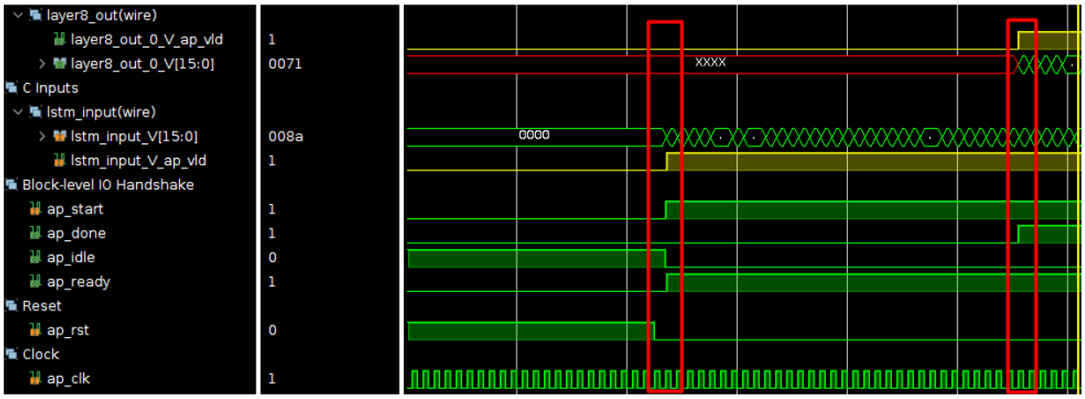


Figure 7.5 RTL simulation waveforms

We observe that to start the process, it is necessary to leave the *ap_idle* mode by passing the *ap_start* and *lstm_input_V* entries to their high level and *ap_rst* to its low level. If *ap_ready* is at its high level, it means that the block is ready to recover a data to be treated through *lstm_input_V*. *ap_done* passes to its high level once the data has been processed and exits the process through *layer8_out_0_V*. At the same time, *layer8_out_0_V_ap_vld* passes to its high level to indicate that the result is available. At this moment, it will be possible to read the result and send a new data to be treated by the acceleration process.

7.3 Hardware design

Once the Vivado HLS IP is exported, it is possible to reuse it in Vivado 2019.2 in order to realize the complete hardware design. It will be possible to import different IPs into the block design in order to connect them together according to the needs of the system. In these imported IPs, we will be able to add the IP from Vivado HLS in the Vivado IP catalog. As we will see, a lot of other Xilinx IP are available by default in the catalog IPs.

First, we will detail the processing system part individually, then the programmable logic part by going through the IPs block which concerns it. Then, the complete block design will be described. We will see that a strategy based on the use of the AXI DMA block was recommended, but in the end, the choice was made to use the AXI GPIO as the main tool to control the IP HLS block in this project.

7.3.1 Processing System (PS)

In this project and based on the board chosen for the realization of our predictive maintenance tool, the hardware part is based on the development of a System on Chip (SoC). In this one, we have two physically separated parts: the processing system part and the programmable logic part. In this subsection, we will discuss the details of the processing system.

The processing system part is composed of different processors that will allow us to perform the main tasks related to the functioning of our tool. As we saw, the PS part specific to our board is composed of three processors. Two CPU and one GPU. The

ARM Cortex A53 processor is an APU (Application Processing Unit), it is this one which was used within the framework of the development of the principal application based on a Linux environment. The other processor is an ARM Cortex R5, which is considered an RPU (Real-time Processing Unit). This processor has been optimized to run in real time. It was not used in this project, because it was necessary to develop a tool with a good adaptation on cheaper boards. Having two processors in the same chip is no longer possible when trying to optimize the development costs of such a tool. The last processor is a Mali 400-MP2, which is a graphics processor. This is useful if you want to display a user interface with graphics that are built in real time for example. The graphic interface is configured by default via the display port. You just need to connect a screen to it to be able to navigate through the different icons from the petalinux environment for example. To control it, it is possible to plug a keyboard and a mouse in the provided USB multi-port. The USB port of the board is also configured by default.

For our use of the processing system part, we decided to embed a Linux OS and to do the development via its environment. The details of this choice as well as the creation method will be described in the following section. The main Linux application will allow us to manipulate the data in the different registers of the main DDR memory and to transform them according to the need. All the processing phases, such as normalization, transformation into fixed or floating point are done via the main application. The FPGA part, as we will see below, is used as a accelerator for the calculations of the deep learning algorithm.

The IP block below from Vivado and specific to our hardware equipment represents the processing system part of the overall hardware design.

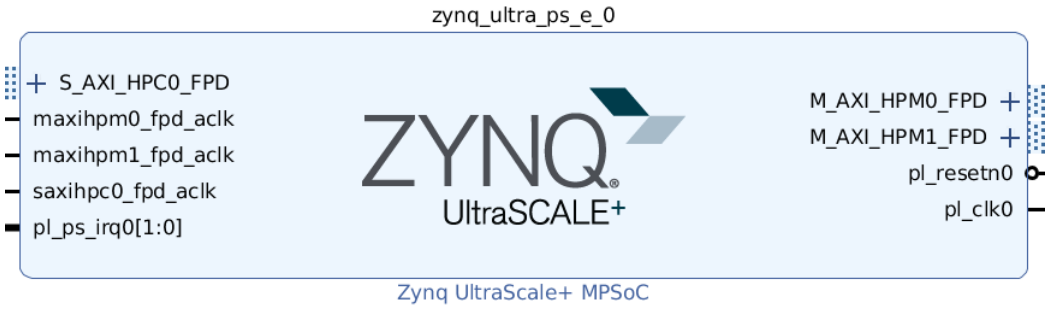


Figure 7.6 Vivado IP of Zynq UltraScale+ MPSoC

Obviously, it is possible to configure this block according to the need. It is possible to add and remove ports at the block level, but also at the physical level. We can for example enable/disable the interfaces from this block, like UART, USB or I2C.

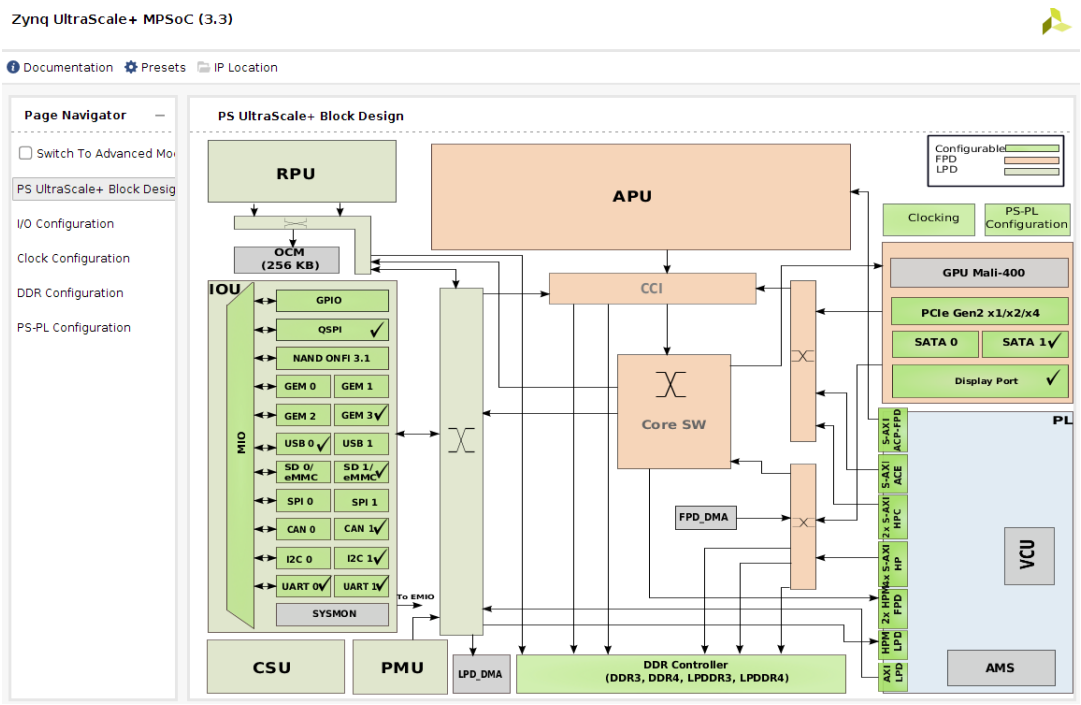


Figure 7.7 *Zynq UltraScale+ MPSoC PS configuration interface*

In this view it is possible to see which drivers are active and usable from the processing system. It is possible to activate and deactivate them easily. This PS block will allow to control the majority of the hardware design through the AXI interface. The processing system is used to implement the main application of the system and a delegation of certain tasks, such as data transfer, is possible with IPs, such as the AXI DMA. The sampling frequency of the hardware design is orchestrated by the PS block with the port *pl_clk*. It is possible to synchronize all blocks from this, or it is also possible to define different clock frequencies.

This PS block has several other ports, which can be removed or added as needed.

7.3.2 Programmable Logic (PL)

The second part of the SoC is the programmable logic or FPGA fabric part. This is the part where we will design and add IPs described in RTL. We will mainly find the AXI blocks and especially the IP HLS block generated from Vivado HLS which represents the hardware acceleration of the deep learning model. The idea of the SoC is to be able to control the PL part from the main application belonging to the PS part. Each RTL block belonging to the FPGA side is linked to a memory address accessible from the PS side and thus potentially accessible from the Linux userspace, if the proper configuration is done.

AXI Interconnect

The AXI Interconnect IP block is the block primarily used to perform transactions from one or more memory mapped master devices to one or more memory mapped slave devices. AXI interconnect blocks are designed to perform memory mapped transfers only.

The master that will mainly drive this AXI Interconnect block for us and in most cases, is the PS block. Through the AXI Interconnect, the PS block will be able to perform transactions with the memory mapped used by the IP blocks on the FPGA side.

AXI GPIO

The AXI GPIO IP block is a block designed to link the AXI interface to a GPIO interface. It can have two channels with a maximum width of 32 bits.

In this project, these AXI GPIO blocks will be in charge of transferring data through the IP HLS block and controlling it. We have used 4 of them with one channel each in order to separate the incoming and outgoing connections. The ports of the HLS block that are not useful for the operation of the main application, but that provide a useful indicator have been physically connected to LEDs.

The use of GPIOs is the simplest way to transfer different types of data. Nevertheless, in a context of optimization with the objective of reducing the CPU load and consequently reducing latency, other possibilities are possible, such as the use of an AXI DMA.

AXI Direct Memory Access (AXI DMA)

DMAs are very important elements in the development of embedded systems. Indeed, they are an element that allows to optimize the global processing of a system by relieving the processor operations. The Direct Memory Accesses will be in charge of performing the transactions through the main memory. The transactions are driven by control ports that are part of the AXI Lite interface. In general, these allow to indicate when transactions can be performed or not, up to which data, etc. In this way, it is a part of the operations that the processor does not need to take care of so that it can concentrate on other essential tasks. The use of a DMA-based strategy optimizes the overall processing time, from acquisition to display of the results.

Nevertheless, we will see shortly that in this project, the chosen strategy is based on the use of AXI GPIOs. There is no optimization of the overall processing time, as the processor can perform all the tasks and display the results in less than ten minutes for a large amount of data, as specified in the project requirements. Obviously, the ideal solution would be to use the AXI DMA to manage the data transfers through the hardware accelerator. The problem is that the generated HLS IP block has default control ports. Ideally, this block should be managed with an AXI Lite interface in order to optimize the system process. For optimization purposes, it is possible to modify the generated block to add this interface at the HLS level. In addition, the AXI DMA is often combined with an AXI Stream FIFO to smooth the data transfer and improve control.

AXI Quad SPI

The AXI Quad SPI IP block provided in the Vivado IP catalog is used as an SPI interface between the FPGA part of the SoC and the SPI device with which it must communicate. This block has an AXI interface to drive it from the PS part. On the other side, it has the standard ports of an SPI device. That is to say the MOSI, MISO, SCK and SS ports. Several configurations are possible with this block, like a dual or quad channel configuration. In our case, it will be used in a standard master mode.

System ILA

The ILA system is a physical Logic Analyzer, which allows to analyze the digital signals connected to the slots of the IP block in question. This allows to analyze in more detail the behavior of different other blocks. Nevertheless, it is a physical block, which will also require resources from the FPGA. This is why it is useful to use it for debugging purposes, but it is also preferable to recreate a final design without this block once it is validated.

7.3.3 IPs Block design AXI DMA based

After research and documentation to find an optimal solution for data transfer through the PS and PL parts, an interesting solution emerged. It is based on a data movement managed by an AXI DMA IP block. The hardware block design is in appendix Q.

The goal is to accelerate the reconstruction of data through the IP HLS block, so it is important not to lose time on other tasks in order not to deteriorate the advantages gained with the use of an FPGA. The AXI DMA IP block is a solution that allows the interface between the main DDR memory and the stream data to pass through other blocks on the FPGA side. In our case, it is a relevant solution, because it would allow to pass through the HLS block automatically with the help of flags managed by an AXI Lite interface. The CPU would not need to take care of this and could concentrate mainly on the data transformation tasks. The AXI DMA is often combined with at least one AXI Data Stream FIFO block, which acts as a buffer and smoothes outgoing and incoming data in memory. For this reason, two AXI Stream Data FIFOs are part of this design. The first one smoothes the transfer from the memory to the input port of the HLS block and the second one from the output port of the HLS block to a new passage in the memory. The AXI DMA has and uses special ports for data output from the memory with the MM2S port and for storage in the memory with the S2MM input port. It is on these two ports that the FIFOs have been connected.

Despite its advantages, this hardware design based on the use of an AXI DMA is not the final design chosen. Indeed, the reason for this choice is based on two aspects. The first is that with a simpler, but less optimal strategy, it is largely possible to meet the requirements of this project, especially in terms of latency. It is therefore not necessary to lighten the CPU load, which is initially relatively low. The second reason is that the HLS block only has a default control interface, which is designed automatically by Vivado HLS. This is not enough to optimally handle AXI DMA or block FIFOs using AXI Lite interfaces. Controlling AXI Lite interface ports individually with the interface ports *ap_ctrl* would mean adding extra AXI GPIOs and would compute the thing a bit unnecessarily. If it is necessary to use a hardware strategy based on the use of an AXI DMA block, the most optimal solution would be to add an AXI Lite interface to the HLS Keras block at the HLS level. This is an optimization step that is not part of this project and is not necessary at this time. It could be part of a second project with different optimization steps.

7.3.4 IPs Block design AXI GPIO based

Finally, the hardware strategy chosen was to use AXI GPIOs to drive the HLS IP block. The hardware block design is in appendix R. With this solution, additional transfer operations are brought to the processor, but this is not a problem, as the processor

handles relatively simple tasks with a fast execution time. The project's requirement of a maximum processing time of 10 minutes gives us some leeway in choosing the fastest solution to implement for our proof of concept. From an optimization perspective, the solution with AXI DMA would be a good alternative.

Here we use one AXI GPIO as input to the HLS IP block and another as output with a data width of 16 bits. We have two additional AXI GPIOs with a data width of 1 bit to control the *ap_start* and *lstm_input_V_ap_vld* ports. Both must be high for the acceleration process to be performed. Then, it was chosen to visualize the indicator ports with LEDs available on the board. The unconnected ports are not useful for this project and have been generated by default.

7.4 Embedded Linux

There are two possibilities for software development. It is either a standard embedded C development or a development based on an embedded Linux environment. Within the framework of this project, it was decided to implement an embedded Linux environment for the processing system part and thus to manage the main application of our tool. Indeed, the Linux OS eco-system offers us a flexible development environment to meet the requirements of our system. The use of drivers at the userspace level is beneficial to correctly structure the operating strategy of the tool. Transactions at the level of memory and external components are done relatively easily. Moreover, the Linux environment allows us to develop an application partly based on higher level languages, like Python for example. This will not be the case in this project, but it is something that can be considered, especially with the use of a model initially designed with Python. Finally, as mentioned before, we can use this tool as if it were a computer with fewer resources. It is possible to connect directly a screen in the display port, as well as a keyboard and a mouse in the USB port. These interfaces are configured by default with the use of the BSP file specific to the ZCU104 to create the Petalinux project. The use of a Linux environment allows to design a user interface in many different ways. There are many advantages to using an embedded Linux environment, both specifically for this project and for future modifications and improvements to the tool. To carry out the development of the embedded Linux environment, the Petalinux toolkit offers a range of useful tools for the development of the Linux project. In this section, the development of this one is presented.

7.4.1 PetaLinux Project

Before developing the Linux project, it is necessary to have designed the tool at the hardware level and to have exported it in order to make available an *.xsa* file describing the hardware design and used for the next steps of the project. In particular the configuration of the Petalinux project.

The use of Petalinux is based on a few commands, which makes the tool relatively easy to use. These commands are the following.

- petalinux-create
- petalinux-config

- petalinux-build
- petalinux-package
- petalinux-boot
- petalinux-util
- petalinux-upgrade

In this subsection we will mainly use the first 4 commands to perform specific tasks. It will therefore be necessary to use flags in addition to the initial command.

Petalinux project creation

To start, there are two ways to create a Linux project. The first way is to use a template based on the choice of the SoC/FPGA. This is mainly applicable when using a custom board where the hardware components and their connections are not clearly identified. Using a template obviously provides little information and most of the configuration is done by the user. To create a Petalinux project on the basis of a template, it is necessary to apply the following command.

```
petalinux-create -t project --template zynqMP -n project_name
```

The second method, which is also the one used for this project, is based on the use of a BSP file targeting a particular development board. This file contains all the default informations that can be used by the board in question. For example, the Ethernet port and the display port are configured by default and ready to be used. This method saves time and is used when working with a known and not customized development board, which has a *.bsp* file available on the official Xilinx website. It is possible to create a petalinux project on the basis of a downloaded BSP file with the following command.

```
petalinux-create -t project -s <path-to-bsp> -n name_project
```

Petalinux configuration

After creating a petalinux project, a directory with the project name was automatically created. This directory contains various other directories to store all the files needed to build the embedded Linux project. The first and most important configuration to be done in the main directory is to identify the hardware design established and exported previously using Vivado. To do this, it is necessary to use the following command.

```
petalinux-config --get-hw-description=<path-to-xsa>
```

This command will initialize and update the XSA file, which describes the hardware configuration to be used in the petalinux project. This command will also open a new window in the terminal, which is the system-level or "generic" configuration window.

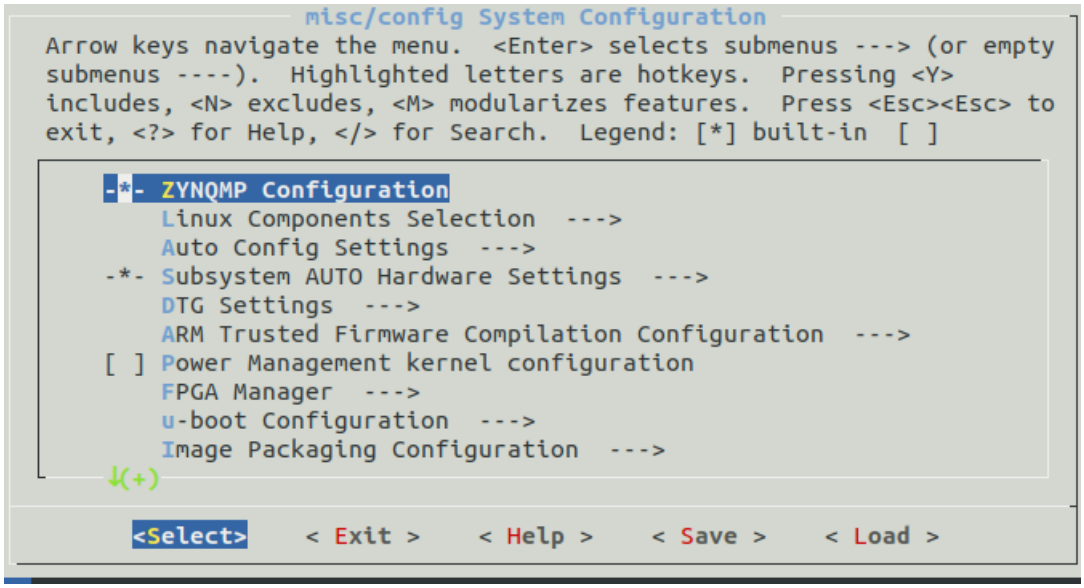


Figure 7.8 Petalinux system configuration

It is from here that it is possible, for example, to specify the meta-data that will be used when creating or modifying the petalinux project. This window can also be opened using only the command *petalinux-config*, without any particular specification. In our case, no system level changes need to be made.

Petalinux kernel configuration

An important configuration part in the creation phase of the Linux project is the configuration of the Linux OS kernel. This configuration is very important, because it allows to activate or deactivate certain parameters to make them accessible at the user space level. It is according to these settings that it will be possible or not to access some specific and useful drivers. The Linux kernel configuration window is accessible from the following command and after a few minutes of initialization.

```
petalinux-config -c kernel
```

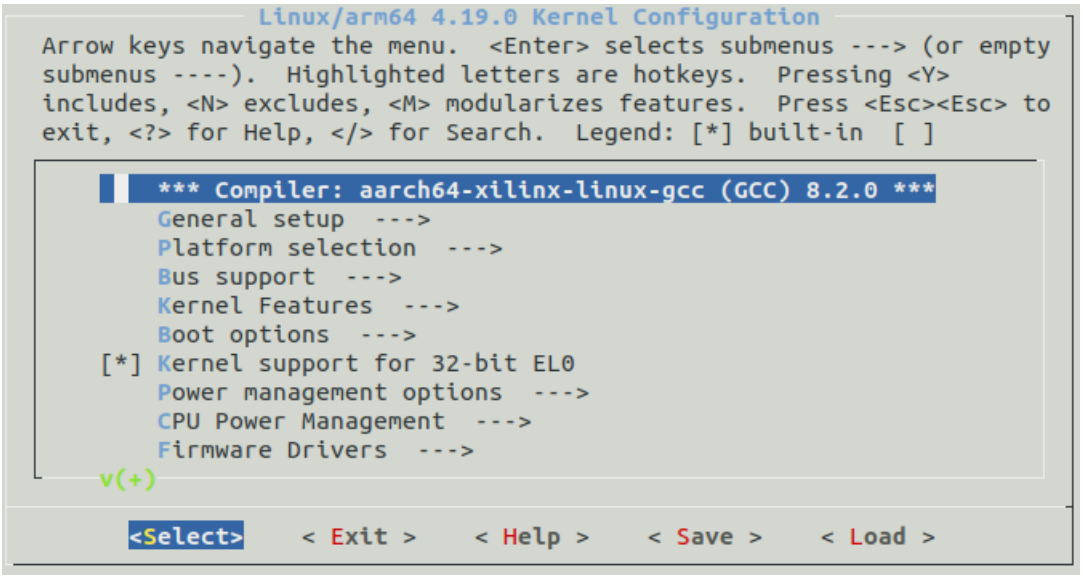


Figure 7.9 Petalinux kernel configuration

In the specific context of this project and based on the experience gained by manipulating different aspects of the Linux user space, it follows that it is important to modify certain parameters at the Linux kernel level. These are the following parameters.

1. **Userspace I/O platform driver with generic IRQ handling**

   This parameter, along with the one that follows, are found in *Device Drivers* → *Userspace I/O drivers* and are used to activate the UIO framework in order to access it from userspace. UIO for *Userspace I/O*, is a driver which will allow to manipulate the memory without having to "hardcode" the memory addresses, as with the use of */dev/mem* driver. In our case, the UIO framework is used to manipulate AXI GPIOs. These are identified in the device tree as compatible with the UIO framework. It will then be possible, from the user space and therefore from the host application, to open the */dev/uioX* driver and to perform transactions from it. Each AXI GPIO has a defined UIO driver to handle individually.

2. **Userspace platform driver with generic irq and dynamic memory**

   This parameter, which is in the same place as the previous one, is also used to make the memory accessible from the UIO framework from user space. Both parameters must be enabled to manipulate the */dev/uio* drivers.

3. **Filter access to /dev/mem**

   This parameter is found in *Kernel hacking* → *Filter access to /dev/mem*. It is necessary to disable it in order to be able to access some parts of the main memory from the Linux user space with the use of the */dev/mem* driver. Without this parameter disabled, it is not possible to access most of the registers from

/dev/mem and thus create a virtual memory for storing the acquired data. This is a security parameter that is enabled by default, but it can be disabled without causing any danger with manipulation at user space level.

4. **User mode SPI device driver support**

   The activation of this parameter, which is found in *Device Drivers* → *SPI support* → *User mode SPI device driver support* must be activated in order to be able to access the *spidev* driver to handle an SPI device. In our case, this SPI device is the accelerometer accessible from the PL part with the AXI Quad SPI block. We will see in the next section that in order for the Linux environment to recognize the SPI device connected to the AXI Quad SPI, it will be necessary to identify it in the device tree. Once this configuration has been activated, it will be possible to access the *spidev* driver from the *root filesystem* in order to perform transactions, such as writing/reading from it.

5. **CPU idle PM support**

   The last parameter to manipulate is in *CPU Power Management* → *CPU Idle* → *CPU idle PM support*. It is simply recommended to disable this one when in the hardware design we use the System ILA block. It is possible that this parameter leads to disturbances on the system ILA, when it is used. Disabling this parameter has no real significant impact, except that the system will consume a little more power at times that do not matter to us. The ILA system allows you to view the signals connected to its slots at hardware level. This is mainly for debugging the system if necessary. This means that once the system is validated, it is possible to recreate a design without the system ILA and therefore without disabling this parameter.

Device tree configuration

An important step to perform before building the petalinux project is to add the complements to the device tree generated from the hardware design. The device tree allows you to describe in *.dtsi* format the different elements of the hardware design with the different addresses allocated, the drivers used, etc. Most of it is generated automatically during the configuration of the petalinux project. It will take the blocks of the hardware design and describe them in *.dtsi* format. In addition, it is possible to add some additional information to activate some specific drivers for example and make them accessible from the embedded Linux root filesystem. To do this, it is not possible to directly modify the automatically generated file. It is necessary to use an additional file which is located in <root-lnx-proj> » project-spec » meta-user » recipes-bsp » device-tree » files and which is named *system-user.dtsi*. It is from this file that it is possible to add additional information to the device tree, as it is the case for this project with the complement below.

```
/include/ "system-conf.dtsi"
/ {


    chosen {
        bootargs = "earlycon clk_ignore_unused   uio_pdrv_genirq.of_id=
    generic-uio";
        stdout-path = "serial0:115200n8";
    };
};

&axi_quad_spi_0 {
  spidev@0 {
    reg = <0>;
    compatible = "spidev";
    spi-max-frequency = <25000000>;
  };
};

&axi_gpio_0 {
    compatible = "generic-uio";
};
&axi_gpio_1 {
    compatible = "generic-uio";
};
&axi_gpio_2 {
    compatible = "generic-uio";
};
&axi_gpio_3 {
    compatible = "generic-uio";
};
```

Firstly, we have added additional information about the AXI Quad SPI in order to be able to drive its slave, which is the accelerometer, from a Linux driver. The addition of this first information will make the driver accessible in */dev/spidev0.1* with 0, representing the slave select (SS) or chip select (CS) number linked to the slave used and specified in *reg* and 1 the bus number. By opening the file descriptor (fd) of */dev/spidev0.1*, in the main application, it will be possible to carry out transactions through it in order to control the slave linked to the AXI Quad SPI Master.

In a second time, the compatibility with the UIO framework has been added to drive the AXI GPIOs and this, in the same way as for the SPI driver. The addition of this information for each AXI GPIO allows to access to a driver each in */dev/uioX* and to make transactions through them.

These drivers allow to perform some transactions with hardware elements in an easier way and adapted to a Linux environment. It is not necessary to write the addresses in raw in the application to drive the hardware blocks. The use of the file descriptors that are linked to each driver allow to write and read in them.

Petalinux build

Once all the configurations are done, the petalinux project is ready to be built. This step is first done with the following command, in the main directory with the project name.

```
petalinux-build
```

This command builds the Linux image to be booted on the used board. This is a step that usually takes time. To do this, the command will generate a complete device tree with the additions made and the elements allowing the image to be booted. It will also create the Linux kernel and it is mainly this step that takes time. Finally, it will generate the root filesystem that will be accessible from the booted Linux environment. It is important to take into account that this step must be repeated each time the Linux configuration is modified. For example, if a new driver is activated with the command *petalinux-config -c kernel*, it will be necessary to rebuild the project so that the changes are taken into account. Obviously, most of the construction will have been done the first time and will not be touched up, so rebuilding after changes takes much less time.

A second use of the *petalinux-build* command can be used with the *–sdk* argument, this time in image » linux. It is the following command.

```
petalinux-build --sdk
```

This one will allow to generate a directory *sdk* in which, we will be able to generate another directory *sysroot* with the following command.

```
petalinux-package --sysroot
```

This directory allows to target, for a type of OS (aarch64 or x86_64), the structure of the root and the elements which must be integrated into it, like libraries, headers files, etc. In Vitis IDE, we will target this path to the root structure of the Linux environment so that it can correctly build the files necessary for the boot of the Linux image.

7.4.2 Boot Linux image using SD card

With a fully built petalinux project, it is now possible to boot the Linux image into the target processor. There are several ways to do this, including via JTAG, QSPI or SD card. For the purpose of this project, we will focus on the SD card method, which is the most reliable and easiest. We will go through two ways of doing this. The first one consists in recovering the good files generated at the end of the construction of the petalinux project. The second is based on the use and compatibility with Vitis IDE. In both cases, we used a 32Gb SD card. An SD card with 16Gb is also sufficient.

From direct access to generated image Linux files

The first method with SD card consists in recovering some specific files generated by the construction of the petalinux project. This method is based on building two partitions in the SD card. Some Linux systems, especially those using a RAM disk, can boot a Linux image only with one partition in FAT format. However, the two-partition method works for all types of Linux systems. It just takes a little extra handling to create the two partitions.

The idea is to create two partitions, one for the boot image and the Linux image and the other for the root filesystem. The best names for these partitions are respectively *boot* and *root*. The first partition *boot* must be under *FAT* format and the second *root* under *EXT4* format. There are several ways to create two partitions customized in terms of configuration (size, format, ...). The easiest way is to use the *lsblk* command to get the name of the driver used for the mounted SD card and then to make the modifications with the following command.

```
sudo fdisk /dev/sdb
```

The idea is to obtain a result, like the one below with a 16Gb SD card.

```
Device     Boot   Start      End  Sectors  Size Id Type
/dev/sdb1  *       2048  2099199  2097152    1G  b W95 FAT32
/dev/sdb2       2099200 31116287 29017088 13.9G 83 Linux
```

Figure 7.10 SD card partitions

The files of the boot image and the Linux image, that is to say respectively *BOOT.BIN* and *image.ub* must be copied in the first partition *boot*. These are located in <root-lnx-proj> > image > linux. The second partition *root* must contain the root filesystem. This one is also available under <root-lnx-proj> > image > linux in compressed format. You just have to unzip it with the admin authorization under the second partition.

Once this step is done, it is possible to plug the SD card in the port of the target board, to place the switches in order to activate the SD card mode and to boot the card. It will be possible to observe the image boot process on a serial terminal. An LED is lit in red, in case the boot is not possible. This happens sometimes and it is enough to simply restart the Zynq board.

From Vitis IDE

Obviously, the previous method does not include the application designed to run in the Linux environment. It is possible to create it in parallel and to add its executable to the first partition to run it after the image boot. Nevertheless, the use of Vitis IDE to build the project after having developed the application in C/C++ is an interesting solution and it is this one which was used mainly during the realization of this project. Vitis IDE is a software tool designed to be adapted with the other Xilinx tools. For this reason, with Vitis, it is possible to target the paths of certain specific files so that it can generate a directory *SD card* containing all the necessary files to import into the SD card. In our case, a partition in *FAT* format was enough to boot the image into the target.

Firstly, after building the petalinux project, it is recommended, but not mandatory, to create a directory that separates certain files to be targeted from Vitis IDE. This allows a better structure of the project with Vitis, but also to have a backup of the configuration copied into the manually created directory. The tree below represents which files and how they are ideally structured in the created *sw_comp*. The files to be copied are all in the directory <root-lnx-proj> > image > linux.

```
sw_comp
└── src
    ├── a53
    │   └── xrt
    │       └── image
    │           ├── boot.src
    │           ├── image.ub
    │           └── rootfs.cpio.gz
    └── boot
        ├── bl31.elf
        ├── linux.bif
        ├── pmufw.elf
        ├── system.bit
        ├── u-boot.elf
        └── zynqmp_fsbl.elf
```

As for the file *linux.bif*, it must be created manually with the following structure.

```
the_ROM_image:
{
  [fsbl_config] a53_x64
  [bootloader] <xilinx-zcu104-2019.2/boot/zynqmp_fsbl.elf>
  [pmufw_image] <xilinx-zcu104-2019.2/boot/pmufw.elf>
  [destination_device=pl] <system.bit>
  [destination_cpu=a53-0, exception_level=el-3, trustzone] <xilinx-zcu104
    -2019.2/boot/bl31.elf>
  [destination_cpu=a53-0, exception_level=el-2] <xilinx-zcu104-2019.2/
    boot/u-boot.elf>
}
```

This BIF file contains the informations to boot the Linux image from the bootable SD card to the target processor. This file and others must be identified when creating the platform project in Vitis IDE. When creating the platform project, simply uncheck *"Generate boot components"* and identify all paths to the requested files/directories. The screenshot below shows an example of configuration from Vitis IDE.

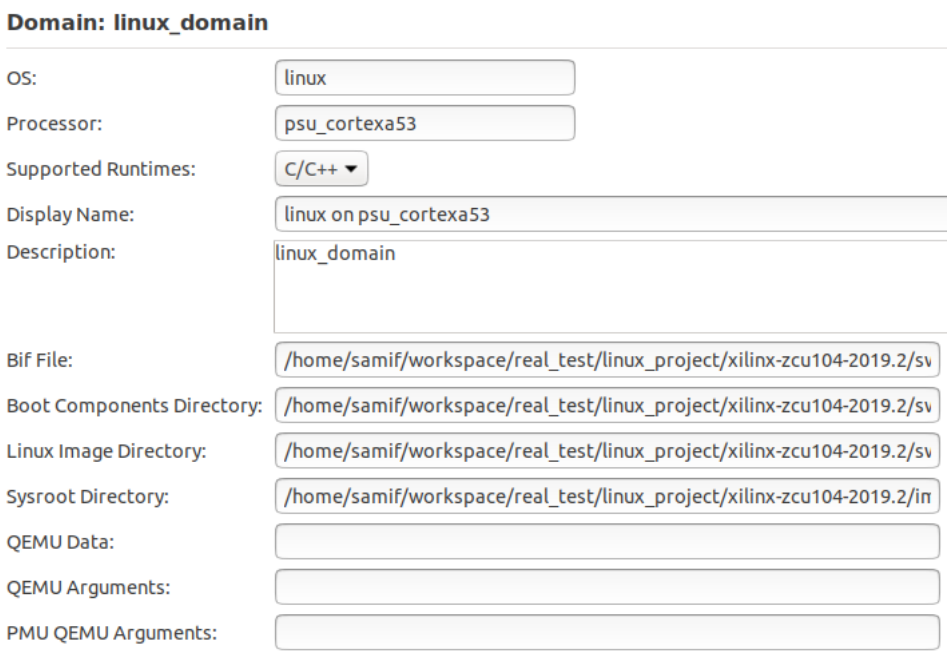


Figure 7.11 Vitis Linux configuration example

After having identified all the necessary paths, it is possible to build the project platform. If this action does not lead to an error in the Vitis console, this means that all the configuration has been correctly carried out and that it is possible to continue the process by creating the Linux application this time.

7.5 Host Application

The main application of the project will allow, in a Linux environment, to carry out all the operations allowing the good functioning of the complete system. It is this application which will allow to carry out the transformations of the data as well as to manage the flow between the PS and PL part while passing by the shared main memory. The application can be developed with different tools, which do not necessarily come from the Xilinx suite. Nevertheless, we will see that the use of Vitis IDE brings important advantages, especially for the debugging part. That's why Vitis IDE has been used in this project for the development of the host application.

7.5.1 Bare metal first test

The first thing to do before realizing the main and final application of our system is to test the critical part of it. That is to say the functioning of the deep learning algorithm at the physical level. The main function of the system as well as the whole meaning of this project is based on it. Until now, we have been able to prove that the deep learning model works at the RTL level thanks to the simulations performed during the verification and validation phase. Obviously, these are signs that the system should work identically at the physical level. However, it is necessary to make sure of this by testing the IP HLS block before carrying out the rest of the development steps of the main application. To do this, it was decided to drive the IP block in question with AXI

GPIOs at the bare metal level. Indeed, this is the fastest way to validate the physical operation of the HLS block, compared to the final solution which will be based on embedded Linux. The figure below shows the strategy used to test the HLS IP block.

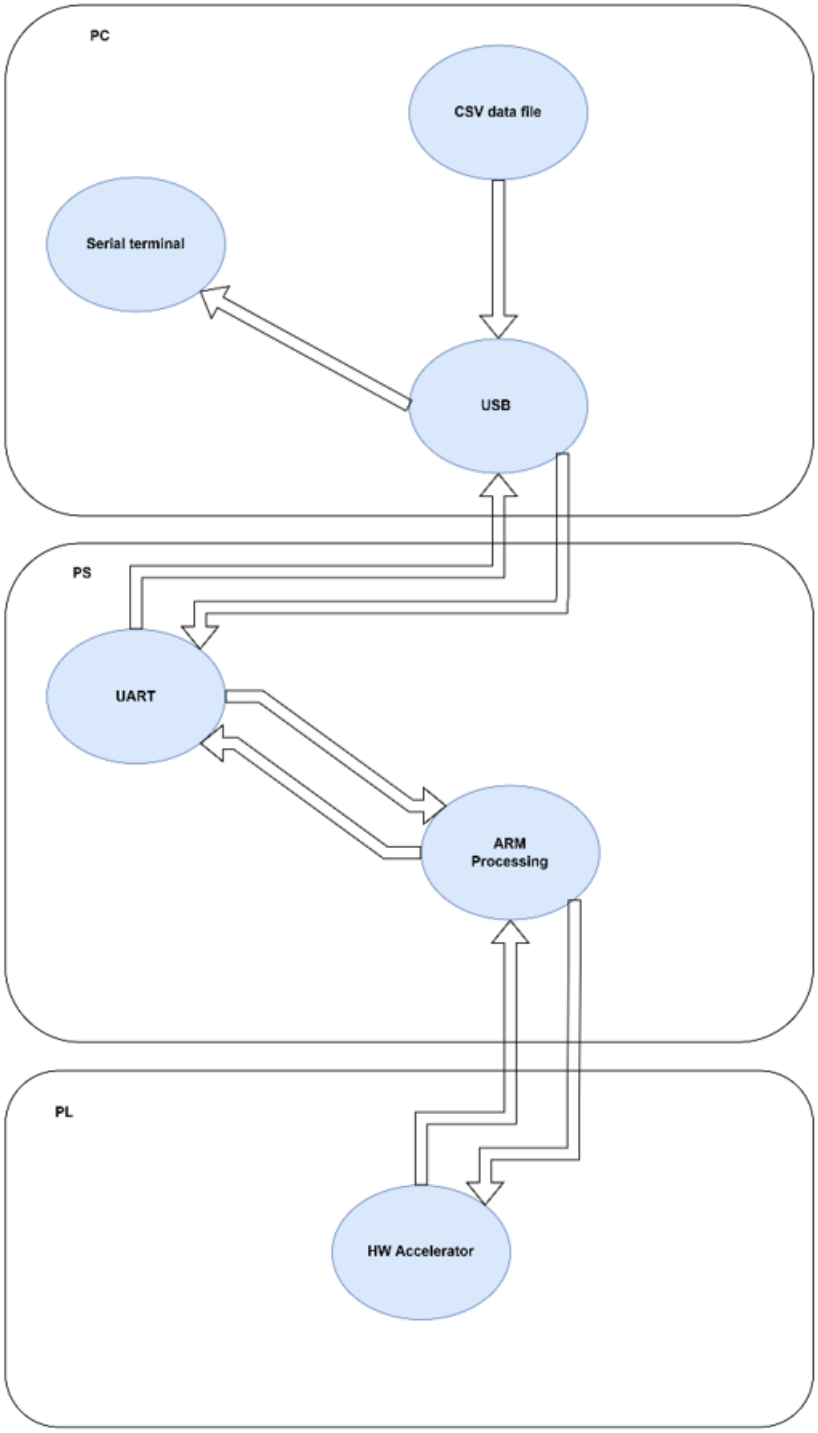


Figure 7.12 IP HLS testing representation

Before running the overall process, it was necessary to ensure that it was possible to provide input values to the HLS IP block and obtain output values similar to the RTL simulation results. We have available a CSV file with values already normalized to the training data, but as a first step it was decided to test only the first five values of the test data. If these first five values provide the same reconstructed results as in RTL simulation, it will then be possible to test all the data in the CSV file from the Linux application this time. Indeed, it will be easier to read a CSV file saved in the SD card from the Linux environment directly than from a bare metal application. Using the Linux environment allows us to do this, as if we were doing it from any computer with Linux OS.

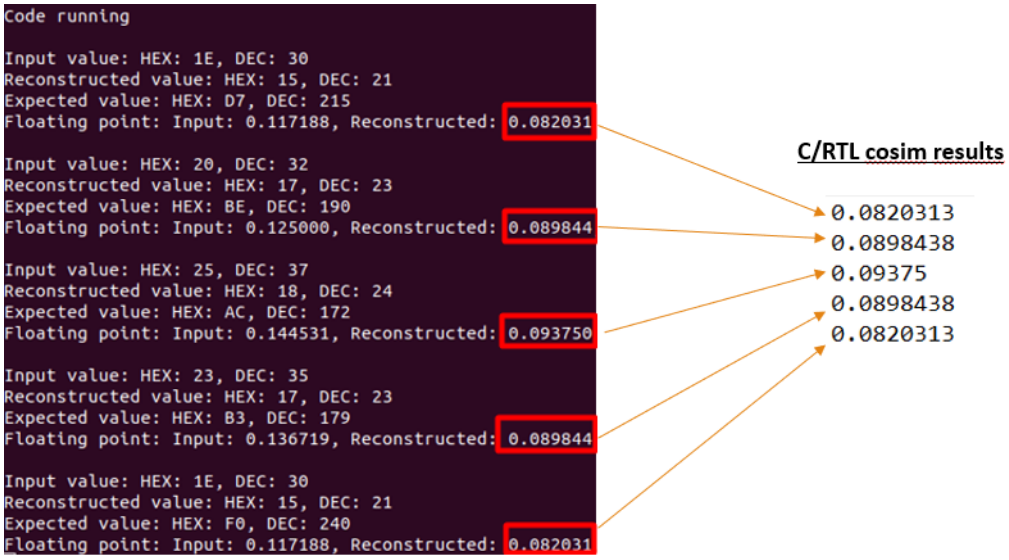


Figure 7.13 Bare metal IP HLS first test results hardware vs simulation

The figure above describes the results obtained at the physical level for the first five test data inputs. The results are displayed via a serial terminal communicating via USB-UART between the PC and the board. On the right, these are the first five results obtained from the C/RTL simulation using the same inputs. It can be seen that the reconstructed values correspond to the simulation values. Now, it is almost possible to affirm that the IP HLS block works correctly also at the physical level. However, to be sure, it is necessary to test all the data in the CSV file and plot them to make sure that the results correctly overlay those obtained in simulation, as presented during the V&V phase.

7.5.2 Host Linux Application

After having manipulated the HLS IP block at the bare metal level with the use of AXI GPIOs in order to demonstrate that it works on some known values and results, it is now time to work on the final application and to develop it in C. The host application must perform the following operations in order to meet the needs of the project.

1. Acquisition of a data window and saving in memory mapped
2. Normalization
3. Floating point transformation to fixed point
4. Hardware acceleration for reconstruction
5. Fixed point transformation to floating point
6. Display results

The application must work with optimally sized acquisition windows in order to process a maximum amount of data in a minimum amount of time. This will allow the new sequence to be restarted as quickly as possible and to be able to visualize the majority of the data acquired by the accelerometer. Initially, the application was developed with all the functions necessary to perform the above operations. Except for the data acquisition step. The program was first designed to process the known data listed in the CSV file that represents the temperature data of a machine. These data are known and so are the results in simulation. It is therefore necessary to validate the operation of the complete program with these data before using data collected in real time.

In order to carry out this step of developing the host application, it was interesting to try to design an application based on the hardware design working with the AXI DMA. This design is described earlier in this section. It was interesting to manipulate the AXI DMA to understand how it works and what advantage it would bring to our project. Finally, as detailed, this design was not chosen to meet the requirements of this project in an optimal way. The host application is therefore based on the use of AXI GPIOs for the manipulation of the HLS IP block and AXI Quad SPI for data acquisition on the PL side.

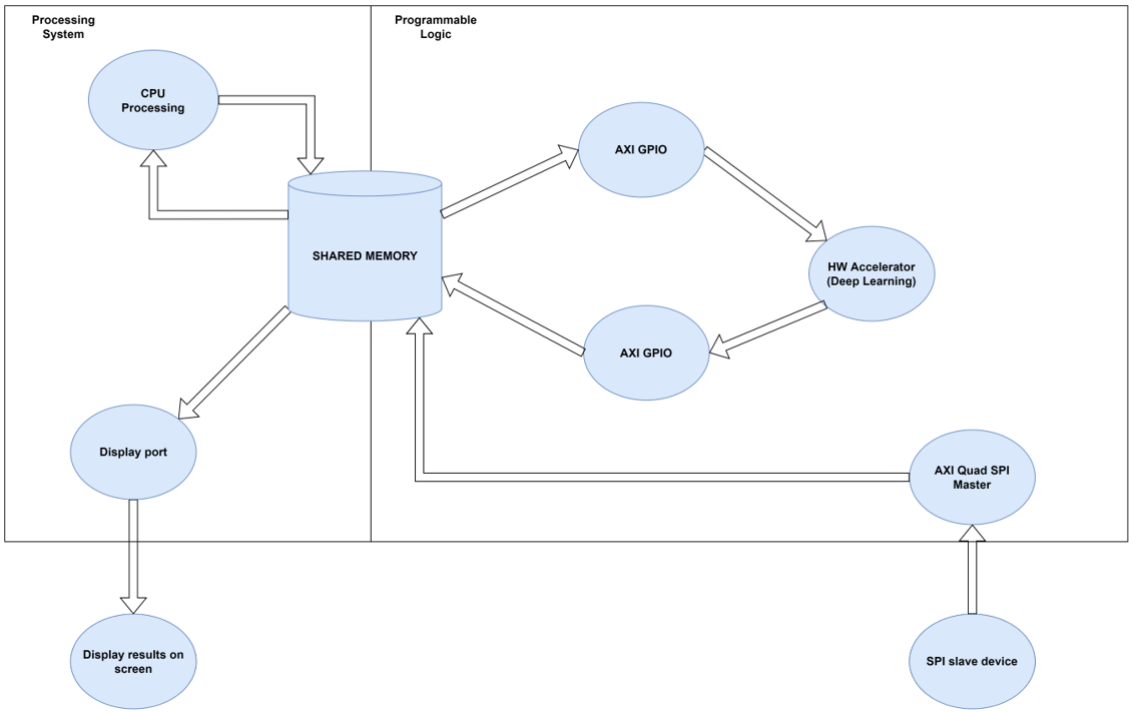


Figure 7.14 Process flow of host application

The communication between the PS side and the PL side is directed by the processor through the main memory shared by both sides. By giving the right information to the right registers, the PL side will be able to meet the needs of the PS side. The generated HLS IP block which represents the deep learning model is driven through AXI GPIOs. The data acquisition is done through an AXI Quad SPI Master, connected to a Slave SPI device, which is the accelerometer. The other operations of the system are done in the processing system.

To go into more detail about the global process of the system that must be driven by the main Linux application, the representation below shows the movement of the data with the transformations they undergo.

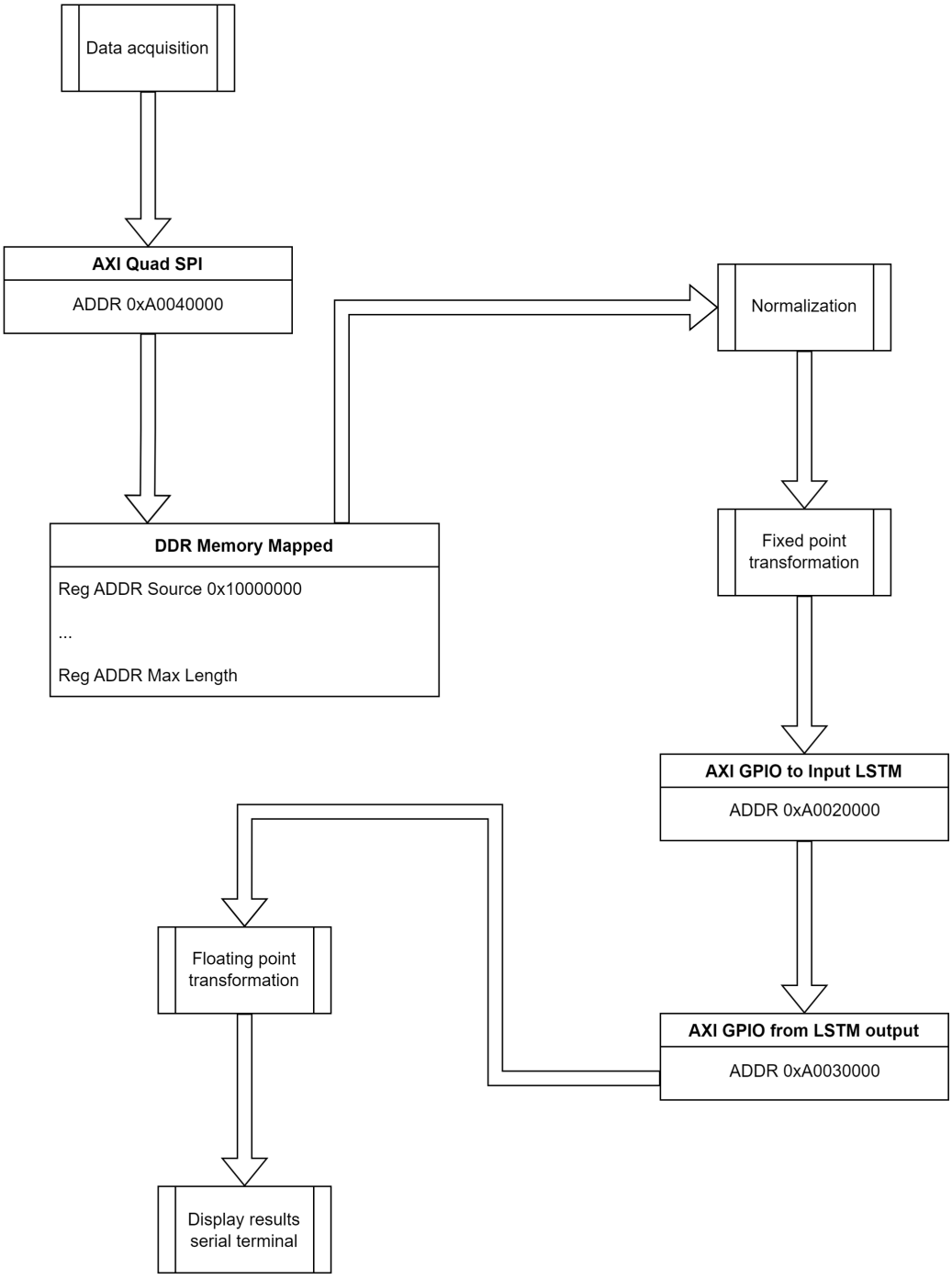


Figure 7.15 Data process flow overview

The first step is the acquisition of the actual data which is done through SPI communication from the AXI Quad SPI master which serves as the SPI interface on the PL side. The main application manages this acquisition through the registers allocated to the AXI Quad SPI. This is controlled through the Linux driver *spidev*, but we will see this in more detail later in the report. This stage ends with a recording of the acquired data in the main memory via a mapped virtual memory.

The second step consists in retrieving the recorded data to normalize them on the basis of the training data. A Python script has been developed to record the training information useful for the calculation of the normalization function. This includes the maximum and minimum values to perform the following calculation.

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

With X, the value to be normalized, X_{min} and X_{max}, which are respectively the minimum and maximum value of the training data and X_{scaled} which is the result. That is to say the normalized value to be used for the rest of the process.

Then, there follows a transformation operation in *Fixed Point* format, which is a very common format in the embedded development field. To realize this one, we are going to use the precision value that we defined initially during the conversion of the model at the RTL level. That is to say *<16, 8>*. We have a precision of 8 fractional bits. So we will multiply the value in floating point format by 2^8 and round the result. This way we can get an accurate integer approximation of the initial value.

$$fixed = round(floating\_input \times 2^{fractional\_bits})$$

With values in fixed point format, it is now possible to pass through the FPGA and more precisely the IP HLS block representing the deep learning model. To do this, we simply send each value to the address of the AXI GPIO in question. We do the same thing to read the output of the IP HLS block. Once the data has passed through the LSTM model, it is possible to transform the results into floating point format. To do so, we just need to perform the reverse operation as the one detailed above. Simply divide the result by 2^8.

$$floating = \frac{fixed\_input}{2^{fractional\_bits}}$$

At the end of this operation, the results are ready to be recorded and displayed in real time. We note that the processor does not really have any large tasks to perform, which could lead to saturation. Assigning it to the tasks of transferring data through the AXI GPIOs, in addition to the transformations, is not problematic in the context of this project.

7.5.3 Debugging

An important and essential step in the development of the main application is the debugging of it. Indeed, the development of a more or less complex program generally requires a debugging time often more important than the rest. That's why it is important to adopt the most suitable methods for the needs. In our case, all the development has been done with Vitis IDE and we will see that it is possible to debug a bare metal application relatively easily, like any other embedded C program. An additional step is added here with the compilation of the FPGA part. The use of Vitis IDE becomes very interesting for the debugging of the Linux application. We will see that this can be done efficiently with an ethernet communication, without having to manipulate the SD card at each program change.

Bare metal debugging

Vitis IDE is designed and adapted to be easy to use with Xilinx elements. That's why we used Vitis IDE for the design of the host application and thus for the debugging part of it. At the bare metal level, it is enough to create an application based on a standalone platform. When creating this application, all the header files and other useful files are made available in the working directory. The most important file to include in the program is the file *xparameters.h*. This file describes the system elements with their registers in order to be able to manipulate them in the application. It is not necessary to go further with the explanations concerning the design at bare metal level, because it is not this type of implementation that we are going to use. Nevertheless, it is useful to have some details about the compilation at bare metal level in order to have an idea of how it works when we work with a SoC.

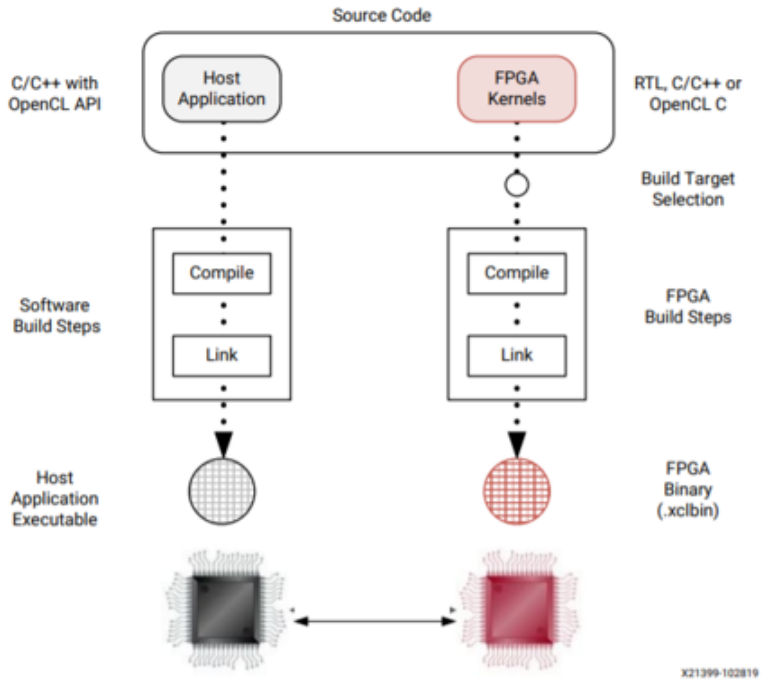


Figure 7.16 Compiling flow host application and FPGA kernels[a]

[a] Extract from [48]

From Vitis IDE, the compilation of the Vitis project is done in two steps. In a first step, the application written in C and intended to run in the ARM processor will be compiled with a standard GCC cross-compiler (g++) which is available from the Vitis installation. When running the compiler, an object file will be generated based on the C program before generating an executable file.

In a second step, the compilation of FPGA kernels will be done with the use of a special compiler, which is Vitis Compiler (v++). This one will first allow to transform hardware designs written in RTL, C/C++ or OpenCL into xilinx object files (.xo). Then, the *.xo* files will be generated and linked into a FPGA binary file (.xclbin) with the use of the target platform.

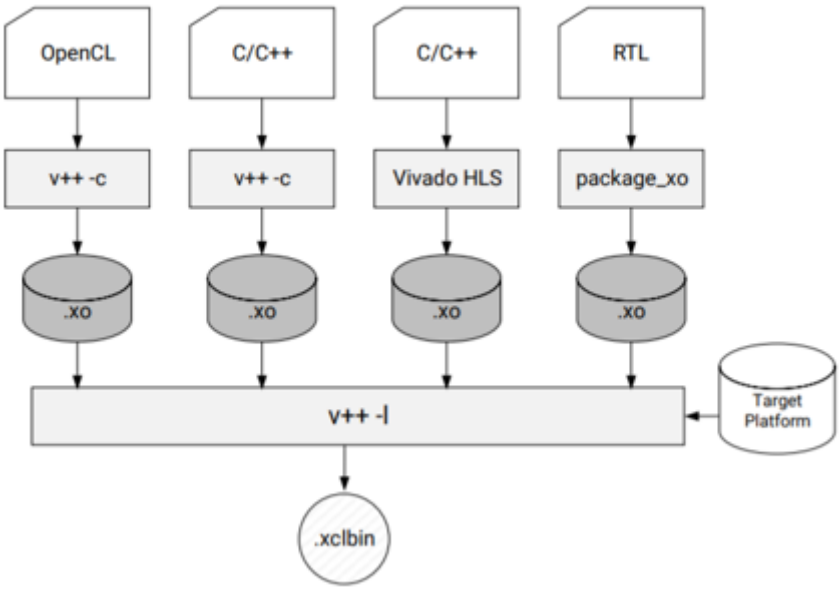


Figure 7.17 FPGA kernels building[a]

[a] Extract from [48]

Linux debugging

At the bare metal level, the boot of the system is done in JTAG mode. Therefore, we have a direct connection between the PC and the board. At the Linux level, to be able to debug the application directly from Vitis IDE, it is necessary to have a communication channel between the PC and the target. This is where the Linux TCF (Target Communication Framework) Agent comes in. It is a channel that has been specially designed to communicate directly between a PC and an embedded target. It is a framework that is available with Eclipse-based IDEs, like Vitis. With this, it is possible to download the application developed on Vitis IDE directly into the target in order to launch it or to debug it.

To achieve this, some conditions are necessary. Firstly, you must know that the Linux TCF Agent is activated and accessible by default when creating a petalinux project. If this has not been disabled voluntarily, there is no reason to make any changes. For this

to work correctly, it is necessary to boot the Linux image on the target with the correct hardware configuration. It is also necessary to have an ethernet connection established between the router used by the PC and the target. This is required because the Linux TCF Agent must connect through the IP address of the Linux image. Once connected to the petalinux interface, you just have to get the IP of the target with the *ifconfig* command. This one must then be copied into the Linux TCF Agent configuration in Vitis IDE.

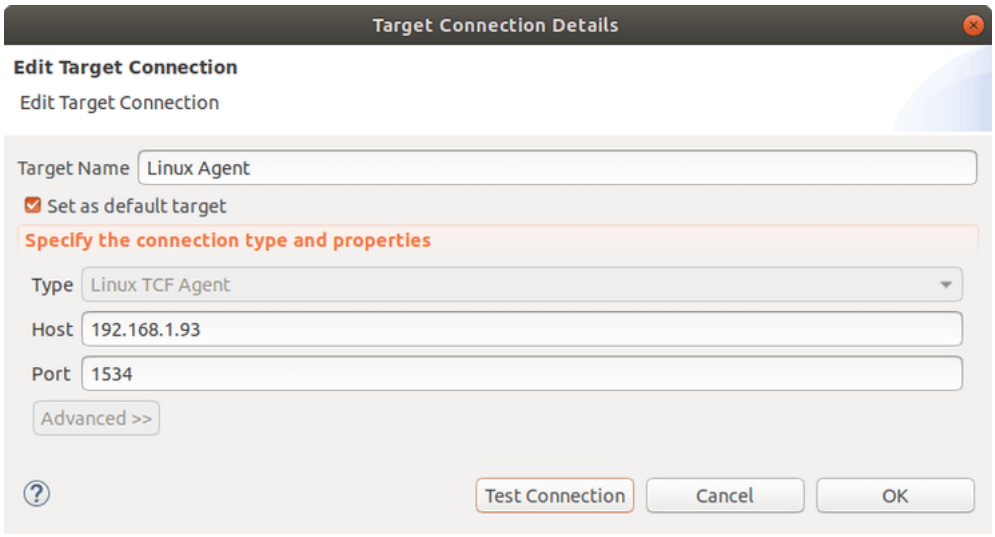


Figure 7.18 Linux TCF Agent configuration[a]

[a] Extract from [19]

Then it is possible to test whether the connection between the PC and the target is well established before working through this channel.

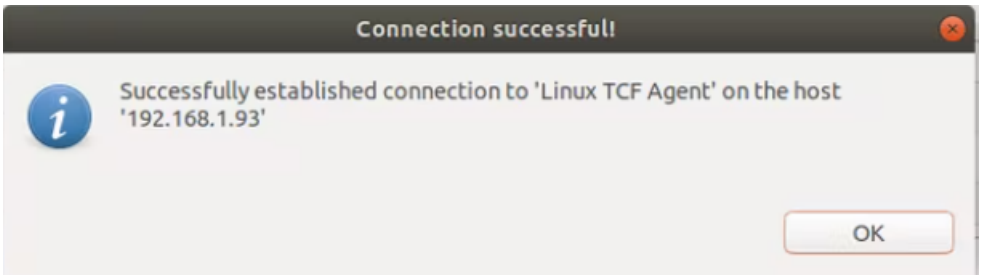


Figure 7.19 Successfully connected to the target[a]

[a] Extract from [19]

Once the connection is established, it is possible to download the Linux application directly into the hardware target. This allows the application to be debugged efficiently and quickly with the debugger configured in the Vitis IDE. This method makes this IDE the ideal tool to develop and test efficiently a Linux application in an FPGA SoC.

7.6 Data Acquisition

At first, the Linux application was developed on the basis of a pseudo-real time acquisition. That is to say that the data used were those recorded in a CSV file. It was possible to save each line of data in the memory and to use it as for a real time acquisition with a sensor. The operation on the basis of already acquired data allowed to validate the general operation of the application. At the end of this, the last challenge was to succeed in communicating with the sensor in order to finalize the operation of the system. Indeed, this project must contain a real-time data acquisition system in order to finalize the proof of concept.

It was thus possible at first to record all the data of the CSV file in the main memory thanks to the driver */dev/mem* which allows the access of the physical memory of the Linux kernel from the user space. These data have been stored in a range of the mapped memory. This range is initially at most 16'384 entries. This value comes from the fact that the maximum length of the mapped memory is 65'536, which corresponds to the parameter of the Linux configuration *max_map_count* and which is found in proc > sys > vm > max_map_count. We will use this same method to record a window of 16'384 data from an MPU-6000 accelerometer. This section describes the protocol used and how it is used in the Xilinx environment.

7.6.1 SPI protocol

Within the framework of this project, for the method of acquisition of the vibratory data, it was decided to use the protocol of communication SPI (Serial Peripheral Interface). It is a protocol developed in the mid-1980s by Motorola. The reason for this choice is initially based on the fact that the company which proposed this project uses relatively fast sensors. An SPI communication allows to work with high frequencies. Moreover, it is an interesting protocol to take in hand in the context of learning and acquisition of experience. However, it is a protocol which is a little more cumbersome because of its 4 connections, contrary to I2C for example which uses two. Moreover, the communication distance is limited to about 10 meters to keep a certain reliability.

The operation of the SPI communication protocol is based on the principle of communication in a single master and one or more slaves. To do this, it is necessary to transmit and receive information through 4 connections.

Figure 7.20 SPI connections overview[a]

[a] Extract from [40]

The first connection *MOSI*, which means *Master Output Slave Input* is used so that the master can transmit information to the slave. This information must conform to the format recognized by the slave so that it understands the instructions received. The second connection *MISO*, which means *Master Input Slave Ouput* allows the slave to transmit information to the master. This is the information expected according to the instruction sent by the master through the MOSI. The third connection *SCLK*, which means *Serial Clock*, allows the master to control the transmission frequency through the slave. The last connection *SS/CS*, which means *Slave Select / Chip Select*, allows the master to drive different slaves. A number *N* of slaves to be driven by the master leads to a number *N* of ports *SS/CS* at the master. Each slave is connected to a chip select so that the master can decide who to deal with and when.

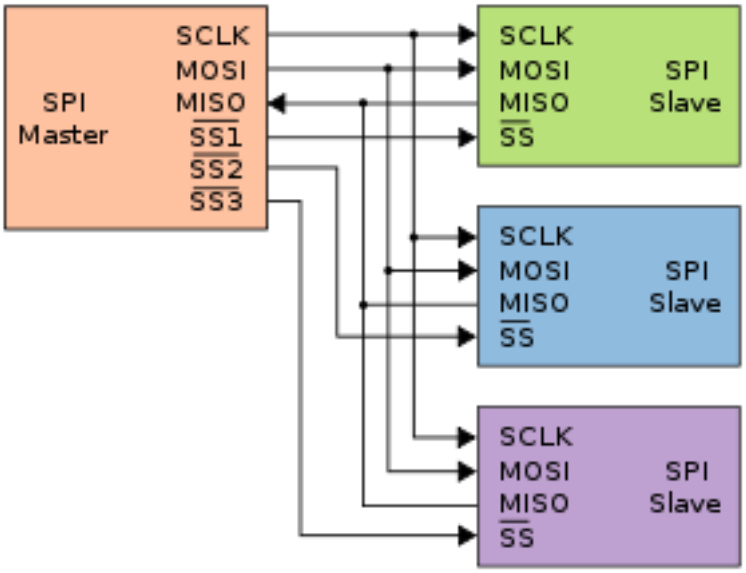


Figure 7.21 SPI multiple slaves overview[a]

[a] Extract from [40]

The SPI communication protocol is a type of serial communication. That is to say that each bit of information is sent in series on each rising or falling edge of clock according to the parameter setting of the clock polarity (*CPOL*). The operation of a SPI communication is based on the representation below.

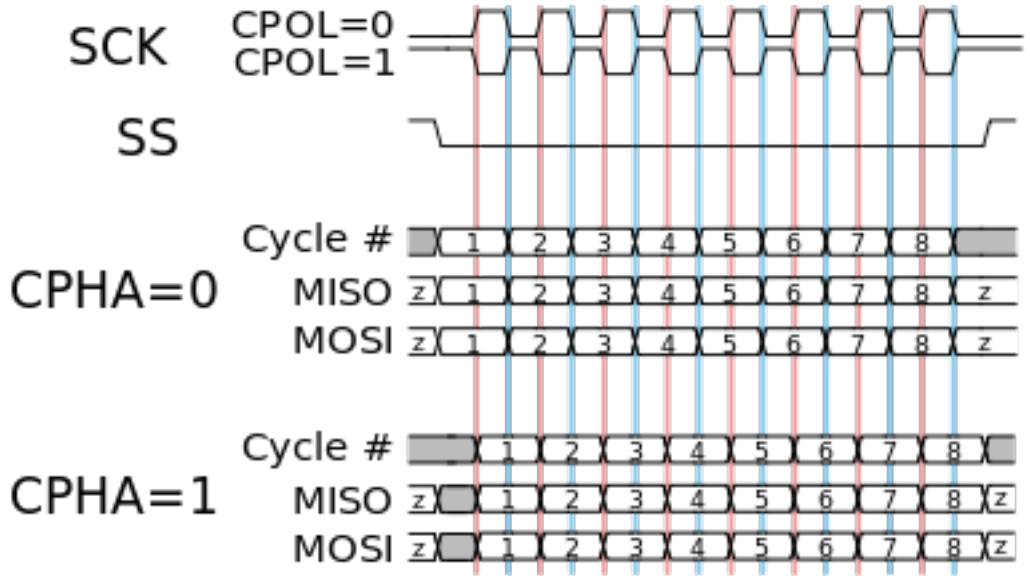


Figure 7.22 SPI timing representation[a]

[a] Extract from [40]

The configuration of the clock phase (*CPHA*) simply determines whether the transaction should start at the first or second clock stroke. The edge of this one is defined according to the chosen polarity. It can be seen that the chip select is initially in the high position and that it must be in the low position to be able to carry out the transactions. If several slaves are connected, only one chip select at a time must go to the low position.

7.6.2 AXI Quad SPI

To carry out the communication with the SPI device, it was decided to carry out this one on the FPGA side of the SoC. That is to say that we used the IP *AXI Quad SPI* proposed in the Xilinx IPs catalog. Again, this choice comes initially from the fact that it is possible to manipulate more easily the data acquisition speed. This makes the system more adaptable to a change of sensor.

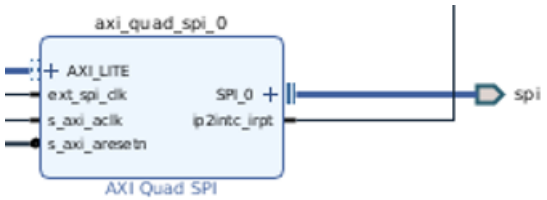


Figure 7.23 AXI Quad SPI IP

The AXI Quad SPI IP block above is the one used in the final hardware design of this project and acts as the SPI interface for the slave SPI device, which in our case is a sensor. It has a port *ext_spi_clk* that directs the frequency of this SPI interface. This frequency should not be confused with the frequency used by the AXI block itself, for its operation. The latter operates in synchronization with all the other AXI blocks in the system at 100 MHz. It is recommended that the SPI interface has a lower frequency than the AXI frequency. Next, we connected the SPI port (*SPI_0*) to an external connector on the SoC, which is the PMOD. The configuration of this can be found in the XDC constraint file, which describes all the connections between the hardware design and the elements that are on the board. We have used 4 connections of the PMOD0 (0, 1, 2, 3) to connect the interfaces MOSI, MISO, SCLK, SS respectively. Thanks to this, it will be possible to connect the 4 connections directly to the sensor using the PMOD0 connector. This block is driven by an AXI interface, as for the other blocks. We have also connected the AXI Quad SPI interrupt to the processing system so that it recognizes the IP block. Each time the AXI Quad SPI is manipulated, an additional interrupt is added to the list of system interrupts in the */proc/interrupts* of the root filesystem.

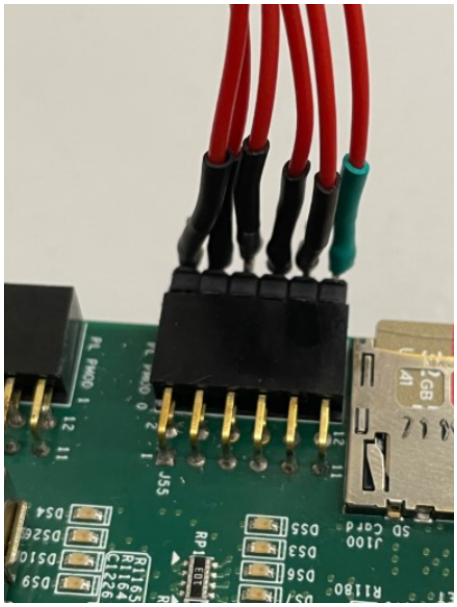


Figure 7.24 SPI connection through PMOD0

As far as the configuration of this block is concerned, it can be done in different ways, but here is the one used in this project.

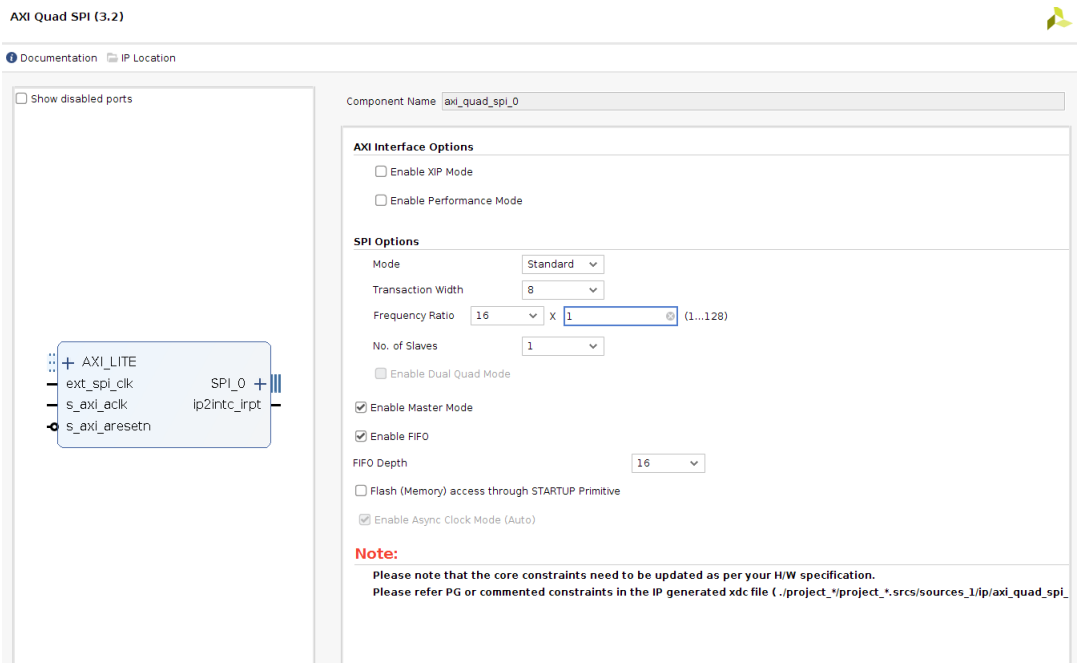


Figure 7.25 AXI Quad SPI configuration

The mode used is the standard one with a possible transaction of 8 bits width. We will see below that despite the fact that the sensor requires information on a minimum of two bytes, we can simply send it two 8-bit values in a row so that it understands the information transmitted. Finally, we are in master mode only, i.e. this AXI Quad SPI block is the only master to carry out the transmission of information and the acquisition of data at the level of the slave SPI device.

7.6.3 Sensor MPU-6000

The sensor used to acquire the acceleration data is the MPU-6000. This one will be used as accelerometer in our case, but it can also be used as gyroscope or temperature sensor. It is a sensor that supports two communication protocols, which are SPI and I2C. It is interesting to know that the SPI is only available with the MPU-6000 and that the other types of MPU-60X0 do not support this protocol.

The MPU-6000 is integrated into an IMU Click MPU board.

Figure 7.26 MPU IMU Click board

We have connected this board to the ZCU104 through the PMOD0 connector in order to perform the data acquisition by SPI communication. A first thing to do before using this board is to make sure that the resistors below are connected to the SPI mode for a use of the concerned ports.

Figure 7.27 Resistance SPI / I2C mode

We notice that this is not the case on the picture above. The I2C ports are connected to the rest of the system. It was therefore necessary in our case to change the location of these 3 resistors to put them under the SPI column.

Then, we connected the ports *SDI, SDO, SCK and CS*, respectively on the ports *PMOD0_0 (MOSI), PMOD0_1 (MISO), PMOD0_2 (SCK) and PMOD0_3 (SS)*. In this way, we could have an adequate connection to an SPI communication between the master (AXI Quad SPI) and the slave device (MPU-6000). The system is supplied with 3.3 VDC. It is necessary to connect the GND and the 3.3VDC of the MPU IMU click to the relevant ports on the PMOD0 connector.

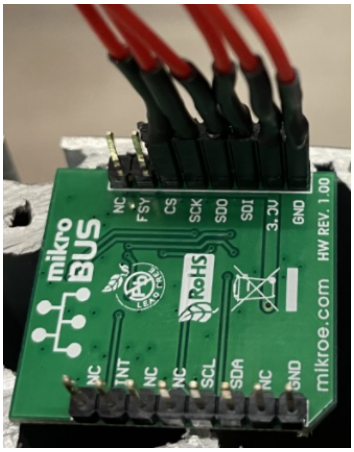


Figure 7.28 SPI connection through MPU IMU Click ports

The burst will adapt to the clock transmitted by the master and to the information transmitted from the MOSI. However, it is also necessary that the information sent from the master be understandable by the slave. In our case, to access the slave's registers and to transmit data to it, it is necessary to send information on at least two bytes. The first byte is dedicated to the register you want to target in the slave device with the first bit of it indicating whether you want to read or write to this register. The second byte is used to indicate the data you want to write through the register indicated to the slave device.

SPI Address format

MSB							**LSB**
R/W	A6	A5	A4	A3	A2	A1	A0

SPI Data format

MSB							**LSB**
D7	D6	D5	D4	D3	D2	D1	D0

Figure 7.29 SPI information structure[a]

[a] Extract from [14]

It is important to carry out transactions with the slave in this format so that it understands the orders given by the master.

7.6.4 Acquisition strategy

Concerning the acquisition strategy itself. We have seen that in our case, the master is the AXI Quad SPI or indirectly the ARM processor that drives it from the SoC. We have a single slave which is the MPU-6000 sensor that we use as an accelerometer. That is to say that we will retrieve the data we are interested in only in the registers concerned by the accelerometer. Moreover, we will only take into account the values acquired on the X axis. Obviously, if we are interested in other axes, it is relatively easy to access the related registers.

To drive this AXI Quad SPI, it is possible to work directly with the registers allocated to it in main memory by using the */dev/mem* driver. To do this, it is advisable to read the Xilinx documentation on the AXI Quad SPI carefully and to follow the same steps as the examples given there. In the case of this project, it was decided to use the linux driver *spidev*, for reasons of user comfort. As for the AXI GPIOs that we have made compatible with the UIO framework, it is no longer necessary to work in raw with the AXI Quad SPI registers. It is possible to make the driver *spidev* compatible with the AXI Quad SPI, when building the device tree, on the user side. To do this, simply add the following lines to the device tree.

```
    &axi_quad_spi_0 {
      spidev@0 {
        reg = <0>;
        compatible = "spidev";
        spi-max-frequency = <25000000>;
      };
    };
```

With this modification of the device tree, we have added to the basic configuration of the AXI Quad SPI (*axi_quad_spi_0*) an additional piece of information which is the *spidev@0* configuration. We have just indicated that this one is compatible with the driver *spidev*. *spidev@0* contains additional information, in particular the *reg* number that we are going to provide it. This information is important when you want to create several drivers of this type to manipulate. It is also possible to add the maximum frequency that the driver can use.

Once this additional information has been provided, it is possible to build a Linux project that will provide a driver */dev/spidev1.0 (1=number of slave, 0=spi register number)*, in which it will be possible to perform transactions. In order to ensure that the SPI communication protocol works correctly up to the PMOD connector, it is worth carrying out a small test that will eliminate many doubts in case of faulty functionality. It is simply a matter of forcing a short circuit between the MOSI and MISO ports. In this way, when we transfer information through the MOSI, we will be able to see in the MISO readout if it is possible to read the same transferred values identically. If this is the case, then the driver is correctly configured for the use of the AXI Quad SPI.

```
./spidev_test -v
spi mode: 0x0
bits per word: 8
max speed: 500000 Hz (500 KHz)
TX | FF FF FF FF FF FF 40 00 00 00 00 95 FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF F0 0D  |
```

```
RX | FF FF FF FF FF FF 40 00 00 00 00 95 FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF F0 0D  |
```

If in reading the MISO, we are constantly reading 0's, it means that the MOSI or the MISO or both, are not properly connected or configured.

```
$ ./spidev_test -v
spi mode: 0x0
bits per word: 8
max speed: 500000 Hz (500 KHz)
TX | FF FF FF FF FF FF 40 00 00 00 00 95 FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF F0 0D  |
RX | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
     00 00 00 00 00 00 00 00 00  |
```

To perform the SPI communication tests, we used an official Linux test file [46] that can be found in different directories and is called *spidev_test.c*. It configures the transaction parameters, such as mode, number of bits per word and speed before sending a dummy data vector to the slave. A readout of the slave's response is done directly afterwards. We used this file until we succeeded in sending the right information to the sensor to receive the expected results. Once this was done, we reused the useful functions of this test file to make them compatible and functional with our main application.

The final configuration used for the spidev driver is mode 0, which indicates that the clock phase (CPHA) and the polarity clock (CPOL) are set to 0. The number of bits per word corresponds to the bit width configured for the AXI Quad SPI, i.e. 8 bits, and the speed has been set to 500kHz, because the MPU-6000 sensor operates with a maximum frequency of 1 MHz.

7.7 Results

This section is intended to present the results obtained after all the steps detailed so far. It is the demonstration of the functioning of the whole system developed. First, we will go over the functioning of the model at the physical level with the NAB machine temperature data used so far. Since these data are known and could be used in simulation, it is the best way to validate if the system works at the physical level. In a second step and with the addition of data acquisition via an accelerometer, we will go through the results obtained on real acquired data.

7.7.1 Based on saved data

As mentioned, to validate in a concrete way if the final developed system works at the physical level, it is necessary to test it with known and approved data during the different previous steps, in particular the C/RTL simulation step. In theory, the results obtained at the physical level must be identical to those obtained in simulation. For this reason, in a first step, the results of the physical implementation will be presented on the basis of the NAB data, listed in a CSV file.

It has already been possible to observe previously that by simply testing the generated HLS IP block with a few known values, it seems to provide expected results. However, the final validation must be performed on the whole test dataset. Therefore, it will be possible to save the reconstruction results and plot the values in order to observe if they overlap with the simulation results presented earlier, during the V&V phase.

At the end of the development of the main application in C and based on an embedded Linux environment, it was possible to manage the data of the CSV file saved in the SD card according to the process described previously. However, in order to save all the data in the main memory and avoid the length limit of the memory mapped described above, it was necessary to increase the value of the Linux parameter *max_map_count*. It is necessary that this value remains on a multiple of 4, which corresponds to the memory page size of 4kB. We modified this value with the following command : *sysctl -w vm.max_map_count=262144*. By multiplying by 4 the initial maximum value of 65'536, it is now possible to save 65'536 entries instead of 16'384. With this new limit, it was possible to save all 20'400 entries of the test data file. The problem with doing this is that every time the system is rebooted, the default value is reassigned as the maximum value. If the data acquisition window is to be larger than 16'384 entries, it will be necessary to change this value in the rootfs used to boot the system so that this value does not change again.

Once the data is saved in the memory, it is possible to transfer and transform it before displaying the results from the terminal. The saved results are presented in the graphs below.

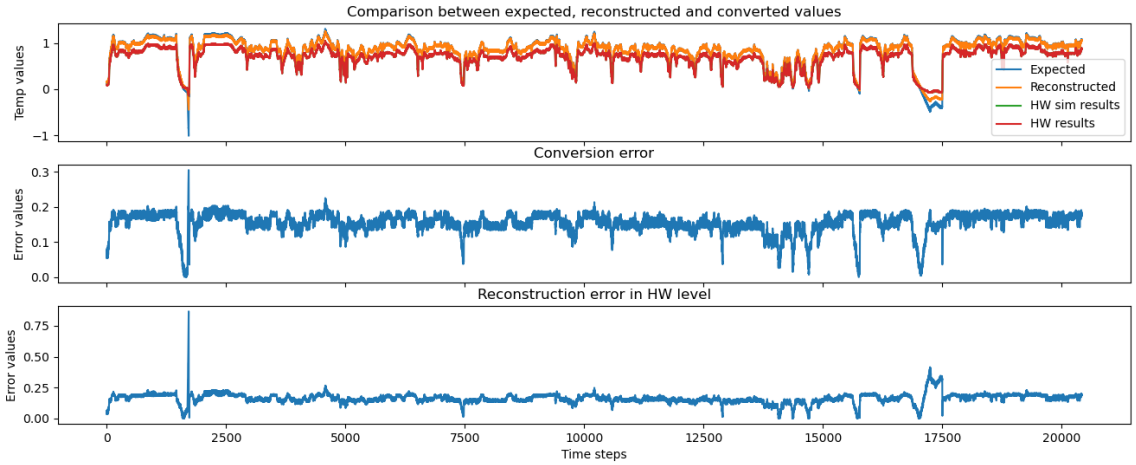


Figure 7.30 Plotting results down to the hardware level

The above results are presented in the same way as the results obtained during the V&V phase. It was added on the first graph, the results recorded in the SD card at the end of the reconstruction process on the FPGA part. It is the red plot and it can be observed that it totally overlaps the green plot which represents the results obtained in simulation. The second graph represents the conversion error between the reconstructed values at the Python level and those at the hardware level. We can see that this error is always small and satisfactory. The last graph represents the reconstruction error between the real values and the values obtained after reconstruction at the physical level in the FPGA. That is to say the error between the blue and red plots of the first graph. The last graph is therefore the one that would be used to detect anomalies in real time with the FPGA.

It is now possible to affirm that the model converted to RTL level reproduces in an identical way the results obtained in C/RTL simulation and validated previously during the V&V phase. All the operations to be performed on the data have been developed in the main application. The only thing to apply at this point is that instead of recording in the memory, data already acquired, it is necessary to record data from a real time acquisition through a sensor. These data acquired in real time can then undergo the same operations that led us to validate the results obtained from the physical implementation.

7.7.2 Based on real-time acquisition using SPI device

A good step to finalize this project is to apply the developed tool to a real application. The last results to be presented in this subsection of results are those from a data window acquisition through the MPU-6000 accelerometer.

To carry out this stage of real-time testing, it is necessary to have a test bench that will serve as a physical structure to be analyzed at the vibratory level. We took a simple structure whose operation is based on a turntable.

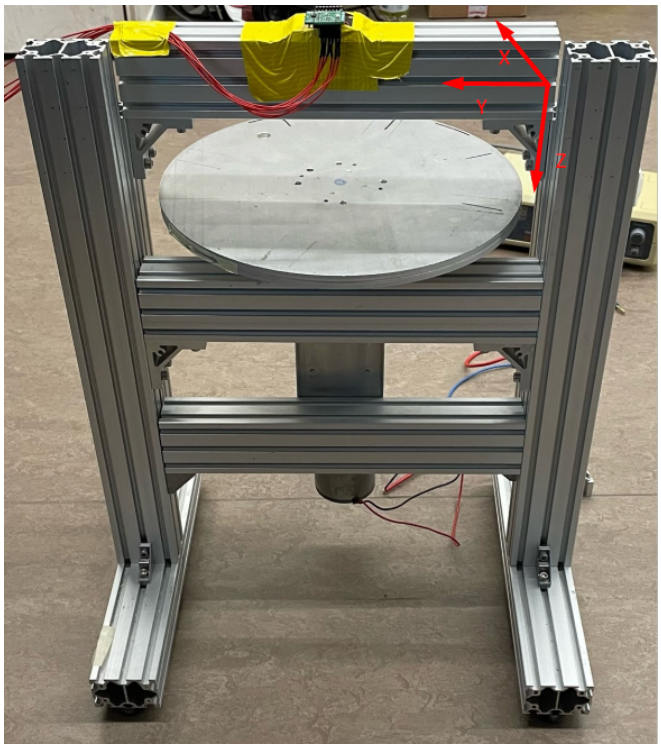


Figure 7.31 Test bench for real application

The accelerometer was placed on top of the structure. This position allows to easily absorb the vibrations generated by the system on the X axis of the sensor (see axis references on the figure 7.31). It is on this axis that we will concentrate for the acquisition and the analysis of the data.

The training data should represent the normal behavior of the system, i.e. without anomalies. We have defined a healthy use of the test bench with the following specifications.

- Voltage at 24 V
- Platform rotation speed at 2.94 rps
- Acquisition at 1 kHz

The speed was calculated based on the test case with friction at each turn of the plate. It was possible to define the number of revolutions per data window which is 48. We know that it takes 16.384 seconds to acquire all the data of a window, therefore,

$$\frac{48}{16,384} = 2.94rps$$

The first thing to do was to record several windows of healthy data. We recorded 327'680 points ($20 \times 16'384$ (window size)) in the SD card in *.dat* format in order to reuse this offline for training the model with Python Keras. Following the standard process, the complete set was split and normalized. The first 90 percent of the complete dataset was used for model training. The other 10 percent was used mainly for validation of the simulation at the C/RTL level. There is no interest in testing the latter at the Python level, as it is also considered to be sound, without anomalies. So there is no reference in this dataset, which allows to know if the model correctly detects anomalies. This is intended, as we want to do this on the FPGA with an already tuned and known model using real acquired data.

The model that was trained is model 28, which we have analyzed and detailed so far. At the end of this, we were able to collect useful information for the continuation, in particular a plot of the cost function, the threshold value as well as the useful values for the normalization of the data.

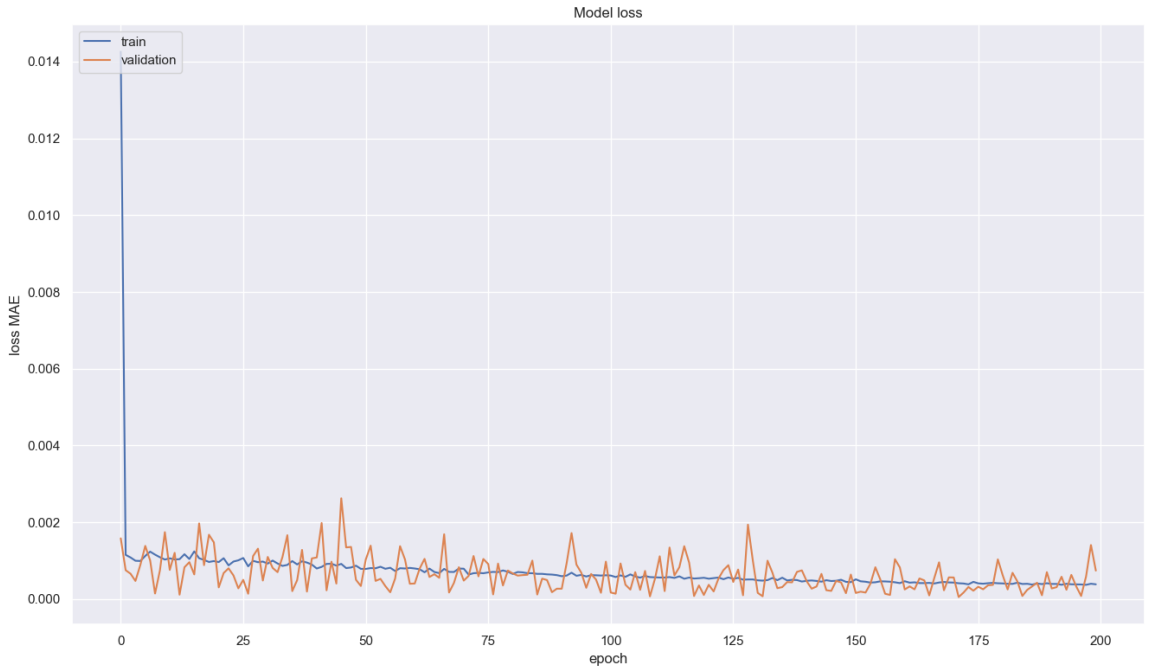


Figure 7.32 Loss function training test bench data

We notice that the validation plot has a low value since the beginning. This is why the scale is small enough to see fluctuations on this plot. The model was able to learn to reconstruct this type of data very quickly. By continuing in the number of epochs, it was possible to further reduce the error in a less drastic way. We can consider that the training was correctly done after 200 epochs in view of the final error value very close to 0.

Concerning the anomaly detection threshold, it should be noted that the method detailed in the report works well for detection on a Python model. However, it is not possible to use the same threshold value on the hardware level, because the accuracy is not the same. It is better to redefine a detection threshold by experiment or by using the same method as at the Python level, but this time with the error values of the hardware level. The latter was done to define the threshold of detection of anomalies which is 0.068. This threshold value will be used in the following test cases

Once the training was completed, the *.json* and *.h5* files could be retrieved in order to perform the conversion of the model to HLS level, thanks to HLS4ML. The rest of the process followed without any problem. Except that a bug occurred when exporting the HLS IP from Vivado HLS. This was a temporary bug due to the year change. A temporary solution to solve this problem was to change the year of the computer used by choosing a date in the year 2021. In Vivado, we created the same design as the one presented before, but without the system ILA, which is no longer useful for the finalization of the project. Of course, the system ILA has no impact on the functioning of the system.

Once the Linux project was created with all the files necessary to boot the image on the target through the SD card, it was possible to test our model at the hardware level on different cases of anomalies with an acquisition and a data processing in real time.

Test cases

5 scenarios have been tested in order to analyze the quality of the detection model physically implemented in the FPGA. Except the first one, which is a scenario representing a normal behavior of the test bench, the others are behaviors that differ from the one on which the model has been trained. The unit of the acquired values are in g and normalized.

- **Train data reconstruction**

  As a first step, it is interesting to perform the hardware level reconstruction of the training data. This allows us to have a good reference to compare the results of the test cases. It should be noted that it is on the basis of this data that the detection threshold has been calculated.

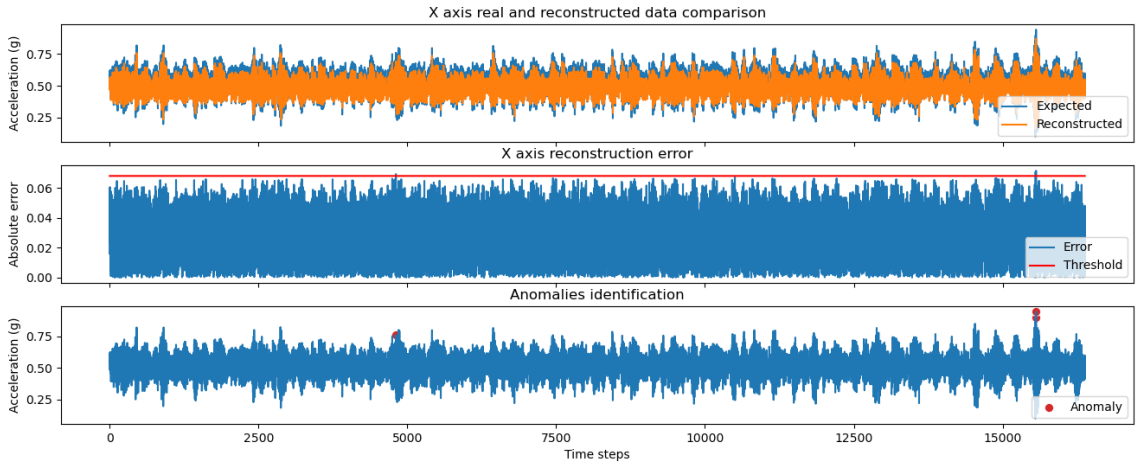


Figure 7.33 Train data reconstruction

The defined threshold detects two negligible anomalies in the training set, because it was not defined with respect to the largest error, but to the rightmost value of the normal distribution of reconstruction errors.

- **Normal utilization**

  The first test case is performed under the same conditions as the data acquired for the model training.

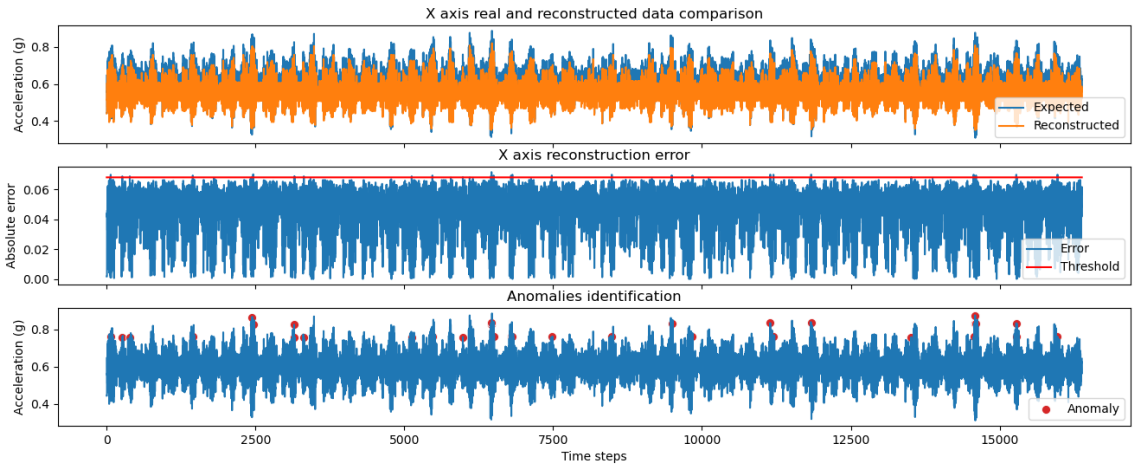


Figure 7.34 Normal utilization test case results

It is possible to see that the reconstruction error is correct and that only a few slight anomalies are detected. With an analysis of the mahalanobis distance, it would have been possible to justify that these anomalies can be ignored because of the low weighting they would have had.

- **Add a contact to each round of the platform**

  The second test case was to add a contact between two bodies fixed on the test bench structure. The contact is made once per turntable and will lead to a propagation of the vibration to the sensor. With our model, we should theoretically be able to detect a regular anomaly that appears at each turn of the platform.

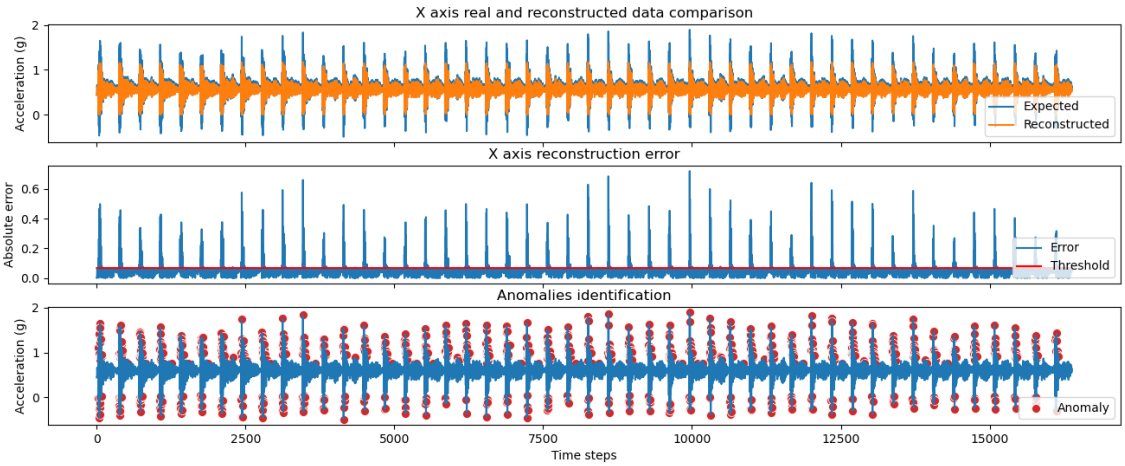


Figure 7.35 Friction test case results

We observe that this is the case, the model does not manage to correctly reconstruct the behavior of this contact, which leads to a significant reconstruction error when it occurs.

- **Run the system at a lower speed**

  Another test that has been performed is to decrease the rotation speed of the plate, compared to the speed used in a normal case. This leads to a slower behavior and consequently lower vibrations. To do this, we reduced the voltage from 24V to 7V, which leads to a speed of about 0.85 rps and here is the result obtained.

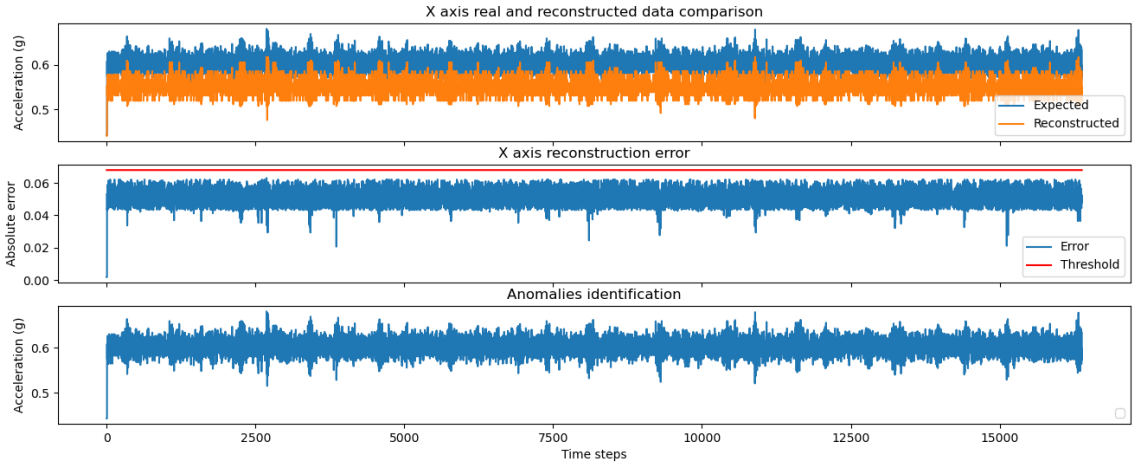


Figure 7.36 Lower speed test case results

It can be observed that the behavior of the model is different from the normal behavior. If we look at the reconstruction errors, they are well below the defined detection threshold. This can be explained by the fact that the model is probably not smart enough to recognize the change in correlation between the points it receives and those it has been trained on. If these changes are not aberrant enough and the model has seen these values individually during training, the model will simply succeed in reconstructing the values without noticing that the overall behavior is different and that is probably what is happening here. In the context of an embedded neural network, resource limitations play a role. The more intelligent and reliable a network has to be, the more resources it will cost. Another project focused mainly on Deep Learning could be considered in order to improve this network and allow it to detect this type of anomalies, if the need is fundamental.

- **Run the system at a higher speed**

  Unlike the previous test, this one was performed based on an increase in the speed of the platter. The test is based on an operating voltage of 30V, which represents a rotational speed of approximately 3.675 rps.

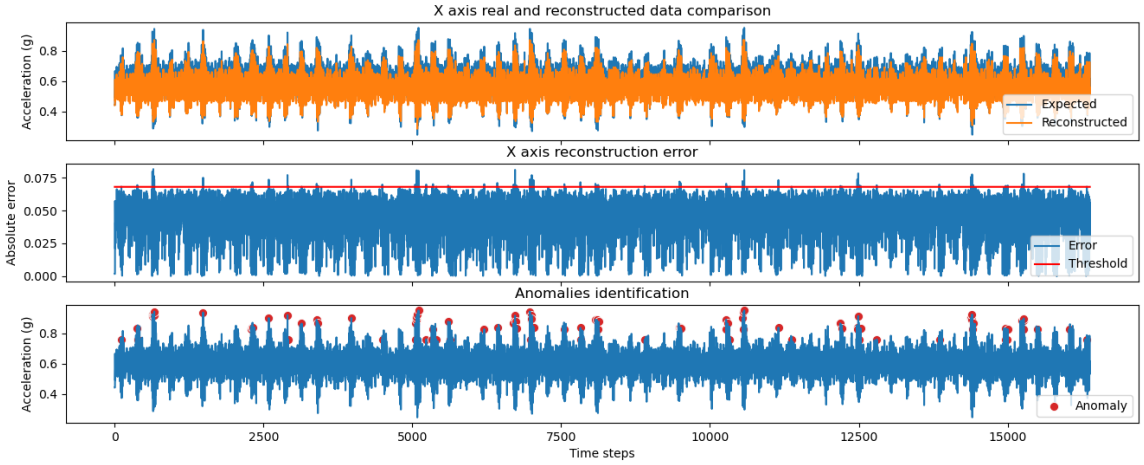


Figure 7.37 Higher speed test case results

Here, we can observe this time that a faster behavior of the system generates several relatively important anomalies. These are probably due to the fact that the vibrations are more accentuated and less damped, contrary to the previous test with a lower speed. The model detects this type of anomalies more easily than in the opposite situation.

- **Run the system with speed variation**

  A last test was carried out, this time with the acquisition since starting of the system and a more or less abrupt variation of the speed of rotation of the plate through the injected voltage.

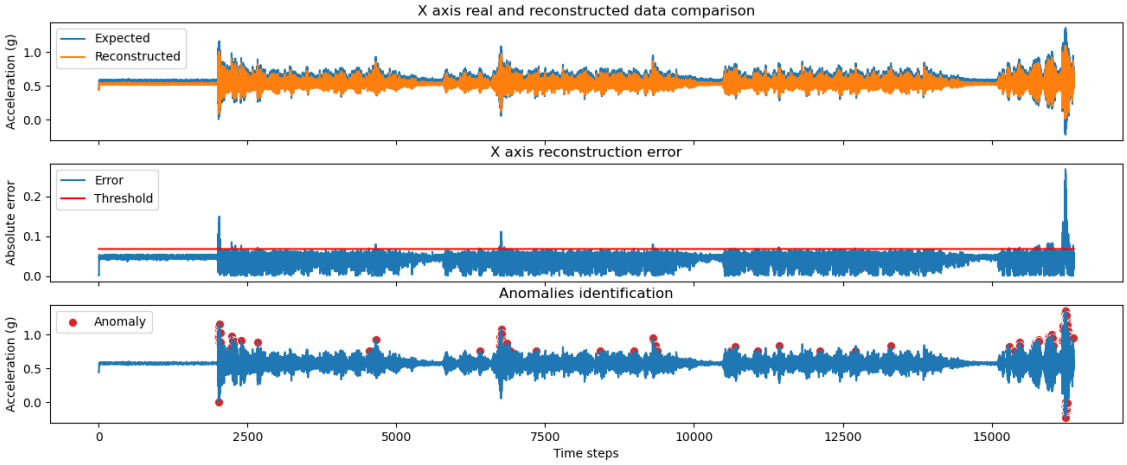


Figure 7.38 Speed variation test case results

Firstly, it is possible to visualize very easily the sudden movement of the motor starting up to the maximum speed, which will bring an important peak in the vibration level. This is easily detected by the reconstruction error. Secondly, the speed change sequence is like this.

1. Decreasing the speed to the minimum
2. Increase the speed to the maximum

These two steps were carried out 3 times. We observe several anomalies with a mixture of importance when we are at maximum speed. It is also possible to see an anomaly that appears just at the beginning of the first two speed runs. This could be an indicative sign not to be neglected, but it is very difficult to affirm it in a sure way.

This step of real time testing on data acquired from a physical test bench allowed to put forward the fact that the system is functional and meets the requirements of the project. With it, it is possible to detect the majority of the anomalies that can be encountered. However, we noticed that the developed and implemented model is not smart enough to detect all types of anomalies and especially those that are more subtle. Here we have tested some scenarios, but there are obviously many others. It is also important to note that it is sometimes very difficult to interpret some anomalies without

having experience in the field of predictive maintenance. The tool and the algorithm are mainly used as indicators, but a good part of it is based on the interpretation capacity of the human in charge.

7.8 Conclusion

To carry out this hardware implementation phase, it was necessary to go through several steps. The first one was to succeed in communicating with the generated HLS IP block in order to demonstrate that it was functional. For a question of speed of implementation, this was done in bare metal with some test values. Then, we were able to do the same with an embedded Linux environment and this time with a whole set of recorded data. Of course, in the meantime it was necessary to boot the Linux image into the target correctly. As far as the hardware design is concerned, we tried a solution based on the use of an AXI DMA and it turned out that this was not necessary to meet the requirements of this project. Nevertheless, in the context of a future optimization project, this strategy could be beneficial. The final strategy was to use AXI GPIO to control the HLS IP block. As far as data acquisition is concerned, it was done on the basis of SPI communication, on the FPGA side. That is to say with the use of the block IP AXI Quad SPI, which serves as SPI interface. The acquisition is done at 1kHz and we manage to perform the entire process of the system in 23 seconds. That is to say, from the beginning of the acquisition of the vibration data, to the display of the results in floating point format on the Petalinux terminal.

In order to optimize the total processing time, it is necessary to have an analysis of the time that each operation in the process takes. This will make it easier to identify a bottleneck for an optimization objective.

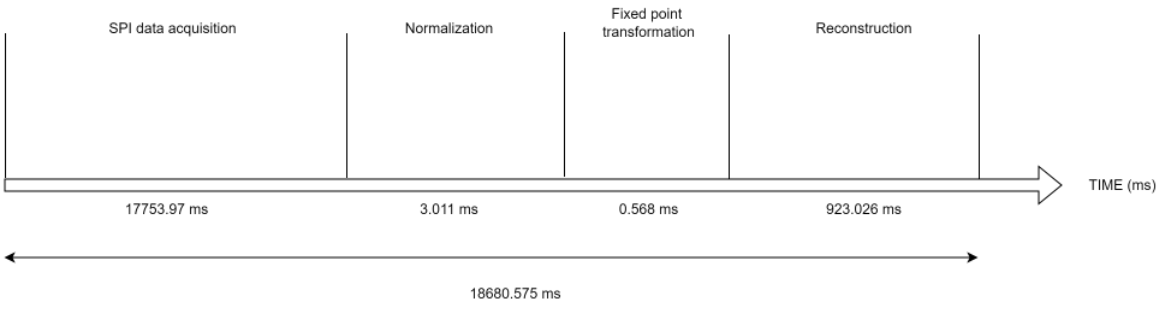


Figure 7.39 Execution time analysis

Data acquisition from the sensor is done at 1kHz. A delay of 1ms has been added to the acquisition loop of the main program in order to acquire each time a new data. What takes the most time here is the data acquisition, but this depends on the type of sensor used. It is not really an aspect to be optimized. To have an optimal behavior, the choice of the sensor should depend on the execution time of the rest of the operations. In fact, if it is possible to use a sensor with an acquisition frequency that is close to the period needed to pass all the other operations, it would be possible to do real time acquisition and processing with maximum speed. Here we see that the acquisition frequency is much slower than the total time of the other operations. This means, that it is possible to let the sensor work independently and to process each of the data faster than the acquisition. This could be further optimized by using a sensor that would approach an acquisition frequency of 17 kHz, which is the current total frequency of the other operations. Then, without taking into account the acquisition, we see that the second most time consuming operation is the reconstruction. An optimization work at this level would allow to decrease the processing time and to use faster sensors.

This phase of physical implementation of the Deep Learning model in the FPGA with all the operations around it was probably the most critical of this project, together with the simulation phase. We had validated during the V&V phase, that the model converted at the RTL level provided satisfactory results and consistent with the results obtained at the Python level. The criticality of this hardware implementation phase was to succeed in reproducing the same results as what we had in C/RTL simulation. Knowing that we used a prototype solution of HLS4ML to convert our Python model to the HLS level and that LSTM networks are relatively complex to implement, there was a chance that some things would get stuck.

However, at the end of this step, it is now possible to affirm that the results obtained in C/RTL simulation could be successfully reproduced at the hardware level. It is thus possible with the chosen FPGA, to send vibratory data in the model and to recover the expected reconstructed results.

The model has been developed to work on vibratory data. It is not guaranteed that it will perform well on all types of data. Nevertheless, the proof of concept is functional. It is therefore possible to tune the Deep Learning model for different use cases.

8 Conclusions

Contents

8.1 Project summary

If we take the initial representation of the project scope in appendix B and modify it according to the real realization of the project, it gives the updated representation in appendix C.

Various changes took place in order to achieve a homogeneous project with the most suitable tools for its realization. At the beginning of the project, there were many unknowns. It was therefore foreseeable that we would have to make some changes to the initial strategy during the project. The first big change was to go through a conversion to HLS. Initially, there was to be a single conversion from the main model (Matlab, Python, ...), to the RTL. This change took place, because LSTM networks are hardly compatible with the majority of RTL synthesis tools. Nevertheless, this HLS switch had many benefits. The optimization phase (quantization and pruning) could be done directly with HLS4ML, during the conversion to HLS level. Then, the verification step was initially planned with the *cocotb* method, directly at the RTL model level. However, the fact of passing by the HLS allowed us to use a test bench written in C provided by HLS4ML to test the HLS and RTL conversion. The verification then took place at the highest modeling level. That is to say that it was necessary to find a tool allowing to quantify the detection quality of the Keras model. To do so, we used the NAB scoring mechanism. Finally, we were able to define a type of communication for the data acquisition, which was SPI.

In general, this project required the development of several skills in order to complete it successfully. It is possible to decompose the project in two big parts. The first one was the development of the Deep Learning model at the Python level so that it can be used for anomaly detection. The second was the hardware and software development of the embedded system. The project also went through a conversion and V&V phase which are not negligible. It is a project that has brought a lot of experience and reflection. At the end of it, multiple aspects have been realized and clarified, which has allowed to justify and put forward a solution that meets the vast majority of the initial requirements of the project. However, it is for the moment a proof of concept that can continue to evolve with a view to optimizing performance, resources and costs. The results obtained are interesting and promising for an edge computing solution in the field of predictive maintenance.

8.2 Comparison with the initial requirements

The main objective of this project was to obtain a solution adapted to the needs with a minimal cost. The company Koord Sàrl provided the functional and non-functional requirements of the tool to be developed with a starting idea based on the use of an LSTM network. The FPGA-oriented solution is based on the fact that the project of the last semester was linked to it in a V&V framework. For this reason, the project focused on an FPGA-based predictive maintenance tool. The goal was to evaluate if it was a viable solution and if it was optimal, all in terms of requirements. If it is possible to do the same thing with a simple processor that is much less expensive, the goal of the project was to have a practical experience that allows to evaluate this and to choose the best solution, with a view to commercialization. This is why it is important to take a global view of the project and make this analysis.

Master Thesis Project
Requirements

Functional requirements

N°	Function	Criterion	Requirement	Flexibility
1	Analysis of the results	Data acquisition time Data processing time Display results time	10 minutes max.	F1
2	Algorithm training	Acceptable time before analysis phase	1 hour	F3
3	Real-time analysis	The tool must allow an online test	Available	F0

Non-fonctional requirements

N°	Function	Criterion	Requirement	Flexibility
1	Hardware composition	Maximal price of the tool	30.-	F1
2	User utilisation	Safety utilisation	Data security	F0
		Complexity of utilisation	Userfriendly	F0

Flexibility level	
F0	No flexibility
F1	Low flexibility
F2	Good flexibility
F3	Strong flexibility

Figure 8.1 Project requirements

8.2.1 Functional requirements

1. **Processing time**

The overall processing time for a data window is defined as a maximum of ten minutes from the start of data acquisition to the monitoring of the results. With the implemented solution, it is possible to process 16'384 data in about 23 seconds. The acceleration in the FPGA part is very fast with a latency of a few hundred nanoseconds, which allows the overall processing to be done in a very short time. What takes the longest time at the moment is the data acquisition which is done at 1kHz maximum with the sensor used. As explained, an optimal solution would be to use a sensor that has an acquisition rate close to the total time to perform the remaining operations. In addition, it is still possible to optimize the hardware implementation by adding an AXI Lite interface to the HLS IP block in order to be able to use it efficiently by combining it with an AXI DMA IP for example. This would reduce the CPU tasks and therefore the latency. The more we can decrease the latency of operations, the faster we can use a sensor for acquisition with accurate visualization of the results. To achieve the results obtained, it was useful to configure a maximum parallelization of the calculations with a minimum reuse factor at the level of the conversion into HLS. A decrease of the parallelization would also meet the requirements of the project.

On the other hand, due to its nature of being optimized for sequential operations, it is obvious that an implementation in a standard processor would result in a higher overall processing time with a loss of performance, but it is possible that this is still within the maximum time allowed. It also depends on the minimum

number of data to be included in a window to be processed in less than ten minutes and this information is not necessarily known in advance. The goal is to process as much as possible in order to be as accurate as possible.

2. **Training time**

   As far as training time is concerned, the requirement was about one hour with significant flexibility. Therefore, it is not a critical requirement. Obviously, this time depends on the model parameters, such as the batch size and the training parameters, such as the number of epochs. It also depends mainly on the amount of data to be trained. As a reference, the training of the final model was done with 300'000 inputs and with the configuration of model 28 on 200 epochs. This requires a total training time of about 30 minutes. The fact that this requirement is not fixed leaves some flexibility in the training of a possible new and more consequent network. In conclusion, this requirement was met for training the final model on a sufficient amount of data to meet the needs of the other requirements.

3. **Online utilization**

   Before this project, Koord practiced predictive maintenance offline. That is to say, with an analysis that is done after data acquisition and on a computer. The requirement to have a tool that operates in real time is therefore important for this project, as this is part of what will make the difference between this project and Koord's current activity.

   To say that a tool must work in real time is not precise enough. It is necessary to define what processing time is considered as such. For this reason, this requirement must be combined with the first. We consider the processing of a window of input data in less than ten minutes as real time. There are no requirements regarding the size of this window. It should simply allow to process a maximum of data as fast as possible to lose a minimum of data during the acquisition. This window size also depends on the acquisition speed of the sensor. A slow sensor will process small windows and start the process again as quickly as the next sensor acquisition. A fast sensor will require larger windows in order to analyze complete series. Obviously, in the case of the latter, data will be lost during the processing time. A work of optimization of the processing time and memory management would allow to limit this loss.

   Regarding what has been done in this project to meet this requirement, it is possible to process a window of 16'384 data in about 23 seconds. This, from the beginning of the acquisition, until the display of the results. It is therefore possible to affirm that the tool works in real time.

8.2.2 Non-functional requirements

1. **Hardware solution**

   The final perspective of this project is to move towards a commercialization of the solution. For this reason, the financial aspect takes an extremely important place. It is therefore necessary to evaluate, with the experience gained from this project, whether the solution is optimal in terms of costs and performance.

   The hardware possibilities for embedding a machine learning model are varied and the most appropriate are the use of FPGAs, ASICs or GPUs, notably for their capacity to parallelize operations. There is also the possibility of implementation on standard processors or microcontrollers, which do not allow parallelization or not much, depending on the number of cores. They are designed and optimized to perform sequential operations. However, they are generally cheap and consume little energy. In this project, the experience gained is based on the use of a processor and an FPGA, integrated in a chip (SoC). It is therefore not obvious that an implementation based on other solutions, perhaps less expensive, will meet the requirements of the tool. Nevertheless, thanks to this experience, it is possible to provide a relatively interesting and useful analysis.

   The use of an SoC offers the advantage of having everything in the same chip, which means that the communication latency between the CPU and the FPGA is reduced. This aspect is not necessarily very relevant for this project as the processing time is relatively large. This means that another option could be to combine a CPU with an FPGA separately. On the other hand, the development time might not be the same and the manipulation between CPU and FPGA might become more complicated, because it would not be possible to use fully adapted tools, as it was the case for this project. There are SoCs, which could support our model as it is with a price that goes around 160$, like those of the Zynq-7000 SoC series from Xilinx. It is still possible to reduce the resource consumption of the reference model by increasing the reuse factor and decreasing its precision. After that, it is possible that this one will fit into SoC ranges that go for less than a hundred dollars. Another interesting possibility would be to implement everything in an FPGA and thus not use an SoC. In the context of this project, the operations performed by the processor are not extremely complex, which means that an FPGA could manage this itself, in addition to the acceleration of the algorithm. Unless an additional processor is used, it would not be possible to use the embedded Linux eco-system with this option.

   An important aspect of using an FPGA is its reconfiguration capability. This characteristic allows, for example, to process different types of data, by reprogramming the system in a hardware way. This avoids having to develop other solutions on other equipment. It is also possible to partially reconfigure a part of the FPGA to adapt it to other applications. Secondly, and in comparison with GPUs, it is important to take into account the fact that FPGAs have a very long lifespan which can be measured in decades, whereas a GPU has a lifespan of 5 years on average. In our case, this is an important aspect since it is necessary

that the developed tool works at all times, over the entire life of a system to be analyzed. With this same constraint, it is important to emphasize that FPGAs will consume much less energy than GPUs, which is not negligible again for a tool that must operate regularly over the long term. According to a Microsoft study [26], FPGAs are about 10 times more efficient in terms of energy consumption than GPUs. Finally, FPGAs are more suitable than GPUs for long term applications, which leads to the conclusion that its use in a predictive maintenance tool is not very appropriate. It is also important to know that some complex systems requiring the processing and monitoring of several subsystems must be composed of several GPUs. Most of these GPUs will have to be changed after 4-5 years of use, which makes this solution not very economically viable compared to an FPGA which alone can manage different subsystems at the hardware level with an efficient power consumption.

In addition to the solutions currently in use, there are relatively new solutions to experiment with that could prove interesting at prices that are much closer to the requirements of this project. These solutions are for example TinyML, which is a machine learning technique allowing the integration of models on low power consumption systems, such as microcontrollers or sensors. The TinyML board from Syntiant costs 35$ and has all the components to do predictive maintenance based on a machine learning model. There is also the Coral Edge TPU (Tensor Processing Unit), which is an ASIC developed by Google and specially designed and optimized for machine learning operations. For example, the USB Accelerator, which has an Edge TPU costs about 60$ and can be used for the inference phase of a machine learning model on an existing system. Both solutions are based on Tensorflow Lite, an optimized version of Tensorflow dedicated to the application of machine learning models on edge devices. With the work done, it is not necessarily difficult to test if these solutions are adapted to the requirements of this project, as it is possible to reuse the Keras model already developed to do so.

Finally, the tool developed in this project is mainly based on the operation of a Deep Learning model. If the use of this tool is intended only for its main function, without necessarily needing great flexibility, as an FPGA could offer, then the most optimal solution would be to use an ASIC or a microcontroller specially developed and hardware optimized to support machine learning algorithms. It is relatively easy to develop a TPU or TinyML based system on a Keras model, like the one designed during this project. At the hardware level, it is for example possible to use a TPU as a co-processor to a main system, SBC Linux, like a Raspberry Pi. A USB Accelerator is available to do this with an affordable price. As for TinyML, the price of the Syntiant Tiny Machine Learning Development Board is also extremely interesting and already has components useful for predictive maintenance, like an accelerometer. Both solutions are very recent, so it is difficult to estimate the performance that can be obtained without having tested the solution.

The use of an FPGA is not to be neglected, it is largely satisfactory in terms of latency, resource consumption, etc.. Nevertheless, it seems to be a solution that is over-adapted to the needs of this particular project. Indeed, if latency is not a

priority, if price is a priority and if the reconfiguration capacity of an FPGA is not exploited, the use of a less flexible solution, but optimized at the hardware level to the main needs is more appropriate.

2. **User space**

   Concerning the requirement that concerns the use of the developed tool, two aspects were put forward. Firstly, the security aspect of use. This one is rather oriented towards information security, because at the physical danger level, the tool does not bring anything particular. The developed tool works in edge computing, contrary to many other predictive maintenance tools where part of the operations are done through a cloud server. This is an important advantage of this system, because the fact that everything is done locally totally eliminates the risk of data theft. Obviously, if the goal is to work with data that is not very sensitive, this requirement is less important, but in the case of more sensitive data, this is a significant advantage. This was also a requirement for this particular project, so it was possible to meet it.

   The second aspect is the "userfriendly" side of the developed system. For the moment, it has a petalinux graphical interface by default. The application is executed from the terminal at the desired location. In fact, the user aspect has not been put forward during this project. The priority was first to have a functional tool. An extension of this project could be done to add an option to launch the application more easily and to display the results more clearly. For example through a graph that plots in real time. A complete monitoring strategy could be reviewed and developed.

Finally and apart from the requirements concerning the costs of the tool and the user space, it is possible to affirm that all the requirements of this project have been successfully met. Indeed, they were all taken into account from the beginning of the project in order to be able to think about the design of the tool in an optimal direction. An extension of this project with a focus on the optimization of the neural network would allow to get closer to the initial price required. Nevertheless, it is important to know that a price set at 30 CHF is difficult to reach if we want to embed a powerful neural network on an FPGA. As far as the user space is concerned, it is simply a less critical aspect in this project that could not be included in the development time of this work. Additional time would allow for easy improvements in the use of the tool.

8.3 Encountered difficulties

During the realization of this work, many difficulties were met and overcome. These have allowed to acquire an important experience in different fields.

First of all, it is important to know that the initial experience in the fields of Deep Learning and embedded FPGA was relatively weak. It was therefore necessary to do a relatively important documentation work in order to understand the different important concepts and to get to an acceptable level before entering the project.

The first difficulties appeared early in the realization of this project. The initial idea was to design and implement an LSTM network using Matlab. It turned out that the tool did not support the implementation of LSTM networks. It should be noted that it is a relatively complex network and difficult to embed. For this reason, it was very difficult to find a solution for implementing an LSTM model in an FPGA. Since the problem came from the type of network, one idea was eventually to try to do anomaly detection on time series with another type of network. The study from the article [23] shows interesting results that can be used as references. It turned out that the LSTM network is the most suitable and the best compromise between performance and resources. So, the ideal was to find a conversion solution that supported LSTM networks. After a lot of research and questioning, it turned out that a prototype version of the open source tool HLS4ML was adapted to support this type of network. The challenge was to find this solution and ensure that it would meet the needs of the project.

Another difficulty we had to face was to find a way to verify and validate the developed Deep learning model in order to quantify its performance in terms of anomaly detection. Using a dataset with no known anomalies was not sufficient to concretely analyze the detection capacity of the model. Using a dataset with known anomalies was better, but still not ideal, as a solution was needed to obtain an analysis based on quantitative results. This was done with the discovery and use of the Numenta Anomaly Benchmark solution. The difficulty was mainly to find a solution adapted to the verification of an anomaly detector and not directly the use of NAB. It is not a question of a functioning that can be verified with standard methods, but it is a question of analyzing if a model behaves correctly when looking at one or several real data sets.

Then, there were difficulties related to the compatibility of the tools used starting from a solution based on HLS4ML. From the start, the official solution recommended that we use a version of Vivado HLS between 2018.2 and 2020.1. For this project we used Vivado HLS 2019.2, which means that we had to be careful to use compatible versions for Vivado, Petalinux and Vitis. In addition, the Vivado documentation recommends using Ubuntu 18.04 when using Vivado 2019.2. By not paying attention to all of this or neglecting certain points, there is a good chance of wasting time on problems that could have been avoided. Still on a compatibility aspect, a problem occurred with the RAM memory of the computer used. It was limited to 16Gb. When synthesizing a large model on Vivado HLS, this limit is largely exceeded, up to about 26Gb. It was necessary to install everything and to redo the process on another computer with a higher RAM limit.

Finally, slight problems were encountered during the hardware implementation phase, in particular when it was necessary to drive hardware blocks from the processor with the AXI interface. This means driving the PL part from the PS part. This considerably increased the difficulty compared to a system driven only by a processor. However, it was a very rewarding experience. It should also be noted that everything was based on an embedded Linux environment, which also increased the difficulty compared to a bare metal based application. This part was also very enriching in terms of knowledge acquisition.

There were still many difficulties that were tackled, but all of them were very instructive and indirectly allowed this project to evolve in the right direction thanks to the acquisition of new useful experiences. To complete a project of the magnitude of this one without having to overcome difficulties would probably have been a personal failure.

8.4 Future perspectives

Another aspect to be discussed in this conclusion are the future perspectives that can be linked to this project. First of all, it should be remembered that this project started from a concept that the company Koord Sàrl wanted to develop further. This concept was implemented on an FPGA, which was a request from me in order to keep a link with the semester project done previously. The objective at the end of this project was to have a functional concept in order to evaluate the results and to have a reference to choose an optimal solution in an optics of commercialization.

We have thus obtained a predictive maintenance tool based on a Deep Learning model implemented on an FPGA that is functional. However, it is not necessarily optimal in terms of performance and resource consumption. An important work of optimization at the modeling level as well as at the hardware level could be done in order to obtain a tool that could be presented as the prototype of a new marketable product. To get closer to this, it is necessary to optimize this concept on three axes, which are the improvement of the performance of the neural network in terms of anomaly detection, the decrease of the resources necessary to make it work on an embedded system and the decrease of the global cost price of the tool, which is closely linked with the resources consumption. It could be added the reduction of the global processing time of the tool with a change of the hardware strategy, but it is not a priority considering the good results already obtained at this level.

Another perspective is to test this concept on another hardware base. That is to say with a use other than a FPGA SoC. It would be interesting to try to implement the developed model in a solution based on Edge TPU or TinyML. This would allow to compare the performances of a less expensive system and to have a more precise idea of the optimal solution to approach in a marketing perspective.

Finally, for this project, the requirements were based on the use of a single type of signal to be analyzed to obtain an anomaly detection. The model was therefore developed on a univariant basis. It would be interesting to deepen this concept with the use of several signals. Indeed, this would probably allow to gain in robustness of detection, because an anomaly could also be identified by comparing the correlation of several different signals between them. Obviously, the goal is not to force the system to use many sensors to acquire many different data. Nevertheless, it would be interesting to analyze how a Deep Learning model can learn to recognize correlated patterns between several signals in order to detect anomalies. Also, LSTM networks are very advantageous when working with multiple features. Still with the use of several signals, it would also be possible and interesting to develop several different models processing different univariant or multivariant signals. A post-processing analysis could be added to compare all the results and get out the important information. The use of an FPGA to do this would also be justified because of its hardware flexibility. Several models could be embedded in the same equipment. An example of extension of this project would be

to add a model allowing to process in the frequency domain, the vibratory data. This could be done on the basis of an image classification model of frequency signals. Thus, it would be possible to compare two sources of information and to better interpret the detected anomalies.

8.5 Personal statement

Finally, the last point to address at the end of this project is a personal feedback on it. It is interesting to analyze the personal added value that this project has brought me.

During the first reflections about the choice of the project, I expressed two main wishes. I wanted to work on a project based on FPGA development as a follow-up to the previous deepening project and I wanted a project based on a request from the industry. I absolutely wanted to be able to use this thesis as a professional experience answering a real need of a company. For this reason, my supervisor proposed to contact the company Koord Sàrl, specialized in mechatronic systems. They directly proposed me an important concept for them. We agreed to develop it on an FPGA solution. I really wanted to take advantage of this project to learn as much as possible in the field of embedded systems and computer science before proposing my profile to the job market, for a position that I am passionate about and that stimulates me.

While doing this thesis, I had the chance to work on a project mainly based on machine learning. My curiosity towards the technologies made me want to approach this one for a while. I have thus acquired many skills and experience in the development of neural networks, which I will be able to use to my advantage. This is not negligible when it comes to a technology that has brought a lot of innovation and is currently in high demand. Then, this project led me to have to look for and find viable and adapted solutions to different needs. For example, a big unknown of this project was how to convert a high level Deep Learning model to a hardware level. Several solutions existed, but almost none was viable for the type of network I wanted to implement. So this experience also allowed me to learn how to deal with problems and limitations, to propose solutions or even alternatives. I also acquired additional skills in the field of V&V, because it was necessary to put forward specific parameters to be tested, such as the quality of anomaly detection. Again, it was necessary to find the most suitable solution to test this type of operation. Finally, the second consequent part of this project was the embedded development. I had the opportunity to acquire skills in the development on SoC with a communication in CPU and FPGA. I was able to familiarize myself with the embedded Linux environment and exploit its advantages.

Finally, this project allowed me to start a first specialization in the fields of machine learning and embedded software. It was a goal for me and I am proud to have been able to reach it by taking full advantage of the opportunity I had with this project. Thanks to this project, I was able to work on a real and concrete development for the need of a company. This aspect stimulated me a lot, because I knew that there was an expectation of the results that I was going to obtain. Moreover, I was able to learn to work in an organized way, to do everything to respect the deadlines of a planning and to communicate at best on technically pointed aspects. At the end of this project, I felt a great satisfaction of achievement, because despite the many unknowns at the beginning,

I managed to make the system work, as required and as I wanted. To achieve this, I have provided a substantial investment, which I am proud of. I remain firmly convinced that nothing is impossible when the objective is clear and the motivation is present.

References

[1] Subutai Ahmad et al. "Unsupervised real-time anomaly detection for streaming data". In: *Neurocomputing* 262 (2017). Online Real-Time Learning Strategies for Data Streams, pp. 134–147. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.04.070. URL: https://www.sciencedirect.com/science/article/pii/S0925231217309864.

[2] Avnet. *FPGA vs. GPU vs. CPU – hardware options for AI applications*. [Online; accessed December-2021]. 2021. URL: https://www.avnet.com/wps/portal/ebv/resources/article/fpga-vs-gpu-vs-cpu-hardware-options-for-ai-applications/.

[3] Christopher Olah (Colah's blog). *Understanding LSTM Networks*. [Online; accessed September-2021]. 2015. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[4] chaitanyaPaikara. *HLS4ML - keras-RNN*. [Online; accessed November-2021]. 2021. URL: https://github.com/chaitanyaPaikara/hls4ml/tree/keras-RNN.

[5] Philippe Coussy et al. "An Introduction to High-Level Synthesis". In: *IEEE Design Test of Computers* 26.4 (2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[6] Julianna Delua. *Supervised vs. Unsupervised Learning: What's the Difference?* [Online; accessed September-2021]. 2021. URL: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning.

[7] Mouser Electronics. *Xilinx SoC FPGA*. [Online; accessed December-2021]. 2021. URL: https://eu.mouser.com/c/semiconductors/embedded-processors-controllers/systems-on-a-chip-soc/soc-fpga/?m=Xilinx&sort=pricing&pg=2.

[8] U.S. Department of Energy. *O&M Best Practices Guide, Chapter 5 Types of Maintenance Programs*. [Online; accessed February-2022]. 2010. URL: https://www1.eere.energy.gov/femp/pdfs/OM_5.pdf.

[9] Tolga Ergen and Suleyman Serdar Kozat. "Unsupervised Anomaly Detection With LSTM Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 31.8 (2020), pp. 3127–3141. DOI: 10.1109/TNNLS.2019.2935975.

[10] Harry Foster. *Part 3: The 2020 Wilson Research Group Functional Verification Study*. [Online; accessed September-2021]. 2020. URL: https://blogs.sw.siemens.com/verificationhorizons/2020/11/18/part-3-the-2020-wilson-research-group-functional-verification-study/.

[11] Glenn Hicks. *How Much Is Equipment Downtime Costing Your Workplace?* [Online; accessed February-2022]. 2019. URL: https://www.iofficecorp.com/blog/equipment-downtime.

[12] HLS4ML. *hls4ml tutorial IEEE Real Time 2020*. [Online; accessed October-2021]. 2020. URL: https://indico.cern.ch/event/737461/contributions/4040692/attachments/2123081/3573914/hls4ml_tutorial.pdf.

[13] Edge Impulse. *Syntiant TinyML Board*. [Online; accessed December-2021]. 2021. URL: https://docs.edgeimpulse.com/docs/syntiant-tinyml-board.

[14] InvenSense. *MPU-6000 and MPU-6050 Product Specification Revision 3.4*. [Online; accessed January-2022]. 2013. URL: https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf.

[15] Popescu Ion-Marian. "Acquisition and processing system of vibration signals, in real time and simultaneously, based on FPGA technology". In: *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*. 2016, pp. 289–294. DOI: 10.1109/ICSTCC.2016.7790680.

[16] Goran Jevtic. *Edge Computing vs Cloud Computing: Key Differences*. [Online; accessed November-2021]. 2019. URL: https://phoenixnap.com/blog/edge-computing-vs-cloud-computing.

[17] Keras. *Callbacks API*. [Online; accessed December-2021]. 2021. URL: https://keras.io/api/callbacks/.

[18] Romeo Kienzler. *Using Keras and TensorFlow for anomaly detection*. [Online; accessed October-2021]. 2018. URL: https://developer.ibm.com/tutorials/iot-deep-learning-anomaly-detection-5/.

[19] Whitney Knitter. *Debugging Your Custom Linux Applications Using Vitis*. [Online; accessed December-2021]. 2021. URL: https://www.hackster.io/whitney-knitter/debugging-your-custom-linux-applications-using-vitis-67c022.

[20] Fast Machine Learning Lab. *hls4ml's documentation*. [Online; accessed September-2021]. 2022. URL: https://fastmachinelearning.org/hls4ml/index.html.

[21] Brent Larzalere. *LSTM Autoencoder for Anomaly Detection*. [Online; accessed October-2021]. 2019. URL: https://towardsdatascience.com/lstm-autoencoder-for-anomaly-detection-e1f4f2ee7ccf.

[22] Alexander Lavin and Subutai Ahmad. "Evaluating Real-Time Anomaly Detection Algorithms – The Numenta Anomaly Benchmark". In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. 2015, pp. 38–44. DOI: 10.1109/ICMLA.2015.141.

[23] Qin Liu et al. "Real-Time FPGA-Based Hardware Neural Network for Fault Detection and Isolation in More Electric Aircraft". In: *IEEE Access* 7 (2019), pp. 159831–159841. DOI: 10.1109/ACCESS.2019.2950918.

[24] Mikroe. *MPU IMU Click*. [Online; accessed January-2022]. 2022. URL: https://www.mikroe.com/mpu-imu-click.

[25] Numenta. *The Numenta Anomaly Benchmark (NAB)*. [Online; accessed October-2021]. 2021. URL: https://github.com/numenta/NAB.

[26] Kalin Ovtcharov et al. *Accelerating Deep Convolutional Neural Networks Using Specialized Hardware*. [Online; accessed January-2022]. 2015. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/CNN20Whitepaper.pdf.

[27] pavithrasv. *Timeseries anomaly detection using an Autoencoder*. [Online; accessed November-2021]. 2020. URL: https://keras.io/examples/timeseries/timeseries_anomaly_detection/.

[28] Alfredo Perez-Castillo et al. "Real Time Monitoring of 3 Axis Accelerometer using an FPGA Zynq - 7000 and Embedded Linux through Ethernet". In: Sept. 2018. DOI: 10.1109/ICEEE.2018.8533999.

[29] Richa Rao. *Implementation of Long Short-Term Memory Neural Networks in High-Level Synthesis Targeting FPGAs*. [Online; accessed September-2021]. 2020. URL: https://cds.cern.ch/record/2729154/files/CERN-THESIS-2020-103.pdf.

[30] ReNom. *LSTM for Anomaly Detection in Time Series Data*. [Online; accessed November-2021]. 2021. URL: https://www.renom.jp/notebooks/tutorial/time_series/lstm-anomalydetection/notebook.html.

[31] Alaa Sagheer and Mostafa Kotb. *Unsupervised Pre-training of a Deep LSTM-based Stacked Autoencoder for Multivariate Time Series Forecasting Problems*. [Online; accessed October-2021]. 2019. URL: https://doi.org/10.1038/s41598-019-55320-6.

[32] Saikat Kumar Shome, Uma Datta, and S.R.K. Vadali. "FPGA based Signal Prefiltering System for Vibration Analysis of Induction Motor Failure Detection". In: *Procedia Technology* 4 (2012). 2nd International Conference on Computer, Communication, Control and Information Technology(C3IT-2012) on February 25 - 26, 2012, pp. 442–448. ISSN: 2212-0173. DOI: https://doi.org/10.1016/j.protcy.2012.05.070. URL: https://www.sciencedirect.com/science/article/pii/S2212017312003490.

[33] Akash Singh. *Anomaly Detection for Temporal Data using Long Short-Term Memory (LSTM)*. [Online; accessed October-2021]. 2017. URL: https://www.diva-portal.org/smash/get/diva2:1149130/FULLTEXT01.pdf.

[34] Rajbir Singh. *Difference between I2C and SPI (I2C VS SPI)*. [Online; accessed January-2022]. 2020. URL: https://medium.com/@rjrajbir24/difference-between-i2c-and-spi-i2c-vs-spi-c6a68d7242c4.

[35] Stephenm. *Creating a Linux user application in Vitis on a Zynq UltraScale Device*. [Online; accessed December-2021]. 2021. URL: https://support.xilinx.com/s/article/1141772?language=en_US.

[36] Tensorflow. *GPU support | Tensorflow*. [Online; accessed October-2021]. 2021. URL: https://www.tensorflow.org/install/gpu.

[37] Venkatesh Wadawadagi. *Accelerating Inference: Neural Network Pruning Explained*. [Online; accessed October-2021]. 2021. URL: https://blog.paperspace.com/neural-network-pruning-explained/.

[38] Confluence Wiki. *How to format SD Card for SD Boot*. [Online; accessed December-2021]. 2021. URL: https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842385/How+to+format+SD+card+for+SD+boot.

[39] Wikipedia. *Field-programmable gate array*. [Online; accessed January-2022]. 2022. URL: https://en.wikipedia.org/wiki/Field-programmable_gate_array.

[40] Wikipedia. *Serial Peripheral Interface*. [Online; accessed January-2022]. 2021. URL: https://fr.wikipedia.org/wiki/Serial_Peripheral_Interface.

[41] Warren Wu. *What is Predictive Maintenance? (+Benefits, Cost, & Examples)*. [Online; accessed February-2022]. 2020. URL: https://coastapp.com/blog/predictive-maintenance/.

[42] Xilinx. *AXI DMA v7.1 - LogiCORE IP Product Guide*. [Online; accessed January-2022]. 2019. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf.

[43] Xilinx. *AXI GPIO v2.0 - LogiCORE IP Product Guide*. [Online; accessed January-2022]. 2016. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_gpio/v2_0/pg144-axi-gpio.pdf.

[44] Xilinx. *AXI Quad SPI v3.2 - LogiCORE IP Product Guide*. [Online; accessed January-2022]. 2021. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_quad_spi/v3_2/pg153-axi-quad-spi.pdf.

[45] Xilinx. *PetaLinux Tools Documentation - Reference Guide*. [Online; accessed December-2021]. 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1144-petalinux-tools-reference-guide.pdf.

[46] Xilinx. *Spidev Test*. [Online; accessed January-2022]. 2020. URL: https://github.com/Xilinx/linux-xlnx/blob/master/tools/spi/spidev_test.c.

[47] Xilinx. *Verifying your Vivado HLS Design*. [Online; accessed November-2021]. 2021. URL: https://www.xilinx.com/video/hardware/verifying-your-vivado-hls-design.html.

[48] Xilinx. *Vitis Unified Software Platform Documentation*. [Online; accessed December-2021]. 2020. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1393-vitis-application-acceleration.pdf.

[49] Xilinx. *Vivado Design Suite User Guide - High-Level Synthesis*. [Online; accessed October-2021]. 2019. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf.

[50] Xilinx. *What is an FPGA?* [Online; accessed November-2021]. 2021. URL: https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

[51] Xilinx. *Xilinx Embedded Linux Build flows: PetaLinux Tools*. [Online; accessed December-2021]. 2020. URL: https://www.xilinx.com/video/hardware/xilinx-embedded-linux-build-flows-petalinux-tools.html.

[52] Xilinx. *ZCU104 Evaluation Board - User Guide*. [Online; accessed December-2021]. 2018. URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[53] Xilinx. *Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit*. [Online; accessed October-2021]. 2021. URL: https://www.xilinx.com/products/boards-and-kits/zcu104.html.

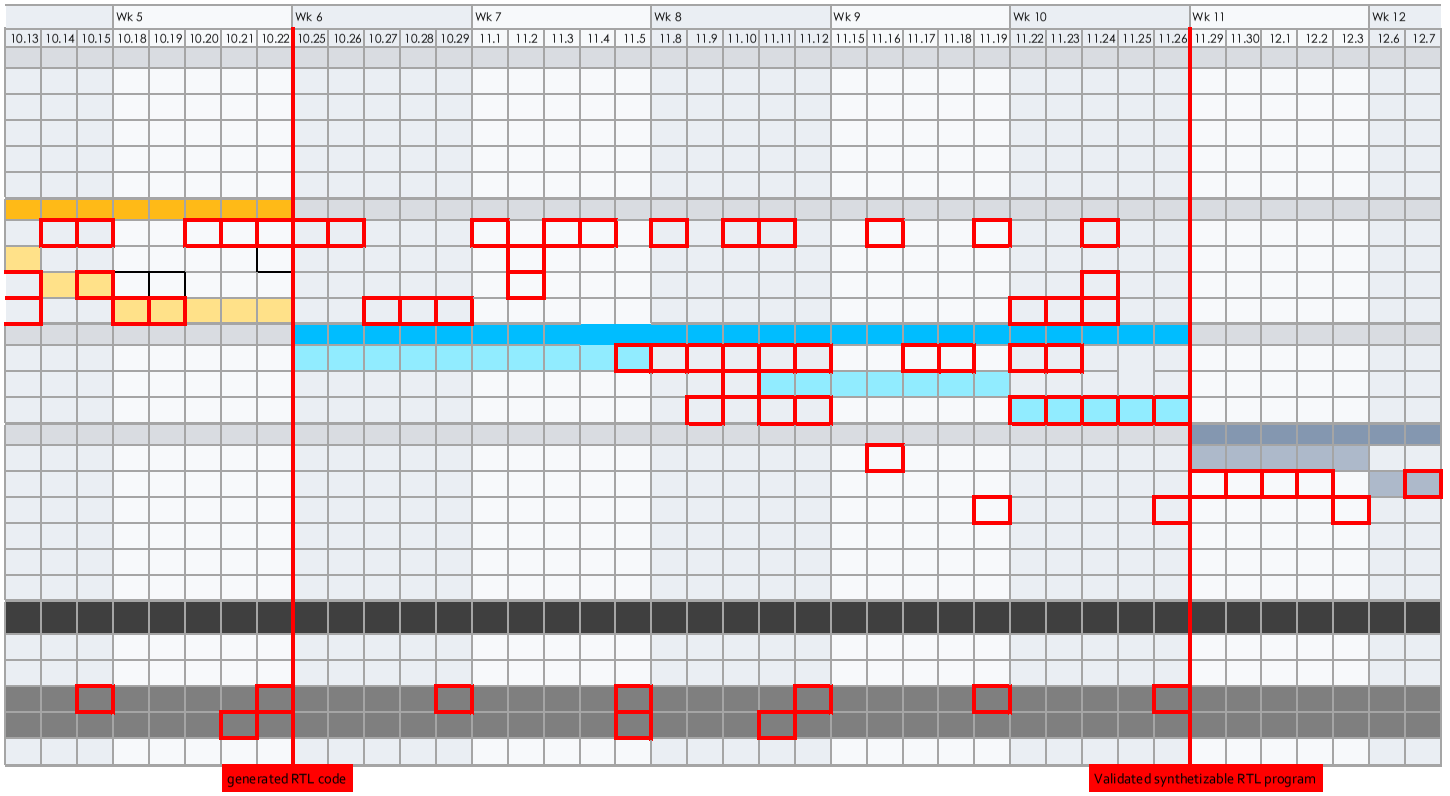
Master Thesis - Model-Based Predictive Maintenance on FPGA

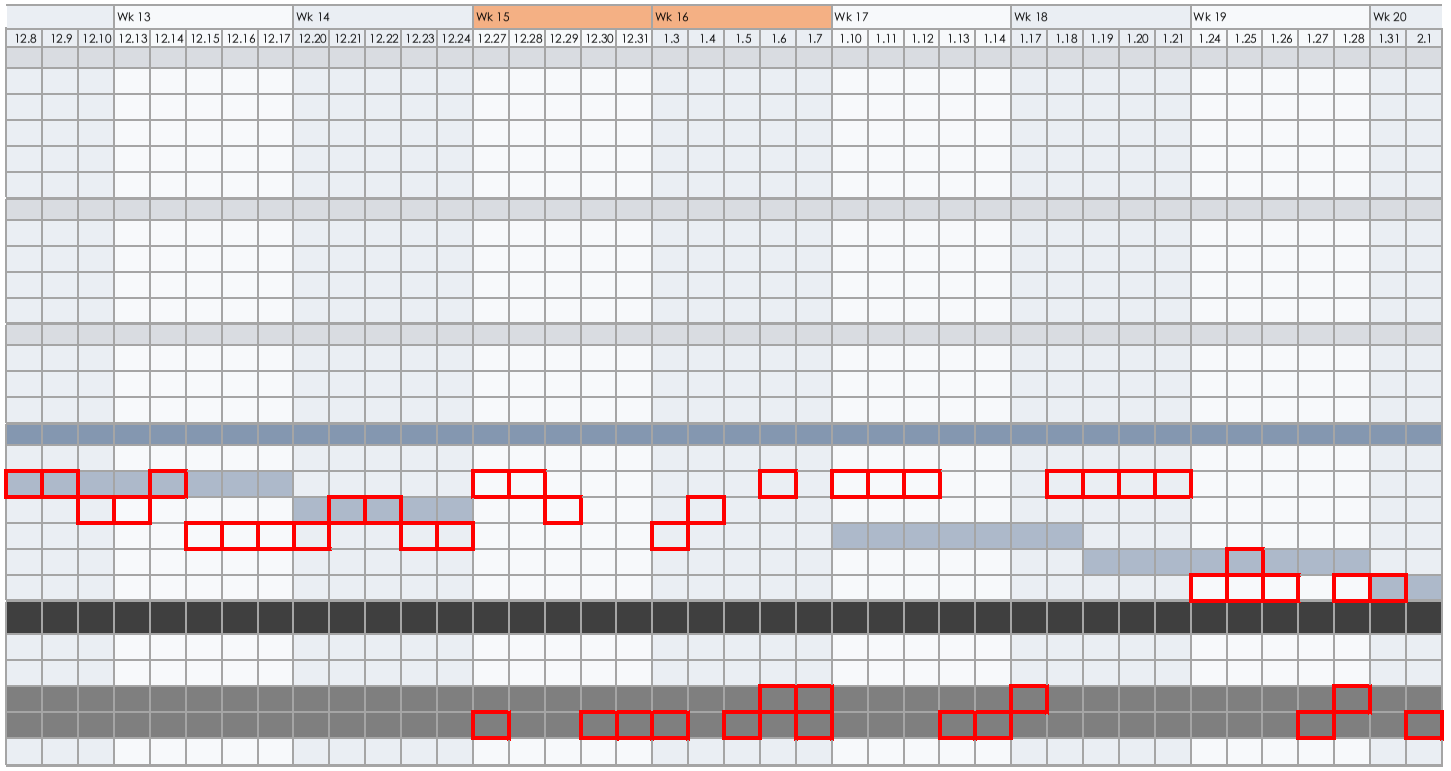
Student	START DATE
Sami Föry	lundi, septembre 20, 2021

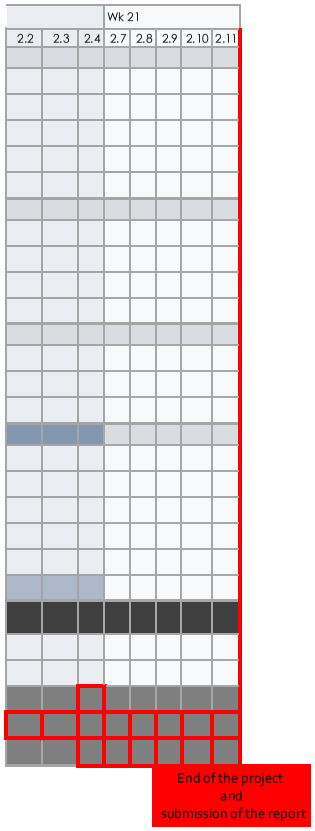
					Wk 1					Wk 2					Wk 3					Wk 4	
ACTIVITY	**STATUS**	**START DATE**	**END DATE**	**# of Days**	9.20	9.21	9.22	9.23	9.24	9.27	9.28	9.29	9.30	10.1	10.4	10.5	10.6	10.7	10.8	10.11	10.12
PHASE 1: **State-of-the-Art**		**09.20.21**	**10.01.21**	**10**																	
Predictive maintenance of vibratory defects using FPGA	Complete	09.20.21	09.22.21	3																	
Deep learning RNN-LSTM network	Complete	09.22.21	09.24.21	3																	
Deep learning LSTM network implementation on FPGA	Complete	09.21.21	09.28.21	6																	
Hardware-in-the-loop simulation	Complete	09.27.21	09.29.21	3																	
FPGA hardware integration and real-time processing	Complete	09.28.21	10.01.21	4																	
PHASE 2: **Modeling and Conversion**		**10.04.21**	**10.22.21**	**15**																	
Python Keras LSTM model development	Complete	10.04.21	10.08.21	5																	
Optimization of the Python LSTM model (pruning+quantization)	Complete	10.07.21	10.13.21	5																	
HLS4ML adaptation and HLS/RTL generation of the model	Complete	10.14.21	10.15.21	2																	
Correction and verification of the HLS/RTL generated code	Complete	10.18.21	10.22.21	5																	
PHASE 3: **Verification & Validation**		**10.25.21**	**11.26.21**	**25**																	
Python level verification & validation	Complete	10.25.21	11.05.21	10																	
C/RTL level verification verification & validation	Complete	11.11.21	11.19.21	7																	
Correction of the model	Complete	11.22.21	11.26.21	5																	
PHASE 4: **Hardware Implementation**		**11.29.21**	**02.04.22**	**50**																	
Appropriated choice and order of the FPGA board	Complete	11.29.21	12.03.21	5																	
Hardware configuration (I/O, bus communication,...)	Complete	12.06.21	12.17.21	10																	
Definition and design of the host application (CPU/ARM)	Complete	12.20.21	12.24.21	5																	
Boot linux OS on processing system	Complete	01.10.22	01.18.22	7																	
Mounting test bench	Complete	01.19.22	01.28.22	8																	
Physical real-time testing and validation	Complete	01.31.22	02.04.22	5																	
PHASE 5: **Thesis Administration**		**09.20.21**	**02.11.22**	**105**																	
Functional/non-functional requirements writing	Complete	09.20.21	09.27.21	6																	
Scope statement	Complete	09.20.21	09.24.21	5																	
Weekly summary	Complete	09.20.21	02.11.22	105																	
Report writing	Complete	09.20.21	02.11.22	105																	
Report correction	Complete	02.02.22	02.11.22	8																	

End of the state-of-the-art

Legends: Holidays, Milestones, Completed


Wk 5
Wk 6
Wk 7
Wk 8
Wk 9
Wk 10
Wk 11
Wk 12
10.13 10.14 10.15 10.18 10.19 10.20 10.21 10.22 10.25 10.26 10.27 10.28 10.29 11.1 11.2 11.3 11.4 11.5 11.8 11.9 11.10 11.11 11.12 11.15 11.16 11.17 11.18 11.19 11.22 11.23 11.24 11.25 11.26 11.29 11.30 12.1 12.2 12.3 12.6 12.7
generated RTL code
Validated synthetizable RTL program


Wk 13
Wk 14
Wk 15
Wk 16
Wk 17
Wk 18
Wk 19
Wk 20
12.8 12.9 12.10 12.13 12.14 12.15 12.16 12.17 12.20 12.21 12.22 12.23 12.24 12.27 12.28 12.29 12.30 12.31 1.3 1.4 1.5 1.6 1.7 1.10 1.11 1.12 1.13 1.14 1.17 1.18 1.19 1.20 1.21 1.24 1.25 1.26 1.27 1.28 1.31 2.1


Wk 21
2.2
2.3
2.4
2.7
2.8
2.9
2.10
2.11
End of the project
and
submission of the report

B | INITIAL SCOPE

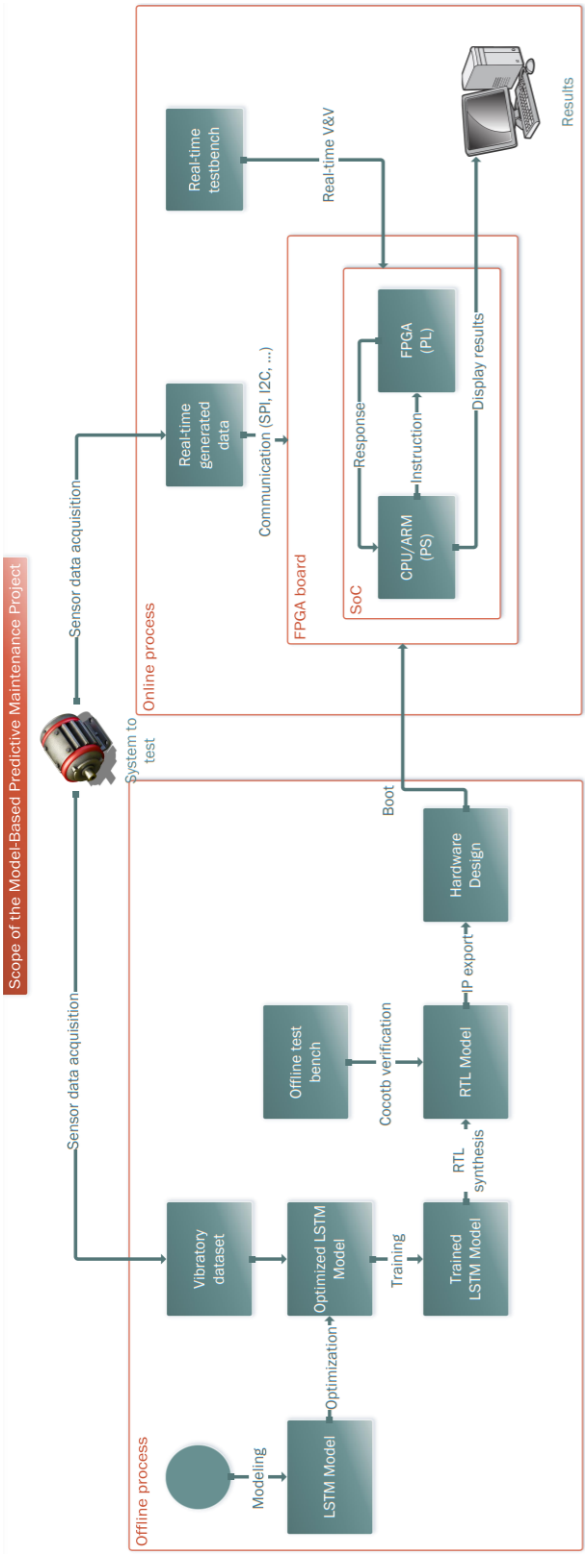


Figure B.1 Initial scope representation

C | FINAL SCOPE

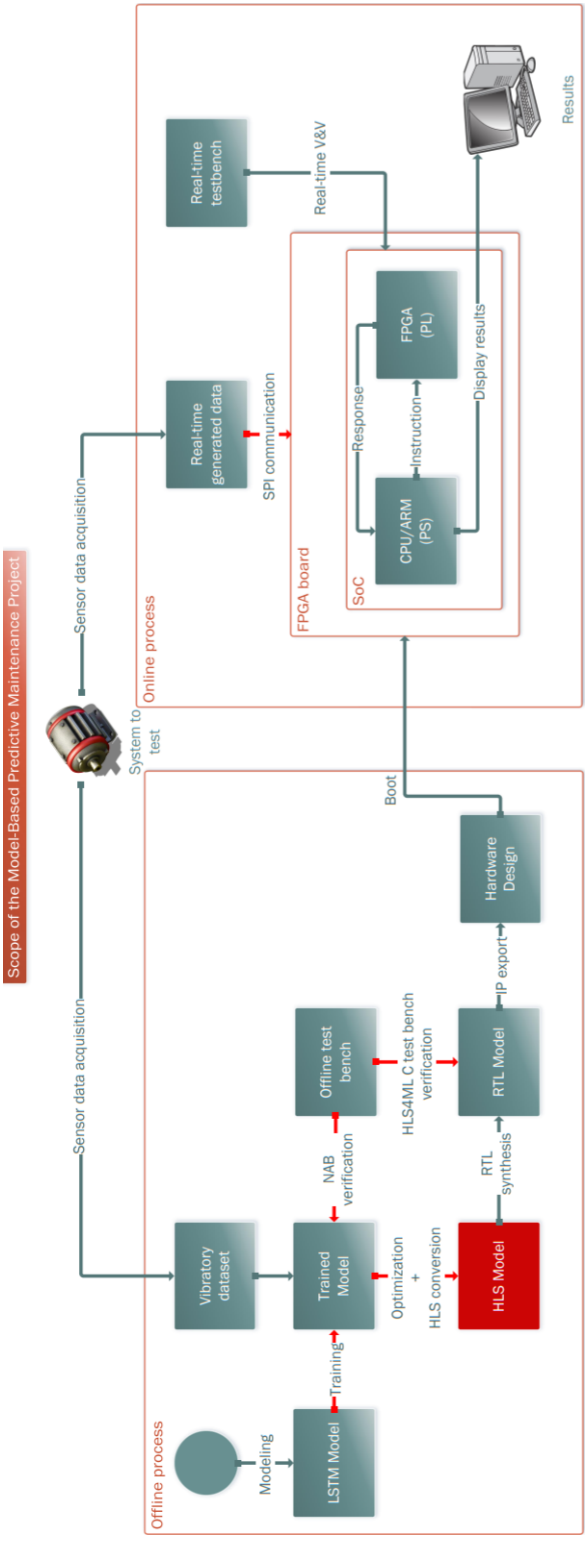


Figure C.1 Final scope representation

D | TUTORIAL

After having provided the explanations concerning the various stages of the project, it is good to devote a document detailing the development process of a custom anomaly detection tool in order to generalize the concept for the use of new source data. To do this, we will go through the different steps of realization in the form of a tutorial to highlight the commands, scripts and methods to use.

D.1 Repository structure

```
conversion
├── hls4ml
data
├── spi_train_tb
hardware
├── wrapper.xsa
modeling
├── ARM_software_infos
├── dataset
├── logs
├── model
├── NAB_files
├── tb_data
├── trained_models_store
├── __init__.py
├── data_structure.py
├── detector_analysis.py
├── gpu_detect.py
├── model_manipulation.py
├── network_config.ini
├── prediction.py
├── save_data_tb.py
├── statistics.py
├── train_analysis.py
├── training.py
results
├── <spi_test_cases>
software
├── main.c
testing
├── spidev-test
├── tb_data
├── NAB
├── plot_results.py
```

The repository is available in GitHub.

D.2 Modeling

The first step in this project is the modeling phase. To do this, this entire step takes place in the *modeling* directory. These describe how to train and test an LSTM model based on a NAB dataset.

1. Choose a NAB dataset to place in the directory dataset > nab_data.

   It is possible to do this with any dataset. If it is not a dataset from the NAB directory, it will simply not be possible to test the model with the NAB method. In this case, it is not necessary to go through the NAB based V&V phase.

2. Customize the parameters of the file *network_config.ini*.

   DATA

   The initial configurations are related to the above hierarchical structure. To begin with, the main modifications to be made concern the *TRAIN_RATE* and the *THRESHOLD*. The train rate is the proportion of the data to be used for training. The rest is automatically dedicated to tests. The threshold to be indicated is the value resulting from the analysis of the normal distribution on the reconstruction errors of the training data. This value is to be entered once the *train_analysis.py* has been performed.

   NETWORK

   These configurations concern the tuning of the LSTM network. In case of modification of the number of layers of the network, this is done directly in the code of the file *training.py*. It is enough to add/remove the lines which concern the layer(s).

   MODEL

   These configurations concern the path and the name of the model generated at the end of the training.

3. Structure the data and visualize if it is ready to be trained.

   ```
   python data_structure.py
   ```

   A plot for the training data and another for the test data can be viewed.

4. Train the model based on the data and configuration provided.

   ```
   python training.py
   ```

   The training may take some time depending on the amount of data to be trained and the model configuration. At the end of the training, a file *model.h5* and another *model.json* are automatically saved in the folder *model*. Moreover, a plot of the cost function to be minimized can be visualized. To validate the training, it is necessary to use the curve *validation* which must be close to 0.

5. Train step analysis

```
python train_analysis.py
```

   The training data are reconstructed. A plot of the reconstruction appears, followed by a plot of the normal distribution. It is important to analyze the latter in order to retrieve the value of *threshold* to add in the *network_config.ini* file under *DATA/THRESHOLD*. The rightmost value of the distribution is a good threshold reference.

6. Test the trained model on test data

```
python prediction.py
```

   The model is loaded to reconstruct the test data. The results are then displayed.

```
python statistics.py
```

   The results obtained are analyzed with the mahalanobis distance. The prediction module is also launched when this program is executed.

D.3 Python level verification

The second step concerns the verification phase at the Python level with the use of the open source solution NAB (Numenta Anomaly Benchmark).

1. Change the parameters *DATA_GEN_NAB_DIR*, *NAB_INPUTS* and *NAB_RESULTS* in the *network_config.ini* file.

   The first parameter indicates the location where the NAB files will be generated. The next two parameters indicate the name of the files. It is necessary to keep the same structure as the example E, i.e. *detector-name_nab-data-file-name*.

2. Run the detector on the NAB data and save the results in NAB format in the path indicated in *DATA_GEN_NAB_DIR*.

```
python detector_analysis.py
```

3. Go to the NAB directory, in testing » NAB and create the directories necessary for the detector to analyze.

```
python scripts/create_new_detector.py --detector LSTM28
```

   LSTM28 is the name of the detector, which can be customized.

4. Retrieve the result file and place it in testing » NAB » results, in the right category (the same directory where the NAB data was retrieved in *data*)

5. Open the file *thresholds.json* in testing » NAB » config and add the following lines in the newly created detector

```
        "reward_low_FN_rate": {
        "threshold": 0.5
        },
        "reward_low_FP_rate": {
            "threshold": 0.5
        },
        "standard": {
            "threshold": 0.5
```

6. Run the NAB scoring

```
python run.py -d LSTM28 --score --normalize
```

The results are listed in testing > NAB > results > final_results.json

For detailed information, the NAB directory is available at `https://github.com/numenta/NAB`. The process must be repeated for each new set to be tested.

D.4 HLS and RTL conversions

Once the model is designed and validated, it is possible to go through the conversion steps, up to the RTL level.

1. Create a directory *workspace* in conversion > hls4ml
2. Create a directory *keras* in the directory *workspace*, in conversion > hls4ml > workspace
3. Retrieve the files *.json* and *.h5* generated previously in modeling > model and place them in the directory *keras*, in conversion > hls4ml > workspace > keras
4. Run the save data file in the *modeling* directory

```
python save_data.py
```

and retrieve the files *input_data.dat* and *reconstruction.dat* in modeling > tb_data and place them in conversion > hls4ml > workspace > keras

5. Create a file *keras-config.yml* in conversion > hls4ml > workspace > keras and copy the following configuration.

```
KerasJson: keras/model.json
KerasH5:   keras/model.h5
InputData: keras/input_data.dat
OutputPredictions: keras/reconstruction.dat
OutputDir: kerasHLS
ProjectName: kerasHLS
XilinxPart: xczu7ev-ffvc1156-2-e
ClockPeriod: 5
Backend: Vivado
```

```
10
11 MaxLoop: 20
12
13 #options: io_serial/io_parallel
14 IOType: io_parallel
15 HLSConfig:
16   Model:
17     Precision: ap_fixed<16,8>
18     ReuseFactor: 1
19     #Strategy: Resource
20     #Compression: True
21 #  LayerType:
22 #    Dense:
23 #      ReuseFactor: 2
24 #      Strategy: Resource
25 #      Compression: True
26 #    LSTM:
27 #      ReuseFactor: 2
28 #      Strategy: Resource
29 #      Compression: True
```

The HLS4ML configuration can be customized according to the need in terms of performance, resources, etc. More details are available in the conversion chapter of the report.

6. Open the file *hls4ml*, in conversion > hls4ml > scripts and add the following line, just below *import sys*.

```
sys.path.append('<path-to-hls4ml-repo>')
```

7. Open a terminal in conversion > hls4ml > workspace and run the following command.

```
python ../scripts/hls4ml convert -c keras-config.yml
```

A new directory will be created with the name defined in the file *keras-config.yml*.

NOTE: If a problem occurs, check that the official hls4ml library is not installed. If it is the case, uninstall hls4ml.

8. Enter the new directory with the terminal and run the following command.

```
vivado_hls -f bluid_prj.tcl "csim=1 synth=1 cosim=1 export
=0"
```

NOTE: In order to use the vivado_hls command, it must be sourced each time the terminal is started. Open the *.bashrc* with

```
vim .bashrc
```

and add the following line before saving the file

```
source ~/<path-to-vivado_hls-installation>/2019.2/settings64
.sh
```

Reload the terminal with the following command.

```
source .bashrc
```

NOTE: The conversion process may take some time depending on the size of the network. Once the conversion is completed, it is possible to view the logs to verify if the HLS4ML test bench has validated the results of the C/RTL cosimulation.

9. Open Vivado HLS

```
vivado_hls -p kerasHLS_prj
```

10. Check performance and resource estimates and others as needed

11. Run IP export by clicking on the icon below.

Figure D.1 Vivado HLS export RTL icone

12. Configure the export according to the following window or in a customized way.

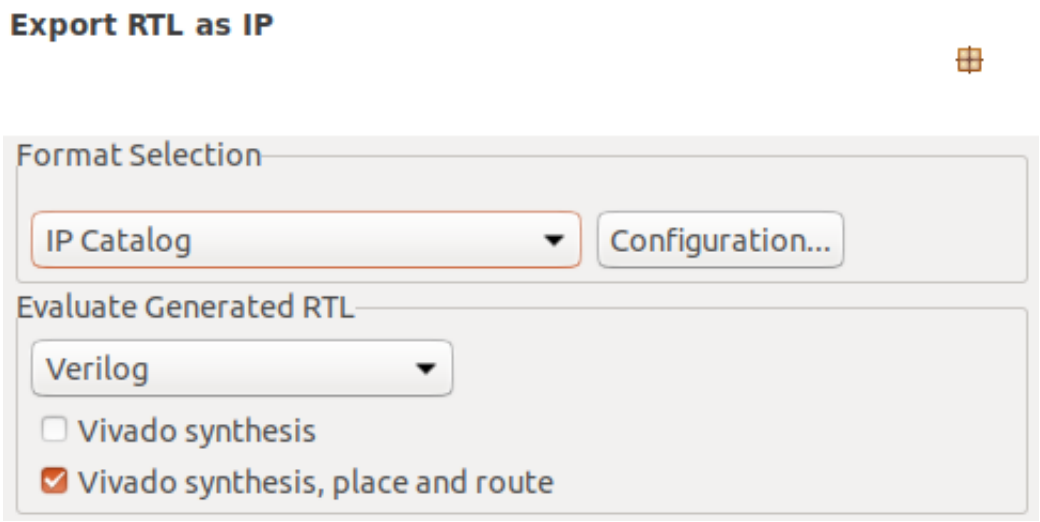


Figure D.2 Vivado HLS export RTL window

NOTE: This export step can also be time consuming.

D.5 C/RTL level verification

The verification and validation step of the model at the C and RTL level is done at the same time as the previous step. The first step before the conversion is the simulation at C level. This already makes it possible to obtain the results in output of the C model in conversion > hls4ml > workspace > kerasHLS > tb_data. It is then already possible to plot these results thanks to the Python script dedicated to this, which is in testing > plot_results.py

```
    python plot_results.py
```

NOTE: It is important to check and modify the paths indicated in the first function of the code and to drag the correct files into testing > tb_data. Except the first function, each call of the other functions is independent. It is therefore preferable to comment out the calls that are not used. This program has not been adapted to be user friendly.

With the results of the C-level simulation, it is possible to visually compare the results at the python and C level. The average error can also be calculated. To do this, use the first two function calls (*loadFiles* and *conversionAnalysis*) and put the rest in comments. At the end of the C/RTL cosimulation, the test bench written in C and provided by HLS4ML will test if the results at the C and RTL levels are identical. If this is the case, an indication of the success of the test is displayed in the terminal.

```
+---------+---------+-------------------------------------------+-------------------------------------------+
|         |         |                  Latency                  |                  Interval                 |
+   RTL   +  Status +-------------------------------------------+-------------------------------------------+
|         |         |      min      |      avg      |      max      |      min      |      avg      |      max      |
+---------+---------+-------------------------------------------+-------------------------------------------+
|     VHDL|       NA|             NA|             NA|             NA|             NA|             NA|             NA|
|  Verilog|     Pass|             22|             22|             22|              1|              1|              1|
+---------+---------+-------------------------------------------+-------------------------------------------+

***** C/RTL SIMULATION COMPLETED IN 0h1m27s *****
***** C/RTL VALIDATION *****
INFO: Test PASSED
```

Figure D.3 C/RTL test passed example

Once this is done, if the conversion to the C level is satisfactory compared to the results of the Python level and the test bench displays a successful test at the end of the C/RTL cosimulation, it is possible to validate the model and go to the next step.

D.6 Hardware implementation

The last step is the hardware implementation of the developed model. To do this, we must go through 4 steps, which are

1. The hardware block design in Vivado 2019.2
2. The creation of the Petalinux project
3. The creation of the project in Vitis with the construction of the files to be slipped into the SD card
4. The boot of the system in the target

To do this, follow the steps below and, if necessary, read the "Hardware Implementation" chapter for more details.

D.6.1 Hardware block design

1. Create a new RTL project Vivado

   Project Name

   No constraints, the user can change these parameters as he wishes.

   Project Type

   Let RTL Project select, without making any other changes.

   Add Sources

   Change *Target language* if necessary, otherwise no change.

   Add Constraints

   Click *Create File* and create an XDC file with a custom name. This file will allow to list the hardware constraints according to the components used on the board.

   Default Part

   Click on the tab *Boards* and select the board used.

   New Project Summary

   Click Finish to create the Vivado project.

2. Add the previously exported IP to the Vivado IP catalog

   Under *PROJECT MANAGER*, click on *IP catalog*

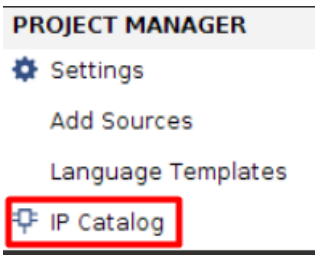


Figure D.4 IP catalog icon

Then, Right click > Add Repository and select the following path conversion > hls4ml > workspace > kerasHLS > kerasHLS_prj > solution1

After this, the IP has been added to the IP catalog and can be used in the block design.

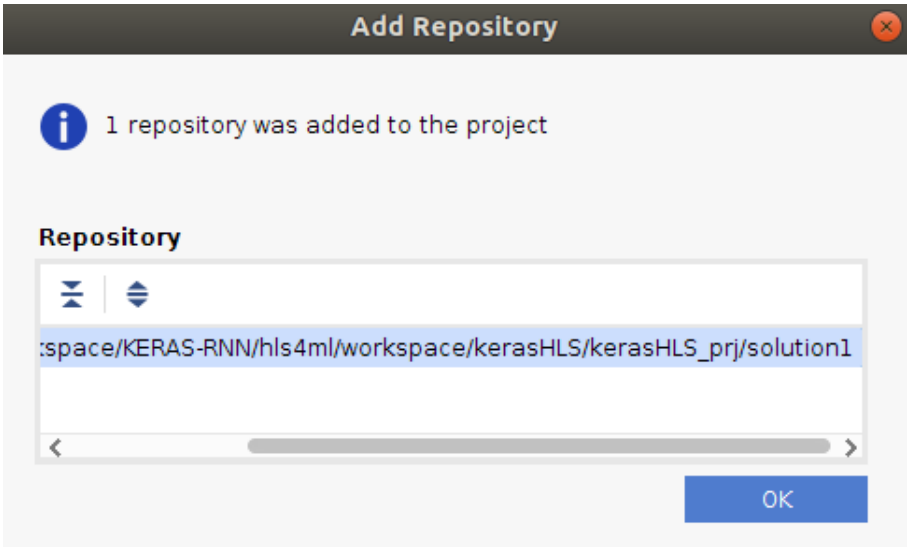


Figure D.5 IP Added in IP catalog

3. Create the block design

   Click on *Create Block Design* and add the IPs by clicking on the + (Add IP). The design made for this project is available in the appendix R. For the main application to work, it is necessary to reproduce this design identically. There is only the IP HLS block representing the model which can vary in its internal functioning.

   At the block configuration level, there is a change to be made on the PS block (Zynq UltraScale+ MPSoC) by adding a second clock *pl_clk1* with the following definition

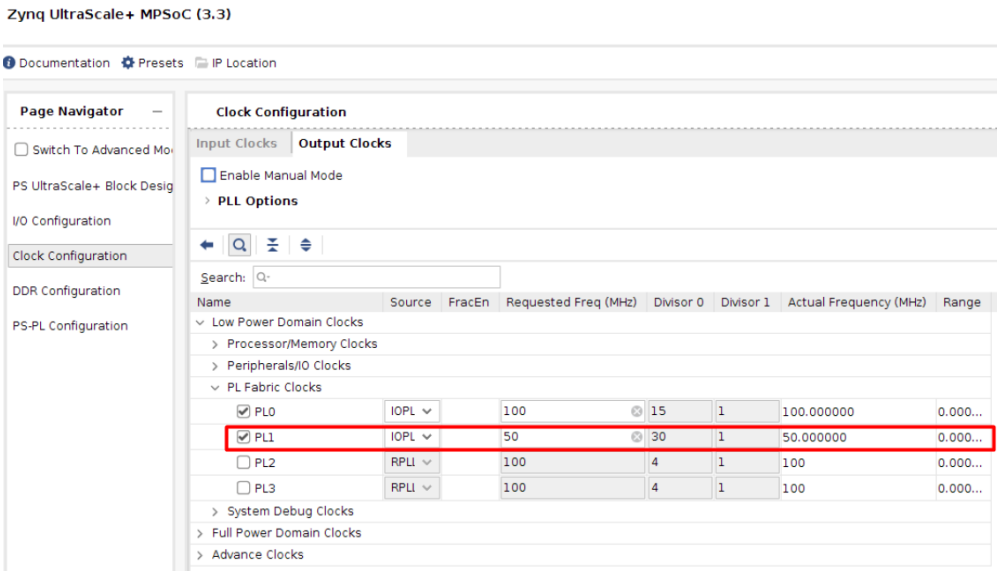


Figure D.6 PS block pl_clk1 configuration

A second parameterization must be done on the AXI GPIOs block. It is necessary to define the width of bits for each of them as well as whether they are inputs or outputs. For example, AXI GPIO 3, connected to the output of the IP HLS block, which generates results on 16 bits wide, must have the following configuration.

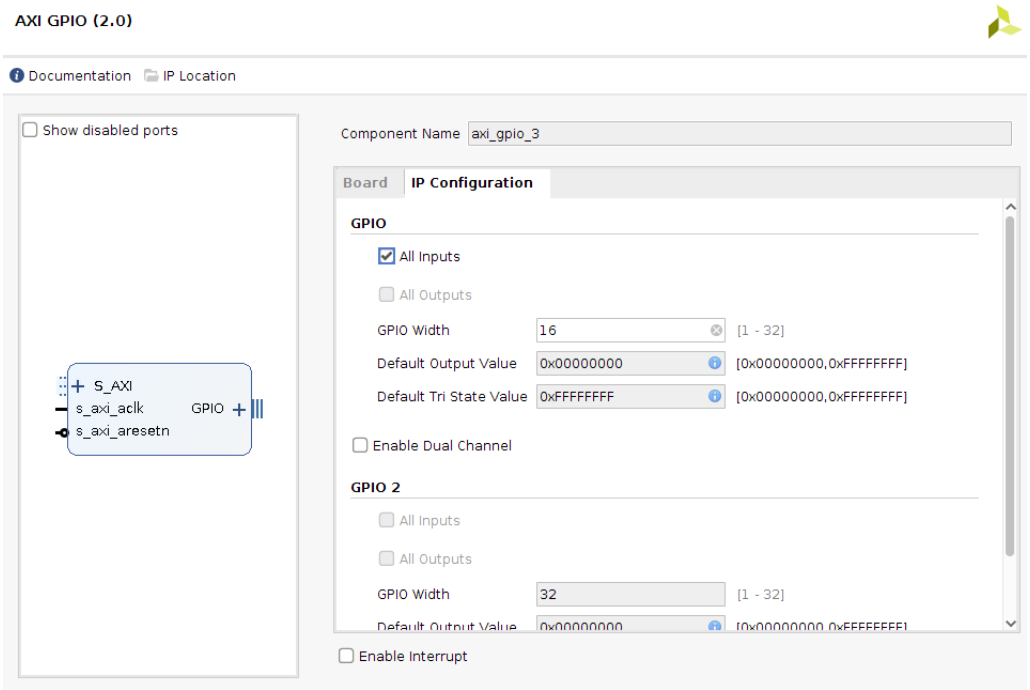


Figure D.7 AXI GPIO configuration example

It is configured as input, because it will receive as input the output value of the IP HLS block.

For the rest, there are no changes to be made.

4. Modify the previously generated XDC file, which is located under the tab Sources > Constraints > constrs_1 with the following elements.

```
    # Clock constraints
    create_clock -name ap_clk -period 5.000 -waveform {0.000
2.500} [get_ports ap_clk]

    # SPI

    #MOSI
    set_property PACKAGE_PIN G8       [get_ports "spi_io0_io"] ;
# Bank  87 VCCO - VCC3V3   - IO_L12N_AD0N_87
    set_property IOSTANDARD  LVCMOS33 [get_ports "spi_io0_io"] ;
# Bank  87 VCCO - VCC3V3   - IO_L12N_AD0N_87

    #MISO
    set_property PACKAGE_PIN H8       [get_ports "spi_io1_io"] ;
# Bank  87 VCCO - VCC3V3   - IO_L12P_AD0P_87
    set_property IOSTANDARD  LVCMOS33 [get_ports "spi_io1_io"] ;
# Bank  87 VCCO - VCC3V3   - IO_L12P_AD0P_87

    #SCK
    set_property PACKAGE_PIN G7       [get_ports "spi_sck_io"] ;
# Bank  87 VCCO - VCC3V3   - IO_L11N_AD1N_87
    set_property IOSTANDARD  LVCMOS33 [get_ports "spi_sck_io"] ;
# Bank  87 VCCO - VCC3V3   - IO_L11N_AD1N_87

    #SS
    set_property PACKAGE_PIN H7       [get_ports "spi_ss_io"] ;#
 Bank  87 VCCO - VCC3V3   - IO_L11P_AD1P_87
    set_property IOSTANDARD  LVCMOS33 [get_ports "spi_ss_io"] ;#
 Bank  87 VCCO - VCC3V3   - IO_L11P_AD1P_87

    # LEDs
    set_property PACKAGE_PIN D5       [get_ports "GPIO_LED_0_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L11N_AD9N_88
    set_property IOSTANDARD  LVCMOS33 [get_ports "GPIO_LED_0_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L11N_AD9N_88
    set_property PACKAGE_PIN D6       [get_ports "GPIO_LED_1_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L11P_AD9P_88
    set_property IOSTANDARD  LVCMOS33 [get_ports "GPIO_LED_1_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L11P_AD9P_88
    set_property PACKAGE_PIN A5       [get_ports "GPIO_LED_2_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L10N_AD10N_88
    set_property IOSTANDARD  LVCMOS33 [get_ports "GPIO_LED_2_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L10N_AD10N_88
    set_property PACKAGE_PIN B5       [get_ports "GPIO_LED_3_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L10P_AD10P_88
    set_property IOSTANDARD  LVCMOS33 [get_ports "GPIO_LED_3_LS"
] ;# Bank  88 VCCO - VCC3V3   - IO_L10P_AD10P_88
```

Then select this file with right click and click on *Set as Target Constraint File*

NOTE: It should be noted that each board has its own Master XDC file available in its documentation. The way of describing the hardware constraints are not the same and especially, the components and circuits change from one board to another. For example for the ZCU104, the file can be downloaded from the documentation in [53]

5. Create HDL Wrapper

   Right click on Sources > Design Sources > <design name> and choose *Create HDL Wrapper*

6. Validate Design

   Right click on the block design and choose *Validate Design*

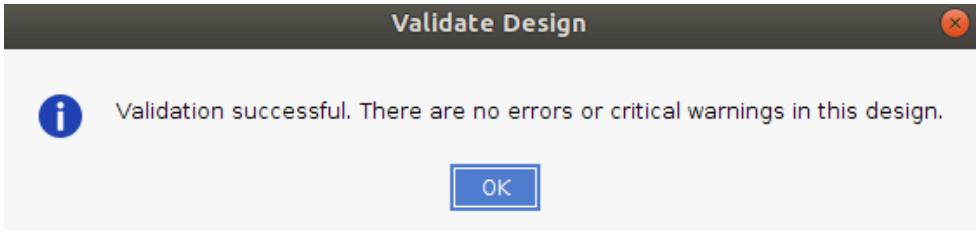


Figure D.8 Block design validation message

7. Export Hardware

   Export the hardware design by selecting File > Export > Export Hardware

D.6.2 Petalinux Project Creation

1. Template creation using BSP file

```
petalinux-create -t project -s <path-to-bsp>
```

   NOTE: The BSP file of the used board is available on the **Xilinx downloads website.**

2. Import Hardware Design

```
petalinux-config --get-hw-description=<path-to-xsa>
```

   The system configuration appears and there are no changes to be made, so exit.

3. Linux Kernel Configuration

```
petalinux-config -c kernel
```

   After a few minutes, the configuration window opens and it is necessary to make 4 settings.

(a) Enable UIO framework

Device Drivers > Userspace I/O drivers > Userspace I/O platform driver with generic IRQ handling [*]
Device Drivers > Userspace I/O drivers > Userspace platform driver with generic irq and dynamic memory [*]

(b) Enable SPI userspace driver

Device Drivers > SPI support > Userspace mode SPI device driver support [*]

(c) Disable /dev/mem filter

Kernel hacking > Filter access to /dev/mem []

4. Replace the *system-user.dtsi* file in <root-lnx-proj> > project-spec > meta-user > recipes-bsp > device-tree > files with the following

```
/include/ "system-conf.dtsi"
/ {


    chosen {
        bootargs = "earlycon clk_ignore_unused
uio_pdrv_genirq.of_id=generic-uio";
        stdout-path = "serial0:115200n8";
    };
};

&axi_gpio_0 {
    compatible = "generic-uio";
};
&axi_gpio_1 {
    compatible = "generic-uio";
};
&axi_gpio_2 {
    compatible = "generic-uio";
};
&axi_gpio_3 {
    compatible = "generic-uio";
};

&axi_quad_spi_0 {
  spidev@0 {
    reg = <0>;
    compatible = "spidev";
    spi-max-frequency = <25000000>;
  };
};
```

5. Build petalinux project

```
petalinux-build
```

NOTE: This step can take time and can fail sometimes. This is normal and probably due to the network connection. In this case, you just need to rerun the command. Of course, the build process time will be shorter than the first time.

6. Go to <root-lnx-prof> » images » linux and build SDK

```
petalinux-build --sdk
```

NOTE: This can also take time.

7. Create system root

```
petalinux-package --sysroot
```

D.6.3 Create the project in Vitis IDE

1. Create boot and root repository in the <root-lnx-proj>

```
mkdir -p sw_comp/src/a53/xrt/image
mkdir sw_comp/src/boot
```

2. In <root-lnx-proj> » images » linux, copy the files

   - image.ub
   - boot.src
   - rootfs.cpio.gz

   In <root-lnx-proj> » sw_comp » src » a53 » xrt » image and copy the files

   - system.bit
   - bl31.elf
   - u-boot.elf
   - zynqmp_fsbl.elf
   - pmufw.elf

   In <root-lnx-proj> » sw_comp » src » boot

3. Also in <root-lnx-proj> » sw_comp » src » boot, create a *linux.bif* file with the following content

```
      the_ROM_image:
      {
      [fsbl_config] a53_x64
      [bootloader] <xilinx-zcu104-2019.2/boot/zynqmp_fsbl.elf>
      [pmufw_image] <xilinx-zcu104-2019.2/boot/pmufw.elf>
      [destination_device=pl] <system.bit>
      [destination_cpu=a53-0, exception_level=el-3, trustzone] <
xilinx-zcu104-2019.2/boot/bl31.elf>
      [destination_cpu=a53-0, exception_level=el-2] <xilinx-zcu104
-2019.2/boot/u-boot.elf>
      }
```

4. Open Vitis IDE and create a new platform project

   (a) Create new platform project

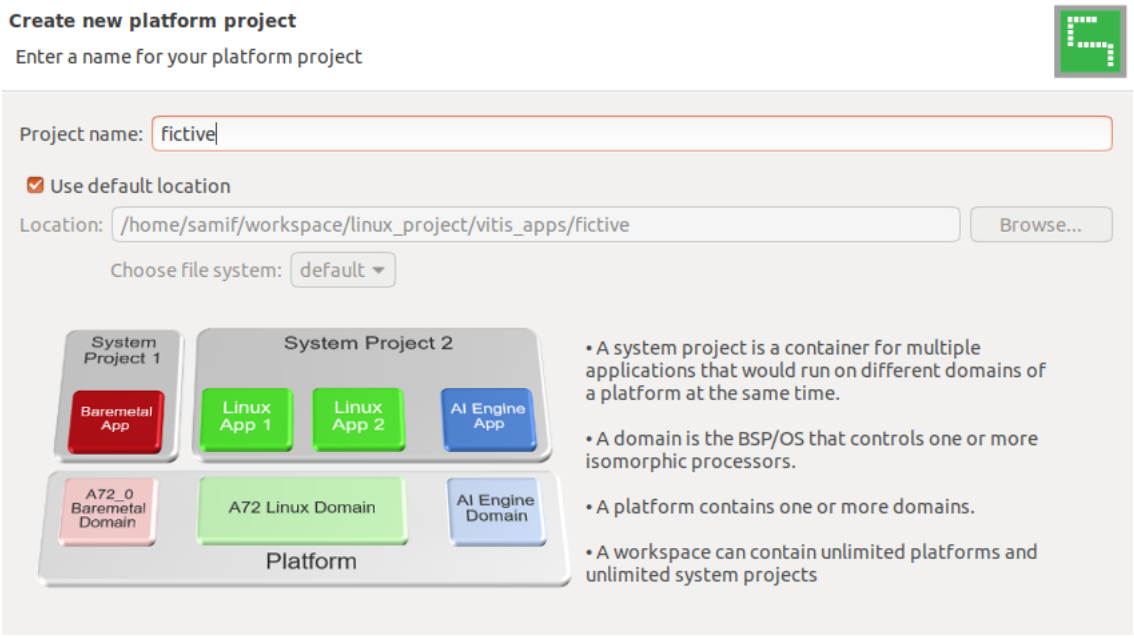


Figure D.9 Vitis new platform interface

   (b) Platform Project

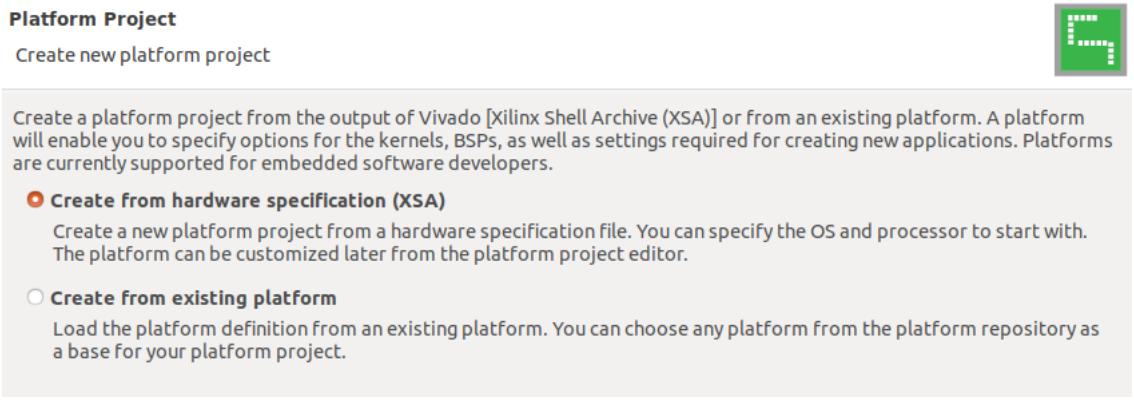


Figure D.10 *Vitis platform project interface*

(c) Platform Project Specification

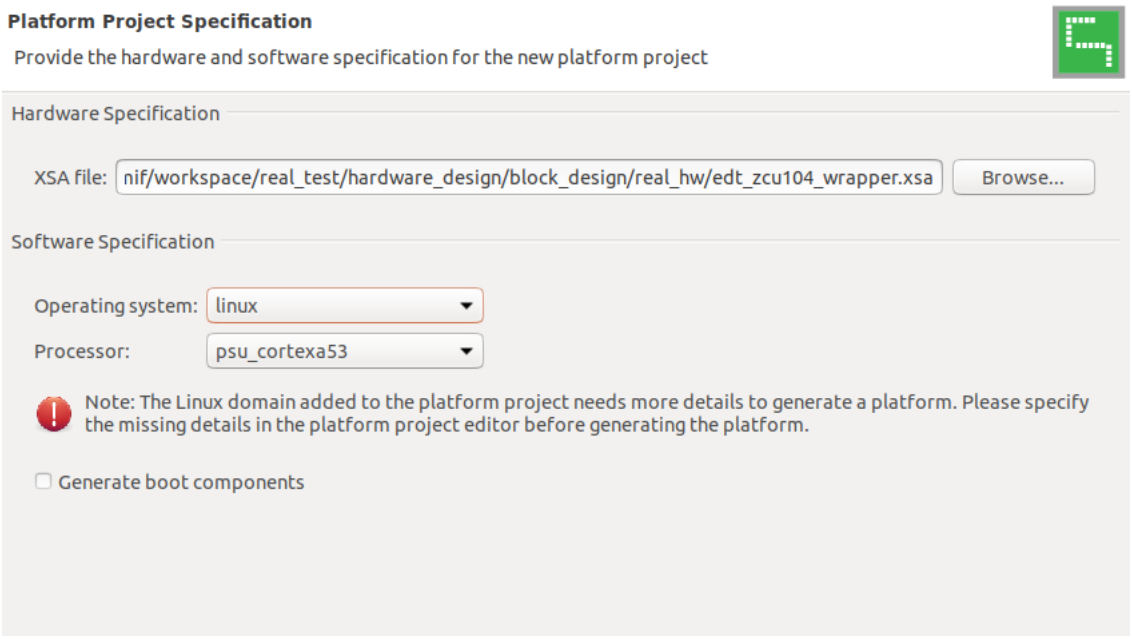


Figure D.11 *Vitis platform specification interface*

5. Locate Linux project specific files/folders

   (a) Insert paths for *zynqmp_fsbl.elf* and *pmufw.elf* files

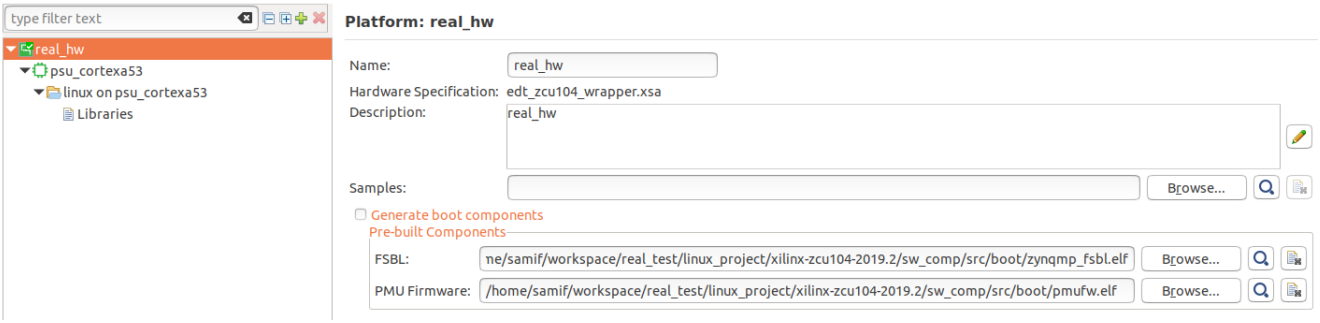


Figure D.12 Vitis Linux paths interface

(b) Insert paths for *linux.bif* file, *boot*, *image* and *sysroots/aarch64-xilinx-linux* folders.

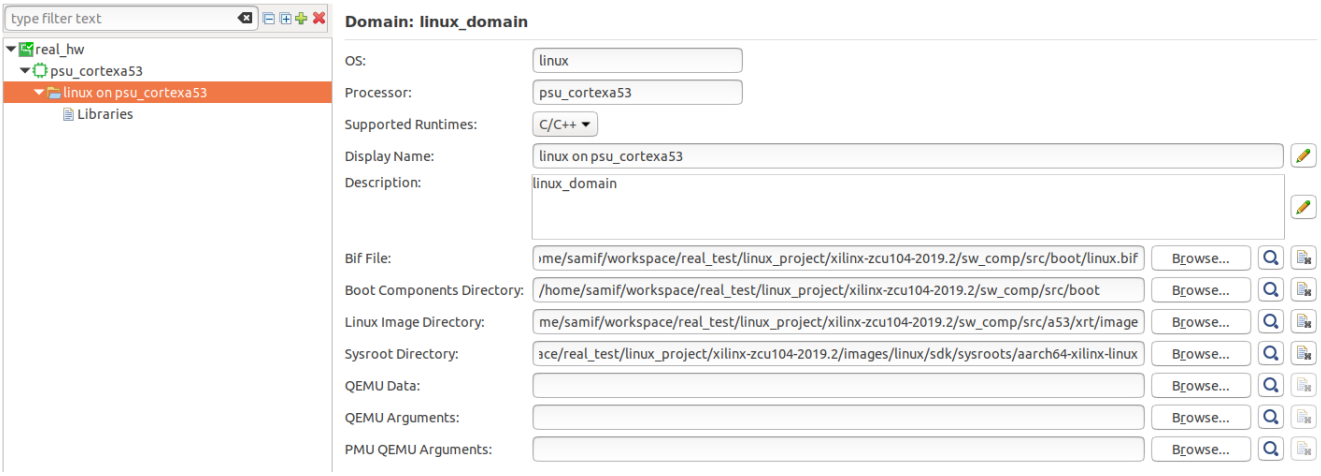


Figure D.13 Vitis Linux paths interface

6. Build the Vitis platform using the following icon

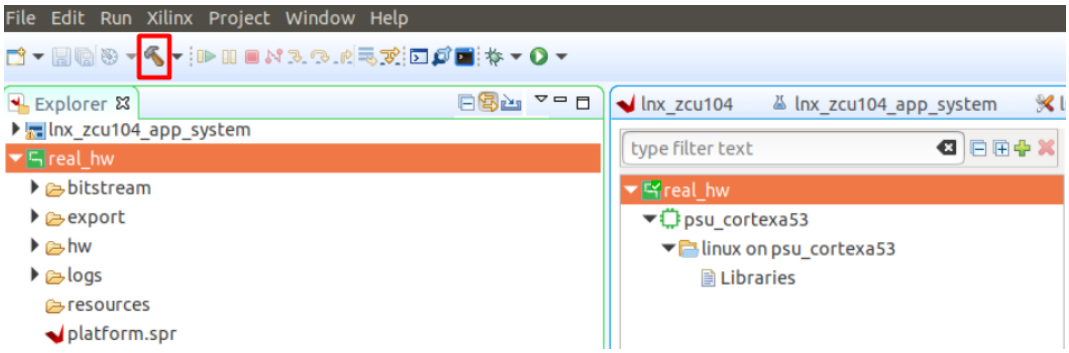


Figure D.14 Vitis build platform

7. Create Linux Application

File > New > Application Project

(a) Create a New Application Project

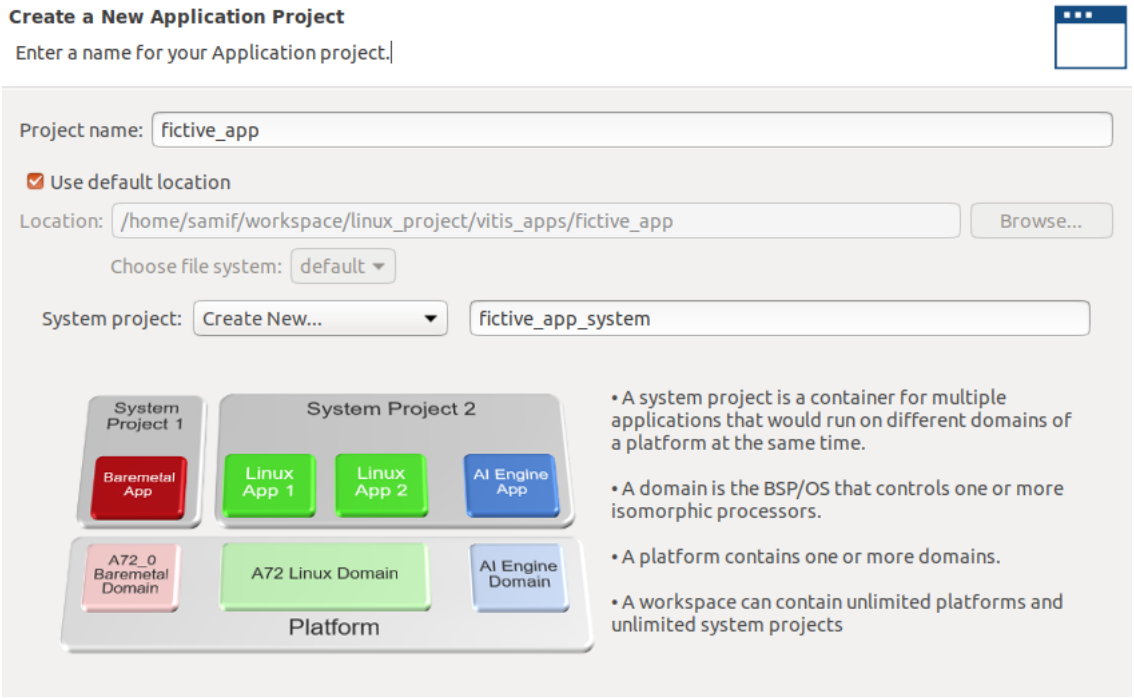


Figure D.15 Vitis create application interface

(b) Platform

Select the Vitis platform that was created previously. If this one is not available, it is possible to go and look for it manually by clicking on the *+*.

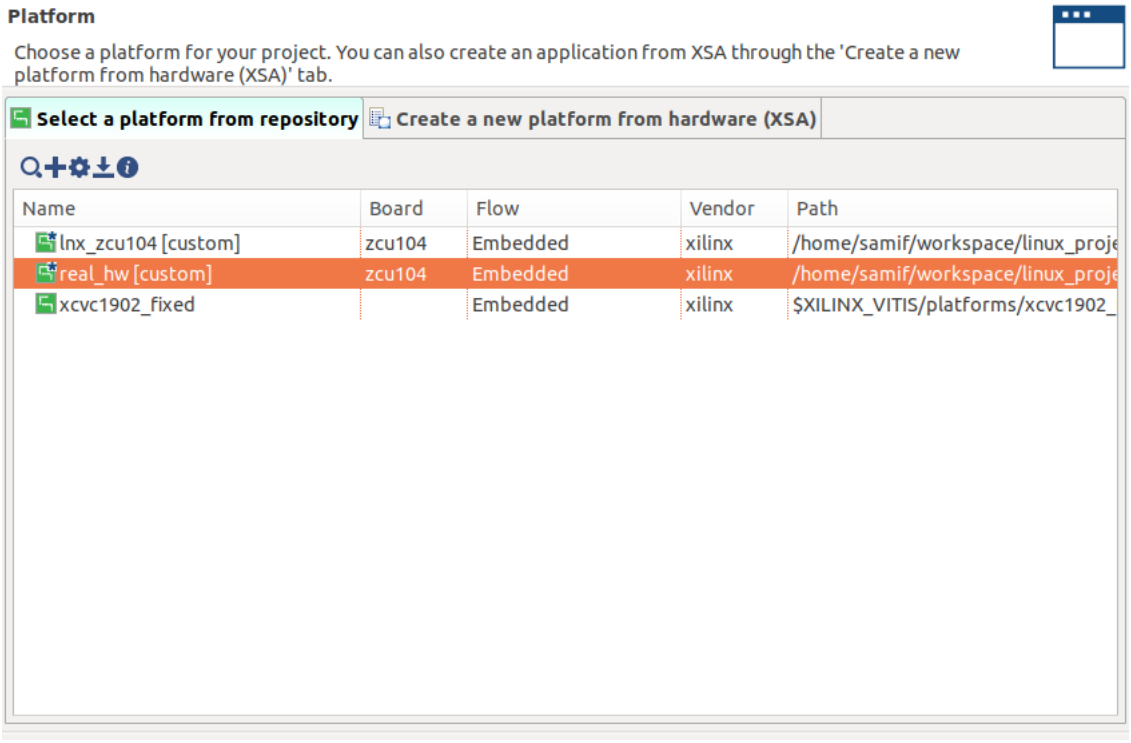


Figure D.16 *Vitis platform application interface*

(c) Domain

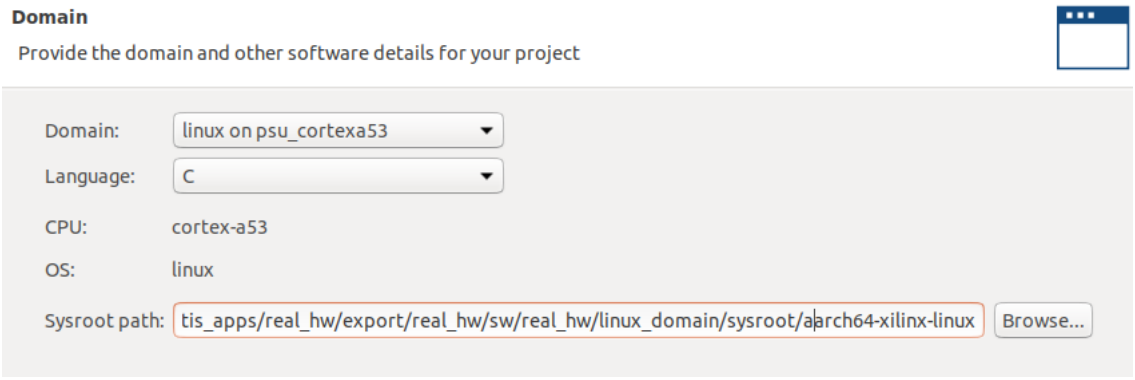


Figure D.17 *Vitis domain application interface*

(d) Templates

For the template, it is better to choose the example *hello world* in order to start with a good structure and all the libraries.

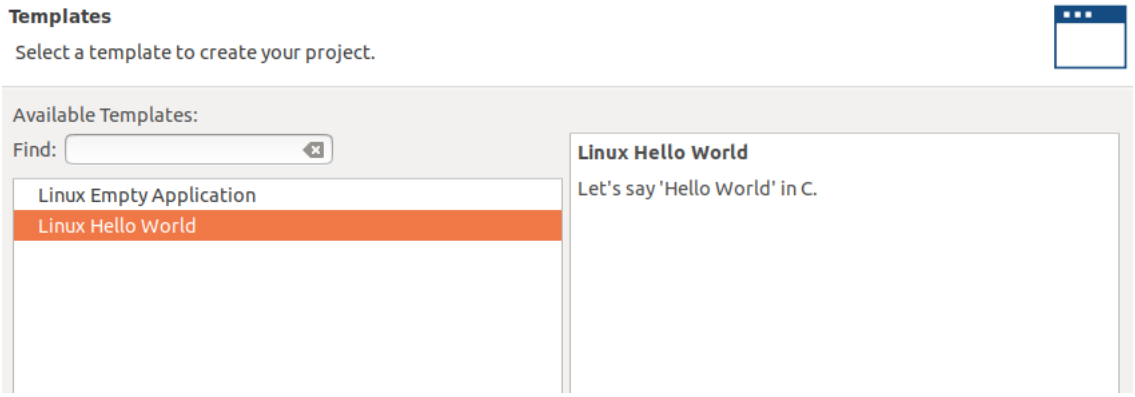


Figure D.18 Vitis template application interface

8. Copy the *main.c* appendix P content in src > helloworld.c file.

9. Build the Linux application using the following icon

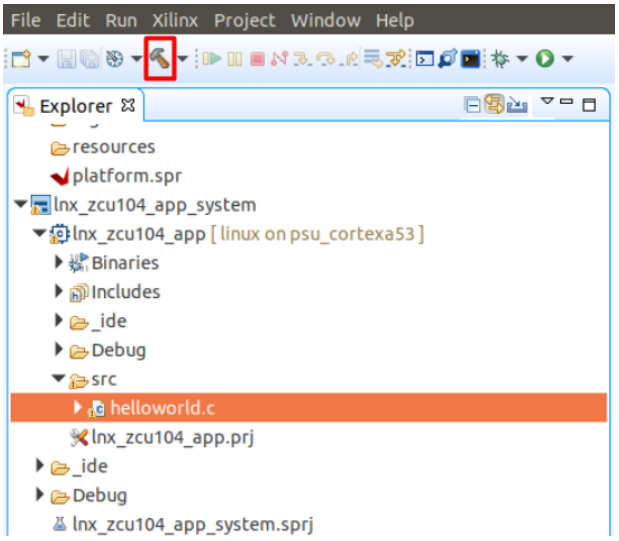


Figure D.19 Vitis build application

D.6.4 Boot Linux image in the target

1. In <vitis-proj-default-path> > <nameapp_system> > Debug > sd_card, copy the files to the SD card in the FAT format partition.

2. Plug the SD card in the SD port of the target

3. Configure the SW6 in SD card boot mode

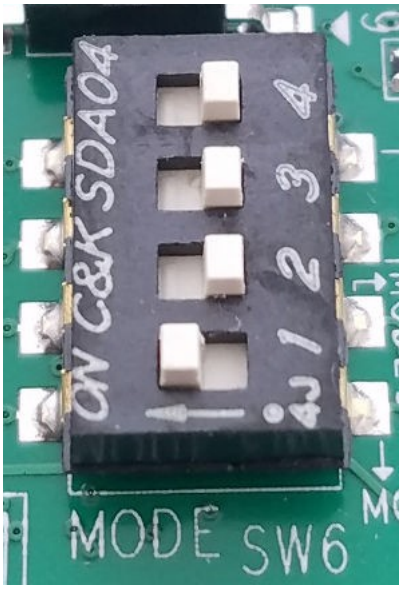


Figure D.20 SW6 SD card boot mode

4. Configure and open a serial terminal using JTAG connection

   Minicom or any other serial terminal.

5. Run the board and wait until you can connect to the petalinux terminal

```
>                  X11 cannot support keycodes above 255.
>                  This warning only shows for the first high keycode.
Errors from xkbcomp are not fatal to the X server
D-BUS per-session daemon address is: unix:abstract=/tmp/dbus-QS0V5wwFge,guid=36b
Failed to launch bus: Failed to execute child process ?/usr/bin? (Permission de6
matchbox: ignoring key shortcut XF86Calendar=!$contacts

matchbox: Cant find a keycode for keysym 2809
matchbox: ignoring key shortcut telephone=!$dates

matchbox: Cant find a keycode for keysym 269025050
matchbox: ignoring key shortcut XF86Start=!matchbox-remote -desktop

[settings daemon] Forking. run with -n to prevent fork

(matchbox-panel:2522): dbind-WARNING **: 13:29:16.022: Error retrieving accessi)

(matchbox-desktop:2521): dbind-WARNING **: 13:29:16.022: Error retrieving acces)

PetaLinux 2019.2 xilinx-zcu104-2019_2 /dev/ttyPS0

xilinx-zcu104-2019_2 login: root
Password:
root@xilinx-zcu104-2019_2:~#
```

Figure D.21 Petalinux terminal connection

NOTE: The login is "root" and the password is "root".

6. Go to the SD card driver location and run the application

```
cd /run/media/mmclplk0p1
./linux_app.elf
```

```
root@xilinx-zcu104-2019_2:/run/media/mmcblk0p1# ./real_lnx_app.elf
Opening a character device file of the Zynq DDR memory...
/dev/mem successfully opened.
Memory map the source address register block.
Source address correctly mapped
Acquisition of data for inference mode . . .
spi mode: 0x0
bits per word: 8
max speed: 50000 Hz (50 KHz)
Communication with slave SPI device successful
Power management successfully configured
Sensibility successfully configured to +/- 2g
Saving acceleration X axis data in the main memory . . .
X axis acceleration data successfully saved in memory
Data normalization using training information . . .

 Creating /run/media/mmcblk0p1/normalized_data.dat file
/run/media/mmcblk0p1/normalized_data.dat file created
Floating point to fixed point transformation . . .
Transformation to fixed point completed!
For input 0.604372, reconstructed value is 0.554688
For input 0.675410, reconstructed value is 0.617188
For input 0.671039, reconstructed value is 0.609375
For input 0.644809, reconstructed value is 0.593750
For input 0.612569, reconstructed value is 0.558594
For input 0.557377, reconstructed value is 0.511719
For input 0.590711, reconstructed value is 0.546875
For input 0.633334, reconstructed value is 0.585938
For input 0.614754, reconstructed value is 0.558594
For input 0.656285, reconstructed value is 0.605469
For input 0.607104, reconstructed value is 0.554688
For input 0.662295, reconstructed value is 0.609375
```

Figure D.22 Petalinux serial terminal running application

NOTE: To debug the application, it is possible to plug an ethernet cable, to get the IP from the petalinux terminal with the *ifconfig* command and to insert it in the Linux TCF Agent configuration of Vitis to generate a connection between Vitis and the board directly. More information is available in the chapter *Hardware Implementation* in the section *Host Application*.

E | MODEL CONFIGURATION FILE

```
#-----------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance Tool on FPGA
#--
#-- File : network_config.ini
#-- Description : Configuration file
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#-----------------------------------------------------------------

[DATA]
TIMESTEP=1
FEATURES=1
DATA_DIR=dataset/final_test
DATA_GEN_NAB_DIR=NAB_files/
NAB_INPUTS=LSTM24_inputs_machine_temperature_system_failure.csv
NAB_RESULTS=LSTM24_machine_temperature_system_failure.csv
TRAIN_RATE=0.9
THRESHOLD=0.0022
WINDOW=len(ds.test_scaled)
[NETWORK]
EPOCHS=200
BATCH_SIZE=64
NEURONS_L1=32
NEURONS_L2=16
NEURONS_L3=4
ONE_STEP=False
[MODEL]
MODEL_DIR=model/
MODEL_NAME=model
```

F | DATA STRUCTURATION

```
#-----------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance on FPGA
#--
#-- File : data_structure.py
#-- Description : Split and transformation of acquired data for training
                                    and testing
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#-----------------------------------------------------------------

import os
import pandas as pd
import configparser
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
sns.set(color_codes=True)
from matplotlib import pyplot as plt
import tensorflow as tf
from numpy.random import seed

print("All libraries are loaded from data structure")

# set random seed
seed(10)
tf.random.set_seed(10)

# load and merge sensor samples
def importData(data_dir):
    merge_data = pd.DataFrame()
    for filename in sorted(os.listdir(data_dir), key=len):
        dataset = pd.read_csv(os.path.join(data_dir, filename), usecols=[
                                    0], header=None)
        dataset.index = np.arange(len(merge_data), len(dataset) + len(
                                    merge_data))
        merge_data = merge_data.append(dataset)

    return merge_data

# split global dataset to train and test datasets
def splitData(merge_data, trainRate):
   train = merge_data[0:int(len(merge_data)*trainRate)]
   test = merge_data[len(train):]

   return train, test

# scale train and test data to [0, 1]
def scale(train, test):
  # fit scaler
  scaler = MinMaxScaler(feature_range=(0, 1))
  scaler = scaler.fit(train)
  # transform train
  train_scaled = scaler.transform(train)
```

```
  train_scaled = train_scaled.astype('float16')
  # transform test
  test_scaled = scaler.transform(test)
  test_scaled = test_scaled.astype('float16')
  return scaler, train_scaled, test_scaled

# inverse scaling for a forecasted value
def invert_scale(scaler, value):
  array = np.array(value)
  array = array.reshape(1, len(array))
  inverted = scaler.inverse_transform(array)
  return inverted[0, -1]

# plot train and test data
def plotTrainTestData(train_data, test_data):
  plt.figure(figsize=(16, 9))
  plt.subplot(211)
  plt.plot(train_data)
  plt.ylabel('Acceleration values (g)')
  plt.title('Train and test data')
  plt.subplot(212)
  plt.plot(test_data)
  plt.xlabel('Time steps')
  plt.ylabel('Acceleration values (g)')
  plt.show()

  return 0

# load configurations
parser = configparser.ConfigParser()
parser.read('network_config.ini')

# variables assignation
data_dir = parser.get('DATA', 'DATA_DIR')
trainRate = parser.getfloat('DATA', 'TRAIN_RATE')

# run data structure processing
all_data = importData(data_dir)
train, test = splitData(all_data, trainRate)
scaler, train_scaled, test_scaled = scale(train, test)
plotData = plotTrainTestData(train_scaled, test_scaled)
print(plotData)
```

G | MODEL TRAINING

```
#-------------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance on FPGA
#--
#-- File : training.py
#-- Description : Model design and training module
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#-------------------------------------------------------------------

import data_structure as ds
import configparser
import datetime
from keras.models import Sequential
from keras.layers import Dense, LSTM
from keras.callbacks import TensorBoard
from model_manipulation import Model
from matplotlib import pyplot as plt

print("All libraries are loaded from training")

# fit a customized LSTM network to training data
def fit_lstm(train, test,
             batch_size,
             nb_epoch,
             timestep,
             neurons_l1, neurons_l2, neurons_l3,
             tensorboard_callback):

    X = train
    Y = test
    X = X.reshape(int(X.shape[0]/timestep), timestep, X.shape[1])  #
                                    reshape into [samples, time steps,
                                    features]
    Y = Y.reshape(int(Y.shape[0]/timestep), timestep, Y.shape[1])
    model = Sequential()

    # hidden layers customization
    model.add(LSTM(neurons_l2, return_sequences=True, input_shape=(X.
                                    shape[1], X.shape[2])))
    model.add(LSTM(neurons_l3, return_sequences=True, input_shape=(X.
                                    shape[1], X.shape[2])))
    model.add(LSTM(neurons_l2, return_sequences=True, input_shape=(X.
                                    shape[1], X.shape[2])))
    model.add(Dense(1))

    model.compile(loss='mae', optimizer='adam')
    model.summary()
    hist = model.fit(X, X,
                     epochs=nb_epoch,
                     batch_size=batch_size,
                     verbose="auto",
                     validation_split=0.1,
                     shuffle=True,
```

```
                    callbacks=[tensorboard_callback])


    return model, hist

# fit a customized DENSE network to training data
def fit_dense(train, test,
              batch_size,
              nb_epoch,
              timestep,
              neurons_l1, neurons_l2,
              tensorboard_callback):

    X = train
    Y = test
    X = X.reshape(int(X.shape[0]/timestep), timestep, X.shape[1])  #
                                    reshape into [samples, time steps,
                                    features]
    Y = Y.reshape(int(Y.shape[0]/timestep), timestep, Y.shape[1])
    model = Sequential()
    model.add(Dense(neurons_l1, activation='sigmoid', input_shape=(X.
                                    shape[1], X.shape[2])))
    model.add(Dense(neurons_l2, activation='sigmoid', input_shape=(X.
                                    shape[1], X.shape[2])))
    model.add(Dense(neurons_l2, activation='sigmoid', input_shape=(X.
                                    shape[1], X.shape[2])))
    model.add(Dense(neurons_l1, activation='sigmoid', input_shape=(X.
                                    shape[1], X.shape[2])))
    model.add(Dense(1))

    model.compile(loss='mae', optimizer='adam')
    model.summary()
    hist = model.fit(X, X,
                     epochs=nb_epoch,
                     batch_size=batch_size,
                     verbose="auto",
                     validation_split=0.1,
                     shuffle=True,
                     callbacks=[tensorboard_callback])

    return model, hist

#  plot the cost function results from training
def cost_function(hist):
    plt.figure(figsize=(16, 9))
    plt.plot(hist.history['loss'])
    plt.plot(hist.history['val_loss'])
    plt.title('Model loss')
    plt.ylabel('loss MAE')
    plt.xlabel('epoch')
    plt.legend(['train', 'validation'], loc='upper left')
    plt.show()


# load configurations
parser = configparser.ConfigParser()
parser.read('network_config.ini')
```

```
# variables assignation
batch_size = parser.getint('NETWORK', 'BATCH_SIZE')
nb_epochs = parser.getint('NETWORK', 'EPOCHS')
timestep = parser.getint('DATA', 'TIMESTEP')
neurons_l1 = parser.getint('NETWORK', 'NEURONS_L1')
neurons_l2 = parser.getint('NETWORK', 'NEURONS_L2')
neurons_l3 = parser.getint('NETWORK', 'NEURONS_L3')
model_dir = parser.get('MODEL', 'MODEL_DIR')
model_name = parser.get('MODEL', 'MODEL_NAME')
LSTM_MODEL = True

# define tensorboard callback
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

# call the fit function for LSTM or Dense model
if LSTM_MODEL:
    model, hist = fit_lstm(
        ds.train_scaled,
        ds.test_scaled,
        batch_size,
        nb_epochs,
        timestep,
        neurons_l1, neurons_l2, neurons_l3,
        tensorboard_callback
    )
else:
    model, hist = fit_dense(
        ds.train_scaled,
        ds.test_scaled,
        batch_size,
        nb_epochs,
        timestep,
        neurons_l1, neurons_l2, neurons_l3,
        tensorboard_callback
    )

# save the model files (.json and .h5) and the history logs of cost
                                        function
save = Model(model_dir, model_name)
save.saveModel(model)
cost_function(hist)

# print the last value of training loss for threshold reference
print(hist.history['val_loss'][-1:])
```

H | SAVE AND LOAD MODEL

```
from keras.models import model_from_json

class Model:

    def __init__(self, path, model_name):
        self.path = path
        self.model_name = model_name

    def saveModel(self, model):
        # serialize model to JSON
        model_json = model.to_json()
        with open(self.path + self.model_name + '.json', "w") as
                                        json_file:
            json_file.write(model_json)
        # serialize weights to HDF5
        model.save_weights(self.path + self.model_name + '.h5')
        print("Saved model to disk")

        return 0

    def loadModel(self):
        # load json and create model
        json_file = open(self.path + self.model_name + '.json', 'r')
        loaded_model_json = json_file.read()
        json_file.close()
        lstm_model = model_from_json(loaded_model_json)
        # load weights into new model
        lstm_model.load_weights(self.path + self.model_name + '.h5')
        print("Loaded model from disk")

        return lstm_model
```

I MODEL TRAINING ANALYSIS

```
#-------------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance on FPGA
#--
#-- File : train_analysis.py
#-- Description : Analysis of training results and recording of useful
                                    information
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#-------------------------------------------------------------------

import data_structure as ds
import pandas as pd
from matplotlib import pyplot as plt
from model_manipulation import Model
import seaborn as sns
sns.set(color_codes=True)
import numpy as np
import configparser

print("All libraries are loaded from train analysis")

# load configurations
parser = configparser.ConfigParser()
parser.read('network_config.ini')

# variables assignation
model_dir = parser.get('MODEL', 'MODEL_DIR')
model_name = parser.get('MODEL', 'MODEL_NAME')
batch_size = parser.getint('NETWORK', 'BATCH_SIZE')
Threshold = parser.get('DATA', 'THRESHOLD')

# load model from json and h5 files
model = Model(model_dir, model_name)
model = model.loadModel()

# reconstruct the entire training dataset to build up state for
                                    reconstruction and threshold
                                    definition
train_reshaped = ds.train_scaled[:, 0].reshape(len(ds.train_scaled), 1, 1
                                    )
print("Running reconstruction on train data")
trainAnalysis = model.predict(train_reshaped, batch_size=batch_size,
                                    verbose=1)
trainAnalysis = trainAnalysis.reshape(trainAnalysis.shape[0],
                                    trainAnalysis.shape[2])
trainAnalysis = pd.DataFrame(trainAnalysis, columns=ds.train.columns)
trainAnalysis['Expected'] = ds.train_scaled
trainAnalysis['Error'] = np.abs(trainAnalysis[0].values - trainAnalysis['
                                    Expected'].values) #CHANGE
trainAnalysis.index = ds.train.index

# plot L1 distribution (loss distribution)
scored = pd.DataFrame(index=ds.train.index)
```

```
Xtrain = ds.train_scaled.reshape(ds.train_scaled.shape[0], ds.
                                 train_scaled.shape[1])
scored['Loss_L1'] = trainAnalysis['Error']
plt.figure(figsize=(16, 9))
plt.title('L1 Loss Distribution', fontsize=16)
sns.distplot(scored['Loss_L1'], bins=20, kde=True, color='blue')
plt.xlim([0.0, .5])
plt.show()

# plot reconstructed vs expected values from train data
fig, ax = plt.subplots(2, 1, figsize=(16, 9), sharex=True)
ax[0].set_title("Reconstructed vs expected train data")
ax[0].set_ylabel('Acceleration values (g)')
ax[0].plot(trainAnalysis['Expected'].values, label='Expected')
ax[0].plot(trainAnalysis[0].values, label='Reconstructed')
ax[1].set_title("L1 reconstruction loss")
ax[1].set_xlabel('Time steps')
ax[1].set_ylabel('Loss values')
ax[1].plot(trainAnalysis['Error'].values, label='L1 error')
plt.show()

# save informations to use in ARM software processing
soft_data = pd.DataFrame(columns=["Values"], index=["Threshold", "Min
                                 train value", "Max train value"])
soft_data.loc["Threshold", "Values"] = Threshold
soft_data.loc["Min train value", "Values"] = np.min(ds.train.values)
soft_data.loc["Max train value", "Values"] = np.max(ds.train.values)
soft_data.to_csv('ARM_software_infos/training_info.txt', header=None)
```

J | MODEL TEST RECONSTRUCTION

```
#------------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance on FPGA
#--
#-- File : prediction.py
#-- Description : Model analysis module on test data
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#------------------------------------------------------------------
import data_structure as ds
import pandas as pd
import configparser
from matplotlib import pyplot as plt
from pandas import DataFrame
from model_manipulation import Model
import numpy as np
import seaborn as sns

print("All libraries are loaded from prediction")

# prediction and create dataframe for storing information
def dataStoring(model, data_scaled, window, timestep, features,
                                    batch_size, one_step):
  dfAnomaly = pd.DataFrame(index=np.arange(window), columns=['Expected',
                                    'Reconstructed', 'Error'], dtype=np.
                                    float64)

  # prediction using one input after the other (batch size of 1)
  if one_step:
    for i in range(window):

      # make one-step forecast
      X = data_scaled[i]
      X = X.reshape(X.shape[0], timestep, features)
      yhat = model.predict(X, batch_size=batch_size)
      yhat = yhat[0,0]

      # store forecast
      dfAnomaly['Reconstructed'].iloc[i] = yhat
      expected = data_scaled[i]
      dfAnomaly['Expected'].iloc[i] = data_scaled[i]
      print('DataNum=%d, Reconstructed=%f, Expected=%f' % (i+1, yhat,
                                    expected))

      # report performance)
      test_mae_loss = np.mean(np.abs(yhat-expected))
      print('Test L1 loss: %.3f' % test_mae_loss)
      dfAnomaly['Error'].iloc[i] = test_mae_loss

  # prediction using several inputs at the same time (batch size greather
                                     than 1)
  else:
```

```
    X = data_scaled[0:window]
    X = X.reshape(X.shape[0], timestep, features)
    print("Running reconstruction of test data")
    yhat = model.predict(X, batch_size=batch_size, verbose=1)
    yhat = yhat.reshape(window, 1)
    dfAnomaly['Reconstructed'] = yhat
    dfAnomaly['Expected'] = data_scaled[0:window]
    dfAnomaly['Error'] = np.abs(dfAnomaly['Reconstructed'].values -
                                dfAnomaly['Expected'].values)

  # looking at the summary of the model
  model.summary()

  return dfAnomaly

# plots of expected vs reconstructed/predicted values
def showResults(dfAnomaly, window, threshold):
  plt.figure(figsize=(16, 9))
  plt.plot(dfAnomaly['Expected'])
  plt.plot(dfAnomaly['Reconstructed'])
  plt.ylabel('Acceleration values (g)')
  plt.xlabel('Time steps')
  plt.legend(['Target', 'Reconstruction'])
  plt.title('Reconstruction of acceleration values')
  plt.show()

  # summarize results
  results = DataFrame()
  results['L1 Loss'] = dfAnomaly['Error'].values
  print(results.describe())
  plt.show()

  # calculate the l1 loss (norm loss)
  scored = pd.DataFrame(index=np.arange(0, window))
  scored['L1 Loss'] = dfAnomaly['Error']
  scored['Threshold'] = threshold
  scored['Anomaly'] = scored['L1 Loss'] > scored['Threshold']
  # plot mae for anomaly detection
  scored.plot(logy=True, figsize=(16, 9), color=['blue', 'red'])
  plt.title("Absolute errors of reconstructed and expected values")
  plt.xlabel("Time steps")
  plt.ylabel("Absolute errors")
  plt.show()

  # plot anomalies in testing data
  anomalies = scored[scored.Anomaly == True]
  anomalies.head()
  ds.test.index = np.arange(0, len(ds.test))
  plt.figure(figsize=(16, 9))
  plt.title("Anomalies identification in real data")
  plt.xlabel("Time steps")
  plt.ylabel("Acceleration values (g)")
  plt.plot(
    ds.test[:window].index,
    ds.test[:window],
    label='Acceleration values (g)'
  )
```

```
  sns.scatterplot(
    x=anomalies.index,
    y=ds.test.loc[anomalies.index, 0],
    color=sns.color_palette()[3],
    s=52,
    label='anomaly'
  )

  plt.xticks(rotation=25)
  plt.legend()
  plt.show()

  return scored

# load configurations
parser = configparser.ConfigParser()
parser.read('network_config.ini')

# variables assignation
model_dir = parser.get('MODEL', 'MODEL_DIR')
model_name = parser.get('MODEL', 'MODEL_NAME')
timestep = parser.getint('DATA', 'TIMESTEP')
features = parser.getint('DATA', 'FEATURES')
threshold = parser.getfloat('DATA', 'THRESHOLD')
batch_size = parser.getint('NETWORK', 'BATCH_SIZE')
one_step = parser.getboolean('NETWORK', 'ONE_STEP')
window = len(ds.test_scaled)

# load model from json and h5 files
model = Model(model_dir, model_name)
model = model.loadModel()

# storing data and results in a dataframe
dataResume = dataStoring(
  model,
  ds.test_scaled,
  window,
  timestep,
  features,
  batch_size,
  one_step
)
```

K | STATISTICS ANALYSIS

```python
#-------------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance on FPGA
#--
#-- File : statistics.py
#-- Description : Statistical complement to the model analysis on the
                                    test data
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#-------------------------------------------------------------------

import numpy as np
from matplotlib import pyplot as plt
from prediction import dataResume, window

print("All libraries are loaded from statistics")

# computation of statistics parameters
def compute_statistics(error):
  error = error.reshape(len(error), 1)
  mean = error.mean()
  cov = 0
  for e in error:
    cov += np.dot((e - mean).reshape(len(e), 1), (e - mean).reshape(1,
                                    len(e)))
  cov /= len(error)

  return error, mean, cov

# computation of the mahalanobis distance
def compute_mahalanobis(x, mean, cov):
  d = np.dot(x-mean, np.linalg.inv(cov))
  d = np.dot(d, (x-mean).T)
  return d

# list and save mahalanobis distance values
def list_mahalanobis(error, mean, cov):
  m_dist = []
  for e in error:
    m_dist.append(compute_mahalanobis(e, mean, cov))

  return m_dist

# plot the mahalanobis distance
def plotMahalanobisDistance(m_dist, window):
  plt.figure(figsize=(16, 9))
  plt.style.use('ggplot')
  plt.plot(m_dist, color='r', label='Mahalanobis Distance')
  plt.title("Mahalanobis distance study")
  plt.xlabel('Time steps')
  plt.ylabel('Mahalanobis Distance')
  plt.xlim(-10, window + 10)
  plt.ylim(0, max(m_dist) + 5)
```

```
  plt.legend(fontsize=15)
  plt.show()

# compute statistics paramaters
error, mean, cov = compute_statistics(dataResume['Error'].values)

# mahalanobis distance list values
m_dist = list_mahalanobis(error, mean, cov)

# plot mahalanobis distance values
plotMahalanobisDistance(m_dist, window)
```

L | NAB FORMAT GENERATION

```
#-------------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance on FPGA
#--
#-- File : detector_analysis.py
#-- Description : Structure and files preparation for NAB analysis
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#-------------------------------------------------------------------

import data_structure as ds
import pandas as pd
import json
import numpy as np
import configparser
from model_manipulation import Model

print("All libraries from detector analysis are loaded")
print("Preparation of CSV files for NAB analysis")

# save inputs in NAB format frame
def getInputs(inputs, timestamp):
    inputsData = pd.DataFrame(index=np.arange(len(inputs)), columns=['
                                    timestamp', 'value'])

    for i in range(len(inputs)):
        inputsData['timestamp'].iloc[i] = timestamp.iloc[i]
        input = inputs[i]
        input = ds.invert_scale(ds.scaler, input)
        inputsData['value'].iloc[i] = input

    return inputsData

# running prediction using NAB data and save results in NAB format frame
def algoRunning(model, data, timestep, features, timestamp):
    reconstruction = pd.DataFrame(index=np.arange(len(data)), columns=['
                                    timestamp', 'value'])
    error = list()
    reconstruction['timestamp'] = timestamp
    for i in range(len(data)):
        X = data[i]
        X = X.reshape(X.shape[0], timestep, features)
        yhat = model.predict(X, batch_size=1)
        yhat = yhat[0,0]
        gap = np.abs(yhat - X)
        yhat = ds.invert_scale(ds.scaler, yhat)
        reconstruction['value'].iloc[i] = yhat
        error.append(gap[0,0])
        print('Loop' + str(i), 'Norm error =' + str(gap[0,0]))

    return reconstruction, error

# anomaly classification using detection threshold
def getAnomalyScore(error, threshold):
```

```
    anomalyBoard = pd.DataFrame(index=np.arange(len(error)))
    anomalyBoard['Error'] = error
    anomalyBoard['Threshold'] = threshold
    anomalyBoard['anomaly_score'] = anomalyBoard['Error'] > anomalyBoard[
                                    'Threshold']
    anomaly_score = anomalyBoard['anomaly_score'].values
    anomaly_score = anomaly_score.reshape(-1, 1)

    return anomaly_score

# add real anomalies in the frame (reference in NAB/labels/
                                    combined_windows.json)
def getLabel(timestamp, timestampLabel):

    label = (timestamp >= timestampLabel[0][0]) & (timestamp <=
                                    timestampLabel[0][1]) |\
        (timestamp >= timestampLabel[1][0]) & (timestamp <=
                                    timestampLabel[1][1]) |\
        ((timestamp >= timestampLabel[2][0]) & (timestamp <=
                                    timestampLabel[2][1])) |\
        ((timestamp >= timestampLabel[3][0]) & (timestamp <=
                                    timestampLabel[3][1]))

    return label

# generate csv file in NAB format
def getCSV(inputs, results):
    inputs.to_csv(gen_nab_dir + nab_inputs, index=False)
    results.to_csv(gen_nab_dir + nab_results, index=False)

    return 0

# load configurations
parser = configparser.ConfigParser()
parser.read('network_config.ini')

# variables assignation
model_dir = parser.get('MODEL', 'MODEL_DIR')
model_name = parser.get('MODEL', 'MODEL_NAME')
timestep = parser.getint('DATA', 'TIMESTEP')
features = parser.getint('DATA', 'FEATURES')
threshold = parser.getfloat('DATA', 'THRESHOLD')
gen_nab_dir = parser.get('DATA', 'DATA_GEN_NAB_DIR')
nab_inputs = parser.get('DATA', 'NAB_INPUTS')
nab_results = parser.get('DATA', 'NAB_RESULTS')
data_dir = parser.get('DATA', 'DATA_DIR')

# load model from json and h5 files
model = Model(model_dir, model_name)
model = model.loadModel()

# load json file for labeling part
with open('dataset/NAB_real_anomalies/combined_windows.json') as f:
   labelWindow = json.load(f)
timestampLabel = labelWindow["realKnownCause/
                                    machine_temperature_system_failure.
                                    csv"]
```

```
# get inputs from nab data
allValues = np.concatenate((ds.train_scaled, ds.test_scaled))
timestamp = pd.read_csv(data_dir + '/machine_temperature_system_failure.
                                    csv', usecols=[0])
inputsNAB = getInputs(allValues, timestamp)
# running model with nab data
results, error = algoRunning(model, allValues, timestep, features,
                                    timestamp)
# get binary anomaly scores (anomalies = 1, no-anomalies = 0)
anomalyScore = getAnomalyScore(np.array(error), threshold)
results['anomaly_score'] = anomalyScore
results['anomaly_score'] = results['anomaly_score'].astype(int)
# get label values (real anomalies datetime windows)
label = getLabel(timestamp, timestampLabel)
results['label'] = label
results['label'] = results['label'].astype(int)
convertCSV = getCSV(inputsNAB, results)

print("CSV inputs and results files are ready")
```

M | RESULTS SAVING FOR C/RTL COSIMULATION

```
#-----------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance Tool FPGA
#--
#-- File : training.py
#-- Description : Saving results in .dat format for HLS4ML conversion
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 14.01.2022
#-----------------------------------------------------------------

import prediction as pr
import pandas as pd

# input data save as .dat
output_df = pd.DataFrame({'X axis': pr.dataResume['Expected'].values})
output_df.to_csv('tb_data/input_data.dat', index=False)

# output reconstruction save data as .dat
output_df = pd.DataFrame({'X axis': pr.dataResume['Reconstructed'].values
                                    })
output_df.to_csv('tb_data/reconstruction.dat', index=False)
```

N | PLOT RESULTS PROGRAM

```
#-----------------------------------------------------------------
#-- Master Thesis - Model-Based Predictive Maintenance on FPGA
#--
#-- File : plot_results.py
#-- Description : Simulation and hardware levels results plotting
#--
#-- Author : Sami Foery
#-- Master : MSE Mechatronics
#-- Date : 03.02.2022
#-----------------------------------------------------------------

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from scipy.stats import norm
import statistics

# load files and build data frames
def loadFiles():
    expected = pd.read_csv('tb_data/tb_input_features.dat')
    reconstruction = pd.read_csv('tb_data/tb_output_predictions.dat')
    csim = pd.read_csv('tb_data/csim_results.log')
    hw = pd.read_csv('tb_data/logs.dat')
    real_data = pd.read_csv('tb_data/data_training.dat')
    real_test = pd.read_csv('tb_data/data_test.dat')
    spi_data = pd.read_csv('tb_data/normalized_data.dat')
    spi_reconstructed = pd.read_csv('tb_data/reconstructed_data.dat')
    dfCompare = pd.DataFrame(
        index=np.arange(len(reconstruction.values)),
        columns=['Expected',
                 'Reconstruction',
                 'Csim',
                 'hw',
                 'Conversion error',
                 'Threshold',
                 'Validation',
                 'Reconstruction error'
                 ]
    )
    dfCompare['Expected'] = expected
    dfCompare['Reconstruction'] = reconstruction
    dfCompare['Csim'] = csim
    dfCompare['hw'] = hw

    dfRealData = pd.DataFrame(
        index=np.arange(len(real_data.values)),
        columns=['X Axis Training',
                 'X Axis Test']
    )
    dfRealData['X Axis Training'] = real_data
    dfRealData['X Axis Test'] = real_test

    dfSPI = pd.DataFrame(
        index=np.arange(len(spi_data.values)),
```

```
        columns=['X Axis Real Data',
                 'X Axis Reconstructed Data',
                 'X Axis Reconstruction Error',
                 'Threshold']
    )
    dfSPI['X Axis Real Data'] = spi_data
    dfSPI['X Axis Reconstructed Data'] = spi_reconstructed
    dfSPI['X Axis Reconstruction Error'] = np.abs(dfSPI['X Axis Real Data
                                    '].values - dfSPI['X Axis
                                    Reconstructed Data'].values)
    dfSPI['Threshold'] = 0.068
    dfSPI['Anomaly'] = dfSPI['X Axis Reconstruction Error'] > dfSPI['
                                    Threshold']

    return dfCompare, dfRealData, dfSPI

# plot C simulation results for comparison
def conversionAnalysis(table):
    table['Conversion error'] = np.abs(table['Reconstruction'].values -
                                    table['hw'].values)
    table['Threshold'] = np.abs(max(table['Conversion error'].values))
    table['Validation'] = table['Conversion error'].values < table['
                                    Threshold'].values
    table['Reconstruction error'] = np.abs(table['Expected'].values -
                                    table['hw'].values)

    return table

# add plot hardware level results
def showResults(fullFrame):
    fig, ax = plt.subplots(3, 1, figsize=(16, 6), sharex=True)
    ax[0].set_title("Comparison between expected, reconstructed and
                                    converted values")
    ax[0].set_ylabel("Values")
    ax[0].plot(fullFrame['Expected'].values)
    ax[0].plot(fullFrame['Reconstruction'].values)
    ax[0].plot(fullFrame['Csim'].values)
    ax[0].plot(fullFrame['hw'].values)
    ax[0].legend(['Expected', 'Reconstructed', 'HW sim results', 'HW
                                    results'])
    ax[1].set_title("Conversion error")
    ax[1].set_ylabel("Values")
    ax[1].plot(fullFrame['Conversion error'].values)
    ax[2].set_title("Reconstruction error in HW level")
    ax[2].set_xlabel("Index")
    ax[2].set_ylabel("Error values")
    ax[2].plot(fullFrame['Reconstruction error'].values)
    plt.show()

    return None

# plot acquired data
def acquisitionDATA(dataFrame):
    plt.figure(figsize=(16, 6))
    plt.title("SPI X axis data acquisition")
    plt.ylabel("Acceleration values (g)")
    plt.plot(dataFrame["X Axis Training"].values)
    plt.plot(dataFrame["X Axis Test"].values)
```

```
    plt.legend(['Real train data', 'Real test data'])
    plt.show()

    return None

# plot results from realtime test logs
def showReal(SPIFrame):
    fig, ax = plt.subplots(3, 1, figsize=(16, 6), sharex=True)
    ax[0].set_title("X axis real and reconstructed data comparison")
    ax[0].set_ylabel("Acceleration (g)")
    ax[0].plot(SPIFrame['X Axis Real Data'].values)
    ax[0].plot(SPIFrame['X Axis Reconstructed Data'].values)
    ax[0].legend(['Expected', 'Reconstructed'], loc="lower right")
    ax[1].set_title("X axis reconstruction error")
    ax[1].set_ylabel("Absolute error")
    ax[1].plot(SPIFrame['X Axis Reconstruction Error'].values)
    ax[1].plot(SPIFrame['Threshold'].values, color="red")
    ax[1].legend(["Error", "Threshold"], loc="lower right")
    ax[2].set_title("Anomalies identification")
    ax[2].set_xlabel("Time steps")
    ax[2].set_ylabel("Acceleration (g)")
    ax[2].plot(SPIFrame['X Axis Real Data'].values)

    # plot anomaly in testing data
    anomalies = SPIFrame[SPIFrame.Anomaly == True]
    anomalies.head()
    SPIFrame['X Axis Real Data'].index = np.arange(0, len(SPIFrame['X
                                    Axis Real Data']))

    sns.scatterplot(
        x=anomalies.index,
        y=SPIFrame['X Axis Real Data'].iloc[anomalies.index],
        color=sns.color_palette()[3],
        s=52,
        label="Anomaly"
    )
    ax[2].legend(loc='lower right')
    plt.show()

    # normal distribution on train reconstruction error
    error = SPIFrame['X Axis Reconstruction Error'].values
    mean = statistics.mean(error)
    sd = statistics.stdev(error)
    plt.plot(error, norm.pdf(error, mean, sd))
    plt.show()

    return None

loadTable, loadRealData, loadSPI  = loadFiles()
fullTable = conversionAnalysis(loadTable)
showResults(fullTable)
acquisitionDATA(loadRealData)
showReal(loadSPI)
```

O RELEVANT TRAINED MODELS LIST

Trained models configuration

Model	Hidden layers	Neurons	Epochs	Batch size	Dropout	Return sequences	Cost function	Activation function	NAB score (STD, FP, FN)	Remark
24	2 LSTM	8\16	200	64	0	True	MAE	tanh	50/50/67	Good results in NAB scoring
25	2 LSTM	8\16	100	64	0.1	True	MAE	tanh	45/37/63	
25.1	2 LSTM	8\16	200	64	0.1	True	MAE	tanh	44/37/63	
26	2 LSTM	8\16	200	64	0	True	MSE	tanh	34/17/56	
27	4 Dense	32/16	200	64	0	True	MAE	Sigmoid	48/46/65	
28	3 LSTM	16\4\16	200	64	0	True	MAE	tanh	49/47/65	Best results in NAB score using multiple datasets

Eliminated model
Possible model

Figure O.1 Relevant trained models list

P | LINUX HOST APPLICATION

```
/*
   **HOST LINUX APPLICATION**

   Master Thesis - Model-Based Predictive Maintenance Tool on FPGA
   Master: Mechatronics Engineering
   Author: Sami Foery
   Date : 26.01.2022

 */

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <getopt.h>
#include <fcntl.h>
#include <memory.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <linux/ioctl.h>
#include <linux/spi/spidev.h>
#include <linux/types.h>

// If not define -> TRAINING MODE (only data acquisition)
#define INFERENCE_MODE

// SPI SLAVE DEVICE REGISTERS
#define ARRAY_SIZE(a)         (sizeof(a) / sizeof((a)[0]))
#define READREG(a)          (a | 0x80) // macro to read register in SPI
    slave
#define WHO_AM_I_REG        0x75
#define PM_REG            0x6B
#define SENSI_REG           0x1C
#define ACCEL_XREG1         0x3B
#define ACCEL_XREG2         0x3C

// AXI GPIO REGISTERS
#define GPIO_DATA           0x0000
#define GPIO_TRI          0x0004

// SOURCE REGISTER TO SAVE DATA
#define SOURCE_ADDRESS        0x10000000

// LIMIT COMPARISON FOR NEGATIVE VALUES
#define LIM             50000

// LENGTH MEMORY MAP AND FRACTIONAL PRECISION
#define LENGTH            getpagesize()*16
#define MAXBUFF           LENGTH/4
#define FIXED_POINT_FRACTIONAL_BITS 8
```

```
// REFERENCES FROM TRAIN DATA
#define TRAIN_MAX         0.248779
#define TRAIN_MIN         -0.197998

// SPI INITIALIZATION VALUES
static uint32_t mode;
static uint8_t bits = 8;
static uint32_t speed = 50000;
static uint16_t delay = 0;

static void pabort(const char *s)
{
  perror(s);
  abort();
}


// SPI transfer function
static void transfer(int fd, uint8_t const *tx, uint8_t const *rx, size_t
    len)
{
  int ret;
  struct spi_ioc_transfer tr = {
    .tx_buf = (unsigned long)tx,
    .rx_buf = (unsigned long)rx,
    .len = len,
    .delay_usecs = delay,
    .speed_hz = speed,
    .bits_per_word = bits,
  };

  ret = ioctl(fd, SPI_IOC_MESSAGE(1), &tr);
  if (ret < 1)
    pabort("can't send spi message");
}

// SPI data acquisition
void* spi_data(float src[MAXBUFF])
{

  uint8_t comm_config_tx[] = {READREG(WHO_AM_I_REG), 0xFF};
  uint8_t pm_config_tx[] = {PM_REG, 0x01};
  uint8_t sensi_config_tx[] = {SENSI_REG, 0x00};

  uint8_t accel_tx[] = {
    READREG(ACCEL_XREG1), READREG(ACCEL_XREG2), 0xFF,
  };

  uint8_t comm_config_rx[ARRAY_SIZE(comm_config_tx)] = {0, };
  uint8_t pm_config_rx[ARRAY_SIZE(pm_config_tx)] = {0, };
  uint8_t sensi_config_rx[ARRAY_SIZE(sensi_config_tx)] = {0, };
  uint8_t accel_rx[ARRAY_SIZE(accel_tx)] = {0, };


  static const char *device = "/dev/spidev1.0";
  int fd;
  int ret = 0;
  int i = 0;
```

```
fd = open(device, O_RDWR);
if (fd <= 0) {
  perror("Unable to open the /dev/spidev1.0 driver\n");
  exit(1);
};

/*
 * Spi mode
 */
ret = ioctl(fd, SPI_IOC_WR_MODE, &mode);
if (ret == -1)
  pabort("can't set spi mode");

ret = ioctl(fd, SPI_IOC_RD_MODE, &mode);
if (ret == -1)
  pabort("can't get spi mode");

/*
 * Bits per word
 */
ret = ioctl(fd, SPI_IOC_WR_BITS_PER_WORD, &bits);
if (ret == -1)
  pabort("can't set bits per word");

ret = ioctl(fd, SPI_IOC_RD_BITS_PER_WORD, &bits);
if (ret == -1)
  pabort("can't get bits per word");

/*
 * Max speed Hz
 */
ret = ioctl(fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed);
if (ret == -1)
  pabort("can't set max speed hz");

ret = ioctl(fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed);
if (ret == -1)
  pabort("can't get max speed hz");

printf("spi mode: 0x%x\n", mode);
printf("bits per word: %d\n", bits);
printf("max speed: %d Hz (%d KHz)\n", speed, speed/1000);

// Communication test with slave SPI device using "WHO_AM_I" register
transfer(fd, comm_config_tx, comm_config_rx, sizeof(comm_config_tx));
  // Control communication with slave device


if (comm_config_rx[1] == 0x68){
  printf("Communication with slave SPI device successful\n");
}
else{
  printf("Communication with slave SPI device failed\n");
  exit(1);
}
```

```
  // Power management configuration
  transfer(fd, pm_config_tx, pm_config_rx, sizeof(pm_config_tx));
  printf("Power management successfully configured\n");

  // Sensibility configuration to +/- 2g (1.12 mg/LSB)
  transfer(fd, sensi_config_tx, sensi_config_rx, sizeof(sensi_config_tx))
    ;
  printf("Sensibility successfully configured to +/- 2g\n");

  // Acceleration X data acquisition
  printf("Saving acceleration X axis data in the main memory . . .\n");
  for (i=0; i<MAXBUFF; i++){

    transfer(fd, accel_tx, accel_rx, sizeof(accel_tx));

    usleep(1000);

    src[i] = ((accel_rx[1] << 8) | accel_rx[2]);

    if (src[i] > LIM){
      src[i] = (src[i] - LENGTH)/16384.0f;
    }
    else{
      src[i] = src[i]/16384.0f;
    }
  }
  printf("X axis acceleration data successfully saved in memory\n");

  close(fd);
}

// Function for loading offline data file to the memory
float* load_data(char* sdmem, float src[MAXBUFF], float save[MAXBUFF])
{
  int i=0;
  char row[MAXBUFF];

  FILE *fp = fopen(sdmem, "r");

  if (fp == NULL) {
    perror("Unable to open the file\n");
    exit(1);
  };

  printf("Loading and saving data to memory . . .\n");

  for (i=0; i<MAXBUFF; i++)
  {
    fgets(row, MAXBUFF, fp);
    src[i] = atof(row);
    save[i] = atof(row);
    i = i+1;
  };
  return src;
}

// Function for the normalization of the data using training data based
void* normalize_data(float src[MAXBUFF], float save[MAXBUFF])
```

```
{
  int i = 0;

  for (i=0; i<MAXBUFF; i++){
    src[i] = (src[i] - TRAIN_MIN) / (TRAIN_MAX - TRAIN_MIN);
    save[i] = src[i];
  }
}

// Floating point data to fixed point transformation
void* to_fixed_point(float* data, unsigned int src[MAXBUFF])
{
      unsigned int i = 0;

      printf("Floating point to fixed point transformation . . .\n");

      for (i=0; i<MAXBUFF; i++){
          src[i] = (int) (data[i] * (1 << FIXED_POINT_FRACTIONAL_BITS));
      };
}

// Logs creation from saved data on source register
void* create_logs(float* logs, const char* name)
{
  char* path = "/run/media/mmcblk0p1/";
  unsigned int i = 0;

  FILE *fps;

    char * filename = (char *) malloc(1 + strlen(path)+ strlen(name) );
    strcpy(filename, path);
    strcat(filename, name);
  printf("\n Creating %s file\n",filename);

  fps = fopen(filename,"w+");
  if (fps == NULL) {
    perror("Unable to open the file\n");
    exit(1);
  };

  for(i=0;i<MAXBUFF;i++){

    fprintf(fps,"%f\n",logs[i]);

  }

  fclose(fps);

  printf("%s file created\n",filename);

}

int main(int argc, char *argv[])
{
  int i=0, fd0, fd1, fd2, fd3;
  char *uiod0 = "/dev/uio0";
  char *uiod1 = "/dev/uio1";
  char *uiod2 = "/dev/uio2";
```

```
char *uiod3 = "/dev/uio3";
char* sdmem = "/run/media/mmcblk0p1/full_train.dat";
float* data, logs[MAXBUFF];
float save_data[MAXBUFF];
float RECSTR;
void *gpio_ptr0;
void *gpio_ptr1;
void *gpio_ptr2;
void *gpio_ptr3;

struct timeval start, end;

  // Open the UIO device file to allow access to the device in user
  space

  fd0 = open(uiod0, O_RDWR);
  if (fd0 < 1) {
    printf("Invalid UIO0 device file:%s.\n", uiod0);
  }

  fd1 = open(uiod1, O_RDWR);
if (fd1 < 1) {
  printf("Invalid UIO1 device file:%s.\n", uiod1);
}
fd2 = open(uiod2, O_RDWR);
if (fd2 < 1) {
  printf("Invalid UIO2 device file:%s.\n", uiod2);
}
fd3 = open(uiod3, O_RDWR);
if (fd3 < 1) {
  printf("Invalid UIO3 device file:%s.\n", uiod3);
}

// Mmap the AXI GPIOs devices into user space

  gpio_ptr0 = mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE,
  MAP_SHARED, fd0, 0);
  // If addr is NULL, then the kernel chooses the (page-aligned)
  address at which to create the mapping
  if (gpio_ptr0 == MAP_FAILED) {
    printf("Mmap call failure.\n");
    return -1;
  }
  gpio_ptr1 = mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE,
  MAP_SHARED, fd1, 0);
  // If addr is NULL, then the kernel chooses the (page-aligned)
  address at which to create the mapping
  if (gpio_ptr1 == MAP_FAILED) {
    printf("Mmap call failure.\n");
    return -1;
  }
  gpio_ptr2 = mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE,
  MAP_SHARED, fd2, 0);
  // If addr is NULL, then the kernel chooses the (page-aligned)
  address at which to create the mapping
  if (gpio_ptr2 == MAP_FAILED) {
    printf("Mmap call failure.\n");
    return -1;
```

```
  }
  gpio_ptr3 = mmap(NULL, getpagesize(), PROT_READ|PROT_WRITE,
  MAP_SHARED, fd3, 0);
  // If addr is NULL, then the kernel chooses the (page-aligned)
  address at which to create the mapping
  if (gpio_ptr3 == MAP_FAILED) {
    printf("Mmap call failure.\n");
    return -1;
  }

  // Set bit0 on the GPIO to be output and bit 1 to be input
  *((volatile unsigned *)(gpio_ptr0 + GPIO_TRI)) = 0x00;
  *((volatile unsigned *)(gpio_ptr1 + GPIO_TRI)) = 0x00;
  *((volatile unsigned *)(gpio_ptr2 + GPIO_TRI)) = 0x00;
  *((volatile unsigned *)(gpio_ptr3 + GPIO_TRI)) = 0x01;

  // Assign start and lstm_input_vld input to 1
  *((volatile unsigned *)(gpio_ptr0 + GPIO_DATA)) = 0x01;
  *((volatile unsigned *)(gpio_ptr1 + GPIO_DATA)) = 0x01;

  printf("Opening a character device file of the Zynq DDR memory...\n")
  ;
  int ddr_memory = open("/dev/mem", O_RDWR | O_SYNC);
  if (ddr_memory == -1) {
      perror("/dev/mem could not be opened.\n");
      exit(1);
  } else {
    printf("/dev/mem successfully opened.\n");
  }

  // Create memory map from source address register
  printf("Memory map the source address register block.\n");
  unsigned int* source_addr  = mmap(NULL, LENGTH, PROT_READ |
  PROT_WRITE, MAP_SHARED, ddr_memory, SOURCE_ADDRESS); // SOURCE address
  if (source_addr == MAP_FAILED) {
      perror("mmap source_addr\n");
      exit(1);
  }
  else{
    printf("Source address correctly mapped\n");
  }

#ifdef INFERENCE_MODE

  while(1){

    printf("Acquisition of data for inference mode . . .\n");

    gettimeofday(&start, NULL);
  spi_data(source_addr);
  gettimeofday(&end, NULL);
  printf("Time taken for spi acquisition is : %ld micro seconds\n",
      ((end.tv_sec * 1000000 + end.tv_usec) -
      (start.tv_sec * 1000000 + start.tv_usec)));
  //create_logs(source_addr, "spi_data.dat");

  //load_data(sdmem, source_addr, save_data);
```

```
    // Normalization of saved data using train data as reference
    gettimeofday(&start, NULL);
    normalize_data(source_addr, save_data);
    gettimeofday(&end, NULL);
    printf("Time taken for normalization is : %ld micro seconds\n",
        ((end.tv_sec * 1000000 + end.tv_usec) -
        (start.tv_sec * 1000000 + start.tv_usec)));

    create_logs(source_addr, "normalized_data.dat");

    // Transformation of floating point data to fixed point
    gettimeofday(&start, NULL);
    to_fixed_point(save_data, source_addr);
    gettimeofday(&end, NULL);
    printf("Time taken for fixed point transformation is : %ld micro
   seconds\n",
        ((end.tv_sec * 1000000 + end.tv_usec) -
        (start.tv_sec * 1000000 + start.tv_usec)));

    printf("Transformation to fixed point completed!\n");

    i=0;

    // Hardware acceleration using AXI GPIOs
    gettimeofday(&start, NULL);
    for (i=0; i<MAXBUFF; i++){
      *((volatile unsigned *)(gpio_ptr2 + GPIO_DATA)) = source_addr[i];
      sleep(0.000000001);
      RECSTR = *((volatile signed *)(gpio_ptr3 + GPIO_DATA));

      if (RECSTR > LIM) // Condition for negative results
      {
        RECSTR = (RECSTR - LENGTH);
      }

      // Fixed point to floating point transformation
      RECSTR = RECSTR/256.0f;

      logs[i] = RECSTR;

      printf("For input %f, reconstructed value is %f\n", save_data[i],
    logs[i]);

    }
    gettimeofday(&end, NULL);
    printf("Time taken for reconstruction is : %ld micro seconds\n",
      ((end.tv_sec * 1000000 + end.tv_usec) -
      (start.tv_sec * 1000000 + start.tv_usec)));

    create_logs(logs, "reconstructed_data.dat");

    }

  #else

    i = 0;
    char buffer[32];
```

```
    // Acquisition of 20 datasets for training mode (327'680 datas)
    for (i=0; i<20; i++){

    printf("Acquisition of data for training mode . . .\n");

    sprintf(buffer, "data_training_%i.dat", i);

    spi_data(source_addr);
    create_logs(source_addr, buffer);
    }

  #endif

    // Unmap the AXI GPIOs devices and source address from user space
    munmap(gpio_ptr0, 4096);
    munmap(gpio_ptr1, 4096);
    munmap(gpio_ptr2, 4096);
    munmap(gpio_ptr3, 4096);
    munmap(source_addr, LENGTH);

    return 0;
}
```

Q | AXI DMA Based Block Design

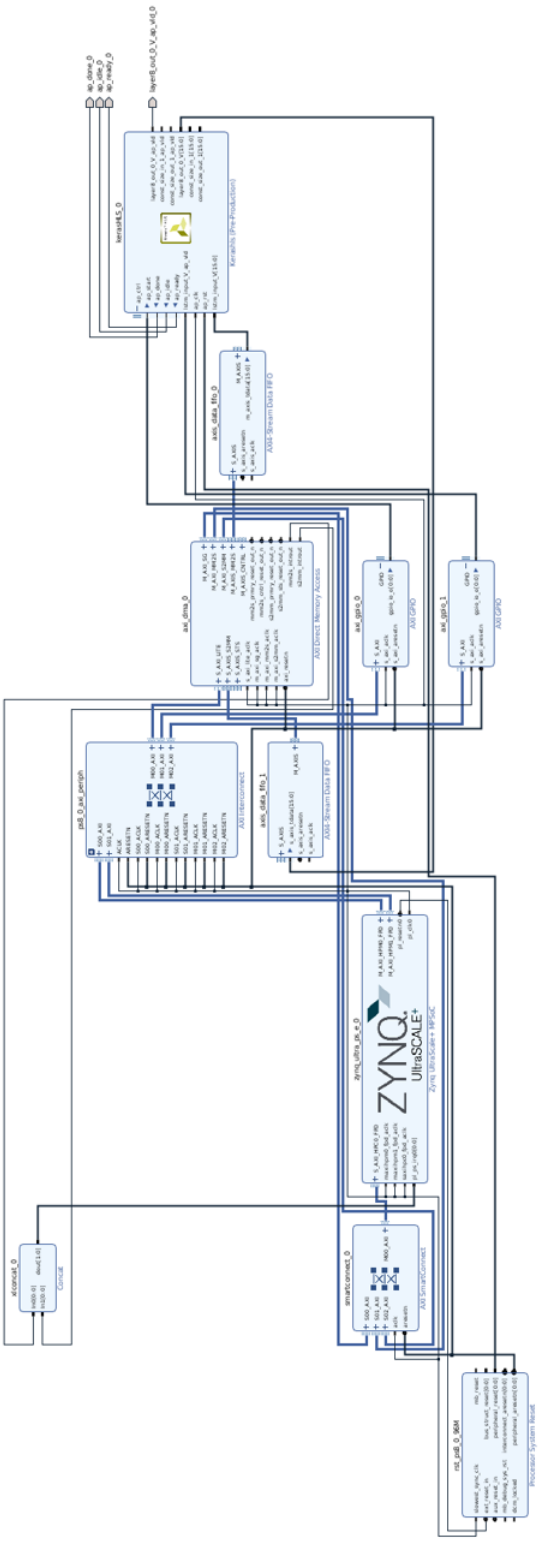


Figure Q.1 Hardware design based on AXI DMA

R | AXI GPIOs Based Block Design

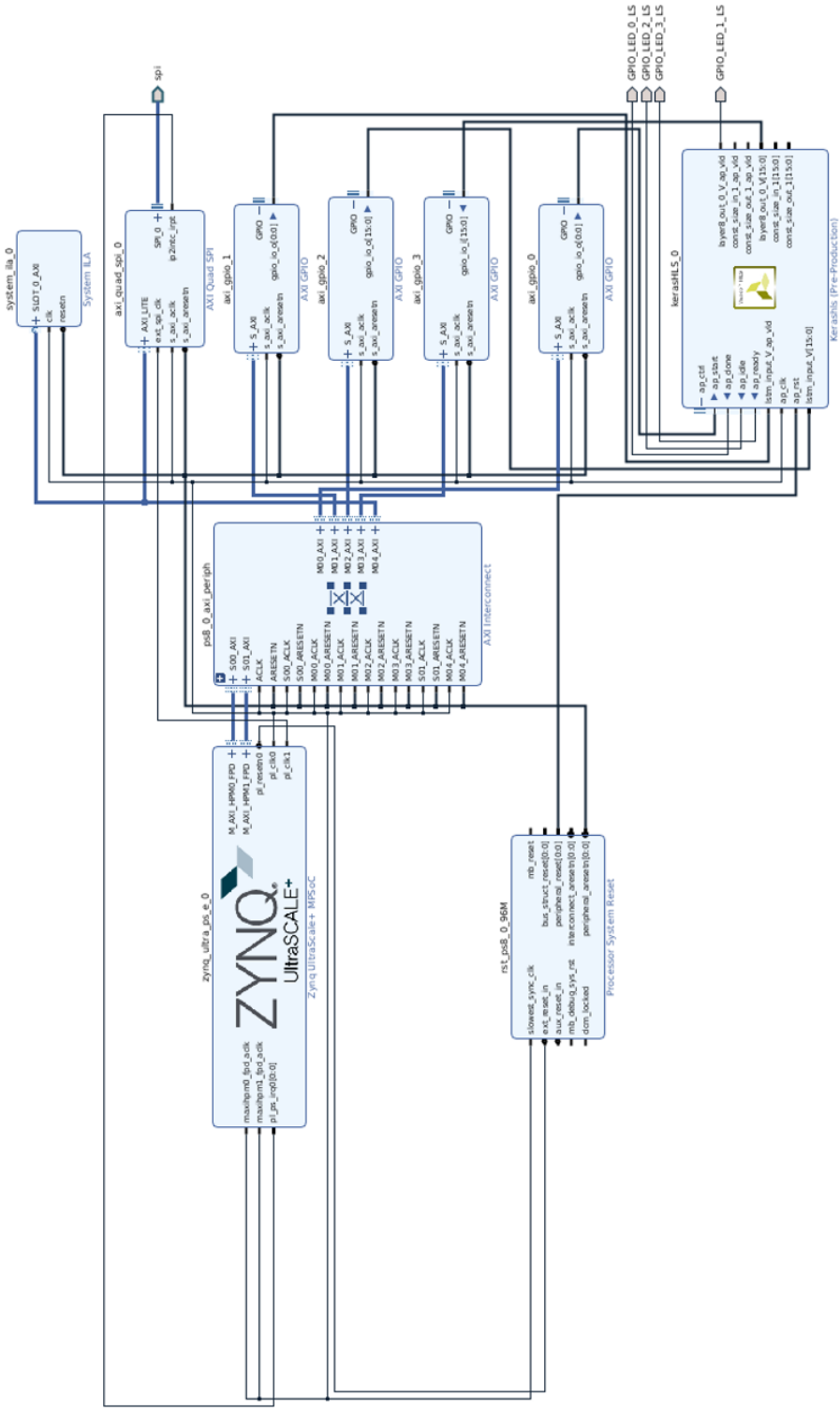


Figure R.1 Hardware design based on AXI GPIOs