POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Design and optimization of an FPGA-based system for real-time human stress level assessment

Tesi di Laurea Magistrale in
Electronics Engineering - Ingegneria Elettronica

Authors: **Nicolò Campanini, Salvatore Torsello**

Student ID: 963168, 944861
Advisor: Prof. Christian Pilato
Academic Year: 2022-23

Abstract

In modern society, humans are completely surrounded by a frenetic and constantly changing world and individuals are subject to a collection of situations that can converge into stressful conditions. This natural response of the body to external stimuli can cause the onset of serious health issues or impair mental clarity.

The focus of this thesis is to build an FPGA-based embedded system for real-time stress level estimation that could be easily widened and improved in the future, thus becoming a development platform for further work in the field of affective computing.

The starting point of the system development was the analysis of stress-related physiological signals by extracting them from datasets collected from literature. The electrocardiogram (ECG) and the electrodermal activity (EDA) were chosen as signals on which to build a high-level model of the problem and develop a processing and estimation algorithm based on Python. After a performance comparison, a three level classifier (relaxed, normal and stressed) based on the XGBoost framework was trained on the data extracted from the algorithm. The algorithm and the classifier were then implemented on an embedded platform. Specifically, the choice fell on the Zynq™ 7000 SoC, consisting of an FPGA and a dual-core CPU, so that a heterogeneous system including both hardware accelerator and Linux application could be designed. The hardware accelerator, developed in high-level synthesis (HLS), is dedicated to the processing of the signals and the extraction of more significant characteristics related to stress conditions. The software application, running on Petalinux, deals with feature extraction from the data produces by the accelerator and performs classification using the chosen model. The work was developed using AMD Xilinx EDA tools.

The validation process of the developed system was focused on comparing the implementation with the simulation model, so that differences in term of timing and classification accuracy can emerge. The heterogeneous platform obtained satisfy the previous requirements in terms of modularity, allowing for further development and integration into more complex monitoring systems.

Keywords: hardware accelerator, stress-estimation, heterogeneous system, FPGA, HLS, Petalinux

Abstract in lingua italiana

Nella società moderna, in cui l'uomo è completamente immerso in un mondo frenetico e in costante cambianto, l'individuo è soggetto ad un assieme di situazioni che possono convergere in condizioni di stress. Questa naturale risposta del corpo agli stimoli esterni può causare l'insorgere di gravi problematiche di salute o compromettere la lucidità mentale.

L'obbiettivo di questa tesi è quello di realizzare un sistema embedded basato su FPGA per la stima del livello di stress in tempo reale che possa poi essere facilmente espanso e migliorato in futuro, fungendo quindi da piattaforma di sviluppo per ulteriori lavori nell'ambito dell'affective computing.

Per realizzare il sistema si è proceduto con una prima analisi dei segnali fisiologici correlati allo stress, estraendoli da dataset raccolti dalla letteratura. L'elettrocardiogramma (ECG) e l'attività elettrodermica (EDA) sono stati scelti come segnali sui quali costruire un modello ad alto livello del problema e sviluppare un algoritmo di processamento e di stima basato su Python. Dopo un confronto delle prestazioni, un classificatore a tre livelli (rilassato, normale e stressato) basato sul framework XGBoost è stato allenato sui dati estratti dall'algoritmo. Si è poi proceduto all'implementazione dell'algoritmo e del classificatore su una piattaforma embedded. In particolare la scelta è ricaduta sul SoC Zynq™ 7000, composto da una FPGA ed una CPU dual-core, così da poter realizzare un sistema eterogeneo comprensivo di un acceleratore hardware e un'applicazione Linux. L'acceleratore hardware, sviluppato in high-level synthesis (HLS), è dedicato al processamento dei segnali e alla conseguente estrazione di caratteristiche più significative, riconducibili a condizioni di stress. L'applicativo software, eseguito su Petalinux, si occupa dell'estrazione delle feature dai dati prodotti dall'acceleratore ed opera la classificazione utilizzando il modello prescelto. Il lavoro è stato sviluppato utilizzando i tool EDA di AMD Xilinx.

Il processo di validazione del sistema progettato si è focalizzato sul confronto fra la sua versione simulata e quella implementata, in modo tale da far emergere differenze a livello di timing e accuracy della classificazione. La piattaforma eterogenea ottenuta soddisfa i requisiti di modularità prefissati e si presta a sviluppi e integrazioni in dispositivi di

monitoraggio più complessi.

Parole chiave: acceleratore hardware, stima dello stress, sistema eterogeneo, FPGA, HLS, Petalinux

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 Problem description **3**

1.1 Signal choice . . . 3

1.1.1 ECG . . . 4

1.1.2 EDA . . . 6

1.2 Stress labeling . . . 7

1.3 Workflow . . . 8

2 Dataset **11**

2.1 Dataset exploration . . . 11

2.1.1 Dataset comparison . . . 12

2.1.2 Data cleaning and conversion . . . 12

3 Algorithm development **15**

3.1 Time windowing . . . 16

3.2 ECG processing chain . . . 17

3.2.1 Pan Tompkins algorithm . . . 18

3.2.2 NN and Cardiotach Extraction . . . 21

3.2.3 Periodogram Extraction . . . 23

3.2.4 NN Features . . . 24

3.2.5 Cardiotach Features . . . 27

3.3 EDA processing chain . . . 27

3.3.1 Signal normalization . . . 28

3.3.2 Signal decomposition . . . 29
3.3.3 Critical points detection . . . 30
3.3.4 Statistical features . . . 33
3.3.5 Syntactical features . . . 35
3.4 Feature set exploration . . . 36
3.4.1 Feature cleaning . . . 37
3.4.2 Feature selection . . . 37
3.5 Classifiers exploration . . . 38
3.5.1 Training Methodology . . . 38
3.5.2 Comparison metrics . . . 39
3.5.3 Tested Classifiers . . . 40
3.5.4 Selected Classifier . . . 41

4 Proposed solution 43
4.1 Platform choice . . . 44
4.1.1 EDA tools . . . 45
4.1.2 Development board . . . 46
4.2 Hardware / Software co-design . . . 48
4.2.1 ADC Emulation . . . 49
4.2.2 Interfaces . . . 50
4.2.3 System Memory Map . . . 53

5 Accelerator design 57
5.1 Downsampling . . . 58
5.1.1 ECG Downsampling . . . 62
5.1.2 EDA Downsampling . . . 65
5.2 Preprocessing . . . 68
5.2.1 ECG Preprocessing . . . 69
5.2.2 EDA Preprocessing . . . 76
5.3 Data collector . . . 83
5.4 Processing . . . 84
5.4.1 ECG Processing . . . 85
5.4.2 EDA Processing . . . 88
5.5 Interrupt generation . . . 91
5.6 HLS design workflow . . . 92
5.6.1 Interrupt validation . . . 93
5.7 Implementation . . . 94

6 Application design **97**
6.1 Peripheral drivers . . . 98
6.1.1 Biosignal Coprocessor driver . . . 98
6.1.2 User-space I/O driver . . . 99
6.2 Memory access . . . 99
6.3 Multithread application . . . 100
6.3.1 ADC Emulation thread . . . 102
6.3.2 Biosignal Coprocessor thread . . . 102
6.3.3 Features extractor thread . . . 103
6.3.4 Classifier thread . . . 105
6.4 Testing methodology . . . 106

7 Results **107**
7.1 Resource occupation . . . 108
7.2 Timing results . . . 109
7.3 Classification results . . . 110

8 Conclusions and future developments **113**
8.1 Future developments . . . 114

Bibliography **117**

A Appendix A **121**
A.1 DRIVEDB dataset . . . 121
A.1.1 DRIVEDB Sensors . . . 121
A.1.2 DRIVEDB Stress labeling . . . 122
A.2 WESAD dataset . . . 122
A.2.1 WESAD Sensors . . . 123
A.2.2 WESAD Stress labeling . . . 124
A.3 SWELL-KW dataset . . . 125
A.3.1 SWELL-KW Sensors . . . 125
A.3.2 SWELL-KW Stress labeling . . . 126

B Appendix B **129**

C Appendix C **131**

D Appendix D **135**

List of Figures **137**

List of Tables **141**

Introduction

In recent decades, especially in last years due to the COVID-19 pandemic, everyday human life is constantly surrounded by a dynamic world, always connected and full of social media. The individual mental condition is indeed exposed to quickly changing situations, which could lead mostly to unstable mental conditions. For those reasons stress monitoring is an actual research activity which covers different fields as medicine, psychology, but also engineering.

In the human being, as well as for animals but with lower intensity, the basic physiological mechanisms which deals with the activation of the stress could be of two different types: nervous and endocrine. The first, which involves the sympathetic (autonomous) and parasympathetic nervous system, consists in the activation by brain of the nervous pathways to generate a commonly response called *fight or flight* reaction. This prepares the individual to cope with an immediate threat. The second one is the neuroendocrine response which consists in the activation of a cerebro-somatic circuit called Hypothalamic-Pituitary-Adrenal (HPA) axis.

There are multiple causes of stress in human life covering individual situations as the presence of internal conflicts, financial instability, frustration and inadequacy sense at the workplace or even worst the absence of an occupation, but also global critical situations as violence and crimes in society [1]. All these elements cover very different aspects of a human being's life and they may cause critical conditions as mental disorders, depression symptoms and clinical problems. Not all stressful situations result in negative consequences. In some special cases the individual could increase his productivity and performance, for instance during competitive and challenging events.

A deeper investigation is therefore necessary to give the chance to predict unpleasant consequences. From the engineering point of view is challenging and subject of research the measuring of how stress is expressed in a more physiological sense and then construct models to predict its presence and intensity.

The main focus of this thesis work is the design and development of a system, in both hardware and software, which processes physiological human signals and give a real-time

estimation of stress level.

The first chapter give a detailed analysis of the system requirements, what are the frequently used measures of stress and which of them were used in this work. Then the available free to use databases will be introduced, with the associated stimulus adopted during the experiments. In Chapter 3 an high-level processing chain for the stress assessment problem is obtained by developing a Python pipeline, and after that, in Chapter 4 to Chapter 6, an embedded-oriented implementation of the system will be designed and explained. The proposed solution was approached with a hardware-software co-design manner, starting from drafting of requirements, the development of the physiological signal processing core, its optimization, to the development of a Linux application dealing with features extraction and classification functions. During the design, particular attention was given in the development of a modular system which could be part of more complex and portable devices, leaving the chance to be widened for future improvements.

1 | Problem description

Two main objectives have been chosen for this thesis:

- The development of a system able to estimate the stress level of a person in real-time through the use of minimally invasive sensors and an embedded hardware.
- The development of the stress estimation system as a platform which can be easily expanded and improved in the future.

We targeted a real-time estimation of the stress level, this was chosen because a real-time estimation allow for more application of the system in the real world when compared to an offline stress estimation.
A real-time stress estimation system can be employed in different fields like health monitoring or high-risk jobs, where having an information about the current stress level of the individual can be helpful in detecting the onset of stress-related illnesses in subject at risk or in increasing work safety.
Moreover, even for normal people, all the benefits of keeping the stress level under control still apply and therefore a real-time stress estimation system could be helpful to improve everyday life.

The development of the system as a platform, more than a specific system, was required to allow for future development and to allow additional parts to be added. Due to the embedded approach taken, the system architecture could become the starting point for the creation of more complex wearable system which processes different physiological signals and estimate health-related parameters of the individual.
A platform like this would have a lot of applications in the affective-computing industry as well as in the medical and healthcare industries.

1.1. Signal choice

Since stress produces different changes in the human physiology, different parameters of the body are influenced by it. Some of those can be measured and used to estimate the level of stress a person is subjected to. The most important physiological characteristics

which are influenced by stress and can be measured with sensors are the following:

- **Cardiac rhythm** (ECG): The heart-rate of a person is influenced by the stress level in its speed and variability.
- **Galvanic skin response** (EDA): The conductance of the skin is related to the activity of the sweat glands, the behaviour of which is altered by changes in the stress level.
- **Brain activity** (EEG): The activity in different parts of the brain and its patterns changes when the individual is subjected to stress.
- **Muscular activity** (EMG): The activity of different muscle groups can be linked to the stress level a person is experiencing.
- **Body temperature**: Changes in the stress level alter the normal behaviour of the body thermoregulatory system and produce alterations that can be measured.
- **Respiration rate**: The respiratory rhythm of a person is increased in the presence of stress due to the *fight of flight* reaction.
- **Blood pressure**: As explained before the stress level is influencing the heart-rate of an individual. The blood pressure is also altered as part of the *fight of flight* response to stress.

For this work of thesis the signal choice was mainly influenced by the requirements of low intrusiveness of the sensors, ease of use and low price. Therefore we choose to use the cardiac signal (**ECG**) and the galvanic skin response (**EDA**) in our work because they satisfy the requirements and a lot of studies have been conducted on their analysis.
Additional sensors could be used and added to the setup as part of future expansions of our work.

The two signals and their measurement methods are now introduced in details.

1.1.1. ECG

The electrocardiogram (ECG) is a signal describing the electrical activity of the heart over a period of time. It is a voltage signal obtained using multiple electrodes placed on the patient skin to detect the small electrical changes that are a consequence of cardiac muscle depolarization followed by re-polarization during each cardiac cycle (heartbeat).
Interpretation of the ECG is about understanding the electrical conduction system of the heart. Normal conduction starts and propagates in a predictable pattern, and deviation

from this pattern can be a normal variation or be pathological.
The typical ECG pattern of an healthy individual is shown in Figure 1.1.

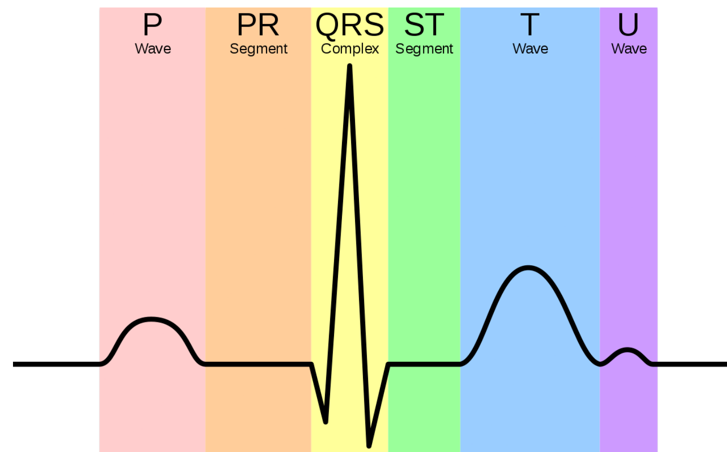


Figure 1.1: ECG signal pattern

Different parts of this pattern have been linked to specific event in the heart cycle, in particular:

- **P** wave: atrial depolarization.
- **QRS** complex: ventricular depolarization.
- **T** wave: Ventricular re-polarization.
- **U** wave: Papillary muscle re-polarization.

The standard ECG measuring setup is comprised of 10 electrodes used to measure 12 potential differences on the body of the subject. This is called a 12 *lead* ECG, here the term *lead* refers to the potential difference between two electrodes.
The standard ECG measurement setup is used to monitor different pathological conditions as well as estimate heart health, simpler setups based on just two or three electrodes can be used to measure the ECG signal when a complete medical-grade analysis is not necessary.

From an electronic point of view the ECG signal is measured using an high impedance differential measurement setup. The signal has a bandwidth of interest in the region $0.5Hz$ to $50Hz$, a voltage range of $0.5mV$ to $5.0mV$ and its usually accompanied by large DC components which act as a baseline.
The measurement setup should include different filtering (both analog and digital) to remove all the unwanted noise components from the signal. The most important sources of noise are the following:

- $50Hz - 60Hz$ power-line interferences.

- Electrode contact noise causing baseline drift.
- Muscle contraction noise.
- Respiration process causing baseline drift.
- EMI caused by electronic devices.

For stress identification and estimation the main ECG parameter of interest is HRV **Heart Rate Variability** [3]. HRV is referring to the beat-to-beat variation of the heart rate which are deeply influenced by the stress level.
The HRV analysis can be performed in different domains [26], in this thesis we explored the time, frequency and non-linear domain of HRV and extracted useful features in all of them.
More information will be provided in Section 3.2.

1.1.2. EDA

First studies about Electrodermal Activity go back up to the half of the nineteenth century when different neurologists began to make laboratory experiments measuring it as a clinical diagnostic sign and discover its link with human psycho-physiological phenomena [8]. In addition, in those years, a lot of electronics devices were built exploiting this interesting signal, such as the E-meter, covering a wide range of applications.

For Electrodermal Activity, also known as Galvanic Skin Response (GSR), is intended the capacity of the human skin to conduct an electrical current in response to different types of stimuli. In response to them, the sympathetic branch of the autonomic nervous system triggers the activity of eccrine sweat glands, located in the hypodermis of the skin, and sweat gland duct (located in the dermis and epidermis) gets filled up and due to the presence of ions in sweat the skin conductance increases as swell. Figure 1.2 describes the skin layers in detail.

There are a lot of body location for EDA measurements and the right choice depends on the specific application. For example a commonly used location is the wrist, but the presence of low density eccrine sweat glands doesn't ensure a reliable and stable measurement. Nonetheless, it is largely used since the wrist represents a position which doesn't interfere with daily activities, becoming equivalent to a common wearable device. The are instead other locations of the body where the density of the sweat gland is higher, such as hands, foots and chest, but the subject under observation may be hindered during the measurement. Since the nature of this characteristics to responds to external stimuli, sweating could be evoked by both emotional states/arousal and temperature

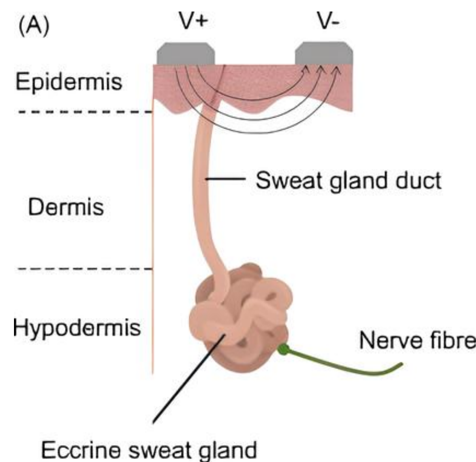


Figure 1.2: Skin layers

control mechanisms.

The measure of the electrodermal activity by the skin conductance level is commonly achieved through the exosomatic technique, which consists into the application of a weak electrical current or a low voltage source (which could be DC or AC) across two electrodes places next to each other on the skin. Then by measuring respectively the voltage or current the output of the readout circuitry can be mapped into a conductance value. Nominally, the measured conductance range from $0\mu S$ and $30\mu S$.

The EDA signal presents its spectral components in a very low frequency range, from quasi-DC component up to 2 Hz components and for this reason a relatively low sampling frequency is sufficient to analyze it. To make a better analysis it is also useful to separate the signal into two parts, as described also in Figure 1.3:

- **Skin Conductance Level** (SCL), also called *tonic*, which corresponds to slow spontaneous electrical fluctuations of sweat gland activity as a reaction to skin temperature and hydration
- **Skin Conductance Responses** (SCRs), also referred to as *phasic*, which instead deals with rapid and smooth transients responses to external events.

1.2. Stress labeling

Stress can be considered a variable whose intensity varies in a continuous and subjective way. In our work we choose to develop a system able to distinguish the state of the individual between three different levels:

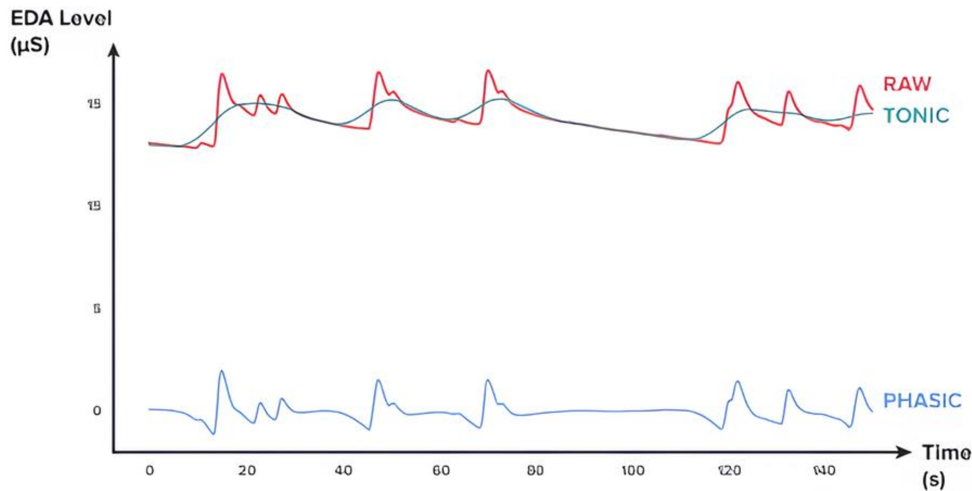


Figure 1.3: EDA components

- **Relaxed**: This state is connected to moments when the individual is not performing any task and is not subjected to any mental workload. It can be generated when an individual is performing meditation or similar activities.
- **Normal**: Connected to moments when the person is performing some tasks requiring a medium mental workload. This can be considered the ideal working condition.
- **Stressed**: This state is linked to moments when the individual is performing some tasks under the influence of an external pressure, this triggers the *fight or flight* response and create an increase in the stress level. An example of this state can be a task requiring an extreme mental effort or a task to be executed under a strict time pressure.

We choose to develop our system as a stress level classifier, therefore the estimated stress level is not provided as a continuous variable but as a discrete variable with the three level shown above.

1.3. Workflow

The development process was divided into two main parts: an high-level part and a low-level part.

- **High-level**: In this part a signal processing algorithm was developed using Python. This allowed us to quickly analyze the data, test different data processing algorithm and train a classifier able to estimate the variable of interest from the data.
  The different steps taken in this part are the following:

  - Research and analysis of available experimental datasets.
  - Dataset selection, cleaning and conversion to a common data format.
  - Design of a data processing pipeline for feature extraction from the data.
  - Design of a classifier training pipeline to test different classifiers on the extracted features.
  - Training of different classifiers and selection of the best performing one.

- **Low-level**: Here, after the development of the Python algorithm, the embedded implementation was created and the obtained results were validated.
  The following steps were performed:
  - Selection of a suitable embedded platform to use.
  - Hardware-Software subdivision of the Python algorithm.
  - Design of the Hardware-Software blocks.
  - Testing and validation of the obtained system.

2 | Dataset

To train a machine learning model able to recognize and classify patterns, experimental data are required. For our specific task we were interested in recognizing the level of stress of a person, therefore experimental data related to human stress were needed.

Ideally, a specific experiment would have to be engineered, a test setup prepared and test subject hired for it. Due to the constraints of our work of thesis this was not feasible, therefore experimental data had to be acquired from public resources.

2.1. Dataset exploration

For our work an experimental dataset containing data for both ECG, EDA and the stress level had to be found. A reasonable stressor had to be used and a good amount of data has to be present for the different stress levels.
Three possibile candidate datasets were identified:

- **DRIVEDB** [11] - Explained in details in Appendix A.1:
  Multi-parameter recordings from healthy volunteers driving on a prescribed route including city streets and highways in and around Boston, Massachusetts.
  The stressor used is a driving task in different road conditions, producing different levels of stress.
- **WESAD** [29] - Explained in details in Appendix A.2:
  A lab study with physiological and motion data, recorded from both a wrist- and a chest-worn device, of 15 subjects.
  Different kind of stressors as well as a relaxation stimulus were used.
- **SWELL-KW** [17] - Explained in details in Appendix A.3:
  Lab study consisting of 25 subjects doing typical office work (writing reports, making presentations, reading e-mail and searching for information), receiving unexpected emails interruptions and pressure to complete their work on time. The stressors used are related to office work.

2.1.1. Dataset comparison

The three dataset were thoroughly analyzed and compared to understand their possible application in our work, a synthesis is shown in Table 2.1.

Dataset	ECG freq	EDA freq	Stress levels	Subjects #
DRIVEDB	$496Hz$	$15.5Hz$	Relaxed/Normal/Stressed	17
WESAD	$700Hz$	$700Hz/4Hz$	Relaxed/Normal/Stressed	17
SWELL-KW	2048	$2048Hz$	Normal/Stressed	25

Table 2.1: Available datasets

The three datasets are different in terms of sampling frequencies, stress levels applied and kind of stressor used. Furthermore the acquisition system used and the sensor placement differ from dataset to dataset.

As can be seen the SWELL-KW dataset differs from the other two in terms of Stress Levels applied. Only two stress levels are present and the *Stressed* level is applied for almost all the time. This is not compatible with our use case where a almost equal representation of the stress levels is needed.
The other two datasets offer a more equal representation of the different stress levels as well as three stress levels, therefore we decided to discard the SWELL-KW dataset and use the other two (DRIVEDB and WESAD) for the continuation of our work.

The basic assumption we did in this phase was to consider the stress intensity produced by the two different experiment as equal. As shown in details in Appendix A the two datasets are using different stressors (driving task vs office work) which for sure produce a different stress level in different people; nevertheless we assumed the stress levels to be comparable, in this way we could merge the two different datasets and produce a bigger dataset on which a more generalized classifier can be trained.

2.1.2. Data cleaning and conversion

Due to the differences between the used datasets and the presence of corrupted data, a cleaning and conversion step was required. This step was needed to produce a clean merged datasets which can be used to experiment with our stress detection algorithm.
Since data are coming from different dataset which have used different sensor, different sampling frequencies and different file formats, we decided that a common file format was

needed to proceed.

To apply the cleaning and conversion steps a specific pipeline was created in Python, the two steps are described in details:

Data Cleaning The different acquisition files of each dataset are loaded and checked for signal artifacts and corruption. The acquisitions with corrupted data are discarded from the dataset.
Different kind of artifacts were found in the signals, varying from artifact caused by electrode disconnection to completely missing signals in acquisition files due to malfunction of the acquisition setup.
In this step we discarded 2 subjects from WESAD dataset and 4 subject from DRIVEDB dataset.

Data Conversion To simplify the data processing in later stages of our work, we decided that a common file format was needed for the data. The target was creating a *.csv* file of the common file format for each subject of each one of the chosen datasets.
The specific file format had to include the ECG signal, all the available EDA signals, a timestamp and a stress label.
The sensor signals were re-sampled at 500Hz using the Scipy Python function *resample_poly* so that they can be treated later at the same manner even if the original frequency was different. 500Hz was chosen as a frequency due to being high enough to not interfere with the filtering operations that will be performed by our algorithm while still keeping the data size manageable.
The chosen labelling is a three stage one (Relaxed, Normal, Stressed), the original labels used in the datasets were remapped to those three levels and the unknown regions of the data were flagged as unuseful. More details about label re-mapping for each dataset are given in Appendix A.

Once the cleaning and conversion pipeline was completed we were left with the data of 15 subjects from WESAD and 13 subjects from DRIVEDB, those data have now all the required signals and are the one on which our algorithm was developed.

3 | Algorithm development

The algorithm was fully developed in the Python programming language with the help of Jupyter notebook [13].

The Python language was chosen due to its versatility and ease of use as well as the wide adoption of it in the machine learning industry. Since all the most important machine learning frameworks are based on Python, this made experimenting with different libraries and framework an easy task.
The choice of Jupyter notebook as the development environment allowed us to work on the algorithm in an interactive fashion testing and tweaking the code easily thus speeding-up the development process.

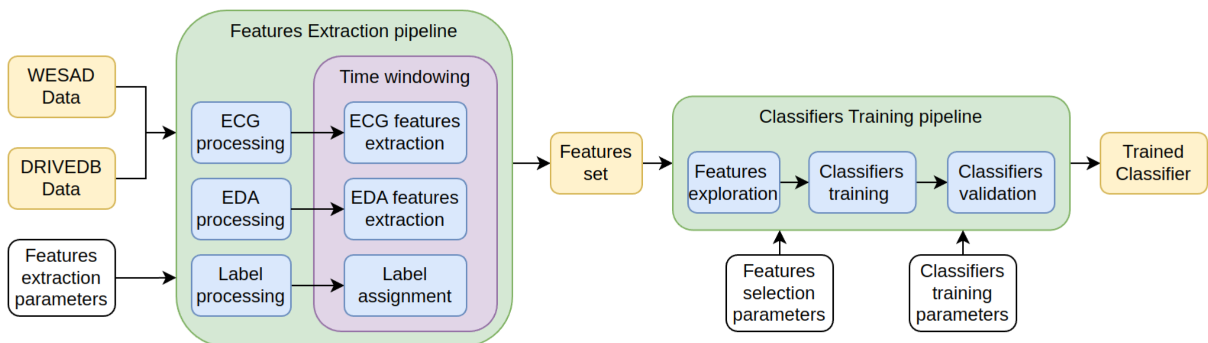


Figure 3.1: Overall structure

As seen in Figure 3.1 the structure is composed of two pipelines.
Each pipeline is based around a single Jupyter notebook which can be configured externally with the use of *papermill*[25], a Jupyter notebook tool.
For example, the *Features Extraction pipeline* can be configured to run only on some specific subjects instead of the two whole datasets while the *Classifier Training pipeline* can be configured to try only a specific classifier instead of all the classifiers it has access to.
The parametrization introduced by the use of *papermill* allowed us to speed-up the development and the exploration of different solutions.

Features extraction pipeline The target of the Features extraction pipeline is the creation of a feature-set on which classifiers can be trained. The pipeline accept as input data from the two selected datasets (WESAD and DRIVEDB) stored in the common file format, process them and apply the correct time window so that the required features could be extracted.
The time window applied can be configure through the use of *papermill* but has been fixed at the end to a duration of $60s$ and a shift every $1s$ therefore producing useful features every $1s$. More information regarding the choice of the time window are provided in Section 3.1.
Three separate data paths are present in the pipeline: ECG, EDA and Label; the ECG and EDA paths will be explained in details in Section 3.2 and Section 3.3.
The label processing path will look at the stress label present in the input data and discard the windows on which the label is not constant. In this way we avoid to store features created where a label change is happening and obtain a more reliable features set. As explained before in Section 1.2, a 3-level label was chosen which are: Relaxed, Normal and Stressed.
The output features set is saved as a *.csv* file which can be later used by the Classifier Training pipeline.

Classifier training pipeline The Classifier training pipeline allows the training of different classifier types on the features set. The chosen classifiers as well as the features selection algorithm used and the number of features to retain can be configured through *papermill*. All those configurations are passed as lists and the pipeline will iteratively try them.
Once started and configured the pipeline will read the input features set and run through all the possible permutations of the configuration lists. The classifiers are trained using the *scikit-learn* framework and the performance are evaluated using 10-fold cross-validation. The pipeline will save all the trained models as *.pikle* files for later use and provide as output a *.csv* file with the computed performance of all the classifiers. Classifiers performance evaluation is explained in more details in Section 3.5.

3.1. Time windowing

In order to analyze the physiological signals in real-time, a signal windowing method was used. Data are collected and processed for the required window size, then from those finite and constant amount of samples the necessary features are computed.
In this work, the Features extraction pipeline was exploited as a way to compare different

window sizes and then choose the best segmentation comparing all the results. A lower and upper limit was anyway fixed for several reasons, respectively to 20 and 60 seconds, and the tests were conducted with step size of 5 seconds.

Short time window The focus of the study is to estimate the emotional state from physiological signals that naturally have very low bandwidth. This measure is derived from several significant characteristics which are time based but also frequency based. Wrong assumptions could be made when trying to detect low spectral components of around $0.05Hz$ in a small time window. For this reason a lower bound equal to $20s$ is introduced.

Long time window On the opposite, a long time window will surely be helpful while analyzing low frequency components, but the amount of data to be stored and processed would inevitably increase, leading to an increase of resources utilization and time required for computation. As the focus of the work is to implement a real-time estimation, an upper bound is introduced at $60s$.

The sampling window is shifted at intervals of $1s$, this was chosen to produce a $1Hz$ output signal which was deemed sufficient for our real-time requirements.

3.2. ECG processing chain

The objective of the ECG processing chain is the extraction of signals which are highly correlated with the stress level from the cardiac signal.
The main focus is the analysis of Heart Rate Variability (**HRV**). HRV is the physiological variation of the time interval between heartbeats and has been shown to be highly correlated with emotional arousal as well as mental workload.
The ECG processing chain is composed of different blocks as shown in Figure 3.2.

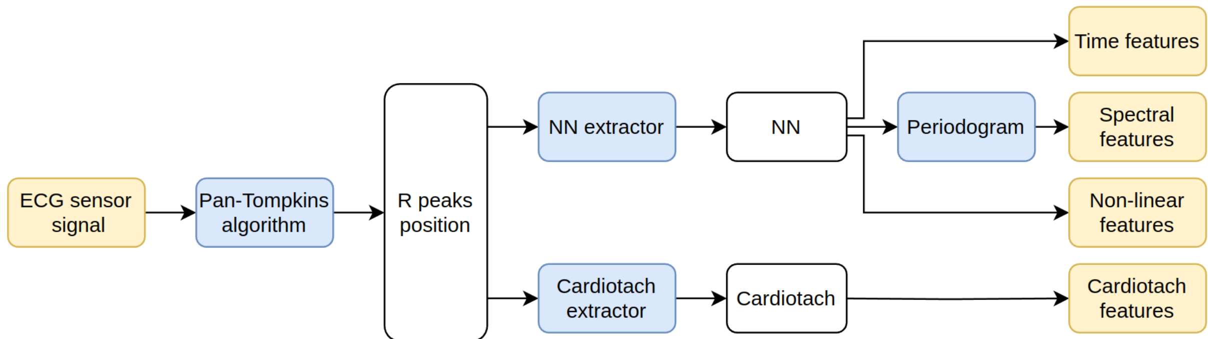


Figure 3.2: Overall ECG processing Chain

The input voltage read from the sensor is initially processed to extract the positions of the heartbeats. This processing is done using the Pan-Tompkins algorithm [24] and will be explained in detail in Section 3.2.1.
From those positions two signals are extracted: NN and Cardiotach; those two signals contain information regarding heart-rate evolution between different beats and in time. The NN signal is further processed to extract its periodogram to explore its spectral content.
From the three meaningful signals obtained 27 features can be extracted providing information on HRV in the time, frequency and non-linear domains.

3.2.1. Pan Tompkins algorithm

The ECG signal is composed of different waves as shown in Figure 3.3.

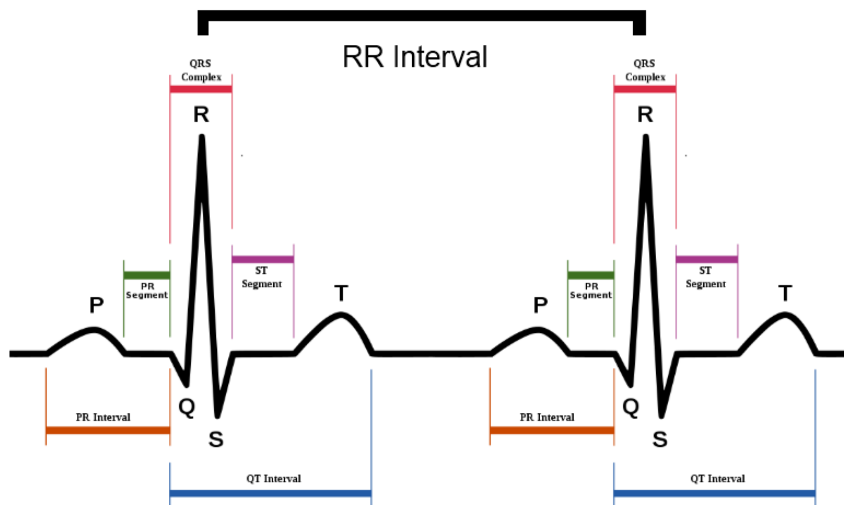


Figure 3.3: QRS complex in ECG signal

We are interested in extracting the heartbeat position from our ECG signal, this translate into locating the peak of the signal. This peak is located at the center of the QRS complex which represents ventricular depolarization of the heart and is composed of three different waves (Q, R and S).
An algorithm to locate the QRS complex and the R peak in particular was thus needed, we have chosen to use the Pan-Tompkins algorithm [24].
This specific algorithm was firstly introduced in 1985 and is one of the most used real-time QRS detection algorithm available due to its low computational requirements and high noise immunity. Those features make it a perfect candidate to be used in our processing chain firstly in Python and later in the target embedded system.

The algorithm is divided into two three blocks: a linear filtering chain, a non-linear transformation and a decision stage. Figure 3.4 shows the algorithm block scheme.

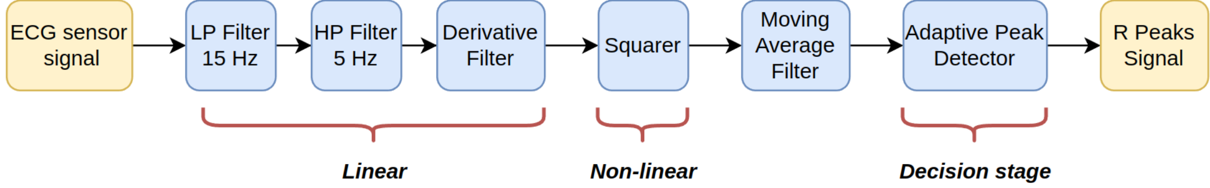
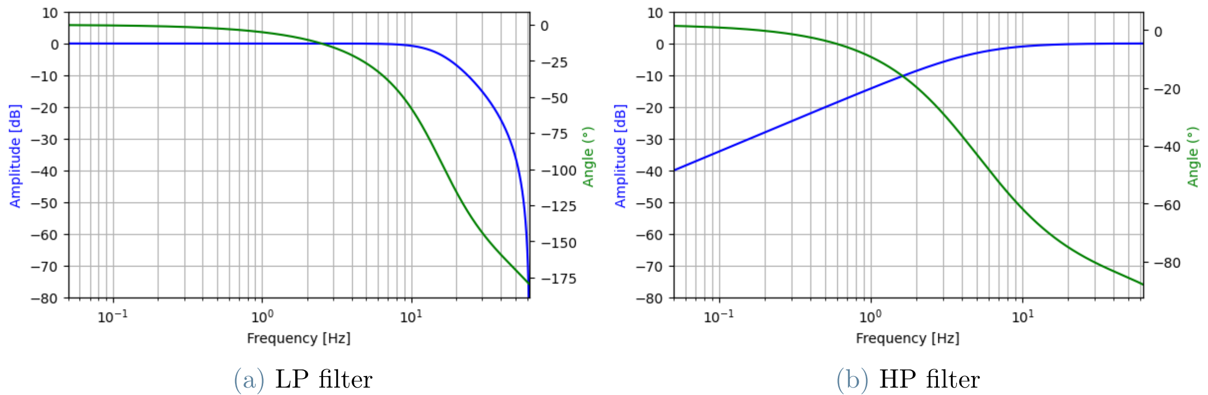


Figure 3.4: Pan Tompkins algorithm

Band-pass filter The first stage of the algorithm is a band-pass filter aimed at reducing noise in the signal band of interest. The noise is mainly composed of muscle noise, $50 - 60Hz$ interference, baseline wander, and T-wave interference.
The Band-pass filtering is obtained with the combination of a 15Hz second order low-pass filter followed by a 5Hz first order high-pass filter which, in the original Pan-Tompkins paper [24], were implemented using recursive filters with integer coefficients.
Due to the use of 125Hz ECG signal instead of the paper original 200Hz different recursive filters were needed in our implementation; in particular we choose to use Butterworth filters for both the low-pass and the high-pass filters leading to the transfer functions shown in Figure 3.5.



(a) LP filter (b) HP filter

Figure 3.5: LP and HP filters

Derivative filter The second stage of the algorithm is composed of a derivative filter aimed at proving the QRS complex slope information.
The original paper used a 5 point derivative providing an additional filtering at high

frequencies. Due to the difference in sampling frequency, the original derivative filter would interfere with the signal bandwidth and thus we moved to a more conventional 5-point derivative filter providing no attenuation at high frequencies.
The original derivator transfer function is shown in Equation (3.1) while the derivator used here is shown in Equation (3.2)

$$H(z) = 1/5z^2 + 1/10z - 1/10z^{-1} - 1/5z^{-2} \tag{3.1}$$
$$H(z) = -1/12z^2 + 2/3z - 2/3z^{-1} + 1/12z^{-2} \tag{3.2}$$

Squarer The third stage of the algorithm is a squaring operation, this is used to apply a nonlinear amplification to the output of the derivative, emphasizing higher frequencies and enhancing the signal we are interested in.

Moving average filter The last filter applied to the signal is a moving average filter which will perform an integration of the previously processed signal.
The moving average filter will produce at its output a trapezoidal signal at the position of the QRS complex. The window size can be finely tuned to obtain a narrower output, in our system we used a window size of 128ms (16 samples at 125Hz). This is similar to the original paper 150ms window size and lead to good performances.

Adaptive peak detector Once the integral signal has been generated we need to correctly detect the peaks present in it.
In general our signal will be composed of signal peaks (the ones we are interested in) and noise peaks (fake peaks introduced by noise). Our peak detector needs to be able to distinguish between the two.

A fixed threshold is not advised to be used as the decision criteria between the two conditions, this is because signal and noise amplitudes can change over time. To account for this, the Pan-Tompkins algorithm uses a dual-threshold technique: a threshold for signal and a threshold for noise are continuously kept in memory and updated once a new peak has been classifier as either noise of signal.
In this way the thresholds continuously adapt to the characteristics of the signal and allow the algorithm to quickly follow changes in the signals, greatly reducing detection errors.

A search-back is also implemented to avoid skipping signal peaks with low amplitude; the search-back override the signal threshold if no peaks has been found after 166 percent of the current average peak-to-peak interval therefore allowing for detection of lower amplitude signal peaks.

Figure 3.6 shows the effect of the Pan-Tompkins algorithm on the signal. The green crosses are placed where the peak is detected.

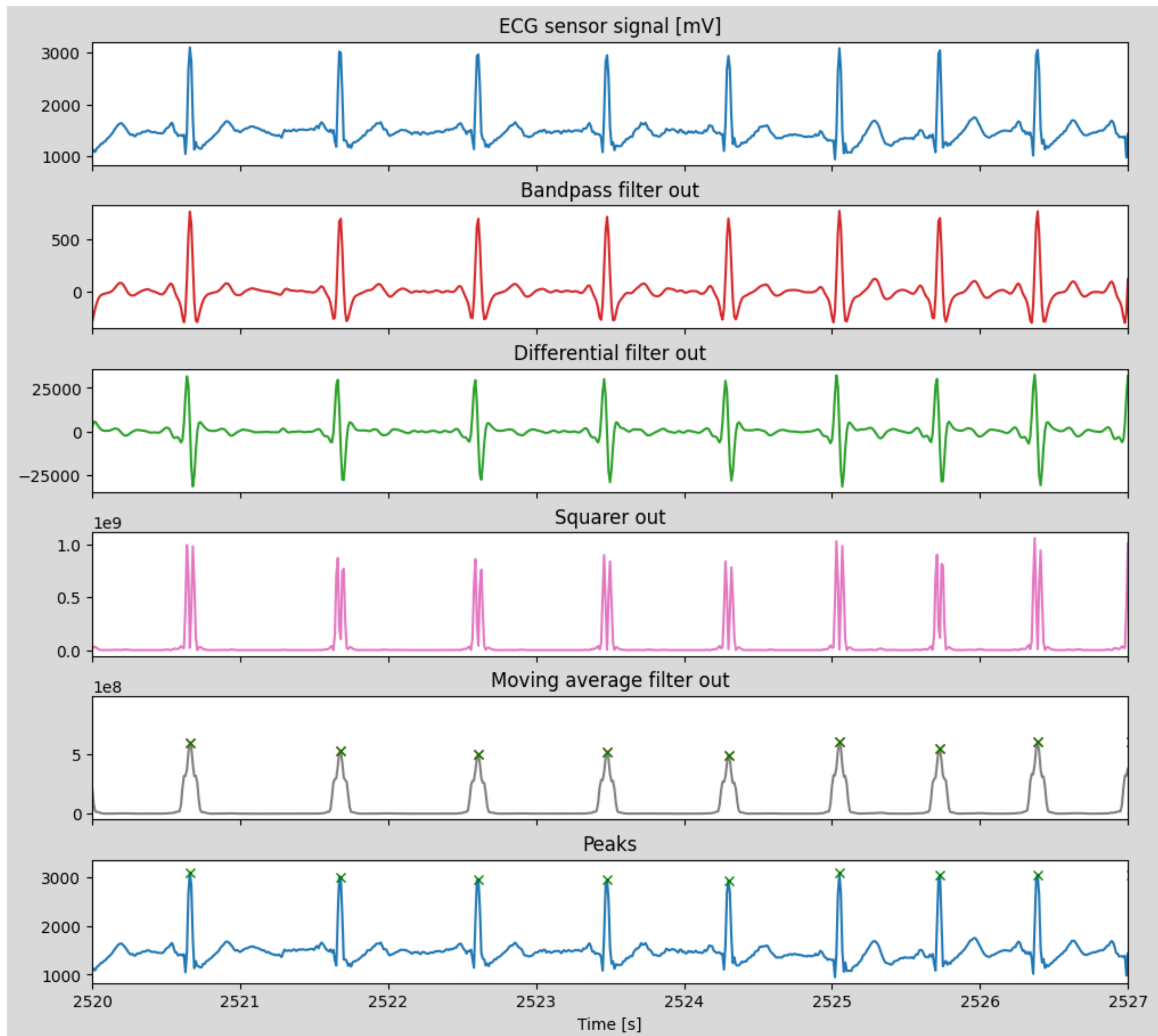


Figure 3.6: Pan Tompkins algorithm signals. Green crosses placed where peaks are detected

3.2.2. NN and Cardiotach Extraction

Once the peaks positions have been obtained the two meaningful signals (**NN** and **Cardiotach**) can be computed in the time window of interest. The two signals are now described in details.

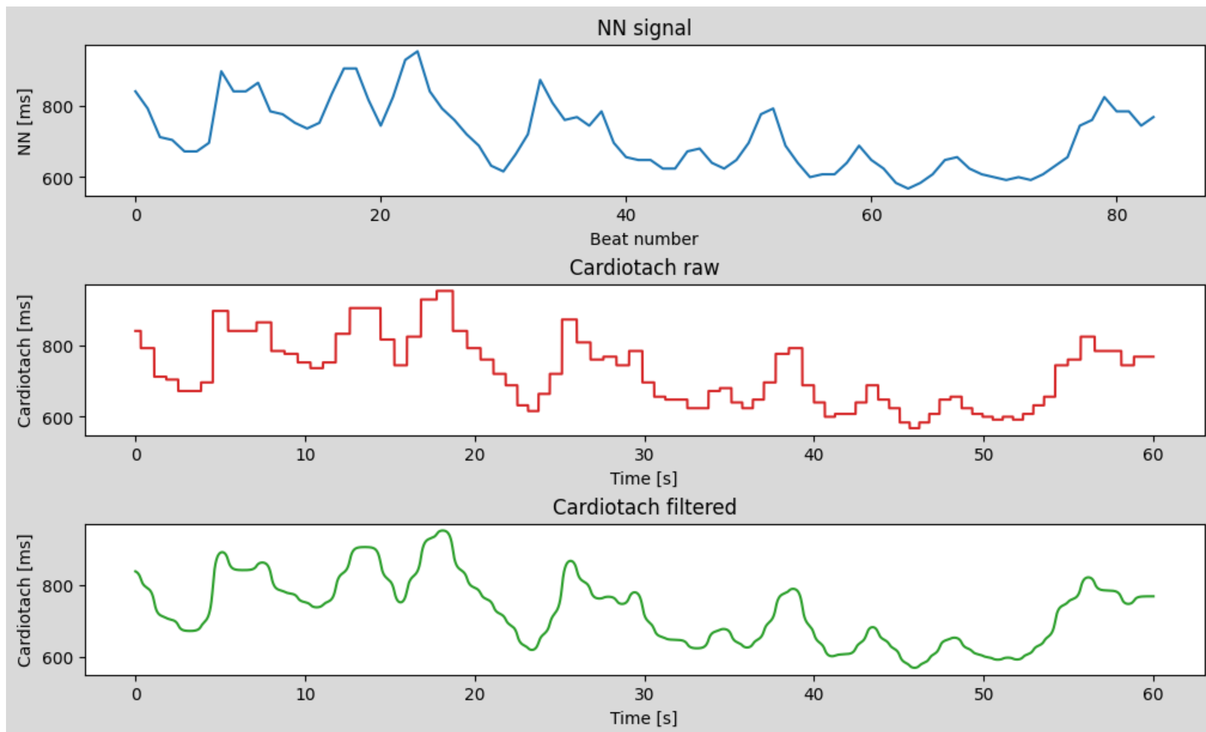


Figure 3.7: NN and Cardiotach signals

NN The NN interval signal is a series composed of the time differences between consecutive signal peaks.
The NN extractor will compute the NNs by running through the peak array, keeping track of the last found peak and calculating the new NN each time a peak is found.
The NN is expressed in ms and can vary between $300ms$ and $2000ms$ assuming a heart-rate range of $30bpm$ to $200bpm$.
An example of the NN signal can be seen in Figure 3.7.

Cardiotach The Cardiotach signal describe the evolution of the heart-rate in time.
It is expressed in ms and is created firstly as a stepped signal by assigning the corresponding NN value to the all the data points between two peaks.
The stepped Cardiotach is then run through a filter to smooth-out its steps obtaining a slowly varying signal describing the heart-rate in time. For this filtering a zero-phase digital filtering approach is used to avoid delays and transient effects in the signal.
The zero-phase filtering approach is only possible because the signal is completely known in the time window of interest. The filter used is a Butterworth filter of order 1 and cut-off frequency of $1Hz$. The zero-phase filtering is obtained by first filtering the entire signal, reversing the resulting array, filtering the resulting array and reversing again the new result.

An exampled of the raw and filtered Cardiotach signals can be seen in Figure 3.7.

3.2.3. Periodogram Extraction

To analyze HRV in the spectral domain a power spectral density (**PSD**) needs to be computed from the NN signal.
Since the heart-rate is changing and the NN signal is "sampled" every heart-beat, we can consider the NN signal as a series of unevenly sampled data. For this reason computing the spectrum is not a simple task and attention needs to be payed to the PSD estimation method used.

Historically three different approaches [2] has been used to compute the periodogram of NN series for HRV analysis:

- **Welch method** [35]
- **Lomb-Scargle method** [18, 28]
- **Burg method** [4, 6]

Welch method The Welch method [35] is the standard periodogram method, to apply this method to a series of unevenly sampled data an interpolation step is used thus obtaining an evenly sampled series from which the periodogram can be computed.
The now evenly sampled data points are divided into different overlapping segments, the periodogram for each segment is computed and the overall periodogram is obtained by averaging the single segment results.
This averaging step allow for a variance reduction in the obtained result.

Lomb-Scargle method The Lomb-Scargle method [18, 28] was originally developed for astronomical research where observations cannot be done uniformly in time.
Due to its working principle it can be directly applied to stochastic, irregularly sampled data and it reduces possible distortions or erroneous results induced by interpolation since it's working directly with the unevenly sampled data. It is based on a least-squares fit of sinusoids to the data samples and is also called Least-squares spectral analysis.

Burg method The Burg [4, 6] estimation method differ from the other two because it assume a specific functional form to model the signal and tries to fit the chosen model to the data. The model parameters are then estimated using the Burg method and the periodogram is obtained [6].

In case of HRV the NN intervals are modeled as an auto regressive process [4], and sometimes the PSD estimation method itself is also called *Autoregressive method*.

A comparison between the three PSD estimation methods for HRV has been made by Bachler [2] and has shown superior performance of the Lomb-Scargle method in terms of variance of the output results and absolute accuracy. For this reason we choose to use the Lomb-Scargle periodogram as our PSD estimation method.

In our algorithm we choose to use the function *lombscargle* from the Python library *Scipy* to compute the periodogram. The frequency range was chosen between 0.001Hz and 0.41Hz with 256 steps to provide a reasonable granularity in the spectrum of interest. In Figure 3.8 an example of PSD extracted from the NN signal is shown.

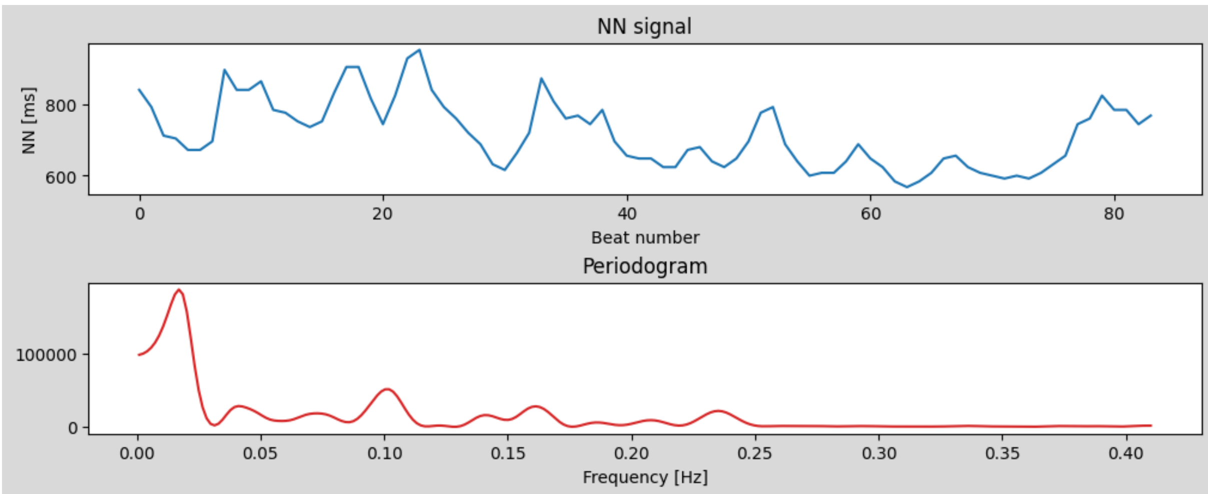


Figure 3.8: NN and Periodogram signals

3.2.4. NN Features

Once the NN signal has been extracted for the time window of interest the required features can be extracted from it.
The features are divided into three domains:

Time domain Those features are extracted from the NN signal directly and are mainly composed of statistical features.
Some of those features requires the computation of an array of successive differences from the NN signal, this is defined as:

$$\Delta D_i = NN_i - NN_{i-1} \tag{3.3}$$

	Feature	Formula
1	Average of NN values	$\frac{1}{N}\sum_{i=1}^{N} x_i$
2	Variance of NN values	$\sigma^2(NN)$
3	Standard deviation of NN values	$\sigma(NN)$
4	RMS value of successive differences in NN	$\sqrt{\frac{1}{N}\sum_{i=1}^{N-1}(\Delta D_i)^2}$
5	Standard deviation of successive differences in NN	$\sigma(\Delta D_i)$
6	Min-max range of the NN values	$\max(NN) - \min(NN)$
7	Number of successive NN differing more than $50ms$ (NN50)	–
8	The proportion of NN50 divided by total number of NN	$100 * \frac{NN50}{N}$
9	Number of successive NN differing more than $20ms$ (NN20)	–
10	The proportion of NN20 divided by total number of NN	$100 * \frac{NN20}{N}$

Table 3.1: NN time features

Spectral domain Those features are extracted from the NN periodogram signal and mainly focus on providing information regarding the distribution of power in the NN spectrum.

Conventionally the HRV spectrum is divided into three regions:

- **VLF band**: $0Hz$ to $0.04Hz$
- **LF band**: $0.04Hz$ to $0.15Hz$
- **HF band**: $0.15Hz$ to $0.4Hz$

	Feature	Formula
11	Total power of the NN spectrum	–
12	Power in the VLF band	–
13	Power in the LF band (LFP)	–
14	Power in the HF band (HFP)	–
15	Ratio between LF and HF powers	$\frac{LFP}{HFP}$
16	Normalized power in the LF band	$\frac{LFP}{LFP+HFP}$
17	Normalized power in the LF band	$\frac{HFP}{LFP+HFP}$
18	Frequency of the peak in the HF band	–

Table 3.2: NN spectral features

Non-linear domain Those features are extracted from the NN signal and explore the presence of non-linear dynamics in the heart-rate process.
The chosen method of non-linear analysis is the Poincarè plot which is a type of recurrence plot used to quantify self-similarity in processes.
Two additional temporary signals needs to be computed to perform the non-linear analysis. Those signals are reported in Equation 3.4 and Equation 3.5:

$$X1_i = NN_i - NN_{i-1} \tag{3.4}$$

$$X2_i = NN_i + NN_{i-1} \tag{3.5}$$

	Feature	Formula
19	Standard deviation of X1 values (SD1)	$\sigma(X1_i)$
20	Standard deviation of X2 values (SD2)	$\sigma(X2_i)$
21	Ratio between SD1 and SD2	$\frac{SD1}{SD2}$

Table 3.3: NN non-linear features

3.2.5. Cardiotach Features

Once the Cardiotach signal has been extracted and filtered for the time window of interest the required features can be extracted from it.
The Cardiotach features mainly focus on the statistical distribution of the Cardiotach values as well as the analysis of the derived heart-rate signal which is computed as shown in Equation 3.6.

$$HR_i = \frac{60 * 1000}{Cardiotach_i} \tag{3.6}$$

	Feature	Formula
22	Mean value of HR	$\frac{1}{N}\sum_{i=1}^{N} HR_i$
23	Variance of HR	$\sigma^2(HR_i)$
24	Standard deviation of HR	$\sigma(HR_i)$
25	$65th$ percentile of the Cardiotach signal distribution	–
26	$70th$ percentile of the Cardiotach signal distribution	–
27	$75th$ percentile of the Cardiotach signal distribution	–

Table 3.4: Cardiotach features

3.3. EDA processing chain

The ElectroDermal Activity is a measure of the skin conductance of the human body, and it has been found to be one of the most robust non-invasive physiological measures of autonomic nervous system activity [11]. The processing chain of this signal was developed in Python and it is organized as shown in Figure 3.9. The main objective of this chain is to compute and extract, from a raw standardized EDA signal (see Section 2.1.2), some characteristics that are strictly correlated with the emotional state and describe the events to which the patient is subjected. Starting from those meaningful signals, a set of 15 features are computed, divided in statistical and syntactical features.

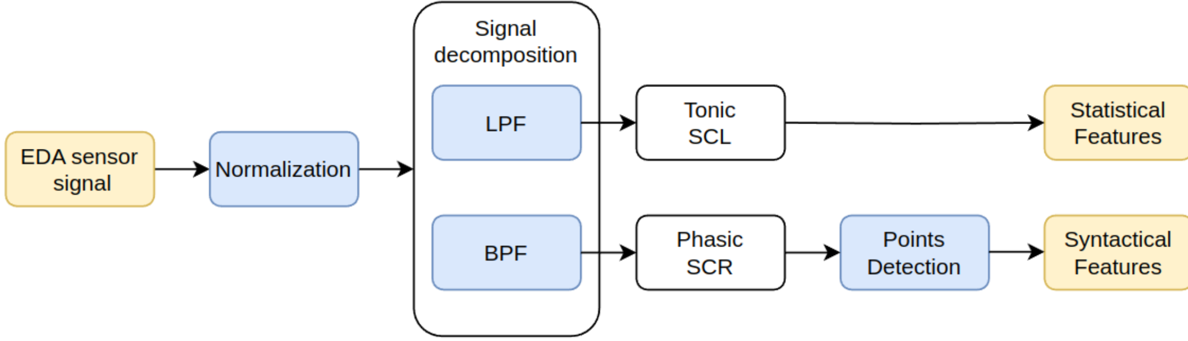


Figure 3.9: Overall EDA processing Chain

3.3.1. Signal normalization

The electrodermal activity is a very sensible signal and it is not completely correct to treat as an universal measure of the subject emotional arousal. Infact, individual-to-individual variations are very common during analysis such as the baseline level and the amplitude of the peaks.
As explained by different studies [8] [11], a normalization step is recommended in order to remove interpersonal variations. In this study two types of normalization were explored: normalization using the baseline and normalization using percentiles.

Min-Max normalization This method is one of the most common normalization techniques and it is done by subtracting from the sample the minimum between the samples acquired during the relaxed phase of the test, then the difference is normalized on the signal baseline range [9], Equation 3.7.

$$x_{norm} = \frac{x - \min X_{relaxed}}{\max X_{relaxed} - \min X_{relaxed}} \tag{3.7}$$

Percentile normalization With this technique the signal is centered at zero by subtracting the median (Q_2), and then scaling the sample by the Inter-Quantile Range (IQR), which is by the definition the difference between the 75th and the 25th percentile [9], Equation 3.8.

$$x_{norm} = \frac{x - Q_2}{IQR} \tag{3.8}$$

In Figure 3.10 a comparison between the two normalization methods is reported. In the later steps of this study only the percentile normalization was used as a normalization method, since it was discovered to be more reliable and effective on electrodermal drifts that occur naturally during the tests.

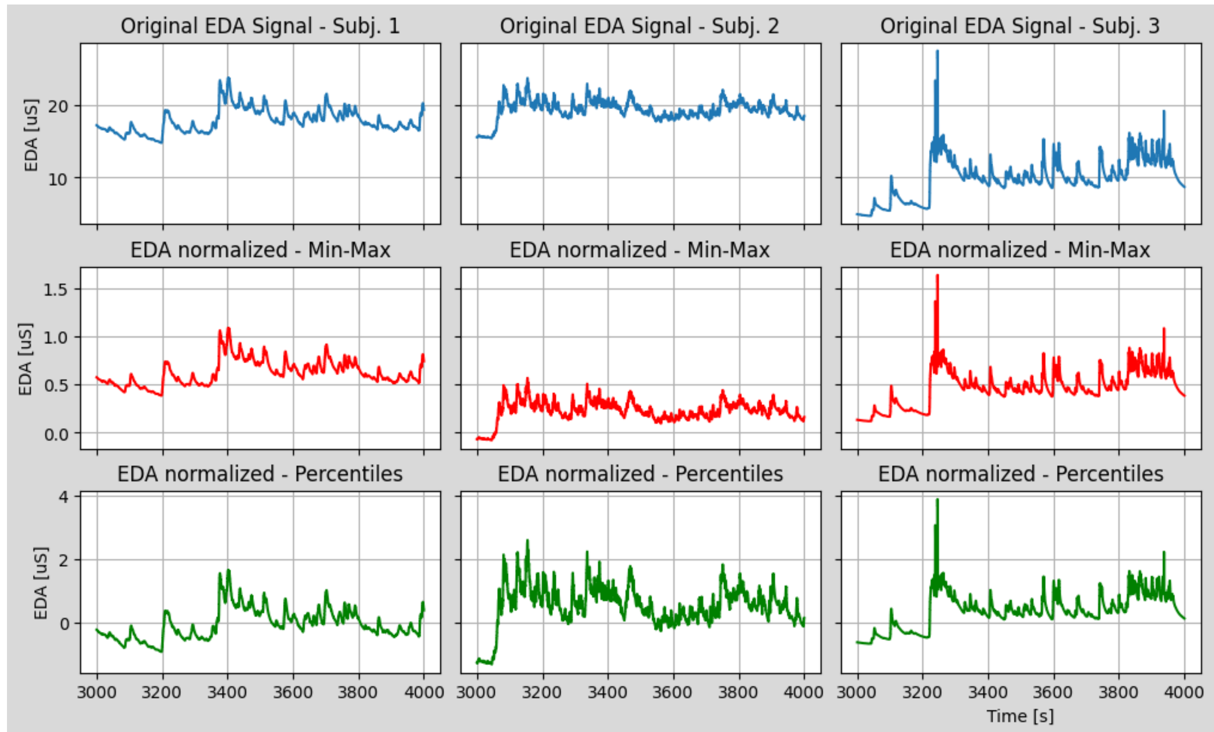


Figure 3.10: EDA normalization techniques comparison

3.3.2. Signal decomposition

It is possible to describe the EDA signal as composed of two main components, the tonic and the phasic. In order to correctly separate those two components, two parallel filters are applied, as depicted in Figure 3.11.

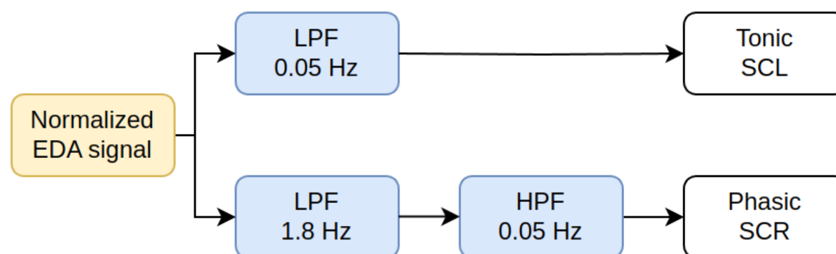


Figure 3.11: EDA signal decomposition chain

Tonic The tonic component, also called Skin Conductance Level (SCL), describes long term changes and corresponds to slow spontaneous electrical fluctuations of sweat gland activity that generate a constantly moving baseline, [5]. In order to extract those background variations a Butterworth low-pass filter of the 3rd order, designed to have a frequency response that is as flat as possible in the passband [38], is applied, with a cutoff

frequency of $0.05Hz$.

Phasic The second component, the phasic, also called Skin Conductance Response (SCR), refers to the faster changing elements of the EDA signal and represents a reactive response compared to the tonic component. To extract this signal a filter with a narrow pass band is applied, composed of the cascade of a 3rd order Butterworh low-pass filter, with a cutoff frequency of $1.8Hz$, and a 9th order Butterworth high-pass filter, with a cutoff frequency of $0.05Hz$.

For both components, the Python *filtfilt()* function of the scipy library was used, which applies a linear digital filter twice, once forward and once backwards. The combined filter has zero phase and a filter order twice that of the original. The filtering effect of the stage is shown in Figure 3.12.

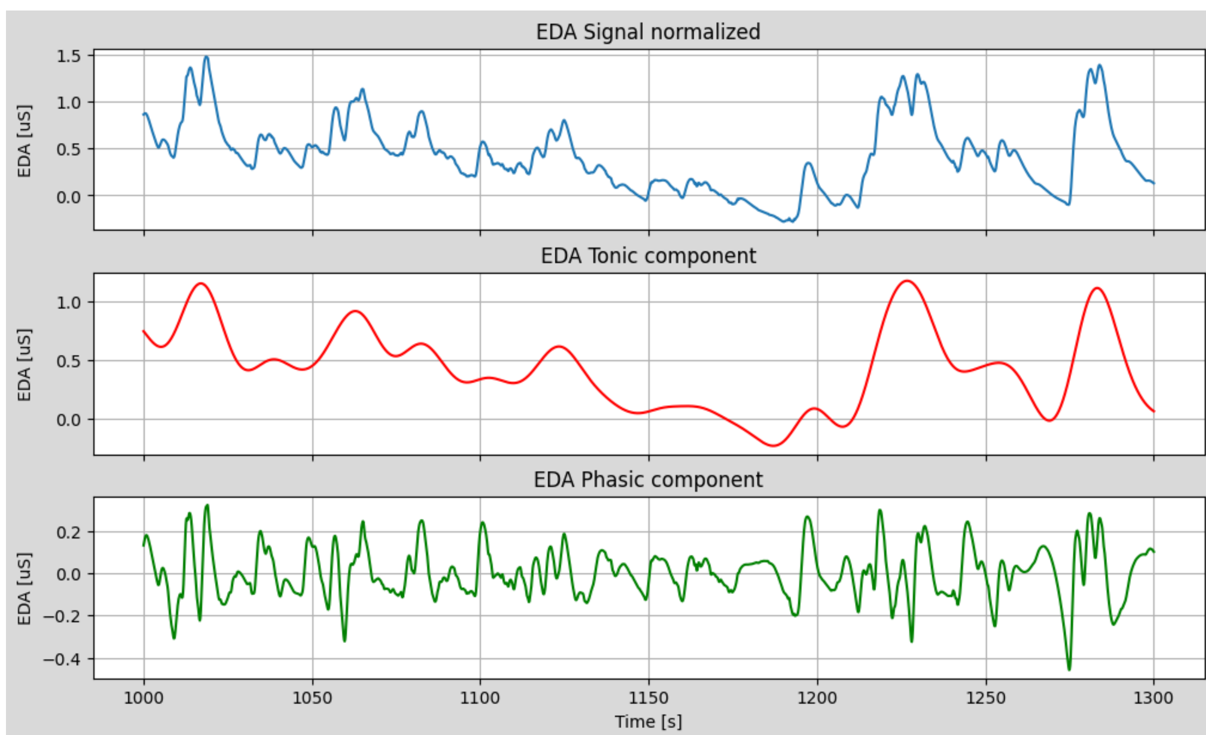


Figure 3.12: EDA Tonic and Phasic

3.3.3. Critical points detection

For human emotional state assessment, short term fluctuations of the phasic component should be captured and analyzed. In order to achieve this result an algorithm was developed, composed of three main parts: a peak detector, a point of onset detector and

an half recovery point detector. To help the detection two useful signal are computed. The first is the zero crossing of the first derivative of the phasic signal, which was derived by filtering with a 3 points digital differential filter described by the transfer function of Equation 3.9. The second is the second order derivative of the phasic signal, computed applying a 7 points digital differential filter with transfer function described in Equation 3.10. Both transfer function frequency responses are plotted in Figure 3.13.

$$H(z) = 1/2z - 1/2z^{-1} \tag{3.9}$$

$$H(z) = 1/10z^3 + 1/20z^2 - 1/10z - 1/10 - 1/10z + 1/20z^{-2} + 1/10z^{-3} \tag{3.10}$$

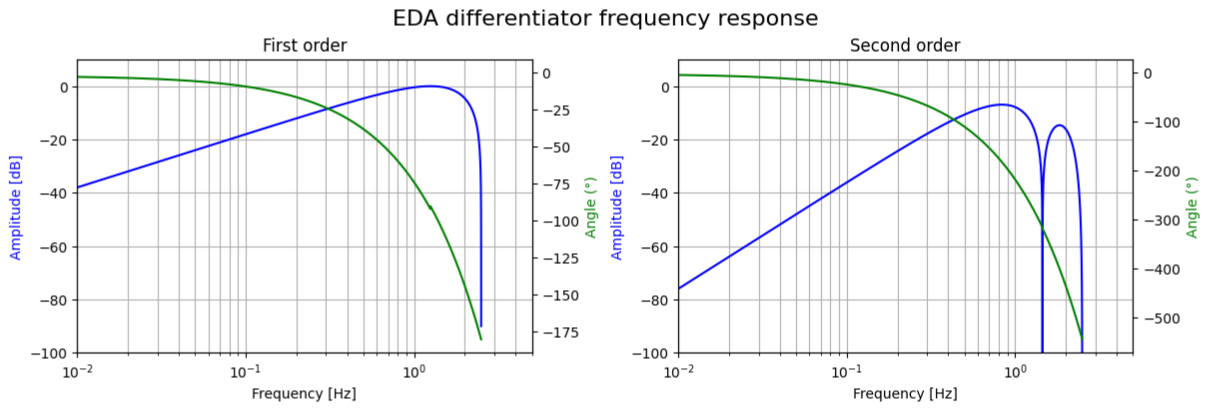


Figure 3.13: First and Second order derivative frequency response

Peak detection The detailed representation of peak detection algorithm is depicted in Figure 3.14. Event-Related Skin Conductance Responses (ER-SCR) reflect on the electrodermal activity signal as fluctuations, but the skin may generate 1 to 3 Non-Specific SCR (NS-SCR) per minute [8]. For this reason the algorithm analyzes 10 seconds of data at a time, dividing the window into 2-second segments and shifting at every cycle by the segment size [32]. First the absolute maximum point of the phasic signal is detected, thanks to the previously computed phasic properties. If a one is detected in the zero crossing array and if the second derivative is lower than zero, the corresponding index is marked as a possible peak. Then a double check on the amplitude is performed. First the phasic value at the peak is compared to a fixed threshold of 0.01μS, which is a largely arbitrary value chosen to discard immediately NS-SCR events. Then the EDA value (tonic plus phasic component) at the peak is compared with a threshold computed according to Equation 3.11. If the value is higher than the threshold, the point is marked as a classified

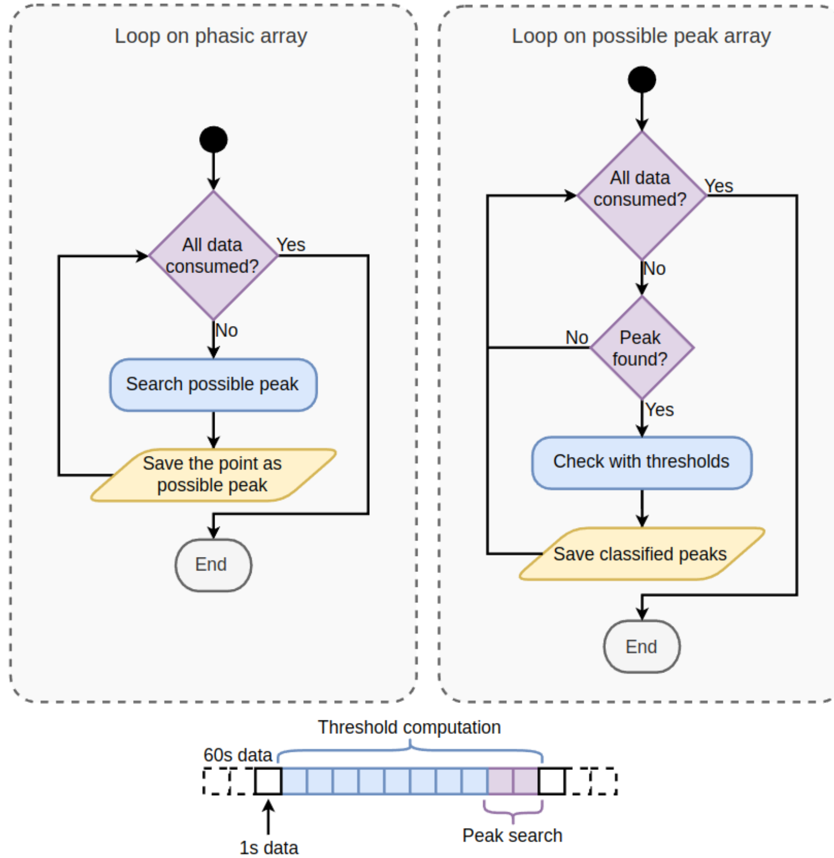


Figure 3.14: Peak detection algorithm

peak, otherwise the point will be discarded.

$$EDA_{peakThrd} = \gamma * \left[\overline{EDA_{window}} + \max EDA_{window})\right] \tag{3.11}$$

The parameter γ is a threshold factor to regulate the severity of the peak detection algorithm and in this work is always equal to 0.4.

Point Of Onset detection For each peak detected, an onset must be identified, which is by definition the point at which a change in the slope of the curve occurs. The representation of the searching procedure is depicted in Figure 3.15. The algorithm loops on the classified peaks array, in order to find the first point, in the range of 3.2 seconds from the pointed peak, where the zero crossing value is one and the second derivative is higher than zero, hence a minimum point [32]. The time distance limit is chosen considering that, starting from an external stimulus and considering the latency of the human sympathetic nervous system, the peak related to that event should be within 1 to 4 seconds [8].

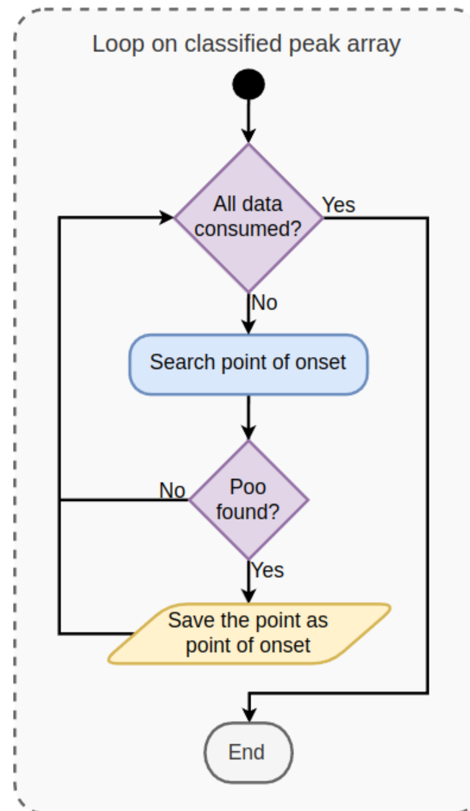


Figure 3.15: Point of Onset detection algorithm

Half Recovery Point detection Once peaks and points of onset are detected, it is also useful to detect the point at which the signal decreases to half of the peak value. The reason the full recovery point is not considered is because such amplitude usually not happens, mostly due to tonic shifts in the EDA signal and increased moisture on the skin. The HRP detection is easily achieved by searching, from one peak to the immediately following onset, as described in Figure 3.16. The half recovery point is not always present, especially if a second SCR begins before conductance levels have dropped sufficiently. In those cases, peak and point of onset are maintained, because they still describe the signal trend.

3.3.4. Statistical features

As random signals, physiological signals have a fairly high degree of uncertainty in their occurrence. Since in the end the focus of the work is to apply a classifier for emotional stress estimation, recognition of possible patterns is important. For this reason five statistical features are extracted, all from the tonic component, listed in Table 3.5.

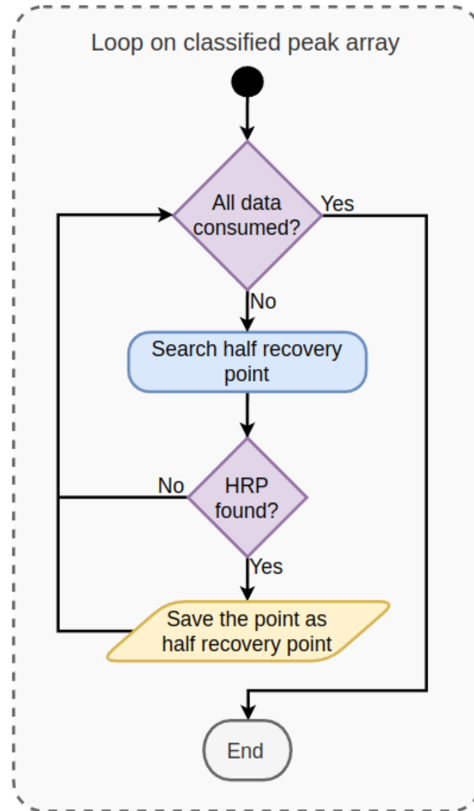


Figure 3.16: Half Recovery Point detection algorithm

	Feature	Formula
28	Tonic mean over the time window (TM)	$\frac{1}{N}\sum_{i=1}^{N} x_i$
29	Tonic variance over the time window	$\frac{1}{N}\sum_{i=1}^{N}(x_i - TM)^2$
30	Tonic energy	$\frac{1}{f_s}\sum_{i=1}^{N} x_i^2$
31	Tonic bandwidth	$\frac{1}{2\pi}\sqrt{\frac{\sum_{i=2}^{N}(x_i - x_{i-1})^2}{\sum_{i=2}^{N} x_i^2}}$

Table 3.5: EDA Statistical features

3.3.5. Syntactical features

From the detected phasic critical points, a geometric description of the signal, as Figure 3.17 describes, in the time window could be made, in the form of syntactical features. Those characteristics are listed in Table 3.6.

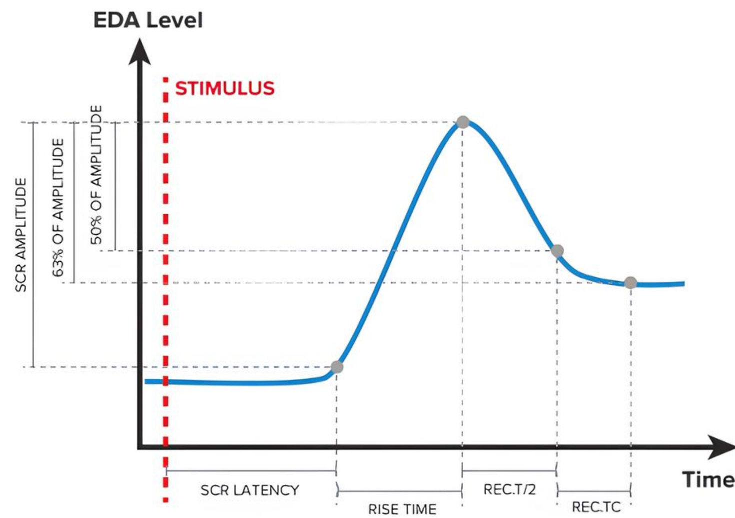


Figure 3.17: EDA phasic response with highlighted critical points

	Feature	Formula
32	Phasic peak rise time sum	$\sum (t_{peak} - t_{poo})$
33	Phasic peak amplitude sum	$\sum (SCR_{peak} - SCR_{poo})$
34	Normalized phasic half recovery sum	$\frac{1}{N} \sum (t_{hrp} - t_{peak})$
35	Phasic peak energy sum	$\frac{1}{2} \sum [(t_{peak} - t_{poo}) * (SCR_{peak} - SCR_{poo})]$
36	Phasic rise rate average	-
37	Phasic decay rate average	-
38	Phasic percentage decay	-
39	Phasic number of peaks	-
40	Phasic highest amplitude (PPHA)	$\max (SCR_{peak} - SCR_{poo})$
41	Phasic highest amplitude rise time	$t_{PPHA_{peak}} - t_{PPHA_{poo}}$

Table 3.6: EDA Syntactical features

The following list is intended to better explain some of the above features that are difficult to express with a formula:

- **Feature 36**: The phasic rise rate average is defined as the sum of the SCR first derivative points that are higher than a threshold fixed at $0.025\mu S/s$
- **Feature 37**: The phasic decay rate is defined as the average of the SCR first derivative points that are lower than a threshold fixed at $-0.025\mu S/s$
- **Feature 38**: The phasic percentage decay is defined as the percentage of samples, over the total number of samples, which has first SCR first derivative lower than 0

3.4. Feature set exploration

After the required features have been extracted a features exploration step is required. This step will perform cleaning and selection on the previously extracted features so that only the features with the most significance will be used for classifier training.
The structure is shown in Figure 3.18.

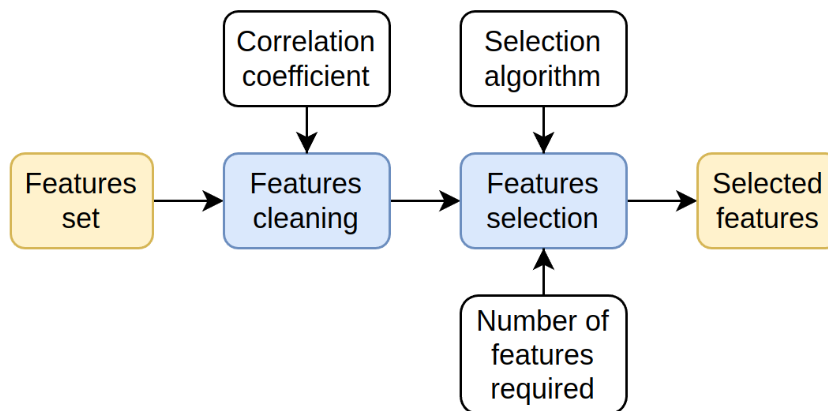


Figure 3.18: Features set exploration

The first block will perform Features cleaning. Features with an high correlation with others need to be removed from the feature set since they will not bring additional information (or just very few), but will increase the complexity of the algorithm, thus increasing the risk of errors. Moreover, for some specific machine learning algorithms, the presence of a pair of highly correlated features will provide a bias to the system further decreasing performances.
Once the features cleaning step has been performed features selection is needed so that only the most informative features as kept for classifier training. The reduction of the

feature count allow for simpler classifiers model to be obtained thus speeding-up training and reducing errors introduced by more complex models.

3.4.1. Feature cleaning

The feature cleaning step is performed by first computing the correlation matrix of the features set and using it to do the cleaning operation.
Once a pair of features with a correlation higher than the specified threshold is found the algorithm will remove the feature which had the highest correlation with all the other features; in this way the most informative feature is kept of each pair.
The correlation threshold can be externally specified with the use of *papermill* and has been kept at 0.95 after some initial exploration.
The correlation matrix of the features set is shown in Appendix B.

3.4.2. Feature selection

Different algorithms can be used for the feature selection step of the pipeline. The algorithm is used to assign a score to each features and then the highest scoring features are preserved for classifier training. The number of features to select as well as the algorithm to use are parameters configurable through *papermill*.
The following features selection algorithm were tried:

ANOVA-F ANOVA[36] is based on the analysis of variance. It is a parametric statistical hypothesis test for determining whether the means from two or more samples of data come from the same distribution or not. Using this analysis method the selector will keep the features which have variances correlated to the target variance.

CHI-2 Chi-2[37] is a statistical test used to examine the differences between categorical variables from a random sample in order to judge the goodness of fit between expected and observed results. Using this test the features which are most likely to be independent from the target are removed.

MI (Mutual Information) MI[22] is the measure of the mutual dependence between two variables. Using this test the algorithm will select the features which provide the most information content overall, thus discarding features which provide similar information content.

3.5. Classifiers exploration

After the feature exploration step is completed, a set of the most informative features of the signals is provided to the Classifiers Training and Validation blocks.
With an iterative approach, this step performs the training of six different classifiers by exploiting the *cross validate* function of the **scikit-learn** Python library [30].

3.5.1. Training Methodology

When dealing with prediction function, learning and testing it on the same set of data leads to unavoidable mistakes. In order to avoid over-fitting and obtain valid results, the database must be divided into two groups: training data and test data.
The method used in our work consists on splitting the training dataset into k smaller sets, called folds. A model is trained using $k-1$ of the folds as training data and the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute performance measures such as accuracy, F1 ...) [30]. After all iterations, the output performance metrics are the average of the k values computed in the loop. At the end, the test dataset is used for a final evaluation. A representation of the database splitting method is reported in Figure 3.19.

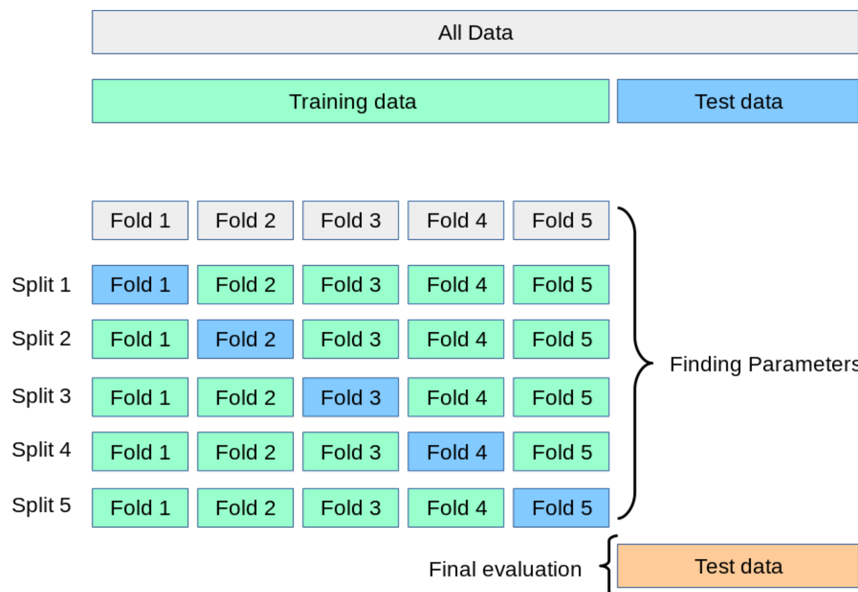


Figure 3.19: Cross-validation methodology [30]

3.5.2. Comparison metrics

To be able to evaluate the performance achieved during training of the classifiers and to choose the best one, nine metrics are extracted, listed below, derived from four commonly characteristics when dealing with machine learning models. The common characteristics are described in Table 3.7.

F1 Weighted Mean The weighed-average F1 score is computed by taking the mean over all folds of all per-class F1 scores, while considering each class's support. Support refers to the number of actual occurrences of the class in the dataset. The 'weight' essentially refers to the proportion of each class's support relative to the sum of all support values.

F1 Weighted STD This metric is a measure of the amount of variation or dispersion of the weighted F1 scores computed over all folds.

F1 Weighted Min This metric is the minimum value between the computed weighted F1 scores over all folds.

F1 Micro Mean Micro F1 score is the normal F1 formula but calculated using the total number of True Positives (TP), False Positives (FP) and False Negatives (FN), instead of individually for each class. Then all the folds corresponding values are averaged.

F1 Micro STD This metric is a measure of the amount of variation or dispersion of the micro F1 scores computed over all folds.

F1 Micro Min This metric is the minimum value between the computed micro F1 scores over all folds.

Accuracy Mean This evaluation parameter measures, for each fold, the number of correct predictions made by a model in relation to the total number of predictions made, and then compute the resulting average.

Accuracy STD This metric is a measure of the amount of variation or dispersion of the accuracy scores computed over all folds.

Accuracy Min This metric is the minimum value between the computed accuracy scores over all folds.

Metric	Formula
Precision	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F1 score	$2*\frac{precision*recall}{precision+recall}$
Accuracy	$\frac{TP+TN}{TP+FP+TN+FN})$

Table 3.7: Machine learning metrics

3.5.3. Tested Classifiers

The following classification algorithm were tried:

KNN The K Nearest Neighbour classifier (KNN) [15] works by classifying a new data point using the labels assigned to its neighbors in the feature hyperspace, with the new point being assigned to the label most common among its k nearest neighbors. In our case we tried the algorithm with $k = 3$, 5 and 7.

MLP The Multilayer Perceptron (MLP) [21] is a feed-forward artificial neural network, consisting of fully connected neurons with a nonlinear kind of activation function and at least three layers. This kind of neural network is able to classify data that are not linearly separables.

Random Forest The Random Forest classifier (RF) [27] is a learning method that operates by constructing a multitude of decision trees at training time and then computing the output label as the label produced by the majority of the trees.
It is classified as an ensemble learning method due to its use of multiple algorithm to obtain a single result.

XGBoost The eXtreme Gradient Boosting (XGB) [39] is a framework for the use of regularized gradient boosting techniques. In particular the obtained model is an ensemble of weak decision trees.
The XGBoost model often achieves higher accuracy than a single decision tree but it sacrifices its intrinsic interpretability.

Gaussian Naive Bayes The Naive Bayes is a classification technique used in Machine Learning based on the probabilistic approach, with the "naive" assumption of conditional independence between every pair of features. When the likelihood of the features is assumed to be Gaussian the method is called Gaussian Naive Bayes (GNB) [23].

SVM Support Vector Machines (SVM) [33] are supervised learning models that operate on the data by finding the best hyperplane that separates the points of a class from the points of another one, where 'best' is intended to be the one with the higher margin between the classes. This method works well with high dimensional spaces (large number of features) and small dataset.

3.5.4. Selected Classifier

The best results obtained after all the permutations of the Classifier training pipeline are shown in Figure 3.20 and Figure 3.21.

The two images show the results for normalized EDA data and not-normalized EDA data. The 10 highest performing options are shown sorted by highest mean accuracy.
It can be easily seen that performance are higher for normalized data, this has to be expected due to the presence of offsets in the EDA signal as explained is Section 3.3.1.

In the end we choose to use non-normalized EDA data even if the performance are lower. This was done due to difficulties of applying the normalization in a real-time setup as explained in more details in Section 5.2.2.

In the end we choose the classifier to use as the top performing one using non-normalized EDA data. The chosen Classifier is the XGBoost model, trained on 25 features selected with the CHI-2 algorithm, running on a 60s windows size.

WINDOW-SELECTOR-CLASSIFIER-# OF FEATURES	F1 WEIGHTED Mean	F1 WEIGHTED Min	F1 MICRO Mean	F1 MICRO Min	ACCURACY Mean	ACCURACY Min
60_1-CHI_2-XGB-28	68.9	51.8	69.8	54.9	69.8	54.9
60_1-MI-XGB-28	68.9	51.8	69.8	54.9	69.8	54.9
60_1-ANOVA-F-XGB-28	68.9	51.8	69.8	54.9	69.8	54.9
60_1-ANOVA-F-XGB-27	68.9	51.8	69.8	54.9	69.8	54.9
60_1-ANOVA-F-XGB-25	68.6	53.5	69.4	56.6	69.4	56.6
60_1-CHI_2-XGB-27	68.3	52.6	69.2	55.6	69.2	55.6
60_1-ANOVA-F-XGB-23	68	50.2	69.1	54.3	69.1	54.3
60_1-ANOVA-F-XGB-24	68.1	51.9	69	54.9	69	54.9
60_1-MI-XGB-24	68.1	51.9	69	54.9	69	54.9
60_1-CHI_2-XGB-26	68.1	51.6	68.9	54.9	68.9	54.9

Figure 3.20: Results of classifiers training (Normalized Data)

WINDOW-SELECTOR-CLASSIFIER-# OF FEATURES	F1 WEIGHTED Mean	F1 WEIGHTED Min	F1 MICRO Mean	F1 MICRO Min	ACCURACY Mean	ACCURACY Min
60_1-CHI_2-XGB-25	59.9	51.6	60.7	53.3	60.7	53.3
55_1-ANOVA-F-XGB-21	59.8	51.1	60.7	51.7	60.7	51.7
50_1-ANOVA-F-XGB-20	59.8	46.8	60.6	49.4	60.6	49.4
50_1-ANOVA-F-XGB-21	59.8	46.8	60.6	49.4	60.6	49.4
55_1-MI-XGB-25	59.7	51	60.6	52.8	60.6	52.8
55_1-ANOVA-F-XGB-25	59.7	51	60.6	52.8	60.6	52.8
60_1-ANOVA-F-XGB-24	59.8	50.3	60.6	50.8	60.6	50.8
45_1-ANOVA-F-XGB-24	59.5	48.4	60.6	52.2	60.6	52.2
45_1-MI-XGB-24	59.5	48.4	60.6	52.2	60.6	52.2
55_1-MI-XGB-26	59.7	51	60.6	52.8	60.6	52.8

Figure 3.21: Results of classifiers training (Non-Normalized Data)

4 | Proposed solution

Once the Python algorithm was developed it was necessary to move it to a suitable embedded system.

The Python algorithm worked in a post-processing fashion: data are read from data files previously acquired, meaningful signals are extracted and then features are computed for all the available time windows. This working behaviour was not compatible with our real-time processing goals and cannot be directly moved to an embedded system.
The embedded system instead should work with real-time data, acquiring the signals directly from sensors and producing an output label once every second in real-time. For this reasons a suitable architecture for the real-time system needs to be developed and the algorithm needs to be adapted to work in a real-time fashion.

It was chosen to develop the embedded system as a **Heterogeneous system**, in this topology a conventional CPU was paired with a specific **Biosignal Coprocessor** to aid in the processing of the signals acquired from the sensors.
The choice of a heterogeneous topology has several advantages in terms of performance and power consumption ease of design with respect to an homogeneous CPU-only system.

Performance advantages The use of specialized hardware components for some part of the system allow for increased throughput and reduced latencies: an example of this are a digital filtering stages (e.g. FIR filters) which can be easily synthesized in hardware with higher performances than their CPU counterparts. Furthermore multiple hardware components can be added to exploit parallelization on tasks which may benefit from it.
The use of specific hardware components can also free the CPU from computationally intensive tasks, thus indirectly increasing the performances of CPU related tasks and lowering the requirements for the CPU part.

Power consumption A specialized hardware block can be made more power efficient than a general purpose one since it can be specifically optimized for its task; on the contrary, a CPU has the potential of performing any task but with a lower power efficiency intrinsically derived from its general purpose design.

The use of those hardware block inside a heterogeneous system can therefore produce a system with an overall lower consumption compared to a conventional system; this is particularly advantageous in case of wearable or portable devices.

The specific Hardware/Software partitioning we selected for the system will be explained and justified in details in Section 4.2.

4.1. Platform choice

Starting from previous experiences with the EDA tools achieved during the master's degree courses about digital electronics design in Politecnico, the easy availability and the presence of plenty documentation, an AMD Xilix platform was chosen for this work. The target device family chosen was the Zynq™ 7000 SoC, which integrates the software programmability of an ARM®-based processor with the hardware programmability of an FPGA.
These products integrate a dual-core ARM® Cortex™-A9 base processing system (PS) and a 28nm Xilinx programmable logic (PL), together with integrated memory, a variety of peripherals, and high-speed communications interfaces [7]. Its simplified model is shown in Figure 4.1.

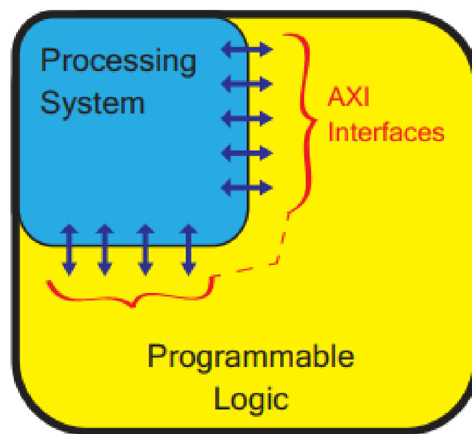


Figure 4.1: Zynq™ 7000 SoC structure model [7]

The Zynq™ 7000 SoC family comprises of different general purpose devices, all with slightly differences in terms of fabric components, as shown in Figure 4.2.

Zynq®-7000 SoC Family

		Cost-Optimized Devices						Mid-Range Devices			
	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Processing System (PS)	Processor Core	Single-Core ARM® Cortex™-A9 MPCore™ Up to 766MHz			Dual-Core ARM Cortex-A9 MPCore Up to 866MHz			Dual-Core ARM Cortex-A9 MPCore Up to 1GHz[1]			
	Processor Extensions	NEON™ SIMD Engine and Single/Double Precision Floating Point Unit per processor									
	L1 Cache	32KB Instruction, 32KB Data per processor									
	L2 Cache	512KB									
	On-Chip Memory	256KB									
	External Memory Support[2]	DDR3, DDR3L, DDR2, LPDDR2									
	External Static Memory Support[2]	2x Quad-SPI, NAND, NOR									
	DMA Channels	8 (4 dedicated to PL)									
	Peripherals	2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO									
	Peripherals w/ built-in DMA[2]	2x USB 2.0 (OTG), 2x Tri-mode Gigabit Ethernet, 2x SD/SDIO									
	Security[3]	RSA Authentication of First Stage Boot Loader, AES and SHA 256b Decryption and Authentication for Secure Boot									
	Processing System to Programmable Logic Interface Ports (Primary Interfaces & Interrupts Only)	2x AXI 32b Master, 2x AXI 32b Slave 4x AXI 64b/32b Memory AXI 64b ACP 16 Interrupts									
Programmable Logic (PL)	7 Series PL Equivalent	Artix®-7	Artix-7	Artix-7	Artix-7	Artix-7	Artix-7	Kintex®-7	Kintex-7	Kintex-7	Kintex-7
	Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Total Block RAM (# 36Kb Blocks)	1.8Mb (50)	2.5Mb (72)	3.8Mb (107)	2.1Mb (60)	3.3Mb (95)	4.9Mb (140)	9.3Mb (265)	17.6Mb (500)	19.2Mb (545)	26.5Mb (755)
	DSP Slices	66	120	170	80	160	220	400	900	900	2,020
	PCI Express®	—	Gen2 x4	—	—	Gen2 x4	—	Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC[2]	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security[3]	AES & SHA 256b Decryption & Authentication for Secure Programmable Logic Config									
	Speed Grades: Commercial	-1			-1			-1			-1
	Speed Grades: Extended	-2			-2,-3			-2,-3			-2
	Speed Grades: Industrial	-1, -2			-1, -2, -1L			-1, -2, -2L			-1, -2, -2L

Figure 4.2: Zynq™ 7000 SoC comparison

4.1.1. EDA tools

When dealing with SoC devices the manufacturers gives a variety of EDA tools to the developer in order to design its own system. EDA Tools are very sophisticated and complex software programs developed to assist with the design and manufacture of chips, providing simulation, design and verification tools.
Since Zynq™ 7000 SoC are manufactured by AMD Xilinx, the respective manufacturer tools that were used in this work are described in the following paragraphs.

Vivado™ ML Vivado™ was introduced in April 2012 by Xilinx and is an integrated design environment (IDE) with system-to-IC level tools for synthesis and analysis of hardware description language (HDL) designs. This new version of the tool features ML-based algorithms to accelerate and optimize the design process, improve delay estimation and automate strategies to reduce timing closure iterations.

Vitis™ HLS This tool allows users to easily create complex FPGA algorithms by synthesizing a C/C++ function into RTL, which could be constrained by applying directives to instruct the compiler on how the logic is desired to be implemented. Since the de-

velopment is completely done in C language, the simulation and validation process is accelerated, allowing faster iterations than a traditional RTL-based simulation.

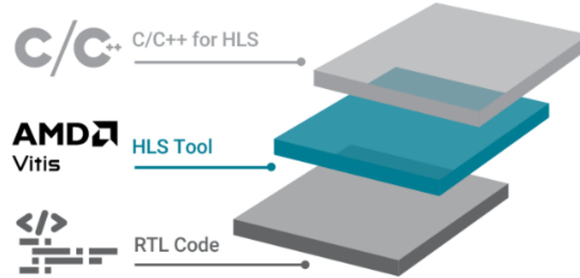


Figure 4.3: Vitis™ HLS tool overview

Vitis™ IDE The Vitis™ IDE is a software useful to develop, run and debug platforms and standalone/Linux application for AMD Xilinx devices.

4.1.2. Development board

Once the Zynq™ 7000 SoC was chosen as the required embedded platform a development board was needed.
Different options available on the market were considered:

- **Zedboard** (Based on Zynq-7020, priced around 580$)
- **Pynq-Z1** (Based on Zynq-7020, priced around 300$)
- **Arty Z7** (Based on Zynq-7010, priced around 200$)
- **Cora 7Z** (Based on Zynq-7010, priced around 150$)
- **MicroZed 7010** (Based on Zynq-7010, priced around 220$)
- **EBAZ-4205** (Based on Zynq-7010, priced around 30$)

We choose to use the EBAZ-4205 board [44] since it could be obtained at an extremely low price and features plenty of useful peripherals which low price boards were missing.
The EBAZ-4205 board was initially built as a control board for Ebit E9+ BTC miners but was being sold as development board after the decommissioning of the miners. Due to its initial intended purpose, no official support was provided for the users therefore making the development experience more difficult with respect to official development boards.
The board is shown in Figure 4.4 and the specification of its Zynq™ 7000 SoC are provided in Table 4.1.

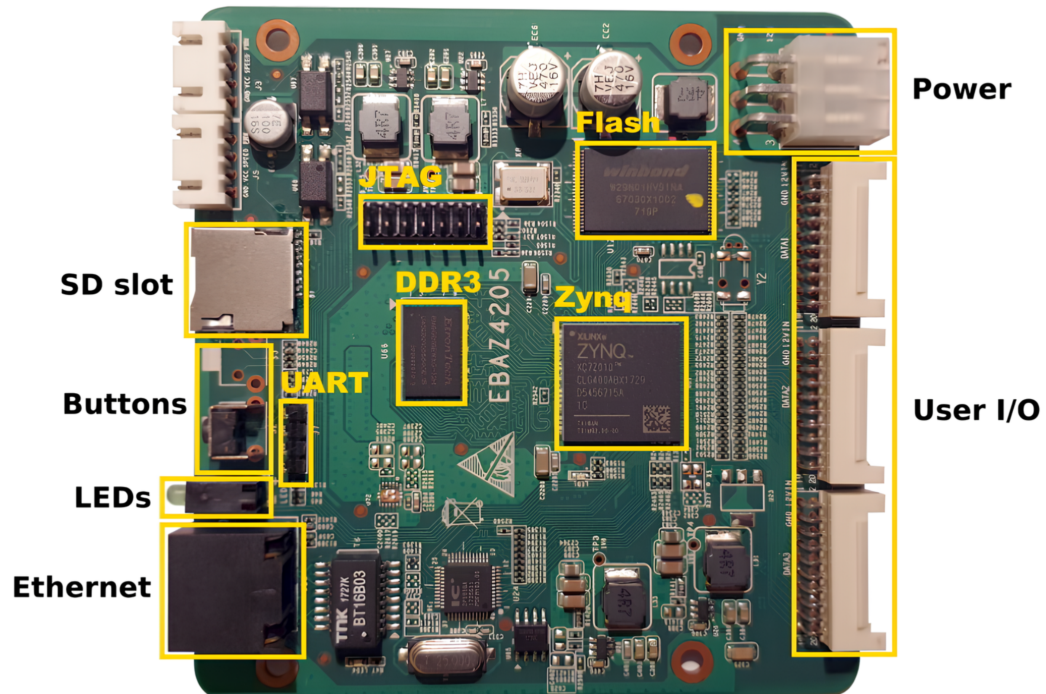


Figure 4.4: EBAZ-4205 board

The EBAZ-4205 has the following specifications:

- **SoC:** XC7Z010CLG400
- **RAM:** 256MB DDR3
- **Flash:** 128MB SLC NAND
- **Networking:** 100BASE-T Ethernet
- **Connectivity:** micro-SD slot, 60 GPIO, 2 Buttons, 2 LEDs

	XC7Z010CLG400
CPU	2 x ARM Cortex-A9 MPCore™
CPU max speed	667 MHz
L1 cache	32 KB
L2 cache	512 KB
On-Chip-Memory	256 KB
PL fabric	Artix-7 FPGA
PL logic cells	28K
PL LUTs	17 600
PL Flip-Flops	35 200
PL Block RAM	2.1 MB
PL DSP Slices	80

Table 4.1: EBAZ-4205 S0C specs

4.2. Hardware / Software co-design

Hardware/software co-design is an crucial step during development on embedded systems and a correct design could lead to important improvements in terms of performance and resource utilization. The main objective of this stage is to offload the CPU from operations that can be parallelized, implementing them in hardware, leaving the processor the chance to execute other tasks. The designed architecture is depicted in Figure 4.5 and consists on the implementation of the signal processing chain on the Programmable Logic, while the execution of the feature extractor and the classifier model on the Processing System. This structure gives multiple benefits:

- Parallelize the downsampling and filtering of the two raw signals implementing it on the fabric
- Parallelize the preprocessing and processing stages providing more significant signals to the processor
- Possibility to add newer features extracted from the signals provided by the PL
- Possibility to test different and more optimized classifiers without run the implementation process
- Give the chance to implement more than one application that can exploit the same

elaborated signals for more advanced and complex system

- Make the system portable and scalable on other platforms for future development

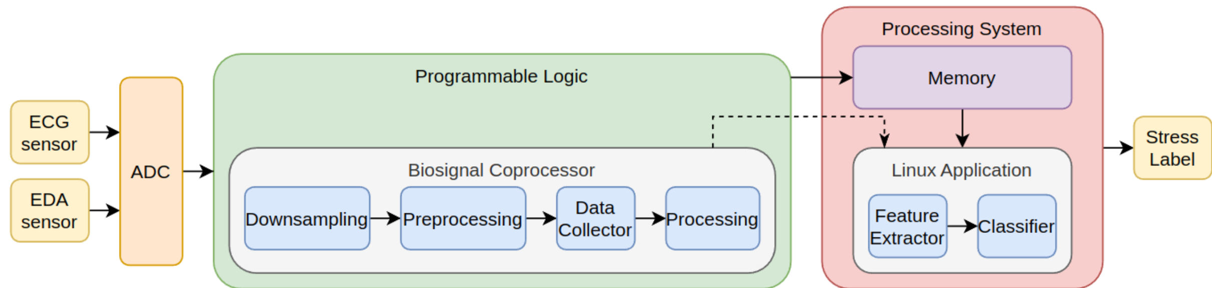


Figure 4.5: System architecture idea

As shown in Figure above the Programmable logic implements a custom **Biosignal Coprocessor** IP Core, composed of the following stages:

- **Downsampling**: it deals with filtering and downsampling of the high frequency raw signal provided from the ADC
- **Preprocessing**: it elaborates the downsampled signals to compute in real-time derived characteristics useful to simply the processing stage
- **Data Collector**: it collects the input streaming of data into arrays ready to be processed by the next block
- **Processing**: it computes significant signals from which the processing system application can calculate the features

The Processing System instead runs a user-space application, composed of different threads:

- **Feature Extractor**: it retrieves the data from the Biosignal Coprocessor core through the memory and it computes all the available features
- **Classifier**: it gets the computed features and runs the classifier, selecting only those needed

The hardware and software coexists together exchanging data through the main memory and different interfaces, which will be explained in details in the following sections.
In Appendix C the Vivado™ block design of the system is shown.

4.2.1. ADC Emulation

Since the time available to develop the system was limited and the need to test all the designed stages it was decided to emulates the real-time sensors data. By combining both

hardware and software we design an ADC Emulator block as depicted in Figure 4.6.

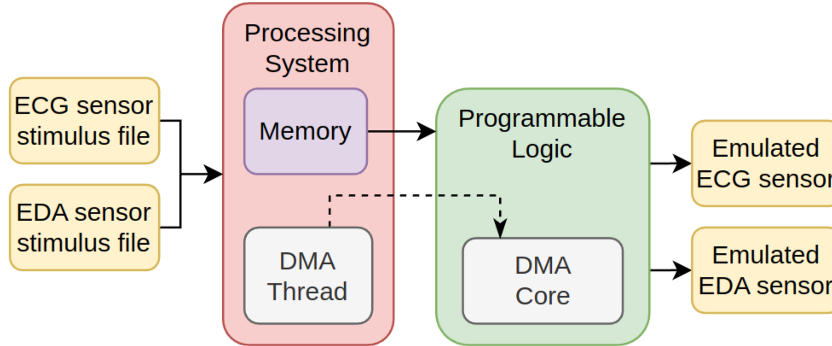


Figure 4.6: ADC Emulation

Before moving to the Linux Application two different stimulus data files were generated in Python from the dataset raw files by applying the *resample_poly* function, in order to generate two signals at the same frequency of $500Hz$. Then it was developed a synchronous thread that runs at the target $500Hz$, running inside the user-space application, that sequentially reads from the stimulus files placed in the file-system, loads the data in the main memory and triggers the data transfer through the DMA.
The AXI DMA is a soft Xilinx IP core which provides high-bandwidth direct memory access between memory and AXI4-Stream target peripherals. For the purpose of this thesis it was used in Direct Register Mode, a lower performance but less FPGA-resource-intensive mode that can be enabled by excluding the Scatter Gather engine. In this mode transfers are commanded by setting a Source Address (for MM2S) or Destination Address (for S2MM) and then specifying a byte count in a length register [41]. For this work two of this IP Core are inserted into the design, one to produce the ECG signal stream and the second to produce the EDA signal stream, as shown in Figure 4.7.

4.2.2. Interfaces

Different interfaces are needed to allow for communication between PL, PS, Memory and DMA. Four different interface types are mainly used and will be explained in more details in the next paragraphs:

AXI-4 The AXI-4 interface [40] is an on-chip communication bus protocol developed by ARM. It was first introduced in 2003 and it is part of the Advanced Microcontroller Bus Architecture 4 specifications (AMBA 4).
It is a burst based protocol allowing for multiple data transfers in a single request; this greatly reduce the overhead necessary for the transfer of contiguous data further improving

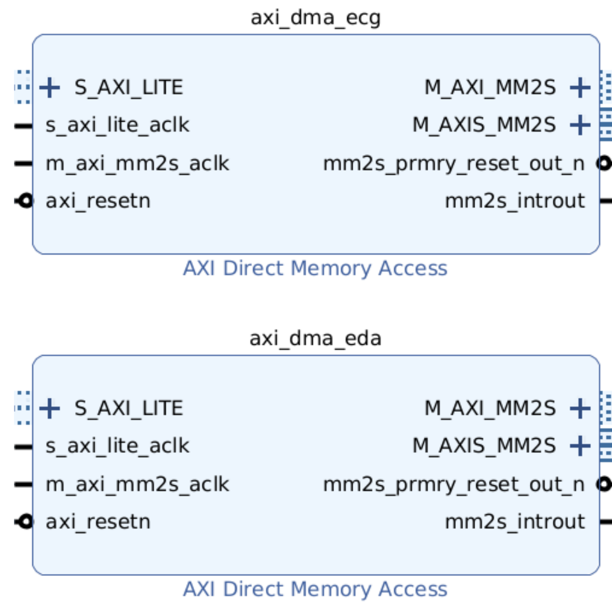


Figure 4.7: AXI DMA blocks in Direct Register Mode for ADC emulation

performances. In the AXI-4 specification, five channels are described:

- Read Address channel (AR)
- Read Data channel (R)
- Write Address channel (AW)
- Write Data channel (W)
- Write Response channel (B)

Each channel is independent from each other and has its own couple of xVALID/xREADY handshake signals.
While the communication over an AXI bus is between a single initiator and a single target, the specification includes detailed description and signals to include N:M interconnects, able to extend the bus to topologies with more initiators and targets.

AXI-4-Lite The AXI-4-Lite interface is defined as a subset of the AXI4 protocol, providing a register-like structure with reduced features and complexity. Burst support is dropped as well as the resizing of the interface data bus.
Being a subset of AXI4, AXI4-Lite transactions are fully compatible with AXI4 devices, permitting the interoperability between AXI4-Lite initiators and AXI4 targets without additional conversion logic.

AXI-4-Stream The AXI-4-Stream interface (AXIS) is a simple interface for the transfer of streaming data. It is a point-to-point protocol, connecting a single Transmitter and a single Receiver.
The basic structure is composed of a TDATA channel with the required number of bits and a pair of VALID/READ handshake signals.
Additional signals like TSTRB, TKEEP, TLAST, TID, TDEST and TUSER can be added to provide more information and aid the interface opeation.

Interrupt An Interrupt is a request for the processor to interrupt currently executing code, so that the event can be processed in a timely manner.
On Zynq™ 7000 SoCs interrupts are connected to the ARM CPU Generic Interrupt Controller (GIC) and multiple sources of interrupts are available. Moreover, interrupts can be created between the PL and PS parts so that custom hardware can signal the completion of tasks to the CPU.

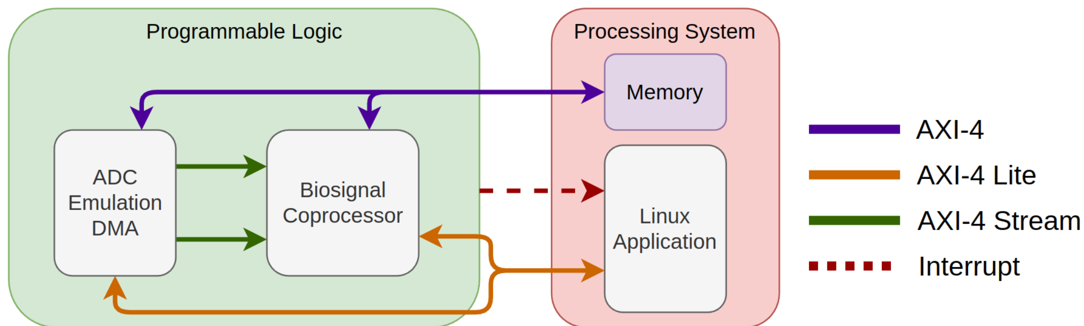


Figure 4.8: System interfaces

Figure 4.8 shows how the previously introduced interface types are used in our system to interconnect the different blocks.
In the following paragraphs those interconnections are explained in details.

Memory to DMA A full AXI-4 interface is used in a Memory-Mapped fashion. The DMA blocks use this interface to retrieve ADC emulation data from system memory. The physical interface used here is the AXI HP0 with 32bit bus width.

ADC/DMA to Biosignal Coprocessor Two AXI-4-Stream interfaces are used here, one for EDA and one for ECG. The streaming interface was chosen since the data are received in a streaming fashion from the ADC or the DMA blocks emulating it.
Physically the two interfaces are based on a 16-bit data bus with only the VALID/READY

handshakes added. The original data width from the DMA blocks is 32-bit, therefore here a AXI-4-Stream Subset Converter is used to extract only the LSBs of the bus.

Biosignal Coprocessor to Memory A full AXI-4 interface is used here, again in a Memory-Mapped fashion. Through this interface the Biosignal Coprocessor is able to write output data in memory without CPU intervention.
The physical interface used here is again the AXI HP0 like the one used for DMAs.

CPU to Biosignal Coprocessor An AXI-4-Lite interface is used here to allow control of the Coprocessor internal registers from the CPU side. Through this interface the Coprocessor can be started, stopped and configured.
Physically the interface is routed to the AXI GP0 interface of the PS.

CPU to DMA Again an AXI-4-Lite interface is used here. Through this interface the CPU can configure and start the DMAs to perform ADC emulation.
The physical interface used is the AXI GP0 with the DMA blocks acting as slaves on the bus.

Interrupts Three different interrupts are used between PL and PS in our system.
Two of those interrupts are connected to the DMA blocks used for ADC emulation, they provide a way for the CPU to know when the DMA transfers have been completed.
The third interrupt comes from the Biosignal Coprocessor, it is used to signal the CPU that a new set of output data has been saved to memory. This interrupt is raised once a second when 500 samples from the input streams have been processed.

The full block design produced in Vivado™ is shown in Appendix C.

4.2.3. System Memory Map

In order to establish a correct communication between PL and PS it is important to understand how the memory of the chosen SoC is organized. The memory is accessed through 32bit address as depicted in Figure 4.9.

As reported in Table 4.2, peripherals registers are mapped starting from the address 0x4000_0000 so for this work the described memory mapping was chosen inside the *Address Editor* of Vivado™.

Start Address	Size	Description
0x0000_0000	1024 MB	DDR DRAM and OCM
0x4000_0000	1024 MB	PL AXI slave port 0
0x8000_0000	1024 MB	PL AXI slave port 1
0xE000_0000	256 MB	IOP devices
0xF000_0000	128 MB	Reserved
0xF800_0000	32 MB	Programmable registers access via AMBA APB
0xFA00_0000	32 MB	Reserved
0xFC00_0000	64 MB — 256 KB	Quad-SPI linear address base (except top 256 KB which is OCM), 64 MB reserved, only 32 MB is currently supported
0xFFFC_0000	256 KB	OCM when mapped to high address space

Figure 4.9: Zynq-7000 SoC memory map [7]

Core	Interface	Start Address	End Address
Biosignal Coprocessor	AXI LITE	0x4000_0000	0x4000_$FFFF$
ECG AXI DMA	AXI LITE	0x4040_0000	0x4040_$FFFF$
EDA AXI DMA	AXI LITE	0x4041_0000	0x4041_$FFFF$

Table 4.2: Peripherals memory map

For the memory mapped interfaces addresses are configured through the slave AXI LITE interfaces. It was decided to organized them as described in Table 4.3.

Data	Type	Size	Start Address	Offset
DMA input			0x0A00_0000	
ECG	uint16	1		0x0000_0000
ECG	uint16	1		0x0100_0000
Biosignal Coprocessor ECG output			0x0C00_0000	
NN array	$uint16$	200		0x0000_0000
CARD array	$uint16$	750		0x0000_2000
NN counter	$uint16$	1		0x0000_4000
Biosignal Coprocessor EDA output			0x0D00_0000	
Tonic array	$int32$	1500		0x0000_0000
Phasic array	$int16$	1500		0x0000_2000
Peaks array	$uint8$	1500		0x0000_4000
POO array	$uint8$	1500		0x0000_6000
HRP array	$uint8$	1500		0x0000_8000

Table 4.3: Data organization in memory

5 | Accelerator design

The **Biosignal Coprocessor** hardware accelerator was designed to offload the main CPU from some computationally intensive tasks to be performed on ECG and EDA signals. In particular, the accelerator is performing all the filtering and processing operations required for the extraction of meaningful signals from the two sensors. It accept as inputs the two data streams at $500Hz$ (ECG and EDA) and produce as output the meaningful signals for the current time window at $1Hz$.

The accelerator was developed as a custom Vivado™ IP core synthesized using Vitis™ HLS, the internal structure is shown in Figure 5.1.

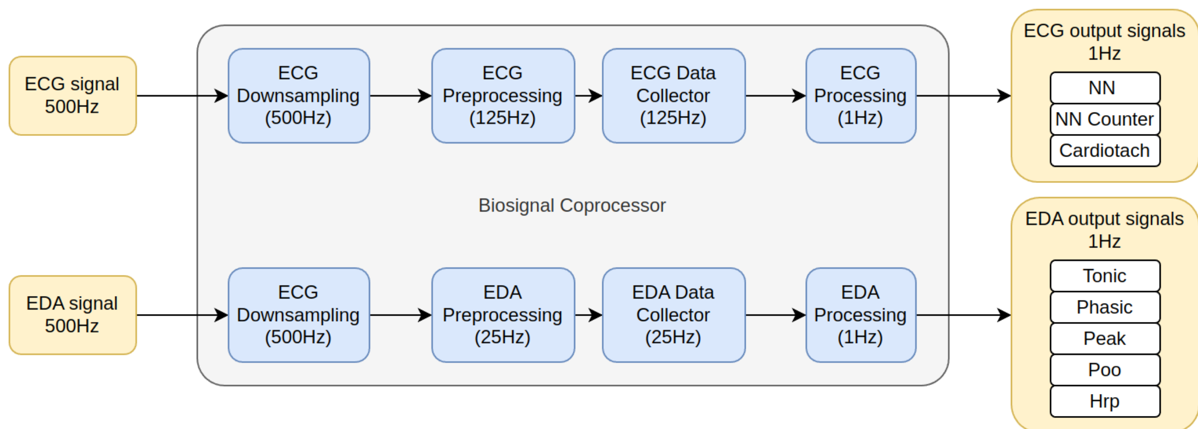


Figure 5.1: Overall Accelerator layout

The following signals are produced at the output of the accelerator:

NN Array containing the beat-to-beat intervals extracted from the ECG signal in the time window of interest.

NN Counter An integer describing the number of heart-beats found in the time window of interest.

Cardiotach Array containing the filtered and down-sampled Cardiotach signal for the time window of interest.

Tonic Array containing the Tonic signal extracted from EDA for the time window of interest.

Phasic Array containing the Phasic signal extracted from EDA for the time window of interest.

Peak Array of flags highlighting the positions of peaks found in the EDA signal for the time window of interest.

Poo Array of flags highlighting the positions of Poos (Point of On-set) found in the EDA signal for the time window of interest.

Hrp Array of flags highlighting the positions of HRPs (Half Recovery Point) found in the EDA signal for the time window of interest.

Interrupt Signal used to signal the completion of data processing for $1s$ of data, when this signal is raised the CPU can read the data from memory and proceed with the next steps.

Additionally to those output signals, the Biosignal Coprocessor is producing an interrupt signal once computations have been completed for the time windows of interest. This happen when 500 samples of both channels have been consumed, therefore the resulting interrupt is raised once every second and it is used to signal to the software part that features extraction can start.

In the following sections all the internal stages used in the Biosignal Coprocessor will be analyzed and their design process will be described.

5.1. Downsampling

The focus of this stage is to reduce the signals frequency without any loss of information or aliasing. In digital signal processing this is achieved in two steps:

1. Reduce the signal bandwidth by low-pass filtering the input signal
2. Decimate the filtered signal by a factor M

The two steps enumerated before could be cascaded as many times as necessary. The number of downsampling stages involved will be called N. The requirements of this block are:

- Design a downsampling pipeline to reduce the ECG signal frequency.
- Design a downsampling pipeline to reduce the EDA signal frequency.
- Maintain the overall delays in an acceptable range.
- Reduce the resource utilization to the minimum possible.

Since filtered data could be negative even if input data are only positive integers, as common data type for this stage a signed integer was used. The two *16 bit unsigned* integer input values were casted to two *17 bit signed* integer data using the *ap_ int.h* Xilinx library. This allowed to not waste unnecessary bit by converting them to *32 bit signed* integers.
The process of filtering before decimate is explained clearly by thinking on the frequency spectrum of a sampled signal. Sampling continuous-time signal by a certain frequency f_s results to periodicity in frequency domain of the spectrum of that signal by multiple of f_s. This is why a band limited signal is required before sampling, correctly filtered up to the Nyquist frequency $f_s/2$.
During downsampling process what it's happening is exactly the same except the fact we are sampling an already sampled signal. The input signal has to be filtered again in according to the target frequency that the stage has to achieve.
In signal processing theory there are some tricks that one can adopt during filters design. In the case of a multi-stage downsampling process, the anti-aliasing filters of the first to the second-to-last stage could be relaxed by designing those filters as **half-band** filters. In addition **coefficients scaling** could be applied to optimize the resource utilization of the implemented downsampling chain.

Half-band filter An half-band filter is a particular low-pass filter which reduces the signal bandwidth by a factor equal or multiple of $M = 2$. Taking for example a two stage downsampling process, as the input signal bandwidth goes from $0Hz$ up to $f_s/2$, considering a bandwidth of interest f_{BW} much lower than $f_s/4$, we only need to make sure that the spectrum between $f_s/2 - f_{SB}$ (where SB means Stop Band) and $f_s/2$ does not alias onto the $0Hz$ to f_{PB} (where PB means Pass Band). The part between $fs/4$ (the new Nyquist frequency of the donwsampled signal) and $f_s/2 - f_{PB}$ will alias anyway, but it will be cleaned up by the next filter stage. The designed filter will have equal pass

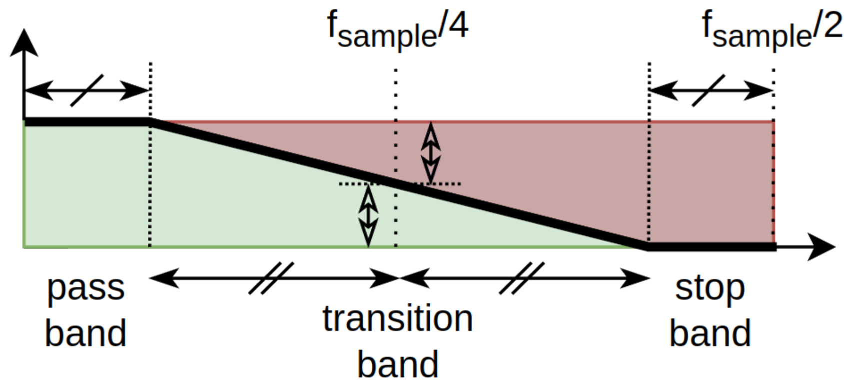


Figure 5.2: Half-band filter design

band and stop band frequencies, so with a transition band centered at $fs/4$. That makes it possible to design a FIR filter whose every other coefficient is zero, and whose non-zero coefficients are symmetrical about the center of the impulse response, as reported in Figure 5.3.

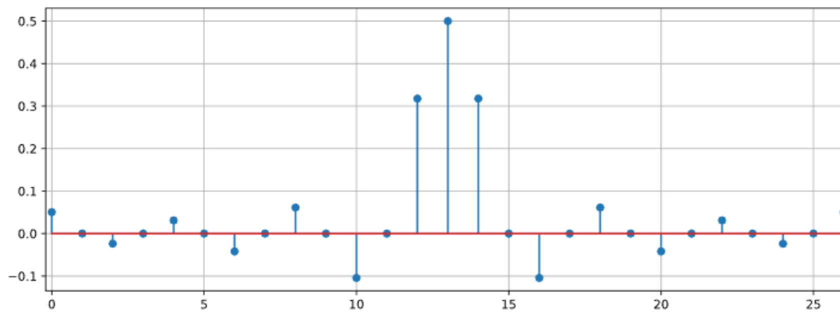


Figure 5.3: Half-band filter impulse response

Coefficients scaling When dealing with digital filters in embedded systems, another important consideration has to be made. Applying a FIR filter to a signal means taking a prefixed number of past samples and produce an output which is a weighted sum of them. Even if the input data is an integer, a generic design of a FIR produce coefficients which are not integer values, and in some cases even smaller than 1.
A very common trick that is applied in this context is to apply a scaling factor to the coefficients of the filter in order to convert the floating point values to signed integers. After applying the scaled filter the output is then scaled down by the same factor.
After setting the data type with which we want to represent the coefficients fixed to *16 bit signed* integer, the procedure followed during the design is described here step by step:

1. Remove the coefficients that are too small, for instance lower than a threshold fixed at 10^{-8}

2. Find the minimum and the maximum coefficients values after step 1
3. Compute the scaling factor of the minimum coefficient mapping it to the MIN_{int16}
4. Compute the scaling factor of the maximum coefficient mapping it to the MAX_{int16}
5. Take the smaller between the two scaling factor computed at step 3 and 4 in order to not commit overflow
6. Compute the equivalent number of bit, indicated by q, that represents the scaling factor computed at step 5, approximating the result by the lower possible integer. Then compute the resulting scaling factor as $\theta = 2^q$
7. Scale the filter coefficients multiplying them by θ and rounding them to the nearest integer value. This step was done in Python and the coefficients were inserted in the C code already scaled
8. Extract the number of bits (Σ), as in Equation 5.1, with which the accumulator variable of the filter function has to be defined in order to not commit overflow,

$$\Sigma = \eta + q + log_2(N_{taps}) \tag{5.1}$$

   where:

   - η represents the input data number of bits, in this case 17
   - q represents scaling factor number of bits
   - N_{taps} represents the number of taps of the FIR filter, so the number of summations that the filter function will perform

   An optimization could be done in the number of bits when dealing with FIR filters. Remembering the fact that signed integer values are represented in CPL2, the representation is cyclic, which means that if an overflow happens due to a summation with a positive value, if the next summation is done with opposite sign and same module, the first will be compensated, restoring the initial sign. This observation, combined to the fact that the impulse response of FIR leads to an overall gain always lower than 1, allows temporary overflows and permits a relaxation on the total number of bit required for the accumulator variable, resulting to Equation 5.2.

$$\Sigma = \eta + q \tag{5.2}$$

9. Apply the filter

10. Scale down the output of the filter by shifting by q

The following two sections describe how the steps described above are applied on the physiological signals in detail.

5.1.1. ECG Downsampling

The ECG Downsampling stage was designed to accept a 500Hz input signal producing a downsampled $125Hz$ signal at the output.
The stage was designed to keep a useful signal bandwith for the ECG signal in the range $0Hz$ - $40Hz$ and works with $N = 2$ and $M = 2$ for both stages. With the initial frequency requirements a single downsampling stage could be used but the obtained result would use more taps than a two-stage design, therefore increasing resource use.
The obtained downsampling stage is shown in Figure 5.4.

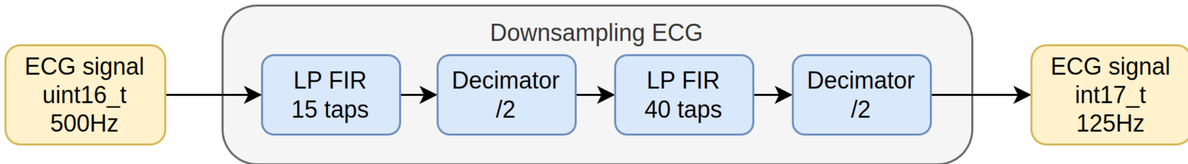


Figure 5.4: ECG downsampling

The four internal blocks employed are now described in details:

First LP filter The first LP filter is designed as an half-band filter with the specifications in Table 5.1:

Parameter	Value
Input Sampling Frequency	$500Hz$
Pass-Band	$0Hz$ to $62.5Hz$
Stop-Band	$187.5Hz$ to $250Hz$
Stop-Band attenuation	$60dB$

Table 5.1: ECG Downsampling FIR 1 requirements

Using the *scipy.signal.remez()* function, which implement a variation of the Parks–McClellan filter design algorithm [19], we found that a FIR filter of order 14 (15 taps) is able to satisfy those requirements.

The filter coefficients were scaled on *16 bit signed* to allow for the use on integer-only algebra in the HLS code. The scaling produced the results of Table 5.2 and Figure 5.5.

Parameter	Value
Scaling coefficient	32768 (2^{15})
Coefficient max relative error	0.39%
Input signal #bits	16
Coefficient #bits	16
Required accumulator #bits	32

Table 5.2: ECG Downsampling FIR 1 results

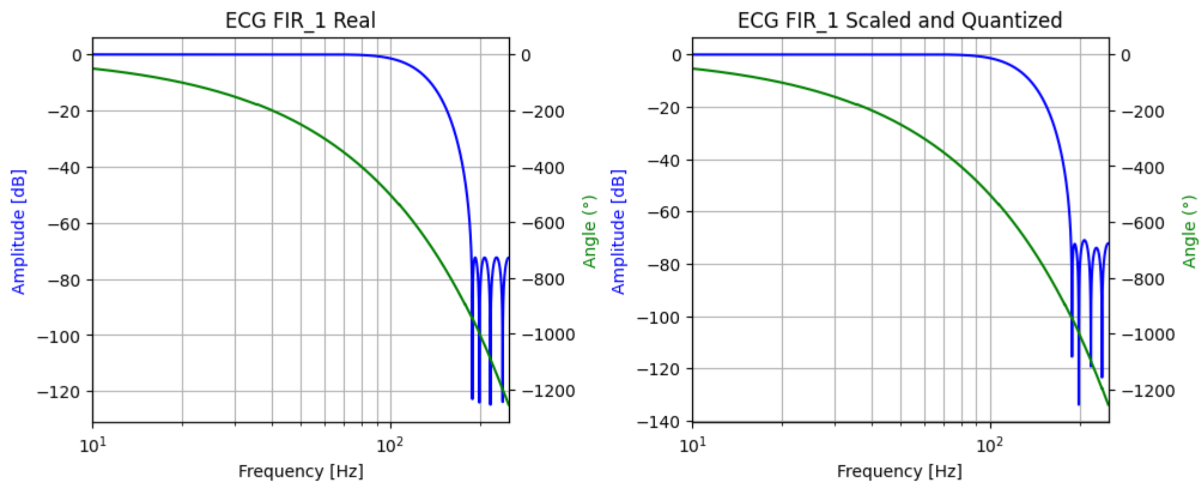


Figure 5.5: ECG downsampling FIR 1

First decimator An order 2 decimator is used here. The decimator will reduce the sampling frequency at half the initial value by keeping a sample every two.

Second LP filter The second LP filter is designed with the specifications of Table 5.3:

Parameter	Value
Input Sampling Frequency	$250Hz$
Pass-Band	$0Hz$ to $40Hz$
Stop-Band	$62.5Hz$ to $125Hz$
Stop-Band attenuation	$60dB$

Table 5.3: ECG Downsampling FIR 2 requirements

The required filter order and the filter coefficients were obtained using an iterative approach still based on the Parks–McClellan filter design algorithm [19]. The required filtering performances were obtained with a FIR filter of order 39 (40 taps).
The filter coefficients were scaled on *16 bit signed* to allow for the use on integer-only algebra in the HLS code. The scaling produced the results of Table 5.9 and Figure 5.6.

Parameter	Value
Scaling coefficient	65536 (2^{16})
Coefficient max relative error	0.54%
Input signal #bits	17
Coefficient #bits	16
Required accumulator #bits	33

Table 5.4: ECG Downsampling FIR 2 results

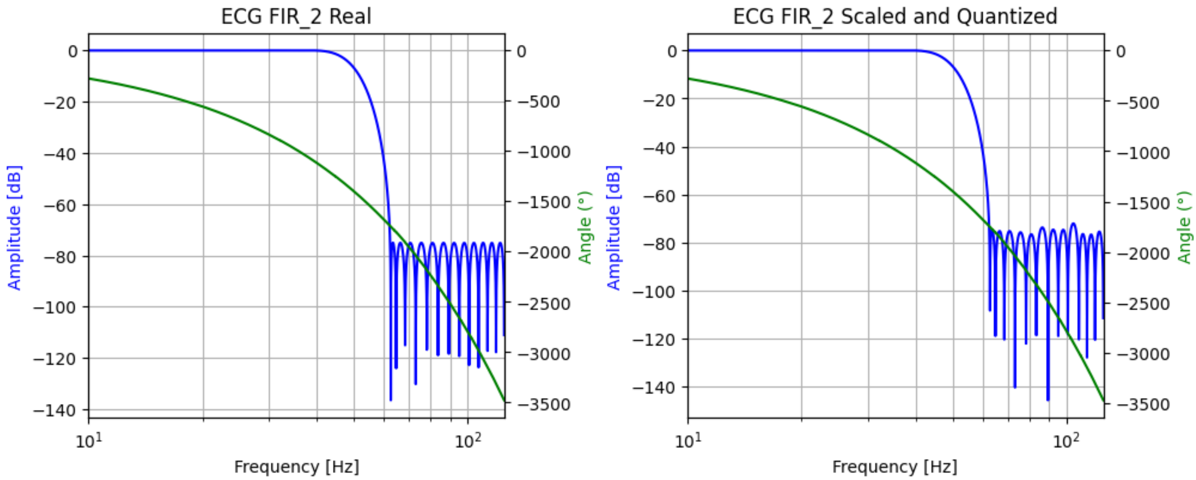


Figure 5.6: ECG downsampling FIR 2

Second decimator Again an order 2 decimator is used. The decimator will reduce the sampling frequency at half the initial value by keeping a sample every two.

Overall delay The delay induced a FIR filter is described by Equation 5.3.

$$\tau = \frac{N-1}{2} * \frac{1}{f} \tag{5.3}$$

where N is the filter length (number of taps) and f is the input frequency of the data. The group delay is constant for all frequencies because FIR filters have linear phase, so all

the input frequency components of the signal will be shifted in time of the same amount of delay. The delay for the two stages are reported in Table 5.5.

Stage	Order	Delay
Decimator 1	15	$0.014s$
Decimator 2	40	$0.078s$

Table 5.5: ECG Downsampling delays

leading to an overall stage delay of $0.092s$.

5.1.2. EDA Downsampling

The EDA downsampling stage was designed to accept a $500Hz$ input signal and producing a $25Hz$ signal at the output.
Even if the bandwidth of the signal ranges from $0Hz$ to $2Hz$, the output frequency of the stage was kept higher than $2 * 2Hz = 4Hz$ to be able to reduce the number of taps of the resulting filter by enlarging the transition band. The block is designed with a number of stage $N = 2$, with downsampling factor equal to $M_1 = 4$ for the first stage, and $M_2 = 5$ for the second stage. The chosen decimation order comes from the half-band filter theory and the minimization of the total downsampling stage group delay.
The obtained stage is depicted in Figure 5.7

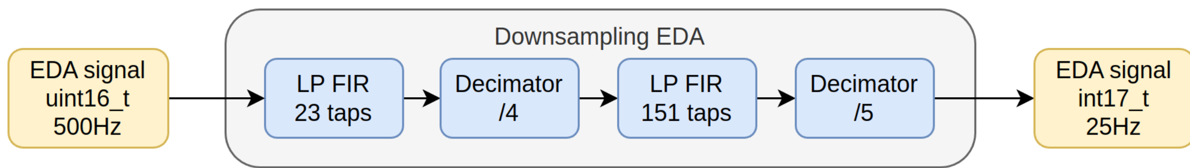


Figure 5.7: EDA downsampling

The four internal blocks employed are now described in details.

First LP filter The first LP filter is designed as an half-band filter with the specifications in Table 5.6:

Parameter	Value
Input Sampling Frequency	$500Hz$
Pass-Band	$0Hz$ to $12.5Hz$
Stop-Band	112.5 to $250Hz$
Pass-Band attenuation	$0.05dB$
Stop-Band attenuation	$60dB$

Table 5.6: EDA Downsampling FIR 1 requirements

The optimized result that satisfies the requirements, obtained by using the *remez* function of the *scipy* library results to be a filter of order 22 (23 taps). The filter coefficients were scaled on $int16_t$ to allow for the use on integer-only algebra in the HLS code. The scaling produced the results of Table 5.7.

Parameter	Value
Scaling coefficient	65535 (2^{16})
Coefficient max relative error	3.9%
Input signal #bits	16
Coefficient #bits	16
Required accumulator #bits	33

Table 5.7: EDA Downsampling FIR 1 results

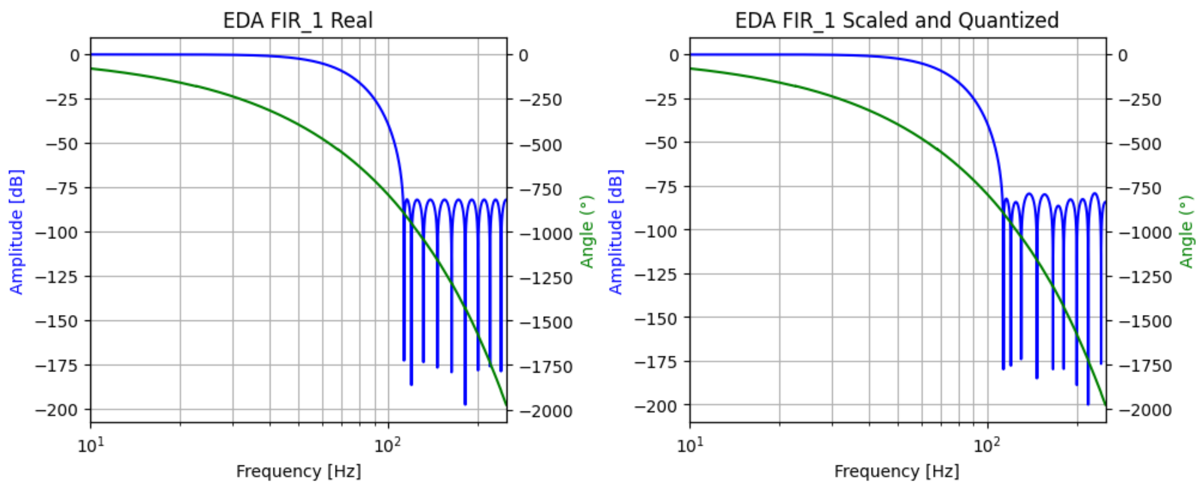


Figure 5.8: EDA downsampling FIR 1

First decimator In this step the decimation factor is 4. This is done by keeping 1 sample every 4.

Second LP filter The second LP filter is designed with the specifications of Table 5.8:

Parameter	Value
Input Sampling Frequency	$125Hz$
Pass-Band	$0Hz$ to $2Hz$
Stop-Band	$5Hz$ to $62.5Hz$
Pass-Band attenuation	$0.05dB$
Stop-Band attenuation	$60dB$

Table 5.8: EDA Downsampling FIR 2 requirements

The required filter order and the filter coefficients were obtained using an iterative approach based on the Parks–McClellan filter design algorithm [19]. The output of the iterations gives a resulting filter of order 150 (151 taps).
The filter coefficients were scaled on *16 bit signed* to allow for the use on integer-only algebra in the HLS code. The scaling produced the results of Table 5.9 and Figure 5.9.

Parameter	Value
Scaling coefficient	524288 (2^{19})
Coefficient max relative error	5.02%
Input signal #bits	17
Coefficient #bits	16
Required accumulator #bits	36

Table 5.9: EDA Downsampling FIR 2 results

Second decimator An order 5 decimator is used. The decimator will reduce the sampling frequency by five, with the same procedure of the previous one.

Overall delay The delay induced a FIR filter is described by Equation 5.4.

$$\tau = \frac{N-1}{2} * \frac{1}{f} \tag{5.4}$$

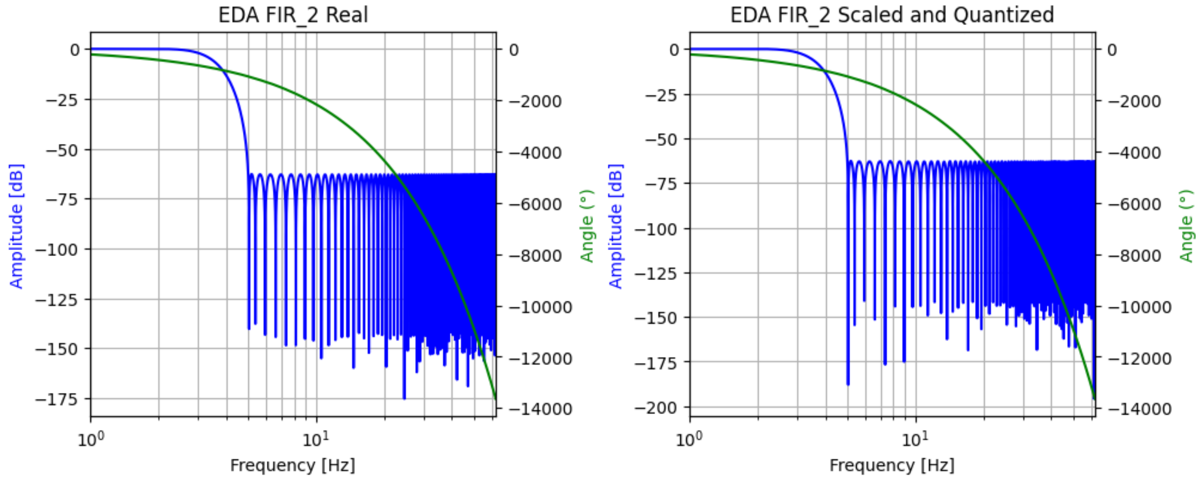


Figure 5.9: EDA downsampling FIR 2

where N is the filter length (number of taps) and f is the input frequency of the data. The group delay is constant for all frequencies because FIR filters have linear phase, so all the input frequency components of the signal will be shifted in time of the same amount of delay. The delay for the two stages are reported in Table 5.10.

Stage	Order	Delay
Decimator 1	22	$0.022s$
Decimator 2	150	$0.6s$

Table 5.10: EDA Downsampling delays

leading to an overall stage delay of $0.622s$.

5.2. Preprocessing

The Preprocessing stage has the purpose on extracting useful signals from the two input signals received from the Downsampling stage.

The ECG preprocessing will detect the heart-beats in real time while the EDA preprocessing will perform the Tonic/Phasic separation and extract additional signals which will be used in later stage of the pipeline.
The two preprocessing blocks works with streaming data, therefore providing an output signal each time an input is received. The operating frequencies are inherited from the sampling frequencies at this stage of the pipeline being $125Hz$ for ECG and $25Hz$ for EDA.

Input data are in *17 bit signed* format as the outputs of the downsampling stage; the output data are instead two data structure containing different data types aggregated together. The *ap_int.h* as well *ap_fixed.h* Xilinx libraries are extensively used to better optimize variables and structure for the FPGA implementation.

Due to the completely different behaviour of the two internal blocks they will be explained separately in the next two sections.

5.2.1. ECG Preprocessing

The ECG Preprocessing stage main task is to correctly identify the positions of the heart-beats in the ECG signal. This is done using the Pan-Tompkins algorithm [24] which was introduced before in Section 3.2.1. The Pan-Tompkins algorithm topology was adapted to an FPGA implementation, mainly in its Peak-Detector stage, this will be explained in details in the next paragraphs.

The stage is working in a streaming fashion at $125 Hz$ receiving as input the downsampled data as *17 bit signed* and producing as output a data structure composed of two data:

Peak flag This *1 bit unsigned* flag signal is set True when the current sample has been identified as an heart-beat peak. The heart-beat position is extracted using a decision stage similar to the one previously used in the Python Algorithm but optimized for FPGA implementation.

NN (Time continuous) This *11 bit unsigned* signal is the current beat-to-beat interval value. It is a stepped-like signal, it is computed keeping track of the distance in ms between the last two previously encountered peaks and it is updated each time a new peak is found.
This signal is not to be confused with the Cardiotach signal, the two are similar but the NN is intrinsically computed with one heartbeat of delay with respect to the Cardiotach. The step duration of the NN signal is linked to the current beat-to-beat interval while its height is the value of the previous beat-to-beat interval. The Cardiotach signal, on the other side, has the step duration as well as the step height equal to the current beat-to-beat interval; it cannot be computed in real-time since the knowledge of the following peak position is needed.

In Figure 5.10 the block structure is shown in details.

The design process of the different internal blocks will now be explained in details:

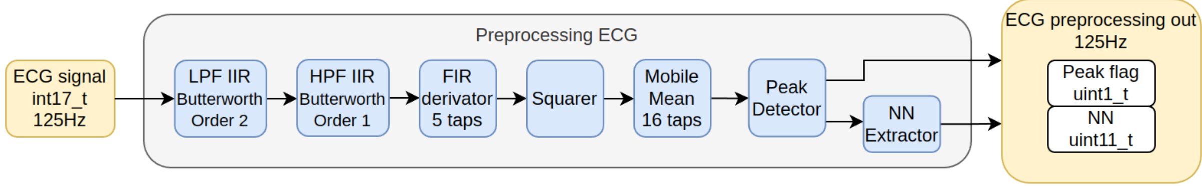


Figure 5.10: ECG preprocessing

Band-pass Filter As previously introduced in Section 3.2.1, the first step of the Pan-Tompkins algorithm is a band-pass filter use to remove unwanted noise from the signal and preserve only the bandwidth of interest.
The required filtering action is obtained with the combination of two Butterworth filters, a low-pass one and an high-pass one. IIRs filters are used here instead of FIRs due to their lower order and to keep the system more consistent with our Python implementation. Attention was payed to keep the quantization error low for the filter coefficients, this is critical for IIR filters due to their recursive nature and can introduce introduce instability otherwise.
In Table 5.11 and Figure 5.11 the performances of the obtained filters are shown.

Parameter	LP Filter	HP Filter
Cut-off frequency	$15Hz$	$5Hz$
Filter order	2	1
Filter type	$Butterworth$	$Butterworth$
Input signal #bits	17	17
Coefficient #bits	16	16
Scaling coefficient	$16384(2^{14})$	$16384(2^{14})$
Coefficient max relative error	0.006%	0.0025%
Required accumulator #bits	31	31

Table 5.11: ECG Preprocessing LP and HP filters

Derivator Filter Following the band-pass filtering a derivative action is needed to enhance the QRS complex slope information in the signal.
The derivative step of the ECG preprocessing was done using the 5 point derivative, reported in Equation 5.5, instead of the original Pan-Tompkins derivative as previously

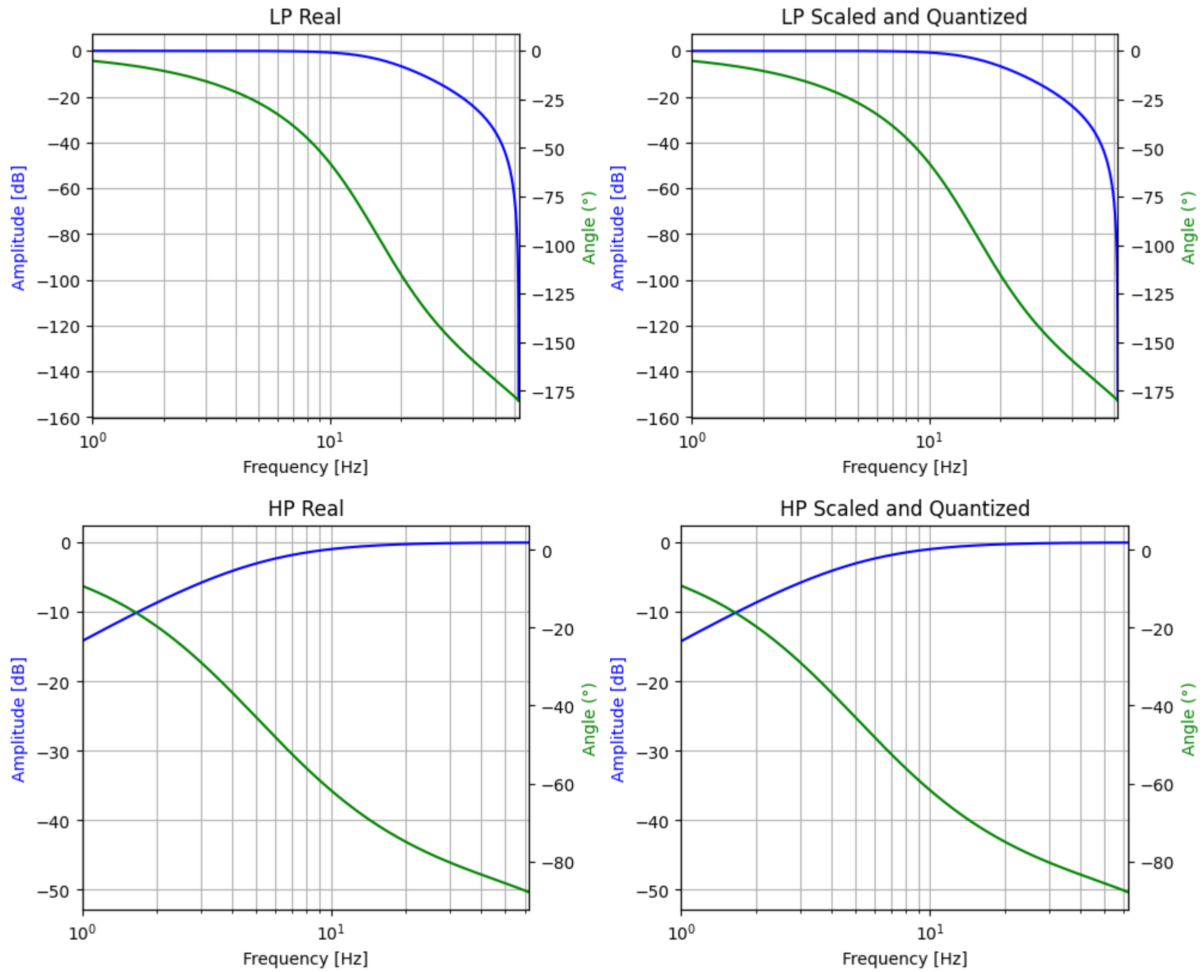


Figure 5.11: ECG Preprocessing LP and HP filters

justified in Section 3.2.1.

$$H(z) = -1/12z^2 + 2/3z - 2/3z^{-1} + 1/12z^{-2} \tag{5.5}$$

The derivative filter was implemented as an FIR filter of order 4 (5 taps) and its coefficients were quantized as *16 bit signed* to allow for the use of integer-only arithmetic. The filter parameters and its frequency response are shown in Table 5.12 and Figure 5.12.

Parameter	Value
Input signal #bits	17
Coefficient #bits	16
Scaling coefficient	$32768(2^{15})$
Coefficient max relative error	0.012%
Required accumulator #bits	32

Table 5.12: ECG Preprocessing Derivator filter

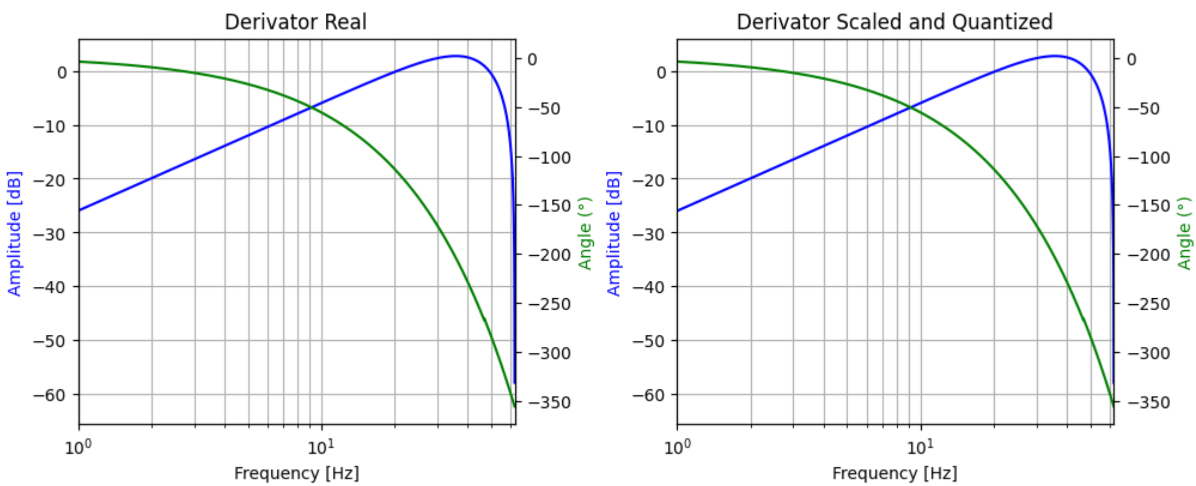


Figure 5.12: ECG Preprocessing Derivator filter

Squarer The squarer block is designed to provide a non-linear gain element to the signal further enhancing out signal of interest (high-slope regions of the QRS complex).
It accept as input a *17 bit signed* signal from the previous block and provide at the output a *32 bit unsigned* signal.

Mobile Mean Filter The signal produced at the output of the squarer block needs to be integrated to produce the required signal shape for the decision stage. A moving average filter is used for this as previously introduced in Section 3.2.1.
The window size was chosen to be 16 samples (128ms) as previously used in the Python code.
We choose to use a FIR filter topology with unitary coefficients to create the Moving Average effect, no scaling was required for the coefficients. A *36 bit unsigned* accumulator was used here since the input data from the squarer are *32 bit unsigned* and the window

size is 16 samples.
In Figure 5.13 the filter transfer function is shown.

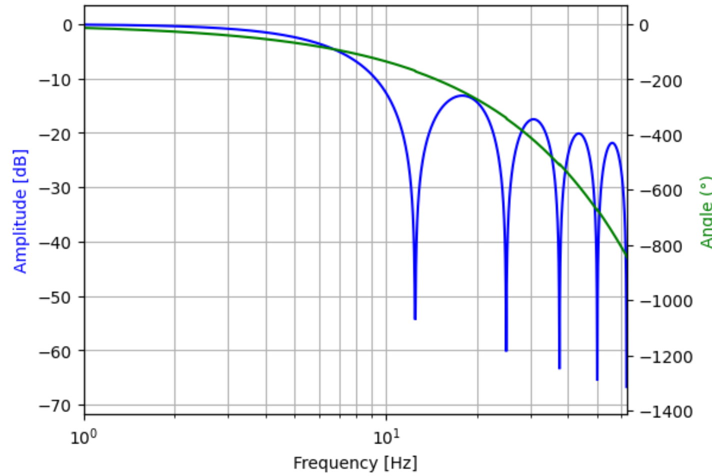


Figure 5.13: ECG Preprocessing Moving Average filter

Peak Detector The peak detector block is the one responsible of analyzing the filtered signal to distinguish where a heart-beat is present.
The original Pan-Tompkins algorithm [24], as well as out Python implementation, used here a complex decision stage with different thresholds to distinguish signal peaks from noise peaks, as well as back-search methodology for the wrongly classified points. This option was quickly ruled-out in the embedded system, due to its complexity and FPGA porting difficulties.
A simpler decision stage was developed based on Shukla and Macchiarulo [31] work providing a much simpler implementation on FPGAs. The newly developed decision stage is based on a pair of thresholds (th_1, th_2) and its dynamically tracking the signal shape to identify the peak; no back-search is necessary further simplifying the algorithm.
The decision stage works thanks to a finite state machine (FSM) shown in Figure 5.14.

The behaviour of the FSM is the following:

1. **PRE_FETCH**: Once the system is started an initial value must be estimated for the amplitude value of a peak. The first 250 samples are sampled and the maximum value between them is kept to be used later for thresholding. The window of 250 samples (2 seconds) is chosen to guarantee the presence of at least an heartbeat, the maximum value extracted will therefore be related to the peak amplitude.

2. **LOAD 1**: In this state the th_1 threshold is applied to the signal, the FSM will move to the next state and produce a peak flag once the input signal goes above

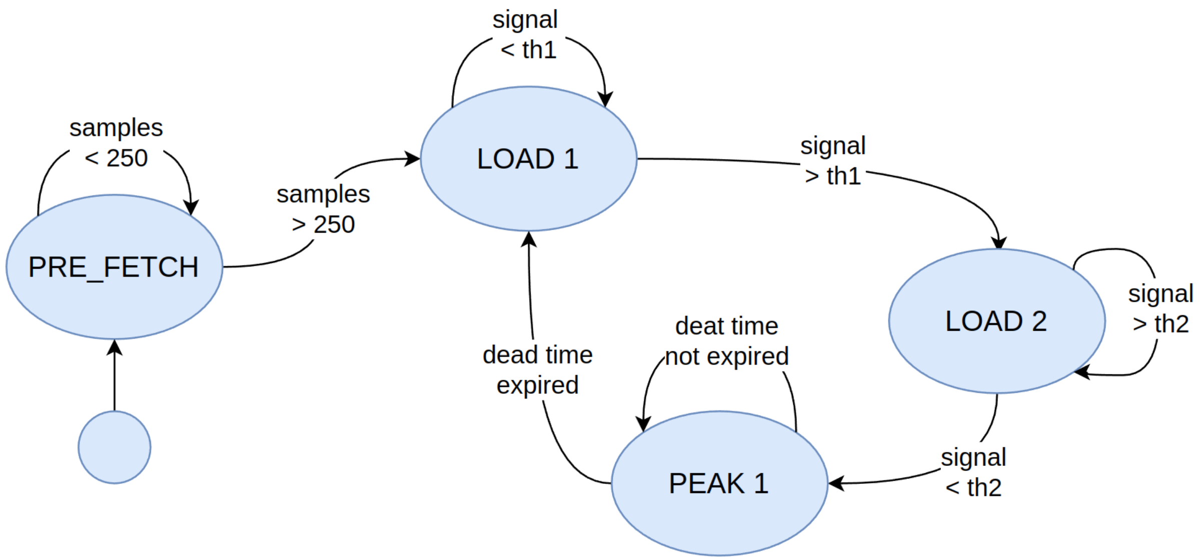


Figure 5.14: ECG Preprocessing Peak Detector FSM

*th*_1.
The threshold level is computed in two different ways: if the detector has identified less than 8 heart-beats, then 25% of the initialization amplitude value found in the PRE_FETCH step is used for *th*_1; otherwise, if more than 8 peaks have been found, then 25% of the median amplitude value of the last 8 found peaks.

3. **LOAD 2**: Here the system has started a peak detection window, the second threshold *th*_2 will be applied to find the end of the peak detection window.
   The maximum amplitude of the signal in the window will be tracked and, at each iteration, the *th*_2 level will be updated to 25% of the maximum value reached in the window.
   When the signal goes below *th*_2 the peak has for sure been passed and therefore the detection window is closed moving the FSM to the next state.

4. **PEAK 1**: This state is reached once a peak has been detected, here a 224ms (28 samples) dead-time is observed before restarting the FSM from the LOAD_1 state. The dead-time is required to avoid the triggering of the algorithm due to noise following a valid heart-beat.

As previously explained the peak flag is raised once the threshold *th*_1 is surpassed. The flag is produced as a *1 bit unsigned* integer signal.
The obtained decision stage was simulated in Python before the final HLS implementation and its behaviour is shown in Figure 5.15. The first peak flag is discarded to allow for NN initialization.

NN Extractor The last block of the ECG preprocessing is the NN-Extractor which accept as input the stream of heart-beat peak flags and compute in real-time the NN value.
The NN computation is done by keeping track of the previous beat-to-beat interval and assigning its value to the NN signal until a new peak is reached. The NN value is computed in ms, an *11 bit unsigned* is used here proving a range between $0ms$ and $2047ms$, satisfactory for beat-to-beat interval representation.

In Figure 5.15 the behaviour of the last two blocks is shown in more details, that the first peak flag is discarded to allow for NN extractor initialization. The second and third plot show the evolution of the two threshold values during operation, note the absence of threshold for the first 2 seconds of data corresponding to the PRE_FETCH state.

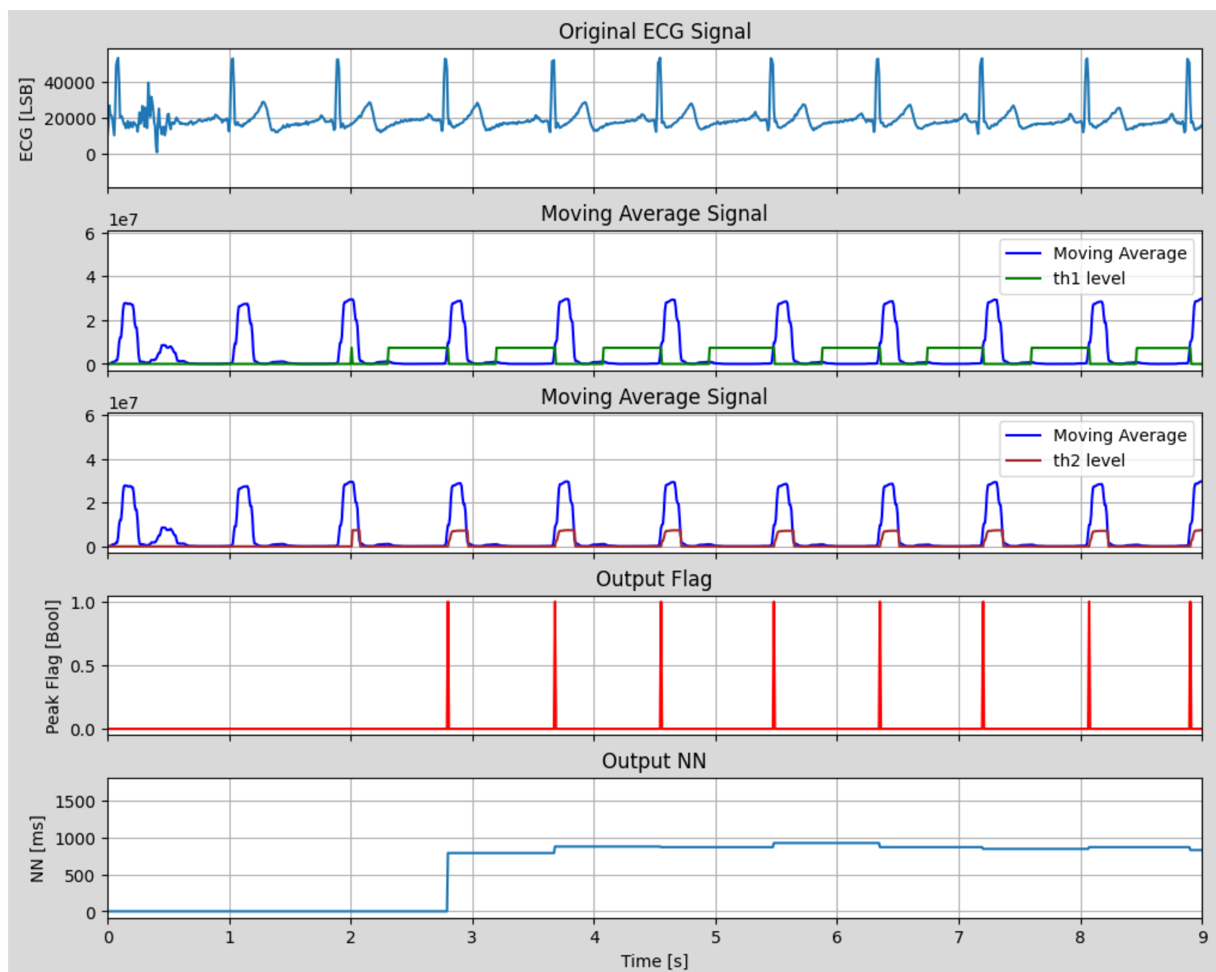


Figure 5.15: ECG Preprocessing Peak detector and NN extractor

5.2.2. EDA Preprocessing

For this stage a perfect analogy couldn't be made with the Python pipeline designed in Chapter 3.2. Since we are now dealing with real-time signal data, presenting at the input under the AXI-4-Stream protocol (Section 4.2.2), processing has to be made fast introducing the smallest delay possible. The aim of the preprocessing block is to elaborate some signal characteristics useful to make the processing stage much simpler and lighter. In Figure 5.16 a schematic is shown.

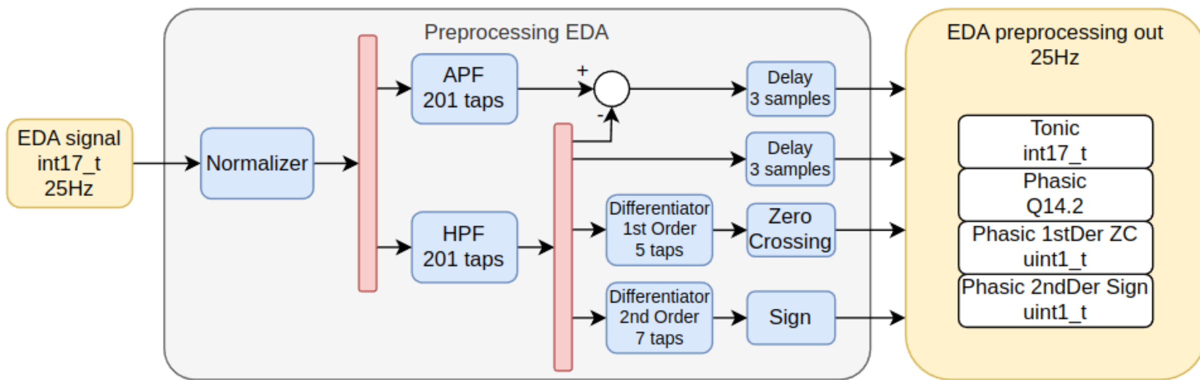


Figure 5.16: EDA preprocessing

The input EDA signal, as a *17 bit signed*, is converted in a structure composed of:

- **Tonic**: *17 bit signed* integer data stream containing the SCL component of EDA
- **Phasic**: *16 bit signed* fixed point data stream containing the SCR component of EDA
- **Phasic 1st Derivative Zero-Crossing**: *1 bit unsigned* integer data stream that flags when a zero-crossing is detected in the first derivative of the SCR component
- **Phasic 2nd Derivative Sign**: *1 bit unsigned* integer data stream that indicates the sign of the second derivative of the SCR component.

Input and output frequency of the data are the same as the downsampling stage output, so $25Hz$.
The preprocessing procedure is described in details in section below.

Normalizer This block is inserted to reproduce the function of the homonym function designed in the Python pipeline, but a direct translation of this block in hardware is not as easy as it looks. Normalizing in that way requires knowing the entire dataset in advance, which it's not feasible for a real-time system. Here are described the issues with each techniques presented in Section 3.3.1:

- **Min-Max normalization**: this method is applied knowing the relaxed period under which the patient is subjected in advance
- **Percentile normalization**: this procedure normalized the EDA signal by computing the its distribution, analyzing the entire samples of dataset.

As shown, none of the before listed could be applied.
As a solution here is proposed the online quantile estimation algorithm from [10]. It is a sequential quantile estimator which allows to estimate quantiles with respect to the given precision ϵ and it doesn't require the knowledge of the total number of samples in the dataset in advance.
For the purpose of this work, to achieve a complete picture of the system, this algorithm wasn't implemented, since it requires additional time under development and testing.

Stream broadcast The red blocks in Figure 5.16 were designed by the need to replicate the normalizer data output in order to provide it to the next blocks.
All the blocks in the chain are linked each other using *hls::stream* objects, which are designed by Xilinx and can be found in the *hls_stream.h* library. This kind of object is useful when interface between functions is needed and it is implemented as a FIFO with a default depth of 2 by default. The library provides to the user ready-to-use functions such as *read* and *write*. Since this object is designed to wrap a fully functional FIFO, a read command will empty the buffer, so multiple reads of the same sample are not feasible. The broadcaster copies the input data to all the elements of the array of streams at the output, without introducing any delay.

Signal decomposition This step comprises two blocks and the summation node of the scheme reported in Figure 5.16. To separate the SCL and SCR components of the EDA, instead of applying two separate filters, respectively a low-pass and a band-pass as it was done for the Python pipeline, a combination of an all-pass and an high-pass filter is used. The benefits that this design choice give are many:

- design just one complex filter, the high-pass FIR filter, reducing the memory occupation needed to store the coefficients
- relax the filter constraints for phasic extraction designing just an high-pass filter and not a band-pass, exploiting the already good low-pass filtering given from the downsampling second stage
- design a naive all-pass filter and just subtract from the output the high-pass filter result, to obtain the tonic component.

Given those assumptions, the focus of the work was moved to the design of the High-Pass FIR filter. The requirements are reported in Table 5.13:

Parameter	Value
Input Sampling Frequency	$125Hz$
Pass-Band	0.2 to $2Hz$
Stop-Band	0 to $0.01Hz$
Pass-Band attenuation	$0.1dB$
Stop-Band attenuation	$35dB$

Table 5.13: EDA Preprocessing High-Pass FIR requirements

The required filter order and the filter coefficients were obtained using an iterative approach based on the Parks–McClellan filter design algorithm [19]. The output of the iterations gives a resulting filter of order 200 (201 taps).
The filter coefficients were scaled on *16 bit signed* to allow for the use on integer-only algebra in the HLS code. Since the phasic signal is the output of an high-pass filter, its modulus is mush lower than the original signal, so it was decided to increase its precision representing it as fixed point value, keeping the same number of total bits. The resulting representation was a *16 bit signed* fixed point, referred here with the notation *Q14.2*, where the first number indicates the integer part (represented by 14bit, including the sign bit) and second one represents the decimal part (2 bit). The precision of this representation is equal to $2^{-2} = 0.25$.
The scaling produced the results of Table 5.14 and Figure 5.17.

Parameter	Value
Scaling coefficient	32768 (2^{15})
Coefficient max relative error	1.67%
Input signal #bits	17
Coefficient #bits	16
Required accumulator #bits	$Q32.2$

Table 5.14: EDA Preprocessing High-Pass FIR results

The delay introduced by this filter is equal to $4s$. Even if this delay seems to be very large, it turned out to be a good compromise while optimizing the in-band ripple of the

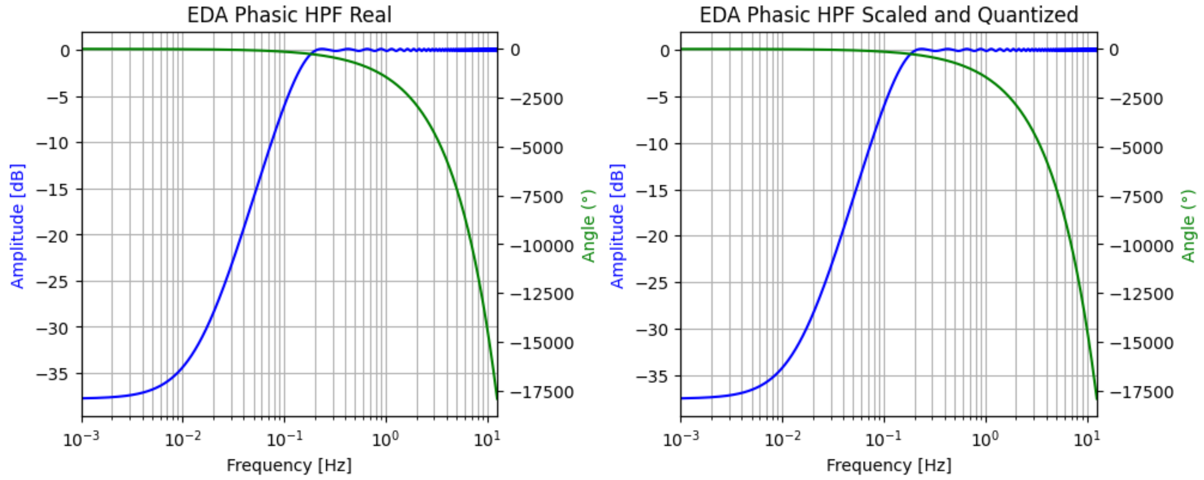


Figure 5.17: EDA Preprocessing High-Pass FIR

filter and the transition bands. It is also useful to remember the procedure followed in the Python pipeline: features from the signals are extracted after collecting $60s$ of samples, so this delay could be accepted.
Once the SCR component is extracted, the SCL component was obtained by subtracting the phasic from the normalizer output, at which first was applied an All-Pass filter. The frequency response of this filter is not reported here since its implementation is naive. Behaving as a pure phase shifter, it was implemented as a FIR exploiting the already developed function. It has 201 taps as the HPF, with all zero coefficients except the central one which is 1. The delay introduced is the same of the HPF, equal to $4s$.
Theoretically the phasic and tonic components are now ready to be provided to the output of the preprocessing block. To be complitely correct, before presenting the signal to the output, it is better to take into account the delay of the other steps in the block. In order to do that a block for each signal is inserted, reporting an equivalent delay of $0.12s$, 3 samples. The meaning of this size will be explained in the next paragraphs.

First Order Differentiator and Zero crossing Detector Reminding the operations done in the Python pipeline, in order to extract the critical points from the SCR component, the first derivative is necessary. The chosen function is reported in Equation 5.6.

$$H(z) = -1/12z^2 + 2/3z + 2/3z^{-1} + 1/12z^{-2} \tag{5.6}$$

The input of the stage is a stream object of *Q14.2* type. It was implemented as FIR filter, scaled and quantized as the previous.
The filter coefficients were scaled on *16 bit signed* to allow for the use on integer-only algebra in the HLS code. Since the differentiator is essential an high-pass filter, it was

decided to increase further more the precision, representing it as a *64 bit signed* fixed point, referred here with the notation *Q59.5*, where the first number indicates the integer part (represented by 59bit, including the sign bit) and second one represents the decimal part (5 bit). The precision of this representation is equal to $2^{-5} = 0.03125$.
The scaling produced the results reported in Table 5.15 and Figure 5.18.

Parameter	Value
Scaling coefficient	32768 (2^{15})
Coefficient max relative error	0.012%
Input signal #bits	$Q14.2$
Coefficient #bits	16
Required accumulator #bits	$Q59.5$

Table 5.15: EDA Preprocessing First order Differentiator FIR results

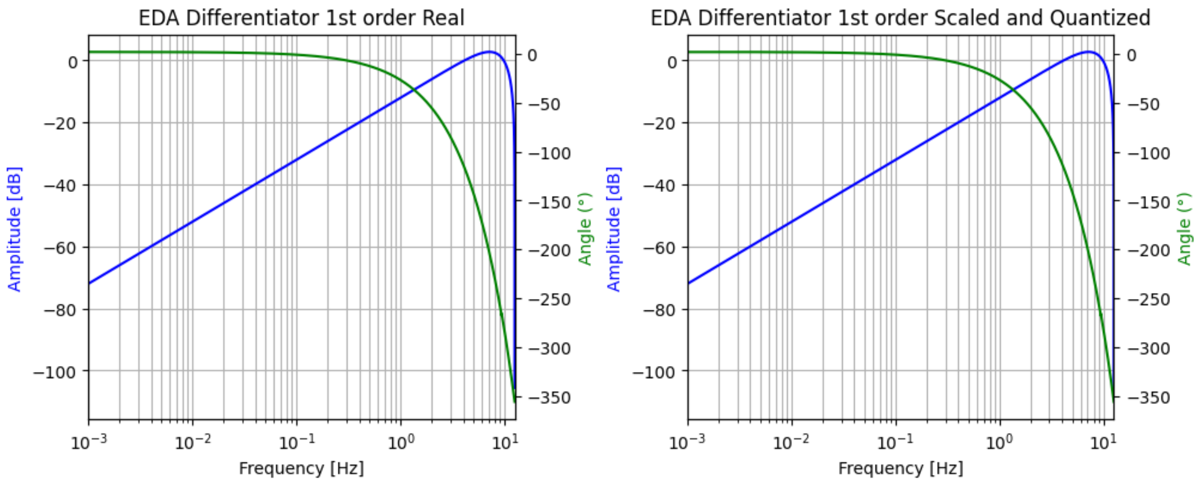


Figure 5.18: EDA Preprocessing First order Differentiator FIR

The delay introduced by this filter is equal to $0.08s$.
To detect critical points of the SCR signal, without propagate the first derivative to the next stage, it was decided to compute in this step the zero-crossing of this function. In this way, it was possible to reduce the size of the data, relaxing the resource occupation. The Zero-Crossing block was designed as a purely digital function, comparing the sign of three samples of the input signal. Table of truth 5.16 and Karnaugh-map 5.17 are reported. The synthesized function is reported in Equation 5.7.

$$Y = A'C + BC' + AB' \tag{5.7}$$

A	B	C	Out
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Table 5.16: Zero-Crossing Truth table

C \ AB	00	01	11	10
0	0	1	1	1
1	1	1	0	1

Table 5.17: Zero-Crossing K-map

The output data of the zero-crossing function is an arbitrary precision integer of *1bit*, referred here as *uint1_t*.
Since the signal is sampled, in order to detect the correct position of the highest o lowest sample value, the function was designed to detect a window of possible samples during which those values could be found, flagged with a 1. The delay introduced by this stage is equal to 1 sample, with the same reasoning of an FIR filter, corresponding to $0.04s$, which perfectly compensate the delay difference with respect to the second order differentiator filter explained in the next paragraph.

Second Order Differentiator and Sign Extractor The last explained block of the preprocessing stage is the Second Order Differentiator FIR and its sign detectior block. Those steps are useful to assess if in the window signaled by the zero-crossing there is a maximum or a minimum point. The chosen function is reported in Equation 5.8.

$$H(z) = 1/10z^3 + 1/20z^2 - 1/10z - 1/10 - 1/10z + 1/20z^{-2} + 1/10 \quad (5.8)$$

The input of the stage is a stream object of *Q14.2* type. The chose representation was the*64 bit signed* fixed point, referred here with the notation *Q57.7*, where the first number indicates the integer part (represented by 57bit, including the sign bit) and second one represents the decimal part (7 bit). The precision of this representation is equal to $2^{-7} = 0.0078125$.
The scaling produced the results of Table 5.18 and Figure 5.19.

Parameter	Value
Scaling coefficient	262144 (2^1)
Coefficient max relative error	0.0015%
Input signal #bits	$Q14.2$
Coefficient #bits	16
Required accumulator #bits	$Q57.7$

Table 5.18: EDA Preprocessing Second order Differentiator FIR results

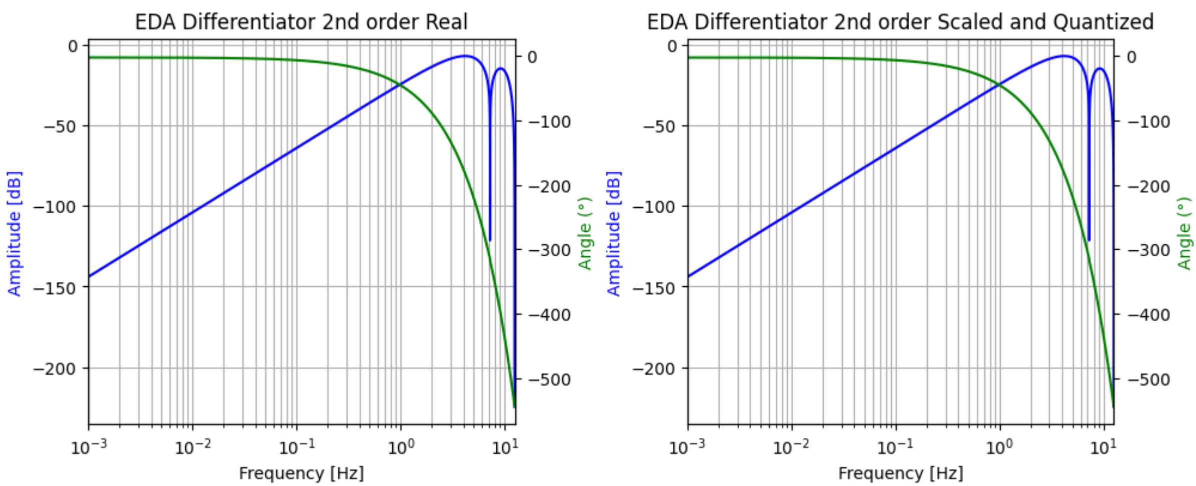


Figure 5.19: EDA Preprocessing Second order Differentiator FIR

The delay introduced by this filter is equal to $0.12s$. With the same reasoning of the zero-crossing stage it was decided to compute here the sign of the second derivative. The sign function does not introduce a further delay, since it is simply an assignment of the output data as:

- **1**: second derivative sample is positive
- **0**: second derivative sample is negative.

Figure 5.20 shows how the signals looks like after the preprocessing stage.

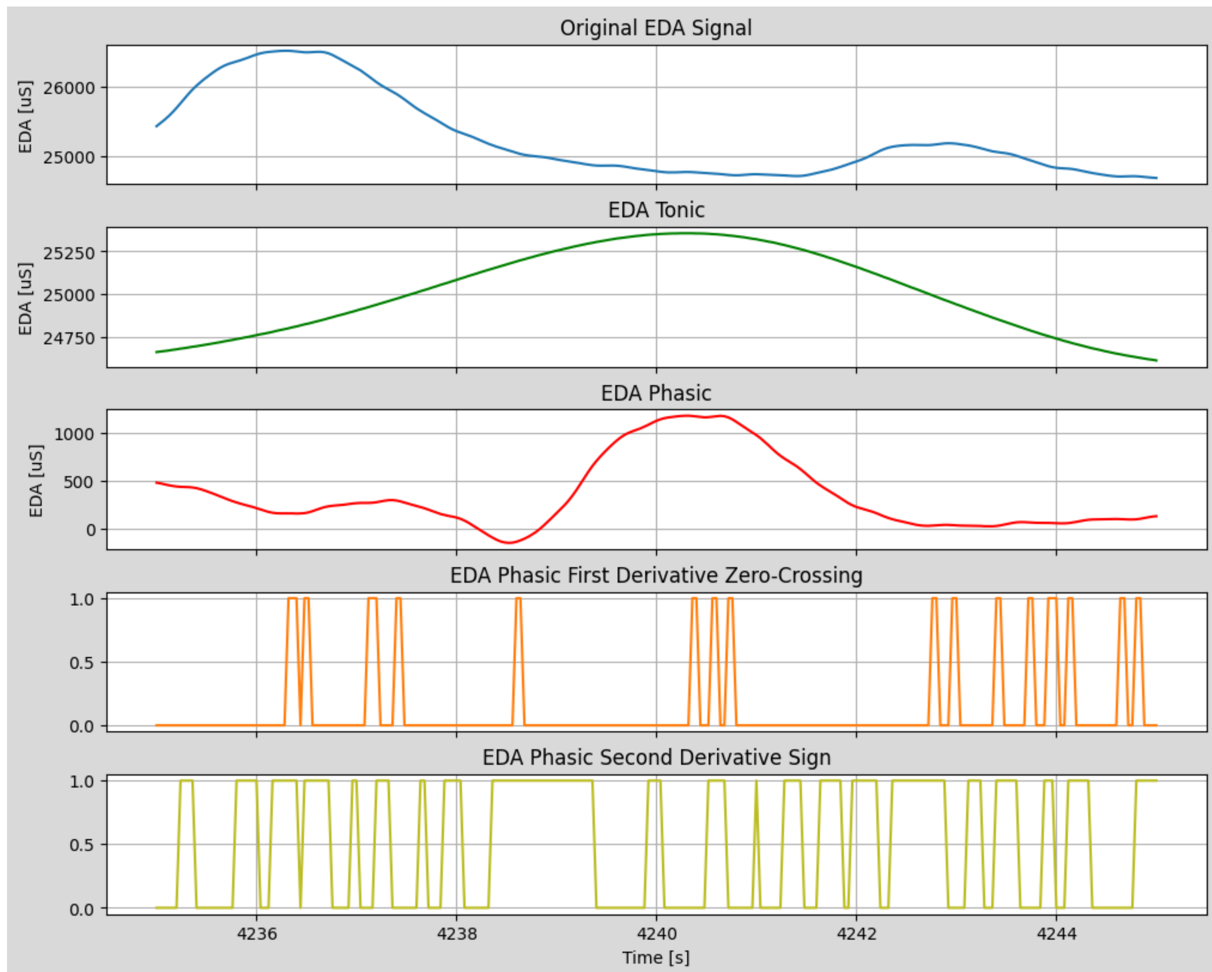


Figure 5.20: EDA Preprocessing Output

5.3. Data collector

The Data Collector stage scope is to aggregate one second of data thus providing the necessary time window shift for the processing stage. This stage, together with the processing stage internal arrays, create the same time windowing effect as in Section 3.1. The structure of the stage in shown in Figure 5.21

The stage is internally divided in ECG and EDA due to the difference in sampling frequencies. Each one of the sub-stages works in the following manner:

1. Read from the input stream the data structure produced by the previous stage (ECG preprocessing or EDA preprocessing).
2. Collect 1s of data samples, storing them in a group of internal arrays. (125 samples for ECG, 25 samples for EDA).

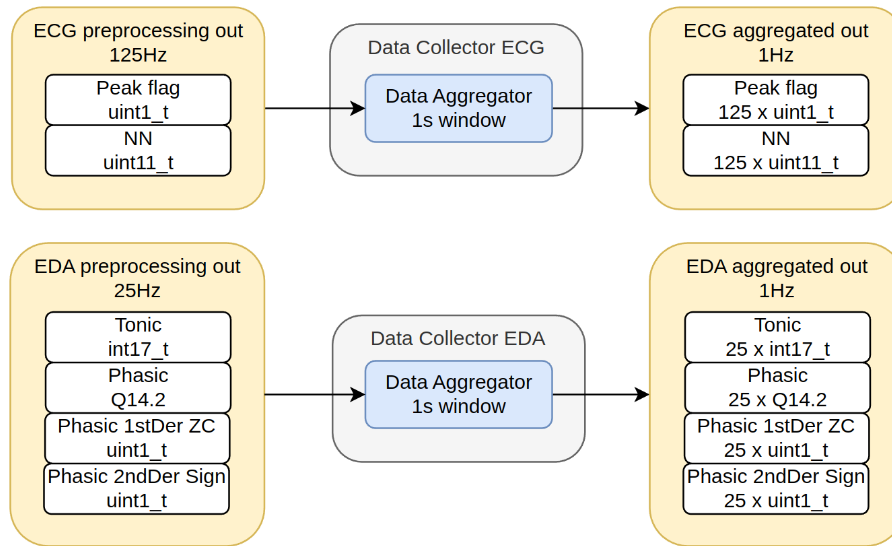


Figure 5.21: Data collector

3. Write on the output stream the aggregated data structure once every second.

The output data for each of the sub-stages is a data structure composed of the same signals generated by the preprocessing stage, just aggregated in 1-second arrays.

5.4. Processing

The Processing stage is the heart of the accelerator IP core, it extract the final signals required for features extraction from the time window of interest, save the data to memory and raise the interrupt to the processor once operations have been completed.
The stage is working at $1Hz$ frequency timed by the arrival of data from the previous data collector stage and, together with it, is performing the time windowing operation.

The Processing stage operates differently on ECG and EDA and its divided internally in two sub-stages. The general behaviour of one of the stage and each sub-stages is the following:

1. A block of 1-second aggregated data is read from the input stream coming from the Data collector. This operation is performed independently for each sub-stage.
2. The $1s$ block of data is saved into the internal array. The array is sized to store $60s$ of data corresponding to the time window of interest and works in a FIFO fashion: the new block of data is added to the FIFO array and the oldest one is removed. This operation is performed independently for each sub-stage.
3. The required operations are done on the array of data. This part is different between

ECG and EDA and will be explained in details in the next Subsections.

4. Once the processing operations have been completed the output data are written directly to system memory using the Biosignal Coprocessor AXI-4 interface. More details regarding the interfaces and the memory access are provided in Section 4.2.2 and Section 4.2.3. This operation is again performed independently for each of the two sub-stages.

The arrays internally used for the 60s of data have the size shown in Table 5.19.

Array	Data type	Sample count	Array bit size
ECG NN	uint11_t	7500	82.5Kb
ECG Peak flags	uint1_t	7500	7.5Kb
EDA Tonic	int17_t	1500	25.5Kb
EDA Phasic	Q14.2	1500	24Kb
EDA Phasic 1stDer ZC	uint1_t	1500	1.5Kb
EDA Phasic 2ndDer Sign	uint1_t	1500	1.5Kb

Table 5.19: Processing Arrays sizes

5.4.1. ECG Processing

The ECG Processing stage has the structure shown in Figure 5.22.
As explained before this stage received 1s data blocks from the Data collector, preserve the last 60s of data and process them to extract the required signals (NN array, NN counter and Cardiotach array).

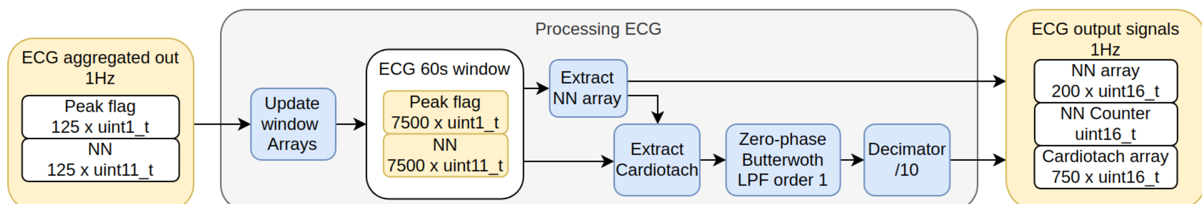


Figure 5.22: ECG processing

The different internal blocks of the stage are the followings:

- **Extract NN array** This block will use the NN continuous time signal and the peak flag signal to extract an array with the different beat-to-beat intervals found

and count them.
It works by running through the NN continuous time signal, sampling the NN continuous time signal when the peak-flag signal is set as True and increasing the NN-counter each time a peak is found in the window.
The NN array as well as the NN counter will be initialized at zero at the start, thus the resulting array will be filled with a number of valid NNs equal to the NN-counter.

- **Extract Cardiotach** This block is used to produce the Cardiotach raw signal from the NN array and the peak flag signal. The signal is generated in a stepped fashion by assigning the correct NN value at the signal between two peak flags.
  The extraction process is explained in Figure 5.23.

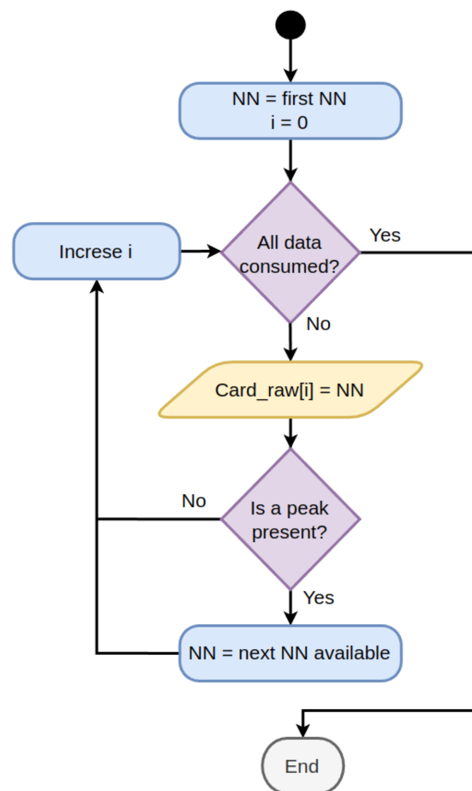


Figure 5.23: ECG processing Cardiotach extraction algorithm

- **Cardiotach Zero-phase filter** The stepped Cardiotach raw array is filtered using a zero-phase filtering to smooth-out the steps and obtain a slowly varying signal in time. The concept of zero-phase filtering was introduced before in Section 3.2.2 again in the context of Cardiotach, it can be used here since the entire signal is available beforehand.
  Details about the filter used are shown in Table 5.20 and Figure 5.24. As the filters used in downsampling and preprocessing, integer scaling was performed allowing for

the use of integer-only arithmetic.

Parameter	Value
Cut-off frequency	$1Hz$
Filter order	1
Filter type	*Butterworth*
Input signal #bits	11
Coefficient #bits	16
Scaling coefficient	$16384(2^{14})$
Coefficient max relative error	0.059%
Required accumulator #bits	25

Table 5.20: ECG Processing Cardiotach filter

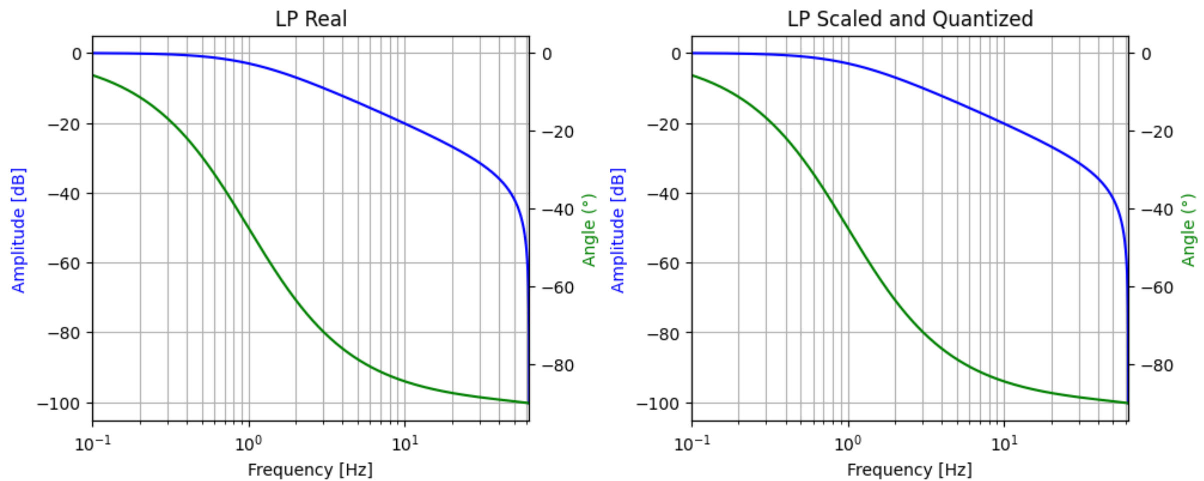


Figure 5.24: ECG processing Cardiotach filter

- **Cardiotach Decimator** Once the Cardiotach raw signal has been filtered with the zero-phase filter the obtained results can be downsampled to $12.5Hz$ using a decimator of order 10.
  The decimation step is performed by keeping a sample each ten, it is used here to reduce the array size without loss of useful information thanks to the previous filtering.

Once all the required data have been extracted, the ECG processing block will perform a memory transfer to save the results in the system memory. A single memory transfer

is issued at the end of the computations, this allow to use bursts better optimizing the AXI-4 performance.

5.4.2. EDA Processing

The EDA Processing stage has the structure shown in Figure 5.25.
As the introduction of this section explains, its input are four arrays containing each 1*s* of data aggregated by the Data Collector block, Section 5.3. The Processing stage focuses on saving those data inside a bigger structure which collects the last 60*s* of data, by discarding the oldest, and processing to extract the critical points (Peak, Point of Onset, Half Recovery Point). Then critical points, besides tonic and phasic components, are written to the output arrays.
Since this data will be elaborated from the CPU, the arbitrary precision data are converted to standard types. In detail:

- Tonic: from *int17_t* to *int32_t*
- Phasic: from *Q14.2* to *int16_t*. In order to not lose the precision of the fixed point value, before conversion the data is left-shifted by 2. It will be converted back by the feature extractor function in the Linux application
- Peak: computed from *uint1_t* data, it is *uint8_t*
- POO: computed from *uint1_t* data, it is *uint8_t*
- HRP: computed from *uint1_t* data, it is *uint8_t*

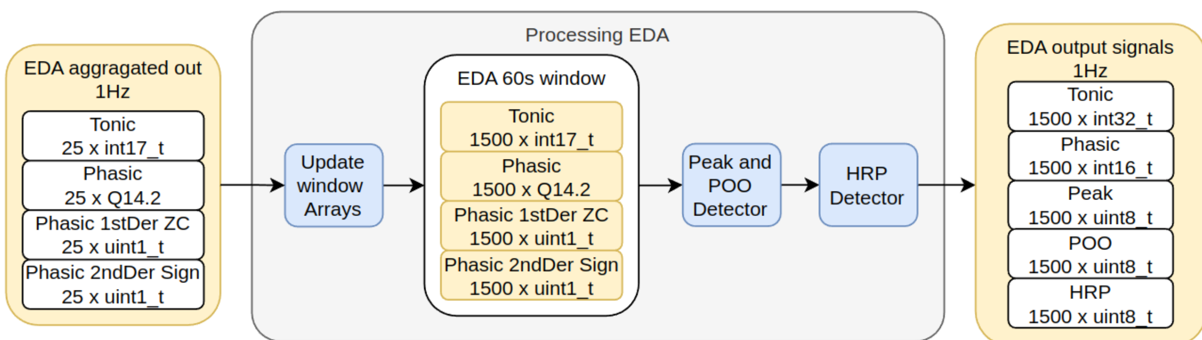


Figure 5.25: EDA processing

The two critical points detection algorithms are explained in the following paragraphs.

Peak and Point of Onset Detector With respect to the Python equivalent, the peak and point of onset detectors were collapsed in one single algorithm, designed to be more

optimized and hardware specific, as Figure 5.26 describes.
Given the $60s$ data array, the algorithm selects a section of $2s$ of input data, in which searches for an absolute maximum point that can be a peak candidate for the phasic signal. If it is found, then two threshold check are performed: the phasic value at the selected sample is compared with a fixed threshold at $0.01\mu S$; then the EDA (tonic+phasic) value at the selected sample is compared with a threshold computed on a window of $10s$, as depicted in bottom of Figure 5.26.
Once the peak point satisfies all the requirements, the algorithm starts looking for a POO, searching back from the peak to the previous one, or for a maximum range of $3.2s$.
If both the Peak and POO are found, they are saved in internal arrays of the processing stage, ready to be written to the output. It is important to notice that this procedure ensures that for each peak selected a point of onset must be found to retain its value, otherwise the candidate will be discarded.

Half Recovery Point Detector Defined as the point were the amplitude of the peak decreases by the 50%, the Half Recovery Points are searched once the previous algorithm ended and it is represented in the flow diagram in Figure 5.27. Also for this procedure, a further optimization of the Python equivalent block was done.
This algorithm takes as input the previously computed Peak and POO arrays, (in addition to the EDA components) and scrolls the $60s$ array of data. In order to compute the amplitude of each peak, it starts looking for a point of onset, savings the corresponding phasic sample value. Since every point of onset is followed by a peak, once the algorithm find a peak, it saves the corresponding half recovery point target value and enables the search of it.
Once the HRP search is enabled a threshold check is performed and the first following sample lower or equal to the half recovery point target value is picked, saving it in an internal processing block array.
It is mandatory to remember that, as described from the Electrodermal Activity studies and as also reported in Section 3.3.3, to each Peak and POO couple of points sometimes there will not correspond an HRP, especially if a second SCR begins before skin conductance levels have dropped sufficiently. In those cases, peak and point of onset are maintained, because they still describe the signal trend. The algorithm includes this kind of possibility by checking at every cycle if the sample is a point of onset, in order to not remain stacked looking for an HRP when it will not be present.

Once all critical points algorithm ended, all internal arrays are ready to be written to the output. This step is done only at the end of the stage, ensuring a single transfer to the system memory by using the burst feature of the AXI-4 interface.

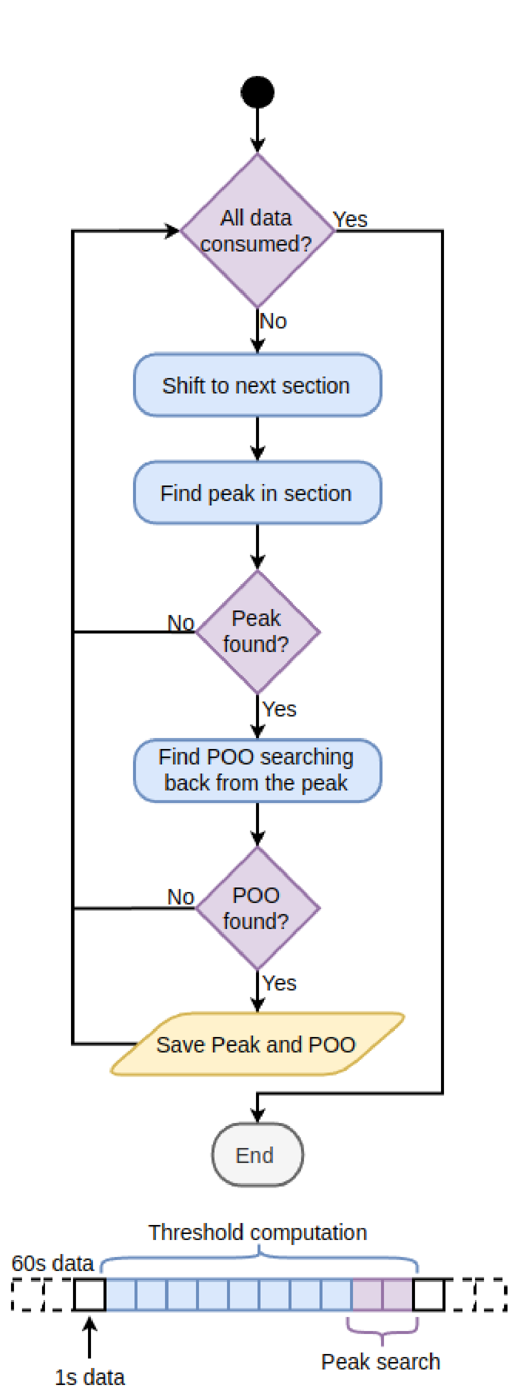


Figure 5.26: Peak and POO detection algorithm

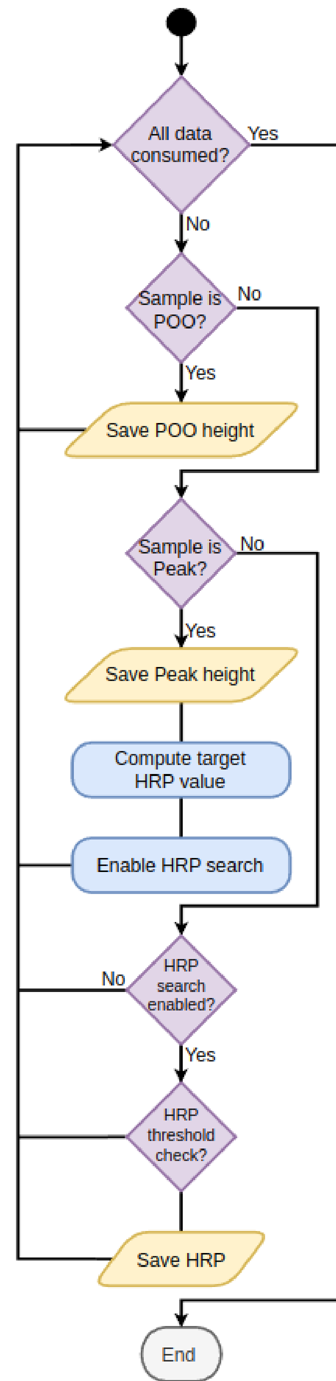


Figure 5.27: HRP detection algorithm

5.5. Interrupt generation

The Biosignal Coprocessor needs to signal to the CPU that data have been written to memory and are ready for the next processing steps, this signalling is done using an interrupt so that the CPU doesn't need to check for the data in a polling manner.

Vitis™ HLS provide a generic interrupt port on all the IP cores which are developed with it, this interrupt port can be configured through the AXI-LITE control interface to be active on different signals. Normally only two interrupt sources are available:

- **ap_done** The interrupt is raised when the IP core has completed its operations.
- **ap_ready** The interrupt is raised when the IP core becomes ready for further processing.

Due to our chosen topology, neither of those two options could be used as interrupt source. Our IP core will be put into auto-restart mode and run 500 times, each time reading a new set of input data from the ADC, before finally writing the output values to memory. For this reason an additional interrupt source was added thus creating the *valid_interrupt* signal.
The interrupt generation process is shown in Figure 5.28.

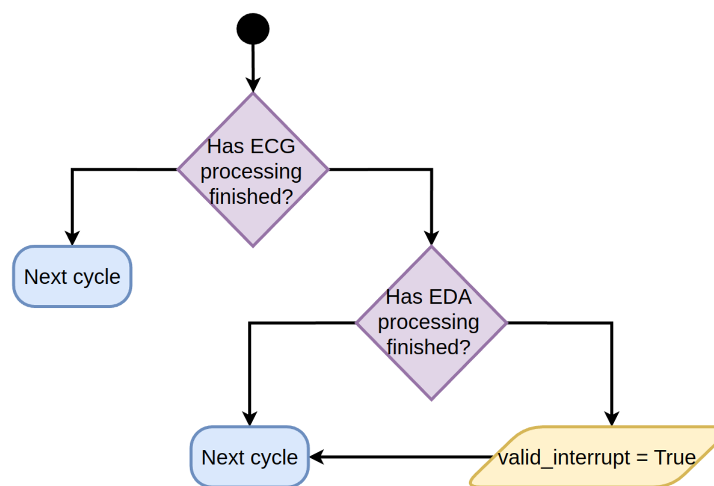


Figure 5.28: Interrupt generation for Biosignal Coprocessor

It works in the following way:

1. The *valid_interrupt* boolean signal was created and added as a register accessible through the AXI-LITE control interface of the Biosignal Coprocessor.

2. The *valid_ interrupt* signal was enabled as an interrupt source on the AXI-LITE interface.
3. Once each one of the processing sub-stages (ECG and EDA) completed its memory transfer, a Boolean status value is set to True and latched at this state for the next cycles.
4. Once both of the boolean status values are in the True state the *valid_ interrupt* is written as True thus raising the interrupt. The two booleans are then reset to False before proceeding with the next cycle.

Attention must be payed to a small detail: interrupt is raised each time the *valid_ interrupt* is written, not when a specific value (True) is written to it. For this reason the *valid_ interrupt* is written only in those cycles on which the system has completed both (ECG and EDA) output data transfer to memory.
More details regarding the validation of the interrupt generation process are explained in Section 5.6.1.

5.6. HLS design workflow

The development workflow sketched in Figure 5.29 was followed during the HLS design of the previously described stages.

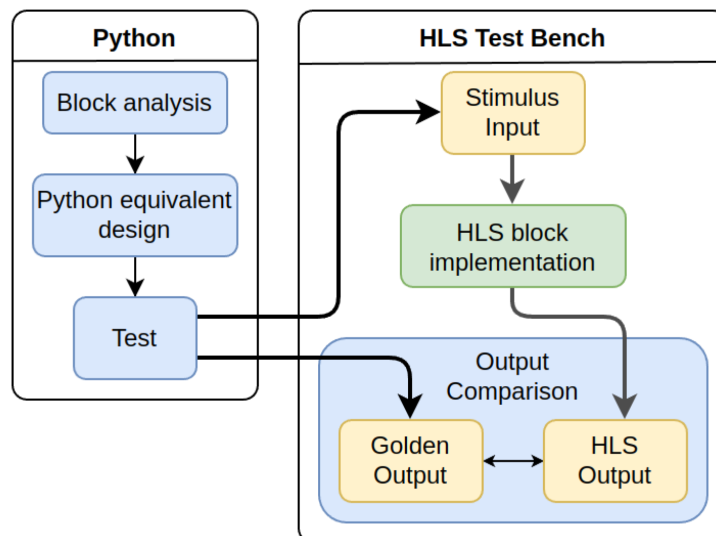


Figure 5.29: HLS design workflow

First, the stage previously designed for the Python pipeline was analyzed, extracting the part that has to be implemented in HLS, then the block was designed in Python following a

more hardware oriented approach, without using all possible array manipulation provided by the language. The advantages of this additional step are different:

- Block functionality can be developed and debugged quickly
- Possibility to test the block with real data coming from the available datasets
- Generation of stimulus input files that could be provided to the HLS equivalent implementation
- Generation of golden output files in order to compare the Python results with the corresponding HLS block output.

Starting from the Python code, the functionality of the stage was reproduced in Vitis™ HLS, making the proper hardware considerations as already described in the preceding chapters. A proper HLS test bench was then written, which:

1. Takes as input the stimulus file produced in Python
2. Collects output from the HLS implementation of the block
3. Compares the output with the golden output files produced in Python.

Once each block has passed all the development steps, the complete pipeline is reconstructed in Python, aggregating all the equivalent design already made. This brought the chance to test the complete Biosignal Coprocessor IP core and validate its functionalities.

5.6.1. Interrupt validation

As described in Section 5.5, when both ECG and EDA Processing stages complete their operations, an interrupt is raised at the output of the Biosignal Coprocessor. This interrupt is used to notify the CPU that data in the memory are ready to be read.

We tried to validate the process of interrupt generation using a Vitis™ HLS test-bench but we encountered some co-simulation errors. The test-bench was not able to run due to conditional writing on the interrupt signal which was stalling the simulation.
A custom RTL test-bench was developed in Vivado™ to overcome this issue: the entire IP core was placed inside a wrapper and the external interfaces were emulated in VHDL.

5.7. Implementation

Once all the stages were developed, tested and connected together, Vitis™ HLS was used to synthesize the required Biosignal Coprocessor IP core to be later used in Vivado™. The results of the synthesis is the IP core seen in Figure 5.30.

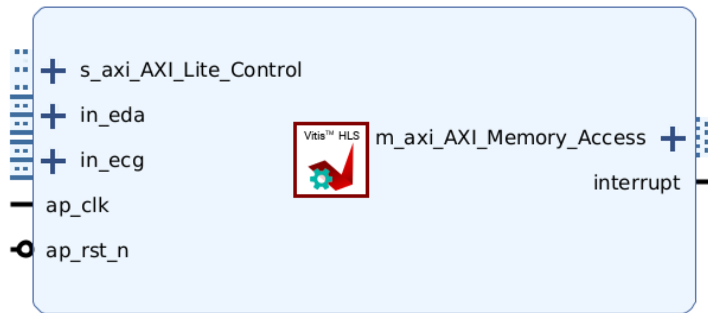


Figure 5.30: Biosignal Coprocessor IP core

Additional information regarding the connections to the IP core are provided in Appendix C.

The different interfaces of the block are clearly visible:

- **ap_clk**, **ap_rst_n** Clock and reset signal of the IP core. They will be provided by the FPGA fabric.
- **in_eda**, **in_ecg** AXI-4-Stream interfaces to receive the ADC data for the two channels.
- **s_axi_AXI_Lite_Control** AXI-4-Lite interface used to control the IP core. The internal registers can be set through this interface. This will be connected to the AXI-GP0 port of the Zynq™ 7000 processing system.
- **m_axi_AXI_Memory_Access** AXI-4 interface used to write output data directly to system memory (DDR). This interface will be connected to AXI-HP0 port of the Zynq™ 7000 processing system.
- **interrupt** Interrupt port used to signal processing completion at the CPU. This will be connected to the Zynq™ 7000 processing system IRQ port.

After the accelerator has been tested and its behaviour and performances validated, as explained in Section 5.6, the resource occupation as well as the timing performances obtained need to be compared with our requirements and constraints.
The solution settings are shown in Table 5.21.

Parameter	Value
Chosen target	$XC7Z010CLG400-1$
Clock Period	$10ns$ $(100MHz)$
Clock Uncertanty	$2.7ns$

Table 5.21: Vitis™ HLS synthesis configuration

6 | Application design

All the work explained in the preceding chapter is the first part on which consists a heterogeneous system. As complementary part, this chapter describes the application development, which collects data processed in hardware and elaborate them to extract useful data features and to run the chosen classifier. The board was equipped with a PetaLinux custom image, built specifically for this work, on which the custom application was developed.
Among the different possible methods, the development of a user-space Linux application was chosen with respect to the use of a RTOS or a bare-metal approach. The reasons are mainly:

- Despite its simplicity, high performance and almost absent overhead, the development of this kind of application with a bare-metal approach would lead to a system with limited scalability and no possibility to be directly integrated in a more complex environment
- A Real Time Operating System (RTOS) would probably have brought to almost the same result achieved by this work, since the great possibility to create multithreading application, but without the availability of device support, file-systems, network connectivity and all the other benefits you get from a general-purpose Linux OS.

The developed application structure is sketched in Figure 6.1.

The platform bitstream, extracted from the Vivado™ project, was used to build the PetaLinux image. Then the application was developed on Vitis™ IDE exploiting the options to run and debug directly on hardware.
As development starting point of the Linux application, the first step was the exploration of the device drivers described in Section 6.1, in order to correctly establish an hardware/software interaction. Once this was done, the application structure was written and tested, following the concepts described in Section 6.3.

PetaLinux It is a Linux build environment released by AMD Xilinx as a productivity layer [43], built on top of a Yocto/BitBake back-end [45]. This tool eases the development

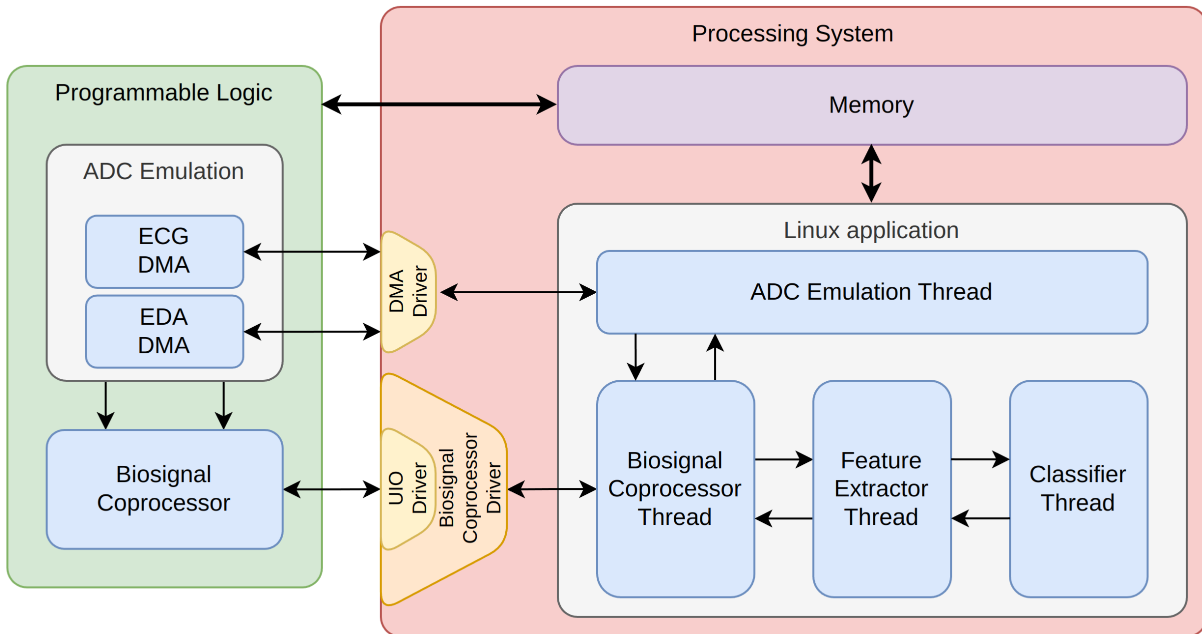


Figure 6.1: Linux application structure

of a Linux-based product with less and more specific commands, enabling evaluation of Xilinx platforms within minutes.
In the standard YOCTO flow preconfigured tested recipes are used, which can be downloaded from GitHub, to build an embedded Linux distribution targeting Xilinx reference platforms. This recipes can also be customized. The PetaLinux tools instead abstracts all the process offering to the users some automation utilities.

6.1. Peripheral drivers

Hardware and software interface each other by a low-level and very specific software called driver. The driver is the abstraction layer between software concepts and hardware circuitry; as such, it needs to talk with both of them.
In the following paragraphs all the peripheral drivers involved in this work are explained in detail.

6.1.1. Biosignal Coprocessor driver

While designing a Vitis™ HLS core, if an AXI4-Lite slave interface is implemented, a set of C driver files are automatically created. Those files provide a set of APIs that can be integrated into any software running on a CPU and used to communicate with the device via the AXI4-Lite slave interface.

The generated driver are wrapped around the UIO Linux driver, getting from the corresponding character device all the basic information to communicate with the peripheral, like the start address and the register map size, mapping user-space virtual variables to the physical memory. Then it provides specific functions, encapsulating read and write register operations.
Appendix D contains the detailed Biosignal Coprocessor register map.

6.1.2. User-space I/O driver

A device driver has basically two main task: access device memory and deal with interrupts generated by the device. In many situations, creating a custom Linux device driver is an overkill, and here it's where UIO drivers comes in. This driver bridges a user-space program with the hardware, acting as a thin layer between them.
Even if Linux is able to map physical memory of the device to an address accessible from user-space, this approach introduces security leaks and stability problem. UIO prevents those issues ensuring that no addresses outside of the device would be mapped.
A step further is made when dealing with interrupts. If the developed application has to wait for an interrupt, it simply does a blocking *read()* from **/dev/uioX**. It must to be remembered that, in a multithread application, so for instance with a thread that uses the UIO drivers, make a blocking read does not prevent the rest of the application from running. Instead only the spcecific thread which calls the *read()* function will be blocked. The function will return immediately as soon as an interrupt occurs [16].

6.2. Memory access

User-space is the area of memory that non-kernel applications run in. A user-space application, as the one designed in this work, runs in user mode, which is the non-privileged execution mode that the process' instructions are executed with.
In order to access the peripherals registers and processed data from the hardware, the user-space application have to perform a memory mapping. As explained in the Linux kernel documentation [42], it consists of mapping a kernel address space to a user address space, eliminating the overhead of copying user space information into the kernel space and vice versa. This can be done through a device driver and the user space device interface called **/dev**, by using the *mmap()* system call on the file descriptor.
First the **/dev/mem** file is open through the system call *open()*. Many flags could be used as opening options of this file. The ones used for this application are the following:

- **O_RDWR**: requests opening the file read/write operations

- **O_SYNC**: write operations on the file will complete according to the requirements of synchronized I/O file integrity completion. This flag cause the accesses to the hardware to be non-cached.

After the mapping, when accessing the mapped virtual address, the MMU translates it into physical addresses with the help of page tables. Specific flags could be used while calling the *mmap()* function. The ones used in this work are:

- **PROT_READ**: indicates that pages may be read
- **PROT_READ**: indicates that pages may be read
- **PROT_WRITE**: indicates that pages may be written
- **MAP_SHARED**: indicates that updates to the mapping are visible to other processes mapping the same region.

6.3. Multithread application

Once that all tools are introduced, this section is dedicated to the functional behaviour of the application.
The application was designed to manage multiple threads, one dealing with a specific function. With this approach it was able to divide the development in single pieces that it was possible to develop and test separately.
At first, the main thread takes care of allocating the thread data structures, maps the physical memory to those structures and call the initialize function of the application submodules. After that spawns one thread for each submodule. The sumodules involved in the application are:

- ADC Emulation
- Biosignal Coprocessor
- Feature Extractor
- Classifier.

To each of them corresponds a thread, with it own data structure composed as follows:

- **Module specific data structure pointer**: it points to peripheral specific or function specific structure which contains variables and/or substructure useful to sample the data at the input or to store the file descriptor of the UIO driver, needed for the interrupt

- **Time structure pointer**: it points to a structure describing CPU time used to catch the execution times of each single operation to make performance analysis
- **Execution time**: it is a local variable containing the thread execution time measured during the last operation
- **Run**: it is a local variable controlled by the thread itself on which the internal loop cycles. It is reset in case the thread receives an **end** condition and has to stop its execution, until then it is always true
- **Producer-Consumer control data structure**: it points to a structure which provides to the thread all it needs to implement the synchronization mechanism. The following custom structure was developed to implement a producer-consumer mechanism:
  - **Ready variable**: it is a shared variable asserted by the consumer once the data are ready to be consumed
  - **End variable**: it is a not commonly used variable, but in our case was used to propagate and **end** condition throughout a produce-consumer chain
  - **Mutex**: it is an object which provides mutually exclusive access to a shared resource.
  - **Conditional variable**: it is an object which implements a synchronization mechanism, which allows a thread to wait for an arbitrary condition, which other threads will make true.

  This structure is pointed as *input* for a consumer and as *output* for a producer. Thread which consumes data, but also produces, have both in separated structures.

By the combination of a **Mutex** and a **Conditional variable** a robust synchronization mechanism was built. The associated mutex is used to ensure that a condition can be checked atomically, and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed. In addition, the thread which is holding the mutex, at the end of its operations, could notify all the waiting threads via a notification.
In the next sub-sections all the thread are explained in detail. The chain starts by the ADC emulation, which produces the data to be consumed by the Biosignal Coprocessor. Once that it is finished its operations, it notifies back the thread which signals the feature extractor that new data are ready to be consumed. After the extraction of the features is done, then the classifier thread is woken up and the function prediction executed.

6.3.1. ADC Emulation thread

This thread focuses on the emulation of real signal coming from sensors by transferring data loaded on physical memory to AXI4-stream channels as input of the Biosignal Coprocessor. The thread has an internal loop which runs at the ADC target frequency of $500Hz$ and keeps reading from stimulus files coming from the already mentioned datasets. With this kind of emulation the goal is to have a measure of comparison between the results of the pipeline implemented entirely in Python and the non ideal hardware specific counterpart implemented in hardware. Schematically the steps the thread deals with are as follows, wrapped by a *while* loop constantly checking the **run** local variable:

1. Get the current time value
2. Read from the stimulus files new ECG and EDA raw data
3. Start the ECG DMA and EDA DMA transfers from the main memory to the AXI4-Stream channels by writing the number of bytes in the registers
4. Wait until both DMA transfers has been completed
5. Lock the output mutex, set the **ready** output flag to *1* and notify all the waiting threads, then unlock the mutex
6. Compute execution time by subtracting the current time value to the one saved at step 1
7. Sleep for $2ms$, equivalent to $500Hz$.

The cycle ends when a predefined number of rows have been read or the end is reached for one of the stimulus files. If this happens the thread asserts both **ready** and **end** output flags and notify through the conditional variable, then it expires.
It is easy to understand that this thread is the first of the chain, so it could be interpreted as a consumer. In reality no data are exchanged between this thread and the next one, since the transfer is made by the hardware. It was still preferred to give this kind of structure to improve application reliability since in this way any thread know the status of those nearby.

6.3.2. Biosignal Coprocessor thread

After the ADC Emulation, the main thread spawns the Biosignal Coprocessor. After the core has been initialized with the previous mentioned output address, the autorestart mode was enabled. Then the thread cycling accomplish these simple but fundamental

operations:

1. Wait for the input **ready** variable to be asserted from the ADC emulation thread which indicates that data have been transferred at the input of the core. This is done by acquiring the input mutex and waiting for a notification through the condition variable. Also the input **end** variable is checked here,
2. Get the current time value
3. Wait for an interrupt from the Biosignal Coprocessor hardware core. This is done by do a blocking *read()* on the **/dev/uioX**. After the function returns the thread operations goes on.
4. Lock the output mutex, set the **ready** output flag to *1* and notify all the waiting threads, then unlock the mutex
5. Clear all the interrupt flags and registers
6. Compute execution time by subtracting the current time value to the one saved at step 2

All the listed steps are wrapped around a *while* loop, constantly checking the **run** local variable. If the input **end** is asserted by the ADC Emulation thread the cycle ends, so the thread asserts both **ready** and **end** output flags and notify through the conditional variable, then it expires.
As the described behaviour explains, this thread indirectly consumes data coming from the ADC Emulation thread and as the interrupt occur produces processed data to the Feature Extractor thread. This thread is bot a producer and a consumer.

A unit-testing process, as the one used for HLS blocks introduced in Section 5.6, was followed for Biosignal Coprocessor thread where the hardware processed array were compared with golden values obtained from the Python equivalent model. This step allowed-us to verify the actual functionality of the core before its integration into the final application.

6.3.3. Features extractor thread

The features extractor thread scope is performing feature extraction on the signals received from the Biosignal Coprocessor. The ECG and EDA signals are again processed in a separate fashion and the resulting features are merged in a single data structure.
The thread internal working behaviour is depicted in Figure 6.2.

The output structure is composed of 42 *double precision floating point* values which contain all the features previously introduced in Section 3.2 and Section 3.3. Floating point

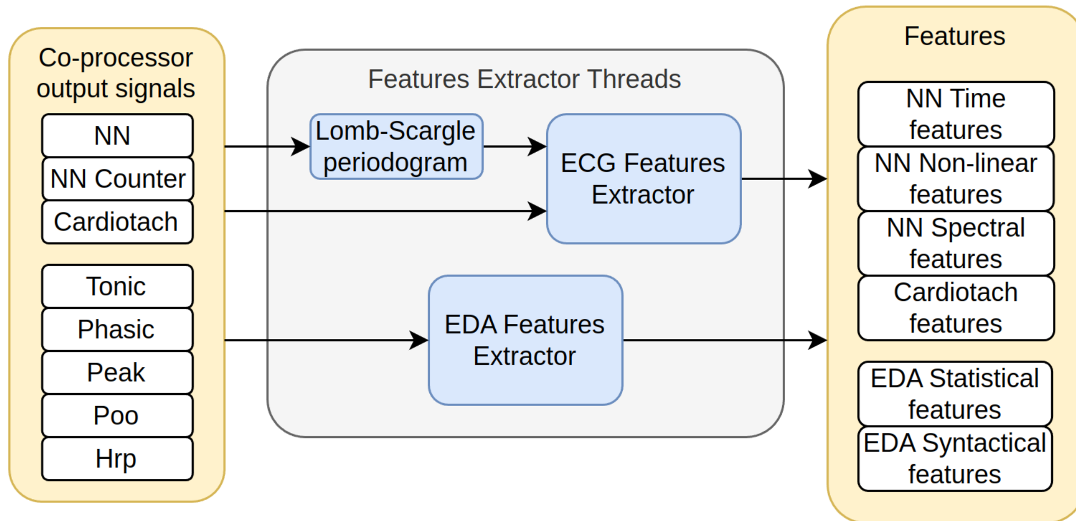


Figure 6.2: Features Extractor Thread

data are used for every feature to allow uniformity in the data passed as input to the classifier.

Internally three blocks are present, those will be explained in the next paragraphs:

Lomb-Scargle Periodogram To extract information about the spectral content of the NN signal a periodogram is needed. As previously explained in Section 3.2.3, the use of the Lomb-Scargle periodogram [18, 28] was chosen due to its higher performance for HRV analysis.
The periodogram was calculated using the algorithm developed by Townsend [34] on which the function used in the Python implementation was also based. This algorithm was initially developed to be used on GPU (Graphical Processing Units) but can easily adapted and used on a CPU, it differs from the original Lomb-Scargle works since it requires only a single pass into the input array, thus simplifying the implementation and increasing performance.

ECG Features Extractor This block perform the features extraction from ECG input data (NN, NN Counter, NN periodogram and Cardiotach). Four separate feature groups are obtained (NN Time domain, NN Spectral domain, NN non-linear domain and Cardiotach).

EDA Features Extractor This block perform the features extraction from EDA input data (Tonic, Phasic, Peak, Poo and Hrp). Two separate feature groups are obtained (Statistical and Syntactical).

The before mentioned blocks were inserted inside the same cyclic structure explained in the preceding two sections. In this case, the Feature Extractor thread consumes data coming from the Biosignal Coprocessor thread and produces new data for the next one, the Classifier thread.

The Features Extractor was separately tested using a similar methodology as the one used for HLS blocks introduced in Section 5.6. A test-bench was wrapped around the Features Extractor functions and stimulus values were applied, the output was then checked against a golden valued obtained from the Python features extractor.
This unit-testing step allowed-us to verify the behaviour of the code before its integration into the final application.

6.3.4. Classifier thread

This is the last step of the entire algorithm, this thread waits for a new set of features to be ready and then apply the stress classifier on them, obtaining as output the stress label.
The structure of the thread is shown in Figure 6.3. The depicted blocks were inserted

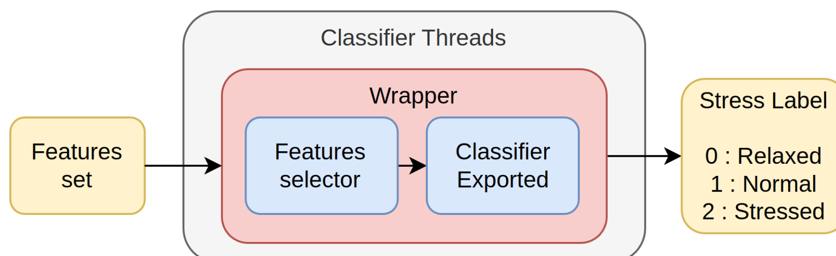


Figure 6.3: Classifier Thread

inside the same cyclic structure explained in the preceding sections. In this case, the Classifier thread consumes data coming from the Feature Extractor thread.

The classifier was developed using the Python pipeline introduced in Chapter 3 and porting to C++ was necessary for it to be used inside the Linux application. The porting operation was done in two steps:

1. **Classifier porting**: The classifier trained in Python was converted into C++ code using the *micromlgen* tool [20]. This simple tool works with sklearn classifiers and generated a *.h* file containing the synthetized classifier Class.
   This porting methodology was chosen instead of others due to the ease of use and simplicity of the obtained code.

2. **Classifier wrapper**: Once the classifier was exported in C++ the required features selection needs to be added. This is necessary because the model trained in Python only requires a certain amounts of features but here the entire feature set is available. To correctly extract only the required features a wrapper was built around the classifier to work as and adapter layer.

The porting operation was developed as a Python pipeline, the classifier code as well as the wrapper code around it are generated automatically thus removing human errors and speeding-up the deployment of a new classifier. The code-generation for the wrapper is handled using Jinja [12], an easy to use template engine for Python which is also internally used in the *micromlgen* tool.

As in the case of the Features Extractor thread, a unit-testing approach was used also used here allowing for verification of the obtained code before the entire application was completed.

6.4. Testing methodology

From the Python pipeline introduced in Chapter 3 we were able to export both stimulus and golden output files for each stage of the chain, performing unit-testing on each component, testing and debugging it before the integration with the application, as described in the preceding sections.
At the end, in order to have a complete picture of the system, a validation process of the entire chain was performed, giving as input the input data of the first stage and comparing the output of the last stage (the stress label) with golden output files from the Python pipeline.

A complete description of these results will be given in the Chapter 7.

7 | Results

The designed solution has to meet specific requirements in order to correctly assess personal stress level in real-time using the available physiological signals. They could be divided in three groups:

- **Resource constraints**
  - The synthesized design has to fit inside the target Zynq™ 7010 SoC, considering both the custom IP core and the additional circuitry needed for interconnection and routing.
- **Timing constraints**
  - The Biosignal Coprocessor logic has to run at the $100MHz$ fabric clock
  - The target system has to elaborate input signals in real-time without the need of a post-processing step
  - Process raw signal data sampled by an ADC at a frequency of $500Hz$ through a custom designed hardware accelerator
  - Compute with a target frequency of $1Hz$ the signal features giving the chance to expand them for further development
  - Estimate the stress label with a target frequency of $1Hz$ through a classifier
- **Classification constraints**
  - The machine learning algorithm has to assess the level of stress classifying it into 3 labels: rest, normal, stressed
  - Train a classifier model which doesn't require parameters tuning on the specific subject

After the development of the system the following described validation methodology was followed.

Validation methodology The results of the proposed solution described in the previous chapters were evaluated by comparing:

- The stress label predicted by a test bench developed in Vitis™ HLS in which raw data are provided to the Biosignal Coprocessor and the output elaborated signals are given to the C equivalent implementation of the feature extractor and classifier functions
- The stress label predicted by the system implemented on the target platform, in both hardware and software part.

As input data for both the above mentioned environments all the data available were given. The sections below present and justify the obtained results.

7.1. Resource occupation

The resource usage estimate of our IP core is shown in Table 7.1 and it is obtained from Vitis™ HLS. Furthermore also the percentage occupation is shown targeting the Zynq™ 7010 device we choose (more details on SoC choice in Section 4.1 and Section 4.1.2.

Resource	Used	Available	Guideline %	Used %
LUT	10474	17600	$< 70\%$	59.51%
FF	17245	35200	$< 50\%$	49.00%
DSP	27	80	$< 80\%$	33.75%
BRAM	46	60	–	–

Table 7.1: Biosignal Coprocessor resource usage estimate from Vitis™ HLS

As could be observed the used resources satisfy the guideline percentages suggested by Vitis™ HLS, thus ensuring that enough resources will still be available for interconnections and routing. In addition, the resource occupation shown is not optimized by Vivado™ on the actual chip and therefore an even lower overall resource occupation is expected.

The entire system shown in Appendix C comprising of the IP core, the blocks for ADC emulation and the interconnections was built in Vivado™ and its resource occupation is reported in Table 7.2. An increase in the number of Flip-Flops (FF) and Look-Up-Tables (LUT) can be seen with respect to the Vitis™ HLS Biosignal Coprocessor occupation estimate. This is justified by the presence of the ADC emulation blocks as well as all the different interconnections between the blocks. Instead, the number of BRAM used in the

Vivado™ system is reduced thanks to optimizations occurring in the bitstream generation process. It must be considered that, for future developments, ADC emulation blocks will be removed since they are needed only for validation purposes and so additional resources will be available for expansions.

Resource	Used	Available	Used %
LUT	12054	17600	68.49%
FF	20240	35200	57.50%
DSP	27	80	33.75%
BRAM	23	60	–

Table 7.2: System resource occupation from Vivado™

7.2. Timing results

The timings obtained from the Biosignal Coprocessor IP core synthesis in Vitis™ HLS and from the implementation of the entire system in Vivado™ are shown in Table 7.3.

Parameter	Value
CP Vitis™ HLS	$8.397ns$ ($119.1MHz$)
CP Vivado™	$9.580ns$ ($104.4MHz$)
WNS Vitis™ HLS	$1.603ns$
WNS Vivado™	$0.420ns$

Table 7.3: Biosignal Coprocessor logic timings

As shown by the obtained clock periods and Worst-Negative-Slack (WNS) for both Vitis™ HLS and Vivado™, the Biosignal Coprocessor as well as the entire system are able to run at the target clock frequency of $100MHz$ with a consistent margin.

The execution latencies of the different blocks of the system are reported in Table 7.4. As explained in Section 5.6.1 the overall cosimulation was challenging due to the interrupt, therefore to extract the timing of the entire IP core a cosimulation with the interrupt removed was used. Timings results show that the IP core can run at frequency higher than $500Hz$ in the worst case, allowing the processing of the $500Hz$ ADC inputs. The worst latency is obtained in the cycles where output data are written to system memory through the AXI-4 interface.

	Min	Avg	Max
Biosignal Coprocessor		$2ms$ $(500Hz)$	
Result [s]	$2.40\mu s$	$9.97\mu s$	$1.39ms$
Result [Hz]	$416.7KHz$	$100.3KHz$	$714.9Hz$
Feature Extractor		$1s$ $(1Hz)$	
Result [s]	$2.23ms$	$29.7ms$	$48.9ms$
Result [Hz]	$448.5Hz$	$33.7Hz$	$20.5Hz$
Classifier		$1s$ $(1Hz)$	
Result [s]	$255.5\mu s$	$438.1\mu s$	$839.3\mu s$
Result [Hz]	$3.91KHz$	$2.28KHz$	$1.12KHz$

Table 7.4: System timing results

To give a complete picture some considerations have to be made:

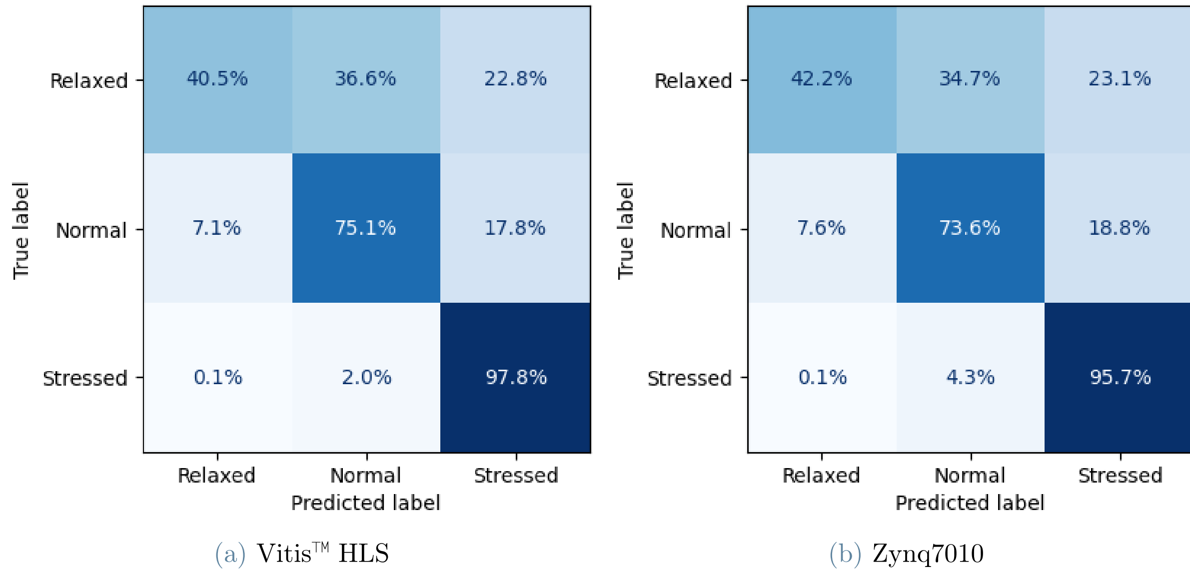
- **Biosignal Coprocessor**
  The Biosignal Coprocessor latencies were extracted from Vitis™ HLS cosimulation. For future validations, it will be better to compare those values with measured one, but the results should be very similar.
- **Feature Extractor and Classifier**
  Those timings were been measured as reported in Section 6.3 through a set of variables that captures the system time before and after the execution of the functions.

7.3. Classification results

In Figure 7.1 the results obtained from the before mentioned validation methodology, comparing the output predicted label with respect to the reference given by the dataset used.
The output label are distributed in the dataset as described in Table 7.5.

Label	Value	Percentage
Relaxed	0	30.0%
Normal	1	44.7%
Stressed	2	25.3%

Table 7.5: Input data equivalent label distribution



(a) Vitis™ HLS (b) Zynq7010

Figure 7.1: Stress label confusion matrix

As it is possible to understand, the processing pipeline running on the target platform gives valid and equivalent results with respect to the Vitis™ HLS C-simulation. This leads to the conclusion that for further developments the Vitis™ HLS environment could be preferred to accelerate the testing and validation of further developments in one of the pipeline stages.

Another result that can be noticed is that, despite the dataset is mainly composed of *normal* and *relaxed* labels, it could be challenging to make a clear distinction between those two events. Patients were indeed stimulated with different stimuli but they could be influenced by the test itself and other personal thoughts.

In addition inter-individual variability play a key role during stress assessment and to make a better evaluation of the system performance it will be better to validate it with a bigger dataset maybe composed of stressors of different types.

8 | Conclusions and future developments

The estimation of human stress level is an hot topic of the affective-computing world and is an important step in the creation of healthier and safer work environment as well as preventing accidents on high-risk jobs. In the years different approach were proposed mainly working in a post-processing fashion.

With the system developed in this work of thesis we were able to estimate the stress level in real-time using ECG and EDA, two signals easy to collect with simple and cheap sensors in a non-invasive fashion. The obtained stress estimation performances were satisfactory knowing that only data from publicly available datasets have been used. A more specific experiment for data collection would be required to obtain better estimation performances.

The proposed heterogeneous architecture, composed of hardware and software subsystems, allowed us to obtained an high-performance system without sacrificing the ability to reconfigure it.
The hardware signal processing chain can process data at high frequency directly from sensor, thus simplifying the extraction of the meaningful signals and avoiding the use of CPU computational power for those tasks.
The software application based on *Petalinux*, a versatile linux-based operating system, allowed for an easy scalability of the system and simplified the development of additional software part of the system. Furthermore, better performing features extractor and classifier can be easily experimented with and added to the system without the need of rebuilding the hardware part.

The obtained system has therefore to be considered as a platform which can be expanded on and further improved in the future. It can be integrated into wearable devices to obtain portable stress-monitoring systems which can have different application in real world scenarios. Fields spacing from medical to high-risk job environments can benefit from a system like this for varying reasons ranging from research purposes to accident prevention.

8.1. Future developments

Different parts of this work of theses can be further expanded and researched-on. In particular the following areas:

- **Real-world testing** The system can be tested in a real-world scenario using real sensors connected to internal ADC of the board and a set of test subjects. A set of experiments should be designed to produce the required stress-levels and the obtained data should be verified against the stress-level reported by the subject. Different kind of stressors should be used to explore the behaviour of the system in different conditions.
- **Filtering delays** The delays introduced by the filtering stages (Downsampling and preprocessing) can be studied in more details and improved with more complex filtering schemes. Different approaches such-as modulation and IIR filtering can be used to reduce the introduced delays.
- **Latencies validation** The IP core latencies can be further validated in the hardware implementation through the use of the Vivado™ Debugger, and additional hardware timer in the block design or by directly measuring the timings on the hardware implementation. This step wold allow to verify the results obtained by co-simulation.
- **EDA Normalization** The real-time normalization step for the EDA signal can be added in hardware further improving system performance, this will allow to obtain a more subject-independent system.
- **EDA decomposition** To improve the EDA decomposition stage in terms of delay introduced by the filters, one possible solution is modulate the EDA signal with a constant frequency one. This shifts the frequency band around $f_m od$ giving the chance to design a filter with an higher number of taps, improving the filter performance, keeping low the overall group delay. Another solution could be to move the EDA decomposition to the processing stage and apply a zero-phase filtering since all the data samples were already collected for the entire $60s$ window.
- **Feature extractor** The feature extractor in the software part can be expanded with other features which can improve the performance. Additionally more features can be added to expand the scope of the signal beyond stress detection.
- **Classifier training** Different kind of classifiers can be tested with additional datasets to improve the estimation accuracy. Ideally, a specific set of experiment should be

designed ad-hoc to produce different levels of stress using different stressors to better generalize the stress detection.

- **Sensors** The sensors to be used can be better researched to allow for an even less invasive placement on the subject. The conventional ECG sensor can be replaced with an optical sensor measuring blood-volume pulses and it can be combined into a single wearable device together with the EDA sensor.

Bibliography

[1] A. P. Association et al. Stress in america: Coping with change, 2017.

[2] M. Bachler. Spectral analysis of unevenly spaced data: Models and application in heart rate variability. *Simul. Notes Eur.*, 27(4):183–190, 2017.

[3] G. E. Billman, H. V. Huikuri, J. Sacha, and K. Trimmel. An introduction to heart rate variability: methodological considerations and clinical applications, 2015.

[4] A. Boardman, F. S. Schlindwein, A. P. Rocha, and A. Leite. A study on the optimum order of autoregressive models for heart rate variability. *Physiological measurement*, 23(2):325, 2002.

[5] J. J. Braithwaite, D. G. Watson, R. Jones, and M. Rowe. A guide for analysing electrodermal activity (eda) & skin conductance responses (scrs) for psychological experiments. *Psychophysiology*, 49(1):1017–1034, 2013.

[6] J. P. Burg. A new analysis technique for time series data. *Paper presented at NATO Advanced Study Institute on Signal Processing, Enschede, Netherlands, 1968*, 1968.

[7] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart. *The Zynq book: embedded processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 all programmable SoC*. Strathclyde Academic Media, 2014.

[8] M. E. Dawson, A. M. Schell, and D. L. Filion. The electrodermal system. *Handbook of psychophysiology*, 2:200–223, 2007.

[9] A. Golgouneh and B. Tarvirdizadeh. Fabrication of a portable device for stress monitoring using wearable sensors and soft computing algorithms. *Neural Computing and Applications*, 32:7515–7537, 2020.

[10] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2):58–66, 2001.

[11] J. A. Healey and R. W. Picard. Detecting stress during real-world driving tasks using

physiological sensors. *IEEE Transactions on intelligent transportation systems*, 6(2): 156–166, 2005.

[12] jinja. Jinja — palletsprojects.com. https://palletsprojects.com/p/jinja/, n.d. [Accessed 31-08-2023].

[13] jupyter. Project Jupyter — jupyter.org. https://jupyter.org/, n.d. [Accessed 24-08-2023].

[14] C. Kirschbaum, K.-M. Pirke, and D. H. Hellhammer. The 'trier social stress test'–a tool for investigating psychobiological stress responses in a laboratory setting. *Neuropsychobiology*, 28(1-2):76–81, 1993.

[15] knn-sklearn. sklearn.neighbors.KNNeighborsclassifier — scikit-learn.org. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html, n.d. [Accessed 24-08-2023].

[16] H. J. Koch and H. L. Gmb. Userspace i/o drivers in a realtime context. In *The 13th Realtime Linux Workshop*, 2011.

[17] S. Koldijk, M. Sappelli, S. Verberne, M. A. Neerincx, and W. Kraaij. The swell knowledge work dataset for stress and user modeling research. In *Proceedings of the 16th international conference on multimodal interaction*, pages 291–298, 2014.

[18] N. R. Lomb. Least-squares frequency analysis of unequally spaced data. *Astrophysics and space science*, 39:447–462, 1976.

[19] J. H. McClellan and T. W. Parks. A personal history of the parks-mcclellan algorithm. *IEEE signal processing magazine*, 22(2):82–86, 2005.

[20] micromlgen. GitHub - eloquentarduino/micromlgen: Generate C code for microcontrollers from Python's sklearn classifiers — github.com. https://github.com/eloquentarduino/micromlgen, n.d. [Accessed 31-08-2023].

[21] mlp-sklearn. sklearn.neural-network.MLPClassifier — scikit-learn.org. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html, n.d. [Accessed 24-08-2023].

[22] mutualInformation. Mutual information - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Mutual_information, n.d. [Accessed 24-08-2023].

[23] nb-scikitlearn. 1.9. Naive Bayes — scikit-learn.org. https://scikit-learn.org/stable/modules/naive_bayes.html#gaussian-naive-bayes, n.d. [Accessed 24-08-2023].

[24] J. Pan and W. J. Tompkins. A real-time qrs detection algorithm. *IEEE transactions on biomedical engineering*, pages 230–236, 1985.

[25] papemill. Home - papermill 2.4.0 documentation — papermill.readthedocs.io. `https://papermill.readthedocs.io/en/latest/`, n.d. [Accessed 24-08-2023].

[26] U. Rajendra Acharya, K. Paul Joseph, N. Kannathal, C. M. Lim, and J. S. Suri. Heart rate variability: a review. *Medical and biological engineering and computing*, 44:1031–1051, 2006.

[27] rf-scikitlearn. sklearn.ensemble.RandomForestClassifier — scikit-learn.org. `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`, n.d. [Accessed 24-08-2023].

[28] J. D. Scargle. Studies in astronomical time series analysis. ii-statistical aspects of spectral analysis of unevenly spaced data. *Astrophysical Journal, Part 1, vol. 263, Dec. 15, 1982, p. 835-853.*, 263:835–853, 1982.

[29] P. Schmidt, A. Reiss, R. Duerichen, C. Marberger, and K. Van Laerhoven. Introducing wesad, a multimodal dataset for wearable stress and affect detection. In *Proceedings of the 20th ACM international conference on multimodal interaction*, pages 400–408, 2018.

[30] scikitlearn. scikit-learn: machine learning in Python 2014; scikit-learn 1.3.0 documentation — scikit-learn.org. `https://scikit-learn.org/stable/index.html`, n.d. [Accessed 24-08-2023].

[31] A. Shukla and L. Macchiarulo. A fast and accurate fpga based qrs detection system. In *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 4828–4831. IEEE, 2008.

[32] R. R. Singh, S. Conjeti, and R. Banerjee. A comparative evaluation of neural network classifiers for stress level analysis of automotive drivers using physiological signals. *Biomedical Signal Processing and Control*, 8(6):740–754, 2013.

[33] svm-scikitlearn. 1.4. Support Vector Machines — scikit-learn.org. `https://scikit-learn.org/stable/modules/svm.html`, n.d. [Accessed 24-08-2023].

[34] R. Townsend. Fast calculation of the lomb–scargle periodogram using graphics processing units. *The Astrophysical Journal Supplement Series*, 191(2):247, 2010.

[35] P. Welch. The use of fast fourier transform for the estimation of power spectra: a

method based on time averaging over short, modified periodograms. *IEEE Transactions on audio and electroacoustics*, 15(2):70–73, 1967.

[36] wiki-anova. Analysis of variance - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Analysis_of_variance, n.d. [Accessed 24-08-2023].

[37] wiki-ChiSquared. Chi-squared test - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Chi-squared_test, n.d. [Accessed 24-08-2023].

[38] Wikipedia. Butterworth filter — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Butterworth%20filter&oldid=1165579362, 2023. [Online; accessed 22-August-2023].

[39] xgboost. XGBoost Python Package 2014; xgboost 1.7.6 documentation — xgboost.readthedocs.io. https://xgboost.readthedocs.io/en/stable/python/index.html, n.d. [Accessed 24-08-2023].

[40] Xilinx. AMBA AXI4 Interface Protocol — xilinx.com. https://www.xilinx.com/products/intellectual-property/axi.html, n.d.. [Accessed 25-08-2023].

[41] Xilinx. AMD Adaptive Computing Documentation Portal — docs.xilinx.com. https://docs.xilinx.com/r/en-US/pg021_axi_dma/AXI-DMA-v7.1-LogiCORE-IP-Product-Guide, n.d.. [Accessed 25-08-2023].

[42] xilinx-memoryMapping. Memory mapping 2014; The Linux Kernel documentation — linux-kernel-labs.github.io. https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html, n.d. [Accessed 31-08-2023].

[43] xilinx-petalinux. AMD Adaptive Computing Documentation Portal — docs.xilinx.com. https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide/Introduction, n.d. [Accessed 31-08-2023].

[44] xjtuecho. GitHub - xjtuecho EBAZ4205: A 5$ Xilinx ZYNQ development board. — github.com. https://github.com/xjtuecho/EBAZ4205, n.d. [Accessed 25-08-2023].

[45] yocto-project. Yocto Project; It's not an embedded Linux distribution; it creates a custom one for you — yoctoproject.org. https://www.yoctoproject.org/, n.d. [Accessed 31-08-2023].

A | Appendix A

The different datasets analyzed and used for this work of thesis are introduced here.

A.1. DRIVEDB dataset

The DRIVED database was created in 2005 following the work of Healey and Picard [11]. It has been published on PhysioNet and contains a collection of multiparameter recordings from healthy volunteers, taken while they were driving on a prescribed route including city streets and highways in and around Boston, Massachusetts.

The objective of the study for which these data were collected was to investigate the feasibility of automated recognition of stress on the basis of the recorded signals, which include ECG, EMG (right trapezius), GSR (galvanic skin resistance) measured on the hand and foot, and respiration.
The stress recognition was done in a post-processing fashion on intervals of aroungd 5 minutes.

A.1.1. DRIVEDB Sensors

Different sensors were used in the experimental setup which produced the DRIVEDB database, the sensor list is the following:

- **ECG**: Cardiac signal mesured on the chest using a modified lead II configuration to minimize motion artifacts and maximize the amplitude of the R-waves. The signal was sampled at $496Hz$.
- **EMG**: Electromyogram signal measured on the trapezius (shoulder), which has been used as an indicator of emotional stress in other researches. The signal was sampled at $31Hz$.
- **EDA**: Galvanic skin response signal, measured in two different point (on the hands palms and on the sole of the feets). Both signals were sampled at $15.5Hz$.
- **RESP**: Respiration signal, measured with a chest belt sensible to the chest expan-

sion and contraction. The signal was sampled at $15.5Hz$.

All the sensors were acquired using a FlexComp analog to digital converter system to log the required data. An additional channel was used as a "marker" to signal the transition between the different driving stages and therefore the different stress level to be expected. The data acquisition system was complemented by a set of cameras able to record the road, the driver and the environment, thus keeping trace of the stimulus encountered.

A.1.2. DRIVEDB Stress labeling

The stressors were not controlled, but the driving protocol was designed to take the driver through situations where stressors were more or less likely to occur.

Three different kind of driving stages were advised, each one able to provide a different level of stress to the driver: rest, highway driving and city driving.
The level of stress of each section was rated in a scale of $0 - 5$ with a questionnaire answered by the driver after the experiment. The questionnaire results was unambiguous and allowed us to rate the stress level in the three different driving stage as shown in Table A.1. The μ and σ values shown are the mean and the standard deviation of the score assigned for that particular stage on a 0-5 scale.

Stage	Stress score μ	Stress score σ	Stress level assigned
Rest	1.16	0.88	Relaxed
Highway	2.00	0.92	Normal
City	2.55	1.02	Stressed

Table A.1: DRIVEDB driving stages stress rating

The region in which the label was undefined,such as the transition zones between the different driving stages, were discarded by us due to their lack of useful information.

A.2. WESAD dataset

The WESAD database was created in 2018 as a a multi-modal dataset for Wearable Stress and Affect Detection [29].
This multi-modal dataset features physiological and motion data, recorded from both a wrist- and a chest-worn device, of 17 (15 once cleaned) subjects during a lab study in which different mental stimuli were applied.

The focus of this dataset was to provide the affective computing community and research environment with a standard dataset for wearable stress detection. The obtained dataset provides multi-modal high-quality data and include multiple affective states. Furthermore the dataset allow to compare the results of sensor placement between chest and wrist to better understand the requirements for future werable devices.

A.2.1. WESAD Sensors

Two different sensor placement have been used for this experiment: chest mounted sensors and wrist mounted sensor. The chest sensors are acquired using a RespiBAN system while on the wrist an Empatica E4 system is used. The different signals with useful details are listed now:

- **RespiBAN**: Chest worn sensor system. All sensor sampled at 700Hz and with 16bit ADC.
  - **ECG**: Cardiac signal measured with a sensor placed on the chest, a 3 electrode configuration is used.
  - **EDA**: Galvanic skin response signal, measured through a sensor placed on the rectus abdominis (abodomen).
  - **EMG**: Electromiography signal extracted through a sensor placed on the upper trapezius muscle on both sides of the spine.
  - **Temp**: Body temperature signal, sensor placed on the sternum.
  - **Acc X/Y/Z**: Accelerations of X/Y/Z axis, measured trough a series of accelerometers placed inside the RespiBan system mounted on the chest.
  - **Respi**: Respiration signal monitored using an inductive plethysmograph sensor included in the RespiBan unit.
- **Empatica E4**: Wrist worn sensor system mounted on the subject non-dominant hand, different acquisiton frequencies are used.
  - **BVP**: Blood volume pulse signal acquired using a plethysmograph sensor, sampled at 64Hz.
  - **EDA**: Galvanic skin response signal, sampled at $4Hz$.
  - **Temp**: Body temperature signal, sampled at $4Hz$.
  - **Acc X/Y/Z**: Accelerations on X/Y/Z axis, measured with accelerometers

inside the wrist mounted unit. Sampled at $32Hz$.

The data for the two devices are logged separately and then synchronized and merged in a single file. The synchronization is performed using a specific acceleration pattern produced at the start of the recording session.

A.2.2. WESAD Stress labeling

For the WESAD experiment the stress level applied as well as the stressors used were tightly controlled. Four different mental stimulus were applied: Baseline, Amusement, Stress and Meditation; all those stimulus will be explained in details:

- **Baseline** The subjects were sitting/standing at a table and neutral reading material (magazines) was provided. The baseline condition aimed at inducing a neutral affective state.
- **Amusement** A set of funny videos were shown to the subject aimed at creating an amusement reaction.
- **Stress** The stress stimulus was created by making the subject perform a public speaking task and a mental arithmetic task. Those two stimuli were studied in the Trier Social Stress Test [14] and have been shown to elicit stress reliably.
- **Meditation** A guided meditation was used to produce a relaxation state. The meditation was based on a controlled breathing exercise, instructed via an audio track.

The applied stimuli were validated with a questionnaire filled by the subjects after the test. The questionaire answers are considered ad subjective reports on how the participants felt during a condition and may be used to train personalised models. This was out of the scope of our work since we were interested in a real-time stress estimator which could work without any kind of personalized model.

Those different mental stimulus needed to be mapped to stress levels to be used in our algorithm, we choose to remove the amusement state as it had no direct mapping in our 3-level stress labeling. The mapping is shown in Table A.2.

Stimulus	Stress level assigned
Baseline	Normal
Amusement	-
Stress	Stressed
Medidation	Relaxed

Table A.2: WESAD stress level mapping

A.3. SWELL-KW dataset

The SWELL-KW dataset was collected as part of the SWELL project (Smart reasoning systems for well-being at work and at home) [17].
The objective of this project is to develop user-centric sensing and reasoning techniques that help to improve physical well-being (mostly in a private context) and to improve well-working (in a work context).

The dataset was collected in an experiment, in which 25 people performed typical knowledge work (writing reports, making presentations, reading e-mail, searching for information). The working conditions were manipulated with different work related stressors while a varied set of data was recorded from the subjects.

A.3.1. SWELL-KW Sensors

Different sensors as well as different data acquisition systems were used for the experiment, the data not only focused on physiological signals but also on computer interactions, postural data and on facial expressions.
Here a list of the signal available is provided, focus is given to the physiological signals due to the requirements of our work of thesis:

- **Computer interactions**
  - Mouse and Keyboard inputs
  - Application data
- **Facial expressions** Measured using a camera and the Face-reader software.
  - Head orientation
  - Facial movements

  - Action Units
  - Emotion
- **Body postures** Measured using a Kinect IR sensor and the Kinect SDK.
  - Distance
  - Joint angles
  - Bone orientations
- **Physiological signals** Recorded with a Mobi device (TMSI). Both signals were recorde at $2048Hz$.
  - **ECG**: Cardiac signal acquired using electrodes mounted across the chest.
  - **EDA**: Skin conductance measured with finger electrodes fixed with Velcro tape around the lower part of the thumb and ring finger of the participant's nondominant hand.

All the acquired data are provide merged and aligned in time between the different data acquisition devices. Additionally to the raw data, features extraction was performed on the acquired data and the resulting features are given.
For the scope of our work of thesis only the data related to the physiological sensors have been analyzed.

A.3.2. SWELL-KW Stress labeling

During the SWELL-KW experiments the 25 subject were assigned tasks related to normal office works. To ensure that the participants worked on the tasks seriously it was stated that all the required tasks had to be finished to receive the full subject fee
Three different stress conditions were applied:

- **Neutral**: The office work was executed normally, with the normal stress level it presented and without an additional stressor.
- **Time pressure**: In this case the time allowed to finish all tasks was 2/3 of the time the participant needed in the neutral condition.
- **Interruptions**: While the subject was performing its task 8 emails were sent to him. Some of those mails were relevant to one of the tasks, others were irrelevant. Some emails required a reply, others did not. In this way different distraction events were created, increasing the stress level.

For our work or thesis we decided to discar the SWELL-KW dataset due to its absence of a "Relaxed" state as well as different "Stressed" states. This was not compatible with the stress labeling we decided to use for our work, it was also not coherent with the level of stress applied by the other two datasets.

B | Appendix B

Correlation matrices for EDA and ECG features are shown in Figure B.1 and B.2. The two features set have been separated for clarity.
The maximum correlation between ECG and EDA features is 0.49.

	tonic_mean	tonic_var	tonic_signalEnergy	tonic_bandwidth	phasic_peakRiseTimeSum	phasic_peakAmplitudeSum	phasic_halfRecoverySumNorm	phasic_peakEnergySum	phasic_riseTimeAverage	phasic_deacyRateAverage	phasic_percentageDecay	phasic_noOfPeaks	phasic_highestAmplitude	phasic_highestAmplitudeRiseTime
tonic_mean	1	0.33	0.95	0.31	0.2	0.47	0.28	0.47	0.59	0.62	0.38	0.18	0.47	0.39
tonic_var	0.33	1	0.32	0.56	0.08	0.43	0.13	0.42	0.48	0.52	0.12	0.092	0.47	0.18
tonic_signalEnergy	0.95	0.32	1	0.19	0.16	0.37	0.21	0.37	0.49	0.52	0.27	0.11	0.37	0.29
tonic_bandwidth	0.31	0.56	0.19	1	0.15	0.6	0.3	0.62	0.66	0.69	0.31	0.24	0.7	0.42
phasic_peakRiseTimeSum	0.2	0.08	0.16	0.15	1	0.49	0.2	0.47	0.38	0.37	0.37	0.63	0.33	0.54
phasic_peakAmplitudeSum	0.47	0.43	0.37	0.6	0.49	1	0.3	0.98	0.92	0.88	0.42	0.017	0.88	0.47
phasic_halfRecoverySumNorm	0.28	0.13	0.21	0.3	0.2	0.3	1	0.34	0.32	0.29	0.43	0.19	0.38	0.58
phasic_peakEnergySum	0.47	0.42	0.37	0.62	0.47	0.98	0.34	1	0.9	0.85	0.43	0.06	0.91	0.52
phasic_riseTimeAverage	0.59	0.48	0.49	0.66	0.38	0.92	0.32	0.9	1	0.96	0.46	0.1	0.86	0.48
phasic_deacyRateAverage	0.62	0.52	0.52	0.69	0.37	0.88	0.29	0.85	0.96	1	0.36	0.1	0.83	0.48
phasic_percentageDecay	0.38	0.12	0.27	0.31	0.37	0.42	0.43	0.43	0.46	0.36	1	0.055	0.42	0.5
phasic_noOfPeaks	0.18	0.092	0.11	0.24	0.63	0.017	0.19	0.06	0.1	0.1	0.055	1	0.13	0.031
phasic_highestAmplitude	0.47	0.47	0.37	0.7	0.33	0.88	0.38	0.91	0.86	0.83	0.42	0.13	1	0.53
phasic_highestAmplitudeRiseTime	0.39	0.18	0.29	0.42	0.54	0.47	0.58	0.52	0.48	0.48	0.5	0.031	0.53	1

Figure B.1: EDA features correlation

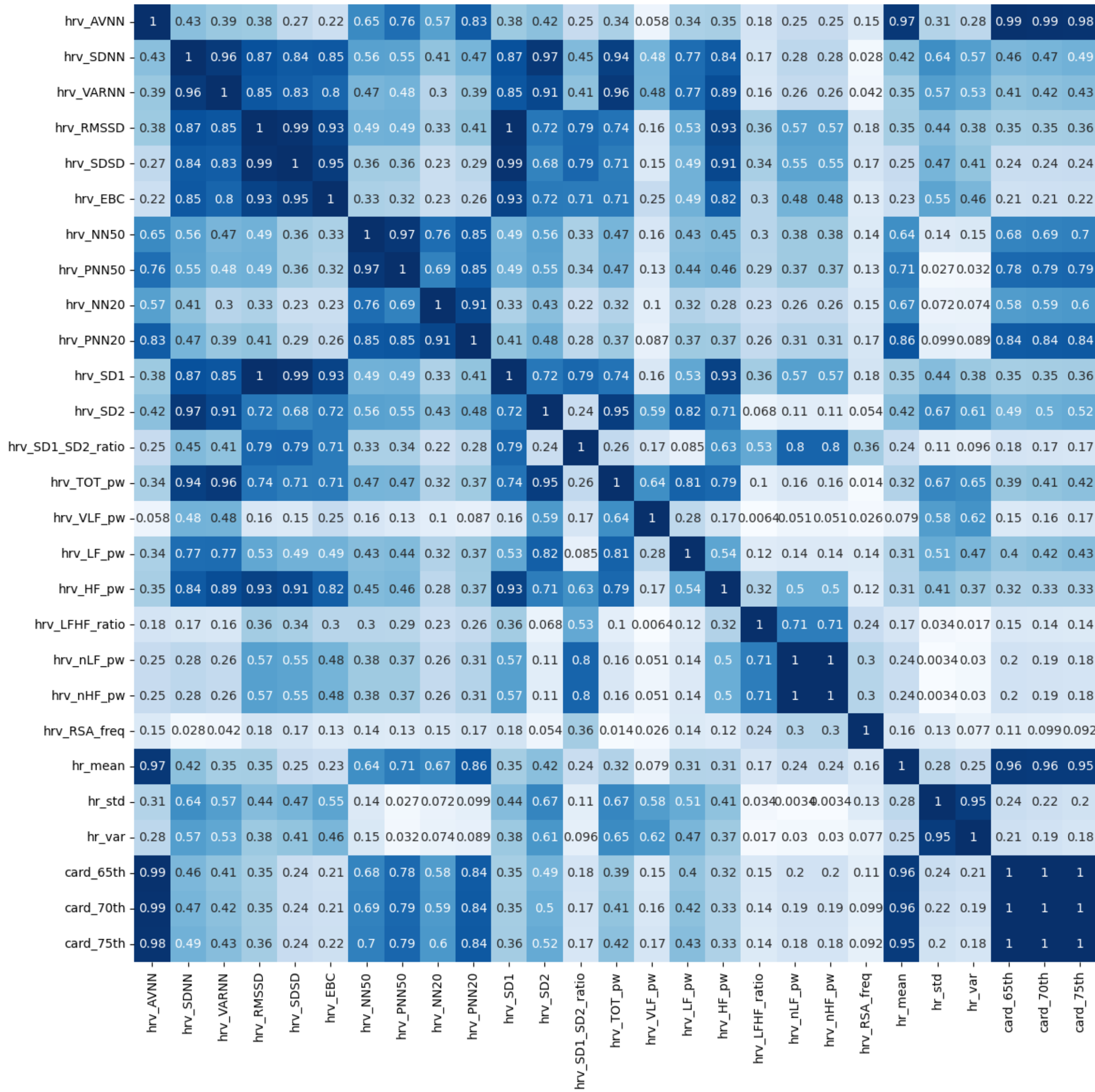


Figure B.2: ECG features correlation

C | Appendix C

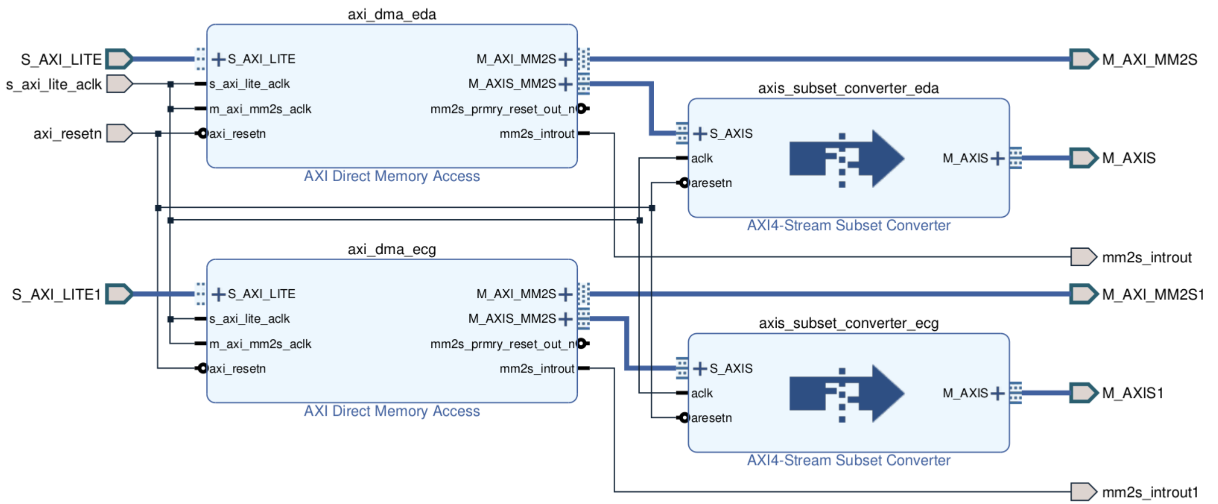


Figure C.1: ADC Emulator

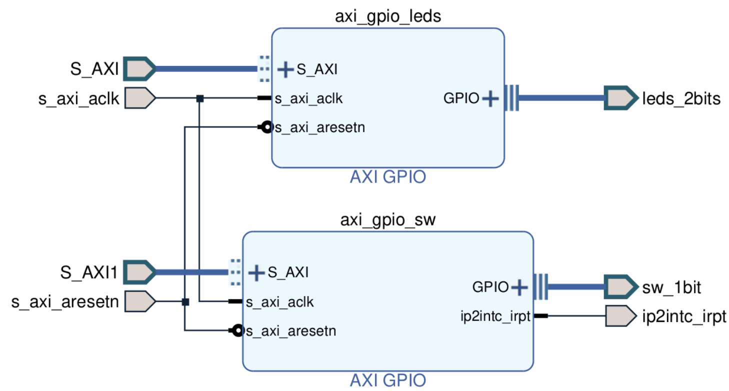


Figure C.2: Board GPIOs

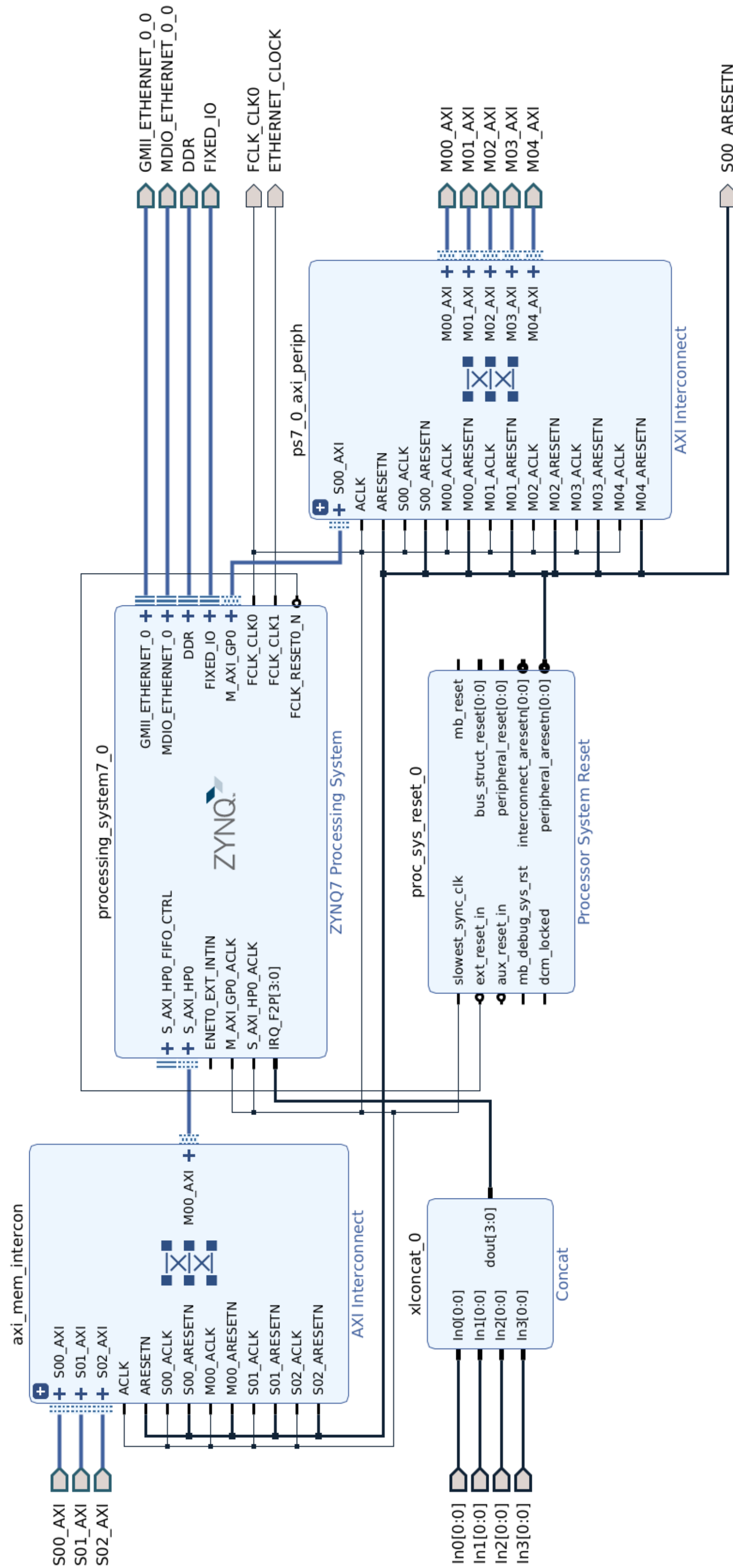


Figure C.3: Processing System and Interconnects

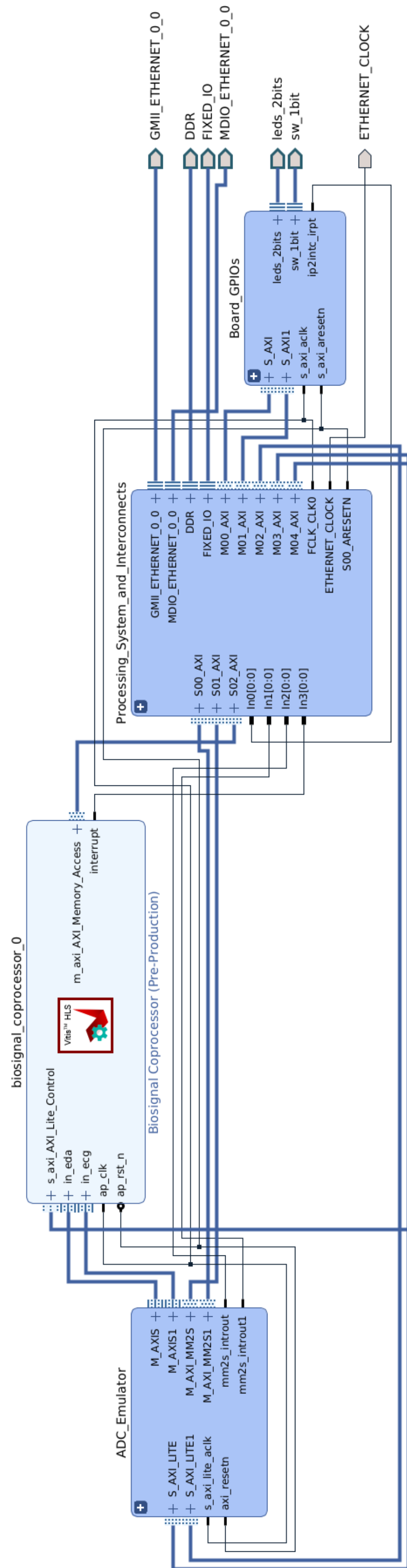


Figure C.4: Main block design

D | Appendix D

Offset	Description	Position	Operations
0x00	**Control signals**		
	ap_start	bit 0	Read/Write/COH
	ap_done	bit 1	Read/COR
	ap_idle	bit 2	Read
	ap_ready	bit 3	Read/COR
	auto_restart	bit 7	Read/Write
	interrupt	bit 9	Read
0x40	**Global Interrupt Enable Register**		
	Global Interrupt Enable	bit 0	Read/Write
0x08	**IP Interrupt Enable Register**		
	ap_done	bit 0	Read/Write
	ap_ready	bit 1	Read/Write
	valid_interrupt	bit 16	Read/Write
0x0*c*	**IP Interrupt Status Register**		
	ap_done	bit 0	Read/TOW
	ap_ready	bit 1	Read/TOW
	valid_interrupt	bit 16	Read/TOW
	Data signal output		
0x0*c*	out_eda_tonic	bit 31∼0	Read/Write
0x1*c*	out_eda_phasic	bit 31∼0	Read/Write
0x28	out_eda_peak	bit 31∼0	Read/Write
0x34	out_eda_poo	bit 31∼0	Read/Write
0x40	out_eda_hrp	bit 31∼0	Read/Write
0x4*c*	out_ecg_nn	bit 31∼0	Read/Write
0x58	out_ecg_nn_counter	bit 31∼0	Read/Write
0x64	out_ecg_card	bit 31∼0	Read/Write
0x70	valid_interrupt	bit 0	Read
0x74	valid_interrupt_ap_vld	bit 0	Read/COR

Table D.1: Biosignal Coprocessor AXI4-Lite Control register map

List of Figures

1.1 ECG signal pattern 5
1.2 Skin layers 7
1.3 EDA components 8

3.1 Overall structure 15
3.2 Overall ECG processing Chain 17
3.3 QRS complex in ECG signal 18
3.4 Pan Tompkins algorithm 19
3.5 LP and HP filters 19
3.6 Pan Tompkins algorithm signals. Green crosses placed where peaks are detected 21
3.7 NN and Cardiotach signals 22
3.8 NN and Periodogram signals 24
3.9 Overall EDA processing Chain 28
3.10 EDA normalization techniques comparison 29
3.11 EDA signal decomposition chain 29
3.12 EDA Tonic and Phasic 30
3.13 First and Second order derivative frequency response 31
3.14 Peak detection algorithm 32
3.15 Point of Onset detection algorithm 33
3.16 Half Recovery Point detection algorithm 34
3.17 EDA phasic response with highlighted critical points 35
3.18 Features set exploration 36
3.19 Cross-validation methodology [30] 38
3.20 Results of classifiers training (Normalized Data) 42
3.21 Results of classifiers training (Non-Normalized Data) 42

4.1 Zynq™ 7000 SoC structure model [7] 44
4.2 Zynq™ 7000 SoC comparison 45
4.3 Vitis™ HLS tool overview 46

4.4 EBAZ-4205 board . . . 47
4.5 System architecture idea . . . 49
4.6 ADC Emulation . . . 50
4.7 AXI DMA blocks in Direct Register Mode for ADC emulation . . . 51
4.8 System interfaces . . . 52
4.9 Zynq-7000 SoC memory map [7] . . . 54

5.1 Overall Accelerator layout . . . 57
5.2 Half-band filter design . . . 60
5.3 Half-band filter impulse response . . . 60
5.4 ECG downsampling . . . 62
5.5 ECG downsampling FIR 1 . . . 63
5.6 ECG downsampling FIR 2 . . . 64
5.7 EDA downsampling . . . 65
5.8 EDA downsampling FIR 1 . . . 66
5.9 EDA downsampling FIR 2 . . . 68
5.10 ECG preprocessing . . . 70
5.11 ECG Preprocessing LP and HP filters . . . 71
5.12 ECG Preprocessing Derivator filter . . . 72
5.13 ECG Preprocessing Moving Average filter . . . 73
5.14 ECG Preprocessing Peak Detector FSM . . . 74
5.15 ECG Preprocessing Peak detector and NN extractor . . . 75
5.16 EDA preprocessing . . . 76
5.17 EDA Preprocessing High-Pass FIR . . . 79
5.18 EDA Preprocessing First order Differentiator FIR . . . 80
5.19 EDA Preprocessing Second order Differentiator FIR . . . 82
5.20 EDA Preprocessing Output . . . 83
5.21 Data collector . . . 84
5.22 ECG processing . . . 85
5.23 ECG processing Cardiotach extraction algorithm . . . 86
5.24 ECG processing Cardiotach filter . . . 87
5.25 EDA processing . . . 88
5.26 Peak and POO detection algorithm . . . 90
5.27 HRP detection algorithm . . . 90
5.28 Interrupt generation for Biosignal Coprocessor . . . 91
5.29 HLS design workflow . . . 92
5.30 Biosignal Coprocessor IP core . . . 94

6.1 Linux application structure 98
6.2 Features Extractor Thread 104
6.3 Classifier Thread 105

7.1 Stress label confusion matrix 111

B.1 EDA features correlation 129
B.2 ECG features correlation 130

C.1 ADC Emulator 131
C.2 Board GPIOs 131
C.3 Processing System and Interconnects 132
C.4 Main block design 133

List of Tables

2.1 Available datasets 12

3.1 NN time features 25
3.2 NN spectral features 26
3.3 NN non-linear features 26
3.4 Cardiotach features 27
3.5 EDA Statistical features 34
3.6 EDA Syntactical features 35
3.7 Machine learning metrics 40

4.1 EBAZ-4205 S0C specs 48
4.2 Peripherals memory map 54
4.3 Data organization in memory 55

5.1 ECG Downsampling FIR 1 requirements 62
5.2 ECG Downsampling FIR 1 results 63
5.3 ECG Downsampling FIR 2 requirements 63
5.4 ECG Downsampling FIR 2 results 64
5.5 ECG Downsampling delays 65
5.6 EDA Downsampling FIR 1 requirements 66
5.7 EDA Downsampling FIR 1 results 66
5.8 EDA Downsampling FIR 2 requirements 67
5.9 EDA Downsampling FIR 2 results 67
5.10 EDA Downsampling delays 68
5.11 ECG Preprocessing LP and HP filters 70
5.12 ECG Preprocessing Derivator filter 72
5.13 EDA Preprocessing High-Pass FIR requirements 78
5.14 EDA Preprocessing High-Pass FIR results 78
5.15 EDA Preprocessing First order Differentiator FIR results 80
5.16 Zero-Crossing Truth table 81
5.17 Zero-Crossing K-map 81

5.18 EDA Preprocessing Second order Differentiator FIR results 82
5.19 Processing Arrays sizes 85
5.20 ECG Processing Cardiotach filter 87
5.21 Vitis™ HLS synthesis configuration 95

7.1 Biosignal Coprocessor resource usage estimate from Vitis™ HLS 108
7.2 System resource occupation from Vivado™ 109
7.3 Biosignal Coprocessor logic timings 109
7.4 System timing results 110
7.5 Input data equivalent label distribution 110

A.1 DRIVEDB driving stages stress rating 122
A.2 WESAD stress level mapping 125

D.1 Biosignal Coprocessor AXI4-Lite Control register map 136