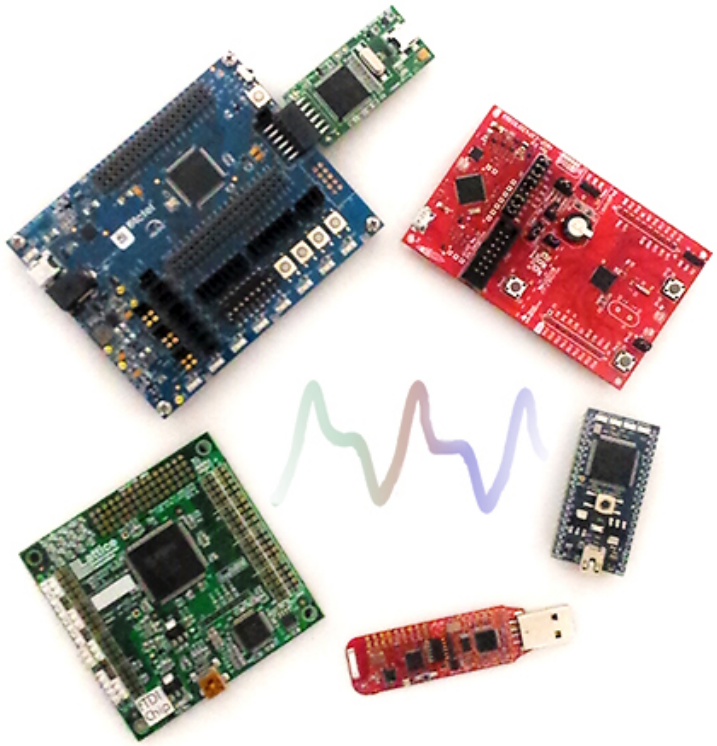
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2016



Project Title:	**Low Power Implementation of Algorithms for Mobile Healthcare Applications**
Student:	**Emad Khan**
CID:	**00733264**
Course:	**4T**
Project Supervisor:	**Professor Esther Rodriguez-Villegas**
Second Marker:	**Professor Christofer Toumazou**

Abstract

The condition of breathing cessations known as apnoea affects a large proportion of people. Since breathing is a vital activity for humans, pauses in the normal respiration cycle can lead to serious repercussions. Although this disease has severe consequences, it has been under-diagnosed and under-treated in the past. The current market solutions used for diagnosis rely on cumbersome invasive techniques which are generally unavailable to the public.

This project tackles the problem of diagnosis by creating a mobile, non-invasive apnoea detection system. An embedded system solution is formed which harnesses the power of today's evolving semiconductor industry. This report focusses on designing a high precision and low power apnoea detector on a range of different platforms. Important topics including fixed-point design, verification and synthesis are covered. The results show that it is possible to design a highly accurate detector on low-end hardware that consumes very little power. In fact the FPGA achieves a measured power consumption as low as 0.07 mW.

Acknowledgements

I would like to thank my supervisor, Professor Esther Rodriguez-Villegas for proposing this project and providing insightful feedback. I would also like to thank Dr. Sorsby for his invaluable advice and guidance throughout this project. In addition, I would like to express my thanks to all the teaching and non-teaching staff at Imperial College London.

Most importantly, I would like to express my gratitude to all my family and friends for their continuous support and encouragement. Particularly my mother and father for the love and motivation they have provided throughout my studies.

Contents

Chapter 1

Introduction

Breathing is a vital activity that is necessary to sustain human life. It is part of the respiration process which involves inhaling and exhaling air out of the lungs. Under certain physiological conditions there can be a temporary loss of external breathing. In medical terms this is known as apnoea. Since breathing is a necessity for humans, the consequences of apnoea can be dire. Loss of breathing may lead to high blood pressure, diabetes or even death [1]. Hence it is very important to detect this condition so that it may be treated.

An innovative method for detecting apnoea has been proposed in a paper written by Phil Corbishley and Esther Rodriguez-Villegas [2]. It consists of monitoring a patients breathing continuously and non-invasively. The results indicate that the breathing detection algorithm performs very well under real life conditions, achieving an impressive average success rate of about 91%. Having verified that this algorithm is behaviourally correct, it can now be converted into a hardware solution.

The main focus of this project is to translate the breathing detection algorithm into an embedded system solution. Over the years, hardware devices have evolved to become increasingly feature rich, powerful and cheaper. This effect is the outcome of cutting edge advancements in the semiconductor industry. It is also a fulfilment of Moore's Law [3]. As a result of this evolution, there are a wide range of affordable and powerful embedded platforms available nowadays. In addition, compared to software, hardware solutions have a big advantage. They generally tend to outperform their software counterparts which makes them sought after. This is particularly the case if their parallel processing nature is exploited. In real-time applications such as this project, the performance acceleration achieved with hardware is very desirable.

Micro-controller unit (MCU), field-programmable gate array (FPGA), complex programmable logic device (CPLD) and application-specific integrated circuit (ASIC) are the primary choices of platforms for this project. A solution has already been developed for the ASIC. Therefore a MCU, FPGA and CPLD solution will need to be created. The project is based on concepts from digital system design as well as from digital signal processing (DSP) and software design. It will mainly explore the various challenges faced with hardware implementation. These include designing to target specific hardware constraints, minimising power consumption and dealing with precision issues. Ultimately the MCU, FPGA and CPLD solution will be compared to each other and that of the ASIC. A conclusion will then be formed on the most optimal embedded system.

Chapter 2

Background

The background section firstly aims to establish the necessity of an apnoea detector. Furthermore it discusses the proposed solution in depth and how it fits in the medical market. The architectures of the different embedded systems are then analysed. Finally the different hardware platforms are compared and contrasted at a high-level to give a taste of the features they have to offer.

The proposed breathing detection algorithm is based on the journal titled 'Breathing Detection: Towards a Miniaturized, Wearable, Battery-Operated Monitoring System'. It is written by Phil Corbishley and Esther Rodriguez-Villegas [2]. This is a basic version of a breathing detection algorithm which has been further developed since the writing of the journal. The improved version of the algorithm is not available due to confidentiality.

2.1 Apnoea

Apnoea (also commonly referred to as apnea) is the condition which involves loss of breathing. The interrupted breathing cycles consist of periods of normal breathing followed by periods of breathing cessations. A sleep disorder directly related to this condition is known as sleep apnoea [4]. Here breathing pauses occur while a person is asleep. Sleep apnoea can be categorised into three types [5]:

1. **Obstructive sleep apnoea (OSA)** – This is the more common form of apnoea [6]. As the name suggests, it is caused as a result of there being a blockage of the airway. The breathing interruption can either be due to muscles in the throat relaxing, causing blockage of the airway occasionally. Or due to there being a partial blockage obstructing the airflow for longer periods of time (also known as hypopnoea [1]).

2. **Central sleep apnoea (CSA)** – It is different to OSA as there is no blockage of airway this time. It is caused by the temporary failure of the brain to send messages to muscles that control breathing. Thus there are no chest movements during these periods. It is seen in infants with an immature respiratory control and in adults with cerebrovascular or neuromuscular disease [6].

3. **Mixed sleep apnoea** – This is a combination of the central and obstructive sleep apnoea. Breathing cycles consist of both CSA and OSA.

According to the National Health Service (NHS) [1], around 4% of middle-aged men and 2% of middle-aged women are estimated to suffer from OSA in the UK. A significant proportion of the population is affected, highlighting the need to understand its effects as well as to develop its detection and treatment.

2.1.1 Complications

Breathing cessation can have serious consequences. As sleep apnoea can cause fragmented sleep, there tends to be daytime sleepiness [7]. This results in fatigue and slower reaction times. Research has shown that a person suffering from sleep apnoea is seven times more likely to be involved in a car accident [8]. Similarly there are also increased risks of work related injuries.

A link between sudden adult death syndrome (SADS) and apnoea has also been established in many medical papers including one written by Ondrej Ludka et al [9]. SADS has thought to have claimed more than 500 lives per annum in the UK [10]. Furthermore there is also proven increased cardiovascular risk in patients who have apnoea [11] [12]. OSA is often associated with other conditions which include high blood pressure (hypertension), insulin resistance and type 2 diabetes [13]. A study particularly focused on patients with heart failure concluded that the mortality rate increased with patients that have OSA [14].

2.1.2 Risk groups

People who are particularly at risk of developing sleep apnoea include [15]:

- Those who are overweight/obese. A build-up of fat around the neck can put a strain on neck muscles. This may obstruct breathing.
- People who have a narrowed airway. This may be due to having a narrow throat, narrow nostrils or enlarged tonsils. These lead to a restricted airflow.
- There is risk to small children who have enlarged tonsils. Also to infants that have an undeveloped/immature respiratory system (linked to CSA).
- High blood pressure, diabetes, stroke and heart failures are also thought to be linked with sleep apnoea. These may be caused due to low oxygen levels in the blood as a result of pauses in breathing.

Although apnoea can occur in people that don't fall into the categories summarised above, it is less likely. Therefore the focus of this report will be to design a sensor to target people who are at a high risk of being affected severely by apnoea. These are the people that will require diagnosing.

2.1.3 Treatments

There are a variety of treatments available to those who have sleep apnoea. These can be as simple as a change of lifestyle. Obesity can be tackled through losing weight and the controlling of diet. A reduction in smoking and drinking particularly before sleeping is also advised by the NHS [1].

Those who suffer from more moderate or severe apnoea can be treated using a continuous positive airway pressure (CPAP) device. This machine allows the person to breathe more easily. CPAP equipment achieves this by applying an increased air pressure in the throat of the patient to prevent the airway from collapsing. The air is usually delivered to the patient via a facial mask that covers the nose and the mouth. Although the mask leads to inconvenience and discomfort, the treatment has been shown to be very effective in managing and reducing the effects of OSA [16].

Surgery may also be considered in some cases. It is usually carried out as a last resort when other treatments have failed [1]. There are a number of surgical procedures available. Tonsils can be removed if they are causing blockage of airway when sleeping. This is known as tonsillectomy. Then there is adenoidectomy which consists of removing lumps of tissue at the back of the throat. Also if obesity is a serious concern, the size of the stomach can be reduced.

In summary, sleep apnoea is a very controllable disease. Even though there are a lot of treatments available to manage it, it has been poorly understood in the past [17] and as a result been under-diagnosed and treated. This highlights the need to investigate the quality of apnoea detection techniques accessible to people nowadays.

2.1.4 Apnoea detectors

Questionnaires

There are a number of questionnaires that aid diagnosis by asking a series of questions based on the patients sleeping patterns. One such questionnaire is known as the Epworth sleepiness scale (see Figure 13.1 in the Appendix). This tool is used to identify if daytime sleepiness is occurring in people who are suspected of having OSA. Another such tool is the STOPBang questionnaire (see Figure 13.2 in the Appendix). It is used in predicting an individual's chances of having OSA.

Although these questionnaires provide an easy and quick method of screening patients for the OSA disease, they are limited in their usefulness. They have shown to be weak classifiers for presence of apnoea [25] and lead to under or over-diagnosis.

Polysomnography

Polysomnography (PSG) is often referred to as the gold standard diagnostic tool for sleep disorders [18]. It is a form of sleep study which involves recording various signals during sleep. Signals recorded typically consist of brain waves (electroencephalogram or EEG), eye movements (electrooculogram or EOG), leg movements (leg electromyogram or EMG), air flow, abdominal movements, heart rate (electrocardiogram or ECG) and oximetry. These recordings are usually performed under the surveillance of a sleep technician in a sleep laboratory [19]. The technician is also responsible for attaching the vast array of sensors to the patient.

An example of a Polysomnogram is shown in Figure 2.1. All the different recorded signals can be seen on this graph. Periods of apnoea occurring can also be clearly observed. The spikes seen on the EEG signals after the occurrence of apnoea, indicate that the type of sleep apnoea experienced is not CSA. This is because the brain appears to provide a response to the lack of breathing which would not be expected to occur in the CSA condition.

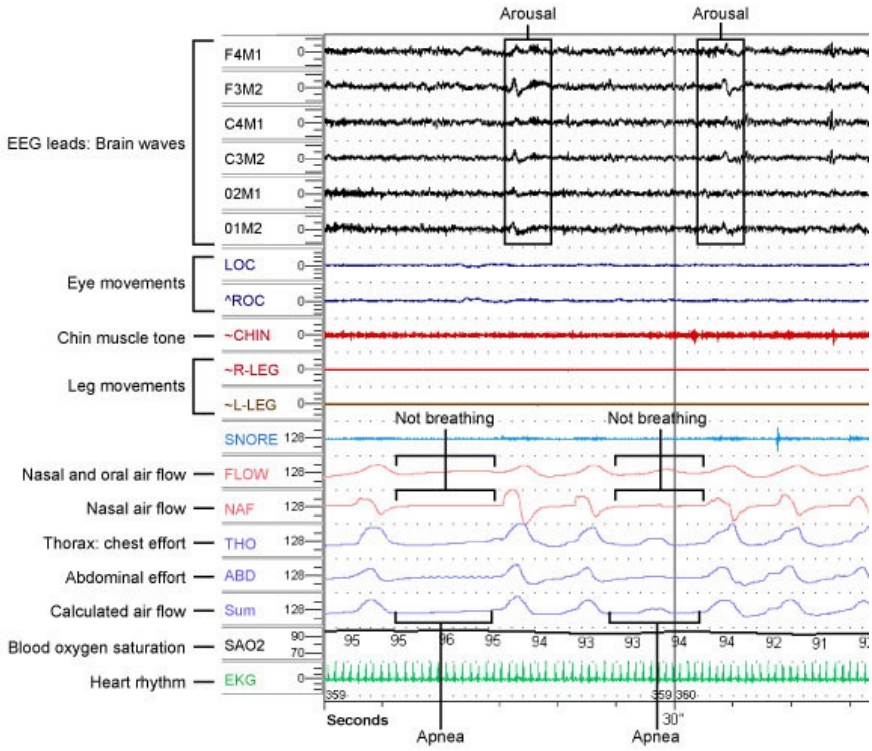


Figure 2.1: A Polysomnogram of a patient [19]

These results are examined by experts to deduce firstly whether or not apnoea is occurring and if so how severe it is. A common measurement used for classifying the severity of apnoea is the apnea-hypopnea index (AHI) [20] [21]. It indicates on average the number of apnoea or hypopnoea events that occurred during one hour of sleep. The classification system is as follows:

- **No apnoea** - AHI < 5 per hour
- **Mild apnoea** - $5 \leq$ AHI < 15 per hour
- **Moderate apnoea** - $15 \leq$ AHI < 30 per hour
- **Severe apnoea** - AHI ≥ 30 per hour

Based on the results from the PSG study and using the AHI scale, a treatment to the patient is recommended. It should be noted that the AHI scale only provides a guidance on the level of apnoea experienced. Due to the use of non-standardised equipment and different test conditions, the AHI results can vary from lab to lab [22].

The PSG method does have some drawbacks. PSGs are regarded as being expensive, they require experts (who are limited in numbers) to interpret the data and are dependent on the availability of beds in the sleep laboratory [23] [24]. This leads to them being largely inaccessible to most people. As a result, other techniques of detecting sleep apnoea have to be considered.

Oximetry

Oximetry is a method for recording an individual's peripheral oxygen saturation (SpO_2) levels. Presence of apnoea can be detected from this measurement. This is because a lack of breathing results in a drop of oxygen supply in the blood. An oximetry device can detect this change. Figure

2.2 illustrates this. When apnoea occurs, after a certain delay the SpO_2 level drops to below 80 (from above 90 when breathing was occurring).

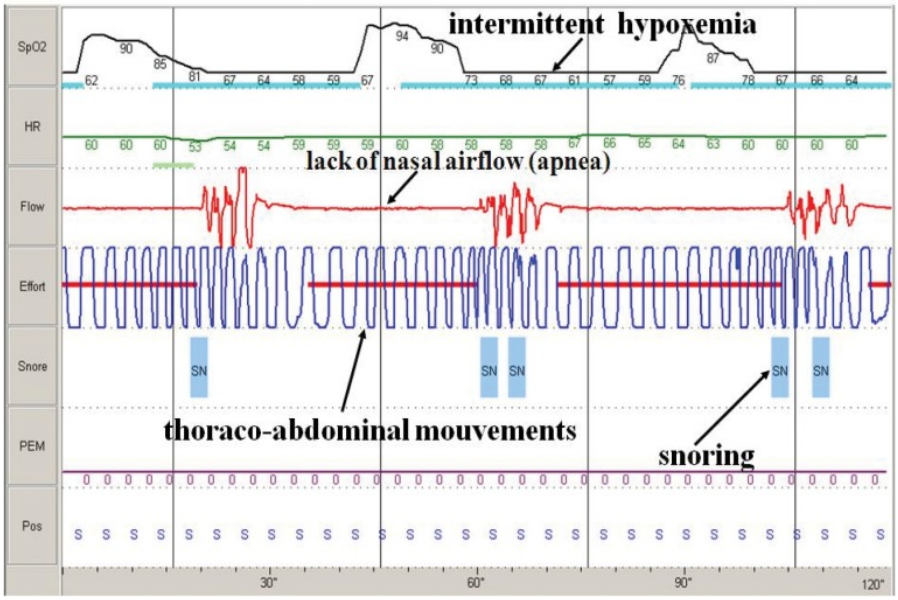


Figure 2.2: Poligraphy showing OSA and its effect on the SpO_2 levels [26]

An oxygen desaturation index (ODI) can be used to classify the SpO_2 level observed. There is no standardised ODI, an example of one is [19]:

- **Normal** - $SpO_2 = 96\%$ or 97%
- **Mild apnoea** - $90\% \leq SpO_2 < 96\%$
- **Moderate apnoea** - $80\% \leq SpO_2 < 90\%$
- **Severe apnoea** - $SpO_2 < 80\%$

As there is no strict definition of what percentage implies which form of apnoea, classification of apnoea is going to vary from person to person like that of PSG.

A pulse oximeter is a device that performs oximetry. Unlike PSG, these are widely available and reasonably priced [27]. This means that it is possible to monitor a person's breathing in the comfort of their own homes. Another advantage of this sensor is that it is fairly non-invasive. It can be attached to the fingertip or ear lobe.

There are significant downsides to a pulse oximeter. Chambrin [28] has shown that the device is very sensitive to movements and poor sensor contact. This leads to a high false alarm rate resulting in poor detection performance. It is also argued that low SpO_2 levels can be caused by conditions other than OSA [19]. In general, oximeters are used for initial screening, for a more complete diagnosis, techniques such as PSG are preferred [29].

ECG

ECG measures the electrical activity of the heart. It is said, that during episodes of OSA, the left and right ventricles of the heart are subjected to haemodynamic stress [30]. These are related to the forces that the heart has to develop to pump blood around the body. Blood pressure may rise

due to the effects of sleep apnoea [31]. The ECG will reflect these changes, thus by monitoring it, sleep apnoea can be detected.

Machine learning techniques can be applied to the ECG data in order to automate the detection process. This is in contrast to PSG where an expert is required to perform analysis on the results manually and then reach a decision on the type of apnoea experienced (through the use of a scale such as AHI). Usage of Support Vector Machines (SVM) in this field have been particularly successful. They have achieved accuracies of over 90% in identifying OSA in some cases [32] [33].

SVMs are essentially binary classifiers that find a hyper-plane which separates two class data. When the data is non-linear, the kernel trick has to be employed. This maps the data to a higher dimension where it may be linearly separated. Its parameters have to be chosen carefully as otherwise there tends to be over-fitting or under-fitting of the model. SVM also requires a lot of training data in order to perform well [34]. Consequently using machine learning can be a computationally expensive task, making it less suitable for embedded platforms or a low power solution.

Infant

Sudden infant death syndrome (SIDS) results in sudden unexpected death of a baby. Obstruction of breathing is thought to be linked with this disease. In the UK, around three hundred babies are affected by SIDS every year [35]. Babies that are born prematurely seem to be especially vulnerable [35].

There are quite a few breathing monitors available to purchase for concerned parents. These usually consist of a mat that the baby is placed on which monitors breathing [36]. This is achieved by observing chest movements. When breathing ceases, an alarm is used to alert the parents.

Summary

Breathing detectors currently available on the market generally rely on invasive or inconvenient techniques. Although PSG is considered the best diagnosis technique, it is far too inaccessible and is very invasive. Oximetry may be carried out at home, however it is unsuitable as the device is sensitive to movements, making it inaccurate for nocturnal monitoring. Detecting sleep apnoea via machine learning techniques requires a powerful computer. This implies that the monitoring device will draw a lot of power, therefore will drain batteries quickly in an embedded system scenario. For the group of people who are at a high risk of being affected by the apnoea syndrome (those with medical conditions i.e. obesity, diabetes, heart disease etc.), a better solution is needed. A non-invasive, low power sensor will need to be designed that is suitable for everyday usage.

2.2 Proposed solution

Even though there are a lot of breathing detectors available in the market, there are none that harness the growing power of embedded systems to provide a discrete non-invasive solution to the problem. This is precisely what the proposed solution in this project sets out to achieve. It envisions a miniaturised breathing detector which provides an alarm if breathing ceases. The sensor

consists of two key components. A microphone to pick up the breathing signals and a hardware device to process these signals. In the hardware, the goal is to perform digital signal processing to deduce whether apnoea is occurring or not. The algorithm that provides this functionality is now briefly explained.

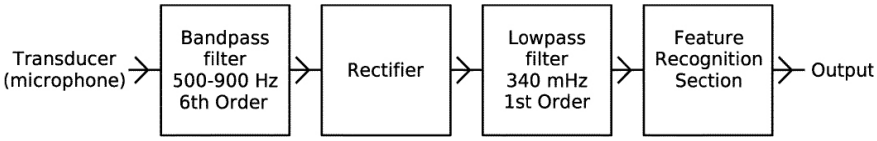


Figure 2.3: Principal components of the breathing monitoring system [2]

The first few stages of the algorithm are composed of extracting wanted features from the signals picked up by the microphone. These can be seen as the first three blocks in Figure 2.3. The 500 to 900 Hz frequency band is where the signal to noise ratio (SNR) of the breathing signal is maximised. Thus a band-pass filter is constructed. The purpose of the band-pass filter is to remove out-of-band signals so that interference can be minimised. Next the filtered signal is passed through a rectifier and a low-pass filter. The low-pass filter is used to provide the acoustic envelope signal of breathing. It represents the inhalation and exhalation cycles during breathing. An example of this is depicted in Figure 2.4.

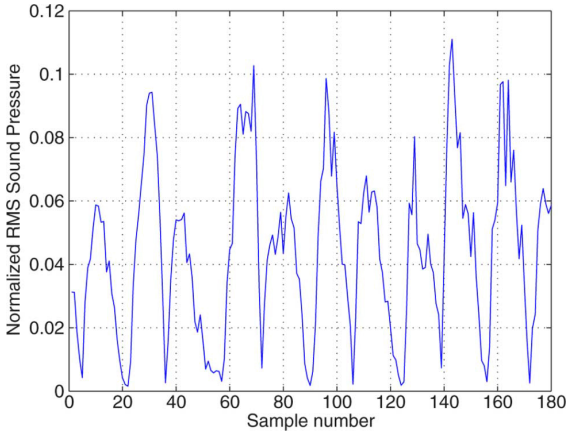


Figure 2.4: Envelope of the acoustic breathing signal [2]

Now the envelope signal can be processed to detect the presence of a breathing signal. This is the feature recognition section of the algorithm which is expanded in Figure 2.5. The bounds labelled A in the diagram are depended on the tuning of the algorithm.

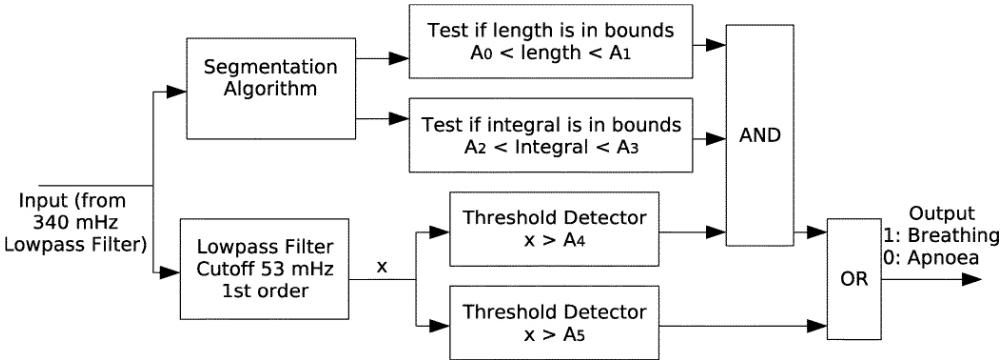


Figure 2.5: Feature recognition section of the algorithm [2]

As artefacts and noise are still present in this envelope signal, detection has to be carried out

carefully. Input to the feature recognition algorithm is split into two channels, bottom and top. The bottom channel consists of a low-pass filter which averages the envelope's power, followed by threshold detectors. Results of the bottom channel can be decomposed into three possible outcomes. If the power of the averaged envelope signal is large enough, then it may be assumed that the user is breathing. In contrast to this, if the power of the averaged envelope signal is too low then apnoea is occurring. Finally if the power of the averaged signal is somewhere in between, then the top channel is needed to make a decision.

The top channel distinguishes breathing from unwanted interferences by using the temporal characteristics of the signal. Breathing occurs in cycles of inhaling and exhaling and has a fairly low frequency ($<<$10 Hz). By measuring the length and integral of the breathing signal, it can be differentiated from noise which tends to be shorter in length and/or lower power. Length and integral are computed in the segmentation algorithm. A flowchart of this is presented in Figure 2.6.

Once the length and integral have been computed they are compared to known bounds. If they are within these bounds then a breathing signal exists otherwise it may not. The overall decision on the presence of respiration is a combination of both the top and bottom channels. This forms the whole detection algorithm.

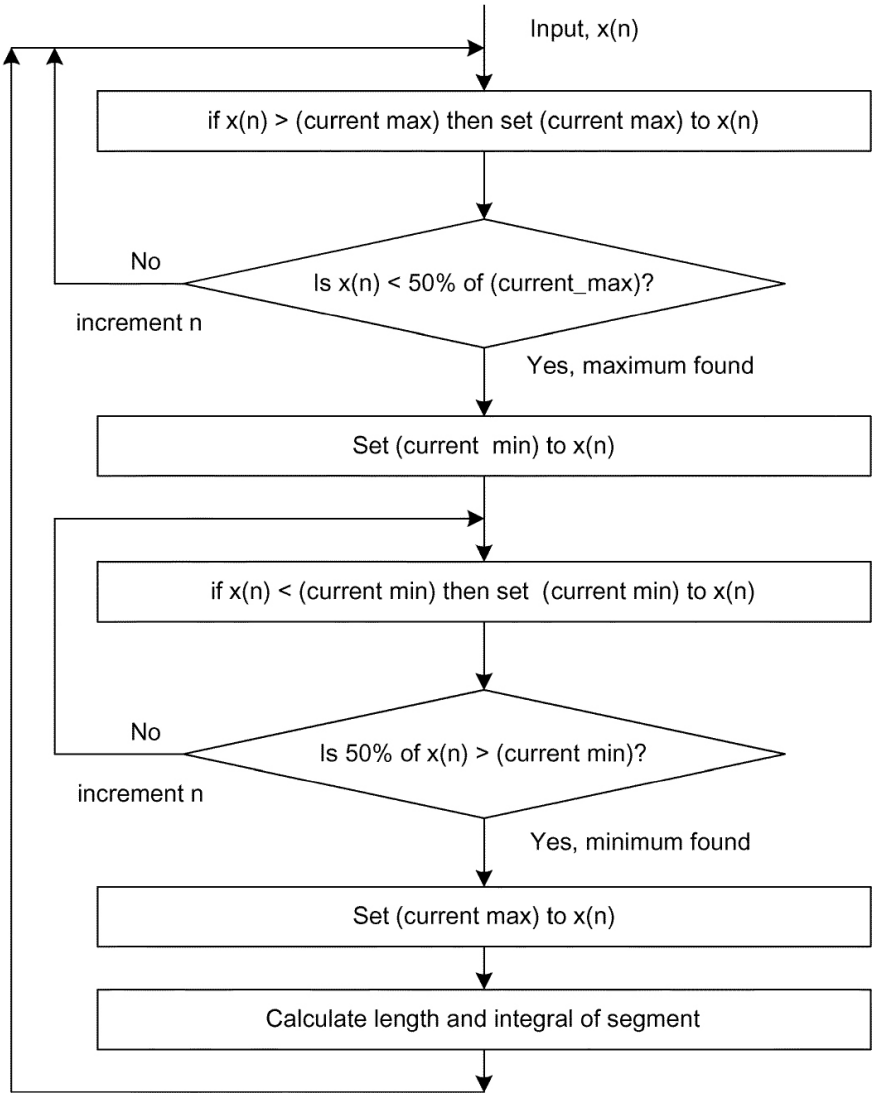


Figure 2.6: Flowchart of the segmentation sub-block [2]

2.3 Embedded systems

Breathing detection algorithms like the one proposed in Section 2.2 are based on some sort of real-time signal processing. Real-time DSP tends to be a computationally intensive task because of the large amounts of data filtering that is involved. Consequently it is important to choose a platform that is powerful enough to cope with these heavy computations. Additionally the target device also has to be energy efficient. This requirement is derived from the fact that the sensor needs to be discrete and mobile. The resulting device will thus need to be powered by a battery which will need to last sufficiently long. The battery life is left deliberately vague at this stage as no desired lifetime target has been set.

2.3.1 MCU

A micro-controller unit is a system on chip (SOC) or a small computer. Exact features that a MCU may contain, differ from manufacturer to manufacturer. The basic components they usually include are a central processing unit (CPU) which is responsible for executing instructions. Some form of memory to store and read data from. Inputs and outputs (I/Os) to interface the MCU to the outside world. The high level design of the Motorola 6801 MCU is shown in Figure 2.7. This is a very early micro-controller which was manufactured in 1974 [39]. It contains all the basic features mentioned before. According to its data-sheet, it is an 8-bit MCU which has 2048-Bytes of memory and a maximum frequency of only 1 MHz [40].

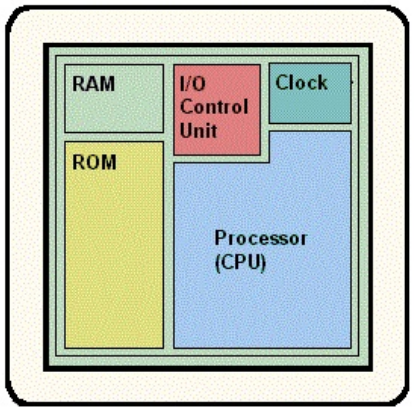


Figure 2.7: Motorola 6801 MCU [37]

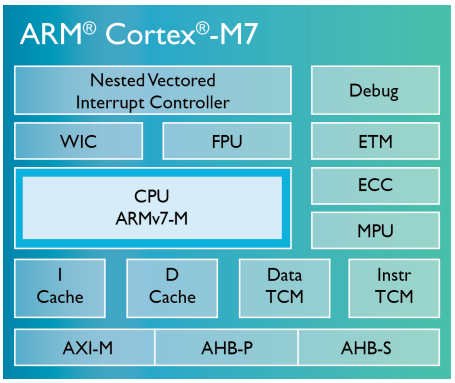


Figure 2.8: ARM Cortex-M7 MCU [38]

Nowadays MCUs are used everywhere. From cars to washing machines and mobile phones. It is estimated that the number of MCUs in the world is forecasted to surpass thirty billion in 2016 [41]. Prices of MCUs on average have been falling [41], making them more affordable to use. Clock speed of these devices has also increased. There are many MCUs available that offer frequencies well above 100 MHz as well as bigger memories of 1024 kB+ [42]. They have become efficient too. Texas Instruments (TI) claims that their MSP430F2001 chip can consume as little as 300 μA (measured at 1 millions of instructions per cycle (MIPS)) of current in active mode [43].

Not only have the MCUs become bigger and faster, they now also contain a lot of useful and powerful features. Figure 2.8 displays a modern MCU. This is produced by the company ARM which markets this product as being suitable for embedded applications [38]. Some of its features include a floating point unit (FPU) which is very useful in arithmetic (this is heavily involved in DSP). It contains debug and trace logic which is helpful in rapid prototyping. Furthermore a wake

interrupt controller (WIC) is included. This can be programmed to wake up the MCU when the processor needs to perform a task. Otherwise the MCU can sit in a low power state.

High performance, powerful features and energy efficiency offered by MCUs make them a good platform to implement the breathing detector on.

2.3.2 Hardware motivation

Hardware devices provide the ideal platform for real-time DSP problems. Inherent parallelism of hardware solutions is one of the key advantages over software. Here hardware is configured to specifically perform only the desired tasks, which leads to most of the work being completed in parallel. By doing so, a high performance boost is achieved over software. Many papers demonstrate this result including one written by Paul Graham and Brent Nelson [44]. In this paper the run-time of software was found to be four times longer than that of hardware for the same algorithm. It is also interesting to point out that the software solution ran at a clock speed which was more than ten times higher than the FPGAs. This shows the scale of the increase in performance.

Also in embedded systems, only the hardware required for the intended application is powered. As an example, a software solution may run on a general purpose MCU. Taking a micro-controller based on the reduced instruction set architecture (RISC). It will usually include an instruction pipeline which will dissipate power. This pipeline will consist of stages where instructions are fetched from memory, decoded and finally executed. A dedicated hardware solution will directly perform the desired task without the need for an instruction pipeline. Thus it will minimise the power consumption.

A combination of high performance and power saving are the major motivations for implementing the proposed algorithm in hardware. The next few subsections discuss the architecture of the hardware platforms (ASIC, FPGA and CPLD) that are used in this project.

2.3.3 ASIC

An application-specific integrated circuit or ASIC is a chip that has been developed for a specific use. Taking this project as an example, an ASIC has been developed to perform a specific task which involves DSP. This customisability means that only the hardware that is needed for this task is implemented in the chip. As a result, the ASIC can be smaller and more efficient than a general purpose non-specific piece of hardware such as FPGA. In fact in a study carried out by Ian Kuon and Jonathan Rose [45], it was concluded that on average an FPGA consumed fourteen times more dynamic power than an ASIC. Also the ASIC is said to be thirty-five times smaller than an FPGA.

ASIC chips are manufactured in a foundry on very thin silicon wafers. They consist of many mask layers that are composed of transistors and wirings. These implement complex circuitry. The developing of an ASIC is not a cheap process. Non-recurring engineering (NRE) costs, where the chip is specified, designed and prototyped are quoted to be over £500,000 [46] in some cases. This is a substantial investment and thus makes ASIC use only suitable when it is mass produced. The NRE cost does vary depending on the type of ASIC used.

All ASICs can generally be categorised into three different types. These are full custom, standard

cell based and gate array based. A full custom design involves designing some or all of the logic cells such as NAND gates and the wiring (interconnects) needed to join different cells together. A full custom ASIC is shown in Figure 2.9. Having full control over the design can result in a higher performance of the chip. However this process requires more skill and time. A paper written by Curt F Fey et al [47] highlighted that full custom ASICs require nearly double the time to design than other techniques. Thus unsurprisingly this is the most expensive type of ASIC design.

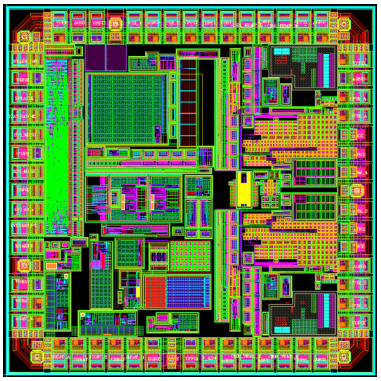


Figure 2.9: Full custom ASIC chip [48]

Then there is the standard cell based design technique. Here pre-designed logic cells are used instead of custom ones. As the logic cells have already been verified, they can be implemented straight away which results in a reduction of time and expense. Finally there is the gate array based ASIC design. This consists of logic layers that have already been defined and only allows for customisation on how the interconnect layers are connected. Although flexibility is reduced, the NRE costs and time to market are made smaller.

In addition to the high costs incurred in the manufacturing and designing stages, there are also other significant disadvantages in developing an ASIC. Firstly since the ASIC is non-reprogrammable, if a bug is found in the design, the costs of solving the bug will be huge as the wafer will probably have to be fabricated again. Then there is the long time to market. It estimated that it can take around 11 months [49] to bring a full custom ASIC to market. In conclusion, although an ASIC chip can implement the desired solution very efficiently, its development can be quite resource heavy. Bringing an ASIC to market is an expensive and time consuming process.

2.3.4 FPGA

Field-programmable gate arrays are integrated circuits. They are based on an array of configurable logic blocks (CLBs), that are connected together using reconfigurable interconnects. This wiring combined with these CLBs allows complex combinational functions to be implemented. The key advantage of FPGAs lies in their re-programmability. Unlike ASICs, FPGAs can be programmed by the designer after being manufactured. This reduces the NRE costs and simplifies the design cycle.

An FPGA is made up of lots of logic elements (LEs). An example of which is shown in Figure 2.10. A LE is typically composed of a look-up table (LUT) (which implements various logic equations), a flip-flop and a multiplexer at the output of the block. The exact structure of this LE varies from vendor to vendor. In addition to this, nowadays FPGAs offer extra features. The Virtex-6 LXT [50] FPGA by Xilinx has DSP, FIFO and multiplier blocks embedded in the device, making it very powerful.

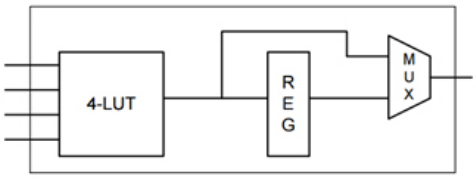


Figure 2.10: FPGA logic element [51]

As FPGAs become cheaper and increasingly powerful, more and more industries are utilising them. Consequently it comes as no surprise that the FPGA market has been growing. According to market research [52], the FPGA industry is forecasted to be worth over a massive £6 billion in 2020. Another benefit of FPGA over ASIC is the large availability of electronic design automation (EDA) tools. These ease the process of designing a digital system on an FPGA. All these factor make the FPGA a very attractive platform.

2.3.5 CPLD

A complex programmable logic device is another type of integrated circuit. It is similar to an FPGA in the sense that it is also reprogrammable. However its architecture differs. It is composed of macro-cells which can implement Boolean expressions in the form of sum of products. Small amounts of large blocks are included in a CPLD making it a coarse-grain structure. On the contrary, FPGAs contain lots of little LEs (fine-grain structure).

The largest commercially available CPLD from Xilinx is the 'CoolRunner-II'. It offers a maximum of 512 macro-cells [53]. Logic density of a CPLD is therefore much less than that of an FPGA. The same CPLD also has a low standby power of 28.8 μW, making it suitable for low power applications.

2.3.6 Comparison

	ASIC	**FPGA**	**CPLD**	**MCU**
Cost	Very high	Low	Low	Very low
Features	Unlimited	Limited	Very limited	Limited
Performance	Very high	High	Low	Very high
Size	Unrestricted	Restricted	Very restricted	Restricted
Power consumption	Very low	Low	Low	Low
Developing time	Very long	Short	Short	Very short

Table 2.1: Comparison of different hardware platforms

Table 2.1 summarises the differences between the four platforms. Without a doubt the ASIC is the ideal platform where performance, power and features are concerned. Its major downfall lies in the cost and time required to develop and bring it to market. This makes it unsuitable for prototyping. In this region, the FPGA, CPLD and MCU offer better substitutes. This is the reason why these devices have been chosen as the target platforms for this project. They provide the ideal balance between development time and performance.

Chapter 3

Project Specification

The main aim of this project is to translate the breathing detection algorithm into a software and hardware solution. The target platforms will be fairly small and low power. Both the FPGA and CPLD will have a modest amount of system gates available. While the MCU will have a limited amount of memory. As a result one of the focus of this project will be to create an efficient design to minimise the hardware usage. Another area that this project will explore is the power consumption of the hardware and MCU solutions. Since the sensor is intended to be portable, it is essential that its battery lasts sufficiently long for convenience.

In the evaluation section of this project, the various embedded solutions created will also be compared with the ASIC. The ASIC will form a benchmark for both power and performance. From these broad general aims, a narrower specification needs to be created. The following subsections discuss the specific requirements of this project.

3.1 Design specification

The input signal will be provided by an analogue to digital converter (ADC). It will be signed and 10-bits in size. The usage of more bits provides an opportunity to increase the precision of calculations. High precision will lead to a reduction of false negatives or positives in the detection of apnoea making it vital. A downside to the increase in bits will be the rise in gate usage (or memory and cycles in MCU). This may result in the design not fitting on the target hardware. A compromise will therefore need to be investigated and found between the precision of calculations and usage of hardware.

Additionally a real-time implementation is also desired in the final deliverable. This is because breathing monitoring is continuously required to diagnose the apnoea condition. The hardware will need to be capable of dealing with input signals sampled at a frequency of around 2205 Hz. If it is unable to meet these requirements then it may miss some signals. This can have disastrous consequences as the algorithm may not detect apnoea when it is occurring, rendering the sensor completely useless.

Since the sensor is portable, it needs to be powered by a battery. The type of battery will depend on the power usage of the embedded system design. Once the hardware is designed and its power consumption is measured, the type of battery required to power the device will be researched.

An embedded system also needs to be scalable. In the case for this project, parameters such as the

coefficients of filters may be modified in the future as more optimal ones are discovered. This provides motivation for the creation of reusable packages to allow for easy modification of parameters. In the next subsections the deliverables from this projects are stated and explained.

3.2 Three channel filtering deliverable

The original apnoea detection algorithm contained just one channel (see Figure 2.3). This consists of a band-pass filter, rectifier, low-pass filter and a feature detection section. Initially this project will investigate the possibility of implementing three channels of filtering instead of just one. This increase in functionality may translate into more hardware, thus the feature detection section won't be included in this deliverable. Figure 3.1 illustrates the initial prototype. It has largely the same structure as the proposed algorithm, except for the inclusion of a decimator at the end of each channel. The purpose of this is to reduce the sampling rate of data at the output. The band-pass filters are required to be 4th order, while the low-pass filters are 1st order.

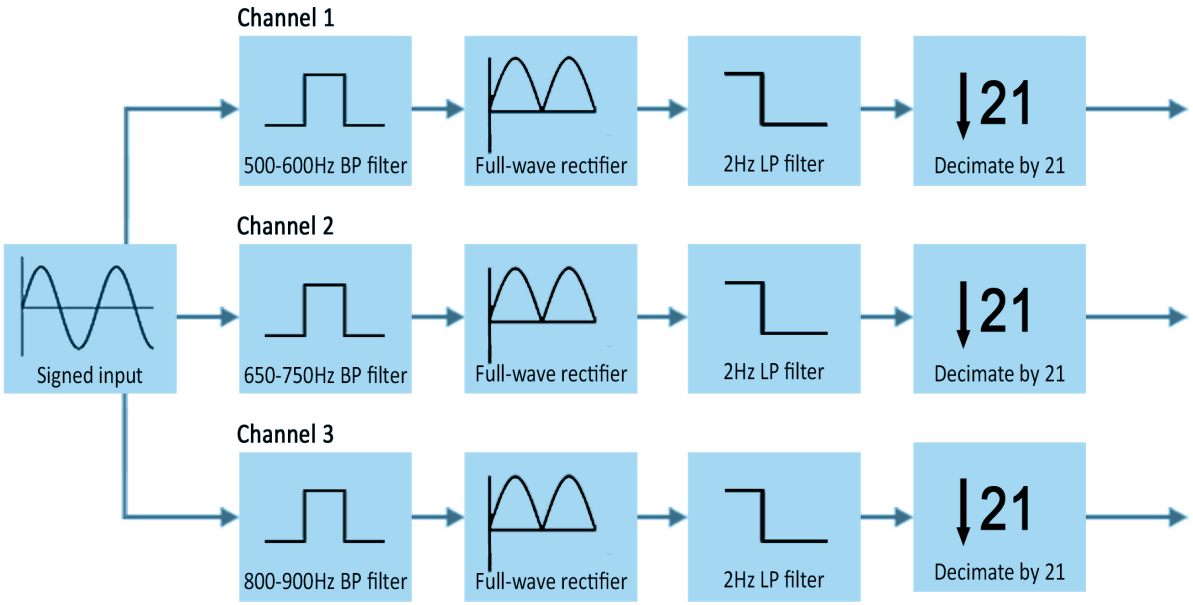


Figure 3.1: Illustration of the initial project specification

This design will be implemented on both the CPLD and FPGA. It will give a demonstration of the capability of the two devices. If the design is implemented with sufficient hardware to spare on the devices, then the feature recognition section of the algorithm will also be included at the end of each channel. This forms the deliverable in Section 3.3. Otherwise a reduction in channels will need to be considered.

3.3 Full breathing detector deliverable

This deliverable is completely dependent on the results from the initial prototype in the previous section. If the CPLD and FPGA are capable of incorporating the feature detection section, then the algorithm illustrated in Figure 2.5 will be translated into hardware. This will be implemented at the end of the three channels of the FPGA and CPLD. It will form the highlighted hardware

solution section shown in Figure 3.2. The three channel initial filtering stages and the breathing detection algorithm will also then be developed for the MCU.

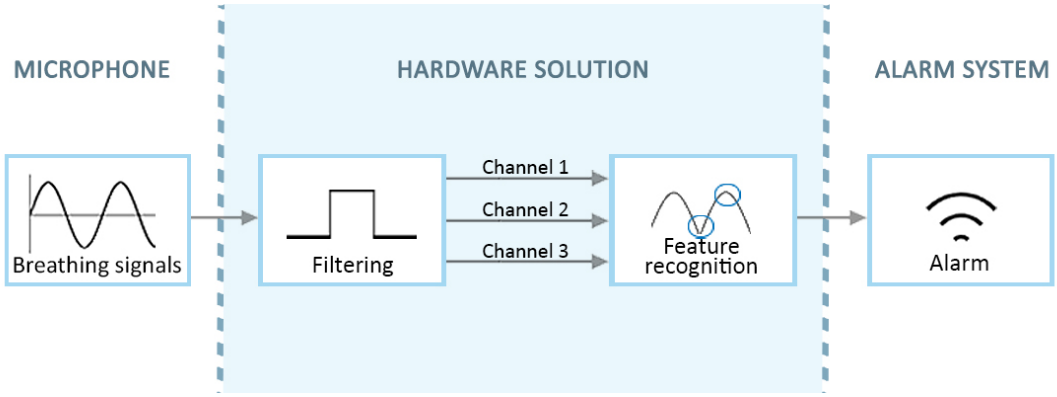


Figure 3.2: Three channel full breathing detection system

On the other hand, if the three channel algorithm doesn't fit onto either of the FPGA or CPLD then only one channel will be used (see Figure 3.3). The proposed algorithm (see Figure 2.3) will be exactly followed in this case. A MCU solution will also be developed following the one channel design.

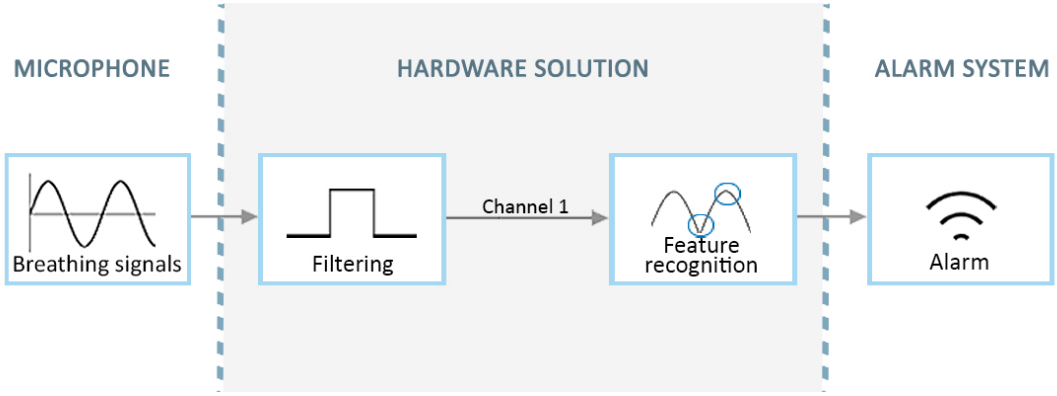


Figure 3.3: One channel full breathing detection system

In all cases the power consumed and the performance of the solutions (where possible) will be measured and compared against each of the target devices and the ASIC. In the worst case scenario, if the complete breathing detector is unable to be converted to hardware (or fit on the MCU) then documentation will be provided of the effort and how this problem can be tackled and solved in the future.

Chapter 4

Chosen Hardware

In this section, technical specifications of the chosen FPGA, CPLD and MCU are analysed. Examining the devices will give an idea of their capabilities. Furthermore these devices are also compared and contrasted against each other.

4.1 CPLD

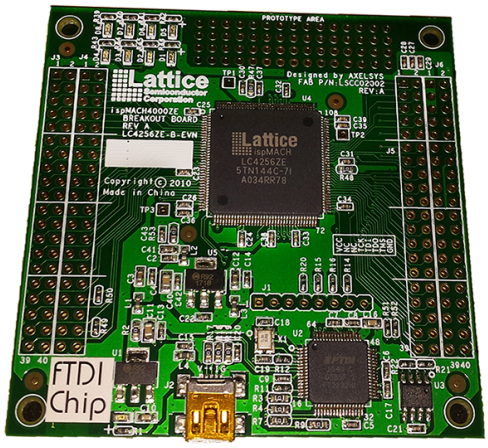


Figure 4.1: CPLD - ispMACH 4256ZE Breakout Board

The chosen CPLD board for this project is the 'ispMACH 4256ZE Breakout Board' by Lattice Semiconductors [54]. This is shown in Figure 4.1. It costs £19.94. The board has an 'LC4256ZE-5TN144C' CPLD embedded on it. The CPLD chip costs around £10.00 [55]. This CPLD is particularly adapted to provide a low power consumption. Large macro-cells and routing hardware can be seen in the functional block diagram of the system (Figure 13.4). The specification of the board including the CPLD are taken from their datasheet [56] and are summarised in a list underneath.

- LC4256ZE-5TN144C CPLD
  - 256 macro-cells (equivalent to around 450 FPGA LUTs)
  - 16 LE

  - 144-pin thin quad flat pack (TQFP) package
  - 96 input/output (I/O)
  - 200 MHz max frequency
  - 0°C to 90°C operating temperature
  - 5.8 ns max propagation delay (tpd)
  - 1.8 V operating voltage
  - 10 μA typical standby current
- 5 MHz on-board oscillator
- USB interface for programming and debugging
- Optional Joint Test Action Group (JTAG) interface
- 8 Light Emitting Diodes (LEDs)

4.2 FPGA

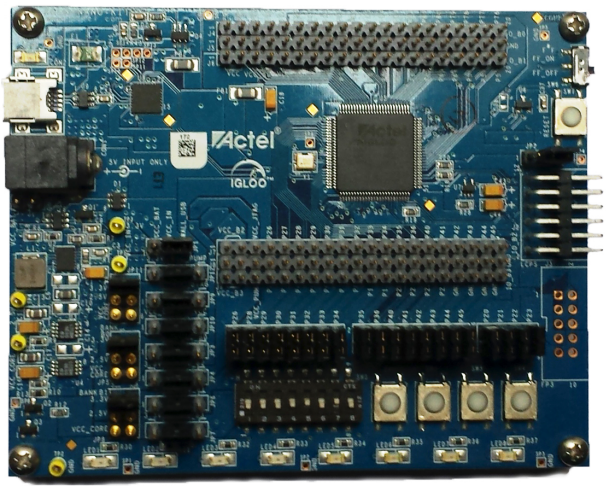


Figure 4.2: FPGA - IGLOO nano Starter Kit

Microsemi's 'IGLOO nano Starter Kit' [57] is the chosen FPGA development board for this project. The board can be seen in Figure 4.2. The cost of this product is £65.45. It features an 'AGLN250V2-VQG100' FPGA chip. The FPGA chip itself costs only about £14.00 [58]. Like the CPLD device, this FPGA has been designed to minimise power usage. It has a unique architecture where the core comprises of 'VersaTiles' (see Figure 13.5). A 'VersaTile' can be configured as a LUT, D-flip-flop or a latch. Efficient use of the FPGA fabric is promoted this way. Some of its noteworthy specifications taken from the datasheet available on Microsemi's website [59] are summarised below.

- AGLN250V2-VQG100 FPGA
  - 250,000 system gates (equivalent to 2,048 macrocells)
  - 6144 VersaTiles

  - Around 3000 equivalent LE
  - 100-pin TQFP package
  - 68 I/O
  - -20°C to 80°C operating temperature
  - 36 kB RAM
  - 250 MHz max frequency
  - 8 × 4,608-bit Blocks
  - Integrated Phase Locked Loop (PLL)
  - 1 Kb FlashROM
  - 1.2 V to 1.5 V core voltage support
  - 24 μW typical standby power with Flash Freeze Mode
- 20 MHz on-board oscillator
- USB interface for programming and debugging
- 8 LEDs
- Low Cost Programming Stick
- 5 push button switches
- Jumpers for battery/voltage/current options
- JTAG pins

4.3 MCU

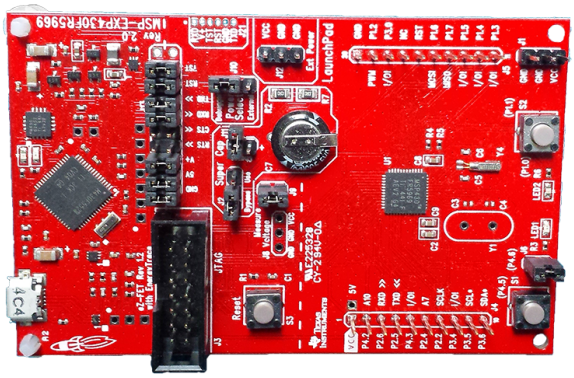


Figure 4.3: MCU - MSP430FR5969 LaunchPad Development Kit

A low power MCU development kit is selected for this project. This is TI's 'MSP430FR5969 LaunchPad Development Kit' [60] and is shown in Figure 4.3. The board has a price tag of £11.42, while the MCU chip costs only around £2.00 [61]. Its functional block diagram is presented in Figure 13.6. Lots of peripherals such as timers and memory elements are included in this chip. The key features of this device are extracted from its datasheet [60] and are listed next.

- MSP430FR5969 MCU
  - 16-bit RISC Architecture
  - Upto 16 MHz clock
  - 100 μA/MHz consumption in active mode and a standby current of 0.4 μA (typical)
  - Ultra-Low-Power Ferroelectric RAM
  - 64 kB of non-volatile memory
  - 2 kB SRAM
  - 32-bit hardware multiplier
  - Wide voltage range - 1.8 V to 3.6 V
  - 48-pin very thin quad flat non-leaded package (VQFN) package
  - -40°C to 85°C operating temperature
  - 40 I/O
- USB interface
- EnergyTrace technology for low power debugging
- 3 push buttons
- 2 LEDs
- 0.1 F SuperCapacitor for standalone power
- JTAG interface

4.4 Comparison

From examining the specifications of the CPLD and FPGA, it is quite obvious that the FPGA packs a lot more hardware than the CPLD. It can also be seen that the MCU can offer a low active and standby power consumption (per MHz). To get a clearer picture of the differences between all these chips, a table is constructed.

	CPLD	**FPGA**	**MCU**
Macro-cells	256	∼2048	N/A
I/O	96	68	40
Max frequency	200 MHz	250 MHz	16 MHz
Standby power (typical)	18.00 μW	24.00 μW	1.44 μW
PLL	0	1	N/A
RAM	0 kB	36 kB	64 kB
Price	£10.00	£14.00	£2.00

Table 4.1: CPLD vs FPGA vs MCU specifications

Table 4.1 above lists the major differences between the CPLD, FPGA and MCU. Note that the standby power is based on the lowest operating core voltage of each of the devices. The number of

macro-cells in the FPGA are nearly ten times that of the CPLD. This suggests that the CPLD will require careful designing to avoid exhausting all of its hardware resources. There are additional hardware features on the FPGA too such as PLL and RAM which are not available on the CPLD. The MCU has the biggest RAM which is needed to store all the instructions and variables.

The maximum possible clock frequency of the FPGA is higher than the CPLD and much higher than the MCU resulting in faster processing, but this may also have the negative impact of a higher power consumption. The typical standby power is lower on the CPLD compared to the FPGA. It is very low on the MCU. This may be because of efficient designing and smaller circuitry requiring less power. Looking at the I/O pins of the devices, the CPLD has the most.

In summary the FPGA has more gates and a faster processing speed than that of the CPLD and MCU. It does however pay a price for all this additional hardware in standby power consumption where the CPLD and MCU perform better. Also it is the slightly more expensive chip.

Chapter 5

Design Flow

5.1 Hardware design flow

Before designing and implementing a hardware system, it is often a good practice to create a flow for this process. This will form the evaluation plan for this project. The flow provides a guideline of the necessary stages needed to be accomplished in order to create a successful functional design. In other words, it creates a recipe for good design that can be followed by the designer. A simple example of an FPGA flow created by industry leading EDA tool providers ALDEC and ALTERA can be seen in Figure 5.1.

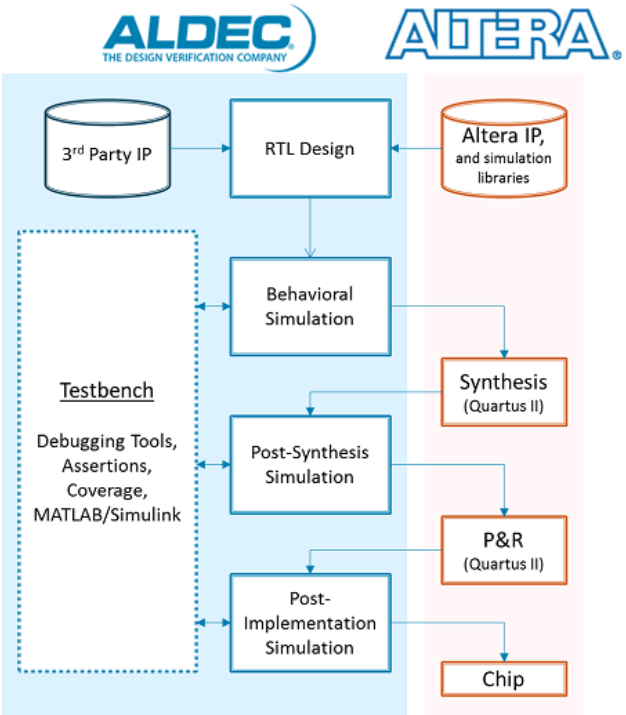


Figure 5.1: ALDEC and ALTERA FPGA flow [62]

The flow displayed in Figure 5.1 consists of several steps which include simulation and synthesis. Based on this, a flow can be created for this project which will incorporate most of what is shown in the ALDEC and ALTERA FPGA flow. Additionally this custom flow will also include other stages that target the specific requirements of this project. The diagram for the flow is attached

below (Figure 5.2) and is followed by a description of all the stages in chronological order.

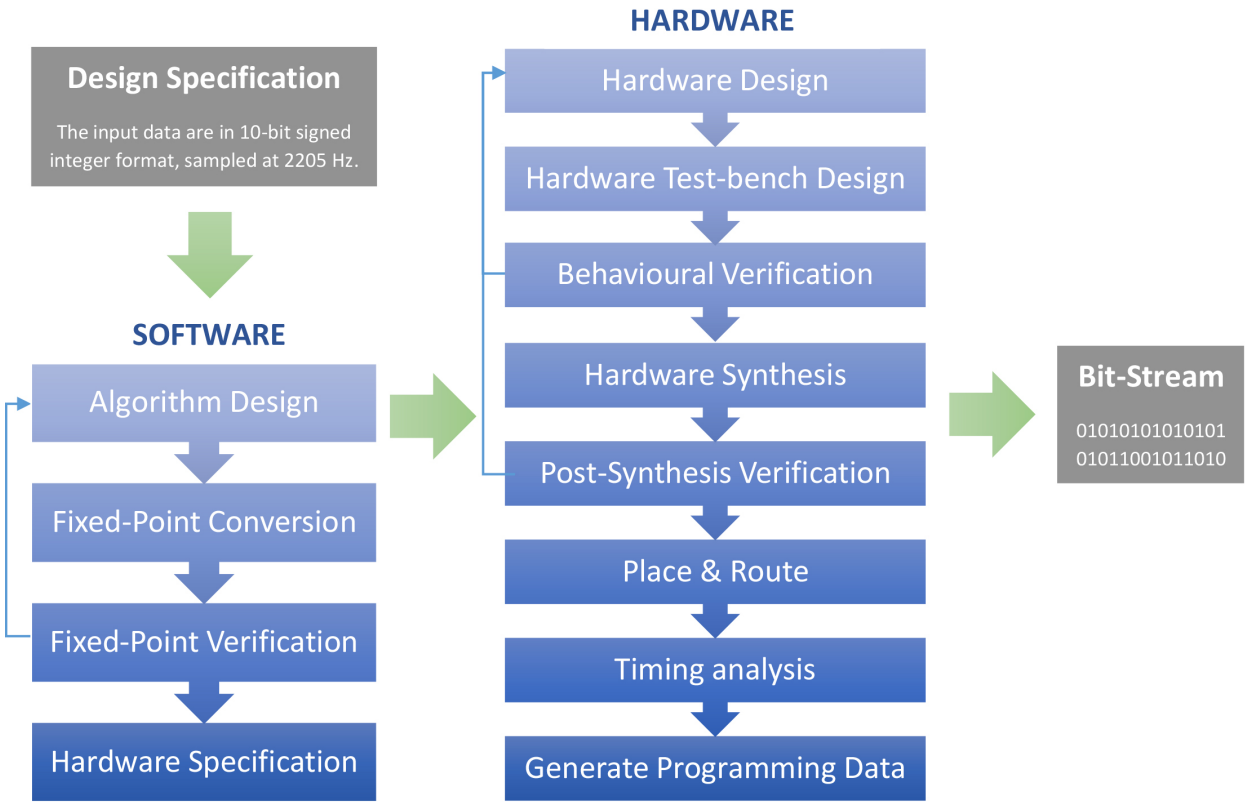


Figure 5.2: Proposed hardware design flow

5.1.1 Design specification

The design specification contains the requirements that the hardware system must fulfil. They are usually written in a text editor (such as Microsoft Word) by the client and are used by the designer to create the system. In this project these specifications are highlighted in Chapter 3 of this report. These include creating a set of filters in hardware.

5.1.2 Software development

Software design is an important aspect of the process. Here the design specification is converted into a software solution. This solution is also used as a correctness measure for the hardware solution later on. Each of the steps under the software flow in Figure 5.2 are expanded below.

Algorithm design

The first step in creating a software solution involves designing an algorithm for the system. Once the algorithm has been designed, it is implemented in a programming language that is the choice of the designer. Some of the more popular [63] programming languages include C, C++, Java and MATLAB. At this stage floating-point accuracy is used.

Fixed-point conversion

Once a floating-point algorithm has been designed, it is usually the case that a fixed-point algorithm is developed to target the digital nature of the hardware. Although floating-point arithmetic units can be created in digital hardware, they are very costly in terms of gates. A Xilinx single precision floating-point multiplier synthesised on a Virtex-6 FPGA (XC6VLX75-1) uses over 600 LUTs and flip-flop's [64]. This is the same story if an adder is synthesised. Thus for hardware which has limited number of gates, it is the best option to implement a fixed-point algorithm. The use of a fixed-point algorithm does lead to loss of precision and therefore accuracy. The consequences of this will be discussed in later chapters.

Fixed-point verification

After creating the fixed-point algorithm, it is necessary to test that it still performs as it should. Obviously the results of the algorithm won't be exactly the same as the floating-point version, due to loss of precision leading to less accuracy. It is therefore important to verify that the fixed-point results doesn't deviate too far away from the expected results. Measures such as SNR or error rate can be used to accomplish this.

The feedback arrow on the flowchart in this stage indicates that if the design doesn't meet the specification then it has to be re-designed.

Hardware specification

Finally from the fixed-point implementation, a hardware specification is created.

5.1.3 Hardware development

Now that there is a hardware specification to follow and there exists a software test-bench for verification, the hardware can be created. The hardware design process contains many steps. These are explained now.

Hardware design

This is the most important part of the flow. It is where the majority of time will be spent by the designer. The hardware design can be created in many ways. These include schematic design or even through the use of a hardware description language's (HDLs) such as Verilog or Very High Speed Integrated Circuits Hardware Description Language (VHDL).

Hardware test-bench design

A test-bench needs to be created in order to verify the correctness of the design. It provides a set of stimuli to the design under test (DUT) and records the output. The output from the hardware design can then be compared to that of the software to check its correctness.

Behavioural verification

Once a test-bench has been created, it can be used in a simulator to verify the DUT. There are many simulators available for this job. One such simulator is called ModelSim [65] by Mentor Graphics. This type of verification is usually known as functional or behavioural verification. It attempts to confirm whether the design conforms to its specifications i.e. does it perform as it should. If it doesn't then it must be re-designed.

Hardware synthesis

Although the high level abstraction of a schematic or HDL is useful for a user designing a hardware system, it cannot be placed on a piece of hardware in this form. This is where the synthesis tool comes in. Synthesis translates the high level description of the hardware design to a lower gate level description. This gate level design can eventually be mapped onto a hardware device such as an FPGA. An example of a logic synthesis tool is Synplify [66] by Synopsys.

Post-Synthesis verification

Following the creation of the low level description from synthesis, its behavioural correctness should be tested. The test-bench created earlier is used again in conjunction with a simulator and the output is compared to that of the software test-bench. If the output differs, the design has to be modified so that it is more synthesis friendly. This requires identifying and modifying those parts of the hardware that are being translated wrongly.

Place and route

This stage is composed of two main steps, placement and routing. Placement comprises of mapping low level gates onto available logic elements on the hardware device. The second step, routing, involves the connection of all the logic with wires. In this stage I/O pins are also assigned to the hardware device.

This is a complex process for a large system. Luckily it can be carried out by EDA tools such as Libero SoC [67] by Microsemi.

Timing analysis

Timing analysis is the process of checking whether the delays in the synthesised circuit meet their timing requirements. This timing requirement may be based on the target frequency of the hardware device. Performing this analysis enables the designer to see what the critical path, i.e. the path with the maximum delay, is. Slack can also be observed. A positive slack indicates a good design, as it means the arrival time of a signal at a node can be delayed, while still meeting its timing requirements.

Again this analysis doesn't need to be performed by a human. EDA tools such as Libero SoC [67] by Microsemi can perform timing analysis.

Generate programming data

A programming data file can be generated of the design. This is used to implement the hardware design on the hardware.

5.1.4 Bit-stream

Finally the hardware device is programmed with the design via a USB cable and the device can now run the algorithm as intended.

5.2 MCU design flow

A design flow is also created for the MCU. This is largely an extension of the software part of the hardware design flow. It is shown in Figure 5.3. The same procedure is again followed from the design specification. An algorithm is first designed. This time the fixed-point conversion and verification stages may be skipped. This is because a MCU may contain a floating point unit so this may not be necessary (this is why it is written in brackets on the figure). The software stages, unique to the MCU are now explained.

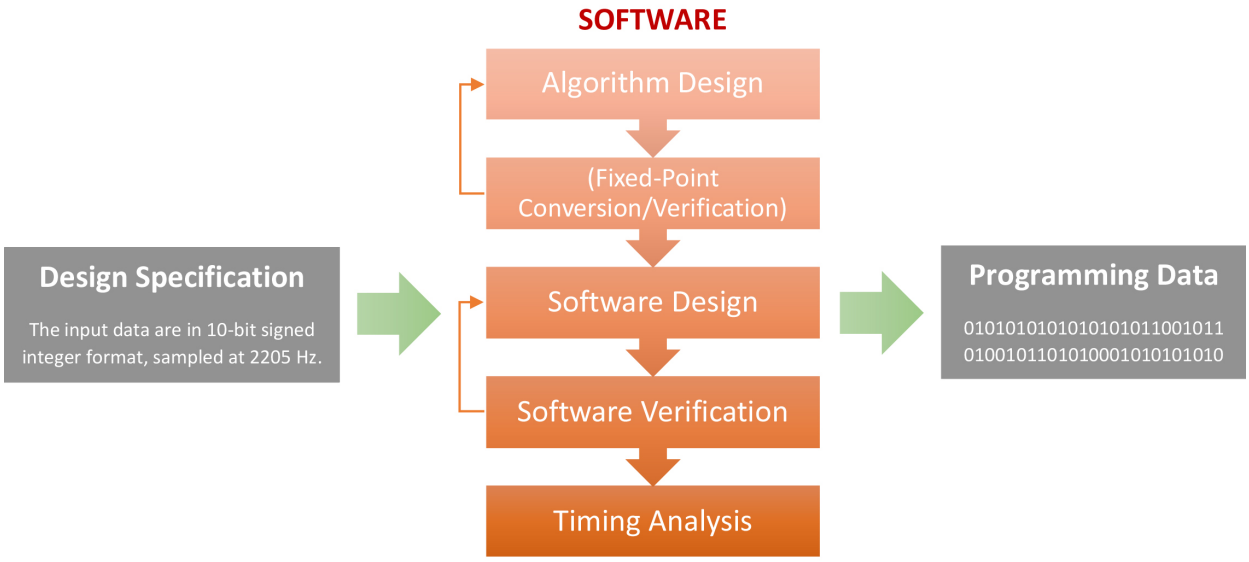


Figure 5.3: Proposed MCU design flow

5.2.1 MCU software development

The algorithm developed and tested in the previous stages is taken and implemented on the MCU. The last few stages on the flowchart in Figure 5.3 are expanded below.

Software design

This is carried out in an integrated development environment (IDE) that is supported by the MCU. The algorithm is usually converted to a high level programming language such as C or C++. It is

also possible to translate the algorithm into assembly code. For ease of coding and debugging this is usually avoided.

In a real-time system, interrupts are used to tell the CPU that some work needs to be done. These interrupts can be programmed to trigger different functions as needed. This also means that when the CPU is idle (i.e. when it is not performing any task), it can be put into a low power state and conserve energy. This interrupt system can be utilised in the software design.

Software verification

After creating the software design, it is necessary to test that it performs as expected. This can either be accomplished by applying test signals to the input pins of the MCU and observing the results on a device such as an oscilloscope or another MCU. Testing can also be simulated in software by providing an input file of data and storing the results in an output file. This output file can then be viewed and the software design can be evaluated.

Timing analysis

Finally it is important to check that the MCU is able to perform the task under real-time signal constraints. For this project, the input signal will arrive at a frequency of 2205 Hz. Therefore it must be ensured that the MCU completes all the processing before the next input data arrives.

Chapter 6

Algorithm Development

The initial step in designing a hardware system requires the creation of a software solution. As discussed in Section 5.1.2, there are a lot of programming languages available for this job. Due to the nature of this project, MATLAB seems to be the most optimal environment for developing this solution. Although filtering, which consists of a multiply accumulate (MAC) algorithm, can easily be implemented in languages like C and C++, they do not have the tools available to create different digital filters. MATLAB natively supports the creation of Butterworth filters and thus is more suited for this task.

Another advantage of MATLAB is its ease of use. Results can quickly be observed with minimal effort. The software also incorporates graphical features such as graphs so data can be visualised easily. Therefore the breathing detection system will be designed and implemented in MATLAB.

6.1 Digital signal processing

Designing filters are an integral part of the DSP discipline. As the implementation of band-pass and low-pass filters is required, the theory behind filtering is briefly explored.

There are two primary forms of digital filters, finite impulse response (FIR) and infinite impulse response (IIR). FIR filters can be modelled as:

$$y(n) = \sum_{k=0}^{M} b_k x(n-k) \tag{6.1}$$

Here b_k represents the coefficients of the filter and $x(n-k)$ is the notation for the delayed input samples. M indicates the order of the filter. Summation of weighted samples is the basis of the MAC algorithm.

IIR filters have the following representation:

$$y(n) = \sum_{k=0}^{M} b_k x(n-k) - \sum_{k=1}^{N} a_k y(n-k) \tag{6.2}$$

The symbols M, b_k and $x(n-k)$ have the same definition as before. IIR filters include an additional summation term dependant on the past outputs. $y(n-k)$ represents the past output values and these are weighted with a_k. N is the order of the IIR filter.

Comparing the two, it can be seen that FIR filters are not dependant on previous output values. This lack of feedback makes them inherently stable. Whereas in IIR filters there is a feedback term. This is an issue particularly in finite precision hardware as rounding errors may lead to the filter becoming unstable (proven later). However a major disadvantage of FIR filters is their inefficiency. They generally need to be a much higher order to achieve the same properties as their IIR counterparts.

To prove this theory, the frequency response of a band-pass filter with a pass-band between 500 and 600 Hz is created and compared for the two filters in MATLAB. The result is shown in Figure 6.1. It can be seen that a 4th order Butterworth IIR filter outperforms a much higher order (16th) Equiripple FIR filter. The IIR filter both has a more accurate pass-band and a lower stop-band than the FIR filter. An FIR filter of a much higher order is needed to achieve a sharper cut-off and lower stop-band, to match the performance of the 4th order IIR filter. A higher order means more multiplications and additions are needed. Therefore for efficient hardware utilisation, IIR filtering is chosen.

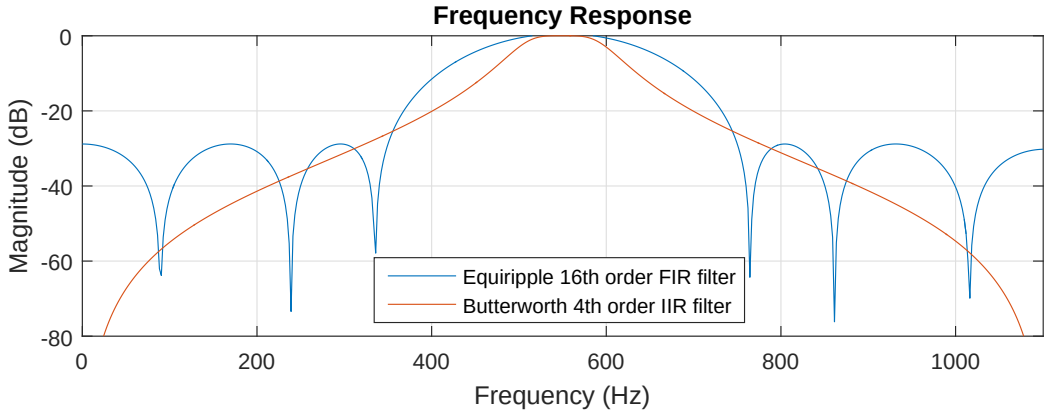


Figure 6.1: Frequency response of an IIR and FIR band-pass filter

6.1.1 Choice of filter

There are quite a few types of IIR filters available. The choice of filter is usually a compromise between certain characteristics of its frequency response. The frequency response of four 4th order IIR filters are displayed in Figure 6.2.

Butterworth

The Butterworth filter is also known as being maximally flat. This is due to there being no ripple in the pass-band or stop-band. The amplitude response for a Butterworth low-pass filter can be written as:

$$G(\omega) = \frac{1}{\sqrt{1 + (\frac{\omega}{\omega_c})^{2n}}} \tag{6.3}$$

Here ω_c is the cut-off frequency and n is the order of the filter. When $\omega = \omega_c$ i.e. at the cut-off frequency, the amplitude response is $1/\sqrt{2}$. As this is a 4th order filter, it has a roll-off of -24 dB per octave.

Chebyshev

There are two types of Chebyshev filters. Type 1 Chebyshev filter has a ripple in its pass-band. A low-pass Chebyshev I filter has an amplitude response that is given by:

$$G(\omega) = \frac{1}{\sqrt{1 + \epsilon^2 C_n^2(\frac{\omega}{\omega_c})}} \tag{6.4}$$

ϵ is the ripple factor and C_n is an nth order Chebyshev polynomial. Then there is the type 2 Chebyshev filter. This has a ripple in its stop-band. Its low pass amplitude response is:

$$G(\omega) = \frac{1}{\sqrt{1 + \frac{1}{\epsilon^2 C_n^2(\frac{\omega}{\omega_c})}}} \tag{6.5}$$

Elliptic

Finally we have the Elliptic filter. It has an equal ripple in the pass-band and stop-band. The response of a low-pass Elliptic filter satisfies:

$$G(\omega) = \frac{1}{\sqrt{1 + \epsilon^2 R_n^2(\zeta, \frac{\omega}{\omega_c})}} \tag{6.6}$$

The function $R_n()$ is the nth order Chebyshev rational function with ζ being the selectivity factor. As there is a ripple in the pass-band, the gain will vary. The gain fluctuates between 1 and $1/\sqrt{1+\epsilon^2}$.

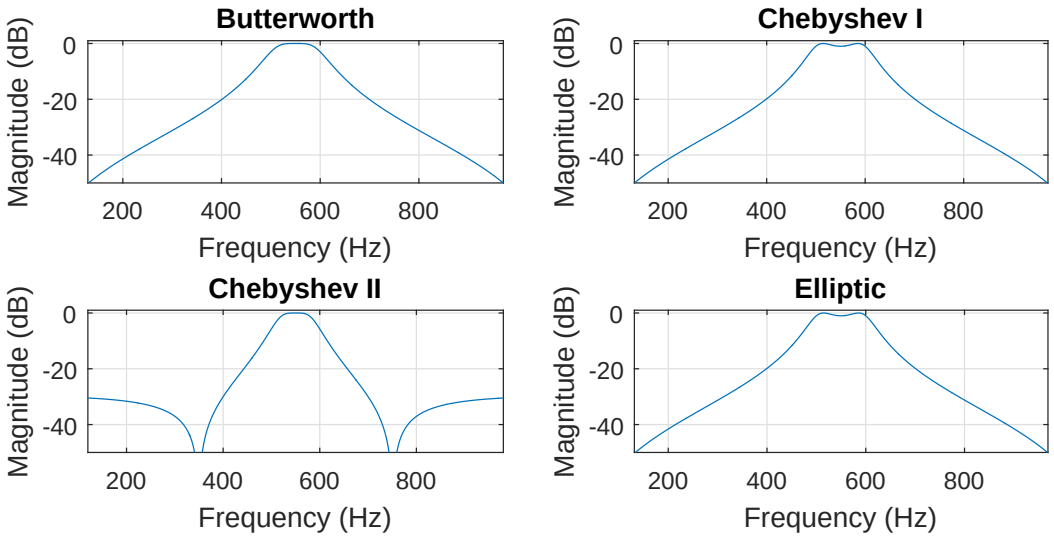


Figure 6.2: Frequency responses of different types of 4th order IIR filters

Comparison

Looking at Figure 6.2, Chebyshev II seems to have the fastest cut-off. However this is dependant on defining the stop-band frequency and attenuation. As the stop-band attenuation is defined as being only 30 dB, it achieves a fast cut-off. The Elliptic filter has the next fastest roll-off. While the Butterworth filter has a roll-off rate that is only marginally worse than the other filters for this low order case. A ripple free pass-band is desired to avoid corrupting the signal, thus ruling out the Elliptic and Chebyshev I filters. The Butterworth filter is preferred over the Chebyshev II filter due to its ease of design and higher out-of-band rejection.

6.1.2 Filter implementation

There are several ways of implementing an IIR filter. Some methods are more efficient than others. In this subsection, different methods are formed and benchmarked to see which is the best to use in hardware and software.

Direct form I

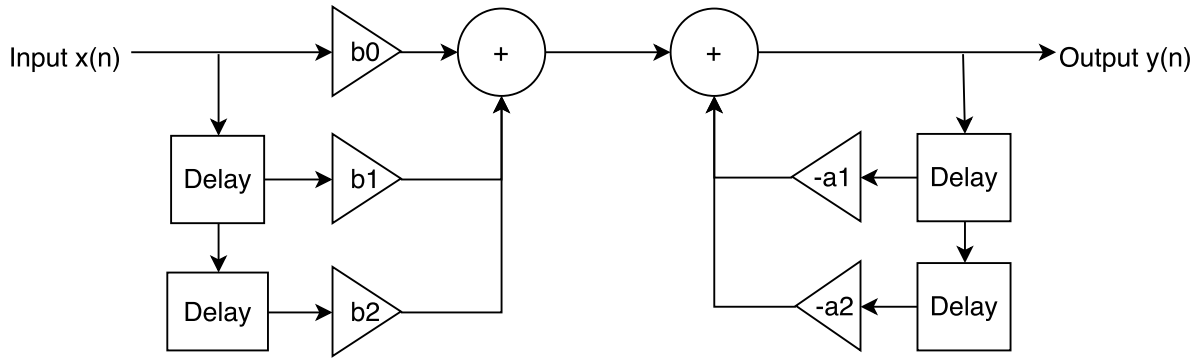


Figure 6.3: Direct form I illustration

This is the most obvious realisation of an IIR filter. Figure 6.3 presents its structure. It is a direct implementation of the difference equation (see Equation 6.7), N is the order of the filter.

$$y(n) = b_0x(n) + b_1x(n-1) + \ldots + b_Nx(n-N) - (a_1y(n-1) + \ldots + a_Ny(n-N)) \tag{6.7}$$

Based on the diagram and equation, pseudo-code can be written (shown in Algorithm 1). In this form of the IIR filter, there needs to be two delay lines (one for the input and another for the output) which result in extra memory usage and computation. Shifting of the delayed samples can be carried out using circular buffers. This avoids copying variables from one location to another which can be slow if large amounts of data needs to be moved (i.e. a high order filter).

Direct form II

The transfer function of the IIR filter can be seen as being, $H(z) = \frac{B(z)}{A(z)}$, with $B(z)$ representing the zeros and $A(z)$ the pole coefficients. In the direct form I filter, $B(z)$ occurs before $\frac{1}{A(z)}$. By reversing the order of these, the direct form II filter is formed. It is shown in Figure 6.4. The number of delays has been halved making it the more efficient than the direct form I implementation.

Algorithm 1 Implementing Direct Form I MAC algorithm

procedure FILTERING($input, output$) ▷ Filters the input. w_in is the past inputs while w_out is the past outputs
  $sum\_b \leftarrow 0$
  $sum\_a \leftarrow 0$
  $w\_in_0 \leftarrow input$ ▷ 0th index is 0th delayed input/output
  for i `= 1 to N step 1` **do** ▷ Perform MAC. N is filter order
    $sum\_b \leftarrow sum\_b + (b_i \times w\_in_i)$
    $sum\_a \leftarrow sum\_a - (a_i \times w\_out_i)$
  end for
  $sum\_b \leftarrow sum\_b + (w\_in_0 \times b_0)$ ▷ Left over 0th sample
  $w\_out_0 \leftarrow sum\_b + sum\_a$ ▷ The current output
  $output \leftarrow w\_out_0$
  for i `= N to 1 step -1` **do** ▷ Shift the delayed samples
    $w\_in_i \leftarrow w\_in_{i-1}$
    $w\_out_i \leftarrow w\_out_{i-1}$
  end for
end procedure

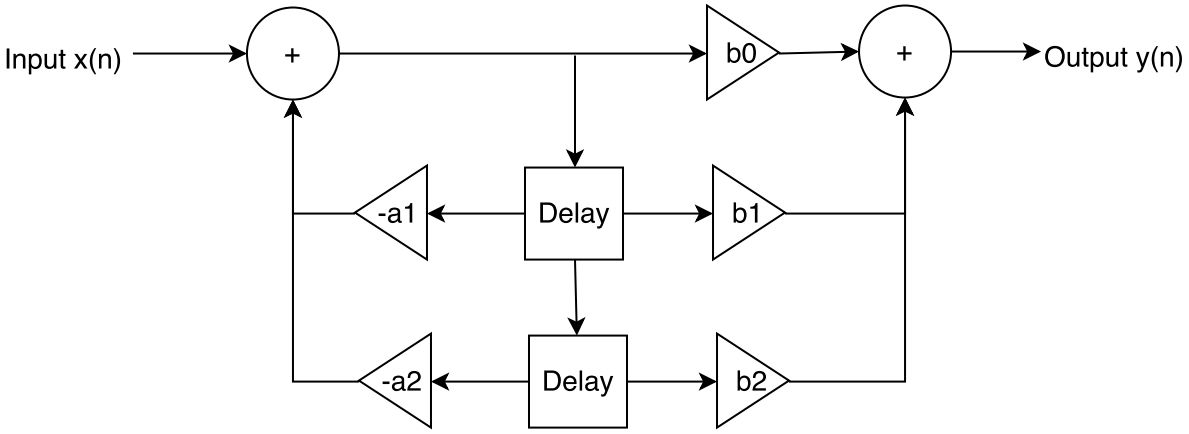


Figure 6.4: Direct form II illustration

To develop the algorithm, it is helpful to consider an intermediate variable between the input and output. This will be known as w and represents a tapped delay line. The equation for this is given by:

$$w(n) = x(n) - (a_1 w(n-1) + a_2 w(n-2) + \ldots + a_N w(n-N)) \tag{6.8}$$

The output can now be written in terms of w which is simply:

$$y(n) = b_0 w(n) + b_1 w(n-1) + b_2 w(n-2) \ldots + b_N w(n-N) \tag{6.9}$$

Algorithm 2 displays how the direct form II filter can be realised. It is much more compact than the direct form I filter.

Algorithm 2 Implementing Direct Form II MAC algorithm

```
procedure FILTERING(input, output)        ▷ Filters the input. w is the tapped delay line
    sum ← 0
    w_0 ← input                           ▷ 0th index is 0th delayed sample
    for i = N to 1 step -1 do             ▷ Perform MAC. N is filter order
        w_0 ← w_0 − (a_i × w_i)
        sum ← sum + (b_i × w_i)
        w_i ← w_{i−1}                     ▷ Shift the delayed samples
    end for
    output ← sum + (w_0 × b_0)
end procedure
```

Direct form II transposed

Starting from the direct form II representation, by swapping the input with the output, reversing the direction of branches and adding the delays between summations, the direct form II transposed filter is created. Unlike the direct form I filter, this again only requires one tapped delay line (like the direct form II filter).

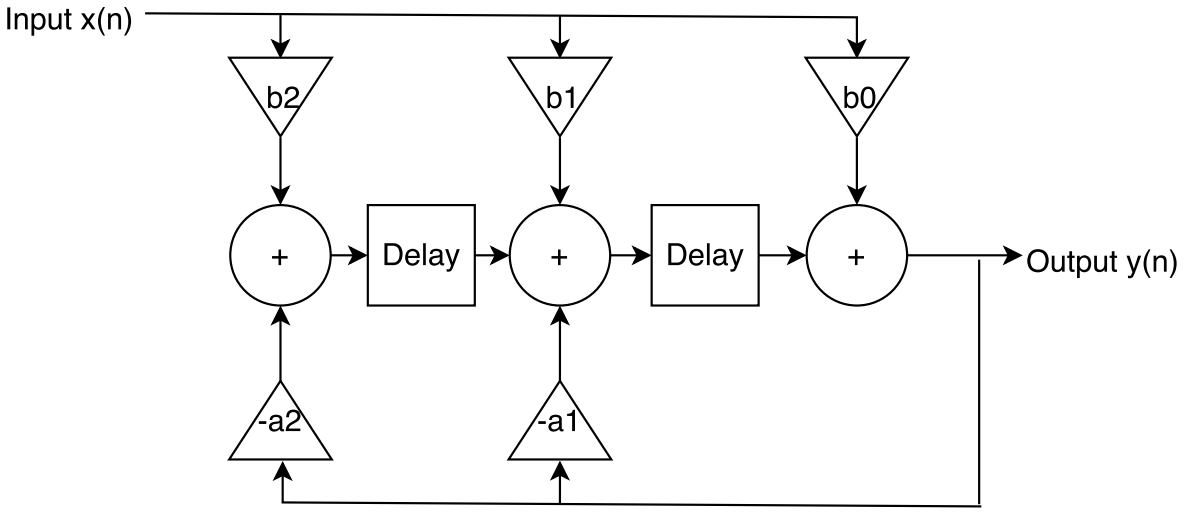


Figure 6.5: Direct form II transposed form illustration

Derivation of the algorithm for this filter is slightly more mathematically involved than for the previous forms. The intermediate variable w is used again. It is between the input and the output. The following equations are formed based on the diagram (Figure 6.5):

$$w(N) = b_0 x(n) + w(N-1) \tag{6.10}$$

$$w(N-1) = b_1 x(n) - a_1 w(N) + w(N-2) \tag{6.11}$$

$$w(N-2) = b_2 x(n) - a_2 w(N) \tag{6.12}$$

$$y(n) = w(N) \tag{6.13}$$

There is a pattern that can be observed from these equations. $w(N)$ and $w(0)$ are corner cases, while for higher order of filters, the computations of indices between N and 0 can be carried out

in a loop. There is also no shifting of delay lines needed, making it the most efficient type of filter. Algorithm 3 shows this.

Algorithm 3 Implementing Direct Form II Transposed MAC algorithm

procedure FILTERING($input, output$) ▷ Filters the input. w is the tapped delay line
  $w_N \leftarrow w_{N-1} + (input \times b_0)$ ▷ Current sample is N (filter output)
  for i `= N-1 to 1 step -1` **do** ▷ Perform MAC
    $w_i \leftarrow w_{i-1} + (input \times b_{N-i}) - (w_N \times a_{N-i})$
  end for
  $w_0 \leftarrow (input \times b_N) - (w_N \times a_N)$ ▷ Left over 0th sample
  $output \leftarrow w_N$
end procedure

Performance analysis

A table is constructed (see Table 6.1) which compares the different types of operations used in each form of the IIR filter. In big O terms, all three forms have the same number of multiplies and additions. These scale linearly with the order of the filter. To be precise, the number of additions needed on the transposed form is the least across all different versions. Decreasing addition operations by one is unlikely to improve performance by much. In the MCU, the add instruction can be executed in a low number of cycles (one to six) [68]. For the hardware, synthesising an adder requires only a few gates [69]. The more expensive operation is the multiply which is the same across all forms.

	Direct Form I	**Direct Form II**	**Direct Form II Transposed**
Multiplies	$2N+1$	$2N+1$	$2N+1$
Additions	$2N+2$	$2N+1$	$2N$
Delay lines	2	1	1
Shifting required	Yes	Yes	No

Table 6.1: Comparison of different forms of the IIR filter

Moving onto the delay lines and shifting required, the direct form II filter is the most efficient. Having less delay lines reduces computation and memory usage in both MCU and hardware. Removing the need for shifting is useful for the MCU since it avoids copying memory or using circular buffers. It does not impact the performance of the hardware by much as shifting can be implemented very efficiently on this platform.

Next the algorithms of all different forms of filters are translated into MATLAB code. Performance profiling is then carried out to see which form is the most efficient practically. An i7 CPU (3632QM) with 8GB DDR3 RAM was used for these tests. The results are available in Figure 6.6 and Figure 6.7. A large number of samples was used (1.7×10^6) for a reliable experiment.

As expected the direct form II transposed filter is the most efficient. Its execution time is the lowest for all filter orders. It is roughly 1.38 and 1.14 times faster than the direct form I and direct form II filters respectively for a filter order of four. Therefore the use of this form is ideal in the software environment. From a hardware point of view, it seems only minutely better than

the direct form II implementation. Thus both forms of direct form II filters are likely be nearly equally efficient.

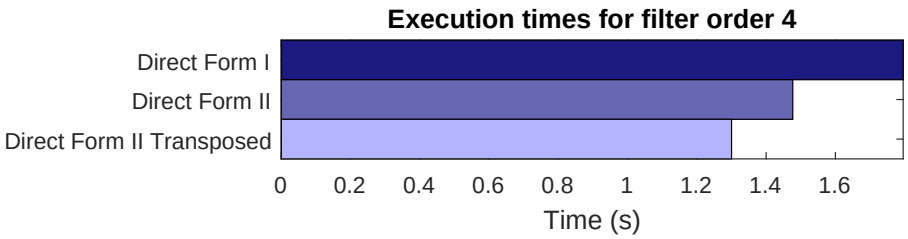


Figure 6.6: Testing execution times for different forms of IIR filter

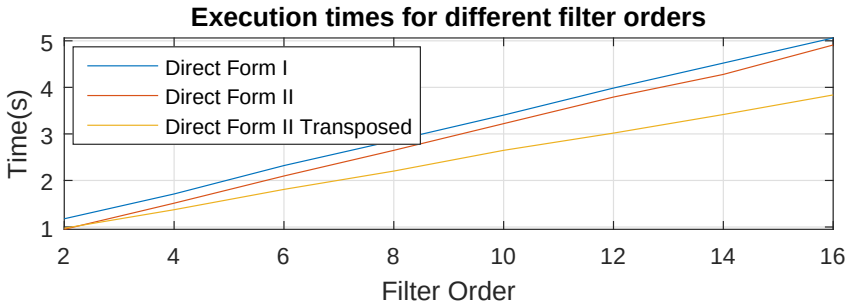


Figure 6.7: Testing execution times for various orders of IIR filter

6.2 Implementation

Initially a floating-point version of all the deliverables is created in MATLAB. There is a choice of building the breathing detector deliverable either based on the results of the three or one channel solution. Since the one channel algorithm has already been tried and tested, it is wise to build the breathing detector based on its output. When required this can easily be adapted to work with the three channel deliverable.

After implementing the double-precision design, a fixed-precision solution will be created. For aforementioned reasons, fixed-point is more suited for devices with limited resources i.e. embedded systems. The breathing detector will again be tuned to the one channel solution.

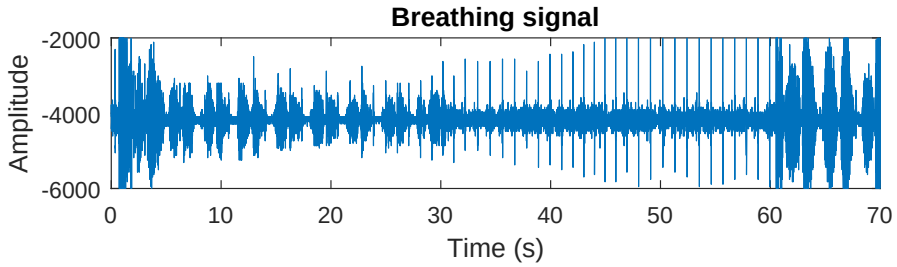


Figure 6.8: 10-bit signed breathing signal (shifted left by 6-bits)

Butterworth band-pass and low-pass filters will be designed. The test input signals consists of a 10-bit signed breathing signal. It is of length 8 minutes and contains periods of breathing followed by periods of apnoea. A short segment of this is depicted in Figure 6.8. Note that purely for convenience it has been multiplied by 64. Because of this it occupies the top 10-bits of a 16-bit

signal. For periods 0 to 30 seconds a breathing signal can vaguely be seen. Whereas between 30 and 60 seconds, the signal is mainly just noise. Noisiness is also observed on the whole signal, displaying a lot of erratic peaks, thus providing motivation for performing filtering on it.

6.2.1 Floating-point version

Three channel deliverable

A 4th order Butterworth band-pass filter is used to suppress noise in the breathing signal. It is easily designed on MATLAB with the `"butter"` command. It should be noted that the function produces filters of order $2n$. The plot in Figure 6.9 shows the frequency response of the filter (t he dotted lines represent the pass-band). A smooth monotonically decreasing curve can be observed in all three channels.

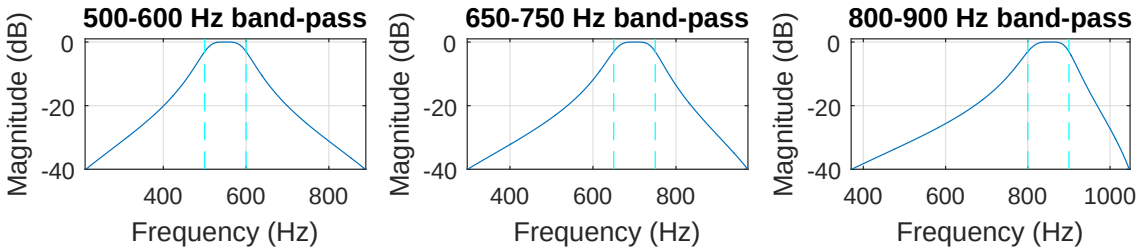


Figure 6.9: Three channel band-pass filtering frequency response

The pole and zero coefficients of the band-pass filters can be used to perform the MAC algorithm. This is carried out for all three channels. Output from the filtering block is then rectified as required. The resulting signals are displayed in Figure 6.10. It is now much easier to identify periods where a breathing signal is present. Rejection of out-of-band noise by the band-pass filtering is therefore very successful. The three channels display differing amplitudes as each of these attenuates different frequencies of the input signal. Around eight occurrences of breathing and apnoea can be seen on all three channels.

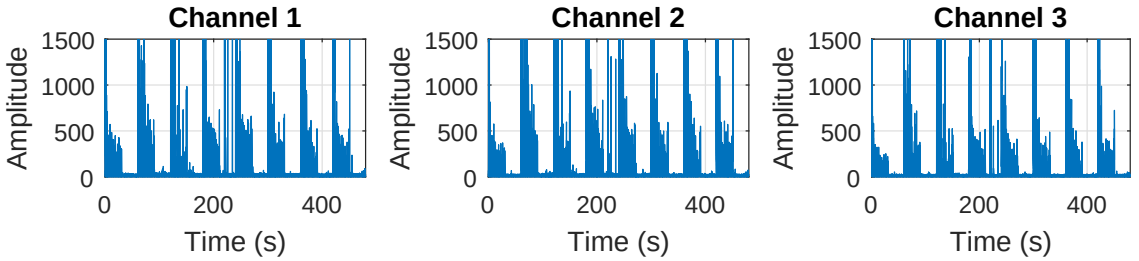


Figure 6.10: Three channel rectified band-pass filtered result

A zoomed in version of Figure 6.10 is also presented (Figure 6.11). It is very clear from this that breathing is occurring up until 30 seconds, after which the signal is low and flat. Respiration resumes again after 60 seconds.

Figure 6.12 displays an unusual peak in an otherwise flat period of signal. It is suspected that this short duration peak could be due to swallowing. This peak is around ten times higher than the average breathing signal.

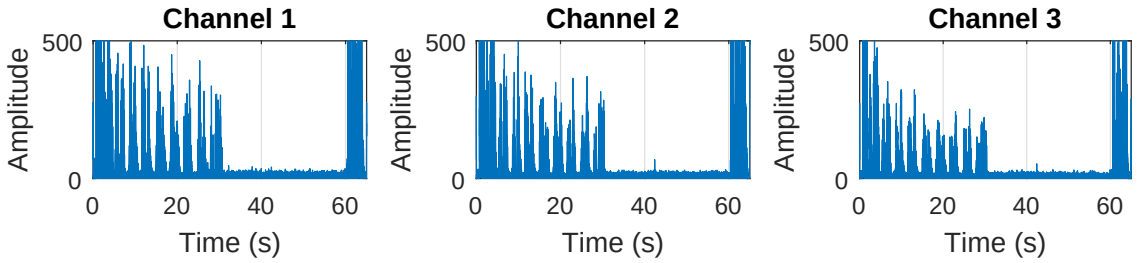


Figure 6.11: Three channel rectified band-pass filtered result - Zoomed in

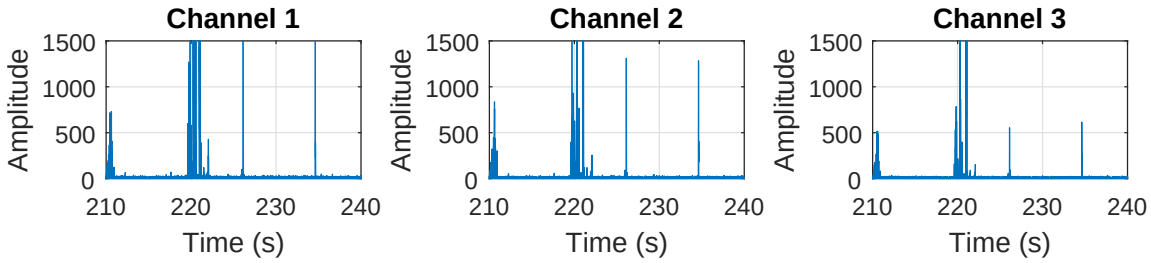


Figure 6.12: Three channel rectified band-pass filtered result - A swallowing artefact

In the next stage, the rectified signal passes through a low-pass filter. This is again constructed in MATLAB by using the `"butter"` command. For a low-pass filter, it produces a filter order of n. A 2 Hz 1st order filter is created. Its frequency response is visualised in Figure 6.13. This low-pass filter smooths the signal.

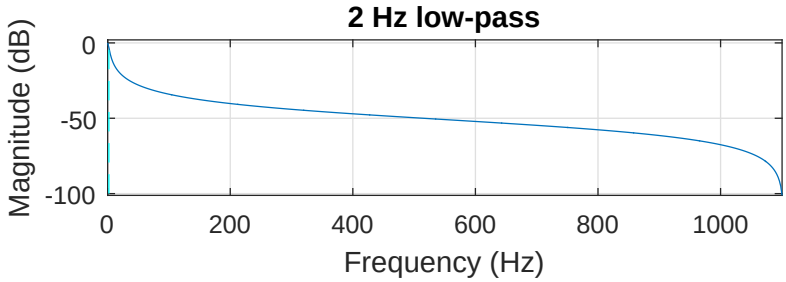


Figure 6.13: Three channel low-pass filtering frequency response

Once the low-pass filter is implemented via a MAC routine, it is applied to all three channels. Finally this smoothed signal is decimated by a factor of twenty-one. This essentially down-samples the result, removing very high frequency variations. The three channel deliverable is now complete.

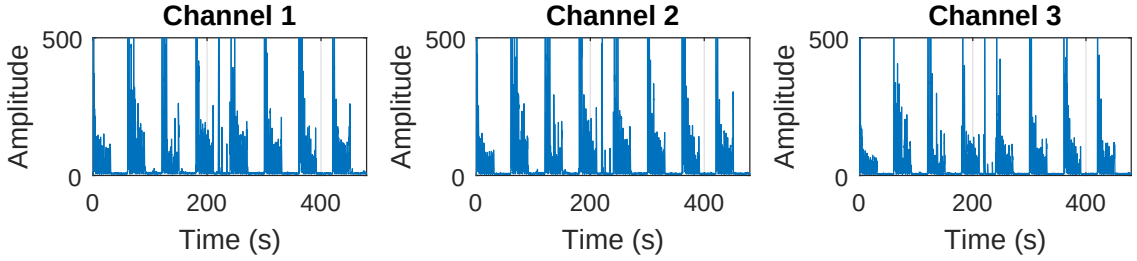


Figure 6.14: Three channel decimated low-pass filtered result

Figure 6.14 shows the result. Contrast between breathing and apnoea is much greater now. It is

perhaps more clearly seen on a zoomed in version of this plot (Figure 6.15). Here the pattern due to inhalation and exhalation can also be observed. This forms the acoustic envelope signal which is useful for seeing properties of the subjects breathing such as rate and power.

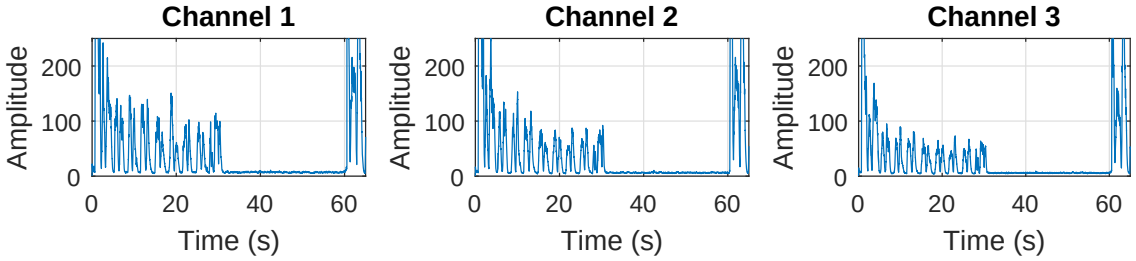


Figure 6.15: Three channel decimated low-pass filtered result - Zoomed in

One channel

Difference in parameters used in filtering and use of a decimator aside, the one channel solution is very similar to the three channel deliverable. Using the `"butter"` function in MATLAB a 6th order band-pass filter is created (see Figure 6.16). This has a pass-band of 500-900 Hz which spans the range of all channels in the three channel design. After filtering the input signal with this band-pass filter, the output is rectified. The result is presented in Figure 6.16 along with a zoomed in version of the first 60 seconds. Visually there isn't much difference that can be seen compared to the three channel deliverable at this stage.

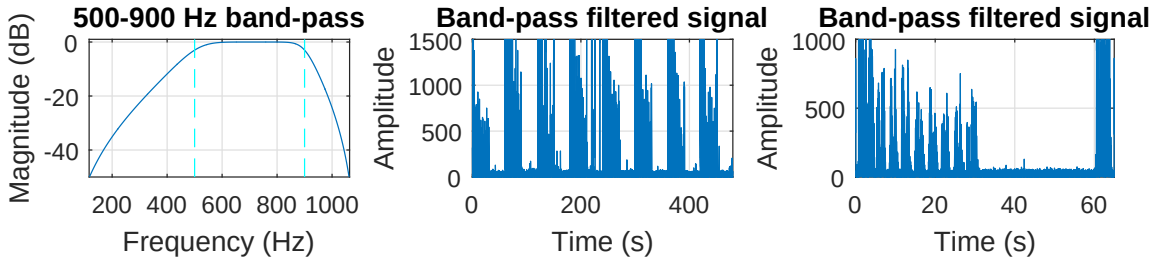


Figure 6.16: Band-pass filter frequency response and the rectified band-pass filtered result

Lastly the rectified band-pass filtered signal is passed through a 340 mHz low-pass filter. As the cut-off frequency is quite low, the frequency response (Figure 6.17) of this filter shows a large attenuation of high frequencies. Consequently the output from this filter is much less spikier than before and compared to the low-pass filter stage of the three channel solution. This is especially seen in the plot that focusses on just the first 60 seconds of the signal.

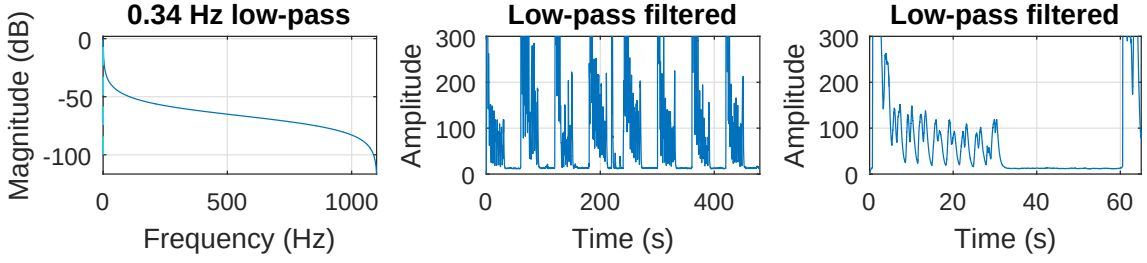


Figure 6.17: Low-pass filter frequency response and the low-pass filtered result

Breathing detector deliverable

From the three or one channel solution, the breathing envelope signal is obtained. Presence of breathing can now be detected using this signal. The feature detection algorithm will be followed in order to form the full detector. This is shown again in Figure 6.18 for reference. The signal from the one channel filtering will be used for this process.

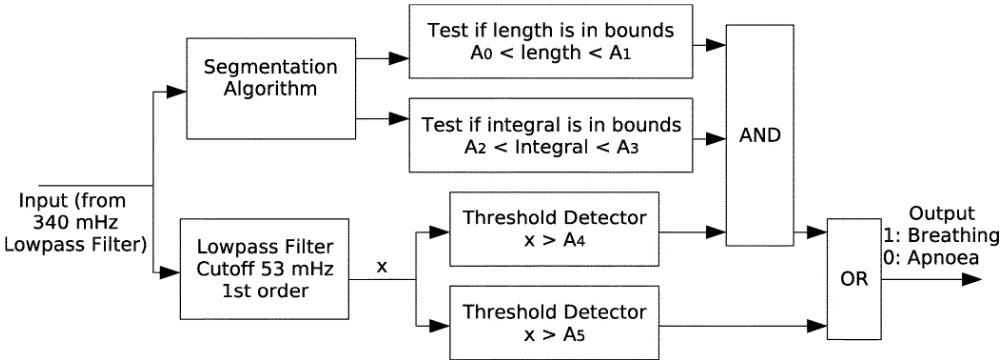


Figure 6.18: Feature recognition section of the algorithm [2]

Starting with the bottom channel, there is initially a bit of filtering involved. A low-pass filter is again designed with a 53 mHz cut-off (see Figure 6.19). The purpose of having such a low cut-off is to average the envelopes power in order to make it easier to identify periods of breathing. This is certainly achieved. Observing the zoomed in filtered plot on the right-hand side of Figure 6.19, there is a lump between 0 and 30 seconds that coincides with the breathing signal.

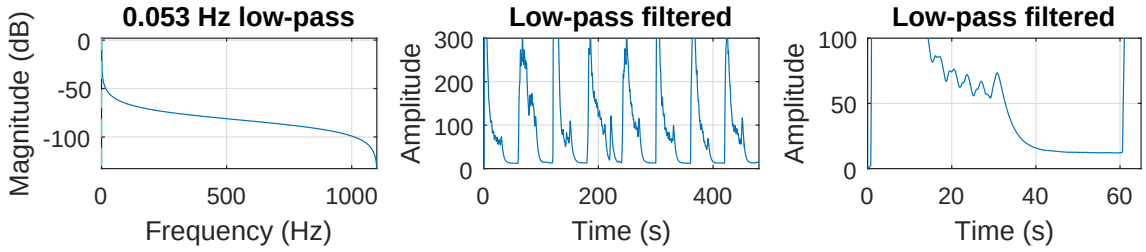


Figure 6.19: Low-pass filter frequency response and the low-pass filtered result

Ideal thresholds are now derived. The A_5 threshold indicates certain presence of respiration. It must be chosen carefully. If it is too low there will be a lot of false positives. By examining the output from the 53 mHz filter, an amplitude of 50 seems reasonable for this set of data. However to make the detector more robust, a higher value of 75 is chosen. This will allow a better rejection of peaks that may occur due to noise during periods of apnoea.

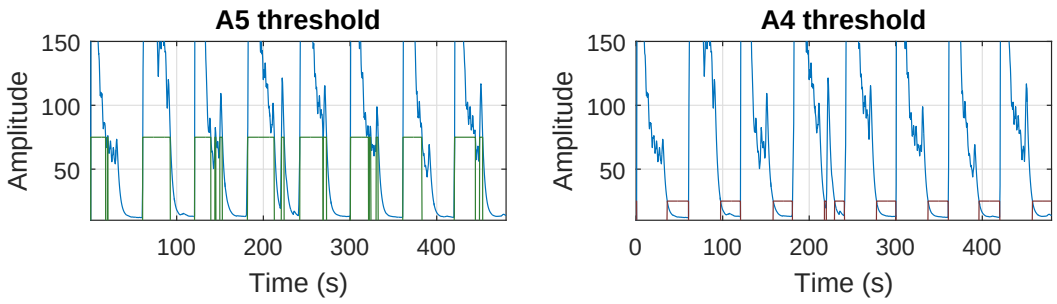


Figure 6.20: Selection of thresholds for the bottom channel

Green rectangular pulses on Figure 6.20 highlight the regions where breathing is confidently thought to be present. A downside to this threshold is the incorrect categorisation of the high power short peak related to swallowing. As this peak is much higher than the average breathing signal, the A_5 threshold will have to be very high to reject it. If the threshold is too high it will effectively play no role in the detection process. As a compromise, for this simple breathing detection system, swallowing will not be rejected. Also it can be argued as swallowing occurs fairly infrequently (compared to breathing), it will not affect the detection system too much.

The role of the A_4 threshold is to identify with certainty periods where apnoea is occurring. It is vital therefore to ensure that a breathing period isn't incorrectly classified as apnoea. The minimum amplitude where there is a lack of breathing is around 12. An A_4 threshold value of 25 is chosen to account for variation about this minimum value. When the signal is below an amplitude of 25, it is highlighted by the red rectangular pulses in Figure 6.20. These pulses fit the areas well where there are breathing pauses. Hence the choice of this threshold is ideal.

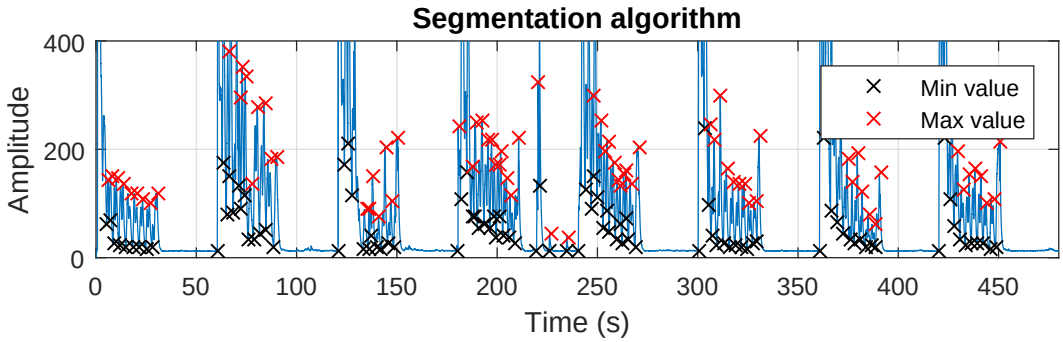


Figure 6.21: Segmentation algorithm implementation

In the top channel, the segmentation algorithm employs the periodic characteristics of the respiratory signal to differentiate breathing from other artefacts in the signal. In order to do this, peaks and troughs of the signal have to be found. The outcome of the minima and maxima finding algorithm is visualised in Figure 6.21.

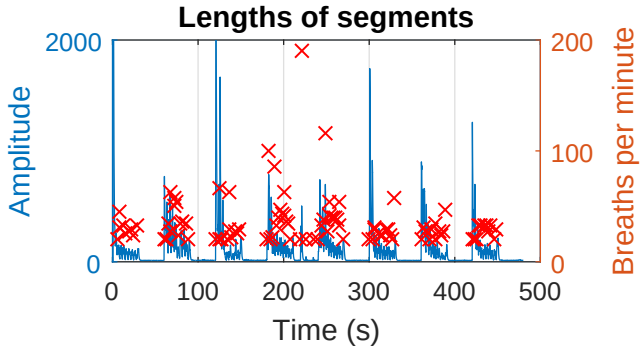


Figure 6.22: Breathing rates

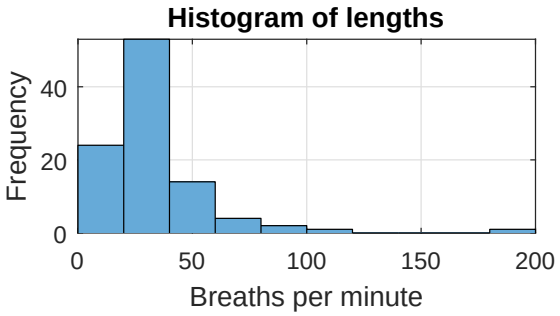


Figure 6.23: Distribution of breathing rates

Once peaks and troughs of the signal are found, the frequency of breathing can be calculated. At rest, respiration occurs at a rate of 18-20 breaths per minute for adults and reaches as high as 44 breaths per minute for newborns [70]. Experimentally the respiratory rate is concentrated at 20 breaths per minute (Figure 6.22). Thus it can be deduced that this data is taken from an adult. The histogram (Figure 6.23) shows the distribution of the breathing rates. These follow an exponential Poisson distribution. Based on this and the expected breathing rates, the lower bound A_0 value is set to 10 breaths per minute and the upper bound A_1 to 45 breaths per minute.

Another interesting observation is that the frequency of the swallowing peak is around 200 breaths per minute. It can therefore be easily filtered out by the A_0 and A_1 thresholds.

Putting the detector based on respiratory rate into practice, a disappointing performance is observed (Figure 6.24). The problem is found to be due to the detection output not being updated while the algorithm is searching for a local minima. Since in the region of apnoea, the signal is very flat, there are no minimums found. Thus the breathing output is only updated when a breathing signal becomes present. Applying a simple fix, a much improved detection system is created (see Figure 6.25). The rectangular pulses indicate presence of breathing.

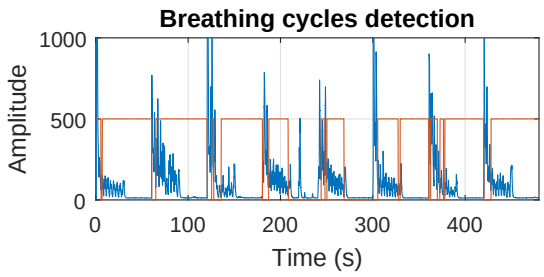


Figure 6.24: Problem in detection

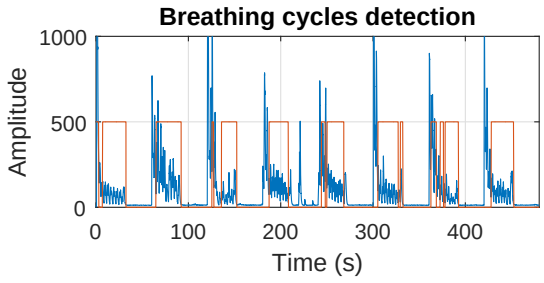


Figure 6.25: Improved detector

Another property that can aid detection of breathing is the power of the signal. As seen in previous figures, there tends to be little or no power where there is a loss of respiration. To compute the power, an integral of the signal needs to be computed. Discrete integration can be carried out in numerous ways. These include approximation techniques such as the trapezoidal rule which can be expressed as being:

$$\int_a^b f(x)dx \approx (b-a)\left(\frac{f(a)+f(b)}{2}\right) \tag{6.14}$$

In real-time, samples ahead aren't available, therefore the area will be calculated simply by summing amplitudes of the signal. Doing so, integrals of all the segments can be seen in Figure 6.26. These are normalised by 2,000 for clarity. Looking at the distribution of these areas, nearly all of them have an amplitude below 200,000. This will be chosen as being the upper threshold A_3. The lower threshold A_2 will be given a value of 30,000 as the lowest integral is around 46,000 (where breathing occurs).

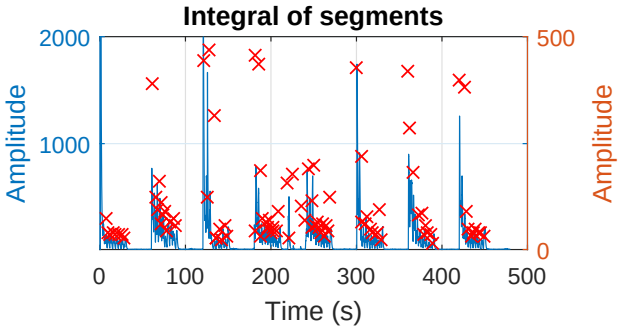


Figure 6.26: Integrals of signal

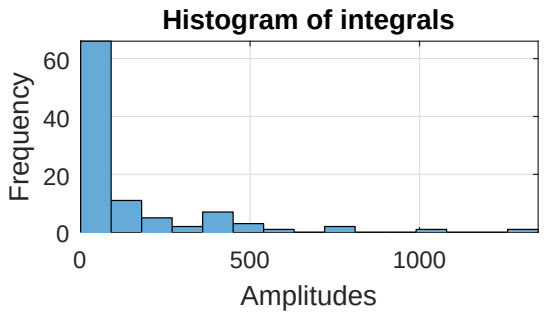


Figure 6.27: Distribution of integrals

On first glance, the power detector (viewable in Figure 6.28) seems to perform similarly to the detector based on breathing cycles. However on closer inspection, it can be seen that the swallowing

artefact is incorrectly identified as being a breathing signal. This is because of its high powered nature. It also incorrectly classifies periods of apnoea as respiratory signals.

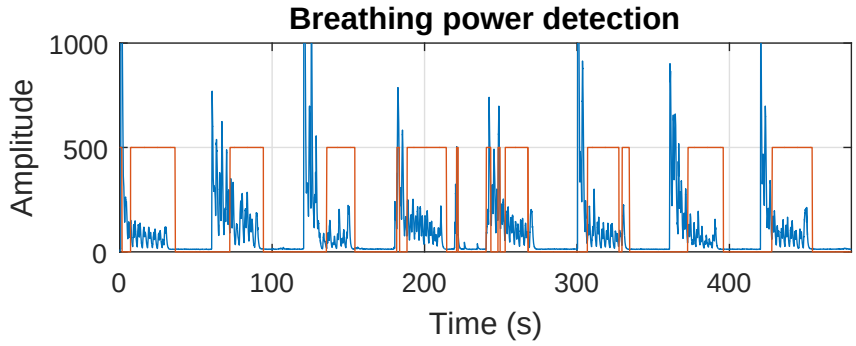


Figure 6.28: Detector based on the power of the signal

Combining both the top and bottom channels, the full breathing detector is formed (see Figure 6.29). It performs pretty well in most cases. The swallowing peak is wrongly identified as a breathing signal as expected. Also there are some small glitches where during breathing, the detection signal drops to zero.

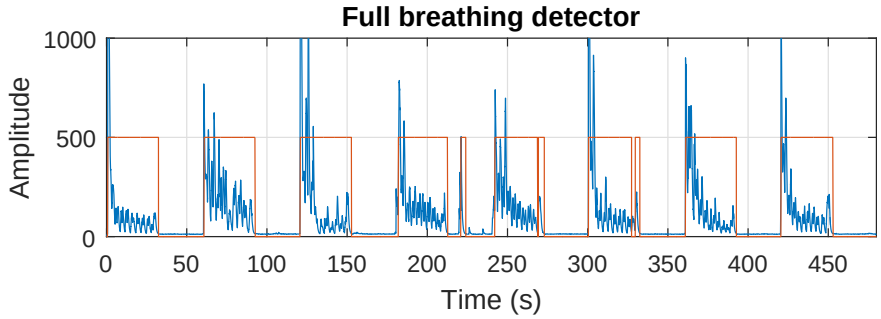


Figure 6.29: Result of the full breathing detector

Evaluating the performance of this detector visually is difficult. Therefore it needs to be carried out by numerical means. By inspection of the input breathing waveform, regions of breathing and apnoea have been identified. This will form the ground truth for testing. Every time the detector incorrectly identifies the signal, it will be penalised. The aim is to find the best parameters to minimise this penalty function. This problem can be posed as an optimisation problem where P is the penalty function.

$$\min_{\forall A} P(n) \tag{6.15}$$

From testing a range of different variables, it is confirmed that the thresholds selected are optimal with respect to the mean square error (MSE) metric. The results are listed in Table 6.2. This testing also shows that the integration/power detector which is part of the top channel segmentation algorithm is pretty much redundant. This is because there is no reduction or improvement in MSE from the removal of it. If the breathing signal was to be more noisy then its shape is likely to be corrupted. In this case the area may be a more useful tool in distinguishing breathing from a lack of it. Based on the breathing signal provided, a very high accuracy of 94.27% is achieved.

Test conditions	MSE	Comment
Default	0.0573	Swallowing not rejected. Slight glitching
$A_5 = 60$	0.0626	Glitching removed
$A_5 = 125$	0.0586	Swallowing rejected
Integration removed	0.0573	Same as default conditions
Breathing rate removed	0.0864	Breathing rate more important than integration
Top channel removed	0.1298	Significantly reduces performance
Bottom channel removed	0.2097	Has the worst performance

Table 6.2: Evaluation of the full breathing detector

6.2.2 Fixed-point version

In the IEEE 754 double-precision floating-point standard, a fractional width of 52-bits is available. Use of a floating binary point allows for a large dynamic range and precision. A representation of this is shown in Figure 6.30. The software solution implemented on MATLAB in the previous section is based on this precision.

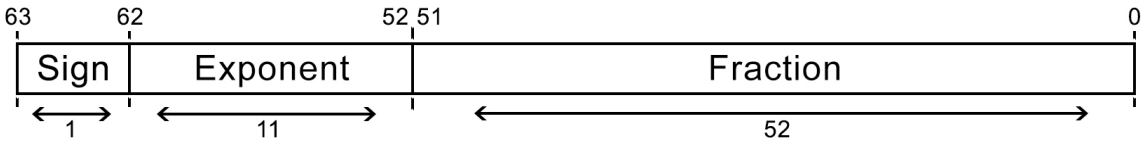


Figure 6.30: Double-precision floating-point representation

However it may not always be feasible to use double-precision. This is particularly the case when the target platforms lack a dedicated floating-point unit. All three of the embedded systems used in this project do not come equipped with this unit. Furthermore the synthesising of a FPU core in an FPGA or CPLD is very resource heavy. Moreover using double-precision in a MCU may consume a lot of cycles. Therefore a more efficient way of representing numbers is desired.

An alternative to using floating-point is the fixed-point representation. Here the binary point is fixed. Subsequently the designer must choose the fractional width carefully to ensure the numerical error is kept to a minimum. Figure 6.31 illustrates a possible segmentation of a 16-bit word. It can be decomposed into a fractional width of 12-bits and an integer range that is 3-bits wide.

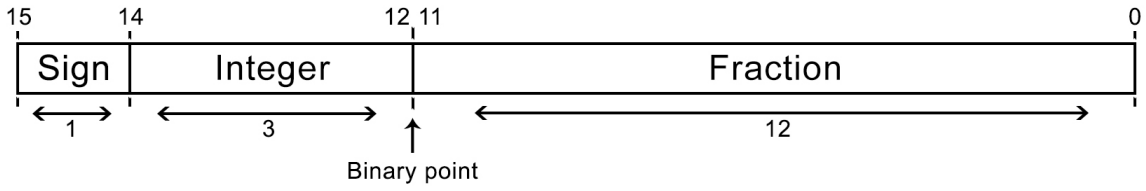


Figure 6.31: A possible 16-bit fixed-point representation

There are some challenges associated with the use of fixed-point arithmetic. Most obvious is the reduction in precision of calculations compared to the double-precision floating-point format. Precision of a fixed-point number can be increased by using more fractional bits (the word length will also have to scale accordingly). This is because precision is dependant on the width after the binary point, 2^{-frac}, ($frac$ is the fractional length). Increasing the length of the fractional section

is clearly a trade-off between an increase of precision and hardware usage (or cycles). An optimal balance must be determined.

Another form of precision loss could occur when the result from multiplying two 16-bit integers for example, is stored in a 16-bit variable. Severe truncation would occur in this case. On the other hand storing the result in a 32-bit variable will result in no truncation at all. Bounds of the resulting number need to be known beforehand to prevent this from happening. Precision errors also arise from division. Dividing 2.5 by 2 is equivalent to shifting the binary bits of 2.5 right by 1. If there are insufficient fractional bits then the result may be rounded down (floored) to 1 instead of being the true value of 1.25.

Using the fixed-point format with a limited range may result in overflow. Overflow is a consequence of there being a shortage of bits to represent the true number. In other words, the number is too large to be handled. This is mainly dependant on the word length and the length of the integer part of the fixed-point number. For a fixed word length there is a conflicting desire between increasing precision through usage of more fractional bits and reducing risk of overflow by having more integer bits. The optimal ratio will be dependant upon the type of problem.

Filtering

With the shortcomings of quantisation in mind, a fixed-point conversion of the one channel solution is now created. Extra care has to be taken in designing IIR filters as they can become unstable. When the feedback becomes positive, the output oscillates and goes to infinity. A pole-zero plot can be used to show how loss of precision can lead to instability. The poles can be modelled as giving a gain. While zeros can be modelled as having an attenuating affect to the frequency response.

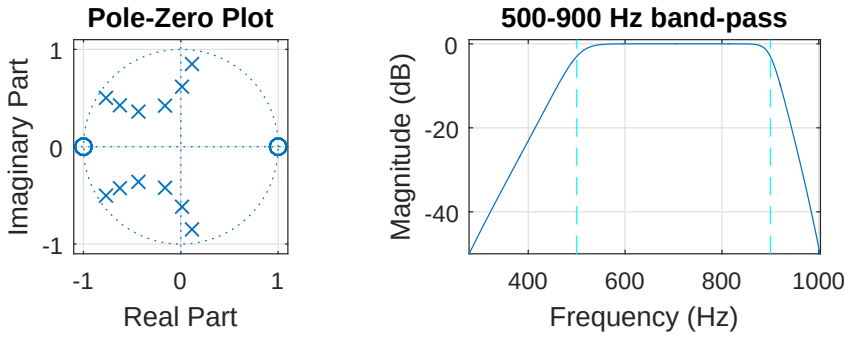


Figure 6.32: A floating-point pole-zero plot and its frequency response (12th order filter)

To demonstrate this effect, a floating-point 12th order Butterworth band-pass filter is implemented. Its pole-zero plot and frequency response is depicted in Figure 6.32. The frequency response of the filter can be deduced from its pole-zero plot. Starting at the (1,0) co-ordinate, there are zeros there which attenuate the response. Going around the unit circle in an anti-clockwise manner, poles are approached starting at an analogue frequency of about 500 Hz. These pull the gain up to form the pass-band. The last pole is located at a frequency of 900 Hz. Finally reaching the co-ordinate (-1,0) there are again zeros there which provide a steep roll-off.

A badly designed fixed-point version of the 12th order Butterworth band-pass filter is also created. The fractional width for this design is only 2-bits. Its pole-zero plot and frequency response is presented in Figure 6.33. When a pole is located outside the unit circle, the filter becomes

unstable. This is exactly what is observed in this case. Behaviour of this frequency response is undesirable.

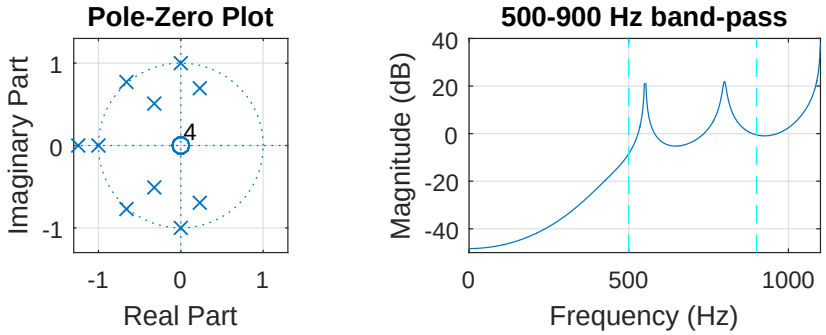


Figure 6.33: An unstable fixed-point pole-zero plot and its frequency response (12th order filter)

MSE between coefficients of the floating-point and fixed-point representations is calculated. This is to determine the optimal fractional length for use in both the band-pass and low-pass filtering of the one channel solution (the same process is also followed for the three channel solution). For the band-pass filter, the result can be seen in Figure 6.34. Initially as the fractional width is increased, there is a large decrease in error. As more bits are used there are diminishing decreases in error. Using these results the optimal segmentation of the word length for each coefficient is found. Similarly the MSE of the quantised low-pass filter for different segmentations is also calculated. This can be seen in Figure 6.35.

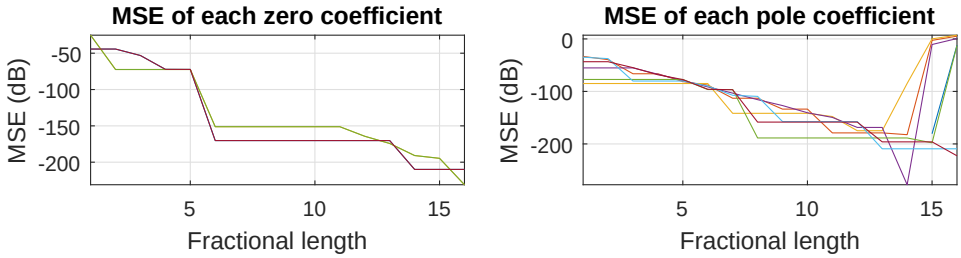


Figure 6.34: Quantising error for the coefficients of the band-pass filter (one channel)

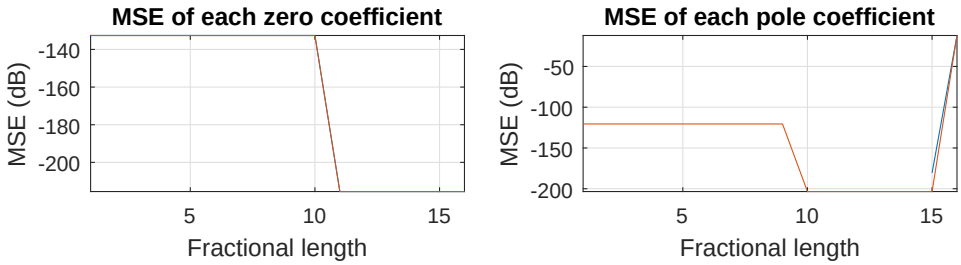


Figure 6.35: Quantising error for the coefficients of the low-pass filter (one channel)

Tables are also constructed to compare the errors of the optimally quantised coefficients. Referring to Tables 6.3 and 6.4, all of the coefficients have very small errors. These have an order of magnitude of 10^{-5} or less. The frequency response of the floating-point coefficients is superimposed onto the fixed-point coefficients in Figure 6.36. This time the frequency response of the three channel quantised coefficients is also displayed in Figure 6.37. No differences can be seen visually in the shape of the response curves. It can thus be concluded that the quantisation process has been very successful.

Pole		Zero	
Coefficient	**Error**	**Coefficient**	**Error**
a_0	0	b_0	-5.645113126423462e-06
a_1	-2.750423514097378e-05	b_1	0
a_2	-4.257754266978608e-05	b_2	1.676550316770387e-06
a_3	-1.140139809230334e-07	b_3	0
a_4	1.127037800197073e-05	b_4	1.676550316770387e-06
a_5	5.864237482999801e-06	b_5	0
a_6	2.718330030002103e-06	b_6	-5.645113126423462e-06

Table 6.3: Errors between the floating-point and fixed-point band-pass coefficients (one channel)

Pole		Zero	
Coefficient	**Error**	**Coefficient**	**Error**
a_0	0	b_0	-4.097844403527784e-06
a_1	-8.195688807055568e-06	b_1	-4.097844403527784e-06

Table 6.4: Errors between the floating-point and fixed-point low-pass coefficients (one channel)

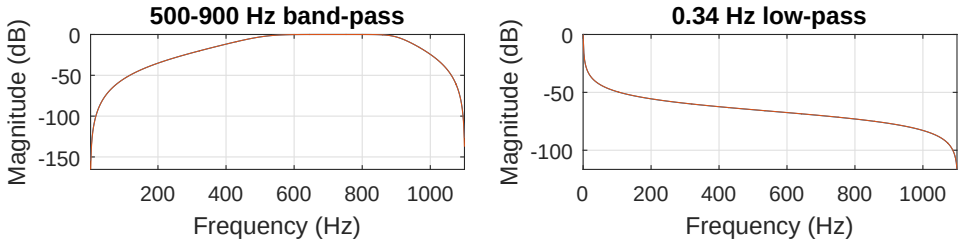


Figure 6.36: Quantised and floating-point frequency response (one channel)

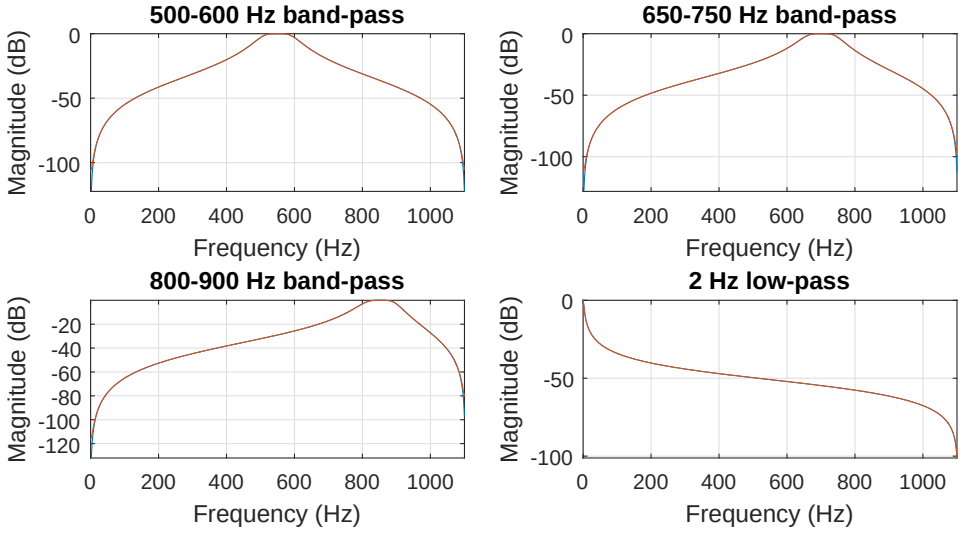


Figure 6.37: Quantised and floating-point frequency response (three channel)

These coefficients can now be used in an IIR filter. There are several forms of direct form filters

that can be utilised. Although the direct form II transposed filter presents the least amount of arithmetic operations, it may not be the most optimal under fixed-point conditions. A way to perform multiplication is to form a rational fraction of the coefficient (also known as Q format). This can be represented as demonstrated in Equations 6.16 and 6.17.

$$x = \frac{Numerator_x}{Denominator_x} \tag{6.16}$$

$$3.25 = \frac{13}{4} \tag{6.17}$$

The numerator of this fraction can then be used for multiplication. Followed by a division of the result with the denominator of this fraction. These actions are equivalent to multiplying by a floating-point decimal number, albeit some precision will be lost in this format when division occurs and the coefficient may be irrational, so quantising will be required which will lead to more precision loss. Equation 6.18 demonstrates the equivalence between this technique and normal multiplication.

$$3.25 \times 5 = \frac{13 \times 5}{4} \tag{6.18}$$

As the coefficients are already quantised due to fixed-point conversion, they are guaranteed to be representable in a rational fraction form. Therefore this technique can be exploited. Furthermore since the denominator is a power of two, division can occur very cheaply in hardware (just requires shifting).

In the transposed filter form, implementing the rational multiplication method poses some problems. If the b_0 coefficient is small then the result of its multiplication may be rounded down to zero due to the finite precision available. As this result is stored in the delay line and a feedback mechanism is used, there will be a huge negative impact on the performance. The repeated feedback of accumulating errors may even lead to the output of the filter overflowing if care isn't taken in its design.

The direct form II filter is chosen instead. Although there is an extra addition required (compared to the transposed form), it is more suited to fixed-point design. Hence it can be implemented more efficiently and as a result consume less power. Considering the application of this implementation is medical, it is crucial to make the detector as accurate as possible. From the floating-point design, it is noted that a lot of attenuation of the signal occurs when low-pass filtering is applied. This would make the dynamic range of the signal very small in fixed-point. To increase the resolution, the signal can be pre-multiplied. This can be achieved by shifting the input bits to the left. Precision would also be increased. A value of six bits is chosen for the shifting pre-multiplication.

The next stage involves converting the filtering algorithm into fixed-point form. This will require bounds to be established in order to deduce the amount of bits needed for precise calculations while keeping the design embedded system friendly. An easy to implement way of finding these bounds is through the use of the so called Monte Carlo simulation. This consists of running a simulation with a certain fixed-point precision driven by a random input and then comparing the output to a floating-point precision simulation. Different precisions are tested. When the error between the two simulations is small enough, then that bit precision is chosen. This can however be a long winded method as it requires a lot of testing. It is also similar to trial and error. A more directed method is desirable.

Since the input data is available (the breathing signal), a random input signal is not needed. Instead the Monte Carlo simulation can be reverse engineered to see how many bits the double-precision floating-point version of filtering actually consumes. This will be recorded by running the simulation on the breathing signal. The fixed-point solution can then be set to the maximum number of bits found from this experiment.

Test	**Delay array**	**Multiplying** a	**Multiplying** b	**Sum/output**
BP 500-900 Hz	19	33	32	17
LP 340 mHz	23	33	23	13
BP 500-600 Hz	20	34	31	16
BP 650-750 Hz	21	35	31	16
BP 800-900 Hz	22	35	31	15
LP Ch1 2 Hz	21	35	27	13
LP Ch2 2 Hz	21	35	28	14
LP Ch3 2 Hz	20	34	27	13
Maximum	**23**	**35**	**32**	**17**

Table 6.5: The maximum number of bits required for the test input data

Table 6.5 lists the results of the simulation. It contains the number of bits needed to represent a number in signed format. Storing the result from multiplying the numerator of the quantised pole coefficient, consistently requires the most amount of bits. A maximum of 35-bits is needed. Multiplying the numerator of the zero rational coefficient also requires a high number of bits (maximum of 32-bits). The output and intermediate summing use a maximum of 16-bits unsigned or 17-bits signed to prevent overflow. Finally the delay array requires at most 23-bits.

An issue with this technique is that it only provides the minimum number of bits required to prevent overflow. No guarantees are provided on the number of bits needed in the system as a whole. Another form of filtering for instance the 53 mHz low-pass filter used in the breathing detection system may demand more bits. Thus an upper-bound needs to be established.

By considering the intervals that can occur as a result of a calculation at each stage of filtering, the upper-bound of bits can be found. This is a form of interval arithmetic and presents the worst case scenario. Starting with the numerator of the quantised pole coefficient, the coefficient itself will be bounded in the range [0 16] bits. Likewise the quantised zero coefficient is also bounded in the range [0 16] bits. The w array can be assumed to be bounded in the region [0 32] bits. It is chosen as this size to maintain a high precision of calculation and to minimise the risk of overflow. Multiplying both the pole and zero numerator with w will result in the usage of [0 48] bits of storage. As the zero coefficients are usually less than one in magnitude, the result can be expected to have a bound of [0 17] bits. The size of the output should be the same as the input since all the filters have unity gain in the pass-band. However to account for finite precision effects, an extra bit is allocated.

Type	**Delay array**	**Multiplying** a	**Multiplying** b	**Sum/output**
Upper bound	32	48	48	17
Lower bound	23	35	32	17

Table 6.6: Upper and lower bounds of the bits required in filtering

Both upper and lower bounds calculated are presented in Table 6.6. A number of bits between these intervals needs to be further researched. Intuitively multiplying a pole coefficient is likely to require more bits than a zero coefficient. This is because pole coefficients have magnitudes that can be greater than one whereas the zero coefficients are usually much smaller in magnitude. Thus the destination of multiplying the zero coefficient can be made smaller. However to allow for freedom of design of these coefficients, the upper bound is the most ideal. Another way of reducing bits is to target the delay array. Since experimental analysis shows that the delay array doesn't seem to exceed 23-bits, it can be set to 25-bits. These two extra bits are there to provide a certain margin to avoid overflowing. Number of bits required for multiplication will also be reduced to 41-bits as a result .

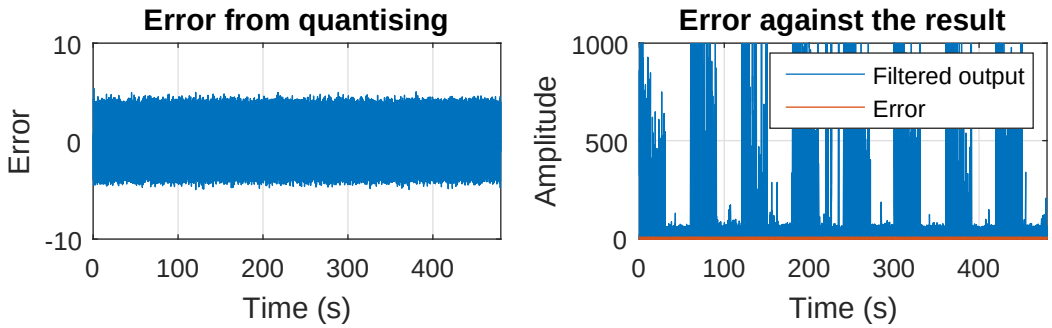


Figure 6.38: Error in the fixed-point one channel band-pass output

The fixed-point one channel filtering is implemented in MATLAB and its result is compared to the floating-point version. This can be seen in Figure 6.38. It shows the difference between the quantised and non-quantised output from the 500-900 Hz band-pass filter. Peak to peak amplitude of the error signal is very small (value of just eight). Comparing this to the filtered signal, the error is of negligible amplitude. Thus the full breathing detector stage can now be tackled.

Full breathing detector

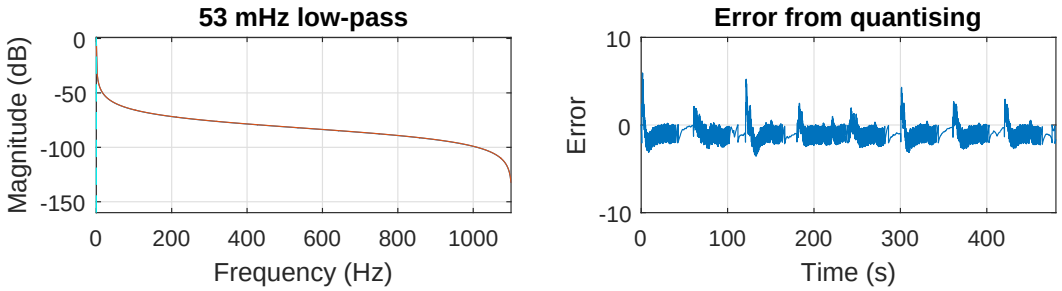


Figure 6.39: 53 mHz frequency response and the filtering quantising error

The full breathing detector is based on the output from the one channel fixed-point solution. Starting with the bottom channel, the same practice is followed as before. Firstly the coefficients of the 53 mHz filter are quantised in a 16-bit Q format. To confirm the success of quantisations, the MSE between the non-quantised and quantised frequency response is calculated. It is found to be a small value of 2.1092e-10. The frequency response of the floating and fixed-point versions is displayed in Figure 6.39. Here the quantised response is almost perfect. Filtering is next performed with the resultant coefficients. Its result is again compared to the double-precision method and the error is plotted on the right-hand side of Figure 6.39. Compared to the amplitude of the

filtered signal, the error is minuscule. The same optimal A_i threshold values are used as the double-precision format.

Following construction of the bottom channel of the breathing algorithm, the top channel needs to be converted into a quantised solution. There is hardly any heavy computation in this section therefore converting to fixed-point is a quick job. Though a few design choices need to be considered. Where the length of a segment is computed needs to be bounded. This can be set to the ceiling of $log_2 A_1$ bits as there is no reason in calculating the length beyond this number since it falls outside the thresholds. This turns out to be 12-bits. Similarly integral computation can be bounded to 18-bits. Applying these changes to the design, the full fixed-point breathing detector is finally formed.

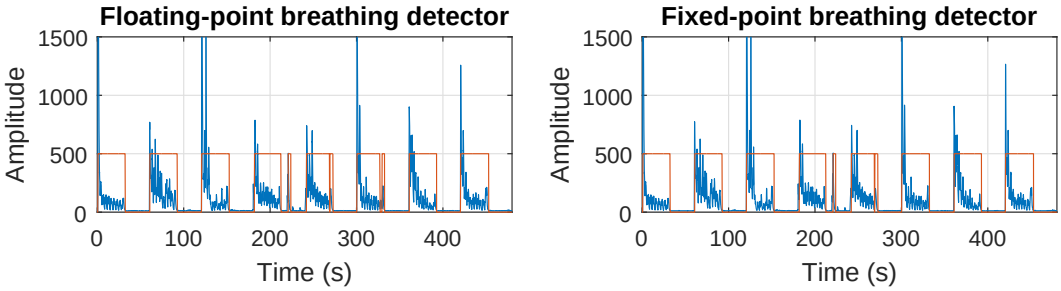


Figure 6.40: Floating-point vs fixed-point breathing detector

	Floating-point version	Fixed-point version
MSE	0.0573	0.0519

Table 6.7: MSE of the two different solutions

The result of the newly developed breathing detector is displayed in Figure 6.40 alongside the non-quantised version. Red signal on both graphs indicates the presence of breathing. When this signal is low, apnoea is thought to be occurring. At first sight both the breathing detectors performance are alike. However by calculating the MSE of the breathing detection signal against the ground truth value, something odd is noticed (see Table 6.7). The error of the fixed-point design is less than that of the floating-point version. The period of time that occurs between 300 s and 400 s in Figure 6.40 shows why the error has reduced. A glitch no longer occurs in this interval due to the precision losses incurred in the quantisation process. This is a pleasant surprise as quantisation is conventionally thought to worsen the performance of a system. It has instead actually marginally improved it. The accuracy of detection has gone up to 94.81%. This concludes the software design section.

Chapter 7

Hardware Solution

Hardware development comprises of multiple stages which include design, verification and synthesis. These along with other steps from the hardware design flow (see Section 5.1) will be followed for both the FPGA and CPLD. Several options exist for the creation of hardware. It can be carried out in a graphical manner by connecting different blocks of hardware together. These are available through a catalogue of designs. Although this method provides ease of designing, there is fundamentally less control over the end product. Since this is conflicting with the aim of this project which is to develop optimised low power solutions, a HDL approach is instead taken. The choice of HDL is purely a matter of the user's preference. VHDL has been decided as the choice of HDL for this project due to its strongly typed nature.

Microsemi's Libero SoC is the chosen development environment for the FPGA. Features such as synthesis, simulators, place and routing, timing analysis and power usage are included this software. Also as this tool is developed by the manufacturer of the selected FPGA, it is ideal due to its compatibility with the device. For the CPLD the ispLEVER software is used. It too contains all the necessary tools for full hardware development. However it lacks some features such as power analysis which are available for the FPGA. The manufacturer of the software is also the manufacturer of the CPLD board.

7.1 Multiplier

Before diving into the complete hardware design of deliverables, a smaller part of it is initially synthesised. This is the multiplier. Multiplication forms the basis of the MAC algorithm used to implement filtering. Hence without it, the whole design will not be possible. Consequently there is high importance placed on its efficient implementation. The basic algorithm of binary multiplication is much simpler than its decimal counterpart. Thanks to two's complement notation, binary multiplication can be easily carried out for both signed and unsigned formats.

$$
\begin{array}{cccccc}
 & & & 1 & 0 & 1 \\
\times & & & 1 & 1 & 0 \\
\hline
 & & & 0 & 0 & 0 \\
 & & 1 & 0 & 1 & \\
+ & 1 & 0 & 1 & & \\
\hline
0 & 1 & 1 & 1 & 1 & 0
\end{array}
$$

Consider the product of the multiplier 110_2 (6_{10}) with the multiplicand 101_2 (5_{10}). As demonstrated in the diagram, three partial products are formed by multiplying the multiplicand with the multiplier and shifting the result appropriately. At the end, all intermediate partial products are summed and this forms the overall result of the multiplication. In this case, taking the numbers as unsigned, the result is 011110_2 which is 30_{10} as expected. A high-level hardware diagram of this multiplication algorithm is shown in Figure 7.1.

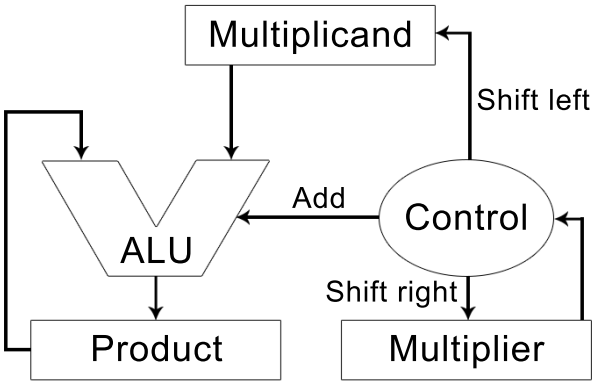


Figure 7.1: Simplified hardware realisation of a multiplier

Despite the fact that this algorithm is easy to design, it is not usually implemented in this form. This is because it is rather inefficient. In sequential mode it requires N cycles and N additions to produce the result. Here N is the number of bits of the multiplier operand. In a parallel implementation, it can produce an output in one cycle at an expense of more hardware usage. Looking at the partial products produced as a result of multiplying 110_2 and 101_2, it can be seen that a row is full of just zeros. This can be optimised especially since the multiplier doesn't change. Multipliers are constant for this project because in the MAC algorithm, they are the coefficients of filters which are fixed throughout the filtering process. Performing radix conversion of the multiplier, the number of additions and shifts required can be reduced. In the case of the 110_2 multiplier, it is equivalent to $2^2 + 2^1 = 6_{10}$. This is a left shift of two followed by a left shift of one of the multiplicand and then the summation of the two partial products produce the output. Therefore the number of additions has reduced by one. Generally, this method can be very efficient provided the multiplier can be decomposed into a small number of shifts and additions.

Another method of reducing operations in a simple shift and accumulate routine is through the use of Booth's recoding algorithm [71]. The principle behind this is that an integer can be represented as additions as well as subtractions of powers of two. Computing the difference between binary numbers can be more efficient than just summations. Also as the two's complement method is used in hardware, there is no additional cost for subtracting. To demonstrate the power of this encoding algorithm, an example of the multiplier, 01111_2 which is 15_{10} is taken. In radix conversion format this will be equivalent to $2^3 + 2^2 + 2^1 + 2^0$. This is a total of four summations and shifts. However by using Booth encoding, it can be reduced to just two, $2^4 - 2^0$. Hence, the number of additions and shifts required have been halved from the radix conversion method.

Algorithm 4 illustrates how the encoding process is implemented. When the current and previous bits encountered are the same, then no shift or summation is needed. Only when the pairs of bits differ, recoding occurs. Inefficient encoding happens when the set of input bits have lots of alternating bits. Considering the decimal number 21_{10}, its binary equivalent is 010101_2. From observing its binary representation, it is quite obvious that this multiplier can be implemented

in as little as three shifts and additions. Booth's radix-2 encoding however ends up requiring six shifts and additions instead of just three due to the alternating nature of the bits. This issue can be solved by applying radix-4 encoding. It involves not only looking at the current and the past bits but also one bit ahead to determine the encoding required. The truth table of this encoding is shown in Table 7.1. It overcomes the shortcomings of the radix-2 encoding to provide a more optimal solution.

Algorithm 4 Implementing Booth's Radix-2 algorithm

procedure ENCODING($input, output$)▷ Encodes the input bits. Should be in two's complement format
  for i `= 0 to` $numBits$ `step 1` **do** ▷ Go from right to left. Assume bit -1 is 0
    if $input_i == 0$ && $input_{i-1} == 1$ **then**
      $recode \leftarrow 1$
    else if $input_i == 1$ && $input_{i-1} == 0$ **then**
      $recode \leftarrow -1$
    else
      $recode \leftarrow 0$
    end if
    $output_i \leftarrow recode$
  end for
end procedure

$i+1$	i	$i-1$	**Radix-4 Booth encoding**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Table 7.1: Radix-4 Booth's encoding truth table

Given there is enough memory available, the results of multiplication can be pre-computed and stored in memory. No multiplier may need to be synthesised in this case as the RAM will form a look-up table for multiplication. A 25×16 bit multiplier is required for this project. Roughly speaking around, $2^{25} \times 2^{16}$ 41-bit wide locations will be needed for storage. The available RAM on the FPGA is only 36 kB which is much smaller than the required storage for a complete look-up table. There is no RAM on the CPLD so this method cannot be used with this device. Instead of storing the end results, only intermediate partial products could be stored in the FPGA RAM to aid the overall calculation. The flaw with this idea is the fact that the solution will be quite slow due to the latency involved in accessing memory.

Other more complex implementations of the multiplier include the fast Wallace tree multiplier [72]. It is based on performing reduction of partial product layers. Tree structure implies $O(logN)$ layers. Full adders and half adders are used to reduce the partial products. Its implementation is depicted in Figure 7.2. There is also the Dadda multiplier which is largely similar in structure to the Wallace

tree multiplier [73]. It is generally wise not to design and implement these complex structures ourselves. This job is more suited for the compiler. It will find the most optimal implementation of a multiplier given certain area, speed and power constraints. Instead it is more worthwhile to focus on designing multipliers that require some form of prior high-level knowledge on the structure of the numbers used in the algorithm. This may not be available to the compiler.

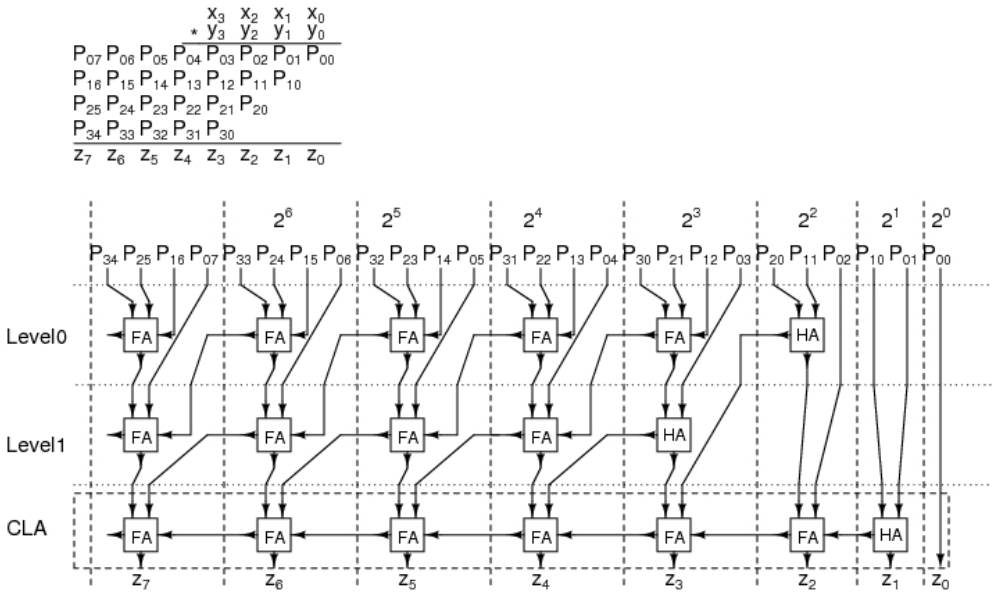


Figure 7.2: Wallace tree 4×4 multiplier [74]

Adders can also be optimised. Different forms exist. There is the ripple-carry adder where the carry bit of each full adder passes through to the next full adder (see Figure 7.3). Rippling does mean that the critical path is quite long which grows with the length of the adder. So this method can be slow. Carry lookahead logic can be added to counteract this issue. Concepts of generation and propagation are displayed in Equations 7.1 and 7.2. Using these, the carry can be calculated quickly and thus the speed of the adder is much higher, at the expense of more gate usage. Again it is better to let the compiler decide which form of adder is most optimal under the circumstances.

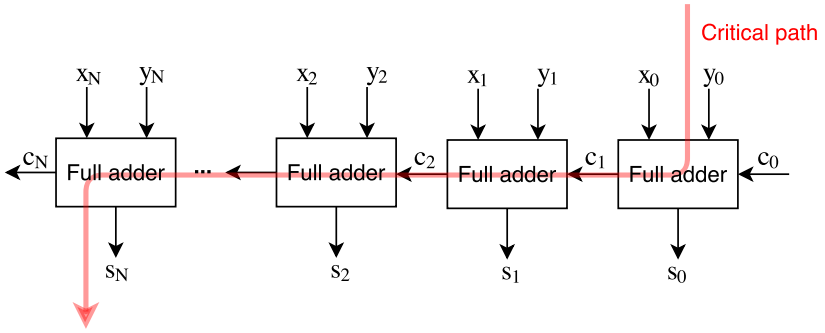


Figure 7.3: Ripple-carry adder

$$G_i(X,Y) = X_i \times Y_i \tag{7.1}$$

$$P_i(X,Y) = X_i \oplus Y_i \tag{7.2}$$

$$C_{i+1} = G_i + (P_i \times C_i) \tag{7.3}$$

Thus in the following subsections only the designs of multipliers that require this prior knowledge such as the Booth encoding and radix decimation will be considered. The hardware usage as a result of implementing these multipliers will be noted and compared. Unless stated, a single quantised coefficient multiplier will be created. In the quantised rational fraction form, the numerator of the coefficient will be multiplied by the input and the outcome of this will be divided by the denominator of the fraction.

Default multiplier

To begin with a multiplier is synthesised behaviourally without stating its implementation. Here the multiplication is represented as an abstract symbol "*" in VHDL. The compiler is given complete freedom in deciding how the multiplier should be optimally implemented in hardware. A schematic of the synthesised form of this multiplier is shown in Figure 7.4. This consists of a simple multiplier block whose structure isn't controllable by the designer followed by a shift register to perform the division.

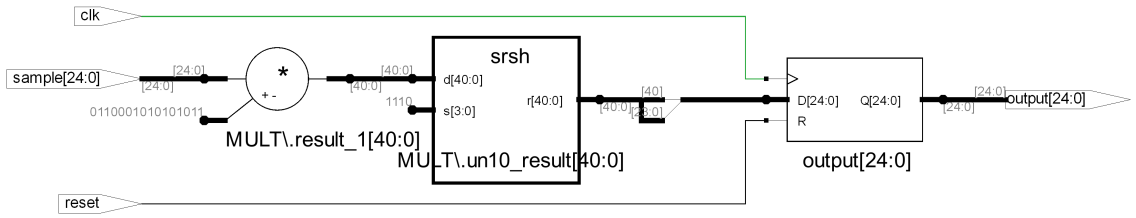


Figure 7.4: Synthesis of the default multiplier

Investigating the impact on the hardware resources, it is found that around 11% of the FPGA's core cells are used. Taking into account the fact that more multipliers will be required in the filtering stage, this usage can be considered fairly high. Moreover this design isn't capable of fitting on the CPLD. It uses 349 logic functions while only 256 are available (136% usage). Hence, especially for the CPLD another method has to be employed.

Radix conversion

Pre-processing needs to be carried out before this technique can be applied in hardware. The creation of appropriate shifts is required. In addition, to account for negative coefficients, it is also necessary to indicate when to subtract in the shift and accumulate scheme. Both of these are generated by running a MATLAB script and are added to a VHDL package for use.

The worst case scenario of all the filtering coefficients used in this project is taken. This turns out to be a zero coefficient from the one channel 500-900 Hz band-pass filter. Radix conversion of this zero coefficient results in fourteen shifts and additions. A part of the synthesised radix conversion multiplier is presented in Figure 7.5. As expected there are a lot of wires used in the design because of the large amount of shifting and summations involved.

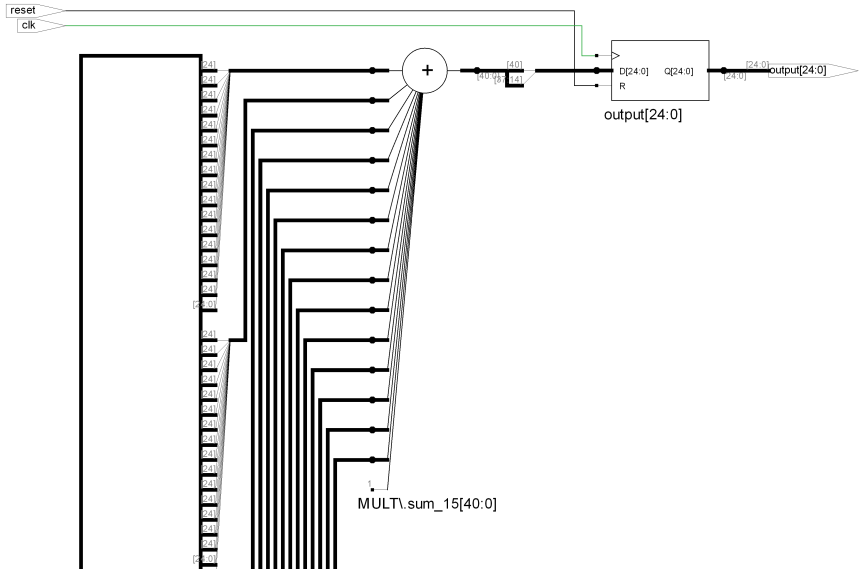


Figure 7.5: Synthesis of the radix conversion multiplier

In terms of hardware usage, it has actually slightly reduced on the FPGA compared to the default multiplier. It uses approximately 597 of the core cells which translates to 10% of total resources. However it fares far worse on the CPLD. The radix conversion multiplier exceeds the available logic by a massive 212%. Therefore it can be deduced that the generic multiplier is being optimised by the compiler in order to reduce area usage for the CPLD.

Radix-4 Booth's recoding

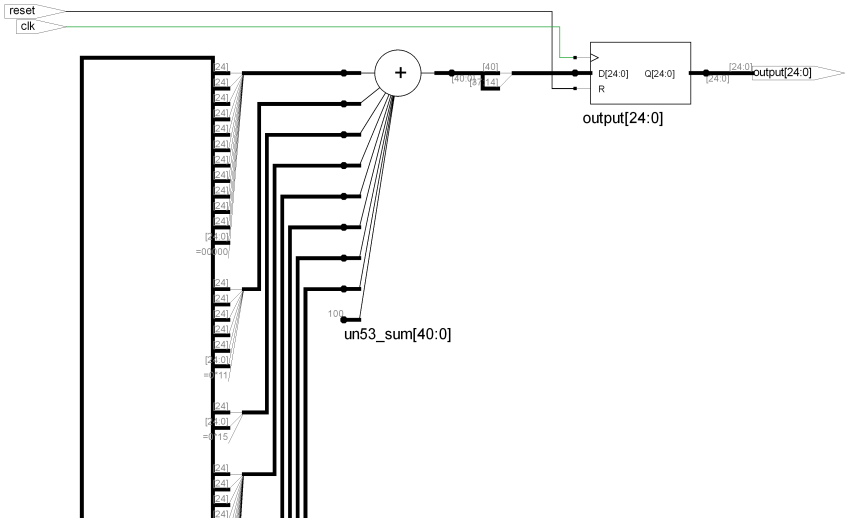


Figure 7.6: Synthesis of the radix-4 Booth multiplier

Like the radix conversion multiplier, this also requires the generation of shifting and negations variables. MATLAB is used to create these and they are stored in a VHDL package. Again the worst case conditions across all the filter coefficients are considered. The highest decomposition arises from the one channel 500-900 Hz band-pass filter pole coefficient. Applying Booth's recoding

to the quantised coefficient leads to eight shifts and summations being needed. This is far less then the fourteen shifts and summations required for the radix conversion multiplier.

The implementation of this multiplier is illustrated in Figure 7.6. A drastic reduction in the density of wires can be seen. Unsurprisingly this is a direct effect of the reduction in shifts and summations. The benefit of this method is that the hardware usage of the FPGA has now reduced to about 7%. Although there is also a reduction of design area on the CPLD, this multiplier is still too large for it to fit on it. 134% of the available board is needed to implement this multiplier.

Sequential multiplier

Since the CPLD is incapable of handling all the shifting and summing required to perform a multiplication in one clock cycle, a sequential method is instead constructed. This will perform just one shift and addition per cycle. As a consequence of this, it may require a huge number of cycles to implement filters that demand a large amount of multiplication. Thus in order to meet the real time constraints, a high clock frequency may be needed.

The structure of this sequential multiplier will be based on the radix-4 Booth multiplier for efficiency. Firstly a simple constant left shift and right shift circuit is designed. This is displayed in Figure 7.7. It shows some promise as it only consumes 11% of the available CPLD hardware. Upon adding logic for storage of intermediate products between cycles, the synthesised design again exceeds the area of the CPLD. Hence, unfortunately this CPLD simply does not contain enough logic to implement the large multipliers and adders required.

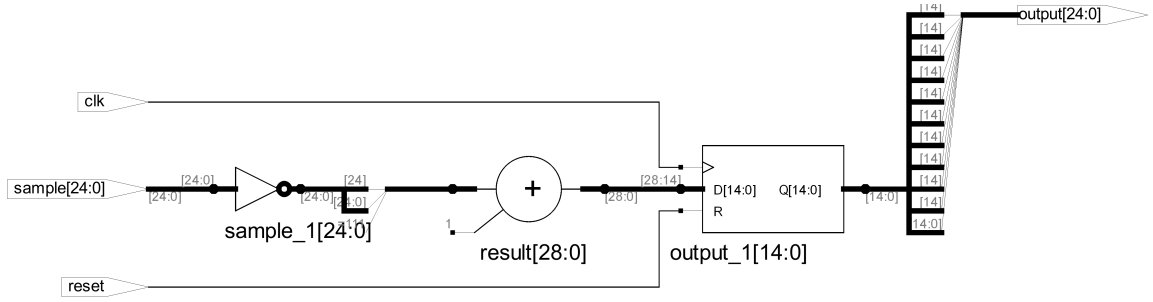


Figure 7.7: Synthesis of a constant sequential radix-4 Booth multiplier

Summary

The results from this investigation have prematurely ended the development of a solution for the CPLD. This is because a single multiplier cannot fit on this device. Since multiplication forms a key component of the MAC algorithm used in filtering, none of the deliverables can be implemented on the CPLD. The chosen CPLD has 256 macro-cells, alternatives offer as high as 512 which still seems insufficient. Therefore it can be concluded that a CPLD is only suitable for low complexity designs, not for DSP applications.

By contrast the FPGA has shown to be a much more powerful platform. It is capable of implementing any form of the multiplier. Comparing the different types of multipliers, the radix-4 Booth implementation is best under the worst conditions. The total additions and shifts required

for each filter are also calculated for the radix conversion and radix-4 Booth multiplier. Figure 7.8 displays the results. Apart from the 53 mHz low-pass filter, the radix-4 Booth version consistently achieves the same multiplication with less number of operations.

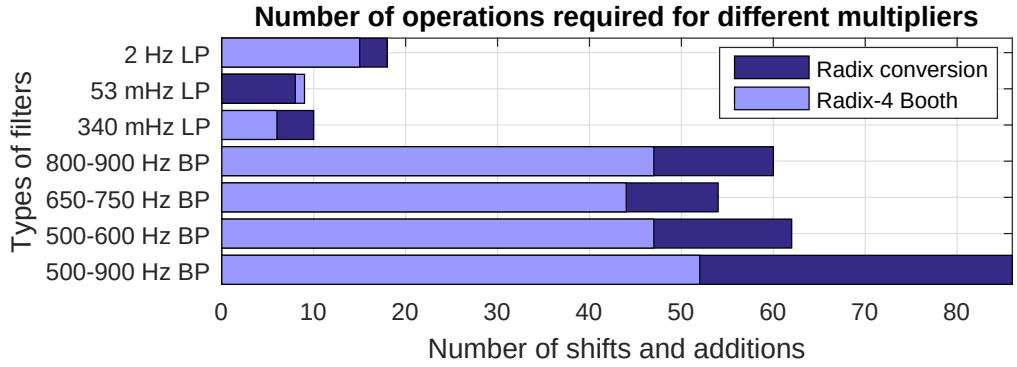


Figure 7.8: Comparison of the number of operations used

Area usage of different synthesised multipliers are summarised in Table 7.2. The radix conversion method can immediately be scrapped in favour of the more efficient radix-4 Booth multiplier. Although this implementation is even more efficient than the default multiplier, this may not be the case when more multipliers and logic are added. The compiler is likely to optimise the design of the default multiplier to make its size smaller. As a result, it is defiantly worth investigating how the area of the radix-4 Booth multiplier scales in the filtering stages.

	Default multiplier	**Radix conversion**	**Radix-4 Booth's encoding**
Area used	11%	10%	7%

Table 7.2: Area used in the FPGA under worst case scenario for different multipliers

7.2 Three channel filtering

The three channel filtering solution is composed of several filtering stages. Firstly the breathing signal is passed through band-pass filters of varying pass-bands. This will be implemented in hardware as MAC blocks. Next the filtered signal is rectified. In hardware, this procedure is very simple because of two's complement notation. From here, the rectified signal is low-pass filtered. This is again implemented via a MAC block. Last of all, the signal is decimated by twenty-one. This decimation can be carried out by using an up counter which wraps counting at twenty-one.

This whole filtering mechanism has to be carried out for three channels. As a result, a lot of hardware may be needed to implement it all in parallel. Therefore the feasibility of implementing this system in a single cycle is explored.

7.2.1 Single cycle

The aim here is to perform all the computation in parallel so that the result is available in just one clock cycle. Referring to Figure 7.9, the major hardware components include three delay lines for the band-pass filters. These have a depth of five (for a 4th order filter) with a width of 25-bits for

each channel. There is also three delay lines for the low-pass filter with a depth of two and width of 25-bits each. Each channel requires two MAC blocks. The first of which implements band-pass filtering. Here nine 41-bit multipliers are required as well as nine adders. The second MAC block is used for the low pass-filtering. It consists of three 41-bit multipliers and three adders. Figure 7.10 represents the MAC block in more detail. It contains multipliers and an accumulator. Taking the band-pass filtering as an example, it will contain nine of these multipliers in parallel.

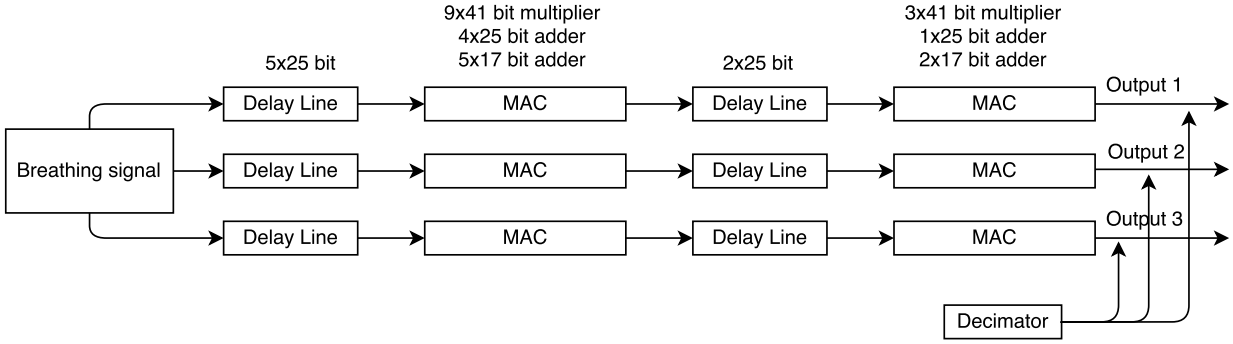


Figure 7.9: Hardware overview of the single cycle three channel filtering

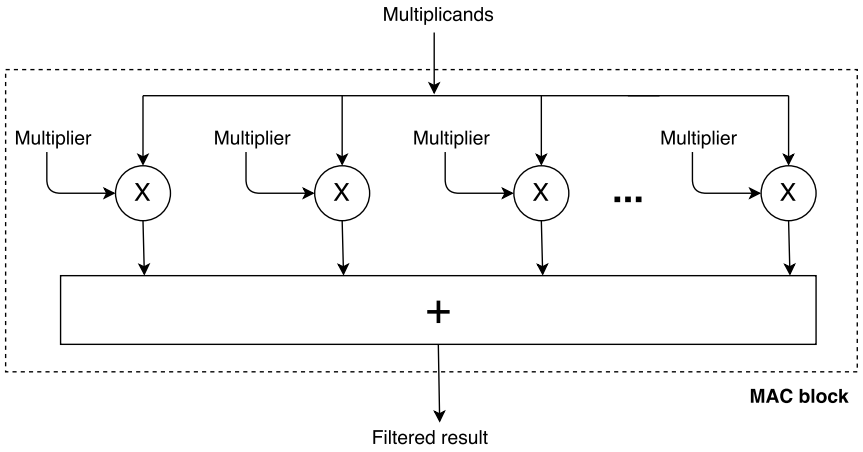


Figure 7.10: Illustration of a MAC block

Storage	Multipliers	Adders (25-bit)	Adders (17-bit)
21×25 bit	36×41 bit	15×25 bit	21×17 bit

Table 7.3: Resources required for parallel implementation

	500-600 Hz BP	650-750 Hz BP	800-900 Hz BP	2 Hz LP	Total
Pole	28	25	28	7	88
Zero	21	21	21	8	71
Total	49	46	49	15	**159**

Table 7.4: 41-bit additions and shifts required for the three channel radix-4 Booth multiplier

All this logic is likely to take up a lot of area on the FPGA. Table 7.3 summarises the resources needed for this design. There are two types of multipliers that could be used. Although the default multiplier has shown a higher usage of gates than a Booth multiplier, as the design becomes more

complex, the compiler may make it more efficient. In the Booth multiplier, the 36 multiplies can be broken down into shifts and additions highlighted in Table 7.4. With a total of 159 additions and shifts required, the question remains whether this is more efficient than the 36 multipliers used in the default multiplier solution. This will now be analysed.

Upon designing and verifying the parallel three channel design's behavioural correctness, it is fitted onto the FPGA. As feared, the parallel implementation of both the default and Booth multiplier solution are far too large. The default multiplier takes up about 320% of the FPGA while the Booth multiplier consumes 319% of the area. The Booth multiplier is only very narrowly more efficient in area. Extrapolating the default multiplier gate usage from one multiplier which was around 11% to the usage when thirty-six multipliers were synthesised which is expected to be over 390%, it is clear that some optimisation is taking place. As suspected, the compiler uses a more efficient implementation of the multiplier when there is scarcity in resources.

7.2.2 Multiple cycles

Breaking down the parallel design into a sequential one is necessary since the single cycle implementation requires far more hardware than there is available. A side effect of this is the higher clock frequency required to meet the real-time timing constraint. This may have a negative impact on the power efficiency (see Section 7.6 for more detail). Since the previous parallel implementation exceeded the space on the FPGA by quite a large amount, heavy down-scaling of the design will be needed. A cut-down sequential design is illustrated in Figure 7.11. This requires twenty-one cycles to perform the whole filtering process. The number of multipliers has been reduced from thirty-six to only two and this is the same case for the additions too. Storage requirements remain the same. Delay lines of the low-pass and high-pass filtering of each channel have been merged. This is done for convenience. Control logic is required for the correct selection of delay lines, channels and coefficients for multiplication each cycle.

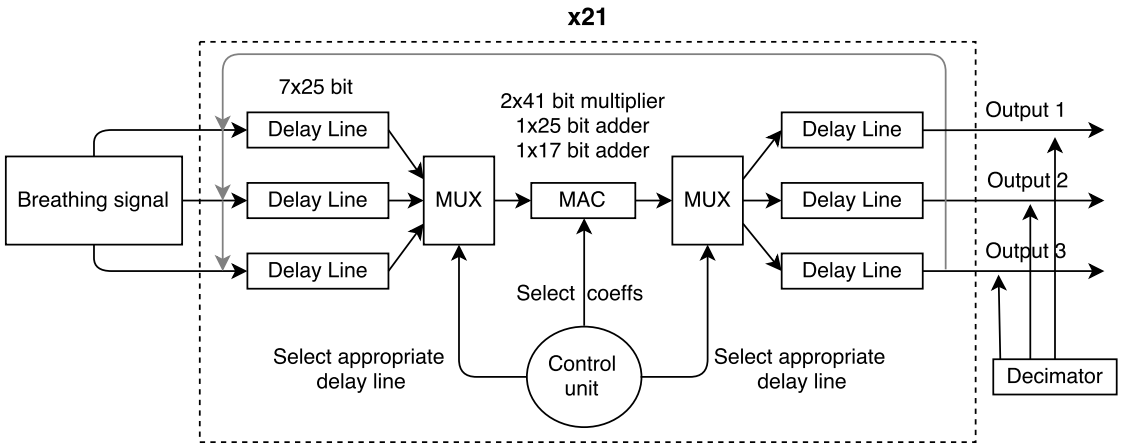


Figure 7.11: Hardware overview of the twenty-one cycle three channel filtering

Synthesising this design with the default multiplier configuration, there is a reduction in area compared to the parallel design. However it only decreases to 158% which is still not enough to fit on the FPGA. The Booth multiplier is also tried in place of the default one. Still the usage is too high, in fact its even higher than the default multiplier at 243%. Sequentially it is no longer beneficial to use the Booth multiplier since the shifts and adds have to be sized for the worst case conditions. For the three channel solution there is a maximum of eight shifts and adds per

coefficient required. As a result instead of performing the 159 shifts and adds in parallel form, there will be 336 shifts and adds. Therefore the Booth multiplier is no longer efficient.

Even though the arithmetic operations have been vastly reduced in this sequential method, the usage still remains high. Thus it can be deduced that the delay lines used for storage are the main culprit. By sacrificing precision, the number of bits for the delay lines can be reduced. Doing so, the size of the delay lines is shrunk down to 16-bits and the multipliers are now 32-bit. Although this again leads a reduction in hardware usage, there is also a severe hit on the performance of the filtering. In Figure 7.12 the error between this low-precision solution and the double-precision floating-point version is demonstrated. All three channels suffer from very high errors due to this precision loss. Inaccuracies make this technique not worthwhile.

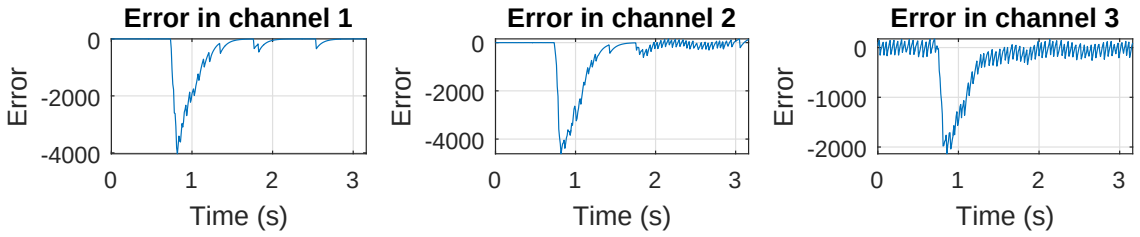


Figure 7.12: Precisions errors from reducing the size of the tap delay line

7.2.3 Discussion

Multiplier type	Area	Cycles	Comment
Default	319.81%	1	Too large to fit
Radix-4 Booth	318.93%	1	Only slightly smaller than the default multiplier
Default	157.62%	21	Executing over multiple cycles reduces usage
Radix-4 Booth	242.85%	21	This method is no longer optimal
Default	110.89%	21	Reduced size of delay lines. Accuracy is poor

Table 7.5: Area usage of different three channel filtering implementations

Table 7.5 provides a summary of all the different attempts at designing the three channel filtering solution. In general spreading the design over multiple cycles has the advantage of reducing hardware usage. Also the default multiplier fares better than the radix-4 Booth multiplier under a sequential design. However it turns out that the storage costs needed for accurate filtering are too high. As a result it is not possible to fit the design on the chosen FPGA.

Other resources on the FPGA include RAM blocks. These are depicted in Figure 7.13. They can be considered for storage of delay lines. This may lead to a bit of latency in the design due to the accessing and writing times of the RAM. Looking at the FPGA's specification the following SRAM configurations are allowed, 256×18, 512×9, 1k×4, 2k×2, and 4k×1. They are in the format depth×width. The maximum width available of 18-bits isn't sufficient as a width of 25-bits is required. Thus RAM blocks cannot be employed for storage of delay lines here.

As a last resort, the only option remains to sacrifice some precision of the calculations. Using less optimal quantising of coefficients and smaller delay lines, the area usage can be reduced. The pre-multiplying of the input can also be removed. Accuracy will inevitably go down as a consequence of these alterations.

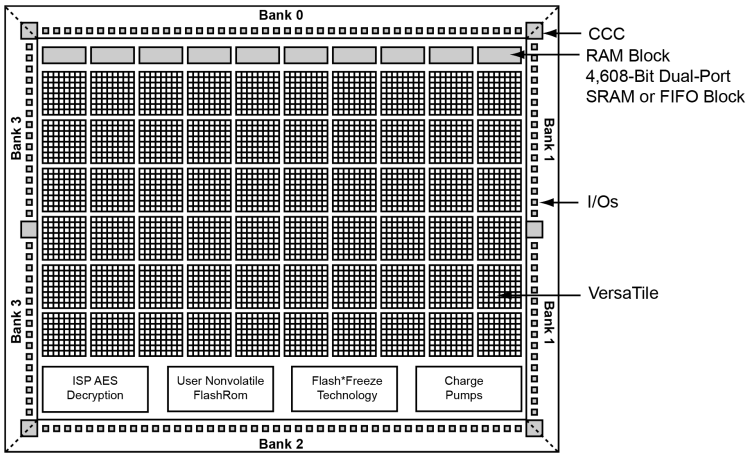


Figure 7.13: Available RAM blocks on the FPGA [59]

7.3 One channel filtering

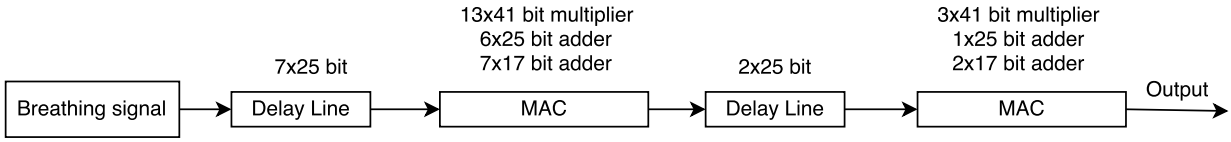


Figure 7.14: Hardware overview of the single cycle one channel filtering

Compared to the three channel filtering, there is a lot less hardware required here. There is only one channel (see Figure 7.14) in this design instead of the three channels used beforehand. The delay lines, multiplies and adds have increased due to the larger 6th order filter used in the band-pass filtering stage. Synthesising this design based on the default multiplier and mapping it onto the FPGA results in a failure again. This time it only exceeds the hardware available by a relatively small amount of 120%.

The radix-4 Booth multiplier is also considered in place of the default multiplier. The number of shifts and additions required for this design are listed in Table 7.6. The synthesised design is again unable to be placed onto the hardware, exceeding the available space by 122%. This is slightly higher than the default multiplier.

	500-900 Hz BP	**340 mHz LP**	**Total**
Pole	36	4	40
Zero	19	2	21
Total	55	6	**61**

Table 7.6: 41-bit additions and shiftings required for the one channel radix-4 Booth multiplier

The area usage of the different multiplier implementations are summed up in Table 7.7. Both multiplier types lead to the design exceeding the available space on the FPGA. Thus a multiple cycle solution will have to be created. This will be carried out after attaching the breathing detection system at the output of this filtering design for ease.

Multiplier type	Area	Cycles	Comment
Default	120.31%	1	Most efficient multiplier in a parallel design
Radix-4 Booth	121.79%	1	Slightly less efficient in area usage

Table 7.7: Area usage of different one channel filtering implementations

7.4 Full breathing detector

7.4.1 Single cycle

Starting with the fully parallel design, the feature recognition section of the breathing detection algorithm is created. This receives the input from the band-pass and low-pass filtering implemented in the one channel solution. Its output determines whether the subject is breathing or not. Hardware overview of the whole breathing system is presented in Figure 7.15. Following on from the one channel filtering, the filtered signal is split into two channels, the top and the bottom. The bottom channel consists of a MAC block yet again to perform 53 mHz low-pass filtering. Since it is a 1st order Butterworth low-pass filter, only 2×25 bits are required for the delay line. Also three multipliers and adders are needed for the MAC block, similarly to the low-pass filter in the one channel filtering. Its result feeds into comparators which perform thresholding and form part of the decision on whether apnoea is occurring or not.

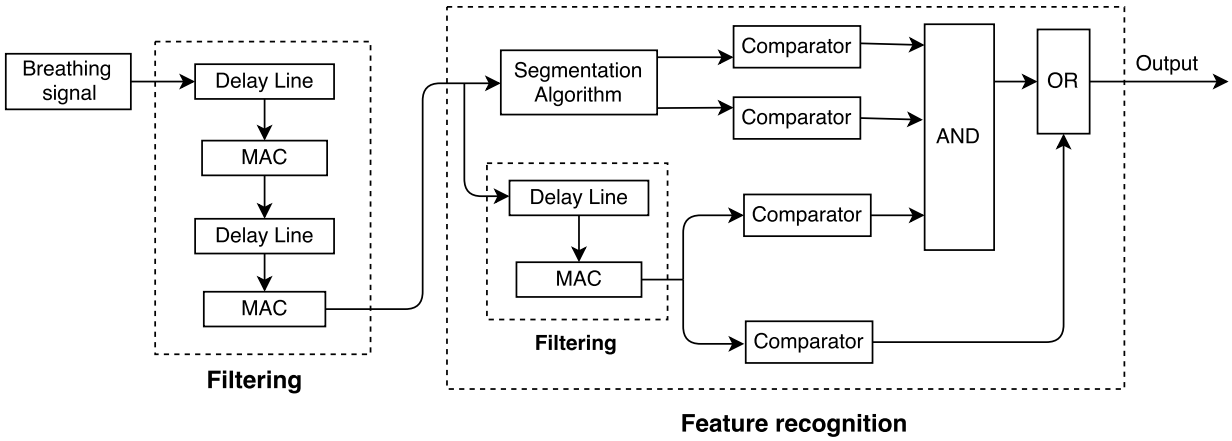


Figure 7.15: Hardware overview of the full breathing detector

The top channel of the feature recognition section firstly comprises of the segmentation block. This is shown in more detail in Figure 7.16. Comparators form the majority of components used in this block. They are used to find the temporal characteristics of the input signal. Adders are also required to calculate the length and integral of the signal. The output from the segmentation algorithm passes through more comparators. Again thresholding is performed and the result is useful in making the decision on the presence of breathing.

Since the one channel solution has already exceeded the capacity of the FPGA, the full breathing detector is not expected to fit on the hardware. The default multiplier configuration is chosen for all the MAC blocks since it performs most optimally in terms of its area usage. Having synthesised the VHDL code for this design, the fitting process is followed. It turns out that the full breathing detector system uses about 157% of the FPGA. Therefore the feature recognition block adds only 37% extra gate usage. Overall the complexity of the system mainly stems from the filtering

process. Consequently a sequential algorithm will have to be developed that reduces the impact of filtering.

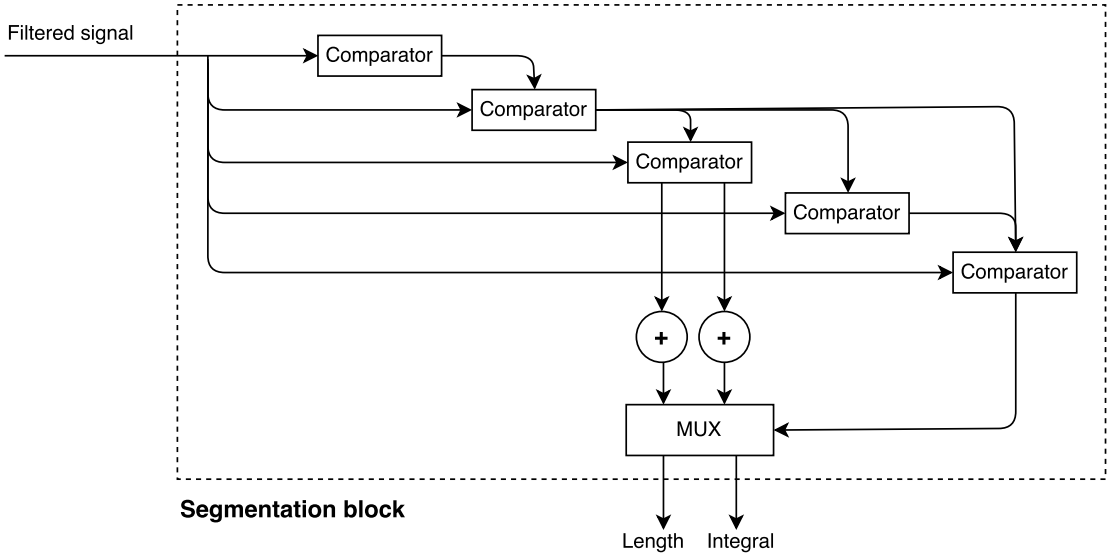


Figure 7.16: Hardware overview of the segmentation algorithm

7.4.2 Multiple cycles

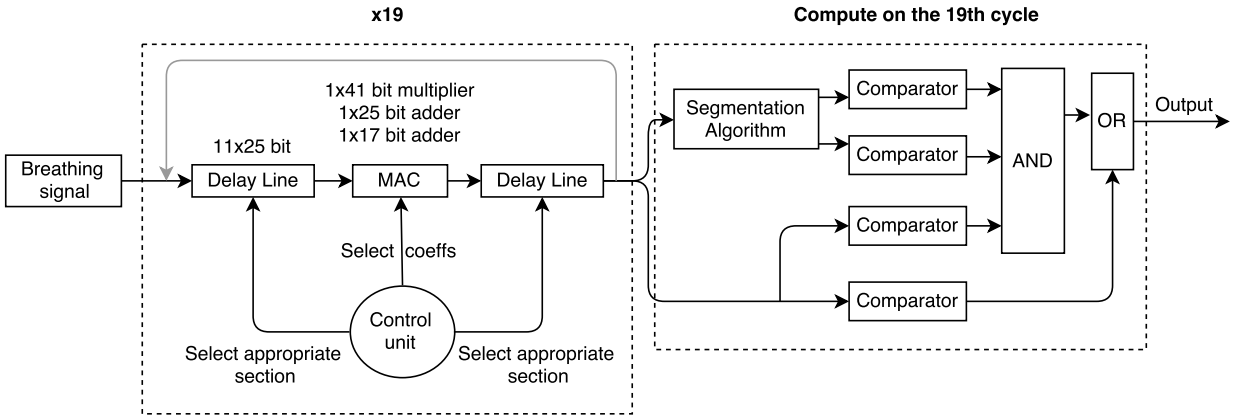


Figure 7.17: Hardware overview of the nineteen cycle full breathing detector

Combining all the filtering together and processing sequentially achieves a much more compact design. Referring to Figure 7.17, the MAC block has been reduced to just one multiply and two adds per cycle. Although a reduction of multiplies from nineteen in the parallel implementation to just one here seems like overkill, it is necessary since there are extra storage costs incurred when a design is made sequential. The delay lines are now eleven in depth as they include the band-pass filter and the two low-pass filters. A control unit is responsible in this design for selecting the correct coefficients, delay elements and storage of results. A total number of nineteen multiplies are required for the whole filtering system, thus nineteen cycles are needed.

On the 19th cycle the part of the feature recognition algorithm without the filtering block comes into play. This is because only at this point the necessary signals are available for it to process. As this block had a low area footprint, it's low-pass filter is only merged with the rest of the filtering block. Otherwise all the other parts of the feature recognition section are carried out

in parallel. Finally this algorithm is converted to VHDL code and is synthesised and compiled. The compilation results show that this design is within the FPGA's capability, i.e. it successfully fits, consuming a large 88% of the FPGA. Rigorous testing of this design is now performed in the following subsection.

7.5 Testing

Figure 7.18: Testing process used in behavioural verification

Behavioural verification is performed in ModelSim. The flow for this procedure is presented in Figure 7.18. Going through this flow from left to right, the first stage involves generating an input file for the test-bench to read from. This test file contains the breathing signal and is created on MATLAB. The test-bench then reads from this file and provides it as an input to the DUT. When it receives outputs from the DUT, it writes them to a file. This file can then be accessed by MATLAB which uses it to compare against a ground truth value and works out the error between the two values. If this error is zero then the design behaves as expected. Otherwise adjustments have to be made to the VHDL code. The ground truth value in this case is the fixed-point version of the full breathing detector.

The input signal will be provided to the DUT at a rate of 2205 Hz. A clock frequency as high as 20 MHz is possible by the oscillator on the FPGA. The minimum clock frequency required to meet the real-time signal constraints will be 19×2205 which results in a frequency of 41895 Hz. This is much lower than the clock that is available therefore a clock divider will need to be made. To make the design more robust, changes in frequency of the input signal can also be accounted for. Designing the clock divider for a maximum possible frequency of 3000 Hz, leads to 136% margin above the nominal 2205 Hz frequency. This will result in the clock divider producing a frequency of 19×3000 = 57000 Hz or 57 kHz. Another corner case is also considered, which is when the input frequency is lower than 2205 Hz. Logic is added to deal with this scenario. Hence the range of frequencies that can be dealt by this design are, 0 Hz < Input frequency < 3000 Hz.

7.5.1 Pre-synthesis verification

Utilising the process described before, verification of the pre-synthesised design is carried out. A waveform view of the simulation is provided in Figure 7.19. Some signals of interest include the 'sample' signal which reads breathing data from a text file and passes it onto the DUT every 2205 Hz (negative edge of 'clk3'). The 'ready' signal is used to indicate to the DUT when a new input signal is available. It too oscillates at 2205 Hz. Most importantly, the output from the DUT is named 'breath_out'. This is stored in a text file by the test-bench.

After running the simulation until all the samples have been fed through to the design, the outputs from the test-bench are compared to the ground-truth. There is zero error between the software

fixed-point algorithm and the hardware implementation. Therefore the hardware design performs as desired.

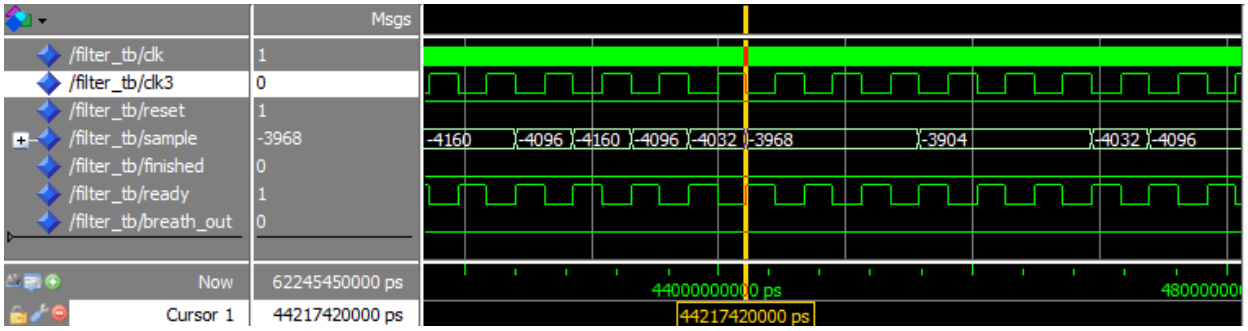


Figure 7.19: Simulation of the pre-synthesis design

7.5.2 Synthesis

Synplify Pro is used to synthesise the behavioural code into a hardware gates format. The breathing detector design successfully passes synthesis without any errors. Though there are some warnings. Researching the warnings which all have the ID 'CL169' (see Code 7.1), it seems they are expected. This is because of the fact that variables in a clocked process form a flip-flop which is removed during optimisation. The compiler deems their synthesis unnecessary as they are only used for storing intermediate values.

```
1 @W: CL169 :"E:\Imperial\Project\FPGA\Report\MultiCycleFullBreathingFinal\hdl\...
      filter.vhd":57:11:57:16|Pruning register result(40 downto 0)
2 @W: CL169 :"E:\Imperial\Project\FPGA\Report\MultiCycleFullBreathingFinal\hdl\...
      filter.vhd":58:11:58:18|Pruning register sample25(24 downto 0)
3 @W: CL169 :"E:\Imperial\Project\FPGA\Report\MultiCycleFullBreathingFinal\hdl\...
      filter.vhd":61:11:61:19|Pruning register a5_result
```

Code 7.1: CL169 Synplify warnings

The highest possible working clock speed for the clock divider is dependant on the critical path of the design. It is estimated to be 3.7 MHz. The clock divider is designed to be oscillating at 57 kHz which is well within the maximum frequency that is feasible. The whole critical path of the design is displayed in Figure 7.20 (also in Figure 13.7 in the Appendix). There are quite a lot of components that contribute to this. A more clearer picture of these is shown in a zoomed in version of the path in Figure 7.21. Gates used in the critical path are visible in this view.

Figure 7.20: The whole critical path of the synthesised design

A more detailed breakdown of the gate usage in the FPGA is available in the Appendix (Code 13.1). Moreover the synthesised register-transfer level (RTL) view of the whole breathing detector system is also shown in the Appendix (Figures 13.8, 13.9, 13.10 and 13.11).

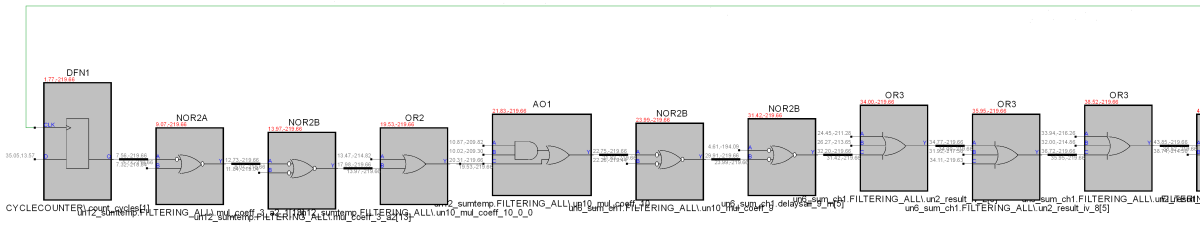


Figure 7.21: Part of the critical path of the synthesised design

7.5.3 Post-synthesis verification

After synthesising the design, it is loaded onto ModelSim and a simulation is again performed. The results of this simulation confirm that the design still performs as expected. Hence no changes need to be made to the VHDL code.

7.5.4 Coverage

Enabling coverage in the ModelSim simulator gives access to additional data to aid the design and verification process. Information on how well the testing is performed, as well as how the DUT reacts to this testing is presented in the form a report. This helps the designer establish whether the design has been fully simulated and is working.

Coverage type	Bins	Hits	Misses	Coverage (%)
Statement	96	96	0	100.00
Branch	54	53	1	98.14
FEC Expression	5	5	0	100.00
FEC Condition	27	20	7	74.07
Toggle	34	33	1	97.05
Weighted Average				97.05

Table 7.8: DUT coverage details

Coverage type	Bins	Hits	Misses	Coverage (%)
Statement	37	33	4	89.18
Branch	10	9	1	90.00
FEC Condition	2	0	2	0.00
Toggle	42	36	6	85.71
Weighted Average				66.22

Table 7.9: Test-bench coverage details

The simulation is again driven by the breathing input signal. In order to produce useful coverage results, the simulation is left running for a long while. After a period of 102 seconds (225 k samples), the simulation is stopped and the coverage report is viewed. The results for both the DUT and test-bench coverage are presented in Tables 7.8 and 7.9 respectively. A summary of the results is also provided in the Appendix (Figure 13.12).

Statement coverage shows whether all the statements in the design have been covered. For the DUT, all the statements have been hit therefore there is no redundant code. In the test-bench there are 4 missed statements. Examining these further, they are part of the finishing process which had not come into play yet as there were still samples available in the text file. Branch coverage indicates whether the 'IF' and 'CASE' statements have been activated. There is only 1 missed case in the DUT which is an empty 'IF' statement. This is mostly likely removed in optimisation by the compiler. Again in the test-bench the missed branch is due to the testing not being completed.

Then there is Focused Expression Coverage (FEC) which looks at how different combinations of the input signal affect the output. All input pattern have occurred during the simulation in expressions therefore the FEC expression coverage is 100% for the DUT. However the FEC condition coverage has some misses. This means although all branches are taken, not all of the conditions in the branches have been fully tested. This is a similar case for the test-bench.

Finally there is the toggle coverage. The different states of input and output signals are recorded here. The DUT displays very high coverage percentages. On the other hand the test-bench has a slightly lower percentage. Looking into this more closely, in the test-bench a counter has been allocated two more bits than necessary which can be removed to improve the design.

7.5.5 Place and Route

Compiling of the design is carried out in which net-list optimisations are performed, as well as confirmation of the design fitting on the target FPGA is given. As stated before, the nineteen cycle breathing detector does indeed fit onto the hardware with a usage of 88.43%. This is followed by the assigning of pins. Fourteen out of the sixty-eight pins available on the FPGA are assigned. Figure 7.22 shows the locked pin layout. The pins with a black background represent those that are assigned.

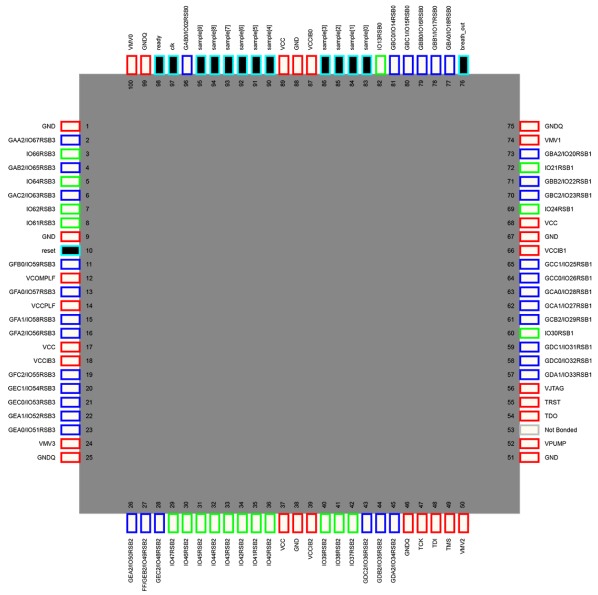


Figure 7.22: FPGA pin assignment

The design is then placed on the FPGA in the layout process. There are several layout options available which affect how the logic is mapped onto the FPGA. This is investigated more in the

power consumption section later on. Currently a timing-driven layout is selected whose primary goal is to meet the timing constraints.

7.5.6 Timing analysis

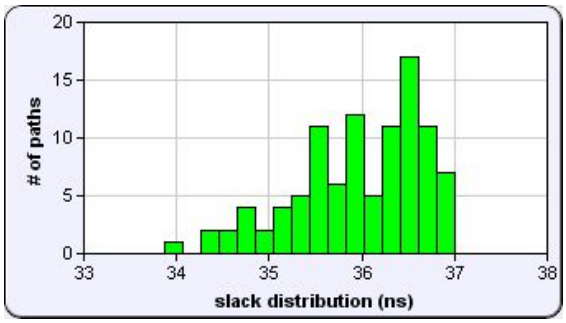


Figure 7.23: 20 MHz max delay slack

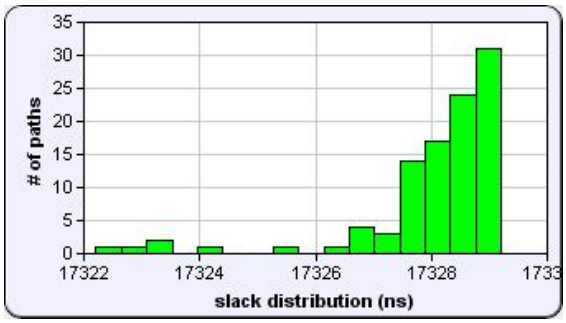


Figure 7.24: 57 kHz max delay slack

Static timing analysis is now performed. Timing constraints for the input clock which is 20 MHz and the clock divider which is 0.057 MHz are added. Then maximum delay analysis is carried out. The slacks for the register to register path for both these clocks are displayed in Figures 7.23 and 7.24. Since there is no negative slack in either of the figures, there are no timing violations. Slack is calculated as the difference between the required time of arrival and actual time that the signal arrives. A positive slack therefore means that the design meets the timing constraints. The clock divider shows the largest positive slacks which implies it meets the timing with the biggest margin. Previously the synthesis tool had estimated a maximum clock speed of 3.7 MHz for the clock divider, since it is set to 0.057 MHz instead, it is no surprise that the slacks are so very positive.

7.6 Power consumption

Power consumption is an important topic for this project. The underlying importance of power efficiency stems from the desire to create a breathing detection system that is small and mobile. Therefore for convenience of use its battery must last as long as possible. As the FPGA is based on the low voltage complementary metal oxide semiconductor (LVCMOS) technology, dynamic power dissipation mainly contributes to its power consumption. Every time a transistor switches from a low to high or high to low state some energy is dissipated. This is because the capacitance have to be charged or discharged. This is known as dynamic power. It is dependant on the capacitive load of the circuit, voltage and the switching frequency [75]. The equation for this can be written as:

$$Power_{dynamic} \propto \frac{1}{2} \times Capacitance_{load} \times Voltage^2 \times Frequency_{switching} \tag{7.4}$$

As the power is proportional to the square of the voltage, it is very advantageous to run the FPGA core at a low voltage. Thanks to the LVCMOS technology, a core voltage as low as 1.2 volts is available. Switching frequency also has an impact on the energy consumed by the FPGA. This is why it is preferred for the whole design to be executed in one cycle so that the clock divider can oscillate at a lower frequency.

Another source of power dissipation is static power. Leakage currents flow even if the transistor is turned off. Therefore the power dissipated is proportional to this current as well as the voltage according to Equation 7.5 [75]. As a result a bigger FPGA would lead to a larger static power dissipation due to more transistor leakage currents. Like the dynamic power, the voltage plays an important role here, minimising it reduces power consumption.

$$Power_{static} \propto Current_{static} \times Voltage \tag{7.5}$$

7.6.1 Simulated power

Before programming the FPGA with the compiled design, simulated power from the Libero SoC software is investigated under different scenarios. Unless stated typical conditions are used for this simulation which consist of a junction temperature of 25° Celsius and a core voltage of 1.2 V. Also a clock frequency of 20 MHz and a clock divider of 57 kHz are used.

Layouts

Several types of layout options exist. Timing-driven place and route is the default choice. Its aim is to meet the timing constraints set by the designer. Then there is the power-driven layout whose primary objective is to minimise the dynamic power while still satisfying the timing constraints. As the reduction in power consumption will lead to a longer battery life, this layout will be the more ideal type. Additionally all the different place and routing can be carried out in multiple passes. This means that the layout is run multiple times with different seeds and the best one is chosen. A downside to this may be the extra computation time needed to perform the layout phase however it may offer a better placement than before which will be worthwhile.

The results of running the different layout options are shown in Table 7.10. The power-driven approach performs the most optimally, shaving nearly 0.1 mW off the timing-driven layout in the single pass scenario. Note the timing constraints are still met in this power-driven approach. Multiple pass tends to perform better than single pass even though they take longer to compute.

Timing-driven		**Power-driven**	
Single pass	Multiple pass	Single pass	Multiple pass
0.694 mW	0.634 mW	0.609 mW	0.608 mW

Table 7.10: Total simulated power of design under different layouts

Further analysis is carried out on the multi pass power-driven layout. The total power of 0.608 mW is dominated by dynamic power which is 0.580 mW. Static power only contributes 0.028 mW to the total power consumption. The different types of power consumption are also presented in Figures 7.25 and 7.26. More than 75% of the power usage is based on the net. Inputs and outputs as well as gates consume very little power. Looking at the clock domain, the 20 MHz clock is a massive proportion of the total power consumption.

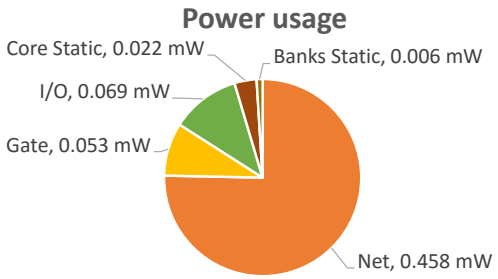


Figure 7.25: Power usage by type

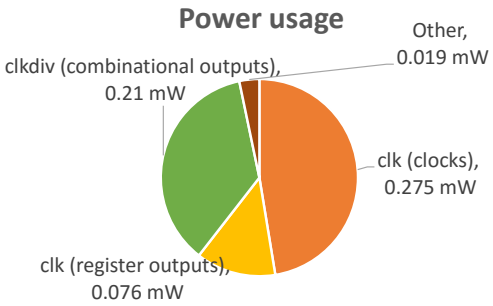


Figure 7.26: Power usage by clock domain

Conditions

Conditions are likely to change as the breathing monitoring system may be used under different weather, seasons and locations. As well as other factors, an increase or decrease in temperature will have an effect on the power consumption of the FPGA. The simulated power for the different conditions is shown in Table 7.11. Best conditions have a junction temperature of 0° Celsius and a core voltage of 1.14 V. All of this leads to a low total power of 0.550 mW. On the other hand the worst conditions are based on a temperature of 70° Celsius and core voltage of 1.26 V. This of course leads to a higher power dissipation of 0.690 mW.

Best conditions	Typical conditions	Worst conditions
0.550 mW	0.608 mW	0.690 mW

Table 7.11: A comparison of power consumption under varying conditions

Clock frequency

The IGLOO nano Starter Kit comes equipped with a 20 MHz clock. As a result this oscillator is used in testing with a clock divider to reduce the frequency down to the required 57 kHz. The clock divider is necessary as the dynamic power is proportional to the switching frequency. A high frequency will thus be a waste of power and does not improve functionality. If a custom design was to be made with only the FPGA chip (AGLN250V2-VQG100) then a 57 kHz clock could instead be used. In this case the clock divider will also no longer be required. As seen before the 20 MHz wastes a significant amount of power. The power saving simulated when a 57 kHz clock is used instead is now investigated.

Under a power-driven layout, there is a huge reduction in power consumption. The total power has now dropped to just 0.045 mW. This is more than ten times lower than with the 20 MHz clock. The dynamic power is just 0.017 mW which is even lower than the static power of 0.028 mW. The pie charts in Figures 7.27 and 7.28 display different breakdowns of this power usage. Core static power is now the largest and the 57 kHz clock draws only 0.008 mW of power.

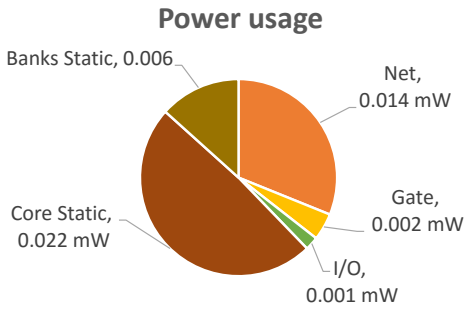


Figure 7.27: Power usage by type

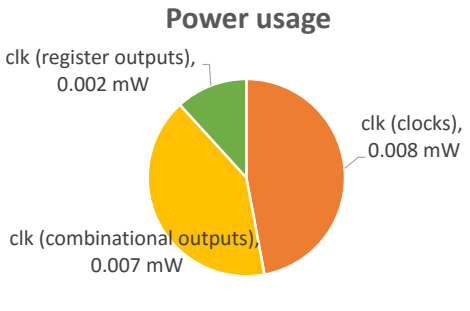


Figure 7.28: Power usage by clock domain

Summary

All the power consumption data is summarised in Table 7.12. The lowest power occurs when the 20 MHz clock is replaced with the 57 kHz clock. This reduces the power consumption down to 0.045 mW. Since temperature conditions cannot be controlled, the optimistic results of the best case are rather meaningless. Instead the typical case provides a much better estimation of what can be expected in the real world. As the name suggests, the power-driven layout provides the least power consumption while still meeting the timing. Also computing multiple passes rather than just one can result in a more energy efficient layout. The FPGA will now be programmed with the design and its power will be measured.

Clock	Clock divider	Conditions	Layout	Passes	Total power
20 MHz	57 kHz	Typical	Timing-driven	Single	0.694 mW
20 MHz	57 kHz	Typical	Timing-driven	Multiple	0.634 mW
20 MHz	57 kHz	Typical	Power-driven	Single	0.609 mW
20 MHz	57 kHz	Typical	Power-driven	Multiple	0.608 mW
57 kHz	None	Typical	Power-driven	Multiple	0.045 mW
20 MHz	57 kHz	Worst	Power-driven	Multiple	0.690 mW
20 MHz	57 kHz	Best	Power-driven	Multiple	0.550 mW

Table 7.12: Power consumptions under different scenarios

7.6.2 Measured power

Although power can be calculated through simulations, it may not reflect the real power consumption of the FPGA. This may be due to a number of reasons. Conditions such as temperature may be different. Transistor models used by simulation could be inaccurate or there may be under/over-estimation of toggling of input bits.

The testing method is highlighted in Figure 7.29. A MCU is used to drive the inputs of the FPGA. It is loaded with a signal which contains parts of breathing and apnoea. The current is then measured across the FPGA chip via an ammeter. As the core is at a constant voltage of 1.2 V, the current is multiplied by this voltage to give the power. Applying this technique for the 20 MHz clock version, the measured power is 0.485 mW. This is slightly lower than expected since

the simulated power is 0.123 mW higher than this. A possible explanation may be that there is an over-estimation of toggling in the simulation. Also the conditions when measuring the power could have differed to the simulated conditions.

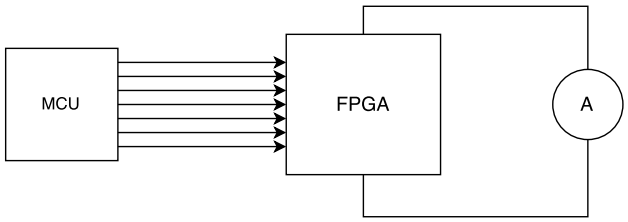


Figure 7.29: Measuring the FPGA's power consumption

Realistically the FPGA will be provided with a 57 kHz clock. Therefore the 20 MHz clock available on the FPGA board can be shut-down. The MCU is programmed to generate this clock frequency for the FPGA. Now the power consumption drops to just 0.068 mW. This time however the measured power consumption is 0.023 mW higher than expected. Static power may be the reason behind this. The simulated power may have used in-accurate transistor models and as a result under-estimated the actual power dissipated.

Chapter 8

MCU Solution

A solution for the micro-controller environment is also designed. As mentioned beforehand the solution can be coded up in assembly. This allows for greater control over how code translates to instructions executed on the chip. However writing assembly can be a time-consuming process and is hard to maintain as code size or complexity increases. As a compromise C, a high-level programming language is chosen instead. Although its high-level nature restricts hardware specific level optimisations, the flexibility and ease of use it provides makes it a powerful and productive language. Moreover compilers allow assembly to be inlined with the C code so the low-level option is still available to the designer.

TI's Code Composer Studio IDE [76] and IAR Embedded Workbench IDE [77] are both compatible with the chosen MCU and are therefore used for code development. They both offer a lot of debugging features. The MSP430FR5969 MCU comes equipped with a eZ-FET on-board emulator. Among other features, it contains the so called EnergyTrace technology. Both the IDE's are able to profile the MCUs power usage through the use of this technology which will be very useful in this project.

8.1 Precision

Once again there is a choice of precision for calculations. Higher precision calculations are likely to be more accurate. However at the cost of consuming more cycles. This claim is tested on the MCU by evaluating the clock cycles taken for a particular multiplication routine. The results of this experiment are summarised in Table 8.1. As the MCU lacks a dedicated FPU, it is no surprise that the floating-point multiplication requires the longest time. A library is used to simulate the floating-point unit which results in a lot of instruction cycles. The fixed-point version is three times faster with the use of the on-board 32×32 hardware integer multiplier. Without this multiplier, even the more efficient fixed-point multiplication would require a massive 201 clock cycles. Fixed-point precision is chosen here due to its lower cycle requirement resulting in a higher efficiency.

Floating-point	Fixed-point	Fixed-point without hardware multiplier
119	39	201

Table 8.1: Clock cycles taken to perform multiplication with no optimisations

8.2 Full breathing detector

8.2.1 Circular buffer

Filtering requires the use of delay lines which are just arrays from a software point of view. The direct form II filter is most suitable for a fixed-point scenario therefore it is implemented. Although the shifting of delay lines required in this form of the filter had not been an issue in hardware, it may be inefficient in a MCU. This is because in software the shifting translates to the copying of memory from one location to another (see Code 8.1). As the order of the filter scales up, the copying needed increases proportionally, thus wasting a lot of clock cycles moving data around.

```
1     for (i=N_BP; i>0; i--){
2         w_BP[i] = w_BP[i-1];
3     }
```

Code 8.1: Shifting arrays in C

Circular buffers avoid this problem. Instead of moving memory, they keep track of where the new sample and past samples are located and adjust each cycle accordingly. As a result the MAC algorithm can be performed without the need for any physical shifting. This technique is perhaps better explained with the aid of a diagram. Two cycles of a circular buffer implementation are shown in Figure 8.1. Here the pointer points towards the latest sample. This is always multiplied with the first zero coefficient (the same idea can be applied for the pole coefficients too). The previous samples chronologically are multiplied by zero coefficients with increasing indices. When the pointer hits the last index, it is wrapped around to the first index (thus the circular name).

A circular buffer version of the one channel filtering and a non-circular version are written in C and then benchmarked. The clock cycles taken for each of these are listed in Table 8.2. With a lower clock cycles of 2819, the circular implementation is much more efficient than the non-circular version. As the overall aim is to produce a power efficient solution, the circular buffer will therefore be implemented.

Non-circular buffer	Circular buffer
3254	2819

Table 8.2: Clock cycles for circular and non-circular buffers with no optimisation

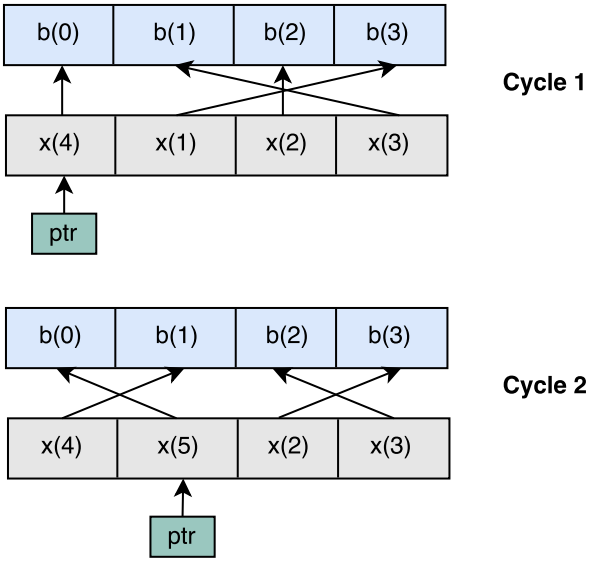


Figure 8.1: Circular buffer representation

8.2.2 Multiplier

The MSP430FR5969 has a dedicated 32×32 bit multiplier. This can be very handy when the operands have a result that is 32-bits or less. However the filtering MAC algorithm requires a destination width of 41-bits. Thus in C the only way to achieve this is to perform a 64×64 bit multiplication. What this means is that the hardware multiplier cannot be directly utilised for the filtering process. As a consequence of this it will be less efficient, taking more clock cycles to produce a result.

64×64 multiplication	**32×32 multiplication**
81	39

Table 8.3: Clock cycles for hardware multiplication with no optimisations

Table 8.3 shows the clock cycles taken for different multiply widths. As suspected the complex 64×64 multiplier takes much longer than a simpler 32×32 multiply (by 42 clock cycles). The current high precision filtering technique needs to be modified to utilise the hardware multiplier. This is essential to make the MCU run faster and provide competitive energy consumption. Going back to the quantising of coefficients, instead of tailoring the fractional lengths of each coefficient, it can be carried out for the filter as a whole. Calculating the MSE between ideal and quantised coefficients, it turns out to be a low value of 1.0072e-07. The MSE of the quantising method where a different fractional width for each coefficient is used was 2.6394e-09 (nearly 100 times lower). An advantage of this new method is that the divider will be constant for the whole filter and division may only be applied at the end, reducing quantisation errors. This method of filtering is illustrated in Algorithm 5.

Accumulation result of the output can then be taken before division, providing a high resolution output. This means that pre-multiplying the input signal by sixty-four (left shift of 6-bits) can be removed. Therefore the delay line can be as small as 16-bits and the result of multiplying with a coefficient will fit a 32-bit wide destination. However there is negative impact on the performance from removing this pre-multiplication. Since the low-pass filtering attenuates the amplitudes, the

Algorithm 5 Implementing Cut-Down Direct Form II MAC algorithm

procedure FILTERING($input, output, outputHigh$) ▷ Filters the input. Provides a filtered output and a high resolution filtered output. w is the tapped delay line

$sum_a \leftarrow 0$

$sum_b \leftarrow 0$

$w_0 \leftarrow input$ ▷ 0th index is 0th delayed sample

for i `= N to 1 step -1` **do** ▷ Perform MAC. N is filter order

$sum_a \leftarrow sum_a + (a_i \times w_i)$

$sum_b \leftarrow sum_b + (b_i \times w_i)$

$w_i \leftarrow w_{i-1}$ ▷ Shift the delayed samples

end for

$sum_a \leftarrow sum_a >> 15$ ▷ Divide by the base (assumed to be 15)

$w_0 \leftarrow w_0 - sum_a$

$sum_b \leftarrow sum_b + (b_0 \times w_0)$

$outputHigh \leftarrow sum_b$ ▷ Output non-divided version for higher resolution

$output \leftarrow sum_b >> 15$ ▷ Divide by the base (assumed to be 15)

end procedure

resolution is reduced. Even the non-divided intermediate summing output is affected by this. Figure 8.2 presents this effect. The fixed-point result now differs quite a lot from the floating-point version. Saturation effects are being observed here due to the low resolution of the fixed-point solution.

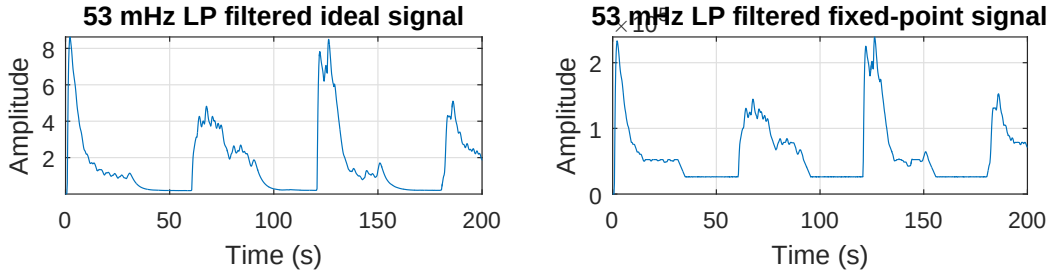


Figure 8.2: Error in filtered result of fixed-point method

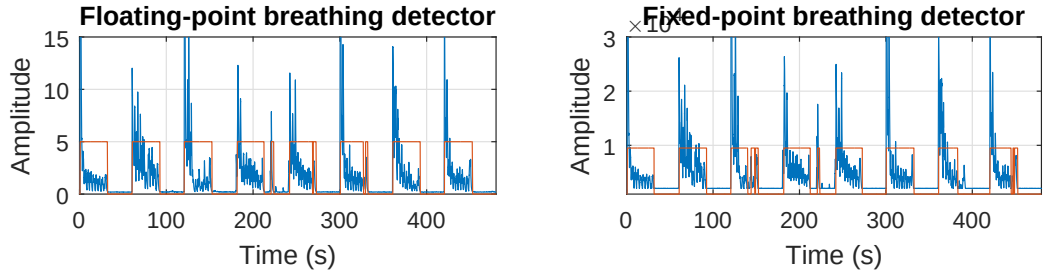


Figure 8.3: Floating-point and fixed-point breathing detector results

Nonetheless, the breathing detection system is still robust enough to identify a breathing signal in the filtered waveforms (see Figure 8.3). Unsurprisingly the accuracy of detection has reduced. This is particularly visible at the 400 s mark where a false negative of the breathing signal occurs. Glitching can also be observed at other time instances. The MSE is now 0.0753 which is over 40% higher than the pre-multiplication fixed-point method used earlier. This solution has however

fulfilled its purpose of utilising the multiplier of the MCU so it has been successful. This is converted into C code and its testing is now performed in the next section.

8.3 Testing

First of all the code written in C needs to be tested in order to verify whether it is behaviourally correct or not. The IAR C-SPY simulator is used for this task. It is a very powerful tool as it simulates the entire MCU's processor at instruction-level. This allows for quick debugging that doesn't require the use of the MCU. Real-time debugging can also be carried out by using the debugger. Communication between the MCU and the computer is established through a fast JTAG or USB interface. This allows information to be sent between the two devices.

Figure 8.4: Testing process used in behavioural verification

The flow for the testing process is presented in Figure 8.4 and is very similar to that of the hardware (FPGA). MATLAB is used to generate a breathing samples file in hex format. This is read into the simulator/debugger which performs the relevant computation on it and returns a value. This value is stored in a text file which is then passed to MATLAB where it is compared to a fixed-point solution. If the error between these two is negligible or zero then the design performs as intended.

8.3.1 Simulation

Breakpoints are inserted in the code, indicating points at which the input file should be read and output file should be written. To automate the reading and writing process, a macro is written. This triggers the reading or writing of a variable from/to a file when the appropriate breakpoint is hit. On the simulator, a disassembly view is present (see Code 8.2) which aids the debugging process. It also shows how the C code roughly maps to lower-level instructions. Viewing results after running the simulation prove that the code behaves as desired. There is zero error between it and the fixed-point solution in MATLAB.

```
1     for(c=N_LP1;c>=0;c--){ // Initialise array to zero
2  004D26   930A             tst.w  R10
3  004D28   380F             jl     0x4D48
```

Code 8.2: Disassembly view in the debugger

8.3.2 Real-time verification

Verification can also be performed on the MCU itself. Breakpoints and a macro for reading and writing input and output variables to the appropriate files are again created. Using the debugger,

the input file from the computer is read into a variable of the MCU. The output variable is then transferred to a text file on the computer. The result from this real-time simulation is as expected. Therefore the code fully works as it should.

8.3.3 Optimisation

There are a number of different types of optimisations that the compiler can perform on the code. They can be aimed at providing a smaller size of code, faster code or a mixture of both. Some of the optimisations include [78]:

- **Loop unrolling** - By unrolling loops, the need for checking bounds is removed. This way the loop overhead is reduced and the code runs faster. However this does increase the size of the code.
- **Function inlining** - When a function is required to be executed, instead of branching to it, the whole body of the function is inserted there. This reduces the overhead of the call. Again this results in faster but longer code.
- **Code motion** - Expressions that do not change are moved so that they are not re-evaluated again. This simple technique is beneficial for both speed and size.
- **Type based alias analysis** - Pointers are known as aliases of each other when they access the same memory. By detecting these, the compiler can perform optimisations to the code to make it more efficient and smaller.

Different levels of optimisation are applied and the clock cycles to execute the code once is recorded (averaged over multiple trials). The memory usage is also noted. These are carried out for the new filtering method which trades in accuracy for less bit-usage and the old filtering method which employs a 64×64 simulated multiplier. This benchmarking is summarised in Tables 8.4 and 8.5. Unsurprisingly as the optimisation level increases the code becomes faster and faster in both cases. At the highest level, the code runs nearly 1000 clock cycles faster than without any optimisations. The benefit of using the hardware multiplier is apparent. On the highest optimisation level, using a hardware multiplier reduces the cycles by a factor of three.

The memory usage displays an interesting pattern. It decreases from no optimisation up to level 2 optimisation for the old filtering method and level 1 for the new method. Beyond this it increases. This may be due to the compiler performing aggressive unrolling and inlining in order to reduce the clock cycles as much as possible. This code is still small enough to fit on the FRAM which has a size of 64 kB. Therefore optimisation level 3 with the new filtering method will be selected for the final MCU solution.

	Opt 0	**Opt 1**	**Opt 2**	**Opt 3**
Clock cycles	4534	4458	3572	3254
Memory usage (Bytes)	3508	3402	2820	3356

Table 8.4: Clock cycles and memory usage for the old filtering method

	Opt 0	Opt 1	Opt 2	Opt 3
Clock cycles	1689	1576	1280	1046
Memory usage (Bytes)	2900	1788	2516	2784

Table 8.5: Clock cycles and memory usage for the new filtering method

8.3.4 Timing analysis

When this system is used in real-time, a signal will appear on the input of the MCU at a frequency of around 2205 Hz. This means that the MCU must finish all its computation at a rate that is faster than the arrival of the input. On the maximum optimisation level it takes 1046 clock cycles to perform the breathing detection algorithm. Using a clock frequency of 16 MHz, this equates to 0.07 ms. The signal on the other hand arrives at every 0.45 ms. This leaves a margin of around 0.38 ms. If the signal arrives at a faster rate of 3000 Hz then the MCU has 0.26 ms before it must perform the breathing detection process again. Thankfully in either case, there is sufficient time for the MCU to perform its task without missing any input samples.

8.4 Power consumption

The on-board debugger has energy tracing capabilities. By measuring the time density of the DC-DC converter charge pulses, power consumption of the MCU can be found. Conventionally power is measured by recording a voltage drop over a shunt resistor. Using this method it isn't possible to detect very short changes in power consumption. Whereas the DC-DC converter's continuous sampling means that the smallest of activity which consumes power on the MCU is detected. All this power information is available live through the device's EnergyTrace technology.

Active power consumption of the micro-controller will vary linearly with its clock frequency and its voltage. Running more peripherals on the MCU will also lead to a higher current draw. While not in active mode, the device can be put into a low power state. They generally remove power and clock to blocks that are not being used. The different states are summarised in the following list:

- **LPM0** - MCLK (master clock) is disabled and so is the CPU.
- **LPM1** - Its mostly the same as LPM0 except the digitally controlled oscillator (DCO) generator may be turned off if no peripheral is using it.
- **LPM2** - The DCO and SMCLK (sub-system clock) are turned off to save power. Only the low frequency ACLK (auxiliary clock) remains on.
- **LPM3** - Only the ACLK is turned on. The rest of the clocks and clock generators are turned off.
- **LPM4** - This turns all the clocks off. Therefore no internal peripherals can be used. Only externally generated interrupts can wake the CPU up. As a result this mode uses the least amount of power. Also the RAM is put into retention mode.

To maximise power efficiency, the MCU should be kept in these low power states for as long as possible. It should only be in active mode when a new input sample arrives and should return back to the sleep state as soon as it finishes any computation required. The ratio of time spent

in active and low power state will determine the MCU's average power consumption. Taking the input frequency of the samples to be 3000 Hz, the power can be expressed by the following formula:

$$Power_{average} = 0.21 \times Power_{active} + 0.79 \times Power_{sleep} \tag{8.1}$$

The last two modes draw the least amount of current. Therefore they are further investigated in the subsections below.

8.4.1 LPM3

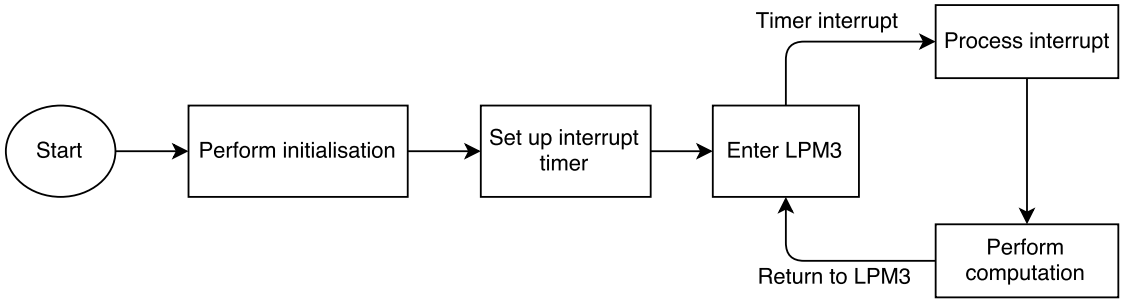


Figure 8.5: LPM3 code flow

In this mode, only the ACLK is available for use while the MCU is sleeping. The ACLK can be set to a low frequency of 32 kHz to minimise power consumption. It may be utilised to simulate an interrupt by setting a timer that triggers an interrupt every 0.33 ms. An up-counter is used for the timer and is set to a value of eleven for a frequency close to 3000 Hz. The flow for this whole process is shown in Figure 8.5. In the interrupt service routine (ISR), the input signal is processed. Here the MCU is in an active state and a 16 MHz clock is available. After completing the task, the device returns to sleep mode.

8.4.2 LPM4

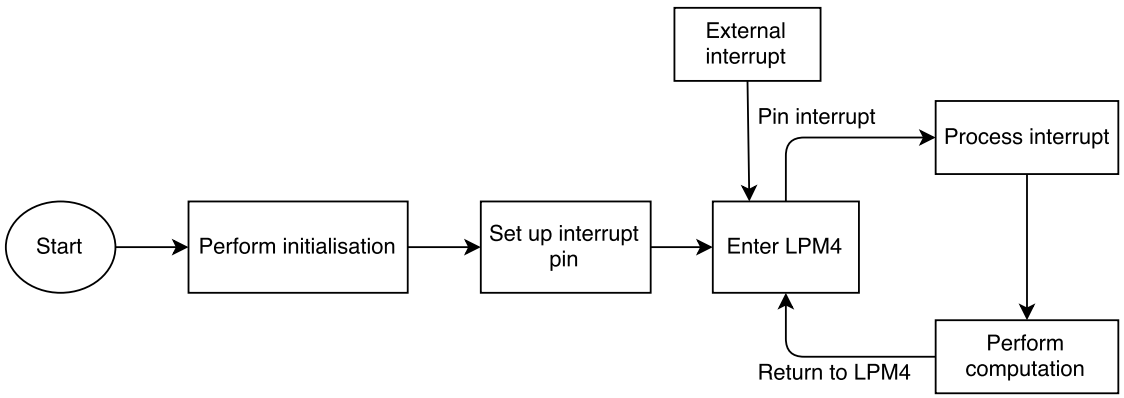


Figure 8.6: LPM4 code flow

LPM4 is an extremely energy efficient mode where all the clocks of the MCU are gated. Moreover the RAM is also put into retention to save additional power. No peripherals can be used in this state due to the lack of a clock. Therefore an interrupt must be provided externally. Referring

to the code flow in Figure 8.6, the process is very similar to that of the LPM3 mode. The major difference being the setting up of a pin for an interrupt instead of a counter.

8.4.3 Measured power

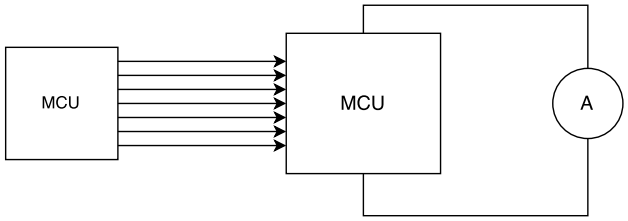


Figure 8.7: Measuring the MCU's power consumption

To measure the power consumption on the MSP430FR5969, the input pins of the device are driven with breathing samples supplied by another MCU. The interrupt is also generated by the secondary chip. Current is measured across the target platform and it is multiplied by the core voltage (3.6 V) to find the power.

When leaving the MCU under active mode, even when it is idle, it consumes a large power of 9.56 mW. Under the LPM3 scenario a mean power of 3.24 mW is recorded from the MCU. This power estimate is likely to be in-accurate since in reality the interrupts will be provided externally. Therefore there will be energy dissipated in I/O pins. The 32 kHz clock and the timer peripheral on the other hand draw unnecessary power while the MCU is sleeping. To further improve the accuracy of the estimated power consumption, the LPM4 mode is programmed. With a measured power of 3.02 mW, LPM4 is indeed the most energy efficient mode. This measure also accurately depicts the expected power consumption in the devices intended use case scenario. However the power saving isn't as large as would be expected. This is because in the deepest sleep mode it takes the MCU a certain amount of time to wake up. That is to start all the necessary clocks needed. Thus some energy is wasted in the starting up process.

8.4.4 Summary

This investigation has first of all highlighted the importance of a hardware multiplier. Clock cycles are the enemy of power efficiency and by using a dedicated multiplier they can be minimised. A three times boost is achieved over a simulated multiply routine. Furthermore it is found that power can be conserved by placing the MCU into sleep mode when it is idle. Putting the MCU into the lowest power state, a power reduction of over three times is achievable for the breathing detector. All the measured powers under different modes are shown in Table 8.6.

Active	**LPM3**	**LPM4**
9.56 mW	3.24 mW	3.02 mW

Table 8.6: Power consumption under different power modes

Chapter 9

Power Source

A mobile breathing detector will need to be powered by a battery. There is a trade-off here between the battery's capacity and its size/weight. A large capacity will be able to last a longer time, reducing the inconvenience of having to replace or recharge the cells. However a battery holding more power will also weigh more or be of a larger dimension. Thus the inconvenience caused by replacing the battery will have to be balanced against the discomfort caused by carrying a bigger battery with more charge. In this section the different types of batteries available are explored and the most optimal type is chosen for the embedded systems.

9.1 Battery types

The ideal power source for an electronic device varies. Manufactures of cells cater for this diverse market by producing a large range of different types of batteries. These have characteristics that are suited to a particular application. Details on the different battery chemistries widely available are listed below:

- **Nickel Cadmium (NiCd)** - Has a flat discharge curve so the voltage stays near the nominal value. It is a low cost solution and has a large number of charging and discharging cycles. On the downside it does suffer from the memory effect where a periodic discharge is required to prevent performance loss. Also they contain toxic metals which make them unfriendly to the environment.
- **Nickel-Metal Hydride (NiMH)** - In contrast to the NiCd type, they contain no toxic metals. They have a high energy density so are suitable for high power applications. However their life-span is shorter. They have fewer cycles than the NiCd battery.
- **Zinc Carbon** - One of the cheapest types of battery. They are not suitable for devices that require high power, their energy density is low. Also leakage is relatively high compared to other forms.
- **Lithium ion (Li-ion)** - It provides a high energy density as well as a flat discharge curve. They can be lightweight and come in small sizes. Moreover low maintenance is required. Drawbacks include their fragility and high-costs.
- **Lead acid** - Provide high power at low cost. Not suitable for devices where weight and size are a concern.

From the size and weight constraints imposed by embedded systems, the choice of chemistry can be narrowed down. The lead acid battery can immediately be excluded due to its large size. The low density combined with the high leakage make zinc batteries unsuitable for longevity. Although NiCd offers a flat discharge curve, its memory effect can be a let down. Therefore only Li-ion and NiMH battery types will now be considered for the platforms.

9.2 FPGA

There is a choice of running the FPGA core at either 1.2 or 1.5 volts. Since power consumption is dependant on the square of voltage, the lower one is chosen. This decision does lower the maximum input clock frequency to 160 MHz [59]. A frequency of only 57 kHz is required so this reduction is irrelevant for the project. The NiMH type is found to dominate 1.2 V batteries. An example of two suitable cells for the FPGA are displayed in Figures 9.1 and 9.2.

Figure 9.1: 300 mAh NiMH battery [79]

Figure 9.2: 240 mAh NiMH battery [80]

Chemistry	Form	Size (D×H)	Weight	Capacity	Price	Economy
NiMH	Cylinder	13.9×16.4 mm	8 g	300 mAh	£0.80	0.002 £/mAh
NiMH	Coin	25.1×6.7 mm	10 g	240 mAh	£2.36	0.010 £/mAh

Table 9.1: Comparing the specifications of different NiMH batteries

A table is also constructed which compares the major features of the two cells (Table 9.1). Note they are both recharageable. These are taken from their respective datasheets [79][80]. The cylinder form of the cell not only offers the highest capacity, it is also the cheapest. When a lower height of battery is desired, the coin shaped cell offers a better alternative. It is however more expensive and thus has a lower economy (five times less). Taking the cylinder form battery, its expected life expectancy is calculated. The FPGA will draw a current of 0.07 mW (under typical conditions) then the battery is expected to last 5142 hours. This amounts to a massive 214 days before it needs to be recharged.

9.3 MCU

Two operating voltages are supported by the MCU. These are 1.8 V and 3.6 V. To run the MCU at 16 MHz, a voltage of 3.6 V is needed [60]. A battery that runs at this voltage needs to be found. This time round cells are based mostly on the Li-ion technology. In Table 9.2, two Li-ion batteries are compared, they are both rechargeable . The cylinder version presents a better economy rate.

The coin battery is quite expensive. For the highest capacity battery (1000 mAh), its lifetime is calculated. The MCU will consume 3.02 mW (typical condition) at 3.6 V which results in a complete discharge time of 1192 hours, about 50 days.

Figure 9.3: 850 mAh Li-ion battery [81]

Figure 9.4: 1000 mAh Li-ion battery [82]

Chemistry	Form	Size (D×H)	Weight	Capacity	Price	Economy
Li-ion	Cylinder	18.2×34.8 mm	21 g	850 mAh	£3.37	0.004 £/mAh
Li-ion	Coin	33×6.2 mm	16 g	1000 mAh	£11.69	0.012 £/mAh

Table 9.2: Comparing the specifications of different Li-ion batteries

9.4 Comparison

A NiMH type battery is suitable for the FPGA. Whilst for the MCU a Li-ion battery is more ideal. The usage of Li-ion battery does mean that the dimensions and weight are going to be bigger as observed. This is because it requires protection circuits to prevent over-charging and dis-charging. Another reason for the larger size is due to the MCU requiring a bigger capacity as it draws more power. As a consequence of these factors, the cells of the MCU have the worst economy. Table 9.3 shows the capacity of the battery needed to achieve a lifetime of 214 days before a recharge. The FPGA needs a fourteen times smaller battery than the MCU which will mean a compactor breathing detection system.

FPGA	MCU
300 mAh	4300 mAh

Table 9.3: Capacity needed for the battery to last 214 days

Chapter 10

Evaluation

10.1 Ease

From experience the development of a MCU solution has been by far the easiest. Writing in C means that an algorithm can quickly be converted to code. Lots of documentation is provided to help in configuring pins, setting up interrupts and programming the MCU. FPGAs require a bit more effort. Designing low-level features can be time consuming and the flow is quite long. Nevertheless a wealth of EDA tools are available to make the designer's life easier. Research has shown that designing an ASIC solution is a very demanding and tedious process. Though there is more control over the design, its time-consuming nature means it is not suitable for prototyping.

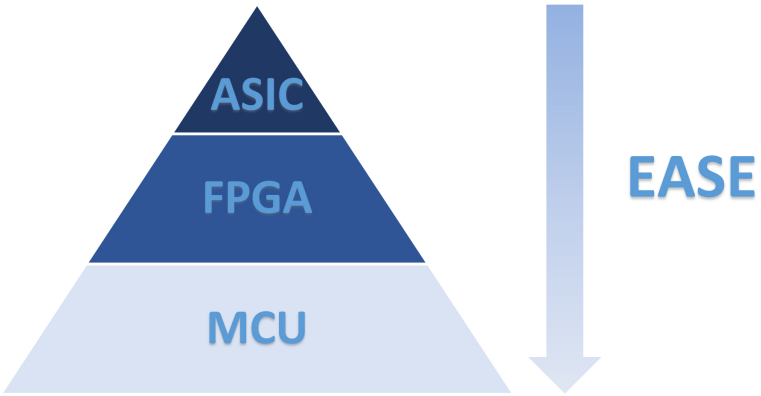


Figure 10.1: Pyramid diagram showing the ease of designing on embedded systems

10.2 Accuracy

Highly accurate solutions are critical in healthcare devices. The breathing detector should be able to provide apnoea diagnosis with very high certainty. For precision critical applications such as this, the floating-point format is the most ideal. However as discussed in this report, this does not suit low power systems. Through careful design, the FPGA has achieved an accuracy that is very similar to the floating-point detector (it is in fact slightly better). The accuracy of the ASIC is limited by the algorithm. Floating-point cores can be placed on an ASIC if needed. Therefore

for this algorithm, it is expected that the ASIC will perform the same as software. On the other hand, the accuracy of the MCU is 40% lower than that of the FPGA. This is because to utilise the integer hardware multiplier, the number of bits used for multiplication have to be reduced.

Software	ASIC	FPGA	MCU
94.27%	94.27%	94.81%	92.47%

Table 10.1: Accuracy of different embedded systems and software

In Table 10.1 the accuracy percentages are displayed. The difference between the MCU and the rest may not seem a lot but in reality it is significant. Continuous breathing monitoring means that a lot of errors can occur over the lifetime of the device. Therefore the accuracy should be ideally 100% or as close to it as feasible. In addition when the algorithm is improved, it may be possible for the FPGAs accuracy to improve as well. However due to less precision, the MCUs detection accuracy may not increase as much or at all. Furthermore when the breathing signal is noisy, or the peak and troughs aren't clear, the MCU solution may deviate a lot from the ideal floating-point version. These are the justifications for using more resources in the FPGA to keep the accuracy as high as possible.

10.3 Performance

Performance wise the ASIC is the most optimal device. It can perform the entire task in a single cycle if necessary. Using specialised components for a particular application means it will have the highest speed over any generic hardware implementation. The FPGA is the next fastest platform. It takes nineteen cycles to provide the breathing detection result. The MCU is the slowest by a large amount. Over 1000 clock cycles are needed to produce a result. The FPGA platform provides about a fifty times boost over the MCU implementation.

ASIC	FPGA	MCU
1	19	1046

Table 10.2: Clock cycles taken for different embedded systems

10.4 Power consumption

For convenience and mobility, power consumption is an essential topic. This will also have a direct impact on the size and weight of the device. Even though the MCU has the lowest standby power of 1.44 μW, it drains the most amount of power (3.02 mW). Putting the MCU in the lowest power state when it is idle, does reduce its power consumption. However it simply doesn't spend enough time in this mode to make up for its high active power. A bigger battery is thus needed to compensate for the quick drainage. This results in more weight and size added to the whole system which is undesired.

The FPGA is found to be a very efficient platform. If the chip can be supplied with a clock frequency of 57 kHz, it will have a very low power usage of 0.07 mW. Note that power dissipated

by the clock isn't included in this measurement, although it is likely to be negligible. In fact the FPGA uses forty-three times less power than the MCU. This means that a smaller, lighter battery can be used in the breathing detection system.

For fairness of comparison, the RTL compiler used for the ASIC has been supplied with the same design as the FPGA. Results from the Cadence RTL compiler using the AMS H18 process indicate that the ASIC is very power efficient. With a total power consumption of just 0.007 mW, it is undoubtedly the best. The complete power breakdown of the ASIC is provided in the Appendix (Section 13.8). It requires ten times less power than the already efficient FPGA and a massive four-hundred times less power than the MCU. Therefore this platform is the lowest maintenance since it lasts the longest. The tables underneath present the power consumption and expected life expectancy of the platforms.

ASIC	**FPGA**	**MCU**
0.007 mW	0.07 mW	3.02 mW

Table 10.3: Power consumption for different embedded systems

ASIC	**FPGA**	**MCU**
2140 days	214 days	15 days

Table 10.4: Expected lifetime on a 300 mAh battery for different embedded systems

10.5 Cost analysis

Expenditure of the hardware processing unit is considered in this section. This does not include the price of the microphone needed to detect the signals. Nor does it contain the costs of manufacturing a body for the system or an alerting mechanism. The MCU chip is the cheapest. However this advantage is lost when considering the price of the battery. Using the economy of the cylinder form of a cell, to match the same life expectancy as an FPGA 300 mAh battery, the cost is estimated to be £17. The total price of this system is therefore £19.

An FPGA chip costs significantly more than the MCU with a price tag of around £14. Despite this high one time fixed cost, the system overall is quite cheap. This is because the running costs of the FPGA are small. A low capacity battery is required which can be obtained for a measly £0.80. Ignoring the costs of an oscillator, the whole system costs less than that of the MCU, at £14.80.

In all other areas the ASIC has excelled, price wise it is the least ideal platform. Huge NRE costs are needed to design and manufacture the chip. There is a lot of expertise and time required which results in high expense. On top of this, due to it being non re-programmable, if a flaw is found in the design at later stages, a lot of time and money can be wasted. Only when economies of scale are put into practice and the chips are mass-produced, the ASIC becomes profitable.

ASIC	FPGA	MCU
£HIGH	£14.80	£19.00

Table 10.5: System costs for different embedded systems

10.6 Summary

The radar chart in Figure 10.2 displays how each platform performs under all the different measures tested in this section. Only in the ease criteria is the MCU better than the other platforms. As expected, the ASIC is very powerful. It is the best choice for performance, power efficiency, accuracy and size. When a quicker and cheaper solution is desired, the FPGA is the best alternative. The radar plot for this device is very balanced. It offers an ease and cheapness comparable to the MCU, while still providing a fully functional low power detection system.

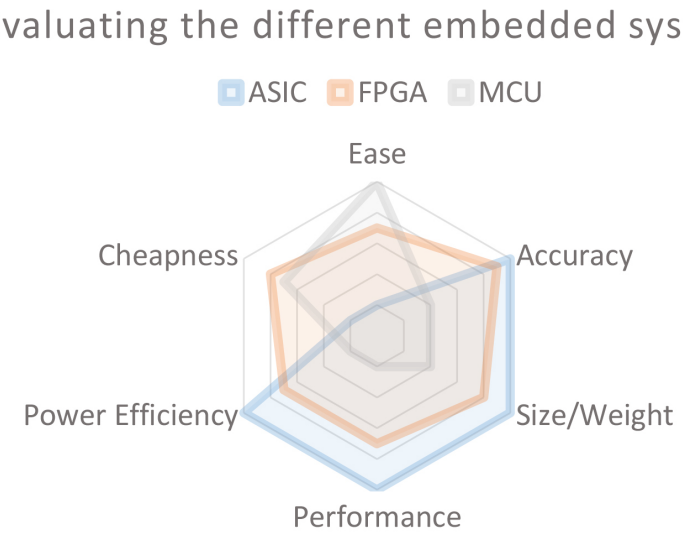


Figure 10.2: Radar chart of all the embedded systems

Chapter 11

Future Work

Perhaps to spend more time in sleep mode, a MCU with a higher clock frequency can be chosen. Computation in the active state would be completed faster as a result. But as dynamic power is dependant on the switching frequency, this may not be beneficial. A MCU with a 32-bit architecture can also be considered. More powerful instructions, a bigger data and control path may equate to faster calculations. The impact on power will need to be investigated.

For the three channel filtering solution, a bigger FPGA is needed. One with more gates can also be considered for the breathing detection system. This is because currently the whole algorithm had to be broken down to fit on the chip. Given an FPGA with more logic, all the detection system could be implemented in one cycle and the clock frequency could drop to just 3000 Hz as a result. Dynamic power would be massively reduced in this case. However it is important not to forget about static power which will increase with the size of the chip. At a size 160% bigger than the current FPGA, the leakage of all transistors can be estimated to cause a power draw of 0.045 mW (0.028×1.6). So no matter what the dynamic power is, the static power is already equal/higher than the current simulated power consumption. Thus for this low power design, static power will dominate therefore the size of the board has to be chosen very carefully.

Another possible way to make the whole system more power friendly is to reduce the number of bits used in calculations. Like the solution for the MCU, optimal quantisation can be carried out for the filter as a whole rather than for each coefficient. Instead of pre-multiplying the signal by sixty-four, a smaller or no multiplication could be examined. The effects on accuracy of the whole solution will have to be investigated thoroughly in this case. As pointed out beforehand, since the intended application of this system is medical, it may not necessarily be a good idea to sacrifice precision and resolution.

The breathing detection algorithm can also be improved. Artefact rejection for swallowing can be incorporated into the system for more accurate results. Thresholds which indicate presence of apnoea are currently fixed by the designer. These can be dynamically set in the algorithm to provide the best results for each patient. Note that since this project is based on an early version of the algorithm, these changes may already have been made or discarded as appropriate.

Chapter 12

Conclusion

One of the main aims of this project was to convert a breathing detection algorithm to an embedded system solution. This has been successfully achieved for two of the three platforms investigated. Unfortunately the CPLD lacked the resources to implement any DSP related solution. It was found to be more suited for non complex glue logic. A high precision solution particularly for the FPGA has been created. This is capable of achieving an impressive 94% apnoea detection accuracy. Its performance is very similar to the double-precision floating-point implementation of the algorithm. Whereas for the MCU a lower resolution of calculation has reduced the accuracy to 92%. In terms of computation speed, the FPGA provided a fifty times boost over the MCU.

Another important aspect of this project was designing low power solutions. Various multipliers were synthesised in hardware. Multiplication forms the basis of filtering used in all deliverables so its efficiency is very important. Radix-4 Booth multiplier was found to be very promising, especially if the quantised coefficients could decompose into a small number of shifts and adds. However the default multiplier whose implementation was chosen by the compiler was found to be the most optimal. Hardware again triumphed over the MCU as the measured power of the FPGA was only 0.07 mW. The MCU consumed over forty times more power.

The three channel filtering solution was unable to fit on the FPGA. This was the case even when all the calculations were carried out sequentially. Results pointed to the need for a bigger chip size or lower precision. A RAM that can hold larger widths may also be beneficial.

Platforms investigated in this project were also compared to the ASIC. The ASIC achieved a power consumption that was ten times lower than the already efficient FPGA. However because its development is very time-consuming and costly, the FPGA is recommended instead for prototyping. The MCU lagged far behind both of these hardware devices in both speed and efficiency.

To conclude, high precision low power algorithm implementation is more than feasible on low-end embedded systems nowadays. As a result, it is possible to create a highly accurate, mobile and lightweight breathing detection system, especially based on the FPGA. This project has also shown how hardware significantly outperforms software/MCU in performance and power efficiency.

Chapter 13

Appendix

13.1 Questionnaires

Epworth sleepiness scale

How likely are you to doze off or fall asleep in the following situations, in contrast to feeling just tired? This refers to your usual way of life in recent times. Even if you have not done some of these things recently try to work out how they would have affected you. Use the following scale to choose the most appropriate number for each situation:

0 = Would never doze or sleep.
1 = Slight chance of dozing or sleeping
2 = Moderate chance of dozing or sleeping
3 = High chance of dozing or sleeping

Situation	Chance of dozing
Sitting and reading	
Watching television	
Sitting inactive in a public place (for example, a theatre or a meeting)	
Being a passenger in a car for an hour without a break	
Lying down to rest in the afternoon when circumstances permit	
Sitting and talking to someone	
Sitting quietly after a lunch without alcohol	
Sitting in a car while stopped for a few minutes in traffic	
Total score	

Scores of 9 or above equate with significant excess daytime sleepiness

Figure 13.1: Epworth sleepiness scale [13]

STOP-Bang Questionnaire

1. ***S***noring: Do you snore loudly (loud enough to be heard through closed doors)?
   Yes No
2. ***T***ired: Do you often feel tired, fatigued, or sleepy during daytime?
   Yes No
3. ***O***bserved: Has anyone observed you stop breathing during your sleep?
   Yes No
4. Blood ***P***ressure: Do you have or are you being treated for high blood pressure?
   Yes No
5. ***B***MI: BMI more than 35 kg/m^2?
   Yes No
6. ***A***ge: Age over 50 years old?
   Yes No
7. ***N***eck circumference: Neck circumference greater than 40 cm?
   Yes No
8. ***G***ender: Male?
   Yes No

High risk of OSA: Yes to 3 or more questions

Low risk of OSA: Yes to less than 3 questions

Chung F et al. *Anesthesiology* 2008;108:812-21.

Figure 13.2: STOPBang questionnaire

13.2 Planning

13.2.1 Strategy of implementation

From Section 4 it is quite clear that the CPLD is much smaller than the FPGA in terms of the number of gates it possess. This indicates that the design for the CPLD will have to minimise the amount of gates used. As the FPGA is a lot larger than the CPLD, this kind of minimisation may not be required. Therefore it seems that two custom designs for each target device, CPLD and FPGA, will have to be created.

However without any prior results available on the gate utilisation of a breathing detection system implemented on these two devices, it's difficult to decide on the modifications needed to be made. Therefore initially a solution that targets both the CPLD and FPGA will be created. If this solution is unsuccessful on either device then it will be customised to make it work on that specific device.

As for the MCU, the lack of FPU when using double-precision may lead to the code taking a lot of cycles to execute. This could result in some input signals being missed. Thus the required timing should strictly be met. The MCU code could be modified to use an integer format at the expense of precision if this scenario occurs. This will need further investigation.

13.2.2 Version control system

A version control system is used to track all the changes made to certain files. This is particularly useful as the changes made can also be reversed when necessary. This system is therefore invaluable for increasing productivity in a project that involves code development. As this project will involve both software and hardware development, the use of a version control system is vital.

There are several different version control systems available on the market. A short analysis is carried out on each of these and then the best system is selected for this project.

Dropbox

Dropbox is a file hosting service that boasts around 400 million users [83]. It offers services such as cloud storage and file synchronisation. Dropbox makes all files available through a web browser, making the users files accessible from anywhere that has an internet connection. Dropbox also offers a desktop application which seamlessly integrates with major operating systems such as Windows and Linux to make file storing and sharing even easier. Users can also share their files for collaboration.

Additional features include version control where Dropbox keeps a snapshot of all previous versions of a file for a period of 30 days. This period can be extended by upgrading to the Dropbox Pro account [83].

As far as price of the service is concerned, Dropbox offers a limited free service which includes 2 GB of storage space and limited version control. The storage can be increased to 1 TB and the version control history can be increased to a year by paying £7.99 per month [83].

Apache Subversion (SVN)

SVN is a software versioning and revision control tool that is available for free. It is a cross platform software which is widely used and popular in industry. In 2007 an independent technology and market research company called Forrester Research concluded SVN as being the "best option for Standalone Software Configuration Management" [84].

The tool offers a wide range of features particularly suited to code development. These include excellent version tracking ability, interactive conflict resolution and collaboration. Additionally there seems to be no limit on the storage [85]. The interface however isn't as simple as the one with Dropbox and requires some learning.

GitHub

GitHub is a web based version control system. It is based on the Git software. Git offers a command line interface whereas GitHub provides a user friendly web and desktop application graphical user interface (GUI). GitHub has strong cross platform support too. It is a growing service which in 2015 had over 21 million collaborators [86].

GitHub offers a service similar to SVN. It is very much suited to software development. As a result, it too has been adopted by various industries [86]. It has rich version control, collaboration and merge tracking abilities. GitHub also follows a decentralised approach. This leads to both added benefit and cost. The advantage compared to SVN is that a user no longer needs access to the master repository all the time. They can enjoy version control on their local copy. However this leads to additional steps in the process. More commands are needed to be understood such as "`commit`" and "`push`" as opposed to just "`push`" in SVN.

The costing of GitHub is similar to that of Dropbox. It offers a free service with limited storage of 1 GB and 0 private repositories. This can be upgraded with various pricing available on the GitHub website to suit the needs of the consumer [86]. GitHub has recently launched a Student Developer Pack which offers a free Micro Account worth $7 per month for students [86].

Summary

	Dropbox	**SVN**	**GitHub**
Price	Free (Up to a limit)	Free	Free (Up to a limit)
Version Control Ability	Limited	Very good	Excellent
Storage	Depends on price plan	Unlimited	Depends on price plan
Interface	Excellent	Good	Excellent
Operating System	Cross-platform	Cross-platform	Cross-platform

Table 13.1: Features of each service summarised

Table 13.1 shows a summary of all the discussed features of each of the version control service. As the aim of the project is based on software/hardware development, the limited version control ability of Dropbox makes it unsuitable for this project. Both SVN and GitHub offer good version control ability. However GitHub thanks to its decentralised approach can be more powerful than

SVN. Also it has an excellent user interface. Therefore GitHub is chosen as the version control service for this project.

13.2.3 Gantt chart

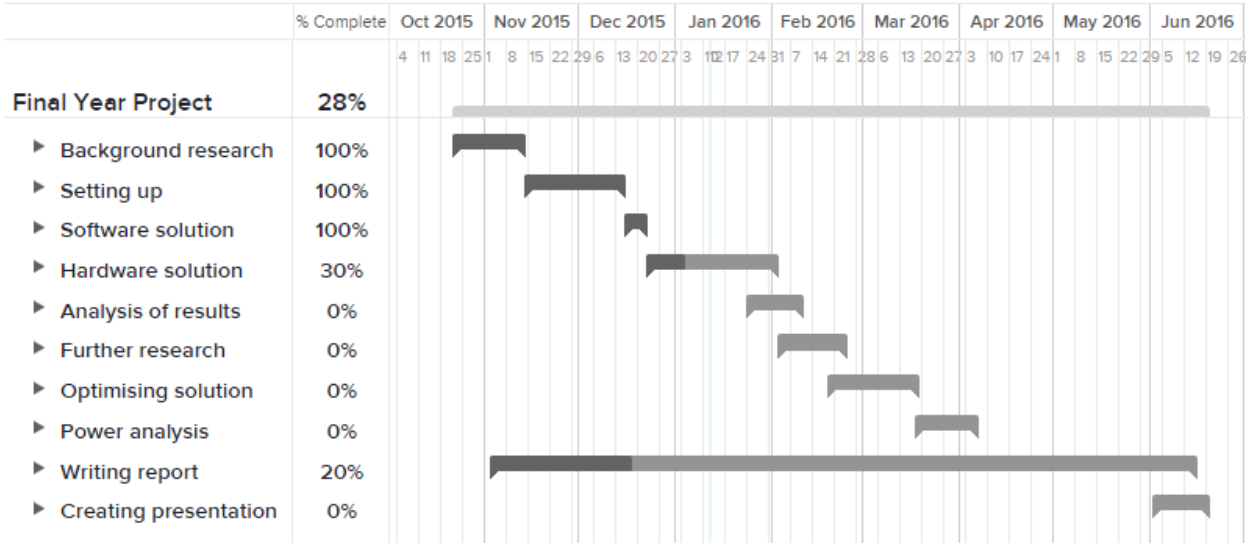


Figure 13.3: Project schedule shown on a Gantt chart

Good organisation and time management are the key to a successful project. This requires allocation of sufficient time to complete all the different tasks. Proper planning helps meeting the deadlines of the many deliverables. One way to achieve this is to use a Gantt chart. A Gantt chart is a visual representation of a project schedule. It highlights the expected start and end dates of each task. It also shows dependencies between the various tasks.

Figure 13.3 above shows the Gantt chart for this project. Each task under the "final year project" headline has been allocated some time. The "writing report" task is an ongoing process throughout this project. The "hardware solution" task has been assigned the next longest time. This is unsurprising as creating and implementing a digital design for both the FPGA and CPLD will take a significant amount of time. The practical tasks are scheduled to be finished around early April. This leaves time for preparation of exams, the writing of the final report and creation of a presentation. It also provides some leeway in case things don't go according to plan.

13.2.4 Risk analysis

As with any project it is wise to analyse the risks associated with the project. Undertaking risk analysis at an early stage prevents the completion of the project being hindered at later stages. The foreseeable risks related to this project have been listed in Table 13.2 alongside their solutions.

Risk	Solution
Not completing the project in time	Gantt chart created for efficient organisation
CPLD being too small for the implementation of the project specification	This risk has been acknowledged and a solution has been proposed
Algorithm doesn't detect apnoea	Reduce fixed-point errors and re-design software and hardware system carefully
MCU too slow in processing data	Reduce precision, make code more efficient & use highest compiler optimisation level

Table 13.2: Risks and their solution

13.3 CPLD architecture

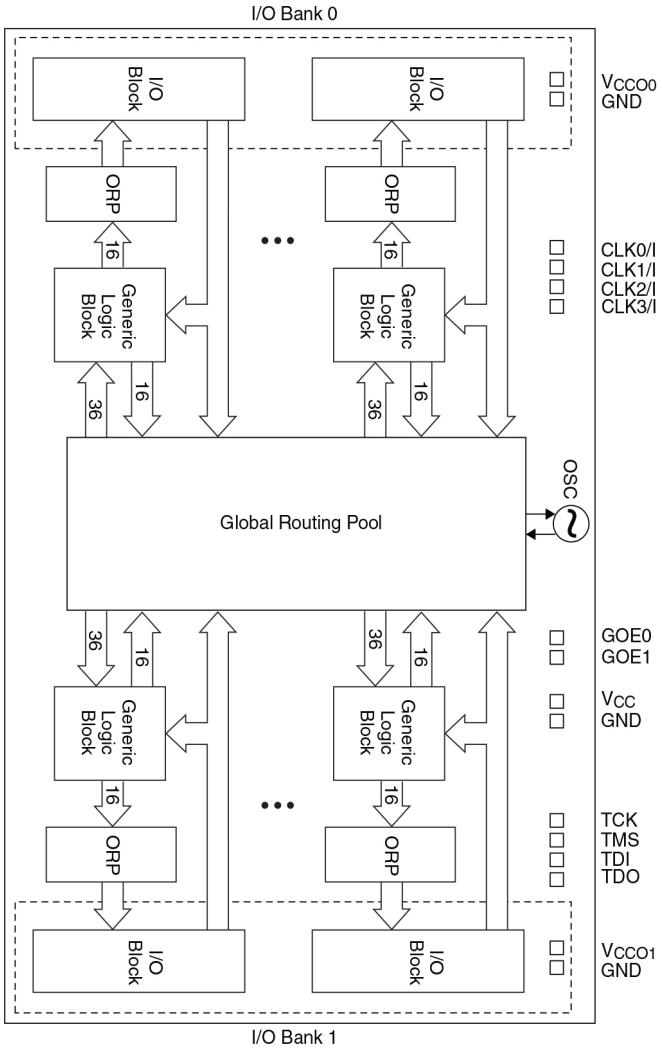


Figure 13.4: CPLD functional block diagram

13.4 FPGA architecture

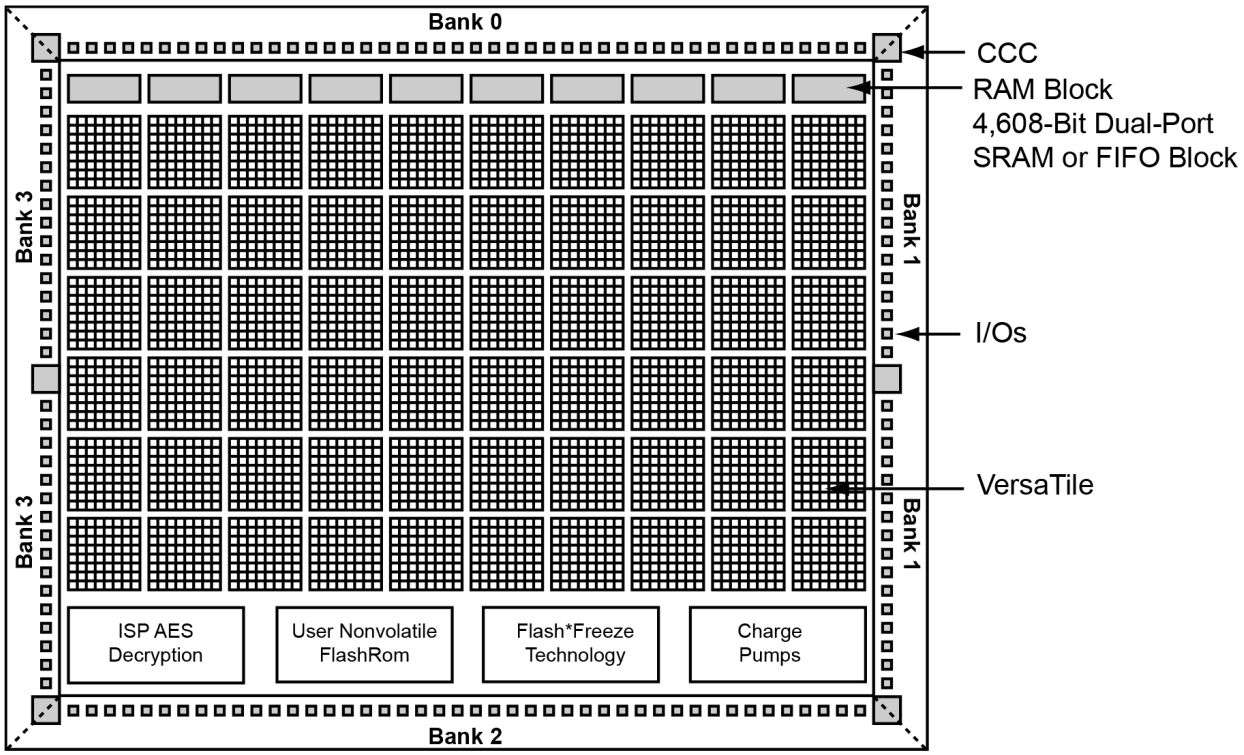


Figure 13.5: FPGA functional block diagram

13.5 MCU architecture

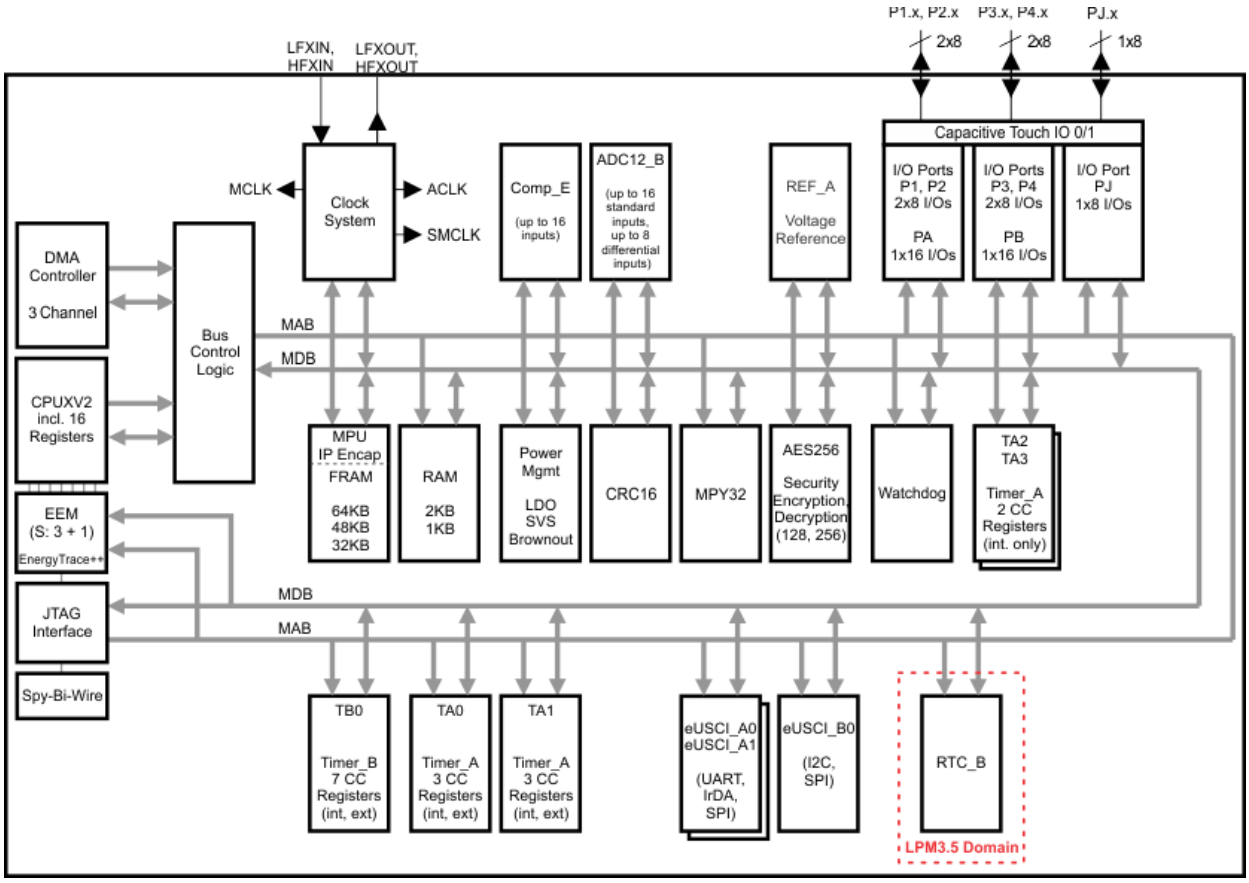


Figure 13.6: MCU functional block diagram

13.6 Synthesis results

```
Report for cell filter.behavioral
  Core Cell usage:
             cell count   area count*area
             AND2    62    1.0      62.0
            AND2A    12    1.0      12.0
             AND3    50    1.0      50.0
              AO1   278    1.0     278.0
             AO13    16    1.0      16.0
             AO18    25    1.0      25.0
             AO1A    22    1.0      22.0
             AO1C    25    1.0      25.0
             AO1D     5    1.0       5.0
             AOI1    10    1.0      10.0
            AOI1A    20    1.0      20.0
            AOI1B     6    1.0       6.0
              AX1     7    1.0       7.0
             AX1B     3    1.0       3.0
             AX1C    25    1.0      25.0
             AX1D    42    1.0      42.0
             AX1E     1    1.0       1.0
             AXO6     1    1.0       1.0
            AXOI3     1    1.0       1.0
           CLKINT     1    0.0       0.0
              GND     1    0.0       0.0
             MAJ3   275    1.0     275.0
             MIN3    14    1.0      14.0
              MX2   830    1.0     830.0
             MX2A    56    1.0      56.0
             MX2B   113    1.0     113.0
             MX2C    62    1.0      62.0
             NOR2    45    1.0      45.0
            NOR2A   592    1.0     592.0
            NOR2B   934    1.0     934.0
             NOR3     8    1.0       8.0
            NOR3A    50    1.0      50.0
            NOR3B    52    1.0      52.0
            NOR3C   135    1.0     135.0
              OA1   167    1.0     167.0
             OA1A    33    1.0      33.0
             OA1B     4    1.0       4.0
             OA1C    14    1.0      14.0
             OAI1     4    1.0       4.0
              OR2   265    1.0     265.0
             OR2A    74    1.0      74.0
             OR2B     5    1.0       5.0
              OR3   166    1.0     166.0
             OR3A     2    1.0       2.0
             OR3B     2    1.0       2.0
             OR3C     1    1.0       1.0
              VCC     1    0.0       0.0
```

```
51                XA1    27      1.0       27.0
52               XA1A     1      1.0        1.0
53               XA1B    12      1.0       12.0
54               XA1C     2      1.0        2.0
55              XAI1A     2      1.0        2.0
56              XNOR2   104      1.0      104.0
57              XNOR3    25      1.0       25.0
58                XO1     8      1.0        8.0
59               XO1A     2      1.0        2.0
60               XOR2   271      1.0      271.0
61               XOR3   252      1.0      252.0
62
63
64               DFN1   419      1.0      419.0
65             DFN1E0    12      1.0       12.0
66                     -----          ----------
67              TOTAL  5654             5651.0
68
69
70    IO Cell usage:
71                cell count
72             CLKBUF     1
73              INBUF    18
74             OUTBUF     1
75                    -----
76              TOTAL    20
77
78
79  Core Cells       : 5651 of 6144 (92%)
80  IO Cells         : 20
81
82    RAM/ROM Usage Summary
83  Block Rams : 0 of 8 (0%)
```

Code 13.1: Core cell usage of the FPGA

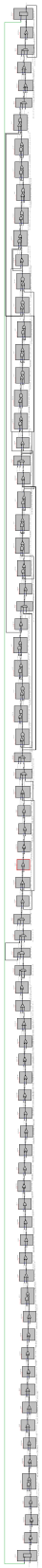


Figure 13.7: The whole critical path of the synthesised breathing detector

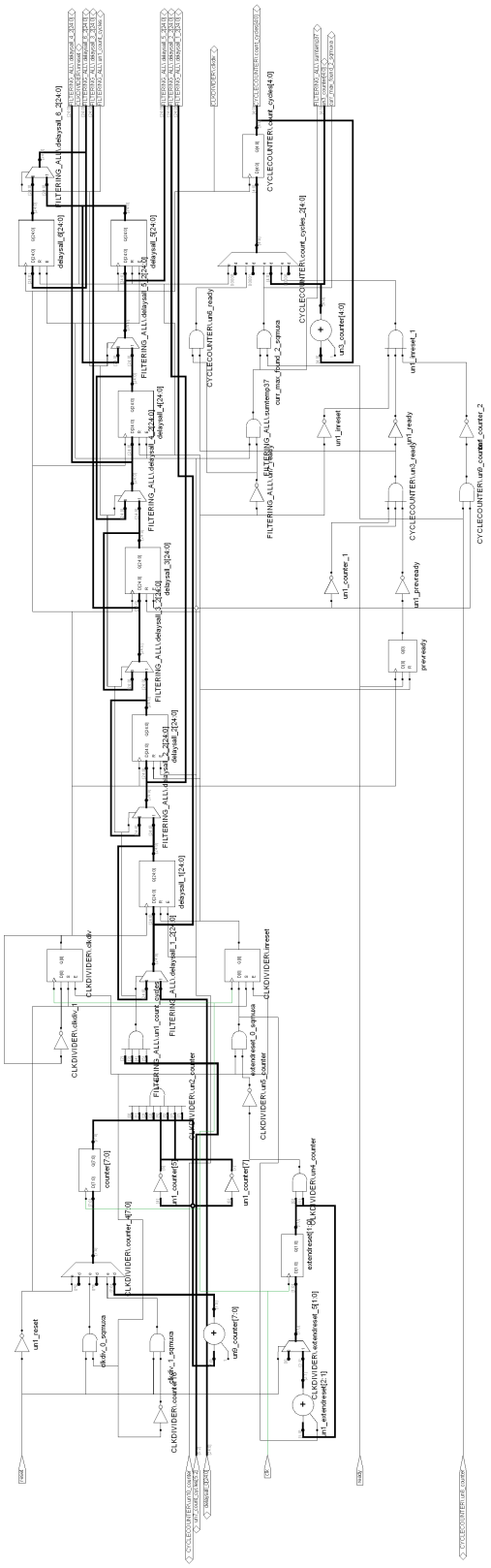


Figure 13.8: Synthesised RTL view of the full breathing detector - Sheet 1

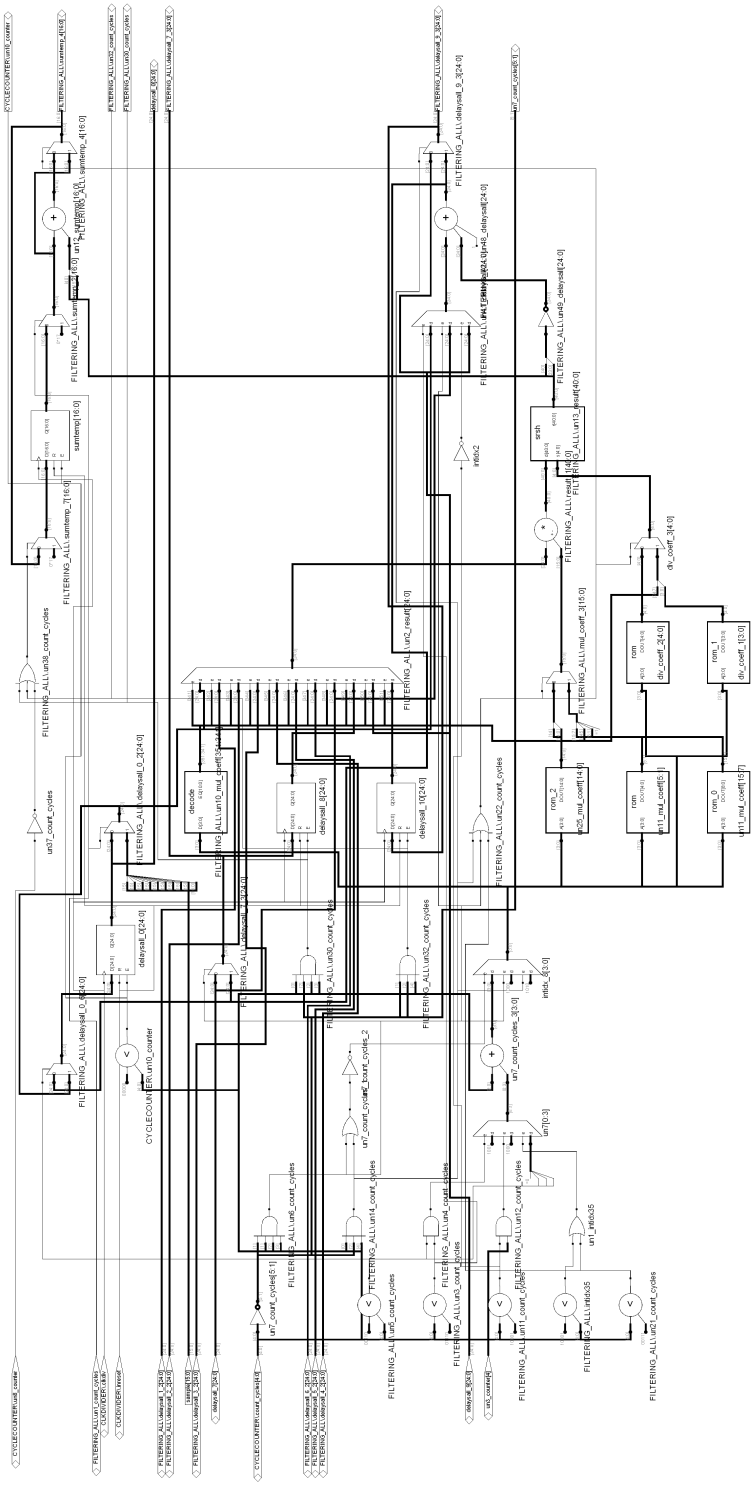


Figure 13.9: Synthesised RTL view of the full breathing detector - Sheet 2

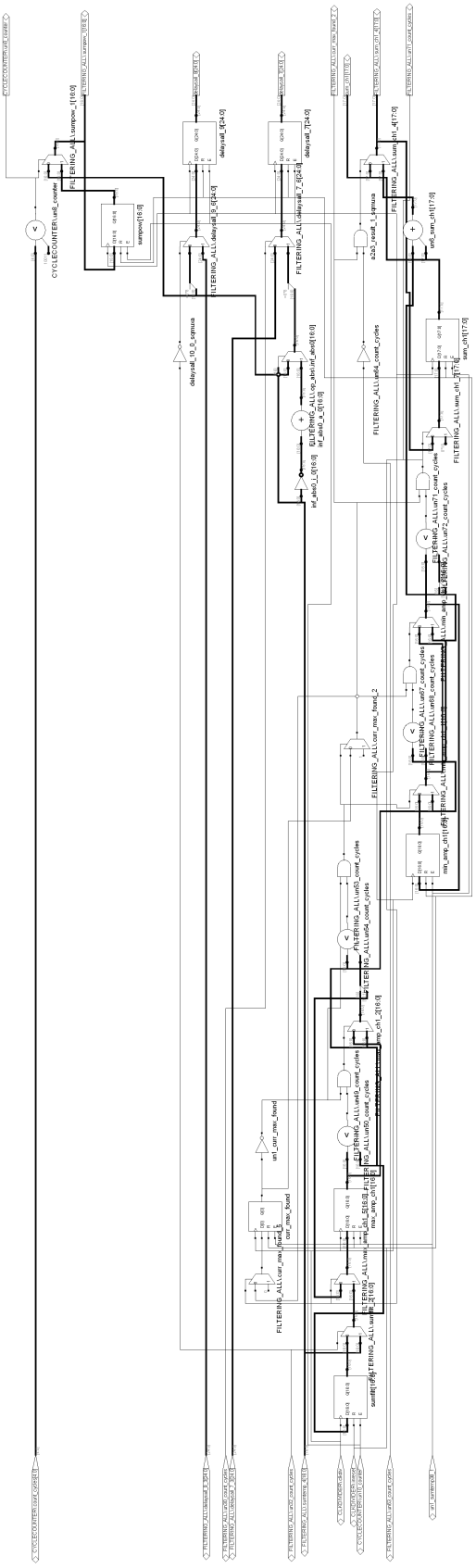


Figure 13.10: Synthesised RTL view of the full breathing detector - Sheet 3

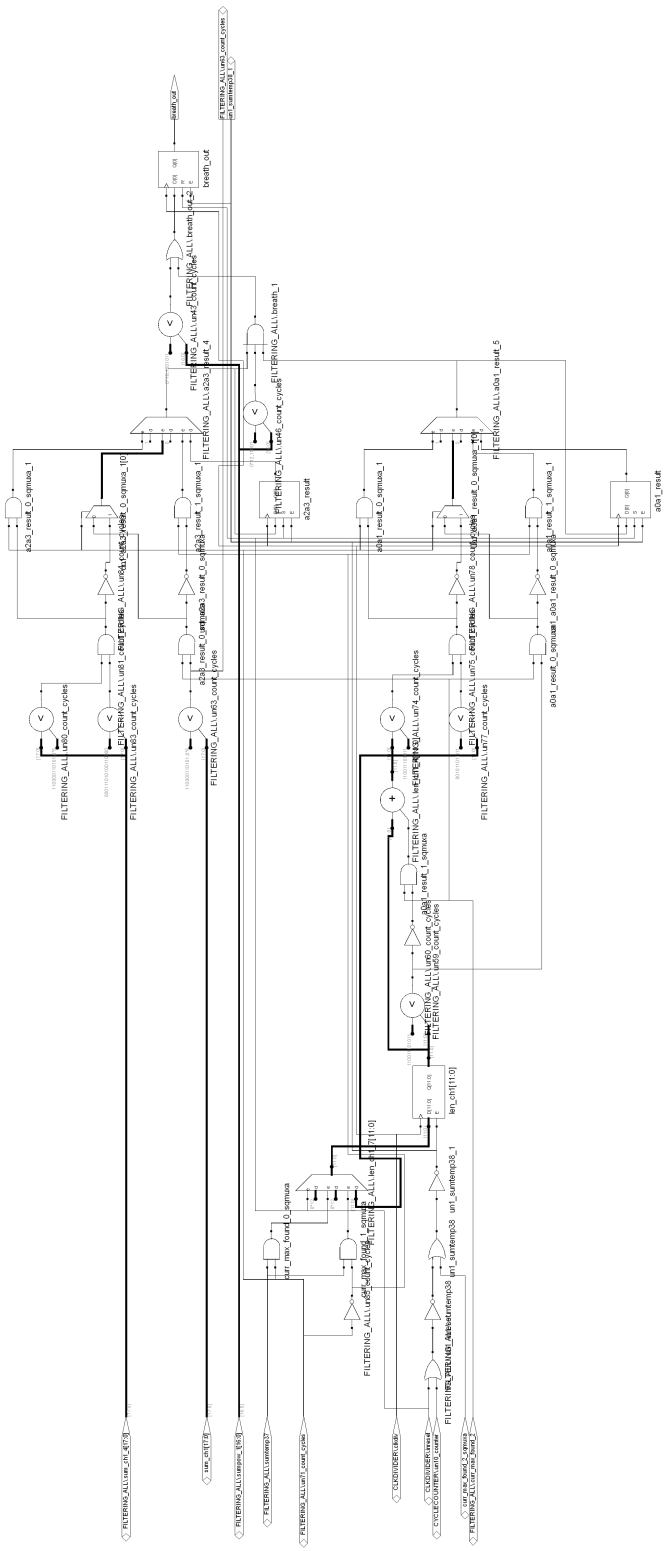
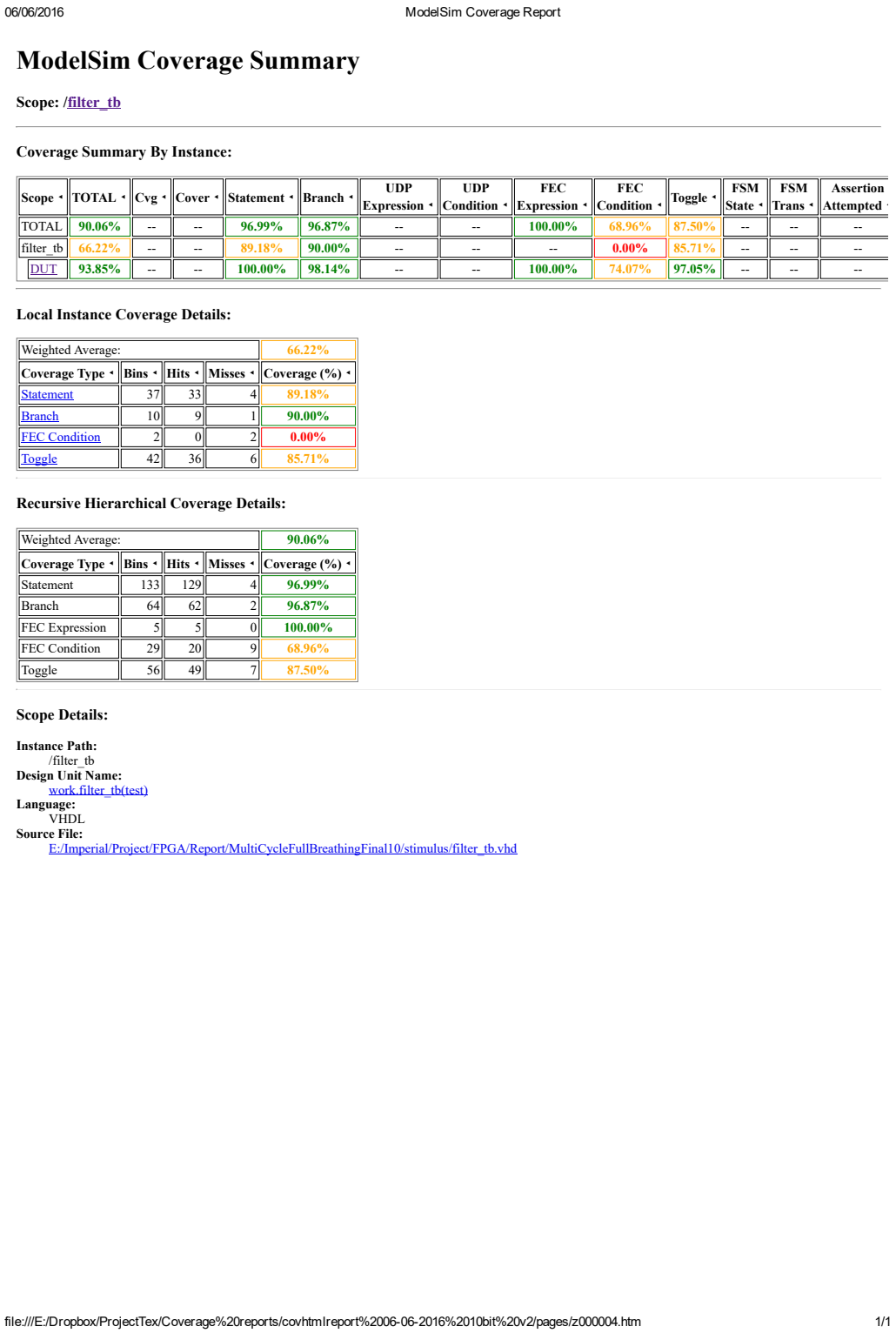


Figure 13.11: Synthesised RTL view of the full breathing detector - Sheet 4

13.7 Coverage report



06/06/2016 ModelSim Coverage Report

ModelSim Coverage Summary

Scope: /filter_tb

Coverage Summary By Instance:

Scope	TOTAL	Cvg	Cover	Statement	Branch	UDP Expression	UDP Condition	FEC Expression	FEC Condition	Toggle	FSM State	FSM Trans	Assertion Attempted
TOTAL	90.06%	--	--	96.99%	96.87%	--	--	100.00%	68.96%	87.50%	--	--	--
filter_tb	66.22%	--	--	89.18%	90.00%	--	--	--	0.00%	85.71%	--	--	--
DUT	93.85%	--	--	100.00%	98.14%	--	--	100.00%	74.07%	97.05%	--	--	--

Local Instance Coverage Details:

Weighted Average:				66.22%
Coverage Type	**Bins**	**Hits**	**Misses**	**Coverage (%)**
Statement	37	33	4	89.18%
Branch	10	9	1	90.00%
FEC Condition	2	0	2	0.00%
Toggle	42	36	6	85.71%

Recursive Hierarchical Coverage Details:

Weighted Average:				90.06%
Coverage Type	**Bins**	**Hits**	**Misses**	**Coverage (%)**
Statement	133	129	4	96.99%
Branch	64	62	2	96.87%
FEC Expression	5	5	0	100.00%
FEC Condition	29	20	9	68.96%
Toggle	56	49	7	87.50%

Scope Details:

Instance Path:
/filter_tb
Design Unit Name:
work.filter_tb(test)
Language:
VHDL
Source File:
E:/Imperial/Project/FPGA/Report/MultiCycleFullBreathingFinal10/stimulus/filter_tb.vhd

file:///E:/Dropbox/ProjectTex/Coverage%20reports/covhtmlreport%2006-06-2016%2010bit%20v2/pages/z000004.htm 1/1

Figure 13.12: Overall coverage report

13.8 ASIC power consumption

```
1  ====================================================================
2    Generated by:         Encounter(R) RTL Compiler RC11.21 - v11.20-s012_1
3    Generated on:         Jun 14 2016 11:45:14 am
4    Module:               filter
5    Technology libraries: h18_CORELIB_HVT_TYP revision 1.0
6                          h18_IOLIB_TYP revision 1.0
7    Operating conditions: typical (balanced_tree)
8    Wireload mode:        enclosed
9    Area mode:            timing library
10 ====================================================================
11
12                         Leakage  Dynamic   Total
13      Instance     Cells Power(nW) Power(nW) Power(nW)
14 --------------------------------------------------------
15 filter             7793   58.571 6460.253 6518.824
16   mul_178_59       3317   26.526    0.000   26.526
17   sra_179_55        457    2.966    0.000    2.966
18   sub_183_73        198    1.306    0.000    1.306
19   add_185_54        162    1.026    0.000    1.026
20   gt_232_18          95    0.804    0.000    0.804
21   gt_263_15          84    0.731    0.000    0.731
22   lt_257_18          88    0.721    0.000    0.721
23   lt_238_18          77    0.650    0.000    0.650
24   abs_195_38         47    0.349    0.000    0.349
25   add_253_54         55    0.311    0.000    0.311
26   inc_add_247_54_1   11    0.169    0.000    0.169
27   gt_273_36          21    0.143    0.000    0.143
28   lt_273_19          10    0.101    0.000    0.101
29   gt_219_17           6    0.101    0.000    0.101
30   sub_160_51         12    0.095    0.000    0.095
31   gt_225_17           7    0.094    0.000    0.094
32   gt_250_29           9    0.090    0.000    0.090
33   lt_149_52           7    0.085    0.000    0.085
34   sub_140_51         12    0.084    0.000    0.084
35   lt_267_19          12    0.073    0.000    0.073
36   gt_267_36           7    0.067    0.000    0.067
37   inc_add_316_51_2    4    0.058    0.000    0.058
38   gt_139_25           4    0.046    0.000    0.046
39   gt_319_19           2    0.038    0.000    0.038
40   lt_139_52           5    0.036    0.000    0.036
41   lt_165_22           5    0.033    0.000    0.033
42   lte_136_22          5    0.031    0.000    0.031
43   gt_159_25           5    0.026    0.000    0.026
44   gt_244_29           4    0.021    0.000    0.021
45   lt_319_41           2    0.018    0.000    0.018
46   gt_149_25           4    0.016    0.000    0.016
47   CLK_BLOCK           0    0.000    0.000    0.000
```

Code 13.2: ASIC power consumption

Bibliography

[1] NHS. *Obstructive sleep apnoea.* `http://www.nhs.uk/Conditions/Sleep-apnoea/Pages/Introduction.aspx`, Accessed 2016.

[2] Corbishley P, Rodríguez-Villegas E. *Breathing detection: towards a miniaturized, wearable, battery-operated monitoring system.* Biomedical Engineering, IEEE Transactions on. 2008;55(1):196-204.

[3] intel. *50 Years of Moore's Law.* `http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html`, Accessed 2016.

[4] C Guilleminault. *Clinical features and evaluation of obstructive sleep apnea.* Principles and practice of sleep medicine. 1989;8:552.

[5] H Gastaut, CA Tassinari, B Duron. *Polygraphic study of the episodic diurnal and nocturnal (hypnic and respiratory) manifestations of the Pickwick syndrome.* Brain Res. 1996;(1):167–186.

[6] G. J Gibson. *Obstructive sleep apnoea syndrome: underestimated and undertreated.* British Medical Bulletin. 2004;(72):49–64.

[7] D Schlosshan, M W Elliott. *Clinical presentation and diagnosis of the obstructive sleep apnoea hypopnoea syndrome.* Thorax. 2004;(59):347–352.

[8] Terán-Santos J, Jiménez-Gómez A, Cordero-Guevara J. *The association between sleep apnea and the risk of traffic accidents.* N Engl J Med. 1999;340(11):847-851.

[9] Ludka Ondrej, Konecny Tomas, Somers Virend. *Sleep Apnea, Cardiac Arrhythmias, and Sudden Death.* Tex Heart Inst J. 2011;38(4):340–343.

[10] E R Behr, A Casey, M Sheppard, M Wright, T J Bowker, M J Davies, W J McKenna, and D A Wood. *Sudden arrhythmic death syndrome: a national survey of sudden unexplained cardiac death.* Heart. 2007;93(5):601–605.

[11] Jean-Louis G, Zizi F, Clark LT, Brown CD, McFarlane SI. *Obstructive sleep apnea and cardiovascular disease: role of the metabolic syndrome and its components.* J Clin Sleep Med 2008;4(3):261–272.

[12] Monahan K, Redline S.*Role of obstructive sleep apnea in cardiovascular disease.* Current opinion in cardiology. 2011;26(6):541.

[13] Sophie D West, Helen A McBeath, John R Stradling. *Obstructive sleep apnoea in adults.* BMJ. 2009.

[14] Wang H, Parker JD, Newton GE, Floras JS, Mak S, Chiu KL. Ruttanaumpawan P, Tomlinson G, Bradley TD: *Influence of obstructive sleep apnea on mortality in patients with heart failure.* J Am Coll Cardiol. 2007;49(15):1625-1631.

[15] National Heart, Lung, and Blood Institute. *Who Is at Risk for Sleep Apnea?.* `http://www.nhlbi.nih.gov/health/health-topics/topics/sleepapnea/atrisk`, Accessed 2016.

[16] Lucia Spicuzza, Daniela Caruso, and Giuseppe Di Maria. *Obstructive sleep apnoea syndrome and its management.* Therapeutic Advances in Chronic Disease. 2015;6(5):273–285.

[17] Ohayon MM, Guilleminault C, Priest RG, Caulet M. *Snoring and breathing pauses during sleep: telephone interview survey of a United Kingdom population sample.* BMJ. 1997;314(7084):860-863.

[18] Kushida, Clete A., et al. *Practice parameters for the indications for polysomnography and related procedures: an update for 2005.* Sleep. 2005;28(4):499-521.

[19] Harvard Medical School. *Testing.* `http://healthysleep.med.harvard.edu/sleep-apnea/diagnosing-osa/testing`, Accessed 2016.

[20] American Academy of Sleep Medicine. *Obstructive Sleep Apnea.* AASM. 2008.

[21] Harvard Medical School. *Understanding the Results.* `http://healthysleep.med.harvard.edu/sleep-apnea/diagnosing-osa/understanding-results`, Accessed 2016.

[22] Redline S, Budhiraja R, Kapur V, Marcus CL, Mateika JH, Mehra R, Parthasarthy S, Somers VK, Strohl KP, Sulit LG, Gozal D. The scoring of respiratory events in sleep: reliability and validity. Journal of clinical sleep medicine: JCSM: official publication of the American Academy of Sleep Medicine. 2007;3(2):169-200.

[23] Pierre ESCOURROU SL, Marlène REHEL HN. *Needs and costs of sleep monitoring.* European Neurological Network: ENN. 2000;78:69.

[24] FLORENCE PORTIER, ADRIANA PORTMANN, PIERRE CZERNICHOW, et al. *Evaluation of Home versus Laboratory Polysomnography in the Diagnosis of Sleep Apnea Syndrome.* Am J Respir Crit Care Med. 2000;162:814-818.

[25] Silva GE; Vana KD; Goodwin JL; Sherrill DL; Quan SF. *Identification of patients with sleep disordered breathing: comparing the Four-Variable screening tool, STOP, STOPBang, and Epworth Sleepiness Scales.* J Clin Sleep Med. 2011;7(5):467-472.

[26] DUMITRACHE-RUJINSKI S, CALCAIANU G, ZAHARIA D, TOMA CL, BOGDAN M. *The Role of Overnight Pulse-Oximetry in Recognition of Obstructive Sleep Apnea Syndrome in Morbidly Obese and Non Obese Patients.* Mædica. 2013;8(3):237-242.

[27] MediSupplies. *Finger Pulse Oximeter MD300C11.* `https://www.medisupplies.co.uk/Diagnostic-Equipment/Pulse-Oximetery/Finger-Pulse-Oximeter-MD300C11`, Accessed 2016.

[28] Chambrin MC. *Alarms in the intensive care unit: how can the number of false alarms be reduced?.* Crit Care. 2001;5(4):184-188.

[29] J A BENNETT, W J M KINNEAR. *Sleep on the cheap: the role of overnight oximetry in the diagnosis of sleep apnoea hypopnoea syndrome.* Thorax, 1999;54:958-959.

[30] Khalil MM, Rifaie OA. *Electrocardiographic changes in obstructive sleep apnoea syndrome.* Respiratory medicine. 1998;92(1):25-27.

[31] Coccagna G, Mantovani M, Brignani F, Parchi C, Lugaresi E. *Continuous recording of the pulmonary and systemic arterial pressure during sleep in syndromes of hypersomnia with periodic breathing.* Bulletin de physio-pathologie respiratoire. 1971;8(5):1159-1172.

[32] Almazaydeh L, Elleithy K, Faezipour M. *Obstructive sleep apnea detection using SVM-based classification of ECG signal features.* InEngineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE. 2012;4938-4941.

[33] Khandoker AH, Palaniswami M, Karmakar CK. *Support vector machines for automated recognition of obstructive sleep apnea syndrome from ECG recordings.* Information Technology in Biomedicine, IEEE Transactions on. 2009;13(1):37-48.

[34] Keerthi SS, Chapelle O, DeCoste D. *Building support vector machines with reduced classifier complexity.* The Journal of Machine Learning Research. 2006;7:1493-515.

[35] NHS. *Sudden infant death syndrome (SIDS).* `http://www.nhs.uk/Conditions/Sudden-infant-death-syndrome/Pages/Introduction.aspx`, Accessed 2016.

[36] Baby Monitors Direct. *Nanny Baby Breathing Monitor Extra Sensor Pad.* `http://www.babymonitorsdirect.co.uk/nanny-baby-breathing-monitor-extra-sensor-pad.html`, Accessed 2016.

[37] thefreedictionary. *microcontroller.* `http://encyclopedia2.thefreedictionary.com/Microcontroller+unit`, Accessed 2016.

[38] ARM. *Cortex-M7 Processor.* `http://www.arm.com/products/processors/cortex-m/cortex-m7-processor.php`, Accessed 2016.

[39] New York: McGraw-Hill. *Motorola joins microprocessor race with 8-bit entry.* Electronics. 1974;47(5):29-30.

[40] Motorola semiconductor techinal data. *MCU/MPU.* `http://datasheets.chipdb.org/Motorola/mc6801_3.pdf`, Accessed 2016.

[41] Rob Lineback. *Microcontroller Unit Shipments Surge but Falling Prices Sap Sales Growth.* IC Insights. 2015;1-2.

[42] RS Components Ltd. *STMicroelectronics STM32F407VGT6, 32bit ARM Cortex M4F Microcontroller, 168MHz, 1024 kB Flash, 100-Pin LQFP.* `http://uk.rs-online.com/web/p/microcontrollers/7468226/`, Accessed 2016.

[43] Zack Albus, Adrian Valenzuela, Mark Buccini. *Ultra Low Power Comparison: MSP430 vs. Microchip XLP Tech Brief.* White paper. 2009.

[44] Graham P, Nelson B. *Genetic algorithms in software and in hardware-a performance analysis of workstation and custom computing machine implementations.* InFPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on 1996;216-225.

[45] Kuon I, Rose J. *Measuring the gap between FPGAs and ASICs.* Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on. 2007;26(2):203-215.

[46] Sigenics. *Custom ASIC Cost Calculator.* `http://www.sigenics.com/custom-asic-cost-calculator/`, Accessed 2016.

[47] Fey CF, Paraskevopoulos DE. *A techno-economic assessment of application-specific integrated circuits: current status and future trends*. Proceedings of the IEEE. 1987;75(6):829-841.

[48] Triad Semiconductor. *Better Than Full Custom ASIC*. http://www.triadsemi.com/2015/05/12/, Accessed 2016.

[49] Triad Semiconductor. *Reconfigurable Full-Custom ASICs*. https://www.triadsemi.com/reconfigurable-full-custom-asic/, Accessed 2016.

[50] Xilinx. *XILINX VIRTEX-6 FPGAS*. http://www.xilinx.com/support/documentation/selection-guides/virtex6-product-table.pdf, Accessed 2016.

[51] Rose J, Francis RJ, Lewis D, Chow P. *Architecture of field-programmable gate arrays: The effect of logic block functionality on area efficiency*. Solid-State Circuits, IEEE Journal of. 1990;25(5):1217-1225.

[52] Grand View Research. *FPGA (Field-Programmable Gate Array) Market Analysis By Application (Automotive, Consumer Electronics, Data Processing, Industrial, Military And Aerospace, Telecom) And Segment Forecasts To 2020*. 2014.

[53] Xilinx. *What is a CPLD?*. http://www.xilinx.com/cpld/, Accessed 2016.

[54] Lattice Semiconductor. *ispMACH 4256ZE Breakout Board*. http://www.latticesemi.com/Products/DevelopmentBoardsAndKits/ispMACH4256ZEBreakoutBoard.aspx, Accessed 2016.

[55] Digi-Key. *Lattice-Semiconductor-Corporation LC4256ZE-5TN144C*. http://www.digikey.co.uk/product-detail/en/lattice-semiconductor-corporation/LC4256ZE-5TN144C/220-1027-ND/2641828, Accessed 2016.

[56] Lattice Semiconductor. *ispMACH 4000ZE Family*. http://www.latticesemi.com/~/media/LatticeSemi/Documents/Solutions/Packaging%20Solutions/ispMACH4000ZE%20Family%20Data%20Sheet1022.pdf, Accessed 2016.

[57] Microsemi. *IGLOO nano Starter Kit*. http://www.microsemi.com/products/fpga-soc/design-resources/dev-kits/igloo/igloo-nano-starter-kit, Accessed 2016.

[58] Digi-Key. *Microsemi-Corporation AGLN250V2-VQG100I*. http://www.digikey.co.uk/product-detail/en/microsemi-corporation/AGLN250V2-VQG100I/AGLN250V2-VQG100I-ND/2860414, Accessed 2016.

[59] Microsemi. *IGLOO nano Low Power Flash FPGAs*. http://www.microsemi.com/index.php?option=com_docman&task=doc_download&gid=130695, Accessed 2016.

[60] Texas Instruments. *MSP430FR5969 LaunchPad Development Kit*. http://www.ti.com/tool/msp-exp430fr5969, Accessed 2016.

[61] Digi-Key. *Texas-Instruments MSP430FR5969IRGZR*. http://www.digikey.co.uk/product-detail/en/texas-instruments/MSP430FR5969IRGZR/296-37875-2-ND/4878748, Accessed 2016.

[62] ALDEC. *Altera Design Flow*. https://www.aldec.com/images/content/solutions/aldec_altera_flow.png, Accessed 2016.

[63] IEEE spectrum. *The 2015 Top Ten Programming Languages*. http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages, Accessed 2016.

[64] Xilinx. *LogiCORE IP Floating-Point*. http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf, Accessed 2016.

[65] Mentor Graphics. *ModelSim*. https://www.mentor.com/products/fv/modelsim/, Accessed 2016.

[66] Synopsys. *Synplify Pro*. http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/SynplifyPro.aspx, Accessed 2016.

[67] Microsemi. *Libero SoC*. http://www.microsemi.com/products/fpga-soc/design-resources/design-software/libero-soc, Accessed 2016.

[68] TI. *MSP430 Family Instruction Set Summary*. https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf, Accessed 2016.

[69] Bhattacharjee S, Sil S, Basak B, Chakrabarti A. *Evaluation of power efficient adder and multiplier circuits for FPGA based DSP applications*. InCommunication and Industrial Application (ICCIA), 2011 International Conference. IEEE. 2011;1-5

[70] Wilburta Q. Lindh, Marilyn Pooler, Carol Tamparo, Barbara M. Dahl. *Delmar's Comprehensive Medical Assisting: Administrative and Clinical Competencies*. Cengage Learning. 2009;573

[71] Booth AD. *A signed binary multiplication technique*. The Quarterly Journal of Mechanics and Applied Mathematics. 1951;4(2):236-40.

[72] Wallace CS. *A suggestion for a fast multiplier*. Electronic Computers, IEEE Transactions on. 1964:14-7.

[73] Dadda L. *Some schemes for parallel multipliers*. Alta frequenza. 1965;34(5):349-56.

[74] 8x8 bit Booth-Encoded Wallace Tree Multiplier. *4*4 multiplier*. http://www.eecs.tufts.edu/~ryun01/vlsi/verilog_simulation.htm, Accessed 2016.

[75] Hennessy JL, Patterson DA. *Computer architecture: a quantitative approach*. Elsevier; 2011.

[76] TI. *Code Composer Studio (CCS) Integrated Development Environment (IDE)*. http://www.ti.com/tool/ccstudio, Accessed 2016.

[77] IAR Systems. *IAR Embedded Workbench*. https://www.iar.com/iar-embedded-workbench/, Accessed 2016.

[78] IAR Systems. *MSP430 IAR C/C++ Compiler*. http://perso.citi.insa-lyon.fr/afraboul/rts6/doc/EW430_CompilerReference.pdf, Accessed 2016.

[79] AA Portable Power Corp. *NiMH Rechargeable Cell: 1/3 AA 1.2V 300mAh NiMH Rechargeable Batteries*. http://www.batteryspace.com/nimh-rechargeable-cell-1-3-aa-1-2v-280mah-nimh-rechargeable-batteries.aspx, Accessed 2016.

[80] Farnell. *VARTA 55625201068 Rechargeable Battery, Robust, Single Cell, Nickel Metal Hydride, 240 mAh, 1.2 V, Solder Tab*. http://uk.farnell.com/varta/55625201068/battery-nimh-1-v250h-tagged/dp/8636290, Accessed 2016.

[81] BatteryJunction. *Xtar IMR 18350 850mAh 3.7V Protected Lithium Ion (Li-ion) Button Top Battery - Boxed (XTAR-18350-850MAH)*. http://www.batteryjunction.com/xtar-18350-850.html, Accessed 2016.

[82] RS Components Ltd. *Tadiran 1/10 D 3.6V Lithium Thionyl Chloride Coin Button Battery.* http://uk.rs-online.com/web/p/coin-button-batteries/5268560/?searchTerm=Tadiran+1%2F10+D+3.6V, Accessed 2016.

[83] Dropbox. *Dropbox.* https://www.dropbox.com/, Accessed 2016.

[84] Forrester. *Subversion Is The Best Option For Standalone Software Configuration Management.* https://www.forrester.com/Subversion+Is+The+Best+Option+For+Standalone+Software+Configuration+Management/fulltext/-/E-res42334, Accessed 2016.

[85] Apache. *Subversion Best Practices.* https://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html, Accessed 2016.

[86] GitHub. *Where software is built.* https://github.com/, Accessed 2016.