An Integrated System for Live Characterization and Adaptive Compensation of Ultrasonic Transducers

Position Measurement & Control Team

N. D. J. Barker
T. Evers

16 June 2023

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Electrical Engineering Programme

Abstract

Ultrasonic transducers are commonly used in a variety of medical, industrial, and consumer devices, converting electrical energy into high-frequency sound waves. These devices find extensive applications in fields such as medical imaging, non-destructive testing, distance measurement, and cleaning processes. However, these transducers suffer from a narrow operating frequency range caused by a steep frequency response curve with a prominent resonance peak. Existing passive compensation methods using filters are limited due to the individual characteristics of transducers and their susceptibility to process variations, making generic compensation filters impractical. Additionally, the frequency response of transducers changes over time, input power, and environmental conditions, further complicating compensation efforts.

The objective of this project is to overcome these challenges by creating an integrated solution that can provide an ultrasonic transducer system with a consistent frequency response despite external disturbances. The proposed system will incorporate non-linear dynamics characterization and compensation, which are currently lacking in integrated solutions. By accurately characterizing the transducer's non-linear behavior and compensating for it, the system will overcome the drawbacks associated with passive compensation.

The proposed integrated system holds promising implications for various applications, including medical imaging, material testing, and industrial processes. By mitigating the limitations associated with the narrow operating frequency range of ceramic piezoelectric transducers, this research project contributes to the advancement of ultrasonic technology and its broader impact on diverse industries.

Preface

We are pleased to present this thesis, which focuses on the development of an integrated system for ultrasonic transducers with a flat frequency response. Throughout this project, we aimed to overcome the limitations associated with existing systems, particularly in dealing with non-linear effects and external disturbances.

We would like to express our gratitude to Warner Venstra for generously sponsoring this project and providing valuable mentorship throughout. Furthermore we are grateful to Arjan van Genderen for his guidance and assistance throughout the entire process. His expertise and support greatly contributed to the quality of this work. We would also like to thank both Anton Montagne and Michiel Pertijs for their critical insights and feedback, which encouraged us to refine our goals and improve the presentation of our work.

Finally we would like to acknowledge the assistance of Ioan Lager, the Bachelor Project coordinator, and Martin Schumacher from the Tellegen Hall for their support in coordinating the project and sourcing components respectively.

Nicolas Barker & Thomas Evers
Delft, June 2023

Contents

1 Introduction

1.1 Problem Definition

Situation Assessment

Ultrasonic transducers are used in a wide variety of medical, industrial, and consumer devices. Among the various types of transducers for higher frequencies, ceramic piezoelectric transducers are commonly used. A common limitation of these devices is a very narrow operating frequency range, due to a steep frequency response curve with a large peak around resonance. This issue is often addressed by making an appropriate compensation filter, however there are drawbacks to this approach. The first is that the characteristics of individual transducers are susceptible to large process variations, which does not allow for a generic compensation filter to be made, and individually characterizing and compensating each transducer is very costly. The frequency response of transducers also changes over time and input power, and with environmental conditions such as temperature and air pressure. A solution is required that can improve the performance of these devices, regardless of the device's unpredictable circumstances.

Scoping and Bounding Analyses

From this situation assessment it is clear that there is a need for an affordable system that can produce undistorted ultrasound waves. While distortion exists in many forms, in this paper the term distortion references to linear distortion, defined as: a change in amplitude or phase with no new frequencies added. The limitation to linear distortion makes this project feasible in the timeline in which it needs to be delivered, but it limits the use in real life applications. The time limitation for this project was a period of eight weeks, in which the design, creation, and validation must be completed. Due to this limitation, the goal of this project is to make no more than a proof of concept prototype. Furthermore, while most design decisions can be made as direct consequences of the program of requirements, without a specific application in mind some tradeoffs cannot be optimized - assumptions must be made. A significant technical constraint that was set was to limit the compensation to linear distortion (as opposed to non-linear), which is often not the case in real systems. Furthermore, the chosen implementation shall only be capable of producing periodic outputs. Another bound is of course development cost. While the funding available was sufficient to complete the project, cost was still a limitation to consider. A final bound was the determination of useful extra non-linear features used. This task warrants an investigation in itself, so simple non-optimal features will be considered such as temperature. This is further discussed in 3.4.5.

OxiNEMS Affiliation

This project is affiliated with OxiNEMS, an MEG research project that requires a similar system to be built. It may be possible to design the system in such a way that a portion of it will be useful to OxiNEMS, which while making this work more useful, could also limit the flexibility in making choices, often in the form of disregarding optimizations to keep the system generic.

1.2 State of the Art Analysis

The aim of this paper is to design a system that can characterize a system that creates ultrasound live. On top of that it should adaptively control the system based on this characterization.

Currently, ultrasound systems are characterized once and then controlled based on this. Advanced methods[1] of characterization are used that can capture non-linear distortion. The downsides of these systems is that they do not capture any changes in dynamics due to permanent physical changes. This leaves the option open for complete signal distortion without the option of compensation. The system in this paper aims to solve that problem.

1.3 Thesis Synopsis

The thesis will explore previous research on live characterization and compensation of ultrasonic transducers. The focus will be on developing a methodology that accurately characterizes the linear distortion of the transducers and employs an adaptive model to dynamically control the transducer's response. The scalability of the proposed solution to work at high frequencies will also be considered, ensuring its applicability to a wide range of ultrasonic transducers.

The key objective of the project is to design and create a prototype for this system. This prototype will be capable of achieving a flat frequency response for high frequency ultrasonic transducers. The project aims to provide researchers and developers in the field of ultrasonics with an integrated system that offers enhanced control over the sound waves they generate, reducing amplitude distortion and enabling precise manipulation of ultrasound for various applications.

By addressing the current limitations in non-linear characterization and control of ultrasonic transducers, this research will contribute to the advancement of ultrasonic technology and its applications. The outcomes of this thesis will have implications for fields such as medical imaging, non-destructive testing, and industrial applications that rely on accurate and precise ultrasound control.

2 Program of Requirements

The requirements for the system as a whole can be divided into the functional and non-functional requirements. The former specifies the general functions desired in the end product, while the latter specifies the desired technical specifications of the system.

2.1 Functional Requirements

The final product must:

1. Allow the user to input a periodic reference waveform
2. Control the position of an ultrasonic transducer
3. Compensate partially for the linear dynamics of the transducer
4. Compensate partially for the non-linear dynamics of the transducer
5. The production of this system should be scalable, implying the price can be lowered when the system is scaled.

2.2 Non-Functional Requirements

Table 2.1: Table of Non-Functional Requirements

#	**Specification**	**Minimum Req.**	**Trade off Req.**
1	Steady State Ripple of the Flatband	3 dB	Lower
2	Bandwidth of Flatband	50 KHz	Higher
3	Noise Power in Flatband	0.001 Watt	Lower
4	Bandwidth Disturbance Rejection	1 KHz	Higher
5	Time Before System Works	120 Sec	Lower
6	Ultrasound Output Power	1 Watt	Higher
7	Operating Center Frequency	1 MHz	Higher
8	Price Prototype	1500 €	Lower
9	Num Frequency Components Waveform	4	Higher

3 Project Overview

3.1 Producing Ultrasound Waves

In order to create ultrasound an ultrasonic transducer was chosen, the reasoning for which can be found the in the paper [2]. The transducer must then be driven with an amplifier than can deliver the electrical power required to power the transducer. The resulting system will have a non-flat frequency response to input signals. In order to compensate for this, a filter is placed after the input, distorting it in such a way that the final outputted sound closer matches the original input signal. If however the frequency response of the combination of amplifier and transducer changes, the compensation filter will also require adjustment. A logical choice for making one such flexible system is to implement it digitally, so that the functionality can easily be reprogrammed. The last component required is some form of sensor to measure the ground truth behavior of the transducer as well as properties of possible disturbances, such as ambient temperature.

3.2 Measuring the Ultrasonic Transducer

Measuring the vibration of ultrasonic transducers is a demanding problem for any measurement device, given the nanometer-scale displacement and high frequency requirements. Existing methods for measurement include using microphones, hydrophones, and laser interferometers. While many types of microphones exist for this purpose, they are restricted in accuracy due to the air coupling between the transducer and sensor. Aside from the limited resolution of hydrophones due to spatial averaging [3], they also must be used underwater, and could therefore not be implemented non-invasively into an ultrasound device. Laser interferometry is a flexible measurement solution that allows for a high resolution and high frequency displacement measurements. Drawbacks include a relatively high cost and sensitivity to environmental disturbances such as vibrations or temperature fluctuations.

3.3 Compensation for Linear Distortion

In the following section some possible methods of compensation for the system's linear distortion are compared. These include feedback, linear feed-forward, non-linear feed-forward, and adaptive feed-forward.

3.3.1 Feedback

Feedback is a method of controlling a system by measuring the output of the system and adapting the control signal based on that measurement. The advantage is that any non-linear error from the desired ultrasound can directly be compensated for. Many forms of feedback have been used to control audio transducers [4], but few attempts have been made for ultrasound transducers. The reasoning for this is that a feedback system can not be stable, if the controller has a delay of more than half the period of the input frequency. At 1.6 MHz, this is a maximum feedback delay of 312.5 ns. For a well controlled system, this can be at most 100 ns. This includes the entire delay from analog measurement, to digital measurement, to calculating digital control signal, to analog control signal. For a price that meets the non-functional requirement #8, this is an unreasonable specification based on current market research.

3.3.2 Linear Feed-forward

Feed forward is a method of creating a flat frequency response that works by distorting an input signal with the inverse of the system, and then sending that signal into the system.

$$Out_{ref} \cdot |G_{est}|^{-1} \cdot |G| = Out \tag{3.1}$$

$$|G_{est}| \approx |G| \tag{3.2}$$

$$Out_{ref} \approx Out \tag{3.3}$$

This can theoretically create a flat response and is commonly used in the compensation of audio systems and ultrasound systems [5]. An advantage of this method is that it is simple to implement and it only requires the characterisation of the device once, which makes it cheaper to implement. The downside of linear feed forward is that it is not able to compensate for any non-linear dynamics in the system.

3.3.3 Non-Linear Feed-Forward

Non-Linear feed forward is a variation of feed forward in which the system model is non-linear. It is able to compensate for non-linear behavior, but still only to the extent that it is included in the model. When the system exhibits behavior not present in the model, the behavior can not be compensated for, which leads to a distorted output. On top of that, non-linear feed forward is much more resource intensive than regular feed forward.

3.3.4 Adaptive Feed-Forward

Adaptive feed forward is similar to feed forward, but the system model keeps updating based on recent system measurements. In this paper it was chosen to implement what could be best described as adaptive linearized feed forward. This implies the estimated model used in the feed forward is a linearized estimation of the non-linear system. The linearization is around an operating point that exists in a k-dimensional space, in which each dimension is a parameter P_k that effects the dynamics of the system. This compensation method has, with enough training time, the ability to create a perfect linear model of the system, based on the parameters that it was provided. This method does require significantly more computational power than regular feed forward, and even more than feedback. This significant difference is that there is no latency requirement. The system can take a lot of time to make a relatively accurate model of the system, after that the system will work and start working better every training cycle. In the paper [6] this method and its implementation is described in more detail.

3.4 Modeling the System

In this section several options of describing the linearized system will be discussed.

3.4.1 IIR Filter

A system model linearized around parameters $\mathbf{P} = [P_1, \ldots, P_k]$ in the shape of an IIR filter, would be represented as the following:

$$G_{IIR} = \frac{a_n(\mathbf{P}) \cdot z^{-n} + \ldots\ldots + a_1(\mathbf{P}) \cdot z^{-1} + a_0(\mathbf{P})}{b_n(\mathbf{P}) \cdot z^{-n} + \ldots\ldots + b_1(\mathbf{P}) \cdot z^{-1} + b_0(\mathbf{P})} \tag{3.4}$$

Creating an IIR model of the system involved would require the following steps:

1. Convert the discrete model to a continuous model
2. Obtain samples of the system gain and phase difference at different frequencies and values of $\mathbf{P}$

3. Optimise the functions $a_n(\mathbf{P}), ..., b_0(\mathbf{P})$ to minimise the gain and phase estimation error.

When considering the model for this system, an IIR is a logical first step, because the lumped element model of the system would have a transfer function of this form. An advantage of this model is that it requires a lower order features than a FIR model. This indicates it may be less resource intensive to implement. The downside is that the optimisation problem that needs to be done in order to obtain an accurate model would be non-convex [7]. This is because the transfer function is not linear in its optimisation coefficients. Non-convexity implies there will be many local minima that the model could settle at that are not optimal. There are methods for overcoming this problem, but they are beyond the scope of this project. For this reason an IIR model was not chosen.

3.4.2 FIR Filter

The IIR model shape was not chosen due to its non-linearity, therefore the next logical choice would be to select a FIR shape to model the system. This form is presented below in Equation 3.5.

$$G_{FIR} = a_n(\mathbf{P}) \cdot z^{-n} + \ ..... \ + a_1(\mathbf{P}) \cdot z^{-1} + a_0(\mathbf{P}) \tag{3.5}$$

Many of the same ideas apply as for a FIR model shape, but with a few distinctions. The main advantage is that a FIR model shape becomes a convex optimisation problem.
The downside however, is that a FIR filter requires a much higher filter order in order to obtain similar model steepness characteristics. In fact, according to the paper [2] the model of the system may require a 30 dB gain difference in a span of 100KHz. With a sampling rate of 65 MHz, we can use a Harris approximation [8] to estimate the minimum order FIR required for a filter to be able to create the steepness required.

$$N = \frac{Fs \cdot \ Atten\,(dB)}{(22 \cdot \ dF)} = \frac{65 \cdot 10^6 \cdot 30}{22 \cdot 10^5} = 887 \tag{3.6}$$

This translates to 887 multiplications every sampling period, this is not implementable for the price requirements 8 of this project. An improvement would be to decrease the sampling frequency 10-fold. A sampling frequency of 6.5 MHz would lead to a filter order of 89.

3.4.3 Frequency Domain Filter

This type of filtering implies multiplying the magnitude spectrum of the input signal with the inverse of the magnitude spectrum of the system. This operation is functionally equivalent to convolution in the time domain such as is done with the IIR and FIR models. The model would have the following shape:

$$|G_{Freq}| = \mathbf{\Phi}(F, \mathbf{P}) \cdot \mathbf{W}_{|\mathbf{G}|} \tag{3.7}$$

$$\angle G_{Freq} = \mathbf{\Phi}(F, \mathbf{P}) \ \cdot \ W_{\angle G} \tag{3.8}$$

In this paper phase will not be compensated for. The reason for this is that it is not a requirement for this project. From now on $W_{|G|}$ will be referred to as W.
For some project requirements, this method of filtering is more efficient. A downside of this method of filtering is that the input signal needs to be split up into time bins in which the frequency components are fixed. This is common practice in audio processing, but the result of this is the frequency components present in the input signal can not change continuously. On top of that the implementation becomes more costly as the number of frequency components increases. Both of these issues are not present in the FIR filter implementation.

3.4.4 Comparison

In this section frequency domain filtering and FIR filtering are compared for this project. The performance indicator for this decision will be the number of multiplications required to implement the filtering. Before

this decision can be made, some variables relevant to these calculations must be clarified or extracted from the program of requirements (Chapter 2). However, as mentioned in the Bounding Analysis (Section 1.1), some parameters such as the $R_{\Delta P}$ and $R_{\Delta In}$ cannot be derived from the program of requirements, and due to the lack of a specific applications, were determined somewhat arbitrarily.

Variable Estimation

The rate at which the parameters will change $R_{\Delta P}$ will be set at 4 KHz as to not restrict the 1 KHz disturbance rejection non-functional requirement #4. The number of features in the feature vector that are not frequency dependant, N_ϕ, will be set to 8. Assuming this is a 8th order polynomial of a single parameter P_1, this will be able to capture the necessary complexities in the bandwidth required. It may also be a combination of 2 parameters P_1 and P_2 with orders 2 and 4 respectively. The maximum number of input frequencies N_F is set to 4. This comes directly from the non-functional requirement #9. The rate of input signal spectrum renewal $R_{\Delta In}$ can not be derived from the program of requirements. The value is chosen to be 100 Hz. Lastly the order of the frequency polynomial features, P_F, is chosen to be 8. As per the Ultrasound team's reasoning [6], this is sufficient to capture the expected complexities of the system.

Multiplication estimation

For an estimation of the multiplications required for the FIR implementation we can round the order require to 100 and obtain the following Equation 3.10.

$$\begin{aligned} F_s &= 6.5 \cdot 10^6 \text{ Hz} \\ R_{\Delta P} &= 4000 \text{ Hz} \\ N_\phi &= 8 \\ R_{M,FIR} \approx F_s \cdot\ 100 + 100 \cdot\ R_{\Delta\ P} \cdot\ N_\phi &\text{ Multiplications/s of FIR} \end{aligned} \tag{3.9}$$

$$R_{M,FIR} = 6.5 \cdot 10^6 \cdot 100\ +\ 100\ \cdot 4000 \cdot 8\ =\ 6.532 \cdot 10^8 \text{ Multiplications/s} \tag{3.10}$$

In the same way an approximation can be made for the multiplications required to filter in the frequency domain. The expression 3.11 can be seen below.

$$\begin{aligned} N_F &= 4 \\ R = max\,(R_{\Delta In},\ R_{\Delta P}) &= 4000 \text{ Hz} \\ R_{\Delta In} &= 100 \text{ Hz} \\ \text{Number of features in the feature vector inc frequency } &= N_\Phi = P_F \cdot N_\phi \\ P_F &= 8 \\ R_{M,\,Freq} \approx N_F \cdot\ R \cdot\ (2 \cdot\ N_\Phi\,) & \end{aligned} \tag{3.11}$$

$$R_{M,\,Freq} = 4 \cdot 4000 \cdot (2 \cdot 8 \cdot 8) = 2.048 \cdot 10^6 \text{ Multiplications/s} \tag{3.12}$$

As can be seen in Eq 3.9 and 3.11, frequency domain filtering is significantly less resource intensive for the requirements of this project. As such this option is chosen over a FIR filter.

3.4.5 Non-Linear Feature Choice

First and foremost the system in this paper will try to compensate for linear effects. This linearity implies the only relevant feature should be frequency. As such this will be a feature that is used to predict the gain of the system. The non-functional requirement #4 state that the system needs to include rejection of disturbances. This implies the system needs to adjust itself based on one or more parameters P_k that affect the behavior of the system. There are three types of parameters that could be used for this system.

Position derived Firstly, this system could extract features from the position measurements made. Of course the position measurements are already used for the modeling of the system, but theoretically the position measurements could be used to extract time domain data of the system and that could be used to construct features that effect the system. Although this would be a system with interesting characteristics approaching a feedback system and more, it would have its downsides. Besides for high complexity, many of the features would also require very low control latency. This makes these kinds of features not very desirable. The decision was made not to use position derived features for this reason.

Control Derived An idea for features that are easy to obtain, is features derived from the input signal. Features such as the input power [1], all the frequencies that are present in the signal, and past input signals could all be used as features. A large advantage of this is that for these features, there is no measurement latency because their values are known or require little calculation to obtain.

Environment Derived Other options for features could come form environmental parameters. Variables such as temperature [1] or air pressure are known to effect the performance of transducers, which implies they would work as useful features. Most features such as this work at a low rate of change, which implies a relatively high high latency between a change in feature and a change in control signal is acceptable. This is good because most commercially available sensors for these parameters have a high latency.

Choice The choice was made to create the system in such a way that any type of single feature could be plugged in and the system will model the effect of that feature. For now the decision was made to allow the possibilities of both temperature and input power as features. However, only a single one of these features can be used as feature at the same time, the choice will be up to the user.

3.4.6 Feature Mapping Method

Feature mapping is the concept of combining all parameters that may effect the system gain, and including their interactions. A definition must be given to $\mathbf{\Phi}(F, \mathbf{P})$. In this paper a polynomial feature mapping [9] is chosen. Reasoning for which can be found in the paper [6].

$$\mathbf{\Phi}\left(F, \mathbf{P}\right) = \begin{pmatrix} 1 \\ F \\ P \\ F \cdot P \\ ..... \\ F^n \cdot P^m \end{pmatrix} \tag{3.13}$$

Polynomial feature vectors have the main advantage that they require fewer multiplications and additions to calculate. Options such as Gaussian RBF's [10] require approximations of exponential functions which are expensive to calculate. On top of that, for polynomial feature vectors the existing feature vector can be grown iteratively. This is shown in Section 6.2.3.

3.4.7 Size of Feature Vector

With the method of feature mapping being chosen, the choice of feature vector Φ must be made. Size 64 was chosen. According to [6], this will be able to capture the dynamics of the system accurately enough to achieve the ripple in the flatband non-functional requirement #1.

3.5 Division of Teams

This project was divided into three teams, each responsible for different functionalities. The Ultrasound team [2] selected the transducer used and designed an amplifier capable of driving the transducer. The

System Modelling [6] team used the position measurements to train a model to improve the frequency response of the system. Finally the Position Measurement and Control team was responsible for collecting the position measurements and calculating the output signal to the amplifier based on the desired output and model parameters.

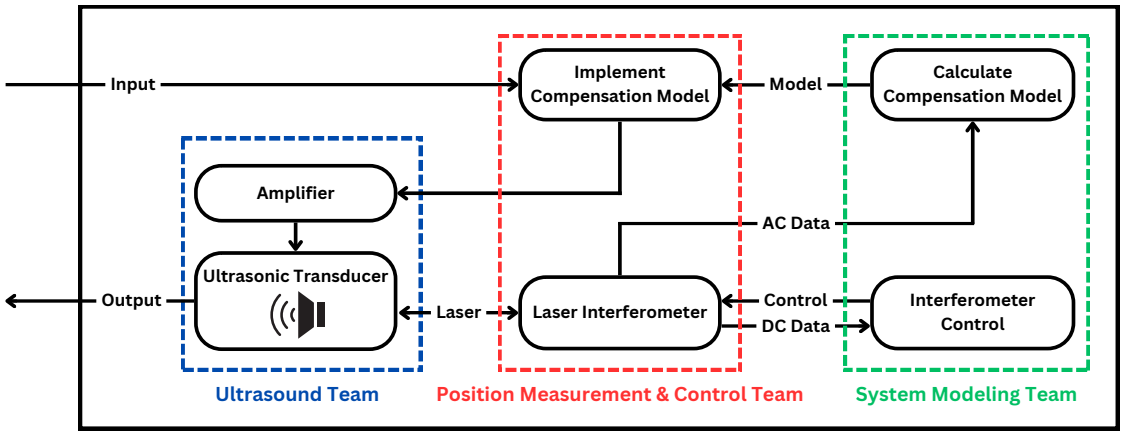


Figure 3.1: System functional diagram with division of tasks

4 System Design

4.1 Division of Specifications

4.1.1 Ripple and Noise

Non-functional requirement #1 specifies the final system is allowed 3 dB total gain ripple, which implies the maximum difference in gain in the passband may be 3 dB.

Distortion in the Ultrasound Signal Path

Distortion in the ultrasound signal path is the distortion caused by any components that are involved in creating the ultrasound. In theory the adaptive feed forward method will compensate for any linear, and some non-linear distortion in the ultrasound signal path. A factor of 1 dB will be budgeted to any non-linear distortions in the ultrasound signal path that are not accounted for, such as hysteresis.

Distortion in the Model Creation Path

Distortion in the model creation path cannot be corrected by the adaptive feed forward model, because it is the loop that creates the model. Because of this, any distortions will directly lead to ripple in the pass band. Any components between the interferometry setup and model calculation computer may add to the distortion in the model creation path. A factor of $3 - 1 = 2$ dB is left for distortion caused in the model creation path. 1.5 dB is attributed to the Position Measurement and Control team. This leaves 0.5 dB for the modeling team.

Noise in the Model Creation Path

Noise in the Model Creation Path will be attributed to the System Modeling team. The reason for this is that as will be shown, they have most control over it. Noise in the model creation path can cause noisy gain data points and colored noise in the model creation path can create distortion in the model. The effect of noise in this system becomes very complex, but several ideas below will come together the justify an approximation on the noise added by the model creation path. Firstly, since the input frequency is known, the SNR can arbitrarily be increased by increasing the size of the bin of time domain samples used for the Fourier transform. The limitation to this is that the size of the time bin used may not exceed the period at which the system compensates for disturbances. Given the rate of disturbance rejection requirement #4, this limit is set to 1ms. This implies an FFT would have a resolution of 1 KHz. This in turn implies that for white noise, 2 percent of the total noise in the flat band will translate to information about the gain. Secondly, the SNR of the signal can be kept at a sufficient level by setting a minimum input power required that can lead to valid information about the system. Lastly, the system modeling team can create a model based on many data points of the system, the effect of this is that noisy data points will get averaged.

The distortion caused by noise may cause 0.5 dB = 1.12202 ripple as per Section 4.1.1. This translates to noise that may at least have a spectral density of $\frac{1 \cdot 0.12202}{0.02 \cdot 50 \cdot 10^3} = 1.2202 \cdot 10^{-4} \frac{W}{Hz}$.

Noise in the Ultrasound Signal Path

Noise in the ultrasound signal path has two effects. Firstly, noise in the ultrasound signal path has the exact same distortion effect as the time domain noise in the model creation path noise. As such they can be added up to form the total noise that may cause distortion. Secondly, it is by definition the noise that is directly translated to ultrasound. Because of this there is a direct limit on this noise. The program of requirements (non-function requirement #3) states that the allowed noise in the passband is 1 mW. For white noise, this implies a maximum noise with spectral density $2 \cdot 10^{-}8 \frac{W}{Hz}$. This noise budget is spread equally over the Position Measurement and control team and the ultrasound team.

4.1.2 Price

The financial budget is divided over the three teams in the following way. The Ultrasound team is given a budget of €300. The System modeling team is given a budget of €50. The position measurement and control team is allotted the remaining budget of €1150. According to the interferometer provider [11], the complete interferometer will cost up to €500. The remaining budget of €650 must be divided between the ADC, DAC, and system platform. The ADC and DAC are therefore allotted roughly €200 each, while the system platform will be allotted €250.

4.2 Top Level Design

Following the decisions made thus far, an integrated system architecture was designed. The resulting system and the required connected hardware can be seen outlined in Figure 4.1. The details of each component will be outlined in the following three chapters, this chapter discusses the choices of hardware.

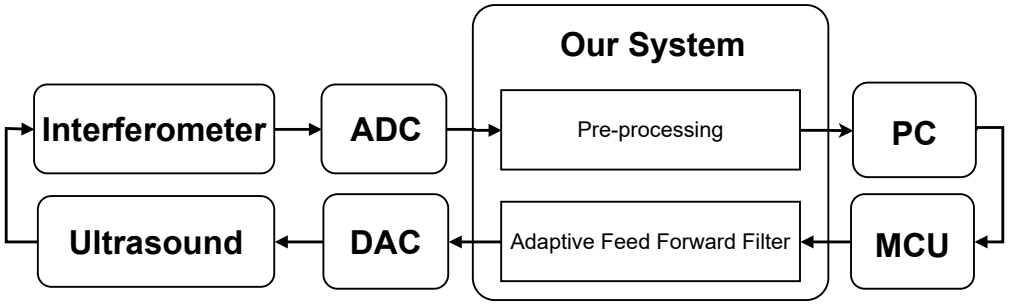


Figure 4.1: Top level system functional diagram with external components

4.3 ADC and DAC

In selecting the ADC and DAC the two most significant specifications to consider are sampling rate and resolution. In general the cost also increases proportionally to these specifications, so it is desirable to pick the minimum viable solution.

Given the input frequency of between 1.625 to 1.75 MHz, a sampling rate of at least 4 MSps is required for the ADC to prevent aliasing. However, due to the small bandwidth of the signal, aliasing may occur without it being destructive. This would decrease the theoretical lower limit of the sampling rate to 100KHz. For the the prototype of this project, a 14-bit ADC [12] with a sampling rate of 65 MSps was chosen. The reason for this was to ensure that the ADC was not a limiting factor. Another ADC with higher or lower specifications could be integrated into the project with relative ease. A 14-bit DAC [13] with a 165MSps was chosen for this same reason.

To ensure the 14-bit resultion does not cause any issues, the noise from both the ADC and DAC will be calculated. The quantisation noise [14] can be described with the following formula.

$$N(f) = \begin{cases} \frac{\Delta^2}{12 \cdot Fs} & -\frac{Fs}{2} < f < \frac{Fs}{2} \\ 0 & else \end{cases} \tag{4.1}$$

Where Δ is the resolution, and Fs is the ADC or DAC frequency. This results in a noise spectral density of

$$\frac{\left(\frac{1}{2^{14}}\right)^2}{12 \cdot 65 \cdot 10^6} \cdot 50 \cdot 10^3 = 2.4 \cdot 10^{-13} \tag{4.2}$$

With the unit being the ratio of noise in the flatband, to full signal power. For the ADC, the noise level can be very high due to the Fourier transform length possible, as explained in Section 4.1.1. For the DAC, whose noise is approximately half that of the ADC, the programme of requirements state this ratio is allowed to be 10^-3. This implies the noise caused by the DAC is negligible.

4.4 ADC and DAC Clock Selection

Both the ADC and DAC clocks will inevitably contain noise in the form of jitter (σ), which affects both the operation of these devices and the communication to/from them. For the ADC in particular the jitter affects the sampling aperture time, which for input signals with high slew rates can quickly degrade measurement capabilities. For a maximum slew rate $max(\frac{dV}{dt}) = A \cdot 2\pi f \approx 10.05\ V\mu s^{-1}$ and $V_{res} = V_{pp}/2^{bits} = 2/2^{14} \approx 122\ \mu V$ of the $f_{in} = 1.6$ MHz input signal, the threshold for jitter that will not contribute any noise can be calculated:

$$\frac{V_{res}}{2} \geq \sigma \cdot \frac{dV}{dt} \Rightarrow \sigma \leq 6.07 \text{ ps} \tag{4.3}$$

However, the noise added by jitter is evenly distributed spectrally, so as discussed in Section 4.1.1 this noise does not directly affect performance. The clocks selected for the ADC in this project are from the DSC1001 series (datasheet: [15]), and specifies a maximum jitter of 50 ps. As derived in an Analog Devices design note [16], Equation 4.4 gives the theoretical limit on the SNR of our measurements, which is 66 dBFS. For identical reasoning as in Section 4.3 this is negligible.

$$\text{SNR(dBFS)} = -20log(2\pi f_{in}\sigma) \approx 66 \text{ dBFS @ } f_{in} = 1.6 \text{ MHz} \tag{4.4}$$

Clock jitter is often less of an issue for a DAC [17], as it affects the timing of the output rather than the level. While this does contribute to phase noise, it does so at the sampling frequency rather than at the signal frequency. Given the same maximum slew rate and a jitter of 50 ps, the maximum noise power added to the ultrasound signal path can be calculated as follows:

$$P_{noise} = \left(\sigma \cdot \frac{dV}{dt}\right)^2 \approx 0.25 \text{ µW @ } f_s = 65 \text{ MHz} \tag{4.5}$$

This high frequency noise is both in a negligible range, and can easily be filtered out of the output signal when the signal frequency is significantly lower than the sampling frequency.

4.5 ADC and DAC Communication

The ADC and DAC communicate at the provided clock rate via parallel interfaces. In order to meet the timing requirements of these high speed interfaces the signals must be carefully routed from the ICs to the FPGA. The ADC outputs a clock to which its data is synced. This clock signal must be routed into the FPGA via a designated clock input pin, which is connected to the internal dedicated clock bus of the FPGA to guarantee low distortion. Most importantly the parallel signals between the FPGA and ADC/DAC must be routed at equal lengths, such that the delay between the signals is identical. Next the delay, setup, and hold times relative to the clocks as specified in their respective datasheets are provided to the FPGA design software, which uses this information to run a timing analysis.

4.6 Model Communication

Each time the PC updates the current model, the parameters of this model must be transferred from the PC to the FPGA. This is a relatively small amount of data, and there are no strict timing requirements for it, so a simple serial protocol such as UART can be used. The one other parameter that is needed is the extra feature, which is sent from the MCU to the FPGA. In order to simplify the design on the FPGA both the

model parameters and extra feature will be received through the same UART interface, meaning that the PC will send the model parameters via the MCU. At the standard rate baud rate of 115.2 KBaud/s, using binary signalling this gives a data rate of 115.2 KBit/s. This will allow for the extra feature to be transferred in $4 * 8/115200 \approx 0.3$ ms, and the entire model to be transferred in $300 * 8/115200 \approx 21$ ms. For extra features with higher bandwidth requirements, the extra feature could be measured by the FGPA directly.

4.7 ADC Sample Memory and Communication

The samples recorded by the ADC must be transferred to the PC for processing. With the sampling rate of 65 MSps and resolution of 14 bits, this equals a rate of 910 Mbit/s (113.75 MB/s).

Some possible options for transferring data at the rates required to a PC are Ethernet (1 Gbit/s), USB 3.0 (4.8 Gbit/s), or some form of internal connection such as PCIe or SATA. The latter are not standard on laptops, and were therefore not considered. Both Ethernet and USB SuperSpeed are both fairly complicated protocols, difficult to implement on an FPGA. Ethernet requires a PHY IC, however IP is readily available to be used. In real applications gigabit Ethernet often only can deliver 900 Mbit/s, which does not meet our requirement of 910Mbit/s, so the margin for error is small. An interesting contender is the FT600 IC, which takes a 16 width FIFO input and outputs the data over USB with a maximum bandwidth of 200MB/s. Due to the availability of a development board with the FT600 IC and the required minimum FPGA IO pin count, USB was selected.

The sample communication will also require a buffer, as the PC will be receiving the data and saving it in blocks. Between blocks the data must be intermittently stored in some form of memory. Based on real worlds tests the time between reads was determined to be 160 µs, or 10400 samples at 65 MSps. A FIFO of $2^{14} = 16384$ depth can be used for this. While this is still small enough to fit on a FPGA, if more time is needed between transfers, an external sample memory must be used.

4.8 System Platform Selection

The filtering and acquisition of position measurements must be done on a digital processing platform, of which there is a large variety. Some examples of possible choices are an MCU, FPGA or programmable DSPs. An MCU would not have the IO nor the processing bandwidth to handle the data handling requirements. A DSP would be able to implement the filtering section, however integrating this with the position measurement functionality could be challenging due to the limited flexibility compared to an FPGA. While an FPGA is not very cheap, it does allow for the greatest amount of flexibility, which is particularly important in prototypes. Given the time and budget constraints, and the fact that this is a prototype, a FPGA development board will be used rather than a custom PCB. Three requirements were considered for the selection of the FPGA development board, namely the available IO pins, logic resources, and peripherals. The pins required are: 15 for ADC, 15 for DAC, 2 for UART, 8 for LEDs, and 20 for FT600 USB IC, giving a total of at least 60 IO pins. The required logic resources can be seen in the Utilization report for the design (see Figure C.3). Finally the only peripheral required was either a Ethernet or USB 3.0 interface (see Section 4.7).

The Alchitry AU FPGA platform met all of the requirements, providing 100 IO pins, a daughter board with the FT600 USB IC, and 33K LUTs and 93 DSPs. The FPGA itself is a Xilinx Artix-7 XC7A35T. This development board also features 256 MB of DDR3 memory which could optionally be used as an equivalent two seconds (130 million samples) of sample memory as suggested in the previous Subsection 4.7. The price of this development board is currently €142, which meets the €250 budget specified in Section 4.1.2.

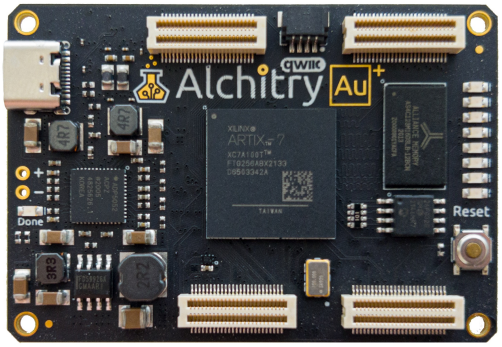


Figure 4.2: Alchitry Au FPGA

4.9 Custom Connection Board

Due to the high frequency (100 MHz) single ended CMOS communication used between the ADC, DAC, and FPGA, it is likely that wiring harnesses will degrade signal integrity significantly, causing errors in communication. To address this issue a PCB was designed to reduce the effects of noise, reflections, and impedance mismatches that wires are more susceptible to. On the PCB the signals are all routed the same length, the trace width is selected to the correct impedance, and there is adequate spacing between traces to reduce crosstalk. This PCB also has multiple clock generator ICs for the DAC and ADC, which are transmitted over controlled impedance coaxial cable.

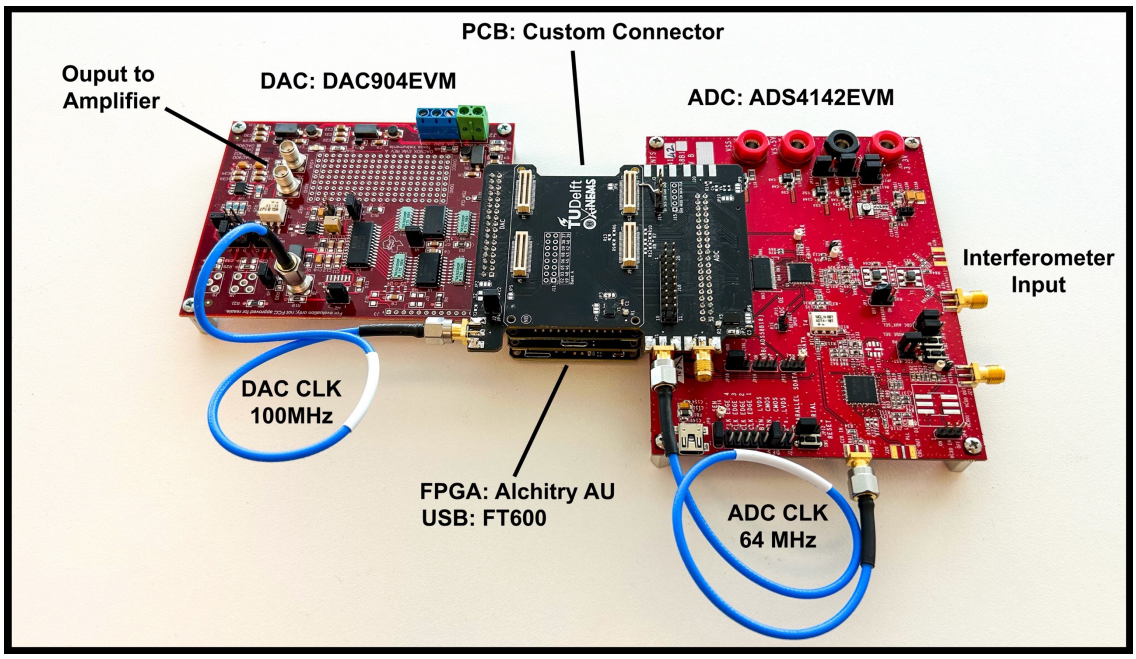


Figure 4.3: PCB designed to connect ADC, DAC, and FPGA

Calculations were required in order to calculate the required trace width to achieve the desired characteristic impedance $Z_c = 50\ \Omega$. As derived by Hammerstad [18] equations 4.6 and 4.7 can be used to approximate the characteristic impedance of a microstrip PCB trace. The electromagnetic wave in a microstrip line exists in both the dielectric substrate and the air above it. Since the relative permittivity of the substrate ε_r differs from that of air, the wave travels through an inhomogeneous medium. As a result, the propagation velocity lies between what it would be in the substrate and in air alone. Equation 4.6 proposes an effective permittivity ε_{eff}, which is the permittivity an equivalent homogeneous medium with the same propagation velocity would have. This approximation has an error $\leq 1\%$ for $0.05 \leq w/h \leq 20$ and $\varepsilon_r \leq 16$.

$$\varepsilon_{eff} = \frac{\varepsilon_r + 1}{2} + \frac{\varepsilon_r - 1}{2} \frac{1}{\sqrt{1 + \frac{12h}{w}}} = 3.2924 \tag{4.6}$$

Next these equations are applied given the substrate relative permittivity $\varepsilon_r = 4.4$, and substrate height $h = 0.2104$ mm, which are made available by the PCB manufacturer, and the desired $Z_c = 50\ \Omega$. Z_0 is the impedance of free space, which is approximately equal to 120π. This resulted in a value of $w = 0.349$ mm for the trace width.

$$Z_c = \frac{Z_0}{\sqrt{\varepsilon_{eff}}} \left[\frac{w}{h} + 1.393 + 0.667 ln\left(\frac{w}{h} + 1.444\right)\right]^{-1} \text{ for } \frac{w}{h} > 1 \tag{4.7}$$

5 Position Measurement

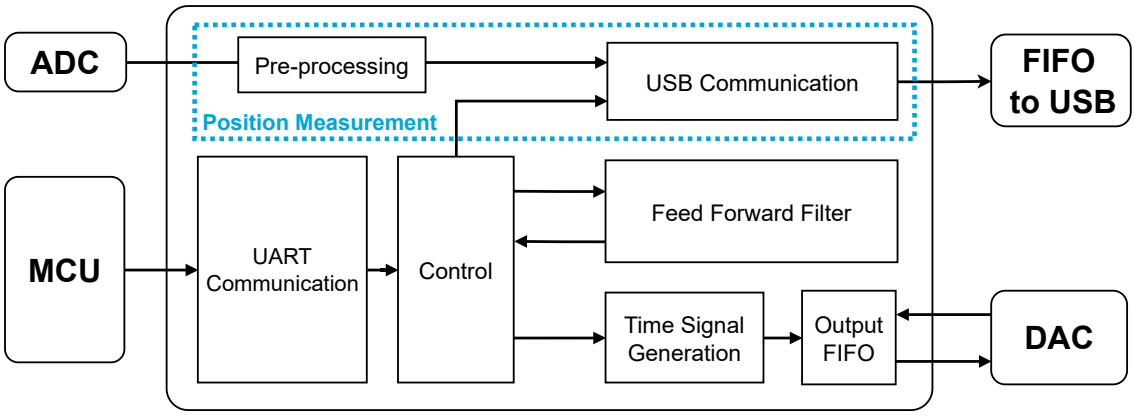


Figure 5.1: System diagram with external components

In this chapter laser interferometry and its use for measuring the vibration of ultrasonic transducers is discussed. This method of measurement leverages the interference property of light, and the fact that the interference pattern in this setup changes with a much lower frequency than that of light. While the underlying principle is relatively straightforward, it is essential to acknowledge the limitations inherent to this setup that must be addressed to ensure the technical requirements are met. The following sections provide an overview of the relevant requirements, a short introduction to optical interferometry, and a thorough analysis of the system.

Signal Integrity The interferometer may effect the position signal integrity by adding noise and by distorting the signal. Firstly, due to the fact that the interferometer is in the model creation path, the noise requirement is a relatively high $1.2 \cdot 10^{-}4\frac{W}{Hz}$ (see Section 4.1.1). Secondly, there is a limit to the distortion that can occur due to the method of position measurement. In Section 4.1.1 1.5 dB of distortion was budgeted to the position measurement of the model creation path. The entire distortion budget can be used for the processing of the interferometry data.

FPGA Resources Ideally the processing implementation requires as few FPGA resources as possible without causing significant distortion.

5.1 Interferometer Theory

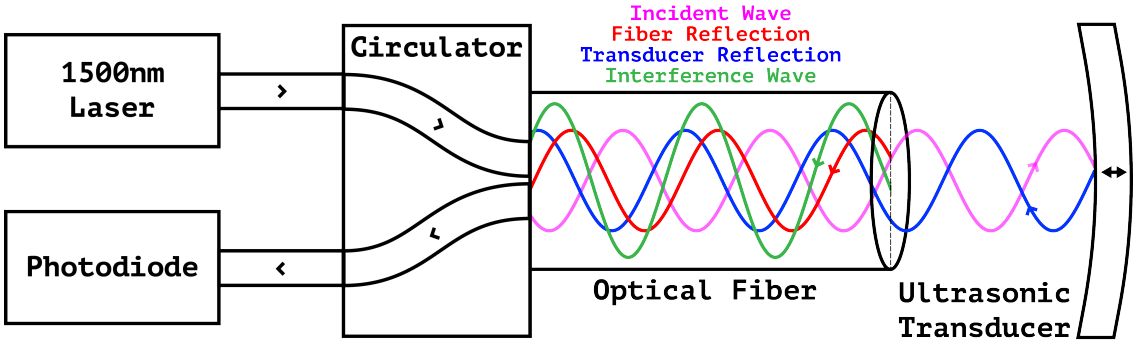


Figure 5.2: Diagram of interferometry setup

The basic principle of interferometry is depicted in Figure 5.2. The photodiode will measure the power intensity P of two light waves, which is dependant on the phase difference θ between the two waves and the ratio R of power that is reflected. In turn, θ is directly dependant on the distance to the transducer x.

$$\lambda = \frac{2 \cdot \pi \cdot c}{\omega_{light}} \tag{5.1}$$

$$\theta\,(x) \;= \frac{4 \cdot x \cdot \pi}{\lambda} \;= \text{Phase difference} \tag{5.2}$$

$$\hat{P} \;= \frac{1}{2} \cdot R(x) \cdot cos\,(\theta\,(x)) + \frac{R(x)^2 + 1}{4} = \text{Normalized power measured} \tag{5.3}$$

For convenience, the normalized power $\hat{P}$, the phase difference θ, and the distance x will be split up into their low and high frequency components.

$$\begin{aligned} x &= x_{LF} + x_{HF} \\ \theta &= \theta_{LF} + \theta_{HF} \\ \hat{P} &= \hat{P}_{LF} + \hat{P}_{HF} \end{aligned} \tag{5.4}$$

The high frequency signal will be considered the signal that carries the information, whereas the low frequency signal will be considered noise caused by disturbances such as thermal expansion. This leads to the following reconstruction formula for θ_{HF}. The derivation of this Equation can be found in Appendix A.1.

$$\theta_{\;HF} \approx arcsin\left(\pm \frac{\left(\hat{P}_{HF} + \hat{P}_{LF} - \frac{R^2+1}{4}\right)}{\frac{1}{2}R}\right) - arcsin\left(\pm \frac{\left(\hat{P}_{LF} - \frac{R^2+1}{4}\right)}{\frac{1}{2}\,R \cdot\, J_0\,(\theta_{\;HF,max})}\right) \tag{5.5}$$

Firstly, in this equation, R is the reflection coefficient that indicates the ratio of the amplitudes of the 2 waves that are interfering. In this equation the approximation is made that R does not change significantly with changes in x. Secondly, $\theta_{HF,max}$ is the maximum value of θ_{HF}. If $\theta_{HF}(t)$ has the form $\theta_{HF,max} \cdot sin(\omega t)$ then the approximation 5.5 becomes an equality. Later in the paper the effect of this approximation will be nullified, thus it is worth no further discussion.
Once θ_{HF} has been reconstructed, x_{HF} can be calculated using the following Equation.

$$x_{HF} \;= (\theta_{\;HF} + k \cdot\, \pi\,) \cdot \lambda \quad k \in \mathbb{Z} \tag{5.6}$$

5.2 Operating Point

In order to maximize the signal P_{HF} that is measured based on any changes of x_{HF}, an operating point of θ_{LF} was found. The derivation can be found in Appendix A.1.3.

$$\begin{aligned} \theta_{LF} &= \frac{\pi}{2} + k \cdot \pi \quad k \in \mathbb{Z} \\ \theta_{err} &= \theta_{LF} - \frac{\pi}{2} + k \cdot \pi \end{aligned} \tag{5.7}$$

5.3 Reflection Coefficient Estimation

As can be seen in Equation 5.5 the signal is dependant on R, the reflection coefficient. Although R does not change significantly, an accurate estimation is required to reconstruct x_{HF}. The value of R can be estimated by varying the wavelength, which is something the interferometry setup is capable of. This light should fluctuate between constructive and deconstructive interference. The maximum and minimum values of $P = Scale \cdot \hat{P}$ can be recorded.

$$P_{max} = \max_{\lambda} P = Scale \cdot \frac{1 + 2R + R^2}{4} \tag{5.8}$$

$$P_{min} = \min_{\lambda} P = Scale \cdot \frac{1 - 2R + R^2}{4} \tag{5.9}$$

P is obtained by measuring P_{LF} and P_{HF} and adding them. Next, Equations 5.8 and 5.9 can be used to obtain R and $Scale$ from the P_{max} and P_{max} that are measured.

5.4 Arcsine Approximation

Equation 5.5 requires a resource intensive implementation of an arcsine function.The system would be easier and cheaper to implement if the following approximation were to be valid:

$$x = arcsin(x) \tag{5.10}$$

This approximation is valid when:

$$|\theta_{err}| << \frac{\pi}{2} \tag{5.11}$$

$$|\theta_{HF}| << \frac{\pi}{2} \tag{5.12}$$

The derivation of which can be found in Appendix A.2.

5.4.1 Error Prediction

A simulation was made in which these approximations were made. In the simulation the following signal was input:

$$\begin{aligned} \hat{P}_{sim} = -\frac{1}{2} \cdot R(x_{LF}) \cdot sin(\theta_{err} + \theta_{HF} + k \cdot \pi) + \frac{R(x_{LF})^2 + 1}{4} \\ \theta_{HF} = \theta_{HF,max} \cdot sin(\omega_{test} t) \end{aligned} \tag{5.13}$$

An attempt was then made to reconstruct θ_{HF} as $\theta_{HF,EST}$ but using the approximation 5.10. A good indicator of the validity of the approximation is the gain of the input frequency component. Any variation in gain will imply distortion.

$$Gain = \frac{\Theta_{HF,EST}(\omega_{test})}{\Theta_{HF}(\omega_{test})} \tag{5.14}$$

Figure 5.3 shows the effect of this as a function of variations in $\theta_{HF,max}$ and θ_{err}.

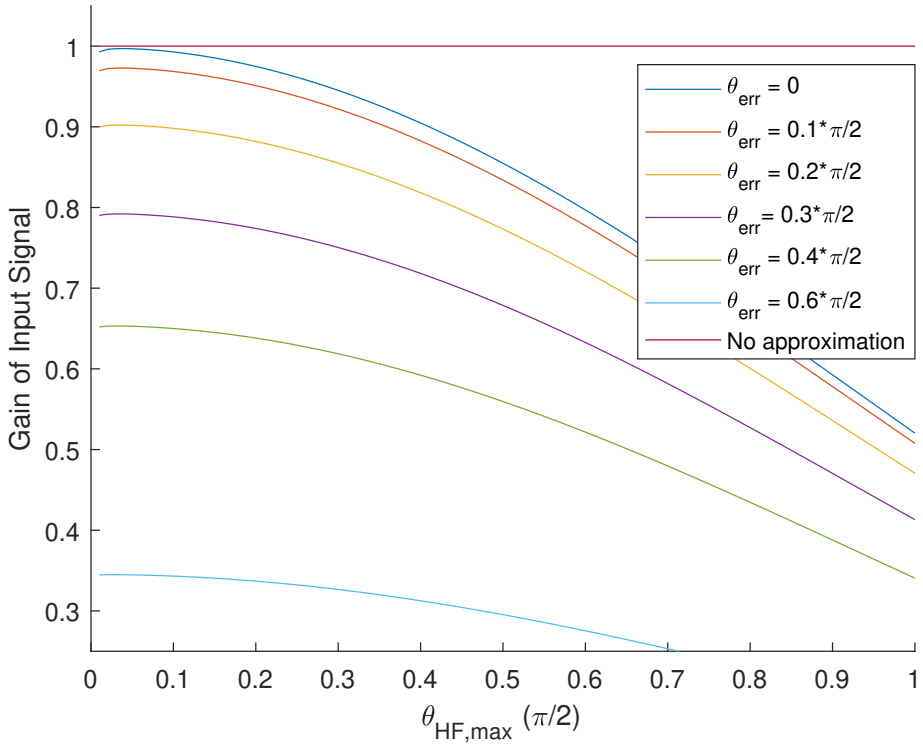


Figure 5.3: Gain variation caused by $Arcsin(x) = x$ approximation

From this figure an acceptable combination of θ_{err} and θ_{HF} can be derived. The functional requirements in Section 2.1 of the project allow a maximum ripple of 3dB in the flatband. In Section 4.1 this total ripple was divided over the entire system, which allows a factor of $1.4 = 1.15$ dB to be caused by the FPGA. Of course we would like to limit this, especially since there will be other sources of noise caused by the FPGA. We will internally allow the arcsine approximation 5.10 to cause a factor of 1.075 error.

5.4.2 Interferometry Simulation

In this section we will predict an achievable value of θ_{err} and use that to determine a value of θ_{HF}.

Phase Error θ_{err} Variation

Approximation 5.10 directly affects the ripple in the passband as shown in Figure 5.3. In this section we determine a proper estimate of the DC Error. Based on a conversation with Warner Venstra [11], the creator of the interferometry setup used in this paper, we modeled the setup to estimate the DC error. The model contains the following assumptions:

1. The position has low (< 0.07 Hz) frequency noise component with amplitude $> \lambda/8$

2. The λ can be tuned slightly to stay in the correct operating region 5.7. The control voltage to frequency transfer is modeled as a lowpass with 3 Hz cutoff and a 0.5 second delay.

The system was simulated with 0.07 hz noise and a PID controller that uses wavelength control to track setpoint $\theta_{err} = 0$. The result can be seen in Figure 5.4 below.

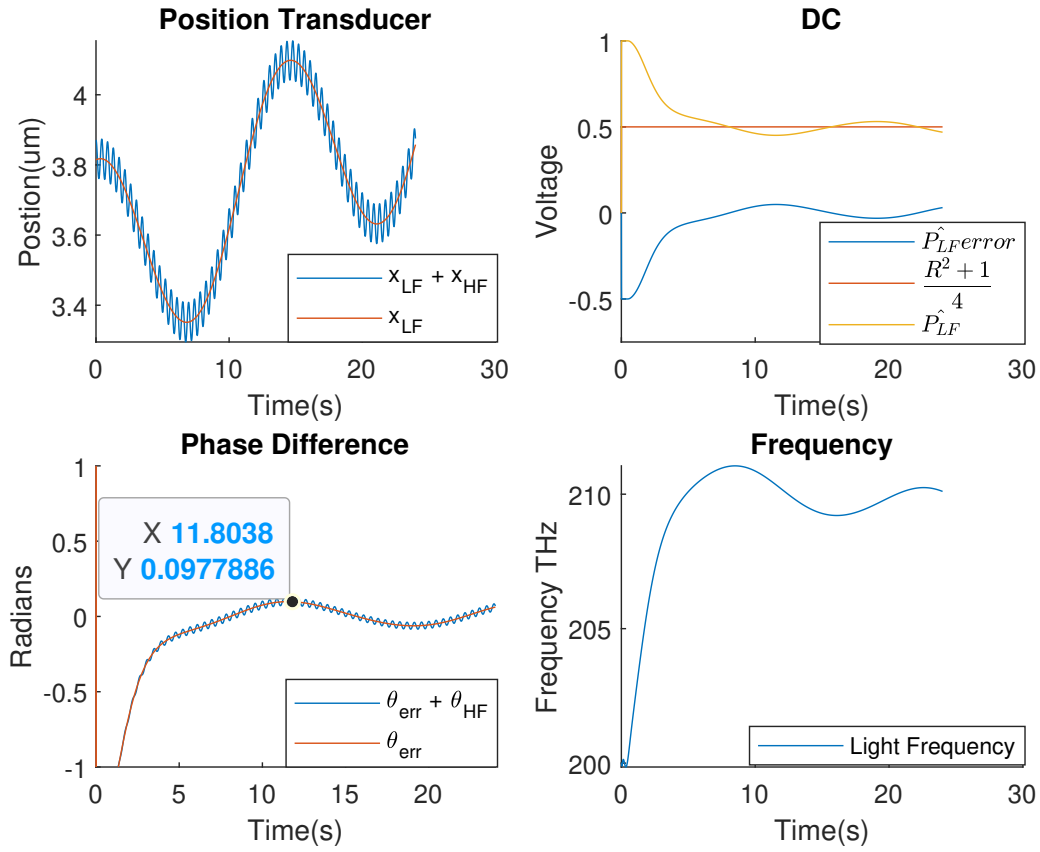


Figure 5.4: Phase Error With Frequency Control

Besides for error at startup, θ_{err} fluctuates to a value of 0.1 rad at most, which is $0.06 \cdot \pi/2$. Of course this is purely based on simulation, we will include a safety margin and round the maximum θ_{err} to $0.1 \cdot \pi/2$. Another team will present a micro controller implementation of the control system that will keep the interferometer at its desired $\pi/2$ operating point.

Choice of $\theta_{HF,Amp}$

The value of $\theta_{HF,Amp}$ can now be chosen based on the estimated maximum value of θ_{err} and the gain variation factor of 1.05 that is budgeted to the arcsin approximation.

Using Figure 5.3 and the conclusion draw from Section 5.4.2 we can safely say that $\theta_{HF,Amp}$ may have a maximum value of $0.3 \cdot \pi/2$ without exceeding the flatband ripple requirement #1.

5.4.3 Selecting a Laser Wavelength λ

We know from the ultrasound team paper [2] that the maximum value of x_{HF} will be 50 nm. This leads to a wavelength of:

$$\lambda_{min} = \frac{4 \cdot x_{HF,Amp} \cdot \pi}{\theta_{HF,Amp}} = 1333\,nm \approx 1500nm \tag{5.15}$$

We will round this up to 1500nm due to the availability of an interferometer with that wavelength. Assuming the validity of the assumptions we have made, with this wavelength interferometer laser, the approximation in this section 5.10 is valid.

5.4.4 Simplifications

Once the arcsin approximation is made, several simplifications can be made. Firstly, the low frequency power P_{LF} cancels out in the Equation A.12. This leads to the Equation below.

$$\theta_{HF} \approx \pm \frac{\hat{P}_{HF}}{\frac{1}{2}R} \tag{5.16}$$

In Equation 5.16 it can be seen that the value of θ_{HF} is now proportional to the high frequency power P_{HF} measured. In order for the proper gain factor, a proper estimation of R would be required. However, due to the fact that the gain is equal for all frequencies, this does not cause any distortion.

The result of this is the following expression.

$$x_{HF} \propto \hat{P_{HF}} \tag{5.17}$$

Which leads to the conclusion that the signal measured does not need to be processed.

5.5 USB Communication Format

To derive an accurate model the PC must receive the ADC samples, but it also must know what model and extra feature value is associated with these samples. In order to address this, along with the samples, the model ID and extra feature used will be sent to the PC. The model ID is a unique code assigned to the model by the PC upon creation. Since these parameters do not update frequently, and in fact must be constant for the entire width of each FFT done on the data, they are sent every N+S samples, where N is the FFT width and S is the amount of cycles needed for the system to settle to a steady state. S is configurable, and should include the time required to: calculate a new output, for the output to change, and for the new signal to propagate through the interferometry system and be measured. The ADC samples are 14 bits wide, while the data transferred over USB is 16 bits wide, meaning the first two bits can be used as markers for the model ID and extra feature in order to avoid problems with synchronization. The source code for this module can be found in Appendix B.12

5.6 ADC Noise Contribution

With the lack of pre-processing done, the only possible source of noise is from the ADC's sampling. The ADC noise would be added to the model creation path noise, is caused by quantization errors 4.3 and clock jitter errors 4.4.

6 Signal Generation

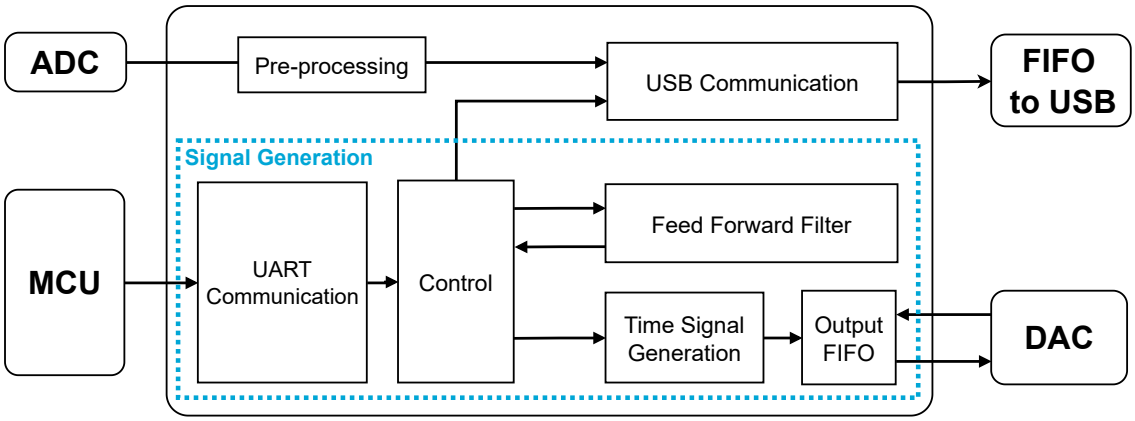


Figure 6.1: System diagram with external components

In this chapter the system components responsible for implementing the feed forward control are discussed. The relevant technical requirements for this subsystem can be divided into three categories, namely the noise, latency, and FPGA resource requirements.

Noise Noise caused by the signal generation can be described as noise in the ultrasound signal path. There are two thing requirements that need to be met regarding this noise. Firstly, the output noise budget for the signal generation section is half (see Section 4.1.1) of the noise allowed by the program of non-functional requirement #3. This implies the signal generation on the FPGA has a budget of 0.0005 watts. This is 0.5 percent of the maximum power. Internally this will be divided equally over the phasor generation and the time signal generation.

Latency As per the non-functional requirement #4 the system needs to be able to reject any disturbances at a bandwidth of 1 KHz. This implies the latency between a change in the external parameter P and a change in the output signal can be a maximum of 1 ms. As can be seen in Section 4.6, the parameter delay from the MCU to the FPGA via UART will be 0.3ms. To this we add the delay of the MCU, which we approximate to be another 0.3ms in the worst case. This leaves 0.4 ms for the feed forward filter and time signal generation. Internally we will allow the the feed forward filter to take 0.35 ms seeing as it requires the most calculation. The filter needs to filter 4 frequencies, which implies each frequency component translates to 87.5 us. On an FPGA with a 100 MHz clock, this translates to $8.75 \cdot 10^3$ clock cycles. This leaves $5 \cdot 10^3$ clock cycles for the time signal generation.

FPGA resources The price budget for an FPGA is chosen to be 250 euros. Our market research from vendors such as Mouser [19] shows that this will yield an FPGA development board resources on the order of 100.000 LUTs and 200 DSPs. Naturally, the price of the FPGA required for implementation needs to be as low as possible, but the limit is set to this number of resources. Seeing as the feed forward filtering is the most calculation intensive component, it will receive a budget of 2/3 of these resources.

6.1 UART Communication

The UART Communication component is responsible for processing the received commands and either updating the model or setting LEDs accordingly. The LEDs are included as a tool for debugging. The communication between the MCU and FPGA was chosen to be implemented using UART. A simple custom data structure that can be seen in Figure 6.2 was designed consisting of one command byte followed by the N data bytes expected based on the command sent. This is possible due to the various parameters having fixed widths.

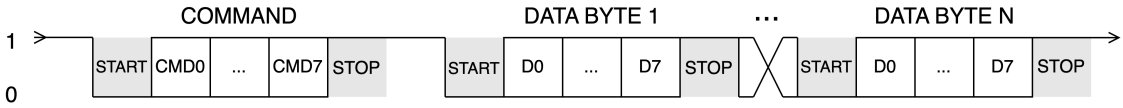


Figure 6.2: UART Protocol Diagram and Data Structure

On the FPGA, the UART protocol itself was implemented using a simple Xilinx IP UART state machine. This IP abstracts the UART to a simple parallel communication with write and ack handshake bits. The UART configuration used is baud rate = 115200 baud, data bits = 8, parity bits = none, stop bits = 1, and flow control = none.

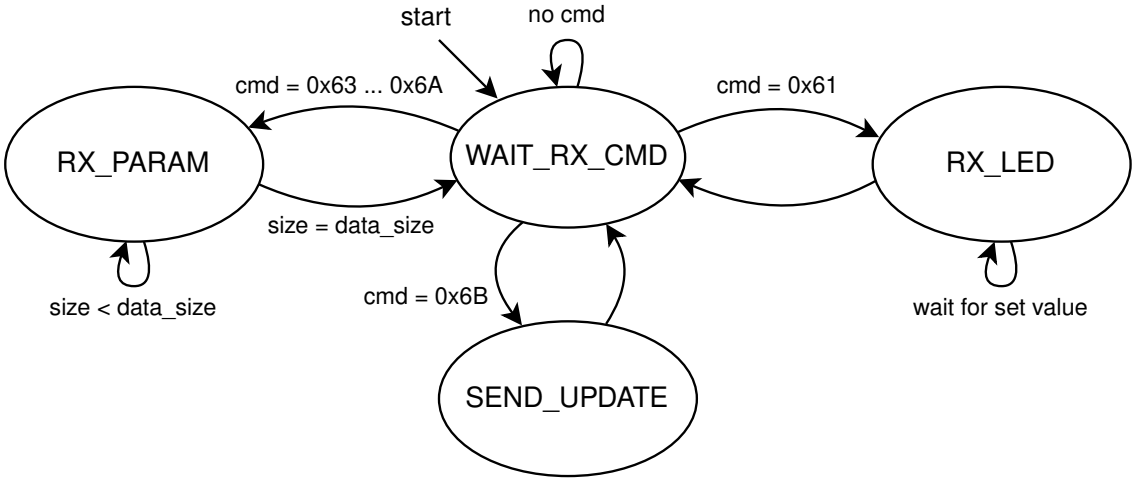


Figure 6.3: State Machine Diagram of UART Communication

The state `SEND_UPDATE` informs the FPGA that all parameters of the model have been updated successfully, after which the new parameters are made available to the control module. The source code for this module can be found in Appendix B.11

6.2 Feed Forward Filter

The feed forward filter module applies the inverse of the system gain $|G_{est}|$ to a single input frequency component F_k. The result of this operation is the magnitude $|In_k|$ at which the frequency component F_k will be sent into the system.

6.2.1 Toplevel Design

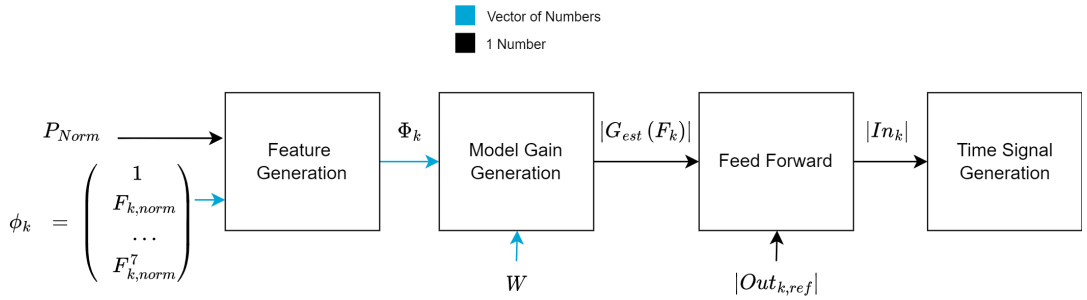


Figure 6.4: Top Level of the Feed Forward Filter Component

As can be seen in Figure 6.4 the feed forward filter component is split up into 3 sub-components. Firstly, the feature generation, which combines the parameter P with a small feature vector ϕ_k to create the final feature vector Φ_k. Secondly, the Model Gain generation sub-component calculates the estimated system gain $|G_{est}|$ at frequency F_k. This is done by scalar multiplying the feature vector Φ_k with the weight vector W. Lastly, the actual feed forward step is applied, in which the reference output magnitude $|Out_{k,ref}|$ for F_k is divided by the estimated system gain $|G_{est}(F_k)|$. The result is the value In_k, which is the magnitude at which the frequency component F_k will be sent into the ultrasound system via the DAC.

6.2.2 Design choices

Arithmetic Format and Size

A choice needs to be made regarding the size and type of representation of the numbers that the calculations in this module are done with. Firstly, the type of representation is chosen, for this, fixed point and floating point are the choices. Fixed point representation has a constant resolution, but it has a limited range. Floating point representation has a much larger range but its resolution is dynamic, implying that its resolution is much lower at high ranges. Even with methods to compensate for it 6.2.2, the values used in this module can vary orders of magnitude. For this reason, floating point representation was chosen. Vivado, the VHDL analysis and synthesis tool that is used for this project, has great IP that allows for easy floating point integration into projects.

Secondly, the bit size of the floating point will effect its range and the relative resolution. More specifically, the exponent will determine the range, and the mantissa will determine the resolution. The datasheet of the Vivado Floating Point IP [20] states that the addition, multiplication, and division operators are all accurate to half a ULP (Unit Last Place) as per the IEEE standard [21]. For a 16-bit float with a 10-bit mantissa, the worst-case ULP is $\frac{1}{1024}$ of the actual value. This implies a $0.5/1024 \approx 0.05$ percent error added every time a floating point IP is used.

In the worst case, the following number of operations in series would be required. In feature generation, the variable P_1^8 will require 8 multiplications. In Model gain generation the scalar multiplication of the feature vector Φ with the weight vector W takes 1 multiplication and $\log_2(64) = 6$ additions. Lastly, the feed forward step requires 1 division. We find that the maximum error due to 16 bit floating point rounding in the feed forward filer is $1.0005^{16} = 1.00803 = 0.803\%$. This is equivalent to 0.0347dB, which directly translates to ripple in the pass band. This is an acceptable number seeing 1.5 dB of ripple was budgeted to the implementations in this Section 4.1.1. It is concluded that a 10-bit mantissa is sufficient for the requirements of this project. This would leave an exponent of 5 bits. This implies the 16-bit floating point has a range of 2^16 till 2^-14 and an identical range with the reverse sign. In this project, this leads to the restriction that no number can exceed these ranges. If this range requirement is met, 16 bit floating point arithmetic will allow the program of requirements to be met.

Feature Format

Due to the range limit of the 16 bit floating point 6.2.2, there is a limit to the size of F and P. This upper limit is $|F \cdot P_1| < \sqrt[8]{2^{16}} = 4$. The lower limit is $|F \cdot P| > \sqrt[8]{2^{-14}} = 2^{-\frac{14}{8}} \approx 0.3$. This range is equally divided over F and P_1. On top of that, to simplify, only the positive range will be used. This implies both the F and P need to be normalised to a value between $\sqrt{0.3} = 0.55$ and $\sqrt{4} = 2$. The following formula will be used to normalise the frequency F and parameter P_1.

$$X_{norm} = \left(\frac{X - \frac{X_{min}+X_{max}}{2}}{\frac{X_{max}-X_{min}}{2}} + 1.275 \right) \cdot 0.725 \in [0.55,\ 2] \text{ where } X \in [X_{min},\ X_{max}] \tag{6.1}$$

Using the mapping equation 6.1 the values of F_{Norm} and $P_{1,Norm}$ are obtained. These calculations are done on the MCU and on the PC.

6.2.3 Feature Generation

This sub-component takes the small feature vector ϕ and combines it with the parameter $P_{1,norm}$ to create $\mathbf{\Phi}$. The main design goals for this module are to generate the feature vector Φ with a low latency, while taking few FPGA resources.

For a frequency polynomial(P_F) of 8th order and an extra parameter order N_ϕ of 8. The number of multiplications required is 56. Assuming an equal division of the latency budget 6 over the feed forward filter component, the maximum latency for the feature generation is 3000 clock cycles. The feed forward filter component was budgeted 66 percent of the total FPGA resources. Both the feature generation and estimated system gain generation will receive 30 percent of the total resources, leaving 6 percent for the actual feed forward step.

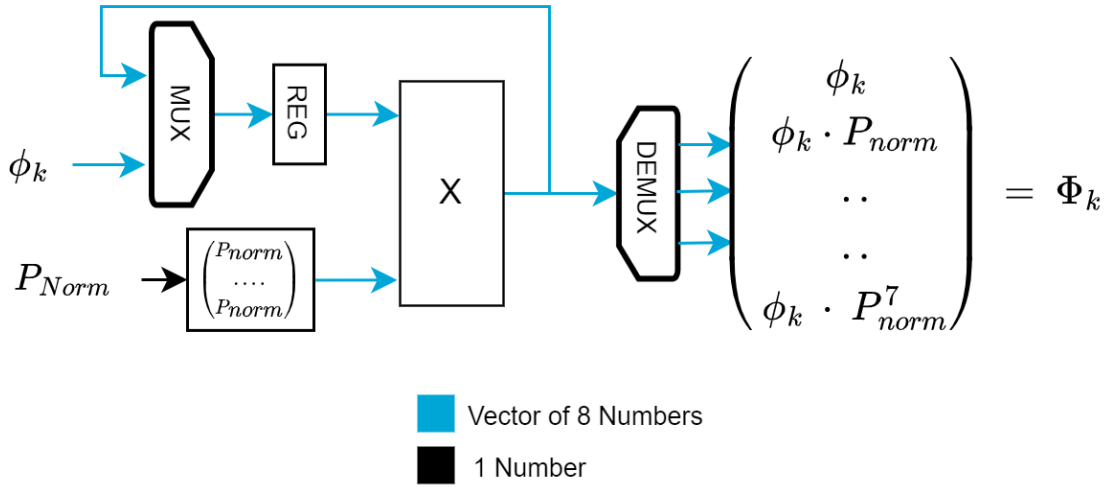


Figure 6.5: Estimated System Gain Generation

For the feature generation, it was chosen to implement the calculation by doing 7 consecutive cycles of 8 parallel multiplications. The implementation that can be seen in Figure 6.5 requires a multiplication unit that does 8 floating point multiplications in parallel. This can be done with floating point multiplier IP from Vivado [22]. The multiplier has several parameters that can be tuned. Firstly, its ”latency”, which according to the documentation is the number of internal registers. The latency is set to 0, the reason for this is that the documentation [22] states that increasing the latency does not create any re-use of resources. Its only advantage is that it allows for a higher clock rate. In the feature generation sub-component, the multiplier in series with a MUX is the longest combinatorial path. According to the documentation, this

is faster than the 100 MHz clock used. The other parameter that can be tuned is the division of LUTs and DSPs. A single 16-bit floating point multiplier can be implemented using 200 logic elements or 90 logic elements and a single DSP. This choice is not binding due to the ease of changing the IP. The choice was made to use purely logic elements. The design implemented in VHDL can be found in Appendix B.3

6.2.4 Estimated System Gain Generation

This sub-component scalar multiplies the weight vector W and the feature vector Φ. The same 8 floating point multiplier block as in the feature generation 6.2.3 is used. The design for the estimated system gain generation can be seen in Figure 6.6.

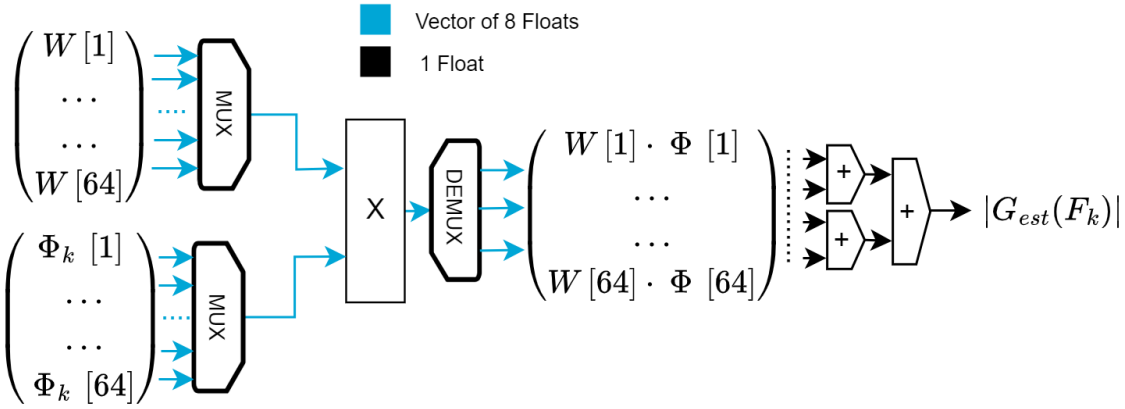


Figure 6.6: Model gain generation system diagram

This implementation does eight of the 64 required multiplications in parallel. Once the multiplications are finished, the results are added up in an adder tree. For a 64 by 64 scalar multiplication, an adder tree of depth 6 is required. This implies the data needs to move through six adders in series before it is saved. With a 100 MHz clock cycle and the floating point adder IP used in this project, this causes timing issues. For this reason, the adders were configured to have a latency of 1 clock cycle, which prevents timing issues.

The documentation of the floating point adder IP [23] states that a 16 bit floating point adder takes 170 LUTs, or 130 LUTs and one DSP. This is approximately the same as the multiplier. The reason for this is that for floating point numbers, addition is a complex operation. For the implementation in this paper, that was overlooked and the adder tree shown in Figure 6.6 was implemented. When adding many numbers, an adder tree is the option with the lowest latency, but it requires the most adders. When adders are cheap to implement, like for fixed point arithmetic, this is not a problem. However, for floating point numbers this significantly increases the number of resources required. The current implementation of the estimated system gain generation component requires 8 multipliers and 127 adders and has a latency of 14 clock cycles. This requires 24k LUTs to implement, which is very close to the 30k LUT budget. A simple improvement that is recommended, is for the implementation in this paper to split the required additions up into at least 16 clock cycles. This would decrease the number of adders required to 8 and decrease the number of FPGA resources required by the estimated system gain generation by a factor of at least 12. This is a substantial improvement for the entire Position measurement & control system, seeing as the feature calculation without this improvement is responsible for more than 90 percent of the total resource requirements. The design implemented in VHDL can be found in Appendix B.2.

6.2.5 Feed Forward Step

Once the estimated system gain has been calculated, a simple calculation can be done to apply the feed forward step. The desired output magnitude $|Out_{k,ref}|$ will be divided by the estimated system gain $|G_{est}(F_k)|$ as explained in section 3.3.2.

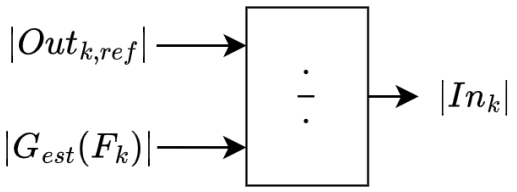


Figure 6.7: Feed Forward Filtering Step

The division is done by floating point IP provided by Vivado [22]. A single division has been configured to take 2 clock cycles which allows for the reuse of hardware. Because of this the latency is also 2 clock cycles. The number of required LUTs for this implementation is 500. The design implemented in VHDL can be found in Appendix B.9

6.2.6 Feed Forward Filter Sub-Component Integration

The 3 sub-components that make up the feed forward filter were combined together. The result is a component that takes 27k LUTs and has a latency of 23 clock cycles. This meets the requirements, but a substantial improvement should be made. If the recommended method of alternative addition were to be implemented, the number of LUTs required for the feed forward filter component would decrease by a factor of approximately 10x. This system would require approximately 3k LUTs and have a latency of 39 clock cycles. This would still be extremely low latency when compared to the minimal latency requirement 6, but the FPGA resources required would be much lower.

The VHDL code for the sub-component integration can be found in Appendix B.7. The simulation can be found in Appendix C.1

6.3 Time Signal Generation

In this component the time domain signal is synthesized. A diagram of the design can be seen in Figure 6.8 below.

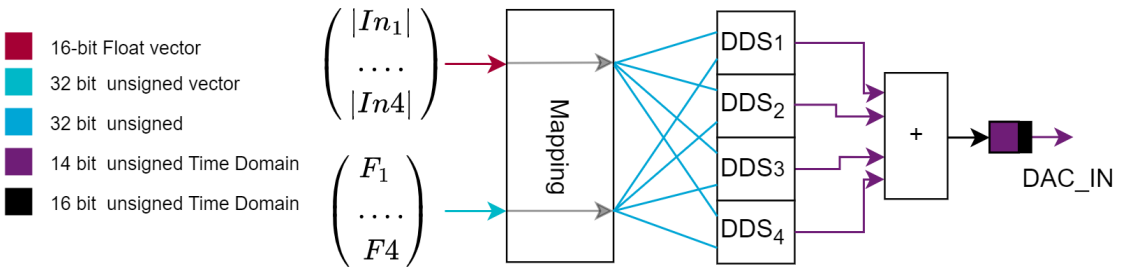


Figure 6.8: Time Signal Generation Diagram

As can be seen 4 DDS modules in parallel are responsible for the the creation of the time domain signal for each frequency component F_k. These are then added to create the signal that is used as input for the

DAC. The DDS (Direct Digital Synthesizer) modules are created with Vivado IP. A single DDS module takes as input a frequency and a magnitude, and outputs a time domain signal. The choice was made to represent the frequency and magnitude as 32 bit numbers and the output as 14 bit numbers. This leads to a frequency resolution of 0.02 Hz, which ensures the model creation team can do a very large FFT [6]. For these parameters, the spurious free dynamic range is 90dB. This approximately translates to a 10^-9 dB ripple in the passband, which is negligible. In total, the time signal generation requires 16 DSP, mainly due to the DDS and a few hundred LUTs. The latency is 5 clock cycles.

The VHDL implementation for both the time signal generation and the feed forward filter can be found in Appendix B.10. The simulation of this component can be found in Appendix C.2.

6.4 Control Module

The control module is responsible for controlling the signal generation and updating the model ID and extra features reported for each sample bin. The new parameters received from the UART Communication component are converted into output magnitudes $|In_k|$ and frequencies Fk for each frequency using the Feed Forward Filter component. These updated phasors are then sent to the Time Signal Generation at the beginning of each new time bin. The counter for the time bins is implemented in this module, the size of which is discussed in detail in Section 5.5. The source code for this module can be found in Appendix B.1

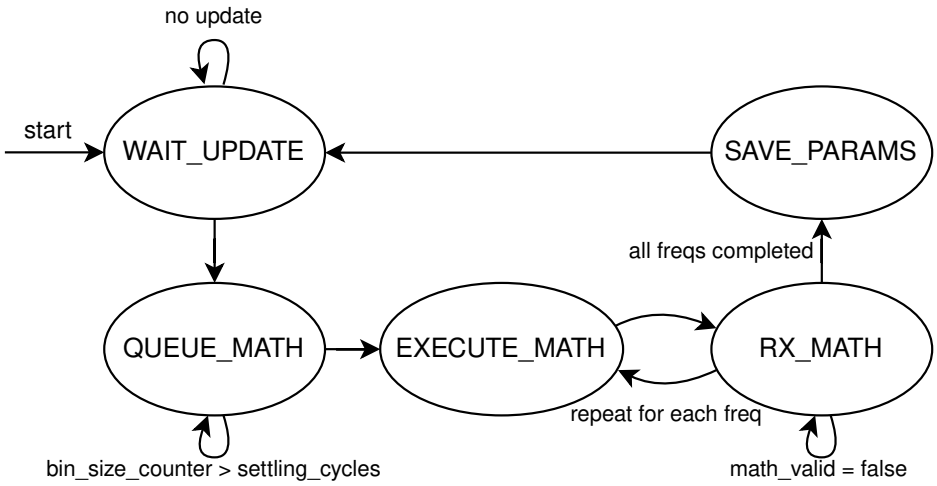


Figure 6.9: State Machine Diagram of Control Module

6.5 DAC Output

The outputs generated by the time domain signal generation cannot be directly sent to the DAC, because the DAC uses an external clock for timing the data input. This clock is also connected to the FPGA, which will output data as requested by the clock. In order to synchronize the output from the internal to the external clock domain, an asynchronous FIFO is used. The internal and external clocks both operate at 100 MHz, but should and frequency differences occur that cause there to be no new output available, the previous output value will be used. The depth of the FIFO is limited to two in order to avoid incurring delays and samples building up between the two domains.

6.6 Signal Generation Integration

The control module, feed forward filter component, and time signal generation component were added together in a VHDL signal generation component. The component was tested in simulation, the result of which can be found in Appendix C.2. The VHDL code for the signal generation component can be found in Appendix B.5.

7 Prototype Implementation and Validation

7.1 Simulation and Synthesis

Throughout the process of development the various individual components were simulated with testbenches to verify operation as well as for debugging issues. Next the individual components were synthesized to further test the feasibility of implementation in practice, as well as estimate the utilization of logic resources. An important part of this was to specify constraints on delays and frequencies from the various external interfaces so that a valid timing analysis can be performed. After this the components were integrated together one at a time, starting at the UART input and moving down the data path to finish with the DAC output. Simulations showing the successful testing of various components and the system as a whole can be found in Appendix C. Once the entire system was integrated, the total required logic resources was reported as can be seen in the utilization report in Appendix C.3.

7.2 Testing and Validation

After the system was verified to work in synthesis, the design was implemented in hardware. This process was split into two steps in order to better isolate errors.

7.2.1 Data Acquisition

The data acquisition, consisting of the ADC, USB Communication component, and FT600 USB interface, was tested as an isolated system. On the PC a driver provided by the FT600 manufacturer was used to read the USB pipe. The measurements from Table 7.1 show that the desired data rate of 113.75 MB/s was met, however the performance did not meet the theoretical limit of 200 MB/s. Using a pattern generator to emulate the ADC, it was verified that the data has a 0% error rate. This also confirms that the internal FIFO size of $2^{14} = 16384$ depth is sufficient to avoid data loss.

Table 7.1: USB data rates vs amount of bytes requested

Transfer Size [KB]	**Average Data Rate [MB/s]**
16	57.7
32	77.1
64	82.0
128	155.8
256	173.3
512	174.7
1024	174.6

In the above speed test data was sent every clock cycle in order to test the maximum bandwidth of the USB interface. It can be seen that the data rate increases as the requested transfer size increases. This is not a limitation of the FPGA implementation or the FT600 USB interface, but rather a limitation within the PC or provided driver.

7.2.2 Complete System

To test our entire system without requiring integration with the ultrasound and model training systems, the output of the DAC was connected to the input of the ADC. Subsequently, carefully chosen model

parameters were transferred to the system via the UART interface. The output on the DAC was then measured by the ADC and transferred via USB to the PC. Here the resulting waveform could be compared to the theoretically expected waveform given the model parameters defined. As can be seen in Figure 7.1, the data collected matched expectations, and the outputs matched the expected frequencies and magnitudes, as well as noise requirements. This procedure was repeated with a variety of input parameters.

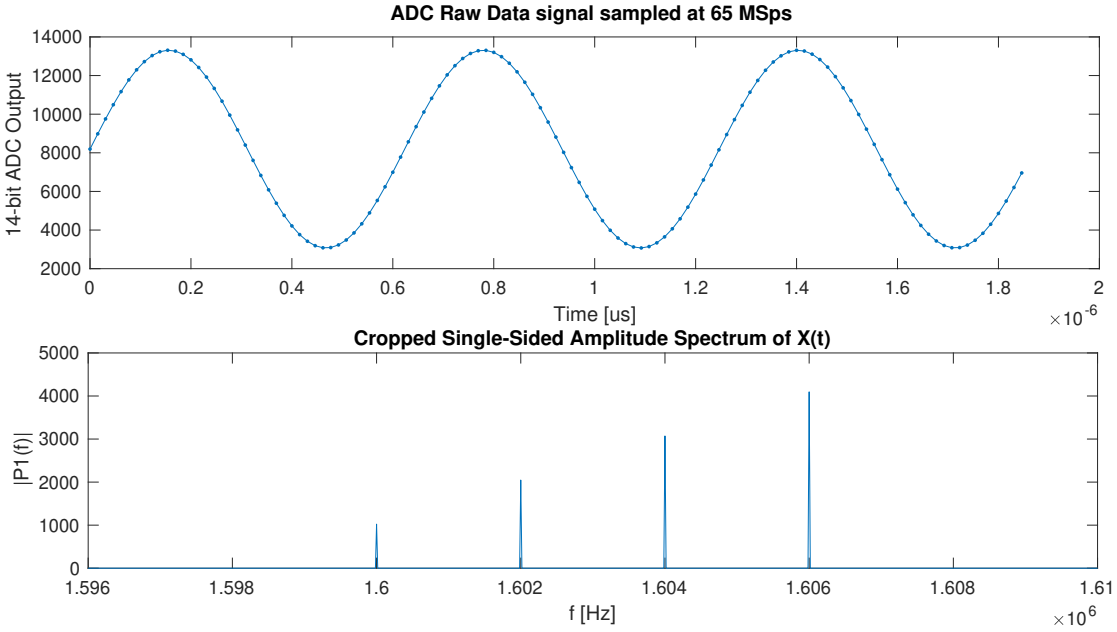


Figure 7.1: Raw measurements and frequency spectrum of complete system test

In the above test the UART parameters were specified such that:

- $F_1 = 1.600$ MHz and $|In_1| = 0.25$
- $F_2 = 1.602$ MHz and $|In_2| = 0.5$
- $F_3 = 1.604$ MHz and $|In_3| = 0.75$
- $F_4 = 1.606$ MHz and $|In_4| = 1$

Additionally a significant spike was observed at 0 Hz, which is to be expected due to the $2^{13} = 8192$ DC bias that can be seen in the above Figure 7.1. This is an expected result of the ADC output format, as the -1V to 1V range of the ADC is distributed from 0 to 2^{14}. Unfortunately integration with the ultrasound and model training systems was not achieved within the time constraints, however this would be the next logical step in testing the system.

8 Discussion

After considering a variety of possible solutions to the problem of flattening the frequency response of an ultrasonic transducer, a design was made considering the program of requirements. Unfortunately due to time constraints, the subsystem presented in this thesis has not yet been integrated with the ultrasonic and model training systems. While the requirements were met, some tradeoffs were made that may not have been optimal. For instance, while the price requirement of €1500 was met, the system in it's current prototype form could be made much more affordably, albeit not in the same short time span. Another non-optimal implementation was the resource consumption of the Feed Forward Filter. As discussed in Section 6.2, the current implementation exceeds the specification for latency by orders of magnitude. As it is the most resource intensive component, by increasing the latency and reducing the resource consumption, a lower cost FPGA could be used.

Another cost optimization could be made by changing the selection of ADC. The system is currently oversampling by a factor of 20 with respect to the Nyquist rate. Furthermore, sub-sampling could be used in this application, as long as destructive aliasing in avoided. A reduction in sampling rate would both reduce the cost of the ADC, and reduce the bandwidth required to transfer the acquired data, simplifying the system. The prototype developed allows for a reduction in sampling rate, so the sub-sampling performance can be evaluated without any changes to the system.

For the position measurement, an extensive analysis of the system was done, resulting in the conclusion that for the given operating range of the system, no pre-processing was needed in order to acquire satisfactory measurements. This greatly simplified the expected design, and allowed for the use of a more cost effective FPGA. A valuable byproduct of this analysis was a deeper understanding of the system's behavior, which proved beneficial for both the design and troubleshooting process.

9 Conclusion

Overall, the system performed its intended functions and met the requirements. Design challenges were faced, particularly in terms of logic resource utilization and cost optimization. These aspects provide valuable insights for future improvements, including exploring alternative model architectures, extending the operating range, and a variety of changes that could reduce system cost. Throughout the development process of this system, many possible avenues of future work were discovered to further enhance the capabilities and functionality of the developed system.

- Frequency Update Latency: The latency bottleneck for updating the frequency components of the output currently lies in the speed of UART. While this meets the specifications per the program of requirements, should this latency be reduced, the system would be capable of generating more dynamic outputs, such as pulses. There are many practical applications in which this functionality would increase the utility of our system.
- Expanded Operating Range: While not required, extending the operating range of the position measurement interferometer would greatly improve the flexibility of the system. Increasing the range would involve accounting for the arcsine distortion as discussed in Section 5.4, as well as accounting for the signal no longer being directly proportional to displacement.
- Time Domain Implementation: While it was not the optimal solution given the provided program of requirements, modelling the distortion with a time domain filter rather than in the frequency domain as explored in Section 3.4 does provide numerous benefits, most notably the ability to input an arbitrary non-periodic signal.
- Non-Linear Modeling and Neural Networks: A promising direction for future work involves the development of a complete non-linear model for the system. Incorporating time samples as input features, such as in a neural network, could potentially enhance the system's ability to accurately predict and control the behavior of ultrasonic transducers, eliminating the need for the linearizations made in the current method.
- Digital versus Analog Systems: An interesting area to explore is the comparison between digital and analog feedback systems. While making a dynamic analog system is complex, they have the clear advantage of operating fast enough to provide direct feedback.
- Data Compression: Considering the implementation of the FFT on the FPGA, would allow for only the relevant frequency information to be sent, greatly reducing the required transmission bandwidth. However, the feasibility and cost-effectiveness of this approach should be further investigated, taking into account the additional complexities and expenses associated with its implementation.

By pursuing these avenues of future work, the system can be further enhanced, expanding its capabilities, improving accuracy, and exploring new possibilities in the field of ultrasonic transducer control and measurement.

Appendix

A

Interferometry Theory Derivation

A.1 Derivation of θ_{HF}

A.1.1 Basic Interferometry

In this section below an expression for the reconstruction of θ_{HF} is derived.

$$\begin{aligned}
\omega_{light} &= \text{Angular frequency of light} \\
x &= \text{Distance optical fiber to transducer} \\
\lambda &= \frac{2 \cdot \pi \cdot c}{\omega_{light}} \\
R(x) &= \text{Reflection coefficient}[0,1] \\
\theta(x) &= \frac{4 \cdot x \cdot \pi}{\lambda} = \text{Phase difference} \\
\text{Signal into diode} &= sin(\omega_{light} \cdot t) + R(x) \cdot sin(\omega_{light} \cdot t + \theta(x)) \\
&= 2 \cdot sin\left(\omega_{light} \cdot t + \frac{\theta(x)}{2}\right) cos(\theta(x)) + (1 - R(x)) \cdot sin(\omega_{light} \cdot t + \theta(x))
\end{aligned}$$

$$\hat{P} = \frac{1}{2} \cdot R(x) \cdot cos(\theta(x)) + \frac{R(x)^2 + 1}{4} = \text{Normalized power measured}^* \tag{A.1}$$

From here on we assume R is constant with position seeing as the absolute distance is much larger than the variation in distance, implying R will vary very little.

As per the equation A.1 we obtain a relationship between the distance and the signal measured.

A.1.2 Low Frequency and High Frequency Division

In the setup used, the normalized signal power P is actually split up into a high frequency and low frequency part (20KHz cutoff). We can also arbitrarily split the position x up into its high and low frequencies. We get:

$$\begin{aligned}
x &= x_{LF} + x_{HF} \\
\theta &= \theta_{LF} + \theta_{HF} \\
\hat{P} &= \hat{P}_{LF} + \hat{P}_{HF}
\end{aligned} \tag{A.2}$$

We can combine Equation A.1 with Equations A.2 to obtain the following expressions:

$$\hat{P}_{LF} = \frac{1}{Period\,\theta_{HF}} \int_0^{Period\,\theta_{HF}} -\frac{R\left(x_{LF}\right)}{2} \cdot \ cos\left(\theta_{LF} + \theta_{HF}\left(t\right)\right) + \frac{R\left(x_{LF}\right)^2 + 1}{2}\, dt \tag{A.3}$$

$$\hat{P}_{HF} = \frac{1}{2} \cdot \ R\left(x_{LF}\right) \cdot \ cos\left(\theta_{LF} + \theta_{HF}\left(t\right)\right) + \frac{R\left(x_{LF}\right)^2 + 1}{4} - \hat{P}_{LF} \tag{A.4}$$

Taking the assumption A.5, equation A.3 simplifies to Eq A.6:

$$\theta_{HF}(t) = \theta_{LF,Amp} * sin(\omega t) \tag{A.5}$$

$$\hat{P}_{LF} = -\frac{1}{2} \cdot R \cdot cos\left(\theta_{LF}\right) \cdot J_0\left(\theta_{HF,Amp}\right) + \frac{R\left(x_{LF}\right)^2 + 1}{4} \tag{A.6}$$

Due to simplification later in the report we will assume the validity of assumption A.5.

We can now interpret x_{LF} and θ_{LF} to be any low frequency noise changing the distance, such as expansion due to temperature. The information of interest is in the high frequency distance signal x_{HF}.

A.1.3 Operating Point

The type of interferometry used in this thesis is called absolute phase interferometry. Compared to interferometry with phase wrapping, this is much less costly to implement, and it leads to higher resolution. The downside is that the operating range is limited, but this not a problem for the magnitudes of distance that are relevant in this project. The choice of absolute phase interferometry implies
$k \in \mathbb{Z}\,|0 < \theta_{HF} - k * 2\pi < 2\pi$ must hold at all times.

The sensitivity of the interferometer can be maximized by operating the setup at an optimal θ_{LF}, which can be achieved by slightly changing the wavelength. The optimal θ_{LF} can be found using Equation A.7.

$$argmax_{\theta_{LF}} \frac{\delta abs\left(\frac{\hat{P}_{HF}}{x_{HF}}\right)}{\delta \theta_{LF}} \tag{A.7}$$

$$0 = \frac{\delta abs\left(\frac{\hat{P}_{HF}}{x_{HF}}\right)}{\delta \theta_{LF}}$$

$$\theta_{LF} = \frac{x_{LF} \cdot \ 4\pi}{\lambda} + \frac{\pi}{2} + k \cdot \pi \quad k \in \mathbb{Z} \tag{A.8}$$

It is worth noting that the solution will change signs based on whether k is odd or even. (pi/2 and -p/2 are both valid solutions but sign of slop is negated)

Seeing as x_{HF} is a high frequency signal, its average value is 0, this implies that the maximizing condition for Eq A.7 is:

$$\theta_{LF} = \frac{\pi}{2} + k \cdot \pi \quad k \in \mathbb{Z}$$

$$\theta_{err} = \theta_{LF} - \frac{\pi}{2} + k \cdot \pi \tag{A.9}$$

A.1.4 Signal Reproduction

This leads to the following expressions for the power measured:

$$\hat{P}_{LF} = -\frac{1}{2} \cdot R \cdot sin\left(\theta err\ + k\pi\right) \cdot J_0\left(\theta_{HF,Amp}\right) + \frac{R\left(x_{LF}\right)^2 + 1}{4} \tag{A.10}$$

$$\hat{P}_{HF} = -\frac{1}{2} \cdot R\left(x_{LF}\right) \cdot \ \ sin\left(\theta_{\ err} + \theta_{\ HF} + k \cdot \ \pi\ \right) + \frac{R\left(x_{LF}\right)^2 + 1}{4} - P_{LF} \tag{A.11}$$

We can then rearrange the equations A.10 and A.11 to obtain an expression for θ_{HF} which can be used to obtain x_{HF}:

$$\theta_{HF} = arcsin\left(\frac{-\left(\hat{P}_{HF} + \hat{P}_{LF} - \frac{R(x_{LF})^2+1}{4}\right)}{\frac{1}{2}R\left(x_{LF}\right)}\right) - arcsin\left(-\frac{\left(\hat{P}_{LF} - \frac{R(x_{LF})^2+1}{4}\right)}{\frac{1}{2}\ R\left(x_{LF}\right) \cdot J_0\left(\theta_{HF,Amp}\right)}\right) \tag{A.12}$$

$$x_{HF=}\ \left(\theta_{\ HF} + k \cdot \ \pi\ \right) \cdot \lambda \tag{A.13}$$

A.2 Arcsine Approximation

Implementing an accurate arcsine function on an FPGA that runs at 65 MHz is costly. The system would be easier and cheaper to implement if the following approximation were to be valid:

$$x = arcsin(x) \tag{A.14}$$

This approximation is valid when $|x| << 1$. For this paper, that implies we assume

$$\left|\frac{\hat{P}_{LF} - \frac{R(x_{LF})^2+1}{4}}{\frac{1}{2}\ R\left(x_{LF}\right)}\right| << 1 \tag{A.15}$$

$$\left|\frac{-\hat{P}_{HF}}{\frac{1}{2}R\left(x_{LF}\right)}\right| << 1 \tag{A.16}$$

This in turn implies:

$$\theta_{err} << \frac{\pi}{2} \tag{A.17}$$

$$\theta_{HF} << \frac{\pi}{2} \tag{A.18}$$

B

Source Code

B.1 Control Module [VHDL]

```vhdl
----------------------------------------------------------------------------
-- CONTROL MODULE
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.my_types_pkg.all;

entity control_module is
    port (
        clk                     : in std_logic;
        reset                   : in std_logic;

        -- Inputs from Communication
        new_frequencies         : in custom_fp_array_32_bit(FREQ_DIM-1 downto 0); -- Array of frequencies used
        new_update              : in std_logic;
        new_polynomial_features : in custom_fp_array_2D(FREQ_DIM-1 downto 0, POLY_DIM-1 downto 0);
        new_extra_feature       : in std_logic_vector(FP_SIZE-1 downto 0);
        new_magnitude_weights   : in custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
        new_phase_weights       : in custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
        new_phasor_magnitude    : in custom_fp_array(FREQ_DIM-1 downto 0);
        new_phasor_phase        : in custom_fp_array(FREQ_DIM-1 downto 0);
        new_model_id            : in std_logic_vector(13 downto 0);

        -- Connections to Math Module
        math_start                  : out std_logic;    -- Start pulse to start math, data to math module is valid on this pulse
        math_polynomial_features    : out custom_fp_array(POLY_DIM-1 downto 0);
        math_extra_feature          : out std_logic_vector(FP_SIZE-1 downto 0);
        math_phase_weights          : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
        math_magnitude_weights      : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
        math_phasor_magnitude       : out std_logic_vector(FP_SIZE-1 downto 0);
        math_phasor_phase           : out std_logic_vector(FP_SIZE-1 downto 0);
        math_result_phasor_magnitude : in std_logic_vector(FP_SIZE-1 downto 0);
        math_result_phasor_phase    : in std_logic_vector(FP_SIZE-1 downto 0);
        math_valid                  : in std_logic;

        -- Connections to Time Signal Generation
        gen_frequencies       : out custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
        gen_phasor_magnitudes : out custom_fp_array(FREQ_DIM-1 downto 0);
        gen_phasor_phases     : out custom_fp_array(FREQ_DIM-1 downto 0);

        bin_update                  : out std_logic;
        bin_extra_feature           : out std_logic_vector(FP_SIZE-1 downto 0);
        bin_model_id                : out std_logic_vector(13 downto 0)
    );
end control_module;

architecture Behavioral of control_module is

type math_states is (WAIT_UPDATE, QUEUE_MATH, EXECUTE_MATH, RX_MATH, SAVE_PARAMS);
signal state : math_states;
```

```vhdl

signal bin_size_counter : integer range 0 to (BIN_SIZE+SETTLING_CYCLES-1);

signal current_freq : integer range 0 to (FREQ_DIM-1); -- Which frequency to calculate for

-- Register signals to maintain output
signal reg_math_polynomial_features     : custom_fp_array(POLY_DIM-1 downto 0);
signal reg_math_extra_feature           : std_logic_vector(FP_SIZE-1 downto 0);
signal reg_math_phase_weights           : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
signal reg_math_magnitude_weights       : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
signal reg_math_phasor_magnitude        : std_logic_vector(FP_SIZE-1 downto 0);
signal reg_math_phasor_phase            : std_logic_vector(FP_SIZE-1 downto 0);

signal reg_all_math_polynomial_features : custom_fp_array_2D(FREQ_DIM-1 downto 0, POLY_DIM-1 downto
↪  0);
signal reg_all_math_phasor_magnitude    : custom_fp_array(FREQ_DIM-1 downto 0);
signal reg_all_math_phasor_phase        : custom_fp_array(FREQ_DIM-1 downto 0);

signal reg_gen_frequencies          : custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
signal reg_gen_phasor_magnitudes    : custom_fp_array(FREQ_DIM-1 downto 0);
signal reg_gen_phasor_phases        : custom_fp_array(FREQ_DIM-1 downto 0);
signal reg_bin_extra_feature        : std_logic_vector(FP_SIZE-1 downto 0);
signal reg_bin_model_id             : std_logic_vector(13 downto 0);

signal reg_next_gen_frequencies        : custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
signal reg_next_gen_phasor_magnitudes  : custom_fp_array(FREQ_DIM-1 downto 0);
signal reg_next_gen_phasor_phases      : custom_fp_array(FREQ_DIM-1 downto 0);
signal reg_next_bin_extra_feature      : std_logic_vector(FP_SIZE-1 downto 0);
signal reg_next_bin_model_id           : std_logic_vector(13 downto 0);

begin

process(clk, reset)
begin
    if reset = '1' then
        -- Reset values
        state               <= WAIT_UPDATE;
        bin_size_counter    <= 0;
        current_freq        <= 0;

        math_start          <= '0';
        reg_math_polynomial_features    <= (others => (others => '0'));
        reg_math_extra_feature          <= (others => '0');
        reg_math_phase_weights          <= (others => (others => '0'));
        reg_math_magnitude_weights      <= (others => (others => '0'));
        reg_math_phasor_magnitude       <= (others => '0');
        reg_math_phasor_phase           <= (others => '0');



        reg_gen_frequencies             <= (others => (others => '0'));
        reg_gen_phasor_magnitudes       <= (others => (others => '0'));
        reg_gen_phasor_phases           <= (others => (others => '0'));
        reg_bin_extra_feature           <= (others => '0');
        reg_bin_model_id                <= (others => '0');

        reg_next_gen_frequencies        <= (others => (others => '0'));
        reg_next_gen_phasor_magnitudes  <= (others => (others => '0'));
        reg_next_gen_phasor_phases      <= (others => (others => '0'));
        reg_next_bin_extra_feature      <= (others => '0');
        reg_next_bin_model_id           <= (others => '0');

        gen_frequencies                 <= (others => (others => '0'));
        gen_phasor_magnitudes           <= (others => (others => '0'));
        gen_phasor_phases               <= (others => (others => '0'));
        bin_extra_feature               <= (others => '0');
        bin_model_id                    <= (others => '0');

    elsif rising_edge(clk) then
        -- Default values
        state                           <= state;
        math_start                      <= '0';
        reg_math_polynomial_features    <= reg_math_polynomial_features;
        reg_math_extra_feature          <= reg_math_extra_feature;
        reg_math_phase_weights          <= reg_math_phase_weights;
        reg_math_magnitude_weights      <= reg_math_magnitude_weights;
        reg_math_phasor_magnitude       <= reg_math_phasor_magnitude;
        reg_math_phasor_phase           <= reg_math_phasor_phase;
        current_freq                    <= current_freq;
        reg_gen_frequencies         <= reg_gen_frequencies;
```

```
130         reg_gen_phasor_magnitudes   <= reg_gen_phasor_magnitudes;
131         reg_gen_phasor_phases       <= reg_gen_phasor_phases;
132         reg_bin_extra_feature       <= reg_bin_extra_feature;
133         reg_bin_model_id            <= reg_bin_model_id;
134
135         case state is
136             when WAIT_UPDATE =>
137                 if new_update = '1' then
138                     -- RX latest outputs from Communication
139                     reg_all_math_polynomial_features    <= new_polynomial_features;
140                     reg_math_extra_feature              <= new_extra_feature;
141                     reg_math_phase_weights              <= new_phase_weights;
142                     reg_math_magnitude_weights          <= new_magnitude_weights;
143                     reg_all_math_phasor_magnitude       <= new_phasor_magnitude;
144                     reg_all_math_phasor_phase           <= new_phasor_phase;
145
146                     reg_next_gen_frequencies     <= new_frequencies;
147                     reg_next_bin_extra_feature   <= new_extra_feature;
148                     reg_next_bin_model_id        <= new_model_id;
149                     -- Start math
150                     state <= QUEUE_MATH;
151                 end if;
152             when QUEUE_MATH =>
153                 if bin_size_counter <= SETTLING_CYCLES then
154                     current_freq <= 0; -- Start by updating 0th frequency
155                     state <= EXECUTE_MATH;
156                 else
157                     state <= QUEUE_MATH;
158                 end if;
159             when EXECUTE_MATH =>    -- Send start pulse and params to math module
160                 math_start <= '1';
161                 for i in POLY_DIM-1 downto 0 loop
162                     reg_math_polynomial_features(i) <= reg_all_math_polynomial_features(current_freq,
    ↪  i);
163                 end loop;
164                 reg_math_phasor_magnitude       <= reg_all_math_phasor_magnitude(current_freq);
165                 reg_math_phasor_phase           <= reg_all_math_phasor_phase(current_freq);
166                 state <= RX_MATH;
167             when RX_MATH =>
168                 if math_valid = '1' then
169                     math_start <= '0';
170                     reg_next_gen_phasor_magnitudes(current_freq) <= math_result_phasor_magnitude;
171                     reg_next_gen_phasor_phases(current_freq) <= math_result_phasor_phase;
172                     if current_freq = (FREQ_DIM-1) then
173                         state <= SAVE_PARAMS;
174                     else
175                         current_freq <= current_freq + 1;   -- Increment frequency to update for
176                         state <= EXECUTE_MATH;
177                     end if;
178                 else
179                     math_start <= '1';
180                     state <= RX_MATH;
181                 end if;
182             when SAVE_PARAMS =>
183                 reg_gen_frequencies         <= reg_next_gen_frequencies;
184                 reg_gen_phasor_magnitudes   <= reg_next_gen_phasor_magnitudes;
185                 reg_gen_phasor_phases       <= reg_next_gen_phasor_phases;
186                 reg_bin_extra_feature       <= reg_next_bin_extra_feature;
187                 reg_bin_model_id            <= reg_next_bin_model_id;
188                 state <= WAIT_UPDATE;
189          end case;
190
191         -- Bin counter
192         if bin_size_counter >= BIN_SIZE+SETTLING_CYCLES-1 then
193             bin_update              <= '1';
194             bin_size_counter        <= 0;
195             gen_frequencies         <= reg_next_gen_frequencies;
196             gen_phasor_magnitudes   <= reg_next_gen_phasor_magnitudes;
197             gen_phasor_phases       <= reg_next_gen_phasor_phases;
198             bin_extra_feature       <= reg_next_bin_extra_feature;
199             bin_model_id            <= reg_next_bin_model_id;
200         else
201             bin_update <= '0';
202             bin_size_counter <= bin_size_counter + 1;
203         end if;
204     end if;
205 end process;
206
207 math_polynomial_features    <= reg_math_polynomial_features;
208 math_extra_feature          <= reg_math_extra_feature;
```

```
209 math_phase_weights          <= reg_math_phase_weights;
210 math_magnitude_weights      <= reg_math_magnitude_weights;
211 math_phasor_magnitude       <= reg_math_phasor_magnitude;
212 math_phasor_phase           <= reg_math_phasor_phase;
213
214
215 end Behavioral;
```

B.2 Control Phasor Generation [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- CONTROL PHASOR GENERATION
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.my_types_pkg.all;
use ieee.math_real.all;

entity Control_Phasor_Generation is
port(
    clk : in std_logic;
    reset : in std_logic;
    input_ready : in std_logic;
          System_Phase    : in  std_logic_vector(FP_SIZE-1 downto 0);
        System_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
        input_Phase    : in  std_logic_vector(FP_SIZE-1 downto 0);
        input_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
          Control_Phase    : out  std_logic_vector(FP_SIZE-1 downto 0);
        Control_Gain : out std_logic_vector(FP_SIZE-1 downto 0);
          Control_Phasor_valid : out std_logic);
end Control_Phasor_Generation;

architecture Behavioral of Control_Phasor_Generation is
component fp_subtract_X_bit is
  Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR (FP_SIZE-1 downto 0);
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR (FP_SIZE-1 downto 0);
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR (FP_SIZE-1 downto 0)
  );

end component;

component fp_divider_X_bit is
  Port (
    aclk : in STD_LOGIC;
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 );
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR ( FP_SIZE-1  downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 )
  );

end component;

signal subtract_valid, divide_valid: std_logic;
begin



subtract_phase: fp_subtract_X_bit
    port map (
      -- Global signals
    -- AXI4-Stream slave channel for operand A
      s_axis_a_tvalid         => input_ready,
      s_axis_a_tdata          => input_Phase,
      -- AXI4-Stream slave channel for operand B
      s_axis_b_tvalid         => input_ready,
      s_axis_b_tdata          => System_Phase,
      -- AXI4-Stream master channel for output result
      m_axis_result_tvalid    => subtract_valid,
      m_axis_result_tdata     => Control_Phase
      );

divide_gain: fp_divider_X_bit
    port map (
      -- Global signals
      aclk                    => clk,
    -- AXI4-Stream slave channel for operand A
      s_axis_a_tvalid         => input_ready,
      s_axis_a_tdata          => input_gain,
```

```
77          -- AXI4-Stream slave channel for operand B
78          s_axis_b_tvalid         => input_ready,
79          s_axis_b_tdata          => System_gain,
80          -- AXI4-Stream master channel for output result
81          m_axis_result_tvalid    => divide_valid,
82          m_axis_result_tdata     => Control_gain
83          );
84
85
86            Control_Phasor_valid <= divide_valid and subtract_valid;
87
88  end Behavioral;
89
90
91
```

B.3 Feature Generation [VHDL]

```vhdl
--------------------------------------------------------------------------------
-- FEATURE GENERATION
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.my_types_pkg.all;

entity Feature_Gen is

  port (
        clk : in std_logic;
        reset : in std_logic;
        Generate_Features: in std_logic;
        input_features : in   custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0);
        extra_feature_value : in  std_logic_vector(FP_SIZE-1 downto 0);
        final_features   : out  custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
        Feature_Gen_Done : out std_logic
    );
end Feature_Gen;

architecture behavior of Feature_Gen is

  component vector_scalar_multiplier
    port (
    clk : in std_logic;
    reset : in STD_LOGIC;
    input_valid : in STD_LOGIC;
          input_mult_vect : in custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0); -- partial features
        input_mult1 : in std_logic_vector(FP_SIZE-1 downto 0);
          output_mult : out custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0);
          mult_valid : out std_logic
      );
  end component;

        type state is (idle,start, calc, done);
        signal current_state : state ;
        signal next_state : state;
        signal count, next_count : unsigned(4 downto 0);
        signal input_mult_vect : custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0); -- partial features
        signal input_mult1 : std_logic_vector(FP_SIZE-1 downto 0);                   -- the extra feature
        signal feat_partial, temp_feat_partial : custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0);
    signal output_features_next: custom_fp_array(INPUT_FEATURE_LENGTH*2*ORDER_EXTRA_FEATURE-1 downto 0);
    signal output_features_temp:  custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
    signal mult_valid, input_mult_valid: std_logic;

begin

  update_state: process (clk, reset)
  begin
  if(reset='1' or Generate_Features='0') then
    current_state <= idle;
    count <= "00000";
    output_features_temp<= (others => (others=>'0'));
  else
        if rising_edge(clk) then

                if(current_state /= done) then
                  output_features_temp <= output_features_next(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
                else
                end if;
                count <= next_count;
                current_state <= next_state;

          end if;
  end if;
  end process update_state;


  execute_state: process (reset,current_state,mult_valid)
```

```vhdl
  begin
   case (current_state) is
            when idle =>
                next_state <= start;
                Feature_Gen_Done <= '0';
                input_mult_vect <= input_features;
                input_mult1 <= "0011110000000000";--"00111111100000000000000000000000";
                next_count <= "00000";
                input_mult_valid <= '0';
                       when start =>

                input_mult_vect <= input_features;
                input_mult1 <= "0011110000000000";--"00111111100000000000000000000000";

                Feature_Gen_Done <= '0';
                next_state <= calc;
                input_mult_valid <= '1';
                next_count <= count+1;

                    when calc =>
                        if mult_valid='1' then
                                    next_count <= count + 1;
                        output_features_next(INPUT_FEATURE_LENGTH*(ORDER_EXTRA_FEATURE+1)-1 downto
↪  INPUT_FEATURE_LENGTH) <= output_features_temp;
                        output_features_next(INPUT_FEATURE_LENGTH-1 downto 0) <= feat_partial;
                        temp_feat_partial <= feat_partial;
                        input_mult_vect <= feat_partial; --temp_feat_partial;
                        input_mult_valid <= '0';
                        input_mult1 <= extra_feature_value;

                        if(count<ORDER_EXTRA_FEATURE)then
                            next_state <= calc;
                        else
                            next_state <=done;
                        end if;
                        Feature_Gen_Done <= '0';

                else
                    next_state <= calc;
--                    input_mult_vect <= temp_feat_partial; Leads to error when first coming from
↪  start because input should be input features
--                    input_mult1 <= extra_feature_value;
                    input_mult_valid <= '1';
                end if;


                        when done =>
                             final_features <= output_features_temp;
                             Feature_Gen_Done <= '1';
                        when others =>


                end case;
                end process execute_state;

uu1: vector_scalar_multiplier port map(
    clk => clk,
    reset => reset,
    input_valid=>input_mult_valid,
    input_mult_vect=>input_mult_vect,
        input_mult1=>input_mult1,
          output_mult=>feat_partial,
          mult_valid=>mult_valid
          );

end behavior;
```

B.4 Map Inputs DDS [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- MAP INPUTS DDS
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.my_types_pkg.all;

entity Map_inputs_DDS is
port(
    clk: in std_logic;
    input_Map_inputs_DDS_valid : in std_logic;
          Control_Phase    : in  std_logic_vector(FP_SIZE-1 downto 0);
          Control_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
          DDS_Phase: out  std_logic_vector(31 downto 0);
          DDS_Gain : out std_logic_vector(15 downto 0);
          output_Map_inputs_DDS_valid: out std_logic
          );
end Map_inputs_DDS;

architecture Behavioral of Map_inputs_DDS is

component float_to_fixed_32_bit     Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( 31 downto 0 )
  );
  end component;


  component fp_16_to_32 is
  Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( 15 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( 31 downto 0 )
  );
  end component;

  component floating_point_mult_32_bit is
  Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( 31 downto 0 )
  );
end component;

component fp_mult_16_bit is
  Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( 15 downto 0 );
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR ( 15 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 )
  );
 end component;

component float_to_fixed_32bit_to_16_bit is
  Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 )
  );
end component;

signal Control_Phase_32, Control_Phase_div_2pi_32 : STD_LOGIC_VECTOR ( 31 downto 0 );
signal  Control_Phase_32_valid, Control_Phase_div_2pi_32_valid: std_logic;
signal DDS_phase_valid, DDS_gain_valid, DDS_gain_pad_valid: std_logic;
signal DDS_phase_pad, DDS_gain_pad:  STD_LOGIC_VECTOR ( 31 downto 0 );
begin
```

```vhdl

Control_phase_to32: fp_16_to_32 port map(
    s_axis_a_tvalid => input_Map_inputs_DDS_valid,
    s_axis_a_tdata  => Control_Phase,
    m_axis_result_tvalid  => Control_Phase_32_valid,
    m_axis_result_tdata  => Control_Phase_32
  );

Divide_by_2pi: floating_point_mult_32_bit port map(
    s_axis_a_tvalid => Control_Phase_32_valid ,
    s_axis_a_tdata => Control_Phase_32 ,
    s_axis_b_tvalid =>Control_Phase_32_valid ,
    s_axis_b_tdata => "01001011101000101111100110000011" , -- (2^27 -1)/2pi
    m_axis_result_tvalid => Control_Phase_div_2pi_32_valid,
    m_axis_result_tdata => Control_Phase_div_2pi_32
);
to_fixed_get_DDS_phase: float_to_fixed_32_bit port map(
    s_axis_a_tvalid => Control_Phase_div_2pi_32_valid,
    s_axis_a_tdata  => Control_Phase_div_2pi_32,
    m_axis_result_tvalid  => DDS_phase_valid,
    m_axis_result_tdata=> DDS_phase_pad
  );
  DDS_phase <= DDS_phase_pad;--(27 downto 0);

DDS_GAIN_TO_32_bit : fp_16_to_32 port map(
    s_axis_a_tvalid => input_Map_inputs_DDS_valid,
    s_axis_a_tdata  => Control_gain,
    m_axis_result_tvalid  => DDS_gain_pad_valid,
    m_axis_result_tdata  => DDS_gain_pad
);

to_fixed_get_DDS_gain:
    float_to_fixed_32bit_to_16_bit port map(
    s_axis_a_tvalid => DDS_gain_pad_valid,
    s_axis_a_tdata  => DDS_gain_pad,
    m_axis_result_tvalid  => DDS_gain_valid,
    m_axis_result_tdata  => DDS_gain
  );

output_Map_inputs_DDS_valid <= DDS_gain_valid and DDS_phase_valid;


end Behavioral;
```

B.5 Multiple Time Signal Generation [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- MULTIPLE TIME SIGNAL GENERATION
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.my_types_pkg.all;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Multiple_time_signal_generation is port(
    clk : in std_logic;
          reset : in std_logic;
          input_valid : in std_logic;
          Control_Phase    : in  custom_fp_array(NUM_FREQS-1 downto 0);
          Control_Gain : in custom_fp_array(NUM_FREQS-1 downto 0);
          phase_increase : in custom_fp_array_32_bit(NUM_FREQS-1 downto 0); -- sent by PC

    DAC_IN : out std_logic_vector(15 downto 0)
          );
end Multiple_time_signal_generation;

architecture Behavioral of Multiple_time_signal_generation is
component Time_Signal_Generation is
port(
        clk : in std_logic;
          reset : in std_logic;
          input_valid : in std_logic;
          Control_Phase    : in  std_logic_vector(FP_SIZE-1 downto 0);
          Control_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
          phase_increase : in std_logic_vector(31 downto 0); -- sent by PC

    SINGLE_FREQ_SIG : out std_logic_vector(15 downto 0);
    time_sig_valid: out std_logic
          );end component;

          component adder_16_bit is
  Port (
    A : in STD_LOGIC_VECTOR ( 15 downto 0 );
    B : in STD_LOGIC_VECTOR ( 15 downto 0 );
    S : out STD_LOGIC_VECTOR ( 16 downto 0 )
  );
  end component;

  component adder_17_bit is
  Port (
    A : in STD_LOGIC_VECTOR ( 16 downto 0 );
    B : in STD_LOGIC_VECTOR ( 16 downto 0 );
    S : out STD_LOGIC_VECTOR ( 17 downto 0 )
  );
  end component;

          signal single_freq_sig: custom_fp_array_16_bit(NUM_FREQS-1 downto 0);
          --signal single_freq_sig_VALID: custom_array_1_bit(NUM_FREQS-1 downto 0);
          signal double_freq_sig1: std_logic_vector(16 downto 0);
          signal double_freq_sig2: std_logic_vector(16 downto 0);
          signal quad_freq_sig: std_logic_vector(17 downto 0);
begin

generate_time_sigs: for i in NUM_FREQS-1 downto 0 generate
    add_freq: Time_Signal_Generation port map(
             clk => clk,
             reset => reset,
             input_valid => input_valid,
             Control_Phase => Control_Phase(i),
             Control_Gain => Control_Gain(i),
             phase_increase => phase_increase(i),
             single_freq_sig => single_freq_sig(i)
        );
end generate;

```

```
77  add_freqs1: adder_16_bit port map(
78      A => single_freq_sig(0),
79      B => single_freq_sig(1),
80      S => double_freq_sig1
81  );
82  add_freqs2: adder_16_bit port map(
83      A => single_freq_sig(2),
84      B => "0000000000000000",
85      S => double_freq_sig2
86  );
87
88  add_freqs3: adder_17_bit port map(
89      A => double_freq_sig1,
90      B => double_freq_sig2,
91      S => quad_freq_sig
92  );
93
94  DAC_IN <= quad_freq_sig(17 downto 2);
95
96  end Behavioral;
```

B.6 My Types Package [VHDL]

```vhdl
--------------------------------------------------------------------------------
-- MY TYPES PACKAGE
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

package my_types_pkg is

--When using this still replace all IP for different float size
--when changing FP size make sure to also change MAP_INPUT_DDS
  constant FP_SIZE :integer := 16;

  type array8 is array (natural range <>) of std_logic_vector(7 downto 0);
  type custom_fp_array is array (natural range <>) of std_logic_vector(FP_SIZE-1 downto 0);
  type custom_fp_array_32_bit is array (natural range <>) of std_logic_vector(32-1 downto 0);
  type custom_fp_array_14_bit is array (natural range <>) of std_logic_vector(14-1 downto 0);
  type custom_fp_array_16_bit is array (natural range <>) of std_logic_vector(16-1 downto 0);
  type custom_array_1_bit is array (natural range <>) of std_logic;
  type int_array is array (natural range <>) of integer range -2**15 to 2**15 -1;


  constant NUM_FREQS : integer := 3;
  -- Vector Vector scalar multiplier only works for 64>ORDER_EXTRA_FEATURE*INPUT_FEATURE_LENGTH
  constant ORDER_EXTRA_FEATURE : integer := 5;
  constant INPUT_FEATURE_LENGTH : integer := 10;
  constant VECTOR_WIDTH : integer := 64; --Max value is 64
  constant ADDER_TREE_DEPTH_SCALAR: integer := 6;
  constant INPUT_SIZE_ADDER_TREE: integer := 2**ADDER_TREE_DEPTH_SCALAR;

  -- Nic's types
  constant POLY_DIM : integer := 10;  -- Order of initial feature vector polynomial
  constant EXTRA_DIM : integer := 5;  -- Amount of times to multiply the initial feature vector with
↪  the extra feature
  constant FREQ_DIM : integer := 3;   -- Amount of frequencies used
  constant BIN_SIZE : integer := 1024; -- Size of FFT bin on PC
  constant SETTLING_CYCLES : integer := 50; -- Amount of extra cycles to include in each bin for
↪  settling
  type custom_fp_array_2D is array (natural range <>, natural range <>) of std_logic_vector(FP_SIZE-1
↪  downto 0);

end package;

```

B.7 Phasor Calc Toplevel [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- PHASOR CALCULATION TOPLEVEL
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use work.my_types_pkg.all;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Phasor_Calc_Toplevel is
port(
    clk : in std_logic;
        reset : in std_logic;
        input_Phasor_calc_valid: in std_logic;
        input_features : in   custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0);
        extra_feature_value : in  std_logic_vector(FP_SIZE-1 downto 0);
    weights_gain : in custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
        weights_phase : in custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
    input_Phase   : in  std_logic_vector(FP_SIZE-1 downto 0);
        input_Gain : in std_logic_vector(FP_SIZE-1 downto 0);

        Control_Phase   : out  std_logic_vector(FP_SIZE-1 downto 0);
        Control_Gain : out std_logic_vector(FP_SIZE-1 downto 0);
          Control_Phasor_valid : out std_logic
        );

end Phasor_Calc_Toplevel;

architecture Behavioral of Phasor_Calc_Toplevel is

component Feature_Gen is
  port (
        clk : in std_logic;
        reset : in std_logic;
        Generate_Features: in std_logic;
        input_features : in   custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0);
        extra_feature_value : in  std_logic_vector(FP_SIZE-1 downto 0);
        final_features   : out  custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
        Feature_Gen_Done : out std_logic
    );
end component;

component System_Phasor_Calc is
port(
    clk : in std_logic;
    reset : in std_logic;
    input_ready : in std_logic;
          in_features   : in  custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
        weights_gain : in custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
        weights_phase : in custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
          System_phase : out std_logic_vector(FP_SIZE-1 downto 0);
          System_gain  : out std_logic_vector(FP_SIZE-1 downto 0);
          output_phasorcalc_ready : out std_logic);
end component;

component Control_Phasor_Generation is
port(
    clk : in std_logic;
    reset : in std_logic;
    input_ready : in std_logic;
          System_Phase   : in  std_logic_vector(FP_SIZE-1 downto 0);
        System_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
        input_Phase   : in  std_logic_vector(FP_SIZE-1 downto 0);
        input_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
          Control_Phase   : out  std_logic_vector(FP_SIZE-1 downto 0);
        Control_Gain : out std_logic_vector(FP_SIZE-1 downto 0);
          Control_Phasor_valid : out std_logic);
end component;

```

```
77  signal final_features: custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
78  signal Feature_Gen_Done, output_phasorcalc_ready : std_logic;
79  signal System_gain, System_phase : std_logic_vector(FP_SIZE-1 downto 0);
80  signal sub_reset, sub_reset1, sub_reset2, sub_reset3: std_logic;
81
82  begin
83  Feature_Gen_map: Feature_Gen port map
84                  ( clk                => clk,
85                    reset              => sub_reset,
86                    Generate_Features   => input_Phasor_calc_valid,
87                    input_features     => input_features,
88                    extra_feature_value => extra_feature_value,
89                    final_features     => final_features,
90                    Feature_Gen_Done    => Feature_Gen_Done );
91
92  System_Phasor_calc_map: System_Phasor_Calc port map
93                  ( clk                    => clk,
94                   reset                   => sub_reset,
95                   input_ready             => Feature_Gen_Done,
96                   in_features             => final_features,
97                   weights_gain            => weights_gain,
98                   weights_phase           => weights_phase,
99                   System_phase                   => System_phase,
100                  System_gain                    => System_gain,
101                  output_phasorcalc_ready => output_phasorcalc_ready );
102
103 Control_Phasor_Generation_map: Control_Phasor_Generation port map(
104                     clk => clk,
105                     reset=> sub_reset,
106                     input_ready =>output_phasorcalc_ready,
107                     System_Phase   => System_phase,
108                     System_Gain    => System_gain,
109                     input_Phase   => input_phase,
110                     input_Gain => input_gain,
111                     Control_Phase   => Control_Phase,
112                     Control_Gain => Control_Gain,
113                     Control_Phasor_valid => Control_Phasor_valid
114 );
115
116 create_reset: process (clk)
117     begin
118     if rising_edge(clk) then
119
120             sub_reset1 <= reset or (not input_Phasor_calc_valid);
121             sub_reset2 <= sub_reset1;
122             sub_reset3 <= sub_reset2 ;
123             sub_reset <= sub_reset3 or sub_reset2 or sub_reset1;
124     end if;
125     end process;
126 end Behavioral;
```

B.8 Project Toplevel [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- PROJECT TOPLEVEL
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.my_types_pkg.all;

entity project_toplevel is
  Port (clk                 : in std_logic;
        rst_n               : in std_logic;
        uart_rx             : in std_logic;
        uart_tx             : out std_logic;
        led                 : out std_logic_vector(7 downto 0);
        dac_in              : out std_logic_vector(15 downto 0)
         );
end project_toplevel;

architecture behavioral of project_toplevel is

component uart_communication
      generic (
          baud                : positive := 115200;
          clock_frequency     : positive := 100000000
      );
      port (
          clk                 : in std_logic;
          rst_n               : in std_logic;
          uart_tx             : out std_logic;
          uart_rx             : in std_logic;
          led                 : out std_logic_vector(7 downto 0);
          frequencies         : out custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
          update              : out std_logic;
          polynomial_features : out custom_fp_array_2D(FREQ_DIM-1 downto 0, POLY_DIM-1 downto 0);
          extra_feature       : out std_logic_vector(FP_SIZE-1 downto 0);
          magnitude_weights   : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
          phase_weights       : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
          phasor_magnitude    : out custom_fp_array(FREQ_DIM-1 downto 0);
          phasor_phase        : out custom_fp_array(FREQ_DIM-1 downto 0);
          model_id            : out std_logic_vector(13 downto 0);
          amplitude_estimate  : in std_logic_vector(FP_SIZE-1 downto 0)
      );
  end component;

  component control_module
      port (
          clk                     : in std_logic;
          reset                   : in std_logic;
          new_frequencies         : in custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
          new_update              : in std_logic;
          new_polynomial_features : in custom_fp_array_2D(FREQ_DIM-1 downto 0, POLY_DIM-1 downto 0);
          new_extra_feature       : in std_logic_vector(FP_SIZE-1 downto 0);
          new_magnitude_weights   : in custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
          new_phase_weights       : in custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
          new_phasor_magnitude    : in custom_fp_array(FREQ_DIM-1 downto 0);
          new_phasor_phase        : in custom_fp_array(FREQ_DIM-1 downto 0);
          new_model_id            : in std_logic_vector(13 downto 0);
          math_start                    : out std_logic;
          math_polynomial_features      : out custom_fp_array(POLY_DIM-1 downto 0);
          math_extra_feature            : out std_logic_vector(FP_SIZE-1 downto 0);
          math_phase_weights            : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
          math_magnitude_weights        : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
          math_phasor_magnitude         : out std_logic_vector(FP_SIZE-1 downto 0);
          math_phasor_phase             : out std_logic_vector(FP_SIZE-1 downto 0);
          math_result_phasor_magnitude  : in std_logic_vector(FP_SIZE-1 downto 0);
          math_result_phasor_phase      : in std_logic_vector(FP_SIZE-1 downto 0);
          math_valid                    : in std_logic;
          gen_frequencies               : out custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
          gen_phasor_magnitudes         : out custom_fp_array(FREQ_DIM-1 downto 0);
          gen_phasor_phases             : out custom_fp_array(FREQ_DIM-1 downto 0);
          bin_update                    : out std_logic;
          bin_extra_feature             : out std_logic_vector(FP_SIZE-1 downto 0);
          bin_model_id                  : out std_logic_vector(13 downto 0)
      );
  end component;

  component Multiple_time_signal_generation is
```

```
77          port(
78              clk : in std_logic;
79              reset : in std_logic;
80              input_valid : in std_logic;
81              Control_Phase : in custom_fp_array(NUM_FREQS-1 downto 0);
82              Control_Gain : in custom_fp_array(NUM_FREQS-1 downto 0);
83              phase_increase : in custom_fp_array_32_bit(NUM_FREQS-1 downto 0);
84              DAC_IN : out std_logic_vector(15 downto 0)
85          );
86      end component;
87
88          component Phasor_Calc_Toplevel
89      port(
90            clk : in std_logic;
91            reset : in std_logic;
92            input_Phasor_calc_valid: in std_logic;
93            input_features : in   custom_fp_array((POLY_DIM-1) downto 0);
94            extra_feature_value : in  std_logic_vector(FP_SIZE-1 downto 0);
95            weights_gain : in custom_fp_array(POLY_DIM*EXTRA_DIM-1 downto 0);
96            weights_phase : in custom_fp_array(POLY_DIM*EXTRA_DIM-1 downto 0);
97            input_Phase   : in  std_logic_vector(FP_SIZE-1 downto 0);
98            input_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
99            Control_Phase   : out  std_logic_vector(FP_SIZE-1 downto 0);
100           Control_Gain : out std_logic_vector(FP_SIZE-1 downto 0);
101               Control_Phasor_valid : out std_logic
102             );
103     end component;
104     -- ctrl signals
105
106     signal reset                      : std_logic;
107     signal new_frequencies            : custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
108     signal new_update                 : std_logic;
109     signal new_polynomial_features    : custom_fp_array_2D(FREQ_DIM-1 downto 0, POLY_DIM-1 downto 0);
110     signal new_extra_feature          : std_logic_vector(FP_SIZE-1 downto 0);
111     signal new_magnitude_weights      : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
112     signal new_phase_weights          : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
113     signal new_phasor_magnitude       : custom_fp_array(FREQ_DIM-1 downto 0);
114     signal new_phasor_phase           : custom_fp_array(FREQ_DIM-1 downto 0);
115     signal new_model_id               : std_logic_vector(13 downto 0);
116     signal math_start                    : std_logic;
117     signal math_polynomial_features      : custom_fp_array(POLY_DIM-1 downto 0);
118     signal math_extra_feature            : std_logic_vector(FP_SIZE-1 downto 0);
119     signal math_phase_weights            : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
120     signal math_magnitude_weights        : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
121     signal math_phasor_magnitude         : std_logic_vector(FP_SIZE-1 downto 0);
122     signal math_phasor_phase             : std_logic_vector(FP_SIZE-1 downto 0);
123     signal math_result_phasor_magnitude  : std_logic_vector(FP_SIZE-1 downto 0);
124     signal math_result_phasor_phase      : std_logic_vector(FP_SIZE-1 downto 0);
125     signal math_valid                    : std_logic;
126     signal gen_frequencies               : custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
127     signal gen_phasor_magnitudes         : custom_fp_array(FREQ_DIM-1 downto 0);
128     signal gen_phasor_phases             : custom_fp_array(FREQ_DIM-1 downto 0);
129     signal bin_update                    : std_logic;
130     signal bin_extra_feature             : std_logic_vector(FP_SIZE-1 downto 0);
131     signal bin_model_id                  : std_logic_vector(13 downto 0);
132
133     constant clock_period: time := 10 ns;
134     constant bit_period : time := ((1.0 / real(115200)) * real(1e9)) * 1 ns;
135     signal stop_the_clock : boolean;
136
137     -- comm signals
138     signal transmit_data: std_logic_vector(7 downto 0);
139     signal amplitude_estimate: std_logic_vector(FP_SIZE-1 downto 0);
140
141
142
143 begin
144 reset <= NOT(rst_n);
145
146     -- Insert values for generic parameters !!
147     comm: uart_communication generic map ( baud                 => 115200,
148                                            clock_frequency      => 100000000)
149                              port map ( clk                     => clk,
150                                         rst_n                   => rst_n,
151                                         uart_tx                 => uart_tx,
152                                         uart_rx                 => uart_rx,
153                                         led                     => led,
154                                         frequencies             => new_frequencies,
155                                         update                  => new_update,
156                                         polynomial_features     => new_polynomial_features,
```

```
                                        extra_feature       => new_extra_feature,
                                        magnitude_weights   => new_magnitude_weights,
                                        phase_weights       => new_phase_weights,
                                        phasor_magnitude    => new_phasor_magnitude,
                                        phasor_phase        => new_phasor_phase,
                                        model_id            => new_model_id,
                                        amplitude_estimate  => amplitude_estimate );

   ctrl: control_module port map (clk                          => clk,
                                  reset                        => reset,
                                  new_frequencies              => new_frequencies,
                                  new_update                   => new_update,
                                  new_polynomial_features      => new_polynomial_features,
                                  new_extra_feature            => new_extra_feature,
                                  new_magnitude_weights        => new_magnitude_weights,
                                  new_phase_weights            => new_phase_weights,
                                  new_phasor_magnitude         => new_phasor_magnitude,
                                  new_phasor_phase             => new_phasor_phase,
                                  new_model_id                 => new_model_id,
                                  math_start                   => math_start,
                                  math_polynomial_features     => math_polynomial_features,
                                  math_extra_feature           => math_extra_feature,
                                  math_phase_weights           => math_phase_weights,
                                  math_magnitude_weights       => math_magnitude_weights,
                                  math_phasor_magnitude        => math_phasor_magnitude,
                                  math_phasor_phase            => math_phasor_phase,
                                  math_result_phasor_magnitude => math_result_phasor_magnitude,
                                  math_result_phasor_phase     => math_result_phasor_phase,
                                  math_valid                   => math_valid,
                                  gen_frequencies              => gen_frequencies,
                                  gen_phasor_magnitudes        => gen_phasor_magnitudes,
                                  gen_phasor_phases            => gen_phasor_phases,
                                  bin_update                   => bin_update,
                                  bin_extra_feature            => bin_extra_feature,
                                  bin_model_id                 => bin_model_id);

   siggen: Multiple_time_signal_generation
         port map(
             clk => clk,
             reset => reset,
             input_valid => '1',
             Control_Phase => gen_phasor_phases,
             Control_Gain => gen_phasor_magnitudes,
             phase_increase => gen_frequencies,
             DAC_IN => DAC_IN);

   math: Phasor_Calc_Toplevel port map (
            clk                     => clk,
            reset                   => reset,
            input_Phasor_calc_valid => math_start,
            input_features          => math_polynomial_features,
            extra_feature_value     => math_extra_feature,
            weights_gain            => math_magnitude_weights,
            weights_phase           => math_phase_weights,
            input_Phase             => math_phasor_phase,
            input_Gain              => math_phasor_magnitude,
            Control_Phase           => math_result_phasor_phase,
            Control_Gain            => math_result_phasor_magnitude,
            Control_Phasor_valid    => math_valid);


end behavioral;
```

B.9 System Phasor Calc [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- SYSTEM PHASOR CALCULATION
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.my_types_pkg.all;
use ieee.math_real.all;

entity System_Phasor_Calc is
port(
    clk : in std_logic;
    reset : in std_logic;
    input_ready : in std_logic;
          in_features   : in  custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
         weights_gain : in custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
         weights_phase : in custom_fp_array(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0);
          System_phase : out std_logic_vector(FP_SIZE-1 downto 0);
          System_gain  : out std_logic_vector(FP_SIZE-1 downto 0);
          output_phasorcalc_ready : out std_logic);
end System_Phasor_Calc;

architecture Behavioral of System_Phasor_Calc is

component Vector_Vector_Scalar_multiplier is

    port(
    clk : in std_logic;
    reset : in std_logic;
    input_scalar_mult_valid : in std_logic;
          input_mult_vect_a : in custom_fp_array(VECTOR_WIDTH -1 downto 0);
         input_mult_vect_b : in custom_fp_array(VECTOR_WIDTH -1 downto 0);
          output_scalar_mult: out std_logic_vector(FP_SIZE-1 downto 0);
          output_scalar_mult_valid : out std_logic
      );
end component Vector_Vector_Scalar_multiplier;

signal input_scalar_mult_valid : std_logic;
signal input_mult_vect_a : custom_fp_array(VECTOR_WIDTH -1 downto 0);
signal input_mult_vect_b : custom_fp_array(VECTOR_WIDTH -1 downto 0);
signal output_scalar_mult: std_logic_vector(FP_SIZE-1 downto 0);
signal output_scalar_mult_valid : std_logic;

signal phase_sum, gain_sum: std_logic_vector(FP_SIZE-1 downto 0);
type state is (start, calc_gain,gain_done, calc_phase,phase_done, done);
signal current_state : state ;
signal next_state : state;
signal count : unsigned(4 downto 0);



begin
update_state: process (clk, reset, input_ready)
  begin
  if(reset='1' or input_ready ='0') then
      current_state <= start;

  else
        if rising_edge(clk) then
               current_state <= next_state;
               if(current_state /= phase_done) then
               count <= count + 1;
               end if;
         end if;
  end if;
  end process update_state;


  execute_state: process (reset,current_state,count, in_features, weights_gain, weights_phase,
↪  output_scalar_mult_valid)
  begin
   case (current_state) is
                        when start =>
                            input_mult_vect_a(VECTOR_WIDTH-1 downto
↪  INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE)<= (others => (others =>'0'));
               input_mult_vect_b(VECTOR_WIDTH-1 downto INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE)<=
↪  (others => (others =>'0'));
```

```
75                          count <= (others=>'0');
76                  gain_sum <= (others=>'0');
77                  phase_sum <= (others=>'0');
78                  input_scalar_mult_valid <= '0';
79                  next_state <= calc_gain;
80                  output_phasorcalc_ready <= '0';
81                  System_phase<=(others=>'0');
82                  System_gain<=(others=>'0');
83                      when calc_gain =>
84                           input_mult_vect_a(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto
    ↪  0)<=in_features;
85                           input_mult_vect_b(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto
    ↪  0)<=weights_gain;
86                  input_scalar_mult_valid <= '1';
87                  if(output_scalar_mult_valid = '0')then
88                      next_state <= calc_gain;
89                  else
90                      next_state <=gain_done;
91                  end if;
92                  output_phasorcalc_ready <= '0';
93                  System_phase<=(others=>'0');
94                  System_gain<=(others=>'0');
95              when gain_done =>
96                      gain_sum <= output_scalar_mult;
97                      input_scalar_mult_valid <= '0';
98                  if(output_scalar_mult_valid = '1')then
99                      next_state <= gain_done;
100                 else
101                     next_state <=calc_phase;
102                 end if;
103             when calc_phase =>
104                  input_mult_vect_a(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto 0)<=in_features;
105                      input_mult_vect_b(INPUT_FEATURE_LENGTH*ORDER_EXTRA_FEATURE-1 downto
    ↪  0)<=weights_phase;
106                  input_scalar_mult_valid <= '1';
107
108                 if(output_scalar_mult_valid = '0')then
109                     next_state <= calc_phase;
110                 else
111                     next_state <=phase_done;
112                 end if;
113                 output_phasorcalc_ready <= '0';
114
115
116             when phase_done =>
117                 phase_sum <= output_scalar_mult;
118                 input_scalar_mult_valid <= '1';
119                 next_state <=done;
120                 output_phasorcalc_ready <= '0';
121                 System_phase<=(others=>'0');
122                 System_gain<=(others=>'0');
123             when done =>
124
125                 input_scalar_mult_valid <= '0';
126                 next_state <=done;
127                 output_phasorcalc_ready <= '1';
128                 System_phase<=phase_sum;
129                 System_gain<=gain_sum;
130                         when others =>
131                                 System_phase<=(others=>'0');
132                     System_gain<=(others=>'0');
133
134                 end case;
135                 end process execute_state;
136
137 multiplier_feat_weight: Vector_Vector_Scalar_multiplier
138                 port map(
139     clk =>clk,
140     reset => reset,
141     input_scalar_mult_valid=>input_scalar_mult_valid,
142           input_mult_vect_a=>input_mult_vect_a,
143         input_mult_vect_b=>input_mult_vect_b,
144           output_scalar_mult=>output_scalar_mult,
145           output_scalar_mult_valid=>output_scalar_mult_valid
146       );
147
148 end Behavioral;
149
150
```

B.10 Time Signal Generation [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- TIME SIGNAL GENERATION
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.my_types_pkg.all;

entity Time_Signal_Generation is
port(
    clk : in std_logic;
          reset : in std_logic;
          input_valid : in std_logic;
          Control_Phase    : in  std_logic_vector(FP_SIZE-1 downto 0); --16 bit float that represents phase increase
          Control_Gain : in std_logic_vector(FP_SIZE-1 downto 0); --16 bit float turns into 16 bit signed
          phase_increase : in std_logic_vector(31 downto 0); -- 32 bit unsigned represents phase increase wehre 2^32 is 2*pi

    single_freq_sig : out std_logic_vector(15 downto 0);
    time_sig_valid: out std_logic
          );end Time_Signal_Generation;

architecture Behavioral of Time_Signal_Generation is

component Map_inputs_DDS is
port(
    clk: in std_logic;
    input_Map_inputs_DDS_valid : in std_logic;
          Control_Phase    : in  std_logic_vector(FP_SIZE-1 downto 0);
          Control_Gain : in std_logic_vector(FP_SIZE-1 downto 0);
          DDS_Phase: out  std_logic_vector(31 downto 0);
          DDS_Gain : out std_logic_vector(15 downto 0);
          output_Map_inputs_DDS_valid: out std_logic
          );
end component;

component dds_compiler_0
  Port (
    aclk : in STD_LOGIC;
    s_axis_phase_tvalid : in STD_LOGIC;
    s_axis_phase_tdata : in STD_LOGIC_VECTOR ( 63 downto 0 );
    m_axis_data_tvalid : out STD_LOGIC;
    m_axis_data_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 )
  );
  end component;

  component floating_point_mult_32_bit is
  Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR ( 31 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( 31 downto 0 )
  );
end component;

component fp_mult_16_bit is
  Port (
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( 15 downto 0 );
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR ( 15 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( 15 downto 0 )
  );

end component;

component X_X_Multiplier is
    Port (
    A : in STD_LOGIC_VECTOR ( 15 downto 0 );
    B : in STD_LOGIC_VECTOR ( 15 downto 0 );
    P : out STD_LOGIC_VECTOR ( 15 downto 0 )
  );
```

```
74
75  end component;
76
77  signal config_data : STD_LOGIC_VECTOR ( 64-1 downto 0 );
78  signal DDS_Phase: STD_LOGIC_VECTOR(31 downto 0);
79  signal DDS_Gain: std_logic_vector(15 downto 0);
80  signal output_Map_inputs_DDS_valid, DDS_output_valid, DDS_output_valid_delayed: std_logic;
81  signal single_freq_sig_temp, DDS_output: std_logic_vector(15 downto 0);
82  signal DDS_Phase_padded: std_logic_vector(31 downto 0);
83  signal DDS_output_padded: std_logic_vector(15 downto 0);
84
85  begin
86
87  map_inputs_dds1: Map_inputs_DDS port map(
88      clk => clk,
89      input_Map_inputs_DDS_valid => input_valid,
90            Control_Phase   => Control_Phase,
91            Control_Gain  => Control_Gain,
92            DDS_Phase => DDS_Phase,
93            DDS_Gain   => DDS_Gain,
94            output_Map_inputs_DDS_valid => output_Map_inputs_DDS_valid
95            );
96
97            DDS_Phase_padded <= DDS_Phase;
98            config_data(31 downto 0) <= phase_increase;
99            config_data(63 downto 32) <= DDS_Phase_padded;
100
101 Attach_DDS: dds_compiler_0 Port map (
102     aclk =>clk,
103     s_axis_phase_tvalid =>output_Map_inputs_DDS_valid,
104     s_axis_phase_tdata => config_data,
105     m_axis_data_tvalid => DDS_output_valid,
106     m_axis_data_tdata =>DDS_output_padded
107   );
108
109   DDS_output <= DDS_output_padded(15 downto 0);
110
111 multiply_with_gain: X_X_Multiplier
112   Port map(
113
114     A => DDS_output,
115     B => DDS_gain,
116     P => single_freq_sig_temp
117   );
118
119 DAC_out_validator: process(clk)
120 begin
121     if(rising_edge(clk)) then
122     DDS_output_valid_delayed <= DDS_output_valid;
123     time_sig_valid <=DDS_output_valid_delayed;
124     single_freq_sig <= single_freq_sig_temp;
125     end if;
126 end process;
127
128
129 end Behavioral;
```

B.11 UART Communication [VHDL]

```vhdl
--------------------------------------------------------------------------------
-- UART COMMUNICATION
-- NOT IMPLEMENTED:
    -- Flexible FP_SIZE that isn't a multiple of 8
    -- Timeout to return to WAIT_RX_CMD state (watchdog)?
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.my_types_pkg.all;

entity uart_communication is
    generic (
        baud                : positive := 115200;
        clock_frequency     : positive := 100000000
    );
    port (
        clk                 : in std_logic;
        rst_n               : in std_logic;

        -- Connections to UART pins
        uart_tx             : out std_logic;
        uart_rx             : in std_logic;

        -- Connections to LEDs
        led                 : out std_logic_vector(7 downto 0);

        -- Connections to Control Module
        frequencies         : out custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
        update              : out std_logic;
        polynomial_features : out custom_fp_array_2D(FREQ_DIM-1 downto 0, POLY_DIM-1 downto 0);
        extra_feature       : out std_logic_vector(FP_SIZE-1 downto 0);
        magnitude_weights   : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
        phase_weights       : out custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
        phasor_magnitude    : out custom_fp_array(FREQ_DIM-1 downto 0);
        phasor_phase        : out custom_fp_array(FREQ_DIM-1 downto 0);
        model_id            : out std_logic_vector(13 downto 0);

        -- Connections to Pre-processing Module
        amplitude_estimate  : in std_logic_vector(FP_SIZE-1 downto 0)
    );
end uart_communication;

architecture rtl of uart_communication is
    component uart is
        generic (
            baud            : positive;
            clock_frequency : positive
        );
        port (
            -- general
            clk                 : in std_logic;
            reset               : in std_logic;
            data_stream_in      : in std_logic_vector(7 downto 0);
            data_stream_in_stb  : in std_logic;
            data_stream_in_ack  : out std_logic := '0';
            data_stream_out     : out std_logic_vector(7 downto 0);
            data_stream_out_stb : out std_logic;
            tx                  : out std_logic;
            rx                  : in std_logic
        );
    end component;

    -- UART Signals
    signal tx_data      : std_logic_vector(7 downto 0);
    signal tx_data_stb  : std_logic;
    signal tx_data_ack  : std_logic := '0';
    signal rx_data      : std_logic_vector(7 downto 0);
    signal rx_data_stb  : std_logic;

    -- Control Module Signals
    signal reg_frequencies         : custom_fp_array_32_bit(FREQ_DIM-1 downto 0);
    signal reg_update              : std_logic;
    signal reg_polynomial_features : custom_fp_array_2D(FREQ_DIM-1 downto 0, POLY_DIM-1 downto 0);
    signal reg_extra_feature       : std_logic_vector(FP_SIZE-1 downto 0);
    signal reg_magnitude_weights   : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
```

```vhdl
    signal reg_phase_weights       : custom_fp_array((EXTRA_DIM*POLY_DIM)-1 downto 0);
    signal reg_phasor_magnitude    : custom_fp_array(FREQ_DIM-1 downto 0);
    signal reg_phasor_phase        : custom_fp_array(FREQ_DIM-1 downto 0);
    signal reg_model_id            : std_logic_vector(15 downto 0);

--    signal amplitude_estimate        : std_logic_vector(FP_SIZE-1 downto 0):=
↪  "01001000011001010111100100001010";
    constant FP_DATA_BYTES : integer := (FP_SIZE + 8 - 1) / 8;
    signal count_byte : integer range 0 to 4;
    signal original_count_byte : integer range 0 to 4;
    signal count_index : integer;
    signal original_count_index : integer;
    signal count2_index : integer;
    type param_types is (PARAM_FREQUENCIES, PARAM_POLYNOMIAL_FEATURES, PARAM_EXTRA_FEATURE,
                         PARAM_MAGNITUDE_WEIGHTS, PARAM_PHASE_WEIGHTS, PARAM_PHASOR_MAGNITUDE,
↪  PARAM_PHASOR_PHASE, PARAM_MODEL_ID, NONE);
    signal parameter : param_types;

    type state_types is (WAIT_RX_CMD, RX_PARAM, SEND_UPDATE, TX_AMPLITUDE, RX_LED);
    signal state : state_types;

begin
    -- Connect uart componenet
    uart_component : uart
    generic map(
        baud                => baud,
        clock_frequency     => clock_frequency
    )
    port map(
        -- general
        clk                 => clk,
        reset               => '0',
        data_stream_in      => tx_data,
        data_stream_in_stb  => tx_data_stb,
        data_stream_in_ack  => tx_data_ack,
        data_stream_out     => rx_data,
        data_stream_out_stb => rx_data_stb,
        tx                  => uart_tx,
        rx                  => uart_rx
    );

    receive : process(clk)
    begin
        if rst_n = '0' then
            -- Initial values
            state <= WAIT_RX_CMD;
            reg_frequencies             <= (others => (others => '0'));
            reg_update                  <= '0';
            reg_polynomial_features     <= (others => (others => (others => '0')));
            reg_extra_feature           <= (others => '0');
            reg_magnitude_weights       <= (others => (others => '0'));
            reg_phase_weights           <= (others => (others => '0'));
            reg_phasor_magnitude        <= (others => (others => '0'));
            reg_phasor_phase            <= (others => (others => '0'));
            reg_model_id                <= (others => '0');
            count_index                 <= 0;
            original_count_index        <= 0;
            count2_index                <= 0;
            count_byte                  <= 0;
            parameter                   <= NONE;
            led                         <= (others => '0');

        elsif rising_edge(clk) then
            -- Default values
            state <= state;
            reg_frequencies             <= reg_frequencies;
            reg_update                  <= '0';
            reg_polynomial_features     <= reg_polynomial_features;
            reg_extra_feature           <= reg_extra_feature;
            reg_magnitude_weights       <= reg_magnitude_weights;
            reg_phase_weights           <= reg_phase_weights;
            reg_phasor_magnitude        <= reg_phasor_magnitude;
            reg_phasor_phase            <= reg_phasor_phase;
            reg_model_id                <= reg_model_id;
            count_index                 <= 0;
            original_count_index        <= 0;
            count2_index                <= 0;
            count_byte                  <= 0;
            parameter                   <= NONE;
            case state is
```

```
155                 when WAIT_RX_CMD =>
156                     if rx_data_stb = '1' then   -- If CMD available
157                         count_byte <= FP_DATA_BYTES;
158                         original_count_byte <= FP_DATA_BYTES;
159                         count2_index <= 1;
160                         case rx_data is         -- Execute command
161                             when "01100001" =>  -- CMD: RX_LED                  [ascii: a]
162                                 state <= RX_LED;
163                             when "01100010" =>  -- CMD: TX_AMPLITUDE            [ascii: b]
164                                 state <= TX_AMPLITUDE;
165                             when "01100011" =>  -- CMD: RX frequencies          [ascii: c]
166                                 parameter <= PARAM_FREQUENCIES;
167                                 count_index <= FREQ_DIM;
168                                 original_count_index <= FREQ_DIM;
169                                 count_byte <= 4;
170                                 original_count_byte <= 4;
171                                 state <= RX_PARAM;
172                             when "01100100" =>  -- CMD: RX polynomial_features  [ascii: d]
173                                 parameter <= PARAM_POLYNOMIAL_FEATURES;
174                                 count_index <= POLY_DIM;
175                                 original_count_index <= POLY_DIM;
176                                 count2_index <= FREQ_DIM;
177                                 state <= RX_PARAM;
178                             when "01100101" =>  -- CMD: RX extra_feature        [ascii: e]
179                                 parameter <= PARAM_EXTRA_FEATURE;
180                                 count_index <= 1;
181                                 original_count_index <= 1;
182                                 state <= RX_PARAM;
183                             when "01100110" =>  -- CMD: RX magnitude_weights    [ascii: f]
184                                 parameter <= PARAM_MAGNITUDE_WEIGHTS;
185                                 count_index <= EXTRA_DIM*POLY_DIM;
186                                 original_count_index <= EXTRA_DIM*POLY_DIM;
187                                 state <= RX_PARAM;
188                             when "01100111" =>  -- CMD: RX phase_weights        [ascii: g]
189                                 parameter <= PARAM_PHASE_WEIGHTS;
190                                 count_index <= EXTRA_DIM*POLY_DIM;
191                                 original_count_index <= EXTRA_DIM*POLY_DIM;
192                                 state <= RX_PARAM;
193                             when "01101000" =>  -- CMD: RX phasor_magnitude     [ascii: h]
194                                 parameter <= PARAM_PHASOR_MAGNITUDE;
195                                 count_index <= FREQ_DIM;
196                                 original_count_index <= FREQ_DIM;
197                                 state <= RX_PARAM;
198                             when "01101001" =>  -- CMD: RX phasor_phase         [ascii: i]
199                                 parameter <= PARAM_PHASOR_PHASE;
200                                 count_index <= FREQ_DIM;
201                                 original_count_index <= FREQ_DIM;
202                                 state <= RX_PARAM;
203                             when "01101010" =>  -- CMD: RX model_id             [ascii: j]
204                                 parameter <= PARAM_MODEL_ID;
205                                 count_index <= 1;
206                                 original_count_index <= 1;
207                                 count_byte <= 2;
208                                 original_count_byte <= 2;
209                                 state <= RX_PARAM;
210                             when "01101011" =>  -- CMD: Update Model            [ascii: k]
211                                 state <= SEND_UPDATE;
212                             when others =>      -- Else keep for another CMD
213                                 state <= WAIT_RX_CMD;
214                         end case;
215                     else
216                         state <= WAIT_RX_CMD;
217                     end if;
218 
219                 when RX_PARAM =>
220                     count_index             <= count_index;
221                     original_count_index    <= original_count_index;
222                     count2_index            <= count2_index;
223                     count_byte              <= count_byte;
224                     original_count_byte     <= original_count_byte;
225                     parameter               <= parameter;
226                     if count2_index > 0 then
227                         if count_index > 0 then
228                             if count_byte > 0 then
229                                 if rx_data_stb = '1' then
230                                     case parameter is
231                                         when PARAM_FREQUENCIES =>
232                                             reg_frequencies(count_index-1)(((count_byte*8)-1) downto
    ↪  ((count_byte-1)*8)) <= rx_data;
233                                         when PARAM_POLYNOMIAL_FEATURES =>
```

```
234                                         reg_polynomial_features(count2_index-1,
    ↪  count_index-1)(((count_byte*8)-1) downto ((count_byte-1)*8)) <= rx_data;
235                                     when PARAM_EXTRA_FEATURE =>
236                                         reg_extra_feature(((count_byte*8)-1) downto
    ↪  ((count_byte-1)*8)) <= rx_data;
237                                     when PARAM_MAGNITUDE_WEIGHTS =>
238                                         reg_magnitude_weights(count_index-1)(((count_byte*8)-1)
    ↪  downto ((count_byte-1)*8)) <= rx_data;
239                                     when PARAM_PHASE_WEIGHTS =>
240                                         reg_phase_weights(count_index-1)(((count_byte*8)-1)
    ↪  downto ((count_byte-1)*8)) <= rx_data;
241                                     when PARAM_PHASOR_MAGNITUDE =>
242                                         reg_phasor_magnitude(count_index-1)(((count_byte*8)-1)
    ↪  downto ((count_byte-1)*8)) <= rx_data;
243                                     when PARAM_PHASOR_PHASE =>
244                                         reg_phasor_phase(count_index-1)(((count_byte*8)-1) downto
    ↪  ((count_byte-1)*8)) <= rx_data;
245                                     when PARAM_MODEL_ID =>
246                                         reg_model_id(((count_byte*8)-1) downto
    ↪  ((count_byte-1)*8)) <= rx_data;
247                                     when NONE =>
248                                         state <= WAIT_RX_CMD;
249                                 end case;
250                                 count_byte <= count_byte - 1;
251                                 state <= RX_PARAM;
252                             end if;
253                         else
254                             count_index <= count_index - 1;
255                             count_byte <= original_count_byte;
256                         end if;
257                     else
258                         count2_index <= count2_index - 1;
259                         count_index <= original_count_index;
260                         count_byte <= original_count_byte;
261                     end if;
262                 else
263                     state <= WAIT_RX_CMD;
264                 end if;
265
266             when SEND_UPDATE =>      -- Send signal to Control Module to update model parameters
267                 reg_update <= '1';
268                 state <= WAIT_RX_CMD;
269
270             when TX_AMPLITUDE =>     -- Send amplitude estimation over UART
271             if count_byte > 0 then
272                 tx_data_stb <= '1'; -- Request TX
273                 tx_data <= amplitude_estimate(((count_byte*8) -1) downto ((count_byte-1)*8)); --
    ↪  Send sections of data 1 byte at a time
274                 if tx_data_ack = '1' then    -- If TX ack received
275                     count_byte <= count_byte - 1;
276                     tx_data_stb <= '0';
277                 end if;
278                 state <= TX_AMPLITUDE;
279             else
280                 state <= WAIT_RX_CMD;
281             end if;
282
283             when RX_LED =>
284                 if rx_data_stb = '1' then    -- If value available
285                     led <= rx_data;          -- Set LEDs
286                     state <= WAIT_RX_CMD;
287                 else
288                     state <= RX_LED;
289                 end if;
290
291         end case;
292     end if;
293 end process;
294
295 frequencies         <= reg_frequencies;
296 update              <= reg_update;
297 polynomial_features <= reg_polynomial_features;
298 extra_feature       <= reg_extra_feature;
299 magnitude_weights   <= reg_magnitude_weights;
300 phase_weights       <= reg_phase_weights;
301 phasor_magnitude    <= reg_phasor_magnitude;
302 phasor_phase        <= reg_phasor_phase;
303 model_id            <= reg_model_id(13 downto 0);
304 end rtl;
305
```

B.12 USB Communication

```
--------------------------------------------------------------------------------
-- USB COMMUNICATION
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_Std.all;

entity usb_communication is
    port (  clk         : in std_logic;      -- System clk
            rst_n       : in std_logic;      -- System rst
            usb_data    : in std_logic_vector(15 downto 0);      -- Data sent to output fifo
            usb_write   : in std_logic;      -- Valid signal for input data
            usb_full    : out std_logic;     -- Buffer is full, no data can be written
            ft_clk      : in std_logic;      -- FT600 clk
            ft_data     : out std_logic_vector(15 downto 0);     -- Data sent to FT600
            ft_wr_n     : out std_logic;     -- FT600 write flag (1 = inactive, 0 = write)
            ft_rd_n     : out std_logic;     -- FT600 read flag (1 = inactive, 0 = read)
            ft_be       : out std_logic_vector(1 downto 0);      -- FT600 byte enable (1 = valid)
            ft_oe_n     : out std_logic;     -- FT600 output enable (1 = FPGA outputs data, 0 = FT600
↪  outputs data)
            ft_txe_n    : in std_logic);    -- FT600 Transmit FIFO Empty (1 = FIFO full, 0 = Space
↪  available)

end usb_communication;

architecture Behavioral of usb_communication is
    component fifo_generator_1 is
        port (  rst     : IN STD_LOGIC;
                wr_clk  : IN STD_LOGIC;
                rd_clk  : IN STD_LOGIC;
                din     : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
                wr_en   : IN STD_LOGIC;
                rd_en   : IN STD_LOGIC;
                dout    : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);
                full    : OUT STD_LOGIC;
                empty   : OUT STD_LOGIC;
                valid   : OUT STD_LOGIC);
    end component;

    signal full     : std_logic;
    signal empty    : std_logic;
    signal rst      : std_logic;
    signal en_count : integer range 0 to 11 := 0;
    signal EN       : std_logic := '0';
    signal rd_en    : std_logic := '0';
    signal wr_en    : std_logic := '0';

    signal valid : std_logic;

    signal was_full : std_logic := '0';
    signal ft_data_int  : std_logic_vector(15 downto 0);
    signal ft_data_int_reg  : std_logic_vector(15 downto 0);
    signal ft_data_reg  : std_logic_vector(15 downto 0);
begin
    rst <= NOT(rst_n);
    ft_fifo: fifo_generator_1
        port map(   rst     => rst,
                    wr_clk  => clk,
                    rd_clk  => ft_clk,
                    din     => usb_data,
                    wr_en   => wr_en,
                    rd_en   => rd_en,
                    dout    => ft_data_int,
                    full    => full,
                    empty   => empty,
                    valid   => valid);
    wr_en   <= EN and usb_write and not(full);
    usb_full <= full;

    ft_rd_n <= '1';
    ft_be   <= "11";
    ft_oe_n <= '1';

    rd_en <= EN and NOT(ft_txe_n) and not(was_full);
    ft_wr_n <= not(EN and not(empty) and NOT(ft_txe_n));

```

```
75      process(ft_clk)
76      begin
77          -- The FIFO is picky and needs a 5 cycle reset followed by 5 cycles of not using the FIFO
78          if rising_edge(ft_clk) then
79              ft_data_reg <= ft_data_reg;
80              ft_data_int_reg <= ft_data_int;
81              if rst_n = '0' then
82                  en_count <= 0;
83                  EN <= '0';
84              elsif en_count > 10 then
85                  EN <= '1';
86              else
87                  en_count <= en_count + 1;
88                  EN <= EN;
89              end if;
90
91              if ft_txe_n = '0' then
92                  if was_full = '1' then
93                      ft_data <= ft_data_reg;
94                      was_full <= '0';
95                  else
96                      ft_data <= ft_data_int;
97                  end if;
98              else
99                  if was_full = '0' then
100                     ft_data_reg <= ft_data_int_reg;
101                 end if;
102                 was_full <= '1';
103             end if;
104         end if;
105     end process;
106
107 end Behavioral;
```

B.13 Vector Scalar Multiplier [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- VECTOR SCALAR MULTIPLIER
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.my_types_pkg.all;
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity vector_scalar_multiplier is
    generic(
        VECTOR_WIDTH : integer := INPUT_FEATURE_LENGTH);
    port(
    clk : in std_logic;
    reset: in std_logic;
    input_valid : in STD_LOGIC;
          input_mult_vect : in custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0); -- partial
↪  features
        input_mult1 : in std_logic_vector(FP_SIZE-1 downto 0);
          output_mult : out custom_fp_array((INPUT_FEATURE_LENGTH-1) downto 0);
          mult_valid : out std_logic
      );
end vector_scalar_multiplier;

architecture Behavioral of vector_scalar_multiplier is
component fp_mult_16_bit
    Port (
    aclk : in STD_LOGIC;
    aresetn : in STD_LOGIC;
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 );
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 )
  );
  end component;
  signal valid :  std_logic_vector((INPUT_FEATURE_LENGTH-1) downto 0);
  signal reset_mult_n: std_logic;

begin

reset_mult_n <= not reset;

gen_multipliers: for i in 0 to VECTOR_WIDTH-1 generate
  mult :  fp_mult_16_bit port map(
        aclk => clk,
        aresetn => reset_mult_n,
        s_axis_a_tvalid =>input_valid,
        s_axis_a_tdata =>input_mult_vect(i),
        s_axis_b_tvalid => input_valid,
        s_axis_b_tdata =>input_mult1,
        m_axis_result_tvalid =>valid(i),
        m_axis_result_tdata =>output_mult(i)
);

end generate gen_multipliers;


process(valid)
variable temp_valid:std_logic;
begin
    temp_valid := '1';
    for i in 0 to VECTOR_WIDTH-1 loop
        temp_valid := temp_valid and valid(i);
    end loop;
    mult_valid <= temp_valid;
end process;

end Behavioral;
```

B.14 Vector Vector Scalar Multiplier [VHDL]

```vhdl
----------------------------------------------------------------------------------
-- VECTOR VECTOR SCALAR MULTIPLIER
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.math_real.all;
use work.my_types_pkg.all;


-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Vector_Vector_Scalar_multiplier is
    port(
    clk : in std_logic;
    reset: in std_logic;
    input_scalar_mult_valid : in std_logic;
          input_mult_vect_a : in custom_fp_array(VECTOR_WIDTH -1 downto 0);
        input_mult_vect_b : in custom_fp_array(VECTOR_WIDTH -1 downto 0);
          output_scalar_mult: out std_logic_vector(FP_SIZE-1 downto 0);
          output_scalar_mult_valid : out std_logic
      );
end Vector_Vector_Scalar_multiplier;

architecture Behavioral of Vector_Vector_Scalar_multiplier is

signal output_mult : custom_fp_array(VECTOR_WIDTH -1 downto 0);
signal intermediate_sums : custom_fp_array(VECTOR_WIDTH*2 -2 downto 0);
signal intermediate_valid: std_logic_vector(VECTOR_WIDTH*2 -2 downto  0);
signal aresetn: std_logic;

component fp_mult_16_bit
    Port (
    aclk: in std_logic;
    aresetn : in STD_LOGIC;
    s_axis_a_tvalid : in STD_LOGIC;
    s_axis_a_tdata : in STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 );
    s_axis_b_tvalid : in STD_LOGIC;
    s_axis_b_tdata : in STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 );
    m_axis_result_tvalid : out STD_LOGIC;
    m_axis_result_tdata : out STD_LOGIC_VECTOR ( FP_SIZE-1 downto 0 )
  );
  end component;

COMPONENT fp_adder_16_bit
  PORT (
    aclk: IN STD_LOGIC;
    aresetn : in STD_LOGIC;
    s_axis_a_tvalid : IN STD_LOGIC;
    s_axis_a_tdata : IN STD_LOGIC_VECTOR(FP_SIZE-1 DOWNTO 0);
    s_axis_b_tvalid : IN STD_LOGIC;
    s_axis_b_tdata : IN STD_LOGIC_VECTOR(FP_SIZE-1 DOWNTO 0);
    m_axis_result_tvalid : OUT STD_LOGIC;
    m_axis_result_tdata : OUT STD_LOGIC_VECTOR(FP_SIZE-1 DOWNTO 0)
  );
  end component;

begin

aresetn <= (not reset);

gen_multipliers: for i in 0 to VECTOR_WIDTH -1 generate

  mult :  fp_mult_16_bit port map(
        aclk => clk,
        aresetn => aresetn,
        s_axis_a_tvalid =>input_scalar_mult_valid,
        s_axis_a_tdata =>input_mult_vect_a(i),
        s_axis_b_tvalid => input_scalar_mult_valid,
        s_axis_b_tdata =>input_mult_vect_b(i),
```

```
77          m_axis_result_tvalid =>intermediate_valid(i),
78          m_axis_result_tdata =>output_mult(i)
79  );
80  end generate gen_multipliers;
81
82  intermediate_sums(VECTOR_WIDTH -1 downto 0) <= output_mult;
83
84
85
86      gen_adders6:for k in 0 to (2**(ADDER_TREE_DEPTH_SCALAR-1))-1 generate
87          begin
88              adder: fp_adder_16_bit port map(
89              aclk => clk,
90              aresetn => aresetn,
91              s_axis_a_tvalid => intermediate_valid(2*k),
92              s_axis_a_tdata => intermediate_sums(2*k),
93              s_axis_b_tvalid => intermediate_valid(2*k+1),
94              s_axis_b_tdata => intermediate_sums(2*k+1),
95              m_axis_result_tvalid => intermediate_valid(k+2**ADDER_TREE_DEPTH_SCALAR),
96              m_axis_result_tdata => intermediate_sums(k+2**ADDER_TREE_DEPTH_SCALAR)
97                  );
98      end generate;
99
100     gen_adders5:for k in 0 to (2**(ADDER_TREE_DEPTH_SCALAR-2))-1 generate
101         begin
102
103             adder: fp_adder_16_bit port map(
104             aclk => clk,
105             aresetn => aresetn,
106             s_axis_a_tvalid => intermediate_valid(2*k+2**ADDER_TREE_DEPTH_SCALAR),
107             s_axis_a_tdata => intermediate_sums(2*k+2**ADDER_TREE_DEPTH_SCALAR),
108             s_axis_b_tvalid => intermediate_valid(2*k+1+2**ADDER_TREE_DEPTH_SCALAR),
109             s_axis_b_tdata => intermediate_sums(2*k+1+2**ADDER_TREE_DEPTH_SCALAR),
110             m_axis_result_tvalid =>
    ↪  intermediate_valid(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)),
111             m_axis_result_tdata =>
    ↪  intermediate_sums(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1))
112                 );
113     end generate;
114     gen_adders4:for k in 0 to (2**(ADDER_TREE_DEPTH_SCALAR-3))-1 generate
115         begin
116
117             adder: fp_adder_16_bit port map(
118             aclk => clk,
119             aresetn => aresetn,
120             s_axis_a_tvalid =>
    ↪  intermediate_valid(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)),
121             s_axis_a_tdata =>
    ↪  intermediate_sums(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)),
122             s_axis_b_tvalid =>
    ↪  intermediate_valid(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)),
123             s_axis_b_tdata =>
    ↪  intermediate_sums(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)),
124             m_axis_result_tvalid =>
    ↪  intermediate_valid(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)),
125             m_axis_result_tdata =>
    ↪  intermediate_sums(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2))
126                 );
127     end generate;
128     gen_adders3:for k in 0 to (2**(ADDER_TREE_DEPTH_SCALAR-4))-1 generate
129         begin
130
131             adder: fp_adder_16_bit port map(
132             aclk => clk,
133             aresetn => aresetn,
134             s_axis_a_tvalid =>
    ↪  intermediate_valid(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)),
135             s_axis_a_tdata =>
    ↪  intermediate_sums(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)),
136             s_axis_b_tvalid =>
    ↪  intermediate_valid(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)),
137             s_axis_b_tdata =>
    ↪  intermediate_sums(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)),
138             m_axis_result_tvalid =>
    ↪  intermediate_valid(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(ADD
139             m_axis_result_tdata =>
    ↪  intermediate_sums(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(ADDE
140                 );
141     end generate;
142
```

```
143     gen_adders2:for k in 0 to (2**(ADDER_TREE_DEPTH_SCALAR-5))-1 generate
144         begin
145 
146             adder: fp_adder_16_bit port map(
147             aclk => clk,
148             aresetn => aresetn,
149             s_axis_a_tvalid =>
    ↪  intermediate_valid(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(A
150             s_axis_a_tdata =>
    ↪  intermediate_sums(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(AD
151             s_axis_b_tvalid =>
    ↪  intermediate_valid(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**
152             s_axis_b_tdata =>
    ↪  intermediate_sums(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(
153             m_axis_result_tvalid =>
    ↪  intermediate_valid(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(ADD
154             m_axis_result_tdata =>
    ↪  intermediate_sums(k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(ADDE
155                 );
156     end generate;
157     gen_adders1:for k in 0 to (2**(ADDER_TREE_DEPTH_SCALAR-6))-1 generate
158         begin
159 
160             adder: fp_adder_16_bit port map(
161             aclk => clk,
162             aresetn => aresetn,
163             s_axis_a_tvalid =>
    ↪  intermediate_valid(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(A
164             s_axis_a_tdata =>
    ↪  intermediate_sums(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(AD
165             s_axis_b_tvalid
    ↪  =>intermediate_valid(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2
166             s_axis_b_tdata =>
    ↪  intermediate_sums(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(
167             m_axis_result_tvalid =>
    ↪  intermediate_valid(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(A
168             m_axis_result_tdata =>
    ↪  intermediate_sums(2*k+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(AD
169                 );
170     end generate;
171 
172     output_scalar_mult <= intermediate_sums(2*VECTOR_WIDTH-2);
173     output_scalar_mult_valid <= intermediate_valid(2*VECTOR_WIDTH-2);
174 --    gen_adders0:for k in 0 to (2**(ADDER_TREE_DEPTH_SCALAR-7))-1 generate
175 --        begin
176 
177 --            adder: fp_adder_16_bit port map(
178 --            aclk => clk,
179 --            aresetn => aresetn,
180 --            s_axis_a_tvalid => '1',
181 --            s_axis_a_tdata =>
    ↪  intermediate_sums(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(
182 --            s_axis_b_tvalid => '1',
183 --            s_axis_b_tdata =>
    ↪  intermediate_sums(2*k+1+2**ADDER_TREE_DEPTH_SCALAR+2**(ADDER_TREE_DEPTH_SCALAR-1)+2**(ADDER_TREE_DEPTH_SCALAR-2)+2**(
184 --            m_axis_result_tvalid => open,
185 --            m_axis_result_tdata => output_scalar_mult
186 --                );
187 --    end generate;
188 
189 
190 end Behavioral;
```

B.15 UART Simulation Creator [Python]

```python
from math import ceil
import struct

import numpy as np

FP_SIZE = 16
FP_DATA_BYTES = ceil(FP_SIZE/8)
POLY_DIM = 10
EXTRA_DIM = 5
FREQ_DIM = 3
cmd = { "set_led": 0b01100001,
        "request_amplitude": 0b01100010,
        "param_frequencies": 0b01100011,
        "param_polynomial_features": 0b01100100,
        "param_extra_feature": 0b01100101,
        "param_magnitude_weights": 0b01100110,
        "param_phase_weights": 0b01100111,
        "param_phasor_magnitude": 0b01101000,
        "param_phasor_phase": 0b01101001,
        "param_model_id": 0b01101010
        }

frequencies = [0x180002AA, 0x300002AA, 0x100002AA]                  # 32 bit unsigned (Phase increase per clk cycle where 2^32 is 2*pi increase)
polynomial_features = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                       [2, 2, 2, 2, 2, 2, 2, 2, 2, 2],
                       [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]]     # FP_SIZE bit float
extra_feature = [1]                                                 # FP_SIZE bit float
magnitude_weights = [3*i for i in range(0, POLY_DIM*EXTRA_DIM)]       # FP_SIZE bit float
phase_weights = [1*i for i in range(0, POLY_DIM*EXTRA_DIM)]          # FP_SIZE bit float
phasor_magnitude = [1000, 4000, 13000]                               # FP_SIZE bit float (Value between -2^15 to 2^15 -1)
phasor_phase = [2, 1, 1.5]                                      # FP_SIZE bit float (Value between 0 to 2*pi)
model_id = [1248]                                                   # 14 bit unsigned (sent as 16 bits)


def flatten(list_of_lists):
    if len(list_of_lists) == 0:
        return list_of_lists
    if isinstance(list_of_lists[0], list):
        return flatten(list_of_lists[0]) + flatten(list_of_lists[1:])
    return list_of_lists[:1] + flatten(list_of_lists[1:])


def std_logic_vector(value, size):
    assert value < 2 ** size
    return '"' + format(value, '0'+str(size)+'b') + '"'


def to_bytes(data, byte_count=FP_DATA_BYTES):
    fmt = { 1: "B",
            2: 'H',
            4: 'I',
            8: 'Q'}
    assert byte_count in fmt.keys()
    return list(struct.unpack(str(byte_count)+'B', struct.pack('>'+fmt[byte_count], data)))


def make_cmd_array(command, data_list, data_byte_count=FP_DATA_BYTES):
    byte_list = flatten([to_bytes(i, data_byte_count) for i in data_list])
    return "({}, {})".format(std_logic_vector(command, 8), ", ".join([std_logic_vector(i, 8) for i in byte_list])), len(byte_list)+1


def float_to_hex(value, size):
    if size == 16:
        return np.float16(value).view(np.int16)
    elif size == 32:
        return np.float32(value).view(np.int32)
    else:
        raise Exception("parameter size not 16 or 32 : " + str(size))


def print_tb_arrays():
```

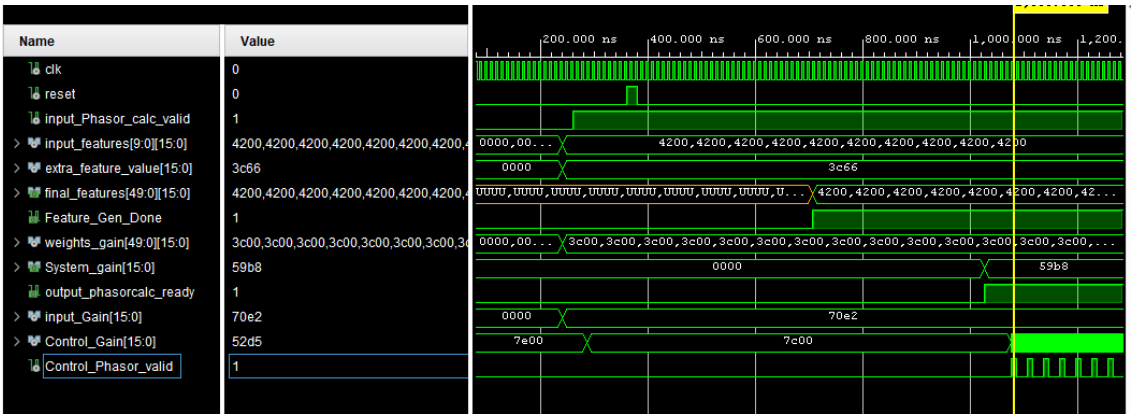
```
    byte_count = 0
    print("type variable_array is array (natural range <>) of std_logic_vector(7 downto 0);")

    # Send param_frequencies
    cmds, length = make_cmd_array(cmd["param_frequencies"], flatten(frequencies), data_byte_count=4)
    byte_count += length
    print("-- frequencies =", str(frequencies))
    print("constant frequencies_cmds: variable_array(FREQ_DIM*" + str(4) + " downto 0) :=", cmds,
    ↪  ";")

    # Send param_polynomial_features
    raw_data = [float_to_hex(a, FP_SIZE) for a in flatten(polynomial_features)]
    cmds, length = make_cmd_array(cmd["param_polynomial_features"], raw_data)
    byte_count += length
    print("-- polynomial_features =", str(polynomial_features))
    print("constant polynomial_features_cmds: variable_array(FREQ_DIM*POLY_DIM*" + str(FP_DATA_BYTES)
    ↪  + " downto 0) :=", cmds, ";")

    # Send param_extra_feature
    raw_data = [float_to_hex(a, FP_SIZE) for a in flatten(extra_feature)]
    cmds, length = make_cmd_array(cmd["param_extra_feature"], raw_data)
    byte_count += length
    print("-- extra_feature =", str(extra_feature))
    print("constant extra_feature_cmds: variable_array(" + str(FP_DATA_BYTES) + " downto 0) :=",
    ↪  cmds, ";")

    # Send param_magnitude_weights
    raw_data = [float_to_hex(a, FP_SIZE) for a in flatten(magnitude_weights)]
    cmds, length = make_cmd_array(cmd["param_magnitude_weights"], raw_data)
    byte_count += length
    print("-- magnitude_weights =", str(magnitude_weights))
    print("constant magnitude_weights_cmds: variable_array(EXTRA_DIM*POLY_DIM*" + str(FP_DATA_BYTES)
    ↪  + " downto 0) :=", cmds, ";")

    # Send param_phase_weights
    raw_data = [float_to_hex(a, FP_SIZE) for a in flatten(phase_weights)]
    cmds, length = make_cmd_array(cmd["param_phase_weights"], raw_data)
    byte_count += length
    print("-- phase_weights =", str(phase_weights))
    print("constant phase_weights_cmds: variable_array(EXTRA_DIM*POLY_DIM*" + str(FP_DATA_BYTES) + "
    ↪  downto 0) :=", cmds, ";")

    # Send param_phasor_magnitude
    raw_data = [float_to_hex(a, FP_SIZE) for a in flatten(phasor_magnitude)]
    cmds, length = make_cmd_array(cmd["param_phasor_magnitude"], raw_data)
    byte_count += length
    print("-- phasor_magnitude =", str(phasor_magnitude))
    print("constant phasor_magnitude_cmds: variable_array(FREQ_DIM*" + str(FP_DATA_BYTES) + " downto
    ↪  0) :=", cmds, ";")

    # Send param_phasor_phase
    raw_data = [float_to_hex(a, FP_SIZE) for a in flatten(phasor_phase)]
    cmds, length = make_cmd_array(cmd["param_phasor_phase"], raw_data)
    byte_count += length
    print("-- phasor_phase =", str(phasor_phase))
    print("constant phasor_phase_cmds: variable_array(FREQ_DIM*" + str(FP_DATA_BYTES) + " downto 0)
    ↪  :=", cmds, ";")

    # Send param_model_id
    cmds, length = make_cmd_array(cmd["param_model_id"], flatten(model_id), data_byte_count=2)
    byte_count += length
    print("-- model_id =", str(model_id))
    print("constant model_id_cmds: variable_array(2 downto 0) :=", cmds, ";")

    return byte_count


if __name__ == "__main__":
    import io
    from contextlib import redirect_stdout
    import subprocess
    import platform

    f = io.StringIO()
    with redirect_stdout(f):
        byte_count = print_tb_arrays()
    out = f.getvalue()

    print("Size of command stream:", byte_count, "bytes")
    if platform.system() == 'Windows':
```

```
145             subprocess.run("clip", text=True, input=out)
146             print("[Windows] Success: Copied output to clipboard")
147         else:
148             subprocess.run("pbcopy", text=True, input=out)
149             print("Success: Copied output to clipboard")
150
```

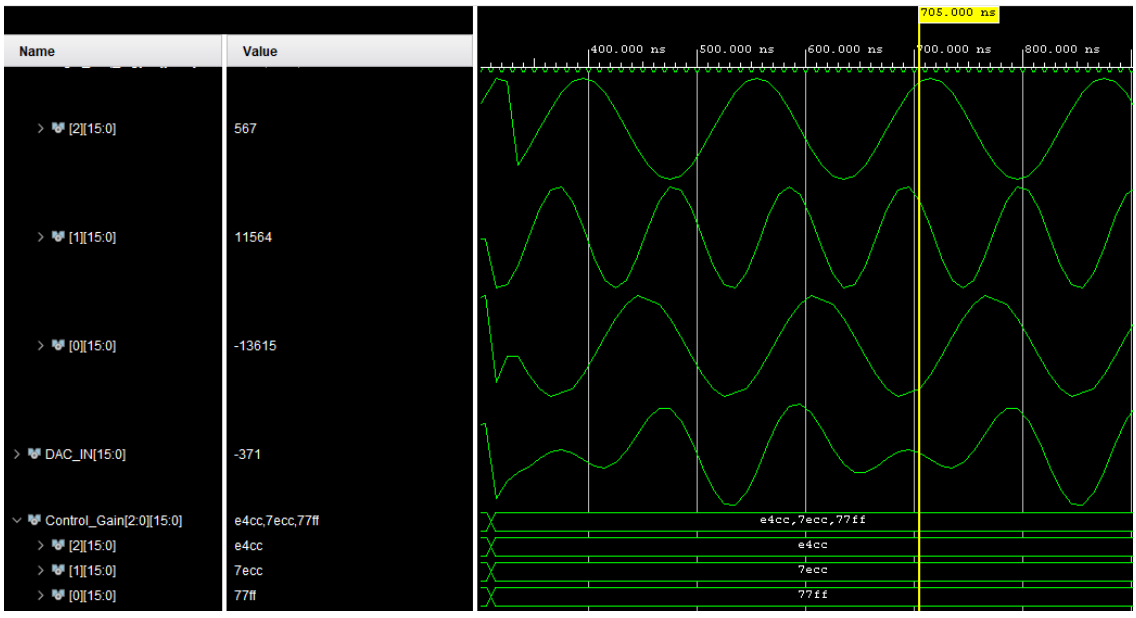
C

Simulation Results

C.1 Feed Forward Filter Simulation



C.2 Time Signal Generation Simulation



C.3 System Utilization Report

Name	LUT as Logic (63400)
project_toplevel	25850
comm (uart_communication)	583
ctrl (control_module)	217
math (Phasor_Calc_Toplevel)	23095
Control_Phasor_Generation_map (Contr	355
Feature_Gen_map (Feature_Gen)	1593
System_Phasor_calc_map (System_Pha	21147
multiplier_feat_weight (Vector_Vector_	20310
siggen (Multiple_time_signal_generation)	1958

Bibliography

[1] M. R. Anderson, *Compensation of nonlinearities in transducers*, 2005. [Online]. Available: https://www2.imm.dtu.dk/pubdb/edoc/imm3871.pdf.

[2] M. Vermeulen and N. van Klaveren, "Amplifier design for a piezoelectric transducer," *TU Delft Repository*, Jun. 2023.

[3] Y. Wu, P. Shankar, and P. Lewin, "Characterization of ultrasonic transducers using a fiberoptic sensor," *Ultrasound in Medicine & Biology*, vol. 20, no. 7, pp. 645–653, 1994. DOI: 10.1016/0301-5629(94)90113-9.

[4] R.-H. Munnig Schmidt, *RMS Acoustics & Mechatronics*, 2011. [Online]. Available: https://www.grimmaudio.com/wp-content/uploads/RMS-white-paper-4-MFB-theory.pdf.

[5] G. M. Clayton, S. Tien, K. K. Leang, Q. Zou, and S. Devasia, *A review of feedforward control approaches in nanopositioning for high-speed spm*, Aug. 2009. [Online]. Available: https://asmedigitalcollection.asme.org/dynamicsystems/article/131/6/061101/456145/A-Review-of-Feedforward-Control-Approaches-in.

[6] J. Jaspers and J. P. Metz, *Model Estimation- & Quadrature-Point Controller & Design*, Jun. 2023.

[7] A. Jiang, "Iir digital filter design using convex optimization," *University of Windsor*, 2010. [Online]. Available: https://scholar.uwindsor.ca/cgi/viewcontent.cgi?article=1431%5C&context=etd.

[8] F. J. Harris, "Multirate signal processing for communication systems," in 2004.

[9] A. Agarwal, *Polynomial regression*, Aug. 2018. [Online]. Available: https://towardsdatascience.com/polynomial-regression-bbe8b9d97491.

[10] S. Sreenivasa, *Radial basis function (rbf) kernel: The go-to kernel*, Aug. 2020. [Online]. Available: https://towardsdatascience.com/radial-basis-function-rbf-kernel-the-go-to-kernel-acf0d22c798a.

[11] W. Venstra, Personal Communication, Apr. 2023.

[12] T. Instruments, "Ads41xx 14-, 12-bit, 65-, 125-msps, ultra-low-power adc ... - ti.com," Feb. 2011. [Online]. Available: https://www.ti.com/lit/ds/symlink/ads4122.pdf.

[13] T. Instruments, *Dac904evm*, Aug. 2007. [Online]. Available: https://www.ti.com/tool/DAC904EVM#tech-docs.

[14] A. Van Der Veen. [Online]. Available: https://sps.ewi.tudelft.nl/Education/courses/ee2s31/slides/DSP6.pdf.

[15] M. Technology, *Dsc1001 - microchip technology*, 2011. [Online]. Available: https://www.microchip.com/en-us/product/DSC1001.

[16] D. Redmayne, E. Trelewicz, and A. Smith, *Understanding the effect of clock jitter on high speed adcs*, Aug. 2013. [Online]. Available: https://www.analog.com/media/en/reference-design-documentation/design-notes/dn1013f.pdf.

[17] T. D. Limited, *Jitter effects on analog to digital and digital to analog converters*, 2000. [Online]. Available: https://www.thewelltemperedcomputer.com/Lib/Troisi.pdf.

[18] E. O. Hammerstad, "Equations for microstrip circuit design," in *1975 5th European Microwave Conference*, 1975, pp. 268–272. DOI: 10.1109/EUMA.1975.332206.

[19] M. Electronics, Apr. 2023. [Online]. Available: https://www.mouser.com/.

[20] Xilinx, *Pg060 logicore reference manual*. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg060-floating-point.

[21] IEEE, "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019. DOI: 10.1109/IEEESTD.2019.8766229.

[22] Xilinx, *Ds335 floating point logicore reference manual*. [Online]. Available: https://docs.xilinx.com/v/u/en-US/floating_point_ds335.

[23] Xilinx, *Floating point adder logicore reference manual*. [Online]. Available: https://www.xilinx.com/htmldocs/ip_docs/pru_files/floating-point.html.