Auxiliary function development for the LISA metrology system

Von der QUEST-Leibniz-Forschungsschule der
Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des Grades

Doktor der Naturwissenschaften
— Dr. rer. nat. —

genehmigte Dissertation
von

Dipl.-Phys. Nils Christopher Brause

geboren am 01.10.1985
in Hannover, Deutschland

2018

Referent:	Prof. Dr. rer. nat. Karsten Danzmann AEI Hannover
Korreferent:	apl. Prof. Dr. rer. nat. Gerhard Heinzel AEI Hannover
Tag der Disputation:	02.05.2018

Abstract

The Laser Interferometer Space Antenna (LISA) is a planned gravitational wave detector to be positioned in space. It consists of three spacecrafts that use Long Range Interferometry (LRI) to measure relative distance changes between them. An important component of LISA is the LISA Metrology System (LMS) which is responsible for the distance measurements as well as various auxiliary functions: The beatnote acquisition allows the LMS to lock to an incoming beatnote signal with an unknown frequency and amplitude. It measures both with a Fast Fourier Transform (FFT) and controls the starting frequencies and gains of the Digital Phase Locked Loops (DPLLs) accordingly. The laser locking algorithm is used to lock the frequency of one laser to the frequency of another laser. This is done by locking the difference frequency between two lasers to a constant target and thus enabling heterodyne interferometry. The amplitude of the incoming beatnote signal can vary greatly over time. To compensate for that, the Automatic Gain Control (AGC) functionality observes the amplitudes and reconfigures the gains of the DPLLs accordingly. In LISA the pointing will be measured using an advanced Differential Wavefront Sensing (DWS) scheme, which track the differential phases between the segments of a Quadrant Photo Diode (QPD) directly instead of calculating them from the measured phases of the segment DPLLs. This improves the Carrier to Noise Density Ratio (CNR) in the DPLLs by a factor of two. The absolute distance between the spacecrafts is also measured to enable Time-Delay Interferometry (TDI) in post-processing. This is done by sending a Pseudo Random Noise (PRN) code via the laser link to a distant spacecraft, where it is correlated with a local copy of the same PRN code to determine the travel distance from the measured delay. Since only one of the three LISA spacecrafts has a radio link to earth, data has to be transferred between the three spacecrafts. This functionality is part of the Delay Locked Loop (DLL), by modulating the data onto the PRN code. In the course of this thesis, all the necessary auxiliary functions will be developed, thoroughly described and measured.

Keywords: interferometry, metrology, auxiliary functions

Zusammenfassung

Die Laser Interferometer Space Antenna (LISA) ist ein geplanter Gravitationswellendetektor, der im Weltraum stationiert werden soll. Sie besteht aus drei Satelliten, die Long Range Interferometry (LRI) nutzen um relative Abstandsänderungen zwischen ihnen zu messen. Eine wichtige Komponente von LISA ist das LISA Metrology System (LMS), welches für die Abstandsmessungen sowie diverse Hilfsfunktionen zuständig ist: Die Beatnote Acquisition ermöglicht dem LMS sich auf eine eingehende Beatnote unbekannter Frequenz und Amplitude zu locken. Sie misst beides mit einer Fast Fourier Transform (FFT) und kontrolliert damit die Startfrequenz und Gains der Digital Phase Locked Loops (DPLLs). Der Laser Lock Algorithmus wird benutzt um die Frequenz eines Lasers auf die eines anderen zu stabilisieren. Dies wird erreicht indem der Frequenzunterschied beider Laser konstant gehalten wird, wodurch Heterodyninterferometrie ermöglicht wird. Die Amplitude des Eingangssignals variiert stark im Laufe der Zeit. Um dem entgegenzuwirken folgt der Automatic Gain Control (AGC) der Amplitude und passt die Gains der DPLLs laufend an. In LISA wird die Richtung der Laserstrahlen mit Hilfe eines weiterentwickelten Differential Wavefront Sensing (DWS) Schemas gemessen, das die differentiellen Phasen zwischen den Segmenten der Quadrant Photo Diode (QPD) direkt misst. Dies verbessert die Carrier to Noise Density Ratio (CNR) in den DPLLs um einen Faktor 2. Der absolute Abstand zwischen den Satelliten wird ebenfalls gemessen um im Postprocessing Time-Delay Interferometry (TDI) zu ermöglichen. Dies wird erreicht indem ein Pseudo Random Noise (PRN) Code über die Laserverbindung zu einem entfernten Satelliten geschickt wird, wo er mit einer lokalen Version davon korreliert und so die Entfernung aus der gemessenen Verzögerung berechnet wird. Da nur einer der drei LISA Satelliten eine Funkverbindung zur Erde hat, müssen die Daten zwischen den Satelliten transferiert werden. Diese Funktionalität ist Teil der Delay Locked Loop (DLL), indem die Daten auf den PRN Code aufmoduliert werden. Im Laufe dieser Doktorarbeit werden alle nötigen Hilfsfunktionen entwickelt, vollständig vorgestellt und vermessen.

Schlagworte: Interferometrie, Messtechnik, Hilfsfunktionen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Gravitational Waves

More than a hundred years ago, Albert Einstein developed his General Theory of Relativity[1]. This theory extends Newton's laws of gravitation[2] to incorporate the effects of high velocities and strong gravitational fields. According to this theory, matter and energy bend the fabric of space-time itself, which in turn tells the contained matter how to move. Among other effects, the theory predicted the existence of so-called gravitational waves[3]. These waves are small ripples in space-time, which are generated by systems with accelerated and spherically asymmetric motion. Two objects orbiting each other is an example of such a system.

Just like electromagnetic waves, gravitational waves carry energy, although this energy has a much smaller impact on the visible matter. This makes it very hard to measure them. Even Einstein believed that a direct measurement could probably never be achieved. Despite this, gravitational waves have finally been measured directly at the Laser Interferometer Gravitational Wave Observatory (LIGO) in September 2015 in the USA[4], origination from the collision and merger of two massive black holes.

1.2 Gravitational Wave Detectors

LIGO is one of several gravitational wave detectors currently in operation on Earth. Others include Virgo[5] in Italy and GEO600[6] in Germany. All those gravitational wave detectors operate using the same basic principles of Long Range Interferometry (LRI). A coherent light beam is generated by a Laser and split into two beams using a half-transparent mirror, a so-called beam-splitter. Both beams travel orthogonally to each other to a distant mirror in each arm,

where they are reflected back to the beam-splitter mentioned above. The distance that both beams travel is the so-called arm length of the detector. At the beam-splitter, both beams are superimposed and generate constructive or destructive interference, depending on the phase difference between both beams.

If both beams travelled the same distance, this phase difference would be zero. When a gravitational wave hits the detector, the space-time will be slightly stretched or compressed in one direction with the opposite effect in the other direction. This leads to the beams travelling different distances and therefore having different phases at the beam-splitter. The emerging interference pattern is measured with a photodiode and converted into an electrical signal. For small phase differences, this electrical signal is proportional to the phase difference of both beams. This measurement method is called homodyne interferometry.

The signals generated by a gravitational wave have usually varying frequencies ranging from the mHz range up to the kHz range[7]. Depending on the construction and other environmental factors, gravitational wave detector are limited to a particular range of frequencies. This is known as the bandwidth of the gravitational wave detector.

1.3 Laser Interferometer Space Antenna

The ground-based gravitational wave detectors are severely limited in bandwidth. At the lower end of their frequency spectrum, they are limited by environmental noise such as gravity gradient noise and seismic noise[8]. That means that they are only able to measure gravitational waves of high frequencies in the range of 10 Hz to 2 kHz.

To be able to measure gravitational waves of lower frequencies, a gravitational wave detector needs to be positioned far away from the disturbances of Earth, i.e. in space. Such a gravitational wave detector in space, the Laser Interferometer Space Antenna (LISA), is currently being developed[9] and its launch is planned for 2034. LISA will consist of three instead of two interferometer arms, forming an equilateral triangle with an edge length of 2.5 Gm. LISA will be able to measure gravitational waves of low frequencies in the range from 0.1 mHz to 1 Hz.

In contrast to the gravitational wave detectors on Earth, LISA will not use the traditional homodyne interferometry mentioned above. Instead, LISA will be using heterodyne interferometry. In contrast to homodyne interferometry, where two beams that have been split off a single Laser beam interfere, in heterodyne interferometry two beams originating from two separate Lasers interfere. The lasers have different frequencies, and the frequency difference between them is held constant. Thus they generate a sinusoidal signal on the photodiode, the so-called beatnote. The phase difference information is embedded in the phase of

this beatnote.

1.4 LISA Metrology System

In comparison to homodyne interferometry, heterodyne interferometry requires substantial more complex measurement electronics to extract the phase signal out of the beatnote. In the case of LISA, this measurement electronics is called the LISA Metrology System (LMS). A prototype of the LMS has been jointly developed by the Albert Einstein Institute (Max Planck Institute for Gravitational Physics) in Hannover, the National Space Institute (Technical University of Denmark) as well as Axcon ApS (The FPGA Power House) in Denmark.

The core functionality of the LMS is digital. It uses 20 Analogue to Digital Converters (ADCs), which are converting the analogue signal from the photodiodes into a digital signal. Furthermore, it consists of 8 Field Programmable Gate Arrays (FPGAs) which are used to process these digital signals. The results can either be transferred to a Personal Computer (PC) or be converted back to an analogue signal using four Digital to Analog Converters (DACs).

The primary function of the LMS consists of measuring the relative phase of an electronic sinusoidal signal as accurately as possible. This phase measurement is done using a so-called Digital Phase Locked Loop (DPLL), which takes the beatnote as its input and outputs its frequency as well as its amplitude and relative phase. Therefore it is also called a phase meter, albeit it has a large number of auxiliary functions[10].

1.5 Auxiliary Functions Outline

This thesis will discuss the auxiliary functions of the LMS.

1.5.1 Beatnote acquisition

For the DPLL to function correctly, it needs three additional parameters: An approximate value of the beatnote frequency as well as loop gain parameters, which depend on the amplitude of the beatnote. In Chapter 3, a beatnote acquisition system will be developed, which is used to determine these three parameters from the beatnote. This is done using a Fast Fourier Transform (FFT), which will be explained in more detail.

1.5.2 Automatic gain control

When the beatnote signal changes its amplitude, the gain parameters of the DPLL have to be adapted to ensure continued functionality of the DPLL. In Chapter 5, an Automatic Gain Control (AGC) system will be developed, which continuously updates the gain parameters without using the FFT from Chapter 3.

1.5.3 Laser Locking

As explained earlier, in contrast to homodyne interferometry, heterodyne interferometry requires two lasers to be kept at a specific difference frequency. In Chapter 4, a Laser locking system will be developed, that continuously measures the beatnote frequency between two lasers and changes the frequency of one of the two lasers if the measured frequency deviates from the specified target.

1.5.4 Differential Wavefront Sensing

In an interferometer, the two interfering beams are usually not perfectly parallel to each other due to misalignment of the optical components of the interferometer or the spacecraft. This leads to different relative phases on different parts on the photodiode. Therefore these different phased need to be measured to allow correction of the alignment. In Chapter 6 an efficient system to measure these phase differences will be developed. It is called Differential Wavefront Sensing (DWS).

1.5.5 Ranging

In the case of LISA, the absolute distance between the spacecraft also needs to be measured. This data is required during post-processing to eliminate Laser noise. In Chapter 7 a ranging system to measure absolute distance using heterodyne interferometry is developed. This used a so-called Delay Locked Loop (DLL), which can also be used to transfer measurement data between the spacecrafts.

In the following chapters, each of these auxiliary functions will be developed, its purpose explained, and its performance measured.

Chapter 2

LISA Metrology System

The LMS is an essential component of the LISA mission. Among other things, it is responsible for scientific measurements, laser control and data transfer. The primary function is the precise phase measurement of various heterodyne signals, including the main beatnote, sidebands and the pilot tone. In this chapter, the basic structure of the current prototype of the LMS will be presented. It is also called Elegant Bread Board (EBB) and can be seen in Figure 2.1. It is used as the primary hardware platform for all technologies that are developed in this thesis. The functions of its key components will be explained in the following sections. More information about the EBB can be found at [10].

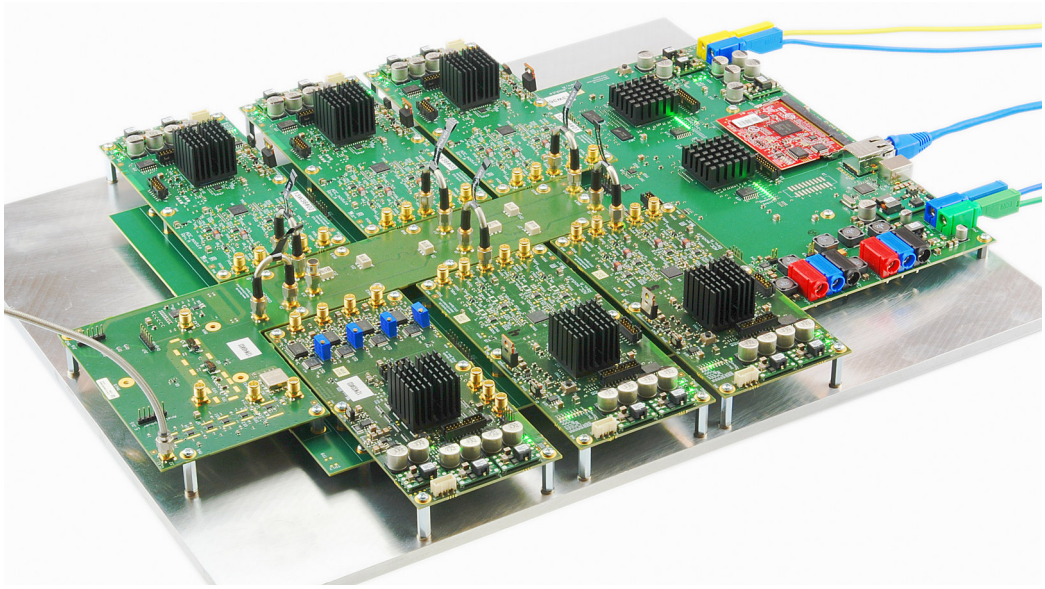


Figure 2.1: The EBB is the current prototype of the LMS on the LISA spacecraft.

2.1 Overview

A schematic representation of the EBB can be seen in Figure 2.2. It consists of the following building blocks:

- Mainboard
- Bridge module
- Clock module
- Five ADC modules
- DAC module
- FFT module
- Micro controller module

The presented modules will be described in more detail in the following sections.

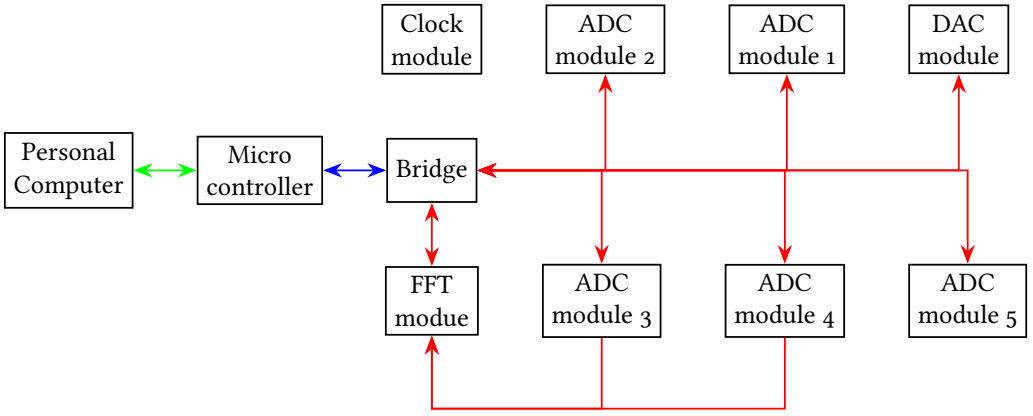


Figure 2.2: A schematic representation of the LMS, including the bridge module, the clock module, five ADC modules, the DAC module, the FFT module and the microcontroller. Red arrows are serial GBit interfaces, the blue arrow is a parallel memory interface, and the green arrow are Ethernet and RS232 interfaces.

2.2 Mainboard

The mainboard provides the underlying infrastructure, such as power supplies for digital and analogue circuits as well as digital interconnects. Most interconnects

are realised as serial GBit links, which are marked with red arrows in Figure 2.2. They have a maximum bandwidth of 3.2 Gbit s^{-1}. The interconnect between the microcontroller and the bridge is a 16 bit parallel memory interface, marked with a blue arrow. External interfaces are 1000BASE-T Ethernet and RS232, both provided by the microcontroller, marked with a green arrow. Scientific measurement data is read out through the Ethernet port, which is also used to control the LMS by setting various parameters. The RS232 port is used primarily for debugging and reprogramming purposes of the microcontroller.

The FFT module and the bridge module are soldered onto the mainboard. All other modules have the form of Add-In cards and can be replaced when deemed necessary.

2.3 Micro controller

The microcontroller module used in the EBB is the Embedded Artists' LPC3250, which is built around the NXP LPC3250 microcontroller. Among other things, it features a fast 32 bit ARM core with an Floating Point Unit (FPU), 64 MB Random Access Memory (RAM), 128 MB Flash storage as well as a 1000BASE-T Ethernet transceiver and an external 16 bit parallel memory bus.

The primary task of the microcontroller is to filter the measurement data using floating-point arithmetic and transmit them to a PC via Ethernet. Other responsibilities include the control of various functions of the EBB autonomously, e.g. the laser lock algorithm which will be explained in Chapter 4, as well as the readout of temperature sensors mounted on the mainboard and the modules and connected via I^2C.

2.4 Bridge

The primary function of the bridge is to interface the parallel memory interface of the microcontroller with the serial GBit interfaces of the other modules. It collects the measurement data from the ADC, DAC and FFT modules, reformats them and forwards them to the microcontroller. At the same time, it receives commands from the microcontroller and sends them to the modules mentioned above. Another essential function is to forward measurement data from the ADC modules to the DAC module to build a closed loop used by the laser locking facility explained in Chapter 4.

The bridge module consists of a Xilinx Spartan 6 XC6SLX75T FPGA, featuring eight GBit Transceivers. Seven of those are used to connect to the ADC, DAC and FFT modules.

2.5 Clock Module

The clock module is used to generate an 80 MHz system clock for the digital part of the EBB (except the microcontroller, which has its own 50 MHz crystal oscillator) as well as a highly phase stable 75 MHz pilot tone for jitter correction in post-processing. Both clocks are generated from a 2.4 GHz clock, which is divided by 30 and 32 to produce the system clock and the pilot tone, respectively. More information about the clock module can be found in [11].

2.6 DAC Module

The DAC module is used to convert digital signals back to analogue signals. It consists of four Texas Instruments DAC5675A DACs with an appropriate analogue back end as well as a Xilinx Spartan 6 XC6SLX75T FPGA connected to them. The DAC is mainly used to control a laser with the laser lock explained in Chapter 4. It is also part of the ranging system, which is explained in Chapter 7.

2.7 FFT Module

The FFT module is solely used by the beatnote acquisition system described in Chapter 3. It is connected to two of the five ADC modules and is used to get a rough estimate of the frequency and amplitude of the heterodyne signals. As the name suggests, this is done using the FFT algorithm, which will be explained in greater detail in the chapter mentioned above. The FFT module consists of a Xilinx Spartan 6 XC6SLX150T FPGA, which is the largest variant in the Xilinx Spartan 6 series of FPGAs to provide enough space for the resource-hungry FFTs.

2.8 ADC Modules

The ADC module consists of the four-channel ADC Texas Instruments ADS6445 with an appropriate analogue front end as well as the Xilinx Spartan 6 XC6SLX75T FPGA connected to it. On the FPGA several Digital Signal Processing (DSP) algorithms are being run. Its primary function is the measurement of the phase and frequency of the heterodyne signals mentioned above. Several DPLLs are used for this purpose, which is described in more detail in Subsection 2.8.1. Another function is ranging, which is explained in Chapter 7.

2.8.1 Digital Phase Locked Loop

A schematic representation of a standard DPLL can be seen in Figure 2.3. It mainly consists of the following parts:

- Phase detector
- Low pass filter
- loop controller
- Start frequency adder
- Phase accumulator
- Sine/Cosine Look-Up Table (LUT)

The phase detector consists of a multiplier, which multiplies a cosine by the input signal. It produces a signal consisting of the sum and the difference of the frequencies of the input signal and the cosine. This signal is low-pass filtered to remove the sum frequency component. The resulting output is called the Q value and describes the phase difference between the cosine and the input signal. If the input signal and the cosine have a phase difference of $\frac{2\pi}{4}$, the Q value is zero.

Therefore the Q value is used as an error signal and is fed into the loop controller, which calculates the so-called actuator signal. The loop controller is a Proportional-Integral (PI) controller with adjustable gains G_P and G_I in this case. A starting frequency f_start which must be near the actual heterodyne frequency is added to the actuator signal, and the result f_out is fed into an Numerically Controlled Oscillator (NCO).

An NCO consists of a Phase Accumulator (PA), which integrates the input frequency to a phase φ. This phase is then converted to a sine or cosine signal using a LUT, which assigns a sine and cosine value to every possible phase value. This NCO generates the cosine mentioned above, which is multiplied by the input signal.

By controlling the actuator signal and thereby the NCO, the PI controller tries to minimise the error signal in such a way that the phase of the cosine tracks the phase of the input signal with a phase difference of $\frac{2\pi}{4}$.

Typically, a sine is also generated by the NCO and multiplied with the input signal in a separate signal chain. When the DPLL is locked, this sine is in-phase with the input signal and can be used to obtain its amplitude (also called the I value) when multiplied. However, this part has been left out from the schematic for the sake of simplicity. It is not essential for the proper function of the DPLL, but will be useful for the AGC algorithm in Chapter 5.

More information about the DPLL can be fount in [12].

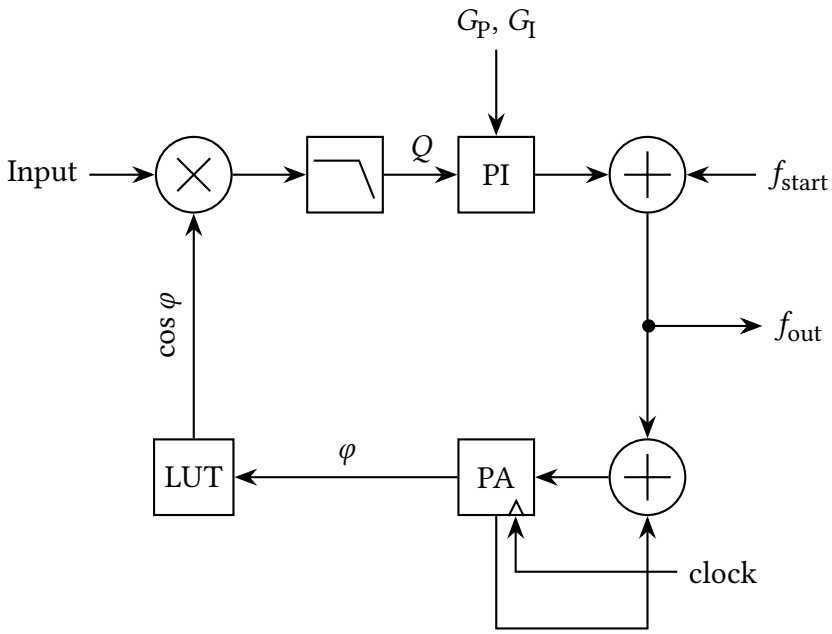


Figure 2.3: A standard DPLL without the I part, showing the phase detector, loop controller (PI), the starting frequency f_{start}, the phase accumulator (PA) and the look-up table (LUT).

Chapter 3

Beatnote Acquisition

The beatnote acquisition functionality of the LMS is used to find the frequency and amplitude of an unknown beatnote signal. The frequency is used as a starting frequency for the DPLLs in the phase measurement system, and the amplitude is used to determine the correct gains used in the DPLLs. The DPLL is described in greater detail in the PhD thesis of Oliver Gerberding[12].

To detect the frequency and amplitude of an unknown beatnote signal, it is transformed into its frequency spectrum. In this form, the beatnote frequency peak should stand out from the surrounding noise and can easily be detected. Its height denotes the amplitude of the beatnote signal. The decomposition into a frequency spectrum is performed by an accelerated discrete version of a Fourier Transform (FT)[13], which is called FFT and is described in the next subsection.

The FFT constitutes the heart of the beatnote acquisition system, alongside a simple peak finding algorithm, a gain calculation algorithm and some miscellaneous helper functionality. An essential feature of the beatnote acquisition system is also the ability to exclude specific frequencies from detection. These include multiples of 10 MHz, which are commonly found in a lab environment, due to its frequent use as reference frequency in electronic equipment, as well as its harmonics. This frequency exclusion feature is implemented inside the peak finding algorithm mentioned above, which will be described in more detail later.

3.1 Fast Fourier Transform

The FFT is a high-speed algorithm that is used to calculate the frequency spectrum of an equidistantly sampled digitised signal. Even though the ADC samples are real numbers, the FFT input has the form of N complex numbers representing the discrete amplitudes in the time domain. These numbers are transformed in such a way that the result also yields N complex numbers but representing discrete

amplitudes in the frequency domain. Therefore such an FFT is also called N point FFT.

Its most popular variant has been developed by James Cooley and John W. Tukey in 1965[14] and will be used throughout the thesis. This particular algorithm has been chosen, because it is very fast, relatively easy to implement and straightforward to parallelise, which is a huge benefit in an FPGA based phase meter.

There are many variants of the FFT[15][16][17]. Most of them are tailored towards a specific N, being very efficient at that N. However, the exact N is not very important and should be changeable in any case. Hence, we will concentrate on the fundamental FFT algorithm in the following.

The LMS does not need a particular high N because the DPLL will also lock, if the starting frequency is a few kHz away from the actual signal frequency. Therefore, an $N = 1024$ has been chosen in this thesis. This results in a frequency resolution of 78.125 kHz at a sampling rate of 80 MHz. This will be explained in greater detail, later.

3.1.1 Theory of Operation

The FFT algorithm works by recursively dividing the processing of the N input data points into smaller FFTs. In each step, the number of points to be computed gets divided into two as equally sized parts as possible. In each of the smaller FFTs, this process is repeated until the number of points in an FFT is a small prime number.

The FFT algorithm from James Cooley and John W. Tukey only works for the prime number 2, which means that N has to be an integer power of 2. This prime number is also called the radix of the FFT, and the described particular FFT algorithm is therefore also called a radix-2 FFT. Other FFT algorithms work for different radices, but they are not as simple to implement and do not have any significant advantages over the radix-2 FFT.

The final 2 point FFTs are simple 2 point Discrete Fourier Transforms (DFTs) and are called butterflies in the context of FFTs. An FFT of the length N consists of

$$N_\mathrm{b} = \frac{N}{2} \log_2 (N) \tag{3.1}$$

such butterflies and therefore has a complexity of $\mathcal{O}(N \log_2 N)$. As a comparison, a DFT that directly implements its defining formula has a complexity of $\mathcal{O}(N^2)$[18], which is much worse.

Each butterfly takes two complex numbers x_1 and x_2 as input and has two complex numbers y_1 and y_2 as output, as shown in Figure 3.1. The butterfly also

is associated with an additional parameter k that depends on the position of the butterfly in the FFT. It will be explained later.

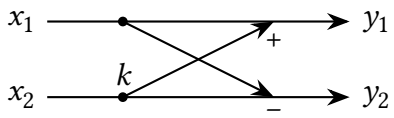


Figure 3.1: Schematic of a butterfly. x_1 and x_2 are the input numbers of the butterfly and y_1 and y_2 are the output numbers. k is the exponent of the twiddle factor.

A butterfly looks as follows:

$$\begin{aligned} y_1(k) &= x_1 + t^k x_2 \\ y_2(k) &= x_1 - t^k x_2 \,, \end{aligned} \tag{3.2}$$

with

$$t = \mathrm{e}^{-\frac{2\pi \mathrm{i}}{N}} \,. \tag{3.3}$$

t is the so-called twiddle-factor. This factor only depends on the size of the FFT.

The computation of an 8 point FFT is exemplarily shown in Figure 3.2.

On the left, the time-dependent input values X_0 to X_7 are shown. They are arranged in bit reversed order. This is done by reversing the binary representations of the input ordinal numbers. For example in an 8 point FFT, the ordinal numbers range from 0 to 7 and can be represented using three bits. To calculate the input number required, e.g. at the third input of the FFT, first, the binary representation of 3 has to be written down: 011. Then the numbers are reversed leading to 110, which represents the number 6. Therefore the third input of the FFT expects the sixth input number.

The input values traverse through several stages of the FFT that are marked by red rectangles. Each stage is further divided into one or more butterfly groups, marked by blue rectangles, and every group of butterflies consists of one or more butterflies, represented by a cross. As can be seen, an 8 point FFT consists of 12 butterflies in accordance with Equation 3.1. On the right side, frequency dependent output values Y_0 to Y_7 are shown. They are in ascending order.

The whole N point FFT is divided into

$$N_\mathrm{s} = \log_2 N \tag{3.4}$$

stages. These stages are further divided into

$$N_\mathrm{gs} = \frac{N}{2^{s+1}} \tag{3.5}$$

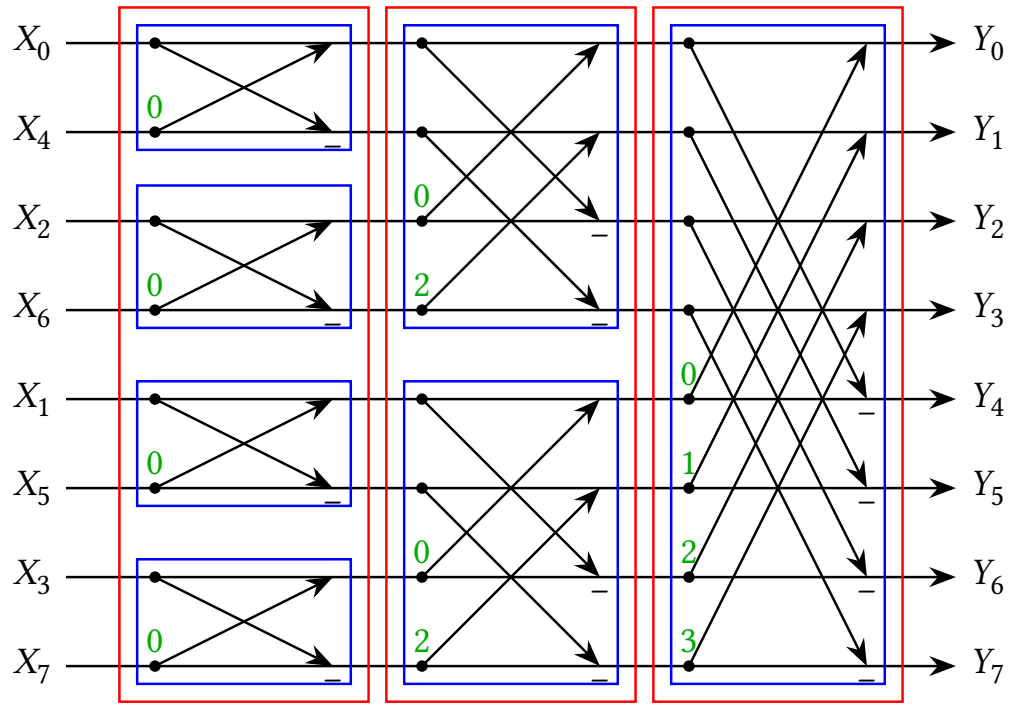


Figure 3.2: Schematic of an 8 point FFT. Each cross represents one butterfly. Each blue rectangle is one group, and each red rectangle is one stage. Labelled in green are exponents k of the twiddle factors.

groups of butterflies, where s is the stage number, beginning at $s = 0$. Each stage consists of

$$N_{\mathrm{bs}} = \frac{N_{\mathrm{b}}}{N_{\mathrm{s}}} = \frac{N}{2} \tag{3.6}$$

butterflies. Therefore, each group consists of

$$N_{\mathrm{bg}} = \frac{N_{\mathrm{bs}}}{N_{\mathrm{gs}}} = 2^s \tag{3.7}$$

butterflies.

Using these numbers, we can finally calculate the exponent of the twiddle-factor:

$$k = \frac{bN}{2N_{\mathrm{bg}}}, \tag{3.8}$$

where b is the butterfly number in its group, beginning at $b = 0$. E.g. for the second butterfly in a group of four butterflies, $b = 1$ and $N_{\mathrm{bg}} = 4$. With $N = 8$, this results in $k = 1$. In Figure 3.2, the parameter k is marked in green.

In contrast to an FT, the DFT and FFT are not computing an integral due to its discrete nature. Therefore, the input and output units are the same. In case of a signal from an AGC, this would be volts. The output of the FFT is not multiplied by any additional normalising factors.

3.2 Real value Input Data

In our case, the input to the FFT is the Alternating Current (AC) from a photodiode that is digitised by the ADCs, as described in Section 1.4, and is completely real data. Therefore, only a real data FFT would be needed, but the FFT algorithm is intrinsically an algorithm dealing with complex numbers. To solve this problem, two approaches have been tried. They will be discussed in the following.

3.2.1 Padding the Input

As a first approach, the imaginary part of the input data was padded with zeros. However, the output data Y_k still consists of complex numbers. They are in this case symmetrical around the Nyquist-Frequency[19]

$$f_{\mathrm{ny}} = \frac{f_{\mathrm{s}}}{2}\,, \tag{3.9}$$

where f_{s} is the sampling frequency of the FFT. Therefore,

$$Y_{N-k} = \overline{Y_k}\,, \tag{3.10}$$

with N being the number of input data points and k the number of an arbitrary data point.

This symmetry shows that a real FFT does not have more independent output data points than input data points. Therefore the number of resulting frequency bins is

$$N_{\mathrm{fb}} = \frac{N}{2}\,. \tag{3.11}$$

However, the more important frequency resolution Δf is not affected by this condition, because it only depends on the number of input values and not on the number frequency bins. The frequency resolution describes the distance between two frequency bins in an FFT and can be calculated as

$$\Delta f = \frac{f_{\mathrm{s}}}{N}\,. \tag{3.12}$$

For example, an FFT with $N = 1024$ points and an input signal with a sampling frequency of $f_{\mathrm{s}} = 80\,\mathrm{MHz}$ results in a frequency resolution of

$$\Delta f = \frac{80\,\mathrm{MHz}}{1024} = 78.125\,\mathrm{kHz}\,. \tag{3.13}$$

For $N_{\mathrm{fb}} = 512$ frequency bins, the available frequency bins are numbered from Y_0 to $Y_{\mathrm{N_{fb}}-1} = Y_{511}$. Therefore the frequencies range from

$$f_0 = 0\Delta f = 0\,\mathrm{Hz} \tag{3.14}$$

to

$$f_{\mathrm{N_{fb}}-1} = f_{511} = 511\Delta f = 39.921\,875\,\mathrm{MHz}\,. \tag{3.15}$$

To get a real spectrum from an FFT, the absolute value of its complex output has to be obtained by multiplying it by its complex conjugate and then taking the square root. This, however, has the side effect of the phase information loss of the input signal, since there are now less real output numbers than real input numbers (512 instead of 1024). Fortunately, the phase information is not relevant for the beatnote acquisition.

3.2.2 Increasing Efficiency

he process mentioned above is not very efficient since only half of the input data of the FFT gets filled with the input signal, and the other half of the FFT stays unused. Fortunately, there are ways to optimise this misuse of precious computational resources.

One approach is to exploit the symmetry in Equation 3.10. Additionally, there is another symmetry when an FFT has purely imaginary input data:

$$Y_{N-k} = -\overline{Y_k}\,. \tag{3.16}$$

Using both symmetries, either an N point FFT can be used to compute the spectrum of two sets of N real data points or an $\frac{N}{2}$ point FFT can be used to compute the spectrum of N real data points[20].

Computing two separate real FFTs with a single complex FFT can result in cross-talk between both real FFTs if the computations are carried out with limited precision, as it is the case on an FPGA. The ADC signal is represented in two's complement format with, in the case of the LMS, a bit-width of 14 bit. This fixed bit-width limits accuracy, since arithmetic operations such as addition and multiplication produce numbers with greater bit-widths, which must be shortened before further processing can happen. E.g., the multiplication of two numbers of the length m results in a number of length $2m$. Trimming this number back two a length of m bits for further processing results in an information loss of 50 %. Having said that, two $\frac{N}{2}$ point FFTs require more logic space than a single N point FFT, because of the additional surrounding logic that is part of every FFT.

Since for the beatnote acquisition we are only interested in the rough amplitude of a signal, there is no need for high precision spectra. Also, the FPGA logic

space is limited, and there is more than one ADC anyway. Therefore, the first method, where an N point FFT is used to compute the spectrum of two sets of N real data points, will be used in the following.

The first set of N real data points are filled into the real part of the FFT input, and the second set of N real data points are filled into the imaginary part of the same FFT input. Then a standard FFT is computed. Extracting the two separate results from the output of the FFT requires some more computation:

$$\begin{aligned} Y_{\mathrm{a}_m} &= \frac{1}{2}\left(Y_m + \overline{Y_{N-m}}\right) \\ Y_{\mathrm{b}_m} &= -\frac{\mathrm{i}}{2}\left(Y_m - \overline{Y_{N-m}}\right), \end{aligned} \tag{3.17}$$

where $m \in \mathbb{N}^+, k < \frac{N}{2}$, y_m is the original output from the FFT and y_{a_m} and y_{b_m} are the extracted results for the first (Y_{a}) and second (Y_{b}) real FFT.

Since the FPGA does not know about imaginary numbers and the results get squared in a later step, the factor –i in the calculation of Y_{b_m} can be omitted to reduce the required computational resources. Finally, the same steps as described in Section 3.2.1 can be executed to obtain a real spectrum.

Note that the Direct Current (DC) part cannot be obtained using this method. According to Equation 3.17, the computation of Y_{a_0} and Y_{b_0} would require the output value Y_N which does not exist, because there are only N output values. Fortunately, the DC part is not relevant for the beatnote acquisition.

3.3 Implementation

The FFT was written in Very high speed integrated circuit Hardware Description Language (VHDL) and features synthesis-time configuration of the bit-widths of its inputs and the number of frequency bins. It consists of three basic parts:

- One butterfly
- dual-port memory blocks
- control logic

The FFT reaches a duty cycle of approximately 50%. That means, assuming the input data is sampled with the same frequency that the FFT is clocked with, the FFT can compute spectra of roughly half of the input data if it runs continuously. This is more than enough for beatnote acquisition since it will only run at a frequency of a few Hertz.

After the processing by the FFT, the absolute value of the output signal is calculated. In this case, however, the square root is omitted, and the output is only

multiplied by its complex conjugate. The reason is that the square root cannot be easily implemented on an FPGA. Since the amplitude from the FFT is now the square of the real amplitude, this has to be considered in the gain calculation algorithm, which will be described in Section 3.5.

The result of the FFT is finally transferred to the peak finding algorithm, which will be described in Section 3.4 and then to the gain calculation algorithm.

The whole implementation will be presented in full detail in this section.

3.3.1 The Butterfly

As stated in the previous section, an N point FFT consists of $n = \frac{N}{2}\log_2(N)$ butterflies. The required powers of the twiddle factors from Equation 3.2 are calculated at synthesis-time for a given N since they do not depend on the input data. They are loaded into a RAM at the initialisation-time of the FPGA.

Equation 3.2 contains two complex multiplications ($t^k x_2$) as well as two complex additions. Since both complex multiplications are the same multiplication, its result can be reused and only counts as a single multiplication. This results in a total of one complex multiplication and two complex additions.

Since the FPGA can only perform real calculations, Equation 3.2 had to be divided into real and imaginary parts:

$$\begin{aligned}
\Re y_1 &= \Re x_1 + \Re t^k \Re x_2 - \Im t^k \Im x_2 \\
\Im y_1 &= \Im x_1 + \Im t^k \Re x_2 + \Re t^k \Im x_2 \\
\Re y_2 &= \Re x_1 - \left(\Re t^k \Re x_2 - \Im t^k \Im x_2\right) \\
\Im y_2 &= \Im x_1 - \left(\Im t^k \Re x_2 + \Re t^k \Im x_2\right) .
\end{aligned} \tag{3.18}$$

Ignoring redundant calculations, this contains four real multiplications and six real additions:

$$\begin{aligned}
M_{\mathrm{rtrx}} &= \Re t^k \Re x_2 \\
M_{\mathrm{itix}} &= \Im t^k \Im x_2 \\
M_{\mathrm{rtix}} &= \Re t^k \Im x_2 \\
M_{\mathrm{itrx}} &= \Im t^k \Re x_2 \\
A_1 &= M_{\mathrm{rtrx}} - M_{\mathrm{itix}} \\
A_2 &= M_{\mathrm{rtix}} + M_{\mathrm{itrx}} \\
\Re y_1 &= \Re x_1 + A_1 \\
\Im y_1 &= \Im x_1 + A_2 \\
\Re y_2 &= \Re x_1 - A_1
\end{aligned}$$

$$\Im y_2 = \Im x_1 - A_2\,, \tag{3.19}$$

where M_{rxtx}, M_{itix}, M_{rtix}, M_{itrx}, A_1 and A_2 are temporary variables.

Figure 3.3 gives a schematic overview of how such a butterfly is implemented in an FPGA.

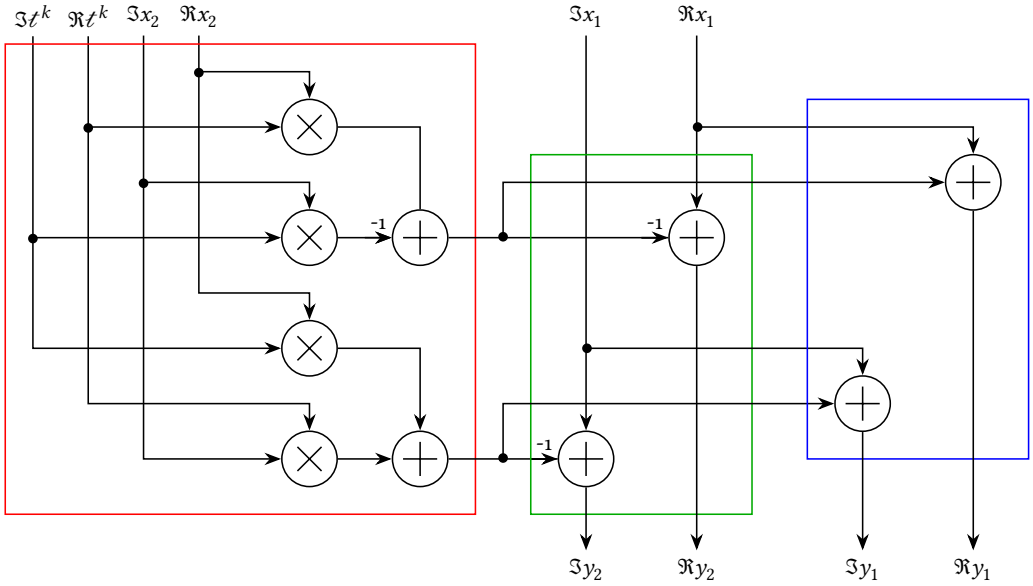


Figure 3.3: Schematic of a butterfly implementation in an FPGA. Red is a complex multiplier, green is a complex subtractor, and blue is a complex adder

The red box shows how a complex multiplication is implemented using real multipliers and real adders. The green and the blue boxes show the implementation of complex addition and subtraction using real adders and subtractors.

According to Equation 3.1, an FFT with $N = 1024$ data points would consist of $N_\text{b} = 5120$ butterflies and would therefore need 20480 real multiplications and 30720 real additions. Unfortunately, this is way out of the capabilities of any modern FPGA[1]. Therefore, our FFT implementation only uses a single butterfly which is getting reused in every computation step. The VHDL source of the butterfly implementation can be found in Section B.1.2.

3.3.2 Dual-port Memory Blocks

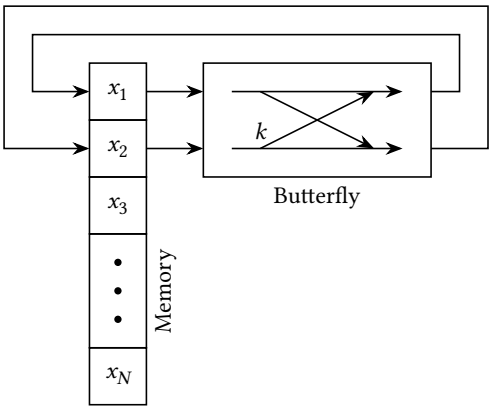
Another vital part of an FFT implementation is the memory arrangement. There are two different variants of how the memory can be arranged in an FFT implementation. Both have in common that dual-port RAM blocks are used. Dual-port

[1]E.g. a Xilinx Virtex-6 has only up to 2016 multipliers and adders[21]

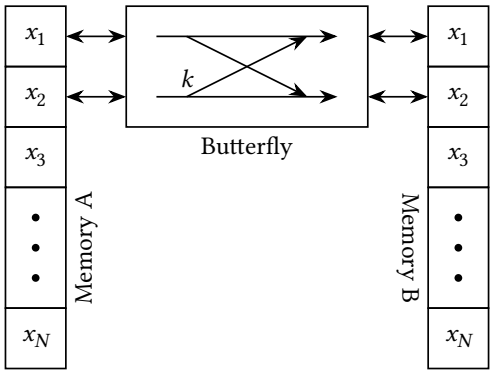
memory is commonly used in Video Memory (VRAM)[22] and register files. It has the advantage that any two memory cells can be read or written to at the same time, as opposed to single-port RAM, which only allows a single read or write operation at a time. This is useful since every butterfly operation always works on two numbers (x_1 and x_2) at the same time. Therefore, these numbers can be retrieved from and stored in the dual-port memory in a single step, thus saving time as well as complexity in the control logic.

In the first variant, one dual-port RAM block is used for the input numbers. Each time a butterfly is computed both input numbers are read simultaneously from the dual-port RAM block. The butterfly processes them and writes the, back to the same memory locations. A schematic overview of this variant can be seen in Figure 3.4a.

In the second variant, two dual-port RAM blocks are used, with one of them holding the initial values of the input numbers. Each time a butterfly is computed, both input numbers are read from one dual-port RAM block. The butterfly processes them, and the result is stored in the other dual-port RAM block. After each stage, the dual-port RAM blocks are exchanged by the control logic. A schematic overview of this variant can be seen in Figure 3.4b.



(a) FFT implementation with only one dual-port RAM block. In each stage the numbers are read from the dual-port RAM block, processed by a butterfly and stored in the same dual-port RAM block.



(b) FFT implementation with two dual-port memories. In each stage, the numbers are read from one dual-port RAM block, processed by the butterfly and stored in the other dual-port RAM block. Then both dual-port RAM blocks are exchanged.

Figure 3.4: Common memory arrangements in FFT implementations

The second variant is faster than the first variant but comes at the cost of twice the memory usage and more complex logic. There are also variants, where only a single-port memory is used, but this is slower since the input numbers

have to be read out sequentially. The latter setup requires even more complex logic and takes much longer.

The single dual-port memory arrangement has been chosen in this implementation to keep space requirements for the FPGA low. The availability of dual-port memory on modern FPGAs was very beneficial to the speed of our FFT implementation.

Memory usage

This particular FFT implementation has been written to be used on the LMS. Therefore it accepts input signals with a width of $b = 14$ bit, which is the resolution of the used ADCs. The number of samples can be configured at synthesis time. In the lab, it has been found to be sufficient to use $N = 1024$ samples. This results in a block RAM usage of $N \times 2 \times b = 28$ kbit per FFT for sample storage. Additionally, Read Only Memory (ROM) for the storage of $\frac{N}{2}$ complex twiddle-factors is needed, which equates to $\frac{N}{2} \times 2 \times b = 14$ kbit. Since the FPGA uses block RAM to store large amounts of ROM data, this leads to a total of 42 kbit of block RAM per FFT. Since one FFT can be used to process two real ADC channels, this amounts to 21 kbit of block RAM per ADC channel. As a comparison, the proprietary FFT core from Xilinx uses 54 kbit of block RAM per FFT or 27 kbit of block RAM per channel. This is slightly more, but in return the proprietary FFT features a 100 % duty cycle.

3.3.3 Control Logic

The operation of the FFT is controlled by an Finite State Machine (FSM) together with a bin counter and a butterfly counter. The bin counter is used when reading new input data or writing the result. It counts from zero to $N - 1$ and stores N input values in the dual-port RAM block and reads N output values from the dual-port RAM block, while the butterfly counter is used to coordinate the butterfly computations. It counts from zero to $N_b - 1$ and sets the memory addresses for x_1, x_2, y_1 and y_2 as well as the t^k parameter of the butterfly according to the current butterfly number. The finite state machine consists of six states. A schematic overview of the state machine can be seen in Figure 3.5.

The initial state is the *idle* state, in which the FFT resides when the reset signal to the FFT is low[2]. Once the reset signal rises, the state machine changes into the *Input* state. In this state, data is read from the input port of the FFT and saved in the dual-port memory blocks. The bin counter counts each input number to

[2]The reset signal is always active low. That means it is active when it is low (logical zero), and it is not active when it is high (logical one)

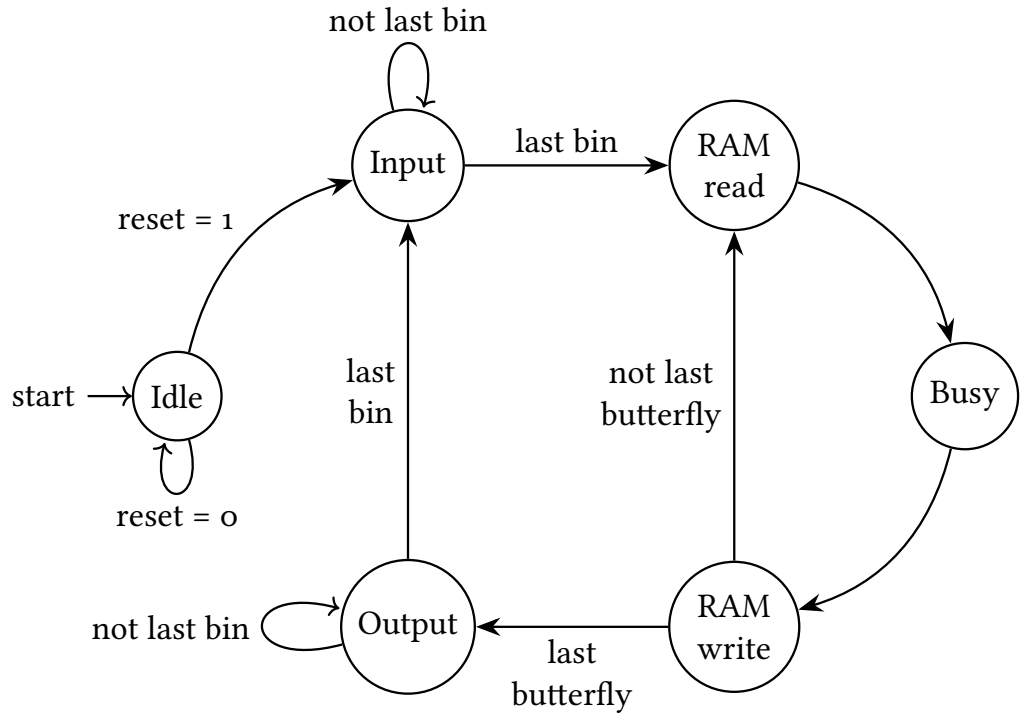


Figure 3.5: Schematic overview of the FFT finite state machine.

make sure that the correct number of input values are read. After the last input number has been read and saved into its corresponding memory bin, the actual FFT computation loop starts. This loop consists of three states.

In the first state, the *RAM read* state, two complex numbers are read from the dual-port RAM block and given to the butterfly. After that, in the *Busy* state, the actual butterfly computation takes place. This is a separate state because the computation is somewhat complicated and takes quite some time. The third state in the computation loop is the *RAM write* state, in which the result from the butterfly is written back to the dual-port memory blocks. If this was the last butterfly to compute the finite state machine changes into the *Output* state, otherwise it increases the butterfly counter and continues the computation cycle.

In the *Output* state, the results are read from the dual-port RAM blocks and written to the output port of the FFT. Again, the bin counter makes sure to output the correct number of complex numbers. From there on, the FFT restarts at the *Input* state to accept the next data set.

The VHDL source of the control logic implementation can be found in Section B.1.1.

3.4 Peak Finder

The peak finder receives a serial stream of frequency/amplitude pairs and outputs the frequency/amplitude pair with the highest amplitude. It also evaluates the user-defined list of ignored frequencies.

The peak finder first initialises an internal frequency/amplitude pair with 0/0. It replaces this frequency/amplitude pair with a new pair, whenever the amplitude of the new pair is higher than the amplitude of the existing pair and if the frequency is not on the list of ignored frequencies. The internal frequency/amplitude pair is reset, whenever a frequency/amplitude pair with a frequency value of 0 arrives, and thus a new spectrum begins.

The frequency/amplitude pair with the highest amplitude found so far is output and can be used as the starting frequency and for the gain calculation for a DPLL later.

The VHDL source of the peak finder implementation can be found in Section B.1.3.

3.5 Gain Calculation

To understand the gain calculation algorithm, the influence of gains on the function of a DPLL must be understood. A linear model will be presented to provide a basic understanding of the relationship between amplitude and gain. To get absolute values for the gain, a proper non-linear low-level simulation will be performed.

3.5.1 Linear Model

A general linear model of a DPLL looks as follows:[12]

$$L(z) = \underbrace{\frac{A}{2}}_{L_{\mathrm{PD}}} \underbrace{2^{G}}_{L_{\mathrm{PG}}} \underbrace{\left(2^{G'_{\mathrm{P}}} z^{-1} + 2^{G'_{\mathrm{I}}} \frac{z^{-1}}{1 - z^{-1}}\right)}_{L_{\mathrm{PI}}} \underbrace{\frac{2\pi z^{-1}}{1 - z^{-1}}}_{L_{\mathrm{NCO}}} \underbrace{\frac{1 + z^{-1} + z^{-2} + z^{-3}}{4}}_{L_{\mathrm{LPF}}}, \tag{3.20}$$

where L_{PD} is the transfer function of the phase detector, with A being the amplitude of the incoming signal. L_{PG} is a pre-gain that is applied just before the PI controller. L_{PI} is the transfer function of the PI controller, with G'_{P} and G'_{I} being the gains of the PI controller itself. L_{NCO} is the transfer function of the NCO. L_{LPF} is the transfer function of the low pass filter, which computes the moving average of the last four values.

The pre-gain 2^G can be factored into the gains of the PI controller, which leaves us with the following linear model:

$$L(z) = \frac{A}{2}\left(2^{G_\mathrm{P}} z^{-1} + 2^{G_\mathrm{I}} \frac{z^{-1}}{1 - z^{-1}}\right) \frac{2\pi z^{-1}}{1 - z^{-1}} \frac{1 + z^{-1} + z^{-2} + z^{-3}}{4}, \tag{3.21}$$

consisting only of the amplitude A of the signal as well as the gains G_P and G_I for the PI controller.

From Equation 3.21 one can already see how the amplitude influences the gains. When the amplitude halves both gains have to increase by one and when the amplitude doubles, both gains have to decrease by one to keep the loop output the same.

Since according to Section 3.3, the amplitude signal coming from the FFT is already squared, the square root has to be taken before this signal can be used in the gain calculation.

With G_Pf being the G_P gain for the maximum amplitude, G_If being the G_I gain for the maximum amplitude, A_S being the squared amplitude from the FFT, the corresponding G_P and G_I gains can be calculated as follows:

$$\begin{aligned} G_\mathrm{P} &= G_\mathrm{Pf} + \left\lfloor \log_2\left(\frac{1}{\sqrt{A_\mathrm{S}}}\right) \right\rfloor \\ G_\mathrm{I} &= G_\mathrm{If} + \left\lfloor \log_2\left(\frac{1}{\sqrt{A_\mathrm{S}}}\right) \right\rfloor, \end{aligned} \tag{3.22}$$

assuming that $A_\mathrm{s} < 1$ and $0 \leq A_\mathrm{S}$.

To identify proper values for G_Pf and G_If, a closer look at the linear model has to be taken. Since the influence of relative amplitude changes on the gains is already known, an amplitude of $A_\mathrm{S} = 1$ will be assumed in the following.

To examine the loop stability, the Nyquist stability criterion will be used[23]. Therefore, the phase margin at the unity gain frequency has to be determined. For a control loop to be stable, the phase margin should be as large as possible.

Figure 3.6 shows the phase margin for a range of different G_P and G_I gains. The darker areas are areas of higher phase margins. There seems to be a triangular area, where the phase margin is particularly large. It is safe to assume that all values of G_P and G_I outside of this triangle will lead to an unstable phase locked loop.

3.5.2 Low Level Simulation

Since a real DPLL is not entirely linear, a G_P and G_I gain resulting in a large phase margin in the linear model is not a sufficient criterion for loop stability. Therefore further investigation with a low-level simulation has been performed.

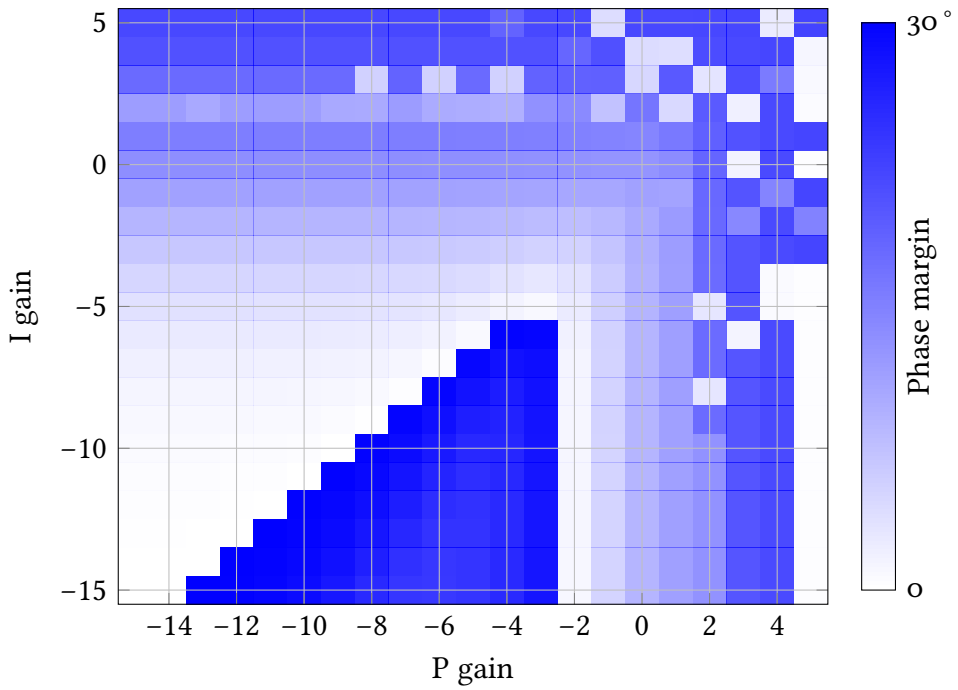


Figure 3.6: Phase margin of a DPLL respect to G_P and G_I gains. The darker an area, the greater the phase margin.

The simulation has been written in C++ and can be found in Appendix A.1.1. It consists of a DPLL that is locked to an NCO. The NCO outputs a sinusoidal signal, whose frequency slowly changes over time. The simulation has been performed multiple times with different G_P and G_I gains and the I output of the phase locked loop has been measured. The result can be seen in Figure 3.7.

Compared to the analysis of the phase margin of the linear model in the previous section, the region where a stable operation is possible is different. This is because of numerous non-linear effects in a low-level simulation that are not respected in the linear model. The examination of these non-linear effects is outside the scope of this thesis.

For a better comparison, both measurements have been put on top of each other in Figure 3.8.

Although the dark areas of both measurements mostly overlap, they are not quite the same. That means that non-linear effects play an important role and should not be neglected in these calculations. The actual gains, where the phase locked loop runs stable and the phase locked loop has enough phase margin lie within the dark overlap of both measurements. Good gain values should be taken

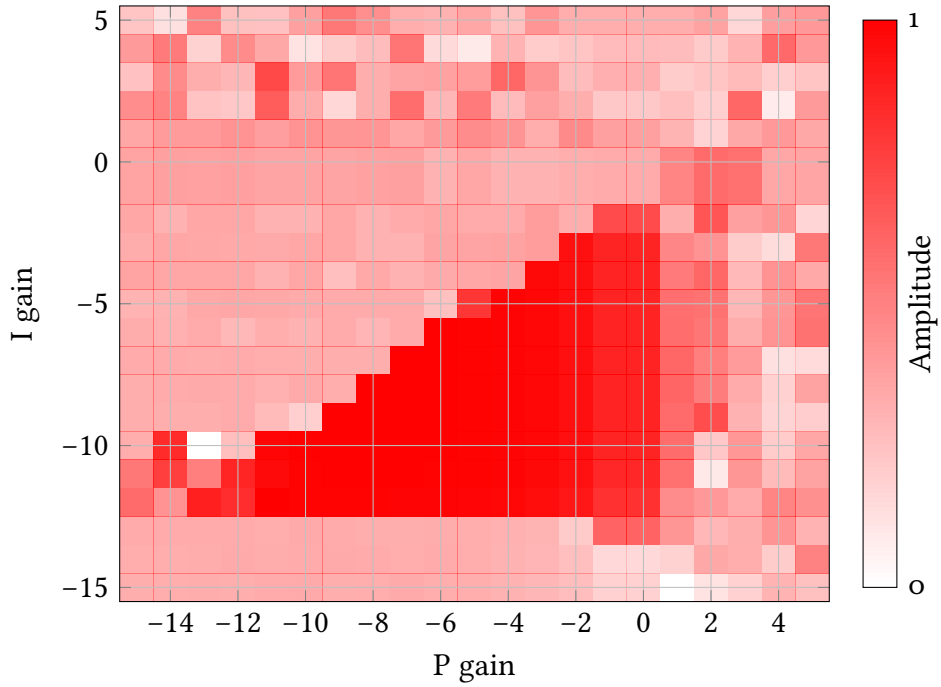


Figure 3.7: Measurement of the I output of a DPLL in a low-level simulation with different G_P and G_I gains. The darker, the greater the I value (arbitrary units).

from this overlapping area. Choosing greater gains results in a higher bandwidth, therefore $G_\mathrm{Pf} = -4$ as well as $G_\mathrm{If} = -8$ have been chosen. These gains will be used in Equation 3.22 in the rest of this chapter.

3.5.3 Bandwidth and Phase Margin

By inserting the calculated gains from the last section into the Equation 3.21 the corresponding Bode plots can be graphed. Figure 3.9 shows the amplitude part of the Bode plot. This can be used to measure the unity gain frequency, at which the amplification is precisely 0 dB:

As it can be seen, the unity gain frequency is approximately 1.45 MHz. This also means, that the bandwidth of the DPLL is 1.45 MHz, which should be plenty to follow a free-running Laser. In the presence of plentiful white noise, this bandwidth might not be enough, but can easily be adjusted if needed. Using this frequency, we can derive the phase margin from the phase part of the Bode plot. This can be seen in Figure 3.10.

Examining the plot at the frequency point of 1.45 MHz, this leads to a phase margin of approximately 42 °. According to the Nyquist stability criterion, the

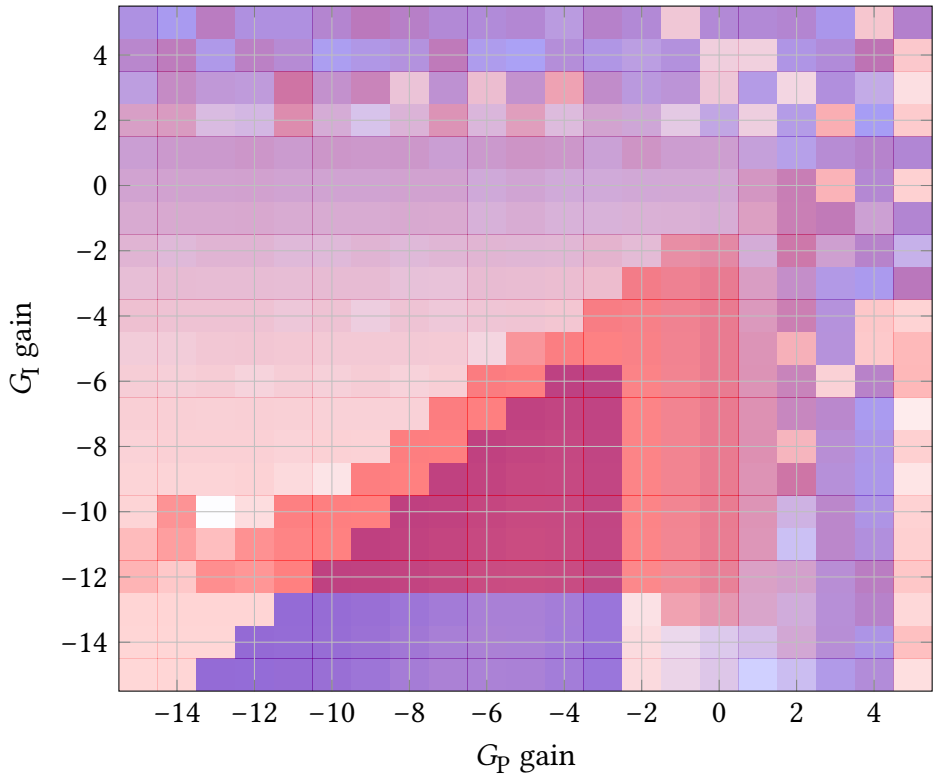


Figure 3.8: Overlay of phase margin calculation and low-level simulation. The overlapping dark area represents the usable gains.

phase margin should be greater than 30 °[24]. Therefore this value should be sufficient for a stable control loop.

Outside a simulation, the real unity gain frequency may be lower due to noise present in the system, e.g. 50 kHz-200 Hz.

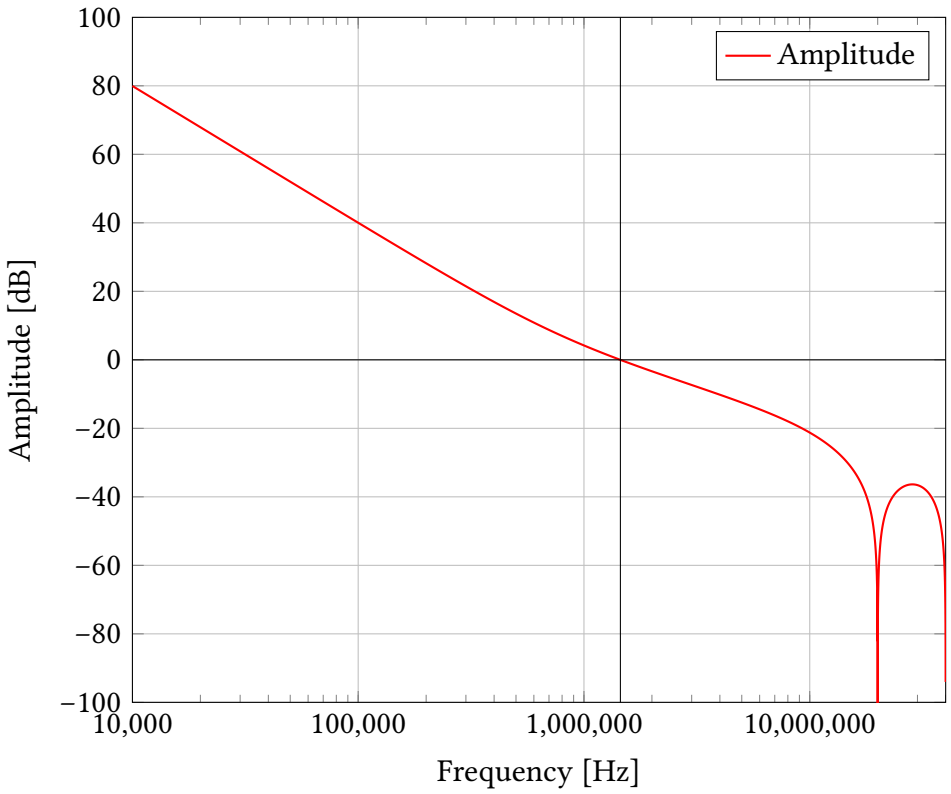


Figure 3.9: The amplitude part of the Bode plots of a phase locked loop using the gains calculated in the previous section.

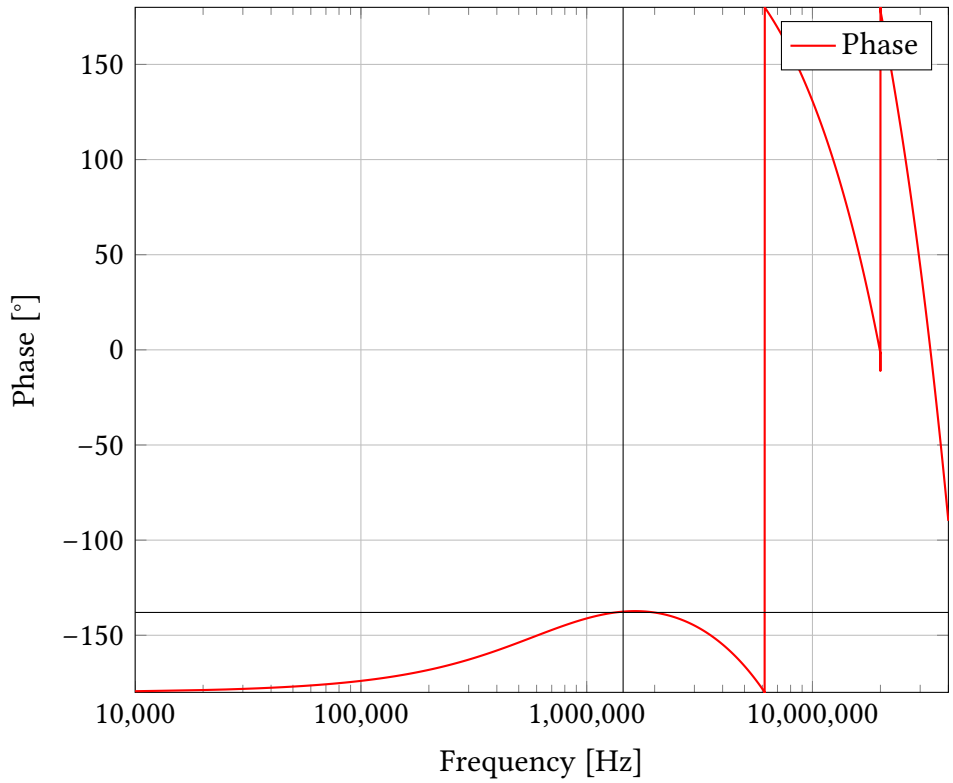


Figure 3.10: The phase part of the Bode plots of a phase locked loop using the gains calculated in the previous section.

3.6 Measurements

To test the beatnote acquisition system, the heterodyne signal of two free-running Non-Planar Ring Oscillator (NPRO) lasers has been connected to the LMS. A DPLL has been locked to this beatnote signal using the beatnote acquisition system, with the FFT running continuously. The experimental set-up can be sen in Figure 3.11.

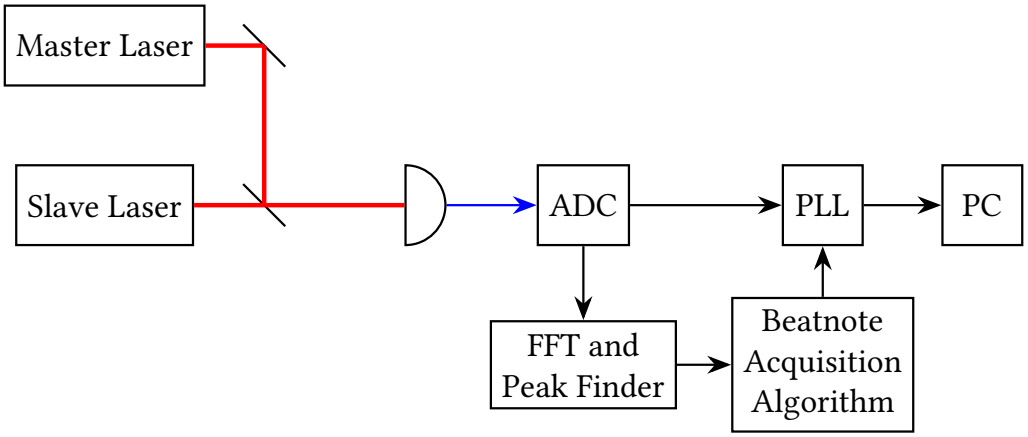


Figure 3.11: Schematic overview of the beatnote acquisition measurement set-up. The red lines denote the path of the laser beam whereas the blue arrows denote analogue electrical signals and the black arrows denote digital signals.

The resulting amplitude and frequency as measured by the FFT, as well as the frequency measured by the DPLL can be seen in Figure 3.12. As it can be seen, the DPLL can successfully lock to the heterodyne frequency and follow it.

More measurements of the beatnote acquisition can be seen in Chapter 4.

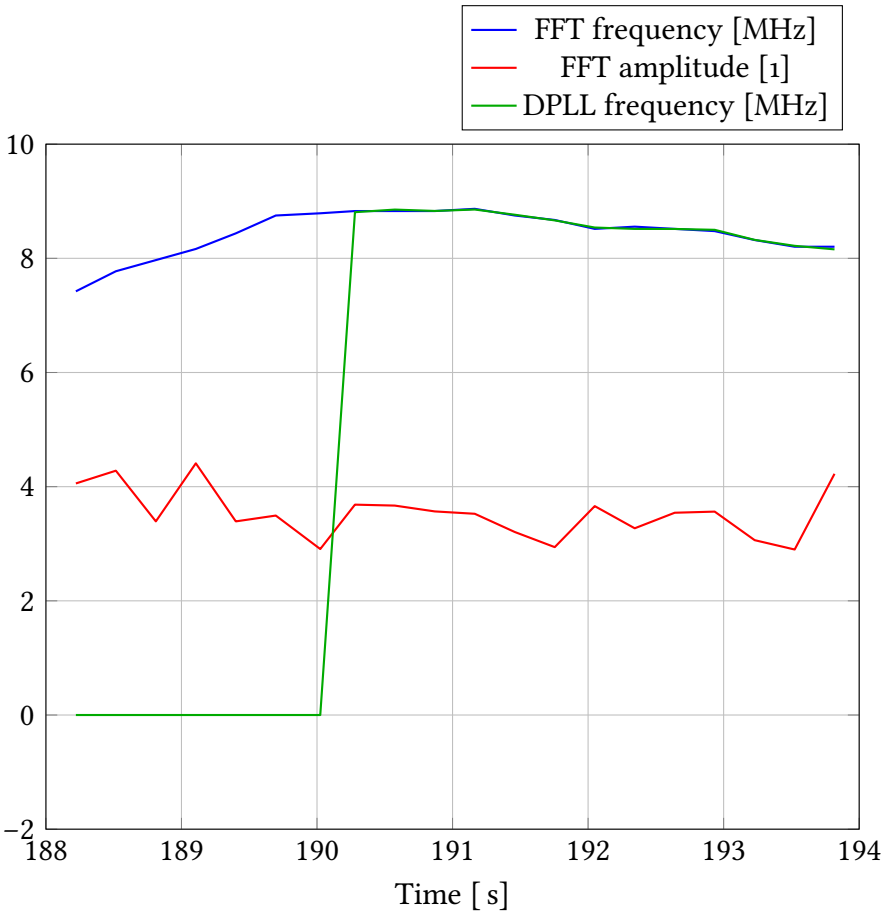


Figure 3.12: The FFT Amplitude (red line) and the FFT Frequency (blue line) of an incoming beatnote signal are used to lock a DPLL (green line).

Chapter 4

Laser Locking

For the phase measurement of the LMS to work, heterodyne interferometry is required. Heterodyne interferometry can be accomplished by two different methods. The first method is using one Laser and an Acoustic-Optic Modulator (AOM) to create a second laser beam with a slightly different frequency[25]. The second method is using two lasers, which are being adjusted in such a way that they have a fixed frequency difference. In both cases, both beams can then interfere with a beam splitter, and the heterodyne frequency can be measured with a photodiode[26].

The current baseline for LISA is to use frequency-locked lasers to establish the heterodyne scheme. Due to varying Doppler shifts between the spacecrafts, a frequency plan has been created that provides the laser lock frequencies to be used at any given time[11]. Since this cannot be accomplished with the first approach, the second method has been chosen to be implemented in the LMS. Having two lasers at a fixed frequency difference is called a laser lock. How this laser lock is accomplished will be discussed in the following sections.

4.1 Traditional approach

Traditionally, a laser lock has been achieved using an analogue Phase Locked Loop (PLL). In this scheme, two free-running lasers are interfered using a beam splitter, creating a heterodyne signal. This heterodyne signal can then be measured with a photodiode. It is mixed with a constant reference frequency, which is usually generated using a signal generator. This generates the sum frequency as well as the difference frequency of both signals. The sum frequency is filtered out using a low-pass filter, and the remaining signal form the error of the PLL. To keep the phase difference between the heterodyne signal and the reference signal at $\frac{2\pi}{4}$, the PLL aims to minimise the error signal. The Q value is processed by a PI controller

generating a suitable actuator signal to achieve this, which is used to actuate one of the two free-running lasers to shift its phase to match the phase of the reference signal. This stabilises the phase of the heterodyne signal at the phase of the constant reference. A schematic representation can be seen in Figure 4.1.

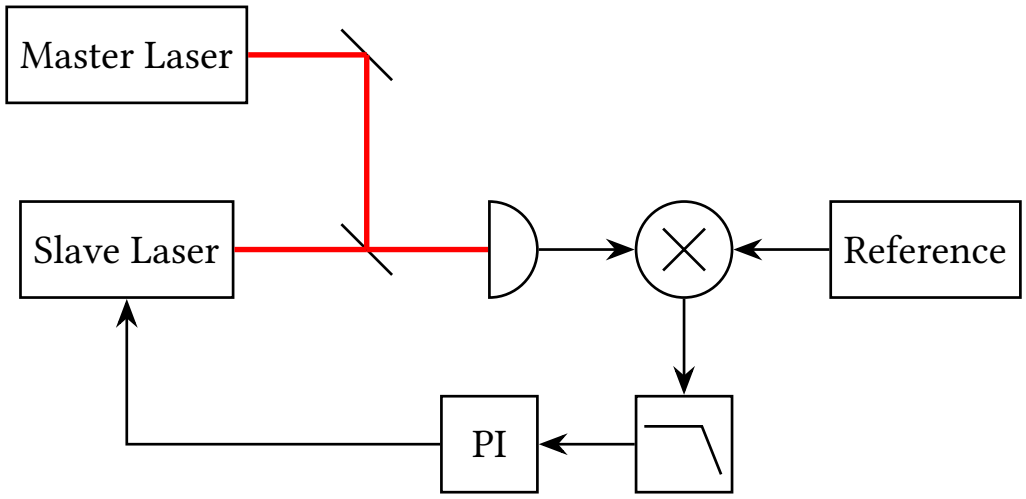


Figure 4.1: Schematic overview of an analogue laser lock. The red lines denote the path of the laser beam whereas the black arrows denote analogue electrical signals. Thee slave laser is controlled by keeping the measured difference phase at a constant target.

There are two significant drawbacks to this approach: Firstly, the frequency of the laser already has to be very close to the reference frequency, otherwise, the PLL will not lock. This is very hard to automate in analogue circuitry. Secondly, the analogue PLL is very prone to cycle slips. They can happen when the phase difference between the reference signal and the heterodyne signal is greater than 180 ° or lower than −180 °. For the PI controller, this looks like a phase difference of the opposite sign and the phase of the heterodyne signal is shifted in the wrong direction.

Therefore this approach is not suitable for LISA. In the following section, a digital frequency lock will be developed instead. It compares the heterodyne frequency and the reference frequency directly instead of its phases. This eliminates the issues of the analogue laser lock.

4.2 Building Blocks

To create a laser lock, the first step is to interfere two free-running lasers using a beam splitter. This creates a heterodyne signal, that can then be measured with a photodiode. The difference frequency of the two lasers must lie within the bandwidth of the photodiode. Since we are using the entirely digital LMS to accomplish the laser lock, the heterodyne frequency also has to lie below the Nyquist frequency of the LMS, which is 40 MHz[10]. Therefore the lasers have to be tuned to a small frequency difference in the order of 20 MHz. This rough tuning is done in software on a microcontroller, which is explained in further detail in Section 4.4. Since the frequencies of free-running lasers tend to drift a lot, the lasers should also have been warmed-up for some time to minimise this drift and allow for an easier lock acquisition.

The heterodyne frequency will then be digitised by one of the ADCs on the LMS. Its frequency and amplitude will be determined using the beatnote acquisition algorithm discussed in Chapter 3. To get a more precise measurement, a DPLL will be locked to the heterodyne frequency. The rough frequency determined by the beatnote acquisition algorithm will be used as starting frequency for the DPLL, and the amplitude will be used to set the corrects gains for the DPLL.

The user has to supply a target frequency difference to which the two lasers shall be locked. This target frequency is then compared to the measured heterodyne frequency. The result of this comparison is called the error value. This value is a measure of the deviation of the current heterodyne frequency from the target.

A controller is then used to calculate the so-called actuator value from the error value. This actuator value is designed in such a way, that the error value gets minimised. The controller has a second actuator output, whose purpose will be described later. The implementation of the controller will be described in full detail in Section 4.3.

Both actuator values will be sent to two DACs, where they are converted back to analogue signals. These analogue signals are used to control one of the two lasers. The laser that is being controlled is called the slave laser, and the laser that is not being controlled is called the master laser since it is still free running and the frequency of the slave laser depends on the frequency of the master laser.

The NPRO laser used in the laboratory experiments can be tuned by either varying the temperature of the laser crystal or by actuating a piezo that slightly changes the geometry of the laser crystal[27]. The first method is used for a coarse adjustment of the laser frequency, whereas the second method is used for fine-tuning the laser frequency[28]. Also, the temperature-based actuation has a bandwidth of under 1 Hz whereas the piezo-based actuation has a bandwidth of up to 30 kHz. The two actuator signals mentioned above are used to actuate the slave laser in both ways.

After being actuated the slave laser will change its frequency accordingly. This leads to a change in the heterodyne frequency of the two lasers. If the controller works correctly, the heterodyne frequency should draw near to the target frequency. This whole laser locking mechanism forms a closed loop, allowing the heterodyne frequency to stabilise very close the target frequency. Even if the frequency of the master laser changes, the slave laser frequency should follow very fast.

A schematic overview of the whole laser lock can be seen in Figure 4.2.

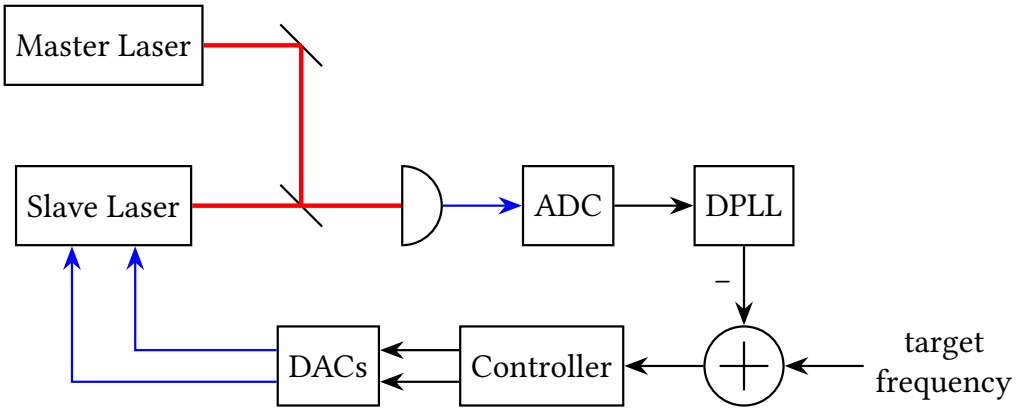


Figure 4.2: Schematic overview of a laser lock. The red lines denote the path of the laser beam whereas the blue arrows denote analogue electrical signals and the black arrows denote digital signals. The slave laser is controlled by keeping the measured difference frequency at a constant target.

As is can be seen, the path of the various optical and electrical signals form a closed loop. Therefore this setup is also called a control loop.

4.3 Laser Lock Controller

The laser lock controller transforms the error signal produced by the subtraction of the current frequency and the target frequency into two actuator signals tailored to minimise the error signal.

In control theory, there are four different base types of controllers, from which any other linear time-invariant controller can be constructed[29]:

- Bang-Bang controller
- Proportional controller

- Integral controller
- Derivative controller

In the laser lock controller, only the proportional and the integral controller types are used. When used together these two are called a PI controller.

The proportional controller works by multiplying its input signal by some constant factor $\varkappa_\mathrm{P}$. This factor is also called the gain. The transfer function $H_\mathrm{P}(z)$ of such an proportional controller with the gain $\varkappa_\mathrm{P}$ can be written as:

$$H_\mathrm{P}(z) = \varkappa_\mathrm{P} \,. \tag{4.1}$$

If the gain is $\varkappa_\mathrm{P} = 1$, the proportional controller does not change the signal. This situation is called unity gain, since $|H| = 1$.

The integral controller works by first integrating its input signal over time and then multiplying the result with a gain $\varkappa_\mathrm{I}$. The transfer function $H_\mathrm{I}(z)$ of such an integral controller with the gain $\varkappa_\mathrm{I}$ can be written as:

$$H_\mathrm{I}(z) = \varkappa_\mathrm{I} \frac{z^{-1}}{1 - z^{-1}} \,. \tag{4.2}$$

The unity gain frequency of an integral controller is at $z = \varkappa_\mathrm{I} + 1$:

$$\begin{aligned} H_\mathrm{I}(z) &= \varkappa_\mathrm{I} \frac{(\varkappa_\mathrm{I} + 1)^{-1}}{1 - (\varkappa_\mathrm{I} + 1)^{-1}} \\ &= \frac{\varkappa_\mathrm{I}}{(\varkappa_\mathrm{I} + 1)(1 - \frac{1}{\varkappa_\mathrm{I}+1})} \\ &= \frac{\varkappa_\mathrm{I}}{(\varkappa_\mathrm{I} + 1) - 1} \\ &= \frac{\varkappa_\mathrm{I}}{\varkappa_\mathrm{I}} = 1 \,. \end{aligned} \tag{4.3}$$

regardless of the value of $\varkappa_\mathrm{I}$.

When combining these two controllers to form a PI controller, the transfer functions are added:

$$H_\mathrm{PI}(z) = \varkappa_\mathrm{P} + \varkappa_\mathrm{I} \frac{z^{-1}}{1 - z^{-1}} \,. \tag{4.4}$$

That means that the input signal is fed to both controllers at the same time and the results of both controllers are added together.

However, in this implementation, an additional delay is added to the proportional controller in such a way that both paths through the PI controller have the

same delay. This is done by multiplying the proportional part t with z^{-1}. This leads to the following equation:

$$H'_{\mathrm{PI}}(z) = \varkappa_{\mathrm{P}} z^{-1} + \varkappa_{\mathrm{I}} \frac{z^{-1}}{1 - z^{-1}} = \left(\varkappa_{\mathrm{P}} + \frac{\varkappa_{\mathrm{I}}}{1 - z^{-1}}\right) z^{-1} . \tag{4.5}$$

A schematic overview of a PI controller can be seen in Figure 4.3.

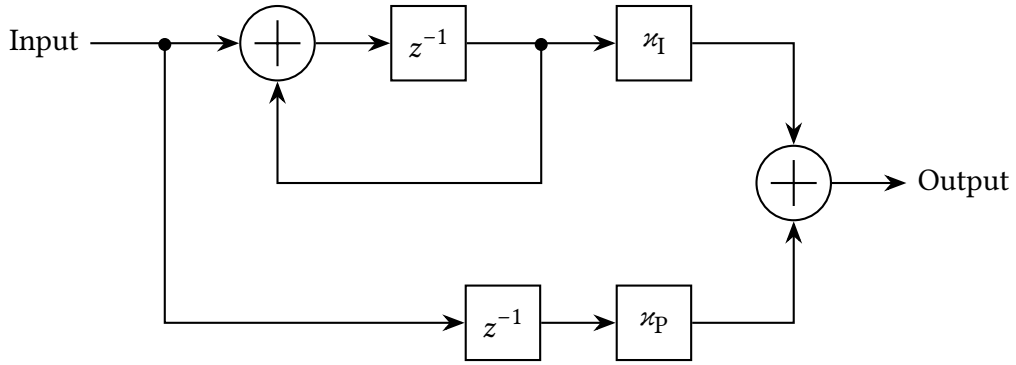


Figure 4.3: Schematic overview of a PI controller used in the laser lock controller

The laser lock controller uses two PI controllers to generate the two actuator signals for the piezo input and the temperature input of the slave laser. The first PI controller uses the error signal to generate the actuator signal for the piezo input of the laser. Hence it is also called piezo controller. Since the piezo input only has a small dynamic range and the temperature input, on the other hand, has a very wide dynamic range[28], the actuator signal for the piezo shall be kept near zero. Therefore, the output of the piezo controller can directly be used as an error signal for the second PI controller[1]. The output of the second PI controller can then be used as the actuator signal for the temperature input of the slave laser. Hence it is also called temperature controller. Whenever the piezo actuator signal gets too large, it will be compensated by the temperature controller by adjusting the temperature actuator signal. A schematic overview of the laser lock control circuit can be seen in Figure 4.4.

The transfer function of the piezo controller is identical to Equation 4.4:

$$H_{\mathrm{pzt}}(z) = \left(\varkappa_{\mathrm{P}_{\mathrm{pzt}}} + \frac{\varkappa_{\mathrm{I}_{\mathrm{pzt}}}}{1 - z^{-1}}\right) z^{-1} , \tag{4.6}$$

where $\varkappa_{\mathrm{P}_{\mathrm{pzt}}}$ is the gain of the proportional controller for the piezo and $\varkappa_{\mathrm{I}_{\{mathrmpzt}}$ is the gain of the integral controller for the piezo. For the temperature controller

[1] As if it were compared against zero.

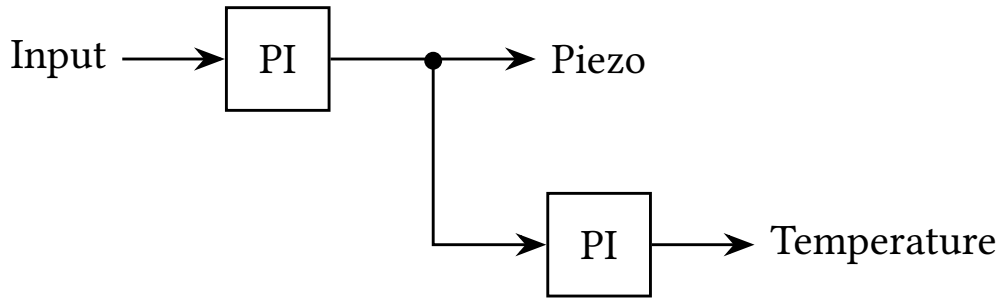


Figure 4.4: Schematic overview of the PI controller arrangement in the laser lock controller

the transfer function can be written as a concatenation of two PI controllers:

$$H_{\text{tmp}}(z) = \left(\varkappa_{\text{P}_{\text{pzt}}} + \frac{\varkappa_{\text{I}_{\text{pzt}}}}{1 - z^{-1}}\right) z^{-1} \left(\varkappa_{\text{P}_{\text{mpt}}} + \frac{\varkappa_{\text{I}_{\text{tmp}}}}{1 - z^{-1}}\right) z^{-1}, \tag{4.7}$$

where $\varkappa_{\text{P}_{\text{tmp}}}$ is the gain of the proportional controller for the temperature and $\varkappa_{\text{I}_{\text{tmp}}}$ is the gain of the integral controller for the temperature.

For an overview how this laser lock controller fits into the bigger picture see Figure 4.2.

4.3.1 Gains

Due to the much lower bandwidth of the temperature actuator in comparison to the piezo actuator, the gains of the temperature controller are much lower than the gains of the piezo controller. The gains in Table 4.1 have been found to work reliably.

	$\varkappa_{\text{P}}$	$\varkappa_{\text{I}}$
First PI controller	−5	−1
Second PI controller	−7	−4

Table 4.1: Gains for the individual PI controllers of the Laser Lock.

By inserting these gains into the transfer functions, the corresponding Bode plots can be graphed. They can be seen in Figure 4.5 and Figure 4.6. The laser lock controller shows a clean integrator-type response and doesn't show a significant phase drop until about 10 MHz. This should not affect the bandwidth of the laser lock, which mainly originates from the bandwidth of the DPLL as the frequency sensor as well as the bandwidth of the piezo and temperature actuator in the laser.

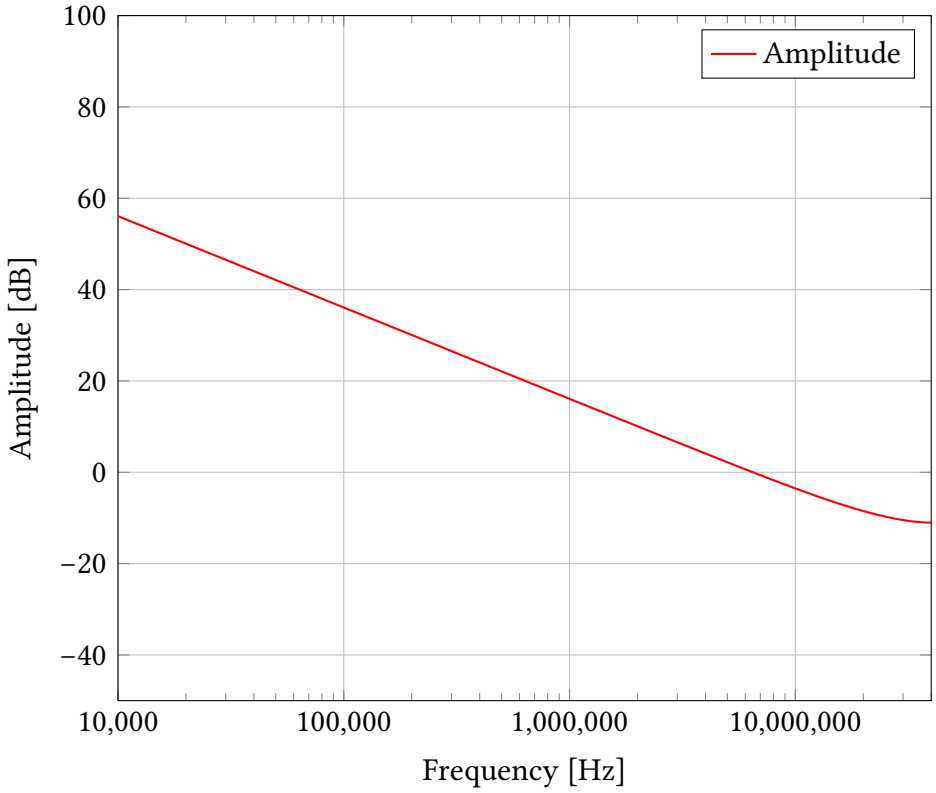


Figure 4.5: The amplitude part of the Bode plots of the laser lock controller.

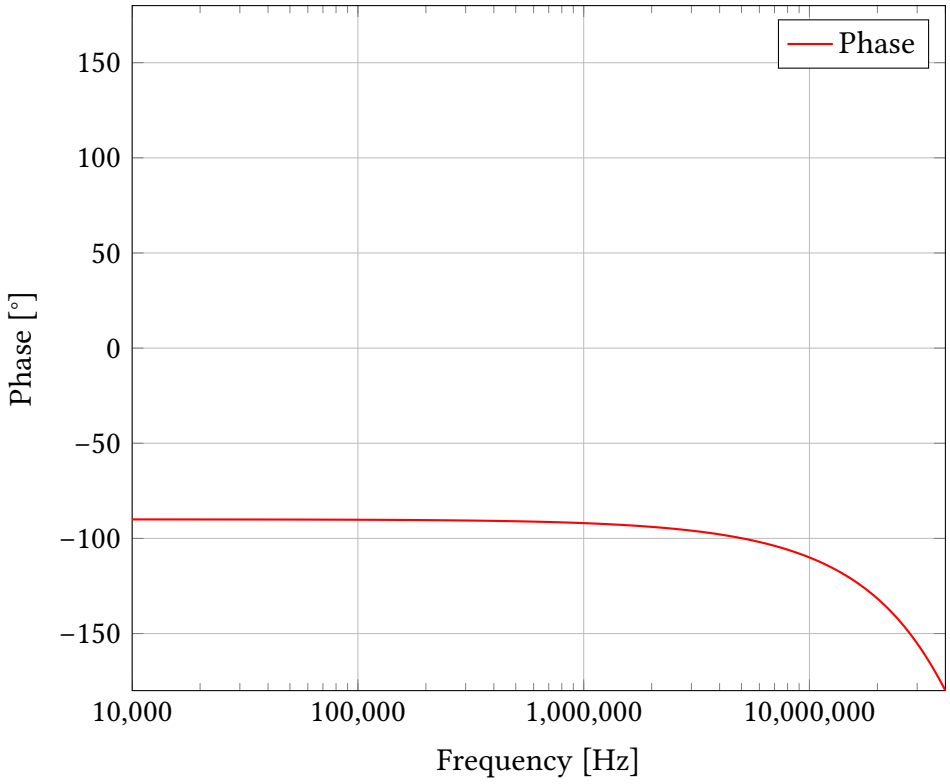


Figure 4.6: The phase part of the Bode plots of the laser lock.

4.4 Automatic Algorithm

As stated in Section 4.2, the heterodyne frequency must be within the bandwidth of the photodiode and below the Nyquist frequency of the LMS, to successfully establish a laser lock. Since this is not always the case and there might not be the possibility to manually adjust the frequency of the slave laser in the future, there is the need for an automatic algorithm to acquire a proper heterodyne signal.

To fulfil this requirement, an algorithm in the form of an FSM has been developed to accomplish this task. This FSM algorithm does not rely on any previous adjustments of the laser heterodyne frequency and only expects two free-running lasers, one of which can be controlled by the LMS.

The FSM is roughly divided into five parts:

- Temperature scan
- Temperature set
- Piezo adjustments
- Lock
- Check

These parts will be described in more detail in the following sections.

4.4.1 Temperature Scan

The temperature scan is the first stage after the LMS is powered up and has initialised itself. A given temperature range will be scanned on the slave laser while the beatnote frequency and amplitude are observed using the FFT from Chapter 3. This has to be done slowly, because of the low bandwidth of the temperature actuator of a few Hz. During the temperature scan, the temperature with the highest beatnote amplitude will be determined. This is the point where the beatnote frequency lies within the bandwidth of the photodiode, which is 100 MHz is this experimental set-up, and below the Nyquist frequency of the LMS.

The temperature scan has two parameters, which are the begin and the end of the temperature range that should be scanned. This range has to be chosen in such a way that the temperature, at which the beatnote frequency lies within the measurement bandwidth, is within this temperature range. The more extensive this temperature range is, the longer the scan takes. Therefore for testing purposes in the context of this thesis, a rather small range of approximately ±0.5 °C has been chosen.

A schamatic overview can be seen in Figure 4.7.

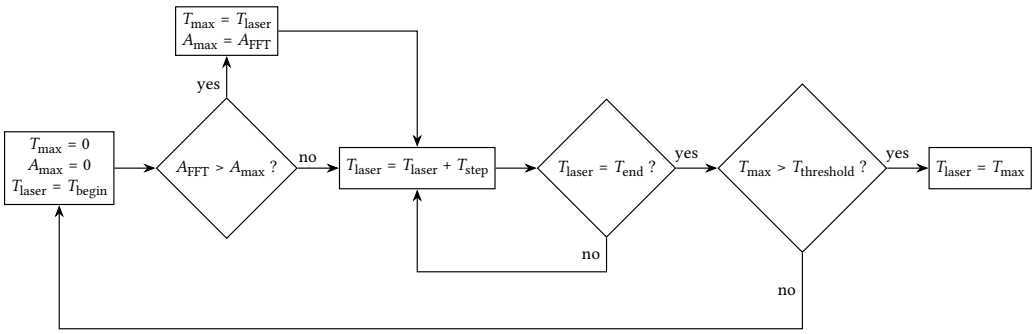


Figure 4.7: A temperature range is scanned for the maximum FFT amplitude. If it is over a given threshold, the Laser is set to the corresponding temperature.

4.4.2 Temperature Set

After the temperature scan has completed the whole range, the found amplitude will be compared against a user set threshold. If the amplitude is higher than the threshold, the corresponding temperature will be set at the slave laser. Otherwise, the temperature scan will start from the beginning.

After the temperature value has been set, the phase meter will wait for 3 s to let the temperature settle and the beatnote frequency stabilise. The beatnote frequency should now lie within the measurement bandwidth of the phase meter. If not, the temperature scan will start from the beginning.

4.4.3 Piezo Adjustments

After making sure that the beatnote frequency lies within the measurement bandwidth of the phase meter, finer adjustments have to be performed. The beatnote frequency has to be shifted near the desired locking frequency using the piezo in the slave laser. To determine how the voltage has to be changed to achieve a particular change in beatnote frequency, the piezo voltage is increased by approximately 0.1 V while observing changes in the beatnote frequency. If the beatnote frequency increases, the coefficient between voltage and frequency is positive, otherwise it is negative. Whether the voltage of the piezo has to be lowered or increased to shift the heterodyne frequency in a given direction will be memorised for later use.

At this point, the sign of the current frequency is also determined. This is done by XNORing the direction of the change in voltage with the direction of the change in heterodyne frequency. If both directions are the same, the heterodyne frequency is positive. Otherwise, it is negative. The signedness of the heterodyne frequency is essential for the LISA heterodyne frequency plan[11].

After that, the target frequency will be approached step by step. In that process, the sign of the target frequency is taken into account. The beatnote frequency should then be within approximately 1 MHz of the target frequency. If these adjustments fail, the process will be retried from the temperature set step. After repeated fails, the temperature scan will be re-initiated.

A schamatic overview can be seen in Figure 4.8.

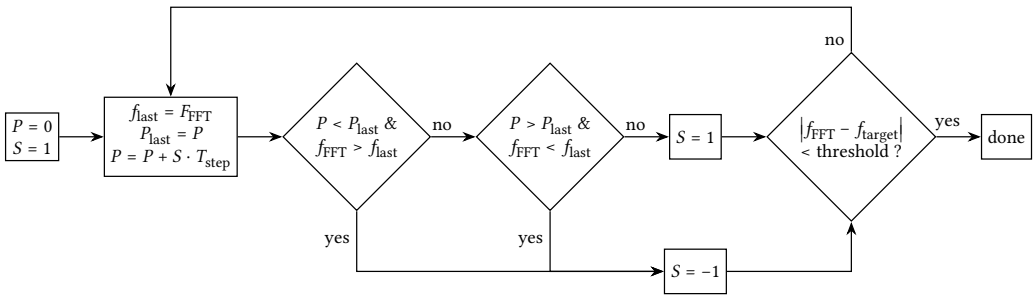


Figure 4.8: Calculate the signedness of the piezo. Draw the heterodyne frequency near the target frequency.

4.4.4 Lock

After the beatnote frequency has been brought near enough to the target frequency, a DPLL is locked to the beatnote frequency using the frequency and gains from the beatnote acquisition algorithm from Chapter 3. Once the DPLL has successfully established a lock, which means the frequency output of the DPLL is close to the frequency measured by the FFT, the laser lock controller is turned on and should lock the slave laser to the master laser in a small amount of time. In case of failure, the process will be restarted at the point of setting the temperature. After repeated fails, the temperature scan will be re-initiated.

4.4.5 Check

Once the laser lock has been established, it will regularly be checked for validity. This is done by comparing the frequency output of the DPLL with the frequency measured by the FFT. If their difference is greater than 0.5 MHz, the lock will be re-initiated.

A schematic overview of this algorithm can be found in Figure 4.9.

The source code for the automatic lock acquisition algorithm can be found in Section A.3.1.

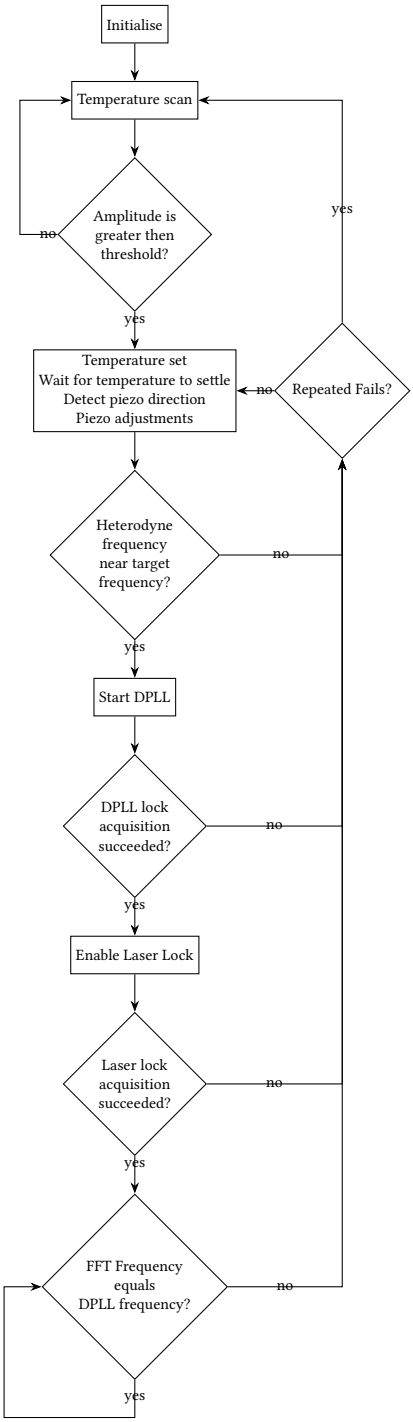


Figure 4.9: Schematic overview of the laser lock FSM.

4.5 Measurements

In this section, various phase meter signals from different acquisition phases will be shown and further analysed. This will also demonstrate the correct function of the beatnote acquisition system.

4.5.1 Temperature Scan

During the temperature scanning phase, the temperature signal of the slave laser is slowly monotonically increased from a predefined lower bound to a predefined upper bound. This can be seen as a green line in Figure 4.10. While the temperature of the master laser is raised, its frequency and therefore also the frequency difference between the master laser and the slave laser changes. This frequency difference is measured by the FFT in the beatnote acquisition system and is depicted by the blue line. Alongside the frequency of the input signal, its amplitude is also measured by the FFT in the beatnote acquisition system, which can be seen as a red line in the figure mentioned above. The closer the frequency difference draws to zero, the higher its measured amplitude gets due to the limited bandwidth of the photodiode, the LMS and other components. Whenever the frequency difference is outside of the bandwidth of the phase meter or the photodiode, the FFT does not measure anything useful anymore, which translates to random frequency changes and a near zero amplitude. This makes the detection of a useful heterodyne signal very easy.

4.5.2 Temperature Set

After a useful heterodyne signal has been found in the previous step, its corresponding frequency is set. This change in the temperature signal of the slave laser is mostly a step function and results in some ringing in the frequency of the slave laser. Therefore, the phase meter will wait a few seconds until the difference frequency stabilised itself at a value of a few MHz. This can be seen in Figure 4.11.

4.5.3 Piezo Adjustments

Once the difference frequency is inside the bandwidth of the photodiode and the LMS, the piezo signal of the slave laser will be used to bring the difference frequency as close as possible to the target locking frequency, which is 9 MHz in this case. The piezo signal activity is depicted as a yellow line in Figure 4.12. As it can be seen, the difference frequency changes proportionally to the piezo signal.

4.5.4 Lock

After the difference frequency has been brought near the target frequency, a DPLL is locked to this frequency. The frequency measured by the DPLL is depicted by the black line in Figure 4.13. As it can be seen the lock of the DPLL is acquired very fast, and its measured frequency is almost identical to the frequency measured by the FFT. The small difference between both measured frequencies is mainly due to the limited precision of the frequency measurement of the FFT. Once the DPLL has acquired a proper lock, both laser lock PI controllers are turned on, and the difference frequency stabilises quite fast at the target lock frequency.

As it can be seen in Figure 4.14 the lock controlling the laser piezo reacts much faster than the lock controlling the temperature of the laser. On the other hand, the piezo signal has only a limited actuator range. Therefore, larger offsets are being compensated by the temperature signal, to keep the piezo signal near zero.

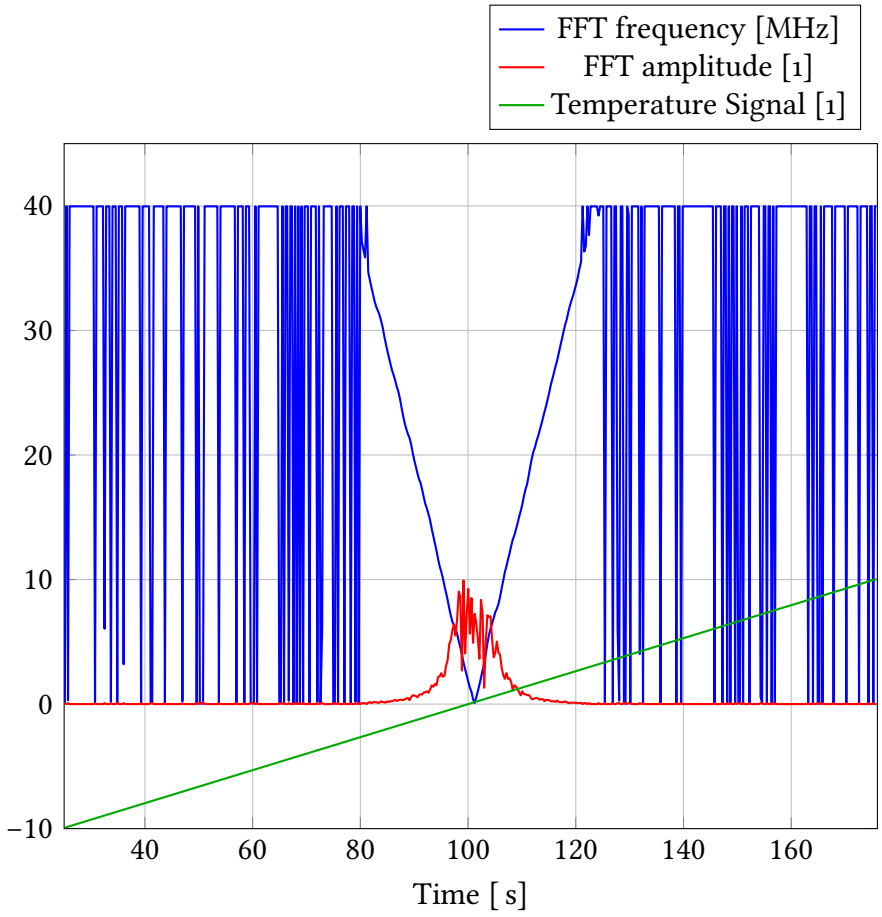


Figure 4.10: The temperature scanning phase of the automatic laser locking algorithm. The green curve shows the slow increasing of the temperature actuator signal of the slave laser. The blue and red coloured curve show the frequency and amplitude measured by the FFT in the beatnote acquisition system.

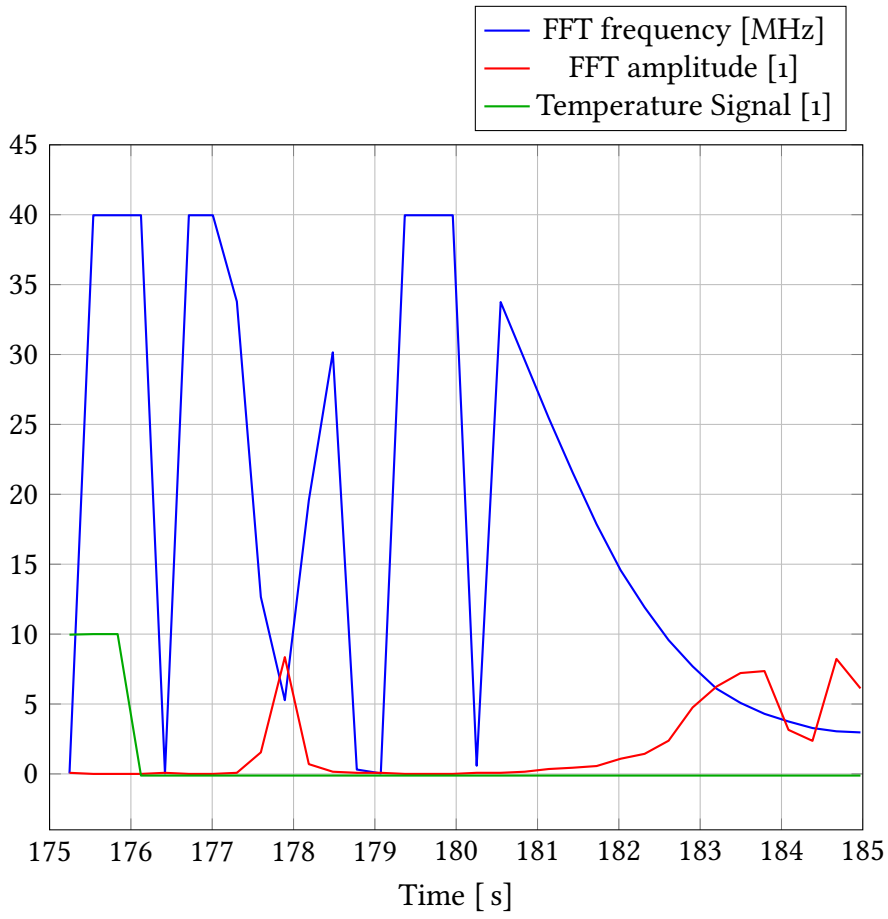


Figure 4.11: When the temperature is set, it behaves like a step function (green line). This results in a lot of ringing in the frequency difference (blue line), which will eventually settle at a usable heterodyne frequency.

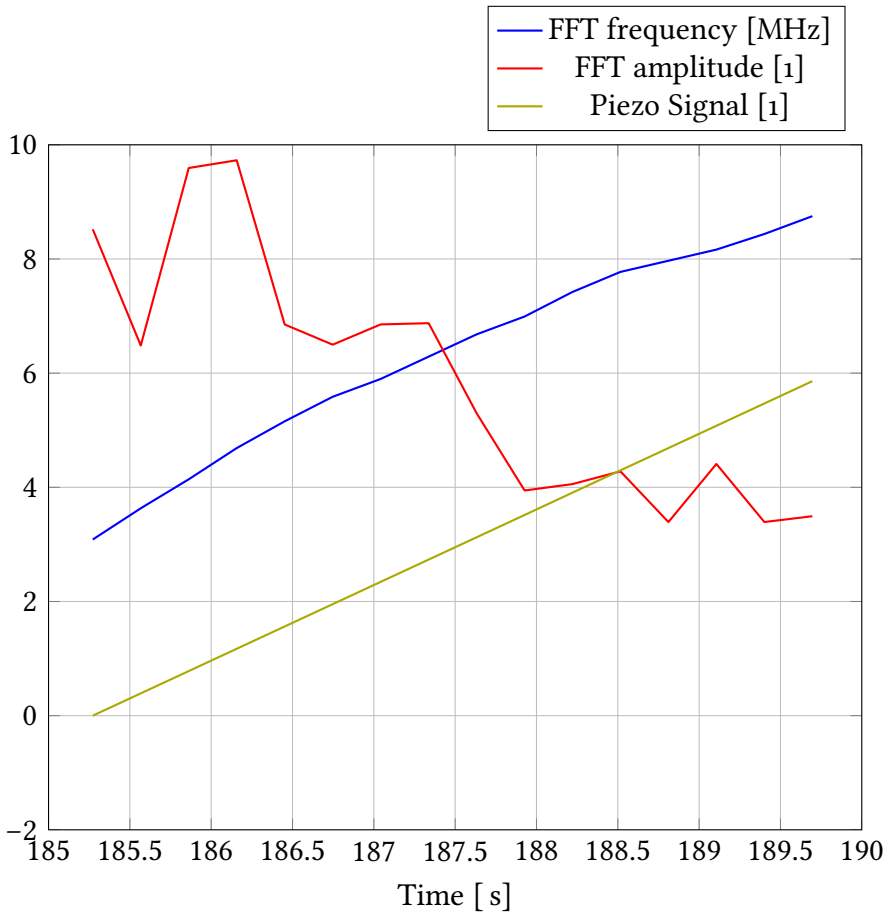


Figure 4.12: The piezo signal (yellow line) is adjusted to bring the frequency difference (blue line) as close as possible to the target locking frequency, which is 9 MHz in this case.

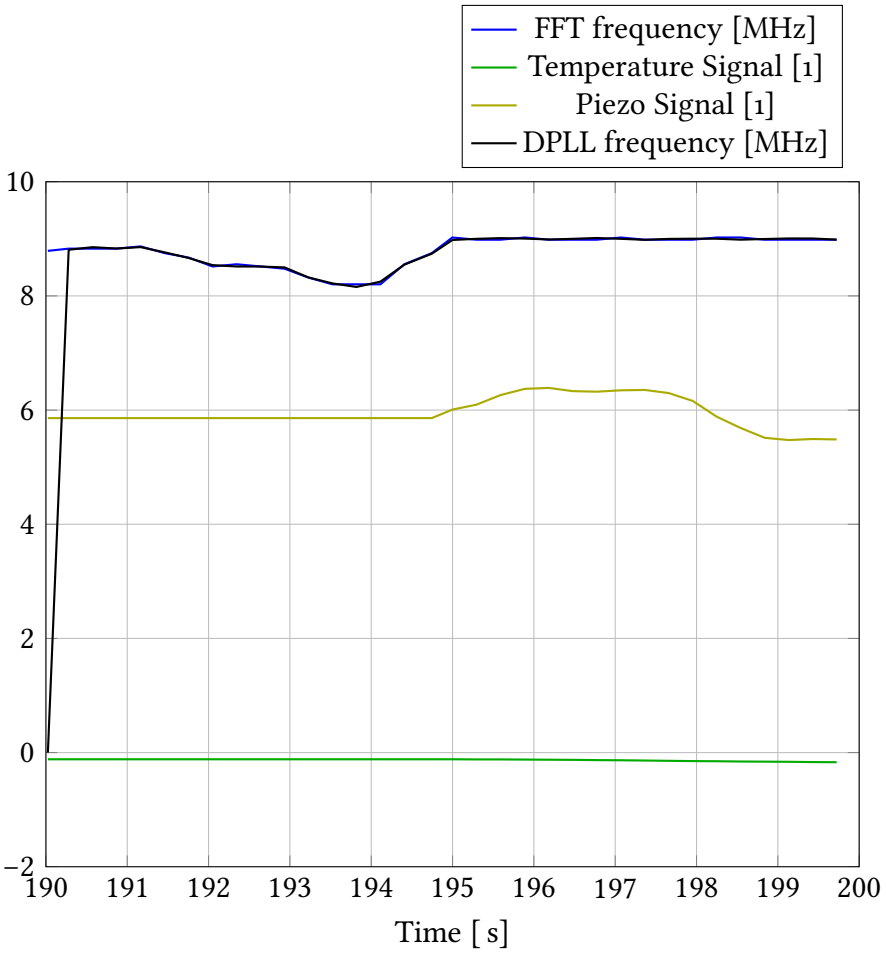


Figure 4.13: First the DPLL is locked to the difference frequency (black line) and then the laser locks for the piezo and temperature control of the laser are turned on to keep the difference frequency at the target frequency.

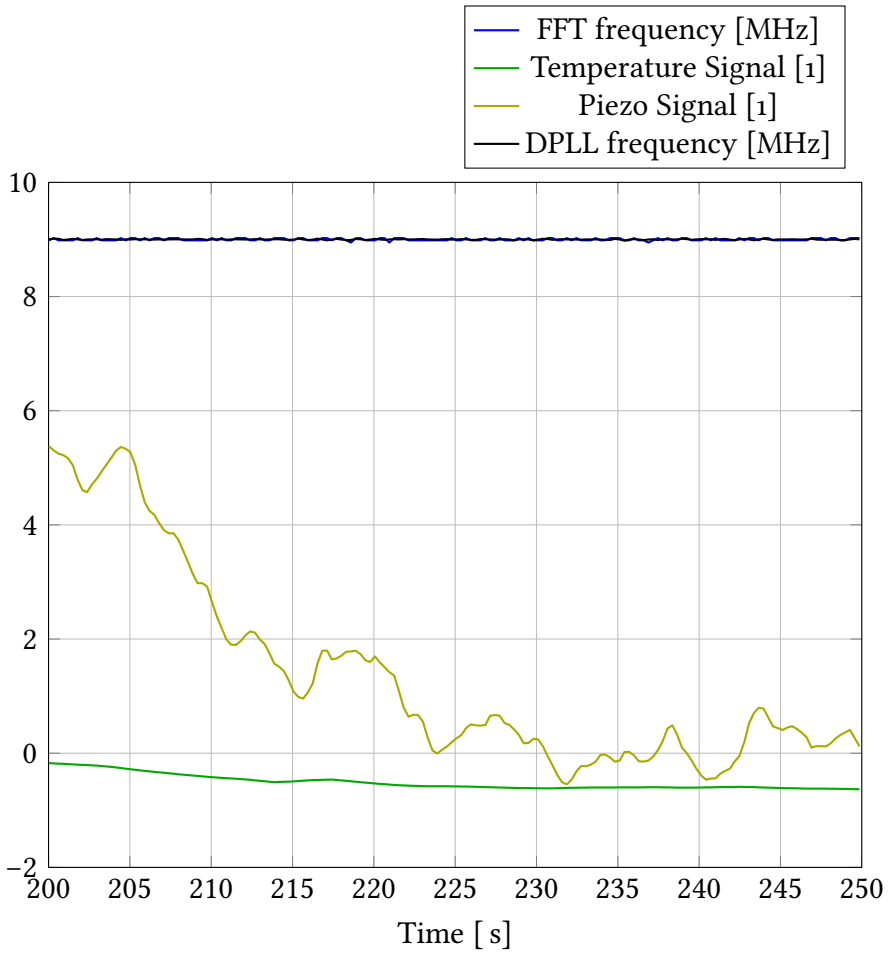


Figure 4.14: The temperature signal keeps the piezo signal near zero.

4.6 Performance

To have a look at the performance of the laser lock, the deviation of the measured lock frequency from the target lock frequency has been plotted. This can be seen in Figure 4.15.

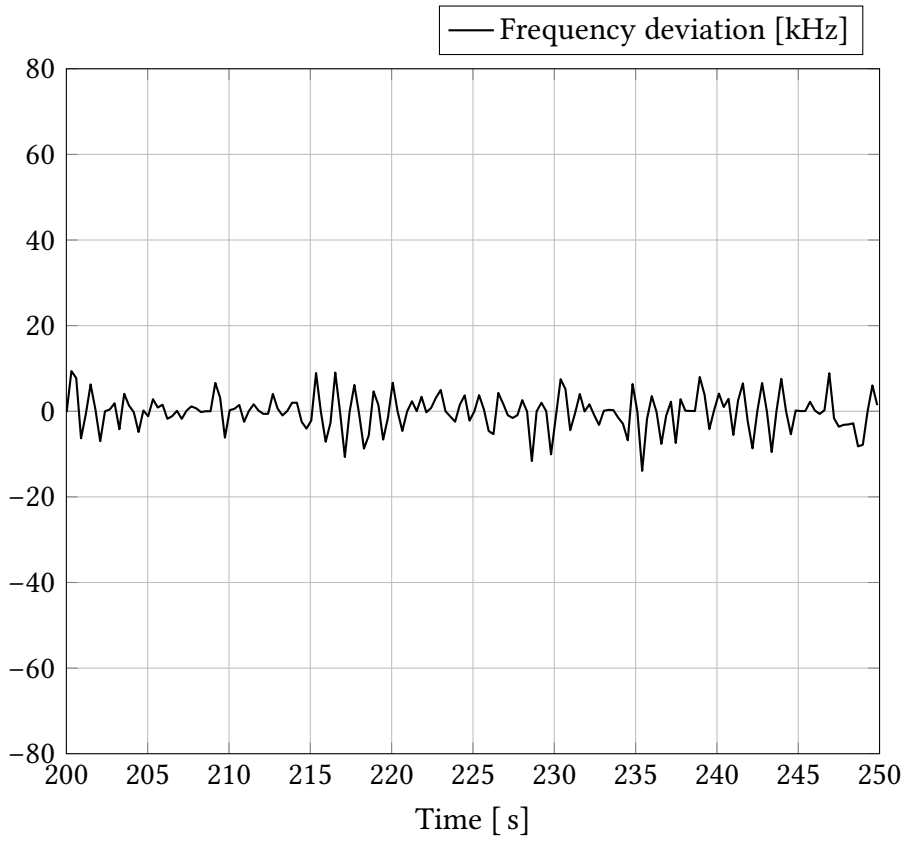


Figure 4.15: Performance of the laser lock: Difference between measured lock frequency and target lock frequency.

As it can be seen, the difference between the measured lock frequency and the target lock frequency is at all times less than 10 kHz, which is about 0.1 % of the target lock frequency of 9 MHz. On average the deviation is even less than 5 kHz, which corresponds to about 0.05 %.

The spectrum of this signal can be seen in Figure 4.16. As expected from a mostly constant signal, there is a reasonably large DC part. Also, the amplitude at the Nyquist frequency is higher than the average, but this does not seem to affect the performance of the laser lock.

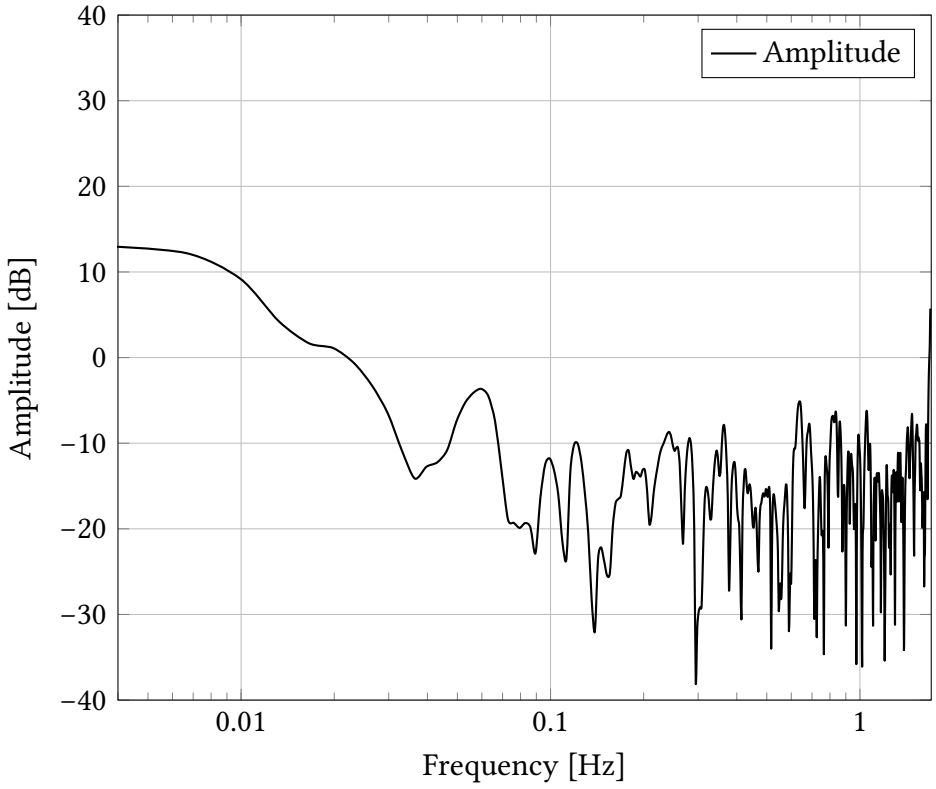


Figure 4.16: Performance of the laser lock: Spectrum of the difference between measured lock frequency and target lock frequency.

The laser lock could be held in its locked state for multiple weeks. Therefore this is a very stable lock.

Chapter 5

Automatic Gain Control

If the amplitude of an incoming signal changes significantly over time, the gains of the DPLL have to be slowly adjusted according to the current amplitude of the input signal. This is called AGC. AGC has traditionally been used in Amplitude Modulation (AM) radio receivers to adapt to changing signal strength[30], which is what is needed here.

Due to the binary logarithmic nature of the DPLL gains in the current implementation of the DPLL (see Equation 3.22), these will only be adjusted if the input amplitude doubles or halves. Fortunately, as the following measurements show, this does not seem to be a problem. Otherwise, there would also be the possibility to implement more fine-grained control of the gains. The amplitude of the incoming signal can be obtained in two ways, either from the FFT described in the previous chapter or from the I value of the DPLL. Both methods will be looked at in the following.

5.1 FFT Amplitude

The FFT has been used in the previous chapter to calculate the initial gains for the DPLL. Unfortunately, the amplitude calculated by the FFT heavily depends on the input frequency. If the signal frequency lies precisely between two FFT frequency bins its measured amplitude is halved in comparison to the measured amplitude of a signal which frequency lies precisely in the middle of a frequency bin. This is because the signal power is distributed amongst both frequency bins. This effect is illustrated in Figure 5.1

As it can be seen, the position of a frequency relative to the frequency bins of an FFT spectrum has a significant effect on the measured amplitude of the peak as well as on the form of the spectrum. This could lead to random fluctuations in the gains and potential performance issues. When applying a flat-top window function for

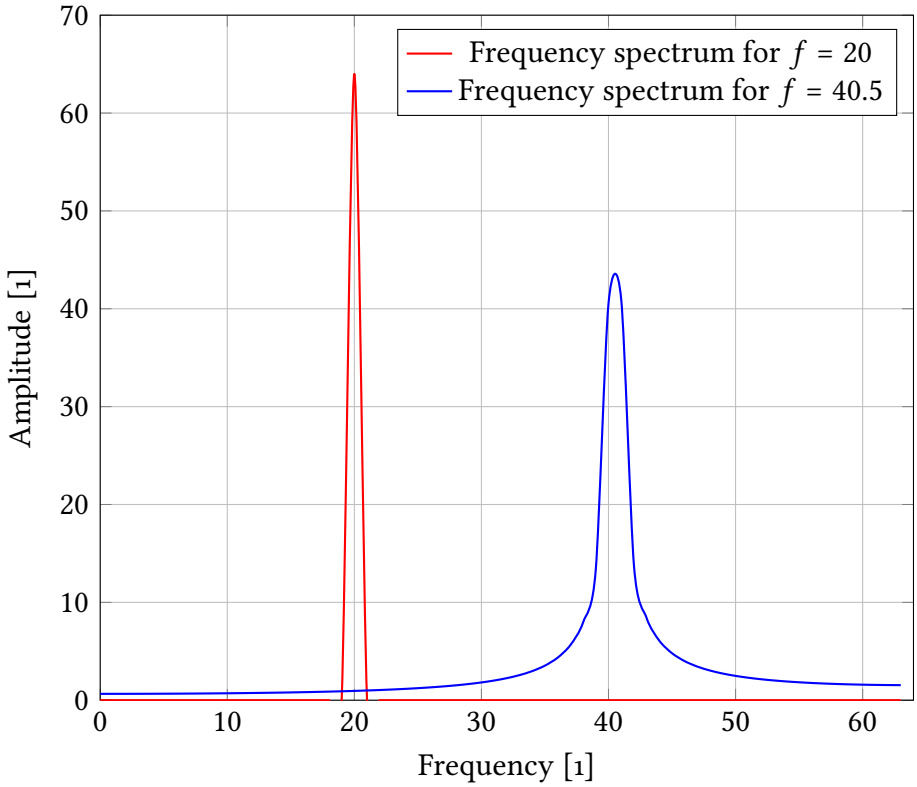


Figure 5.1: The frequency of the red spectrum lies in the middle of a frequency bin, the frequency of the blue spectrum lies in the middle between two frequency bins. This has a huge effect on the measured amplitude.

the FFT input signal, this effect can be reduced[31]. Also, the amplitude loss due to the offset from the bin centre is deterministic and could be corrected using an appropriate algorithm. However, both methods cannot be easily implemented in the LMS. Therefore the FFT cannot reliably be used to perform continuous adjustments of the DPLL gains.

5.2 Phase Locked Loop I Value

On the other hand, the I value of the DPLL is not frequency dependent and will, therefore, be used in the following.

The current gains G_P and G_I can be calculated using the I value and the full amplitude gains G_Pf and G_If calculated in Section 3.5:

$$G_{\mathrm{P}} = G_{\mathrm{Pf}} + \left\lfloor \log_2\left(\frac{1}{I}\right) \right\rfloor$$
$$G_{\mathrm{I}} = G_{\mathrm{If}} + \left\lfloor \log_2\left(\frac{1}{I}\right) \right\rfloor . \tag{5.1}$$

This result is similar to Equation 3.22 with the difference, that the I value from the DPLL is not squared in contrast to the amplitude value of the FFT which eliminated the need for an additional square root.

To simplify the above design, instead of recalculating each gain from the current amplitude, a common additional gain can be computed. This additional gain G has the following form:

$$G = \left\lfloor \log_2\left(\frac{1}{|A|}\right) \right\rfloor . \tag{5.2}$$

With this additional gain, the G_{P} and G_{I} gains could stay fixed at their full-amplitude values G_{Pf} and G_{If}, and only the new pre-gain needs to be modified at runtime.

This method works because of the properties of the logarithm. Whenever the absolute Amplitude $|A|$ halves, its inverse $\frac{1}{|A|}$ doubles. Therefore, when the argument of the logarithm to the base two doubles, its result increased by one:

$$\log_2(2x) = \frac{\ln(2x)}{\ln(2)} = \frac{\ln(2) + \ln(x)}{\ln(2)} = 1 + \frac{\ln(x)}{\ln(2)} = \log_2(x) + 1 . \tag{5.3}$$

5.2.1 Additional Gain Calculation

After the startup of the AGC algorithm, it will wait for a 1 ms to let the I value of the DPLL settle. The currently set gains for the DPLL are assumed to be the correct gains for the current amplitude. Therefore the current amplitude is saved. All further gain calculations will use this amplification as a reference.

Unfortunately, equation 5.2 cannot be implemented directly in VHDL. To calculate the additional gain from the I value, first, its absolute value is taken. Then the leading zeros of the two's complement representation are counted[32]. From this, the amount of leading zeros of the two's complement of the reference amplitude is subtracted. The resulting value is then used as the additional gain.

5.2.2 Applying the Additional Gain

There are three possible ways to apply the additional gain G to the system:

- Apply directly to the input signal just before the phase detector.

- Apply directly to the error signal just after the phase detector and before the PI controller.
- Add to the G_P and G_I gains.

All of these ways are equivalent because the phase detector is a multiplier just like a gain. Multiplication is a linear operation and therefore commutative[33]. Hence, the order of the multiplications does not matter in principle. However, when using finite precision arithmetic, the multiplication order does matter in practice. In this thesis, the additional gain will be added to the G_P and G_I gains, because this is the most straightforward way to implement. The current gains can, therefore, be calculated with:

$$\begin{aligned} G_P &= G_{Pf} + G \\ G_I &= G_{If} + G. \end{aligned} \tag{5.4}$$

5.2.3 Averaging the I Value

For a given input signal intensity, the I value of the DPLL is not constant. Instead it has the form of a $\cos(x)^2$ function as shown in Figure 5.2. If the AGC would directly use this signal, the pre-gain G would rapidly change its value, which would lead to an unstable or non-functional DPLL.

There are two possibilities to convert this periodical signal into a usable slowly varying signal for the AGC:

- Take the maximum from a given number of samples.
- Moving average over a given amount of time.

The first method would only work with a perfectly sinusoidal input signal. Unfortunately, in the real world, there will be noise on top of the input signal. Therefore, any transients or spikes that are bigger than the average amplitude will directly be visible to the AGC and cause the same problems as with the raw amplitude.

The second method acts as a low-pass filter and would remove any transients and spikes. This would result in a much more smooth signal for the AGC. On the downside, the averaged amplitude would only be about half as big as the unprocessed amplitude. However, this can easily be accounted for in the AGC algorithm. Therefore this method has been chosen in this thesis.

This finally leads to the AGC scheme presented in Figure 5.3.

The averaging is done with a Cascaded Integrator Comb (CIC) filter[34] of order 2 and a reduction rate of $1 : 2^{10}$. At a sampling rate of 80 MHz, this results in a new set of gains every 12.8 µs.

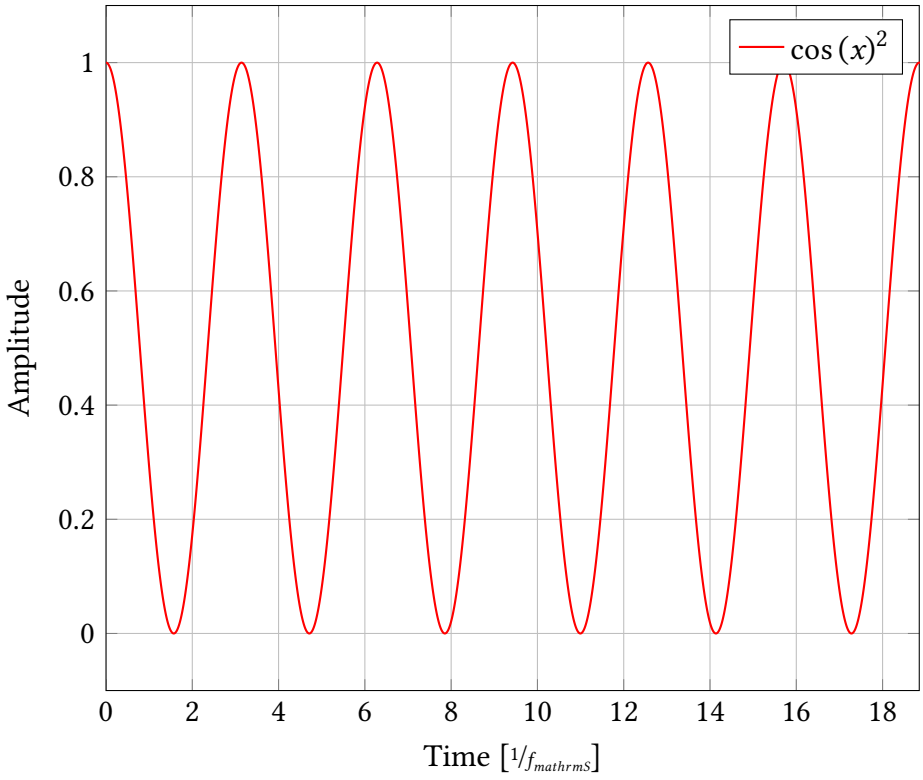


Figure 5.2: The DPLL *I* value is a $\cos(x)^2$.

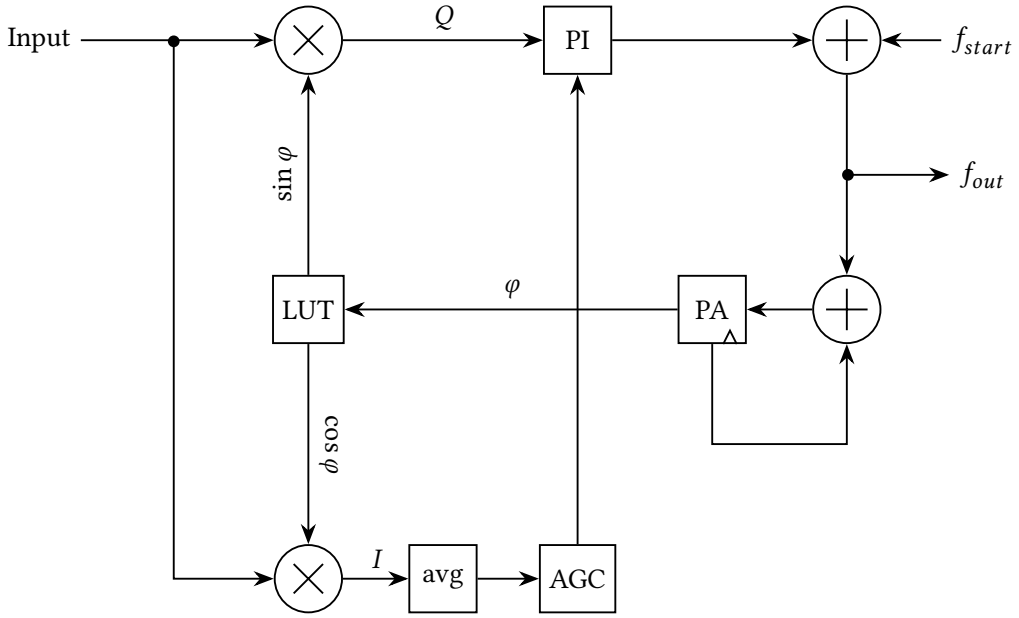


Figure 5.3: DPLL with AGC. The I value is averaged, processed by the AGC algorithm and the result is applied as an additional gain to the PI controller.

5.3 C++ Simulation

To prove that this approach to AGC is actually working, a simulation has been performed. The simulation has been written in C++ and can be found an Appendix A.2.1.

The simulation consists of an NCO, whose amplitude is varied over time from approximately 5 % to 100 %, a DPLL to track the the output of the NCO as well as the AGC block as explained above.

Figure 5.4 shows the amplitude of the amplitude modulated signal from the NCO as well as the amplitude measured by the DPLL that tracks the signal. Both values match quite well.

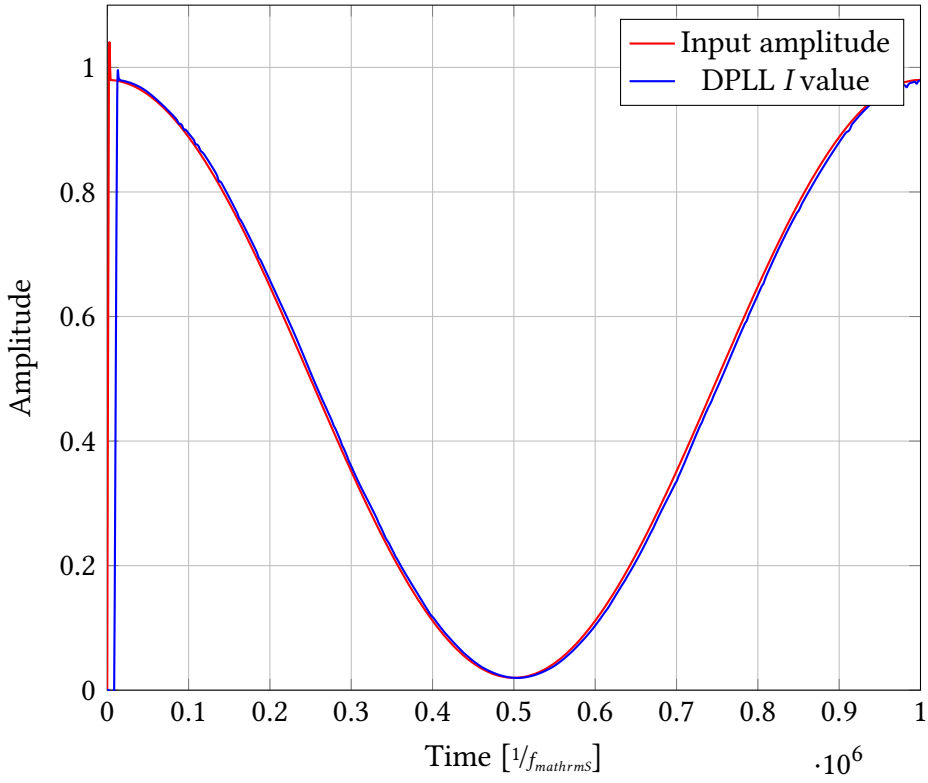


Figure 5.4: Input and output amplitude of a DPLL with AGC

Figure 5.5 shows the frequency of the NCO as well as the measured frequency of the DPLL. As it can be seen, both frequencies match each other very well. That means that the DPLL can track the signal from the NCO very well, even at very low amplitudes, thanks to the AGC.

As a comparison, Figure 5.6 shows the measured frequency of the DPLL with the AGC block disabled. As it can be seen, the DPLL fails at very low amplitudes.

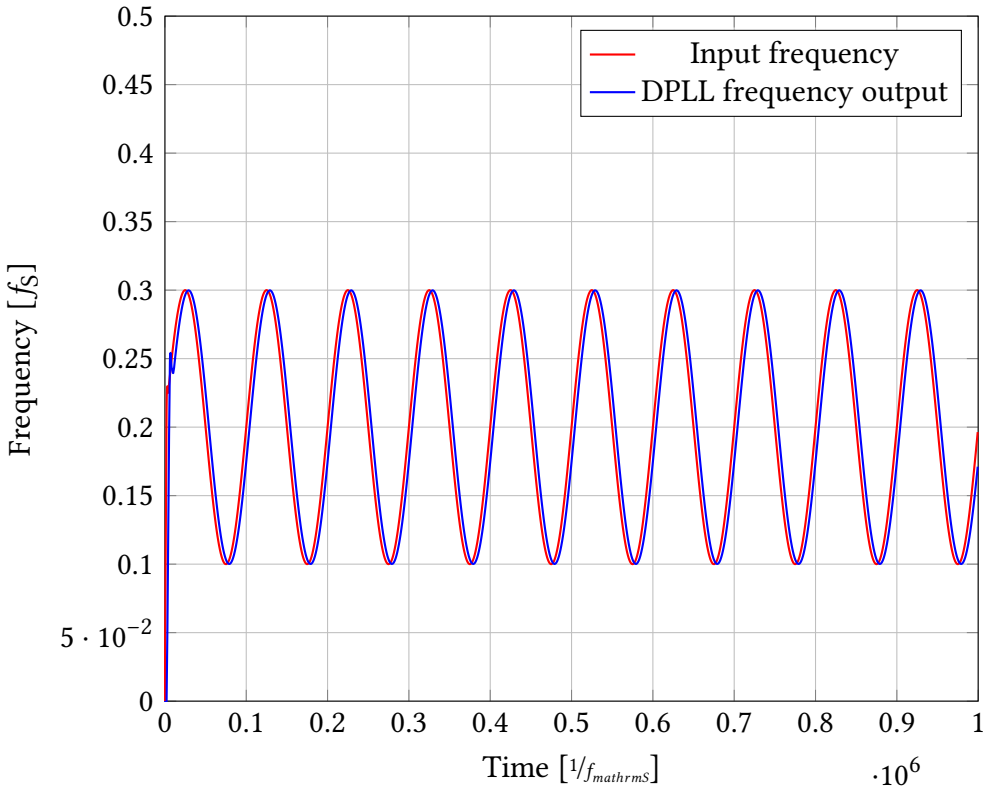


Figure 5.5: Input and output frequency of a DPLL with AGC

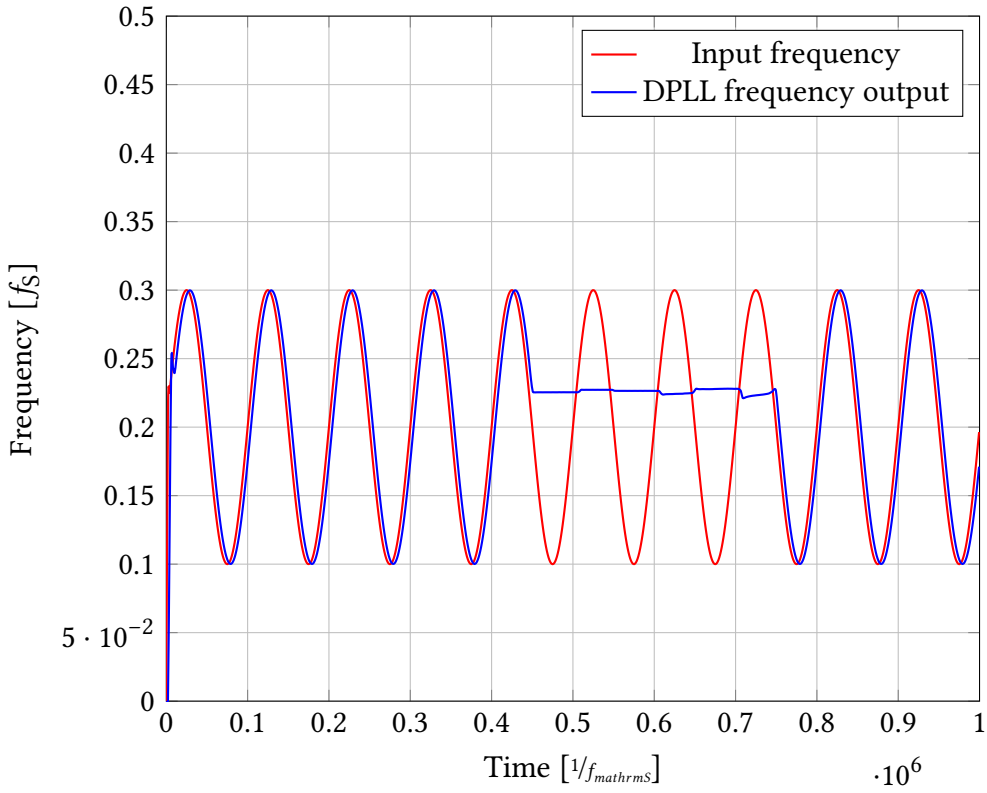


Figure 5.6: Input and output frequency of a DPLL without AGC

5.4 Implementation

The AGC algorithm has been implemented in VHDL for use in the LMS. The implementation can be found in Section B.2.1.

It takes the amplitude of the input signal, which is calculated as described in Section 5.2.3 and emits the processed gains as described in Section 5.2.2 as its output.

5.4.1 VHDL Simulation

Before using the VHDL implementation in the LMS it has been simulated to ensure its proper function and to make further small optimisations along the way. The corresponding VHDL testbench that is used to test the VHDL implementation can be found in Section B.2.2. The test conditions were identical to those in the C++ simulation.

Figure 5.7 shows the amplitude of the amplitude modulated signal from the NCO as well as the amplitude measured by the DPLL that tracks the signal. As in the C++ simulation, both values match quite well.

Figure 5.8 shows the frequency of the NCO as well as the measured frequency of the DPLL. As with the C++ simulation, both frequencies match each other very well. That means that the DPLL can track the signal from the NCO very well, even at very low amplitudes, thanks to the AGC.

As a comparison, Figure 5.9 shows the measured frequency of the DPLL with the AGC block disabled. As with the C++ simulation, the DPLL fails at very low amplitudes. Also, in contrast to the C++ simulation, the DPLL does not regain control over the lock and stays unlocked.

5.4.2 Performance Measurement

To test the AGC algorithm in an experiment, a signal generator is used to generate a 9 MHz sine signal, that can be tracked by the DPLL. This sine signal is slowly decreased in amplitude using a simple variable voltage divider. The result is tracked by an AGC enabled DPLL.

Figure 5.10 shows the frequency measured by the DPLL as well as the amplitude of the input signal as measured by an FFT. As it can be seen, the DPLL has no problems tracking the input signal down to very low amplitudes, thanks to the AGC.

For a comparison, Figure 5.11 shows the same setup but with the AGC disabled. As it can be seen, the DPLL fails to track the input signal at low amplitudes. That means that the AGC algorithm is working correctly.

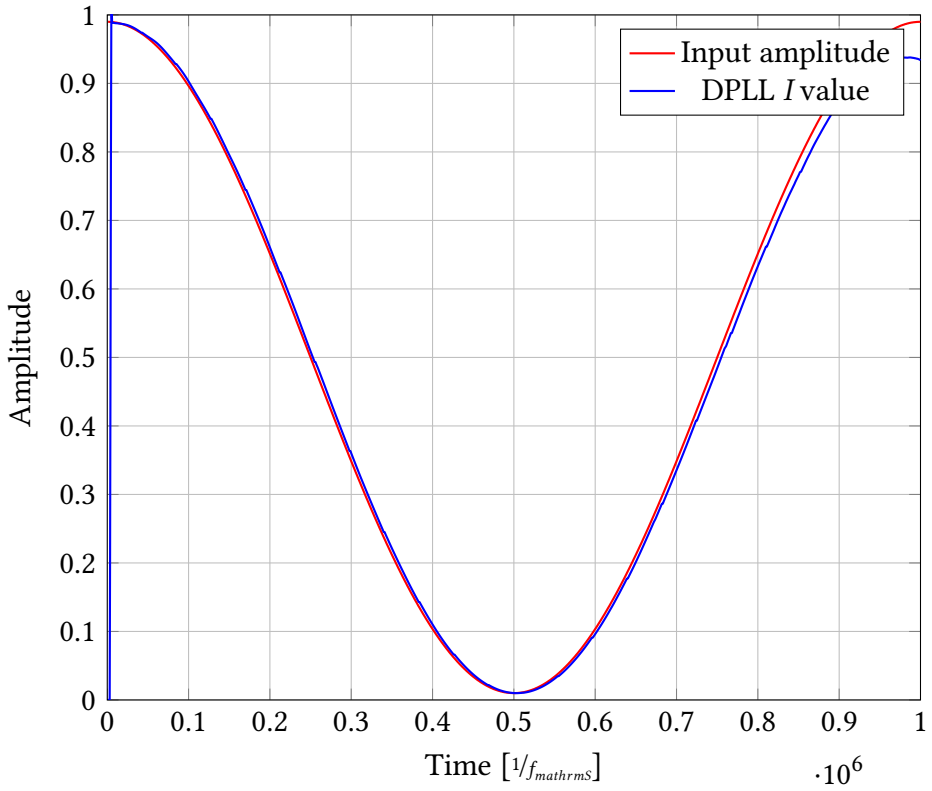


Figure 5.7: Input and output amplitude of a DPLL with AGC

The reason for the failing lock is the PI controller of the DPLL. When its gains are too low for the current amplitude, the error signal is not amplified enough, resulting in a too small actuator signal. In this case, the NCO is not able to follow the input frequency fast enough, and the lock fails. When the gains are too high for the current amplitude, the error signal is amplified too much, resulting in substantial overshoot in the NCO frequency. This also results in the lock failing.

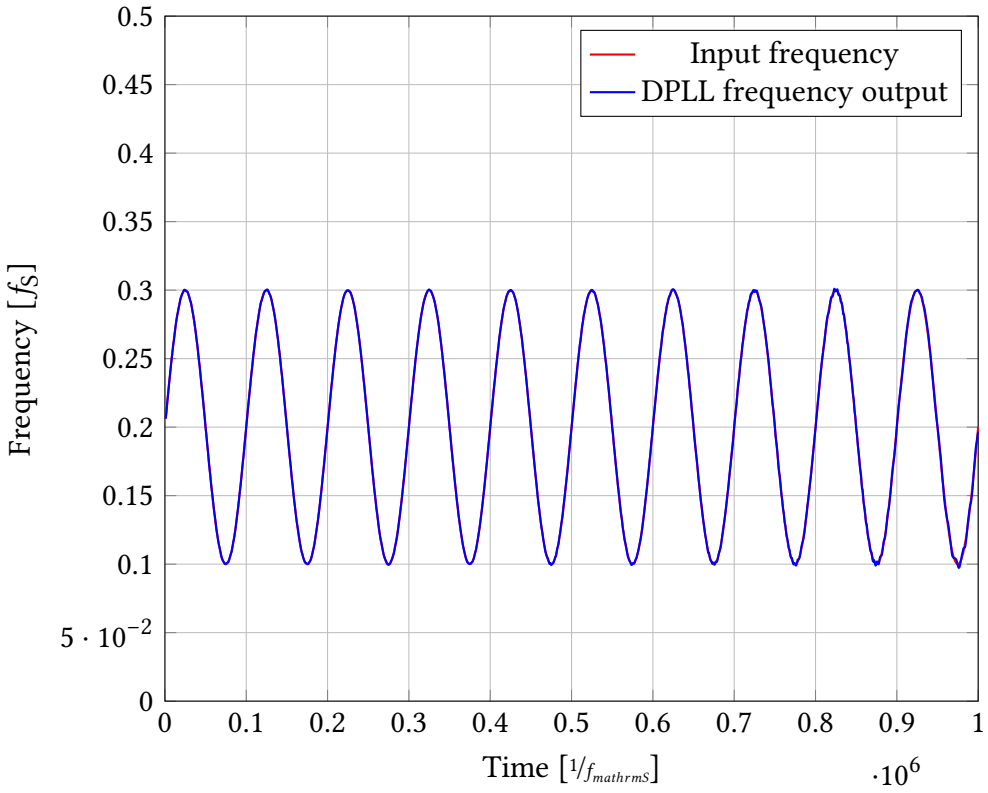


Figure 5.8: Input and output frequency of a DPLL with AGC

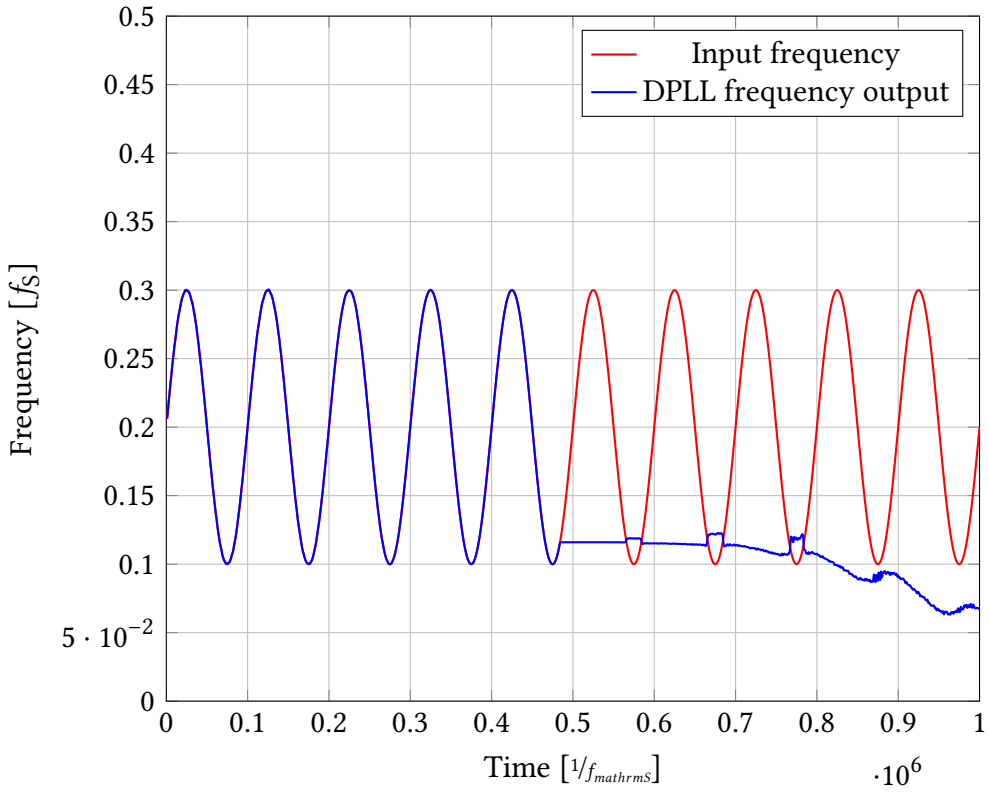


Figure 5.9: Input and output frequency of a DPLL without AGC

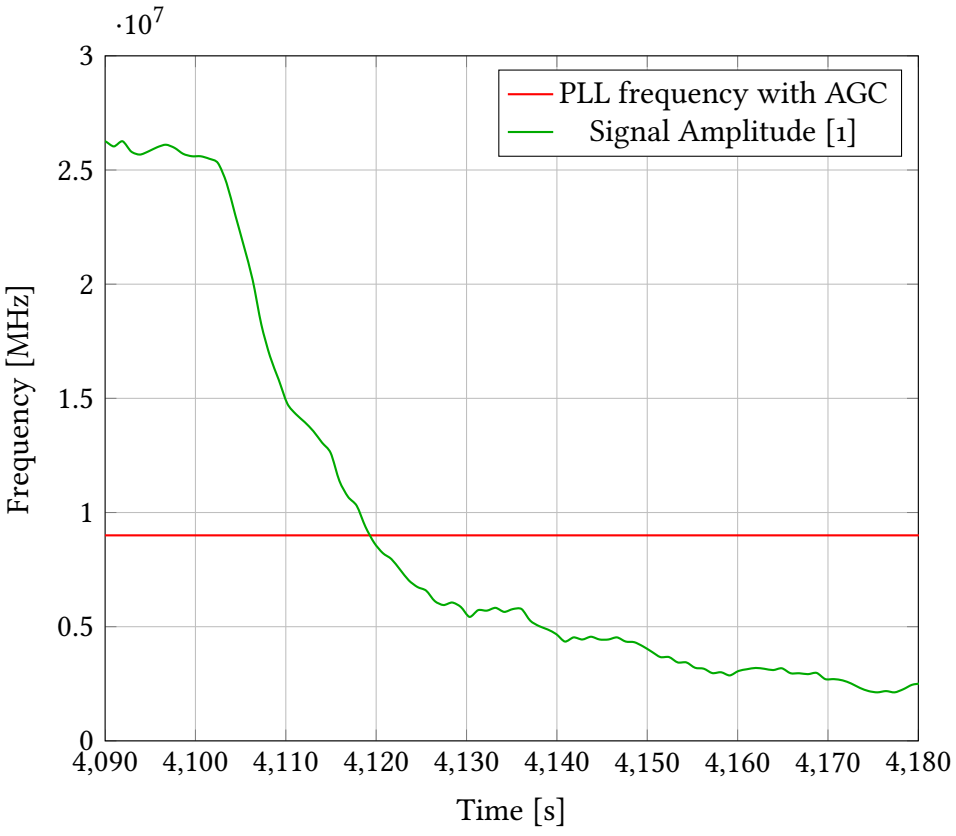


Figure 5.10: Frequency measured by a DPLL with AGC

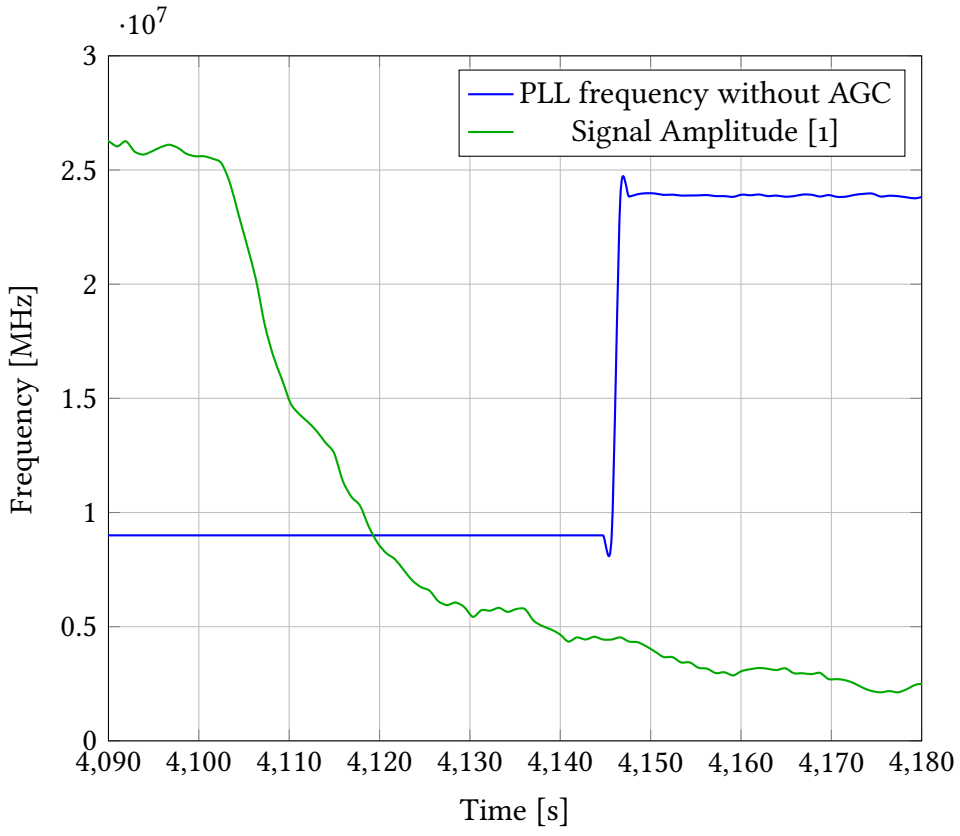


Figure 5.11: Frequency measured by a DPLL without AGC

Chapter 6

Differential Wavefront Sensing

For LRI and LISA in particular, heterodyne interferometry is the method of choice for phase measurements. In these applications, proper pointing is very important to achieve high measurement performance. Pointing is sensed using DWS, which works by interfering the local reference laser and the remote laser on a Quadrant Photo Diode (QPD)[35]. This results in different phases on each segment, which can be read out as phase differences $\Delta\varphi$. This is illustrated in Figure 6.1.

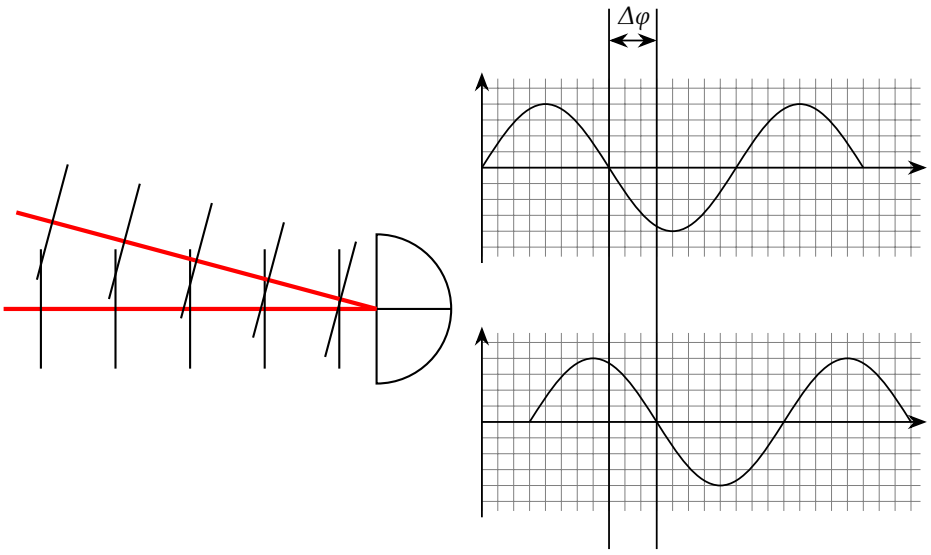


Figure 6.1: Interfering two laser beams on a QPD results in phase differences $\Delta\varphi$ between the segments.

DWS has numerous advantages in comparison with the also commonly used Differential Power Sensing (DPS). Instead of measuring the differences in phase between the quadrants of the QPD, in DPS the differences in power between

the quadrants of the photodiode are measured. This has the disadvantage of a lower optical gain as well as more susceptibility to amplitude noise. Also, with DWS, it is precisely measured what is needed to increase the heterodyne contrast. Therefore LISA is using DWS.

Phase differences can be read out with a phase meter using a DPLL in an FPGA. Up until now, this is implemented with four independent DPLLs. Each DPLL detects the phase of a single quadrant of a QPD. These four phases are then linearly combined to calculate the DWS signals:

$$\begin{aligned} \Delta x &= \varphi_A - \varphi_B + \varphi_C - \varphi_D \\ \Delta y &= \varphi_A + \varphi_B - \varphi_C - \varphi_D \,, \end{aligned} \tag{6.1}$$

where Δx is the phase difference in the x direction and Δy is the phase difference in the y direction. φ_A to φ_D are the relative phases on the respective segments of the QPD as denoted in Figure 6.2.

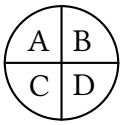


Figure 6.2: Arrangement and names of the segments of a QPD

Each of the signals has a power C and a noise density of N_0. This leads to a Carrier to Noise Density Ratio (CNR) for a single signal of:

$$S = \frac{C}{N_0} \,. \tag{6.2}$$

Since the frequency measured in those four DPLLs (hereafter called segment DPLLs) is approximately the same when used with a QPD, the overall frequency can be measured by feeding the sum of the four signals from the QPD into a separate DPLL (hereafter called common DPLL). Assuming the noise in the four channels are uncorrelated to each other and the signal, this will result in the noise getting added incoherently. Therefore the overall CNR equates to:[36]

$$\begin{aligned} O &= \frac{C + C + C + C}{\sqrt{N_0^2 + N_0^2 + N_0^2 + N_0^2}} \\ &= \frac{4C}{\sqrt{4N_0^2}} = \frac{4C}{2N_0} = 2\frac{C}{N_0} \end{aligned}$$

$$= 2S\,. \tag{6.3}$$

After adding the four signals from the QPD, the CNR increases by a factor of 2. This greatly improves the stability of the common DPLL, assuming that the DWS signals are small. Otherwise destructive interference in the summation of the segment signals can reduce the CNR.

Unfortunately, the segment DPLLs do not benefit at all from this higher CNR of the common DPLL even though the individual frequencies are very close to each other and not of great interest. Furthermore, these five DPLLs take up a significant amount of space inside the FPGA. Therefore the question arises whether these five DPLLs can be merged so that the DWS subsystem benefits from the improved CNR of the common DPLL.

The new method (initial idea by Prof. Dr. Gerhard Heinzel) described in this chapter aims to improve this situation significantly by combining these five DPLLs into a single so-called DWS DPLL. This features high signal to noise ratio measurements resulting in a more stable operation as well as lower space requirements in the FPGA while still being able to measure the overall frequency as well as the DWS signals. This new approach also will allow Equation 6.3 to hold for larger DWS signals. Due to its construction, no destructive interference can happen in the DWS DPLL.

6.1 New Approach

Instead of tracking the phase of each individual quadrant of the QPD like in a traditional DWS setup as described above, the DWS phase differences Δx and Δy as defined in Equation 6.1 as well as the average phase φ_{avg} of the whole QPD are tracked directly. The average phase is defined as:

$$\varphi_{\text{avg}} = \frac{1}{4}\left(\varphi_{\text{A}} + \varphi_{\text{B}} + \varphi_{\text{C}} + \varphi_{\text{D}}\right)\,. \tag{6.4}$$

Since the signals from a QPD have four degrees of freedom in phase, which would all be tracked by a traditional DWS system, a fourth phase value has to be tracked here as well to have the same amount of degrees of freedom. This fourth phase value is called the ellipticity ε of the QPD signals and is defined as:

$$\varepsilon = \varphi_{\text{A}} - \varphi_{\text{B}} - \varphi_{\text{C}} + \varphi_{\text{D}}\,. \tag{6.5}$$

The ellipticity usually is not measured as it is roughly constant and of little interest, since it cannot be controlled. However, its value is necessary for the function of the DWS DPLL design.

To understand how the DWS DPLL design works, a standard DPLL as used in [12] will be extended step by step in the next section, until the DWS DPLL has been constructed.

6.2 Design

The DPLL presented in Section 2.8.1 will be extended in the following subsections until the alternative DWS DPLL design has been constructed.

6.2.1 Phase Detector

To extract the phase error information from all four channels of a quadrant photodiode, four separate cosines, as well as four separate phase detectors in the form of multipliers, are needed instead of just one of each. This can be seen in Figure 6.3.

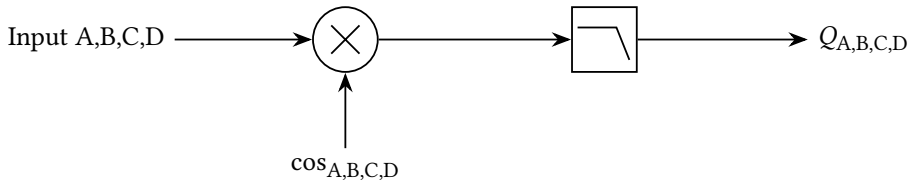


Figure 6.3: The phase detector of the DWS DPLL design consists of a multiplier and a low pass filter.

These four phase error signals are then summed up to obtain the average phase error, which is processed as before. Analogue to the calculation of the average phase error, the DWS error signals E_x, E_y as well as the additional phase error signal E are also calculated. This can be seen in Figure 6.4.

6.2.2 Phase Calculation

In a traditional DWS design, the phases of each QPD segment are tracked separately and therefore have their corresponding DPLLs and PI controllers. In the new DWS design, each DWS phase is tracked independently of the others. Therefore each phase error signal is connected to a PI controller.

Additionally, the PI controller for the average phase also accepts the starting frequency as the starting value for its integrator. It does not have to be added separately, anymore. Therefore the output of this PI controller is the actuator frequency, which is then fed into the phase accumulator to obtain the actuator

phase. All other PI controllers have starting values of zero and therefore directly output their respective actuator phases. This can be seen in Figure 6.5.

These actuator phases then have to be recombined to be fed to the sine/cosine look-up table of each QPD channel. To calculate the phases for each QPD channel, Equation 6.1, Equation 6.4 and Equation 6.5 can be inverted. This leads to:

$$\begin{aligned} \varphi_{\mathrm{A}} &= \varphi_{\mathrm{avg}} + \Delta x + \Delta y + \varepsilon \\ \varphi_{\mathrm{B}} &= \varphi_{\mathrm{avg}} - \Delta x + \Delta y - \varepsilon \\ \varphi_{\mathrm{C}} &= \varphi_{\mathrm{avg}} + \Delta x - \Delta y - \varepsilon \\ \varphi_{\mathrm{D}} &= \varphi_{\mathrm{avg}} - \Delta x - \Delta y + \varepsilon\,. \end{aligned} \tag{6.6}$$

A schematic representation of the implementation can be seen in Figure 6.6.

6.2.3 Complete Picture

Putting all the components that have been developed in the course of the last section together leads to a DWS DPLL design. A schematic overview of that design can be seen in Figure 6.7.

In the next sections, the DWS DPLL will be implemented in C++ and VHDL, several simulations will be performed to validate the design, and some performance measurements will be performed to confirm its performance.

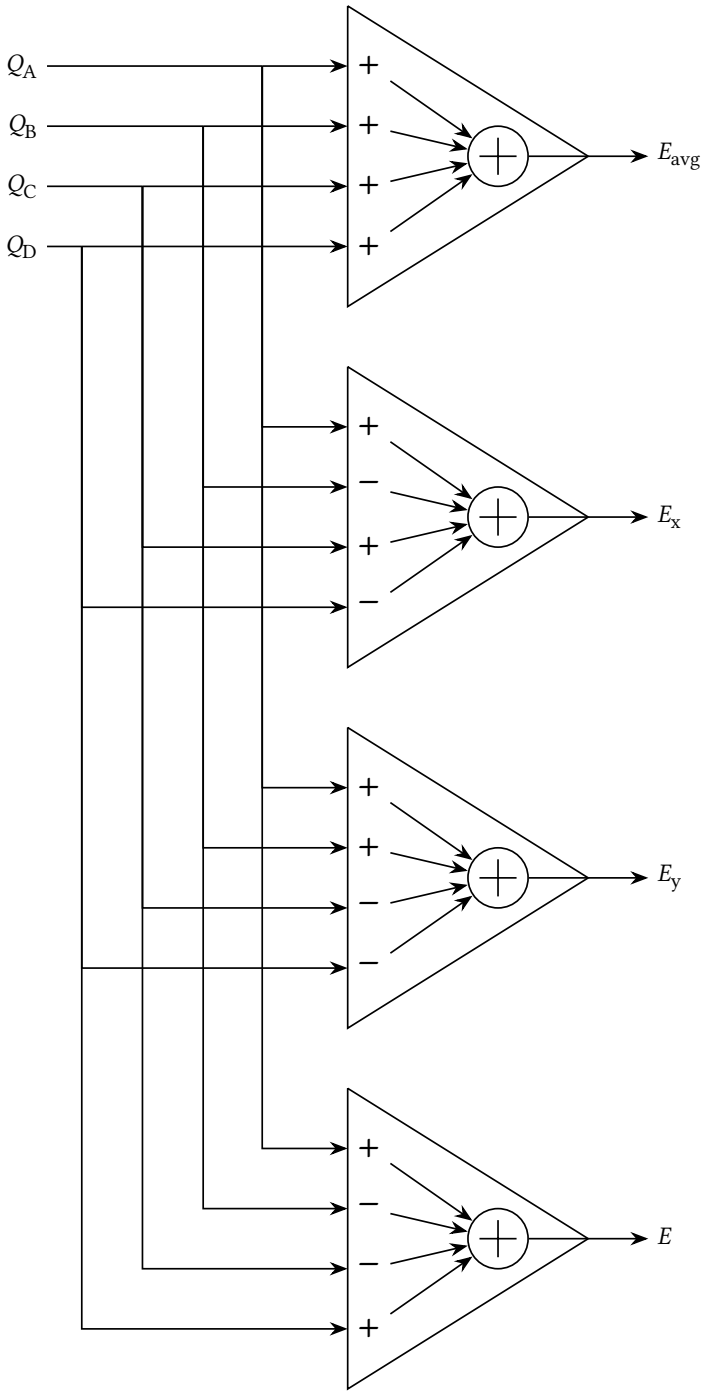


Figure 6.4: Phase Error Calculation in of the DWS DPLL design.

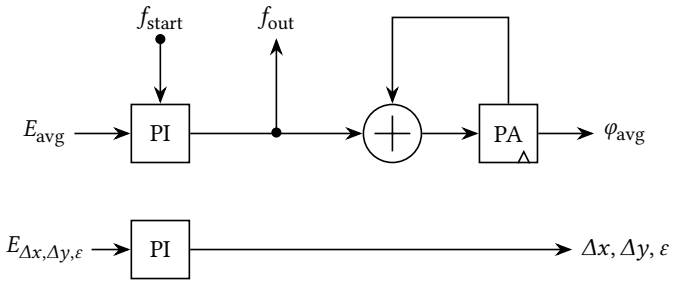


Figure 6.5: Calculation of the actuator phases of the DWS DPLL design.

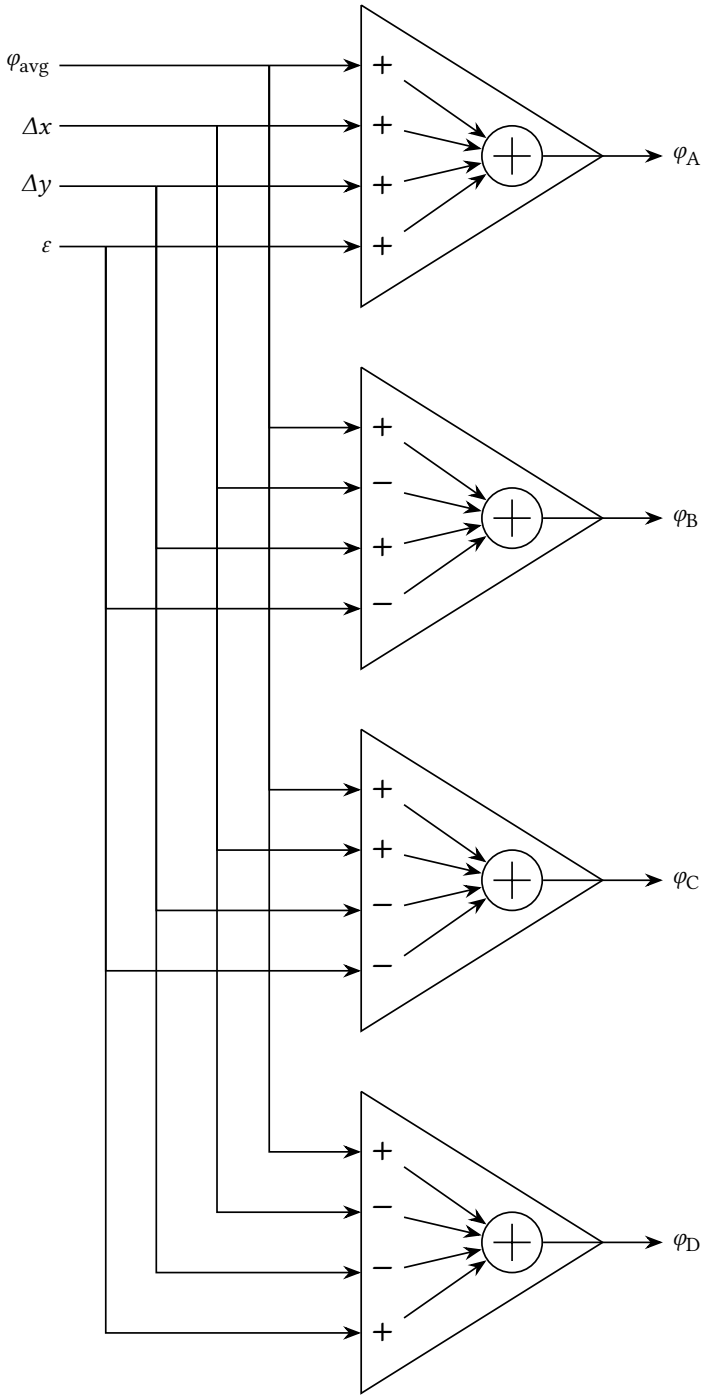


Figure 6.6: Calculation of the phases of the individual QPD quadrants in of the DWS DPLL design.

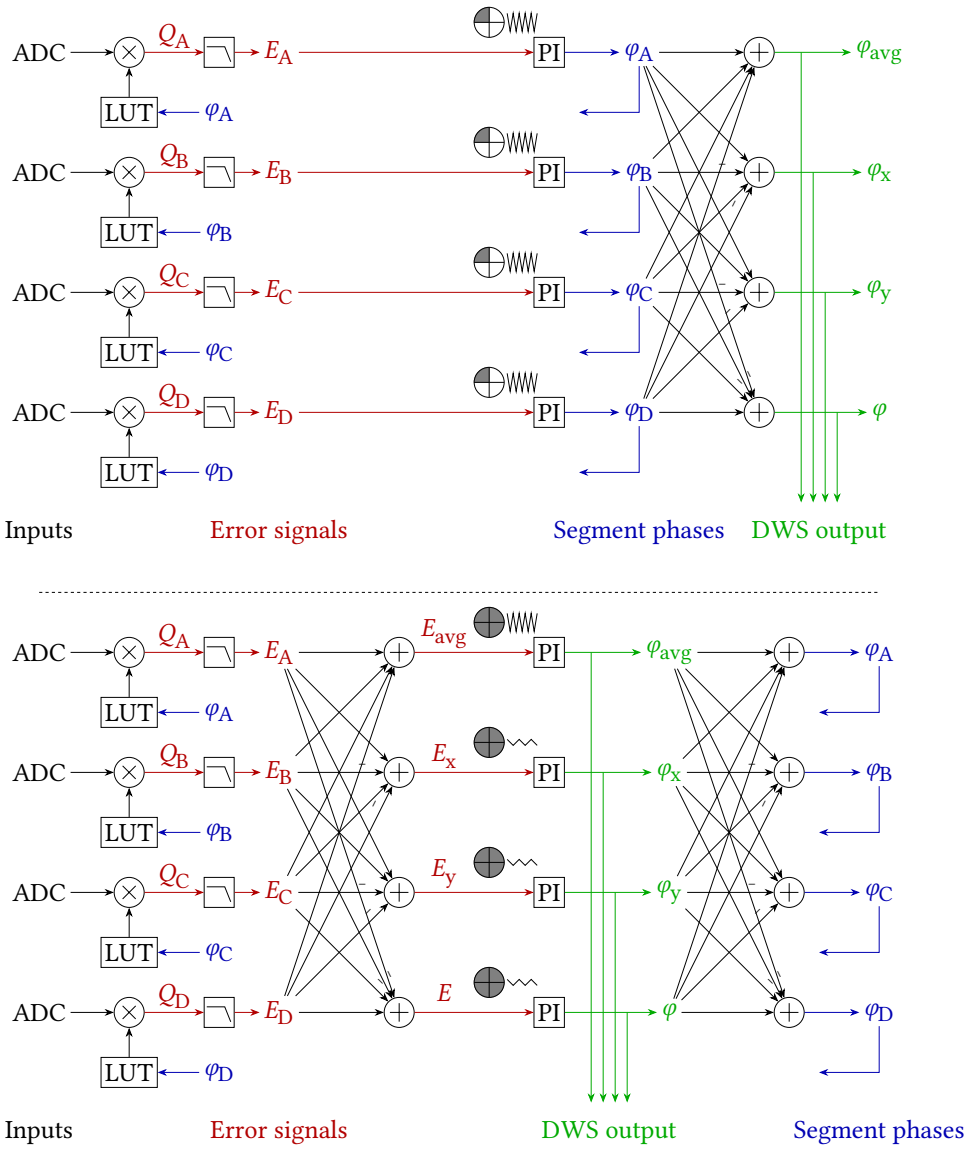


Figure 6.7: Stitching all the pieces together yields this complete picture of the DWS DPLL (bottom). A traditional DWS design is also shown for comparison (top).

6.3 C++ Simulation

To prove that this new approach to DWS is working, a low-level simulation has been performed. The simulation has been written in C++ and can be found an Appendix A.4.1.

The simulation implements the DWS DPLL as well as four additional NCOs, each simulating one channel of a QPD.

Over the run time of 10^6 time steps, the average frequency of the four NCOs is varied between $0.1\,f_S$ and $0.3\,f_S$ in a sinusoidal manner with a frequency of 10^{-5} cycles per time step. Also the DWS phase is varied between 0.12π and 0.32π in a sinusoidal manner with a variation frequency of 10^{-6} cycles per time step.

Using the following values for the gains of the PI controllers, the DPLL could successfully track each DWS phase as well as the overall frequency:

Controlled phase	P gain	I gain
φ_{avg}	−10	−12
Δx	−12	−14
Δy	−12	−14
ε	−12	−14

Table 6.1: Gains for the individual PI controllers of the DWS PLL.

The result of the simulation can be seen in the following two figures. Figure 6.8 shows a comparison of the overall input frequency to the measured frequency of the simulation. The difference between both can be seen in Figure 6.9, which is in the order ofo.1 % of the sampling frequency. Figure 6.10 shows a comparison of the simulated DWS phase to the measured phase of the simulation.

As it can be seen, the DPLL can successfully track the overall frequency of the input signal, and the DWS phase can successfully be followed.

As mentioned before, there is no such thing as a “starting phase” in the DWS DPLL as there is a “starting frequency” in a DPLL. Therefore the phase tracking always starts at zero. Due to the lower gains in the PI controllers for the DWS phases, the measured phase difference lags behind the input phase difference by a smidgen. This is not a problem since the DWS signals are expected to change slowly compared to the frequency. Even lower gains are therefore possible.

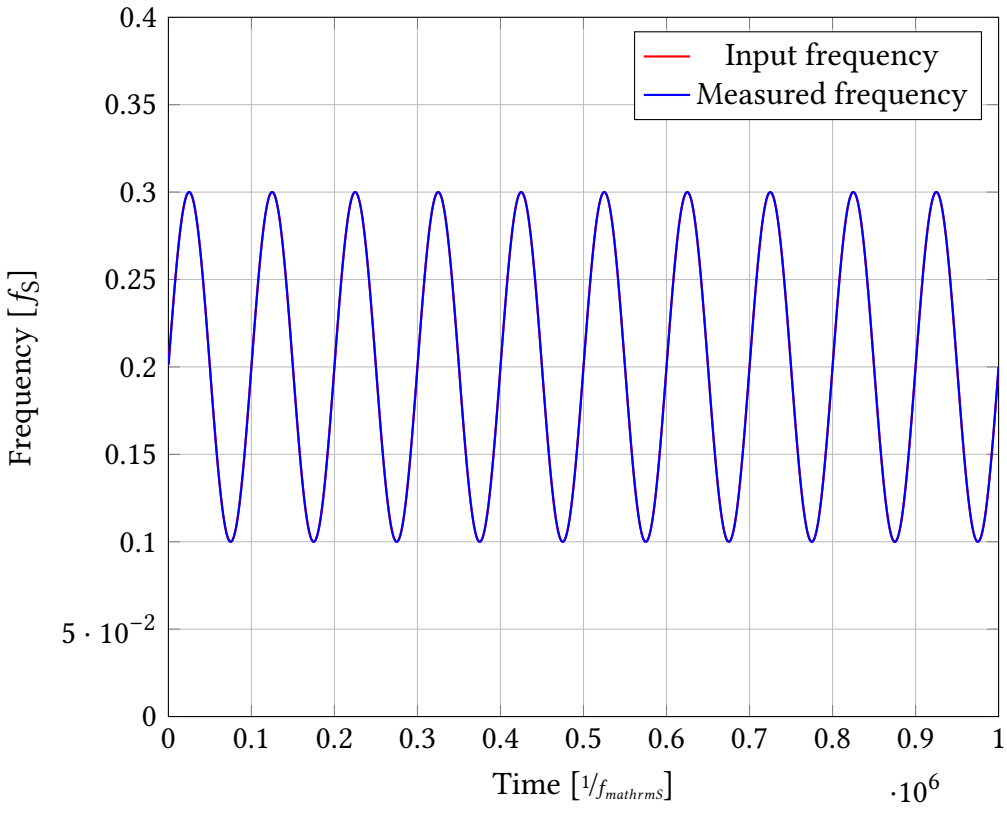


Figure 6.8: Simulation of the DWS DPLL, showing the ability to successfully track the average input frequency.

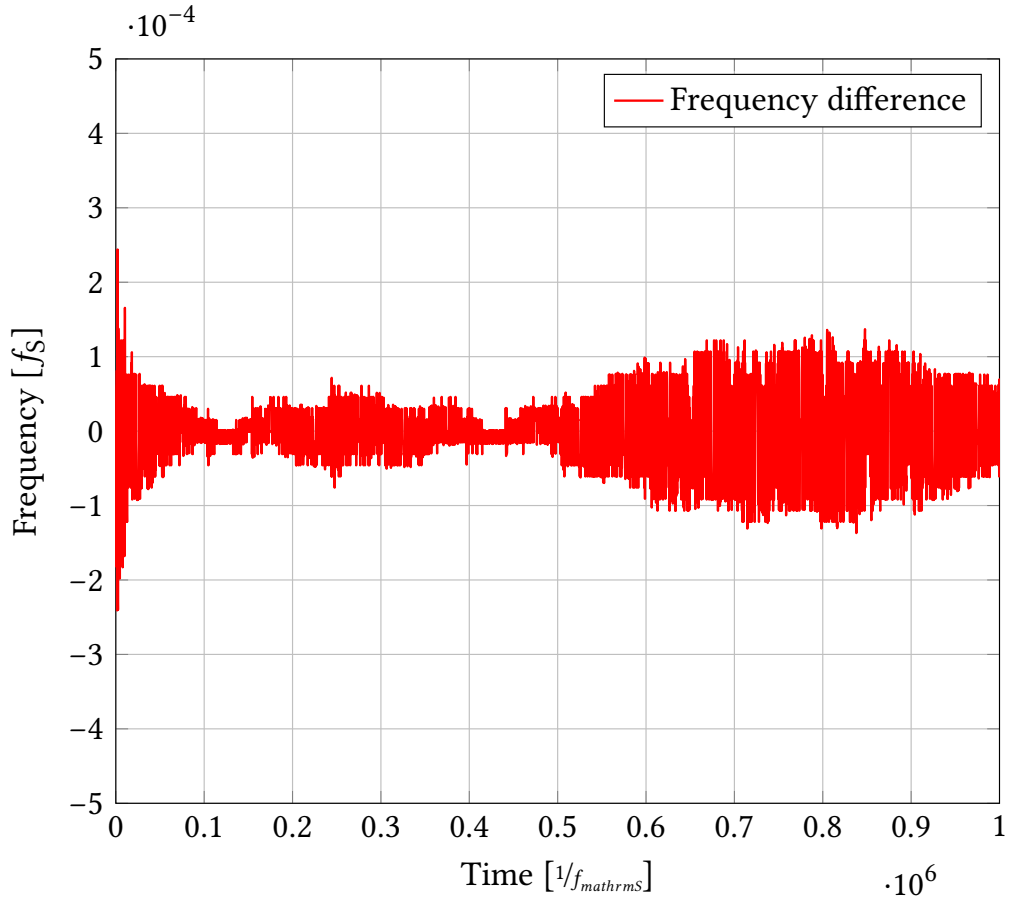


Figure 6.9: Simulation of the DWS DPLL, showing difference between its input and output frequency.

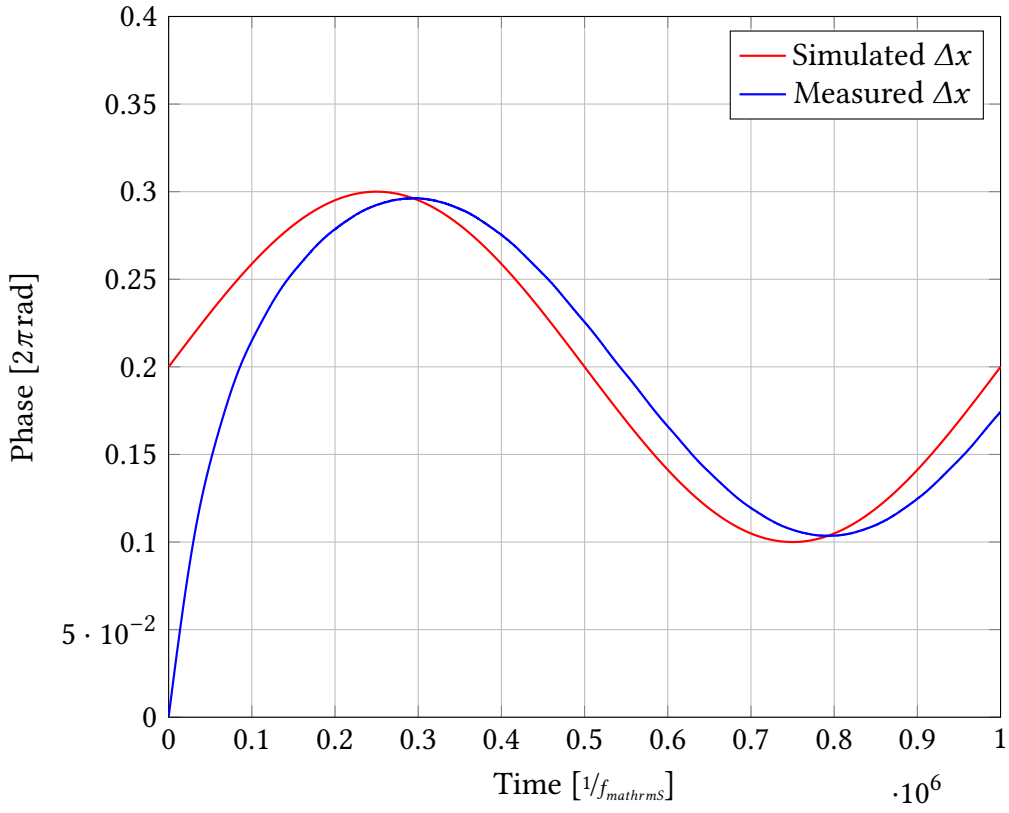


Figure 6.10: Simulation of the DWS DPLL, showing the ability to successfully track the phase difference in the horizontal direction.

6.4 Implementation

Since the underlying idea has been proven to work in a low-level simulation, the DWS DPLL needs to be implemented in actual VHDL code. The implementation can be seen in Appendix B.3.1.

6.4.1 VHDL Simulation

This implementation has also been simulated with a test bench similar to the low-level C++ simulation. The test bench can be found in Appendix B.3.2. The result of the simulation can be seen in the following two figures. Figure 6.11 shows a comparison of the simulated overall input frequency to the measured overall frequency of the simulation. The difference between both can be seen in Figure 6.12, which is in the order ofo.01 % of the sampling frequency. Figure 6.13 shows a comparison of the simulated DWS phase to the measured DWS phase of the simulation.

As it can be seen, that the DPLL can successfully track the overall frequency of the input signal and the DWS phase can successfully be followed.

As expected, the results of the VHDL simulation are identical to the results of the C++ simulation. Therefore the code can now be tested in a circuit in the following section.

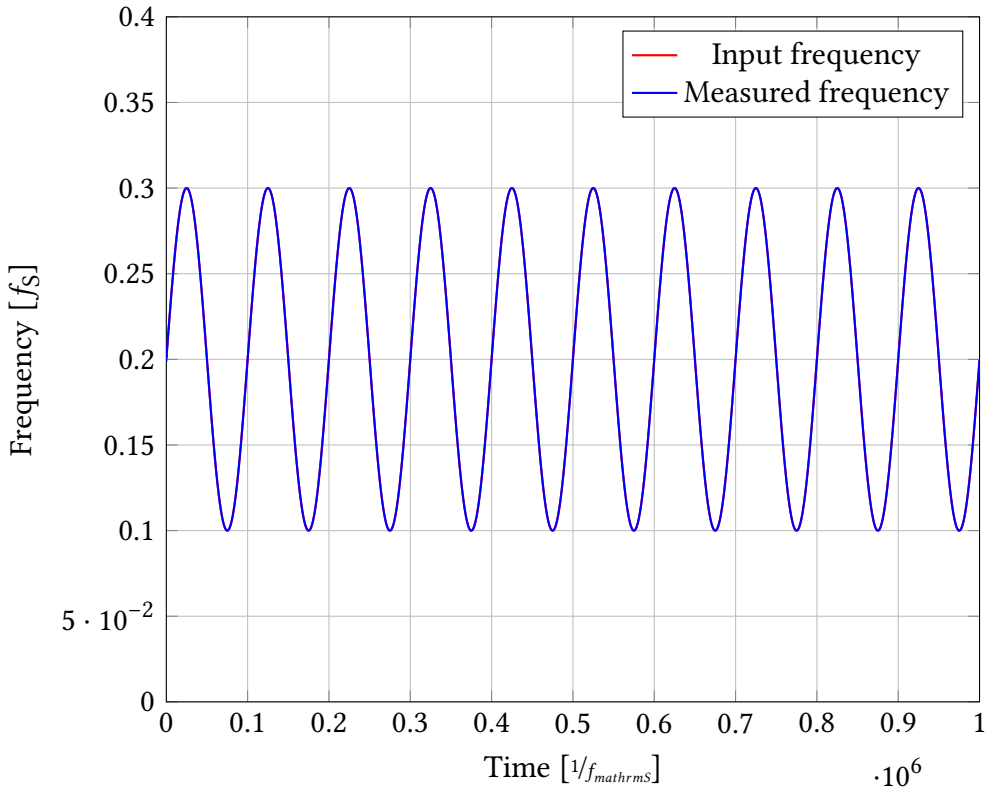


Figure 6.11: Simulation of the DWS DPLL, showing the ability to successfully track the average input frequency.

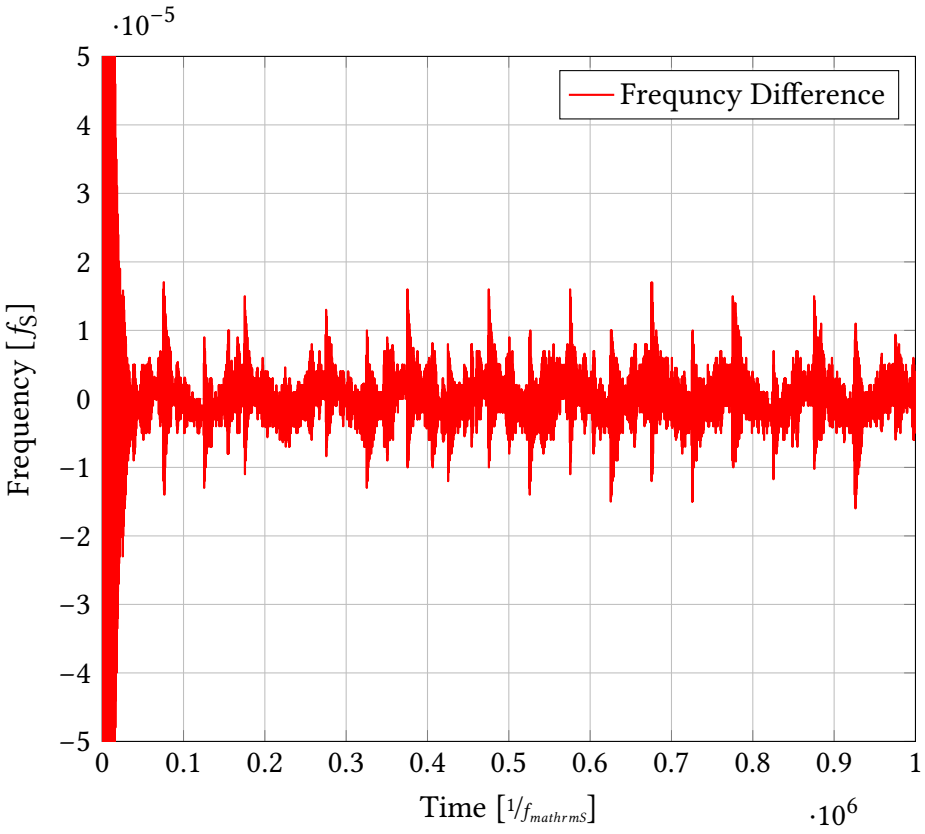


Figure 6.12: Simulation of the DWS DPLL, showing difference between its input and output frequency.

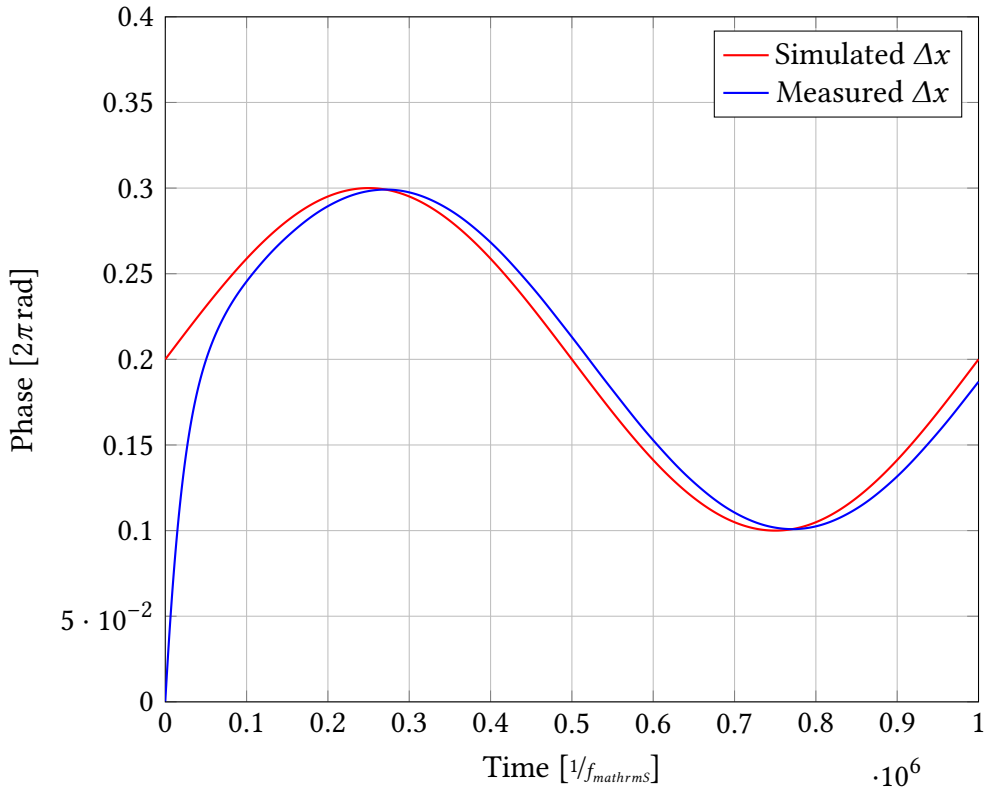


Figure 6.13: Simulation of the DWS DPLL, showing the ability to successfully track the phase difference in the horizontal direction.

6.5 Measurements

Multiple measurements have been performed, which will be described in the following subsections.

6.5.1 Functional Measurements

The first measurement has been performed with all four ADC inputs being tied to the same Single Element Photo Diode (SEPD) using a four-way signal splitter. Using the digital laser lock described in Section 4, two lasers have been locked to a difference frequency of 9 MHz, interfered with a beam splitter and measured with the SEPD mentioned above. The results of this measurement can be seen in Figure 6.14. Both DWS values show deviation of less than $5 \cdot 10^{-4} \cdot 2\pi$ from zero.

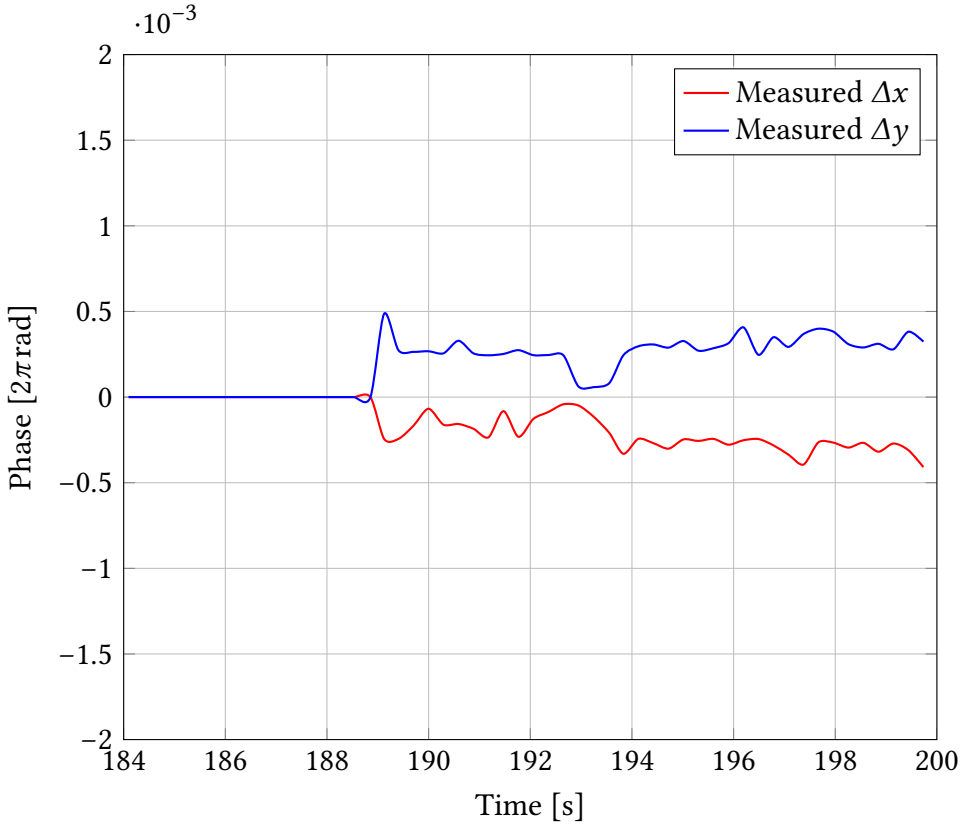


Figure 6.14: Measurement of the DWS DPLL with an SEPD.

It can be seen that the DWS DPLL can successfully track the overall frequency of the input signal. Since all four ADC inputs are measuring the very same signal,

there are no differential phases.

The second measurement has been performed with the four ADC inputs connected to a QPD, while both laser beams were not perfectly parallel. The beatnote frequency is left at 9 MHz. The results of this measurement can be seen in Figure 6.15.

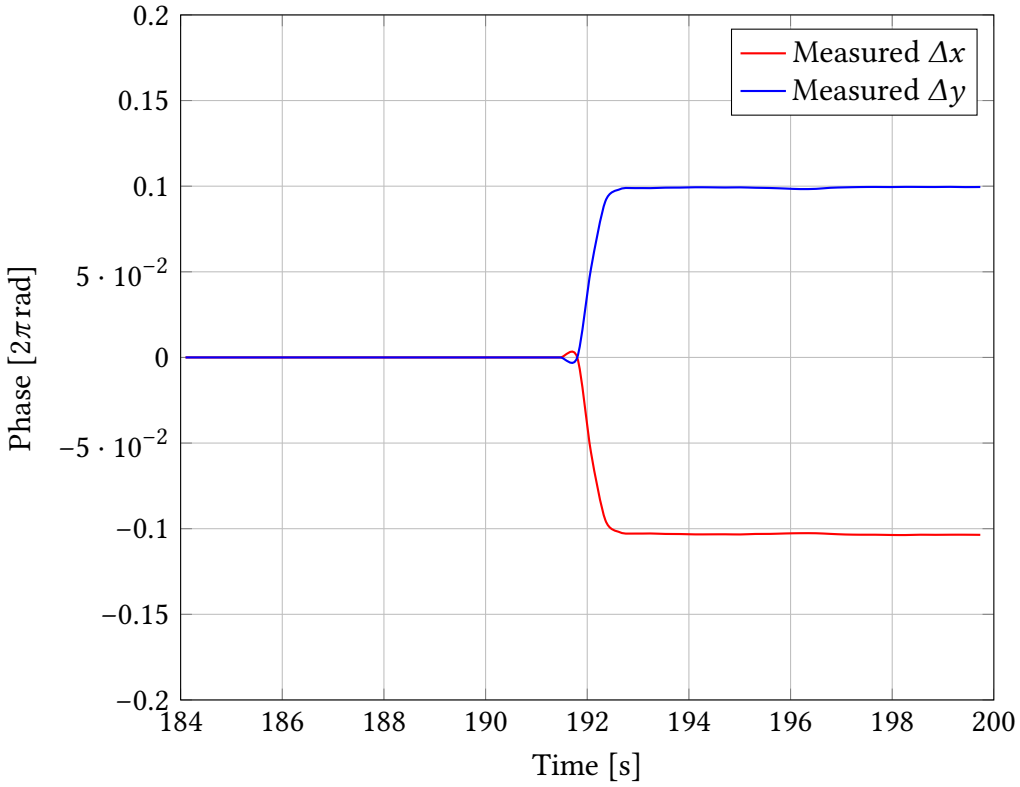


Figure 6.15: Measurement of the DWS DPLL with a QPD.

As it can be seen, the DWS angles have been measured successfully.

6.5.2 Performance Measurements

The increased CNR should improve the overall stability of the DWS DPLL in contrast to a single DPLL. This should result in being able to lock onto signals with smaller amplitude and a lower CNR in comparison with a single DPLL.

Varying Amplitude

To test the ability to lock onto signals with a smaller amplitude, the DWS DPLL as well as a single DPLL have been fed with signals of varying amplitudes ranging from 10 mV up to 1000 mV (peak to peak). The single DPLL could acquire a proper lock down to 85 mV, which corresponds to a digital signal with a width of about 3 bit. The DWS DPLL, on the other hand, could acquire a proper lock down to 45 mV, which corresponds to a digital signal with a width of about 2 bit.

This is an increase of a factor of approximately two as expected.

Varying Noise

To test the ability to lock onto signals with a lower CNR, the DWS DPLL as well as a single DPLL have been fed with signals of varying CNRs ranging from 86.5 dB Hz down to 43.9 dB Hz. This has been accomplished by adding noise onto a signal with a constant amplitude using a simple op-amp based circuit. The single DPLL could acquire a proper lock down to 53.7 dB Hz, while the DWS DPLL, on the other hand, could acquire a proper lock down to 45.8 dB Hz.

This is an increase of approximately 8 dB, which is even more than expected.

Chapter 7

Ranging and Data Transfer

Ranging allows the LMS to measure absolute distances between the LISA spacecrafts. Theses absolute distances are needed in post-processing for Time-Delay Interferometry (TDI) to remove laser frequency noise[37]. Also, there is a need for data transfer between the three LISA spacecrafts, because only one of them has a connection to the earth at a given time. The ranging and data transfer functionality of the LMS is accomplished through the already existing laser links between the satellites and is implemented through a DLL. Initial development has been done by Juan José Esteban Delgado[38] on different hardware using "The MathWorks Simulink". The VHDL implementation, further development, optimisations and extensions will be shown in the next sections. In the course of its development, it has been subsequently extended to increase its reliability and performance to fulfil the strict requirements of LISA.

7.1 Operational Principle

To make ranging possible, a Pseudo Random Noise (PRN) code is phase modulated onto the laser beam on the transmitting side. This phase modulation generates multiple sidebands, whose collective power does not exceed more than 1% of the carrier power. The PRN code has been hand-crafted using numerical optimisation techniques with an even length of 1024 so-called chips[39]. Each chip can have a value of either +1 or −1 and is 32 clock cycles in length, which means the chip rate is 2.5 MHz at a clock frequency of f_S = 80 MHz. This leads to signals of at least 1.25 MHz in the phase modulation as well as its harmonics.

The DPLL on the receiving side does not track those megahertz signals. They are directly visible in the quadrature output of the IQ-demodulator of the DPLL, i.e. in its error signal. The DPLL, therefore, demodulates the PRN code from the error signal of the DPLL, where they are insignificantly suppressed.

On the receiving side, the remote PRN code will then be correlated with a locally generated one. The offset in time between the local PRN code and the remote PRN code that maximises the correlation thus equals to the travel time of the transmission. Using this technique, the time which the PRN code needs to travel from the transmitter to the receiver can be measured absolutely.

A block diagram of the whole set-up can be seen in Figure 7.1.

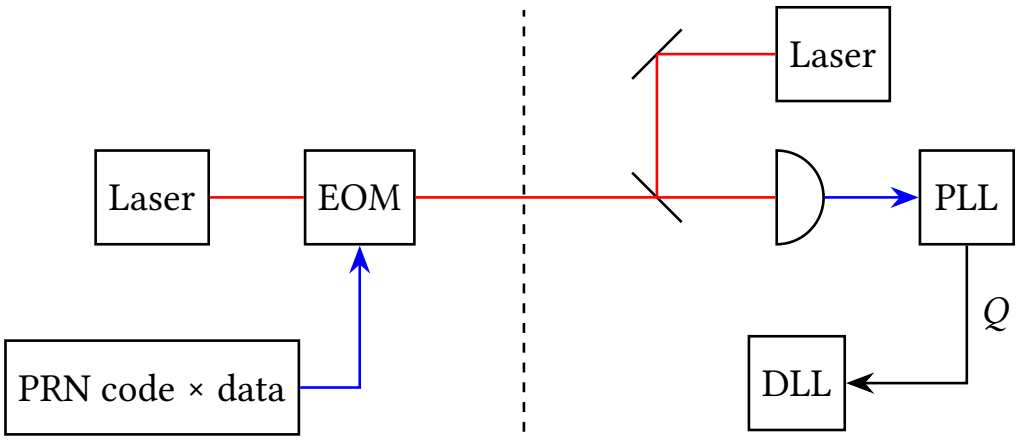


Figure 7.1: Schematic block diagram of the ranging subsystem of the LMS. On the transmitting side (left) a PRN code is modulated onto the laser beam. This laser beam is interfered with a second laser beam on the receiving side (right), generating a heterodyne signal. This signal is measured by a photodiode, and its frequency is tracked by a DPLL. The PRN code is demodulated by the DLL from the quadrature output of the DPLL. The Laser beams are marked as red, analogue signals are marked in blue and digital signals are marked in black.

Four spectra of a PRN code modulated signal with typical modulation indices can be seen in Figure 7.2. This figure has been generated with the C++ code in Appendix A.5.1. The different PRN codes that can be used can also be found there.

In addition to ranging, the DLL is also used to transfer data. With each PRN sequence, 32 data bits can be transmitted, where each data bit is 32 chips in length. The data to be transmitted is first transformed into values of +1 and −1, where a 0 corresponds to a −1 and a 1 corresponds to a +1. Then it is attached to the PRN code by using multiplication. The data modulated PRN code will then be phase modulated onto the laser beam on the transmitting side.

On the receiving side, a simple multiplication of the local PRN code and the remote PRN code reveals the transmitted data. At the end of this process, the data has to be transformed back to 0s and 1s before it can be further processed using the same mapping as on the transmitting side. The actual modulated data does not

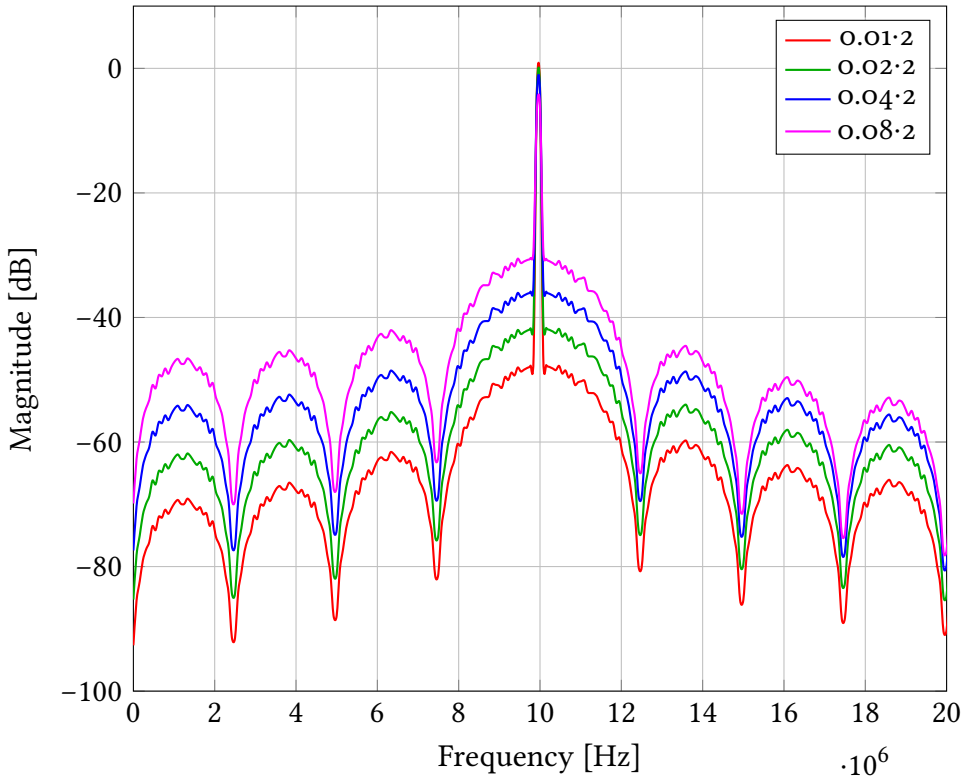


Figure 7.2: Linear spectrum of a PRN code modulated 10 MHz carrier with four different modulation indices

have any influence on the performance and stability of theDLL. However, without the presence of data, a higher performance DLL could be built. An explanation for this can be found in Section 7.2.1.

7.2 Structural Overview

A DLL consists of four basic parts:

- The local PRN code generation
- The PRN code correlator
- The loop controller
- The control FSM

and can operate in two different modes:

- Acquisition mode
- Tracking mode

The different modes will be described later.

The generation of the local PRN code starts with a counter which is continuously counting. To its value a static start offset determined during the acquisition phase as well the actuator offset calculated by the servo loop controller is added. The origin of these offsets will be described in more detail below. The result is then used as an address for a LUT that contains six different PRN codes, one for each of the six one-way links of LISA. The same code as used on the transmitting side is selected and fed into the correlators. The result of the correlators is then used in the servo loop controller as well as in the control FSM during the acquisition phase.

7.2.1 PRN Code Correlator

The PRN correlator is used to correlate the input signal with the local PRN code and to recover the embedded data. To accomplish that, the input signal is multiplied by the locally generated PRN code, and its result is then sent through a series of two Integrate-And-Dump (IAD) filters to calculate the correlation and recover the data.

In this implementation, the input of a correlator has a width of 16 bit. After the multiplication with the PRN code, the signal has a width of 40 bit, which stays constant for the rest of the correlator.

An IAD filter continuously integrates over its input signal. After a fixed period it "dumps" its integration value to its output and resets its integration value to zero. Then the process starts from the beginning.

The first IAD filter dumps every data period, which is every 12.8 µs. This results in a data rate of 78.125 kbit s^{-1}. To recover the transmitted data, the sign of the output of this first IAD filter is read and transformed back to binary data as described earlier.

After the first IAD filter, the absolute value of the output is calculated and sent to the second IAD filter. Due to the usage of the absolute value, the modulated data is not present anymore has no impact on the rest of the DLL.

The second IAD filter then dumps every PRN code period, which is approximately every 0.4 ms. Since the absolute filter eliminated the sign, the output of the second IAD filter is always positive and corresponds to the amount of correlation between the input signal and the locally generated PRN code.

Without the presence of data, the first IAD filter could be omitted, resulting in a single longer coherent IAD filter, and thus improving the performance of the DLL.

A schematic block diagram of the correlator can be seen in Figure 7.3.

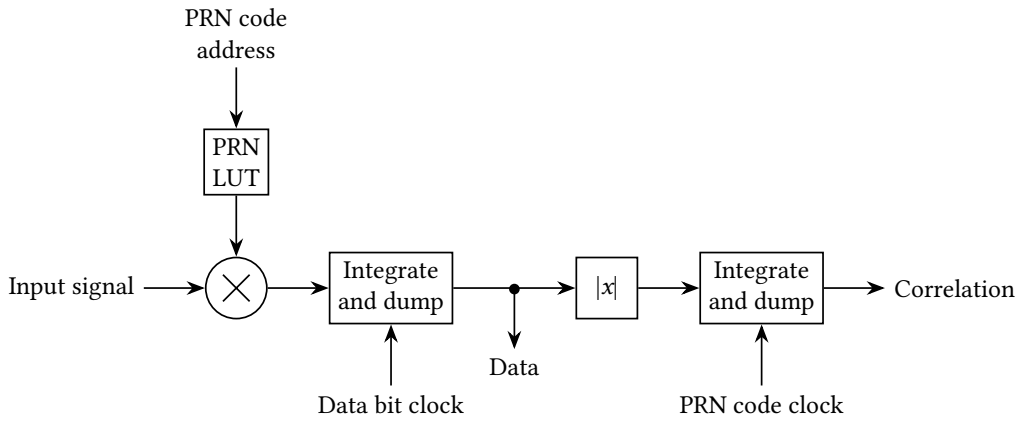


Figure 7.3: Schematic block diagram of a PRN correlator. The Input signal gets multiplied with the local PRN code and travels through a series of two IAD filters to extract the transmitted data and calculate the correlation.

There are three of these correlators inside the DLL. One is the punctual correlator, which functions as described above, and the other two are the early and late correlators. The difference between the early and late correlators and the punctual correlator is that the former ones use a local PRN positively or negatively delayed by half a chip, which corresponds to 200 ns. In case the punctual correlator has the maximal correlation, the early and the late correlator output the same amount of correlation. If the offset of the punctual correlator is slightly off, one of the early and late correlators has a slightly higher correlation than the other one. Therefore the difference of the correlation of the early and late correlator is a measure for the direction in which the offset of the punctual correlator has to be shifted to achieve maximum correlation. This can thus be used as an error signal for the loop controller. A schematic block diagram of the error signal generation can be seen in Figure 7.4.

7.2.2 Loop Controller

The loop controller consists of a simple PI controller, which takes the difference between the early and late correlator as its input error signal. This error signal is plotted as a function of the delay in Figure 7.5. As it can be seen, the loop controller only works for a limited amount of delay. Therefore the control FSM is used to set a rough delay as a starting point. This is described in greater detail in the next subsection. The output of the PI controller is used as actuator signal and

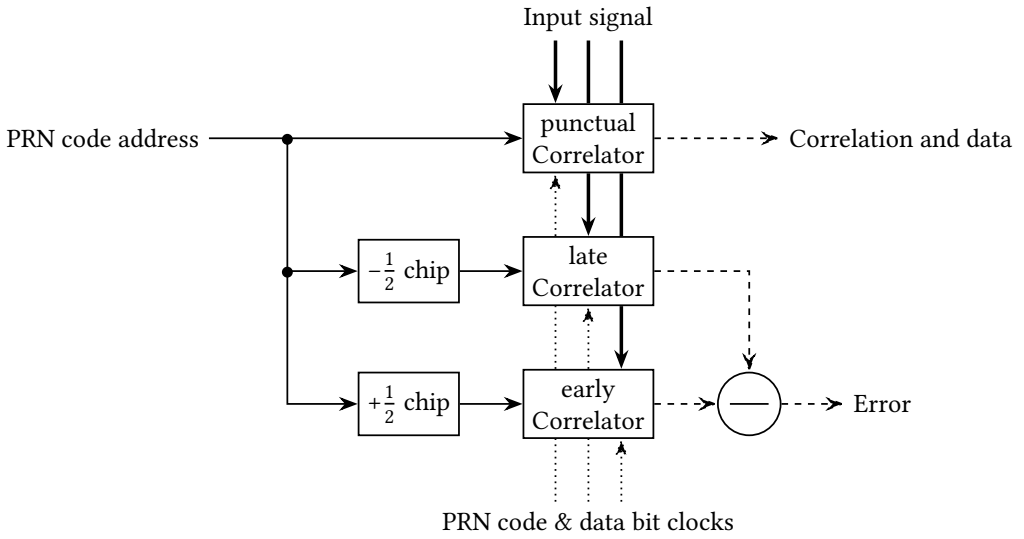


Figure 7.4: Three correlators are used in the DLL. One gives the correlation and the other two combined give the error signal for the loop controller. Delay signals are represented as solid lines, the input signal is represented as a thick line, clocks are represented by dotted lines and correlation signals are represented by dashed lines.

is added as an offset to the PRN code address counter as described earlier. In this implementation width of the input and output width of the PI controller is 40 bit.

7.2.3 Control Finite State Machine

The control FSM controls the transition between acquisition mode and tracking mode. When the DLL starts, the FSM is in acquisition mode. In this mode all possible PRN code offsets are scanned through until an offset with a correlation above 10 % is found. After that, the FSM switches to tracking mode where the offset mentioned above is not modified anymore. In tracking mode, the loop controller as well as the early and late correlators are switched on to form a closed loop. It is possible to leave the tracking mode and switch back to acquisition mode when the measured correlation falls below 10 %. However, this does only happen when the DPLL unlocks or the transmitted PRN code changes or vanishes.

As a side function, the control FSM also generates the timing signals for the IAD filters in the correlators. For that purpose, it is using the clock of the glsPRN code address counter mentioned above as a time base. To generate the dump signals for the first IAD filter, the clock is divided by 32. The resulting clock is divided by 32 a second time to generate the dump signals for the second IAD

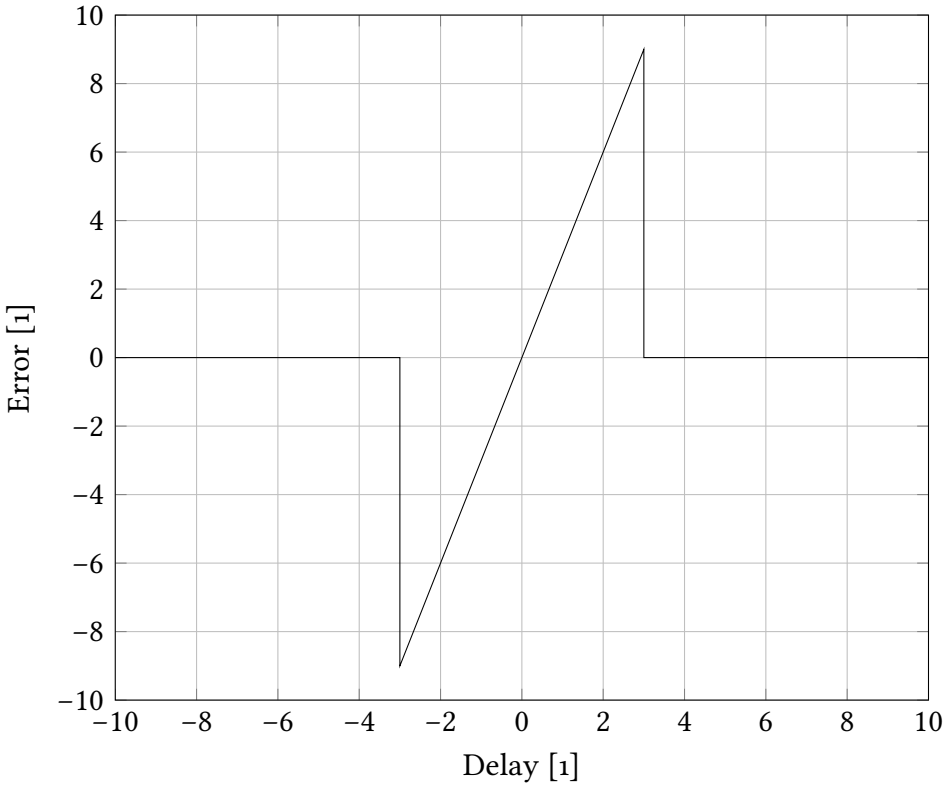


Figure 7.5: The difference between the early and late correlator as a function of the delay.

filter.

A complete block diagram of the DLL can be seen in Figure 7.6.

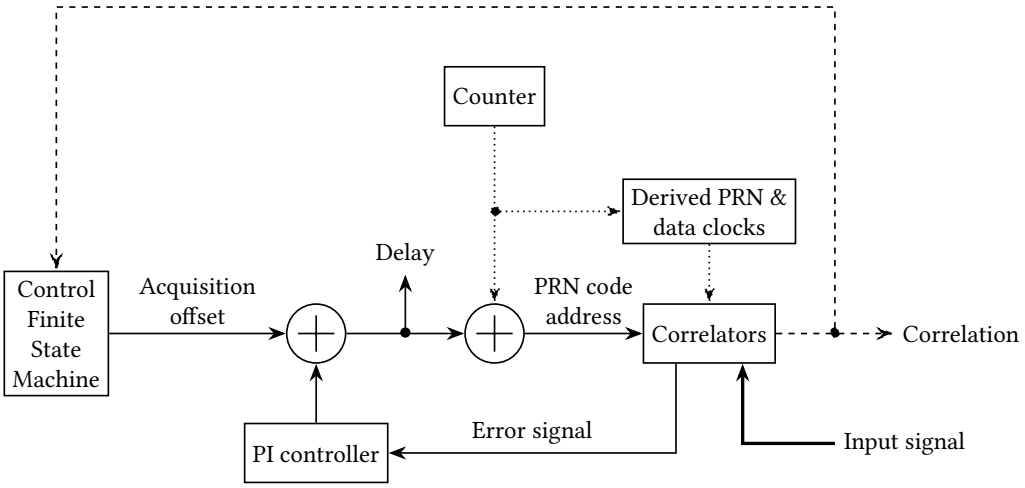


Figure 7.6: Schematic block diagram of a DLL. This includes the counters for the local PRN code address, the early, punctual and late correlators, the loop filter and the control FSM. Delay signals are represented as solid lines, the input signal is represented as a thick line, clocks are represented by dotted lines and correlation signals are represented by dashed lines.

7.3 Detailed Enhancements

To improve the performance and reliability of the DLL to a level that meets the requirements of LISA, numerous modifications and optimisations to the DLL had to be made, especially to the data recovery part.

7.3.1 Data Recovery Improvements

Data recovery in the DLL as presented up until now has been found to only work for small delays without a big dynamic range. The reason for this is that the timing signals used by the correlators are directly derived from the PRN code address counter, without taking any offset from the acquisition phase or the loop controller into account. If the measured delay now approaches 16 chips, which is half a data bit, each local data period contains half of two different remote data bits, which causes many errors. This is illustrated in Figure 7.7.

To prevent this from happening, the current implementation of the DLL has been modified. The timing signals are now being derived after the offsets from the acquisition phase, and the loop controller has been added to the PRN code address counter. This corresponds to the effective PRN code address, which is also used to drive the PRN code LUT. With these modifications, the timing signals and the

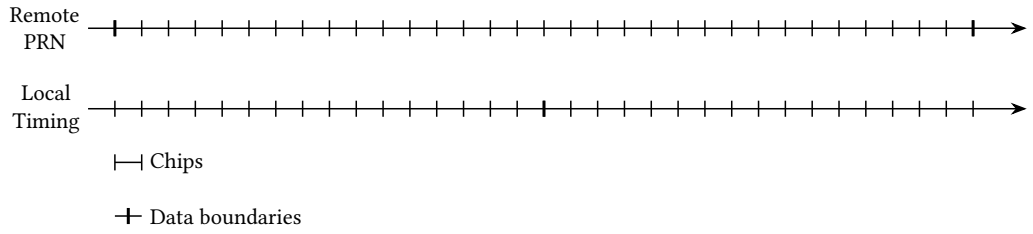


Figure 7.7: If the delay between the local and remote PRN code approaches 16 chips, the mismatch between remote data bits and local data timing signals can lead to a high Bit Error Rates (BERs)

data bits are in sync at all times. A version of Figure 7.6 with this implemented can be seen in Figure 7.8.

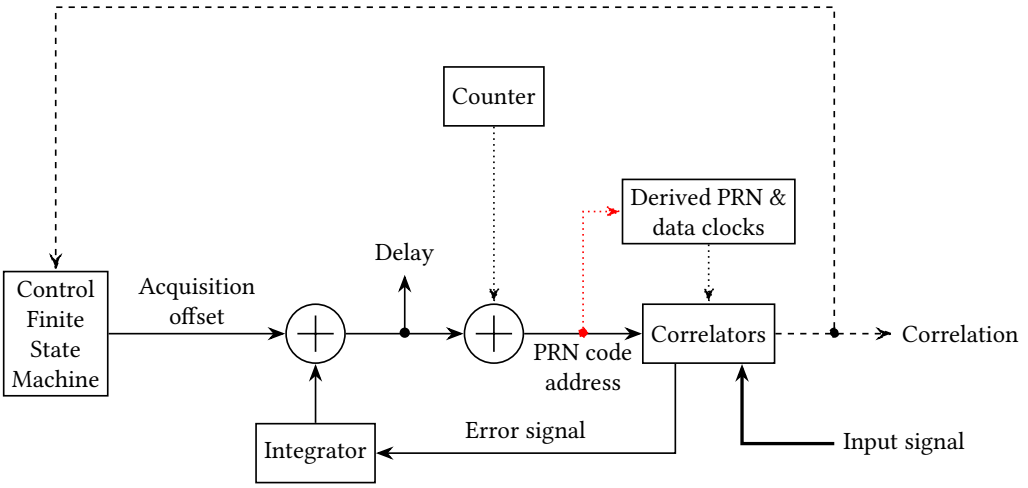


Figure 7.8: To fix the high BERs originating from the mismatch between the local data timing signals and the remote data bit boundaries, the offsets from the acquisition mode and the loop controller are taken into account when generating the data clock signal. The change from Figure 7.6 is marked in red.

7.3.2 Timing Glitches

Now that the offsets from the acquisition mode and the loop controller are taken into account, every time the delay is recalculated by the loop controller, this also affects the timing signals and leads to another problem that can be a source of bit errors. This problem has its roots in the particular way the timing signals are

derived from the PRN code address. The data timing signal is a clock and should have a rising edge every 32 chips. Therefore the 5^{th} bit of the effective PRN code address is used for this purpose. Every time the address passes a multiple of 32, there is a rising edge[1] in the data timing signal. If the delay calculated by the loop controller gets smaller, the chance that the PRN code address jumps from just over a multiple of 32 to just under a multiple of 32 gets higher. This causes an extra rising edge in the data timing signal and therefore an extra (erroneous) data bit. The higher the dynamic range of the delay of the remote PRN code is, the higher is the possibility for this to happen. This effect is illustrated in Figure 7.9

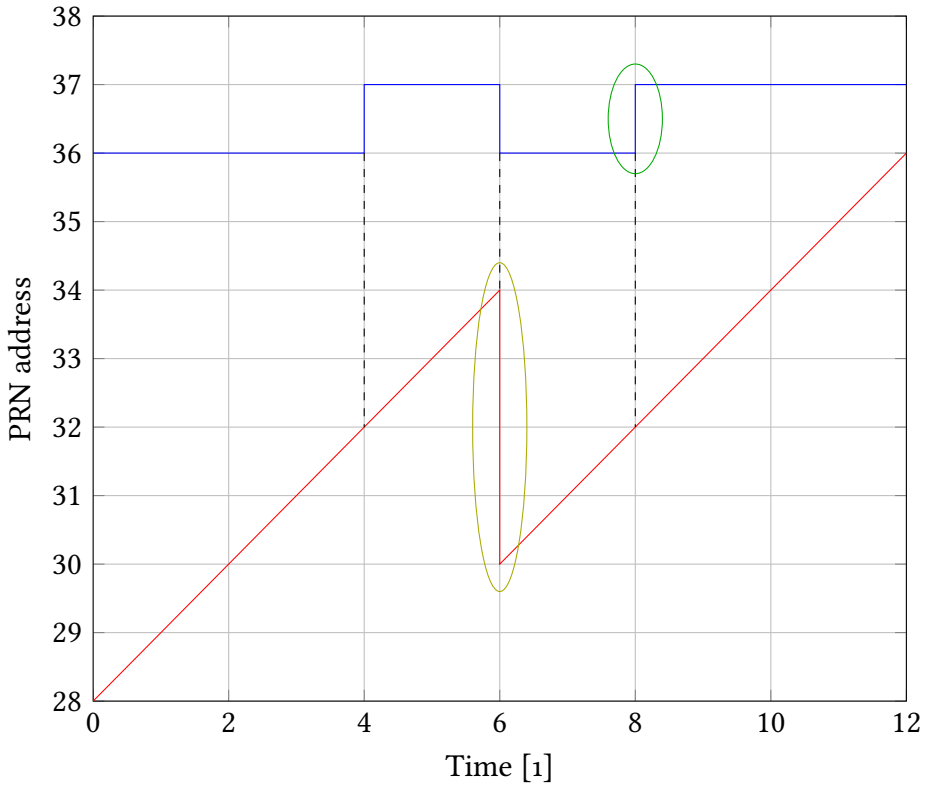


Figure 7.9: When the PRN address gets smaller, in certain circumstances this can cause an additional rising edge in the timing signal for the data and therefore in erroneous data.

To prevent this effect, a filter was developed and installed between the output of the loop controller and the offset adder. At each clock cycle, the output of this filter can only change by no more than ±1. Since the PRN code address counter

[1] A signal change from 0 to 1

only counts up by one per clock cycle, this leads to a flat line, if the output of the loop controller gets smaller, thus not allowing the PRN address counter to have a negative slope. The source code of this filter can be found in Appendix B.4.1. The effect of this filter to the PRN code address can be seen in Figure 7.10

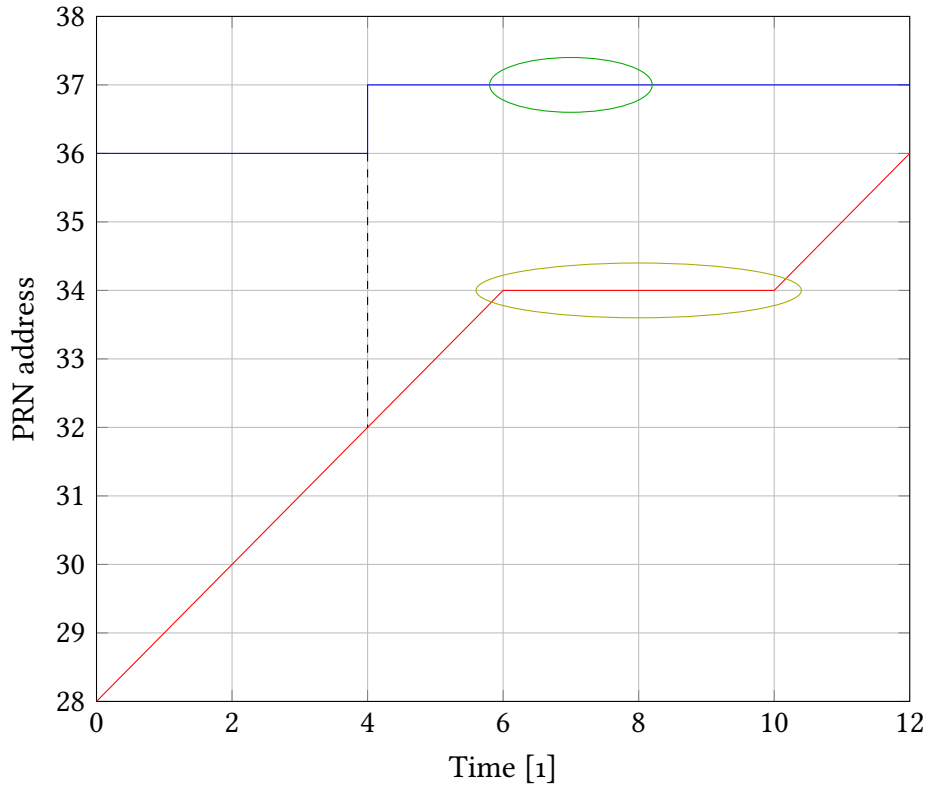


Figure 7.10: By not allowing the PRN code address to have a negative slope, the glitch in the data timing signal can be prevented. This results in a low BERs.

7.4 Measurements

In the following section, the performance of the DLL implementation will be examined. To do this, the LMS will be fed with an artificial signal that mimics a real signal that is to be expected on the LISA spacecraft. It contains the main beatnote, sidebands, pilot tone, the PRN code modulation, which will be demodulated by the DLL as well as some noise.

The artificial signal is generated by the so-called Digital Signal Simulator (DSS). This device has been developed by *Iouri Bykov* at the Albert-Einstein-Institute

in Hanover in the context of the development of the LMS[10]. It can be seen in Figure 7.11.

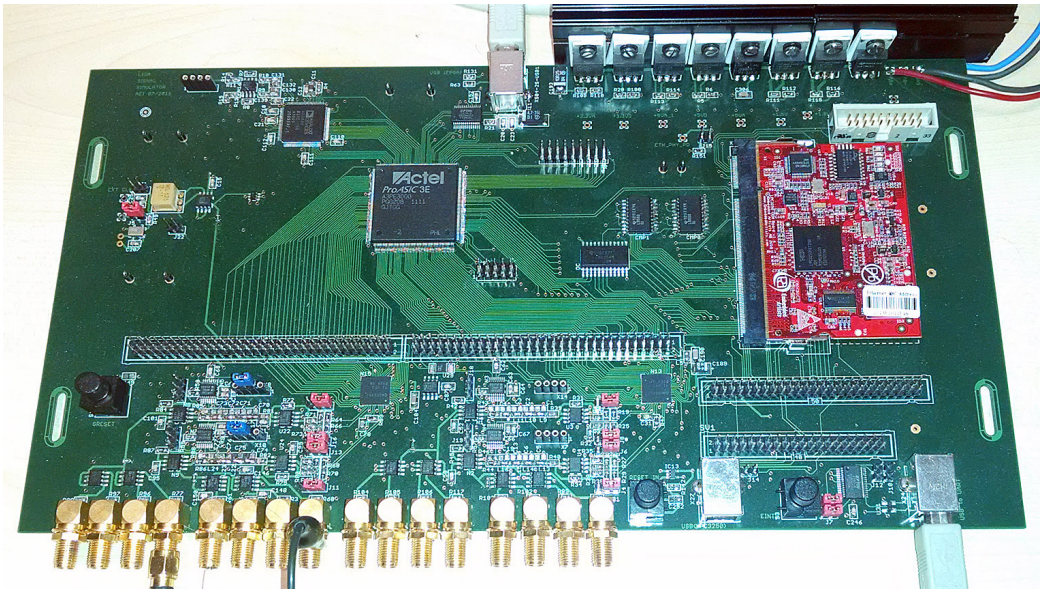


Figure 7.11: The DSS is use to create artificial signals that mimic those expected in the LISA spacecraft.

7.4.1 Timing Performance

The timing performance is measured by comparing the change in the delay measured by the DLL with the change in the measured frequency of the DPLL. Due to the slight difference in the clock frequencies of the DSS and the LMS[40], there is a measurable frequency shift on the LMS. This frequency can also be used to calculate the change in the delay, which can be compared to the change in the delay measured by the DLL.

Change in delay from the DPLL

To calculate the change in the delay from the measured frequency of the DPLL, first the difference between the clock frequencies of the DSS and the LMS must be determined.

In an experiment, the DSS has been set to a carrier frequency of $f_D = 17\,\text{MHz}$. The frequency measured by the DPLL on the LMS can be seen in Figure 7.12.

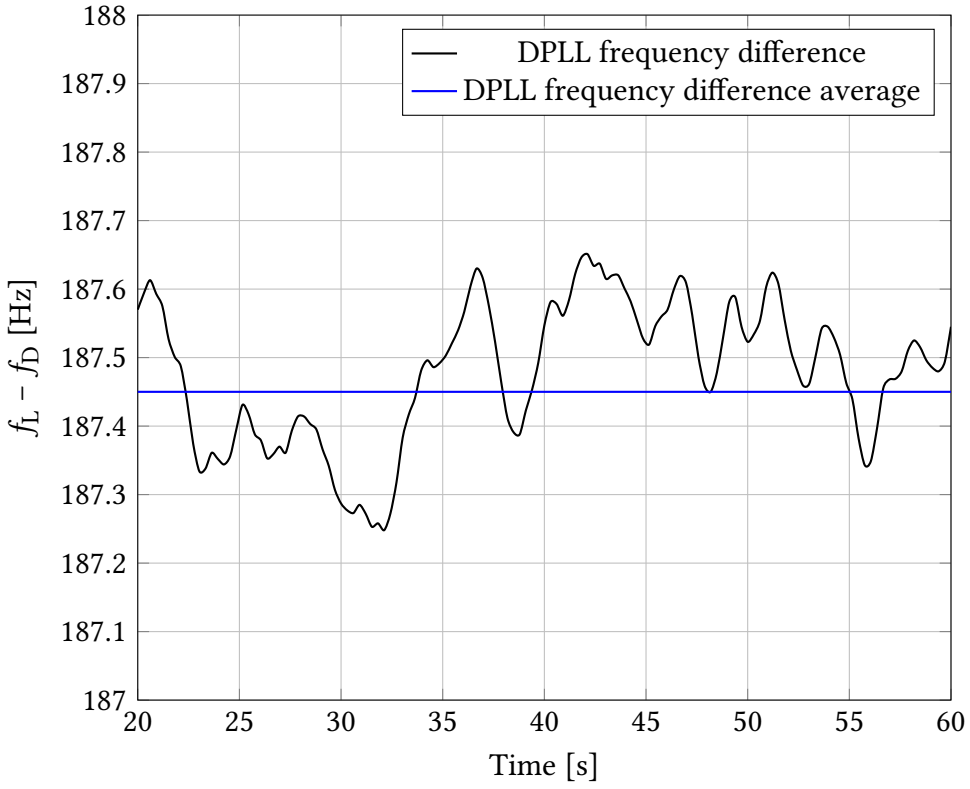


Figure 7.12: Frequency difference measured by the DPLL

It is approximately $f_L = 17.000\,187\,45\,\text{MHz}$. Since the clock frequency of both systems is $f_{clk} \approx 80\,\text{MHz}$, this leads to a clock frequency difference Δf_{clk} of:

$$\Delta f_{clk} = (f_L - f_D)\,\frac{f_D}{f_{clk}} \approx 882\,\text{Hz}\,. \tag{7.1}$$

This means that a PRN code sequence is $\frac{f_{clk}+\Delta f_{clk}}{f_{clk}} \approx 1.000011$ times longer on the DSS compared to the LMS. Since each PRN code consists of 1024 chips, each with a length of 32 clock cycles, this leads to an accumulation of delay of

$$\Delta t_{PRN} = \frac{1024 \times 32}{f_{clk}} - \frac{1024 \times 32}{f_{clk} + \Delta f_{clk}} \approx 4.52\,\text{ns} \tag{7.2}$$

each PRN code sequence. At a PRN code sequence rate of $N_{PRN} = \frac{f_{clk}}{1024\times32} \approx 2441\,\text{Hz}$, this leads to a change in delay of:

$$\Delta t = N_{PRN}\Delta t_{PRN} \approx 11\,\mu\text{s}\,\text{s}^{-1}\,. \tag{7.3}$$

Change in delay from the DLL

The delay measured by the DLL in the same period can be seen in Figure 7.13.

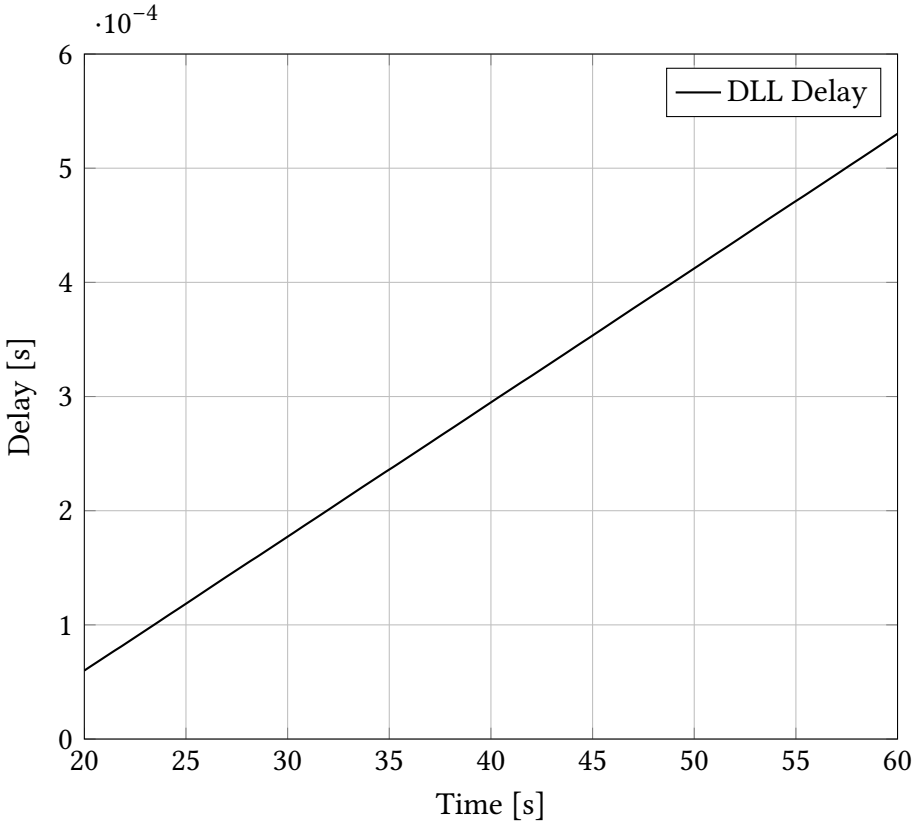


Figure 7.13: Delay measured by theDLL

The delay curve is linear and has a slope of approximately Δt =12 µs s^{-1}, which fits quite well to the result from the DPLL.

7.4.2 Bit Error Rate

In the presence of noise, the data recovery in the DLL can produce incorrect bits from time to time. A measure for the amount of these errors is the BER, which is measured in bit s^{-1}. The requirements for the LMS state, that at a bit rate of 15 kbit s^{-1}, the BER shall not be higher than 1 µbit s^{-1}[10].

In our implementation, the bit rate is b = 78.125 kbit s^{-1}, which leave a lot of room for Forward Error Correction (FEC) codes. A so-called (n,k) FEC code encodes n data bits with k code bits. This reduces the usable data rate by a factor

of $R = \frac{n}{k}$. This factor is also called the reduction rate. Due to our high bit rate, we can use (n,k) FEC codes with a reduction rate as low as $R = 0.192$.

Every set of k code bits that represents n data bits is called a valid codeword. Every other set of k code bits is called an invalid codeword. An invalid codeword should not appear in FEC encoded data and indicates an error, that may be correctable depending on the particular FEC code and the number of erroneous bits. The so-called codeword distance d of an (n,k) FEC code is the number of bits that need to be changed to get from one valid codeword to another valid codeword. Generally speaking, the higher the codeword distance d, the better the ability to correct errors. With a given bit rate b the maximum BER an (n,k) FEC is able to correct is[41]:

$$E = \frac{d-1}{2n} \cdot \frac{n}{b} \tag{7.4}$$

Table 7.1 lists a selection of FEC codes that could be used in the LMS along with their reduction rate R as well as their codeword distance d and the resulting maximum BER E they are able to correct.

Name	Reduction rate R	Code word distance d	Bit error rate E
(3,1) Hamming[42]	0.333	3	$1.28 \cdot 10^{-5}$
(5,1) Repetition	0.2	5	$2.56 \cdot 10^{-5}$
(16,4) Hadamard	0.25	8	$4.48 \cdot 10^{-5}$
(26,5) Reed-Solomon[43]	0.192	22	$1.34 \cdot 10^{-4}$

Table 7.1: An incomplete list of FEC codes, that can be used in the LMS to reduce the BER of the data demodulated by the DLL.

In lab measurements with our enhanced DLL with a CNR of up to 75 dB Hz without any FEC, a BER of up to 100 µbit s^{-1} have been measured. That means with any of the above FEC code applied the requirements could easily be achieved.

Chapter 8

Summary

In the course of this thesis, many technologies have been developed for the LMS. All of these auxiliary functions of the LMS will help to make LISA possible.

In Chapter 3 a system to acquire a DPLL lock to an unknown beatnote frequency has been developed. It uses an FFT to compute a frequency spectrum of the input signal of the LMS. The peak in this frequency spectrum is then used to get the approximate frequency of the input signal as well as its amplitude. Its frequency is used as the starting frequency for the DPLL, and the amplitude is used to calculate its initial gains. This has turned out to be very reliable, being able to establish a DPLL lock to LISA-like signals automatically.

In Chapter 4 a fully digital laser frequency offset lock has been developed. Two separate lasers are being interfered on a beam splitter, and the resulting beatnote is measured with a photodiode and digitised with an ADC. The difference frequency of the two lasers is then continuously measured with a DPLL. It is compared to a target frequency, and the resulting error value is further processed by two PI controllers. The resulting actuator values are used to change the frequency of one of the two lasers. This leads to the frequencies of the two lasers being locked to one another and thus a constant difference frequency. All this is governed by a FSM, which uses the beatnote acquisition from the last chapter to control the DPLL and the PI controllers The system performs very well, being able to establish a frequency lock between free running NPRO lasers automatically. This enables heterodyne interferometry for LISA.

In Chapter 5 the DPLL has been extended with an AGC algorithm. The amplitude of the input signal can change significantly during the operation of the LMS. Therefore the amplitude is measured continuously through the *I* output of the IQ-Demodulator in the DPLL. When the measured amplitude changes, the gains of the DPLL are regularly adjusted accordingly. This results in a stable lock of the DPLL even down to very small input amplitudes. It has been shown that this could not have been achieved without the AGC.

In Chapter 6 the DPLL has been developed further to directly track DWS signals. The DWS is an integral part of the LMS that tracks differential phases between the segments of a QPD. The differential phases usually are calculated by adding and subtracting the measured phases from four independent DPLLs, with each of them being connected to a separate segment of the QPD. The new so-called DWS DPLL developed in this chapter can track these phases directly. Apart from that it also tracks the ellipticity ϵ of the laser beam as well as the overall phase on the QPD. That results in a twice as high CNR in the DWS DPLL as compared to the traditional approach. This means that the new DWS DPLL is more resistant to noise, as several measurements have shown.

In Chapter 7 a system for absolute distance measurements as well as data transfer over the laser links has been developed. The absolute distance measurement between spacecrafts is called ranging. In the case of LISA, ranging is needed in post-processing for TDI. The data transfer function is needed because only one of the three LISA spacecrafts has a radio link to the earth. Both functionalities have been implemented using a DLL. The transmitting spacecraft modulates a PRN onto the laser beam, which is demodulated by a DPLL on the receiving spacecraft. In the DLL the demodulated PRN code is then correlated with a local copy of the same PRN code shifted by a specific delay. From the delay that results in the highest correlation, the distance between the two spacecrafts can be calculated. To transfer data between spacecrafts, data bits can be modulated onto the PRN code without interfering with the ranging. They are extracted by the DLL. In the course of this chapter, there have also been made many improvements in comparison to a previous DLL implementation written in "The MathWorks Simulink". Also FEC have been looked at to reduce the BER to meet the requirements of LISA. This allowed the DLL to operate with high stability and reliability as well as at the data rate required by LISA.

Appendix A

C++ Source Code

A.1 Beatnote Acquisition

A.1.1 C++ Simulation

```
#include <iomanip>
#include <iostream>
#include <fstream>
#include <sstream>
#include <tuple>

#include <hdlsim.hpp>

using namespace hdl;

//#define SINGLE

template<unsigned int freq_bits = 16,
         unsigned int bits = 14,
         unsigned int int_bits = 3*bits,
         unsigned int n = 2,
         unsigned int r = 4>
class gain_sim
{
private:
  // declare signals
  wire<std_logic> clk;
  wire<std_logic> clk2;
  wire<std_logic> reset;
  wire<fixed_t<false, 0, freq_bits>> freq;
  wire<fixed_t<false, 0, freq_bits>> freq_start;
  wire<fixed_t<false, 0, freq_bits>> freq_out;
  wire<fixed_t<false, 0, freq_bits>> freq_out_slow;
  wire<fixed_t<true, log2ceil(int_bits)+1, 0>> p_gain;
  wire<fixed_t<true, log2ceil(int_bits)+1, 0>> i_gain;
  wire<fixed_t<false, 0, freq_bits>> phase;
  wire<fixed_t<true, 0, bits>> sine;
  wire<fixed_t<true, 0, bits>> factor;
  wire<fixed_t<true, 0, 2*bits>> i;
  wire<fixed_t<true, 0, 2*bits>> q;
  wire<fixed_t<true, 0, 2*bits>> i_slow;
```

```
37    wire<fixed_t<true, 0, 2*bits>> q_slow;
38
39    // implement testbench
40    part testbench;
41    void tb_func(uint64_t time)
42    {
43      switch(time % 2)
44        {
45        case 0:
46          clk = 0;
47          break;
48        case 1:
49          clk = 1;
50          // slowly vary frequency
51          freq = 0.15l*sin(2.l*std::acos(-1.l)*static_cast<long double>(time)
                /100000.l)+0.25l;
52  #ifdef SINGLE
53          std::cout << time << " "
54                    << sine << " "
55                    << freq << " "
56                    << freq_out_slow << " "
57                    << i_slow << " "
58                    << q_slow << " "
59                    << std::endl;
60  #endif
61          break;
62        }
63
64      if(time < 10)
65        {
66          reset = 0;
67          freq = freq_start;
68        }
69      else
70        reset = 1;
71    }
72
73  public:
74    gain_sim()
75    {
76      // set initial values
77      freq_start = 0.25;
78      factor = 1.;
79
80      // connect components
81      nco(clk,
82          reset,
83          wire<std_logic>(1),
84          freq,
85          wire<fixed_t<false, 0, freq_bits>>(0.),
86          sine,
87          wire<fixed_t<true, 0, bits>>(),
88          wire<fixed_t<false, 0, freq_bits>>());
89
90      pll<0, int_bits>(clk,
91                       reset,
92                       wire<std_logic>(1),
93                       sine,
94                       freq_start,
95                       p_gain,
96                       i_gain,
97                       freq_out,
```

```
 98                      i,
 99                      q,
100                      q);
101
102     clkdiv<power(2, r)>(clk,
103                         reset,
104                         wire<std_logic>(1),
105                         clk2);
106
107     cic_down<n, r>(clk,
108                    clk2,
109                    reset,
110                    wire<std_logic>(1),
111                    freq_out,
112                    freq_out_slow);
113
114     cic_down<n, r>(clk,
115                    clk2,
116                    reset,
117                    wire<std_logic>(1),
118                    i,
119                    i_slow);
120
121     cic_down<n, r>(clk,
122                    clk2,
123                    reset,
124                    wire<std_logic>(1),
125                    q,
126                    q_slow);
127
128     // create testbench part
129     testbench = part({ }, { clk, reset, freq }, [this] (uint64_t time) { this->
            tb_func(time); });
130   }
131
132   ~gain_sim()
133   {
134     hdl::cleanup();
135   }
136
137   void run(unsigned int duration, int pgain, int igain)
138   {
139     p_gain = pgain;
140     i_gain = igain;
141     simulator sim(testbench);
142     sim.run(duration);
143 #ifndef SINGLE
144     std::cout << pgain << " " << igain << " " << i_slow << " " << std::endl;
145 #endif
146   }
147 };
148
149 int main()
150 {
151   wire<int> freq;
152   wire<int> freq_start;
153   freq = freq_start;
154
155 #ifdef SINGLE
156   int pgain = -3;
157   int igain = -5;
158 #else
```

```
159   // loop through gains
160   int lower = -15;
161   int upper = 5;
162   for(int pgain = lower; pgain <= upper; pgain++)
163     for(int igain = lower; igain <= upper; igain++)
164 #endif
165       {
166         gain_sim<> sim;
167         sim.run(20000, pgain, igain);
168       }
169   return 0;
170 }
```

A.2 Automatic Gain Control

A.2.1 C++ Simulation

```
1  #include <array>
2  #include <iomanip>
3  #include <iostream>
4  #include <fstream>
5  #include <sstream>
6  #include <tuple>
7
8  #include <hdlsim.hpp>
9
10 using namespace hdl;
11
12 // automatic gain control module
13 template <typename B, bool sign, unsigned int fbits, unsigned int fbits2>
14 void agc(wire<B> clk,
15          wire<B> reset,
16          wire<fixed_t<sign, 0, fbits>> amp,
17          wire<fixed_t<sign, 0, fbits2>> in,
18          wire<fixed_t<sign, 0, fbits2>> out)
19 {
20   wire<B> reset2(0);
21   wire<fixed_t<true, log2ceil(fbits2)+1, 0>> gain;
22
23   // wait for amplitude to be non-NULL until reset is lifed.
24   part({ clk, reset, },
25        { reset2 },
26        [=] (uint64_t)
27        {
28          if(reset == static_cast<B>(false))
29            reset2 = static_cast<B>(false);
30          else if(amp != fixed_t<sign, 0, fbits>(0))
31            reset2 = static_cast<B>(true);
32        }, "");
33
34   part({ clk, reset2, amp},
35        { gain },
36        [=] (uint64_t)
37        {
38          if(reset2 == static_cast<B>(false))
39            gain = fixed_t<true, log2ceil(fbits2)+1, 0>(0);
40          else
```

```
41            {
42              gain = fixed_t<true, log2ceil(fbits2)+1, 0>(0);
43              // increase gain if amplitude halves.
44              for(unsigned int c = 1; c < fbits; c++)
45                if(!amp.get().at(fbits-1-c))
46                  gain = fixed_t<true, log2ceil(fbits2)+1, 0>((signed)c-6);
47                else
48                  break;
49            }
50        }, "agc");
51
52    // apply gain
53    barrel_shift(in, gain, out);
54  }
55
56  // testbench class
57  template <unsigned int freq_bits = 16,
58            unsigned int bits = 14,
59            unsigned int int_bits = 3*bits,
60            unsigned int n = 2,
61            unsigned int r = 10>
62  class test
63  {
64  private:
65    // declare signals
66    wire<std_logic> clk;
67    wire<std_logic> clk2;
68    wire<std_logic> reset;
69    wire<fixed_t<false, 0, freq_bits>> freq;
70    wire<fixed_t<false, 0, freq_bits>> freq_start;
71    wire<fixed_t<false, 0, freq_bits>> freq_out;
72    wire<fixed_t<false, 0, freq_bits>> freq_out_slow;
73    wire<fixed_t<true, log2ceil(int_bits)+1, 0>> p_gain;
74    wire<fixed_t<true, log2ceil(int_bits)+1, 0>> i_gain;
75    wire<fixed_t<true, log2ceil(2*bits)+1, 0>> gain;
76    wire<fixed_t<true, 0, bits>> amplitude;
77    wire<fixed_t<true, 0, bits>> sine_tmp;
78    wire<fixed_t<true, 0, 2*bits>> sine_long;
79    wire<fixed_t<true, 0, bits>> sine;
80    wire<fixed_t<true, 0, 2*bits>> i;
81    wire<fixed_t<true, 0, 2*bits>> q_out;
82    wire<fixed_t<true, 0, 2*bits>> q_in;
83    wire<fixed_t<true, 0, 2*bits>> i_slow;
84    wire<fixed_t<true, 0, 2*bits>> q_slow;
85
86    // implement testbench
87    part testbench;
88    void tb_func(uint64_t time)
89    {
90      switch(time % 2)
91        {
92        case 0:
93          clk = 0;
94          break;
95        case 1:
96          clk = 1;
97          freq = 0.2+0.1l*sin(2.l*std::acos(-1.l)*static_cast<long double>(time)
                /100000.l);
98          amplitude = 0.25l+0.24l*cos(2.l*std::acos(-1.l)*static_cast<long double>(
                time)/1000000.l);
99          break;
100       }
```

```
101
102     if(time % 2048 == 0)
103       std::cout << time << " "
104                 << freq << " "
105                 << freq_out_slow << " "
106                 << i_slow << " "
107                 << q_slow << " "
108                 << amplitude << " "
109                 << std::endl;
110
111     if(time < 10)
112       {
113         reset = 0;
114         freq = freq_start;
115         amplitude = 0.5;
116       }
117     else
118       reset = 1;
119   }
120
121 public:
122   test()
123   {
124     // set initial values
125     freq_start = 0.25;
126     p_gain = -5;
127     i_gain = -7;
128
129     // connect components
130     nco(clk,
131         reset,
132         wire<std_logic>(1),
133         freq,
134         wire<fixed_t<false, 0, freq_bits>>(0.),
135         sine_tmp,
136         wire<fixed_t<true, 0, bits>>(),
137         wire<fixed_t<false, 0, freq_bits>>());
138
139     mul(sine_tmp, amplitude, sine_long); // amplitude modulation
140     round(sine_long, sine);
141
142     pll<0, int_bits>(clk,
143                      reset,
144                      wire<std_logic>(1),
145                      sine,
146                      freq_start,
147                      p_gain,
148                      i_gain,
149                      freq_out,
150                      i,
151                      q_out,
152                      q_in);
153
154     clkdiv<power(2, r)>(clk,
155                         reset,
156                         wire<std_logic>(1),
157                         clk2);
158
159     cic_down<n, r>(clk,
160                    clk2,
161                    reset,
162                    wire<std_logic>(1),
```

```
                    freq_out,
                    freq_out_slow);

    cic_down<n, r>(clk,
                    clk2,
                    reset,
                    wire<std_logic>(1),
                    i,
                    i_slow);

    cic_down<n, r>(clk,
                    clk2,
                    reset,
                    wire<std_logic>(1),
                    q_out,
                    q_slow);

#ifdef NOAGC
    assign(q_out, q_in);
#else
    // automatic gain control
    agc(clk, reset, i_slow, q_out, q_in);
#endif

    // create testbench part
    testbench = part({ }, { clk, reset, freq, amplitude }, [this] (uint64_t time)
         { this->tb_func(time); });
  }

  void run(unsigned int duration)
  {
    simulator sim(testbench);
    sim.run(duration);
  }
};

int main()
{
  test<> t;
  t.run(1000000);
  return 0;
}
```

A.3 Laser Locking

A.3.1 Automatic Algorithm

```
#include <cmath>
#include <cstdio>

#include "state_machine.h"
#include "utils.h"

#define DEBUG

// register defs
uint32_t dac1_ctrl = 0xFFFFFFFF; // slot 1
```

```
11  uint32_t adc2_ctrl = 0xFFFFFFFF; // slot 2
12  uint32_t adc3_ctrl = 0xFFFFFFFF; // slot 3
13  uint32_t adc4_ctrl = 0xFFFFFFFF; // slot 4
14  uint32_t adc5_ctrl = 0xFFFFFFFF; // slot 5
15  uint32_t adc6_ctrl = 0xFFFFFFFF; // slot 6
16
17  void update_dac1_ctrl()
18  {
19    // carry out changes to CTRL1 registers
20    write_reg(1, sRegw_dac_dsp_DSP_CTRL1, dac1_ctrl);
21  }
22
23  void update_adc2_ctrl()
24  {
25    // carry out changes to CTRL1 registers
26    write_reg(2, sRegw_adc_dsp_DSP_CTRL1, adc2_ctrl);
27  }
28
29  void update_adc3_ctrl()
30  {
31    // carry out changes to CTRL1 registers
32    write_reg(3, sRegw_adc_dsp_DSP_CTRL1, adc3_ctrl);
33  }
34
35  void update_adc4_ctrl()
36  {
37    // carry out changes to CTRL1 registers
38    write_reg(4, sRegw_adc_dsp_DSP_CTRL1, adc4_ctrl);
39  }
40
41  void update_adc5_ctrl()
42  {
43    // carry out changes to CTRL1 registers
44    write_reg(5, sRegw_adc_dsp_DSP_CTRL1, adc5_ctrl);
45  }
46
47  void update_adc6_ctrl()
48  {
49    // carry out changes to CTRL1 registers
50    write_reg(6, sRegw_adc_dsp_DSP_CTRL1, adc6_ctrl);
51  }
52
53  // ADC
54  #define MAIN_A (1 << 0)
55  #define MAIN_B (1 << 1)
56  #define MAIN_C (1 << 2)
57  #define MAIN_D (1 << 3)
58  #define PILOT_A (1 << 4)
59  #define PILOT_B (1 << 5)
60  #define PILOT_C (1 << 6)
61  #define PILOT_D (1 << 7)
62  #define SB_1 (1 << 8)
63  #define SB_2 (1 << 9)
64  #define DLL_1 (1 << 10)
65  #define DLL_2 (1 << 11)
66
67  // DAC
68  #define LOCK_1 (1 << 0)
69  #define LOCK_2 (1 << 1)
70
71  void laser_lock::write_pzt(uint32_t value)
72  {
```

```
73    if(channel == 1)
74      write_reg(1, sRegw_dac_dsp_LOCK_CH1_PZT_OFF, value);
75    else if (channel == 2)
76      write_reg(1, sRegw_dac_dsp_LOCK_CH2_PZT_OFF, value);
77  }
78
79  void laser_lock::write_temp(uint32_t value)
80  {
81    if(channel == 1)
82      write_reg(1, sRegw_dac_dsp_LOCK_CH1_TEMP_OFF, value);
83    else if(channel == 2)
84      write_reg(1, sRegw_dac_dsp_LOCK_CH2_TEMP_OFF, value);
85  }
86
87  void laser_lock::update_plls(int32_t p, int32_t i, int32_t i2)
88  {
89    int slot = (channel == 1 ? 4 : 5);
90    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_A_GAIN_P, p);
91    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_B_GAIN_P, p-5);
92    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_C_GAIN_P, p-5);
93    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_D_GAIN_P, p-5);
94    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_A_GAIN_I, i);
95    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_B_GAIN_I, i-5);
96    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_C_GAIN_I, i-5);
97    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_D_GAIN_I, i-5);
98    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_A_GAIN_I2, i2);
99    write_reg(slot, sRegw_adc_dsp_MAIN_PLL_B_GAIN_I2, i2-5);
100   write_reg(slot, sRegw_adc_dsp_MAIN_PLL_C_GAIN_I2, i2-5);
101   write_reg(slot, sRegw_adc_dsp_MAIN_PLL_D_GAIN_I2, i2-5);
102 }
103
104 void laser_lock::lock_plls(uint32_t pir, int32_t p, int32_t i, int32_t i2)
105 {
106   int slot = (channel == 1 ? 4 : 5);
107   write_reg(slot, sRegw_adc_dsp_MAIN_PLL_A_FREQ, pir);
108   write_reg(slot, sRegw_adc_dsp_MAIN_PLL_B_FREQ, pir);
109   write_reg(slot, sRegw_adc_dsp_MAIN_PLL_C_FREQ, pir);
110   write_reg(slot, sRegw_adc_dsp_MAIN_PLL_D_FREQ, pir);
111   update_plls(p, i, i2);
112
113   if(channel == 1)
114     {
115       adc4_ctrl &= ~(MAIN_A | MAIN_B | MAIN_C | MAIN_D);
116       update_adc4_ctrl();
117     }
118   else if (channel == 2)
119     {
120       adc5_ctrl &= ~(MAIN_A | MAIN_B | MAIN_C | MAIN_D);
121       update_adc5_ctrl();
122     }
123 }
124
125 void laser_lock::unlock_plls()
126 {
127   if(channel == 1)
128     {
129       adc4_ctrl |= MAIN_A | MAIN_B | MAIN_C | MAIN_D;
130       update_adc4_ctrl();
131     }
132   else if (channel == 2)
133     {
134       adc5_ctrl |= MAIN_A | MAIN_B | MAIN_C | MAIN_D;
```

```
135       update_adc5_ctrl();
136     }
137 }
138
139 void laser_lock::lock_pid(uint32_t pir, int32_t pzt_p, int32_t pzt_i, int32_t
        temp_p, int32_t temp_i)
140 {
141   if(channel == 1)
142     {
143       write_reg(1, sRegw_dac_dsp_LOCK_CH1_PIR_OFF, pir);
144       write_reg(1, sRegw_dac_dsp_LOCK_CH1_PZT_GAIN_P, pzt_p);
145       write_reg(1, sRegw_dac_dsp_LOCK_CH1_PZT_GAIN_I, pzt_i);
146       write_reg(1, sRegw_dac_dsp_LOCK_CH1_TEMP_GAIN_P, temp_p);
147       write_reg(1, sRegw_dac_dsp_LOCK_CH1_TEMP_GAIN_I, temp_i);
148       dac1_ctrl &= ~LOCK_1;
149     }
150   else if(channel == 2)
151     {
152       write_reg(1, sRegw_dac_dsp_LOCK_CH2_PIR_OFF, pir);
153       write_reg(1, sRegw_dac_dsp_LOCK_CH2_PZT_GAIN_P, pzt_p);
154       write_reg(1, sRegw_dac_dsp_LOCK_CH2_PZT_GAIN_I, pzt_i);
155       write_reg(1, sRegw_dac_dsp_LOCK_CH2_TEMP_GAIN_P, temp_p);
156       write_reg(1, sRegw_dac_dsp_LOCK_CH2_TEMP_GAIN_I, temp_i);
157       dac1_ctrl &= ~LOCK_2;
158     }
159   update_dac1_ctrl();
160 }
161
162 void laser_lock::unlock_pid()
163 {
164   if(channel == 1)
165     dac1_ctrl |= LOCK_1;
166   else if(channel == 2)
167     dac1_ctrl |= LOCK_2;
168   update_dac1_ctrl();
169 }
170
171 void laser_lock::write_sign(int sign)
172 {
173   if(channel == 1)
174     {
175       write_reg(1, sRegw_dac_dsp_LOCK_CH1_PZT_SIGN, sign > 0 ? 1 : 0);
176       write_reg(1, sRegw_dac_dsp_LOCK_CH1_TEMP_SIGN, sign > 0 ? 0 : 1);
177     }
178   else if(channel == 2)
179     {
180       write_reg(1, sRegw_dac_dsp_LOCK_CH2_PZT_SIGN, sign > 0 ? 1 : 0);
181       write_reg(1, sRegw_dac_dsp_LOCK_CH2_TEMP_SIGN, sign > 0 ? 0 : 1);
182     }
183 }
184
185 laser_lock::laser_lock(int channel)
186   : channel(channel), state(init)
187 {
188 }
189
190 void laser_lock::reset()
191 {
192   state = init;
193 }
194
195 void laser_lock::state_machine(std::shared_ptr<str_data> sdata, uint64_t cpu_cnt)
```

```
196 {
197   if(!sdata) return;
198
199   float fft_freq;
200   float fft_amp;
201   float pll_i;
202   double pll_freq;
203
204   if(channel == 1)
205     {
206       fft_freq = (sdata->s_float.fft_dsp_01_freq +
207                   sdata->s_float.fft_dsp_02_freq +
208                   sdata->s_float.fft_dsp_03_freq +
209                   sdata->s_float.fft_dsp_04_freq)/4.0;
210
211       fft_amp = (sdata->s_float.fft_dsp_01_amp +
212                  sdata->s_float.fft_dsp_02_amp +
213                  sdata->s_float.fft_dsp_03_amp +
214                  sdata->s_float.fft_dsp_04_amp)/4.0;
215
216       pll_i = minimum(minimum(sdata->s_float.adc4_dsp_main_a_i,
217                               sdata->s_float.adc4_dsp_main_b_i),
218                       minimum(sdata->s_float.adc4_dsp_main_c_i,
219                               sdata->s_float.adc4_dsp_main_d_i));
220
221       pll_freq = (sdata->s_double.adc4_dsp_main_a_pir +
222                   sdata->s_double.adc4_dsp_main_b_pir +
223                   sdata->s_double.adc4_dsp_main_c_pir +
224                   sdata->s_double.adc4_dsp_main_d_pir)/4.0;
225     }
226   else if(channel == 2)
227     {
228       fft_freq = (sdata->s_float.fft_dsp_05_freq +
229                   sdata->s_float.fft_dsp_06_freq +
230                   sdata->s_float.fft_dsp_07_freq +
231                   sdata->s_float.fft_dsp_08_freq)/3.0;
232
233       fft_amp = (sdata->s_float.fft_dsp_05_amp +
234                  sdata->s_float.fft_dsp_06_amp +
235                  sdata->s_float.fft_dsp_07_amp +
236                  sdata->s_float.fft_dsp_08_amp)/3.0;
237
238       pll_i = minimum(minimum(sdata->s_float.adc5_dsp_main_a_i,
239                               sdata->s_float.adc5_dsp_main_b_i),
240                       minimum(sdata->s_float.adc5_dsp_main_c_i,
241                               sdata->s_float.adc5_dsp_main_d_i));
242
243       pll_freq = (sdata->s_double.adc5_dsp_main_a_pir +
244                   sdata->s_double.adc5_dsp_main_b_pir +
245                   sdata->s_double.adc5_dsp_main_c_pir +
246                   sdata->s_double.adc5_dsp_main_d_pir)/4.0;
247     }
248   else
249     return;
250
251   cpu_cnt /= (6*10*3);
252   fft_freq *= 80e6;
253   pll_freq *= 80e6;
254
255 #ifdef DEBUG
256   printk("[llk %d] FFT frequency: %d kHz, FFT amplitude: %d, PLL freqency: %d kHz
       , PLL I: %d\n",
```

```
257           channel, static_cast<int32_t>(fft_freq/1000), static_cast<int32_t>(
                  fft_amp*1000),
258           static_cast<int32_t>(pll_freq/1000), static_cast<int32_t>(pll_i*1000));
259 #endif
260
261   switch(state)
262     {
263     case init:
264 #ifdef DEBUG
265       printk("[llk %d] Initialization.\n", channel);
266 #endif
267
268       // reset ADC/DAC
269       unlock_plls();
270       unlock_pid();
271
272       // initital piezo/temp values
273       cur_pzt = 0;
274       write_pzt(cur_pzt);
275       cur_temp = temp_min;
276       write_temp(cur_temp);
277       maximum_temp = 0;
278       maximum_amp = 0.0;
279       sign = 1;
280
281       old_cpu_cnt = cpu_cnt;
282       state = test;
283
284       break;
285     case test:
286 #ifdef DEBUG
287       printk("[llk %d] Test.\n", channel);
288 #endif
289
290       // test outputs
291       if(cpu_cnt - old_cpu_cnt == 1)
292         cur_pzt = -pzt_step;
293       else if(cpu_cnt - old_cpu_cnt == 2)
294         cur_pzt = 0;
295       else if(cpu_cnt - old_cpu_cnt == 3)
296         cur_pzt = pzt_step;
297       else if(cpu_cnt - old_cpu_cnt == 4)
298         cur_pzt = 0;
299       else
300         {
301           old_cpu_cnt = cpu_cnt;
302           wait_cnt = 32;
303           state = scan_temp;
304         }
305
306       write_pzt(cur_pzt);
307       break;
308     case scan_temp:
309       if(cpu_cnt - old_cpu_cnt < wait_cnt)
310         break;
311
312 #ifdef DEBUG
313       printk("[llk %d] Scanning Temperature.\n", channel);
314 #endif
315
316       // find maximum
317       if(fft_amp > maximum_amp)
```

```
318         {
319           maximum_amp = fft_amp;
320           maximum_temp = cur_temp;
321 #ifdef DEBUG
322           printk("[llk %d] New Maximum.\n", channel);
323 #endif
324         }
325 
326       if(cur_temp < temp_max)
327         {
328           cur_temp += temp_step;
329           write_temp(cur_temp);
330         }
331       else
332         state = set_temp;
333 
334       // wait for temperature to change
335       old_cpu_cnt = cpu_cnt;
336       wait_cnt = 1;
337 
338       break;
339     case set_temp:
340       if(cpu_cnt - old_cpu_cnt < wait_cnt)
341         break;
342 
343 #ifdef DEBUG
344       printk("[llk %d] Setting Temperature.\n", channel);
345 #endif
346 
347       // reset pzt
348       cur_pzt = 0;
349       write_pzt(cur_pzt);
350 
351       // go to maximum
352       cur_temp = maximum_temp;
353       write_temp(cur_temp);
354 
355       // wait for temperature to settle
356       old_cpu_cnt = cpu_cnt;
357       wait_cnt = 32;
358       state = adjust_pzt;
359 
360       break;
361     case adjust_pzt:
362       if(cpu_cnt - old_cpu_cnt < wait_cnt)
363         break;
364 
365 #ifdef DEBUG
366       printk("[llk %d] Adjust PZT.\n", channel);
367 #endif
368 
369       // calculate current sign
370       if(cur_pzt > last_pzt)
371         sign = fft_freq >= last_freq ? 1 : -1;
372       else if(cur_pzt < last_pzt)
373         sign = fft_freq >= last_freq ? -1 : 1;
374 
375       // save last value
376       last_freq = fft_freq;
377       last_pzt = cur_pzt;
378 
379       // ajust pzt
```

```
380       if(target_freq - sign*fft_freq > pzt_diff)
381         cur_pzt += pzt_step;
382       else if(target_freq - sign*fft_freq < -pzt_diff)
383         cur_pzt -= pzt_step;
384       else
385         state = lockpll;
386
387       // we're at the wrong temperature
388       if(cur_pzt <= pzt_min || cur_pzt >= pzt_max)
389         {
390 #ifdef DEBUG
391           printk("[llk %d] Wrong Temperature.\n", channel);
392 #endif
393           state = init;
394           break;
395         }
396
397       write_pzt(cur_pzt);
398
399       // wait for piezo to change
400       old_cpu_cnt = cpu_cnt;
401       wait_cnt = 1;
402
403       break;
404     case lockpll:
405       if(cpu_cnt - old_cpu_cnt < wait_cnt)
406         break;
407
408 #ifdef DEBUG
409       printk("[llk %d] Lock PLL.\n", channel);
410 #endif
411
412       if(fft_amp > amp_threshold)
413         {
414           // set initial frequency
415           uint32_t pir1 = static_cast<uint32_t>(fft_freq/80e6*pow(2, 32));
416
417           // calculate gain
418           float add_gain = log2(1./fft_amp);
419
420 #ifdef DEBUG
421           printk("[llk %d] Add Gain: %d.\n", channel, add_gain);
422 #endif
423
424           // set gains
425           cur_p_gain = p_base_gain + add_gain;
426           cur_i_gain = i_base_gain + add_gain;
427
428           // start PLLs
429           lock_plls(pir1, cur_p_gain, cur_i_gain, 0);
430
431           // save for later
432           last_p_gain = cur_p_gain;
433           last_i_gain = cur_i_gain;
434
435           // wait for PLLs to stabilize
436           old_cpu_cnt = cpu_cnt;
437           wait_cnt = 16;
438           state = lock_laser;
439         }
440       else
441         state = init;
```

```
442
443       break;
444     case lock_laser:
445       if(cpu_cnt - old_cpu_cnt < wait_cnt)
446         break;
447
448 #ifdef DEBUG
449       printk("[llk %d] Lock Laser.\n", channel);
450 #endif
451
452       // try again if PLLs unlocked
453       if(std::abs(fft_freq - pll_freq) > check_freq_diff)
454         {
455 #ifdef DEBUG
456       printk("[llk %d] PLL unlocked.\n", channel);
457 #endif
458           unlock_pid();
459           unlock_plls();
460           state = set_temp;
461           break;
462         }
463
464       // set signs and enable locks
465       write_sign(sign);
466       lock_pid(static_cast<uint32_t>(std::fabs(target_freq)/80e6*std::pow(2.0,
              32)),
467                pzt_p, pzt_i, temp_p, temp_i);
468
469       // wait for lock loop to stabilize
470       old_cpu_cnt = cpu_cnt;
471       state = reset_pzt;
472       wait_cnt = 4;
473
474       break;
475     case reset_pzt:
476       if(cpu_cnt - old_cpu_cnt < wait_cnt)
477         break;
478
479 #ifdef DEBUG
480       printk("[llk %d] Reset PZT offset.\n", channel);
481 #endif
482
483       // try again if PLLs unlocked
484       if(std::abs(fft_freq - pll_freq) > check_freq_diff)
485         {
486 #ifdef DEBUG
487           printk("[llk %d] PLL unlocked.\n", channel);
488 #endif
489           unlock_pid();
490           unlock_plls();
491           state = set_temp;
492           break;
493         }
494
495
496       // slowly remove pzt offset
497       if(cur_pzt > 8)
498         cur_pzt -= 1024*1024;
499       else if(cur_pzt < -8)
500         cur_pzt += 1024*1024;
501       else
502         state = reset_temp;
```

```
503       write_pzt(cur_pzt);
504
505       // wait for lock to follow
506       old_cpu_cnt = cpu_cnt;
507       wait_cnt = 1;
508
509       break;
510     case reset_temp:
511       if(cpu_cnt - old_cpu_cnt < wait_cnt)
512         break;
513
514 #ifdef DEBUG
515       printk("[llk %d] Reset Temperature offset.\n", channel);
516 #endif
517
518       // try again if PLLs unlocked
519       if(std::abs(fft_freq - pll_freq) > check_freq_diff)
520         {
521 #ifdef DEBUG
522           printk("[llk %d] PLL unlocked.\n", channel);
523 #endif
524           unlock_pid();
525           unlock_plls();
526           state = set_temp;
527           break;
528         }
529
530       // slowly remove temp offset
531       if(cur_temp > 8)
532         cur_temp -= 1024*1024;
533       else if(cur_temp < -8)
534         cur_temp += 1024*1024;
535       else
536         state = check;
537       write_temp(cur_temp);
538
539       // wait for lock to follow
540       old_cpu_cnt = cpu_cnt;
541       wait_cnt = 1;
542
543       break;
544     case check:
545       if(cpu_cnt - old_cpu_cnt < wait_cnt)
546         break;
547
548 #ifdef DEBUG
549       printk("[llk %d] Check.\n", channel);
550 #endif
551
552       // try again if PLLs unlocked
553       if(std::abs(fft_freq - pll_freq) > check_freq_diff)
554         {
555 #ifdef DEBUG
556           printk("[llk %d] PLL unlocked.\n", channel);
557 #endif
558
559           unlock_pid();
560           unlock_plls();
561           state = set_temp;
562           break;
563         }
564
```

```
565       // Wait a bit
566       old_cpu_cnt = cpu_cnt;
567       wait_cnt = 1;
568
569       break;
570     default:
571       // This shouldn't happen
572       state = init;
573       break;
574     }
575
576 #ifdef DEBUG
577   printk("[llk %d] cur_temp: %d, cur_pzt: %d, sign: %d\n",
578          channel, cur_temp, cur_pzt, sign);
579 #endif
580 }
```

A.4 Differential Wavefront Sensing

A.4.1 C++ Simulation

```
1  #include <iostream>
2
3  #include <hdlsim.hpp>
4
5  using namespace hdl;
6
7  template<unsigned int int_mbits, unsigned int int_fbits,
8           typename B, unsigned int mbits, unsigned int fbits, unsigned int
               freq_bits>
9  void qpd_pll(wire<B> clk,
10              wire<B> reset,
11              wire<B> enable,
12              wire<fixed_t<true, mbits, fbits>> inputa,
13              wire<fixed_t<true, mbits, fbits>> inputb,
14              wire<fixed_t<true, mbits, fbits>> inputc,
15              wire<fixed_t<true, mbits, fbits>> inputd,
16              wire<fixed_t<false, 0, freq_bits>> freq_start, // f/fs
17              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> pgain_sum,
18              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> igain_sum,
19              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> pgain_dx,
20              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> igain_dx,
21              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> pgain_dy,
22              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> igain_dy,
23              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> pgain_ell,
24              wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> igain_ell,
25              wire<fixed_t<false, 0, freq_bits>> freq_out, // f/fs
26              wire<fixed_t<true, 2*mbits, 2*fbits>> ia,
27              wire<fixed_t<true, 2*mbits, 2*fbits>> qa,
28              wire<fixed_t<true, 2*mbits, 2*fbits>> ib,
29              wire<fixed_t<true, 2*mbits, 2*fbits>> qb,
30              wire<fixed_t<true, 2*mbits, 2*fbits>> ic,
31              wire<fixed_t<true, 2*mbits, 2*fbits>> qc,
32              wire<fixed_t<true, 2*mbits, 2*fbits>> id,
33              wire<fixed_t<true, 2*mbits, 2*fbits>> qd,
34              wire<fixed_t<true, 2*mbits, 2*fbits>> errora,
35              wire<fixed_t<true, 2*mbits, 2*fbits>> errorb,
```

```
36              wire<fixed_t<true, 2*mbits, 2*fbits>> errorc,
37              wire<fixed_t<true, 2*mbits, 2*fbits>> errord,
38              wire<fixed_t<false, 0, freq_bits>> phase_sum,
39              wire<fixed_t<false, 0, freq_bits>> phase_dx,
40              wire<fixed_t<false, 0, freq_bits>> phase_dy,
41              wire<fixed_t<false, 0, freq_bits>> phase_ell,
42              wire<fixed_t<false, 0, freq_bits>> phasea,
43              wire<fixed_t<false, 0, freq_bits>> phaseb,
44              wire<fixed_t<false, 0, freq_bits>> phasec,
45              wire<fixed_t<false, 0, freq_bits>> phased)
46 {
47   wire<fixed_t<true, mbits, fbits>>
48     sinea, cosinea,
49     sineb, cosineb,
50     sinec, cosinec,
51     sined, cosined;
52
53   // IQ demodulation
54   mul(inputa, sinea, ia);
55   mul(inputa, cosinea, qa);
56   mul(inputb, sineb, ib);
57   mul(inputb, cosineb, qb);
58   mul(inputc, sinec, ic);
59   mul(inputc, cosinec, qc);
60   mul(inputd, sined, id);
61   mul(inputd, cosined, qd);
62
63   // divide error signals by 4 before adding to prevent overflow
64   wire<fixed_t<true, 2*mbits, 2*fbits>> errora2, errorb2, errorc2, errord2;
65   barrel_shift_fixed(errora, -2, errora2);
66   barrel_shift_fixed(errorb, -2, errorb2);
67   barrel_shift_fixed(errorc, -2, errorc2);
68   barrel_shift_fixed(errord, -2, errord2);
69
70   // combine error signals
71   wire<fixed_t<true, 2*mbits, 2*fbits>> error_sum, error_dx, error_dy, error_ell,
72     tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8;
73   add(errora2, errorb2, tmp1);
74   add(errorc2, errord2, tmp2);
75   add(tmp1, tmp2, error_sum);
76   sub(errora2, errorb2, tmp3);
77   sub(errorc2, errord2, tmp4);
78   add(tmp3, tmp4, error_dx);
79   sub(errora2, errorc2, tmp5);
80   sub(errorb2, errord2, tmp6);
81   add(tmp5, tmp6, error_dy);
82   sub(errora2, errorb2, tmp7);
83   sub(errord2, errorc2, tmp8);
84   add(tmp7, tmp8, error_ell);
85
86   // PID filter
87   wire<fixed_t<true, int_mbits, int_fbits>> pidout_sum, pidout_dx, pidout_dy,
         pidout_ell;
88   wire<fixed_t<true, log2ceil(int_mbits+int_fbits)+1, 0>> dgain(0.);
89   pidctl<true, true, false, int_mbits, int_fbits>
90     (clk, reset, enable, error_sum, pgain_sum, igain_sum, dgain, pidout_sum);
91   pidctl<true, true, false, int_mbits, int_fbits>
92     (clk, reset, enable, error_dx, pgain_dx, igain_dx, dgain, pidout_dx);
93   pidctl<true, true, false, int_mbits, int_fbits>
94     (clk, reset, enable, error_dy, pgain_dy, igain_dy, dgain, pidout_dy);
95   pidctl<true, true, false, int_mbits, int_fbits>
96     (clk, reset, enable, error_ell, pgain_ell, igain_ell, dgain, pidout_ell);
```

```
97
98    // resize pid results
99    wire<fixed_t<true, 0, freq_bits>> pidout_sum2, pidout_dx2, pidout_dy2,
          pidout_ell2;
100   resize(pidout_sum, pidout_sum2);
101   resize(pidout_dx, pidout_dx2);
102   resize(pidout_dy, pidout_dy2);
103   resize(pidout_ell, pidout_ell2);
104
105   // add start frequency
106   add(pidout_sum2, freq_start, freq_out);
107
108   // integrate frequency to phase
109   integrator(clk, reset, enable, freq_out, phase_sum);
110
111   wire<fixed_t<true, 0, freq_bits>> phase_dx2, phase_dy2, phase_ell2;
112   reg(clk, reset, enable, pidout_dx2, phase_dx2);
113   reg(clk, reset, enable, pidout_dy2, phase_dy2);
114   reg(clk, reset, enable, pidout_ell2, phase_ell2);
115   assign(phase_dx2, phase_dx);
116   assign(phase_dy2, phase_dy);
117   assign(phase_ell2, phase_ell);
118
119   // combine phases
120   wire<fixed_t<false, 0, freq_bits>> tmp11, tmp12, tmp13, tmp14, tmp15, tmp16,
          tmp17, tmp18;
121   add(phase_sum, phase_dx, tmp11);
122   add(phase_dy, phase_ell, tmp12);
123   add(tmp11, tmp12, phasea);
124   sub(phase_sum, phase_dx, tmp13);
125   sub(phase_dy, phase_ell, tmp14);
126   add(tmp13, tmp14, phaseb);
127   sub(phase_sum, phase_dy, tmp15);
128   sub(phase_dx, phase_ell, tmp16);
129   add(tmp15, tmp16, phasec);
130   sub(phase_sum, phase_dx, tmp17);
131   sub(phase_ell, phase_dy, tmp18);
132   add(tmp17, tmp18, phased);
133
134   // LUTs
135   sincos(phasea, sinea, cosinea);
136   sincos(phaseb, sineb, cosineb);
137   sincos(phasec, sinec, cosinec);
138   sincos(phased, sined, cosined);
139 }
140
141 template <unsigned int bits = 14,
142           unsigned int freq_bits = 16,
143           unsigned int int_bits = 3*bits>
144 class test
145 {
146 private:
147   // declare signals
148   wire<std_logic> clk;
149   wire<std_logic> clk2;
150   wire<std_logic> reset;
151   wire<fixed_t<true, 0, bits>> sine, sine2;
152   wire<fixed_t<true, 0, bits>> factor;
153   wire<fixed_t<false, 0, freq_bits>> freq;
154   wire<fixed_t<false, 0, freq_bits>> phase;
155   wire<fixed_t<false, 0, freq_bits>> freq_start;
156   wire<fixed_t<false, 0, freq_bits>> freq_out;
```

```
157   wire<fixed_t<true, 0, 2*bits>> ia, qa, ib, qb, ic, qc, id, qd;
158   wire<fixed_t<false, 0, freq_bits>> phase_sum, phase_dx, phase_dy, phase_ell,
          phasea, phaseb, phasec, phased;
159
160   // implement testbench
161   part testbench;
162   void tb_func(uint64_t time)
163   {
164     switch(time % 2)
165       {
166       case 0:
167         clk = 0;
168         break;
169       case 1:
170         clk = 1;
171         // slowly vary frequency and differential phase
172         freq = 0.1l*sin(2.l*std::acos(-1.l)*static_cast<long double>(time)
              /100000.l)+0.2l;
173         phase = 0.1l*sin(2.l*std::acos(-1.l)*static_cast<long double>(time)
              /1000000.l)+0.2l;
174         if((time+1) % 200 == 0)
175           {
176             std::cout << time << " " << freq << " " << freq_out << " "
177                       << phase << " " << phase_dx << " " << phase_dy << " " <<
                              phase_ell << " "
178                       << std::endl;
179           }
180         break;
181       }
182
183     if(time < 10)
184       reset = 0;
185     else
186       reset = 1;
187   }
188
189 public:
190   test()
191   {
192     // set initial values
193     freq_start = 0.2;
194     factor = 1.;
195
196     // connect components
197     nco(clk,
198         reset,
199         wire<std_logic>(1),
200         freq,
201         wire<fixed_t<false, 0, freq_bits>>(0.),
202         sine,
203         wire<fixed_t<true, 0, bits>>(),
204         wire<fixed_t<false, 0, freq_bits>>());
205
206     nco(clk,
207         reset,
208         wire<std_logic>(1),
209         freq,
210         phase,
211         sine2,
212         wire<fixed_t<true, 0, bits>>(),
213         wire<fixed_t<false, 0, freq_bits>>());
214
```

```
    qpd_pll<0, int_bits>(clk,
                         reset,
                         wire<std_logic>(1),
                         sine2, sine,
                         sine2, sine,
                         freq_start,
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-10),
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-12),
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-12),
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-14),
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-12),
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-14),
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-12),
                         wire<fixed_t<true, log2ceil(int_bits)+1, 0> >(-14),
                         freq_out,
                         ia, qa, ib, qb, ic, qc, id, qd,
                         qa, qb, qc, qd,
                         phase_sum,
                         phase_dx,
                         phase_dy,
                         phase_ell,
                         phasea,
                         phaseb,
                         phasec,
                         phased);

    // create testbench part
    testbench = part({ }, { clk, reset, freq, phase }, [this] (uint64_t time) {
        this->tb_func(time); });
  }

  void run(unsigned int duration)
  {
    simulator sim(testbench);
    sim.run(duration);
  }
};

int main()
{
  test<> t;
  t.run(1000000);
  return 0;
}
```

A.5 Ranging and Data Transfer

A.5.1 Ranging Spectra Generator

```
/*
 * Copyright (c) 2014, Nils Christopher Brause
 * All rights reserved.
 *
 * Permission to use, copy, modify, and/or distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
```

```
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 *
 * The views and conclusions contained in the software and documentation are
 * those of the authors and should not be interpreted as representing official
 * policies, either expressed or implied, of the Max Planck Institute for
 * Gravitational Physics (Albert Einstein Institute).
 */

#include <array>
#include <cmath>
#include <complex>
#include <fstream>
#include <functional>
#include <iostream>
#include <limits>
#include <memory>
#include <sstream>
#include <stdexcept>
#include <string>
#include <vector>
#include <fftw3.h>

long double pi = std::acos(-1.l);

enum fft_type { PS, LS, PSD, LSD };

// real fft
template <typename T, unsigned long int N>
void rfft(std::array<T, N> &input, std::array<T, N> &output, fft_type type,
          long double fs, std::function<T(T)> window)
{
  // convert to long double
  T ldn = N;

  // Window sums
  T s1 = 0;
  T s2 = 0;
  for(unsigned int c = 0; c < N; c++)
    {
      T ldc = c;
      s1 += window(ldc/ldn);
      s2 += std::pow(window(ldc/ldn), 2);
    }

  // initialize fftw
  static fftw_complex *in = NULL;
  static fftw_complex *out = NULL;
  static fftw_plan p;
  in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
  out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
  p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);

  // multiply data with window
  for(unsigned int c = 0; c < N; c++)
    {
      T ldc = c;
```

```
71        in[c][0] = input.at(c)*window(ldc/ldn);
72        in[c][1] = 0.0;
73      }
74
75    // actual fft
76    fftw_execute(p);
77
78    // Create PS(D)/LS(D)
79    for(unsigned int c = 0; c < N/2; c++)
80      {
81        std::complex<T> tmp(out[c][0], out[c][1]);
82        switch(type)
83          {
84          case PS:
85            output.at(c) = 2.l*std::norm(tmp)/std::pow(s1, 2.l);
86            break;
87          case LS:
88            output.at(c) = std::sqrt(2.l*std::norm(tmp)/std::pow(s1, 2.l));
89            break;
90          case PSD:
91            output.at(c) = 2.l*std::norm(tmp)/(fs*s2);
92            break;
93          case LSD:
94            output.at(c) = std::sqrt(2.l*std::norm(tmp)/(fs*s2));
95            break;
96          }
97      }
98  }
99
100 ////////////////////////////////////////////////////////////////////////////////
101
102 std::array<std::string, 1024> prn_lut = {
103   "110110", "111101", "101110", "101110", "010101", "100010", "101011", "001011",
104   "011110", "011001", "011110", "001001", "000011", "111110", "010101", "111110",
105   "010010", "100010", "111111", "100101", "111111", "110001", "010100", "001000",
106   "110000", "000110", "000110", "001000", "101011", "001011", "111010", "010000",
107   "000110", "110011", "000000", "100000", "001011", "110111", "111011", "110101",
108   "011000", "011101", "111101", "000000", "010111", "111000", "101111", "001101",
109   "000100", "001100", "001100", "000111", "010000", "111010", "000000", "000110",
110   "100100", "010011", "001111", "111100", "101101", "000001", "111001", "000001",
111   "101101", "011101", "000010", "110001", "101010", "000000", "011110", "101000",
112   "100100", "111111", "110000", "110101", "001000", "000000", "101111", "000001",
113   "000010", "011111", "001110", "111000", "001011", "110001", "100110", "110001",
114   "111010", "111000", "010111", "101100", "111001", "110001", "000011", "000000",
115   "101000", "001111", "011111", "111101", "101011", "100011", "011101", "000001",
116   "000000", "000100", "001010", "110111", "010010", "000111", "111011", "000111",
117   "001101", "010110", "000100", "111001", "111111", "010001", "101110", "001010",
118   "000110", "010100", "000111", "010100", "111010", "111000", "110010", "011101",
119   "001011", "001100", "011101", "001000", "001111", "001110", "101001", "111110",
120   "111001", "110100", "010101", "000001", "011111", "010110", "101010", "001100",
121   "110110", "100110", "011110", "100011", "010001", "011001", "011001", "001011",
122   "100111", "110011", "000111", "010100", "001110", "001000", "100010", "111100",
123   "000000", "101110", "110011", "111111", "110001", "010100", "001001", "000100",
124   "010001", "101101", "111010", "110110", "110010", "001001", "011000", "110011",
125   "100101", "001101", "011011", "000000", "100111", "000100", "000101", "011100",
126   "011010", "101101", "100100", "100110", "101010", "111111", "101111", "010011",
127   "010110", "010011", "100010", "101011", "010010", "100010", "101101", "010000",
128   "101010", "110010", "101001", "011110", "011000", "100011", "111011", "011110",
129   "001000", "110011", "011110", "010010", "101010", "011001", "111111", "100100",
130   "001101", "111010", "100010", "111001", "111011", "100010", "111010", "001110",
131   "001001", "111000", "011100", "100011", "001111", "011100", "100011", "001101",
132   "100001", "110101", "010000", "101000", "011000", "010011", "000110", "101101",
```

```
133   "110001", "010000", "101011", "000100", "111100", "100100", "011110", "110101",
134   "010011", "001101", "111110", "111001", "011011", "100110", "101010", "011001",
135   "100110", "010001", "010101", "010001", "101110", "111000", "111111", "011100",
136   "110111", "110000", "010001", "110000", "000001", "100100", "010110", "100001",
137   "011011", "111010", "000100", "110000", "100000", "100001", "100010", "110100",
138   "100101", "100011", "100000", "111101", "001011", "101101", "001110", "111100",
139   "010101", "011100", "111011", "111110", "110100", "011010", "101111", "110101",
140   "110100", "110001", "000111", "001100", "010001", "001011", "011000", "011011",
141   "000011", "110100", "011000", "001100", "000000", "100001", "111100", "000001",
142   "001110", "111100", "101111", "100011", "110011", "010111", "101010", "101001",
143   "001011", "100010", "110010", "011001", "110100", "101010", "110101", "011000",
144   "111110", "111001", "101000", "000001", "110001", "111101", "101100", "100010",
145   "101000", "001001", "010001", "001000", "001011", "111011", "111110", "011000",
146   "100110", "111001", "010011", "000010", "011100", "011101", "101001", "100011",
147   "000010", "010000", "010001", "100000", "000110", "110001", "110011", "011000",
148   "110000", "000111", "100101", "101101", "010001", "001001", "010110", "011011",
149   "100001", "111011", "110110", "001110", "100111", "101001", "101110", "101111",
150   "010011", "011100", "111110", "100000", "000101", "011100", "001111", "110011",
151   "011011", "100111", "000000", "000001", "000011", "010100", "110000", "110011",
152   "000010", "101011", "111011", "111001", "101101", "110110", "101001", "001100",
153   "101000", "110111", "011101", "100101", "010101", "110000", "111000", "011010",
154   "011000", "000110", "011000", "101111", "000000", "010001", "101100", "111111",
155   "111001", "000000", "110101", "100110", "101111", "111000", "001100", "100111",
156   "100010", "011111", "100101", "100101", "111010", "100111", "011011", "101100",
157   "010101", "110101", "010000", "111001", "110101", "000101", "010100", "110010",
158   "011110", "101001", "011110", "000011", "011011", "101110", "111110", "010100",
159   "011010", "111001", "011101", "000010", "001001", "001010", "000111", "001101",
160   "000110", "001110", "011011", "000010", "011111", "000000", "100100", "111101",
161   "101110", "100010", "011101", "010001", "010001", "101011", "011110", "001011",
162   "110100", "001001", "011111", "000101", "000111", "110111", "011101", "101010",
163   "100001", "011110", "011010", "000010", "000011", "010111", "101101", "010011",
164   "010100", "001111", "000011", "011001", "110100", "100100", "011100", "110101",
165   "110001", "011110", "000111", "111010", "111001", "000110", "001100", "110000",
166   "100111", "000110", "110010", "111000", "111001", "001011", "011011", "010111",
167   "110110", "100000", "001010", "110010", "011100", "011011", "011000", "011110",
168   "000100", "101000", "000100", "011101", "100101", "001010", "111111", "110101",
169   "110010", "001000", "100001", "001111", "100100", "000001", "111011", "001100",
170   "110100", "000101", "101011", "000110", "110101", "110111", "010010", "110011",
171   "101111", "110110", "100001", "111110", "101110", "100011", "101111", "101011",
172   "110000", "110000", "110010", "001111", "111000", "010110", "010101", "111000",
173   "110000", "100101", "111001", "011101", "100111", "101111", "101110", "001000",
174   "000010", "110001", "001100", "001110", "000101", "110011", "100100", "011101",
175   "111100", "111010", "110011", "000000", "011110", "110001", "101010", "110110",
176   "000110", "111111", "111001", "001011", "111111", "000000", "100010", "011100",
177   "111110", "100011", "011011", "110100", "100001", "110101", "010110", "101010",
178   "101100", "100010", "011001", "111101", "101011", "001001", "100000", "110100",
179   "101110", "000011", "101010", "010101", "000110", "000010", "001111", "010110",
180   "000111", "010100", "011111", "010101", "001010", "100001", "011001", "101011",
181   "101101", "100000", "001010", "011110", "100101", "100011", "100000", "001111",
182   "011111", "111110", "011110", "100001", "010000", "001111", "000000", "110010",
183   "011111", "001111", "110110", "111011", "001011", "111001", "110011", "011101",
184   "111000", "111111", "010110", "111100", "101111", "110111", "101101", "110100",
185   "011110", "111000", "001111", "001011", "010000", "010011", "110011", "000110",
186   "111010", "111011", "111101", "000111", "100101", "011100", "100111", "111100",
187   "011011", "110110", "010100", "011000", "011100", "100101", "111101", "011110",
188   "010101", "000000", "111001", "111110", "111101", "100010", "100111", "110100",
189   "001100", "010100", "110101", "100001", "110010", "000100", "010100", "100001",
190   "111110", "001001", "001110", "000100", "100010", "011111", "000110", "000110",
191   "011011", "100001", "111011", "111011", "100111", "000011", "110000", "011110",
192   "111100", "101110", "011000", "001100", "011111", "101110", "001011", "110101",
193   "101111", "010110", "100101", "010101", "110110", "010001", "100010", "010001",
194   "000011", "101010", "010101", "110001", "110111", "100000", "111101", "100011",
```

```
195   "110110", "101010", "000111", "011001", "000111", "110010", "110111", "100110",
196   "010100", "000010", "111011", "100010", "000111", "000101", "011110", "001110",
197   "100011", "110111", "101110", "111111", "101100", "000010", "010001", "111111",
198   "010011", "011001", "100101", "011110", "111011", "001110", "001101", "011111",
199   "110101", "000100", "000110", "100000", "001000", "101100", "000100", "000100",
200   "101110", "000011", "101111", "101101", "110111", "000011", "010001", "111101",
201   "101000", "011010", "001100", "101001", "001010", "010010", "101100", "101111",
202   "000110", "000101", "101010", "000110", "100010", "000011", "011101", "001110",
203   "011111", "011100", "110011", "000111", "001100", "101011", "111001", "000100",
204   "111010", "101010", "101101", "000101", "101011", "011110", "101000", "001101",
205   "011010", "111010", "000000", "001011", "110000", "010100", "001110", "000100",
206   "001101", "011100", "000111", "100000", "001111", "011010", "101000", "000111",
207   "101100", "000010", "111010", "101000", "100011", "100101", "011111", "010100",
208   "000001", "101001", "111111", "110010", "111110", "001100", "100110", "010001",
209   "101110", "100010", "101010", "101111", "011100", "001000", "001101", "000010",
210   "010100", "010011", "000101", "100100", "001100", "011110", "000010", "101001",
211   "010110", "011110", "000110", "010100", "100101", "010110", "110110", "010000",
212   "101100", "100001", "110011", "111011", "101100", "101011", "011100", "111101",
213   "000101", "010000", "001011", "100110", "110000", "101011", "110011", "000010",
214   "110011", "000011", "011110", "011000", "001101", "101101", "010011", "001100",
215   "010101", "001110", "000001", "100011", "010000", "010000", "011111", "111010",
216   "100100", "010010", "000001", "100100", "100000", "010001", "101101", "010100",
217   "110010", "110011", "100011", "100110", "111011", "011000", "001110", "110101",
218   "101100", "011011", "000111", "111100", "111000", "011111", "101010", "111001",
219   "101100", "001011", "100010", "010100", "111000", "010111", "101110", "011100",
220   "010010", "010011", "111100", "011110", "000011", "011111", "110101", "111001",
221   "110001", "100111", "000011", "011100", "100011", "100000", "000010", "000101",
222   "010010", "011010", "110101", "001110", "011101", "101000", "000000", "110110",
223   "111001", "011101", "000011", "111110", "100001", "111001", "100001", "100100",
224   "100100", "001001", "000001", "100111", "110001", "000100", "001100", "001011",
225   "010001", "000101", "110011", "010000", "001011", "000101", "010000", "010000",
226   "111001", "011100", "111111", "110111", "000010", "011010", "001000", "011110",
227   "100001", "111000", "010111", "010100", "000110", "101101", "111011", "100011",
228   "111111", "001101", "100000", "101011", "001000", "100100", "110111", "010001",
229   "011000", "000010", "001000", "110000", "100000", "111101", "101001", "001111",
230   "101100", "111000", "011111", "010011", "101001", "000011", "010001", "111011"
          };
231
232 // standard hanning window
233 long double hanning(long double x)
234 {
235   return 0.5*(1.0-cos(2.0*pi*x));
236 }
237
238 // Flat-Top window from GEO600
239 long double hft248d(long double x)
240 {
241   long double z = 2.l*pi*x;
242   return 1.l - 1.985844164102l*cos(z) + 1.791176438506l*cos(2.l*z)
243     - 1.282075284005l*cos(3.l*z) + 0.667777530266l*cos(4.l*z)
244     - 0.240160796576l*cos(5.l*z) + 0.056656381764l*cos(6.l*z)
245     - 0.008134974479l*cos(7.l*z) + 0.000624544650l*cos(8.l*z)
246     - 0.000019808998l*cos(9.l*z) + 0.000000132974l*cos(10.l*z);
247 }
248
249 int main(int argc, char *argv[])
250 {
251   const int size = 1024;
252   const int mul = 1024;
253   const long double fs = 80e6;
254   const long double freq = 10e6;
255
```

```
256   const int code = 0;
257   const long double rate = 2.5e6;
258
259   std::array<long double, size*mul> signal;
260   std::vector<std::array<long double, size*mul>> spectra;
261
262   for(auto &depth : {0.02l*pi, 0.04l*pi, 0.08l*pi, 0.16l*pi})
263     {
264       // create a signal
265       long double time = 0;
266       for(int c = 0; c < size*mul; c++, time += 1.l/fs)
267         {
268           int prn = prn_lut.at(static_cast<unsigned int>(time*rate)%1024).at(code
                ) == '0' ? 0 : 1;
269           long double phase = 2*pi*freq*time + prn*depth*pi; // no data
270           signal.at(c) = sin(phase);
271         }
272
273       // do the fft
274       std::array<long double, size*mul> spectrum;
275       rfft<long double, size*mul>(signal, spectrum, PS, fs, hft248d);
276       spectra.push_back(spectrum);
277     }
278
279   // output
280   for(int c = 0; c < size/2-1; c++)
281     {
282       std::cout << static_cast<long double>(c)/size*fs << " ";
283       for(auto &spectrum : spectra)
284         {
285           // reduce points
286           long double max = 0;
287           for(int d = 0; d < mul-1; d++)
288             max = spectrum.at(c*mul+d) > max ? spectrum.at(c*mul+d) : max;
289           std::cout << 10.l*std::log10(max) << " ";
290         }
291       std::cout << std::endl;
292     }
293
294   return 0;
295 }
```

Appendix B

VHDL Source Code

B.1 Beatnote Acquisition

B.1.1 Fast Fourier Transform

```
1  -- Copyright (c) 2013, Nils Christopher Brause
2  -- All rights reserved.
3  --
4  -- Permission to use, copy, modify, and/or distribute this software for any
5  -- purpose with or without fee is hereby granted, provided that the above
6  -- copyright notice and this permission notice appear in all copies.
7  --
8  -- THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
9  -- WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
10 -- MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
11 -- ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
12 -- WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
13 -- ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
14 -- OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
15 --
16 -- The views and conclusions contained in the software and documentation are
17 -- those of the authors and should not be interpreted as representing official
18 -- policies, either expressed or implied, of the Max Planck Institute for
19 -- Gravitational Physics (Albert Einstein Institute).
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use ieee.numeric_std.all;
24 use ieee.math_real.all;
25 use work.utils.all;
26
27 entity sfft is
28   generic (
29     bits    : natural := 16;
30     radix   : natural := 2;   --! only supported value atm.
31     logbins : natural := 8;
32     single  : boolean := false;
33     stage   : natural := 0);
34   port (
35     clk          : in  std_logic;
36     reset        : in  std_logic;
```

```
37      input_real   : in  std_logic_vector(bits-1 downto 0);
38      input_imag   : in  std_logic_vector(bits-1 downto 0);
39      input_valid  : in  std_logic;
40      output_real  : out std_logic_vector(bits-1 downto 0);
41      output_imag  : out std_logic_vector(bits-1 downto 0);
42      output_valid : out std_logic;
43      output_bin   : out std_logic_vector(log2ceil(radix**logbins)-1 downto 0));
44  end entity sfft;
45
46  architecture behav of sfft is
47
48    constant bins       : natural := radix**logbins;
49    constant log2bins   : natural := log2ceil(bins);  -- needed for vector widths
50    constant butterflys : natural := sel(single, bins/radix, logbins*bins/radix);
51    constant bfs_bits   : natural := log2ceil(butterflys);
52    constant phase_bits : natural := log2ceil(radix**logbins);
53
54    type bin_array is array (natural range<>)
55      of std_logic_vector(log2bins-1 downto 0);
56    type phase_array is array (natural range<>)
57      of std_logic_vector(phase_bits-1 downto 0);
58    type bits_array is array (natural range<>)
59      of std_logic_vector(bits-1 downto 0);
60
61    type flycfg_t is record
62      x     : bin_array(0 to radix-1);
63      phase : phase_array(0 to radix-1);
64      sin   : bits_array(0 to radix-1);
65      cos   : bits_array(0 to radix-1);
66    end record flycfg_t;
67
68    type flycfgs_t is array (0 to 2**bfs_bits-1) of flycfg_t;
69
70    function make_flycfgs return flycfgs_t is
71      variable result : flycfgs_t;
72      variable n : natural := 0;
73      variable cmax : natural := 0;
74    begin
75      n := 0;
76      if single = false then
77        cmax := logbins-1;
78      else
79        cmax := stage;
80      end if;
81      for c in stage to cmax loop
82        for d in 0 to radix**(logbins-c-1)-1 loop
83          for e in 0 to radix**c-1 loop
84            for f in 0 to radix-1 loop
85              result(n).x(f)
86                := std_logic_vector(to_unsigned(radix**(c+1)*d+e+f*radix**c,
87                                                logbins));
88              result(n).phase(f)
89                := std_logic_vector(to_unsigned(f*e*2**phase_bits/radix**(c+1),
90                                                phase_bits));
91              result(n).cos(f) := icos(f*e, radix**(c+1), bits);
92              result(n).sin(f) := std_logic_vector(-signed(
93                isin(f*e, radix**(c+1), bits)));
94            end loop;  -- f
95            n := n + 1;
96          end loop;  -- e
97        end loop;  -- d
98      end loop;  -- c
```

```
 99    return result;
100  end make_flycfgs;
101
102  constant flycfgs : flycfgs_t := make_flycfgs;
103
104  constant last_bf : std_logic_vector(bfs_bits-1 downto 0)
105    := std_logic_vector(to_unsigned(butterflys-1, bfs_bits));
106
107  signal bf_counter            : std_logic_vector(bfs_bits-1 downto 0);
108  signal bf_counter_reset      : std_logic;
109  signal bf_counter_reset_tmp  : std_logic;
110  signal bf_counter_enable     : std_logic;
111  signal bf_counter_enable_tmp : std_logic;
112
113  signal cur_bf            : flycfg_t;
114
115  ------------------------------------------------------------------------------
116
117  constant state_bits : natural := 4;
118  subtype state_t is std_logic_vector(state_bits-1 downto 0);
119
120  signal state      : state_t;
121  signal next_state : state_t;
122
123  constant idle     : state_t := x"0";
124  constant inp      : state_t := x"1";
125  constant wait1    : state_t := x"2";
126  constant ramread  : state_t := x"3";
127  constant busy1    : state_t := x"4";
128  constant busy2    : state_t := x"5";
129  constant ramwrite : state_t := x"6";
130  constant wait2    : state_t := x"7";
131  constant outp     : state_t := x"8";
132  constant wait3    : state_t := x"9";
133
134  signal busy : std_logic;
135
136  signal bin_counter            : std_logic_vector(logbins-1 downto 0);
137  signal bin_counter_reset      : std_logic;
138  signal bin_counter_reset_tmp  : std_logic;
139  signal bin_counter_enable     : std_logic;
140  signal bin_counter_enable_tmp : std_logic;
141
142  constant bin_max : std_logic_vector(logbins-1 downto 0)
143    := (others => '1');
144
145  ------------------------------------------------------------------------------
146
147  signal we1       : std_logic;
148  signal we2       : std_logic;
149  signal sel1      : std_logic_vector(log2bins-1 downto 0);
150  signal sel2      : std_logic_vector(log2bins-1 downto 0);
151  signal real_in1  : std_logic_vector(bits-1 downto 0);
152  signal real_in2  : std_logic_vector(bits-1 downto 0);
153  signal imag_in1  : std_logic_vector(bits-1 downto 0);
154  signal imag_in2  : std_logic_vector(bits-1 downto 0);
155  signal real_out1 : std_logic_vector(bits-1 downto 0);
156  signal real_out2 : std_logic_vector(bits-1 downto 0);
157  signal imag_out1 : std_logic_vector(bits-1 downto 0);
158  signal imag_out2 : std_logic_vector(bits-1 downto 0);
159
160  ------------------------------------------------------------------------------
```

```
161
162   signal input1_real  : std_logic_vector(bits-1 downto 0);
163   signal input1_imag  : std_logic_vector(bits-1 downto 0);
164   signal input2_real  : std_logic_vector(bits-1 downto 0);
165   signal input2_imag  : std_logic_vector(bits-1 downto 0);
166   signal output1_real : std_logic_vector(bits-1 downto 0);
167   signal output1_imag : std_logic_vector(bits-1 downto 0);
168   signal output2_real : std_logic_vector(bits-1 downto 0);
169   signal output2_imag : std_logic_vector(bits-1 downto 0);
170
171   ------------------------------------------------------------------------------
172
173   signal output_real_tmp   : std_logic_vector(bits-1 downto 0);
174   signal output_imag_tmp   : std_logic_vector(bits-1 downto 0);
175   signal output_valid_tmp : std_logic;
176   signal output_valid_tmp2 : std_logic;
177   signal bin_num_tmp      : std_logic_vector(log2bins-1 downto 0);
178   signal bin_num_tmp2      : std_logic_vector(log2bins-1 downto 0);
179
180 begin  -- architecture behav
181
182   ------------------------------------------------------------------------------
183   -- State Machine
184   ------------------------------------------------------------------------------
185
186   state_reg: entity work.reg
187     generic map (
188       bits => state_bits)
189     port map (
190       clk      => clk,
191       reset    => reset,
192       enable   => '1',
193       data_in  => next_state,
194       data_out => state);
195
196   -- status signals
197   output_valid_tmp <= '1' when state = outp else
198                       '0';
199
200   bin_num_tmp <= bin_counter;
201
202   -- memory control
203   we1 <= '1' when (state = inp and input_valid = '1') or state = ramwrite else
204          '0';
205
206   we2 <= '1' when state = ramwrite else
207          '0';
208
209   sel1 <= bitreverse(bin_counter) when state = inp
210           and (single = false or stage = 0) else
211           bin_counter when state = inp and single = true and stage > 0 else
212           cur_bf.x(0) when state = ramread or busy = '1'
213           or state = ramwrite else
214           (others => '0');
215
216   sel2 <= bin_counter when state = outp else
217           cur_bf.x(1) when state = ramread or busy = '1'
218           or state = ramwrite else
219           (others => '0');
220
221   -- data flow to ram
222   real_in1 <= input_real when state = inp else
```

```
223               output1_real when state = ramwrite else
224               (others => '0');
225
226   real_in2 <= output2_real when state = ramwrite else
227               (others => '0');
228
229   imag_in1 <= input_imag when state = inp else
230               output1_imag when state = ramwrite else
231               (others => '0');
232
233   imag_in2 <= output2_imag when state = ramwrite else
234               (others => '0');
235
236   -- data flow from ram
237   input1_real <= real_out1;
238   input1_imag <= imag_out1;
239
240   input2_real <= real_out2;
241   input2_imag <= imag_out2;
242
243   output_real_tmp <= real_out2 when output_valid_tmp2 = '1' else
244                      (others => '0');
245   output_imag_tmp <= imag_out2 when output_valid_tmp2 = '1' else
246                      (others => '0');
247
248   -- counter control
249
250   bin_counter_reset <= '1' when state = inp or state = outp else
251                        '0';
252
253   bin_counter_enable <= '1' when (state = inp and input_valid = '1')
254                         or state = outp else
255                         '0';
256
257   bf_counter_reset <= '1' when state = ramread or busy = '1'
258                       or state = ramwrite else
259                       '0';
260
261   bf_counter_enable <= '1' when state = ramwrite else
262                        '0';
263
264   -- state stransitions
265   next_state <= idle when reset = '0' else
266                 -- input data
267                 inp when (state = idle and reset = '1') or state = wait3 else
268                 -- wait
269                 wait1 when state = inp and bin_counter = bin_max else
270                 -- do fft
271                 ramread when state = wait1 or (state = ramwrite
272                                                and bf_counter /= last_bf) else
273                 busy1 when state = ramread else
274                 busy2  when state = busy1 else
275                 ramwrite when state = busy2 else
276                 -- wait
277                 wait2 when state = ramwrite and bf_counter = last_bf else
278                 -- output data
279                 outp when state = wait2 else
280                 -- wait
281                 wait3 when state = outp and bin_counter = bin_max else
282                 state;
283
284   -- busy flag
```

```
285   busy <= '1' when state = busy1 or state = busy2 else '0';
286
287   -- bin counter
288   counter_1: entity work.counter
289     generic map (
290       bits      => log2bins,
291       direction => '1')
292     port map (
293       clk    => clk,
294       reset  => bin_counter_reset,
295       enable => bin_counter_enable,
296       output => bin_counter);
297
298   ------------------------------------------------------------------------------
299   -- Buttefly control
300   ------------------------------------------------------------------------------
301
302   -- butterfly counter
303   counter_2: entity work.counter
304     generic map (
305       bits      => bfs_bits,
306       direction => '1')
307     port map (
308       clk    => clk,
309       reset  => bf_counter_reset,
310       enable => bf_counter_enable,
311       output => bf_counter);
312
313   -- make synthesizable RAM
314   lutram: process (clk, reset) is
315   begin
316     if rising_edge(clk) then
317       cur_bf <= flycfgs(to_integer(unsigned(bf_counter)));
318     end if;
319   end process lutram;
320
321   -- the magic happens here
322   butterfly_1: entity work.butterfly
323     generic map (
324       bits => bits,
325       use_registers => '1')
326     port map (
327       clk          => clk,
328       reset        => reset,
329       cos_in       => cur_bf.cos(1),
330       msin_in      => cur_bf.sin(1),
331       input1_real  => input1_real,
332       input1_imag  => input1_imag,
333       input2_real  => input2_real,
334       input2_imag  => input2_imag,
335       output1_real => output1_real,
336       output1_imag => output1_imag,
337       output2_real => output2_real,
338       output2_imag => output2_imag);
339
340   ------------------------------------------------------------------------------
341   -- Memory
342   ------------------------------------------------------------------------------
343
344   -- real part
345   ram_1: entity work.ram
346     generic map (
```

```
347       bits  => bits,
348       bytes => bins)
349     port map (
350       clk1      => clk,
351       clk2      => clk,
352       we1       => we1,
353       we2       => we2,
354       addr1     => sel1,
355       addr2     => sel2,
356       data1_in  => real_in1,
357       data1_out => real_out1,
358       data2_in  => real_in2,
359       data2_out => real_out2);
360
361   -- imaginary part
362   ram_2: entity work.ram
363     generic map (
364       bits  => bits,
365       bytes => bins)
366     port map (
367       clk1      => clk,
368       clk2      => clk,
369       we1       => we1,
370       we2       => we2,
371       addr1     => sel1,
372       addr2     => sel2,
373       data1_in  => imag_in1,
374       data1_out => imag_out1,
375       data2_in  => imag_in2,
376       data2_out => imag_out2);
377
378   ------------------------------------------------------------------------------
379   -- synchronize output (RAM takes one clock period)
380   ------------------------------------------------------------------------------
381
382   reg1_valid: entity work.reg1
383     port map (
384       clk      => clk,
385       reset    => reset,
386       enable   => '1',
387       data_in  => output_valid_tmp,
388       data_out => output_valid_tmp2);
389
390   reg2_valid: entity work.reg1
391     port map (
392       clk      => clk,
393       reset    => reset,
394       enable   => '1',
395       data_in  => output_valid_tmp2,
396       data_out => output_valid);
397
398   reg1_bin_num: entity work.reg
399     generic map (
400       bits => logbins)
401     port map (
402       clk      => clk,
403       reset    => reset,
404       enable   => '1',
405       data_in  => bin_num_tmp,
406       data_out => bin_num_tmp2);
407
408   reg2_bin_num: entity work.reg
```

```
409     generic map (
410       bits => logbins)
411     port map (
412       clk      => clk,
413       reset    => reset,
414       enable   => '1',
415       data_in  => bin_num_tmp2,
416       data_out => output_bin);
417
418   reg_out_real: entity work.reg
419     generic map (
420       bits => bits)
421     port map (
422       clk      => clk,
423       reset    => reset,
424       enable   => '1',
425       data_in  => output_real_tmp,
426       data_out => output_real);
427
428   reg_out_imag: entity work.reg
429     generic map (
430       bits => bits)
431     port map (
432       clk      => clk,
433       reset    => reset,
434       enable   => '1',
435       data_in  => output_imag_tmp,
436       data_out => output_imag);
437
438 end architecture behav;
```

B.1.2 Butterfly

```
1  -- Copyright (c) 2013, Nils Christopher Brause
2  -- All rights reserved.
3  --
4  -- Permission to use, copy, modify, and/or distribute this software for any
5  -- purpose with or without fee is hereby granted, provided that the above
6  -- copyright notice and this permission notice appear in all copies.
7  --
8  -- THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
9  -- WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
10 -- MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
11 -- ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
12 -- WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
13 -- ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
14 -- OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
15 --
16 -- The views and conclusions contained in the software and documentation are
17 -- those of the authors and should not be interpreted as representing official
18 -- policies, either expressed or implied, of the Max Planck Institute for
19 -- Gravitational Physics (Albert Einstein Institute).
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use ieee.numeric_std.all;
24 use ieee.math_real.all;
25
26 entity butterfly is
```

```
27    generic (
28      bits            : natural;
29      k               : natural := 0;
30      N               : natural := 1;
31      use_kn          : bit := '0';
32      signed_arith    : bit := '1';       --! use signed arithmetic
33      use_registers   : bit := '1';       --! use additional regs on slow FPGAs
34      use_kogge_stone : bit := '0');      --! use an optimized Kogge Stone adder
35    port (
36      clk          : in  std_logic;
37      reset        : in  std_logic;
38      cos_in       : in  std_logic_vector(bits-1 downto 0);
39      msin_in      : in  std_logic_vector(bits-1 downto 0);
40      input1_real  : in  std_logic_vector(bits-1 downto 0);
41      input1_imag  : in  std_logic_vector(bits-1 downto 0);
42      input2_real  : in  std_logic_vector(bits-1 downto 0);
43      input2_imag  : in  std_logic_vector(bits-1 downto 0);
44      output1_real : out std_logic_vector(bits-1 downto 0);
45      output1_imag : out std_logic_vector(bits-1 downto 0);
46      output2_real : out std_logic_vector(bits-1 downto 0);
47      output2_imag : out std_logic_vector(bits-1 downto 0));
48  end entity butterfly;
49
50  architecture behav of butterfly is
51
52    -- sin(2*pi*k/N)
53    function isin(k : integer; N : integer) return std_logic_vector is
54      variable tmp : real;
55    begin
56      tmp := sin(real(k)/real(N)*MATH_PI*real(2))*real(2**(bits-1)-1);
57      return std_logic_vector(to_signed(integer(tmp), bits));
58    end isin;
59
60    -- cos(2*pi*k/N)
61    function icos(k : integer; N : integer) return std_logic_vector is
62      variable tmp : real;
63    begin
64      tmp := cos(real(k)/real(N)*MATH_PI*real(2))*real(2**(bits-1)-1);
65      return std_logic_vector(to_signed(integer(tmp), bits));
66    end icos;
67
68    signal cos2         : std_logic_vector(bits-1 downto 0);
69    signal msin2        : std_logic_vector(bits-1 downto 0);
70    signal input1_real2 : std_logic_vector(bits-1 downto 0);
71    signal input1_imag2 : std_logic_vector(bits-1 downto 0);
72    signal input1_real3 : std_logic_vector(bits-1 downto 0);
73    signal input1_imag3 : std_logic_vector(bits-1 downto 0);
74    signal input1_real4 : std_logic_vector(bits-1 downto 0);
75    signal input1_imag4 : std_logic_vector(bits-1 downto 0);
76    signal input2_real2 : std_logic_vector(bits-1 downto 0);
77    signal input2_imag2 : std_logic_vector(bits-1 downto 0);
78    signal input2_real3 : std_logic_vector(bits-1 downto 0);
79    signal input2_imag3 : std_logic_vector(bits-1 downto 0);
80    signal input2_real4 : std_logic_vector(bits-1 downto 0);
81    signal input2_imag4 : std_logic_vector(bits-1 downto 0);
82
83  begin  -- architecture behav
84    -- wk = exp(-2*pi*i*k) = cos(2*pi*k) - i*sin(2*pi*k)
85    -- t = x1 * wk
86    -- y0 = x0 + t
87    -- y1 = x0 - t
88
```

```
89   -- calculate wk = exp(-2*pi*i*k) = cos(2*pi*k) - i*sin(2*pi*k)
90   cos2 <= icos(k, N) when use_kn = '1' else cos_in;
91   msin2 <= std_logic_vector(-signed(isin(k, N))) when use_kn = '1' else msin_in;
92
93   -- calculate t = x1 * wk
94   cmplx_mul_1: entity work.cmplx_mul
95     generic map (
96       bits1           => bits,
97       bits2           => bits,
98       out_bits        => bits,
99       signed_arith    => signed_arith,
100      use_registers   => '0',
101      use_kogge_stone => use_kogge_stone)
102    port map (
103      clk         => clk,
104      reset       => reset,
105      input1_real => input2_real,
106      input1_imag => input2_imag,
107      input2_real => cos2,
108      input2_imag => msin2,
109      output_real => input2_real2,
110      output_imag => input2_imag2);
111
112  input1_real2 <= input1_real;
113  input1_imag2 <= input1_imag;
114
115  use_registers_yes: if use_registers = '1' generate
116    reg_input1_real: entity work.reg
117      generic map (
118        bits  => bits)
119      port map (
120        clk      => clk,
121        reset    => reset,
122        enable   => '1',
123        data_in  => input1_real2,
124        data_out => input1_real3);
125
126    reg_input1_imag: entity work.reg
127      generic map (
128        bits  => bits)
129      port map (
130        clk      => clk,
131        reset    => reset,
132        enable   => '1',
133        data_in  => input1_imag2,
134        data_out => input1_imag3);
135
136    reg_input2_real: entity work.reg
137      generic map (
138        bits  => bits)
139      port map (
140        clk      => clk,
141        reset    => reset,
142        enable   => '1',
143        data_in  => input2_real2,
144        data_out => input2_real3);
145
146    reg_input2_imag: entity work.reg
147      generic map (
148        bits  => bits)
149      port map (
150        clk      => clk,
```

```
151         reset    => reset,
152         enable   => '1',
153         data_in  => input2_imag2,
154         data_out => input2_imag3);
155   end generate use_registers_yes;
156
157   use_registers_no: if use_registers = '0' generate
158     input1_real3 <= input1_real2;
159     input1_imag3 <= input1_imag2;
160     input2_real3 <= input1_real2;
161     input2_imag3 <= input1_imag2;
162   end generate use_registers_no;
163
164   -- attenuation to prevent overflow
165   input1_real4 <= input1_real3(bits-1) & input1_real3(bits-1 downto 1);
166   input1_imag4 <= input1_imag3(bits-1) & input1_imag3(bits-1 downto 1);
167   input2_real4 <= input2_real3(bits-1) & input2_real3(bits-1 downto 1);
168   input2_imag4 <= input2_imag3(bits-1) & input2_imag3(bits-1 downto 1);
169
170   -- calculate y0 = x0 + t
171   cmplx_add_1: entity work.cmplx_add
172     generic map (
173       bits             => bits,
174       use_registers    => '0',
175       use_kogge_stone => use_kogge_stone)
176     port map (
177       clk         => clk,
178       reset       => reset,
179       input1_real => input1_real4,
180       input1_imag => input1_imag4,
181       input2_real => input2_real4,
182       input2_imag => input2_imag4,
183       output_real => output1_real,
184       output_imag => output1_imag,
185       overflow    => open);
186
187   -- calculate y1 = x0 - t
188   cmplx_sub_1: entity work.cmplx_sub
189     generic map (
190       bits             => bits,
191       use_registers    => '0',
192       use_kogge_stone => use_kogge_stone)
193     port map (
194       clk         => clk,
195       reset       => reset,
196       input1_real => input1_real4,
197       input1_imag => input1_imag4,
198       input2_real => input2_real4,
199       input2_imag => input2_imag4,
200       output_real => output2_real,
201       output_imag => output2_imag,
202       underflow   => open);
203 end architecture behav;
```

B.1.3 Peak Finder

```
1 -- Copyright (c) 2012, Nils Christopher Brause
2 -- All rights reserved.
3 --
```

```
4  -- Permission to use, copy, modify, and/or distribute this software for any
5  -- purpose with or without fee is hereby granted, provided that the above
6  -- copyright notice and this permission notice appear in all copies.
7  --
8  -- THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
9  -- WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
10 -- MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
11 -- ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
12 -- WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
13 -- ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
14 -- OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
15 --
16 -- The views and conclusions contained in the software and documentation are
17 -- those of the authors and should not be interpreted as representing official
18 -- policies, either expressed or implied, of the Max Planck Institute for
19 -- Gravitational Physics (Albert Einstein Institute).
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use ieee.numeric_std.all;
24
25 --! Maximum detector
26
27 --! The maximum detector receives a set of number-value pairs and gives out the
28 --! the number of the highest value and the value itself.
29 entity maximum is
30   generic (
31     value_bits : natural;
32     num_bits   : natural);
33   port (
34     clk         : in  std_logic;        --! clock input
35     reset       : in  std_logic;        --! asynchronous reset (active low)
36     input_value : in  std_logic_vector(value_bits-1 downto 0);  --! value
37     input_num   : in  std_logic_vector(num_bits-1 downto 0);  --! number
38     input_valid : in  std_logic;        --! value and number are valid
39     input_first : in  std_logic;        --! first value-number pair (resets max)
40     input_last  : in  std_logic;        --! last value-number pair (outputs max)
41     exclude0    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num0
42     exclude1    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num1
43     exclude2    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num2
44     exclude3    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num3
45     exclude4    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num4
46     exclude5    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num5
47     exclude6    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num6
48     exclude7    : in  std_logic_vector(num_bits-1 downto 0);  --! excluded num7
49     maximum     : out std_logic_vector(num_bits-1 downto 0);  --! max. number
50     max_value   : out std_logic_vector(value_bits-1 downto 0);  --! max. value
51     new_maximum : out std_logic);       --! maximum computation finished
52 end entity maximum;
53
54 architecture behav of maximum is
55
56   signal max_val_in    : std_logic_vector(value_bits-1 downto 0);
57   signal max_val_out   : std_logic_vector(value_bits-1 downto 0);
58   signal max_num_in    : std_logic_vector(num_bits-1 downto 0);
59   signal max_num_out   : std_logic_vector(num_bits-1 downto 0);
60   signal last          : std_logic;
61   signal done          : std_logic;
62   signal found_new_max : std_logic;
63
64 begin  -- architecture behav
65
```

```
66  found_new_max <= '1' when (unsigned(input_value) > unsigned(max_val_out)
67                             or input_first = '1') and input_valid = '1'
68                    and input_num /= exclude0 and input_num /= exclude1
69                    and input_num /= exclude2 and input_num /= exclude3
70                    and input_num /= exclude4 and input_num /= exclude5
71                    and input_num /= exclude6 and input_num /= exclude7
72                    else '0';
73
74  max_val_in <= input_value when found_new_max = '1' else
75                max_val_out;
76
77  max_num_in <= input_num when found_new_max = '1' else
78                max_num_out;
79
80  reg_val: entity work.reg
81    generic map (
82      bits => value_bits)
83    port map (
84      clk      => clk,
85      reset    => reset,
86      enable   => '1',
87      data_in  => max_val_in,
88      data_out => max_val_out);
89
90  reg_num: entity work.reg
91    generic map (
92      bits => num_bits)
93    port map (
94      clk      => clk,
95      reset    => reset,
96      enable   => '1',
97      data_in  => max_num_in,
98      data_out => max_num_out);
99
100 last <= input_last and input_valid;
101
102 -- 'done' asserts just after the last number-value pair.
103 reg1_last: entity work.reg1
104   port map (
105     clk      => clk,
106     reset    => reset,
107     enable   => '1',
108     data_in  => last,
109     data_out => done);
110
111 reg1_new_max: entity work.reg1
112   port map (
113     clk      => clk,
114     reset    => reset,
115     enable   => '1',
116     data_in  => done,
117     data_out => new_maximum);
118
119 reg_val2: entity work.reg
120   generic map (
121     bits => value_bits)
122   port map (
123     clk      => clk,
124     reset    => reset,
125     enable   => done,
126     data_in  => max_val_out,
127     data_out => max_value);
```

```
128
129   reg_num2: entity work.reg
130     generic map (
131       bits => num_bits)
132     port map (
133       clk      => clk,
134       reset    => reset,
135       enable   => done,
136       data_in  => max_num_out,
137       data_out => maximum);
138
139 end architecture behav;
```

B.2 Automatic Gain Control

B.2.1 Implementation

```
1  -- Copyright (c) 2016, Nils Christopher Brause
2  -- All rights reserved.
3  --
4  -- Permission to use, copy, modify, and/or distribute this software for any
5  -- purpose with or without fee is hereby granted, provided that the above
6  -- copyright notice and this permission notice appear in all copies.
7  --
8  -- THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
9  -- WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
10 -- MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
11 -- ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
12 -- WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
13 -- ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
14 -- OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
15 --
16 -- The views and conclusions contained in the software and documentation are
17 -- those of the authors and should not be interpreted as representing official
18 -- policies, either expressed or implied, of the Max Planck Institute for
19 -- Gravitational Physics (Albert Einstein Institute).
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use ieee.numeric_std.all;
24 use work.log2.all;
25
26 --! Automatic Gain controller
27
28 entity agc is
29   generic (
30     bits     : natural;
31     gainbits : natural);
32   port (
33     clk       : in  std_logic;
34     reset     : in  std_logic;
35     amplitude : in  std_logic_vector(bits-1 downto 0);
36     pgain_in  : in  std_logic_vector(gainbits-1 downto 0);
37     igain_in  : in  std_logic_vector(gainbits-1 downto 0);
38     pgain_out : out std_logic_vector(gainbits-1 downto 0);
39     igain_out : out std_logic_vector(gainbits-1 downto 0));
40 end entity agc;
```

```
41
42  architecture behav of agc is
43
44    -- state machine
45    constant statebits : natural := 3;
46    subtype state_t is std_logic_vector(statebits-1 downto 0);
47    constant rst : state_t := "000";
48    constant idle : state_t := "001";
49    constant wait1 : state_t := "010";
50    constant wait2 : state_t := "011";
51    constant wait3 : state_t := "100";
52    constant wait4 : state_t := "101";
53    constant scan : state_t := "110";
54    constant run : state_t := "111";
55    signal state : state_t := idle;
56    signal next_state : state_t := idle;
57
58    signal amplitude2 : std_logic_vector(bits-1 downto 0);
59    signal amount_tmp : std_logic_vector(log2ceil(bits) downto 0);
60    signal full_enable : std_logic;
61    signal full : std_logic_vector(log2ceil(bits) downto 0);
62    signal amount : std_logic_vector(log2ceil(bits) downto 0);
63    signal shift : std_logic_vector(log2ceil(bits) downto 0);
64    signal shift2 : std_logic_vector(gainbits-1 downto 0);
65    signal shift3 : std_logic_vector(gainbits-1 downto 0);
66
67    function clz (input : std_logic_vector(bits-1 downto 0))
68      return std_logic_vector is
69    begin
70      for c in 0 to bits-1 loop
71        if input(bits-1-c) = '1' then
72          return std_logic_vector(to_unsigned(c, gainbits));
73        end if;
74      end loop;  -- c
75      return std_logic_vector(to_unsigned(bits, gainbits));
76    end clz;
77
78  begin  -- architecture behav
79
80    amplitude_reg: entity work.reg
81      generic map (
82        bits => bits)
83      port map (
84        clk      => clk,
85        reset    => reset,
86        enable   => '1',
87        data_in  => amplitude,
88        data_out => amplitude2);
89
90    -- count leading zeros
91    amount_tmp <= clz(amplitude2);
92
93    state_reg: entity work.reg
94      generic map (
95        bits => statebits)
96      port map (
97        clk      => clk,
98        reset    => reset,
99        enable   => '1',
100       data_in  => next_state,
101       data_out => state);
102
```

```
103   next_state <= rst when reset = '0' else
104                 idle when state = rst and reset = '1' else
105                 wait1 when state = idle and unsigned(amplitude2) /= to_unsigned
                        (0, bits) else
106                 wait2 when state = wait1 else
107                 wait3 when state = wait2 else
108                 wait4 when state = wait3 else
109                 scan when state = wait4 else
110                 run when state = scan else
111                 state;
112
113   full_enable <= '1' when state = scan else '0';
114
115   full_reg: entity work.reg
116     generic map (
117       bits => log2ceil(bits)+1)
118     port map (
119       clk      => clk,
120       reset    => reset,
121       enable   => full_enable,
122       data_in  => amount_tmp,
123       data_out => full);
124
125   sub_1: entity work.sub
126     generic map (
127       bits            => log2ceil(bits)+1,
128       use_registers   => '0',
129       use_kogge_stone => '0')
130     port map (
131       clk        => clk,
132       reset      => reset,
133       input1     => amount_tmp,
134       input2     => full,
135       output     => amount,
136       borrow_in  => '0',
137       borrow_out => open,
138       underflow  => open);
139
140   shift <= amount when state = run else
141            (others => '0');
142
143   shift2(log2ceil(bits)-1 downto 0) <= shift(log2ceil(bits)-1 downto 0);
144   shift2(gainbits-1 downto log2ceil(bits)) <= (others => shift(log2ceil(bits)));
145
146   pgain_add: entity work.add
147     generic map (
148       bits            => log2ceil(bits)+1,
149       use_registers   => '1',
150       use_kogge_stone => '0')
151     port map (
152       clk       => clk,
153       reset     => reset,
154       input1    => pgain_in,
155       input2    => shift2,
156       output    => pgain_out,
157       carry_in  => '0',
158       carry_out => open,
159       overflow  => open);
160
161   igain_add: entity work.add
162     generic map (
163       bits            => log2ceil(bits)+1,
```

```
164       use_registers   => '1',
165       use_kogge_stone => '0')
166     port map (
167       clk       => clk,
168       reset     => reset,
169       input1    => igain_in,
170       input2    => shift2,
171       output    => igain_out,
172       carry_in  => '0',
173       carry_out => open,
174       overflow  => open);
175
176 end architecture behav;
```

B.2.2 Testbench

```
 1 library ieee;
 2 use ieee.std_logic_1164.all;
 3 use ieee.numeric_std.all;
 4 use ieee.math_real.all;
 5 use std.textio.all;
 6 use work.log2.all;
 7
 8 entity testbench is
 9
10 end entity testbench;
11
12 architecture behav of testbench is
13
14   constant bits      : natural := 14;
15   constant nco_bits  : natural := bits;
16   constant lut_bits  : natural := bits;
17   constant int_bits  : natural := 3*bits;
18   constant freq_bits : natural := 16;
19
20   constant n : natural := 2;
21   constant r : natural := 10;
22
23   constant signed_arith    : bit := '1';
24   constant use_registers   : bit := '0';
25   constant use_kogge_stone : bit := '0';
26
27   signal clk   : std_logic := '0';
28   signal clk2  : std_logic;
29   signal reset : std_logic;
30   signal t     : natural := 0;
31
32   signal freq : std_logic_vector(freq_bits-1 downto 0);
33   signal amp  : std_logic_vector(bits-1 downto 0);
34   signal sin1 : std_logic_vector(bits-1 downto 0);
35   signal sin2 : std_logic_vector(2*bits-1 downto 0);
36   signal sin3 : std_logic_vector(bits-1 downto 0);
37
38   signal i          : std_logic_vector(bits+nco_bits-1 downto 0);
39   signal i_slow     : std_logic_vector(bits+nco_bits-1 downto 0);
40   signal i_abs      : std_logic_vector(bits+nco_bits-1 downto 0);
41   signal q          : std_logic_vector(bits+nco_bits-1 downto 0);
42   signal pgain      : std_logic_vector(log2ceil(int_bits)-1 downto 0);
43   signal igain      : std_logic_vector(log2ceil(int_bits)-1 downto 0);
```

```
44    signal pgain2     : std_logic_vector(log2ceil(int_bits)-1 downto 0);
45    signal igain2     : std_logic_vector(log2ceil(int_bits)-1 downto 0);
46    signal start_freq : std_logic_vector(freq_bits-1 downto 0);
47    signal freq_out   : std_logic_vector(freq_bits-1 downto 0);
48
49    file log : text open write_mode is "log";
50
51  begin  -- architecture bhav
52
53    clk <= not clk after 6.25 ns;
54    t <= t + 1 after 12.5 ns;
55    reset <= '0' when t < 10 else '1';
56
57    logger: process (clk, reset) is
58      variable l : line;
59    begin
60      if rising_edge(clk) then
61        write(l, t);
62        write(l, " ");
63        write(l, real(to_integer(unsigned(freq)))/real(2**freq_bits-1));
64        write(l, " ");
65        write(l, real(to_integer(unsigned(freq_out)))/real(2**freq_bits-1));
66        write(l, " ");
67        write(l, real(to_integer(unsigned(amp)))/real(2**bits-1));
68        write(l, " ");
69        write(l, real(to_integer(signed(i_slow)))/real(2**bits-1));
70        write(l, " ");
71        write(l, to_integer(unsigned(pgain2)));
72        writeline(log, l);
73      end if;
74    end process logger;
75
76    -- modulation
77    freq <= std_logic_vector(to_unsigned(integer((sin(real(t)/real(100000)*real(2)*
          MATH_PI)*real(0.1)+real(0.2))*real(2**freq_bits-1)), freq_bits));
78    amp <= std_logic_vector(to_signed(integer((cos(real(t)/real(1000000)*real(2)*
          MATH_PI)*real(0.49)+real(0.5))*real(2**(bits-1)-1)), bits));
79
80    start_freq <= std_logic_vector(to_unsigned(integer(real(0.2)*real(2**freq_bits
          -1)), freq_bits));
81    pgain <= std_logic_vector(to_signed(-6, log2ceil(int_bits)));
82    igain <= std_logic_vector(to_signed(-8, log2ceil(int_bits)));
83
84    nco_1: entity work.nco
85      generic map (
86        freq_bits       => freq_bits,
87        lut_bits        => lut_bits,
88        bits            => bits,
89        use_registers   => use_registers,
90        use_kogge_stone => use_kogge_stone)
91      port map (
92        clk   => clk,
93        reset => reset,
94        freq  => freq,
95        pm    => (others => '0'),
96        sin   => sin1,
97        cos   => open,
98        saw   => open);
99
100   mul_1: entity work.mul
101     generic map (
102       bits1           => bits,
```

```
103       bits2           => bits,
104       signed_arith    => '1',
105       use_registers   => '0',
106       use_kogge_stone => '0')
107     port map (
108       clk    => clk,
109       reset  => reset,
110       input1 => sin1,
111       input2 => amp,
112       output => sin2);
113
114   sin3 <= sin2(2*bits-2 downto bits-1);
115
116   pll2_1: entity work.pll2
117     generic map (
118       bits            => bits,
119       int_bits        => int_bits,
120       lut_bits        => lut_bits,
121       nco_bits        => nco_bits,
122       freq_bits       => freq_bits,
123       signed_arith    => signed_arith,
124       use_registers   => use_registers,
125       use_kogge_stone => use_kogge_stone)
126     port map (
127       clk        => clk,
128       reset      => reset,
129       input      => sin3,
130       i          => i,
131       q          => q,
132       error      => q,
133       pgain      => pgain2,
134       igain      => igain2,
135       start_freq => start_freq,
136       freq_out   => freq_out,
137       freq_in    => freq_out,
138       phase      => open);
139
140   clkdiv_1: entity work.clkdiv
141     generic map (
142       div             => 2**r,
143       duty_cycle      => '1',
144       use_kogge_stone => '0')
145     port map (
146       clk     => clk,
147       reset   => reset,
148       enable  => '1',
149       clk_out => clk2);
150
151   gcic_1: entity work.gcic
152     generic map (
153       bits            => bits+nco_bits,
154       out_bits        => bits+nco_bits,
155       r               => r,
156       n               => n,
157       signed_arith    => '0',
158       use_kogge_stone => '0')
159     port map (
160       clk    => clk,
161       clk2   => clk2,
162       reset  => reset,
163       input  => i,
164       output => i_slow);
```

```
165
166   absolute_1: entity work.absolute
167     generic map (
168       bits            => bits+nco_bits,
169       use_registers   => '0',
170       use_kogge_stone => '0')
171     port map (
172       clk    => clk,
173       reset  => reset,
174       input  => i_slow,
175       output => i_abs);
176
177   agc_1: entity work.agc
178     generic map (
179       bits     => bits+nco_bits,
180       gainbits => log2ceil(int_bits))
181     port map (
182       clk       => clk2,
183       reset     => reset,
184       amplitude => i_abs,
185       pgain_in  => pgain,
186       pgain_out => pgain2,
187       igain_in  => igain,
188       igain_out => igain2);
189
190 end architecture behav;
```

B.3 Differential Wavefront Sensing

B.3.1 Implementation

```
1  -- Copyright (c) 2016, Nils Christopher Brause
2  -- All rights reserved.
3  --
4  -- Permission to use, copy, modify, and/or distribute this software for any
5  -- purpose with or without fee is hereby granted, provided that the above
6  -- copyright notice and this permission notice appear in all copies.
7  --
8  -- THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
9  -- WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
10 -- MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
11 -- ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
12 -- WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
13 -- ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
14 -- OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
15 --
16 -- The views and conclusions contained in the software and documentation are
17 -- those of the authors and should not be interpreted as representing official
18 -- policies, either expressed or implied, of the Max Planck Institute for
19 -- Gravitational Physics (Albert Einstein Institute).
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use work.log2.all;
24
25 --! phase locked loop for QPDs
26
```

```
27  entity qpd_pll is
28    generic (
29      bits            : natural;          --! width of input
30      int_bits        : natural;          --! internal signal width
31      lut_bits        : natural;          --! width of LUT input
32      nco_bits        : natural;          --! width of nco output
33      freq_bits       : natural;          --! width of frequency input/output
34      signed_arith    : bit := '1';       --! assume input is signed
35      use_registers   : bit := '0';       --! use additional registers on slow
            FPGAs
36      use_kogge_stone : bit := '0');      --! use an optimized Kogge Stone adder
37    port (
38      clk        : in  std_logic;         --! clock input
39      reset      : in  std_logic;         --! asynchronous reset (active low)
40      enable     : in  std_logic;         --! enable component
41      inputa     : in  std_logic_vector(bits-1 downto 0);  --! input signal
42      inputb     : in  std_logic_vector(bits-1 downto 0);  --! input signal
43      inputc     : in  std_logic_vector(bits-1 downto 0);  --! input signal
44      inputd     : in  std_logic_vector(bits-1 downto 0);  --! input signal
45      ia         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! intensity
            output
46      qa         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! quality
            output
47      ib         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! intensity
            output
48      qb         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! quality
            output
49      ic         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! intensity
            output
50      qc         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! quality
            output
51      id         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! intensity
            output
52      qd         : out std_logic_vector(bits+nco_bits-1 downto 0);  --! quality
            output
53      errora     : in  std_logic_vector(bits+nco_bits-1 downto 0);  --! error input
             (connect to q)
54      errorb     : in  std_logic_vector(bits+nco_bits-1 downto 0);  --! error input
             (connect to q)
55      errorc     : in  std_logic_vector(bits+nco_bits-1 downto 0);  --! error input
             (connect to q)
56      errord     : in  std_logic_vector(bits+nco_bits-1 downto 0);  --! error input
             (connect to q)
57      pgain_sum  : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            proportional gain
58      igain_sum  : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            integral gain
59      pgain_dx   : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            proportional gain
60      igain_dx   : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            integral gain
61      pgain_dy   : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            proportional gain
62      igain_dy   : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            integral gain
63      pgain_ell  : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            proportional gain
64      igain_ell  : in  std_logic_vector(log2ceil(int_bits)-1 downto 0);  --!
            integral gain
65      start_freq : in  std_logic_vector(freq_bits-1 downto 0);  --! start frequency
66      freq_out   : out std_logic_vector(freq_bits-1 downto 0);  --! measured
            frequency
```

```
67      freq_in    : in  std_logic_vector(freq_bits-1 downto 0);  --! frequency input
            (connect to freq_in)
68      phase_sum  : out std_logic_vector(freq_bits-1 downto 0);  --! phase output
69      phase_dx   : out std_logic_vector(freq_bits-1 downto 0);  --! phase output
70      phase_dy   : out std_logic_vector(freq_bits-1 downto 0);  --! phase output
71      phase_ell  : out std_logic_vector(freq_bits-1 downto 0));  --! phase output
72  end entity qpd_pll;
73
74  architecture behav of qpd_pll is
75
76    signal sinea : std_logic_vector(nco_bits-1 downto 0);
77    signal sineb : std_logic_vector(nco_bits-1 downto 0);
78    signal sinec : std_logic_vector(nco_bits-1 downto 0);
79    signal sined : std_logic_vector(nco_bits-1 downto 0);
80    signal cosinea : std_logic_vector(nco_bits-1 downto 0);
81    signal cosineb : std_logic_vector(nco_bits-1 downto 0);
82    signal cosinec : std_logic_vector(nco_bits-1 downto 0);
83    signal cosined : std_logic_vector(nco_bits-1 downto 0);
84
85    signal errora2 : std_logic_vector(bits+nco_bits-1 downto 0);
86    signal errorb2 : std_logic_vector(bits+nco_bits-1 downto 0);
87    signal errorc2 : std_logic_vector(bits+nco_bits-1 downto 0);
88    signal errord2 : std_logic_vector(bits+nco_bits-1 downto 0);
89    signal tmp1 : std_logic_vector(bits+nco_bits-1 downto 0);
90    signal tmp2 : std_logic_vector(bits+nco_bits-1 downto 0);
91    signal tmp3 : std_logic_vector(bits+nco_bits-1 downto 0);
92    signal tmp4 : std_logic_vector(bits+nco_bits-1 downto 0);
93    signal tmp5 : std_logic_vector(bits+nco_bits-1 downto 0);
94    signal tmp6 : std_logic_vector(bits+nco_bits-1 downto 0);
95    signal tmp7 : std_logic_vector(bits+nco_bits-1 downto 0);
96    signal tmp8 : std_logic_vector(bits+nco_bits-1 downto 0);
97    signal error_sum : std_logic_vector(bits+nco_bits-1 downto 0);
98    signal error_dx : std_logic_vector(bits+nco_bits-1 downto 0);
99    signal error_dy : std_logic_vector(bits+nco_bits-1 downto 0);
100   signal error_ell : std_logic_vector(bits+nco_bits-1 downto 0);
101
102   signal pidout_sum : std_logic_vector(int_bits-1 downto 0);
103   signal pidout_dx : std_logic_vector(int_bits-1 downto 0);
104   signal pidout_dy : std_logic_vector(int_bits-1 downto 0);
105   signal pidout_ell : std_logic_vector(int_bits-1 downto 0);
106   signal pidout_sum2 : std_logic_vector(freq_bits-1 downto 0);
107   signal pidout_dx2 : std_logic_vector(freq_bits-1 downto 0);
108   signal pidout_dy2 : std_logic_vector(freq_bits-1 downto 0);
109   signal pidout_ell2 : std_logic_vector(freq_bits-1 downto 0);
110
111   signal phase_sum_tmp : std_logic_vector(freq_bits-1 downto 0);
112   signal phase_dx_tmp : std_logic_vector(freq_bits-1 downto 0);
113   signal phase_dy_tmp : std_logic_vector(freq_bits-1 downto 0);
114   signal phase_ell_tmp : std_logic_vector(freq_bits-1 downto 0);
115   signal tmp11 : std_logic_vector(freq_bits-1 downto 0);
116   signal tmp12 : std_logic_vector(freq_bits-1 downto 0);
117   signal tmp13 : std_logic_vector(freq_bits-1 downto 0);
118   signal tmp14 : std_logic_vector(freq_bits-1 downto 0);
119   signal tmp15 : std_logic_vector(freq_bits-1 downto 0);
120   signal tmp16 : std_logic_vector(freq_bits-1 downto 0);
121   signal tmp17 : std_logic_vector(freq_bits-1 downto 0);
122   signal tmp18 : std_logic_vector(freq_bits-1 downto 0);
123   signal phasea : std_logic_vector(freq_bits-1 downto 0);
124   signal phaseb : std_logic_vector(freq_bits-1 downto 0);
125   signal phasec : std_logic_vector(freq_bits-1 downto 0);
126   signal phased : std_logic_vector(freq_bits-1 downto 0);
127
```

```
128 begin  -- architecture behav
129
130   -- IQ demodulation
131
132   mul_ia: entity work.mul
133     generic map (
134       bits1           => bits,
135       bits2           => nco_bits,
136       signed_arith    => signed_arith,
137       use_registers   => use_registers,
138       use_kogge_stone => use_kogge_stone)
139     port map (
140       clk    => clk,
141       reset  => reset,
142       input1 => inputa,
143       input2 => sinea,
144       output => ia);
145
146   mul_ib: entity work.mul
147     generic map (
148       bits1           => bits,
149       bits2           => nco_bits,
150       signed_arith    => signed_arith,
151       use_registers   => use_registers,
152       use_kogge_stone => use_kogge_stone)
153     port map (
154       clk    => clk,
155       reset  => reset,
156       input1 => inputb,
157       input2 => sineb,
158       output => ib);
159
160   mul_ic: entity work.mul
161     generic map (
162       bits1           => bits,
163       bits2           => nco_bits,
164       signed_arith    => signed_arith,
165       use_registers   => use_registers,
166       use_kogge_stone => use_kogge_stone)
167     port map (
168       clk    => clk,
169       reset  => reset,
170       input1 => inputc,
171       input2 => sinec,
172       output => ic);
173
174   mul_id: entity work.mul
175     generic map (
176       bits1           => bits,
177       bits2           => nco_bits,
178       signed_arith    => signed_arith,
179       use_registers   => use_registers,
180       use_kogge_stone => use_kogge_stone)
181     port map (
182       clk    => clk,
183       reset  => reset,
184       input1 => inputd,
185       input2 => sined,
186       output => id);
187
188   mul_qa: entity work.mul
189     generic map (
```

```
190       bits1           => bits,
191       bits2           => nco_bits,
192       signed_arith    => signed_arith,
193       use_registers   => use_registers,
194       use_kogge_stone => use_kogge_stone)
195     port map (
196       clk    => clk,
197       reset  => reset,
198       input1 => inputa,
199       input2 => cosinea,
200       output => qa);
201
202   mul_qb: entity work.mul
203     generic map (
204       bits1           => bits,
205       bits2           => nco_bits,
206       signed_arith    => signed_arith,
207       use_registers   => use_registers,
208       use_kogge_stone => use_kogge_stone)
209     port map (
210       clk    => clk,
211       reset  => reset,
212       input1 => inputb,
213       input2 => cosineb,
214       output => qb);
215
216   mul_qc: entity work.mul
217     generic map (
218       bits1           => bits,
219       bits2           => nco_bits,
220       signed_arith    => signed_arith,
221       use_registers   => use_registers,
222       use_kogge_stone => use_kogge_stone)
223     port map (
224       clk    => clk,
225       reset  => reset,
226       input1 => inputc,
227       input2 => cosinec,
228       output => qc);
229
230   mul_qd: entity work.mul
231     generic map (
232       bits1           => bits,
233       bits2           => nco_bits,
234       signed_arith    => signed_arith,
235       use_registers   => use_registers,
236       use_kogge_stone => use_kogge_stone)
237     port map (
238       clk    => clk,
239       reset  => reset,
240       input1 => inputd,
241       input2 => cosined,
242       output => qd);
243
244   -- divide error signals by 4 before adding to prevent overflow
245
246   barrel_shift_int_a: entity work.barrel_shift_int
247     generic map (
248       bits         => bits+nco_bits,
249       value        => 2,
250       signed_arith => signed_arith,
251       direction    => '0')
```

```
252     port map (
253       input  => errora,
254       output => errora2);
255
256   barrel_shift_int_b: entity work.barrel_shift_int
257     generic map (
258       bits         => bits+nco_bits,
259       value        => 2,
260       signed_arith => signed_arith,
261       direction    => '0')
262     port map (
263       input  => errorb,
264       output => errorb2);
265
266   barrel_shift_int_c: entity work.barrel_shift_int
267     generic map (
268       bits         => bits+nco_bits,
269       value        => 2,
270       signed_arith => signed_arith,
271       direction    => '0')
272     port map (
273       input  => errorc,
274       output => errorc2);
275
276   barrel_shift_int_d: entity work.barrel_shift_int
277     generic map (
278       bits         => bits+nco_bits,
279       value        => 2,
280       signed_arith => signed_arith,
281       direction    => '0')
282     port map (
283       input  => errord,
284       output => errord2);
285
286   -- combine error signals
287
288   add_sum1: entity work.add
289     generic map (
290       bits            => bits+nco_bits,
291       use_registers   => use_registers,
292       use_kogge_stone => use_kogge_stone)
293     port map (
294       clk       => clk,
295       reset     => reset,
296       input1    => errora2,
297       input2    => errorb2,
298       output    => tmp1,
299       carry_in  => '0',
300       carry_out => open,
301       overflow  => open);
302
303   add_sum2: entity work.add
304     generic map (
305       bits            => bits+nco_bits,
306       use_registers   => use_registers,
307       use_kogge_stone => use_kogge_stone)
308     port map (
309       clk       => clk,
310       reset     => reset,
311       input1    => errorc2,
312       input2    => errord2,
313       output    => tmp2,
```

```
314       carry_in  => '0',
315       carry_out => open,
316       overflow  => open);
317
318   add_sum3: entity work.add
319     generic map (
320       bits            => bits+nco_bits,
321       use_registers   => use_registers,
322       use_kogge_stone => use_kogge_stone)
323     port map (
324       clk       => clk,
325       reset     => reset,
326       input1    => tmp1,
327       input2    => tmp2,
328       output    => error_sum,
329       carry_in  => '0',
330       carry_out => open,
331       overflow  => open);
332
333   sub_dx1: entity work.sub
334     generic map (
335       bits            => bits+nco_bits,
336       use_registers   => use_registers,
337       use_kogge_stone => use_kogge_stone)
338     port map (
339       clk        => clk,
340       reset      => reset,
341       input1     => errora2,
342       input2     => errorb2,
343       output     => tmp3,
344       borrow_in  => '0',
345       borrow_out => open,
346       underflow  => open);
347
348   sub_dx2: entity work.sub
349     generic map (
350       bits            => bits+nco_bits,
351       use_registers   => use_registers,
352       use_kogge_stone => use_kogge_stone)
353     port map (
354       clk        => clk,
355       reset      => reset,
356       input1     => errorc2,
357       input2     => errord2,
358       output     => tmp4,
359       borrow_in  => '0',
360       borrow_out => open,
361       underflow  => open);
362
363   add_dx3: entity work.add
364     generic map (
365       bits            => bits+nco_bits,
366       use_registers   => use_registers,
367       use_kogge_stone => use_kogge_stone)
368     port map (
369       clk       => clk,
370       reset     => reset,
371       input1    => tmp3,
372       input2    => tmp4,
373       output    => error_dx,
374       carry_in  => '0',
375       carry_out => open,
```

```
376       overflow  => open);
377
378   sub_dy1: entity work.sub
379     generic map (
380       bits            => bits+nco_bits,
381       use_registers   => use_registers,
382       use_kogge_stone => use_kogge_stone)
383     port map (
384       clk        => clk,
385       reset      => reset,
386       input1     => errora2,
387       input2     => errorc2,
388       output     => tmp5,
389       borrow_in  => '0',
390       borrow_out => open,
391       underflow  => open);
392
393   sub_dy2: entity work.sub
394     generic map (
395       bits            => bits+nco_bits,
396       use_registers   => use_registers,
397       use_kogge_stone => use_kogge_stone)
398     port map (
399       clk        => clk,
400       reset      => reset,
401       input1     => errorb2,
402       input2     => errord2,
403       output     => tmp6,
404       borrow_in  => '0',
405       borrow_out => open,
406       underflow  => open);
407
408   add_dy3: entity work.add
409     generic map (
410       bits            => bits+nco_bits,
411       use_registers   => use_registers,
412       use_kogge_stone => use_kogge_stone)
413     port map (
414       clk       => clk,
415       reset     => reset,
416       input1    => tmp5,
417       input2    => tmp6,
418       output    => error_dy,
419       carry_in  => '0',
420       carry_out => open,
421       overflow  => open);
422
423   sub_ell1: entity work.sub
424     generic map (
425       bits            => bits+nco_bits,
426       use_registers   => use_registers,
427       use_kogge_stone => use_kogge_stone)
428     port map (
429       clk        => clk,
430       reset      => reset,
431       input1     => errora2,
432       input2     => errorb2,
433       output     => tmp7,
434       borrow_in  => '0',
435       borrow_out => open,
436       underflow  => open);
437
```

```
438   sub_ell2: entity work.sub
439     generic map (
440       bits            => bits+nco_bits,
441       use_registers   => use_registers,
442       use_kogge_stone => use_kogge_stone)
443     port map (
444       clk        => clk,
445       reset      => reset,
446       input1     => errord2,
447       input2     => errorc2,
448       output     => tmp8,
449       borrow_in  => '0',
450       borrow_out => open,
451       underflow  => open);
452
453   add_ell3: entity work.add
454     generic map (
455       bits            => bits+nco_bits,
456       use_registers   => use_registers,
457       use_kogge_stone => use_kogge_stone)
458     port map (
459       clk       => clk,
460       reset     => reset,
461       input1    => tmp7,
462       input2    => tmp8,
463       output    => error_ell,
464       carry_in  => '0',
465       carry_out => open,
466       overflow  => open);
467
468   -- PID filter
469
470   pidctrl_sum: entity work.pidctrl
471     generic map (
472       bits            => bits+nco_bits,
473       int_bits        => int_bits,
474       signed_arith    => signed_arith,
475       gains_first     => '1',
476       use_prop        => '1',
477       use_int         => '1',
478       use_diff        => '0',
479       use_registers   => use_registers,
480       use_kogge_stone => use_kogge_stone)
481     port map (
482       clk    => clk,
483       reset  => reset,
484       enable => enable,
485       input  => error_sum,
486       pgain  => pgain_sum,
487       igain  => igain_sum,
488       dgain  => (others => '0'),
489       output => pidout_sum);
490
491   pidctrl_dx: entity work.pidctrl
492     generic map (
493       bits            => bits+nco_bits,
494       int_bits        => int_bits,
495       signed_arith    => signed_arith,
496       gains_first     => '1',
497       use_prop        => '1',
498       use_int         => '1',
499       use_diff        => '0',
```

```
500       use_registers   => use_registers,
501       use_kogge_stone => use_kogge_stone)
502     port map (
503       clk    => clk,
504       reset  => reset,
505       enable => enable,
506       input  => error_dx,
507       pgain  => pgain_dx,
508       igain  => igain_dx,
509       dgain  => (others => '0'),
510       output => pidout_dx);
511
512   pidctrl_dy: entity work.pidctrl
513     generic map (
514       bits            => bits+nco_bits,
515       int_bits        => int_bits,
516       signed_arith    => signed_arith,
517       gains_first     => '1',
518       use_prop        => '1',
519       use_int         => '1',
520       use_diff        => '0',
521       use_registers   => use_registers,
522       use_kogge_stone => use_kogge_stone)
523     port map (
524       clk    => clk,
525       reset  => reset,
526       enable => enable,
527       input  => error_dy,
528       pgain  => pgain_dy,
529       igain  => igain_dy,
530       dgain  => (others => '0'),
531       output => pidout_dy);
532
533   pidctrl_ell: entity work.pidctrl
534     generic map (
535       bits            => bits+nco_bits,
536       int_bits        => int_bits,
537       signed_arith    => signed_arith,
538       gains_first     => '1',
539       use_prop        => '1',
540       use_int         => '1',
541       use_diff        => '0',
542       use_registers   => use_registers,
543       use_kogge_stone => use_kogge_stone)
544     port map (
545       clk    => clk,
546       reset  => reset,
547       enable => enable,
548       input  => error_ell,
549       pgain  => pgain_ell,
550       igain  => igain_ell,
551       dgain  => (others => '0'),
552       output => pidout_ell);
553
554   -- resize pid results
555
556   round_sum: entity work.round
557     generic map (
558       inp_bits        => int_bits,
559       outp_bits       => freq_bits,
560       signed_arith    => signed_arith,
561       use_registers   => use_registers,
```

```
562       use_kogge_stone => use_kogge_stone)
563     port map (
564       clk    => clk,
565       reset  => reset,
566       input  => pidout_sum,
567       output => pidout_sum2);
568
569   round_dx: entity work.round
570     generic map (
571       inp_bits        => int_bits,
572       outp_bits       => freq_bits,
573       signed_arith    => signed_arith,
574       use_registers   => use_registers,
575       use_kogge_stone => use_kogge_stone)
576     port map (
577       clk    => clk,
578       reset  => reset,
579       input  => pidout_dx,
580       output => pidout_dx2);
581
582   round_dy: entity work.round
583     generic map (
584       inp_bits        => int_bits,
585       outp_bits       => freq_bits,
586       signed_arith    => signed_arith,
587       use_registers   => use_registers,
588       use_kogge_stone => use_kogge_stone)
589     port map (
590       clk    => clk,
591       reset  => reset,
592       input  => pidout_dy,
593       output => pidout_dy2);
594
595   round_ell: entity work.round
596     generic map (
597       inp_bits        => int_bits,
598       outp_bits       => freq_bits,
599       signed_arith    => signed_arith,
600       use_registers   => use_registers,
601       use_kogge_stone => use_kogge_stone)
602     port map (
603       clk    => clk,
604       reset  => reset,
605       input  => pidout_ell,
606       output => pidout_ell2);
607
608   -- add start frequency
609
610   add_freq: entity work.add
611     generic map (
612       bits            => freq_bits,
613       use_registers   => use_registers,
614       use_kogge_stone => use_kogge_stone)
615     port map (
616       clk       => clk,
617       reset     => reset,
618       input1    => pidout_sum2,
619       input2    => start_freq,
620       output    => freq_out,
621       carry_in  => '0',
622       carry_out => open,
623       overflow  => open);
```

```
624
625   -- integrate frequency to phase
626
627   accumulator_sum: entity work.accumulator
628     generic map (
629       bits            => freq_bits,
630       use_kogge_stone => use_kogge_stone)
631     port map (
632       clk    => clk,
633       reset  => reset,
634       enable => enable,
635       input  => freq_in,
636       output => phase_sum_tmp);
637   phase_sum <= phase_sum_tmp;
638
639   reg_dx: entity work.reg
640     generic map (
641       bits => freq_bits)
642     port map (
643       clk      => clk,
644       reset    => reset,
645       enable   => enable,
646       data_in  => pidout_dx2,
647       data_out => phase_dx_tmp);
648   phase_dx <= phase_dx_tmp;
649
650   reg_dy: entity work.reg
651     generic map (
652       bits => freq_bits)
653     port map (
654       clk      => clk,
655       reset    => reset,
656       enable   => enable,
657       data_in  => pidout_dy2,
658       data_out => phase_dy_tmp);
659   phase_dy <= phase_dy_tmp;
660
661   reg_ell: entity work.reg
662     generic map (
663       bits => freq_bits)
664     port map (
665       clk      => clk,
666       reset    => reset,
667       enable   => enable,
668       data_in  => pidout_ell2,
669       data_out => phase_ell_tmp);
670   phase_ell <= phase_ell_tmp;
671
672   -- combine phases
673
674   add_a1: entity work.add
675     generic map (
676       bits            => freq_bits,
677       use_registers   => use_registers,
678       use_kogge_stone => use_kogge_stone)
679     port map (
680       clk       => clk,
681       reset     => reset,
682       input1    => phase_sum_tmp,
683       input2    => phase_dx_tmp,
684       output    => tmp11,
685       carry_in  => '0',
```

```
686       carry_out => open,
687       overflow  => open);
688
689   add_a2: entity work.add
690     generic map (
691       bits            => freq_bits,
692       use_registers   => use_registers,
693       use_kogge_stone => use_kogge_stone)
694     port map (
695       clk       => clk,
696       reset     => reset,
697       input1    => phase_dy_tmp,
698       input2    => phase_ell_tmp,
699       output    => tmp12,
700       carry_in  => '0',
701       carry_out => open,
702       overflow  => open);
703
704   add_a3: entity work.add
705     generic map (
706       bits            => freq_bits,
707       use_registers   => use_registers,
708       use_kogge_stone => use_kogge_stone)
709     port map (
710       clk       => clk,
711       reset     => reset,
712       input1    => tmp11,
713       input2    => tmp12,
714       output    => phasea,
715       carry_in  => '0',
716       carry_out => open,
717       overflow  => open);
718
719   sub_b1: entity work.sub
720     generic map (
721       bits            => freq_bits,
722       use_registers   => use_registers,
723       use_kogge_stone => use_kogge_stone)
724     port map (
725       clk        => clk,
726       reset      => reset,
727       input1     => phase_sum_tmp,
728       input2     => phase_dx_tmp,
729       output     => tmp13,
730       borrow_in  => '0',
731       borrow_out => open,
732       underflow  => open);
733
734   sub_b2: entity work.sub
735     generic map (
736       bits            => freq_bits,
737       use_registers   => use_registers,
738       use_kogge_stone => use_kogge_stone)
739     port map (
740       clk        => clk,
741       reset      => reset,
742       input1     => phase_dy_tmp,
743       input2     => phase_ell_tmp,
744       output     => tmp14,
745       borrow_in  => '0',
746       borrow_out => open,
747       underflow  => open);
```

```
748
749   add_b3: entity work.add
750     generic map (
751       bits            => freq_bits,
752       use_registers   => use_registers,
753       use_kogge_stone => use_kogge_stone)
754     port map (
755       clk       => clk,
756       reset     => reset,
757       input1    => tmp13,
758       input2    => tmp14,
759       output    => phaseb,
760       carry_in  => '0',
761       carry_out => open,
762       overflow  => open);
763
764   sub_c1: entity work.sub
765     generic map (
766       bits            => freq_bits,
767       use_registers   => use_registers,
768       use_kogge_stone => use_kogge_stone)
769     port map (
770       clk        => clk,
771       reset      => reset,
772       input1     => phase_sum_tmp,
773       input2     => phase_dy_tmp,
774       output     => tmp15,
775       borrow_in  => '0',
776       borrow_out => open,
777       underflow  => open);
778
779   sub_c2: entity work.sub
780     generic map (
781       bits            => freq_bits,
782       use_registers   => use_registers,
783       use_kogge_stone => use_kogge_stone)
784     port map (
785       clk        => clk,
786       reset      => reset,
787       input1     => phase_dx_tmp,
788       input2     => phase_ell_tmp,
789       output     => tmp16,
790       borrow_in  => '0',
791       borrow_out => open,
792       underflow  => open);
793
794   add_c3: entity work.add
795     generic map (
796       bits            => freq_bits,
797       use_registers   => use_registers,
798       use_kogge_stone => use_kogge_stone)
799     port map (
800       clk       => clk,
801       reset     => reset,
802       input1    => tmp15,
803       input2    => tmp16,
804       output    => phasec,
805       carry_in  => '0',
806       carry_out => open,
807       overflow  => open);
808
809   sub_d1: entity work.sub
```

```
810     generic map (
811       bits            => freq_bits,
812       use_registers   => use_registers,
813       use_kogge_stone => use_kogge_stone)
814     port map (
815       clk        => clk,
816       reset      => reset,
817       input1     => phase_sum_tmp,
818       input2     => phase_dx_tmp,
819       output     => tmp17,
820       borrow_in  => '0',
821       borrow_out => open,
822       underflow  => open);
823
824   sub_d2: entity work.sub
825     generic map (
826       bits            => freq_bits,
827       use_registers   => use_registers,
828       use_kogge_stone => use_kogge_stone)
829     port map (
830       clk        => clk,
831       reset      => reset,
832       input1     => phase_ell_tmp,
833       input2     => phase_dy_tmp,
834       output     => tmp18,
835       borrow_in  => '0',
836       borrow_out => open,
837       underflow  => open);
838
839   add_d3: entity work.add
840     generic map (
841       bits            => freq_bits,
842       use_registers   => use_registers,
843       use_kogge_stone => use_kogge_stone)
844     port map (
845       clk       => clk,
846       reset     => reset,
847       input1    => tmp17,
848       input2    => tmp18,
849       output    => phased,
850       carry_in  => '0',
851       carry_out => open,
852       overflow  => open);
853
854   -- look up tables
855
856   sincos_a: entity work.sincos
857     generic map (
858       phase_bits    => freq_bits,
859       bits          => nco_bits,
860       use_registers => use_registers,
861       lut_type      => 1)
862     port map (
863       clk    => clk,
864       reset  => reset,
865       phase  => phasea,
866       sinout => sinea,
867       cosout => cosinea);
868
869   sincos_b: entity work.sincos
870     generic map (
871       phase_bits    => freq_bits,
```

```
872       bits          => nco_bits,
873       use_registers => use_registers,
874       lut_type      => 1)
875     port map (
876       clk    => clk,
877       reset  => reset,
878       phase  => phaseb,
879       sinout => sineb,
880       cosout => cosineb);
881
882   sincos_c: entity work.sincos
883     generic map (
884       phase_bits    => freq_bits,
885       bits          => nco_bits,
886       use_registers => use_registers,
887       lut_type      => 1)
888     port map (
889       clk    => clk,
890       reset  => reset,
891       phase  => phasec,
892       sinout => sinec,
893       cosout => cosinec);
894
895   sincos_d: entity work.sincos
896     generic map (
897       phase_bits    => freq_bits,
898       bits          => nco_bits,
899       use_registers => use_registers,
900       lut_type      => 1)
901     port map (
902       clk    => clk,
903       reset  => reset,
904       phase  => phased,
905       sinout => sined,
906       cosout => cosined);
907
908 end architecture behav;
```

B.3.2 Testbench

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  use ieee.math_real.all;
5  use std.textio.all;
6  use work.log2.all;
7
8  entity testbench is
9
10 end entity testbench;
11
12 architecture behav of testbench is
13
14   constant bits      : natural := 14;
15   constant nco_bits  : natural := bits;
16   constant lut_bits  : natural := bits;
17   constant int_bits  : natural := 3*bits;
18   constant freq_bits : natural := 16;
19
```

```
20    constant signed_arith    : bit := '1';
21    constant use_registers   : bit := '0';
22    constant use_kogge_stone : bit := '0';
23
24    signal clk   : std_logic := '0';
25    signal reset : std_logic;
26    signal t     : natural := 0;
27
28    signal freq  : std_logic_vector(freq_bits-1 downto 0);
29    signal pm    : std_logic_vector(freq_bits-1 downto 0);
30    signal sin1  : std_logic_vector(bits-1 downto 0);
31    signal sin2  : std_logic_vector(bits-1 downto 0);
32
33    signal inputa     : std_logic_vector(bits-1 downto 0);
34    signal inputb     : std_logic_vector(bits-1 downto 0);
35    signal inputc     : std_logic_vector(bits-1 downto 0);
36    signal inputd     : std_logic_vector(bits-1 downto 0);
37    signal ia         : std_logic_vector(bits+nco_bits-1 downto 0);
38    signal qa         : std_logic_vector(bits+nco_bits-1 downto 0);
39    signal ib         : std_logic_vector(bits+nco_bits-1 downto 0);
40    signal qb         : std_logic_vector(bits+nco_bits-1 downto 0);
41    signal ic         : std_logic_vector(bits+nco_bits-1 downto 0);
42    signal qc         : std_logic_vector(bits+nco_bits-1 downto 0);
43    signal id         : std_logic_vector(bits+nco_bits-1 downto 0);
44    signal qd         : std_logic_vector(bits+nco_bits-1 downto 0);
45    signal errora     : std_logic_vector(bits+nco_bits-1 downto 0);
46    signal errorb     : std_logic_vector(bits+nco_bits-1 downto 0);
47    signal errorc     : std_logic_vector(bits+nco_bits-1 downto 0);
48    signal errord     : std_logic_vector(bits+nco_bits-1 downto 0);
49    signal pgain_sum  : std_logic_vector(log2ceil(int_bits)-1 downto 0);
50    signal igain_sum  : std_logic_vector(log2ceil(int_bits)-1 downto 0);
51    signal pgain_dx   : std_logic_vector(log2ceil(int_bits)-1 downto 0);
52    signal igain_dx   : std_logic_vector(log2ceil(int_bits)-1 downto 0);
53    signal pgain_dy   : std_logic_vector(log2ceil(int_bits)-1 downto 0);
54    signal igain_dy   : std_logic_vector(log2ceil(int_bits)-1 downto 0);
55    signal pgain_ell  : std_logic_vector(log2ceil(int_bits)-1 downto 0);
56    signal igain_ell  : std_logic_vector(log2ceil(int_bits)-1 downto 0);
57    signal start_freq : std_logic_vector(freq_bits-1 downto 0);
58    signal freq_out   : std_logic_vector(freq_bits-1 downto 0);
59    signal freq_in    : std_logic_vector(freq_bits-1 downto 0);
60    signal phase_sum  : std_logic_vector(freq_bits-1 downto 0);
61    signal phase_dx   : std_logic_vector(freq_bits-1 downto 0);
62    signal phase_dy   : std_logic_vector(freq_bits-1 downto 0);
63    signal phase_ell  : std_logic_vector(freq_bits-1 downto 0);
64
65    file log : text open write_mode is "log";
66
67  begin  -- architecture bhav
68
69    clk <= not clk after 6.25 ns;
70    t <= t + 1 after 12.5 ns;
71    reset <= '0' when t < 10 else '1';
72
73    logger: process (clk, reset) is
74      variable l : line;
75    begin
76      if clk'event and clk = '1' then
77        write(l, t);
78        write(l, " ");
79        write(l, real(to_integer(unsigned(freq)))/real(2**freq_bits-1));
80        write(l, " ");
81        write(l, real(to_integer(unsigned(freq_out)))/real(2**freq_bits-1));
```

```
82        write(l, " ");
83        write(l, real(to_integer(unsigned(pm)))/real(2**freq_bits-1));
84        write(l, " ");
85        write(l, real(-to_integer(signed(phase_dx)))/real(2**freq_bits-1));
86        writeline(log, l);
87      end if;
88    end process logger;
89
90    freq <= std_logic_vector(to_unsigned(integer((sin(real(t)/real(100000)*real(2)*
          MATH_PI)*real(0.1)+real(0.2))*real(2**freq_bits-1)), freq_bits));
91    pm <= std_logic_vector(to_unsigned(integer((sin(real(t)/real(1000000)*real(2)*
          MATH_PI)*real(0.1)+real(0.2))*real(2**freq_bits-1)), freq_bits));
92
93    start_freq <= std_logic_vector(to_unsigned(integer(real(0.2)*real(2**freq_bits
          -1)), freq_bits));
94    pgain_sum <= std_logic_vector(to_signed(-10, log2ceil(int_bits)));
95    igain_sum <= std_logic_vector(to_signed(-12, log2ceil(int_bits)));
96    pgain_dx <= std_logic_vector(to_signed(-12, log2ceil(int_bits)));
97    igain_dx <= std_logic_vector(to_signed(-14, log2ceil(int_bits)));
98    pgain_dy <= std_logic_vector(to_signed(-12, log2ceil(int_bits)));
99    igain_dy <= std_logic_vector(to_signed(-14, log2ceil(int_bits)));
100   pgain_ell <= std_logic_vector(to_signed(-12, log2ceil(int_bits)));
101   igain_ell <= std_logic_vector(to_signed(-14, log2ceil(int_bits)));
102
103   nco_1: entity work.nco
104     generic map (
105       freq_bits       => freq_bits,
106       lut_bits        => lut_bits,
107       bits            => bits,
108       use_registers   => use_registers,
109       use_kogge_stone => use_kogge_stone)
110     port map (
111       clk   => clk,
112       reset => reset,
113       freq  => freq,
114       pm    => (others => '0'),
115       sin   => sin1,
116       cos   => open,
117       saw   => open);
118
119   nco_2: entity work.nco
120     generic map (
121       freq_bits       => freq_bits,
122       lut_bits        => lut_bits,
123       bits            => bits,
124       use_registers   => use_registers,
125       use_kogge_stone => use_kogge_stone)
126     port map (
127       clk   => clk,
128       reset => reset,
129       freq  => freq,
130       pm    => pm,
131       sin   => sin2,
132       cos   => open,
133       saw   => open);
134
135   inputa <= sin1; inputb <= sin2;
136   inputc <= sin1; inputd <= sin2;
137
138   qpd_pll_1: entity work.qpd_pll
139     generic map (
140       bits            => bits,
```

```
      int_bits        => int_bits,
      lut_bits        => lut_bits,
      nco_bits        => nco_bits,
      freq_bits       => freq_bits,
      signed_arith    => signed_arith,
      use_registers   => use_registers,
      use_kogge_stone => use_kogge_stone)
    port map (
      clk        => clk,
      reset      => reset,
      enable     => '1',
      inputa     => inputa,
      inputb     => inputb,
      inputc     => inputc,
      inputd     => inputd,
      ia         => ia,
      qa         => qa,
      ib         => ib,
      qb         => qb,
      ic         => ic,
      qc         => qc,
      id         => id,
      qd         => qd,
      errora     => errora,
      errorb     => errorb,
      errorc     => errorc,
      errord     => errord,
      pgain_sum  => pgain_sum,
      igain_sum  => igain_sum,
      pgain_dx   => pgain_dx,
      igain_dx   => igain_dx,
      pgain_dy   => pgain_dy,
      igain_dy   => igain_dy,
      pgain_ell  => pgain_ell,
      igain_ell  => igain_ell,
      start_freq => start_freq,
      freq_out   => freq_out,
      freq_in    => freq_in,
      phase_sum  => phase_sum,
      phase_dx   => phase_dx,
      phase_dy   => phase_dy,
      phase_ell  => phase_ell);

  errora <= qa;
  errorb <= qb;
  errorc <= qc;
  errord <= qd;
  freq_in <= freq_out;

end architecture behav;
```

B.4 Ranging and data transfer

B.4.1 Actuator signal filter

```
-- Copyright (c) 2012, Nils Christopher Brause
-- All rights reserved.
```

```
3  --
4  -- Permission to use, copy, modify, and/or distribute this software for any
5  -- purpose with or without fee is hereby granted, provided that the above
6  -- copyright notice and this permission notice appear in all copies.
7  --
8  -- THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
9  -- WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
10 -- MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
11 -- ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
12 -- WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
13 -- ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
14 -- OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
15 --
16 -- The views and conclusions contained in the software and documentation are
17 -- those of the authors and should not be interpreted as representing official
18 -- policies, either expressed or implied, of the Max Planck Institute for
19 -- Gravitational Physics (Albert Einstein Institute).
20
21 library ieee;
22 use ieee.std_logic_1164.all;
23 use ieee.numeric_std.all;
24
25 entity slowlyadd is
26   generic (
27     bits : natural;                     --! width of input
28     use_registers   : bit := '0';       --! use additional registers on slow
           FPGAs
29     use_kogge_stone : bit := '0');      --! use an optimized Kogge Stone adder
30   port (
31     clk       : in  std_logic;          --! input clock
32     reset     : in  std_logic;          --! asynchronous reset
33     input1    : in  std_logic_vector(bits-1 downto 0);  --! first summand
34     input2    : in  std_logic_vector(bits-1 downto 0);  --! second summand (slow)
35     output    : out std_logic_vector(bits-1 downto 0);  --! output sum
36     carry_in  : in  std_logic;          --! carry imput (unused)
37     carry_out : out std_logic;          --! carry output
38     overflow  : out std_logic);         --! signed overflow
39 end entity slowlyadd;
40
41 architecture behav of slowlyadd is
42
43   constant one : std_logic_vector(bits-1 downto 0) := std_logic_vector(to_signed
         (1, bits));
44   signal slow : std_logic_vector(bits-1 downto 0);
45   signal slow_next : std_logic_vector(bits-1 downto 0);
46   signal slow_plus : std_logic_vector(bits-1 downto 0);
47   signal slow_minus : std_logic_vector(bits-1 downto 0);
48
49 begin  -- architecture behav
50
51   slow_add_one: entity work.add
52     generic map (
53       bits            => bits,
54       use_registers   => '0',
55       use_kogge_stone => use_kogge_stone)
56     port map (
57       clk       => clk,
58       reset     => reset,
59       input1    => slow,
60       input2    => one,
61       output    => slow_plus,
62       carry_in  => '0',
```

```
63        carry_out => open,
64        overflow  => open);
65
66    slow_sub_one: entity work.sub
67      generic map (
68        bits            => bits,
69        use_registers   => '0',
70        use_kogge_stone => use_kogge_stone)
71      port map (
72        clk        => clk,
73        reset      => reset,
74        input1     => slow,
75        input2     => one,
76        output     => slow_minus,
77        borrow_in  => '0',
78        borrow_out => open,
79        underflow  => open);
80
81    slow_next <= slow_plus when signed(slow) < signed(input2) else
82                 slow_minus when signed(slow) > signed(input2) else
83                 slow;
84
85    slow_reg: entity work.reg
86      generic map (
87        bits => bits)
88      port map (
89        clk      => clk,
90        reset    => reset,
91        enable   => '1',
92        data_in  => slow_next,
93        data_out => slow);
94
95    slow_add: entity work.add
96      generic map (
97        bits            => bits,
98        use_registers   => '1',
99        use_kogge_stone => use_kogge_stone)
100     port map (
101       clk       => clk,
102       reset     => reset,
103       input1    => input1,
104       input2    => slow,
105       output    => output,
106       carry_in  => '0',
107       carry_out => carry_out,
108       overflow  => overflow);
109
110 end architecture behav;
```

Curriculum Vitae

Name: Dipl.-Phys. Nils Christopher Brause
Birthday: 01.10.1985

Schools

1992-1996:	Grundschule Lühnde
1996-2005:	Kooperative Gesamtschule Sehnde (KGS) Degree: Abitur

Internships

06.10.2000-24.11.2000:	Chemische Fabrik Lehrte
13.01.2003-24.01.2003:	Max-Planck-Institut für Atom- und Molekülphysik

University

2005-2011:	Leibniz Universität Hannover Degree: Diplom Physics

Publications

- P. Amaro Seoane, S. Aoudia, H. Audley, G. Auger, S. Babak, J. Baker, E. Barausse, S. Barke, M. Bassan, V. Beckmann, M. Benacquista, P. L. Bender, E. Berti, P. Binétruy, J. Bogenstahl, C. Bonvin, D. Bortoluzzi, N. C. Brause, J. Brossard, S. Buchman, I. Bykov, J. Camp, C. Caprini, A. Cavalleri, M. Cerdonio, G. Ciani, M. Colpi, G. Congedo, J. Conklin, N. Cornish, K. Danzmann, G. de Vine, D. DeBra, M. Dewi Freitag, L. Di Fiore, M. Diaz Aguilo, I. Diepholz, R. Dolesi, M. Dotti, G. Fernández Barranco, L. Ferraioli, V. Ferroni, N. Finetti, E. Fitzsimons, J. Gair, F. Galeazzi, A. Garcia, O. Gerberding, L.

Gesa, D. Giardini, F. Gibert, C. Grimani, P. Groot, F. Guzman Cervantes, Z. Haiman, H. Halloin, G. Heinzel, M. Hewitson, C. Hogan, D. Holz, A. Hornstrup, D. Hoyland, C.D. Hoyle, M. Hueller, S. Hughes, P. Jetzer, V. Kalogera, N. Karnesis, M. Kilic, C. Killow, W. Klipstein, E. Kochkina, N. Korsakova, A. Krolak, S. Larson, M. Lieser, T. Littenberg, J. Livas, I. Lloro, D. Mance, P. Madau, P. Maghami, C. Mahrdt, T. Marsh, I. Mateos, L. Mayer, D. McClelland, K. McKenzie, S. McWilliams, S. Merkowitz, C. Miller, S. Mitryk, J. Moerschell, S. Mohanty, A. Monsky, G. Mueller, V. Müller, G. Nelemans, D. Nicolodi, S. Nissanke, M. Nofrarias, K. Numata, F. Ohme, M. Otto, M. Perreur-Lloyd, A. Petiteau, E. S. Phinney, E. Plagnol, S. Pollack, E. Porter, P. Prat, A. Preston, T. Prince, J. Reiche, D. Richstone, D. Robertson, E. M. Rossi, S. Rosswog, L. Rubbo, A. Ruiter, J. Sanjuan, B.S. Sathyaprakash, S. Schlamminger, B. Schutz, D. Schütze, A. Sesana, D. Shaddock, S. Shah, B. Sheard, C. F. Sopuerta, A. Spector, R. Spero, R. Stanga, R. Stebbins, G. Stede, F. Steier, T. Sumner, K.-X. Sun, A. Sutton, T. Tanaka, D. Tanner, I. Thorpe, M. Tröbs, M. Tinto, H.-B. Tu, M. Vallisneri, D. Vetrugno, S. Vitale, M. Volonteri, V. Wand, Y. Wang, G. Wanner, H. Ward, B. Ware, P. Wass, W. J. Weber, Y. Yu, N. Yunes, and P. Zweifel: "The Gravitational Universe". In: arXiv:1305.5720 [astro-ph.CO] (2013).

- Simon Barke, Nils Brause, Iouri Bykov, Juan Jose Esteban Delgado, Anders Enggaard, Oliver Gerberding, Gerhard Heinzel, Joachim Kullmann, Søren Møller Pedersen and Torben Rasmussen: "LISA Metrology System: Final Report". In: `http://hdl.handle.net/11858/00-001M-0000-0023-E266-6` (2014)

- Oliver Gerberding, Christian Diekmann, Joachim Kullmann, Michael Tröbs, Ioury Bykov, Simon Barke, Nils Christopher Brause, Juan José Esteban Delgado, Thomas S. Schwarze, Jens Reiche, Karsten Danzmann, Torben Rasmussen, Torben Vendt Hansen, Anders Enggaard, Søren Møller Pedersen, Oliver Jennrich, Martin Suess, Zoran Sodnik, and Gerhard Heinzel: "Readout for intersatellite laser interferometry: Measuring low frequency phase fluctuations of high-frequency signals with microradian precision". In: Review of scientific instruments 86 (2015), p. 074501.

Acronyms

AC Alternating Current. 15

ADC Analogue to Digital Converter. 3, 6–8, 11, 15–17, 21, 35, 79, 88, 89, 107

AGC Automatic Gain Control. 4, 9, 15, 55, 58, 60–69, 107, III, V

AM Amplitude Modulation. 55

AOM Acoustic-Optic Modulator. 33

BER Bit Error Rate. 99, 101, 104, 105, 108

CIC Cascaded Integrator Comb. 58

CNR Carrier to Noise Density Ratio. 72, 73, 90, 105, 108, III, V

DAC Digital to Analog Converter. 3, 6–8, 35

DC Direct Current. 17, 53

DFT Discrete Fourier Transform. 12, 15

DLL Delay Locked Loop. 4, 91–98, 101, 102, 104, 105, 108, III, V

DPLL Digital Phase Locked Loop. 3, 4, 8–12, 23–26, 30, 31, 35, 39, 44, 47, 55–69, 72–92, 96, 102–104, 107, 108, III, V

DPS Differential Power Sensing. 71

DSP Digital Signal Processing. 8

DSS Digital Signal Simulator. 101–103

DWS Differential Wavefront Sensing. 4, 71–90, 108, III, V

EBB Elegant Bread Board. 5–8

FEC Forward Error Correction. 104, 105, 108

FFT Fast Fourier Transform. 3, 4, 6–8, 11–22, 30, 31, 42, 44, 46–48, 55–57, 64, 107, III, V

FPGA Field Programmable Gate Array. 3, 8, 12, 16–19, 21, 72, 73

FPU Floating Point Unit. 7

FSM Finite State Machine. 21, 42, 45, 93–96, 98, 107

FT Fourier Transform. 11, 15

IAD Integrate-And-Dump. 94–96

LIGO Laser Interferometer Gravitational Wave Observatory. 1

LISA Laser Interferometer Space Antenna. 2–5, 33, 34, 43, 71, 72, 91, 94, 98, 101, 102, 107, 108, III, V

LMS LISA Metrology System. 3, 5–7, 11, 12, 16, 21, 30, 33, 35, 42, 46, 56, 64, 91, 92, 101–105, 107, 108, III, V

LRI Long Range Interferometry. 1, 71, III, V

LUT Look-Up Table. 9, 10, 79, 94, 95, 98

NCO Numerically Controlled Oscillator. 9, 23, 25, 61, 64, 65, 80

NPRO Non-Planar Ring Oscillator. 30, 35, 107

PA Phase Accumulator. 9, 10, 77

PC Personal Computer. 3, 7

PI Proportional-Integral. 9, 10, 23, 24, 33, 34, 37–39, 47, 58, 60, 65, 74, 75, 77, 79, 80, 95, 96, 107

PLL Phase Locked Loop. 33, 34, 80

PRN Pseudo Random Noise. 91–96, 98–101, 103, 108, III, V

QPD Quadrant Photo Diode. 71–75, 78, 80, 89, 108, III, V

RAM Random Access Memory. 7, 18–22

ROM Read Only Memory. 21

SEPD Single Element Photo Diode. 88

TDI Time-Delay Interferometry. 91, 108, III, V

VHDL Very high speed integrated circuit Hardware Description Language. 17, 19, 22, 23, 57, 64, 75, 84

VRAM Video Memory. 20

Bibliography

[1] Albert Einstein and Marcel Grossmann. "Die Grundlagen der allgemeinen Relativitätstheorie". In: *Annalen der Physik* 354.7 (1916), pp. 769–822.

[2] I. Newton. *Philosophiæ naturalis principia mathematica*. 1687.

[3] Albert Einstein. "Über Gravitationswellen". In: *Sitzungsberichte der Königlich Preußischen Akademie der Wissenschaften* (1918), pp. 154–167.

[4] B. P. Abbott et al. "Observation of Gravitational Waves from a 22-Solar-Mass Binary Black Hole Coalescence". In: *Physical Review Letters* 116.15 (2016), p. 241103.

[5] T. Accadia et al. "Virgo: a laser interferometer to detect gravitational waves". In: *JINST* 7 (2002), p. 03012.

[6] G. Woan et al. "The GEO 600 Gravitational Wave Detector – Pulsar Prospects". In: *ASP Conference Series* (2003).

[7] Pau Amaro-Seoane, Sofiane Aoudia, Stanislav Babak, Pierre Binétruy, Emanuele Berti, Alejandro Bohé, Chiara Caprini, Monica Colpi, Neil J. Cornish, Karsten Danzmann, Jean-François, Dufaux, Jonathan Gair, Ian Hinder, Oliver Jennrich, Philippe Jetzer, Antoine Klein, Ryan N. Lang, Alberto Lobo, Tyson Littenberg, Sean T. McWilliams, Gijs Nelemans, Antoine Petiteau, Edward K. Porter, Bernard F. Schutz, Alberto Sesana, Robin Stebbins, Tim Sumner, Michele Vallisneri, Stefano Vitale, Marta Volonteri, Henry Ward26, and Barry Wardell. *Notes & News for GW science: eLISA/NGO, revealing a hidden universe*. 2013.

[8] Gregory M. Harry and LIGO Scientific the Collaboration. "Advanced LIGO: the next generation of gravitational wave detectors". In: *Classical and Quantum Gravity* 27.8 (2010), p. 084006.

[9] Karsten Danzmann. *Laser Interferometer Space Antenna - A proposal in response to the ESA call for L3 mission concepts*. Tech. rep. AEI, 2017. URL: https://www.elisascience.org/files/publications/LISA_L3_20170120.pdf.

[10] Simon Barke, Nils Christopher Brause, Iouri Bykov, Juan Jose Esteban Delgado, Andrew Enggaard, and Oliver Gerberding. *LISA Metrology System - Final Report*. Tech. rep. 2014. URL: http://hdl.handle.net/11858/00-001M-0000-0023-E266-6.

[11] Simon Barke. "Inter-Spacecraft Frequency Distributuion for Future Gravitational Wave Observatories". PhD Thesis. QUEST-Leibniz-Forschungsschule der Gottfried Wilhelm Leibniz Universität Hannover, 2015.

[12] Oliver Gerberding. "Phase readout for satellite interferometry". PhD thesis. Leibniz Universität Hannover, 2014.

[13] Jean-Baptise-Joseph Fourier. *Theorie analytique de la chaleur*. Paris, France: F. Didot, 1822.

[14] James W. Cooley and John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* 19 (1965), pp. 297–301.

[15] Charles M. Rader. "Discrete Fourier Transforms When the Number of Data Samples Is Prime". In: *Proceedings of the IEEE* 56.6 (1988), pp. 1107–1108.

[16] L. Bluestein. "A linear filtering approach to the computation of discrete Fourier transform". In: *IEEE Transactions on Audio and Electroacoustics* 18.4 (1970), pp. 451–455.

[17] Georg Bruun. "z-transform DFT filters and FFT's". In: *IEEE Transactions on Acoustics Speech and Signal Processing* 26.1 (1978), pp. 56–63.

[18] Tilman Butz. *Fouriertransformation für Fußgänger*. Vieweg+Teubner Verlag, 2011.

[19] Lawrence R. Rabiner. "On the Use of Symmetry in FFT Computation". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 27.3 (1979), pp. 233–239.

[20] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1988. ISBN: 0521431085.

[21] Xilinx Inc. *Virtex-6 Family Overview*. 2012. URL: http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[22] F.R.A. Hopgood, R.J. Hubbold, and D.A. Duce. *Advances in Computer Graphics II*. Springer-Verlag, 1986.

[23] Harry Nyquist. "Regeneration Theory". In: *Bell System Technical Journal* 11 (1932).

[24] E. A. Faulkner. *Introduction to the Theory of Linear Systems*. Chapman & Hall, 1969.

[25] J. Sapriel, S. Francis, and B. Kelly. *Acousto-Optics*. Wiley, 1979.

[26] Stephan Jacobs. *Heterodyne detectoin in optical communication*. Technical Note TRG-168-TDR-1. 2 Aerial Way, Syosset, New York: Technical Research Group, Inc., 1962.

[27] Thomas J. Kane and Emily A. P. Cheng. "Fast frequency tuning and phase locking of diode-pumped Nd:YAG ring lasers". In: *Optics Letters* 13.11 (1988), pp. 970–972.

[28] Coherent. *Mephisto Ultra-Narrow Linewidth CW DPSS Laser*. 2013. URL: https://www.coherent.com/downloads/Mephisto_DS_1013revA_2.pdf.

[29] Alex Abramovici and Jake Chapsky. *FEEDBACK CONTROL SYSTEMS: A Fast-Track guide for Scientists and Engineers*. Kluwer Academic Publishers, 2000.

[30] K. Küpfmüller. "On the Dynamics of Automatic Gain Controllers". In: *Elektrische Nachrichtentechnik* 5.11 (1928), pp. 459–467.

[31] G. Heinzel, A. Rüdiger, and R. Schilling. "Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows". In: (2002).

[32] Ivan Flores. *The Logic of Computer Arithmetic*. Prentice-Hall, 1963.

[33] Sheldon Axler. *Linear Algebra Done Right*. 2nd ed. Springer, 1997.

[34] E. Hogenauer. "An Economical Class of Digital Filters for Decimation and Interpolation". In: *IEEE Transactions on Acoustics, Speech and Signal Processing* 29.2 (1981), pp. 155–162.

[35] Euan Morrison, Brian J. Meers, David I. Robertson, and Henry Ward. "Automatic alignment of optical interferometers". In: *Applied Optics* 33 (1994), p. 5041.

[36] Julius S. Bendat and Allan G. Piersol. *Random Data: Analysis and Measurement Procedures*. Wiley, 2010. ISBN: 9780470248775.

[37] Markus Otto. "Time-Delay Interferometry Simulations for the Laser Interferometer Space Antenna". PhD thesis. Leibniz Universität Hannover, 2016.

[38] Juan José Esteban Delgado. "Laser Raging and Data Communication for the Laser Inerferometer Space Antenna". PhD thesis. Gottfried Wilhelm Leibniz Universität Hannover, 2012.

[39] Juan José Esteban, Antonio F. García, Johannes Eichholz, Antonio M. Peinado, Iouri Bykov, Gerhard Heinzel, and Karsten Danzmann. "Ranging and phase measurement for LISA". In: *8th Edoardo Amaldi Conference on Gravitational Waves*. Vol. 228. Journal of Physics: Conference Series. 2010, p. 012045.

[40] Hui Zhou, Charles Nicholls, Thomas Kunz, and Howard Schwartz. *Frequency Accuracy and Stability Dependencies of Crystal Oscillators*. Technical Report SCE-08-12. Carleton University, Systems and Computer Engineering, 2008.

[41] Derek J. S. Robinson. *An Introduction to Abstract Algebra*. Walter de Gruyter, 2003.

[42] Richard W. Hamming. "Error Detection and Error Correction Codes". In: *The Bell System Technical Journal* 29.2 (1950), pp. 147–160.

[43] Irving S. Reed and Gustave Solomon. "Polynomial Codes over Certain Finite Fields". In: *Journal of the Society for Industrial and Applied Mathematics* 8.2 (1960), pp. 300–304.