Design and Verification of an All-Digital Phase-Locked Loop for synchronizing a hard real-time power converter controller

Aleksi Korsman

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 16.2.2022

Supervisor

Prof. Jussi Ryynänen

Advisor

Dr. Slawosz Uznanski

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Aleksi Korsman

Title Design and Verification of an All-Digital Phase-Locked Loop for synchronizing a hard real-time power converter controller

Degree programme Electronics and Nanotechnology

Major Micro- and Nanoelectronic Circuit Design **Code of major** ELEC3036

Supervisor Prof. Jussi Ryynänen

Advisor Dr. Slawosz Uznanski

Date 16.2.2022 **Number of pages** 78 **Language** English

Abstract

Accurate time synchronization of CERN power converters is vital to ensure the synchronous performance of actions, such as the start of an acceleration ramp or stopping the beam, in particle accelerators. Additionally, it is required to provide accurate timestamps on certain events, such as faults. This thesis aims to design, verify, and characterize the PLL that belongs to a system that synchronizes a next-generation CERN power converter controller with external UTC time in µs-accuracy.

The PLL in this thesis consists of an FPGA-based phase detector, a software-based proportional-integral loop filter, and an external DAC and VCXO. The PLL design is based on an old implementation that was used in the previous controller platform but needs to be re-written due to hardware changes. The main focus is developing and verifying a modular, Wishbone-interface phase detector in VHDL, as well as creating a prototype version for the loop filter in Python.

After implementing the different PLL blocks, an analytical system model is derived. Then, the phase detector is verified with unit tests using VHDL and VUnit framework. The complete PLL is verified with system-level tests in both simulation environment, which is implemented with cocotb framework and Python, and hardware environment, which utilizes Python and a UART interface to access FPGA registers.

The system-level tests test lock-in time, jitter filtering, and VCXO drift under different conditions. The tests prove that the PLL acquires phase-lock predictably, and the system model describes the behavior accurately. Decreasing the proportional gain factor improves jitter filtering but decreases lock-in time. Decreasing the time-constant decreases lock-in time but reduces stability. Furthermore, the phase drift in the absence of the synchronization pulse is reduced by 45x compared to the old implementation.

The thesis consists of five chapters: Chapter 1 provides the motivation for the thesis. Chapter 2 introduces the required PLL theory to construct the system model. The methodology this thesis uses is explained in Chapter 3. Then, Chapter 4 describes the design and verification flow and displays the results. Finally, Chapter 5 summarizes the thesis.

Keywords PLL, FPGA, verification, programmable logic

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Aleksi Korsman

Työn nimi Digitaalinen vaihelukittu silmukka kovan reaaliaikaisen tehomuuntajaohjaimen synkronoimiseksi

Koulutusohjelma Elektroniikka ja nanoteknologia

Pääaine Mikro- ja nanoelektroninen piirisuunnittelu **Pääaineen koodi** ELEC3036

Työn valvoja Prof. Jussi Ryynänen

Työn ohjaaja TkT Slawosz Uznanski

Päivämäärä 16.2.2022 **Sivumäärä** 78 **Kieli** Englanti

Tiivistelmä

CERN:n tehomuuntajien tarkka aikasynkronointi on elintärkeää toimintojen, kuten kiihdytysramppien aloituksen tai hiukkasuihkun pysäyttämisen, samanaikaisen suorittamisen varmistamiseksi hiukkaskiihdyttimissä. Sen lisäksi sitä tarvitaan tarkkojen aikaleimojen tuottamiseksi tapahtumille, kuten virheille. Tämän diplomityön tavoite on suunnitella, verifioida, ja karakterisoida vaihelukittu silmukka (PLL), joka kuuluu järjestelmään, joka synkronoi seuraavan sukupolven CERN tehomuuntajaohjaimen ulkoisen UTC ajan kanssa mikrosekunnin tarkkuudella.

Tämän työn PLL koostuu FPGA-pohjaisesta vaihevertaimesta, ohjelmistopohjaisesta PI-säädin silmukkasuodattimesta, ja erillisestä DA-muuntimesta (DAC) ja jänniteohjatusta kideoskillaattorista (VCXO). PLL toteutus perustuu vanhaan toteutukseen, jota käytettiin edellisessä ohjainalustassa, mutta joka tulee kirjoittaa uudelleen elektroniikan muutoksien vuoksi. Painopiste on modulaarisen, Wishbone-rajapintaisen vaihevertaimen kehittämisessä VHDL-kielellä. Silmukkasuodatin kehitetään Python-kielellä, ja sen on tarkoitus olla prototyyppiversio.

Sen jälkeen, kun eri PLL osat on toteutettu, johdetaan analyyttinen järjestelmämalli. Sitten vaihevertain verifioidaan yksikkötesteillä käyttäen VHDL:ää ja VUnit kehystä. Valmis PLL verifioidaan järjestelmätason testeillä sekä simulaatiotasolla, joka toteutetaan cocotb kehyksellä ja Python-kielellä, että laitteistotasolla, joka hyödyntää Python-kieltä ja UART-rajapintaa FPGA rekisterien käsiksipääsemiseen.

Järjestelmätason testit testaavat lukittumisaikaa, värinäsuodatusta ja kideoskillaattorin ajelehtimista eri olosuhteissa. Testit osoittavat, että PLL lukittuu ennustettavasti, ja että järjestelmämalli kuvaa PLL:n käyttäytymistä tarkasti. Proportionaalisen vahvistuksen pienentäminen parantaa värinäsuodatusta, mutta pidentää lukittumisaikaa. Aikavakion pienentäminen lyhentää lukittumisaikaa, mutta heikentää stabiilisuutta. Sen lisäksi vaiheen ajelehtiminen synkronointipulssin puuttuessa pieneni 45-kertaisesti verrattuna vanhaan alustaan.

Tämä työ koostuu viidestä luvusta: Luku 1 tarjoaa motivaation työlle. Luku 2 esittelee tarvittavan PLL teorian, jotta systeemimalli voidaan luoda. Työssä käytetty metodologia kerrotaan Luvussa 3. Seuraavaksi Luku 4 kuvailee toteutuksen ja verifioinnin etenemisen ja esittelee tulokset. Lopuksi Luku 5 vetää työn yhteen.

Avainsanat vaihelukittu silmukka, FPGA, verifiointi, ohjelmoitava logiikka

Preface

I want to thank CERN and my supervisor and thesis-instructor Slawosz Uznanski for providing me with the opportunity of working in such a high-level organization for a year, and letting me work on my thesis during the stay. The year will surely be one that I will remember for the rest of my life. Also, I say thank you to prof. Jussi Ryynänen for great feedback during the thesis writing process. Finally, I thank my family and friends for providing support and understanding over the study years.

Nurmijärvi, 22.2.2022

Aleksi Korsman

Contents

Symbols, operators, and abbreviations

Symbols

A_i	PLL input signal amplitude (V)
A_o	VCO output signal amplitude (V)
D	Delay (clock cycles)
DAC_{res}	DAC resolution (number of bits)
f_{clk}	System clock frequency (Hz)
$H(s)$	Transfer function
κ_d	Phase-detector gain factor (discrete domain)
κ_i	Integral gain factor
κ_o	VCO gain factor (discrete domain)
κ_p	Proportional gain factor
K	Loop gain
K_d	Phase-detector gain factor (continuous domain)
K_o	VCO gain factor (continuous domain)
ω	Angular frequency (rad/s)
$\Delta\omega_{o,max}$	VCO pullability (rad/s)
$\Delta\omega_o$	VCO frequency resolution (rad/s)
ω_n	Natural frequency (rad/s)
s	Laplace variable
θ_i	PLL input phase (rad)
θ_o	VCO output phase (rad)
θ_e	Phase error $\theta_i - \theta_o$ (rad)
T_{clk}	System clock period (s)
T_{v_i}	Period of v_i (s)
t_s	PLL sampling interval (s)
τ_2	Time constant of stabilizing zero
$v_{c,max}$	DAC control scale (V)
v_d	PD output signal (V)
v_i	PLL input signal (V)
v_o	VCO output signal (V)
V_c	LF output signal (V)
V_{co}	Control voltage in lock (V)
V_{dd}	Positive supply voltage (V)
V_{do}	Free-running voltage (V)
ζ	Damping factor

Operators

$\mathcal{L}\{\}$	Laplace-transform
$\mathcal{Z}\{\}$	Z-transform

Abbreviations

ADC	Analog-To-Digital Converter
ADPLL	All-Digital Phase-Locked Loop
ASIC	Application-Specific Integrated Circuit
CERN	European Organization for Nuclear Research
DAC	Digital-To-Analog Converter
DCO	Digitally Controlled Oscillator
DUV	Design Under Verification
FIFO	First-In First-Out
FGC	Function Generator/Controller
FOSS	Free and Open Source Software
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
LF	Loop Filter
LHC	Large Hadron Collider
PCB	Printed Circuit Board
PD	Phase Detector
PI	Proportional-Integrator
PLL	Phase-Locked Loop
ppm	parts-per-million
RF	Radio Frequency
RTL	Register-Transfer Level
SOC	System-On-Chip
UTC	Coordinated Universal Time
VCO	Voltage Controlled Oscillator
VCXO	Voltage Controlled Crystal Oscillator
VHDL	VHSIC Hardware Description Language

1 Introduction

It is well known that real-time implementations of control algorithms are greatly affected by time-varying delays, also known as jitter, by degrading the stability and the overall performance of the system [1, 2]. The effect is amplified when the number of controlled elements in the system increases. Often, these effects are difficult to predict in the control design phase of the system and emerge only after the implementation. Therefore, control system designers strive to identify the sources of jitter in their system and discover methods to minimize their effect.

These challenges are also present in the control of enormous particle accelerators at CERN, the largest particle physics laboratory in the world. One of these accelerators is the most powerful particle accelerator in the world, the Large Hadron Collider (LHC). The LHC is a circular particle accelerator and a collider with a circumference of 27 km, buried approximately 100 meters underground. Inside two beam pipes, two counter-circulating proton beams are first accelerated from 450 GeV to 6.5 TeV [3], which corresponds to a velocity of 0.99999999% of the speed of light. Then, magnets steer the beams to collide with each other in four experimental points (ATLAS, ALICE, LHCb, and CMS), resulting in 13 TeV of center-of-mass collision energy.

In a circular accelerator, such as the LHC, the beam pipe bends at specific parts of the accelerator, eventually creating a complete loop. Thus, the trajectory of the particle beam must be manipulated in some way to ensure it remains within the beam pipe. The accelerator includes powerful electromagnets that accomplish the manipulation by generating a magnetic field inside the beampipe. Since the traversing particles have an electric charge, they experience a force according to the Lorentz force, thus curving inwards.

The strength of the magnetic field is directly related to the amount of electric current flowing through electromagnets. Therefore, very high currents are required to keep the high-energy particles on their trajectory. For example, some of the superconducting magnets in the LHC can draw over 10 kA of current [4]. Moreover, the current must also be precise to allow the manipulation of the beam in the micrometer scale, which is required to keep the protons in a tight bunch and enable accurate collisions. General target accuracy is one part-per-million (ppm) of the reference set by the accelerator operator [5]. The current for, for example, a 13 kA superconducting magnet must be accurate to 13 mA to achieve this target.

Custom-design Electric Power Converters (EPCs) supply the necessary current for superconducting and normal electromagnets. There are many different EPCs in use depending on the application, but fundamentally, their operating principle is to convert power from the power grid into a desired voltage or current. In one accelerator, there can be over a thousand EPCs to supply all magnets with the desired current. Therefore, all EPCs form a distributed system that must operate synchronously with each other to perform actions simultaneously, decreasing jitter in the system.

To achieve high performance and accuracy, the EPCs include a high-performance power converter controller connected to a high-accuracy current measurement device. The main task of a power converter controller is to run a current regulation

loop. One loop consists of the controller receiving a measurement of the output current, calculating the difference between the measured and desired values, and then communicating to the EPC to decrease that difference.

The latest high-performance power converter controller at CERN is the Function Generator/Controller 3.1 (FGC3.1). It is a part of a control platform called RegFGC3, a modular platform in which modules can be added and removed to control different EPCs. The FGC3.1 is the brain of that platform that includes a high-performance processor for actions, such as performing the calculations for the current regulation loop. The FGC3.1 runs a regulation loop every 100 microseconds, resulting in a regulation frequency of 10 kHz [6].

However, some power converters at CERN require higher frequency regulation. Therefore, CERN launched a new project called FGC3.2 with the main target of increasing the regulation frequency [6]. In addition, the hardware and software architectures have drastically changed due to the introduction of a new processor and a Linux-based operating system instead of bare-metal. These changes pose another challenge, as many software and hardware modules must be re-designed to function on the new platform. One of the modules to be re-designed is the digital Phase-Locked Loop (PLL).

PLLs are widely used in technologies, such as telecommunications, computers, and motor controllers, to realize functionalities, such as FM demodulation, frequency synthesis, and timing distribution [7, 8, 9]. In the FGC3.1, a PLL is used to synchronize the onboard clock with an external UTC clock to microsecond accuracy [10, 11]. This is required to enable operating all EPCs in a synchronized way to minimize the effects of time-varying delays and to provide measurements with accurate timestamps.

The absolute UTC time is distributed in an Ethernet-based fieldbus called FGC_Ether [10]. The FGC receives an Ethernet frame, which includes the UTC time, periodically every 20 ms. However, the Ethernet frames can have minor variations in transmission delays, which can cause a time uncertainty of 10s of microseconds. Therefore, a 50 Hz reference pulse is distributed parallel with the Ethernet frames to provide an accurate 20-millisecond tick. The PLL will lock the internal program cycle, which also loops every 20 ms, of the FGC to this reference pulse. This way, a time precision of less than 2.5 microseconds has been achieved [11].

The PLL in FGC3.1 is implemented as a combination of programmable logic on a Field-Programmable Gate Array (FPGA) and software on a microprocessor. The FPGA program on FGC3.1 has been written with little modularity, thus preventing it from being directly reused on FGC3.2. Moreover, FGC3.2 includes updated hardware, like a higher-resolution Digital-to-Analog Converter (DAC), which shall change the operating characteristics. Therefore, the FPGA program for the PLL must be re-developed and verified.

This thesis aims to develop and verify the programmable logic of the PLL for FGC3.2 in a modular and reuse-friendly way. This goal is achieved by utilizing industry-standard methods, including implementing a Wishbone memory interface [12], and running unit-tests using the VUnit [13] framework.

Secondly, this thesis aims to design a prototype software program to verify the operation of the complete PLL, create a simulation environment for the full PLL, and characterize the full PLL by developing a system model and observing the behavior with different loop-parameters both in simulation and hardware environments. The prototype software program is designed with Python programming language by adapting from the software program of FGC3.1. A framework called cocotb [14] is used to create the simulation environment for simulating programmable logic and Python simultaneously. Finally, test cases are run in simulation and hardware environments to verify the PLL operation. The PLL is characterized according to the system model and the results from these tests.

The thesis is structured as follows: First, Chapter 2 introduces the theory of digital PLLs and lays the background for composing the system model. Then, Chapter 3 describes the methodology used to develop the modules and execute the tests in this thesis. The thesis culminates in Chapter 4, in which the design and verification processes are described, alongside the composition of the system model and presentation of simulation and test results. Finally, Chapter 5 summarizes the thesis.

2 All-Digital Phase-Locked Loop Theory

This Chapter provides the required theory of phase-locked loops (PLLs) to create an analytical system model of the implemented system. It introduces the essential concepts and equations by citing some of the most popular books in the field. Additionally, some common implementations of the separate PLL blocks are provided.

2.1 Introduction and Architecture

PLL is a widely used component in telecommunications, computers, and motor controllers to realize functionalities like FM demodulation, frequency synthesis, and timing distribution. Fundamentally, it is an oscillator whose phase is locked to the phase of the input signal. In other words, at all times, the PLL attempts to minimize the phase difference between the input signal v_i and the output signal v_o. As displayed in Figure 1, the PLL is a closed-loop system consisting of three main building blocks:

- Phase detector (PD)
- Loop filter (LF)
- Voltage-controlled oscillator (VCO)

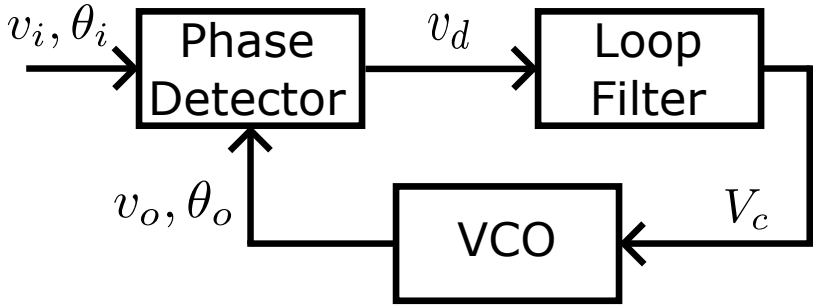


Figure 1: PLL basic building blocks

Here, the output signal v_o, generated by the VCO, is compared with the input signal v_i in the phase detector. Their phase difference, $\theta_i - \theta_o$, is converted into a signal v_d. This signal continues to a loop filter that performs some filter function to generate a suitable output signal V_c that controls the output frequency of the VCO. Small changes in the frequency of v_o effectively change its phase with respect to v_i, preferably decreasing the phase difference. In the following subsections, these individual blocks are handled in further detail.

The blocks can be implemented using analog or digital components or a mixture of both. Recently, digital variants have often been chosen thanks to their easier fabrication, high accuracy, and close-to-perfect predictability. [7, 15]

It is useful here to make a distinction between a *digital* and an *all-digital* PLL implementation. According to the definition in [9], the digital PLL still incorporates

analog signals and only utilizes digital components like flip-flops in the phase detector. In an all-digital PLL, also the signals are digital. The PLL of this thesis will be an all-digital version, mainly due to the easy and flexible implementation on an FPGA and a microcontroller.

The theoretical analysis of PLLs is convenient to be performed in either Laplace-domain or z-domain. For an all-digital PLL, the z-domain analysis is, naturally, more suitable. However, the common formulas in the literature are often presented in the Laplace-domain, and, as will be evident, the two domains do not differ much from each other. Therefore, both domains are used, but the final system model will be derived using the z-domain.

2.2 Basic equations

In this section, the PLL essential operation is introduced in the continuous-time domain for better intuition. Later in this chapter, the corresponding discrete-time equations are presented.

First of all, to simplify the theoretical analysis of the system, the input signal v_i in figure 1 is assumed a sine wave with angular frequency ω, amplitude A_i, and a constant phase offset θ_i:

$$v_i = A_i \sin(\omega t + \theta_i) \tag{1}$$

In practice, the input can be any periodical signal: it is usually a square wave for clock synchronization. The other input to the PD is the output from the VCO. It is also assumed sinusoidal with the same angular frequency ω, but a different phase θ_o and amplitude A_o:

$$v_o = A_o \sin(\omega t + \theta_o) \tag{2}$$

The purpose of the PD is to find the phase difference $\theta_e = \theta_i - \theta_o$, and multiply it with a *phase-detector gain factor* K_d. To the result is added an offset called *free-running voltage* V_{do}, which essentially indicates the PD output at an absence of input signal v_i, or at zero phase difference θ_e. The PD equation becomes:

$$v_d = K_d \theta_e + V_{do} \tag{3}$$

The LF filters the PD output by applying a filter function $F(s)$. Here, the output from PD is converted to the Laplace domain. The LF equation becomes:

$$V_c = F(s) V_d(s) \tag{4}$$

To decrease the phase difference θ_e, the VCO output frequency must deviate from the input signal frequency. This deviation is called *output frequency deviation* $\Delta\omega = \omega_o - \omega_i$. The deviation is then multiplied by a VCO property called *VCO gain factor* K_o. The VCO equation becomes:

$$\Delta\omega = K_o(v_c - V_{co}), \tag{5}$$

where V_{co} is the *control voltage in lock* and indicates the voltage when $\Delta\omega = 0$.

The equations (3), (4), and (5) describe the essential operation of each PLL block and how they are interconnected. In the following sections, the blocks are investigated in more detail.

2.3 Phase detector

Phase detector is a device that takes in two signals ((1) and (2)) and outputs a signal that is proportional to their phase difference (eq. (3)). The phase difference of two sinusoidal signals is illustrated in Figure 2.

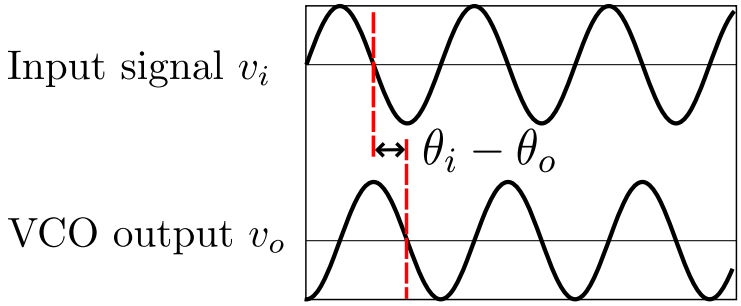


Figure 2: Phase Detector working principle

For the analytical analysis in the Laplace domain, the free-running voltage V_{co} is omitted for simplicity. In other words, the PD output signal is assumed not to have an offset. Then, the Laplace notation of (3) becomes:

$$V_d(s) = \mathcal{L}\{v_d(t)\} = K_d\theta_e(s) = K_d(\theta_i(s) - \theta_o(s)) \tag{6}$$

The digital representation replaces the continuous-time domain equation with a discrete-time domain equation. The variable for time, t, which can have any value, changes into the sample number n, which can only be an integer. The resulting equation is called the difference equation, and, for the PD, it looks very similar to the continuous-time equation:

$$v_d[n] = \kappa_d(\theta_i[n] - \theta_o[n]) \tag{7}$$

Here, the PD gain factor K_d has been replaced with its discrete-time replacement, κ_d. To convert the difference equation to z-domain equation, an operation called *z-transformation* is performed:

$$V_d(z) = \mathcal{Z}\{v_d[n]\} = \kappa_d(\theta_i(z) - \theta_o(z)) \tag{8}$$

where κ_d also denotes the phase detector gain, but is separated from K_d in the analog formula for clarity purposes.

Something noteworthy is that both of these are linear equations, whereas real-life PDs always have nonlinearities. An analog implementation, which can be realized with, e.g., a four-quadrant mixer, is, typically, linear only for small phase errors.

The behavior might need to be separately analysed on nonlinear regions. Digital implementations, in turn, suffer from quantization noise, especially at small phase errors. Therefore, the quantization steps must be small enough to approximate the PD as linear. To summarize, if linear equations are used, it must be ensured that these approximations are valid.

As previously mentioned, all-digital solutions rely on logic devices and purely digital signals. Some all-digital PDs are introduced in [9]. They are based on counters, digital multiplication, zero-crossing, Hilbert transform, or digital averaging. Except for the counter-based one, all types require an analog-to-digital converter (ADC) to sample the input signal.

Nowadays, another widely popular component for all-digital PDs is the time-to-digital converter (TDC). The TDC measures the phase difference by calculating the number of inverter propagation delays between the two input signals and can achieve a resolution of picoseconds [16]. However, its implementation usually requires dedicated hardware, which is possible and feasible on an Application-Specific Integrated Circuit (ASIC) but more difficult on an FPGA. Even though it is implementable on an FPGA by using low-level functions and manual placement and routing, it requires extra effort and drastically reduces the predictability and portability of the design.

Regarding the application of this thesis, all ADC-based solutions can be considered irrelevant since the input signal will be a square wave with no modulated information. The TDC-based PD can also be ignored for the issues mentioned above with an FPGA implementation. Therefore, this thesis's most relevant PD type is the counter-based PD. It is easy to implement on an FPGA and delivers satisfactory performance, as the sampled signal is only 50 Hz.

The simple counter-based PD is displayed in Figure 3 and functions as follows: A rising-edge of the reference signal v_i triggers the flip-flop which, after a propagation delay t_p, flips the output Q from a low to a high state. This transition, in turn, first resets and then enables the counter, which starts incrementing its internal register by one every time it receives a clock signal from the system clock clk_{sys}. This incrementation continues until the rising edge of v_o resets the flip-flop output Q back to the low state, disabling the counter. In other words, the counter stops incrementing its internal register, and the value stored within indicates the phase difference between v_i and v_o. The phase error, $\theta_{e,dig}$, is in the form of a digital word and, therefore, must be converted to time by multiplying it with the clock period $T_{clk} = 1/f_{clk_{sys}}$.

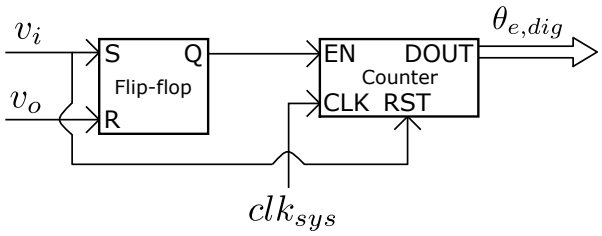


Figure 3: Simple counter-based PD

The resolution, i.e., the smallest detectable θ_e, for this type of PD is determined

by the frequency of clk_{sys}. State-of-the-art high-performance 8-bit counters can be run with a clock in the gigahertz range [17, p. 89], so it is possible to gain an ns-resolution. Whether the system clock provides sufficient resolution depends mainly on the application and the frequency of v_i: a better resolution is generally required for a higher frequency. The phase difference θ_e can be calculated with the following formulas either in time or in phase:

$$\theta_{e,time} = \theta_{e,dig} T_{clk} = \frac{\theta_{e,dig}}{f_{clk_{sys}}} \tag{9}$$

$$\theta_{e,phase} = 2\pi \frac{\theta_{e,time}}{T_{v_i}} = 2\pi f_{v_i} \theta_{e,time} \tag{10}$$

where f_{v_i} and $T_{v_i} = 1/f_{v_i}$ indicate the frequency and the period of the input signal v_i, respectively.

This type of PD allows a straightforward transformation of the phase error into the digital domain, allowing it to be readily transmitted to a fully digital loop filter. Thus, κ_d is composed of the conversion of phase into a digital word. Nevertheless, as the signal is in the digital domain, additional multipliers can be added with little cost, increasing the system's flexibility.

2.4 Loop filter

Loop filter receives the output of the PD and applies a filter function $F(s)$ (eq. (4)). Loop filters are not primarily filters but loop controllers, delivering a suitable control signal to the VCO and establishing desired loop dynamics [7]. Only their secondary function is filtering out any unwanted high-frequency signals. Indeed, the circuits used as loop filters do not necessarily resemble filters in their traditional definition.

A digital loop filter consists of proportional, integral, and delay elements. In addition, some high-frequency filtering elements can be included. The proportional element simply multiplies the phase error with a multiplier κ_p:

$$f_p[n] = \kappa_p v_d[n] \tag{11}$$

$$F_p(z) = \mathcal{Z}\{f_p[n]\} = \kappa_p V_d(z) \tag{12}$$

The integrator block can be formulated in a few different ways, depending on the location of the delay element. The one presented here is used for the rest of the thesis:

$$f_i[n] = \kappa_i v_d[n] + f_i[n-1] \tag{13}$$

$$F_i(z) = \mathcal{Z}\{f_i[n]\} = \frac{K_i}{1 - z^{-1}} \tag{14}$$

Finally, the delay elements are defined as follows:

$$y_d[n] = x_d[n - D] \tag{15}$$

$$Y_d(z) = \mathcal{Z}\{y_d[n]\} = z^{-D} X_d(z) \tag{16}$$

As explained in [7] and [8], the integrator elements have a significant effect on the whole PLL loop. The number of them defines the PLL *type*. It is also indicated by the number of poles at origin for the open-loop transfer function. Usually, the PLL is type-1 if the LF consists of only proportional and delay elements and no integrators. In that case, there is only one integrator in the loop, the VCO. Having one integrator in the LF means the PLL would be type-2. The type determines the steady-state error behavior. For example, the type-1 PLL cannot remove the steady-state phase error for a step-change in input frequency, but the type-2 PLL is. However, in the presence of a constant change in frequency, even the type-2 PLL cannot remove the steady-state phase error: a type-3 PLL is needed.

Furthermore, another property for a PLL is its *order*. The order is determined from the degree of the characteristic polynomial of the transfer function, i.e., the denominator. The roots of the characteristic equation are the poles of the transfer function: thus, the order indicates the number of poles for the system. However, the order does not affect the PLL characteristics as the type does. The order is always at least as high as the type. However, if the PLL includes additional nonintegrating filtering, the order can be larger than the type.

At least one delay element is needed in the entire all-digital PLL: this is fundamental for a digital system. Their number can vary and is directed mainly by the difference between the system and the sampling clock. If the system clock is much higher frequency than the sampling clock, one delay element may be enough for the whole loop. However, data pipelining is required if they are close to each other, as operations such as multiplications require multiple system clock cycles. Pipelining introduces more delays around the PLL loop.

A common way to combine these blocks in a PLL is a proportional-integral (PI) controller. It incorporates one proportional and one integrator block in parallel, thus resulting in, at least, a type-2 PLL. The PI-block is illustrated in Figure 4, in which the delay block in the integrator path is arranged in a way that results in the same transfer function as (14). An important part of the PLL design is to choose values for the two parameters: κ_p and κ_i. This process is described later in Section 2.6. Ideally, the parameters can be programmed to allow experimentation to find the best values.

The transfer function for the whole block is then:

$$F_{p+i}(z) = \kappa_p + \frac{\kappa_i}{1 - z^{-1}} = \frac{\kappa_p(1 - z^{-1}) + \kappa_i}{1 - z^{-1}} \tag{17}$$

The PI-controller is also common in analog implementations, in which it can be realized with, for example, an operational amplifier. It leads to a type-2 PLL in that case as well. However, the coefficients are realized with passive components like resistors and capacitors, which are subject to aging and temperature drifts,

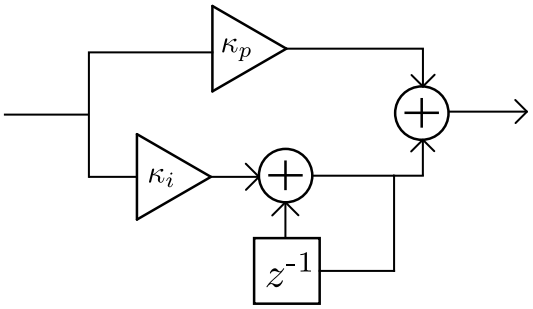


Figure 4: PI-block

affecting the PLL characteristics. With an all-digital implementation, as they can be programmatically set as digital values in a register, they do not change over time unless the user wants to update them for some reason. The programmability is one great advantage for the all-digital PLLs.

As the PLL in this thesis is all-digital, it will utilize an all-digital loop filter. The PI-controller is sufficient in the combined performance of following the reference and filtering jitter from the input signal.

2.5 Voltage-controlled oscillator

A voltage-controlled oscillator (VCO) generates a signal with a frequency ω_o, controlled by the voltage v_c. The increase of ω_o for a change in v_c depends on the VCO gain factor K_o. The specific v_c that generates a ω_o of the desired center frequency is called the in-lock control voltage V_{co}.

The time-domain equation for the VCO is given in (5). At first, the phase-shifting property of the VCO might not be obvious. However, the angular frequency ω and the phase θ are interconnected: $\omega = \frac{d\theta}{dt}$ and $\Delta\omega = \frac{d\theta_i}{dt} - \frac{d\theta_o}{dt}$. The frequency deviation $\Delta\omega$ tells how much faster or slower θ_o is changing compared to θ_i. Eventually, it is desirable for the deviation to be zero, but during the lock-in process it must be non-zero so that the two phases can reach equality.

Now, if the angular frequencies are replaced with phases in (5), the analog format for the VCO equation becomes:

$$\frac{d\theta_o}{dt} = \frac{d\theta_i}{dt} + K_o(v_c - V_{co}) \tag{18}$$

As described in [7], the VCO can be implemented in multiple ways. Some examples include quartz crystals, LC oscillators, and ring oscillators: the type decision depends on the particular application. For example, if it is desirable to implement the whole PLL on a single chip, a ring oscillator is an attractive choice, as it consists of inverters that can be conveniently adapted to every manufacturing process. However, if minimal phase noise is required, a quartz crystal oscillator is the most suitable one, but it is not integrable on a chip and, thus, requires an external component. The LC oscillator offers a compromise, where it is possible to integrate

it on a chip and has lower phase noise than the ring oscillator, but it requires sizeable components and does not scale with the ever-diminishing manufacturing processes.

The VCO is replaced with a numerically (NCO) or digitally controlled oscillator (DCO) in all-digital implementations. A common theme is that the oscillator output frequency is controlled with a digital word instead of an analog voltage. One trivial method of implementing this is utilizing an ordinary VCO and controlling it with a DAC. Another method is to deploy a so-called Direct Digital Synthesizer (DDS), in which the analog waveform is directly generated with a DAC. The DDS has a much wider frequency range than the VCO but cannot be utilized for very high frequencies and suffers from spurious signals [18].

The DCO equation for an implementation that utilizes a VCO and a DAC can be formed as:

$$\omega_o[n] = \kappa_o(v_c[n] - V_{co}) \left(\frac{\text{rad}}{\text{cycle}}\right) \tag{19}$$

where the output frequency ω_o is determined by the discrete DAC output voltage $v_c[n]$. It should be noted that the unit is radians per cycle, instead of radians per second, as the equation solves the phase shift during one clock cycle. Therefore, a different gain factor must be used compared to the analog VCO. Usually, the DAC output voltage is unipolar. Therefore, the term V_{co} is still required to indicate the voltage for the center frequency. However, for theoretical analysis, it can be assumed zero.

The DCO gain factor, κ_o can be derived from K_o:

$$\kappa_o = K_o t_s \tag{20}$$

where t_s is the PLL sampling interval. For example, if an analog VCO had a frequency of 25 rad/s, its DCO counterpart would have a frequency of 25 rad/cycle if the sampling frequency is 1 second. If it is 2 seconds, the DCO would shift 50 radians per clock cycle, and the frequency would be 50 rad/cycles. Thus, the scaling factor for the gain multiplier is obvious.

Next, to obtain the z-domain transfer function, the frequency must be integrated to phase:

$$\theta_o[n] = \sum_{n=0}^{\infty} \omega_o[n] = \kappa_o \sum_{n=0}^{\infty} v_c[n] \tag{21}$$

Now, the z-transformation can be applied:

$$\theta_o(z) = \mathcal{Z}\{\theta_o[n]\} = \kappa_o \frac{z}{z-1} v_c(z) \tag{22}$$

The VCO has two main design parameters: the center frequency, which is often the same as the input frequency ω_i, and the VCO gain K_o. Two other properties are deeply related to K_o, namely the pullability, i.e., the tuning range, and the control scale. Pullabililty indicates the maximum amount that ω_o can deviate from the center frequency: the higher the pullability, the more the ω_o can be "pulled" away from the center. The control scale indicates the minimum and maximum allowed

values of v_c. K_o can be defined by them alone, assuming that K_o is linear for every value of v_c. If pullability is defined as $\Delta\omega_{o,max}$, and control scale as $v_{c,max}$, K_o can be calculated as:

$$K_o = \frac{\Delta\omega_{o,max}}{v_{c,max}} \quad \left(\frac{\text{rad}}{\text{V}s}\right) \tag{23}$$

As VCOs are imperfect devices, they are also given their stability characteristics. The value means how much the VCO center frequency can vary. This value includes the variation caused by temperature differences within a given temperature range and the drift of center frequency due to aging. Absolute pullability around the center frequency is affected by this drift. Therefore, datasheets often mention pullability as absolute pull range (APR). This range describes the absolute pullability around the declared center frequency, regardless of the drift. For example, if the stability is informed as ±10 ppm and relative pullability as ±150 ppm, the APR would be $150 - 10 = 140$ ppm.

The DAC resolution also limits the DCO that comprises a VCO and a DAC. The frequency resolution derives from it. This nonlinearity is especially critical at low phase errors, as it will define the slight jitter around the zero phase error when the PLL is locked. The frequency resolution $\Delta\omega_o$ can be calculated from the DAC resolution and VCO pullability:

$$\Delta\omega_o = \frac{\Delta\omega_{o,max}}{\text{DAC}_{res}} \tag{24}$$

Here, DAC_{res} means the number of discrete steps the output scale is divided into. For example, in an 8-bit DAC, the output value is devided into 256 discrete steps, and DAC_{res} would be 256.

This thesis utilized a combination of a quartz crystal VCO, which is also called the Voltage Controlled Crystal Oscillator (VCXO), with an external DAC. The properties of the VCXO and the DAC are presented later in Section 4.2.

2.6 Full system properties

Understanding the complete system is essential to verify the operation of the PLL. For this task, the transfer function is an important model. It describes the behavior of a control system by expressing the input-output relationship, and it can be conveniently organized into block diagrams and signal-flow graphs like Figures 1 and 4. The same principles apply as a PLL is fundamentally a control system. From transfer functions, specific properties can be derived. These properties are mostly defined for continuous-time systems, i.e., analog systems, but after applying some approximations, they can also be applied to discrete-time systems.

2.6.1 Analog PLL

The common analog second-order type-2 PLL has a transfer function that looks like the following [7]:

$$H(s) = \frac{2\zeta\omega_n s + \omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{25}$$

The two variables present in the transfer function are the damping factor ζ and the natural frequency ω_n. These variables have a direct relationship with the system behavior. The damping factor determines the response to a step input. A system with a low damping factor ($\zeta < 1$) is called an underdamped system, and a system with a high damping factor ($\zeta > 1$) is called an overdamped system. A damping factor of exactly one causes a critically damped system. The decision of damping factor is generally a compromise between overshoot and response time: a lower ζ causes a faster response but higher overshoot. For PLL systems, a value of 0.707 is a common choice, but sometimes higher damping is required to reduce overshoot [7].

The natural frequency ω_n indicates the system's oscillation frequency when there is no damping at all ($\zeta = 0$). The value can be decided from a wide range of values, from millihertz to gigahertz. In general, it should be high enough to have a fast lock but low enough to filter out noise [7]. In the frequency response of a PLL, the spectrum is relatively flat until reaching approximately the natural frequency [9]. However, it is not the best indicator for the PLL bandwidth, although it is sometimes treated as one [7].

Another loop parameter is the loop gain K. It indicates the open-loop gain through the proportional path, i.e., it ignores the integrator blocks. This parameter is highly valuable as it is a good indicator of the PLL bandwidth, and it applies to all PLL types [7]. For type-2, the loop gain can be derived from ω_n and ζ:

$$K = 2\zeta\omega_n \tag{26}$$

The final property to be introduced is the loop filter time constant τ_2. For a type-2 PLL, it indicates the position of the transfer function zero. Additionally, the time constant determines the capture range, i.e., the range of frequencies around the VCO center frequency, where the PLL can lock onto the input signal. The higher τ_2, the lower the capture range. It can also be derived from the previously introduced parameters:

$$\tau_2 = \frac{2\zeta}{\omega_n} \tag{27}$$

Out of the four introduced parameters, the PLL designer must choose two. However, no matter how well the analytic part of the design is performed, trial and error often find the best parameters. Nevertheless, knowing the physical properties behind the parameters will ease finding appropriate starting values for the experimentation process.

2.6.2 Digital PLL

A typical transfer function for a second-order type-2 digital PLL, adapted from [7], is:

$$H(z) = \frac{\kappa z^{-D}(1 - z^{-1} + \kappa_2)}{(1 - z^{-1})^2 + \kappa z^{-D}(1 - z^{-1} + \kappa_2)} \tag{28}$$

Here, D corresponds to the number of delays in the system, and the minimum value is 1. κ and κ_2 are composed of proportional, integral, PD, and VCO gains.

The properties introduced in the previous section are not directly translatable to digital PLLs. However, an approximation called *time-continuous approximation* can be introduced. Following first-order approximations can be exercised if $|Dst_s| \ll 1$ applies (t_s is the sampling period) [7]:

$$z^{-1} \approx 1, \qquad z^{-D} \approx 1, \qquad 1 - z^{-1} \approx st_s \tag{29}$$

When applied to (28), the following transfer function is acquired:

$$H(s) \approx \frac{\kappa(st_s + \kappa_2)}{s^2 t_s^2 + \kappa(st_s + \kappa_2)} = \frac{s\kappa/t_s + \kappa\kappa_2/t_s^2}{s^2 + \kappa s/t_s + \kappa\kappa_2/t_s^2} \tag{30}$$

Now, comparing to (25), the familiar system properties ζ and ω_n can be derived:

$$\omega_n \approx \frac{1}{t_s}\sqrt{\kappa\kappa_2} \qquad \zeta \approx \frac{1}{2}\sqrt{\frac{\kappa}{\kappa_2}} \tag{31}$$

Similarly, K and τ_2 can be derived using (26), (27), and (31).

$$K = 2\zeta\omega_n \approx \frac{\kappa}{t_s} \qquad \tau_2 = \frac{2\zeta}{\omega_n} \approx \frac{t_s}{\kappa_2} \tag{32}$$

2.7 Stability

Stability is a concept that is familiar from control theory and similarly important for PLLs. As defined in [19], "A stable system is a dynamic system with a bounded response to a bounded input." In other words: For any displacement in the input, the response should decrease over time and eventually converge to a steady-state.

An unstable PLL would be unable to lock to the input phase. Instead of reaching a steady state, it would keep oscillating with a constant or increasing amplitude. For a PLL, stability is a desired design goal. Often, stable behavior is established by proper loop filter design. As a part of characterizing the PLL, it is vital to define its stability limits.

2.7.1 Analog PLL

Transfer function poles play a crucial role in stability analysis. For an analog PLL, if all poles are located in the left-hand portion of the s-plane, the system is stable [7, 19]. This is the case if the poles include a negative real part. Poles located on the $j\omega$-axis will result in a neutral response. On the other hand, poles located on the right-hand portion results in an unstable system. Therefore, ensuring that the poles of the transfer function lie on the left-hand portion is critical.

The stability analysis for a second-order type-2 PLL can be made by assigning the characteristic equation, i.e., the denominator, of (25) as zero, and solving the equation:

$$\begin{aligned} & s^2 + 2\zeta\omega_n s + \omega_n^2 = 0 \\ & s = \frac{2\zeta\omega_n \pm \sqrt{(2\zeta\omega_n)^2 - 4\omega_n^2}}{2} \\ & s = \omega_n(-\zeta \pm \sqrt{\zeta^2 - 1}) \end{aligned} \tag{33}$$

Here, it is clear from (33) that, for all positive ζ and ω_n, s is a pair of values with a negative real part. For $\zeta > 1$, the poles are real and separate. For $0 < \zeta < 1$, the poles are a complex-conjugate pair. The poles are real and coincident for the special case of $\zeta = 1$. In all cases, the poles are located in the left-hand portion of the s-plane. Thus, the second-order type-2 PLL is always stable with these conditions. Negative ζ or ω_n would result in positive poles, making the system unstable.

2.7.2 Digital PLL

In digital sampled systems, stability exists if all the poles of the closed-loop transfer function lie within the unit circle of the z-plane. The s-plane imaginary axis corresponds to the z-plane unit circle, and the s-plane left-hand side corresponds to the inside of the unit circle. Generally, a sampled system can be unstable for increasing gain values, whereas a similar continuous system would be stable for all gain values [19]. This difference in behavior applies to the second-order type-2 PLL introduced above: for an analog PLL, it is stable for all gain values. However, the type-2 all-digital PLL implemented in this thesis will not be stable for all gain values. Therefore, more attention must be paid to ensure the stability of an all-digital PLL.

The poles for a typical type-2 PLL can be calculated by finding the roots of the characteristic equation of (28). The number of poles depends drastically from D, the number of delays in the system. To ease the analysis, it is assumed that $D = 1$.

$$\begin{aligned} & (1 - z^{-1})^2 + \kappa z^{-1}(1 - z^{-1} + \kappa_2) = 0 \\ \Rightarrow\ & 1 - 2z^{-1} + z^{-2} + \kappa z^{-1} - \kappa z^{-2} + \kappa\kappa_2 z^{-1} = 0 \\ \Rightarrow\ & z^2 + (\kappa(1 + \kappa_2) - 2)z + 1 - \kappa = 0 \\ \Rightarrow\ & z = \frac{2 - \kappa(1 + \kappa_2) \pm \sqrt{(\kappa(1 + \kappa_2) - 2)^2 - 4(1 - \kappa)}}{2} \\ \Rightarrow\ & z = 1 - \frac{\kappa}{2}(1 + \kappa_2) \pm \frac{\kappa}{2}\sqrt{(1 + \kappa_2)^2 - 4\kappa_2/\kappa} \end{aligned} \tag{34}$$

The poles will be real and separate if the discriminate is positive:

$$\begin{aligned} & (1 + \kappa_2)^2 - 4\kappa_2/\kappa > 0 \\ \Rightarrow\ & \kappa > \frac{4\kappa_2}{(1 + \kappa_2)^2} \end{aligned} \tag{35}$$

Whereas, they will be complex and separate if negative:

$$\begin{aligned} &(1+\kappa_2)^2 - 4\kappa_2/\kappa < 0 \\ \Rightarrow\ &\kappa < \frac{4\kappa_2}{(1+\kappa_2)^2} \end{aligned} \tag{36}$$

And, the poles will coincide if the discriminate is exactly zero:

$$\begin{aligned} &(1+\kappa_2)^2 - 4\kappa_2/\kappa = 0 \\ \Rightarrow\ &\kappa = \frac{4\kappa_2}{(1+\kappa_2)^2} \end{aligned} \tag{37}$$

A graphical method called root locus analysis can be deployed to find which parameter values result in the poles residing within the unit circle. In this method, one of the parameters, for example, κ, is kept constant, while the other parameter, κ_2, is changed. The poles are calculated and plotted on the z-plane for each new κ_2. Eventually, the result will be a graph with values inside and outside the unit circle. Then, κ can be changed slightly, and another graph can be drawn similarly. Finally, the values that result in stable behavior should be clear. An example root-locus plot from [7] is presented in Figure 5.

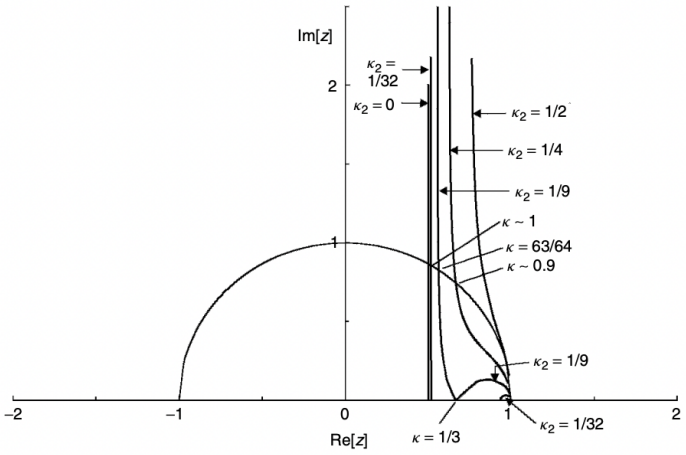


Figure 4.3 Root-locus plots of a type 2 DPLL with $D = 2$ and various κ_2. Real poles and the lower half plane are omitted.

Figure 5: Example of a root-locus plot [7]

Now, this Chapter has introduced the required PLL theory for constructing an analytical system model of the implemented all-digital PLL. The complete system will have a transfer function that largely resembles (28), from which loop parameters and stability boundaries can be derived. Then, these parameters will be used as design parameters. Additionally, the stability of the system will be observed with different design parameter values. The aim is to have a system model that presents the real system as accurately as possible.

3 Methodology

This chapter introduces the flow used for designing and verifying the design for the application of this thesis. The focus is on programmable logic, as the core design will be implemented on an FPGA using the VHSIC Hardware Description Language (VHDL), a programming language for describing programmable logic. However, a part of the PLL system and the software driver is designed using the Python programming language.

For programmable logic design and verification, many frameworks have been developed. The most widely used ones are proprietary tools provided by companies like Mentor Graphics, Intel, and Xilinx. Using these tools requires purchasing an often-costly license. Fortunately, the programmable logic design has recently benefited from the open-source movement, with community-developed frameworks that are free to use and whose source code is open for anyone to observe and modify. The design and verification flow described will consist mainly of these Free and Open-Source Software (FOSS) tools.

The flow is displayed in Figure 6 and will consist of the following steps: First, the programmable logic design is written according to the specification using a Hardware Description Language (HDL). Then, design verification is executed on a unit level. Next, the software part of the PLL is written, a system-level testbench is composed, and system tests are performed. Finally, the PLL is integrated into real hardware, in which integration tests are executed. In the following sections, these steps are described in further detail.

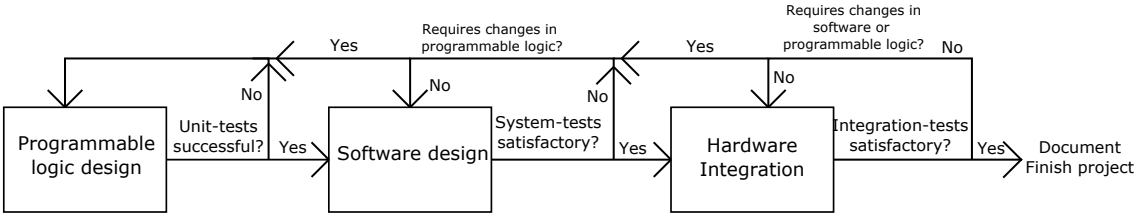


Figure 6: Design and verification flow

3.1 Programmable Logic Design

In this section, the process of designing programmable logic is explained. As displayed in Figure 6, it is the first part of the design flow. The design will eventually be uploaded on an FPGA in the integration step, but first, it will undergo rigorous tests to discover any bugs: finding and fixing them is much easier in the design phase than in the integration phase.

Describing a digital hardware system can be divided into three different representations: functional, structural, and geometric [20]. Here, the functional representation describes the system on an algorithmic level. In practice, this would be the code written by the designer in some HDL and possible higher-level abstractions and lower-level representations. This representation is common for both software and programmable

logic design. The structural representation explains how the functional model is transferred into a set of interconnected components, like logic gates, flip-flops, and registers. This transfer step is often called *synthesis*. The geometric representation transfers the structural representation into physical hardware by drawing transistors on silicon or utilizing the available resources on a programmable logic device like an FPGA. This step is often called *implementation* or *place-and-route*.

Nowadays, the functional representations of programmable logic designs are usually realized by writing the circuit description using a hardware description language (HDL). HDLs look similar to traditional programming languages, but fundamentally, they are different. Where traditional programming languages always get compiled to be executed on a general-purpose microprocessor eventually, an HDL program's final product is physical hardware that inherently executes concurrently. The hardware consists of interconnected transistors or higher-level systems, such as flip-flops, registers, and latches, that perform the functions described in the HDL. The inherent concurrency is what enables the enormous efficiency gains. For example, it is possible to program an FPGA to perform 1000 addition operations in parallel, whereas, on a generic microprocessor, the operations will always be executed sequentially.

Commonly, this level of development is referred to as *Register-Transfer Level* (RTL) [21], referring to the fact that the program describes how digital signals flow and get manipulated between hardware registers. Therefore, the designer must consider what type of structures result in an efficient hardware implementation. Usually, the most efficient structures are pretty different from efficient program structures for general-purpose microprocessors. For example, the for-loop is an efficient structure in traditional software to execute repetitive tasks, but using it will often bring trouble with programmable logic if not used properly. Moreover, some structures might not even be physically implementable.

The HDL used in this thesis is the VHSIC Hardware Description Language, also known as VHDL. It is one of the two most used HDLs, the other one being Verilog. The choice between the two HDLs is usually determined by the preferences and previous experiences of the design team. This thesis also uses VHDL for the same reason: it is the most commonly used HDL at CERN.

VHDL development starts by writing the first draft of the hardware system with a text editor of the designer's choice. Once the first version is drafted, the VHDL design is either *simulated* or *synthesized* [21]. Simulation has the same meaning as execution in traditional programming languages. However, as VHDL programs are meant to be, eventually, realized on hardware, the simulation part is more to ensure that the program functions as intended before uploading it onto real hardware. Hence, it is called simulation instead of execution. The synthesis part is unique to HDLs. It means building a netlist of interconnected hardware components, i.e., transferring the design from the functional description into the structural description. The end product from this process is then used to realize the design on hardware, i.e., transferring the structural description into the geometrical description.

Both operations require two preparatory steps: *analysis* and *elaboration*. These processes can also be described as *compiling* the design. Analysis verifies that

the VHDL code conforms to the syntactic and semantic rules of the specification. Additionally, the design is transferred into an intermediary format that is easier for the later parts to use. Next, the elaboration part constructs the final product where all design hierarchy has been unraveled into one flat design. This process replaces each submodule instantiation with their actual implementation until only processes interconnected with signals remain. [21]

A variety of tools exist that perform these VHDL-related tasks. This thesis utilizes an open-source VHDL compiler called *GHDL* (G Hardware Design Language) [22]. It operates only by calling it from the command line, i.e., it does not have a graphical interface like some proprietary tools do. However, its open-source nature has enabled it to be integrated into many other open-source design frameworks, such as libraries focused on efficiently creating environments for simulating the VHDL design. It has recently added support for synthesis, but this is still in an experimental state and, thus, is not used in this thesis.

3.1.1 Unit-tests

An essential step in the design process is verification. It broadly means inspecting and testing whether the RTL conforms to the specified requirements. The verification process often consumes over 70% of the whole design effort [23]. Failing in this task can be a costly mistake, as it can lead to unwanted behavior in the field. One of the most famous failures in the verification is attached to the Intel Pentium FDIV bug in 1994, where the processor would yield incorrect results for a division operation on certain occasions. Intel was forced to recall all affected units, leading to an extra cost of around 475 million US dollars. [24]

Verification is performed on all levels of a project, and it is crucial to understand the level when composing a verification plan. The levels described in [23] include the following, from bottom to the top:

- Designer-level
- Unit-level
- Core-level
- Chip-level
- System-level

Here, the designer-level is the lowest level that aims to verify the smallest RTL modules, such as an arbiter or a first-in-first-out buffer. Unit, core, and chip-level verification refers to verification on systems that consist of subsystems of different complexity. Finally, the system-level verification verifies the operation of the whole system.

This section focuses on the unit-level verifications. It is performed on complete units composed of many RTL modules that have already been verified on the designer level and have a bus interface for register access. These units usually have a

specification of how they are supposed to function, and, therefore, generating realistic test cases is possible. Ideally, the unit-level verification should aim for such high confidence that bugs related to the unit can be excluded from the list of candidates when errors are occurring on higher levels [23].

The most common method of conducting verification is *simulation-based verification*. It is performed by constructing a test environment called *testbench*. In this environment, the Device Under Verification (DUV) is instantiated as it would be in a real system. The testbench consists of three main components: *stimulus driver, monitor*, and *checker*. These components are illustrated in Figure 7. The stimulus driver creates stimuli signals to stimulate the DUV, including the clock, enable, and data signals. The monitor observes the internal signals and verifies that the interfacing signals adhere to the interface protocol. The final part of the testbench, the checker, compares the outputs to the expected values to see that the output signals are correct.

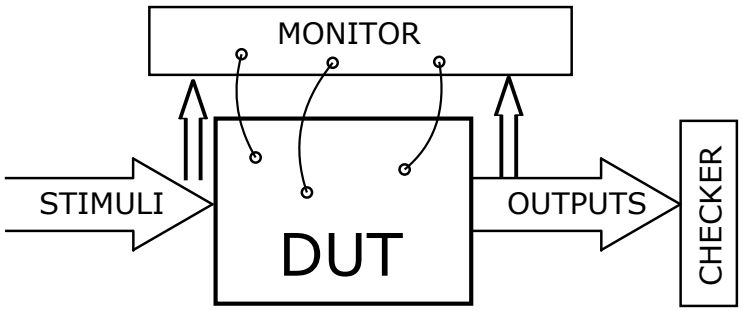


Figure 7: Testbench blocks

There are a few methods of creating a *self-checking* testbench, i.e., a testbench that automatically checks that the outputs values are as expected. The first one is called *golden vector checking*. Here, a list of inputs and correct outputs are manually crafted. An entry in this list is called a golden vector. The testbench is then made to read the inputs, drive the DUV, and compare the outputs with the entries in the golden vector. Another method is *checking-against-reference-model* where a reference model is constructed in a higher-level programming language. Then, the same inputs are applied to both the DUV and the reference model, and outputs are compared. Finally, there is *transaction-based* checking. In this method, inputs are generated in the form of transactions. A new testbench component is introduced: *scoreboard*. The scoreboard records the input commands and the expected output data. Then, the checker compares the actual outputs with the records queried from the scoreboard and issues an error if they do not match.

After determining the structure of the testbench, metrics for measuring the verification quality must be established. These metrics indicate how well the operation of the DUV has been verified. Some popular metrics include *code* and *branch coverage*. Code coverage indicates the share of the number of exercised lines of HDL code compared to the total number of lines of HDL code. The branch coverage evaluates the share of visited branches: a new branch is created every time the code has an if or case statement. These two metrics are called structural metrics, and they are

easy to define and measure. Another important metric, but a more ambiguous one, is the *functional coverage*. Here, the designer must determine the functions that the verification should cover. Then, tests should be created that simulate these functions and verify that the outputs are as expected.

Finally, the generation of stimuli must be defined. Frequently, it is impossible to test the DUV with all possible input combinations, as the input space can become so large that the simulation time of all cases would be years. As a remedy, two methods are commonly in use. First is *constrained random verification*. Here, the designer determines a range of values that the stimuli can take. Then, a program is crafted that automatically generates multiple test cases with randomly picked stimuli from the provided input range. The second method is *coverage driven verification*, in which the stimuli are manually decided to reach the highest possible coverage. Here, the designer must discover new scenarios to increase the coverage score. Further increasing the score can become non-trivial when the coverage starts approaching 100%.

Instead of manually composing a testbench from scratch, this thesis uses a verification framework called *VUnit* [13]. It provides an environment for creating automated self-checking test benches with any verification methodology. The framework's benefits can be exploited whether tests are written for short-running unit tests or long-running top-level tests, using directed or constrained random testing. In addition, the framework provides verification components, such as Wishbone or AXI-master and slave modules. With these components, a transaction-based testbench, in which the stimuli are generated by sending Wishbone transactions to the DUV, can be constructed.

The tests for a VUnit-testbench are written using VHDL, which is convenient for designers accustomed to writing test benches in that language. However, the test administration is handled with a Python file, which allows testing with different generics, continuing tests even after a fatal run-time error, and independent instantiation of each test case. Additionally, the Python-side of VUnit takes care of the correct compilation order of VHDL modules and allows running tests in parallel on multi-core machines. Finally, it presents the test results in a clear format on the terminal, generating a results file if needed. Therefore, VUnit can be seen to be an enhanced version of traditional testbenching verification.

3.1.2 Wishbone

Wishbone [12] is an open-source interconnection architecture to connect modules inside a chip. This thesis uses it to access the internal registers of the PLL module from the main FGC3.2 project. All modules in the project are interfaced similarly, which allows the software to access the whole FPGA memory as one memory space. The FGC3.2 FPGA project is displayed in Figure 8. The green and red lines correspond to the Wishbone data and address buses, respectively. There are two Wishbone masters on the left side of the figure: IFC master and UART master. The former is used when accessing the FPGA from the onboard microprocessor, whereas the latter is when accessing the FPGA from an external computer.

Each slave includes a memory map (MMAP) that maps the global Wishbone addresses to the local register locations. The memory map is generated for all modules with an open-source memory map tool *Cheby* [25]. In Cheby, the memory map is defined in a text file. For example, the properties to be defined include the bus type, register name, whether access is read or write, preset value, fields within the register, to name a few. Out of these properties, the Cheby tool generates the VHDL file that provides a Wishbone (or another, depending on the bus type given) access to the internal registers. If desired, Cheby can also generate constant files in VHDL, C, and Python, that provide constants for accessing specific memory addresses.

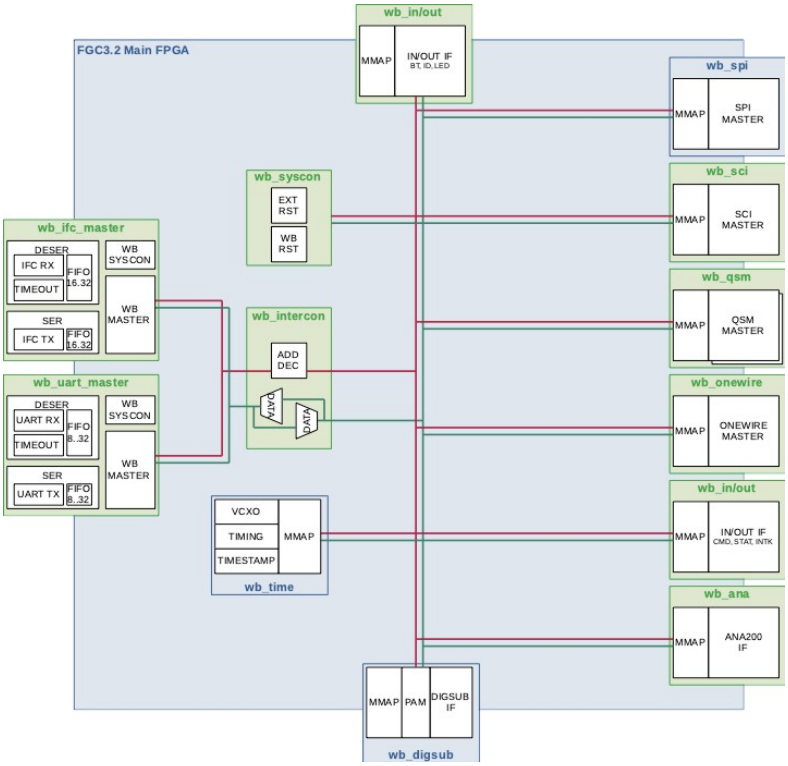


Figure 8: FGC3.2 FPGA project

3.2 Software Design

This thesis uses software to develop the loop filter, a system-level testbench, hardware integration tests, and a driver for the programmable logic. The software written for this thesis will not be used in production. Therefore, the process is not verified as rigorously as the programmable logic.

Python is an efficient language with broad support for freely available libraries and frameworks, making it an excellent tool for rapid prototyping. This thesis uses it for several components of the test setup. One of these components, the system-level test environment, is introduced next.

3.2.1 System-level simulation

A system-level simulation testbench is constructed to simulate the entire PLL before integrating everything on hardware. The testbench allows testing the effect of different loop parameters on various environmental conditions, including optimal environment, unideal VCXO, and jitter. The system-level simulation achieves a broader understanding of different factors, which is beneficial as the final hardware will only allow testing on one environment due to having only one hardware prototype.

This step can be considered system-level verification, as it tests the integration of programmable logic and software. However, no exhaustive automated test setup shall be implemented, as the designed software will not be used in production. The goal is to visually observe that the software and hardware communicate correctly and the PLL functions as a whole. Once this is assured, various sweeps can be performed that show the effect of different parameters on the PLL.

Few options allow constructing such a test environment with a high-level programming language like Python. Thankfully, there exists an open-source framework called *cocotb* that allows co-simulating VHDL and Python [14]. It enables manipulating signals in the VHDL module with Python, which, in turn, allows manipulating data in Python. As will become evident later, this allows realistically simulating the whole PLL system. However, as cocotb is not a simulator, it requires a VHDL simulator to call. The VHDL simulator used is, obviously, GHDL.

Cocotb operates by alternating between running Python and running the HDL simulator. The framework uses Python coroutines that are used to schedule concurrent execution of the test instances. The *await* keyword is used to indicate when control is passed back to either another coroutine or the HDL simulator. For example, it can be set to wait until a specific time passes or a signal changes in the simulator.

Once the test environment is established, the PLL locking process is documented by logging the phase error over time until the PLL reaches lock. Then, this process can be visualized by plotting it on a figure. After repeating the process with various parameters, the same phase error versus time graphs can be plotted on the same figure, showing the effects of different parameters quite clearly.

3.3 Hardware integration

Once the system-level simulations show that the PLL successfully predictably reaches lock, the VHDL design and Python software are uploaded on real hardware. The VHDL code is synthesized, implemented, and uploaded on a *field-programmable gate array* (FPGA). A Python driver is developed to enable rapid prototyping of software scripts for interfacing the FPGA.

3.3.1 Field-programmable Gate Arrays

Field-programmable gate arrays, or FPGAs, are devices capable of implementing custom digital circuits with high flexibility and low cost. The hardware configuration can be reprogrammed as many times as required, making them a much more flexible solution than fully-fledged application-specific integrated circuits, or ASICs, in which

the hardware circuits are printed on a silicon wafer and cannot be reconfigured afterward. FPGAs also allow rapid design iteration cycles, as they can be reprogrammed in a matter of seconds, whereas design cycles for an ASIC can be months long. Therefore, FPGAs provide a convenient and flexible platform for implementing custom hardware for products that are not mass-produced in the millions and need not be the absolute state-of-art in terms of performance. Additionally, they can be used for ASIC prototyping.

The FPGA consists of three main components: programmable logic elements, programmable I/O (input/output) elements, and programmable interconnect elements [26]. Programmable logic elements are commonly referred to as configurable logic blocks (CLBs), and they consist of look-up tables (LUTs), flip-flops (FFs), multiplexers, and arithmetic carry-logic cells. The LUTs are memory blocks that incorporate the truth table that performs the desired logic function, which means that for an LUT with k inputs, 2^k bits of memory are required. Nowadays, the LUTs tend to have six inputs, as this offers good trade-offs with both area and propagation delay. The FFs are used as memory elements in sequential circuits: they store the result of the arithmetic operation by the LUT and are ready to distribute it forward on the next rising clock edge. The carry logic circuits enable a high-speed carry signal propagation in arithmetic operations like addition. The multiplexer decides the output from the CLB, as the CLB may have several potential outputs.

Alongside the FFs, FPGAs also include dedicated memory elements. They are called block random access memory, or BRAM. One BRAM element has a fixed memory size, commonly 36 kilobits, but multiple BRAM blocks can be configured to act as one sizeable memory element. The BRAM can be configured to function as dual-port memory, allowing two simultaneous memory accesses: this enables using one BRAM block as FIFO (first-in-first-out) memory, a fundamental building block in digital systems. Furthermore, some LUTs can also be utilized as memory, if necessary. In this case, the memory elements are often called distributed RAM. One benefit, compared to BRAM, is that distributed RAM can be read and written asynchronously, whereas BRAM access is always synchronous, i.e., triggered by a rising clock edge. Therefore, the access to distributed RAM is one clock cycle faster than BRAM.

Programmable interconnect elements are used to connect CLBs with other elements on the FPGA. Nowadays, the connections are often made in an island-style, in which a channel of wires is laid between all CLBs in vertical and horizontal directions. Here, the programmable interconnect elements are used to connect the CLBs to the wires with input and output connection blocks (CBs) and the horizontal wires to vertical wires with switch blocks (SBs). CBs and SBs are often designed not to connect all wires. This is to save area, allowing for more CLBs and other elements. However, routing between CLBs may become impossible if there are too few connections.

The programmable I/O elements are placed along the periphery of the FPGA chip. One of the elements, an I/O block, is placed between a CLB and an I/O pad, i.e., the connecting wire to the outer world. The I/O block can be configured to interface with the outer world in various ways. For example, it can be configured to

act as an output buffer, handle differential signals, or enable a pull-up or pull-down resistor. It can have dedicated hardware to handle different communication protocols, such as the high-speed LVDS protocol that has a data rate of gigabits per second, a data rate that the standard CLBs cannot handle by themselves. Nowadays, as FPGAs support an ever-increasing number of communication protocols, the I/O blocks are separated into I/O banks, which means that not all I/O blocks are capable of handling all protocols supported by the FPGA.

Contemporary FPGAs also incorporate additional specific hardware for some generic operations. For example, a DSP (digital signal processing) block includes complicated hardware to enable fast multiplication and accumulator operations for mathematical operations like signal filtering. Additional blocks include the clock management tiles (CMTs), which enable high precision and low jitter frequency synthesis, and analog-to-digital converters. The purpose of all these extra blocks is to make the FPGA more versatile while keeping the possibility of defining custom hardware.

The FPGA used in this thesis is Xilinx Artix-7 XC7A50T, because it is the FPGA used in the FGC3.2 project. It includes 8150 CLBs, 2700 kilobits of BRAM, 120 DSP slices, 5 CMTs, 250 I/O blocks, and one ADC block. A summary of the resources is provided in table 1. Only a small part of the resources can be occupied by the PLL of this thesis, as the FPGA has multiple other functionalities in the FGC3.2 project.

Part	Logic Cells	CLBs	DSP Slices	BRAM	CMTs	PCIe	GTPs	XADC Blocks	I/O Banks	Max User I/O
XC7A50T	52,160	8,150	120	2,700 (kb)	5	1	4	1	5	250

Table 1: Specification sheet for XC7A50T [27]

3.3.2 Synthesis, Implementation, and Uploading the Bitstream

Synthesis was partly touched on previously in this chapter. In short, it means translating the functional description in VHDL to a structural description. In practice, synthesis means transferring the HDL representation into gate-level schematics, i.e., into basic digital logic blocks such as AND, OR, and NOT gates, or even some macro-level gates like adders, multiplexers, and flip-flops. This process includes many optimization steps to minimize the number of elements used. This thesis will use a proprietary tool called *Vivado Design Suite* by Xilinx for the synthesis and implementation.

In synthesis, the HDL source code is mapped and optimized for the target technology to produce a gate-level netlist. Alongside the HDL source code, a constraints file should be provided. From a synthesis perspective, the constraints file should include the timing constraints for the project, i.e., the frequency requirements. Then, the tool optimizes the design to meet these timing requirements. If a constraints file is not provided, the design is optimized only for wire length and placement congestion.

The implementation step transfers the structural description into a geometrical description. This step is also called place and route. Here, a constraints file is

essential to provide the physical constraints for the FPGA implementation. For example, the constraints file should indicate which FPGA pin should be connected to which signal in the HDL code. Also, physical properties like the I/O block type and voltage levels are set here. Once the constraints are provided, the implementation can begin. First, the netlist generated by the previous step is optimized. Then, it is placed onto the target device, and optimizations to stay within timing constraints are performed. Finally, the blocks are routed to allow signals to propagate. After a successful implementation, a bitstream can be generated: the file to be uploaded to the FPGA.

Then, the bitstream can be uploaded to the FPGA by using the Hardware Manager provided by the Vivado Design Suite. If the FPGA is connected to the computer with a JTAG-programmer, the target device should be visible to establish a connection. Once this is successfully done, the bitstream can be uploaded to the target FPGA.

3.3.3 Python Drivers

To access the FPGA registers, a Python driver is designed. The driver becomes the final part of the chain of multiple components, whose purpose is to enable easy access to internal registers of the FPGA from a microprocessor domain, be it either a desktop test computer or a microcontroller on the same board. This chain of components is displayed in Figure 9: the components implemented in this thesis are bordered with red color. The figure displays the flow for the setup where the registers are interfaced from an external computer. Essentially, the PLL-specific commands are broken down into UART transactions sent in ASCII format via a UART cable. The transactions are then reconstructed into Wishbone transactions on the FPGA.

This approach allows easy and rapid prototyping of the separate VHDL modules on the FPGA. Naturally, the project's top-level must be in place, including the required modules for reconstructing Wishbone transactions from UART transactions, so the project must be built with a top-down approach.

Furthermore, this approach allows the driver class to be completely interface-independent. A different interface can be used by replacing uartWbSyscon and pySerial with the corresponding modules for another interface. As long as the replacement for uartWbSyscon provides read() and write() functions, the same wb_pll-driver can be used by simply instantiating it with a different bus instance. For example, the onboard microcontroller will have a high-speed IFC interface to the FPGA: the same Python driver can be used once the two modules have been replaced with IFC-compatible ones.

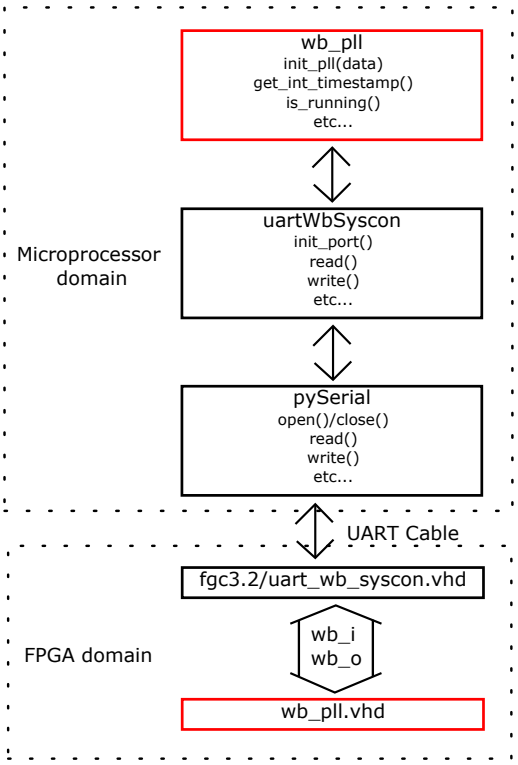


Figure 9: Flowchart from Python-driver to VHDL module. The red ones are implemented in this thesis.

4 Design Process and Results

In this chapter, the design flow is executed by using the methods described in the previous chapter. First, the specifications for the implementation are introduced. Then, the hardware properties for the VCXO and the DAC are presented. Next, the system is analyzed analytically by deriving a transfer function, finding its poles, and performing a stability analysis.

Then, the RTL design is introduced on a high level. The purpose is not to introduce the entire codebase, as it is thousands of lines of code, but rather to show the functionality by using simple logic blocks and abstractions where suitable. Then, the process of conducting unit tests is explained. The testbench is described, and all test cases are covered on a high level. Finally, the coverage metrics are presented for unit tests.

Next, the prototype software for the PLL is explained. Then, the system-level simulation test bench is presented. The first results for the complete PLL appear from tests conducted on this testbench: plots and figures are presented.

The final part presents the hardware test setup and conducts the final set of tests. Tests are similar to the previous section, and results are expected to follow the simulation results. Finally, this Chapter presents a discussion on the results.

4.1 Specification

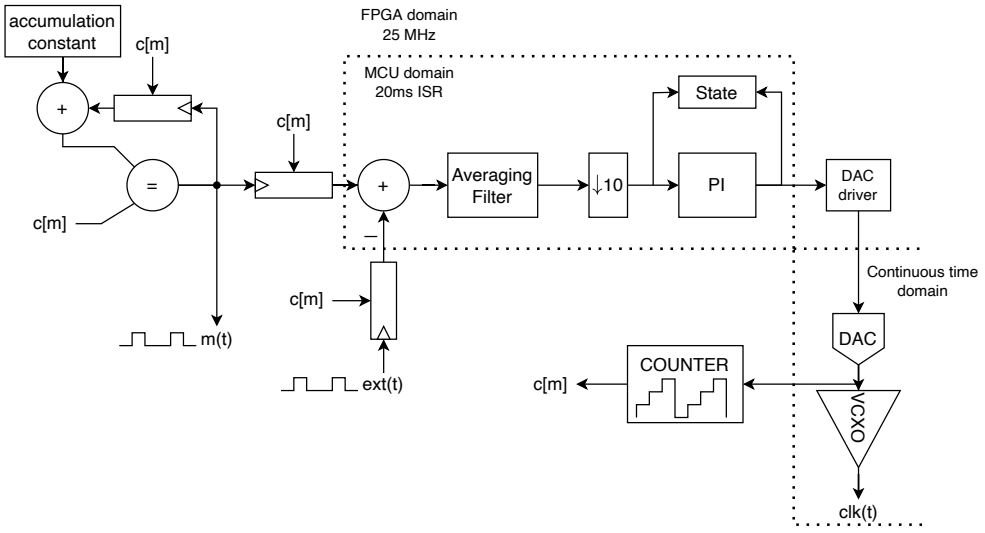


Figure 10: Block diagram of the entire PLL

This section provides the top-level block diagram for the PLL, and also the detailed explanation of how the overall synchronization scheme works. The synchronization scheme and the PLL structure remain mostly the same as in the previous implementation, FGC3.1, and thus, are not the results of this thesis. The main contributions of this thesis will be provided in later sections, in which the RTL design and different verification steps are presented.

Data transmission and time synchronization in FGC3.1 and FGC3.2 are achieved with an Ethernet-based fieldbus called FGC_Ether, which is described in [10]. It consists of a front-end computer which is connected to up to 64 FGCs via an Ethernet switch and a pulse generator. The latter generates low-jitter 50 Hz synchronization pulses that are fanned out to all connected FGCs. The Ethernet cable used to connect the FGCs use one pair of wires for the data transmission and one pair for the distribution of the 50 Hz pulse. Therefore, both data and time synchronization are distributed using only one cable.

The main objective for the PLL is to synchronize the FGC internal 20 ms program cycle with the 50 Hz synchronization pulse. Once all FGCs have been synchronized, their program cycles are aligned in very high accuracy, and they can initiate actions simultaneously. In addition, the front-end computer distributes a UTC time packet every 20 milliseconds. When the offset between a received time packet and a received synchronization pulse is known, the UTC time can be reconstructed on each FGC very accurately, allowing accurate timestamping of events, such as faults. The arrival of these time packets can also be used to generate an interrupt signal, which acts as a backup synchronization pulse when the primary 50 Hz pulse is absent.

The PLL structure and algorithm for FGC3.1 is provided in [11]. The structure remains the same for FGC3.2, and thus, is presented also here in Figure 10. Notably, there are three different time domains. First, the FPGA domain which operates in 25 MHz clock frequency, which is the system clock frequency for the PLL. This domain operates a counter and latches to store timestamps upon the arrival of the synchronization pulse $ext(t)$, achieving the functionality of the phase detector. Furthermore, it generates the internal 50 Hz tick $m(t)$. Second, the MCU domain executes once every 20 milliseconds. This domain is also called the software domain, and it performs the functionality of the loop filter. Finally, there is the continuous-time domain, in which the DAC drives the VCXO with an analog voltage to generate a controllable clock signal of 25 MHz, $clk(t)$.

In [10], the FPGA domain is presented in more detail. This structure is used in FGC3.1. The same register structure will also be used in this thesis to minimize required changes in software. Main changes in the programmable logic will be in the memory interface, which is invisible in the structure presented in [10]. In FGC3.1, the memory is addressed with a custom memory interface with hard-coded memory addresses, which is sub-optimal for reuse purposes. In this thesis, the memory interface is replaced with a Wishbone interface, which improves reuse and even allows a possible replacement of the memory interface without touching the rest of the module. Additionally, some parameters are added to allow instantiating the PLL with different characteristics, such as the clock frequency and synchronization pulse frequency.

In every software loop, the timestamp registers for the internal 50 Hz tick and external 50 Hz pulse are read and subtracted. The result is the phase error in clock cycles. Ten phase errors are averaged before forwarding the average to the PI-controller. Therefore, the PI-algorithm executes every 2 milliseconds, resulting in a PLL sampling rate of 5 Hz. The PI-controller then calculates how much the VCXO center frequency should be shifted, and sends this value to a DAC driver module.

The generated internal 50 Hz tick is then used to drive a tick generator that generates ticks with different frequencies, such as 10 kHz and 1 kHz. These ticks act as triggers for, for example, the start of a real-time task on the MCU, and other components on the FGC, such as ADC and DAC.

In [11], also the fundamental requirements for the whole synchronization scheme are stated. The requirements include a time precision of 1 µs and jitter of under 100 ns. These requirements are for the combined performance of FGC_Ether and the PLL; therefore, the PLL itself must surpass them. Sufficient performance is reached with the low-jitter 50 Hz synchronization pulse. Using the reserve method, the arrival of time packets, achieves a time accuracy of 20 µs.

In the same paper, the open-loop transfer function is presented. However, it has not been derived, and the PD and VCO gains have not been explained. Also, some test results are presented, including the lock time (2-3 seconds) and the phase drift after losing both synchronization pulses for multiple hours (1 µs/s). These results are the comparison point for the results later in this chapter.

4.2 Hardware Properties

There are two crucial external components in the PLL, namely the digital-to-analog converter (DAC) and the VCXO. Their properties determine some of the loop parameters.

Properties for the DAC are presented in Table 2. It is a 16-bit DAC, meaning that the output can have 2^{16} different values. The resolution is an improvement over the DAC in FGC3.1, which was only 14-bit, meaning four times worse resolution. The DAC is programmed to use its internal reference voltage that is fixed to 2.5 volts. The reference voltage determines the highest possible output value. Therefore, the output range is 0 to 2.5 volts.

Manufacturer	Part	Resolution	Output Range
Texas Instruments	DAC80501	16 bits	0-2.5 V

Table 2: DAC properties

VCXO properties are presented in Table 3. The absolute pull range is given as ±100 ppm, which means that the frequency can always be pulled ±100 ppm from 25 MHz in all temperature and aging conditions. In reality, the VCXO can be pulled even more. Figure 11 displays the characteristic curve for the VCXO used in hardware testing, generated by measuring the frequency with an accurate frequency counter while changing the DAC output. It can be seen that the VCXO can be pulled almost 250 ppm from the center frequency of 25 MHz.

Additionally, the VCO gain is given. This value describes how many ppm the frequency is shifted for each change in input voltage. Figure 11 proves that the given number is relatively accurate.

By observing the values given and Fig. 11, the DAC output range and VCXO control range do not quite match in this hardware configuration. The DAC cannot

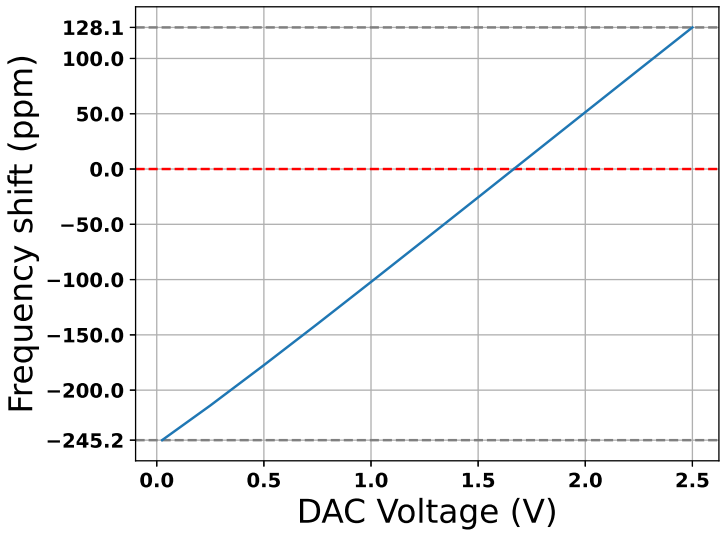


Figure 11: VCXO characteristic curve

Manufacturer	Part	Center Freq.	Temp. Stability	APR @ 3.3 V	Control Voltage Range	Kv (ppm/V)
Silicon Labs	Si515	25 MHz	20 ppm	± 100 ppm	$0.1V_{dd}$ - $0.9V_{dd}$	±150

Table 3: VCXO properties

reach $0.9V_{dd}$, which accounts to approximately 3 V when $V_{dd} = 3.3$ V. Also, the bottom range of the DAC is wasted, as values $0 - 0.1V_{dd}$ (approx. $0 - 0.3$ V) do not belong to the VCXO control range. The DAC reference voltage should be set somewhere around $3 - 3.3$ V in a more efficient configuration. Moreover, the asymmetry of the DAC and VCXO range centers can cause problems with the PI-controller, which will be shown later.

4.3 System analysis

In this section, the whole PLL system is analyzed analytically. This means composing the closed-loop transfer function, deriving the system properties, and determining the stability by finding the transfer function poles. Block diagram for the complete system is drawn in Figure 12.

First, calculating the open-loop transfer function $G(z) = \frac{\theta_o}{\theta_e}$:

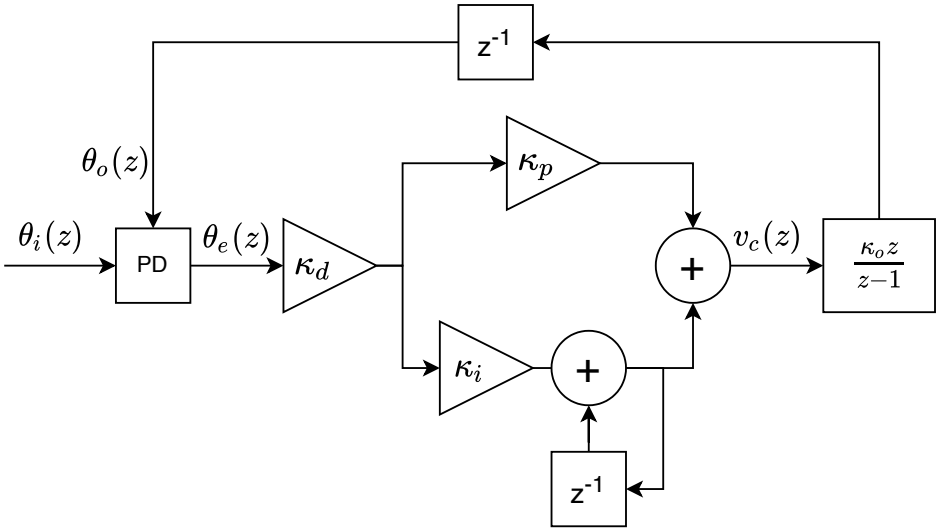


Figure 12: Block diagram of the entire PLL

$$
\begin{aligned}
\theta_o &= z^{-1}\frac{\kappa_o z}{z-1}\Big(\kappa_p + \frac{\kappa_i}{1-z^{-1}}\Big)\kappa_d\theta_e \\
\Rightarrow \frac{\theta_o}{\theta_e} &= \frac{\kappa_o\kappa_d}{z-1}\Big(\frac{\kappa_p(z-1)}{z-1} + \frac{\kappa_i z}{z-1}\Big) \\
&= \frac{\kappa_o\kappa_d}{(z-1)^2}\Big(\kappa_p z - \kappa_p + \kappa_i z\Big) \\
&= \frac{\kappa_o\kappa_d(\kappa_i+\kappa_p)}{(z-1)^2}\Big(z - \frac{\kappa_p}{\kappa_i+\kappa_p}\Big)
\end{aligned} \tag{38}
$$

The result for $G(z)$ is the same as in [11], except for the addition of PD gain κ_d. From $G(z)$, the closed-loop transfer function $H(z)$ can be calculated:

$$
\begin{aligned}
H(z) &= \frac{G(z)}{1+G(z)} = \frac{\frac{\kappa_o\kappa_d(\kappa_i+\kappa_p)}{(z-1)^2}\Big(z - \frac{\kappa_p}{\kappa_i+\kappa_p}\Big)}{1 + \frac{\kappa_o\kappa_d(\kappa_i+\kappa_p)}{(z-1)^2}\Big(z - \frac{\kappa_p}{\kappa_i+\kappa_p}\Big)} \\
H(z) &= \frac{\kappa_o\kappa_d\kappa_p z^{-1}(1 - z^{-1} + \kappa_i/\kappa_p)}{(1-z^{-1})^2 + \kappa_o\kappa_d\kappa_p z^{-1}(1 - z^{-1} + \kappa_i/\kappa_p)}
\end{aligned} \tag{39}
$$

By comparing to (28), it is obvious that $\kappa = \kappa_o\kappa_d\kappa_p$, $\kappa_2 = \kappa_i/\kappa_p$, and $D = 1$. This means that there is only one delay in the loop, which is the minimum for a digital system. This delay comes from the fact that on a rising clock edge, the PD calculates the phase difference $\theta_i[n] - \theta_o[n]$, and when the software calculates the new value for the VCXO, the effect, i.e. the shifted phase, needs one clock cycle occur.

Now, the full system properties given in (31) can be derived:

$$\omega_n = \frac{1}{t_s}\sqrt{\kappa\kappa_2} = \frac{1}{t_s}\sqrt{\kappa_o\kappa_d\kappa_i} \tag{40}$$

$$\zeta = \frac{1}{2}\sqrt{\frac{\kappa}{\kappa_2}} = \frac{1}{2}\sqrt{\frac{\kappa_o\kappa_d\kappa_p^2}{\kappa_i}} = \frac{\kappa_p}{2}\sqrt{\frac{\kappa_o\kappa_d}{\kappa_i}} \tag{41}$$

Furthermore, K and τ_2 can be derived:

$$K = 2\zeta\omega_n = \kappa_p\sqrt{\kappa_o\kappa_d/\kappa_i}\frac{1}{t_s}\sqrt{\kappa_o\kappa_d\kappa_i} = \frac{\kappa_p\kappa_o\kappa_d}{t_s} \tag{42}$$

$$\tau_2 = \frac{2\zeta}{\omega_n} = \frac{\kappa_p\sqrt{\kappa_o\kappa_d/\kappa_i}}{\frac{1}{t_s}\sqrt{\kappa_o\kappa_d\kappa_i}} = \frac{\kappa_p t_s}{\kappa_i} \tag{43}$$

The PD gain, κ_d, is derived by transferring the phase difference from radians to clock cycles and further constraining the phase difference to be declared as ppm. The first transformation occurs naturally, as the phase difference is counted as clock cycles between two rising edges with a hardware counter. The second transformation from clock cycles to ppm's is performed in software. Using ppm's allows the software algorithm, specifically the PI-controller constants, to be independent of the clock frequency used for the hardware counter, which improves reusability. Therefore, κ_d can be calculated as follows:

$$\kappa_d = \frac{500000}{2\pi}\left(\frac{\text{cycles}}{\text{rad}}\right)\cdot\frac{10^6}{500000}\left(\frac{\text{ppm}}{\text{cycles}}\right) = \frac{10^6}{2\pi}\left(\frac{\text{ppm}}{\text{rad}}\right) = 159155 \tag{44}$$

The VCO gain κ_o consists of multiple steps that transform the PI-controller output to an angular frequency. These steps are displayed in Figure 13. First, the value received from the PI-controller is converted into a binary value that corresponds to the DAC resolution, which in this case is 16 bits. Next, the DAC converts this value into a voltage ranging from 0 to 2.5 V. Then, this voltage transforms into a frequency shift of 150 ppm per volt. The result is multiplied by 2π to convert hertz to radians. Finally, to convert the frequency into a phase shift per cycle, the result is multiplied by the clock cycle. The output has a unit of rad/cycle, and it expresses how much the output phase will shift in one clock cycle.

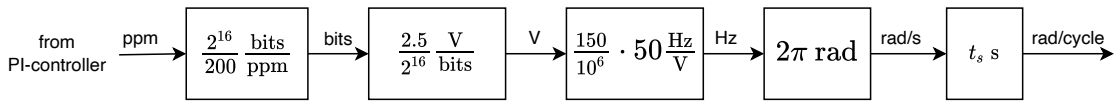


Figure 13: Block diagram of the composition of VCO gain

Thus, the VCO gain can be calculated as a combination of all steps:

$$\kappa_o = \frac{2^{16}}{200}\cdot\frac{2.5}{2^{16}}\cdot\frac{150\cdot 50}{10^6}\cdot 2\pi\cdot t_s = \frac{2\pi\cdot 2.5\cdot 150\cdot 50\cdot 0.2}{200\cdot 10^6} = 0.00011781 \tag{45}$$

To study stability, pole locations are critical. The system is stable if the poles are located within the unit circle, as previously explained. Here, the poles can be found by substituting the discovered values to (34):

$$
\begin{aligned}
z &= 1 - \frac{\kappa_o \kappa_d \kappa_p}{2}(1 + \kappa_i/\kappa_p) \pm \frac{\kappa_o \kappa_d \kappa_p}{2}\sqrt{(1 + \kappa_i/\kappa_p)^2 - 4\kappa_i/(\kappa_o \kappa_d)} \\
&= 1 - \frac{18.75\kappa_p}{2}(1 + \kappa_i/\kappa_p) \pm \frac{18.75\kappa_p}{2}\sqrt{(1 + \kappa_i/\kappa_p)^2 - 4\kappa_i/18.75} \qquad (46) \\
&= 1 - \frac{18.75\kappa_p}{2}(1 + t_s/\tau_2) \pm \frac{18.75\kappa_p}{2}\sqrt{(1 + t_s/\tau_2)^2 - 4t_s\kappa_p/(18.75\tau_2)} \qquad (47)
\end{aligned}
$$

From (47), stability analysis can be conducted by drawing root-locus plots. Results are shown in Figure 14. The unit circle, the border for stability, is drawn in a dashed black line in the figure. The other pole is drawn in red color, whereas the other pole is drawn in blue color. τ_2 is held constant, while κ_p is swept from 0.02 to 0.15. The pole values drawn in blue all remain within the unit circle, but the red values can drift outside. Also, all poles are purely real, which means the discriminant of (47) is positive. Evidently, the lower τ_2, the lower the κ_p can be. Also, when increasing τ_2, the limiting value does not increase much further. For $\tau_2 = 3$, the limiting value is around 0.1, and for $\tau_2 = 50$ the value is only slightly larger at around 0.11. Therefore, the usable values for κ_p lie below this value.

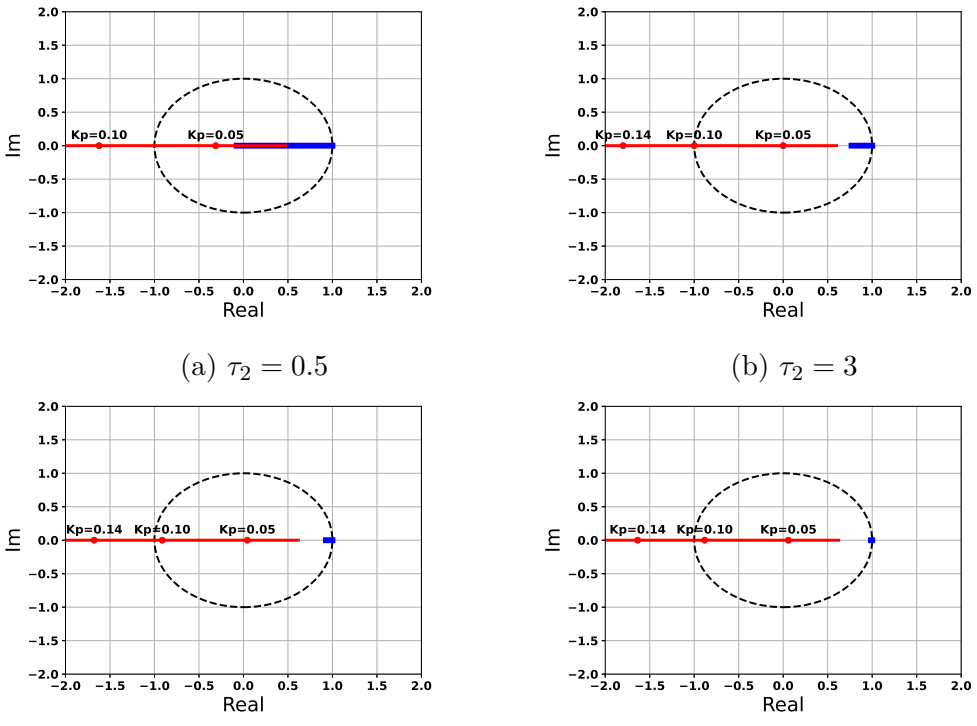
In conclusion, the suitable values for κ_p and τ_2 should lie somewhere around < 0.1 and > 3, respectively. The more the values deviate from these two values, the further from instability the system is. However, there are some trade-offs regarding lock-time and jitter filtering that enforce using values close to the stability boundaries. This will become evident in later sections.

4.4 RTL Design

The part implemented on the FPGA is the phase detector. The phase detector is counter-based, in which a counter calculates the number of clock cycles between the internally generated 50 Hz pulse and the received external 50 Hz pulse. Naturally, the objective of the entire PLL is to narrow this difference down to zero. The register structure of the PD is shown in Figure 15, and it is very similar to the previous implementation, visible in [10], to minimize required software changes.

However, even if the register structure is similar, the whole module needs to be re-designed to support the new Wishbone interface. At the same time, the module is parametrized to support instantiating the module with different parameters, such as different clock frequencies. The FGC3.1 implementation was fixed to operate on a single clock frequency.

This thesis implements the FPGA domain registers indicated in Figure 15, except for the two DAC modules with dashed lines. There are three latching registers: INT_SYNC_TIME, referring to internal sync time, which latches when the 20-bit counter reaches its value, and PRIM_SYNC_TIME and RES_SYNC_TIME, which refer to primary and reserve sync time, that latch the counter value once an external



(a) $\tau_2 = 0.5$ (b) $\tau_2 = 3$

(c) $\tau_2 = 10$ (d) $\tau_2 = 50$

Figure 14: Pole locations with different τ_2 values.

sync signal (external pulse or network interrupt) is received. All registers, including the counter value, can be read by software.

The top-level of the RTL developed in this thesis is presented in Figure 16. Most of the phase detector functionality is inside pll_top, and its precise operation will be presented later in this section. Pll_regs is a module, generated with Cheby, that forms the Wishbone interface. Pll_top connects the signals, that should be accessable by software, to pll_regs. Wb_pll is a wrapper module that contains these two modules. It also includes the logic to detect a read-access by software by setting the OLD flag in a timestamp register when this occurs.

Next follows the detailed description of the implemented RTL design inside the pll_top module, and the OLD flag logic in wb_pll.

The internal pulse is initiated when the reset-bit in the INT_SYNC_TIME_LOAD register is cleared. At reset, the bit is set. This will also load the contents of that register into INT_SYNC_TIME. This functionality allows initializing the internal pulse very close to the external pulse, which decreases the lock time by multiple seconds. More on this process is presented in the software section. Once the register has been loaded, it cannot be reloaded: the only way to load new data is to reset the module and repeat the process.

Figure 17 displays how the internal sync time register is updated. At the center is the synchronous accumulator (the circle with a plus in Fig. 15) module that has three inputs. The input_i port provides the input signal, which can have two effects:

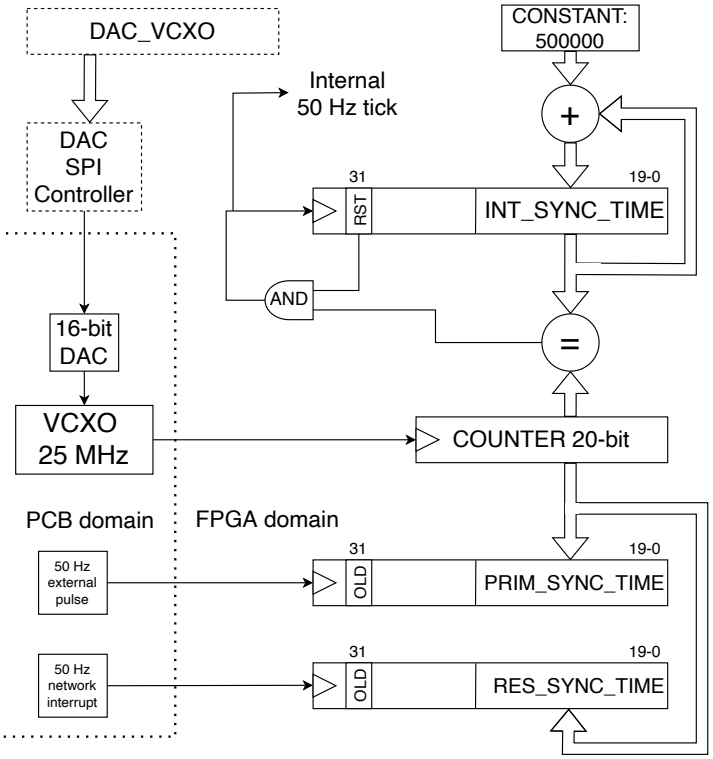


Figure 15: Internal register structure

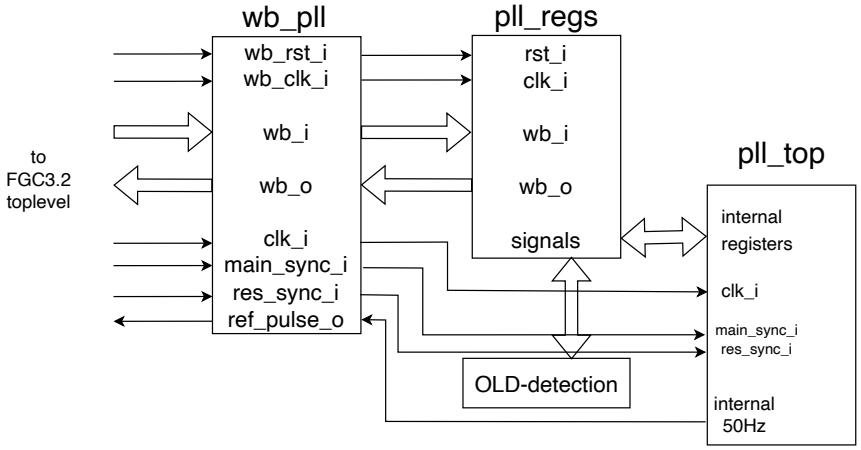


Figure 16: Top-level structure

When the load_i and en_i signals are simultaneously triggered, the module loads the internal register with the signal in input_i. When only en_i is triggered, the module accumulates the internal register with the signal in input_i. The internal register is always tied to the output port sum_o that is directly connected to the int_sync_time register. Now, the operation goes as follows: When the PLL is enabled, a start pulse is generated: this triggers load_i and en_i and selects the load value from the multiplexer. Once the PLL is running, started-signal is set high. Now, the accumulator is enabled only when comp_out is high. This is the output from

the comparator that compares int_sync_time with the internal counter value: this process is presented later. The accumulation constant is defined as 20 milliseconds (one 50 Hz period) in clock cycles, i.e., int_sync_time will progress by one 50 Hz cycle when the accumulator is triggered.

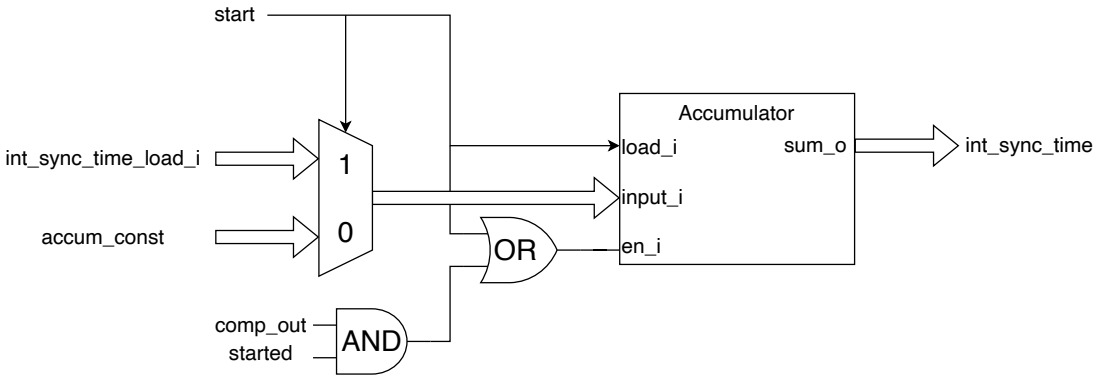


Figure 17: Accumulating internal sync time register

The operations of the internal free-running counter and the comparator are displayed in Figure 18. The free-running counter is constantly running and is zeroed when the PLL is reset. The value of this counter is compared to the internal sync time, and once they match, a pulse is sent that accumulates that sync time, as previously explained.

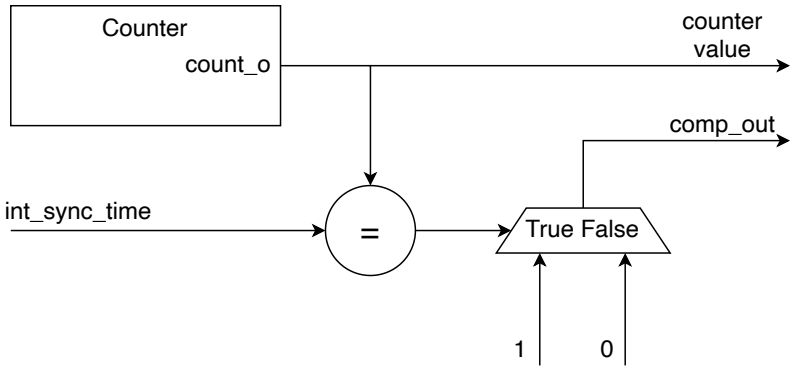


Figure 18: Comparator and free-running counter operation

Figure 19 shows the generation of the start pulse and the started flag. Once an enable-signal is received, a start-pulse is generated, and the started-flag is raised. This is achieved by instantiating two edge detectors, one configured to generate a pulse and one to hold the output high. Once the module has been once enabled, it cannot be re-enabled, i.e., the start-pulse can be generated only once. Thus, the enable process is edge-triggered, and only a pulse is required to enable the module. The incoming enable signal is synchronized with an external inputs-module, in which the signal propagates through two flip-flops to settle possible metastable states because it comes from a possibly different clock domain.

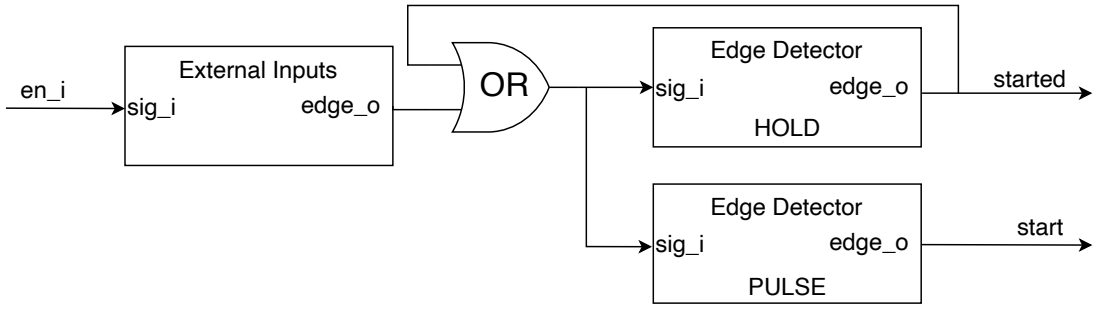


Figure 19: Process to generate start pulse and started flag

Figure 20 displays the generation of the enable-signal. A pulse is generated when a write-access by software triggers a write-strobe, and, simultaneously, the reset-bit in the load-register is cleared. The register and the strobe work in a different clock frequency than the rest of the PLL, and thus, there is a pulse extender to ensure that the enable-pulse is long enough to meet a rising clock edge on the other clock domain.

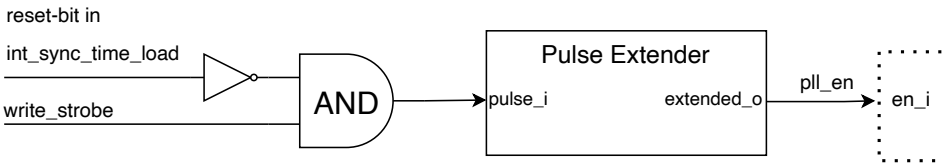


Figure 20: PLL enabling process

Figure 21 shows how the timestamps are latched into the two (primary and reserve) sync_time registers. Upon incoming external pulse, the pulse is first synchronized into the local clock domain with the external inputs module. After that, the pulse travels to an edge detector that generates a pulse of one clock cycle, as the length of the external pulse itself can be arbitrary. The output of the edge detector drives a synchronous multiplexer that drives the specific sync_time register with either its previous value or the value of the free-running counter. This functionality is wrapped into a submodule called sync_latch, and two of them are instantiated in the project.

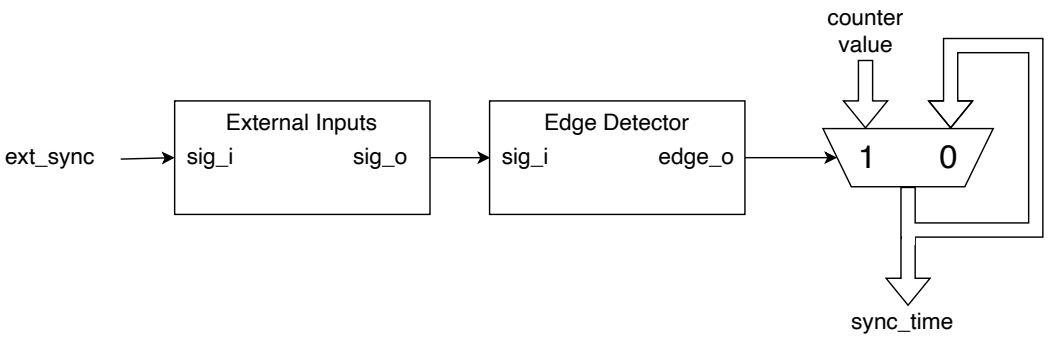


Figure 21: Synchrnonizing external pulse and latching the counter value

In Figure 22, the process for setting and clearing old-flags are shown. The old-flags are located in reserve and primary timestamp registers, and they indicate whether the register value has already been read by software. The figure also shows how the sync_time data flows from the PLL internal register into the Wishbone register sync_time_reg that is interfaceable with Wishbone transactions. The edge detector input read_strobe comes from the Wishbone interface and indicates when the software has made a read request, which the edge detector detects by setting the output, old_flag, high. This edge detector is configured so that the output stays high until an acknowledgment is sent to ack_i. This signal is triggered when a new value is latched to sync_time (the same signal as in Fig. 21). Finally, there are two delay steps from sync_time to sync_time_reg – the register that the software can read. The delay steps ensure that the old-flag and the new timestamp are written simultaneously.

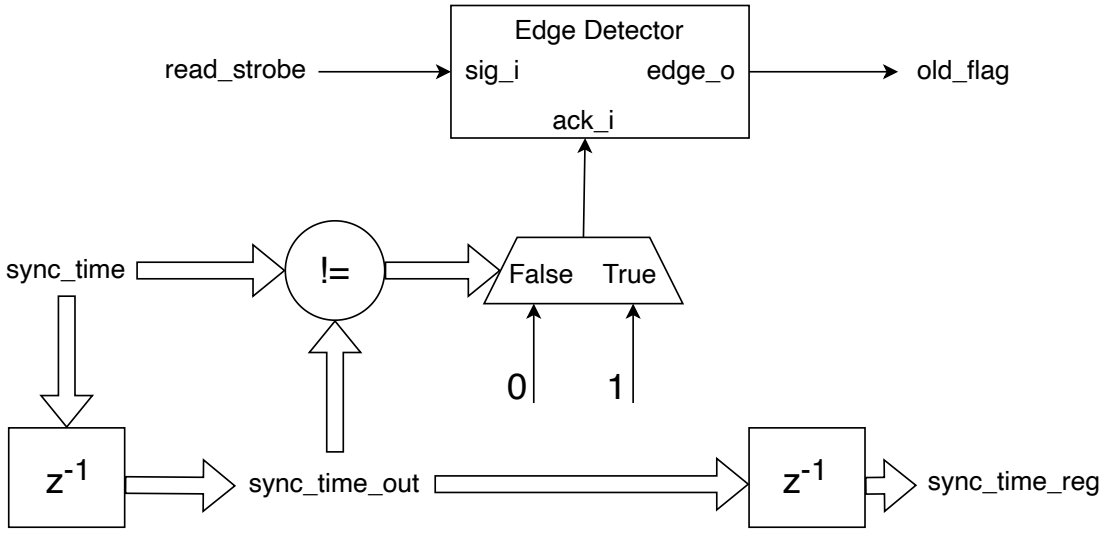


Figure 22: Process to set OLD-flags

Table 4 provides the register structure for accessing RTL registers. The leftmost column indicates the register's name, while the horizontal column indicates the bit location. The remaining slots indicate the function of the bit or bits and whether the register can only be read (ro), written (wo), or both read and written (rw) by the software.

The first register, int_sync_time_load, is utilized to set the initial value for int_sync_time and start the module by setting the rst-bit low. The following three registers have been discussed previously, the internal sync time register and the two timestamp registers. Control-register includes only one bit that allows resetting the module. Finally, there are two status registers, in which the first one stores the counter width and the counter value, and the second one stores the accumulation constant. The software algorithm requires counter width and accumulation constant, so it is convenient to provide it in registers.

The introduced memory map program, Cheby, can automatically generate the VHDL module to handle the Wishbone interactions and provide constants files

	31	30-24	23-0
int_sync_time_load	rst (rw)	unused	value (rw)

	31	30-24	23-0
int_sync_time	rst (ro)	unused	value (ro)

	31	30-24	23-0
prim_sync_time	old (ro)	unused	value (ro)

	31	30-24	23-0
res_sync_time	old (ro)	unused	value (ro)

	31	30-0
Control	reset (wo)	unused

	31-29	28-24	23-0
Status	unused	counter_width (ro)	counter_value (ro)

	31-24	23-0
Status_b	unused	accumulation_constant (ro)

Table 4: Register structure

for interfacing the registers with software. In Code 1 is an extract of the Python constants file generated for the register structure in Table 4.

```
PLL_REGS_SIZE = 28
# ...
ADDR_PLL_REGS_INT_SYNC_TIME = 0x4
PLL_REGS_INT_SYNC_TIME_VALUE_OFFSET = 0
PLL_REGS_INT_SYNC_TIME_VALUE = 0xffffff
PLL_REGS_INT_SYNC_TIME_UNUSED_OFFSET = 24
PLL_REGS_INT_SYNC_TIME_UNUSED = 0x7f000000
PLL_REGS_INT_SYNC_TIME_RST_OFFSET = 31
PLL_REGS_INT_SYNC_TIME_RST = 0x80000000
# ...
PLL_REGS_CONTROL_UNUSED_OFFSET = 0
PLL_REGS_CONTROL_UNUSED = 0x7fffffff
PLL_REGS_CONTROL_RST_OFFSET = 31
PLL_REGS_CONTROL_RST = 0x80000000
# ...
```

Code 1: Register constants

4.5 Software Design

This section covers the development of the prototype PLL program and the development of the Python driver to access FPGA registers. The prototype PLL

program is adapted from the C-program written for FGC3.1. In the final application, performance is essential, and therefore C or C++ programming language should be used. However, here Python allows rapid prototyping and provides satisfactory performance. The Python driver, in turn, is a result of this thesis and enables creating hardware integration tests in Python.

4.5.1 PLL program

First, the prototype PLL program is introduced. The program itself is relatively complex, and most of its functionalities are out of scope for this thesis. Therefore, only the essential functions are presented. Additionally, some variable names are written as numbers to improve readability. In the actual program, hard-coded values are used as little as possible to improve reusability. Those functions include the initialization of the PLL, the PI-algorithm, and setting a value for the DAC.

The generic algorithm that is presented here is enclosed within a class called *pll_algo*. This class will be instantiated in both simulation and hardware test environments, with only slight alterations in the initialization parameters. Thus, it will be ensured that both tests utilize the same algorithm.

The different constants, such as PID_KI and PID_KP, are provided in a config class called *pll_config*. This class conveniently displays the different constants that determine the loop characteristics. Once again, this file is shared between the simulation and hardware test environments to ensure similarity in results.

Next follows a brief description of the main functions of the prototype PLL program. The basic operation can be summarized in five steps:

- Read the internal and external timestamps
- Calculate phase error by subtracting the two timestamps
- Average 10 phase errors
- Send the average to the PI-controller
- PI-controller calculates the value and sends it to DAC

In the beginning of the program, an initialization procedure is performed. If the reset-bit is still set, the program will write an initialization value to start the PLL. The initialization value is the previous timestamp for the network interrupt pulse (reserve pulse), added to an offset value eth_sync_cal, which indicates the time difference between the arrivals of primary and reserve pulses. This offset value is a calibrated value that ensures the phase difference is as close to zero as possible at the beginning of the program. The code snippet for this process is provided below in Code 2.

The phase error is acquired by reading the FPGA registers. Once acquired, the phase error is calculated by subtracting the sync pulse timestamp from the internal sync time. The subtraction is operated with unsigned 32-bit integers with overflow. This is to achieve a realistic phase error even if the other timestamp value has just

```
# If the internal 50 Hz is still in reset state
if int_sync_time & consts.PLL_REGS_INT_SYNC_TIME_RST:
  # Check if a valid ethernet sync was received
  # If so, set the INT_SYNC_TIME to start the internal 50 Hz
  # aligned with UTC
  if not self.eth.sync_time & consts.PLL_REGS_PRIM_SYNC_TIME_OLD:
      int_sync_time = self.eth.sync_time +  self.eth_sync_cal
  return int_sync_time
```

Code 2: Calculating the initial value for the internal counter

overflown the 32-bit counter while the other has not. After that, the resulting value is constrained between $-\pi$ and π. The result is accumulated into a variable that collects the sum of phase errors.

After acquiring ten phase errors, their average is then distributed to the PI-controller. Here, the phase error is converted from clock cycles into ppm. Then, the next value to the DAC is calculated with the basic PI algorithm. The DAC value is in ppm and describes how much the VCXO frequency should deviate from the center frequency. The scaling into 16-bit binary value occurs later. Next, the DAC value is limited to ± 100 ppm, as there is no point in increasing the control signal if the VCXO is pulling as fast as possible. Finally, the integral part of the PI algorithm is accumulated, but only if the DAC is not clipped.

Finally, the dac_ppm is converted to a binary value. The ppm value, ranging from -100 to 100, is mapped as a value from 0 to 2^{16}. Here, a so-called zeroing algorithm is introduced. As seen in Section 4.2, the VCXO center frequency is not in the middle of the DAC range. This is unideal for the PI-algorithm, as dac_ppm=0 does not mean that VCXO frequency would be exactly 25 MHz. This asymmetry increases lock time by approximately ten seconds, as will be seen later. Therefore, an offset is applied, ensuring that dac_ppm=0 corresponds to approximately 1.7 V, resulting in 25 MHz VCXO frequency.

In the Code 5, this mapping process with zeroing algorithm is presented. DAC_ZERO is a hand-picked value that indicates the DAC value that leads to 25 MHz VCXO frequency, in this case, the DAC value that leads to 1.7 V. DAC_HALF_RANGE indicates the halfway point of the DAC full range, which here is $2^{16}/2 = 32768$. DAC_FULLSCALE indicates the number of bits in the DAC, which here is $2^{16} = 65536$. Finally, this process potentially shifts the dacvcxo value outside of the allowed values for the DAC, so the value is constrained between 0 and 65536. The resulting value, dacvcxo_int, is the integer part of dacvcxo, and it is distributed to the FPGA that will communicate the value to the DAC. Additionally, the version without the zeroing algorithm is presented. This version assumes that the middle point in the DAC control range achieves the VCXO center frequency.

```
def unsigned_sub(a, b, bits):
if a >= 2 ** bits or b >= 2 ** bits:
    raise Exception('Arguments must be below 2^bits')
if a < 0 or b < 0:
    raise Exception('Arguments should be positive')
if b > a:
    return 2 ** bits + a - b
return a - b

def limit_phase_err(phase_err, accum_const):
if phase_err >= accum_const/2:
    phase_err -= accum_const*round(phase_err / accum_const)
elif phase_err <= -accum_const/2:
    phase_err -= accum_const*round(phase_err / accum_const)
return phase_err

sync.phase_err = (unsigned_sub(
                  int_sync_time, sync.sync_time, self.counter_width)
                      & consts.PLL_REGS_INT_SYNC_TIME_VALUE) \
                      - phase_ref
sync.phase_err = limit_phase_err(
                      sync.phase_err, self.iter_period_ticks)
# Accumulate the sum of the phase errors
err.sync_count += 1
err.phase_err_sum += sync.phase_err

# ... after 10 samples ...

self.external.phase_err_average = \
  self.external.phase_err_sum / 10
```

Code 3: Calculating phase error average

4.5.2 Python Driver

The Python driver is developed to enable the access of the PLL registers in the FPGA. The driver is required for hardware verification. The operation of the Python-driver was explained in section 3.3.3. Here, the driver and its functions are presented.

The driver itself is wrapped into a class called wb_pll, referring to Wishbone PLL. It is initialized with the ___init___ function, requiring *bus* and *offset* as parameters. Bus instance is the instance that handles the particular communication interface from the computer to the FPGA. In this thesis, the interface is UART, but it could be any other communication interface, such as JTAG, SPI, I2C, et cetera, as long as it incorporates read() and write() functions. Offset provides the global Wishbone address of the PLL module: this value is always added to the address-parameter

```
# Convert phase error to parts-per-million compared to
# maximum phase error
# This makes the PI constants clock frequency independent
phase_err = phase_err/self.iter_period_ticks * 1e6

# Calculate DAC using PI algorithm
self.dac_ppm = self.integrator_ppm + \
    (pll_config.PID_KP + pll_config.PID_KI)*phase_err

# Clip DAC to working range
if self.dac_ppm < -100:
    self.dac_ppm = -100
    self.dac_clipped_f = 1
elif self.dac_ppm > 100:
    self.dac_ppm = 100
    self.dac_clipped_f = 1
else:
    # DAC not clipped
    self.dac_clipped_f = 0
    # Integrate new phase error
    self.integrator_ppm += pll_config.PID_KI * phase_err
```

Code 4: Running PI-algorithm and clipping the output

when using write and read functions. The ___init___(), read(), and write() functions are presented in the code snippet below. The init-function also includes acquiring the counter width, which also determines the widths of all timestamp registers. This value is converted into a bitmask.

```
def __init__(self, bus, offset):
  self._bus = bus
  self._offset = offset

  self.cntr_w = self.get_counter_width()
  self.cntr_w_mask = 2**(self.cntr_w + 1) - 1

def read(self, addr):
  assert addr <= const.PLL_REGS_SIZE
  return self._bus.read(self._offset + addr)

def write(self, addr, data):
  assert addr <= const.PLL_REGS_SIZE
  self._bus.write(self._offset + addr, data)
```

When the PLL program is starting, the FPGA program should be reset. This

```
# not zeroing
if self.sim_mode:
    dacvcxo = (self.dac_ppm / pll_config.VCXO_PULL_PPM
               + 1) / 2 * pll_config.DAC_FULLSCALE
# zeroing
else:
    dacvcxo = (self.dac_ppm / 100
               + pll_config.DAC_ZERO / pll_config.DAC_HALF_RANGE) / 2 \
               * pll_config.DAC_FULLSCALE

# Clip to allowed range
if self.dacvcxo <= 0:
    self.dacvcxo = 0
if self.dacvcxo >= pll_config.DAC_FULLSCALE:
    self.dacvcxo = pll_config.DAC_FULLSCALE - 1

self.dacvcxo_int = int(dacvcxo)
```

Code 5: Zeroing algorithm

process consists of setting the reset-bit in the control register. Once set, the program waits until the reset-bit is set in int_sync_time register.

```
def reset(self):
  """Reset the PLL.
     Waits until the reset is complete, or until timeout.
  """
  self.write(const.ADDR_PLL_REGS_CONTROL, const.PLL_REGS_CONTROL_RST)
  timeout = time.time() + RESET_TIMEOUT
  while True:
      val = self.read(const.ADDR_PLL_REGS_INT_SYNC_TIME)
      if val & const.PLL_REGS_INT_SYNC_TIME_RST:
          break
      elif time.time() > timeout:
          raise Exception("Timeout in PLL reset")
```

As previously mentioned, the software algorithm requires the information on the accumulation constant and the counter data width. These can be acquired with simple methods:

```
def get_counter_width(self):
  """Get the width of the counter.
  """
  cntr = self.read(const.ADDR_PLL_REGS_STATUS)
  return ((cntr & const.PLL_REGS_STATUS_CNTR_W)
```

```
            >> const.PLL_REGS_STATUS_CNTR_W_OFFSET)

def get_accum_const(self):
  """Get the accumulation constant.
  """
  accum = self.read(const.ADDR_PLL_REGS_STATUS_B)
  accum_field = ((accum & const.PLL_REGS_STATUS_B_ACCUM_CONST
                  >> const.PLL_REGS_STATUS_B_ACCUM_CONST_OFFSET))
  return accum_field & self.cntr_w_mask
```

The PLL is started by loading a value to the int_sync_time_load-register. For this task, a method is provided. It is called with two parameters: the first one provides the value to be loaded, and the second parameter is a boolean value, indicating whether the PLL should be started or not.

```
def init_pll(self, data, start):
"""Initialize the PLL with initial latch value.

:param data: the data to initialize PLL with, integer
:param start: start the PLL, boolean
"""
val = data << const.PLL_REGS_INT_SYNC_TIME_LOAD_VALUE_OFFSET
if not start:
    val |= 1 << const.PLL_REGS_INT_SYNC_TIME_LOAD_RST_OFFSET
self.write(const.ADDR_PLL_REGS_INT_SYNC_TIME_LOAD, val)
```

Next, all the timestamp registers can be acquired in one of two ways. Either the entire register can be queried, or only the timestamp itself. The latter also provides the old-flag as a separate return parameter, when one of the two external timestamp registers are accessed. Here, only the primary timestamp register is presented, but also the internal and reserve timestamp registers can be accessed with similar methods.

```
def get_prim_reg(self):
    """Get the register for primary sync
    """
    return self.read(const.ADDR_PLL_REGS_PRIM_SYNC_TIME)

def get_prim_timestamp(self):
  """Get the timestamp and oldness of the primary sync pulse.
  :returns Tuple with the timestamp, and 1 if the data is old, 0 if new.
  """
  val = self.read(const.ADDR_PLL_REGS_PRIM_SYNC_TIME)
  ts = ((val & const.PLL_REGS_PRIM_SYNC_TIME_VALUE)
        >> const.PLL_REGS_PRIM_SYNC_TIME_VALUE_OFFSET)
  ts_val = ts & self.cntr_w_mask
```

```
  old = ((val & const.PLL_REGS_PRIM_SYNC_TIME_OLD)
         >> const.PLL_REGS_PRIM_SYNC_TIME_OLD_OFFSET)
  return [ts_val, old]
```

Finally, there is one additional method to acquire the state of the PLL. This method returns whether the PLL is running, or whether it is still in reset state.

```
def is_running(self):
  """Returns 1 if the PLL is running, 0 otherwise.
  """
  val = self.read(const.ADDR_PLL_REGS_INT_SYNC_TIME)
  return 1 ^ ((val & const.PLL_REGS_INT_SYNC_TIME_RST)
              >> const.PLL_REGS_INT_SYNC_TIME_RST_OFFSET)
```

4.6 Unit Tests

The verification tests described in this section are unit-level tests executed for the VHDL part of the PLL, i.e., the phase detector. The purpose of these tests is to prove the functionality of all functions of the module, divided into separate test cases so that if one test fails, it is easy to discover the root cause. Furthermore, the tests have a specific hierarchy: they depend on the proper operation of functions tested in a separate test. Therefore, if multiple tests fail, the lowest test in the hierarchy shall point to the root cause.

The tests are conducted with a verification framework called VUnit. The framework provides functions and modules for the stimulus driver, monitor, scoreboard, and checker, simplifies the testbench structure, and enables transaction-based testing. All tests are executed with Wishbone transactions directed by a VUnit verification component called Wishbone Master. A framework called OSVVM is also used for some functionalities that VUnit does not directly provide. For example, the capability of providing random values is enabled by this framework.

The verification is designed to cover all functions of the phase detector module. Therefore, it is coverage-driven, while the metric is the functional coverage. Additionally, the code and branch coverages are logged, and especially the code coverage is expected to reach 100%. All test cases are run on different parameters, including different clock frequencies and synchronization pulse frequencies. The fact that tests pass on all parameters shows that the module has been parametrized succesfully.

4.6.1 Test Bench Structure

The testbench consists of two modules: the Wishbone PLL and a Wishbone Master, as displayed in figure 23. The PLL module is the design under verification (DUV). Its inputs are driven both manually and by the Wishbone Master. The clock signals, sync_clk_i and wb_clk_i, are generated to have synchronous rising edges. The two synchronization pulses, main_sync_i and res_sync_i, are created to be asynchronous with respect to each other and the other clock signals. The only output, ref_pulse_o, is stored into a signal and checked by the tests but not used in any other way.

The Wishbone Master handles the Wishbone transactions by driving the wb_i signals and receiving the wb_o signals (the signal directions are from the perspective of the PLL). It receives the same clock signal as the WB PLL, wb_clk_i. Additionally, one input is provided: a bus handle. This is required for transaction-based testing. Thus, Wishbone-transactions can be initiated with read_bus and write_bus commands, which take the bus handle, data, and address as parameters.

Therefore, the stimulus driver consists of manually created signal generators and the Wishbone Master. The VUnit framework handles the other parts of a generic testbench. The checker commands can be manually created with check() and check_bus() commands. Results from these checks are automatically sent to the scoreboard. After all tests have been run, the results are assembled into an overview that shows which tests passed and which failed, alongside some technical information like run time for every separate test. VUnit can also generate coverage information, namely code coverage and branch coverage. Next, using these functionalities and modules, unit-level tests are composed to reach as high coverage scores as possible.

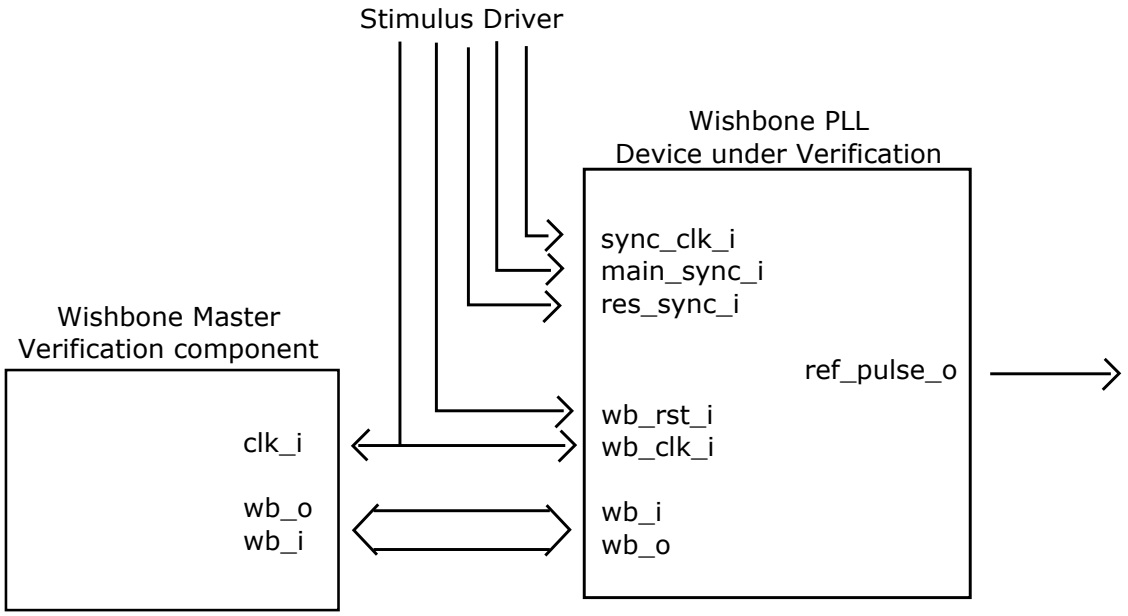


Figure 23: Testbench blocks

4.6.2 Test reading Wishbone registers

The first test is to check whether writing to and reading from the Wishbone registers works. The success of this test indicates that the Wishbone Master has been correctly instantiated in the testbench, that the Wishbone signals have been correctly wired, and that the Wishbone registers inside the PLL module function correctly. It also acts as a sanity test that the testbench and modules are syntactically correct. This is a fundamental test: passing it is the first step in verification. It should be instantiated as soon as possible, and every time significant changes are made in the modules, this test should be executed first to see that the fundamental operations still work.

Furthermore, every other test depends on Wishbone operations: thus, if this test fails, every other test will fail too.

The only WB register both readable and writable is the INT_SYNC_TIME_LOAD-register. Therefore, these operations must be targeted to this register. An additional read-operation is performed on one of the STATUS-registers to see that the generics have been correctly transferred to the registers. In this test, setting the reset-bit in INT_SYNC_TIME_LOAD is crucial to prevent the PLL from starting. Testing that functionality is reserved for a separate unit test.

4.6.3 Test starting internal counter

This unit test is to verify that the internal counter, a fundamental part of the PD functionality, works as intended. The test is performed by first enabling the PLL by setting the reset-bit in INT_SYNC_TIME_LOAD-register as zero. This will enable the PLL, load the contents of the aforementioned register into INT_SYNC_TIME-register, and start the internal counter.

The test is executed by first writing to the INT_SYNC_TIME_LOAD-register and waiting for a few clock cycles to let the enable-signal propagate throughout the design. Next, the counter-bits are read from the STATUS-register and stored into a variable. Then, a wait of several clock cycles is performed. Finally, after the wait, the counter-bits are reread. The read counter value should equal the previous counter value added to the number of clock cycles waited.

4.6.4 Test latching of the registers

This test verifies whether the internal registers latch the counter value when receiving a synchronization pulse. An identical test is run for both PRIM_SYNC_TIME and RES_SYNC_TIME registers.

First, the internal counter is initiated like in the previous test. Then, both registers are read and stored into variables. Next, the simulation waits until a rising edge of the primary synchronization pulse appears before rereading the PRIM_SYNC_TIME-register. In between, five clock cycles of delay is added to allow signals to propagate. The new register value is compared to the previous one: if there is a difference, then the register has latched the counter value successfully. Finally, the same procedure is performed for the reserve synchronization pulse and the respective register.

4.6.5 Test OLD-flag functionality

This test is very similar to the previous one, but with the addition that the OLD-flag operation is checked. The flag should trigger when the register is accessed, and it should reset when new data latches in.

Both registers are read twice. On the first read, the OLD-flags should be zero. On the second read, the flags should read as one. Next, the simulation waits until a rising edge of both the main and the reserve synchronization pulse to ensure that both registers have new data. Then, the registers are, again, read twice, and the flag behavior should be like the previous reads.

4.6.6 Test the accumulation of the internal synchronization register

This test is to ensure that the internal accumulator functions as expected. As explained earlier, the accumulator outputs the comparison value for the internal counter, and every time the counter reaches its value, the value inside the accumulator is increased by the accumulation constant.

The test is conducted by initiating the internal counter like in previous tests. Then, the value inside INT_SYNC_TIME-register is read and saved into a variable. Next, another variable is created, which holds the value of the previous variable added with the accumulation constant. Then, the simulation waits and constantly reads the register until it switches to a value the same as the second variable. Additionally, there is a timeout of two synchronization pulse periods for the case where the register would not update as expected. The test will fail if this timeout is triggered.

4.6.7 Test the generation of internal reference pulse

This test checks whether the PLL generates a reference pulse when the internal counter equals the value stored in the internal accumulator. This pulse is essential, as other modules in the project use it for synchronization purposes.

The test initiates the internal counter as in previous tests, and the simulation then waits until a rising edge of the reference pulse. A timeout of one synchronization pulse period indicates if the pulse is not correctly generated within the desired time window. The test is then repeated for a second time for the sake of safety.

4.6.8 Test resetting the module

This test tests the functionality of resetting the module by writing to the CONTROL-register. This test ensures that the registers reset appropriately to their intended values.

The test is conducted by first checking the registers after the initial manual reset, then enabling the PLL by writing to the INT_SYNC_TIME_LOAD-register. Next, the module is reset again by writing to the CONTROL-register. Then, the registers are checked to have the same values as after the manual reset.

4.6.9 Metrics

Coverage data presented in table 5. The coverage metrics used here are the code and branch coverage, as they are easily available from the results generated by GHDL and VUnit. As explained, the functional coverage is quite arbitrary, and the designer must decide whether full functional coverage has been reached. Here, to the author's best understanding, all functionalities of the phase detector have been tested.

The code coverage reaches 100% for all the main modules developed in this thesis. However, this only counts the lines of code in the highest hierarchy level, not in possible submodules, such as the edge detector and external input modules. Nevertheless, these modules are assumed to have already been verified.

Metric	wb_pll	pll_top	sync_latch	wb_pll_regs	Total
Code coverage (%)	100	100	100	100	100
Branch coverage (%)	71.3	71.4	72.9	78.5	74.9

Table 5: Coverage metrics for the unit-level tests

4.7 Full system simulations

Full-system simulations are conducted with a Python-based framework called cocotb. It empowers interfacing a VHDL design via a Python interface. This allows the easy execution of complex software-based algorithms simultaneously with the VHDL design, which, in this case, enables the simulation of the entire PLL. This, in turn, allows the tuning of PLL parameters in a simulation environment, which is much faster than in a hardware testbench environment.

However, as the PLL also consists of hardware components, the VCXO, a software model must be created for this. In the next section, the simulation model is introduced.

4.7.1 System-Level Testbench

The testbench for the system-level tests consists of four separate components: the wb_pll VHDL-module, custom clock generators for generating synchronization pulses, a custom VCXO-model, and a WishboneMaster-module. The relation between them is shown in figure 24. For the two cocotb-domain models, the relevant methods are visible. The clock generators generate a pulse that represent the synchronization pulses every 20 milliseconds, to which jitter can be added. Finally, it should be noted that the system clock frequency for the testbench was set to 1 MHz (the hardware system frequency is 25 MHz) to decrease simulation time. Nevertheless, using a different clock frequency does not drastically change the characteristics: it only reduces the phase detector resolution.

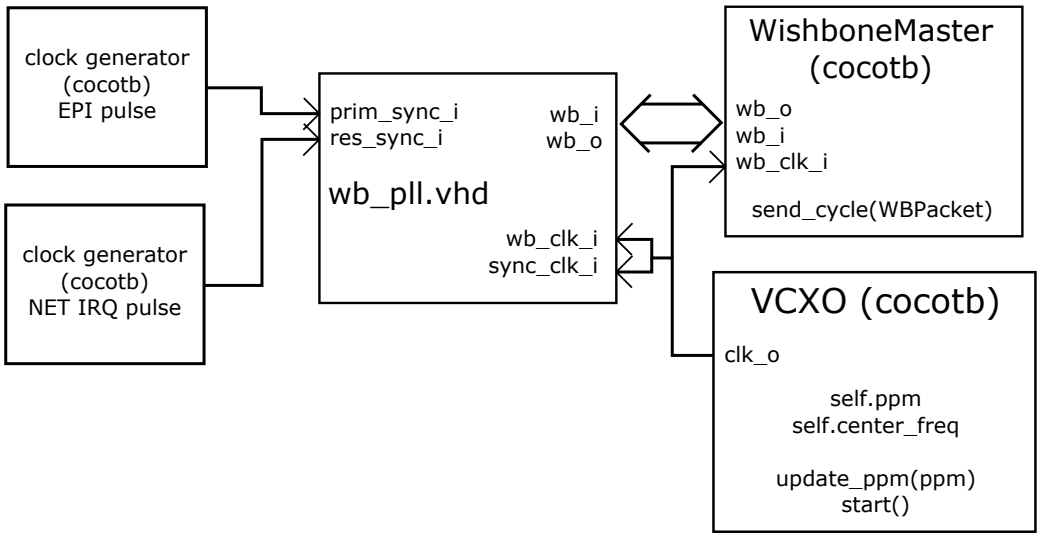


Figure 24: System model testbench

The VCXO-model provides a modifiable clock signal for the testbench. During initialization, it is provided with the center frequency *self.center_freq*, and the clock signal to be driven. The deviation from center frequency, *self.ppm*, is initialized as zero, but this can be updated at any time by using the method *update_ppm()*. The deviation is expressed as ppm, corresponding to the VCXO frequency deviation. Finally, the VCXO starts to generate the clock signal when the *start()* method is called. Below, the code snippet for the VCXO model is provided. It should be noted that the *Timer()* is a class provided by cocotb, and the input parameter *signal* is also a cocotb-type. The rest is pure Python.

However, it should be noted that this model of the VCXO does not take into account the asymmetry of the DAC and VCXO control ranges, as explained in section 4.2. Additionally, the pullability is constrained to ±100 ppm, whereas the real VCXO is pullable between approximately −250 ppm and +130 ppm. Therefore, slight differences between the simulation model and real hardware are expected.

```
class VCXO_model:
"""Generates clock signal with a given center frequency.
The center frequency can be updated like in a real VCXO.
"""
def __init__(self, signal, cf):
    self.clk = signal
    self.ppm = 0            # deviation from center freq in ppm
    self.center_freq = cf   # center frequency

def update_ppm(self, ppm):
    self.ppm = ppm

def update_bit(self, bit):
    """Updates the VCXO by bit value.
    Resolution is determined by pll_config file

    Args:
        bit (int): Bit word to be written
    """
    self.ppm = (bit - pll_config.DAC_OFFSET) \
        * pll_config.VCXO_PULL_PPM / pll_config.DAC_HALF_RANGE

async def start(self):
    it = itertools.count()
    # clock loop
    for _ in it:
        self.clk.value = 0
        await Timer(round(1e15/((1+self.ppm*1e-6)*self.center_freq)
                    * 0.5), units='fs')
        self.clk.value = 1
```

```
        await Timer(round(1e15/((1+self.ppm*1e-6)*self.center_freq)
                    * 0.5), units='fs')
```

The WishboneMaster-module was downloaded from GitHub [28]. The module is initialized by providing the Wishbone-bus name (in this case *wb*), the clock signal, the data and address width (32), a timeout limit, and the Wishbone-signal names. Thereafter, Wishbone-transactions can be initiated by using the method *send_cycle()*. The parameter for this method is a special Wishbone operator, which includes the address and data, the latter being blank for a read-only operation. If a response is expected from the Wishbone-slave, it is returned as a Wishbone Result Wrapper Class. For example, the result can be converted from the Wrapper Class to a binary string.

Additionally, two clock signals are generated: the external and reserve synchronization 50 Hz pulses. The generators are manually designed to generate a pulse with a width of 1 millisecond. They are wired to the DUV as shown in figure 24. The generators are also configured so that it is possible to generate a jitter to test the PLL phase-locking capability. The jitter is provided as a peak-to-peak value in microseconds.

Furthermore, a function to enable and disable the pulse generation is provided. Disabling the clock means that the clock signal is always low. This functionality is required to test the absence of the primary pulse. The code for these pulse generators is provided below.

```
"""Generates pulses every 20 ms with
1 ms pulse width with controllable jitter.
"""
def __init__(self, signal):
    self.clk = signal
    self.jitter_us = 0  # peak-to-peak jitter in microseconds
    self.t_low_ms = 19  # low time in milliseconds
    self.t_high_ms = 1  # high time in milliseconds
    self.enabled = True
    self.cyc_jitter_ns = 0  # Jitter in one clock cycle

def update_jitter(self, new_jitter_us):
    self.jitter_us = new_jitter_us

def disable(self):
    self.enabled = False

def enable(self):
    self.enabled = True

async def start(self):
    it = itertools.count()
    for _ in it:
```

```
        # generates a random value within the given jitter interval
        # then converts from microseconds to nanoseconds
        self.cyc_jitter_ns = round(1e3*random.uniform(
            -self.jitter_us, self.jitter_us), 2)
        # making sure the total cycle time remains 20 ms
        t_low_ns = self.t_low_ms*1e6 + self.cyc_jitter_ns
        t_high_ns = self.t_high_ms*1e6 - self.cyc_jitter_ns
        self.clk.value = 0
        await Timer(t_low_ns, units='ns')
        if self.enabled:
            self.clk.value = 1
        await Timer(t_high_ns, units='ns')
```

The main program is relatively simple. The pll_algo-class is initialized with an initial phase such that the initial phase difference is 100 µs. If desired, the simulation could be initialized with a zero phase difference, but this would not represent a realistic test scenario, as there is always an initial phase difference in real life. The pll_regs module holds the Python constants for addressing the Wishbone registers. Once the algorithm, config, and constants classes have been instantiated, the main loop begins. The main loop executes the basic commands to perform the PI-controller tasks and then waits for a fixed time of 20 ms. The basic functionality of the loop is the following:

```
sim_time = 0
isr_interval = Timer(20, units='ms')  # interval for the sim - 20ms
while sim_time < max_sim_time:
    # request data from WB registers
    wbRes = await wbs.send_cycle([
                    WBOp(pll_regs.ADDR_PLL_REGS_INT_SYNC_TIME),
                    WBOp(pll_regs.ADDR_PLL_REGS_PRIM_SYNC_TIME),
                    WBOp(pll_regs.ADDR_PLL_REGS_RES_SYNC_TIME)])
    # extracting the data
    int_sync = int(wbRes[0].datrd.binstr, 2)
    prim_sync = int(wbRes[1].datrd.binstr, 2)
    res_sync = int(wbRes[2].datrd.binstr, 2)

    # store the register data to local variables
    # used by the pll_algo
    pll_algo.ext.sync_time = prim_sync
    pll_algo.eth.sync_time = res_sync

    # run the PLL algorithm. Returns the value to write
    # to the register if the PLL was not yet running
    int_sync_new = pll_algo.pll_calc(int_sync, t_now=sim_time)

    clk.update_bit(pll_algo.dacvcxo_int)
```

```
# If internal sync is still in reset state,
# write to the internal register to start
if int_sync & pll_regs.PLL_REGS_INT_SYNC_TIME_RST:
    await wbs.send_cycle(
        [WBOp(pll_regs.ADDR_PLL_REGS_INT_SYNC_TIME_LOAD,
              int_sync_new)])

sim_time += 0.02
await isr_interval
```

Here, WBOp is a Wishbone packet transmitted to the VHDL module with wbs.send_cycle method. The wbs-class has been instantiated earlier with parameters pointing to the name of the VHDL module and its ports that handle the Wishbone transactions. The simulation executes until maximum simulation time is reached. Data is logged into a file that allows plotting the data later.

The optimal values in FGC3.1 are presented in [11], stating $\kappa_p = 0.05$ and $\kappa_i = 0.003$. This results in $\tau_2 = 3$. These values will be used as initial values. However, slight alterations must be made, as the algorithm in FGC3.1 does not convert the phase difference from clock cycles into ppm. Therefore, the PD gain is only $500000/2\pi = 79578$, whereas the PD gain for the software algorithm used in this thesis is 159155. The difference is $159155/79578 = 2.0$. Thus, the κ_p value must be divided by 2.0 to reach the same loop characteristics, resulting in $\kappa_p = 0.025$. κ_i must be divided similarly by 2.0. However, τ_2 remains the same regardless of the algorithm differences, as the multipliers in κ_p and κ_i cancel each other out.

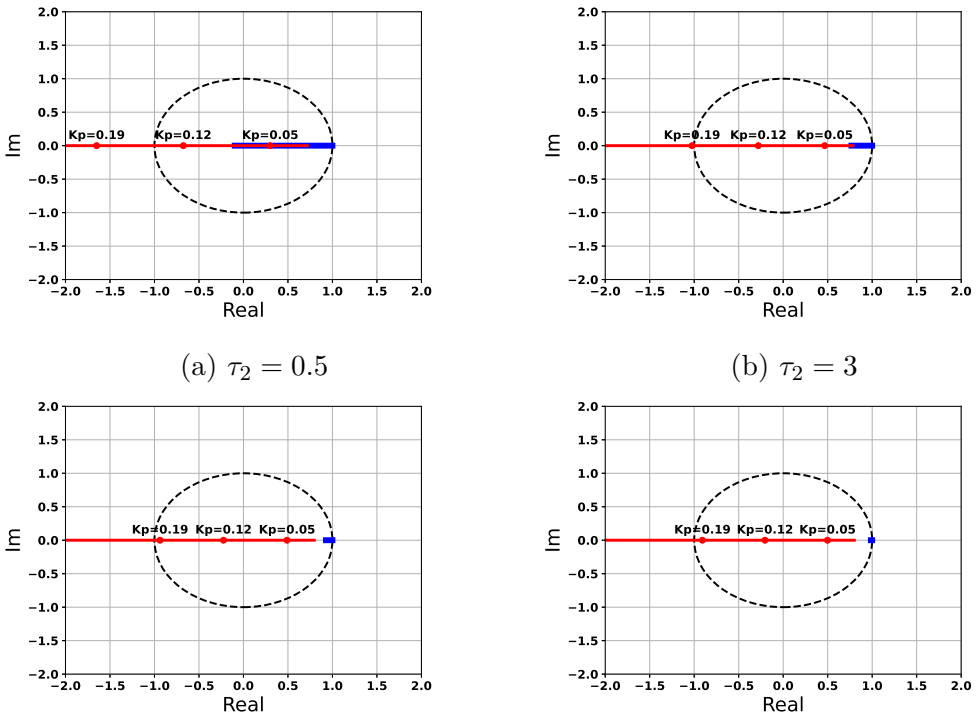
The system model should be adjusted slightly to correspond to the system-level testbench. As previously explained, the nonidealities are not present in the system-level testbench, which means that the DAC range is between 0 V and 3.3 V. Additionally, the pullability is set as ±100 ppm, which means that K_V is 200ppm/3.3V = 60.6 ppm/V, instead of 150 ppm/V. This change changes the stability boundaries slightly. The root-locus plots for the simulation model are presented in figure 25. For reasonable τ_2 values, the boundary condition for stability is around $\kappa_p = 0.19$ instead of $\kappa_p = 0.1$.

4.7.2 Test lock time for step input with different parameters

In this test, the time to reach phase-lock is observed for different PI-block parameters. This test is to narrow down the range of suitable parameters. The two parameters to be swept are κ_p and τ_2. For κ_p sweep, τ_2 is held at 3, while in τ_2 sweep, κ_p is held at 0.025, as these are the parameters provided in [11], with appropriate adjustments.

This test aims to verify that the complete PLL system operates in a deterministic way when the PI algorithm is executed with different parameters. The behavior is expected to be similar to a second-order system. Additionally, the boundaries for stability are verified: the system is expected to become unstable at around $\kappa_p = 0.19$.

Figure 26 shows the result of the κ_p-sweep. Firstly, the system functions as expected, as the phase error converges to zero in a way that resembles a second-order



(a) $\tau_2 = 0.5$ (b) $\tau_2 = 3$

(c) $\tau_2 = 10$ (d) $\tau_2 = 50$

Figure 25: Pole locations for the simulation model with different τ_2 values.

system. Evidently, the higher the κ_p, the faster the lock time until the boundary condition for stability is reached. $\kappa_p = 0.15$ still converges to zero, while $\kappa_p = 0.2$ does not. This is expected behavior, and if the lock time was the only optimized parameter, κ_p close to the stability boundary should be chosen. However, having a high κ_p decreases jitter filtering, as will become evident later.

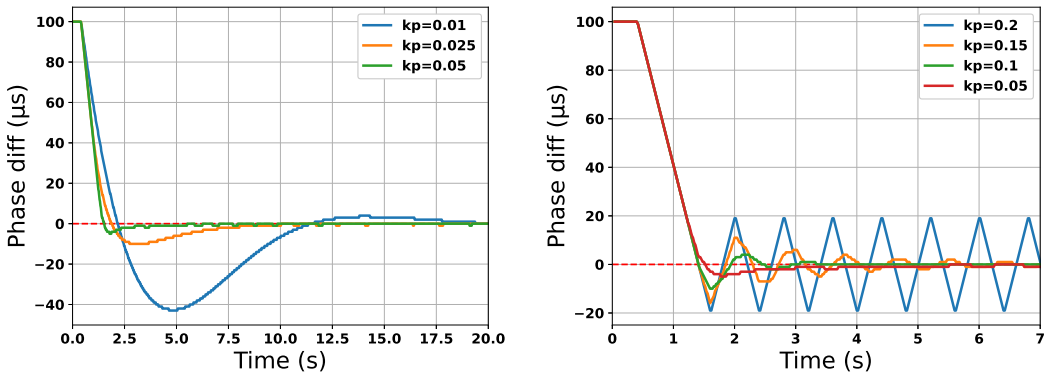


Figure 26: κ_p sweep with 1 MHz clock frequency

Figure 27 shows the result of the τ_2 sweep. Higher τ_2 seems to result in a smaller overshoot, but reaching absolute zero phase difference lasts longer. This is because the higher τ_2, the lower κ_i, which results in slower accumulation of the PI controller

integral part. As seen in section 4.5.1, the integrator part does not accumulate when the VCXO is pulled with maximum value. Therefore, when the phase error approaches zero, and the DAC output is regulated, there is a tiny integrator part to deplete, resulting in a small overshoot. However, depleting even the small integrator part takes a long time due to low κ_i, which is the reason for slow convergence to absolute zero.

Finally, it should be noted that the lock time, as indicated in the figure, does not refer to a best- or worst-case lock time. The phase difference primarily determines the lock time at the beginning of the program and the pullability of the VCXO. In these tests, the initial phase difference was forced at 100 µs. Pullability is constrained to values between ±100 ppm.

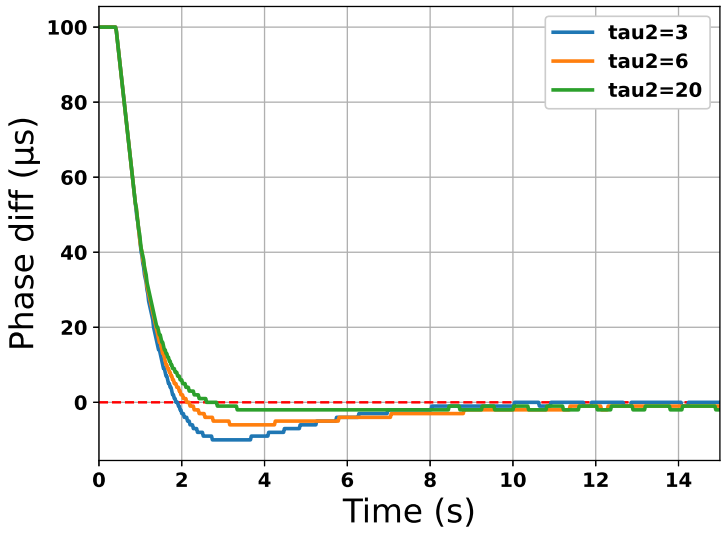


Figure 27: τ_2 sweep with 1 MHz clock frequency and $\kappa_p = 0.025$

4.7.3 Test lock time with offset in VCXO center frequency

The VCXO center frequency is not always precisely where the datasheet promises. For the VCXO used in this thesis, the stability is defined as ± 20 ppm. Since the pullability is defined in relation to the actual center frequency, this deviation may significantly impact the lock time. For example, if the center frequency is 25 MHz + 20 ppm, the VCXO runs $25 \times 20 = 500$ Hz faster than expected. If the internal phase is ahead of the external phase (int_sync_time < ext_sync_time), the VCXO must be slowed down to decrease the phase difference. Now, however, applying -100 ppm to the VCXO center frequency will result in a frequency of only 25 MHz - 80 ppm. Therefore, the phase difference will decrease twice slower than with an ideal VCXO, whose frequency would be 25 MHz - 100 ppm after the same operation.

However, in the other direction, i.e., if the internal phase were lagging the external phase, the VCXO would be pulled to 25 MHz + 120 ppm. As a result, the phase

difference would decrease 20% faster than optimal VCXO and 50% faster than worst-case.

This test case is critical because, as seen in section 4.2, the VCXO control range center does not align with the DAC control range. This test displays the effect when the software algorithm is not zeroed around the new center.

In the following tests, the internal phase initially lags behind the external phase. Similar κ_p-sweeps are applied as in the previous section. The results are displayed in figures 28 and 29. In the first test, the VCXO center frequency is 20 ppm higher than optimal, whereas it is 20 ppm lower in the second test. As is evident, the loop characteristics change: in the first case, the overshoot increases for all κ_p values, whereas in the other case, overshoot has changed to undershoot. The first case reaches zero phase error in a shorter time but, due to the overshoot, converges slower than the second case.

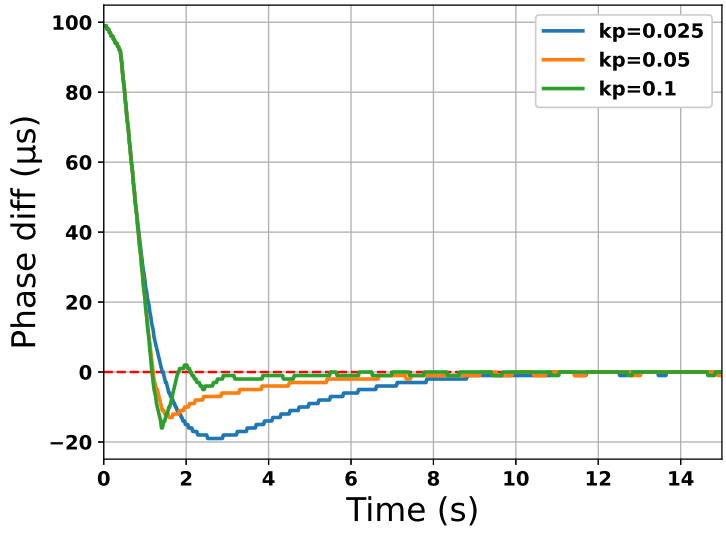


Figure 28: κ_p sweep with 1 MHz + 20 ppm clock frequency

Figure 30 displays a τ_2 sweep with $\kappa_p = 0.025$. Here, the disadvantage of choosing a high value for τ_2 is evident. Because zero in the software program does not indicate exactly 25 MHz, the PI-algorithm must accumulate an integrator part to remove the steady-state error. The higher τ_2, the slower this process is. Therefore, in this case, the lower τ_2, the faster the lock-in process is, which is the opposite compared to figure 27.

The zeroing algorithm was discussed in section 4.5.1. In figure 31, the result from figure 28 is compared with a new result that uses the same clock frequency but with the introduced zeroing algorithm. As can be seen, the overshoot is smaller, while there is little difference in the lock time. The result is highly similar to the case with ideal VCXO in figure 27.

In conclusion: a deviation of the VCXO center frequency results in an asymmetry in the lock time behavior of whether the internal phase is preceding or lagging the external phase. The lock is reached in a reasonable time also in the worst-case

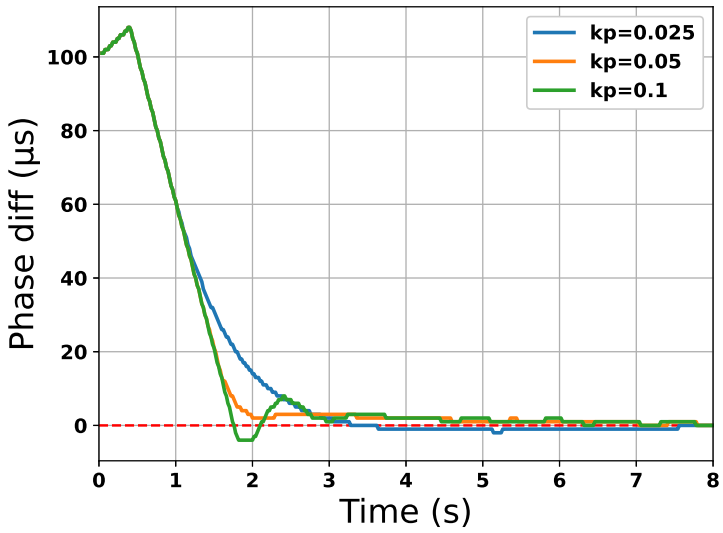


Figure 29: κ_p sweep with 1 MHz - 20 ppm clock frequency

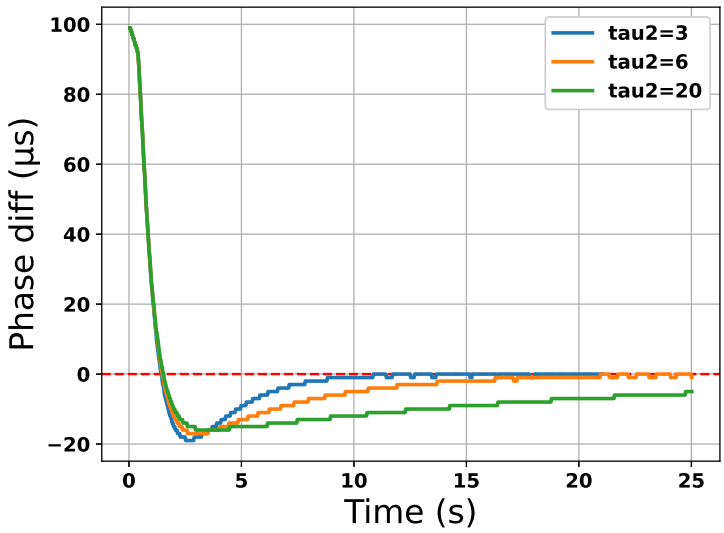


Figure 30: τ_2 sweep with 1 MHz + 20 ppm clock frequency and $\kappa_p = 0.025$

scenario, but a certain asymmetry in the behavior should be expected. The presented zeroing algorithm can be used to remove the asymmetry. Additionally, a high tau_2 value increases the time required to acquire the PI-algorithm integral part to remove the steady-steady error.

4.7.4 Test jitter filtering with different parameters

This simulation aims to demonstrate the jitter filtering capability of the PI-block. Both the effect of κ_p and τ_2 are observed.

Figure 32 displays the effect of τ_2 on the jitter filtering. This variable does not

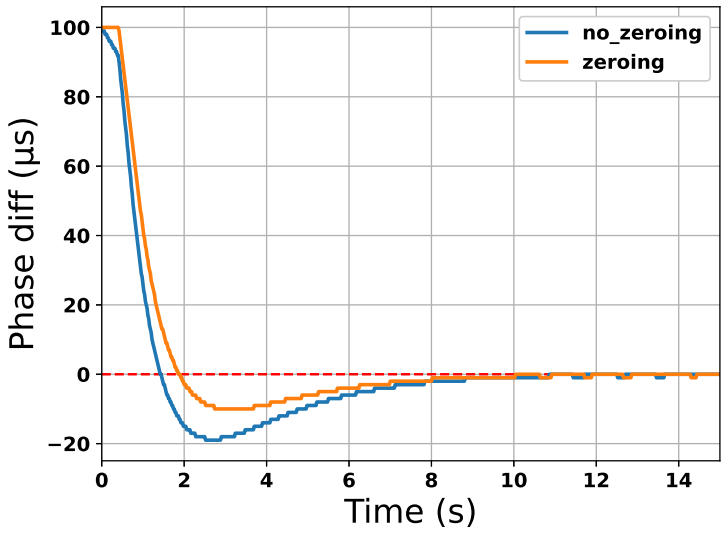


Figure 31: Comparison of zeroing algorithm vs normal algorithm. $f_{clk} = 1$ MHz + 20 ppm, $\kappa_p = 0.025$ and $\tau_2 = 3$

significantly affect the filtering capabilities. All graphs seem to incorporate a jitter of approximately two clock cycles peak-to-peak. Considering that the added filter is ten cycles peak-to-peak, the filtering is quite effective.

Figure 33 shows the effect of κ_p. Here, the difference is clear. The lower the κ_p is, the better jitter filtering. Therefore, selecting the κ_p value compromises jitter filtering and lock time.

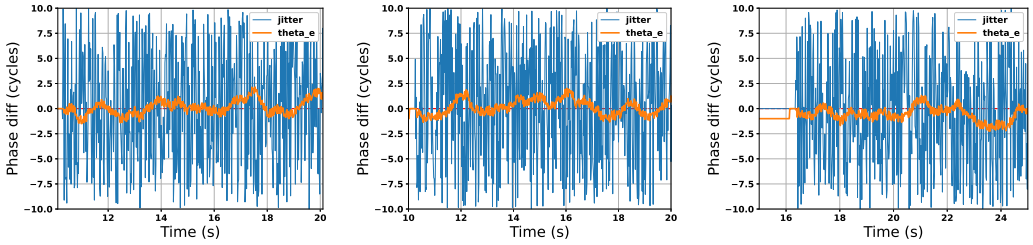


Figure 32: Jitter filtering with $\tau_2 = 1$, $\tau_2 = 3$ and $\tau_2 = 6$

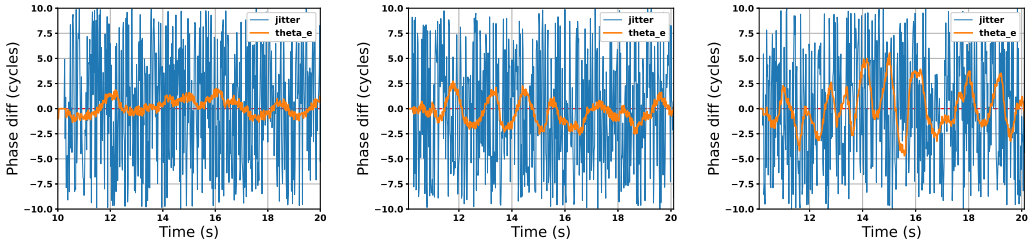


Figure 33: Jitter filtering with $K_p = 0.025$, $K_p = 0.05$, and $K_p = 0.1$

4.8 Hardware Tests

For the hardware tests, a slightly altered software routine is utilized. The main loop is executed here until the user stops the program with a keyboard interrupt. Each new loop begins once new data is available in the primary timestamp register. This, effectively, creates a 20 ms interrupt service routine because a new pulse arrives every 20 ms.

It was noticed that the computer used in the test setup, an Intel NUC minicomputer, was not powerful enough to execute this loop in 20 ms while logging data to a file. Therefore, the data was printed on the terminal. After the run had finished, the data was copy-pasted to a csv-file. Thus, the loop executes under 20 milliseconds, allowing the algorithm to perform once per each new timestamp as intended.

```
t_zero = time.time()
# 20ms ISR-mimic
while True:
    prim_sync = pll.get_prim_reg()
    # If data is old, repeat
    if prim_sync & pll_regs.PLL_REGS_PRIM_SYNC_TIME_OLD:
        continue

    int_sync = pll.get_int_reg()

    pll_algo.ext.sync_time = prim_sync
    pll_algo.eth.sync_time = 0 # not used, to improve performance

    int_sync_new = pll_algo.pll_calc(
      int_sync, t_now=time.time()-t_zero)

    write_vcxo(vcxo, pll_algo.dacvcxo_int, verbose)

    # Init PLL if still in reset state
    if int_sync & pll_regs.PLL_REGS_INT_SYNC_TIME_RST:
        pll.init_pll(int_sync_new, start=True)
```

4.8.1 Test centering algorithm

This test illustrates the difference between using and not using the centering algorithm. In short, the center algorithm aligns the VCXO zero in software to the actual zero in hardware. More details are presented in section 4.5.1.

Figure 34 displays the difference between the two algorithms. The left-hand figure shows two runs in which the centering algorithm was used. In the first run, the phase difference was, initially, positive, whereas, in the second run, it was negative. The convergence to zero is almost symmetrical for both cases, including the final lock-time. The right-hand figure shows two runs in which the zeroing algorithm was not used. Here, the behavior is asymmetrical. Overshoot is larger in the second run

than in the first run. Nevertheless, the lock-time is almost equal, regardless of the asymmetry.

The lock-time is lower for the centering algorithm by a couple of seconds. This occurs because the PI algorithm must accumulate an integral part to generate a control signal that removes the steady-state error.

The tests in the following sections will use the centering algorithm to provide more predictable results. The asymmetry between DAC and VCXO will probably be fixed in a future prototype, and, therefore, the results will be more useful if the algorithm is utilized as if the asymmetry did not exist.

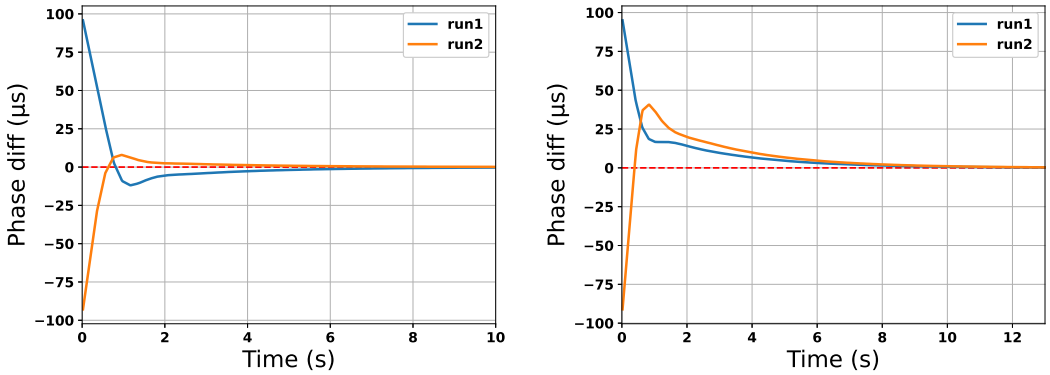


Figure 34: Difference between zeroing algorithm and no zeroing algorithm

4.8.2 Test lock time for step input with different parameters

This test conducts the same test as in section 4.7.2 but in a hardware environment. The results should differ predictably, as the system include slightly different loop multipliers. In short, the hardware PLL should become unstable with lower κ_p values than the software model because the poles exit the unit circle in lower values. For the hardware model, the boundary is approximately $\kappa_p = 0.1$, whereas for the simulation model it was $\kappa_p = 0.19$.

The results for κ_p-sweep are provided in figure 35. The lock-time is slightly faster than in the simulation (figure 27) because the pullability of the VCXO is much higher than the one used in simulations, ±100 ppm. The stability boundary, which is between 0.7 and 0.8, is slightly lower than anticipated by the system model.

Results for τ_2-sweep are visible in figure 36. Lower τ_2 seems to result in larger overshoot but lower lock-time. Therefore, a lower τ_2 value appears to be the better choice. However, a lower τ_2 also results in higher loop gain K, which effectively increases loop bandwidth and, thus, decreases noise filtering.

4.8.3 Test phase drift after losing the sync

This test is to repeat the test in [11]. After reaching the lock, the synchronization pulse is removed for a few hours, which freezes the PI algorithm. The phase difference is expected to drift slowly out of sync due to the inaccuracy of the DAC output.

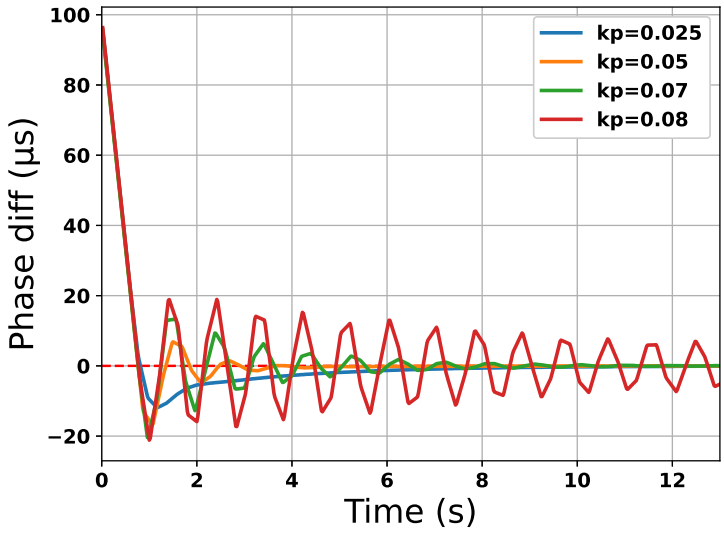


Figure 35: κ_p sweep in hardware

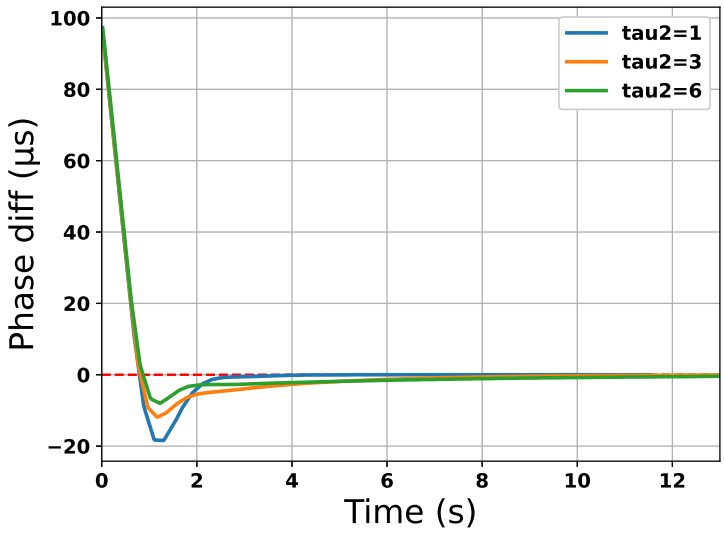


Figure 36: τ_2 sweep in hardware

Once the synchronization pulse is re-attached, the phase error should converge to zero.

This test is fascinating, as one of the hardware changes from FGC3.1 to FGC3.2 is the increase of DAC resolution from 14 bits to 16 bits. Therefore, the DAC output is more accurate and, theoretically, should have a smaller drift.

Figure 37 displays the phase difference after the synchronization pulse is re-attached. The pulse was absent for more than 2.5 hours. During this time, the phase difference drifted to approximately 200 µs. This accounts for a drift of 0.022 µs/s. Compared to the reported drift in [11], 1 µs/s, the difference is 45x. This is a clear improvement that has been achieved from the hardware upgrades.

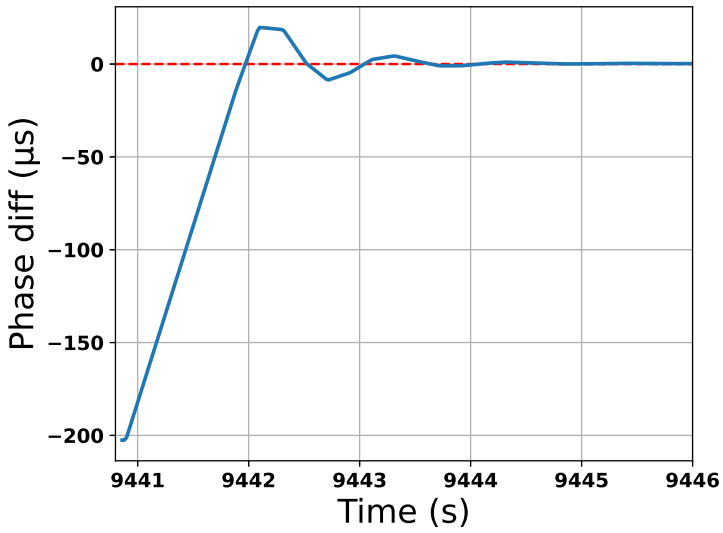


Figure 37: Phase error after losing sync for almost 3 hours. $\kappa_p = 0.05$ and $\tau_2 = 3$.

4.8.4 Discussion

A few remarks about the results should be made. First of all, it is difficult to define an absolute lock-in time for this PLL. Primarily, it is determined by the initial phase error and the VCXO pullability, and only secondarily by the chosen loop parameters. Also, this thesis does not decide the final parameter values. In the final application, the phase error is initialized by loading int_sync_time with the reserve pulse timestamp. In this way, the initial phase error is relatively small, probably smaller than the 100 µs (2500 clock cycles) that was used in these tests. So, the lock-in time can be said to be less than the values acquired in these tests. However, the reserve pulse could not be used in these tests, so this initialization scheme could not be performed.

Once the PLL reached lock, the phase error remained zero also in hardware tests. Occasionally, the phase error drifts to one clock cycle, but then the PI-controller would adjust its output so that the error drifts back to zero, before reaching a phase error of two clock cycles. Therefore, the time accuracy of the PLL with the primary synchronization pulse can be said to be better than two clock cycles. In time, this means better than 80 ns. The time accuracy could be improved even further by using a higher clock frequency for the phase detector. However, according to [11], the primary source of jitter is the varying cable lengths in the installations, so improving the clock frequency might actually not be that helpful in the big picture.

Evidently, the primary 50 Hz synchronization pulse has very low jitter. Therefore, the jitter filtering capability is not very meaningful, when this pulse is available. However, the reserve pulse has much higher and unpredictable jitter that is essentially boundless for some installations. In these cases, a jitter filtering capability is needed.

The main results are provided in Table 6 for parameter combinations that seem most suitable for the author. The lock-in time was measured by finding the time when the phase error reaches one clock cycle (40 ns), and does not become larger

than that anymore. The initial phase error is 100 µs (2500 clock cycles). The peak phase error under jitter is rounded to the nearest integer. The phase drift was tested for only one parameter combination, but all combinations are expected to behave similarly. Additionally, it is provided whether the result is derived from hardware tests or simulations.

Metric	$\kappa_p = 0.025$ $\tau_2 = 3$	$\kappa_p = 0.05$ $\tau_2 = 3$	$\kappa_p = 0.025$ $\tau_2 = 1$
Lock-in time (s) (HW)	14.3	9.6	4.5
Peak θ_e under jitter (µs) (SIM)	2	3	2
θ_e drift when no pulse (HW)	not tested	0.022 µs/s	not tested

Table 6: Main results from simulation and hardware tests

5 Summary

This thesis implemented and verified a PLL that reaches a phase-lock with a constant 50 Hz synchronization pulse. The PLL is part of a system that synchronizes CERN power converters with UTC time in microsecond accuracy.

First, PLL theory was introduced, so that a system model could be constructed for the simulation and the hardware models. Stability boundaries were found by utilizing root-locus plots, and they were at $\kappa_p = 0.19$ and $\kappa_p = 0.1$ for the simulation and hardware models, respectively, for $\tau_2 \geq 3$. Tests proved that these predictions were quite accurate in both environments.

The PLL was designed for a platform that utilizes a combination of an FPGA and a microprocessor. The FPGA implemented the phase detector and microprocessor the loop filter. The VCO was an external crystal oscillator controlled with a DAC.

The phase detector was designed using programmable logic on the FPGA. The RTL was introduced as logic diagrams. The design used the generator tool Cheby to generate a module that enables accessing internal registers via a Wishbone interface. However, none of the RTL is specifically Wishbone-dependent, so the same module can later be instantiated with a different bus, for example AXI, with minor changes. Additionally, the module can now be instantiated with different parameters, such as clock freqency. The RTL was then verified with unit tests, which utilized the VUnit framework. The RTL design reached an excellent verification quality: a code coverage of 100% and branch coverage of 74.9%.

The loop filter design was performed by translating an old C implementation into Python, which was used for ease prototyping, and is not meant to be the final production version. At the center of the LF is the PI-controller that regulates the VCXO control signal. The PI-controller includes two constants, κ_p and τ_2, which define the loop characteristics.

The complete PLL system was verified using a framework called cocotb. The environment allowed executing tests, such as κ_p and τ_2 sweeps, to observe the effect of the loop parameters. It also verified the system transfer function, as instability occurred with almost the same parameter values that the root-locus plots predicted. Additionally, the simulation environment enabled testing random jitter rejection, in which κ_p had a significant effect, with lower values increasing jitter rejection.

Finally, hardware tests were conducted to verify the simulation model and the transfer function. Results between simulation and hardware tests differed only slightly. Here, it was clear that lower τ_2 values decrease lock-time. Instability occurred with slightly lower κ_p values than predicted by the system model. In addition, the phase drift in the absence of a synchronization pulse was tested. The measured drift was 0.022 µs/s, which is 45x smaller than the PLL in FGC3.1.

All in all, this thesis has successfully verified the operation of the PLL in the new hardware environment of FGC3.2. It has also succeeded in characterizing the PLL by providing a transfer function and explaining how it was derived. It also discovered a nonideality in the hardware configuration that should be fixed for a future version. This study can be utilized to develop future platforms to observe how a change in hardware conditions affects the PLL operation.

References

[1] K. J. Åström and R. M. Murray, *Feedback Systems*. Princeton University Press, 2009, vol. 2.10b.

[2] C. Kao and B. Lincoln, "Simple stability criteria for systems with time-varying delays," *Automatica*, vol. 40, 2004.

[3] R. Jung, P. Komorowski, L. Ponce, and D. Tommasini, "The LHC 450 GeV to 7 TeV Synchrotron Radiation Profile Monitor using a Superconducting Undulator," *AIP Conference Proceedings*, December 2002, doi: 10.1063/1.1524404.

[4] F. Bordry, "Power converters for particle accelerators," in *2005 European Conference on Power Electronics and Applications*, 2005, doi: 10.1109/EPE.2005.219762.

[5] Y. Thurel, D. Nisbet, and B. Favre, "Four quadrant 120 a, 10 v power converters for lhc," in *2007 IEEE Particle Accelerator Conference (PAC)*, 2007, pp. 347–349, doi: 10.1109/PAC.2007.4440207.

[6] S. Page, C. Ghabrous Larrea, Q. King, B. Todd, S. Uznanski, and D. Zielinski, "FGC3.2: A new generation of embedded controls computer for power converters at CERN," in *Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems*, New York, NY, USA, October 2019. [Online]. Available: https://inspirehep.net/files/e6aed2e10ceb337015475baa779c39f2

[7] F. M. Gardner, *Phaselock techniques*, 3rd ed. Hobocken, NJ, USA: John Wiley & Sons, Inc., 2005.

[8] D. H. Wolaver, *Phase-Locked Loop Circuit Design*. Englewood Cliffs, NJ, USA: Prentice-Hall, Inc., 1991.

[9] R. E. Best, *Phase-Locked Loops: Design, Simulation and Applications*, 5th ed. USA: The McGraw-Hill Companies Inc., 2003.

[10] S. Page, Q. King, H. Lebreton, and P. Semanaz, "Migration from WorldFIP to a low-cost Ethernet fieldbus for Power Converter Control at CERN," in *Proceedings of the 14th International Conference on Accelerator and Large Experimental Physics Control Systems*, San Francisco, CA, USA, October 2013. [Online]. Available: https://accelconf.web.cern.ch/ICALEPCS2013/papers/tuppc096.pdf

[11] M. Magrans De Abril, Q. King, and R. Murillo-Garcia, "The Phase-locked loop Algorithm of the Function Generation Controller," in *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, Melbourne, Australia, October 2015. [Online]. Available: https://cds.cern.ch/record/2064389/files/CERN-ACC-2015-0136.pdf

[12] OpenCores, *Wishbone B4*, 2010, [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b4.pdf.

[13] L. Asplund et al. (2021) VUnit - unit testing framework for VHDL/SystemVerilog (Version 4.6.0) [Source code]. https://github.com/VUnit/vunit.

[14] C. Higgs et al. (2021) cocotb - Coroutine based Cosimulation Testbench for verifying VHDL and SystemVerilog RTL using Python (Version 1.6.0) [Source code]. https://github.com/cocotb/cocotb.

[15] M. Kumm, H. Klingbeil, and P. Zipf, "An FPGA-Based Linear All-Digital Phase-Locked Loop," *IEEE Transactions on Circuits and Systems I: Refular Papers*, vol. 59, no. 9, pp. 2487–2497, Sept. 2010, doi: 10.1109/TCSI.2010.2046237.

[16] S. Tancock, E. Arabul, and N. Dahnoun, "A Review of New Time-to-Digital Conversion Techniques," *IEEE Transactions on Instrumentation and Measurement*, August 2019, doi: 10.1109/TIM.2019.2936717.

[17] R. B. Staszewski and P. T. Balsara, *All-Digital Frequency Synthesizer in Deep-Submicron CMOS*. Hoboken, NJ, USA: Wiley-Interscience, 2006.

[18] V. F. Kroupa, *Direct digital frequency synthesizers*. John Wiley & Sons, 1998.

[19] R. C. Dorf and R. H. Bishop, *Modern Control Systems*, 12th ed. Pearson Education, Inc., 2011.

[20] D. D. Gajski and R. H. Kuhn, "New VLSI Tools," *IEEE Computer*, vol. 16, no. 12, December 1983.

[21] P. J. Ashenden, *Designer's Guide to VHDL*, 3rd ed. Elsevier Inc., 2008.

[22] T. Gingold et al. (2021) GHDL - VHDL 2008/93/87 simulator (Version 2.0.0-dev) [Source code]. https://github.com/ghdl/ghdl.

[23] L. Wang, Y. Chang, and K. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Burlington, MA, USA: Elsevier Inc., 2009.

[24] E. Seligman, T. Schubert, and M. Kumar, *Formal Verification*. Waltham, MA, USA: Elsevier Inc., 2015.

[25] T. Gingold et al. (2021) Cheby - File format and tools to describe and generate HW/SW interface (Version 1.5 (dev)) [Source code]. https://gitlab.cern.ch/cohtdrivers/cheby.

[26] M. Amagasaki and Y. Shibata, *Principles and Structures of FPGAs*, H. Amano, Ed. Springer, 2018.

[27] Xilinx, "7 Series FPGAs Data Sheet: Overview," September 2020, [Online] Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

[28] S. Wallentowitz (2021) cocotbext-wishbone (Commit d1f772b9e9087c8d7c2e6a2083981f71019228a8) [Source code]. https://github.com/wallento/cocotbext-wishbone.