Implementation of Costas Loop for BPSK Demodulation on a Microsemi FPGA

NIIT University

Project Work at ISAC, ISRO, Bangalore

Prabhpreet Singh Dua
U101112FEC101

July 12, 2016

Acknowledgements

The successful completion of this project work is due to the encouragement and guidance provided by a lot of people without whose help this would have been difficult. I wish to express my heartfelt gratitude to Mr. Yateendra Mehta, Group Director, CSG (Communication Systems Group), ISRO Satellite Centre and Dr. Parimal Mandke, Dean Academics, NIIT University for their support in doing the project work.

The execution of my project work would have been impossible without the constant encouragement and proficient guidance of my guide Mr. Aditya Ganesh, Scientist/Engineer SE, CSG. I am grateful for his mentorship in design work, testing and taking time out from his busy schedule to discuss and resolve a lot of my doubts. I would also like to thank Prof. Mandeep Goyal, NIIT University, for his mentorship during this project.

I express my sincere gratitude to Prof. Mandeep Goyal, NIIT University, Prof. Naveen Kapoor, HOD, Dept. Of ECE, NIIT University, and Prof. Vikas Upadhyay, NIIT University for inculcating strong academic fundamentals in their respective fields of Communication Systems, Control Theory and Digital Electronics to approach this challenging project.

Finally, I would like to thank my project partner Devanshi Upadhyay who was always helpful with the project. Thanks to all the individuals who have directly or indirectly helped me in the completion of this project.

Declaration

I hereby declare that the project work titled Implementation of Costas Loop for BPSK demodulation on a Microsemi Actel FPGA (Field Programmable Array) for carrier signal of 8/16 KHz and data rate of 2/4 kbps is an authentic record of my own work carried out at CSG, ISAC for fullfilling the partial requirement, Industry Practice, for the award of degree of B.Tech, NIIT University, Neemrana under the guidance of Mr. Aditya Ganesh and Prof. Mandeep Goyal during 11th January to 11th July,2016 and has not been submitted elsewhere for academic credit in any other institution.

PrabhpreetSDua

Prabhpreet Singh Dua

Dated: July 12, 2016

CERTIFICATE

This is to certify that the report for the project titled

"Implementation of Costas Loop for BPSK demodulation on a Microsemi Actel FPGA (Field Programmable Array) for carrier signal of 8/16 KHz and data rate of 2/4 kbps"

submitted by

PRABHPREET SINGH DUA

Enrollment No. U101112FEC101

is an authentic record of work carried out by him, under my mentorship and supervision at CSG, ISAC, in the partial fulfilment of the requirement, Industry Practice, for the award of degree of Bachelor of Technology in Electronics and Communication Engineering under NIIT University, Neemrana, Rajasthan.

[illegible]

Mr. Aditya Ganesh
Scientist/Engineer SE
Communication Systems Group
ISRO Satellite Centre
Bangalore

Abstract

The present telecommand system uses a BPSK demodulator onboard satellite for demodulating command signals. The BPSK demodulator requires a reference carrier coherent with the phase of the incoming signal for successful demodulation. Two ways of recovering the carrier are possible- one is to process a residual carrier using additional power for a conventional PLL to track. Another method is to reconstruct the carrier from the power of the data signal carrier itself. The Costas loop is such a system invented by John Costas in 1956 to achieve phase tracking, acquisition, synchronization and demodulation of double- band suppressed-carrier AM signals. Although the original intent of the Costas loop was to track and demodulate double- band suppressed-carrier AM signals, it can very well be used to demodulate other suppressed-carrier modulation techniques. Moreover, it can be readily used without modifications for BPSK demodulation.

In this project, a digital implementation of this varient of a PLL is implemented on an Actel Microsemi ProASIC3E FPGA.

Contents

List of Figures

Part I

Technical

Chapter 1

Introduction

1.1 Role and Organization

1.1.1 Role

As a project trainee under the Communication Systems Group at ISRO Satellite Center(ISAC), I was assigned to develop a BPSK Costas Loop Demodulator on an Actel Microsemi FPGA.

More information on the work performed is in the following sections of the report.

1.1.2 Organization Profile

ISRO is the space agency of the Indian government. Its vision is to "harness space technology for national development, while pursuing space science research and planetary exploration".

ISRO Satellite Centre (ISAC) Bangalore is the leading centre of Indian Space Research Organization (ISRO) for design, development, fabrication and testing of all Indian made satellites. ISAC's mission statement is that its "committed to total quality and zero defect in space systems and services."

ISAC is organised under Matrix Management Structure. The centre is divided into many Functional Areas like Control and Mission Area, Electronic Systems Area, Mechanical Systems Area, Reliability and Components Area etc.. Each Area is further divided into Groups. For instance, Control and Mission area has many groups like Control Systems Group, Flight Dynamics Group etc. Again Groups get divided into Divisions. For example, Control Systems Group has many divisions like Control Electronics Division, Control Dynamics and Analysis Division etc.

The Communication Systems Group (CSG) is a department at ISAC under the Communications and Power Area and is dedicated to developing and testing communication systems base for ISRO's satellites.

1.2 Problem definition

1.2.1 General Description

Digital communication systems on satellite systems consist of three functions:

- Telemetry: Where information goes from spacecraft to earth. Information is experimental data, system status data and imaging data.
- Command: Where information goes from earth to spacecraft and the information controls spacecraft functions.
- Tracking: Where signals from spacecraft are measured on earth to determine trajectory and other properties of spacecraft, the propogation medium and the properties of the solar system.

Modulation is the process of varying the phase, frequency or amplitude of a sinusoidal carrier using an information bearing signal. When the information signal is analog, then the phase,frequency or amplitude is changing at every instant of time. In digital communication, these signals are kept constant over a certain period of time and are discretized. Information is mapped (encoded) to discrete "symbols" using an encoder and then modulated on a carrier and sent through a medium, as illustrated in Figure 1.1. A demodulator is required to interpret back the sent symbols.

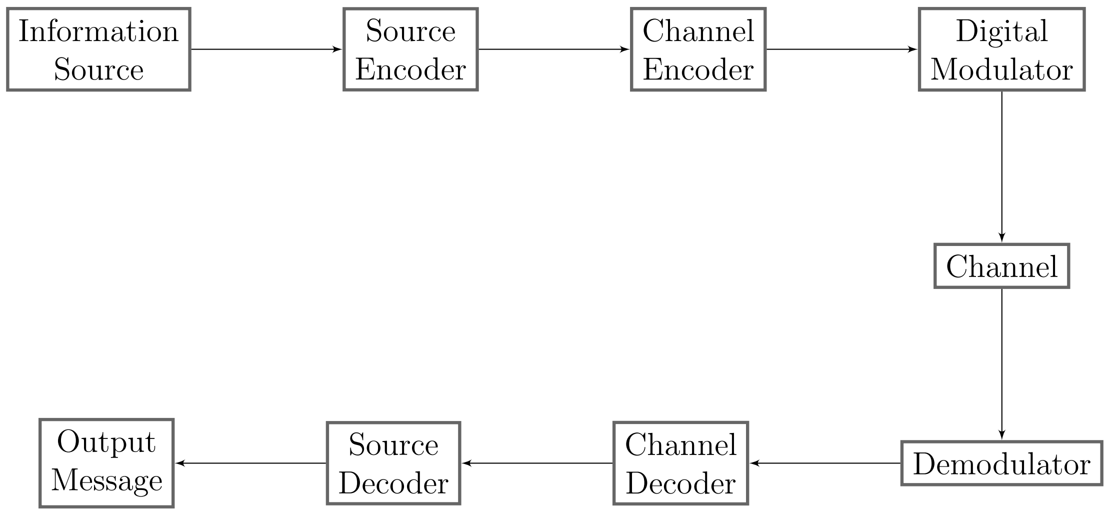


Figure 1.1: Block diagram of communication systems

The command function of satellite systems requires a demodulator on the satellite to obtain the symbols for decoding.

Since it's possible to change three parameters, i.e. amplitude, phase and frequency of a carrier, three digital modulation methods arise: Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK), and Phase Shift Keying (PSK). Amplitude shift keying (ASK) which represents digital data as the variations in amplitude of the carrier wave is not often used due to high noise susceptibility. Frequency shift keying (FSK), in which a finite number of frequencies are utilized for representing the digital data, has been predominantly used for uplink in Spacecraft Telecommand System due to the simplicity in demodulator realization. However, because of the better spectral efficiency, the new generation spacecrafts are currently using Phase Shift Keying(PSK) even though the implementation complexity is higher than that of ASK and FSK. Developed during the early days of the deep space programs, phase-shift keying now finds widespread use in both military and commercial communication systems. For telemetry applications, PSK is considered an efficient form of data modulation because it provides the lowest probability of error for a given received signal level, when measured over one symbol period. Terrestrial microwave radio links and satellite communication systems also frequently employ PSK as their modulation format.

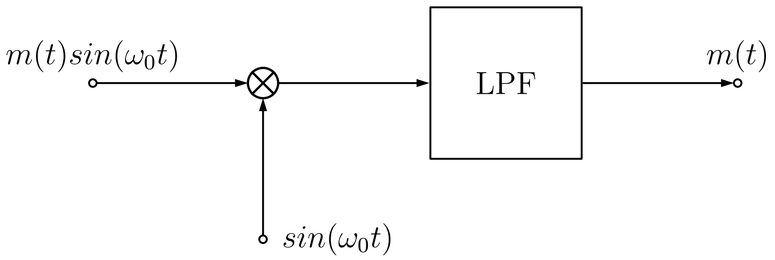


Figure 1.2: A BPSK demodulator

1.2.2 Phase Shift Keying

Phase-shift keying (PSK) is a modulation process whereby the input signal, a binary PCM waveform, shifts the phase of the output waveform to one of a fixed number of states.

A M-ary PSK modulated signal is described as

$$v_o(t) = \sqrt{2P} sin(\omega_0 t + \frac{2\pi(i-1)}{M}) \tag{1.1}$$

where $i = 1, 2, ..., M$, P is the average signal power over the symbol period, T_S, ω_0 is the carrier frequency, and $M = 2^N$ are the number of allowable phase states for any natural number N.

PSK requires that a copy of the carrier be at the reciever to demodulate the signal, i.e. the phase of the carrier and the copy at the reciever should be *coherent* with each other. To generate a coherent carrier, synchronous communication techniques are used. An example of a BPSK (where binary symbols are encoded as ± 1) demodulator is shown in Figure 1.2 where the exact copy of $sin(\omega_0 t)$ is required for proper demodulation.

An early synchronous communication scheme was the system of a residual carrier where some of the power of the transmitted signal was used solely for transmitting a reference carrier signal so it could be replicated through something called a Phase Locked Loop (PLL). A PLL essentially is a negative feedback control system operating in the phase domain which aims to minimize the phase difference between the reference signal and the carrier it generates.

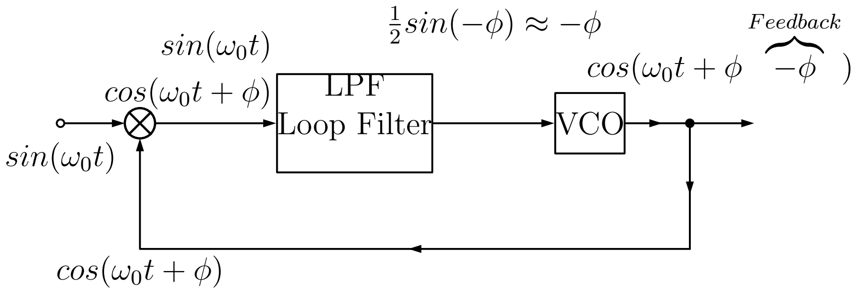


Figure 1.3: PLL operating on a residual carrier in acquisition mode

The equation of a reference carrier is given by:

$$y(t) = \sqrt{2P} sin(\omega_0 t + \theta_m m(t)) \tag{1.2}$$

$$= \underbrace{\sqrt{2P} cos(\theta_m) sin(\omega_0 t)}_{ResidualCarrier} + \underbrace{\sqrt{2P} sin(\theta_m m(t)) cos(\omega_0 t)}_{Data} \tag{1.3}$$

The power of the residual carrier and data signal can be varied by choosing θ_m such that $0 < \theta_m < \frac{\pi}{2}$. An illustration on how a PLL operates on a residual carrier is given in Figure 1.3. Here, the PLL tries tries to minimize ϕ by giving a $-\phi$ feedback so there is zero phase difference. When the PLL is trying to minimize the phase difference, it is said to be in acquisition mode. When the phase difference is minimized, the phase is said to be in lock, and enters tracking mode.

However, there are a couple of problems with this approach. Firstly, a significant amount of power is dedicated to sending the residual carrier which could be used for increasing the data signal's signal to noise ratio (SNR). The performance of a system, i.e. the symbol error rate reduces if the power of the data signal transmitted is high. A second problem is that the power of the data signal acts as noise at the input of the given signal, and thus reduces the ability of the PLL to track the residual carrier component.

1.2.3 Costas Loop

In 1956, J. Costas introduced a technique to recover the carrier directly from the modulated signal itself [3]. It is called Costas Loop. It has the property

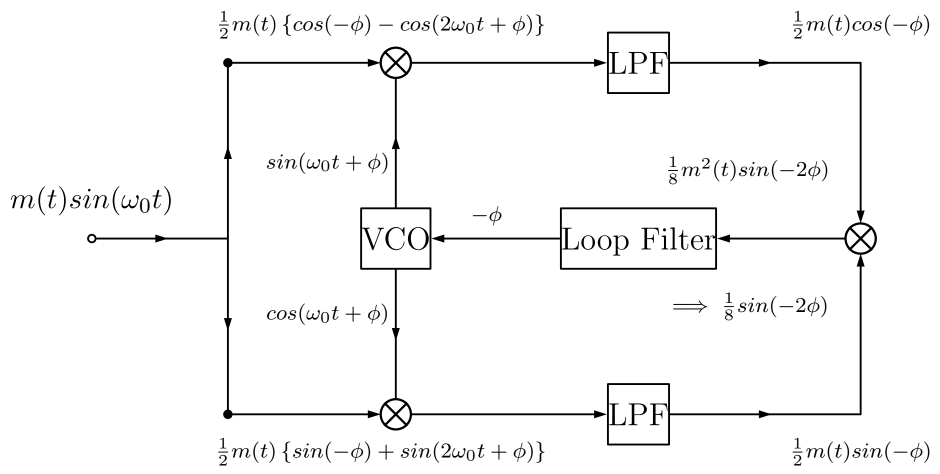

$\frac{1}{2}m(t)\{cos(-\phi) - cos(2\omega_0 t + \phi)\}$
LPF
$\frac{1}{2}m(t)cos(-\phi)$
$sin(\omega_0 t + \phi)$
$\frac{1}{8}m^2(t)sin(-2\phi)$
$m(t)sin(\omega_0 t)$
VCO
$-\phi$
Loop Filter
$cos(\omega_0 t + \phi)$
$\Longrightarrow \frac{1}{8}sin(-2\phi)$
LPF
$\frac{1}{2}m(t)\{sin(-\phi) + sin(2\omega_0 t + \phi)\}$
$\frac{1}{2}m(t)sin(-\phi)$

Figure 1.4: Costas Loop in acquisition mode [1]

of being able to derive a carrier from the received signal, even when there is no component at carrier frequency present in that signal (eg: DSBSC). The requirement is that the amplitude spectrum of the received signal be symmetrical about this frequency. In Fig. 1.4, Costas Loop is shown in acquisition mode for a BPSK supressed carrier signal (when $\theta_m = 0$, i.e. no signal power is dedicated to the carrier). The Costas loop is based on a pair of quadrature modulators - two multipliers fed with carriers in phase-quadrature (a $\frac{\pi}{2}$ shift).These multipliers are in the in-phase (I) and quadrature phase (Q) arms of the arrangement. Each of these multipliers is part of separate synchronous demodulators. The outputs of the modulators, after filtering, are multiplied together in a third multiplier, and the low pass components in this product are used to adjust the phase of the local carrier source - a VCO - with respect to the received signal. In the loop, we can see that the error input to the VCO $-\phi$ tells the VCO to shift its phase difference back to achieve synchronization. The operation is to maximize the output of the I arm, and minimize that from the Q arm.

Interestingly enough, the 'I' arm of a Costas Loop can also be used to demodulate a BPSK signal directly while in lock, as shown in Fig. 1.5.

An optimized design of the Costas Loop is essential as an optimal design can demodulate information with much lesser power. Since attenuation is large for satellite links, it can save huge amounts of transmission power

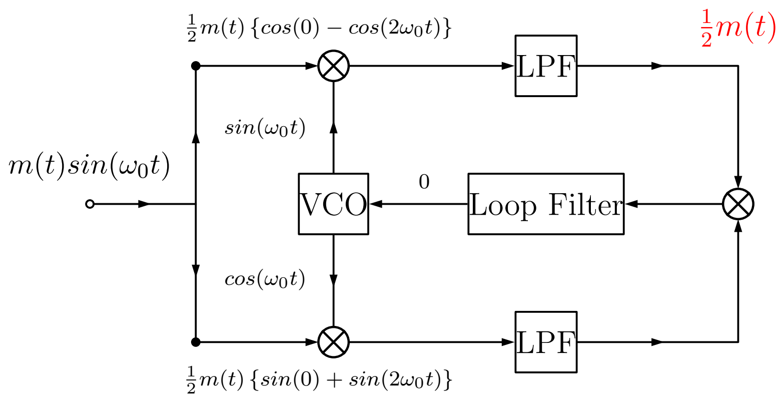

$\frac{1}{2}m(t)\,\{cos(0) - cos(2\omega_0 t)\}$
$\frac{1}{2}m(t)$
LPF
$sin(\omega_0 t)$
$m(t)sin(\omega_0 t)$
VCO
0
Loop Filter
$cos(\omega_0 t)$
LPF
$\frac{1}{2}m(t)\,\{sin(0) + sin(2\omega_0 t)\}$

Figure 1.5: A locked Costas Loop

required for uplink and therefore saves costs.

1.2.4 FPGAs

In this project, the Costas Loop will be implemented on a Field Programmable Gate Array. FPGAs are digital circuits consisting of huge numbers of logic blocks and routes interconnecting these blocks which can be customized by changing FPGA's in-built memory to create programmable digital circuits. They are particularly useful in fields where the demand for digital chips with certain function is low. Since fabrication of a chip is expensive, FPGAs are used to meet the niche requirements in a cost effective manner in industries such as defense, space and aeronautics. Actel's Microsemi ProASIC3E FPGA chip is particularly used here because the memory used to configure the digital circuits in the FPGA is tolerant to effects of electromagnetic radiation frequently encountered in space.

1.3 Objectives and Scope

In this project, it was necessary to meet the primary objectives and the secondary objectives were optional. They were as follows:

1.3.1 Primary Outcome

To implement supressed carrier BPSK demodulation using Costas Loop on a Microsemi Actel FPGA (Field Programmable Array) for 8/16KHz carrier signal and data rate of 2/4 kbps.

1.3.2 Secondary Outcomes

The secondary outcomes were the following features in the Costas Loop:

- Lock indicator: To indicate that phase lock has been achieved
- Infinite Impulse Response filter implementation (instead of FIR filters)

Chapter 2

Literature Review and Analysis

2.1 Preliminary topics

2.1.1 Fixed Point Representation

Numbers have precision limited by the bit word length in digital systems. To represent decimal valued numbers instead of integers in a digital system, fixed point representation is used. A fixed point representation is simply a base 2 system where the decimal is applied after a particular bit for reference. As with base 2 system numbers, fixed point numbers are of two types:

- Unsigned: Represented by $UQm.n$
- Signed: Represented by $Qm.n$

In an unsigned fixed point number $UQm.n$ of length $m+n+1$, the bit word $< a_m ... a_2 a_1 a_0 a_{-1} ... a_{-n} >$ is represented in decimal as:

$$a_m 2^m + a_{m-1} 2^{m-1} + a_{m-2} 2^{m-2} + ... + a_0 2^0 + a_{-1} 2^{-1} + ... + a_n 2^{-n} \qquad (2.1)$$

For a signed fixed point number $Qm.n$ of length $m+n+1$, the bit word $< s a_{m-1} ... a_2 a_1 a_0 a_{-1} ... a_{-n} >$ is represented in decimal as:

$$s(-2)^m + a_{m-1}2^{m-1} + a_{m-2}2^{m-2} + ... + a_0 2^0 + a_{-1}2^{-1} + ... + a_n 2^{-n} \quad (2.2)$$

2.2 Costas Loop

2.2.1 Analog Linearized Model of Costas Loop

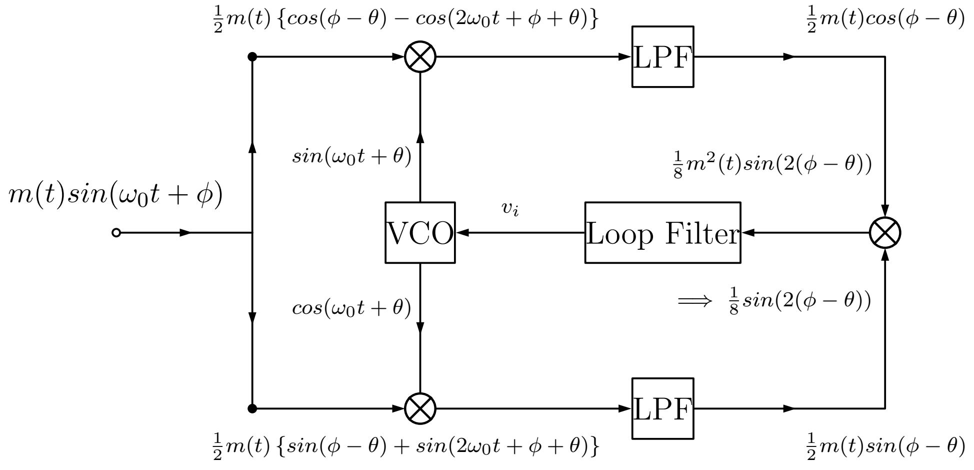

$\frac{1}{2}m(t)\{cos(\phi-\theta) - cos(2\omega_0 t + \phi + \theta)\}$
LPF
$\frac{1}{2}m(t)cos(\phi-\theta)$
$sin(\omega_0 t + \theta)$
$\frac{1}{8}m^2(t)sin(2(\phi-\theta))$
$m(t)sin(\omega_0 t + \phi)$
VCO
v_i
Loop Filter
$cos(\omega_0 t + \theta)$
$\Longrightarrow \frac{1}{8}sin(2(\phi-\theta))$
LPF
$\frac{1}{2}m(t)\{sin(\phi-\theta) + sin(2\omega_0 t + \phi + \theta)\}$
$\frac{1}{2}m(t)sin(\phi-\theta)$

Figure 2.1: Costas Loop in acquisition mode

As introduced earlier, a Costas Loop essentially is a special type of PLL used to recover a supressed carrier. A Costas Loop consists of the following components:

- Voltage Controlled Oscillator
- Multiplier
- Arm Filters
- Loop Filter

By linearizing its components by analyzing the function when $\phi - \theta \approx 0$, we can express the model as an LTI system between input ϕ and θ and use the existing classical control theory framework to analyze its performance (similar to what is done for an analog PLL).

The function of these components and their linearized model is described in the upcoming subsections.

Voltage Controlled Oscillator

A voltage controlled oscillator generates a sinusoid centered at an acquiesent frequency ω_0 with frequency shift $\omega_{shifted}$ proportional to input voltage, i.e.

$$\begin{aligned} &\omega_{shifted}(t) \propto v_i(t) \\ &\implies \frac{d\theta}{dt} = k_v v_i(t) \\ &\implies v_o(t) = sin(\omega_0 t + \int_0^t v_i(u)du) \implies v_o(t) \quad = sin(\omega_0 t + \theta) \end{aligned}$$

In Costas Loop, the VCO produces a sinusoid with a $\frac{\pi}{2}$ phase difference between the two arms of the loop.

In our linearized control system, the VCO assumes the role of the plant and its transfer function between θ and v_i can be given as

$$\begin{aligned} \frac{d\theta}{dt} &= k_v v_i(t) \\ \frac{\theta(s)}{v_i(s)} &= \frac{k_v}{s} \end{aligned} \tag{2.3}$$

Arm Filters

The arm filters are low pass filters used for removing the $2\omega_0$ sinusoid component from the mixer outputs. Either FIR (finite impulse response) or IIR (infinite impulse response) LTI filters can be used. FIR filters are used because of their linear phase characteristics, but are typically of a higher order than their IIR magnitude response counterparts. Typically, the higher the order of the filter, the closer is their response to ideal low pass filter magnitude frequency response characteristics, but at the cost of more hardware complexity.

It is important that the two filters have the exact same frequency response in order for the loop to function properly. This is why a digital implementation of the filters is preferred.

In later sections, we will see that the design of arm filters is crucial to the performance characteristics of a Costas Loop, as it defines the squaring loss, S_L in the loop SNR.

The arm filter assumes ideal operation in the linearized model, and it's operation is assumed in the phase detector.

Multiplier and the phase detector

The multipliers do a simple job- they multiply two signals. There are three multipliers in the loop. These multipliers and the arm filter in the linear model are approximated to the phase detector, which generates the error term input into the loop filter when $\phi - \theta \approx 0$. As described in Fig. 2.1, that component is

$$\begin{aligned} &\frac{1}{8}m^2(t)sin(2(\phi-\theta)) \\ \implies &\frac{1}{8}sin(2(\phi-\theta)) \approx \frac{1}{4}(\phi-\theta) \\ &\implies k_p(\phi-\theta) \end{aligned} \tag{2.4}$$

where k_p is the gain of the phase detector.

Loop Filter and the linearized model

The loop filter plays a key role in the loop. In our linearized model, it assumes the role of the controller and therefore defines the transient and steady state parameters of the loop.

Since the role of the loop filter is so essential to our linearized model, we will discuss it in this subsubsection itself. From approximations (Equations

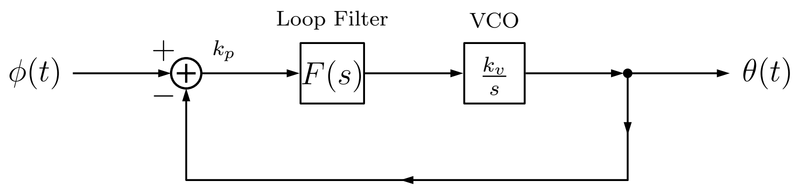


Figure 2.2: Linearized model of a Costas Loop

2.3, 2.4) from the previous sections, Figure 2.2 illustrates the linearized model of Costas Loop in phase domain. To recall, this is applicable only when $\phi - \theta \approx 0$.

The open loop transfer function and the closed loop transfer function of the loop respectively are given by

$$G(s) = \frac{k_p k_v F(s)}{s} \tag{2.5}$$

$$H(s) = \frac{\theta(s)}{\phi(s)} = \frac{k_p k_v F(s)}{s + k_p k_v F(s)} \tag{2.6}$$

Here, $F(s)$ is the loop filter and $\frac{k_v}{s}$ is the linearized loop filter.

For a LTI control system, the number of poles at zero (i.e. integrators) in a unit negative feedback's open loop transfer function $G(s)$ determine what kind of response the system can follow (track) with a steady state error and which it can replicate (acquire) when reaching steady state. This property is known as the type of the system. In PLLs, this property essentially indicates whether the system can track phase (Type 0), can acquire phase and follow frequency deviation (Type I), or acquire phase and frequency deviation and follow a constant change in frequency (Type II) [4].

Our desired system is a Type II Costas Loop which can acquire both phase and frequency. We use a PI controller, which along with the VCO integrator becomes a type II system, i.e.

$$F(s) = \alpha + \frac{\beta}{s}, \alpha, \beta > 0 \tag{2.7}$$

$$\implies G(s) = k_p k_v \frac{\alpha s + \beta}{s^2} \tag{2.8}$$

Let $K = k_p k_v \alpha$ and call it the loop gain and $K^{'} = \frac{\beta}{\alpha}$. Then,

$$G(s) = K\Big(1 + \frac{K^{'}}{s}\Big) \tag{2.9}$$

Since there are two poles at zero in the open loop transfer function, this is a type II system.

The closed loop transfer function therefore becomes

$$H(s) = \frac{K(s + K^{'})}{s^2 + Ks + KK^{'}}$$

Here,

$$\omega_n = \sqrt{KK'} \tag{2.10}$$

$$\xi = \sqrt{\frac{K}{4K'}} \tag{2.11}$$

where ω_n is the natural frequency of the second order system and ξ is the damping ratio [5].

The root locus for a fixed value of $K^{'}$ is

Since the root locus never goes to the left half plane, the loop in the analog domain is always stable for all values of gain.

By varying different values of $K^{'}$, the root locus looks like,

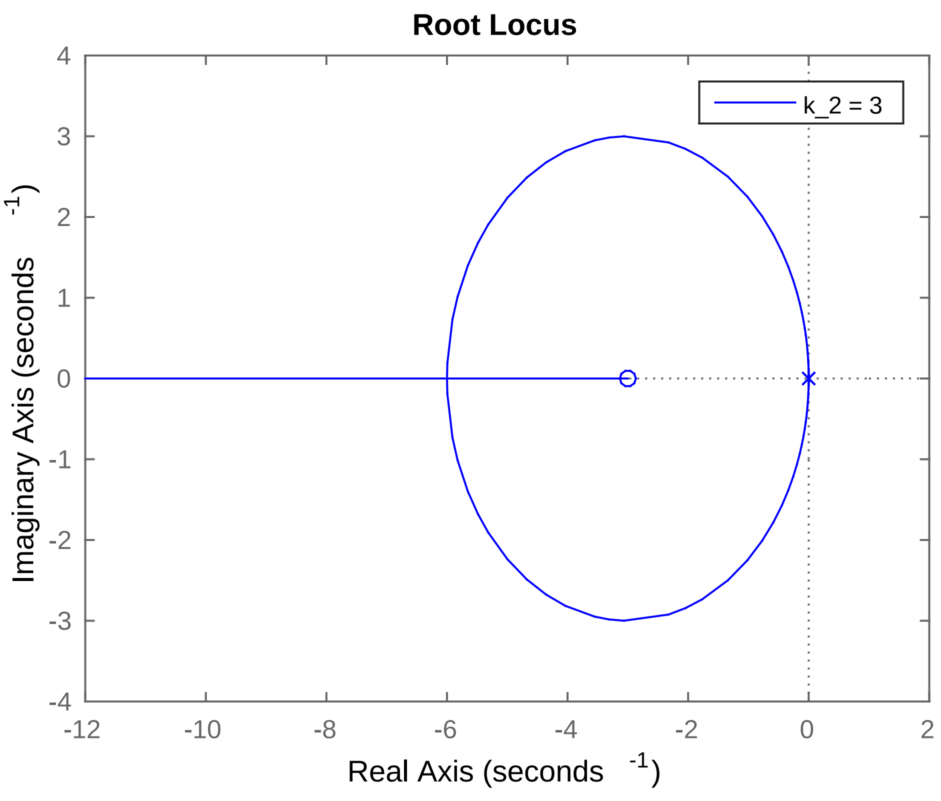


Figure 2.3: Root Locus of PI controller for fixed $K^{'}$

2.2.2 Digital implementation of Costas Loop

Digital Implementation of the Costas Loop is acquired by shifting the linearized model from the s domain to the z domain.

The digital equivalent of the components are:

- Numerically Controlled Oscillator: Equivalent of VCO
- Arm Filters: Lowpass filters in digital domain
- Loop Filters: PI controllers in digital domain

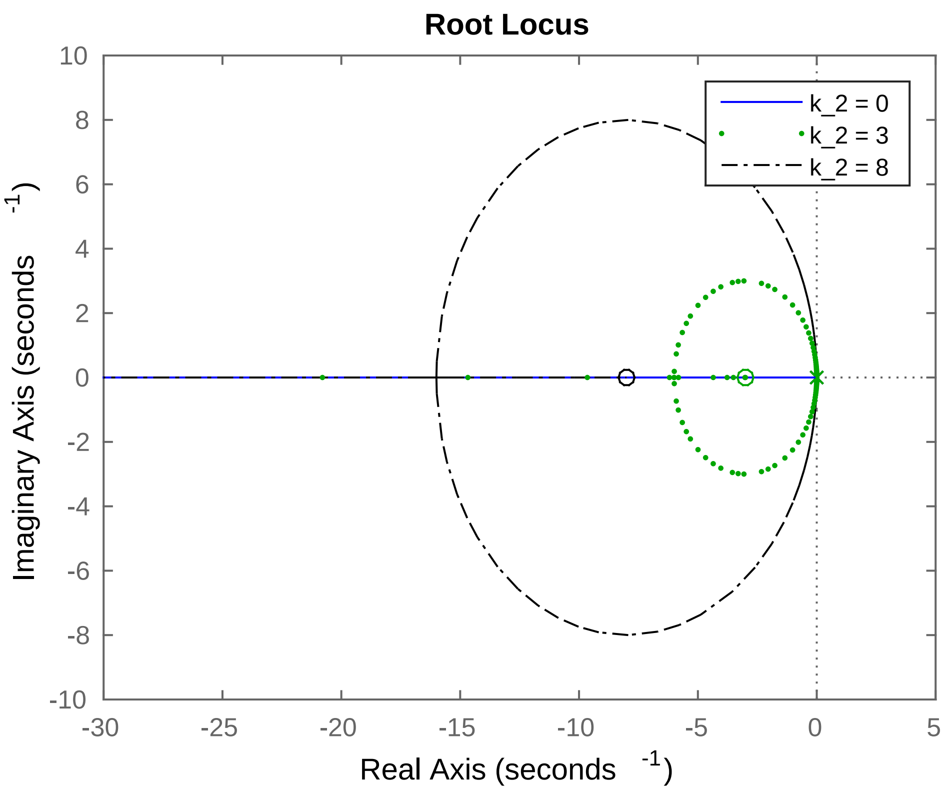


Figure 2.4: Root Locus of PI controller for varying K'

Numerically Controlled Oscillator

A Numerically Controlled Oscillator (NCO) is the digital equivalent of a VCO. Here the input in given by a bit length N frequency word, and output is given by bit length P word. Though other architectures do exist for an NCO, we are going to use the Digital Direct Synthesis (DDS) architecture for the NCO illustrated in Fig. 2.5. In this architecture of an NCO usually comprises of two parts:

- Phase Accumlator (PA): A simple counter which adds the input word at every clock cycle, accumlating phase.
- Phase to Amplitude Converter (PAC): Maps phase accumalator output to an amplitude value.

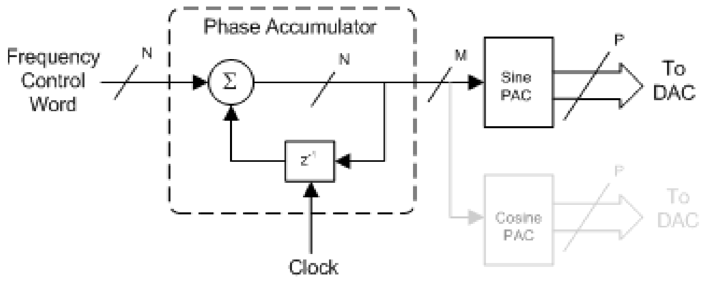


Figure 2.5: DDS architecture for NCO

The phase accumlator has many types of implementations and optimizations. The most common one is a Look Up Table (LUT) containing samples for the lowest frequency input possible.

The LUT here is going to have the sine waveform. Due to the symmetry of the sine waveform, we only need the sample $\frac{1}{4}$ of the whole period.

The minimum frequency resolution is given by:

$$F_{res} = \frac{F_{clock}}{2^N} \tag{2.12}$$

The operating frequency for frequency word ΔF is given by

$$F_{operating} = \Delta F \frac{F_{clock}}{2^N} \tag{2.13}$$

Certain spurious products arrive out of phase truncation and amplitude truncation. Phase truncation can be avoided by $M = N$ and amplitude truncation effects can be improved by increasing the bit length.

The difference equation of the phase accumalator, assumping phase is represented in $cycles^{-1}$ i.e. $[0, 2\pi]rad \implies [0, 1)cycles^{-1}$

$$\theta[n] = \{k_v\theta[n-1] + \epsilon[n]\}\, mod(1)$$

To use it in our linearized model, the non-linear mod component needs to be ignored for now. Therefore,

$$\begin{aligned} \theta[n] &= k_v\theta[n-1] + \epsilon[n] \\ \frac{\theta(z)}{\epsilon(z)} &= \frac{k_v z^{-1}}{1 - z^{-1}} \end{aligned} \tag{2.14}$$

Arm Filters

The goal of the arm filters is to eliminate the $2\omega_0$ frequency but it should also allow the data modulated component. Please refer to Section 2.3 for filter design implementation.

Since arm filters are going to placed in a pipeline-like architecture, it is going to add a delay to an input data sample. In our linear model, we assume an ideal lowpass arm filter and therefore we abstract it to a delay

$$\begin{aligned}\phi_d[n] &= \phi[n-1] \\ \frac{\phi_d(z)}{\phi(z)} &= z^{-1}\end{aligned} \tag{2.15}$$

Loop Filter

As with its analog counterpart, the loop filter is the most critical component of the design of a costas loop. We will extend the linear model in the analog domain with a digital controller and plant, as illustrated in Fig. 2.6.

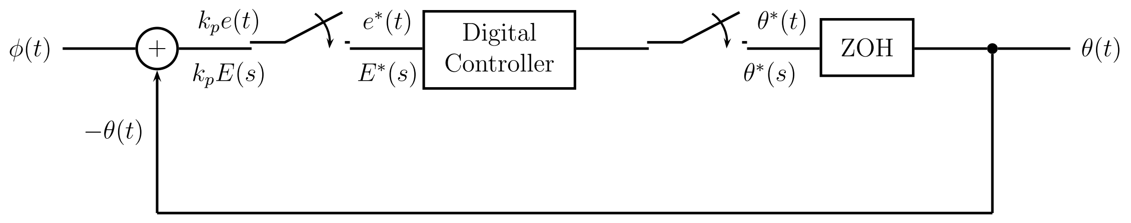


Figure 2.6: Digital linear model in analog domain

Here, the digital controller consists of the arm filter delay, Loop filter and NCO (Equations 2.15, 2.14). That is given in z-domain by

$$D(s) = \frac{\theta(z)}{E(z)} = \frac{k_p k_v z^{-2} F(z)}{1 - z^{-1}} \tag{2.16}$$

ZOH stands for zero order hold and is given by the transfer function

$$G_{oh}(s) = \frac{1 - e^{-Ts}}{s} \tag{2.17}$$

where T is the sampling time. k_p is the gain of the phase detector. By applying pulse transform to the analog model [6],

$$\frac{\theta^*(s)}{\phi^*(s)} = \frac{G^*_{oh}(s)D^*(s)}{1 + G^*_{oh}(s)D^*(s)}$$

and substituting $z = e^{Ts}$,

$$\frac{\theta(z)}{\phi(z)} = \frac{k_p k_v z^{-2} F(z)}{1 - z^{-1} + k_p k_v z^{-2} F(z)} [\because G_{oh}(z) = 1] \tag{2.18}$$

For tracking frequency step, we need two integrators in the open loop transfer function. Therefore, we use a digital equivalent PI controller,

$$F(z) = k_1 \left(1 + k_2 \frac{z^{-1}}{1 - z^{-1}}\right) \tag{2.19}$$

Let $k = k_p k_v k_1$. Then, as illustrated in Fig. 2.7,

$$\frac{\theta(z)}{\phi(z)} = k \frac{k_2 + z - 1}{z\left(z - 1\right)^2} \tag{2.20}$$

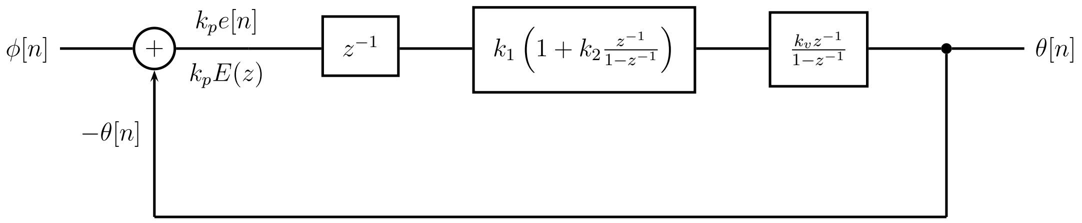

$\phi[n]$
$+$
$k_p e[n]$
$k_p E(z)$
$-\theta[n]$
z^{-1}
$k_1\left(1 + k_2 \frac{z^{-1}}{1-z^{-1}}\right)$
$\frac{k_v z^{-1}}{1-z^{-1}}$
$\theta[n]$

Figure 2.7: Digital linear model in z domain

For a stable system in the z-domain, all poles must reside inside the unit circle. By obtaining the characteristic equation (same as its analog counterpart),

$$k\ (k_2 + z - 1) + z\left(z - 1\right)^2 = 0 \tag{2.21}$$

we can analyze the stability and root locus. By analyzing inequalities obtained using Jury Stability Test, (calculations in Appendix A), we can find the values for which the system is stable in Fig. 2.8 where x-axis is k, and y-axis is k_2. Unlike the analog PI controller, the system is unstable for all $k > 0$ if $k_2 > 0.5$ and k is limited to $0 < k < 1$ for the system to be stable.

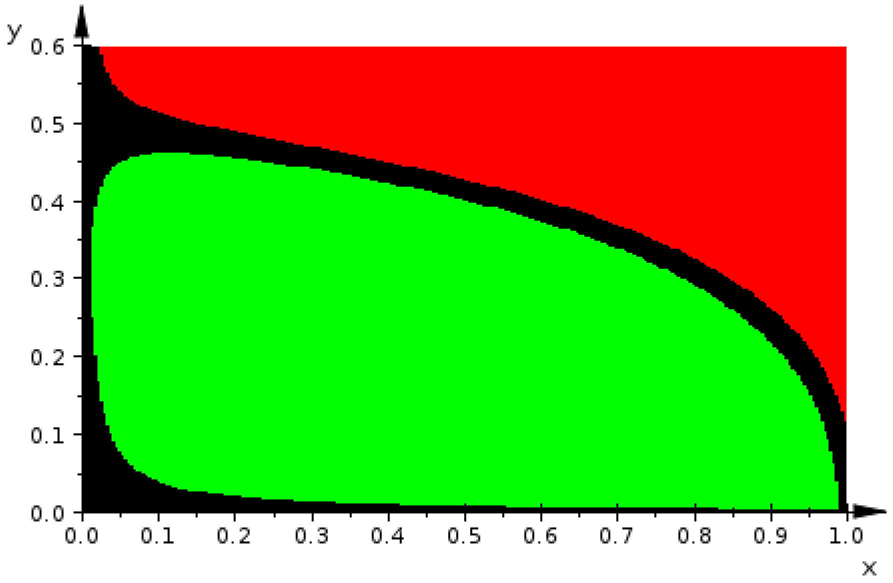


Figure 2.8: Stability analysis of a PI controller in DPLL

Observing the root locus by varying values of k_2 in Fig. 2.9, we observe three cases. For $k_2 < \frac{1}{9}$, the root locus has two real poles till some value and then they become complex conjugates. For $k_2 = \frac{1}{9}$, one pole is real until all poles merge at one value on the real axis. For $\frac{1}{9} < k_2 < \frac{1}{2}$, two poles are always complex conjugates of each other, and one pole is always on the real axis.

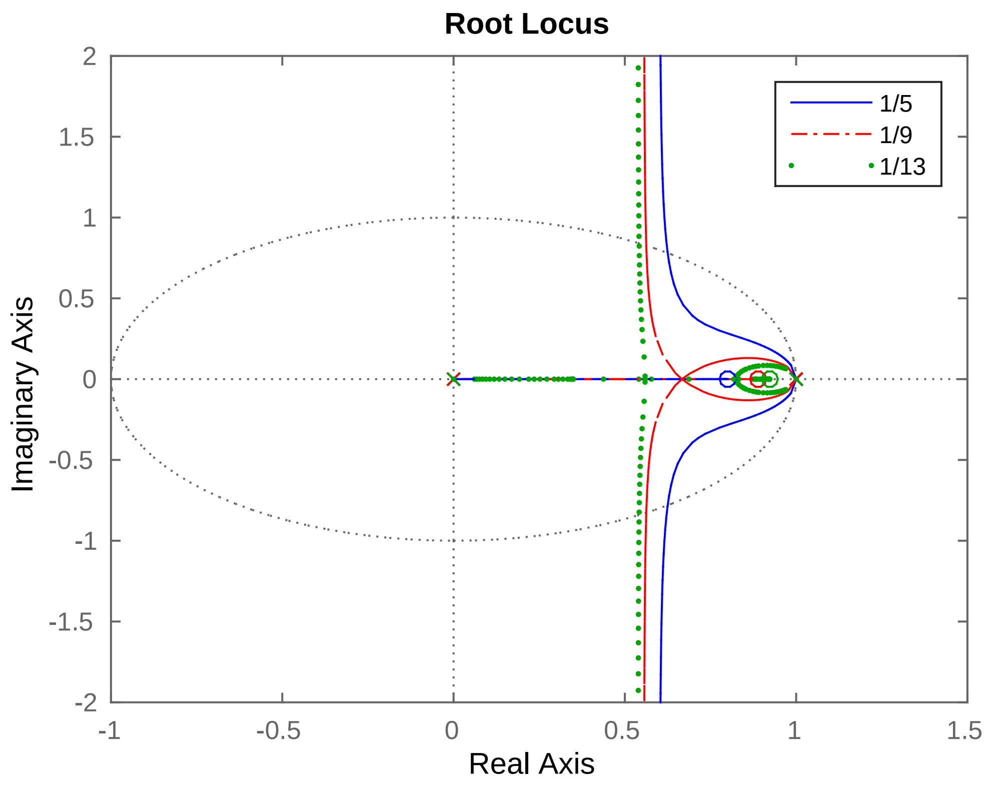


Figure 2.9: Root locus of DPLL PI controller

2.3 Practical Filter Design

2.3.1 Filter Design Overview

Filters essentially are systems used to remove unwanted frequency components in a signal, and perhaps enhance wanted ones. A filter which removes high frequency signals and allows passing of lowpass signals is a low pass filter. In the implementation of this project, both the arm filters and the loop filter are low pass filters.

We will use Linear Time Invarient systems as filters since they are easier to analyze, don't introduce unwanted harmonics, and have the property of multiplication of their transfer functions in the frequency domain ($Y(\omega) = H(\omega)X(\omega)$, $H(\omega)$ being the system here effectively acting as a weighting function or a spectral shaping function on the input signal's spectrum, $X(\omega)$. Also, our system must be causal, i.e. it shouldn't require values from the future to determine the current output since our the filter needs to operate in real time, and here we cannot have future values. The system input/output equation of an causal LTI system in discrete time domain is:

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k), \forall a_k, b_k \in \mathbb{R} \tag{2.22}$$

These filters are classified based on their impulse response (i.e. the value of $y(n)$ if $x(n) = \delta(n)$. These are:

- Finite impulse response: $\exists y[n] = 0$ for $n > n_1 \& n < n_2$. Also, in Eq. 2.22, $y[n-k]$ terms don't exist.
- Infinite impulse response: $y[n]$ is infinite.

The causal LTI digital filter design consists of two processes: the approximation problem and the realization problem [7].

Approximation problem

To design a filter, we must know the characteristics of our ideal filter, and what deviations from ideal behaviour we may allow in terms of some standard metrics. The approximation problem is to find the best filter transfer function given these parameters. The characteristics of an ideal filters are:

- Constant gain in passband, Zero gain in stopband
- Linear phase response: All frequencies components must be delayed by the same amount, i.e. no distortion

In terms of low pass filter, the ideal frequency response is:

$$H_{LP}(\omega) = \begin{cases} Ce^{-j\omega n_0}, & \omega < \omega_1 \\ 0, & \text{otherwise} \end{cases} \tag{2.23}$$

The deviations allowed from these characteristics are illustrated in Fig. 2.10, and given in terms of the following parameters:

- Passband: $[0, \omega_p]$
- Stopband: $[\omega_s, \infty]$
- Transition band: $[\omega_p, \omega_s]$
- Passband ripple: $R_p = -20log_{10}\frac{1-\delta_p}{1+\delta_p}$
- Stopband ripple: $A_s = -20log_{10}\frac{\delta_s}{1+\delta_s}$

Realization problem

The realization problem is about choosing a structure to implement the transfer function, i.e. the implementation of the filter. There are different methods for FIR and IIR filters.

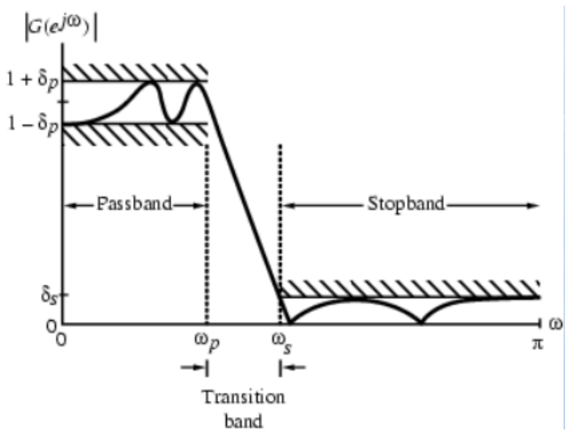


Figure 2.10: Filter design parameters for a LPF

The reason the structure matters is because of finite configurable logic blocks on an FPGA. A structure using less resources leaves space for other features to be impplemented on the FPGA.

On an FPGA, numbers need to be represented in binary, and can be represented only in a finite number of bits. Conserving bit length means more CLBs remaining for other hardware on the FPGA. Performance issues called finite word effects arise from quantization of input amplitude and filter coefficients that result in problems such as quantization noise. Such effects are analyzed and compensated for in addressing the realization problem.

2.3.2 FIR Filter Design

Approximation problem

An FIR filter is expressed as:

$$y(n) = \sum_{k=0}^{M} b_k x(n-k), \forall b_k \in \mathbb{R} \tag{2.24}$$

In the frequency domain, the equation is a polynomial:

$$Y(\omega) = \sum_{k=0}^{M} b_k e^{-j\omega k}, \forall b_k \in \mathbb{R} \tag{2.25}$$

To meet the ideal low-pass requirement, they must also meet linear phase response, although it is possible to design FIR filters with non linear phase which are superior in terms of matching the magnitude response requirement.

For a low pass linear phase filter, the coefficients have to be even symmetric[8], i.e.

$$b_k = b_{M-k}, \forall 0 \leq k \leq M \tag{2.26}$$

As discussed in the previous subsection, approximation response is based on some parameters a system can tolerate. In FIR filter design, the transition width and peak passband/stopband ripple have to be specified, and the order of the filter has to be set. Each of these parameters has a tradeoff with another parameter. A metaphor described in reference [9], describes the filter design tradeoffs as to think of each specification as one of the angles in a triangle as in Figure 2.11:

> The metaphor is used to understand the degrees of freedom available when designating design specifications. Because the sum of the angles is fixed, one can at most select the values of two of the specifications. The third specification will be determined by the design algorithm specified.

Therefore, by fixing two parameters, by adjusting the third parameter, Eq.2.25 can be approximated to a linear optimization (or linear programming) problem where a cost function comparing the filter frequency response to the ideal frequency repsonse can be minimzed.

For example, if the transition width and the filter order are fixed, the following cost function is to be minimized by adjusting the passband/stopband ripple under a certain region:

$$E(\omega) = W(\omega)[H(\omega) - H_{LP}(\omega)] \tag{2.27}$$

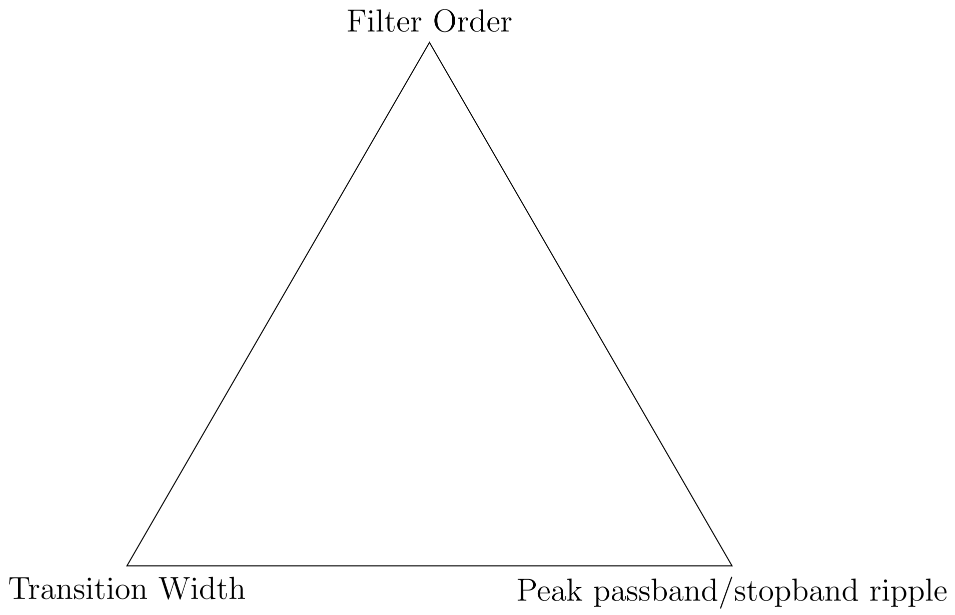


Figure 2.11: FIR design specifications represented as a triangle

where $W(\omega)$ is a weighting function used to allow for different peak ripples in the passband and stop band. There are different measures to determine the size of $E(\omega)$. Two important measures are used:

- Minimizing the maximum error between ideal and actual filters: Equiripple filters
- Minimize the energy of the error between ideal and actual filters: Least squares design filters

This method is superior to other methods of designing FIR filters, i.e. through windowing, etc. MATLAB's Filter Design Toolbox provides all the tools for visualizing and designing such filters.

Realization problem

There are two problems that arise due to a finite word length in FIR filters [7]:

- Errors in representing coefficients as finite fixed-point numbers
- Errors due to finite-precision arithmetic operations of addition, multiplication and storage which make the filter a non-linear system

The errors in representing coefficients are resolved by using algorithms that optimize quantized coefficients with the idealized response.

Non linear operation of quantization leads to quantization noise. This operation can be linearized to an additive noise process in which the noise is uniformly distributed [7]. Therefore a signal to noise ratio can be calculated when a input signal $v \leq$ with scaling factor G and a word with bit length B is given as:

$$SNR = 10log[E(Gv)^2] + 20Blog2 + 4.77 \tag{2.28}$$

The conclusion from this equation is that the value of SNR depends on the value of G and B. The increase in scaling factor G does increase the SNR but it also increases the probablity of overflow. Thus, the tradeoff in the value of G needs to be managed.

Quantization noise also shows up in operations like truncation of bit words after accumalation or output, which reduces the accuracy of the system. Therefore, the structure in an FPGA needs to be designed by keeping the parameters of scaling and bit word length in mind.

Two structures exist for FIR filters: the direct structure and the transpose structure. Since both the structures use the same number of multiplier and accumalators and we are free to set the bit word length in an FPGA, any implementation can be used.

2.3.3 IIR Filter Design

Approximation problem

The difference equation for an Nth order IIR filter is given by

$$y(n) = -\sum_{k=1}^{N} a_k y(n-k) + \sum_{k=0}^{M} b_k x(n-k), \forall a_k, b_k \in \mathbb{R} \tag{2.29}$$

Unlike FIR filters, IIR filters are not able to achieve linear phase response characteristic. Moreover, a technique like linear optimization cannot be used to approximate the filter response since the transfer function is rational rather than a polynomial, i.e.

$$Y(\omega) = \frac{\sum_{k=0}^{M} b_k e^{-j\omega k}}{\sum_{k=1}^{N} a_k e^{-j\omega k}}, \forall b_k \in \mathbb{R} \tag{2.30}$$

Therefore, the design method for digital IIR filter design is to choose from various analog IIR filter designs with analog frequency requirements meeting the desired specification to digital domain and by using the bilinear transform [10], which is given by:

$$s = \frac{2}{T}\frac{z-1}{z+1} \tag{2.31}$$

The characteristic of common IIR continous time filters are:

- Butterworth: Monotonic in both pass/stop band
- Chebyshev Type I: passband equiripple, monotonic stopband
- Chebyshev Type II: passband monotonic, equiripple stopband
- Elliptic: Equiripple in both passband/stopband

An equiripple allows for a narrower transition band in exchange for a little ripple in the respective band. Order of the filter can be found by determining the analog response of the filter.

Realization problem

The realization problem for an IIR filter is much more complicated compared to its FIR counterpart.

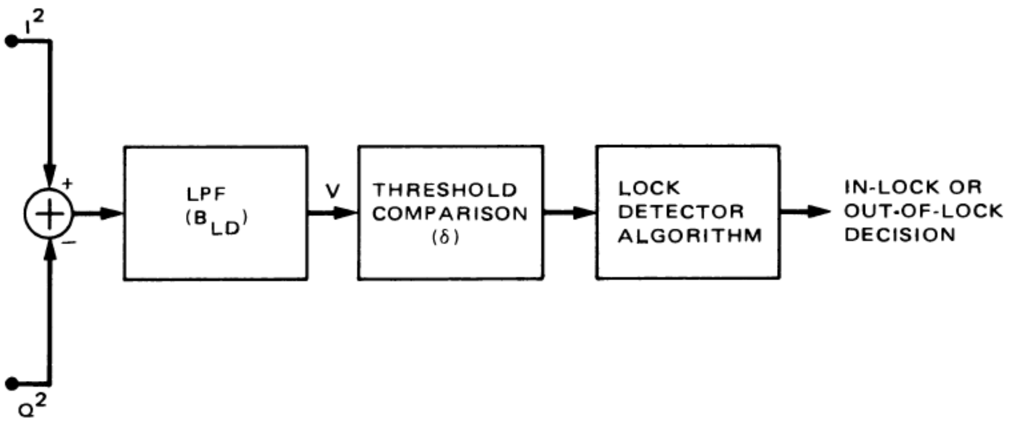


Figure 2.12: Costas Loop Lock Detector Algorithm for Biphase reception [2]

2.4 Lock Indicator

A lock indicator generates a binary signal to indicate whether the loop has switched from acquisition mode to the tracking mode and vice versa.

The lock indicator algorithm illustrated in Fig. 2.12 for supressed carrier biphase reception proposed in Reference [2] involves two steps: basic indication of whether loop is inlock and a verification of whether loop is in lock.

The difference between the square of in phase and quadrature phase arm filter outputs is modeled as

$$I^2 - Q^2 = P\hat{m}^2(t)cos2\varphi(t) + \underbrace{\upsilon_2[t, 2\varphi(t) + \frac{\pi}{2}]}_{\text{Additive noise part}} \tag{2.32}$$

where $\varphi(t) = \phi(t) - \hat{\phi}(t)$ and $\hat{m}(t)$ is the signal $m(t)$ emerging from the arm filters of the loop.

This component is passed through a low pass filter with bandwidth B_{LD} such that the $I^2 - Q^2$ signal is small compared to the data rate R_S , i.e. $B_{LD} << 1$ so that the time varying effect of $\hat{m}(t)$ due to data modulation and high frequency noise is removed from the signal.

Assuming the filter output voltage V is a Gaussian random variable, and the phase noise φ being a uniformly distributed random variable over $(0, 2\pi)$, the mean and variance of out of lock and in lock stages are:

$$\bar{V}_{O.L.} = 0 \tag{2.33}$$
$$\sigma_V^2 = N_{sq} B_{LD} \tag{2.34}$$
$$\bar{V}_{I.L.} = P K_2 \tag{2.35}$$
$$\sigma_V^2 = N_{sq} B_{LD} \tag{2.36}$$

where K_2 is the modulation distortion factor defined as

$$K_2 = \overline{< \hat{m}^2(t) >} = \int_{-\infty}^{\infty} S_m(f) |G(j2\pi f)|^2 df \tag{2.37}$$

where the $<>$ operator is time averaging, and the bar operator is the statistical expectation operator.

From this, analysis of the probability crossing a threshold δ during in lock and out of lock condition are found numerically.

The binary output of the thresholded filter output $V > \delta$ then is passed through a state machine for verification algorithm. The system's performance, including the verification algorithm is based on four parameters:

- False alarm probability, P_{FA}: Probability that loop is declared in lock when loop is actually out of lock
- Mean time to false alarm, $\bar{T}_{FA}$: Mean time for false alarm event to occur
- False dismissal probability, P_{FD}: Probability that loop is declared out of lock but in actually in lock.
- Mean time to false dismissal, $\bar{T}_{FD}$: mean time for false disimissal event to occur.

A verification algorithm given in Fig. 2.13 given in the reference [2] is as follows:

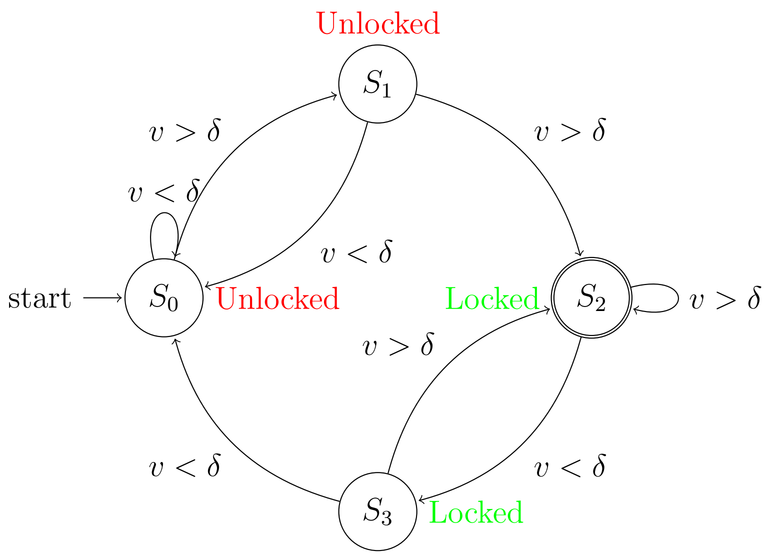


Figure 2.13: A lock verification algorithm

The specifications described before are calculated with the theory of Finite Markov Chains [11], where the idea is that the probabilty of an outcome depends at most upon the outcome of the immediately preciding event. This idea is true in the case of analyzing state machines.

The probabilities were calculated, and visualized in a graph in the reference (Pg. 117, [2]), and it was shown that P_{FA} reduces upon increasing the ratio $\frac{R_S}{B_{LD}}$ where R_S is the symbol rate and B_{LD} is the bandwidth of the filter in the lock detector. For a fixed R_S, the narrower B_{LD}, the better the performance.

Chapter 3

Methodology, System Architecture and Outcomes

3.1 Methodology

The objectives of the project, i.e. implementation of the following features have been met:

- Basic Costas Loop
- IIR filter implementation
- Lock Indicator

There were three features to be implemented sequentially (including the basic costas loop). For each feature, the work was done according to the particular modules/components of the whole feature.

Each feature had the following steps:

1. Initial Survey: Read general literature review & theory and understood the working of the feature

2. Design module/component: Read literature review, theory and implemented simulation on MATLAB. Sometimes this step was combined with the next step to see it working in the overall system.

3. Integrate Design: Combined all components/modules into one whole feature, and simulated on MATLAB.

4. Convert design to fixed point implemented: Each component in the design was converted to their fixed point counterparts using Fixed Point Designer which took into account non linear elements in the system like truncation and overflow.

5. Build module/component: Created VHDL code and appropriate testbench with MATLAB, and simulated the task on a testbench on GHDL compiler using VHDL 93 standard.

6. Integrate Build: Incorporated all components/modules to integrate the whole feature in VHDL and simulated the task with a testbench with MATLAB and GHDL compiler.

7. Pre-Synthesis Functional Simulation: Added the VHDL files to a Libero IDE project, and simulated on Modelsim using MATLAB testbench.

8. Synthesis: Here, Synplify IDE maps components to configurable logic block. Checked if module can be synthesized with cores used and slack for the particular clock frequency. If more cores are occupied than the FPGA can support, both the design and build stage need to be revised for a leaner design.

9. Post-synthesis simulation: Another simulation that accounts for delays caused by combinational logic between flipflops in design. Failure here accounts for some ambiguity within the code about logical states.

10. Placement and Layout: Here, Designer software is used to assign pins on the FPGA, and the software automatically figures out the placement of different logic blocks and their routes.

11. Post Layout Simulation: The last simulation on Modelsim where path delay is also taken into account after components have been placed into the FPGA.

12. Programming and Testing: The FPGA is programmed and testing is done using standard laboratory tools like Oscilloscopes, Function Generators and such.

In reality, these steps were often non linear in nature, and literature review was part of almost every major step. For the Costas Loop feature, there were several iterations of the modules based on problems in some phases.

More information on the Libero IDE testing platform is provided in Appendix B.

The results of the simulation, testing, etc. are indicated in later sections.

3.2 Basic Costas Loop Architectrure

As mentioned earlier, there were several iterations of the Costas Loop module designs. The modules implemented were as follows:

- NCO
- Arm Filter
- Loop Filter

3.2.1 Common Modules

Some common modules were introduced for reusability.

dffregister

`dffregister` is a register where the bit word length can be set in the parent entity using the architecture. Such a design is known as a `generic` design and is quite useful in creating reusable components.

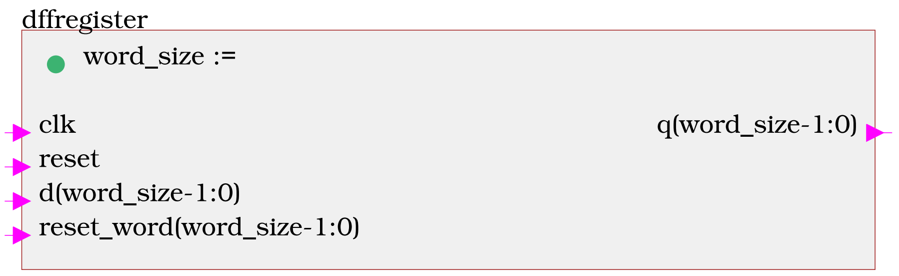


Figure 3.1: Entity `dffregister`

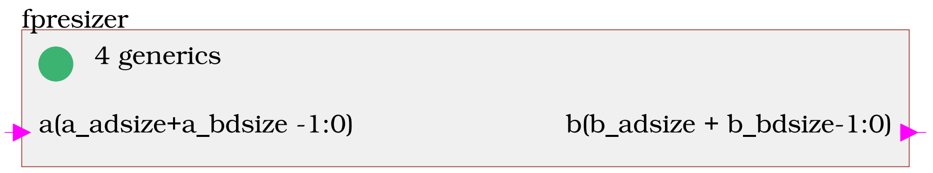


Figure 3.2: Entity `fpresizer`

fpresizer

`fpresizer` is a `generic` entity used to resize signed fixed point numbers to bigger signed fixed point number format, keeping the integer and the fraction part in line for operations like addition. Here input is $Qm.n$ where `a_adsize` is m and `a_bdsize` is n. The same pattern is followed for the output.

fptruncate

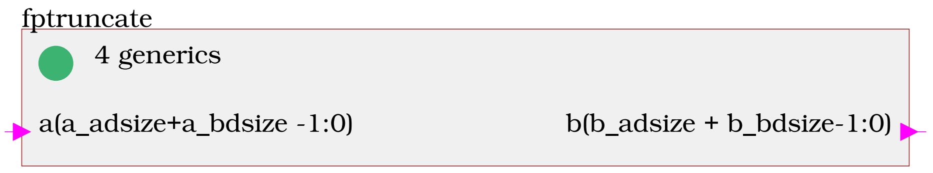


Figure 3.3: Entity `fptruncate`

`fptruncate` is a `generic` entity used to resize signed fixed point numbers to smaller signed fixed point number format. Same notation is followed as `fpresizer`.

3.2.2 NCO

Taylor Series Architecture

At first, the NCO was implemented as a phase array with combinational taylor series of a sine wave approximated as:

$$sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

However, the approximation wasn't symmetric and therefore didn't have good performance.

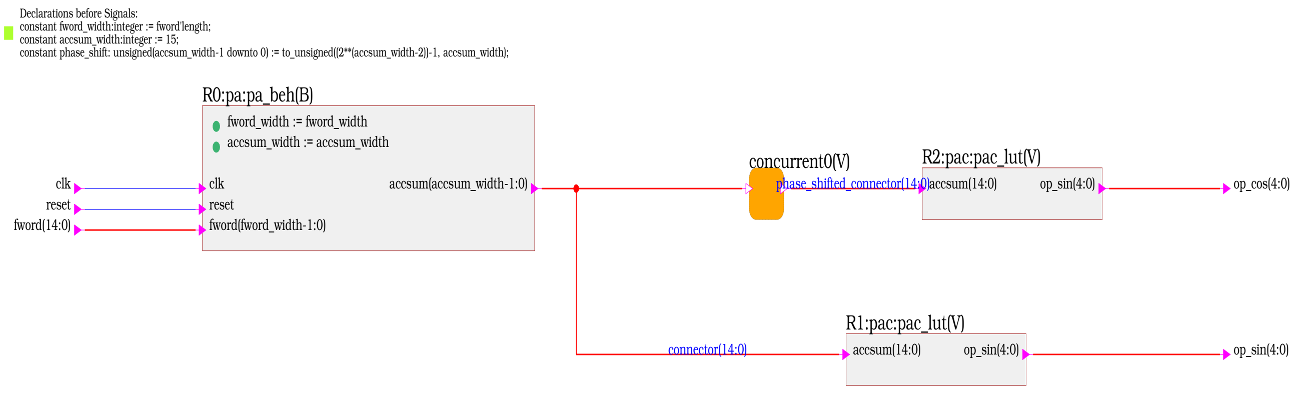
DDS Architecture

As mentioned in the literature survey, the DDS architecture consists of a phase accumalator (an integrator, essentially) and a phase to amplitude convertor (PAC). Refer to the entities below, where there is only one PA and 2 PACs, one phase shifted by 90 degrees as required in the Costas Loop Architecture. The `phase_shifted_connector` is the adder entity that shifts the phase and then feeds the phase word into another PAC.

Here, frequency word bit length, $N = 15$, accumalator bit length $M = 15$ and amplitude resolution, $P = 5$, where the output is a $Q1.4$ fixed point. The minimum frequency resolution,

$$F_{res} = \frac{F_{clock}}{2^N} = \frac{512,000}{2^{15}} = 15.625Hz$$

To generate frequencies at 8Khz and 16Khz, the frequency word ΔF is 64 and 32 respectively. In a testbench, it gives a sinusoidal output as such:



```
Declarations before Signals:
constant fword_width:integer := fword'length;
constant accsum_width:integer := 15;
constant phase_shift: unsigned(accsum_width-1 downto 0) := to_unsigned((2**(accsum_width-2))-1, accsum_width);
```

Figure 3.4: NCO architecture

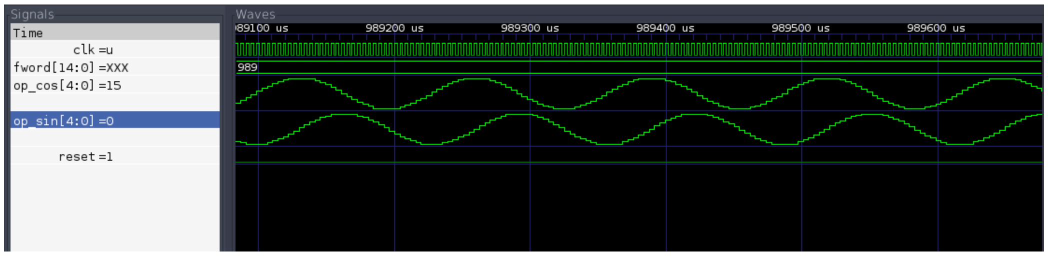


Figure 3.5: NCO Testbench Output

3.2.3 Arm Filters

As mentioned in the literature survey, the aim of the arm filters is to remove the $2\omega_0$ frequency component. Since our requirement is to track 8Khz and 16Khz supressed carriers, components of 16Khz and 32Khz will be generated after the multiplier. Therefore, the stopband frequency needs to be atleast 15Khz or so.

The arm filter had two implementations: FIR and IIR. These will be discussed now.

FIR Filter Arm Implementation

The literature survey mentions that designing any filter comprises of two problems: approximation and realization.

In the approximation problem for FIR filters, we can choose two constraints and let algorithms optimize the third. Here, we chose the filter order and the transition width to be fixed. A 10th Order Least Squares Method Optimization FIR is implemented with the following specifications (MATLAB Code):

```
order = 10;
Fpass = 6000;   % Passband Frequency
Fstop = 8000;  % Stopband Frequency
Wpass = 1;      % Passband Weight
Wstop = 0.05;      % Stopband Weight
arm_coeffs  = firls(order, [0 Fpass Fstop f_sampling/2]/
(f_sampling/2), [1 1 0 0], [Wpass Wstop]);
```

The frequency response of the filter is given in Fig. 3.6. Note that this filter was built for clock frequency of 128Khz.

To solve the realization problem, a direct form FIR filter was used, and a MATLAB custom class was created to simulate the filter in the whole system. For tolerable performance, coefficients were found to be Q1.17 fixed point. The output is also trunctated after multiplication. The input words are stored in the same word length. An RTL view of the LPF is given below.

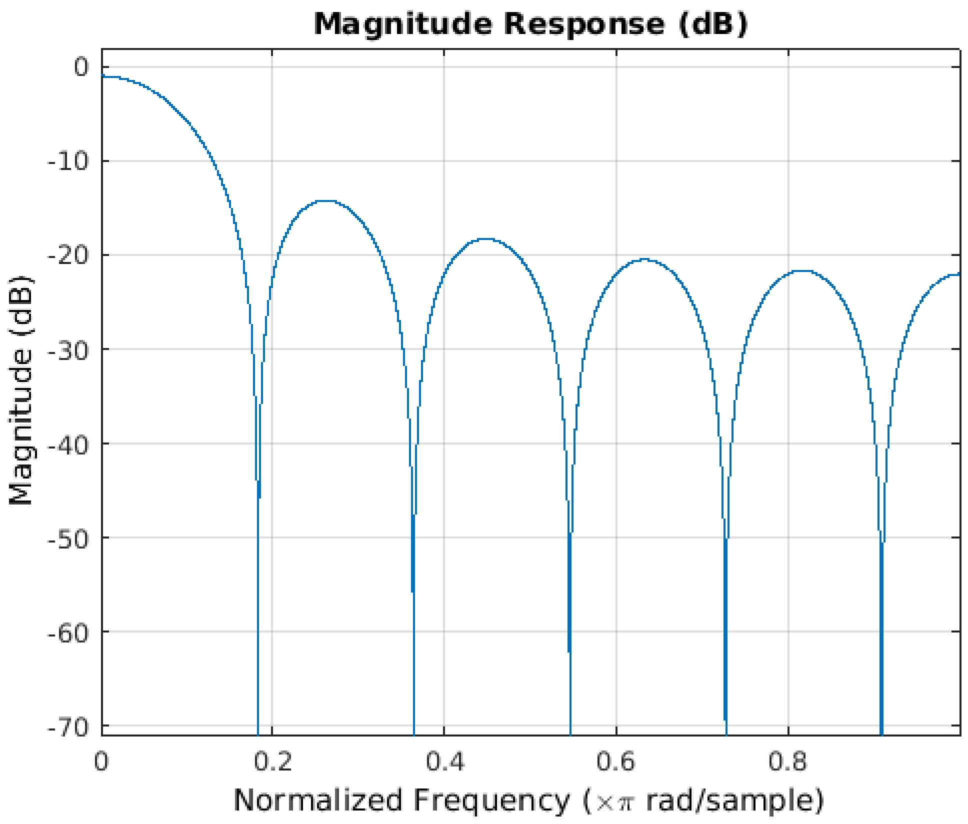


Figure 3.6: FIR Filter Frequency Response

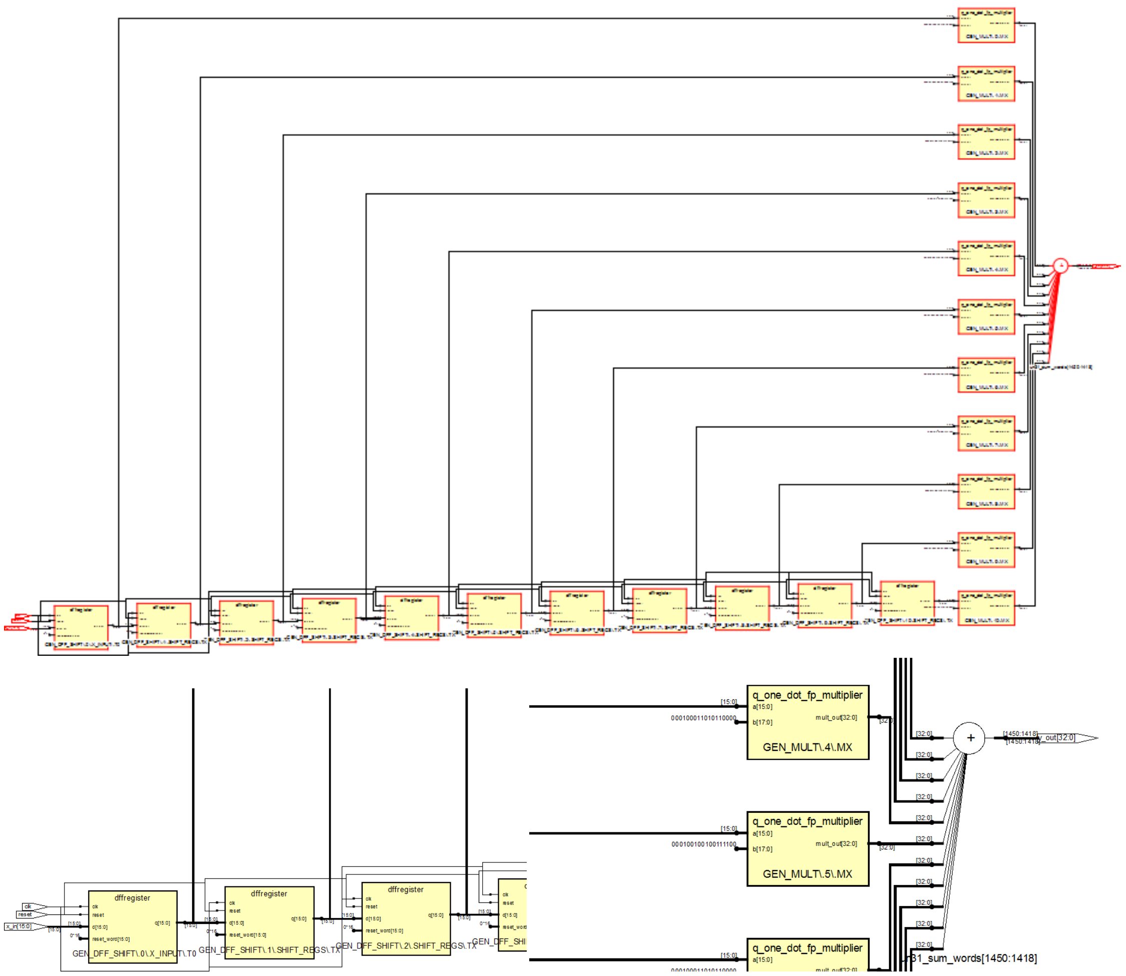


Figure 3.7: FIR filter architecture RTL view

IIR Filter Implementation

A problem faced with the FIR implementation was that it occupied 110% of the logic blocks during the synthesis stage, making actual hardware implementation on this particular FPGA impossible. Therefore, an IIR filter of the 3rd order was built using the design process described in the Literature Review, i.e. using an analog filter counterpart and then converting it. A Chebyshev II filter was used with the following specifications:

```
Fpass = 5000;   % Passband Frequency: Original 6000
Fstop = 15000;  % Stopband Frequency
PassbandRipple = 5;
StopbandAttentuation = 40; %dB

[order, cutoff] = cheb2ord(Fpass/(f_sampling/2), Fstop/(f_sampling/2),
PassbandRipple, StopbandAttentuation);

[arm_b_double, arm_a_double] = cheby2(order, StopbandAttentuation, cutoff);
```

The frequency response of this filter for a input clock frequency of 512Khz is given in Fig. 3.8.

To realize this model, we used the Direct Form I implementation (due to it's simplicity at the time of implementation). The word length of the output plus a_k coefficients were used to store past data. The structure of the filter is given in Fig. 3.9.

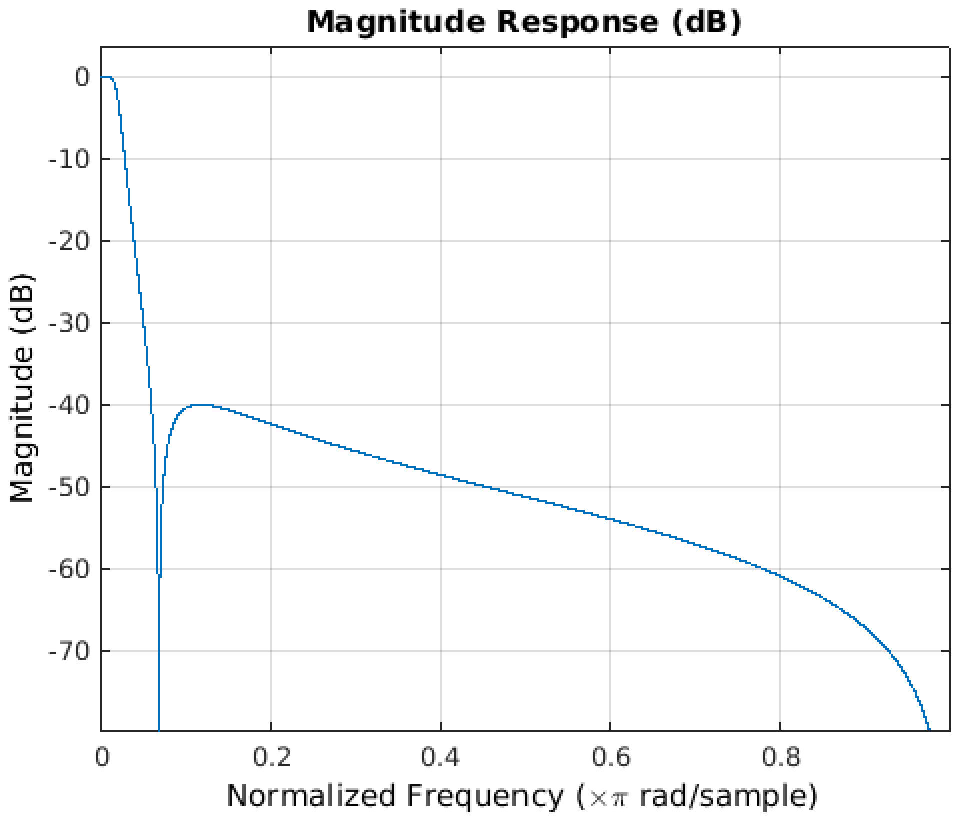


Figure 3.8: Frequency response of IIR filter

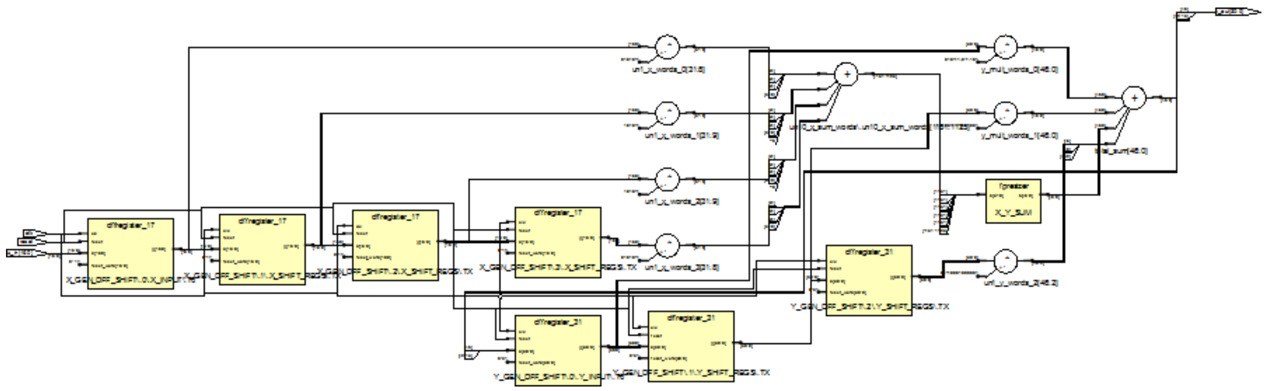


Figure 3.9: IIR filter RTL structure

3.2.4 Loop Filter

The loop filter architecture is given below. The loop filter was first designed on MATLAB using the gain constraints derived in Literature Review.

3.3 Lock Indicator Architecture

The lock indicator consists of three entities: a square truncater, a low pass filter, and a state machine. Here are their architectures:

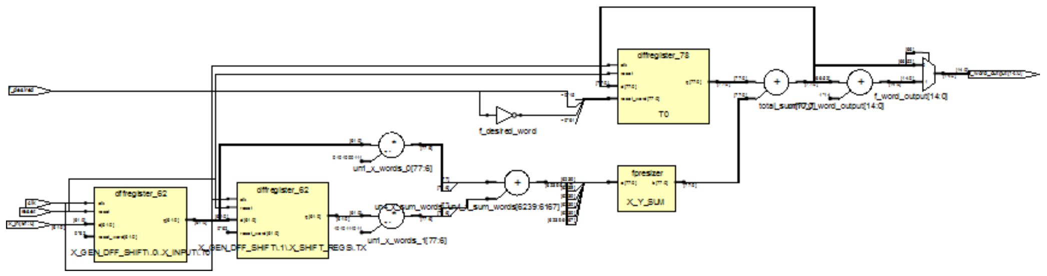


Figure 3.10: Loop filter architecture

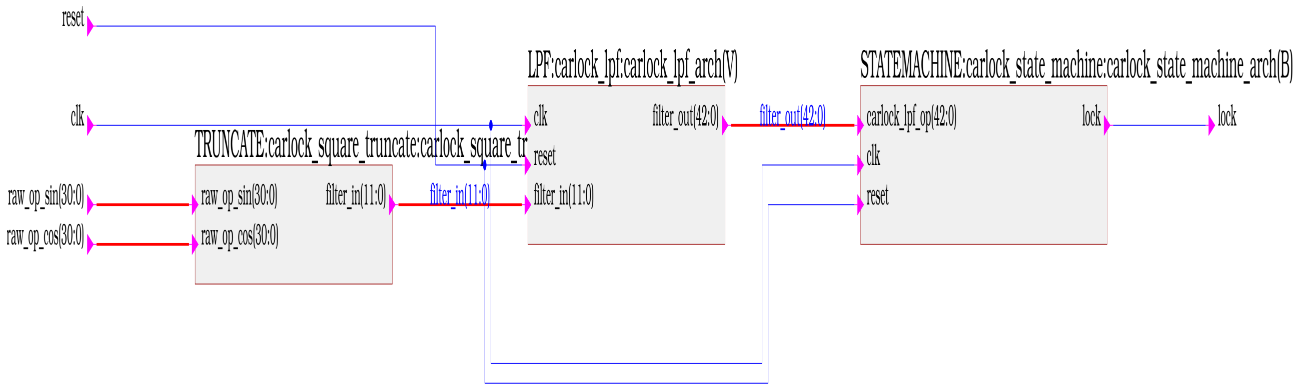


Figure 3.11: Lock Indicator Architecture

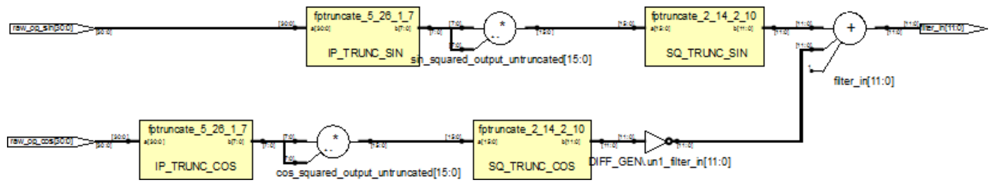


Figure 3.12: Lock Indicator Square Truncator Architecture

3.3.1 Square truncater

The square truncator initially truncates the I & the Q components, then squares them, and then truncates them further. An analysis of the energy difference between the truncations before and after was carried out on MATLAB and a minimal point for one fixed parameter was found.

3.3.2 Low pass filter

A Direct Form II low pass Chebyshev II filter with the following specifications was made:

```
locklpfpassbandfreq = 300;
locklpfstopbandfreq = 5000;
locklpfPassbandAttenuation = 1; %dB
locklpfStopbandAttenuation = 40; %dB
```

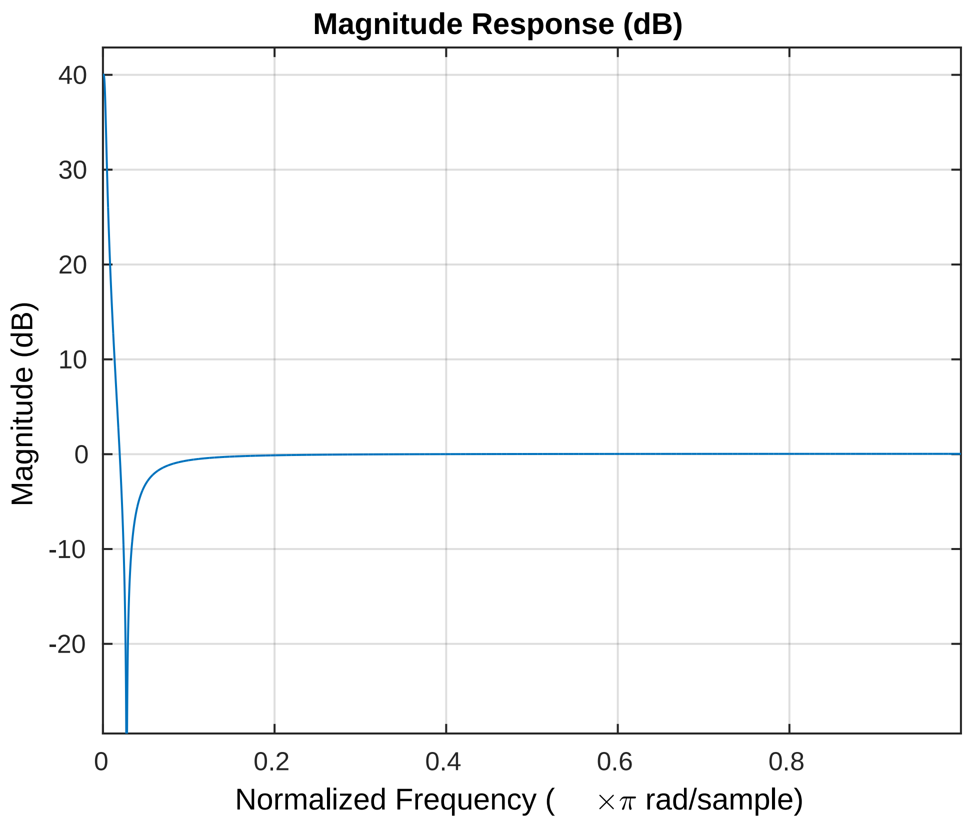


Figure 3.13: Frequency response of LPF of lock indicator

```
locklpfmatch = 'stopband'; %Band to match exactly

locklpfspecs = fdesign.lowpass(locklpfpassbandfreq,locklpfstopbandfreq,
locklpfPassbandAttenuation,locklpfStopbandAttenuation, frequency_downsample);
locklpffilterobj= design(locklpfspecs, 'cheby2', 'MatchExactly',locklpfmatch);
```

The frequency response of the LPF is in Fig. 3.13.

3.3.3 State Machine

The state machine working was illustrated in Fig. 2.13 in the Literature Review. It's implementation is illustrated below in Fig. 3.14.

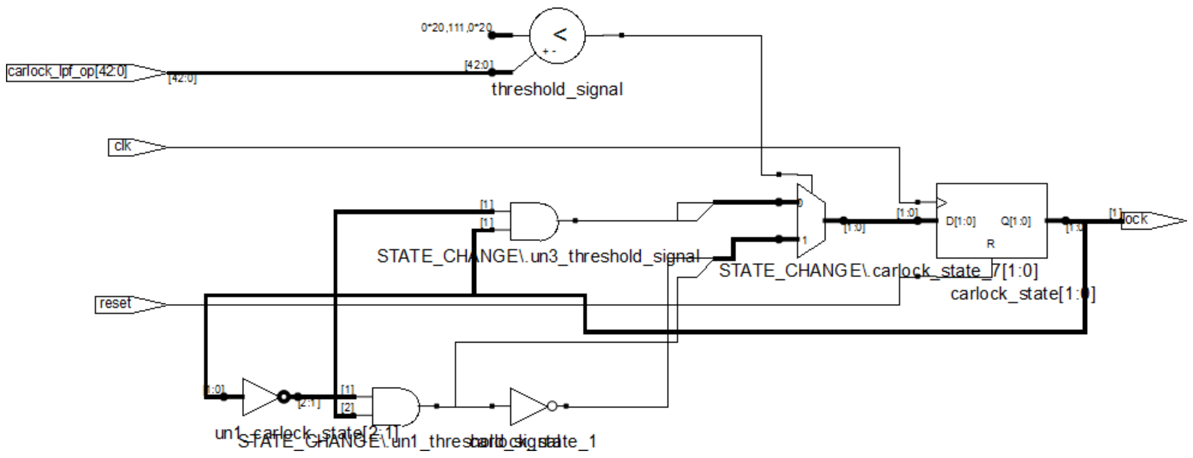


Figure 3.14: Lock Indicator State Machine Architecture

3.4 Simulation Results

As described in the sections before, the loop was first implemented as a MATLAB script and then was implemented on hardware. This involved creating custom classes for the NCO, FIR filter, IIR filter DFI and DFII structures, and a testbench helper class which could interface to both ghdl and Libero IDE's Modelsim Simulator. Using text input/output, the testbenches could input randomized simulation data from MATLAB and could output that data to be read by MATLAB again. The results for these simulations are shown below.

3.4.1 MATLAB Simulation Output

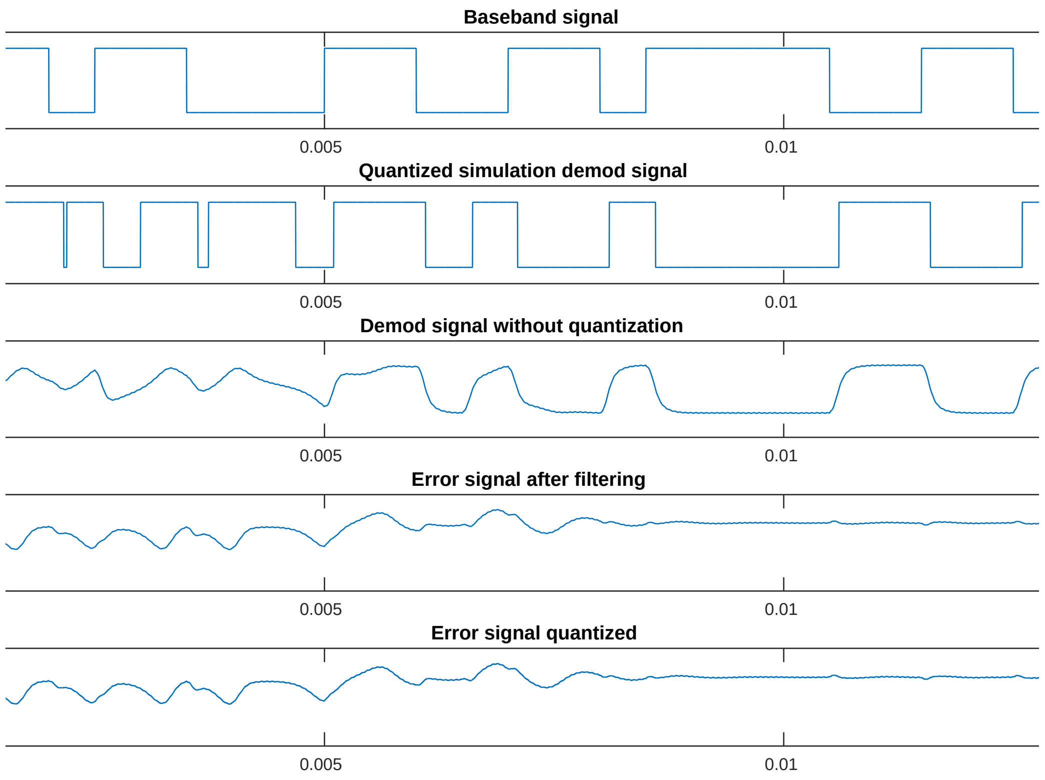


Figure 3.15: MATLAB Model Simulation Result

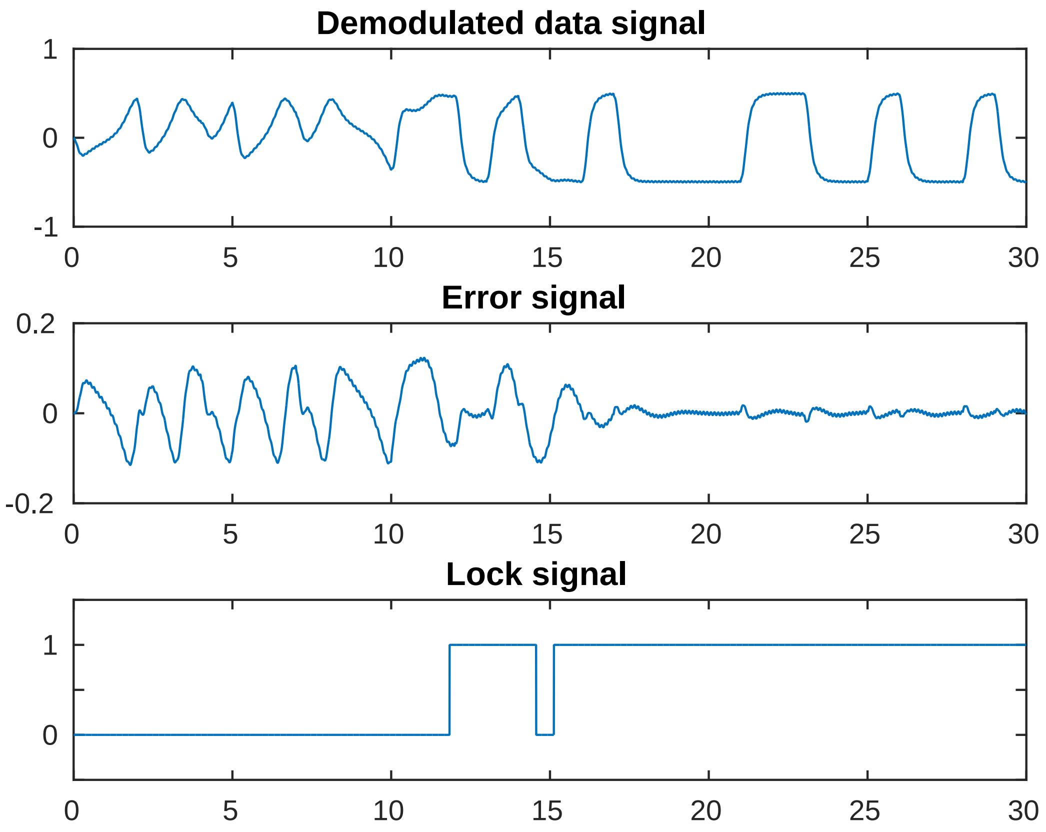


Figure 3.16: MATLAB Lock Model Simulation Result

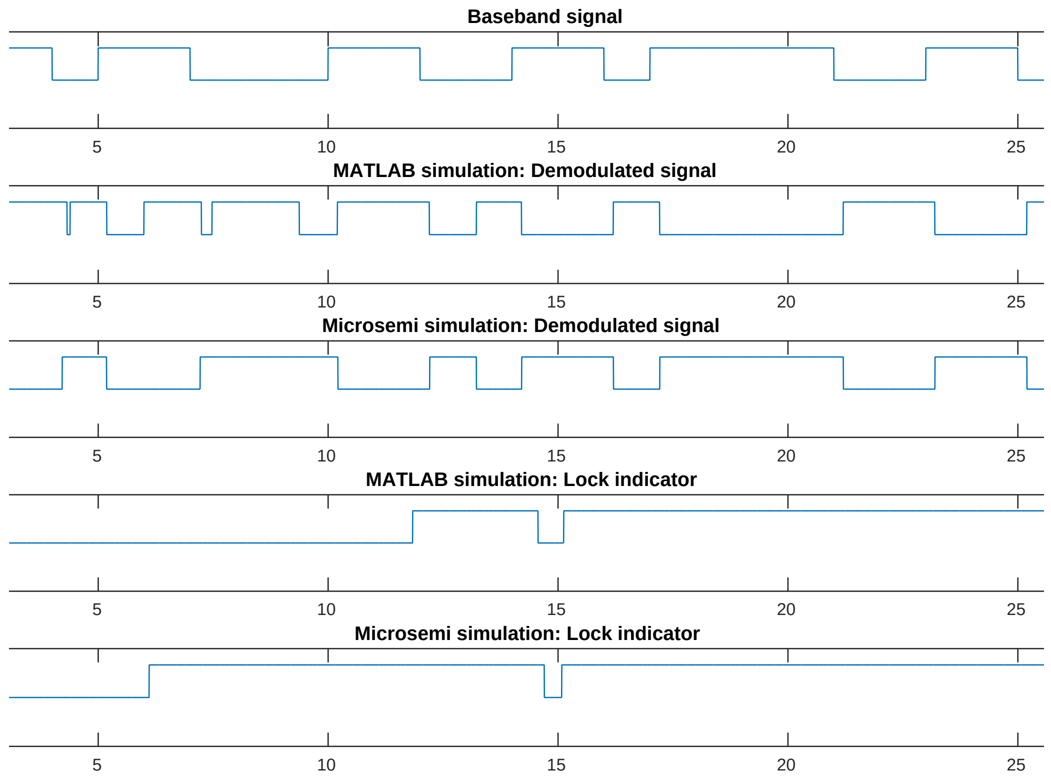
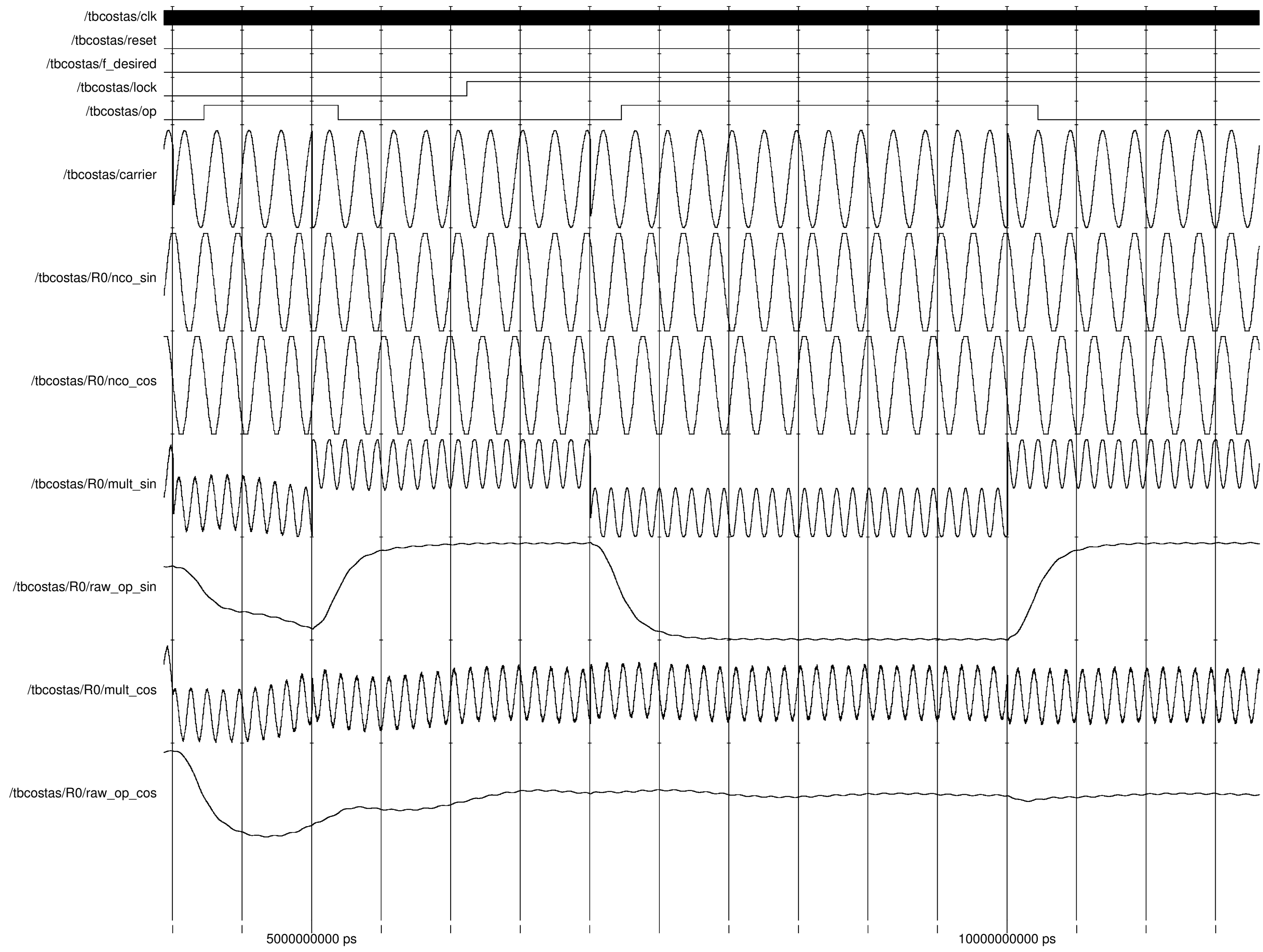
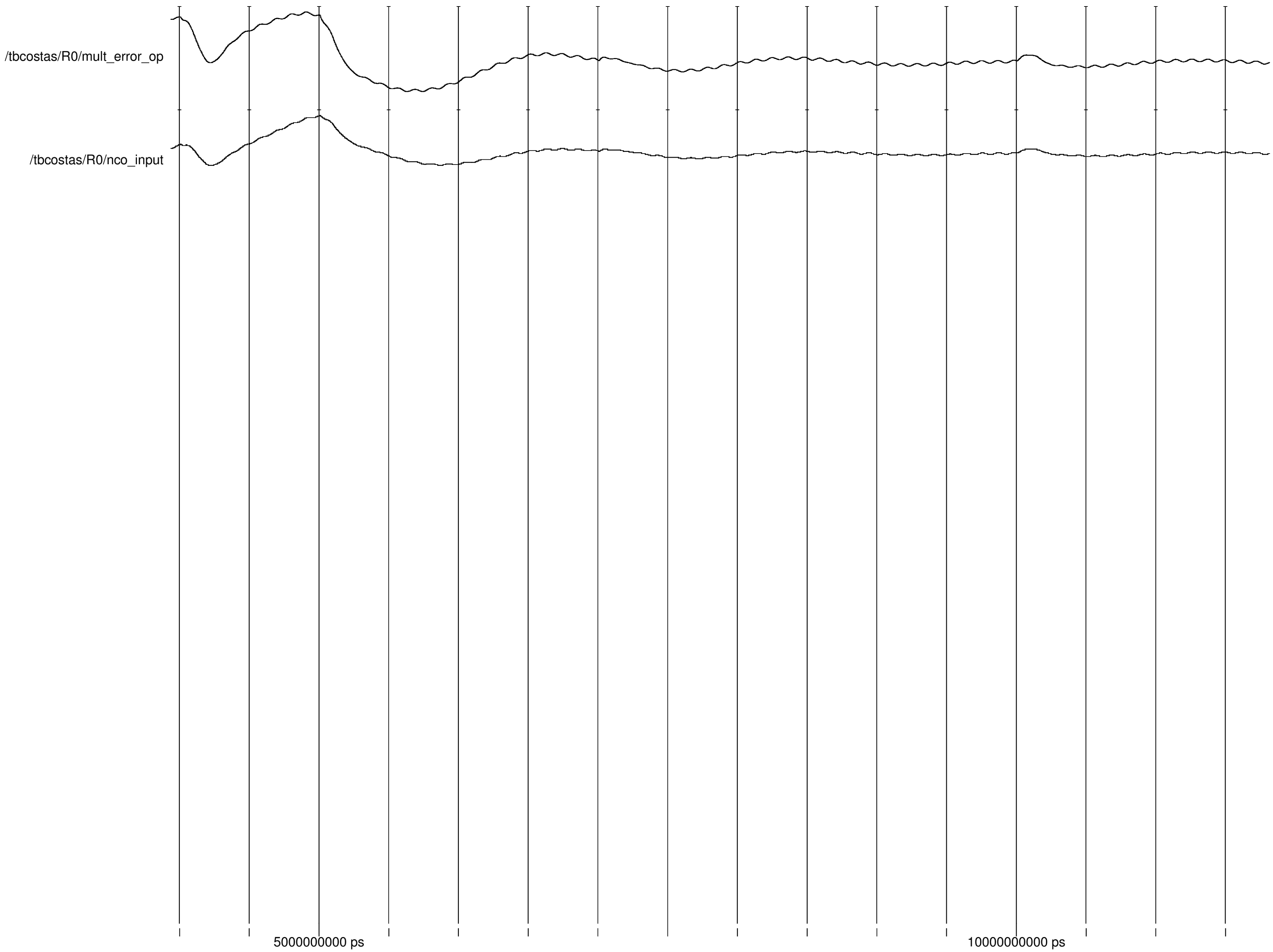


Figure 3.17: Comparison of Original, MATLAB, and VHDL Implementation Simulation Results

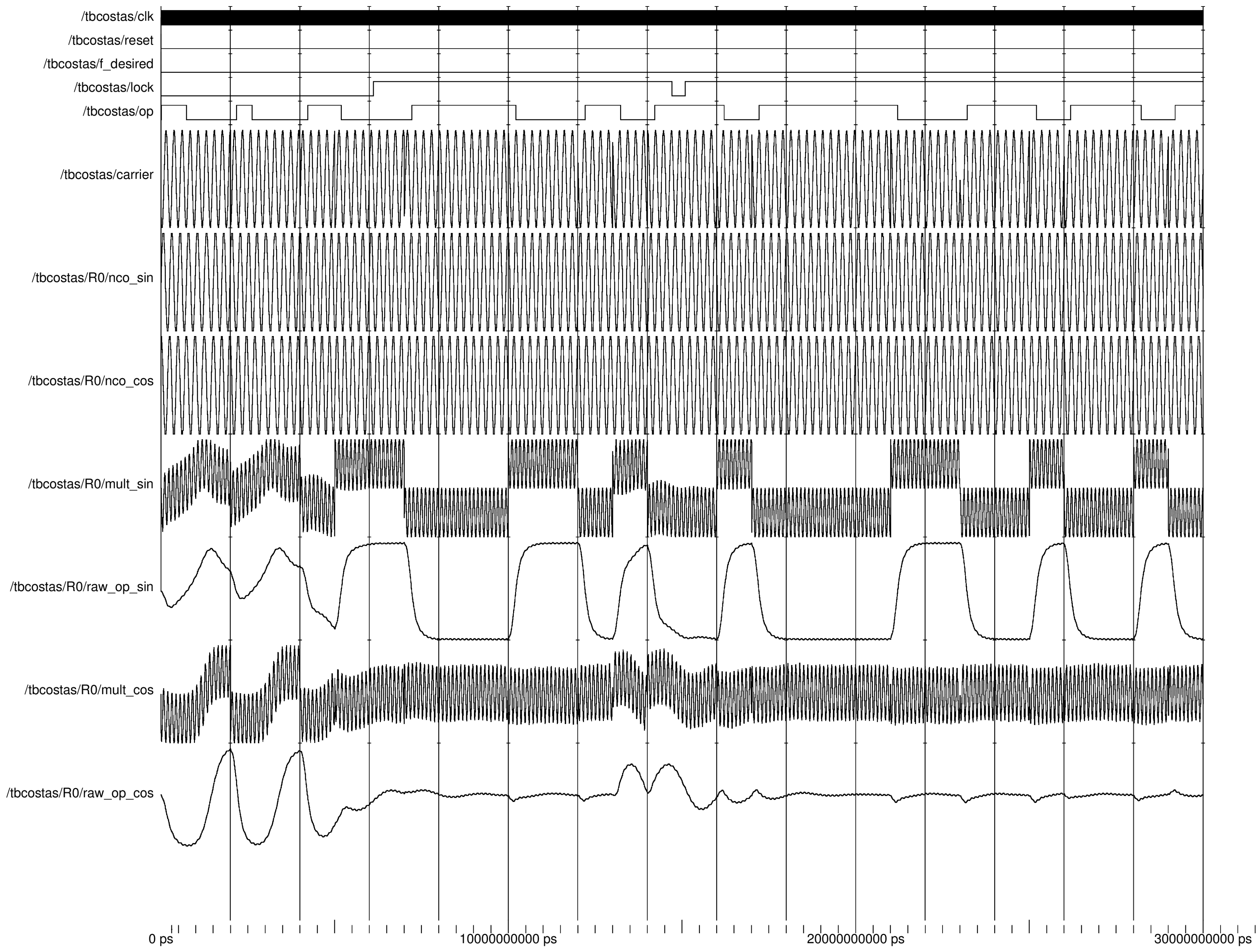
3.4.2 Modelsim Simulation Output



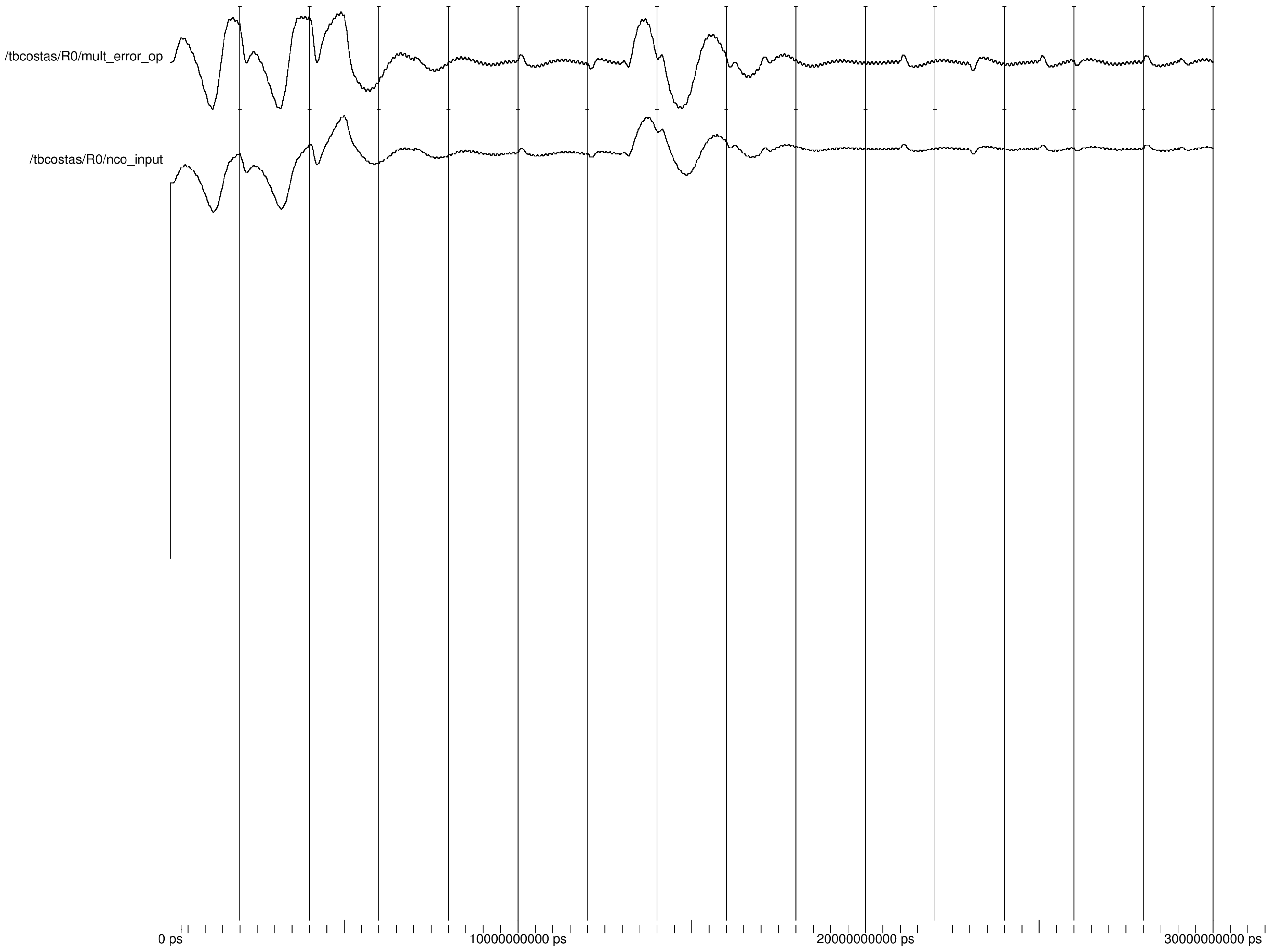
Entity:tbcostas Architecture:tbcostas_arch Date: Mon Jul 11 22:55:59 +0530 2016   Row: 1 Page: 1



Entity:tbcostas Architecture:tbcostas_arch Date: Mon Jul 11 22:55:59 +0530 2016   Row: 1 Page: 2



Entity:tbcostas Architecture:tbcostas_arch Date: Mon Jul 11 22:57:24 +0530 2016   Row: 1 Page: 1


/tbcostas/R0/mult_error_op
/tbcostas/R0/nco_input
0 ps
10000000000 ps
20000000000 ps
30000000000 ps

3.5 Hardware Implementation Results

To obtain these results, a DAC was attached to 12 output pins and two bits were toggled by a multiplexer to change what signal was observable through the DAC. The signals were generated by a function generator which had the capability of generating Phase Modulated signals and frequency modulated signals. The input signal of these modulations was a clock periodically varying between '0' and '1'. This was tested for two carrier frequencies: 8Khz and 16Khz.

3.5.1 8Khz Input Carrier

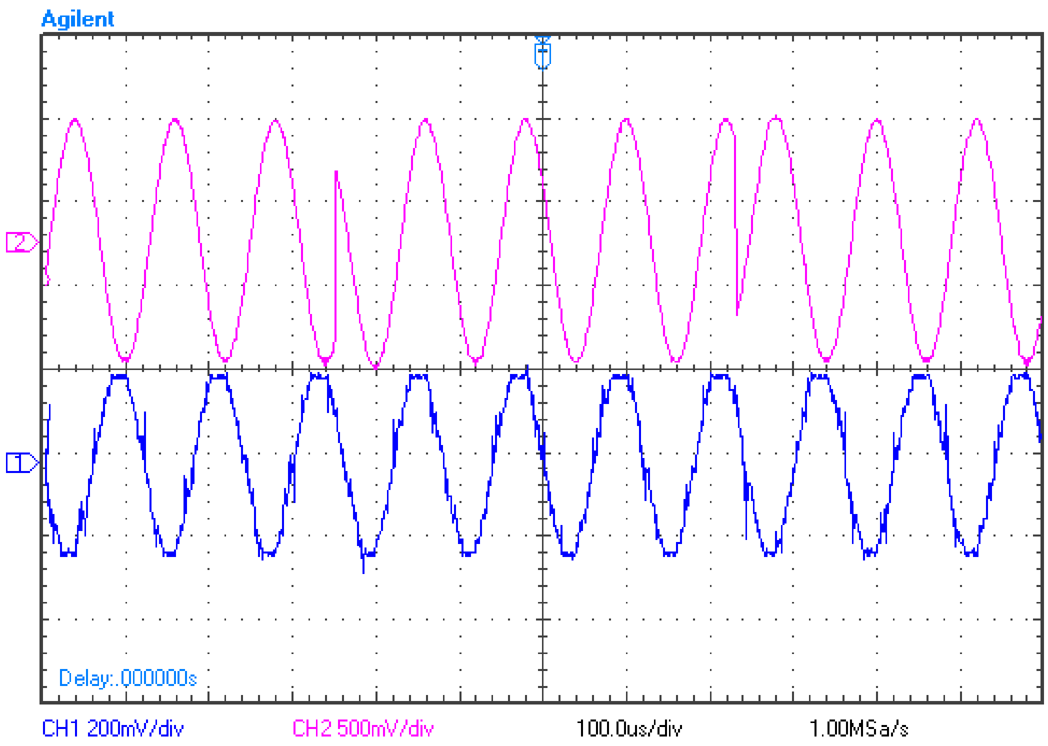


Figure 3.20: Input In-Phase NCO waveform under lock

1: NCO In-Phase component, 2: Input

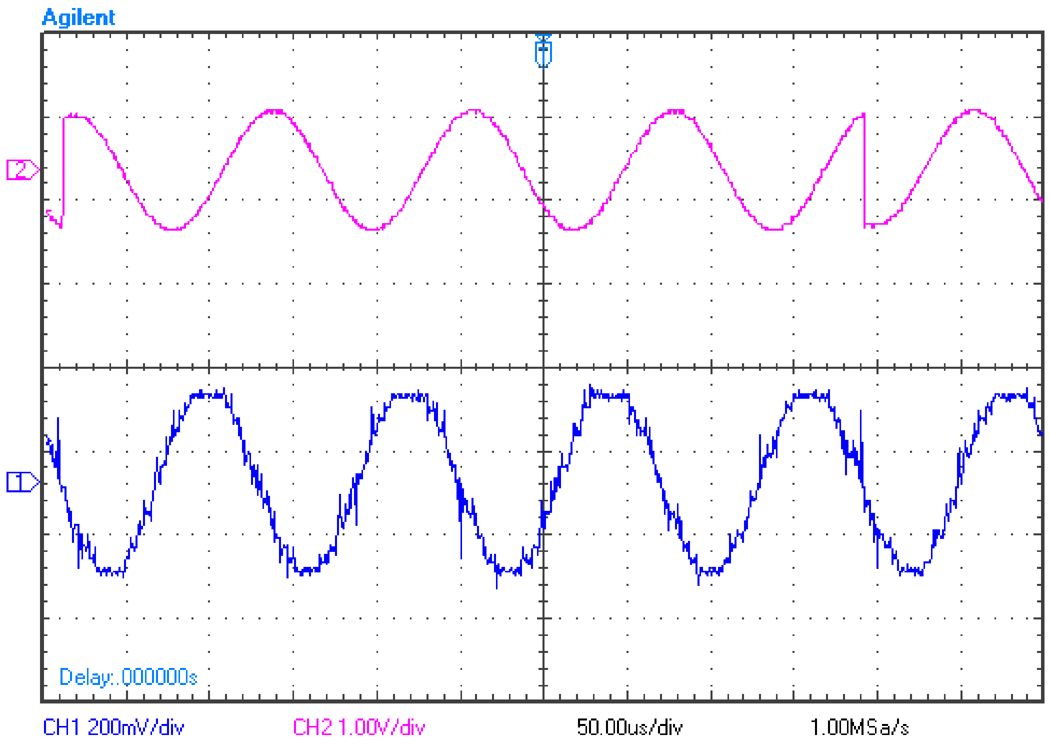


Figure 3.21: Input Out-of-Phase NCO waveform under lock

1: NCO Out-of-Phase component, 2: Input

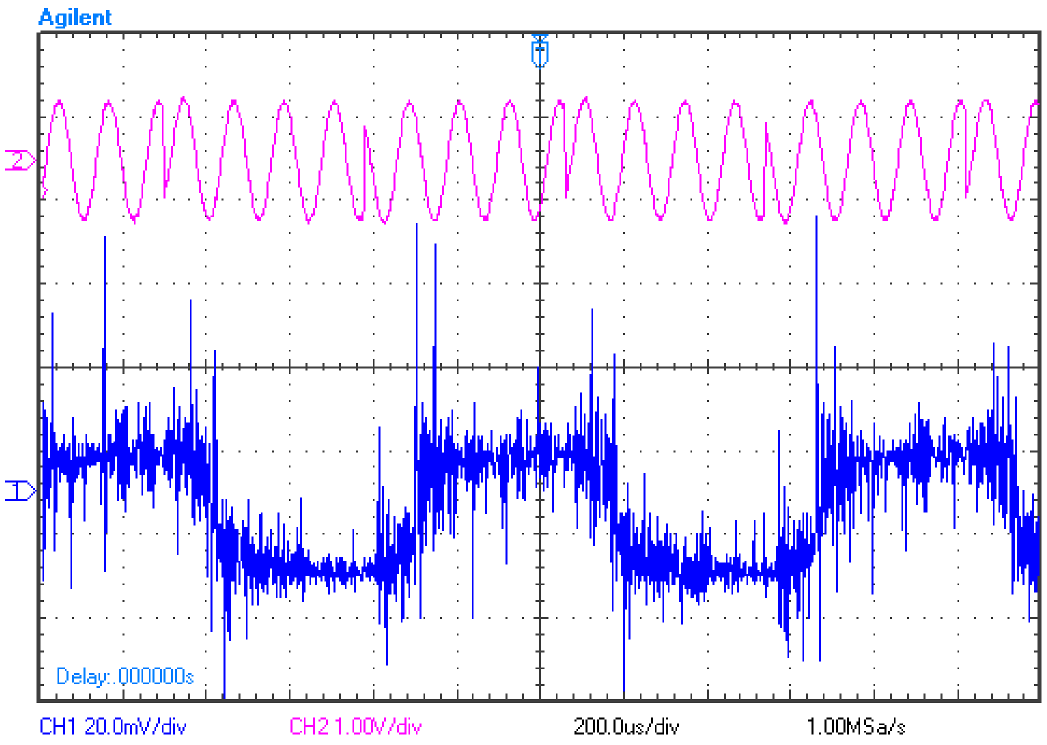


Figure 3.22: Input I-Arm output waveform under lock

1:I-Arm output, 2: Input

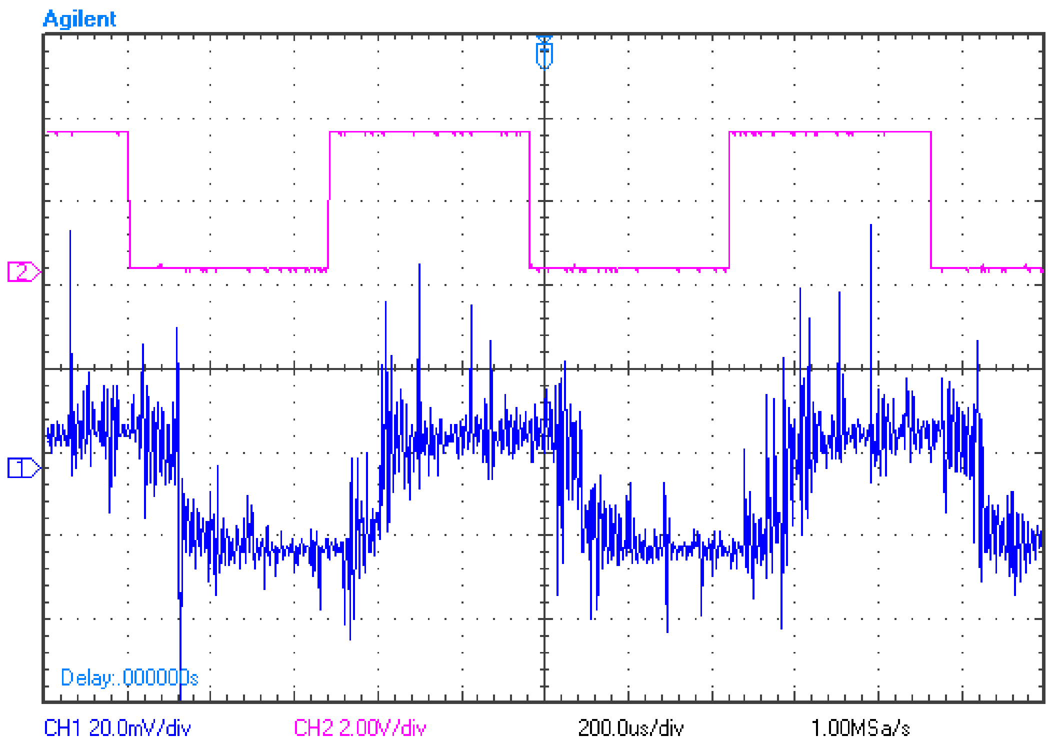


Figure 3.23: Input I-Arm output waveform under lock

1:Input Data, 2: I-Arm Output

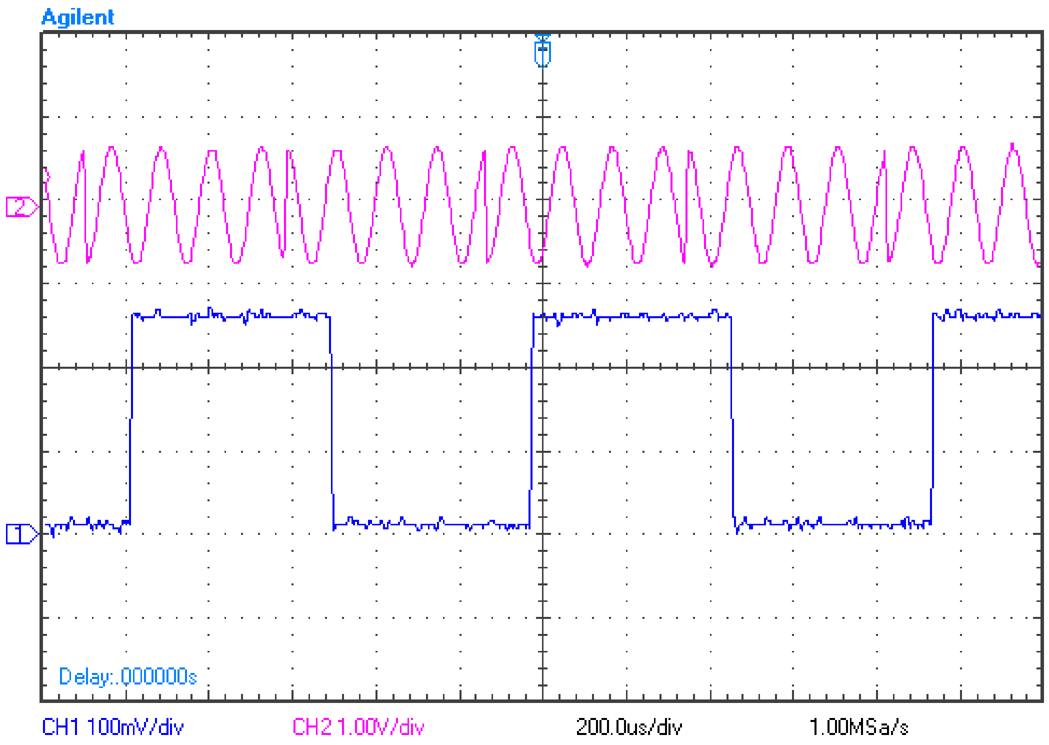


Figure 3.24: Input and quantized output waveform under lock

1:Quantized output, 2: Input waveform

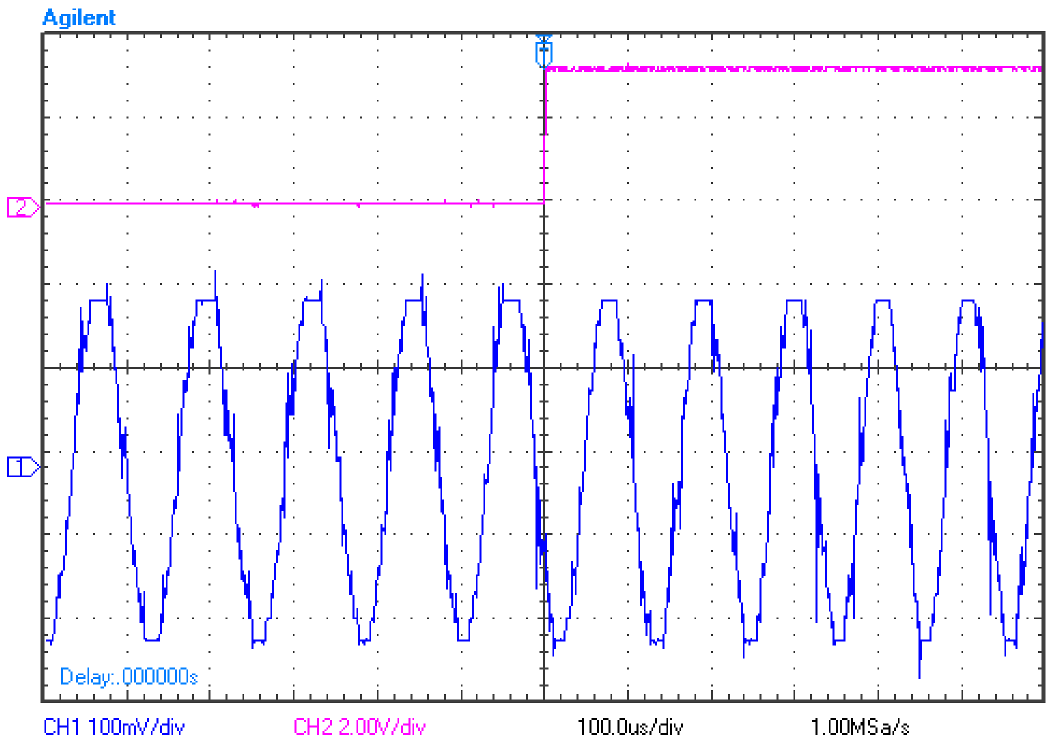


Figure 3.25: FSK shifted NCO output waveform under lock

1:NCO output, 2: FSK shift time instant

3.5.2 16Khz Input Carrier

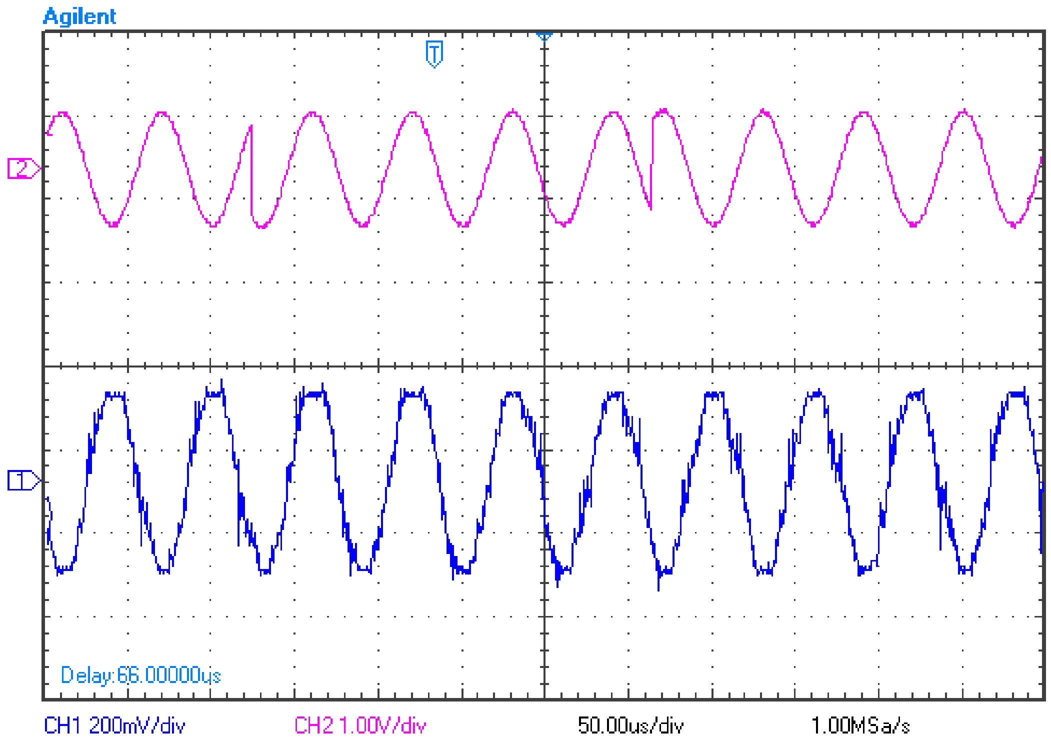


Figure 3.26: Input In-Phase NCO waveform under lock

1: NCO In-Phase component, 2: Input

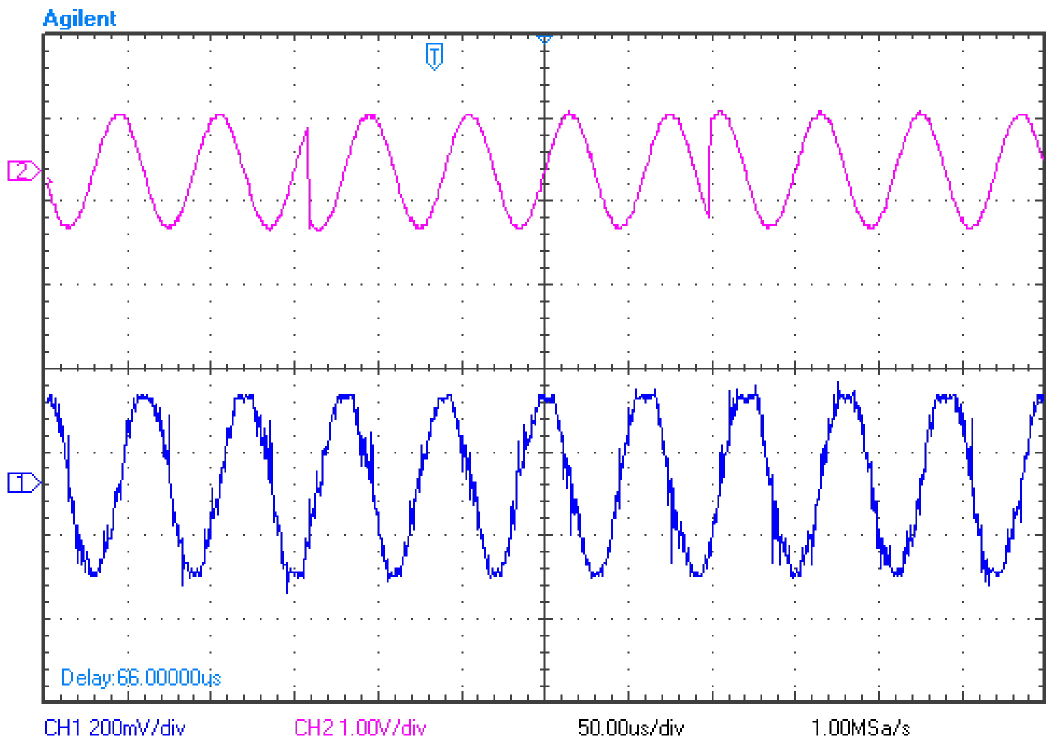


Figure 3.27: Input Out-of-Phase NCO waveform under lock

1: NCO Out-of-Phase component, 2: Input

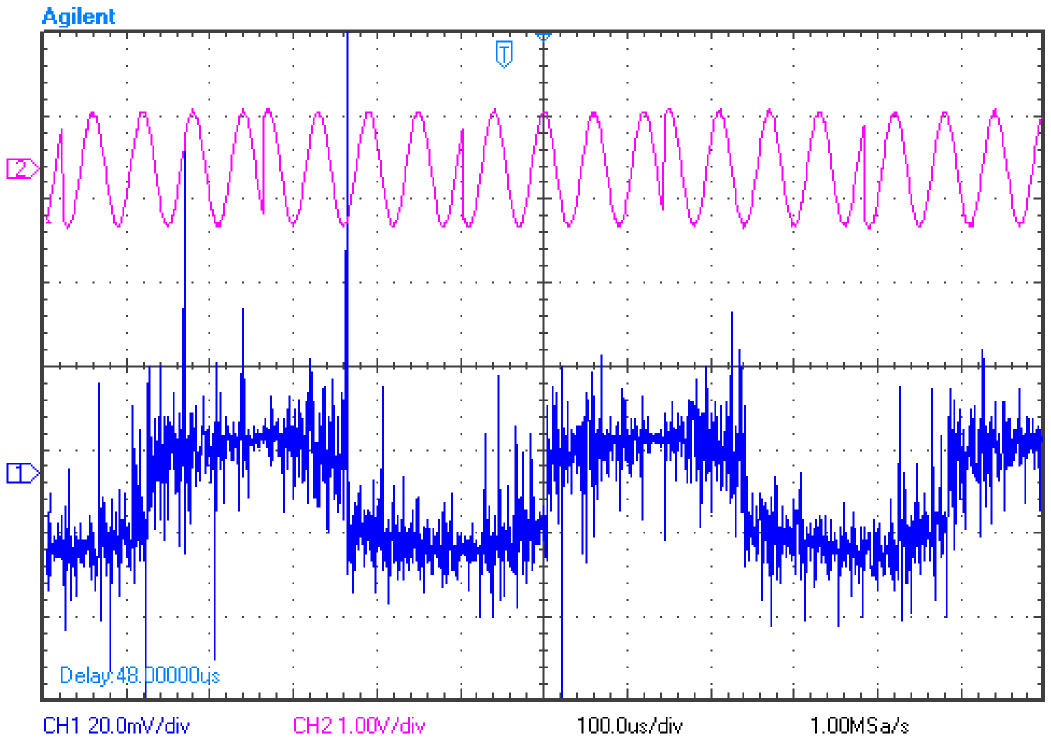


Figure 3.28: Input waveform and I-Arm filter output under lock

1: I-Arm filter output, 2: Input

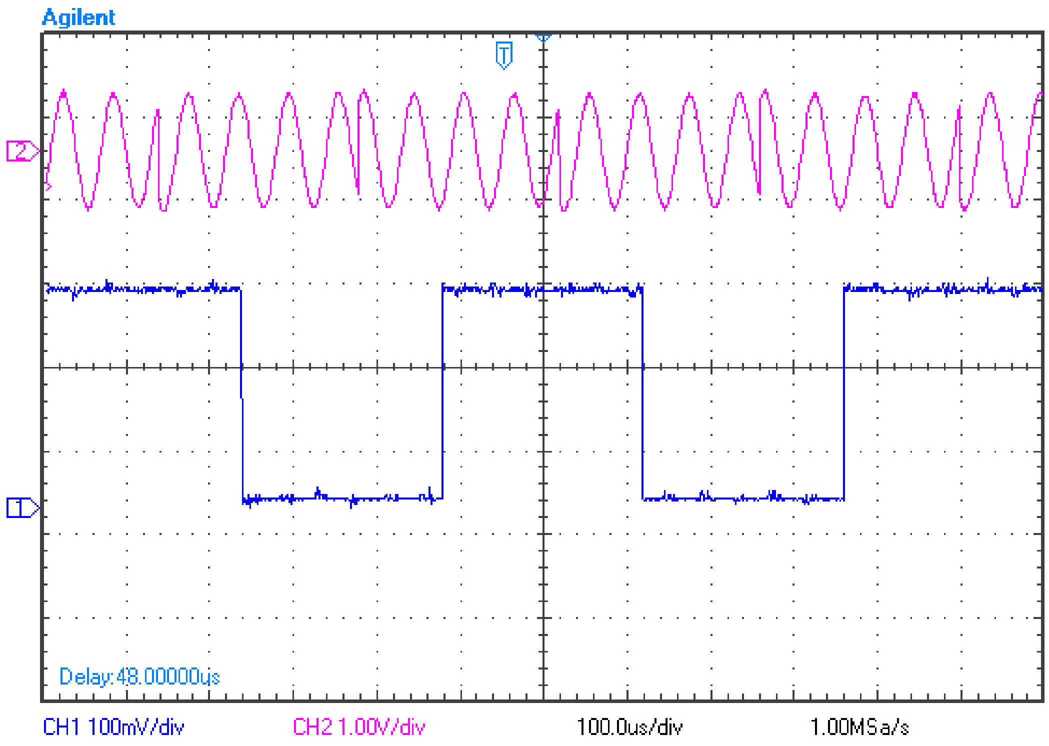


Figure 3.29: Input waveform and quantized output waveform

1: Quantized output, 2: Input

Chapter 4

Discussion, Conclusions, and Recommendations

A BPSK demodulator with lock indicator was sucessfully built on an FPGA. This project involved the various domains of control theory, communication systems, and digital electronics. Techniques from various literature were surveyed and described, and a working model was built from the knowledge gained.

The future scope of this work can include the following :

- Literature survey of using non-linear control techniques and probabilistic models to examine the operation of the loop
- Using the CORDIC algorithm to extract phase from the carrier
- Implementing a clock extraction PLL
- Automation of creating the costas loop directly from software given few specifications
- Analyzing the effects of different ADCs

Part II

Non-Technical

Chapter 5

Soft-Skills

I believe the internship gave me an opportunity to also develop my soft skills in various domains, particularly personal, interpersonal and professional skills.

- Communication Skills: I felt that I was weak in this domain, after this internship, I have gained immense experience. It has made me a more confident individual.
- Professional skills: Skills like time management and organization have been vastly improved due to the structure the internship gave me.
- Personal skills: A large chunk of the project required study from literature which gave me the discipline and rigour required in the industry for acquiring new concepts.

This internship was an immensely prolific experience and I thank everyone involved in it to improve my soft skills.

Chapter 6

Technical Skills Review

6.1 Most useful subjects taught at NIIT University

I believe that the project required knowledge of plethora of subjects pertaining to my degree of specialization, Electronics and Communication Engineering. They were:

- Signals and Systems
- Digital Electronics
- Digital Signal Processing
- Analog Communications
- Digital Communications
- Wireless Communications
- Control Theory

Additionally, I felt that the experience from the R&D project made me familiar in reading research papers, which I feel was immense for the discipline required to understand certain topics.

6.2 Skills to focus on at NU

- FPGA hands-on experience: Although Digital Electronics did involve simulation using VHDL to some end, I've found that people from other universities have already had some hands on experience with FPGAs, particularly with Xilinx and Altera IDEs which use Verilog. An experience like this gives one an understanding of both digital design and some understanding of what goes into chip design.

- Simulink, MuPad, newer MATLAB toolboxes: Although MATLAB is a great tool for learning scripting and the basics of simulation, newer tools built inside MATLAB such as Simulink, Filter Design Toolbox, Control Systems Toolbox and other advanced features provide superior understanding and immediate practical applications in industry. Simulink, by itself is a tool which is much easier to use and is practically used everywhere where MATLAB can be. These tools were taught to people from some other universities.

- Digital Control Theory: Since Control Theory was taught to us in such an intuitive way, I was able to catchup and understand a bit about digital controllers. But today's scenario, digital controllers are everywhere and therefore some introduction should be included either in Advanced DSP, Microcontrollers, or Control Theory classes or as an extra elective for the student's benefit.

References

[1] R. T. R, N. R, S. Joy, A. T. J, and A. V., "Design and implementation of digital costas loop and bit synchronizer in fpga for bpsk demodulation," in *Control Communication and Computing (ICCC), 2013 International Conference on*, pp. 39–44, Dec 2013.

[2] J. H. Y. e. Joseph H. Yuen (auth.), *Deep Space Telecommunications Systems Engineering*. Applications of Communications Theory, Springer US, 1 ed., 1983.

[3] J. P. Costas, "Synchronous communications," *Proceedings of the IRE*, vol. 44, no. 12, pp. 1713–1718, 1956.

[4] F. Gardner, *Phaselock Techniques*. Wiley, 2005.

[5] K. Ogata, *Modern Control Engineering*. graph. Darst, Prentice Hall, 2002.

[6] I. Kar and S. Majhi, "Digital Control System, NPTEL Electrical Engineering."

[7] T. W. Parks and C. Burrus, *Digital filter design*. Topics in digital signal processing, Wiley, 1987.

[8] B. V. Veen, *Linear Phase FIR Filters*. 2012.

[9] R. A. Losada, "Practical fir filter design in matlabr," 2004.

[10] B. V. Veen, *IIR Filter Design Procedure*. 2012.

[11] J. G. Kemeny, *Introduction to finite mathematics*. Prentice-Hall, 3d ed ed., 1974.

Appendices

Appendix A

2nd Order Digital PLL Stability Analysis on MuPad

```
assume(z in C_), assume(k in R_), assume(k_2 in R_)
```

```
integrator := expand((z^-1)/(1 - (z^-1)))
```

$$\frac{1}{z-1}$$

```
controller := (1 + k_2*integrator)
```

$$\frac{k_2}{z-1}+1$$

```
nco := integrator
```

$$\frac{1}{z-1}$$

```
delay := z^-1
```

$$\frac{1}{z}$$

```
gh := simplify(delay*nco*controller)
```

$$\frac{k_2 + z - 1}{z\,(z - 1)^2}$$

```
p_func := k*numer(gh) + denom(gh)
```

$$k\,(k_2 + z - 1) + z\,(z - 1)^2$$

```
p := poly(p_func ,[z])
```

$$\mathrm{poly}\left(z^3 - 2\,z^2 + (k + 1)\,z + k\,(k_2 - 1),\,[z]\right)$$

```
simplify(solve(p = 0, z, MaxDegree = 3))
```

$$\begin{cases} \left\{\frac{\sigma_4}{3} + \frac{2}{3}, \frac{2}{3} - \frac{\sigma_4}{6} - \sigma_2, \frac{2}{3} - \frac{\sigma_4}{6} + \sigma_2\right\} & \text{if } k = \frac{1}{3} \\ \left\{\sigma_3 - \frac{\sigma_5}{\sigma_3} + \frac{2}{3}, \frac{\sigma_5}{2\,\sigma_3} - \frac{\sigma_3}{2} + \frac{2}{3} - \sigma_1, \frac{\sigma_5}{2\,\sigma_3} - \frac{\sigma_3}{2} + \frac{2}{3} + \sigma_1\right\} & \text{if } k \neq \frac{1}{3} \end{cases}$$

where

$$\sigma_1 = \frac{\sqrt{3}\left(\frac{\sigma_5}{\sigma_3} + \sigma_3\right) i}{2}$$

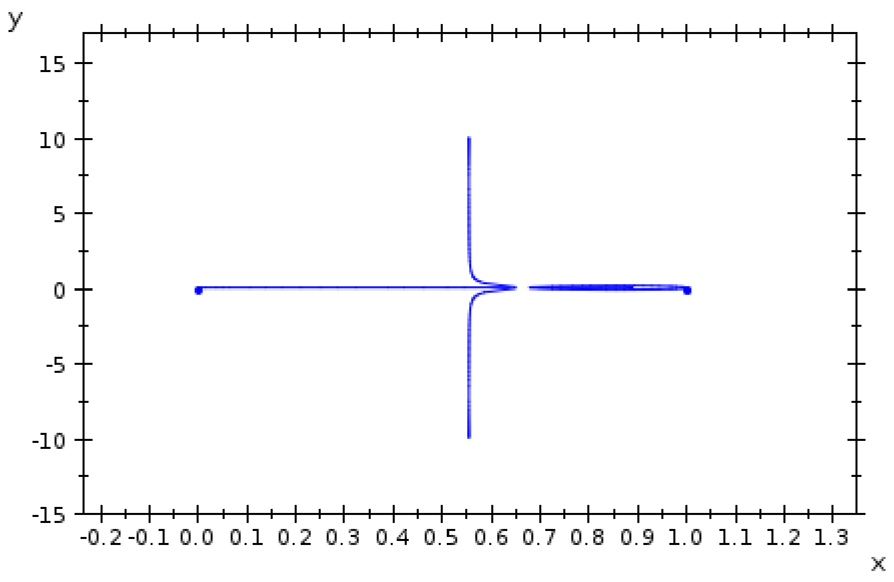
$$\sigma_2 = \frac{\sqrt{3}\,\sigma_4\, i}{6}$$

$$\sigma_3 = \left(\frac{k}{6} + \sqrt{{\sigma_5}^3 + \left(\frac{k\,k_2}{2} - \frac{k}{6} + \frac{1}{27}\right)^2} - \frac{k\,k_2}{2} - \frac{1}{27}\right)^{1/3}$$

$$\sigma_4 = (1 - 9\,k_2)^{1/3}$$

$$\sigma_5 = \frac{k}{3} - \frac{1}{9}$$

```
plot(plot::Rootlocus(subs(p, k_2 = 1/9), k = 0..100))
```



```
result_1 :=~ abs(coeff(p,z,0)) < coeff(p,z,3)
```

$$|k\ (k_2 - 1)| < 1$$

```
result_2 := evalp(p, z = 1) > 0
```

$$0 < k + k\ (k_2 - 1)$$

```
result_3 := evalp(p,z = -1) < 0
```

$$k\ (k_2 - 1) - k - 4 < 0$$

```
B_2 := matrix([[coeff(p,z,0),
coeff(p,z,3)],[coeff(p,z,3),coeff(p,z,0)]])
```

$$\begin{pmatrix} k\,(k_2-1) & 1 \\ 1 & k\,(k_2-1) \end{pmatrix}$$

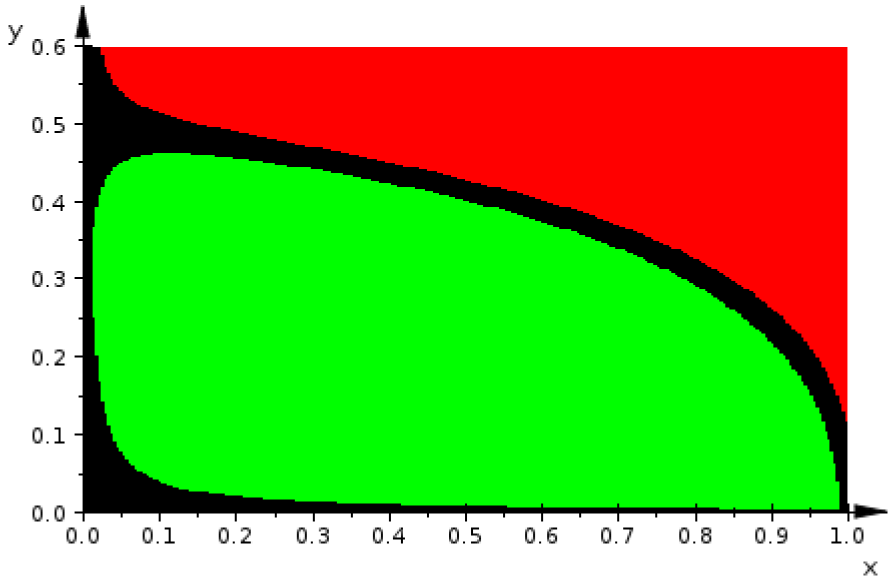
```
B_0 := matrix([[coeff(p,z,0), coeff(p, z, 1)], [coeff(p,z,3),
coeff(p,z,2)]])
```

$$\begin{pmatrix} k\,(k_2-1) & k+1 \\ 1 & -2 \end{pmatrix}$$

```
result_4 := abs(det(B_2)) > abs(det(B_0))
```

$$|2\,k\,k_2-k+1| < |k^2\,k_2{}^2-2\,k^2\,k_2+k^2-1|$$

```
plot(plot::Inequality([result_1, result_2, result_3, result_4],
k = 0..1, k_2 = 0..0.6))
```



Appendix B

FPGA Development Methodology

An IDE provided by Microsemi called Libero was used. The design flow of Libero is explained here.

Step One - Design Creation

Plan out the design and enter it as HDL (VHDL or Verilog), structural schematic, or mixed-mode (schematic and RTL).

Step Two - Design Verification - Functional Simulation

After the design is defined, it must be verified to test that it functions the way they intended. After creating a testbench using WaveFormer Lite, the ModelSim VHDL or Verilog simulator can be used to perform functional simulation on the schematic or HDL design.

Step Three - Synthesis/EDIF Generation

A design must be synthesized if the design was created using VHDL or Verilog. Using Synplify AE or Synplify Pro from Synplicity, the EDIF netlist is generated. Re-verification of the design "post-synthesis" using the VHDL or Verilog ModelSim simulator used in step two is performed.

While all RTL code must be synthesized, pure schematic designs are automatically "netlisted" out via the Libero tools to create a structural VHDL

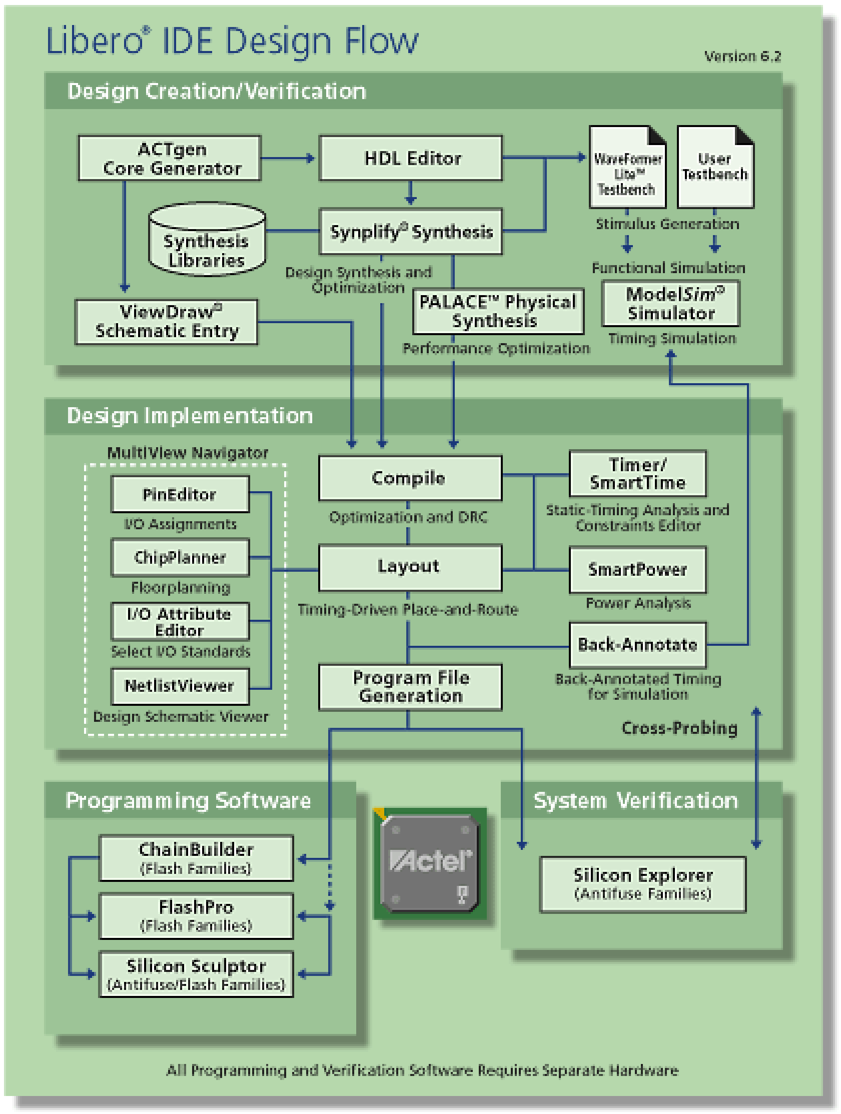


Figure B.1: Libero Design Flow

or structural Verilog netlist.

Step Four - Design Implementation

After the design has been functionally verified, the next step is to implement the design using the Actel Designer software. The Designer software automatically places and routes the design and returns timing information. Use the tools that come with Designer to further optimize the design. Use Timer to perform static timing analysis on the design, ChipEditor or ChipPlanner to customize the I/O macro placement, PinEditor for I/O customization, SmartPower for power analysis, and NetlistViewer to view the netlist.

Step Five - Timing Simulation

After design implementation is done, it must be verified that the design meets timing specifications. After creating a test bench using WaveFormer Lite, a timing simulation is performed using the ModelSim VHDL or Verilog simulator.

Step Six - Device Programming

Once the timing simulation has been verified, a the programming file is created. Depending upon the device family, a Fuse, Bitstream, or STAPL programming file is generated.

Appendix C

MATLAB Code

Listing C.1: `lock_indicator.m`

```
disp 'Running␣lock␣indicator...'
downsample_factor = 0;
frequency_downsample = f_sampling/(2^downsample_factor);% No
    downsampling for now
threshold = 0.1;

downsample_factor = floor(f_sampling/frequency_downsample);

y_truncation_frac_resolution = 7;
y_sq_truncation_frac_resolution  = 10;
y_sq_truncation_resolution = y_sq_truncation_frac_resolution
    + 1;
y_truncation_resolution = y_truncation_frac_resolution + 1;
y_sq_truncation_resolution = y_sq_truncation_frac_resolution
    + 1;

y_truncated = fi_custom(y, 1, y_truncation_resolution,
    y_truncation_frac_resolution);
y_q_truncated = fi_custom(y_q, 1, y_truncation_resolution,
    y_truncation_frac_resolution);

y_sq = y_truncated.^2;
y_sq_q = y_q_truncated.^2;

y_sq_truncated = fi_custom(y_sq, 1,
    y_sq_truncation_resolution,
    y_sq_truncation_frac_resolution);
```

```
y_sq_q_truncated = fi_custom(y_sq_q, 1,
    y_sq_truncation_resolution,
    y_sq_truncation_frac_resolution);

y_sq_truncated_sum = y_sq_truncated - y_sq_q_truncated;

sum_y_resolution = y_sq_truncation_resolution+1;
sum_y_frac_resolution = y_sq_truncation_frac_resolution;

sum_y = downsample(fi_custom(y_sq_truncated_sum, 1,
    sum_y_resolution, sum_y_frac_resolution),
    downsample_factor);

locklpfpassbandfreq = 300;
locklpfstopbandfreq = 5000;
locklpfPassbandAttenuation = 1; %dB
locklpfStopbandAttenuation = 40; %dB
locklpfmatch = 'stopband'; %Band to match exactly

locklpfspecs = fdesign.lowpass(locklpfpassbandfreq,
    locklpfstopbandfreq, locklpfPassbandAttenuation,
    locklpfStopbandAttenuation, frequency_downsample);
locklpffilterobj= design(locklpfspecs, 'cheby2', '
    MatchExactly',locklpfmatch);

locklpfsosMatrix = locklpffilterobj.sosMatrix;
locklpfscale = locklpffilterobj.ScaleValues;
locklpforder =  locklpffilterobj.order;

[lock_lpf_b_double, lock_lpf_a_double] = sos2tf(
    locklpfsosMatrix, locklpfscale(2));
%threshold = threshold/locklpfscale(1);
threshold = 7;

%threshold = 10;
%[locklpforder, locklpfstop] = cheb2ord(2*locklpfpassbandfreq
    /frequency_downsample, 2*locklpfstopbandfreq/
    frequency_downsample, locklpfPassbandAttenuation,
    locklpfStopbandAttenuation) ;


%[lock_lpf_b_double, lock_lpf_a_double] = cheby2(locklpforder
    ,locklpfStopbandAttenuation , locklpfstop);

%locklpfresolution
```

```
%quantized_lock_lpf_arm_b_double = fi(lock_lpf_arm_b_double)
%quantized_lock_lpf_arm_b_double = fi(-lock_lpf_arm_a_double
   (2:3))

fvtool(lock_lpf_b_double, lock_lpf_a_double);

subcarop = filter(lock_lpf_b_double, lock_lpf_a_double,
   double(sum_y));

lock_lpf_b_coeff_frac_resolution = 14;
lock_lpf_b_coeff_resolution = 16;
lock_lpf_a_coeff_frac_resolution = 14;
lock_lpf_a_coeff_resolution = 16;
lock_lpf_word_overflow_extra_bits = 7;
lock_lpf_word_resolution = 25; %36
lock_lpf_word_frac_resolution = 6; %17

lock_lpf_b_coeffs = lock_lpf_b_double/lock_lpf_a_double(1);
lock_lpf_a_coeffs = lock_lpf_a_double(2:end)/
   lock_lpf_a_double(1);
lock_lpf_b = fi_custom(lock_lpf_b_coeffs, 1,
   lock_lpf_b_coeff_resolution ,
   lock_lpf_b_coeff_frac_resolution );
lock_lpf_a = fi_custom(lock_lpf_a_coeffs, 1,
   lock_lpf_a_coeff_resolution ,
   lock_lpf_a_coeff_frac_resolution );
lock_lpf_word_init= fi_custom(zeros(1,locklpforder), 1,
   lock_lpf_word_resolution , lock_lpf_word_frac_resolution )
   ;
lock_lpf_filter = iirdf2filter(locklpforder,lock_lpf_b,
   lock_lpf_a, lock_lpf_word_init, lock_lpf_word_resolution ,
    lock_lpf_word_frac_resolution);

lock_lpf_op = [];
for i = 1:length(sum_y)
  lock_lpf_op = [lock_lpf_op filter(lock_lpf_filter, sum_y(i)
     )];
end;

%lock_lpf_op = filter(locklpffilterobj, sum_y);
lock_lpf_op_1 = lock_lpf_op(1);
lock_lpf_op_frac_resolution = lock_lpf_op_1.FractionLength;
lock_lpf_op_resolution = lock_lpf_op_1.WordLength;

threshold_fi = fi_custom(threshold,1, lock_lpf_op_resolution,
    lock_lpf_op_frac_resolution);
subcarop_threshold = subcarop > threshold;
lock_lpf_op_threshold = lock_lpf_op > threshold_fi;

%Markov chain: Two consequtive positives: lock, two
   consequtive negatives: unlock

%States: unlocked, unconfirmed, locked, locked_unconfirmed
   (-1, 0, 1, 2)

state = -1; %confirmed

markovop = zeros(1,length(lock_lpf_op_threshold));

for i = 1:length(lock_lpf_op_threshold)
  if lock_lpf_op_threshold(i) == 1 && state == -1
    state = 0;
  elseif lock_lpf_op_threshold(i) == 1 && state == 0
    state = 1;
  elseif lock_lpf_op_threshold(i) == 0 && state == 1
    state = 2;
  elseif lock_lpf_op_threshold(i) == 0 && state == 2
    state = -1;
  elseif lock_lpf_op_threshold(i) == 0 && state == 0
    state = -1;
  elseif lock_lpf_op_threshold(i) == 1 && state == 2
    state = 1;
  end;

  if state == 1 || state == 2
    markovop(i) = 1;
  end;

end;

    t_p = 0:1/samples_per_bit:data_samples - (1/
       samples_per_bit);
figure, subplot(211), plot(t_p,e);
subplot(212), plot(t_p,subcarop);

figure, subplot(411), plot(t_p,e);
subplot(412), plot(t_p,lock_lpf_op);
subplot(413), plot(t_p,lock_lpf_op_threshold);
subplot(414), plot(t_p,markovop);
```

```
figure, subplot(311), plot(t_p,(y));
title('Demodulated␣data␣signal');
subplot(312), plot(t_p,e);
title('Error␣signal');
subplot(313), plot(t_p, markovop);
title('Lock␣signal');
axis([0 2 -0.5 1.5]);
xlim auto;
```

Listing C.2: `runsim.m`

```
%% Next iteration of Costas Loop Simulation
clear;
close all;
addpath('visualaids/');
addpath('testsamples/');
addpath('test/');
addpath('test/lock');
addpath('methods/');
addpath('classes/');

set_nco_carrier_random_phase = true;
run_testbench = true;
run_virtualization = true;
feedback = true;
only_noise_input = false;
gaussian_random_noise = true %False: uniform distribution
f_error_max = 1000;
f_errors_num = 2;
error_bits = 5;
data_samples =30;


f_error =  randi([0 f_error_max], 1, f_errors_num);
data_f_error = [1,randi([3, data_samples-1], 1, f_errors_num
   -1)];

if isunix
if run_virtualization
  tbdir='/home/prabhpreet/Documents/College/Notes/SEM8/
     DSPProject/Work/Libero/CostasLoop/simulation/';
  pause_instead_of_execute_testbench=true;
else
    tbdir='/home/prabhpreet/Documents/College/Notes/SEM8/
       DSPProject/Work/vhdl/';
    pause_instead_of_execute_testbench = false;
```

```
end
elseif ispc
    tbdir='P:\prabhpreet\Documents\College\Notes\SEM8\
       DSPProject\Work_Libero\CostasLoop\simulation\';
    pause_instead_of_execute_testbench = true;
end;
%% Testbench object
fwr = tbgen(tbdir);
%% System clock

f_system = 512*10^3;
f_sampling = f_system;
t_s = 1/f_sampling;

%% Input

adc_resolution = 12;
f_desired = randi([0 1]); %16kHz = 1
disp 'Input␣specs:␣carrier␣rate,␣data␣rate,␣f_error'
if f_desired
  f_input_carrier = 16*10^3
  f_data_rate = 4*10^3
else
  f_input_carrier = 8*10^3
  f_data_rate = 2*10^3
end
f_error
data_f_error

t_data_period = 1/f_data_rate;
samples_per_bit = f_sampling/f_data_rate;

t = 0:t_s:(t_data_period*data_samples)-t_s;

data = randi(0:1, 1, data_samples);
data(data == 0) = -1;

if set_nco_carrier_random_phase
    carrier_phase = rand(1)
else
        carrier_phase = 0.0625
end

baseband = [];
carrier = [];
```

```
f_error_current = 0;
for i = 1:length(data)
    baseband = [baseband data(i).*ones(1,samples_per_bit)];
    if any(i == data_f_error)
        f_error_current = f_error(data_f_error==i);
    end;
    carrier = [carrier sin(2*pi*(f_input_carrier+
       f_error_current)*t((i-1)*samples_per_bit+1:i*
       samples_per_bit) + 2*pi*carrier_phase)];

end

%carrier = sin(2*pi*(f_input_carrier+f_error)*t + 2*pi*
   carrier_phase);

if only_noise_input
  if gaussian_random_noise
    input_noise = normrnd(0,1,1,length(t));
  else
  input_noise = -1 + 2*randi(1, length(t));
end
else
input = (baseband.* carrier);% + ((2^-(adc_resolution-
   error_bits - 1)).*(randi([-((2^error_bits)) ((2^error_bits
   )-1)], 1, length(carrier))));

input_noise = bitnoise(input, adc_resolution, adc_resolution
   -1, error_bits);
%input = awgn(input, -100); %Noise
end

input_sampled = fi_custom(input_noise, 1, adc_resolution,
   adc_resolution-1); % 12 bit adc


%% NCO

nco_output_resolution = 5;
acc_word_resolution = 15;
frequency_word_resolution = 15;


intial_acc_phase = 0;

nco_obj = nco(frequency_word_resolution, acc_word_resolution,
    nco_output_resolution, intial_acc_phase);

f_desired_word = (f_input_carrier/f_sampling)*(2^
   acc_word_resolution);

x_resolution = adc_resolution+nco_output_resolution;
x_frac_resolution = x_resolution-2;

nco_phase = [];

i_nco = fi_custom([],1,nco_output_resolution,
   nco_output_resolution-1);
q_nco = fi_custom([], 1, nco_output_resolution,
   nco_output_resolution-1);


x = fi_custom([], 1,x_resolution, x_frac_resolution);
x_q = fi_custom([], 1,x_resolution, x_frac_resolution);

%% Arm filters
x_arm_coeff_resolution = 15;
x_arm_coeff_frac_resolution = 14;

y_arm_coeff_resolution = 16;
y_arm_coeff_frac_resolution = 13;

%y_resolution = x_resolution+y_arm_coeff_resolution;
%y_frac_resolution = x_frac_resolution +
   arm_coeff_frac_resolution;

y_frac_resolution = 26;
y_resolution = y_frac_resolution+(y_arm_coeff_resolution-
   y_arm_coeff_frac_resolution)+(x_resolution -
   x_frac_resolution);
y = fi_custom([],1, y_resolution, y_frac_resolution);
y_q = fi_custom([],1, y_resolution, y_frac_resolution);

Fpass = 5000;   % Passband Frequency: Original 6000
Fstop = 15000;  % Stopband Frequency
PassbandRipple = 5;
StopbandAttentuation = 40; %dB

[order, cutoff] = cheb2ord(Fpass/(f_sampling/2), Fstop/(
   f_sampling/2), PassbandRipple, StopbandAttentuation);

```

```
[arm_b_double, arm_a_double] = cheby2(order,
    StopbandAttentuation, cutoff);

%fvtool(arm_b_double, arm_a_double);

arm_b_coeffs = arm_b_double/arm_a_double(1);
arm_a_coeffs = arm_a_double(2:end)/arm_a_double(1);
arm_b = fi_custom(arm_b_coeffs, 1, x_arm_coeff_resolution,
    x_arm_coeff_frac_resolution);
arm_a = fi_custom(arm_a_coeffs, 1, y_arm_coeff_resolution,
    y_arm_coeff_frac_resolution);

x_init = fi_custom(zeros(1,order+1), 1, x_resolution,
    x_resolution-1);
y_init = fi_custom(zeros(1,order), 1, y_resolution,
    y_frac_resolution);
arm_filter_in_phase = iircustomFilter(order,arm_b, arm_a,
    x_init, y_init, y_resolution, y_frac_resolution);
arm_filter_quadrature = iircustomFilter(order,arm_b, arm_a,
    x_init, y_init, y_resolution, y_frac_resolution);

e_resolution = (2*y_resolution);
e_frac_resolution = 2*y_frac_resolution;

e = fi_custom([], 1, e_resolution, e_frac_resolution);
%% Loop filter

loop_filter_coeff_bw = 16;
loop_filter_coeff_bw_frac = loop_filter_coeff_bw-1;
lfv = 0.01;
%alpha = fi_custom(lfv, 1,loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);

%alpha = fi_custom(lfv*(1+(lfv*t_s/8)), 1,
    loop_filter_coeff_bw, loop_filter_coeff_bw_frac);
%beta = fi_custom(0, 1, loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);
%beta = fi_custom(0.0001, 1, loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);
%
% freq_add = fi_custom([],1,n_total_resolution,
    n_total_frac_resolution);

alpha = lfv*(1+(lfv*t_s/8));
beta = 0.0001;
```

```
%beta = (alpha^2)*t_s/4;

n_total_resolution = loop_filter_coeff_bw + e_resolution;
n_total_frac_resolution =  loop_filter_coeff_bw_frac +
    e_frac_resolution;
n_total = fi_custom([],1,n_total_resolution,
    n_total_frac_resolution);

loopfilter_order = 1;
loopfilter_b = fi_custom([alpha (beta-alpha)], 1,
    loop_filter_coeff_bw,loop_filter_coeff_bw_frac);
loopfilter_a = fi_custom(-1, 1, loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);


f_desired_word_fi = fi_custom(convert_natural(nco_obj,
    f_desired_word),1,n_total_resolution,
    n_total_frac_resolution);

e_init = fi_custom(zeros(1,loopfilter_order+1), 1,
    e_resolution, e_frac_resolution);
n_total_init = f_desired_word_fi;

loopfilter = iircustomFilter(loopfilter_order,loopfilter_b,
    loopfilter_a, e_init, n_total_init, n_total_resolution,
    n_total_frac_resolution);
%% Signal initialization


freq_add(1) = f_desired_word_fi;
nco_e = [convert_natural(nco_obj, f_desired_word)];

disp 'Simulating...'
for i = 1:length(t)
    nco_phase(i) =  nco_obj.accumalator;
    if feedback
        i_nco(i) = iterate(nco_obj,nco_e(i));
    else
        i_nco(i) = iterate(nco_obj, nco_e(1));
    end
    q_nco(i) = quadrature(nco_obj);

    x(i) = fi_custom(input_sampled(i)*i_nco(i), 1,
        x_resolution, x_frac_resolution);
    x_q(i) = fi_custom(input_sampled(i)*q_nco(i), 1,
        x_resolution, x_frac_resolution);

    y = [y fi_custom(filter(arm_filter_in_phase, x(i)), 1,
        y_resolution, y_frac_resolution)];
    y_q = [y_q fi_custom(filter(arm_filter_quadrature, x_q(i)
        ), 1, y_resolution, y_frac_resolution)];

    e = [e y(i)*y_q(i)];
    n_total = [n_total fi_custom(filter(loopfilter, e(i)), 1,
         n_total_resolution, n_total_frac_resolution)];
    nco_e(i+1) =  convert_signed(nco_obj, n_total(i));

end

discretization = zeros(1,length(y));
discretization(y < 0) = -1;
discretization(y >= 0) = 1;

subplot(511);
plot(t, baseband);
axis([1 2 -1.5 1.5]);
xlim auto;
title('Baseband␣signal');
subplot(512);
plot(t, discretization);
title('Quantized␣simulation␣demod␣signal');
axis([1 2 -1.5 1.5]);
xlim auto;
subplot(513);
plot(t, double(y));
title('Demod␣signal␣without␣quantization');
%subplot(614);
%plot(t, double(input_sampled));
subplot(514); plot(t, double(n_total));
title('Error␣signal␣after␣filtering');
% plot(t, double(carrier));
% title('Carrier');
% plot(t, double(i_nco));
% title('NCO in phase o/p');
% plot(t, double(q_nco));
% title('NCO quadrature phase o/p');
% plot(t, double(y_q));
% title('Demod quad signal without quantization');
subplot(515);plot(t, double(nco_e(2:end)));
title('Error␣signal␣quantized');
```

```

% if plot_n <= plot_r plot_temp = (plot_n*plot_c)-1; else
    plot_temp = (mod(plot_n, plot_r) * plot_c); end; subplot(
    plot_r, plot_c, plot_temp); plot_n = plot_n + 1;
% plot(t, double(nco_e(1:length(t))));
% title('NCO input');
% if plot_n <= plot_r plot_temp = (plot_n*plot_c)-1; else
    plot_temp = (mod(plot_n, plot_r) * plot_c); end; subplot(
    plot_r, plot_c, plot_temp); plot_n = plot_n + 1;
% plot(t, mod(double(nco_phase), 1));
% title('NCO accumalator phase');

figure
subplot(311)
plot(t, double(e));
subplot(312);
plot(t, double(n_total));
subplot(313);
plot(t, double(nco_e(2:end)));

lock_indicator
run_testbench_cmd
```

Listing C.3: runsim_noise.m

```
%% Next iteration of Costas Loop Simulation
clear;
close all;
addpath('visualaids/');
addpath('testsamples/');
addpath('test/');
addpath('test/lock');
addpath('methods/');
addpath('classes/');

set_nco_carrier_random_phase = true;
run_testbench = true;
run_virtualization = false;
feedback = true;
f_error_max = 1000;
f_errors_num = 2;
error_bits = 13;
data_samples =10;


f_error =  randi([0 f_error_max], 1, f_errors_num);
```

```matlab
data_f_error = [1,randi([3, data_samples-1], 1, f_errors_num-1)];

if isunix
if run_virtualization
  tbdir='/home/prabhpreet/Documents/College/Notes/SEM8/DSPProject/Work/Libero/CostasLoop/simulation/';
  pause_instead_of_execute_testbench=true;
else
    tbdir='/home/prabhpreet/Documents/College/Notes/SEM8/DSPProject/Work/vhdl/';
    pause_instead_of_execute_testbench = false;
end
elseif ispc
    tbdir='P:\prabhpreet\Documents\College\Notes\SEM8\DSPProject\Work_Libero\CostasLoop\simulation\';
    pause_instead_of_execute_testbench = true;
end;
%% Testbench object
fwr = tbgen(tbdir);
%% System clock

f_system = 512*10^3;
f_sampling = f_system;
t_s = 1/f_sampling;

%% Input

adc_resolution = 12;
f_desired = randi([0 1]); %16kHz = 1
disp 'Input specs: carrier rate, data rate, f_error'
if f_desired
  f_input_carrier = 16*10^3
  f_data_rate = 4*10^3
else
  f_input_carrier = 8*10^3
  f_data_rate = 2*10^3
end
f_error
data_f_error

t_data_period = 1/f_data_rate;
samples_per_bit = f_sampling/f_data_rate;

t = 0:t_s:(t_data_period*data_samples)-t_s;
```

```

data = randi(0:1, 1, data_samples);
data(data == 0) = -1;

if set_nco_carrier_random_phase
    carrier_phase = rand(1)
else
        carrier_phase = 0.0625
end

baseband = [];
carrier = [];
f_error_current = 0;
for i = 1:length(data)
    baseband = [baseband data(i).*ones(1,samples_per_bit)];
    if any(i == data_f_error)
        f_error_current = f_error(data_f_error==i);
    end;
    carrier = [carrier sin(2*pi*(f_input_carrier+
       f_error_current)*t((i-1)*samples_per_bit+1:i*
       samples_per_bit) + 2*pi*carrier_phase)];

end

%carrier = sin(2*pi*(f_input_carrier+f_error)*t + 2*pi*
   carrier_phase);

%input = (baseband.* carrier);% + ((2^-(adc_resolution-
   error_bits - 1)).*(randi([-((2^error_bits)) ((2^error_bits
   )-1)], 1, length(carrier))));

%input_noise = bitnoise(input, adc_resolution, adc_resolution
   -1, error_bits);
%input = awgn(input, -100); %Noise
input_noise = -1 + 2*rand(1,length(baseband));
input_sampled = fi_custom(input_noise, 1, adc_resolution,
   adc_resolution-1); % 12 bit adc


%% NCO

nco_output_resolution = 5;
acc_word_resolution = 15;
frequency_word_resolution = 15;

```

```

intial_acc_phase = 0;

nco_obj = nco(frequency_word_resolution, acc_word_resolution,
     nco_output_resolution, intial_acc_phase);

f_desired_word = (f_input_carrier/f_sampling)*(2^
    acc_word_resolution);

x_resolution = adc_resolution+nco_output_resolution;
x_frac_resolution = x_resolution-2;

nco_phase = [];

i_nco = fi_custom([],1,nco_output_resolution,
    nco_output_resolution-1);
q_nco = fi_custom([], 1, nco_output_resolution,
    nco_output_resolution-1);


x = fi_custom([], 1,x_resolution, x_frac_resolution);
x_q = fi_custom([], 1,x_resolution, x_frac_resolution);

%% Arm filters
x_arm_coeff_resolution = 15;
x_arm_coeff_frac_resolution = 14;

y_arm_coeff_resolution = 16;
y_arm_coeff_frac_resolution = 13;

%y_resolution = x_resolution+y_arm_coeff_resolution;
%y_frac_resolution = x_frac_resolution +
    arm_coeff_frac_resolution;

y_frac_resolution = 26;
y_resolution = y_frac_resolution+(y_arm_coeff_resolution-
    y_arm_coeff_frac_resolution)+(x_resolution -
    x_frac_resolution);
y = fi_custom([],1, y_resolution, y_frac_resolution);
y_q = fi_custom([],1, y_resolution, y_frac_resolution);

Fpass = 5000;    % Passband Frequency: Original 6000
Fstop = 15000;   % Stopband Frequency
PassbandRipple = 5;
StopbandAttentuation = 40; %dB
```

```

[order, cutoff] = cheb2ord(Fpass/(f_sampling/2), Fstop/(
    f_sampling/2), PassbandRipple, StopbandAttentuation);

[arm_b_double, arm_a_double] = cheby2(order,
    StopbandAttentuation, cutoff);

%fvtool(arm_b_double, arm_a_double);

arm_b_coeffs = arm_b_double/arm_a_double(1);
arm_a_coeffs = arm_a_double(2:end)/arm_a_double(1);
arm_b = fi_custom(arm_b_coeffs, 1, x_arm_coeff_resolution,
    x_arm_coeff_frac_resolution);
arm_a = fi_custom(arm_a_coeffs, 1, y_arm_coeff_resolution,
    y_arm_coeff_frac_resolution);

x_init = fi_custom(zeros(1,order+1), 1, x_resolution,
    x_resolution-1);
y_init = fi_custom(zeros(1,order), 1, y_resolution,
    y_frac_resolution);
arm_filter_in_phase = iircustomFilter(order,arm_b, arm_a,
    x_init, y_init, y_resolution, y_frac_resolution);
arm_filter_quadrature = iircustomFilter(order,arm_b, arm_a,
    x_init, y_init, y_resolution, y_frac_resolution);

e_resolution = (2*y_resolution);
e_frac_resolution = 2*y_frac_resolution;

e = fi_custom([], 1, e_resolution, e_frac_resolution);
%% Loop filter

loop_filter_coeff_bw = 16;
loop_filter_coeff_bw_frac = loop_filter_coeff_bw-1;
lfv = 0.01;
%alpha = fi_custom(lfv, 1,loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);

%alpha = fi_custom(lfv*(1+(lfv*t_s/8)), 1,
    loop_filter_coeff_bw, loop_filter_coeff_bw_frac);
%beta = fi_custom(0, 1, loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);
%beta = fi_custom(0.0001, 1, loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);
%
% freq_add = fi_custom([],1,n_total_resolution,
    n_total_frac_resolution);

alpha = lfv*(1+(lfv*t_s/8));
beta = 0.0001;
%beta = (alpha^2)*t_s/4;

n_total_resolution = loop_filter_coeff_bw + e_resolution;
n_total_frac_resolution =  loop_filter_coeff_bw_frac +
    e_frac_resolution;
n_total = fi_custom([],1,n_total_resolution,
    n_total_frac_resolution);

loopfilter_order = 1;
loopfilter_b = fi_custom([alpha (beta-alpha)], 1,
    loop_filter_coeff_bw,loop_filter_coeff_bw_frac);
loopfilter_a = fi_custom(-1, 1, loop_filter_coeff_bw,
    loop_filter_coeff_bw_frac);


f_desired_word_fi = fi_custom(convert_natural(nco_obj,
    f_desired_word),1,n_total_resolution,
    n_total_frac_resolution);

e_init = fi_custom(zeros(1,loopfilter_order+1), 1,
    e_resolution, e_frac_resolution);
n_total_init = f_desired_word_fi;

loopfilter = iircustomFilter(loopfilter_order,loopfilter_b,
    loopfilter_a, e_init, n_total_init, n_total_resolution,
    n_total_frac_resolution);
%% Signal initialization


freq_add(1) = f_desired_word_fi;
nco_e = [convert_natural(nco_obj, f_desired_word)];

disp 'Simulating...'
for i = 1:length(t)
    nco_phase(i) =  nco_obj.accumalator;
    if feedback
        i_nco(i) = iterate(nco_obj,nco_e(i));
    else
        i_nco(i) = iterate(nco_obj, nco_e(1));
    end
    q_nco(i) = quadrature(nco_obj);
    x(i) = fi_custom(input_sampled(i)*i_nco(i), 1,
        x_resolution, x_frac_resolution);
    x_q(i) = fi_custom(input_sampled(i)*q_nco(i), 1,
        x_resolution, x_frac_resolution);

    y = [y fi_custom(filter(arm_filter_in_phase, x(i)), 1,
        y_resolution, y_frac_resolution)];
    y_q = [y_q fi_custom(filter(arm_filter_quadrature, x_q(i)
        ), 1, y_resolution, y_frac_resolution)];

    e = [e y(i)*y_q(i)];
    n_total = [n_total fi_custom(filter(loopfilter, e(i)), 1,
         n_total_resolution, n_total_frac_resolution)];
    nco_e(i+1) =  convert_signed(nco_obj, n_total(i));

end

discretization = zeros(1,length(y));
discretization(y < 0) = -1;
discretization(y >= 0) = 1;

subplot(511);
axis([1 2 -1.5 1.5]);
xlim auto;
plot(t, baseband);
title('Baseband␣signal');
subplot(512);
axis([1 2 -1.5 1.5]);
xlim auto;
plot(t, discretization);
title('Quantized␣simulation␣demod␣signal');
subplot(513);
plot(t, double(y));
title('Demod␣signal␣without␣quantization');
%subplot(614);
%plot(t, double(input_sampled));
subplot(514); plot(t, double(n_total));
title('Error␣signal␣after␣filtering');
% plot(t, double(carrier));
% title('Carrier');
% plot(t, double(i_nco));
% title('NCO in phase o/p');
% plot(t, double(q_nco));
% title('NCO quadrature phase o/p');
% plot(t, double(y_q));
% title('Demod quad signal without quantization');
subplot(515);plot(t, double(nco_e(2:end)));
title('Error␣signal␣quantized');

% if plot_n <= plot_r plot_temp = (plot_n*plot_c)-1; else
    plot_temp = (mod(plot_n, plot_r) * plot_c); end; subplot(
    plot_r, plot_c, plot_temp); plot_n = plot_n + 1;
% plot(t, double(nco_e(1:length(t))));
% title('NCO input');
% if plot_n <= plot_r plot_temp = (plot_n*plot_c)-1; else
    plot_temp = (mod(plot_n, plot_r) * plot_c); end; subplot(
    plot_r, plot_c, plot_temp); plot_n = plot_n + 1;
% plot(t, mod(double(nco_phase), 1));
% title('NCO accumalator phase');

figure
subplot(311)
plot(t, double(e));
subplot(312);
plot(t, double(n_total));
subplot(313);
plot(t, double(nco_e(2:end)));

lock_indicator
run_testbench_cmd
```

Listing C.4: `customFilter.m`

```
classdef customFilter < handle
    %customfilter FIR filter class
    properties
        coeffs
        x
        order
    end

    methods
        function obj = customFilter(order, coeffs, x_init)
            obj.coeffs = coeffs;
            obj.order = order;
            obj.x = x_init;
        end

        function y = filter(obj, input)
            obj.x = [input obj.x(1:obj.order)];
            y = sum(obj.x.*obj.coeffs);
        end
    end

end
```

Listing C.5: `iircustomFilter.m`

```
classdef iircustomFilter < handle
    %customfilter FIR filter class
    properties
        b
        a
        x
        y
        order
        y_resolution
        y_frac_resolution
    end

    methods
        function obj = iircustomFilter(order, b, a, x_init,
            y_init, y_resolution, y_frac_resolution)
            obj.b = b;
            obj.a = a;
            obj.order = order;
            obj.x = x_init;
            obj.y = y_init;
            obj.y_resolution = y_resolution;
            obj.y_frac_resolution = y_frac_resolution;
        end

        function op = filter(obj, input, y_resolution,
            y_frac_resolution)
            obj.x = [input obj.x(1:obj.order)  ];
            b = sum(obj.x.*obj.b);
            a = sum(obj.a.* obj.y);
            op = fi_custom(b - a, 1, obj.y_resolution, obj.
                y_frac_resolution);
            if obj.order ~= 1
                obj.y = [op obj.y(1:obj.order -1)];
            else
                obj.y = op;
            end;
        end
    end
```

```

end
```

Listing C.6: `iirdf2filter.m`

```
classdef iirdf2filter< handle
    %Direct Form II IIR filter class
    properties
        b
        a
        word
        order
        word_resolution
        word_frac_resolution
    end

    methods
        function obj = iirdf2filter(order, b, a, word_init,
           word_resolution, word_frac_resolution)
            obj.b = b;
            obj.a = a;
            obj.order = order;
            obj.word = word_init;
            obj.word_resolution = word_resolution;
            obj.word_frac_resolution = word_frac_resolution;
        end

        function op = filter(obj, input)
     truncated_product_a = fi(-(obj.word.*obj.a), 1, obj.
        word_resolution, obj.word_frac_resolution);
     resized_input = fi(input, 1, obj.word_resolution, obj.
        word_frac_resolution);
     word_next =fi_custom(sum(truncated_product_a) +
        resized_input,1, obj.word_resolution,obj.
        word_frac_resolution);
      op = sum([word_next obj.word].*obj.b);
      obj.word = [word_next obj.word(1:obj.order-1)];
        end
    end

end
```

Listing C.7: `nco.m`

```
classdef nco < handle
    %NCO NCO class

    properties
        freq_bw
        accum_bw
        output_bw
        accumalator
    end

    methods
        function obj = nco(freq_bw,accum_bw,output_bw,
           phase_init)
            obj.freq_bw = freq_bw;
            obj.accum_bw = accum_bw;
            obj.output_bw = output_bw;
            obj.accumalator = fi(phase_init, 0, obj.accum_bw,
                 obj.accum_bw);
        end

        function op = iterate(obj, fw)
            obj.accumalator = obj.accumalator + convert(obj,
               fw);
            op = fi(sin(2*pi*double(obj.accumalator)), 1, obj
               .output_bw, obj.output_bw -1);
            %if(op == 0) op = 2^-obj.output_bw; end
        end

        function op = quadrature(obj)
            acc = obj.accumalator + fi(0.25, 0, obj.accum_bw,
                 obj.accum_bw);
            op = fi(sin(2*pi*double(acc)), 1, obj.output_bw,
               obj.output_bw -1);
            %if(op == 0) op = 2^-obj.output_bw; end
        end

        function op = iterate_natural(obj, fw)
           op = iterate(obj,convert_natural(obj,fw));
        end

        function f = convert_natural(obj, fw)
            m = mod(double(fw*2^-(obj.accum_bw)), 2^-(obj.
               accum_bw - obj.freq_bw));
            f =  fi(m, 0, obj.accum_bw, obj.accum_bw);
        end

        function f = convert(obj, fw)
```

```
            f =  fi(mod(double(fw), 2^-(obj.accum_bw - obj.
                freq_bw)), 0, obj.accum_bw, obj.accum_bw);
        end

        function f = convert_signed(obj, fw)
            temp = mod((mod(1 +double(fw), 1)*2^obj.accum_bw)
                , 2^obj.freq_bw);
            f = convert_natural(obj, floor(temp));
        end

        function [words, op] = get_coeffs(obj, filename)
            words = 0:(2^(obj.accum_bw-2)-1);
            temp = fi(sin(2*pi*(words/2^obj.accum_bw)), 1,
                obj.output_bw, obj.output_bw -1);
            op = bin(temp.');
            words_bin = bin(fi(words.', 0, obj.accum_bw-2, 0)
                );
            f = fopen(filename, 'w');
            for i = 1:length(words)
                fprintf(f, '"%s",␣', op(i,:));
            end
            fclose(f);
        end
    end

end
```

Listing C.8: `tbgen.m`

```
classdef tbgen < handle
    %TBGEN Testbench generation class

    properties
        filedir
    end

    methods


        function obj = tbgen(filedir)
            obj.filedir= filedir;
        end
        function write_file(obj, filename, input, power_scale
            )
            fileID=fopen(strcat(obj.filedir, filename), 'w');
            fprintf(fileID, '%d\n', floor(input.*2^(
                power_scale)));
            fclose(fileID);


        end

         function write_file_bin_frac(obj, filename, input)
            fileID=fopen(strcat(obj.filedir, filename), 'w');

            for i = 1:length(input)
                fprintf(fileID, '%s\n', bin(input(i)));
            end
            fclose(fileID);


        end

         function write_file_bin(obj, filename, input,
            power_scale, signed)
            fileID=fopen(strcat(obj.filedir, filename), 'w');
            if signed
                ip = fi(input, 1, power_scale+1, power_scale)
                    ;
            else
                ip = fi(input, 0, power_scale, power_scale);
            end;
            for i = 1:length(ip)
                fprintf(fileID, '%s\n', bin(ip(i)));
            end
            fclose(fileID);


        end

        function op=read_file_bin(obj, filename, signed)
            fileID = fopen(strcat(obj.filedir,filename), 'r')
                ;
            x = fgetl(fileID);
            op = [];
            while x ~= -1
                if signed
                    if(x(1)=='0')
                        y = bin2dec(x(2:end));
                    else
                        y = bin2dec(x(2:end)) - 2 ^ (length(x
                                ) - 1);
                    end
                else
                    y = bin2dec(x);
                end
                op = [op y];
                x = fgetl(fileID);
            end
            fclose(fileID);
        end

        function op=read_file(obj, filename)
            fileID = fopen(strcat(obj.filedir,filename), 'r')
                ;
            output = fscanf(fileID, '%d\n');
            op = output.';
            fclose(fileID);
        end


    end

end
```

Listing C.9: `bitnoise.m`

```
function output = bitnoise(input, word_length, fract_length,
    fract_noise_length)

  shift = fract_length - fract_noise_length;
  floored_output = floor((input).* (2^shift));
  shifted_output = floored_output .*(2^-shift) ;
  noise = (randi([0 ((2^fract_noise_length) - 1)],1, length(
      input) ).*2^(-fract_length));
output = shifted_output + noise;
end
```

Listing C.10: `fi_custom.m`

```
function output = fi_custom( input, signed, resolution,
    fract_word_size )
%fi_custom Used for removing/enforcing fixed point by editing
     function

     %output = input;
     %quantized_input = floor((input).* (2^fract_word_size)).*(2^-fract_word_size);
     %output = fi(quantized_input, signed, resolution, fract_word_size);
     %output = fi(input, signed, resolution, fract_word_size, 'RoundingMethod', 'Floor', 'OverflowAction', 'Wrap');
    output = fi(input, signed, resolution, fract_word_size, 'RoundingMethod', 'Floor');
 end
```

Listing C.11: test_carlock_lpf.m

```
write_file_bin_frac(fwr,'f_filter_in.txt',sum_y);
current_dir = pwd();
    cd(tbdir)
    if pause_instead_of_execute_testbench
        pause
    else
        system('./tbcarlock_lpf␣--vcd=tbcarlock_lpf.vcd')
    end
    cd(current_dir);
sum_y_int = double(sum_y)*2^(sum_y_frac_resolution);

lock_lpf_op_int = double(lock_lpf_op)*2^(lock_lpf_op_frac_resolution);
lock_lpf_op_cmp = read_file_bin(fwr, 'f_filter_out.txt', true);

disp 'Number␣of␣non␣matching␣samples'
sum(lock_lpf_op_cmp ~= lock_lpf_op_int)
lock_lpf_op_cmp(1:5)
lock_lpf_op_int(1:5)
lock_lpf_op_index = find(lock_lpf_op_int ~= lock_lpf_op_cmp);
```

Listing C.12: test_carlock.m

```
write_file_bin_frac(fwr,'f_raw_op_sin_in.txt',y);
write_file_bin_frac(fwr,'f_raw_op_cos_in.txt',y_q);

current_dir = pwd();
    cd(tbdir)
    if pause_instead_of_execute_testbench
        pause
    else
        system('./tbcarlock␣--vcd=tbcarlock.vcd')
    end
    cd(current_dir);
y_int = double(y)*2^(y_frac_resolution);
y_q_int = double(y_q)*2^(y_frac_resolution);
lock_int = markovop;
lock_cmp = read_file(fwr, 'f_lock_out.txt');

disp 'Number␣of␣non␣matching␣samples'
sum(lock_cmp ~= lock_int)
lock_cmp(1:5)
lock_int(1:5)
lock_index = find(lock_int ~= lock_cmp);
```

Listing C.13: `test_carlock_state_machine.m`

```
write_file_bin_frac(fwr,'f_carlock_lpf_op_in.txt',lock_lpf_op
   );
current_dir = pwd();
    cd(tbdir)
    if pause_instead_of_execute_testbench
        pause
    else
        system('./tbcarlock_state_machine␣--vcd=
           tbcarlock_state_machine.vcd')
    end
    cd(current_dir);

lock_lpf_op_int = double(lock_lpf_op)*2^(
   lock_lpf_op_frac_resolution);
lock_int = markovop;
lock_cmp = read_file(fwr, 'f_lock_out.txt');

disp 'Number␣of␣non␣matching␣samples'
sum(lock_cmp ~= lock_int)
lock_cmp(1:5)
lock_int(1:5)
lock_index = find(lock_int ~= lock_cmp);
```

Listing C.14: `test_carlock_square_truncate.m`

```
write_file_bin_frac(fwr,'f_raw_op_sin_in.txt',y);
write_file_bin_frac(fwr,'f_raw_op_cos_in.txt',y_q);
current_dir = pwd();
    cd(tbdir)
    if pause_instead_of_execute_testbench
        pause
    else
```

```
        system('./tbcarlock_square_truncate␣--vcd=
           tbcarlock_square_truncate.vcd')
    end
    cd(current_dir);
y_int = double(y)*2^(y_frac_resolution);
y_q_int = double(y_q)*2^(y_frac_resolution);
y_truncated_int = double(y_truncated)*2^(
   y_truncation_resolution);
y_q_truncated_int = double(y_q_truncated)*2^(
   y_truncation_resolution);
y_sq_int = double(y_sq)*2^(2*y_truncation_resolution);
y_sq_q_int = double(y_sq_q)*2^(2*y_truncation_resolution);
y_sq_truncated_int = double(y_sq_truncated)*2^(
   y_sq_truncation_frac_resolution);
y_sq_q_truncated_int = double(y_sq_q_truncated)*2^(
   y_sq_truncation_frac_resolution);
y_sq_truncated_sum_int = double(y_sq_truncated_sum)*2^(
   y_sq_truncation_frac_resolution);
sum_y_int = double(sum_y)*2^(sum_y_frac_resolution);
sum_y_cmp = read_file_bin(fwr, 'f_filter_in.txt', true);
disp 'Number␣of␣non␣matching␣samples'
sum(sum_y_cmp ~= sum_y_int)
sum_y_cmp(1:5)
sum_y_int(1:5)
sum_y_index = find(sum_y_int ~= sum_y_cmp);
```

Listing C.15: `test_lpf.m`

```
write_file_bin_frac(fwr,'x_data_in.txt', x);
current_dir = pwd();
    cd(tbdir)
    if pause_instead_of_execute_testbench
        pause
    else
        system('./tblpf␣--vcd=tblpf.vcd')
    end
    cd(current_dir);
x_int = double(x)*2^(x_frac_resolution);
y_int = double(y)*2^(y_frac_resolution);
y_cmp = read_file_bin(fwr, 'y_data_out.txt', true);
sum(y_cmp ~= y_int)
y_cmp(1:5)
y_int(1:5)
```

Listing C.16: `run_testbench_cmd.m`

```
if run_testbench
  disp 'Running␣testbench'
  write_file(fwr, 'f_desired.txt', f_desired, 0);
  write_file(fwr, 'carrier_in.txt', double(input_sampled),
     11);
  write_file(fwr, 'adc_mux_in.txt', 3, 0);
  baseband(baseband < 0) = 0;
  discretization(discretization < 0) = 0;
  current_dir = pwd();
  cd(tbdir)
  if pause_instead_of_execute_testbench
    pause
    else
      system('./tbcostas␣--vcd=tbcostas.vcd');
    end
    cd(current_dir);
    dop = read_file(fwr, 'op_out.txt');
    lock= read_file(fwr, 'lock_out.txt');

    disp 'Input␣specs:␣carrier␣rate,␣data␣rate,␣f_error'
    f_input_carrier
    f_data_rate
    f_error
    data_f_error

    disp 'Matching␣samples␣with␣discretized␣output'
    sum(dop == discretization)

    figure;
    t_p = 0:1/samples_per_bit:data_samples - (1/
       samples_per_bit);
    subplot(5,1,1), plot(t_p,baseband);axis([1 2 -1.5 1.5]);
    xlim auto;

    title('Baseband␣signal');
    subplot(5,1,2), plot(t_p,discretization);axis([1 2 -1.5
       1.5]);
    xlim auto;

    title('MATLAB␣simulation:␣Demodulated␣signal');
    subplot(5,1,3), plot(t_p,dop);axis([1 2 -1.5 1.5]);
    xlim auto;
    if pause_instead_of_execute_testbench
      title('Microsemi␣simulation:␣Demodulated␣signal');
    else
```

```
      title('GHDL␣simulation:␣Demodulated␣signal');
end

subplot(5,1,4), plot(t_p,markovop);axis([1 2 -1.5 1.5]);
xlim auto;
title('MATLAB␣simulation:␣Lock␣indicator');

subplot(5,1,5), plot(t_p,lock);axis([1 2 -1.5 1.5]);
xlim auto;
if pause_instead_of_execute_testbench
  title('Microsemi␣simulation:␣Lock␣indicator');
else
  title('GHDL␣simulation:␣Lock␣indicator');
end

 end
```

Listing C.17: `test_loopfilter.m`

```
write_file_bin_frac(fwr, 'mult_error_op_in.txt', e);
current_dir = pwd();
    cd(tbdir)
    if pause_instead_of_execute_testbench
        pause
    else
        system('./tbloopfilter␣--vcd=tbloopfilter.vcd')
    end
    cd(current_dir);
e_int = double(e)*2^(e_frac_resolution);
nco_e_int = double(nco_e)*2^(frequency_word_resolution);
nco_e_cmp = read_file_bin(fwr, 'f_word_out.txt', false);
sum(nco_e_int(1:end-1) ~= nco_e_cmp)
nco_e_int(1:5)
nco_e_cmp(1:5)
figure, subplot(211), plot(nco_e_int)
subplot(212), plot(nco_e_cmp)
```

Listing C.18: `test_filter.m`

```
n_total_test = fi_custom([],1,n_total_resolution,
    n_total_frac_resolution);

for i = 1:length(t)
    n_total_test = [n_total_test fi_custom(filter(loopfilter,
        e_cmp_test(i)), 1, n_total_resolution,
       n_total_frac_resolution)];
```

```
end
```

Listing C.19: `fi_custom.m`

```
function output = fi_custom( input, signed, resolution,
    fract_word_size )
%fi_custom Used for removing/enforcing fixed point by editing
     function
    output = floor((input).* (2^fract_word_size)).*(2^-
        fract_word_size);
end
```

Listing C.20: `runsim_data_only.m`

```
%% Assist testbench without MATLAB simulation
clear;
close all;

set_nco_carrier_random_phase = true;
run_testbench = true;
run_virtualization = false;
feedback = true;
only_noise_input = true
gaussian_random_noise = true %False: uniform distribution
f_error_max = 1000;
f_errors_num = 3;
error_bits = 5;
data_samples =10000;


f_error =  randi([0 f_error_max], 1, f_errors_num);
data_f_error = [1,randi([3, data_samples-1], 1, f_errors_num
    -1)];

if isunix
if run_virtualization
  tbdir='/home/prabhpreet/Documents/College/Notes/SEM8/
      DSPProject/Work/Libero/CostasLoop/simulation/';
  pause_instead_of_execute_testbench=true;
else
    tbdir='/home/prabhpreet/Documents/College/Notes/SEM8/
        DSPProject/Work/vhdl/';
    pause_instead_of_execute_testbench = false;
end
elseif ispc
```

```
    tbdir='P:\prabhpreet\Documents\College\Notes\SEM8\
       DSPProject\Work_Libero\CostasLoop\simulation\';
    pause_instead_of_execute_testbench = true;
end;
%% Testbench object
fwr = tbgen(tbdir);
%% System clock

f_system = 512*10^3;
f_sampling = f_system;
t_s = 1/f_sampling;

%% Input

adc_resolution = 12;
f_desired = randi([0 1]); %16kHz = 1
if f_desired
  f_input_carrier = 16*10^3;
  f_data_rate = 4*10^3;
else
  f_input_carrier = 8*10^3;
  f_data_rate = 2*10^3;
end
t_data_period = 1/f_data_rate;
samples_per_bit = f_sampling/f_data_rate;

t = 0:t_s:(t_data_period*data_samples)-t_s;

data = randi(0:1, 1, data_samples);
data(data == 0) = -1;

if set_nco_carrier_random_phase
    carrier_phase = rand(1)
else
        carrier_phase = 0.0625
end

baseband = [];
carrier = [];
f_error_current = 0;
for i = 1:length(data)
    baseband = [baseband data(i).*ones(1,samples_per_bit)];
    if any(i == data_f_error)
        f_error_current = f_error(data_f_error==i);
    end;
    carrier = [carrier sin(2*pi*(f_input_carrier+f_error_current)*t((i-1)*samples_per_bit+1:i*samples_per_bit) + 2*pi*carrier_phase)];

end

if only_noise_input
  if gaussian_random_noise
    input_noise = wgn(1,length(t),0);
  else
  input_noise = -1 + 2*randi(1, length(t));
end
else

input = (baseband.* carrier);
input_noise = bitnoise(input, adc_resolution, adc_resolution-1, error_bits);
end

input_sampled = bitnoise(input_noise, adc_resolution, adc_resolution-1, error_bits);
disp 'Running␣testbench'
write_file(fwr, 'f_desired.txt', f_desired, 0);
write_file(fwr, 'carrier_in.txt', double(input_sampled), 11);
write_file(fwr, 'adc_mux_in.txt', 3, 0);
baseband(baseband < 0) = 0;
current_dir = pwd();
cd(tbdir)
if pause_instead_of_execute_testbench
  disp 'Paused.␣Press␣enter␣to␣continue'
  pause
else
  system('./tbcostas␣--vcd=tbcostas.vcd');
end
cd(current_dir);
dop = read_file(fwr, 'op_out.txt');
lock= read_file(fwr, 'lock_out.txt');

disp 'Input␣specs:␣carrier␣rate,␣data␣rate,␣f_error'
f_input_carrier
f_data_rate
f_error
data_f_error


```

```
figure;
t_p = 0:1/samples_per_bit:data_samples - (1/samples_per_bit);
subplot(3,1,1), plot(t_p,baseband);axis([1 2 -1.5 1.5]);
xlim auto;
title('Baseband␣signal');

subplot(3,1,2), plot(t_p,dop);axis([1 2 -1.5 1.5]);
xlim auto;
if pause_instead_of_execute_testbench
  title('Microsemi␣simulation:␣Demodulated␣signal');
else
  title('GHDL␣simulation:␣Demodulated␣signal');
end

subplot(3,1,3), plot(t_p,lock);axis([1 2 -1.5 1.5]);
xlim auto;
if pause_instead_of_execute_testbench
  title('Microsemi␣simulation:␣Lock␣indicator');
else
  title('GHDL␣simulation:␣Lock␣indicator');
end
```

Listing C.21: `bitnoise.m`

```
function output = bitnoise(input, word_length, fract_length,
    fract_noise_length)

  shift = fract_length - fract_noise_length;
    output = floor((input).* (2^shift)).*(2^-shift) + (randi
        ([0 ((2^fract_noise_length) - 1)],1, length(input) )
        .*2^(-fract_length));
end
```

Listing C.22: `tbgen.m`

```
classdef tbgen < handle
    %TBGEN Testbench generation class

    properties
        filedir
    end

    methods


        function obj = tbgen(filedir)
```

```
            obj.filedir= filedir;
        end
        function write_file(obj, filename, input, power_scale
           )
            fileID=fopen(strcat(obj.filedir, filename), 'w');
            fprintf(fileID, '%d\n', floor(input.*2^(
               power_scale)));
            fclose(fileID);


        end
        function op=read_file(obj, filename)
            fileID = fopen(strcat(obj.filedir,filename), 'r')
               ;
            output = fscanf(fileID, '%d\n');
            op = output.';
            fclose(fileID);
        end


    end

end
```

Appendix D

VHDL Code

Listing D.1: `tbloopfilter.vhd`

```vhdl
-- tbloopfilter.vhd

-- tbloopfilter.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tbloopfilter is
end tbloopfilter;

architecture tbloopfilter_arch of tbloopfilter is
  component loopfilter is
    port(clk, reset: in std_logic;
        x_in:in signed(61 downto 0);
        f_desired: in std_logic;
        f_word_output: out unsigned(14 downto 0));
  end component;


    --TB only:
    constant t: time := 3906200 ps; --time period of 64khz,
       to gen 64khz clock
    signal clk, reset: std_logic;
    signal mult_error_op: signed(61 downto 0);
    signal f_desired: std_logic;
    signal f_word_output: unsigned(14 downto 0);
```

```
--for all : nco_input_multiplier use entity work.
   nco_input_multiplier;

begin
M0:f_desired <= '0';
R0:loopfilter port map(clk, reset, mult_error_op,f_desired,
    f_word_output);

TB:process
  file mult_error_op_in: text open read_mode is "
     mult_error_op_in.txt";
  file f_word_out: text open write_mode is "f_word_out.txt"
     ;

  variable l:line;
  variable mult_error_op_bit_vector: bit_vector(
     mult_error_op'range);
  variable i:integer;
  begin

    reset <= '0';
    wait for t/2;
    reset <= '1';
    clk <= '1';
    wait for t/2;
    reset <= '0';
    clk <= '0';
    wait for t/2;
    while not endfile(mult_error_op_in) loop
      readline(mult_error_op_in, l);
      read(l, mult_error_op_bit_vector);
      mult_error_op <= signed(to_stdlogicvector(
         mult_error_op_bit_vector));
      wait for t/3;
      clk <= '0';
      wait for t/3;
      write(l, to_bitvector(std_ulogic_vector(f_word_output
         )));
      writeline(f_word_out, l);
      clk <= '1';
      wait for t/3;
    end loop;
    report "End␣simulation" severity failure;
end process;

end tbloopfilter_arch;
```

Listing D.2: `loopfilter.vhd`

```
-- loopfilter.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity loopfilter is
  port(clk, reset: in std_logic;
      x_in:in signed(61 downto 0);
      f_desired: in std_logic;
      f_word_output: out unsigned(14 downto 0));
end loopfilter;


architecture loopfilter_arch of loopfilter is
  --Bit shift by 4 bits.
  constant taps: integer := 1;
  constant x_coeff_length : integer := 16;
  constant x_coeff_frac_length: integer := 15;
  constant x_frac_length: integer := 52;
  signal total_sum: signed(x_in'length + x_coeff_length-1
     downto 0);
  signal f_desired_word: signed(total_sum'range);
  constant y_frac_length: integer := x_coeff_frac_length +
     x_frac_length;
  constant f_word_frac_length: integer := f_word_output'
     length -1;


  component dffregister is
    generic (word_size: integer);
    port(clk, reset: in std_logic;
    d:in signed(word_size-1 downto 0);
    reset_word: in signed(word_size-1 downto 0);
    q:out signed(word_size-1 downto 0));
  end component;

  component fpresizer is
    generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
    port(a: in signed(a_adsize+a_bdsize -1 downto 0);
        b: out signed(b_adsize + b_bdsize-1 downto 0));
  end component;


  type x_word_array is array(taps downto 0) of signed(x_in'
     range);
  type x_coeff_array is array(taps downto 0) of signed(
     x_coeff_length-1 downto 0);
  type x_mult_array is array(taps downto 0) of signed(
     x_coeff_length+x_in'length -1 downto 0);


  constant x_coeffs: x_coeff_array := ("1111111010111011","
     0000000101000111");

  signal x_words: x_word_array;
  signal y_words, x_y_sum_arr: signed(total_sum'range);
  signal x_mult_words, x_sum_words: x_mult_array;
  signal f_word: signed(f_word_output'range);
  constant f_word_plus_one: signed(f_word_output'range) :=
     ('0', others => '1');
  begin
    X_GEN_DFF_SHIFT: for i in 0 to taps generate
      X_INPUT: if i = 0 generate
        T0:dffregister
          generic map(word_size => x_in'length)
          port map(clk, reset, x_in, (others => '0'), x_words
             (i));
        end generate X_INPUT;

      X_SHIFT_REGS: if i > 0 generate
      TX: dffregister
        generic map(word_size => x_in'length)
        port map(clk, reset, x_words(i-1),(others => '0'),
           x_words(i));
      end generate X_SHIFT_REGS;

    end generate X_GEN_DFF_SHIFT;

    FD: f_desired_word <= (62=> '1', others => '0') when
       f_desired = '1' else (61=> '1', others => '0');
    T0:dffregister
          generic map(word_size => total_sum'length)
          port map(clk, reset, total_sum, f_desired_word ,
             y_words);

    X_GEN_MULT: for i in 0 to taps generate
```

```
      MX: x_mult_words(i) <= x_words(i) * x_coeffs(i);
    end generate X_GEN_MULT;

    X_GEN_SUM: for i in 0 to taps generate
      X_SUM_0: if i = 0 generate
        S0: x_sum_words(0) <= x_mult_words(0);
      end generate X_SUM_0;

      X_SUM_ELSE: if i > 0 generate
        SX: x_sum_words(i) <= x_sum_words(i-1) + x_mult_words
            (i);
      end generate X_SUM_ELSE;
    end generate X_GEN_SUM;

    X_Y_SUM: fpresizer
            generic map(a_adsize =>(x_in'length -
               x_frac_length)+(x_coeff_length -
               x_coeff_frac_length),
                        a_bdsize=> (x_frac_length+
                           x_coeff_frac_length),
                        b_adsize=> (total_sum'length -
                           y_frac_length),
                        b_bdsize=> y_frac_length )
            port map(x_sum_words(taps), x_y_sum_arr);

    GEN_Y:total_sum <= y_words + x_y_sum_arr;
    f_word(f_word'length -1 downto 0) <= total_sum(
       y_frac_length-1 downto y_frac_length-
       f_word_frac_length-1);
    f_word_output <= unsigned(f_word) when f_word(f_word'
       length-1) = '0' else
                           unsigned(f_word + f_word_plus_one)
                               when f_word(f_word'length-1) =
                               '1';


end loopfilter_arch;
```

Listing D.3: tbcarlock_lpf.vhd

```
--tbcarlock_lpf.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

```

```
entity tbcarlock_lpf is
end tbcarlock_lpf;

architecture tbcarlock_lpf_arch of tbcarlock_lpf is
  constant t: time := 3906200 ps; --time period of 64khz, to
     gen 64khz clock
  component carlock_lpf is
    port(clk,reset: in std_logic;
      filter_in: in signed(11 downto 0);
      filter_out: out signed(42 downto 0));
  end component;
  signal clk,reset: std_logic;
        signal filter_in: signed(11 downto 0);
  signal filter_out:signed(42 downto 0);
begin
  C0: carlock_lpf port map(clk,reset, filter_in,filter_out);

  tb:process
    file f_filter_in: text open read_mode is "f_filter_in.txt
       ";
    file f_filter_out: text open write_mode is "f_filter_out.
       txt";

    variable l:line;
    variable filter_in_vector: bit_vector(filter_in'range);

  begin
    reset <= '0';
    clk <= '0';
    wait for t/2;
    reset <= '1';
    wait for t/2;
    reset <= '0';
    wait for t/2;
    while not endfile(f_filter_in) loop
      readline(f_filter_in, l);
      read(l,filter_in_vector);
      filter_in <= signed(to_stdlogicvector(filter_in_vector)
         );
      wait for t/2;
      clk <= '1';
      wait for t/2;
      clk <= '0';

      write(l, to_bitvector(std_logic_vector(filter_out)));
      writeline(f_filter_out, l);

    end loop;
    report "End␣simulation" severity failure;
  end process;


end tbcarlock_lpf_arch;
```

Listing D.4: carlock_square_truncate.vhd

```
--carlock_square_truncate.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity carlock_square_truncate is
    port(raw_op_sin,raw_op_cos: in signed(30 downto 0); --Q5
        .26
      filter_in: out signed(11 downto 0)); --Q2.10
end entity;

architecture carlock_square_truncate_arch of
   carlock_square_truncate is
  signal sin_input_truncation, cos_input_truncation: signed(7
      downto 0); --Q1.7 from Q5.26
  signal sin_squared_output_untruncated,
     cos_squared_output_untruncated: signed(15 downto 0); --
     Q2.14
  signal sin_squared_output_truncated,
     cos_squared_output_truncated: signed(11 downto 0);
  component fptruncate is
    generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
    port(a: in signed(a_adsize+a_bdsize -1 downto 0);
         b: out signed(b_adsize + b_bdsize-1 downto 0));
  end component;
begin
  IP_TRUNC_SIN: fptruncate
    generic map(a_adsize => 5, a_bdsize => 26, b_adsize => 1,
        b_bdsize => 7)
    port map(raw_op_sin, sin_input_truncation);

  IP_TRUNC_COS: fptruncate
    generic map(a_adsize => 5, a_bdsize => 26, b_adsize => 1,
        b_bdsize => 7)
    port map(raw_op_cos, cos_input_truncation);


  --input_truncation <= raw_op_sin(26 downto 19);
  SQ_SIN:sin_squared_output_untruncated <=
     sin_input_truncation*sin_input_truncation;
  SQ_COS:cos_squared_output_untruncated <=
     cos_input_truncation*cos_input_truncation;


  --filter_in <= squared_output_untruncated(15 downto 4);
  SQ_TRUNC_SIN: fptruncate
    generic map(a_adsize=> 2, a_bdsize=> 14, b_adsize=>2,
       b_bdsize=> 10)
    port map(sin_squared_output_untruncated,
       sin_squared_output_truncated);
    SQ_TRUNC_COS: fptruncate
    generic map(a_adsize=> 2, a_bdsize=> 14, b_adsize=>2,
       b_bdsize=> 10)
    port map(cos_squared_output_untruncated,
       cos_squared_output_truncated);

  DIFF_GEN: filter_in <= sin_squared_output_truncated -
     cos_squared_output_truncated;

end carlock_square_truncate_arch;
```

Listing D.5: tbcarlock_state_machine.vhd

```
--tbcarlock_state_machine.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tbcarlock_state_machine is
end tbcarlock_state_machine;

architecture tbcarlock_state_machine_arch of
   tbcarlock_state_machine is
  constant t: time := 3906200 ps; --time period of 64khz, to
     gen 64khz clock
  component carlock_state_machine is
    port(carlock_lpf_op:in signed(42 downto 0); --Q5.11
      clk, reset: in std_logic;
      lock: out std_logic);
    end component;
  signal clk, reset,lock: std_logic;
  signal carlock_lpf_op:signed(42 downto 0);
begin
  C0: carlock_state_machine port map(carlock_lpf_op, clk,
     reset, lock);

  tb:process
    file f_carlock_lpf_op_in: text open read_mode is "
       f_carlock_lpf_op_in.txt";
    file f_lock_out: text open write_mode is "f_lock_out.txt"
       ;

    variable l:line;
    variable carlock_lpf_op_vector: bit_vector(carlock_lpf_op
       'range);

  begin
    reset <= '0';
    clk <= '0';
    wait for t/2;
    reset <= '1';
    wait for t/2;
    reset <= '0';
    wait for t/2;
    while not endfile(f_carlock_lpf_op_in) loop
      readline(f_carlock_lpf_op_in, l);
      read(l,carlock_lpf_op_vector);
      carlock_lpf_op <= signed(to_stdlogicvector(
         carlock_lpf_op_vector));

    wait for t/2;
      clk <= '1';
      wait for t/2;
      clk <= '0';
      write(l, to_bit(lock));
      writeline(f_lock_out, l);

    end loop;
    report "End␣simulation" severity failure;
  end process;


end tbcarlock_state_machine_arch;
```

Listing D.6: `carlock_state_machine.vhd`

```vhdl
--carlock_state_machine.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- Moore machine: OP doesn't depend directly on input
entity carlock_state_machine is
    port(carlock_lpf_op:in signed(42 downto 0); --Q5.11
      clk, reset: in std_logic;
      lock: out std_logic);
  end carlock_state_machine;

architecture carlock_state_machine_arch of
    carlock_state_machine is
  type carlock_state_type is (s0,s1,s2,s3);
  signal carlock_state: carlock_state_type;
  signal threshold_signal: boolean;

  constant threshold: signed(carlock_lpf_op'range):= "
      0000000000000000000011100000000000000000000";
-- :="0000000000000000000101000001010110101101111";
begin
  STATE_CHANGE:process(reset, clk, carlock_lpf_op)
  begin
    if (reset = '1') then
      carlock_state <= s0;
    elsif rising_edge(clk) then
      if threshold_signal = true then
        case carlock_state is
          when s0 =>
            carlock_state <= s1;
          when s1 =>
            carlock_state <= s2;
          when s2 =>
            carlock_state <= s2;
          when s3 =>
            carlock_state <= s2;
        end case;
      elsif threshold_signal = false then
        case carlock_state is
          when s0 =>
            carlock_state <= s0;
          when s1 =>
            carlock_state <= s0;
          when s2 =>
```

```
            carlock_state <= s3;
          when s3 =>
            carlock_state <= s0;
        end case;

      end if;
    end if;
  end process;

  THRESH: threshold_signal <= carlock_lpf_op > threshold;

  STATE_BASED_OP: with carlock_state select lock <=
      '0' when s0,
      '0' when s1,
      '1' when s2,
      '1' when s3;

end carlock_state_machine_arch;
```

Listing D.7: tbcarlock_square_truncate.vhd

```
--tbcarlock_square_truncate.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tbcarlock_square_truncate is
end tbcarlock_square_truncate;

architecture tbcarlock_square_truncate_arch of
    tbcarlock_square_truncate is
  constant t: time := 3906200 ps; --time period of 64khz, to
     gen 64khz clock
  component carlock_square_truncate is
    port(raw_op_sin,raw_op_cos: in signed(30 downto 0); --Q5
        .26
      filter_in: out signed(11 downto 0)); --Q2.10
  end component;
  signal raw_op_sin,raw_op_cos:signed(30 downto 0);
  signal filter_in:signed(11 downto 0); --Q2.10
begin
  C0: carlock_square_truncate port map(raw_op_sin,raw_op_cos,
      filter_in);

  tb:process
```

```
    file f_raw_op_sin_in: text open read_mode is "
       f_raw_op_sin_in.txt";
    file f_raw_op_cos_in: text open read_mode is "
       f_raw_op_cos_in.txt";
    file f_filter_in: text open write_mode is "f_filter_in.
       txt";

    variable l:line;
    variable raw_op_sin_vector: bit_vector(raw_op_sin'range);
    variable raw_op_cos_vector: bit_vector(raw_op_cos'range);

  begin

    wait for t;
    while not endfile(f_raw_op_sin_in) loop
      readline(f_raw_op_sin_in, l);
      read(l,raw_op_sin_vector);
      raw_op_sin <= signed(to_stdlogicvector(
         raw_op_sin_vector));
      readline(f_raw_op_cos_in, l);
      read(l,raw_op_cos_vector);
      raw_op_cos <= signed(to_stdlogicvector(
         raw_op_cos_vector));
      wait for t;
      write(l, to_bitvector(std_logic_vector(filter_in)));
      writeline(f_filter_in, l);

    end loop;
    report "End␣simulation" severity failure;
  end process;


end tbcarlock_square_truncate_arch;
```

Listing D.8: `tbcarlock.vhd`

```
--tbcarlock.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tbcarlock is
end tbcarlock;

architecture tbcarlock_arch of tbcarlock is
```

```
  constant t: time := 3906200 ps; --time period of 64khz, to
     gen 64khz clock
  component carlock is
  port(clk, reset: in std_logic;
       raw_op_sin,raw_op_cos:in signed(30 downto 0);
    lock: out std_logic);
  end component;
  signal clk, reset,lock: std_logic;
  signal raw_op_sin,raw_op_cos:signed(30 downto 0);
begin
  C0: carlock port map(clk, reset, raw_op_sin,raw_op_cos,
     lock);

  tb:process
    file f_raw_op_sin_in: text open read_mode is "
       f_raw_op_sin_in.txt";
    file f_raw_op_cos_in: text open read_mode is "
       f_raw_op_cos_in.txt";
    file f_lock_out: text open write_mode is "f_lock_out.txt"
       ;

    variable l:line;
    variable raw_op_sin_vector: bit_vector(raw_op_sin'range);
    variable raw_op_cos_vector: bit_vector(raw_op_cos'range);

  begin
    reset <= '0';
    clk <= '0';
    wait for t/2;
    reset <= '1';
    wait for t/2;
    reset <= '0';
    wait for t/2;
    while not endfile(f_raw_op_sin_in) loop
      readline(f_raw_op_sin_in, l);
      read(l,raw_op_sin_vector);
      raw_op_sin <= signed(to_stdlogicvector(
         raw_op_sin_vector));

      readline(f_raw_op_cos_in, l);
      read(l,raw_op_cos_vector);
      raw_op_cos <= signed(to_stdlogicvector(
         raw_op_cos_vector));
      wait for t/2;
      clk <= '1';
      wait for t/2;
      clk <= '0';
      write(l, to_bit(lock));
      writeline(f_lock_out, l);

    end loop;
    report "End␣simulation" severity failure;
  end process;


end tbcarlock_arch;
```

Listing D.9: `carlock.vhd`

```
--carlock.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity carlock is
  port(clk, reset: in std_logic;
       raw_op_sin,raw_op_cos:in signed(30 downto 0);
    lock: out std_logic);
end carlock;


architecture carlock_arch of carlock is

  component carlock_square_truncate
    port(raw_op_sin,raw_op_cos: in signed(30 downto 0);
      filter_in: out signed(11 downto 0));
  end component;

  component carlock_lpf is
    port(clk,reset: in std_logic;
      filter_in: in signed(11 downto 0);
      filter_out: out signed(42 downto 0));
  end component;

  component carlock_state_machine is
    port(carlock_lpf_op:in signed(42 downto 0);
      clk, reset: in std_logic;
      lock: out std_logic);
  end component;

  signal filter_in: signed(11 downto 0);
  signal filter_out: signed(42 downto 0);
begin
  TRUNCATE: carlock_square_truncate port map(raw_op_sin,
     raw_op_cos, filter_in);
  LPF: carlock_lpf port map(clk,reset,filter_in, filter_out);
  STATEMACHINE: carlock_state_machine port map(filter_out,
     clk, reset, lock);

end carlock_arch;
```

Listing D.10: carlock_lpf.vhd

```
-- carlock_lpf.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity carlock_lpf is
    port(clk,reset: in std_logic;
      filter_in: in signed(11 downto 0);
      filter_out: out signed(42 downto 0));
end carlock_lpf;


architecture carlock_lpf_arch of carlock_lpf is
  component dffregister is
    generic (word_size: integer);
    port(clk, reset: in std_logic;
         d:in signed(word_size-1 downto 0);
         reset_word: in signed(word_size-1 downto 0);
         q:out signed(word_size-1 downto 0));
  end component;

  component fpresizer is
    generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
    port(a: in signed(a_adsize+a_bdsize -1 downto 0);
         b: out signed(b_adsize + b_bdsize-1 downto 0));
  end component;

  component fptruncate is
    generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
    port(a: in signed(a_adsize+a_bdsize -1 downto 0);
         b: out signed(b_adsize + b_bdsize-1 downto 0));
  end component;

  --TODO:Constants to adjust
  constant input_frac_length:integer := 10;
  constant output_frac_length: integer := 11;
  constant order:integer := 2;
  constant a_coeffs_length:integer := 16;
  constant a_coeffs_frac_length:integer := 14;
  constant b_coeffs_length:integer := 16;
  constant b_coeffs_frac_length:integer := 14;
  constant word_length:integer := 25;
  constant word_frac_length:integer := 6;
  constant b_sum_length:integer := filter_out'length;
  constant b_sum_frac_length:integer := 20;

  type word_array is array(order downto 0) of signed(
     word_length-1 downto 0);
  type a_coeff_array is array(order downto 1) of signed(
     a_coeffs_length-1 downto 0);
  type b_coeff_array is array(order downto 0) of signed(
     b_coeffs_length-1 downto 0);
  type a_word_mult_array is array(order downto 1) of signed(
     a_coeffs_length+word_length -1 downto 0);
  type b_word_mult_array is array(order downto 0) of signed(
     b_coeffs_length+word_length-1 downto 0);
  type b_word_sum_array is array(order downto 0) of signed(
     b_sum_length -1 downto 0);
  --TODO:Array constants
  constant a_coeffs: a_coeff_array := ("1100000011000111","
     0111111100111000"); --TODO:Make them negative
  constant b_coeffs: b_coeff_array := ("0100000000000000", "
     1000000001111011", "0100000000000000");


  --Signals
  signal words, a_word_mult_truncate, a_word_mult_truncated,
     a_word_mult_sum: word_array;
  signal a_word_mult: a_word_mult_array;
  signal b_word_mult: b_word_mult_array;
  signal b_word_sum,  b_word_mult_resized: b_word_sum_array;
  signal filter_in_resized: signed(word_length-1 downto 0);

  --TODO: Input modified signals, see todo at Input resize
  signal filter_in_expanded: signed((word_length-
     word_frac_length)+(input_frac_length)-1 downto 0);
begin
  --Input register (for proper reset operation)
    --Input resize
```

```vhdl
  --TODO: See range, and use fprezier and fptruncate accordingly
  IP_EXPAND_RESIZE: fpresizer
    generic map(a_adsize => filter_in'length - input_frac_length,
      a_bdsize => input_frac_length,
      b_adsize => word_length-word_frac_length,
      b_bdsize => input_frac_length)
    port map(filter_in, filter_in_expanded);

  IP_TRUNCATE_RESIZE: fptruncate
    generic map(a_adsize => word_length-word_frac_length,
      a_bdsize => input_frac_length,
      b_adsize => word_length-word_frac_length,
      b_bdsize => word_frac_length)
    port map(filter_in_expanded, filter_in_resized);
  --Words shifting, f/f initialization
  WORDS_GEN: for i in 1 to order generate

    W_INPUT: if i=1 generate
      W1: dffregister
        generic map(word_size => word_length)
        port map(clk, reset, a_word_mult_sum(0), (others =>'0'), words(1));
    end generate W_INPUT;

    W_ELSE: if i > 1 generate
      WX: dffregister
        generic map(word_size => word_length)
        port map(clk, reset, words(i-1), (others=> '0'), words(i));
    end generate W_ELSE;
  end generate WORDS_GEN;

  --Word multiplication
  A_COEFFS_MULT: for i in 1 to order generate
    ACMX: a_word_mult(i) <= a_coeffs(i)*words(i);
  end generate A_COEFFS_MULT;

  --Word addition into acc, with A_COEFFS_MULT truncation
  A_WORD_MULT_SUM_GEN: for i in 0 to order generate
    AWMSG_TRUNC_GEN:if i > 0 generate
      ACMTRUNCX: fptruncate
      generic map(
             a_adsize => a_coeffs_length+word_length-
```

```
                a_coeffs_frac_length-word_frac_length,
             a_bdsize => a_coeffs_frac_length+
                word_frac_length,
             b_adsize => word_length-word_frac_length,
             b_bdsize => word_frac_length)
      port map(a_word_mult(i), a_word_mult_truncated(i));
    end generate AWMSG_TRUNC_GEN;

    AWMSG_INPUT: if i=0 generate
      AWM0: a_word_mult_sum(0) <= a_word_mult_sum(1) +
         filter_in_resized;
    end generate AWMSG_INPUT;

    AWMSG_ORDER: if i=order generate
      AWM0: a_word_mult_sum(order) <= a_word_mult_truncated(
         order);
    end generate AWMSG_ORDER;


    AWMSG_ELSE: if i > 0 and i < order generate
      AWMX: a_word_mult_sum(i) <= a_word_mult_sum(i+1) +
         a_word_mult_truncated(i);
    end generate AWMSG_ELSE;

  end generate A_WORD_MULT_SUM_GEN;



  --RHS mult
  B_COEFFS_MULT: for i in 0 to order generate
    BCM_INPUT: if i=0 generate
      BCM0: b_word_mult(0) <= b_coeffs(0)*a_word_mult_sum(0);
    end generate BCM_INPUT;

    BCM_ELSE: if i > 0 generate
      BCMX: b_word_mult(i) <= b_coeffs(i)*words(i);
    end generate BCM_ELSE;
  end generate B_COEFFS_MULT;

  --RHS add
  B_WORD_MULT_SUM_GEN: for i in 0 to order generate
    BWMSRESIZERX: fpresizer
    generic map(
           a_adsize => b_coeffs_length+word_length-
              b_coeffs_frac_length-word_frac_length,
          a_bdsize => b_coeffs_frac_length +
              word_frac_length,
          b_adsize => b_sum_length-b_sum_frac_length,
          b_bdsize => b_sum_frac_length)
    port map(b_word_mult(i), b_word_mult_resized(i));


    BWMSO: if i=order generate
      BWMSO: b_word_sum(order) <= b_word_mult_resized(order);
    end generate BWMSO;

    BWMS_ELSE: if i < order generate
      BWMSX: b_word_sum(i) <= b_word_sum(i+1) +
          b_word_mult_resized(i);
    end generate BWMS_ELSE;

  end generate B_WORD_MULT_SUM_GEN;

  --output
  OUTPUT_GEN: filter_out <= b_word_sum(0);

end carlock_lpf_arch;
```

Listing D.11: pa.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pa is
  generic (fword_width, accsum_width: integer);
port(clk, reset: in std_logic;
  fword: in unsigned(fword_width-1 downto 0);
  accsum: out unsigned(accsum_width-1 downto 0));
end pa;

architecture pa_beh of pa is
begin
  process(reset,clk)
  variable count: unsigned(accsum_width-1 downto 0);
  begin
    if reset = '1' then
      --reset the variable
      count := (others=> '0');
    elsif rising_edge(clk) then
      --increase counter of accumalator
      count := count + fword;
      end if;
        accsum <= count;
  end process;
end architecture;
```

Listing D.12: `tbnco.vhd`

```
-- testbench.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tbnco is
end tbnco;


architecture test_nco of tbnco is
    component nco is
  port(clk, reset: in std_logic;
  fword: in unsigned(14 downto 0);
  op_sin: out signed(4 downto 0);
  op_cos: out signed(4 downto 0));
    end component;

    signal clk, reset: std_logic;
    signal fword:unsigned(14 downto 0);
    signal op_sin:signed(4 downto 0);
    signal op_cos: signed(4 downto 0);
    constant t: time := 3906200 ps; --time period of 64khz,
       to gen 64khz clock

begin


    R0: nco port map (clk, reset, fword, op_sin, op_cos);

    R1: process
        variable i, runs: integer;
        file data_out_sin: text open write_mode is "tbncosin.
           txt";
        file data_out_cos: text open write_mode is "tbncocos.
           txt";

        variable l:line;
        begin
            reset <= '1';
            wait for t;
            reset <= '0';

            for i in 0 to (2**fword'length)-1 loop
               fword <= to_unsigned(i, fword'length);
               for runs in 0 to 255 loop
                    clk <= '0';
                    wait for t/2;
                    clk <= '1';
                    wait for t/2;
                    write(l, to_integer(op_sin));
                    writeline(data_out_sin, l);
                    write(l, to_integer(op_cos));
                    writeline(data_out_cos, l);
                end loop;

            end loop;
            report "End␣simulation" severity failure;
        end process;

end test_nco;
```

Listing D.13: `nco.vhd`

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity nco is
port(clk, reset: in std_logic;
  fword: in unsigned(14 downto 0);
  op_sin: out signed(4 downto 0);
  op_cos: out signed(4 downto 0));
end entity;

architecture nco_arch of nco is

  component pa is
    generic (fword_width, accsum_width: integer);
  port(clk, reset: in std_logic;
    fword: in unsigned(fword_width-1 downto 0);
    accsum: out unsigned(accsum_width-1 downto 0));
  end component;

```

```vhdl

component pac is
  port(accsum: in unsigned(14 downto 0);
  op_sin: out signed(4 downto 0));
end component;

signal connector, phase_shifted_connector: unsigned(14 downto
     0);
constant fword_width:integer := fword'length;
constant accsum_width:integer := 15;
constant phase_shift: unsigned(accsum_width-1 downto 0) :=
    to_unsigned((2**(accsum_width-2))-1, accsum_width);
begin
  -- link signal to MSBs of variable
    R0: pa generic map(fword_width=>fword_width, accsum_width
        => accsum_width)port map(clk, reset, fword, connector)
        ;
    R1: pac port map(connector, op_sin);
    phase_shifted_connector <= connector + phase_shift;
  R2: pac port map(phase_shifted_connector, op_cos);
end nco_arch;
```

Listing D.14: `tblpf.vhd`

```vhdl
--tblpf.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tblpf is
end tblpf;

architecture tblpf_arch of tblpf is
  component lpf is
    port(clk, reset: in std_logic;
    x_in: in signed(16 downto 0);
      y_out: out signed(30 downto 0));
  end component;

  signal clk, reset: std_logic;
  signal x_in: signed(16 downto 0);
  signal y_out: signed(30 downto 0);
  constant t: time := 3906200 ps; --time period of 64khz, to
      gen 64khz clock

  --for all : nco_input_multiplier use entity work.
     nco_input_multiplier;

  begin

  R0:lpf port map(clk, reset, x_in, y_out);

  R1:process
    file x_data_in: text open read_mode is "x_data_in.txt";
    file y_data_out: text open write_mode is "y_data_out.txt"
       ;

    variable l:line;
    variable i:integer;
    variable x_in_bit_vector: bit_vector(x_in'range);

    begin
      reset <= '1';
      wait for t/2;
      reset <= '0';
      clk <= '0';
      wait for t/2;
      while not endfile(x_data_in) loop
        readline(x_data_in, l);
        read(l, x_in_bit_vector);
        x_in <= signed(to_stdlogicvector(x_in_bit_vector));
        clk <= '1';
        wait for t/2;
        clk <= '0';
        wait for t/2;
        write(l, to_bitvector(std_logic_vector(y_out)));
        writeline(y_data_out, l);
        wait for t;
      end loop;
      report "End␣simulation" severity failure;
  end process;
end tblpf_arch;
```

Listing D.15: `lpf.vhd`

```
-- lpf.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity lpf is
```

```
  port(clk, reset: in std_logic;
  x_in: in signed(16 downto 0);
    y_out: out signed(30 downto 0));
end lpf;

architecture lpf_arch of lpf is
  --Bit shift by 4 bits.
  constant taps: integer := 3;
  constant x_coeff_length : integer := 15;
  constant x_coeff_frac_length: integer := 14;
  constant y_coeff_length : integer := 16;
  constant y_coeff_frac_length: integer := 13;
  constant x_frac_length: integer := 15;
  constant y_frac_length: integer := 26;



  component dffregister is
    generic (word_size: integer);
    port(clk, reset: in std_logic;
    d:in signed(word_size-1 downto 0);
    reset_word: in signed(word_size-1 downto 0);
    q:out signed(word_size-1 downto 0));
  end component;

  component fpresizer is
    generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
    port(a: in signed(a_adsize+a_bdsize -1 downto 0);
        b: out signed(b_adsize + b_bdsize-1 downto 0));
  end component;


  type x_word_array is array(taps downto 0) of signed(x_in'
     range);
  type x_coeff_array is array(taps downto 0) of signed(
     x_coeff_length-1 downto 0);

  type y_word_array is array(taps-1 downto 0) of signed(y_out
     'range);
  type y_coeff_array is array(taps-1 downto 0) of signed(
     y_coeff_length-1 downto 0);

  type x_mult_array is array(taps downto 0) of signed(
     x_coeff_length+x_in'length -1 downto 0);
  type one_less_sum_array is array(taps-1 downto 0) of signed
```

```
     (y_coeff_length+y_out'length -1 downto 0);

  constant x_coeffs: x_coeff_array := ("0000000000101011","
     1111111111010110","1111111111010110","0000000000101011");
  constant y_coeffs: y_coeff_array := ("0001110001000001","
     1010011110111001","0101110000000101");

  signal x_words: x_word_array;
  signal y_words: y_word_array;
  signal x_mult_words, x_sum_words: x_mult_array;
  signal x_m0, x_m1, x_m2, x_m3,x_s0, x_s1, x_s2, x_sum:
     signed(x_coeff_length+x_in'length -1 downto 0);
  signal x_y_sum_arr:  signed(y_coeff_length+y_out'length -1
     downto 0);
  signal y_mult_words, y_sum_words: one_less_sum_array;
  signal sum_y, total_sum: signed(y_coeff_length+y_out'length
     -1 downto 0);
  signal y_sum_out: signed(y_out'range);


  begin
    X_GEN_DFF_SHIFT: for i in 0 to taps generate
      X_INPUT: if i = 0 generate
        T0:dffregister
          generic map(word_size => x_in'length)
          port map(clk, reset, x_in, (others => '0'), x_words
              (i));
        end generate X_INPUT;

      X_SHIFT_REGS: if i > 0 generate
      TX: dffregister
        generic map(word_size => x_in'length)
        port map(clk, reset, x_words(i-1),(others => '0'),
            x_words(i));
      end generate X_SHIFT_REGS;

    end generate X_GEN_DFF_SHIFT;

    Y_GEN_DFF_SHIFT: for i in 0 to taps-1 generate
      Y_INPUT: if i = 0 generate
        T0:dffregister
          generic map(word_size => y_out'length)
          port map(clk, reset, y_sum_out, (others => '0'),
              y_words(i));
        end generate Y_INPUT;

      Y_SHIFT_REGS: if i > 0 generate
      TX: dffregister
        generic map(word_size => y_out'length)
        port map(clk, reset, y_words(i-1),(others => '0'),
            y_words(i));
      end generate Y_SHIFT_REGS;

    end generate Y_GEN_DFF_SHIFT;

    X_GEN_MULT: for i in 0 to taps generate
      MX: x_mult_words(i) <= x_words(i) * x_coeffs(i);
    end generate X_GEN_MULT;

    Y_GEN_MULT: for i in 0 to taps -1 generate
      MY: y_mult_words(i) <= y_words(i) * y_coeffs(i);
    end generate Y_GEN_MULT;

    X_GEN_SUM: for i in 0 to taps generate
      X_SUM_0: if i = 0 generate
        S0: x_sum_words(0) <= x_mult_words(0);
      end generate X_SUM_0;

      X_SUM_ELSE: if i > 0 generate
        SX: x_sum_words(i) <= x_sum_words(i-1) + x_mult_words
            (i);
      end generate X_SUM_ELSE;
    end generate X_GEN_SUM;

    Y_GEN_SUM: for i in 0 to taps-1 generate
      Y_SUM_0: if i = 0 generate
        S0: y_sum_words(0) <= y_mult_words(0);
      end generate Y_SUM_0;

      Y_SUM_ELSE: if i > 0 generate
        SX: y_sum_words(i) <= y_sum_words(i-1) + y_mult_words
            (i);
      end generate Y_SUM_ELSE;
    end generate Y_GEN_SUM;

    X_Y_SUM: fpresizer
            generic map(a_adsize =>(x_in'length -
                x_frac_length)+(x_coeff_length -
                x_coeff_frac_length),
                          a_bdsize=> (x_frac_length+
                                x_coeff_frac_length),
                            b_adsize=> (y_out'length -
                                y_frac_length) + (y_coeff_length -
                                 y_coeff_frac_length),
                            b_bdsize=> y_frac_length +
                                y_coeff_frac_length)
                port map(x_sum_words(taps), x_y_sum_arr);
    X_SUM_GEN:x_m0  <= x_mult_words(0);
              x_m1  <= x_mult_words(1);
              x_m2  <= x_mult_words(2);
              x_m3  <= x_mult_words(3);
              x_s0  <= x_sum_words(0);
              x_s1  <= x_sum_words(1);
              x_s2  <= x_sum_words(2);
              x_sum  <= x_sum_words(taps);
    GEN_Y:sum_y  <= y_sum_words(taps-1);
          total_sum  <= sum_y + x_y_sum_arr;
          y_sum_out(y_sum_out'length-1) <= total_sum(
             total_sum'length-1);
          y_sum_out(y_sum_out'length -2 downto 0) <=
             total_sum(y_out'length + y_coeff_frac_length-2
             downto y_coeff_frac_length);
          y_out  <= y_sum_out;

end lpf_arch;
```

Listing D.16: tbcostas.vhd

```
-- tbcostas.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tbcostas is
end tbcostas;

architecture tbcostas_arch of tbcostas is
  component costasloop is
  port(carrier: in std_logic_vector(11 downto 0);
  clk, reset, f_desired: in std_logic;
  adc_mux: in std_logic_vector(1 downto 0);
  op, lock: out std_logic;
  adc_out: out std_logic_vector(11 downto 0));
  end component;

```

```
signal clk, reset: std_logic;
signal carrier: std_logic_vector(11 downto 0);
signal scarrier: signed(carrier'range);
signal op,lock, f_desired: std_logic;
signal adc_mux: std_logic_vector(1 downto 0);
signal adc_out: std_logic_vector(11 downto 0);

constant t: time := 3906200 ps; --time period of 64khz, to
    gen 64khz clock

--for all : nco_input_multiplier use entity work.
    nco_input_multiplier;

begin

R0:costasloop port map(carrier, clk, reset, f_desired,
    adc_mux, op, lock, adc_out);
tb:process
  file fdesired_in: text open read_mode is "f_desired.txt";
  file carrier_in: text open read_mode is "carrier_in.txt";
  file adc_mux_in: text open read_mode is "adc_mux_in.txt";
  file op_out: text open write_mode is "op_out.txt";
  file lock_out: text open write_mode is "lock_out.txt";

  variable l:line;
  variable i:integer;
  variable f_desired_bit: bit;

  begin
    readline(fdesired_in, l);
    read(l, f_desired_bit);
    f_desired <= to_stdulogic(f_desired_bit);
    readline(adc_mux_in, l);
    read(l, i);
    adc_mux<= std_logic_vector(to_signed(i,adc_mux'length))
        ;
    reset <= '0';
    clk <= '0';
    wait for t/2;
    reset <= '1';
    wait for t/2;
    reset <= '0';
    wait for t/2;
    while not endfile(carrier_in) loop
      readline(carrier_in, l);
```

```
        read(l, i);
        scarrier <= to_signed(i, scarrier'length);
        wait for t/2;
        clk <= '1';
        wait for t/2;
        clk <= '0';
        write(l, to_bit(op));
        writeline(op_out, l);
        write(l, to_bit(lock));
        writeline(lock_out, l);
      end loop;
      report "End␣simulation" severity failure;
  end process;

  carrier <= std_logic_vector(scarrier);
end tbcostas_arch;
```

Listing D.17: dffregister.vhd

```
-- dffregister.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity dffregister is
  generic (word_size: integer);
  port(clk, reset: in std_logic;
  d:in signed(word_size-1 downto 0);
  reset_word: in signed(word_size-1 downto 0);
  q:out signed(word_size-1 downto 0));
end dffregister;

architecture dffregister_arch of dffregister is

begin

  process(reset, clk)
  variable arr:signed(word_size -1 downto 0) := reset_word;

  begin
    if (reset = '1') then
      arr := reset_word;
  elsif rising_edge(clk) then
      arr := d;
    end if;
    q <= arr;
  end process;

end dffregister_arch;
```

Listing D.18: `tbdffregister.vhd`

```
-- tbdffregister.vhd

-- tbdffregister.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;

entity tbdffregister is
end tbdffregister;

architecture tbdffregister_arch of tbdffregister is
  component loopfilter is
    port(clk, reset: in std_logic;
        mult_error_op:in signed(38 downto 0);
        f_desired: in std_logic; --1 for 16, 0 for 8
        f_word_output: out unsigned(5 downto 0));
  end component;

  signal clk, reset: std_logic;
  signal mult_error_op: signed(38 downto 0);
  signal f_desired: std_logic;
  signal f_word_output: unsigned(5 downto 0);
  constant t: time := 3906200 ps; --time period of 64khz, to
     gen 64khz clock

  --for all : nco_input_multiplier use entity work.
     nco_input_multiplier;

  begin
  M0:f_desired <= '1';
  R0:loopfilter port map(clk, reset, mult_error_op,f_desired,
      f_word_output);

  R1:process
    file mult_error_op_in: text open read_mode is "
       mult_error_op_in.txt";
    file f_word_out: text open write_mode is "f_word_out.txt"
       ;

    variable l:line;
    variable i:integer;
    begin

      reset <= '1';
      wait for t/2;
      reset <= '0';
      clk <= '0';
      wait for t/2;
      while not endfile(mult_error_op_in) loop
        readline(mult_error_op_in, l);
        read(l, i);
        mult_error_op <= to_signed(i, mult_error_op'length);
        clk <= '1';
        wait for t/2;
        write(l, to_integer(f_word_output));
        writeline(f_word_out, l);
        clk <= '0';
        wait for t/2;
      end loop;
      report "End␣simulation" severity failure;
  end process;
end tbdffregister_arch;
```

Listing D.19: `fpresizer.vhd`

```
--fpresizer.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fpresizer is
  generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
  port(a: in signed(a_adsize+a_bdsize -1 downto 0);
      b: out signed(b_adsize + b_bdsize-1 downto 0));
end fpresizer;

architecture fpresizer_arch of fpresizer is
  signal minus_a: signed(a'range);
  signal a_std, minus_a_std: std_logic_vector(a'range);
  signal abs_b: std_logic_vector(b'range);
begin
  minus_a <= -a;
  a_std <= std_logic_vector(a);
  minus_a_std <= std_logic_vector(minus_a);
  abs_b(b_bdsize + a_adsize - 2 downto b_bdsize-a_bdsize) <=
     a_std(a_std'length-2 downto 0) when a(a'length -1) = '0'
      else
  minus_a_std(minus_a'length-2 downto 0) when a(a'length -1)
     = '1' else
  (others => '0');

  abs_b(abs_b'length-1 downto b_bdsize + a_adsize - 1) <= (
     others => '0');
  abs_b(b_bdsize -a_bdsize-1 downto 0) <= (others => '0');

    b <= signed(abs_b) when a(a'length -1) = '0' else
          -signed(abs_b) when a(a'length -1) = '1' else
            (others => 'U');
end fpresizer_arch;
```

Listing D.20: `fptruncate.vhd`

```
--fptruncate.vhd

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fptruncate is
  generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
  --a_adsize > b_adsize, a_bdsize > b_bdsize. BD: rhs decimal
  port(a: in signed(a_adsize+a_bdsize -1 downto 0);
      b: out signed(b_adsize + b_bdsize-1 downto 0));
end fptruncate;

architecture fptruncate_arch of fptruncate is
begin
  b(b'length-1) <= a(a'length-1);
   b(b'length-2 downto 0) <=  a(b_adsize+a_bdsize-2 downto
      a_bdsize-b_bdsize);
end fptruncate_arch;
```

Listing D.21: `costasloop.vhd`

```
-- costasloop.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


```

```vhdl
entity costasloop is
  port(carrier: in std_logic_vector(11 downto 0);
  clk, reset, f_desired: in std_logic;
  adc_mux: in std_logic_vector(1 downto 0);
  op, lock: out std_logic;
  adc_out: out std_logic_vector(11 downto 0));
end costasloop;



architecture costasloop_arch of costasloop is
  component nco is
    port(clk, reset: in std_logic;
         fword: in unsigned(14 downto 0);
         op_sin: out signed(4 downto 0);
         op_cos: out signed(4 downto 0));
  end component;

  component lpf is
    port(clk, reset: in std_logic;
         x_in: in signed(16 downto 0);
         y_out: out signed(30 downto 0));
  end component;

  component loopfilter is
    port(clk, reset: in std_logic;
         x_in:in signed(61 downto 0);
         f_desired: in std_logic;
         f_word_output: out unsigned(14 downto 0));
  end component;

  component dffregister is
    generic (word_size: integer);
    port(clk, reset: in std_logic;
         d:in signed(word_size-1 downto 0);
         reset_word: in signed(word_size-1 downto 0);
         q:out signed(word_size-1 downto 0));
  end component;

  component carlock is
    port(clk, reset: in std_logic;
    raw_op_sin,raw_op_cos:in signed(30 downto 0);
    lock: out std_logic);
  end component;

  component fpresizer is
    generic(a_adsize, a_bdsize, b_adsize, b_bdsize: integer);
    port(a: in signed(a_adsize+a_bdsize -1 downto 0);
         b: out signed(b_adsize + b_bdsize-1 downto 0));
  end component;

  signal nco_input: unsigned(14 downto 0);
  signal nco_sin, nco_cos: signed(4 downto 0);
  signal mult_sin, mult_cos: signed(16 downto 0);
  signal raw_op_sin, raw_op_cos: signed(30 downto 0);
  signal mult_error_op: signed(61 downto 0);
  signal op_signed_input, op_signed: signed(0 downto 0);

  --adc test stuff
  signal adc_signed, nco_sin_extended, nco_cos_extended:
     signed(adc_out'range);
  signal adc_adder_in, adc_adder_out: signed(adc_out'length
     downto 0);
  constant adc_offset: signed(adc_out'length downto 0) := (
     adc_signed'length-1 => '1', others => '0');
begin
  --NCO phase multiplier
  N: nco port map(clk, reset, nco_input, nco_sin, nco_cos);

  --Multiplier
  M0: mult_sin <= nco_sin * signed(carrier);
  M1: mult_cos <= nco_cos * signed(carrier);

  --FIR Filter
  L0: lpf port map(clk, reset, mult_sin, raw_op_sin);
  L1: lpf port map(clk, reset, mult_cos, raw_op_cos);

  --Extract output (Comparator)
  COMPARATOR: op <= op_signed(0); --Sign bit
  --Error Multiplier
  EM: mult_error_op <= raw_op_sin * raw_op_cos;
  --Loop Filter
  --NCO mapping to error
  LF: loopfilter port map(clk, reset, mult_error_op,
     f_desired, nco_input);

  FILTER_OP_INPUT: op_signed_input(0) <= (raw_op_sin(
     raw_op_sin'length -1));

  FILTER_OP: dffregister
  generic map(word_size => 1)
  port map(clk, reset, op_signed_input, (others => '0'),
     op_signed);

  --Lock Indicator
  LOCK_INDICATOR: carlock port map(clk, reset, raw_op_sin,
     raw_op_cos, lock);

  RESIZE_SIN: fpresizer
      generic map(a_adsize => 1, a_bdsize => 4, b_adsize =>
         1, b_bdsize => 11)
      port map(nco_sin, nco_sin_extended);

  RESIZE_COS: fpresizer
      generic map(a_adsize => 1, a_bdsize => 4, b_adsize =>
         1, b_bdsize => 11)
      port map(nco_cos, nco_cos_extended);

  ADC_MUX_GEN: with adc_mux select adc_signed <=
  nco_sin_extended when "00",
  nco_cos_extended when "01",
  mult_error_op(61 downto 50) when "10",
  raw_op_sin(30 downto 19) when "11",
  (others => 'U') when others;

  ADC_OFFSET_GEN: adc_adder_in <= (adc_signed(adc_signed'
     length-1) & adc_signed);
       adc_adder_out <= adc_adder_in + adc_offset;
    adc_out <=std_logic_vector(adc_adder_out(adc_signed'
       length -1 downto 0));


end costasloop_arch;
```