FPGA Implementation of an FFT-Based Carrier Frequency Estimation Algorithm

Bernaldo Luc

Submitted to the graduate degree program in Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of M.S. in Electrical Engineering.

Dr. Erik Perrins, Chairperson

Committee members

Dr. Rongqing Hui

Dr. Morteza Hashemi

Date defended: December 9, 2022

The Thesis Committee for Bernaldo Luc certifies
that this is the approved version of the following thesis :

FPGA Implementation of an FFT-Based Carrier Frequency Estimation Algorithm

Dr. Erik Perrins, Chairperson

Date approved: December 9, 2022

Abstract

Carrier synchronization is an essential part of digital communication systems. In essence, carrier synchronization is the process of estimating and correcting any carrier phase and frequency differences between the transmitted and received signals. Typically, carrier synchronization is achieved using a phase lock loop (PLL) system; however, this method is unreliable when experiencing frequency offsets larger than 30 kHz. This thesis evaluates the FPGA implementation of a combined FFT and PLL-based carrier phase synchronization system. The algorithm includes non-data-aided, FFT-based, frequency estimator used to initialize a data-aided, PLL-based phase estimator. The frequency estimator algorithm employs a resource-efficient strategy of averaging several small FFTs instead of using one large FFT, which results in a rough estimate of the frequency offset. Since it is initialized with a rough frequency estimate, this hybrid design allows the PLL to start in a state close to frequency lock and focus mainly on phase synchronization. The results show that the algorithm demonstrates comparable performance, based on performance metrics such as bit-error rate (BER) and estimator error variance, to alternative frequency estimation strategies and simulation models. Moreover, the FFT-initialized PLL approach improves the frequency acquisition range of the PLL while achieving similar BER performance as the PLL-only system.

Acknowledgements

I am grateful to Dr. Perrins for giving me this opportunity to work on this project. Your mentorship and guidance made it possible to complete this graduate journey. Special thanks to Dan and Ed for your support and patience. Your help and encouragement during the challenges were crucial in completing this thesis.

My graduate career has come to an end, and the best way to summarize is by using a common saying in a different context. It takes a village to finish graduate school. To my family and bonus families, thank you for your encouragement and support, which made all of this possible. I am thankful for my friends, my colleagues and my professors who impacted my graduate life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In digital communication systems, carrier synchronization involves estimating and correcting any phase and frequency differences observed between the transmitted signal and the received signal. Traditionally, carrier synchronization is tackled using a phase lock loop (PLL) to recover both carrier frequency and carrier phase information. However, this thesis investigates separating the carrier synchronization process into seperate frequency estimation and phase estimation procedures. Specifically, this thesis seeks to improve the carrier synchronization system of 4+12 amplitude and phase-shift keying (16-APSK) receiver implemented on a field-programmable gate array (FPGA) with the addition of a proposed carrier frequency estimation algorithm. Carrier frequency estimation is a crucial part of carrier synchronization that is concerned with correcting carrier frequency offsets.

1.1 Motivation and Objectives

The work in this thesis focuses on improving the phase synchronization process of a 16-APSK receiver by implementing a FFT-based carrier frequency estimation process. The FPGA-implemented, 16-APSK receiver demonstrates near-ideal BER performance for frequency offsets of less than 10 kHz. However, it becomes impossible for the PLL to lock in the presence of larger carrier frequency offsets.

This thesis proposes a non-data-aided, FFT-based frequency estimator that is used to initialize a PLL. The resulting hybrid carrier synchronization system is a more robust system that is able to compensate for both small and large frequency offsets. The FFT-based frequency estimator provides a rough estimate of carrier frequency offsets and the PLL, which can track instantaneous phase errors, corrects phase offsets. Since the receiver is FPGA-based, the suggested frequency estimator must also remain resource efficient.

In short, the main goal of this project is to implement a FFT-based frequency offset estimator, used in addition to the PLL, to accurately compensate for large frequency offsets. This two-pronged approach to carrier synchronization promises several advantages. When the receiver is unaffected by large frequncy offsets, the PLL can operate as usual. On the flip side, the frequency estimator, with its increased estimation range, can provide an initial estimate for the PLL. This new hybrid design is sensitive to instantaneous phase errors because of the PLL, but it requires a fairly accurate frequency offset estimate because inaccurate frequency estimates can lead to an accumulation of phase errors [1] . In the event of phase error accumulations, the carrier synchronization system includes a data-aided phase ambiguity resolution process that can correct it.

1.2 Thesis Structure

The thesis is made up of 7 chapters. Chapter 2, which covers the digital communications aspects, introduces the 16-APSK receiver and includes a brief description of the system model. It also briefly decsribes the PLL-based carrier phase synchronization system, the symbol timing synchronation system, and their relevant subsystems and functions. Chapter 3 presents a theoretical description of the carrier frequency offset estimation algorithm that is implemented in this project. Chapter 4 describes the hardware implementation of the proposed frequency estimation algorithm. This chapter also glosses over several relevant concepts such as FFTs and the CORDIC algorithm. Chapter 5 talks about the testing procedure and the simulation model. Chapter 6 discusses the results of the project. The performance of the frequency estimator design and the resulting “hybrid” carrier synchronization system are compared to simulation results. The previous PLL-based carrier synchronization system is also used as a performance benchmark. Finally, Chapter 7 presents the conclusion of this work, and proposes future research endeavors.

Chapter 2

Receiver System

The focus of this work is the implementation of an FFT-based frequency offset estimator that aids a digital communications receiver in achieving phase synchronization in the presence of large carrier frequency offsets. This sections introduces the 16-APSK receiver and briefly describes a few of its relevant components.

2.1 Constellation

The modulation scheme used by the pre-exising receiver is critical to understanding the frequency offset estimation algorithm presented in the following chapters. 16-APSK is a modulation scheme made up of 16 symbols each representing 4 bits. Its constellation, which is illustrated in Figure 2.1, is composed of 2 concentric rings: the inner ring consists of four points and the outer ring has 12 points. The ratio of the radii, r_2/r_1, is set to 2.75 for this thesis.

2.2 Carrier Phase Synchronization

Carrier phase offsets occur when there is a mismatch in phase between the oscillator at the transmitter and the receiver oscillator, which results in a offset in the received signal illustrated in Figure 2.2. In this noise-free example, the received symbols (the blue circles) are out of phase with the correct decisions (or the transmit symbols). Phase offsets can be even more problematic at low SNR, so a phase synchronization system is necessary to correct it. Although they work mostly independently in this project, the carrier frequency estimator can even be treated as part of the phase synchronization system. As such, it is important to introduce the overall system and a

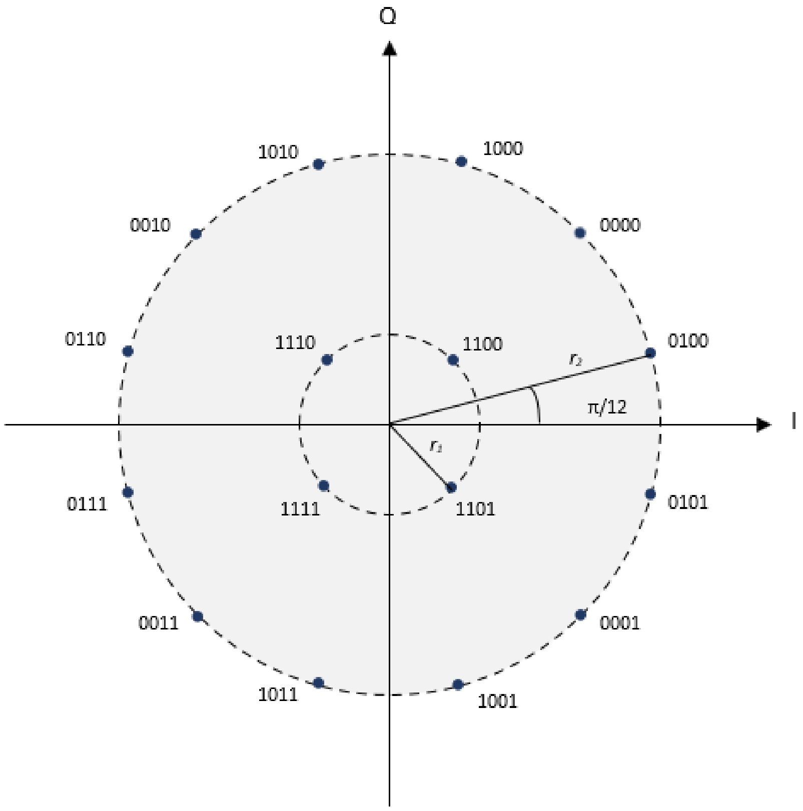

Figure 2.1: The 16-APSK constellation

few of its relevant modules.

Carrier phase synchronization is typically achieved using a phase locked loop (PLL), which is a feedback approach. The goal of the PLL is to compare and match the phase of the input signal to the phase of a reference signal. The PLL used in this work is composed of three parts: a phase error detector (PED), a loop filter, and a direct digital synthesizer (DDS).

The interpolated matched filter outputs $r'(kT_s) = x'(kT_s) + jy'(kT_s)$, which have also been rotated, serve as the inputs for the PED. Generally, using the decision $\hat{a}(k)$ that corresponds to $r'(kT_s)$,

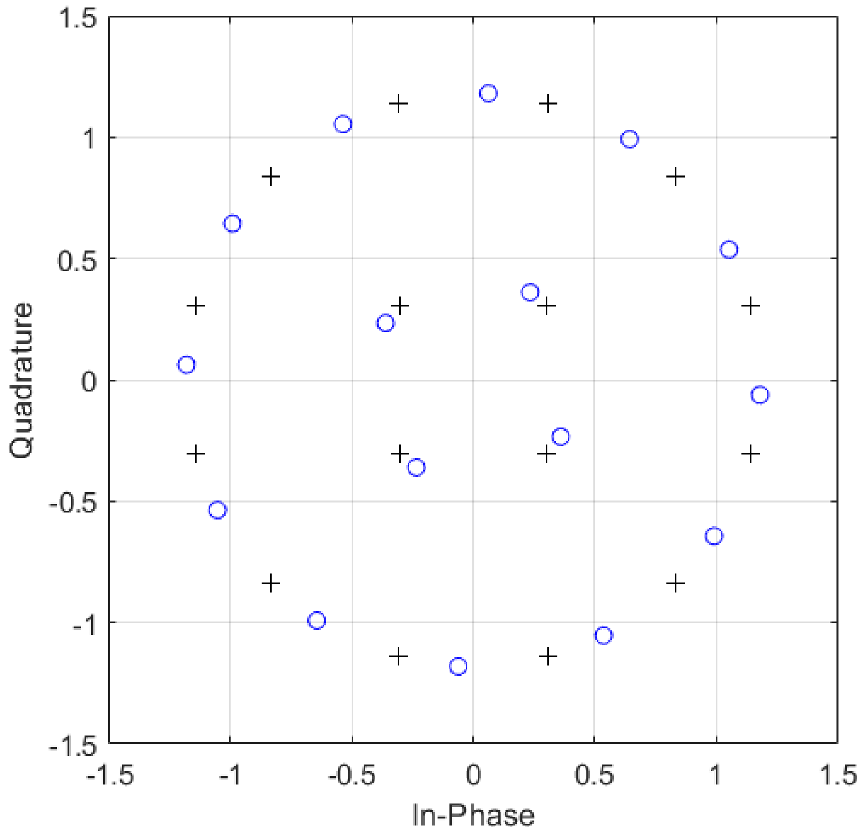

Figure 2.2: The received symbols with phase offset

the PED can be derived.

$$e(k) = \tan^{-1}\left(\frac{y'(kT_s)}{x'(kT_s)}\right) - \tan^{-1}\left(\frac{\hat{a}_Q(k)}{\hat{a}_I(k)}\right) \tag{2.1}$$

However, a less computationally costly PED, the maximum-likelihood PED (MLPED), is implemented to bypass the two divisions and inverse tangent operations. The MLPED, which seeks to minimize the error signal $e(k)$, uses the conditional probability to estimate phase given a noisy sample [3]. The resulting MLPED error signal is given as:

$$e(k) = y'(kT_s)\hat{a}_I(k) - x'(kT_s)\hat{a}_Q(k) \tag{2.2}$$

The PED output is then filtered by the loop filter. The loop filter used in this case, a proportionalplus-integrator (PPI) filter, makes the PLL the second-order PLL shown in Figure 2.3. The filter has a transfer function $H(z) = K_1 + K_2 \frac{1}{1-z^{-1}}$, were K_1 and K_2 are gain constants discussed in [3].

After filtering, the error signal is fed to the DDS. The output of the DDS is the phase estimate used to rotate the next sample

$$\hat{\theta}(k+1) = K_0 \sum_k v(k) \tag{2.3}$$

where K_0 is a gain constant and $v(k)$ is the loop filter output.

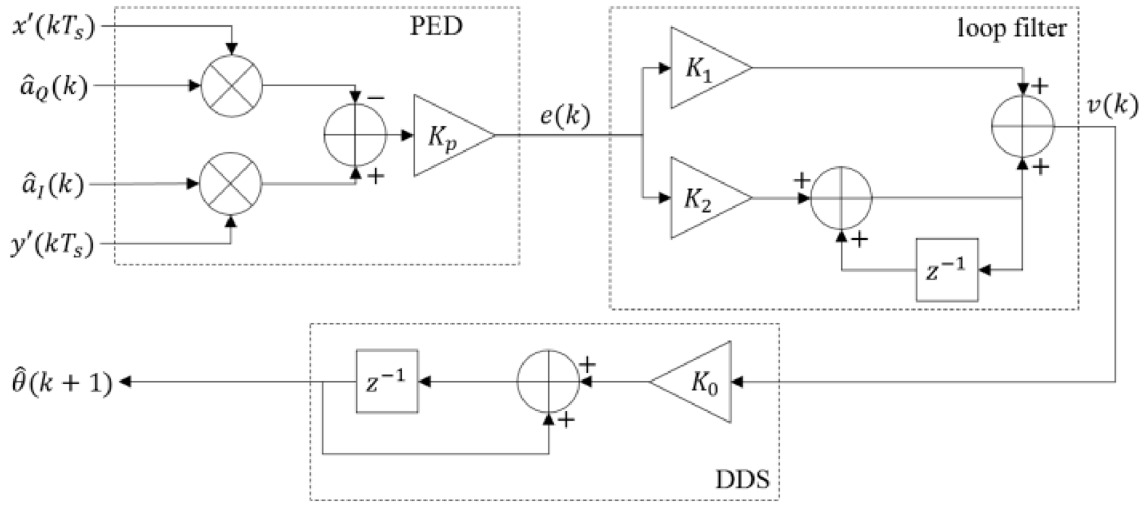

Figure 2.3: Second-Order PLL from [3]

The carrier phase synchronization system is incomplete without a proper frequency offset estimator as it struggles to reach synchronization for large frequency offsets. Likewise, the course frequency estimator presented in this work isn't precise enough, by design, to achieve phase synchronization independently. By combining the the methods into a hybrid design, the carrier synchronization system is able to benefit from both the frequency estimator's estimation range and the PLL's precision in compensating for phase errors.

2.3 Symbol Timing Synchronization

Symbol timing synchronization is another step that is crucial to successfully estimating carrier frequency offsets. Symbol timing errors occur when the analog-to-digital converter (ADC) does not sample the received signal at optimum instant. Even for a phase synchronized and noise-free signal, the received samples approach the decision boundaries instead of aligning with the constellation points.

The 2 main modules of the timing synchronization system are the interpolator and the timing error detector (TED). The TED employed in this case is a zero-crossing timing error detector (ZCTED). The ZCTED uses the interpolator outputs, sampled at two samples per symbol, for its input like the PED. Its output, which operates at one sample per symbol, is also dependent on the current and previous decisions $\hat{a}(k)$ and $\hat{a}(k-1)$

$$e(k) = x'((k-1/2)T_s+\tau)[\hat{a}_I(k-1)-\hat{a}_I(k)]+y'((k-1/2)T_s+\tau)[\hat{a}_Q(k-1)-\hat{a}_Q(k)] \tag{2.4}$$

The error signal $e(k)$ is then filtered and used to derive an interpolation control term $\mu(k)$ as described in [2]. This term $\mu(k)$, which is an estimate of the interval between the ideal sampling time and the actual sampled time, is used to control the interpolator.

The other key module, an interpolator, is responsible for estimating the value of the received signal if the samples were taken at the optimum sampling time. Because of its low computational complexity, the piecewise quadratic interpolator was chosen for this system [4]. The interpolator, which is implemented as a Farrow structure, can be written as

$$r((t(k)+\mu(k))T) = \sum_{p=0}^{2} \mu(k)^p \sum_{i=-2}^{1} b_p(i) r(t(k-1)T) \tag{2.5}$$

where T is the symbol time, $r(t(k))$ is the sample corresponding to the k^{th} symbol, and $t(k)$ is the time when the sample is taken.

2.4 System Model

The continuous-time APSK signal can be represented simply as

$$s(t) = \sum_n a(n)p(t - nT) \tag{2.6}$$

where *a(n)* is the n^{th} complex-valued 16-APSK symbol, and *p(t)* is a square-root raised- cosine (SRRC) pulse shaping filter chosen to satisfy the Nyquist no-ISI criterion. The pulse referred to in this work is derived from [7]. The received signal can be represented as

$$r(t) = s(t)e^{j(2\pi f_o nT+\theta)} + v(n), \quad n = 0,\cdots,N-1 \tag{2.7}$$

where f_o is the carrier frequency offset, θ is the phase offset, and *v(n)* is additive white Gaussian noise (AWGN). The matched filter is chosen to be *p(-t)* so it matches the transmit pulse shape and thus maximizes SNR. The sampled matched filter output is given by

$$z(nT) = a(n)e^{j(2\pi f_o nT+\theta)} + v(n) \tag{2.8}$$

Other relevant components of the PLL-based carrier phase and symbol timing synchronization subsystem are shown in the Figure 2.4.

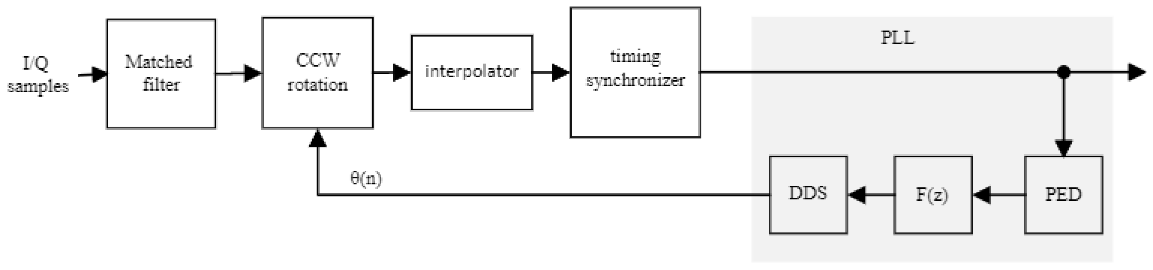

Figure 2.4: Summary of the PLL-based system

The matched filter output goes into the counterclockwise (CCW) rotation block, and is rotated to correct the carrier frequency offset and achieve phase synchronization. The CCW rotation process is expanded on in the next chapter. The rotated matched filter output is then interpolated, and undergoes timing synchronization. Finally, it reaches the PLL, which contains a phase error detector (PED) block, a loop filter that denoted as *F(z)*, and a direct digital synthesizer (DDS). The output of the DDS is a phase correction estimate which is then used in the CCW rotation block for phase correction.

Chapter 3

Carrier Frequency Offset Estimation

3.1 Presentation of Problem

In addition to carrier phase offsets, symbol timing errors, and noise; receivers often experience undesirable conditions in the form of carrier frequency offset. There are two main factors that can cause carrier frequency offset:

- Doppler effect caused by either the transmitter or the receiver moving.
- The local oscillator at the receiver operating at a different frequency than the local oscillator at the transmitter.

While the second case is more common, since the two local oscillators will never operate at the exact same frequency, the first case is also significant in aeronautical telemetry. The PLL-based method introduced in the previous chapter has been shown to be capable of correcting carrier frequency offsets of up to 30 kHz [1].

3.2 Frequency Offset Estimation

Two methods for estimating carrier frequency offsets, the PLL-based method and the FFT-based estimator, can be combined to form a new "hybrid", FFT+PLL-based design. This hybrid design, shown in Figure 3.2, uses the FFT-based estimator to initialize the PLL and achieve lock state faster.

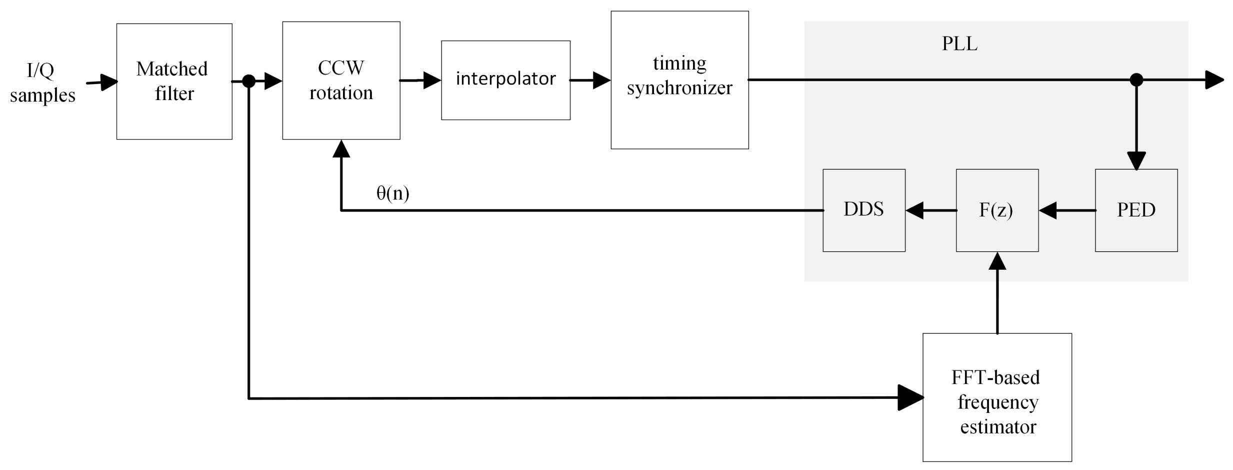

Figure 3.1: Block diagram summarizing the feed-forward approach of the FFT+PLL-based system

The hybrid design itself can be implemented in two manners. The first approach is the feed-forward design shown in Figure 3.1. In short, the frequency estimator uses the matched filter output samples (ideally after timing synchronization) to get a rough estimate of the carrier frequency offset. Since it is an "out of the loop" method that uses uncorrected received samples as input, the algorithm includes that only updates it's frequency offset estimate if it is significantly different from the previous estimate. Otherwise, the output of the frequency estimator is zero.

It should be noted that the PLL can track instantaneous phase errors; however, the system requires an accurate frequency offset estimate from the frequency estimator. The feed-forward algorithm is especially vulnerable to phase error accumulation caused by incorrect frequency estimates. Moreover, the matched filter outputs are rotated and interpolated before timing synchronization. Because of the resulting latency, the feed-forward algorithm is not effective since the matched filter samples used to estimate the frequency offset are unsynchronized. As mentioned in Section 2.3, symbol timing offsets are a problem must be tackled before achieving proper synchronization between the transmitter and receiver.

The second approach, presented in Figure 3.2, is the one implemented in this project. In this algorithm, the frequency offset estimator is fed the interpolated samples. The interpolation process, which is described in Section 2.3, is responsible for correcting any symbol timing offset. The

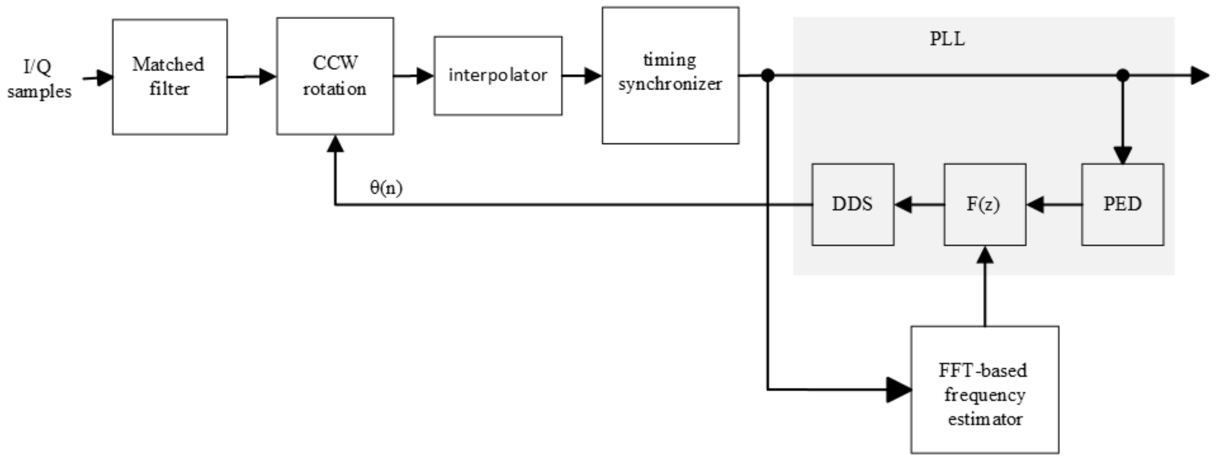

Figure 3.2: Block diagram of the feedback FFT+PLL-based system

frequency offset estimate is then used in the CCW rotation step to rotate the matched filter outputs. Since all of the operations occur in the loop, this method (referred to as the feedback design) uses phase corrected samples to derive its frequency offset estimate. This feedback design results in a more effective hybrid design because it also utilizes synchronized samples for its frequency estimate. By combining this feedback FFT-based algorithm with a PLL, the receiver uses a two-pronged plan for carrier phase synchronization. The frequency estimator algorithm provides the PLL with a rough frequency estimate, and PLL is responsible for correcting the phase offset and any residual frequency offset. In addition, the PLL handles the small frequency offsets (less than 1 kHz) and the FFT-based estimator handles large ones (greater than 1 kHz).

3.2.1 Coarse Frequency Estimation

To understand how to correct this carrier frequency offset, it is important to understand the frequency offset estimation process. The sampled matched filter outputs that are used for estimation after undergoing interpolation and timing synchronization can be represented mathematically by:

$$r'(nT) = s(nT)e^{j(2\pi f_o nT+\theta)} + v(nT), \quad n = 0, \cdots, N-1 \tag{3.1}$$

where *s(n)* represents the transmitted 16-APSK symbols illustrated in Figure 2.1, f_o is the carrier frequency offset, θ is the phase offset, and *v(n)* is AWGN. After timing synchronization, the interpolator output samples, which are sampled at one sample per symbol, undergo a non-linear transformation

$$g[r'(nT)] = |r'(nT)|^l e^{j12\arg r'(nT)} \tag{3.2}$$

where l is an integer. In order to account for the signal's amplitude modulation, $l = 2$ is used in this project. The purpose of the transformation is to ensure that a significant peak is present in the discrete-time Fourier transform (DTFT) of the transformed samples at the corresponding frequency offset. After the transformation, the DFT of the received signal becomes

$$R_k = \sum_{n=0}^{L_F-1} g[r'(nT)]e^{-j2\pi kn/N}, \quad k = 0, \cdots, L_F - 1 \tag{3.3}$$

where L_F is the length of the FFT used. The coarse estimate is found using the expression

$$\begin{aligned} \hat{k} &= \arg\max_k \{|R_k|^2\} \\ &= \arg\max_k \left| \sum_{n=0}^{L_F-1} g[r'(nT)]e^{-j2\pi kn/N} \right|^2 \end{aligned} \tag{3.4}$$

where $\hat{k}$ is the argument that corresponds to the maximum of the periodogram formed by $|R_k|^2$. The argmax, $\hat{k}$, can be converted to the corresponding positive and negative frequency indices

$$\hat{\omega} = \begin{cases} \hat{k} - L_F, & \text{if } \hat{k} \geq \frac{L_F}{2} \\ \hat{k}, & \text{otherwise.} \end{cases} \tag{3.5}$$

From the following formula, we can then calculate the frequency offset estimate

$$\hat{f}_o = \frac{\hat{\omega}}{12 \times L_F}. \tag{3.6}$$

This value is only a rough estimate of the frequency offset. However, since this estimate is used as a starting point for a PLL, it is sufficiently accurate. One point to be noted is that the inaccurate estimates can lead to phase error accumulation [1]. Nevertheless, the data-aided phase estimator is capable of correcting any residual phase offsets, in addition to smaller instantaneous phase errors, resulting in a more robust system.

The frequency estimation range of the proposed estimator is of utmost importance since this is one of the main objectives of this thesis. The estimation range is typically $|\Delta f| < f_s/2$, where f_s is the sampling frequency of the discrete-time system [9]. However, this formula can be modified to derive the estimation range of this proposed estimator as:

$$|\Delta f| < \frac{f_s}{2 \cdot 12} \tag{3.7}$$

3.2.2 FFT-based Algorithm

To accurately estimate large carrier frequency offsets, the FFT-based frequency estimator shown in Figure 3.3 is employed. First, the interpolated samples are converted from rectangular representation to polar representation to perform the necessary transformation. Next, the square of the magnitude of the complex samples is taken and the phase is multiplied by 12. The phase angles of the outer constellation points in Figure 2.1 are given by

$$\phi_{outer} = \frac{\pi}{12} n \tag{3.8}$$

where n represents odd integer values $1,\ 3,\ \cdots,\ 23$. For the inner constellation points, their phases are simply

$$\phi_{inner} = \frac{\pi}{4} n \tag{3.9}$$

where similarly n represents the odd integers 1, 3, 5, and 7. Thus, by multiplying the phase of the received samples by 12, the samples now all have them same phase

$$\begin{aligned}\phi_{outer} \cdot 12 &= \pi n = \pi \\ \phi_{inner} \cdot 12 &= 3\pi n = \pi\end{aligned} \tag{3.10}$$

since all possible n values are odd integers. In this simple case, there is no noise present let alone any frequency offsets. However, in the presence of a frequency offset, the new phase of the received samples can now be written as:

$$\phi_{new}(nT) = \pi + 2\pi f_o nT, \tag{3.11}$$

where f_o represents the frequency offset. After the transformation, the discrete Fourier transform (DFT) of the samples is taken using a resource-efficient FFT algorithm. To save resources, instead of taking a large FFT, an alternative approach is to take an equivalent number of small FFTs and accumulate the results to obtain an average of the DFT of the signal. This method can be shown to achieve similar performance based on the estimator error variance plots in Figure ???. Moreover, since the frequency offset estimate is used to initialize the PLL, it only needs a rough estimate to do so. For instance, a large FFT of length 8192 is replaced by 32 successive 256-sample FFTs. Since a hybrid PLL + FFT-based design is used, a trade-off between the preciseness of the frequency estimate and resource utilization is necessary. From a periodogram formed using the accumulated FFT results, the index of the FFT possessing the largest magnitude is used to find the frequency offset estimate. After finding the maximum index of the FFT, that value is then divided by 12 and the length of the FFT, L_F, to calculate the frequency offset estimate as given in Equation (3.6). This frequency offset estimate is then used to initialize the PLL.

3.3 Frequency Offset Correction

The effect of carrier frequency offset on the received signal can be seen in samples collected at the output of the matched filter. In Figure 3.4, the received samples appear to form a circle instead

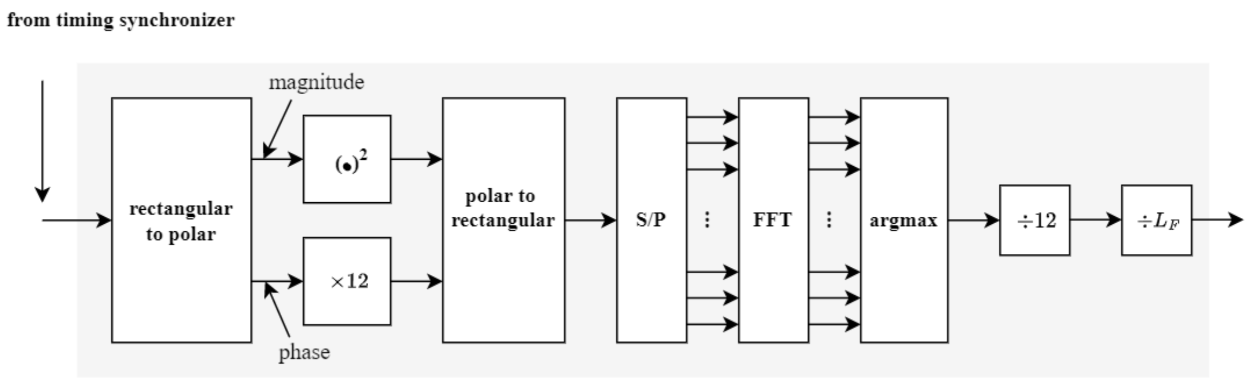

Figure 3.3: FFT-based frequency estimator

of grouping around the constellation points. As a result, the receiver is unable to make correct decisions even in the absence of channel noise.

The existing phase synchronization system is responsible for correcting the carrier frequency offset. In short, it is a derotation of the matched filter output samples since the frequency offset is simply a rotation. The undoing of that rotation is implemented using the counter-clockwise rotation (CCW) matrix given in (3.13). The estimate of the carrier phase offset, $\hat{\theta}$, provided by the DDS described in Section 2.2 is used to derotate the matched filter outputs $x(kT_s)$ and $y(kT_s)$. The phase offset correction operation performed using the CCW rotation matrix can be derived from (3.1) :

$$\begin{aligned} r_{corr}(n) &= r(n)e^{j\hat{\theta}(k)} \\ &= s(n)e^{j(2\pi f_o nT+\theta(n)+\hat{\theta}(n))}+v(n) \\ &= s(n)e^{j2\pi f_o nT}+v(n) \end{aligned} \tag{3.12}$$

$$\begin{bmatrix} x'(kT_s) \\ y'(kT_s) \end{bmatrix} = \begin{bmatrix} \cos(\hat{\theta}(k)) & -\sin(\hat{\theta}(k)) \\ \sin(\hat{\theta}(k)) & \cos(\hat{\theta}(k)) \end{bmatrix} \begin{bmatrix} x(kT_s) \\ y(kT_s) \end{bmatrix} \tag{3.13}$$

Once the coarse frequency offset estimate has been determined, it is added to the PLL where it is used to derotate the matched filter output samples accordingly. After correcting the carrier frequency offset, the derotated samples now group around the constellation points in clusters. This

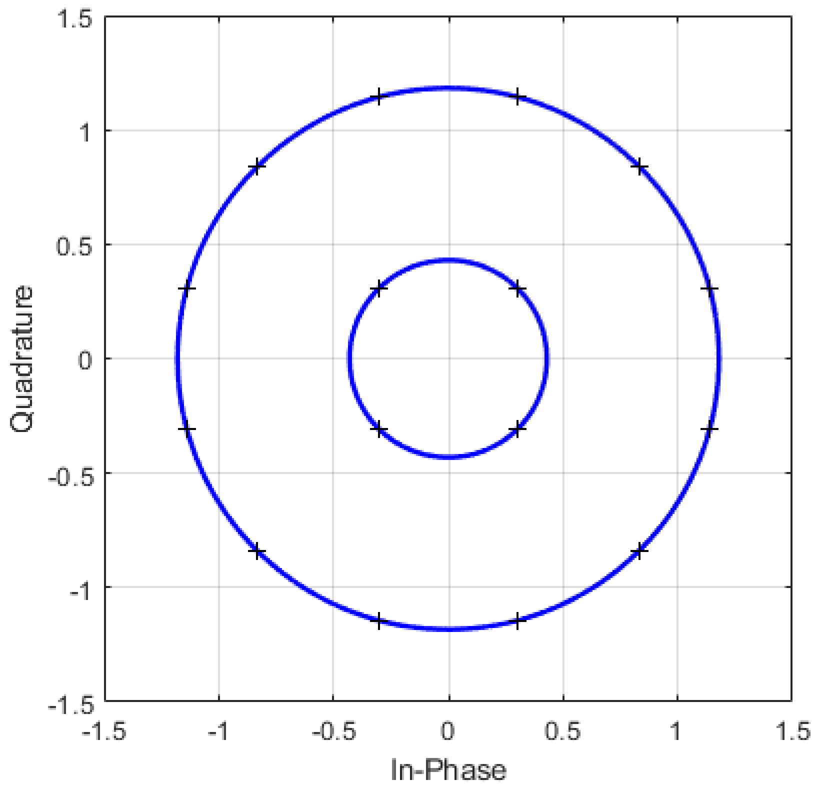

Figure 3.4: Received constellation after carrier frequency offset

behavior is expected and is the result of channel noise. Once other phase and timing synchronization issues have been addressed, the receiver is ready to make decisions on the received samples.

Chapter 4

Hardware Implementation

This chapter presents the components of the frequency offset estimator algorithm and details their FPGA implementation. The Cordic algorithm is elaborated upon and the FFT algorithm is introduced. The last section describes the FPGA resource utilization of the receiver before and after the addition of the frequency offset estimator.

4.1 Cordic Algorithm

The Coordinate Rotation Digital Computer (CORDIC) algorithm is an efficient method for calculating a wide range of functions, such as trigonometric functions, square roots, and exponentials. Furthermore, the CORDIC algorithm avoids performing multiplications and relies only on bit shifts and additions/subtractions, so it is a hardware-efficient algorithm commonly used on FPGAs.

There are two main factors behind the efficiency of the CORDIC algorithm. Firstly, the CORDIC algorithm takes an iterative approach to approximating the desired angle of rotation

$$\theta_d = \sum_{i=0}^{n-1} \theta_i \tag{4.1}$$

where θ_i represents small angles that are stored in a lookup table, and n is the number of iterations. Secondly, by storing angles θ_i in a lookup table such that

$$\tan(\theta_i) = \sigma_i 2^{-i}, \tag{4.2}$$

where $\sigma_i = \pm 1$. Consequently, multiplications involving $\tan(\theta_i)$ can be simply replaced by bit shift operations, which are both faster and simpler. However, there is a trade-off between accuracy and speed that relies heavily on the number of iterations, n, chosen.

The rotation matrix given in (3.13), can be simplified using the trigonometric identities $\sin(\theta) = \frac{\tan(\theta)}{\sqrt{1+\tan^2(\theta)}}$ and $\cos(\theta) = \frac{1}{\sqrt{1+\tan^2(\theta)}}$

$$\begin{bmatrix} x'_{i+1}(kT_s) \\ y'_{i+1}(kT_s) \end{bmatrix} = \frac{1}{\sqrt{1+\tan^2(\theta_i)}} \begin{bmatrix} 1 & -\tan(\theta_i) \\ \tan(\theta_i) & 1 \end{bmatrix} \begin{bmatrix} x_i(kT_s) \\ y_i(kT_s) \end{bmatrix} \tag{4.3}$$

for iterations $i = 0, 1, \cdots, n-1$. By using the approximation given in (4.2), (4.3) can be further simplified as

$$\begin{bmatrix} x'_{i+1}(kT_s) \\ y'_{i+1}(kT_s) \end{bmatrix} = \frac{1}{\sqrt{1+2^{-2i}}} \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i(kT_s) \\ y_i(kT_s) \end{bmatrix} \tag{4.4}$$

where $\sigma_i = \pm 1$ determines the direction of the rotation.

The CORDIC algorithm includes a scale factor correction term, $\frac{1}{\sqrt{1+2^{-2i}}}$, that is responsible for compensating for the scaling that inherently affects the output amplitude. However, the scale factor terms can be ignored until the end of the process to further optimize the algorithm. The scale factor compensation term, which can be applied as a single term, is expressed as

$$Z_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{4.5}$$

This value, which is dependent on the number on the number of iterations n, is typically calculated and stored as a constant can be axproximated as

$$Z = \lim_{n\to\infty} Z_i \approx 0.6072529.$$

The scale factor term can be ignored altogether to save resources and reduce latency. In this case, the output vector undergoes a gain that is the inverse of the scale factor term Z.

4.1.1 Polar-to-Rectangular Translation

The CORDIC algorithm has two main modes of operation which can be used to calculate several different functions, as described in [5]. However, this work focuses on using the CORDIC algorithm for just two operations: the vector rotation mode used for polar-to-rectangular translation and vector translation mode used in the rectangular-to-polar translation step.

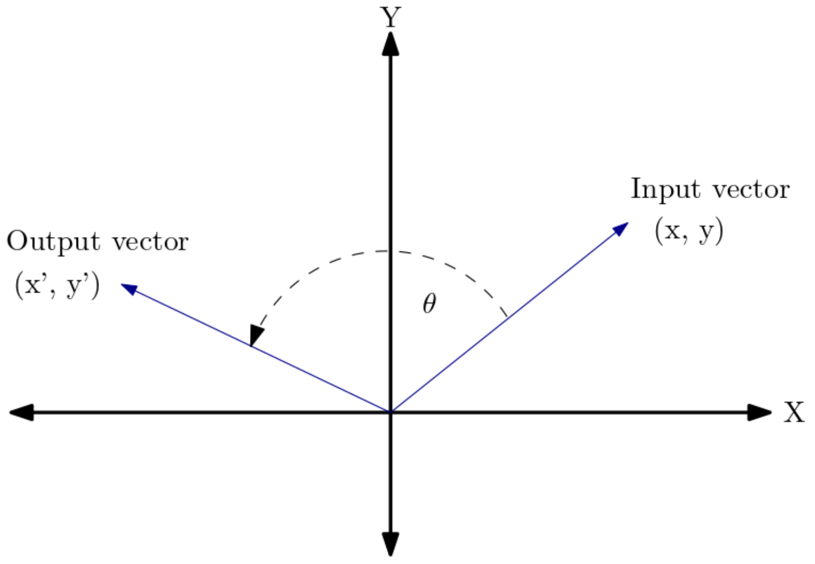

Figure 4.1: An illustration of vector rotation

As shown in Figure 4.1, vector rotation mode involves rotating a given vector by a desired angle to obtain a new vector. By using a lookup table to store known angles $\arctan(2^{-i})$, the CORDIC algorithm can perform a series of micro-rotations that grow smaller and smaller. In rotation mode, the goal of the algorithm is to minimize the error signal. As such, σ_i is selected so that the error signal derived from (4.1) converges to zero. By setting $\tilde{\theta_0} = \theta_d$ and $\sigma_i = sign(\tilde{\theta_n})$, the algorithm slowly drives $\tilde{\theta_n}$ to zero.

$$\begin{aligned} \tilde{\theta_n} &= \theta_d - \sum_{i=0}^{n-1} \theta_i \\ &= \theta_d - \sum_{i=0}^{n-1} \sigma_i \arctan(2^{-i}) \end{aligned} \tag{4.6}$$

The rotation mode of the CORDIC algorithm can be used to perform a CCW rotation or a polar-to-rectangular translation. For the CCW rotation case, an input vector (x, y) is rotated by a desired angle θ. Polar-to-rectangular translation can be implemented similarly by providing an input vector $(r, 0)$ (r here is the magnitude of the vector) and the desired rotation angle θ.

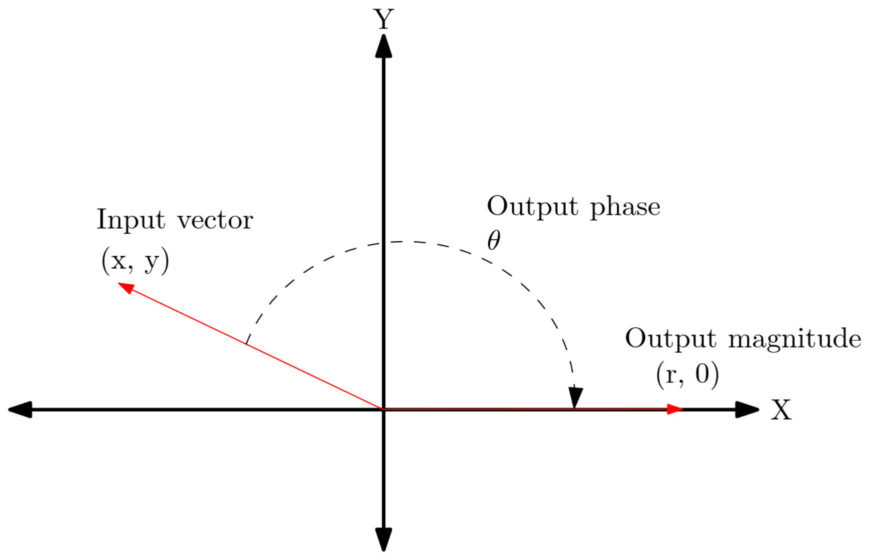

Figure 4.2: Rectangular to polar translation using vector rotation mode

4.1.2 Rectangular-to-Polar Translation

While the goal of the CORDIC algorthm in rotation mode is to drive $\tilde{\theta_n}$ to zero, vector translation mode needs a small change in the algorithm. In translation mode, the CORDIC algorithm rotates an input vector (x, y) until its y component becomes zero as illustrated in Figure 4.2. Simply put, when provided an initial vector (x, y) the output is (r, θ). The control term used to drive the y

coordinate to zero is now set to

$$\sigma_i = -\text{sign}(x_i) \cdot \text{sign}(y_i) \tag{4.7}$$

4.2 Phase Wrapper

The rectangular-to-polar CORDIC generates the magnitude and phase of the input vector or matched filter output samples. Before undergoing polar-to-rectangular translation, the phase is multiplied by 12. As discussed in Section 3.2.2, multiplying the phase by 12 gives all the samples equivalent phases on the unit circle. However, the CORDIC requires that input phase be constrained to the range $[-\pi, \pi]$ radians.

Although the new phases all map to the same angle, they still need to be wrapped to the interval $[-\pi, \pi]$. The function used to wrap the phase to the required interval can be summarized as

$$\phi_{wrapped} = \begin{cases} 2\pi(\phi_{frac} - 1), & \text{if } \phi_{frac} > 1/2 \\ 2\pi(\phi_{frac} + 1), & \text{if } \phi_{frac} < -1/2 \\ 2\pi\phi_{frac}, & \text{otherwise} \end{cases} \tag{4.8}$$

where

$$\phi_{frac} = \phi_{new} - \text{fix}(\phi_{new}) \tag{4.9}$$

$$\phi_{new} = \frac{\phi \times 12}{2\pi} \tag{4.10}$$

Dividing by 2π and removing its integer multiples leaves only the decimal portion ranging from $[-0.999, 0.999]$. This fractional component maps to $[-1.998\pi, 1.998\pi]$ when it is re-multiplied by a factor of 2π. Any resulting phase that corresponds to an angle larger than π is minused by 2π and any angle smaller than $-\pi$ is increased by 2π. The equivalent transformation described in (4.8) is used to save FPGA resources by doing a post-multiplication by a factor of 2π.

4.3 FFT

An FFT is an algorithm used to efficiently and quickly compute the DFT of a signal. Using equally-spaced samples of a continuous- time signal, the DFT can convert the signal to an approximation of its frequency domain representation. The DFT of a sequence of complex samples can be expressed as

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{kn}, \quad for\, k = 0, 1, \cdots, N-1 \tag{4.11}$$

where

$$W_N = e^{-\frac{i2\pi}{N}}.$$

However, computing the DFT is computationally slow and costly. For N data points, the DFT has a complexity of $O(N^2)$; meaning the computation requires $O(N^2)$ operations. An FFT can reduce that complexity to $O(N \log N)$ while also being much faster. From this it is evident why the FFT is the basis for the frequency offset estimation algorithm.

The hardware implementation of the FFT algorithm, as with the CORDIC algorithm, is fairly straightforward because of IP cores. IP cores are reusable modules used in FPGA designs to perform signal processing functions. Nevertheless, it is important to have a general understanding of FFTs. The algorithm used for calculating the DFT in this work is the Cooley-Tukey FFT algorithm. There are two types of Cooley-Tuker algorithms which are relevant to this work: the radix-2 algorithms and radix-4 algorithms. The radix-2 algorithms require the FFTS be of length N that is a power of 2. Similarly, the radix-4 algorithms is concerned with powers of 4. A third hybrid algorithm, the split-radix algorithm does stages of both radix-4 and radix-2 when N is a power of neither. It should be noted that, for the FPGA implementation of these algorithms, $N = 2^m$, for $m = 3, \cdots, 16$.

4.3.1 Radix-2 Decimation-In-Time

The radix-2 decimation-in-time (DIT) FFT algorithm based on the Cooley and Tukey approach involves recursively breaking up the DFT into smaller DFTs that can be recombined to give the total. Equation 4.11 can be rewritten in terms of its even ($n = 2r$) and odd ($n = 2r+1$) indices as

$$X_k = \sum_{r=0}^{\frac{N}{2}-1} x_{2r} W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x_{2r+1} W_N^{(2r+1)k}, \quad \text{for } k = 0, 1, \cdots, N-1 \tag{4.12}$$

Noting that $W_N^{2rk} = W_{\frac{N}{2}}^{rk}$, (4.12) can be further simplified

$$\begin{aligned} X_k &= \sum_{r=0}^{\frac{N}{2}-1} x_{2r} W_{\frac{N}{2}}^{rk} + \sum_{r=0}^{\frac{N}{2}-1} x_{2r+1} W_{\frac{N}{2}}^{rk} \cdot W_N^k \\ &= G_k + W_N^k \cdot H_k, \quad \text{for } k = 0, 1, \cdots, N-1 \end{aligned} \tag{4.13}$$

where

$$G_k = \sum_{r=0}^{\frac{N}{2}-1} x_{2r} W_{\frac{N}{2}}^{rk}$$

is a $N/2$-point DFT of an even-numbered indices, and

$$H_k = \sum_{r=0}^{\frac{N}{2}-1} x_{2r+1} W_{\frac{N}{2}}^{rk}$$

is its odd-indexed counterpart. Because of the periodic nature of complex exponential and consequently the DFT, X_k can be equivalently expressed as

$$X_k = \begin{cases} G_k + W_N^k \cdot H_k, & \text{for } k = 0, \cdots, \frac{N}{2}-1 \\ G_{k-\frac{N}{2}} - W_N^{k-\frac{N}{2}} \cdot H_{k-\frac{N}{2}}, & \text{for } k = \frac{N}{2}, \cdots, N-1 \end{cases} \tag{4.14}$$

The first processing stage of the DIT FFT algorithm is shown by Figure 4.3. To further optimize the algorithm, the radix-2 decimation-in-time process is then applied recursively to the two half-length DFTs until they become length-2 DFTs. For instance, the two length-$\frac{N}{2}$ DFTs is decomposed into four length-$\frac{N}{4}$ DFTs; and so forth.

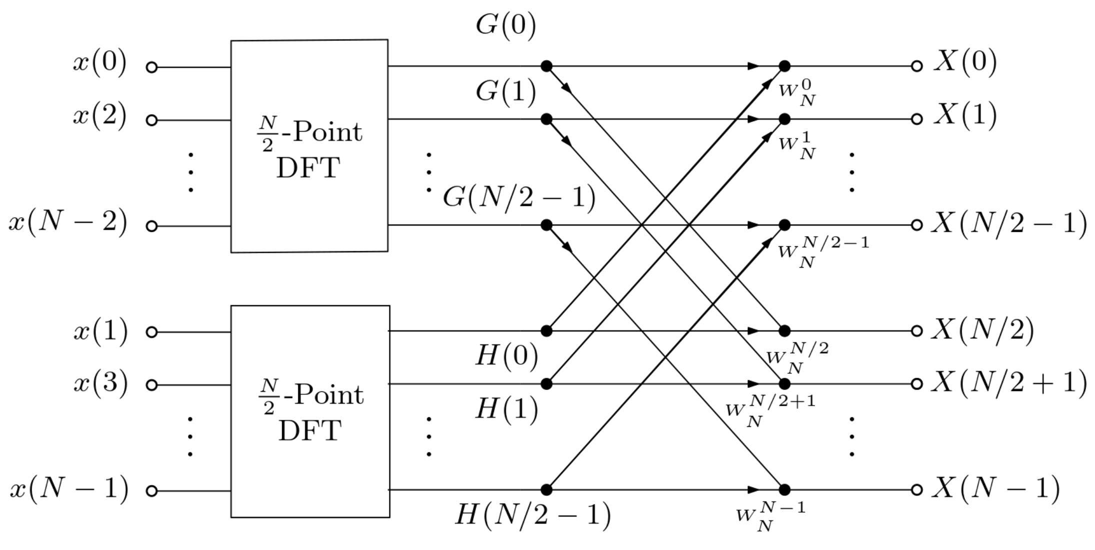

Figure 4.3: Combining stage of length-N DFT using a decimation-in-time radix-2 FFT

Overall, the algorithm consists of $\log_2 N$ stages with each stage made up $\frac{N}{2}$ butterflies. The DIT butterfly operation, illustrated in Figure 4.4, is the representation of the basic computational element. Each butterfly requires a complex multiply and two complex additions. The second butterfly shown in Figure 4.4 is a simplified butterfly that reduces the number of complex multiplications from two to one.

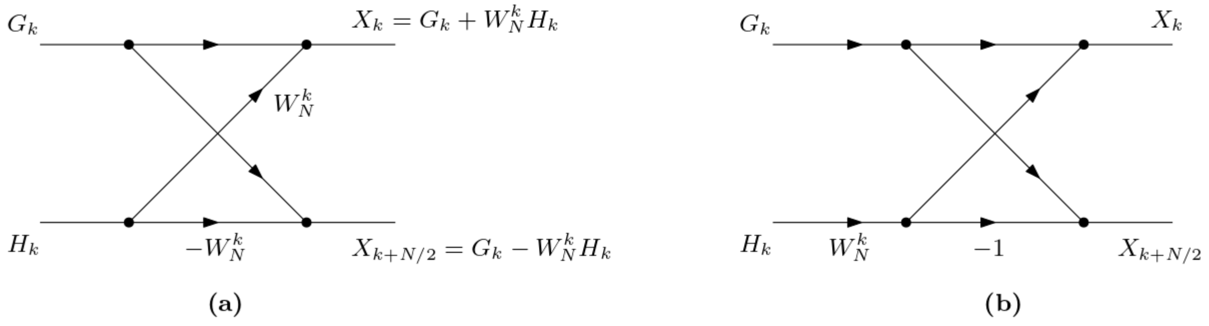

Figure 4.4: The radix-2 decimation-in-time FFT butterfly: (b) is a simplification of (a)

4.3.2 Radix-2 Decimation-In-Frequency

The radix-2 decimation-in-frequency (DIF) FFT can be defined by rewriting equation (4.11) as

$$\begin{aligned} X_k &= \sum_{n=0}^{\frac{N}{2}-1} x_n W_N^{kn} + \sum_{n=\frac{N}{2}}^{N-1} x_n W_N^{kn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x_n W_N^{kn} + \sum_{n=0}^{\frac{N}{2}-1} x_{n+\frac{N}{2}} W_N^{k(n+\frac{N}{2})} \\ &= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{n+\frac{N}{2}} W_N^{k\frac{N}{2}}) W_N^{kn}, \quad \text{for } k = 0, 1, \cdots, N-1 \end{aligned} \tag{4.15}$$

This equation can be further rearranged by computing the even and odd-numbered indices separately. After grouping the even-numbered indices by setting $k = 2r$, using the identities $W_N^N = 1$, $W_N^{\frac{N}{2}} = -1$, and $W_N^2 = W_{N/2}$ yields

$$\begin{aligned} X_{2r} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{n+\frac{N}{2}} W_N^{rN}) W_N^{2rn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{n+\frac{N}{2}}) W_{\frac{N}{2}}^{rn}, \quad \text{for } r = 0, 1, \cdots, \frac{N}{2}-1 \end{aligned} \tag{4.16}$$

Similarly, the odd-numbered indices yields

$$\begin{aligned} X_{2r+1} &= \sum_{n=0}^{\frac{N}{2}-1} (x_n + x_{n+\frac{N}{2}} W_N^{(2r+1)\frac{N}{2}}) W_N^{(2r+1)n} \\ &= \sum_{n=0}^{\frac{N}{2}-1} ((x_n - x_{n+\frac{N}{2}}) W_N^{n}) W_{\frac{N}{2}}^{rn}, \quad \text{for } r = 0, 1, \cdots, \frac{N}{2}-1 \end{aligned} \tag{4.17}$$

Defining $g_n = x_n + x_{n+\frac{N}{2}}$ and $h_n = x_n - x_{n+\frac{N}{2}}$, we can solve

$$X_k = \begin{cases} \sum_{n=0}^{\frac{N}{2}-1} g_n W_{\frac{N}{2}}^{rn}, & \text{for } k \text{ even} \\ \sum_{n=0}^{\frac{N}{2}-1} h_n W_N^{n} \cdot W_{\frac{N}{2}}^{rn}, & \text{for } k \text{ odd} \end{cases} \tag{4.18}$$

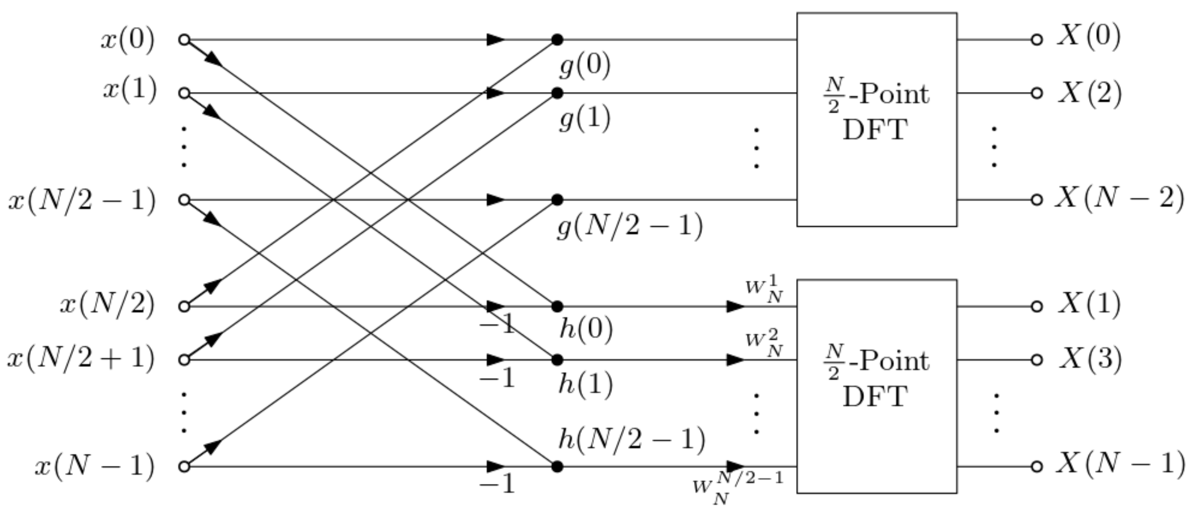

Figure 4.5: Combining stage of N-length DFT using a decimation-in-frequency radix-2 FFT

Similarly to the DIT case, the radix-2 DIF involves dividing a length-N DFT into two length-$\frac{N}{2}$ DFTs. This process is applied recursively to the two length-$\frac{N}{2}$ DFTs. The resulting DFTs become smaller and smaller until the final stage is a combination of $\frac{N}{2}$ length-2 DFTs as illustrated in Figure 4.6. It should be noted that the input of the radix-2 DIF FFT is in the linear order, whereas its output is arranged in bit-reversed order. In linear/ natural order, the indices are sorted in numerical order. However, in bit-reversed order, the indices are transformed into their binary representation; and then that binary number is reversed before being translated back into decimal form. Since the output or frequency indices are bit-reversed, the algorithm is named *decimation-in-frequency*. In the DIT flow diagram shown in Figure 4.3, the opposite behavior is observed. The input or time indices are shuffled hence the name *decimation-in-time*.

4.3.3 Configurations

The FFT IP core has four available architecture options with different trade-offs: Pipelined Streaming I/O, Radix-4 Burst I/O, Radix-2 Burst I/O, and Radix-2 Lite Burst I/O. Pipelined Streaming I/O is configured to continually stream data unlike the Radix-4 Burst I/O, which loads the data and has a wait-state before accepting the next frame. Radix-2 Burst I/O and Radix-2 Lite

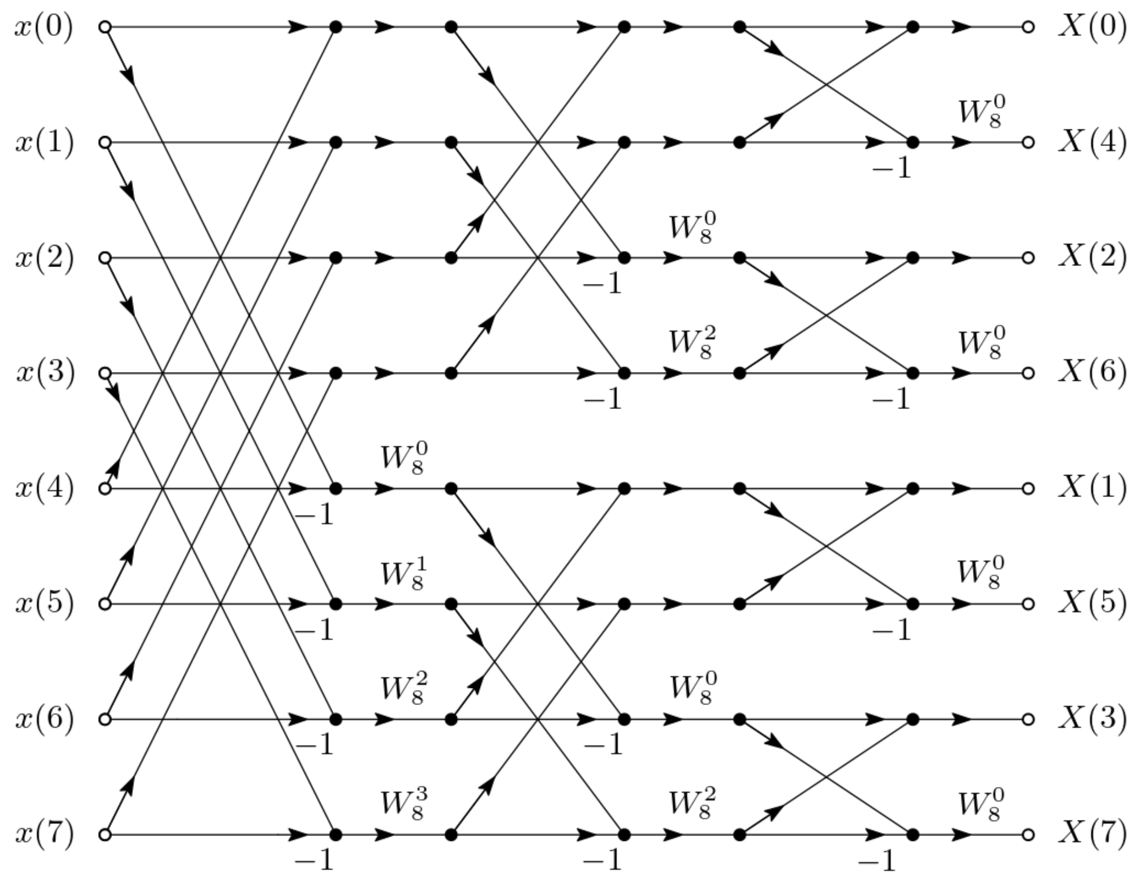

Figure 4.6: Example of an 8-point radix-2 decimation-in-time FFT

Burst I/O are very similar to the Radix-4 architecture, but they use smaller butterflies at the cost of longer transform time. So while Streaming architecture has the highest throughput, its trade-off is FPGA resources. However, because the other architectures would require incoming data to be stored in memory during the processing stage, the pipelined architecture was selected for this design. The burst I/O architectures all use the decimation-in-time (DIT) method; however, pipelined streaming I/O architecture uses the decimation-in-frequency (DIF) method.

4.3.4 Latency & Resource Utilization

The latency of the FFT core is one of its most defining features. Using pipelined streaming architectures typically help with power consumption and also provide lower latency. After the frame of input data (256 samples) is fully loaded, the core starts to process the data with a latency of 868 cycles. For perspective, the slower of the two CORDIC cores has a latency of 27 clock cycles. The core continues to load and buffer the next frames while it is still processing the previous one. Originally the received signal contained 32 samples per symbol (one sample every clock cycle), by the time it reaches the FFT it is down to one sample per symbol. However, at this point the data has been downsampled to one sample roughly every 32 clock cycles. Because of this slow input data rate, the overlap between consecutive frames isn't significant; however, it minimizes the potential of frames being corrupted.

The FFT core takes up nearly half the resource utilization of the design. The selected Pipelined Streaming architecture establishes a necessary trade-off between resources and throughput. There are several important characteristics of the selected FFT algorithm that affect the resource utilization. Each architecture can be configured for how the output data is ordered. The output of the FFTs can be ordered in the two ways discussed in Section 4.3.2 : natural/linear order or bit/digit reversed order. Bit reversed order would have required the use of a lookup table to keep track of the FFT index. For simplicity, natural order was chosen since the FFT index could be tracked with a counter. However, natural order uses more memory than bit reversed order. The core was also configured to use fixed-point data format (floating point was the other option) with scaled arithmetic. A scaling schedule that helps avoid potential overflow is necessary in scaled arithmetic mode. In contrast, scaled fixed point arithmetic uses significantly less resources than floating point arithmetic.

4.4 argMax Operation

Once the FFT starts generating outputs, the magnitude square of each complex output is taken. The resulting periodogram of the signal is used to find the dominant frequencies present in the signal. Any frequency offset or dominant frequency present in the signal corresponds to the argument of the maximum (argmax) of the periodogram. Finding the maximum, would be a simple task if it were of just one frame. Nevertheless, to find the maximum of 32 successive FFTs and the corresponding FFT index a block memory (BRAM) is used. Since the output of the FFT block are linearly order, the 256 FFT outputs are added to respective results from the previous frames, continuously overwriting the values stored in the BRAM. The 8-bit address of the BRAM is controlled by a counter that is also used to find the argmax. A lookup table of the 128 possible frequency offset estimates is used to avoid doing the division in (3.6). The lookup table only needs to include 128 values because the sign of the estimate can be stored using a variable. After 32 consecutive FFTs, the argmax is used to find the correct frequency offset estimate in the lookup table.

4.5 Clocking

Every single component of an HDL design uses a clock signal. In an FPGA, a clock is a signal that is periodically driven from high voltage to low voltage and vice versa. Clock signals control how long it takes for flip-flops and other FPGA devices to run. Typically, the rising or falling edge of the clock signal is used to sample data signals. As a result, a relatively low clock rate (several hundreds of MHz) is normally used to improve the timing performance of the HDL design. Low clock rates correspond to longer clock periods, which allows the data signals to stabilize within the clock period.

Nevertheless, it is sometimes necessary to use a faster clock to compensate for the latency of a module. For instance, the primary clock of the overall receiver design is a 125 MHz clock. However, several of its components, including the symbol timing synchronization system, are driven by a 400 MHz clock. It can be helpful to consider the timing of the design in terms of clock cycles. The received signal is upsampled at a rate of $N = 32$ samples/symbol. The system has a decimation factor of $\frac{N}{2} = 16$, so the interpolator input is sampled at 2 sample per symbol. On the FPGA, the received signal is sampled at one sample per clock cycle. Consequently, the interpolator input is sampled at 16 clock cycles per sample. 16 clock cycles of the 125 MHz clock is equivalent to 128 ns, however that isn't enough time for the interpolation control and phase correction terms to be computed. Switching to the 400 MHz clock now provides the affected modules approximately 51 clock cycles to work with in 128 ns. Since the frequency estimator output is used to initialize the PLL, it has less strict timing requirements even though it introduces the largest latency of any system in the receiver. Therefore, the frequency estimator design uses the slower 125 MHz clock to drive its operations

However, using an HDL design with multiple clocks isn't without it's own set of issues. Crossing clock domains occur when a signal or flip-flop driven by one clock interacts with a signal driven by a different clock. Flip-flops are the most commonly used memory elements in HDL design. Flip-flops have setup and hold times that should be met when properly crossing between clock domains. Setup time is a period for which the input signal to a flip-flop must remain stable before the active edge (i.e. rising or falling edge) of the clock. Similarly, hold time is the required amount of time that the flip-flop's input must be stable for after the triggering edge of the clock.

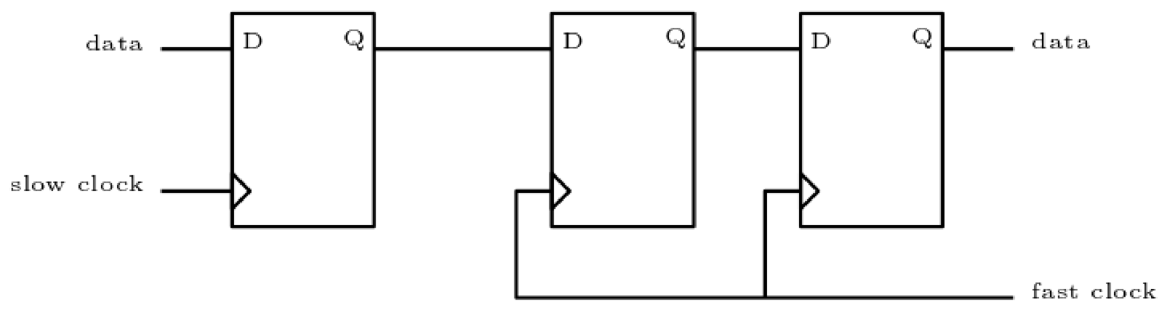

Figure 4.7: Double flip-flop technique for synchronizing data between clock domains

Two different cases of signals crossing between clock domains must be considered: signals crossing from a slow to fast clock domain and signals crossing from a fast to slow clock domain. The simpler of the two cases is when a signal goes from a slow clock domain to a faster one. The issue that arises is that the slow clock signal could still be in an unstable state when it is sampled at the edge of the fast clock. As a result, the signal in the fast clock domain has a metastable output. The strategy to solve this problem is to use a second flip-flop to re-register the signal in the first flip-flop. This approach gives the new signal sufficient time to stabilize. Although the output of the second flip-fop shown in Figure 4.7 is now stable, the ensuing signal is delayed.

The more complicated case is the opposite of the previous situation. This time the signal goes from a fast clock domain to a slower one. It is especially troublesome when the signal needing to cross between domain clocks is a pulse or flag that occurs for one clock cycle. The double flip-flop strategy doesn't work because the edge of the slow clock might completely miss the pulse depending the clock speeds. Instead, one solution is to stretch the pulse in the fast domain for at least two full clock cycles of the slow clock. Stretching the signal for a minimum of two clock cycles helps avoid completely missing the pulse by violating the setup time one the first rising clock edge and the hold time on the second clock edge, assuming the triggering edge is the rising edge. The ensuing slow clock domain signal will last at least two clock cycles, so an edge detector mechanism can be used to convert it into a pulse lasting one clock cycle if necessary. When the signal crossing between clock domains isn't a pulse, a multiple flip-flop technique similar to the one presented in Figure 4.7 can be utilized.

4.6 FPGA Resource Utilization

The resource utilization of the FPGA used is extremely important because it can negatively affect its performance. As the FPGA resources get pushed to the limits, the synthesis tools may struggle to meet timing requirements. Furthermore, even if the it gets synthesized it may still prove difficult to route the design. The post-implementation, resource utilizations of the previous design and this project are offered in Table 4.1. The overview of the utilization presented here is in terms of look-up tables (LUT), LUTs used as distributed RAM (LUTRAM), flip-flops (FF), block memory or block random access memory (BRAM) and DSP slices (used by the FPGA to accelerate signal processing functions). Of the values shown in Table 4.1, the FFT core is responsible close to 50% of the increase in resource utilization. The two CORDIC modules are the biggest contributors for the remaining half. Nevertheless, the addition of the frequency estimator is only a small increase in the resource utilization in comparison to the previous receiver design.

Table 4.1: FPGA Resource Utilization

Resource	Previous Utilization (%)	Current Utilization (%)
LUT	12.12	16.36
LUTRAM	1.46	2.72
FF	8.07	10.83
BRAM	6.16	7.12
DSP	37.57	41.89

Chapter 5

Testing Procedure

The purpose of this chapter is to describe the testing environment used to generate the results presented in this work. MATLAB and Vivado are the main software used for this thesis. MATLAB was used to both create a model of the digital communications system to be used as a benchmark for the FPGA implementation and generate test signals for it. Using Vivado to write VHDL (and some Verilog) code, the frequency offset estimator was designed based on the MATLAB model. Vivado's behavioral simulation was used to verify that the implemented receiver operated. It was also used to test the frequency offset estimator design separately before its addition to the receiver system. Vivado simulation was relied upon heavily during the design process even though it took several hours to obtain any results.

After verifying that the FPGA design operated like its MATLAB counterpart, the FPGA was ready to undergo tests. First, the test signals generated using MATLAB were used. As things progressed, a signal generator was used for bit-error rate (BER) performance tests.

5.1 Simulation

MATLAB is used to both simulate a theoretical model of the complete 16-APSK receiver's behavior and generate data to test the functionality of the frequency offset estimator. Matched filter outputs are collected from the MATLAB receiver model and used to evaluate the frequency estimator on its own. After comparing the FFT outputs and the subsequent frequency offset estimate of the FPGA implementation with the MATLAB model, the frequency offset estimator is added to the receiver for further testing.

In addition to being used to generate test data for the different subsystems of the FPGA receiver, MATLAB simulation is also used as a performance benchmark. The performance of the FPGA implementation is measured in terms of the demodulated signal's BER and error vector magnitude (EVM). EVM is a commonly used performance metric in digital communications that measures how closely the demodulated signal matches the ideal constellation points. The EVM, which is typically expressed in terms of percentage, is defined as

$$\mathrm{EVM}_{\%} = \frac{\sqrt{\frac{1}{N}\sum_{n=0}^{N-1}|r(kT_s)-\hat{a}(k)|^2}}{\max_{a\in\mathbb{A}}|a|} \times 100\%, \tag{5.1}$$

where N is the total number of received symbols. All the received symbols contribute in the calculation the EVM. The average of the magnitudes of the error vectors is normalized by the maximum amplitude of the constellation before converting it to a percentage.

5.2 Test Data

5.2.1 Payload

The are two important things to consider concerning the test data. Firstly, the project uses low density parity check (LDPC) codes to improve BER performance. The LDPC system has two important characteristics described in [3]: its code rate and the length of its codeword. In this project, the code rate — the ratio of information bits total bits — is 4/5. The length of the codeword or block length is 4096. Secondly, an attached synchronization marker (ASM) is placed at the beginning of each LDPC frame to help achieve phase ambiguity resolution. Phase ambiguity occurs when the initial phase offset is large enough for the received symbols to fall into the wrong decisions. The ASM, which is 256 bits long, is known by the receiver. As such, it is also used by the receiver to recognize the beginning of a new frame. After LDPC encoding, the frame length is $\frac{4096}{4/5} = 5120$ bits. Since the ASM bits are not LDPC encoded, the final ratio of information bits to total bits $\frac{4096}{5120+256} = 16/21$.

5.2.2 Data Rate

The received C-band RF signal is downconverted to the typical 70 MHz IF signal. The ADC is then responsible for sampling that 70 MHz IF signal at $93\frac{1}{3}$ MHz. This sampling rate and the resulting information rate of 8.888 Mbps is derived in [3]. The common sampling rate of $93\frac{1}{3}$ Mhz is typically used because it facilitates converting the signal from IF to baseband. The mixing process can simply be performed by multiplying the samples by the sequence $1, j, -1, -j, 1, j, -1, \cdots$.

Table 5.1: Parameters for the test signal

Waveform	16-APSK
Frequency	70 MHz
Data Pattern	PN11
Data Rate	8.889 Mbps
Output Power Level	-20 dBm

5.2.3 Input Data Generation

The input data used for initial tests was generated in MATLAB. First, a sequence of random bits was converted into random APSK symbols. The 256-bit ASM is also converted into a 64 symbol APSK message and appended to the front of the payload symbols. Next, the signal is filtered using the pulse shaping filter mentioned briefly in Section 2.4. White Gaussian noise can be added to the signal as desired to model the AWGN channel. Finally, any necessary channel gain and channel phase conditions are included.

5.3 Test Setup

The equipment setup shown in Figure 5.1 is used to test the receiver. The signal generator produces an IF signal with the parameters listed in Table 5.1 which is fed to the ADC. For BER measurement tests, the signal generator is also used to add White Gaussian noise to the transmit signal. Additionally, the signal generator's output power level is set to -20 dBm to accommodate the ADC maximum input power level.

The setup also includes a host PC, which isn't shown in the diagram in Figure 5.1, that is responsible for communication between the other components. The host PC normalizes the digital signal from the ADC to unit energy before passing it the first FPGA which is the receiver. After processing and demodulating the received signal, the first FPGA sends the log-likelihood ratio (LLR) values to the graphics processing unit (GPU) by means of the host PC. Upon receiving the LLRs from the PC, the GPU can perform LDPC decoding and make decisions. The decisions are sent to the PC for BER performance analysis.

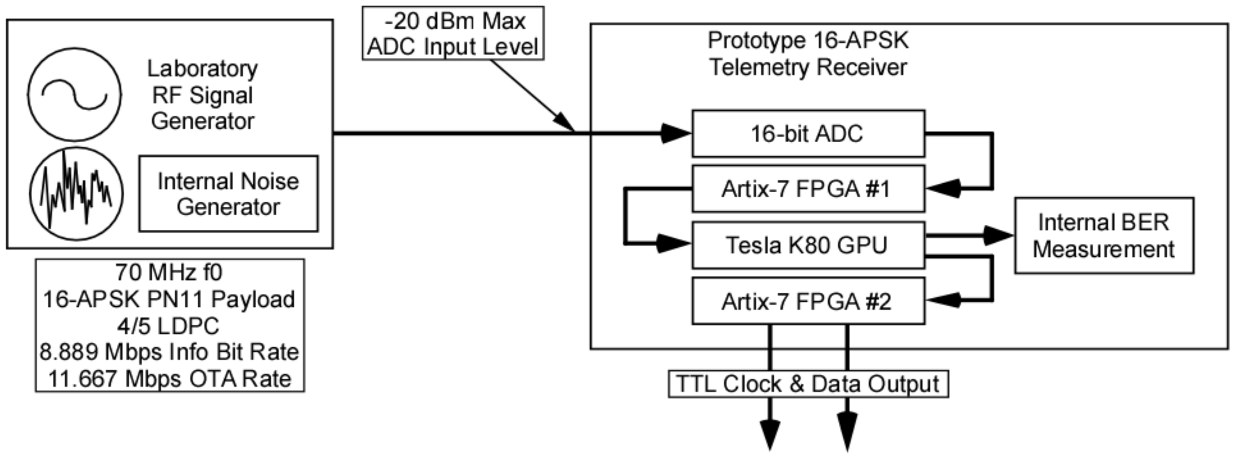

Figure 5.1: Test Equipment Setup

Chapter 6

Results & Discussion

This chapter examines the results of the FPGA implementation of the frequency offset estimator. Section 6.1 discusses the two similar approaches to carrier frequency estimation. Section 6.2 presents the simulation results for the proposed carrier frequency estimation algorithm. Section 6.3 presents the performance results of the receiver after implementing the proposed carrier frequency estimation algorithm. Section 6.4 includes a discussion of the results and analysis of the FPGA performance in comparison to simulation.

6.1 Feed-forward vs Feedback Implementation

As discussed in Section 3.2, there are two proposed methods for implementing the FFT+PLL-based carrier phase synchronization system. The first approach utilizes a feed-forward frequency estimate to initialize the PLL. Similarly, the second approach uses a feedback frequency estimate to initialize the PLL. In both approaches, the estimator algorithm itself is the same. Because both methods achieved identical simulation results, they were both in consideration to be added to the FPGA implementation.

The main difference in the two methods is where the received samples used for frequency offset estimation are collected. The feed-forward or "out of the loop" method uses the matched filter output samples, whereas the feedback or "in the loop" method uses the output of the interpolator to calculate its frequency estimate. Figure 6.1 shows matched filter output samples collected from the FPGA. In this case, the signal generator is configured for zero-noise plus a 10 kHz offset.

After downsampling, the filtered samples have a rate of 2 samples per symbols, so the re-

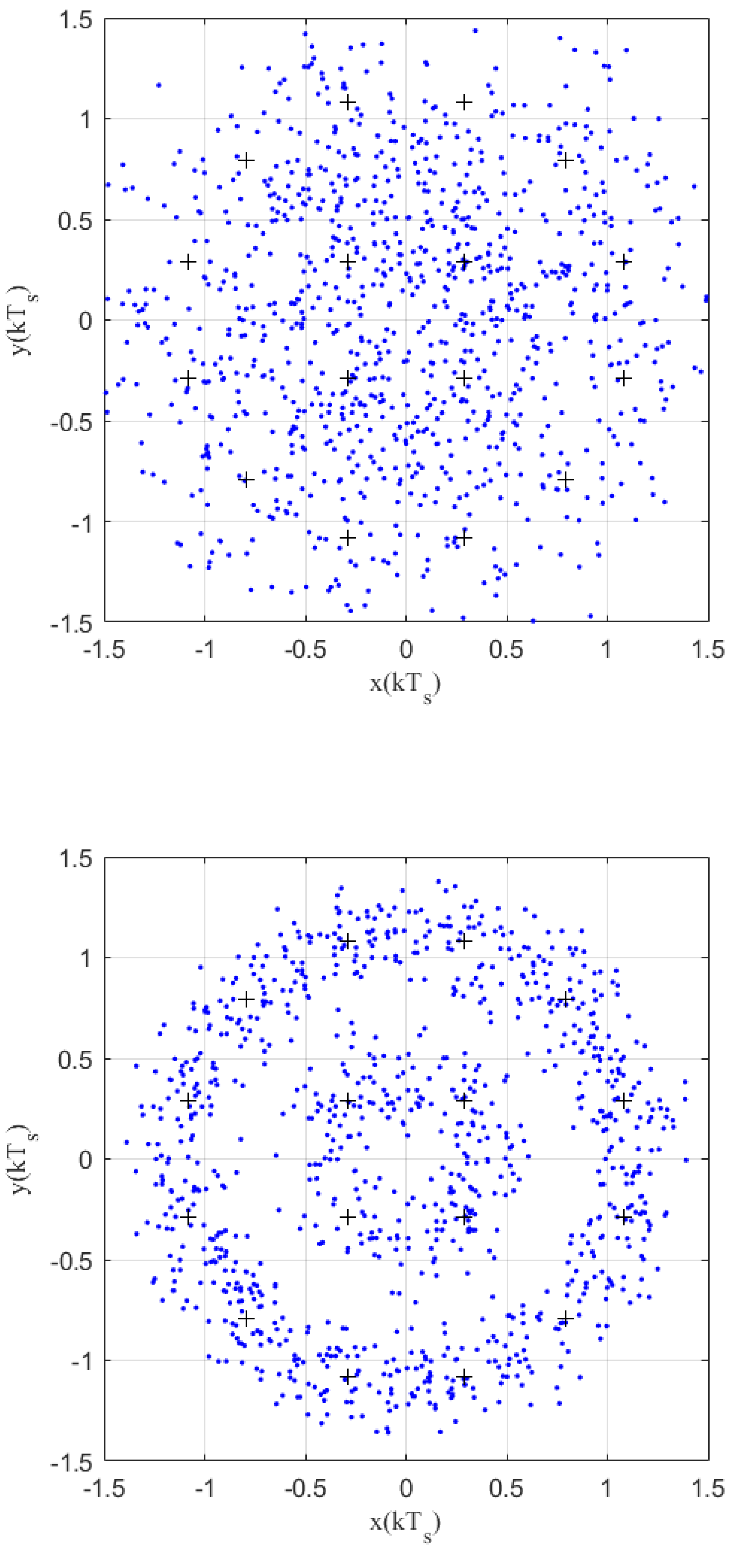

Figure 6.1: Matched filter output: Even samples (top) and odd samples (bottom)

ceived signal is separated into odd and even samples. Since the matched filter output samples are unsynchronized, the intention is that the feed-forward estimator uses one out every two samples (essentially down-converting the signal to 1 sample per symbol). However, from Figure 6.1, it becomes apparent how important symbol timing synchronization is. In the top plot, the constellation shows the effect of a sample timing offset as a result of received samples being sampled late. Even without any noise being added to the signal, the filtered samples lacks any discernible grouping around the constellation points.

From the bottom plot, the samples form visible, circular patterns that traces the constellation points. Although this is still not ideal behavior because of the noisiness present, the circular patterns, which is the result of the frequency offset, is expected. The effects of the symbol timing offset are still visible; this time the samples are only slightly early (or slightly late). Nevertheless, the estimator is unable to tell which sample, the even sample or odd sample, is sampled closer to the ideal sampling time. As a result, the feed-forward estimator is unable to correctly estimate frequency offsets even without any noise present.

Figure 6.2 shows the interpolated data before and after compensating for the phase and frequency offsets. In the top plot, before frequency estimation, the samples form a two circles that go through the constellation points. This is the previously mentioned effect of carrier frequency offset when unaffected by noise or timing errors. The bottom plot shows the samples after successfully compensating for this frequency offset. This time the received samples group around the corresponding constellation points. After the synchronization system locks, the EVM of 2.56% confirms how closely the received sample matches the transmitted symbol.

6.2 Carrier Frequency Estimator Performance

The performance of the implemented estimator is analysed through simulation using estimator error variance. Figure 6.3 shows the estimator error variance of the coarse estimate used in the FPGA implementation compared to a fine estimate that is interpolated from the values provided

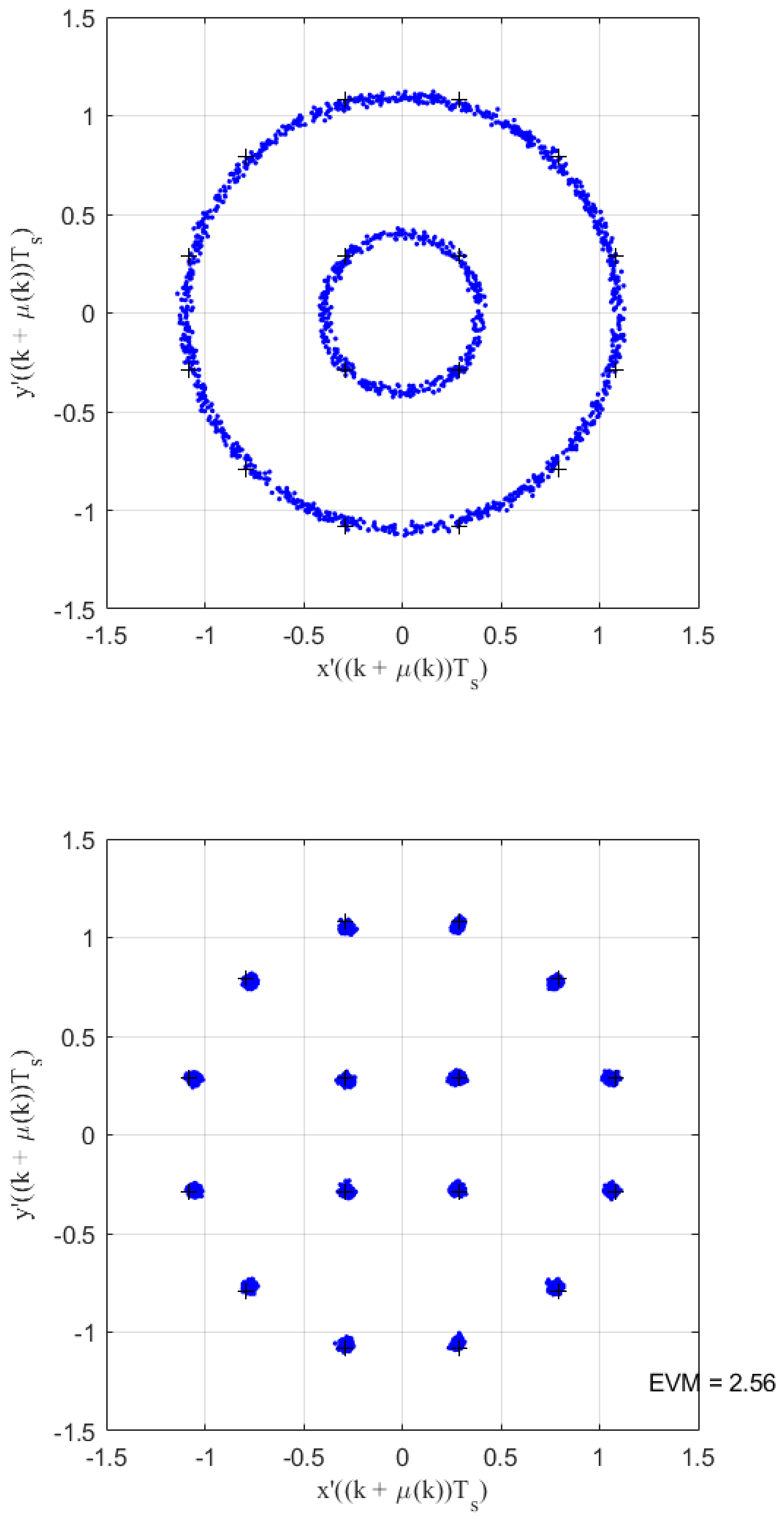

Figure 6.2: Interpolator output samples: before (top) and after (bottom) frequency offset correction

by coarse estimation. To analyze the effectiveness of the estimator, the "corner value" or the elbow of the plot is useful for visualizing the SNR range in which the estimator demonstrates good performance. The "corner value" can be described as the point at which the change in error variance becomes more gradual as E_b/N_0 increases. The results in Figure 6.3 show a "corner value" around E_b/N_0 = 8 dB. Furthermore, the interpolator improves the estimator error variance by approximately one order of magnitude.

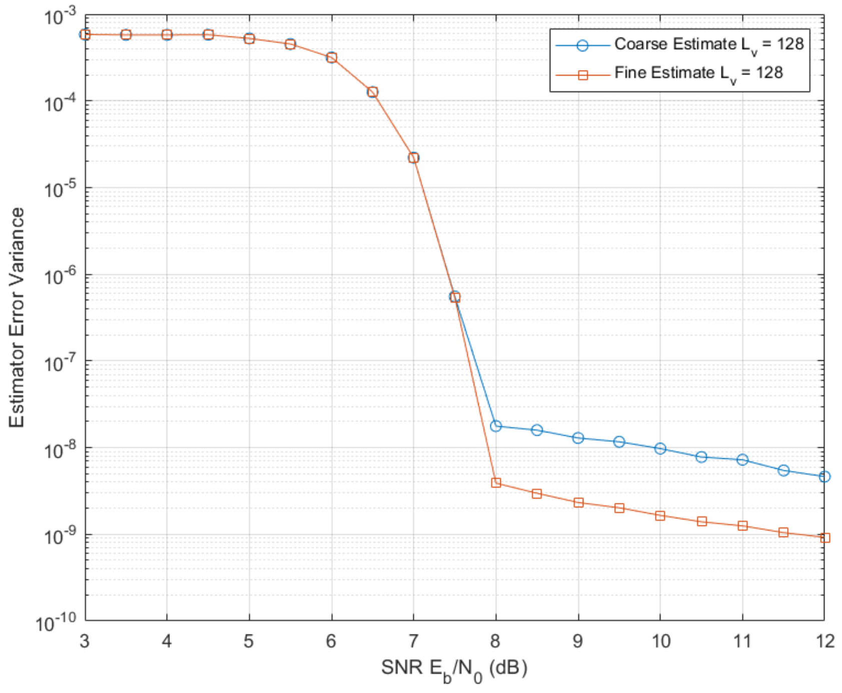

Figure 6.3: Estimation error variance of the implemented estimator

Figure 6.4 shows how much the performance of the estimator improves by doubling the length of the FFTs and performing interpolation. The simulation results compares the error variance for $L_v = 128$ and $L_v = 256$. The length of the FFTs used are $L_F = 2L_v = 256$ and $L_F = 512$; however, 32 FFTs are accumulated in each case. The resulting algorithms use 8,192 symbols and 16,384 symbols, respectively. Consequently, the "corner value" improves from around E_b/N_0 = 8 dB to E_b/N_0 = 6.5 dB. In addition, the estimator error variance is nearly three orders of magnitude better than the implemented estimator.

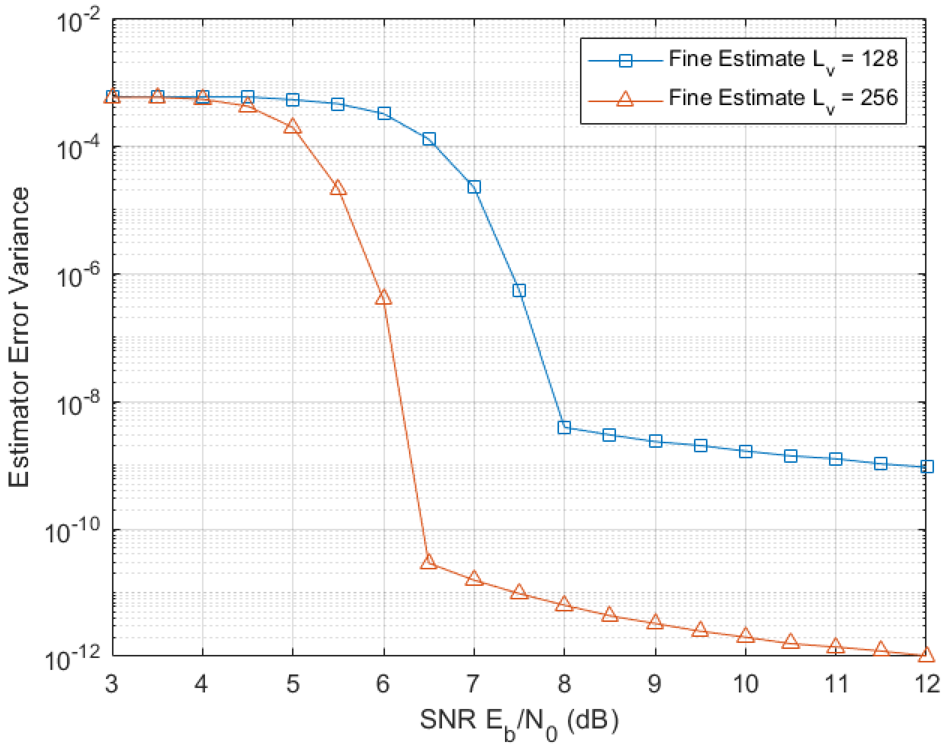

Figure 6.4: Estimation error variance for $L_v = 128$ vs $L_v = 256$

6.3 Receiver Performance Assessment

The FPGA-implemented receiver performance is evaluated in the presence of noise and a frequency offset. Figure 6.5 shows the BER performance of the receiver when affected by a 100 kHz frequency offset. The 100 kHz offset is near the limits of the frequency estimator's acquisition range. For comparison, the figure also includes the BER of the receiver, without a frequency estimator, when the carrier frequency is not offset. This receiver, with a PLL-based carrier synchronization system, is shown to have performance that is only 0.2 dB worse than the theoretical BER in [3]. Nevertheless, in Figure 6.5, an additional, minuscule BER penalty can be observed. This negligible penalty is likely caused by error accumulation from the estimate provided by the frequency estimator. This observation is explained further in the following section.

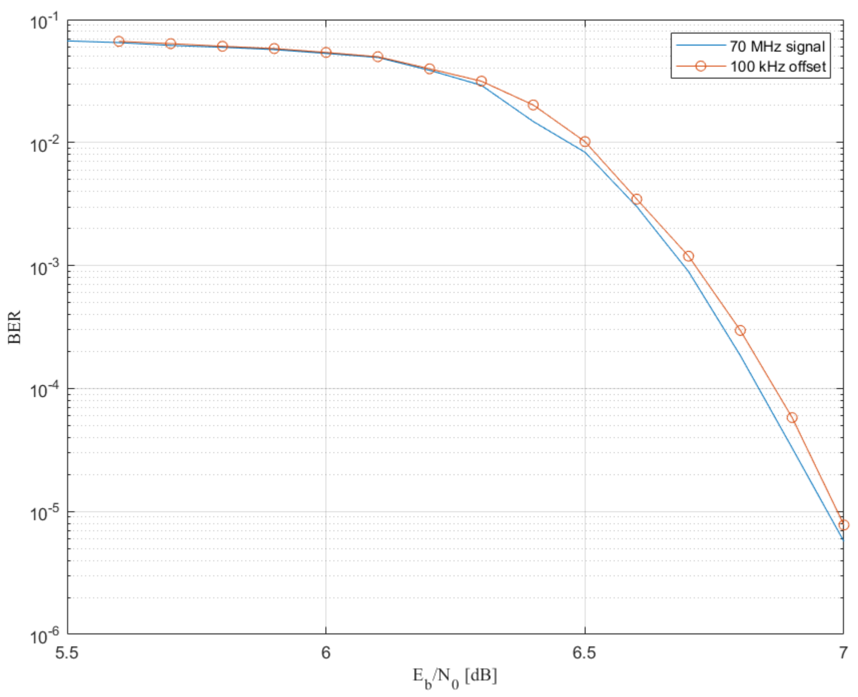

Figure 6.5: Receiver BER performance results

6.4 Observations

Initially, the objective was to implement a feed-forward frequency estimation algorithm. After all, there are several benefits to using a feed-forward estimator, including a typically faster acquisition time than its feedback counterpart. However, there is a negligible difference in the acquisition intervals of the two methods. The feedback algorithm could potentially include a bias in its estimate, but the phase estimator would correct it along with any residual frequency offset. Moreover, because the symbol timing synchronization system is implemented using the phase-corrected symbols, using the synchronized samples to derive the frequency estimate yields an inherently feedback design. Nevertheless, the results in Figure 6.2 show that implemented algorithm achieves an EVM of less than 3%. This EVM is close to the ideal value of 0% error, which corresponds to all the received symbols perfectly superimposing the constellation points.

One way the proposed frequency estimator can be assessed independently is by observing its "corner value" from the error variance plots shown in Figures 6.3 and 6.4. These results show that the frequency estimator is able to achieve good performance for $E_b/N_0 \geq 8$ dB. Furthermore, simply doubling the length of the FFT, demonstrated a considerable reduction in error variance for $E_b/N_0 \geq 6$ dB. These simulation results are comparable to the frequency estimation methods discussed in [1] and [4]. The FFTs used for this estimation algorithm are of length $2L_v$. Although the number of symbols used for estimation here are still similar to alternative FFT-based algorithms, the averaging effect is achieved by accumulating the results of several small FFTs. In an FPGA-implemented design when resources are limited and all important, this algorithm promises to be much more resource efficient than other alternatives.

Another way to improve the coarse frequency estimate is by interpolation. The interpolator mentioned in the previous section uses three points from the accumulated periodogram to refine the coarse estimate [4]. The coarse estimate is derived from the maximum value of the periodogram, so the interpolation process also uses the points on either side of this peak. By denoting the maximum of the periodogram as P_2, the point to its left as P_1, and the point to its right as P_3; a parabolic interpolation technique is performed [6]. The interpolation formula becomes

$$\hat{f} = \hat{f}_o + \frac{1}{2 \times 12 \times L_F} \frac{P_1 - P_3}{P_1 - 2P_2 + P_3}, \tag{6.1}$$

where $\hat{f}_o$ is the coarse estimate and $L_F = 2L_v$ is the length of the FFT used.

While interpolation can be implemented with minimal additional cost, some of the suggested methods for refining the frequency estimate include some drawbacks. Doubling the length of the FFTs used in the estimator would double the acquisition time. To get an understanding of the acquisition time, the following points can be made:

1. The frequency estimator's observation length is 8192 symbols
2. The phase estimator's observation length is less than 1344 symbols (within one LDPC frame) [3]

Doubling the length of the FFTs would require that the number of FFTs that are accumulated be halved in order to not increase the observation length. This change could also be implemented along with the interpolator if it were deemed necessary.

Figure 6.5 shows that the 16-APSK receiver is able to successfully demodulate a transmit signal with a frequency offset of at least 100 kHz. However, that is well within the estimation range of the frequency estimator. Using Equation 3.7 and the sampling rate listed in Section 5.2.2, the actual estimation range can be calculated:

$$\begin{aligned} |\Delta f| &< \frac{F_s/N}{2 \cdot 12} \\ &< \frac{\frac{93.333\text{ MHz}}{32\text{ samples/symbol}}}{2 \cdot 12} \\ &< 121.5\text{ kHz}. \end{aligned} \tag{6.2}$$

This estimation range is a great improvement to the experimental estimation range of the PLL-based system, which was measured to be roughly 10 kHz.

Chapter 7

Conclusions

The goal of this thesis is to implement a frequency estimator to provide an initial frequency offset estimate for the carrier synchronization system of an FPGA-implemented 16-APSK receiver. The proposed frequency estimation algorithm was expected be resource-efficient enough without requiring a long observation interval. Using simulation, the frequency estimator was determined to be sufficiently accurate, in terms of its error variance, for $E_b/N_0 \geq 8$ dB. The 16-APSK receiver's performance was measured in terms of BER and EVM. The proposed "hybrid" carrier synchronization system demonstrated the capability to compensate for phase error accumulations caused by the frequency estimator.

The results also showed that the frequency estimator successfully improved the frequency estimation range of the receiver to approximately 121.5 kHz while maintaining BER performance close to the theoretical expectations. In contrast, the PLL-based system is shown to theoretically only achieve frequency lock for up to 30 kHz offsets [1]. Furthermore, the frequency estimator was able to attain an acceptable trade-off between estimation accuracy, which was discussed in terms of both error variance and BER, and acquisition time (based on its observation length).

One way this work can be further extended is to experimentally improving the BER performance. Specifically, investigating whether the slight BER worsening can be prevented by improving the coarse estimate using interpolation and/or by increasing the observation length of the frequency estimator. Both of these options have shown promising results in simulation; however, further research is required to determine how useful this modification is in the context of this work. Since the resource utilization of the FPGA is not an immediate concern based on Table 4.1, it may be possible to achieve a better trade-off between acquisition time and estimator accuracy, and subsequently compensate for the frequency quantization error.

References

[1] M. Rice, B. Redd, and X. Briceno, “On carrier frequency and phase synchronization for coded 16-apsk in aeronautical mobile telemetry.” International Foundation for Telemetering, 2019.

[2] M. Rice, *Digital communications: a discrete-time approach*. Pearson Education India, 2009.

[3] J. Baxter, “An fpga implementation of carrier phase and symbol timing synchronization for 16-apsk,” Master’s thesis, University of Kansas, 2020.

[4] B. Redd, J. Ebert, and A. Twitchell, “Dft-based frequency offset estimators for 16-apsk.” International Foundation for Telemetering, 2019.

[5] J. E. Volder, “The cordic trigonometric computing technique,” *IRE Transactions on electronic computers*, no. 3, pp. 330–334, 1959.

[6] A. A. D’Amico, A. N. D’Andrea, and R. Regiannini, “Efficient non-data-aided carrier and clock recovery for satellite dvb at very low signal-to-noise ratios,” *IEEE Journal on selected areas in communications*, vol. 19, no. 12, pp. 2320–2330, 2001.

[7] C. Josephson, “On the design of a square-root nyquist pulse shaping filter for aeronautical telemetry.” International Foundation for Telemetering, 2017.

[8] E. Chu and A. George, *Inside the FFT black box: serial and parallel fast Fourier transform algorithms*. CRC press, 1999.

[9] G. N. Tavares, L. M. Tavares, and A. Petrolino, “An improved feedforward frequency offset estimator,” *IEEE transactions on signal processing*, vol. 56, no. 5, pp. 2155–2160, 2008.

[10] M. Morelli and U. Mengali, “Feedforward frequency estimation for psk: A tutorial review,” *European Transactions on Telecommunications*, vol. 9, no. 2, pp. 103–116, 1998.

[11] Y. Wang, E. Serpedin, and P. Ciblat, “Non-data aided feedforward estimation of psk-modulated carrier frequency offset,” in *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No. 02CH37333)*, vol. 1. IEEE, 2002, pp. 192–196.