UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

IMPLEMENTATION AND ANALYSIS OF ADAPTIVE SPECTRUM SENSING

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

RYLEE MATTINGLY
Norman, Oklahoma
2021

IMPLEMENTATION AND ANALYSIS OF ADAPTIVE SPECTRUM SENSING

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Acknowledgments

I want to thank Dr. Justin Metcalf and Dr. Jay McDaniel for giving me the opportunity to work at the ARRC and discovery the incredible world of radar systems and electromagnetics. Additionally, I want to thank Dr. Nathan Goodman and Dr. Cliff Fitzmorris for agreeing to be on my committee and taking the time to review my work.

I would also like to thank my sister, Cassi Marlow, and my parents, Ricky and Treva Mattingly. I am always grateful for their incredible support and encouragement to pursue my goals and persevere through great challenges.

Finally, I want like to thank all of my friends at the ARRC for helping provide support when encountering especially difficult challenges.

I thank the Defense Advanced Research Projects Agency (DARPA) for their support of this research under grant HR0011-20-1-0007.

Table of Contents

List of Tables

List of Figures

Abstract

The electromagnetic spectrum is a finite resource that has become increasingly crowded as the day-to-day operation of the world has become increasingly reliant on wireless devices. With the growing deployment of the Internet-of-Things (IoT), 5G Networks, and broadband internet systems, the available spectrum for radar applications has been reduced and instances of interference across all device types have increased. To mitigate this problem going forward, devices need to be better able to intelligently access the spectrum based on the presence of other users.

A cognitive radio or radar system functions by using adaptive spectrum sensing to detect existing users in the frequency band and adapt to use 'open' spectrum bands. To ensure the predictable performance of the system and systems that it shares spectrum with, it must detect new users and adapt without interrupting its operation or interfering with the other users. Because modern communications networks can update their spectrum utilization on a sub-millisecond timescale, the critical detection and adaption phase must operate in real-time.

This work presents an implementation of a fast spectrum sensing (FSS) algorithm deployed on the field-programmable gate array (FPGA) of an Ettus USRP$^{\text{TM}}$ software-defined radio. This implementation allows for microsecond scale updates of the environment's spectrum availability. Unfortunately, this FSS algorithm is limited by its knowledge of the spectrum, which is ever-changing. To help improve the system's dynamic performance a new adaptive detection algorithm is proposed to replace the static threshold of the first implementation. The new detection algorithm is a constant false alarm rate (CFAR) inspired detector which allows a cognitive sensor to work in a dynamic environment without a-priori information about the spectrum. Combining the FSS algorithm with dynamic signal detection allows the cognitive radio system to adapt to the ever-changing environment without requiring extensive 'listen before talk' periods before operation.

Chapter 1

Introduction

Today, almost everyone takes advantage of technology or devices that use the electromagnetic spectrum for operation. With a projected 29.3 billion networked devices by 2023, 23.5 billion in IoT and mobile categories, a huge number of wireless devices exist in the world [2]. Without deterministic access to this resource, many critical systems would fail or become unreliable. To mitigate interference, spectrum is licensed to primary users restricting the use of those frequencies to specific applications. While this licensing protects primary users of the band, the band utilization varies widely based on geographic location, leaving gigahertz of spectrum underutilized [3].

Recently, unlicensed bands have been approved for wireless communications use. These bands share spectrum using a listen-before-talk (LBT) technique [4] that requires an incoming unlicensed user to 'listen' for other users in the spectrum for some time before operating. Similarly, the unlicensed user must be prepared to immediately stop operating whenever a primary user is detected. While effective, operation time can be greatly impacted as new primary users enter and leave the environment, diminishing the performance for the secondary users.

LBT technique adoption and the availability of unlicensed bands shows a willingness on the part of regulators and the industry to try, at least on a limited basis, spectrum sharing to be used. The expansion to faster and more performant techniques of spectrum sensing and avoidance is necessary to spur further adoption and better spectrum management in the future.

This thesis analyzes existing fast spectrum sensing algorithms that allow a system to detect other users of the electromagnetic spectrum. By detecting a primary user, a secondary emitter to limit its interference to the primary users. Further, the implementation of these algorithms in an FPGA on commercial off-the-shelf (COTS) hardware is investigated, with a focus on real-time performance.

While the real-time knowledge of other emitters would be useful to all wireless users to improve performance and avoid interference, a special focus on radar systems is considered. Specifically, the pulsed radars allow spectrum estimates to be gathered while the radar is not transmitting, eliminating the need to censor the system's emissions when creating estimates. Additionally, the sensitivity of radar makes its interference avoidance that much more important.

The initial implementations of this system will use a static threshold, but because of the dynamic nature of the electromagnetic spectrum, a dynamic threshold estimation technique will also be analyzed. By estimating the noise of the spectrum, a better understanding of the environment over time can be used to set the spectrum threshold. This necessitates an investigation into estimation and signal detection techniques.

1.1 Contributions

This thesis makes the following contributions:

- Real-time FPGA implementation of static threshold, greedy fast spectrum sensing (FSS) algorithm. Developed novel rotating bucket architecture to minimize latency and utilization resources.

- Demonstrates hardware acceleration on commercial off-the-shelf USRP radios. Leveraged open-source RF Network-on-Chip (RFNoC) framework to implement algorithms with real-time performance.

- Develops CFAR inspired noise estimation method designed with implementation on FPGA in mind. Proposed pipelined implementation to integrate threshold estimate and FSS into a single block.

1.2 Thesis Outline

This thesis is divided into the following chapters: In Chapter 2, a background on cognitive radios is given with a discussion on their functional cycle. The fast spectrum sensing algorithm is presented as the static threshold case. Next, is a discussion on dynamic signal detection techniques that are used as a primer for the dynamic threshold method implemented later. Included is a complexity analysis of the various types of signal detection methods.

Chapter 3 describes the implementation platform that was used for development. Details about the RF Network-on-Chip (RFNoC$^{\text{TM}}$) FPGA framework are provided, including issues that were addressed and limitations that were discovered. This chapter also provides a high-level overview of the software used with the platform and how the Ettus X310 radio and host computer operate together.

In Chapter 4, the architecture for the static FSS algorithm is presented. The state machine is discussed with an overview of the data flow in and out of the algorithm block. An analysis of the clock level timing of the block is presented as a basis for the discussion on performance. Performance metrics will be derived that provide a numeric performance figure so that the performance of the adaptive implementation can be compared later. Finally, FPGA resource utilization will be presented using the Vivado reports generated when the FPGA design was synthesized.

In Chapter 5, a detailed description of the proposed hardware optimized signal detection method is presented. Real-world data will be presented that provides a basis for the need for the new algorithm. Additionally, a 5G Downlink signal simulator that was used to benchmark this new process is presented. Performance analysis from the simulation across SNR is described with the probability of false alarm, probability of detection, and resulting FSS of particular interest. The chapter concludes with the new algorithm being applied to the same real-world data as was presented at the beginning of the chapter.

Chapter 6 proposes an FPGA architecture to implement the detection method described in Chapter 5. An analysis of the hardware structures necessary to the device's operation and the resources required will be given along with a discussion on operational performance. A special look at the latency of the FPGA block will be provided through a cycle-level timing analysis of the block and the previously derived performance metrics.

In Chapter 7, a conclusion to the work is provided with a discussion about opportunities for future work.

Chapter 2

Background

Before discussing the implementation details of fast spectrum sensing or signal detection, the background for the algorithms and existing methods that served as the foundation of this work should be introduced. First, an introduction to the operation of a cognitive sensor is given with an introduction to the FSS algorithm. This chapter concludes with a discussion on detection methods. Specifically, energy detection and cyclostationary detection will be analyzed.

2.1 Cognitive Sensors and Greedy Fast Spectrum Sensing

The cognitive perception-action cycle (PAC) underlays the primary operation loop of a cognitive sensor. The cycle works in three stages: sense, decide, and adapt. Sensing should be occurring at all times on the sensor. Continuous sampling of the spectrum provides the system with a constant flow of data. The stream can be divided into 'frames' for which the fast Fourier transform (FFT) is applied.

These frames then act as spectral snapshots that can be sent to the second stage of the PAC. To generate the spectral frames the input stream is directly cut into frames of K samples with no sample overlapping across frames. K is an arbitrary value that can be set based on the desired frequency resolution given the bandwidth of the system.

When the spectral frames arrive in the second stage, they are used to determine where signals may exist and the location of optimal band unoccupied spectrum for the system to operate in. The operation information is then sent into the third stage of the PAC: 'adapt'. Inside this third stage, a waveform selection system takes in the operation recommendation and adapts by transmitting the selected waveform in the appropriate spectrum. The PAC described above is shown as a streaming methodology in Figure 2.1.

The learning stage is where this work focuses and where the FSS algorithm lives. In this second stage, the system must 'learn' from the spectral frames and decide where other signals exist and what frequencies are available for its function. To determine where signals exist, we can analyze the magnitude output of the FFT to classify each frequency bin as either a high-power or low-power sub-band [5]. It

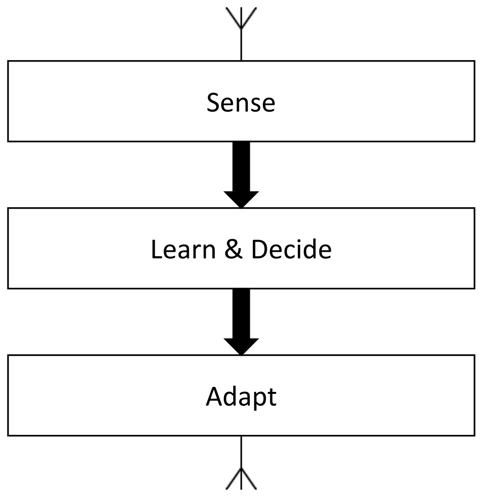


Figure 2.1: Perception action cycle for a cognitive sensor streaming data from an antenna into the sense stage. A transmission band is calculated and the corresponding signal is generated by the adapt stage and transmitted.

is assumed that all low-power sub-bands are unoccupied or noise only and therefore available for the system to occupy. After the initial classification of each bin, the bins must be examined again to group the bins. The grouping works to eliminate narrow, low-power frequency gaps that exist between high-power bands [6]. These gaps are closed and eliminated from operational consideration as they represent either a low-power component of the present signal or a gap between signals too narrow to be useful.

Figure 2.2 shows a threshold applied to a spectral frame with a 5G downlink signal present. The 5G waveform has variation within its signal structure. This variability causes a frequency bin, clearly within the 5G waveform's operational frequency, to fall below the threshold. The sub-band merging process would catch cases like this and prevent that single bin, or even an operator-defined number of

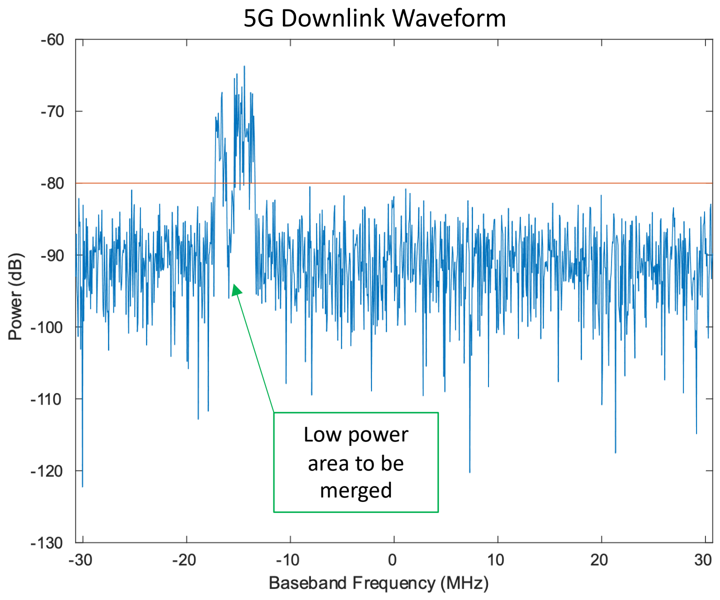


Figure 2.2: A 5G downlink signal shown with a threshold used to sort bins into high and low power groups. High power groupings with a small number of low power bins are grouped as they are part of a signal.

continuous bins, from being processed further for consideration. By merging these closely spaced bands, a better understanding of the location of the signal emerges.

For this work, a greedy algorithm is used to find the largest number of continuous open bins. This makes the merging of small bins seem redundant as those bins won't be selected anyway. It is still important to do the merging, however, so the bins don't have to be considered when looking at the continuous spaces. Merging is critically important for other types of uses of FSS that optimize for criteria other than the largest band. This could mean more complex computation may be carried out on the low power sub-bands than the greedy approach requires, which would be more costly.

FSS, in its original form, has a static threshold set manually by the operator from spectrum information gathered before the system is put into operation. This can cause the system to repeatedly fail or give a false positive over time. For this reason, signal detection techniques will now be introduced as a basis for the proposed dynamic threshold estimation discussed later.

2.2 Signal Detection

Many methods of signal detection exist today. All attempt to provide the highest possible probability of detection while balancing a minimum number of false alarms . This work also considers mathematical complexity and data requirements as additional constraints. This section will discuss two detection methodologies: cyclostationary feature detection and energy detection. Three different implementations of energy detection will be shown: classic energy detection, cell averaging CFAR, and order statistic CFAR.

2.2.1 Cyclostationary Feature Detection

Human-generated signals have structures and “features” that are not found in natural noise which is usually modeled as a white Gaussian process [7]. If these features can be detected then signals should be detectable even in low signal-to-noise ratio (SNR) circumstances. Detection of signals in low power is one of the strengths of cyclostationary feature detection.

The autocorrelation function, $R_{XX}(t_1, t_2)$, provides the expected value of the product of the random process at time t_1 and t_2. Further, the distance between t_1 and t_2 or lag time, τ has its own properties for certain processes. One such case of interest is for wide sense stationary (WSS) signals.

WSS signals have an autocorrelation function that is strictly dependent on the lag time [8]. That is to say, for a WSS, if the lag between two points is the same, then the autocorrelation function is the same. This stationary autocorrelation function can be described by

$$R_{xx}(\tau) = E\{\mathbf{x}(t + \frac{\tau}{2})\mathbf{x}^*(t - \frac{\tau}{2})\} \tag{2.1}$$

where $\tau = t_1 - t_2$.

However, man-made signals are not strictly stationary and are instead called cyclostationary [9]. A cyclostationary signal’s autocorrelation function is represented as a periodic or almost periodic function [10]. Unsurprisingly, the periodicity of the autocorrelation function of a cyclostationary process can be described as the Fourier coefficients.

$$R_{xx}(\tau) = \sum_{\alpha} R_x^{\alpha}(\tau) e^{j2\pi\alpha t} \tag{2.2}$$

where α is the fundamental cyclic frequency.

For each value of τ, the series in alpha describes the periodic nature of its autocorrelation. This function, $R_x^\alpha(\tau)$, is called the cyclic autocorrelation function (CAF) [10]. A frequency-domain dual of the CAF also exists and is named the spectral correlation function (SCF). Similar to the CAF, the SCF provides the cyclic frequency content of a process over its frequency.

The SCF function can be found by taking the Fourier transform of the CAF.

$$S_x^\alpha(f) = \int_{-\infty}^{\infty} R_x^\alpha(\tau) e^{j2\pi f\tau} d\tau \tag{2.3}$$

It is of note that the $\alpha = 0$ value across frequencies is the power spectral density (PSD) of the signal. This is where this method excels, when the PSD is completely hidden within noise the other cyclic frequencies remain prominent. By remaining visible across other α values, the signal is still detectable.

Unfortunately, it is not practical to directly calculate a set of SCFs for a given real-world signal. Therefore the SCF must be estimated as a cyclic periodogram, in this case using frequency smoothing [11],

$$S_x^\alpha(f) \triangleq \frac{1}{\Delta f} \int_{f-\Delta/2}^{f+\Delta/2} \frac{1}{T} X_T(v + \alpha/2) X_T^*(v - \alpha/2) dv \tag{2.4}$$

Here $X_T(v + \alpha/2)$ is the Fourier transform of $x(t)$.

To carry out the frequency smoothing approach, a method called the FFT accumulation method (FAM) is used to demonstrate the computational complexity of the algorithm. The FAM method starts with several parameters: N, N', P and, L. N describes the number of samples in the signal observation. N' sets the frequency resolution as it is the length of the initial FFT operation. P sets the number of spectral frames of data that will be considered, while L is the overlap factor of the FFT.

P and L are derived from N' and N and are given by the following:

$$L = N'/4 \tag{2.5}$$

$$P = N/L \tag{2.6}$$

Equation 2.5 shows a value of four selected to set the overlap. As the value increases and the overlap approaches N' so does the computational complexity of the algorithm. A divisor of four was shown to be a sufficient middle ground for computational complexity and preventing cycle leakage [12].

Before considering the FAM method the resulting domain should be formed. The desired output is a frequency-alpha matrix. If the sampling frequency is defined as f_s, the frequency dimension of the resulting field ranges from $-f_s/2$ to $f_s/2$ across $2N'$ bins. The α dimension ranges from $-f_s$ to f_s populating $2N$ bins. the resulting field is quite large even considering that only a diamond-shaped area of it will be used.

Starting the estimate, an $N' \times P$ matrix is constructed such that a sliding FFT can be applied to the columns. The sliding factor of L is used to create a matrix from the signal, s, shown as:

$$\boldsymbol{Y} = \begin{bmatrix} s(1) & \dots & s((P-1)L+1) \\ s(2) & \dots & s((P-1)L+2) \\ \vdots & \ddots & \vdots \\ s(N') & \dots & s((P-1)L+N') \end{bmatrix} \tag{2.7}$$

A window is then applied down the columns before the FFT of each column is taken.

Each element of the matrix resulting from the previous step is multiplied by an exponential to smooth across frequency and compensate for any phase distortions introduced by the FFT. A complex conjugate matrix is generated, yielding the following 2 matricies:

$$\bar{\boldsymbol{Y}} = \begin{bmatrix} S(0,f_1) & \dots & S((P-1)L,f_1) \\ S(0,f_2) & \dots & S((P-1)L,f_2) \\ \vdots & \ddots & \vdots \\ S(0,f_{N'}) & \dots & S((P-1)L,f_{N'}) \end{bmatrix} \tag{2.8}$$

$$\bar{\boldsymbol{Y}}^* = \begin{bmatrix} S^*(0,f_1) & \dots & S^*((P-1)L,f_1) \\ S^*(0,f_2) & \dots & S^*((P-1)L,f_2) \\ \vdots & \ddots & \vdots \\ S^*(0,f_{N'}) & \dots & S^*((P-1)L,f_{N'}) \end{bmatrix} \tag{2.9}$$

An element multiply is carried out for each combination of rows in the matrix. The result of the operation is N^2 P length vectors, each at a new frequency and central alpha calculated using the following.

$$f_j = \frac{(f_k + f_l)}{2} \tag{2.10}$$

$$\alpha = f_k - f_l \tag{2.11}$$

Here f_j denotes the new frequency, f_k is the frequency from the row of the non-conjugate matrix and f_l is frequency from the row of the conjugate matrix. The new frequency is calculated for every combination The frequency value is used to index into the frequency-alpha matrix while the α is the center of the alpha vector denoted by

$$\alpha + q\Delta\alpha \tag{2.12}$$

where $\Delta\alpha$ is the resolution of the alpha component of the frequency-alpha matrix and q ranges from $-\frac{P}{2}$ to $\frac{P}{2}$ with a length of P.

A P-point FFT is taken across the rows after the multiplication. Unfortunately, only the central half of the elements of the resulting vectors are considered a good estimate of α, which means calculated values were not all used [12]. Therefore, the central components of the estimate are inserted on the frequency-alpha matrix, across alpha, with an index of (f_j, α_i) denoting the center point. This populates a diamond shape in the resulting matrix.

To determine the presence of a signal at any given frequency-cyclic frequency location, a simple hypothesis test can be employed. This is trivial as the threshold for this method can be pre-generated using the estimated SCF of a noise-only signal.

The parameters used to create the matrices that power this method can be used to describe the calculation complexities. There are a total of $2N'P$ multiplies required to taper the data, $PN'N'$-point FFTs must be carried out and $2N'P$ frequency shifts are carried out [12]. This is a massive amount of calculations, although most of the operations could be carried out in parallel, but instantiating many hundred or thousands of parallel FFT logic blocks would quickly devour the available floor space on the FPGA. A GPU could be used to accomplish this task but that would introduce latency as the data is transferred to the host PC.

Another big constraint that is connected to the complexity issue is observation length. To take advantage of the repeating structures of signals the observation of signal must be longer than a single spectrum estimate that may be used for simpler methods. The FAM method itself requires what are essentially spectral estimates to populate the columns of the first matrix. These constraints are the reasons that this thesis concentrates on energy detection methods. However, future work will examine efficient implementations of FAM or other cyclostaionary estimation methods to enable signal detection.

2.2.2 Energy Detection

Energy detection describes an entire class of algorithms that are used for determining the presence of a signal based on power present in a sample of the electromagnetic spectrum. These methods work by estimating the noise floor of the environment and calculating a test statistic to apply to determine if a signal is present in a given frequency bin. Because this is only considering the power in a single snapshot the detection quality is based on the quality of the noise estimate. This section will discuss three specific methods: classic energy detection, cell averaging constant false alarm rate detection (CA-CFAR), and order statistic CFAR (OS-CFAR).

2.2.2.1 Classic Energy Detection

Classic energy detection defines the two probability distributions as Gaussian for the noise-only case, and noncentral chi-squared for the signal present case [13]. For this method, the test statistic, V, is defined as the sum of the squared energy over a time T, defined by

$$V = \frac{1}{T}\int_{t-T}^{t} x^2(t)dt \tag{2.13}$$

For a signal of length T and bandwidth B to be captured effectively, $2TB$ samples must be taken or TB complex samples. This means that the test statistic becomes the sum of $2TB$ samples. $2TB$, therefore, becomes the non-centrality parameter while the per-sample input signal to noise ratio (SNR) sets the mean of the noncentral chi-squared distribution. The signal-not-present distribution can therefore be defined as:

$$\boldsymbol{x_0} \sim \chi^2_{TB} \tag{2.14}$$

The signal present distribution is defined as:

$$\boldsymbol{x_0} \sim \chi^2_{TB}(\mu) \tag{2.15}$$

with the μ as the non-centrality parameter.

With the previous parameters set, the threshold, V_T', for the test statistic can then be calculated using the following expression with the desired probability of false alarm [13].

$$P_{FA} = Prob\,[\boldsymbol{x_0} > V_T'] \tag{2.16}$$

Further, the probability of detection can be calculated using a similar probability expression or with the complementary error function erf(•).

$$P_D = Prob\,[\boldsymbol{x_p} > V_T'] = \frac{1}{2}\,\mathrm{erfc}\left[\frac{V_T' - 2TB - SNR}{2\sqrt{2}\sqrt{TB + SNR}}\right] \tag{2.17}$$

The SNR term is very limiting in this estimation approach. Signal SNR is one of the unknowns that signal detection and estimation are trying to solve. CFAR detectors mitigate this problem by employing a noise estimate that does not rely on SNR to create the threshold for their hypothesis tests.

Since both classic energy detection and CFAR use hypothesis testing, a brief discussion of the topic is presented here to give an intuition of what each probability value means. Figure 2.3 shows two Gaussian distributions with a vertical line denoting the threshold selected. These distributions were used for the sake of this

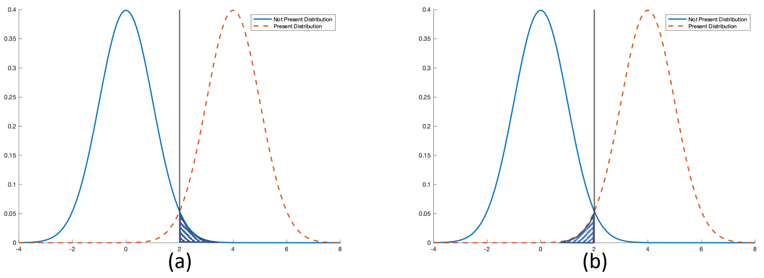


Figure 2.3: Two Gaussian distributions. The not present distribution with a mean of 0 and the present distribution with a mean of 4. (a) The probability of false alarm shown as the highlighted area. (b) The probability of missed detection is shown as the shaded area.

discussion and do not necessarily represent any detection method. The area in plot (a) shows the probability of false alarm. That is the area on the side of the threshold that registers a detection that is still under the not present curve. Similarly, the probability of a missed detection is the area to the left of the threshold that still lies below the present curve.

2.2.2.2 Constant False Alarm Rate Detectors

CFAR detectors, as the name implies, adjust the threshold for each test, or each set of tests, to maintain a consistent number of false alarms coming through the system. Keeping this metric consistent helps to set the performance requirements for the systems that will be receiving the output of the detector. This is accomplished by observing the relationship between the threshold, T, and the probability of false alarm, P_{FA}, for a square law detector for a signal in complex Gaussian noise using the following expression [14].

$$T = -\sigma_w^2 ln(P_{FA}) \tag{2.18}$$

This relation relies strictly on the interference power, σ_w^2, which is not known. The interference power must, therefore, be estimated. Although there are numerous ways to accomplish this, only two will be discussed here.

It can be shown that the maximum likelihood estimate of σ_w^2 is the average of the N available training samples [14]

$$\widehat{\sigma_w^2} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{2.19}$$

Cell averaging CFAR (CA-CFAR) takes advantage of this and takes the average of the 'cells' surrounding the cell under test. To prevent a signal that may be present in the cell under test from skewing the estimate, the cells around the cell under test are not included in the average. These excluded cells are called guard cells while the cells used are called reference cells. Figure 2.4 demonstrates this kind of structure.

Because an estimate for σ_w^2 is now being used, $ln(P_{FA})$ is no longer the correct scalar to be applied to find the threshold. This scalar is generalized as α and must be determined for each type of CFAR detection. For CA-CFAR the relation between

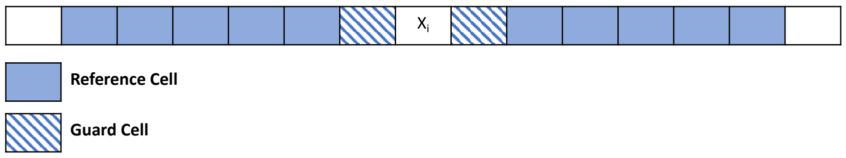


Figure 2.4: The structure of CA-CFAR used when estimating the $\widehat{\sigma_w^2}$ for cell x_i. Reference cells are evenly split on either side of the cell under test to provide a better estimate.

α_{CA} and P_{FA} are defined by

$$P_{FA} = \left(1 + \frac{\alpha_{CA}}{N}\right)^{-N} \tag{2.20}$$

From this equation, it follows that α_{CA} is defined as

$$\alpha_{CA} = N\left(P_{FA}^{-1/N} - 1\right) \tag{2.21}$$

Order Statistic CFAR (OS-CFAR) uses the same structure described in Figure 2.4, but instead of averaging the cells to create an estimate of σ, the cells in the reference are ordered and the k^{th} is selected as the estimate. Unsurprisingly, a new α value is required for this method. However, no direct solution for α exists. Therefore, it must be solved numerically using the following equation for P_{FA}

$$P_{FA} = \frac{N!\,(\alpha_{OS} + N - k)!}{(N-k)!\,(\alpha_{OS} + N)!} \tag{2.22}$$

Notice, the calculation of α in both CFAR cases only depend on P_{FA}. This means that the α value can be precalculated and given to the system as a parameter. That makes these CFAR methods very computationally efficient. These algorithms can also operate with a small spectral snapshot.

The simplified operations of CFAR, and classic energy detection to a lesser extent, do come at the cost of reduced performance. Energy detection methods require a higher SNR for detection when compared to other, more complex methods. This degraded performance is one of the consequences of minimizing computation but also have limited knowledge of the signals to be detected [13].

Chapter 3

Implementation Platform

This chapter will discuss the implementation platform as a whole, before then driving down into the FPGA frameworks that enable the architecture discussed in Chapter 4. Software-defined radio will be discussed with the interaction between the radio hardware and the host PC described. An overview of the RFNoC$^{\text{TM}}$ framework is given in 3.2 with specific attention to the interconnects that allow data to move through the device. An analysis of the available interface options within the framework will be discussed with a justification for the interface that was selected for this work. The chapter concludes with a discussion on creating and modifying framework-generated files to make a custom RFNoC block.

3.1 Software-Defined Radio

A software-defined radio (SDR), as the name suggests, allows the user to define the operation of the radio using only software without the need to change front-end hardware. Designed to be used for communications, SDRs have allowed for a shift away from application-specific hardware solutions to allow for more flexibility in application and use [15]. These systems use tunable oscillators and transceivers to allow for baseband data to be obtained from a wide range of operating frequencies.

Similarly, these radios can take baseband data and mix it up to be transmitted at higher frequencies. Many radios allow for this to be done with minimal interruption to operation, only needing to wait long enough for an oscillator lock to be achieved.

An Ettus USRP X310 radio was used for this work and has a Xilinx Kintex®7-410T FPGA on the main motherboard to handle data movement through the various RF channels. with dual UBX-160 daughter cards, pictured in Figure 3.1. This hardware allows for dynamic transmit and receive from 10 MHz to 6 GHz with up to 160 MHz of instantaneous bandwidth [16]. The X310 provides options of dual 10-gigabit Ethernet and PCI express to enable control and data transfer to and from the host PC [17]. Dual 10 gigabit Ethernet connections were used to interface with the radio to ensure the highest data throughput.

Control of the hardware is accomplished using Ettus' software library, USRP Hardware Driver (UHD). This hardware driver communicates with the radio and

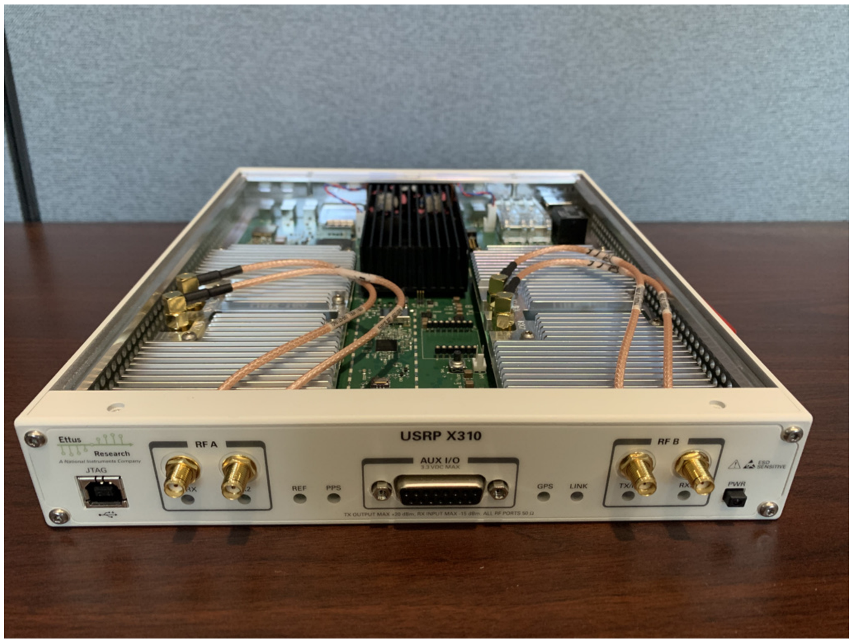
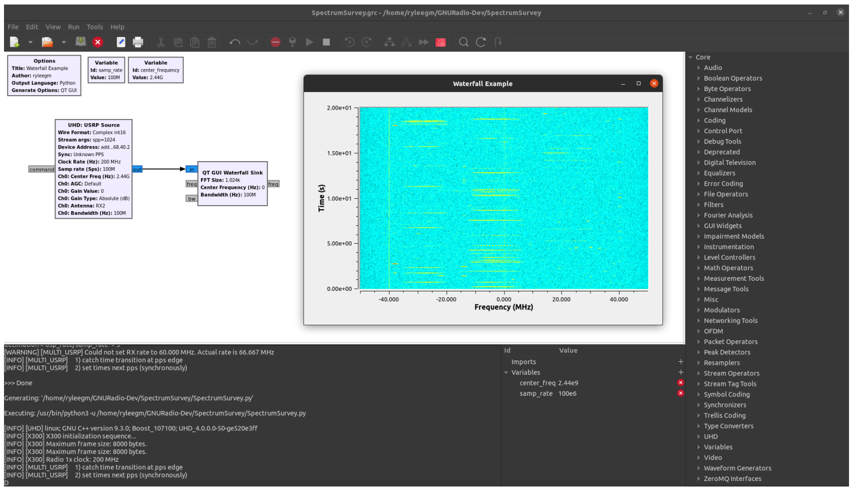


Figure 3.1: X310 pictured with top cover removed to show daughter cards installed.

allows for the transmission of data between the host computer and the radio [18]. UHD supports a C++ and Python API that can be used to create custom scripts. The open-source nature of UHD has seen it also be incorporated and extended into other software packages that can greatly simplify the use of the hardware.

One such open-source project is GNURadio. GNURadio uses drag and drop blocks connected with virtual wires to build 'flowgraphs' that visually model the data processing flow of the program. Figure 3.2 shows a flow graph that streams data received by the radio to the host computer where the host computer then generates a waterfall plot showing the frequency content of the collected data. Although the flowgraph shown is very simple, GNURadio offers a large variety of radio blocks to enable many different types of complex behavior. All of the GNURadio blocks included with the software do the computation on the host PC and send data back and forth to the radio for transmit or receive as needed.

GNURadio is more flexible than just the included blocks, it is a platform that allows for the development of streaming algorithms that can be run on the host



Options
Title: Waterfall Example
Author: ryleegm
Output Language: Python
Generate Options: QT GUI

Variable
Id: samp_rate
Value: 100M

Variable
Id: center_frequency
Value: 2.44G

UHD: USRP Source
Wire Format: Complex int16
Stream args: spp=1024
Device Address: add...68.40.2
Sync: Unknown PPS
Clock Rate (Hz): 200 MHz
Samp rate (Sps): 100M
Ch0: Center Freq (Hz): 2.44G
Ch0: AGC: Default
Ch0: Gain Value: 0
Ch0: Gain Type: Absolute (dB)
Ch0: Antenna: RX2
Ch0: Bandwidth (Hz): 100M

QT GUI Waterfall Sink
FFT Size: 1.024k
Center Frequency (Hz): 0
Bandwidth (Hz): 100M

```
[WARNING] [MULTI_USRP] Could not set RX rate to 60.000 MHz. Actual rate is 66.667 MHz
[INFO] [MULTI_USRP]     1) catch time transition at pps edge
[INFO] [MULTI_USRP]     2) set times next pps (synchronously)

>>> Done

Generating: '/home/ryleegm/GNURadio-Dev/SpectrumSurvey/SpectrumSurvey.py'

Executing: /usr/bin/python3 -u /home/ryleegm/GNURadio-Dev/SpectrumSurvey/SpectrumSurvey.py

[INFO] [UHD] linux; GNU C++ version 9.3.0; Boost_107100; UHD_4.0.0.0-50-ge520e3ff
[INFO] [X300] X300 initialization sequence...
[INFO] [X300] Maximum frame size: 8000 bytes.
[INFO] [X300] Maximum frame size: 8000 bytes.
[INFO] [X300] Radio 1x clock: 200 MHz
[INFO] [MULTI_USRP]     1) catch time transition at pps edge
[INFO] [MULTI_USRP]     2) set times next pps (synchronously)
D
```

Id	Value
Imports	
Variables	
center_freq	2.44e9
samp_rate	100e6

Figure 3.2: GNURadio Companion software shown with a simple, two block flowgraph that takes data streamed from the radio and process it into a waterfall plot.

PC. To enable this development, GNURadio provides tools and tutorials that help streamline the use of the platform. Moreover, a strong user community exists with a large number of out-of-tree modules available to build on the already robust library of preinstalled apps. This helps to bridge the gap between the initial installation and developing the first block.

3.2 RF Network-on-Chip

Some of the Ettus radios, like the X310, have an FPGA inside with free space that can be utilized for hardware-accelerated processing [17]. Offloading computation from the host computer to application-specific FPGA designs not only provides a performance uplift but also removes the latency associated with the data transfer to the host PC. To allow for the development of additional capabilities utilizing the available FPGA space, a common framework is used to interconnect processing blocks and control data flow. RFNoC provides this function with a block-based architecture connected through two main operational planes, the control plane and the data plane [1]. The design steps needed to generate one of these NoC blocks will be presented in Section 3.4. This step-by-step discussion provides information that may not be otherwise available in a single place. The design flow also provides lessons learned to assist future users of the framework and reduce the learning curve for use.

Both planes use the same bus type called the Compressed Hierarchical Datagram for RFNoC (CHDR). This format is packetized with each packet containing a header, optional timestamp, a user-defined number of optional metadata words, and some number of payload words. The width of the busses is flexible at the design stage and can be changed for the entire architecture. This bus width is usually the width of each of the payload words, as the header has a set format width of 64 bits. One of the only constraints enforced with this bus is that at least one payload word must be in every packet.

There are four main types of structures that make up the RFNoC framework: the RFNoC block, the streaming endpoint, the data plane and control plane routing crossbars, and the main CHDR crossbar. An RFNoC block is where the user develops and implements their algorithm. Data is fed into the block from the framework, some operation is carried out on the data, and then that data is transmitted out to the next block.

Each block or sequence of blocks sends and receives data and control packets from the block streaming endpoint. These streaming endpoints are responsible for routing the data to and from the block. Streaming endpoints are the origin of the data for user blocks and the destination point for the output data of the user blocks [1].

Between the RFNoC block and the streaming endpoint is the static data crossbar and the full mesh control crossbar. The control crossbar is mainly tasked with routing control packets from the origin stream endpoint to the appropriate consumer. Since this crossbar is full mesh, however, it also carries control packets from block to block without the need of routing to the stream endpoint.

Data is routed from the stream endpoint to block and back through the static router. Routes through this crossbar are fixed and require a new FPGA image to be synthesized to change. There is no rule about each RFNoC block requiring its own streaming endpoint, in fact, it is possible to link RFNoC blocks directly together across this crossbar.

Static routing is a critical design decision and the desired flexibility of a block connection must be considered when making routing determinations. Data flow between two blocks through static routing is preferred over that of the streaming endpoint as it reduces the FPGA resources and reduces the number of clock cycles between data leaving one block and arriving at another. So, while every block could have its own streaming endpoint for full flexibility, there is an incentive to limit block to block transmission to static connections when appropriate.

A great example of the use of the static crossbar is the radio block and the digital down-conversion block (DDC). These two blocks are always connected and therefore all of the default images have the Radio block and the DDC statically connected. Other blocks, like a hardware FFT, are not always going to be used in the data chain and therefore need to be connected to streaming endpoints. A single streaming endpoint connects to the input and output of its assigned RFNoC block and is connected to the main CHDR crossbar.

The main CHDR crossbar is where all of the dynamic connections take place. A connection between any two arbitrary ports connected to the crossbar can be established as it is a fully meshed structure. Additionally, this main crossbar is full bandwidth and can connect two stream endpoints dynamically without data loss. Importantly, the CHDR crossbar not only connects the stream endpoints but also provides the connections to the interfaces that connect to the host PC. Figure 3.3 shows how these structures are arranged for operation.

3.3 Block Interface

The CHDR bus itself is a lightweight implementation of the AXI bus interface introduced by Xilinx to standardize the connection between Xilinx-provided IP blocks. The RFNoC implementation of this interface utilizes only tdata, tvalid, tready, and tlast signal lines for each the input and the output [1]. This means

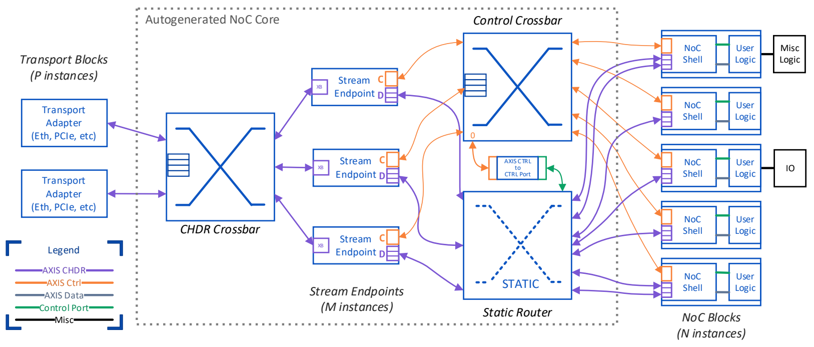


Figure 3.3: An example showing how the internal RFNoC structures may connected for operation [1].

that a framer/de-framer is needed to divide the data packets into their respective header, metadata, and data elements. Fortunately, Ettus provides two framer/de-framer interfaces, AXI-Stream Payload Context and AXI-Stream Data, as part of their development tools.

Since both of these interfaces use the AXI-Stream interface to some extent, a brief overview should be provided before continuing. The AXI-Stream bus is a subset of a larger set of AXI bus protocols [19]. Table 3.1 describes the signals that are used in AXI. Notably, this interface provides a valid signal that is fed into

Signal Name	**Signal Description**
tdata	The payload word for the data stream.
tlast	Asserted on the last payload word of the packet
tvalid	Asserted when the value on tdata is valid.
tready	Asserted when by the recipient to signal readiness.
tuser	Describes the word type for the current word on the bus.

Table 3.1: A description of the signal used in a simple AXI-Stream interface.

the receiving block and a ready signal that comes from the receiving block to the sender. This allows for the blocks to agree on a successful transmission before the transaction is complete. This also makes the logic for advancing the stream incredibly simple. If the valid signal and ready signals are both asserted on a clock cycle then the stream can be advanced. The user signal is not strictly required for the AXI-Implementation but is used in the AXI-Stream Payload Context interface. This signal provides information about the currently presented word on the data line, which is useful if different types of data encodings are used in a single data stream.

The AXI-Data interface is the simpler of the two available interfaces. The interface provides a standard AXI-Stream interface for the payload data of the packet but removes user-facing complexity by presenting the header data as separate signals that are valid throughout the receipt of the data packet. Table 3.2 provides a list of signals that are provided to the block in addition to the previously described AXI-Stream signals that are used for the data stream. Although the AXI-Data inter-

Signal Name	**Signal Description**
ttimestamp	Timestamp associated with the packet.
tlength	Length of the packet in bytes.
teov	Signals the end of a vector.
teob	Signals the end of a burst of associated packets.

Table 3.2: The relevant signals for the AXI-Stream Data user interface.

face does provide an easier interface to use, it does introduce some ambiguity into the timing of the packet transmission introduced by the need to generate its header data and reconstruct a new header if it is changed by the user block.

The other interface option is AXI-Stream Payload Context. This interface provides an AXI interface for data, like the previous option, but it also provides the header, timestamp, and metadata information in an additional AXI stream called the context stream. Because both of these streams are being put onto a single bus by the framer/de-framer logic it is important to serve the context data before the payload data. Both of the streams use the signals as they are described in Table

3.1 to provide the AXI-Streams, except that the user signal is only provided in the context stream as the data stream only has one encoding type. The timing diagram shown in Figure 3.4 shows how the context and payload streams are linked together and the structure that must be maintained when the packet is forwarded or a new packet is created. Although this interface requires more consideration for two separate data streams that are linked in time, it allows a consistent model to be applied to all data and allows metadata to be used if desired. It is for those reasons that the AXI-Stream Payload Context interface was chosen for this work.

The framer/de-framer for these interfaces is located in a module called the NoC shell and can be seen on the architecture graph inside the RFNoC blocks in Figure 3.3. The NoC shell is mostly a black box to the user, except for a first in first out (FIFO) buffer on each of the ports. This buffer allows the user block to stop accepting data, using the tready signal, without dropping packets or packet words. A backend interface is also implemented in the NoC shell as a 512-bit status interface and a separate control interface that is used by the larger framework and is not of use to the user [1].

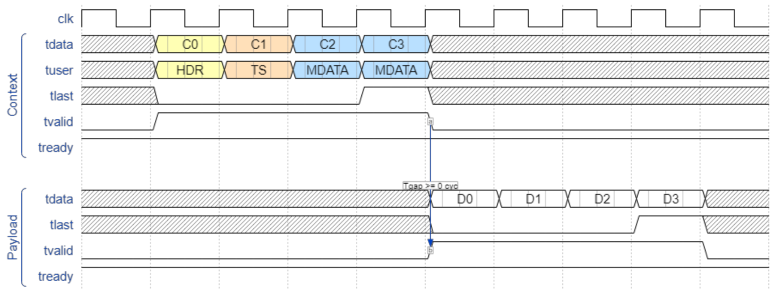


Figure 3.4: The timing diagram for a the AXI-Stream Payload Context interface. It shows a 4 word data packet with timestamp [1].

3.4 Design and Control of a Block

RFNoC provides several tools to try to reduce the overhead required to build a custom FPGA block. One of the ways this reduction is accomplished is by managing abstraction and requiring the user to interact with and manage a bare minimum of files required to successfully implement a design. All development took place on a desktop machine with a Intel®Xeon®processor and 256 gigabytes of ram running Ubuntu 20.04. The process described in this section was tested using RFNoC 4.0 with no out-of-branch patches applied. Of course, the more complex a block is and the more the structure deviates from the default configuration, the more management must take place. This section seeks to describe the design flow of creating a block of similar complexity to those presented later in this work. Numerous topics will be covered in an order that mimics the actual order that a user may take to build a block. The topics discussed include: generating framework HDL using a configuration file, adding additional user registers to the block, changing the number of ports and their identifier, and making changes to the GNURadio block to reflect the new additions of the block.

The development journey begins with the rfnocmodtool, a command-line tool that is used to create RFNoC modules and blocks. Modules contain RFNoC blocks, this hierarchy allows users to group similar blocks together and help manage files when many blocks are in development [20]. A module can be created using the following command:

```
# rfnocmodtool newmod
```

The user will be prompted to enter information about the module that will be used to create a directory with the required folder structure. Navigating inside of the file directory that was created from the newmod command, a user can generate the files necessary to start work on the block.

```
# rfnocmodtool add
```

The add command above prompts the user for information about the block, of which only the name is a required field, the defaults for the remaining options are sufficient for a successful start to the process.

There are seven main files that the user needs to manipulate to make a fully flexible RFNoC block. Table 3.3 lists the files of interest and their location relative to the top level of the module. These files will be used throughout the development flow and their purpose will be discussed as they are used.

The Verilog files rfnoc_block_blockName.v and rfnoc_shell_blockName.v are both generated automatically. rfnoc_block_blockName.v is the file that describes the RFNoC block under development and this is where the HDL for the user's design will go. The rfnoc_shell_blockName.v is instantiated inside the main design and is responsible for the framer/de-framer of the selected interface as discussed in the previous section. The RFNoC shell also implements a backend interface that helps control RFNoC that is completely invisible to the user and should not be modified.

Before jumping into the Verilog files the user should first configure the block and generate new HDL files to meet their needs. Block configuration settings are the

File Name	Location
rfnoc_block_blockName.v	/rfnoc/fpga
noc_shell_blockName.v	/rfnoc/fpga
blockName_x310_rfnoc_image_core.yml	/rfnoc/icores
blockName_block_ctl_impl.cpp	/lib
blockName_impl.cc	/lib
blockName.yml	/rfnoc/blocks
module_blockName.block.yml	/grc

Table 3.3: The relevant files for the RFNoC block development cycle.

responsibility of the blockName.yml file. YAML Ain't Markup Language (YAML), is a human readable format that is used throughout RFNoC and GNURadio to define parameters and describe blocks and is denoted with the .yml file extension [21].

There are four main sections to the blockName.yml file: block information, clocks, control ports, and data ports. Block information should not be changed as it is mostly informative. The clocks section controls the clock frequencies of the various busses, there are two clocks that can be chosen from on the X310: 200 MHz and 184.32 MHz. It is critical to ensure that the CE clock is in the list and that 200 MHz is selected as its clock speed, enabling full speed operation. Setting the clock is as simple as

```
clocks:
  - name: ce
        freq: "[200 MHz]"
```

This work did not utilize the control port as a master port and therefore the default settings of this section were always used.

The final section defines the configuration of the data ports. Multiple named ports can be defined using the following parameters.

```
inputs:
    input1:
      item_width: 32
      nipc: 1
      info_fifo_depth: 32
      context_fifo_depth: 32
      payload_fifo_depth: 32
      format: sc16
      mdata_sig: ~
```

Buffer sizes are mostly up to the user to determine. These FIFOs sit at the port edges and help to prevent overflows if the user block very briefly stops accepting input. Ideally these FIFOs should be about one packet length as data beyond a full packet will be backed up to the streaming endpoints. The same settings apply to the output ports that have identical parameters under an outputs tag instead of an inputs tag.

Once the block YAML is configured to the users desired state, new Verilog files can be generated. Ettus provides a script called rfnoc_create_verilog.py that generates the framework for the configuration defined in blockName.yml. The script is a part of UHD and exits in the UHD install directory under /host/utils/rfnoc_backtool directory. When the user is in the same directory as the script it can be executed from the command line using the following,

```
./rfnoc_create_verilog.py -c path/to/block/yaml
-d path/to/Verilog
```

Now that the two Verilog files have been updated, the user can begin to decide how many user registers they want. Register logic is generated inside of the rfnoc_block_blockName.yml file and a single register is defined by default, providing a template for the addition of more registers. Each register has an address parameter and a reset parameter, with the default address set to zero and a default value of 0. These two parameters should be copied and used for each new register, incrementing the address by four as the register words are passed in as 32-bit integers.

First, the user should instantiate a 32-bit Verilog register for each new RFNoC controlled register that is needed. The next part of the user logic is an *always* block synchronized with the control port clock. Inside this *always* block there are 3 *if* statements that control register functions. The first of these *if* blocks checks the reset line, here the user should add a simple register assignment resetting the Verilog register to the default value parameter

The other two *if* statements check for a register read or a register write. Both of these contain *case* statements using the address to multiplex between the registers. A copy of the default register logic with the appropriate changes to the parameters will successfully set up the hardware of the registers. The Verilog code in listing 3.2 provides this user logic as it can sometimes be absent from generated files in version RFNoC 4.0.

```
localparam REG_USER1_ADDR = 0; // Address for user register
localparam REG_USER1_DEFAULT = 0; // Defult value

reg [31:0] Reg_One_Value = REG_USER_DEFAULT;

always @(posedge ctrlport_clk) begin
 if (ctrlport_rst) begin
   Reg_One_Value = REG_USER_DEFAULT;
 end else begin
   // Default assignment
   m_ctrlport_resp_ack <= 0;

   // Read user register
   if (m_ctrlport_req_rd) begin // Read request
     case (m_ctrlport_req_addr)
       REG_USER_ADDR: begin
          m_ctrlport_resp_ack <= 1;
          m_ctrlport_resp_data <= Reg_One_Value;
       end
     endcase
   end

   // Write user register
   if (m_ctrlport_req_wr) begin // Write requst
     case (m_ctrlport_req_addr)
       REG_USER_ADDR: begin
         m_ctrlport_resp_ack <= 1;
         Reg_One_Value     <= m_ctrlport_req_data[31:0];
       end
     endcase
   end
 end
```

```
end
```
Listing 3.1: The default register logic described above. Provided as it is not always correctly generated in version 4.0.

Immediately below the register logic, the port signals are set to default values. The user can replace these default connections and insert their well tested and simulated Verilog that they would like to operate on their data. User logic is the last part of the file and concludes the work that needs to be done in the Verilog files.

Next, the software interface must be informed of the changes to the port configuration and the additional user registers. blockName_block_ctl_impl.cpp is where the the API goes to generate the control signals necessary to configure the block when the block constructor is called by the user script or GNURadio. First, the user needs to create new uint32_t constants for the registers that were just created in the Verilog. A call to the register_property function should be made inside of the private _register_props function with a closure to provide the functional information. After the property is registered, a custom property type needs to be instantiated at the bottom of the file, this is the property that is passed into the register_property. The constant declaration, property call, and property type should look similar to:

```
    const uint32_t Name_block_ctl::REG_USER1_ADDR = 0;
    const uint32_t Name_block_ctl::REG_USER1_DEFAULT = 0;

    void _register_props() {
  register_property(&_user_reg, [this]() {
      int user_reg = this->_user_reg.get()
         this->regs().poke32(REG_USER_ADDR, user_reg);
  });
     ...
     ...
    }
property_t<int> _user_reg{"user_reg", REG_USER1_DEFAULT,
    {res_source_info::USER}}
```

Similar to the process to create a register property, a block edge property is required for each of the ports that are used on the block. Just like with the register, constants need to be defined for each of the ports, each property must be registered and a property type must be instantiated. Unlike the register, the constants should only be incremented by 1 and the input and output edges are indexed separately so there should be an output zero and an input zero. A further difference from the register is an additional property resolver that helps the software to understand what data types should be sent in and out of the block. This should match the value that was used in the port definition of the block YAML, but there is not a check to make sure that the values are consistent.

Below is a sample snapshot of code that helps define the necessary statements that are required to successfully set up the hardware:

```
constexpr uint32_t input1_port = 0;
constexpr uint32_t output1_port = 0;

void _register_props() {
    ...
    register_property(&_type_in_input1);
    register_property(&_type_out_output1)
    ...
    add_property_resolver({&_type_in_input1},
        {&_type_in_input1}, [this](){
        _type_in_input1.set(IO_TYPE_SC16)
    });
   add_property_resolver({&_type_out_output1},
        {&_type_out_output1}, [this](){
        _type_out_output1.set(IO_TYPE_SC16)
    });
...
}
...
property_t<std::string> _type_in_input1 =
    property_t<std:string>{ PROP_KEY_TYPE,
      IO_TYPE_SC16, {res_source_info::INPUT_EDGE,
          input1_port}};
property_t<std::string> _type_out_output1 =
    property_t<std:string>{PROP_KEY_TYPE,
        IO_TYPE_SC16, {res_source_info::OUTPUT_EDGE,
           output1_port}};
```

If the user is planning on using the block with GNURadio then there is one more step before moving on to the final image synthesis tasks. The graphical representation of the block that is used for drag and drop connection of the block needs to know what the port configuration is and what each of the register values should be. All of this configuration is done in the module_blockName.block.yml file, utilizing three main sections.

The first section is the template section, this provides the import argument and the constructor parameters so that GNURadio can correctly setup the auto generated script. Of these parameters only the callbacks need to be manipulated. These callbacks connect through to the previous properties that were instantiated to manage the control read and writes to the registers in the blocks. One call back is needed for each of the registers that are used and it is important that the first argument in the callback is the same as the one provided in the last property object in the implementation file.

The second section, parameters, is all of the fields within the GNURadio block. Register fields for the user to input values into the GUI block are created here. It is critical that the ID of this value is the same as the one given to the callback in the previous section of this file. The label and default value can be anything the user wants to have visible in the GUI. It is a good idea to have the block defaults in this file be the same as the default parameters previously set but, again, this is not a requirement.

The last section of this file is used to create the actual ports on the GUI block. Each port has three parameters: domain, label and dtype. For the purpose of this work the domain will always be rfnoc and the dtype will be sc16, the label can be whatever the user wants to display. The port assignment between these definitions and the properties defined in the previous file are based on the port number assigned previously, meaning that the first import port reference in the .block.yml file corresponds to the zeroth port in the property definition.

An example of each of the necessary fields are provided below for illustration.

```
callbacks:
- set_int_property('user_reg', $(user_reg))
...
parameters:
- id: user_reg
  label: User Register Name
  dtype: int
  default: 0
...
inputs:
- domain: rfnoc
  label: Input Port Name
  dtype: 'sc16'
outputs:
- domain: rfnoc
  label: Output Port Name
  dtype: 'sc16'
```

With all of the initial block setup and design complete the user can define the blocks that are needed in the FPGA image and how they are connected through the various crossbars. Settings for the layout of the blocks are handled in the blockName_rfnoc_image_core.yml file. This file is structured into four section: streaming endpoint definitions, block definitions, block connections, and clock assignment.

First the streaming endpoints need to be defined, each have three block parameters: ctrl, data and buff_size. The ctrl and data parameters determine whether or not traffic from that plane flows through the endpoint. buff_size determines the size of the buffer that is used to prevent dropped packets if the recipient of the stream halts operation momentarily. The parameters are formated as follows

```
stream_endpoints:
  ep0: #Label can be anything
    ctrl: False
    data: True
    buff_size: 32768
```

Block definitions are next with each having a different set of parameters. The only common parameter is the block_desc, this is the name of the blockName.yml file and should be formatted as seen below.

```
noc_blocks:
        blockName0: #Label can be anything
          block_desc: 'blockName.yml'
```

Since the block and module were created by the modtool then the software knows the directory structure and where to find these files.

Next, connections must be defined for each of the blocks. Decisions about the routing through an endpoint or through the static crossbar only are made at this point. Two different connection examples are given below. The first provides a connection from the ep0 as the source of the data to the input of duc0 (Digital Up Converter) block, then it is routed from the duc0 output to the input of the radio0 block. This is an example of a static connection between two block using only the static crossbar between the DUC and radio, this forms the transmit chain to the radio hardware.

The second example shows a block connected with the streaming endpoint on each end. It is important to note that each endpoint should only be used for a single pair of signals. If the user's block has an odd number of ports or multiple pairs of ports, multiple streaming endpoints are necessary.

```
connections:
  - {srcblk: ep0, srcport: out0, dstblk: duc0, dstport:
     in_0 }
  - {srcblk: suc0, srcport: out_0 dstblk: radio0, dstport:
     in_0 }
```

The final piece of this file defines the clock connections to the block. Clocking is a critical piece for ensuring that the operation of all of the blocks goes smoothly. For this work all of the clocks were defined using the compute engine (CE) clock except for radio blocks which are connected to the radio clock. An example of both types of clocking connections are shown as,

```
clk_domains:
   - { srcblk: _device_, srcport: radio, dstblk: radio0,
       dstport: radio }
   - { srcblk: _device_, srcport: ce, dstblk: ddc0,
       dstport: ce }
```

Finally, after all of these files have been updated to set up the configuration of the block, the module can be built and the FPGA image synthesized. To start the build and synthesis process the user needs to navigate, within the terminal, to the module build directory. Once in the directory, the make files for the install need to be created using

```
# sudo cmake -DUHD_FPGA_DIR=./path/to/uhd/fpga/dir
```

After this process completes the make files can be used to install the module into GNURadio with the command,

```
# sudo make install
```

At this point the block should be visible inside of GNURadio at the bottom of the module tree. The make files are now ready for synthesis to the FPGA. Synthesis requires that the Xilinx Vivado is already installed with version 2019.1 being the tested version. The Vivado build is initiated by executing,

```
# make blockName_rfnoc_image_core.yml
```

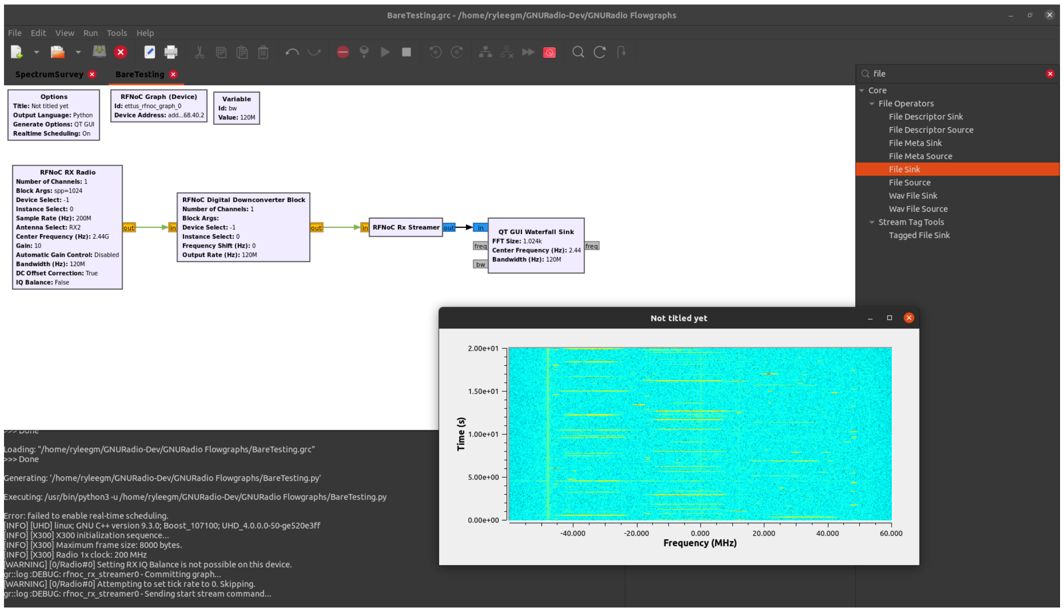
in the build directory.

The build process can take a significant amount of time, throughout this work 45 minutes was found to be very common but times of up to four or five hours were also encountered depending on the content of the user logic. Once the build is complete a .bit file is generated in the uhd/rfnoc/top/usrp3/x300/build directory. The .bit file is used to program the FPGA using the uhd image loader tool. The image loader was the most successful when using the following argument configurations:

```
# uhd\_image\_loader --args "addr:Radio.IP.Address"
     --fpga\_path ./path/to/.bit
```

After the image loader has completed its operation, the user can power cycle the radio and test their new hardware block using the UHD API or GNURadio. Figure 3.5 shows the flowgraph enabling the waterfall display implemented with RFNoC blocks.

It is important to note that when using any RFNoC blocks the entire radio must be built using the RFNoC block variants. Using the RFNoC components means that instead of using the USRP source block that is packaged in GNURadio a user must instead use the RFNoC radio block. Because the RFNoC radio block is statically connected to the Digital Down Converter (DDC) it also must be inserted into the flowgraph. Having to model the receive chain using NoC blocks instead of the provided GNURadio source can present additional concerns. However, this is still simpler than the alternative of scripting from the UHD API directly, which has to instantiate the radio chain individually whether custom RFNoC blocks are used or not.



Options
Title: Not titled yet
Output Language: Python
Generate Options: QT GUI
Realtime Scheduling: On

RFNoC Graph (Device)
Id: ettus_rfnoc_graph_0
Device Address: add...68.40.2

Variable
Id: bw
Value: 120M

RFNoC RX Radio
Number of Channels: 1
Block Args: spp=1024
Device Select: -1
Instance Select: 0
Sample Rate (Hz): 200M
Antenna Select: RX2
Center Frequency (Hz): 2.44G
Gain: 10
Automatic Gain Control: Disabled
Bandwidth (Hz): 120M
DC Offset Correction: True
IQ Balance: False

RFNoC Digital Downconverter Block
Number of Channels: 1
Block Args:
Device Select: -1
Instance Select: 0
Frequency Shift (Hz): 0
Output Rate (Hz): 120M

RFNoC Rx Streamer

QT GUI Waterfall Sink
FFT Size: 1.024k
Center Frequency (Hz): 2.44
Bandwidth (Hz): 120M

```
>>> Done

Loading: "/home/ryleegm/GNURadio-Dev/GNURadio Flowgraphs/BareTesting.grc"
>>> Done

Generating: '/home/ryleegm/GNURadio-Dev/GNURadio Flowgraphs/BareTesting.py'

Executing: /usr/bin/python3 -u /home/ryleegm/GNURadio-Dev/GNURadio Flowgraphs/BareTesting.py

Error: failed to enable real-time scheduling.
[INFO] [UHD] linux; GNU C++ version 9.3.0; Boost_107100; UHD_4.0.0.0-50-ge520e3ff
[INFO] [X300] X300 initialization sequence...
[INFO] [X300] Maximum frame size: 8000 bytes.
[INFO] [X300] Radio 1x clock: 200 MHz
[WARNING] [0/Radio#0] Setting RX IQ Balance is not possible on this device.
gr::log :DEBUG: rfnoc_rx_streamer0 - Committing graph...
[WARNING] [0/Radio#0] Attempting to set tick rate to 0. Skipping.
gr::log :DEBUG: rfnoc_rx_streamer0 - Sending start stream command...
```

Figure 3.5: An example showing how the provided radio and DUC blocks can be used to replace the USRP Source block used previously [1].

Chapter 4

Simple FSS Architecture

The FSS algorithm discussed in section 2.1 was implemented on the FPGA inside the X310 radio. By moving the algorithm to the FPGA inside the radio the performance can be maximized by optimizing a section of the FPGA hardware just for this algorithm. Additionally, if the waveform selection and generation algorithm is deployed to the FPGA then the host PC is not in the critical path of the perception-action cycle. Not requiring the host PC CPU or GPU for calculations would eliminate the latency of the Ethernet transfer.

4.1 Changes to the Algorithm

Some adjustments must be made to the greedy FSS algorithm to take advantage of the hardware implementation. The first major step in this endeavor is to remove the reiterative search that merges closely spaced high power bands. Optimizing for the desired output accomplishes this goal. The algorithm as discussed in 2.1 not only wants to make the selection of the largest available space but also wants to have the power band groupings if additional multi-objective optimization is desired [6].

The block developed in this work focuses solely on finding the largest continuous frequency bins with no signal present as defined by the threshold. Therefore, the merging parameter, previously used to determine the maximum spacing between high power bands before they are no longer eligible for merging, can be used as the minimum usable spectrum size. Consequently, if the largest continuous bins of the available spectrum are less than the merge parameter then the algorithm would determine no spectrum to be available for secondary use. This leaves the areas of the data frame that are small enough to be merged discarded, effectively eliminating the need to reiteratively check for these merging cases.

4.2 Theory of Operation

With the reiterative component of the algorithm removed, a discussion about the hardware operation can be presented. This block must scan through the data stream as it arrives and classify each element as being high power or low power based on the threshold. The payload AXI-Stream that feeds data into the block provides a single bin value per clock cycle. Therefore a simple comparison can be carried out on each element to determine if it is low power or not.

If a bin is the first low-power signal to be encountered, then a 'bucket', with start and size fields, should be initialized with a start point of the current index and a size of one. For each consecutive low power sample after a bucket has been initialized the size of the existing bucket should be incremented. Once a high-power bin is encountered the bucket that was being used should be frozen.

This bucket initialization could lead to a very large number of buckets being instantiated, which is unnecessary since the only bucket of interest is the largest one. This means that only two buckets should ever be needed to retrieve the desired data. Instead of initializing a new bucket when a new section of the available spectrum is encountered then the smallest of the two buckets should be selected to be overwritten. This is a simple comparison between the size parameter of the buckets. Selecting the bucket with the smallest size to be the new write space preserves the data of the currently longest sample set.

Once the largest bucket is found, it needs to be sent out of the block. There are a few options available. The first option is to add the start point and size to the metadata at the front of the packet. This was quickly ruled out as it would require the entire packet to be cached until after a decision had been made. Ideally, the block would allow the data to stream through such that there was no interruption to the stream. That means that the block will need a secondary port to move the data along.

Secondly, the data could be sent across the control infrastructure to the next block. This would allow the FSS block to utilize a single stream endpoint. Unfortunately, this would prevent the data from being streamed directly to the host PC without an intermediary block to convert the control data to a standard data stream. Although the block data is intended to be consumed by another block in hardware without traveling to the PC, for this work it is necessary to offload the data to the computer for verification and analysis. This meant that the block would need to employ a secondary data output stream to enable the full flexibility that is needed. This comes with the additional overhead of a secondary stream endpoint to dynamically connect the secondary data output to other blocks of interest.

4.3 Implementation

With the theory of operation now solidified, the hardware elements that make the operation possible should be discussed. The state machine that powers the logic will also be shown before the section closes with a discussion of the send state machine.

4.3.1 Data and Registers

The two buckets need to have each of their fields maintained as each frame of data is processed. The size of these registers should be determined by the length of the packet, as each of the registers should be able to store this value. The block designed for this work is fed by a hardware FFT block with an FFT length of 1024 setting the packet size. 1024 was selected as the FFT length because 1024 is approaching the largest power of two of samples that will comfortably fit in a jumbo Ethernet frame. The default transporter block puts each CHDR packet in its own Ethernet frames and cannot split large CHDR packets between multiple Ethernet frames. This means that our bucket sizes are stored in 11-bit registers so that they could, if necessary, store 1024 as the possible maximum size of available spectrum in a frame. The start point fields are stored in 10-bit registers so that that the index packet word index from 0 to 1023 can be stored.

The packet words that are passed into the block are not tagged with their number in the packet. That means that an 11 bit counter should also be kept. This packet counter allows the block to keep track of the word index in the packet. This will be used to generate the index value that will be stored in the start field of a bucket when a low power bin is detected.

A final set of buckets is needed to allow for seamless operation. A send bucket is maintained and on the last sample of a packet, the data associated with the largest bucket is transferred to this set of registers. This allows all of the fields associated with the FSS processing state machine to be reset on this last sample. Of course, it is important to consider the effect that the last sample has on the data. This means that if the last bin is below the threshold then the largest bucket comparison should be made with this change considered. Similarly, if the value belongs in the bucket that is being selected, then the value of the size of the send bucket should be incremented with the transfer.

4.3.2 FSS State Machine

With the variables and structures laid out, the FSS state machine can be presented. Figure 4.1 shows the FSS state machine with the transition conditions and transition actions listed.

Consider the stream of samples shown in Table 4.1 as a streaming input with a packet size of 10 samples with a threshold of one. On startup, the block is in the

Sample Number	0	1	2	3	4	5	6	7	8	9
Value	1	0	0	1	1	1	0	0	0	0

Table 4.1: Values used for an example run of the FSS state machine.

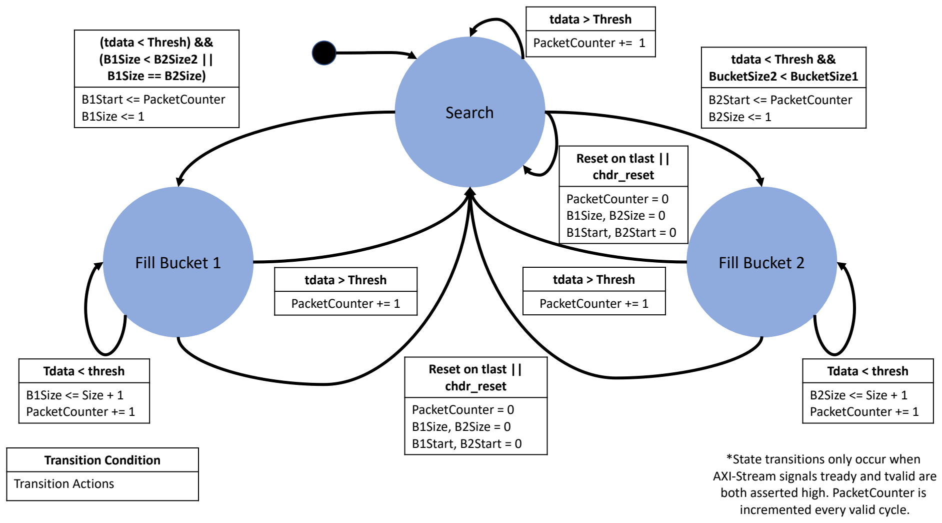


Figure 4.1: The basic FSS state machine shown with transition conditions and transition actions.

searching state. When tvalid and tready are both asserted then the value on the data bus is valid and the stream will advance on the next rising edge of the clock. The first sample in the example stream is a 1, since this is greater than or equal to the threshold the state machine stays in the searching state. The second sample is the first sample in the stream that is below the threshold, this triggers a state transition. Since both buckets have a size of zero, a transition to the Fill Bucket 1 state is chosen. In addition to this, the size value and start value of bucket 1 is set to one since this is sample one.

It should be noted that the Packet Counter is incremented every valid transition that is not a tlast transition where the counter is reset. With that in mind, the third sample is reviewed next. This value is also below the threshold so the state is maintained and the Bucket 1 size is incremented. Sample number three is above the threshold, so the state is set back to searching.

Like sample number three, samples four and five are above the threshold. This means the state stays set to searching and the only action is the incrementing of Packet Counter. When sample six comes through less than the threshold, a comparison of the bucket sizes is carried out. This finds the zero in the bucket 2 fields to be less than the two in bucket 1 and the state is set to the Fill Bucket 2 State. The transition actions are also done, which sets bucket 2's size to 1 and the value to 6 from the Packet Counter.

Samples seven and eight are also above the threshold, leaving the state as Fill Bucket 1 for both. This brings us to the final sample. Because the value is below the threshold, it must be considered in the decision on which bucket to store in the send bucket fields. The comparison is between the bucket sizes to determine which bucket is larger. The current state is Fill Bucket 2 so the comparison is carried out between the bucket one register and the bucket two registers plus one. A transfer of the bucket two data is triggered as its size plus one is larger. The state machine then resets its packet counter and the bucket values before asserting the send flag and returning to the Search state. The state machine is now ready to process the next packet.

4.3.3 Send State Machine

When the send flag is asserted on the tlast of an input packet, the send state machine is triggered to run. Figure 4.2 shows the state machine diagram for sending a packet on the secondary output. The state machine works by faithfully implementing the AXI-Stream protocol across both the context and payload streams. For the first state, a header must be generated for the output packet. The fields necessary are shown in Table 4.2 with the values used to send the results packet. The most important value is the length. The value of length is calculated in *bytes*, not in words, and also includes any header, timestamp. or metadata information. For the packet that is being sent, a single 64-bit header word is used with a single 32-bit data word for a total of 12 bytes of data being transferred.

Once the header is constructed and put on the data lines, tvalid is asserted in

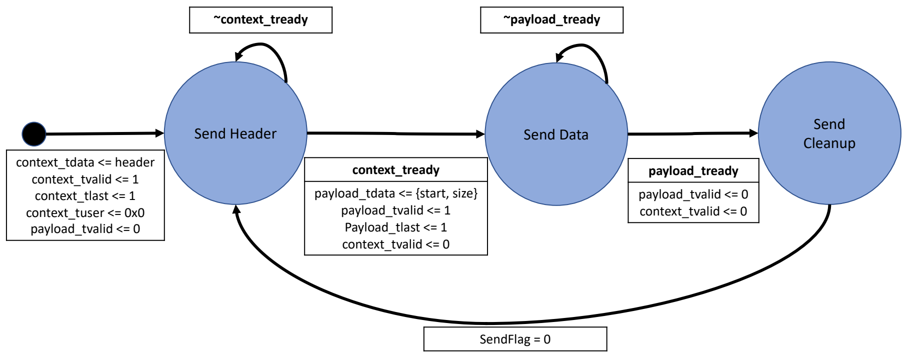


Figure 4.2: The state machine used for transmitting the a packet of data out of an AXI-Stream port on an RFNoC block.

Field	Bits	Description	Use
Virtual Channel	6	Used for routing the packet. Utilized by framework	Field overwritten by framer
EoB/EoV	2	End of burst and end of vector are fields left to the user.	These are unused in this work
Packet Type	3	This is used to signify what the packet is so that the number of header words is known.	0x6 Data Packet, No timestamp
Number Metadata	5	Number of metadata words	0. No metadata used.
Sequence Number	16	A sequencer so that the framework can detect dropped packets.	Field overwritten by framer
Length	16	Length of the packet in bytes, including the header.	12 Bytes for this packet.
DstEPID	16	Used by the framework to route to the correct streaming endpoint.	Field overwritten by framer

Table 4.2: Values used for an example run of the FSS state machine.

the same clock cycle. From here the state is maintained until the context tvalid line coming in from the receiving block is asserted, at which point the state is advanced to the send data state. The send data state de-asserts the context tvalid signal and constructs the data word by placing the 11 size bits on the lower eleven bits, while the 10 start index bits are assigned to bits 16 through 25. Once these are on the payload data bus the payload tvalid is asserted in the same clock cycle.

Finally, after the payload, the tready line is asserted high then the state machine advances to the cleanup state. The cleanup state de-asserts all of the valid lines on the port and sets the state to send header before setting the send flag to zero. After the cleanup state is finished the state machine is back in its starting configuration and waits for the FSS state machine to set the send flag back high.

4.4 Timing and Performance

It is important to set a uniform way to measure performance such that performance across different implementations can be assessed and compared. Creating a metric for the designs in this work is even more challenging, as much of the design exists as a module within a black box. Therefore, the metric must be defined for only the logic that is managed by the user block itself.

Two metrics are developed for the user blocks in this work. The first describes the streaming efficiency of the block and is the number of clock cycles between the receipt of the final word in the packet and the transmission of the final word in the packet. This data latency metric helps classify the efficiency of data management for blocks that require any data retention or data stream halts. When a result is produced based on the data in a packet, it is useful to know how many cycles after the last word of a packet is received does the resultant data product get transmitted. This is the measure of the second metric that this work will refer to as product latency.

The first metric is easy to determine. The design goal of this block was to implement the algorithm without interrupting the AXI-Stream flow through the block. This was accomplished as the input ports and output AXI ports are wired directly together. This means that any stoppage of the stream comes from the block connected to the output of the FSS block. So the last sample is registered into the output on the clock cycle after it is registered onto the wire running through the block. This is the absolute minimum data latency possible for an RFNoC block other than not being present in the stream chain at all.

To understand the result product latency, a timing diagram showing the end of the packet must be examined. Figure 4.3 shows a timing diagram at the end of the packet where the send flag is triggered by the tlast signal. Here the send flag is asserted by the tlast signal during that same cycle that the copy of one FSS bucket into the send bucket is carried out. On the clock after the tlast, the send bucket is loaded, the FSS buckets are clear and a valid marked header is on the context bus. Two clocks after tlast, assuming the consumer of the result is ready, the header is removed from the context bus and the data is on the payload bus. This means that the data is consumed by the output block on the rising edge of the third clock after tlast is asserted. This means that the product latency is three clock cycles. On the Ettus X310, the FPGA operates at 200 MHz, giving a total time of 15 nanoseconds between the receipt of the last data word and the output consuming the result.

4.5 Utilization

FPGAs are made up of lookup tables (LUTs) that are used to implement digital logic, block ram tiles (BRAM), different types of registers, and various specialty components such as DSP blocks and input/output (I/O) structures. While LUTs,

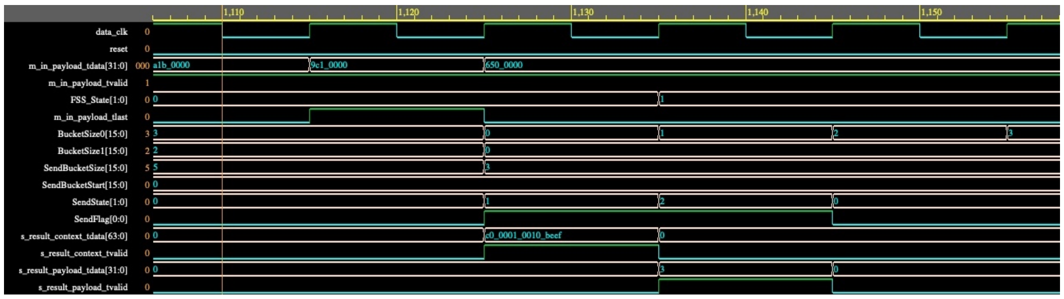


Figure 4.3: The state machine used for transmitting the a packet of data out of an AXI-Stream port on an RFNoC block.

registers, and BRAM exist in seemingly large quantities in the FPGA, they are finite resources and therefore it is important to be understand the utilization of each block to ensure resources aren't needlessly wasted. It is for this reason that the total utilization of the design is presented here. Table 4.3 shows the part utilization in both quantity and as a percentage of the total available on the specific FPGA in the X310. Here we show that the two streaming endpoints that are required for a block

	Default Image		Default W/ Streamers		Full FSS	
Resource	Used	%	Used	%	Used	%
LUTs	133,282	52.43	140,371	55.21	146,693	57.71
Registers	204,487	40.22	216,068	42.50	224,988	44.25
BRAM Tiles	400.5	50.38	408.5	51.38	412.5	51.89

Table 4.3: Kintex®7-410T utilization of the FSS design shown in 3 stages: default image with FFT, default image with stream endpoints for FSS, and image with FSS block. This data was provided by the Xilinx Vivado synthesis tools when the RFNoC images were built.

with two output ports require more resources than the FSS block implementation itself. The block RAM (BRAM) used by the design is used exclusively in the NoC core, discussed previously, that parses the raw CHDR bus into the specific AXI-Stream interface+. In total, the design itself takes up 2.5 % of lookup tables (LUTs) and just under 2% of the Registers of the FPGA. When the required RFNoC infrastructure is considered with the design, the total is about 5.3% LUT use and just over 4% use of the registers. With no DSPs or other limited complex blocks consumed by the design, it is very small and leaves plenty of resources for the implementation of other processing hardware blocks.

4.6 Usage and Verification

Implementation within the RFNoC framework, in tandem with the gr-ettus GNURadio extension, allowed the hardware block to be controlled using GNURadio flowgraphs. Figure 4.4 shows the block generated for use within GNURadio companion flowgraphs. This block contains a user register field that allows a user to select and change the threshold being used by the FSS block at any time. This user value must be an integer, however, as the RFNoC block uses fixed point integers and not floating-point values.

To verify the operation of this block, the flowgraph in Figure 4.5 was used for testing. This graph sources data from the radio frontend and sends it through a hardware FFT before sending it into the FSS block. From here the flowgraph takes the pass-through data from the throughput port and the results data from the results port and stores them in their own files. This data is was then taken into MATLAB where the simulated FSS algorithm was run on it and verified against the results from the block. This method was able to verify parity between the operation of the hardware implementation and the MATLAB script and also show the block was

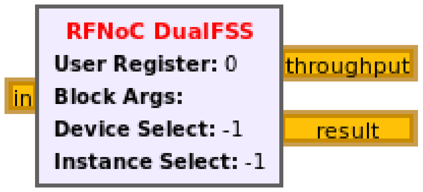
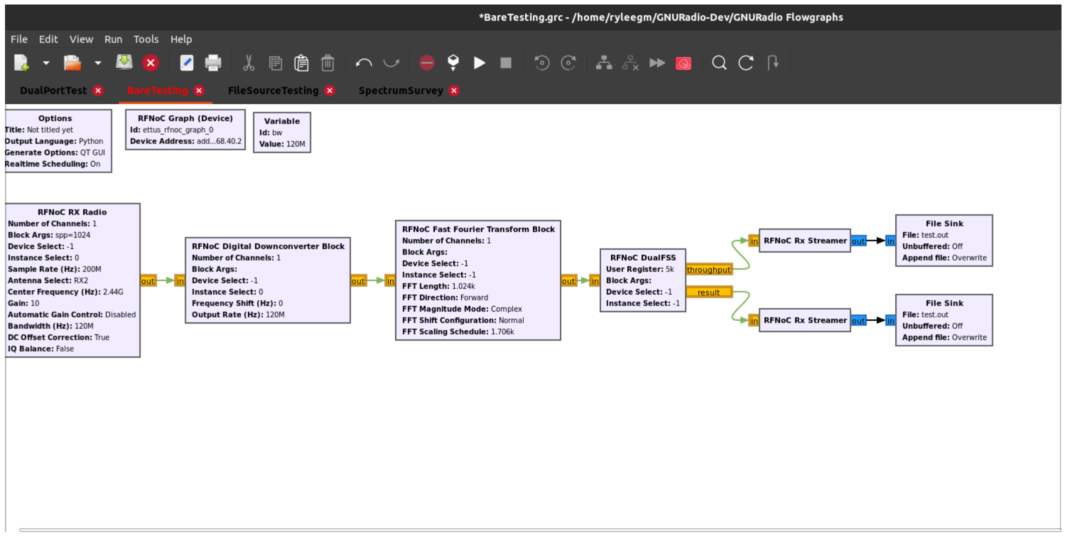

RFNoC DualFSS
User Register: 0
Block Args:
Device Select: -1
Instance Select: -1

Figure 4.4: GNURadio block for the RFNoC FSS block.



Options
Title: Not titled yet
Output Language: Python
Generate Options: QT GUI
Realtime Scheduling: On

RFNoC Graph (Device)
Id: ettus_rfnoc_graph_0
Device Address: add...68.40.2

Variable
Id: bw
Value: 120M

RFNoC RX Radio
Number of Channels: 1
Block Args: spp=1024
Device Select: -1
Instance Select: 0
Sample Rate (Hz): 200M
Antenna Select: RX2
Center Frequency (Hz): 2.44G
Gain: 10
Automatic Gain Control: Disabled
Bandwidth (Hz): 120M
DC Offset Correction: True
IQ Balance: False

RFNoC Digital Downconverter Block
Number of Channels: 1
Block Args:
Device Select: -1
Instance Select: 0
Frequency Shift (Hz): 0
Output Rate (Hz): 120M

RFNoC Fast Fourier Transform Block
Number of Channels: 1
Block Args:
Device Select: -1
Instance Select: -1
FFT Length: 1.024k
FFT Direction: Forward
FFT Magnitude Mode: Complex
FFT Shift Configuration: Normal
FFT Scaling Schedule: 1.706k

RFNoC DualFSS
User Register: 5k
Block Args:
Device Select: -1
Instance Select: -1

RFNoC Rx Streamer

File Sink
File: test.out
Unbuffered: Off
Append file: Overwrite

RFNoC Rx Streamer

File Sink
File: test.out
Unbuffered: Off
Append file: Overwrite

Figure 4.5: The state machine used for transmitting the a packet of data out of an AXI-Stream port on an RFNoC block.

able to operate at the full 160 MHz bandwidth of the daughter cards in the FPGA. Additionally, the block was able to cope with a full 200 MS/s streaming rate of the radio without dropping or needlessly caching packets which would be required if the processing time of the block is greater than the packet generation time.

Chapter 5

Hardware Optimized Cell Averaging Estimator

The static threshold method is effective if the electromagnetic environment is known before the device begins operation. Unfortunately, the noise and interference in a given operating environment can change dramatically over time rendering the threshold method inadequate if set statically. This work then sought a method that would enable dynamic threshold selection by estimating the noise and interference environment of each data frame. As discussed in section 2.2, many methods for estimating the noise and detecting signals in the spectrum exist.

Because of the timing goals of this project, a computationally efficient method was sought to solve this problem. For this reason, CFAR detection was the main method of detection investigated, specifically CA-CFAR and OS-CFAR were of particular interest. Here the application of those methods on real and simulated data is discussed as a motivation for a new CFAR inspired detection method.

5.1 Data Sets

The performance of the baseline CFAR detection algorithms will be applied to real-world data and a set of simulated data. This section will discuss what those data sets look like and what signals they contain. A discussion on the data collection and signal generation methods will be presented.

5.1.1 Simulated Data

The simulated data was generated by a modified version of the 5G downlink script provided as part of the 5G toolbox in MATLAB [22]. The data generation script employed the nrWaveformGenerator function and the data sources for all carrier parameters were randomized. By randomizing the data source signals will have the same envelope but the amplitude within the envelope and the sidelobes will vary with each iteration. 614,400 samples are generated with a sample rate of 61.44 MS/s with added noise to yield an SNR of 5 dB. A spectrogram is formed from the data by dividing it into non-overlapping frames of size 1024 for which the FFT is applied. Figure 5.1 shows the 600 frames that result from this operation while Figure 5.2 shows the first cut frequency cut of spectrogram..

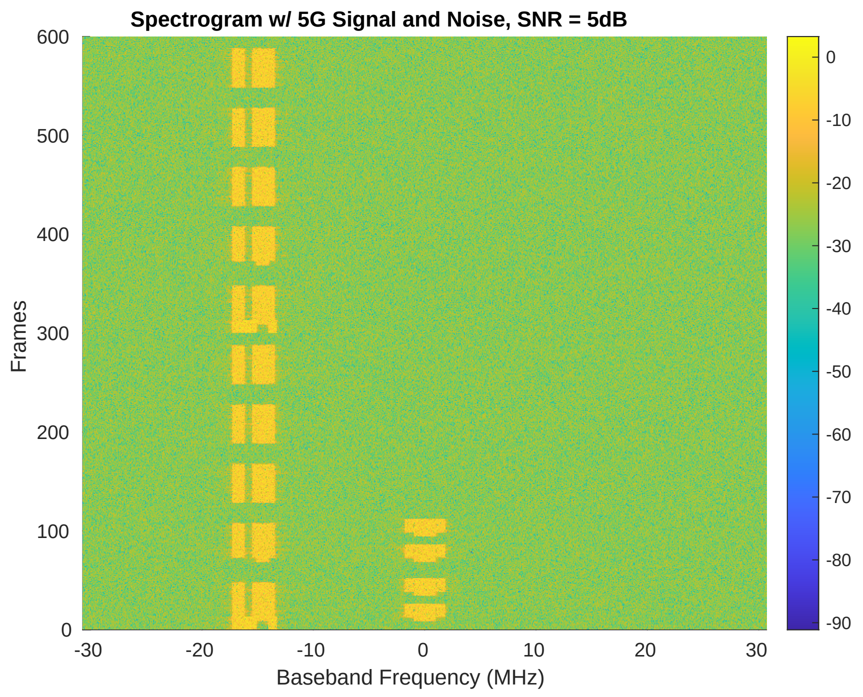


Figure 5.1: Simulated 5G NR Downlink signal spectrogram with 5 dB SNR.

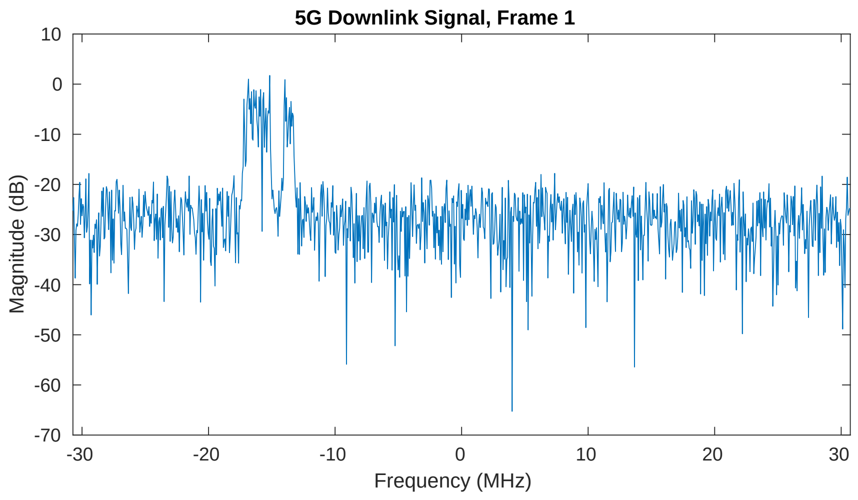


Figure 5.2: Frame 1 of the of the 5G Downlink spectrogram with 5 dB SNR.

5.1.2 Real World Data

Real-world data was captured on the Ettus X310 radio in the Radar Innovations Lab at the University of Oklahoma. The center frequency was set to 2.44 GHz to utilize the operating frequency of the VERT2450 antenna. Complex data was captured at 100 MS/s providing 100 MHz of bandwidth. Data was stored in pairs of 16-bit integers to help reduce file size and when processing the magnitude is taken to give a 32-bit value. These values are then divided by 2^{31} to normalize these values from 0 to 1.

Figure 5.3 shows the spectrogram of the real-world data with three distinct regions of interest highlighted. Note that there is a constant harmonic that exists at about -40 MHz. This is believed to be an artifact from the internal amplifier of the radio as it does not appear if the amplifier is disabled. Similarly, when the amplifier is on the artifact persists even if the RF input is connected to a 50-ohm load. Figure 5.4 shows the first frame of the first area of interest. Here three signals of interest are shown. Signal A is the artifact which will be ignored for most of this

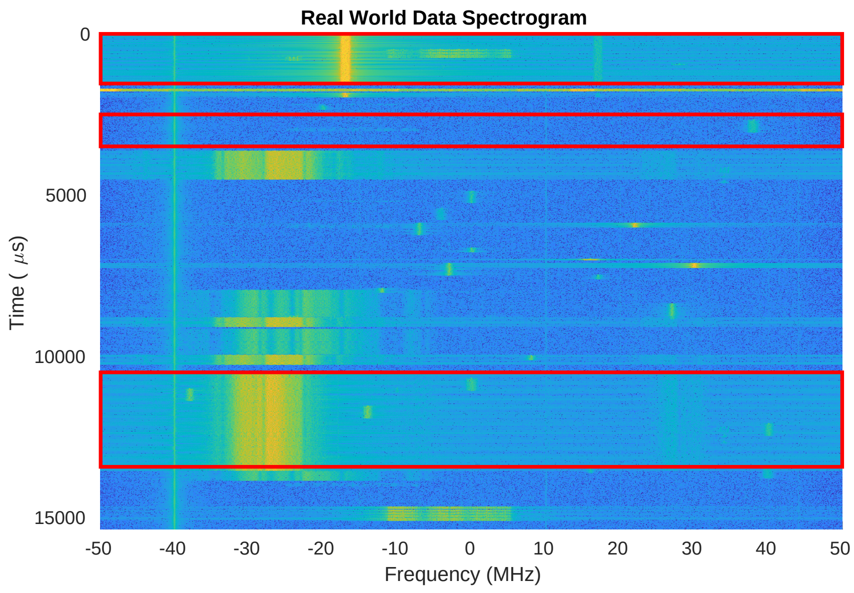


Figure 5.3: Real world spectrogram with 100 MHz of bandwidth. Three distinct regions of interest are highlighted to capture four different signals as well as a noise only case.

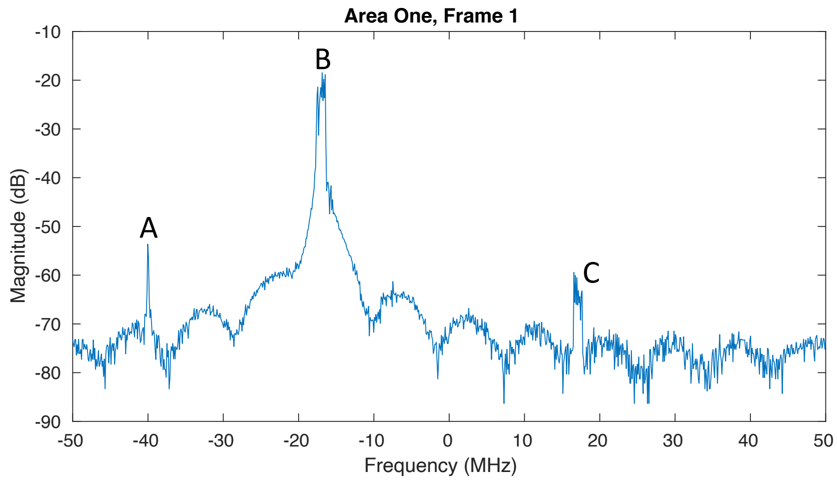


Figure 5.4: Area of interest one with amplifier artifact and two distinct signals.

section. The signal marked B represents a fairly high-energy signal whose sidelobe structure is visible above the noise floor. The strong signal provides a challenging detection environment for signal C as any algorithm with a frame static threshold must be robust to the roll-off of B.

Figure 5.5 presents the first frame of the second area of interest. Area two presents as close to a noise-only area as was observed in the collection from the real-world environment. This provides a fantastic baseline for each of the algorithms to be tested against.

Finally, Figure 5.6 gives the first frame of the third area of interest. Here the widest band signal in the data set is present at relatively high signal power. This

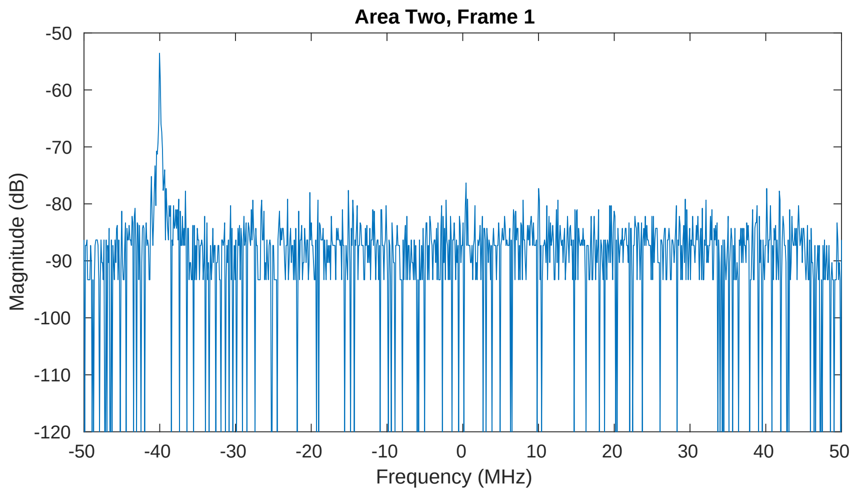


Figure 5.5: Area of interest two that contains only noise.

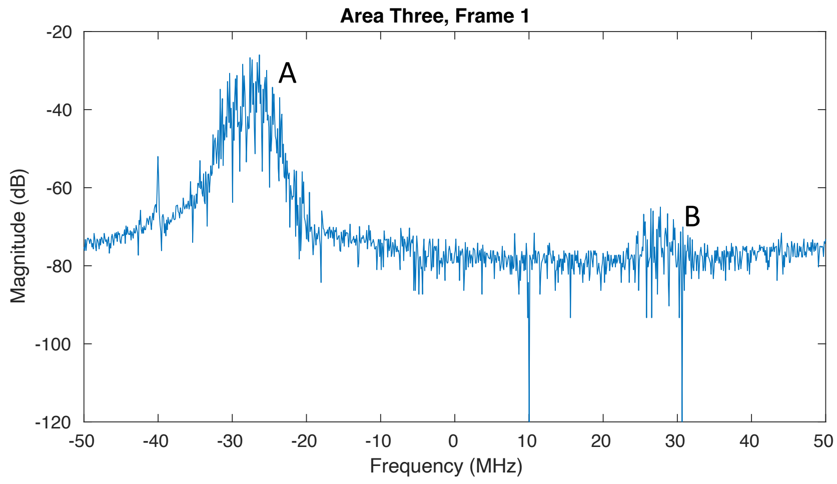


Figure 5.6: Area of interest three containing a wide band high power signal with a low power signal.

high power signal, marked as A in the figure, demonstrates how easy it is to mask other signals present in a frame. To detect the *entirety* of signal B with a uniform frame threshold, some of the noise around signal A would necessarily be above the threshold.

5.2 CA-CFAR Performance

This work focuses on hardware implementations, so when setting the parameters of the algorithms the hardware was considered. Because the CA-CFAR must take the average of the reference cells, the total number of cells in the reference set should be a power of two. This allows the divide operation of the average to be carried out using bit shifting.

For this example the total reference cell was set to 32, leaving 16 reference cells on each side of the cell under test. 14 Guard cells were inserted between the cell under test and the reference window on each side. Unfortunately, this means that the first 30 cells and the last 30 cells of the frame are not tested as they do not have the cells available to the left or right that can support the estimate. For all testing, the probability of false alarm was set to 10^{-6}.

Figure 5.7 shows the resulting threshold when CA-CFAR is applied to the first frame of the first area of interest. Here it is observed that CA-CFAR does a great job detecting signal B in the area. The threshold hugs B tightly and captures the entire body of the signal. The performance is not repeated on signal C. Although the leading edge of the signal is detected, the signal drops its trailing shoulder and avoids detection.

Figure 5.8 presents the CA-CFAR on the leading frame of the second area. The detection method performs as expected across this frame of data. Throughout the

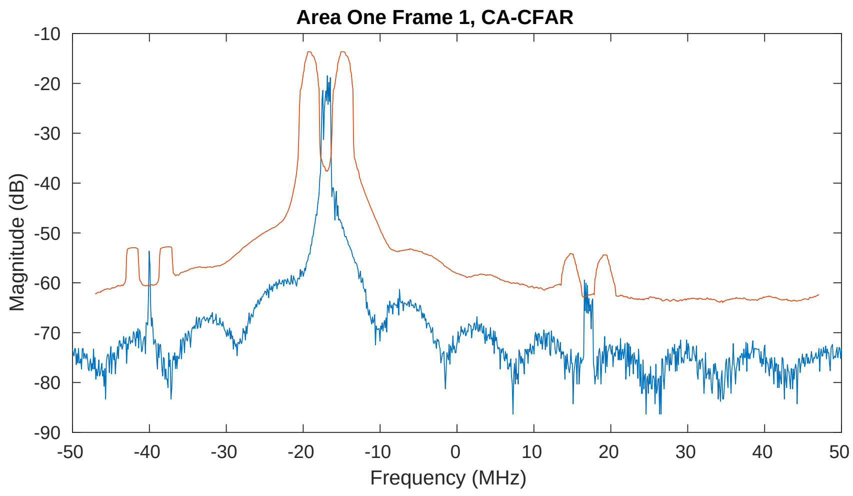


Figure 5.7: CA-CFAR applied to the first area of interest. N = 32, G = 28

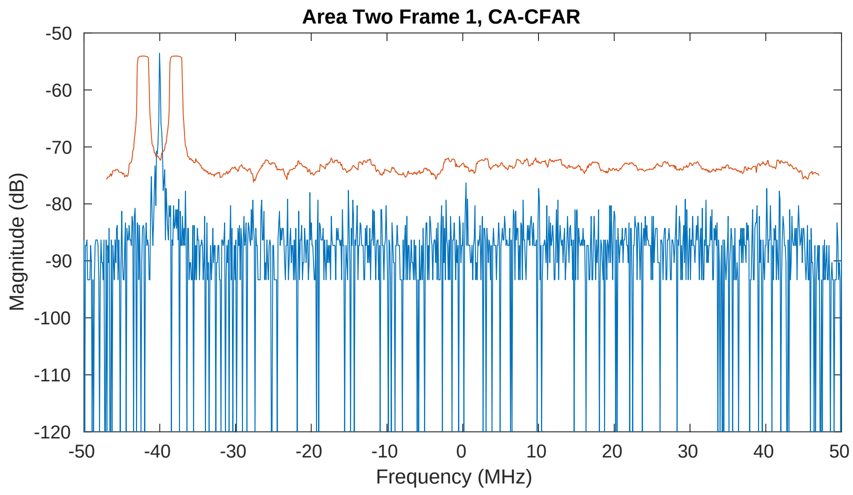


Figure 5.8: CA-CFAR applied to the second area of interest of mostly noise. N = 32, G = 28

frame the threshold stays above the spurious noise, even managing to avoid some pretty high peaks emerging from the noise.

So far CA-CFAR has shown to be effective at detecting signals with narrow bandwidths. Figure 5.9 gives an insight into the performance of detection against wider band signals that were present in area three. Here it becomes clear that the guard cells fail to be sufficient to detect either of the signals in the third area. When the signal exists across many bins it influences the guard cells in a way that raises

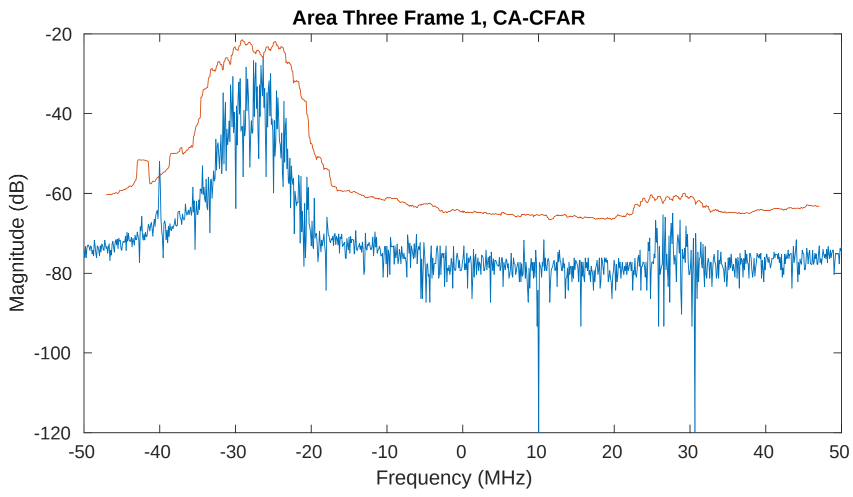


Figure 5.9: CA-CFAR applied to the first area of interest. N = 32, G = 28

the threshold to the point where the signal can no longer be detected. Unfortunately, this is also the case for the simulated downlink signal also as shown in Figure 5.10.

To have a better chance of detecting the signals that occupy a large number of bins, it would be necessary to increase the size of the guard cells on each side of the cell under test. This would reduce the number of cells that could be tested using

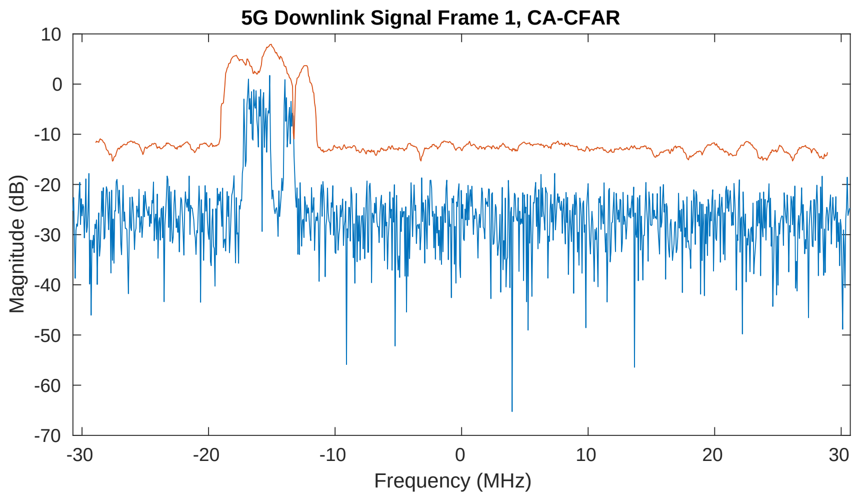


Figure 5.10: CA-CFAR applied to the first frame of the simulated downlink signal. N = 32, G = 28

this method as an increased number of guard cells would leave a wider area on each end of the frame that would not have a sufficient number of flanking cells to create an estimate. Increasing the number of untestable cells is unacceptable for this work as it seeks to maximize the amount of usable bandwidth for operation.

5.3 OS-CFAR Performance

OS-CFAR provides more parameter flexibility, as it does not require averaging and therefore does not require any limit on reference band length. Similarly, order statistic CFAR does not have guard bands, allowing the reference window to be longer than those on CA-CFAR. To benchmark the detection performance of the OS-CFAR detector the reference window was set to 27 on each side with the 37th bin chosen for the estimate after sorting. The order statistic value, k, of 37 was chosen as it lies just within the top one-third of the statistics [14].

Figure 5.11 shows OS-CFAR being applied across the first frame of area one of the real-world data. This provides a similar performance to the CA-CFAR method

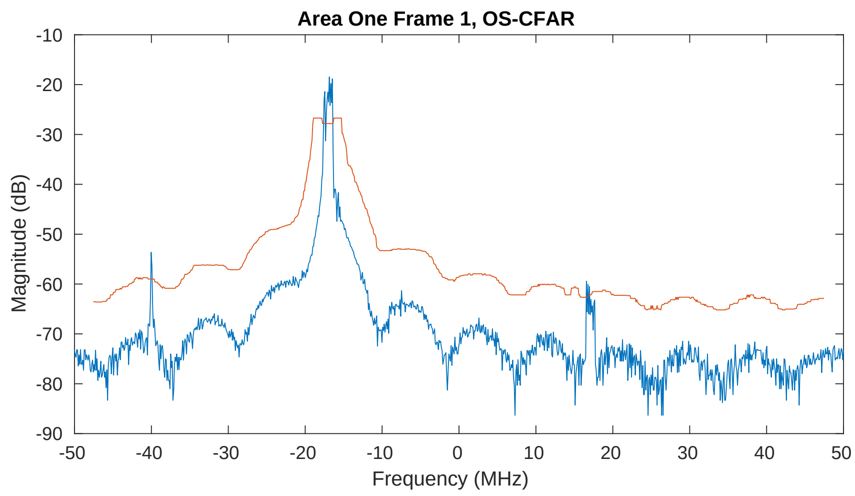


Figure 5.11: OS-CFAR applied to the first area of interest with parameters N = 54, K = 37

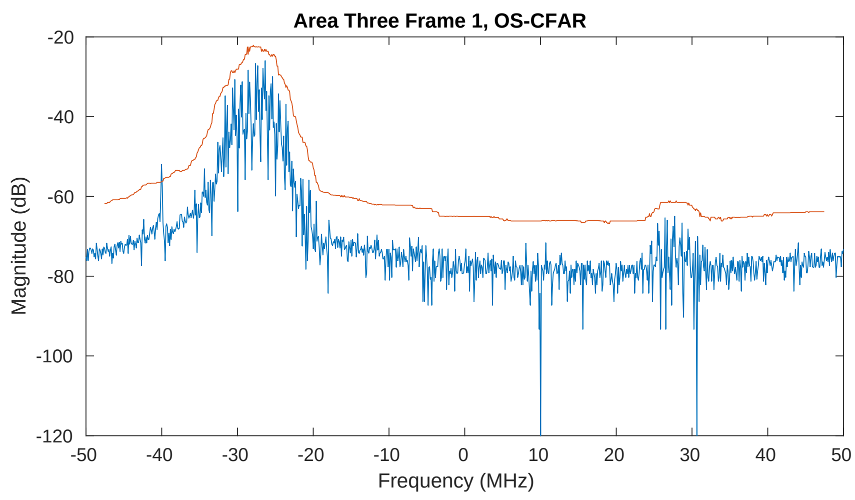


Figure 5.12: OS-CFAR applied to the interest where the wideband signals are missed. N = 54, K = 37

with a similar resolution of detection around signal B but again fails to detect the trailing half of signal C. Similar performance to CA-CFAR is also observed in the second and third area with the OS-CFAR missing all of the wideband signals in area three as shown in Figure 5.12. Again it is shown that the reference window required to detect these types of signals is too long to be practical for this application. It is this limitation that motivates the creation of a new type of detector.

5.4 Hardware Optimized Cell Averaging Estimation (HO-CAE)

Hardware Optimized Cell Averaging Estimation (HO-CAE) uses CA-CFAR as the starting point for creating estimates of the noise in the environment. Specifically, windows of reference data are used to multiple estimates of the noise variance. However, drawing inspiration from OS-CFAR, the threshold will be drawn from a particular ranked estimate. Unlike CA-CFAR, HO-CAE seeks a threshold that can be applied to the entire frame of data. This will allow the system to run a threshold check on every cell of the frame without having to ignore the bins at the end of each cell.

To accomplish this a series of estimates are generated by taking an average across the first N samples of the frame to create the first estimate $\widehat{\sigma^2_{W(1)}}$. The averaging window is then slid down the frame by N/2 samples to create the second estimate. This continues down the frame as shown in Figure 5.13. Of course, this means that the value of N should be selected such that the length of the frame is divisible by N/2.

After the window has made it across the frame the set of estimates can be ordered to generate a set of estimates in the form

$$\widehat{\boldsymbol{\sigma^2_W}} = \left[\widehat{\sigma^2_{W(1)}}, \widehat{\sigma^2_{W(2)}}, \widehat{\sigma^2_{W(3)}}, ..., \widehat{\sigma^2_{W(n)}}\right] \tag{5.1}$$

It is then important to determine which of these estimates should be used. As an initial approach, we use a Monte Carlo simulation of the noise-only case to guide an analysis of the performance to determine which $\hat{\sigma^2_{W(i)}}$ should be used to provide a false alarm rate closest to the desired P_{FA} of 10^{-6}. To narrow the initial search space, two values will be tested: $(i) \in 1, 5$.

Before this Monte Carlo search could be performed, a method for calculating the value of the multiplier α had to be identified. Because of this methods similarity to the CA-CFAR estimation type the CA-CFAR multiplier is proposed. The equation

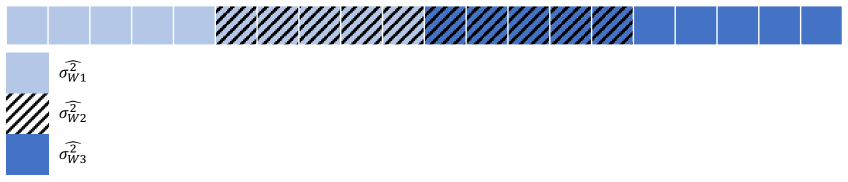

$\widehat{\sigma^2_{W1}}$
$\widehat{\sigma^2_{W2}}$
$\widehat{\sigma^2_{W3}}$

Figure 5.13: The sliding averaging window used to generate the estimates of $\widehat{\sigma^2_W}$. Here a frame length of 20 is shown with a window length of 10.

governing α is shown again as [14]

$$\alpha = N\left(P_{FA}^{-1/N} - 1\right) \tag{5.2}$$

5.4.1 Simulation and Performance

With the procedure and equations for calculating a threshold determined a simulation can be constructed to determine the achieved P_{FA} and the performance when the resulting threshold is used to set the FSS algorithm. To obtain the false alarm probability for the two selection options a series of noise-only frames were generated for which the HO-CAE detector was then applied. The noise was generated within the framework of the 5G waveform generator yielding blocks with 600 frames of 1024 samples. Table 5.1 shows the P_{FA} from selecting the first estimate and fifth estimate for 100, 200 and 1000 blocks with an input P_{FA} of 10^{-6} used to calculate α and window size of 64 samples.

Blocks	**Samples**	**Select 1^{st} P_{FA}**	**Select $5^{th} P_{FA}$**
100	61,440,000	9.0495E-06	1.8717E-06
200	122,880,000	1.0116E-05	2.712E-06
1000	614,400,000	9.8291E-06	2.2021E-06

Table 5.1: The resulting P_{FA} from noise only simulation with P_{FA} of 10^6 and window size of 64.

From the results, we can see that the fifth estimate from the ordered list provides a P_{FA} closer to that of the input across all sample sizes. Further, the resultant P_{FA} appears to be relatively stable, with smallest estimate of σ_W^2 providing approximately an order of magnitude greater rate of false alarms than is desired, and the 5^{th} estimate providing approximately a factor of 2 increase in desired false alarm rate. To get a full understanding of the performance, the approach is simulated further. This was accomplished using the 5G downlink signal generation techniques discussed previously. The method allows for the quick generation of 'blocks' that each contains 600 frames of data where each frame is 1024 samples in the frequency domain. These blocks have the same shape and location in the frequency domain making it easy to derive detection statistics, but are random in their content and roll-off outside of the signal's defined location.

Once a simulated signal is created it has no noise and only contains the 5G downlink signal in the time domain. Noise is applied across the entire signal using the measured signal power to achieve the desired SNR. From here the 614,400 samples of the block are divided into frames of 1024 samples with no overlap. The FFT is taken of each frame to generate a spectrogram. HO-CAE is then applied to each frame, generating a threshold. These thresholds are used to make a logical matrix from each frame and then combined to create a logical block for the whole cube. Once this has been done FSS can be applied to the logical matrix to determine what frequency band would be selected based on the decision from that frame. A logical mask was generated that defines where the simulated signal is located in the spectrogram. Since the signal is located in the same place in the spectrogram for each block generated the mask is valid for all simulated signals. This logical mask helps to derive the detection statistics and the probability of false alarm. Similarly, by using the logical mask we can derive statistics about how the detection affected the results of FSS. Figure 5.14 shows this process in 4 major steps: signal generation, noise addition, logical creation, and FSS.

An analysis of the data can provide insight into the detection probabilities and how they affect FSS when powered by HO-CAE-generated thresholds. Consider Table 5.2 which shows the data results for both selection options when tested with 100 blocks for a total of 60,000 frames of data.

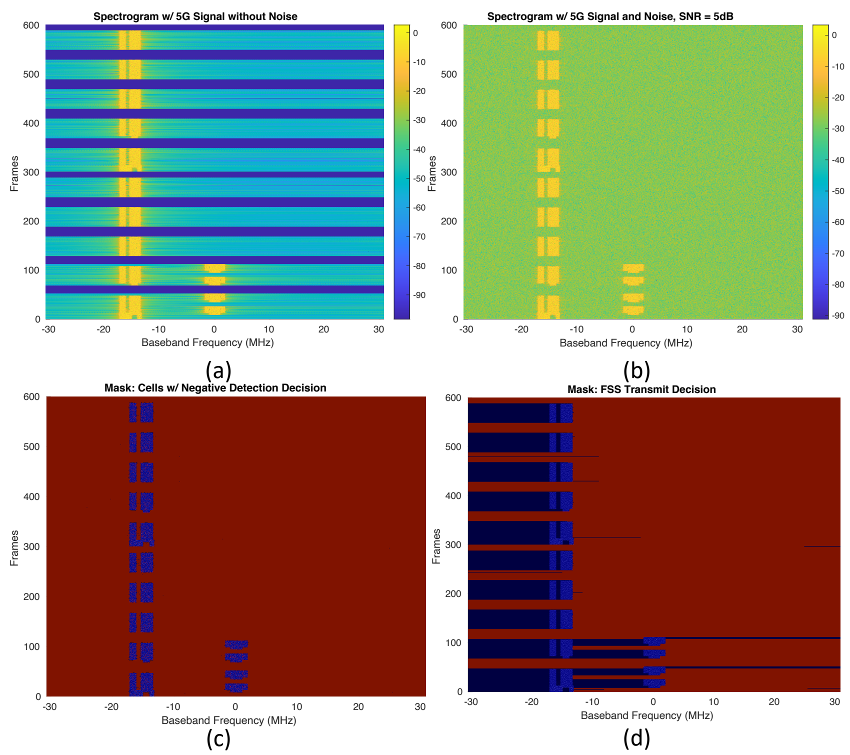


Figure 5.14: Steps to generating and simulating a 5G Downlink waveform with FSS. (a) The no noise signal after being partitioned into a block. (b) Noise added to the signal. (c) A logical cube on top of a 5G mask. (d) FSS decision based based on logical grid.

	SNR = 20		**SNR = 10**		**SNR = 5**	
	Sel. 1^{st}	**Sel. 5^{th}**	**Sel. 1^{st}**	**Sel. 5^{th}**	**Sel. 1^{st}**	**Sel. 5^{th}**
P_D	99.4%	99.3%	95.0%	94.4%	88.4%	86.9%
P_{FA}	0.021	0.019	3.93E-03	3.3E-03	1.35E-03	1.06E-03
FSS Exact Match	32.2%	33.4%	58.1%	61.5%	69.8%	71.1%
Start Within 2 Bins	40.0%	42.7%	84.7%	88.5%	94.3%	95.7%
Size Within 5 Bins	49.8%	54.6%	93.6%	95.8%	97.1%	97.9%
Size Within 10 Bins	62.7%	69.3%	96.6%	97.7%	97.6%	98.3%

Table 5.2: The results of probability simulations for 5, 10 and 20 dB SNR signals. Probability data is given as wells as data compared to FSS with perfect knowledge.

Reviewing these results reveals a very curious trend. With the two estimate selection options, there is a very narrow difference in the probability of false alarm and probability of detection. Selecting the first estimate consistently yields a higher probability of detection, but of course at the price of a slightly higher probability of false alarm. As might be expected, the selection of the 5^{th} estimate yields a smaller detection probability but also a smaller false alarm rate.

Of most significant note for this work is the performance of FSS using HO-CAE thresholds against the FSS with perfect knowledge. To properly accomplish the comparison several metrics are introduced. First, FSS exact match helps to describe what percentage of frames HO-CAE thresholds yield the same FSS match as the perfect knowledge indicator. Similarly, the second metric compares the start bin. Of particular interest is what percentage of frames does HO-CAE thresholds provide a start location within 2 bins of the ideal. Finally, the size of the operational band is compared as the goal of this system is to utilize as much available spectrum as possible. Of note, if the FSS algorithm is operating at 100 MHz of bandwidth generating frames of length 1024, a size differential of 10 bins is equivalent to 976 kHz of lost or interrupted bandwidth. Across all of the above mentioned metrics, the 5^{th} edges out the lowest estimate. A fairly high percentage of exact matches were achieved for both selections.

Looking at the exact FSS matches across the several different SNRs, as shown in Figure 5.15, has unexpected behavior. Where the matches improve from zero to eight dB SNR it then starts to decline before coming to its lowest point at 20 dB SNR. A review of the logical plots resulting from the threshold will help to identify the FSS match trend. Figures 5.16 and 5.17 show the logical mask resulting from SNR 20 and SNR 0. From there it is clear that the sidelobes resulting from the 20dB SNR signal are visible above the noise floor and are pushing the detected band

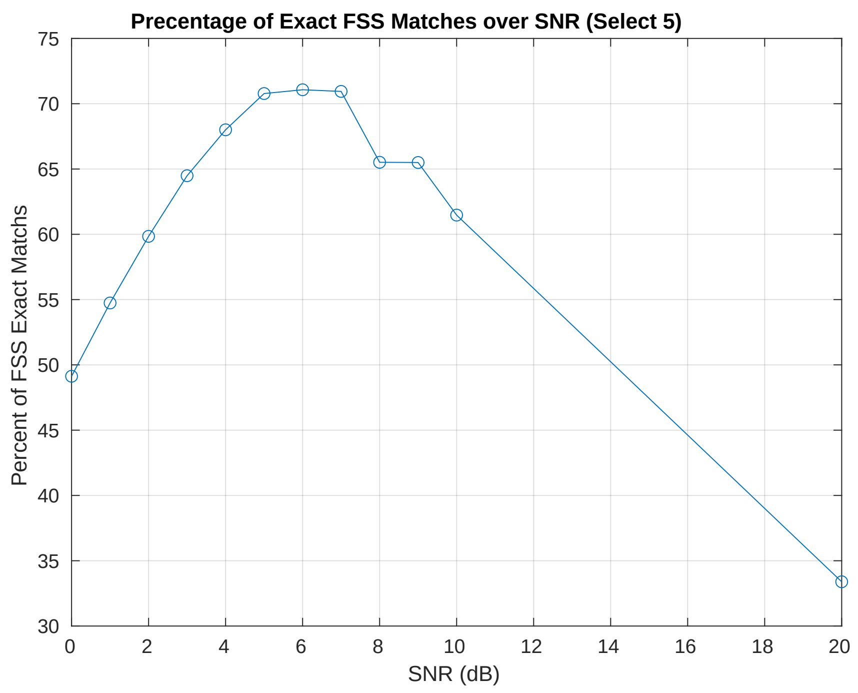


Figure 5.15: The percentage of exact FSS matches between the perfect case and HO-CAE over the simulated SNR values.

wider around the signal. Similarly, a look at the 0dB SNR plot gives insight into why a slightly lower probability of detection is not required to have decent exact match performance. Although there are missed detections inside of the signal, FSS is still able to mostly avoid them as bins near their edges are detected to define their boundary. FSS, as studied here, seeks the largest bin and is therefore tolerant to some missed detection in the gaussian structure of the signal so long as there is a detection near the boundaries of the signal.

5.5 Real-World Performance

This chapter will conclude as it opened, with a look at performance on the set of real-world data. Each of the previously examined areas of interest from the real world collect from Section 5.1.2 will be reexamined with HO-CAE detection.

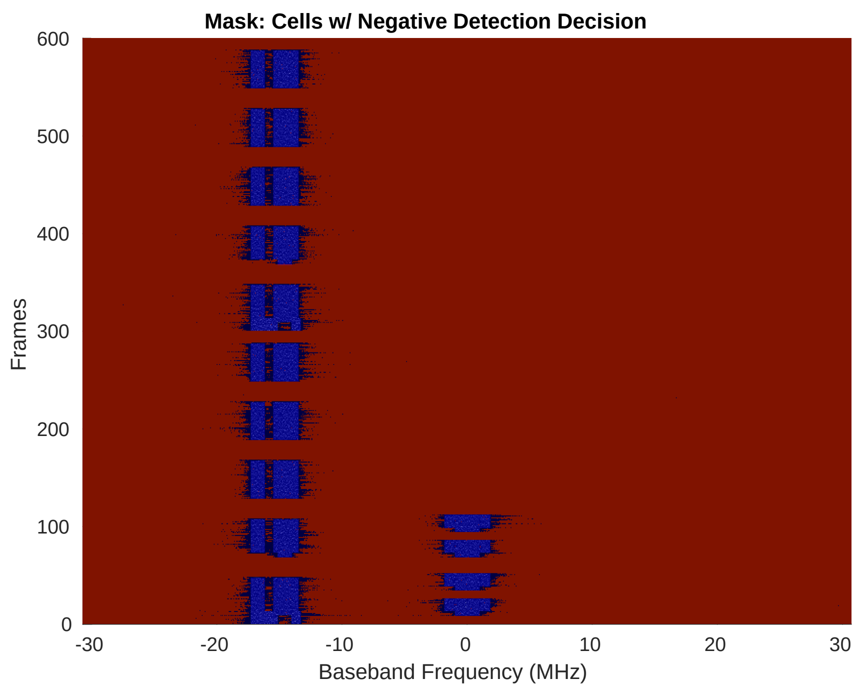


Figure 5.16: The result of logical mask where non-detected bins are marked for a 20dB SNR 5G downlink signal.

Figure 5.18 shows the first frame of area one. Here we see very similar performance to that seen with the order statistic and cell averaging CFAR techniques. One key difference is shown in C. Where both of the previous methods caught the leading edge and missed the trailing edge, the HO-CAE method just narrowly captures the trailing edge. Although it misses some of the bins within that third signal, recall catching the edges is the most important goal for FSS as the interior points between detections are too narrow in frequency to be considered a viable band.

Next, the mostly noise case is examined in 5.19. Again we see almost identical performance to that of the previous CFAR techniques. Notably, this threshold is generated closer to the noise with some of the noise peaks coming close to the threshold. Importantly none of the peaks breach the threshold here.

Finally, the very wide signals of the third area of interest are presented in 5.20. It is in this case where HO-CAE stands out among the others. The threshold can

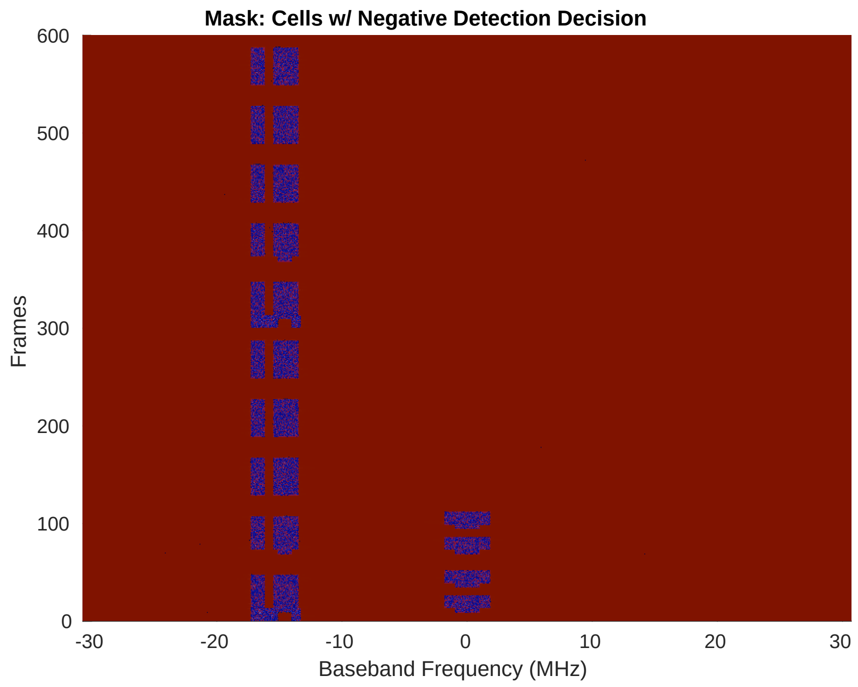


Figure 5.17: The result of logical mask where non-detected bins are marked for a 0 dB SNR 5G downlink signal.

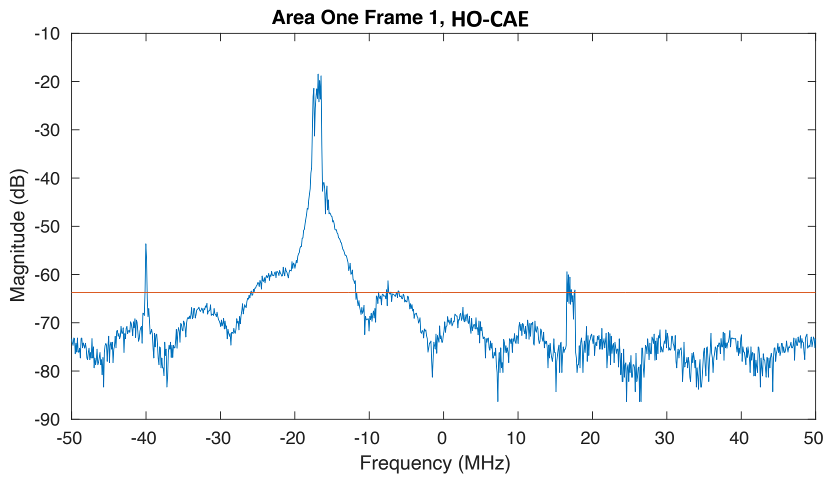


Figure 5.18: The first frame of the first area of interest from the real-world data with HO-CAE applied.

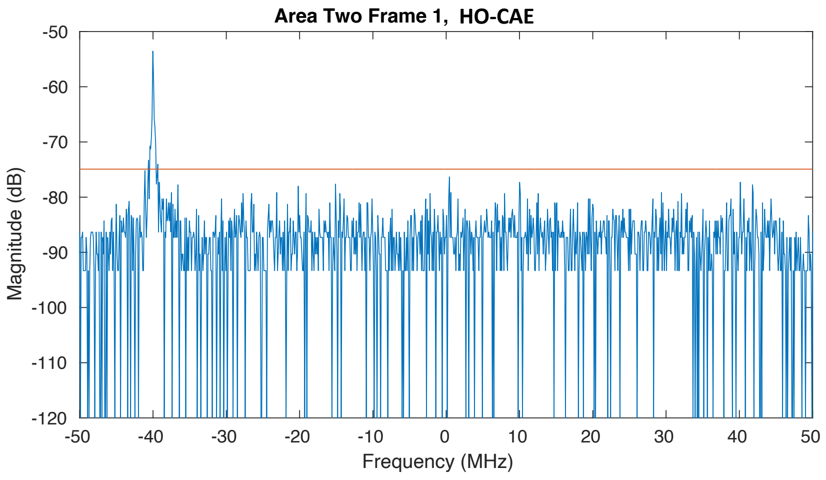


Figure 5.19: The first frame of the second area of interest from the real-world data with HO-CAE applied.

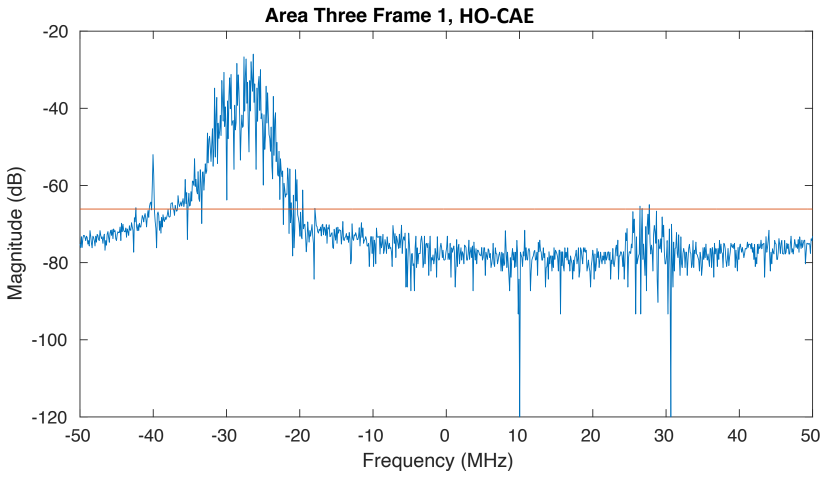


Figure 5.20: The third frame of the second area of interest from the real-world data with HO-CAE applied.

capture all of wideband signals A and even manages to capture a few bins in the middle of low power signal B. Unfortunately, no threshold can be applied to the entire frame that would detect both of these signals without the frame appearing almost completely closed.

With all of these the previously mentioned performance data considered, from simulation to real-world, it is determined that this method yields worthwhile performance. The hardware considerations built-in to the creation of this algorithm similarly should yield latency and resource savings as discussed in the next chapter.

Chapter 6

HO-CAE Hardware Architecture

This chapter will discuss the proposed hardware implementation of the hardware optimized, cell averaging estimator discussed in the previous chapter. First, a high-level theory of operation will be provided as a basis for the hardware elements. Next, the hardware elements necessary to the operation of this system will be discussed with a focus on three major pieces: frame buffer, adder blocks, and estimate selectors. The chapter concludes by considering all of these structures together and presenting the projected performance using the same metrics defined in Chapter 4.

6.1 Theory of Operation

Unlike the previously presented FSS algorithm, this algorithm requires the data samples within a frame to sit idle in cache and wait for the threshold to be calculated. Buffering of the samples could happen in several places, mostly dependent on where FSS is going to be applied. Three options were considered when first designing the outline for this block.

The first option considered was to pass them through the HO-CAE block and buffer them in a separate FSS block until the HO-CAE block passed the threshold on a separate data channel. This was dismissed fairly quickly. It simply did not make sense to redesign the existing standalone FSS block to buffer the data samples while waiting for the threshold to be passed.

Next, the buffer was considered in the HO-CAE block itself. Samples would be considered for the algorithm as they entered the block and stored into a FIFO buffer. A threshold decision would be reached shortly after the receipt of the last sample in the packet. At this point, the threshold value would be attached to the front of an output packet as metadata and transmitted to the FSS block.

This approach was mostly adopted, except the threshold transmit. Even attaching the threshold to the packet would require an (albeit small) rework of the existing FSS block. Since some work would be required anyway, it was decided to integrate the FSS block straight into the detection.

By integrating the FSS right into the threshold estimate block the hardware resources required to operate a separate block are mitigated. Effectively, the number of endpoints and de-framer structures would be the same for a HO-CAE-FSS implementation. If the HO-CAE and FSS blocks were separate, depending on the port configuration it would require at least an additional two end streaming endpoints and two additional de-framers.

Caching is required for this block so the appropriate cache size must be determined. The packet length, P, will be the key metric here with a cache size capable of storing $2K$ data words is desired. An extra frame worth of buffer space provides at least an additional K clock cycles to begin re-transmitting a received packet before the stream would be halted, which is more than sufficient.

Samples will be read into these buffers and supplied to accumulators as they come into the block from the data stream. Once an accumulator has reached the end of its window, the estimated candidate will be shifted and then supplied to a smaller cache. The estimate buffer stores the lowest five estimates in sorted order.

Once the end of the frame is reached, the largest estimate will be fed through a multiplier which will apply the α value.

From here, the threshold is ready to be applied to the data frame and the same FSS state machine discussed in Chapter 4 is used to process the frames. As this data is going out of the block with FSS being applied other data is being read into the other buffer and the threshold of the next frame is being calculated. When the bucket has been calculated it will be sent out of its special data port just like with the standard FSS algorithm.

6.2 Major Structures

The theory of operation discussed three main parts of the implementation and each are discussed in this section. The operation of the sample buffer, accumulators, and estimate sorting buffer is of critical importance to the operation of the system.

6.2.1 The Buffer

As previously mentioned a buffer of two times the size of the packet length (i.e., $2 \cdot K$) is necessary to ensure constant operation. Conceptually, the buffer is maintained as two distinct FIFOs that allow the packets to be kept separate. Counters are employed in each to track where the packet boundary as the first packet is read out of the buffer and the third packet is read in. As long as the number of clock cycles required to start sending data does not exceed K, then there will not be any overruns of the buffers.

In the implementation, the buffer is just a single FIFO, and three counters are used to track the whole packet in the buffer, the packet being read out, and the packet being read in. These counters help to keep track of the current state and also are used to help run the next structure, the accumulators.

6.2.2 The Accumulators

For this work, the samples coming into the accumulators will be 16-bit magnitude values but the block design should be flexible to accept up to 32-bit magnitude values. Considering the larger case, the window size for this implementation is 64 bins, which is a 7-bit number, with 32-bit magnitude values, a 39-bit value is the maximum value the accumulator would yield with saturated ADCs.

The 7 series FPGA in the X310 radio contains 1,540 Xilinx DSP48E1 slices that handle the math operations and is shown in Figure 6.1. Slices of this architecture are capable of performing up 48-bit add/accumulate operations in a single clock latency [23]. Performance like this will enable the samples to be fed sequentially into the accumulator with each clock cycle.

To reduce the utilization of the FPGA resources, this proposed implementation only uses two accumulator blocks. A reduced number of DSP slices is achieved

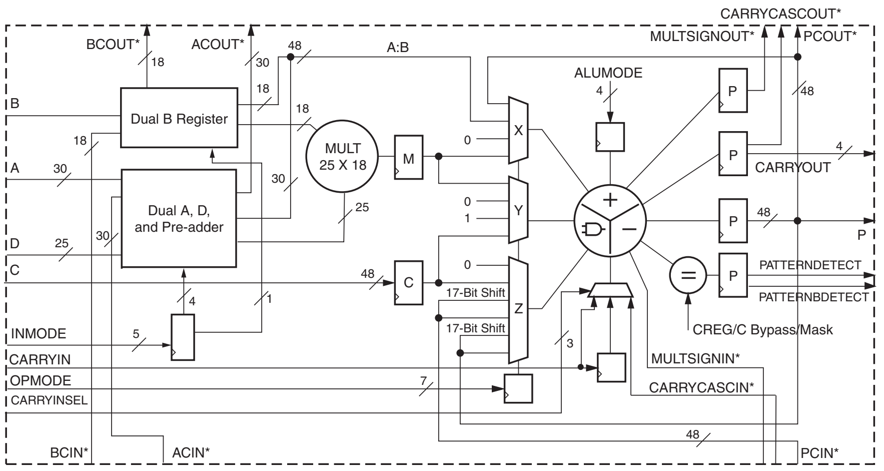


Figure 6.1: The DSP48E1 Slice inside 7 Series Xilinx FPGAs

by utilizing the packet counters employed in the buffers. For each packet, the first accumulator consumes all of the samples except the last $\frac{N}{2}$ samples of a packet, spinning off an estimate and resetting the accumulator for N samples. The second accumulator operates similarly but instead, it consumes all of the samples in the packet except for the first $\frac{N}{2}$.

Both of these structures are fed into a single non-circular shifter, putting zeros into the leading bits. The data cadence coming out of the accumulators makes this possible. Given that the window is shifted by more than one sample per estimate, then there will be sufficient time between the generation of each estimate to shift the amount and send it to the sorting estimate buffer.

6.2.3 Sorting Estimate Buffer

The sorting estimate buffer is designed to keep just the lowest, in this case, five values that come from the shifters and store them in sorted order which can be accomplished in a single clock cycle. Effectively, this structure is a chain of 39-bit multifunction registers with four different operations. The number of registers in the chain is dependent on which selection the user intends to make, for this work the 5^{th} value is used but ten registers should be instantiated to provide some flexibility. For discussion of this structure, the left-most register will be register number one and considered the lowest value with the registers numbered up and considered larger moving right across the bank.

The first operation state simply holds the register's current value. Second, the register is capable of storing an estimate from the shifter. The third operation accepts the value from the register to left, this makes room for a new value to be inserted using the second operation. Finally, this register can reset itself to a state where all values are the maximum 39-bit number. The requirements to trigger each of these operations are critical to a successful operation.

The first operation to be considered is operation two, storing the input received from the computation. It is clear that this set of registers should only have one of each value in it and that it has a specific location in the set that it belongs to. To determine where the value goes in the block of the register, two comparators are used. The condition that grants a value assigned to a register is if the value is smaller than the register being considered and larger than the register to the left. All of the registers are tested simultaneously with the value and only one is selected.

When one register chooses to accept the value then all registers to the right are commanded to execute the third operation. When a value is inserted into a register in the chain, the registers to the right of the insertion point accept the value from the left. This means that if register one is assigned the value then register two takes on register one's value affecting all registers up the chain. By writing the new value to the assigned register and writing each shifted register on the same rising edge of the clock, no data is lost except for the data that was previously in the largest register. Table **??** summarizes the operation of the sorted estimate buffer.

When the end of the packet is detected and the required value is extracted from the registers, the reset condition is triggered. Because this set of registers is looking for small values they need to be reset to large values. It is for this reason that a reset

Function	Description
Reset	Reset all bits high.
Hold	Maintain the current value
Shift	Accept the value from the right. Essentially shifting the values.
Accept	Accept the value from the accumulator structure.

Table 6.1: A summary of operation carried out by each register in the sorted estimate buffer.

triggers all bits in each register high. Leading into the final operation, when none of these conditions are met the registers hold their value.

6.3 Performance

Performance metrics described in the previous chapter should now be discussed for the HO-CAE-FSS block. The data latency, the amount of time measured from when the last sample is received to when it is transmitted is of much more interest for this block. Re-transmission of the blocks adds K clock cycles to the data latency of the block. In the case of this design, 1024 clock cycles.

The time it takes for the last estimate to propagate through the iterator and hit the sorted estimate buffer before the multiplication of alpha should also be considered. One clock cycle after the last sample is read in, the last estimate is shifted. Another clock cycle and the estimate is selected.

Two clock cycles are allotted for the alpha multiply. If the value of the estimate is larger than 28-bits then the multiply will take two cycles [23]. Estimates taking on the maximum should not ever exist. If an estimate this large did come through the system, that indicates that the ADC in the RF front end is saturated to all ones and no bin value will exceed the threshold value.

The total data latency is 102 clock cycles. When considering the operation clock rate of 200 MHz the total data latency time is 5.14 microseconds. The second metric, product latency, can be easily calculated using the product latency of FSS and data latency. Combining the two latencies bring the total product latency to 1031 clock cycles or 5.155 microseconds.

Any implementation that requires an analysis of the last sample of the packet before FSS can be applied, will require buffering somewhere and will have an extended product and data latency. The critical path to be minimized is the threshold calculation time that requires the last sample of the packet and the time required to compute and transmit the largest continuous space in the frame. The hardware optimized, cell averaging estimator was designed with this critical path in mind and therefore yields an efficient method of calculating the threshold and applying FSS.

Chapter 7

Conclusion and Future Work

This work set out to analyze fast spectrum sensing for detecting users of the electromagnetic spectrum. To make effective use of such sensing techniques, the implementation of the algorithms would need to operate in real-time, ideally without specialized hardware. Successful strategies and implementations were presented in this work that met this real-time criterion.

A hardware implementation for using a static threshold was presented. Nanosecond latency between the receipt of the last sample and the generation of the algorithms results was achieved. This is an impressive performance that should allow any system to react rapidly to avoid emitters entering the field above the threshold in the current environment. But, because of the changing nature of the environment, a dynamic method of setting the threshold that would minimize impact to the latency was required.

To meet the challenge of the dynamic environment problem a new hardware optimized estimator based on CFAR was developed to estimate the threshold that should be applied to FSS. HO-CAE was shown to be an effective threshold estimator for the FSS algorithm that could be implemented on FPGA to provide high-speed results. Performance metrics showed a HO-CAE-FSS block capable of providing an FSS result 5.145 microseconds after receiving the last data sample of the spectral frame.

There are many ways to expand on this work in the future. First, a formal analysis of the impact of the window size on the performance of HO-CAE should be examined. Additionally, more values for the ordered selection of the estimator should be investigated beyond the smallest and 5_{th} estimates. The potential combination of these two factors should be fully examined to determine if any additional performance can be achieved.

To expand the system, the development of a hardware signal generation block for FPGA would enable loopback and transmission tests that are capable of further demonstrating the practical application of these implementations. Such a block would need to use direct digital synthesis to deterministically generate baseband signals with linear frequency modulation (LFM) across any of the selected transmission bands regardless of the size or baseband location of the interval.

Finally, to be able to expand to the transmission of any number of waveforms, a further processing step is necessary. Signal censoring would become critical to remove any self transmitted signals from the environment before running the FSS algorithms on the data. This would be critical to ensuring that the radio's signals aren't masking new emitters, which would be the result of just ignoring its transmission bands. With the transmission and censoring features built in the system would then be ready for independent operation ready for the development of further hardware-generated waveforms.

References

[1] *RF Network-On-Chip (RFNoC^{TM}) Specification*, 1st ed., Ettus Research, Oct. 2020.

[2] "Cisco annual internet report (2018-2023) white paper," Cisco, Tech. Rep., 2020.

[3] A. Petrin and P. G. Staffes, "Analysis and comparison of spectrum measurements performed in urban and rural areas to determine the total amount of spectrum usage," *International Symposium on Advanced Radio Technologies (ISART)*, 2005.

[4] J. Jeon, H. Niu, Q. Li, A. Papathanassiou, and G. Wu, "Lte with listen-before-talk in unlicensed spectrum," *IEEE International Conference on Communication Workshop (ICCW)*, 2015.

[5] B. H. Kirk, R. M. Narayanan, K. A. Gallagher, A. F. Martone, and K. D. Sherbondy, "Avoidance of time-varying radio frequency interference with software-defined cognitive radar," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 33, no. 3, pp. 1090–1107, Nov. 2018.

[6] A. F. Martone, K. I. Ranney, K. Sherbondy, K. A. Gallagher, and S. D. Blunt, "Spectrum allocation for noncooperative radar coexistence," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 54, no. 1, pp. 90–105, feb 2018.

[7] S. Shankar, C. Cordeiro, and K. Challapali, "Spectrum agile radios: Utilization and sensing architectures," *IEEE International Symposium on Dynamic Spectrum Access Networks (DySPAN)*, 2005.

[8] A. Papoulis and S. U. Pillai, *Probability, Random Variables and Stochastic Processes*, S. W. Director, Ed. McGraw Hill Education, 2002.

[9] W. A. Gardner, "An introduction to cyclostartionary signals," in *Cyclostationarity in Communications and Signal Processing*, W. A. Gardner, Ed. IEEE Press, 1994, ch. Chapter 1.

[10] C. Spooner, "The cyclic autocorrelation function," 2015. [Online]. Available: https://cyclostationary.blog/2015/09/28/the-cyclic-autocorrelation/

[11] K. Kim, I. A. Akbar, K. K. Bae, J. sun Um, C. M. Spooner, and J. H. Reed, "Cyclostationary approaches to signal detection and classification in cognitive radio," *IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, 2007.

[12] R. S. Roberts, W. A. Brown, and H. H. Loomis, "Computationally efficient algorithms for cyclic spectral analysis," *IEEE Signal Processing Magazine*, 1992.

[13] H. Urkowitz, "Energy detection of unknown deterministic signals," *Proceedings of the IEEE*, vol. 55, no. 4, p. 9, April 1967.

[14] M. A. Richards, *Fundamentals of Radar Signal Processing*. McGraw-Hill Education, 2014.

[15] T. Ulversøy, "Software defined radio: Challenges and opportunities," *IEEE Communication Surveys & Tutorials*, 2010.

[16] *UBX Duaghterboard Datasheet*, Ettus Research.

[17] *USRP^{TM} X300 ans X310 X Series Datasheet*, Ettus Research.

[18] "Uhd — ettus knowledge base," Ettus Knowledge Base, 2020, [Online; accessed 3-July-2021]. [Online]. Available: https://kb.ettus.com/index.php?title=UHD&oldid=4720

[19] *AXI Reference Guide*, Xilinx, 2017.

[20] "Getting started with rfnoc development," Ettus Research, Tech. Rep., Oct. 2020.

[21] "Yaml ain't markup language (yaml) version 1.2," [Accessed 10-July-2021]. [Online]. Available: https://yaml.org/spec/1.2/spec.html

[22] *5G NR Downlink Vector Waveform Generation*, R2021a ed., MathWorks. [Online]. Available: https://www.mathworks.com/help/5g/ug/downlink-carrier-waveform-generation.html

[23] *7 Series DSP48E1 Slice User Guide*, Xilinx.