UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

# GPS DISCIPLINED RFSOC SYNCHRONIZATION, TIMING, AND PERFORMANCE CHARACTERIZATION IN BISTATIC RADAR SYSTEMS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

MICHAEL ORTIZ-CHEEK

Norman, Oklahoma

2023

# GPS DISCIPLINED RFSOC SYNCHRONIZATION, TIMING, AND PERFORMANCE CHARACTERIZATION IN BISTATIC RADAR SYSTEMS

A THESIS APPROVED FOR THE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF


Dr. Nathan Goodman, Chair

Dr. Justin Metcalf

Dr. David Schvartzman Cohenca

© Copyright by MICHAEL ORTIZ-CHEEK 2023

All Rights Reserved.

# Table of Contents

List of Figures ... ix

Abstract ... xiii

Chapter 1: Introduction ... 1

1.1 Background and Motivation ... 1

1.2 Proposed Solution ... 4

1.3 Outline ... 7

Chapter 2: Signal Characterization ... 9

2.1 Motivation ... 9

2.2 EndRun Ninja Precision Timing Module ... 10

2.2.1 Layout ... 10

2.2.2 Command Line Interface on Ninja and Commands Used ... 12

2.2.3 Bash Scripting ... 12

2.3 Signal Acquisition with the 5950 RFSoC Module ... 13

2.3.1 Linux Command Line Interface ... 13

2.3.2 “acquire” Example Program and Initialization Files ... 13

2.3.3 Bash Scripting and Automating File Transfer ... 15

Chapter 3: Signal Characterization Results ........................................................ 19
3.1 Frequency Drift Characterization.............................................................. 19
3.1.1 Experiment Objectives .......................................................................... 19
3.1.2 Challenges ............................................................................................. 20
3.1.3 Setup ...................................................................................................... 22
3.1.4 Signal Analysis ...................................................................................... 24
3.1.5 Results ................................................................................................... 29
3.2 Timing Drift Characterization ..................................................................... 34
3.2.1 Experiment Objectives .......................................................................... 34
3.2.2 Setup ...................................................................................................... 34
3.2.3 Signal Analysis ...................................................................................... 36
3.2.4 Results ................................................................................................... 38
3.3 Absolute GPS Time Single Pulse Characterization .................................... 43
3.3.1 Experiment Objectives .......................................................................... 43
3.3.2 Setup ...................................................................................................... 44
3.3.3 Signal Analysis ...................................................................................... 46
3.3.4 Results ................................................................................................... 49
Chapter 4: Synchronization Architecture............................................................ 51

4.1 Motivation and Objectives ..... 51
4.2 Equipment ..... 53
4.2.1 Pentek Model 5950 ..... 53
4.2.2 Ninja ..... 59
4.3 Field Programmable Gate Array Code ..... 59
4.3.1 Overview ..... 59
4.3.2 Block Diagrams and Intellectual Property Blocks ..... 60
4.3.3 AXI Communication Protocol and AXI4-Stream ..... 62
4.4 Coherent Processing Interval Control Block ..... 63
4.5 Modifying Analog-to-Digital Converter Functionality ..... 71
4.6 Modifying Digital-to-Analog Converter Functionality ..... 74
4.6.1 Original DAC Configuration ..... 74
4.6.2 Custom AXI4-Stream IP ..... 75
4.7 Timing Tolerances of Solution ..... 82
4.8 Implementation Issues ..... 86
Chapter 5: Conclusion ..... 90
5.1 Conclusions ..... 90
5.2 Future Work ..... 91

Appendix A: Code ........ 93
Signal Characterization Setup: ........ 93
Frequency Drift Bash Script ........ 93
Timing Drift Bash Script ........ 96
GPS Single Pulse Bash Script ........ 99
Signal Characterization Analysis: ........ 104
Main (Shared by all three experiments) ........ 104
Setup (Shared by all three experiments) ........ 105
Analysis (Frequency Drift) ........ 112
rfsoc_analyze (Frequency Drift) ........ 116
Plot (Frequency Drift) ........ 117
Analysis (Timing Drift) ........ 120
Plot (Timing Drift) ........ 123
Analysis (GPS Single Pulse) ........ 126
Plot (GPS Single Pulse) ........ 129
Solution Code ........ 130
radar_control.v ........ 130
radar_control_reset.v ........ 138

axis_dac_switch.v ........................................................................ 141

References ........................................................................................ 145

# List of Figures

Figure 1: Example of frequency drift between transmitted and received signals... 3

Figure 2: EndRun Ninja Port Diagram ............................................................... 11

Figure 3: A sample from the bash script used in the first characterization experiment showing the length of the call needed to start "acquire" without using a script... 16

Figure 4: UI created by bash script for easy modification of "acquire" operation. ........................................................................................................................ 17

Figure 5: An example of jagged I/Q results from a data and a close zoom of the peaks of each channels' I/Q FFT. ......................................................................... 25

Figure 6: Manual frequency drift I/Q results at a 101 Hz offset.......................... 27

Figure 7: Manual frequency drift results. ............................................................ 28

Figure 8: Frequency drift results for two Ninja units operating without GPS discipline. ........................................................................................................... 30

Figure 9: Frequency drift results for two GPS-disciplined Ninja units. ............... 32

Figure 10: The results of cross-correlation between a single set of 10MPPS signals captured from two GPS-disciplined EndRun units.............................................. 37

Figure 11: All 1000 cross correlations generated from data acquired by the 5950. ........................................................................................................................ 38

Figure 12: The highly aliased results of a timing drift experiment done between the two Ninja units without GPS discipline.............................................................. 39

Figure 13: The lag within each individual cross-correlation at which the cross-correlation value was highest. 40
Figure 14: The first hour of the above results shown in more detail. 41
Figure 15: Maximum Lag in the 10MPPS signal after reliable GPS lock was achieved. 42
Figure 16: A sample of "simultaneously" triggered single pulses from each EndRun. 47
Figure 17: Highlighting the precise time between each of the pulses displayed above. 48
Figure 18: The results of the characterization of the single pulses. 50
Figure 19: The beginning of the path an arbitrary waveform takes within the 5950's original block diagram. 54
Figure 20: An arbitrary waveform unable to be correctly synthesized by the 5950's DAC due to frequency issues. 57
Figure 21: An arbitrary waveform (LFM) that can be more correctly synthesized by the 5950's DAC.. 58
Figure 22: An overview of the complete modified block diagram used in the 5950. 61
Figure 23: A timing diagram showing how two IP blocks interact using the AXI4-Stream protocol. 63
Figure 24: The "radar_control" block and each of its inputs and outputs. 65

Figure 25: The “radar_control” block, along with the constants and reset logic, including “radar_control_reset”. ........................................................................ 68

Figure 26: Vivado timing diagram showing operation of “radar_control” block given certain inputs. ........................................................................................ 69

Figure 27: The path the signal associated with the TRIG front-panel input on the 5950 takes originally. ...................................................................................... 72

Figure 28: The same TRIG signal is rerouted to the “radar_control” block, and the ADC trigger output from the “radar_control” block is routed to where the TRIG signal used to go. ................................................................................................ 73

Figure 29: A section of the block diagram whose purpose is controlling the DAC and ADC, showing no trigger signal for the DAC .............................................. 75

Figure 30: The new path an arbitrary waveform takes through the block diagram. ........................................................................................................................... 79

Figure 31: A timing diagram showing the outputs of the “radar_control” and axis_dac_switch blocks in a simulated environment in Vivado. .......................... 81

Figure 32: A closer look at the above timing diagram, confirming that the signal streamed by axis_dac_switch is indeed 32768 samples long at the 1GS/s DAC sampling rate used in initialization files throughout this research effort. ............. 82

Figure 33: Clock cycle period ................................................................................ 83

Figure 34: Delay between dac_trigger rising edge and axis_dac_switch streaming first sample of arbitrary waveform. ..................................................................... 84

Figure 35: Ideal delay between dac_trigger rising edge and adc_trigger rising edge.
.............................................................................................................................. 85

Figure 36: The error constant throughout all attempts to compile........................ 87

# Abstract

Distributed radar geometries offer multiple advantages over monostatic pulse-Doppler radar, but synchronizing frequency and timing for transmitting and receiving nodes in a distributed system is required to more accurately detect range and Doppler frequency. A GPS-disciplined bistatic radar synchronization system design running on an RF System-on-Chip (RFSoC) transceiver and GPS-disciplined precision timing reference is detailed, and the features and limitations of these two individual systems are examined. To better understand error tolerances of relevant signals produced by the timing reference used in the synchronization system, in-depth analyses of frequency drift, timing drift, and jitter are conducted and described both with and without GPS disciplining. Custom-designed FPGA IP designed to implement transmit and receive pulse-Doppler radar functionality in the RFSoC-based system is introduced.

# Chapter 1: Introduction

## 1.1 Background and Motivation

Bistatic radar, defined simply as a radar system in which the transmitting and receiving antennas are not collocated, can be useful in military and other applications. Although the main condition for a radar system to be considered bistatic is if the transmitting and receiving antennas are in different locations, often the entire system supporting the antennas will be split in two as well [1]. This includes the onboard clocks, reference signals, local oscillator (LO) and other timing-critical infrastructure, which can introduce problems when said clocks are not properly synchronized.

When the transmitting and receiving nodes in a bistatic radar system are separated by enough physical distance to preclude use of the same timing references, both units will use their own clock to perform radar operations. Using the same reference, or at least synchronizing each node's reference, is especially important in pulse-Doppler radar applications, where the timing and frequency coherence of the pulse repetition intervals (PRIs) allows the detection of Doppler frequencies [2]. Even if the model and manufacturer of the references in both the transmitter and receiver are the same, if the two separated units each use their own

clock, physical differences in each clock from manufacturing imperfections, differences in temperature, and more ultimately lead to timing errors of different kinds that are not present in a traditional monostatic system.

Primary among these timing errors are frequency drift and jitter, which affect bistatic pulse-Doppler radar's measurements by shifting target range and Doppler frequency. The transmitting node of a bistatic system knows the precise time at which it transmits a pulse, but if the receiving side of the system does not know the start time, the perceived distance of a target will be shifted by the speed of light times the amount of time the receiver is off by. Similarly, if there is a difference in LO between the transmitter and receiver, the receiver will interpret that difference as a Doppler frequency, even if the target is stationary. Therefore, a synchronization system becomes necessary to keep the transmitter and receiver in line. Figure 1 shows the concept of frequency drift graphically. In this example, the receiving node's clock runs faster than the transmitting node, so the signal it receives appears to have a lower frequency. The difference in frequency can be seen when sampling the received signal at period intervals from the perspective of the transmitting node. This concept is at the heart of the frequency drift calculations made during this research effort.

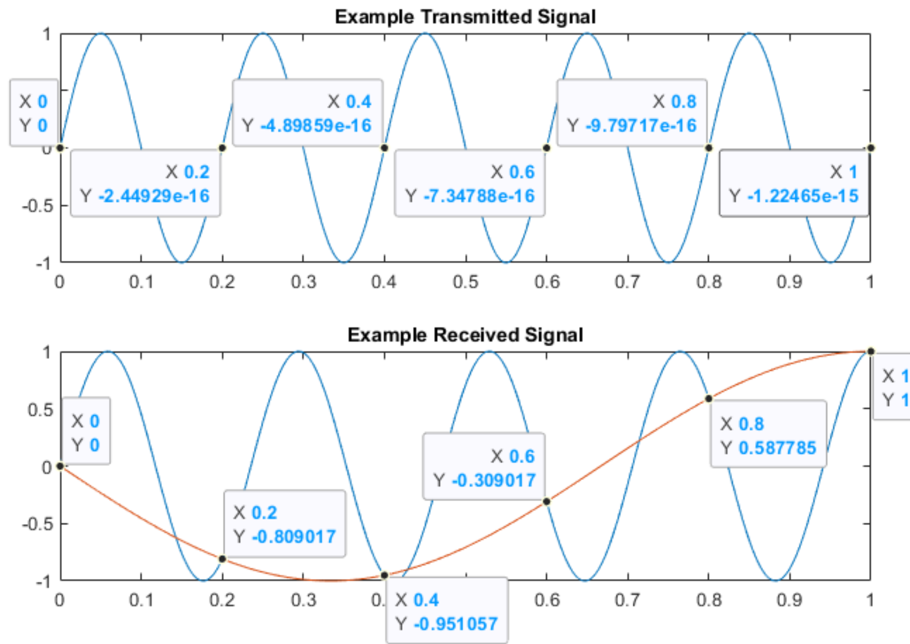


*Figure 1: Example of frequency drift between transmitted and received signals.*

Because the transmitter and receiver are physically separated and prone to frequency drift and timing jitter, not only is it important to synchronize the two systems to minimize relative drift between the two, but information about the frequency and jitter tolerances of the synchronizing subsystem are needed in post-processing to create the best possible error tolerance estimates.

## 1.2 Proposed Solution

This research effort aims to not only create a solution that synchronizes both frequency and absolute initial start time between arbitrarily separated transmitters and receivers of a bistatic radar system in order to address the issues described above, but also to characterize the frequency and jitter tolerances of the system used to synchronize the transmitter and receiver to mitigate the issues described above.

This research effort was designed around the Pentek Quartz Model 5950 Radio Frequency System on a Chip (RFSoC), henceforth referred to as the 5950. This Pentek board is powered by the Xilinx Zynq UltraScale+ RFSoC Field Programmable Gate Array (FPGA) and features an eight channel Analog to Digital Converter (ADC) and an eight channel Digital to Analog Converter (DAC), which are fully customizable using the onboard FPGA's block diagram, an editable visual representation of the FPGA's functionality, hard Intellectual Property (IP) (defined in Section 4.3.2), and example programs provided by the Pentek company.

As only one RFSoC was available, testing on two separate units was not possible, but because the overall goal was to synchronize a separate transmitter and receiver, a fully functional setup would include two 5950 units, one for transmitting and a separate for receiving.

Timing synchronization is provided by signals transmitted by Global Positioning System (GPS) satellites. Because the 5950 does not natively receive GPS signals[1], Ninja Precision Timing Modules by EndRun Technologies are used to synchronize the RFSoC. The 5950 accepts a 10 MHz reference signal that it uses to discipline its own internal clocks, and this reference signal is provided by a Ninja unit. In addition to the 10 MHz reference signal, the Ninja units have the capability of creating square waves of varying frequency, as well as single pulses at user-specified absolute GPS times. As discussed later in this thesis, these signals will be used in varying ways to synchronize the RFSoC with GPS time. Chapter 2 discusses in-depth experiments conducted to characterize signal stability, providing an end user with relevant signal tolerances that can be taken into account in performance characterization.

As there are two separate synchronization issues present in bistatic radar systems, the issues of frequency and time synchronization can be addressed separately. The 5950 includes the ability to bypass internally generated clock signals and synchronize its clocks to an externally generated 10MHz reference signal. This feature is tested and verified to work in the first signal characterization experiment (Section 3.1.5). Frequency synchronization is therefore solved 'out of

---

[1] The Pentek 5950 RFSoC's product description and datasheet mention native GPS support but do not go into depth on the matter, and evidence of GPS integration was not found in the 5950's block diagram or provided example programs.

the box' by the 5950, given a 10MHz signal with known frequency drift characteristics.

Chapter 3 discusses the other issue present in bistatic radar systems, that of timing synchronization, or synchronizing the transmitter's and receiver's start time triggers, as this issue is not solved out of the box. In summary, custom FPGA code was developed to convert a trigger signal from the Ninja units into internal triggers for the 5950's DAC and ADC to match a Coherent Processing Interval (CPI) pattern, with fixed PRI and duty cycle. To deal with a lack of support for trigger signals in the 5950's DAC (discussed in Section 4.6), an additional block of FPGA code was developed to provide basic trigger support, allowing the 5950's DAC to transmit a predetermined arbitrary waveform for a predetermined amount of time starting after the rising edge of its new trigger signal.

The problem of bistatic radar synchronization is not a new one, and researchers have discussed different methods of synchronizing separate bistatic transmitters and receivers. In his book "Bistatic Radar", Nicholas J. Willis discusses sending synchronization signal from transmitter to receiver over a variety of methods, such as landline or RF using direct or indirect line of sight, all of which either require extra infrastructure or could limit the locations at which the bistatic system us used [1]. Willis also briefly mentions using GPS to synchronize both signals, coincidentally the topic of this paper, but provides no other information on

this method. Other methods of timing synchronization discussed in published literature also employ GPS, using a 1 pulse-per-second (PPS) signal to synchronize two nodes of a bistatic system in time, frequency, and phase [3]. Another research team discussed a way to estimate frequency drift in post-processing by mixing the transmitted and received signals and applying a band-pass filter [4]. Efforts to implement multistatic radar for 3-D wind fields have led to the creation of passive multistatic radar systems whose receiving nodes are synchronized perfectly by signals sent from the transmitting node [5]. Although all innovative and valid methods of bistatic radar synchronization, the solution outlined through this research effort attempts to implement active, rather than estimated, synchronization without any restrictions on location or communications infrastructure.

## 1.3 Outline

The work in this paper involves characterizing time and frequency performance of an EndRun Ninja Precision Timing Module, specifically with respect to how it affects bistatic radar performance in relation to the timing and coherency issues discussed earlier in this chapter. The research this thesis describes also involves how to use the signals produced by the Ninja to discipline a Pentek Model 5950, modified to be able to transmit and receive signals in a pulsed radar pattern. In Chapter 2, the thesis will describe the characterization of three types of

signals produced by the Ninja with respect to timing and frequency issues, namely frequency drift of the 10MHz sinusoidal signal, timing drift of the 10 million pulses-per-second (MPPS) square wave, and the jitter of a single pulse generated at a specific GPS UTC time. After thorough characterization, in Chapter 3 the paper will explore a solution to reduce the aforementioned timing and coherence issues in undisciplined bistatic radar systems using the Ninja as a GPS-disciplined timing and frequency reference and a pair of Pentek 5950 units. Finally in Chapter 4, the thesis will discuss shortcomings of the currently implemented solution and outline areas of further development to address those shortcomings.

# Chapter 2: Signal Characterization

## 2.1 Motivation

In order to know the limitations of the reference signals being proposed for synchronization, the signals need to be measured, and characterized in terms of their timing errors. Because the 10MHz signal from the Ninja is the signal proposed to be used to synchronize the 5950's clocks, it is important to characterize this reference signal's stability in the frequency and time domains. In a bistatic system where both the transmitter and receiver are run on their own clocks, one clock's frequency drift with respect to the other results in a perceived Doppler shift, even if the target is stationary with respect to both the transmitter and receiver. Characterizing the frequency drift will allow for tolerances in Doppler frequencies to be accounted for when creating error estimates.

Although the 10MPPS signals from the two Ninjas' PPO ports are not planned to be used directly in the research effort's solution, this signal still provides valuable insight to the stability of the 10MHz signal. The two signals are derived from the same oscillator onboard the Ninja [6], therefore characterizing the timing shift of the 10MPPS signal will also characterize the 10MHz's drift in the time domain, rather than frequency domain. Characterizing timing is easier with a square

wave than with a sinusoidal wave, since pure or near-pure tones have little to no bandwidth compared to a square wave.

The absolute GPS time single pulse from two separate Ninja units are the signals proposed to be used as "starting pistols" for both the transmitter and receiver. The start time trigger signals provide absolute coherent processing interval (CPI) start time to the transmitter and receiver, and because error in the absolute start time of transmitter and receiver results in error in perceived target distance, the amount of jitter of these trigger pulses with respect to each other will define target distance tolerances.

This chapter discusses three experiments that were devised to characterize these three signals in relevant ways, including the equipment used, relevant settings for the equipment, the methods used to extract the characterizations from raw data, and discussions providing context for the results of the characterizations.

## 2.2 EndRun Ninja Precision Timing Module

### 2.2.1 Layout

The Ninja Precision Timing Module (PTM) can provide many output customized connections, but only a few were used in this research effort. Figure 2 shows a diagram of the available connections [7].

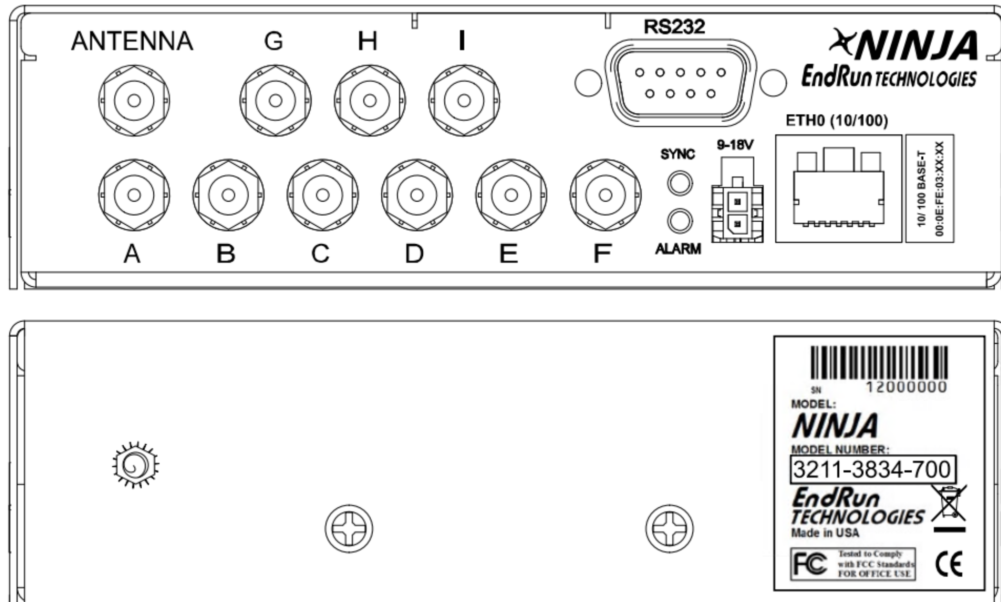


*Figure 2: EndRun Ninja Port Diagram*

Three of the available connections were used throughout this research. Port A was configured as a 10MHz output. Port G, the programmable pulse output (PPO), a versatile port used to output pulse trains of varying frequencies and duty cycles as well as the absolute GPS time single pulse, was configured as both a square wave and a GPS trigger. Port E was used as a 1PPS signal, where each pulse occurs as close to each GPS second as possible. The antenna port was also used in experiments where the Ninja units were disciplined by GPS.

### 2.2.2 Command Line Interface on Ninja and Commands Used

The Ninja is configured and operated using a command line interface (CLI) and runs the standard Linux command shell bash. This CLI can be accessed by serial port or SSH. EndRun provides a detailed list of commands in the Ninja user manual. The most commonly used commands in this research effort were:

- cpuio – Shows the status of the PPO port.
- cpuioconfig – Allows the user to change the frequency of the pulse train from the PPO port or to turn the port off.
- triggerppo – Allows the user to schedule an absolute GPS time single pulse.

Another use for the CLI on the Ninja units was running "Joe's Own Editor" (JOE). This text editor is loaded on the Ninja units by default and can be used to create bash scripts.

### 2.2.3 Bash Scripting

Like the 5950, the Ninja supports bash scripting to automate processes. Because the kinds of commands used in the Ninja CLI are limited, the only time bash scripting was used on the Ninja during the three signal characterization experiments was during the absolute GPS time single pulse jitter characterization experiment. The script developed for this experiment automated the process of

setting multiple single pulses on a pair of Ninja units using the triggerppo command so that the jitter between them could be characterized.

## 2.3 Signal Acquisition with the 5950 RFSoC Module

### 2.3.1 Linux Command Line Interface

The 5950 features a Linux-based CLI accessible by serial port or Secure Shell (SSH), allowing the user to easily interact with the 5950. The CLI allows the user to create bash scripts, connect to other computers on a local network, edit initialization files, and run programs.

### 2.3.2 “acquire” Example Program and Initialization Files

Pentek included four example programs, written in C, with the 5950. The first three, “acquire”, “acquireEth”, and “dacquire”, handle different use cases of the ADC and DAC onboard the 5950. The fourth example program, “show_info”, provides the user information about the 5950. The program “dacquire” is discussed in Chapter 3, as it was not used in the signal characterization experiments but was used later in this research effort.

The “acquire” program builds on an extensive Board Support Package (BSP), which contains hundreds of C and C header (.h) files that interface with the

RFSoC's FPGA to access the 5950's ADC data through the 5950's Direct Memory Access (DMA) cores. The original "acquire" example program shipped by Pentek was used during all three characterization experiments.

While the C code for "acquire" is included with the BSP to allow the end user to modify its operation, the program itself relies heavily on hard-coded and undocumented memory locations to interface with the FPGA, making it difficult to understand the full details of various commands. A user can, however, modify how "acquire" operates through initialization files and command line arguments. Initialization file arguments, all defined in the BSP User's Manual, that were modified to meet the signal acquisition needs of this research effort include:

- chanmask – Specifies the ADC channel(s) to be used.
- loop – Specifies the number of individual data buffers to be captured.
- xfersize – Specifies the size of individual buffers in bytes. (Sample data size was set to 16 bits/sample consistently throughout the research effort)
- numbuf – Specifies the number of DMA buffers of 'xfersize' each to be created for use in a circular buffer.
- brdfreq – Specifies the board clock frequency.
- adcfreq – Specifies the frequency of the ADC clock, which is derived from the board clock.

- adcgatesrc – Specifies which input provides the trigger signal to the program, where a trigger signal initiates the capture of an individual data buffer.
- wfile – specifies the name of the file created. Because multiple files were created per experiment, this argument was only modified in bash scripts to serialize file names.
- wsize – Specifies the number of bytes to be saved to the file whose name is specified by 'wfile' per loop count. This is distinct from the total number of bytes captured as determined by 'xfersize', but cannot exceed the value set for 'xfersize'.

Unlike the FPGA code and "acquire" example program, custom initialization files were developed for each characterization experiment to meet the specific data collection needs of the experiment.

### 2.3.3 Bash Scripting and Automating File Transfer

Running BASH scripts on the 5950 module proved to be an invaluable part of the signal acquisition process. As shown in the results of the three timing characterization experiments discussed below, each experiment involved hundreds of individual signal acquisitions and transfers to the PC used to analyze the signals over Trivial File Transfer Protocol (TFTP). Ideally, these acquisitions would be

evenly spaced apart, and automating the acquisition and transfer process allows acquisitions to be as evenly spaced as possible. Additionally, because bash scripts allow the use of variables set during runtime, running “acquire” from a bash script rather than manually allows for much easier overrides of the initialization file than manually appending all command line arguments to the end of the executable call. Figure 3 shows the length of the command line argument needed to call “acquire” without the bash script, while Figure 4 shows the bash script guiding the user through different setup options before calling “acquire” itself.

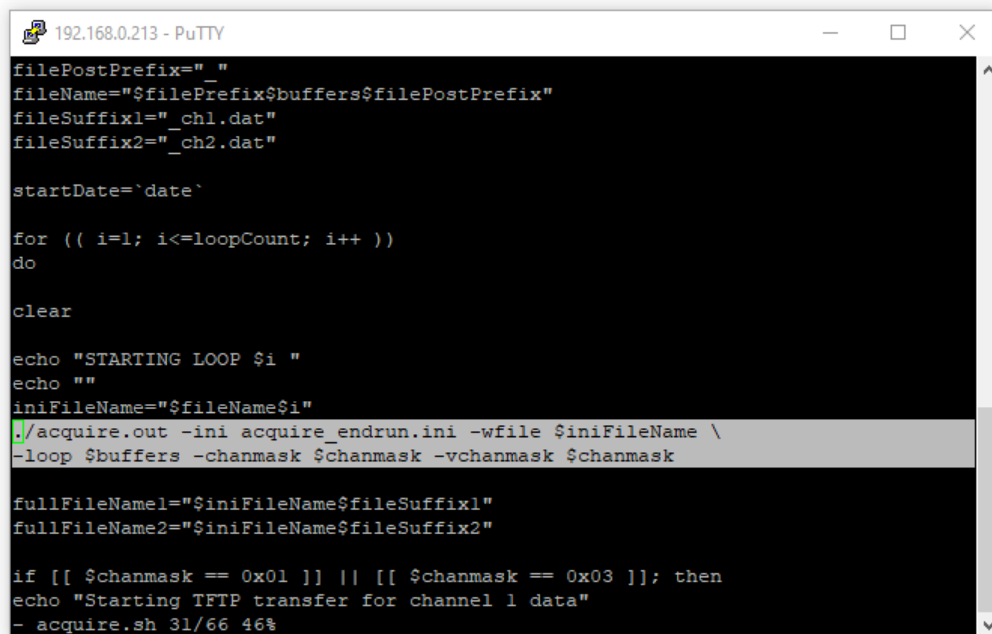


*Figure 3: A sample from the bash script used in the first characterization experiment showing the length of the call needed to start “acquire” without using a script.*

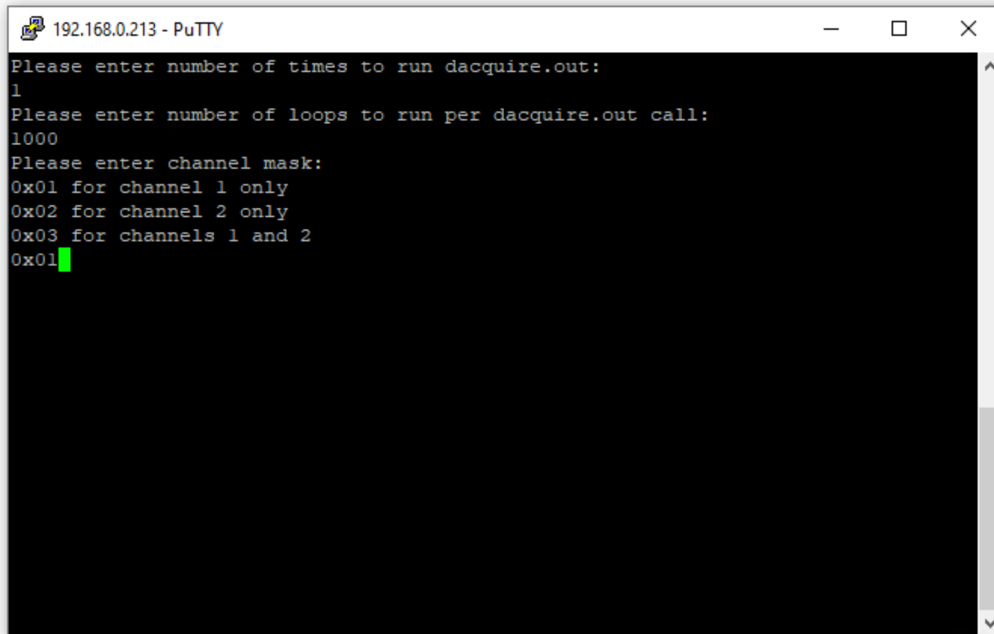


*Figure 4: UI created by bash script for easy modification of "acquire" operation.*

The use of bash scripting in the following three characterization experiments resulted in greater uniformity of the time between signal acquisitions and an extremely reduced workload in running the experiments. Greater uniformity of time between signal acquisitions leads to more accurate depictions of the data points obtained from signal analysis. The results shown later plot data points evenly spaced, as the time the signals used to create each data point were captured was not recorded. Unlike the initialization files, all three characterization experiments used

nearly identical bash scripts, since the bash scripts allowed for were set during runtime rather than hard-coded.

# Chapter 3: Signal Characterization Results

## 3.1 Frequency Drift Characterization

As discussed in Section 1.1, a combination of real-time synchronization between signals and knowledge of frequency tolerances is critical in accurately measuring Doppler velocities of targets. The real-time synchronization of separate transmitter and receiver clocks are discussed in Chapter 4, but the signals used to achieve real-time synchronization are discussed here. Specifically, the 10MHz signals generated by two GPS-disciplined Ninja PTMs were captured using a single Pentek 5950 and analyzed for frequency drift in MATLAB.

### 3.1.1 Experiment Objectives

The first experiment was designed to characterize how much the 10MHz signals produced by both Ninja units drift in frequency with respect to each other. Mean and standard deviation characterize whether the drift has an offset and how much the drift varies within the timeframe of the experiment. These characteristics are relevant to the end user, as they can be used to quantify error statistics when using a Ninja to discipline a radar system in the field.

### 3.1.2 Challenges

Collecting enough of the required data at a high enough sample rate to analyze for a minute frequency drift presented several challenges, and a significant amount of time was spent on hardware and software configurations that ultimately were not able to provide the data needed.

The first attempts made at characterizing frequency drift employed a brute force strategy. The idea was that if enough continuous data is captured at once, that the frequency drift could be easily observed in the Fourier transform of the collected data as a secondary peak after the much larger 10MHz peak. The issues with this approach soon became clear, as the Tektronix DPO 70604 oscilloscope used could not capture and transmit the required amount of data over the VISA over Ethernet interface implemented in MATLAB to be able to discern such a small frequency in the frequency domain. Anti-aliasing features on the oscilloscope also prevented us from observing the frequency drift directly by undersampling the signals.

The next approach was to take short, coherent data captures using the oscilloscope and a modified MATLAB script that prompted the oscilloscope for those data captures at defined time intervals to maintain coherence. The aim of this method was to determine the phase of each data capture compared to a MATLAB-generated 10MHz sinusoidal signal. The phases for each data capture would form a new signal, the frequency of which would be the frequency drift $\delta f$ between the

two signals. Figure 1 displays this concept on a much smaller scale. In that example, the receiving node's clock runs 0.75 Hz faster than the transmitting node's clock. The transmitted signal was 5 Hz, so the signal that the receiving node captured appeared to be 4.25 Hz. This 0.75 Hz offset can be shown by sampling the received signal at the frequency of the transmitted signal. For this signal characterization experiment, however, the signals being characterized are several orders of magnitude larger than the expected frequency difference. It would be impractical to sample one ~10 MHz signal at the other signal's ~10 MHz frequency to discern a sub-1 Hz frequency difference. Instead, the MATLAB-generated signal was used to create a matched filter for the captured signal. This would reveal how the phase of each captured signal compared to the ideal signal over time. Rather than sampling the imperfect captured signal at the transmitted signal's frequency, this effectively samples the captured signal at a multiple of the transmitted signal's frequency. This approach ultimately failed on the Tektronix oscilloscope mentioned earlier as well as on a newer Agilent Infiniium DSO80804B oscilloscope that also supported VISA commands over Ethernet. The underlying issues seemed to be related to the amount of time needed for the oscilloscopes to receive and process the commands being longer than the timing requirements for the data collections.

After determining that programming an oscilloscope for coherent data capture was neither practical nor likely the best option, our attention shifted to the

5950 RFSoC as a means for data collection and transmission, due to the onboard FPGA's high sample rate and ability to acquire and store much larger amounts of data than the oscilloscopes previously being used. However, a challenge with preparing the 5950 for the manner of data acquisition needed to characterize frequency drift was the lack of documentation for the options available when running the onboard data acquisition program "acquire". An appropriate sample rate of 250 MHz was selected, but when this value was used for the overall board's frequency in the initialization file used to customize the operation of "acquire", collected data did not appear at the correct frequency. It was not until Pentek support was contacted that it was learned that the board clock rate must fall between 1 and 4GHz inclusively, but the ADC clock could in fact be set to 250MHz. Such information was not included in the documentation for the "acquire" program or the 5950, in general, and several other options listed in the initialization file were either ill defined, had no supporting documentation (including value limits), or both.

### 3.1.3 Setup

The 10 MHz reference signals from two Ninja units were connected to channels 1 and 2 of the 5950. The first Ninja's 10MHz output was also connected to the 5950's clock input to discipline the 5950's internal board clock. As shown in

Figures 8 and 9, using this reference signal locks the frequency drift results for the first Ninja around 0, allowing the difference between the two Ninjas to be more easily observed. Finally, the first Ninja's PPO output was used as an experiment trigger; hence, it was connected to the 5950's trigger (TRIG) front panel input. The Ninja's PPO setting was set to 100PPS so that data captures would occur every 0.01 seconds.

The initialization file for this experiment was similar to the initialization files used to characterize other signals from the Ninja units. The relevant values set in the initialization file for this experiment were:

- loop is set to 1000.
- xfersize is set to 65536 bytes.
- numbuf is set to 1000.
- brdfreq is set to 1GHz.
- adcfreq is set to 250MHz.
- wsize is set to 65536 bytes.

These settings were chosen to meet the needs of the experiment. A balance was needed between high-resolution data and data file sizes that were practical to transfer and analyze, so a board frequency and an ADC frequency of 250MHz were chosen, along with transfer and write sizes of 65536 bytes. The number of loops

and buffers being set to 1000 allowed the 5950 to capture 1000 individual data captures per overall call of “acquire” without the DMA running out of space.

### 3.1.4 Signal Analysis

Analysis of the signals’ relative frequency drift with respect to each other relies on the 100PPS trigger signal supplied to the 5950 by the first Ninja and can be thought of in terms of the idea of fast time and slow time used in monostatic pulsed radar. The 100PPS signal can be thought of as slow time, signaling a capture of 32768 samples every 0.01 seconds, the equivalent of a pulsed radar’s PRI. The ADC’s 250MHz clock, which triggers each individual sample capture, can be thought of as fast time. Each 1000 sets of 32768 samples were correlated with a complex ideal 10MHz signal in MATLAB, which served as a downconversion and low pass filter of the data. This created a 1000-point dataset for each channel representing the phase of each of the 1000 data captures with respect to the ideal 10MHz. The FFTs of each phase data are taken, and the peaks of the FFTs are recorded as the dominant frequency offset from 10MHz. This is the same process described in Section 3.1.2. If the peak of the FFT is at 0 Hz as shown for channel 1 in Figure 5, then the $\delta f$ of that signal is 0Hz. Channel 2 in Figure 5, for example, would have a $\delta f$ value of approximately -3 millihertz.

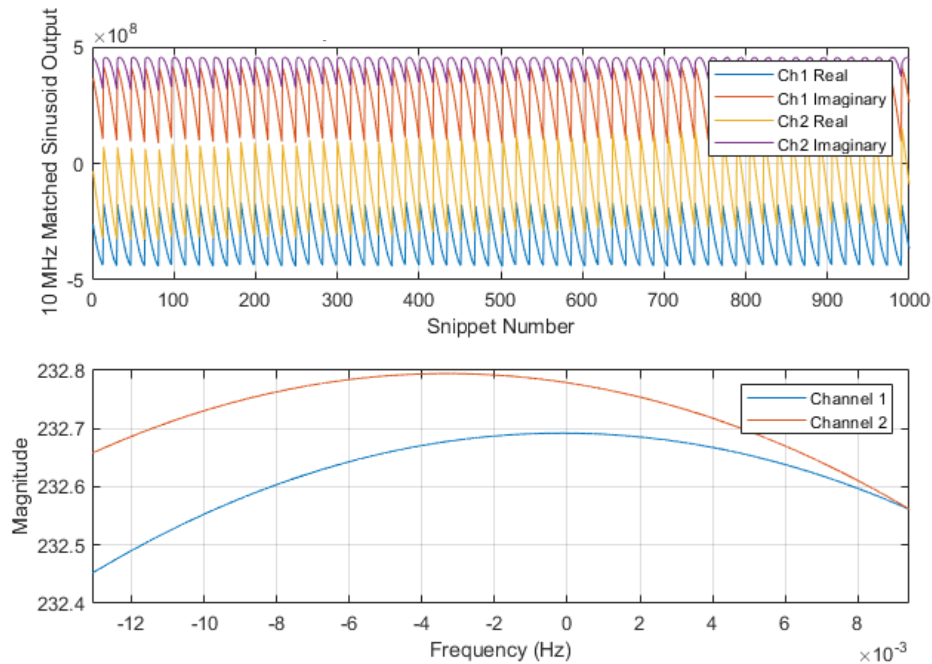


*Figure 5: An example of jagged I/Q results from a data and a close zoom of the peaks of each channels' I/Q FFT.*

Figure 5 shows an example of phase data from the matched filter process. The FFT of the phase data shows that while the frequency of channel 1's phase is centered around 0 Hz, channel 2's phase data is centered around approximately -3 Hz. When jagged phase results such as those shown in Figure 5 were discovered, the conclusion was made that the RFSoC required arbitrary amounts of time between data captures to write the data from memory into a file. [8]. These arbitrary amounts of time caused the data collection to be incoherent, causing the phase data

to appeared jagged, just as they do in Figure 5, rather than a smoother slowly oscillating line corresponding to δf, as was originally expected.

Because the goal was to gain phase information from the matched filter process, this seemed to present an obstacle in the data collection process, unless it could be shown that this slightly incoherent phase information could still produce reliable δf results. After another data collection on the RFSoC using an external signal generator as a trigger also showed similar coherency issues, a manual test was done to determine to what extent data captured slightly incoherently produced predicted results in a more controlled environment. The test consisted of the 5950 acquiring a 10MHz signal from an external signal generator, but the frequency was manually adjusted by 2Hz between every set of "coherent" data captures. The phase and frequency results of this experiment are shown in Figure 6:

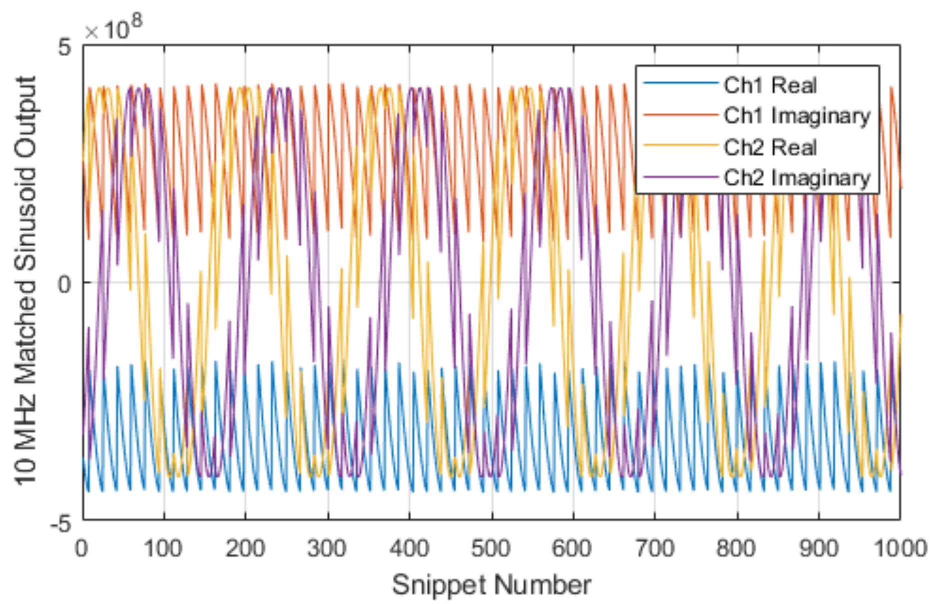


*Figure 6: Manual frequency drift I/Q results at a 101 Hz offset.*

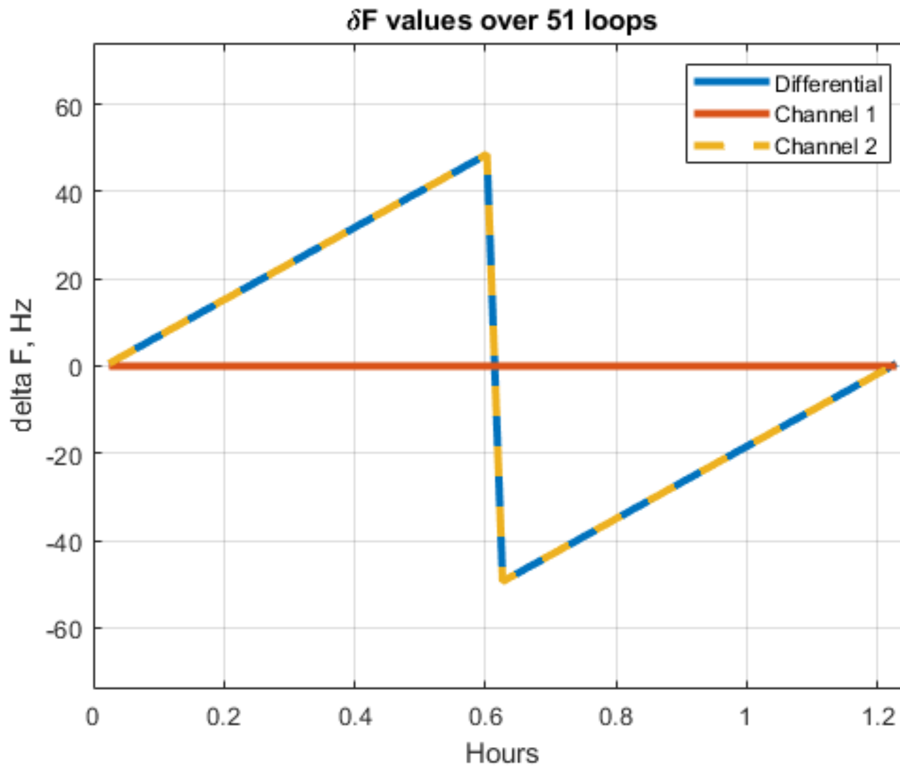


*Figure 7: Manual frequency drift results.*

Figure 6 displays the phase data from the final offset tested, which was 100 Hz. Despite the jaggedness of both channels' phase data, the overall flatness of channel 1's phase compared to the sinusoidal nature of channel 2's phase data help show the validity of this method. The coherency issues are still present, but the manual frequency drift using the same matched filtering process described above clearly shows the correct frequencies (accounting for aliasing) in Figure 7, which shows the frequency drift observed throughout the frequency sweep. Each data

point was found by calculating the peak of the FFT of that frequency offset's phase results. The data shown in Figure 7 aliases at ±50 Hz. This experiment verified that frequency drift information can still be reliably captured from slightly incoherent data captures, and this method of determining frequency drift from slightly incoherent captures was determined to be valid.

### 3.1.5 Results

Initial versions of this experiment were run before either Ninja unit were ever connected to their GPS antennas or locked on to a GPS signal. Figure 8 shows a clear offset of approximately 3.77 millihertz with a standard deviation of 0.406 millihertz and a change in drift over the course of over 30 hours can be seen between the two Ninjas. The 5950's clock was referenced to the 10MHz signal from the first Ninja, so Figure 8 only provides information about the relative drift between the Ninjas, rather than information about the absolute drift of either of the two. Spikes that occur at the same time in both channels' results are believed to be the result of timing issues internal to the 5950, as the two Ninja units were not directly connected throughout the entire experiment.

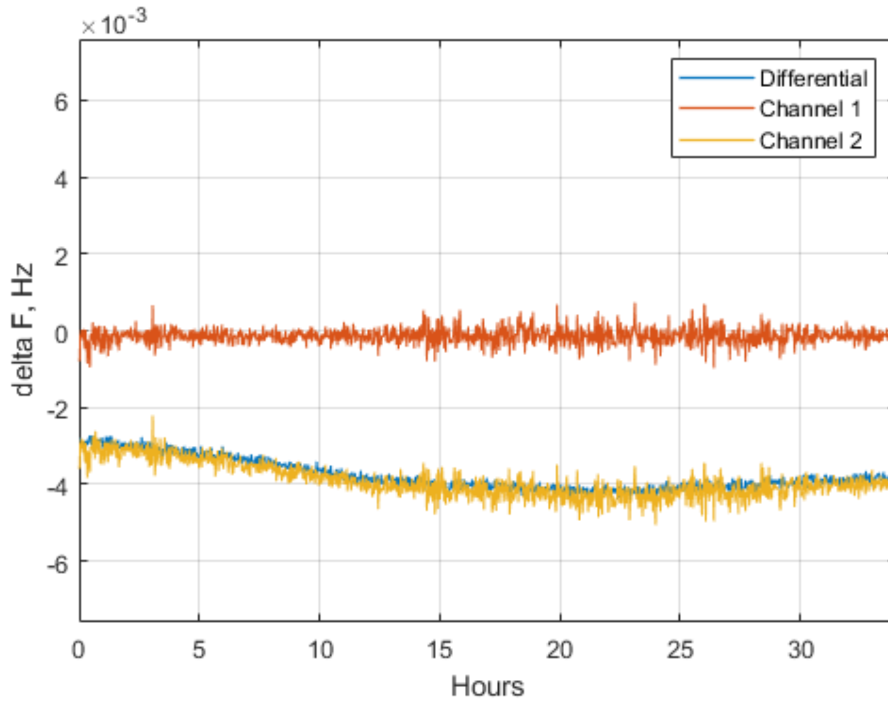


*Figure 8: Frequency drift results for two Ninja units operating without GPS discipline.*

In addition to characterizing the frequency drift of undisciplined Ninja units, Figure 8 provides an additional piece of information crucial to the overall research effort: verification that the 5950 supports clock disciplining for both its ADC and DAC out of the box. The default setting for brdclksrc (Board Clock Source) in the original initialization file for "acquire" provided by Pentek is "int", which stands for internal. This setting, according to the initialization file, instructs the 5950 to generate a clock signal internally and to use an external reference clock to discipline it if available. An external clock reference signal was available to the

5950 during this experiment and the setting for brdclksrc was left unchanged, but the assumption that the 5950 uses this clock reference in the advertised way cannot blindly be made. As the reference clock was the same 10MHz signal connected to the 5950's channel 1, if the 5950 did in fact use the clock reference to discipline its own board clock (from which the ADC and DAC clocks are derived) one could expect the $\delta f$ results of that channel to appear locked at 0Hz, before accounting for random error. This is, in fact, exactly shown in Figure 8. There was nothing different or special about the Ninja connected to Channel 1. Because this run of the experiment shows that undisciplined Ninjas are prone to visible frequency drift, if the 5950 used its own internally generated board clock that was in no way influenced by either one of the Ninjas, it would be expected that the $\delta f$ values for neither of the Ninjas would be locked at 0, which is not the case. Verifying that the 5950's board clock can be disciplined out of the box solves one of the two issues with bistatic radar that this research effort seeks to solve.

Once the Ninjas were connected to their antennas and locked on to a GPS signal, the frequency drift experiment was repeated. Figure 9 shows the difference that GPS disciplining can make on frequency drift. The 5950 was locked on to the Ninja connected to channel 1, just as in the previous run of the experiment without GPS discipline.

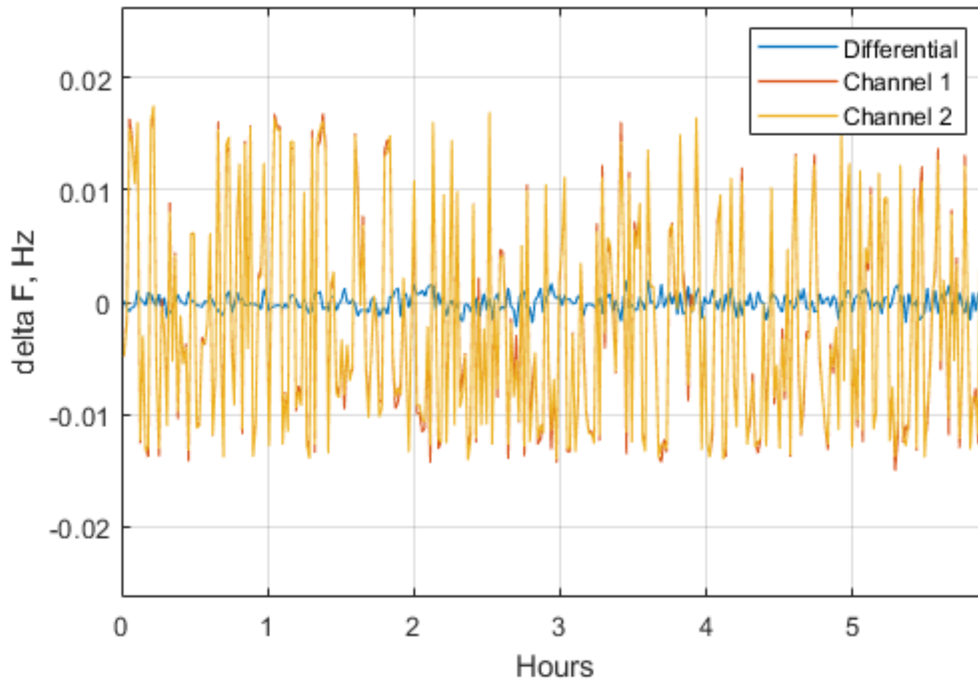


*Figure 9: Frequency drift results for two GPS-disciplined Ninja units.*

Although the duration of the experiment was approximately one seventh of the previous undisciplined experiment, the differences are immediately visible., with both Ninjas' δf results appearing to be locked at approximately 0 Hz with no clearly visible change in drift over time, as was seen in Figure 8. The average offset between the two was approximately 29.8 microhertz, and the standard deviation of the relative drift between the two was approximately 0.792 millihertz. As seen in the undisciplined run of the experiment, the spikes occurring in the results for both channels is also seen here and is again believed to be caused by timing issues within the 5950. Because the signal connected to channel 1 is also the 5950's board clock

reference, its δf results should ideally be 0 Hz. Results showing otherwise would mean some mismatch internal to the RFSoC between the 10 MHz input reference and the ADC clock. Finding the standard deviation of channel 1's δf results, therefore, should provide insight as to how much phase jitter can be expected on a 5950. When using a single 5950 as a monostatic radar, this phase jitter should not present much issue – as shown in Figures 8 and 9, both channels' results experience the same jitter, so subtracting the two results should remove it. When using two separate 5950 units in a bistatic radar configuration, however, this jitter cannot be assumed to be the same between the transmitter and receiver. The standard deviation of channel 1's results is 9.51 millihertz, which is significantly larger than the standard deviation of the relative frequency drift between the two Ninja units.

Since 10 MHz signals in a radar system can be scaled up, discussing these frequency drift results scaled up is relevant. For the undisciplined results, scaling the ~10 MHz signal with ~3.8 mHz of error to a 3 GHz signal would result in approximately 1.14 Hz of drift on average. Scaling the disciplined results from 10 MHz to 3 GHz would result in approximately 9 mHz of drift on average. Taking the jitter within the RFSoC, the standard deviation of frequency drift in the disciplined case would be approximately 2.9 Hz of drift. On the order of 3 GHz, these frequencies would barely be noticeable whether discipline by GPS or not, but the GPS-disciplined results on average show much lower drift.

## 3.2 Timing Drift Characterization

### 3.2.1 Experiment Objectives

Although the 10MPPS signal is not used directly in the solution developed to synchronize transmitting and receiving nodes of a bistatic radar system, understanding the limits of this signal provide more understanding of signals that were in fact used. As the 10MPPS signal created by the EndRun is tied to the 10MHz signal characterized in Section 3.1 through the Ninja's internal main oscillator [6], characterizing timing errors of this signal in the time domain will aid in understanding the 10MHz signal's timing stability.

### 3.2.2 Setup

Because the 10MPPS signal created by the Ninja units is accessed through the unit's PPO port, the 1PPS signal (port E) on Ninja 1 was used as the RFSoC's ADC trigger. Ninja 1's PPO port was connected to the 5950's first ADC channel, and Ninja 2's PPO port was similarly connected to the 5950's second ADC channel. Both Ninjas' PPO ports were set to the 10MPPS option, as this was the signal being analyzed.

The initialization file for this experiment was similar to the initialization files used to characterize other signals from the Ninja units. The relevant values set in the initialization file for this experiment are:

- loop is set to 1.
- xfersize is set to 65536 bytes.
- numbuf is set to 1000.
- brdfreq is set to 4GHz.
- adcfreq is set to 4GHz.
- wsize is set to 65536 bytes.

These settings allowed the 5950 to capture large amounts data at its highest sampling rate, without sacrificing the rate at which captures occurred for unnecessarily long captures. As the signal analysis methods compared the two Ninjas' signals within one capture rather than over the course of multiple captures, as was the case in the previous experiment, the number of loops of data capture per "acquire" call did not need to be higher than 1. Before the number of buffers was set to 1000, the experiment often crashed due to the 5950's DMA running out of space.

### 3.2.3 Signal Analysis

As the goal of this experiment is to determine the stability of the Ninja in the time domain and the signals being analyzed are of the same shape (a square wave at 10MPPS), cross-correlation was used to analyze the signal.

Just as with the frequency drift characterizations, data was processed loop by loop to ultimately create a set of datapoints (in this case shown in Figure 13), each of which represents the results of the analysis for that loop. To generate the cross correlations for analysis, both data captures for each loop were loaded into MATLAB. In MATLAB, each set of signals from each loop were cross-correlated to quantify the relative delay between signals. One example cross-correlation output is shown in Figure 10, followed by all cross correlations across all loops in Figure 11. Although not visible in Figure 11, the maximum value for each loop's cross correlation was recorded in an array, shown in Figure 13.

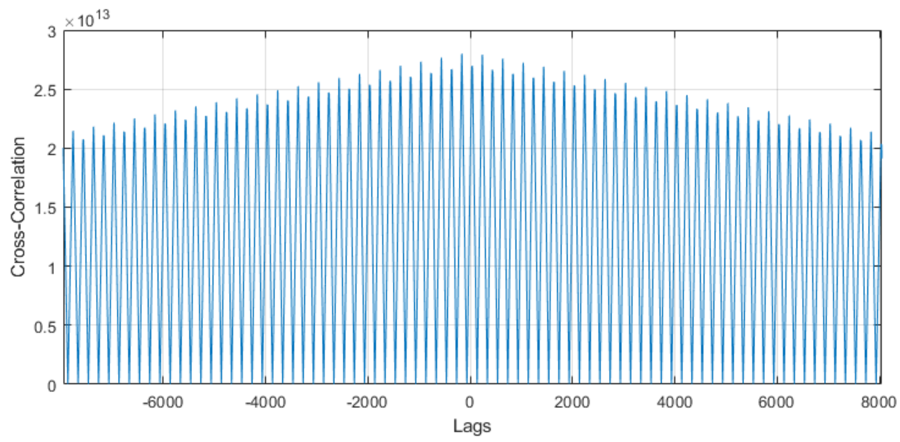


*Figure 10: The results of cross-correlation between a single set of 10MPPS signals captured from two GPS-disciplined EndRun units*

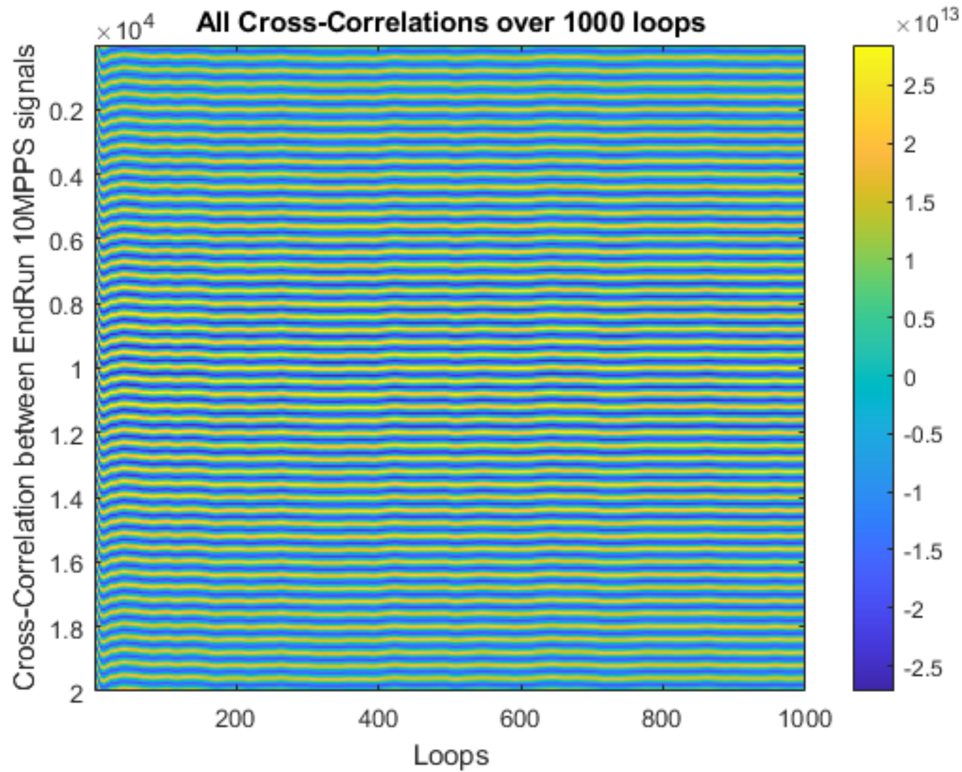


*Figure 11: All 1000 cross correlations generated from data acquired by the 5950.*

### 3.2.4 Results

Similar to the frequency stability experiment, first iterations of this experiment occurred without GPS disciplining on the Ninja units to allow for the timing stability of undisciplined Ninja units to be analyzed. Figure 12 shows the results of one such experiment.

:

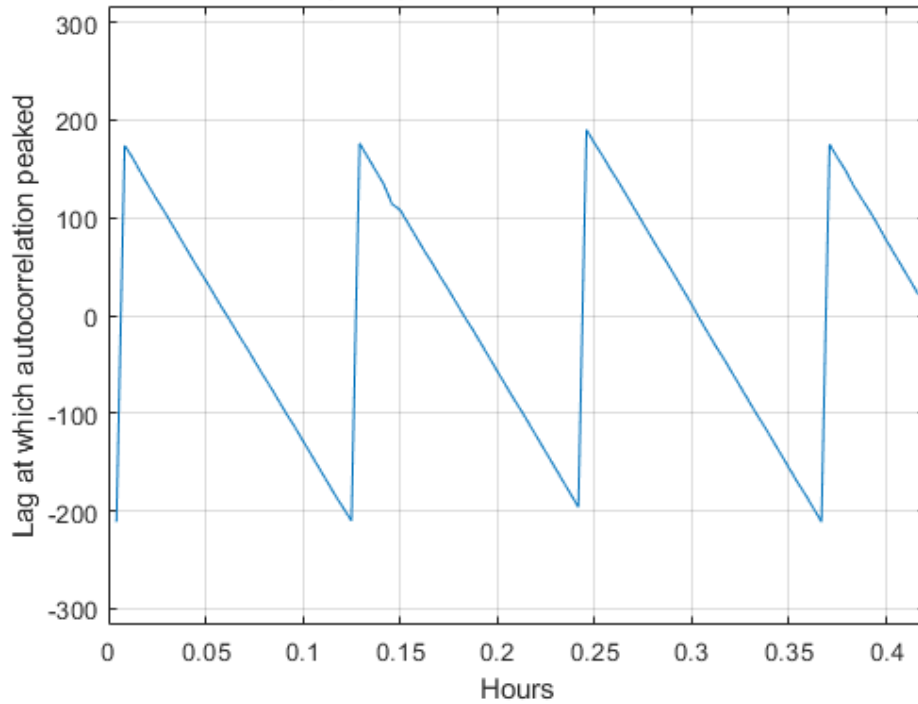


*Figure 12: The highly aliased results of a timing drift experiment done between the two Ninja units without GPS discipline.*

The relative stability of the second Ninja's 10MPPS signal with respect to the first's is visibly low, as indicated by the severe slope and the aliasing, caused by the periodic shape of the waveforms. The period of the PPS signal is 0.1 microseconds, and as the drift cycle repeats approximately every 0.08 hours, the rate of drift is calculated as approximately 0.35 nanoseconds/second, resulting in approximately 5 millimeters of perceived target movement over a 50ms CPI.

Figure 13 shows the results of an iteration experiment done with GPS disciplining, with Figure 14 highlighting the first hour of the same results for the purposes of examining the Ninjas' speed in locking on to a GPS signal.

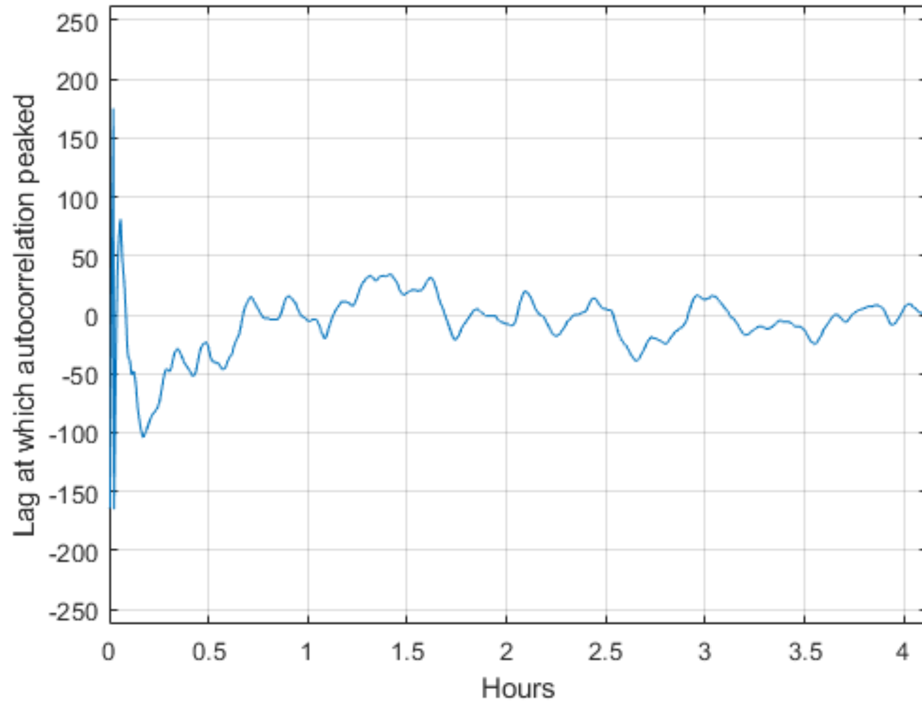


*Figure 13: The lag within each individual cross-correlation at which the cross-correlation value was highest.*

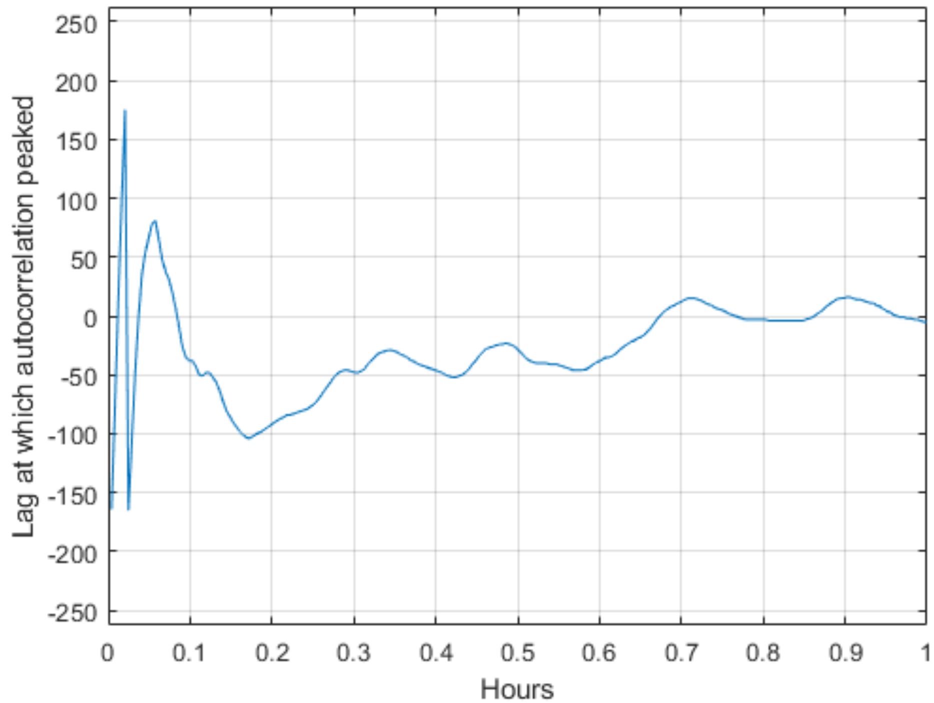


*Figure 14: The first hour of the above results shown in more detail.*

The severe drift during the first 4 minutes of the experiment, seen in Figure 14 is due to the Ninja units not being connected or locked on to a GPS signal. The choice to wait to connect the Ninja units to GPS until after the experiment started was made to characterize the length of time required to lock on to a GPS signal. Although the Ninja units connected to a GPS signal within approximately 4 minutes, it took the units approximately 15 minutes in ideal weather with ideal GPS constellation visibility to lock on well enough to stabilize drift, with even better results after 45 minutes, despite the Ninja user manual stating that users can expect

lock times of around 10 minutes [7]. Although the time the Ninjas took to fully lock on to a GPS signal does not directly affect the results of the experiment, it is an important note for future systems using a Ninja unit. After determining the time at which the Ninja units were reliably locked on to a GPS signal, it is then possible to isolate the stabilized data for analysis.

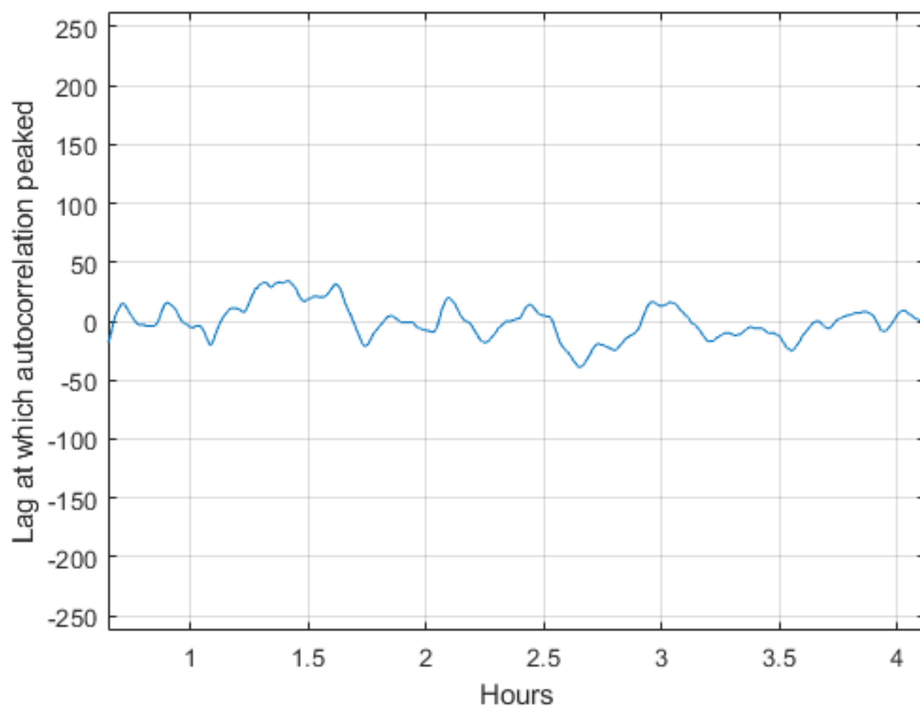


*Figure 15: Maximum Lag in the 10MPPS signal after reliable GPS lock was achieved.*

Figure 15 shows the stabilized timing drift data. Compared to Figure 13, the data are much more stable. Because the sampling frequency of the data is 4GHz, a

lag of 1 correlates to a time offset between the pulses of 0.25 nanoseconds. Given an average and standard deviation of lag shown in Figure 15, the two signals were offset on average by 19.208 picoseconds, with a standard deviation of 3.7209 nanoseconds over a timescale on the order of hours. Compared to the above-mentioned CPI timescale of 50ms, timing drift of two Ninja units reliably locked on to GPS signals is not something likely to make any difference during a CPI.

## 3.3 Absolute GPS Time Single Pulse Characterization

As described in Section 1.1, the ability to synchronize the start times for the transmitter and receiver of a bistatic radar system both in real time and in post-processing is required in order to correctly estimate a target's distance.

### 3.3.1 Experiment Objectives

This signal characterization experiment aims to determine the drift of relative jitter between two Ninja units scheduled to produce absolute GPS time single pulses, characterized by the standard deviation of all jitter observed in data captures. Standard deviation of jitter is relevant to this research effort, because it shows how stable the trigger for the CPI (this signal's intended use in this research effort) can be expected to be.

### 3.3.2 Setup

The single pulses are a feature of the Ninja's PPO port (port G), a versatile port also used as the trigger for the first experiment and the 10MPPS signal analyzed in the second experiment. Ninja 1's PPO port is connected to the first ADC channel of the 5950, and Ninja 2's PPO port is likewise connected to channel 2 of the 5950's ADC.

This experiment was the only to utilize the Ninjas' support for bash scripting. The script run on the Ninjas automated setting their PPO triggers, and was carefully timed so that the total number of triggers scheduled on a Ninja at any one time would stay relatively constant. If the frequency at which the triggers were set was much larger than the frequency the triggers themselves occurred, the Ninjas would run out of memory and crash, ending the experiment prematurely. If the rate of setting triggers was lower than the frequency at which the triggers occurred, the Ninjas would run out of triggers, and the experiment would similarly be ended prematurely. The script relied on the Ninja's triggerppo command, as well as hard-coded time values updated in the script before running the experiment.

Because the PPO output on the Ninja is the only port able to create a single pulse, the PPO port on Ninja 1 was also used as the ADC trigger for the 5950. Originally, the 1PPS signal was used as the trigger signal, but this resulted in several issues. The primary issue resulted from timing delays on the single pulse,

as reported in the Ninja terminal. Although the single pulses are scheduled to occur on the GPS second, the pulses were consistently delayed by approximately a third of a second, while the 1PPS signal's rising edges occurred faithfully on the GPS second. This third of a second delay resulted in the ADC capture missing the single pulses entirely unless settings in "acquire"'s initialization file were used that made the data captures far too large to transmit over TFTP in a reasonable amount of time. Secondly, single pulses can only be scheduled a minimum of two seconds separated from each other, according to the Ninja user manual. Because the trigger rate was twice this minimum spacing, half of the captures occurred at seconds the Ninjas were not scheduled to produce single pulses. To alleviate these two issues, the decision was made to use a splitter on the first Ninja's PPO output and for each of cables carrying the single pulses to the ADC be approximately 50 feet. This length of cable creates a delay long enough to allow the ADC to process the trigger and start the capture before the pulses arrive. Because this added delay was slightly different for each Ninja due to a lack of uniformity between cables available, the true mean of jitter between the Ninja units is not measurable with this setup. However, finding the mean jitter was not the goal of the experiment, finding the drift in jitter was. Because the length of cables and therefore delay due to the cables did not change between captures, the differences in jitter would therefore have to come from the Ninja units themselves.

The initialization file for this experiment was largely similar to the other two initialization files, and the relevant variables are:

- loop is set to 1.
- xfersize is set to 268,435,456 bytes.
- numbuf is set to 4.
- brdfreq is set to 1GHz.
- adcfreq is set to 62.5MHz.
- wsize is set to 268,435,456 bytes.

These changes reflect the signal acquisition needs of the experiment: Capturing one loop per trigger, long enough to capture both pulses should they drift significantly, and at a low enough resolution to practically be able to transfer and process the resulting files. Although the 268 Megabyte value for both the transfer size and write size could have realistically been decreased, the choice was made through the trial-and-error process, as initial attempts at this experiment did not result in both pulses being captured.

### 3.3.3 Signal Analysis

Figures 16 and 17 show the signals captured by the 5950 in a setup configured as described above, with the only difference between the two figures being scale on the x-axis. The general shape and amplitude of the waveforms shown

in the two figures is a typical representation of all waveforms captured in this experiment.

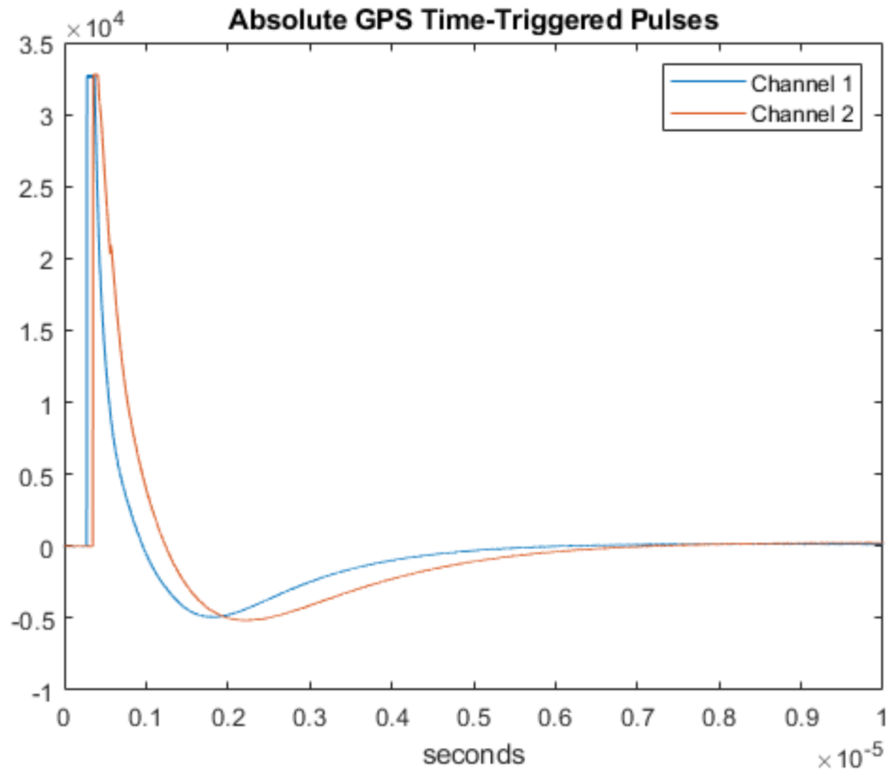


*Figure 16: A sample of "simultaneously" triggered single pulses from each EndRun.*

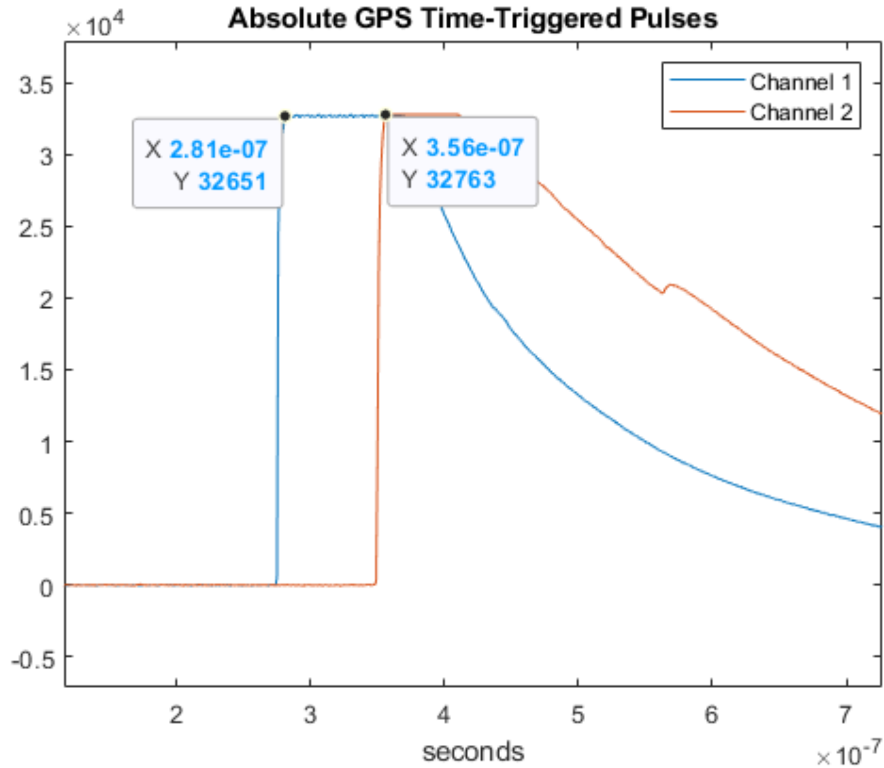


*Figure 17: Highlighting the precise time between each of the pulses displayed above.*

Analyzing the waveforms to determine standard deviation of jitter between the two signals was straightforward. The initial plan to measure jitter was to identify the points shown in Figure 17 algorithmically in MATLAB and determine how far apart they are on the x axis. Given the slight variation in maximum and slight noise, the algorithm developed to find this first maximum point for each signal proved unreliable. Instead, the difference in time between pulses was measured using slope. The slopes of the rising edges of both channels are at their maximum in

approximately the center of their rising edges, and this was found to be true for all pulses observed in the experiment. Therefore, using the gradient command in MATLAB on the entire waveform and locating the maximum of this new array would reliably and accurately locate not necessarily the “first maximum” point shown in Figure 17, but instead the same point of the rising edge in each waveform. Once the index of this maximum point was determined, the index was divided by the sampling frequency of the ADC to determine the time at which the maximum occurred. Subtracting the two times from each signal then provides the jitter relative to each other. The relevant MATLAB scripts for this experiment can be found in the appendix.

### 3.3.4 Results

Figure 18 shows the results of this experiment. As noted in Section 3.3.2, the mean jitter is not a relevant or valid measurement, because the slightly different lengths of cables connecting the Ninjas’ PPO ports to the 5950’s ADC create slightly different constant delays for the signals being measured, and because the scope of this experiment was limited to drift in jitter. However, as this difference in delay due to differing cable lengths is constant, the difference in jitter observed can be attributed to trigger instability within the Ninja. Using this signal as a trigger for a bistatic radar system, one could expect a standard deviation of approximately

1.8 nanoseconds of jitter between two Ninja units when both Ninja units are GPS disciplined. Translating this value to target range using the speed of light, error in target range due to using this signal as a CPI trigger alone could be expected to have a standard deviation of approximately 0.54 meters.

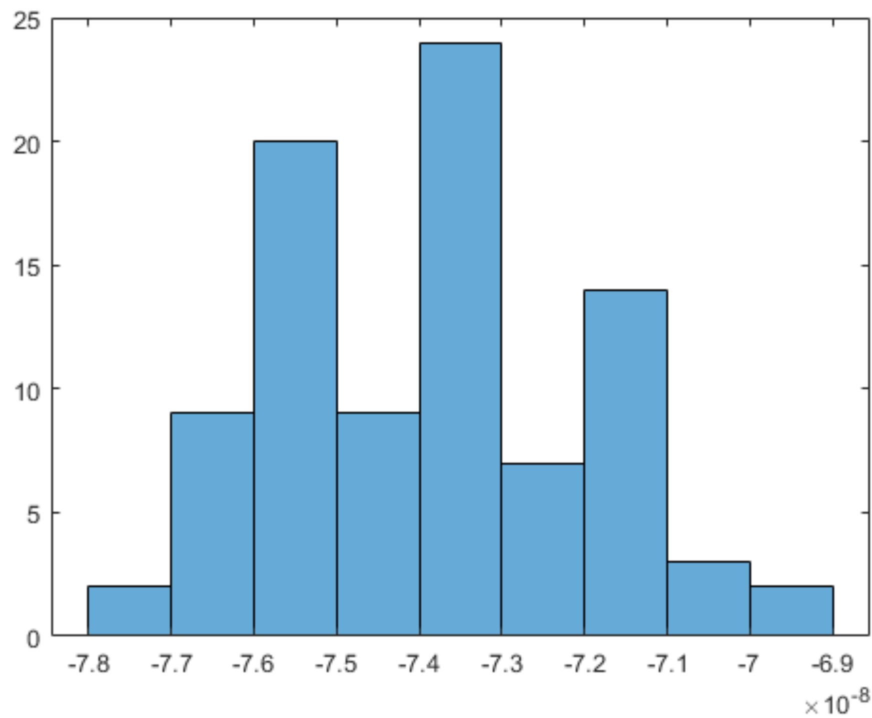


*Figure 18: The results of the characterization of the single pulses.*

# Chapter 4: Synchronization Architecture

## 4.1 Motivation and Objectives

After characterizing the signals from the EndRun Ninja, a solution was needed to incorporate those signals into a system that could synchronize a transmitter and receiver. This chapter discusses that solution. For the purposes of this research effort, "solution" refers to the combination of new code developed and modifications made to existing code and other structures created by Pentek in order to create a system capable of achieving the goals of this research effort, discussed in Chapter 1 and below. "System" refers to the combination of physical components and their respective code required to implement the solution in the field, such as the Ninja, its antenna, the 5950, all required cables, and the code running on the Ninja and 5950.

At a high level, this solution would function in two parts: one system would act as a transmitter, and another would act as a receiver. To reduce code replication and to add flexibility to the solution, the solution was designed to include both transmitter and receiver functionality on both systems, where the end user can decide which system acts as a transmitter and which acts as a receiver simply by making a modification to the initialization file (or selecting a different initialization file that includes the modification) and connecting or disconnecting cables from the

front panel. The solution would need to include a way for each system to be able to accept a signal that would start the synchronized pulse train on both systems at the same time and a signal to keep the clocks of both systems synchronized.

The first requirement for an acceptable solution is the ability to use the 5950 to generate pulses in a precise way with respect to timing and frequency. Specifically, the 5950 needs to be able to transmit, receive, and save signals in a pattern matching the specifications of the intended CPI, such as pulse width, pulse repetition interval (PRI), and number of pulses. The 5950 must accomplish this while adhering to the frequency and timing specifications, outlined in Chapter 2, that the Ninja is capable of. The frequency drift of the 10MHz signal, used to discipline the 5950's internal clock, is characterized in Section 2.4, so the solution's frequency drift should approximately match this characterization. Similarly, the jitter of the single pulse signal, used as a start time trigger for the CPI, is characterized in Section 2.6, and the jitter observed in relative CPI start time between systems should approximately match this characterization.

## 4.2 Equipment

### 4.2.1 Pentek Model 5950

Because transmit signals were not required for characterizing the Ninja reference signals, only the example program “acquire” was used. Because transmitting signals is a requirement for a pulsed radar system, the example program dacquire (DACquire is a version of “acquire” that allows access to the DAC) was used. Dacquire functions very similarly to “acquire”, and the user interfaces with it in the same way. Initialization files between the two programs are nearly identical, with the dacquire initialization files including DAC-specific and Digital Upconverter (DUC)-specific arguments that are not defined in “acquire”’s source code. The DUC was not used in this research effort, and the DAC-specific arguments will be discussed later in this section.

Looking at dacquire’s source code, the first difference that is immediately clear is the definition of four DAC modes: SIGGEN, CHIRP, ARB, and LOOPBACK. CHIRP and LOOPBACK were not used in this research effort, but SIGGEN and ARB were. Each of these DAC modes is accompanied by a relevant section of code, but just as much of “acquire”’s source code consists of memory calls to undocumented and hard-coded memory locations, so do much of these sections of code, making it difficult to obtain full operational understanding.

A feature mentioned in documentation for the Pentek 5950 is the ability to transmit arbitrary waveforms through the 5950's DAC from on-board memory. [9] Although this is technically true, some trial and error was required to harness this ability. The process to upload an arbitrary waveform to RAM, where it is clearly accessed in the block diagram (uppermost highlighted IP block in Figure 19), was not found in the documentation available.

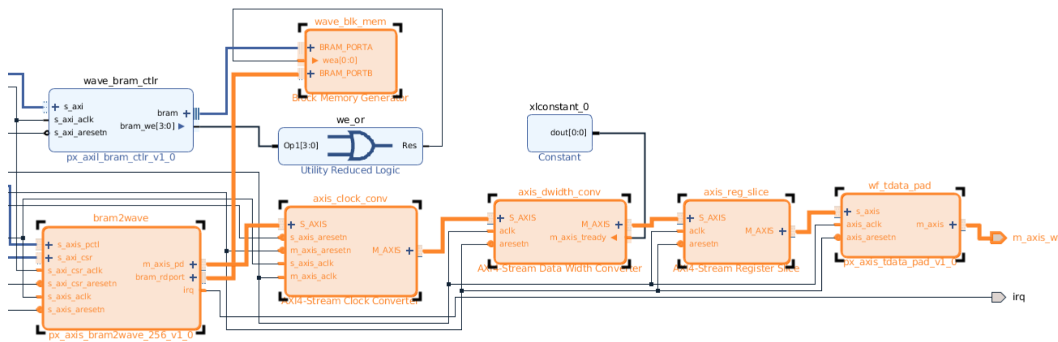


*Figure 19: The beginning of the path an arbitrary waveform takes within the 5950's original block diagram.*

Upon inspection of dacquire's source code, it was found that dacquire stores the arbitrary waveform in an array, which is not declared in the C code file, indicating that it might be declared in one of the header files included in the source code, and would likely be hard-coded and unable to be edited after compiling dacquire. The header file in question, table.h, was found to be relevant to dacquire's

arbitrary waveform functionality. The number of samples to be stored in RAM from this header file was determined in dacquire's source code by a variable, 'length', which was originally set to a value far lower than the 32769 (not 32768) samples originally present in table.h. Because samples are loaded into RAM in a for loop which repeats a total of 'length' number of times, any samples present in table.h past the number specified would be ignored and not loaded into RAM. Compiling dacquire with a table header file containing fewer than 'length' samples, however, would cause the remaining available memory locations to retain their initial unusable random value, causing considerable noise at the end of the transmission. No documentation was provided to define a maximum number of samples, but through inspection of the 5950's memory map in Vivado and the memory addresses used in dacquire's source code, it was found that 32768 16-bit samples can be stored in RAM. Although this theory was not tested, based on the memory map found in Vivado, potentially more than 32768 samples could be used if the number of channels available is decreased, as the spaces in memory for each channel's arbitrary waveform are consecutive.

As mentioned earlier, the BSP User's Manual defines initialization file arguments that are DAC-specific. Unfortunately, it was discovered that these initialization file arguments do not actually modify the DAC's operation. The following arguments are analogous to arguments relevant to the ADC (listed in

Section 2.3.2) that were defined in the BSP User's Manual, but were found to have no effect on dacquire's operation:

- dacfreq (analogue of adcfreq) – Should specify DAC clock frequency, but the DAC operates instead at the board clock frequency, regardless of what this value is set to.
- dactrigmode (analogue of adctrigmode) – Should specify the source of the trigger signal used to start DAC transmissions, but the signals available to the ADC in the 5950's original block diagram were found to not be available to the DAC. In testing, the DAC transmitted the uploaded arbitrary waveform in a continuous loop without respect for the trigger signal specified under this argument. (discussed more in Section 3.6.1)
- dacdatasrc – Seems to be similar to the four DAC modes specified near the top of dacquire's source code, but lists 'dma', 'ram', 'sine', and 'ramp' instead. The four DAC modes in dacquire's source code are not editable by this setting, as each mode is enabled or disabled by defining each mode as either 1 or 0 in the source code.

These discrepancies from what was published in the BSP User's Manual and how dacquire and the 5950's DAC actually operate added months to this research effort, and necessitated a second IP block to be developed for this research effort, discussed in Section 4.6.2.

Initial tests of the 5950's arbitrary waveform transmission abilities also uncovered an important detail in creating a waveform suitable for the 5950: the 3dB passband for the DAC, listed as 10MHz – 3700MHz in the 5950's datasheet. This detail was initially ignored when creating waveforms to test dacquire with, which led to distortions observed in the DAC's output. This issue was easily resolved by creating an LFM waveform that meets the frequency specifications. Figure 20 not only shows the 5950 RFSoC failing to correctly synthesize the waveform, but it also shows garbage values at the end of the synthesized waveform due to unused spaces in memory. Figure 21, however, shows a correctly synthesized waveform. The distortions in the waveform's envelope are thought to be caused at least in part by the duty cycle the DAC is running at. Under this assumption, transmitting this waveform at a PRI more realistic for a pulsed radar system could help improve the envelope's distortions.

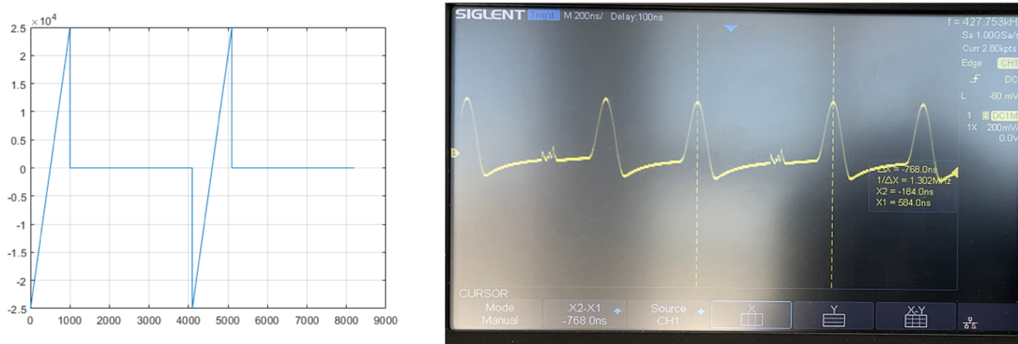


*Figure 20: An arbitrary waveform unable to be correctly synthesized by the 5950's DAC due to frequency issues.*

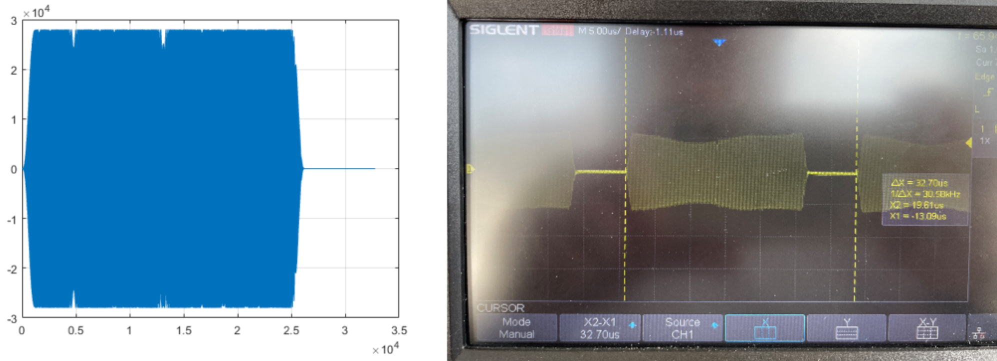


*Figure 21: An arbitrary waveform (LFM) that can be more correctly synthesized by the 5950's DAC..*

Previous attempts to find a solution to the problems addressed by this research effort included modifying dacquire to control precisely when the 5950 transmitted the arbitrary waveform to create a pulsed radar pattern, rather than modifying the block diagram. Between the ill-documented memory calls and the complexity of the block diagram alone, this effort was cut short to focus on modifying the block diagram (defined in Section 4.3.2). The only modifications to dacquire's original source code that remain in the solution are:

- Redefining SIGGEN as 0 and ARB as 1 to enable dacquire's arbitrary waveform mode and disable the internal signal generator.
- Changing the value of 'length' to allow dacquire to use all memory originally allocated to it.

- Changing the array in table.h to include the LFM waveform generated for this research effort.

### 4.2.2 Ninja

The Ninja is used as an external trigger signal and frequency reference, but the primary timing and synchronization features are implemented on the Pentek 5950.

## 4.3 Field Programmable Gate Array Code

### 4.3.1 Overview

Hardware Description Languages (HDLs) are distinctly different than other, more traditional programming languages such as C, Java, or MATLAB. HDLs model physical digital circuits containing components such as flip-flops, look up tables (LUTs), and more, rather than provide sequential instructions for a processor to interpret. FPGAs use HDL code to model desired functionality, and this code is then synthesized and mapped into the various elements that make up an FPGA [10]. Because HDL code models physical circuits, multiple tasks can execute simultaneously in parallel, helping to tightly control timing. The core functionality

of the 5950 RFSoC is contained within the onboard Xilinx Zynq Ultrascale+ FPGA, and the solutions described in this chapter rely on HDL code.

An important concept in HDL programming is the wrapper. As in other aspects of computer science and programming, wrappers are pieces of code that encapsulate other pieces of code for the purpose of abstraction, adding functionality, or bridging the gap between incompatible communication protocols. A direct analogy would be an integrated circuit chip: The chip on its own is not very useful, but a properly designed circuit board that connects to all inputs and outputs of the chip and ensures all requirements for those inputs and outputs are met allows the chip to be used to its fullest potential. In this scenario, the circuit board would be the wrapper for the integrated circuit, the HDL code. It is important to note that just as a circuit board is as much a piece of hardware as the integrated circuit, the wrapper is also written in HDL code.

### 4.3.2 Block Diagrams and Intellectual Property Blocks

In the context of FPGA Programming, Intellectual Property (IP) blocks are pre-built blocks that represent different kinds of functionality and are used in a block diagram, the visual representation of how an FPGA is being programmed. An example of a block diagram is shown in Figure 22. There are two types of IP, hard and soft. Hard IP is generally published by the manufacturer of the FPGA chip

and is optimized for the specific model of FPGA being used, often predetermining the physical gate connections within the FPGA chip itself [11]. Hard IP can be placed in a block diagram and can be connected to other IP blocks, but oftentimes hard IP cannot be modified by the user. Soft IP, on the other hand, is more broadly compatible, is written in HDL, and can be more easily customized [11]. Without very specific and in-depth gate-level knowledge of the specific FPGA chip present on the 5950 (Zynq Ultrascale+), soft IP is the most practical kind of IP to develop, despite its slightly less efficient allocation of an FPGA's resources due to lack of gate-level optimization [11].

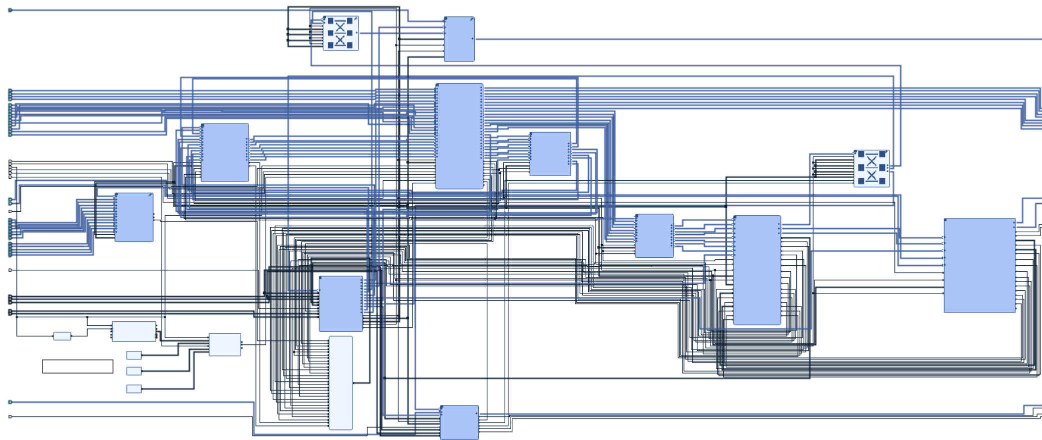

*Figure 22: An overview of the complete modified block diagram used in the 5950.*

### 4.3.3 AXI Communication Protocol and AXI4-Stream

The AXI Communication Protocol was developed by ARM in 2003 for the purpose of standardizing fast and efficient communication between separate components. [12] Xilinx and Pentek rely heavily on the AXI Communication Protocol for clocked parallel communication between IP blocks, and this protocol is extremely common in the block diagram provided by Pentek for use with the 5950.

The AXI4-Stream protocol relies on simple handshaking between upstream and downstream components to allow for efficient parallel data transfer. The essential components of an AXI4-Stream link are the clock, the data bus, tvalid signal, and tready signal. Because AXI is a clocked protocol, both upstream and downstream blocks use the same clock. The data bus can theoretically be any number of bits. The tready signal (downstream to upstream) and tvalid signal (upstream to downstream) signals are how both blocks communicate to each other that they are ready to send and receive data. If either of these signals is deasserted, data flow stops until both upstream and downstream are ready to continue. Another signal, tlast, is commonly included, but is not necessary, and is not used in this work. The tlast signal is sent from upstream to downstream and indicates to the downstream block that the data currently on the data bus is the last it has to offer. Because the AXI4-Stream applications in this paper are constantly streaming, tlast

is not applicable. Figure 23 shows how two IP blocks interact over AXI4-Stream. [13]

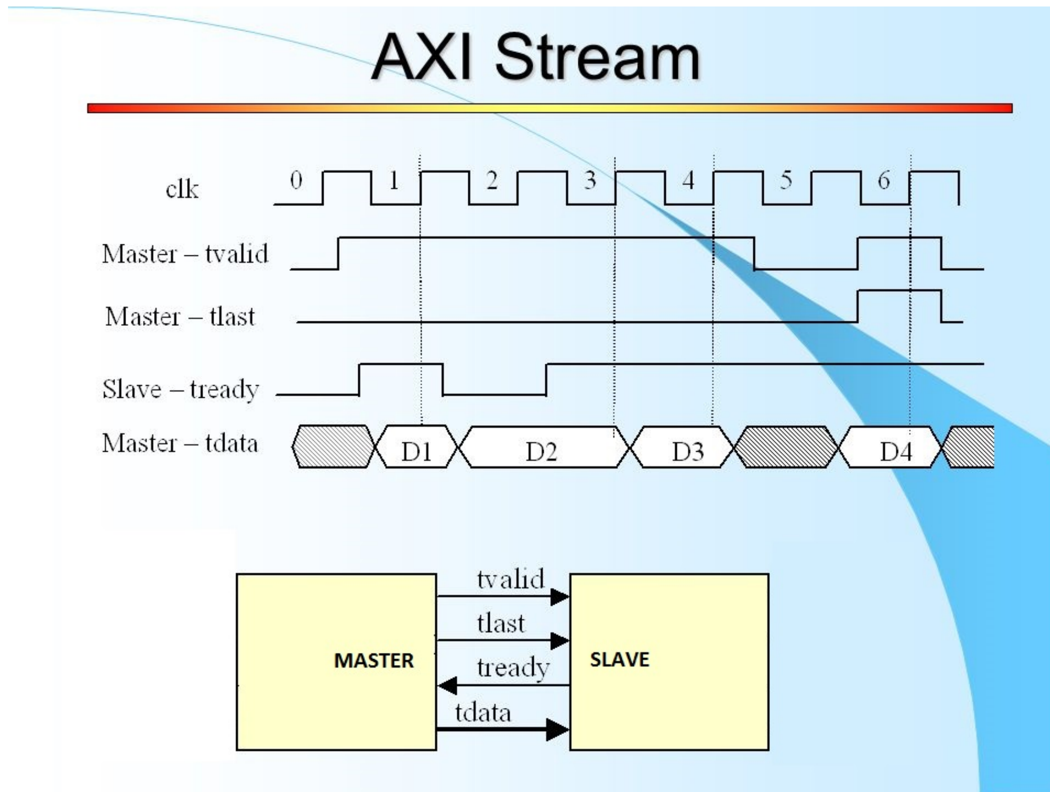


*Figure 23: A timing diagram showing how two IP blocks interact using the AXI4-Stream protocol.*

## 4.4 Coherent Processing Interval Control Block

As discussed in Chapter 1, one of the primary timing requirements for a bistatic pulsed radar system is synchronizing the absolute start time of a CPI. An

IP-based solution to this problem would require a start trigger, internal timing, a clock, and the ability to generate triggers that work with the DAC and ADC systems in the 5950. A soft IP block, “radar_control”, was developed to generate trigger signals for both the DAC and the ADC. The code for the block is written in Verilog, which is an HDL. For initial use, the block's PRI and duty cycle parameters were hard coded, and the block does not incorporate any AXI communication, but future development could create an AXI wrapper that could provide PRI and duty cycle parameters from information provided by the user during runtime. Figure 24 shows this IP block and its connections.

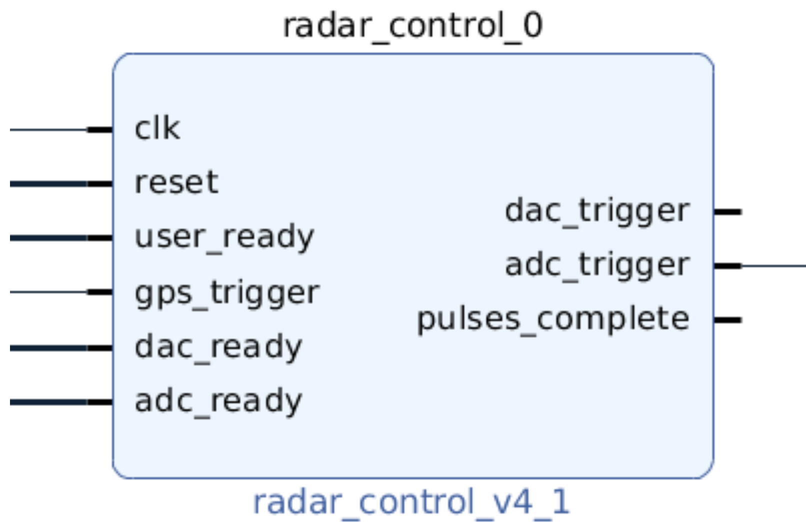


*Figure 24: The "radar_control" block and each of its inputs and outputs.*

The choice to avoid using the AXI protocol to communicate with the rest of the block diagram about the status of the DAC and ADC triggers is motivated by timing constraints. Due to external and internal signals being inherently unaligned in frequency or phase, any amount of delay could cause timing error if it causes the FPGA to wait a clock cycle to enable or disable the DAC or ADC. In order to minimize any delay in the logic, an asynchronous approach was chosen. Because the ADC is accustomed to using an asynchronous external trigger signal, it will not be affected by this choice. The DAC system, as described in Section 3.6.1, uses

AXI4 Stream to retrieve and stream the arbitrary waveform hardcoded into BRAM, and the custom IP developed to finely control the DAC (discussed in Section 3.6.2) uses AXI4 Stream to operate as well. Because AXI4 Stream is a clocked interface, the trigger signal is not registered by the IP block until the following clock rising edge. The delays introduced by this asynchronous to synchronous approach are discussed in section 3.7.

The Ninja offers an absolute GPS time-based single pulse option, characterized in Section 2.6, which offers a convenient way to provide a start trigger and interface with this solution. Assuming that each set of Ninja and 5950 units are already running, with the 5950 units running dacquire, operators of the separate transmitter and receiver would only have to agree on a start time for a CPI and write one command in the Ninja terminal (triggerppo – discussed in Section 2.2.2), setting this predetermined start time, to have both transmitter and receiver receive a CPI start trigger at the same time. As discussed in Section 2.6, the Ninja self-reports delay on its absolute GPS time-based single pulse, so even if the delays for each Ninja are different, the information is readily available in the Ninja terminal for alignment in post-processing.

Other inputs to the "radar_control" block include external reset and ready inputs, inspired by the AXI protocol. The three ready inputs could allow the user, the DAC, and the ADC to stop or prevent operation of the "radar_control" block if

necessary. Because the 5950 does not offer a simple way of interfacing externally with such a flag, and neither the DAC nor ADC offer such signals, these inputs are tied high in the current form of the design using a Xilinx-designed IP block [14] that provides a constant value. Even though these ready functions are currently not in use, their existence allows for a more robust implementation in the future without having to redesign and retest the functionality of the Verilog code powering the IP block. The external reset input is included for similar reasons, but is operational in the modified block diagram, because the IP block will not properly start without an external reset signal. This external reset is a synchronous reset that, when a rising edge occurs, will initialize the IP block to its default state, including all registers. This initialization is required at startup, or none of the registers will have been set to their correct values. This reset is separate from the self-resetting capability of the IP block after a CPI is completed, as indicated by the pulses_complete flag, where the IP block automatically readies itself for another external trigger when this flag goes high. In order to generate the initial "external" reset signal, another IP block was created, "radar_control"_reset, whose sole purpose is to create a single pulse to feed into the "radar_control" block's reset input. After initial testing of this separate IP block, the Processor System Reset block, developed by Xilinx, was found to be necessary. This block ensures that an external reset signal is properly synchronized to a clock and is placed between the "radar_control_reset" and "radar_control" blocks. [15] In future development, this external reset signal could

be made more robust and user-accessible, allowing for an operator to manually reset this IP block in case of malfunction. The entire reset functionality of the “radar_control” system, including “radar_control_reset”, were developed after consultation with an engineer at the University of Oklahoma’s Advanced Radar Research Center after issues with compiling the overall Vivado project were narrowed down to the “radar_control” block (see Section 3.8). Figure 25 shows the “radar_control” block alongside its supplementary IP blocks, including “radar_control_reset”.

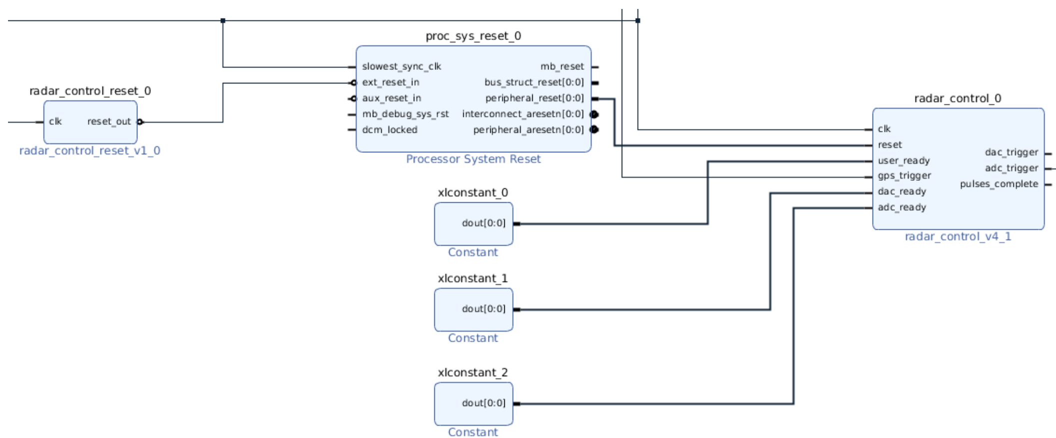


*Figure 25: The “radar_control” block, along with the constants and reset logic, including “radar_control_reset”.*

The most important outputs of the “radar_control” block are the two trigger signals for the DAC and the ADC. Because the ADC and DAC require trigger signals rather than enable signals (where the DAC or ADC is not only enabled when

the signal goes high, but also disabled when the signal goes low), the duty cycles for each signal are set to 50%. The final output of the “radar_control” block is the pulses_complete flag, which goes high after the final pulse in a CPI is complete. This flag being raised also signifies that the block is ready to start a new CPI when a new trigger rising edge is detected. Any trigger rising edge detected during a CPI will be ignored. Figure 26 shows the basic functionality of “radar_control” through a timing diagram simulated in Vivado. Note that this timing diagram does not factor in the reset signal.

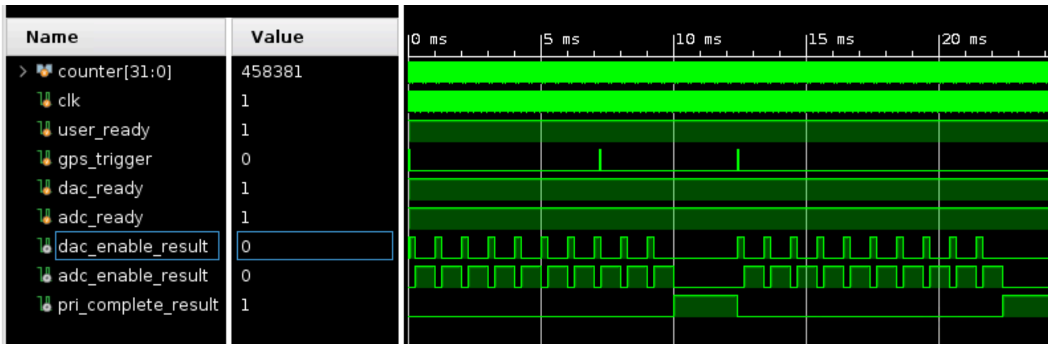


*Figure 26: Vivado timing diagram showing operation of “radar_control” block given certain inputs.*

The “radar_control” block primarily operates using a number of constant values and counters that trigger different actions when particular values are reached. The two counters, “fast_counter” and “slow_counter”, keep track of fast and slow time, respectively, during a CPI. The fast counter counts clock cycles of the “radar_control” block’s 200MHz clock and counts up to multiple values. The

fast counter begins counting when the IP block first recognizes that it is within the bounds of a pulse repetition interval (PRI). This recognition can occur either when the IP block registers a GPS trigger rising edge or when a new pulse is started directly following a previous one. The fast counter, along with many of the constants defined in the reset condition, allows the IP block to check for several conditions each clock cycle. The first level of conditions checked are whether the DAC trigger or ADC trigger should be asserted, whether the DAC trigger or ADC trigger are already asserted, and whether the PRI is complete. The next level of conditions checked for the DAC or ADC take action if the fast counter's value indicates it's time to deassert their respective triggers. Within the PRI complete condition, the slow counter is referenced to know whether to start another pulse or assert the pulses_complete flag. The slow counter counts the number of pulses completed, and counts up to the pulse limit, at which the IP block asserts the "pulses_complete" flag and prepares for the next GPS trigger. Between both a reset signal and a GPS trigger rising edge and a "pulses_complete" flag rising edge and a GPS trigger rising edge, the IP block is in an idle state, with no counters in operation, simply waiting for a GPS trigger.

Because only one 5950 was available for development and testing, this process was originally designed with monostatic pulsed radar design in mind. Most monostatic pulsed radars do not enable their receivers until their transmitters are disabled to prevent direct path interference, so "radar_control" was originally

designed with a timing condition that asserts the ADC trigger tied to the DAC's trigger being deasserted. Because a bistatic radar's transmitter and receiver are separated by definition [1], this limitation serves no purpose. A specific constant value, "dac_adc_offset", was introduced in a later version of the code for "radar_control" to allow for an arbitrary (nonzero and non-negative) offset between the DAC trigger rising edge and the ADC trigger rising edge, measured in clock cycles. In testing, this value was set to 1 clock cycle. At the 200MHz clock used by "radar_control", this equates to a 5ns delay between these rising edges, which would only cause the receiver to miss the start of the transmission if the combined distance the transmission traveled from transmitter to target and from target to receiver was approximately 1.5m.

## 4.5 Modifying Analog-to-Digital Converter Functionality

The ADC's modes of operation rely on how the initialization file or the example programs "acquire" or dacquire are configured. The trigger option used and tested throughout the signal characterization process accepts an external trigger signal and uses it to start a data capture with the indicated channel for the specified number of bytes at the specified bytes per sample and ADC clock rate. Rather than directly modifying the ADC, the "radar_control" block's ADC trigger supplants the previous external signal, which is instead used as the external signal that provides

“radar_control” with the Ninja’s GPS-based CPI start trigger. Figure 27 shows the original path the trigger signal takes within the 5950’s block diagram, and Figure 28 shows the modified path the trigger signal now takes, where the front panel input is repurposed as the GPS trigger and the ADC trigger output of “radar_control” connects to where the original trigger signal used to connect.

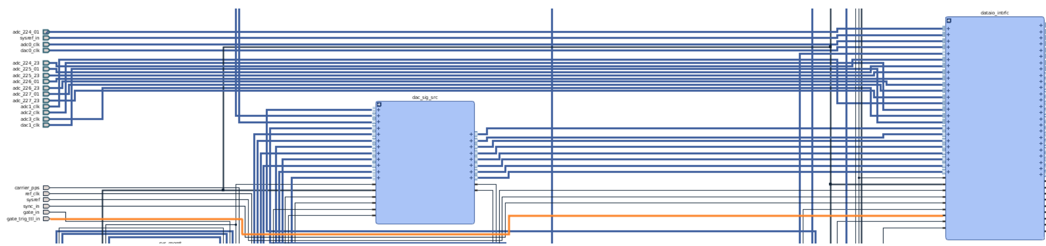

*Figure 27: The path the signal associated with the TRIG front-panel input on the 5950 takes originally.*

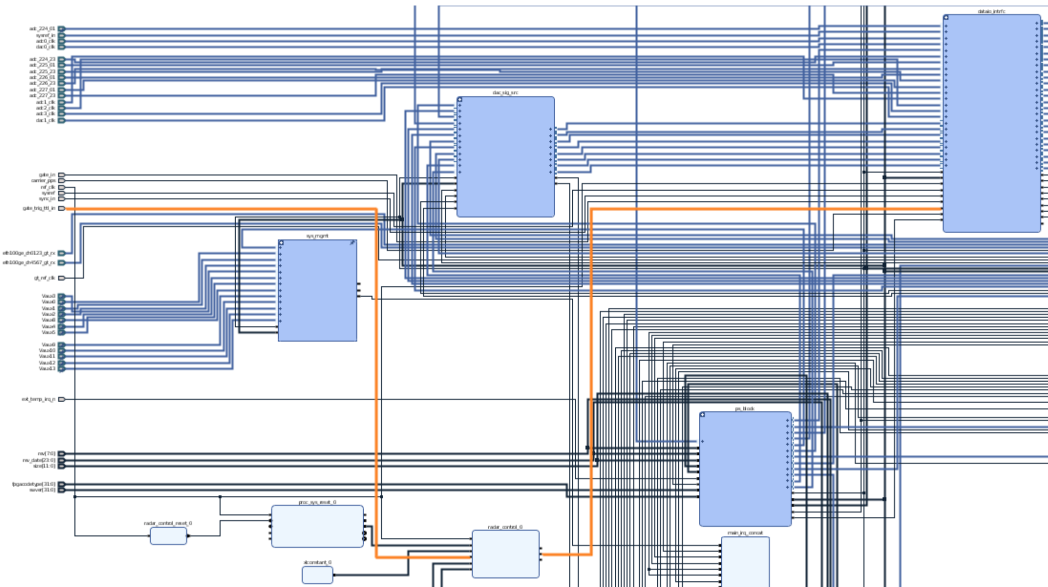

*Figure 28: The same TRIG signal is rerouted to the "radar_control" block, and the ADC trigger output from the "radar_control" block is routed to where the TRIG signal used to go.*

This signal modification, combined with initialization file settings (specifically the transfer size and write size) that allow for sufficiently long captures per PRI, satisfy the solution's requirement that the ADC be able to successfully capture the CPI.

## 4.6 Modifying Digital-to-Analog Converter Functionality

### 4.6.1 Original DAC Configuration

In testing done on the 5950 after modifications were made to dacquire.c, including a waveform matching the physical frequency requirements of the onboard synthesizer (discussed in Section 3.2.1), it was found that the waveform would be immediately transmitted through the DAC continuously. Specifically, the DAC transmitted the correct signal at the correct sampling frequency, but the 5950 restarted the waveform, transmitting the first sample of the waveform immediately following the last sample and continued transmitting the waveform again, repeating this cycle a random number of times. Unlike the ADC operation, the number of waveform transmissions observed when the loop value (defined in Section 2.3.3) was set to a nonzero value did not conform to a pattern recognizable to be deriving from the initialization file. The only indication that the loop value affected the operation of the DAC in any way was the observation that the DAC would only not stop transmitting continuously when loop was set to 0, similar to how the ADC's operation is defined in Section 2.3.2.

A closer look into the block diagram in the Vivado project provided by Pentek, however, shed light on why the DAC did not respect a trigger signal, regardless of how the initialization file configured dacquire. No trigger signal nets

were found in the areas of the block diagram associated with generating and transmitting DAC signals, and within a multipurpose subsection of the block diagram that handles both ADC and DAC operation, multiple IP blocks that appeared to process trigger signals from the front panel were present for ADC operation but not for DAC operation. Figure 29 shows exactly this: The trigger net (first highlighted net on the left) only leads to an ADC-specific IP block, of which there is no DAC counterpart. The two IP cores highlighted on the right are also ADC-specific with no DAC counterpart.

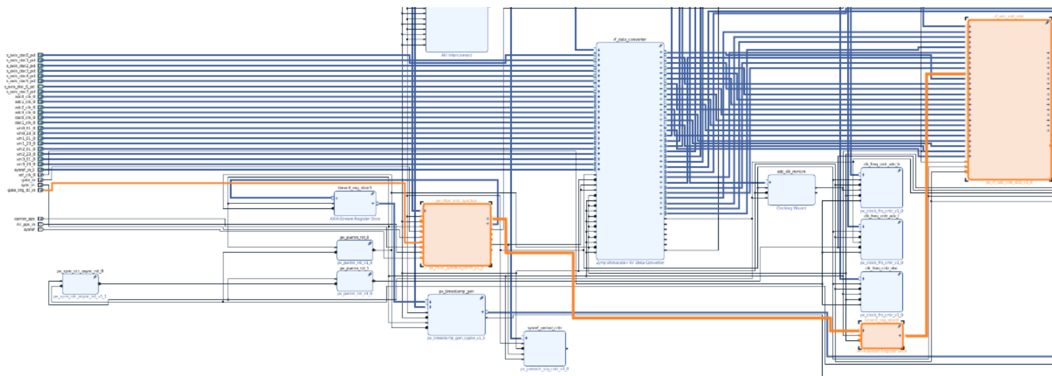

*Figure 29: A section of the block diagram whose purpose is controlling the DAC and ADC, showing no trigger signal for the DAC.*

### 4.6.2 Custom AXI4-Stream IP

Because the DAC was not observed respecting any frequency of trigger signal supplied to the TRIG port on the front panel of the 5950 unit, a custom

solution was required to finely control when and for how long the DAC would transmit the arbitrary waveform compiled into the dacquire program. This proposed solution would need the ability to accept a trigger signal (provided by the "radar_control" IP block), intercept the AXI4-Stream link at its source, and only allow waveform samples to stream through at the appropriate times. At all other times, a zero value would need to be transmitted, effectively shutting off DAC transmission.

The solution designed takes advantage of an integral feature in the AXI standard, defined in Section 4.3.3: the ready signal. In a pair of upstream and downstream IP blocks, each block can halt the flow of data using the valid and ready signals. If the upstream block deasserts its valid signal, the flow of data stops. Similarly, if the downstream block is not ready to accept data from upstream, it can deassert its ready signal, and the upstream IP block will pause until ready is reasserted. In this case, the ready signal is the perfect method to control when the waveform is transmitted through the DAC, as custom IP that accepts a trigger can be placed within the data stream, as far upstream as possible, and manage the data flow.

The design of the solution implemented into the block diagram follows this approach, but in two stages. On the block diagram, the AXI4-Stream link is disconnected between the "bram2wave" and the "axis_clock_conv" blocks, shown

in Figure 30 and rerouted to a First-In-First-Out (FIFO). The FIFO's purpose is similar to the custom AXI4-Stream block built to switch its ready signal based on a trigger and preset timing parameters. When the modified version of dacquire (discussed in Section 4.2.1) starts, the DAC immediately starts transmitting the waveform baked into the program at compile time. The FIFO is configured to match the number of samples of the waveform (32768), so when dacquire starts on the 5950 running the modified block diagram, the FIFO fills up with the 32768 samples of the waveform. This is the first, or holding, stage of the DAC modification.

The second, or streaming, stage is more complicated. It involves the custom IP block created for DAC operation modification, axis_dac_switch. Because the DAC transmits the waveform immediately, the holding stage is not connected to a trigger signal, but because the goal of this modification is to gain control over when the waveform is transmitted, this IP block does require a trigger signal at its input. Because the operation of blocks downstream when denied data over AXI4-Stream is undocumented, to avoid any issues the custom IP block streams a zero value downstream until it receives a rising edge on its trigger port. At that point, the IP block "empties out" the FIFO and streams its contents downstream. As the FIFO empties out into axis_dac_switch, its own ready signal is asserted, allowing it to replenish itself from the BRAM IP block upstream. The axis_dac_switch block keeps track of its own timing once it receives a rising edge, so once it has streamed all 32768 samples, the block "switches" back to streaming a zero value

downstream, deasserts its ready signal, which causes the FIFO upstream to also deassert its ready signal. This design will cause the FIFO to be full at all times after its initial intake at the start of runtime, but the timing built into axis_dac_switch will cause the FIFO to start and stop streaming data when the first and last samples of the waveform take up the first and last positions, respectively, in the FIFO. This ensures that the axis_dac_switch IP block will always have a complete waveform at its disposal to stream whenever it receives a trigger rising edge.

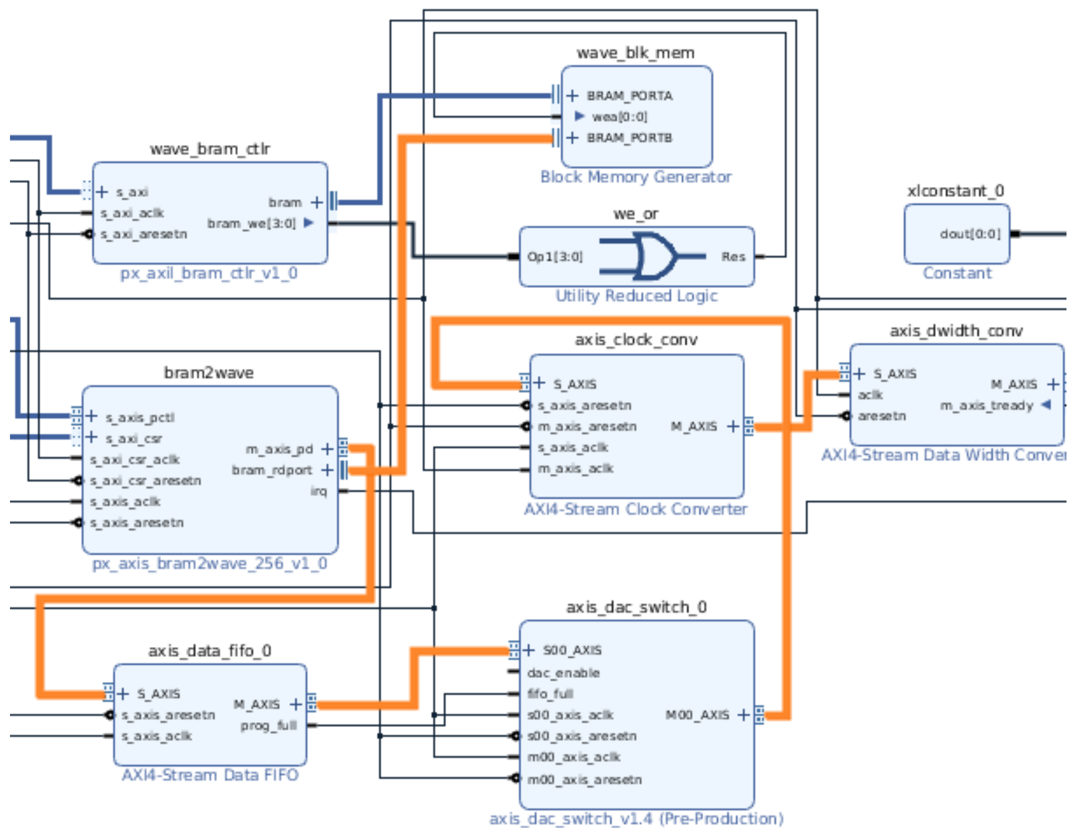


*Figure 30: The new path an arbitrary waveform takes through the block diagram.*

Vivado treats AXI connections differently than standard wire connections in a block diagram. As shown in Figure 30, AXI connections are thicker than normal and blue, distinguishing them from wires. To package an IP block to use these special AXI connections, the "Create a new AXI4 Peripheral" option in the Create and Package New IP tool must be invoked, allowing the Verilog code to be instantiated within an AXI4-Stream wrapper. This tool creates a new Vivado

project with several autogenerated Verilog files: one representing the functionality of the overall IP core, and one per upstream or downstream connection present in the IP core. In the case of axis_dac_switch, only one upstream and one downstream connection are present, so only three autogenerated files were present. The file representing the overall functionality instantiates (or "calls") each of the files, making internal connections between values of the top-level Verilog file and instantiated values in each of the upstream and downstream files. The functionality of the streaming stage described in the previous paragraph is encapsulated in a single Verilog file (axis_dac_switch.v), which includes all required inputs and outputs required to work with AXI4-Stream connections and achieve the desired functionality. Not much documentation is present in Vivado regarding implementing a previously designed Verilog file (axis_dac_switch.v) into an official AXI IP block design, but the simplest solution was to disregard the two autogenerated files representing each of the upstream and downstream connections, and instantiating axis_dac_switch.v into the top-level Verilog file, taking the places of both upstream and downstream files.

This multi-stage modification of the 5950's DAC allows for a trigger, generated by the "radar_control" block, to precisely control when the hard-coded waveform is transmitted, as no suitable trigger option was operational in the block diagram provided by Pentek. Combined with the "radar_control" IP block, the axis_dac_switch block allows the 5950 to adequately transmit pulses for a pulsed

radar system, whether bistatic or monostatic. The timing diagram showing the DAC and ADC triggers, as well as the output of the axis_dac_switch block, are shown below in Figure 31. Please note that all timing diagrams generated in this research effort come from simulations of the Verilog HDL code in Vivado, rather than from real-world tests. Figure 32 zooms into the same timing diagram, but gives confirmation that the axis_dac_switch block works precisely as intended, transmitting a pulse of exactly 32768 samples at a sampling rate of 1GS/s.

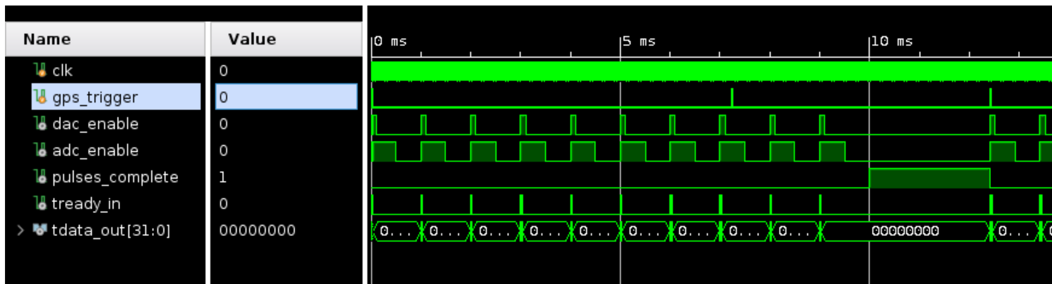


*Figure 31: A timing diagram showing the outputs of the "radar_control" and axis_dac_switch blocks in a simulated environment in Vivado.*

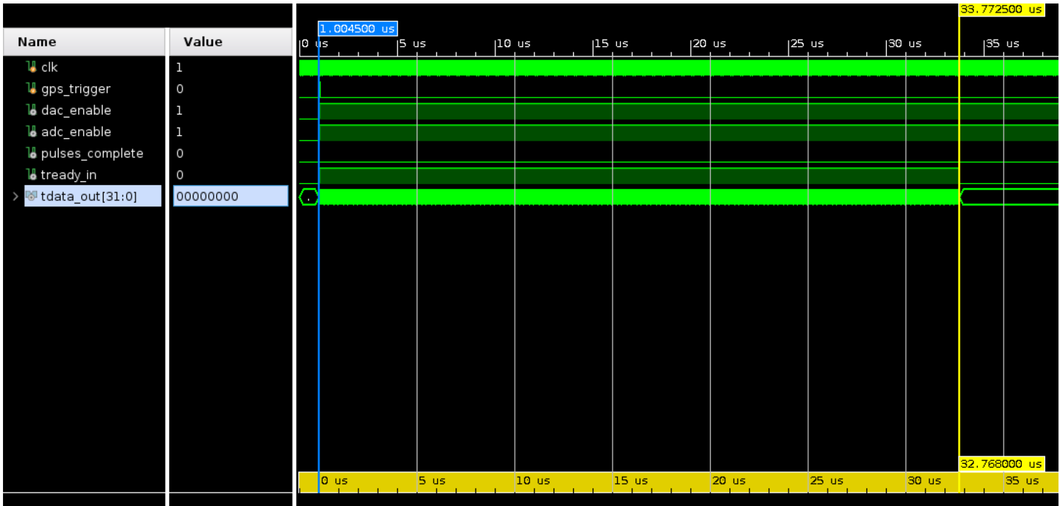


*Figure 32: A closer look at the above timing diagram, confirming that the signal streamed by axis_dac_switch is indeed 32768 samples long at the 1GS/s DAC sampling rate used in initialization files throughout this research effort.*

## 4.7 Timing Tolerances of Solution

The specific timing tolerances of the two IP blocks in tandem are relevant as well. Figures 33, 34, and 35 show the same timing diagram as above, but zoomed in even further, with timing markers defining relevant types of delays inevitable in a clocked, synchronized design such as the one developed for this research effort.

One of the two problems this research effort sought to solve was synchronizing jitter in separated transmitter and receiver systems. The solution developed to process external triggers, “radar_control”, is a clocked IP block – it uses its clock to keep track of the times at which it must assert or deassert different

signals. The clock that disciplines "radar_control" is a 200MHz clock, with a period of 5.000ns as shown below in Figure 33:

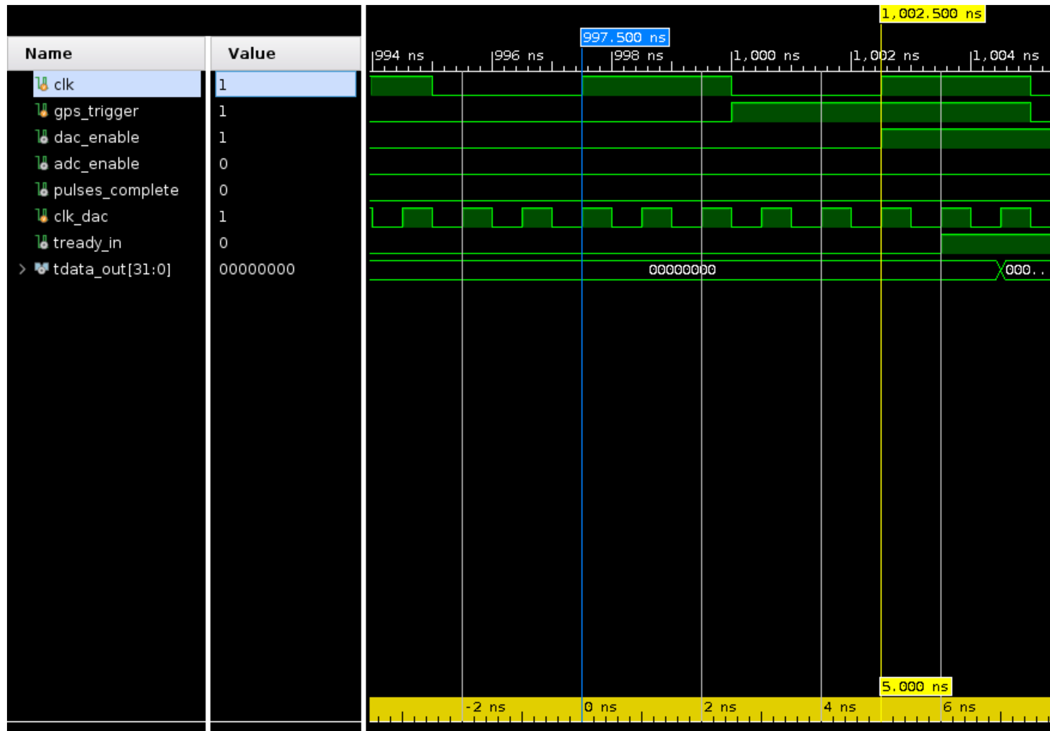


*Figure 33: Clock cycle period*

This 5ns window represents the first relevant delay in this solution, as any trigger occurring between clock rising edges will not be registered by "radar_control" until the next rising edge. As shown in Figure 17, the signal used as a trigger is maintained at its peak for at least 50ns, often longer, so there is no risk of the trigger being lost between clock rising edges. This delay will never be longer than 5ns, as the following rising edge will always pick up the trigger.

Another relevant delay present in the solution is the ideal delay between "radar_control" asserting the DAC trigger and axis_dac_switch streaming the first sample of the arbitrary waveform, as shown in Figure 34. The 2ns, or two 1GHz clock cycle, delay is ideal in that the propagation delay of the DAC trigger from "radar_control" to axis_dac_switch might cause an extra one 1GHz clock cycle delay, bringing the total delay to 3ns. This delay is not as useful on its own as it is in conjunction with the trigger delay discussed above, so the total delay added to the DAC system solely due to this solution is at most 8ns.

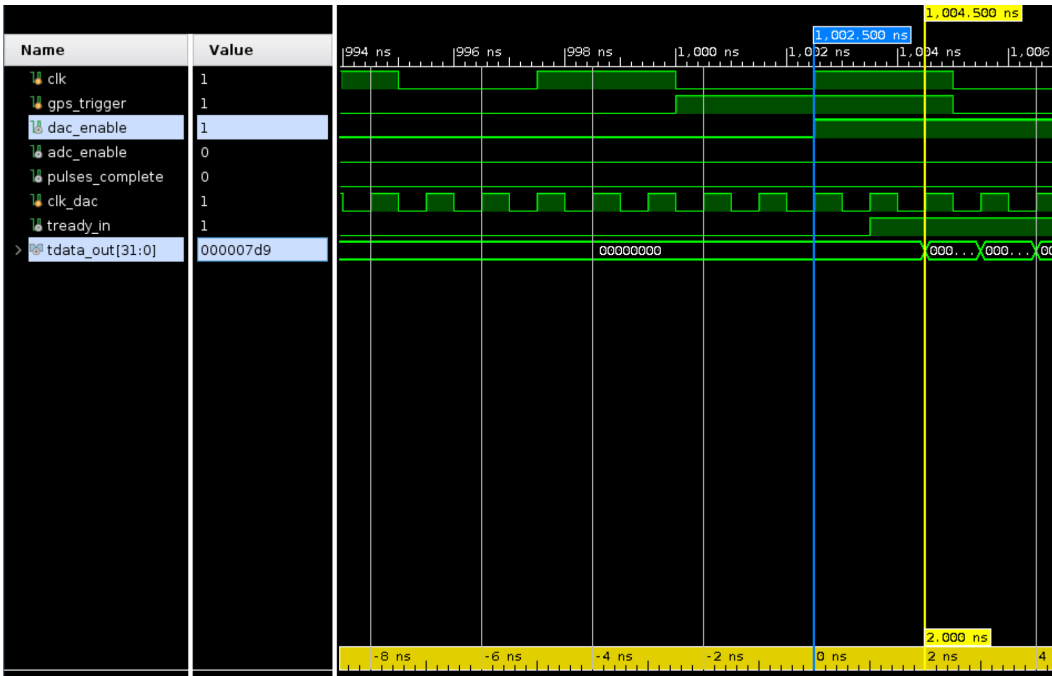


*Figure 34: Delay between dac_trigger rising edge and axis_dac_switch streaming first sample of arbitrary waveform.*

The final relevant delay is that between the DAC and ADC triggers. This delay was first discussed at the end of Section 4.4 and is shown in Figure 35. This delay is customizable, and in the code as it is currently, this delay is set to one 200MHz clock cycle, or 5ns. This is the minimum possible value of this delay, and as the timing for both the DAC and ADC triggers resides in the same IP block, there is no risk of an extra clock cycle delay being added.

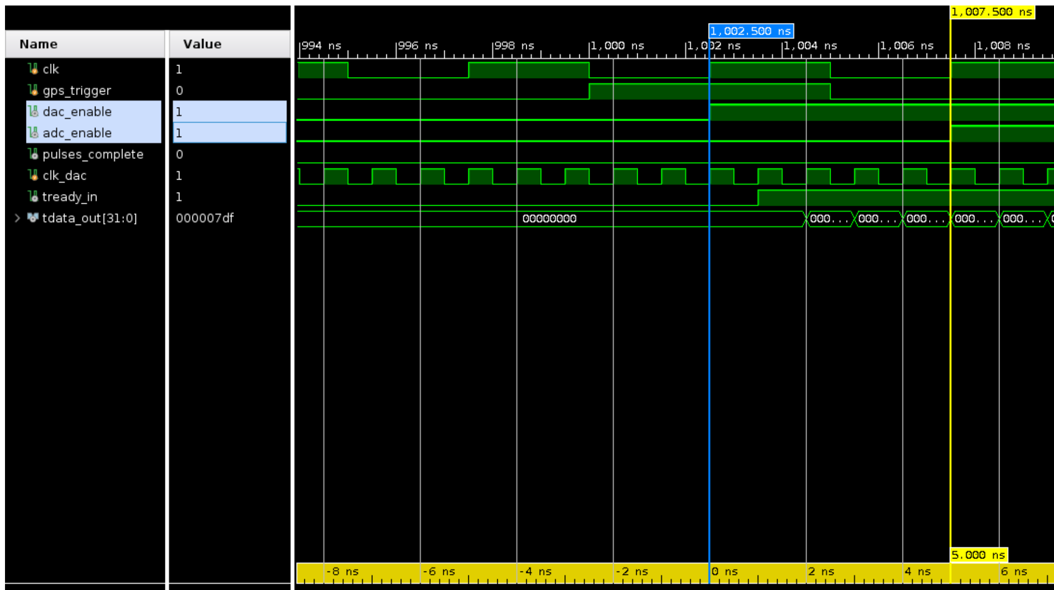


*Figure 35: Ideal delay between dac_trigger rising edge and adc_trigger rising edge.*

## 4.8 Implementation Issues

After designing and testing the Verilog code powering the IP blocks developed for this solution, the next step was to compile the Vivado project, replace the original FPGA code with the newly compiled code, run dacquire, and run testing on the overall system to gauge how effective the solution is at achieving the goals defined in this research effort. The steps of compiling a Vivado project are:

1. Validate Design – This step analyzes the block diagram for potential issues.
2. Synthesis – This step creates a netlist from the block diagram [10].
3. Implementation – This step maps the netlist created in the Synthesis stage to the physical architecture and resources of the FPGA [10].
4. Generate Bitstream – This step creates a file that the FPGA uses to program itself to match the map created in the Implementation stage [16].

Initial attempts to compile the project resulted in two previously undiscovered issues coming to light, both of which seem to be issues with Vivado 2018.3, the version of Vivado used throughout this research effort, as well as the only version compatible with the hard IP provided by Pentek in the FDK.

The first issue encountered dealt with a parameter of one of the busses on the axis_dac_switch block, specifically the FREQ_HZ parameter, which defines the frequency expected on that bus. This issue was resolved by finding the

FREQ_HZ parameter values of the IP blocks surrounding axis_dac_switch and repackaging the IP block with the correct value. Vivado's lack of ability to adjust this parameter when an IP block is placed in a block diagram has been documented online as a potential bug in multiple Vivado versions, including 2018.3 [17].

The second issue occurred during implementation and is confirmed to be a bug that has been resolved in later versions of Vivado [18]. Implementation would freeze and crash due to an inability to create a design checkpoint. The only resolution for users using Vivado 2018.3 seems to be restarting Vivado and trying implementation again, which worked in testing.

As these problems were identified and resolved, one error, shown in Figure 36, continued to occur during implementation:

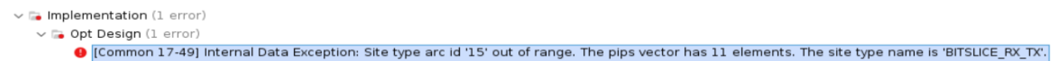


*Figure 36: The error constant throughout all attempts to compile.*

This error was not recognized by any graduate students or engineers at the University of Oklahoma's Advanced Radar Research Center consulted throughout the research effort. Discussion of this error is only found online in three places, all on Xilinx's online forum, a website for FPGA developers to ask and answer questions regarding developing on Xilinx FPGAs [19] [20] [21]. None of the three

discussions provided a viable solution or explanation. Without any outside information to expedite the process of resolving this error, multiple copies of the original Vivado project were made to check each modification, step by step.

The first copy of the original Vivado project was left unmodified in order to verify that the four steps of compiling worked on the original project in Vivado 2018.3. As expected, this project was able to be compiled completely, albeit after three tries, due to the design checkpoint bug. Similarly, the copy of the original project modified to only include the modifications made to the DAC, namely the FIFO and axis_dac_switch, was also able to be compiled completely. Logically, the only possibility was that the “radar_control” system was at fault for the Internal Data Exception error during implementation. Upon attempting to compile the version of the project only containing the “radar_control” system, without the reset logic, which had yet to be developed, this conclusion was confirmed, as the same error consistently caused implementation to fail.

The next step in debugging was to return to “radar_control”’s Verilog source code and verify that nothing in the source code was preventing implementation. Such issues could easily be overlooked in the coding and simulation stages of developing “radar_control”, as Vivado’s simulator does not account for the physical architecture of the 5950’s FPGA, and code that successfully runs in simulation is by no means guaranteed to be able to be

successfully implemented. One issue found was an inconsistent usage of blocking and non-blocking assignments. Of these two ways of assigning values to variables in Verilog, non-blocking is always the appropriate method in the "always" block, the structure within the code that is run at each rising edge of the clock [22]. This issue was easily fixed. The other issue found was the lack of reset logic, leading to the creation of the reset logic discussed near the end of Section 3.4. After these revisions, implementation still failed with the same error message.

At this point, no modified version of the code containing all functionality developed in this solution has been able to be implemented, so no real-world testing of the solution has been able to occur to verify that the pulsed radar pattern produced by "radar_control" and axis_dac_switch conforms to the timing and frequency characteristics found in Chapter 2.

# Chapter 5: Conclusion

## 5.1 Conclusions

Synchronizing two nodes of a bistatic radar system using GPS-disciplined signal generators and an FPGA-based RFSoC has been explored in detail. The three relevant types of signals generated by the EndRun Ninja GPS units, the 10MHz sinusoidal signal, the 10MPPS square wave signal, and the absolute GPS time single pulse have been characterized in terms of frequency drift, timing stability, and jitter, respectively. The Pentek Model 5950 RFSoC unit's example programs, "acquire" and "dacquire", have been studied and tested to verify functionality and better understand their operation. FPGA block diagram changes and custom IP have been considered to account for lapses in functionality when attempting to generate LFM signals for a pulsed bistatic radar system. The work explored in this research effort outlines a very modular and customizable synchronized bistatic radar implementation. This modularity is achieved through the customization of the Pentek 5950's original block diagram combined with custom-designed soft IP that can be repackaged with different pulsed radar parameters and Pentek's example programs, which can be recompiled with different arbitrary waveforms. Synchronization is achieved by using the 10MHz and single pulse signals from the EndRun units to discipline the modified 5950. However, many areas of

improvement are plainly visible. The IP block designed to generate the internal trigger signals is unable to be modified on-the-fly, meaning that changing pulsed radar parameters such as PRT and duty cycle could require long waits as Vivado processes the synthesis and implementation of the design with new parameters. Additionally, not all goals set forth during this research effort were able to be achieved. Implementation errors prevented a working prototype from being tested. Overall, this research effort provides a platform on which future work on this subject could continue, possibly solving the issues with implementation and being able to verify that a working prototype or prototypes indeed remain disciplined by a GPS source.

## 5.2 Future Work

The clearest next step in this research must be solving the errors in implementation detailed in this research effort and developing a working prototype. A working prototype or pair of prototypes could then be characterized with and without GPS discipline to verify that the GPS discipline successfully reduces jitter and frequency drift in the LFM signals generated by “dacquire”, just as this research effort has shown that the GPS discipline successfully reduces frequency and timing drift without any modifications to the 5950 unit’s block diagram. Had a working prototype been able to be developed, the lack of a second 5950 unit for testing

would have limited this research's ability to verify the synchronization capabilities being aimed for. If possible, future research should test on two separate 5950 RFSoC units to truly simulate a bistatic radar system. Additionally, improvements to the block diagram modifications would be welcome and add value and practicality. Specifically, the inability to change pulsed radar parameters from "radar_control" would add significant time in the field if a change in PRT, duty cycle, or the number of pulses per trigger were necessary. Being able to change these parameters using the 5950's command line interface would greatly reduce the amount of time needed to operate this bistatic radar system using different parameters, making the overall radar implementation much more customizable.

# Appendix A: Code

## Signal Characterization Setup:

### Frequency Drift Bash Script

```
# -----------------------------------------------------
--------

#!/bin/bash


clear

echo "Please enter number of times to run acquire.out
in MHz mode:"

read loopCount

echo "Please enter number of loops to run per
acquire.out call:"

read buffers

echo "Please enter channel mask:"

echo "0x01 for channel 1 only"

echo "0x02 for channel 2 only"

echo "0x03 for channels 1 and 2"

read chanmask


filePrefix="acquire_adc_"

filePostPrefix="_"

fileName="$filePrefix$buffers$filePostPrefix"
```

```
fileSuffix1="_ch1.dat"
fileSuffix2="_ch2.dat"

startDate=`date`

for (( i=1; i<=loopCount; i++ ))
do

clear

echo "STARTING LOOP $i "
echo ""
iniFileName="$fileName$i"
./acquire.out -ini acquire_endrun.ini -wfile
$iniFileName \
-loop $buffers -chanmask $chanmask -vchanmask
$chanmask

fullFileName1="$iniFileName$fileSuffix1"
fullFileName2="$iniFileName$fileSuffix2"

if [[ $chanmask == 0x01 ]] || [[ $chanmask == 0x03 ]];
then
echo "Starting TFTP transfer for channel 1 data"
tftp -pl $fullFileName1 192.168.0.210
echo "Deleting channel 1 data"
rm $fullFileName1
fi
```

```
if [[ $chanmask == 0x02 ]] || [[ $chanmask == 0x03 ]];
then
echo "Starting TFTP transfer for channel 2 data"
tftp -pl $fullFileName2 192.168.0.210
echo "Deleting channel 2 data"
rm $fullFileName2
fi

done

endDate=`date`

echo "Starting TFTP transfer of metadata"
tftp -pl acquire_mdata.txt 192.168.0.210
echo "Deleting metadata"
rm acquire_mdata.txt

./rfsoc2tftp.sh

echo ""
echo $startDate
echo $endDate
echo ""

#------------------------------------------------------
-------
```

Timing Drift Bash Script

```
# -----------------------------------------------------
--------

#!/bin/bash


clear

echo "Please enter number of times to run acquire.out
in PPO mode:"

read loopCount

echo "Please enter number of loops to run per
acquire.out call:"

read buffers

echo "Please enter channel mask:"

echo "0x04 for channel 3 only"

echo "0x08 for channel 4 only"

echo "0x0C for channels 3 and 4"

read chanmask


filePrefix="acquire_adc_"

filePostPrefix="_"

fileName="$filePrefix$buffers$filePostPrefix"

fileSuffix1="_ch3.dat"

fileSuffix2="_ch4.dat"


startDate=`date`


for (( i=1; i<=loopCount; i++ ))
```

```
do

clear

echo "STARTING LOOP $i "
echo ""
iniFileName="$fileName$i"
./acquire.out -ini acquire_endrunPPO.ini -wfile
$iniFileName \
-loop $buffers -chanmask $chanmask -vchanmask
$chanmask

fullFileName1="$iniFileName$fileSuffix1"
fullFileName2="$iniFileName$fileSuffix2"

if [[ $chanmask == 0x04 ]] || [[ $chanmask == 0x0C ]];
then
echo "Starting TFTP transfer for channel 3 data"
tftp -pl $fullFileName1 192.168.0.210
echo "Deleting channel 3 data"
rm $fullFileName1
fi

if [[ $chanmask == 0x08 ]] || [[ $chanmask == 0x0C ]];
then
echo "Starting TFTP transfer for channel 4 data"
tftp -pl $fullFileName2 192.168.0.210
echo "Deleting channel 4 data"
```

```
rm $fullFileName2
fi

done

endDate=`date`

echo "Starting TFTP transfer of metadata"
tftp -pl acquire_mdata.txt 192.168.0.210
echo "Deleting metadata"
rm acquire_mdata.txt

./rfsoc2tftp.sh

echo ""
echo $startDate
echo $endDate
echo ""

#-----------------------------------------------------
-------
```

GPS Single Pulse Bash Script

```
# ----------------------------------------------------------
--------
#!/bin/bash


clear
echo "Please enter number of times to run acquire.out
in PPO mode:"
read loopCount
echo "Please enter number of loops to run per
acquire.out call:"
read buffers
echo "Please enter channel mask:"
echo "0x01 for channel 1 only"
echo "0x02 for channel 2 only"
echo "0x03 for channels 1 and 2"
echo "0x04 for channel 3 only"
echo "0x08 for channel 4 only"
echo "0x0C for channels 3 and 4"
echo "0x80 for channel 8 (signal generator)"
read chanmask


filePrefix="acquire_adc_"
filePostPrefix="_"
fileName="$filePrefix$buffers$filePostPrefix"
fileSuffix1="_ch1.dat"
fileSuffix2="_ch2.dat"
```

```
fileSuffix3="_ch3.dat"
fileSuffix4="_ch4.dat"
fileSuffix5="_ch5.dat"
fileSuffix6="_ch6.dat"
fileSuffix7="_ch7.dat"
fileSuffix8="_ch8.dat"

startDate=`date`

for (( i=1; i<=loopCount; i++ ))
do

clear

echo "STARTING LOOP $i "
echo ""
iniFileName="$fileName$i"
./acquire.out -ini acquire_endrunGPS.ini -wfile
$iniFileName \
-loop $buffers -chanmask $chanmask -vchanmask
$chanmask

fullFileName1="$iniFileName$fileSuffix1"
fullFileName2="$iniFileName$fileSuffix2"
fullFileName3="$iniFileName$fileSuffix3"
fullFileName4="$iniFileName$fileSuffix4"
fullFileName5="$iniFileName$fileSuffix5"
```

```
fullFileName6="$iniFileName$fileSuffix6"

fullFileName7="$iniFileName$fileSuffix7"

fullFileName8="$iniFileName$fileSuffix8"


if [[ $chanmask == 0x01 ]] || [[ $chanmask == 0x03 ]];
then

echo "Starting TFTP transfer for channel 1 data"

tftp -pl $fullFileName1 192.168.0.210

echo "Deleting channel 1 data"

rm $fullFileName1

fi


if [[ $chanmask == 0x02 ]] || [[ $chanmask == 0x03 ]];
then

echo "Starting TFTP transfer for channel 2 data"

tftp -pl $fullFileName2 192.168.0.210

echo "Deleting channel 2 data"

rm $fullFileName2

fi


if [[ $chanmask == 0x04 ]] || [[ $chanmask == 0x0C ]];
then

echo "Starting TFTP transfer for channel 3 data"

tftp -pl $fullFileName3 192.168.0.210

echo "Deleting channel 3 data"

rm $fullFileName3

fi
```

```
if [[ $chanmask == 0x08 ]] || [[ $chanmask == 0x0C ]];
then

echo "Starting TFTP transfer for channel 4 data"

tftp -pl $fullFileName4 192.168.0.210

echo "Deleting channel 4 data"

rm $fullFileName4

fi


if [[ $chanmask == 0x80 ]]; then

echo "Starting TFTP transfer for channel 8 data"

tftp -pl $fullFileName8 192.168.0.210

echo "Deleting channel 8 data"

rm $fullFileName8

fi


done


endDate=`date`


echo "Starting TFTP transfer of metadata"

tftp -pl acquire_mdata.txt 192.168.0.210

echo "Deleting metadata"

rm acquire_mdata.txt


./rfsoc2tftp.sh


echo ""
```

```
echo $startDate

echo $endDate

echo ""

#-----------------------------------------------------
-------
```

Signal Characterization Analysis:

Main (Shared by all three experiments)

```
%% Main
%  Written by Michael Cheek (cheek@ou.edu)
%  Advised by Dr. Nathan Goodman (goodman@ou.edu)
%  Centralized MATLAB code to analyze data acquired
from Pentek 5950 RFSoC

Setup;

if mode == 0     % MHz mode is selected
    AnalysisMHz;
    Plot;
elseif mode == 1 % PPO mode is selected
    AnalysisPPO;
    PlotPPO;
elseif mode == 2 % GPS Calibration mode is selected
    AnalysisGPS;
else
    error("Please select an appropriate analysis
mode");
end
```

Setup (Shared by all three experiments)

```
%% Setup
%  Written by Michael Cheek (cheek@ou.edu) and
%  Dr. Nathan Goodman (goodman@ou.edu)
%  MATLAB code to initialize variables and parameters
to analyze data
%  acquired from Pentek 5950 RFSoC

% Clear MATLAB workspace
clear;
clc;

% Default Values
% -----------------------------------------------------
---------------------
% UNLESS SOMETHING VERY DRASTIC HAS CHANGED, THESE
SHOULD BE THE ONLY
% VALUES YOU SHOULD NEED TO CHANGE. FOR ONE OFF
CHANGES, THEY CAN BE
% CHANGED AT THE BEGINNING OF THIS SCRIPT'S RUNTIME.
% -----------------------------------------------------
---------------------
defaultMode = 0;
defaultLoops = 1;
defaultNCaptureMHz = 1000;
defaultNCapturePPO = 1;
defaultNCaptureGPS = 1;
```

```
defaultFsMHz = 1e9;
defaultFsPPO = 4e9;
defaultFsGPS = 1e9;
defaultXfersize = 65536;
% defaultXfersizeGPS = 1073741824;
% defaultXfersizeGPS = 536870912;
defaultXfersizeGPS = 268435456;
% defaultXfersizeGPS = 134217728;
defaultWsize = defaultXfersize;
defaultNumBits = 16;
defaultPath = 'D:/TFTP/';
% -----------------------------------------------------
---------------------

% User Inputs
disp('Press enter to continue with default values.');

% Mode (MHz or PPO)
mode = input("MHz [0], PPO [1], or GPS [2] (default "
+ string(defaultMode) + "): ");
if isempty(mode)
    mode = defaultMode; end

% Update mode-specific default values;
if mode == 1
    defaultNCapture = defaultNCapturePPO;
    defaultFs = defaultFsPPO;
```

```
elseif mode == 2
    defaultNCapture = defaultNCaptureGPS;
    defaultFs = defaultFsGPS;
    defaultXfersize = defaultXfersizeGPS;
    defaultWsize = defaultXfersizeGPS;
else
    defaultNCapture = defaultNCaptureMHz;
    defaultFs = defaultFsMHz;
end

% Loops
loops = input("Number of loops              (default "
+ string(defaultLoops) + "): ");
if isempty(loops)
    loops = defaultLoops; end

% Number of slow time captures per loop
nCapture = input("Number of snapshots
(default " + string(defaultNCapture) + "): ");
if isempty(nCapture)
    nCapture = defaultNCapture; end

% ADC Sampling Rate
Fs = input("ADC Sampling Rate            (default " +
string(defaultFs) + "): ");
if isempty(Fs)
    Fs = defaultFs; end
```

```
% Transfer size used in acquire.out

xfersize = input("xfersize
(default " + string(defaultXfersize) + "): ");

% Number of bytes per buffer

if isempty(xfersize)

    xfersize = defaultXfersize; end


% Write size used in acquire.out

wsize = input("wsize                          (default "
+ string(defaultWsize) + "): ");

% Number of bytes written per buffer (aka 1 buffer = 1
snapshot)

if isempty(wsize)

    wsize = defaultWsize; end


% Number of bits per datapoint

numBits = input("Number of bits per sample    (default
" + string(defaultNumBits) + "): ");

if isempty(numBits)

    numBits = defaultNumBits; end


% Folder to take data from

disp("Folder name within " + string(defaultPath) + "
(default " + string(defaultPath) + "): ");

dirs = dir(defaultPath);

dirs(1:2) = [];

dirs = dirs([dirs.isdir]);

dirs = {dirs.name};
```

```
disp(dirs); % Displays available folders

folder = string(defaultPath) + input("","s");


% Total script run time on RFSoC

% totalRunTimeSeconds = input("Total script run time
in seconds: ");

startTime = input("Time script started on RFSoC ([Y M
D H MI S]): ");

if isempty(startTime)

    startTime = [2023 1 1 0 0 0]; end

startTime = datetime(startTime);

endTime = input("Time script ended on RFSoC   ([Y M D
H MI S]): ");

if isempty(endTime)

    endTime = [2023 1 1 1 0 0]; end

endTime = datetime(endTime);


% Initializations

bytes_per_sample = numBits/8; % Number of bytes per
int16 sample

samples_per_snapshot = wsize/bytes_per_sample; %
Number of ADC samples written from each snapshot

f_ref = 10e6; % Ideal frequency of the reference
signal

T_buffer = 0.01;  % The time duration of the full
buffer in a single snapshot

% T_buffer = samples_per_snapshot/Fs;

fftLength = 2^24;

t_axis_snapshot = (0:(samples_per_snapshot - 1)).'/Fs;
%Time axis for 1 snapshot at the ADC sample rate
```

```
t_cap = (0:(nCapture-1))'/Fs;

frequencyDomain = -0.5*(1/T_buffer) + (0:(fftLength-
1))'*(1/T_buffer)/fftLength;

xcorrSize = 10000; % Number of lags to calculate in
PPO mode

minPulseHeight = 1e4; % Minimum pulse height - used
for excluding
                      % datasets that don't include
GPS time-triggered
                      % pulses on both channels in GPS
mode



% Loop Parameters

allDeltaFCh1 = zeros(loops, 1);

allDeltaFCh2 = zeros(loops, 1);

allDeltaF = zeros(loops, 1);

allCrossCorr = zeros(2*xcorrSize+1,loops);

allMaxLags = zeros(loops,1);

secondsPerLoop = 1.3015;

secondsPerLoopPPO = 0.0025;

averageTemp = 0;

totalRunTimeHours = hours(endTime - startTime);

allLagTimeCh1 = NaN(loops, 1);

allLagTimeCh2 = NaN(loops, 1);

allDeltaLag = NaN(loops, 1);


% Quality Control Parameters
```

```
% DO NOT ERASE ---------------------------------------
---------------------
randomSampleIteration = ceil(loops*rand(1));
% DO NOT ERASE ---------------------------------------
---------------------
% randomSampleIteration = loops;
sampleIterationString = string(randomSampleIteration)
+ ' of ' + string(loops);


% Structure Setup
setup = struct();
setup.Fs = Fs;
setup.t_axis_snapshot = t_axis_snapshot;
setup.f_ref = f_ref;
setup.nCapture = nCapture;
setup.T_buffer = T_buffer;
setup.fftLength = fftLength;
setup.randomSampleIteration = randomSampleIteration;
setup.sampleIterationString = sampleIterationString;
```

Analysis (Frequency Drift)

```
%% Data Analysis Loop
%  Written by Michael Cheek (cheek@ou.edu) and
%  Dr. Nathan Goodman (goodman@ou.edu)
%  MATLAB code to analyze MHz data acquired from
Pentek 5950 RFSoC

for index = 1:loops
    % ----------------------------------------------------
---------------------
    % Data Acquisition
    % ----------------------------------------------------
---------------------

    clc;
    disp("Loop " + string(index) + "/" +
string(loops));
    disp(string(floor(100*index/loops)) + "%
complete");
    minutesRemaining = (loops-
index)*secondsPerLoop/60;
    disp("Approximately " +
string(ceil(minutesRemaining)) + " minutes
remaining");
    oldAvg = averageTemp;
    averageTemp = mean(allDeltaF(1:index));
    disp("Average Delta F so far: " +
string(averageTemp) + " Hz");
```

```
    disp("Change in average from last loop: " +
string(averageTemp-oldAvg) + " Hz");
    setup.loopNumber = index;


    %Channel 1 Data
    fileName1 = folder + "/acquire_adc_" ...
        + num2str(nCapture) + "_" + num2str(index) +
"_ch1.dat";
    dataCh1 = readdata(fileName1, numBits);
    dataCh1 =
reshape(dataCh1,samples_per_snapshot,[]);  %Reshape
the data


    %Channel 2 Data
    fileName2 = folder + "/acquire_adc_" ...
        + num2str(nCapture) + "_"  + num2str(index) +
"_ch2.dat";
    dataCh2 = readdata(fileName2, numBits);
    dataCh2 =
reshape(dataCh2,samples_per_snapshot,[]);  %Reshape
the data


    % --------------------------------------------------
---------------------
    % Data Analysis
    % --------------------------------------------------
---------------------


    setup.channel = 1;
    snapCh1 = rfsoc_analyze(dataCh1, setup);
```

```
    fftData1 =
db(abs(fftshift(fft(snapCh1,fftLength))));
    setup.channel = 2;
    snapCh2 = rfsoc_analyze(dataCh2, setup);
    fftData2 =
db(abs(fftshift(fft(snapCh2,fftLength))));

    [~, maxFrequency1] = max(fftData1);
    [~, maxFrequency2] = max(fftData2);
    deltaFCh1 = frequencyDomain(maxFrequency1);
    deltaFCh2 = frequencyDomain(maxFrequency2);
    deltaF = deltaFCh2 - deltaFCh1;
    allDeltaFCh1(index,:) = deltaFCh1; %#ok<SAGROW>
    allDeltaFCh2(index,:) = deltaFCh2; %#ok<SAGROW>
    allDeltaF(index,:) = deltaF; %#ok<SAGROW>

    % ------------------------------------------------------
---------------------
    % Quality Control
    % ------------------------------------------------------
---------------------

    if index == randomSampleIteration
        sampleSnap1 = snapCh1;
        sampleSnap2 = snapCh2;
        sampleFFT1 = fftData1;
        sampleFFT2 = fftData2;
        sampleMaxFreq1 = maxFrequency1;
```

```
        sampleMaxFreq2 = maxFrequency2;
    end

end
```

rfsoc_analyze (Frequency Drift)

```
%% RFSoC_Analyze

%  Written by Dr. Nathan Goodman (goodman@ou.edu)

%  Adapted into a function by Michael Cheek
(cheek@ou.edu)

%  Analyze phase of each snapshot to determine
frequency drift over time


function snap_out_corrected = rfsoc_analyze(a, setup)


h_mf = cos(2*pi*setup.f_ref*setup.t_axis_snapshot) -
1j*sin(2*pi*setup.f_ref*setup.t_axis_snapshot); %
Create a 10 MHz sinusoid for correlating with the
snapshots

snap_out = h_mf.'*a; % Correlate the data with 10 MHz
signal - this is essentially a downconversion and LPF


t_buffer = (0:(setup.nCapture-1))*0;       % The time
between snapshots converted to a time array

h_st = exp(-1j*2*pi*setup.f_ref*t_buffer); % Create a
10 MHz sinusoid at the snapshot rate for compensating
phase

snap_out_corrected = snap_out.*h_st;       % Apply the
phase correction to the outputs from each snapshot
```

Plot (Frequency Drift)

```
%% Plot Results (MHz Mode)
%  Written by Michael Cheek (cheek@ou.edu) and
%  Dr. Nathan Goodman (goodman@ou.edu)
%  MATLAB code to plot analyses of data acquired from
Pentek 5950 RFSoC

%% Delta F
timeAxis = (1:loops)*(totalRunTimeHours/loops);
figure('Name','Delta F','NumberTitle','off');
plot(timeAxis,allDeltaF);
hold on
plot(timeAxis,allDeltaFCh1);
plot(timeAxis,allDeltaFCh2);
hold off
xlabel('Hours');
ylabel('delta F, Hz');
xlim([0 totalRunTimeHours+(totalRunTimeHours/loops)]);
maxDFCh1 = max(abs(allDeltaFCh1));
maxDFCh2 = max(abs(allDeltaFCh2));
maxDF = max(abs(allDeltaF));
maxOverall = max([maxDFCh1 maxDFCh2 maxDF]);
ylim([-(1.5*maxOverall) (1.5*maxOverall)]);
title('\deltaF values over ' + string(loops) + '
loops')
subtitle_text = {'Average \deltaF (differential) was '
+ string(mean(allDeltaF)) ...
```

```
    + ' Hz over ' + string(totalRunTimeHours) + '
hours',

    'Standard Deviation (differential) was ' +
string(std(allDeltaF)) + ' Hz'};

subtitle(subtitle_text);

legend('Differential', 'Channel 1', 'Channel 2')

grid on;


%% Phase Samples

% Plot the result from downconverting each snapshot
and applying phase correction

% This should show a constant phase for a true 10 Mhz
signal, or slowly rotating phase

% for a reference signal that is slightly different
than 10 MHz

figure('Name','Sample Phase
Data','NumberTitle','off');

subplot(2,1,1)

plot(real(sampleSnap1));

hold on

plot(imag(sampleSnap1));

plot(real(sampleSnap2));

plot(imag(sampleSnap2));

hold off

title('Sample Phase Data from Iteration ' +
sampleIterationString)

xlabel('Snippet Number')

ylabel('10 MHz Matched Sinusoid Output')

legend('Ch1 Real', 'Ch1 Imaginary', 'Ch2 Real', 'Ch2
Imaginary')
```

```
grid on;


subplot(2,1,2)
plot(frequencyDomain,sampleFFT1);
hold on
plot(frequencyDomain,sampleFFT2);
hold off
maxMaxFreq = max([sampleMaxFreq1 sampleMaxFreq2]);
minMaxFreq = min([sampleMaxFreq1 sampleMaxFreq2]);
diffMaxFreq = maxMaxFreq - minMaxFreq;
% DO NOT ERASE ---------------------------------------
---------------------
% minX = frequencyDomain(minMaxFreq -
abs(floor(diffMaxFreq*3)));
% maxX = frequencyDomain(maxMaxFreq +
abs(ceil(diffMaxFreq*3)));
% xlim([minX maxX])
% DO NOT ERASE ---------------------------------------
---------------------
title('Sample Phase FFT Data from Iteration ' +
sampleIterationString)
xlabel('Frequency (Hz)')
ylabel('Magnitude')
legend('Channel 1', 'Channel 2')
grid on;
```

Analysis (Timing Drift)

```
%% Data Analysis Loop (PPO Mode)
%  Written by Michael Cheek (cheek@ou.edu) and
%  Dr. Nathan Goodman (goodman@ou.edu)
%  MATLAB code to analyze PPO data acquired from
Pentek 5950 RFSoC

for index = 1:loops
    % -----------------------------------------------------
---------------------
    % Data Acquisition
    % -----------------------------------------------------
---------------------

    clc;
    disp("Loop " + string(index) + "/" +
string(loops));
    disp(string(floor(100*index/loops)) + "%
complete");
    minutesRemaining = (loops-
index)*secondsPerLoopPPO/60;
    disp("Approximately " +
string(ceil(minutesRemaining)) + " minutes
remaining");
    oldAvg = averageTemp;
    averageTemp = mean(allMaxLags(1:index));
    disp("Average Max Lag so far: " +
string(averageTemp));
```

```
    disp("Change in average from last loop: " +
string(averageTemp-oldAvg));


    %Channel 1 Data
    fileName1 = folder + "/acquire_adc_" ...
        + num2str(nCapture) + "_" + num2str(index) +
"_ch3.dat";
    dataCh1 = readdata(fileName1, numBits);
    dataCh1 =
reshape(dataCh1,samples_per_snapshot,[]);  %Reshape
the data


    %Channel 2 Data
    fileName2 = folder + "/acquire_adc_" ...
        + num2str(nCapture) + "_"  + num2str(index) +
"_ch4.dat";
    dataCh2 = readdata(fileName2, numBits);
    dataCh2 =
reshape(dataCh2,samples_per_snapshot,[]);  %Reshape
the data


    % ------------------------------------------------
--------------------
    % Data Analysis
    % ------------------------------------------------
--------------------


    [crosscorr, lags] = xcorr(dataCh1, dataCh2,
xcorrSize);
    allCrossCorr(:,index) = crosscorr; %#ok<SAGROW>
```

```
    [~, maxLagIdx] = max(crosscorr);
    maxLag = lags(maxLagIdx);
    allMaxLags(index,:) = maxLag; %#ok<SAGROW>

    % ------------------------------------------------------
---------------------
    % Quality Control
    % ------------------------------------------------------
---------------------

    if index == randomSampleIteration
        sampleCrossCorr = crosscorr;
        sampleLags = lags;
        sampleMaxLag = maxLag;
        sampleMaxLagIdx = maxLagIdx;
    end

end
```

Plot (Timing Drift)

```
%% Plot Results (PPO Mode)
%  Written by Michael Cheek (cheek@ou.edu)
%  MATLAB code to plot analyses of data acquired from
Pentek 5950 RFSoC


%% Max Lag
timeAxis = (1:loops)*(totalRunTimeHours/loops);
figure('Name','Max Lag','NumberTitle','off');
plot(timeAxis,allMaxLags);
% hold on
% plot(timeAxis,smoothdata(allMaxLags,'loess',200));
% hold off
xlabel('Hours');
ylabel('Lag at which autocorrelation peaked');
xlim([0 totalRunTimeHours+(totalRunTimeHours/loops)]);
maxOverall = max(abs(allMaxLags));
ylim([-(1.5*maxOverall) (1.5*maxOverall)]);
% title('Maximum Cross-correlation Lag values over ' +
string(loops) + ' loops')
title('Maximum Cross-correlation Lag values over ' +
string(totalRunTimeHours) + ' hours')
subtitle('Average Lag was ' + string(mean(allMaxLags))
+ ...
    ', Standard Deviation of Lag was ' +
string(std(allMaxLags)));
% legend('Original', 'Smoothed')
```

```
grid on;


%% Max Lag after 1 hour

figure('Name','Max Lag','NumberTitle','off');

plot(timeAxis,allMaxLags);

xlabel('Hours');

ylabel('Lag at which autocorrelation peaked');

xlim([0 1]);

maxOverall = max(abs(allMaxLags));

ylim([-(1.5*maxOverall) (1.5*maxOverall)]);

title('Maximum Cross-correlation Lag values over the
first hour')

subtitle('Average Lag was ' +
string(mean(allMaxLags(1:247))) + ...

    ', Standard Deviation of Lag was ' +
string(std(allMaxLags(1:247))));

grid on;


%% Max Lag once GPS Lock Achieved

figure('Name','Max Lag post GPS
Lock','NumberTitle','off');

plot(timeAxis,allMaxLags);

% plot(allMaxLags);

xlabel('Hours');

ylabel('Lag at which autocorrelation peaked');

xlim([0.65
totalRunTimeHours+(totalRunTimeHours/loops)]);

maxOverall = max(abs(allMaxLags));
```

```
ylim([-(1.5*maxOverall) (1.5*maxOverall)]);

title('Maximum Cross-correlation Lag values once GPS
Lock was Achieved')

txt = {'Average Lag was ' +
string(mean(allMaxLags(155:end))) + ...

    ', Standard Deviation of Lag was ' +
string(std(allMaxLags(155:end)))};

subtitle(txt);

grid on;


%% Sample Cross-Correlation

figure('Name','Sample Xcorr
Data','NumberTitle','off');

plot(sampleLags,abs(sampleCrossCorr));

title('Sample Cross-Correlation Data from Iteration '
+ sampleIterationString)

xlabel('Lags')

ylabel('Cross-Correlation')

grid on;


%% All Cross-Correlations

figure('Name','All Cross-
Correlations','NumberTitle','off')

imagesc(allCrossCorr);

colorbar;

title('All Cross-Correlations over ' + string(loops) +
' loops')

xlabel('Loops')

ylabel('Cross-Correlation between EndRun 10MPPS
signals')
```

Analysis (GPS Single Pulse)

```
%% Data Analysis Loop
%  Written by Michael Cheek (cheek@ou.edu) and
%  Dr. Nathan Goodman (goodman@ou.edu)
%  MATLAB code to analyze MHz data acquired from
Pentek 5950 RFSoC


for index = 1:loops
    % -----------------------------------------------------
---------------------
    % Data Acquisition
    % -----------------------------------------------------
---------------------


    clc;
    disp("Loop " + string(index) + "/" +
string(loops));
    disp(string(floor(100*index/loops)) + "%
complete");


    %Channel 1 Data
    fileName1 = folder + "/acquire_adc_" ...
        + num2str(nCapture) + "_" + num2str(index) +
"_ch3.dat";
    dataCh1 = readdata(fileName1, numBits);
    dataCh1 =
reshape(dataCh1,samples_per_snapshot,[]);  %Reshape
the data
```

```
    %Channel 2 Data
    fileName2 = folder + "/acquire_adc_" ...
        + num2str(nCapture) + "_"  + num2str(index) +
"_ch4.dat";
    dataCh2 = readdata(fileName2, numBits);
    dataCh2 =
reshape(dataCh2,samples_per_snapshot,[]);  %Reshape
the data

    % -----------------------------------------------------
---------------------
    % Data Analysis
    % -----------------------------------------------------
---------------------
    [lagHeightCh1, ~] = max(dataCh1(20:end));
    [lagHeightCh2, ~] = max(dataCh2(20:end));
    if lagHeightCh1 >= minPulseHeight
        if lagHeightCh2 >= minPulseHeight
            [~, lagIdxCh1] =
max(gradient(dataCh1(20:end)));
            [~, lagIdxCh2] =
max(gradient(dataCh2(20:end)));
            allLagTimeCh1(index) = lagIdxCh1/Fs;
%#ok<SAGROW>
            allLagTimeCh2(index) = lagIdxCh2/Fs;
%#ok<SAGROW>
            allDeltaLag(index) = allLagTimeCh1(index)
- allLagTimeCh2(index); %#ok<SAGROW>

        end
```

```
    end

end

allDeltaLag = rmmissing(allDeltaLag);
allDeltaLag(allDeltaLag==0) = [];

figure;
histogram(allDeltaLag)
title('Lag values between Channels 1 and 2')
subtitle('Standard Deviation: ' +
string(std(allDeltaLag)) ...
    + ' over ' + string(totalRunTimeHours) + '
hours');
```

Plot (GPS Single Pulse)

```
timeAxis = (1:1e4)/1e9;

figure('Name','Dataset ' +
string(index),'NumberTitle','off');

plot(timeAxis,dataCh1(1:1e4));

hold on

plot(timeAxis,dataCh2(1:1e4));

% plot(gradient(dataCh1(20:2e6)));

hold off

legend('Channel 1', 'Channel 2')

title("Absolute GPS Time-Triggered Pulses")

xlabel("seconds")

grid on;
```

## Solution Code

### radar_control.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////
////////////////////////////
// Company: The University of Oklahoma - Advanced
Radar Research Center
// Engineer: Michael Ortiz-Cheek (cheek@ou.edu,
mcheek0@icloud.com)
// Advisor: Dr. Nathan Goodman (goodman@ou.edu)
//
// Create Date: 03/28/2023 10:55:01 AM
// Design Name:
// Module Name: radar_control
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 4.0 - Fixed blocking assignments in always
block
// Additional Comments:
```

```
//
/////////////////////////////////////////////////////////
/////////////////////////////

module radar_control(
    input clk,
    input reset,
    input user_ready,
    input gps_trigger,
    input dac_ready,
    input adc_ready,
    output dac_trigger,
    output adc_trigger,
    output pulses_complete
    );

    reg dac_enable_internal;
    reg adc_enable_internal;
    reg pulses_complete_internal;
    reg in_pri;

    reg [7:0] slow_counter;
    reg [7:0] pulse_limit; // Number of scans in the
CPI
```

```
    // Fast Counter information: FPGA clock runs at
x_MHz, so in order to enable DAC for y_ms and then ADC
for z_ms

    //                           with an A_ms offset,
dac_limit must be x*y*10^3 and adc_limit must be

    //                           dac_adc_offset +
x*z*10^3. Total PRI must >= dac_limit + adc_limit -
dac_adc_offset.

    //                           If ADC should be
disabled until DAC is complete, set dac_adc_offset to
dac_limit.

    //                           *** Please note that
when used in conjunction with axis_dac_switch.v, these
values ***

    //                           *** do not control
duty cycle, they are only for controlling rising edge
timing.   ***

    //                           *** The RFSoC might
have duty cycle requirements for trigger signals.
***

    //                           x = 200

    //                           y = 0.10

    //                           z = 0.50

    //                           A = 0.00

    reg [31:0] fast_counter;

    reg [31:0] dac_limit;

    reg [31:0] adc_limit;

    reg [31:0] dac_adc_offset; // must be greater than
0.

    reg [31:0] pri_limit;
```

```
    // This always block handles counting clock cycles
for timing.
    always @(posedge clk) begin
        if (reset) begin
          fast_counter <= 32'd0;
          slow_counter <= 8'd0;
          pulse_limit <= 8'd10; // Number of scans in
the CPI
          dac_limit <= 32'd100000;
          adc_limit <= 32'd100000;
          dac_adc_offset <= 32'd1; // must be greater
than 0.
          pri_limit <= 32'd200000;
          dac_enable_internal <= 1'b0;
          adc_enable_internal <= 1'b0;
          pulses_complete_internal <= 1'b0;
          in_pri <= 1'b0;
        end // reset

        else begin // not in reset condition
          // Increments fast_counter
          if (in_pri) begin
              fast_counter <= fast_counter + 32'd1;
          end // increments fast counter

          // This if block handles the trigger from
the GPS unit.
          if (gps_trigger) begin
```

```
              // if all ready signals are asserted
high, enable the DAC.
              if (user_ready && dac_ready && adc_ready
&& (slow_counter == 8'd0) && (fast_counter == 32'd0))
begin
                  dac_enable_internal <= 1'b1;
                  in_pri <= 1'b1;
                  pulses_complete_internal <= 1'b0;
              end // if all ready signals are asserted
high


          end // if (gps_trigger)


          // if the DAC is enabled
          if (dac_enable_internal) begin


              // if DAC timer is complete
              if (fast_counter >= dac_limit) begin
                  dac_enable_internal <= 1'b0; //
disable DAC
              end // if DAC timer is complete


          end // if the DAC is enabled


          // if it's time to enable the ADC
          if ((fast_counter >= dac_adc_offset) &&
(fast_counter < adc_limit + dac_adc_offset) &&
adc_ready) begin
```

```
              adc_enable_internal <= 1'b1; // enable
ADC
          end // enable ADC

          // if the ADC is enabled
          if (adc_enable_internal) begin

              // if ADC timer is complete
              if (fast_counter >= (adc_limit +
dac_adc_offset)) begin
                  adc_enable_internal <= 1'b0; //
disable ADC
              end // if ADC timer is complete

          end // if the ADC is enabled

          // if the PRI is complete
          if ((fast_counter >= pri_limit) && in_pri)
begin
              fast_counter <= 32'd0; // reset the fast
counter
              in_pri <= 1'b0;
              slow_counter <= slow_counter + 8'd1; //
increment slow counter

              // if the pulse train is complete
              if (slow_counter >= (pulse_limit -
8'd1)) begin
                  pulses_complete_internal <= 1'b1; //
set Pulses Complete flag
```

```
                  slow_counter <= 8'b0; // reset slow
counter
                  dac_enable_internal <= 1'b0; // just
to be safe, disable DAC
                  adc_enable_internal <= 1'b0; // just
to be safe, disable ADC
              end // if pulse train is complete

              else begin // if pulse train is not
complete

//                  if ((pulses_complete_internal ==
1'b0) && (slow_counter >= 8'b0)) begin
                  if (pulses_complete_internal ==
1'b0) begin
                      in_pri <= 1'b1; // restart PRI
                      dac_enable_internal <= 1'b1; //
restart DAC for next repitition
                  end

              end // if pulse train is not complete

          end // if the PRI is complete

        end // if not in reset condition

    end // always @(posedge clk)

    assign dac_trigger = dac_enable_internal;
```

```
    assign adc_trigger = adc_enable_internal;
    assign pulses_complete = pulses_complete_internal;

endmodule
```

radar_control_reset.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////
////////////////////////////
// Company: The University of Oklahoma - Advanced
Radar Research Center
// Engineer: Michael Ortiz-Cheek (cheek@ou.edu,
mcheek0@icloud.com)
// Advisor: Dr. Nathan Goodman (goodman@ou.edu)
//
// Create Date: 08/24/2023 02:43:35 AM
// Design Name:
// Module Name: radar_control_reset
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////
////////////////////////////
```

```
module radar_control_reset(
    input clk,
    output reset_out
    );

    reg reset_internal = 1'b0;
    reg [7:0] disposable_counter = 8'd0;
    reg [7:0] start_count = 8'd8;
    reg [7:0] end_count = 8'd10;

    always @(posedge clk) begin

        if (disposable_counter < start_count) begin
            reset_internal <= 1'b0;
            disposable_counter <= disposable_counter +
8'd1;
        end else if (disposable_counter < end_count)
begin
            reset_internal <= 1'b1;
            disposable_counter <= disposable_counter +
8'd1;
        end else begin
            reset_internal <= 1'b0;
        end

    end
```

```
    assign reset_out = reset_internal;


endmodule
```

axis_dac_switch.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////
////////////////////////////
// Company: The University of Oklahoma - Advanced
Radar Research Center
// Engineer: Michael Ortiz-Cheek (cheek@ou.edu,
mcheek0@icloud.com)
// Advisor: Dr. Nathan Goodman (goodman@ou.edu)
//
// Create Date: 03/27/2023 04:09:38 PM
// Design Name:
// Module Name: axis_dac_switch
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////
////////////////////////////
```

```
module axis_dac_switch(
    input [255:0] tdata_in,
    output[255:0] tdata_out,
    input tvalid_in,
    input fifo_full,
    output tvalid_out,
    output tready_in,
    input tready_out,
    input clk,
    input enable
    );

    reg [255:0] tdata_out_internal = 256'b0;
    reg [0:0] tvalid_out_internal = 1'b1;
    reg [0:0] tready_in_internal = 1'b0;
    reg [0:0] enable_old = 1'b0;

    reg [31:0] sample_counter = 32'd0;
    reg [31:0] sample_limit = 32'd32768;

//    always @(posedge enable) begin
//        if (tvalid_in == 1 && fifo_full == 1) begin

//            tready_in_internal <= 1'b1;

//        end
```

```
//    end

    always @(posedge clk) begin

        enable_old <= enable;
        if (enable) begin
            if (tvalid_in == 1'b1 && fifo_full == 1'b1
&& enable_old == 1'b0) begin
                tready_in_internal <= 1'b1;
            end
        end

        if (tvalid_in == 1 && tready_out == 1) begin

            tvalid_out_internal <= 1'b1;

            if (tready_in_internal == 1) begin

                tdata_out_internal <= tdata_in;
                sample_counter <= sample_counter + 1;

                if (sample_counter >= sample_limit)
begin
                    tready_in_internal <= 1'b0;
                    sample_counter <= 32'd0;
                end
```

```
            end

            else begin
                tdata_out_internal <= 256'b0;
            end

        end

        else begin
            tvalid_out_internal <= 1'b0;
        end

    end

    assign tdata_out = tdata_out_internal;
    assign tvalid_out = tvalid_out_internal;
    assign tready_in = tready_in_internal;

endmodule
```

# References


[1] N. J. Willis, Bistatic Radar, Raleigh, NC: SciTech Publishing, Inc., 2005.

[2] G. W. Stimson, Introduction to Airborne Radar, 2nd Edition, Raleigh, NC: SciTech Publishing, Inc., 1998.

[3] H. Yulin, Y. Jianyu, W. Junjie and X. Jintao, "Precise time frequency synchronization technology," *Journal of Systems Engineering and Electronics,* vol. 19, no. 5, p. pp.929–933, 2008.

[4] J. Kim, J. Chun and I. Choi, "Time and Frequency Synchronization of Bistatic FMCW Radar," Agency for Defense Development, Daejeon, South Korea.

[5] A. D. Byrd, R. D. Palmer and C. J. Fulton, "Development of a Low-Cost Multistatic Passive Weather Radar Network," *IEEE Transactions on Geoscience and Remote Sensing,* vol. 58, no. 4, pp. 2796-2808, 2020.

[6] Glen, Interviewee, *Support call with EndRun Technologies.* [Interview].

[7] EndRun Technologies, "Ninja Precision Timing Module User Manual".

[8] B. Marcotte, Interviewee, *Pentek 5950 RFSoC Timing Support.* [Interview]. 17 November 2022.

[9] R. Sgandurra, "Strategies for Deploying Xilinx's Zynq Ultrascale+ RFSoC," Pentek, Upper Saddle River, NJ.

[10] "The difference between Implementation and Synthesize," Xilinx, 7 January 2019. [Online]. Available: https://support.xilinx.com/s/question/0D52E00006hpkc2SAA/the-difference-between-implementation-and-synthesize?language=en_US.

[11] E. Worthman, "It's all IP in an SoC," Semiconductor Engineering, 5 June 2014. [Online]. Available: https://semiengineering.com/its-all-ip-in-an-soc/.

[12] ARM, "AMBA AXI and ACE Protocol Specifications," ARM.

[13] FPGA Site, "Xilinx AXI Stream Tutorial - Part 1," 15 July 2017. [Online]. Available: http://fpgasite.blogspot.com/2017/07/xilinx-axi-stream-tutorial-part-1.html.

[14] Xilinx, "pb040-xilinx-com-ip-xlconstant.pdf - Viewer - AMD Adaptive Computing Documentation Portal," 9 April 2018. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pb040-xilinx-com-ip-xlconstant.

[15] Xilinx, "pg164-proc-sys-reset - Viewer - AMD Adaptive Computing Documentation Portal," 18 November 2015. [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg164-proc-sys-reset.

[16] Xilinx, "FPGA Bitstream," Xilinx, 4 April 2018. [Online]. Available: https://www.xilinx.com/htmldocs/xilinx2018_1/SDK_Doc/SDK_concepts/concept_fpgabitstream.html.

[17] "[BD 41-238] FREQ_HZ do not match," Xilinx, 18 March 2021. [Online]. Available: https://support.xilinx.com/s/question/0D52E00006hpQaGSAU/bd-41238-freqhz-do-not-match?language=en_US.

[18] "Vivado 2018.3 crash: ERROR: [Common 17-69] Command failed: Failed to create design checkpoint," Xilinx, 15 July 2019. [Online]. Available: https://support.xilinx.com/s/question/0D52E00006lLwqCSAS/vivado-20183-crash-error-common-1769-command-failed-failed-to-create-design-checkpoint?language=en_US.

[19] "[Common 17-49] Internal Data Exception," Xilinx, 20 December 2017. [Online]. Available: https://support.xilinx.com/s/question/0D52E00006lLh7qSAC/common-1749-internal-data-exception?language=en_US.

[20] "Implementation error [Common 17-49]," Xilinx, 16 March 2018. [Online]. Available: https://support.xilinx.com/s/question/0D52E00006iI5eLSAS/implementation-error-common-1749?language=en_US.

[21] "[Common 17-49] Internal Data Exception," Xilinx, 4 November 2019. [Online]. Available: https://support.xilinx.com/s/question/0D52E00006iHpgkSAC/common-1749-internal-data-exception?language=en_US.

[22] C. E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles that Kill!," 2000. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA_rev1_2.pdf.