THESIS

OPTIMIZATION OF A CYCLOSTATIONARY SIGNAL PROCESSING ALGORITHM USING MULTIPLE FIELD PROGRAMMABLE GATE ARRAYS ON THE SRC-6 RECONFIGURABLE COMPUTER

by

Wesley A. Simon

September 2009

Thesis Advisor:	Douglas J. Fouts
Co-Advisor:	Phillip E. Pace

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE Optimization of Cyclostationary Signal Processing Algorithms Using Multiple Field Programmable Gate Arrays on the SRC-6 Reconfigurable Computer.	**5. FUNDING NUMBERS**
6. AUTHOR(S) Wesley A. Simon	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Center for Joint Services Electronic Warfare, Naval Postgraduate School, Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis implements a cyclostationary estimation technique called the time-smoothing FFT accumulation method on a reconfigurable computer to generate a frequency vs. cycle frequency approximation of the input signal. This signal processing method can be used to identify signal modulation type and extract the parameters of low probability of intercept signals in electronic intelligence discrimination receivers. This implementation builds on previous work at the Naval Postgraduate School and focuses on reducing the overall runtime to approach real-time processing. The focus of the implementation is to utilize dual field programmable gate arrays (FPGAs) within a single multi-adaptive processor (MAP). Hardware decisions are made by analyzing the relationships between frequency resolution, Grenander's Uncertainly Condition and desired cycle frequency resolution. Implemented on the SRC-6 reconfigurable computer utilizing Xilinx Virtex 2 FPGAs, this work uses the cyclostationary algorithm and takes advantage of the techniques for which the SRC-6 is optimized, such as pipelining, array processing and memory access techniques.

14. SUBJECT TERMS SRC-6, reconfigurable computers, FPGA, cyclostationary processing, Time-Smoothing FFT Accumulation Method.	**15. NUMBER OF PAGES** 123
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

OPTIMIZATION OF CYCLOSTATIONARY SIGNAL PROCESSING ALGORITHM USING MULTIPLE FIELD PROGRAMMABLE GATE ARRAYS ON THE SRC6 RECONFIGURABLE COMPUTER

Wesley A. Simon
Lieutenant, United States Navy
B.S., Texas A&M University, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2009

Author: Wesley A. Simon

Approved by: Douglas J. Fouts
Thesis Advisor

Phillip E. Pace
Co-Advisor

Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis implements a cyclostationary estimation technique called the time-smoothing FFT accumulation method on a reconfigurable computer to generate a frequency vs. cycle frequency approximation of the input signal. This signal processing method can be used to identify signal modulation type and extract the parameters of low probability of intercept signals in electronic intelligence discrimination receivers. This implementation builds on previous work at the Naval Postgraduate School and focuses on reducing the overall runtime to approach real-time processing. The focus of the implementation is to utilize dual field programmable gate arrays (FPGAs) within a single multi-adaptive processor (MAP). Hardware decisions are made by analyzing the relationships between frequency resolution, Grenander's Uncertainly Condition and desired cycle frequency resolution. Implemented on the SRC-6 reconfigurable computer utilizing Xilinx Virtex 2 FPGAs, this work uses the cyclostationary algorithm and takes advantage of the techniques for which the SRC-6 is optimized, such as pipelining, array processing and memory access techniques.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Advanced missiles and radars are employing low probability of intercept (LPI) techniques to appear nearly invisible to adversary intercept receiver systems. An autonomous system is under development at the Naval Postgraduate School that analyzes LPI radar signals. Repeatable, accurate, timely classification benefits electronic intelligence platforms, ships, airplanes or any military combat system. The basic design of this autonomous system is comprised of a collection of advanced algorithms that process a signal and pass along the signal modulation and the extracted signal parameters to a decision module that classifies the signal. The algorithms currently under consideration for this system are the

- Choi-Williams Signal Processing;
- Quadrature Mirror Filtering;
- Cyclostationary Signal Processing.

This thesis is a continued investigation into the cyclostationary signal processing algorithm as described in [1] and implemented in [2]. The time smoothing FFT accumulation method (FAM) was previously implemented onto a single field programmable gate array (FPGA) on a single multi-adaptive processor (MAP) onboard the SRC-6 [2]. This thesis shows that by utilizing pipelining, array processing, and memory accessing techniques, the overall runtime is significantly reduced to improve the usability of the cyclostationary FAM algorithm implemented in [2]. Consequently, the iteration runtime approaches a value that allows the cyclostationary FAM technique to be used in a real-time electronics discrimination receiver. To achieve better performance, both FPGAs on a single MAP and the data pipeline are optimized. A reduction in memory dependencies and the development of a memory management plan are two fundamental techniques that are employed to achieve a significant reduction in runtime by an order of 10 times the original FAM algorithm implemented in [2]. These techniques are not specific to the SRC-6 or Virtex FPGAs. They are techniques that can be applied on any FPGA.

Two types of LPI radar signals were used to investigate the algorithm performance. Exploring both Frank continuous phase modulation and frequency modulated continuous waveform signals generated in MATLAB (using code from [1]), hardware decisions were made when implementing the FAM in hardware.

The FAM algorithm consists of first windowing the data and performing an N' point FFT. The output data is split and modulated before being multiplied together. A second P point FFT is used to get the output spectral correlation density function.

When the sampling period is chosen by the selection of the analog-to-digital conversion hardware, the frequency and cycle-frequency resolution are set based upon the chosen Grenander's Uncertainty Condition. This also sets the FFT sizes that are used. The product of FFT sizes greater than 1024 will yield sufficient resolution in the frequency and cycle-frequency axis to give meaningful results. This decision tool assists the designer of a cyclostationary signal processing FAM algorithm to utilize the available resources to provide maximum flexibility when designing intercept receivers.

The FAM algorithm is used to identify the LPI radar modulations under nominal signal-to-noise ratios (0 to -6 dB). In addition, the extraction of the signal parameters is also important for emitter classification. Careful examination of LPI signals force choices in hardware. Smart implementation of pipelines, memory management plans, and array utilization on the SRC-6 continue to improve upon the usefulness of the cyclostationary FAM algorithm.

LIST OF ABBREVIATIONS, ACRONYMS, AND SYMBOLS

BRAM	Block RAM
CA	Cyclostationary Algorithm
CSA	Cyclostationary Spectral Analysis
DC	Dual Chip
DMA	Direct Memory Access
FAM	Cyclostationary Time Smoothing FFT Accumulation Method
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
LPI	Low Probability of Intercept
MAP	Multi-Adaptive Processor
OBM	On-Board Memory
µP	Microprocessor

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Many thanks to:

Professor Douglas J. Fouts and Professor Phillip E Pace of the Naval Postgraduate School for guiding me through the thesis process. Their guidance and teaching have empowered me with the knowledge to accomplish this project. Thank you for your patience and dedication to education .

Dave Caliga of SRC for providing countless hours of distance support in attempting to optimize the code found in this thesis and teaching me a few tricks of the trade when utilizing OBM memory.

Dan Zulaica at the Naval Postgraduate School for updating and maintaining the SRC-6 and providing around-the-clock support keeping it operational for the students at NPS nearly 100% of the time. Thanks!

The United States Navy for the opportunity to complete a Master’s degree. Without the opportunity for postgraduate education at the Naval Postgraduate School, I would not have the opportunity for a Master’s degree.

My wonderful wife, Susan, whose dedication to the Navy mission throughout the years helps make my goals a reality. Without her understanding of long work hours and time away from the family, our marriage would be nonexistent. Thanks for an unwavering friendship.

This work was supported by ONR -313, Washington D.C.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. LOW PROBABILITY OF INTERCEPT RADARS

Low probability of intercept (LPI) radars is a useful tool in the art of modern warfare. Military planners find them useful and despise their use by the opposition. By design, LPI radars are difficult to detect either because of transmitted energy levels or by using advanced techniques of signal modulation. The threat alone and the capabilities of these LPI radars is justification enough to develop techniques to detect and analyze LPI radars. Countries that develop capabilities that successfully discriminate between LPI waveforms have the upper hand in the theater of electronic intelligence.

Since the early 1980s, engineers and mathematicians have been working to implement methods to be able to successfully discriminate between LPI waveforms. There are several mathematical techniques that allow an operator to successfully 'classify' signals. Each algorithm is good for certain signal types. No one algorithm is good for all types of signal modulations and extraction of signal parameters. The need for a system to cover this task completely is a strong requirement. Professor Phillip Pace and his colleagues have been working to develop, model, and test such a system at the Naval Postgraduate School in Monterey, California. A block diagram of the proposed system is shown in Figure 1. This thesis is a continued investigation into the cyclostationary signal-processing block, which is a part of the larger project to create an autonomous system to allow for classification of non-cooperative emitters. For an in depth discussion of the other blocks, see [1]. Ultimately, the goal for the cyclostationary block is to extract accurate and reliable information about a signal that can be compared with other blocks to make autonomous decisions about a signal of interest.

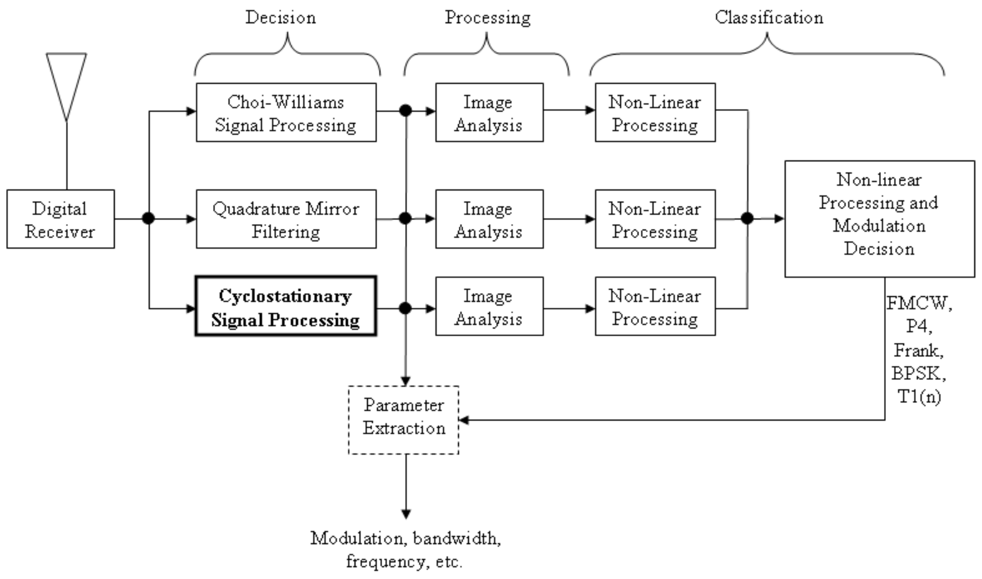

Decision
Processing
Classification
Digital Receiver
Choi-Williams Signal Processing
Quadrature Mirror Filtering
Cyclostationary Signal Processing
Image Analysis
Image Analysis
Image Analysis
Non-Linear Processing
Non-Linear Processing
Non-Linear Processing
Non-linear Processing and Modulation Decision
FMCW, P4, Frank, BPSK, T1(n)
Parameter Extraction
Modulation, bandwidth, frequency, etc.

Figure 1. Autonomous LPI System (From: [1])

B. OBJECTIVE

The objective of this thesis is to develop a functional program that implements the cyclostationary time smoothing fast Fourier transform accumulation method (FAM) algorithm found in [1], starting from the previously developed single field programmable gate array (FPGA) implementation in [2]. Two adjacent FPGAs will be utilized with the major benefit being twice the amount of available logic. The ultimate goal of this project is to reduce the overall runtime of the program to approach near real time operations. A secondary goal is to extract data at sufficient resolutions to provide meaningful results.

As a result of analyzing LPI signals, certain decisions are reached that help determine specifications for hardware and software. These decisions are the result of developing an understanding of the interrelationship between the hardware and the signals analyzed. Future designers can make smart software choices based upon the selection of data acquisition hardware and a phenomena called Grenander's Uncertainty Condition.

The cyclostationary block of Figure 1 has several tasks that must be accomplished for a successful project. Digesting the intricacies of the FAM algorithm is key to developing an efficient implementation onto any platform. Second, the algorithm must be tailored for the target platform. This is accomplished by taking advantage of functions that the target platform does well and employing known techniques for maximum efficiency. The target platform for this thesis is the SRC-6 reconfigurable computer. In particular, the target is the E series map that has two user-programmable Xilinx Virtex II FPGAs onboard.

C. RELATED WORK

As part of a larger project, this thesis continues the work and expands upon [2]. Upperman's use of a single FPGA implementation of the Cyclostationary algorithm describes several space and speed limitations that limit the useable data because of inadequate resolutions in both frequency and cycle-frequency. To summarize Upperman's work, he was able to successfully implement four versions of the FAM algorithm. The C and MATLAB models performed well and are the target of performance for this thesis. The SRC implementation is fast, but it is not as fast as the C and MATLAB models that have "unlimited memory" and a processor in the Gigahertz range. Table 1 is the timing summary of the single chip custom FFT implementation. The resulting frequency and cycle-frequency resolution are not sufficient to determine anything more than the center frequency and bandwidth approximation of the signal of interest. The code rate (R_c) is not discovered because the frequency and cycle-frequency resolutions are insufficient.

Table 1. Single Chip Timing Results

```
Execution times:
  2.170836 seconds total
Of the total time:
  1.313759 seconds were spent on the CALLS to FFTs
  0.458690 seconds were spent on the MAP for FFTs
  Of the time spent on the MAP for FFTs:
    0.382998 seconds were spent on DMAs
    0.075691 seconds were spent in the FFT loop
  0.116433 seconds were spent on the CALLS to Channelize
  0.000157 seconds were spent on the MAP for Channelize
  Of the time spent on the MAP for Channelize:
    0.000024 seconds were spent on DMAs
    0.000133 seconds were spent channelizing
  0.106429 seconds were spent on the CALLS to Downconvert
  0.003294 seconds were spent on the MAP for Downconvert
  Of the time spent on the MAP for Downconvert:
    0.000730 seconds were spent on DMAs
    0.002564 seconds were spent downconverting
Execution time not including calls and data transfers: 0.250462
seconds
Results from Cyclostationary FAM algorithm written to:
FAM_result.txt
```

This thesis is only a small block of the proposed system in Figure 1. Other theses cover the other algorithms [3], [4], [5]. After purchasing the SRC-6 reconfigurable computer, the Naval Postgraduate School began testing the suitability of this architecture for developing algorithms to discriminate LPI signals [6], [7]. To date, all work has shown that the SRC-6 and newer variants would be suitable for the advanced signal processing needed to develop the Autonomous system described by Figure 1. Upperman and Macklin agree that the SRC-6 is a difficult system to master but has great potential for signal processing algorithms.

D. THESIS ORGANIZATION

This thesis is organized by chapters that develop the project from concept to code and is laid out as follows:

- Chapter II provides information on low probability of intercept systems, cyclostationary spectral analysis, and Grenander’s Uncertainty Condition.
- Chapter III provides information on the SRC-6 reconfigurable computing including both hardware considerations, and software considerations.

- Chapter IV provides information on the SRC-6 specific techniques to include memory implementation, loops and dependencies, arrays, parallel regions, and using two logic chips.
- Chapter V provides information on the Dual Chip flow through design.
- Chapter VI provides the Timing analysis.
- Chapter VII provides information on Achieving Usable Data and Future direction.
- Chapter VIII provides a conclusion for this thesis.
- Appendix A is a datasheet that shows the subtle differences between MAPs that are compatible with the SRC-6.
- Appendices B-F provides the Dual Chip Flow Through Design code. Code is broken up into various Appendices to act as markers.

THIS PAGE INTENTIONALLY LEFT BLANK

II. LOW PROBABILITY OF DETECTION SYSTEMS

An LPI radar is defined as a radar that uses a special emitted waveform intended to prevent a non-cooperative intercept receiver from intercepting and detecting its emission [1]. LPI is also described as a property of a radar that because of its low power, wide bandwidth, and frequency variability makes it difficult to be identified by passive intercept receivers [1]. The threat of LPI radar is real and growing.

For example, designed to replace the popular P-18 radar, the Vostok-E is new up and coming mobile radar that utilizes several cutting edge technologies that define it as an LPI radar. Capable of using low-power noise-like probing signals it provides reliable protection against anti-radiation missiles (ARM) [8]. Designed in Belarus, this radar is designed to detect air contacts, measure their range, azimuth and range rate utilizing a solid-state 2D digital VHF radar. Targets are tracked in two-dimensional space, automatically classified, and integrated into networked command and control equipment. Complete with an integrated diesel generator this unit is an asset to any anti-air campaign. What makes this radar so dangerous is that it is hard to detect its emissions. In addition, it boasts enhanced jamming immunity because it employs wide dynamic range radio receiving devices and pulse-by-pulse automatic carrier frequency tuning [8]. Figure 2 is a picture of the Vostok-E in combat mode.

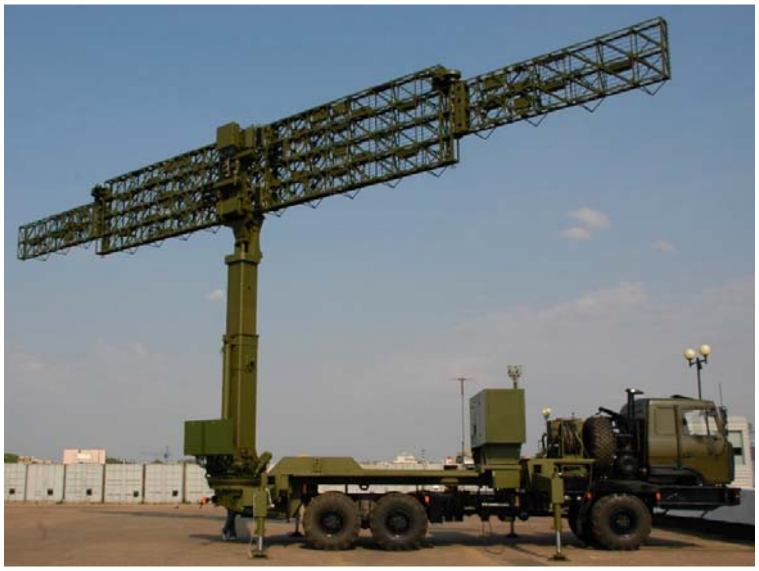


Figure 2. “Vostok-E” Mobile Solid-State 2D Digital VHF Radar (From: [8])

A subsonic LPI missile is currently under development by Saab Bofors Dynamics. The next generation RBS-15 will be a LPI version of the successful RBS-15 MK3. To make this missile LPI, the design engineers are designing the seeker using frequency-modulated continuous wave spread-spectrum technology [9]. The signals used by this radar are similar to the type analyzed in Chapter I section B. Along with LPI radar, an imaging IR seeker will be used. Another advanced feature considered for integration into this missile is a two-way data link for updating targeting data [9]. See Figure 3 for a diagram of the RBS-15.

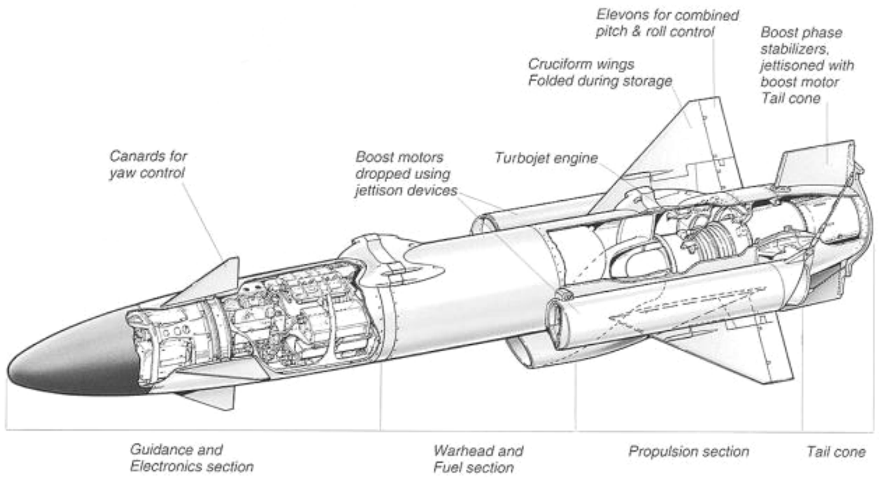

Elevons for combined pitch & roll control
Boost phase stabilizers, jettisoned with boost motor Tail cone
Cruciform wings Folded during storage
Canards for yaw control
Boost motors dropped using jettison devices
Turbojet engine
Guidance and Electronics section
Warhead and Fuel section
Propulsion section
Tail cone

Figure 3. RBS 15 mk3 Missile (From :[9])

A. CYCLOSTATIONARY SPECTRAL ANALYSIS (CSA)

1. Cyclostationary Definition

Cyclostationary spectral analysis (CSA) is based on modeling the signal as a cyclostationary process rather than a stationary process. A signal is cyclostationary of order n, if and only if, one can find some nth order nonlinear transformation of the signal that will generate finite-strength additive sine wave components that result from spectral lines [1].

CSA is a valuable tool in LPI analysis because of its ability to show the user, the modulation type and the parameters for many LPI signal types. Bandwidth (*B*), center frequency (f_c), code rate (R_c), and modulation period (t_m) are four parameters of the signal of interest that can be extracted. Another benefit of CSA is that it is able to perform well in signals with added noise. This thesis will show in a later chapter that signals with signal-to-noise ratios up to -6db of noise can be analyzed and all parameters can still be extracted. CSA performs well on the following types of signals

- Binary Phase shift keying (BPSK);
- Frequency Modulation Continuous Waveform (FMCW);
- Frank Code – (Polyphase Code).

This thesis will show, by example, how the operator using CSA processing can extract parameters for FMCW and Frank code signals. One drawback of the CSA is that it is not able to extract any time-dependence information. Time information is lost during the transformation into the frequency-cycle frequency domain. For example, the CSA cannot determine a frequency-hopping signal's order of frequencies. Time information is justification of the need for a secondary set of time-frequency algorithms such as Wigner-Ville, Choi-Williams, or Quadrature Mirror Filter Bank techniques, as indicated in Figure 1.

2. Time-Smoothing FFT Accumulation Method (FAM)

One efficient method of implementing a hardware computation of the cyclostationary spectrum is the time- smoothing FFT accumulation method. Equation (1.1) from [1] is the mathematical representation of Figure 4. The CSA can be written as

$$S_{X^{N'}}^{\gamma}(n,k) = \frac{1}{N}\sum_{n=0}^{N-1}\left[\frac{1}{N'}X_{N'}(n,k+\frac{\gamma}{2})X_{N'}^{*}(n,k-\frac{\gamma}{2})\right]$$
$$where \qquad (1.1)$$
$$X_{N'}(n,k) \triangleq \sum_{n=0}^{N'-1} w(n)x(n)e^{-(j2\pi kn)/N'}$$

k is the frequency (discrete);

N is the total number of discrete samples in the observation;

N' is the number of points in the discrete (sliding) FFT;

$w(n)$ is the windowing function, (a Hamming window is used);

$x(n)$ is the sampled complex-valued signal;

$X_{N'}^{*}$ is the complex conjugate of $X_{N'}$;

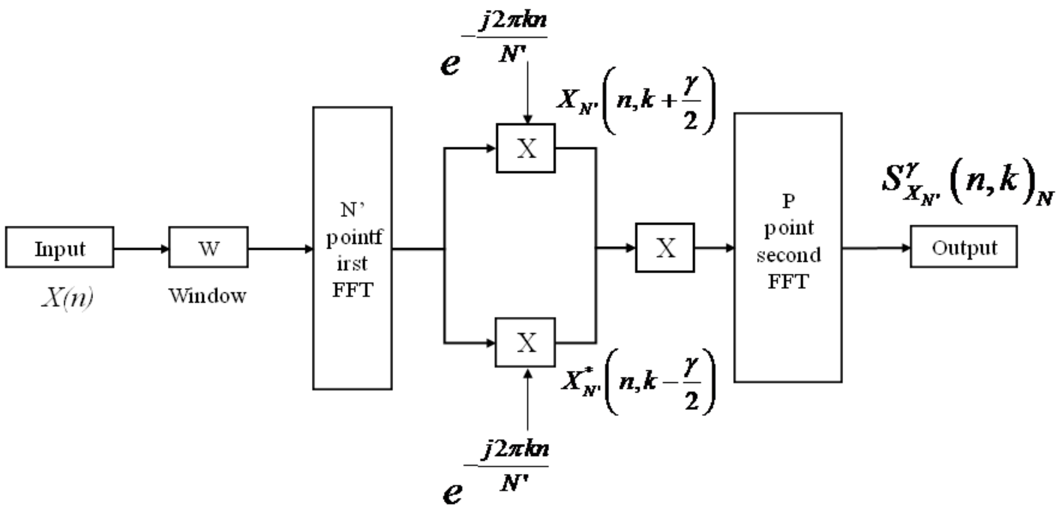
γ is the cycle frequency (discrete).

Note this form shows the CSA as a spectral correlation.

3. Data Path

The time-smoothing FFT accumulation method, is shown in Figure 4, and is implemented in [2] utilizing the SRC-6 and is the starting point for this thesis. The data path as described in [1] has three stages known as:

- Computation of complex demodulates;
    - Data tapering (Hamming Window)
    - Sliding N' point Fourier transform
    - Baseband frequency translation
- Computation of product sequences;
- Smoothing of product sequences (P point FFT)



$e^{-\frac{j2\pi kn}{N'}}$
$X_{N'}\left(n,k+\frac{\gamma}{2}\right)$
X
$S^{\gamma}_{X_{N'}}(n,k)_N$
P point second FFT
N' pointf irst FFT
Input
$X(n)$
W
Window
X
Output
X
$X^*_{N'}\left(n,k-\frac{\gamma}{2}\right)$
$e^{-\frac{j2\pi kn}{N'}}$

Figure 4. Data Path (From: [1])

B. GRENANDER'S UNCERTAINTY CONDITION

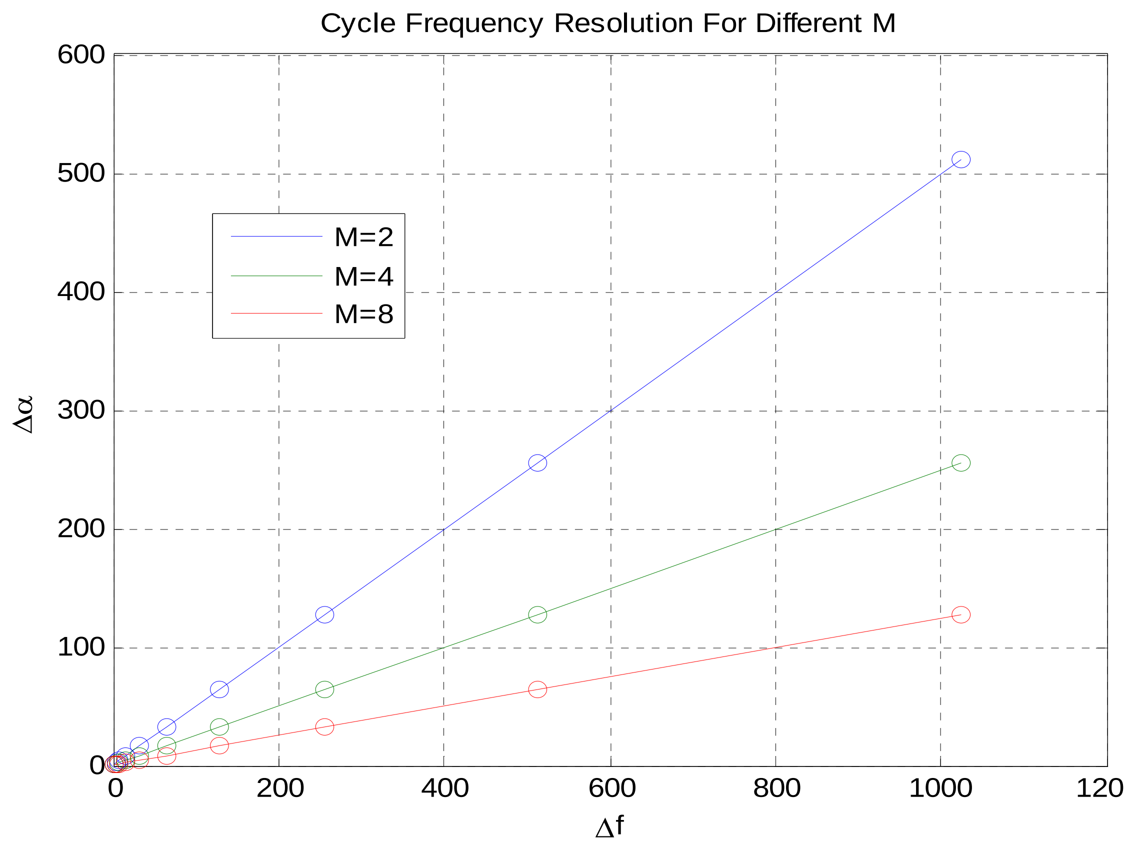
Grenander's Uncertainty Condition (*M*) is a relationship that when followed will keep the ratios of frequency and cycle frequency aligned to gives the best output. Equations (1.2) (continuous time) and (1.3) (discrete variables) show that as long as the ratio of $\Delta f / \Delta\alpha$ is relatively larger than one that Grenander's Uncertainty Condition will hold. Table 2 is a visual representation that shows the linear affect of a chosen *M* given a Δf . Since Δf is normally chosen by the operator as a function of the data acquisition process, the only remaining variable to choose is *M*. The result is the corresponding $\Delta\alpha$. This decision tool can be implemented in the software that runs a CSA.

$$M = (\Delta f / \Delta\alpha) >> 1 \quad (1.2)$$

$$M = (N / N') >> 1 \tag{1.3}$$

Table 2. Cycle Frequency Resolution for different *M* based on Δf .

$\Delta\alpha$	Frequency Resolution (Δf)									
M	2	4	8	16	32	64	128	256	512	1024
2	1	2	4	8	16	32	64	128	256	512
4	.5	1	2	4	8	16	32	64	128	256
8	.25	0.5	1	2	4	8	16	32	64	128



1. Effects of *M* on FMCW and Frank Signals

To help show how a chosen *M* affects the ability to extract important information from the signal of interest, experiments were conducted using MATLAB software from [1]. Two signals were examined without noise to show how a given *M* affects the detect ability of desired parameters. This experiment was repeated for several frequency resolutions. Studying frequency vs. cycle frequency plots, one learns that there are usually four main masses on any given plot. Approximately one on each of the main axis (0°, 90°,180°, 270°). Observe Figure 5 for a general example. For clarity and consistency, each of the plots (Figures 5 through 29 excluding Figure 18) represents a close-up view of the mass at 0°. Looking at the 90° view (or 270°) also yields accurate information about the center frequency of the signal. The views at 0° and 180° are twice the center frequency. Bandwidth is another parameter that is extractable from these graphs and is determined by measuring the strongest portion of the data. As resolution is improved, (smaller numbers are best) the estimates from the plots become more accurate. The exact parameters of the signal under investigation are known; implying that data comparison and extraction should be easy. In real life, this can be troublesome because the exact signal parameters may not be known.

The code rate is one parameter that is unique to Cyclostationary Algorithms. The code rate (R_c) gives way to other parameters based upon signal of interest. For FMCW signals $R_c = 1/2t_m$. For Frank code signals $R_c = 1/(N_c t_b)$ where N_c is the number of subcodes and t_b is the subcode period. The next two sections will review several plots to help the reader become familiar with the affects of Δf, R_c, f_c, and B given M for a selected frequency resolution (Δf).

a. FMCW

FMCW signals have three directly measurable parameters when viewing the cycle frequency vs frequency plots. For a given FMCW signal, the observer should be able to extract modulation bandwidth (ΔF), center frequency (f_c), and code rate (R_c). The first signal under investigation has the following parameters, as created by the LPIT toolbox found in [1]:

Table 3. FMCW Parameters

Signal Name	F_1_7_250_20_s.mat
Signal Type	FMCW
Center Frequency (f_c)	1 kHz
Sampling Frequency (f_s)	7 kHz
Modulation Bandwidth (ΔF)	250 Hz
Modulation Period (t_m)	20 ms
Noise Added	None- Signal only

Figures 5 through 17 show that for this FMCW signal, a minimum of Δf = 64 Hz and M = 8 should be used to measure all desirable parameters. This combination also sets N to 1024 or greater. The examples in this thesis suggest that an N of 1024 or greater yields the best resolution combinations.

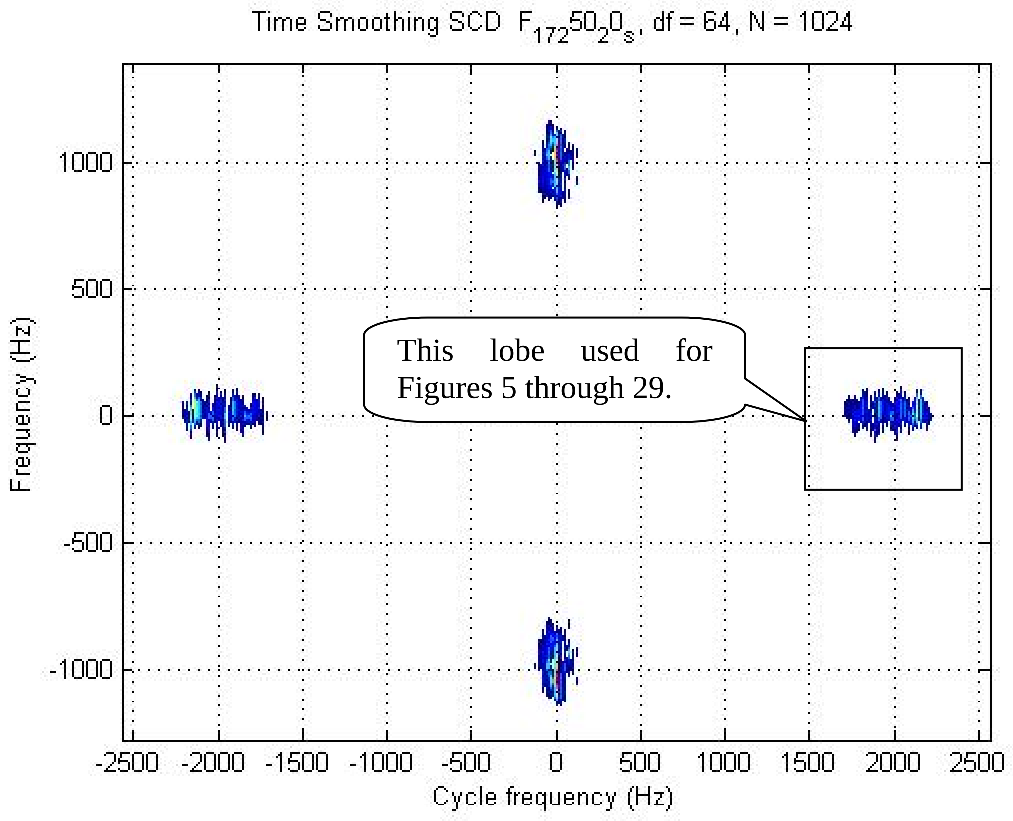


Figure 5. FMCW signal default view. Shows four lobes

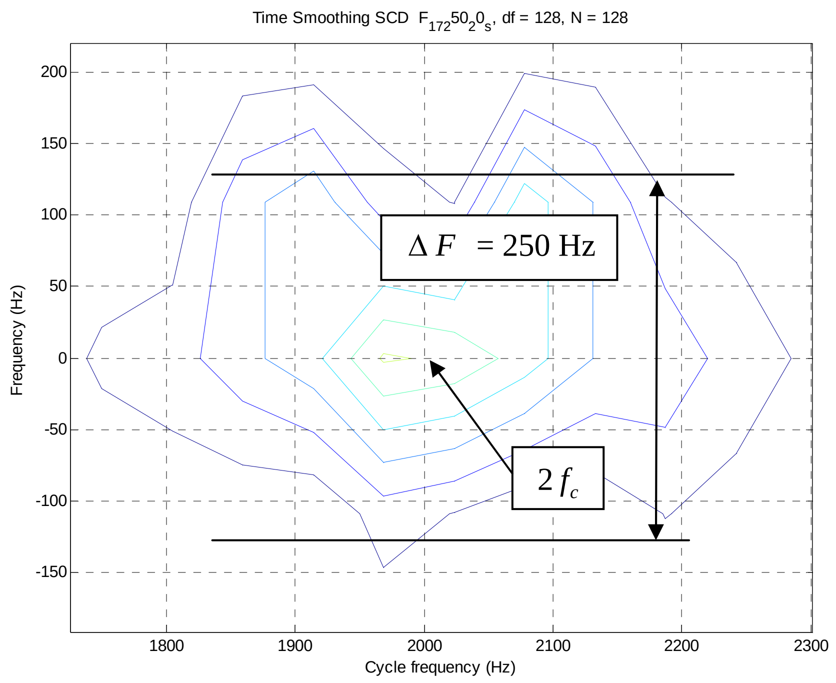


Figure 6. Δf = 128 M=2. Unable to determine R_c

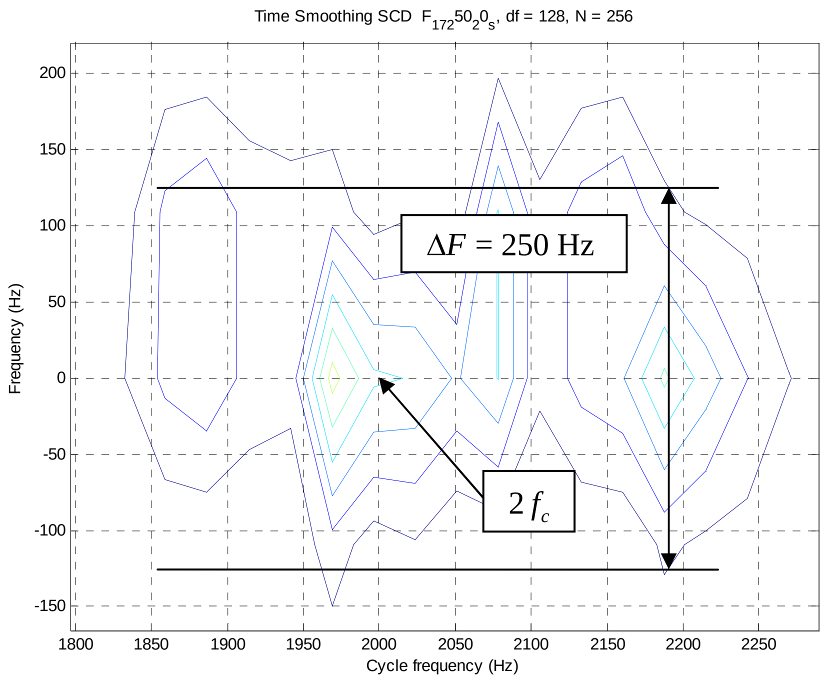


Figure 7. Δf= 128 M=4. Unable to determine R_c

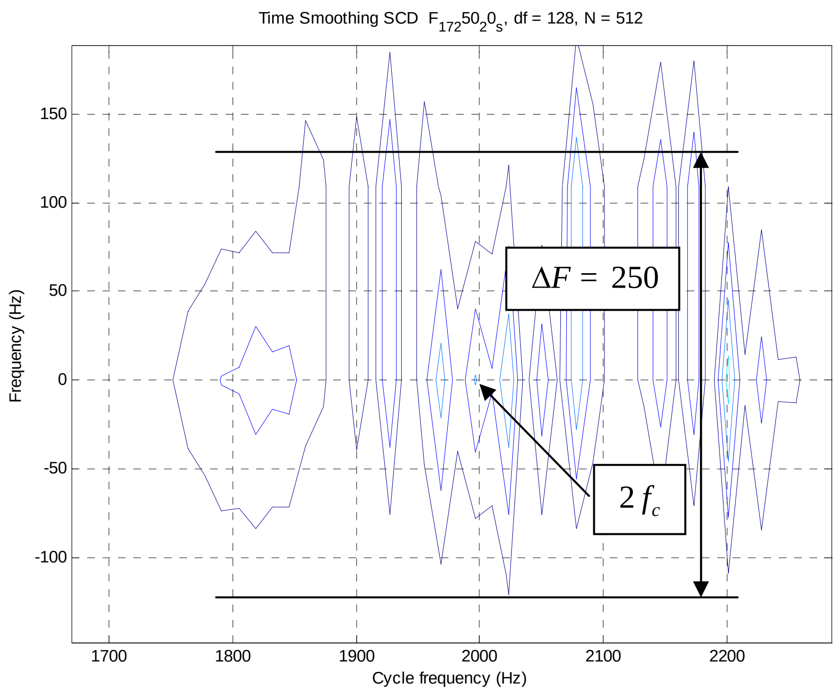


Figure 8. Δf = 128 M = 8 Unable to measure R_c

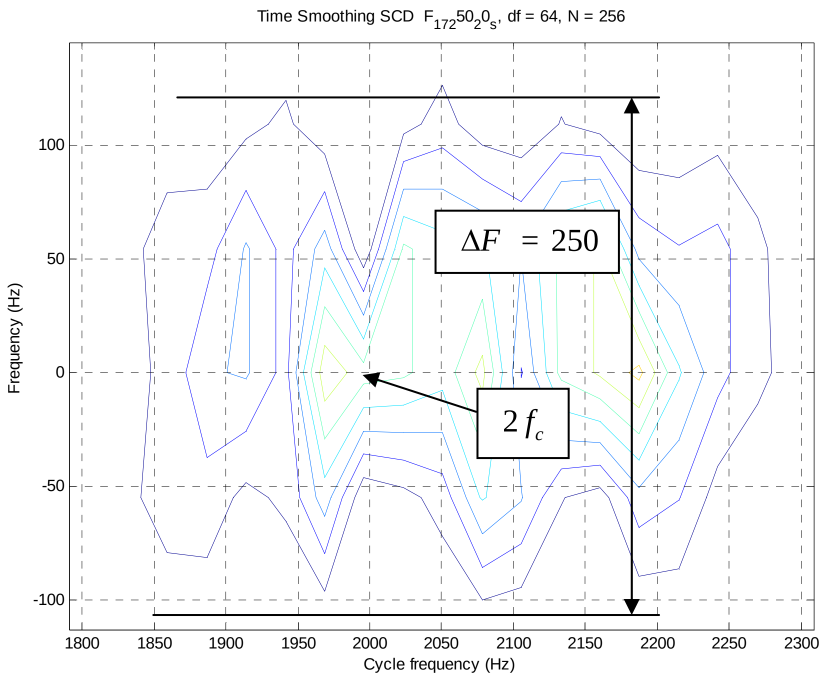


Figure 9. Δf = 64 M = 2 Unable to measure R_c

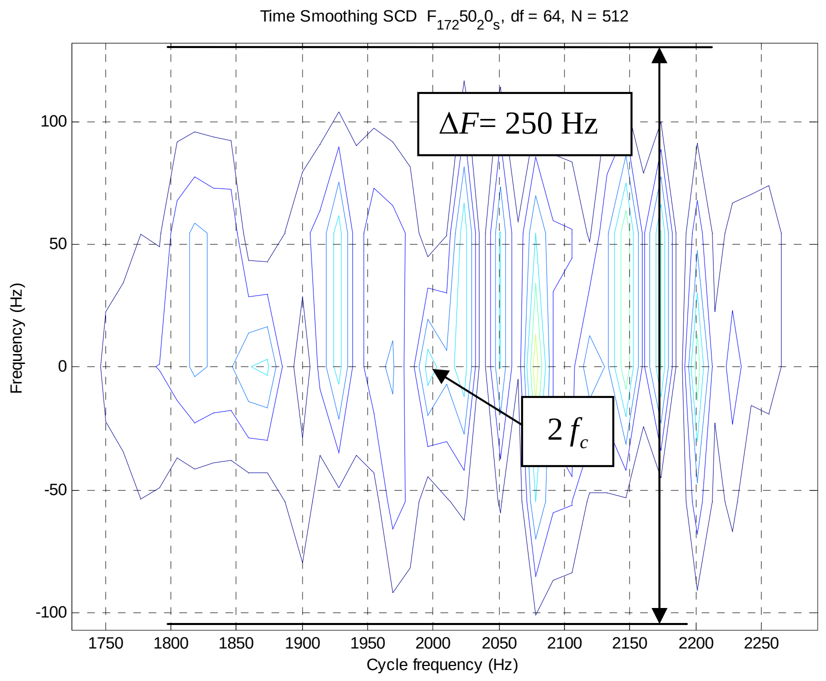
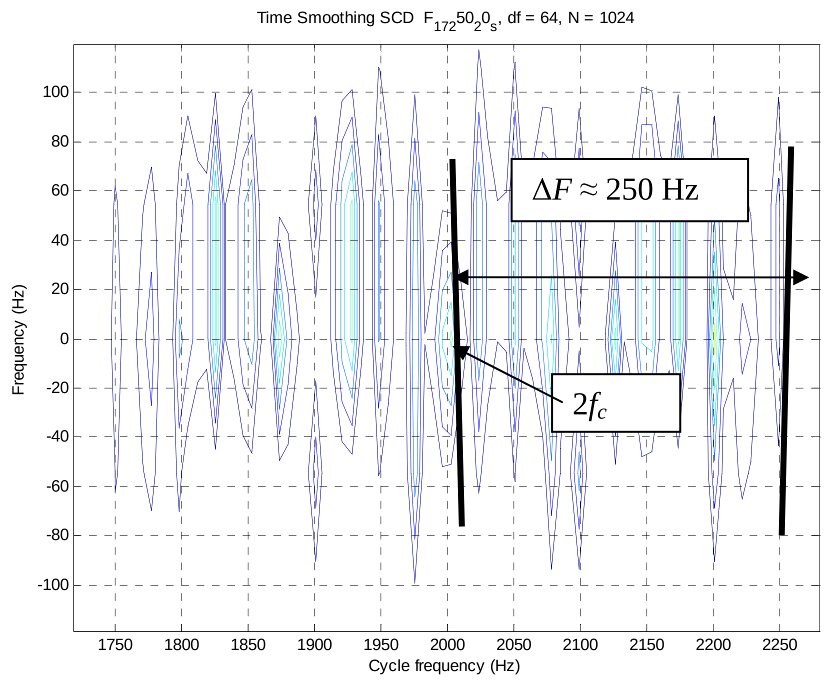


Figure 10. $\Delta f = 64$ $M = 4$. Almost able extract $R_c = 25$ Hz consistently



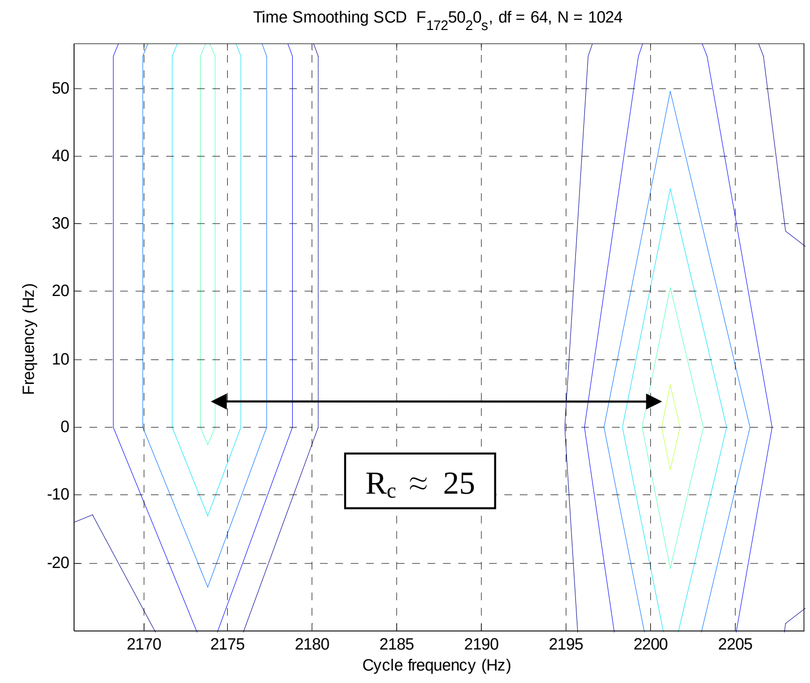
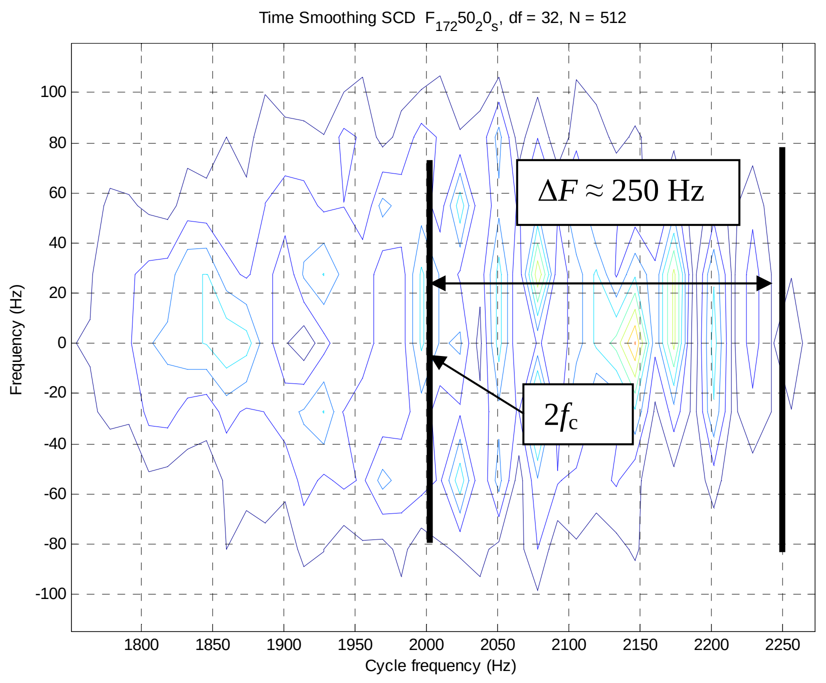


Figure 11. $\Delta f = 64$ $M = 8$. Able to measure all parameters



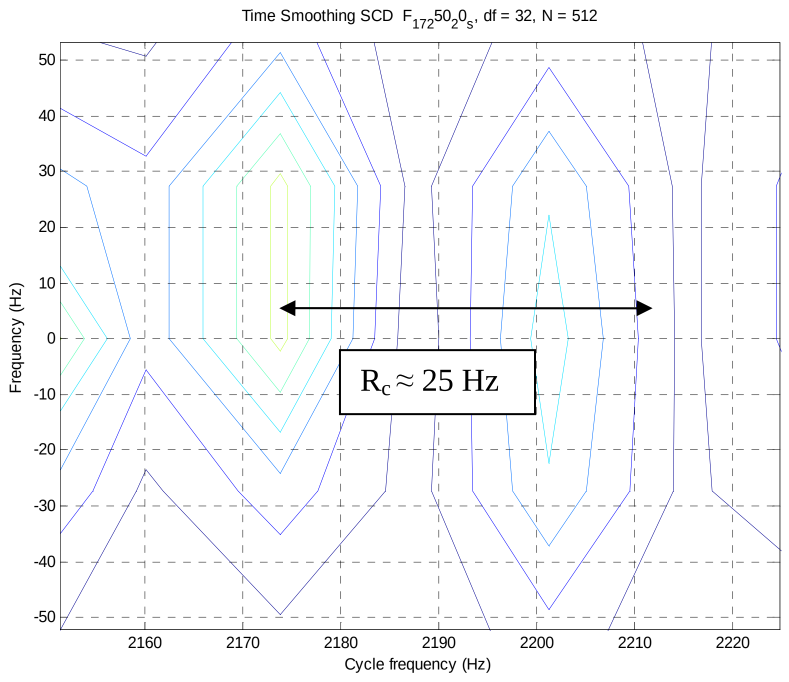
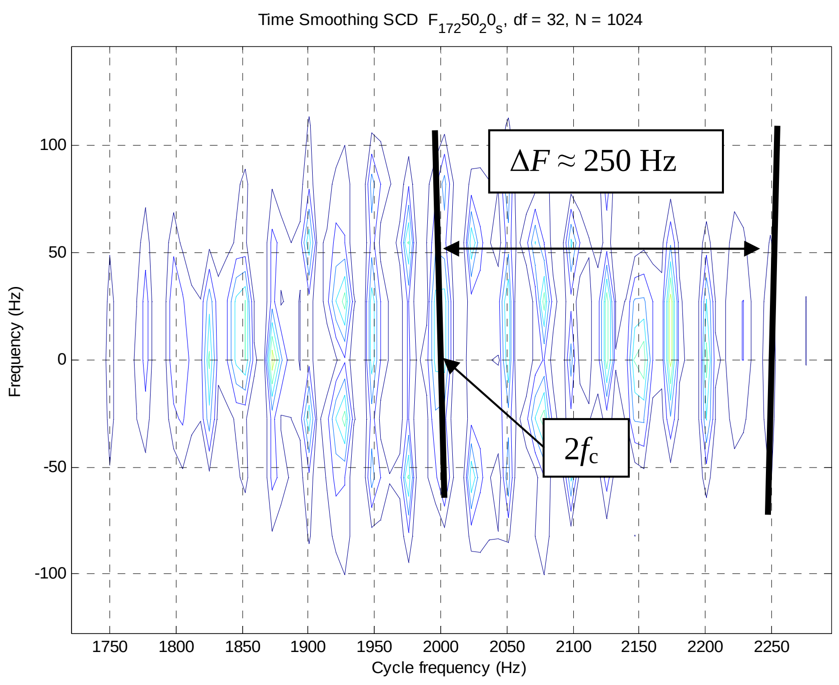


Figure 12. $\Delta f = 32$ $M = 2$. Able to see all parameters, but on right side only



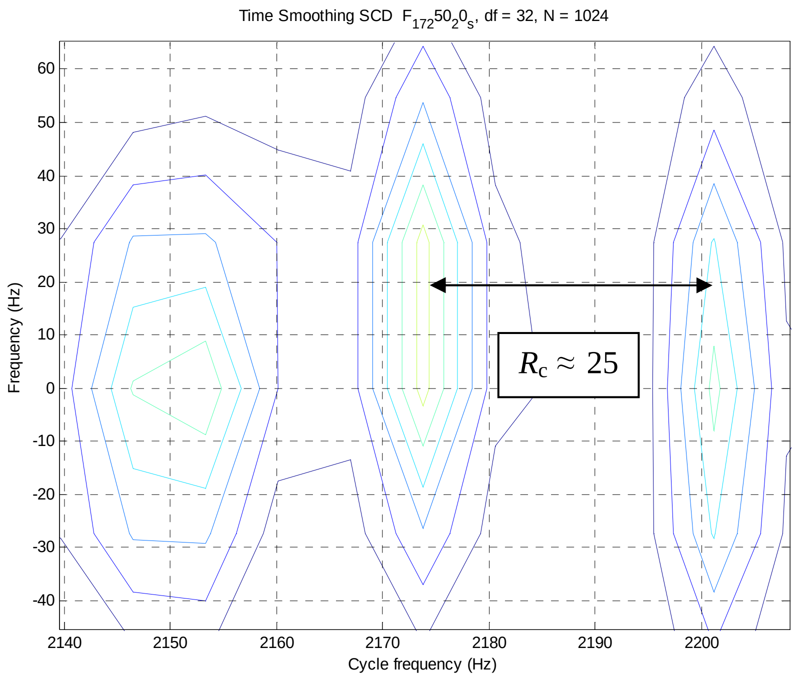
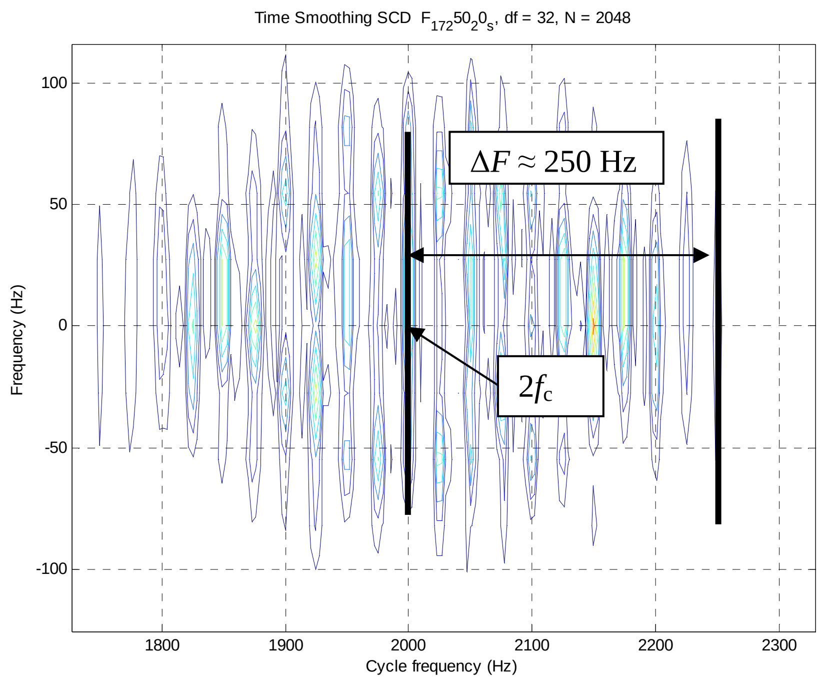


Figure 13. $\Delta f = 32$ $M = 4$. Able to extract all parameters



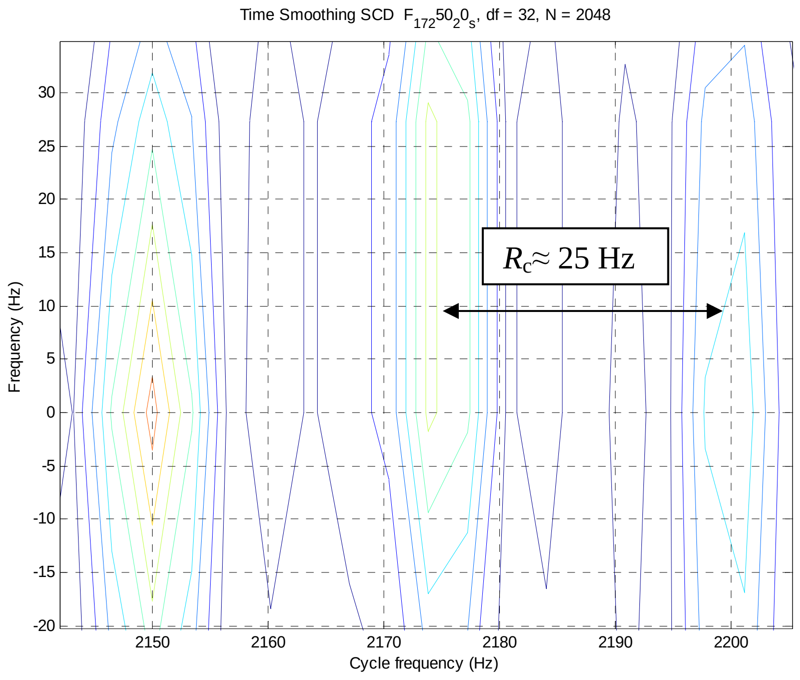
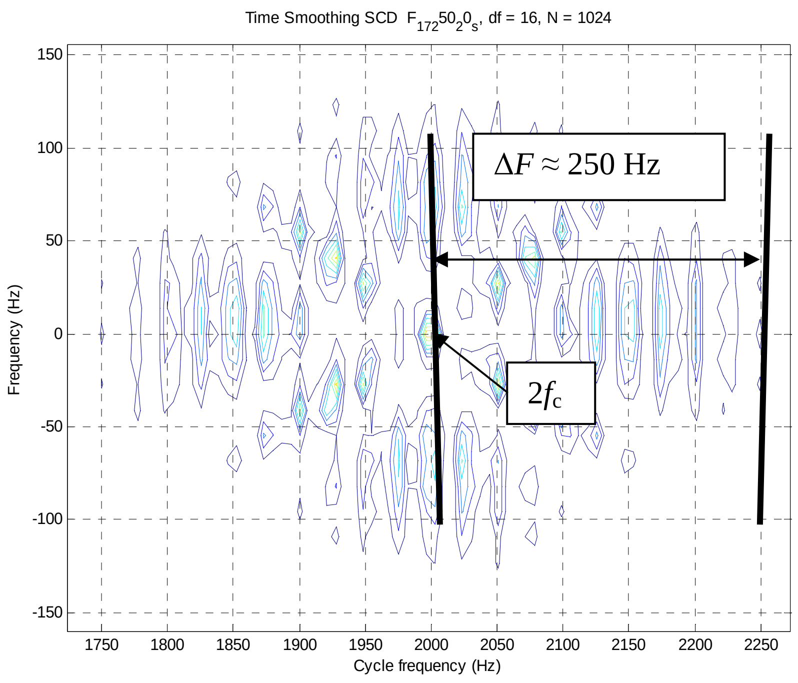


Figure 14. $\Delta f = 32$ $M = 8$ all parameters extractable



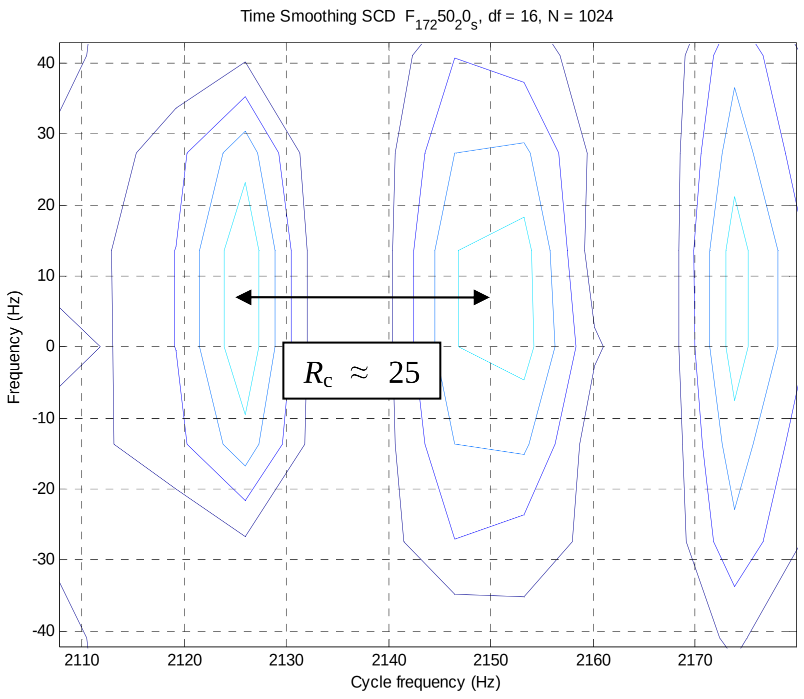
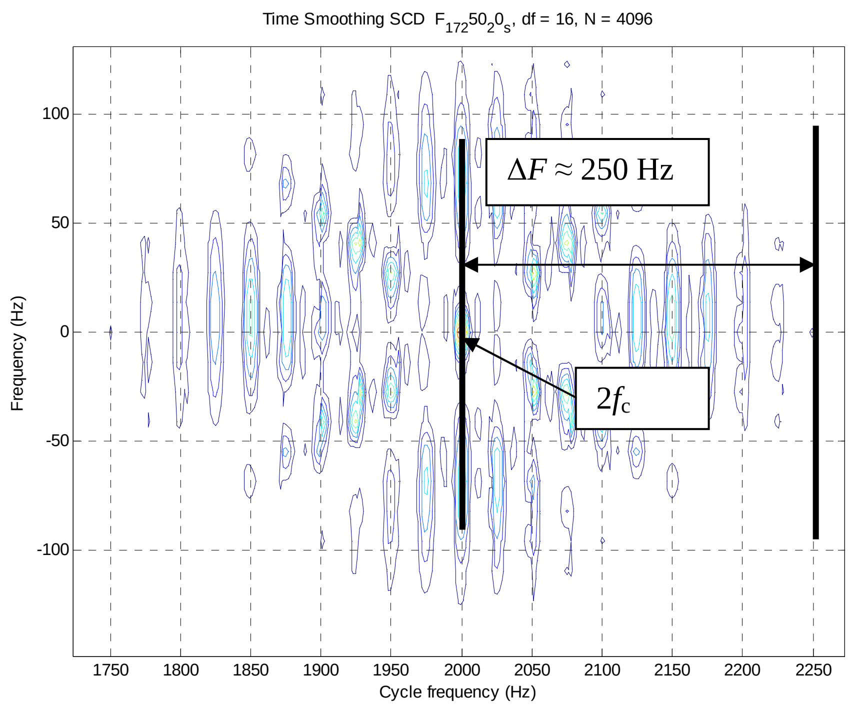


Figure 15. $\Delta f = 16$ $M = 2$- all parameters extractable



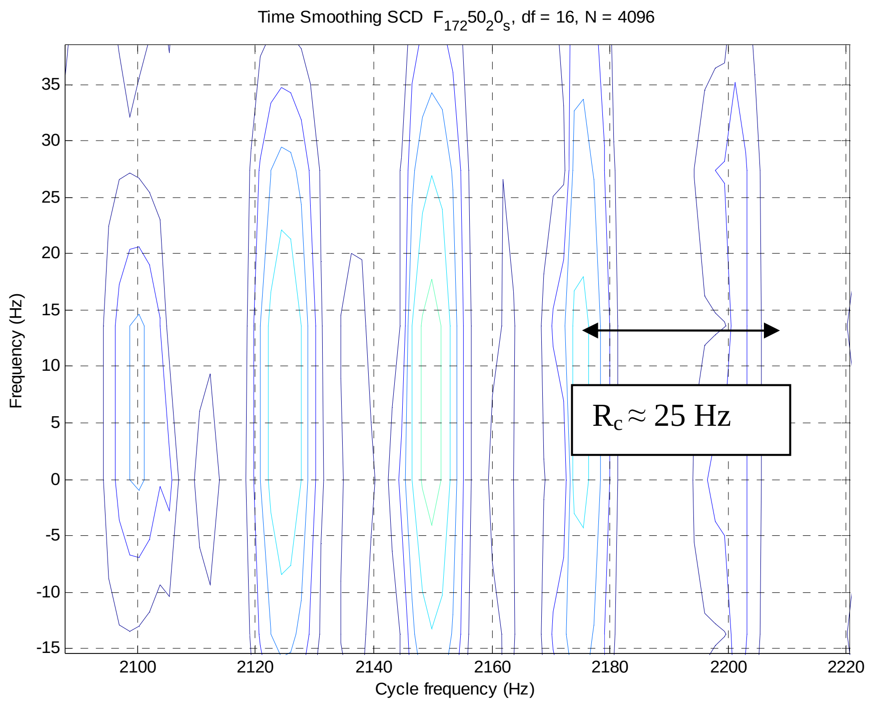


Figure 16. $\Delta f = 16$ $M = 8$. Able to extract parameters

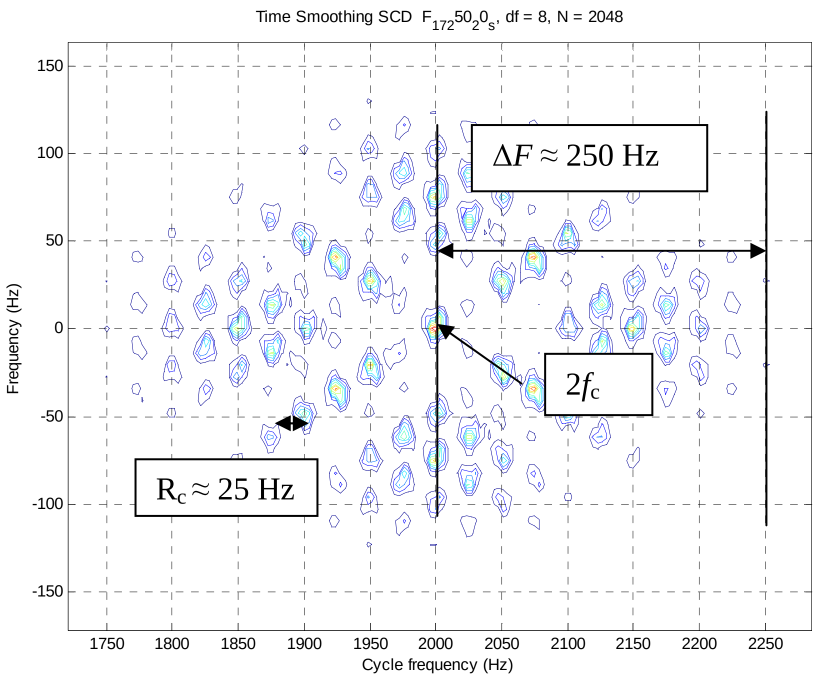


Figure 17. $\Delta f = 8$ $M = 2$. Able to extract all parameters

b. Frank Code Signal

The Frank code signal will be analyzed in the same fashion as the FMCW was by choosing Δf and M and attempting to detect the various parameters. For the Frank Code, the observer should be able to extract bandwidth (B), center frequency (f_c), and code rate (R_c). The number of subcodes (N_c) can be calculated because R_c and B are measurable as described by the equation:

$$N_c = \frac{B}{R_c} \tag{1.4}$$

Table 4. Frank Parameters

Signal Name	FR_1_7_8_1_s.mat
Signal Type	Frank
Center Frequency (f_c)	1 kHz
Sampling Frequency (f_s)	7 kHz
Number of Phase Codes (N_c)	8^2= 64
Cycles per phase (cpp)	1
Noise Added	None- Signal only

Figures 18 through 29 show that for this Frank signal, a minimum of Δf = 64 Hz and M = 8 should be used to measure all desirable parameters. This combination also sets N to 1024 or greater. This example signal also suggests that an N of 1024 or greater yields the best resolution combinations.

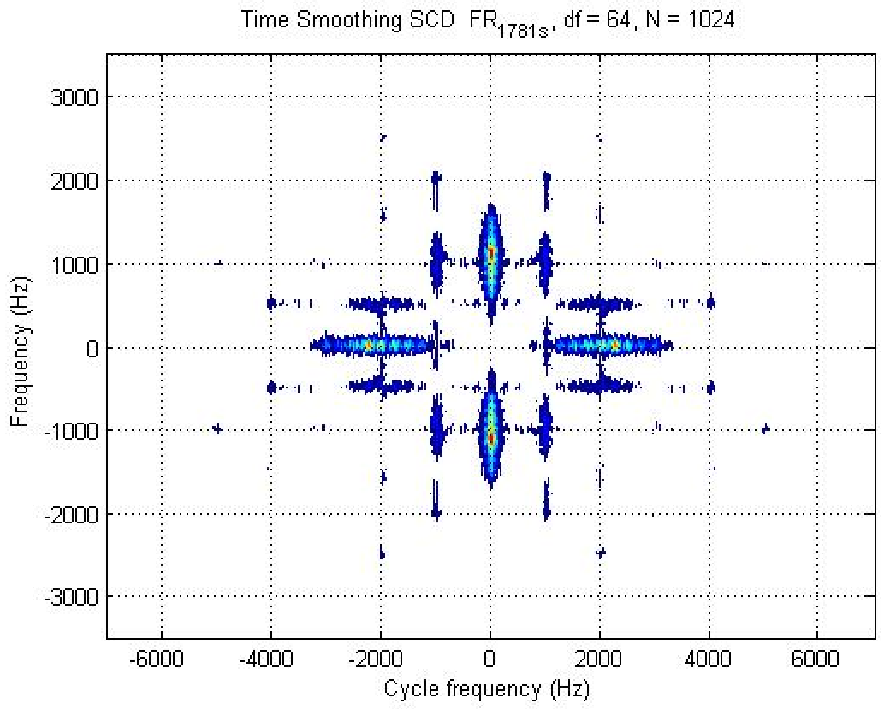


Figure 18. Frank code entire signal showing all 4 lobes.

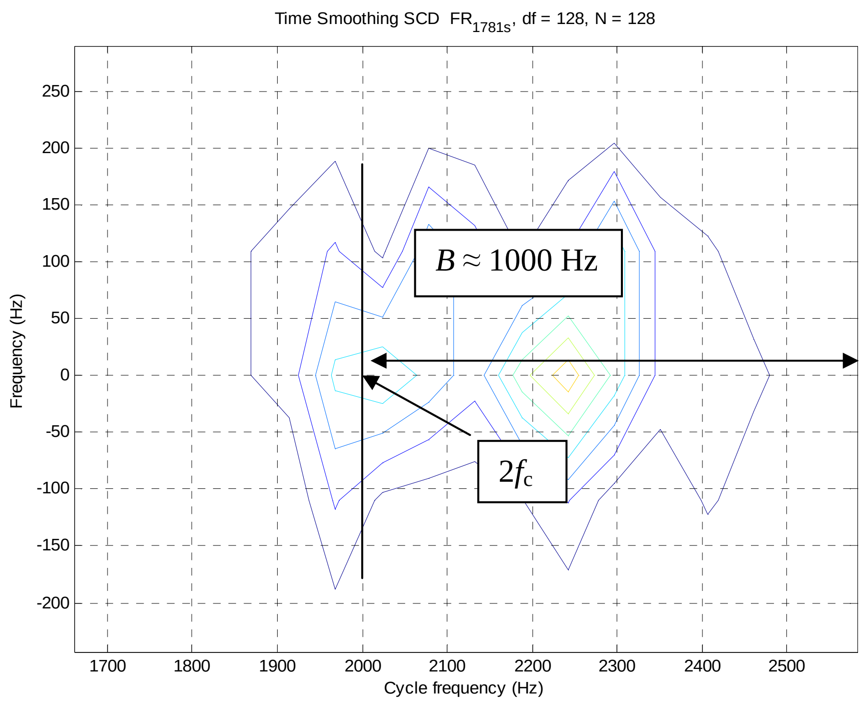


Figure 19. $\Delta f = 128$ $M = 2$. No confidence in determining B or F_c

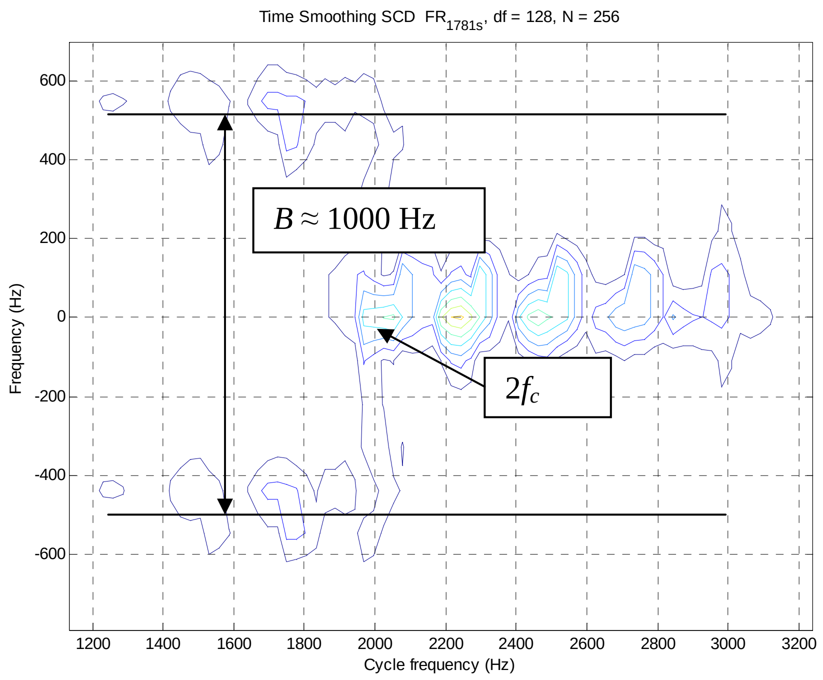


Figure 20. $\Delta f = 128$, $M = 4$ Not able to measure R_c

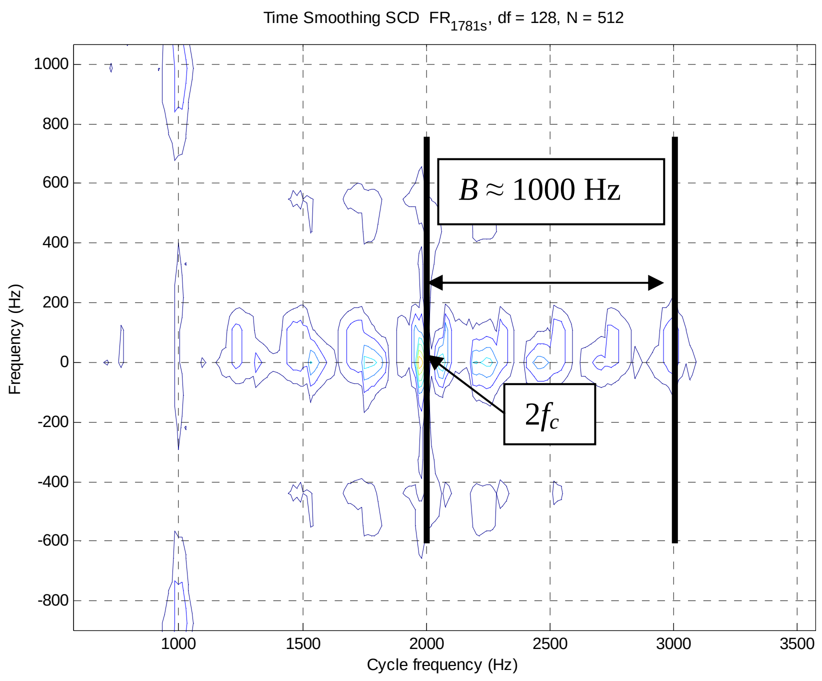


Figure 21. Δf = 128, M = 8. Not able to measure R_c

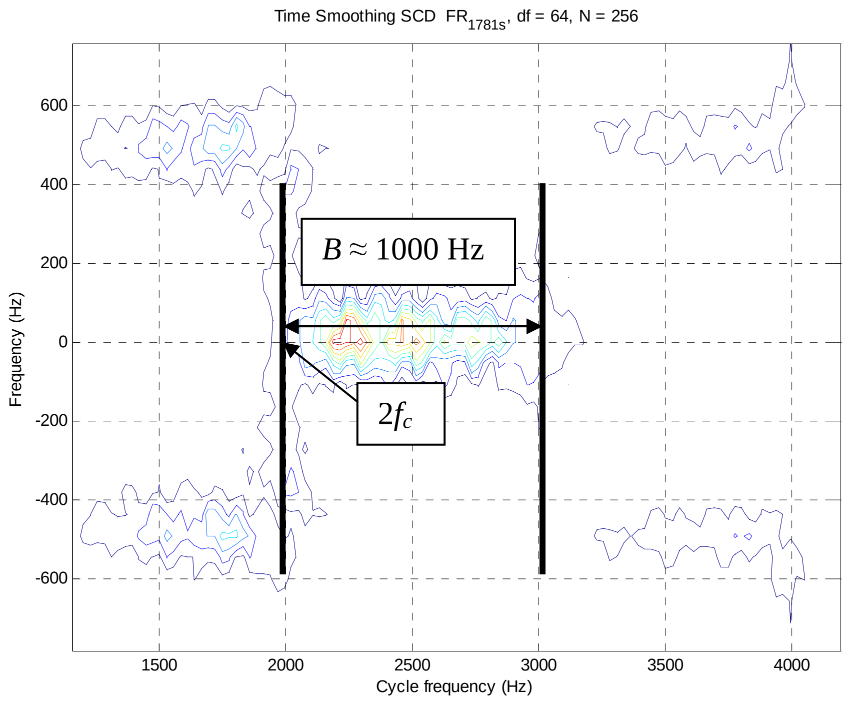


Figure 22. Δf = 64, M = 2. Unable to measure R_c

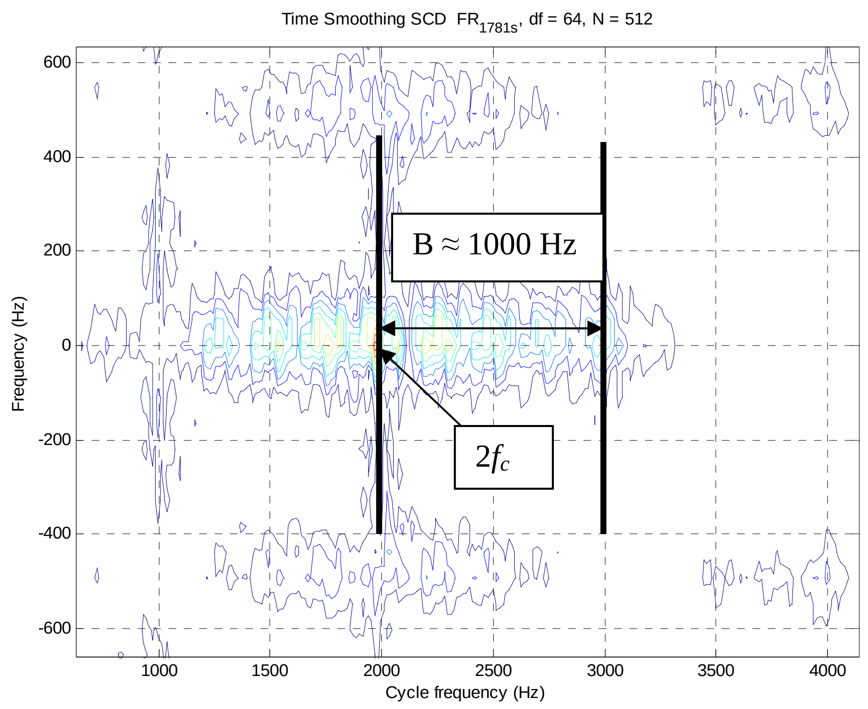
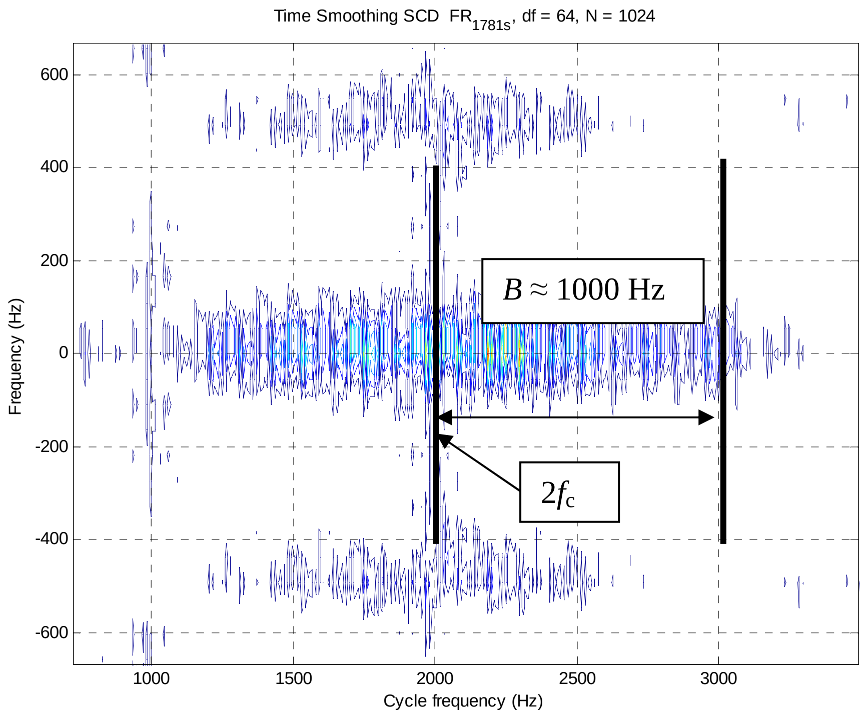


Figure 23. $\Delta f = 64$, $M = 4$. Unable to extract R_c information.



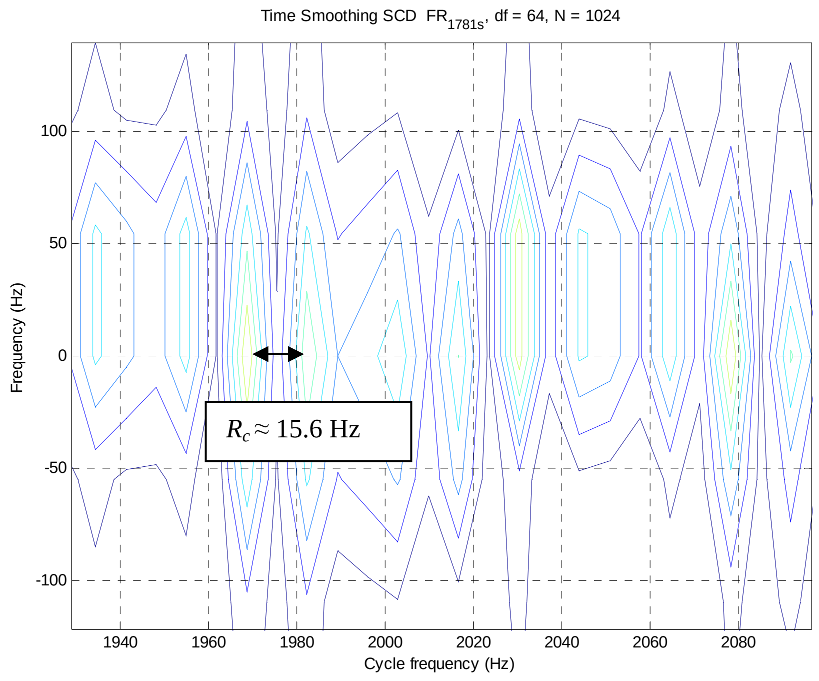
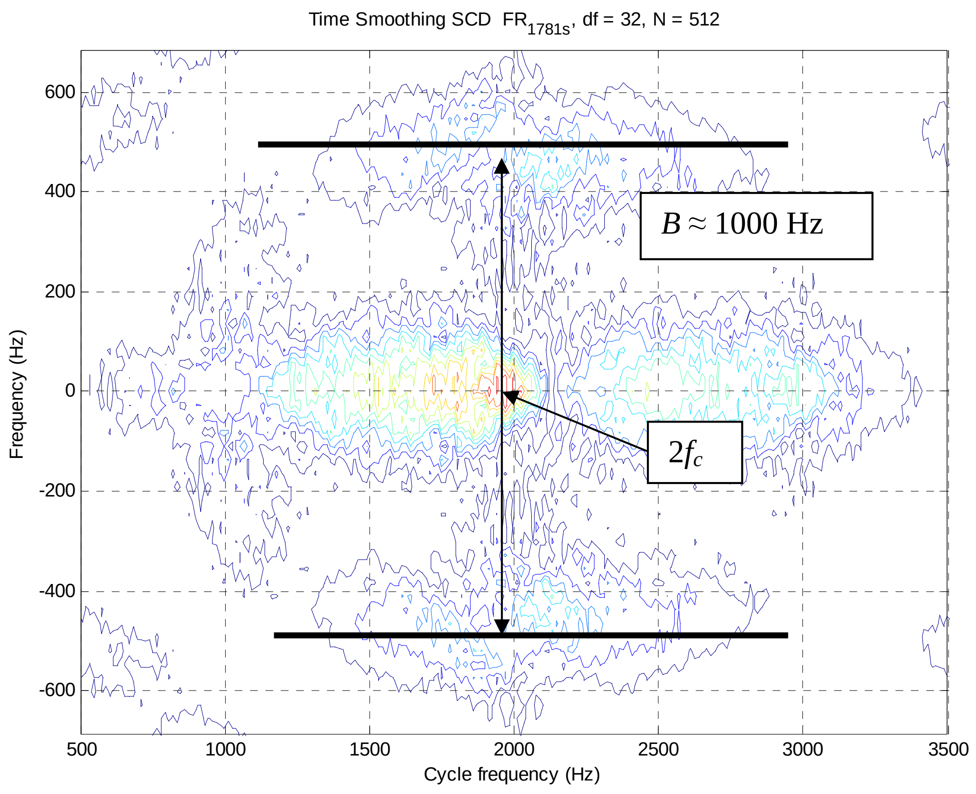


Figure 24. Δf = 64, M = 8. R_c is measurable in several places.



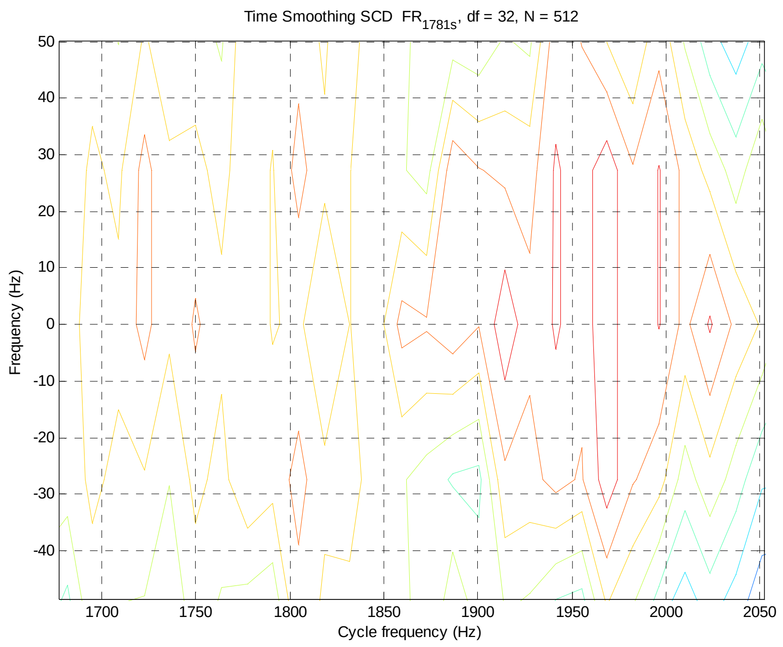
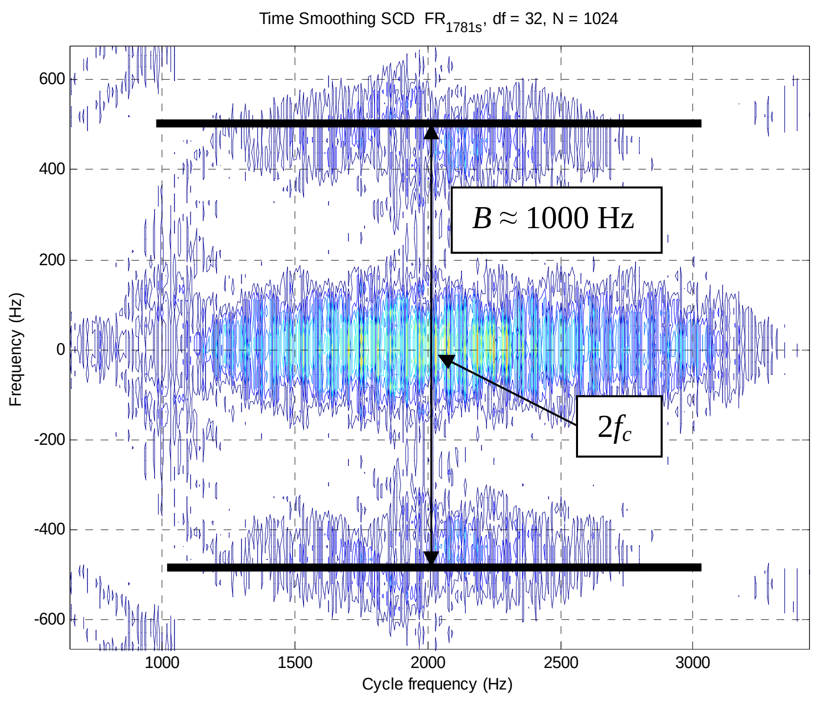


Figure 25. $\Delta f = 32$, $M = 2$.

Upper plot shows how B is also measurable on the Frequency axis. Not able to measure R_c consistently



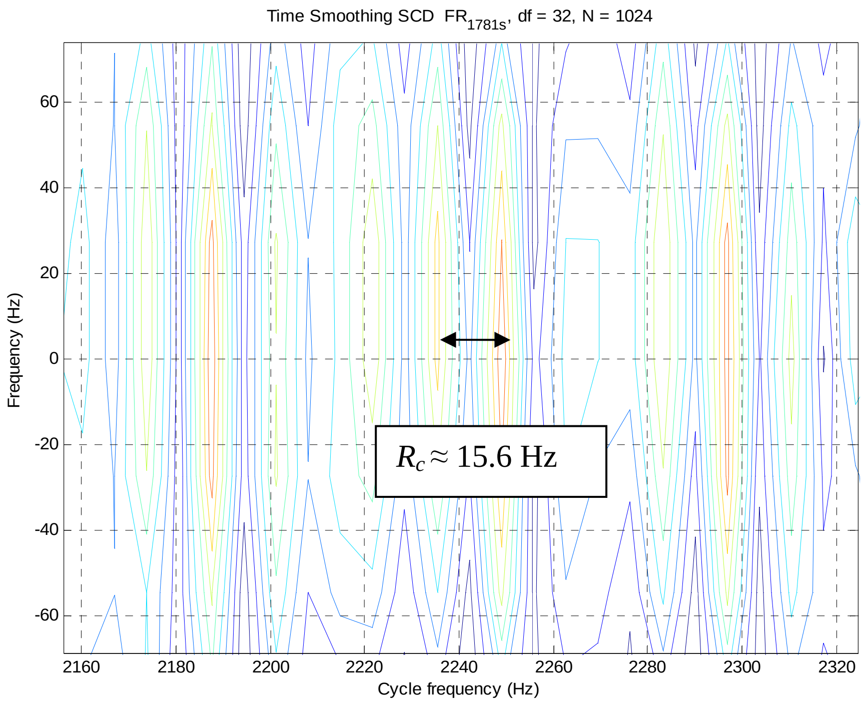
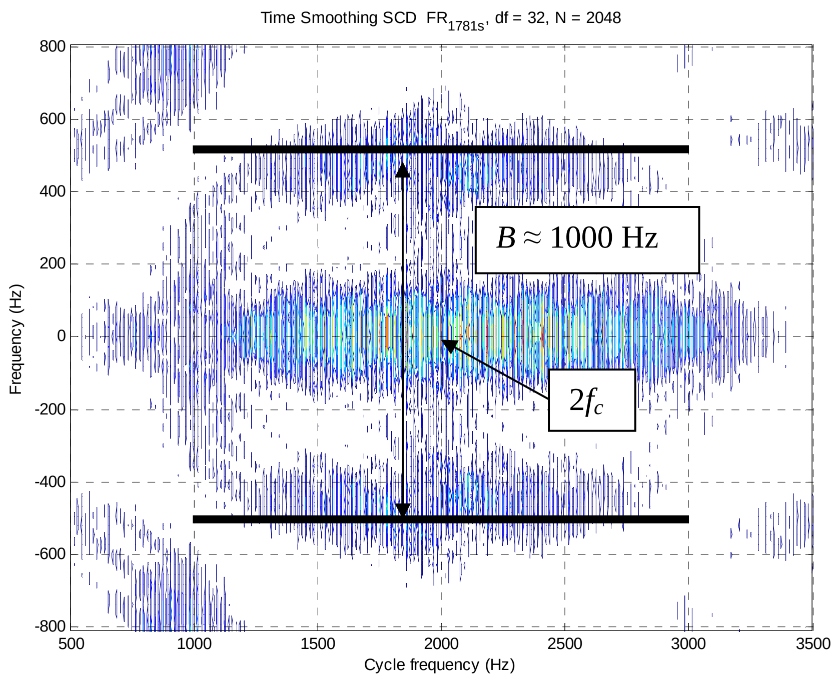


Figure 26. Δf = 32, M = 4 Able to extract all parameters



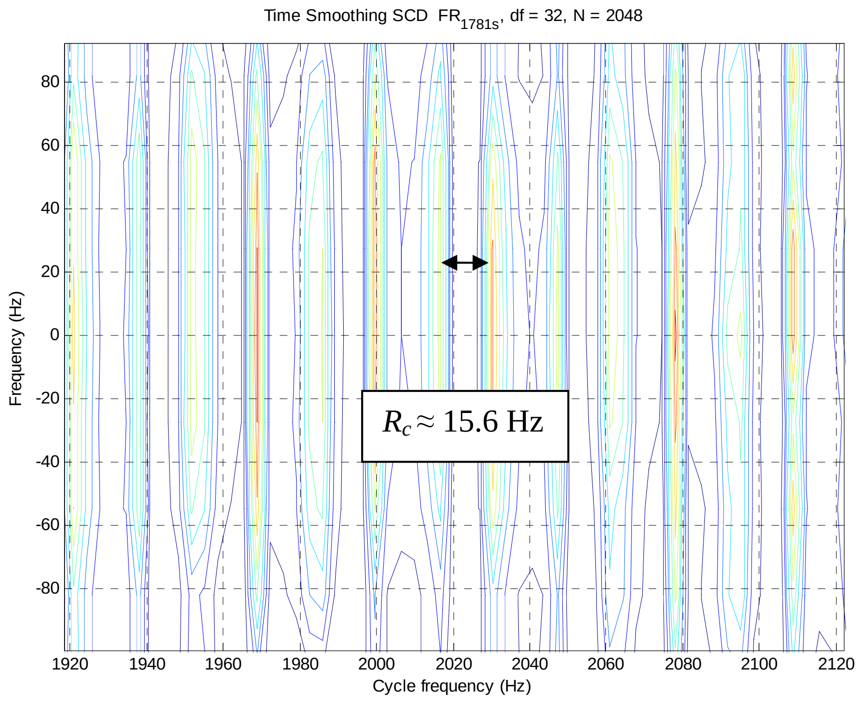
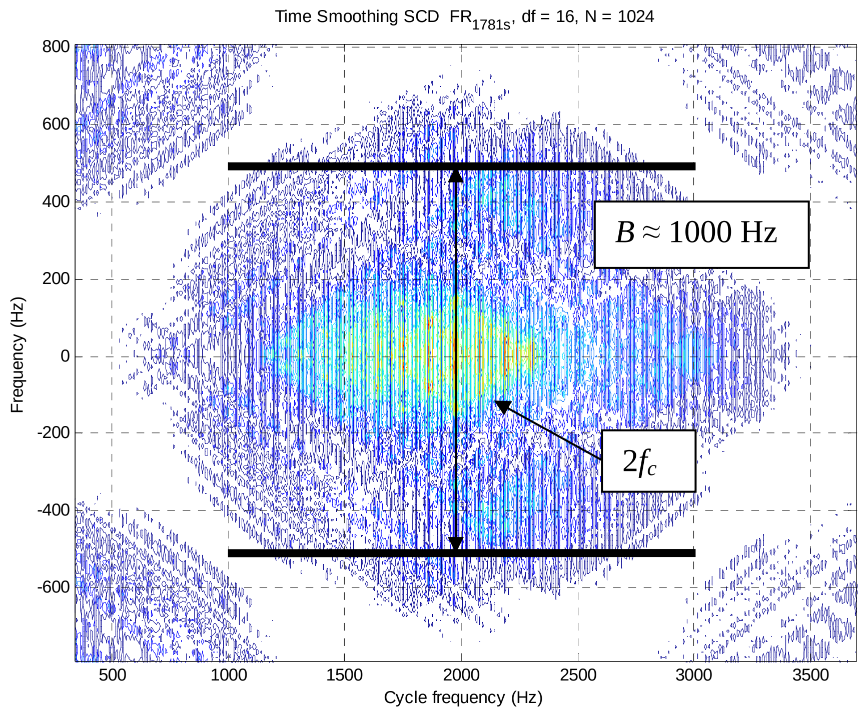


Figure 27. $\Delta f = 32$, $M = 8$. Able to extract all parameters



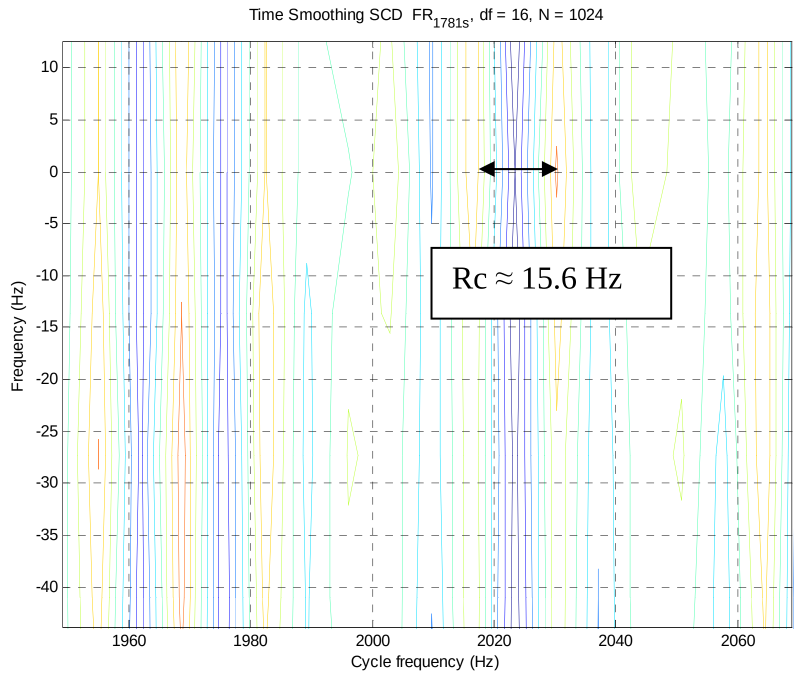
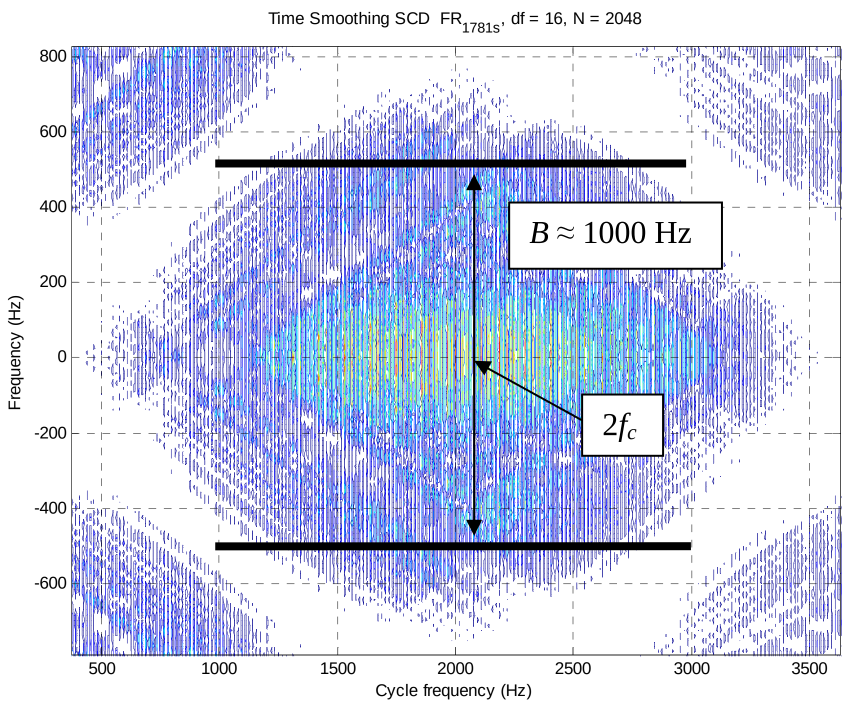


Figure 28. $\Delta f = 16$, $M = 2$. Able to extract all parameters



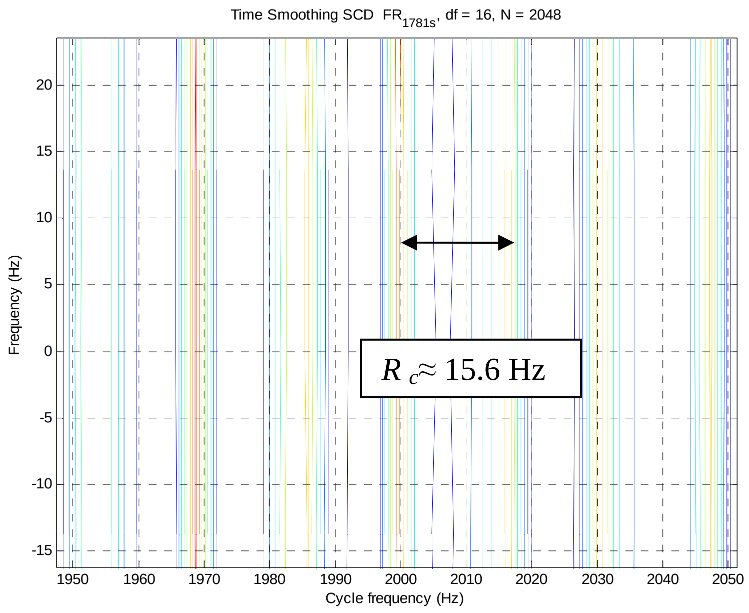


Figure 29. $\Delta f = 16$, $M = 4$. Able to extract all parameters

2. Hardware Decisions

From observing the previous plots for the Frank and FMCW signals, hardware implementations of the CA FAM suggest to use Δf and M combination that yields an N of 1024 or higher. N is described as:

$$N = PL \tag{1.5}$$

$$\Delta\alpha = \frac{\Delta f}{M} \tag{1.6}$$

$$L = pow2(nextpow2(f_s / \Delta f) / 4) \tag{1.7}$$

$$P = pow2(nextpow2(f_s / \Delta\alpha / L)) \tag{1.8}$$

Using a value of N that is greater than 1024 yields frequency cycle-frequency plots yielding resolutions that are sufficient to extract all parameters that will allow an operator to identify and rebuild the signal of interest. Frequency resolution was set by hardware when the sampling frequency of 7000 Hz was chosen. This sampling frequency is used for consistency from the previous thesis [2] and the assumptions made for the MATLAB code found in [1]. Cycle-frequency resolution is set as a result of Δf and M. This suggests that a minimum of 1024 points are needed to get sufficient resolution in both cycle-frequency and frequency. This allows the engineer to pick values for the two FFTs based upon size and speed limitations of the targeted hardware. This procedure should also be repeated for other modulation types to help validate the usefulness of this technique with the proposed minimum settings for Δf and M.

C. NOISE EFFECTS

The Cyclostationary FAM algorithm is noise resistant. Several examples below help validate this claim. Each signal is generated using LPIT from [1]. Figures 28 through 31 are FMCW signals with noise levels of 0, -3, -6, and -10 dB respectfully. Careful observation by a trained operator suggests that all of the parameters are extractable. Looking at the -6 and -10 dB plots, it is difficult to extract but still able to be done.

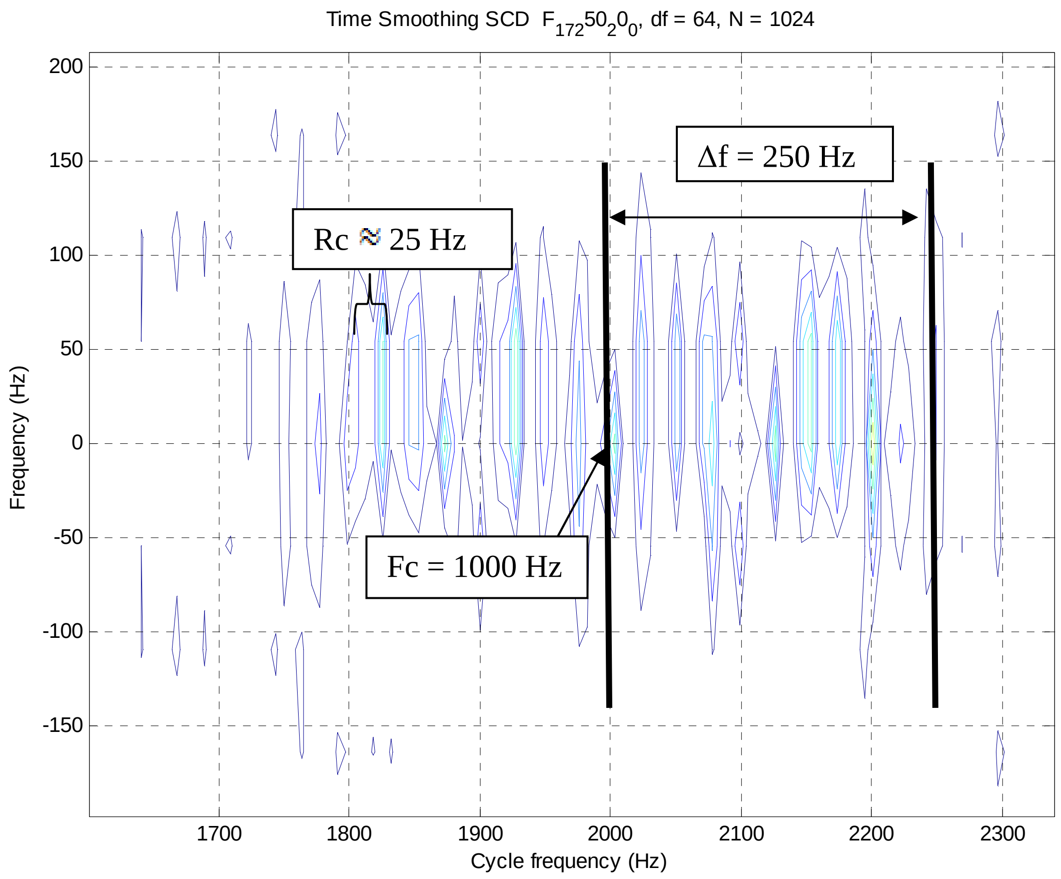


Figure 30. Signal with 0 db noise. All parameters extracted

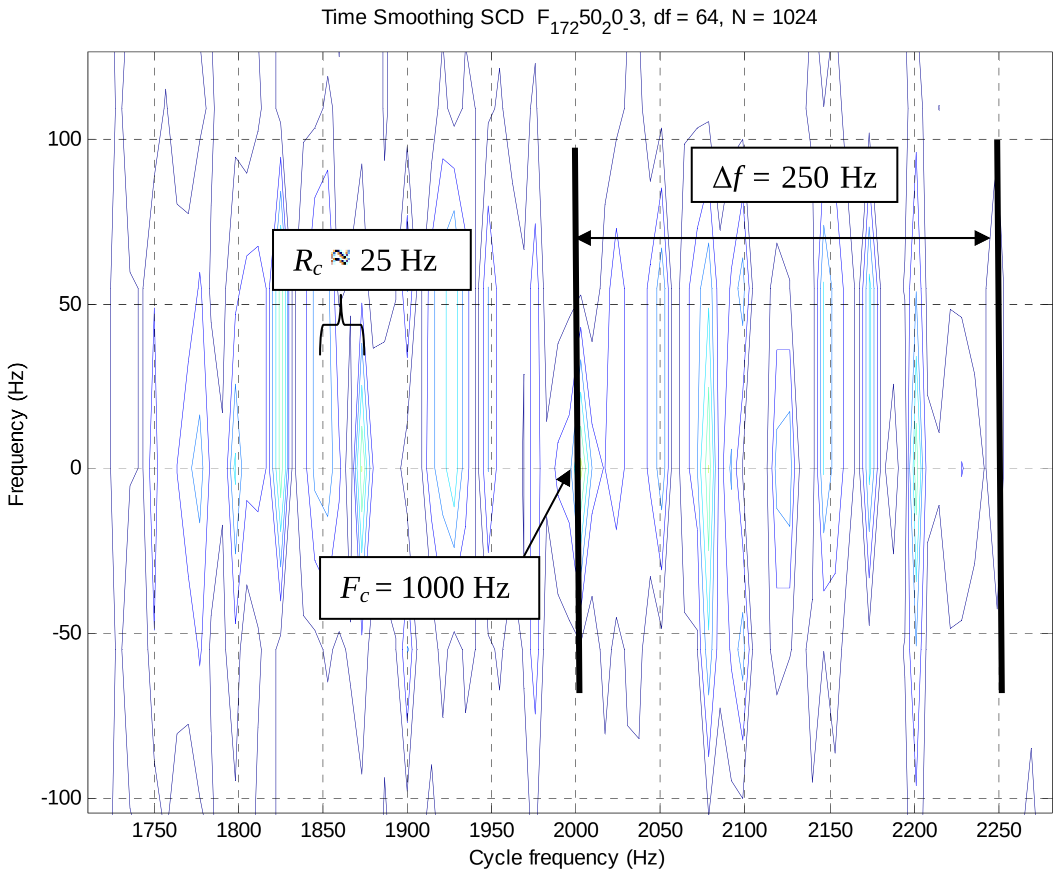


Figure 31. Signal -3 db noise. All parameters extracted.

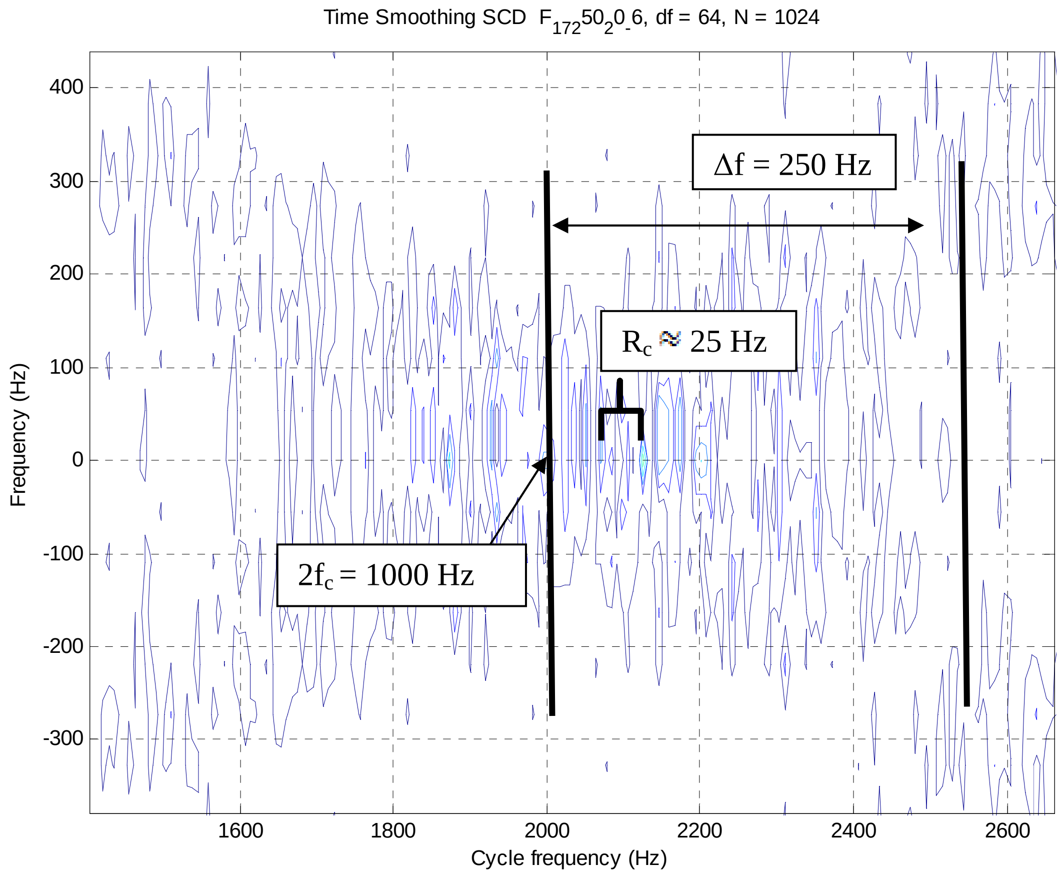


Figure 32. Signal with -6dB noise- R_c is harder to extract.

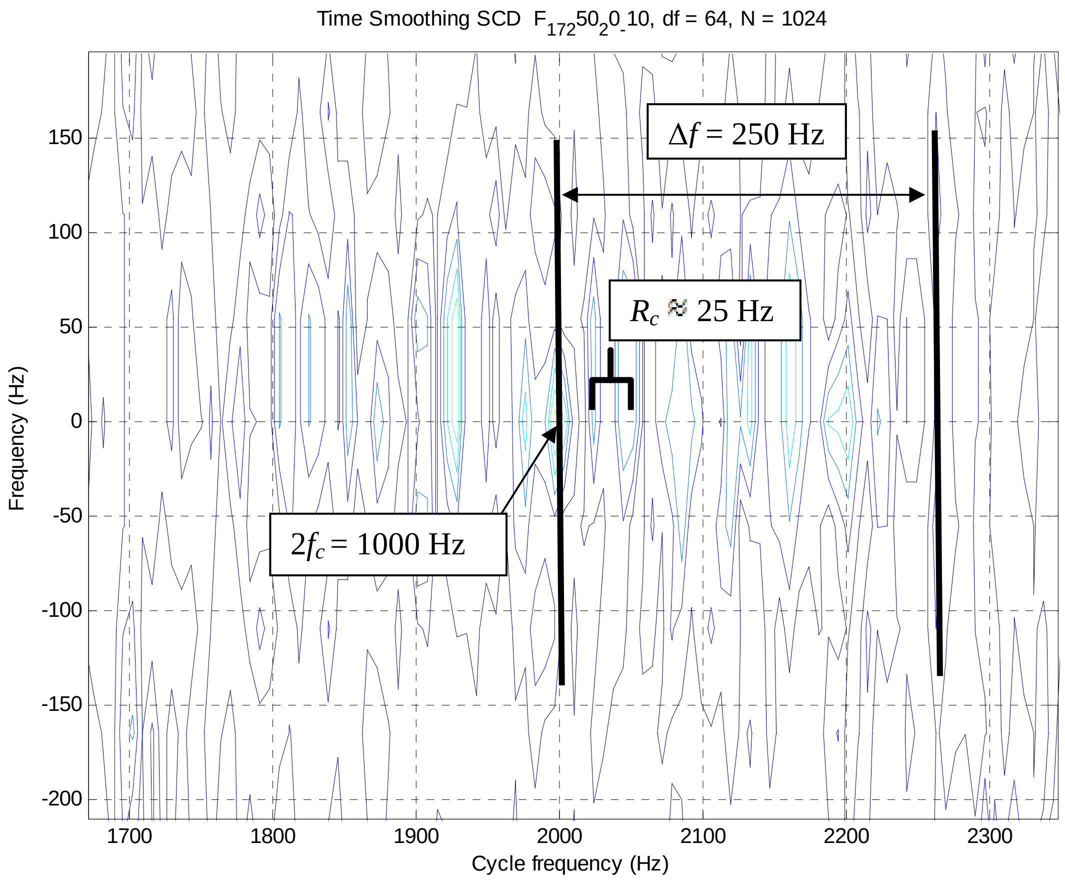


Figure 33. Signal with -10dB noise. R_C is difficult to extract accurately

D. SUMMARY

The use of LPI signals and radar is on the rise throughout the world. Knowing this, Naval Postgraduate School is working to develop an autonomous system to reduce the decision time of classification and extraction of parameters from signals of interest.

The time smoothing FFT accumulation method was chosen for this thesis based upon previous work. As part of the optimization of the previous algorithm developed in [2], this research discovered that Grenander's Uncertainty condition helps decide minimum hardware needed to created the desired resolutions in both the frequency and cycle-frequency domains. The minimum product of the two FFT sizes is 1024.

Validation of the robustness of the cyclostationary algorithm was accomplished by examining how noise affects a signal of interest using MATLAB code. Signals of interest as shown to be noise tolerant visually up to -6 dB of noise.

Examining the signals of interest was an important step during the initial research stage to allow important decisions to be made with respect to hardware choices. Software decisions are also influenced by knowledge of the cyclostationary FAM algorithm.

III. RECONFIGURABLE COMPUTING

A. SRC-6

The SRC-6 is a reconfigurable computer designed by SRC Computers, LLC in Colorado Springs, Colorado. SRC developed a hardware and software architecture they call the IMPLICIT+EXPLICIT Architecture™, which fully integrates Dense Logic Device (DLD) technology and reconfigurable Direct Execution Logic (DEL). Systems built with this architecture execute user code, written in high-level languages such as C or FORTRAN, on a mixture of tightly coupled implicitly and explicitly controlled processors [10]. Figure 34 describes the high-level points of the implicit and explicit device differences. The implicit controlled device is a DLD that operates at a higher clock rate and is made up of fixed logic. In the case of the NPS SRC, it is an Intel microprocessor (μP). The Explicit controlled device is DEL hardware that runs at a lower clock rate (100Mhz) and is a reconfigurable FPGA.

The DLD for the SRC is the Intel IA-32 line of microprocessors.   The SNAP™ interface bridges the μP to the MAP by a 1400 MB/sec interface. This interface is one of the bottle necks of the system.

Implicit Controlled Device	Explicit Controlled Device
Dense logic device (DLD) Higher clock rates Typically fixed logic µP	Direct execution logic (DEL) Lower clock rates Typically reconfigurable FPGA

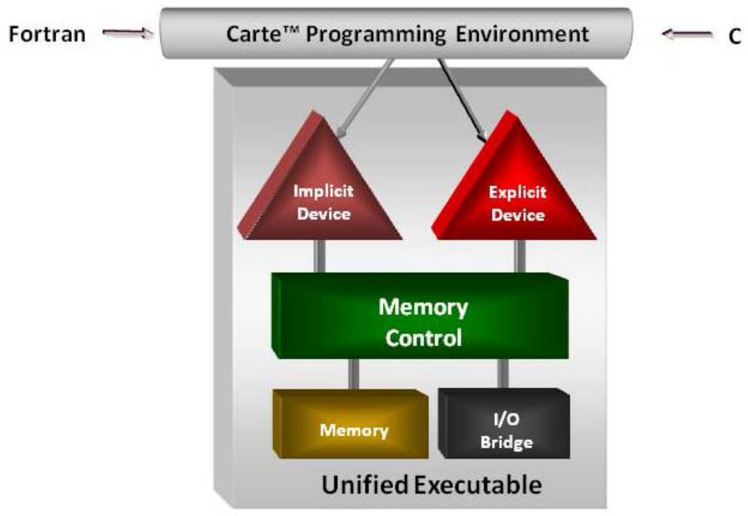


Figure 34. Implicit + Explicit Architecture (After: [10])

To aid the user in programming the SRC-6, SRC has developed the Carte™ Programming environment. The Naval Postgraduate School is currently using Carte version 2.2 as of this writing. Carte is further covered in Chapter III Section C.

At the heart of this system is the MAP®, which is the SRC high-performance Direct Execution Logic processor. MAP, an acronym for Multiple Adaptive Processor, houses two user logic areas, associated memory, and a control processor. There are several ports to the 'outside world' allowing for interconnection with other MAPs. Figure 35 shows the various data rates and components of a typical MAP. Appendix A compares the different variants of MAPs compatible with the SRC-6 architecture.

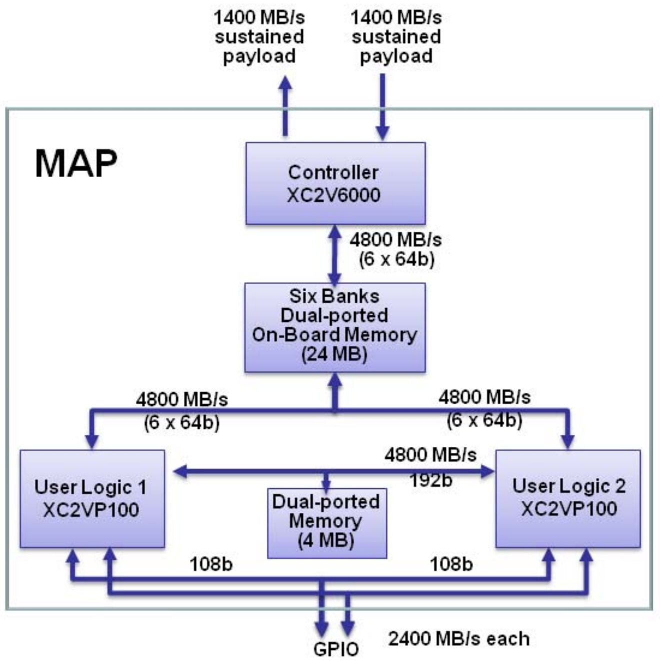


Figure 35. MAP E Overview

The SRC-6 was chosen because it is expandable, flexible, and available for use at the Naval Postgraduate School. Learning to utilize the FPGA logic also has other applications to porting this project to other (newer) FPGAs. The unique SRC architecture has tremendous possibilities for a realizable, upgradable system, as shown in Figure 1, by interconnecting several MAPs and connecting radars as input peripherals and connecting monitors as output peripherals or possibly ship or aircraft early warning systems.

B. HARDWARE

With every design project, the hardware is analyzed, so that software may be optimized. This project is no different. The FPGAs within the E-Series map were chosen over the other available MAPs in the system because they have more available memory (BRAM) and logic slices. Optimally, this project would perform better given the newer F, G, or H series maps that have a faster clock, more slices and/or built in floating point hardware. Working with what is available, the software will be targeted to the E-Series Map Xilinx Virtex II pro xc2vp100 FPGAs.

1. Xilinx Virtex II pro~ xc2vp100-ff1696-5

The model of FPGA used in the E-Series MAP is the xc2vp100-ff1696-5. The size of each FPGA is 42.5mm x 42.5mm [11]. The speed grade of these particular chips is -5 (indicating a 300 Mhz PowerPC Processor Block) [12]. Quick reference information about this FPGA is found in Table 5. A visual representation of the architecture is seen in Figure 36.

Two major components make up the FPGA. The input/output blocks (IOBs) provide the interface between package pins and the internal configurable logic [12]. Internal configurable logic blocks (CLBs) are used to implement the configurable network of logic. CLBs have four major components.

- Combinatorial and Synchronous logic
- Block SelectRAM (dual-port RAM)
- Dedicated Multipliers (18-bit x 18-bit)
- Digital Clock Manager (DCM)

All the components are able to connect via the General Routing Matrix (GRM). The GRM connects each programmable element together during compilation and allows for fast reprogramming of an FPGA.

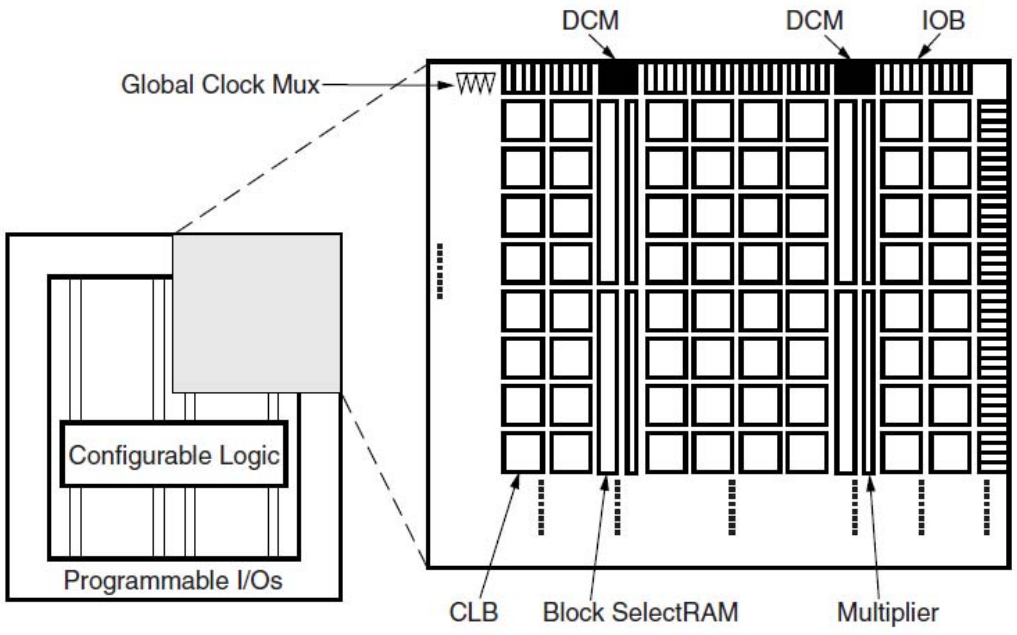


Figure 36. Virtex II Architecture Overview (From: [12])

Table 5. Virtex-II Pro XC2VP100 quick information (After: [12])

Device	Logic Cells	CLB (1=4 slices= max 128 bits		18 x 18 Bit Multipliers	Block Select RAM +	
		Slices	Max Distr RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)
XC2VP100	99,216	44,096	1,378	444	444	7,992

2. Memory

Available memory on each MAP comes in three types, On Board Memory, Block RAM, and global (common) memory. Of the memory available, OBM and BRAM are fixed sizes. Global memory may be upgraded by purchasing more memory and placing it into available slots. The hardest to use and most costly in terms of latency is global memory. Memory utilization and initialization is done by macros and is covered in a later section.

a. On Board Memory (OBM)

Memory onboard the MAP is limited to six banks of 523776 - 64 bit words. Each bank also has 512 words available to hold scalars. The banks are named A, B, C, D, E, and F. When designing arrays the size must be known at compile time [10].

b. Block RAM (BRAM)

BRAM is available to the programmer in the quantity of 444 units with each unit containing 2048bytes. BRAM allocation size will be the smallest table value that is able to contain the allocated size [10]. BRAM is fast dual ported memory.

C. SOFTWARE

Programming an SRC-6 reconfigurable computer is not for the novice with a project due this coming weekend. The system requires an understanding of several programming languages and an in-depth understanding of the hardware to take advantage of the architecture[1]. The compiling of the source files is done utilizing the SRC CARTE™ programming environment. The files of a project are the subject of this section.

Complex and custom circuitry is one advantage of designing with this unique architecture. To do so, programming is done in a hardware description language such as Verilog or VHDL. Code for custom macros and the microprocessor can also be written in C or Fortran. These custom codes should be targeted for the MAP because the programmer can take advantage of the unique capabilities of reprogrammable logic. Creating a unique combination of custom circuitry and code is where the art is in utilizing this system. No one-stop programming book exists that describes how to approach designing code for the SRC-6. The best recommendation is to experiment and gain knowledge with the system. Develop as much code as you can in the language you are most comfortable with (ANSI C or FORTRAN) when coding the bulk of the program. The compiler does a decent job converting your code into circuitry to implement either

[1] It is possible to only use one programming language to make a complete program. To take advantage of FPGAs precise circuit design should be utilized by employing Verilog or VHDL.

on the FPGA or on the microcontroller. Use Verilog to create custom macros for circuits that are difficult to program in C or can take advantage of hardware implemented on the MAP that is not on the µP.

To best utilize the FPGAs, move any code that is designable with custom hardware to a macro. As mentioned earlier, another technique developed for this thesis is to avoid costly memory transfers (back and forth transfers). Also, move the adjoining functions that can be done as part of the logical thought progression. Build an assembly line of operations that need to be performed on multiple pieces of data. This process has no benefit if you are performing operations on only a few pieces of data. After all, there is a latency penalty for the downloading and uploading of the data to an FPGA. The compiler will attempt to pipeline as much as it can. This benefit allows for an increased throughput of data. This is why it is best to perform the chosen operations on several pieces of data.

Data operations are not limited to logic gates; the pipelines can take advantage of successive mathematical operations as is done in the code of this thesis. The Fast Fourier Transform is one operation that is full of different mathematical computations. Applying Hamming Windows, down-converting, and multiplication are all operations in this thesis that also lend themselves to being pipelined. Moving them to the FPGA made sense.

Throughout the development of the code for this and other projects, the most efficient method for designing the custom codes for macros is to develop Verilog code in a program such as Xilinx ISE or Active HDL. These programs have the added benefit of testing and simulating the code. If you purchase the appropriate versions of the software, you can even see how your code should perform on your targeted system. Xilinx ISE is especially helpful because it utilizes the exact software that the SRC-6 uses for compiling the executable code.

The recommendation of this thesis is to design custom circuitry that you do not have a simple function call for on the microcontroller. In addition, any series of computations that lend themselves to a pipeline operation are great for the FPGA. To fully exercise the FPGAs and conceptualize the larger system of dedicating one MAP for

Cyclostationary Processing the entire algorithm was targeted for the two FPGAs. At the end of the project, only a fraction of the post-data processing was moved back to the microprocessor because of size restrictions.

CARTE is best thought of as a complex compiler that takes all the possible types of input files and optimizes and links them together to create an application executable. To create the application executable, the linker brings together the MAP Compiler and µProcessor '.o files,' called object files. CARTE runs on Fedora Linux.

The MAP compilation process is best visualized by Figure 37. The steps of Parsing, CFG/DFG Generation, Optimizations, HDL Generation, and Place and Route are completed prior to creating the Unified Executable. HLL source files written in C, Map Macros, Customer Macros, and the runtime library comprise the different types of files that come together during compilation. This thesis utilizes all types of files. For the beginning user, it is simpler and more efficient to maximize the use of the runtime library and Map Macros. These files are already optimized for use and provide easy interfaces for use and ample documentation for implementation. Another popular option is to import intellectual property (IP) that has been created for the targeted hardware. Xilinx has a robust library that was considered for this thesis[2].

[2] Consult Chapter V for discussion of possible use of IP for FFT implementation.

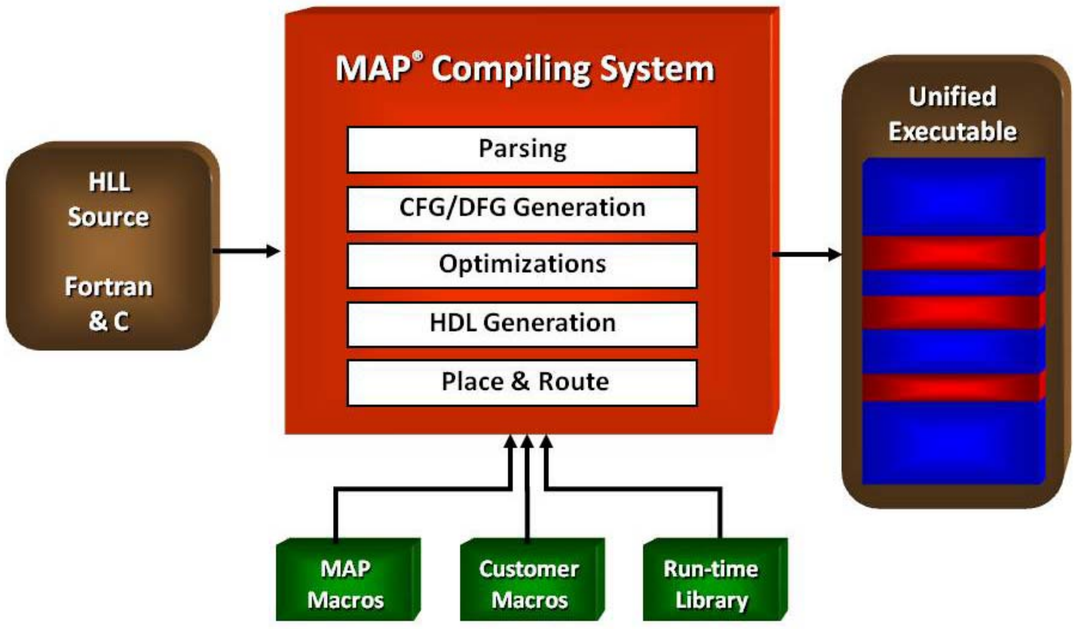


Figure 37. MAP Compilation process (From: [13])

D. SUMMARY

The SRC-6 is a reconfigurable computer comprised of two user programmable areas (microprocessor and FPGAs). This flexible computing system allows for the creation of unique problem specific code that can implement hardware not found in the microprocessor. This software and hardware design platform has a steep learning curve that can be overcome via training and practice. Optimization of code for the SRC-6 is accomplished by adhering to techniques that reduce memory dependencies and take advantage of pipelining.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SRC SPECIFIC TECHNIQUES

This chapter outlines a few lessons that helped reduce overall latency. In summary, the simple introduction of another variable reduces latency at the cost of another array. Often times, common C programming practices end up having a negative effect when utilized on the SRC-6. Arrays are one example. Reviewing the technical documents [10] and [13], and working one chapter at a time from both to learn about the unique techniques, will save a lot of time during the debugging phase. Overall latency of a program can be significantly reduced if one adheres to the techniques.

A. MEMORY IMPLEMENTATION

Recalling from Chapter III, there are several types of memory available for the programmer to use. This section will begin to cover a few of the limitations in dealing with the different types of memories.

1. On Board Memory

OBM is arranged into six banks. Each bank has a unique name (A, B, C, D, E, and F). Memory can only be accessed in 64b (64 bits) words. Using macros split_X_Y and combine_X_Y, a programmer can pack multiple data words into one memory address if the data elements are smaller than 64b. Where X indicates reduction parameter (64to4, 64to32, etc.) and Y is the data type combination (flt_flt, flt_int, ect). It is very important to efficiently utilize memory because the max OBM size is 523776 64b words.

Declaring OBM memory is done at the top of a MAP routine. There are only three predefined data structures that can be utilized with OBM. One can declare an Array, a 2d Array, or two arrays in the same bank. See Section C, Arrays, for a technique to create as many arrays as desired in a single OBM bank. Scalar variables are better left to BRAM.

```
OBM_BANK_A (IN, double, 100)

OBM_BANK_B_2D (OUT, double, 30, 20)

OBM_BANK_C_2_ARRAYS (A, double, 1000, B, int64_t, 900)
```

The first declares an array named IN made up of 100 elements with data type double in OBM bank A. The second declares a 30 x 20 2 dimensional array named OUT in bank B of data type double. The third declares two arrays in the same bank. A is an array of doubles of 1000 elements. B is an array of 900 elements of data type int64_t (64 bit interger).

To move data into and out of an OBM (to or from the microprocessor) requires two specific macros. DMA_CPU does the moving either to or from OBM memory and wait_DMA is a stall function to make the program wait till the data is transferred.

```
      DMA_CPU (<direction>, <OBM adr>, <OBM stripe>, <CM adr>, <CM stride>, <length>, <server>);

      wait_DMA (<server>);
```

`<direction>`	`Either CM2OBM or OBM2CM, specifies direction of data transfer`
`<OBM adr>`	`Start address in OBM`
`<OBM stripe>`	`Stripe/Stride address. Use MAP_OBM_stripe (1,”X”) where X is a combination of banks being striped across.`
`<CM adr>`	`CPU address`
`<CM stride>`	`Stride for the CPU address`
`<length>`	`Transfer length in bytes.`
`<server>`	`Server number in the range zero to eleven.`

2. Block RAM

This type of local memory is both flexible and fast. Existing within the FPGA, this is the preferred type of memory for variables and small arrays. BRAM can be made into any data type that is standard within the C language. Most often used in this thesis are integer (int), float (32 bit type), and double. As previously mentioned, there is a capacity of 444 BRAM units. Each unit consists of 2048 bytes. Declaring BRAM variables is as simple as they are in C. Here are a few examples:

```
float IN; //declares 32 bit floating point variable named IN
double OUT1[100]; //declare double named OUT1 with 100 elements
int i,j,k;  //declares 3 variables for loops
```

B. LOOPS & DEPENDENCIES

1. Loop-Carried Memory Dependency

Loop-Carried Memory Dependencies happen when a (for) loop writes to a memory location during one iteration and then reads it the next iteration. To compensate, the loop is slowed down to insure that a write in one iteration takes effect before the read occurs in the following iteration [10]. Table 6 shows one example of memory dependencies and how to fix it by introducing an extra variable to remember the previous value. This technique appears to be wasteful in memory but it will save a significant amount of time, two clock cycles per iteration run.

Table 6. Memory Dependency Example (From: [10])

Memory Dependant	**No Dependencies**

`for (i=1;  i<n;  i++){`
`      temp= a[i-1]+j;`
`      a[i]=temp;`
`}`
	`Prev=a[0];`
	`for (i=1;  i<n;  i++){`
	`    temp= prev+j;`
	`    a[i]=temp;`
	`    prev=temp;`
	`}`

2. Multiple Accesses to the Same Memory Bank

Use of OBM can be troublesome when accessing arrays. Accessing multiple arrays from the same bank causes the loop to slow down to allow for the reading/writing of values. Each extra read or write will cause a penalty of two clock cycles per read or write. The only way a loop will be fully pipelined is if there is only one memory access in the loop body. All other memory accesses add two clocks per iteration. Table 7 shows an example of a multiple access problem and how to work around it. Another way to work around this problem is to break up where the data is stored. If the data is stored in three BRAM arrays then the problem of reading the same array three times for one loop is avoided.

Table 7. Multiple accesses to same memory bank (After: [10])

Multiple Accesses to same memory	Work around

`for (i=0; i<n-2; i++)`
`    b[i]= a[i]+ a[i+1]+ a[i+3];`
	`for (i=0; i<n; i++){`
	`    a0=a1;`
	`    a1=a2;`
	`    a2=a[i];`
	`    if (n>=2) then`
	`        b[i-2] =a0+a1+a2;`
	`}`

C. ARRAYS

Program writing can become difficult when the number of needed arrays becomes greater than twice the number of OBMs, especially when one is trying to work through techniques such as those found in Chapter IV, Section B. Creating multiple 2-D arrays in a single OBM bank is not defined by Carte and to do so requires the use of pointers. The method is demonstrated in Table 8, which shows how several two-dimensional arrays can be implemented in one OBM. This technique is difficult to manage but can be very helpful when several two-dimensional arrays are needed and a programmer has run out of OBM banks to store data in individually. Again, be cautious of creating multiple accesses to the same memory array.

Table 8. Multiple 2d Arrays in one OBM (From: [14])

```
OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)
int array1,array2, offset;

   array1 = 0;
   array2 = 200000;

 for (i=0;i<nrow;i++) {
   for (j=0;j<ncol;j++)  {
   offset = i*ncol + j;
   value1 = AL[array1 + offset];
   }
   }
 for (i=0;i<nrow;i++) {
   for (j=0;j<ncol;j++)  {
   offset = i*ncol + j;
   value2 = AL[array2 + offset];
   }
                    }
```

D. PARALLEL REGIONS

Parallel Regions operate two independent sections of code at the same time. Each section of code can include normal constructs such as loops, pipelined loops and external macros [10]. Utilizing parallel regions, it is now possible to optimize code by executing two independent sections of code at the same time. Efficiency is maximized, if the section of code happens to take the same amount of time. For example, two independent for loops of the same iteration size, accessing independent OBMs would benefit from parallel sections. It is permissible to reuse loop names within the same parallel region. Table 9 shows the construct for parallel regions.

Table 9. Parallel Regions

```
#pragma src parallel sections{
#pragma src section{
int i;
for(i=0;i<100, i++)
a[i]=b[i]+c[i];
}
#pragma src section{
int i;
for(i=0;i<100, i++)
e[i]=d[i]+f[i];
}
}
```

E. USING TWO LOGIC CHIPS

This thesis is based upon the concept of expanding the previous code found in [2] onto two logic chips to reduce overall latency. The utilization of two logic chips adds new restrictions to what can be done but has the overarching positive effects of doubling the amount of logic and BRAM available to the programmer.

The logic must be broken into two separate routines. The decision of where to separate the logic is a difficult one and must be uniquely decided for each situation. Designing the code for this thesis led to a decision to break the code in a place where the load was approximately half, and the mathematical computations were easily separated because the current computation was complete.

One of the two subroutines must be designated for the “primary” chip and the other will be downloaded to the secondary chip. Certain restrictions apply to using two logic chips. A summary of the restrictions is found in Table 10. It is easiest to remember that the primary chip has control of all memory functions.

The scheme developed for this thesis that saves an enormous amount of time in DMA transfers is to utilize the OBMs for information that is needed by the other chip, and use BRAM for intermediate calculations. Using BRAM in-between also helps keep the latency low, as loop dependencies are avoided. The call to pass permissions takes only a few clock cycles compared to (number of bytes*data type) number of transfers. Using the permission passing technique also allows for maximum overall pipelining because the pipeline is not stopped for a data transfer but only for a few clock cycles to pass memory permissions.

Table 10. Dual Chip Restrictions (From: [10])

Primary Routine	Secondary Routine
• Issues all DMAs • Initially controls all accesses to OBM • Issues ONLY send_perms	• Has no access to MAP calling parameters • Cannot issue DMAs • Issues only recv_perms

To allow for synchronization between the two FPGAs during runtime, there are three 64b ports (named A, B, and C) that allow for the passing of one word on every clock cycle. These ports are best utilized to synchronize the two chips and to pass loop increment variables or for updating constants. Synchronization can be initialized from either routine and is implemented by the use of the flowing macro calls:

```
send_to_bridge<x> (<send args>); //on primary
recv_from_bridge<x> (<receive args>); //on secondary

<x> is either A,B,or C
<send args> must be a 64b scalar value
<receive args> must be a pointer to a 64b scalar value
```

Memory permissions have to be managed. When the MAP starts up, all memory permissions reside on the primary chip and must be passed to the secondary chip prior to the use of the OBM bank on the secondary chip. During this time, the primary chip is unable to access the OBM banks and it does not retain permission to use. When all routines on the secondary chip are complete, memory permissions should be returned to the primary chip. The primary chip is the only chip that has the ability to DMA to and from the microprocessor. A simple technique for sending the permissions has been developed in [10]. Create a mask of which permissions should be sent by ORing the names of the banks to sent. Pass the mask name as an argument in the call to send_perms. To remove permissions to the secondary chip, set the mask equal to zero. The primary chip controls memory permissions. For an example, see Table 11.

Table 11. OBM Permission Example

Send permissions	Remove permissions

`mask=  OBM_A || OBM_D;`
`send_perms (mask);`
	`mask=0;`
	`send_perms(mask);`

Load balancing of the two subroutines is important because when the MAP is turned on, both chips will begin to execute code at the top of each subroutine. Not critical, but taking advantage of dead time by doing any precalculations, array initialization, or memory management is best done during the “idle” time. One example is to precalculate all sine and cosine values needed for an FFT and store them into one two-dimensional array. Sine and cosine are computationally expensive and using a FOR loop to calculate all needed values prevents multiple implementations of the sine/cosine hardware.

F. SPACE SAVING TIP

One way to save space on the FPGA is to become aware of functions that take up a lot of space. One such item is the Sine and Cosine module. To prevent multiple implementations, create an array of sine and cosine values that are needed. Then index into the array and read the value needed. Another learning point is that for the cost of calculating the sine, the cosine is free. Make a two-dimensional array with these values. Another way to save even more space is to create the sine/cosine array on the µP and download by DMA to the MAP. Remember to pass memory permissions to the second FPGA when the values are needed on the second chip. The example below shows how to generate the sine/cosine two-dimensional array. A variable named Pi2 was created as another space saving variable that prevents from multiple calculations of π/2.

```
//build sin_cos array  - used to save sine and cosine resources
for (index =1; index<=4; index++){
   SIN_Array[index]=sinf(pi2/(1<<index));
   COS_Array[index]=cosf(pi2/(1<<index));
      }
```

Another way to save space is to pay special attention to format conversion. Format conversions come from implicit casts from one data type to another. One type cast that is particularly expensive is the cast from float to double or double to float. This can be prevented by ensuring that both variables are of the same type.

G. SUMMARY

To take advantage of the SRC-6 reconfigurable computer architecture the programmer should take advantage of the techniques laid out within this chapter. Careful utilization of memory, avoiding loop dependencies, using parallel regions, and utilizing both user programmer logic chips are all ways to optimize your code when using the SRC-6. These techniques will prove to save time and space when implementing a dual chip design.

V. DUAL CHIP DATA STREAMING DESIGN

A. CONCEPT OF OPERATIONS

The data path for the dual chip data streaming design is shown in Figure 38 and starts in the top left of the diagram on the primary chip. As data is processed along the path through Channelize and Hamming steps, the secondary chip sits idle and progresses up to the first synchronization point. Synchronization is done at several steps to make sure the flow path is maintained. Memory permissions are passed from the primary to the secondary chip.

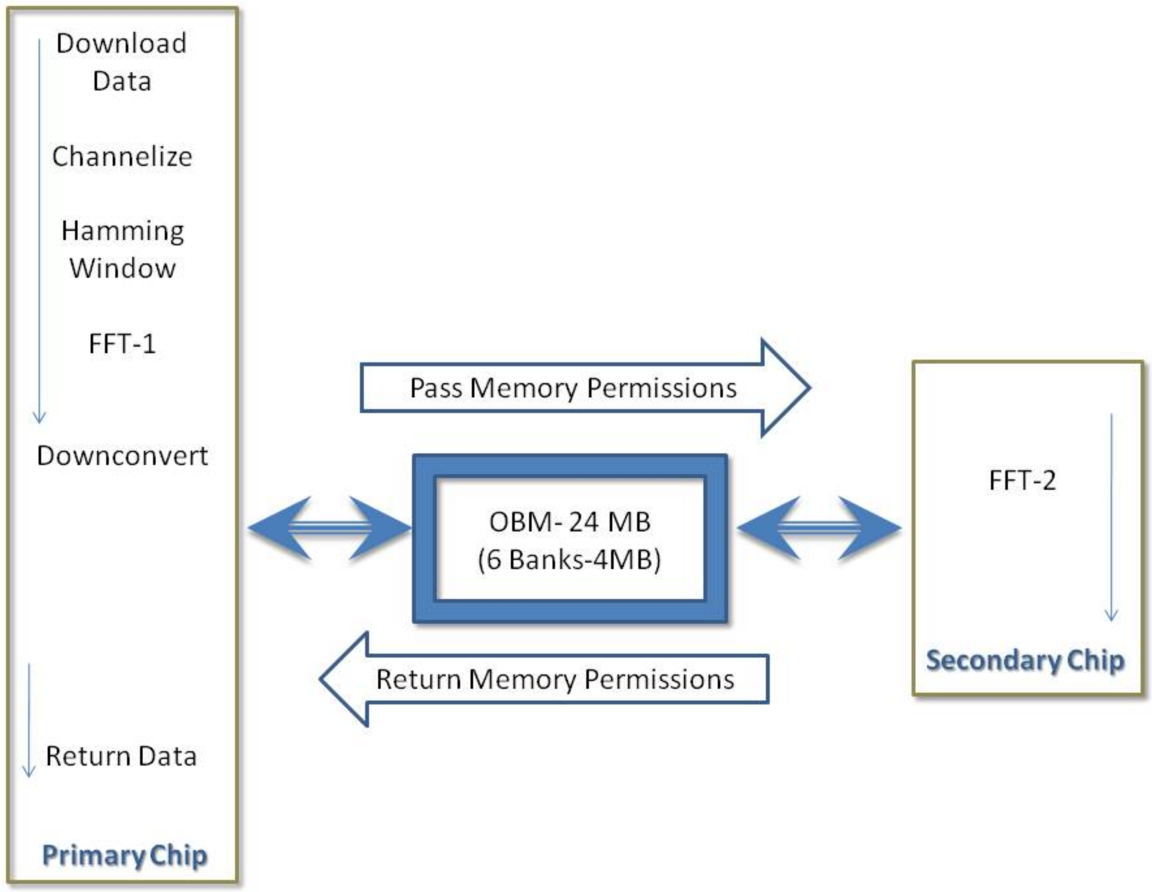


Figure 38. Dual FPGA Data Streaming Design

Once FFT2 is complete, memory permissions are passed back to the primary chip. The data can only be returned to the microprocessor from the primary chip, as this function is only available on the primary chip. Time is saved in this design by avoiding passing data back and forth between the MAPs and the microcontroller. Instead, only memory permissions are passed. The length of time to pass memory permissions stays constant no matter what the sizes of the data packets are. As this project grew, the amount of data being used also grew. Almost .34 seconds are expended in the original design by transferring data back and forth in-between stages. This is avoided by passing the OBM permissions and costs only a few clock cycles in comparison.

B. MEMORY USAGE

A memory management strategy was developed for this project. Memory managed in banks limits the maximum size of a memory structure to 523776 64b words in each OBM.

1. Onboard Memory (OBM) Management

OBM usage required planning to prevent running out of memory or overwriting the same location with new data before the old data is utilized. To manage the data, a memory usage plan was developed. Considerations for OBM usage was for any array that needed to be passed to the other chip. Incoming and outgoing data, with respect to the µProcessor, was stored in OBM. Table 12. represents the OBM usage plan for this project. The incoming data (array a1 in bank E) and the outgoing data (finalI and finalQ in banks A and B) are in the primary chip because only the primary can DMA with the µProcessor. The decision was made to logically split the data path in Figure 4 after the first FFT, and before the second FFT, where the multiplication is accomplished. After completing the multiplication, the size of the working arrays changes. This fact alone is the reason why the code was split after the first FFT and prior to the multiplication for the Computation of product sequences. This also creates a balanced load for each chip.

To allow the data to be utilized by both chips, the same array has to be declared in both macros, as is done in Table 12 in banks C and D.

Table 12. OBM Usage Plan

Primary Chip
`OBM_BANK_A_2D (finalI, double, P, Np*Np)`
`OBM_BANK_B_2D (finalQ, double, P, Np*Np)`
`OBM_BANK_C_2D (I_postFFT1, double, Np, P)`
`OBM_BANK_D_2D (Q_postFFT1, double, Np, P)`
`OBM_BANK_E    (al, double, NN  ) //input data`
Secondary Chip
`OBM_BANK_A_2D (finalI, double, P, Np*Np)`
`OBM_BANK_B_2D (finalQ, double, P, Np*Np)`
`OBM_BANK_C_2D (I_postFFT1, double, Np, P)`
`OBM_BANK_D_2D (Q_postFFT1, double, Np, P)`

Note there are restrictions for the data types that can be stored in OBM. Recall from chapter IV that OBMs can only transfer 64b data types. To store 32-bit floats requires a data packing operation to be done. This concept will help conserve memory because one can store two floats in one memory location. Remember that if you are combining data to an outbound array, the proper data type pointers need to be declared in main.c.

2. Block RAM (BRAM)

BRAM was used for the remaining variables and arrays. BRAMs are the memory of choice when creating pipelined structures because perfect pipelines can be created without the possibility of dirty data. Dirty data is data that is written back to the same array or variable it was read from creating a possibility of cross-contaminating memory. Keeping the data flowing through the pipeline also maintains high throughput. One of the goals listed in this thesis was to increase throughput at the expense of hardware. That is why extra care is taken to move along data and prevent loop slowdowns and memory dependencies by utilizing extra BRAMs. As resolution is decreased, the need for more memory is increased and compromises will be made with respect to the BRAM and OBM that may slow down the algorithm because of the need to reuse memory.

C. FILE TYPES

Previous chapters have mentioned that several types of files are required for the compilation process to generate the application executable file. The software described by this thesis uses three file types that the programmer is responsible for maintaining. The Makefile, .c, and.mc file types are needed for the described work. The Makefile is a template file found in [10]. SRC recommends starting with their template and customizing it. The Makefile for this work started from the same template. The Makefile is the instruction list for the compiler to know which files are to be compiled. There is a section for compiler options and flags that can be set by the user. The SRC documentation overlooks discussion of compiler options.

Place and route, Make, and bitgen options have an enourmous impact on the compilability of more complicated programs. SRC documentation does not list these options in the CARTE documentation [10] but they can be found in Xilinx documentation [15], see Table 13. The SRC compiler for the Virtex FPGAs uses Xilinx software for parts of the compilation process.

Table 13. Effort Level Options (From: [15])

Option	Function	Range	Default
–ol *overall effort_level*	Placement and routing effort level	std, med, high	std (Overall effort level std)
–pl *placer_effort_level*	Placement effort level (overrides –ol value for the placer)	std, med, high	Determined by the –ol setting
–rl *router_effort_level*	Routing effort level (overrides–ol value for the router)	std, med, high	Determined by the –ol setting
–xe *extra_effort_level*	Set extra effort level	normal, continue	No extra effort

The .c and .mc file types are similar in appearance because they are both written in ANSI C. The only difference is that the .c will be specified for the µProcessor and the .mc will be compiled for the MAP. This is accomplished in the beginning of the Makefile. For this thesis, three additional files in addition to the Makefile were written. Dcmain.c, dcp.mc, and dcs.mc. The following code segment is from the Makefile showing how the files are targeted for the appropriate hardware. The Makefile also allows for custom designation of the name of the executable. This is set by the BIN variable. The variable dc is used for this thesis.

```
#Files targeted to µProcessor
FILES       = dcmain.c # dual chip main file
#Files targeted to MAP
MAP_E_FILES = dcp.mc \ # dual chip primary chip file
              dcs.mc \ # dual chip secondary chip file
BIN         = dc        #name of executable

# ---------------------------------
# Multi chip info provided here
# Designate files for specific FPGAs
# ---------------------------------
PRIMARY     =dcp.mc
SECONDARY   =dcs.mc
CHIP2       =dcs.mc
```

This thesis uses multiple FPGAs on one MAP so a file for each FPGA is needed. Dcp.mc is the Primary file, dcs.mc is the secondary file. The flag CHIP2 must be set to the name of the file that is targeted for the second chip. The complete code listing is in:

APPENDIX B: makefile.
APPENDIX C: dcmain.c
APPENDIX D: dcp.m
APPENDIX E: dcs.m
APPENDIX F: misc files.

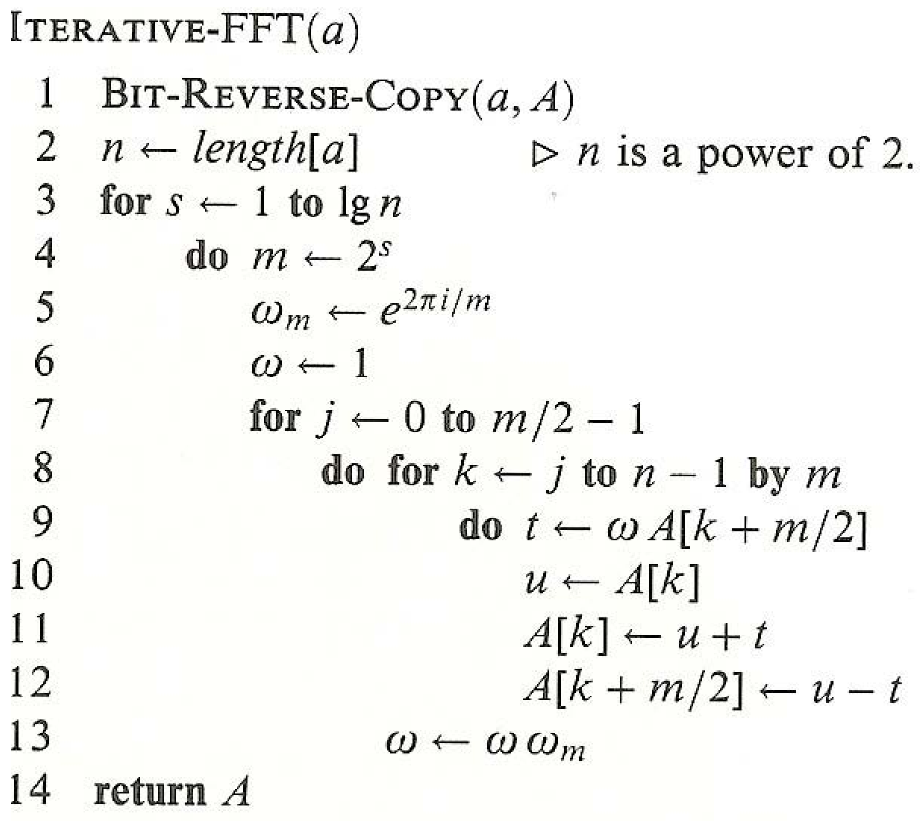
D. FAST FOURIER TRANSFORM

There are two FFTs in the Cyclostationary FAM algorithm. Once the data management plan was implemented, the majority of time is spent on the FFTs. (Chapter VI covers timing analysis in detail.) The second FFT is smaller in size (8 point) but performs more calculations. The second FFT is performed 4096 iterations. The FFT algorithm, as designed in [2], which came originally from [16], was very efficient when implemented as a standard C program. However, when ported to SRC-6 code, several memory dependencies and loop slowdowns were introduced. These slowdowns (similar to the topics covered in Chapter IV) resulted in a clock per iteration of 47 and pipeline depth of 52. When this research was started, this was identified as one place that could use the techniques from Chapter IV. The improvements were incremental, as shown in Table 13. To achieve the results of Revision 2, the FFT algorithm in Figure 39 was utilized as a starting point. To achieve pure pipelining, the outer loop was removed and in its place n (three for an eight point FFT) copies of the inner loop are used. This changes the further usefulness of the FFT algorithm as it is no longer scalable. This was acceptable because of memory limitations within the system. Having a fixed size FFT ensures that everything will fit within two FPGAs. Reconfiguration times are unacceptable for a real-time system. Recompiling all of the files required usually takes approximately 4 hours from start to finish. The calculations for latency found in Table 14 are representative of one complete computation of a single FFT on n points. The formula for Latency is shown in equation 1.8 and is simply the clocks per iteration times the pipeline depth. For revision 1 and 2, this number is multiplied by three because there are 3 loops of each pipeline. Revision 2 is made up of two loops to achieve 1 clock per iteration. An additional pipelined loop of 5 is needed to reformat the memory structure utilized into the one that is expected by the main program.

$$\text{Latency} = \text{Clocks per Iteration} * \text{Pipeline Depth} \qquad (1.9)$$

Table 14. FFT timing improvements

	Original	Revision 1 (3x)	Revision 2 (3x)
Clocks per Iteration	47	2	1
Pipeline Depth	52	34	(33+5)
Latency	2444	204	114



```
ITERATIVE-FFT($a$)
BIT-REVERSE-COPY($a, A$)
$n \leftarrow length[a]$                ▷ $n$ is a power of 2.
for $s \leftarrow 1$ to $\lg n$
    do $m \leftarrow 2^s$
       $\omega_m \leftarrow e^{2\pi i/m}$
       $\omega \leftarrow 1$
       for $j \leftarrow 0$ to $m/2 - 1$
           do for $k \leftarrow j$ to $n - 1$ by $m$
                  do $t \leftarrow \omega A[k + m/2]$
                     $u \leftarrow A[k]$
                     $A[k] \leftarrow u + t$
                     $A[k + m/2] \leftarrow u - t$
              $\omega \leftarrow \omega \omega_m$
return $A$
```

Figure 39. FFT pseudo code (From :[16])

1. In-place Butterfly Design

The design of this FFT, as realized by the code in Appendix B, is based off implementing several parallel copies of a traditional butterfly design. The image in Figure 40 shows what the three-stage pipeline architecture of FFT2 looks like, as implemented by this thesis.

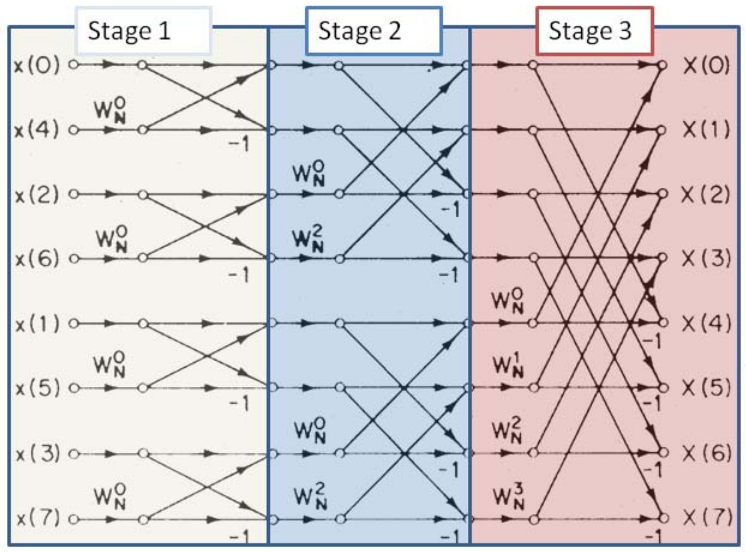


Figure 40. Three stage pipeline of FFT2.

Each stage in Figure 40 is separated by a BRAM of data type float. When stage 1 has finished with the computation the results are written to BRAM. Stage 2 then reads from memory, performs its calculation, and passes the results along in the same manner. This pipeline eliminates most of the data dependencies of the original version. Another point to note is all butterflies in each stage are computed in parallel. This also reduces latency over other designs. Because each butterfly is implemented in hardware, this design is fast and utilizes the most hardware.

A second technique was applied by creating a copy of the input data to eliminate multiple accesses to the same memory bank. This requires a small additional step of making a copy of the data. This can either be done when the data is created or as a small additional loop. To keep the memory dependencies down throughout the FFT, copies of each stage are also created for the next stage. The code below shows how necessary code was altered to create a second copy of the data with minimal increase in delay. This section of code performs two functions. To arrange the data for the first stage, the data is rearranged to the sequence shown on the left side of Figure 40. At the same time, the data is split into the real and complex parts to do the multiplications separately. The SRC-6 does not have complex multipliers. This code segment originated in [2] and was altered to create the a1_copy and a2_copy arrays to eliminate the data dependencies. The code segment below shows how additional copies of data are created for use in this thesis.

```
/* reorder input and split input into real and complex parts */
      for (i=0; i<n; i++)
      {
         /* reverse bits 0 thru k-1 in the integer "a" */
            for (ii=o=0, p = 1, q = 1<<(log2n-1);
               ii<log2n;
               ii++, p <<= 1, q >>= 1 ) if (i & q) o = o | p;

         j = (int)o;
         a1[j] = I[i];
         a2[j] = Q[i];
         a1_copy[j] = I[i];
         a2_copy[j] = Q[i];
      }
```

2. Growth of the FFT

The FFT implemented for FFT2 is fast and minimizes memory dependencies and loop dependencies. This FFT is still flawed in that it is not scalable on the fly. Changing the size of the FFT requires several hours of programming and testing prior to recompilation.

Growing this FFT requires a basic understanding of the data path shown in Figure 41. To make a larger FFT, calculate the number of required stages (S) utilizing:

$$\begin{aligned} &S = \log_2(n) \\ &\text{where } n = \text{ number of points of FFT} \end{aligned} \qquad (1.10)$$

The number of stages is the number of loops needed for the FFT. Additional variables are also needed to go between stages. The output from one stage becomes the input to the next stage. The output of the last stage must be the input to the data flip portion to ensure that all stages are implemented in the pipeline. The variable n at the top of the program must be adjusted to the new n so the bit reversal section works at designed.

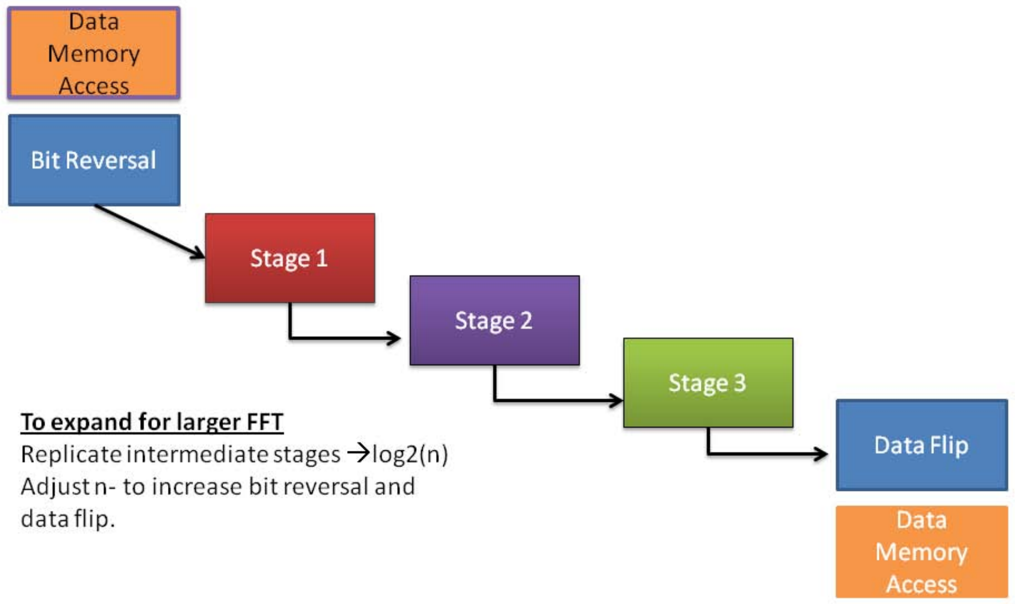


Figure 41. Growth of FFT2

E. TIMING ANALYSIS

The goal of this thesis was to reduce the overall latency compared to the previous thesis [2], utilizing the same algorithm and modifying it further to take advantage of the architecture of the SRC-6. When this research was started, there was no specific number in mind, other than the thought that the extra expense in hardware should justify a significant reduction in latency between data in and data out. This goal was accomplished successfully, as shown in Table 15.

The overall run time is the most important because it is the measure of when the SRC-6 can provide an output that is usable to the consumer of the data. When missiles or airplanes are flying in toward a ship, seconds count. Having a hardware solution that can give the user a meaningful output in less than a second is critical. Comparing 2.170836 seconds to .38093 seconds, one can conclude that this is a significant improvement in the right direction. A second comparison is to observe the amount of time that both algorithms spent on the FPGAs is practically identical. Careful understanding of what is happening in the dual chip design would highlight that the .25 seconds spent on the FPGAs is for both FPGAs working in parallel and is another example of how parallel computing can lead to efficient computing.

Another major goal of this thesis was to reduce the amount of time performing DMAs. A major pay off was accomplished by eliminating all but the initial download and return of data to the µP. A reduction from .3898 seconds to .000698 seconds helps account for the significant reduction in run time for the processed data.

Table 15. Final timing comparison.

Upperman's Thesis	This Thesis
Execution times: 2.170836 seconds total Of the total time: 1.313759 seconds were spent on the CALLS to FFTs 0.458690 seconds were spent on the MAP for FFTs Of the time spent on the MAP for FFTs: 0.382998 seconds were spent on DMAs 0.075691 seconds were spent in the FFT loop 0.116433 seconds were spent on the CALLS to Channelize 0.000157 seconds were spent on the MAP for Channelize Of the time spent on the MAP for Channelize: 0.000024 seconds were spent on DMAs 0.000133 seconds were spent channelizing 0.106429 seconds were spent on the CALLS to Downconvert 0.003294 seconds were spent on the MAP for Downconvert Of the time spent on the MAP for Downconvert: 0.000730 seconds were spent on DMAs 0.002564 seconds were spent downconverting Execution time not including calls and data transfers: 0.250462 seconds Results from Cyclostationary FAM algorithm written to: FAM_result.txt	Execution times: 0.380397 seconds total 0.250055 seconds on FPAG Of the total time: 0.038027 seconds were spent performing FFTs Of the time spent on the MAP for FFTs: 0.000958 seconds were spent on FFT1 0.037069 seconds were spent on FFT2 0.000149 seconds were spent on Channelize 0.004611 seconds were spent on Downconvert 0.006939 seconds were spent postFPGA data processing 0.000686 seconds were spent on DMAs

F. FUTURE WORK

Greater resolution allows for data that is more useable by the ultimate customer. To achieve a resolution greater than or equal to the recommended N=1024, more work is required. To reach these combinations, different FFTs are needed, as mentioned previously. One idea not yet tested is to use Intellectual Property of previously designed FFTs found in the Xilinx Project Navigator to implement the different FFTs that are needed. The SRC-6 has an FFT macro that does an FFT for sizes greater than n=256. Upperman experimented with this and found it to waste a lot of time calculating the FFT for points that were zero. This was accomplished by stuffing zeros anywhere there was not a point.

The original algorithm implemented for the FFTs has memory dependencies and loop dependencies that were eliminated by utilizing the recommended techniques mentioned previously in this thesis. Utilizing the technique outlined by Figure 41 yielded a design that exceeded available slices on one FPGA. NPS is scheduled to receive a CARTE update that may help with this problem. The update to CARTE has new floating-point macros that should save approximately 10% in space.

Recall that the benefit of the Cyclostationary algorithm is the extractability of the parameter for Code Rate. The minimum resolution determined by experimentation in this thesis is a result of N >= 1024. Not meeting the minimum resolution makes the Cyclostationary Block equal in performance to the other blocks of Figure 1. Other algorithms are more efficient to implement and would make better algorithms for obtaining center frequency and bandwidth. Once new FFTs are designed, it would be easy to update the code in Appendices B-F for a higher resolution.

G. SUMMARY

The Dual Chip Data Streaming Design developed for this thesis was able to successfully utilize both FPGAs on a single MAP by implementing techniques of the previous chapter. Overall runtime is improved to approach desired specifications that make sense implementation into a real-time system. A path of development is laid that can take this project to the next level by developing a SRC-6 specific FFT that has less than 256 points. The future of the Autonomous LPI System is bright as lessons learned in this thesis can be applied to the development of all modules in Figure 1.

VI. CONCLUSIONS

The goals for this thesis have been achieved in both reducing overall latency and implementing the Cyclostationary FAM algorithm on one MAP utilizing two FPGAs. Achieving a run time for a sample of data that is less than a second is satisfactory proof of concept for utilizing MAPs for creating the Autonomous LPI system in Figure 1. A greater reduction in overall runtime would result from the implementation a more time and space efficient FFT. Another possibility is to use faster MAPs as they become available. The new H series map is clocked at 150 MHz vs. the 100 MHz of the E-series.

A path for future development has been laid out and requires that N be greater than 1024. This will ensure that frequency and cycle-frequency resolutions are sufficient to obtain the most useable data from a single iteration of the program. Chapter V highlighted a few methods for moving forward to achieve these resolutions by designing a new FFT or utilizing IP from Xilinx. FFTs are a common macro needed for several of the blocks of Figure 1. Further study of an SRC-6 FFT would be beneficial to the overall project.

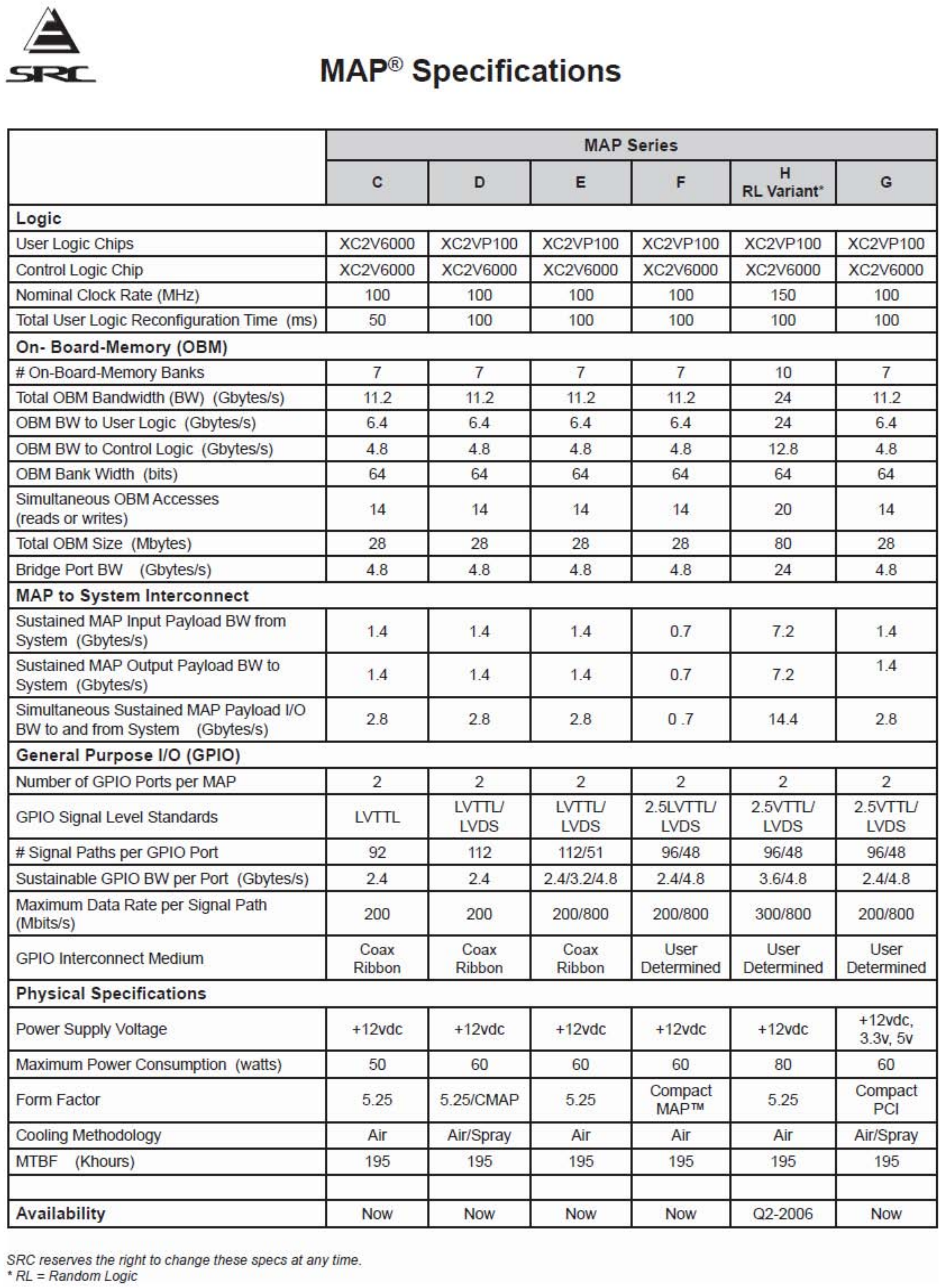
The SRC-6 is more difficult to utilize than advertised but does provide amazing results once the programmer understands one basic concept. Take advantage of what the SRC-6 does well and avoid things (floating point, sine/cosine operations) that are costly in both timing and space.

Another concept realized by this thesis is that at the cost of hardware, overall latency can be reduced. This important trade off is necessary when more work needs to be done in the same amount of time. Removing unnecessary data transfers by implementing a data management plan that passes memory permissions saves time over moving data that takes a long time. The time needed for data transfer is directly proportional to the amount of data being transferred.

The path forward for this project is optimistic but will be difficult. Careful attention to resources is required, as memory will become more limited and number of slices available dwindles. Consideration of multi-core processors may lead to another solution that will demonstrate timing better than those presented in this thesis.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MAP SPECIFICATIONS



MAP® Specifications

	MAP Series					
	C	D	E	F	H RL Variant*	G
Logic						
User Logic Chips	XC2V6000	XC2VP100	XC2VP100	XC2VP100	XC2VP100	XC2VP100
Control Logic Chip	XC2V6000	XC2V6000	XC2V6000	XC2V6000	XC2V6000	XC2V6000
Nominal Clock Rate (MHz)	100	100	100	100	150	100
Total User Logic Reconfiguration Time (ms)	50	100	100	100	100	100
On- Board-Memory (OBM)						
# On-Board-Memory Banks	7	7	7	7	10	7
Total OBM Bandwidth (BW) (Gbytes/s)	11.2	11.2	11.2	11.2	24	11.2
OBM BW to User Logic (Gbytes/s)	6.4	6.4	6.4	6.4	24	6.4
OBM BW to Control Logic (Gbytes/s)	4.8	4.8	4.8	4.8	12.8	4.8
OBM Bank Width (bits)	64	64	64	64	64	64
Simultaneous OBM Accesses (reads or writes)	14	14	14	14	20	14
Total OBM Size (Mbytes)	28	28	28	28	80	28
Bridge Port BW (Gbytes/s)	4.8	4.8	4.8	4.8	24	4.8
MAP to System Interconnect						
Sustained MAP Input Payload BW from System (Gbytes/s)	1.4	1.4	1.4	0.7	7.2	1.4
Sustained MAP Output Payload BW to System (Gbytes/s)	1.4	1.4	1.4	0.7	7.2	1.4
Simultaneous Sustained MAP Payload I/O BW to and from System (Gbytes/s)	2.8	2.8	2.8	0 .7	14.4	2.8
General Purpose I/O (GPIO)						
Number of GPIO Ports per MAP	2	2	2	2	2	2
GPIO Signal Level Standards	LVTTL	LVTTL/ LVDS	LVTTL/ LVDS	2.5LVTTL/ LVDS	2.5VTTL/ LVDS	2.5VTTL/ LVDS
# Signal Paths per GPIO Port	92	112	112/51	96/48	96/48	96/48
Sustainable GPIO BW per Port (Gbytes/s)	2.4	2.4	2.4/3.2/4.8	2.4/4.8	3.6/4.8	2.4/4.8
Maximum Data Rate per Signal Path (Mbits/s)	200	200	200/800	200/800	300/800	200/800
GPIO Interconnect Medium	Coax Ribbon	Coax Ribbon	Coax Ribbon	User Determined	User Determined	User Determined
Physical Specifications						
Power Supply Voltage	+12vdc	+12vdc	+12vdc	+12vdc	+12vdc	+12vdc, 3.3v, 5v
Maximum Power Consumption (watts)	50	60	60	60	80	60
Form Factor	5.25	5.25/CMAP	5.25	Compact MAP™	5.25	Compact PCI
Cooling Methodology	Air	Air/Spray	Air	Air	Air	Air/Spray
MTBF (Khours)	195	195	195	195	195	195
Availability	Now	Now	Now	Now	Q2-2006	Now

SRC reserves the right to change these specs at any time.
** RL = Random Logic*

Figure 42. MAP Processor Specifications (From : [17])

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. MAKEFILE

```
# ---------------------------------
# Origional Template from CARTE 2.2
# Modified by: LT Wesley Simon
# For        : thesis
# Date       : Septemeber 2009
# ---------------------------------

FILES       = dcmain.c
MAP_E_FILES = dcp.mc\
              dcs.mc
BIN         = dc

# ---------------------------------
# Multi chip info provided here
# (Leave commented out if not used)
# ---------------------------------
PRIMARY     = dcp.mc
SECONDARY   = dcs.mc
CHIP2       = dcs.mc
MAPTARGET   = map_e

# ----------------------------------
# User supplied MCC and MFTN flags
# ----------------------------------

MCCFLAGS    = -log -use_par -v -keep
MFTNFLAGS   = -log

# ----------------------------------
# User supplied flags for C & Fortran compilers
# ----------------------------------
CC          = icc       # icc   for Intel cc for Gnu
FC          = ifort     # ifort for Intel f77 for Gnu
LD          = icc       # for C codes

CFLAGS      =-O3 -tpp7 -xW
MY_FFLAGS   =

PAR_OPTIONS = -ol high -t 50

# ----------------------------------
# No modifications are required below
# ----------------------------------
MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)

mydebug: debug
myhw: hw
myclean: clobber
      rm -rf *~
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C.   DCMAIN.C

```
// ---------------------------------
// FILE      : dcmain.c
// description: File will reside on microprocessor and is responsible
for downloading
//             data to the FPGA and returning data. some post FPGA
processing is done
//             along with writing to files.
// Original author: Gary Upperman
// Modified by: LT Wesley Simon
// For             : thesis
// Date      : Septemeber 2009
// ---------------------------------

//Include files
#include <libmap.h>
#include <map.h>
#include <time.h>
#include <math.h>
#include "high_prec_time.c"

#define pi 3.141592653589793

FILE *I_ptr; // pointer to the I-channel input file name
FILE *IFFT1_Out;  // pointer to the I-channel output file name
                  // for the first FFT results
FILE *QFFT1_Out;  // pointer to the Q-channel output file name
                  // for the first FFT results
FILE *IFFT2_Out;  // pointer to the I-channel output file name
                  // for the second FFT results
FILE *QFFT2_Out;  // pointer to the Q-channel output file name
                  // for the second FFT results

FILE *Output;     // pointer to the final data output file name


void PrimaryChip (double *, double *, double (*)[], double
(*)[],int64_t *,int64_t *,int64_t *,int64_t *,int64_t *, int);

//dcmain function
int main()
{
   /* DECLARE VARIABLES AND CONSTANTS */
      /* declare file names and path */
         char I_file[] = "I_channel.txt";
         char IFFT1_Out_file[] = "IFFT1_out.txt";
         char QFFT1_Out_file[] = "QFFT1_out.txt";
         char IFFT2_Out_file[] = "IFFT2_out.txt";
         char QFFT2_Out_file[] = "QFFT2_out.txt";
         char Output_file[] = "FAM_result.txt";

      /* Declare Input Variables */
```

```
         int fs = 7000; // sample frequency
         int df = 128;  // frequency resolution
         int M = 2;   // M = df/alpha

      /* Declare all timing variables and get first time hac;
         start timing */
         struct timeval start1, start2, start3, temp_stop, time1;
         struct timeval subr_go, subr_return, endofprogram,
startprogram, post_dataread;
         float overall_time = 0.0;
         float timeonFPGA = 0.0;
         float DMAtime = 0.0;
         float channelize =0.0;
         float downconvert= 0.0;
         float FFT1 =0.0,FFT2 = 0.0,FFT_time= 0.0;
         float datainput= 0.0;
         float postFPGA= 0.0;
         int64_t tchannel, tfft1, tfft2, tdownconv,tDMA;
         int timei;

      //INITIAL TIMING MARKER
         gettimeofday(&startprogram, NULL);

      //*****declare additional variables*******
      /* calculate dalpha */
         double dalpha = df/M;
      /* determine number of input channels: fs/df */
         double Np = pow(2.0, ceil(log10(fs/df)/log10(2)) );
      /* overlap factor in order to reduce the number of short time
fft's.
         L is the offset between points in the same column at
consecutive
         rows.  L shoud be less than or equal to Np/4
         (Prof. Loomis paper) */
         double L = Np/4;
      /* determine number of columns formed in the
         channelization matrix (x) */
         double P = pow(2.0, ceil(log10(fs/dalpha/L)/log10(2)) );
      /* determine total number of points in the input data to
         be processed */
         double N = P*L;
      /* declare other variables and arrays to be used
         Note: I tried to declare them in the order in which they
         are needed.  Some were consolidated. */
      /* Loop Indexes */
         int i = 0, j = 0, k=0, index = 0;
      /* Array to contain values from input file */
         float *I_Values;
         I_Values = (float*)malloc(N * sizeof(float));
      /* Initial Array and Matrix */
         double NN = (P-1)*L+Np; // resizes x array
         double *x;
         x  = (double*)malloc(NN * sizeof(double));
         double *HAM;
         HAM  = (double*)malloc(Np * sizeof(double));
         int xInitialMax; // book-keeping on x array
      /* Declare Variables used for MAP allocation */
         int nmap = 1, mapnum = 0;
         double (*Ifinal)[(int)(Np*Np)];
         double (*Qfinal)[(int)(Np*Np)];
         Ifinal = Cache_Aligned_Allocate(P * Np * Np *sizeof(double));
         Qfinal = Cache_Aligned_Allocate(P * Np * Np *sizeof(double));
         int a;
      /* Declare Output Variables */
         float IXF2[(int)P][(int)(Np*Np)], QXF2[(int)P][(int)(Np*Np)];
         double (*MM)[(int)(Np*Np)], (*Sx)[2*(int)N+1];
         float c, p, alpha, f, kk, ll, Sx_max;
         double Iscr, Qscr;
         int64_t joverNp, rem;
         Sx = Cache_Aligned_Allocate((Np+1) * ((2 * N)+1)
*sizeof(double));
         MM = Cache_Aligned_Allocate( ( (int)((3*P/4)-(P/4))+1) * Np *
Np * sizeof(double));

   /* GET SECOND TIME HAC; STOP TIMING TO BRING DATA IN */
      gettimeofday(&temp_stop, NULL);

      printf("\n");
   /* OPEN THE INPUT FILES */
      I_ptr = fopen(I_file, "r");
      if (I_ptr==NULL)
      {
         printf("Error opening I-channel input file.\n");
         return(1);
      }

   /* READ IN THE I-CHANNEL FILE */
      while ( (fscanf(I_ptr, "%f", &I_Values[i]) != EOF) && (i<N) )
      {
         x[i] = I_Values[i];
         i++;
      }

   /* This Loop fills the x array with zeros if there wasn't N
      rows of data in the input file */
         if (i < N)
            while (i < N)
            {
               x[i] = 0;
               i++;
            }
         xInitialMax = i;
         fclose(I_ptr);

   /* GET THIRD TIME HAC; RESTART TIMING */
      gettimeofday(&post_dataread, NULL);
      timei = timeval_subtract (&time1,&post_dataread, &temp_stop);
      datainput = time1.tv_sec + time1.tv_usec*1.0e-6;
```

```
/*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%
   INPUT CHANNELIZATION - this part limits the total number of points
to be
          analyzed. It also generates a Np-by-P matrix, X, with shifted
          versions of the input vector in each column.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% */
   /* Zero fill x if we don't have NN samples.  The loop does the
      xx(NN) = 0 loop in the Matlab code. */
      for(i=xInitialMax; i<NN;i++)  x[i] = 0;

   /* Reserve MAP */
      if (map_allocate(nmap))
      {
         fprintf(stdout, "Map allocation failed for
channelization.\n");
         exit(1);
      }
 /* SET up Hamming window for MAP */
 for (i=0; i<Np; i++)
      {
         HAM[i] = 0.54 - 0.46*cosf(2*pi*i/(Np-1));
      printf("\t %1.5f",HAM[i]);
      }
   /* Take time hac */
      gettimeofday(&subr_go, NULL);

/******** Call subroutine and Restart Timing ***********************/

      PrimaryChip(x,HAM,Ifinal,Qfinal,&tchannel, &tfft1, &tfft2,
&tdownconv,&tDMA, mapnum);

/******************  Return from FPGAs  *************************/

      gettimeofday(&subr_return, NULL);
      channelize= tchannel*1e-8;
      downconvert=tdownconv*1e-8;
      FFT1 = tfft1*1e-8;
      FFT2 = tfft2*1e-8;
      FFT_time+=FFT1+FFT2;
      DMAtime=tDMA*1e-8;
      timei = timeval_subtract(&time1, &subr_return, &subr_go);
      timeonFPGA+= time1.tv_sec + time1.tv_usec*1.0e-6; // time spent
on FPGAs

     //partial printout for verification
     printf("\n********* ON CPU Readout of FFT 2 *****\n");
     for (a=0; a<5; a++)
      {
         printf("\t%2.8f+i*%2.8f\n", Ifinal[a][0], Qfinal[a][0]);
      }
   /* Free map */
      if (map_free (nmap))
      {
         printf("Map deallocation failed for downconversion.\n");
         exit(1);
      }
//Post FPGA Data Maniuation
/*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
   MATRIX MANIPULATION - implements the FFT shift and left/right flip
in
          the matlab code in one single loop
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% */
   /* Swap bottom and top halves: */
      for(i=0; i<=(P/2-1); i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            /* Bottom real half becomes top real half */
               IXF2[i][j] = Ifinal[i+(int)(P/2)][j];
            /* Top real half becomes bottom real half */
               IXF2[i + (int)(P/2)][j] = Ifinal[i][j];

            /* Bottom imaginary half becomes top imaginary half */
               QXF2[i][j] = Qfinal[i+(int)(P/2)][j];
            /* Top imaginary half becomes bottom imaginary half */
               QXF2[i + (int)(P/2)][j] = Qfinal[i][j];
         } // for j
      } // for i

   /* Obtain the magnitude of the complex values */
      for(i=(P/4)-1; i<(3*P/4); i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            Iscr = IXF2[i][j];
            Qscr = QXF2[i][j];
            MM[i-(int)(P/4)+1][j] = sqrtf( (Iscr*Iscr) + (Qscr*Qscr) );
         } // for j
      } // for i

/*
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
   DATA DISPLAY - display only the data inside the range of interest -
          centralizes the bi-frequency plane according to alpha0 and f0
          vectors.  Note: the alpha0 and f0 vectors are defined as
follows
          (in matlab terms):
             alpha0 = -fs  :fs/N :fs;
             f0     = -fs/2:fs/Np:fs/2;
          but are not declared in this program since they are only used
          for plotting the results.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% */
   Sx_max = 0;

   /* Clear Sx matrix since not every location is necessarily written
to.
      Seems like this loop is unnecessary, but I had instances where
old
      data in the memory was being used. */
      for (i = 0; i<Np+1; i++)
      {
         for (j=0; j<2*N+1; j++)
         {
            Sx[i][j] = 0;
         } // for j
      } // for i

   /* Determine Final Output */
      for(i=0; i<=.5*P; i++)
      {
         for(j=0; j<Np*Np; j++)
         {
            joverNp = (int)((j+1)/Np);
            rem = (j+1) - Np*joverNp;

            if(rem == 0)
            {
               c = .5*Np - 1;
            }
            else
            {
               c = rem - .5*Np - 1;
            }

            k = joverNp - .5*Np;
            p = i - .25*P;

            alpha = ((k-c)/Np) + ((p-1)/N);
            f = .5*(k+c)/Np;

           if (((alpha > -1) & (alpha < 1)) | ((f >-.5) & (f < .5)))
            {
               kk = 1+Np*(f + .5);
           if ( (kk-(int)kk) < .5) kk = (int)kk; else kk = (int)kk + 1;

               ll = 1+N*(alpha + 1);
           if ( (ll-(int)ll) < .5) ll = (int)ll; else ll = (int)ll + 1;

               Sx[(int)kk-1][(int)ll-1] = MM[i][j];

               /* find max value of Sx so it can be normalized later */
                  if(MM[i][j] > Sx_max) {Sx_max = MM[i][j];}
            } // end if
         } // for j
      } // for i
   /* Normalize Sx - ORIGINAL */
      for(i=0; i<Np+1; i++)
      {
         for(j=0; j<2*N+1; j++)
         {
            Sx[i][j] = Sx[i][j]/Sx_max;
         }
      }

   /* get fourth time hac (stop timing) and display: */

      gettimeofday(&endofprogram, NULL);
      timei = timeval_subtract (&time1, &endofprogram, &subr_return);
      postFPGA += time1.tv_sec + time1.tv_usec*1.0e-6;
      timei = timeval_subtract (&time1, &endofprogram, &startprogram);
      overall_time = time1.tv_sec + time1.tv_usec*1.0e-6;

      //display timing to terminal
      printf("Execution times:\n");
      printf("   %3.6f seconds total\n", overall_time);
      printf("   %3.6f seconds on FPAG\n", timeonFPGA);
      printf("\nOf the total time:\n");
      printf("   %3.6f seconds were spent performing FFTs\n",
FFT_time);
      printf("   Of the time spent on the MAP for FFTs:\n");
      printf("      %3.6f seconds were spent on FFT1\n", FFT1);
      printf("      %3.6f seconds were spent on FFT2\n", FFT2);
      printf("   %3.6f seconds were spent on Channelize\n",
channelize);
      printf("   %3.6f seconds were spent on Downconvert\n",
downconvert);
      printf("      %3.6f seconds were spent postFPGA data
processing\n", postFPGA);
      printf("      %3.6f seconds were spent on DMAs\n", DMAtime);
      printf("\nExecution time not including data input and DMA time:
%3.6f seconds\n", overall_time - datainput-DMAtime);

   /* PRINT OUTPUT FILE */
      Output=fopen(Output_file, "w");
      if(Output == NULL)
      {
         puts("Error creating output file.");
         return(1);
      }

      for(i=0; i<Np + 1; i++) // Np + 1 for Sx, 5 for MM
      {
         for(j=0; j<2*N + 1; j++)  // 2*N + 1 for Sx, Np*Np for MM
         {
            fprintf(Output, "%6.32f\t", Sx[i][j]);
         }
         fprintf(Output, "\n");
      }
      fclose(Output);
      printf("Results from Cyclostationary FAM algorithm written to:
%s\n", Output_file);

   printf("\nEnd of FAM Program Execution\n");
   return 0;
}
```

APPENDIX D. DCP.MC

```
// ---------------------------------
// FILE      : dcp.c
// description: File resides on the MAP's primary Chip
//
// Modified by: LT Wesley Simon
// For            : thesis
// Date      : Septemeber 2009
// ---------------------------------

#include <libmap.h>
#include "FAM_const.c"
#define n 64

void PrimaryChip (double xx[],double ham[], double Ifinal[], double
Qfinal[],int64_t *t_channelize, int64_t *t_fft1, int64_t *t_fft2,
int64_t *t_downconvert,int64_t *t_DMA, int mapno) {

/* Memory Management Plan */

      OBM_BANK_C_2D (IXE, double, P, Np)
      OBM_BANK_D_2D (QXE, double, P, Np)
      OBM_BANK_A_2D (finalI, double, P, Np*Np)
      OBM_BANK_B_2D (finalQ, double, P, Np*Np)
      OBM_BANK_E    (al, double, NN  ) //input data
      OBM_BANK_F    (HAM,double,Np) //hamming window data


   /* Declare Other Variables */
      int64_t i, j, k,r,a, nbytes, index, k2, s, log2n;
      float hamming;
      int L = Np/4;

   /* Timing Variables*/
      int64_t Tstart, TafterC,Tfft1start,Tfft1done,fft2,dc;
      int64_t Tdma,dmastart,dmastop,dmain, dmaout;
   /*Declare channelize variables */
      float X[Np][P];//input data stored here
      float XW[Np][P];//post hamming window

   //FFT1 Variables
      float I[n],Q[n];
      float a1_copy[(int)Np], a2_copy[(int)Np];
      float aI[(int)Np], aQ[(int)Np];
      float temp_I_out[(int)Np],temp_Q_out[(int)Np];
      float FFTI_out[(int)Np][(int)P],FFTQ_out[(int)Np][(int)P];
      int m,o,z;
      unsigned int ii, p, q;
      float wm1, wm2, w1, w2, t0, t1, t2, u1, u2,u1a,u2a;

   //downconvert variables
      float Ii, Qi, Ij, Qj;  //temporary I and Q scratch variables
      float IXF1[(int)Np][(int)P], QXF1[(int)Np][(int)P];
      float twdlI[(int)Np][(int)P], twdlQ[(int)Np][(int)P];

   /* Start MAP timing */
      start_timer();
      read_timer(&Tstart);

   /* Transfer Data to MAP from CM */
      nbytes = NN * sizeof(double);
      read_timer(&dmastart);
      DMA_CPU(CM2OBM, al, MAP_OBM_stripe(1, "E"), xx, 1, nbytes, 0);
      wait_DMA (0);
      DMA_CPU(CM2OBM, HAM, MAP_OBM_stripe(1, "F"), ham, 1,
Np*sizeof(double), 0);
      wait_DMA (0);
      read_timer(&dmastop);
      dmain= dmastop-dmastart; //get dma time

   //build twiddle table
      for (r=0; r<Np; r++)
      {
         k = r - ((int)Np/2);

         for (j=0; j<P; j++)
         {
            twdlI[r][j] = cosf(-pi2*k*j*L/Np);
            twdlQ[r][j] = sinf(-pi2*k*j*L/Np);
         }
      }
   /* Channelize: Turn Array into Matrix */
      for (i=0; i<P; i++)
      {
         index = 0;
         for (j=i*L+1; j<=i*L+Np; j++)
         {
            X[index][i] = al[j-1];
            index++;
         }
      }

   /* The following loop was used to generate data for G. Upperman's
      Thesis.  Note: printf statement only works in debug mode. */
      printf("********** THESIS DATA: CHANNELIZATION *******\n");
      for (i=0; i<5; i++)
      {
         printf("\t");
         for (j=0; j<2; j++)
         {
            printf("%2.8f\t", X[i][j]);
         }
         printf("\n");
      }

   /* Apply Hamming window */
      for (j=0; j<P; j++)
```

```
         {
            XW[i][j] = HAM[i]*X[i][j];
         }
      } // for i

   /* Get Time Hac */
      read_timer(&TafterC);//get timing post channelize
      *t_channelize = TafterC - Tstart;


//**************************FIRST FFT********************************
/* Get Time Hac and calculate timing */
      read_timer(&Tfft1start);
/* Determine Log Base 2 */
      for (i=8*sizeof(int)-1; i>=0 && ((1<<i) & n)==0; i--);
      log2n = i;

//build input matrix
for (a=0; a< P; a++)
{
      /* Build FFT Input Matrix */
         for (j=0; j<Np; j++)
         {
          I[j] = XW[j][a];
          Q[j] = 0.0f;
} // for j

   /* reorder input and split input into real and complex parts */
      for (i=0; i<n; i++)
      {
         /* reverse bits 0 thru k-1 in the integer "a" */
            for (ii=o=0, p = 1, q = 1<<(log2n-1);
               ii<log2n;
               ii++, p <<= 1, q >>= 1 ) if (i & q) o = o | p;

         j = (int)o;
         aI[j] = I[i];
         aQ[j] = Q[i];
       }

/* loop on FFT stages */
      for (s=1; s<=log2n; s++)
      {
         m = 1<<s;  /* m = 2^s */
         t0  = pi2/m;
         wm1 = cosf(t0);   /* wm = exp(q*2*pi*i/m); */
         wm2 = sinf(t0);
         w1 = 1.0f;
         w2 = 0.0f;

         for (j=0; j<m/2; j++)
         {
            for (k=j; k<n; k+=m)
            {
               /* t = w*a[k+m/2]; */
               u1a = aI[k];
               u2a = aQ[k];
               k2 = k+ m/2;
               u1 = aI[k2];
               u2 = aQ[k2];
               t1  = w1 * u1 - w2 * u2;
               t2 =  w1 * u2 + w2 * u1;
               aI[k] = u1a + t1;
               aQ[k] = u2a + t2;
               aI[k2] = u1a - t1;
               aQ[k2] = u2a - t2;
            } // for k

         /* w = w * wm; */
         t1 = w1 * wm1  -  w2 * wm2 ;
         w2 = w1 * wm2  +  w2 * wm1 ;
         w1 = t1;
      } // for j
   } // for s

   /* flip the final stage */
      temp_I_out[0] = aI[0];
      temp_I_out[(int)n/2] = aI[(int)n/2];
      temp_Q_out[0] = aQ[0];
      temp_Q_out[(int)n/2] = aQ[(int)n/2];


#pragma src parallel sections
   {
#pragma src section
      {
         for (r=1; r<n/2; r++)  {temp_I_out[r] = aI[n-r];}
         for (r=1; r<n/2; r++)  {temp_I_out[n-r] = aI[r];}
      }
#pragma src section
      {
         for (j=1; j<n/2; j++)  {temp_Q_out[j] = aQ[n-j];}
         for (j=1; j<n/2; j++)  {temp_Q_out[n-j] = aQ[j];}
      }
   }

   /* Take time hac */
      read_timer(&Tfft1done);
for (j=0; j<Np; j++)
         {
            FFTI_out[j][a] = temp_I_out[j];
            FFTQ_out[j][a] = temp_Q_out[j];
            }
}   // for i

*t_fft1= Tfft1done-Tfft1start;

  /* Implement FFT shift: End Result swaps the top and bottom halves:
*/
      for(i=0; i<(Np/2); i++)
      {
         for(j=0; j<P; j++)
         {
            /* Bottom real half becomes top real half */
               IXF1[i][j] = FFTI_out[i+(int)(Np/2)][j];
            /* Top real half becomes bottom real half */
               IXF1[i+(int)(Np/2)][j] = FFTI_out[i][j];

            /* Bottom imag half becomes top imag half */
               QXF1[i][j] = FFTQ_out[i+(int)(Np/2)][j];
            /* Top imag half becomes bottom imag half */
               QXF1[i+(int)(Np/2)][j] = FFTQ_out[i][j];
         } // for j
      } // for i

   /* The following nested loop was used to generate data for G.
      Upperman's Thesis.  Note: printf only works in debug mode */
      printf("********* THESIS DATA: FFT 1 AND SHIFT *****\n");
      for (i=0; i<5; i++)
      {
         printf("\t%2.8f+i*%2.8f\n", IXF1[i][0], QXF1[i][0]);
      }

   /* Downconversion - the short sliding FFT's results are shifted
      to baseband to obtain decimated complex demodulate sequences.
      The transpose of the matrix is taken at the same time. */

      for(i=0; i<Np; i++)
      {

      for(j=0; j<P; j++)
         {
            Ii = twdlI[i][j];
            Qi = twdlQ[i][j];
            IXE[j][i] = (IXF1[i][j] * Ii) - (QXF1[i][j] * Qi);
            //IXE_copy[j][i]=IXE[j][i]; //identical copy for
multiplicaiton step
            QXE[j][i] = (IXF1[i][j] * Qi) + (QXF1[i][j] * Ii);
            //QXE_copy[j][i]=QXE[j][i]; //identical copy for
multiplication step
         }
      }


//Done with first FFT and Channelize give to Downconvert and FFT2
//Synchronize with other chip to make sure its ready
   send_to_bridge_a ();
// give banks B and C to the other chip
   send_perms (OBM_A|OBM_B|OBM_C|OBM_D);
//second sync point
send_to_bridge_a ();
//final sync and recieve timing information
recv_from_bridge_a (&fft2,&dc);
//DONE WORKING ON OTHER CHIP
*t_fft2=fft2;
```

```
*t_downconvert=dc;

//Remove memory permissions on secondary chip
send_perms (0);
//timing marker
read_timer(&dmastart);

// DMA the results back to CPU
    DMA_CPU (OBM2CM, finalI, MAP_OBM_stripe(1,"A"), Ifinal, 1,
P*Np*Np*sizeof(double), 0);
    wait_DMA (0);

    DMA_CPU (OBM2CM, finalQ, MAP_OBM_stripe(1,"B"), Qfinal, 1,
P*Np*Np*sizeof(double), 0);
    wait_DMA (0);
read_timer(&dmastop);
dmaout= dmastop-dmastart;
Tdma=dmain+dmaout;
*t_DMA=Tdma;
}
```

APPENDIX E.   DCS.M

```
// ---------------------------------
// FILE      : dcs.c
// description: File resides on the MAP's Secondary Chip
//
// Modified by: LT Wesley Simon
// For             : thesis
// Date      : Septemeber 2009
// ---------------------------------

#include <libmap.h>
#include "FAM_const.c"
#define n 8 //size of FFT 2

void SecondaryChip ()
{

   /* Declare Arrays in OBM */
      OBM_BANK_C_2D (IXE, double, P, Np)
      OBM_BANK_D_2D (QXE, double, P, Np)
      OBM_BANK_A_2D (finalI, double, P, Np*Np)
      OBM_BANK_B_2D (finalQ, double, P, Np*Np)
      OBM_BANK_E    (al, double, NN  ) //input data

   /* Declare Other Variables */
      //timing variables
      int64_t scstart,tdcstop,tfft2stop, fft2,dc;
      int64_t nbytes, i, j, k, k2, s, log2n;
      unsigned int ii, p, q;
      int ma,mb,mc, o,r,a,row,col,d,index;
      float wmA1, wmA2, wA1, wA2, tA0, tA1, tA2, uA1, uA2, uA1a, uA2a,
na1, na2;
      float wmB1, wmB2, wB1, wB2, tB0, tB1, tB2, uB1, uB2,uB1a,uB2a;
      float wmC1, wmC2, wC1, wC2, tC0, tC1, tC2, uC1, uC2,uC1a,uC2a;
      float a1_copy[(int)Np], a2_copy[(int)Np];
      float Ia_copy[(int)Np], Qa_copy[(int)Np];
      float Ib_copy[(int)Np], Qb_copy[(int)Np];
      float Ia[(int)Np], Qa[(int)Np];
      float Ib[(int)Np], Qb[(int)Np];
      float Ibeven[(int)Np], Qbeven[(int)Np];
      float Ibodd[(int)Np], Qbodd[(int)Np];
      float Iceven[(int)Np], Qceven[(int)Np];
      float Icodd[(int)Np], Qcodd[(int)Np];
      float Ic[(int)Np], Qc[(int)Np];
      float Iaeven[(int)Np], Iaodd[(int)Np];
      float Qaeven[(int)Np], Qaodd[(int)Np];

/*SINE and COSINE array variables */
      float SIN_Array[4];//set to log2n +1
      float COS_Array[4];

/* Declare downconvert Variables */
```

```
      int L = Np/4;
      float Ii, Qi, Ij, Qj;  //temporary I and Q scratch variables
      float IXF1[(int)Np][(int)P], QXF1[(int)Np][(int)P];
      float  IXE_bram[(int)P][(int)Np],  QXE_bram[(int)P][(int)Np];
      float  IXE_copy[(int)P][(int)Np],  QXE_copy[(int)P][(int)Np];
      float  IXM[(int)P][(int)Np*Np],  QXM[(int)P][(int)Np*Np];
      float I[n],Q[n];
      float a1[(int)Np], a2[(int)Np];
      float temp_I_out[(int)n],temp_Q_out[(int)n];
      recv_from_bridge_a ();
      recv_perms ();

  //initalize final answer arrays to all zeros
   for (row=0; row<P;row++){
   for (col=0; col<Np*Np; col++){
      finalI[row][col]=0;
      finalQ[row][col]=0;
      }}
    recv_from_bridge_a ();

 //build sin_cos array  - used to save sine and cosine resources
for (index =1; index<=4; index++){
   SIN_Array[index]=sinf(pi2/(1<<index));
   COS_Array[index]=cosf(pi2/(1<<index));
}

//Start downconvert
//*************DOWNCONVERT- Continued****************************
  //make second copy of IXE and Qxe to remove read write dependancies
in multiplication
      for(i=0; i<Np; i++)
      {

      for(j=0; j<P; j++)
         {
            IXE_bram[j][i]= IXE[j][i];
            IXE_copy[j][i]= IXE[j][i]; //identical copy for
multiplicaiton step
            QXE_bram[j][i]= QXE[j][i];
            QXE_copy[j][i]= QXE[j][i]; //identical copy for
multiplication step
         }
      }


  /* Multiplication - the product sequences between each one of the
     complex demodulates and the complex conjugate of the others
     are formed.  This forms the area in the bi-frequency plane. */
     for (i=0; i<Np; i++)
     {
        for (j=0; j<Np; j++)
        {
           for (k=0; k<P; k++)
           {
              Ii = IXE_bram[k][i];
```

```
                Qi = QXE_bram[k][i];
                Ij = IXE_copy[k][j];
                Qj = QXE_copy[k][j];
                IXM[k][i*(int)Np+j] =  (Ii * Ij) + (Qi * Qj);
                QXM[k][i*(int)Np+j] = -(Ii * Qj) + (Qi * Ij);
             } // for k
          } // for j
       } // for i


     /* Get last time hac and report */
       read_timer(&tdcstop);

      /* Determine Log Base 2 */
        for (i=8*sizeof(int)-1; i>=0 && ((1<<i) & n)==0; i--);
        log2n = i;

     //build input matrix
  printf("Iteration of FFT");
     for (a=0; a< Np*Np; a++) // col
     {
  printf ("\r%d",a);
        /* Build FFT Input matrix */
           for (j=0; j<P; j++)
           {
              I[j] = IXM[j][a];
              Q[j] = QXM[j][a];
           } // for j

  /* reorder input and split input into real and complex parts */
        for (i=0; i<n; i++)
        {
           /* reverse bits 0 thru k-1 in the integer "a" */
              for (ii=o=0, p = 1, q = 1<<(log2n-1);
                 ii<log2n;
                 ii++, p <<= 1, q >>= 1 ) if (i & q) o = o | p;

           j = (int)o;
           a1[j] = I[i];
           a2[j] = Q[i];
           a1_copy[j] = I[i];//make copy of data to reduce multiple reads
           a2_copy[j] = Q[i];//make copy of data to reduce multiple reads
        }

  //        printf("\tFirst stage");
     /* loop on FFT stages */
           //ma = 1<<1;  /* m = 2^s */
           ma=2;
           wmA1 = COS_Array[1];   /* wm = exp(q*2*pi*i/m); */
           wmA2 = SIN_Array[1];
           wA1 = 1.0f;
           wA2 = 0.0f;

           for (j=0; j<ma/2; j++)
           {
              for (k=j; k<n; k+=ma)
              {
                 /* t = w*a[k+m/2]; */
                 k2 = k+ ma/2;

                 uA1a = a1_copy[k2];
                 uA2a = a2_copy[k2];
                 tA1  = wA1 * uA1a - wA2 * uA2a;
                 tA2 =  wA1 * uA2a + wA2 * uA1a;
                 uA1 = a1[k];
                 uA2 = a2[k];
                 Iaeven[k] = uA1 + tA1;
                 Qaeven[k] = uA2 + tA2;
                 Iaodd[k2] = uA1 - tA1;
                 Qaodd[k2] = uA2 - tA2;

              } // for k

           /* w = w * wm; */
           na1 = wA1 * wmA1  -  wA2 * wmA2 ;
           na2 = wA1 * wmA2  +  wA2 * wmA1 ;
           wA1 = na1;
           wA2 = na2;

        } // for j
           Ia[0]=Iaeven[0];
           Ia[1]=Iaodd[1];
           Ia[2]=Iaeven[2];
           Ia[3]=Iaodd[3];
           Ia[4]=Iaeven[4];
           Ia[5]=Iaodd[5];
           Ia[6]=Iaeven[6];
           Ia[7]=Iaodd[7];
           Qa[0]=Qaeven[0];
           Qa[1]=Qaodd[1];
           Qa[2]=Qaeven[2];
           Qa[3]=Qaodd[3];
           Qa[4]=Qaeven[4];
           Qa[5]=Qaodd[5];
           Qa[6]=Qaeven[6];
           Qa[7]=Qaodd[7];

  for (k=0; k<n;k++){
        Ia_copy[k]=Ia[k];
        Qa_copy[k]=Qa[k];
  }


  //*************S=2*****************************

           //mb = 1<<2;  /* m = 2^s */
           mb=4;
           wmB1 = COS_Array[2];   /* wm = exp(q*2*pi*i/m); */
           wmB2 = SIN_Array[2];
           wB1 = 1.0f;
```

```
        wB2 = 0.0f;
        for (j=0; j<mb/2; j++)
        {
            for (k=j; k<n; k+=mb)
           {

              /* t = w*a[k+m/2]; */
              k2 = k+ mb/2;
              //fetch data needed
              uB1a = Ia_copy[k2];
              uB2a = Qa_copy[k2];
              tB1  = wB1 * uB1a - wB2 * uB2a;
              tB2 =  wB1 * uB2a + wB2 * uB1a;
              uB1 = Ia[k];
              uB2 = Qa[k];
              Ibeven[k] = uB1 + tB1;
              Qbeven[k] = uB2 + tB2;
              Ibodd[k2] = uB1 - tB1;
              Qbodd[k2] = uB2 - tB2;
           } // for k
        /* w = w * wm; */
        tB1 = wB1 * wmB1  -  wB2 * wmB2 ;
        wB2 = wB1 * wmB2  +  wB2 * wmB1 ;
        wB1 = tB1;
      } // for j

        Ib[0]=Ibeven[0];
        Ib[1]=Ibeven[1];
        Ib[2]=Ibodd[2];
        Ib[3]=Ibodd[3];
        Ib[4]=Ibeven[4];
        Ib[5]=Ibeven[5];
        Ib[6]=Ibodd[6];
        Ib[7]=Ibodd[7];
        Qb[0]=Qbeven[0];
        Qb[1]=Qbeven[1];
        Qb[2]=Qbodd[2];
        Qb[3]=Qbodd[3];
        Qb[4]=Qbeven[4];
        Qb[5]=Qbeven[5];
        Qb[6]=Qbodd[6];
        Qb[7]=Qbodd[7];
for (k=0; k<n;k++){
      Ib_copy[k]=Ib[k];
      Qb_copy[k]=Qb[k];
}

//************S=3*****************************
//     printf("...Third stage\n");
        //mc = 1<<3;  /* m = 2^s */
        mc=8;
        wmC1 = COS_Array[3];   /* wm = exp(q*2*pi*i/m); */
        wmC2 = SIN_Array[3];
        wC1= 1.0f;
        wC2= 0.0f;
```

```
         for (j=0; j<mc/2; j++)
         {
            for (k=j; k<n; k+=mc)
            {
               /* t = w*a[k+m/2]; */
               k2 = k+ mc/2;
               uC1a = Ib_copy[k2];
               uC2a = Qb_copy[k2];
               tC1  = wC1 * uC1a - wC2 * uC2a;
               tC2 =  wC1 * uC2a + wC2 * uC1a;
               uC1 = Ib[k];
               uC2 = Qb[k];
               Iceven[k] = uC1 + tC1;
               Qceven[k] = uC2 + tC2;
               Icodd[k2] = uC1 - tC1;
               Qcodd[k2] = uC2 - tC2;
            } // for k


         /* w = w * wm; */
         tC1 = wC1 * wmC1  -  wC2 * wmC2 ;
         wC2 = wC1 * wmC2  +  wC2 * wmC1 ;
         wC1 = tC1;
      } // for j
         Ic[0]=Iceven[0];
         Ic[1]=Iceven[1];
         Ic[2]=Iceven[2];
         Ic[3]=Iceven[3];
         Ic[4]=Icodd[4];
         Ic[5]=Icodd[5];
         Ic[6]=Icodd[6];
         Ic[7]=Icodd[7];
         Qc[0]=Qceven[0];
         Qc[1]=Qceven[1];
         Qc[2]=Qceven[2];
         Qc[3]=Qceven[3];
         Qc[4]=Qcodd[4];
         Qc[5]=Qcodd[5];
         Qc[6]=Qcodd[6];
         Qc[7]=Qcodd[7];

//END STAGES

   /* flip the final stage */
      temp_I_out[0] = Ic[0];
      temp_I_out[(int)n/2] = Ic[(int)n/2];
      temp_Q_out[0] = Qc[0];
      temp_Q_out[(int)n/2] = Qc[(int)n/2];


#pragma src parallel sections
   {
#pragma src section
      {
         for (r=1; r<n/2; r++)  {temp_I_out[r] = Ic[n-r];}
         for (r=1; r<n/2; r++)  {temp_I_out[n-r] = Ic[r];}
      }
#pragma src section
      {
         for (r=1; r<n/2; r++)  {temp_Q_out[r] = Qc[n-r];}
         for (r=1; r<n/2; r++)  {temp_Q_out[n-r] = Qc[r];}
      }
   }


 for (r=0; r<P; r++)
         {
            finalI[r][a] = temp_I_out[r];
            finalQ[r][a] = temp_Q_out[r];
         }

}//for a
/* Get time hac */
     read_timer(&tfft2stop);
fft2=tfft2stop-tdcstop;
dc=tdcstop-scstart;
send_to_bridge_a (fft2,dc);
recv_perms();
printf("\ndone with program returning to primary\n");
      }
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F. MISC CODE

A. HIGH PRECISION TIMING

This file creates a structure used to keep track of timing.

```
int timeval_subtract (struct timeval *result, struct timeval *x, struct
timeval *y);

/* Subtract the `struct timeval' values X and Y,
        storing the result in RESULT.
        Return 1 if the difference is negative, otherwise 0.  */

     int
     timeval_subtract (result, x, y)
          struct timeval *result, *x, *y;
     {
       /* Perform the carry for the later subtraction by updating y. */
       if (x->tv_usec < y->tv_usec) {
         int nsec = (y->tv_usec - x->tv_usec) / 1000000 + 1;
         y->tv_usec -= 1000000 * nsec;
         y->tv_sec += nsec;
       }
       if (x->tv_usec - y->tv_usec > 1000000) {
         int nsec = (x->tv_usec - y->tv_usec) / 1000000;
         y->tv_usec += 1000000 * nsec;
         y->tv_sec -= nsec;
       }

       /* Compute the time remaining to wait.
          tv_usec is certainly positive. */
       result->tv_sec = x->tv_sec - y->tv_sec;
       result->tv_usec = x->tv_usec - y->tv_usec;

       /* Return 1 if result is negative. */
       return x->tv_sec < y->tv_sec;
     }
```

B. GETFAM_CONSTANT.C

This file is utilized to create a file of constants that are common to both the .mc files. Compile and execute this file to generate the output file FAM_const.c.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

/*getFAM_const.c
original work by Gary Upperman
Edited by:  LT Wesley Simon
```

```
Added pi2 variable and changed constants from doubles to float by
adding 'f' to pi and pi2
*/

FILE *FAM_const;

main()
{
   /* DECLARE VARIABLES AND CONSTANTS */
      /* declare file names and path */
         char const_file[] = "FAM_const.c";
      /* declare sample frequency, frequency resolution, and M */
         int fs, df, M;
      /* declare FAM-spcific varibles */
         double dalpha, Np, L, P, N, NN;
   /* Get data from user */
      printf("What is the sampling frequency (fs)(Hz)? ");
      scanf("%d", &fs);
      printf("What is the frequency resolution desired (df)(Hz)? ");
      scanf("%d", &df);
      printf("What is M? ");
      scanf("%d", &M);
   /* Calculate Values */
      dalpha = df/M;
      Np = pow(2.0, ceil(log10(fs/df)/log10(2)) );
      L = Np/4;
      P = pow(2.0, ceil(log10(fs/dalpha/L)/log10(2)) );
      N = P*L;
      NN = (P-1)*L+Np;
   /* PRINT CONSTANT FILE */
      FAM_const=fopen("FAM_const.c", "w");
      if(FAM_const == NULL)
      {
         printf("Error creating FAM constant file.");
         return(1);
      }
      fprintf(FAM_const, "#define N %i\n", (int)N);
      fprintf(FAM_const, "#define NN %i\n", (int)NN);
      fprintf(FAM_const, "#define Np %i\n", (int)Np);
      fprintf(FAM_const, "#define P %i\n", (int)P);
      fprintf(FAM_const, "#define fs %i\n", (int)fs);
      fprintf(FAM_const, "#define df %i\n", (int)df);
      fprintf(FAM_const, "#define M %i\n", (int)M);
      fprintf(FAM_const, "\n#define pi 3.14159265358979f\n");
      fprintf(FAM_const, "\n#define pi2 6.28318530717959f\n");

      fclose(FAM_const);

   /* END FUNCTION */
      printf("\nN: %i, NN: %i, Np: %i, P: %i\n",
         (int)N, (int)NN, (int)Np, (int)P);
      printf("Constant File written for Cyclostationary FAM
Analysis.\n\n");
      return 0;
}
```

LIST OF REFERENCES

[1] P. E. Pace, Detecting and Classifying Low Probability of Intercept Radar, 2nd ed. Boston, Ma: Artech House, 2009.

[2] G. J. Upperman, "Implementation of a Cyclostationary Spectral Analysis Algorithm on an SRC Reconfigurable Computer For Real-Time Signal Processing," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2008.

[3] T. L. Upperman, "ELINT Signal Processing Using Choi-Williams Distribution on Reconfigurable Computers for Detection and Classification of LPI Emitters," M.S. thesis, Electrical and Computer Eng, Naval Postgraduate School, Monterey, CA, 2008.

[4] D. A. Brown, "ELINT Signal Processing on Reconfigurable Computers for Detection and Classification of LPI Emitters," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2006.

[5] K. M. Stoffell, "Implementation of a Quadrature Mirrir Filter Bank on an SRC Reconfigurable Computer for Real-Time SIgnal Processing," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2006.

[6] K. R. Macklin, "Suitability of the SRC-6 Reconfigurable Computing System for Generating False Radar Images," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2004.

[7] K. R. Macklin, "Benchmarking and Analysis of the SRC-6 Reconfigurable Computing System," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2003.

[8] Radar Design Bureau, "'Vostok-E' Mobile Solid-State 2D Digital VHF Radar," http://kbradar.by/text/pages-view-37.html Accessed 23 JUL 2009.

[9] Jane's Information Group, "JNWS 29-May-2009*Surface-to-surface missies/ Sweden/RBS 15M/RBS 15 MK 3," Accessed 23 JUL 2009.

[10] SRC Computers, Inc, "SRC Carte C Programming Environment v2.2 Guide," August 21, 2006.

[11] SRC Computers, Inc. (2006) MAP Processors. PDF.

[12] Xilinx, "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Datasheet," Product Specification, November 2007.

[13] SRC Computers, Inc. (2005) SRC Training Course. PDF.

[14] D. E. Caliga (private communication), 2009.

[15] Xilinx, Inc. (2003) Development System Reference Guide. PDF.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, Ma: The MIT Press, 1990.

[17] SRC Computers, Inc. (2006) MAP Processors. PDF.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Douglas J. Fouts
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

5. Phillip E. Pace
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. John T. Butler
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

7. Dr. Peter Craig
   Office of Naval Research
   Washington, DC

8. Dr. Chip Grounds
   Office of Naval Research
   Washington, DC