CONTINUOUS REAL-TIME RECOVERY OF OPTICAL SPECTRAL FEATURES DISTORTED BY FAST-CHIRPED READOUT

by

Scott Henry Bekker

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Electrical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2006

APPROVAL

of a thesis submitted by

Scott Henry Bekker

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Dr. Ross K. Snider

Approved for the Department of Electrical and Computer Engineering

Dr. James Peterson

Approved for the Division of Graduate Education

Dr. Joseph J. Fedock

STATEMENT OF PERMISSION OF USE

Scott Henry Bekker

April 2006

TABLE OF CONENTS

TABLE OF CONTENTS - CONTINUED

LIST OF FIGURES

ABSTRACT

Optical signal processors can analyze the spectra of RF signals with bandwidths of hundreds of gigahertz and achieve spectral resolutions of tens of kilohertz, far exceeding the capabilities of conventional analyzers. Modulating a broadband RF signal onto a laser beam and exposing an optical memory material to the modulated light stores the power spectrum of the input signal in the material temporarily. The power spectrum contained within the material is then extracted by measuring the intensity of the light exiting the material while exposing it to a linear frequency chirped laser spanning the bandwidth of the input signal. The major benefit of this technique is that the readout bandwidth is much lower than the bandwidth of the input signal allowing conventional photodetectors and analog to digital converters to capture the readout.

A problem arises when reading out a large bandwidth in the time the signal's power spectrum remains in the material. This requires a fast chirp rate, but as the chirp rate increases, so does the distortion to the readout signal. A recently developed post-processing technique removes this distortion, allowing one to obtain high-resolution spectral information with extremely fast chirp rates. The focus of this work is the development of a practical post-processing system capable of removing the readout distortion continuously and in real-time.

The original spectral recovery algorithm requires the entire readout sequence prior to processing, thus making it unsuitable for continuous real-time recovery. Therefore, the algorithm was modified to use only part of the readout signal allowing recovery during readout. Although the new algorithm exploits the computational efficiency of a fast Fourier transform, real-time recovery processing presents a computational load exceeding the capabilities of conventional programmable digital signal processors. For this reason, a field programmable gate array (FPGA) was used to meet the processing requirements by means of its parallel processing capability. The FPGA based post-processor recovers spectral features as narrow as 28 kHz read out with a chirp rate of 1 MHz/μs while maintaining a processing latency less than 310 μs. This work provides, for the first time, continuous real-time spectral feature recovery.

INTRODUCTION

Optical spectroscopy is the study of spectra in light. Applications of optical spectroscopy are broad and include identification of materials based on their chemical composition, the determination of velocity and composition of objects in space, high-performance radar processing [8], and broadband spectral analysis [9]. Spectral analysis with optics is of particular interest in commercial and military applications because of the ability of spatial spectral spectrum analyzers (S2SA) to process much higher bandwidths than can be done by conventional electronic (digital or analog) spectrum analyzers. An S2SA is so named because it processes and stores information at different spatial as well as different spectral locations in the optical memory material. Spectrum analyzers based on S2 technology can directly (without the need for down conversion) process signals with bandwidths of hundreds of gigahertz and simultaneously achieve fine spectral resolution of tens of kilohertz. Another benefit of using S2 based spectrum analyzers in the analysis of signals for communication and spectral surveillance is the absolute certainty that it will detect a frequency-hopping communication signal. For example, a conventional spectrum analyzer performing a frequency sweep may miss the frequency-hopping signal if the sweep does not coincide with the signal in frequency and time. The S2SA, however, will always detect the signal regardless of when it occurs because it is continuously processing and storing the power spectrum of the signal over its entire bandwidth. Known as unity-probability of intercept, this ensures the detection of brief narrowband signals.

The specific target application of this work is the detection of open communication channels while networking in extreme RF environments. The goal is to quickly identify clear channels in a congested RF spectrum and use these open channels to communicate. Given the broadband fine resolution processing and the unity probability of intercept, S2 spectrum analyzers are ideal for this application.

Spectral analysis with an S2SA uses S2 optical memory materials capable of processing and temporarily storing broadband information from a signal. Irradiating an S2 material with a single frequency laser excites atoms and molecules in the material whose resonant frequency corresponds to the laser's frequency. The material absorbs some of the energy from the signal and the rest passes through the material. The more the material is exposed to light at a particular frequency, the less it is able to absorb. Thus by measuring the intensity of the light exiting the S2 material, one can determine the extent that light has irradiated the material at that frequency in the past.

Exposing the material to a complicated broadband signal produces the same basic effect. Now, however, the spectral information of the entire bandwidth is stored rather than simply that from a single frequency. If light with a single frequency now passes through the material, one can determine the amount of energy present in the original signal at this frequency based on the intensity of the light exiting the crystal.

For spectral analysis, one is generally interested in the energy present across a signal's entire bandwidth rather than that of just a single frequency. Therefore, after irradiating the material with the analog input signal, the entire bandwidth is read out by exposing the material to a laser whose frequency is changed over time (linear frequency chirp) while measuring the intensity of the light exiting the material [1]. The chirp is a

signal whose frequency changes linearly with time. This chirp phase modulates the laser beam performing the readout. The frequency chirp covers the entire frequency band of interest over a given time producing a time domain map of the spectral information of the original signal.

Using S2 materials for spectral analysis as described above poses several technical challenges that require overcoming prior to development of a practical S2SA.

First, achieving fine spectral resolution, of tens of kilohertz, requires the use of a laser with a spectral linewidth narrower than that of the required resolution. Most laser systems have natural linewidths greater than a megahertz. Thus, a control system capable of maintaining a narrow laser linewidth is often necessary for applications needing fine spectral resolution. The good news is that these systems exist and active research is improving them [11, 12].

Another practical difficulty is keeping the S2 material cold. To achieve the spectral feature resolution noted above, one must cool the S2 material to cryogenic temperatures. This poses a problem when moving out of the laboratory. Liquid helium tanks are bulky and need frequent refilling, while phase change cryogenic coolers introduce vibration that must be isolated from the S2 material. Recent progress in vibration damping in closed cycle cryogenic coolers is alleviating this problem [4].

Another technical challenge is the development of a broadband linear RF chirp generator. As mentioned above, reading the spectral information out of the S2 material requires a linear chirp spanning the entire bandwidth of the RF input signal. Only a handful of devices exist that are able to generate these broadband linear chirps. A nonlinear chirp will produce a distorted temporal map of the spectrum. A portion of the

work performed for this thesis is the development of such a chirp generator. This is discussed in appendix A.

Finally, performing chirped readout with a fast chirp rate introduces quadratic phase distortion into the temporal map of the spectrum. Specifically, the phase of the readout signal is the true phase plus a phase that increases as a quadratic function of readout signal frequency. A fast chirp rate is desired to enable the readout of a large bandwidth during the time that the atoms and molecules remain in their excited state, known as the persistence time. Obtaining an accurate map of the spectrum requires the removal of this distortion.

A technique, developed by Chang et al [3], recovers spectral features from a temporal map containing quadratic phase distortion. Chang's recovery process takes a Fourier transform of the entire temporal readout map, removes the quadratic phase distortion in the frequency domain by multiplying the transformed signal by a phase compensation factor, and then inverse Fourier transforming the product producing the result, which is free of quadratic phase distortion. Spectral recovery is a new technique and this thesis is the result of developing a practical implementation of it that is suitable for S2 based spectrum analyzers.

In terms of the target application, finding open channels in a congested RF environment and subsequently using these channels for communication requires a post processor capable of continuously recovering a readout signal with low latency and real-time processing throughput.

The solution described in the rest of this thesis adapts Chang's recovery algorithm for continuous processing by creating a time domain filter that is functionally equivalent

to the original recovery algorithm. During the readout process, dedicated hardware convolves the recovery filter with the distorted temporal map. Filter duration and data rate requirements imposed by the S2 material physics in conjunction with spectral resolution, chirp rate, and processing latency goals make recovery a computationally intensive task. The computational load is so great that current digital signal processing (DSP) hardware is incapable of directly convolving the recovery filter with the readout signal. Therefore, the post processor uses block convolution with a fast Fourier transform (FFT) algorithm to perform the convolution much more efficiently. Even with this increased efficiency, the computational load is still too demanding for conventional programmable digital signal processors (PDSP). Thus a programmable logic device, known as a field programmable gate array (FPGA), performs the required post processing because of its greater DSP performance. The FPGA computes many of the processing steps in parallel allowing it to meet the computational requirements of this application.

The recovery processor meets the goals set forth by the Defense Advanced Research Projects Agency (DARPA) funding this work. Specifically, it achieves real-time recovery of spectral features as narrow as 28 kHz that are read out at 1 MHz/μs with a processing latency of 310 μs, which is well within the 1 ms stated goal. This chirp rate is over 1000 times faster than a chirp rate that would not need post processing to achieve that same resolution.

SPECTRAL ANALYSIS WITH S2 MATERIALS

As noted above, optical signal processing based upon S2 optical memory materials allows for ultra broadband RF spectral analysis. The technique uses the S2 material as a temporal impedance matching system. First, the S2 material captures and stores the spectrum of the broadband signal as an absorption profile in the material. Next, a frequency chirped readout laser probes the S2 material producing a temporal map of the input signal's spectrum. This readout signal is relatively low bandwidth allowing a conventional low bandwidth (high dynamic range) analog to digital converter (ADC) to capture it. When using the S2 material in this way, the readout chirp rate and spectral feature resolution, rather than the bandwidth of the analog input signal, determine the bandwidth of the signal at the photo-detector

Thulium doped yttrium aluminum garnet (Tm:YAG) is the S2 material used for this work. When cooled to 4° K, it can capture the power spectrum of a signal with a bandwidth of 30 GHz with a resolution of 25 kHz and 40 dB of dynamic range [9]. This capability makes Tm:YAG an appropriate material for use in broadband spectral analysis. Newer materials, such as thulium doped lithium niobate, are also being researched which have greater than 300 GHz of processing bandwidth and sub-500 kHz resolution at 4° K.

Figure 1 shows a detailed schematic for an S2 material based spectrum analyzer. The analog signal being analyzed, shown in the upper left corner of the figure, is phase modulated onto a frequency-stabilized laser source using a broadband electro-optic phase

modulator (EOPM). The modulated optical signal irradiates the S2 material storing the power spectrum of the analog input signal as an absorption profile in the material.

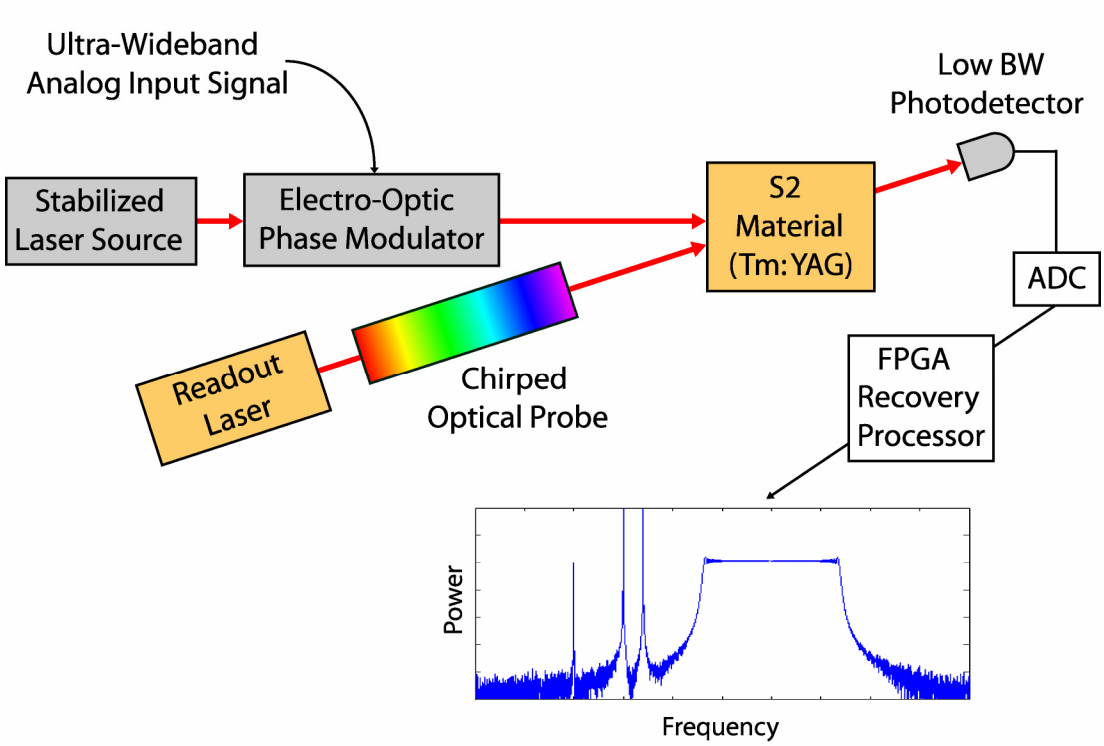

Figure 1: Spatial-Spectral Spectrum Analyzer block diagram

After storage, a readout laser probes the S2 material with a linear frequency chirped optical field over the frequency band of the analog input signal. In essence, when scanning the frequency, the light beam experiences different absorption based upon the stored power spectrum. Capturing the intensity of the light exiting the S2 material with a low-bandwidth photo detector and ADC during the frequency scan produces a temporal map of the analog signal's spectrum as shown in Figure 2.

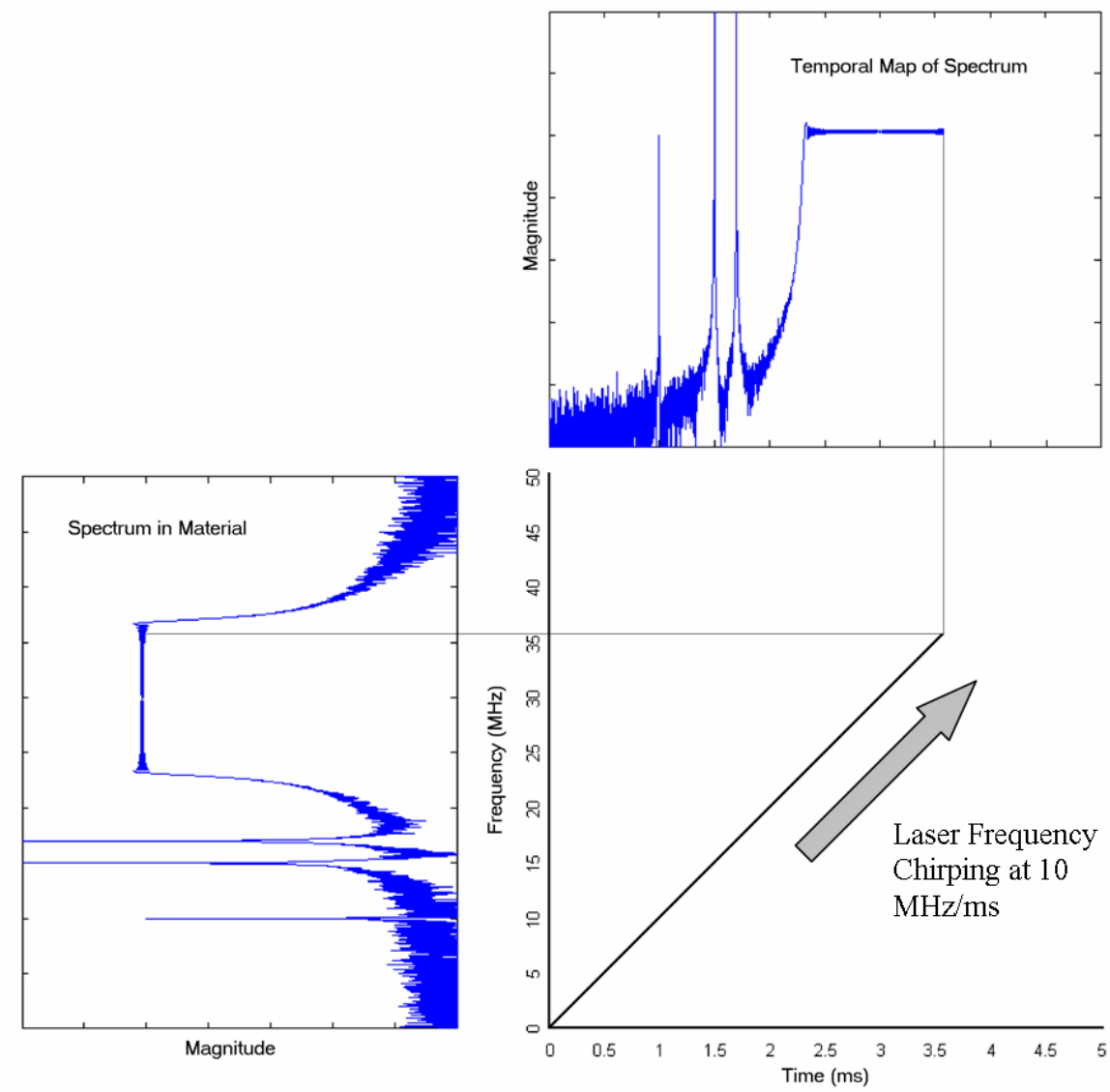

Figure 2: Chirped readout creates a temporal map of the spectrum.

Chirped Readout Distortion

Fast readout chirps are desirable to enable the chirp to scan a wide bandwidth during the persistence time of the S2 material, although as reported by Chang et al [1], there is a problem with fast-chirped readout. In particular, when scanning over a

featureless frequency band, the material absorbs most of the optical energy from the chirp. When the readout chirp encounters a spectral feature, however, most atoms at that frequency are already in the excited state and thus they emit, rather than absorb, most of the energy from the chirp. This optical energy emitted from the S2 material as the atoms fall from their excited state is called free induction decay. If the chirp rate is slow relative to the square of the feature width, then when the readout chirp scans over the feature, the light from the free induction decay exits the material with minimal interference with the chirp. Conversely, if the chirp rate is fast relative to the square of the feature width, the light from the free induction decay interferes significantly with the chirp sweeping up in frequency. By the time the free induction decay is finished, the frequency of the readout chirp is much higher than the frequency of the light from the free induction decay. The photo-detector picks up the beat note caused by the interference between the two light sources. Because the light from the free induction decay is decaying and the chirp is increasing in frequency, the beat note caused by chirping too fast over a narrow feature is an increasing frequency chirp whose envelope decays with time.

This effect can be seen in Figure 3 [1]. Here a spectral-hole in the S2 material is depicted in the lower window and temporal maps of that spectral-hole read out at different chirp rates are shown in the upper window. As can be seen, increasing the chirp rate causes the temporal spectrum mapping to differ increasingly from the true spectrum in the crystal. To avoid this distortion, the readout laser must dwell on a feature for $1/\delta\nu$ seconds where $\delta\nu$ is the full width of the feature in hertz at half its maximum amplitude

(FWHM). Thus, the chirp must sweep less than δν Hz in 1/δν seconds to prevent distortion.

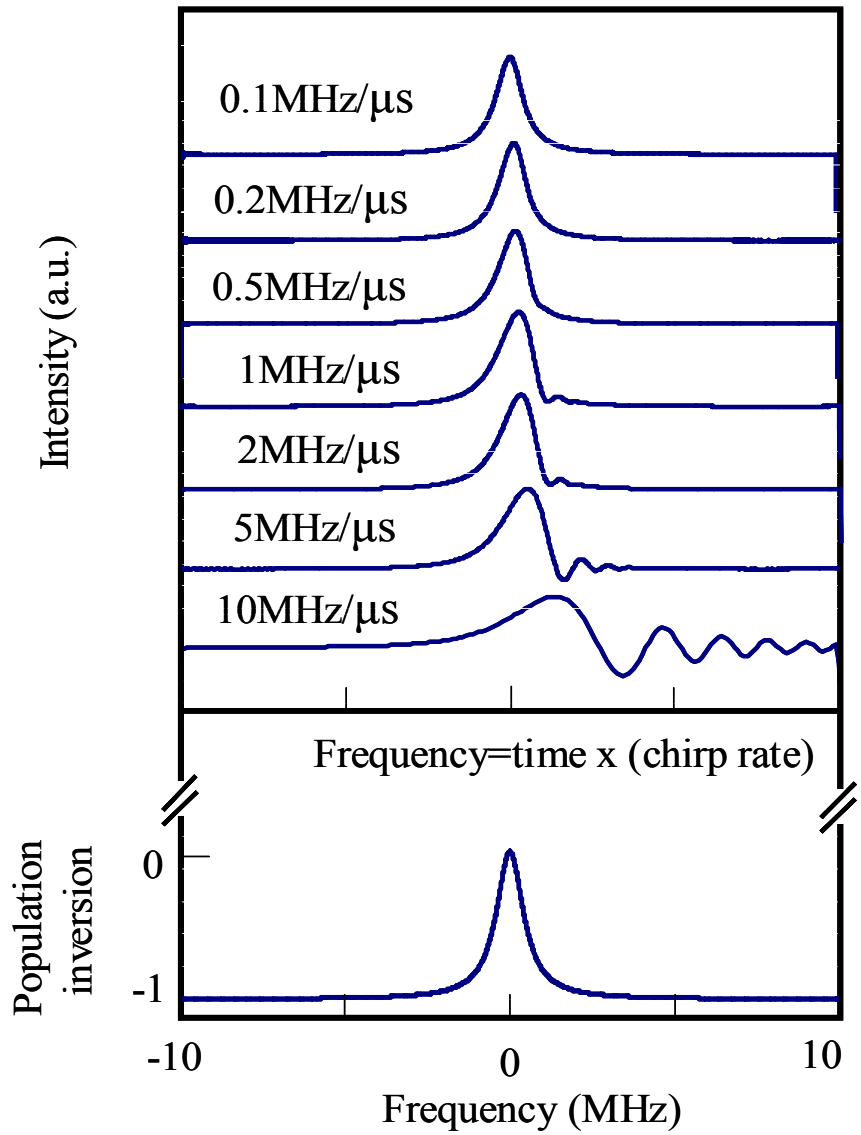

Figure 3: Simulation of chirped readout of a spectral hole. Full width half maximum (FWHM) Γ = 2 MHz. Note the distortion increase as the chirp rate is increased.

This imposes a maximum value on the chirp rate, κ, given by equation (1), known as the conventional limit. Using this order of magnitude approximation, the maximum chirp rate for the example in Figure 3 is $(2\ \text{MHz})^2 = 4\ \text{MHz}/\mu\text{s}$.

$$\kappa_{\max} = \frac{\delta\nu}{1/\delta\nu} = \delta\nu^2 \tag{1}$$

If we want to resolve features as narrow as 25 kHz, the goal for this application, the conventional limit on the chirp rate is $(25\text{ kHz})^2 = 0.625$ MHz/ms. Not only is this chirp rate restriction inconvenient from a latency perspective, it also limits the bandwidth that the chirp spans due to the persistence time of the S2 material. After the persistence time, the information stored in the material rapidly decays away. Chirping at 0.625 MHz/ms for the 10 ms persistence time of Tm:YAG, spans a bandwidth of only 6.25 MHz. Reading out the entire 30 GHz of potential bandwidth of Tm:YAG in the persistence time at this rate would require $\frac{30GHz}{6.25MHz} = 4800$ readout systems working simultaneously, an impractical number. Given the desire to read out broadband spectral information with high resolution in a limited time and not use 4800 readout systems, a solution to the distortion problem is required.

As discussed by Chang et al [1, 3], the readout process modifies the phase of the temporal map as a quadratic function of readout signal frequency. This distortion increases with chirp rate. Although the readout distortion is undesirable, Chang has shown that it is deterministic and removable with appropriate post processing. The processing technique described below, recovers an undistorted temporal map of the information stored in the crystal even when the chirp rate is faster than the conventional chirp rate limit of equation (1). Removing this distortion with post processing allows capturing high-resolution temporal maps of the spectral information with extremely fast chirp rates.

In Figure 4 [3], an analog signal consisting of four separate frequencies at 262 MHz, 263 MHz, 264 MHz, and 270 MHz caused the laser to burn four spectral holes into the S2 material. Two chirped readouts were then performed at rates of 0.2 MHz/µs and 1 MHz/µs. The windows on the left side of Figure 4 depict readout signals of the four spectral holes and the windows on the right show the corresponding recovered spectra. As can be seen, the faster chirp rate causes the readout signal to become less recognizable; however, both recovered signals accurately portray the true spectral features.

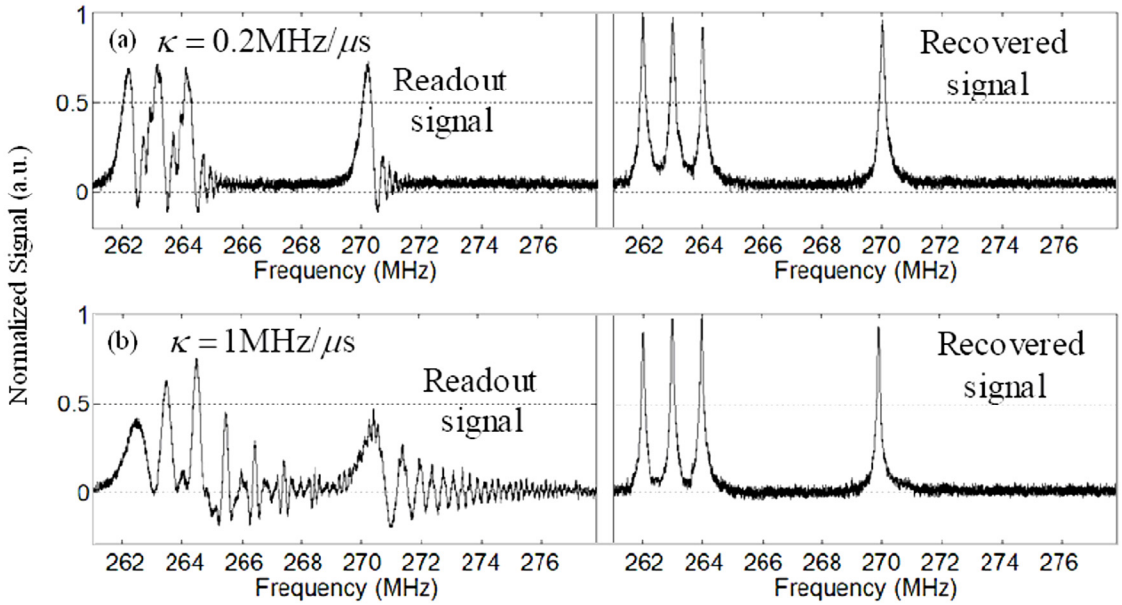

Figure 4: Left side: Experimental readout of four spectral holes burned at 262 MHz, 263 MHz, 264 MHz, and 270 MHz. (a) Readout chirp at 0.2 MHz/µs, (b) Readout chirp at 1 MHz/µs
Right side: corresponding recovered spectra.

Spectral Feature Recovery

Frequency chirped readout uses an input laser field of the form [3]

$$E_c^{in}(t) = E_0 \cos(2\pi\nu_s t + \pi\kappa t^2) \tag{2}$$

to probe the material where the amplitude of the field is E_0, κ is the chirp rate in Hz^2, and ν_s is the starting frequency of the chirp in Hz. When probing with a chirped laser field of this form, the photo-detector captures the intensity of the light exiting the S2 material as

$$\left|E_c^{out}(t)\right|^2 = E_0^2 - 2E_0^2 \int_0^\infty \gamma(\tau)\cos[2\pi\kappa t\tau + \phi_l + \phi_q + \varphi(\tau)]\mathrm{d}\tau\,, \tag{3}$$

where τ is the variable of integration over time, $\gamma(\tau)$ is the spectral grating amplitude, $\varphi(\tau)$ is the phase of the spectral grating, $\phi_l = 2\pi\nu_s\tau$ is the starting frequency dependent phase, and $\phi_q = -\pi\kappa\tau^2$ is the quadratic phase. The quadratic phase term causes the readout signal to differ from the true spectrum contained within the material and this distortion increases with chirp rate. The recovery algorithm developed by Chang et al removes the distortion by transforming the readout signal to the Fourier domain, compensating for the quadratic phase distortion, and then inverse transforming the resultant signal. Recovery is achieved as follows:

$$S'(\nu) = S(\nu)R(\nu, \nu_s, \kappa)\,, \tag{4}$$

where

$$S(\nu) = [A\gamma(|\nu/\kappa|)/2\kappa]\exp\{j[2\pi\nu_s\nu/\kappa - \mathrm{sgn}(\nu)\pi\nu^2/\kappa + \mathrm{sgn}(\nu)\varphi(|\nu/\kappa|)]\} \tag{5}$$

is the Fourier transform of the readout signal, and

$$R(\nu,\nu_s,\kappa) = \exp\{-j\pi[2\nu_s\nu - \mathrm{sgn}(\nu)\nu^2]/\kappa\} \tag{6}$$

is the phase compensation factor in the frequency domain. This process removes the linear and quadratic phases giving

$$S'(\nu) = A[\gamma(|\nu/\kappa|)/2\kappa]\exp[j\,\mathrm{sgn}(\nu)\varphi(|\nu/\kappa|)]\,. \tag{7}$$

Finally, the algorithm inverse Fourier transforms $S'(\nu)$, producing the distortion free time domain spectral map

$$S(t) = A\int_0^\infty \gamma(\tau_\nu)\cos[2\pi\tau_\nu\kappa t + \varphi(\tau_\nu)]d\tau_\nu = A\alpha(\kappa t)\,, \tag{8}$$

which is an exact mapping of the features within the crystal. This technique is effective for distortion removal from spectral holes as shown in Figure 4 as well as for any arbitrary spectral features. The recovery algorithm is simply a multiplication in the frequency domain and is therefore a linear and time invariant (LTI) operation. Since the system is linear, superposition applies. If an arbitrary spectrum is thought of as a number of shifted and scaled individual spectral holes, then the readout is the sum of the shifted and scaled responses of the recovery algorithm to those holes. Thus, the algorithm recovers an arbitrary spectrum equally as well as a single spectral hole.

CONTINUOUS RECOVERY PROCESSING

Work performed for this thesis adapts the recovery algorithm for continuous spectral feature recovery and implements it using a digital signal processor. The requirements, as given by DARPA, are to continuously recover spectral features as narrow as 25 kHz read out at a chirp rate of 1 MHz/μs, and keep the latency between when the chirp scans to when the processor fully recovers the signal less than 1 ms. This chirp rate is $\frac{1MHz/\mu s}{(25kHz)^2} = 1600$ times the conventional chirp rate limit allowed without post processing, and the latency is much less than the persistence time of Tm:YAG.

With the frequency domain based recovery approach, the post processing system must capture the entire distorted temporal map before performing recovery. With that technique, continuous processing is impossible. In addition, the latency as defined above would be as long as the time required to readout the entire spectrum plus the time needed to perform a DFT, multiplication, and inverse DFT on vectors as long as the entire readout sequence. If analyzing a 10 GHz analog signal with a chirp rate of 1 MHz/μs, the readout time itself will exceed the maximum allowable processing latency by a factor of ten. Therefore, to meet the requirements, processing must begin before reading out the entire temporal map.

Since recovery is a linear and time invariant filtering operation, the post processor can perform the distortion removal in the frequency domain as outlined above or in the time domain with a filter. Performing recovery in the time domain has the benefit that

processing begins as soon as the readout process starts and the filter only needs to be as large as is necessary to achieve a desired frequency resolution. This approach allows for continuous recovery while minimizing computation and memory requirements on the processor.

Designing the Time-Domain Filter

With the desired frequency response of the filter known (see equation (6)), the goal is to create a time domain filter to match this frequency response as closely as possible. The time-domain compensation could use either an infinite impulse response (IIR) or a finite impulse response (FIR) filter.

Designing an IIR filter corresponds to the problem of choosing the filter's pole and zero locations that result in the best match of the filter response to the desired response. This task is difficult as the phase response of a filter is a nonlinear function of the filter coefficients. Because the recovery filter is an all-pass filter, meaning its magnitude response is unity for all frequencies, the zeros are located at the conjugate reciprocal locations of the poles, and thus we only need to determine the pole locations to define the filter. The general equation for an Nth order all-pass filter with coefficients a_k is as follows:

$$A(z) = \frac{a_N + a_{N-1}z^{-1} + \ldots + a_1 z^{-N+1} + z^{-N}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}}. \tag{9}$$

The phase response of $A(e^{j\omega})$ is

$$\angle A(e^{j\omega}) = -N\omega + 2\arctan\left(\frac{\sum_{k=1}^{N} a_k \sin(k\omega)}{1+\sum_{k=1}^{N} a_k \cos(k\omega)}\right) [6]. \tag{10}$$

Because of the nonlinear relationship between filter coefficients and phase response, designing the filter requires the use of a nonlinear optimization technique such as Newton's method. The idea is to minimize a scalar objective function with respect to the filter coefficients, where the objective function is a vector norm of the error between the all-pass filter's phase response and the prescribed phase response.

Designing a FIR filter using the frequency sampling method is more straightforward than designing an IIR filter. The frequency sampling method uses the inverse DFT on equally spaced samples of the desired frequency response to determine the FIR filter coefficients. The frequency response of the FIR filter will exactly match the desired response at the DFT sample frequencies. Between the sample frequencies, the response may vary from the desired response. Nevertheless, by choosing the frequency spacing appropriately, one can make the frequency response of the filter arbitrarily close to a desired response.

Given a desired accuracy for phase compensation, an IIR filter will in general require fewer taps than does an FIR filter. Even so, because of the nonlinear optimization required in the design of IIR filters and the large number of filter taps required for this application, FIR filters are more attractive from a design perspective. Owing to the ease of determining the filter coefficients, unconditional stability, and simplicity of implementation, the continuous recovery processor developed here uses an FIR rather than an IIR filter.

A technique to improve the accuracy of the FIR filter's frequency response is to use N frequency response comparison points and M filter taps where N > M, and optimize the coefficients in the least squared error sense using the matrix pseudo-inverse. The technique [5] finds a set of filter coefficients *h* that minimizes the residuals of the over-determined set of equations

$$Wh = H_d . \tag{11}$$

Where

$$W = \begin{bmatrix} e^{-(j\omega_0 0)} & \cdots & e^{-(j\omega_0 (M-1))} \\ \vdots & \ddots & \vdots \\ e^{-(j\omega_{N-1} 0)} & \cdots & e^{-(j\omega_{N-1} (M-1))} \end{bmatrix},$$

$$H_d = \begin{bmatrix} Hd(\omega_0) \\ Hd(\omega_1) \\ \vdots \\ Hd(\omega_{N-1}) \end{bmatrix},$$

ω_k is the kth equally spaced sample frequency, and $H_d(\omega_k)$ is the desired frequency response at frequency ω_k. This technique reduces the deviation of the actual frequency response from the desired frequency response in the regions between the sample points; however, it does not guarantee that the actual frequency response matches the desired frequency response at any sample point.

When deriving the time-domain filter from the frequency-domain compensation factor, given in equation (6), one must sample the frequency-domain compensation factor frequently enough to prevent aliasing. Aliasing is an effect, usually undesirable, that occurs when sampling a signal at a rate that is less than what is required to reconstruct

unambiguously the analog signal from its discrete samples. According to the Nyquist theorem, if the bandwidth of a signal exceeds half the sample rate, aliasing will occur. The frequency-domain compensation factor has unity magnitude and a phase response similar to the one shown in Figure 5 where the left half of the curve is a negative quadratic and the right half is a positive quadratic. Preventing aliasing requires that the phase of the signal change by fewer than π radians between each frequency sample step Δf. Given that the phase of the compensation factor changes most rapidly at high frequencies, the maximum frequency change between sample points is obtained as follows. First, equation (6) is modified by removing the linear phase compensation portion since linear phase distortion is simply a delay and cannot be removed with a causal filter.

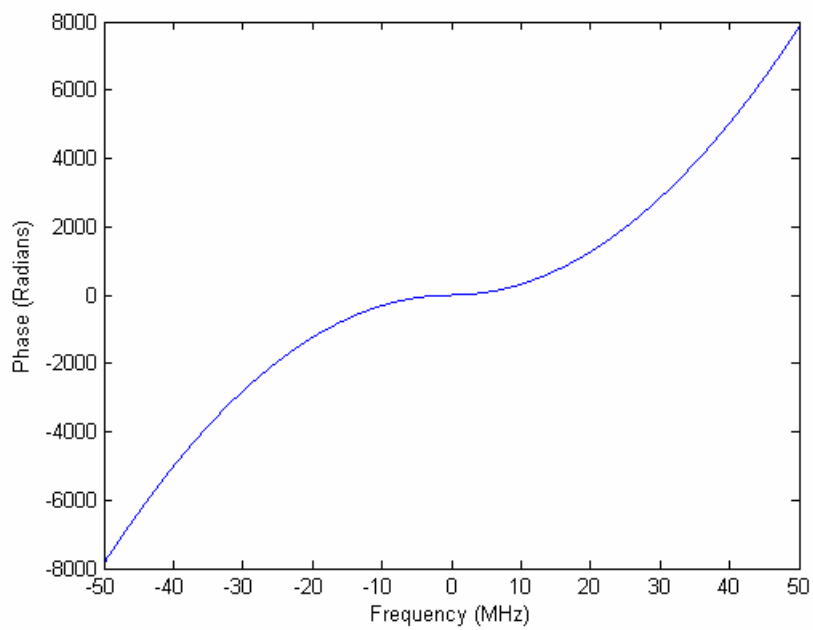

Figure 5: Phase Response of Typical Spectral Recovery Factor

The new equation,

$$R_q(\nu,\kappa) = \exp\{j\pi\,\mathrm{sgn}(\nu)\nu^2/\kappa\}\,, \tag{12}$$

corrects only for quadratic phase distortion. Equation (12) has the phase response:

$$\angle R_q(\nu,\kappa) = \pi \operatorname{sgn}(\nu)\nu^2 / \kappa \,. \tag{13}$$

By considering the slope of the phase response, one can determine the minimum sample step, Δf, such that the phase change between consecutive frequency samples is always fewer than π radians. The slope of the phase response with respect to the frequency is

$$\frac{\partial}{\partial \nu} \angle R_q(\nu,\kappa) = \left|2\pi\nu / \kappa\right| . \tag{14}$$

Provided the slope given in equation (14) is less than $\pi/\Delta f$, aliasing will not occur. Setting equation (14) equal to $\pi/\Delta f$ and solving for Δf where the slope is at its maximum, i.e. when ν=Fs/2 where Fs is the sample rate of the time domain compensation factor, will yield the maximum allowable frequency step size as shown in equation (15) through equation (17).

$$\frac{\pi}{\Delta f} = \left|2\pi\nu / \kappa\right| \tag{15}$$

$$\frac{\pi}{\Delta f} = \left|\pi Fs / \kappa\right| \tag{16}$$

$$\Delta f = \frac{\kappa}{Fs} \tag{17}$$

Equation (17) gives the maximum frequency step size that avoids aliasing when sampling the frequency-domain compensation factor.

Sampling the frequency-domain compensation factor at a rate above its minimum and calculating the IDFT produces the time-domain compensation filter. Removing the portion of the filter consisting of zeros reduces the cost of evaluating the FIR filter

without hindering the filter's ability to remove quadratic phase distortion. Figure 6 shows the non-zero portion of a time-domain recovery filter. The filter is essentially a decreasing linear frequency chirp spanning from Fs/2 to DC. Notice the amplitude is essentially constant over the frequency chirp because this is an all-pass filter and as such has a constant magnitude response for all frequencies. Convolving this FIR filter with the distorted readout signal removes the quadratic phase distortion.

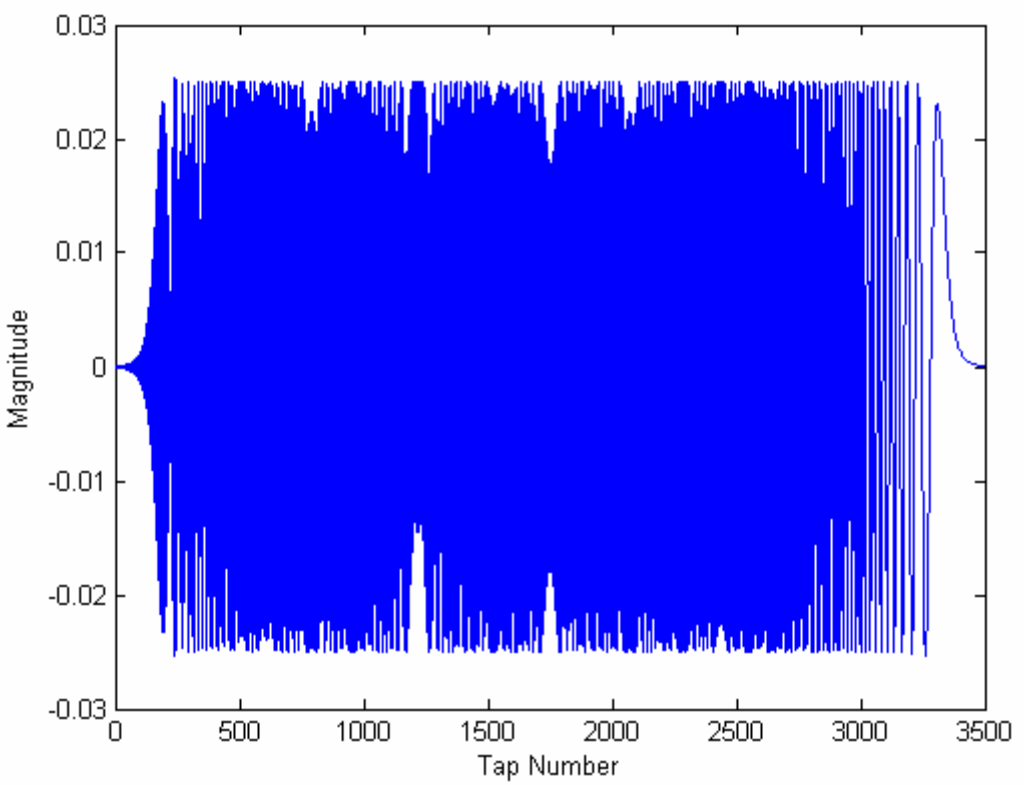

Figure 6: Typical time-domain compensation filter

Filter Length Requirements

The required filter length is determined by the duration of the ringing caused by the narrowest spectral feature of interest and the readout bandwidth at the detector. The

ringing due to a 25 kHz feature will decay to e^{-1} of its initial value in $\frac{1}{25kHz} = 40\mu s$. This provides an order of magnitude approximation to the filter duration required to recover a 25 kHz feature. Readout bandwidth determines the required ADC sample rate. The readout bandwidth is approximately equal to $\frac{\kappa}{\delta\nu}$. With a required chirp rate of κ=1 MHz/μs and a minimum feature width of $\delta\nu$=25 kHz, the readout bandwidth is about 40 MHz. To satisfy the Nyquist criterion, the sample rate must be at least 80 MSPS for this signal. If we choose a sample rate of 80 MSPS, the FIR filter must have at least 40 μs * 80 MSPS = 3200 taps. Since the filter in Figure 6 uses the same sample rate and chirp rate, the length of the non-zero portion of the filter, approximately 3200 taps, corresponds to the approximation for filter length given above.

In reality, the bandwidth at the photo-detector when a 1 MHz/μs chirp encounters a 25 kHz wide feature is greater than 40 MHz. This is because the intensity of the light from the free induction decay has not completely reached zero after $\frac{1}{25kHz} = 40\mu s$, but instead has only decayed to e^{-1} of its initial value. Light from the free induction decay is still present after this time and hence produces higher readout bandwidth owing to the free induction decay light beating with still higher frequencies from the chirp. When the bandwidth is artificially limited to 40 MHz, the recovered feature width becomes greater than 25 kHz. Figure 7 illustrates what happens to the recovered feature width when the readout signal is low-pass filtered before recovery.

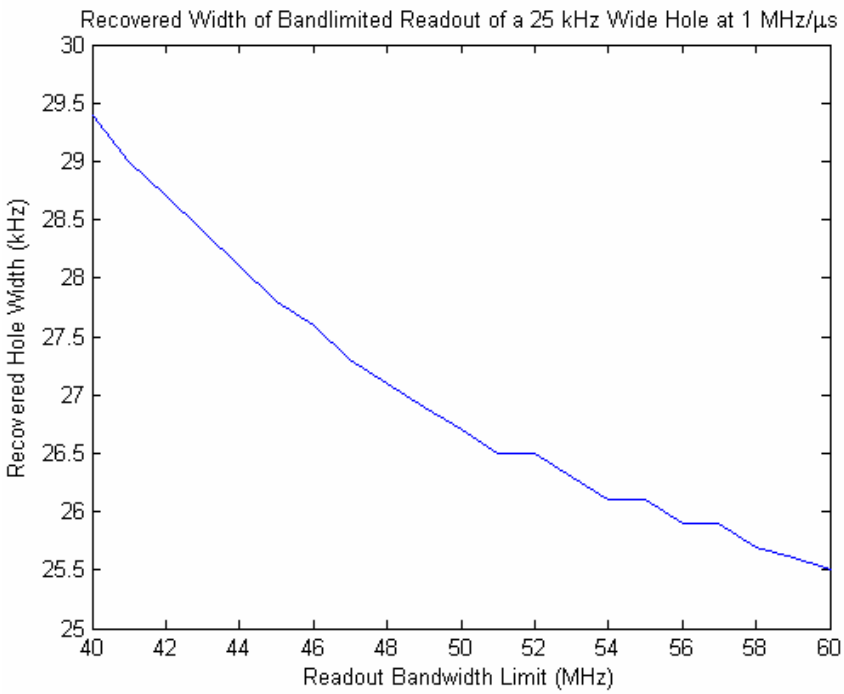

Figure 7: Limiting the bandwidth of the readout signal increases the width of the recovered feature. Bandwidth limit corresponds to the 3 dB point of the low pass linear phase FIR filter used.

Limiting the readout bandwidth to 40 MHz causes the 25 kHz feature in the crystal to appear as 29.4 kHz after recovery. Approaching the actual feature width following recovery requires one to capture much more than 40 MHz of readout bandwidth. This is because the free induction decay endures for more than 40 µs and the light from it beats with the chirp causing a readout signal with greater than 40 MHz of bandwidth.

Several factors limit the overall readout bandwidth. First, the photo-detector may pose a limit. Second, the anti-aliasing filter before the ADC may limit the bandwidth. Finally, the recovery filter itself may artificially limit the readout bandwidth. Figure 8 shows a spectrogram of the time-domain compensation factor with the zeros removed. One can see that the nonzero portion of the recovery filter is simply a linear frequency chirp from Fs/2 to DC. Reducing the filter length by removing a portion from the

beginning of the filter to lessen the computation requirements has the same effect as low-pass filtering the readout signal and thus widens the recovered features.

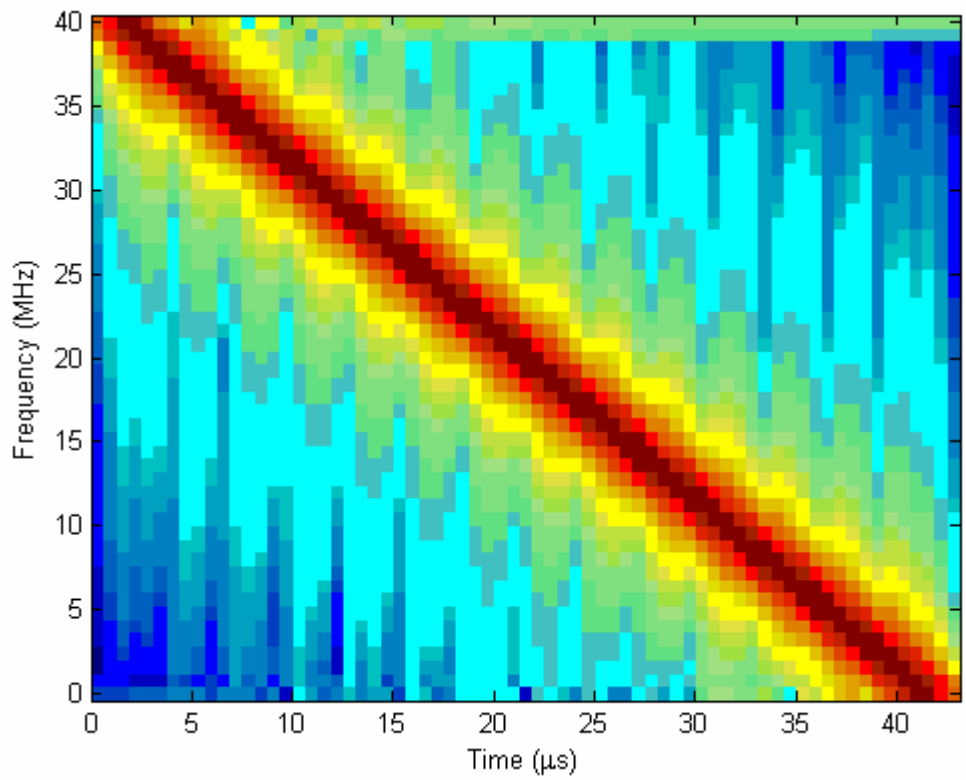

Figure 8: Spectrogram of a time-domain recovery filter

Sampling the frequency-domain compensation factor at its minimum rate as given in equation (17) and subsequently reducing the filter length to L reduces the effective readout bandwidth to

$$B = L\Delta f = \frac{L\kappa}{Fs}. \tag{18}$$

For example if the 3200 tap filter shown in Figure 6 is reduced to 2000 taps, the effective readout bandwidth will be reduced from 40 MHz to $B = \frac{2000 \cdot 1MHz/\mu s}{80MHz} = 25MHz$ as

shown in the spectrogram in Figure 9, however, this will also increase the spectral feature width to greater than 25 kHz.

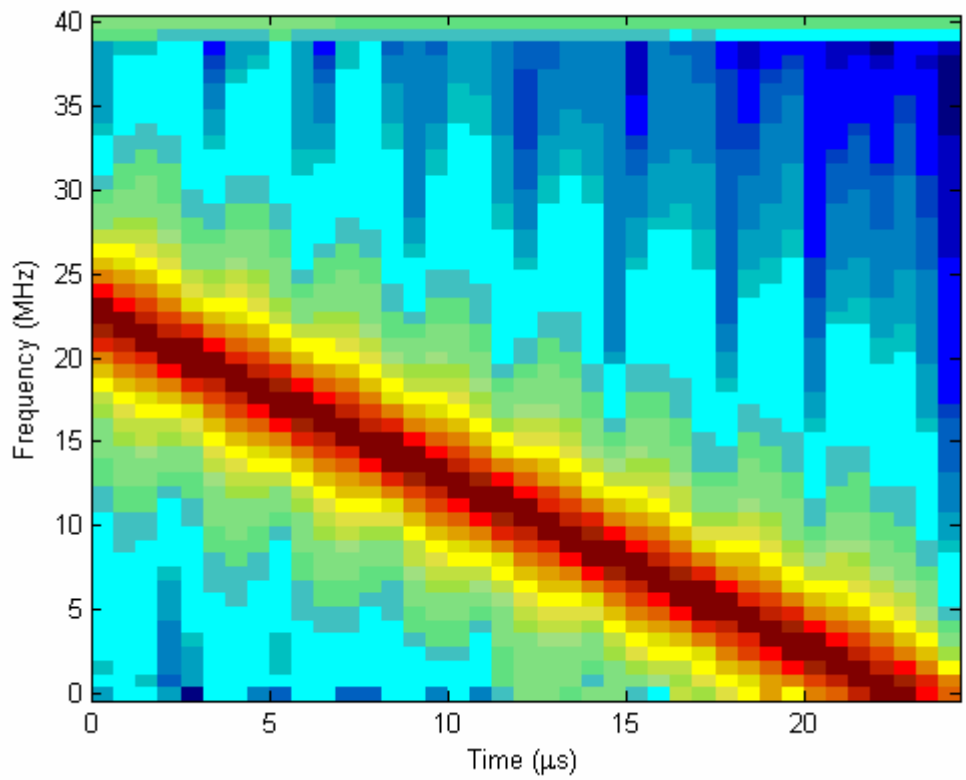

Figure 9: Spectrogram of filter shortened to 2000 taps

Given the goal of recovering 25 kHz features as closely as possible to their original width, it is clear that one must capture more than 40 MHz of readout bandwidth, and the filter length must be greater than 40 µs as given by the order of magnitude approximations. From Figure 7, capturing 45 MHz of readout bandwidth and recovering with a filter that endures for 50 µs will recover the readout of a 25 kHz feature to 27.8 kHz. This is an increase of $\frac{27.8-25}{25}100\% = 11.2\%$, which has been deemed an acceptable amount. Our attention can now be turned towards implementing a real-time recovery algorithm in hardware.

LINEAR CONVOLUTION WITH THE DFT

An FIR filter enduring for 50 μs when sampled at 100 MSPS consists of 5000 filter taps. Directly computing the convolution of a 5000 tap FIR filter with a signal at 100 MSPS requires 500 billion multiplies and accumulates per second (500 GMAC/s). This is significantly more computation than is achievable on the fastest signal processing devices. An application specific integrated circuit (ASIC) could achieve this; however the cost is prohibitively high for this application. Fortunately, there exist convolution algorithms that exploit the efficiency of fast Fourier transform (FFT) algorithms to reduce significantly the computational requirements of evaluating large FIR filters.

From linear system theory, we know that the inverse Fourier transform of the product of the Fourier transforms of two signals is equal to the linear convolution of the two signals. Signals in a computer, however, are discrete and we must use the DFT rather than the continuous Fourier transform. Multiplying the DFTs of two signals and performing the inverse DFT (IDFT) on the product produces a circular convolution of the two signals rather than a linear convolution [10]. Since FIR-filtering requires linear convolution, one must take special care to ensure that the circular convolution performed is equivalent to the linear convolution required. Linearly convolving discrete signals of length L and length P sequence produces a length L+P-1 sequence. In order for the IDFT of the product of the DFTs of each signal to equal the linear convolution of the two signals, we must pad each sequence with zeros to a length greater than or equal to L+P-1 prior to performing the transforms. Under this condition, the DFT, multiply, and IDFT

steps will produce a linear convolution of two finite length discrete signals. Using the DFT to perform convolution is useful because of the existence of fast Fourier transform (FFT) algorithms. The radix-2 Cooley-Tukey [10] FFT algorithm has a computational complexity proportional to $N\log_2 N$, where N is the transform length, rather than the N^2 complexity of the DFT and direct convolution. Therefore, performing FIR filtering with the FFT becomes computationally efficient, especially for long filter lengths.

Block convolution methods

Two widely known methods exist that are well suited to performing linear convolution of arbitrarily and possibly infinitely long signals with an FIR filter. The two methods, known as overlap-add and overlap-save, perform block convolution. Block convolution is a method that segments a signal into blocks, convolves each block with the filter coefficients, and fits the convolution results from adjacent blocks together appropriately to yield the overall convolution. Block convolution allows the use of any method to perform the intermediate convolutions, although algorithms often use the DFT method noted above because of the efficiency with which the FFT performs the DFT.

Overlap-Add

Since convolution is a linear operation, superposition applies. Therefore, the overall response of the filter due to the entire input signal is equal to the sum of appropriately delayed responses to segments of the input signal. This allows one to

perform a series of fixed length convolutions to convolve a filter with an input signal of unknown and possibly infinite length. Fixed length convolutions are especially useful when performing them with the DFT. To illustrate this method consider convolving the signal x = [1 1 -1 -1 1 1 1 1] with the filter h = [-1 -1 1 -1]. This yields the result y = [-1 -2 1 2 -2 -2 0 -2 -1 0 -1]. One can also obtain this result by convolving segments of x with h and adding the partially overlapping individual responses. The process of convolving x with h using the overlap-add method is shown in Figure 10. The technique divides the input signal into two segments each of length four, individually convolves the two segments with h, and overlaps and adds the results to yield the overall convolution.

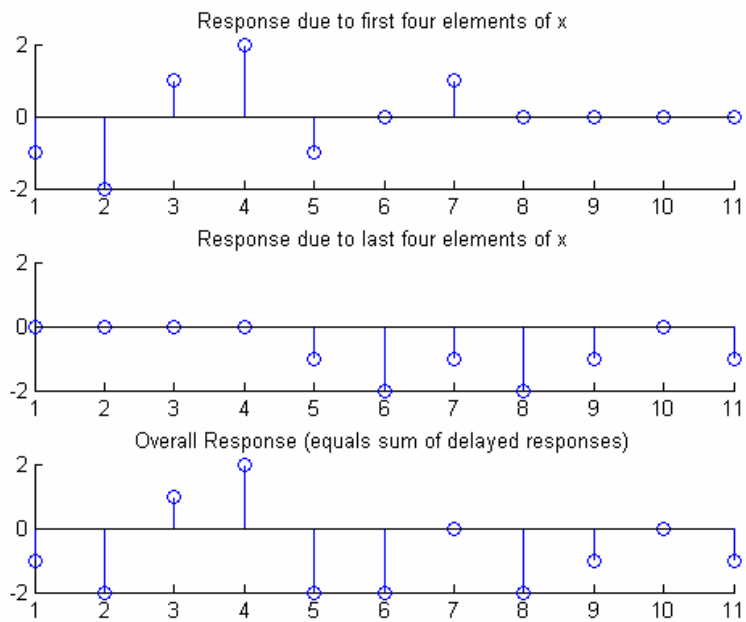

Figure 10: Example of the overlap-add method. The input signal is divided into two segments each of length four, each segment is convolved with h, and the delayed results are added to yield the overall convolution.

Dividing the input signal into multiple segments and performing a convolution, overlap, and addition for each segment as shown in Figure 11 extends this method allowing it to convolve a filter with an arbitrary length input sequence.

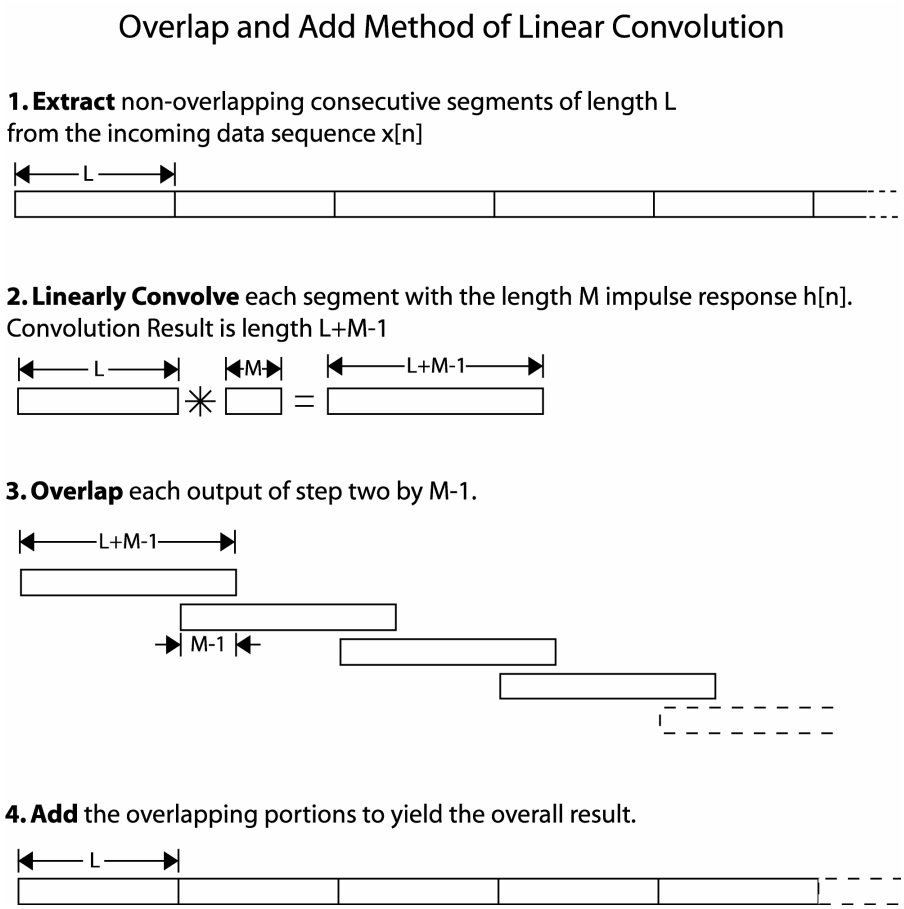

Figure 11: Overlap-add convolution algorithm

<u>Overlap-Save</u>

The overlap-save method works by choosing the values of the circular convolution that are equivalent to linear convolution and discarding the remaining samples. Again using the example input sequence $x = \begin{bmatrix} 1 & 1 & -1 & -1 & 1 & 1 & 1 & 1 \end{bmatrix}$ and the filter $h = \begin{bmatrix} -1 & -1 & 1 & -1 \end{bmatrix}$, we can convolve the two with the overlap-save method.

The first step required is to pad x with M-1 zeros at the beginning of the sequence, where M is the filter length. This produces the sequence $\hat{x} = \begin{bmatrix} 0 & 0 & 0 & x \end{bmatrix}$. We also pad h with L-M zeros producing $\hat{h} = \begin{bmatrix} h & 0 & 0 & 0 \end{bmatrix}$, where L is the segment length that is set arbitrarily to seven in this example. Next we can extract overlapping segments of the signal each of length seven and circularly convolve each segment with $\hat{h}$ as follows where $\bigcirc$ represents circular convolution.

$$
\begin{aligned}
y_1 &= \hat{x}(1:7) \bigcirc \hat{h} \\
&= \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & -1 & -1 \end{bmatrix} \bigcirc \hat{h} \\
&= \begin{bmatrix} -1 & 0 & 1 & -1 & -2 & 1 & 2 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
y_2 &= \hat{x}(5:11) \bigcirc \hat{h} \\
&= \begin{bmatrix} 1 & -1 & -1 & 1 & 1 & 1 & 1 \end{bmatrix} \bigcirc \hat{h} \\
&= \begin{bmatrix} -2 & 0 & 2 & -2 & -2 & 0 & -2 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
y_3 &= [\hat{x}(9:11), 0, 0, 0, 0] \bigcirc \hat{h} \\
&= \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \bigcirc \hat{h} \\
&= \begin{bmatrix} -1 & -2 & -1 & -1 & 0 & -1 & 0 \end{bmatrix}
\end{aligned}
$$

Finally, discarding the first M-1 = 3 samples of each circular convolution above and concatenating the remaining four samples of each yields the overall linear convolution result $y = \begin{bmatrix} -1 & -2 & 1 & 2 & -2 & -2 & 0 & -2 & -1 & 0 & -1 & 0 \end{bmatrix}$.

The overlap-save method lends itself well to using the DFT because a circular convolution is precisely what is required in the intermediate convolutions. As with the overlap-add method, the overlap-save method can also be extended to arbitrarily long input sequences x as shown in Figure 12.

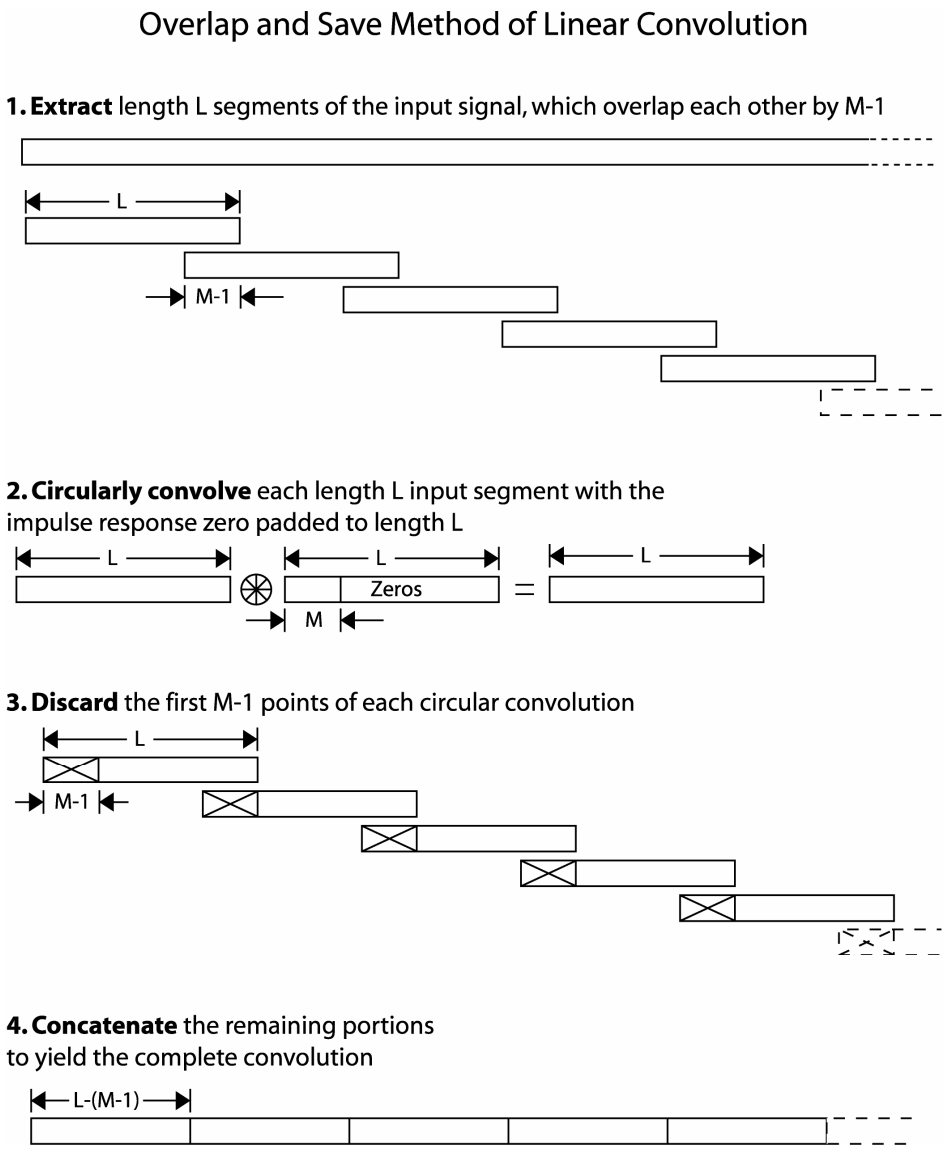

Figure 12: Overlap-save convolution algorithm

Computational Complexity comparison of Overlap-Add vs. Overlap-Save

By comparing the fundamental operations required per input sample as a function of DFT size, we can determine which convolution algorithm is most efficient. In this study, the fundamental operations are the sum of the number of real multiplies and the number of real additions required when convolving with the radix-two Cooley-Tukey FFT algorithm. To compute a DFT of length N_{DFT}, this FFT algorithm requires $\log_2(N_{DFT})$ stages with $\frac{N_{DFT}}{2}$ butterflies per stage. Each butterfly consists of one complex multiply and two complex additions. Given that each complex multiply requires four real multiplies and two real additions while each complex addition requires two real additions, each butterfly consists of four real multiplies and six real additions. Therefore, an N_{DFT} point radix-two FFT entails computing $2N_{DFT}\log_2(N_{DFT})$ real multiplications and $3N_{DFT}\log_2(N_{DFT})$ real additions.

The overlap-add method consists of performing a length L+M-1 = N_{DFT} FFT, N_{DFT} complex multiplies, a length N_{DFT} IFFT, and M-1 real additions for every L = N_{DFT} – M + 1 input samples. Thus the overlap-add method requires $4N_{DFT}\log_2 N_{DFT} + 4N_{DFT}$ real multiplies and $6N_{DFT}\log_2 N_{DFT} + 2N_{DFT} + M - 1$ real additions for every L = N_{DFT} – M + 1 input samples. Combining multiplies and additions and simplifying gives the following formula for real operations per input sample:

$$ops\_per\_sample_{OLA}(N_{DFT}, M) = \frac{10N_{DFT}\log_2 N_{DFT} + 6N_{DFT} + M - 1}{N_{DFT} - M + 1}. \qquad (19)$$

For the overlap-save method, a length L = N_{DFT} point FFT, N_{DFT} complex multiplies, and a length N_{DFT} IFFT are required for every N_{DFT} -M+1 input samples. Combining multiplies and additions and simplifying yields the following equation:

$$ops\_per\_sample_{OLS}(N_{DFT}, M) = \frac{10N_{DFT}\log_2 N_{DFT} + 6N_{DFT}}{N_{DFT} - M + 1}. \tag{20}$$

Both complexity equations are only meaningful when $N_{DFT} \geq$ filter length. Notice that the only difference between the two equations is the extra M-1 addition operations per input data segment required in the overlap and add algorithm.

The plot in Figure 13 compares the operation count per input sample as a function of N_{DFT} for a 4000 tap FIR filter.

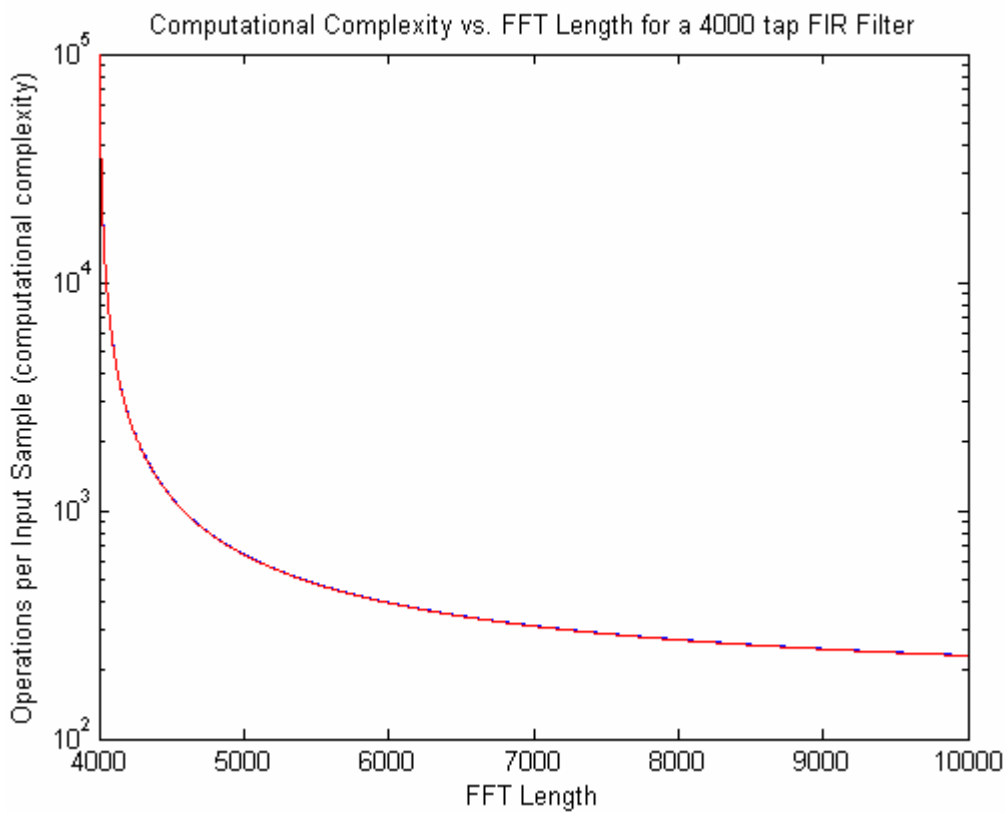

Figure 13: Comparison of real operations per input sample versus FFT length for the two block convolution methods. The two methods have essentially equal complexity.

Given that computation of a FIR filter in the time- domain requires multiplying each input sample by every filter coefficient and accumulating the products, a 4000 tap FIR filter requires 8000 fundamental operations per input sample with direct convolution. The plot shows that both block convolution methods allow reducing the computational complexity substantially below that of direct convolution.

From a computational complexity standpoint, the two block convolution algorithms are essentially equal. The increased computation required to perform overlap-add versus overlap-save is less than one percent for FFT lengths above 4000 and less than 0.5 percent for FFT lengths above 6100.

Although the computational complexity analysis gives a close approximation of the processing hardware requirements, it ignores implementation details and is thus only an estimate. For the two block convolution methods, the computational complexity and memory requirements differ from one another in the portions of the algorithm preceding and following the convolution step.

The overlap and add method requires relatively simple processing before the convolution as it only needs to pad incoming segments with zeros, whereas the overlap and save method requires a more complicated algorithm before the convolution to save portions of the incoming segments such that it can later send them to the convolution unit. Storing data segments causes the overlap-save method to require more memory for the processing step preceding the convolution.

The structure following the convolution, however, is more complicated for overlap-add than it is for overlap-save. In overlap-add, the processor must appropriately overlap and add the convolved segments. With overlap-save, the processor simply

discards portions of the convolved segments, concatenates the remaining portions with each other, and then sends them out. In this area, the overlap-add method will require more computation and more memory.

Since the theoretical complexity difference is slight, implementation details will dominate any actual differences. Only realizing both methods in hardware and then performing a side-by-side comparison will reveal these differences. The processor developed for this thesis implements the overlap-add method, but should one desire a rigorous comparison, the overlap-save method should also be implemented. However, this is beyond the scope of this thesis.

DEVELOPMENT OF THE CONTINUOUS RECOVERY ARCHITECTURE

From the computational complexity analysis performed above, it is possible to determine the processing requirements of the physical implementation hardware. Assuming, from Figure 13, that the overlap-add algorithm will require at least 200 operations per input sample and the input data rate is approximately 100 MSPS, the recovery processor must compute at least $200 \cdot 100 \cdot 10^6 = 2 \cdot 10^{10}$ operations per second. Although many conventional programmable digital signal processors (PDSP) can compute several multiply and accumulate operations per clock cycle, a PDSP is fundamentally a serial data processor. Given that high-end PDSPs have maximum clock rates of approximately 1 GHz, at least twenty multiply or accumulate operations per clock cycle would be required of the PDSP in addition to data flow control. Therefore, recovery processing is too large of a computational load for a single PDSP. For this reason, the recovery processor is implemented in an FPGA rather than a conventional PDSP. An FPGA is a device that contains programmable logic and a programmable interconnect array between the logic. By connecting the logic gates appropriately, one can configure an FPGA to perform operations ranging from simple logic or math functions to complex digital signal processing algorithms. Although the clock frequency of an FPGA is usually less than that of a PDSP, an FPGA makes up for this discrepancy by performing many operations in parallel. For algorithms with a large amount of parallelism, such as the pipelined FFT used in the recovery processor where many MACs are performed simultaneously, an FPGA significantly outperforms a PDSP. Whereas a

PDSP uses a single arithmetic and logic unit (ALU) to compute serially the MACs in the FFT, an FPGA uses its dedicated hardware MAC units, numbering in the hundreds in some FPGAs, to compute many of the MACs for the FFT in parallel.

Work performed for this thesis includes development of an FPGA based processing architecture for efficient computation of FIR filters specifically for use in real-time spectral recovery. The architecture, shown in Figure 14, implements the overlap-add convolution algorithm developed in the previous section. Data continuously streams into the input and after the initial processing latency, the filtered data streams from the output.

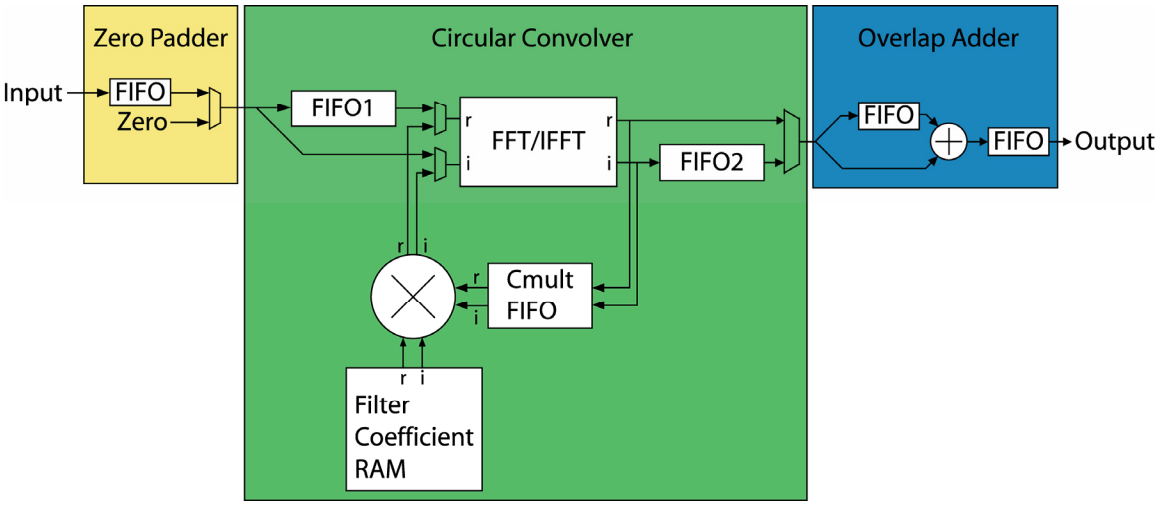

Figure 14: FPGA architecture implementing the overlap-add algorithm

First, the input data is passed to the zero-padder block. This takes in a continuous stream of samples and outputs segments of the input data padded with zeros at a higher sample rate. The circular convolver block takes the zero padded input segments, transforms each segment to the frequency domain, multiplies it by the DFT of the zero-padded filter, and brings the product back to the time domain with an IDFT. Finally, the overlap-adder block adds overlapping portions of consecutive outputs from the circular

convolver bringing the data rate back down to the ADC rate and producing the overall linear convolution result.

Zero Padder

The zero-padder takes in a continuous data stream at the rate of the ADC, segments it into portions of length L, pads each segment with M-1 zeros where M is the filter length, and outputs the padded segments at $\frac{L+M-1}{L}$ times the ADC rate as shown in Figure 15, where t_{FFT} is the time required to process one DFT.

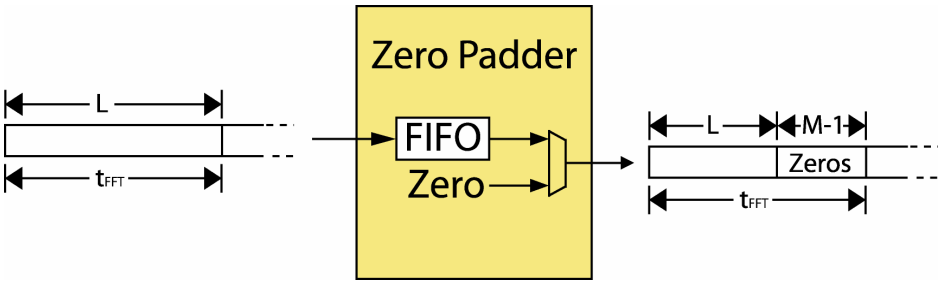

Figure 15: Block diagram of the zero-padder as implemented in the FPGA

The input data continuously enters the FIFO at the ADC rate. A multiplexer then selects whether to output zeros or data from the FIFO at the high data rate ($F_{ADC}\frac{L+M-1}{L}$). Because of the clock rate difference between the input and the output, the zero-padder uses an asynchronous FIFO to ensure reliable data transfer between the two clock domains.

Circular Convolver

The circular convolver block circularly convolves the zero-padded signal segments with the zero-padded filter coefficients. It performs a DFT on the incoming zero-padded data segments, multiplies each segment by the DFT of the zero-padded filter stored in block RAM, and performs the inverse DFT on the product. The architecture makes efficient use of the hardware by routing two segments through the FFT core simultaneously. This technique works by exploiting the symmetry properties of the DFT. The DFT of a purely real signal is complex conjugate symmetric, and the DFT of a purely imaginary signal is anti-symmetric. In addition, the IDFT of a symmetric signal is purely real, and the IDFT of an anti-symmetric signal is purely imaginary [10]. The following steps show this mathematically.

Vectors a and b represent purely real signals, and c represents the purely real filter. All signals are zero-padded appropriately. The desired computation is as follows.

$$\begin{aligned} out_a &= IDFT(DFT(a)DFT(c)) \\ out_b &= IDFT(DFT(b)DFT(c)) \end{aligned} \tag{21}$$

The same result, however, is achieved with one fewer DFT computation and one fewer complex vector multiplication by applying one input vector to the real input of the DFT and the other input vector to the imaginary input of the DFT as shown below.

$$DFT(a + jb) = DFT(a) + DFT(jb) \tag{22}$$

The first term is conjugate-symmetric and the second term is anti-symmetric. Multiplying this quantity by the DFT of the padded filter yields the following equation.

$$DFT(a + jb)DFT(c) = DFT(a)DFT(c) + DFT(jb)DFT(c) \tag{23}$$

The first term remains conjugate symmetric and the second term remains anti-symmetric because the DFT of c is conjugate-symmetric. Performing the inverse DFT yields the following equation where the first term is purely real and the second term is purely imaginary.

$$IDFT(DFT(a+jb)DFT(c)) = IDFT(DFT(a)DFT(c)) + IDFT(DFT(jb)DFT(c)) \quad (24)$$

The outputs are determined by extracting the real and imaginary parts.

$$\begin{aligned} out_a &= \Re[IDFT(DFT(a+jb)DFT(c))] \\ out_b &= \Im[IDFT(DFT(a+jb)DFT(c))] \end{aligned} \quad (25)$$

This technique essentially doubles the processing rate of a given DFT/IDFT engine for real signals by allowing it to perform two circular convolutions simultaneously.

The buffer labeled FIFO1 in the circular convolver delays every other zero-padded data segment. By doing this, two consecutive data segments enter the FFT core simultaneously. One segment enters the real input to the FFT core while the second segment enters the imaginary input to the FFT core. The FFT core computes the DFT, the data is bit-reversed, and then unloaded. While the FFT core unloads the data, another FIFO data buffer stores the output. Next, the delayed output from the complex multiplier FIFO is multiplied by the DFT of the zero-padded filter stored in block RAM. The circular convolver then routes the data from the complex multiplier back into the FFT core now configured to perform the inverse DFT. The FFT core computes the inverse DFT, bit-reverses the data, and unloads the result. Now the buffer labeled FIFO2 delays processed segments from the imaginary output of the FFT core making the simultaneous output sequential again. The timing diagram in Figure 16 shows the entire process with each time step corresponding to N clock cycles where N is the length of the DFT.

Step	Input data goes to:	FIFO 1	FFT Core			Complex multiplier FIFO	Complex multiplier	FIFO 2	Output data is from:
			Loading data from:	Processing FFT/IFFT	Unloading data to:				
1	FIFO 1	Filling	--------	--------	--------	--------	--------	--------	--------
2	FFT	Emptying	In/FIFO 1	--------	--------	--------	--------	--------	--------
3	FIFO 1	Filling	--------	FFT	--------	--------	--------	--------	--------
4	FFT	Emptying	In/FIFO 1	--------	Cmult FIFO	Filling	--------	--------	--------
5	FIFO 1	Filling	Multiplier	FFT	--------	Emptying	Multiplying	--------	--------
6	FFT	Emptying	In/FIFO 1	IFFT	Cmult FIFO	Filling	--------	--------	--------
7	FIFO 1	Filling	Multiplier	FFT	Out/FIFO 2	Emptying	Multiplying	Filling	IFFT
8	FFT	Emptying	In/FIFO 1	IFFT	Cmult FIFO	Filling	--------	Emptying	FIFO 2
9	FIFO 1	Filling	Multiplier	FFT	Out/FIFO 2	Emptying	Multiplying	Filling	IFFT
10	FFT	Emptying	In/FIFO 1	IFFT	Cmult FIFO	Filling	--------	Emptying	FIFO 2

Legend
Data input at step 1
Data input at step 2
Combined data from 1 and 2
Data input at step 3
Data input at step 4
Combined data from 3 and 4

Figure 16: Circular convolver timing diagram as implemented in the FPGA

As can be seen in Figure 16, after an initial latency, the FFT core continuously processes data in both the real and imaginary inputs and outputs. This technique keeps the hardware constantly busy making efficient use of the FPGA resources.

Overlap Adder

The overlap-adder block takes the output segments from the circular convolver block at the high clock rate, overlaps each L+M-1 length segment by M-1 with the previous segment, adds the overlapping portion, and outputs the result at the ADC rate. Figure 17 below shows the overlapping process.

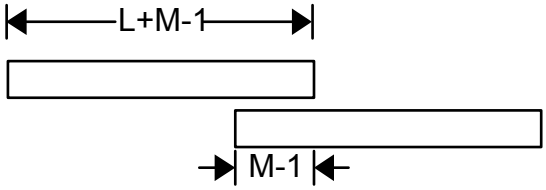

Figure 17: Overlapping portions of the processed segments are added together

With the zero-padder, convolver, and overlap-adder working in conjunction, the architecture performs a linear convolution of the incoming data sequence with the stored filter.

Processing Latency

With the architecture configured as described above, the processing latency is determined as follows, where F_{FFT} is the clock rate of the FFT core and NFFT is size of the DFT being processed. Again, L is the segment length and M the filter length.

$$\tau_L = \frac{8NFFT}{F_{FFT}} = \frac{8(L+M-1)}{F_{FFT}}$$

Figure 18 depicts the processing latency versus filter length for FFT lengths up to 16384 points. Depending on the speed of the FPGA, this allows filter lengths up to nearly 11000 taps at 90 MSPS. The shaded regions indicate the minimum size Virtex 4 SX FPGA required to implement the given filter. Note that the recovery processor is implemented, as discussed later, in a Virtex 4 SX35 with a speed grade of -10. The faster speed grade curve shown on the plot is an estimate based on the relative speed increase between the two speed grades as seen in Xilinx's FFT core datasheet [13]. Latency data for the SX25 and SX55 are estimated based on available logic and memory resources in these devices.

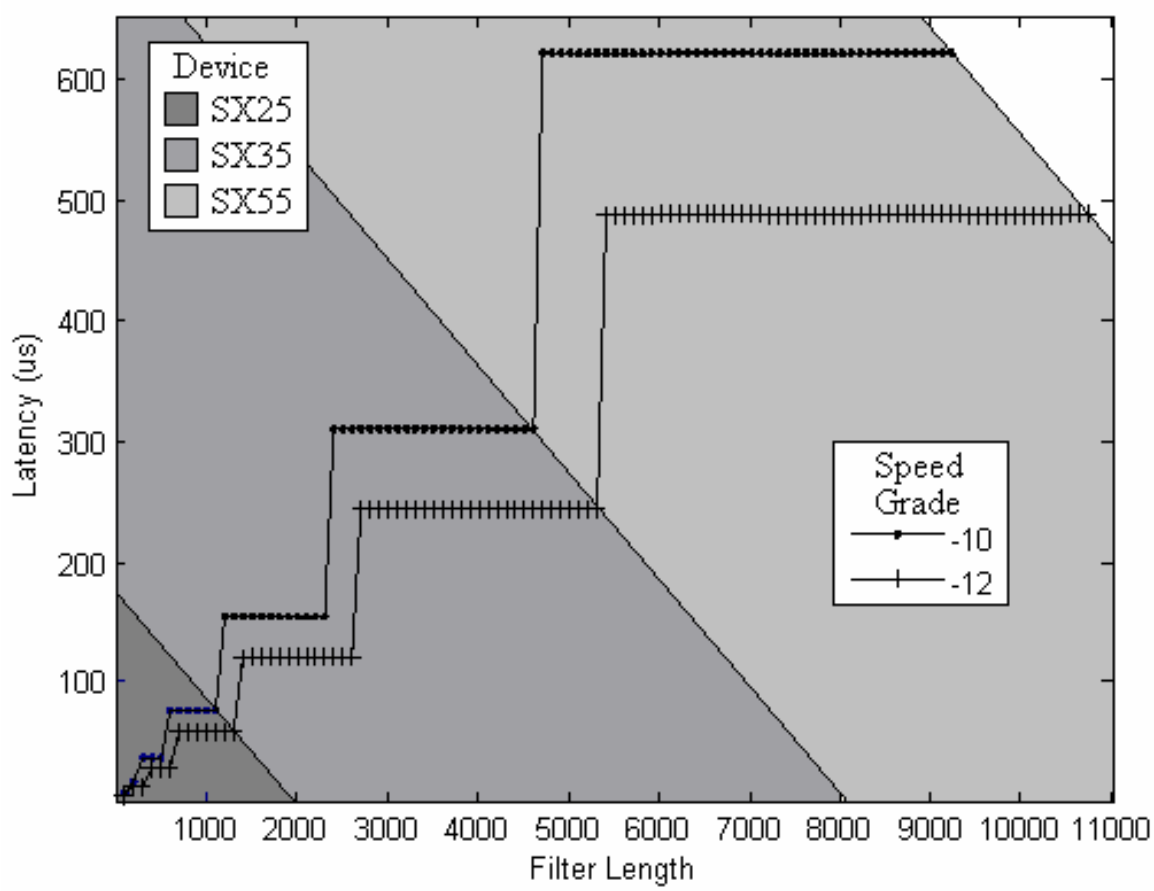

Figure 18: Estimated processing latency versus filter length for Virtex4 SX FPGA with 16 bit precision and F_{ADC} = 90 MSPS

The FFT core used in the FPGA is only available in sizes that are even powers of two causing the "stair-stepping" effect seen in Figure 18. Any filter length in the range achievable by a given FPGA may be used; however, choosing the maximum filter length for a given latency most efficiently uses the FPGA resources.

Simulation of Architecture

The entire processing architecture was described with Very high speed integrated circuit Hardware Description Language (VHDL), which is given in appendix B, and

simulated with ModelSim to ensure correct operation. The simulation reads in data from a file representing the distorted readout of a spectral hole, filters it with an FIR filter using the architecture described above, and outputs the results to another file. Figure 19 is a screenshot of the ModelSim simulation of the recovery process. The simulated readout hole is the upper trace and the recovered hole is the lower trace.

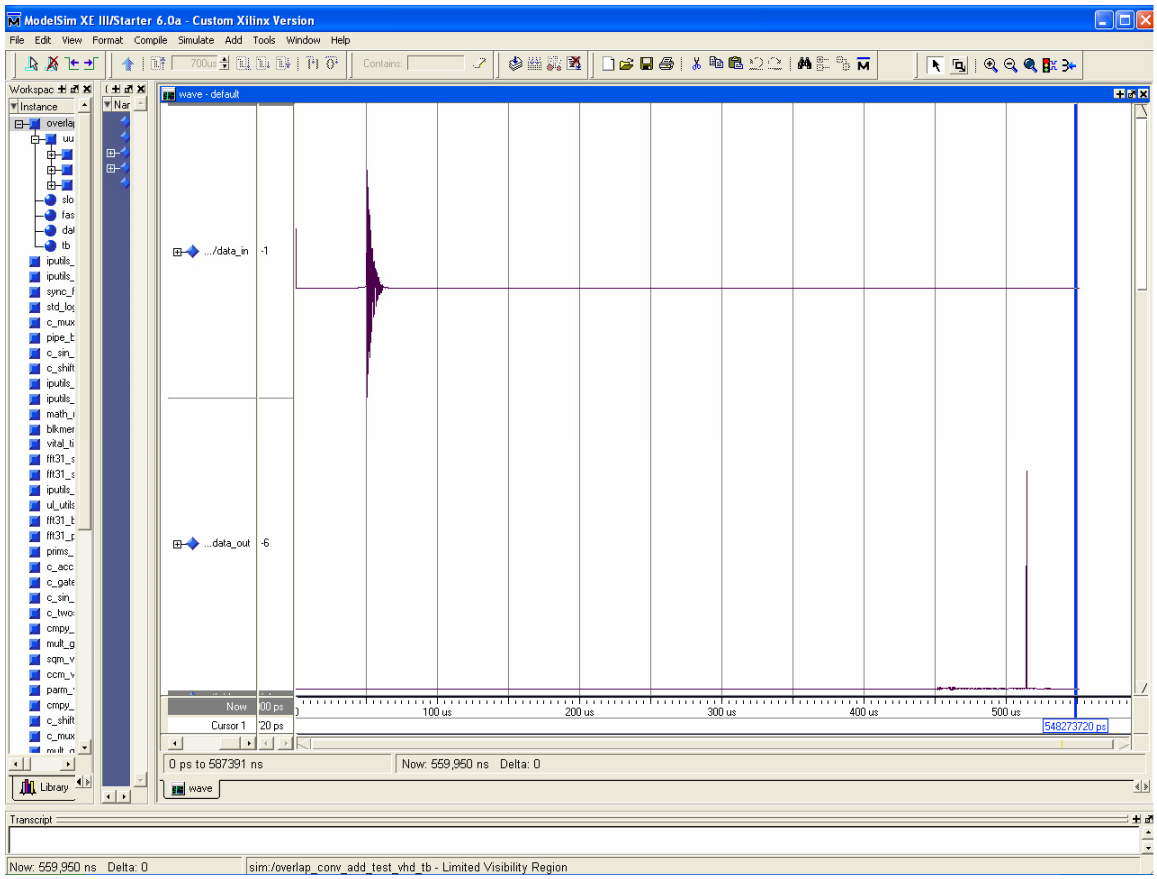

Figure 19: Simulation of the convolution architecture performing real-time recovery.

The input data represents a 100 kHz wide spectral hole read out with a chirp rate of 4 MHz/μs. This produces a readout bandwidth of $B_R > \frac{\kappa}{\delta\nu} = \frac{4MHz/\mu s}{100kHz} = 40$ MHz. The readout signal is band limited to 50 MHz prior to sampling to avoid aliasing and then

digitized with a simulated 16-bit ADC sampling at 100 MSPS. The recovery system uses a 2561 tap FIR filter using an 8192-point FFT with 5632 samples per segment. The FFT core runs at $\frac{8192}{5632} F_{ADC} = 145.45 MHz$. Note the predicted processing latency $\tau_L = \frac{8NFFT}{F_{FFT}} = \frac{65536}{145.45 MHz} = 450 \mu s$ matches closely to the latency shown in the plot. The zero-padder and overlap-adder cause the additional latency seen in the simulation. Although the above is a specific example of filter length, sample rate, and fixed-point word width, the architecture is highly configurable and depending on the FPGA chosen for implementation, can process longer filters, achieve higher data rates, or represent numbers with greater precision in the computations. A longer filter allows recovering spectral features with finer resolution. Higher data rates allow use of a faster ADC and hence capture the increased bandwidth from a faster readout chirp rate, and greater precision in the computations provides more computational dynamic range.

FIXED-POINT WORD WIDTH CONSIDERATIONS

The FPGA implementation of the recovery algorithm uses fixed-point operations exclusively. Fixed-point numbers allow manipulating fractional values using integer math. Unlike a floating-point representation of a number, the binary point of a fixed-point number remains stationary. An FPGA processes data more efficiently with fixed-point number representations than with floating point math operations. Fixed-point multipliers and adders require fewer FPGA logic resources and run faster than their floating-point counterparts are capable of running. When using fixed-point math, the number of bits used in the computation affects the error present in the result. Two kinds of error are possible when performing fixed point operations. Quantization error occurs when truncating or rounding a value. Overflow occurs when a destination register is too small to contain the result of an operation. When performing a fixed-point multiply, the number of bits in the product is equal to the sum of the number of bits in multiplier and multiplicand. To avoid excessively large word sizes, it is usually necessary to quantize a product after multiplication. In a fixed-point addition, if a carry occurs on the most significant bit, the sum will contain one more bit than the number of bits in the largest operand, thus the sum may overflow the destination register.

The recovery processor quantizes intermediate values at four locations to limit the number of bits required to represent these values. Although this reduces the signal to noise ratio of the recovered result, it is necessary to allow the recovery architecture to fit in an FPGA with limited logic and memory resources. First, the architecture may scale

values immediately upon entering the FPGA. For example, the FPGA may quantize the 16 ADC bits down to 14 bits for use in recovery. The second location for quantization is immediately following the FFT processor. When the FFT is performed, an extra bit is added for each stage plus one additional bit. If the input to the FFT processor is 16 bits and the FFT has a length of $2^{12} = 4096$ then the output word size of the FFT processor will be 16+12+1 = 29 bits. Quantization can occur following the FFT core and prior to multiplication by the filter coefficients. The third quantization location is following the multiplication by the filter. The multiplier multiplies two complex numbers. Each word out of the multiplier is the sum or difference of the product of two of the inputs to the multiplier. If the four input operands are the same width, the product will contain two times the number of bits in each input operand plus an additional bit to account for a possible overflow in the addition. Thus if the operands input to the multiplier are 16 bits, the output words will each be 33 bits. Quantization can occur between the multiplier output and IFFT input. The fourth and final quantization location is at the output of the inverse DFT. Exactly as the forward DFT, the inverse DFT grows one bit per stage plus one additional bit and thus requires a quantization step to limit the output word size.

A simulation performed using Matlab's fixed-point toolbox assesses the effects of quantization in the recovery process. The simulation quantizes the data at each of the four previously mentioned locations to an equal number of bits. Then it varies the number of bits and filter-length while monitoring the signal to noise ratio (SNR) of the recovered signal to generate data that quantify the performance of the filter. The simulation is not entirely true to the FPGA processing because the simulation uses a floating point FFT followed by quantization. A fixed-point FFT would more accurately

simulate the real situation. Figure 20 is a screenshot of a data viewing graphical user interface (GUI) showing the SNR as a function of word size and filter length. Here, SNR is defined in the statistical sense as the variance of the signal divided by the variance of the noise and is expressed in dB, i.e. $SNR = 10\log_{10}\left(\frac{E[y_s[n]^2]}{E[y_n[n]^2]}\right)$, where E[] is the expected value that is calculated as the sample mean, y_s is the signal, and y_n is the noise. The numerator of the expression is the sample variance of the undistorted readout signal and the denominator is the sample variance of the difference or error between the recovered signal and the undistorted readout signal. This difference is not noise in the conventional sense, but rather a discrepancy between the two signals. The difference is correlated with the recovered signal.

The SNR surface plot resembles a plateau because of the limited readout bandwidth used for this simulation. The readout signal is band limited to 40 MHz and sampled at 100 MSPS prior to recovery in the simulation. This readout bandwidth limits the maximum SNR to approximately 37 dB. Increasing the filter length or using more bits to represent numbers in the computation will not improve the SNR beyond this point.

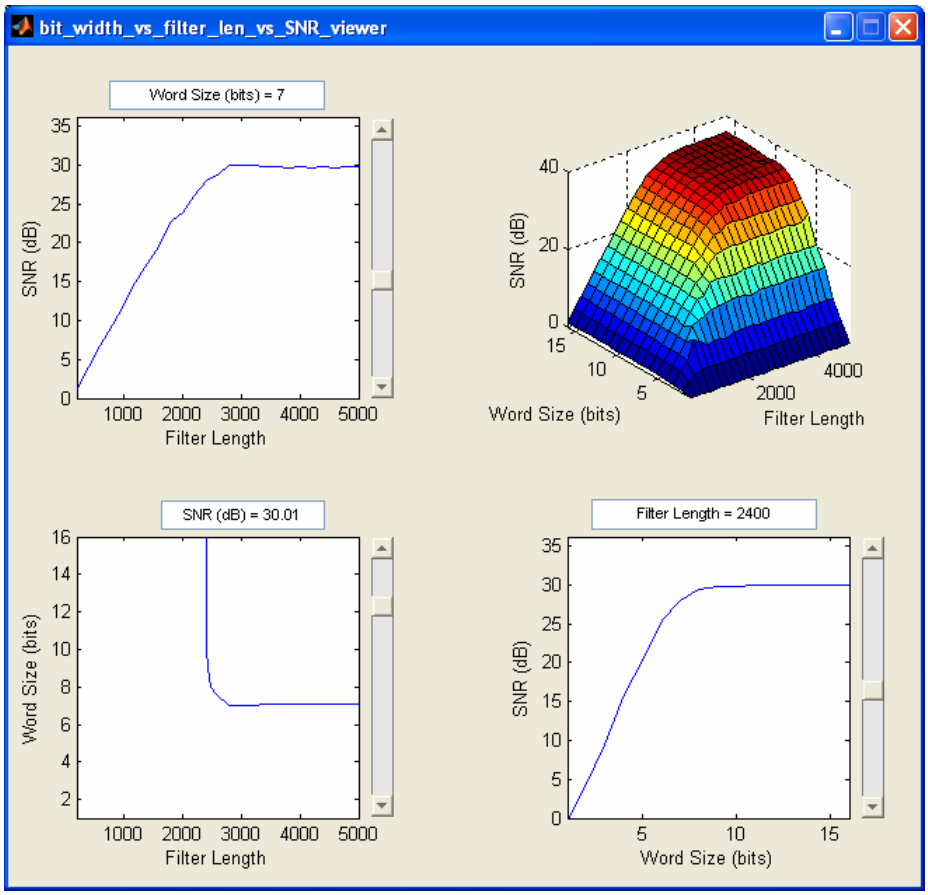

Figure 20: A simple data viewer showing the effects of filter length and fixed-point word width on recovery SNR

The surface plot in the upper right window of Figure 20 show the data collected during the simulation and the three other plots show a slice of the surface plot in all three dimensions. Using this data-viewing tool allows one to make optimal choices of filter length and word size to achieve a desired SNR in the computation. Making an optimal choice in filter length and bit width is necessary because of the high computational demands of this application. Choosing a word size or filter length too large will result in excessive computation and requiring an unnecessarily large and therefore costly FPGA. Using the above plots, we could choose a 2400 tap FIR filter using 7-bit math operations

to achieve a signal to noise ratio in the computation of approximately 30 dB. The theoretical SNR of a 7-bit ADC sampling a full-scale signal is $7bits \cdot 6.02dB/bit \approx 42dB$. Limited filter length, limited readout bandwidth, and multiple quantization steps in the recovery architecture, however, reduce the SNR below its theoretical maximum.

IMPLEMENTATION OF RECOVERY ARCHITECTURE IN HARDWARE

The processing architecture described above is implemented in a Xilinx Virtex 4 XC4VSX35-10, a medium density medium speed FPGA targeted at digital signal processing applications. This FPGA, two 14-bit 160 MSPS digital to analog converters (DAC), and two 14-bit 105 MSPS ADCs are on a development board made by Nallatech. The objective is to configure the recovery architecture such that it fully utilizes the available processing resources in the FPGA while maintaining an appropriate compromise between filter length, fixed-point word width, and input data rate. The available block ram in the FPGA limits the FFT core size and hence filter length. With the SX35, the largest FFT core able to fit along with the rest of the architecture into the FPGA is 8192 points and operates on 16 bit input data. This FFT size means the maximum filter length will be less than 8192 taps. How much less depends on the maximum achievable clock frequency and required ADC sample rate.

The recovery processor uses a pipelined streaming I/O FFT core developed by Xilinx [13] capable of processing an 8192-point FFT (or IFFT) every 8192 FPGA clock cycles. Since the recovery processor must complete this FFT within the time taken by the ADC to sample N_{SEG} points, where N_{SEG} is the number of samples in the segment of the overlap and add algorithm, the FFT core must be clocked at $\frac{8192}{N_{SEG}} F_{ADC}$ where F_{ADC} is the sample rate of the ADC.

Through the addition of pipeline-registers to the recovery architecture and trial and error with the FPGA development tool settings, the FFT core and the surrounding logic that operates at the high clock rate achieved a clock frequency of 206 MHz. Now the relationship between N_{SEG} and F_{ADC} is fixed at $\frac{8192}{N_{SEG}} F_{ADC} = 206MHz$, or equivalently $\frac{206MHz}{F_{ADC}} = \frac{8192}{N_{SEG}}$. The digital clock managers (DCM) in the FPGA generate the high-speed clock from the incoming ADC clock by scaling it by a rational number. The numerator and denominator of this scale factor are integers in the range 1 to 32. Thus this puts another constraint on the clock ratios and hence the ratio of FFT length to segment length. A convenient ratio to choose is two, meaning the ADC will run at 103 MSPS and the segment length will be 4096 samples. This implies that the filter length is 8192 – 4096 + 1 = 4097 taps. The sample rate is a high enough to obtain sufficient readout bandwidth, but the filter length is limiting at this rate as it endures for fewer than 40 μs. Therefore, a clock ratio of 16/7 is used because it increases the achievable filter length. With this ratio, the segment length is $8192\frac{7}{16} = 3584$, and thus the filter length is 8192 – 3584 + 1 = 4609 taps. Now the ADC operates at $206MHz\frac{7}{16} \approx 90$ MSPS. With this ratio, the filter is sufficiently long to capture the ringing due to a 25 kHz feature. Although the sample rate is just barely high enough to capture the readout bandwidth, it is anticipated that future optimization of the recovery processor such as adding more pipeline registers and floor planning will enable pushing the FPGA to a higher clock frequency.

Architecture Performance versus Direct Convolution

As noted above, performing convolution in the frequency domain via an FFT algorithm and block convolution enables impressive performance gains versus directly performing the convolution in the time domain. This section attempts to quantify those gains by comparing the performance of the FPGA implementation of the recovery algorithm with the performance of the same FPGA performing convolution directly in the time domain. To make the comparison, a merit function is required. The merit function used here is the product of the number of taps in the FIR filter and the ADC sample rate. This is also the number of MACs per second required to evaluate the filter in the time domain. Figure 21 graphically illustrates the remarkable FIR filter performance gains achieved by exploiting the efficiency of the FFT algorithm with the recovery architecture. The actual MAC/s value is achieved by multiplying the number of hardwired multiply accumulate units, 192, in the FPGA by the maximum clock rate of the MAC units, 400 MHz in this device. Similarly, the equivalent MAC/s value is the product of the filter length, 4609 taps, and the achieved filter input rate of 90 MHz.

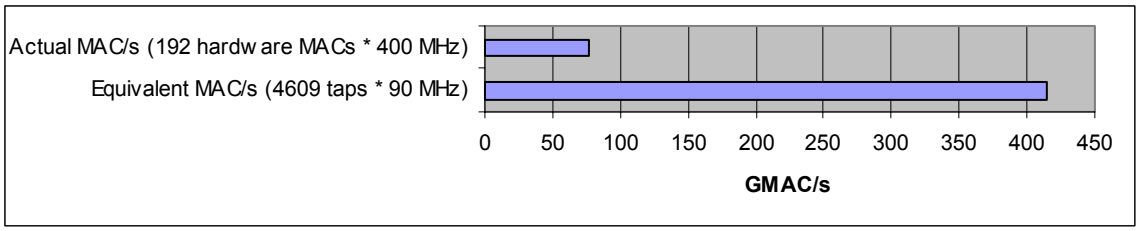

Figure 21: Comparison of Equivalent GMACs/s of the Recovery Architecture vs. Actual GMACs/s for when evaluating FIR filters in the V4SX35-10 FPGA

Note that the hardwired MAC units have 18 bit input operands and 48-bit accumulators while the recovery architecture only maintains 16-bit precision throughout all processing operations.

Filter Configuration

In many situations, it is desirable to change the readout chirp rate. Increasing the chirp rate reduces readout latency and decreasing the chirp rate reduces the readout bandwidth allowing the recovery processor to resolve narrower features with a given readout bandwidth. A different chirp rate requires a different set of filter coefficients to perform recovery. Therefore, an easy means of updating the filter is required. Xilinx FPGAs provide dual asynchronous read and write ports to every block RAM in the device. This provides a convenient way to update filter coefficients. The recovery processor reads the filter coefficients at the high clock rate through one of the ports in the block RAM. If a filter update is required, a Picoblaze microcontroller embedded in the FPGA fabric receives filter coefficients via a serial connection to a PC and writes the new coefficients into the block RAM through the second port at the slow clock rate. Figure 22 is a block diagram showing the interface between the PC and the filter coefficient RAM.

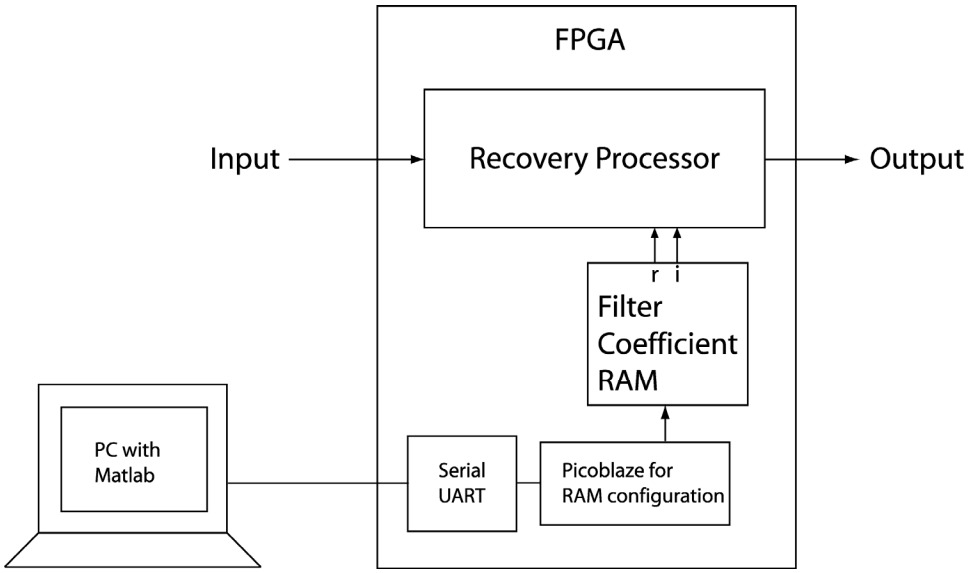

Figure 22: Overview of Filter Coefficient Update Scheme

EXPERIMENTAL RESULTS

Two sets of experiments were performed to assess the performance of the recovery processor. The first evaluates how the system performs on readout signals directly from a photo-detector during an S2 spectral analysis experiment. The second tests the FPGA alone on simulated readout data to quantify the recovery performance further.

Experimental Setup One

In the first set of experiments, depicted in Figure 23, an the RF electronics alternate between producing an analog waveform to analyze and a linear frequency chirp to read the spectrum of the analog waveform out of the S2 material.

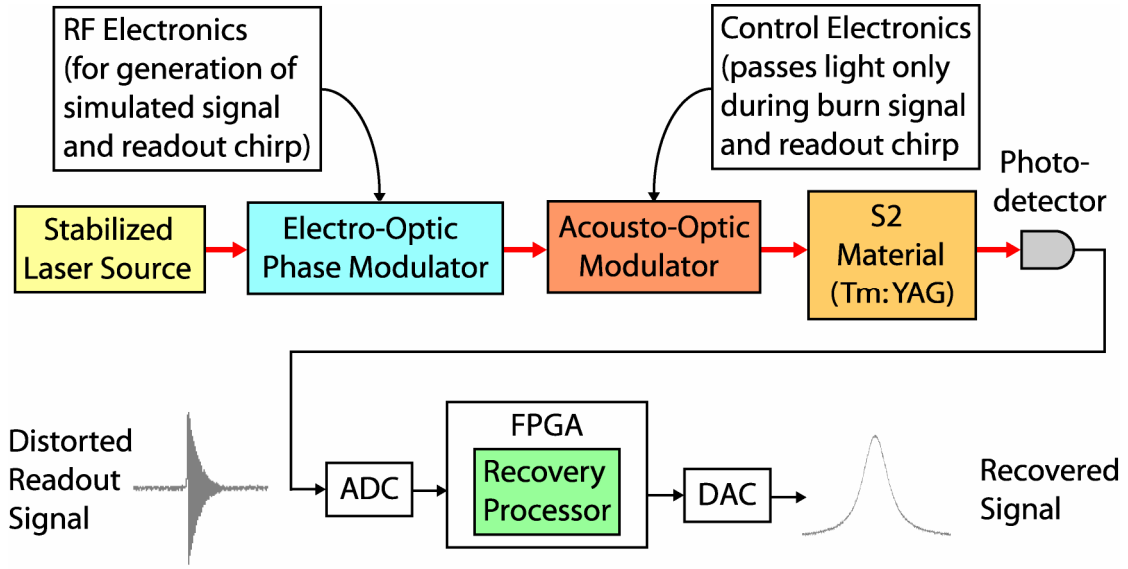

Figure 23: Experimental setup for testing recovery processor.

The electro-optic phase modulator (EOPM) modulates the electrical signal onto a stabilized laser source. This modulated laser beam then irradiates the S2 material saving the spectrum of the signal as an absorption profile. A photo-detector converts the optical signal passing through the crystal back into an electrical signal that the ADC captures at 90 MSPS. Finally, the recovery processor convolves the readout signal with a 4609 tap filter removing the phase distortion caused by fast readout.

Burning a spectral hole at 262 MHz and then performing a frequency-chirped readout across the feature at 0.5 MHz/µs produces a signal such as the typical example depicted in Figure 24. The readout signal is then recovered using both Matlab and the FPGA recovery processor.

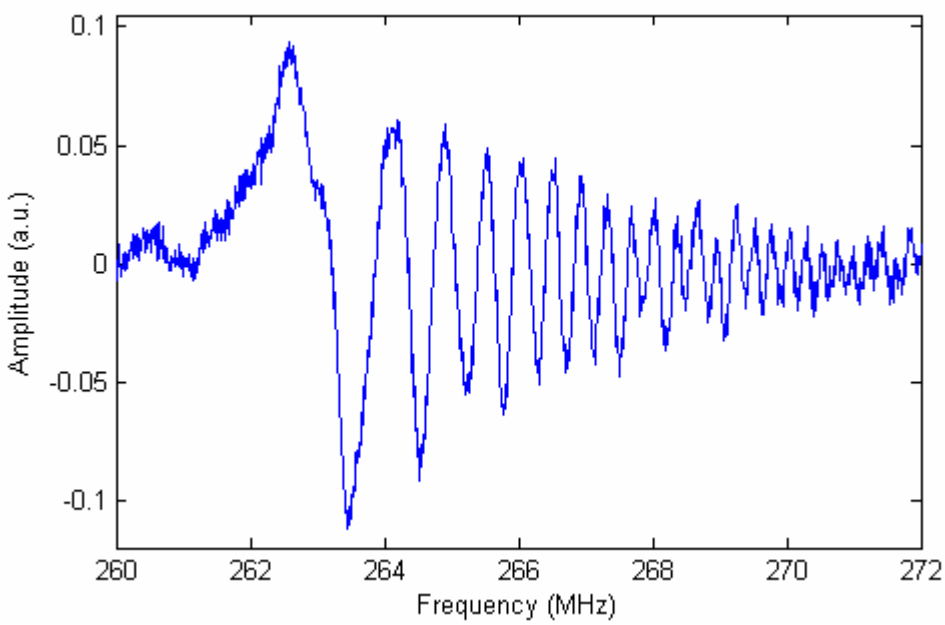

Figure 24: Three typical experimental hole readouts at 0.5 MHz/µs

Figure 25 shows the readout signal recovered with (a) Matlab and (b) the FPGA. The FPGA recovered signal contains slightly less high frequency noise due to an anti-aliasing

filter with a cutoff of 58 MHz applied to the signal prior to recovery. Both methods produce a recovered hole with a width of approximately 192 kHz.

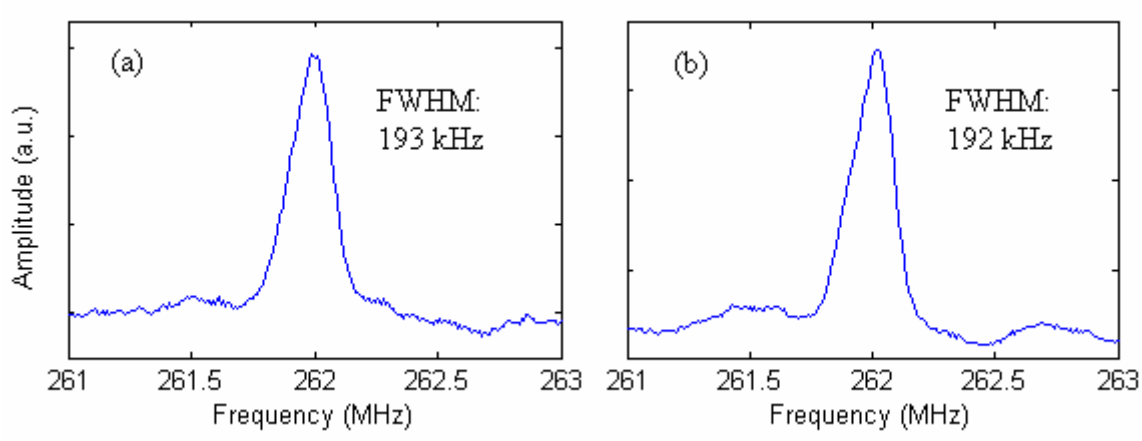

Figure 25: Feature read out at 0.5 MHz/μs recovered with (a) Matlab and (b) the FPGA.

Next, increasing the readout chirp rate from 0.5 MHz/μs to 1 MHz/μs tests how the FPGA recovery processor performs with increased readout bandwidth. Figure 26 shows a representative readout example of chirping at 1 MHz/μs over a hole centered at 145 MHz.

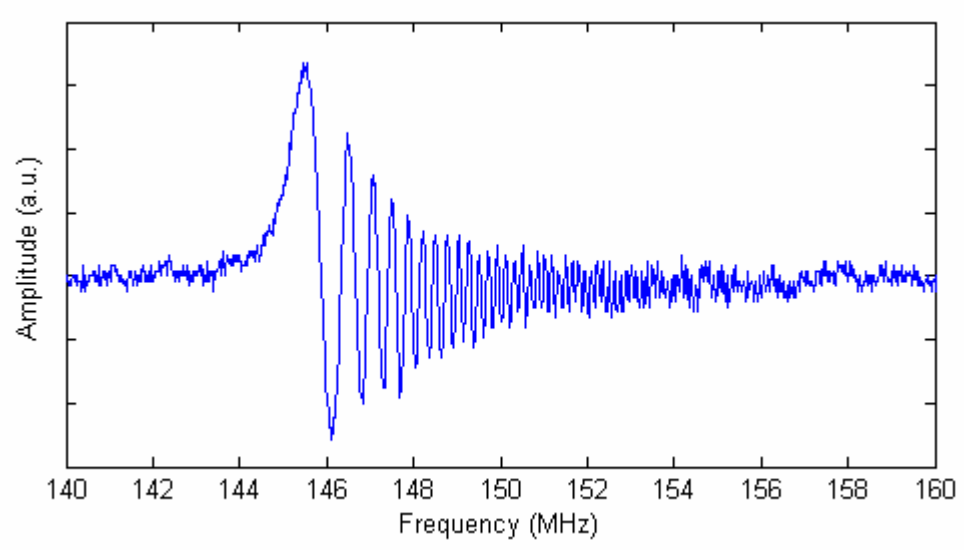

Figure 26: Typical experimental hole readouts at 1 MHz/μs

Recovering the holes from the fast-chirped readout with Matlab produces the plot seen in Figure 27(a), the FPGA in Figure 27(b).

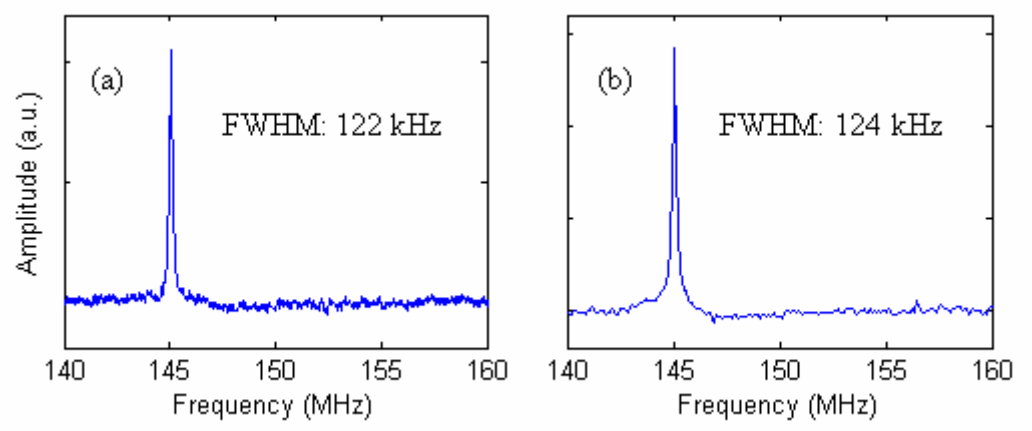

Figure 27: Feature read out at 1 MHz/μs recovered with (a) Matlab and (b) the FPGA

With the increased chirprate, the recovered result from the FPGA again closely matches the result obtained with Matlab.

Finally, a signal consisting of multiple spectral features is burned into the crystal, read out, and recovered. The signal consists of two strong features at 261.8 MHz and 262.2 MHz combined with two weaker features at 261.4 MHz and 262.6 MHz. Burning this signal into the crystal and reading out at 0.5 MHz/μs produces the temporal map plotted as a function of chirp frequency as shown in Figure 28.

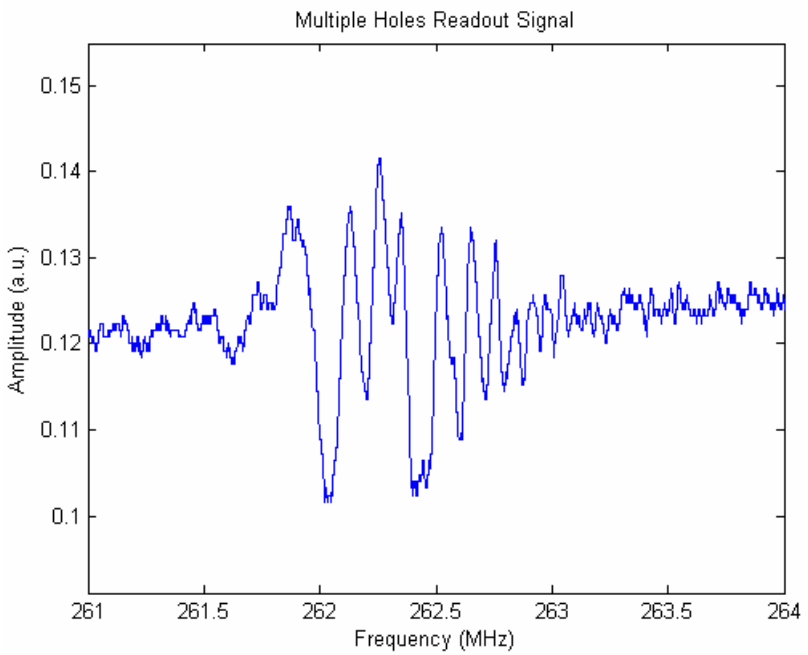

Figure 28: Readout signal from spectrum with multiple holes.

For comparison, the readout signal is recovered with both Matlab and the FPGA recovery processor as seen in Figure 29. The strong and weak features are clearly present at the correct frequencies in both recovered spectra showing that the recovery processor works well for arbitrary spectra.

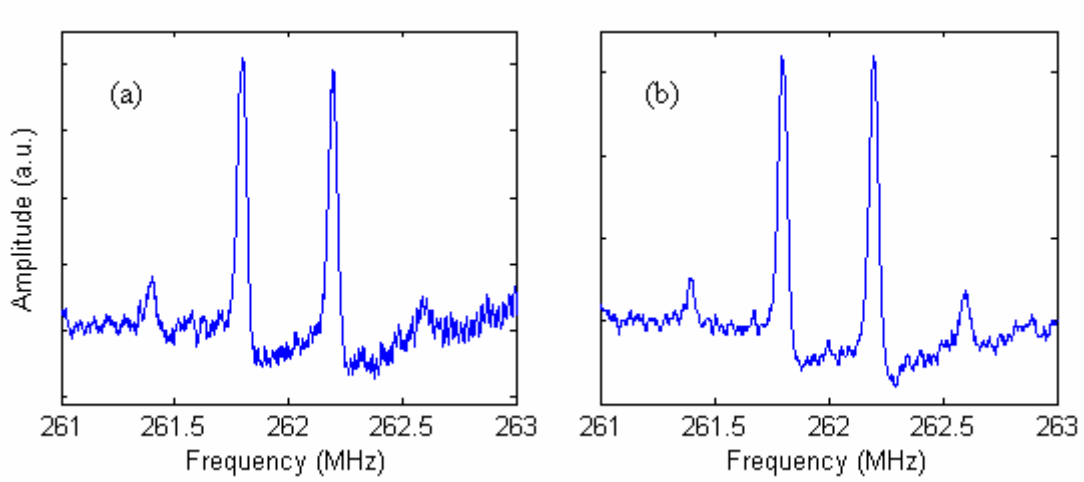

Figure 29: Multiple features read out at 0.5 MHz/μs recovered with (a) Matlab and (b) the FPGA

Experimental Setup Two

In the second experiment, rather than sampling a signal directly from the photo-detector during a hole-burning experiment, a simulated readout signal [2] is stored in read-only memory (ROM) in the FPGA as shown in Figure 30. This experiment allows for an accurate measurement of processing latency and recovered feature width without the extra noise associated with a hole-burning experiment or the bandwidth limit imposed by the AOM.

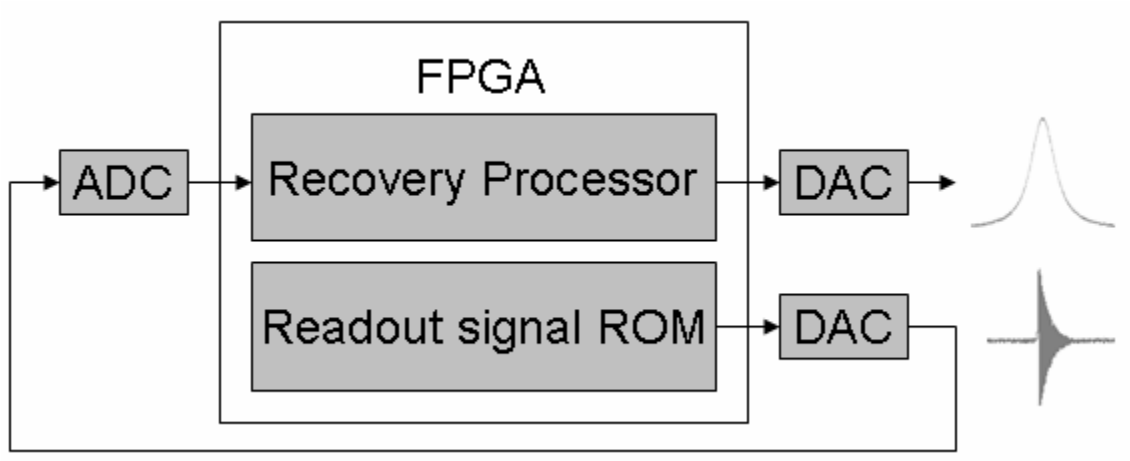

Figure 30: Setup for recovering a simulated hole readout with the FPGA

The ROM stores simulated data representing the ringing due to a 25 kHz wide hole read out at 1 MHz/μs, bandlimited to 45 MHz, and sampled at 90 MSPS. In this experiment, the FPGA input also operates at 90 MHz to show how the processor will perform when reading out 25 kHz features at 1 MHz/μs in an actual hole-burning experiment.

First, simultaneously capturing both the readout signal and the recovered signal reveals the processing latency. In Figure 31, one can see both signals and the corresponding 309.8 µs latency. This is well within the 1 ms latency limit requirement.

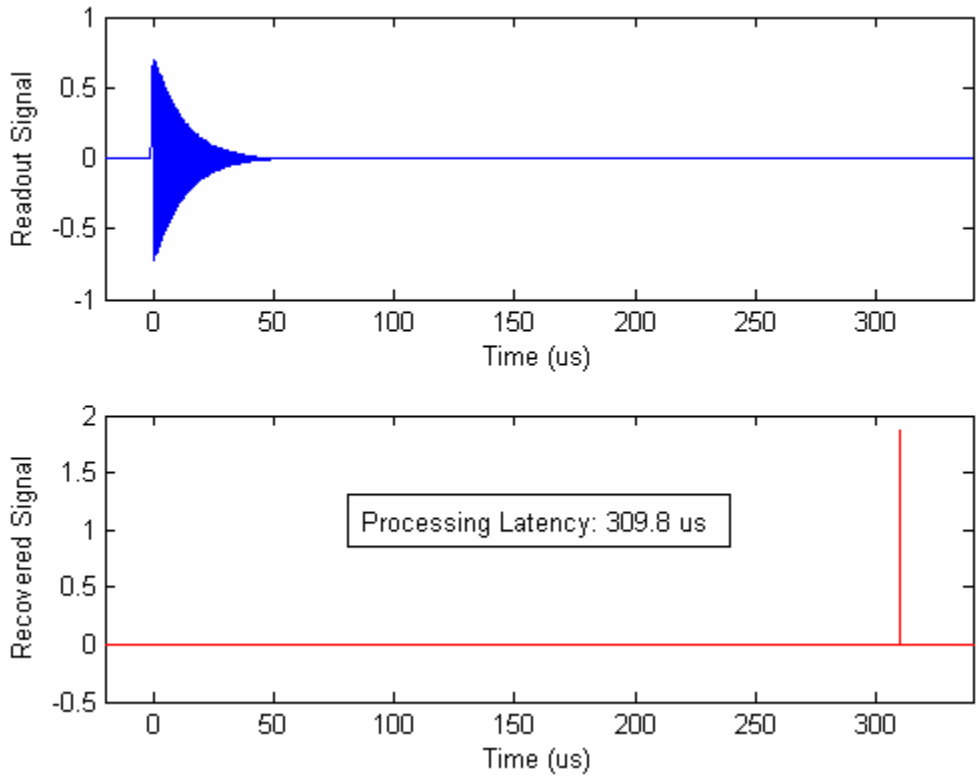

Figure 31: Oscilloscope captured traces (with averaging turned on) of FPGA performing recovery of a 25 kHz wide hole read out at 1 MHz/µs and sampled at 90 MSPS. As shown, the processing latency is 309.8 µs.

By zooming in on the recovered hole, one can make an accurate measurement of the hole-width. Figure 32 shows the recovery processor achieving a recovered hole-width of 27.83 kHz from a 25 kHz hole read out at 1 MHz/µs.

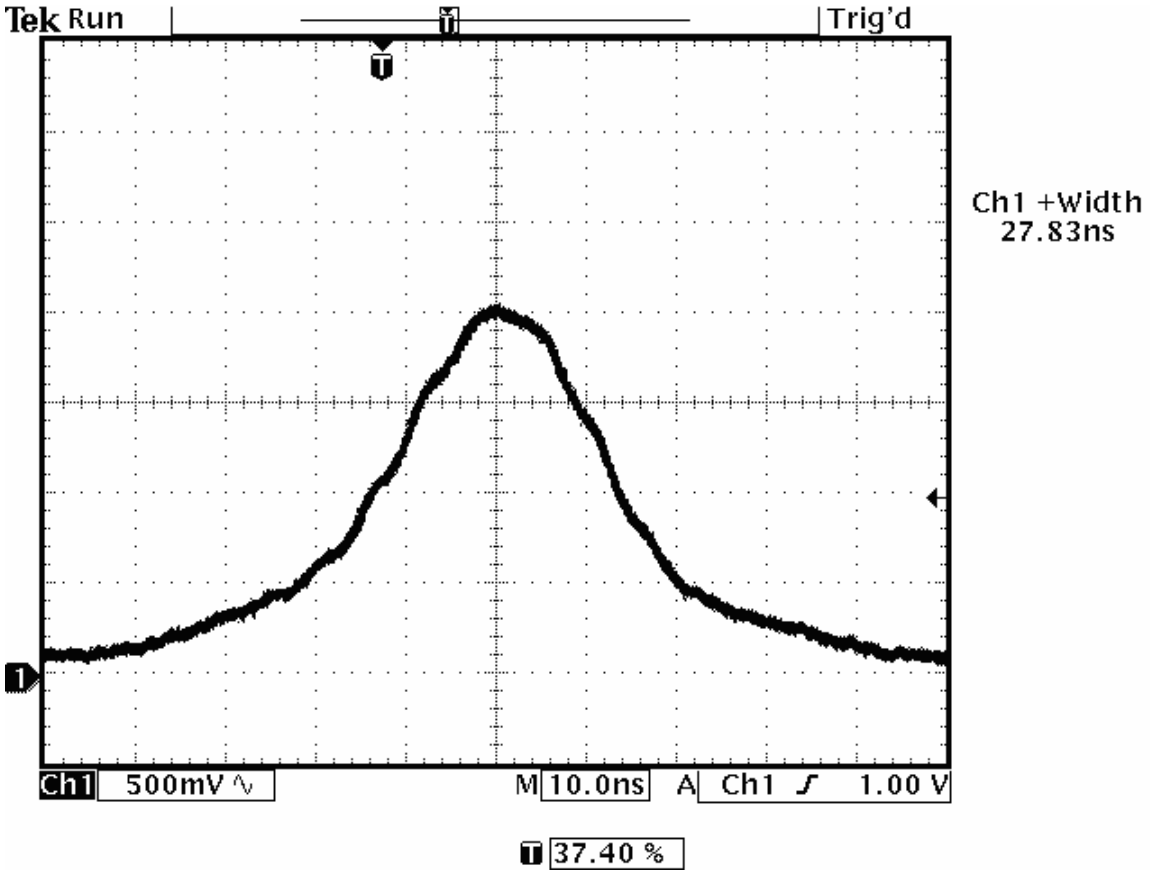

Figure 32: Recovered feature from FPGA as captured by an oscilloscope (no averaging, full bandwidth). FWHM Γ = 27.83 ns * 1 MHz/μs = 27.83 kHz.

The width of the hole recovered by the FPGA closely matches the 27.8 kHz width predicted by the simulation when the readout bandwidth is limited to 45 MHz as shown in Figure 7.

CONCLUSION

For the first time, continuous real-time removal of fast-chirped readout phase distortion is achievable with only a short processing latency. The processor developed here recovers spectral features quickly enough, and with sufficient resolution, to enable finding and using open communication channels in a crowded RF environment. This work brings S2 based spectral analysis one-step closer to a practical implementation.

Electronic noise on the FPGA development board limits the dynamic range of the processor. Noise on the DAC output increases to several millivolts when connecting it to the ADC input, indicating that the ADC is contributing the noise. This severely limits the achievable dynamic range of the recovery processor. Future work should address this issue and optimize numerical scaling in the recovery processor to maximize the processing dynamic range. In addition, employing more pipelining or device floor planning in the processor will increase the input data rate. Ideally, this rate could be increased to the 105 MHz limit imposed by the ADC. This will improve the recovered feature resolution by enabling the processor to capture more readout bandwidth.

Experimental results are encouraging, showing real-time recovery is a viable solution to the fast-chirped readout distortion problem. Additional testing of and modification to this device should ultimately lead to a practical recovery system.

APPENDICES

APPENDIX A

FPGA BASED LINEAR FREQUENCY CHIRP GENERATOR

Background

Some modern FPGAs contain high-speed parallel to serial converters that provide serial bit rates up to 10 Gb/s. Although these serializers are intended for communication, they are generic enough for generation of arbitrary binary waveforms. Because of the high bit rate of the serializers and the large amount of processing resources provided by the FPGAs, the combination of the two provides a possible solution to the broadband chirp generation problem.

Chirp Generation

This chirp generator implementation combines ideas of frequency synthesis using high-speed serializers [14] with those of direct digital synthesizers (DDS) [7], to create linear frequency chirps intended to read spectral features out of S2 materials. The chirp generator uses frequency and phase accumulators arranged as in Figure 33. Commercial direct digital synthesizers commonly use this configuration to generate a linear frequency chirp with continuous phase. The serializers only provide one bit of vertical resolution and thus produce binary square-wave chirps rather than ideal sinusoidal chirps. Therefore, only a comparator is required as shown in Figure 33, rather than a complete sine lookup table following the phase accumulator register.

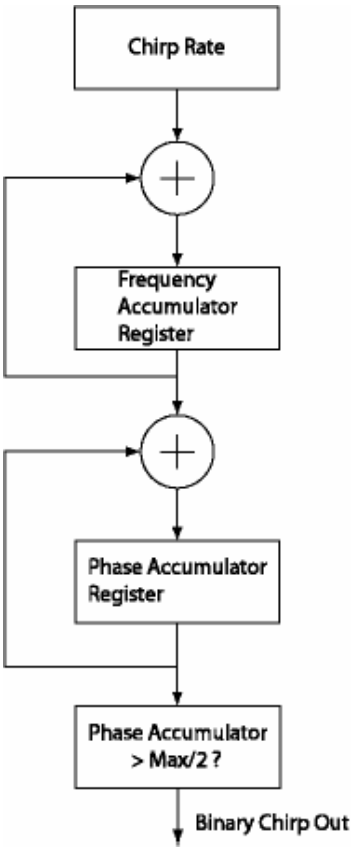

Figure 33: Structure to generate one bit of the serializer input

Although the chirp signal produced from the serializer is a square wave rather than a sinusoid, it can still be used for readout. One either ignores the distortion introduced by the harmonics or restricts the chirp to a sub-octave frequency band and low pass filters it to remove the higher harmonics prior to presenting it to the S2 material.

Because the serializers take in N bits in parallel and output these N bits serially on every FPGA clock cycle, it is necessary to extend slightly the DDS concept. Instead of using a single phase and frequency accumulator, N of them are used with each of their outputs connecting to one of the parallel inputs of the serializer as shown in Figure 34.

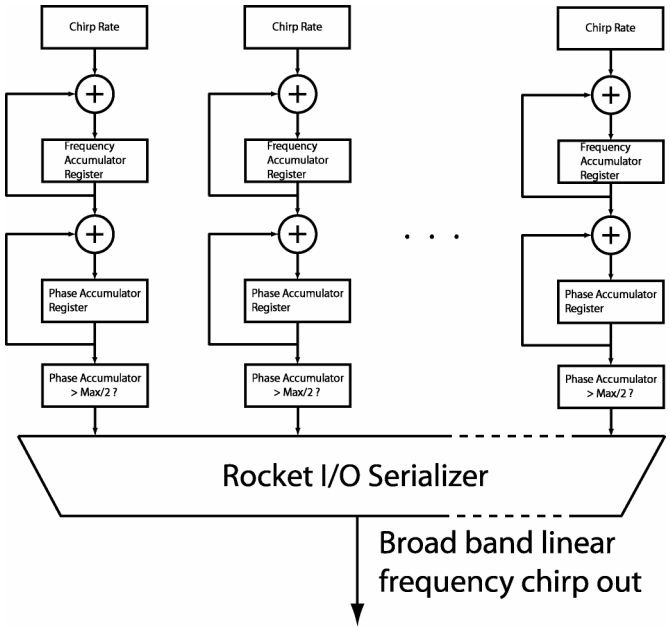

Figure 34: Parallel connection of multiple chirp generators to the serializer

By initializing the registers such that consecutive phase accumulators are quadratically shifted from each other and consecutive frequency accumulators are linearly shifted from each other, it is possible to make the binary chirp generators work together creating a single linear frequency chirp out of the serializer at the high serial data rate.

The chirp generator VHDL, given in appendix C, implements a state machine with two states. When in the initialization state, all phase and frequency accumulators are set to their initial values, and the output is zero. Upon receiving a chirp start signal, the state machine enters the sending chirp state in which the phase and frequency registers are incremented, and the contents of the phase accumulators determine the output.

Chirp Simulation

Simulating the architecture shown in Figure 34 ensures proper behavior. In the simulation, the chirp generator creates a chirp from 50 MHz to 5 GHz at 10 MHz/μs using a data rate of 10 Gb/s. Figure 35 shows segments of the generated chirp waveform for various times during the chirp.

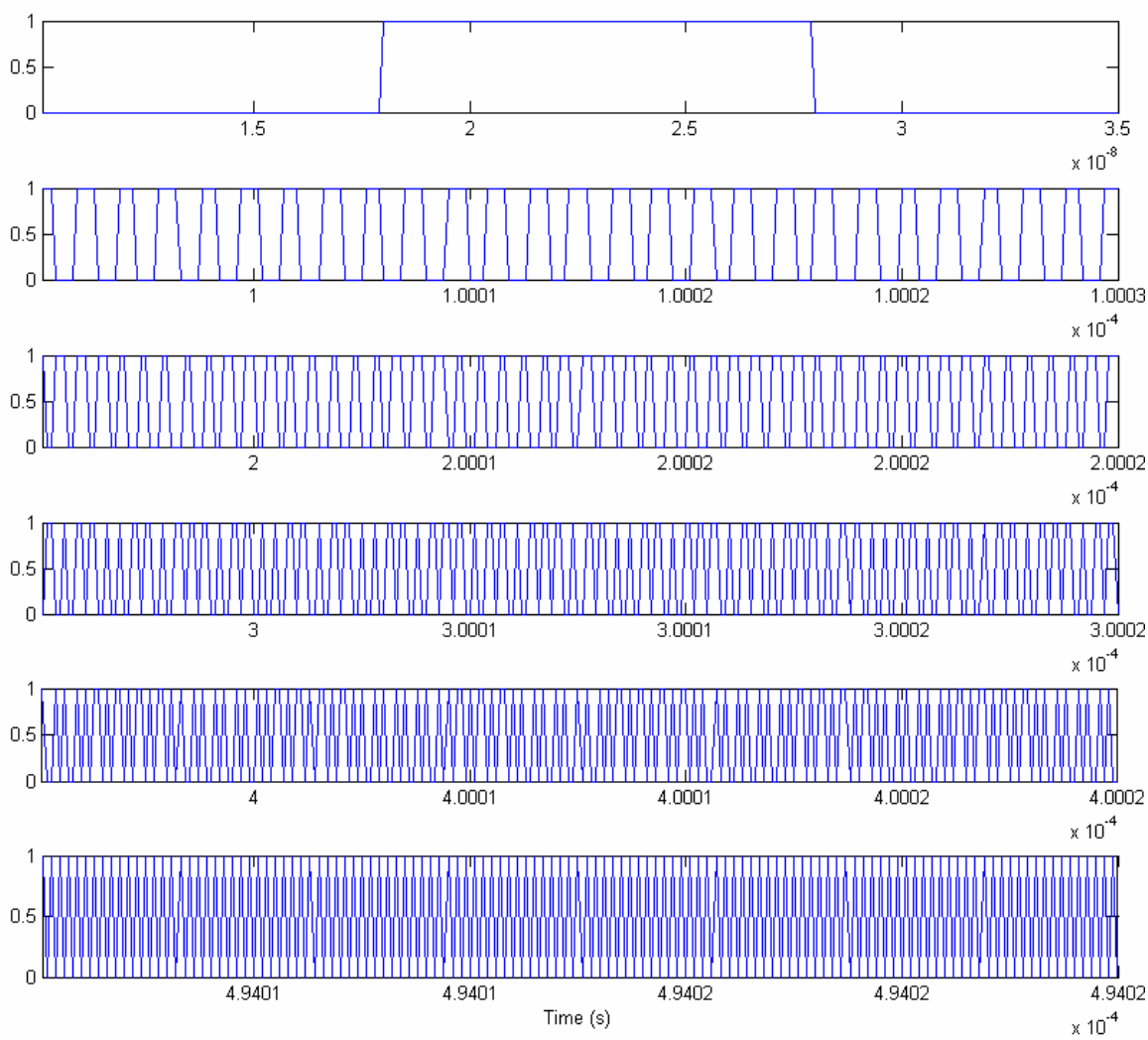

Figure 35: Time domain chirp data generated with VHDL simulator

In Figure 36, one can see a spectrogram generated from the chirp waveform data. On the spectrogram, the fundamental frequency chirps linearly from 50 MHz to 5 GHz in 495 μs. Odd harmonics are also seen, as this is a square-wave rather than a single sinusoid.

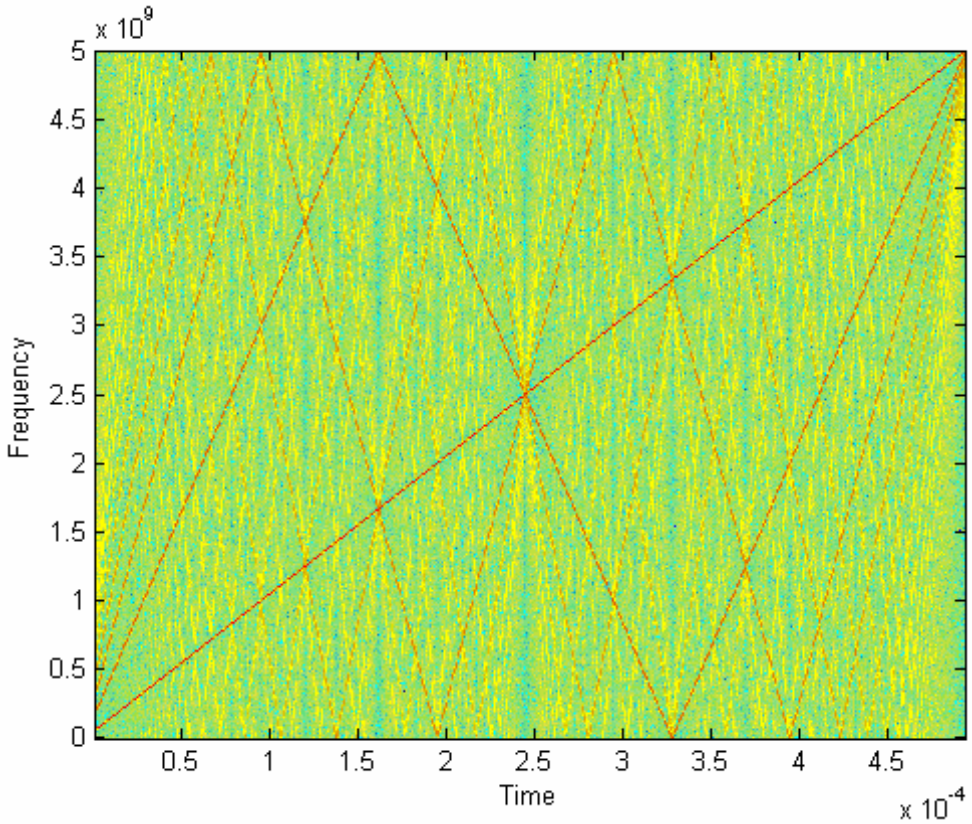

Figure 36: Spectrogram of chirp generated during VHDL code simulation

Taking the power spectral density of this chirp from 500 MHz to 700 MHz reveals more detail about the relative amplitude of the harmonics as shown in Figure 37.

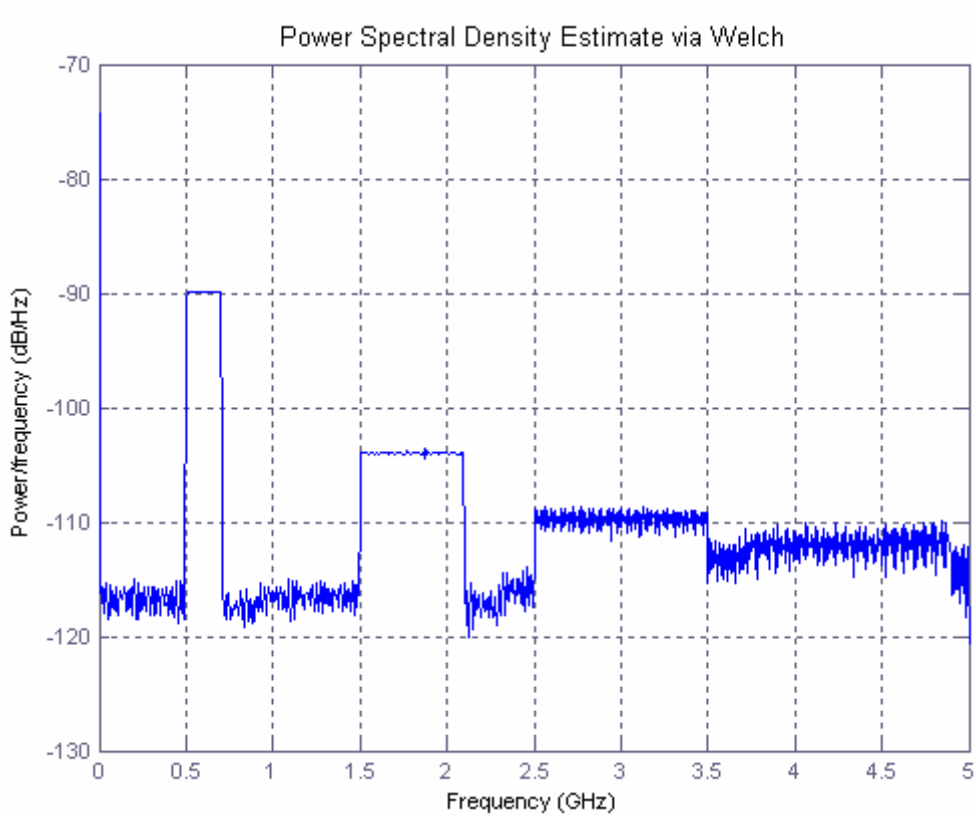

Figure 37: PSD of VHDL simulated chirp from 500 MHz to 700 MHz

The time-domain chirp waveform and the spectrogram verify the functionality of the architecture in Figure 34 as well as the correct behavior of the VHDL.

Experimental Chirp Results

Figure 38 shows a spectrogram generated from the data captured by an oscilloscope connected to the FPGA set to generate a chirp from 50 MHz to 600 MHz at 20 MHz/µs. As shown, the digitally generated chirp is linear and at the correct rate. One major difference between this spectrogram and the spectrogram of the chirp generated

with the simulator is the appearance of the second harmonic. The second harmonic is relatively strong here, but it is absent from an ideal square wave.

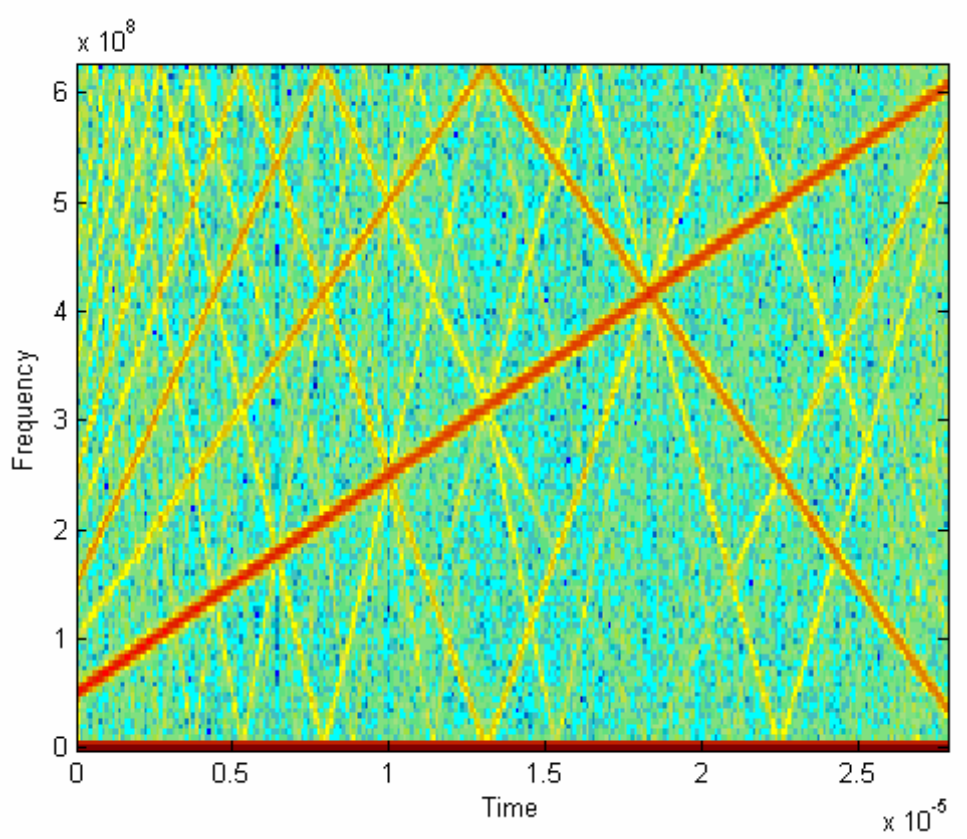

Figure 38: Spectrogram created with experimental data captured with an oscilloscope

Sending the chirp generator output to a spectrum analyzer more clearly shows the relative amplitudes of the harmonics. Figure 39 shows the chirp spectrum captured by the spectrum analyzer with the FPGA configured to chirp from 500 MHz to 700 MHz.

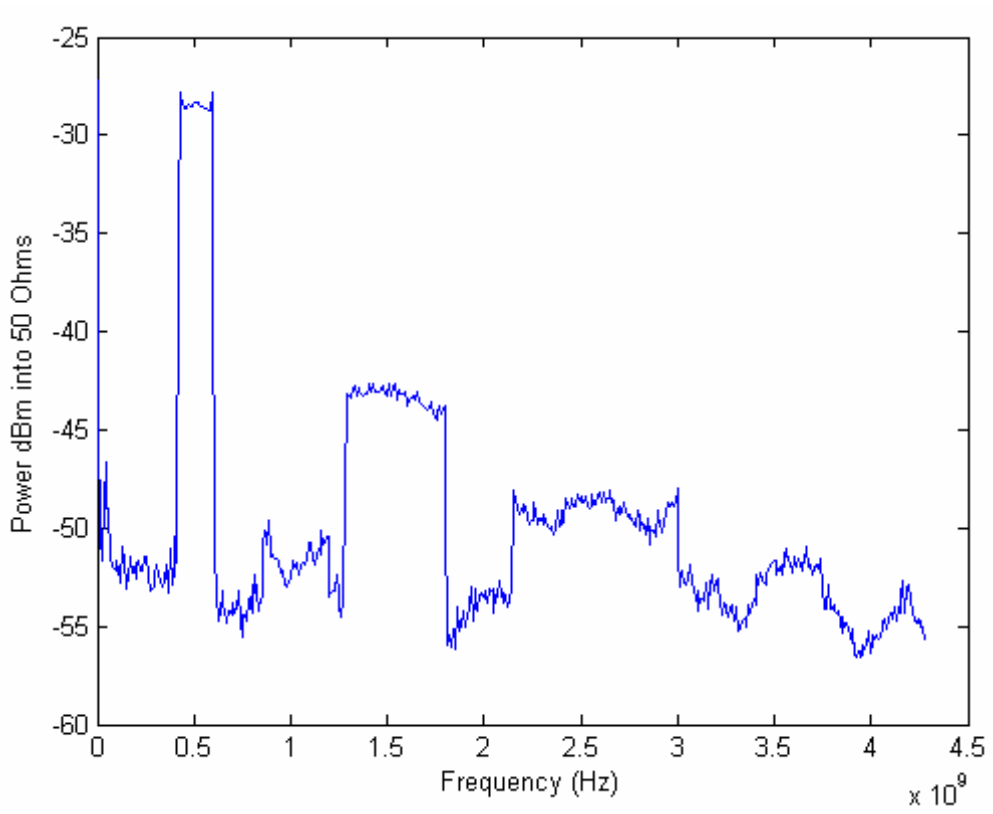

Figure 39: Chirp spectrum as captured by a spectrum analyzer

The experimental PSD shown in Figure 39 is similar to the simulation generated PSD of Figure 37. In the experimental PSD, however, the second harmonic is clearly visible. This work shows that FPGAs with high speed serial I/O provide a viable solution to the readout chirp generation problem.

APPENDIX B

RECOVERY PROCESSOR VHDL

Recovery Processor Top

```
-----------------------------------------------------------------------
-- overlap_conv_add_top.vhd
--
-- Connects zero-padder, circular convolver, and overlap-adder
-- components together in a top level file.  Output is the linear
-- convolution of the input with the stored filter.
--
-----------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity overlap_conv_add_top is
    Port (  slow_clk_in : in std_logic;
            fast_clk_in : in std_logic;
            data_in : in std_logic_vector(15 downto 0);
            data_out : out std_logic_vector(15 downto 0);
            ADC_data1 : out std_logic_vector(15 downto 0);
            ADC_data2 : out std_logic_vector(15 downto 0);
            leds : out std_logic_vector(2 downto 0);
            filt_ram_addr : in std_logic_vector(12 downto 0);
            filt_ram_data : in std_logic_vector(31 downto 0);
            filt_ram_we : in std_logic);
end overlap_conv_add_top;

architecture Behavioral of overlap_conv_add_top is

signal zero_padder_to_convolver_sig : std_logic_vector(15 downto 0);
signal convolver_to_ov_adder_sig : std_logic_vector(15 downto 0);
signal send_input_sig, send_output_sig : std_logic;
signal fast_clk, slow_clk, slow_clk_int, fast_clk_int : std_logic;
signal xk_index_sig : std_logic_vector(12 downto 0);

component ZeroPad is
    Port (  fast_clk : in std_logic;
            slow_clk : in std_logic;
            data_in : in std_logic_vector(15 downto 0);
            data_out : out std_logic_vector(15 downto 0);
            start_sequence : in std_logic;
            ADC_data1 : out std_logic_vector(15 downto 0));
end component;

component convolver is
    Port (  clk : in std_logic;
            slow_clk : in std_logic;
            data_in : in std_logic_vector(15 downto 0);
            data_out : out std_logic_vector(15 downto 0);
            xk_index_out : out std_logic_vector(12 downto 0);
```

```
            ready_for_input : out std_logic;
            ADC_data1 : out std_logic_vector(15 downto 0);
            ADC_data2 : out std_logic_vector(15 downto 0);
            leds : out std_logic_vector(2 downto 0);

            filt_ram_addr : in std_logic_vector(12 downto 0);
            filt_ram_data : in std_logic_vector(31 downto 0);
            filt_ram_we : in std_logic);
end component;

component OverlapAdd is
    Port ( fast_clk : in std_logic;
           slow_clk : in std_logic;
           data_in : in std_logic_vector(15 downto 0);
           data_out : out std_logic_vector(15 downto 0);
           xk_index_sig : in std_logic_vector(12 downto 0);
           ADC_data1 : out std_logic_vector(15 downto 0));
end component;

COMPONENT my_dcm
PORT(
            CLKIN_IN : IN std_logic;
            CLKFX_OUT : OUT std_logic;
            CLKIN_IBUFG_OUT : OUT std_logic;
            CLK0_OUT : OUT std_logic;
            LOCKED_OUT : OUT std_logic
      );
END COMPONENT;

begin
data_out <= data_out_sig;

fast_clk_int <= fast_clk_in;
slow_clk_int <= slow_clk_in;

zeropad1: ZeroPad
Port map(   fast_clk => fast_clk_int,
            slow_clk => slow_clk_int,
            data_in => data_in,
            data_out => zero_padder_to_convolver_sig,
            start_sequence => send_input_sig,
            ADC_data1 => open);

convolver1: convolver
Port map(   clk => fast_clk_int,
            slow_clk => slow_clk_int,
            data_in => zero_padder_to_convolver_sig,
            data_out => convolver_to_ov_adder_sig,
            xk_index_out => xk_index_sig,
            ready_for_input => send_input_sig,
            ADC_data1 => open,
            ADC_data2 => ADC_data2,
            leds => leds,

            filt_ram_addr => filt_ram_addr,
```

```
            filt_ram_data => filt_ram_data,
            filt_ram_we => filt_ram_we);

overlapadd1: OverlapAdd
Port map(   fast_clk => fast_clk_int,
            slow_clk => slow_clk_int,
            data_in => convolver_to_ov_adder_sig,
            data_out => data_out_sig,
            xk_index_sig => xk_index_sig,
            ADC_data1 => open);

end Behavioral;
```

Zero Padder

```
-----------------------------------------------------------------------
-- zeropad.vhd
--
-- The zero padder takes in a continuous stream of data at the ADC
-- rate, breaks it into segments, and pads each segment with zeros.  It
-- outputs these zero-padded segments at the rate of the FFT core,
-- 16/7*ADC rate in this case.
--
-----------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ZeroPad is
    Port ( fast_clk : in std_logic;
           slow_clk : in std_logic;
           data_in : in std_logic_vector(15 downto 0);
           data_out : out std_logic_vector(15 downto 0);
           start_sequence : in std_logic;
           ADC_data1 : out std_logic_vector(15 downto 0));
end ZeroPad;

architecture Behavioral of ZeroPad is

signal f1_rd_en_sig : std_logic;
signal f1_data_out_sig : std_logic_vector(15 downto 0);

constant L : integer := 3584; --segment size
signal dataout_cnt : integer range 0 to 8191 := L;

constant fft_len : integer := 8192; --required FFT length
constant M : integer := fft_len-L+1; --Filter length M = fft_len-L+1
```

```
component fifo
      port (
      din: IN std_logic_VECTOR(15 downto 0);
      rd_clk: IN std_logic;
      rd_en: IN std_logic;
      rst: IN std_logic;
      wr_clk: IN std_logic;
      wr_en: IN std_logic;
      dout: OUT std_logic_VECTOR(15 downto 0);
      empty: OUT std_logic;
      full: OUT std_logic;
      rd_data_count: OUT std_logic_VECTOR(12 downto 0);
      wr_data_count: OUT std_logic_VECTOR(12 downto 0));
end component;
component fifo3
      port (
      din: IN std_logic_VECTOR(15 downto 0);
      wr_en: IN std_logic;
      wr_clk: IN std_logic;
      rd_en: IN std_logic;
      rd_clk: IN std_logic;
      ainit: IN std_logic;
      dout: OUT std_logic_VECTOR(15 downto 0);
      full: OUT std_logic;
      empty: OUT std_logic;
      wr_count: OUT std_logic_VECTOR(12 downto 0);
      rd_count: OUT std_logic_VECTOR(12 downto 0));
end component;
component a_fifo_2k
      port (
      din: IN std_logic_VECTOR(15 downto 0);
      rd_clk: IN std_logic;
      rd_en: IN std_logic;
      rst: IN std_logic;
      wr_clk: IN std_logic;
      wr_en: IN std_logic;
      dout: OUT std_logic_VECTOR(15 downto 0);
      empty: OUT std_logic;
      full: OUT std_logic);
end component;
begin
ADC_data1(15 downto 12) <= (others => '0');

fifo1 : a_fifo_2k
            port map (
                  din => data_in,
                  rd_clk => fast_clk,
                  rd_en => f1_rd_en_sig,
                  rst => '0',
                  wr_clk => slow_clk,
                  wr_en => '1',
                  dout => f1_data_out_sig,
                  empty => open,
                  full => open);
```

```
fast_clock_ctrl: process(fast_clk)
begin
      if rising_edge(fast_clk) then
            if f1_rd_en_sig = '1' then
                  data_out <= f1_data_out_sig;
            else
                  data_out <= (others => '0');
            end if;

            if dataout_cnt > L-1 then
                  f1_rd_en_sig <= '0';
            else
                  f1_rd_en_sig <= '1';
            end if;

            --synchronize dataout_cnt to start_sequence pulse
            if start_sequence = '1' or dataout_cnt = 8191 then
                  dataout_cnt <= 0;
            else
                  dataout_cnt <= dataout_cnt + 1;
            end if;

      end if;
end process fast_clock_ctrl;

end Behavioral;
```

Circular Convolver

```
-----------------------------------------------------------------------
-- convolver.vhd
--
-- Performs circular convolution of its input with a filter stored in a
-- RAM.  Uses a streaming I/O FFT core from Xilinx to perform the DFT
-- and IDFT.  Filter updates are allowed by writing to the RAM storing
-- the filter.
--
-----------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;


entity convolver is
```

```
    sss
    Port ( clk : in std_logic;
           slow_clk : in std_logic;
           data_in : in std_logic_vector(15 downto 0);
           data_out : out std_logic_vector(15 downto 0);
           xk_index_out : out std_logic_vector(12 downto 0);
           ready_for_input : out std_logic;
           ADC_data1 : out std_logic_vector(15 downto 0);
           ADC_data2 : out std_logic_vector(15 downto 0);
           leds : out std_logic_vector(2 downto 0);

           filt_ram_addr : in std_logic_vector(12 downto 0);
           filt_ram_data : in std_logic_vector(31 downto 0);
           filt_ram_we : in std_logic);
end convolver;

architecture Behavioral of convolver is

--constants defining data widths and FFT length parameters
-- note, some constants have been replaced with their actual values in
-- this document to allow it to fit the page width
constant word_size: integer := 16;
constant fft_len: integer := 8192;
constant log2_fft_len: integer := 13;
constant fft_in_word_size: integer := 16;
constant fft_out_word_size: integer := 30;
constant product_word_size: integer := 33;

--signal declarations for connecting components
signal fft2cmult_real_fifo_din_sig : std_logic_vector(15 downto 0);
signal fft2cmult_imag_fifo_din_sig : std_logic_vector(15 downto 0);

signal fft2cmult_fifo_rd_en_sig : std_logic := '0';
signal fft2cmult_fifo_wr_en_sig : std_logic := '0';
signal fft2cmult_fifo_rd_en_sigR : std_logic;
signal fft2cmult_fifo_wr_en_sigR : std_logic;

signal fft_r_din_sig, fft_i_din_sig : std_logic_vector(15 downto 0);
signal fft_r_dout_sig, fft_i_dout_sig : std_logic_vector(29 downto 0);
signal xk_re, xk_im : std_logic_vector(29 downto 0);
signal fwd_inv_sig, fwd_inv_we_sig : std_logic;

signal xk_index_sig, xn_index_sig : std_logic_vector(12 downto 0);
signal xk_index : std_logic_vector(12 downto 0);

signal fifo1_din_sig, fifo1_dout_sig : std_logic_vector(15 downto 0);
signal fifo2_din_sig, fifo2_dout_sig : std_logic_vector(15 downto 0);

signal fifo1_wr_en_sig, fifo1_rd_en_sig : std_logic;
signal fifo2_wr_en_sig, fifo2_rd_en_sig : std_logic;
signal ar_sig, ai_sig : std_logic_vector(15 downto 0);
signal br_sig, bi_sig : std_logic_vector(15 downto 0);
signal pr_sig, pi_sig : std_logic_vector(32 downto 0);

signal pulse_cnt : std_logic := '0';
```

```
signal factor_rom_address : std_logic_vector(12 downto 0);

signal xn_index_ext, xk_index_ext : integer range 0 to fft_len*2-1;
signal xn_index_ext_msb, xk_index_ext_msb : std_logic := '0';

signal douta_sig : std_logic_vector(31 downto 0);
signal data_outR : std_logic_vector(15 downto 0);

--component declarations

component fifo_generator_v2_2
      port (
      clk: IN std_logic;
      din: IN std_logic_VECTOR(15 downto 0);
      rd_en: IN std_logic;
      rst: IN std_logic;
      wr_en: IN std_logic;
      data_count: OUT std_logic_VECTOR(12 downto 0);
      dout: OUT std_logic_VECTOR(15 downto 0);
      empty: OUT std_logic;
      full: OUT std_logic);
end component;

component my_fft_core is
  port (
    fwd_inv_we : in STD_LOGIC;
    rfd : out STD_LOGIC;
    start : in STD_LOGIC;
    fwd_inv : in STD_LOGIC;
    dv : out STD_LOGIC;
    done : out STD_LOGIC;
    clk : in STD_LOGIC;
    busy : out STD_LOGIC;
    edone : out STD_LOGIC;
    xn_re : in STD_LOGIC_VECTOR ( fft_in_word_size-1 downto 0 );
    xk_im : out STD_LOGIC_VECTOR ( fft_out_word_size-1 downto 0 );
    xn_index : out STD_LOGIC_VECTOR ( log2_fft_len-1 downto 0 );
    xk_re : out STD_LOGIC_VECTOR ( fft_out_word_size-1 downto 0 );
    xn_im : in STD_LOGIC_VECTOR ( fft_in_word_size-1 downto 0 );
    xk_index : out STD_LOGIC_VECTOR ( log2_fft_len-1 downto 0 )
  );
end component;

component cmult IS
      port (
      ar: IN std_logic_VECTOR(word_size-1 downto 0);
      ai: IN std_logic_VECTOR(word_size-1 downto 0);
      br: IN std_logic_VECTOR(word_size-1 downto 0);
      bi: IN std_logic_VECTOR(word_size-1 downto 0);
      pr: OUT std_logic_VECTOR(product_word_size-1 downto 0);
      pi: OUT std_logic_VECTOR(product_word_size-1 downto 0);
      clk: IN std_logic);
END component;
```

```
component freq_domain_factor_ram
      port (
      addra: IN std_logic_VECTOR(log2_fft_len-1 downto 0);
      addrb: IN std_logic_VECTOR(log2_fft_len-1 downto 0);
      clka: IN std_logic;
      clkb: IN std_logic;
      dinb: IN std_logic_VECTOR(31 downto 0);
      douta: OUT std_logic_VECTOR(31 downto 0);
      web: IN std_logic);
end component;

begin

fft_core1: my_fft_core -- instantiate fft core
      port map(
      xn_re => fft_r_din_sig,
      xn_im => fft_i_din_sig,
      start => '1',
      fwd_inv => fwd_inv_sig,
      fwd_inv_we => fwd_inv_we_sig,
      clk => clk,
      xk_re => xk_re,
      xk_im => xk_im,
      xn_index => xn_index_sig,
      xk_index => xk_index,
      rfd => open,
      busy => open,
      dv => open,
      edone => open,
      done => open);

-- instantiate FIFO data buffers
fifo1 : fifo_generator_v2_2
            port map (
                  clk => clk,
                  din => fifo1_din_sig,
                  rd_en => fifo1_rd_en_sig,
                  rst => '0',
                  wr_en => fifo1_wr_en_sig,
                  data_count => open,
                  dout => fifo1_dout_sig,
                  empty => open,
                  full => open);

fifo2 : fifo_generator_v2_2
            port map (
                  clk => clk,
                  din => fifo2_din_sig,
                  rd_en => fifo2_rd_en_sig,
                  rst => '0',
                  wr_en => fifo2_wr_en_sig,
                  data_count => open,
                  dout => fifo2_dout_sig,
                  empty => open,
                  full => open);
```

```
cmult1: cmult --instantiate complex multiplier
      port map(
      ar => ar_sig,
      ai => ai_sig,
      br => br_sig,
      bi => bi_sig,
      pr => pr_sig,
      pi => pi_sig,
      clk => clk);

--instantiate filter RAM
freq_domain_factor_ram0 : freq_domain_factor_ram
            port map (
                  addra => factor_rom_address(log2_fft_len-1 downto 0),
                  addrb => filt_ram_addr,
                  clka => clk,
                  clkb => slow_clk,
                  dinb => filt_ram_data,
                  douta => douta_sig,
                  web => filt_ram_we);

--connect filter RAM to complex multiplier inputs
br_sig <= douta_sig(31 downto 16);
bi_sig <= douta_sig(15 downto 0);

--instantiate fifos between fft core and complex multiplier
fft2cmult_real_fifo: fifo_generator_v2_2
      port map(
      clk => clk,
      rst => '0',
      din => fft2cmult_real_fifo_din_sig,
      wr_en => fft2cmult_fifo_wr_en_sigR,
      rd_en => fft2cmult_fifo_rd_en_sigR,
      dout => ar_sig,
      full => open,
      empty => open,
      data_count => open);

--adjust scaling into multiplier
fft2cmult_real_fifo_din_sig <= fft_r_dout_sig(22 downto 7);
fft2cmult_imag_fifo_din_sig <= fft_i_dout_sig(22 downto 7);

fft2cmult_imag_fifo: fifo_generator_v2_2
      port map(
      clk => clk,
      rst => '0',
      din => fft2cmult_imag_fifo_din_sig,
      wr_en => fft2cmult_fifo_wr_en_sigR,
      rd_en => fft2cmult_fifo_rd_en_sigR,
      dout => ai_sig,
      full => open,
      empty => open,
      data_count => open);
```

```
--connect signals that have a constant connection
xk_index_out <= xk_index_sig;

fifo1_din_sig <= data_in;
fifo2_din_sig <= fft_i_dout_sig(24 downto 9); --adjust scaling here

--fifo1 read enable signal controls forward/inverse fft control signal
fwd_inv_sig <= not fifo1_rd_en_sig;

xn_index_ext <= conv_integer(xn_index_ext_msb & xn_index_sig);
xk_index_ext <= conv_integer(xk_index_ext_msb & xk_index_sig);

controller: process(clk)
begin
if rising_edge(clk) then
      factor_rom_address <= xn_index_sig + 6;
      fft2cmult_fifo_wr_en_sigR <= fft2cmult_fifo_wr_en_sig;
      fft2cmult_fifo_rd_en_sigR <= fft2cmult_fifo_rd_en_sig;
      data_out <= data_outR; --register output

      --synchronize fft output signals to the clock
      fft_r_dout_sig <= xk_re;
      fft_i_dout_sig <= xk_im;
      xk_index_sig <= xk_index;

      if xn_index_sig = conv_std_logic_vector(8191,log2_fft_len) then
            xn_index_ext_msb <= not xn_index_ext_msb;
      end if;

      if xk_index_sig = conv_std_logic_vector(8191, log2_fft_len) then
            xk_index_ext_msb <= not xk_index_ext_msb;
      end if;

      if xn_index_sig = conv_std_logic_vector(4096,log2_fft_len) then
            fwd_inv_we_sig <= '1';
      else
            fwd_inv_we_sig <= '0';
      end if;

      --control fifo1 read enable signal
      if xn_index_ext = fft_len then
             fifo1_rd_en_sig <= '1';
      elsif xn_index_ext = 0 then
            fifo1_rd_en_sig <= '0';
      end if;

      --control fifo1 write enable signal
      if xn_index_ext = fft_len + 1 then
             fifo1_wr_en_sig <= '0';
      elsif xn_index_ext = 1 then
            fifo1_wr_en_sig <= '1';
      end if;

      --control fifo2 read enable signal
      if xk_index_ext = fft_len - 2 then
```

```
            fifo2_rd_en_sig <= '1';
      elsif xk_index_ext = fft_len*2 - 1 then
            fifo2_rd_en_sig <= '0';
      end if;
      --control fifo2 write enable signal
      if xk_index_ext = fft_len*2-1 then
            fifo2_wr_en_sig <= '1';
      elsif xk_index_ext = fft_len - 0 then
            fifo2_wr_en_sig <= '0';
      end if;

      --route data into fft core
      if xn_index_ext > fft_len + 1 or xn_index_ext < 2 then
                  --route data from input to fft core sgn ext if needed
                  fft_r_din_sig <= fifo1_dout_sig(15 downto 0);
                  fft_i_din_sig <= data_in(15 downto 0);
      else
                  --adjust scaling here
                  fft_r_din_sig <= pr_sig(30 downto 15);
                  fft_i_din_sig <= pi_sig(30 downto 15);
      end if;

      --control writing to fft2cmult_fifo
      if xk_index_ext = fft_len - 2 then
            fft2cmult_fifo_wr_en_sig <= '1';
      elsif xk_index_ext = fft_len*2 - 2 then
            fft2cmult_fifo_wr_en_sig <= '0';
      end if;

      --control reading from fft2cmult_fifo
      if xn_index_ext = fft_len*2 - 5 then
            fft2cmult_fifo_rd_en_sig <= '1';
      elsif xn_index_ext = fft_len - 5 then
            fft2cmult_fifo_rd_en_sig <= '0';
      end if;

      --route data to output
      if xk_index_ext > 8191 and xk_index_ext <= fft_len*2-1 then
                  data_outR <= fifo2_dout_sig;
      else
                  --adjust scaling here
                  data_outR <= fft_r_dout_sig(24 downto 9);
      end if;

      --send out ready_for_input pulse 3 cycles before input is needed
      if xn_index_ext = fft_len*2-2 or xn_index_ext = fft_len - 2 then
            pulse_cnt <= '1';
            ready_for_input <= pulse_cnt;
      else
            ready_for_input <= '0';
      end if;

end if;
end process;
```

```
end Behavioral;
```

Overlap-Adder

```
-----------------------------------------------------------------------
-- overlapadd.vhd
--
-- The overlap-adder recieves data segments from the circular
-- convolver, overlaps them appropriately, and adds the overlapping
-- portions to produces the final linear convolution result.
--
-----------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity OverlapAdd is
    Port ( fast_clk : in std_logic;
           slow_clk : in std_logic;
           data_in : in std_logic_vector(15 downto 0);
           data_out : out std_logic_vector(15 downto 0);
             xk_index_sig : in std_logic_vector(12 downto 0);
             ADC_data1 : out std_logic_vector(15 downto 0));
            --      data_send : in std_logic);
end OverlapAdd;

architecture Behavioral of OverlapAdd is
--signal definitions
--signals suffixed with Rx indicate registered signals
signal f1_wr_en_sig, f2_wr_en_sig : std_logic := '0';
signal f2_wr_en_sigR, f1_rd_en_sig : std_logic := '0';
signal f2_rd_en_sig, f1_rd_en_sig_delay : std_logic := '0';
signal f1_rd_en_sig_delay_delay : std_logic := '0';
signal f3_wr_en_sig : std_logic := '0';
signal f1_data_out_sig : std_logic_vector(15 downto 0);
signal f1_data_out_sigR : std_logic_vector(15 downto 0);
signal f1_data_out_sigR2 : std_logic_vector(15 downto 0);
signal f1_data_out_sigR3 : std_logic_vector(15 downto 0);
signal f2_data_out_sig : std_logic_vector(15 downto 0);
signal f2_data_out_sigR : std_logic_vector(15 downto 0);
signal f2_data_out_sigR2 : std_logic_vector(15 downto 0);
signal f2_data_in_sig, f3_data_in_sig : std_logic_vector(15 downto 0);

signal xk_index_sigR, xk_index_sigR2 : std_logic_vector(12 downto 0);
signal slow_pulse_cnt : std_logic_vector(3 downto 0) := "0000";
signal fast_pulse_cnt : std_logic_vector(3 downto 0) := "0000";
```

```
signal data_inR, data_inR2 : std_logic_vector(15 downto 0);
signal data_inR3, data_inR4 : std_logic_vector(15 downto 0);

signal f2_data_cnt_sig : std_logic_vector(10 downto 0);
signal f2_data_cnt_sigR : std_logic_vector(10 downto 0);

signal datain_cnt : integer range 0 to 100000 := 0;
signal datain_cnt2 : integer range 0 to 100000 := 0;
signal dataout_cnt : integer range 0 to 100000 := 11;
--constant definitions
constant L : integer := 3584;--segment size
constant fft_len : integer := 8192;--length of FFT
constant log2_fft_len : integer := 13;
constant M : integer := fft_len-L+1; -- filter length

signal data_send_slow_clock_sync : std_logic;
signal data_send_slow_clock_sync2 : std_logic;
signal enable_fifo1_read, enable_write : std_logic := '0';

--component definitions

component a_fifo_2k
      port (
      din: IN std_logic_VECTOR(15 downto 0);
      rd_clk: IN std_logic;
      rd_en: IN std_logic;
      rst: IN std_logic;
      wr_clk: IN std_logic;
      wr_en: IN std_logic;
      dout: OUT std_logic_VECTOR(15 downto 0);
      empty: OUT std_logic;
      full: OUT std_logic);
end component;

component s_fifo_2k
      port (
      clk: IN std_logic;
      din: IN std_logic_VECTOR(15 downto 0);
      rd_en: IN std_logic;
      rst: IN std_logic;
      wr_en: IN std_logic;
      data_count: OUT std_logic_VECTOR(10 downto 0);
      dout: OUT std_logic_VECTOR(15 downto 0);
      empty: OUT std_logic;
      full: OUT std_logic);
end component;

component s_fifo_4k
      port (
      clk: IN std_logic;
      din: IN std_logic_VECTOR(15 downto 0);
      rd_en: IN std_logic;
      rst: IN std_logic;
      wr_en: IN std_logic;
      data_count: OUT std_logic_VECTOR(11 downto 0);
```

```
      dout: OUT std_logic_VECTOR(15 downto 0);
      empty: OUT std_logic;
      full: OUT std_logic);
end component;

begin

fifo1 : s_fifo_4k
            port map (
                  clk => fast_clk,
                  din => data_inR2,
                  rd_en => f1_rd_en_sig,
                  rst => '0',
                  wr_en => f1_wr_en_sig,
                  data_count => open,
                  dout => f1_data_out_sig,
                  empty => open,
                  full => open);

fifo2 : s_fifo_2k
            port map (
                  clk => fast_clk,
                  din => data_inR2,
                  rd_en => f2_rd_en_sig,
                  rst => '0',
                  wr_en => f2_wr_en_sig,
                  data_count => f2_data_cnt_sig,
                  dout => f2_data_out_sig,
                  empty => open,
                  full => open);

fifo_3 : a_fifo_2k
            port map (
                  din => f3_data_in_sig,
                  rd_clk => slow_clk,
                  rd_en => '1',
                  rst => '0',
                  wr_clk => fast_clk,
                  wr_en => f3_wr_en_sig,
                  dout => data_out,
                  empty => open,
                  full => open);

fast_clock_ctrl: process(fast_clk) --controls data flow
begin
      if rising_edge(fast_clk) then
      f1_data_out_sigR <= f1_data_out_sig; --register fifo1 output
      f1_data_out_sigR2 <= f1_data_out_sigR;
      f1_data_out_sigR3 <= f1_data_out_sigR2;
      f2_data_out_sigR <= f2_data_out_sig; --register fifo2 output
      f2_data_out_sigR2 <= f2_data_out_sigR;

      f2_data_cnt_sigR <= f2_data_cnt_sig; --register data count signal

      data_inR <= data_in;
```

```
      data_inR3 <= data_inR;
      data_inR2 <= data_inR3;
      data_inR4 <= data_inR2;

      f2_wr_en_sigR <= f2_wr_en_sig;
      xk_index_sigR <= xk_index_sig;

            --control writing to fifo1
            if xk_index_sigR = conv_std_logic_vector(L+3,…
… log2_fft_len) then
                  f1_wr_en_sig <= '1';
            elsif xk_index_sigR = conv_std_logic_vector(2*L+3,…
… log2_fft_len) then
                  f1_wr_en_sig <= '0';
            end if;

            --control writing to fifo2
            if xk_index_sigR = conv_std_logic_vector(2*L + 3,…
… log2_fft_len) then
                  f2_wr_en_sig <= '1';
            elsif xk_index_sigR = conv_std_logic_vector(3, …
… log2_fft_len) then
                  f2_wr_en_sig <= '0';
            end if;

            --control data flow into async output fifo (fifo3)
            if xk_index_sigR > conv_std_logic_vector(2, log2_fft_len)…
… and xk_index_sigR < conv_std_logic_vector(M-L+2, log2_fft_len) then
                  f3_data_in_sig <= data_inR3 + f1_data_out_sigR2 …
… + f2_data_out_sigR2;
            else
                  f3_data_in_sig <= data_inR3 + f1_data_out_sigR2;
            end if;

            --control reading from fifo1
            if xk_index_sigR = conv_std_logic_vector(fft_len-1, …
… log2_fft_len) then
                  f1_rd_en_sig <= '1';
            elsif xk_index_sigR = conv_std_logic_vector(L-1, …
… log2_fft_len) then
                  f1_rd_en_sig <= '0';
            end if;

            --control reading from fifo2
            if xk_index_sigR = conv_std_logic_vector(fft_len-1, …
… log2_fft_len) and f2_data_cnt_sigR > conv_std_logic_vector(1200,…
… 12) then
                  f2_rd_en_sig <= '1';
            elsif xk_index_sigR = conv_std_logic_vector(M-L-2, …
… log2_fft_len) then
                  f2_rd_en_sig <= '0';
            end if;

            --control writing to async fifo3
            if xk_index_sigR = conv_std_logic_vector(3,…
```

```
… log2_fft_len) then
                  f3_wr_en_sig <= '1';
            elsif xk_index_sigR = conv_std_logic_vector(L+3,…
… log2_fft_len) then
                  f3_wr_en_sig <= '0';
            end if;

      end if;
end process fast_clock_ctrl;
end Behavioral;
```

APPENDIX C

CHIRP GENERATOR VHDL

Chirp Generator

```
-----------------------------------------------------------------------
-- chirp_gen.vhd
--
-- The chirp generator creates the parallel data word, which when
-- passed to a Rocket I/O serializer, generates a single bit linear
-- frequency chirped signal.  Chirp rate is determined by the
-- chirp_rate input signal, phase init, and frequency init values.
-- Chirp time is determined by the input chirp_dur
--
-----------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity chirp_gen is
Port (clk : in std_logic;
           chirp_start : in std_logic;
           chirp_done : out std_logic;
           chirp_out : out std_logic_vector(39 downto 0);

           chirp_rate : in std_logic_vector(23 downto 0);
           chirp_dur : in std_logic_vector(23 downto 0);

           phase_init_shift_in : in std_logic_vector(23 downto 0);
           phase_init_shift_en : in std_logic;
           freq_init_shift_in : in std_logic_vector(23 downto 0);
           freq_init_shift_en : in std_logic);
end chirp_gen;

architecture Behavioral of chirp_gen is

-- define array subtypes for phase_accum and freq_accum vectors
SUBTYPE sr_width IS STD_LOGIC_VECTOR(23 DOWNTO 0);
TYPE sr_length IS ARRAY (39 DOWNTO 0) OF sr_width;

SIGNAL phase_accum, freq_accum : sr_length;

-- define states for state machine
type states is (init, send_chirp);
signal chirp_state : states := init;

signal cycle : std_logic_vector(31 downto 0);
signal enabler : std_logic_vector(3 downto 0) := "0000";

begin

chirp_gen: process(chirp_start,clk)
begin
```

```
if rising_edge(clk) then

case chirp_state is
      when init =>
            -- load in new phase init values
            if (phase_init_shift_en = '1') then
                  phase_accum(39 downto 1) <= phase_accum(38 downto 0);
                  phase_accum(0) <= phase_init_shift_in;
            end if;

            -- load in new frequency init values
            if (freq_init_shift_en = '1') then
                  freq_accum(39 downto 1) <= freq_accum(38 downto 0);
                  freq_accum(0) <= freq_init_shift_in;
            end if;

            --set output to zeros
            chirp_out <= (others => '0');

            --clear cycle counter
            cycle <= (others => '0');

            chirp_done <= '0'; --chirp not done

            --send chirp if chirp_start recieved
            if chirp_start = '1' then
                  chirp_state <= send_chirp;
            else
                  chirp_state <= init;
            end if;

      when send_chirp =>
            --increment phase accumulators
            phase_accum_update: for i in 0 to 39 loop
                  phase_accum(i) <= phase_accum(i) + freq_accum(i);
            end loop;

            --increment frequency accumulators
            freq_accum_update: for i in 0 to 39 loop
                  freq_accum(i) <= freq_accum(i) + chirp_rate;
            end loop;

            --set output bits
            output_update: for i in 0 to 39 loop
                  chirp_out(i) <= phase_accum(i)(23);
            end loop;

            --stay in this state until chirp is finished
            if cycle < chirp_dur then
                  cycle <= cycle + '1';
                  chirp_done <= '0';
                  chirp_state <= send_chirp;
            else
                  cycle <= (others => '0');
                  chirp_done <= '1';
```

```
                chirp_state <= init;
            end if;

      when others =>
            chirp_state <= init;

end case;
end if;
end process;
end Behavioral;
```

APPENDIX D

MATLAB RECOVERY CODE

Matlab Recovery Simulation

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% fpga_recovery_sim.m recovers spectral features
% using a variety of methods including: original
% frequency domain method, using convolution, using
% overlap-add method, and using overlap-save method.
%
% variants of this script using overlap-add and
% fixed point math were used to generate the SNR
% surface plot.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

method = 0;
while method < 1 || method > 4
    disp(sprintf('1) Frequency Domain Recovery Algorithm'));
    disp(sprintf('2) Time Domain Convolution'));
    disp(sprintf('3) Overlap and Add Method'));
    disp(sprintf('4) Overlap and Save Method'));
    method = input('Enter method: ');
end

%subtract background from readout signal and convert sample rate from 2
%GSPS to 100 MSPS
data1 = decimate(h0p1k4_Intensity(:,1) - bgk4_Intensity(:,1), 20)';

N = length(data1);

Fs = 100;%sample rate (MSPS)
T = N/Fs;%time of entire vector (us)
K=1;% chirp rate (MHz/us)

if method > 1
    fl = input('Enter Filter Length: ');
    df =K/Fs;
    min_taps = Fs/df;
    freq = linspace(-Fs/2,Fs/2,min_taps);
    factor = exp(j*pi*sign(freq).*freq.^2/K);  %frequency domain factor
    tdf = real(fftshift(ifft(fftshift(factor)))); % time domain factor
    right_cut_off = floor(1.1*min_taps/2);
    tdf = tdf(right_cut_off + 1 - fl:right_cut_off);
else
    freq = linspace(-Fs/2,Fs/2,N);
    factor = exp(j*pi*sign(freq).*freq.^2/K);   %frequency domain
factor
end

if method > 2
    seg_size = input('Enter input vector segment size: ');
end
```

```
switch method
    case 1, %perform original recovery algorithm
        out = real(ifft(fftshift(  fftshift(fft(data1)) .* factor  )));

    case 2, %perform recovery with time domain convolution
       out = myconv(tdf, data1);
       out = real(out(fl-(right_cut_off - min_taps/2)+1:...
           N+fl-1-(right_cut_off - min_taps/2)+1));

    case 3, %perform recovery with overlap and add method
        padded_data1 = [data1, zeros(1,...
            seg_size*ceil(length(data1)/seg_size) - length(data1))];
        out = zeros(1,length(padded_data1)+length(tdf)-1);
        for k = 0:length(padded_data1)/seg_size-1
            segment = padded_data1(k*seg_size+1:(k+1)*seg_size);
            conv_seg = myconv(segment, tdf);
%             conv_seg = myconv_fixpt(segment, tdf, word_size);
            out(k*seg_size+1:(k+1)*seg_size+length(tdf)-1) = ...
                out(k*seg_size+1:(k+1)*seg_size+length(tdf)-1) +
conv_seg;
        end
        out = real(out(fl-(right_cut_off - min_taps/2)+1:...
            N+fl-1-(right_cut_off - min_taps/2)+1));

    case 4, %perform recovery with overlap and save method
        step_size = seg_size - (fl-1);
        padded_data1 = [zeros(1,seg_size), data1, zeros(1,...
            step_size*ceil((length(data1)-
seg_size)/step_size)+seg_size)];
        out = zeros(1,length(padded_data1));
        for k = 0:(length(padded_data1)-seg_size)/step_size
            segment = padded_data1(k*step_size+1:k*step_size+seg_size);
            cir_conv_seg = real(ifft(fft(segment) .* fft([tdf,...
            zeros(1,seg_size - fl)])));
            out(k*step_size+1:(k+1)*step_size) = out(k*step_size+1:...
                (k+1)*step_size) +
cir_conv_seg(fl:length(cir_conv_seg));
        end
        out = real(out(seg_size-(right_cut_off - min_taps/2)+2:...
        N + seg_size-(right_cut_off - min_taps/2)+1));
end

plot(out)
```

Convolution with the DFT

```
function out = myconv(a, b)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%myconv_fixpt convolves a with b in the frequency
% domain
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%pad input data to size of output
padded_a = [a(:); zeros(length(b)-1, 1)];
padded_b = [b(:); zeros(length(a)-1, 1)];

%find DFT of padded segments
fft_padded_a = fft(padded_a);
fft_padded_b = fft(padded_b);

%multiply transformed segments
product = fft_padded_a .* fft_padded_b;

%convolution result is the real part of the IDFT of the product
out = real(ifft(product)).';
```

Fixed-Point Convolution with the DFT

```
function out = myconv_fixpt(a, b, word_size)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%myconv_fixpt convolves a with b in the frequency
% domain, and quantizes after each step to word_size
% bits
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%zero pad to length of convolution and quantize to word_size bits
padded_a = double(fi([a(:); zeros(length(b)-1, 1)], 1, word_size));
padded_b = double(fi([b(:); zeros(length(a)-1, 1)], 1, word_size));

%find DFT and quantize to word_size bits
fft_padded_a = double(fi(fft(padded_a), 1, word_size));
fft_padded_b = double(fi(fft(padded_b), 1, word_size));

%multiply and quantize to word_size bits
product = double(fi(fft_padded_a .* fft_padded_b, 1, word_size));

%IDFT product and requantize to word_size bits
out = double(fi(real(ifft(product)).', 1, word_size));
```

REFERENCES CITED

[1] Chang, T., Mohan, R. Krishna, Tian, M., Harris, T. L., Babbitt, W. R., and Merkel, K. D., "Frequency-chirped readout of spatial-spectral absorption features". Physical Review A, Volume 70, 063803 (2004)

[2] Chang, T., Tian, M., Babbitt, W. R., "Numerical modeling of optical coherent transient processes with complex configurations—I. Angled bean geometry". Journal of Luminescence 107 (2004) 129-137

[3] Chang, T., Tian, M., Mohan, R. Krishna, Renner, C., Merkel, K. D., and Babbitt, W. R., "Recovery of spectral features readout with frequency-chirped laser fields" Opt. Lett. Vol. 30, No. 10, May 15, 2005

[4] CRYOMECH 113 Falso Drive, Syracuse, New York 13211. "http://www.cryomech.com/developments.cfm?id=12"

[5] Jones, D. L., *Frequency Sampling Design Method for FIR Filters version 1.2.* The Connexions Project, Jun 9, 2005 "http://cnx.org/content/m12789/latest/"

[6] Lang M., and Laakso, T. I., "Simple and Robust Method for the Design of Allpass Filters Using Least-Squares Phase Error Criterion". IEEE Trans. on Circuits and Systems, vol. 41, pp. 40-47, Jan. 1994

[7] Manassewitsch, V., Frequency Synthesizers Theory and Design. A Wiley-Interscience Publication John Wiley & Sons, 1987

[8] Merkel, K. D., Mohan, R. Krishna, Cole, Z., Chang, T., Olson, A., Babbitt, W. R., "Multi-Gigahertz radar range processing of baseband and RF carrier modulated signals in Tm:YAG". Journal of Luminescence 107 (2004) 62-74

[9] Mohan, R. Krishna, Cole, Z., Reibel, R., Chang, T., Merkel, K. D., Babbitt, W. R., Colice, M., Schlottau, F., and Wagner, K.H., "Microwave Spectral Analysis Using Optical Spectral Holeburning". Proceedings of Microwave Photonics 2004, Microwave Photonics 2004, Ogunquit, Maine, October 4-5 2004.

[10] Oppenheim, A. V., and R. W. Schafer. Discrete-Time Signal Processing 2nd ed. Prentice Hall Signal Processing Series, Upper Saddle River, New Jersey, 1999

[11] Sellin, P. B., Strickland, N. M., Böttger , T., Carlsten, J. L., and Cone, R. L., "Laser stabilization at 1536 nm using regenerative spectral hole burning". Physical Review B, Volume 63, 155111 (2001)

[12] Sellin, P. B., Strickland, N. M., Carlsten, J. L., and Cone, R. L., "Programmable frequency reference for subkilohertz laser stabilization by use of persistent spectral hole burning". Opt. Lett.Vol. 24, No. 15, Aug 1, 1999

[13] Xilinx, DS260 *Fast Fourier Transform v3.2.* August 31, 2005, "http://www.xilinx.com/"

[14] Xilinx, XAPP656 *Using the Virtex-II Pro RocketIO MGT for Frequency Multiplication.* "http://www.xilinx.com/"