An FPGA Based Digital Radio for Meteor Radar Applications

by

L. R. Rochester

B.S., University of Colorado, 2004

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Masters of Engineering
Department of Electrical Engineering
2007
This thesis entitled:
An FPGA Based Digital Radio for Meteor Radar Applications
written by L. R. Rochester
has been approved for the Department of Electrical Engineering

__________________________________________
Prof. Scott E. Palo (Advisor)

__________________________________________
Prof. James Avery

__________________________________________
Prof. Dennis Akos

Date ________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.
High speed analog to digital conversion and dedicated digital signal processors offer the potential to revolutionize the radio science community. The increase in sampling speed and computing performance has drastically improved the bandwidth of processing that can be accomplished digitally allowing a push of the analog-to-digital conversion process further up the RF/IF chain from baseband. As such, the advent of software radios and digital receivers has moved much of the RF/IF chain from analog processing to digital processing. Interest has been growing in the radio science community to develop new, more capable and flexible digital receivers, to replace aging analog technology and provide new instruments with capabilities never before considered. Evidence of this is the current receiver development work occurring in conjunction with the new AMISR (Advanced Modular Incoherent Scatter Radar) [33] system and other upper atmosphere facilities such as Arecibo [34]and Jicamarca [35].

The implementation goal of this thesis is to develop a simple, agile, and inexpensive multi-channel digital receiver for meteor radar applications that could also be extended to other applications suitable for deployment on unmanned aerial vehicles. This digital receiver design exploits the low complexity and power, small weight and size of analog receivers, and also offers simplicity and low cost over current commercially available digital receivers. It also exploits recent advances in analog-to-digital conversion to greatly reduce analog intermediate frequency processing.

This digital receiver uses a multichannel analog-to-digital converter from that encodes in the 20-50Msps range, a Field Programmable Gate Array (FPGA), and a High Speed USB Transceiver to digitize multiple analog signals. The FPGA is used to
perform all conditioning and signal processing of the digital receiver, as well as to provide a memory interface to a USB Transceiver. The USB Transceiver allows a high-speed and low overhead data path to a host computer running the Linux operating system. Through the Linux operating system data can be saved to a mass storage device for post processing.

The reprogrammable nature of the FPGA provides tremendous flexibility for receiver configurations and requirements. The FPGA also provides a FIFO memory structure to ensure valid data, and glue logic for a USB interface to a host computer running a UNIX based operating system. Current USB specifications limit the combined output rate of all channels to 480Mbps and we have benchmarked the interface at 40MB/s using the Cypress FX2 USB interface and a host computer running the Linux operating system.
Acknowledgements

First and foremost I would like to thank the National Science Foundation for funding the research for this thesis. Funding for this thesis came from an NSF Grant with Award Number 00449985 under Scott Palo. I would like to acknowledge Professor Scott Palo for giving valuable ideas, support, and advice. Stephan Esterhuizen for his previous work on the streaming interface for USB. Phil Erickson and Frank Lind of MIT Haystack observatory for their advise on implementation of a digital receiver. Cody Vaudrin for giving invaluable lessons on how to use the Xilinx tools for debugging and verifying logic. I could not have done this project without the open source community projects GNU Radio and LibUSB. As well as all the small hardware companies that make prototyping boards, namely Kai Klein from Braintechnology, Charles Sweeney from Orange Tree Technologies, and Martin Schoeberl from JOP Design. I am very thankful for getting the opportunity to be funded and advised for this thesis.
## Contents

### Chapter

<table>
<thead>
<tr>
<th>Contents</th>
<th>vi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tables</td>
<td>vii</td>
</tr>
<tr>
<td>Figures</td>
<td>ix</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 Thesis Focus</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Design of a Digital Receiver</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2 Architecture of the Digital Receiver</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Bandpass Sampling the Operating Frequency</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Signal Processing Stage</td>
<td>4</td>
</tr>
<tr>
<td>2.3 USB Transceiver</td>
<td>12</td>
</tr>
<tr>
<td>2.4 USB Data Flow Model</td>
<td>13</td>
</tr>
<tr>
<td>2.5 USB Slave Mode</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Design Implementation and Theory</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 LVDS (Low Voltage Differential Signaling)</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Decimation</td>
<td>26</td>
</tr>
<tr>
<td>3.3 Sampling</td>
<td>27</td>
</tr>
</tbody>
</table>
3.4 Decimation .................................................. 28
3.5 Decimation Filters ......................................... 29

4 Evolution of the Digital Receiver Design 31
4.1 AD6654 Wideband IF to Baseband Receiver .............. 32
4.2 AD6654 ASIC Digital Receiver Board ....................... 39
4.3 ZestSC1 FPGA USB Board ................................ 45

5 Anomaly in the Streaming USB Interface 50
5.1 FPGA Logic .................................................. 52
5.2 FX2 Firmware ............................................... 53
5.3 Host Software Configuration ................................ 55
5.4 Hardware .................................................... 58
5.5 Results ....................................................... 59
5.6 Debugging Methods ........................................ 66

6 Summary and Future Work 69

Bibliography 73
Tables

| Table | Relevant timing delays for the Xilinx Spartan 3, Altera Cyclone II, and the FX2 USB Transceiver | 59 |
Figures

Figure

2.1 Digital Receiver Block Diagram ................................. 3
2.2 AD9259 default timing diagram showing serial data output .... 6
2.3 Processing of 4 channels (A,B,C,D) within the FPGA ......... 8
2.4 FIFO Memory interface from the FPGA to the USB Transceiver .... 11
2.5 All 12 possible configurations for FX2 endpoints from the USB Technical Reference Manual [22] ......................... 17
2.6 Control Signals for Slave Mode between the FX2 and the External Master from the USB Technical Reference Manual [22] ................. 18
3.1 LVDS signaling scheme voltages ............................... 23
3.2 LVDS Transmitter and Receiver Hardware ...................... 24
3.3 Microstrip transmission lines for LVDS PCB Design ............ 25
3.4 Dimensions for a microstrip sandwiched between a dielectric and a ground plane ................................................. 25
3.5 Characteristic Impedance for changing microstrip widths ....... 26
3.6 Frequency Spreading as a Result of Decimation ....................... 29
3.7 Block Diagram of a Decimating Filter .............................. 29

4.1 Functional Block Diagram of the AD6654 Wideband Receiver from the
AD6654 Data Sheet [24] ........................................... 33
4.2 Cascade Integrating Comb Filter Block Diagram ..................... 34
4.3 Frequency Response of an example CIC Filter consisting of 5 stages and
a decimation M=8. ............................................... 36
4.4 The 9 proposed Goodman-Carey Filters for efficient Hardware Implemen-
tation .............................................................. 37
4.5 Left: The Goodman Carey filters from 1 to 4. Right: Goodman Carey
Filters from 5 to 9. ............................................... 38
4.6 Block Diagram of the AD6654 ASIC Digital Receiver Board ....... 39
4.7 Simplified Logic Interface ........................................ 41
4.8 Digital Pictures of the Altera Cyclone Interface Board ............. 41
4.9 Theoretical and Experimental Impulse Response for the AD6654 ASIC . 42
4.10 I and Q samples of a processed radar pulse by the AD6654 receiver ... 44
4.11 Block Diagram a digital receiver architecture with a ZestSC1 FPGA USB
Board ................................................................. 46
4.12 ZestSC1 Prototyping FPGA USB Board Block Diagram ............ 47
4.13 Digital Picture of the ZestSC1 Prototyping Board, AD9259 Evaluation
board, and interfacing board ....................................... 48

5.1 Status of Logic when Anomaly Occurs ............................... 61
5.2 Top: Ten 512 byte packets received with bit-errors and discontinuities
from the Xilinx Board. Bottom: Ten 512 byte packets received correctly
from the Altera Board ............................................. 62
5.3 Top: Sawtooth Waveform sent at 40MB/s with the Xilinx FPGA using GNU Radio USB Library. Bottom: Sawtooth waveform sent at 40MB/s with the Xilinx FPGA using LibUSB.
Chapter 1

Thesis Focus

The focus of this thesis is to develop a versatile digital receiver with the primary but not only focus as a replacement to the analog receivers for the COBRA meteor radar system. The goal of this receiver is to have many independent channels for multiple antennas and timing stages. As well as to move higher up the RF-IF chain, and replace much of the analog filtering stages in the current COBRA meteor radar system. This receiver will make use of the high A/D sampling rates and digital filtering methods, as well as the simplicity of utilizing the USB bus for data transfer.

1.1 Design of a Digital Receiver

This subsection outlines the constraints and specifications of the COBRA meteor radar system. The COBRA system operates in the VHF (30-300 MHz) band typically between in the 30 to 46 MHz range, thus, the digital receiver must be able to properly translate radio echoes of different frequencies down to baseband. The primary science goal of the COBRA system is determine the direction and velocity of upper atmospheric winds at altitudes of 80-100km. Details on meter radar systems can be found in the following references [33], [34], [35]. To find the direction of the upper atmospheric winds interferometric methods are employed with the radial velocity and range of the meteor echo. Currently, the interferometric method uses five cross dipoles for reception, and four directional Uda-Yagi arrays used for transmitting. This interferometric method
requires the receiver to have multiple channels for the cross dipoles, the a clock for the A/D converter, the transmitter channel, as well as other potential signals of interest. For the COBRA meteor radar system we are constrained to a minimum of 6 channels. The bandwidth per channel of the receiver is driven by range resolution. The baseband sampling frequency is matched to the transmitted pulse width, and this pulse width is used to maximize the signal to noise ratio. Typically the pulse width for the COBRA system range is on the order of 10\(\mu\)sec requiring a matched filter bandwidth of 100kHz. By Nyquist we must sample at a rate of 200kHz. If we take the sampling rate to be 500kHz we can provide the output rate of the system. Each channel has an in-phase and quadrature component, and and uses two bytes. Using six channels the receiver will have an output rate of 12MB/s. This 12MB/s sampling rate is well under our USB maximum throughput of 40MB/s. However, we leave the requirement to have variable bandwidth per channel built into the design. The radial velocity of the upper atmospheric winds are no more than 200m/s. Since we are using a 10m wavelength, Doppler shifts on the order of 40Hz will be detected.

Future potential projects for the COBRA system are to upgrade the existing mono-static system to a bi-static system. Bi-static radar systems must have a common clock to properly estimate range and Doppler, making a potential need for another receiver channel to store timing information. Another constraint of the digital receiver is to be able to save data for post processing on a host computer. We have chosen the Linux operating system for a host computer to save, manage, and interface with this digital receiver.

A constraint for the bistatic implementation of the digital receiver is a stable clock source. This clock source must be coherent to the transmitter clock, thus a devoted channel to this clock must be used for the digital receiver. Likely solutions are GPS based clocks, however this problem will not be considered herein.
Chapter 2

Architecture of the Digital Receiver

Many off-the-shelf digital receivers [29] [30] utilize a dedicated ASIC for most signal processing, as well as an FPGA further down the processing chain for added functionality. These receivers interface through the PCI bus. The PCI bus is a robust architecture that has the capability to transfer at rates of 533 MB/s (PCI version 3.0) [28]. However, one large downside to using the PCI bus is the complicated kernel driver software associated with the PCI bus.

The architecture proposed by herein is to use only an A/D converter, an FPGA and a USB transceiver for digital reception, as shown in Figure 2.1. This architecture can be used with multiple channels streaming to the FPGA, and current FPGAs offer enough logic to perform a significant amount of signal processing on each channel. As outlined below the USB bus is a simple and robust means to transfer the received data to a host computer for real-time or post processing.

![Digital Receiver Block Diagram](image)

Figure 2.1: Digital Receiver Block Diagram
2.1 Bandpass Sampling the Operating Frequency

The sampling frequency of the COBRA radar system is much higher than the bandwidth of the transmitted pulse. Because the bandwidth of this transmitted pulse is small comparable to the sampling frequency a bandpass sampling technique also called undersampling can be utilized [36]. Existing COBRA radar systems operate in the range of 30-50 MHz, while the sampling rate of the digital receiver ranges from 20-50 MHz. The lower and upper bound for the A/D converter constrains the bandpass sampling rate. Because the sampling rate is less than the Radar operating frequency bandpass sampling techniques must be used in order to properly translate the received signal to baseband. As an example for the 30MHz COBRA radar system an example bandpass sampling regime could sample at 30MHz, and alias the signal down to baseband. For example the 46MHz system could be sampled at 23MHz, and the signal would be translated to baseband. As for sampling frequencies where the radar operating frequency is not an integer multiple of the radar operating frequencies, architectures with a mixer must be employed for proper translation down to baseband. When determining the sample rate for a general bandpass system the bandwidth of interest must be considered. As well as where integer multiples of the sampling rate fall within the center frequency of that signal.

2.2 Signal Processing Stage

The FPGA plays the most active role in the digital receiver design. It performs three main tasks: deserialization of the A/D serial bit streams, signal processing of the digital channels, and providing an interface to the USB Transceiver. Each task of the FPGA will be discussed in the following subsections.
2.2.0.1 Deserialization of LVDS signals

Typical low cost FPGA’s provide capability to receive and transmit LVDS signals. These signals can be transferred at a rate nearly 500Mbps. The LVDS inputs only require a special buffer for the positive and negative inputs from an FPGA standpoint. The LVDS buffer simply converts the LVDS signal to native internal logic within the FPGA. The AD9259 A/D converter sends its encoded waveform as a 14-bit code in serial form. In order to deserialize this bit stream a shift register is used to capture each bit, then every 14 shifts the digital sample can be used for the 14-bit encoded sample. The Spartan 3 from Xilinx for example, is capable of receiving LVDS signals at 666Mb/s [25], which is high enough to support the 140Mb/s input produced by running the A/D converter at 20MHz. LVDS signaling is a differential scheme where logic levels are represented by the difference between the signal pair. Typically the pair has a + and - sign to differentiate the individual signals in the LVDS pair. This can be seen in Figure 2.2. More discussion of this signaling scheme can be found in the LVDS section of this thesis.

There is one caveat to the serial bit stream the AD9259 outputs after encoding the analog signal, that is bits are valid on every rising and falling edge of the data clock this is commonly known as known as double data rate (DDR). A timing diagram can be seen in Figure 2.2. DDR signals require data to be captured on the rising and falling edge of the clock. Typically, low cost FPGA’s consist only of rising edge Flip-Flops, which presents a problem for sampling data that occurs on the falling edge of the DDR clock. This presents a problem from an architectural standpoint in the FPGA.

To capture bits on the falling edge of the clock, the clock signal can be either split into multiple phases, where the original DDR clock is used for the rising edge valid data, and the DDR clock phase shifted 180 degrees out of phase can be used for capturing data on the respectively falling edges of the DDR clock. The second method to capture
DDR data is to multiply the DDR clock frequency by a factor of 2; then data can be latched every rising edge of this new doubled clock frequency. In order to split the clock into multiple phases at low data rates an inverter can be used, however, this can present problems when the clock duty cycle changes or data rates become comparable to the delay through an inverter.

The best method of doubling the clock rate, or providing multiple phases for a clock is to use dedicated circuitry to lock onto the phase of the incoming clock. Both Xilinx and Altera use a delay lock loop (DLL) and phased locked loop (PLL) respectively to manipulate the frequency and phase of a clock signal. The method employed to deserialize bits from the AD9259 is to increase the DDR clock frequency by a factor of 2.

Shown in Figure 2.2 is the timing diagram that the FPGA must deserialize, to capture encoded waveforms from the AD9259. This timing diagram consists of what is called the data clock oscillator (DCO), the frame clock oscillator (FCO), and the data D. All these signals are in LVDS differential pairs from the AD9259 IC. The FCO signal is meant to align the starting point of a 14-bit word. However, this signal is not
synchronized to the rising edge of the DCO, and thus should not be used as a latch for the FPGA to capture the 14-bit sample. A deserializer consisting of a 1-bit shift register can easily be implemented to put this data stream into 14-bit words, but careful consideration must be considered when using the frame clock. See the Xilinx Application XAPP245 Eight channel, One Clock, One Frame LVDS Transmitter/Receiver [15], or the source code I wrote for this thesis for more information. A good design method is to use the FCO frame clock to start a counter which will keep track of where in the 14-bit data stream data is being deserialized.

The binary format of the serial data produced from the encoded signal outputs is offset binary. This binary system has the MSB as the sign bit and uses the smallest number for zero, meaning all numbers are relative to the smallest value. Binary offset meaning all values are offset from the smallest value. The format of this number system is easiest to illustrate with an example. We will compare this with the typical two’s complement numbering system.

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Binary Offset</th>
<th>Two’s Complement</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>111</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>011</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>000</td>
</tr>
<tr>
<td>-1</td>
<td>010</td>
<td>111</td>
</tr>
<tr>
<td>-2</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>-3</td>
<td>000</td>
<td>101</td>
</tr>
<tr>
<td>-4</td>
<td>-</td>
<td>100</td>
</tr>
</tbody>
</table>

Converting from a binary offset representation to a two’s compliment representation...
The main purpose of the FPGA in our design is to take advantage of the parallel processing capabilities of the FPGA for multiple channels. The FPGA must process four complex channels simultaneously and then hand data to a FIFO for buffering. A high level block diagram of the DSP portion in the FPGA can be seen in Figure 2.3. This figure first depicts channels A-D in the FPGA. These filters are then mixed by a complex exponential $e^{-j2\pi f_{mix}}$, where $f_{mix}$ is the mixer frequency. The digitized channel gets multiplied by the real and imaginary part of this mixer (in-phase and quadrature components). The channels in this mixer each have an I and Q component, this is
indicated by the double arrows in each channel in Figure 2.3. This effectively doubles the number of filters after mixing.

A complex mixer must be used to find sign of the Doppler shift. The most efficient complex mixer design is to use a LUT (Look Up Table) that multiplies the current sample with two values from the sinusoid in the LUT. This complex mixer requires $2N$ multiplies per sample, where $N$ is the number of channels. In order to implement this mixer a tone is generated in Matlab, and quantized to the desired number of bits for the FPGA. The pointer for the I and Q channels are simply set $1/4$ of the length of the LUT apart, for the proper phase relationship. The number of dedicated hardware multipliers in FPGAs are steadily increasing thus, hardware multiplies can be used for mixing multiple channels. The Spartan 3 XC3S1000 we are using has 24 18x18 bit hardware multipliers on board.

After the complex mixer stage the FPGA will use three multi-rate filters shown in Figure 2.3 that reduce the sample rate, to adjust the sampling rate closer to the true Nyquist rate of the digitized signal. By reducing the sample rate we are effectively reducing the number of signals we can reconstruct. Because we have sampled at a rate much higher than that meteor echo our bandwidth is so wide unwanted noise is let into the system. To reduce our bandwidth closer to the matched filter bandwidth we utilize decimation. Because the sample rate is so high comparable to the matched filter bandwidth, we must cascade decimation filters to reduce our sample rate closer to the matched filter bandwidth. This sample rate is reduced by using a decimation filter, then disregarding every other sample.

Decimating filters are used so that when samples are disregarded the resulting signal does not have unwanted frequency components aliased into the spectrum from decimation. More information is found herein on in the theory section of this thesis. The three Goodman-Carey filters are used as a minimum to reduce the sampling rate without aliasing in unwanted frequency components. A single Goodman-Carey Filter disregards
every other sample, thus cascading three of these filters will reduce the sampling rate by 8. The filtering performed by 3 decimating filters does not produce the flattest passband, nor does the stopband have the lowest attenuation that could be provided by our FPGA architecture, however, at a first cut, 3 stages are a good start.

The Goodman-Carey filters are implemented as a delay line equal to the order of the filter. For example the filter taps for a Goodman-Carey filter of order 7 (F3 in Figure 4.4 are \{-1 0 9 16 9 0 -1\}. When a sample is passed through this delay line it will be multiplied by -1 for the first delay, then 0 for the second delay, then 9, and so on. The output of this filter will be a weighted average of the samples as they pass through the delay line. Each delayed sample is multiplied by its corresponding filter tap, and a sum of all the taps produces the filter output. A toggle signal asserts on every other rising edge of the clock for a decimation of two, this is the signal that disregards every other sample. At the assertion of this toggle signal line the output data for this filter is provided. Although, every other sample is disregarded, its information is not lost, because each output from the filter contains information from the weighted sum. There is a vast array of information on FIR filters, as well as their implementation on an FPGA. For more information see [4], [18], and [2].

### 2.2.0.3 Interface to the USB Transceiver

The FPGA must provide a gateway to the USB Transceiver, this aspect of the digital receiver is the most challenging from a timing, simulation, and implementation perspective. In this digital receiver design the FPGA and USB Transceiver have master and slave roles. The FPGA takes the master role by keeping the USB transceiver busy with data transfers. Because the encoding rate from the A/D converter is different from the 48MHz clock signal for the FX2 logic, we have two different clock domains. This makes for a challenging implementation. An asynchronous FIFO is generally used as a way to interface between different clock domains. The FPGA has an internal FIFO that
buffers the channel data, this data is then moved from the internal FPGA FIFO to the USB FIFO where data is sent to the host computer via the USB bus. By choosing the FPGA as master, data will be transferred when the internal FPGA FIFO is not empty and the USB Transceivers FIFO is not full.

The FPGA must buffer I and Q samples from each channel on the receiver. For a 4 channel receiver 8 FIFO's are needed, double the amount because each channel must have an I and Q sample. The simplification of 128 bits in Figure 2.4 is a representation of a sum of 8 16 bit input FIFO buffers. The Xilinx intellectual property core for an asynchronous FIFO can have a high aspect ratio (a ratio of input bits to that of output bits); however, the aspect ratio of 16 is currently unsupported. The FIFO buffers on the FPGA have a read clock from the FX2 running at 48MHz, and a write clock from the decimated encode signal from the A/D converter. On the rising edge of the write clock 8 samples are always written. However, on the read side of the clock more logic must be placed.

There are many ways to interface the FIFO buffers on the FPGA and the FX2, namely because to empty flag, full flag, and programmable flag yield a number of different ways to fill the FIFO. Also, a programmable full flag can be configured to assert on
any buffer level, and also considers the number of committed packets in the endpoint. All of these flags are software programmable, and can be output on various pin locations. This programmable flag is set such that it is asserted when EP6 has been filled with 3 packets, and 256 bytes, or 3 and \( \frac{1}{2} \) packets. The \( \frac{1}{2} \) packet is used to ensure the buffer always remains full with 3 committed packets and overflows or underflows are not possible.

As one can imagine there are many different logic combinations to interface two cascading FIFOs. For this design mostly every combination has been tried. An anomaly is was found in the ZestSC1 board, and one workaround method was to fill the FIFO buffer on the FX2 completely full when the empty flag asserts. The minimal logic required to interface the FIFO on the FPGA and the FX2 is shown in Figure 2.4. The logic in Figure 2.4 is self describing and will not be discussed further.

### 2.3 USB Transceiver

Historical motivation for the Universal Serial Bus originally came from three interrelated considerations [7]. The first need was for a connection from the PC to a telephone. Unfortunately, this consideration never lifted off. The second consideration was Ease-of-Use. It is well known many of the PC’s I/O interfaces lack the flexibility and ease of use offered by USB. The third consideration that went into design of USB was port expansion. Port expansion is usually not used in most digital receiver and software radio designs due to their heavy uses of the bus. In April 27, 2000 the USB 2.0 specification [7] was created in order to keep up with increasing needs for a higher transfer rate required by industry, research, and consumers. The USB 2.0 specification adds a third device speed of 480Mbps. USB devices come in three types, low speed, full speed, and high speed, with their respective transfer rates of 1.5Mb/s, 12Mb/s and 480Mb/s for high speed devices. This corresponds to a rate of 187.5kB/s, 1.5MB/s, and 60MB/s, This transfer rate does not include the error correction, and overhead.
The high speed USB device is great for digital receivers and software radio applications because of its high transfer rate and ease of use. USB applications can be written in user space, which evades the necessity to develop low level kernel drivers that become complex and hard to maintain for differing kernel versions.

Cypress Semiconductor has the largest market share for USB 2.0 Transceivers with its EZ-USB FX2 transceiver. This transceiver is powered by an enhanced 8051 microcontroller legacy Intel architecture. This transceiver supports both full and high speed transfer modes. When a USB device is plugged into a port, it is enumerated, meaning it reveals its default configuration for data transfers, along with its endpoint configurations and identification information. Cypress has patented what they call ReNumeration. ReNumeration is performed after a device has enumerated the first time, then new configuration data that can be loaded onto the FX2 device for a custom configuration. ReNumeration is basically a way to download new firmware onto the USB device once it has been attached to the USB Hub. The new configuration data from ReNumeration is stored in 8051 op-code executable format and is sent through the configuration endpoint EP0 such that the 8051 will execute the firmware [22]. The 8051 executes the new firmware to change the FX2's default configuration. In addition to USB transactions the FX2 does other tasks such as responding to host requests through the configuration endpoint, loading and or storing data from EEPROM, port I/O, and can be configured to reply to specific configuration requests called vendor requests through the configuration endpoint.

2.4 USB Data Flow Model

The Universal Serial Bus has various transfer types in its data flow model that are implemented to best suit the particular USB device. The USB specification outlines that data transfers are always in relation to the host. Therefore, a data transfer from the host to the device is an OUT transfer, and a transfer from the device to the host is
an IN transfer. In the USB 2.0 specification [7] data is transferred through what is called a pipe. The USB 2.0 specification outlines that endpoints exist at the end of a pipe. The FX2 has 4 endpoints that can be used for high speed transfers, they are named 2, 4, 6, and 8. Each of these endpoints can be configured as an IN, or OUT endpoint, as well as other special features that will be discussed later. For a high speed application the following constrains must be considered for the type of transfer: packet size, bus access time, latency, and error handling. USB devices implement four transfer types, which are named, control transfers, isochronous transfers, bulk transfers, and interrupt transfers.

Control transfers are used by every USB device and can only be accessed through endpoint 0, also named the control endpoint. The USB host uses special SETUP tokens to configure the USB device as well as to gain information through EP0. Some examples of SETUP tokens to configure a USB device through endpoint 0 include setting certain values in memory, getting USB descriptors which describe the device’s vendor ID, product ID, power usage, and endpoint configuration, changing interfaces which configure the endpoints in various ways, and synchronizing USB frames, just to name a few. In order to learn about all USB SETUP tokens see the EZ-USB Technical Reference Manual. [22]. The Technical Reference Manual Chapter 2 outlines Endpoint 0 which is the most complex endpoint. Endpoint 0 can also have vendor requests which are specific tasks that can be written for the FX2 to execute, these vendor requests can be used to transfer setup information, call firmware routines, and transfer debugging information. Also, endpoint zero is used to transfer the firmware to the FX2.

Isochronous transfers are used when data must be sent in a time-critical manner, such as audio or video. Isochronous packets can be 1024 bytes, although, the packets have no handshaking protocol. Thus, packets have no retry mechanism, cannot stall when then host or USB device is not ready, and are limited 16-bit CRC error correction. Despite the downsides to the Isochronous transfer, they usually are unimplemented or
have limited support for a host side USB API [8] [9].

Bulk transfers are used for bursty data, and packets at high speed can be up to 512 bytes in size. Full handshaking is used for bulk transfers as well as 16-bit CRC error correction (better than Isochronous because of the smaller packet size). Bulk transfers are the most popular implementation choice, and usually have the largest amount of support for a USB API. Bulk transfers are the transfer type of choice for the digital receiver.

Interrupt transfers can have packets sizes up to 1024 bytes at high speed, and are polled by the host such that a packet will not be missed. The host issues requests for the packet and the device acknowledges when a packet is ready for a transfer. This transfer type is supported by most USB APIs, and can be a good alternative to bulk transfers, because of the larger packet size allowed.

To learn more about any aspect of USB see the USB 2.0 Specification [7]. This specification gives a very clear and concise description of all aspects of the USB specification.

## 2.5 USB Slave Mode

Generally speaking a USB Transceiver must have a means to receive or source information from the USB Bus. The FX2 can be configured as a slave FIFO or with a general purpose interface GPIF.

The FX2 can be configured to be in slave mode or in GPIF mode. In GPIF mode the FX2 device acts as a master to its external periphery. Decision logic can be programmed into the FX2 such that it contains the logic to interface with external hardware to send or receive information. This decision logic can be quite complex and can be used for example to read or write to FIFO external to the FX2. The GPIF mode has many limitations however, namely being able to control 8 FIFOs in the FPGA based digital receiver. Using the GPIF mode requires translating a state diagram to read
from external periphery into firmware registers. The complexity of controlling external periphery is not worth the complexity of using an FPGA to control the FX2 in slave mode. Stephan Esterhuizen’s USB interface [?] uses GPIF mode, however, this mode only works when the clock source is externally sourced, which cannot be performed due to prototyping board limitations. Translating a desired state diagram into firmware is done most efficiently by using provided Cypress Semiconductor GPIF Designer software [32]. Due to the complexity of the GPIF mode and the simplicity of the slave mode, the slave mode was chosen as the design choice for the FX2 device.

Slave mode requires some further introduction to the FX2’s endpoints. The endpoints 2 and 6 on the FX2 can have configurations with multiple buffering levels. These levels can be configured to have single, triple, or quad buffering. The advantage to multiple buffering levels is that one buffer can be sent over the USB bus while another buffer is being filled. When multiple buffering is used the FIFO can be thought of as multiple FIFO buffers within the same memory space. Cypress recommends a quad buffered 512 byte endpoint for maximum throughput with the AUTOIN feature enabled. This requires the endpoint to occupy 2048 bytes in memory; however the endpoint is divided into four memory sections for 512 byte packets. When the FIFO reaches a level of 512, 1024, 1536, or 2048 bytes then a respective 1, 2, 3, or 4 packets are in the buffer, respectively. The AUTOIN feature allows the FX2 to automatically commit packets to the USB domain when the FIFO level is at the packet boundaries. The meaning of having packets committed is the packet is no longer available to be filled or examined, and is queued to be sent over the USB bus. It is necessary for full USB throughput to use the FX2 to automatically commit packets; this is because if the 8051 is used to commit the packets it is running at a much slower rate than that of the USB bus. The USB bus runs at 480 MHz, while the 8051 is running 10 times slower, at 48MHz. The AUTOIN feature means custom logic automatically allows the packet to be sent over the USB bus when enough bytes are received to form a packet. The AUTOIN feature leaves the
8051 completely out of all transferring transactions through the high speed endpoint. If the 8051 were involved in these transactions, the data rate would be crippled since it is running at such a low speed compared to the USB bus. All possible configurations of the FX2 endpoints are shown in Figure 2.5. The digital receiver uses Endpoint 6 quad buffered with 512 byte packets, which is shown in configurations 2, 5, and 8 (numbers in the lower row) in Figure 2.5. To summarize, the buffer the FX2 uses for receiving is EP6, which is a quad buffered 512 byte packet FIFO.

Previous work was done by Stephan Esterhuizen [27] to use the GNU radio code [9] to provide a streaming interface from an A/D converter to the FX2. The GNU radio code, written in C and compiled for an 8051 microcontroller, used the GPIF mode in firmware discussed above, and this code needed to be converted to slave mode in order to handshake with a FIFO on the FPGA. Stephan’s software was first converted so that the FX2 used slave mode then was later converted to assembly for simplicity reasons.

Figure 2.6 outlines the control signals that are present between the FX2 and an external master, the FPGA in this case. A description of these flags will be outlined in the following list. Some of the pins are not described because they pertain to reading
Figure 2.6: Control Signals for Slave Mode between the FX2 and the External Master from the USB Technical Reference Manual [22]
an endpoint from the FX2 since information is only flowing from the FX2 to the host computer only writing to endpoints is performed in the digital receiver design.

- **IFCLK**: This is the clock all logic is synchronized to. All signal transitions will occur on the rising edge of this clock. This clock is source from the FX2 internal clock running 48 MHz.

- **FLAG[A,B,C,D]**: These flags are used to show the status of the selected endpoint of the FX2. The behavior of these flags is software programmable and can be used to indicate if the selected endpoint is empty, full, and programmable full. The programmable full status is software programmable as well. The external master will use the status of these flags to fill, or stop filling the FX2 selected endpoint.

- **SLWR**: The slave write signal, when this signal is asserted on the rising edge of IFCLK data will be latched from the FD[15:0] data bus.

- **PKTEND**: This signal can be used to commit a packet short of its length. This signal is not used in the digital receiver design.

- **FD[15:0]**: This is the data bus for the endpoint. This data bus can be configured to be 8 or 16 bits wide.

- **FIFOADR[1:0]**: These signals select endpoint 2,4,6, or 8 for the destination. The FPGA for this design holds these lines constant to select endpoint 6.

The state machine and the timing diagram shown in Figure 2.7 is the task of the FPGA to implement. This state machine must be implemented in some form to fill the FX2 endpoint buffer. When the endpoint is not full indicated by FLAG B, the FPGA should not fill the endpoint. The FPGA logic must be synchronous to the IFCLK, and must control the SLWR, FIFOADR[1:0], and FD[15:0] control lines. Figure 2.7 does
IDLE: When write event occurs, transition to State 1.

STATE 1: Point to IN FIFO, assert FIFOADR[1:0] (setup time must be met with respect to the rising edge of IFCLK), transition to State 2.

STATE 2: If FIFO-Full flag is false (FIFO not full), transition to State 3 else remain in State 2.

STATE 3: Drive data on the bus, assert SLWR (setup and hold times must be met with respect to the rising edge of IFCLK), de-assert SLWR. Transition to State 4.

STATE 4: If more data to write, transition to State 2 else transition to IDLE.

Figure 2.7: Top: A state machine to perform slave FIFO writes. Bottom: A timing diagram depicting writes to the FX2 in slave mode EZ-USB Technical Reference Manual [22]
depict the FULL, EMPTY, SLWR, and PKTEND signals as complimentary logic, also known as not logic. This logic has programmable polarity and can be configured to normal polarity in the FX2 firmware.
Chapter 3

Design Implementation and Theory

This section of the thesis provides background for some of the implementation aspects of the digital receiver such as LVDS signaling, and a theoretical background that leads to an explanation of decimation filters.

3.1 LVDS (Low Voltage Differential Signaling)

When data is transferred electronically high data rates are typically achieved through parallelism. Data paths are widened to their corresponding word sizes 8-bit, 16-bit, 32-bit, and in some cases even as high as 128-bits to send information at a higher rate. Sending information in parallel increases PCB board complexity by adding bus traces, as well as IC complexity by adding more signal pins. An obvious alternative to sending data in parallel is serial transmission. To compete with parallel transmission data rates, robust serial transmission schemes with high data rates must be devised. A motivation to reduce PCB complexity, and IC cost has given a need for serial transmission. The LVDS standard answers this need and is outlined in the Scalable Coherent Interconnect (SCI document, specified in the IEEE 1596.3 standard, as well as the ANSI/TIA/EIA-644-A standard). LVDS is a serial scheme where binary data is transmitted along a differential pair. A typical LVDS driver sources current through the differential pair in the 25-40mA range. A differential load resistor usually 100Ω is placed in between differential signal traces, yielding a voltage swing of 250-400mV.
LVDS can be thought of as a current source where the direction of current is dependent on polarity.

The AD9259 A/D converter has 4 channels for the encoded waveforms and two timing signals, all of which are output from LVDS pairs. Each LVDS signal requires two traces, and having only 2 traces per channel significantly reduces the number of traces on a PCB board, and also minimizes the number of I/O pins needed for the FPGA. As one can imagine a digital receiver with 8 parallel channels would be cumbersome to route on a 4-Layer PCB board because there would be nearly $8 \times 14 = 112$ traces (assuming a 14-bit A/D converter).

The common mode voltage specified by LVDS is 1.25V. With a voltage swing of 250-400mV the LVDS maximum differential voltage will be 0.85-1.65V. This range makes easy use of low voltages typically used in CMOS circuit designs. Figure 3.1 shows a graphical description of a LVDS signal.

From Figure 3.2 we can see that the LVDS driver is a current source that will allow current to flow in either direction across the 100 Ω termination resistor that is placed next to the LVDS receiver. The LVDS receiver is high impedance, thus most of the driver current flows across the 100 Ω termination resistor. When implementing a
receiver in an FPGA for LVDS we must place this termination resistor as close to the FPGA as possible.

For the ZestSC1 digital receiver design we are mating the AD9254 outputs to user I/O pins the ZestSC1 board provides. These IO pins are on a standard 0.1” pitch header which route through the PCB board to pads on the FPGA. A PCB must be designed to route the AD9259 outputs to pins on the ZestSC1 board. This PCB board must be designed to have 50Ω microstrip lines that mate the outputs from the AD9259 to the I/O pins on the ZestSC1 board as well as the 100Ω termination resistors between differential pairs. Because the user I/O pins on the ZestSC1 board have nearly 1cm of length before connecting to the FPGA pads, we will not be able to route the termination resistors nearly right against the FPGA I/O pads as recommended. Without redesigning the PCB board, this as close as we can come to the FPGA package.

We will now show the method of determining the trace thickness of the PCB board for 50Ω microstrip lines that connect the A/D converter to the FPGA. This analysis is based from David Pozar’s analysis of an approximate electrostatic solution on pg. 146 [23]. In order to ensure the correct width of the microstrip lines the PCB trace is modeled as a microstrip sandwiched between a dielectric and a ground plane. The characteristic impedance $C$ of this microstrip can be found by first computing the capacitance per unit length of microstrip without a dielectric, letting $\epsilon_r = 1$ and then
finding the capacitance per unit length $C_0$ with the dielectric. The effective dielectric constant $\epsilon_r$ can then be represented as.

$$
\epsilon_e = \frac{C}{C_0}
$$

(3.1)

The characteristic impedance $Z_0$ is related to $\epsilon_r$, $C$, $C_0$ and the speed of light in a vacuum $c$ by the following equation.

$$
Z_0 = \frac{\sqrt{\epsilon_e}}{\epsilon C}
$$

(3.2)

In order to compute the capacitance per unit length $C_0$ without the dielectric, and the capacitance per unit length $C_0$ with the dielectric we must compute equation 3.3 for the dimensions shown in Figure 3.4.

$$
C = \frac{1}{\sum_{n=1,n\text{odd}}^{\infty} \frac{4a \sin(n\pi \frac{W}{2a}) \sinh(n\pi \frac{d}{2a})}{(n\pi)^2 W_0 \sinh(n\pi \frac{d}{2a}) + \epsilon_r \cosh(n\pi \frac{d}{2a})}}
$$

(3.3)

Figure 3.4: Dimensions for a microstrip sandwiched between a dielectric and a ground plane
For a typical 4 layer PCB board made from FR-4 (Flame resistant 4, a typical dielectric for PCB boards) the thickness between the top layer and an inner layer for ground is known to be \( d = 0.3048 \text{mm} \), the relative permittivity \( \epsilon_r = 4.6 \), and for the 5-inch PCB board we used for our design the width parameter nearly \( 2a = 12\text{cm} \). We computed the characteristic impedance for our PCB dimensions for various widths \( W \) of the microstrip.

We can see from Figure 3.5 that the width of the microstrip should be \( W = 0.75\text{mm} \) according to this model. However, this width is quite wide when considering the connector the microstrip lines are mating with. Narrowing \( W \) to 0.5mm changes the impedance by only 3Ω. This was the choice used on the PCB board and has been verified by an oscilloscope that the digitized LVDS signals are resolute.

\[ \textbf{3.2 Decimation} \]

Typical digital receiver systems sample the incoming analog signal much higher than the Nyquist frequency. When signals are over sampled, i.e. the sampling frequency is much greater than the bandwidth of the signal, and then decimation is performed in order to adjust the bandwidth of the sample signal closer to the Nyquist frequency.
3.3 Sampling

It is often convenient to represent a sampled signal by the product of a continuous signal, and an infinite train of Dirac delta functions, commonly written as the Dirac Comb function. We can define the Dirac Comb function as:

\[ \Delta(t) = T \sum_{k \in \mathbb{Z}} \delta(t - kT) \] (3.4)

The Dirac Comb function is periodic with period \( T \), thus, it can be represented by a Fourier Series. The Fourier Series coefficients \( c_k \), are easily found to be:

\[ c_k = \frac{1}{T} \int_{-\frac{T}{2}}^{\frac{T}{2}} \delta(t) e^{-j2\pi \frac{k}{T} t} dt = T e^{-j0} = 1 \] (3.5)

The above relation holds by the duality of the Fourier Transform. The Dirac comb \( \Delta(t) \) in its Fourier Series Representation can now be shown to be

\[ \Delta(t) = \sum_{k \in \mathbb{Z}} c_k e^{-j2\pi \frac{k}{T} t} = \sum_{k \in \mathbb{Z}} e^{-j2\pi \frac{k}{T} t} \] (3.6)

A sampled signal \( x_s(t) \) can now be represented by taking the product of the Dirac Comb and the continuous signal \( x_s(t) \).

\[ x_s(t) = x(t)\Delta(t) = \sum_{k \in \mathbb{Z}} x(t) e^{-j2\pi \frac{k}{T} t} \] (3.7)

The Fourier Transform of \( x_s(t) \) can now be found

\[ X_s(f) = \int_{-\infty}^{\infty} x_s(t) e^{-j2\pi f t} dt = \int_{-\infty}^{\infty} \sum_{k \in \mathbb{Z}} \delta(t - kT) e^{-j2\pi f t} dt = \sum_{k \in \mathbb{Z}} x(kT) e^{-j2\pi f kT} \] (3.8)

The result of this equation is known as the Discrete Time Fourier Transform (DTFT). If we think of \( x(kT) \) as “samples” of the continuous time signal \( x(t) \), then the DTFT represents a signal that is discrete in time, and continuous in frequency. It is
more convenient to let $n = kT$ then we can then represent the DTFT more compactly as

$$X(e^{j2\pi f}) = \sum_{n \in \mathbb{Z}} x[n]e^{-j2\pi fn} \quad (3.9)$$

It can be easily verified that equation 3.9 is has a period of $\frac{k}{T}$. The periodic nature of the DTFT plays an important role in decimation.

### 3.4 Decimation

After understanding that the DTFT has a period of $\frac{k}{T}$ in the frequency domain, we can develop an understanding of decimation.

To decimate a signal by an integer $M$ is to keep every $M$th sample, and disregard the other samples. For a discrete time signal $x[n]$ where $n \in \mathbb{Z}$ a new signal $y[n] = x[Mn]$ can be defined as $x[n]$ decimated by $M$. Lets take a look at how the frequency spectrum of the decimated signal $y[n]$ looks in relation to $x[n]$.

By the DTFT the frequency spectrum $Y(e^{j2\pi \frac{f}{T}})$ of $y[n]$ is defined to be

$$Y(e^{j2\pi \frac{f}{T}}) = \sum_{n \in \mathbb{Z}} y[nT]e^{-j2\pi fnT} \quad (3.10)$$

However, $y[n]$ is sampled at intervals of $MT$ for $x[n]$ making

$$Y(e^{j2\pi \frac{f}{MT}}) = \sum_{n \in \mathbb{Z}} x[nMT]e^{-j2\pi fnMT} = X(e^{j2\pi \frac{f}{MT}}) \quad (3.11)$$

From this we can see that $Y(e^{j2\pi \frac{f}{MT}})$ is the same signal as $X(e^{j2\pi \frac{f}{MT}})$, only it now has a period of $\frac{k}{MT}$. We can note that for $M > 1$ the aliases of $Y(e^{j2\pi \frac{f}{MT}})$ repeat more often than that of $Y(e^{j2\pi f})$. This can be best illustrated by Figure 3.6.

In Figure 3.6 the signal $x[n]$ is decimated by 3. The spectrum of $x[n]$ is widened by a factor of 3. We can see from Figure 3.6 the spectral widening due to decimation. With decimation we are primarily concerned with the widening of the spectrum of interest.
This motivates a need for filtering prior to decimation, so signals do no alias into the spectrum of interest.

### 3.5 Decimation Filters

Knowing the effect decimation plays on widening the frequency spectrum, we must have a filter prior to taking every $M^{th}$ sample so higher frequencies do not alias into the frequency spectrum of interest. A classic example of how decimation filters are implemented is shown in Figure 3.7.

A filter with a cut-off frequency at $\frac{f_s}{2M}$ Hz must be placed prior to the decimation block. The cutoff frequency will remove unwanted signals what could alias into our spectrum as a result of decimation. A decimation filter has an input/output sampling ratio of M. For example, a decimation filter with $M = 2$, will have an output rate of $2T_s$ Hz where $T_s$ is the input sampling rate.
\[ y[Mn] = (h \ast x)[Mn] = \sum_{k=0}^{N} h[k]x[Mn - k] \quad (3.12) \]

The output of the decimation filter \( y[Mn] \) can be shown by the convolutional sum of the input signal \( x[n] \) with that of the decimation filter \( h[k] \). In many implementations of this filter every \( n^{th} \) sample does not need to be computed since only every \( Mn^{th} \) sample is used. These implementations are called poly-phase filters and their implementations will not be discussed here. As a brief example we can consider any undersampling case where aliasing occurs. Say for example a system sampling at 800Hz is decimated to 400Hz, where a 350Hz tone is present in the spectrum. The decimation filter will have a cutoff frequency of 200Hz, which will wipe out the 350Hz tone. If a decimation filter were not used the 350Hz tone would alias into the spectrum as a 50Hz tone.
Chapter 4

Evolution of the Digital Receiver Design

The digital receiver design has evolved in ideology and hardware throughout its life-span. Original ideas were to use a dedicated ASIC (Application Specific Integrated Circuit) to perform all digitizing and signal processing and stream the received data to the USB bus. The most adaptable and supported ASIC for this task is manufactured by Analog Devices, who offer vast documentation and a working evaluation board along with software to analyze the received data through USB. However, as time progressed we found using an ASIC for our digital receiver was not the best design choice.

Typically, multi-channel ASIC digital receiver circuits have many address lines, data lines and control signals that can be very tedious and time consuming to implement on a printed circuit board. To adapt an ASIC into a digital receiver design a methodology to store configuration data onto the ASIC must be devised. This configuration data includes filter taps, mixing values, and a broad array of individual parameters inherent to the ASIC. In order to properly configure the ASIC before data can be received, a fairly robust microcontroller setup or FPGA circuit must be employed to store the configuration data and interface with the handshaking protocols to hand the configuration data onto the ASIC. Another pitfall to using an ASIC is adaptability. The ASIC circuitry is meant to be general, although, slight processing changes can be unrealizable with the given hardware.

With the complexity of using an ASIC, and the need for a dedicated microcon-
controller or FPGA to program the integrated circuit, it becomes logical to import all signal processing tasks onto an FPGA architecture, as well to interfacing to a streaming interface such as USB to store data for post processing. This makes for a simple design consisting of a A/D converter, an FPGA, and a USB transceiver. FPGAs offer the capability to implement all the functions ASICs provide for this digital receiver architecture.

4.1 AD6654 Wideband IF to Baseband Receiver

4.1.0.4 Theory of Operation

The first potential design for the digital receiver the Analog Devices AD6654 [24]. The AD6654 is a dedicated ASIC for digitizing and processing multiple channels. This chip incorporates all signal processing for 6 channels, can digitize at a rate of 92.16Msps, and has many configurable options. Some of the applications for the AD6654 include multi-carrier receivers, digital cellular telephony schemes (e.g. EDGE, GSM, CDMA2000, etc ...), micro and pico cell systems, software radio, smart antenna systems, wireless local loop, broadband data applications, as well as instrument and test equipment systems. This chip has the capability to digitize a single broadband signal, modulate the signal (code or tone), and perform various filtering operations. This chip is a down converting chip, thus, the output sample rate is less than or equal to the input rate.

In this paragraph we will outline all the blocks in the AD6654 ASIC shown in Figure 4.1. Referring to Figure 4.1 the signal goes through a single 14-Bit ADC Front End, then to a input matrix which simply chooses which channels are used, and the physical channel they are routed to. A copy of the 14-bit sample goes to each channel in the receiver chip for an I and Q channel, denoted by the double arrow for each channel. The 32-bit Numerically Controlled Oscillator (NCO) is then used as a complex mixer.
Figure 4.1: Functional Block Diagram of the AD6654 Wideband Receiver from the AD6654 Data Sheet [24]
to translate incoming signal to a desired frequency band. A cascade integrating comb (CIC) filter is used in each channel which filters the signal and decimates the sample rate by M. Following the CIC filter multiple stages of simple FIR filters and decimating half band filters (HB) are placed. A half band filter refers to a decimating low-pass FIR filter which has its 3dB point at a quarter of the Nyquist Rate, or half the frequencies up to the Nyquist rate are attenuated, and only every other sample is used from the output of this filter. Following the FIR/HB filter blocks is a data router which can be used if the desired channels are to be routed to different locations. Next comes the Mono-Rate RAM Coefficient Filter (MRCF) which is a filter with programmable taps, this filter is non-decimating. The MRCF is followed by a Decimating RAM Coefficient Filter (DRCF) which is nearly identical to the MRCF, except the filter has a programmable decimation from 1 to 16. Following the MRCF and DRCF is the Channel RAM Coefficient filter (CRCF) which is a decimating filter with programmable taps. The last stage of the AD9259 consists of an interpolating half band filter which doubles the frequency rate, opposite to that of a decimating filter. The magnitude and phase response as well as number of taps, bit width, and gain can be found in in the AD6654 Data Sheet [24]. Some of these stages will be elaborated for their interesting features and advantages for digital receiver architectures, which were employed in the FPGA design. In the following paragraphs we will discuss the CIC filter and Goodman-Carey filters, these filters are considered to be hardware efficient filters and are used in many digital receiver designs.

Following the NCO stage is a cascade integrating comb filter (CIC) filter. This
filter uses only delay blocks and has no multipliers, thus is efficient for hardware use. This filter is ideal for simplicity, speed, and is popular for implementations inside an FPGA. As seen in Figure 4.2 this filter can be cascaded for a more desirable impulse response. The difference equation can be written as:

\[ y[n] = x[n] - x[n - M] + y[n - 1] \] (4.1)

The Z-Transform can be taken of the difference equation in 4.1 and for N cascades of the filter we arrive at the following result.

\[ H(z) = \left( \frac{1 - z^{-M}}{1 - z^{-1}} \right)^N \] (4.2)

From the Z-Transform equation 4.2 we can write the frequency response of the CIC filter as follows.

\[ |H(f)| = \left( \frac{\sin(\pi M f)}{\sin(\pi f)} \right)^N \] (4.3)

As an example a 5 stage (N = 5) CIC filter with the decimation M=8 is shown below in Figure 4.3. It is important to notice that this filter has a very fast roll-off and hence not a very flat passband, however, this filter is utilized when the bandwidth of the signal of interest is very small in relation to the sampling frequency. When this is the case the passband will be nearly linear. This will make the quick roll-off of the CIC filter less of an impact. Also, the CIC filter will have \( \frac{M}{2} \) nulls up to the Nyquist rate when M is even, and \( \frac{M-1}{2} \) nulls when M is odd, this can be easily found by finding where the numerator in equation 4.3 is an integer multiple of \( \pi \).

Another common filter architecture which is ideal for FPGA implementations was proposed by Goodman-Carey in 1977 [6]. The Goodman-Carey Filters have simple integer taps, which are usually very close to powers of two, and are always scaled by a factor that is a power of two. The Goodman-Carey filters also have zeros for the odd
Figure 4.3: Frequency Response of an example CIC Filter consisting of 5 stages and a decimation M=8.
Figure 4.4: The 9 proposed Goodman-Carey Filters for efficient Hardware Implementation.

Table 4.1: Filter Coefficients

<table>
<thead>
<tr>
<th>Filter</th>
<th>Order</th>
<th>$h(0)$</th>
<th>$h(1)$</th>
<th>$h(2)$</th>
<th>$h(3)$</th>
<th>$h(5)$</th>
<th>$h(7)$</th>
<th>$h(9)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td>-9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F4</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F5</td>
<td>11</td>
<td>236</td>
<td>159</td>
<td>-25</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F6</td>
<td>11</td>
<td>346</td>
<td>208</td>
<td>-44</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F7</td>
<td>11</td>
<td>512</td>
<td>302</td>
<td>-53</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F8</td>
<td>11</td>
<td>802</td>
<td>490</td>
<td>-116</td>
<td>33</td>
<td>-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F9</td>
<td>19</td>
<td>8192</td>
<td>5042</td>
<td>-1277</td>
<td>429</td>
<td>-116</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

*F1 can be of any order, with all $h(i) = 1$. F2-F9 are half-band filters with $h(-i) = h(i)$ and $h(2) = h(4) = h(6) = 0$.*

Shown in Figure 4.5 is the frequency response for all the proposed Goodman-Carey filters. As expected as the order increases the more the filter approximates an ideal low pass filter. The Goodman-Carey filters are by no means the best filters for use in an FPGA, and in most cases more taps can be utilized for a better frequency response. It is not uncommon for many hardware architectures to have 512 or more taps. However, these filters are a good example and a benchmark for a simple and preliminary design for logic within an FPGA. It can be seen from Figure 4.5 that the gain of the filters is non-unity. The DC gain of a digital filter $h$ is shown as $|H(0)|$ below, where $N$ is the order of the $k$th Goodman Carey filter.

$$|H(0)| = \sum_{i=0}^{N-1} h_k[i]$$  \hspace{1cm} (4.4)

For ease of implementation the gain in equation 4.4 of the filter should be unity, requiring $|H(0)|$ to sum to a power of 2, so the final stage of the filter can perform a logical shift right to adjust the gain. A filter for minimal hardware constrains the filter taps to integers near powers of two, as well as the gain in equation 4.4 to be a power of
Figure 4.5: Left: The Goodman Carey filters from 1 to 4. Right: Goodman Carey Filters from 5 to 9.
two.

4.2 AD6654 ASIC Digital Receiver Board

To interface the AD6654 Wideband multi-channel digital receiver discussed previously with the Brainintechology USB board [20], an adapter board was needed. This adapter board provides glue logic via use of an Altera Cyclone FPGA, and was created to provide a means to buffer the multiple channels through a FIFO, then write the data to the FX2 USB transceiver. Because of hardware constraints the FIFO buffer in the FPGA becomes the least complex by having a clock for writing to the FX2 buffer, as well as a clock synchronized to the encoded data from the ASIC.

To implement an asynchronous FIFO in VHDL is quite difficult due to crossing clock domains. This is due with the fact that the write and read pointers of the FIFO are incremented by the respective read and write clock. When computing the difference between the write and read pointer to find if the FIFO is empty of full extra precautions must be met to make sure the read or write pointer is not changing. In order to take a difference between the write and read pointers that are changing asynchronously a Gray counter is typically used, as outlined in a application note by Xilinx [14]. A Gray counter is a counter that uses the Gray code. The hamming distance is only one between successive values in the Gray code. Thus, when a buffer full or empty check is computed, only 1 byte can be in error because of the Gray Code structure. Various
asynchronous FIFO implementations are provided by Altera as intellectual property. An Altera asynchronous FIFO for synchronizing the AD6654 to the FX2 was used and configured to have a 16-bit input and output data bus and depth of 512 words. This asynchronous FIFO was used to interface the AD6654 ASIC to that of the USB transceiver. This simple interface can be shown in the Figure 4.7 below.

The internal FIFO shown in Figure 4.7 is the basis for a FIFO buffer to interface asynchronous read and write clocks. This FIFO is filled when the FIFO is not full, indicative of the inverter between the full flag and the write enable flag. The read enable to this FIFO is asserted when the FPGA FIFO is not empty and the FIFO on the FX2 is not full, hence the nor gate. An extra D-flip-flop is used for synchronization.

Digital Pictures of the PCB board that interfaces the Altera Cyclone board from JOP Designs [21], the USB Transceiver from Braintechnology [20], and the AD6654 ASIC can be shown in Figure 4.8. This board is 4-layers, with the inner layers as power and ground. It has a simple power regulation circuitry as well as LEDs to display states of logic signals, such as the full and empty status of the asynchronous FIFO.

A test experiment was constructed with a 10MHz encode rate to the AD6654. This experiment enabled the CIC filter with a decimation of 2, FIR1, HB2, the DRCF with a decimation of 2, and the CRCF with a decimation of 2. Both the DRCF and CRCF FIR filters used all 128 taps, configured with a quantized equiripple low pass filter, constructed using Matlab. The output rate will be the input rate divided by $2^4$ or 652kHz for this experiment. The theoretical impulse response was analyzed in Matlab, and compared to the experimental impulse response, shown in Figure 4.9. Points on this plot are generated by encoding one channel of the AD9259 with a desired frequency and measuring the output amplitude with a spectrum analyzer. The source of error between the theoretical and experimental results occurs from frequency bias and fluctuation of the signal generator, as well as amplitude errors from the signal generator. The theoretical results consider all quantization errors from the filters.
Figure 4.7: Simplified Logic Interface

Figure 4.8: Digital Pictures of the Altera Cyclone Interface Board
Figure 4.9: Theoretical and Experimental Impulse Response for the AD6654 ASIC
The time series was also analyzed by modulating a square pulse by a sinusoid, similar to that of a meteor echo. The non-zero portion of the square pulse has a width of 3 µs, and is modulated by a tone of 30MHz. The mixer was set to 2kHz such that a tone could be analyzed. The frequency of this tone is well within the passband of this configuration. The in-Phase and quadrature channels are shown below in Figure 4.10.

This configuration worked seamlessly. Data rates on the order of of a few kHz to nearly 40MHz could be transferred for hours upon end with the GNU Radio host code, while data rates from a few kHz to 22 MHz could be transferred with the LibUSB software. This test configuration was used to transfer 12GB of data, and no buffer overruns, or anomalies occurring during the transfer.

After interfacing to the AD6654 ASIC the Altera Cyclone board was used to interface to the AD9259 4 Channel A/D converter and perform all signal processing within the Cyclone. A simple 4 channel complex digital receiver architecture was implemented in logic inside the Altera Cyclone. This receiver architecture consists of a mixer, three cascaded decimating Goodman-Carey Filters of order 7, and 8 asynchronous FIFOs. However, what could not be implemented on this board is the LVDS deserializer for the AD9259 A/D converter. There are two reasons why this interface cannot be implemented. The first reason being the LVDS signal pairs are not routed cleanly enough on the board. More information on LVDS is explained in the LVDS section of this thesis. The 100Ω load matching resistors must be placed very close to the pins on the FPGA, and this was not the case for the current design. Another revision of this board could route the resistors closer to the IO pins on the Cyclone. The second reason is a PLL is needed to double the DCO clock frequency which cannot be done unless the LVDS signal are routed through dedicated clock IO pins on the Cyclone. The JOP Design Cyclone board does not have dedicated clock IO pins as user signals, thus, there is no access to the mandatory PLL for deserialization of the serial data from the A/D converter. It should be noted that general purpose IO lines have no access to the clock network in
Figure 4.10: I and Q samples of a processed radar pulse by the AD6654 receiver
the Cyclone however, this is not the case in the Spartan FPGA where general purpose IO lines can be fed into the global clock network via a clock buffer.

### 4.2.0.5 Conclusions

The AD6654 ASIC was used to successfully stream data through an Altera Cyclone FPGA using an asynchronous FIFO on the Altera FPGA. However, after the streaming interface there would be many downfalls to using the AD6654. Amongst these pitfalls include the basic nature of how the AD6654 splits a single digitized signal into multiple channels. This is accomplished by giving each channel a dedicated complex mixer. In order to use this design for the Cobra Radar the receiving channels would have to be multiplexed in frequency so that the received signal is not a linear sum from each received antenna. This is a large design constraint that could be very cumbersome to implement. Secondly, the AD6654 chip downloads configuration data through a proprietary interface by Analog Devices that includes no documentation or source code. Thus, in order to program this chip a custom interface would have to be developed with a new PCB. To design this PCB a significant amount of hardware and routing would have to be done. The AD6654 chip is in a (Ball Grid Array) BGA package which requires expensive PCB software. Having to frequency multiplex each channel, and having to implement a complex PCB for the AD6654 package was deemed too cumbersome to implement. With the ASIC not meeting the specific needs for the COBRA radar system, and implementation issues that would be difficult to overcome different architectures for the digital receiver were explored.

### 4.3 ZestSC1 FPGA USB Board

After deciding to move away from dedicated signal processing integrated circuits such as the AD6654 wideband IF to baseband receiver from Analog Devices, the focus was to move to a more flexible design consisting of only an FPGA. A block diagram of
this system can be shown in Figure 4.11. The ZestSC1 FPGA USB Board from Orange Tree Technologies [19] offers a USB Transceiver and Spartan 3 FPGA on a prototyping PCB. There are 49 I/O lines exported to a user header as seen in Figure 4.12. These pins can be used to interface to the AD9259 A/D converter, with use of an adapter PCB. The Spartan 3 FPGA is capable of receiving LVDS signals at 666Mb/s which is sufficient for the 280-700Mb/s range the AD9259 A/D converter transmits. We do not plan to use the full 50MHz or 700Mb/s data rate offered by the AD9259 but a rate within the range that the FPGA can receive with minimal bit errors.

The Zest SC1 board is ideal to interface to the AD9259 4 Channel A/D converter because the Spartan 3 FPGA has the capability to deserialize multiple signals from the AD9259, digitally mix and filter these signals, then hand the data off to the FX2 chip to interface to USB. A block diagram of this process is shown in Figure 4.11.

The goal for the Spartan FX2 interface logic was aimed be identical to that of the Altera Cyclone board, less the architectural issues between the Xilinx and Altera manufacturers. Obviously, the FIFO cores provided by both FPGA manufacturers is not identical, however their basic behavior is similar. The preliminary design for the Altera logic was implemented in both schematic and VHDL. The VHDL code could be ported directly to the Xilinx design. The logic for the signal processing stages, and
Figure 4.12: ZestSC1 Prototyping FPGA USB Board Block Diagram
deserializing of the encoded serial data from the A/D converter is discussed in the other sections of this thesis. The logic design for the digital receiver was written entirely in VHDL.

The digital picture of the ZestSC1 receiving system in Figure 4.13 shows the AD9259 evaluation board, interfacing PCB, and the ZestSC1 board. The AD9259 has 4 SMA connectors (Sub Miniature version A) for each channel, and another SMA connector for the clock signal for the A/D converter. This board then mates to the ZestSC1 board with a PCB that has microstrip lines and terminating resistors for the LVDS pairs. The ZestSC1 board has a FX2 USB transceiver, Spartan 3 FPGA and memory chip.

There are a couple of hardware issues that are worth explanation on the ZestSC1 board. The first issue being is a means to configure the FPGA when the board is powered on. Currently, the FPGA is configured from another computer, and this will not suffice for deployment in the field. It would be advantageous to have the configuration stored in EEPROM, although, EEPROM is not a part of the ZestSC1 hardware. The configuration of the FPGA can also be done by sending the FPGA configuration file
through the FX2, and using the FX2 to configure the FPGA. This is the method intended for the ZestSC1 board, and is entirely feasible. Another hardware issue of the ZestSC1 is that the user I/O lines are not configured for LVDS signaling. These I/O lines do not have 50Ω microstrip lines leading to the FPGA, and also do not have pads for the termination resistors in close proximity to the FPGA package. The termination resistors will have to be routed to the user I/O header where they header pins to through standard traces to the FPGA. This workaround is shown to be good enough for prototyping. The third issue is the Spartan 3 FPGA interfaces to an external memory chip. This memory is not used and only adds complexity. No provisions need to be taken for this memory chip, however, it is advantageous to know it exists on the board.
Chapter 5

Anomaly in the Streaming USB Interface

An anomaly in the streaming interface with the ZestSC1 board has prevented the ultimate completion of the digital receiver design. Despite our greatest efforts, and diligent ideas for workarounds methods we have been unable to find the exact source or a fix to this anomaly.

In this section we will outline the simplest experiment to replicate the anomaly in transferring data. The meaning of simple means all logic, software, firmware, and hardware not necessary to reproduce the anomaly is left out of the design. The description of this anomaly will attempt to explain all background necessary, and then compare the setup for this system to that of the working Altera board.

In particular this anomaly pertains only to streaming data from the FPGA to the host computer, where the FX2 is an intermediate subsystem. The logic to stream data from the FPGA to the FX2 is meant to be an exact replica of the logic on the Altera Cyclone FPGA which streams data continuously without anomalies, and is known to be a working system. With this experiment there are three subsystems to consider, namely, the FPGA, the FX2, and the host computer.

When considering any anomaly or bug the basic strategy is to isolate the source of the anomaly. In order to isolate the source of the anomaly all components not related to the anomaly must be in some way eliminated from the system. The isolation techniques must also become more sophisticated when different subsystems such as the FPGA,
the USB transceiver, and interface from the USB transceiver to the host computer are tightly coupled, which is the case for the digital receiver. This experiment will use the FPGA to fill the FX2 buffer at a given rate, while the host computer is requesting packets.

We will now elaborate on the configurations for the FPGA logic, FX2 firmware, host software on the Linux Computer, and hardware that reproduces this anomaly. This anomaly occurs only with the ZestSC1 prototyping board. Recall this board has a Xilinx Spartan 3 FPGA, and an FX2 USB Transceiver. The setup for this experiment will be divided into a respective FPGA logic section, FX2 firmware section, host software section, and hardware section. The following items are meant to be a broad overview of a simple configuration to reproduce the anomaly.

- Synthesize logic in the VHDL to transfer a known data word every 32 clock cycles from the 48MHz clock sent from the FX2.
- Have the FPGA write a known the data word when the programmable full flag on the FX2 is not asserted, and 32 clock cycles have occurred.
- Use the modified firmware written in C, as well as the firmware written in 8051 assembler for the FX2 to compare results from a data transfer.
- Use the modified GNU Radio host software, as well as the LibUSB host software to compare results from a data transfer.

When the host computer requests packets from the FX2 transceiver an unpredictable number of packets from the FPGA to the host computer will be transferred before transmission stops unexpectedly. In order to elaborate more on what this description means a detailed outline of the experiment used to produce this anomaly is presented. These three subsystems: the FPGA, the FX2 transceiver, and the host computer, must be performing their specific tasks in order for this experiment to replicate
5.1 FPGA Logic

The task of the FPGA logic is to fill the FX2’s buffer at a rate slower than the USB maximum bandwidth. Here we have chosen an arbitrary rate of 3MHz. This rate is realized by filling 2 bytes every 32 clock cycles of the 48MHz clock, or \( \frac{2 \times 48 \text{MHz}}{32} = 3 \text{MHz} \). Because the Altera Cyclone board transfers data flawlessly with this logic, we have measured the rate to indeed be 3MHz. The behavior of this logic in the FPGA is simple and can be summarized in one sentence. Fill the FX2 with 2 bytes every 32 clock cycles, unless the status of the buffer is full, indicated by the programmable full flag. Below a more in depth summary of the behavior of this logic is itemized.

- A process on the rising edge of the clock will increment a 2^5 bit counter (counter1). Another 2^8 bit counter (counter2) in this process will increment every 2^5 counts, and will be transmitted to the FX2.

- Compare the contents counter1 to zero, when the contents is non-zero a signal named toggle is logic low, when the contents of the counter is equal to zero the signal toggle is logic high. Toggle = 1 when counter1 = 0, else 0.

- Assert the SLWR signal to latch a word into the FX2 buffer, when a logical nor of the toggle signal and the programmable full flag is logic high. We will write a byte when the programmable full flag is not high, and 32 clock cycles have passed. SLWR = 1 when toggle = 0 and programmable_full = 0, else 0. usb_data = counter1 when slwr = 1, elze high_impedance.

When we analyze this data we should be able to see a constantly increasing data stream. That is a sawtooth waveform that ranges from 0-255, and incrementing by 1 when each word is transferred to the FX2. Note that the lower byte of the word is the
count, and the upper byte is all zeros. Thus, the sawtooth function has a period of 512 bytes instead of 256.

Because the FX2 can empty the buffer at 48MHz, and we are filling at a rate of 3MHz we expect the programmable full flag to never assert, however, in the case that it does logic is built in to not overfill the FX2. Since the programmable full flag will assert when 3 packets are committed, and 256 bytes are in the current uncommitted buffer, the FX2 will have plenty of time and buffer space to transfer packets to the host computer without having to assert its programmable full flag.

The FPGA for this experiment uses the clock source from the FX2 Transceiver logic is synchronized with. Thus, all clock signals in this experiment are synchronous. Oscilloscope traces indicate a clean clock signal from the FX2 and FPGA, with no significant amplitude or phase variations. This clock signal enters the FPGA through a general purpose IO pin, and is running at a slow enough rate that a DLL does not need to be used. However, the Xilinx digital clock manager (DCM) [10] has been used and the same results occur.

The DCM is circuitry built around an onboard DLL, and is used to phase lock the incoming clock signal to the global clock network in the FPGA. A simulation in Xilinx Modelsim was performed to ensure the logic discussed in this section was correct. The results from this simulation will not be provided herein.

### 5.2 FX2 Firmware

Two types of firmware for this experiment are used, the modified GNU Radio firmware written in C, and the 8051 assembler firmware. Both of these firmware codes have been verified to work seamlessly with the Altera Cyclone board. The 8051 firmware has more features for debugging. It can be used to re-initialize at any point, whereas the GNU Radio firmware lacks this feature. Also, the 8051 assembler firmware returns important status of the endpoint buffer flags, and byte and packet levels.
Both firmware sets configure the FX2 to be in FIFO slave mode where the FX2 appears to be a generic FIFO to the FPGA. Endpoint 6 is configured for bulk transfers going into the host computer, also known as BULK IN transfers. This endpoint utilizes the AUTOIN functionality for the FX2 device, the only way to assure a bandwidth close to the USB 2.0 specification of 480 Mbps. The following items describe the configuration of EP6 to produce the anomaly. The configuration code is executed after the FX2 has enumerated what is known as the initialization routine in the firmware is shown in the items below.

- Slave mode: The FPGA will manage filling the FX2 endpoint through the programmable full flag, and the SLWR control signal. The FX2 will appear to be a generic FIFO with SLWR as the write enable, and export the programmable full flag to the FPGA.

- IFCLK: The FX2 will output its 48MHz clock for the global FPGA clock.

- BULK IN mode: Packets are sent with the USB Bulk transfer mode to the host.

- AUTO IN: Packets are automatically committed to the USB domain once 512 bytes are received, the 8051 is left out of all bulk transactions after the initialization code has been executed.

- Quad buffering: Endpoint 6 is divided into 4 512 byte buffers, and each buffer is 1 packet.

- Programmable Full: Asserts when 3 packets are committed and 256 bytes remain in the current uncommitted packet.

- Wordwide: When the SLRW signal is asserted the FX2 will latch 2 bytes, or one word, hence, each write fills 2 bytes.
There are many other configuration parameters that do not pertain to parameters for bulk transfers in slave mode. These parameters are not discussed here, namely because there are so many configurations that the firmware must provide to have a properly operating FX2. The firmware that configures the FX2 was originally written by the GNU radio Group [9] in C and compiled using the Small Device C Compiler [31] for 8051 assembly, then modified by Stephan Esterhuizen. Stephan Esterhuizen modified the code to work in another GPIF configuration outlined in this thesis. I then modified the code further to configure the FX2 in slave mode, which is the code being used for this experiment.

The 8051 assembler firmware was written to greatly simplify the modified C firmware. The C firmware code uses nearly 50 different files many of which have no application to the digital receiver. After becoming familiar with the inner workings of the C code, the firmware was re-written in 8051 assembler to better understand the FX2. Re-writing this firmware was invaluable because all the aspects of the FX2 were learned. Without this firmware many doubts about the exact configuration of the FX2 could not be disregarded. As an independent observer one might assume the firmware for the FX2 could be somewhat simplistic. However, the inner workings of the FX2 are quite intricate due to the vast array of the configuration parameters, and interrupt handling for the control endpoint. For the replication of this anomaly both the C firmware modified from the GNU Radio software and the 8051 assembler firmware are used. Both of these firmware sets replicate the anomaly.

5.3 Host Software Configuration

Two different host software versions were also used for the experiment, to ensure the anomaly is not caused by an interaction between the USB hub and the host software. Each host software application was written for its respective firmware. The host software uses two different APIs, namely the GNU Radio USB API [9], as well as the LibUSB
API [8]. The GNU Radio API is much more complex and offers a higher throughput than that of the LibUSB, however LibUSB offers an advantage of simplicity.

The GNU radio host configuration for this experiment uses Stephan Esterhuizen’s software modified from the GNU software radio project [9]. This code is nearly unaltered except for a buffer overrun check that is handled differently, and a transfer rate calculation, that is more accurate. The LibUSB host software is modified to have a small user interface where the initialize function of the device can be called, individual packets can be read, and have their contents displayed, crucial debugging information can be displayed, and multiple packets can be read and displayed. As stated previously this software is simpler that that of the GNU Radio host software, and is much more versatile due to the user interface, and the vendor requests.

Both of these software programs have worked for many experiments with absolutely no anomalies on the Altera Cyclone Board. It should be noted that the 8051 assembler firmware can be used interchangeably between both host software applications, but the C firmware only can be used with the modified GNU Radio host code. This reason is because the added functionality in the C firmware has not been added to provide vendor requests to send debugging and status information through the control endpoint. The inner workings of the host software APIs are the least understood of all subsystems, however, what is known about this software will be documented below.

A packet is abstracted in the USB specification and can be any size, however, for this experiment a packet is 512 bytes, the maximum size for bulk transfers outlined by the USB 2.0 Specification. Typically a bulk read function call on the host computer has the following parameters: a glorified pointer to the USB device, the transfer size in bytes (can be an integer multiple of a packet size), and a timeout indicating how long the host should wait before the transfer takes to complete. In the Linux operating system these transfers are wrappers around the ioctl() function. The ioctl() function manipulates the underlying device parameters of special files. In particular, many op-
erating characteristics of character special files (e.g. terminals) may be controlled with ioctl() requests. These ioctl() function calls interact with the USB hub. The behavior of the USB hub, and the interaction of the USB hub with the operating systems kernel are the particular aspects of the host code that are least understood. As an example, to read 5 packets from the FX2 a simple call to usb_bulk_read() can be performed with 512*5 bytes, no timing is needed for the function call, as all timing is handled by the USB hub and kernel.

The API also provides a means to locate the device on the USB bus, and returns a pointer to a structure containing information about the USB device. Once the device is located on the USB bus the API also provides a means to claim the interface used to transfer data. Based on the product and vendor identification numbers for the device, various drivers in the operating system may claim the device, the API provides this functionality. This functionality is crucial to transferring information from the FX2, but will not be discussed further.

When the host software executes it will request 16384 bytes, or 32 packets at a time, and requests will not stop until the FX2 has fulfilled each of these requests. These requests are known as USB Request Blocks (URBs). The bulk read functionality can simply wait until a request has timed out, or if a previous URB request is pending, another URB request can be inserted into the queue. Without, having the ability to queue URBs, URBs must be requested on each function call for a bulk read, which as a consequence gives a much larger execution time for data to be requested. The GNU Radio code is the only known open source package that offers queuing of URB requests. Also, it should be noted that a request size of 16384 bytes for a URB is used and is known to be a somewhat magic number for the maximum USB throughput. When using a USB API that does not queue URB requests one should request 32 512 byte packets, or 16384 bytes for a near maximum throughput. As an example when a call to a USB bulk read function requests a single 512 byte packet in a loop with a USB API
that does not queue URB requests, the data rate will be around 1.66 MB/s, however, if
in a loop a call to a USB bulk read function 16384 bytes are requested, the throughput
will yield a rate around 25MB/s. Since the GNU Radio code queues URB requests it
has a maximum transfer rate of 40MB/s, much larger to that of LibUSB which uses
no buffering for URB requests, and has a maximum rate of 25MB/s. It should be
noted that changing the number of packets for each USB bulk read function call does
not change the behavior of the anomaly. Various numbers of packets were requested
in effort to fix this anomaly and no different behavior was found. Also, packet sizes
where changed in the FX2 firmware, and this also had no effect. The interrupt transfer
protocol was also tried, and the anomaly also occurs.

5.4 Hardware

There exist two main hardware differences between the ZestSC1 receiving system,
and the Altera Cyclone board. These differences are the FPGA architecture and the
FX2 package. When considering delays within flip-flops in the two devices they are
on the order of hundreds of Pico seconds. The maximum timing specifications for the
Spartan 3, Cyclone II, and FX2 can be shown in table below. Although, the Cyclone II
narrowly meets the timing specifications it has been proven to work, and the Spartan 3
easily exceeds all timing specifications since it is 4 speed grades faster than the Cyclone
II [10],[11], [12]. These timing specifications are for low voltage CMOS logic, where we
are running a 3.3V core, called LVCMOS33 by Xilinx.

The Altera Cyclone uses the Braintechology prototyping board [20]. This pro-
totyping board uses the 56-pin FX2 package [12] (CY7C68013A/ 56-pin SSOP). While
the ZestSC1 uses the 128-pin FX2 (CY7C68013A/ 128-pin TQFP). Cypress Semi-
Table 5.1: Relevant timing delays for the Xilinx Spartan 3, Altera Cyclone II, and the FX2 USB Transceiver

<table>
<thead>
<tr>
<th>Delay Description</th>
<th>Spartan 3</th>
<th>Cyclone II</th>
<th>FX2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay from Input Pin to Flip-Flop</td>
<td>2.31 ns</td>
<td>4.349 ns</td>
<td>-</td>
</tr>
<tr>
<td>Delay from Output Pin to Flip-Flop</td>
<td>2.23 ns</td>
<td>4.289 ns</td>
<td>-</td>
</tr>
<tr>
<td>Clock Period</td>
<td>-</td>
<td>-</td>
<td>20.83 ns</td>
</tr>
<tr>
<td>Data Hold set-up time</td>
<td>-</td>
<td>-</td>
<td>9.2 ns</td>
</tr>
<tr>
<td>Flag Propagation set-up time</td>
<td>-</td>
<td>-</td>
<td>9.5 ns</td>
</tr>
<tr>
<td>SLWR set-up time</td>
<td>-</td>
<td>-</td>
<td>18.1 ns</td>
</tr>
</tbody>
</table>

ductor emphasizes these packages are fully compatible with one another. The 56-pin and 128-pin packages have identical functionality for device pins, and all registers and architecture are identical. The primary difference between the 56-pin and the 128-pin package are the added ports used to interface to external memory. Both the 56-pin and 128-pin devices use an enhanced Harvard Architecture so data and code can be stored in the same memory space. The 128-pin package adds a 16-bit address bus, an 8-bit data bus, as well as the address/data bus control signals [22]. When configuring the ZestSC1 all the extra pins that the 128-pin package offers are left unconfigured, and the FPGA keeps these pins in high impedance. The FPGA has been also configured to output the correct logic level for these pins that route to the FX2; however, no different results have been recorded. Cypress semiconductor cites no errata associated with the anomaly that we are citing.

5.5 Results

In this experiment 1MB of data is requested from the host computer using the GNU Radio host code, as well as the GNU Radio firmware modified for slave mode. Upon requesting 1MB of data the device fills an unpredictable amount of data before the anomaly occurs, the FX2 fails to answer host requests after an unpredictable amount of time. Transfer sizes range from 4K-1MB, however, the amount of data that will be transferred to the host will always be an unpredictable size. Using an invaluable tool
by Xilinx called Chipscope the state of logic signals can be captured and stored. This data can be analyzed to find the state of the logic signals when the anomaly occurs. In order to fully explain what happens when the anomaly occurs, we will outline the entire timeline of events that produce the anomaly.

- Firmware for the FX2 is loaded. The host is not requesting any packets.

- The FX2 firmware resets and empties its EP6 buffer, this reset is also sent to the FPGA for a logic reset.

- The FPGA fills endpoint 6 on the FX2 until the programmable full flag is asserted. The host is still not requesting packets.

- Endpoint 6 filled to its desired level and the programmable full flag remains asserted. The FPGA will wait until the programmable full flag is not asserted in order to write more bytes. This desired level can only decrease when the host requests and receives a packet from the FX2, which will allow the programmable full flag to de-assert.

- The host now begins requesting packets from the FX2, and stops requests under one condition, that 1MB of data is received. The host requests are much faster than the FPGA can fill the buffer. Once these requests start we see the programmable flag de-assert.

- Packets are received, until, at an unpredictable amount of time the programmable full flag, or full flag, goes high, then never goes low again.

The last condition above should never happen. The FX2 has packets armed and ready, indicated by an assertion of the programmable full flag, and the host is requesting those packets, however the FX2 stops sending packets. Since, this condition is met no data can be transferred. The anomaly occurs when we see the programmable full flag
go low when packets are first requested, then unexpectedly assert, and stay asserted indefinitely. It can be noted that the programmable flag can assert during a transfer, however, it is expected to become logic low at some later time, because the host is emptying the endpoint. Chipscope was used to find the case discussed above, a timing diagram can be seen in Figure 5.1. What is not shown in Figure 5.1, are the logic states after the programmable full flag asserts. They are not shown because no state changes occur after the programmable full flag asserts, even while the host is requesting data that is available, indicative of the programmable full flag assertion. On one hand the FX2 indicates there are packets armed and ready by assertion of the programmable full flag, yet on the other hand the host can receive no data. This predicament inhibits data transfer.

Shown in Figure 5.1 the SLWR signal can be seen asserting, indicating a write of 2 bytes to endpoint 6. The signals rd_en, and usb_flagd, are unused in Figure 5.1. At some undetermined time the programmable full flag (usb_pf) will assert indicating the FX2 has 3 packets committed and ready to be sent to the host, and 256 bytes in the current packet, the setting of the programmable flag. After the programmable flag goes high the host software is unable to empty packets from the FX2.

Recall that the value of a counter is sent to endpoint 6, we expect the contents of the packets to be a sawtooth function going from 0-255, every 512 bytes. We can analyze the contents of this data and compare it contents to the Altera Board receives.
Figure 5.2: Top: Ten 512 byte packets received with bit-errors and discontinuities from the Xilinx Board. Bottom: Ten 512 byte packets received correctly from the Altera Board
From Figure 5.2 (Top) we can see that the data in the received sawtooth function is erroneous, and counter values are skipped. We can compare this plot to that of the expected received sequence shown in Figure 5.2 (Bottom), which is the sawtooth function transferred from the FPGA to the FX2, then from the FX2 to the host. Many provisions have been made to analyze this erroneous data, however, no pattern has been found. The erroneous data stream from Figure 5.2 can be replicated for the modified GNU Radio host code and firmware, as well as the LibUSB host software and 8051 assembler firmware. It can further be replicated by running the modified GNU Radio host software with the 8051 assembler firmware.

The LibUSB host software and 8051 assembler firmware provides a simple user interface to analyze valuable debugging information. The status of the FX2 with respect to internal registers and logic of the pins will disagree after the anomaly has occurred! A typical example after the anomaly has occurred will be that the number of bytes in the FIFO is 256, and 2 packets are committed. The programmable full flag will be logic high, although, the programmable full flag can only assert when 3 packets are committed and 256 bytes are in the current uncommitted packet. The reason as to why or how this happens is unknown.

The status of the bulk read request on the host computer side will timeout, indicating the function call has waited the specified time, and the FX2 has failed to transfer data. This is expected because the FX2 fails to send information. The timeout value here is set to 1 second, more than enough time for the FX2 to fill 512 bytes. The transfer rate is not an too slow to cause a timeout from the USB bulk request.

A skeptic could argue that the FX2 firmware could be entering an unknown state, however, by using a variable to monitor what code the FX2 has executed it was shown that the FX2 firmware is sitting in an idle state and not executing undesired instructions, the expected behavior of the firmware.
The notion that the host computer has faulty USB hardware can be eliminated since the Altera Cyclone board is used to consistently check transfer data seamlessly, with no skips in the data sequence, or no abrupt transfer halts.

This anomaly could be a misconfiguration between the FPGA to the FX2, or possibly between the FX2 and the host computer. A mishap in communication between the host computer and the FX2 is nearly impossible to isolate without a USB 2.0 Bus analyzer. The exact source is still unknown. The most baffling part about the anomaly is that transfer sizes are random.

Recall that the transfer rate throughout this experiment was 3MHz, or 1/32nd the 48Mhz clock rate. It should be noted that when this clock rate is changed to 48MHz the anomaly does not occur where the transfer rate abruptly stops! That is host data requests of any size can be transfered without the programmable full flag asserting in the middle of host requests, and never de-asserting. The contents of the data is different depending on which host software is used. We can repeat the same experiment described in this section, only changing the fact that the counter value for the sawtooth waveform will be transferred on every rising edge of the USB IFCLK. The sawtooth waveform is captured perfectly for the modified GNU Radio software, however, problems occur with the LibUSB host software. This can been seen in Figure 5.3.

When examining Figure 5.3 keep in mind data is only sent when the FX2 indicates that it is not full, indicated by the programmable full flag. What can be gained from this plot is that the LibUSB software has periods where counter values are not transferred. This is indicated by where the sawtooth waveform has a derivative of zero. This difference in host software is due to queuing of the URB requests. It is however, quite strange that the anomaly does not occur when data is transferred at full rate. All other data rates, including half the maximum throughput have reproduced the abrupt data transfer anomaly.
Figure 5.3: Top: Sawtooth Waveform sent at 40MB/s with the Xilinx FPGA using GNU Radio USB Library. Bottom: Sawtooth waveform sent at 40MB/s with the Xilinx FPGA using LibUSB.
5.6 Debugging Methods

The preliminary stages to isolate the anomaly the state of signals was output on general purpose pins, and or connected to LEDs. This is a fast way to eliminate any large mistakes, use the LEDs to show the logic levels of the individual signals, and an oscilloscope to trigger on interesting events that occur. This method becomes very difficult when trying to analyze many signals at once, and this method is totally impractical when using the oscilloscope to trigger off of multiple signal transitions.

The logic analyzer is helpful in analyzing many signals at once. The logic analyzer used however, did not have a desirable sampling frequency, and was difficult to analyze the state of the logic signals before and after a trigger event. Late in the debugging stages Cody Vaudrin assisted me in using Chipscope for analyzing the signal states. This software can be configured from a graphical interface. A logic core is inserted onto the FPGA that saves all the state transitions. This core is configured to trigger off a user defined event. When this event occurs the memory contents storing all state transitions are uploaded through the JTAG port and can then be analyzed using Chipscope. This is the best debugging tool when debugging an FPGA. Chipscope can easily save many signal states, and trigger off a sequence of complex logic events, then graphically show the results. This tool was used to find the state at which the anomaly occurs, however, it was not able to isolate the source of the anomaly.

The internal state of the FX2 could be found by changing the FX2 firmware. The firmware was changed to send the status of registers pertaining to the hardware through the configuration endpoint via vendor requests [22]. This firmware, and accompanying hardware was used to send packets one at a time and analyze the status of the registers. The status would mainly consist of whether or not an endpoint is stalled, and the number of bytes and status of the flags for the endpoint. Interrupts pertaining to events in the FX2 were also traced to update variables for debugging. The FX2 will interrupt on
USB bus error events. No USB bus errors were ever detected when the anomaly occurs. In the USB Technical Reference Manual [22] it outlines that the host can flood the USB bus with transfer requests and reek havoc on the FX2. Cypress Semiconductor fixed this problem in some sense for OUT endpoints by allowing functionality for the USB host send a “ping” to determine if the FX2 has a packet ready to be sent over the bus. There are also “In-Bulk-Nak” or (IBN) for IN endpoints that are sent by the FX2 when the host requests a slew of bulk requests. These IBNs were counted by an ISR and a number on the order of 10 where found when the anomaly occurred. This number could play a role in the anomaly. Here is what the EZ-USB TRM [22] outlines for an IBN. “Until the endpoint is armed, a flood of IN-NAKs can tie up bus bandwidth. If the IN endpoints arent always kept full and armed, it may be useful to know when the host is knocking at the door, requesting IN data.” This is precisely what was done, to count the number of times the host “knocks at the door” of the FX2 when requests are made and no packets are available. Unfortunately, a comparison to the Altera Cyclone’s system has not been performed.

The logic levels of the pins on the FX2 where checked to ensure they meet LVC-MOS 3.3 voltage levels. There were some pins that were found outside the threshold for the LVCMOS 3.3 high and low logic level. All pins that are connected to the FPGA were also placed in their correct logic levels, whether it be logic high, logic low, or high impedance. It has always been somewhat of a suspicion as to whether the FX2 was sourcing too much current and consequently negatively affecting the internal logic.

Another debugging tool used was Cypress technical support. After opening a large number of technical support cases in response to this anomaly and not receiving any valuable information, I finally received a phone number for the FX2 experts. I spoke with these experts and was able to have them test our slave mode configuration software on their hardware configuration. They were able to verify the correctness, and were able to stream data with no problems. Cypress does not support any USB API, or
firmware for the Linux Operating system, thus, they are unable to help with any other software or firmware issues.

Another tool used for debugging was the Xilinx Simulator and ModelSim by Mentor Graphics. This software is used to verify the behavior of logic in the FPGA. Test processes for logic are outlined and the state of the logic is shown versus time. This is very useful, and time saving method to greatly aid in the behavior of the logic.

A desired tool for debugging is a USB 2.0 Bus analyzer which analyzes the traffic on the bus as transactions between the USB hub and the FX2 occur. A software trial version for this purpose was used, however, it was only for the Windows operating system, and thus no Linux Host Code could be used.
Chapter 6

Summary and Future Work

This section will outline the large parts of the research on the digital receiver project. These items show the direction of my research throughout this project, and are meant to be a roadmap to sections in this thesis. These items represent significant progress points in the thesis.

• Learn the inner workings of the AD6654 Wideband ASIC digital receiver.

• Research the methods Stephan Esterhuizen used for his streaming USB interface, and augment his interface to work with our system.

• Research dedicated hardware and FPGA methods to stream data from an ASIC to the USB bus.

• Learn VHDL synthesize logic in an FPGA, and use it in our design. Build a PCB board to interface the ASIC digital receiver, an Altera Cyclone, and a USB Prototyping Board.

• Successfully stream data through the ASIC digital receiver, to a host computer by means of an FPGA equipped with an Asynchronous FIFO.

• Discover the pitfalls of designing a digital receiver with this particular ASIC, and change our momentum toward designing a digital receiver with only A/D converters, an FPGA and a USB interface.
• Begin researching multichannel A/D converters and decide the Analog Devices AD9259 4 channel A/D converter is best for our design. Research various FPGA architectures that can sustain the requirements of a digital receiver.

• Find a prototyping board called the Zest SC1 that has an on board USB transceiver, and a Xilinx FPGA that is ideal to interface to the AD9259 A/D converter.

• Research methods to interface the FPGA and A/D converter which must communicate through the LVDS signalling scheme.

• Build a PCB interface board, with LVDS transmission lines to interface the AD9259 A/D converter to that of the Zest SC1 prototyping board.

• Design the digital receiver architecture for the Xilinx FPGA, and run into an anomaly with the USB streaming interface.

• Perform extensive debugging on the anomaly. Research work around methods for the streaming interface.

• Compare the working streaming interface of the Altera FPGA and FX2 duo, to that of the Xilinx FPGA and FX2 duo.

• Design digital receiver logic on the Altera board.

• Realize a hardware aspect of the Altera FPGA inhibits deserialization from the A/D converter. Fine tune the constraints an FPGA must have to deserialize the serial bit stream from the AD9259 A/D converter.

• A solution to fix the anomaly for the digital receiver using the Xilinx board is still unknown.

This thesis will cover the material researched and used for the design of multiple digital receiver architectures, as well as the various hardware ideas we implemented and
used for the digital receiver. Our constraints and design issues matured by using and studying the various ways to implement the digital receiver, and those ideas and designs are covered within this document. There are many hardware and implementation choices for implementation of a digital receiver, and much of the work done was to find the best design. Also, a large anomaly is present in the streaming interface for our most mature digital receiver design. The analysis and background of this anomaly is included herein.

Before I began this thesis I was interested in signal processing in embedded systems, and the digital receiver presented an interesting opportunity. Having a background in signal processing and embedded systems gave me the required background to start designing the digital receiver. During this thesis I learned much more than I had expected, primarily because of the strange anomaly that occurs in the streaming interface in the ZestSC1 prototyping board. Having an anomaly of this nature takes a tremendous amount of time to isolate, and requires extensive understanding of all parts of the digital receiver. As well as being able to use and understand all the design tools for the digital receiver. From this thesis I learned valuable debugging skills for digital circuits.

I previously had been exposed to FPGAs, however, was completely new to VHDL, as well as signal processing with an FPGA. FPGA design takes knowledge of many intricacies about the hardware and software. Because I do not have much of a digital logic background, learning how the hardware description language translates into the digital logic was the most challenging aspect of FPGA design. By implementing many different parts of the digital receiver architecture in an FPGA I was able to get a grasp how the hardware description language can be translated into sequential and combinational logic.

As the digital receiver architecture changed, and we tested different designs I learned a great deal about various A/D converters and FPGA architectures. Having the opportunity to fully rewrite the firmware for the FX2 gave me an in depth under-
standing of how complex microcontroller driven integrated circuits function. The FX2 architecture is deceivingly complex, as the 8051 assembler firmware is nearly 2000 lines. Learning the USB system for this thesis took a large share of the total work.

There is a great deal of work still needed to upgrade the COBRA meteor radar system with digital receivers. The COBRA meteor radar system is dependant on more than four channels, however the AD9259 evaluation board only provides four channels. There are two options have been explored to double the amount of channels. The first option is to modify the daughter board to interface two AD9259 evaluation boards onto a single ZestSC1 board. This option is entirely possible and a full implementation of 8 channels on the Spartan 3 could be easily met due to the number of logic gates. The second option is to use multiple device capabilities of the USB bus, and have two digital receivers running on the same bus. Having two USB devices on the USB bus could theoretically be extended to many receivers running simultaneously. This option requires that the sum of the data rates from both receivers does not exceed the maximum throughput of 40MB/s, and is a primary advantage to using the USB bus. In order to implement two receivers running on the same bus, the devices would be enumerated with the same vendor identification numbers, and different product identification numbers. The host software requires multiple instances of the FPGA/USB board, and will request packets from each board. Being able to have many instances of the digital receiver on the same bus has many advantages. By having multiple instances of FPGA/USB boards for the host software it becomes and elegant and simple way to add and remove channels from the digital receiving system.
Bibliography


[26] Vladimir Dergachev, FX2 Device Programmer software


[34] Arecibo Observatory (National Astronomy and Ionosphere Center), http://www.naic.edu/

