

CRUSH: Cognitive Radio Universal Software Hardware

A Thesis Presented

by

George Fredric Eichinger III

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

in the field of

Communications and Digital Signal Processing

Northeastern University
Boston, Massachusetts

April, 2012

This work is sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. The opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

© Copyright 2012 by George Fredric Eichinger III
All Rights Reserved

Acknowledgments

I would like to thank my advisor Professor Miriam Leeser for her support and dedication throughout my time at Northeastern. Without her encouragement and guidance this thesis would not have been possible. I would also like to thank Professor Kaushik Chowdhury whose class introduced me to the concept of Cognitive Radio and fostered many of the ideas in this thesis.

I'd also like to thank MIT Lincoln Laboratory for their support of my Master's degree through the Lincoln Scholars Program. Additionally I would like to thank my colleagues at Lincoln and the Reconfigurable Computing Laboratory for their advice and encouragement. Specifically, I could not have done this without the help, advice and support of my Lincoln advisor, Scott Bailie.

I would like to thank my parents George and Donetta for always believing in me and supporting me throughout my college education.

Last, and most importantly, I would like to thank my wife Lindsey and our cat Bailey. Without their love and support I would not have made it through the many late nights and countless hours of work.

Abstract

The FPGA is an integral component of a software defined radio (SDR) that provides the needed reconfigurability for dynamically adapting its transceiver and data processing functions. Because of the desire to process data faster and with less latency, researchers are looking at FPGA-based SDR. Our architecture, called CRUSH (Cognitive Radio Universal Software Hardware), is composed of a Xilinx ML605 connected to an Ettus USRP through a custom interface board, allowing flexible data transfer between them. In addition, we provide a framework that supports ease of use, independent programming on both devices, and integration with software running on the host. To demonstrate our platform we implemented spectrum sensing, a key step in determining channel availability before transmission in dynamic spectrum access networks. Spectrum sensing is implemented on CRUSH using FFTs for a 100x speedup; the complete sensing cycles is 10x faster than the same design without CRUSH. By reducing the load on the host and allowing a powerful FPGA extension for off-the-shelf devices, CRUSH enables advances in both protocol design and reconfigurable hardware targeting radio applications.

Contents

1	Introduction	1
2	Background	4
2.1	Software Defined Radio Platforms	5
2.1.1	USRP	5
	USRP N2x0	6
	Other USRPs	7
	Daughterboards	7
	Software	8
2.1.2	WARP	8
2.2	Cognitive Radio	10
2.3	Spectrum Sensing	12
2.4	Conclusions	14
3	CRUSH Platform Overview	15
3.1	Hardware Overview	16
3.1.1	Ettus Research USRP N210	16
3.1.2	Xilinx ML605 Development Board	17
3.1.3	Custom Interface Board	18
	FMC	19
	MICTOR Connectors	20
	MIMO Connector	21
	Impact on USRP Functionality	22
3.2	HDL Overview	22
3.2.1	USRP HDL Framework	22
3.2.2	Interface between USRP and ML605	26
	DDR I/Q Bus	26
	Control Bus	27
3.2.3	ML605 HDL Framework	28
3.2.4	Interface between ML605 and Host	31
3.3	Software Overview	33
3.3.1	Host Software	33

	MATLAB	34
	Serial Port	35
	UHD	36
3.3.2	ML605 Software	37
3.4	Conclusions	37
4	Spectrum Sensing	39
4.1	Related Work in Software Spectrum Sensing	40
4.1.1	Spectrum Sensing Surveys	40
4.1.2	Spectrum Sensing in Software	41
4.2	Spectrum Sensing Algorithm in Software	41
4.3	Related Work on Hardware Spectrum Sensing	42
4.3.1	Spectrum Sensing on COTS Hardware	43
4.3.2	Spectrum Sensing on Custom Hardware	43
4.4	Spectrum Sensing on CRUSH	44
4.4.1	FFT	45
4.4.2	I and Q Magnitude	46
4.4.3	Thresholding	47
4.4.4	Fixed point Conversion	47
4.4.5	Verification	48
4.4.6	Configurability	49
4.5	Conclusions	50
5	Experimental Setup and Results	51
5.1	Functional Verification	52
5.1.1	USRP Master Reference	52
5.1.2	CRUSH FFT Verification	53
5.2	Raw FFT Timing	56
5.2.1	Testing Setup	56
5.2.2	Detailed look at 256 point FFT on CRUSH	58
5.2.3	Overall Timing Results	59
5.3	End to End Timing	62
5.3.1	Testing Method	63
	Test Setup	63
	Precision Program Timing	64
	UDP Packet Control	65
5.3.2	Results	66
5.4	Available FPGA Resources	67
5.5	Conclusions	69

6	Conclusions and Future Work	70
6.1	Conclusions	70
6.2	Future Work	71
A	List of Acronyms	72

List of Figures

2.1	Ettus Research USRP N210	6
2.2	WARP MIMO Kit and Accessory Boards	9
2.3	Spectrum Holes Diagram	10
2.4	Cognitive Cycle	12
2.5	Spectrum Sensing Algorithm	13
3.1	CRUSH Platform	16
3.2	Xilinx ML605 Development Board	18
3.3	Custom Interface Board	19
3.4	USRP HDL Framework	23
3.5	Relative Timing Diagram for one Control Bus Transfer	28
3.6	ML605 HDL Framework	28
3.7	CRUSH Demo: 70 MHz Center, 73 MHz CW tone, 256 point FFT . .	34
3.8	CRUSH Serial Interface	36
4.1	Spectrum Sensing Algorithm in Software	41
4.2	Spectrum Sensing Algorithm on CRUSH	45

4.3	User Block for Spectrum Sensing Algorithm on CRUSH	45
5.1	USRP Test FFT Data	55
5.2	CRUSH FFT Data	55
5.3	Modified Design for Timing Tests	57
5.4	Chipscope of 256 point FFT	58
5.5	Host versus FPGA Runtime	60
5.6	Roundtrip CRUSH Time	67
5.7	Overall Timing Comparison	68

List of Tables

3.1	Control Bus Addresses	25
3.2	Control Bus Modes	25
3.3	Host to ML605 Packet Structure	32
3.4	Modes for ML605 Packet	33
5.1	Timing Analysis of 256 point FFT	59
5.2	Summary of FFT Timing Results	62

Chapter 1

Introduction

There is an increasing interest in using Software Defined Radios (SDRs) for real-time processing and for more and more sophisticated algorithms. The Universal Software Radio Peripheral (USRP) [1] is a widely proliferated SDR platform used for implementing radio schemes. There is a wide variety of research using these devices with both software and hardware implementations [2][3][4]. However, real-time processing is not possible for software implementations with the USRP due to the latency of transmitting data between the host and the USRP over Ethernet. The solution is to accelerate algorithms in reconfigurable hardware that is connected directly to the USRP platform.

The USRP has an on-board FPGA, but it is small and has very little space for user functions as it already contains the logic needed to implement existing radio features. Additionally, it is written in such a way that it is difficult for a novice user to modify the code. An external FPGA board allows for more processing close to the radio front end without the constraints of using the existing FPGA. There are

several different radio front ends, including the USRP, as well as different FPGA boards. New FPGA boards tend to be introduced at a faster rate than radio front ends, but both are rapidly changing over time.

This project decouples high end FPGA processing from the agile radio frequency (RF) front end so that either device can be upgraded independently. Cognitive Radio Universal Software Hardware (CRUSH) is a platform consisting of a USRP, Xilinx FPGA board and Custom Interface Board (CIB) as well as FPGA designs and software support to allow the easy integration and use of the platform. While CRUSH is demonstrated with an Ettus N210 and a Xilinx ML605 board, the framework is designed to easily connect to other radio front ends and FPGA cards. This means that a university could pair their USRP 2's purchased in 2008 with a brand new Xilinx KC705 FPGA development board using our Custom Interface Board (CIB) and the CRUSH concept to perform leading edge FPGA SDR research without significant cost increases.

The main contribution of this thesis are:

- The CRUSH platform, which enables FPGA acceleration of cognitive radio applications using standard parts that are inexpensive, widely available, and widely used,
- A framework of hardware and software on the USRP, FPGA and host so that CRUSH is easy to use, and
- The use of CRUSH to implement spectrum sensing.

Dynamic reconfiguration of the transmission and processing parameters is a defining aspect for SDRs. Our platform allows a real-time evolution of both the processing algorithm and the software defined components, by independently altering the code for the component blocks of the system. CRUSH has the ability to integrate up to three off-the-shelf radio front ends with a single FPGA board, thereby creating a *super radio* with significantly increased bandwidth and processing capabilities. This provides an unprecedented level of flexibility at very low cost and allows new insights on the tradeoffs between operational overhead and performance gain when using multiple co-located radios.

The remainder of this thesis is organized as follows: Chapter 2 covers the background required for CRUSH. This includes current software defined radio platforms, an overview of Cognitive Radio and the basics of spectrum sensing. Chapter 3 details the CRUSH platform hardware, HDL and software. Chapter 4 explains how our particular spectrum sensing algorithm is coded in software and then how we implemented it in hardware on CRUSH. Next, in Chapter 5, we see the results of the testing performed on CRUSH. We performed three different levels of testing on CRUSH including functional verification, FFT timing and end to end system timing. Finally, in Chapter 6, we conclude the thesis and include directions for future work. A list of acronyms is included as an appendix to help the reader.

Chapter 2

Background

This chapter contains background information necessary for the understanding of this thesis. Specifically, we discuss state of the art hardware platforms for research in Software Defined Radio (SDR) and Cognitive Radio (CR) followed by a short introduction into the area of Cognitive Radio and how the work presented fits into the larger picture. Finally, we discuss the specifics of the spectrum sensing algorithm implemented in this thesis.

2.1 Software Defined Radio Platforms

There are a number of SDR platforms currently on the market for use by researchers. In this section we describe the most popular current SDR platforms and the features that make them unique. Specifically, we discuss the two most popular SDR platforms, the Universal Software Radio Peripheral (USRP) by Ettus Research [1] and the Wireless Open Access Research Platform (WARP) by Rice University [5]. There are additional SDRs that have been used in research that we do not discuss such as the Berkeley Emulation Engine (BEE) [6] line of products and Networking over White Spaces (KNOWS) by Microsoft Research [7]. Additionally, there are several commercial products that are advertised as capable of SDR such as FPGA boards from Pentek, Lyratech, 4DSP, Acromag and Innovative Integration. However, these do not include a framework supporting existing SDR research.

2.1.1 USRP

The USRP is a line of products from Ettus Research that is used to implement various SDR schemes. The device has several different versions depending on the intended usage. A picture of the USRP N210 is shown in Figure 2.1 and is representative of the size and appearance of the whole line. The USRP N210 is the SDR used in our research.

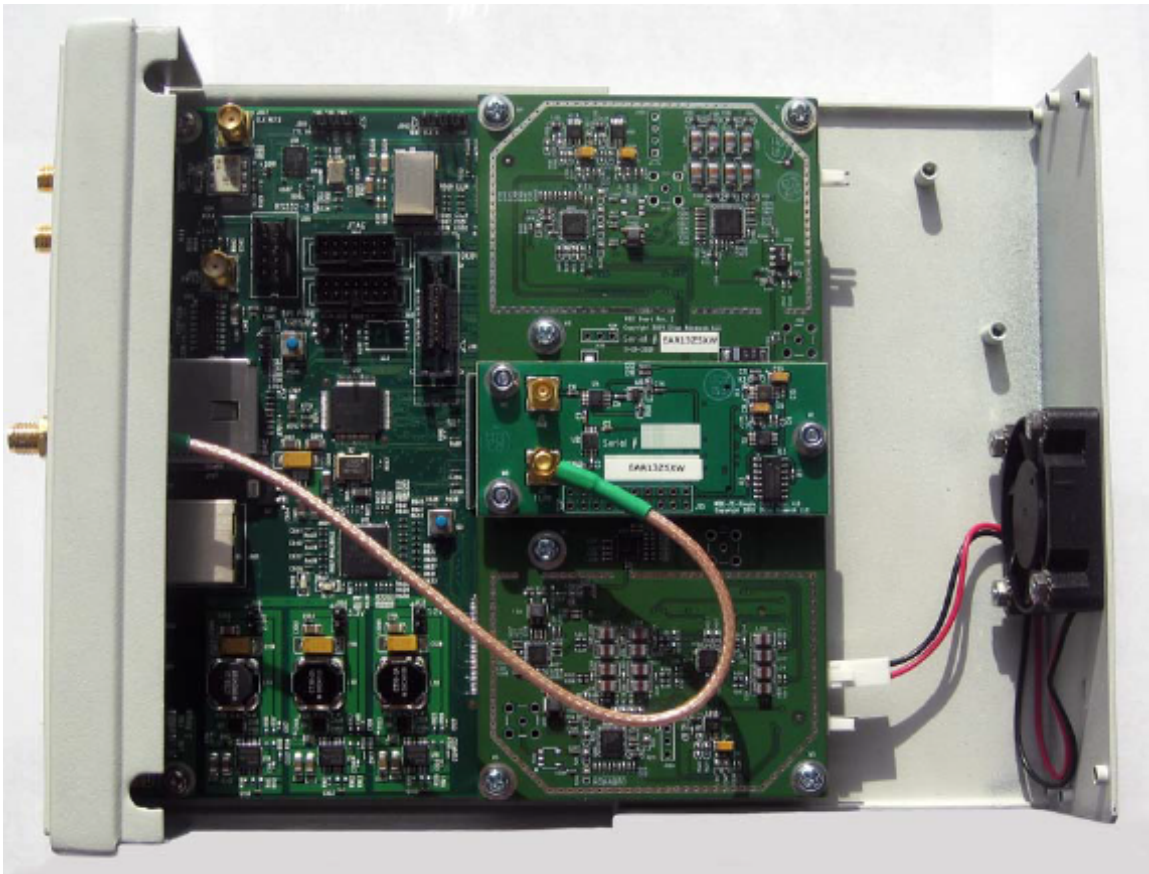


Figure 2.1: Ettus Research USRP N210

USRP N2x0

The N2x0 series is the flagship USRP product and the device used for CRUSH. It interfaces with a host computer via a gigabit Ethernet link (GbE) that is connected to a soft core on an FPGA. The device contains a Xilinx Spartan 3A-DSP series FPGA. Depending in the model this can either be the 3A-DSP1800 (N200) or the 3A-DSP3400 (N210). All of the logic and control of the system is programmed on the FPGA. The baseboard contains two 100 Mega Sample Per Second (MSPS) Analog to Digital Converters (ADCs) and two 400 MSPS Digital to Analog Converters (DACs).

The device has a transmit and receive daughterboard connector that is compatible with any of the USRPs Daughterboards (Sec. 2.1.1). The N2x0 device supports up to 25 MHz of 16 bit In Phase (I) and Quadrature Phase (Q) data to the host over the GbE connection. There is also a Multiple Input Multiple Output (MIMO) port that allows multiple USRPs to be connected together. This device is an upgrade from the older USRP2 which utilized a smaller FPGA with fewer features.

Other USRPs

The current Ettus Research product line includes two other USRPs, the E1x0 series and the B1x0 series. The E1x0 series is one of the latest to be released by Ettus Research and it combines an onboard embedded processor with the radio frequency (RF) features and design of the standard USRP. It has slightly reduced bandwidth capabilities but the advantage of this device is that no other computer is required to perform radio functions. The B1x0 series devices are designed as a legacy support product replacing the USRP 1. The distinguishing feature of the B1x0 series is the USB2 host interface, which limits the bandwidth to 16 MHz. The original USRP 1 device was USB based with similar features and this product provides that legacy capability and price point but with a modern FPGA and updated software support.

Daughterboards

The USRP has a variety of daughterboards that are available for the end user. The USRP contains the software and HDL required to implement an SDR as well as the ADCs and DACs to receive and transmit. However, the designers of the USRP

have moved the RF front end of the device to daughterboards. This design choice allows one device to support many different use cases through application specific daughterboards. Also, since all of the source files and schematics are published, it is possible for users to create custom daughterboards. USRP daughterboards can either be receive only or transceivers (transmit and receive). In both groups, they range from narrowly tuned for specific applications to wideband for general use.

Software

An open source driver platform called USRP Hardware Driver (UHD) [8] is used to communicate with the USRP. UHD is a robust, multi-platform driver that utilizes the BOOST open source libraries [9]. For analyzing the USRP data and controlling the device, there are three main options. The first is GNUradio, an open source SDR platform that provides signal processing blocks for implementing SDRs [10]. It is not specifically linked to USRP but it does fully support the USRP platform. Additionally, The Mathworks MATLAB supports the USRP through its Simulink package [11]. Finally, applications can be written in C++ directly based on the UHD framework.

2.1.2 WARP

Wireless Open Access Research Platform (WARP) is the SDR platform developed and maintained by Rice University for prototyping advanced wireless networks [5]. The WARP system is comprised of three different parts, the hardware, the platform

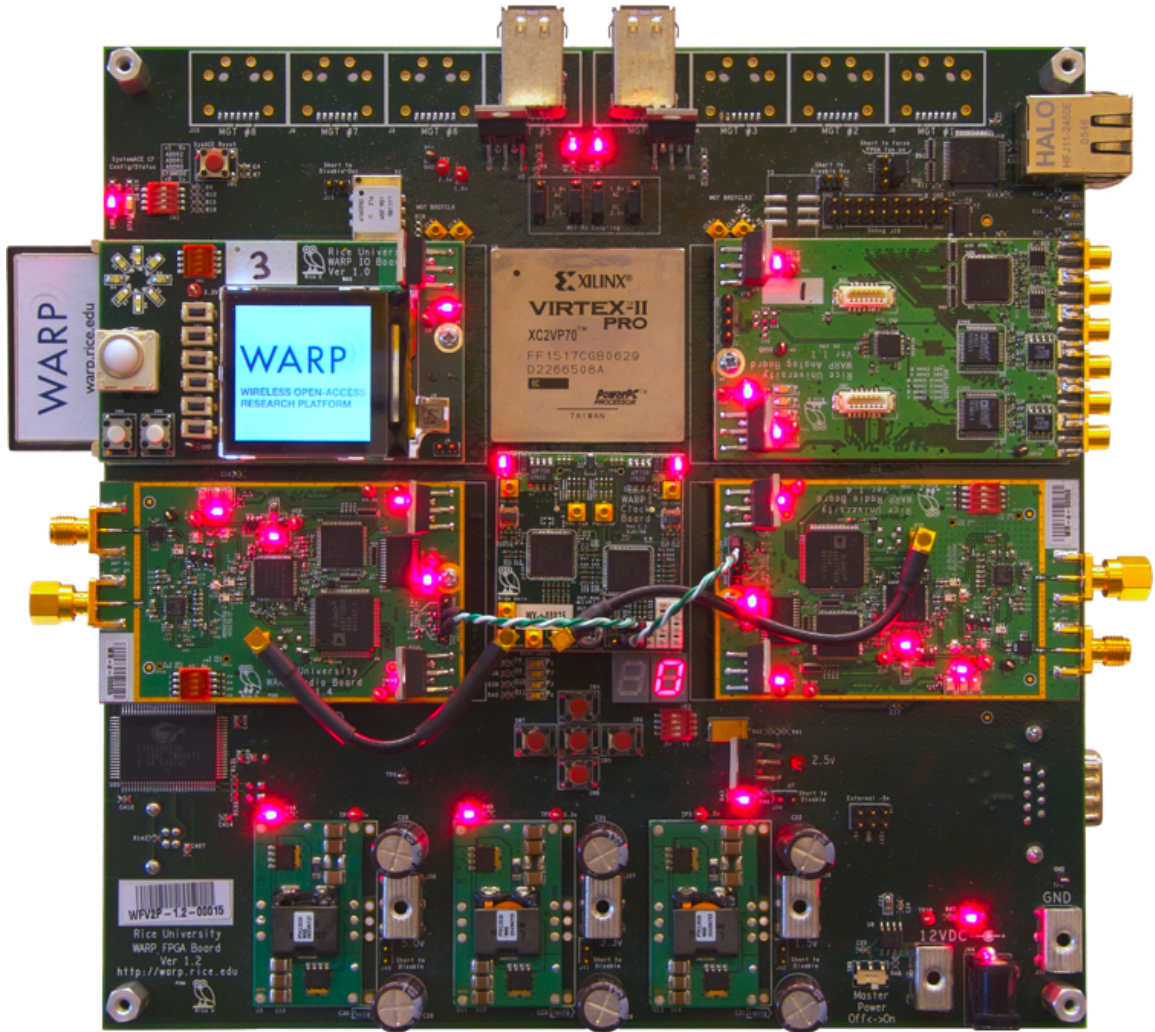


Figure 2.2: WARP MIMO Kit and Accessory Boards

support packages and the specific research applications.

In terms of the hardware, the system is broken up into the FPGA main board and various daughterboards that plug into the main board. The latest model utilizes a Xilinx Virtex 4 FX100 FPGA which includes an embedded Power PC (PPC) Core. The base board supports four daughterboard slots as well as GbE, SATA and various other computer protocols. Unlike the USRP, the ADCs and DACs are on the WARP

daughterboards instead of on the main board. This means that their daughterboard connectors are more general and can support a greater variety of boards than just RF.

Compared to the USRP, the WARP platform has more flexibility in terms of the number of daughterboards and the space for HDL. However, the WARP platform is several times more expensive than the USRP. Additionally, it is based on the FX series Xilinx FPGAs that include an embedded Power PC core. This type of FPGA has been discontinued by Xilinx and could be the reason why the platform is still using Virtex 4 level devices. Currently, Xilinx has already released boards utilizing Virtex 7 chips, three generations past the Virtex 4.

2.2 Cognitive Radio

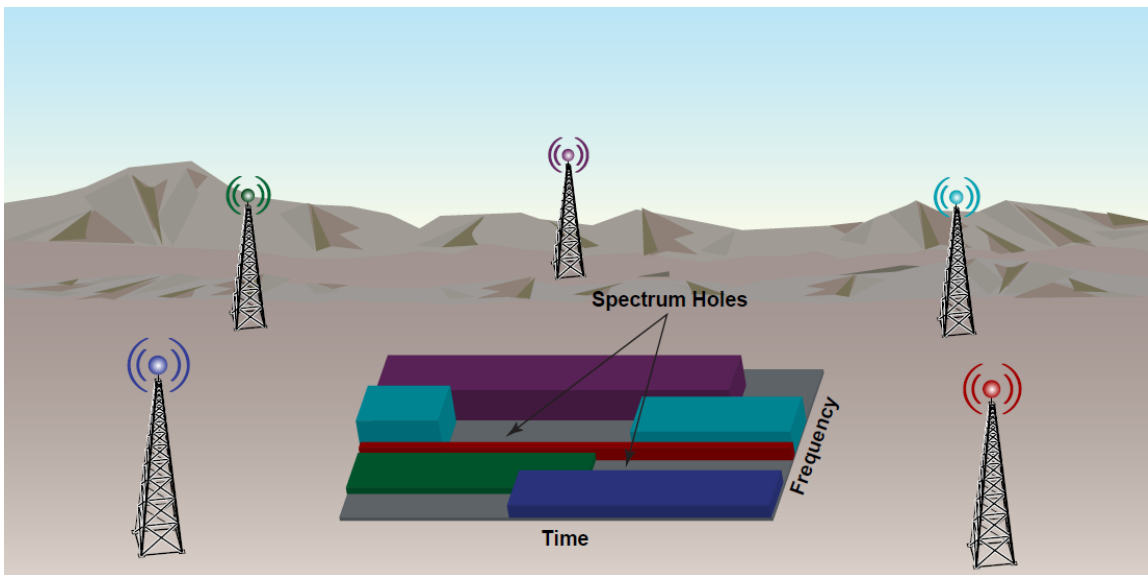


Figure 2.3: Spectrum Holes Diagram

Cognitive Radio (CR) is the concept that a SDR can opportunistically utilize available RF spectrum when the primary user is not present. To better understand this idea we can imagine a scenario of several towers transmitting and view their spectrum representation (Fig. 2.3). Each tower is color coded and represents one primary user (PU). A PU is someone who is licensed to use a particular band. The graph in the center shows the utilization of various PUs with the horizontal axis representing time and the vertical axis representing frequency space. Some PUs use their spectrum for all time as represented by the red transmitter. An example of such a transmitter would be a TV station that is always broadcasting. However, if we look at the teal colored station we can see that it transmits for a time, stops, then transmits again. This gap between transmissions is referred to as a spectrum hole. CRs take advantage of these holes to operate in unused portions of the RF spectrum.

CRs operate on a principle known as the Cognitive Cycle (Fig. 2.4). The process starts by observing at the RF stimuli in the radio environment. The first step in the cycle is spectrum sensing where the radio determines which portions of spectrum are empty. This is synonymous with determining the spectrum holes from Figure 2.3 and is crucial to the functionality of a CR because it locates possible spectrum for operation. After spectrum sensing the system is aware of the location of current PUs as well as spectrum holes. The spectrum decision block aggregates all of the sensing parameters and decides where in spectrum space the radio will operate. It may take into account spectrum sensing data, databases, previous observations and even data

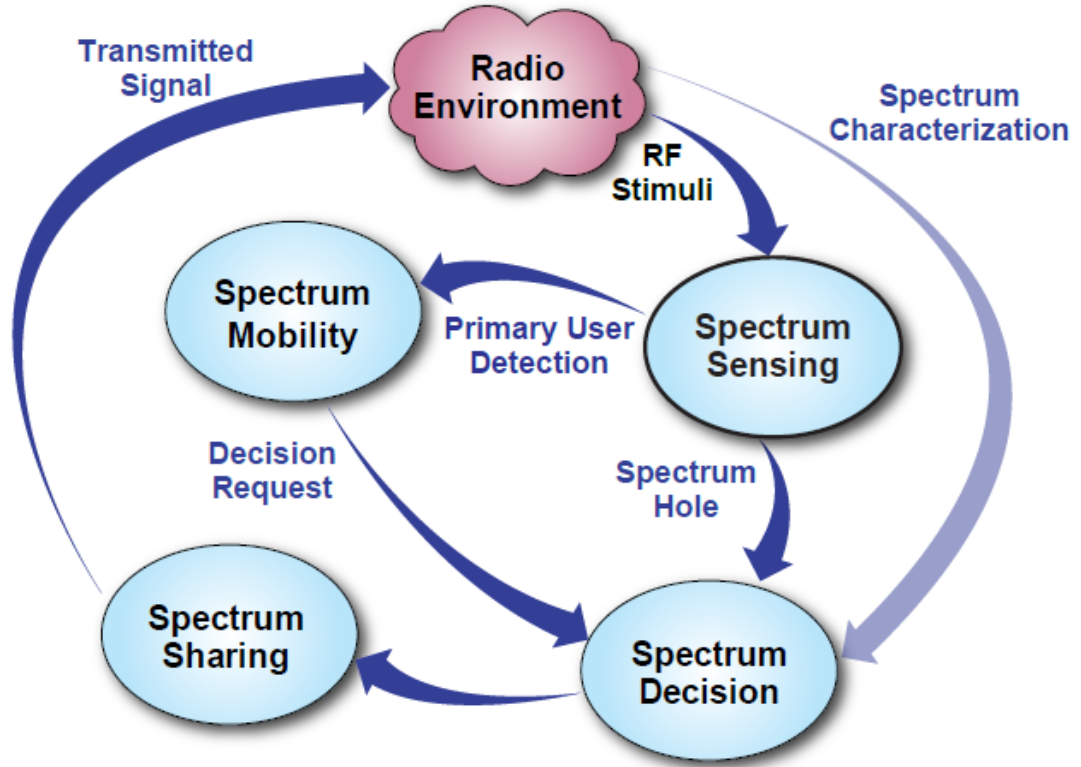


Figure 2.4: Cognitive Cycle

from other CRs. Finally, the CR transmits its signal into the radio environment and the cycle starts again.

2.3 Spectrum Sensing

As discussed in Section 2.2, spectrum sensing is part of the critical path of the cognitive radio cycle. The basic premise of the spectrum sensing algorithm is to analyze RF data and report what sections of spectrum are occupied and what portions are empty. There are different ways to implement spectrum sensing and for the purpose of this thesis we implemented the algorithm as shown in Figure 2.5. The

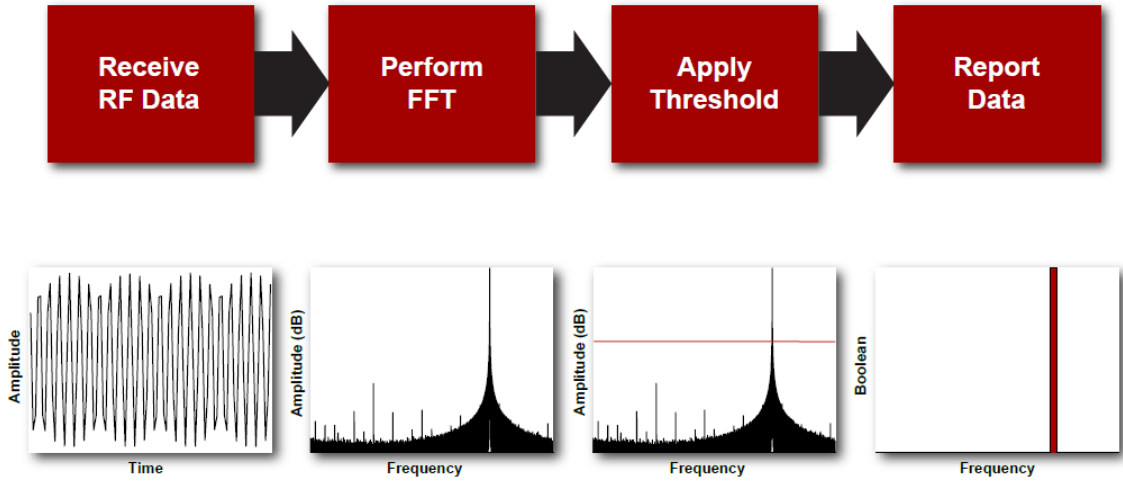


Figure 2.5: Spectrum Sensing Algorithm

first step receives the RF data and digitizes it via an ADC. The resulting data are the time domain representation of the RF input. The next step is a Fast Fourier Transform (FFT) to convert from the time domain to the frequency domain. FFTs are referred to by their point size where the point number represents how many output bins the FFT will produce. If a 256-point FFT is performed, it produces 256 bins, where each bin represents the energy present in a fraction of the total bandwidth. At this point the data are thresholded. If the value of the bin exceeds the threshold, the channel is considered occupied and assigned a value of 1. If the value is below the threshold, the channel is considered unoccupied and assigned the value 0. Once all of the FFT output values have been thresholded and assigned a value of 1 or 0, the thresholded results are aggregated and reported back to the user.

2.4 Conclusions

In this chapter we introduced the required background information for understanding this thesis. Specifically, we gave an overview of the existing SDR platforms including the USRP N210 which is used in this project. Next, we introduced the concept of Cognitive Radio and discussed how a CR can opportunistically utilize unoccupied spectrum space. Finally, we introduced spectrum sensing, the application implemented by the CRUSH platform to find free spectrum space.

Chapter 3

CRUSH Platform Overview

In this chapter we will introduce the Cognitive Radio Universal Software Hardware (CRUSH) platform. This platform serves as a high performance signal processing system allowing hardware level cognitive radio research. The CRUSH platform consists of hardware, HDL and software. In this chapter we outline the components and features of CRUSH.

3.1 Hardware Overview

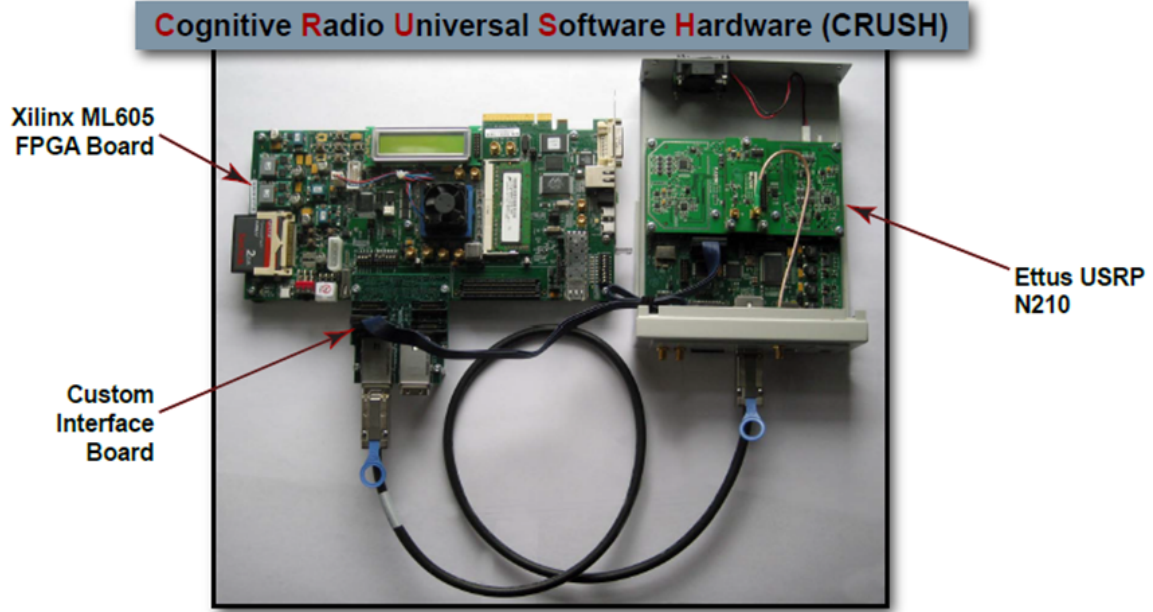


Figure 3.1: CRUSH Platform

The CRUSH system (Fig. 3.1) is comprised of three components (1) an Ettus Research USRP N210 software defined radio, (2) a Xilinx ML605 Development Board and (3) a custom interface board (CIB) to route signals between the USRP and the ML605. The interconnects between the boards are commercial off the shelf (COTS) cables.

3.1.1 Ettus Research USRP N210

The USRP N210 (Fig. 2.1) implements the front end for several radio schemes that can be implemented with a mix of hardware and host software. The device has a 100 MSPS ADC and a 400 MSPS DAC. A daughterboard implements the RF portion of the device and there are different daughterboards that can be utilized depending

on the radio requirements. The logic and control of the system is programmed on a Xilinx Spartan 3A-DSP3400 FPGA. All base system and RF daughterboard controls and data are transmitted over a gigabit Ethernet link to the host. The daughterboard used in this project, the WBX, transmits and receives anywhere between 50 MHz and 2.2 GHz with 25 MHz of instantaneous bandwidth.

The USRP is very capable but its one weakness is the FPGA. The Xilinx Spartan series is a low-end FPGA that is not designed to function at high speeds and lacks advanced FPGA features compared to the Virtex series. It has less RAM and DSP blocks; features that are essential to many DSP algorithms such as FFTs. Because of the complex design required to support USRP functionality, it is difficult for a user to make simple changes to this FPGA without first understanding the rest of the system. Additionally, minor changes can cause the design to not meet timing closure. Timing closure is an FPGA concept indicating whether the logic can operate at the clock rate desired by the user. Modification of the existing HDL is possible but the design complexity makes this difficult.

3.1.2 Xilinx ML605 Development Board

To overcome the shortcomings of the USRP we added a second FPGA board. The board, an ML605, has a Xilinx V6 LX240T FPGA which has 6.6x more Block RAM, 4.49x more LUTs and the baseline logic blocks can run ~ 2.5 x faster when compared to the FPGA on the USRP. The ML605 board serves as a blank slate for the hardware developer because it is not cluttered with an existing complex design like the USRP.

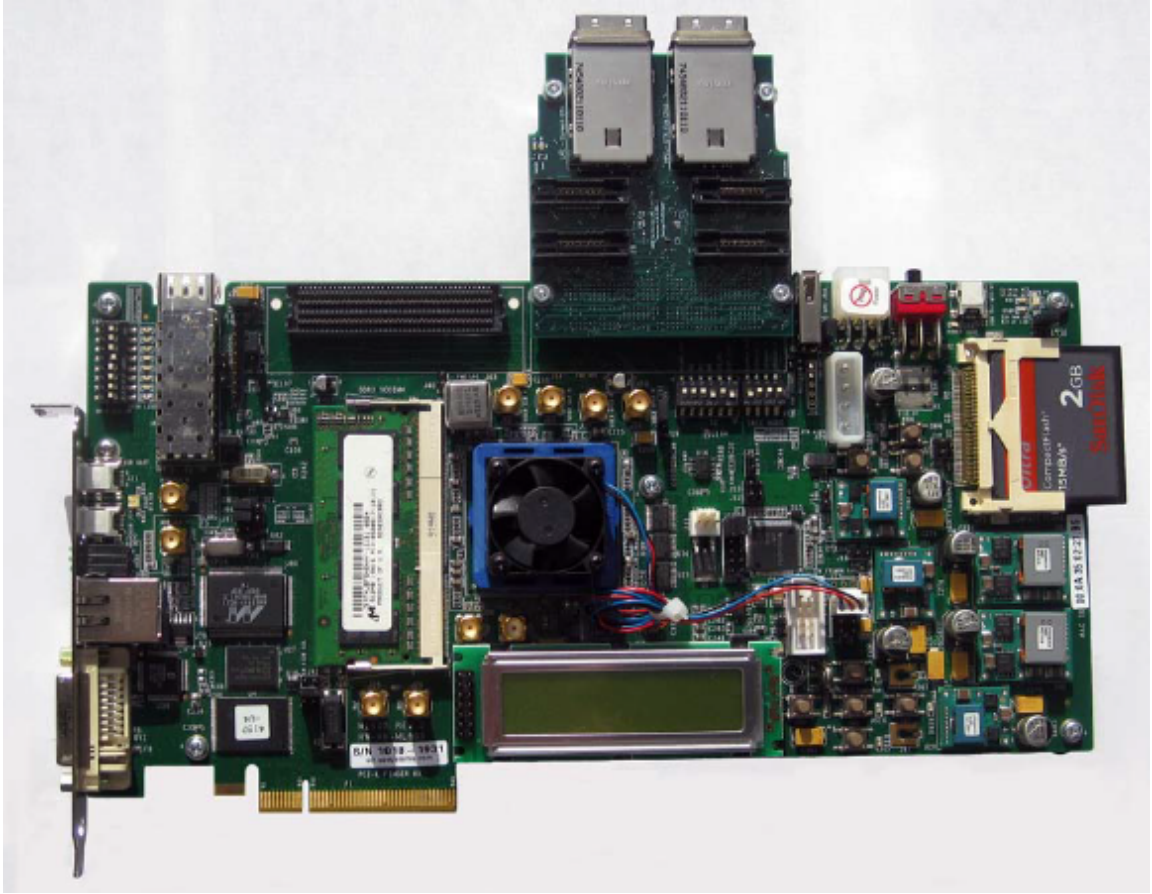


Figure 3.2: Xilinx ML605 Development Board

3.1.3 Custom Interface Board

The custom interface board (CIB) was designed as a way to connect the ML605 and the USRP platform, however it can be paired with many existing boards because it implements standard interfaces. We use a commercial off the shelf (COTS) cable designed to handle high data rates. We designed the CIB schematic using Mentor Graphics DxDesigner and routed it with Mentor Graphics Expedition. Once the board was routed, it was fabricated by Sunstone Circuits and assembled by Screaming Circuits. All parts are standard and can be purchased from an online electronic parts

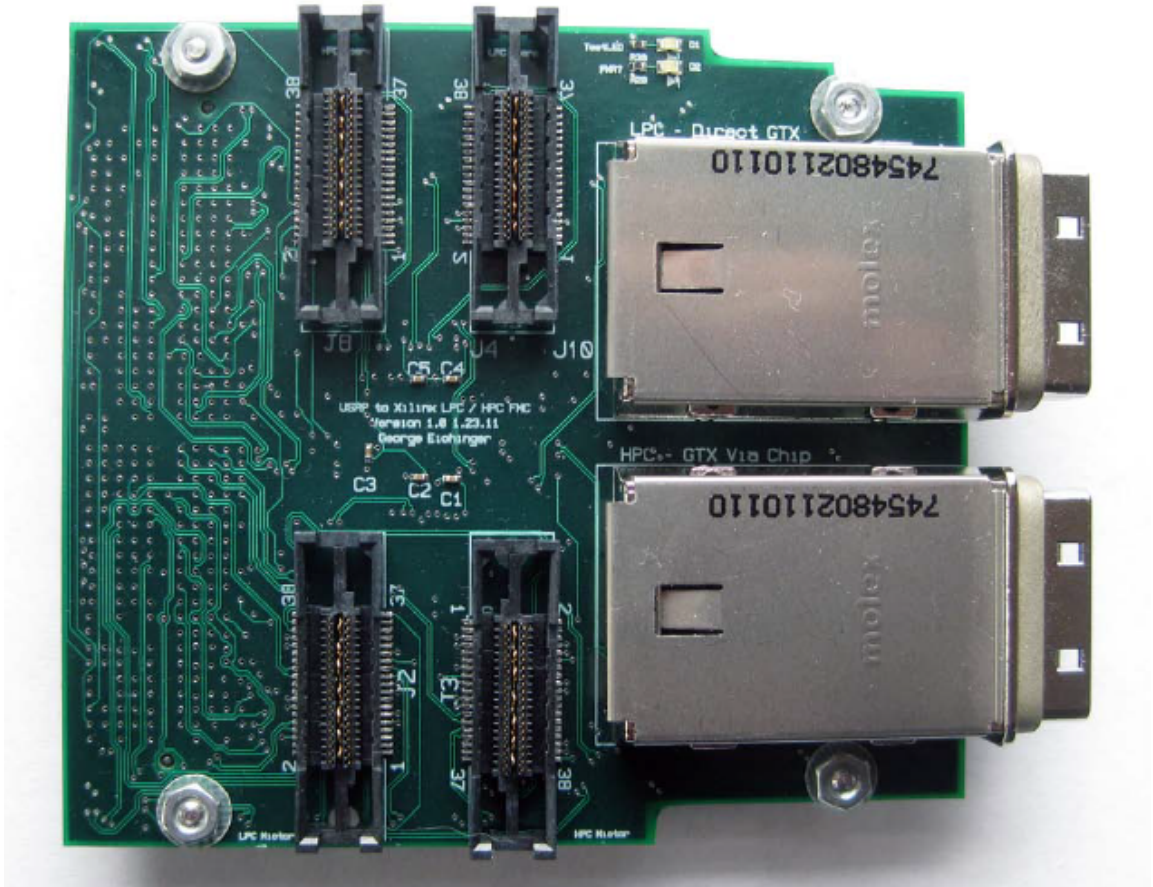


Figure 3.3: Custom Interface Board

company such as Digi-key. The final assembled board is shown in Figure 3.3. The following sections describe the design and functions of the CIB.

FMC

The CIB is an FPGA Mezzanine Card (FMC) which is quickly becoming an industry standard for custom FPGA interface boards [12]. The FMC standard allows for two different types of boards, low pin count (LPC) and high pin count (HPC). The LPC provides 160 IO to the FMC site while the HPC provides 400 IO. To remain as flexible as possible, the CIB is designed to function with either version of FMC

by implementing USRP connections on both the HPC and LPC pins. This way the board can be used with other FPGA development boards besides the ML605, such as the SP605, which is a less expensive Spartan 6 series FPGA development board containing only a single LPC FMC connector. The ML605 has two FMC sites, one LPC and one HPC. As a result, the ML605 is physically wired to hold two CIBs. With one CIB on the ML605's LPC FMC and another on its HPC FMC, one ML605 could support three simultaneous USRPs. Special care was taken to conform to the physical design specifications of FMC so the CIB could function with other board and chassis manufacturers.

MICTOR Connectors

The primary connection between the CRUSH platform and the USRP is the MICTOR connector. MICTOR is a brand of cable and connector that is designed for matched impedance, high data rate applications [13]. This variant of MICTOR has 34 data lines and 4 ground lines. The USRP's MICTOR connector was designed by Ettus as a debug port to connect a digital logic analyzer to view internal FPGA signals. Because the MICTOR cable is designed as a debugging interface, the default pinout defines two clocks with a bank of 16 data lines per clock. However, since we are using this as an FPGA to FPGA link, we can assign any signals to the 34 pins¹. The CIB has four MICTOR connectors in total; two are designed to communicate with USRPs and the remaining two break out spare FMC IO pins. Of the two USRP MICTORs, one is connected to the pins present on the LPC FMC and the second to the pins

on the HPC FMC. This allows the board to work with either LPC or HPC capable FPGA boards.

MIMO Connector

The second board to board interface supported by the USRP is the Multiple Input Multiple Output (MIMO) connector. Ettus Research intends this connector to transport I and Q data, clocks, and alignment signals to a second USRP so that they can perform MIMO operations. Much like the MICTOR cable, this interface can be used for any signal inside the FPGA and is not limited to MIMO signals. The physical interface for the MIMO connection is a commercial miniSAS port. The Spartan 3A-DSP series does not have any gigabit transceivers so in order to add a Serializer/Deserializer (SERDES) Ettus utilized a TI TLK2701 Transceiver chip [14]. This chip accepts parallel signals with a clock and creates a serial data link. On the CIB board we placed two miniSAS connectors. The Xilinx Virtex 6 series has built in gigabit transceivers. On the CIB we implemented the MIMO interface with two separate SERDES connection methods. On the LPC FMC the MIMO port is directly connected to the ML605 FPGA. On the HPC FMC we copied the exact design from the USRP schematics using the TLK2701. As of publication, the MIMO port on the CIB has not been tested but is available for future users.

¹While all 34 lines can be assigned to any IO, adjacent IO drivers on the Xilinx Spartan 3A-DSP FPGA share a clock. The USRP Debug pins share adjacent IO drivers with other functions so not all of the 34 pins can be assigned in certain clocking situations.

Impact on USRP Functionality

Minimizing the impact of CRUSH on the existing USRP functionality was a goal of this project. We designed the modified USRP HDL to function identically with or without the presence of CRUSH. To accomplish this, pullup resistors were implemented on the data lines between the ML605 and the USRP. If the ML605 is not present, all of the control lines are pulled high and the USRP functions as a standalone unit. The USRP HDL has been modified so that the data from the ADC are forked and sent over the interface between the USRP and the ML605. The original path is still intact so it can be used as a standard radio in parallel with the FPGA board.

3.2 HDL Overview

In this section we discuss the hardware description language (HDL code), the programming language of FPGAs, that was written to allow the various elements of the CRUSH platform to communicate. The next two sections describe the interfaces between the USRP, ML605 and the host.

3.2.1 USRP HDL Framework

As part of the CRUSH platform, the USRP HDL was modified to allow communication with the ML605. An overview diagram of the additional modules is shown in Figure 3.4. All of the modules for the USRP reside in a Verilog file called `u2plus_core.v`. In theory, the USRP receives raw data from the ADC and ships it to the host over



Ethernet. In practice, there are several complex DSP steps that occur inside the FPGA between the ADC and the host. The data from the ADC enters the FPGA as a signed 14 bit number consisting of 100 MSPS samples of I and Q. Inside the USRP FPGA, the raw ADC data enters the `rx_frontend` block, where a DC offset correction is performed that removes the DC offset from the analog front end and centers both I and Q around zero volts. The `rx_frontend` block also contains a magnitude and phase correlator designed to equalize the signals across multiple USRPs in a MIMO formation. After the `rx_frontend` block, the data are represented as signed 24 bit I and Q values. This module also contains calibration steps that can be dynamically set via the host for the specific RF daughterboard and the resulting data is less noisy than the original stream. Internal to the USRP, after the `rx_frontend`,

the data goes into the `dsp_core_rx` block that filters and decimates the data per the settings in software on the host. After the `dsp_core_rx` block are FIFOs that store the data before it is sent over Ethernet to the host. This signal processing chain is not modified by the CRUSH platform and functions independently from CRUSH operation.

The blue modules in Figure 3.4 all reside inside `crush.v` and represent the additions necessary for CRUSH. The first additional module is the Control Bus Slave module which receives a parallel address (ADDR) and control data (CDATA) stream over the MICTOR cable from the ML605. Table 3.1 lists all of the currently configured addresses defined for this bus. The mode word tells the MUX which signal to forward over the MICTOR cable to the ML605. There are currently 10 different options as defined in Table 3.2. Most of the modes are for debugging such as all 1's, all 0's, echo, etc. The default mode is 8 which forwards data to the ML605 after it has left the `rx_frontend` block. In addition to mode, the ML605 can send a remote frequency and an echo. The remote frequency is sent to the NCO block in Figure 3.4 and is useful for testing algorithms for accuracy. The NCO allows a user to simulate incoming tones anywhere in the 100 MHz of available bandwidth. The final feature of the control bus is an echo word. The user can set an echo word, for example 0xDEADBEEF and it will be transmitted in place of I and Q. This serves as an additional debugging feature.

Once the user has selected the mode, the corresponding I and Q values will be

Table 3.1: Control Bus Addresses

Address	Name	Description
0	Mode	Sets the mode
1	Remote_Freq1	Sets the remote frequency
2	Remote_Freq2	Required 2 words
3	Echo1	Sets the echo
4	Echo2	Echo requires 4 words
5	Echo3	...
6	Echo4	...

Table 3.2: Control Bus Modes

Mode	Name	Description
0	Raw ADC	Forwards the raw I and Q
1	NCO	Outputs user specified Sine wave
2	Timer Test	Outputs a special packet to test timing
3	Counter	Outputs a constantly running up counter
4	All 0	Outputs all 0's
5	All 1	Outputs all 1's
6	0/1	Outputs all 0's on I, all 1's on Q
7	Echo	Echos a user specified value over the bus
8	Post frontend High	I/Q from after rx_frontend, upper bits
9	Post frontend Low	I/Q from after rx_frontend, lower bits
10	Post DSP	I/Q from after dsp_rx_core

routed through the MUX and sent to the DDR Out module. This module sends the data over the MICTOR cable. The functionality of this module will be covered in the next section.

3.2.2 Interface between USRP and ML605

The physical connection between the USRP and the ML605 is the MICTOR cable, a COTS cable that was designed for high speed data transmission. With the 34 pins available on the MICTOR connector, we have to accomplish two goals on the CRUSH platform. The first is to transmit the raw ADC data from the USRP to the ML605. The second requirement is to create a control bus so that the ML605 and USRP can share configuration data. To accomplish this, we split the bus into two portions, one for the DDR I/Q data and the second for the Control Bus signals.

DDR I/Q Bus

As discussed in the last section, the user selects the mode via the control bus which tells the MUX which data path to use for I and Q. For this project we are using the output of the `rx_frontend` block. This block outputs 24 bits each of I and Q data, 48 bits total, which is more data than can be accommodated over the MICTOR's 34 wires. As a result we use the topmost 15 bits of the stream which is a safe choice because the input data from the ADC is only 14 bits. We use a DDR interface which provides double the throughput by transmitting data on both the rising and falling clock edges. This allows I and Q to be transmitted on rising and falling edges of the clock creating an effective transmission rate of 200 MHz. Note that the clock rate for this system is 100 MHz which results in a 10 ns clock period. This clock rate is the same as the ADC clock on the USRP which simplifies the design flow. The DDR is source synchronous so the USRP generates the clock and forwards it to the ML605

in parallel with the data. This half of the MICTOR cable utilizes 17 total lines, 15 for I and Q, one for clock and one that is unusable because of USRP routing issues.

Control Bus

The remaining 17 lines of the MICTOR connector are used for the control interface. With this many lines available we chose to implement a parallel interface to achieve low latency. Additionally, we wanted it to be expandable for future users of the system so we chose a master/slave parallel interface with control data, address and handshaking lines. We allocate eight bits to control data, six to address, two to handshaking and one to clock. Table 3.1 shows the current address space of the control bus, of which we are only using 7 of the total 64 available addresses. The master (ML605) provides a 100 MHz clock which is forwarded to the slave (USRP). On the USRP this clock feeds into a Digital Clock Manager (DCM) to create a source synchronous clock for sending data back to the ML605. A diagram showing a transfer on the bus is shown in Figure 3.5. Data transmission begins when the master (ML605) asserts a '0' on the REQ line. In response, the slave (USRP) asserts a '0' on ACK line. Following this the master retrieves the data it is going to send from a FIFO and then sends the data to the slave. The master then sets its REQ flag high and waits for the slave to assert its ACK flag to '1' at which point the process is ready to start over. This is a common four phase handshake protocol. Although it is shown here for a one directional link, the protocol was designed to support bidirectional communication and the Addr and Data buses are capable of working

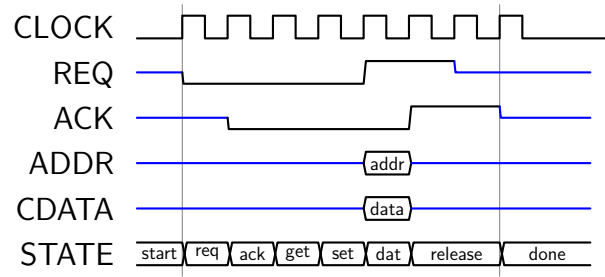


Figure 3.5: Relative Timing Diagram for one Control Bus Transfer

in both directions. All of the internal clocking and FIFOs have been designed to support bidirectional operation.

3.2.3 ML605 HDL Framework

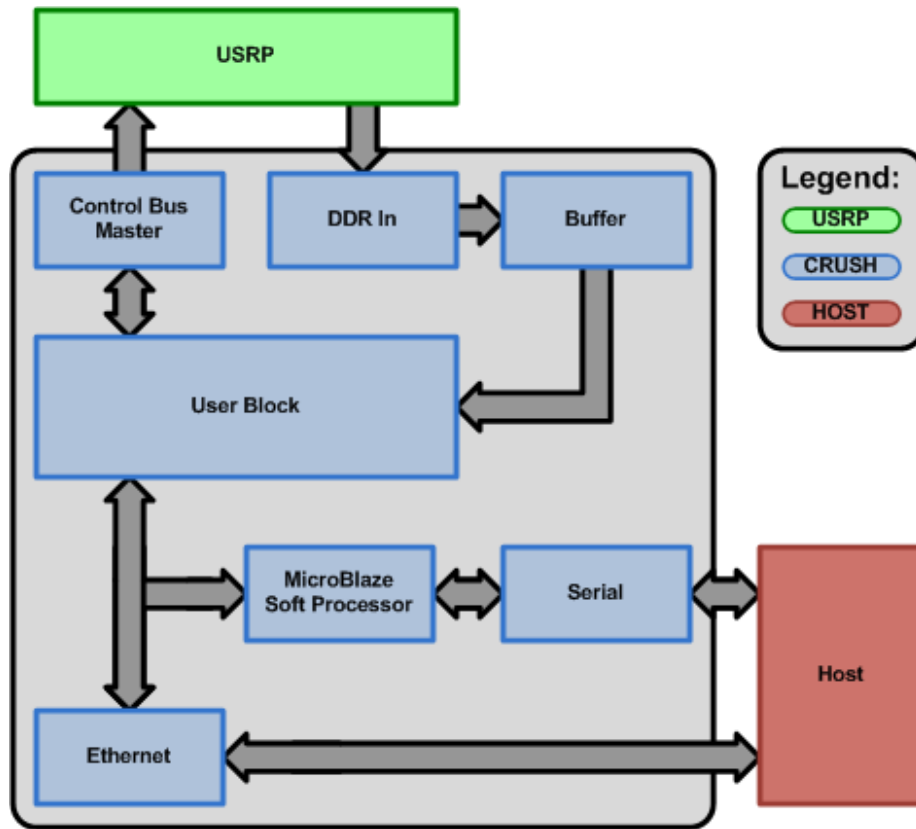


Figure 3.6: ML605 HDL Framework

The bulk of the CRUSH development lies in the ML605 HDL Framework (Fig. 3.6). The core of the ML605 HDL Framework is the User Block where the CR or SDR application resides. In addition to the external interfaces, the top module contains the clocks, resets and glue logic. The goal of this design is to make it as easy as possible for a new user to insert their code into the project. A program such as MATLAB HDL coder could be used to synthesize a module and insert it into the User Block. The current framework uses 3 % of the FPGA and leaves 97 % free for the User Block. The ML605 has three different external interfaces, the MICTOR connector to the USRP, an Ethernet connection to the host and a serial connection to the host.

The MICTOR connection serves two purposes. First, it transmits data over a control bus to the USRP. Second, it receives 100 MSPS I and Q samples over a 200 MHz DDR interface. The ML605 is the control bus master and provides the clock, address and control data signals. The functionality of this interface is the same as described in Section 3.2.2 except the ML605 is the master. All commands transmitted over the control bus originate from the host via either the Ethernet or Serial port. The DDR signals from the MICTOR connector are demuxed into 15 bits each of I and Q in a special block on the ML605. At this point, the data are still synchronized to the USRP clock domain so an additional block puts the signals through a dual clock FIFO to synchronize with the ML605 clock. Now I and Q are routed to the User Block where a CR or SDR algorithm can be implemented on CRUSH.

The Ethernet interface is based on the Xilinx Embedded Trimode Ethernet Media Access Control or TEMAC [15]. This is a module provided by Xilinx for use with their Virtex 6 line of FPGAs and it provides a transmit and receive FIFO controlled by start and end frame flags. The user provides a frame of data including properly formed headers and the TEMAC module transmits the data over Ethernet. Receiving operates on a similar principle where data is constantly streaming into the FPGA with start and stop flags indicating the beginning and end of a frame. In order to use the TEMAC core, we wrapped it with glue logic that would feed frames to the transmit module and screen the headers of the receiver module for our packets. For the packets, CRUSH uses UDP, User Datagram Protocol, as the method of transport because of its low overhead and ease of use. During typical operation, the host sends the ML605 a UDP packet that contains the required configuration bits and operation mode of the CRUSH system. The ML605 receives this packet and dissects it in a state machine. Depending on the mode indicated by the packet, the ML605 performs the required action. If the host requests a spectrum sensing session, the ML605 will perform the analysis and transmit the result back to the host. Throughout this process great care has been taken to reduce the latency added by the ML605. Ethernet packets are complex to form because they require a MAC header, an IP header and a UDP header in addition to the actual data [16]. The headers require information such as header length, packet length, source MAC address, destination MAC address and checksum. To form these headers each time a packet is sent

would be prohibitively time consuming. Instead, an embedded Microblaze processor calculates the header once and stores it to a dual ported Block RAM. When a packet is ready for transmission by the ML605, it reads the header out of the Block RAM and into the transmit FIFO. Once the header is loaded into the FIFO, a state machine starts loading the payload. Our algorithm stores all of the possible payload lengths and corresponding checksum values so that no matter the packet length, it can load the proper checksum without recalculating. The header is calculated in software on the Microblaze so that it be easily modified. This is preferable over recompiling the entire FPGA design.

The serial port connection performs the same functions as the Ethernet interface but via an interactive console application rather than a packet based byte interface. The serial port connection interfaces directly to the embedded Microblaze processor and can also be reprogrammed without changing the FPGA configuration.

3.2.4 Interface between ML605 and Host

The primary interface between the ML605 and the host is UDP over Gigabit Ethernet. UDP is a low overhead packet type which is easy to implement on the host side. The first step on the host when communicating with CRUSH is to form a packet of the proper structure (Table 3.3). When forming this packet, the general information section and the channel 1 section are required. Channels 2 and 3 are optional and are not currently implemented in CRUSH. In order to allow for easy expansion, we added placeholders for these future channels so the hardware and software would not

need to be rewritten.

Table 3.3: Host to ML605 Packet Structure

Name	Length	Description
General Information [Required]		
identWord	4	Packet identification word
numChannels	1	Number of channels this packet contains [1-4]
spare	3	spare
Channel 1 Information [Required]		
Threshold	4	Spectrum sensing threshold value $[0, 2^{32} - 1]$
fftSize	1	Size of the FFT in log; 2^x where x is fftSize [3,15]
Mode	1	Sets the mode of this channel
frequency	4	USRP Frequency [optional]
spare	2	Spare
Channel 2 Information [Optional]		
Threshold	4	Spectrum sensing threshold value $[0, 2^{32} - 1]$
fftSize	1	Size of the FFT in log; 2^x where x is fftSize [3,15]
Mode	1	Sets the mode of this channel
frequency	4	USRP Frequency [optional]
spare	2	Spare
Channel 3 Information [Optional]		
Threshold	4	Spectrum sensing threshold value $[0, 2^{32} - 1]$
fftSize	1	Size of the FFT in log; 2^x where x is fftSize [3,15]
Mode	1	Sets the mode of this channel
frequency	4	USRP Frequency [optional]
spare	2	Spare

The first section of the packet contains general information and tells the ML605 how many channels the packet contains. At this time the CRUSH HDL is only configured for one channel but we built expandability for more into the packet structure. The second section is the channel specific configuration. This has been setup for our specific spectrum sensing application. The first word in this section is the user

supplied threshold. Next is the size of the FFT, followed by the mode. A list of possible modes are shown in Table 3.4. Only modes 2, 4, and 5 are implemented, the rest of the modes are placeholders for future functionality. The description of these modes will follow in a later chapter after we have introduced the spectrum sensing algorithm. These are the only fields required at this time. Spare bytes have been added to each channel's allocation to allow for additional variables to be added to the system.

Table 3.4: Modes for ML605 Packet

Value	Impl?	Name	Description
0	no	status	ML605 responds with current settings
1	no	set	Just set ML605 settings
2	yes	man	Run FFT, threshold and report results
3	no	auto	Run FFT using autothreshold algorithm, report results
4	yes	raw	Run FFT, report FFT values
5	yes	raw+man	Run FFT, threshold, report FFT and threshold results

3.3 Software Overview

The third and final part of the CRUSH platform is the software that interconnects the hardware and HDL. Software is present in two different locations in the platform, the host and the Microblaze embedded microprocessor on the ML605.

3.3.1 Host Software

We implemented three different host software platforms: (1) a custom MATLAB GUI demo, an interactive serial port command line interface, and a speed optimized

interface in C++. Each of these interfaces accomplishes a specific goal and are useful for the platform.

MATLAB

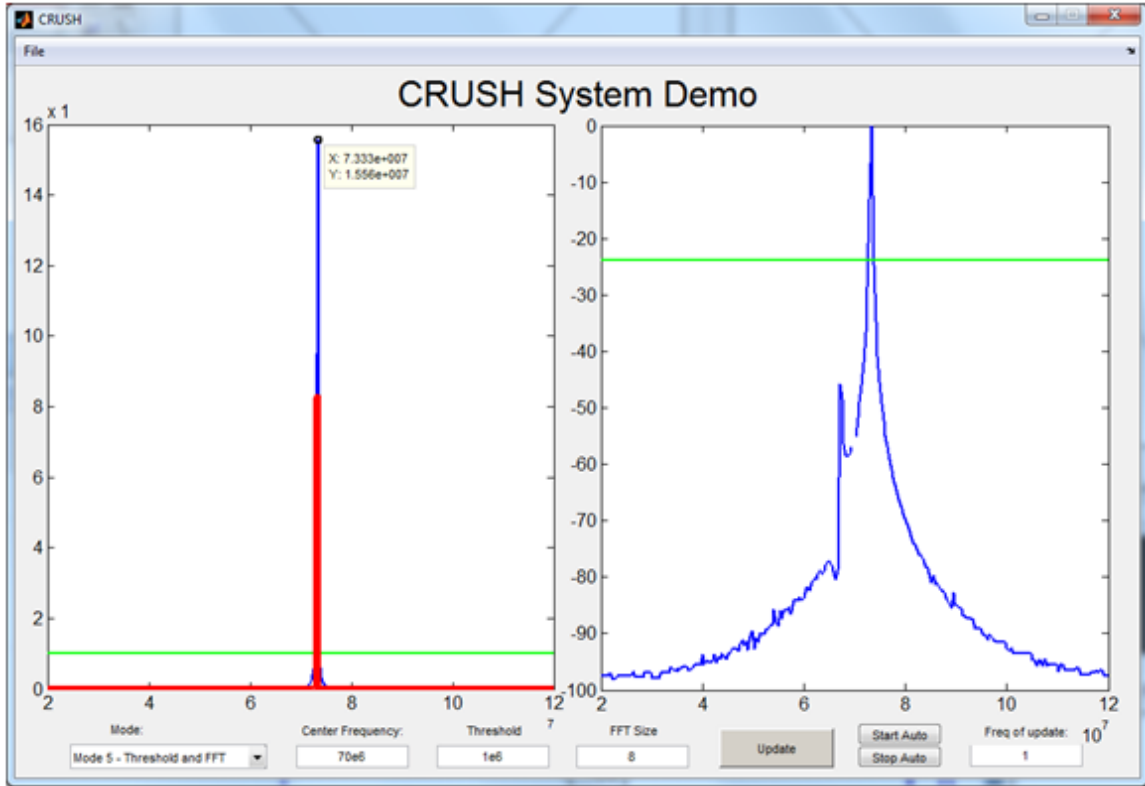


Figure 3.7: CRUSH Demo: 70 MHz Center, 73 MHz CW tone, 256 point FFT

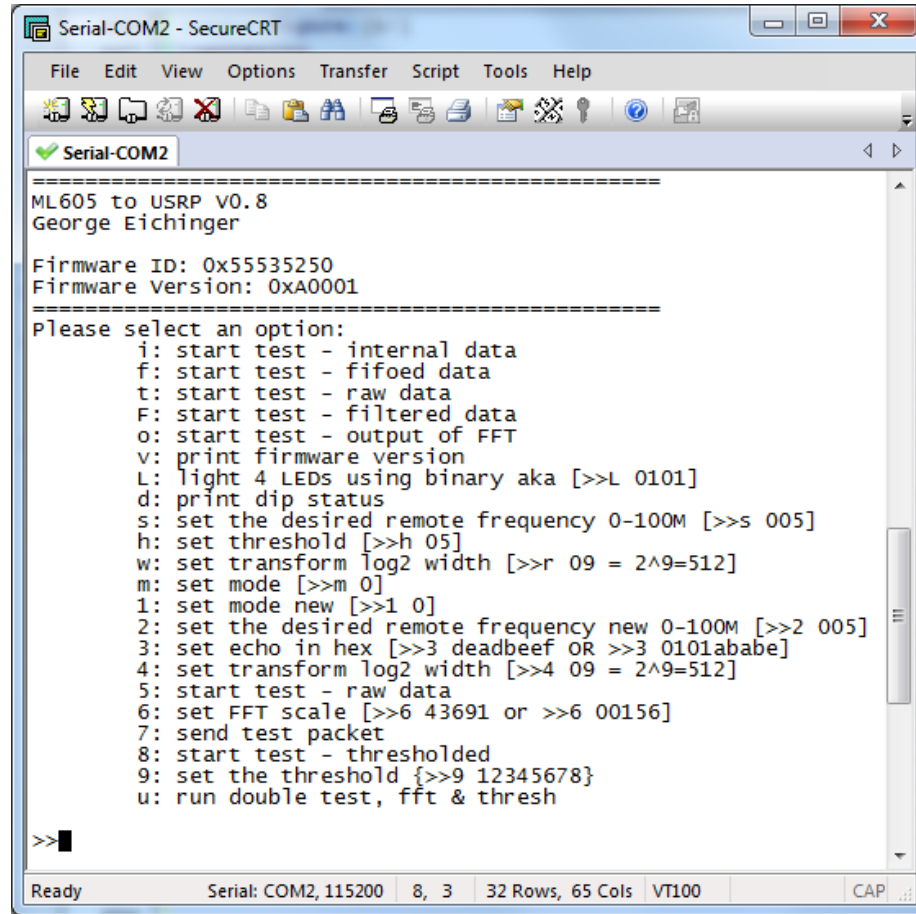
On the host, we have developed a simple GUI demo using The Mathworks GUIDE to show the functionality of the CRUSH system. A picture of the GUI is shown in Figure 3.7. The demo consists of the same data plotted on both linear and log scales. The graph on the left shows a magnitude plot with the x axis representing 100 MHz of frequency and the y axis representing the magnitude of the FFT bin. The green line shows the current threshold value. The blue plot is the output of the FFT

from the ML605 and in this plot we were injecting a 73 MHz CW tone. The red plot is a binary representation of the thresholding results from the spectrum sensing algorithm. The plot on the right uses a log scale for the y axis plotted in decibels relative to the carrier (dBc) where the peak has been set to 0 dB and the rest of the values are relative to this point. Again, the green line is shown to indicate the threshold value. The bottom of the GUI has text boxes where all of the parameters of the CRUSH system can be set. The user has the option to run the demo manually with a button push or to update the plots automatically.

The backend of the GUI contains a MATLAB function that processes the parameters on the bottom of the GUI and creates a packet as defined in Table 3.3. It then opens a UDP port to CRUSH and sends the packet. Finally, it waits until it receives the requested data and then creates the plots in Figure 3.7.

Serial Port

Although not strictly software, the second host interface is the serial port connection. This serves as a debug port and allows a user to type commands instead of forming UDP packets. It is useful for testing basic functionality and follows a standard command line interface. All of the functions available over the Ethernet UDP interface are available over the serial port. There are also additional debugging options that are not available elsewhere. For requests over the debug port that generate data, it will print the data values to the screen in hexadecimal format. We have written a separate Matlab program that processes the values copied from the screen and



```

Serial-COM2 - SecureCRT
File Edit View Options Transfer Script Tools Help
Serial-COM2
=====
ML605 to USRP V0.8
George Eichinger
Firmware ID: 0x55535250
Firmware Version: 0xA0001
=====
Please select an option:
i: start test - internal data
f: start test - fifoed data
t: start test - raw data
F: start test - filtered data
o: start test - output of FFT
v: print firmware version
L: light 4 LEDs using binary aka [>>L 0101]
d: print dip status
s: set the desired remote frequency 0-100M [>>s 005]
h: set threshold [>>h 05]
w: set transform log2 width [>>r 09 = 2^9=512]
m: set mode [>>m 0]
1: set mode new [>>1 0]
2: set the desired remote frequency new 0-100M [>>2 005]
3: set echo in hex [>>3 deadbeef OR >>3 0101ababe]
4: set transform log2 width [>>4 09 = 2^9=512]
5: start test - raw data
6: set FFT scale [>>6 43691 or >>6 00156]
7: send test packet
8: start test - thresholded
9: set the threshold [>>9 12345678]
u: run double test, fft & thresh

>>
Ready Serial: COM2, 115200 8, 3 32 Rows, 65 Cols VT100 CAP

```

Figure 3.8: CRUSH Serial Interface

generates plots. The serial port is controlled on the FPGA by the Microblaze. This yields an advantage over the Ethernet interface because the Microblaze software is coded in C and can be modified without re-synthesizing the entire design.

UHD

The USRP Hardware Driver (UHD) is the last and highest speed interface designed for CRUSH. The code is written in C++ and is based on the UHD code from Ettus Research for communicating with the USRP over Ethernet. The code is based on

the `rx_samples_to_file.c` example that comes with a standard distribution of the UHD software. This example was picked as the basis for our interface because it can control both the USRP and CRUSH at the same time. UHD is based on the Boost libraries [9] so we used the UDP primitives native to Boost to implement our Ethernet interface. Using this program allows for fair speed comparison tests between CRUSH and the USRP since they are running identical versions of the Boost library.

3.3.2 ML605 Software

The only C code on the ML605 runs on the Xilinx Microblaze embedded processor. This onboard device serves two purposes within the CRUSH framework. First, it implements the header for the Ethernet packet. This is done in software so that it can easily be changed without reprogramming the entire FPGA. If something like a MAC address or the number of bytes in the packet changes, the C code can be easily modified. The second purpose of the Microblaze software is to implement the serial command line interface shown in Figure 3.8.

3.4 Conclusions

In this chapter we described the CRUSH platform in terms of the hardware, HDL and software. The CRUSH hardware is based on the Ettus Research USRP N210, the Xilinx ML605 FPGA Development board and the custom interface board. The CIB is the major hardware contribution of this thesis because it allows CRUSH to combine the powerful FPGA backend of the ML605 with the versatile RF front end of the

USRP. HDL was written for both the USRP and the ML605. For the USRP, a small number of non-intrusive HDL modules were added to allow communication with the ML605. The major contribution in this area was the CRUSH HDL Framework on the ML605. This allows CRUSH to communicate with the USRP as well as the host. Additionally, it creates the User Block which allows for algorithms to process RF data without significant effort by the developer. Several different software contributions were made including Matlab code, C++ code and a serial debug interface. All three of these software modules contribute different features to CRUSH. The Matlab code serves as a visual demo to demonstrate the system and verify functionality. The C++ code allows for high speed testing and would be the preferred implementation method for using CRUSH in a complete CR system. Finally the serial debug module allows developers to quickly asses what is happening in the platform and modify parameters without resynthesizing the FPGA design. One of the main contribution of the CRUSH platform is the decoupling of the high performance FPGA hardware from the RF front end processing. This, combined with the rest of the platform development, creates a versatile, upgradeable platform for Cognitive Radio and SDR Research.

Chapter 4

Spectrum Sensing

The previous chapter introduced the CRUSH platform and its hardware, software and HDL. Spectrum sensing is the algorithm that we implemented on CRUSH to accelerate Cognitive Radio functionality. This chapter first describes a software only version of spectrum sensing using C++ and the UHD framework, followed by the details of the hardware implementation of the same algorithm on CRUSH.

4.1 Related Work in Software Spectrum Sensing

Spectrum sensing is not a new concept and has been implemented with a variety of different algorithms and platforms as described in the literature. In presenting the prior work on spectrum sensing we reference some general surveys followed by a division of existing work into several broad categories. First we will discuss algorithms created for software. In these algorithms, data are obtained from some source, e.g. the USRP, and analyzed in either MATLAB or a custom software program. These generally contain new and unique spectrum sensing ideas but are limited to the research setting and have not progressed to implementation on a real-time SDR.

4.1.1 Spectrum Sensing Surveys

There are several different surveys on spectrum sensing. One such survey is from Yücek and Arslan (2007) where they overview a broad category of sensing algorithms and look at spectrum sensing from the top down [3]. This paper also discusses current wireless standards and how they deal with the spectrum sensing issue such as how Bluetooth uses sensing to determine other industrial, scientific and medical (ISM) band transmitters to avoid. Akyildiz et al. presents a survey of available spectrum sensing techniques and further breaks down the problem into three categories: (1) detecting the transmitter, (2) detecting the receiver, and (3) interference temperature management [2]. The authors identify many of the existing spectrum sensing techniques including energy detection, feature detection and the matched filter technique.

CRUSH utilizes the energy detection technique combining FFTs and thresholds.

4.1.2 Spectrum Sensing in Software

Software techniques for spectrum sensing show diverse and innovative solutions. The long term goal is that these techniques could one day be implemented in hardware but most of these algorithms have not been attempted on an FPGA. MacKenzie et al. discuss their approach to spectrum sensing in their survey on CR research performed at Virginia Tech [4]. In this paper they advocate a distributed cyclic feature-based automatic modulation classification (AMC) where each radio extracts features from the spectrum and all the radio observations are fused together. Extensive work and research has been put into cyclostationary detection by Haykin et al. [17]. In this work they detail how this method can be used both theoretically and experimentally to extract features from the data to recognize primary users (PUs).

4.2 Spectrum Sensing Algorithm in Software

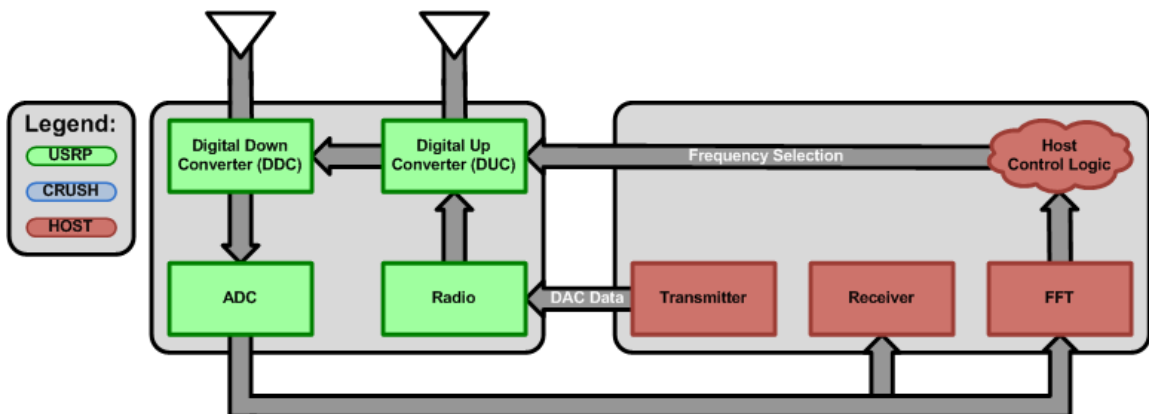


Figure 4.1: Spectrum Sensing Algorithm in Software

In a software implementation, the USRP downconverts, digitizes and transports the RF data to the host where it is received inside a UHD program. Once inside the computer, the data is processed by an FFT and the output values are compared to a threshold (Fig. 4.1). The results of this computation are reported to the cognitive radio system as part of the cognitive radio cycle (Fig. 2.4). This method is slow and does not allow for real-time operation. In this sense real-time refers to algorithms on the μs scale. The FFT is performed via the highly optimized package FFTW [18]. However, it is not the speed of the software on the host that is the issue, rather the fact that the data needs to be forwarded over an Ethernet interface in order to be analyzed. The packet and buffer overhead for transmitting the raw RF data over Ethernet is high. Additionally, the data, composed of 16 bits of I and Q, are sent over the link at 25 MSPS. This results in 100 Mega Bytes per second (MBps) which is very close to the peak theoretical value of GbE at 125 MBps. A faster way to do this is to move the signal processing closer to the source of the data and perform this analysis in hardware. The next section describes implementing this same algorithm on CRUSH.

4.3 Related Work on Hardware Spectrum Sensing

Spectrum sensing has also been implemented in hardware on various SDR platforms such as those discussed in Section 2.1. Some spectrum sensing algorithms are implemented in real-time and others are implemented simply as a proof of concept.

The final category of spectrum sensing research involves building a custom hardware platform. These are algorithms that either are implemented in an ASIC or other custom FPGA board that is not commercially available.

4.3.1 Spectrum Sensing on COTS Hardware

Spectrum sensing has been implemented in hardware before and one example is an energy detection algorithm with double thresholding on the WARP platform by Hänninen [19]. This work is similar to CRUSH, however their algorithm is not parameterized and has a fixed FFT size. As a result, they could only process data if the input matched a specific form.

4.3.2 Spectrum Sensing on Custom Hardware

There is a section of work that focuses on implementing spectrum sensing in hardware. If CR enters the mainstream consumer market, there will likely be application-specific integrated circuits (ASICs) performing spectrum sensing. One of the most likely areas of spectrum to see hardware CR is in white space (allocated but unused frequencies for broadcast services). As a result of an FCC ruling, users are allowed to utilize unoccupied TV channels [20]. One example of spectrum sensing in hardware is Rahman et al. who built a hardware prototype to operate in white space in Singapore [21]. In this paper the authors describe a technique called correlation based method (CBM) where they store time domain samples of recorded TV signals and correlate the incoming RF signal to check channel occupancy. This is a technique

specific to TV transmissions that would not transfer to a more general Cognitive Radio.

There is one similar set of work to CRUSH by Shishkin et al. where they create an adapter for existing RF boards designed for WARP to interface with an ML605 [22]. In this work they utilize commercial RF boards from either WARP or a COTS FMC board from 4DSP. Unlike CRUSH, they are putting the entire SDR HDL on the ML605. The advantage to CRUSH is that the ML605 is almost entirely empty with a simple Verilog or VHDL User Block for the algorithm. In Shishkin's work, they are aiming to create a new SDR platform from scratch utilizing existing RF downconverter boards. At the time of publication their work is in the advanced planning stages with hardware currently being built.

4.4 Spectrum Sensing on CRUSH

In the CRUSH version of spectrum sensing the ML605 serves as an accelerator. A block diagram of the system is shown in Figure 4.2. The data from the USRP ADC are forked and sent to both the ML605 and the host; the rest of the software defined radio functionality can continue in parallel with the CRUSH platform. On the CRUSH platform an FFT is applied to the data and the results are thresholded and then reported back to the host over Ethernet. A more detailed description of the functionality is below.

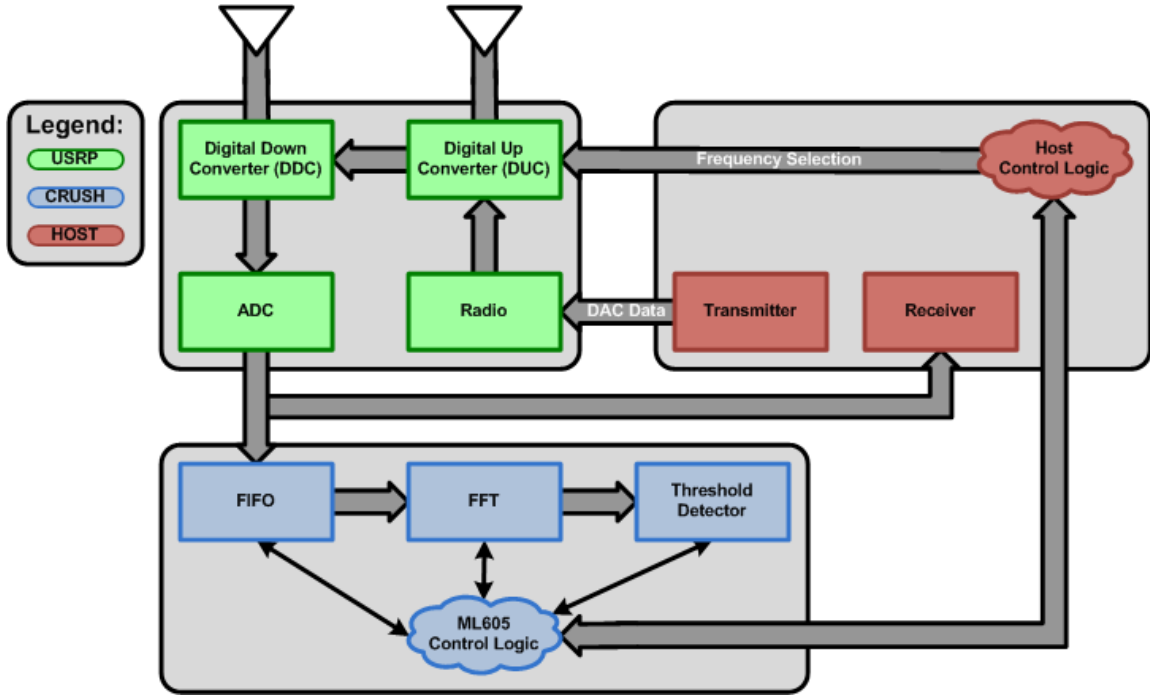


Figure 4.2: Spectrum Sensing Algorithm on CRUSH

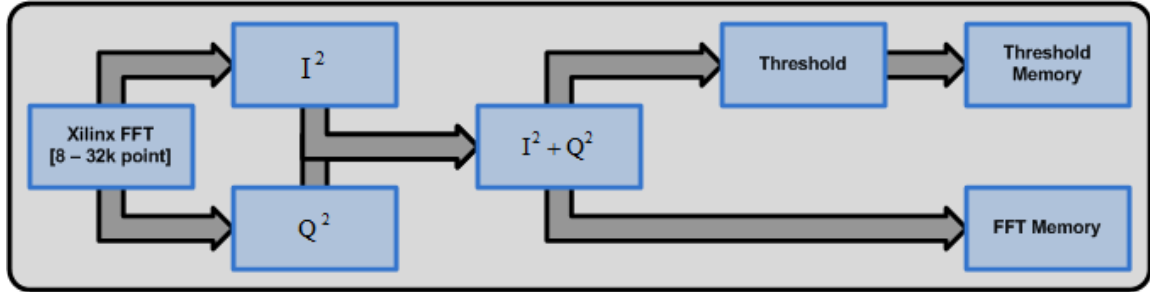


Figure 4.3: User Block for Spectrum Sensing Algorithm on CRUSH

4.4.1 FFT

Section 3.2 discussed the HDL and how the User Block is the location where algorithms can be placed with minimal effort. Figure 4.3 shows the details of the User Block for the spectrum sensing algorithm. The data entering the User Block has already gone through a clock crossing FIFO to align itself to the FPGA clock domain.

After the buffering stage, the I and Q data are fed into the User Block. The first step of the spectrum sensing algorithm is the Xilinx Streaming FFT. With this FFT, samples are constantly entering and exiting at a rate of one to one; we can take one I and Q sample every clock cycle and output one result every clock cycle. In contrast, most software FFTs input all of the data, process it, and then output the results many clocks later. The Xilinx FFT core has a runtime configurable FFT size and our spectrum sensing algorithm supports point sizes between 8 and 32768. The core supports up to 64k point sizes but this is very large and takes up almost all of the Block RAM on the device. We expect the required point size to be between 64 and 256 points.

4.4.2 I and Q Magnitude

The streaming FFT produces both I and Q values which are both signed. These data are converted to unsigned squared magnitude value by taking $I^2 + Q^2$. Unsigned squared magnitude is much easier to threshold than signed I and Q because there is no possibility of a negative value. Additionally, the sign bit is removed which saves space. Typically one would use magnitude which is $\sqrt{I^2 + Q^2}$. However, the square root operation is expensive on the FPGA, both in terms of the number of gates and latency to perform the operation. It is not a problem to make this simplification because we can just move this calculation to the host and take the original threshold and square it before passing it to the ML605. The output of the IQMagnitude signal is forked and sent to two different places. First, we store these values in an indexed

Block RAM where they can be recalled. If the mode is set to raw in the UDP packet (Table 3.4), it will read the FFT values out of this Block RAM and report them to the user. It should be noted that this may require more than one UDP packet depending on the FFT size.

4.4.3 Thresholding

For thresholding, the IQMagnitude signal is compared to a threshold value and each bin is determined to either be above a threshold, 1, or below a threshold, 0. These data are then combined into 32 bit words and stored in a second Block RAM. If you set the mode to man, for manual thresholding, in the UDP packet (see Table 3.4) it will read the values out of this memory and send them over Ethernet to the user. These packets are much smaller than the raw packets because of the data compression when reducing a 32 bit value to a 1 or a 0. Additionally, we specifically designed the Ethernet controller to output variable sized packets so that a 256 point FFT threshold packet will be drastically smaller (faster) than the packet for a 32k point FFT.

4.4.4 Fixed point Conversion

One of the hardest parts of converting an algorithm from a software to hardware is converting from floating point to fixed point. By default, the UHD framework performs all math in floating point. For this algorithm, we had two main design parameters. The input to the system is 15 bit signed I and Q values. The threshold

is a 32 bit unsigned integer and we would like our output FFT values to fit within 32 bits. Other fixed operations within the system are the adder and the squarer. The adder results in a one bit growth and the square module doubles the number of bits less one for the lack of a sign bit. It would be exactly double the number of bits but after the square operation the data are unsigned. Based on these constraints we chose to sign extend the input by one and have the FFT inputs be signed 16 bit integers. We then used a scaled FFT so that the output is also 16 bit integers of I and Q. These values get squared to 30 bits. Then when they are added we have 31 bits which is within our 32 bit limit. For the FFT, since it is 16 bits in and 16 bits out, there will be rounding and scaling occurring. For the streaming FFT the core utilizes a Radix-2 butterfly arrangement with a possibility of scaling after each Radix block [23]. We have chosen a conservative scaling schedule that will not overflow no matter the input. This is sufficient because we are looking for relatively large signals in this analysis so we are biasing our rounding to cut the low end. If a user of this platform has a priori knowledge of their input signal, this scaling schedule could be modified to be more aggressive. For example, if the user's average signal is -50 dBm and there is a limiter on the input set to -30, then the scaling could be adjusted so that signals near the -30 dBm range output maximum FFT bin sizes.

4.4.5 Verification

An important part of the design of any system is the ability to verify that an algorithm is functioning properly. In order to make this easier, when a raw+man packet is

requested, CRUSH will store both the raw and thresholded values in Block RAM and send them over Ethernet. The thresholded values will be from the same FFT values as the ones sent over Ethernet. Therefore, one can verify the functionality of the system by comparing the raw and thresholded data through post-processing.

4.4.6 Configurability

All of the variables in the spectrum sensing block are configurable including the FFT size and the threshold. Additionally, using the debug serial interface other variables can be modified such as the scaling factors for the FFT. Also, hooks have been placed into the HDL for additional features such as more than one USRP connected to the CRUSH system and more advanced algorithms that incorporate automatic thresholding. These will be discussed more in Chapter 6 under future work.

When integrated into the full system, the host computer sets all of the desired parameters before starting a spectrum sensing run. It commands the USRP to a specific frequency. It then tells the ML605 the threshold value for that frequency range. The ML605 immediately starts performing the spectrum sensing algorithm with the parameters given. Once done, it responds to the host with one or more bytes with each bit representing whether a PU was found in that region. The host can then change the frequency to perform more sensing or move on to another portion of the cognitive cycle.

4.5 Conclusions

While much work has been published on spectrum sensing, the main focus has been from the top down, focusing on theoretical methods and models for sensing spectrum. When algorithms are implemented, they are normally in software and using recorded data. The CRUSH platform was developed to create real-time algorithms that can be implemented in an online CR. We have focused on speed and parameterization to make this possible. It is the goal of this project that CRUSH be used as a platform to quickly test out other spectrum sensing algorithms while trying to keep the processing chain fast and efficient.

Also in this chapter we introduced the specifics of the spectrum sensing algorithm implemented on CRUSH. We described both the software implementation that we use as a timing reference and the HDL version that we created specifically for CRUSH. The major contribution is the creation of HDL to perform an FFT, threshold, and report the unoccupied portions of the RF spectrum to the user. The HDL has been designed to be as versatile as possible with its parameters fully configurable via its interfaces with the host.

Chapter 5

Experimental Setup and Results

The previous chapters introduces the CRUSH platform and describes the spectrum sensing algorithm. This chapter details the results of the tests we performed on the spectrum sensing algorithm. The first section details the functional verification where we prove that the link between the USRP and the ML605 is functional and that the device performs as expected. The second section proves that performing spectrum sensing on an FPGA is faster than performing the analysis in software. Finally, we show a round trip timing comparison of the spectrum sensing application between a software approach and CRUSH.

5.1 Functional Verification

To verify the functionality of CRUSH we tested over a range of frequencies and power levels. In this section we are reporting on one example frequency where we injected a signal into the USRP at 73 MHz with an amplitude of -16 dBm (decibels referenced to one milliwatt). This frequency was chosen because it is within the 25 MHz bandwidth of the USRP and within the 80 MHz range of the waveform generator used for testing. Additionally, -16 dBm is 100 mV_{p-p} which is the standard output for the device. In order to look at the functionality of the system, we wanted to look at the data in two different forms. The first is directly recorded by the USRP using the standard software included with the device. We will compare this with the output of the FFT inside the User Block.

5.1.1 USRP Master Reference

To look at the data flowing through the standard USRP path, we used the program `rx_samples_to_file` which comes with the UHD software. We configured it to record 10,000 samples to a file at a rate of 25 MHz and a center frequency of 70 MHz. In this mode, the USRP will provide signed 16 bit I and Q data consisting of 25 MHz RF bandwidth centered around 70 MHz. After the data was recorded, we plotted it in MATLAB as shown in Figure 5.1. We performed a 256 point FFT in order to compare the results to CRUSH. As shown in Figure 5.1, there is a peak at 73 MHz. The y-axis of this graph is presented in dBc. DBc stands for decibels relative to the

carrier and it means that the carrier signal was set to 0 dBc and the rest of the graph is in decibels relative to that measurement. This allows us to compare relative results even if the powers represented are not identical. This measurement will serve as our gold standard for this test. This is what a current software only radio would see and we will compare this to CRUSH.

Note that a peak is visible in Figures 5.1 and 5.2 at 67 MHz. This is the result of an I and Q mismatch on the analog front end of the USRP that creates a mirror image of the input around the center frequency. In this particular example the 73 MHz input signal is mirrored around 70 MHz and appears as an artifact at 67 MHz. The peak is less prominent in Figure 5.1 because the USRP framework decimates and filters the original data stream and only looks at 25 MHz whereas CRUSH views the stream closer to the ADC before digital filtering and sees 100 MHz. Ettus Research is aware of the issue and is working on a solution.

5.1.2 CRUSH FFT Verification

The figure from the previous section showed us what the RF spectrum should look like according to the USRP. The next step was to verify that the data was properly transmitted to the CRUSH platform. As discussed in Chapter 3, the data that flows into CRUSH comes from the output of the `rx_frontend` block which performs a DC offset correction and a phase alignment. This data is inherently different from the data the USRP records through `rx_samples_to_file`. The data through the UHD goes through additional decimation and filtering steps in `dsp_core_rx`. The data presented

to CRUSH is actually closer to the real RF spectrum than the USRP data because it is less processed and we are looking at 100 MHz of bandwidth instead of 25 MHz. On top of processing in HDL, the CRUSH data stream goes through a DDR out and DDR in process before it arrives at the User Block.

The last step in the verification process is to look at the data after the FFT has been performed in hardware on CRUSH. For this test we instructed the CRUSH platform to perform a 1024 point FFT on the incoming data and to send the raw results to the host over Ethernet. Note that the data traveling into the ML605 represents 100 MHz of data as opposed to the 25 MHz that is available to the USRP. To make the graphs more clear, Figure 5.2 has been cropped to just show 25 MHz. This is why we performed a 256 point FFT on the USRP data and a 1024 point FFT on CRUSH, so that both plots show the same resolution. As with the USRP, a sharp peak is visible at 73 MHz. The peak is not exactly at 73 MHz because a 1024 point FFT will quantize the frequency bins and each will represent approximately 100 KHz of bandwidth so the boundaries do not fall on every frequency value. This proves that our system worked correctly and the same RF energy that went into the front end of the USRP was correctly sent to the User Block and then reported properly by our FFT on the ML605.

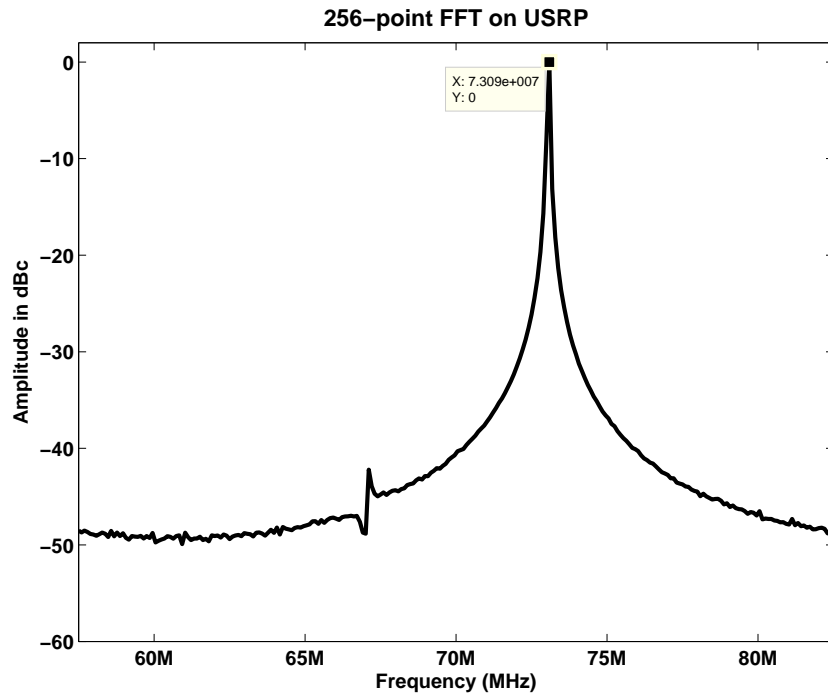


Figure 5.1: USRP Test FFT Data

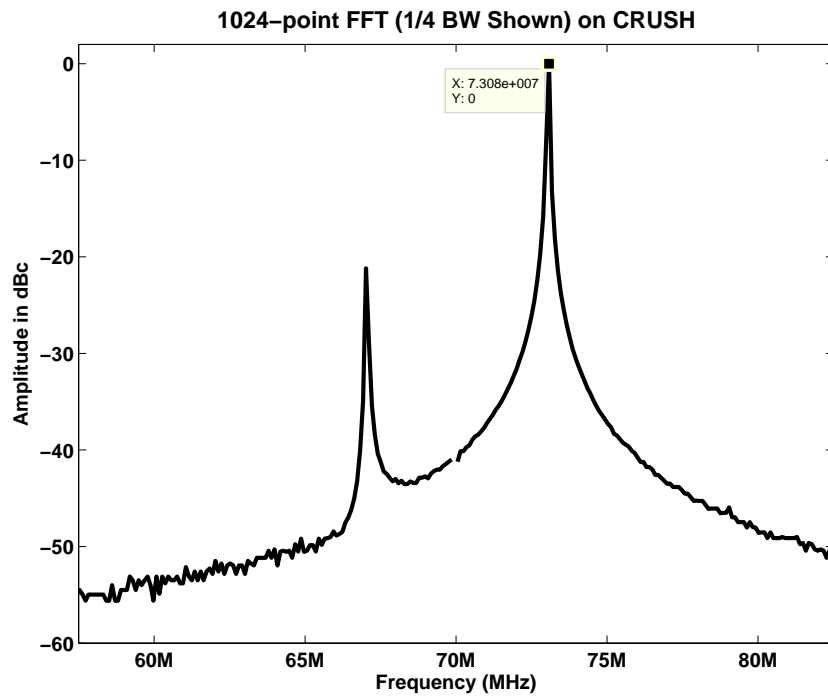


Figure 5.2: CRUSH FFT Data

5.2 Raw FFT Timing

CRUSH provides faster data manipulation by moving the signal processing closer to the receiver which results in decreased latency. Showing the exact transport and processing time for the CRUSH platform is difficult because the system consists of three asynchronous systems: the USRP, the ML605 and the host. In the following subsections we describe the process of setting up the system for testing the timing as well as the results.

5.2.1 Testing Setup

It is difficult to measure the exact amount of time that a process takes when you are dealing with three asynchronous systems with no time synchronization. Additionally, Ethernet packet systems are full of buffers and hard to time. As a result of these variables, we made the ML605 in the CRUSH system the basis for the timing analysis because of the hard real-time nature of FPGAs. To perform these tests we took the system from the original design in Figure 4.2 and added timing specific modifications (Fig. 5.3).

Two additions were made to the system to perform this analysis. The first was the timer module inside the ML605. To perform this test, we modified the system so that various components send flags to this special timer module. The timer results are stored and can later be read out by the user. The second addition was a state machine inside the USRP that allowed the data stream to be rerouted and for test

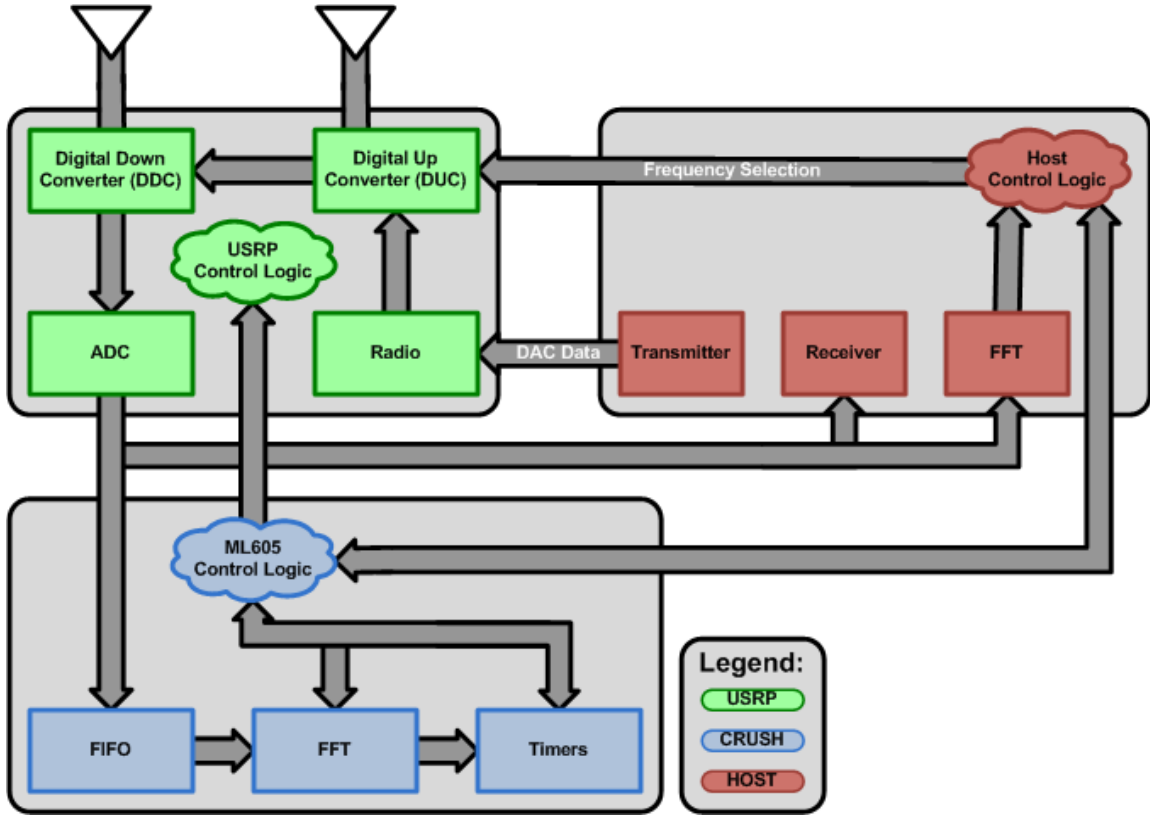


Figure 5.3: Modified Design for Timing Tests

packets to be injected. When in test mode, the system replaces the incoming ADC data with all zeros. Then, when it receives the signal to start a test, it sends out a patterned signal of size $2^{FFTSize}$ and then back to zeros. This length is chosen so a complete FFT can be calculated on the test pattern. The arrival and completion of these special packets is what yields the timing results. The test follows these steps:

1. Host tells ML605 to start test
2. ML605 resets all timers
3. ML605 tells the USRP to send the special packet
4. USRP hijacks the ADC data stream and sends the special packet, it goes to

both the host and the ML605

5. ML605 stops a timer when it completes its FFT
6. Host tells the ML605 that it has completed its FFT
7. ML605 stops the host timer
8. Host calibrates communication time

5.2.2 Detailed look at 256 point FFT on CRUSH

To get a feel for what contributes to an average time for running the CRUSH spectrum sensing algorithm, we will thoroughly examine the 256 point FFT case. To visually see the time it takes to run the algorithm, we have placed a Xilinx Chipscope core inside the ML605 FPGA. Chipscope is a debugging tool that can trigger off of any logic signal in the FPGA and then store the values of various signals in a Block RAM. It reads these values out of the Block RAM over a debug cable called JTAG. A screen capture of the timing of the 256 point FFT from Chipscope is shown in Figure 5.4.

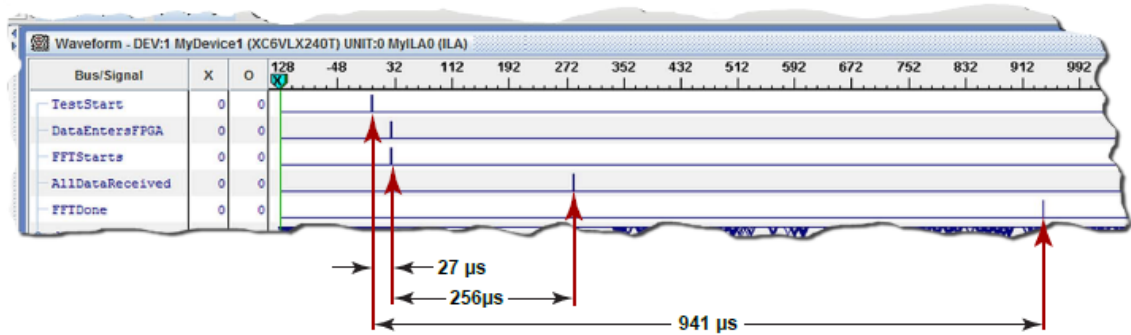


Figure 5.4: Chipscope of 256 point FFT

Time zero is when the ML605 tells the USRP to start the test, and consequently send the special packet. It takes 27 clocks from the beginning of the test for the data

to enter the User Block section of the FPGA code where processing begins. The reason for this time is various clock crossing FIFOs, registration steps, and several clocks in state machines to get the system setup. As soon as the data is inside the User Block on the ML605, the FFT starts. Since this example is a 256 point FFT it takes exactly 256 clocks to receive the packet. The majority of the time for this process is waiting for the streaming FFT to complete which in this case is $9.14 \mu\text{s}$. It should be noted that the clock rate for this system is 100 MHz which results in a 10 ns clock period. This clock rate is the same as the data coming off of the ADC on the USRP which simplifies the design flow. The timing of this particular CRUSH run is summarized in Table 5.1.

Table 5.1: Timing Analysis of 256 point FFT

Action	Cycles	Time (μs)
Start Test	0	0.0
Data enters FPGA Clock Domain	27	0.27
FFT Starts	27	0.27
All Data inside FPGA	256	2.56
FFT Complete	941	9.41

5.2.3 Overall Timing Results

After looking at one particular run through the FFT chain, we now look at all FFT sizes possible from 8 to 4k. For each FFT size, we ran the test 500 times and averaged the results. We performed the analysis on both the CRUSH and on the host. This information is shown in Figure 5.5. The blue dashed line represents the time the

host takes to complete the given FFT size and the red line represents CRUSH. Both the x and y axes are in log scale; the FFT size is on the x-axis and time in μs is on the y-axis. The gray filled portion represents \pm one standard deviation. There is a standard deviation plotted for the CRUSH platform but the FPGA timing does not vary so it is not visible.

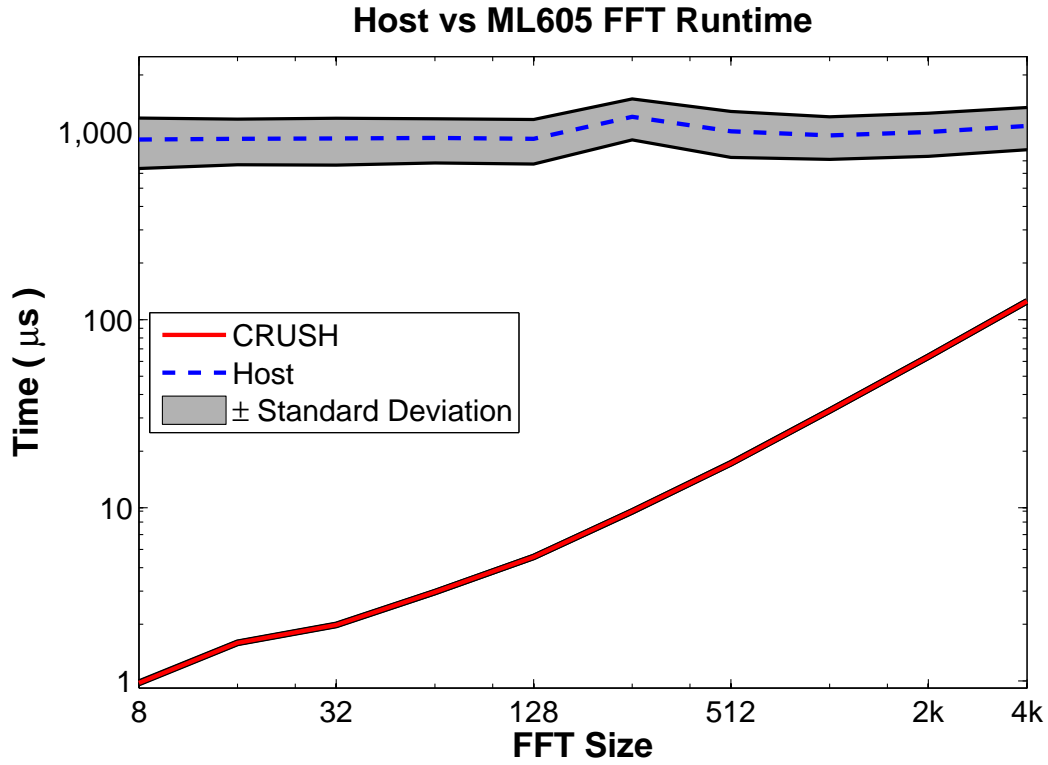


Figure 5.5: Host versus FPGA Runtime

There are several interesting things to note from these results. First, the host takes almost the same amount of time regardless of FFT size. The driving factor for the completion time for the host is not the FFT, but rather the latency of transmitting the data over Ethernet. The data for these FFTs is transmitted in one block as

defined by the USRP structure and therefore the system never needs to wait for two or more blocks. For the host side FFT, we are using FFTW; this is a very efficient implementation for FFTs of these sizes on a standard computer [18]. Great care was taken to optimize the host processing so that it would be a fair comparison. The standard host code provided by Ettus performed far worse and we determined that the USRP inherently keeps stale data in its buffers. While it does indeed stream data at a high speed, 100 MB/s on a 125 MB/s link (GbE), the data path has a high latency. After speaking with Ettus, we determined a way to flush the buffers before the start of each test. Even then, it takes approximately 6 to 7 blocks of empty data from the USRP to the host before we see the special packet that we are requesting. An additional issue occurs with larger FFT sizes. They may take more than one block of packets to transfer all of the data needed for a given FFT. This is not shown in this small subset but we did perform tests up to 32k; for 8k or larger FFTs, it takes several blocks to transmit one FFT of data and the latency increases. To our knowledge this is the first time the latency of the USRP has been explored in this much detail because it requires a complex setup of FPGA modifications and external timing circuitry that CRUSH provides.

The processing time for CRUSH scales linearly in a log plot because of the way the algorithm is designed for the FPGA. There is almost no variation at each test point because the FPGA is a hard real-time device and no other processing is going on. Table 5.1 shows that the longest task when using CRUSH is the FFT algorithm.

Table 5.2: Summary of FFT Timing Results

FFT Size	FPGA (μs)	Host (μs)	Speed up
8	1.17	907.72	774
16	1.91	915.89	479
32	2.38	920.07	386
64	3.56	925.47	259
128	5.47	916.54	167
256	9.56	1198.19	125
512	17.23	1003.83	58
1024	32.84	955.3	29
2048	63.55	995.26	15
4096	125.24	1071.79	8

Table 5.2 shows the actual timing numbers in μs as well as the speedup between CRUSH and software. CRUSH is faster than the host because of its close proximity to the data source. Typical required FFT sizes for spectrum sensing range from 64 point to 256 point because fine resolution is not needed to determine if a PU is present [19]. For 64 and 256 point sizes we see a 259x and 125x speed up respectively. This moves the timing for this application from ms to μs levels which allows CRUSH to implement near real time radio operations. In [19] the authors implement a method called localization algorithm based on doublethresholding (LAD) and utilize a 64 point FFT taking 6 μs as compared to 3.56 μs for CRUSH.

5.3 End to End Timing

The final step in the timing analysis of CRUSH is a complete, end to end timing test. This simulates the scenario where a user requests a spectrum sensing cycle and

CRUSH reports back the results, exactly as it would be used in an SDR implementing the cognitive cycle.

5.3.1 Testing Method

In the previous section we focused on precise timing of the CRUSH algorithm on the FPGA. For these next tests, we will instead look at timing from the perspective of the host computer.

Test Setup

All of the tests were performed on a Lenovo T400 with an Intel Core 2 Duo T9600 dual core processor with 8 GB of RAM and a solid state hard drive. Several extra steps were taken to ensure that the timing results were consistent. This is necessary because real-time performance characteristics of modern laptops vary depending on their present environment. A modern laptop can vary its CPU voltage and frequency depending on the current workload, temperature and software. To minimize these effects we changed the power mode to maximum performance and ran all tests when connected to an external power adapter. This forces the computer into the same high performance, high frequency mode for all tests. Additionally, the computer was restarted into safe mode with networking to make sure no extra tasks or services were running. We made sure to disable antivirus during the tests as these tools can eat up to 50 % of the processor time and will scan any files created by the program. Lastly, in Windows, it is possible to assign a program to a specific processor and

also to define a priority level for that program to operate under. For these tests, the timing program was assigned to the second processor with high priority. This was accomplished via the following command:

```
>> start /AFFINITY 0x2 /HIGH rx_samples_to_file.exe
```

As a result, the standard deviation of the tests decreased and our results are as consistent as possible on the Windows testing environment.

Precision Program Timing

The next part of the test setup is finding a precision source to use for timing measurements. On the FPGA we had a 100 MHz clock that could measure time to 10 ns precision. Even though modern computers run at extremely high clock rates, they lack precise time functions. After much research, the best we found is an Intel Assembly command called `rdtsc()` that returns the number of cycles since reset [24]. This is as precise as one can get on a computer but it still takes some time to read these counters. An outline of example code using the `rdtsc()` function is below:

```
unsigned __int64 start;

unsigned __int64 stop;

unsigned __int64 procFreq=2.8E9;

start = __rdtsc();

packetValid = a.send(endp_a, sstr.str()); //this thread is mutex locked
```

```
stop = __rdtsc();  
  
printf("%f\n", (double(stop-start)/double(procFreq))*1e6);
```

During implementation we tested rdtsc versus many other timing modules available in Windows and this method was superior, being able to measure as precisely as 50 ns compared to as much as 1000 ns with traditional Windows timers. It is very important that we measure as small of a delta as possible since our algorithm run time is on the order of 10's of μs .

UDP Packet Control

The last part of the setup for the timing results is the actual UDP sender and receiver code in C++ on the host computer. In order to better compare results to the USRP system, we utilized the same software library present on the USRP, Boost [9]. Boost has built-in support for UDP packets. We created an asynchronous receiver thread and functions to send a packet.

When running timing tests we use the following procedure. First we start the rdtsc timer. Then we send a UDP packet to CRUSH requesting spectrum sensing results. At this point we set a mutex lock on the main thread. This mutex is then unlocked inside the asynchronous receive thread when the packet is received. The receive thread also checks to make sure the packet is properly formed. Once the mutex is unlocked, the main thread continues. The next section of code stops the rdtsc timer and appends the results to a file on the hard drive. The above process is

repeated over many repetitions and also across many possible FFT sizes.

5.3.2 Results

Following the above described test plan, we ran a series of tests to measure the roundtrip time of a complete spectrum sensing cycle. For this test, all timing results represent the host time from `rdtsc` using the C++ program described previously, built on top of UHD and Boost.

The test examined FFT sizes from 8 to 4096 and averaged over 400 packets per point size (Fig. 5.6). The black line on the figure represents the average and the gray zone shows \pm one standard deviation. The trend of the line shows a slow increase in time up until 512 points. From 512 to 4096 points there is a much steeper curve. An explanation can be found by examining Table 5.2. FFT run times below 512 take less than $10 \mu s$. This is within the standard deviation of the roundtrip time and as a result is indistinguishable on the graph. Also, the difference in packet size for smaller FFTs is very minimal. So it is not until we reach larger FFT sizes that the runtime of the FFT impacts the roundtrip time. Again looking at Table 5.2, FFT size 4096 takes $125 \mu s$ which is almost exactly $125 \mu s$ more than FFT size 8.

Finally, we compare these roundtrip measurements to the Raw FFT timing to see the total speedup of CRUSH over a software only approach using the USRP (Fig. 5.7). In this plot, the FFT sizes are shown on the x axis just like the plot above. However, the y axis is now shown on a log scale. The CRUSH roundtrip time comes from Figure 5.6 and the other two lines come from Figure 5.5. Analyzing this

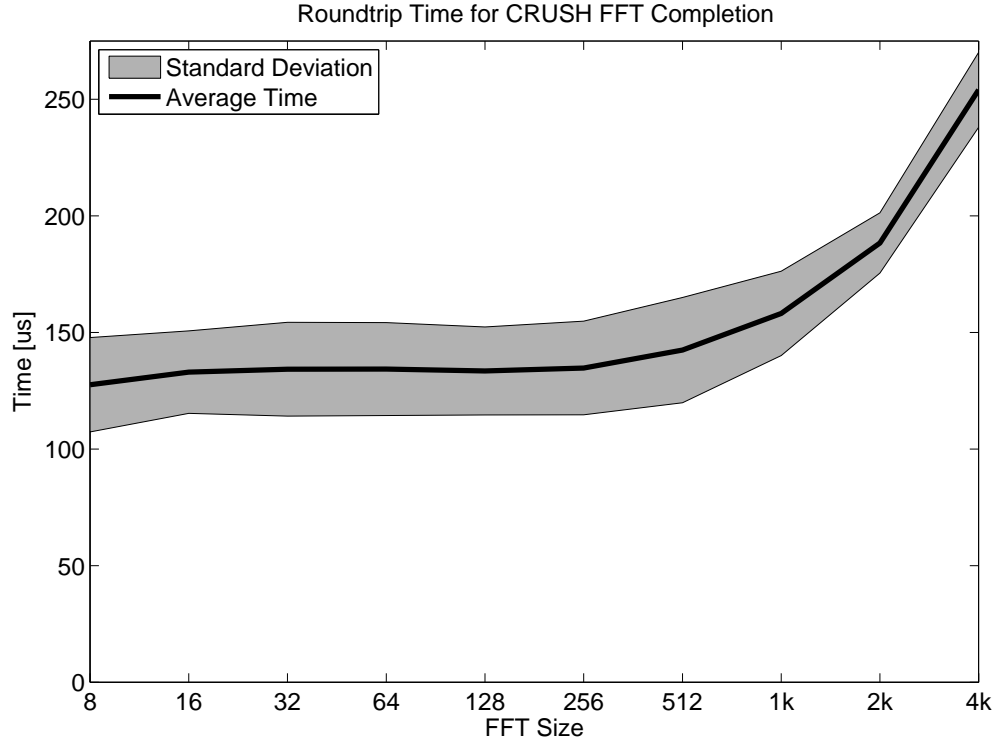


Figure 5.6: Roundtrip CRUSH Time

graph, we can see that CRUSH is approximately 10x faster than a host only version of the same algorithm. This proves that for the given FFT sizes, CRUSH is faster than software.

5.4 Available FPGA Resources

After the implementation of the CRUSH framework and the spectrum sensing application that uses a simple threshold based energy detection, the ML605 still has ample space to implement additional algorithms. The CRUSH framework uses 1.2% Slice Registers, 7.8% Block RAM and 0.4% DSP48 blocks. Spectrum sensing adds 2.6% Slice registers, 26% Block RAM and 8.3% DSP48 blocks. The majority of the

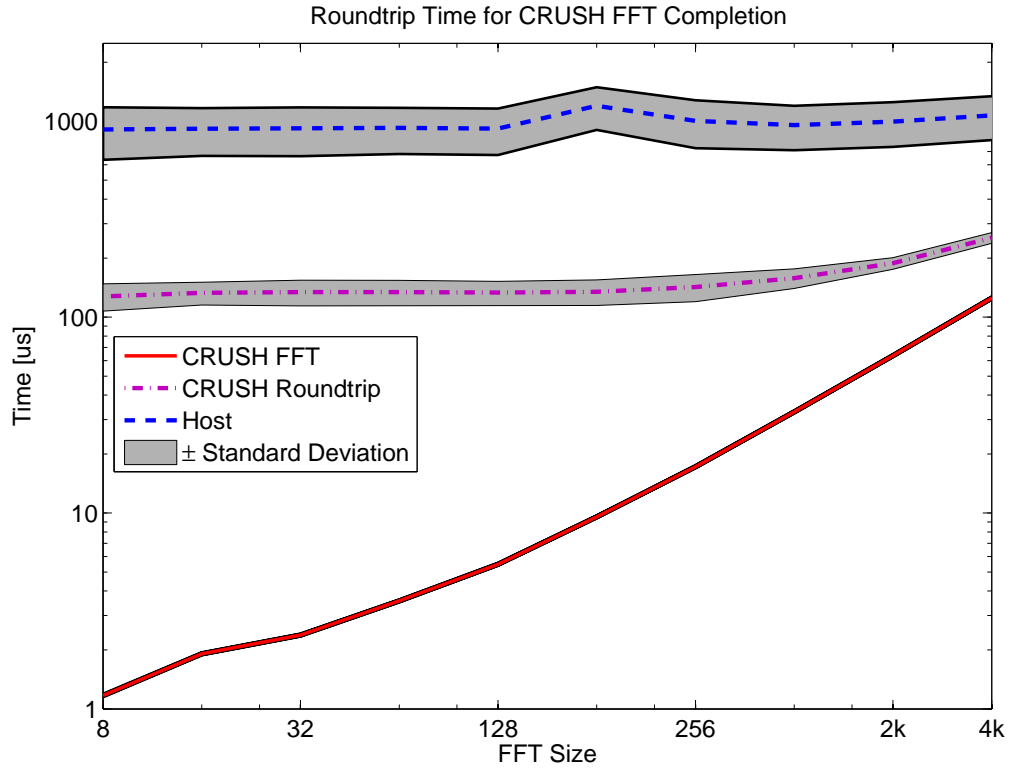


Figure 5.7: Overall Timing Comparison

spectrum sensing hardware is Block RAM used for calculating the FFT. Overall with both the framework and spectrum sensing implemented, the system still has 96.1% Slice Registers, 66.3% Block RAM and 91.2% DSP48 blocks free. One candidate for using this space is to implement additional MAC functionality in reconfigurable hardware. This could lead to novel split-MAC designs, with time critical functions of the MAC performed on the FPGA, and policy decisions undertaken by the host.

5.5 Conclusions

This chapter showed the performance of the CRUSH platform in terms of real, measurable numbers. First, we examined the data entering the CRUSH platform over the CIB. We confirmed that the data traveling over the MICTOR cable is error free and functions as expected. We also verified the functionality of the Xilinx FFT core. Next, we looked at the timing of the CRUSH platform versus software for just the completion of an FFT. In this case, CRUSH beat software by over 100x in FFT sizes of interest to CR, namely 64 and 256 point. Finally, we looked at the complete, roundtrip time as measured by the host for the CRUSH system to complete a thresholding operation. CRUSH outperformed the software solution by 10x. This chapter proves that CRUSH is an effective tool for spectrum sensing and that moving the processing closer to the host can dramatically improve the time required to complete CR tasks.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We have introduced a versatile computing platform using advanced FPGA technology. CRUSH combines a powerful FPGA with a versatile RF front end. To enable this, we created a custom interface board (CIB) that allows high speed data transfer between two widely used COTS platforms. The signal processing is now closer to the receiver allowing for implementation of high speed, real-time algorithms. This also decouples the FPGA computing resources from the SDR allowing for either to be updated independently. We implemented spectrum sensing as the first application on CRUSH. CRUSH can implement FFTs at 100x the rate of software and can perform a complete sensing cycle 10x faster than software for FFT sizes of interest to cognitive radios. We have reduced the load on the host computer by running this algorithm in hardware and we have kept the system fully configurable.

6.2 Future Work

The next stage for the CRUSH platform is integration into existing research on cognitive radio at Northeastern University. We plan to look into other approaches to perform spectrum sensing. Future iterations will dynamically threshold the FFT values instead of using precalibrated data provided by the host. Further steps might include advanced feature detection which looks at who and what is transmitting instead of just whether energy is present in a given band. These advanced methods might include wavelet analysis or cyclostationary feature detection [25][26]. Additional MAC functionality of a CR could be integrated in hardware, leading to novel split-MAC designs, with time critical functions of the MAC performed on the FPGA, and policy decisions undertaken by the host. Finally, one could use this platform for research outside of the realm of software defined radio. The combination of an ADC, DAC and a powerful FPGA backend could be used in other signal processing research fields such as radar.

Appendix A

List of Acronyms

ACK	Acknowledge
ADC	Analog to Digital Converter
AMC	Automatic Modulation Classification
ASIC	Application-Specific Integrated Circuits
BEE	Berkeley Emulation Engine
CBM	Correlation Based Method
CIB	Custom Interface Board
COTS	Commercial Off the Shelf
CR	Cognitive Radio
CRUSH	Cognitive Radio Universal Software Hardware
DAC	Digital to Analog Converter
dBc	Decibels relative to the Carrier
dBm	Decibels relative to 1 mW
DCM	Digital Clock Manager
DDR	Double Data Rate
DSP	Digital Signal Processing
FIFO	First In First Out
FFT	Fast Fourier Transform

FMC	FPGA Mezzanine Card
FPGA	Field-Programmable Gate Array
GbE	Gigabit Ethernet
GUI	Graphical User Interface
GUIDE	GUI Design Environment
HDL	Hardware Description Language
HPC	High Pin Count
ISM	Industrial Scientific and Medical
JTAG	Joint Test Action Group
KNOWS	Networking Over White Spaces
LPC	Low Pin Count
MAC	Media Access Control
MICTOR	Matched Impedance Connector
MIMO	Multiple Input Multiple Output
MBps	Megabytes per second
MSPS	Mega-Samples Per Second
MUX	Multiplexer
PPC	PowerPC
PU	Primary User
REQ	Request
SATA	Serial Advanced Technology Attachment
SDR	Software Defined Radio
SERDES	Serializer / Deserializer
TEMAC	Trimode Ethernet Media Access Control
USRP	Universal Software Radio Peripheral

UHD Universal Hardware Driver

WARP Wireless Open Access Research Platform

Bibliography

- [1] Ettus Research. USRP - Ettus Research. [Online]. Available: <http://www.ettus.com/>.
- [2] I. F. Akyildiz, W.-Y. Lee, and K. R. Chowdhury, "CRAHNS: Cognitive Radio Ad Hoc Networks," *Ad Hoc Networks*, vol. 7, no. 5, pp. 810 – 836, 2009.
- [3] T. Yucek and H. Arslan, "A survey of spectrum sensing algorithms for cognitive radio applications," *Communications Surveys Tutorials, IEEE*, vol. 11, no. 1, pp. 116 –130, 2009.
- [4] A. MacKenzie, J. Reed, P. Athanas, C. Bostian, R. Buehrer, L. DaSilva, S. Ellingson, Y. Hou, M. Hsiao, J.-M. Park, C. Patterson, S. Raman, and C. da Silva, "Cognitive radio and networking research at virginia tech," *Proceedings of the IEEE*, vol. 97, no. 4, pp. 660 –688, April 2009.
- [5] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. Cavallaro, and A. Sabharwal, "Warp, a unified wireless network testbed for education and research," in *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference on*, june 2007, pp. 53 –54.
- [6] BEEcube. Berkeley emulation engine. [Online]. Available: <http://beecube.com/>.
- [7] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh, "White space networking with wi-fi like connectivity," *SIGCOMM Comput. Commun. Rev.*, vol. 39, pp. 27–38, Aug. 2009.
- [8] Ettus Research. Usrc hardware driver (uhd). [Online]. Available: <http://code.ettus.com/redmine/ettus/projects/uhd/wiki/>.
- [9] Boost. Boost. [Online]. Available: <http://www.boost.org/>.
- [10] GNURadio. Gnuradio. [Online]. Available: <http://gnuradio.org/redmine/projects/gnuradio/wiki>.
- [11] Mathworks. Matlab and simulink support package for usrp hardware. [Online]. Available: <http://www.mathworks.com/discovery/sdr/usrp.html>.

- [12] ANSI. (2010) FPGA Mezzanine Card (FMC) standard. [Online]. Available: <http://www.vita.com/fmc.html>.
- [13] TE Connectivity. Matched impedance connector (mictor). [Online]. Available: <http://www.te.com/catalog/minf/en/428/>.
- [14] Texas Instruments. Tlk2701 transceiver. [Online]. Available: <http://www.ti.com/product/tlk2701>.
- [15] XILINX. (2011) Virtex-6 FPGA TEMAC. [Online]. Available: www.xilinx.com/products/intellectual-property/TEMAC.htm.
- [16] Network Sorcery. (2011) UDP. [Online]. Available: <http://www.networksorcery.com/enp/protocol/udp.htm>.
- [17] S. Haykin, D. Thomson, and J. Reed, "Spectrum sensing for cognitive radio," *Proceedings of the IEEE*, vol. 97, no. 5, pp. 849–877, may 2009.
- [18] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
- [19] T. Hanninen, J. Vartiainen, M. Juntti, and M. Raustia, "Implementation of spectrum sensing on wireless open-access research platform," in *Applied Sciences in Biomedical and Communication Technologies (ISABEL), 2010 3rd International Symposium on*, nov. 2010, pp. 1–5.
- [20] FCC. White Space. [Online]. Available: <http://www.fcc.gov/topic/white-space/>.
- [21] M. Rahman, C. Song, and H. Harada, "Development of a TV white space cognitive radio prototype and its spectrum sensing performance," in *Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM), 2011 Sixth International ICST Conference on*, june 2011, pp. 231–235.
- [22] B. Shishkin, D. Pfeil, D. Nguyen, K. Wanuga, J. Chacko, J. Johnson, N. Kandasamy, T. Kurzweg, and K. Dandekar, "SDC testbed: Software defined communications testbed for wireless radio and optical networking," in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt), 2011 International Symposium on*, may 2011, pp. 300–306.
- [23] XILINX INC. (2011) Ds260: Fast fourier transform v7.1. [Online]. Available: www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf.
- [24] Intel. Architecture Manual. [Online]. Available: download.intel.com/design/processor/manuals/253669.pdf.

- [25] Z. Tian and G. B. Giannakis, "A wavelet approach to wideband spectrum sensing for cognitive radios," in *Cognitive Radio Oriented Wireless Networks and Communications, 2006. 1st International Conference on*, june 2006, pp. 1–5.
- [26] A. Fehske, J. Gaeddert, and J. Reed, "A new approach to signal classification using spectral correlation and neural networks," in *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, nov. 2005, pp. 144–150.