Enhancing GNU Radio for Hardware Accelerated Radio Design

Charles Robert Irick

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter Athanas, Chair
Joseph Tront
Patrick Schaumont

June 4, 2010
Blacksburg, Virginia

Keywords: GNU Radio, FPGA, Virtex-5, SDR
Copyright 2010, Charles Robert Irick

Enhancing GNU Radio for Hardware Accelerated Radio Design

Charles Robert Irick

(ABSTRACT)

As technology evolves and new methods for designing radios arise, it becomes necessary to continue the search for fast and flexible development environments. Some of these new technologies include software defined radio (SDR), Field Programmable Gate Arrays (FPGAs), and the open source project GNU Radio. Software defined radio is a concept that GNU Radio has harnessed to allow developers to quickly create flexible radio designs. In terms of hardware, the maturity of FPGAs give radio designers new opportunities to develop high-speed radios having high-throughput and low-latency, yet the conventional build-time for FPGAs is a limiting factor for productivity. Recent research has lead to reductions in build-time by using FPGAs in a non-traditional manner, meaning productivity no longer has to be sacrificed. The AgileHW project demonstrated this concept and will be used as a basis to develop an overlaying architecture that uses a combination of the technologies mentioned to create a flexible, open, and efficient environment for radio development. This thesis discusses the realization of this architecture with the use of Xilinx FPGAs as a hardware accelerator for an enhanced GNU Radio.

This work has been funded in part by the Strategic Technology Office (STO) at DARPA.

Acknowledgments

Where I am today would not be possible without the help and support of many people. For those not mentioned specifically, thank you.

I would like to thank Dr. Peter Athanas, my advisor, for giving me the opportunity to work in the CCM lab and providing me with the guidance needed for achieving my goals.

Thanks to Dr. Joseph Tront for being on my committee and helping me throughout my education at Virginia Tech.

I would also like to thank Dr. Patrick Schaumont for answering all of the questions I had in your classes (which was sometimes a lot), and for being a member on my committee.

Thanks to Adolfo Recio for sharing your knowledge in the field of communications and helping me develop a better understanding of radio design.

Thank you to Tannous Frangieh and Ali Sohanghpurwala for being great friends and helping me to stay grounded throughout my journey.

I would like to dedicate this thesis to my family. Thank you for being so supportive and helping me throughout my academic career. A special thanks to my father for showing me the importance of being well-educated. Without you, I would not be where I am today.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objective 3

1.3 Organization 5

2 Background **6**

2.1 SDR Platforms 6

2.1.1 SORA 7

2.1.2 WARP 8

2.1.3 OSSIE 9

2.2 CUDA 10

2.3 Hybrid Architectures 11

2.4 MAC Layer Development 13

2.5 Agile Hardware Work at Virginia Tech 13

3 GNU Radio **15**

3.1 Current GNU Radio Model 15
3.1.1 Streaming Dataflow and Scheduling 17
3.1.2 Modular Component Design 18
3.2 Improvements 20
3.2.1 Expanding GNU Radio with Hardware 20
3.2.2 Accessability of AgileHW 21

4 Agile HW **23**

4.1 Compile-time 24
4.1.1 Current Static Design 26
4.1.2 Static Design using GNU Radio 27
4.1.3 Dynamic Region 28
4.2 Run-time 30
4.2.1 Resource Allocation 31
4.2.2 Place and Route 31
4.2.3 Systembits 31

5 Implementation **34**

5.1 System Usage 34
5.2 Hardware Layout 36
5.3 Data Packet Format 37
5.4 Software Layout 39

5.4.1 Python Modules 39
5.4.2 Module Implementation 41
5.4.3 Low-level C++ 42
5.5 Hardware Modules 43
5.5.1 Ethernet Module 43
5.5.2 Radio Transmission 44
5.6 Challenges 47
5.6.1 Hardware Connectivity 47
5.6.2 Dataflow 48
5.6.3 Static vs. Dynamic Run-time 49
5.6.4 Streaming Radio Modules 49

6 Results **51**
6.1 Testing and Hardware 51
6.2 Proof of Concept 52
6.3 Quantitative Results 56
6.3.1 Latency 56
6.3.2 Throughput 58
6.3.3 Utilization 63
6.4 Attribute Comparison 64
6.5 Analysis of Code Complexity and Productivity 65

7 Conclusion **68**

7.1 Future Work 69

7.1.1 MAC Layer Protocols 69

7.1.2 Partial Run-time Reconfiguration 69

7.1.3 Radio Designs 69

Bibliography **71**

A Extras **76**

A.1 Code Organization 76

A.2 Autotools 77

List of Figures

2.1 WARP Support Model 9

3.1 Generalized GNU Radio 16

3.2 Simple GNU Radio Application 19

3.3 Correlation of Modular Concepts 21

4.1 Diagram of AgileHW Tool-flow 29

5.1 Usage diagram for enhancements 35

5.2 Physical Layout 36

5.3 Packet Formats 38

5.4 Software Class Structure 40

5.5 Hardware Modules 43

5.6 Ethernet Block Diagram 44

6.1 Spectrum Analyzer for Sin Wave 53

6.2 Capture of file transmission 55

6.3 Capture of file reception 55

6.4 FPGA Utilization 63

List of Tables

6.1 Latency Analysis on Network with Switch. 56

6.2 Latency Analysis off Network with Switch. 57

6.3 Latency Analysis off Network without Switch. 57

6.4 Throughput Analysis without Burst 59

6.5 Throughput Analysis 2 Packet Burst 60

6.6 Throughput Analysis 4 Packet Burst 61

6.7 Throughput Analysis 64 Packet Burst 62

6.8 Comparison of features and attributes for SDR platforms 64

Chapter 1

Introduction

1.1 Motivation

Wireless technologies have drastically changed the way people live and communicate over the last hundred years. Many of these technologies such as, AM/FM radios, television, navigation systems, mobile phones, and wireless LAN have become an integral part of peoples lives.

Although wireless technologies have become ubiquitous, there are still many problems and challenges in this domain. A large portion of the problems originate from complexities in hardware design. Design time and productivity are two items that suffer from this. Advancements in technology, design, tool-flows, and software philosophies have helped to resist the negative aspects of hardware design. Even with these advancements many major issues still exist.

One major issue for radio design is the gap between algorithm design and implementation. The tasks of algorithm design and implementation are generally split between employees or students. The separation of work and many translations between people can cause problems in reaching the proper goals. Work separation stems from differences in design

and implementation languages where a solution is not readily available. Implementation is also separate because it can be a complicated and arduous task.

Hardware implementation requires synthesis from a general Register Transfer Language (RTL) description into a platform specific specification, much like the translation from a software programming language into machine code for a processor. RTL languages complicate the design process by taking the designer away from the specification domain. Also, hardware synthesis suffers from long compile times when compared to software mostly due to the lack of a true incremental build solution. Software compilation creates intermediate object files so when changes are made to the code only the object files being affected need to be recompiled. Hardware design is maturing to the point where this model is being observed [1], yet the technology is in its infancy. Once the designer has their language of choice, regardless of platform target, a software philosophy that the code should be openly accessible has many benefits.

Open source software [2] is a philosophy that source code should be openly available to the public. The benefits from open source software come from the fact that many people are allowed to view and modify a project's source code. Some of the tangible benefits include cost, stability, and security. Cost is an obvious benefit as the code does not require any licensing to view. Stability and security both arise from the chance of a large developer base to provide support as well as checks and balances. Even with many developers, productivity is still an issue separate from open source software. Not all problems are ones in which a large number of developers can be assigned.

Productivity is an important aspect in radio design to both research and industry. In research there is an interest in having students up and running, producing quality work, within a short period of time. A quick initial learning curve leaves more time to make real advances. In industry the concept of productivity rolls over directly to cost. It benefits companies financially to have high productivity when testing, designing, or rolling out new products. For instance, Company A would like to make an improvement to Radio

B, yet the designer of Radio B no longer works there. A high-level, easy to use, design environment would allow new Employee C to quickly understand the way Radio B was created and how it works. New Employee C could start making the desired improvements to Radio B in a short amount of time. The speed that new Employee C completes the improvements to Radio B directly affects the cost of the upgrade for Company A. Software can help because of its agile nature, but this solution can also have its drawbacks.

As General Purpose Processors (GPP) continue to advance, their power in computation increases. Even with these increases GPPs are rarely a desired target for complex and high speed computational problems, especially in signal processing. The reason for this stems from the GPP model being one that is meant to juggle a tasks. The GPP has a complex software operating system controlling its use and managing tasks. The large number of concurrent tasks leads to time sharing of the GPP resource making for a non-deterministic environment that is impractical for a real-time signal processing application. Dedicated hardware provides a much more suitable environment for carrying out modern computational problems as they can be designed around one task as efficiently as possible. Problems with signal processing on GPPs, as well as the problems previously stated, give good motivation for the work in this thesis.

1.2 Objective

The goal of this work is to create a framework, with productivity in mind, that aids in the design of contemporary high-speed radios by augmenting the GNU Radio with concepts from the AgileHW project and a flexible co-processor. The enhancements will be designed to supplement the existing tool and dataflow allowing for increased flexibility while maintaining the productivity and fast-to-deploy nature of the existing framework.

The GNU Radio project [3] provides a natural environment to demonstrate the potential impact of what the AgileHW project [4][5][6] has to offer to a larger ”mainstream”

audience. The GNU Radio infrastructure already provides a high-level framework that allows for fast prototyping and design while being easy to use, openly available, and easy to modify. These factors alone make it attractive to both a hardware-oriented and theoretical communications audience. While GNU Radio provides a feature-rich set of tools to a radio designer, there still exists the possibility to break new ground in this area.

An external hardware co-processor can resolve computational issues with GPPs. The design of GNU Radio supports additions and modifications easily. If care is taken to ensure addition of the accelerator is flexible, it can mesh into the existing GNU Radio flow with minimal effort. It is the aim of this thesis to provide a flexible enough co-processor as to not inhibit the standard GNU Radio flow, yet allow the designer to have the option of hardware acceleration. An agile co-processor will allow computationally complex algorithms without sacrificing productivity.

Productivity and agility are important metrics when designing an environment that is used to develop complex systems. Yet, increasingly complex radio designs naturally counteract these ideas. The GNU Radio project is at the point where innovation to its framework could allow a designer to continue to achieve more complex designs without sacrificing agility or productivity. The goal of this work is to design a flexible, robust, and open architecture for the development of radios, with the principles of AgileHW in mind, using the existing framework of GNU Radio and a Xilinx FPGA co-processor.

Summary of Contributions:

- Develop an efficient and robust framework
- Aid in advancement of AgileHW with said framework
- Increase platform freedom for AgileHW
- Use open source software development for wide acceptance
- Provide the option of hardware acceleration for GNU Radio modules
- Maintain existing GNU Radio flow

1.3 Organization

The thesis is organized in the following way. Chapter 2 provides a background for understand the work of this thesis as well as informing the reader about related research. Chapter 3 talks about the GNU Radio project and the reasoning behind its use. Chapter 4 gives some background into the AgileHW project and how it has helped with advancing Partial Reconfiguration for FPGAs. Chapter 5 provides an in-depth look at how the enhanced GNU Radio environment was implemented. Chapter 6 shows the results of the work performed, and Chapter 7 summarizes the contributions of this thesis, provides a conclusion to the work performed, and presenting ideas for future work.

Chapter 2

Background

Wireless radio communication began in the late 1800's and early 1900's with many different methods of implementation. These methods have changed drastically over the last century starting with devices like the crystal radio all the way to today's modern digital radios. On the forefront of radio design are technologies like SDR, FPGAs, and GNU Radio. This chapter will highlight a range of the past and current work relating to SDR, FPGAs, and GNU Radio.

2.1 SDR Platforms

Software Defined Radio has sparked the interest of industry and academia over the course of the last decade [7][8][9][10][11][12]. Many ideas and applications have branched out of these platforms creating a rich environment of choices for engineers. Some of these choices have become successful while others have failed. This section provides background into modern SDR platforms.

2.1.1 SORA

SORA [13] is a project, by Microsoft, which aims at improving the performance of a SDR environment by using advanced multi-core CPU techniques and a dedicated hardware front-end. Their radio control board (RCB) uses PCI-e and DMA transfers to efficiently move data between the host and RCB. The RCB is used as the radio front-end for performing the close-to-antenna work like down-conversions and A/D. Tan et al. [13] states a short-coming of the current GNU Radio being that it only achieves a few Kbps over an 8MHz channel whereas a modern radio standard like IEEE 802.11 (e.g., WiFi) requires multiple Mbps on a wider 20MHz channel. They use this argument to demonstrate a SoftWifi implementation, using the RCB, that meets the IEEE 802.11a/b/g standards.

The software system for SORA takes advantage of modern features in multi-core architectures like lookup tables (LUTs) and single instruction multiple data (SIMD) extensions. Tan et al. use the large amounts of cache memory to store pre-calculated results into LUTs. They also use dedicated cores, providing guarantees on return, to help meet the real-time requirements of IEEE 802.11.

With the use of performance oriented features of multi-core CPUs, PCIe, and FPGA technologies, the SORA platform does well in terms of performance. Using a GPP has the benefit of software agility, yet the heavy optimizations required for its use may be an issue in future designs. One of the benefits of using FPGAs and AgileHW is that the flexibility of software can be brought into hardware. The other advantage of FPGAs is the radio algorithm can be designed at the PHY layer without the concern for heavy scheduling optimizations and pre-computing datapaths.

GNU Radio, which will be discussed in detail in the following chapter, comes with a well defined framework for expandability and creating new designs. Again, a special purpose system requires care to cleverly optimize a solution for use on a GPP because they are not natively suited for radio processing tasks. GNU Radio lets the designer script their radio design for quick modifications and easy gratification. The modularity and standard

structure of the datapath make changes simple. The flexibility shown in this work will be displayed using the enhanced GNU Radio framework.

2.1.2 WARP

WARP [14][15] is a SDR platform in development at Rice University. The aim of this platform is to provide support for the development of modern radios with academia and research in mind. Rice have developed a custom board with an emphasis on four levels of support, as shown in Figure 2.1. The first layer is an FPGA at the center of the system for communication logic and computation. The second layer is a dedicated Power PC core in the FPGA for implementing higher level concepts in software. Layer three gives the ability to add custom daughter-cards for radio front-end needs. The final layer is inter-board communication using MGT pin-outs directly from the FPGA for communication between multiple instances of WARP boards. The result is a well supported platform for researching new wireless communication algorithms.

This platform takes advantage of FPGAs and an existing library of example designs, yet it lacks agility and is tied down to the traditional use of FPGAs. If a radio design does not exist there is no clear picture on how easy or difficult it is to create something new. Even if the user wishes to switch pre-existing designs they will have to re-synthesize the desired design, program the entire FPGA, and load up a new software executable for controlling the dataflow.

WARP uses the traditional model of FPGA design. This yields high-throughput and low-latency designs without restrictions, yet does not address aspects related to productivity. The application layer is also special purpose and requires more attention to memory interfaces and communication between software and hardware. In the scripted environment of GNU Radio, many of these details are taken care of. The scripting capabilities of GNU Radio and the flexibility and computational power of the FPGA yield a model that benefits from the capabilities of WARP without sacrificing productivity.

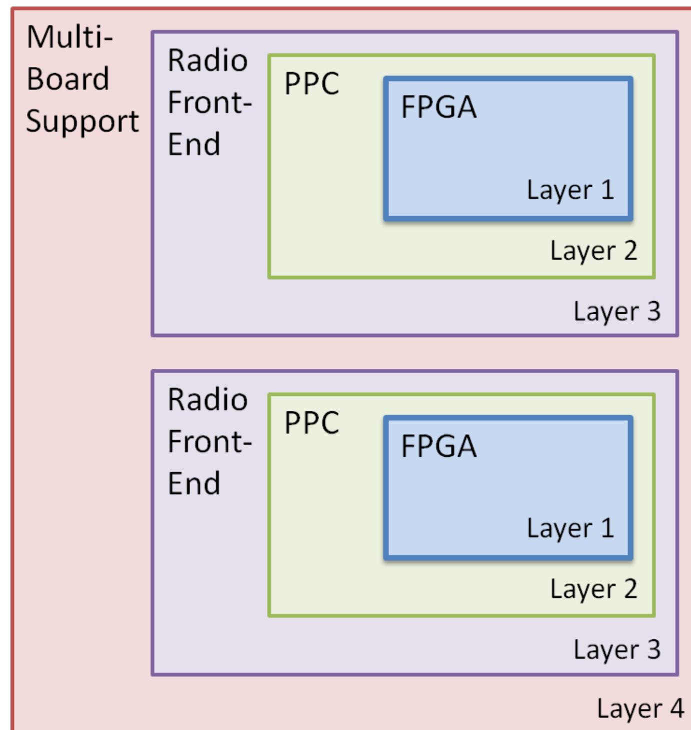

Figure 2.1: The four layers of WARP.

2.1.3 OSSIE

A project by the Wireless@Virginia Tech team known as Open-Source SCA Implementation - Embedded (OSSIE) [11] is being used to develop a set of tools for the rapid development of SDR components in a user-friendly environment. The tools are based on the U.S. Department of Defense's Software Communications Architecture (SCA). This architecture is currently used for the popular Joint Tactical Radio System (JTRS) that is used to build current U.S. Military radios. The aim of OSSIE is to simplify and develop a framework that is easy to use, open source, and built upon SCA, that provides a solid background for SDR development.

The OSSIE project is targeted at graduate students of Electrical and Computer Engineering. It uses C++ as the implementation language due to its familiarity to the target audience. The idea is to present an architecture with a small learning curve so students

are able to understand the environment and contribute significant designs within the usual one to two year period of a Masters education. This means isolating the developer from some of the more complicated methods used in SCA, like CORBA [16].

With the success of OSSIE has come the acceptance of its use in industry as well. The project is supported by the NSF, but has also been used and supported by many companies including Office of Naval Research (ONR), Science Applications International Corporation (SAIC), Tektronix, and Texas Instruments. Support has been given in the form of finances as well as improvements to the framework and tools.

As with any SDR environment the key idea is usually related to rapid prototyping. This is the case with the GNU Radio architecture as well. The parallel to OSSIE is that GNU Radio is also an open source environment that is easy to jump into while also providing the user with powerful tools. The OSSIE project has made use of a hybrid design environment using an FPGA to offload signal processing tasks from the GPP [17]. This is very similar to the goals of this thesis and parallels will be drawn in the results.

2.2 CUDA

The release of Nvidia's CUDA language [18] combined with an effort towards general purpose computing has the GPU emerging as a contender in high performance computing and SDR. An initial example shows an attempt to use GNU Radio with a GPU using CUDA [19]. Their experiment suffered from a lot of overhead, which is a common problem with CUDA. CUDA is a new language and in its infancy is somewhat difficult to use. The programmer is required to handle tasks between the thousands of available cores manually, which makes it very easy to write inefficient code. Each core has its own unshared memory, so much care is needed to develop parallel algorithms that ensure efficient memory transfers. The programmer must be clever in their design to reduce

overhead by limiting the number of system calls. This is currently the "Wild West" [1] of high performance computing, but it seems to be something Nvidia is taking seriously and will continue to improve in the future.

Kim, Hyeon, and Choi [20] use a GPU to accelerate signal processing tasks in an SDR environment. The authors create an implementation of a mobile WiMAX terminal for a proof of concept. The hardware and software systems were interfaced with the use of DMA transfers. The GPU used was a Nvidia GeForce 9800GTX and for comparison some of the signal processing tasks were performed on a TMS320C6414 DSP. Their results found a 90 times speed up for a Viterbi decoder operation, being a subset of functionality for a full radio modulation chain. The final system was capable of 2Mbps throughput and is speculated to be able to handle IEEE 802.11.

With a GPU, a developer can benefit from software agility while maintaining high computational powered designs. Comparing GPUs to FPGAs, the FPGA generally has the advantage in processing and power [21], but depends on the application. Full utilization of an FPGA will result in much lower power than full utilization of a GPU [22][23]. With enhanced GNU Radio the benefits of agile software development will be realized without a large power requirement.

2.3 Hybrid Architectures

As radio technologies advance, new implementation platforms are tested and integrated along-side existing systems. Using a combination of platforms can lead to a complex heterogeneous environment. Many systems, as noted by the previous section, take advantage of multiple technologies by combining them into one hybrid environment.

A popular hybrid environment for SDR has software running on a GPP with modulation schemes offloaded to a dedicated high speed computing device like an FPGA or a GPU.

[1] http://en.wikipedia.org/wiki/American_Old_West

Tachwali et al. [24] shows the design of a BPSK Transceiver that is built on a hybrid platform consisting of a GPP/DSP/FPGA system. The DSP processors were used to generate and receive sample data as well as communicate with the FPGA over a Video Processing Sub-system (VPSS) link designed for high speed communication. A low speed shared memory interface was also designed and used for synchronization. The FPGA was responsible for serializing, modulation/demodulation, up-sampling, filtering, interpolation/decimation, up-conversions, and ADC/DAC. For the results they used Simulink to generate radio data that was sent over Ethernet to the hardware platform. The signals that resulted at each stage of the process were plotted. This is important as it brings forward some of the issues related to cross platform development and how they solved them.

Communication interfaces are a common problem in hybrid systems. Different platforms may not be designed to work together yet can be connected to obtain reliable and efficient dataflow. Connecting unlike devices at the physical level can be simple in SDR because modulation schemes tend to act in a linear dataflow producer/consumer model. There is work being done [25] that uses a GPP/DSP/FPGA environment, and aspects of partial reconfiguration of FPGAs, to build radio systems . A Hierarchical Distributed Configuration Management system was proposed to handle different levels of granularity involved with reconfiguration on a heterogeneous system. This work proposes a special wrapper interface for the radio components. The interface uses asynchronous external communication that has the benefit of being compatible with any underlying architecture. This is an important idea as the interface between the USRP2 and host system in the GNU Radio environment is also asynchronous. Some of the problems that this work has proposed to solve are similar to problems encountered in the work of this thesis.

2.4 MAC Layer Development

GNU Radio provides a good environment for the development of physical level constructs. The need exists for a MAC layer protocol that provides more control over communication between software blocks. The MAC layer is also responsible for handling acknowledgement packets who's required round trip time can be on the order of tens of microseconds, so efficiency is necessary. Not only do MAC layer protocols require efficient implementations but also a dataflow model that handles robust protocols. Attempts at a true MAC layer protocol have been fielded using GNU Radio but have had minimal success.

A few papers have been written exploring MAC layer development using GNU Radio. One project [26] attempted MAC layer concepts by combining all physical layer modules into one unit and using the software Click to develop the MAC layer on top. This was necessary because the GNU Radio blocks and scheduler are made for PHY layer development and do not have robust intermodule communication. A second project [27] used a similar approach with the Click software but built a TCP/IP stack on top of GNU Radio that provided support for proper file transmission and data streaming.

2.5 Agile Hardware Work at Virginia Tech

The configurable computing lab at Virginia Tech has based a lot of its work towards the advancement of productivity in radio design. Two major projects in the lab have been RapidRadio and Wires-On-Demand.

RapidRadio [28][29] was a human-in-the-loop system that allowed a novice hardware designer to quickly build a radio system targeted at an FPGA. This reduced the burden of HDL development required by the user with only minor sacrifices in hardware efficiency.

Wires-On-Demand [4][5][6] has been a continuing project related to the use of dynamic partial run-time reconfiguration on FPGAs. The model for Wires-On-Demand is to have

a large undefined ”sandbox” region for placement of pre-designed hardware modules. Some design effort is required on the front-end to develop the modules and the static FPGA design that communicates with, and houses, the sandbox. The pay-off is that once the modules have been designed the entire system can be abstracted to simply dragging and dropping modules to design an FPGA system. The target for Wires-On-Demand has mostly been the radio and signal processing domains because it meshes well with problems in this area.

The second version of Wires-On-Demand, called AgileHW, aims to improve the use and scope of dynamic run-time partial reconfiguration. New modules have been designed to support higher speed radios (up to 6Mbps) as well as dual-channel radios. A dual-channel radio allows a dedicated channel for radio configuration while the secondary channel is used for desired radio tasks, be it data transfer or voice communication. The use of a dual-channel radio with AgileHW means modules can be inserted or removed from the sandbox region over the command channel without interference on the data channel. Channel separation can also make security easier by restricting the configuration channel to trusted users. The idea of a dual-channel radio is not etched into the framework or goals of AgileHW, but it shows the power and flexibility achievable.

Common themes between the work presented in this chapter have been flexibility, computational power, and ease of use. Many of the solutions have solved issues in multiple categories. For instance SORA has proved computational power and flexibility, yet it remains to be seen if ease of use is something it can claim. The solution provided by this thesis aims to tie together many benefits that other solutions have shown. An enhanced GNU Radio framework that takes advantage of the flexibility and ease of use of GNU Radio, as well as the computational power and agility of AgileHW, can provide a solution to these problems.

Chapter 3

GNU Radio

The GNU Radio project was started in 2001 by Eric Blossom and John Gilmore as a way to deploy software defined radio systems. Since then, the project has been used and improved upon by many contributors including hobbyists and universities. Contributions to the project have been made in many different research areas and topics.

3.1 Current GNU Radio Model

The current GNU Radio environment relies on the Universal Software Radio Peripheral (USRP) as the preferred radio front-end for performing radio signal processing. The USRP provides a common radio front-end hardware platform that can be interfaced to a host machine allowing radio back-end signal processing operations, such as demodulation, in software. The host machine is a standard computer running GNU/Linux, Windows, or OSX and uses the concept of software defined radio (SDR) to process incoming and outgoing radio data with common high-level software programming languages.

The ease of use provided by GNU Radio stems from the underlying Python programming language foundation, which provides nearly instant gratification for scripting a radio

design. The GNU Radio programming model is fairly simple: the radio designer describes a radio as a flow graph. A flow graph presents a hierarchy defining how radio data moves through the desired system. The nodes in a graph represent the signal processing functions. A generalized model for this can be seen in Figure 3.1 showing how the nodes correlate with a specific function implementation while the flow graph connections represent communication and interfacing of the nodes. Each node then fits into an overall graph through a highly organized connectivity model. The object-oriented nature of Python provides a logical means of extending node functionality and resolving issues in connecting modules of vastly differing types. A robust library of pre-designed radio modules lower development time and provide a simple interface for instantiation.

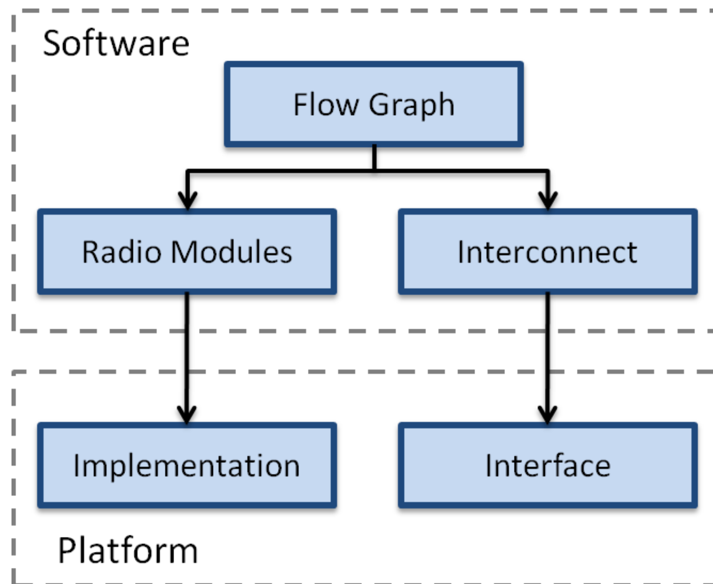

Figure 3.1: A generalized view of GNU Radio.

The GNU Radio programming model is not only elegant, but also quite powerful. A few hundred lines of object-oriented Python code is enough to make an elaborate radio. The project is a significant contribution to the communications community in that it (a) is a productivity enhancement for anyone trying to quickly prototype and deploy a radio transmitter or receiver, (b) greatly expands the audience of designers that can create radios, and (c) provides an ideal test-bed for researchers to perform wired and wireless

networking research.

Implemented radios in the GNU Radio environment are only partially executed on the general-purpose host computer. The front-end signal processing is done in an FPGA contained in the USB-attached USRP or the Gigabit Ethernet-attached USRP2. The FPGA provides an essential part of the GNU Radio chain. It is responsible for the high data-rate and repetitive high-throughput computations associated with radio front-end processing (digital down-conversion and up-conversion, filtering, equalization, data decimation and interpolation). It is also responsible for controlling the radio analog components and data acquisition components. Unlike the rest of the GNU Radio flow, the FPGA is a static component that does not participate in the same dynamic flexibility as the data streaming through the Python flow graph components. The FPGA is statically configured with an assortment of parameterized components. As a result, the principle agility of GNU Radio comes from software alone.

3.1.1 Streaming Dataflow and Scheduling

At the top-level, the modules and flow graph are controlled by a simple scheduling mechanism that samples and executes based on current dataflow needs. Each module that is connected in the flow graph is joined with a buffer for moving data between the processing units. Each module has I/O requirements that need to be met before the scheduler will allow a module to execute. Each module is required to have a specific number of output items once a computation is complete and this requirement passes upstream to the input of the modules. The scheduler checks the number of input items and if they meet the requirements for execution the scheduler will call a special `work()` function that contains the calculations for that module. The `work()` function will be discussed more in Section 5.4.

The implementation of signal processing modules at the python level are built around the idea of streaming dataflow. The scheduler works on a producer consumer type of

model where it inspects the input and output requirements of each module and makes decisions based on these parameters. Modules can only communicate in one direction with one other module. In this model the scheduler iterates over all blocks in the flow graph and executes based on input and output requirements only. It becomes easy to see how this simplicity might cause issues if higher level control is desired. In Section 2.4 it was shown that MAC layer development has suffered in GNU Radio and this idea is one of the main reasons.

3.1.2 Modular Component Design

The ability to quickly script radio designs by use of a flow graph comes from the concept of modularity. Modularity is one of the key design elements that make GNU Radio easy to use and powerful. Each of the nodes in the flow graph represents a signal processing module that performs some computational task. A large library of preexisting components are available for use, but if a specific function is needed and does not exist the designer will be responsible for creating the modules they need.

GNU Radio comes with a basic template for designing components that can be found in their tutorial [30]. For basic signal processing the designer is mostly concerned with the C++ layer and efficiency. The functionality defined at the C++ layer will be put into the `work()` function that is called when the block is used in a flow graph.

Figure 3.2 shows a simple Python script for the "hello world" of GNU Radio. This code takes two sine waves of different frequencies and connects them to the left and right channels of an audio out. The concept of modularity is shown in lines 13, 14, and 15. If the designer wants to change the behavior of the tone, a simple parameter modification will suffice. If the designer wants to plot the waveforms they can take the signal points and pass them to a file sink module instead of the audio sink. Lines 16 and 17 show how the flow graph connects by linking each waveform to the desired audio channel.

```
 1 #!/usr/bin/env python
 2
 3 from gnuradio import gr
 4 from gnuradio import audio
 5
 6 class my_top_block(gr.top_block):
 7     def __init__(self):
 8         gr.top_block.__init__(self)
 9
10         sample_rate = 32000
11         ampl = 0.1
12
13         src0 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 350, ampl)
14         src1 = gr.sig_source_f (sample_rate, gr.GR_SIN_WAVE, 440, ampl)
15         dst = audio.sink (sample_rate, "")
16         self.connect (src0, (dst, 0))
17         self.connect (src1, (dst, 1))
18
19 if __name__ == '__main__':
20     try:
21         my_top_block().run()
22     except KeyboardInterrupt:
23         pass
```

Figure 3.2: A simple example of a GNU Radio script.

Module design in hardware is very similar to the GNU Radio model as well as standard digital design. The designer will write and test the hardware model that is desired, but the difference comes when generating a bitstream. The standard tool-flow produces a full bitstream for programming the entire FPGA device. When designing modules the designer will have to use the Partial Reconfiguration (PR) flow to generate a partial bitstream for only modifying a specific portion of the FPGA. More on the PR flow will be discussed in the AgileHW section 4. The use of hardware modules is one of the main improvements proposed for the expanded GNU Radio.

3.2 Improvements

3.2.1 Expanding GNU Radio with Hardware

The productivity of the GNU Radio environment is characterized by the ability to quickly design radios with the use of existing library components. The library components are software defined and assembled in a top-level Python script. Each component is defined in C++ for efficiency and linked to Python for instantiation in the top-level. This model makes use of the efficiency of C++ as well as the dynamic scripting capabilities of Python. The resulting environment is excellent for simple radios; yet leaves something to be desired for high-speed designs.

The USRP2 contains a Xilinx Spartan 3 FPGA that is used mainly for handing communication with hardware components on the USRP2 and Ethernet packets from the host machine. The on-board FPGA would allow signal processing to be offloaded from the host machine improving performance. Currently, this idea cannot be fully realized due to the limited capabilities of the Spartan 3. A larger external FPGA would allow the opportunity to fully execute the tasks required by a high speed radio. The aim is to allow more flexibility to GNU Radio by making the FPGA a more prominent player. This is accomplished with a modified software system that integrates well with current methods, and an auxiliary FGPA (A-FPGA) for the acceleration of digital signal processing. A software block representing the A-FPGA controls dataflow within the software environment while the physical A-FPGA offloads the software signal processing without impacting agility.

The hardware system running on the A-FPGA will ultimately be a slightly modified version of the AgileHW work, which will be discussed further in Chapter 4. GNU Radio and AgileHW mesh well because both environments are based on modularity. Figure 3.3 shows the one to one relationship between designing a module in software using C++ and Python, and designing a module in hardware with VHDL or Verilog. The first step

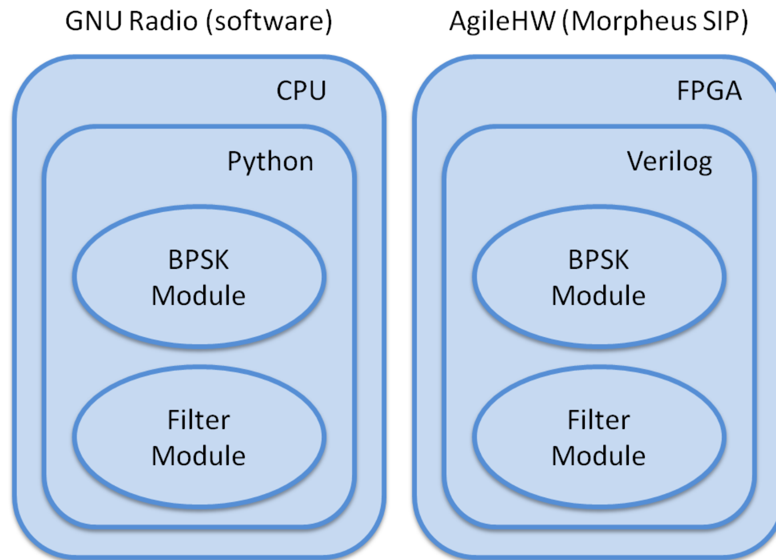

Figure 3.3: A comparison of modularity in software and hardware.

for this solution is to design a robust framework that supports both GNU Radio and AgileHW.

The solution is intended to be transparent to a GNU Radio programmer. The mixture of hardware and software components will provide additional power and agility in radio design, yet the proposed solution will not force any additional difficulty on a user. Chapter 5 will provide a window into the implementation details for the purpose of explaining what is really happening, yet these details will be unnecessary for a GNU Radio user. The main focus of this work is to target a mixture of hardware and software components with a single high-level software description. This solution will improve GNU Radio while maintaining the goals of AgileHW.

3.2.2 Accessability of AgileHW

The current state of AgileHW has the entire system running on the Harris Morpheus SIP. This board has four Virtex-4 LX60T FPGAs and an ARM core processor for software and FPGA control. It is a powerful platform yet the latest FPGAs have modified architectures

that partial reconfiguration depend heavily on. Continued use of this platform forces all efforts at improvement down a road that is ending.

The GNU Radio project is an ideal platform for reaching the goals of this work and the benefit of its use with AgileHW are two-fold. First, GNU Radio can benefit from AgileHW by gaining a more robust feature set without abandoning the principles of their work. Secondly, partial reconfiguration has many merits yet its acceptance has been minimal in industry. For it to gain traction significantly more interest needs to be shown. By generalizing the hardware platform used in AgileHW and overlaying it with a popular and open source GNU Radio, this technology can be shown useful to a much wider audience.

Chapter 4

Agile HW

This chapter introduces and details the AgileHW project (from the background Section 4). The AgileHW project has been one of the main focal points for the Configurable Computing Machines Lab at Virginia Tech. The core idea of AgileHW is to use FPGAs in a non-traditional fashion for increased agility and productivity. This is done with the use of partial reconfiguration (PR), which is a special modification of the standard tool-flow that supports configuration of select partial regions on an FPGA. Using PR, with some modifications, gives increased agility and is the main idea that AgileHW builds on.

Concepts within GNU Radio, like modularity, coincide with the idea of PR. Modular functionality in a PR environment can provide a dynamic system similar to the standard GNU Radio use model. GNU Radio has flexibility by using software and scripting capabilities, while a PR system gives similar flexibility to FPGAs with the ability to quickly swap out modular components.

A module is a set of similar data processing steps, or techniques, that fit well into a single unit of computation. This definition matches with both GNU Radio and AgileHW making it easy to mesh the two ideas. The AgileHW environment has a large dynamic region supporting the placement of modules during run time, and an overlaying static

system for interaction with the dynamic region and any external device.

The use of AgileHW is split into two modes. The first is a compile-time flow that builds the static portions of the system, including the large dynamic region wrapper and interfacing logic. The second is the run-time flow that is responsible for communicating to the static design what modules are to be placed and how to link together multiple modules.

The original platform for AgileHW was the Harris Morpheus SIP. The SIP contained an ARM core for software processing and four Xilinx Virtex-4 FPGAs for hardware acceleration. Changes to both the run-time and compile-time flows needed to be made to move AgileHW into the GNU Radio environment. Some of these challenges are related to platform differences and others are related to FPGA manufacturing differences. A description of the AgileHW model, the challenges, and modifications are detailed in the sections below.

4.1 Compile-time

Compile time is the first step in the Agile Hardware flow. It is responsible for establishing communication between the static and dynamic region, and creating partial modules. The static region includes the clocking, I/O, interface support, module support, and placement of the dynamic region wrapper. The other responsibility of compile time is to build the partial modules that act as the building blocks for a radio processing chain. Xilinx provides a basic environment for PR, but a custom tool-flow can add extra flexibility.

The Xilinx Partial Reconfiguration Early Access Software Tools for ISE 9.2i (here on referred to as the Xilinx PR Toolkit) has been a useful contribution to the reconfigurable computing community, and has served as the basis for many research projects. This toolkit, while not formally released with the ISE toolkit, was intended to augment the

modular design flow. The underlying principle is that the toolkit enables slot-based partial reconfiguration. A "slot" in this case is a fixed geometric region – fixed in size and fixed in position for a selected device. Multiple partial designs can be created with this toolkit that can one-by-one populate a given slot.

For the domain of SDR, it is easy to show why a slot-based partial reconfiguration model is not desired. The slot locations and sizes are allocated at design time, which make them a compile-time constant. This means if there is a need to add an additional module, an FFT for instance, there may not be an available slot based on the current design. The size of the slots are also fixed, which can lead to wasted space if the modules allocated to the region exhibit a high degree of footprint disparity. A collection of filters designed for placement in a given slot where one filter is substantially larger than any other will make the targeted slot an inefficient use of FPGA fabric the majority of the time. Finally, the I/O ports on these regions are static, which force unnecessary constraints on the modules desired for a particular region while also hurting flexibility by disallowing a module to relocate to different slots.

The solution is to deviate from the standard Xilinx PR flow, and instead create a large reconfigurable region to house multiple modules where pre-fabricated modules are relocated as needed in the region. In addition, the interconnectivity for these newly placed modules is performed in a fraction of a second outside of the ISE tool suite, eliminating the costly PAR process. In the context of the work presented here, there are two regions where both the transmit and receive chains have dedicated regions for incoming or outgoing radio processing. Each region will need the ability to dynamically remove specific radio modules to maintain the productivity presented in GNU Radio. The AgileHW runtime environment allows for on-the-fly modifications to the reconfigurable regions giving similar granularity to that of software. Each of the modules contain a wrapper structure that provides anchor points for routing. Modules are relocated and placed based on an algorithm aimed at efficient use of the reconfigurable region and reducing routing delays between modules. The router used for module connections is both lightweight in terms

of memory usage and fast in execution. The set of tools presented will allow the design to break away from the vendor tools giving significant gains in productivity.

4.1.1 Current Static Design

As stated, the current implementation of AgileHW is running on the Harris Morpheus SIP. This device requires a special static region design to interface between the FPGA logic and software running on the ARM core. The SIP has a CAD bus for this communication. Commands and data can be sent over this bus using a simple addressing scheme. This type of communication is unique to the SIP, and shows how moving from this platform will require changes. The static region is also responsible for tasks other than communication.

The static region of the FPGA is split into two sections: static logic for interfacing with the cad bus and dynamic region, and an empty dynamic region wrapper where the modules will be placed. The static logic of the FPGA also has clock management functions, data I/O modules, ICAP logic, and bus macros.

- Clock Management : The modules in this section use the FPGA clock or external clock and modulate it with DCMs (digital clock modulator) to generate the desired frequency needed for the radios.
- Data I/O : Modules such as the CAD bus, used for local communications; FIFOs, to process data; and other functions that are necessary to properly create the data packets that are transmitted and received by the radios.
- ICAP : This is an integral part of the reconfiguration process. Partial bitstreams are loaded into the ICAP which feeds the reconfiguration logic that controls the PR region of the FPGA.
- Bus Macros : These Xilinx cores act as the connection between the dynamic and

static regions of the design. Bus macros ensure that signals crossing the PR region are consistent with both the static and dynamic regions.

4.1.2 Static Design using GNU Radio

Items like the CAD bus, ICAP logic, and bus macros are specific to the SIP and Virtex-4 FPGAs. In order to move the AgileHW system into GNU Radio and add support for Virtex-5, some changes need to be made. One of the main differences is the way communication between software and hardware is done. Reconfiguration is still done via the ICAP, but there are some differences in ICAP control. The dynamic region also presents new challenges for Virtex-5.

Ethernet

The Morpheus platform uses a CAD bus system for communicating between the arm processor and Virtex 4 FPGAs. The CAD bus defines how data flows and how reconfiguration happens. With GNU Radio the communication will change to Ethernet. The FPGA is now physically separated from the host GPP and uses Ethernet to transfer data.

Ethernet communication differs from the CAD bus for the internal logic in the way incoming data or commands are interpreted. For the radios on Morpheus we specify writes to certain addresses as commands. For the A-FPGA there is no notion of a bus address and each command needs to have a method for implementation. This is an easy fix with the type field described in Section 5.3. Each command, such as adding a module, resetting the modules, and resetting the ICAP all can have their own type instead of an associated address. Data transfers are trivial to port and just require proper directing of traffic.

ICAP

The new ICAP glue logic needs to accept Ethernet frames instead of CAD bus data. A decoding scheme will be detailed in Section 5.5.2, but this is one of the key changes for using Ethernet frames. The ICAP control is still relatively simple and requires control over enable lines and byte reversal of the incoming data.

4.1.3 Dynamic Region

Half of the static design is related to building the dynamic region wrapper and how modules are created for insertion to the dynamic region. The placement of the dynamic region is relative to a particular instance of an FPGA. Also, partial modules need to be designed in a special way to support proper usage in the dynamic region.

Placement

Placement of the dynamic reconfiguration region is now the only residual from the original slot-based model. The location of the region is determined based on the layout of the special resources in a given FPGA (DSP48's, BRAM's, etc.). It is not necessary to limit the user to a specific Xilinx FPGA within a family, yet all have slightly different resource layouts. The resolution is to automate this process with the use of the Xilinx PlanAhead floorplanning tool, which has knowledge of many FPGA layouts even throughout different families. The layouts are stored in XML files, which can be parsed for information relative to the task of floorplanning. This automation ensures the user will not be required to have in-depth knowledge of the FPGA of their choosing. This tool will only be used for determining the placement of the large reconfiguration regions and does not play a roll in the run-time aspects of library component placement and routing.

Partial Module Creation

The partial modules make up the parts of the radio transmitter and receiver. These modules are placed in the dynamic region and connected on-the-fly to create the radios. In this section, the tool flow to build the partial modules, and the dataflow of the modules in the dynamic region will be described. An illustration of the tool-flow can be seen in Figure 4.1.

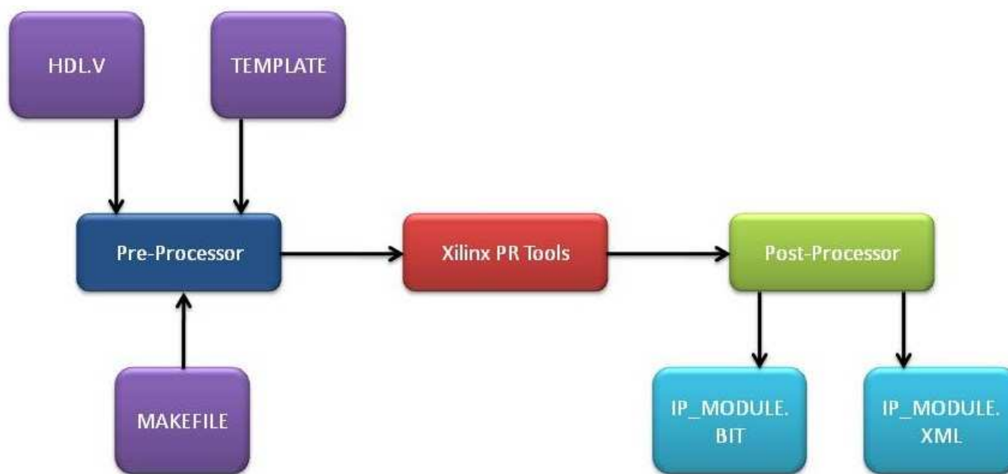

Figure 4.1: Illustration of AgileHW tool-flow including pre and post processing.

There is a set structure to build the partial modules in the Agile Hardware flow. The partial module tools accept two different types of files, HDL files that describe the module's logic and a template file that defines the module's size and port information. The HDL files consist of user defined logic for radio specific functions like filtering, interpolation, scrambling, descrambling, and any other functionality deemed necessary. The template file defines the modules port information such as the type, size, direction, and name. The template file also defines the module's size so it can have adequate resources to incorporate all the logic.

Once the HDL and template files are prepared, it is passed to the pre-processor with a Makefile. This pre-processor ensures the template file and module names correspond,

parses the template file to set up the module boundaries and resources. This is then passed to the Xilinx PR tools which compile the HDL files and place and route the design on the FPGA. Finally, the post-processor receives the compiled HDL and template file information, preps the module, and places it in the assigned coordinates of the FPGA. It also generates an XML file that is used by the run-time tools to understand the module size and port information. At the end of the tool-flow, the user receives a partial bitstream and an XML file with the module port definitions to be used by the run-time tools.

Inter-module Communication

After the partial modules are built, the run-time flow connects them in the dynamic region. The current dataflow in Agile Hardware is north and south, meaning that data flows vertically across the stacked modules. The run-time tools use the XML files created by the partial module tool-flow to connect the modules to each other. The first module in the sandbox receives its inputs from the sandbox. When the next module is placed in the chain, the outputs of the first module are connected to the input of the second module. The output of the final module is then connected to the output of the sandbox. The tools identify which ports are connected by the port type: data, reset, ready. So a output port of type data is connected to and input port of type data. The clock is not passed through the modules because the whole sandbox is clocked by the static logic.

4.2 Run-time

The previous section discussed how the partial modules were created, and briefly introduced on-the-fly routing. When a partial module is placed in the dynamic region, it needs to be stitched with the static bitstream so the static and dynamic logic work together. This section will discuss the run-time flow, the second tier of AgileHW.

4.2.1 Resource Allocation

When the partial module is designed and built, it has specific resource requirements such as DSPs, BRAMs and CLBs. The router ensures that when the partial module is placed in its assigned location, all the links with the appropriate FPGA sites are connected properly .

4.2.2 Place and Route

After the partial module is placed in the dynamic region, the router connects the data input nets with the outputs of the module above and the data output nets with the sandbox output. It also routes the clock line used by the sandbox to the logic, so all the modules are operating at the correct frequency. The channel routing scheme, used by the router to connect the different nets and resources, is very similar to the Left-Edge Algorithm. The router connects the lowest number pin starting from the left to its right most equivalent.

4.2.3 Systembits

Systembits is the penultimate piece of the Agile Hardware run-time flow. It is responsible for the on-the-fly bitstream manipulation that makes the design dynamic. Systembits is responsible for placing and removing configuration data in the system, adding and removing new routing in the bitstream, and modifying designated LUTs and PIPs. The current version of systembits is capable of performing these functions on Virtex 2, Virtex 2 Pro, and Virtex 4 FPGAs. Information about Virtex 5 and future FPGAs are being added to the systembits database to make it available across multiple platforms. A breakdown of the systembits tool flow is described in the below subsections.

Device & Architecture Query

The first task at hand is to identify which device systembits is modifying. The device information is used by systembits to identify the architectural changes in the device, the locations of said sites, and other information that systembits needs that are specific to each device.

Base Tasks

Once the device being manipulated has been determined, systembits parses the base bitstream consisting of the static region. It also checks the PIP and LUT database for the site information and functions.

Place and Remove Configuration Data

After systembits has acquired the information for the device, it can add and remove the user specified modules. The tools logically 'OR' the partial and base bitstream to add the module information to the system. There are rules in place to check that the module is within bounds and the resources are available. When the tools need to remove a placed module in the bitstream, they use a mask operation to write over the designated area.

Apply Modifications

When the partial module is added to the base bitstream, there are several PIPs and LUTs that change in the module's designated location. Once the sites are modified, new routes to the base bitstream need to be established. A list of all the modified sites is created and passed to the router.

Add and Remove Routes

The router receives the modified LUT and PIP database information and uses a left edge algorithm to connect them in the bitstream. The tools iterate over each of the sinks and sources to tie them together using the LUTs and PIPs. When the tools need to remove the routes, they iterate over the corresponding LUTs and PIPs and simply turn the sites off.

The AgileHW project is an environment well suited for the design of radios and SDR. An SDR system using software modulation provides productivity and agility at the sacrifice of computational complexity. AgileHW gives the designer the choice to offload signal processing tasks to hardware while retaining productivity and agility. Another core design element of AgileHW is modularity of components. Modularity aids in ease of use and extensibility. This philosophy is also used by GNU Radio making for a suitable paring of technologies. The implementation of such a system is not trivial, but as long as the design is done properly the details can be hidden from the user of said system in order to maintain ease of use.

Chapter 5

Implementation

This chapter will detail how the enhanced GNU Radio framework was implemented, starting with a system-level abstraction down to communication primitives. As mentioned, the details will be abstracted from the user, but are important for understanding how changes were made and why.

5.1 System Usage

A description of how a designer would use the system can be seen in Figure 5.1. The first set of commands are related to the USRP2 and how to set-up the front-end communication parameters. For instance, the `self.u.set_interp(zigbee_interp)` command is used for setting the interpolation of the radio stream. Python scripts follow the standard procedural model, so these commands will be called first. Once the USRP2 is set up the A-FPGA can be configured.

The next set of commands are responsible for adding signal processing modules into the A-FPGA. The `self.a.add_module("add",bit_to_sym,0)` command is used to add a selected module into the dynamic region of the A-FPGA. These commands dictate the

datapath for signal processing. A detailed description of this process will be given in Section 5.4.3.

Finally, the flow graph is connected to set up the dataflow and execution can begin. The GNU Radio scheduler will manage dataflow between the modules connected. In this case, it would be between the A-FPGA and a file. It is important to remember and understand this figure when reading through the remainder of the chapter as it depicts the main goals of a scripted and modular environment.

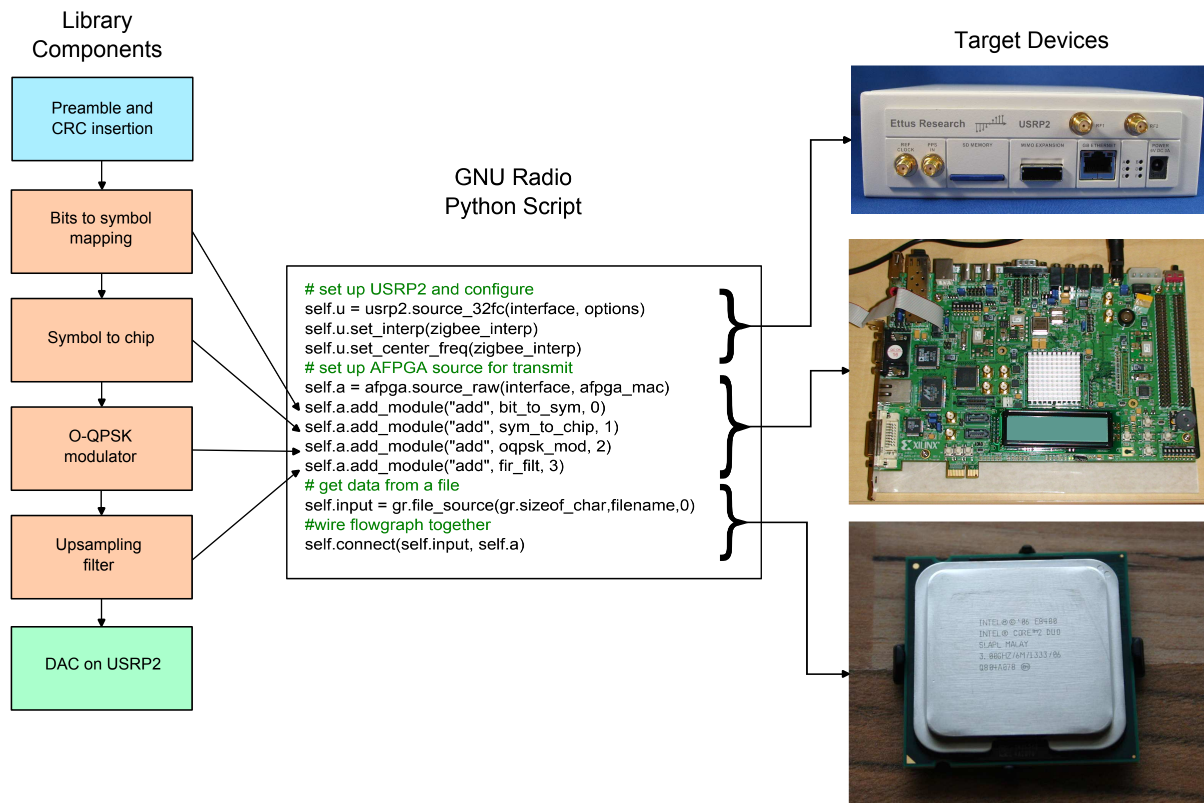

Figure 5.1: What a Python script for GNU Radio looks like, the devices that each code section targets, and how a datapath of radio modules correspond to Python code.

5.2 Hardware Layout

Within the GNU Radio framework, dispatching tasks to external components adds complications to system-level communications. The idea is to not hinder the existing GNU Radio dataflow while allowing hardware acceleration if the user chooses to do so.

The proposed solution, shown in Figure 5.2, now has three logical links to consider for communication between devices. There are three bi-directional links that separate functionality and communication. There is a link between the host machine and the A-FPGA (Data Channel) for transmission and reception of raw radio data, and for sending configuration packets to the A-FPGA for radio modification. The second link is between the A-FPGA and the USRP2 (Sample Channel) and is strictly a data link for sending and receiving modulated data. The final link is between the host machine and USRP2 (Command Channel) and supports current GNU Radio functionality as well as provides a link for configuring and controlling the USRP2.

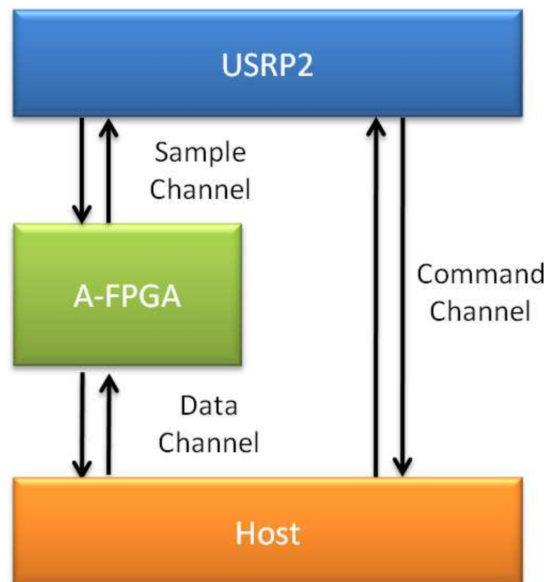

Figure 5.2: Shows the logical connection of hardware platforms.

It is important to note that the communication links described are logical and not physical. For understanding the dataflow of the modified architecture it is important to view the data links in this way. The actual physical connections consist simply of a gigabit Ethernet switch that receives and forwards packets between the three nodes of the system (USRP2, A-FPGA, and host). The way the devices are separated logically is with proper packet formation and addressing schemes. The packet formation and addressing is unique to each of the logical channels.

5.3 Data Packet Format

The data packet organization, shown in Figure 5.3, is an important design decision that ensures compatibility with future designs and ideas. The Type C packet contains the USRP2 header, which is already defined by GNU Radio, and remains the same for direct communication between the host and USRP2 (Command Channel). These commands are related to configuration and use of the USRP2. Each command has an operation identifier and associated data for meaningful communication. The host machine is responsible for issuing these commands that are made based on the Python code system description written by the user.

The Type B packet is also defined by GNU Radio and is used for radio sample transmission and reception. It is used as the link between the USRP2 and the A-FPGA (Sample Channel) that connects the radio modules to the radio front-end. As the USRP2 is already designed to handle radio samples, the A-FPGA sends sample packets in the accepted format.

The changes in packet formation occur for packets of Type A that are sent to the A-FPGA by the host machine (Data Channel). The addition of a type, size, and CRC fields as well as the elimination of the USRP2 header are the necessary changes. The USRP2 header contained the Ethernet header inside of it (source and destination MAC addresses, and

Ethernet type). The removal of the USRP2 header leaves only the standard Ethernet header behind.

As for the logic behind the Type A packet, the type field is necessary because the A-FGPA needs to know if the incoming packet is a configuration packet or a data packet. If the packet is a configuration packet then the hardware logic forwards the payload data to the ICAP for reconfiguration. If the packet is a data packet then the size field informs the radio of the number of samples incoming. The CRC is used to inform the receiver if the incoming packet has sustained an error during transmission. Packets sent in the reverse direction, from A-FPGA to host, are only radio data (text, audio, video) that has been demodulated. The CRC is also sent back to the host along with the radio data to inform the host if an error has occurred. Thus, the type and size field can be ignored for incoming packets.

Figure 5.3: Shows the packet layouts for each type of packet.

5.4 Software Layout

The software written for GNU Radio follows a set of well defined and maintained standards. The coding standards and guidelines [31] are laid out so that modifications and improvements to GNU Radio will be uniform. Some of the coding styles include ways in which variables are named and comments are made.

Not only does the code follow certain style standards but it also has some unique features in terms of layout and usage. One key feature is the use of Smart Pointers [32] from the Boost libraries. Smart Pointers provide dynamic memory allocation where deallocation is taken care of, much like garbage collection in Java. The use of Smart Pointers is also abstracted in a way to prevent any programmer from allocating GNU Radio class objects from C++ with standard pointers. This is done with the use of a `make()` function that constructs the object and returns a Smart Pointer to the caller. This `make()` function is able to be called publicly while having access to the constructor of the class with the C++ "friend" property.

These are a few of the key concepts that were followed in the changes and improvements made. It was necessary to follow the style and coding standards so any person in the future that wants to make changes will understand very easily what the code is doing. The following sections will go into greater depth about how the changes to the software level of GNU Radio were made.

5.4.1 Python Modules

At the highest abstraction level is a Python script that declares the radio modules and defines their behavior, as shown in Figure 5.4. Lines of code requesting baseband frequencies, gain, and interpolation parameters to the USRP2 are written at this level. To maintain the current user interface to GNU Radio, the additions made are written to be compatible. All of the additional functionality and code is executed at lower abstraction

levels while the top-level remains the same. The change to the top-level consists of the inclusion of a new module, as there would be for any new software module written using the standard flow. This is the level of abstraction where the GNU Radio software flow is first tapped into and data is redirected for the enhanced framework.

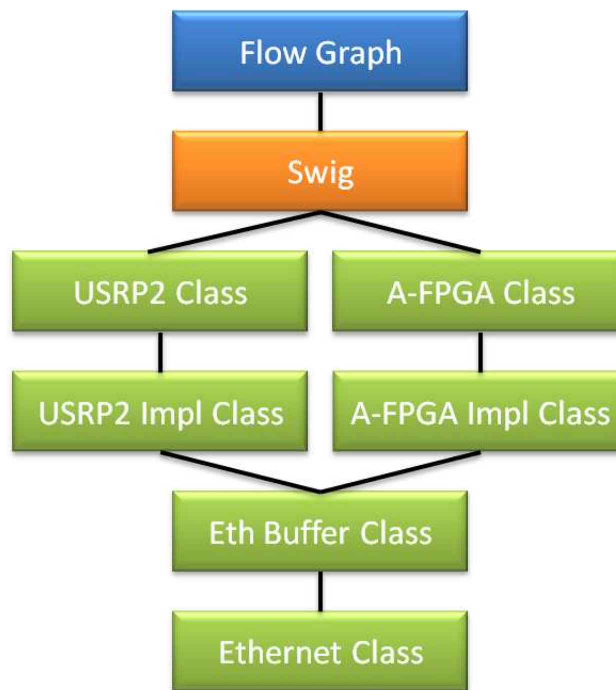

Figure 5.4: Layout of class hierarchy for software chain.

The additional module, which provides the redirection, is appropriately be named the A-FPGA module. Declaring this module presents the designer with access to all of the underlying functionality with the use function calls. At this level, things remain simple and easy to use as the user is only concerned with how to add radio modules and how to send and receive data. The set of functions at this level include `add_module()` and `run()` to complete these tasks. The user declares the A-FPGA object, defines which radio modules to add, inserts the A-FPGA module tag into the flow graph hierarchy, and then uses textttrun() or texttttstart() to start dataflow.

5.4.2 Module Implementation

Taking a step down in abstraction presents the glue logic that ties the upper level Python script to its lower level C++ implementation. The details of how the module is presented to the user and what lower-level functionality is used are stitched together at this level. The code is written in C++ and the module is built into a class object that SWIG [33] uses to translate into Python.

SWIG is an open source project that is used for connecting programs written in C or C++ to be connected to higher level languages like, in this case, Python. This is what allows for the ability to have pre-compiled C++ signal processing blocks that are called from the Python environment.

Each module that is called from Python has a C++ class that defines its behavior. Several boilerplate functions need to be defined to allow the module to be of use in Python. The main function of interest is the `work()` function. This function defines what code will be executed when the module is in run mode (doing whatever it is the module is intended to do). This is the function that is called when the commands mentioned above, such as `start()` or `run()` are called.

The code in the work function is generally at the top of a complex, yet well defined, C++ class hierarchy that is built to carry out many efficient tasks. For instance, the call to the constructor for the A-FPGA module calls the constructors of the A-FPGA implementation class, the Ethernet buffers class, and Ethernet implementation class. These are all classes that are required to form and distribute packets for communication.

Function members for the A-FPGA class include the `add_module()` function from the top-level. This function takes a command such as add or remove, the radio module in question, and the index for its position with respect to other radio modules. This information is used to build an Ethernet packet that is sent to the A-FPGA where the request is carried out. The functionality below this level, which is concerned with

physically sending and receiving an Ethernet frame, is already defined by GNU Radio and also used in the implementation of the A-FPGA module.

5.4.3 Low-level C++

Packet construction is at the lowest level directly above the Ethernet calls. The `add_module()` function calls a GNU radio function called `tx_raw()` that passes along with it data from the `tx_metadata` class. The `tx_metadata` class is responsible for providing information about the packet necessary. It carries information such as configuration flags, module numbers, and commands.

The nice thing about this code hierarchy is that the lower-level Ethernet code does not need to be aware of this while the upper level calling functions remain easy to use. This is demonstrated in the way that the same function, `tx_raw()`, sends configuration packets after a call to `add_module()` as well as sends and receives data packets from the `work()` function. This `tx_raw()` function exists in the implementation class and uses metadata that is passed as a parameter to construct packets or to inform the receiving code what it should expect. The metadata can use its type member, which corresponds with the data packet structure described earlier, to inform the send and receive functions what type of packet it needs to construct or should be expecting to receive. The concepts at this level conclude the redirection and show how re-entry into the the GNU Radio software flow occurs.

5.5 Hardware Modules

The hardware modules were created using Verilog and the Xilinx version 11.3 tools. Some modules including the Ethernet implementation and FIFOs used Xilinx Coregen for producing the necessary Verilog and .ngc files. The description of the modules used in the A-FPGA will work from the outside-in.

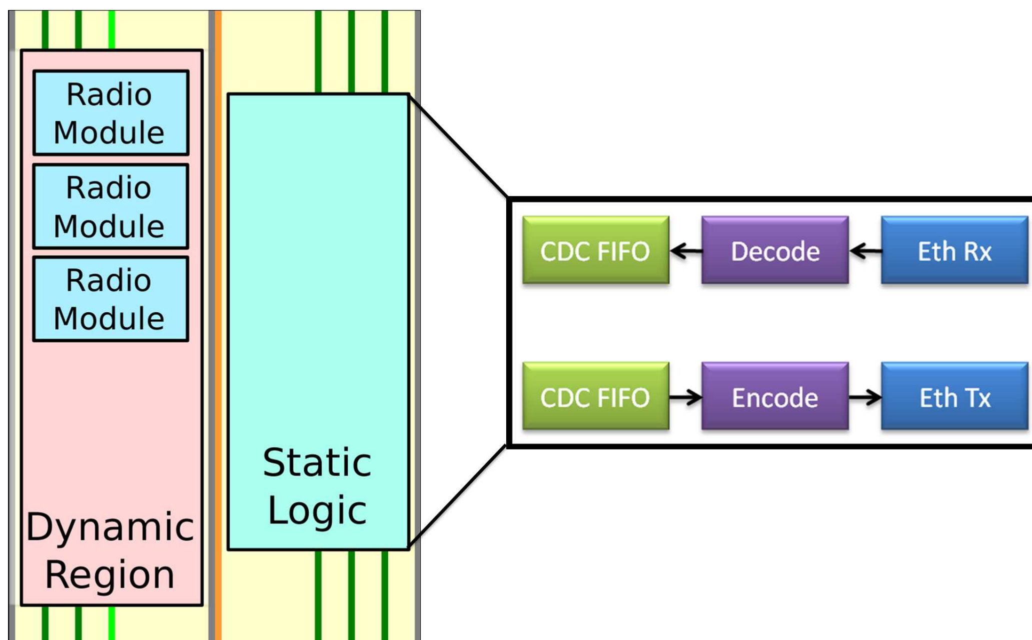

Figure 5.5: Datapath through hardware modules.

5.5.1 Ethernet Module

Any packet that is inbound to the A-FPGA is first seen by the Ethernet MAC module. A wrapper is provided by Xilinx Coregen that includes a link-layer protocol as well as simple flow control using FIFOs. This presents an interface to easily build the interior logic upon. The Ethernet MAC wrapper is bi-directional and has support for sending outgoing packets using the same interface. From this, we can deduce that this module is also the last thing an outgoing packet will encounter. An overall layout of the system

can be seen in Figure 5.6. Any properly formatted packet is forwarded for inspection by interior logic.

The interface for the Ethernet MAC wrapper is a link-layer protocol that contains a start-of-frame and end-of-frame signal as well as source and destination ready signals. This gives control over starting or stopping incoming and outgoing frames and knowledge of frame boundaries. Once the start-of-frame signal is pulsed, the data are streamed in or out through an 8-bit wide interface, one byte at a time, until the end-of-frame signal is pulsed. The interior logic can pause this communication at any time by de-asserting its ready line. To analyze the interior logic from this point, the case of sending a sample through the transmit lane will be used.

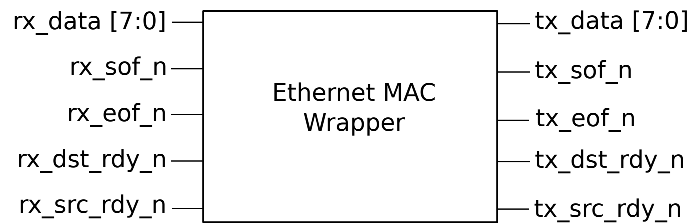

Figure 5.6: Shows input and output ports of Ethernet Module

5.5.2 Radio Transmission

Receiving a Radio Frame

It is necessary to decode the incoming packet by means of parsing the header information present in the Ethernet frame. The decoding is done with a module containing a dual-ported RAM as well as a controlling state machine. The incoming words are sampled and then written into the dual-port RAM. Once the information is decoded a valid signal is pulsed letting the next level of logic know that data is ready. The payload data is then streamed out of the RAM and directed based on the results of the decoding. The Ethernet header is no longer needed at the next level of logic so only the size word and

data payload are streamed out of the decoder module. The decoder module also handles the flow control parameters for the Ethernet MAC. If the internal FIFO becomes full the decoder will de-assert the destination ready signal telling the Ethernet module to pause.

A dual-port RAM was used because of its ability to both read and write on every clock cycle. Once the header information has been streamed into the RAM, a single clock cycle is used to decode the parameters that were sampled on reception. The RAM block also provides a persistent storage of subsequent incoming payload elements that would otherwise require the halting of the Ethernet MAC during the decode state. This solution provides minimal delay for decoding and supports a near constant influx of data from the Ethernet MAC wrapper.

Decoding

An Ethernet header gives information about where the packet has come from and provides useful information from directing data. On the receive interface, the SRC address is what provides this information. The decoder informs the next level circuitry on how to proceed with the frame via a flag and ready system. A valid flag is asserted when the data exiting the decoder is valid and a 4-bit word informs what action should be taken. On the receive end the possibilities are as follows: an incoming raw data packet that contains radio data from the host machine that needs to be forwarded to the transmit radio, incoming radio data from the USRP2 that needs to be forwarded to the receiver radio for demodulation, or a configuration packet from the host that contains a bitstream for reconfiguration.

FIFOs

The data coming out of the decoder is forwarded to a specific FIFO that buffers data between the decoder and the intended radio. The TX and RX radios both contain their own individual FIFO whose main purpose is crossing clock domains between the

Ethernet MAC and on-board radio. Currently, the USRP2 ADC and DAC use a 100MHz reference clock that puts some restrictions on the clock frequency of the radio modules. The modules are clocked at a multiple of the 100MHz reference so the samples will line up with the sampling rate once they reach USRP2.

Encoding

Once the data has exited the radio, it is fully modulated and in the same form as it would be leaving the host machine in the standard GNU Radio flow. From here, the data needs to be framed, and sent to the USRP2 in the format required by the firmware on the USRP2. There is a specific header format that is defined and includes information such as sequence numbers and timestamps. The required header information is easily appended to data leaving the radio with the use of a state machine. An encoder module is used to retrieve data from the transmitting clock domain crossing FIFO. The encoder is either in a state where it is constructing the header and pausing the incoming FIFO data, or constructing the payload where it streams the incoming data from the FIFO. This module also contains the logic for controlling the Ethernet MAC for sample transmission to the USRP2 or host.

The dataflow design works in both directions regardless of the radio design in the middle. Looking at the receive lane where data is flowing from the USRP2 on route to the host machine serves as a good example. The data enters the Ethernet MAC receive lane and is decoded as before, but this time it is passed to the transmit radio's FIFO instead of the receiver radio. After it has been demodulated back into raw data it is framed and transmitted to the host machine for processing. This concept is necessary because the radio region can change while the framework must continue to work around these changes. The framework was designed in a manner to not only support these changes, but also to support AgileHW.

5.6 Challenges

Many challenges arose throughout the design and implementation process. This section outlines a few of the major hurdles encountered, and how they were overcome.

5.6.1 Hardware Connectivity

There are a few interfaces to each device being used in this system, which gave many possibilities for interconnections. The USRP2 has both GigE and mini-SAS interfaces, the Virtex5 has GigE, PCI-E, and SATA, and the host has GigE and PCI-E. The commonality between these devices is GigE, which has a substantial throughput limit and is very well supported. The mini-SAS connector on the USRP2 has the ability to break out into 4 lanes of SATA. One of these lanes can be used to hook up to the Virtex5 SATA ports. Finally the Virtex5 board could also connect to the host machine via PCI-E. There are pros and cons to each of these solutions.

All of these solutions can handle large amounts of bandwidth and there differences really only arise in latency and implementation. If bandwidth is the sole concern then gigabit Ethernet with 1Gbps is easiest to implement, has the most support, and is available on all of the suggested devices. If latency is the main concern then gigabit Ethernet has some minor issues, which will be discussed in Section 6.3.1.

In a low-latency environment, the ideal setup would be to have a PCIe link between the host and A-FPGA for transferring data to the accelerator. A high-speed serial link or on board daughterboard would be ideal for communication between radio front-end and A-FPGA. This setup is similar to what SORA (Section 2.1) used and they were able to achieve low enough latency for IEEE802.11 compliance. One of the main issues with this setup for GNU Radio is that it is difficult to find the proper cable. The cable needed is a external mini-SAS to quad breakout host SATA cable, which is rare. For an open source system trying to get wide acceptance, this cable would be a hindrance. If it were offered

as a standard part from Ettus LLC., like the MIMO cable, it might be easier to accept.

5.6.2 Dataflow

The challenges related to dataflow were mostly related to the receive datapath as the transmit datapath is straightforward. The only change that was made to the transmit datapath was to use the MAC address of the A-FGPA as the target instead of the USRP2. The receive datapath is more complicated because the lower-level functions were designed around the idea of a single external hardware device. There were two items in particular that provided a challenge in adding a secondary hardware platform – redirecting the receive chain and packet filtering.

The USRP2 is set up to send data or command responses to the interface and MAC address it received the packets from. So, when a command is sent from the host to request samples from the receive line, the samples are sent back to the host machine where they came from. This is not the desired behavior as now the data samples should be sent to the A-FPGA for demodulation. This was resolved with the creation of a new command that contained the MAC address of the A-FPGA. The user will specify the MAC address of the A-FGPA in the creation of the module in the top-level Python script so the MAC address can be used downstream. The command is the `start_rx_stream_afpga` command and is used to inform the USRP2 to send samples to the MAC address specified in the command packet.

The second challenge was related to the receive interface on the host machine. There exists a set of driver-level filters for reducing the number of packets forwarded from kernel space to user space. This helps with performance of the user code and is a good improvement for the existing GNU Radio flow, but provides problems for this augmented system. It now requires packets to be received from two different devices – command responses from the USRP2 and demodulated data samples from the A-FPGA. The filter was removed to allow for incoming packets on two interfaces.

5.6.3 Static vs. Dynamic Run-time

Scripting radios with Python makes a static environment in the sense that once you start the script, there is no way to modify radio components. The way this is dealt with in AgileHW is to use a client/server model. The server waits for commands to process from the client and can run indefinitely.

This same idea can move into GNU Radio due to the design of the A-FPGA modules. This is because 'wiring' of the modules in the Python flow graph can remain static since the changes to modulation schemes now happen on the A-FPGA. For the GNU Radio scheduler it will simply dispatch and receive data regardless of modulation scheme. A server could be set up to receive commands, like in AgileHW, and issue the `add_module()` command, discussed in Section 5.4.2, to change the configuration of the A-FPGA.

A client/server model supports on-demand modifications more easily then a scripted environment. With standard Python scripts there is a very procedural top-to-bottom flow of execution. A client/server model requires a loop structure to poll or receive interrupts for commands. This idea can provide another level of dynamic capabilities on top of the standard scripting environment.

5.6.4 Streaming Radio Modules

As discussed in Section 3.1.1 the scheduler is set up for streaming data, which works well in a simple radio environment. If the physical level is the only design concern then the scheduler does enough to meet those needs. Though, some of the limitations this model brings are obvious with the use of even a simple test.

The problem arises if the model desired requires more intermodule communication than this or works asynchronously. A simple first test was to have a host machine file transfer with an A-FPGA loopback (containing both transmit and receive radios) and file reception back at the host. The issue is that this application is more asynchronous than

streaming and could benefit from an increase in communication between transmit and receive modules. If the transmit and receive paths do not interleave properly, data will not flow as desired. Also, when it comes time to halt after the file has been transmitted the receiver will continue to wait on data and has no knowledge of when dataflow should halt. The streaming top-to-bottom model does not mesh well with these ideas.

The standard GNU Radio dataflow was broken into at the top-layer where modules are declared. A new module defining the desired behavior of the A-FPGA was created as an alternate datapath. The A-FPGA module was responsible for its own set of functionality including properly directing data, and sending modules to the A-FPGA. The GNU Radio flow is merged back together at the Ethernet level where generalized functions take whatever data you have formed and send it. This model allows for seamless module additions at the Python level and offloading of costly computational tasks to an FPGA accelerator. Some parts of GNU Radio were designed around a model dissimilar to the one of this thesis causing issues in extending its use, but a few tweaks within the branched datapath made it possible to overcome the issues.

Chapter 6

Results

6.1 Testing and Hardware

This chapter demonstrates the results from implementation of the hardware accelerated GNU Radio. First, two use cases are demonstrated as a proof of concept for the framework. Next, quantitative results are given by analyzing the addition of a node in the network and device utilization. Finally, a discussion about code complexity and productivity gives insight into qualitative benefits.

The hardware setup is as follows: A host machine running an Intel Core2Duo E8400 at 3GHz, a Digilent XUPV5 with a Xilinx Virtex-5 LX110T FPGA as the A-FPGA, a TRENDnet TEG-S80g gigabit Ethernet switch, and the USRP2 from Ettus Research LLC as the radio front-end. All tests were run on Ubuntu 9.04 32-bit using version 3.2.2 of GNU Radio. All designs for the Virtex-5 were synthesized using version 11.1 of the Xilinx tools.

6.2 Proof of Concept

Two tests were used to evaluate the overall functionality of the framework. The first test was to demonstrate transmission capabilities by generating a sine wave from the Virtex-5 board, forming a packet with the sine wave data, sending the packet to the USRP2, and viewing the output from the USRP2 on a spectrum analyzer. The second test was to demonstrate reception capabilities by transmitting file data from the host machine to the Virtex-5, modulating the signal through the transmit radio chain, looping the modulated waveform back through the receive radio chain, and sending the original file data back to the host machine.

The first test was to prove the FPGA was capable of properly communicating with other devices using Ethernet frames. A packet generator on the Virtex-5 device was responsible for the formation of Ethernet packets destined for the USRP2. The payload data consisted of interleaved I and Q samples that represented real and imaginary values respectively. The data format for USRP2 communication was already specified by the standard GNU Radio. The local oscillator on the USRP2 was able to automatically generate a sine wave from a constant value input. To demonstrate this, Ethernet packets were formed with constant I values and zeros for Q values. The frames were sent from the Virtex-5 to the USRP2 and the output waveform of the USRP2 was measured by the spectrum analyzer, shown in Figure 6.1.

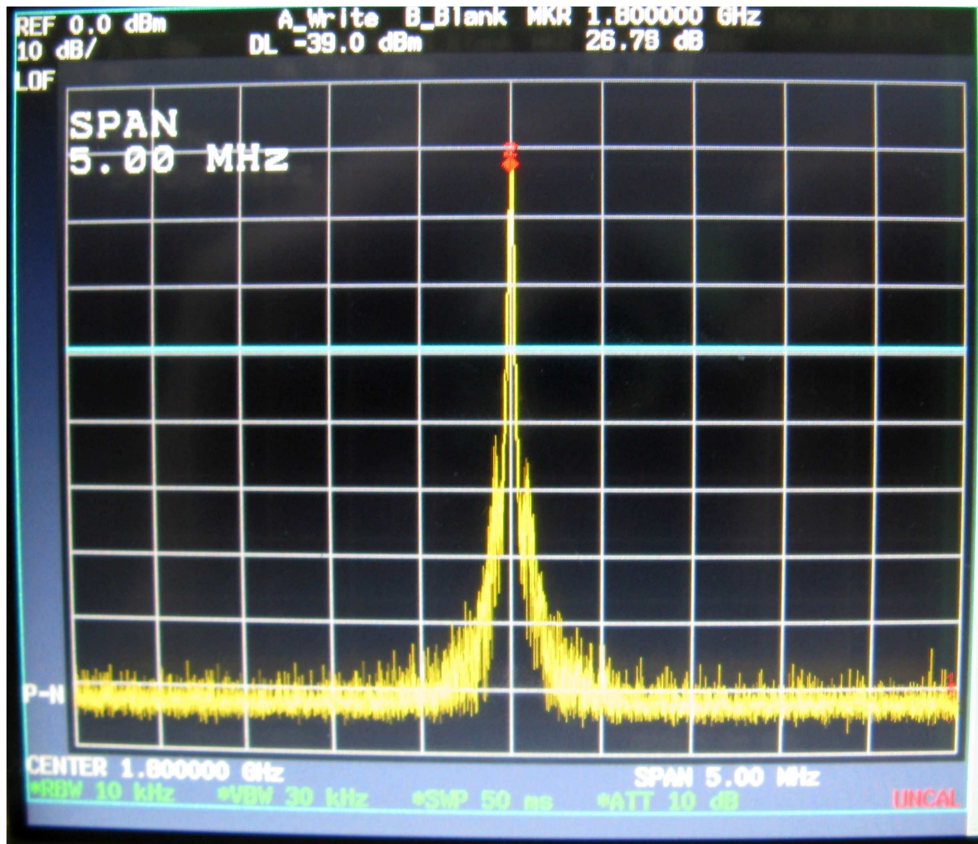

Figure 6.1: Spectrum Analyzer displaying a sin wave at 1.8GHz

The second test was to use an actual radio design to modulate and demodulate a signal on the Virtex-5. This was used to demonstrate end-to-end communication on the FPGA. This means the AgileHW method of CAD bus communication was properly ported to the XUPV5 platform using Ethernet. The radio used was a 250Kbps BPSK radio with an internal loopback between the modulation and demodulation chains. The radio modulates the incoming data as if it were going to the USRP2, then sends it through the demodulation chain to get back the original data.

This test also demonstrates the communication abilities from the software side. The host machine is responsible for reading a file, sending the Ethernet frames, receiving the frames back, and writing the result to a file. In Figure 6.2 a capture of the packets leaving

the host machine and going to the FPGA shows the file being sent. Figure 6.3 shows the file returning from the FPGA in proper condition.

```
14:24:27.656363 00:21:97:45:6b:3b (oui Unknown) > 00:0a:35:00:01:02
(oui Unknown), ethertype Unknown (0xbeef), length 772:
 0x0000:  0000 0500 0000 0000 0006 ffff ffff 0000  ................
 0x0010:  0173 5468 6520 4465 636c 6172 6174 696f  .sThe.Declaratio
 0x0020:  6e20 6f66 2049 6e64 6570 656e 6465 6e63  n.of.Independenc
 0x0030:  650a 0a57 6865 6e20 696e 2074 6865       e..When.in.the
14:24:27.656377 00:21:97:45:6b:3b (oui Unknown) > 00:0a:35:00:01:02
(oui Unknown), ethertype Unknown (0xbeef), length 776:
 0x0000:  0000 0500 0100 0000 0004 0000 0000 0000  ................
 0x0010:  0175 7220 6a75 7374 2070 6f77 6572 7320  .ur.just.powers.
 0x0020:  6672 6f6d 200a 7468 6520 636f 6e73 656e  from..the.consen
 0x0030:  7420 6f66 2074 6865 2067 6f76 6572       t.of.the.gover
14:24:27.656399 00:21:97:45:6b:3b (oui Unknown) > 00:0a:35:00:01:02
(oui Unknown), ethertype Unknown (0xbeef), length 592:
 0x0000:  0000 2469 0200 0000 0006 ffff ffff 0000  ..$i............
 0x0010:  0119 696e 7661 7269 6162 6c79 2074 6865  ..invariably.the
 0x0020:  2073 616d 6520 4f62 6a65 6374 200a 6576  .same.Object..ev
 0x0030:  696e 6365 7320 6120 6465 7369 676e       inces.a.design
```

Figure 6.2: A tcpdump capture of file being sent to A-FPGA

```
14:24:27.680449 00:0a:35:00:01:02 (oui Unknown) > 00:21:97:45:6b:3b
(oui Unknown), ethertype Unknown (0xbeef), length 772:
 0x0000:  0000 0000 0300 0000 0006 ffff ffff 0173  ...............s
 0x0010:  5468 6520 4465 636c 6172 6174 696f 6e20  The.Declaration.
 0x0020:  6f66 2049 6e64 6570 656e 6465 6e63 650a  of.Independence.
 0x0030:  0a57 6865 6e20 696e 2074 6865 2043 6f75  .When.in.the.Cou
 0x0040:  7273 6520 6f66 2068 756d 616e            rse.of.human
14:24:27.704519 00:0a:35:00:01:02 (oui Unknown) > 00:21:97:45:6b:3b
(oui Unknown), ethertype Unknown (0xbeef), length 776:
 0x0000:  0000 0000 0400 0000 0006 ffff ffff 0175  ...............u
 0x0010:  7220 6a75 7374 2070 6f77 6572 7320 6672  r.just.powers.fr
 0x0020:  6f6d 200a 7468 6520 636f 6e73 656e 7420  om..the.consent.
 0x0030:  6f66 2074 6865 2067 6f76 6572 6e65 642c  of.the.governed,
 0x0040:  20e2 8094 2054 6861 7420 7768            .....That.wh
14:24:27.722696 00:0a:35:00:01:02 (oui Unknown) > 00:21:97:45:6b:3b
(oui Unknown), ethertype Unknown (0xbeef), length 592:
 0x0000:  0000 0000 0500 0000 0006 ffff ffff 0119  ................
 0x0010:  696e 7661 7269 6162 6c79 2074 6865 2073  invariably.the.s
 0x0020:  616d 6520 4f62 6a65 6374 200a 6576 696e  ame.Object..evin
 0x0030:  6365 7320 6120 6465 7369 676e 2074 6f20  ces.a.design.to.
 0x0040:  7265 6475 6365 2074 6865 6d20            reduce.them.
```

Figure 6.3: A tcpdump capture of file coming back to A-FPGA

6.3 Quantitative Results

6.3.1 Latency

There is a concern with adding a node in-line to the existing flow and how this will affect latency. To test the latency of the link the XUPV5 sends out an Ethernet frame to a host program written in C++. The host program loops the Ethernet frame back to the XUPV5 where it notes its reception and registers the number of clock cycles it took. Note that with this model the number of clock cycles and latency calculations represent a round-trip. The first test was to test the latency with the switch and network/internet connection present in the loop.

Table 6.1: Latency Analysis on Network with Switch.

Packet Size	64	128	256	512	1024	1514
Max Cycles	2362	2518	3070	4856	6932	51618
Min Cycles	1838	2156	2714	3880	6256	8530
Avg Cycles	1946.9806	2229.3669	2804.0712	4031.3585	6418.1131	8731.3785
Avg Latency (µs)	15.5758	17.8349	22.4326	32.2509	51.3449	69.8510
Exp. Throughput (Mbps)	32.8714	57.4154	91.2958	127.0043	159.5484	173.3976

From the table it can be determined that packet size has a noticeable affect on latency. The change in latency is most likely a result of buffering at the driver level on the host machine as well as on the FPGA. This means the physical wire latency may be closer to constant while the transport cost increases at the driver level. In other words, there is a penalty for having to run through a software layer. The next iteration was to remove the network/internet connection and see if any network packets were interrupting communication latency.

Table 6.2: Latency Analysis off Network with Switch.

Packet Size	64	128	256	512	1024	1514
Max Cycles	2232	2610	3060	4372	6816	9006
Min Cycles	1844	2150	2724	3884	6272	8522
Avg Cycles	1947.3925	2268.4245	2809.2717	4022.7121	6416.8372	8626.0245
Avg Latency (µs)	15.5791	18.1474	22.4742	32.1817	51.3347	69.0082
Exp. Throughput (Mbps)	32.8645	56.4268	91.1268	127.2773	159.5802	175.5154

It is apparent that the network had a negligible affect on the latency. This becomes clear when remembering that dataflow is being controlled at the driver level. The communication with the host machine is occupying full use of the link, blocking any packets meant for network communication. The final iteration was to remove the network switch and quantify the link using a direct connection from host to FPGA. The results should be very similar to the standard GNU Radio model (host to USRP2).

Table 6.3: Latency Analysis off Network without Switch.

Packet Size	64	128	256	512	1024	1514
Max Cycles	1606	1696	2106	3276	4224	5578
Min Cycles	1298	1438	1768	2450	3822	5042
Avg Cycles	1360.6874	1502.5280	1833.8193	2518.2362	3893.7357	5132.8907
Avg Latency (µs)	10.8855	12.0202	14.6706	20.1459	31.1499	41.0631
Exp. Throughput (Mbps)	47.0350	85.1898	139.5994	203.3169	262.9865	294.9605

There is a noticeable reduction in latency by removing the switch. The latency looks to be linear, not fixed, for the switch as well. Though the latency impact is noticeable, it is not overwhelming. It might seem desirable to use smaller packets to keep latency down, but the next section will show that this has its own set of drawbacks.

PCIe is a recent technology that has the benefit of high-bandwidth and low-latency. The 1x PCIe 2.0 lane on the XUPV5 is capable of 4Gbps throughput in each direction. PCIe is also capable of achieving sub 5µs one-way latency [34] using packet message sizes similar to this experiment.

PCIe is the preferred interconnect for designs requiring extremely stringent latency requirements like IEEE802.11. If this standard is desired for the current set-up then acknowledgement packets could be handled on the USRP2 and, given there is proper remaining resources on the Spartan 3, it would possible. Given that the majority of latency increases appear to be the result of software intervention, any tasks requiring low-latency should be the responsibility of the USRP2 or A-FPGA.

6.3.2 Throughput

For throughput calculations a C program was written, using GNU Radio Ethernet primitives, to send and receive Ethernet frames between the host and XUPV5 board. The Virtex-5 had loopback logic for swapping the source and destination MAC address fields. The system was designed to have as minimal intervention as possible on both ends. The first test interleaves the send and receive commands. One frame is sent from the host machine to the XUPV5. The Virtex-5 receives the packet, swaps the MAC address fields, and sends it back to the host. The host will receive the frame and, only then, send the next frame out. This process iterates 100,000 times and time measurements are taken to derive the bi-directional throughput calculations.

Table 6.4: Throughput Analysis without Burst

With Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	2.0002	2.2667	2.8467	4.0308	6.3272	8.4855
Min Time (s)	1.9020	2.1889	2.7658	3.9320	6.2635	8.3328
Avg Time (s)	1.9396	2.2200	2.8145	3.9720	6.3031	8.4321
Avg Throughput (Mbps)	26.3966	46.1254	72.7664	103.1207	129.9677	143.6419
Without Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	1.8804	2.2967	2.3903	2.7625	4.3149	5.7164
Min Time (s)	1.6263	1.9395	2.1304	2.7137	4.2045	5.6330
Avg Time (s)	1.7453	2.0932	2.2864	2.7341	4.2587	5.6906
Avg Throughput (Mbps)	29.3353	48.9212	89.5751	149.8142	192.3575	212.8438

The actual throughput is slightly lower than expected from the latency calculations, and this may be due to the address swapping logic on the Virtex-5. As expected, the switch does have an impact on the throughput. One important thing to notice is that the larger packet sizes result in higher throughput, but as shown in the last section have higher latency as well. This makes for a trade-off between link performance and packet sizes. Gigabit Ethernet supports up to 1Gbps throughput and with 143Mbps the line utilization is only around 15%. To improve this, the buffering capabilities of Ethernet were used to do burst send and receive.

Table 6.5: Throughput Analysis 2 Packet Burst

With Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	4.8316	1.6442	99.5223	3.4264	3.5811	4.9030
Min Time (s)	1.3268	1.4676	45.9863	2.3965	3.5300	4.8422
Avg Time (s)	2.6158	1.5710	63.9072	3.0968	3.5521	4.8721
Avg Throughput (Mbps)	19.5737	65.1814	3.2046	132.2642	230.6262	248.6015
Without Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	0.8689	1.5950	37.9993	2.4506	2.6284	3.4798
Min Time (s)	0.6719	1.2790	6.4505	1.6521	2.5403	3.4709
Avg Time (s)	0.7680	1.4269	19.1135	2.0645	2.5918	3.4751
Avg Throughput (Mbps)	66.6708	71.7663	10.7150	198.4048	316.0692	348.5406

Sending two packets at a time resulted in a significant speed up. The throughput with the switch using burst was better than the throughput without the switch and without burst, for the most part. The smaller packet sizes suffered from burst sending, most likely due to high numbers of collisions on the line. Without the switch having to act as the traffic cop, the smaller packets were able to achieve respectable throughput gains through reduced collisions. Something unexplainable is why the 256 packet size did very poorly. Next, four frame bursts were used to see if some of these issues would be fixed, or worsened.

Table 6.6: Throughput Analysis 4 Packet Burst

With Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	13.1374	13.9980	38.2438	1.9426	2.2384	3.0831
Min Time (s)	3.3345	1.4152	10.9498	1.3325	2.1985	3.0518
Avg Time (s)	6.8778	9.7919	28.0575	1.6073	2.2162	3.0625
Avg Throughput (Mbps)	7.4442	10.4576	7.2993	254.8377	369.6500	395.4875
Without Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	52.4976	4.1072	22.1401	1.7019	1.8790	2.3559
Min Time (s)	0.6769	1.5733	4.2839	1.4215	1.6793	2.3371
Avg Time (s)	9.4050	2.7841	9.5367	1.5872	1.7202	2.3428
Avg Throughput (Mbps)	5.4439	36.7808	21.4748	258.0595	476.2181	516.9792

The first thing to notice is the line utilization is above 50% without the switch. Even with the switch, the throughput is reaching good utilization. The second thing to notice is that small packets continue to suffer. The 256 Byte frame still does poorly, even compared to the 128 Byte packet that should be slower. This behavior continues through eight and sixteen frame bursts gradually making anything under 512 Byte packets unusable. The next case will demonstrate the maximum burst amount that was reasonably achievable.

Table 6.7: Throughput Analysis 64 Packet Burst

With Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	-	-	-	-	0.9340	1.3445
Min Time (s)	-	-	-	-	0.9231	1.3435
Avg Time (s)	-	-	-	-	0.9269	1.3439
Avg Throughput (Mbps)	-	-	-	-	883.8316	901.2386
Without Switch						
Packet Size	64	128	256	512	1024	1514
Max Time (s)	-	-	-	-	0.9060	1.3098
Min Time (s)	-	-	-	-	0.8917	1.2995
Avg Time (s)	-	-	-	-	0.8950	1.3008
Avg Throughput (Mbps)	-	-	-	-	915.3175	931.1449

Using 64 frame burst the line utilization is above 90% for both cases. The impact of the switch is minimal at this level. The smaller packets suffered in this test to the point of being unusable. For the switch to have the lowest impact on the system, the burst model with large packets is the preferred method. The current high-speed radio being used for AgileHW has a data throughput of 6Mbps making the line speed shown here acceptable for the use of current AgileHW radios.

6.3.3 Utilization

Figure 6.4 shows the utilization summary for the FPGA given by the Xilinx tools. The design used here was test two from Section 6.2. The design consists of both the send and receive modulation paths for a 250Kbps BPSK radio as well as the glue logic and Ethernet communication modules. This is a simple radio, yet shows that even a full design only takes around 5% of the entire LX110T's resources. With this level of utilization for a full radio design the XUPV5 will be a scalable platform for future designs.

```
Device Utilization Summary:

   Number of BUFDSs                            1 out of 8      12%
   Number of BUFGs                             4 out of 32     12%
   Number of BUFRs                             1 out of 32      3%
   Number of DCM_ADVs                          1 out of 12      8%
   Number of DSP48Es                          13 out of 64     20%
   Number of GTP_DUALs                         1 out of 8      12%
      Number of LOCed GTP_DUALs                1 out of 1     100%

   Number of External IOBs                     3 out of 640     1%
      Number of LOCed IOBs                     3 out of 3     100%

   Number of External IPADs                    4 out of 690     1%
      Number of LOCed IPADs                    2 out of 4      50%

   Number of OLOGICs                           1 out of 800     1%
   Number of External OPADs                    2 out of 32      6%
      Number of LOCed OPADs                    0 out of 2       0%

   Number of RAMB18X2s                         4 out of 148     2%
   Number of RAMB36_EXPs                       8 out of 148     5%
   Number of TEMACs                            1 out of 2      50%
   Number of Slice Registers                3039 out of 69120   4%
      Number used as Flip Flops             3031
      Number used as Latches                   8
      Number used as LatchThrus                0

   Number of Slice LUTS                     2386 out of 69120   3%
   Number of Slice LUT-Flip Flop pairs      3584 out of 69120   5%
```

Figure 6.4: Xilinx report of FPGA utilization of use case two

6.4 Attribute Comparison

The table below is a comparison of different attributes shared by the SDR platforms presented in Section 2.1. The work for this thesis falls under "Enhanced GNU Radio". Note that is has some distinct advantages over some of the existing solutions. It is the only open source platform with hardware support, compile times in the software range, and modularity for ease of use.

Table 6.8: Comparison of features and attributes for SDR platforms

	SORA	WARP	OSSIE	GNU Radio	Enhanced GNU Radio	GPU SDR
Open Source			✓	✓	✓	
Modularity			✓	✓	✓	
Compile Time[1]	S	H	S	S	S	S
Number of Languages[2]	1	1	*	2	3	1
Interface	PCIe	Mem. Mapped		GigE	GigE	PCIe
Embedded Use		✓				
Hardware Signal Processing		✓			✓	✓
Low-Latency	✓	✓				✓
Scalability		✓			✓	✓
Extensibility			✓	✓	✓	
Scriptable				✓	✓	
Cost[3]	$2000	$6500	*	$1400/*	$2150	$30-$2500

[1]"S" means software compile times (<1s) and "H" means hardware compile times (5-60m)

[2]OSSIE provides many tools that use different underlying languages. Most of the interaction is at the GUI level. In GNU Radio the language for designing radios is meant to be Python though C++ is used for underlying performance tasks. Verilog is used for Hardware acceleration in the Enhanced GNU Radio.

[3]OSSIE and GNU Radio have optional hardware requirements. Hardware is necessary for over the air transmission.

General Observations:

- Modular environments naturally lead to extensible systems by giving the user a template for how to make new designs.
- If the designer is willing to sacrifice design time as a constraint it will generally lead to a more efficient design.
- It is more difficult to ensure scalability in a system done entirely in software.
- Software Defined Radio does not mean that everything must be done in software.

6.5 Analysis of Code Complexity and Productivity

One of the big draws for SDR is the impact it can have on productivity. It is a long standing issue that hardware design takes significant effort and time when compared to software. The modular solution attempts to resolve some of the issues with hardware design productivity. This section will assess how the solution in this thesis aids or hurts productivity in certain areas of development.

The standard method of hardware design is to write Verilog or VHDL code describing the desired circuit, run this code through synthesis tools provided by the target FPGA manufacture, and place the resulting bitstream on the FPGA. The process described can be very tedious because of clock level details and bitstream generation times. Any non-trivial design will require changes and analysis of timing where test iterations can take from 5 to 60 minutes depending on the design size and CPU running the tools. It is apparent that productivity will suffer in this environment.

In a modular environment the design process looks different. Productivity in a modular environment will vary depending on whether the desired building blocks are available in a pre-compiled library or not. If the modules are available then the time taken to

design a non-trivial system could be as little as a day. Using Python and the hardware accelerated GNU Radio as the design environment, the designer would write a script, in software, defining the desired radio. This will be called scenario one. For scenario two, if the modules desired were not present the designer would need to write the HDL description of the desired module, running it through a slightly modified version of the tool-chain, then adding it to the system by including it in the software script. The design of the additional module might still be a significant undertaking but the benefit is the designer does not need to worry about the details of the rest of the system. Also, once the design is complete, any future work can take advantage of re-use and designing like scenario one.

Taking a look at scenario one, the designer has everything available to them. Though, the difference in architectures can still play a roll in productivity even when everything is present. In the case of a static hardware framework, once the pre-designed hardware modules are laid out in HDL the designer will need to wait 5-60 minutes for the bitstream to be generated. Any simple mistake in the HDL will take another iteration of this process to rectify. The ease of linking up the modules in hardware is also assuming there is a pre-existing common interface between the modules, otherwise glue logic would need to be designed. If the designer is using the standard GNU Radio flow they would get instant gratification from Python, where small errors would not result in major losses. So, the time to physically code a radio when given pre-designed components is the same, but with GNU Radio bitstream generation time is not a factor and small errors are not as much of a hindrance. With the modified GNU Radio flow these benefits are still available along with higher speed hardware modules.

Using the best case scenario for static hardware design where a hardware module library exists, each module has a compatible interface, and the designer makes no mistakes, the time savings would be 5-60 minutes. This is not a huge savings, yet this scenario is extremely rare. From the second use case in Section 6.2 it took one week to glue the radio logic into the existing Ethernet framework having already had working radio modules.

The same loopback test took one day to script in Python. This is a more realistic comparison of the two methods even when the radio components are pre-designed.

If the radio components are not pre-designed or do not exhibit the desired functionality it may result in a significant loss in productivity. Starting from scratch, a very simple 250Kbps BPSK radio took three months to design and implement. If the designer already has some code to work with this time can be reduced. An improved 6Mbps version of this radio took only three weeks to design and implement. A more advanced radio like OFDM might take up to a year to complete. This shows the importance of including a robust set of functionality.

The concept of reuse and productivity really drives this work towards the academic community rather than industry. Companies using radio frequency technologies are generally after the latest improvements in the field and will be designing and implementing new ideas rather than reusing current ones. There is some overlap, yet it might be very minimal. In the academic community you have students who are new to communications and would benefit from the availability of pre-designed modules. It would allow students to test theories, learn, and come up to speed with modern advancements in communications quickly.

If the target is the academic community it is very important to have everything well documented. GNU Radio has tutorials on creating your own modules as well as many basic designs to learn from. It took roughly a month to get up to speed with GNU Radio, which is an acceptable timeframe for most students.

Chapter 7

Conclusion

The idea of SDR is to provide flexibility and instant gratification in signal processing, and GNU Radio uses these concepts in both the design and execution phases of radio deployment. Using a microprocessor to compute all signal processing data does give you this flexibility but this is a power hungry device that is not well suited for this type of work. An FPGA provides a much more efficient and powerful platform for signal processing, yet the traditional build time for FPGAs is too long and insufficiently agile. This research provides a robust framework to tie the concepts of FPGAs, partial reconfiguration, and software defined radio into an easily accessible and high-productivity solution.

The enhanced GNU Radio is a framework that is efficient and robust for the development of radio designs. The enhancements maintain the existing GNU Radio flow giving any designer an easy opportunity for the use of hardware acceleration. The enhancements also conserve the open source philosophy of GNU Radio while providing a platform independent solution for AgileHW. Having the framework as open source also supports the idea of making AgileHW more mainstream. The additions made to GNU Radio along with the foresight of AgileHW make this framework an ideal solution for radio designers.

7.1 Future Work

7.1.1 MAC Layer Protocols

There is currently no intention of providing a high-level guaranteed transmission service, such as TCP. A protocol that supports guaranteed transmission can be supported in the future at a high software level, though is not absolutely essential to the overall function of the system. The decision behind this related to the fact that GNU Radio currently does not have any native support for MAC layer protocols. There exists a notion of sequence numbers for informing the user if an error has occurred, yet there is no way to recover from this. This logic is the reasoning behind the CRC field which might help to support future improvements.

7.1.2 Partial Run-time Reconfiguration

The idea of partial reconfiguration is large portion of the ultimate goal of this work. The framework described above is intended to be designed in robust manner that will support the future work in this area. The idea of a hardware co-processor can be realized with the framework alone, yet lacks the realization of the productivity desired.

7.1.3 Radio Designs

OFDM and ZigBee There is a need to develop radios that are able to communicate with a standard communication protocol. This will enable the testing of the framework in a real-life scenario against standardized hardware. Zigbee is a very useful low-speed WLAN standard based on the IEEE 802.15.4. The simplicity to implement Zigbee makes it an ideal standard to be developed as a proof-of-concept design for the framework. Previous work by the CCM Lab using Zigbee radio [35] makes it a strong candidate for the use

case. It is intended to deploy this ZigBee radio on the enhanced GNU Radio platform, and to interact with an off-the-shelf ZigBee component.

Bibliography

[1] Incremental compilation resource center. [Online]. Available: http://www.altera.com/support/software/incremental/sof-qts-increment-comp.html

[2] The free software definition. [Online]. Available: http://www.gnu.org/philosophy/free-sw.html

[3] E. Blossom. (2010) Gnu radio. [Online]. Available: http://www.gnuradio.org

[4] P. Athanas, J. Bowen, T. Dunham, C. Patterson, J. Rice, M. Shelburne, J. Suris, M. Bucciero, and J. Graf, "Wires on demand: Run-time communication synthesis for reconfigurable computing." Proceedings of the 2007 International Conference on Field Programmable Logic and Applications, FPL 2007, Amsterdam, Netherlands, Aug 2007.

[5] J. Suris, M. Shelburne, C. Patterson, P. Athanas, J. Bowen, T. Dunham, and J. Rice, "Untethered on-the-fly radio assembly with wires-on-demand." Proceedings of the 2008 National Aerospace and Electronics Conference, NAECON 2008, Fairborn, Ohio, Jul 2008.

[6] J. Suris, P. Athanas, and C. Patterson, "An efficient run-time router for connecting modules in fpgas." Proceedings of the 2008 International Conference on Field Programmable Logic and Applications, FPL 2008, Heidelberg, Germany, Sep 2008.

[7] J. M. III. [Online]. Available: http://web.it.kth.se/ maguire/jmitola/

[8] ——, “The software radio.” National Telesystems Conference, 1992 - Digital Object Identifier 10.1109/NTC.1992.267870.

[9] E. Blossom, “Gnu radio: Tools for exploring the radio frequency spectrum,” *Linux journal*, vol. 122, pp. 76–81, 2004.

[10] Software-defined radio (sdr) system. [Online]. Available: http://sine.ni.com/nips/cds/view/p/lang/en/nid/207092

[11] M. Robert, S. Sayed, C. Aguayo, R. Menon, K. Channak, C. Valk, C. Neely, T. Tsou, J. Mandeville, and J. Reed, “Ossie: Open source sca for researchers.” Proceedings of the SDR 04 Technical Conference and Product Exposition, 2004.

[12] J. Gluck. Wireless innovation forum. [Online]. Available: http://www.wirelessinnovation.org

[13] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. Voelker, “Sora: High performance software radio using general purpose multi-core processors,” *NSDI 2009*.

[14] P. Murphy, A. Sabharwal, and B. Aazhang, “Design of warp: A wireless open-access research platform,” in *European Signal Processing Conference*, 2006.

[15] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal, “WARP, a Unified Wireless Network Testbed for Education and Research,” *2007 IEEE International Conference on Microelectronic Systems Education (MSE’07)*, pp. 53–54, June 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4231446

[16] P. Balister, M. Robert, and J. Reed, “Impact of the use of corba for inter-component communication in sca based radio.” Proceedings of the SDR 06 Technical Conference and Product Exposition, 2006.

[17] M. Carrick, "Logical representation of fpgas and fpga circuits within the sca," Master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, 2009.

[18] (2010) Cuda zone. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[19] (2008) Gnuradio plus cuda. [Online]. Available: http://www.smallwhitecube.com/php/dokuwiki/doku.php?id=howto:gnuradio-with-cuda

[20] J. Kim, S. Hyeon, and S. Choi, "Implementation of an SDR system using graphics processing unit," *IEEE Communications Magazine*, pp. 156–162, March 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4231446

[21] D. Hefenbrock, J. Oberg, N. Nguyen, R. Kastner, and S. Baden, "Accelerating Viola-Jones Face Detection to FPGA-Level using GPUs." Orlando: Field-Programmable Custom Computing Machines, 2010.

[22] Nvidia. (2010) Geforce gtx 480. [Online]. Available: http://www.nvidia.com/object/product_geforce_gtx_480_us.html

[23] Xilinx. (2010) Virtex-6 family overview. [Online]. Available: http://www.xilinx.com/products/virtex6/lxt.htm

[24] Y. Tachwali and H. Refai, "Implementation of a BPSK Transceiver on Hybrid Software Defined Radio Platforms," in *2008 3rd International Conference on Information and Communication Technologies: From Theory to Applications*. Ieee, April 2008, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4530253

[25] J.-p. Delahaye, P. Leray, L. Godard, A. Nafkha, C. Moy, D. G. A. Celar, F. Mod, and C. Rennes, "Designing A Reconfigurable Processing Datapath For SDR Over Heterogeneous Reconfigurable Platforms," in *SDR Forum Technical Conference*, Orlando, 2007.

[26] R. Dhar, G. George, A. Malani, and P. Steenkiste, "Supporting integrated MAC and PHY software development for the USRP SDR," *IEEE Workshop on Networking Technologies for Software Defined Radio (SDR) Networks*, 2006.

[27] H. Yousefi'zadeh and A. Qureshi, "A case study of a MIMO SDR implementation," *MILCOM 2008 - 2008 IEEE Military Communications Conference*, pp. 1–7, November 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4753441

[28] J. Suris, A. Recio, and P. Athanas, "Rapid radio: A framework for human-assisted signal classification and receiver implementation." Proceedings of the 2008 National Aerospace and Electronics Conference, NAECON 2008, Fairborn, Ohio, Dec 2007.

[29] ——, "Enhancing the productivity of radio designers with rapidradio." 2009 International Conference on ReConFigurable Computing and FPGAs, Cancun, Mexico, Dec 2009.

[30] (2006) How to write a signal processing block. [Online]. Available: http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html

[31] (2009) Gnu coding standards. [Online]. Available: http://www.gnu.org/prep/standards

[32] (2009) Boost smart pointers. [Online]. Available: http://www.boost.org/doc/libs/1_42_0/libs/smart_ptr/smart_ptr.htm

[33] (2008) Welcome to swig. [Online]. Available: http://www.swig.org/

[34] M. Koop, W. Huang, K. Gopalakrishnan, and D. Panda, "Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand," *16th IEEE Symposium on High Performance Interconnects*, pp. 85–92, August 2008.

[35] T. Meng, C. Zhang, and P. Athanas, "An fpga-based zigbee receiver on the harris software defined radio sip." Software Defined Radio Forum, SDRForum, Denver, CO, May 2007.

[36] (2009) Usrp2 user faqs. [Online]. Available: http://gnuradio.org/redmine/wiki/gnuradio/USRP2UserFAQ

[37] (2006) Autotools tutorial. [Online]. Available: http://sources.redhat.com/autobook

Appendix A

Extras

A.1 Code Organization

This section illustrates the layout for the code that was written. It provides a good starting point for anyone looking to make modifications to the work in this thesis, or as supplemental documentation for understanding the code layout of GNU Radio.

/gnuradio

Root directory for the entire project. Contains subdirectories for all modules that are directories starting with gr-*

/gnuradio/afpga

All of the C++ and Verilog implementation code for the A-FPGA system

/gnuradio/afpga/host

The C++ code that is run on the host machine

/gnuradio/afpga/host/apps

Applications directory for everything running on the host

/gnuradio/afpga/host/lib

The low-level C++ code that defines the A-FPGA system

/gnuradio/afpga/host/include

The C++ header files

/gnuradio/afpga/fpga

The Verilog code that defines the FPGA

The firmware is programmed to flash from here with the u2_flash_tool [36]

/gnuradio/usrp2

All of the C++ and Verilog implementation code for the USRP2

This directory is laid out in the same way as the A-FPGA directory

/gnuradio/gr-afpga

The C++ and SWIG code for defining the module interface for use in Python

/gnuradio/gnuradio-examples/python

Provides a lot of examples for how to use GNU Radio and Python

/gnuradio/gr-utils

Provides some simple utilities that are useful and supplement the /gnuradio-examples directory

A.2 Autotools

GNU Radio is a highly complex system with many files and dependencies. A set of tools were developed by the GNU project to aid open source projects in building complex systems. The standard is a tool flow that abstracts the user from the build process with a small set of commands high-level commands. These commands are interpreted by the tool flow to produce a very robust build system. The autotools allow for cross-platform development and allows for code that will run on many Unix-like operating systems [37]. This section is intended to give a new user of GNU Radio information about which files are important, and how to make changes to GNU Radio.

Using the autotools means working with three utilities: Autoconf, Automake, and Libtool. The user is responsible for creating a `configure.ac` file, a `Makefile.am` file, as well as helper files like `Makefile.in` and `.m4` files for Autoconf. These files exist at the top-level and use a set of relatively simple commands. There are many examples throughout the GNU Radio directory on how to set up these files.

If simple changes are to be made, such as adding an application, the user should only concern themselves with the `Makefile.am` in the corresponding directory. There will be a location within the file for declaring the executable and what source files it uses. After making the necessary changes to the `Makefile.am` the user needs to run `configure` again from the root directory. This will produce a new `Makefile` that will build the intended executable properly.

If the user wishes to create an entirely new module there are some more complicated steps involved. The `configure.ac` file will need to be modified in the root directory and a new `.m4` file will need to be made in the `/config` directory. The new module should follow the standard directory layout to ensure everything is properly built. For more information on the autotools and how to build your own module there is a well written tutorial on the GNU Radio webpage entitled "How to Write a Signal Processing Block" [30]. This tutorial in combination with observing existing blocks makes for a good starting point.