Enhancing GNU Radio for Run-Time Assembly of FPGA-Based Accelerators

Richard H. L. Stroop

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

August 7, 2012
Blacksburg, Virginia

Keywords: GNU Radio, FPGA, SDR, qFlow

Enhancing GNU Radio for Run-Time Assembly of FPGA-Based Accelerators

Richard H. L. Stroop

(ABSTRACT)

Software defined radios (SDRs) have changed the paradigm of slowly designing custom radios, instead allowing designers to quickly iterate designs with a large range of functionality. With the help of environments like the open-source project, GNU Radio, a designer can prototype radios with greatly improved productivity. Unfortunately, due to software performance limitations, there is no way to achieve the range of radio designs made possible with actual physical radio hardware. In order for SDRs to become more prevalent in radio prototyping and development, accelerators must be added to high-throughput and computationally intensive portions. Custom DSPs, GPUs, and FPGAs have all been added to SDRs to try and expand their computational capabilities. One difficulty in this is that by adding these accelerators, the "instant gratification" dynamic of the GNU Radio is lost.

In this thesis, an enhanced GNU Radio flow is presented that seamlessly augments the GNU Radio software-only model with FPGAs, yet preserves the GNU Radio dynamics by providing full-custom radio hardware/software structures in seconds. By delegating portions of a GNU Radio flow graph to networked FPGAs, a larger class of software-defined radios can be implemented. Assembly of the signal processing structures within the FPGAs is accomplished using an enhanced flow where modules are customized, placed, and routed in a fraction of the time required by the vendor tools. With rapid FPGA assembly, a GNU Radio designer retains the ability to perform "what-if" experiments, which in turn greatly enhances productivity.

Due to the modular nature of GNU Radio and of the FPGA designs, a modular assembly of the FPGA hardware is used. In the flow presented here, optimized hardware library components are designed by a domain expert, and stored as compact placed-and-routed modules. When a designer requests the assembly of one or more components within a given FPGA via a GNU Radio Python script, the necessary library components are accessed and translated to an appropriate location within the chip. Then the ports of the modules are stitched together using a custom FPGA router. This process reduces the large compile times of hardware for an FPGA to reasonable software-like times.

To the radio designer, the complexity of the underlying hardware is abstracted away, making it appear as if everything compiles and runs in software, allowing many iterations to be realized quickly. Radio design can continue at the speeds that GNU Radio designers are accustomed to but with the range of possible waveforms and general functionality extended.

This work received support from sponsors of the CCM Lab who wish to remain unnamed.

Contents

List of Figures

List of Tables

List of Abbreviations

ADC	Analog to Digital Converter
AFPGA	Auxiliary FPGA
ASCII	American Standard Code for Information Interchange
BPSK	Binary Phase Shift Keying
CORBA	Common Object Request Broker Architecture
CUDA	Compute Unified Device Architecture
DAC	Digital to Analog Converter
DSP	Digital Signal Processor
EDIF	Electronic Design Interchange Format
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field-Programmable Gate Array
GNU	GNU's Not Unix!
GPP	General Purpose Processor
GPU	Graphics Processing Unit
GRC	GNU Radio Companion
ISE	Integrated Synthesis Environment
LED	Light Emitting Diode
LVDS	Low-Voltage Differential Signaling

MAC	Media Access Control
MGT	Multi-Gigabit Transceivers
MPRG	Mobile and Portable Radio Research Group
OSSIE	Open-Source SCA Implementation Embedded
PAR	Place and Route
PLL	Phase-Locked Loop
qFlow	Quick Flow
QUIP	Quartus II University Interface Program
SDR	Software Defined Radio
RAM	Random Access Memory
RF	Radio Frequency
SATA	Serial AT Attachment
SCA	Software Communications Architecture
SDR	Software-Defined Radio
SORA	SOftware-developed RAdio
tFlow	Turbo Flow
TORC	Tools for Open Reconfigurable Computing
USB	Universal Serial Bus
USRP	Universal Software Radio Peripheral
WARP	Wireless Open-Access Research Platform
XDL	Xilinx Description Language
XML	Extensible Markup Language

Chapter 1

Introduction

Wireless technologies are growing and evolving rapidly, driven by the increasing desire by people for continuous connectivity. More and more antennas and protocols have been added to devices that need to be completely connected. People cannot afford to have countless gadgets in order to talk to all of the other devices and services available, but everyone wants their devices to be small and cheap. This is seemingly impossible since every time a new protocol or standard is introduced another specialized chip has to be added to a device. The need for reconfigurable signal processing chips is quickly becoming unavoidable.

As processors have become more powerful, they have gained the ability to implement a variety of signal processing functions. This ability has fostered the field of software-defined radios (SDRs), which circumvents specialized signal processing chips by implementing them as software subroutines. The elimination of the need for specialized hardware not only increases the flexibility of systems to implement current radio protocols but also allows for the rapid prototyping of new designs. Not having to design for multiple pieces of hardware or to wait for them to be built dramatically decreases the time it takes radio designers to iterate through ideas. The inherent reconfigurability of CPUs makes them useful tools for realizing this rapid development but their inability to process large amounts of data at once has always limited their use for high-throughput signal processing [40]. Even the most sophisticated processor can only achieve limited data rates and it becomes difficult to meet latency requirements as more processes are added.

Processing limitations of contemporary CPUs restrict the number of protocols that can be implemented and the prototypes that can be rapidly designed for SDRs. To speed up processor intensive protocols, specialized hardware must be used by software radios that accelerates the signal processing back to a tolerable speed. Unfortunately adding hardware back into a flow designed for only software brings prototyping back to a crawl. The basics of setting up a physical link that allows an SDR to communicate with specialized hardware at reasonable speeds can be too complex [13]. SDR designers are also required to have a deep knowledge of the hardware they are trying to add and this is considered too difficult for basic radio researchers [23]. This makes adding hardware acceleration for real-time performance unrealistic for prototyped systems.

A welcome advance for radio designers would be the ability to incorporate hardware acceleration with their designs to allow for easy prototyping of complex waveforms. It would be useful to add this ability to an environment that SDR designers are already used to. The environment would also need to retain the ability to quickly iterate through prototypes even with the added hardware. Maintaining the flexibility and reconfigurability of current SDR systems would also be a high priority.

1.1 Contribution

This thesis presents work done to enhance the GNU Radio development environment by delivering an easy method for adding hardware acceleration to a software radio. GNU Radio operates by piecing together software modules, either graphically or via a Python script, that call different functions on data as it streams through a flow graph. Field Programmable Gate Arrays (FPGAs) are used for the hardware acceleration because they offer the ability to write completely customizable hardware for high-throughput signal processing. Despite their powerful processing capabilities, FPGAs normally slow design time down. They require knowledge of a hardware description language that is no where near as abstract as the software languages radio designers are used to and their compile times are orders of magnitude greater than software. Both of these limitations are addressed in this thesis. The communication interfaces are implemented for the FPGA so that radio hardware designers do not have to develop their own system. The data are abstracted to types that GNU Radio software blocks use and presented in a standard way to the designer, regardless of the interface used to receive the data. In this way only the basic signal processing blocks must be designed in order to see them work within the framework of GNU Radio. The signal processing blocks that are used often can be added to a community-based library for easy reuse. And the slow compile times are overcome with off-the-clock pre-processing and modular based assembly. The project is titled GReasy, GNU Radio easy, because now implementing signal processing with an FPGA accelerator is as easy and fast as creating a normal flow graph in GNU Radio. Radio designers can now continue to rapidly prototype in an environment that they are used to but with the added functionality of hardware acceleration that does not slow them down.

In GNU Radio, the signal processing sections of the flow graph are broken down into blocks specified by an underlying C/C++ model. In this thesis, the GNU Radio framework has been extended to include the concept of *hardware blocks*, which are designed to run on FPGAs alongside blocks running on a processor. These blocks are added to GNU Radio as easily as adding a software block is already. To a GNU Radio user, these blocks simply describe ports and how the data should be moved around; all of the processing is left to the hardware. The FPGA designer is not left out of this process but their role is limited to building a library of components for radio designers to use. Custom blocks can still be written by wireless domain experts and added to the flow. The actual processing must be designed in some hardware description language, but an entire system does not need to be designed to run it. As long as the block follows a standard format used for passing signal data in GNU Radio, it will fit in with any other block.

All of the communication between hardware blocks and GNU Radio is handled for the designer already with the use of a static region on the FPGA containing common interface logic. This lets hardware designers focus on getting their specific problems solved without having to worry about an entire system. They also do not have to redesign blocks that are already available with a standard structure. The radio designers can then easily use these blocks in GNU Radio without having to worry about the hardware designers. The set of blocks that go on the FPGA are built at run-time and maintain flexibility within GNU Radio. The normally slow compile times of FPGAs are overcome with a back-end accelerator called qFlow [16] that fits the block model of piecing together a radio nicely. With these additions, the instant gratification of GNU Radio is preserved while greatly increasing the number of possible radios that can be designed.

The specific contributions of this work include:

- Creating a class of block in GNU Radio that is recognized as operating on hardware,
- Writing a program that converts GNU Radio block connections into the Electronic Design Interchange Format (EDIF) for building hardware,
- Creating a static region for an FPGA that communicates with GNU Radio,
- Connecting GNU Radio with qFlow to maintain software-like compile times for the sections that assemble hardware,
- Scripting the synthesis process for adding hardware blocks to GNU Radio, and
- Extending the hardware blocks to the graphical front-end for GNU Radio the GNU Radio Companion (GRC) [8].

1.2 Organization

The rest of this thesis is organized as follows. Chapter 2 discusses the field of software-defined radios and includes some of the current development systems and paradigms used. Hardware acceleration with these systems is explored and a focus on FPGAs is given to support the work done for this thesis. Previous methods of hardware assembly are presented to compare to the use of qFlow. Finally related work with respect to accelerating GNU Radio that was used to motivate this project is given. Chapter 3 discusses the improvements made to GNU Radio from the work presented here and from previous work. This chapter describes how and why GNU Radio is used. Chapter 4 explains the development done in hardware to facilitate the ease of use for designers in the future. It also outlines the standard that was decided upon for use in all of the modular systems. Chapter 5 gives an overview of qFlow and a similar system tFlow that allow for rapid assembly of modular designs and allow for even more productivity and flexibility in adding hardware acceleration. Chapter 6 provides details on the specific work that was done to implement and test the improvements to GNU Radio. A comparison of implementation choices is done and used to justify the final results. Chapter 7 concludes the thesis and presents ideas for future work.

Chapter 2

Background

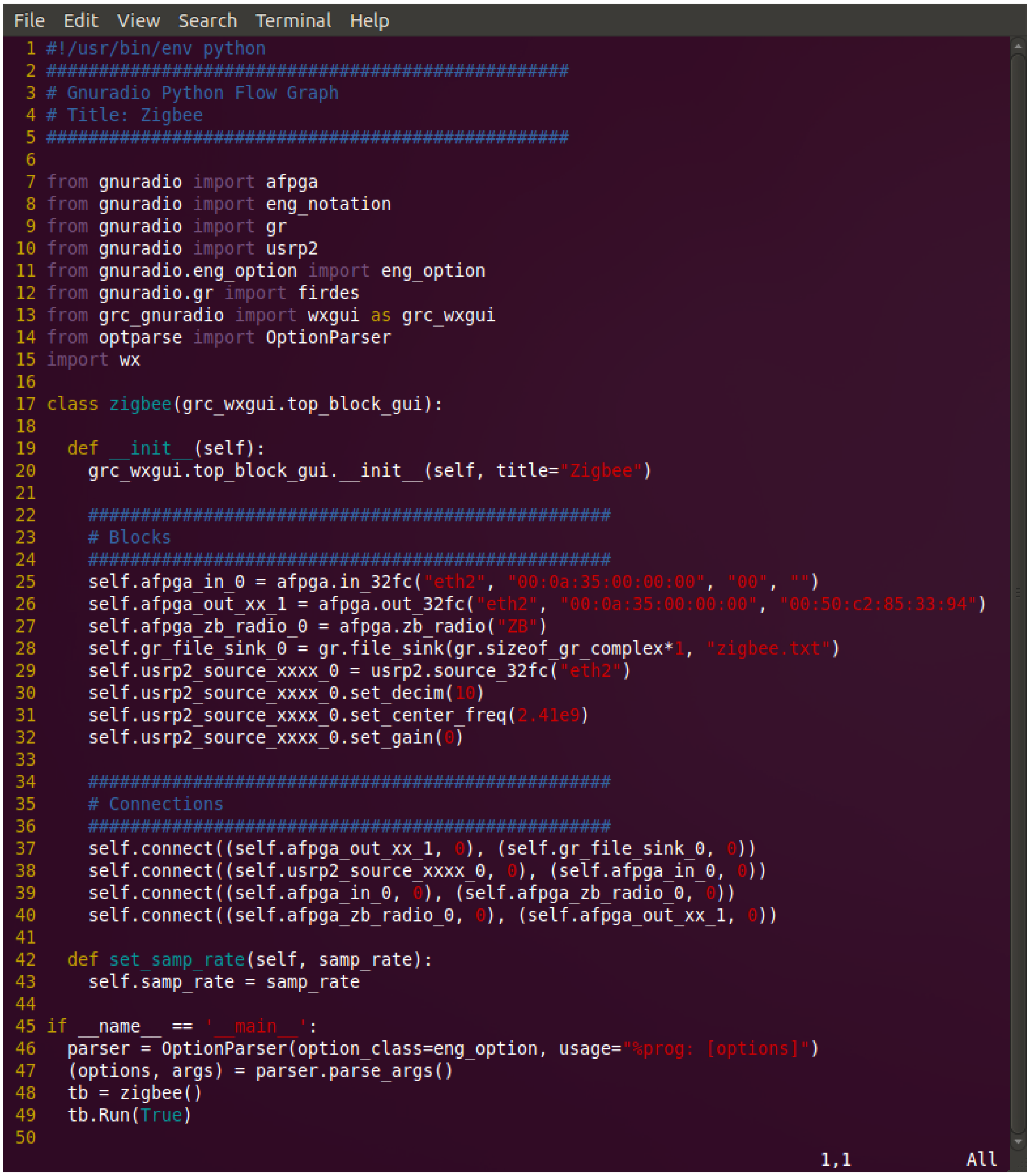
This chapter discusses the field of software-defined radios and the tools that are used to implement them in both software-only and hardware-accelerated models. Common hardware is discussed as well as projects that relate to their use. A focus on FPGAs and methods for their development is presented to give more information on the problems addressed in this thesis. Finally related works that drove this project are detailed.

2.1 SDR Software

Software Defined Radio and its benefits are already clearly documented [14] [34]. This section will review related issues regarding the software being used to develop them. Any radio system developed to run on a General Purpose Processor (GPP) will require some sort of hardware front-end to receive or transmit actual data. If a front-end is not available, test data can be read from a file or created with certain functions to implement the system before using it in a real-world application. This allows for the design and prototyping of a software-defined radio system without any additional hardware. In order to do this, two popular open-source design tools are available: GNU Radio and OSSIE. Simulink is another popular design tool but must be purchased from MathWorks.

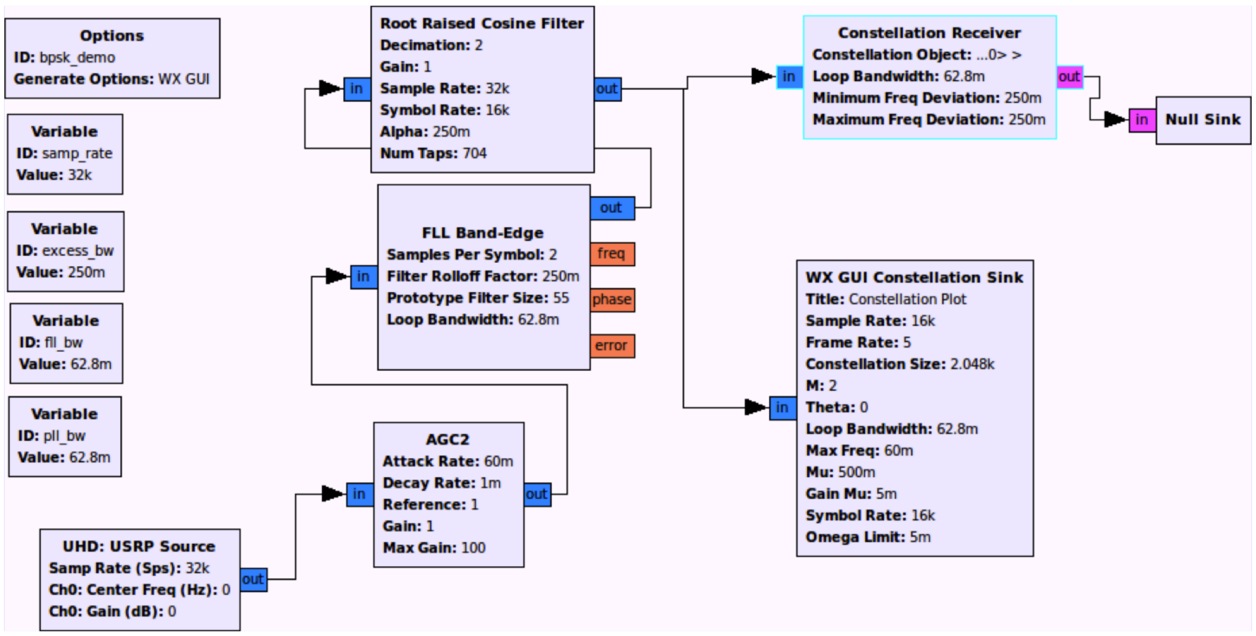
2.1.1 GNU Radio

GNU Radio is an extensive framework that enables many complex designs to be prototyped and built with only a GPP and off-the-self radio hardware. It is open source and commonly used due to its wide range of available radio blocks [7] and simple user interface. All of the signal processing is done in C++, a common language for software development, so that adding another custom block to GNU Radio is simple [1]. GNU Radio builds SDRs as flow graphs that can either be coded together in a Python script (Figure 2.1) or graphically wired together (Figure 2.2) with a tool called GNU Radio Companion (GRC) [8].



```
#!/usr/bin/env python
##################################################
# Gnuradio Python Flow Graph
# Title: Zigbee
##################################################

from gnuradio import afpga
from gnuradio import eng_notation
from gnuradio import gr
from gnuradio import usrp2
from gnuradio.eng_option import eng_option
from gnuradio.gr import firdes
from grc_gnuradio import wxgui as grc_wxgui
from optparse import OptionParser
import wx

class zigbee(grc_wxgui.top_block_gui):

  def __init__(self):
    grc_wxgui.top_block_gui.__init__(self, title="Zigbee")

    ##################################################
    # Blocks
    ##################################################
    self.afpga_in_0 = afpga.in_32fc("eth2", "00:0a:35:00:00:00", "00", "")
    self.afpga_out_xx_1 = afpga.out_32fc("eth2", "00:0a:35:00:00:00", "00:50:c2:85:33:94")
    self.afpga_zb_radio_0 = afpga.zb_radio("ZB")
    self.gr_file_sink_0 = gr.file_sink(gr.sizeof_gr_complex*1, "zigbee.txt")
    self.usrp2_source_xxxx_0 = usrp2.source_32fc("eth2")
    self.usrp2_source_xxxx_0.set_decim(10)
    self.usrp2_source_xxxx_0.set_center_freq(2.41e9)
    self.usrp2_source_xxxx_0.set_gain(0)

    ##################################################
    # Connections
    ##################################################
    self.connect((self.afpga_out_xx_1, 0), (self.gr_file_sink_0, 0))
    self.connect((self.usrp2_source_xxxx_0, 0), (self.afpga_in_0, 0))
    self.connect((self.afpga_in_0, 0), (self.afpga_zb_radio_0, 0))
    self.connect((self.afpga_zb_radio_0, 0), (self.afpga_out_xx_1, 0))

  def set_samp_rate(self, samp_rate):
    self.samp_rate = samp_rate

if __name__ == '__main__':
  parser = OptionParser(option_class=eng_option, usage="%prog: [options]")
  (options, args) = parser.parse_args()
  tb = zigbee()
  tb.Run(True)

                                                                          1,1           All
```

Figure 2.1: GNU Radio Python Flow Graph



Options
ID: bpsk_demo
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 32k

Variable
ID: excess_bw
Value: 250m

Variable
ID: fll_bw
Value: 62.8m

Variable
ID: pll_bw
Value: 62.8m

UHD: USRP Source
Samp Rate (Sps): 32k
Ch0: Center Freq (Hz): 0
Ch0: Gain (dB): 0

AGC2
Attack Rate: 60m
Decay Rate: 1m
Reference: 1
Gain: 1
Max Gain: 100

FLL Band-Edge
Samples Per Symbol: 2
Filter Rolloff Factor: 250m
Prototype Filter Size: 55
Loop Bandwidth: 62.8m

Root Raised Cosine Filter
Decimation: 2
Gain: 1
Sample Rate: 32k
Symbol Rate: 16k
Alpha: 250m
Num Taps: 704

Constellation Receiver
Constellation Object: ...0> >
Loop Bandwidth: 62.8m
Minimum Freq Deviation: 250m
Maximum Freq Deviation: 250m

Null Sink

WX GUI Constellation Sink
Title: Constellation Plot
Sample Rate: 16k
Frame Rate: 5
Constellation Size: 2.048k
M: 2
Theta: 0
Loop Bandwidth: 62.8m
Max Freq: 60m
Mu: 500m
Gain Mu: 5m
Symbol Rate: 16k
Omega Limit: 5m

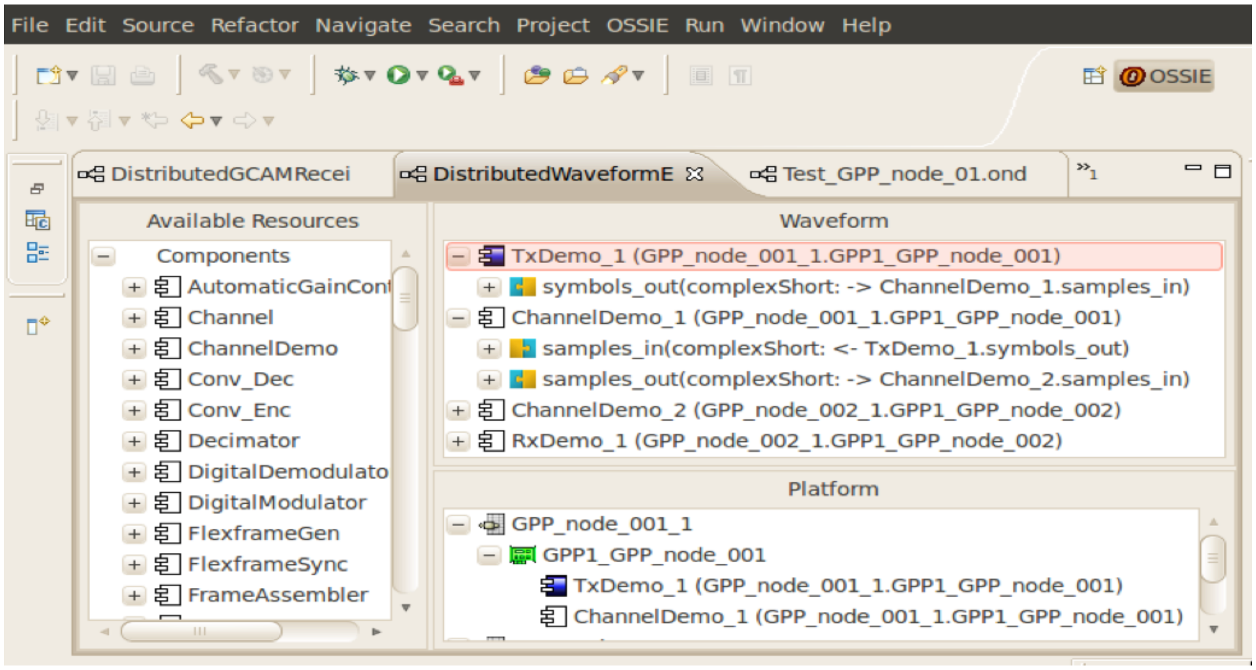
Figure 2.2: GNU Radio Companion Flow Graph

Each block runs a signal-processing task in a dedicated thread that passes data to other threads through shared memory buffers. The GNU Radio scheduler handles the threads and data with little overhead, but requires the blocks to be written in their format and only for GPPs. Currently the only supported off-loading of the signal in the flow graph is for transmission and reception of analog signals with a radio front-end. No other processing hardware can be added easily but GNU Radio does allow for rapid software-only prototypes.

2.1.2 OSSIE

The Open-Source SCA Implementation Embedded (OSSIE) platform was developed by Wireless@Virginia Tech in order to create an environment that allowed for the rapid development of SDRs [39]. It is available for free to anyone who wants to design SDRs and simplifies the experience of creating systems that approximate the Department of Defense's Software Communications Architecture (SCA). OSSIE is still being developed and used by Virginia Tech, the Office of Naval Research, Science Applications International Corporation, Tektronix, and Texas Instruments.

The underlying processing is written in C++ and new code can be added easily in Eclipse (Figure 2.3) to build custom SDRs because the communication standards are abstracted away from the designer. This abstraction occurs because SCA uses the Common Object Request Broker Architecture (CORBA) to manage data passing and type checking between components written in any language for any hardware. This allows OSSIE to design heterogeneous systems but the overhead from CORBA can quickly take over the available processing power. This has not stopped developers and the ability to provide rapid prototyping of SDR designs has made OSSIE a useful tool in the field.



```
Components
  AutomaticGainCon
  Channel
  ChannelDemo
  Conv_Dec
  Conv_Enc
  Decimator
  DigitalDemodulato
  DigitalModulator
  FlexframeGen
  FlexframeSync
  FrameAssembler

TxDemo_1 (GPP_node_001_1.GPP1_GPP_node_001)
  symbols_out(complexShort: -> ChannelDemo_1.samples_in)
ChannelDemo_1 (GPP_node_001_1.GPP1_GPP_node_001)
  samples_in(complexShort: <- TxDemo_1.symbols_out)
  samples_out(complexShort: -> ChannelDemo_2.samples_in)
ChannelDemo_2 (GPP_node_002_1.GPP1_GPP_node_002)
RxDemo_1 (GPP_node_002_1.GPP1_GPP_node_002)

GPP_node_001_1
  GPP1_GPP_node_001
    TxDemo_1 (GPP_node_001_1.GPP1_GPP_node_001)
    ChannelDemo_1 (GPP_node_001_1.GPP1_GPP_node_001)
```

Figure 2.3: OSSIE Flow Graph in Eclipse

2.1.3 Simulink & System Generator

Simulink is a tool produced by MathWorks that provides a graphical environment for designing a large range of time-varying systems, which includes SDRs [26]. It is useful for many processing tasks because it is integrated directly with MATLAB, a widely used numerical computing environment. Simulink can be used with System Generator for DSP to build Xilinx FPGA modules for SDR processing. System Generator for DSP can build a completely hardware based system, or a software/hardware co-processor system [42]. This makes these tools effective for prototyping and building complete designs, but both of them must have a purchased license every year to be used. They are both designed for so much more than SDR as well, so a radio designer must navigate through a complex system to produce a desired radio. These tools are also still limited by the lengthy compile times of FPGAs because they use the conventional Xilinx compilation processes. They are shown together in Figure 2.4.

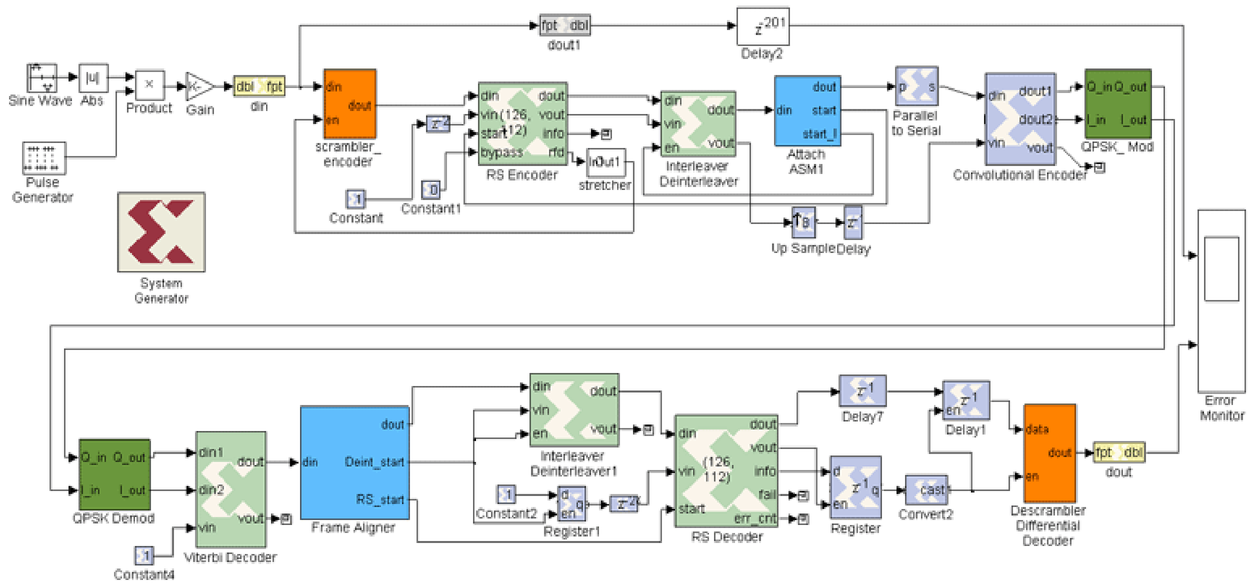


Figure 2.4: Simulink Flow Graph [26]

2.2 SDR Hardware

Despite being called software-defined radios, a real radio can never be completely built with only a processor. There must be an antenna and front-end capable of receiving a signal with an Analog to Digital Converter (ADC) and possibly doing some filtering to transform that signal into a format usable by a software processor. A Digital to Analog Converter (DAC) must be used in order to transmit a signal from an SDR. Work is being done to improve the bandwidth of these front-ends so that new hardware does not have to be added for every specific frequency range [19]. The more general antennas are more expensive and require major processing to extract usable data out of the received signals. A common way to handle this processing requirement is by adding a small amount of appropriate hardware back into the flow.

2.2.1 USRPs

One common hardware device for SDRs is a Universal Software Radio Peripheral (USRP). There are different iterations designed by Ettus Research [12] based on what a user needs. All of them are relatively inexpensive as far as hardware goes so they are common in research and among SDR hobbyists. With the use of replaceable daughter cards the hardware can be quickly changed to handle different frequencies for transmitting and receiving. They are well supported in the popular GNU Radio community and can be connected either by USB or

Gigabit Ethernet for more bandwidth. When used as receivers, the USRPs convert analog signals to digital signals and decimate them to appropriate sample rates before passing the signal to a host computer for additional signal processing. When used as transmitters, the USRPs perform the inverse of the functions described for reception.

The USRP is an example of a required hardware accelerator to provide an RF front-end. It contains a motherboard, multiple ADCs and DACs, and a million gate FPGA [6]. High-speed general-purpose operations are done on the embedded FPGA such as decimation, interpolation, and digital conversions [10]. By moving some processing to hardware, GNU Radio has already enabled many real-time radio prototypes [29].

2.2.2 DSPs and GPUs

Today many different signal processing fields are used to build different types of radios in different application settings. Many types of hardware exist that can be used to process signals but two of the most common ones are DSPs and GPUs.

Digital Signal Processors (DSPs) are a set of specific microprocessors that perform signal processing tasks well but are limited due to their specialization. They are optimized hardware designed for an explicit task. They enable low-power real-time signal processing in phones, modems, and audio systems [33].

Graphics Processing Units (GPUs) are relatively new to the field of radio signal processing. Their usefulness in processing large amounts of repetitive data for visual systems is now seen as applicable to other signal processing. They are composed of hundreds of processing cores [30] that make them perfect for implementing parallel pieces of radio protocols.

2.2.3 FPGAs

Although there are many powerful options in the field of signal processing, FPGAs provide a reconfigurable means of handling intense processing tasks. Field Programmable Gate Arrays (FPGAs) consist of a large set of hierarchically connected logic blocks that are reprogrammable to any implementation that can be created with a hardware description language. Their near infinite possibilities of use make them as advantageous as General Purpose Processors when it comes to creating multiple radios with only one piece of hardware; however, they take much longer to design for because hardware languages are not as abstract as software languages. Their compile times are also orders of magnitude larger than those of software. Where GPPs do everything in sequence, FPGAs can implement large parallel operations on all of their logic blocks at once. This presents another limitation as the amount of space on any given FPGA is always fixed. It cannot implement designs that require more logic than it has available, and any logic that is not used in a design is effectively wasted space. Despite these few limitations, FPGAs are huge players in the SDR field [11].

Major players in the field of FPGAs with SDR development are SORA and WARP. SORA is a Microsoft Research project that built their own specialized hardware system to support rapid development of SDRs [38]. Their in-house physical layer communication allows for high-throughput radio design with FPGAs and GPPs. The Wireless Open-Access Research Platform (WARP) developed at Rice University provides a configurable design platform for prototyping radios using FPGAs as one layer of their system [3]. Their system uses a library of pre-designed hardware and scales easily with the use of more boards; however, a method for developing new blocks for this system is not presented. Theoretically OSSIE could be used with any hardware, including FPGAs, which implemented CORBA; however, this is not common. These designs make use of the computational power of FPGAs but are still limited by the slow nature of their assembly tools.

2.3 Hardware Assembly

When FPGAs are used, a hardware description must be assembled into a bitstream that actually programs a chip. This is a multi-stage process that has been implemented many different ways, but all of which take a large amount of time.

2.3.1 Xilinx & Altera

Xilinx produces a set of FPGA families from high-end Virtex to low-end Spartan boards. They are the current leading seller of FPGAs [41]. Their flow involves moving from a generic hardware description language, to the EDIF format as the design becomes family specific, to XDL as the design is placed and routed for a specific board, and finally to a bitstream before being programmed on the chip. Each of these steps produces a file that is exposed to the designer and can be modified before the next step is run.

Altera produces the Cyclone, Arria, and Stratix family of FPGAs (from [2]). Their flow involves the same basic steps as Xilinx but is not open to the designer to modify along the way. Altera does have a toolkit called QUIP that allows for some customization of the flow, but this tool is not effective. Instead a project is created and any adjustments must be done at the beginning with their set of options. Custom tools cannot interact with Altera FPGAs at this time.

Both FPGA assembly methods are time consuming but are effective in optimizing a design for user constraints. Xilinx and Altera are moving towards System-C and C/C++ environment tools that allow for even more abstract description languages as input for designs. Currently their tools still take orders of magnitude longer than software to compile.

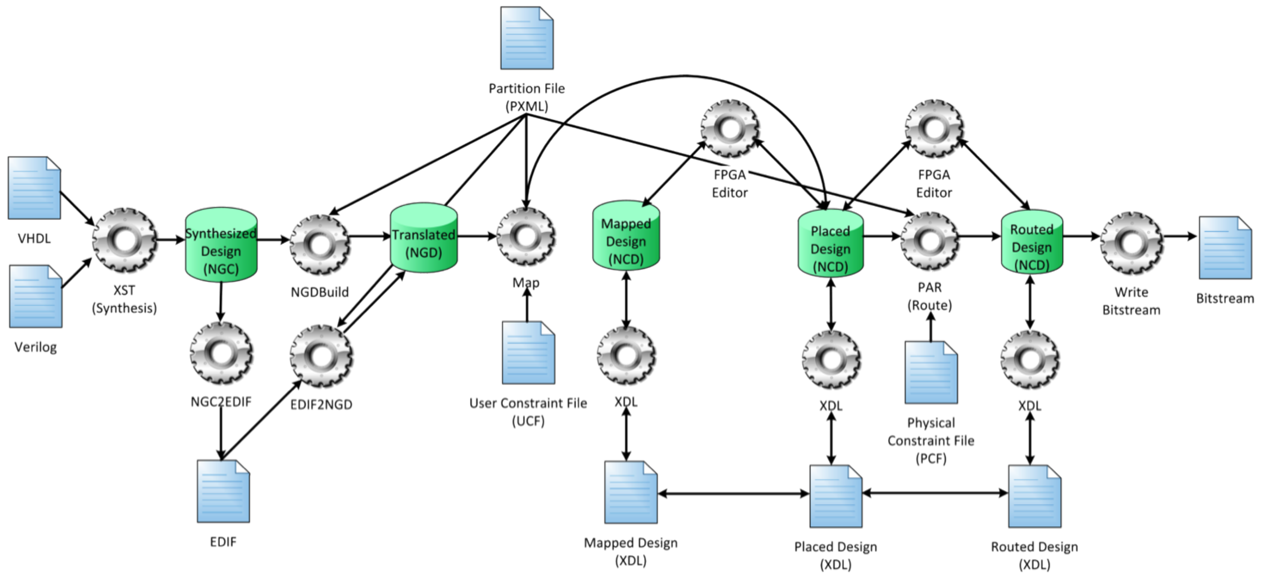


Figure 2.5: Xilinx Tool Chain Flow

2.3.2 Partial Reconfiguration

Hardware is often not assembled based on what has changed in a design; the whole system is rebuilt and optimized from start to finish. This is a major reason designs take so long to iterate through. A method that attempts to solve this major problem in slow assembly times is partial reconfiguration. Partial reconfiguration partitions the FPGA into two regions: a persistent region and a sandbox region. The persistent region cannot change and holds pieces of the hardware that will remain between designs. The sandbox region can be reprogrammed with a new partial bitstream without affecting the persistent region [22].

Unfortunately the sandbox does not have any flexibility, it must be completely programmed every time and still has to go through the slow process of optimization before it can be used. The only added value is that it programs the chip faster and while it is still running. This is useful for SDRs that want to change from one demodulation scheme to another quickly and without turning a radio off, but the design time still remains too high. The process of building a partial bitstream is also still complex as both Xilinx and Altera are only beginning to come out with useful tools for the job.

2.3.3 TORC

The Tools for Open-source Reconfigurable Computing (TORC) are useful for user manipulation of EDIF and XDL files as well as bitstream packets [36]. This software tool suite provides a C++ library for abstracting the process of editing the Xilinx flow in Figure 2.5. Although this is not technically a method of assembly, an intricate knowledge of the Xilinx flow allows the custom tools discussed in Chapter 5 to use TORC to assemble a working bitstream. If Altera follows suit and opens their flow, TORC could be used to do the same thing for their FPGAs.

2.4 Related Work

GNU Radio has seen many iterations and enhancements throughout its development. As a community there are constantly software processing blocks that are being produced and new methods of implementing radios faster are being shared. Developments more concerned with this thesis are those that have attempted to add hardware to a largely software based design environment.

2.4.1 GRGPU

GRGPU is a system that adds GPUs to the GNU Radio environment [32]. The work shows one way that hardware accelerators can be used to enhance the rapid prototyping of GNU Radio. It successfully implements blocks that run on a CUDA-based Nvidia card alongside GPP blocks in the same flow graph. This is accomplished by creating a separate scheduler that runs with the GNU Radio scheduler to handle data transfer between the two devices. This limits the speed at which the GPU can run but does show improvements in speed with large samples that overcome a fixed latency penalty. The GRGPU project is a good first step in the direction of adding much needed hardware functionality to the GNU Radio framework.

2.4.2 FPGA-Based Accelerators

Many projects have been developed to make use of FPGAs for software-defined radio acceleration. These projects include the Kansas University Agile Radio [27], the Japanese National Institute of Information and Communications Technology SDR Platform [18], and the Berkeley Cognitive Radio Platform [28] to name a few. These implementations all required using special boards and software for communication rather than augmenting an already available system. They were also limited by the slow compile times of FPGAs, although their gains in performance were seen as worth this cost.

FPGAs have not been added to stock GNU Radio yet as "hardware is strictly not part of GNU Radio" [7]. There is an FPGA on the USRP devices that can be modified [9] for custom applications though. With the recent increase in FPGA size on the newer devices these customizations have become more popular. Unfortunately the changes made are permanently left on the board and run before every flow graph receiving data from the USRP. All flexibility is lost and their operation is not presented to the user in the flow graph.

2.4.3 Enhancing GNU Radio for Hardware Accelerated Radio Design

Previous work at Virginia Tech by Charles Irick [21] was done to develop a block in GNU Radio that could communicate with an FPGA. This work allowed a designer to add hardware to an external FPGA that was not on a USRP and still receive data from the device in a GNU Radio flow graph. The same raw networking techniques used by a USRP were implemented so that GNU Radio could interface with the FPGA without a custom controller. The USRP code was modified slightly so that data could be sent directly to the FPGA without originating from the host machine. This lowered the requirements of the network bandwidth but meant a new bitstream had to be placed on the USRP before it was usable in the system. Also the FPGA could not be programmed from GNU Radio, although this was planned. This meant that designers could still not see what was being placed on the external FPGA and the only benefit over using the USRP's FPGA was that the designer had complete control over the chip being used. A new class for the auxiliary FPGA was created as a controller for input and output to GNU Radio. This class was labeled *afpga* and was designed to work with AgileHW [4] to allow for quicker FPGA designs and thus continue the rapid prototyping of GNU Radio.

Chapter 3

GNU Radio Enhancements

This chapter discusses the changes made to GNU Radio and how additional software is integrated into it to facilitate the building of attached hardware. GNU Radio maintains all of its previous functionality but slight changes to the system are outlined to show how different sections interact. The design decisions are also presented so that future work can build upon the direction these enhancements intended.

3.1 GNU Radio FPGA Extension Class

GNU Radio organizes their software blocks into classes. There are separate classes for filters, operators, input/output, converters, and more. The entire structure is set up to allow for the easy addition of new processing blocks simply by adding another class. The classes inherit all of the necessary block information from GNU Radio and thus never create new dependencies within the rest of the system. This same model was used to develop a hardware class that would fit alongside all of the other blocks.

As mentioned in Chapter 2, prior work was done to develop an auxiliary FPGA (*afpga*) class for GNU Radio [21]. This class contained an *afpga_source* and *afpga_sink* that mimicked the behavior of a USRP2 source and sink using raw Ethernet frames. The only difference was that an *afpga_sink* sent a packet to a connected USRP2 that redirected the stream of data to the FPGA.

This paradigm was changed because it did not allow for the user to have control over how hardware was added to the FPGA. The prior work assumed that the only source of data was a USRP2 streaming to an FPGA. It also assumed that data would always be handled sequentially by the FPGA and then returned to the GNU Radio host as decoded ASCII.

These legacy blocks evolved into the new *afpga_in* and *afpga_out* blocks of the current system. The *afpga_in* block replaced the *afpga_sink* block. Instead of presenting data as disappearing

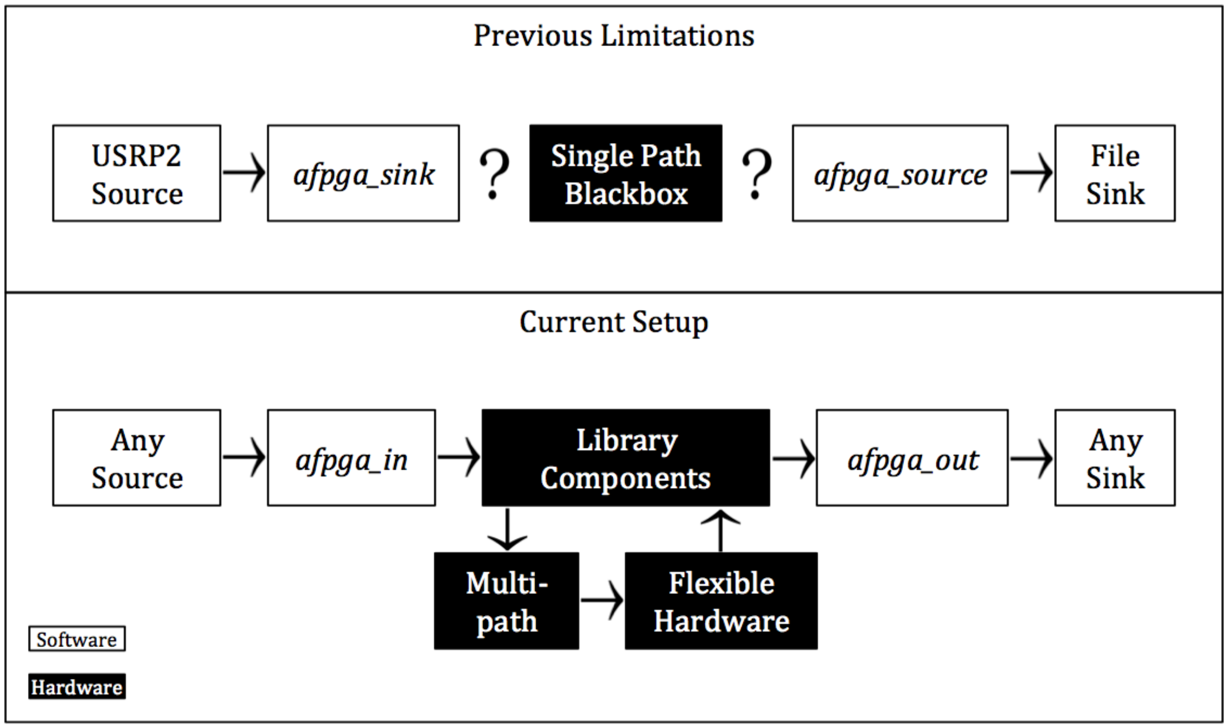


Figure 3.1: Previous vs. Current *afpga* Model

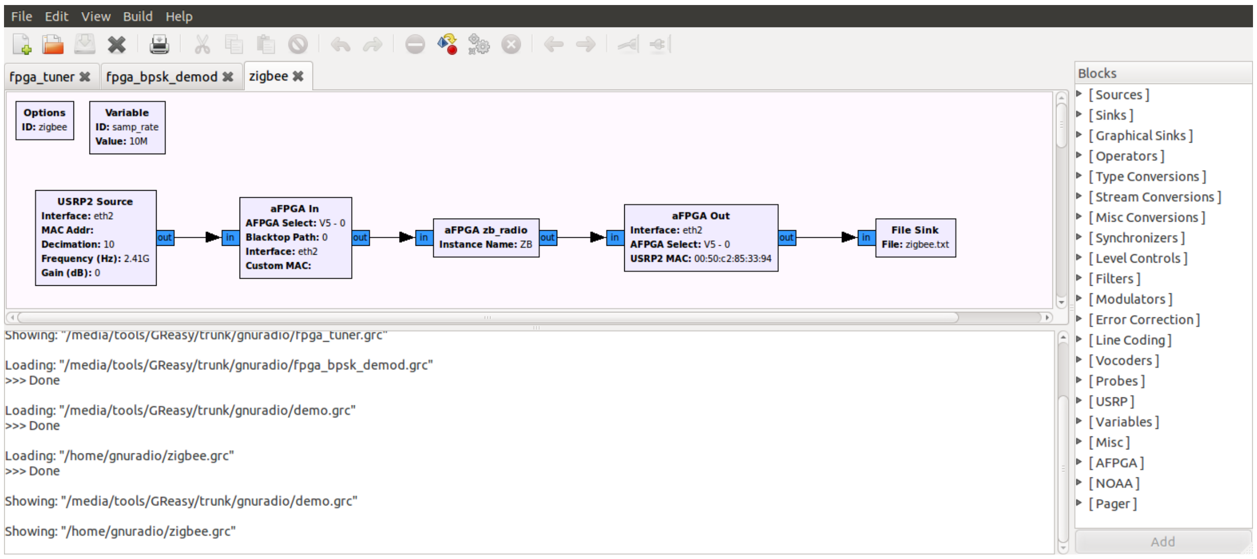
into a sink, data is now presented as entering an FPGA. The new block enables GNU Radio data from the computer to be sent to the FPGA as well as from the USRP2 and even from other FPGAs.

The *afpga_out* block has replaced the *afpga_source* block. The data are now represented as the output of a controllable system rather than some unknown source. The ability to pipe the ASCII data to a readable file has been added to the block. Also the standard GNU Radio complex I/Q data format is included as an output type so that further processing can occur on the host if desired.

The real shift in how hardware is represented in GNU Radio comes with the addition of some signal processing *afpga* blocks. Instead of having an uncontrollable or single path hardware setup, each module gets its own block that must be placed and connected within GNU Radio. By separating these blocks from a forced sequential flow the designer is free to build hardware in any way that can be imagined. Multiple paths can now exist and modules can talk to one or more other blocks that may be connected to it. This new scheme lends itself to improved prototyping because it now has hardware blocks that are free to operate the same way GNU Radio already treats software blocks. Presenting the hardware to radio designers in a familiar and flexible fashion is only the first step to enhancing this SDR environment.

3.2 GNU Radio Companion

GNU Radio can be run with additional software known as GNU Radio Companion (GRC) (Figure 3.2) that uses XML descriptions of blocks (Figure 3.3) to create a visual representation of a flow graph [8]. These XML blocks must be created by the user to represent their C++ counterparts. GRC uses a flow graph built with its visual interface to create a Python script that runs GNU Radio. Since GRC is not actually running GNU Radio, some manipulation can be done in the XML code in order to represent multiple blocks and all of their parameters with only one visual block. GRC does not change any of the original functionality but represents the available blocks in an organized way and includes graph checking to make sure that there are no empty connections or sinks with multiple sources. XML descriptions have now been made for all of the *afpga* hardware blocks as well.



Options
ID: zigbee

Variable
ID: samp_rate
Value: 10M

USRP2 Source
Interface: eth2
MAC Addr:
Decimation: 10
Frequency (Hz): 2.41G
Gain (dB): 0

aFPGA In
AFPGA Select: V5 - 0
Blacktop Path: 0
Interface: eth2
Custom MAC:

aFPGA zb_radio
Instance Name: ZB

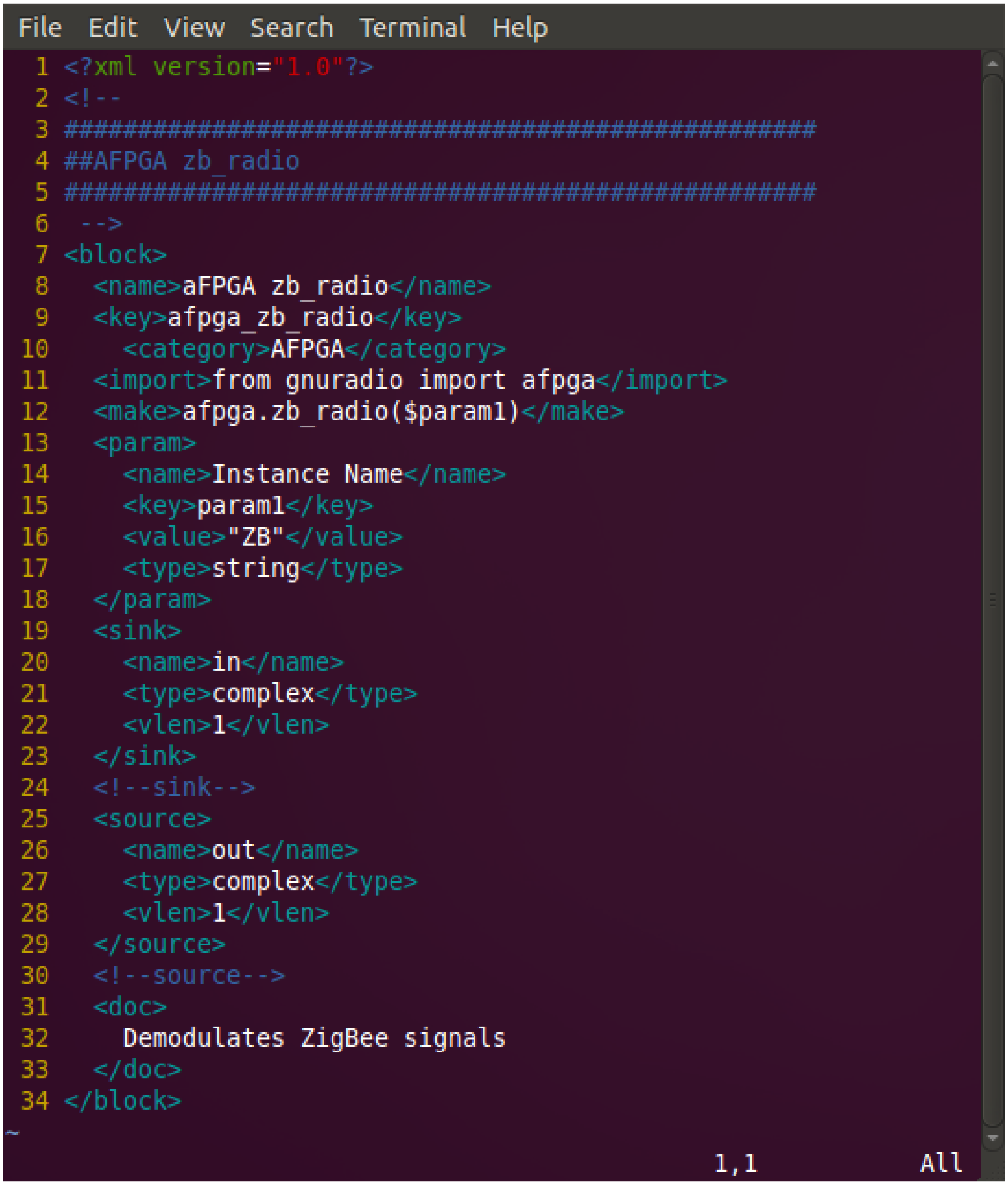
aFPGA Out
Interface: eth2
AFPGA Select: V5 - 0
USRP2 MAC: 00:50:c2:85:33:94

File Sink
File: zigbee.txt

```
Showing: "/media/tools/GReasy/trunk/gnuradio/fpga_tuner.grc"

Loading: "/media/tools/GReasy/trunk/gnuradio/fpga_bpsk_demod.grc"
>>> Done

Loading: "/media/tools/GReasy/trunk/gnuradio/demo.grc"
>>> Done

Loading: "/home/gnuradio/zigbee.grc"
>>> Done

Showing: "/media/tools/GReasy/trunk/gnuradio/demo.grc"

Showing: "/home/gnuradio/zigbee.grc"
```

Figure 3.2: Sample GRC Flow Graph with Hardware

Since the *afpga_in* block is not a sink anymore, it does not terminate but presents a connection that should be tied to the blocks that are going to be placed on an FPGA. In the same way, *afpga_out* now has an input connection that expects an FPGA block. FPGA processing blocks have input ports and output ports that match what is seen in a Verilog module. The clock and reset ports are excluded and automatically connected later. With GRC a completely visual representation of the software and hardware can be built. By enabling a visual component, prototypes can be presented for the designers in a clearer way. If a designer does not want to use GRC, they can continue using the GNU Radio Python scripts without losing any functionality of this work.



```
<?xml version="1.0"?>
<!--
###################################################
##AFPGA zb_radio
###################################################
 -->
<block>
  <name>aFPGA zb_radio</name>
  <key>afpga_zb_radio</key>
    <category>AFPGA</category>
  <import>from gnuradio import afpga</import>
  <make>afpga.zb_radio($param1)</make>
  <param>
    <name>Instance Name</name>
    <key>param1</key>
    <value>"ZB"</value>
    <type>string</type>
  </param>
  <sink>
    <name>in</name>
    <type>complex</type>
    <vlen>1</vlen>
  </sink>
  <!--sink-->
  <source>
    <name>out</name>
    <type>complex</type>
    <vlen>1</vlen>
  </source>
  <!--source-->
  <doc>
    Demodulates ZigBee signals
  </doc>
</block>
~
                                                  1,1           All
```

Figure 3.3: Sample XML Block

3.3 Core Code Changes

The changes to *afpga_in* and *afpga_out* for GRC are purely visual. Those modules are still sources and sinks within the core of GNU Radio. The FPGA processing blocks are not executed on the host machine so the data does enter a blackbox as far as the processor is concerned. In order to make this work the GNU Radio scheduler is ignored for these blocks. When two blocks are connected from a GNU Radio script, the function gr_flowgraph::connect(source, destination) is run. This function is part of 'gr_flowgraph.cc' that has been modified as seen in Listing 3.1.

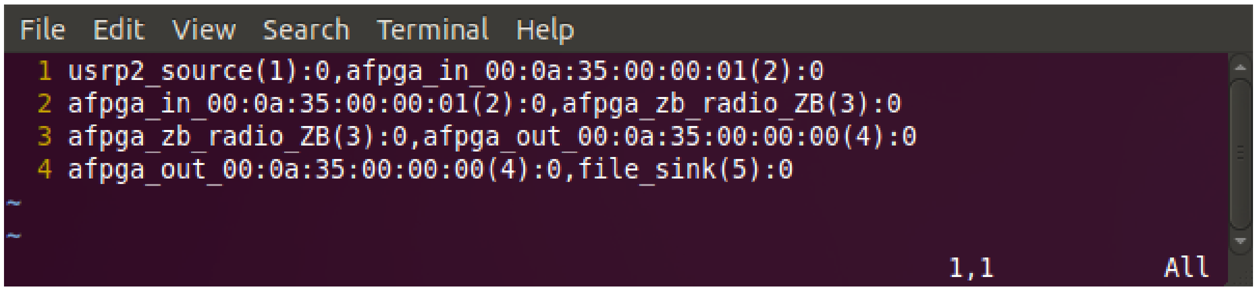
Listing 3.1: Connect Function of gr_flowgraph

```
void
gr_flowgraph::connect(const gr_endpoint &src, const gr_endpoint &dst) {
  if(src.block()->name().compare(0,5,"usrp2") == 0 && dst.block()->name
     ().compare(0,5,"afpga") == 0){
  // Do not connect the usrp2 block to afpga blocks (forcing both not
     to run past the constructor)
  }
  else if(src.block()->name().compare(0,5,"afpga") == 0 && dst.block()
    ->name().compare(0,5,"afpga") == 0){
  // Do not connect afpga blocks to afpga blocks (their connections are
      handled later in gr_top_block_impl)
  }
  else{
    check_valid_port(src.block()->output_signature(), src.port());
    check_valid_port(dst.block()->input_signature(), dst.port());
    check_dst_not_used(dst);
    check_type_match(src, dst);

    // All ist klar, Herr Kommisar
    d_edges.push_back(gr_edge(src,dst));
  }

  //If an afpga block is the source or destination, log the connection
     in fpga_connections.txt
  if(src.block()->name().compare(0,5,"afpga") == 0 || dst.block()->name
     ().compare(0,5,"afpga") == 0){
    std::ofstream myfile;
    myfile.open ("fpga_connections.txt", std::ios::app);
    myfile << src << "," << dst << "\n";
    myfile.close();
  }
}
```

This does a check to make sure the modules have the same type and are not already connected. Once this is done the modules are added to the scheduler to be run later. Since no data is being sent through the hardware blocks, they are not scheduled. The process for ignoring the FPGA blocks is currently rudimentary: if both the source and the sink contain *afpga* in the name, then they are not scheduled. Also if a USRP2 block is connected to an *afpga* block, neither are scheduled. The USRP2 was modified to send directly to an FPGA, which clears an overload of traffic to the host machine. This means that the block on the host is no longer doing any processing, which is why it too is no longer scheduled.

Work with GNU Radio developers is currently underway to improve this system. The idea is that, instead of inheriting a basic block, FPGA blocks will inherit a hardware type block that is then not scheduled. The principle is still the same but the implementation will be more robust and no longer require all of the FPGA blocks to have *afpga_* in front of their name. It will also allow blocks that may run on other hardware to also avoid being scheduled without needing a specific name.

If an FPGA block connects to another FPGA block, since GNU Radio is no longer recognizing this connection, it is logged in 'fpga_connections.txt'. These connections will be used to build a netlist and eventually a bitstream for every FPGA in the flow graph.



```
usrp2_source(1):0,afpga_in_00:0a:35:00:00:01(2):0
afpga_in_00:0a:35:00:00:01(2):0,afpga_zb_radio_ZB(3):0
afpga_zb_radio_ZB(3):0,afpga_out_00:0a:35:00:00:00(4):0
afpga_out_00:0a:35:00:00:00(4):0,file_sink(5):0
~
~
                                                    1,1           All
```

Figure 3.4: Sample fpga_connections.txt

3.4 Runtime Built Netlist

GNU Radio does not run the unscheduled hardware blocks, but it does call their constructor functions. Each block's constructor opens a file called 'edif.dat' and appends one line of information to it (Listing 3.2). This line is a condensed description of the ports available to the Verilog module (Listing 3.3). The file is used by the program EdifWriter, shown in Appendix A, to build a completely Xilinx compatible EDIF file (Listing 3.4). EdifWriter is part of the qFlow package that will be discussed further in Chapter 5. The information that is written by each block contains a unique name for that block as well as all of the port information. The ports that are buses are described as ARRAYS with a certain DIRECTION, NAME, and SIZE. The ports that are only one bit are described as PORTS with just a

DIRECTION and NAME. This means that most data lines are described as ARRAYS and the clock/reset lines are usually PORTS.

Listing 3.2: Sample Module edif.dat Line

```
Cell;zb_radio;ZB3;Array;output;out;33;Array;input;in;33;Port;input;rst;
   Port;input;clk;
```

Listing 3.3: Verilog of Sample Module

```
module zb_radio (
  output reg [32:0] out,
  input [32:0] in,
  input rst,
  input clk
);
```

Listing 3.4: Compiled EDIF of Sample Module

```
(cell zb_radio
  (cellType GENERIC)
    (view view_1
      (viewType NETLIST)
      (interface
        (port
          (array (rename out "out<32:0>") 33 )
          (direction OUTPUT)
        )
        (port
          (array (rename in "in<32:0>") 33 )
          (direction INPUT)
        )
        (port rst
          (direction INPUT)
        )
        (port clk
          (direction INPUT)
        )
      )
    )
)
```

Within the framework of GNU Radio, all of the blocks are created, then connected, and then run. Their creation yields 'edif.dat' and their connections generate 'fpga_connections.txt'. After all of this is done, the flow graph is started. This normally calls the start function on all of the blocks that have been scheduled, but code was inserted just before this step to assemble and program the necessary FPGAs first. The code was placed into a separate C++ file called 'edif_connector.h' so that modifications could be made to it without disrupting more of the GNU Radio core.

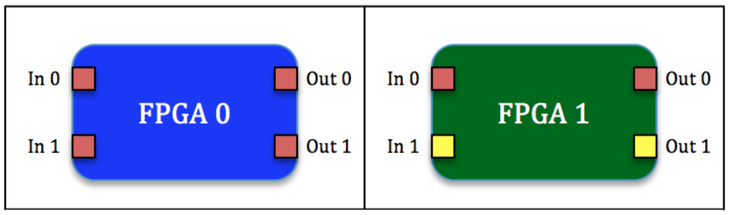
Listing 3.5: Excerpt of gr_top_block_impl.cc with a Two Line Change to Call connect_edif()

```cpp
void
gr_top_block_impl::start()
{

  ...

  if (!connect_edif())
    throw std::runtime_error("gr_top_block_impl: The AFPGA section of
       the flow graph could not be connected!");

  ...

}
```

This connect_edif() function shown completely in Appendix B starts by cleaning up 'fpga_connections.txt'. This means clearing duplicates and moving valid information to a file called 'connections.txt'. Since 'fpga_connections.txt' is appended to, it has to be deleted after each run or the next run will contain all of the information from both runs. The 'connections.txt' file stays between runs so it can be used for debugging purposes. The information contained in it is already in memory, so it can be removed if it is no longer desired.

Using this connection information, the function builds a netlist for each FPGA. The different FPGAs are identified by different base MAC addresses associated with the *afpga_in* and *afpga_out* blocks. The format of the MAC address can be seen in Figure 3.5. The last two bytes of the MAC address are used for routing between paths inside of one FPGA and are not used once the path information is extracted. Once the ends of the FPGAs are discovered, their connections are followed to determine what is placed on each FPGA. As each new block going on the FPGA is found, its CELL information from 'edif.dat' is retrieved and placed into a new FPGA-specific file. The new file is simply called 'edifMAC.dat', where MAC is the hexadecimal value of the base MAC address for a given FPGA (e.g. 'edif000a350001.dat').

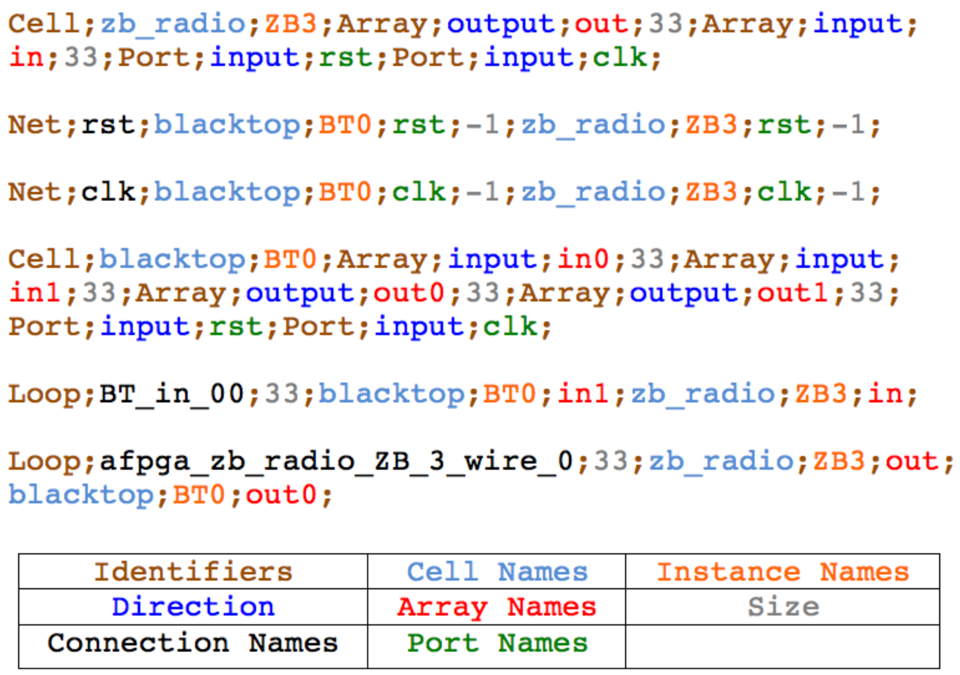
After all of the block's CELL information is added to the file, the connection information is added. The connections are denoted as either NET or LOOP. A NET is a one bit connection that declares a NAME for itself and then points to the bits that it is connecting. The bits



Standard Header	FPGA #	Path #
00:0A:35:00	01	01

Figure 3.5: MAC Address Example

are identified by pointing to a CELL with a certain INSTANCE and then to the desired PORT. If the single bit being connected is part of an ARRAY of wires, then the INDEX is also given. Otherwise a negative one INDEX tells that code that the PORT is only one bit and should be treated as such. If more than one bit should be connected, a LOOP is used instead. A LOOP also starts by declaring a NAME but includes a SIZE to make sure that everything connecting to it has the same width. After that, all of the wires are identified by pointing to a CELL with a certain INSTANCE and then to the desired ARRAY.



```
Cell;zb_radio;ZB3;Array;output;out;33;Array;input;
in;33;Port;input;rst;Port;input;clk;

Net;rst;blacktop;BT0;rst;-1;zb_radio;ZB3;rst;-1;

Net;clk;blacktop;BT0;clk;-1;zb_radio;ZB3;clk;-1;

Cell;blacktop;BT0;Array;input;in0;33;Array;input;
in1;33;Array;output;out0;33;Array;output;out1;33;
Port;input;rst;Port;input;clk;

Loop;BT_in_00;33;blacktop;BT0;in1;zb_radio;ZB3;in;

Loop;afpga_zb_radio_ZB_3_wire_0;33;zb_radio;ZB3;out;
blacktop;BT0;out0;
```

Identifiers	Cell Names	Instance Names
Direction	Array Names	Size
Connection Names	Port Names	

Figure 3.6: Sample edifMAC.dat with Highlighted Structure

Every CELL is automatically given a NET that connects the clock and a NET that connects the reset to a global clock and reset respectively. If a NET or LOOP has the same name as another NET or LOOP on a different line, they will be automatically concatenated by EdifWriter. This allows for multiple lines containing the same clock and reset information, but can be used to connect a complicated wire set in the future.

It is necessary to know that these 'edifMAC.dat' files contain all of the same information as an EDIF file but are presented in an easily writable, readable, and compact format. EdifWriter parses this information and uses TORC to build the more complex netlist. Although these files are automatically generated and run, they can be modified or written from scratch by any user who wishes to describe a netlist in a simpler format before running the Xilinx or qFlow tools, which require an EDIF.

3.5 Fast Bitstream Creation

Once a connections list is built, one or more bitstreams must be generated in order to actually program any of the FPGAs. The remainder of the code mostly calls external scripts to accomplish this goal. Scripts were used for two reasons. The first is that the tools being used are still in active development as a separate project. It would be impossible to incorporate them into the core of GNU Radio. The second reason is that these scripts are easily modifiable and can be run outside of the GNU Radio framework to interface with the tools. This allows any user or program to take an 'edif.dat' file and run through the process of putting a bitstream on an FPGA. Also if GNU Radio fails, the process can be picked back up from the scripts without running everything again.

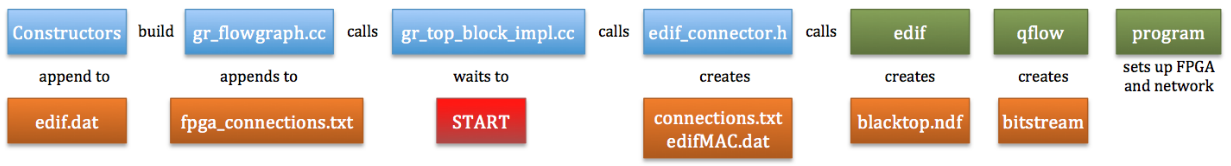


Figure 3.7: File Order and Operations

The first script is called 'edif', which is shown in Appendix C. The script takes one input: the base MAC address of the FPGA it is building. This calls EdifWriter on the respective 'edifMAC.dat'. Once a true EDIF is created, a checksum is generated with crc32 [25] to represent the contents of the FPGA. This checksum is stored on the FPGA so that it can be requested later to determine if anything has changed. If the FPGA has never been programmed, then there is no checksum on it and the next script will be called. If it has been programmed before with the exact same netlist, in the case where GNU Radio is run twice with only software changes occurring, then the rest of the scripts are not run for this FPGA.

The second script is called 'qflow', which is shown in Appendix D. This script actually builds a bitstream using a rapid modular based assembler called qFlow [16]. The same script can be modified to call tFlow [24], which does bitstream manipulation instead and yields the same result much faster. There are limitations to both systems, which will be discussed in Chapter 5. The script could also run the original Xilinx tools commonly used today to produce an optimal bitstream at the cost of a long run time. No matter which process is run, the final output of the 'qflow' script is a bitstream for an FPGA.

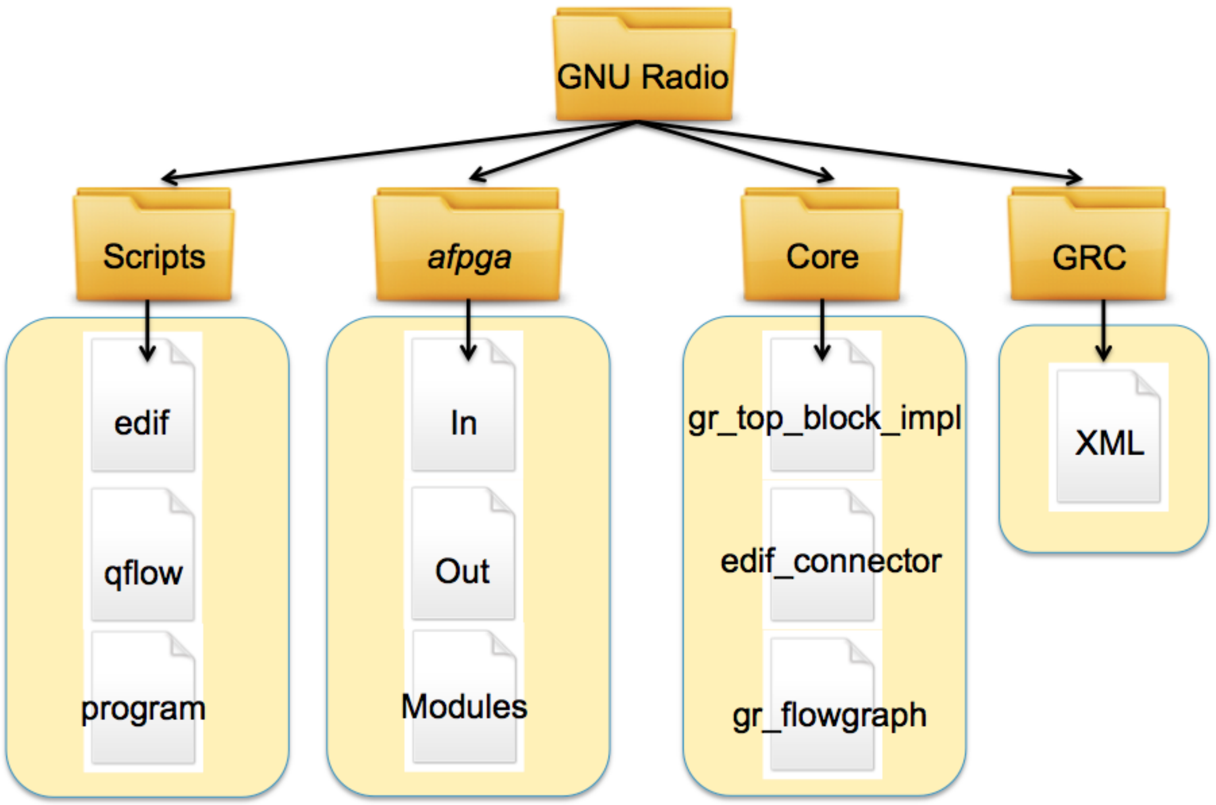


Figure 3.8: File Structure of GNU Radio with *afpga* Additions

The final script called 'program', which is shown in Appendix E, places the bitstream that was just built onto the appropriate FPGA. Currently this is accomplished using a tool called Impact provided by Xilinx from the command line. There is code set up to program the FPGA over an Ethernet connection instead but this has not been integrated or tested yet.

Once every script has been run, the code moves on to another FPGA if one exists. All of the scripts are run again for each FPGA. The final step is sending control data to the FPGAs to tell them where they should direct data. This control data, along with the checksum data from earlier, is sent using raw Ethernet packets using the code in 'PacketCreator.h' and

'RawEthernet.h', which are included in Appendices F and G respectively. Most commonly the data is sent back to the host machine but it can be directed to another FPGA or USRP2 or any other system that is listening on the network. When all of the operations in the connect_edif() function are completed successfully, it returns a true value allowing the rest of the program to start the software blocks. If the program detects errors it will stop the flow graph and throw a runtime error. The function can also determine that there are no appropriate FPGA blocks to run and allow GNU Radio to run normally with only software blocks.

Chapter 4

Hardware Developments

To facilitate the easy integration of hardware with GNU Radio, certain steps are taken on the hardware end to manage communication. By organizing the hardware in a certain way, the time it takes to assemble is significantly reduced. These steps are discussed in this chapter. The most beneficial modification to the hardware is the segregation of static and dynamic regions on the FPGA.

4.1 Static Region

There are a few core modules that are necessary for every design to communicate with GNU Radio. These make up the static region, which never changes and thus do not have to be rebuilt every time. This saves a large amount of time and is similar to the concept of partial reconfiguration where only a section of the FPGA is reprogrammed. The difference in this system is that a whole bitstream is still built and programmed, but it is done faster with a custom assembler that integrates the static and dynamic regions.

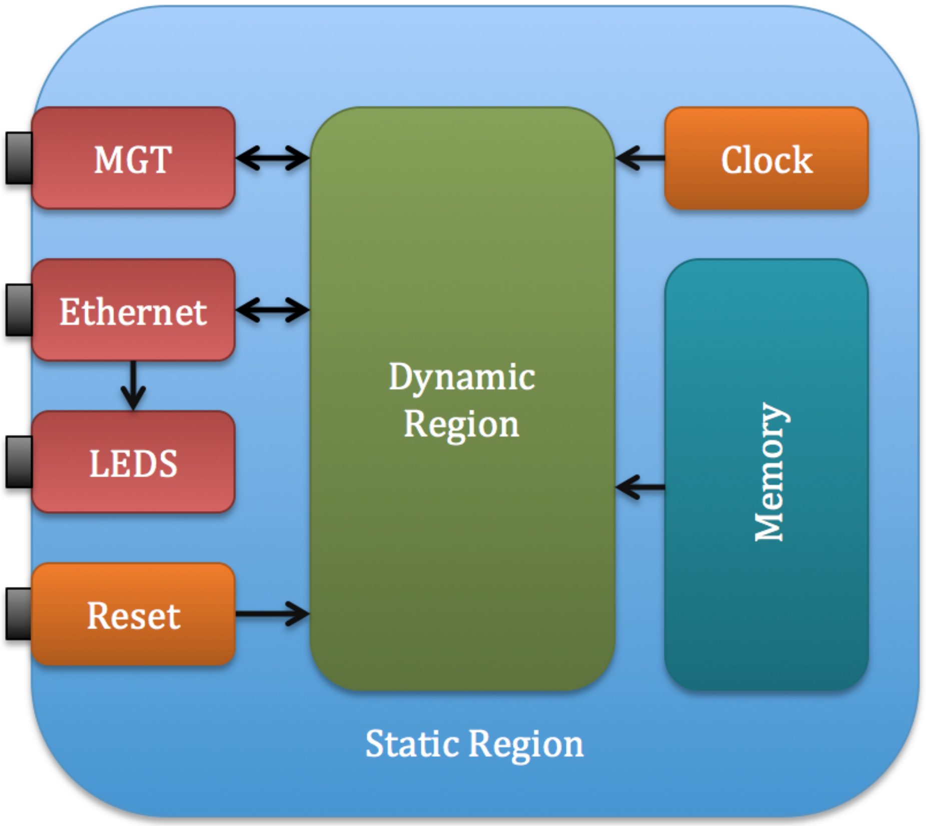
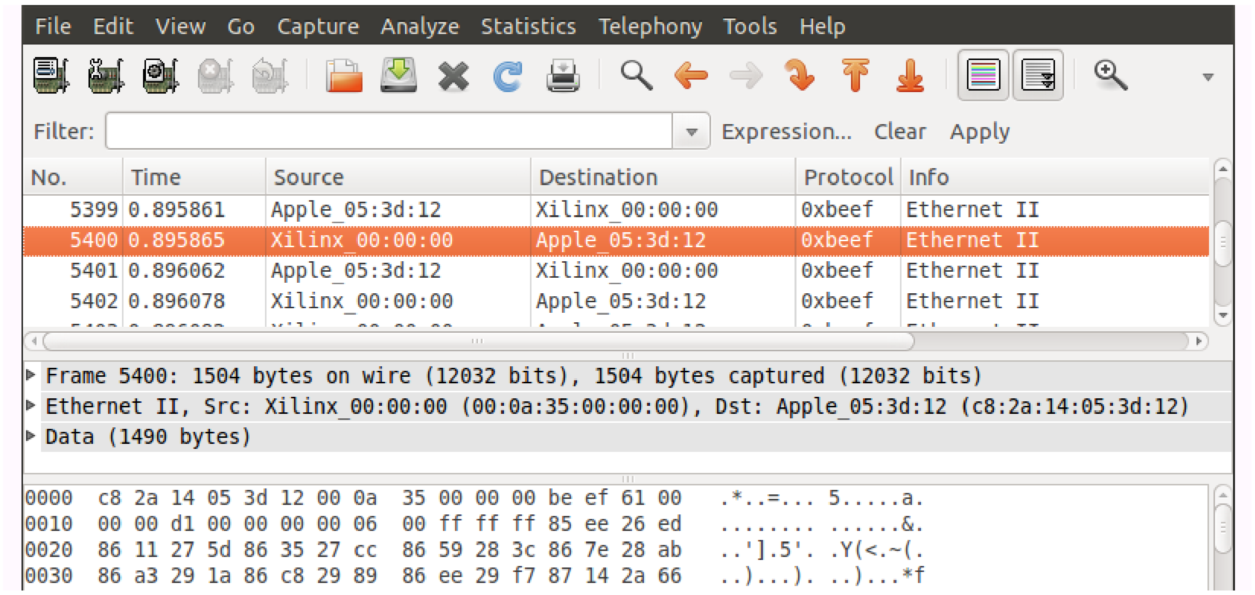


Figure 4.1: Static with I/O buffers, Ethernet and Dynamic Region

The static region interfaces with I/O buffers that cannot be accessed by the custom assembler. If part of the dynamic region needs access to an LED or communication peripheral, it has to pass through the static region. This requires that the static region decode data from the Ethernet or Multi-gigabit Transceivers (MGTs) and present them to the dynamic region as usable data.

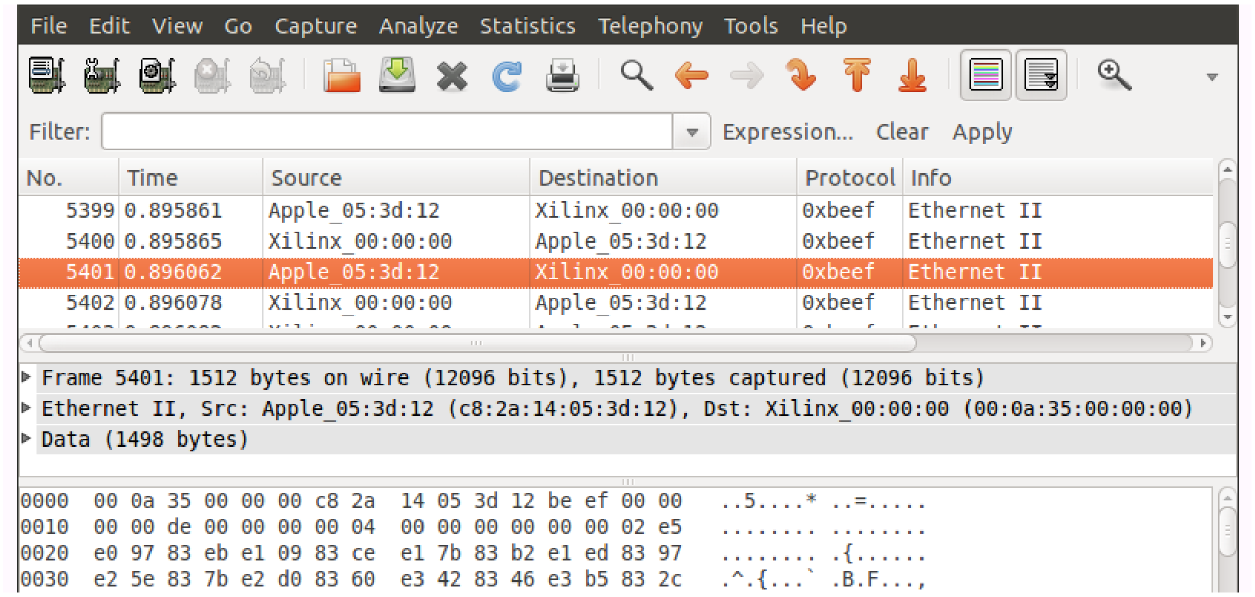
The first interface that is managed this way is the Ethernet. The Ethernet is used to pass data as well as control many of the functions on the FPGA. The majority of the interfacing is done by Xilinx CoreGen modules, and the decoding is a modification of the work done by Charles Irick [21]. The improvements include the ability to filter data based on MAC address and the addition of control packets. The filtering works by looking at the last two bytes of the MAC address and then sending the data out on a separate line for each MAC address. Currently only 00 and 01 are implemented but adding more would be as simple as declaring another wire in Verilog. Packets that are not destined for either of these MAC addresses are thrown away saving some processing time.


File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression...
Clear
Apply

```
No.   Time      Source           Destination      Protocol Info
 5399 0.895861  Apple_05:3d:12   Xilinx_00:00:00  0xbeef   Ethernet II
 5400 0.895865  Xilinx_00:00:00  Apple_05:3d:12   0xbeef   Ethernet II
 5401 0.896062  Apple_05:3d:12   Xilinx_00:00:00  0xbeef   Ethernet II
 5402 0.896078  Xilinx_00:00:00  Apple_05:3d:12   0xbeef   Ethernet II

▸ Frame 5400: 1504 bytes on wire (12032 bits), 1504 bytes captured (12032 bits)
▸ Ethernet II, Src: Xilinx_00:00:00 (00:0a:35:00:00:00), Dst: Apple_05:3d:12 (c8:2a:14:05:3d:12)
▸ Data (1490 bytes)

0000  c8 2a 14 05 3d 12 00 0a  35 00 00 00 be ef 61 00   .*..=... 5.....a.
0010  00 00 d1 00 00 00 00 06  00 ff ff ff 85 ee 26 ed   ........ ......&.
0020  86 11 27 5d 86 35 27 cc  86 59 28 3c 86 7e 28 ab   ..'].5'. .Y(<.~(.
0030  86 a3 29 1a 86 c8 29 89  86 ee 29 f7 87 14 2a 66   ..)...). ..)...*f
```

Figure 4.2: Data Packet Being Received by GNU Radio from the First Path on the First FPGA

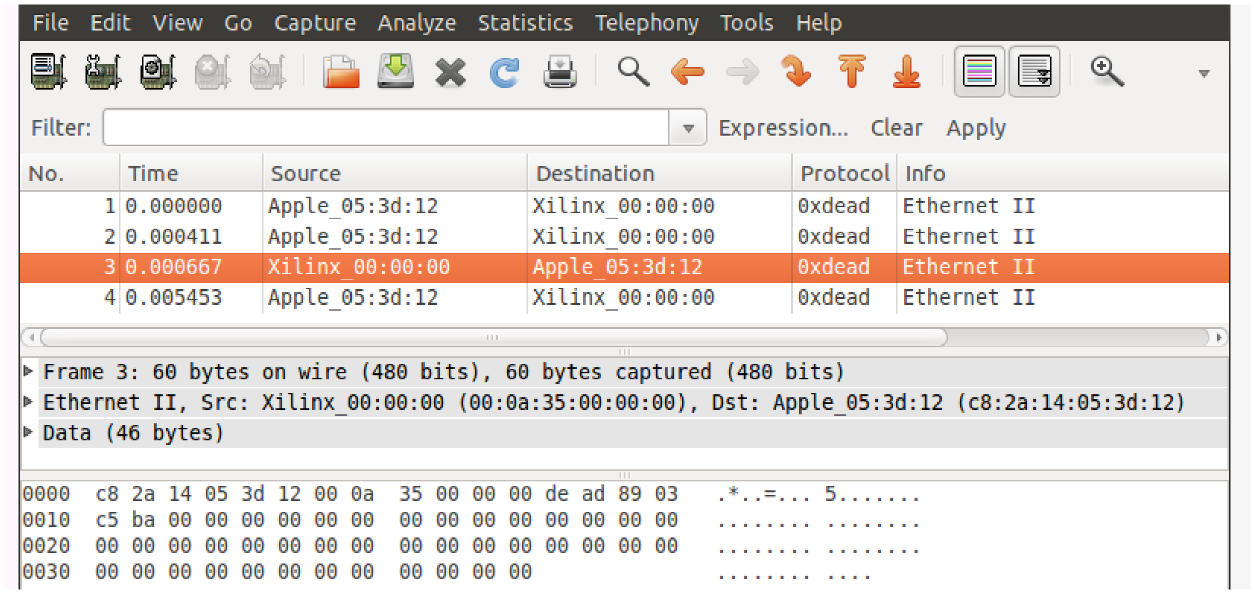
The data coming out of the dynamic region is formed in the same way. Data from the first path is sent from a MAC address ending in 00 and data from the second is sent from one ending in 01. This is expandable up to 256 paths on one FPGA.


File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression...
Clear
Apply

```
No.   Time      Source           Destination      Protocol Info
 5399 0.895861  Apple_05:3d:12   Xilinx_00:00:00  0xbeef   Ethernet II
 5400 0.895865  Xilinx_00:00:00  Apple_05:3d:12   0xbeef   Ethernet II
 5401 0.896062  Apple_05:3d:12   Xilinx_00:00:00  0xbeef   Ethernet II
 5402 0.896078  Xilinx_00:00:00  Apple_05:3d:12   0xbeef   Ethernet II

▸ Frame 5401: 1512 bytes on wire (12096 bits), 1512 bytes captured (12096 bits)
▸ Ethernet II, Src: Apple_05:3d:12 (c8:2a:14:05:3d:12), Dst: Xilinx_00:00:00 (00:0a:35:00:00:00)
▸ Data (1498 bytes)

0000  00 0a 35 00 00 00 c8 2a  14 05 3d 12 be ef 00 00   ..5....* ..=.....
0010  00 00 de 00 00 00 00 04  00 00 00 00 00 00 02 e5   ........ ........
0020  e0 97 83 eb e1 09 83 ce  e1 7b 83 b2 e1 ed 83 97   ........ .{......
0030  e2 5e 83 7b e2 d0 83 60  e3 42 83 46 e3 b5 83 2c   .^.{...` .B.F...,
```

Figure 4.3: Data Packet Being Sent by GNU Radio to the First Path on the First FPGA

The control packets are sent with a completely different packet type. Instead of using a different MAC address their type is changed from 0xBEEF to 0xDEAD. This type is arbitrary and 0xDEAD was only used because it was easily distinguishable in a set of packets from a network dump. The control packets are used most commonly to direct where the output data will go. They can also be used to save or present a checksum of the currently loaded EDIF with the host computer (Figure 4.4). The control packets can also be used to store values in known registers that can be accessed from the dynamic region. These known registers can contain parameters for blocks that just need a value assigned to them. This is not used with any blocks yet but could allow for simple changes on the hardware without requiring any new programming. The control packets are also set up to allow for reprogramming the FPGA using the flash memory. As mentioned before this is not implemented yet but the pieces are available on hardware to be used as soon as the software is added to GNU Radio.



No.	Time	Source	Destination	Protocol	Info
1	0.000000	Apple_05:3d:12	Xilinx_00:00:00	0xdead	Ethernet II
2	0.000411	Apple_05:3d:12	Xilinx_00:00:00	0xdead	Ethernet II
3	0.000667	Xilinx_00:00:00	Apple_05:3d:12	0xdead	Ethernet II
4	0.005453	Apple_05:3d:12	Xilinx_00:00:00	0xdead	Ethernet II

```
Frame 3: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
Ethernet II, Src: Xilinx_00:00:00 (00:0a:35:00:00:00), Dst: Apple_05:3d:12 (c8:2a:14:05:3d:12)
Data (46 bytes)

0000  c8 2a 14 05 3d 12 00 0a  35 00 00 00 de ad 89 03   .*..=... 5.......
0010  c5 ba 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0030  00 00 00 00 00 00 00 00  00 00 00 00              ........ ....
```

Figure 4.4: FPGA Responding to Checksum Request Using a Control Packet

The MGTs operate in a similar fashion to the Ethernet except that for now they only decode data and present it to the dynamic region in the standard data format. Because there can only be 256 paths in the dynamic region, some of them have to be dedicated to the Ethernet, and some to other interfaces. The current design assigns all Ethernet input data to the first path, and all MGT data to the second path. Both paths output to the Ethernet for now. More interfaces could be added as well as more paths, but they currently have to be hard coded in the static region. These interfaces cannot be changed and must be known by the user in order to piece together the flow graph properly.

4.2 Dynamic Region

Synthesis black boxes are used in ISE to provide the ability to separate modules from the static region and dynamic design. The top of the dynamic region is a black box that is named ‘blacktop’ because as a black box it appears to be the top of all of the modules being placed on the FPGA. All of the signal processing modules are also black boxed so that ‘blacktop’ only has to see the connections between them. This means that after synthesis there is an EDIF of the static top layer, an EDIF of all the modules, and an EDIF of just the connections in ‘blacktop’. This connection-only EDIF is created using TORC with the modifications made to the GNU Radio Core discussed in the previous chapter. ISE is able to use these EDIFs while qFlow/tFlow are being developed in order to continue testing the functionality of the blocks and their integration with GNU Radio.

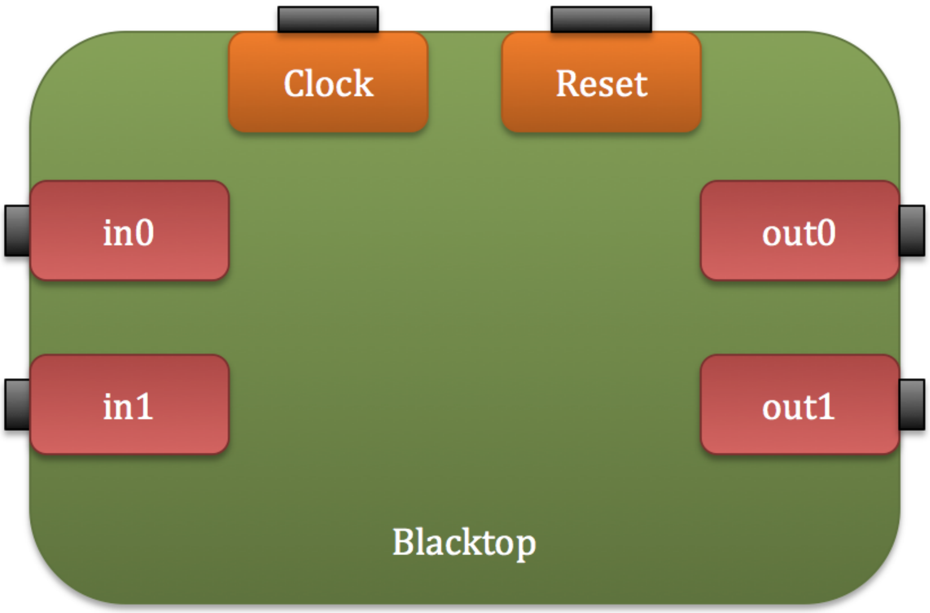


Figure 4.5: Blacktop Configuration

The current ‘blacktop’, as mentioned before, only has two paths so the interfaces with it are simple. There is a clock and reset port that gets tied to every module. Then there are in0, in1, out0, and out1. The *In* ports are for receive data and the *Out* ports are for transmit data. Their numbers denote the path they belong to and thus the MAC address they are assigned. The lengths of these ports are all 33 bits. For complex signals the first 16 bits contain I data, the second 16 bits contain Q data, and the last bit is used as a valid flag. The reason this data is presented as one bus instead of three separate buses is so that less things have to be wired up by the user and custom assembler later. In hardware it is often necessary to present every piece of data to the designer so that it can be manipulated properly. Radio designers are only interested in moving around signal data. Instead of forcing radio designers to connect two data wires and a valid line, they can simply connect one block to another and everything is handled for them.

The dynamic region is what 'edif.dat' from Chapter 3 represents. The static region is hard coded and only signal processing modules are placed inside of the dynamic region. The outer layer of the dynamic region is represented by the CELL called 'blacktop' with INSTANCE name BT0. Each path is represented by an in and out ARRAY. The clock is provided by a PLL in the static region, and the reset is tied through the static region to a button on the FPGA.

Listing 4.1: Blacktop Description Line from edifMAC.dat

```
Cell;blacktop;BT0;Array;input;in0;33;Array;input;in1;33;Array;output;
   out0;33;Array;output;out1;33;Port;input;rst;Port;input;clk;
```

4.3 Hardware Library

To make this flow more desirable, there is a library of hardware components in development currently. The community can also contribute to the hardware library development since they understand the blocks that are necessary to make their radios run. A set of standards are being used to ensure that all of the blocks can work properly together on the targeted hardware.

4.3.1 Standard Interfaces

The biggest problem in developing a heterogeneous system is the communication. This is why SCA was developed. GNU Radio uses a different method because it was not originally designed for use with hardware, other than the USRPs. The current method for sending data to and from USRP hardware is to transfer I/Q data at baseband over Ethernet. Baseband means that the signal appears to originate at 0 Hz. I and Q are real and imaginary parts that describe the instantaneous amplitude and phase of a wave in rectangular notation (see Figure 4.6). The I and Q parts are 16 bits each for a total of 32 bits per sample of the wave. This is a common format for signal processing so it is used in all of the FPGA hardware as well. The only difference is that a valid flag is added for flow control.

The 33 bit standard is not enforced by any of the code. It is recommended to keep designs simple and interchangeable. The current width of the data lines lends itself to holding integers, four characters at a time, or the standard complex data type. There is no type

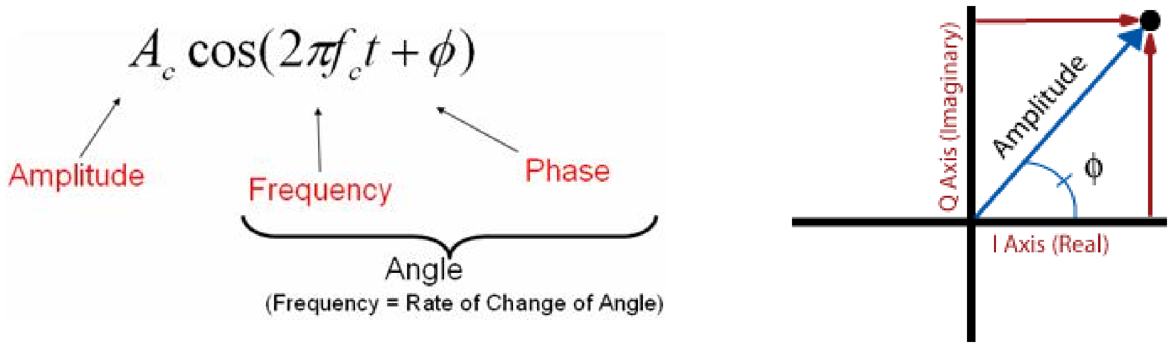
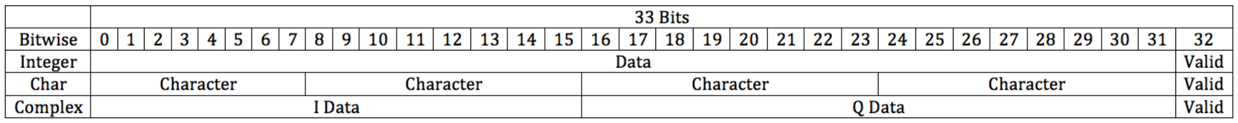


Figure 4.6: Mathematical Representation of a Sine Wave as I/Q Data [20]

checking implemented for hardware blocks yet; it is up to the radio designer to only connect one type of output to the same type of input. The hardware will assemble any connection as long as the ports are the same size. The blocks that are built make it clear what output is produced so, just as in hardware, modules must be wired together carefully. The ability to type check has been identified as important and is explained as Future Work in Chapter 7.



	33 Bits																																
Bitwise	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
Integer	Data																																Valid
Char	Character								Character								Character								Character								Valid
Complex	I Data																Q Data																Valid

Figure 4.7: Multiple Encoding for a 33 Bit Bus

If a custom block uses a different size port than the 33 bit standard, then it can only be connected to other blocks with a port of matching size. This can be useful if there are a small subset of blocks that should only ever be connected to each other, but in this case it is generally recommended to combine these into one block that can work with everything else. Having specific blocks can be great for one project but does not benefit the community or lend itself to reuse.

The argument can be made that different sized ports would allow for type checking but that severely limits the output of any design. Due to the unchanging nature of the static design, the output of blacktop must be 33 bits. This means that if 9 bits are used for ASCII coded characters, 8 for data plus a valid flag, then ASCII could never be piped out of blacktop without compressing it first. Instead of wasting the designers' time by forcing them to use a conversion block and wire up yet another connection that only clutters their flow graph, a standard 33 bit line is always used. Standardizing everything makes the prototyping of radios simpler and quicker.

4.3.2 Registered Ports

The custom assembler does not have the ability to check timing yet. To alleviate this problem, every input and output port is registered. This means that for every clock-cycle, data only has to go *through* one block OR travel *to* one block. In this way, if every block meets timing requirements on its own, the whole design will meet timing requirements. This does introduce an eight nanosecond delay for every block and every connection when the clock is running at 125 MHz, which is the standard clock to run Ethernet at full speed. For prototyping and for streaming data in one direction, this latency is perfectly acceptable.

4.4 High-Bandwidth Channelizer

To show the true flexibility of the GNU Radio enhancements a channelizer has been built and seamlessly added to GNU Radio as a configurable and high-bandwidth input. The channelizer consists of a 3.6 Gsps ADC connected over LVDS to a Virtex 6 FPGA that filters and splits the signal into down-sampled channels that can be processed or transmitted. In the current system the channels are hard coded to be sent over SATA connections to a stack of four Virtex 5 FPGAs, but these channels are easily selectable from the receiver. The data is decoded from the MGTs into the 33 bit format and then directed into 'blacktop'. The dynamic region sees no difference in the data coming from any source, except that the valid flag will be high more often because there is more bandwidth from the channelizer. This gives the FPGAs a way of getting 225 MHz of data instead of the current 25 MHz from a USRP2. FPGAs shine in high-bandwidth applications and are often limited by the connections to them rather than by their own processing power. The channelizer addition shows that it is possible to add high-bandwidth signal processing to GNU Radio instead of limiting designers to only adding FPGAs to the already available low-bandwidth applications. Even higher bandwidths can be achieved if a custom board with more connections is built for offloading the signal from the Virtex 6 FPGA.

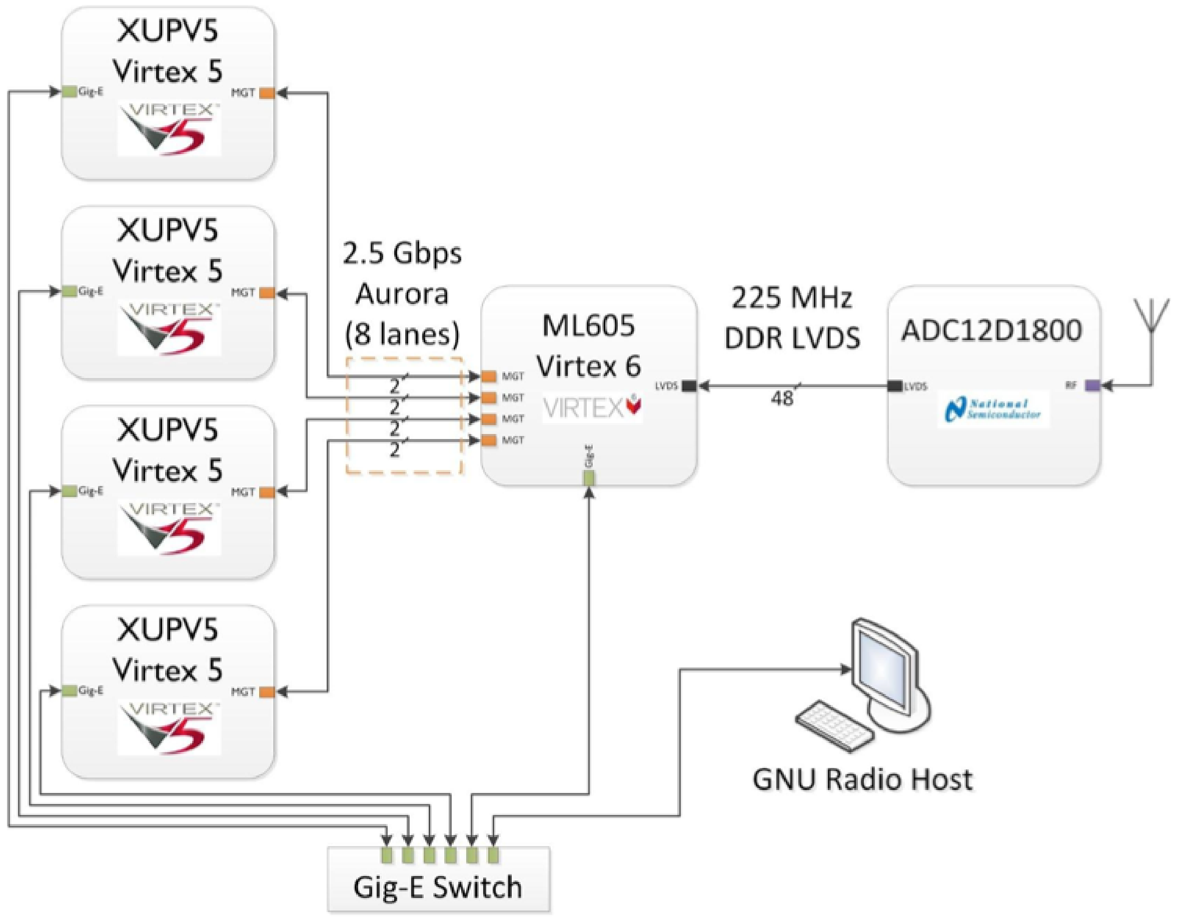


Figure 4.8: Channelizer Hardware Setup

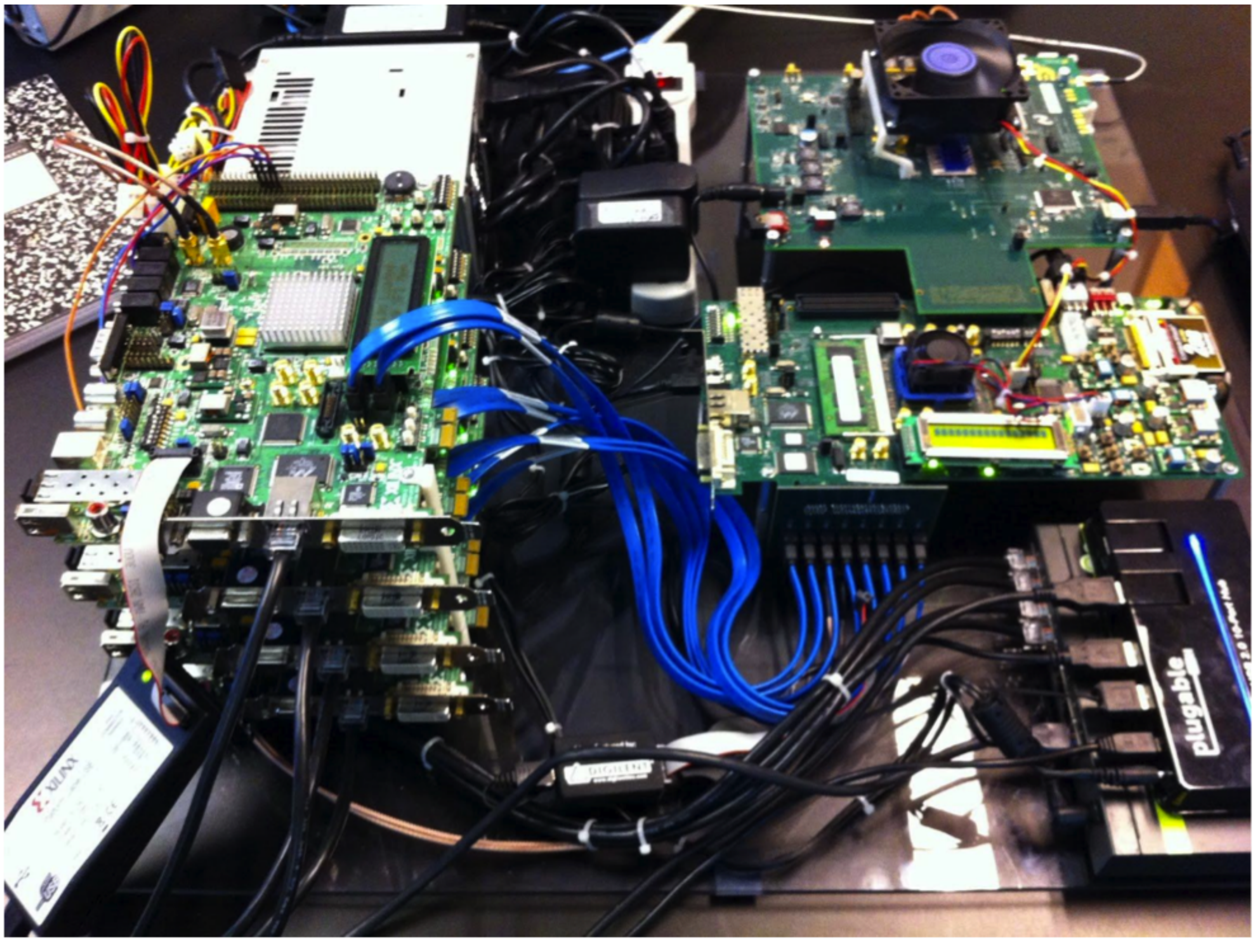


Figure 4.9: Complete Channelizer Platform

Chapter 5

qFlow/tFlow

This chapter discusses the usefulness and purpose of the custom assembler tools qFlow and tFlow. They are a completely separate project that can be researched here [16] [17]. The specific details of their operation are not discussed here, yet the details can be found in [15]. Instead, the how and why they are used is explained.

5.1 Operation

A custom bitstream assembly process is used to provide quick compile times for the hardware that is placed in a GNU Radio flow graph. To maintain rapid iterations of designs, traditional hardware compile times are untenable. GNU Radio is a system that has a quick turn around time, this means that the hardware needs to compile as quickly as the software so that prototyping is not hindered. The commercially available Xilinx tools offer the most optimized, and often only, way of compiling a hardware design for their FPGAs. The price for this optimization is a long build time, which on large systems can take over a day to complete [31]. By relaxing the placement optimization, qFlow is able to build a basic radio design in around two minutes. By removing the routing optimization, tFlow is able to build a basic radio in around twenty seconds.

5.1.1 Registering Static

To speed up the process even more, the static region (Figure 4.1) of the FPGA is prepared by the custom assemblers in advance. This means that all of the modules that are in every design are placed and routed before any flow graph is run. This saves time by isolating the hardest part of the design: the communication interfaces. The communication needs to meet strict timing requirements that cannot be easily met by qFlow or tFlow. Most importantly the static region isolates the infrastructure modules from *afpga* blocks. This makes the GNU Radio flow graph only require signal processing blocks.

5.1.2 Registering Modules

The modules are also registered in advance with the tools. They are registered after synthesis as EDIFs. This tells the tools exactly what has to be placed or routed in advance. By having a library of all of the processing blocks, the connection EDIF generated by GNU Radio does not need to contain any information except the connections and the names of the blocks. The tools can then pull the appropriate blocks from the library based on their names.

5.1.3 Off-Clock Processing

Registering the modules in advance gives the tools a chance to build them in different sizes and shapes. Instead of finding the optimal shape for a block at runtime, the tools just pick a block that will fit well with the other blocks. The placer effectively only has to play a quick game of Tetris instead of weaving together a perfect masterpiece. The block shaping takes place before runtime so, although it does take some time to process, this is never seen by the radio designer.

5.2 Integration

The qFlow tools fit nicely with the GNU Radio paradigm of having a library of blocks that can be added and removed easily from a flow graph. Both qFlow and tFlow already treat hardware the same way that GNU Radio treats software. Having them work together gives the designers the best of both worlds.

5.2.1 Modules

The modular based assembly of qFlow is what makes it so suited for GNU Radio. GNU Radio has the ability to drop any block into a flow graph and see it run instantly with everything that is already set up. It does not require the designer to re-plan their whole design to make it work, so the hardware should not either. By treating the blocks as modules that can be placed anywhere, rather than a whole design that needs to be implemented, the tools work quickly and effectively together.

Table 5.1: Resource Usage Comparison for BPSK on FPGA

	Slice Registers	Slice LUTs	Block RAM/FIFO	DSP48Es
ISE	3927	4357	11	11
qFlow	4026	4538	11	11
tFlow	4178	4931	11	11

5.2.2 Speed Over Optimization

GNU Radio designers are used to running designs in a matter of seconds. Although it has become acceptable in the hardware realm to wait on designs, this slows productivity and can be overcome one way by trading some optimization. During the prototyping phase, a designer is not interested in how perfect the design is but rather in how quickly the design iterates until a working one is found. By placing modules without optimizing their resource utilization, the build times are significantly reduced. This means that multiple blocks on the FPGA take up more resources than they would normally. For example if Module 1 uses resources A and B, while Module 2 uses resources B and C, the whole design would take resources A, B, B, and C. An optimized design would share Resource B between the two blocks instead of replicating it. None of the blocks use more resources than they have to, but as a whole system there are wasted overlaps. The resource usage for the BPSK implementation discussed in Chapter 6 can be seen in Table 5.1. The overhead for the custom tools on this small design is less than one percent of the available resources.

5.3 Trade-Offs

Other than losing resource optimization on the FPGA there are a few trade-offs that designers have to deal with to get software like compile times with hardware.

5.3.1 Hard Drive Space

Registering multiple shapes for every block in advance quickly takes up a large amount of space. Any library is expected to take some amount of space and today Gigabytes are practically free for computers; however, this is a trade off that the designer has to be aware of. The hardware libraries will consume a large amount of space and could take a long time to download for a first run. This is a price that is easily traded away for near instant prototyping with hardware. It should also be noted that the size is directly proportional to the number of blocks saved. There is no combining of blocks to pre-build a bitstream like partial reconfiguration does, there is only shaping on a per block basis.

5.3.2 Timing/Latency

As mentioned before, the qFlow tools are not able to correct timing issues although they do alert the user if timing is not met. This means that all of the blocks have to meet timing on their own and no optimization can happen here either. In order to protect the entire design, each block has registers on all of its inputs and outputs. This ensures that the clock does not cycle before data is ready at any register but quickly adds a large amount of latency to any complex design.

5.3.3 qFlow Versus tFlow

Although qFlow and tFlow both yield a working bitstream as an output, they operate under slightly different principles. qFlow implements a custom placer that quickly decides where to place pre-synthesized modules on any FPGA. It then makes use of Xilinx's router to wire everything up. This routing is still optimized and thus takes around two minutes to complete with the shortest paths.

tFlow has a little bit more information about the organization of the Virtex 5 family, so it is able to do bit manipulation to place already routed versions of the modules on the FPGA. It then does a small amount of routing at the bit level to connect the modules that were placed. This process is currently the fastest of the custom tools for building a working bitstream. It requires that all of the pieces be compiled to the bitstream level in advance but all of the blocks in the current library have been registered with tFlow to make this possible.

The only downside to tFlow is that it requires an intimate knowledge of the FPGA family that is being worked on. This knowledge currently only exists for the Virtex 5 boards, so any work on another family will have to use qFlow. There is still a large time saving from Xilinx tools to qFlow, but as more work is done on tFlow it should find its way onto more FPGAs. None of these tools work with Altera boards yet, but should Altera open up their flow it could be possible to add their boards to the library as well.

Chapter 6

Implementations & Results

This chapter discusses the methods for using the enhancements to GNU Radio. It outlines two cases that were developed to use GReasy and discusses the settings used for correct performance. The results are described and compared to the vendor tools for assessment.

6.1 Library

The enhancements to GNU Radio allow for FPGA blocks to be dragged and dropped into flow graphs and still keep the ability to perform 'what-if' experiments. The rapid iteration of designs with hardware makes the prototyping of more complex radios feasible. But in order to build the hardware a library has to exist.

6.1.1 Available Blocks

Using a library of pre-built and pre-registered hardware blocks, a radio can be stitched together for an FPGA just like software designers are used to doing. When this library has grown into the size that GNU Radio software blocks are at now, then any designer can pick up the system and prototype applicable radios. The currently available blocks are limited to the ones built for demonstrating proofs of concept. More are being worked on at Virginia Tech to increase the basic available block library.

6.1.2 Adding Custom Blocks

As with any open source project, the community builds what is needed and shares it with everyone. The process of adding custom hardware blocks to the library has been described in a separate document [37]. The designer is only required to write one hardware block in order to see it working on an FPGA. The static region already on the FPGA handles communication between hardware and GNU Radio. The static region also manages flow control between new blocks if they follow the outlined standard. Once a block is coded, there are scripts that will automatically build and register it with qFlow/tFlow. Then the block will be automatically built for GNU Radio and added to the GRC library as well. All of these steps are documented in detail so that any designer can build a hardware block for GNU Radio.

6.2 Proofs Of Concept

The signal processing blocks used to test this project were either already available or developed by The Mobile and Portable Radio Research Group (MPRG) at Virginia Tech. The purpose of this section is to discuss the usability of GReasy not the development of the individual blocks. The proofs used in this section were chosen because the radio functionality already existed or was simple to create on an FPGA. Each one shows how the hardware and software can be set up to achieve a working radio.

Both demonstrations run in a virtual machine on a MacBook Pro with a 2.3 GHz Core i7 processor, 8 MB shared level 3 cache, 8 GB of 1333 MHz DDR3 SDRAM, and a 251 GB Apple SSD. All of the components are in a repository that is working on multiple computers. The software is not limited to one specialized computer, any computer that can run GNU Radio can run GReasy. The laptop is used because it offers portability and easy access to the attached hardware.

6.2.1 ZigBee

A ZigBee demonstration was done for the original work [21] on this project because it is a complex standard that can benefit from hardware acceleration. The extent to which it was done before was limited to building it in Xilinx's ISE tool and placing it on the FPGA manually. A ZigBee signal from an XBee transmitter was received by a USRP2 and redirected to an FPGA by GNU Radio. The FPGA sent the decoded ASCII data back to the host computer for interpretation. The hardware setup is shown in Figure 6.1 with the only physical addition being multiple networked FPGAs to handle processing.

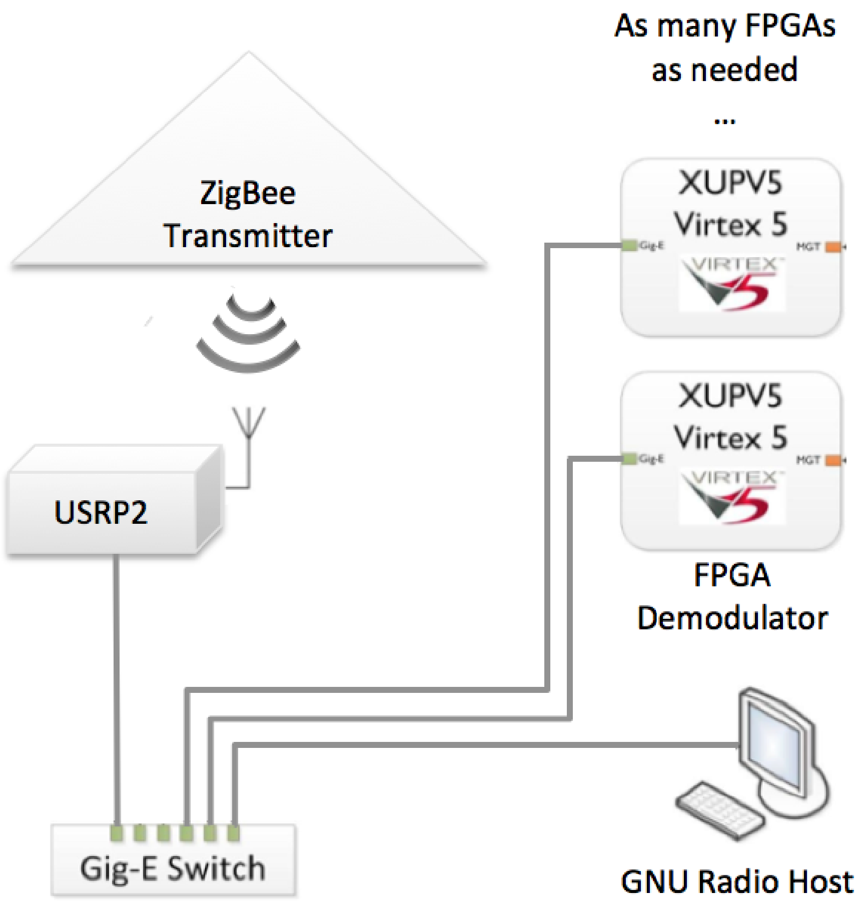
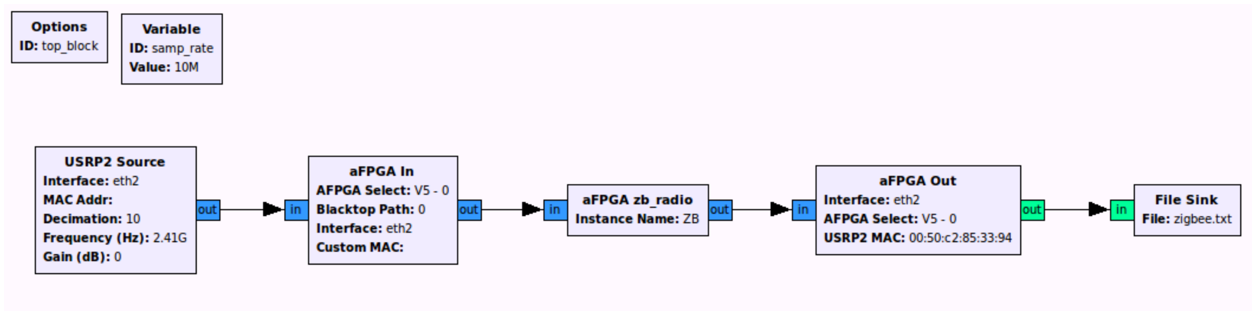


Figure 6.1: Improved ZigBee Hardware Setup

The addition of individual hardware blocks to GReasy lets the designer view what is being placed on the FPGA and control how it is connected. In Figure 6.2 the USRP2 now connects to the *afpga_in* block, which connects to the ZigBee demodulator, which connects to the *afpga_out* block, which pipes the decoded ASCII to a file. The USRP2 is set to run at a center frequency of 2.41 GHz with a decimation of 10 and unity gain. The USRP2's constructor sends these settings over Ethernet and the host computer is set up as the receiver for data. GReasy needs a USRP2 MAC address in order to redirect network traffic from the USRP2 to an FPGA instead.


Options
ID: top_block

Variable
ID: samp_rate
Value: 10M

USRP2 Source
Interface: eth2
MAC Addr:
Decimation: 10
Frequency (Hz): 2.41G
Gain (dB): 0

aFPGA In
AFPGA Select: V5 - 0
Blacktop Path: 0
Interface: eth2
Custom MAC:

aFPGA zb_radio
Instance Name: ZB

aFPGA Out
Interface: eth2
AFPGA Select: V5 - 0
USRP2 MAC: 00:50:c2:85:33:94

File Sink
File: zigbee.txt

Figure 6.2: ZigBee Flow Graph in GRC

The *afpga_in* block is set to run FPGA-0 on eth2, this means that it tries to program a board that has a base MAC address of 00:0A:35:00:**00**:00 using the second Ethernet interface on the computer. The 'Blacktop Path' is set to 0 so that qFlow/tFlow can connect blocks to the first path, which outputs on the Ethernet. The only option for the ZigBee demodulator (*afpga_zb_radio*) is to set the instance name. This name is simply a way of identifying which blocks are which in the final EDIF created by GNU Radio; it is not necessary to set these names for proper operation. The *afpga_out* block is also running on FPGA-0 across the second Ethernet interface. Its output is green in GRC because it is decoding the data as an integer containing four characters. In GRC the software types have to match, so the file sink is green as well for the demo. A file sink can also be used to capture complex data (blue), but this assumes every 16 bits of I/Q data is a scaled value between -1 and 1. GNU Radio handles the conversion of the data to the proper type before exposing it to the software blocks.

6.2.2 BPSK

Due to the limited bandwidth of the USRP2 a channelizer was used to enable processing multiple signals at once with one GNU Radio flow graph. This required building the custom receiver discussed in Chapter 4. The channelizer highlights GNU Radio's new ability to interface with multiple pieces of processing hardware. With their own development platform, radio designers are no longer limited by a USRP in order to receive real signals.

BPSK was chosen because it is simple to implement but requires multiple blocks that could test the routing capabilities of the custom assembly tools. The different blocks are used to show how GNU Radio can connect more than one module inside of the dynamic region on the FPGA. All of the following examples start with a Virtex 6 source. The Virtex 6 is not yet supported with these tools so this block is just a null source functioning as a placeholder for a complete flow graph to exist. The connection to the Virtex 5 is hardwired so the Virtex 6 is constantly streaming data to the second path on the dynamic region. [The *afpga_in* block in every example represents this change as 'Blacktop Path' is now set to 1, instead of 0 for Ethernet traffic.] It is worth noting that eth2 is set because control packets are still being sent to the FPGA over Ethernet. For all of the following flow graphs different FPGAs will be chosen and can be seen in the FPGA select parameter of the *afpga_in* and *afpga_out* blocks. It does not matter which FPGA the processing occurs in, as long as *afpga_in* and *afpga_out* match. Data cannot be sent onto one FPGA and then received from a completely separate one without first connecting those two FPGAs.

In Figure 6.3 a simple decimation is performed to reduce the bandwidth of the wide channel before piping it back over Ethernet for the host to do an FFT shown in Figure 6.4. The BPSK waveform can be seen in this way to make sure that the transmitter is working. The current transmitter is set up using another GNU Radio flow graph (Figure 6.5) on a separate computer that is running software-only blocks and transmitting two data channels with two USRP2s that are being wired directly into the ADC to lower noise.

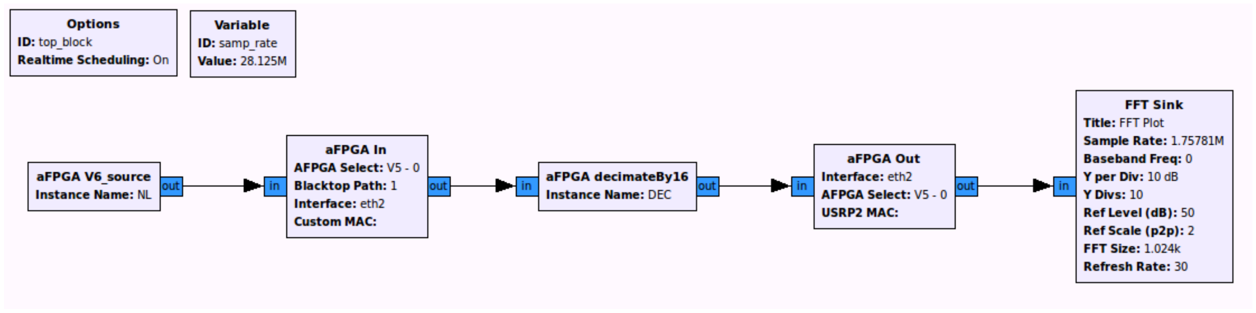


Figure 6.3: GRC Flow Graph of Signal Decimation

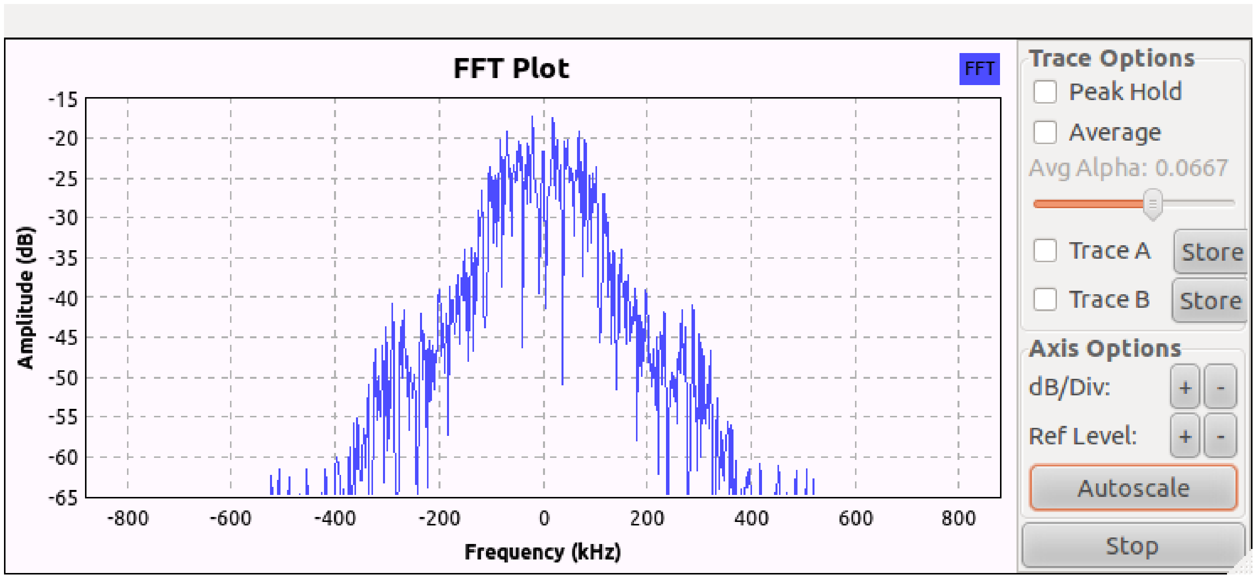
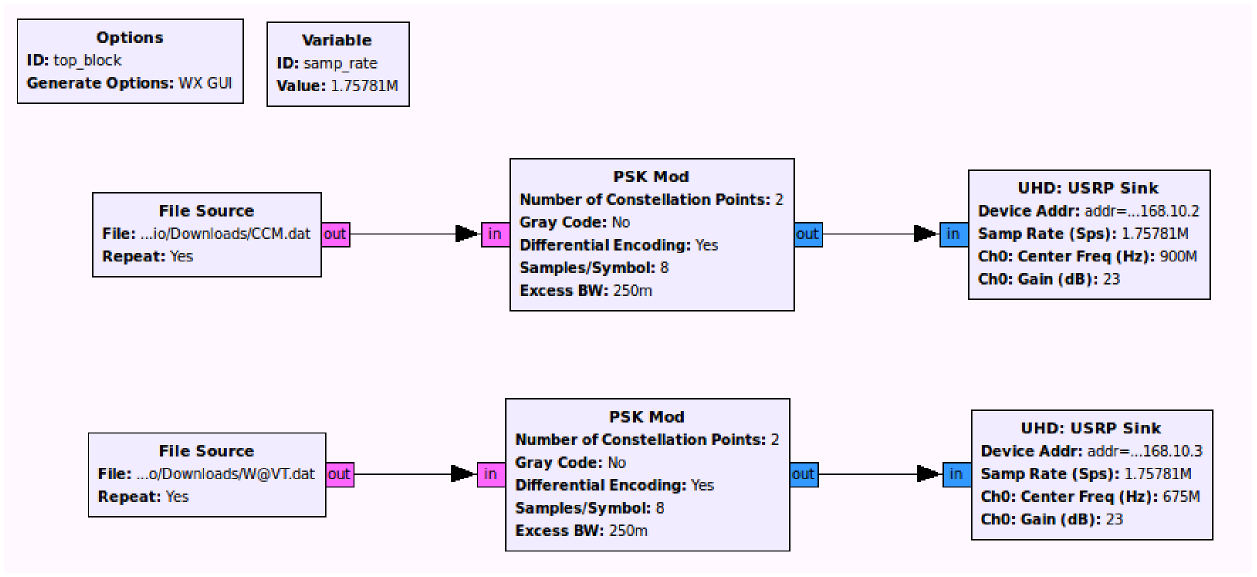


Figure 6.4: FFT of Signal Decimation



Options
ID: top_block
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 1.75781M

File Source
File: ...io/Downloads/CCM.dat
Repeat: Yes

PSK Mod
Number of Constellation Points: 2
Gray Code: No
Differential Encoding: Yes
Samples/Symbol: 8
Excess BW: 250m

UHD: USRP Sink
Device Addr: addr=...168.10.2
Samp Rate (Sps): 1.75781M
Ch0: Center Freq (Hz): 900M
Ch0: Gain (dB): 23

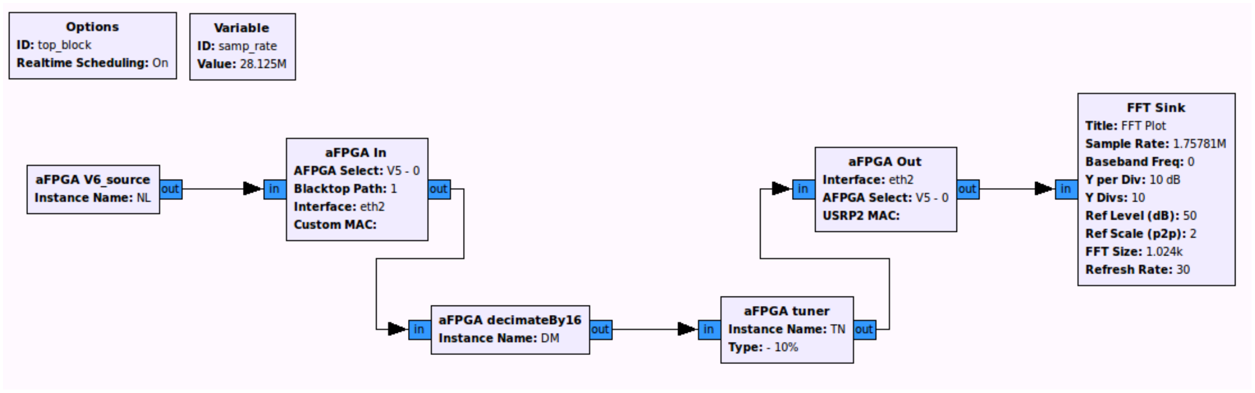
File Source
File: ...o/Downloads/W@VT.dat
Repeat: Yes

PSK Mod
Number of Constellation Points: 2
Gray Code: No
Differential Encoding: Yes
Samples/Symbol: 8
Excess BW: 250m

UHD: USRP Sink
Device Addr: addr=...168.10.3
Samp Rate (Sps): 1.75781M
Ch0: Center Freq (Hz): 675M
Ch0: Gain (dB): 23

Figure 6.5: GRC Flow Graph of BPSK Transmission

Figure 6.6 shows a tuner block that can be used to move the center frequency if the signal is not already aligned. The signal is already aligned but this block is worth showing because it is the first parameterizable block. The amount and direction of tuning can be changed from a drop down menu within the graphical block. The current settings are a negative 10% shift. All this does is call a different hardware block to put on the FPGA, but this has been abstracted to the graphical user as only one block. The shift can be seen in Figure 6.7.


Options
ID: top_block
Realtime Scheduling: On

Variable
ID: samp_rate
Value: 28.125M

aFPGA V6_source
Instance Name: NL

aFPGA In
AFPGA Select: V5 - 0
Blacktop Path: 1
Interface: eth2
Custom MAC:

aFPGA decimateBy16
Instance Name: DM

aFPGA tuner
Instance Name: TN
Type: - 10%

aFPGA Out
Interface: eth2
AFPGA Select: V5 - 0
USRP2 MAC:

FFT Sink
Title: FFT Plot
Sample Rate: 1.75781M
Baseband Freq: 0
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): 50
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 30

Figure 6.6: GRC Flow Graph of Signal Tuning

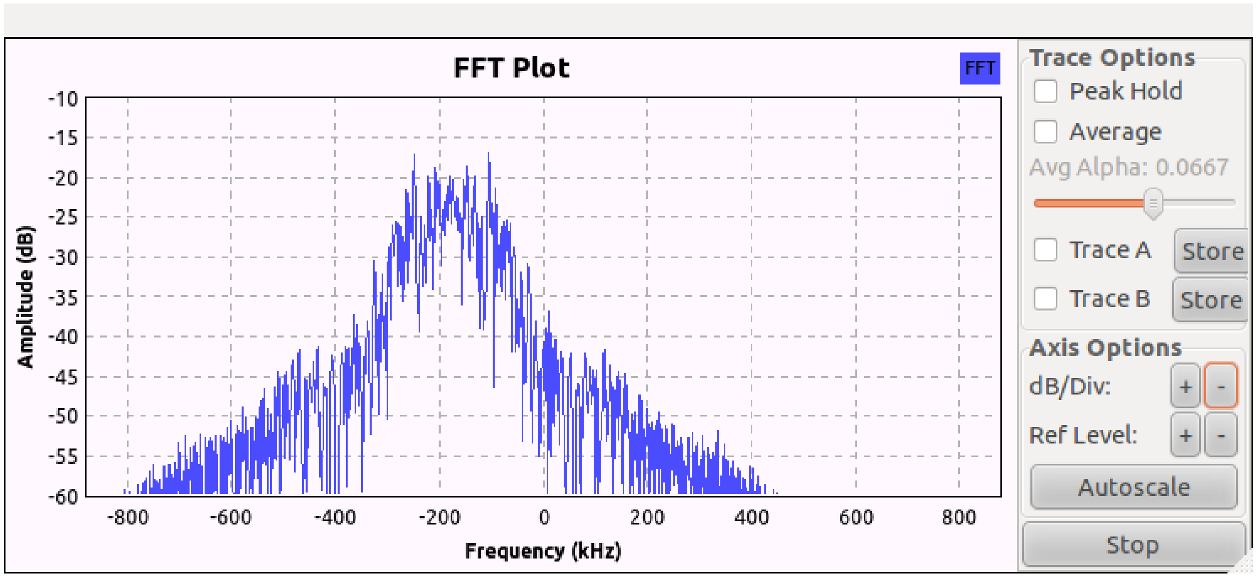


Figure 6.7: FFT of Signal Tuning

The next flow graph in Figure 6.8 is made up of a decimate block that brings the signal to a manageable sample rate, the actual BPSK demodulation that converts the signal into its binary encoding, and finally a data recovery block that identifies a specific preamble before selecting 8 bit characters to send as ASCII data. The types of all these wires appears to be complex to GNU Radio so that any FPGA block can be connected together, and although all of these connections are 33 bits wide, they are each carrying completely different data types. Any of these types can be sent over Ethernet to another appropriate block, but type checking is not currently available so it is up to the user to connect modules properly.

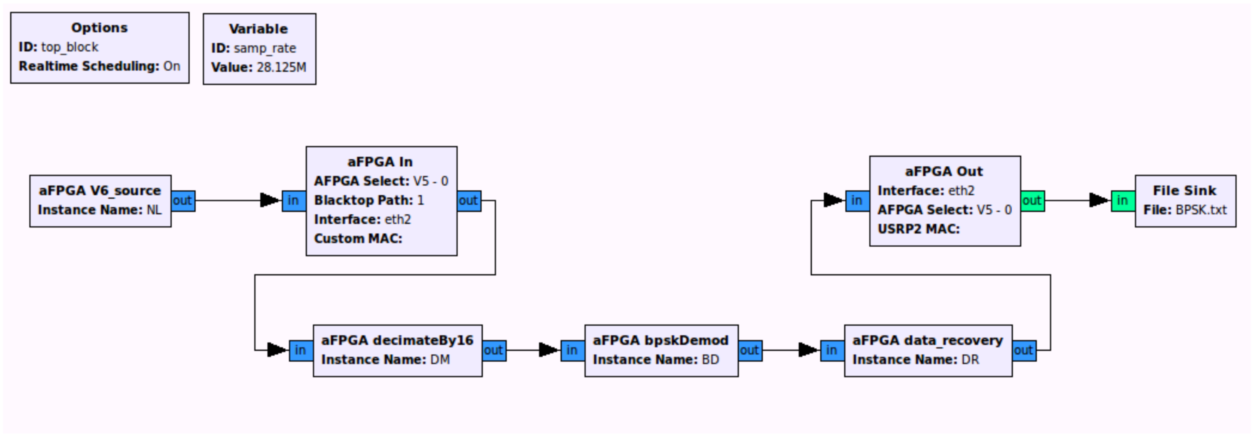
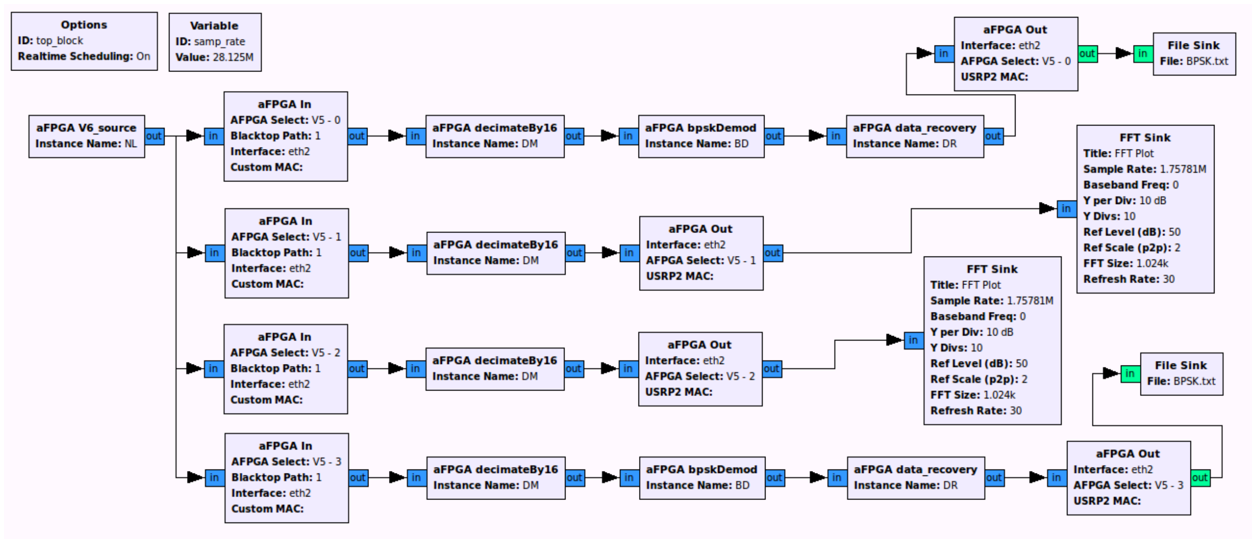


Figure 6.8: GRC Flow Graph of BPSK Demodulation

There are four FPGAs in the current stack so the Virtex 6 can offload up to four different channels. For Figure 6.9 there are only two channels being sent because there are only two USRP2s transmitting. A channel at 900 MHz is sent to the first two boards, and a channel at 1150 MHz is sent to the second two boards. This allows a designer to see an FFT of the data being received and decode the same data on another FPGA in parallel. This is being done for two 250 MHz channels at the same time, and can be easily increased to do more. Given multiple paths on the FPGA and enough network bandwidth, all of the processing shown can be done on one board. The use of multiple boards shows their ability to work together.

The last flow graph, Figure 6.10 shows a signal starting on the Virtex 6, being decimated on one FPGA, being tuned on another FPGA, and finally being tuned again and sent back to the host from a final Virtex 5. A shift of 10% on two boards can be seen as a total of 20% in the FFT of Figure 6.11.



Options ID: top_block Realtime Scheduling: On

Variable ID: samp_rate Value: 28.125M

aFPGA V6_source Instance Name: NL

aFPGA In AFPGA Select: V5 - 0 Blacktop Path: 1 Interface: eth2 Custom MAC:

aFPGA decimateBy16 Instance Name: DM

aFPGA bpskDemod Instance Name: BD

aFPGA data_recovery Instance Name: DR

aFPGA Out Interface: eth2 AFPGA Select: V5 - 0 USRP2 MAC:

File Sink File: BPSK.txt

aFPGA In AFPGA Select: V5 - 1 Blacktop Path: 1 Interface: eth2 Custom MAC:

aFPGA decimateBy16 Instance Name: DM

aFPGA Out Interface: eth2 AFPGA Select: V5 - 1 USRP2 MAC:

FFT Sink Title: FFT Plot Sample Rate: 1.75781M Baseband Freq: 0 Y per Div: 10 dB Y Divs: 10 Ref Level (dB): 50 Ref Scale (p2p): 2 FFT Size: 1.024k Refresh Rate: 30

aFPGA In AFPGA Select: V5 - 2 Blacktop Path: 1 Interface: eth2 Custom MAC:

aFPGA decimateBy16 Instance Name: DM

aFPGA Out Interface: eth2 AFPGA Select: V5 - 2 USRP2 MAC:

FFT Sink Title: FFT Plot Sample Rate: 1.75781M Baseband Freq: 0 Y per Div: 10 dB Y Divs: 10 Ref Level (dB): 50 Ref Scale (p2p): 2 FFT Size: 1.024k Refresh Rate: 30

aFPGA In AFPGA Select: V5 - 3 Blacktop Path: 1 Interface: eth2 Custom MAC:

aFPGA decimateBy16 Instance Name: DM

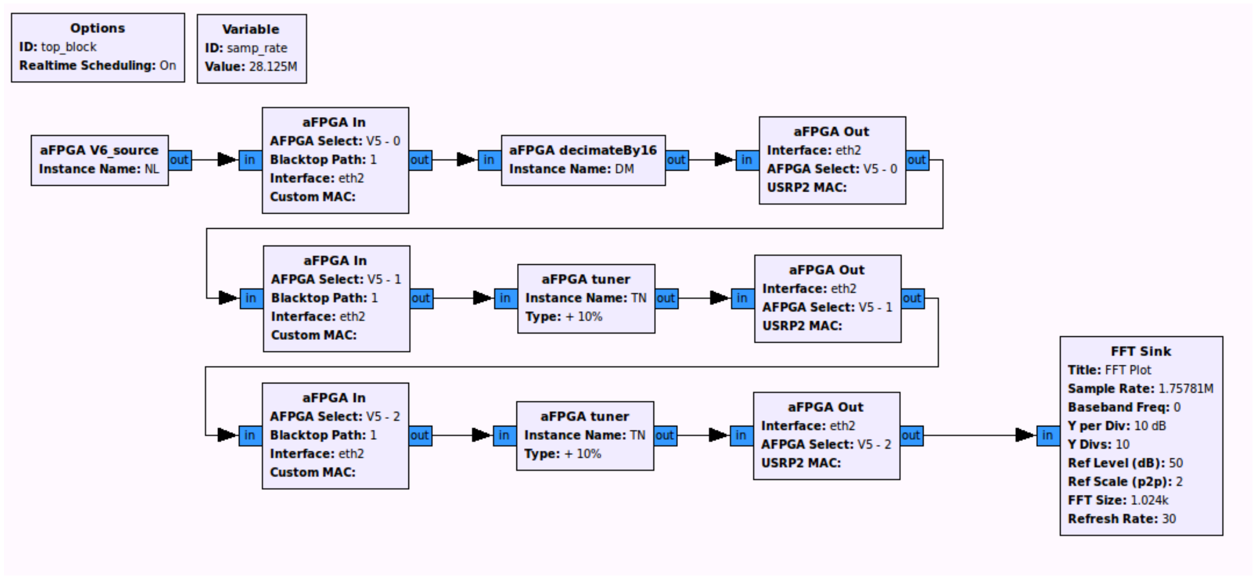
aFPGA bpskDemod Instance Name: BD

aFPGA data_recovery Instance Name: DR

aFPGA Out Interface: eth2 AFPGA Select: V5 - 3 USRP2 MAC:

File Sink File: BPSK.txt

Figure 6.9: GRC Flow Graph of Decimation/Demodulation on Four Separate FPGAs



Options ID: top_block Realtime Scheduling: On

Variable ID: samp_rate Value: 28.125M

aFPGA V6_source Instance Name: NL

aFPGA In AFPGA Select: V5 - 0 Blacktop Path: 1 Interface: eth2 Custom MAC:

aFPGA decimateBy16 Instance Name: DM

aFPGA Out Interface: eth2 AFPGA Select: V5 - 0 USRP2 MAC:

aFPGA In AFPGA Select: V5 - 1 Blacktop Path: 1 Interface: eth2 Custom MAC:

aFPGA tuner Instance Name: TN Type: + 10%

aFPGA Out Interface: eth2 AFPGA Select: V5 - 1 USRP2 MAC:

aFPGA In AFPGA Select: V5 - 2 Blacktop Path: 1 Interface: eth2 Custom MAC:

aFPGA tuner Instance Name: TN Type: + 10%

aFPGA Out Interface: eth2 AFPGA Select: V5 - 2 USRP2 MAC:

FFT Sink Title: FFT Plot Sample Rate: 1.75781M Baseband Freq: 0 Y per Div: 10 dB Y Divs: 10 Ref Level (dB): 50 Ref Scale (p2p): 2 FFT Size: 1.024k Refresh Rate: 30

Figure 6.10: GRC Flow Graph of FPGAs in Series

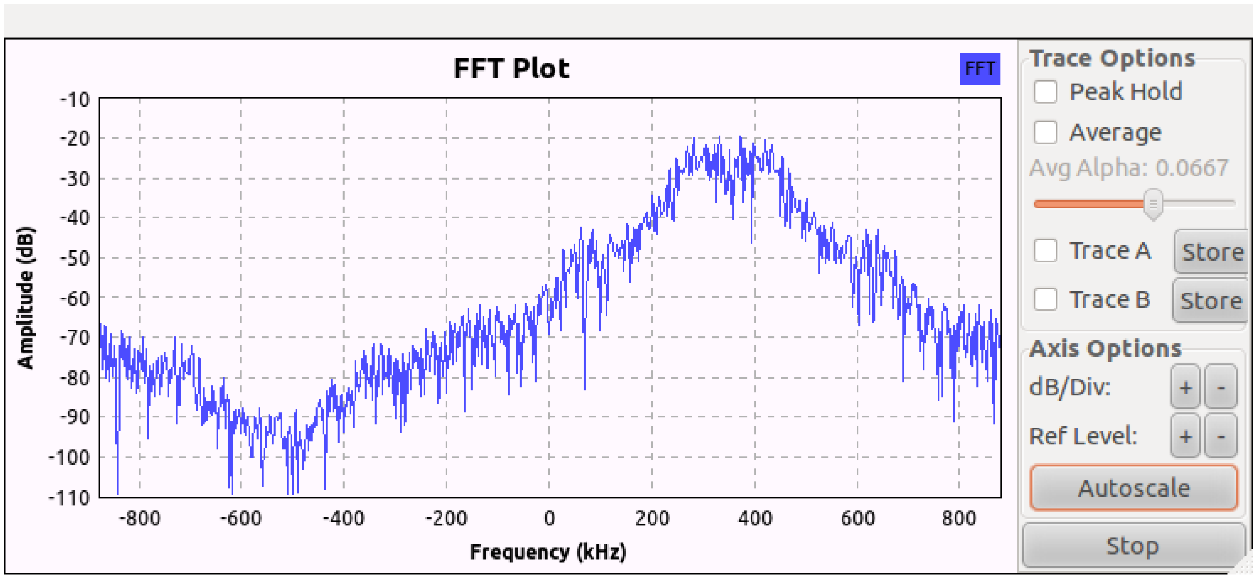


Figure 6.11: FFT of Two Positive Shifts

6.3 Results

Except for changing the different receivers, these implementations can be switched between one another with ease on any computer. Although both of these setups do not show off the true processing power of an FPGA, they do display the new ease-of-use for programming a hardware block that runs in sync with software.

6.3.1 Design Process

The design process now consists of placing a desired processing block for hardware in a GNU Radio script and having it just work at software like speeds. The process is enhanced even further with the use of GRC, which gives a designer a list of blocks that can be dragged and dropped until a desired flow graph is built. This visual graph shows software and hardware blocks working together, all represented in one space as a complete heterogeneous radio.

6.3.2 Abstraction

The hardware is compartmentalized into block-level processing so that the hardware design process is easier. More generalized blocks can now be created without the need to understand all of the hardware. This abstracts hardware to a usable level for SDR designers. Developers that want to design hardware do not have to worry about building a communication interface or recreating blocks that are already available to the community. The designers that want to focus on building radios and not getting into hardware can now select an FPGA block from the same library of software blocks they are used to.

6.3.3 Compile Times

The compile times for FPGA hardware in a prototyped radio have satisfied the instant gratification requirements of GNU Radio. The addition of hardware does not slow down the design process and only enhances the number of radios which can be successfully implemented. Synthesis takes the same amount of time for a new hardware block, but this happens well before the radio designer is ready to start prototyping. The off-clock processing is a huge benefit to the design time and is something that current Xilinx tools cannot offer. If the design requires a re-implementation of any logical components, instead of just structural re-organization, then synthesis must be run for the custom tools. This is not common for prototyping, but is required when changes are made to the hardware description or if a new block is introduced.

Tables 6.1 and 6.2 are comparisons of the time it takes to assemble the ZigBee and BPSK radios using the Xilinx tool ISE against the custom tools qFlow and tFlow. ISE still runs Synthesis and Map even when no changes are made to the design logic. Though BPSK has less logic than ZigBee, the custom tools take longer to run BPSK because it has more connections. The custom tools are only concerned with placing and routing as long as there is still room on the FPGA. ISE implements the entire design, so the more resources used the longer assembly takes. Other tools are not discussed here because their speed does not matter. The custom tools used have performed at more than reasonable speeds and this comparison is meant to show where the accelerations occur in relation to the tools designed for the FPGA.

Table 6.1: Average ZigBee Assembly Times in Seconds

	Synthesis	Map/Place	Route	Bit Generate	Total
ISE	198	89	46	33	366
qFlow	0	35	46	39	120
tFlow	0	10	5	0	15

Table 6.2: Average BPSK Assembly Times in Seconds

	Synthesis	Map/Place	Route	Bit Generate	Total
ISE	94	109	47	34	284
qFlow	0	43	49	42	134
tFlow	0	11	6	0	17

Chapter 7

Conclusion

This chapter reiterates the improvements made to the design process of GNU Radio when using hardware. It also describes areas of future work that will be useful to the community and this system.

7.1 Conclusion

Enhancements to GNU Radio were presented that allowed for the use of FPGAs in prototyping SDRs without introducing normal hardware compile times. The standard FPGA compile times were orders-of-magnitude larger than what GNU Radio users expected. The nature of the prototypes allowed for the use of modular assembly tools qFlow and tFlow to achieve the significant increases in build speeds over vendor tools. The FPGA modules were organized in a library of hardware signal processing blocks as the new *afpga* class.

Software was written to place the GNU Radio hardware blocks directly on an FPGA by building the connections in EDIF format and passing them to qFlow/tFlow. A static region was built for the FPGA that handles communication so that signal processing designers do not have to. The synthesis process was scripted so that new blocks could be added to GNU Radio easily. The graphical front-end GRC was used to present the hardware library in a compact and organized way.

From the perspective of a radio designer, library-based assembly is more natural than low-level hardware description languages and hides the complexity of FPGAs. In the flow presented here, chains of computation were specified for FPGA implementation within the GNU Radio framework just as if they were original radio blocks in the flow. Once standard radios and filters have been written in hardware by one designer they can be passed around in the form of blocks, which anyone can drop into their design without the tedious step of developing for FPGAs.

The enhanced GNU Radio flow was demonstrated using USRP2s, a 3.6 Gsps ADC connected to a Virtex 6, and a stack of four Virtex 5 FPGAs, all networked to a host computer with gigabit Ethernet. One clear benefit of this flow was that the FPGAs could be added or taken away just like any other module, and were not a forced part of the design. Any number of FPGAs could be added, and all of the communication and interconnects would be handled implicitly. The radio designer could pick the composition of a radio using parameterized components, and chose where they go in the flow.

These enhancements to GNU Radio showed how an FPGA system can be built in near real-time for an SDR environment. GNU Radio was used as a test bench for qFlow and tFlow because of its open nature and current lack of FPGA integration. These tools could easily make their way into other faster systems as designers balance software like compile times for hardware over complete optimization.

7.2 Future Work

There are still many limitations that will be addressed in future versions of GReasy. Some for the assembly tools, and some for the GNU Radio presentation and management of hardware.

7.2.1 Improvements

The first improvement could be to come up with a method for type checking the signals on the FPGA. The standard GNU Radio types could be used to describe each block's input and output and then GNU Radio would handle the type checking automatically. This would require a change in the way modules are automatically registered to include a type option, as currently everything is registered with the complex type.

Another improvement could be to add more paths to the dynamic region. GReasy is already designed to understand up to 256 paths but the static region would have to increase in size and complexity to actually implement them. Also there is currently not a good method for denoting which paths are designated for Ethernet and which are for MGTs. If another interface were added, it would also just get assigned to the next available path. Unless the designer knows exactly how many paths exist and which ones are tied to which interfaces, they cannot currently set up an effective radio. Perhaps a range could be established where all paths between 0-15 are dedicated to Ethernet, whether they are used or not, and all paths between 16-31 are dedicated to MGTs. This way, as more paths are added, the range will still always use the same interface but with a fixed cap on the number of paths per interface. This should be acceptable, as the number of paths will eventually be limited by the space on the FPGA long before it takes up the address space.

A second improvement to the hardware could be the addition of a control bus within the dynamic region. Currently, only data moves through the signal processing blocks that are added; but with control data these could be become run-time parameterized. Also with a separate bus running through the dynamic region, debug information could be output to another channel for reading while the radio is in operation. A bus running through the dynamic region and requiring control logic on each module would quickly take up resources on the FPGA; however, research is being done to use the FPGA's unused configuration circuitry to build a network on chip that would work for this project without requiring more resources [35].

Many improvements are still being made to the hardware assembly tools as well. qFlow is only designed to assemble for the Virtex 5 family of FPGAs, but will have Virtex 6 functionality soon. tFlow is being actively developed to increase the speed and effectiveness of the TORC router. With these improvements the GReasy project could produce on-the-fly hardware building so that new modules do not have to be registered in advance. The bitstream manipulation improvements could yield parameterization of modules without any assembly time and possibly without programming the entire chip to reflect a change in only one module. All of these improvements would decrease the time designers waste waiting for hardware.

In order for this project to be fully integrated with GNU Radio, the changes have to be checked into the GNU Radio repository. This would require updating many of the current files to work with the newest version of GNU Radio first. The version used for GReasy was 3.3 and as of this writing version 3.6 is available. Modifying the core files and adding the *afpga* class to the newest version would be simple. The problem would be upgrading the networking on the FPGA. GNU Radio has abandoned the use of RAW packets for data transfer to the USRPs. This also means that the USRP2 image would have to be modified and recompiled with a new command to send data to the FPGA. This would make some major changes to the way GReasy operates but would then allow for it to be used by a larger community.

7.2.2 Completely Heterogeneous Systems

The enhancements to GNU Radio were made with the paradigm of adding other hardware accelerators in the future. It should be possible to use the same settings and enable DSPs or GPUs to run alongside software blocks with GReasy. The idea of a system where CPUs, FPGAs, GPUs, and DSPs all work together to easily build the most efficient radio hardware for a given design is starting to become a reality.

Work was done at Virginia Tech to try and create a measure for how a completely heterogeneous system could choose among the best hardware available to create a software-defined radio [5]. The work done in the GReasy project for this thesis has enabled FPGAs and CPUs to interact in a simple and effective way. The current system still forces the designer to decide what blocks to put on what hardware, but soon perhaps the radios will be able to choose for themselves what to put where for a useful design.

Bibliography

[1] Asier Alonso. GNU Radio Tutorials, 2012. `http://gnuradio.org/redmine/projects/gnuradio/wiki/Tutorials`.

[2] Altera. Altera FPGAs, 2012. `http://www.altera.com/devices/fpga/fpga-index.html`.

[3] Kiarash Amiri, Yang Sun, Patrick Murphy, Chris Hunter, Joseph R. Cavallaro, and Ashutosh Sabharwal. WARP, a unified wireless network testbed for education and research. *Microelectronics Systems Education, IEEE International Conference on*, 0, 2007.

[4] Prabhaav Bhardwaj. Framework for hardware agility on FPGAs. Master's thesis, Virginia Polytechnic Institue and State University, 2010.

[5] Frank B. Bieberly. *Heterogeneous Processing in Software Dened Radio: Flexible Implementation and Optimal Resource Mapping*. PhD thesis, Virginia Polytechnic Institue and State University, 2012.

[6] Eric Blossom. Exploring GNU Radio, 2004. `http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html#{}fpga`.

[7] Eric Blossom. GNU Radio, 2012. `http://gnuradio.org`.

[8] Josh Blum. GNU Radio Companion, 2012. `http://gnuradio.org/redmine/projects/gnuradio/wiki/GNURadioCompanion`.

[9] Johnathan Corgan. USRP FPGA Verilog, 2011. `http://gnuradio.org/redmine/projects/gnuradio/wiki/UsrpFAQFpgaVerilog`.

[10] Johnathan Corgan. USRP Intro, 2012. `http://gnuradio.org/redmine/projects/gnuradio/wiki/UsrpFAQIntro`.

[11] Chris Dick. A case for using FPGAs in SDR PHY. *EE Times Design*, 2002. `http://www.eetimes.com/design/communications-design/4142853/A-case-for-using-FPGAs-in-SDR-PHY`.

[12] Matt Ettus. Ettus research, 2012. `http://www.ettus.com/`.

[13] Ronan Farrell, Magdalena Sanchez, and Gerry Corley. Software-defined radio demonstrators: An example and future trends. *International Journal of Digital Multimedia Broadcasting*, 2009, 2009. `http://www.hindawi.com/journals/ijdmb/2009/547650/`.

[14] Wireless Innovation Forum. Software defined radio, 2012. `http://www.wirelessinnovation.org/assets/documents/SoftwareDefinedRadio.pdf`.

[15] Tannous Frangieh. A design assembly framework for FPGA back-end acceleration. preprint (2012).

[16] Tannous Frangieh. *Design Assembly Techniques for FPGA Back-End Acceleration.* PhD thesis, Virginia Polytechnic Institue and State University, In Progress.

[17] Tannous Frangieh, Richard Stroop, Peter Athanas, and Teresa Cervero. A modular-based assembly framework for autonomous reconfigurable systems. In Oliver Choy, Ray Cheung, Peter Athanas, and Kentaro Sano, editors, *Reconfigurable Computing: Architectures, Tools and Applications*, volume 7199 of *Lecture Notes in Computer Science*, pages 314–319. Springer Berlin / Heidelberg, 2012. 10.1007/978-3-642-28365-9_26.

[18] H. Harada. Software defined radio prototype toward cognitive radio communication systems. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 539 –547, nov. 2005.

[19] S.M.S. Hasan, R. Nealy, T.J. Brisebois, T.R. Newman, T. Bose, and J.H. Reed. Wideband RF front end design considerations for a flexible white space software defined radio. In *Radio and Wireless Symposium (RWS), 2010 IEEE*, pages 484 –487, jan. 2010.

[20] Nation Instrument. What is I/Q Data?, 2011. `http://www.ni.com/white-paper/4805/en`.

[21] Charles Irick. Enhancing GNU Radio for hardware accelerated radio design. Master's thesis, Virginia Polytechnic Institue and State University, 2010.

[22] Neenu Joseph and P. Nirmal Kumar. Realization of SDR in partial reconfigurable FPGA using different types of modulation techniques. In Natarajan Meghanathan, Nabendu Chaki, Dhinaharan Nagamalai, Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, and Geoffrey Coulson, editors, *Advances in Computer Science and Information Technology. Networks and Communications*, volume 84 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 83–89. Springer Berlin Heidelberg, 2012. 10.1007/978-3-642-27299-8_9.

[23] Jorg Lotze, Suhaib A. Fahmy, Juanjo Noguera, and Linda E. Doyle. A model-based approach to cognitive radio design. *IEEE Journal on Selected Areas in Communications*, 29(2), 2011.

[24] Andrew Love. *Why tFlow is better than qFlow; an Epic in three parts.* PhD thesis, Virginia Polytechnic Institue and State University, In Progress.

[25] Ubuntu Manuals. crc32, 2005. `http://manpages.ubuntu.com/manpages/lucid/man1/crc32.1.html`.

[26] MathWorks. Simulink, simulation and model-based design, 2012. `http://www.mathworks.com/products/simulink/`.

[27] G.J. Minden, J.B. Evans, L. Searl, D. DePardo, V.R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A.M. Wyglinski, and A. Agah. KUAR: A flexible software-defined radio development platform. In *New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on*, pages 428 –439, april 2007.

[28] S.M. Mishra, D. Cabric, C. Chang, D. Willkomm, B. van Schewick, S. Wolisz, and B.W. Brodersen. A real time cognitive radio testbed for physical and link layer experiments. In *New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005. 2005 First IEEE International Symposium on*, pages 562 –567, nov. 2005.

[29] The Comprehensive GNU Radio Archive Network. Available projects, 2012. `https://www.cgran.org/wiki/Projects`.

[30] Nvidia. What is GPU Computing, June 2012. `http://www.nvidia.com/object/what-is-gpu-computing.html`.

[31] Karl Pereira. Characterization of FPGA-based high performance computers. Master's thesis, Virginia Polytechnic Institue and State University, 2011.

[32] William Plishker, George F. Zaki, Shuvra S. Bhattacharyya, Charles Clancy, and John Kuykendall. Applying graphics processor acceleration in a software dened radio prototyping environment. In *Proceedings of the International Symposium on Rapid System Prototyping*. IEEE, 2011. `http://www.ece.umd.edu/DSPCAD/papers/plis2011x1.pdf`.

[33] V. Rajaraman. Digital signal processors. *Resonance*, 4:57–66, 1999. `http://dx.doi.org/10.1007/BF02834636`.

[34] Jeffrey H. Reed. *Software Radio: A Modern Approach to Radio Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, 2002.

[35] M. Shelburne, C. Patterson, P. Athanas, M. Jones, B. Martin, and R. Fong. Metawire: Using fpga configuration circuitry to emulate a network-on-chip. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 257 –262, sept. 2008.

[36] Neil Steiner. Tools for open reconfigurable computing, October 2011. `http://torc-isi.sourceforge.net/index.php`.

[37] Richard Stroop. Creating a GReasy block, 2012. `http://www.ccm.ece.vt.edu/usvn/svn/GReasy/trunk/docs/guides/Creating_A_GReasy_Block.doc`.

[38] Kun Tan, Jiansong Zhang, Ji Fang, He Liu, Yusheng Ye, Shen Wang, Yongguang Zhang, Haitao Wu, Wei Wang, and Geoffrey M. Voelker. Sora: high performance software radio using general purpose multi-core processors. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 75–90, Berkeley, CA, USA, 2009. USENIX Association. `http://dl.acm.org/citation.cfm?id=1558977.1558983`.

[39] Wireless @ Virginia Tech. OSSIE: SCA-based open source software defined radio, 2012. `http://ossie.wireless.vt.edu/`.

[40] T. Ulversoy. Software defined radio: Challenges and opportunities. *Communications Surveys Tutorials, IEEE*, 12(4):531 –550, quarter 2010.

[41] Xilinx. All programmable, 2012. `http://www.xilinx.com`.

[42] Xilinx. System Generator for DSP, 2012. `http://www.xilinx.com/tools/sysgen.htm`.

Appendix A

EdifWriter.cpp

Listing A.1: /qflow/applications/EdifWriter.cpp

```
// Torc - Copyright 2011 University of Southern California.  All Rights
    Reserved.
// This program is free software: you can redistribute it and/or modify
    it under the terms of the
// GNU General Public License as published by the Free Software
   Foundation, either version 3 of the
// License, or (at your option) any later version.
//
// This program is distributed in the hope that it will be useful, but
   WITHOUT ANY WARRANTY;
// without even the implied warranty of MERCHANTABILITY or FITNESS FOR
   A PARTICULAR PURPOSE.  See
// the GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
   along with this program.  If
// not, see <http://www.gnu.org/licenses/>.

/// \brief Program to read in an edif.dat file and convert it to an
   EDIF file.

#include "torc/Generic.hpp"
#include "torc/Common.hpp"
#include <fstream>
#include <boost/regex.hpp>
#include <iostream>

```

```cpp
using namespace std;
using namespace torc::generic;

//Removes leading zeros from a number in a string "0033" = "33"
string removePaddedZeroes(string num) {
  int count = 0;
  for(unsigned int iter = 0; iter < num.length(); iter++) {
    if(num.at(iter) == '0')
      count++;
    else break; }
  num.erase(0,count);
  return num;
}

//Subtracts 1 from a number in a string "33" = "32"
string SM1_2(string num) {
  unsigned int iter = num.length() -1;
  while(iter >= 0) {
    if(num.at(iter) > 57 || num.at(iter) < 48)
      return "error";
    else if(num.at(iter) != '0') {
      num.at(iter) = (num.at(iter) - 1);
      return num;
    }
    else num.at(iter) = '9';
    iter--;
  }
  return num;
}


int main(int argc, char* argv[]) {

  //Generate Objects to build edif
  boost::shared_ptr<ObjectFactory> factoryPtr(new ObjectFactory());
  boost::shared_ptr<Root> rootPtr;
  factoryPtr->create(rootPtr);
  rootPtr->setName("blacktop");

  torc::generic::PortDirection input = ePortDirectionIn;
  torc::generic::PortDirection output = ePortDirectionOut;
  torc::generic::View::Type netlist = View::eTypeNetlist;

  boost::shared_ptr<Library> lib;
  boost::shared_ptr<Cell> cell;
  boost::shared_ptr<View> view;
  boost::shared_ptr<Property> property;
  boost::shared_ptr<ScalarPort> port;
  boost::shared_ptr<VectorPort> port_array;
  boost::shared_ptr<VectorPortReference> vectorRef;
  boost::shared_ptr<ScalarPortReference> scalarRef;
  boost::shared_ptr<SingleInstance> instance;
  vector< boost::shared_ptr<SingleInstance> > instances;
  boost::shared_ptr<ScalarNet> net;
  boost::shared_ptr<Design> design;

  //Add UNISIM library (qFlow no longer needs this, but may again in
      the future)
/*  factoryPtr->create(lib);
  lib->setName("UNISIMS");
  lib->setIsExtern(true);
  factoryPtr->create(cell);
  cell->setName("GND");
  factoryPtr->create(view);
  view->setName("view_1");
  view->setType(netlist);
  factoryPtr->create(port);
  port->setName("G");
  port->setDirection(output);
  view->addPort(port);
  factoryPtr->create(instance);
  instance->setName("XST_GND");
  instance->bindToMasterView(view);
  instances.push_back(instance);
  cell->addView(view);
  lib->addCell(cell);
  factoryPtr->create(cell);
  cell->setName("VCC");
  factoryPtr->create(view);
  view->setName("view_1");
  view->setType(netlist);
  factoryPtr->create(port);
  port->setName("P");
  port->setDirection(output);
  view->addPort(port);
  factoryPtr->create(instance);
  instance->setName("XST_VCC");
  instance->bindToMasterView(view);
  instances.push_back(instance);
  cell->addView(view);
  lib->addCell(cell);
  rootPtr->addLibrary(lib);
*/

  //Create Blacktop library!!!
  factoryPtr->create(lib);

  vector<string> lines;
  string line;
  if(argc > 1)
  {
    ifstream myfile(argv[1]);
    if (myfile.is_open())
    {
      while ( myfile.good() )
      {
        getline (myfile,line);
        if(line.find("Cell") != string::npos) lines.push_back(line);
        if(line.find("Net") != string::npos) lines.push_back(line);
        if(line.find("Loop") != string::npos) lines.push_back(line);
        //cout << line << endl;
      }
      myfile.close();
    }
  }
  else
  {
    ifstream myfile ("edif.dat");
    if (myfile.is_open())
    {
      while ( myfile.good() )
      {
        getline (myfile,line);
        if(line.find("Cell") != string::npos) lines.push_back(line);
        if(line.find("Net") != string::npos) lines.push_back(line);
        if(line.find("Loop") != string::npos) lines.push_back(line);
        //cout << line << endl;
      }
      myfile.close();
    }
  }
  size_t pos;
  unsigned int i;
  unsigned int iter=0;
  bool allClear = true;
  vector<vector<string> > result(lines.size());
  while(iter<lines.size())
  {
    line = lines.at(iter);

    //Split up the information to handle as we go
    pos = 0;
    while( true ) {
      size_t nextPos = line.find(';',pos);
      if( nextPos == line.npos )
        break;
      result[iter].push_back( string( line.substr( pos, nextPos - pos )
          ) );
      pos = nextPos + 1;
    }

    i=0;
    while(i<result[iter].size())
    {
      allClear = true;
      //Start by finding Cells
      if(result[iter].at(i).compare("Cell")==0)
      {
        if(lib->findCell(result[iter].at(i+1))==NULL)
        {
          factoryPtr->create(cell);
          cell->setName(result[iter].at(i+1));
          factoryPtr->create(view);
          view->setName("view_1");
          view->setType(netlist);

          //Create Instances
          factoryPtr->create(instance);
          instance->setName(result[iter].at(i+2));
          instance->bindToMasterView(view);
          instances.push_back(instance);

          i=i+3;
        }
        else
        {
          //Create Instances
          factoryPtr->create(instance);
          instance->setName(result[iter].at(i+2));
          instance->bindToMasterView(lib->findCell(result[iter].at(i+1)
             )->findView("view_1"));
          instances.push_back(instance);

          i=result[iter].size();
          allClear=false;
        }
      }
      //Find Ports
      else if(result[iter].at(i).compare("Port")==0)
      {
        factoryPtr->create(port);
        port->setName(result[iter].at(i+2));
        if(result[iter].at(i+1).compare("input")==0) port->setDirection
           (input);
        if(result[iter].at(i+1).compare("output")==0) port->
           setDirection(output);
  view->addPort(port);
        i=i+3;

      }
      //Find Arrays
      else if(result[iter].at(i).compare("Array")==0)
      {
        factoryPtr->create(port_array);
        port_array->setName(result[iter].at(i+2));
        // the following block decrements the string "test" by 1, valid
            for all numbers 1->999
        std::string arrLen = result[iter].at(i+3);
        arrLen = SM1_2(removePaddedZeroes(arrLen));
        if(arrLen == "error")
          cerr << "Error: string contained characters that are not
             numbers.\n";
        port_array->setOriginalName(result[iter].at(i+2)+"<"+arrLen+"
           :0>");
        std::vector<size_t> limits;
        limits.push_back(atoi(result[iter].at(i+3).c_str()));
        port_array->constructChildren( factoryPtr, limits);
        if(result[iter].at(i+1).compare("input")==0) port_array->
           setDirection(input);
        if(result[iter].at(i+1).compare("output")==0) port_array->
           setDirection(output);
        view->addPort(port_array);
        i=i+4;
      }
      //This combines nets that have the same input source
      else if(result[iter].at(i).compare("Net")==0)
      {
  /* compare to all other nets created */
  for(unsigned int checker=0; checker<iter; checker++){
    if(result[checker].at(0).compare("Net")==0 && result[checker].at(3)
       .compare(result[iter].at(3))==0 && result[checker].at(4).compare
       (result[iter].at(4))==0){
      for(unsigned int appender=6; appender<result[iter].size();
         appender++){
        result[checker].push_back(result[iter].at(appender));
      }
      result.erase(result.begin()+iter);
      lines.erase(lines.begin()+iter);
      iter--;
      break;
    }
  }
  i=result[iter].size();
        allClear=false;
      }
      // This combines loops that have the same input source
      else if(result[iter].at(i).compare("Loop")==0)
      {
  /* compare to all other loops created */
  for(unsigned int checker=0; checker<iter; checker++){
    if(result[checker].at(0).compare("Loop")==0 && result[checker].at
       (4).compare(result[iter].at(4))==0 && result[checker].at(5).
       compare(result[iter].at(5))==0){
      for(unsigned int appender=10; appender<result[iter].size();
         appender++){
        result[checker].push_back(result[iter].at(appender));
      }
      result.erase(result.begin()+iter);
      lines.erase(lines.begin()+iter);
      iter--;
      break;
    }
  }
        i=result[iter].size();
        allClear=false;
      }
      else
      {
        cerr << "Invalid line of input: " << line << endl;
        return 0;
      }
    }
    if(allClear)
    {
      cell->addView(view);
      lib->addCell(cell);
    }
    iter++;
  }


  //Add instances (all of them should go here)
  for(unsigned int it=0; it<instances.size()-1; it++)
  {
    //cout << it << " of " << instances.size() << endl;
    view->addInstance(instances.at(it));
  }

  //Create Nets
  int loopLength;
  char numstr[21]; // enough to hold all numbers up to 64-bits
  VectorPortBit::List list;
  VectorPortBitReference::List listRef;
  std::vector<size_t> limits;
  iter=0;
  while(iter<lines.size())
  {
    line = lines.at(iter); //just used for output if there is an error

    i=0;
    while(i<result[iter].size())
    {
      //Start with Nets
      if(result[iter].at(i).compare("Net")==0)
      {
        factoryPtr->create(net);
        net->setName(result[iter].at(i+1));
        i=i+2;
        while(i<result[iter].size())
        {
          //Connections to the static are handled first
          if(result[iter].at(i).compare("blacktop")==0)
          {
            if(atoi(result[iter].at(i+3).c_str())==-1)
            {
              view->findPort(result[iter].at(i+2))->connect(net);
            }
            else
            {
              list.clear();
              view->findPort(result[iter].at(i+2))->getChildren(list);
              list.at(atoi(result[iter].at(i+3).c_str()))->connect(net)
                  ;
            }
          }
          //Unisim Connections
          else if(result[iter].at(i).compare("VCC")==0)
          {
            factoryPtr->create(scalarRef);
            scalarRef->bindToMasterPort(rootPtr->findLibrary("UNISIMS")
                ->findCell(result[iter].at(i))->findView("view_1")->
                findPort("P"));
            view->findInstance("XST_VCC")->addPortReference(scalarRef);
            scalarRef->connect(net);
          }
          else if(result[iter].at(i).compare("GND")==0)
          {
            factoryPtr->create(scalarRef);
            scalarRef->bindToMasterPort(rootPtr->findLibrary("UNISIMS")
                ->findCell(result[iter].at(i))->findView("view_1")->
                findPort("G"));
            view->findInstance("XST_GND")->addPortReference(scalarRef);
            scalarRef->connect(net);
          }
          //All other Nets to each other inside of the sandbox
          else
          {
            //This will be the case most of the time
            //-1 means it is actually just a Port
            if(atoi(result[iter].at(i+3).c_str())==-1)
            {
              factoryPtr->create(scalarRef);
              scalarRef->bindToMasterPort(lib->findCell(result[iter].at
                  (i))->findView("view_1")->findPort(result[iter].at(i
                  +2)));
              view->findInstance(result[iter].at(i+1))->
                  addPortReference(scalarRef);
              scalarRef->connect(net);
            }
```

```
            //However if you want to connect just one net from an array
            //just put the bit number you would like to connect
            else
            {
              list.clear();
              lib->findCell(result[iter].at(i))->findView("view_1")->
                 findPort(result[iter].at(i+2))->getChildren(list);
              list.at(atoi(result[iter].at(i+3).c_str()))->setParent(
                 lib->findCell(result[iter].at(i))->findView("view_1"))
                 ;
              if(view->findInstance(result[iter].at(i+1))->
                 findPortReference(result[iter].at(i+2))==NULL)
              {
                factoryPtr->create(vectorRef);
                vectorRef->bindToMasterPort(lib->findCell(result[iter].
                   at(i))->findView("view_1")->findPort(result[iter].at
                   (i+2)));
                limits.push_back(list.size());
                vectorRef->constructChildren( factoryPtr, limits);
                limits.pop_back();
                listRef.clear();
                vectorRef->getChildren(listRef);
                view->findInstance(result[iter].at(i+1))->
                   addPortReference(vectorRef);
                listRef.at(atoi(result[iter].at(i+3).c_str()))->connect
                   (net);
              }
              else
              {
                listRef.clear();
                view->findInstance(result[iter].at(i+1))->
                   findPortReference(result[iter].at(i+2))->getChildren
                   (listRef);
                listRef.at(atoi(result[iter].at(i+3).c_str()))->connect
                   (net);
              }
            }
          }
          i=i+4;
        }
        view->addNet(net);
      }
      //Connect the loops last
      else if(result[iter].at(i).compare("Loop")==0)
      {
        loopLength = atoi(result[iter].at(i+2).c_str());
        //ISE does this exactly backwards counting from (length-1) down
            to 0
        for(int loop=0; loop<loopLength; loop++)
        {
          factoryPtr->create(net);
          sprintf(numstr, "%d", loop);
          net->setName(result[iter].at(1)+"_"+numstr+"_");
          net->setOriginalName(result[iter].at(1)+"<"+numstr+">");

          i=3;
          while(i<result[iter].size())
          {
            //connections to the static region are handled first
            if(result[iter].at(i).compare("blacktop")==0)
            {
              list.clear();
              view->findPort(result[iter].at(i+2))->getChildren(list);
              list.at(loop)->connect(net);
            }
            //Then all other loops, there should not be unisims here
            else
            {
              list.clear();
              lib->findCell(result[iter].at(i))->findView("view_1")->
                 findPort(result[iter].at(i+2))->getChildren(list);
              list.at(loop)->setParent(lib->findCell(result[iter].at(i)
                 )->findView("view_1"));
              if(view->findInstance(result[iter].at(i+1))->
                 findPortReference(result[iter].at(i+2))==NULL)
              {
                factoryPtr->create(vectorRef);
                vectorRef->bindToMasterPort(lib->findCell(result[iter].
                   at(i))->findView("view_1")->findPort(result[iter].at
                   (i+2)));
                limits.push_back(list.size());
                vectorRef->constructChildren( factoryPtr, limits);
                limits.pop_back();
                view->findInstance(result[iter].at(i+1))->
                   addPortReference(vectorRef);
                listRef.clear();
                vectorRef->getChildren(listRef);
                listRef.at(loop)->connect(net);
              }
              else
              {
                listRef.clear();
                view->findInstance(result[iter].at(i+1))->
                   findPortReference(result[iter].at(i+2))->getChildren
                   (listRef);
                listRef.at(loop)->connect(net);
              }
            }
            i=i+3;
          }
          view->addNet(net);
        }

      }
      //Ignore Cells now
      else if(result[iter].at(i).compare("Cell")==0)
      {
        i=result[iter].size();
      }
      else
      {
        cerr << "Invalid line of input: " << line << endl;
        return 0;
      }
    }
    iter++;
  }

  /* Uncomment to output Port list for Debugging
  vector<PortSharedPtr> portList;
  view->getPorts(portList);
  for(unsigned int it=0; it<portList.size(); it++)
  {
    cout << portList.at(it)->getName() << endl;
  }
  */


  lib->setName("blacktop_lib");
  rootPtr->addLibrary(lib);

  factoryPtr->create(design);
  design->setName("blacktop");
  design->setCellRefName("blacktop");
  design->setLibraryRefName("blacktop_lib");
```

```


  factoryPtr->create(property);
  Value value;
  value.setType(Value::eValueTypeString);
  //FIXME
  //The board type is hard-coded into this EdifWriter, this should
     eventually be an input
  value.set<Value::String>("xc5vlx110t-1-ff1136");
  value.setIsSet(true);

  property->setOwner("Xilinx");
  property->setValue(value);
  property->setName("PART");
  design->setProperty("PART",property);

  rootPtr->addDesign(design);

  // export the EDIF design
  string outFileName = "blacktop.ndf";
  fstream edifExport(outFileName.c_str(), ios_base::out);
  EdifExporter exporter(edifExport);
  exporter(rootPtr);

  return 0;
}
```

Appendix B

edif_connector.h

Listing B.1: /gnuradio/gnuradio-core/src/lib/runtime/edif_connector.h

```
/*
Connects edif files together and handles networking
*/


#ifndef INCLUDED_EDIF_CONNECTOR_H
#define INCLUDED_EDIF_CONNECTOR_H


#include <stdexcept>
#include <iostream>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <fstream>
#include <vector>

#include "PacketCreator.h"

int status; //for system calls

//Used for sending control data between multiple FPGAs
std::vector<std::string> dest_macs;
std::vector<std::string> src_macs;
std::vector<std::string> valid_macs;

//Converts string integers to actual integers for math purposes
```

```cpp
int convertInt(std::string number)
{
  int temp = 0;
  for(unsigned int i=0; i<number.size(); i++){
    temp *= 10;
    temp += int(number.at(i))-48;
  }

  return temp;
}

//Read the edif_connections list to find the port name associated with a port number in GRC
std::vector<std::vector<std::string> > edif_connections;
bool getConn(std::string cell, std::string type, std::string direction, int port_number, std::string &size, std::string &name, int &cell_number){
  for(unsigned int connPos=0; connPos<edif_connections.size(); connPos++){
    //if(strcmp(edif_connections[connPos].at(0).c_str(),cell.c_str()) == 0){
    if(edif_connections[connPos].at(0) == cell){
      int count=-1;
      for(unsigned int index=1; index<edif_connections[connPos].size(); index++){
        if(edif_connections[connPos].at(index-1) == type && edif_connections[connPos].at(index) == direction){
          count++;
          if(count==port_number){
            //size
            if(strcmp(type.c_str(),"Array")==0) size = edif_connections[connPos].at(index+2);
            else size = "-1";
            //name
            name = edif_connections[connPos].at(1) + ";" + edif_connections[connPos].at(2) + ";" + edif_connections[connPos].at(index+1) + ";";
            cell_number = connPos;
            return true;
          }
        }
      }
      break;
    }
  }
  return false;
}



bool connect_edif(){

  int size=0;/*{{{*/
  std::string cell;
  std::ifstream myfile ("/tmp/fpga_connections.txt");
  //Read in the original file created by the GRC constructor blocks to
     get the number of connections
  if (myfile.is_open()){
    while ( myfile.good() ){
      size++;
      std::getline (myfile,cell);
    }
    myfile.close();
  }
  else{
    //no connections file, thus no afpga connections
    //let GNU Radio continue going
    return true;
  }
  //fpga_connections.txt contains GNURadios connections but for some
     reason contains double copies.  After this bit of code, line[
     valid_connections] contains only one version of each connection
     which is printed to connections.txt.
  //Read in the file again to get only one copy of each connection
  std::ifstream myfile2("/tmp/fpga_connections.txt");
  std::string * line;
  line = new std::string[size];
  int valid_connections=0;
  if (myfile2.is_open()){
    while ( myfile2.good() ){
      std::getline (myfile2,line[valid_connections]);
      //Check to make sure the line does not exist already
      if(&line[valid_connections]==std::find(line,&line[size],line[
         valid_connections]))
      {
        valid_connections++;
      }
    }
    myfile2.close();
    status = system("sudo rm /tmp/fpga_connections.txt");
    status = system("sudo rm /tmp/connections.txt");
  }


  bool contains_afpga = false;
  std::ofstream connections;
  connections.open ("/tmp/connections.txt", std::ios::app);
  for(int valid_connection=0; valid_connection<valid_connections-1;
     valid_connection++){
    connections << line[valid_connection] << "\n";
    if(line[valid_connection].find("afpga") != std::string::npos)
       contains_afpga = true;
  }
  connections.close();

  //No need to continue if there is no afpga modules connected
  if(!contains_afpga){
    //There was a connections file but no blocks were meant for the
       fpga device
    //let GNU Radio continue going
    return true;
  }


  std::vector<std::vector<std::string> > interfaces;
  std::ifstream edifReader("/tmp/edif.dat");
  unsigned int connPos = -1;
  int macsPos = -1;
  bool addedBT = false;
  //std::cout << "getting Cells" << std::endl;
  //Populate the edif_connections list with Cell specific information
  if (edifReader.is_open()){
    while(edifReader.good() || !addedBT){
      //On the first call, the blacktop interfaces need to be set.
         This should be identical between boards so it is hard-coded
         here.  It could be made to be customizable based on the static
          being used on any given board.
      if(edifReader.good()) std::getline(edifReader,cell);
      else{
        cell = "Cell;blacktop;BT;0;Array;input;in0;33;Array;input;in1
           ;33;Array;output;out0;33;Array;output;out1;33;Port;input;rst
           ;Port;input;clk;";
        addedBT = true;
      }
      if(cell.find("Cell") != std::string::npos){
        size_t pos = 0;
        edif_connections.push_back(std::vector<std::string>());
        connPos++;
        while( true ) {
          size_t nextPos = cell.find(';',pos);
          if( nextPos == cell.npos )
            break;
          edif_connections[connPos].push_back( std::string( cell.substr
             ( pos, nextPos - pos ) ) );
          pos = nextPos + 1;
        }
        edif_connections[connPos].erase(edif_connections[connPos].begin
           ());
        edif_connections[connPos].insert(edif_connections[connPos].
           begin(),"afpga_"+edif_connections[connPos].at(0) + "_" +
           edif_connections[connPos].at(1) + "(" + edif_connections[
           connPos].at(2) + ")");

        // Combines the instance name with the unique id for a unique
           instance name amoug cells
        edif_connections[connPos].insert(edif_connections[connPos].
           begin()+2, edif_connections[connPos].at(2)+edif_connections[
           connPos].at(3) );
        // Delete the original instance name, moving the id to the
           third position
        edif_connections[connPos].erase(edif_connections[connPos].begin
           ()+3);
        // Now remove the id
        edif_connections[connPos].erase(edif_connections[connPos].begin
           ()+3);
      }
      //used to get Ethernet interface information for each FPGA
      if(cell.find("PARAM") != std::string::npos){
        size_t pos = 0;
        interfaces.push_back(std::vector<std::string>());
        macsPos++;
        while( true ) {
          size_t nextPos = cell.find(';',pos);
          if( nextPos == cell.npos )
            break;
          interfaces[macsPos].push_back( std::string( cell.substr( pos,
              nextPos - pos ) ) );
          pos = nextPos + 1;
        }
      }
```

```
    }
  edifReader.close();
  }
  else{
    //If there were connections, there should be an edif.dat file, so
       if cannot be opened there is a file permissions problem!
    std::cout << "could not open edif.dat" << std::endl;
    return false;
  }

  //std::cout << "getting connections" << std::endl;
  std::ifstream connReader("/tmp/connections.txt");
  std::vector<std::string > searches;
  std::vector<std::string > numbers;
  // Read GNU Radio's connection list, based on names and port numbers
     only
  if (connReader.is_open()){
    while(connReader.good()){
      //If you ever decide to get rid of connections.txt, just replace
         this while loop with a for loop of size lines, and put each
         line[valid_connections] into the string cell instead of the
         following getline.
      std::getline(connReader,cell);
      if(cell.find("afpga") != std::string::npos){
        std::string input;
        std::string output;
        std::string input_search;
        std::string output_search;
        std::string input_number;
        std::string output_number;
        size_t splitPos = cell.find(',',0);
        input = std::string( cell.substr( 0, splitPos ) );
        output = std::string( cell.substr( splitPos+1, cell.npos-
           splitPos ) );
        splitPos = input.find("):",0);
        splitPos++;
        input_search = std::string( input.substr( 0, splitPos ) );
        input_number = std::string( input.substr( splitPos+1, input.
           npos-splitPos ) );
        splitPos = output.find("):",0);
        splitPos++;
        output_search = std::string( output.substr( 0, splitPos ) );
        output_number = std::string( output.substr( splitPos+1, output.
           npos-splitPos ) );

        // Dont add if the input is afpga_out, this is handled by the
           host
        // Same if the output is afpga_in
        if(input_search.find("afpga_out",0)==std::string::npos &&
           output_search.find("afpga_in",0)==std::string::npos){
          searches.push_back(input_search);
          searches.push_back(output_search);
          numbers.push_back(input_number);
          numbers.push_back(output_number);
        }
        if(input_search.find("afpga_out",0)!=std::string::npos &&
           output_search.find("afpga_in",0)!=std::string::npos){
    dest_macs.push_back( std::string( output_search.substr( 9, 17 ) ) )
       ;
    src_macs.push_back( std::string( input_search.substr( 10, 17 ) ) );
  }

        //std::cout << input_search << "+" << input_number << "  <->  "
            << output_search << "+" << output_number << std::endl;
      }
    }
  }
  else{
    //This really is not possible, if we were able to write it, we
       better be able to read it.  Good luck solving this error if it
       ever breaks here :-)
    std::cout << "could not open connections.txt" << std::endl;
    return false;
  }


  //Create our connections data structure
  std::vector<std::vector<std::string> > loops_nets;
  std::vector<std::string > extras;
  std::vector<std::vector<int> > cell_tracker;
  //While we haven't checked everything
  while(searches.size() !=0){
    //Go through based on each afpga_in path, which should have a
       logical end.
    loops_nets.push_back(std::vector<std::string>());
    cell_tracker.push_back(std::vector<int>(edif_connections.size(),0))
       ;
    std::string search;
    search = "afpga_in";
    unsigned int i=0;
```

```cpp
    std::string mac_check;
    mac_check = "";
    int port_number;
    std::string port_string;
    std::string input_size,input_name,output_size,output_name;
    bool not_found=true;
    int cell_number=-1;
    while(i<searches.size()){
      //std::cout << i << " < " << searches.size() << std::endl;
      if(searches.at(i).find(search) != std::string::npos){
        std::cout << searches.at(i) << " -> ";
        if(mac_check.size()==0){
          //only check 14 characters of the 17, the colon and the last 2 only matter for ports
          mac_check = std::string( searches.at(i).substr( 9, 14 ) );
          // Store the MAC check at the very first arguement for use in generating a unique filename
          loops_nets[loops_nets.size()-1].push_back(mac_check);
          port_string = std::string( searches.at(i).substr( 9+14+1, 2) );
          port_number = convertInt( port_string );
          // Erase the last colon and 2 digits of MAC plus the unique-id so that the extras search only looks for other afpga_ins that have the same beginning mac
          searches.at(i).erase( 9+14, 6);
          //std::cout << mac_check << " " << port_number << std::endl;
          std::string insert = "Loop;BT_in_";
          insert += port_string;
          insert += ";";
          // Check BT ports
          if(!getConn("afpga_blacktop_BT(0)","Array","input", port_number, input_size, input_name, cell_number)){
            std::cerr << "Could not find input port " << port_number << " of blacktop" << std::endl;
            return false;
          }
          cell_tracker[cell_tracker.size()-1].at(cell_number) = 1;
          if(searches.at(i+1).find("afpga_out") != std::string::npos){
            // Check BT ports and size
            port_number = convertInt( std::string( searches.at(i+1).substr( 10+14+1, 2) ));
            if(!getConn("afpga_blacktop_BT(0)","Array","output", port_number, output_size, output_name, cell_number)){
              std::cerr << "Could not find output port " << port_number << " of blacktop" << std::endl;
              return false;
            }
          }
          else {
            // Check output ports and size
            if(!getConn(searches.at(i+1),"Array","input", convertInt(numbers.at(i+1)), output_size, output_name, cell_number)){
              std::cerr << "Could not find port " << convertInt(numbers.at(i+1)) << " of " << searches.at(i+1) << std::endl;
              return false;
            }
          }
          cell_tracker[cell_tracker.size()-1].at(cell_number) = 1;
          // Check size alignment
          if(input_size == output_size){
            // Insert into the loops_nets
            insert += input_size;
            insert += ";";
            insert += input_name;
            insert += output_name; //this is Cell;Instance;Name not just name
          }
          else{
            std::cerr << "Size of " <<input_name<< " and " << output_name<< " do NOT match" << std::endl;
            return false;
          }
          // Actually insert the loop
          loops_nets[loops_nets.size()-1].push_back(insert.c_str());
          //and do this for the common_case and end_case
        }
        else {
          //std::cout << ”inserting loop” << std::endl;
          std::string insert = "Loop;";
          if(searches.at(i).find("afpga_in") != std::string::npos){
            // Check BT ports and size
            port_string = std::string( searches.at(i).substr( 9+14+1, 2) );
            port_number = convertInt( port_string );
            if(!getConn("afpga_blacktop_BT(0)","Array","input", port_number, output_size, output_name, cell_number)){
              std::cerr << "Could not find input port " << port_number << " of blacktop" << std::endl;
              return false;
            }
            insert += "BT_in_";
            insert += port_string;
            insert += ";";
          }
          else{
            //block_name(id) => block_name_id
            search.replace(search.find('('),1,"_");
            search.replace(search.find(')'),1,"_wire");
            insert += search;
            insert += "_";
            insert += numbers.at(i);
            insert += ";";
            // Check output ports
            if(!getConn(searches.at(i),"Array","output", convertInt(
               numbers.at(i)), output_size, output_name, cell_number)){
              std::cerr << "Could not find port " << convertInt(numbers
                 .at(i)) << " of " << searches.at(i) << std::endl;
              return false;
            }
          }
          cell_tracker[cell_tracker.size()-1].at(cell_number) = 1;
          if(searches.at(i+1).find("afpga_out") != std::string::npos){
            // Check BT ports and size
            port_number = convertInt( std::string( searches.at(i+1).
               substr( 10+14+1, 2) ));
            if(!getConn("afpga_blacktop_BT(0)","Array","output",
               port_number, input_size, input_name, cell_number)){
              std::cerr << "Could not find output port " <<
                 port_number << " of blacktop" << std::endl;
              return false;
            }
          }
          else {
            // Check input ports and size
            if(!getConn(searches.at(i+1),"Array","input", convertInt(
               numbers.at(i+1)), input_size, input_name, cell_number)){
              std::cerr << "Could not find port " << convertInt(numbers
                 .at(i+1)) << " of " << searches.at(i+1) << std::endl;
              return false;
            }
          }
          cell_tracker[cell_tracker.size()-1].at(cell_number) = 1;
          // Check size alignment
          if(input_size == output_size){
```

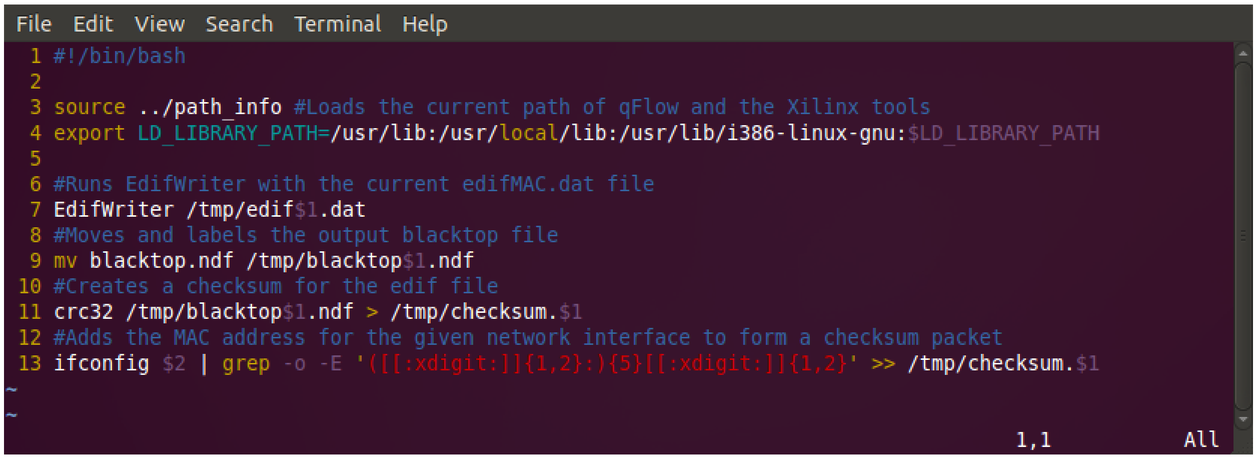
```
        // Insert into the loops_nets
        insert += output_size;
        insert += ";";
        insert += output_name;
        insert += input_name; //this is Cell;Instance;Name not just name
      }
      else{
        std::cerr << "Size of " <<output_name<< " and " << input_name<< " do NOT match" << std::endl;
        return false;
      }
      // Actually insert the loop
      loops_nets[loops_nets.size()-1].push_back(insert.c_str());

    }

    search = searches.at(i+1);

    if(search.find("afpga_out") != std::string::npos){
      if(mac_check.compare(std::string( search.substr( 10, 14 ) ))!=0){
        std::cerr << std::endl << "The MAC address of the input and output path do not match!" << std::endl;
        return false;
      }
      not_found = false; //end_case
      std::cout << search << std::endl;
      //tell these valid mac addresses to send data to the computer
      valid_macs.push_back( search.substr( 10, 17 ) );
    }
    else{
      // This section checks to see if the output block is already in the flow so that it does not have to keep searching for an end
      //block_name(id) => block_name_id
      searches.at(i+1).replace(searches.at(i+1).find('('),1,"_");
      searches.at(i+1).replace(searches.at(i+1).find(')'),1,"_wire");
      for(unsigned int ii=0; ii<loops_nets[loops_nets.size()-1].size(); ii++){
        if(loops_nets[loops_nets.size()-1].at(ii).find(searches.at(i+1)) != std::string::npos){
          not_found = false; //end_case
        }
```

```
            }
          }

          // This is designed to account for single source, multiple sinks.
          // Just checking again and creating the same loop name will combine the loop in EdifWriter
          for(unsigned int ii=i+2;ii<searches.size();ii=ii+2){
            //std::cout << "searching for: " << searches.at(ii) << std::endl;
              //std::cout << "against: " << ii << " " << searches.at(i) << std::endl;
            if(searches.at(ii).find(searches.at(i)) != std::string::npos){
              extras.push_back(searches.at(ii));
            }
          }

          searches.erase(searches.begin()+i);
          searches.erase(searches.begin()+i);//plus 1
          numbers.erase(numbers.begin()+i);
          numbers.erase(numbers.begin()+i);//plus 1

          //if search was found, start at the top of the list again looking for the next item
          i=-2;

          //unless we are done
          if(!not_found){
             i=searches.size()-2;
          }
        }// End if search was found
        i=i+2;
        //std::cout << "Searching for: " << search << " at position: " << i << " of " << searches.size() << std::endl;
        if(i==searches.size()){
          //if we have checked every connection for afpga_in
          if(search.find("afpga_in") != std::string::npos){
            //eventually this will mean you need to find a non-afpga_in source
            //for now, just end!
            loops_nets.pop_back();
            while(searches.size()!=0){
              searches.erase(searches.begin());
              numbers.erase(numbers.begin());
              //This will let the outter most WHILE loop end.
            }
          }
          //else we have checked every position and it still did not find an end_case
          else if(not_found){
              std::cerr << std::endl << "The block " << search  << " appears to be a sink, only complete paths from afpga_in to afpga_out are allowed currently" << std::endl;
              return false;
          }
          //otherwise we reached the end and there are no extras to search for
          else if(extras.size()!=0){
            //std::cout << "running extras" << std::endl;
            search = *extras.begin();
            extras.erase(extras.begin());
            i=0;
            not_found = true;
          }
        }
        //std::cout << "one cycle" << std::endl;
      }
      //std::cout << "done" << std::endl;
    }

    //std::cout << "writing edif" << std::endl;


    std::string edif_number = "";
    std::string this_mac = "";
    for(unsigned int ii=0; ii<loops_nets.size(); ii++){
      std::string edif_name = "/tmp/edif";
      std::string fpga_check = loops_nets[ii].at(0);
      edif_number = "";
      edif_number += loops_nets[ii].at(0).at(0);
      edif_number += loops_nets[ii].at(0).at(1);
      edif_number += loops_nets[ii].at(0).at(3);
      edif_number += loops_nets[ii].at(0).at(4);
      edif_number += loops_nets[ii].at(0).at(6);
      edif_number += loops_nets[ii].at(0).at(7);
      edif_number += loops_nets[ii].at(0).at(9);
      edif_number += loops_nets[ii].at(0).at(10);
      edif_number += loops_nets[ii].at(0).at(12);
      edif_number += loops_nets[ii].at(0).at(13);
      edif_name += edif_number;
      edif_name += ".dat";
      std::ofstream myfile;
      myfile.open (edif_name.c_str());//, std::ios::app);
      for(connPos=0; connPos<edif_connections.size(); connPos++){
        //If  a cell is used for this edif flow, then cell tracker will
           be 1 for that vector set
        if(cell_tracker[ii].at(connPos)==1){
          myfile << "Cell;";
          for(unsigned int i=1; i<edif_connections[connPos].size(); i++){
            myfile << edif_connections[connPos].at(i) << ";";
          }
          myfile << std::endl;
          //After adding the Cell, we add the rst and clk for the module
             as long as it is not the last one which will always be
             Blacktop
          if(connPos != edif_connections.size()-1){
            myfile << "Net;rst;blacktop;BT0;rst;-1;" << edif_connections[
               connPos].at(1) <<";"<< edif_connections[connPos].at(2) <<"
               ;"<< edif_connections[connPos].at(edif_connections[connPos
               ].size()-4) << ";-1;" << std::endl;
          //std::cout << "Net;rst;blacktop;BT0;rst;-1;" <<
             edif_connections[connPos].at(1) <<";"<< edif_connections[
             connPos].at(2) <<";" << edif_connections[connPos].at(
             edif_connections[connPos].size()-4) << ";-1;" << std::endl;
            myfile << "Net;clk;blacktop;BT0;clk;-1;" << edif_connections[
               connPos].at(1) <<";"<< edif_connections[connPos].at(2) <<"
               ;"<< edif_connections[connPos].at(edif_connections[connPos
               ].size()-1) << ";-1;" << std::endl;
          }
        }
      }
      for(unsigned int i=1; i<loops_nets[ii].size(); i++){
        myfile << loops_nets[ii].at(i) << std::endl;
      }
      myfile.close();

      std::string eth_inter = ""; //default ethernet interface
      for(unsigned int i=0; i<interfaces.size(); i++){
        //std::cout << "found mac: " << interfaces[i].at(1) << std::endl;
        if(interfaces[i].at(1).find(fpga_check) != std::string::npos){
          //std::cout << "using interface: " << interfaces[i].at(2) <<
             std::endl;
          eth_inter = interfaces[i].at(2); //set interface based on
             settings in GRC
          break;
        }
      }

      //Create edif
      std::string edif = "./edif "; //eventually these scripts should have a permanent location
      edif += edif_number;
      edif += " ";
      edif += eth_inter;
      status = system(edif.c_str());
      std::string this_checksum = "/tmp/checksum."; //this string is used for opening the file and then storing the checksum for this edif
      this_checksum += edif_number;
      std::ifstream checksum_file;
      checksum_file.open (this_checksum.c_str());
      std::getline (checksum_file,this_checksum);
      std::getline (checksum_file,this_mac);
      checksum_file.close();
      std::string fpga_mac = fpga_check + ":00";

      //Now we can read the fpga_checksum from the fpga with fpga_mac and compare it to the this_checksum on this_mac :-)
      std::cout << "this checksum: " << this_checksum << std::endl;
      std::cout << "fpga mac: " << fpga_mac << std::endl;
      std::cout << "this mac: " << this_mac << std::endl;

      char *ETH = NULL;
      ETH = (char *)eth_inter.c_str();
      //If there are valid mac address values and a checksum (so the paths are set up right)
      if(this_mac.size()>0 && fpga_mac.size()>0 && this_checksum.size()>0){
        //Start by telling the fpga where to send stuff
        PacketCreator* pc = new PacketCreator(ETH);
        //set_dest_mac_address usage (src computer mac, fpga current mac, newdestination)
        pc -> set_dest_mac_address (this_mac, fpga_mac, this_mac);
        //Then get the Checksum
        std::string fpga_checksum = "";
        fpga_checksum = pc -> get_checksum(this_mac, fpga_mac);

        std::cout << "the checksum on the board is: " << fpga_checksum << " and the flow-graph checksum is: " << this_checksum << std::
           endl;
        //The board is already programmed with the right mac, dont do
           anything!
        if(fpga_checksum == this_checksum){
          std::cout << "Skipping blacktop" << edif_number << " because
             the board appears to already have the right bitstream on it.
             " << std::endl;
        }
        else{
          //RUN qFlow

          std::string qflow = "./qflow ";
          qflow += edif_number;
          status = system(qflow.c_str());
          std::string program = "./program blacktop";
          program += edif_number;
          status = system(program.c_str());

          //Then we have to set the checksum
          pc -> set_checksum(this_checksum, this_mac, fpga_mac);
        }

        unsigned int i;
        for(i = 0; i < valid_macs.size(); i++){
          if(valid_macs[i].substr( 0, 14 ).compare(fpga_check) == 0){
            //Now we have to tell the board to send to us again because
               we reprogrammed it
            pc -> set_dest_mac_address ( this_mac, valid_macs[i],
               this_mac);
            valid_macs.erase(valid_macs.begin()+i);
            break;
          }
        }

        //IF THERE IS AN FPGA TO FPGA set DEST HERE

        for(i = 0; i < src_macs.size(); i++){
          if(src_macs[i].substr( 0, 14).compare(fpga_check) == 0){
            pc -> set_dest_mac_address ( this_mac, src_macs[i], dest_macs
               [i]);
            src_macs.erase(src_macs.begin()+i);
            dest_macs.erase(dest_macs.begin()+i);
            break;
          }
        }
```

```

    }

  }//end of fpga loop


  //remove the original constructor data
  status = system("sudo rm /tmp/edif.dat");

  /*}}}*/
 return true;

}


#endif /* INCLUDED_EDIF_CONNECTOR_H */
```

Appendix C

edif

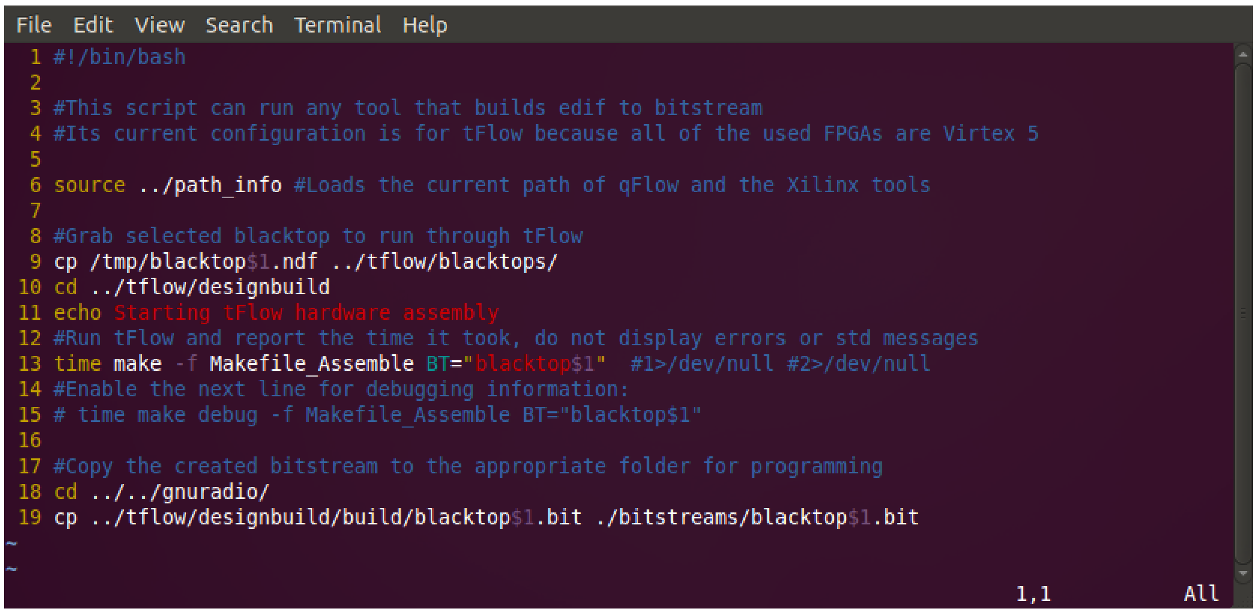


```
#!/bin/bash

source ../path_info #Loads the current path of qFlow and the Xilinx tools
export LD_LIBRARY_PATH=/usr/lib:/usr/local/lib:/usr/lib/i386-linux-gnu:$LD_LIBRARY_PATH

#Runs EdifWriter with the current edifMAC.dat file
EdifWriter /tmp/edif$1.dat
#Moves and labels the output blacktop file
mv blacktop.ndf /tmp/blacktop$1.ndf
#Creates a checksum for the edif file
crc32 /tmp/blacktop$1.ndf > /tmp/checksum.$1
#Adds the MAC address for the given network interface to form a checksum packet
ifconfig $2 | grep -o -E '([[:xdigit:]]{1,2}:){5}[[:xdigit:]]{1,2}' >> /tmp/checksum.$1
~
~
                                                                    1,1           All
```

Figure C.1: Bash Script: edif

Appendix D

qflow

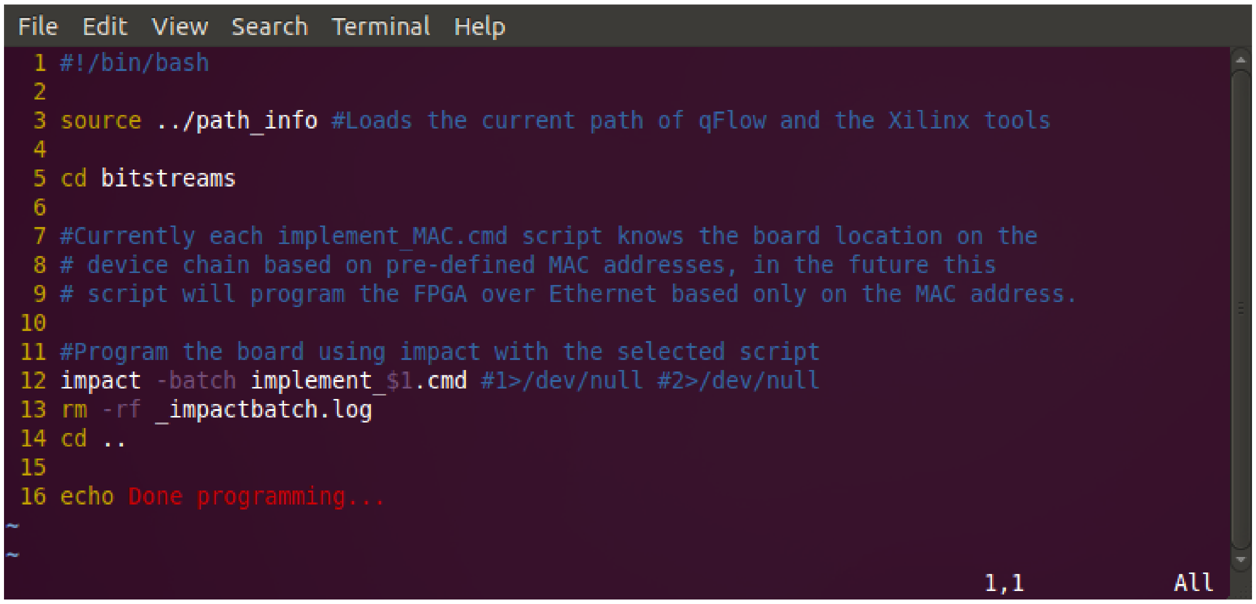


```
#!/bin/bash

#This script can run any tool that builds edif to bitstream
#Its current configuration is for tFlow because all of the used FPGAs are Virtex 5

source ../path_info #Loads the current path of qFlow and the Xilinx tools

#Grab selected blacktop to run through tFlow
cp /tmp/blacktop$1.ndf ../tflow/blacktops/
cd ../tflow/designbuild
echo Starting tFlow hardware assembly
#Run tFlow and report the time it took, do not display errors or std messages
time make -f Makefile_Assemble BT="blacktop$1"  #1>/dev/null #2>/dev/null
#Enable the next line for debugging information:
# time make debug -f Makefile_Assemble BT="blacktop$1"

#Copy the created bitstream to the appropriate folder for programming
cd ../../gnuradio/
cp ../tflow/designbuild/build/blacktop$1.bit ./bitstreams/blacktop$1.bit
~
~
                                                                   1,1           All
```

Figure D.1: Bash Script: qflow

Appendix E

program



```
#!/bin/bash

source ../path_info #Loads the current path of qFlow and the Xilinx tools

cd bitstreams

#Currently each implement_MAC.cmd script knows the board location on the
# device chain based on pre-defined MAC addresses, in the future this
# script will program the FPGA over Ethernet based only on the MAC address.

#Program the board using impact with the selected script
impact -batch implement_$1.cmd #1>/dev/null #2>/dev/null
rm -rf _impactbatch.log
cd ..

echo Done programming...
```

Figure E.1: Bash Script: program

Appendix F

PacketCreator.h

Listing F.1: /gnuradio/gnuradio-core/src/lib/runtime/PacketCreator.h

```
// PacketCreator.h

#ifndef PACKETCREATOR_H
#define PACKETCREATOR_H

#include "RawEthernet.h"
#include <iostream>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sstream>

typedef unsigned char byte;

class PacketCreator
{

private:
  RawEthernet *enet;
public:
  PacketCreator (char *ETH);
  ~PacketCreator ();

  void set_checksum (byte data[], int len, unsigned short protocol,
     byte byteSRC[], byte byteDST[]);
  void set_checksum (std::string checksum, std::string SRC, std::string DST);
  void char_array_to_byte_array (char* array1, byte* data, int n);
  std::string get_checksum (std::string SRC, std::string DST);

  void set_dest_mac_address (byte dst_mac_address_of_FPGA[], int len, unsigned short protocol, byte byteSRC[], byte byteDST[]);
  void set_dest_mac_address (std::string SRC, std::string DST, std::string newDST);
};


PacketCreator::PacketCreator(char *ETH)
{
  this-> enet = new RawEthernet(ETH);
}


PacketCreator::~PacketCreator()
{
  delete this->enet;
}

//takes a string and adds colons between values for checksum, to send the checksum to the fpga
void PacketCreator::set_checksum (std::string checksum, std::string SRC, std::string DST){

  char *tempSRC=new char[SRC.size()+1];
  tempSRC[SRC.size()]=0;
  memcpy(tempSRC,SRC.c_str(),SRC.size());
  char *tempDST=new char[DST.size()+1];
  tempDST[DST.size()]=0;
  memcpy(tempDST,DST.c_str(),DST.size());
  byte byteDST[6];
  byte byteSRC[6];
  char_array_to_byte_array(tempSRC, byteSRC, 6);
  char_array_to_byte_array(tempDST, byteDST, 6);
  unsigned short protocol = (unsigned short)0xDEAD;

  int len = 4;
  checksum.insert(2,":");
  checksum.insert(5,":");
  checksum.insert(8,":");
  char *array1=new char[checksum.size()+1];
  array1[checksum.size()]=0;
  memcpy(array1,checksum.c_str(),checksum.size());
  byte checksum_p[4];
  char_array_to_byte_array(array1, checksum_p, len);
  set_checksum (checksum_p, len, protocol, byteSRC, byteDST);
}


//takes a byte array containing checksum and sends that checksum to the
    fpga
void PacketCreator::set_checksum (byte checksum[], int len, unsigned
   short protocol, byte byteSRC[], byte byteDST[])
{
  enet->connect(byteDST);
  byte command_pkt [len + 2];
  command_pkt[0] = 0;
  command_pkt[1] = 5;
  for (int i = 2; i < len + 2; i++){
    command_pkt[i] = checksum[i-2];
  }
  enet -> sendData(command_pkt, len + 2, protocol, byteSRC);
}

std::string PacketCreator::get_checksum (std::string SRC, std::string
   DST)
{
  char *tempSRC=new char[SRC.size()+1];
  tempSRC[SRC.size()]=0;
  memcpy(tempSRC,SRC.c_str(),SRC.size());
  char *tempDST=new char[DST.size()+1];
  tempDST[DST.size()]=0;
  memcpy(tempDST,DST.c_str(),DST.size());
  byte byteDST[6];
  byte byteSRC[6];
  char_array_to_byte_array(tempSRC, byteSRC, 6);
  char_array_to_byte_array(tempDST, byteDST, 6);

  int len = 4;
  enet->connect(byteDST);
  byte command_pkt [6];
  command_pkt[0] = 64;
  command_pkt[1] = 0;
  enet-> sendData(command_pkt, 2, (unsigned short)0xDEAD, byteSRC);
  byte data[4];

  enet->requestData(len, data, byteSRC);

  char * checksum = new char[8];
  sprintf(checksum, "%.2x%.2x%.2x%.2x", data[0],data[1],data[2],data[3]);
  return (std::string)checksum;

}

//sends a packet to the fpga to set the destination of packets sent from the fpga
//byteDST is destination of packet sent from the host
//byte_dst_mac_address_of_FPGA is the destination being set on the fpga
//takes a byte array and sends that destination_address to the fpga
void PacketCreator::set_dest_mac_address (byte dst_mac_address_of_FPGA[], int len, unsigned short protocol, byte byteSRC[], byte byteDST[])
{
  enet->connect(byteDST);
  byte command_pkt [len + 2];
  command_pkt[0] = 0;
  command_pkt[1] = 4;

  for (int i = 2; i < len + 2; i++){
    command_pkt[i] = dst_mac_address_of_FPGA[i-2];
  }
  enet -> sendData(command_pkt, len + 2, protocol, byteSRC);


}

void PacketCreator::set_dest_mac_address (std::string SRC, std::string DST, std::string newDST)
{
  char *tempSRC=new char[SRC.size()+1];
  tempSRC[SRC.size()]=0;
  memcpy(tempSRC,SRC.c_str(),SRC.size());
  char *tempDST=new char[DST.size()+1];
  tempDST[DST.size()]=0;
  memcpy(tempDST,DST.c_str(),DST.size());
  char *tempnewDST=new char[newDST.size()+1];
  tempnewDST[newDST.size()]=0;
  memcpy(tempnewDST,newDST.c_str(),newDST.size());
  byte byteDST[6];
  byte byteSRC[6];
  byte bytenewDST[6];
  char_array_to_byte_array(tempnewDST, bytenewDST, 6);
  char_array_to_byte_array(tempSRC, byteSRC, 6);
  char_array_to_byte_array(tempDST, byteDST, 6);
  unsigned short protocol = (unsigned short)0xDEAD;

  set_dest_mac_address (bytenewDST, 6, protocol, byteSRC, byteDST);

}

//parses char array on colon and takes first n values and assigns them
   to the first n bytes of a byte array.
void PacketCreator::char_array_to_byte_array (char* array1, byte* data,
    int n){
    char* pch;
    int index = 0;
    pch = strtok (array1,":");
    while (pch != NULL && index < n)
    {
    int val = 0;
    sscanf(pch, "%x", &val);
    //printf ("pch = %s -> %u\n",pch,val);
    data[index] = (unsigned char)val;
    //printf("checksum = %u\n",data[index]);
    index++;
    pch = strtok (NULL, ":");
    }
  }

#endif
```

Appendix G

RawEthernet.h

Listing G.1: /gnuradio/gnuradio-core/src/lib/runtime/RawEthernet.h

```
// RawEthernet.h

#ifndef RAWETHERNET_H
#define RAWETHERNET_H

#include <sys/socket.h>
#include <netpacket/packet.h>
#include <net/ethernet.h>

#include <netpacket/packet.h>
#include <net/ethernet.h>
#include <net/if.h>
#include <arpa/inet.h>
#include <cstdio>
#include <iostream>
#include <cstring>
#include <errno.h>

#include <sys/types.h>
#include <sys/socket.h>

typedef unsigned char byte;

class RawEthernet
{
private:
  byte buffer[1514];
  int socketid;
  struct sockaddr_ll socket_address;

public:
  RawEthernet (char *ETH);
  ~RawEthernet ();
  bool connect(byte addr[]);
  bool connect(struct sockaddr_ll *socket_address);
  void disconnect(void);
  bool sendData(byte data[], int len, unsigned short protocol, byte
     src_mac[]);
  bool requestData(int len, byte* data, byte* byteSRC);
  unsigned short protocol;
  char *ETH;

};

using namespace std;

RawEthernet::RawEthernet(char *ETH)
{
  this->protocol = 0xDEAD;
  this->ETH = ETH;
}

RawEthernet::~RawEthernet()
{

}

bool RawEthernet::connect(byte addr[])
{

  struct sockaddr_ll socket_address;
  socket_address.sll_family = AF_PACKET;
  socket_address.sll_protocol = 0;//htons(ETH_P_IP);
  socket_address.sll_ifindex = if_nametoindex(this->ETH); //set to
     ethernet connection being used
  socket_address.sll_hatype = 0;//ARPHRD_ETHER;
  socket_address.sll_pkttype = 0;//PACKET_OTHERHOST;
  socket_address.sll_halen = ETH_ALEN;

  /*MAC - begin*/
  socket_address.sll_addr[0]  = addr[0];
  socket_address.sll_addr[1]  = addr[1];
  socket_address.sll_addr[2]  = addr[2];
  socket_address.sll_addr[3]  = addr[3];
  socket_address.sll_addr[4]  = addr[4];
  socket_address.sll_addr[5]  = addr[5];
  /*MAC - end*/
  socket_address.sll_addr[6]  = 0x00;/*not used*/
  socket_address.sll_addr[7]  = 0x00;/*not used*/

  return this->connect(&socket_address);
}

bool RawEthernet::connect(struct sockaddr_ll *socket_address)
{
  this->socketid = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

  if (this->socketid < 0)
  {
    cout << "Socket Error!" << endl;
    perror("socket");
    return false;
  }
  memcpy((void*)&(this->socket_address), socket_address, sizeof(struct
     sockaddr_ll));
  if (bind(this->socketid, (struct sockaddr*)socket_address, sizeof(
     struct sockaddr_ll))<0)
  {
    cout << "Binding Error!" << endl;
    perror("bind");
    return false;
  }
  //cout << "raw ethernet connected";
  return true;
}

void RawEthernet::disconnect()
{

}

bool RawEthernet::sendData(byte data[], int len, unsigned short
    protocol, byte src_mac[])
{

  memcpy((void*)this->buffer, (void*)(this->socket_address.sll_addr),
     ETH_ALEN);


  memcpy((void*)(this->buffer+ETH_ALEN), (void*)(src_mac), ETH_ALEN);

    *((short*)&(this->buffer[ETH_ALEN+ETH_ALEN])) = htons(protocol);

    memcpy((void*)(this->buffer+ETH_HLEN), (void*)data, len);

  if (len < ETH_DATA_LEN)
  {
    memset((void*)(this->buffer+ETH_HLEN+len), 0, ETH_DATA_LEN-len);
  }

  int send_result = sendto(this->socketid, this->buffer, ETH_FRAME_LEN,
      0,
              (struct sockaddr*)&(this->socket_address), sizeof(struct
                   sockaddr_ll));

  //cout << "errno is " << errno;
  //cout << "send_result is: ";
  //cout << send_result << "\t";
  if (send_result == -1)
  {
    cout << "Send Errer!" << endl;
    perror("sendto");
    return false;
  }
  return true;
}

bool RawEthernet::requestData(int len, byte* data, byte* byteSRC)
{

  struct timeval stTimeOut;
    // Timeout of one second
  stTimeOut.tv_sec = 3;
  stTimeOut.tv_usec = 0;

  fd_set stReadFDS;

  FD_ZERO(&stReadFDS);//initializes set of read sockets
  FD_SET(this->socketid, &stReadFDS);//adds socket to set of read
      sockets


```

```

   int t = select(sizeof(stReadFDS)*8, &stReadFDS, 0, 0, &stTimeOut);// checks set of read sockets for data until timeout
   if (t == -1) {
        fprintf(stderr, "Call to select() failed");
        return false;
   }
   else if (t == 0) {
        printf("Timeout occurred\n");
        return false;

   }
   else {
      int length = recvfrom(this->socketid, this->buffer, ETH_FRAME_LEN, 0, NULL, NULL);
      if (length == -1)
      {
        cout << "Receive Error!" << endl;
        return false;
      }

        if (ntohs(*((short*)(&this->buffer[12]))) !=  this->protocol) return false;
        if (memcmp (this-> buffer, byteSRC, 6) != 0) return false;
        //cout << "passed protocol check\n";
        memcpy((void*)data, (void*)(this->buffer+14), len);
        return true;
   }


}

#endif
```