A Zynq-based Cluster Cognitive Radio

Kurtis Michael Rooks

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

June 17, 2014
Blacksburg, VA

Keywords: Cognitive Radio, Zynq, Cluster

A Zynq-based Cluster Cognitive Radio

Kurtis Michael Rooks

(ABSTRACT)

Traditional hardware radios provide very rigid solutions to radio problems. Intelligent software defined radios, also known as cognitive radios, provide flexibility and agility compared to hardware radio systems. Cognitive radios are well suited for radio applications in a changing radio frequency environment, such as dynamic spectrum access. In this thesis, a cognitive radio is demonstrated where the system self reconfigures to demodulate a detected waveform. The GNU Radio framework is used to provide basic software defined radio building blocks and is supplemented with FPGA accelerators. The use of GNU Radio compliant hardware interfaces allows for seamless hardware/software radio deployments. Dynamic resource mapping allows radio designers to operate at a layer of abstraction above the physical radio implementation. By establishing lower level abstraction layers, future researchers can focus on larger picture concepts such as learning algorithms and behavioral models for the cognitive engine.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increase in processing power of general purpose processors has made it possible for radios to be implemented largely in software. A traditional radio system typically consists of a radio frequency front end such as an antenna and radio specific hardware. In this manner, a traditional radio can only transmit and receive a specific type of signal. For example, a FM radio cannot receive and transmit Bluetooth. Software defined radios still require a radio frequency front end; however, the remaining processing is performed in software. With this model, different radios can be implemented on the same platform by simply running different programs. A software defined radio platform with the appropriate radio frequency front end could conceivably transmit and receive both FM and Bluetooth. Additionally, as new radios are invented, software can be written to enable a software defined radio platform to transmit and receive the new radio data.

Cognitive radio expands upon software defined radio by adding a layer of intelligent command and control software. Cognitive radios have the ability to change radios and radio parameters based on the current environment. A cognitive radio that has the same radio capabilities of a modern smartphone with WiFi and 4G radios could be programmed to conserve 4G data by using WiFi when available. To achieve this action, the cognitive radio must first have the ability to sense the radio frequency spectrum and determine whether or not WiFi is available. Secondly, the cognitive radio must have the ability to use the spectrum information it gathered to reconfigure itself for either 4G or WiFi as appropriate.

The flexibility of software defined radios comes at the cost of power efficiency. Hardware radios are specifically designed to transmit and receive set radios. As such, application specific hardware radios typically consume less power than a solution on a general purpose platform. For mobile applications such as phones and tablets where the main power supply is a battery, power consumption and efficiency are a concern. A more power efficient general purpose processor, such as the ARM processor, fills the gap between application specific hardware and high performance desktop processors. The continuum of efficiency spans between application specific hardware and general purpose processors. Field programmable gate arrays (FPGAs) fill the gap between application specific hardware and power efficient general purpose processors.

In recent years, Xilinx has released the Zynq processor which contains a dual core ARM processor as well as a field programmable gate array. This unique combination provides an interesting platform for implementing software defined radios with hardware (FPGA) and general-purpose processor (ARM) components. The ARM processor is well supported and can be configured to run a Linux-based operating system. By running Linux, existing software defined radio frameworks such as GNU Radio can be used to significantly reduce the amount of effort required to create software defined radio applications.

Motivation

Low-power software defined radio platforms suffer from performance limitations that make it difficult to realize real time radio systems. The aim of this project is to enhance the capabilities of the GNU Radio software defined radio framework through tight integration with FPGA accelerators. In this thesis, a cluster of Zynq processors is explored to provide a platform for mixed hardware and software cognitive radio deployments. Fast and efficient deployment of hardware/software radios is achieved by creating standard interfaces to hardware accelerators that seamlessly integrate with GNU Radio software components.

Contributions

This thesis establishes a platform for future cognitive radio system research. Future researchers can leverage this groundwork to construct a more advanced cognitive radio system. Future efforts can be conducted at a higher level of abstraction, building upon the work presented in this thesis. Specific contributions include:

- Developed a GNU Radio compatible interface for reliable board to board communication. When used in combination with dynamic resource allocation a layer of abstraction is established that allows users to design hardware agnostic radios.
- Created a set of input and output functions that establish an application programming interface (API) for future cognitive radio applications. Creating a API significantly reduces design effort for future researchers, allowing focus to be focused on higher level concepts, such as cognitive algorithms.

- Implemented GReasy compatible Fast Fourier Transform accelerator. Contributing to common library of parts creates a set of reusable components available to other designers. Standard libraries enable users to leverage proven designs and greatly reduces implementation and testing efforts.
- Demonstrated Tflow as an enabling technology for cognitive radio system. Previous cognitive radio systems utilize pre-generated partial bit files as an adaptive technology. Tflow provides significantly more flexibility than partial bit files and allows for dynamically generated radios to be realized in autonomous systems.

Document Organization

The remaining chapters are organized as follows. Chapter 2 will provide background information on software defined radio and cognitive radio. Two software packages, GNU Radio and Tflow will also be discussed. Finally, related works will be explored and analyzed. Chapter 3 will present the hardware components used in a first generation cognitive radio platform. An analysis of the shortcomings of this first generation platform will be discussed as motivating factors in selection of the second generation hardware. A detailed overview of the resulting system architecture will then be presented. Chapter 4 will present the software components of the cognitive radio system beginning at energy detection and ending with system reconfiguration. Chapter 5 describes a demonstration of the cognitive radio system. In the demonstration the cognitive radio detects two transmitted waveforms then self reconfigures to deploy an associated radio receiver. Chapter 6 will present conclusions and opportunities for future work.

Chapter 2
Background

In this chapter, software defined radio and cognitive radio will be further explored. First, formal definitions will be given, as well as a sample application of cognitive radio. Next, an overview of two software packages, GNU Radio and TFlow, will be presented. These two software packages enable software defined radio and cognitive radio respectively. Finally, the state of the art in software defined radio and cognitive radio will be explored.

Software Defined Radio

In 1992, Dr. Joe Mitola published the first paper on the subject of software defined radio (SDR). His paper [1], outlines the fundamental components of a software defined radio platform. In addition to general purpose processors and digital signal processing (DSP) chips, a software defined radio also consists of an analog to digital converter (ADC) and a digital to analog converter (DAC). The ADC and DAC convert data to/from the analog signals into discrete digital signals where they can be manipulated with software. Traditional radio frequency (RF) equipment, such as antennas, are still required for data transmission and reception.

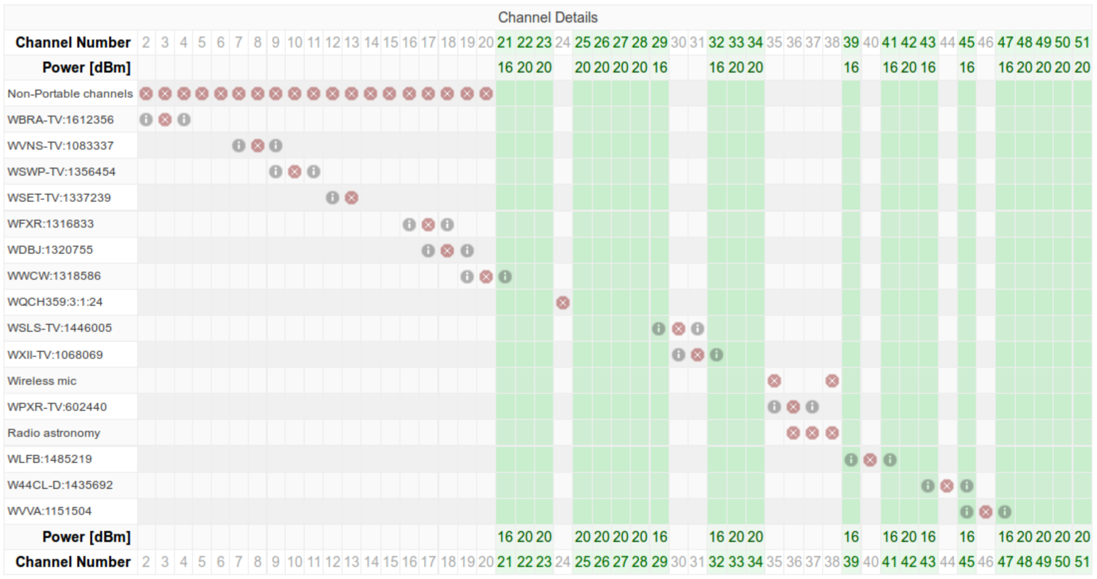
By performing the majority of signal processing functions in software, a software defined radio platform can implement many different waveforms on one set of hardware. As new waveforms are developed, software can be written for the existing software defined radio platform, without changing any of the underlying hardware. By implementing radios in software, SDRs prove to be extremely flexible and adaptable. In contrast, deploying a new waveform on a strictly hardware radio would likely require adding new hardware or modifying the existing hardware, which is a much more time consuming process.

Cognitive Radio

A cognitive radio is a software defined radio that has the ability to change its configuration based on information about its environment. According to [2], a cognitive radio has three key aspects: observation, reconfiguration and cognition. Observation refers to the radio's ability to gather information about its environment, reconfiguration refers to its ability to change, and cognition refers to its ability to make changes based on the data collected about the environment.

One proposed application of cognitive radios has been in dynamic spectrum access. Worldwide, spectrum has been allocated by government regulators such as the Federal Communication Commission (FCC) in the United States and the International Telecommunications Union - Radiocommunication sector (ITU-R). Licenses are issued for use of spectrum by frequency and location and are often held for long periods of time. It has been reported [3] that spectral usage varies from between 15% and 85%. Such gaps in spectral usage exists both temporally and spatially. Increasing demand for spectrum access has led to discussions about secondary users utilizing licensed spectrum when the primary user is not transmitting. The FCC has certified

Google's Spectrum Database [4] to index such "whitespaces" based on location. Figure 1 shows the unused television frequencies for Blacksburg, Virginia in 2014.



Channel Details

Channel Number	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
Power [dBm]																				16	20	20		20	20
Non-Portable channels	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗	⊗						
WBRA-TV:1612356	ⓘ	⊗	ⓘ																						
WVNS-TV:1083337						ⓘ	⊗	ⓘ																	
WSWP-TV:1356454								ⓘ	⊗	ⓘ															
WSET-TV:1337239											ⓘ	⊗													
WFXR:1316833															ⓘ	⊗	ⓘ								
WDBJ:1320755																ⓘ	⊗	ⓘ							
WWCW:1318586																		ⓘ	⊗	ⓘ					
WQCH359:3:1:24																							⊗		
WSLS-TV:1446005																									
WXII-TV:1068069																									
Wireless mic																									
WPXR-TV:602440																									
Radio astronomy																									
WLFB:1485219																									
W44CL-D:1435692																									
WVVA:1151504																									
Power [dBm]																				16	20	20		20	20
Channel Number	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

Channel Number	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51
Power [dBm]	20	20	16			16	20	20					16		16	20	16		16		16	20	20	20	20
Non-Portable channels																									
WBRA-TV:1612356																									
WVNS-TV:1083337																									
WSWP-TV:1356454																									
WSET-TV:1337239																									
WFXR:1316833																									
WDBJ:1320755																									
WWCW:1318586																									
WQCH359:3:1:24																									
WSLS-TV:1446005			ⓘ	⊗	ⓘ																				
WXII-TV:1068069				ⓘ	⊗	ⓘ																			
Wireless mic									⊗			⊗													
WPXR-TV:602440									ⓘ	⊗	ⓘ														
Radio astronomy										⊗	⊗	⊗													
WLFB:1485219													ⓘ	⊗	ⓘ										
W44CL-D:1435692																	ⓘ	⊗	ⓘ						
WVVA:1151504																			ⓘ	⊗	ⓘ				
Power [dBm]	20	20	16			16	20	20					16		16	20	16		16		16	20	20	20	20
Channel Number	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51

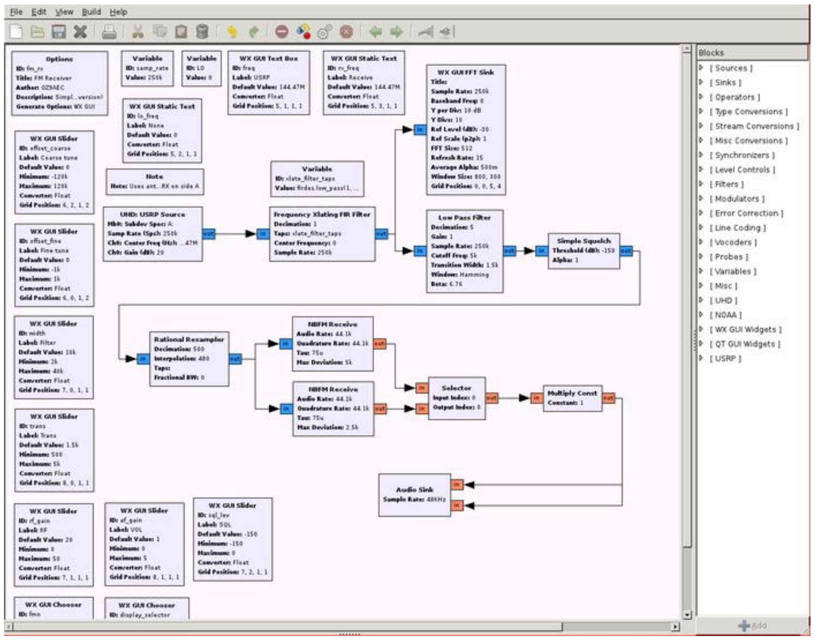
Figure 1 - Television "whitespace" for Blacksburg, Virginia Google, "Google Spectrum Database." [Online]. Available: https://www.google.com/get/spectrumdatabase/channel/. Used under fair use, 2014.

Cognitive radios would allow users to opportunistically access this spectrum by first sensing whether or not the primary user is transmitting. Then, if it is determined that there is spectrum that is currently unused by a primary user, the radio could be reconfigured to use that spectral gap. However, this configuration is only legally ethical while a primary user is not present since the primary user is licensed to operate in specified frequency band. If the secondary user travels to a new location and detects a primary user, the radio should stop using the spectrum. Cognitive radio could also be used to select the best unused spectrum and reconfigure when a better channel becomes available.

GNU Radio

GNU Radio is an open-source software toolkit for implementing software defined radios. Users are provided with a library of functions, often referred to as “blocks”. These blocks range from signal processing to data conversion, and are designed to enable users to quickly create software defined radio applications without re-inventing fundamental components. Since GNU radio is an open source software project, it can be compiled and run on many different platforms (such as ARM). GNU Radio has a large user community, with many sample radio implementations. Additional building blocks can be added to GNU Radio to increase its capabilities.

GNU Radio is written in two programming languages, C++ and Python. Performance critical blocks, such as signal processing blocks are written in C++. Less computationally intensive software, such as the overall system framework is written in Python. GNU Radio comes with an interactive graphical user interface (GUI), called GNU Radio companion (GRC), similar to National Instrument's LabVIEW [4]. Users can graphically place and connect blocks without writing a single line of code. Figure 2 shows an example of a GRC flow-graph.



```
Options
ID: fm_rx
Title: FM Receiver
Author: OZ9AEC
Description: Simpl...version)
Generate Options: WX GUI

Variable
ID: samp_rate
Value: 250k

Variable
ID: LO
Value: 0

WX GUI Text Box
ID: freq
Label: USRP
Default Value: 144.47M
Converter: Float
Grid Position: 5, 1, 1, 1

WX GUI Static Text
ID: rx_freq
Label: Receive
Default Value: 144.47M
Converter: Float
Grid Position: 5, 3, 1, 1

WX GUI Static Text
ID: lo_freq
Label: None
Default Value: 0
Converter: Float
Grid Position: 5, 2, 1, 1

WX GUI FFT Sink
Title:
Sample Rate: 250k
Baseband Freq: 0
Y per Div: 10 dB
Y Divs: 10
Ref Level (dB): -30
Ref Scale (p2p): 1
FFT Size: 512
Refresh Rate: 15
Average Alpha: 500m
Window Size: 800, 300
Grid Position: 0, 0, 5, 4

WX GUI Slider
ID: offset_coarse
Label: Coarse tune
Default Value: 0
Minimum: -120k
Maximum: 120k
Converter: Float
Grid Position: 6, 2, 1, 2

Note
Note: Uses ant...RX on side A

Variable
ID: xlate_filter_taps
Value: firdes.low_pass(1, ...

UHD: USRP Source
Mb0: Subdev Spec: A:
Samp Rate (Sps): 250k
Ch0: Center Freq (Hz): ...47M
Ch0: Gain (dB): 20

Frequency Xlating FIR Filter
Decimation: 1
Taps: xlate_filter_taps
Center Frequency: 0
Sample Rate: 250k

Low Pass Filter
Decimation: 5
Gain: 1
Sample Rate: 250k
Cutoff Freq: 5k
Transition Width: 1.5k
Window: Hamming
Beta: 6.76

Simple Squelch
Threshold (dB): -150
Alpha: 1

WX GUI Slider
ID: offset_fine
Label: Fine tune
Default Value: 0
Minimum: -1k
Maximum: 1k
Converter: Float
Grid Position: 6, 0, 1, 2

WX GUI Slider
ID: width
Label: Filter
Default Value: 10k
Minimum: 2k
Maximum: 40k
Converter: Float
Grid Position: 7, 0, 1, 1

Rational Resampler
Decimation: 500
Interpolation: 480
Taps:
Fractional BW: 0

NBFM Receive
Audio Rate: 44.1k
Quadrature Rate: 44.1k
Tau: 75u
Max Deviation: 5k

NBFM Receive
Audio Rate: 44.1k
Quadrature Rate: 44.1k
Tau: 75u
Max Deviation: 2.5k

Selector
Input Index: 0
Output Index: 0

Multiply Const
Constant: 1

WX GUI Slider
ID: trans
Label: Trans
Default Value: 1.5k
Minimum: 500
Maximum: 5k
Converter: Float
Grid Position: 8, 0, 1, 1

Audio Sink
Sample Rate: 48KHz

WX GUI Slider
ID: rf_gain
Label: RF
Default Value: 20
Minimum: 0
Maximum: 50
Converter: Float
Grid Position: 7, 1, 1, 1

WX GUI Slider
ID: af_gain
Label: VOL
Default Value: 1
Minimum: 0
Maximum: 5
Converter: Float
Grid Position: 8, 1, 1, 1

WX GUI Slider
ID: sql_lev
Label: SQL
Default Value: -150
Minimum: -150
Maximum: 0
Converter: Float
Grid Position: 7, 2, 1, 1

WX GUI Chooser
ID: fmn

WX GUI Chooser
ID: display_selector
```

Figure 2 – Example GNU Radio companion flowgraph GNU Radio, “screenshot-grc-nbfm” [Online]. Available: http://gnuradio.org/redmine/attachments/266/screenshot-grc-nbfm.png. Used under fair use, 2014.

GNU Radio operates on a one thread per block model, where each block in a flowgraph is run as a separate thread. Every pair of connected blocks has a buffer in between them, which creates backpressure in the system. Before a GNU Radio block is scheduled to run, a check is performed to determine whether or not there are sufficient samples in the input buffer and if there is sufficient space in the output buffer to hold results. If either condition is not met, the GNU Radio scheduler moves on to the next block in the chain and performs the same checks. GNU Radio natively supports the data types shown in Table 1. GNU Radio uses two single precision floating point numbers to represent the real and imaginary portions of complex data.

Data Type	# Bits
Float	32
Complex (Float real, Float imaginary)	64
Int	32
Short	16
Byte	8

Table 1 - GNU Radio data types

Tflow

Generating configuration files, also known as bit files, for FPGAs is often a slow process. Complicated designs can take hours to complete the process of synthesis, placement, routing and bit file generation. The Virginia Tech developed software package, Tflow [5], attempts to reduce the amount of time required to generate FPGA configuration files. Tflow splits FPGA designs into two distinct regions, a non-changing "static" region and a dynamic "blacktop" region. The static region is comprised of portions of a design that do not change between rebuilds. The blocktop region consists of portions of a design that change between builds.

Functional components utilized in the blacktop are pre-generated and put into a library of parts. These library components are also known as "macro" blocks. Macro blocks can be combined to create more complicated designs in the blacktop. Tflow achieves speedups in bit file generation by skipping the placement and routing for both the static region and all of the macro blocks.

Tflow only needs to perform placement of the macro blocks, the routing between blocks and bit file generation.

Tflow greatly reduces the amount of time spent generating new FPGA configuration files and thereby makes it possible for hardware accelerators to be efficiently used in conjunction with software components. Tflow enables another Virginia Tech software project, GReasy [6], to deploy hardware/software radios quickly. This makes hardware/software prototyping much faster, enabling users to concentrate on radio designs instead of hardware implementations. As of 2014, Tflow supports Xilinx Virtex 4, Xilinx Virtex 5, and Xilinx 7 Series FPGAs.

IRIS on Zynq

A group from Trinity College in Dublin, Ireland have created a Xilinx Zynq based software defined radio platform based on the SDR framework, Implementing Radio in Software (IRIS) [7]. IRIS, like GNU Radio, is a framework for constructing software defined radios. In this framework, software components are connected together to form complete radios. The Universal Software Radio Peripheral (USRP), a well established product by Ettus Research, is used as a RF front end to the system. The Zynq board runs a Xilinx Linux distribution for an operating system. FPGA accelerators are used to reduce bottlenecks in some of the more computationally intensive IRIS components. Using two complete systems (USRP + Zynq), the Trinity researchers were able to achieve a 800 kbps orthogonal frequency division-multiplexing (OFDM) video stream end to end.

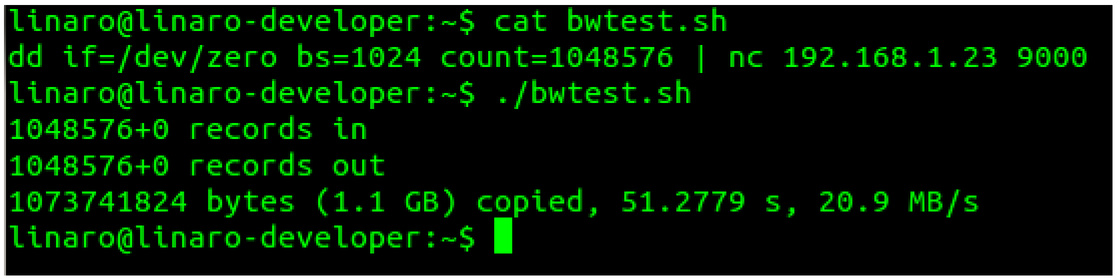
The OFDM radio implementation demonstrates the Zynq processor’s ability to implement software defined radios. The selection of mature products such as IRIS and the USRP provides a stable platform for software defined radio development. IRIS was not an open-source project until 2013, and as such has a smaller user community than GNU Radio. A large user community means there are more example applications to leverage and more support is available when developing new applications. A larger user group also means that software bugs are more likely to be found. Both GNU Radio and IRIS use XML to represent radio designs. However, GNU Radio provides a convenient, user-friendly GUI to generate the XML for radio designs.

Zcluster

Researchers from the University of Toronto have constructed Zcluster [8], a cluster of Zynq processors. The cluster uses the open source Apache Hadoop framework to enable distributed processing across nodes in the cluster. The cluster consists of eight Zedboards, which feature a Zynq XC7Z020 processor. Each board runs Xillinux [9] as its operating system, a Linux distribution based on Ubuntu 12.04 for the ARM processor. Nodes communicate with each other through a 100 Mbps switch.

An x86-based host system is also present on the network to function as the “NameNode” for the Hadoop Distributed File System (HDFS). This “NameNode” was deemed too memory and computationally intensive to be run on one of the ARM processors. Xillybus, the Xilinx intellectual property (IP) core, was used to communicate between accelerators in the FPGA and the ARM cores through the AXI bus. A Finite Impulse Response (FIR) filter was implemented both as a FPGA accelerator (running at 100 MHz) and as an optimized C program. The FPGA implementation was measured to execute 2.4 times faster than the software implementation.

Zcluster's use of Hadoop is appropriate for inherently parallel tasks, where large datasets can be broken into smaller datasets. However, software defined radios have many sequential dependencies. Data must be processed in a specific order to be correctly demodulated. In a real time radio system, data is processed as it arrives and must be processed in real time to avoid data loss. This type of data would not benefit from the parallel framework provided by Hadoop. Additionally, transferring large amounts of radio data between boards via ethernet consumes a large percentage of the ARMs processing which could be utilized for data processing. The network speed test shown in Figure 5 required 100% of an ARM core to transfer data at 20.9 MBps from a Zedboard to a x86 host via gigabit Ethernet. Running two such speed tests in parallel maxes out around 31 MBps with both ARM cores at 100% utilization.



```
linaro@linaro-developer:~$ cat bwtest.sh
dd if=/dev/zero bs=1024 count=1048576 | nc 192.168.1.23 9000
linaro@linaro-developer:~$ ./bwtest.sh
1048576+0 records in
1048576+0 records out
1073741824 bytes (1.1 GB) copied, 51.2779 s, 20.9 MB/s
linaro@linaro-developer:~$
```

Figure 3 - Zedboard network speed test

Chapter 3
Hardware Components

This chapter will cover the hardware components of the cognitive radio system. The first iteration of this system will be reviewed and analyzed. Conclusions from the first generation system will be drawn upon to discuss component selection for the second generation system featured in this thesis. The use of a cluster topology is discussed and a detailed review of inter-node communication is presented. Finally, the use of hardware accelerators is explained.

First Generation

The objective of the encompassing project has been to construct a highly agile hardware accelerated cognitive radio, based upon the GNU Radio software model, and designed for low-power autonomous operation. The first generation platform of this project was centered on a Tilera TilePro64 processor [10]. The selection of the Tilera TilePro64 as a SDR platform was largely driven by the number of concurrent software threads the Tilera could support. As previously mentioned, GNU Radio operates on a thread per block model; every component in a radio flowgraph is assigned to a unique thread. The TilePro64 features 64 homogenous general purpose cores running at 866 MHz. In this manner, it was theorized that each core in the Tilera would handle a thread and the computational load of the desired software defined radio would be evenly distributed across all 64 cores.

The TilePro64 features four memory controllers, connected to eight gigabytes of DDR2 memory total. Memory can be configured to optionally “stripe” across the four memory controllers, such that each 8 KB of memory is a controller by a different memory controller. This memory striping effectively load balances memory access between all of the memory controller. The TilePro64 also features two 10 gigabit Ethernet controllers, two gigabit Ethernet controllers and two 4x PCIe controllers.

The processors are connected in an 8x8 mesh grid as seen in Figure 4. Each processor has five separate 32-bit wide buses to each neighboring core: the I/O dynamic network (IDN), the Memory dynamic network (MDN), the Coherence dynamic network (CDN), the User dynamic network (UDN) and the Tile dynamic network (TDN). The IDN provides a two way connection between I/O devices and memory. The MDN is for reads/writes to memory and other core’s cache. The CDN is used to maintain cache coherency between cores. The UDN provides users with low level first in first out (FIFO) interfaces for low latency data transfers between cores. The TDN, similar to the MDN, supports cache reads and writes.

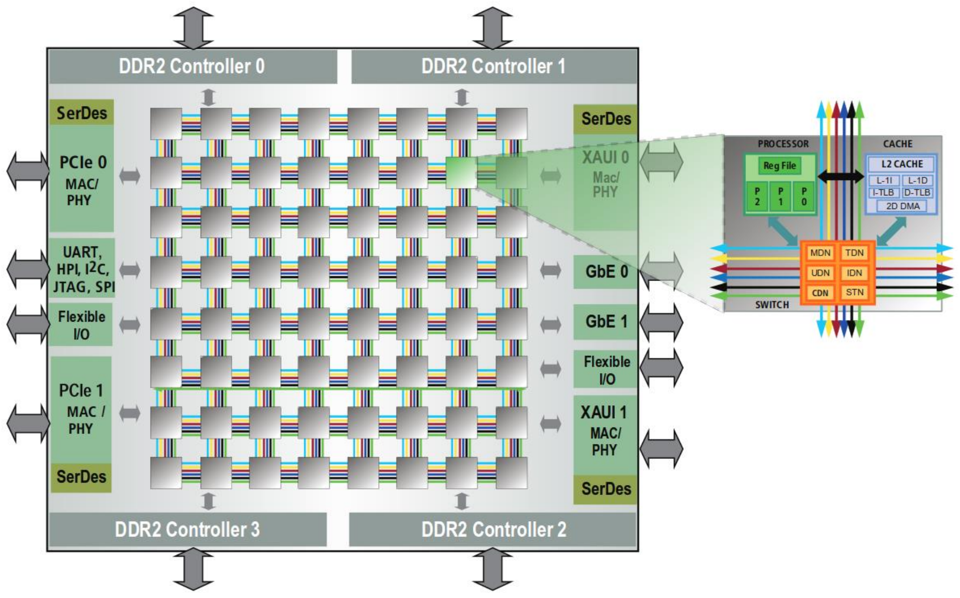


Figure 4 – TilePro64 core layout Tilera Corporation (San Jose), "TILE Pro 64 Processor – Product Brief," 2011. [Online]. Available: http://www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_PB019_v4.pdf. Used under fair use, 2014.

The Tilera runs a custom version of symmetric multiprocessing (SMP) Linux. All applications not provided with the Tilera were cross compiled using the included Tilera specific GNU Compiler Collection (GCC). In this manner, GNU Radio and all its dependencies were also cross compiled for this platform. A host machine acted as a Peripheral Component Interconnect Express (PCIe) bus master. The Tilera communicated with a Virtex 6 FPGA via the PCIe bus.

Several shortcomings were discovered while developing software for the TilePro64. One of the most significant shortcomings of the Tilera processor was the lack of a floating-point unit (FPU). Many GNU Radio components utilize floating point numbers, and the emulated floating-point operations performed by the Tilera proved to be a significant bottleneck in the overall system performance. Another shortcoming of the Tilera system was the maturity of the C compiler. Upon careful observation of compiled code, it was determined that the compiler was not utilizing all of the Tilera's operation codes (opcodes). In order to achieve the necessary performance for critical portions of code, the use of intrinsic functions was required.

Second Generation

The shortcomings discovered with the Tilera platform heavily influenced the selection of components in the second phase of this project. An ARM processor was selected for its FPU, stable C compiler and general maturity of the architecture. Linux for ARM processors is readily available, enabling GNU Radio to continue to be utilized in the second generation system. Next, a replacement for the Virtex-6 FPGA was considered. In the first generation system, a host machine was necessary to act as a PCIe bus master to enable Tilera to FPGA communication. To reduce the number of extra components in the system, a solution was sought that would allow the ARM to communicate directly with the FPGA.

The PCIe connected between the Tilera and the FPGA only offered a single datapath. Even if Tilera were replaced with a desktop system, all interactions between the GPP and the FPGA would be through a single PCIe connection. This could prove to be a difficult system to design for as a resource sharing system would need to be implemented. For the second generation system, multiple datapaths were desired in order to match the data flow model of GNU Radio flowgraphs. Additionally, the first generation system, the Virtex-6 FPGA image was completely static. For the second generation system, it was desired to have a more flexible FPGA interface. Any change would require generating a new bit file using a traditional Xilinx tools. Transitioning from the Xilinx tool-chain to Tflow allowed for increased flexibility. However, the Virtex-6 is not supported by the Tflow software. A Virtex 5 series or a 7 Series Xilinx FPGA was required to be compatible with Tflow.

The Xilinx Zynq processor met the requirements for an ARM processor, it has a tightly integrated FPGA and is supported by Tflow. The Xilinx Zynq architecture shown in Figure 5 illustrates the unique combination of a dual-core ARM Cortex A9 processor with a Artix-7 Field Programmable Gate Array (FPGA) on the same physical die. The maximum clock frequencies for the processor is 866 MHz for the smaller chips (XC7Z010, XC7Z015, XC7Z020) and 1 GHz for the larger chips (XC7Z030, XC7Z045, XC7Z100). Each of the two ARM processors has a NEON general-purpose single instruction multiple data (SIMD) engine.

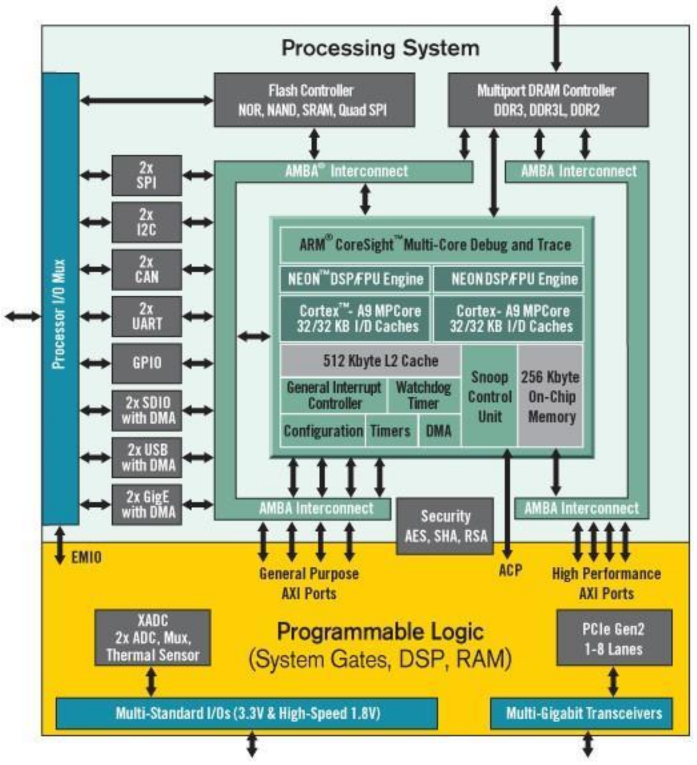

Processing System
Flash Controller NOR, NAND, SRAM, Quad SPI
Multiport DRAM Controller DDR3, DDR3L, DDR2
AMBA® Interconnect
AMBA Interconnect
2x SPI
2x I2C
2x CAN
2x UART
GPIO
2x SDIO with DMA
2x USB with DMA
2x GigE with DMA
Processor I/O Mux
ARM® CoreSight™ Multi-Core Debug and Trace
NEON™ DSP/FPU Engine
NEON DSP/FPU Engine
Cortex™- A9 MPCore 32/32 KB I/D Caches
Cortex- A9 MPCore 32/32 KB I/D Caches
512 Kbyte L2 Cache
General Interrupt Controller
Watchdog Timer
Configuration
Timers
DMA
Snoop Control Unit
256 Kbyte On-Chip Memory
AMBA Interconnect
Security AES, SHA, RSA
AMBA Interconnect
EMIO
General Purpose AXI Ports
ACP
High Performance AXI Ports
XADC 2x ADC, Mux, Thermal Sensor
Programmable Logic (System Gates, DSP, RAM)
PCIe Gen2 1-8 Lanes
Multi-Standard I/Os (3.3V & High-Speed 1.8V)
Multi-Gigabit Transceivers

Figure 5 - Zynq 7000 Architecture Xilinx, Inc (San Jose), “Zynq-7000 All Programmable SoC.” [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/. Used under fair use, 2014.

The ARM processors and the FPGA fabric are connected through an Advanced eXtensible Interface (AXI). The existence of both devices on the same chip allows for tightly integrated hardware and software designs. Larger versions of the Zynq series (XC7Z030, XC7Z045, XC7Z100) feature additional programmable logic from a Kintex-7 FPGA. These larger devices also include multi-gigabit transceivers (MGTs).

The FPGA resources for the Zynq XC7Z020 used in this thesis are shown in Table 2 below. The information in Table 2 was obtained from the Zynq 7000 product table [11].

Resource	**# of Resources**
Look Up Tables	53,200
Flip Flops	106,400
Block RAM (36 Kb)	240
DSP Slices	220

Table 2 - Zynq XC7Z020 FPGA resources

In order to run GNU Radio and TFlow, the Zynq boards were configured to run Linaro Ubuntu [12], an ARM branch of Ubuntu Linux. The boards are running a full Linux operating system (OS), which simplifies the installation of new software through the “apt-get” tool in Ubuntu. Leveraging “apt-get” provides a large productivity boost, because it removes the need to compile (or cross-compile) many required programs and libraries. The “apt-get” version of GNU Radio (v3.6) was ignored in favor of the latest version of GNU Radio (v3.7). As such, GNU Radio was compiled natively on the Zynq boards.

The file system for the Zynq boards can be either a Secure Digital (SD) card or a Network File System (NFS) mount. For this demonstration, a NFS system was used and mounted on an x86 host machine. The SD card solution would be better suited for a mobile application where size and power constraints are more stringent. The NFS solution provides more storage, is much faster, and is well suited for a development environment.

Cluster Architecture

A driving factor in initially selecting the Tilera processor was to have dedicated cores for each GNU Radio thread. The first generation Tilera-based system featured 64 general purpose processors. In contrast, the Zynq platform selected for the second generation system only has two general-purpose processors. In an effort to emulate a larger, ARM based multi-core processor, a cluster architecture was explored. Four Xilinx Zynq-7000 ZC702 development boards provided a cost effective platform to explore such an architecture.

Each node in the cluster is driven by a Xilinx Zynq XC7Z020 system-on-a-chip (SoC). Nodes in the cluster are connected together with a gigabit Ethernet and FPGA Mezzanine Card (FMC) connectors. Gigabit Ethernet was used for configuration and control while the FMC connectors were used for low-latency, high-bandwidth data transfers. For communication between the ARM processors and the FPGA fabric, a direct memory access (DMA) interface was created using the AXI interface.

The Zynq ZC702 boards have two Low Pin Count (LPC) FMC connectors that interface directly to the FPGA fabric on each board. An FMC to Serial ATA (SATA) adapter board was fabricated to interconnect the ZC702 boards. The adapter board is seen in Figure 6. The design is an open hardware project created by Dan Strother [17]. Since the ZC702 version of the Zynq processor lacks multi-gigabit transceivers (MGT), the existing data transfer IP such as the Xilinx Aurora was not available. Instead, a Xilinx SelectIO interface was generated to handle data serialization and deserialization (SerDes). A custom data synchronization scheme was implemented to compensate for time skew and data inversion.

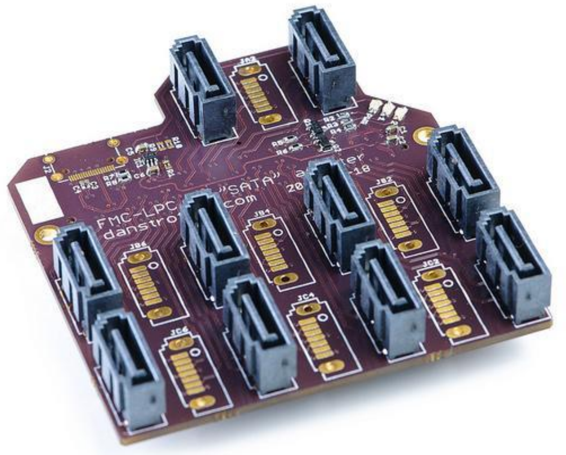


Figure 6 – FMC to SATA adapter board D. Strother, “FMC-LPC to SATA adapter board.” [Online]. Available: http://danstrother.com/2010/12/04/fmc-lpc-to-sata-adapter-board/. Used under fair use, 2014.

The custom synchronization scheme was devised in to address the three factors. First, each individual board in the cluster is reset independently of all other boards, and because of that, they exit reset at different times. Second, it was discovered that the FMC to SATA board (seen in Figure 6) inverts some of the differential data pairs so that there are data inversions on some of the board links, but not on all. The final factor in creating a custom synchronization scheme is system scalability. It is highly undesirable to create board specific versions of the same functional design to compensate for data inversion.

The solution was for each transmitter to send a known sequence of data for a few seconds after reset. Likewise, each receiver comes out of reset and begins comparing the data it received with the expected data. The receiver performs a sequence of data inversions (XOR with a counting sequence) and time-domain bit shifts. The receiver continues cycling through all possible combinations until the expected sequence is received. The receiver saves the correction vector and applies the correction to all future received data. Figure 7 shows the state machine for the link synchronization scheme.

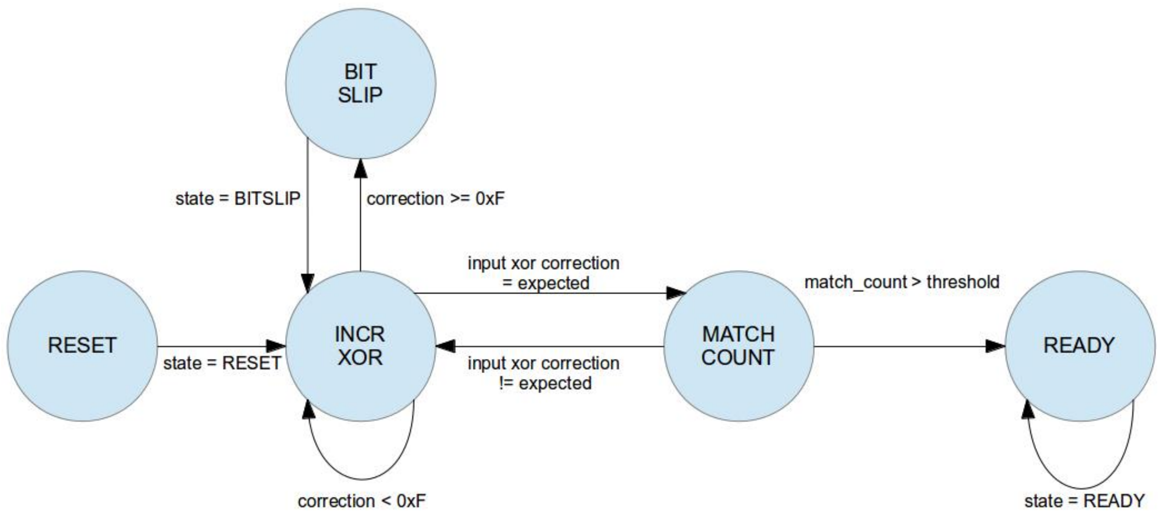


Figure 7 – Link synchronization state machine

The link synchronization state machine above prevents unaligned and inverted data from passing to the rest of the system using a gated buffer (as seen in Figure 8). The link synchronization state machine is labelled as the “Correction Vector” block Figure 8. The gated buffer is disabled until the link has been synchronized and the transmitter stops sending the known data sequence. Data are also sent with an extra bit indicating whether or not the data are valid. This is necessary because the transmitter sends data every cycle regardless of whether or not it has received input.

The known sequence transmitted during link synchronization does not have a valid bit set, so the receiver will not pass this synchronization data to the reset of the system.

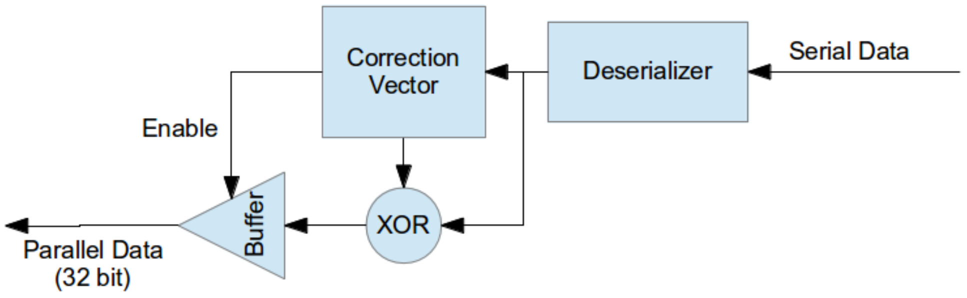


Figure 8 - Link synchronization block diagram

To test the stability of the links, a series data transfer test was conducted. One iteration of the test consisted of reconfiguring all the FPGAs in the cluster, resetting the DMA and FMC subsystems, then writing and reading 2 MB of counter data on every link in the cluster. Each counting sequence was checked by the receiver for data integrity. Several hundred iterations of this test were run automatically overnight with no detected bit flips. For the test described, the serial clock ran at 100 MHz and the parallel clock ran at 10 MHz. A maximum data rate of 640 Mbps was achieved with a serial clock speed of 200 MHz and a parallel clock speed of 20 MHz. When running the serial clock at 300 MHz, bit flips were too frequent to be reliable.

The FMC to SATA board has 17 SATA connectors, each containing two differential pairs. This setup provides a highly customizable interface, allowing the cluster interconnects to be rewired to meet system requirements. Figure 9 illustrates a fully connected configuration between Zynq boards and each board connected to the same data source (shown as Zed1). The cluster can be rewired for any subset of connections (seen in Figure 9). The cluster can also be rewired to create multiple connections between boards.

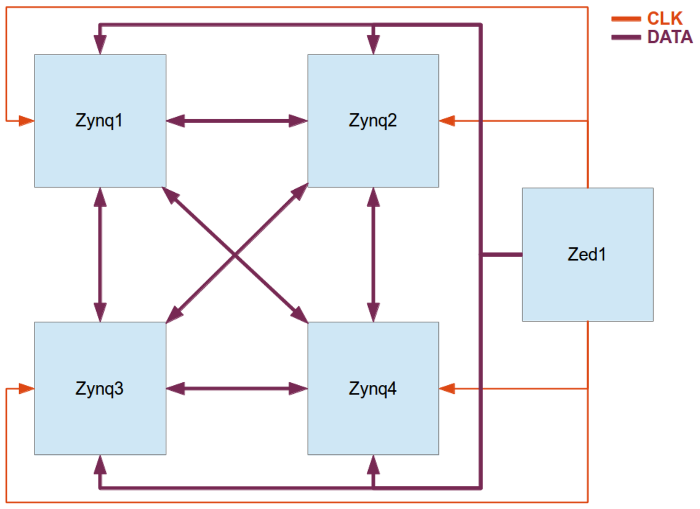


Figure 9 - Zynq board interconnects

For the cognitive radio application described in this paper, a fully connected configuration was selected. Each of the four boards in the cluster has a 32-bit wide connection to and from every other board. Each Zynq board in the cluster receives a clock from the same external clock. Figure 10 illustrates how the fully connected configuration in Figure 9 was realized using the FMC to SATA board.

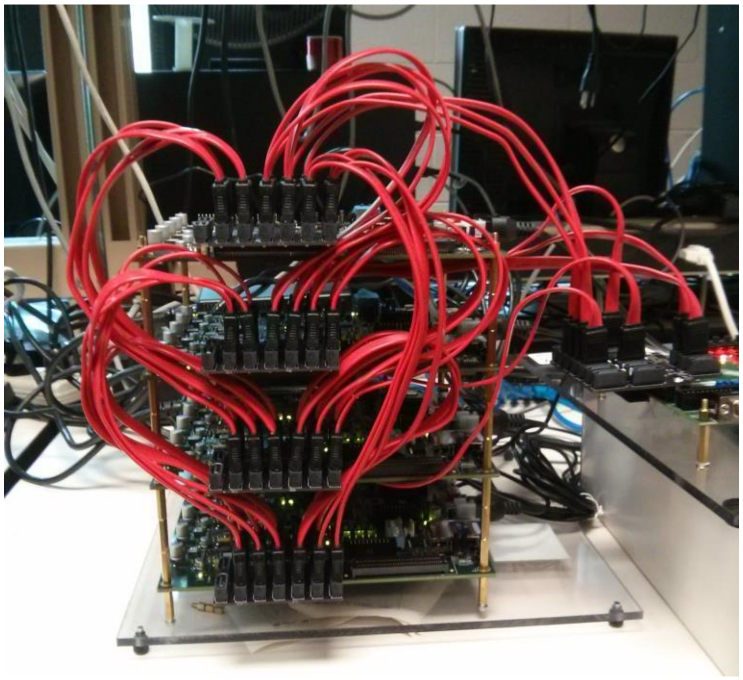


Figure 10 - Zynq cluster connected to data source

Hardware Accelerators

To supplement the ARM’s computation abilities, FPGA-based accelerators were developed. To maintain compatibility with the GNU Radio programming model, each FPGA accelerator is wrapped with a standard interface. The standard interface consists of a 32-bit input with a valid bit and a 32-bit output with a valid bit. For complex data, the upper 16 bits represent imaginary samples and the lower 16 bits represent real samples. This interface was selected in order to be compliant with the GReasy project. Any hardware accelerator developed for this project can be used in GReasy and vice versa. Two such accelerators featured later in this thesis are a fast Fourier transform (FFT) and a finite impulse response (FIR) filter.

Chapter 4
Cognitive Radio

This chapter covers the software components of the cognitive radio system developed in this thesis. The cognitive radio consists of seven parts, as seen in Figure 11. With the exception of an FPGA based FFT, all components are implemented as software on the ARM cores of the Zynq processors. The signal source seen in Figure 11 is not necessarily a component of the cognitive radio, rather the source of data.

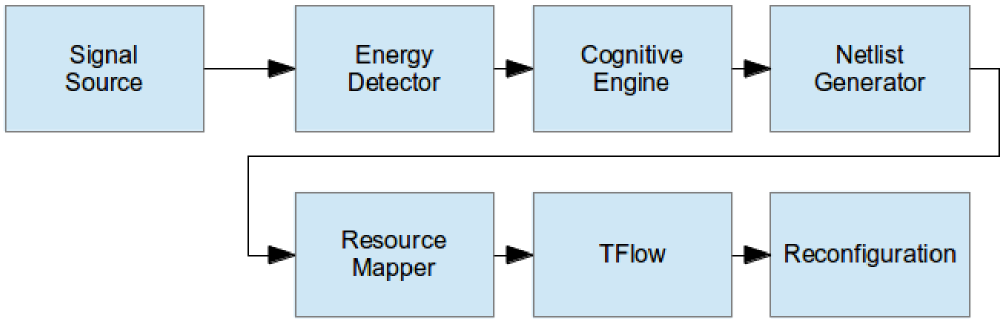


Figure 11 - Software Flow Diagram

Signal Source

A single Zedboard containing the same Zynq XC7Z020 as the rest of the cluster was used to provide data to the rest of the system. GNU Radio was utilized to generate signals that were sent to each individual board via their individual FMC connectors. Using GNU Radio allows easy creation on different input waveforms, as well as the ability to shift each waveform to the desired transmission frequency. Waveforms can either be generated offline and replayed from a GNU Radio file source or generated at runtime. Playing pre-generated data proves useful when debugging radio implementations, as the transmitted data is known and consistent. In an earlier demonstration, a Zynq XC7Z045 was used in combination with an FMC-based ADC card to provide off-the-air, real time data. Alternatively, the Zedboard could be configured to interface to a USRP. The variety of data source options illustrates the advantage of having a well-supported GPP in the system.

Energy Detector

The goal of the energy detector is to find signals present in the spectrum and identify their center frequency and bandwidth. The energy detector receives spectrum data from the FPGA FFT through the ARM's AXI bus. One node in the cluster is always configured to contain a FPGA FFT. By running the FFT on the FPGA, ARM resources are freed up to run algorithms less suited to FPGAs. The designated node performs a FFT on time-domain samples received from a data source and produces frequency-domain samples to the ARM processor.

The ARM processors perform a two-step peak finding process. First, the average of the spectrum and the maximum value are calculated. These two characteristics are used to establish a minimum threshold above the noise floor that signals must exceed in order to be considered in the second step. Next, the spectral bins that exceed the minimum threshold are put into groups by proximity. The center frequency and bandwidth of each cluster is reported to the cognitive engine.

Cognitive Engine

The role of the cognitive engine is to make decisions about what radios to deploy based on two factors: the current state of the system and the current RF spectrum. The state of the system can be further broken down in several components: active radios, time and available radio configurations. In this manner, the cognitive engine looks at spectrum data from the energy detector and compares that spectrum against a list of known waveforms. If there is a match, the cognitive engine checks if there is a comparable radio already running (to avoid duplicates). The time since the last reconfiguration is also taken into account, in order to avoid excessive reconfiguration. With all of these parameters taken into account, the cognitive engine produces a list of signals of interest and corresponding radios.

The cognitive engine determines which radios to deploy based on a set of user defined criteria loaded at runtime. The radio deployment criteria are stored in a comma separated values (CSV) file. For example, the CSV line “100000,5000,3,fm,fm100.grc”, indicates to the cognitive engine to use the GNU Radio companion file, “fm100.grc”, if a signal has a center frequency of 100 MHz and has a bandwidth of at least 5 KHz. The CSV line also indicates that the signal has a priority of three. Signals with lower priority numbers are selected when there are limited resources or the user puts a cap on the maximum number of demodulators. A CSV file can contain many entries describing the desired behavior of the radio given its RF environment. An adjustable tolerance is given to allow for variations between ideal and reported spectrum information.

Netlist Generation

After the cognitive engine has determined which signals are of interest and it has selected the corresponding radio(s) to deploy, a netlist of required parts is generated out of available GNU Radio components and FPGA accelerators. Individual radios are stored as GNU Radio companion files. Every GNU Radio block and FPGA accelerator used in the radios has a corresponding XML file describing its input, outputs and parameter. FPGA accelerators and GNU Radio components all share a common format. By using the same format, all components can be connected using the GNU Radio companion tool. FPGA accelerator components have an additional parameter that indicates to the Resource Mapper to instantiate an FPGA component instead of a GNU Radio software component. In this manner, constructing a mixed netlist of hardware accelerators and software is seamless to the user.

All of the individual XML files that describe the radio components and interconnects are combined into a single XML file. This file is then parsed and translated into a vector of GNU Radio blocks, FPGA accelerators and connections. Each block in the vector contains Python formatted commands to instantiate the block and import any Python library it requires. The functionality of the netlist generation program effectively creates an embedded version of GNU Radio companion translating XML into Python.

At runtime, the netlist generator creates a list of available GNU Radio blocks by searching specific install directories for GRC formatted XML files. The available part list associates part names with their corresponding XML files. After receiving a list of radios to deploy from the cognitive engine, the netlist generator concatenates the GRC files into one file adding prefixes to avoid duplicate names. Next, the netlist generator parses the combined GRC files to generate a list of blocks and connections. The XML file associated with each block in the list is then parsed for two items, the import string and the make string.

The import string contains all of the required Python imports necessary to use the block. The make string is the Python command to instantiate the block. The make string by itself contains a list of variables to be evaluated before the make statement can be used. The values for each of the variables comes from the GRC netlist. Therefore, the netlist generator iteratively replaces the variables in each make string with the user defined values from GRC. Table 3 shows an example conversion of a GNU Radio file source.

File Source XML

```
<block>
            <name>File Source</name>
            <key>blocks_file_source</key>
            <import>from gnuradio import blocks</import>
            <make>blocks.file_source($type.size*$vlen, $file, $repeat)</make>
</block>
```

GNU Radio Companion XML

```
<block>
            <key>blocks_file_source</key>
            <param>
            <key>id</key>
            <value>blocks_file_source_0_0</value>
            </param>
            <param>
            <key>file</key>
            <value>portal_320k_filesink</value>
            </param>
            <param>
            <key>type</key>
            <value>complex</value>
            </param>
            <param>
            <key>repeat</key>
            <value>True</value>
            </param>
            <param>
            <key>vlen</key>
            <value>1</value>
            </param>
  </block>
```

Evaluated Make String

```
blocks_file_source_0_0 = blocks.file_source(gr.sizeof_gr_complex*1, "portal_320k_filesink", True)
```

Evaluated Import String

```
from gnuradio import block
```

Table 3 - Evaluating make string for GRC XML blocks

Resource Mapper

The resource mapper receives a netlist from the netlist generator and assigns components to available resources. The mapper takes care of any intermediate connections needed to span between resources in the cluster. Consider the case where an FPGA accelerator on one FPGA is connected to a file sink on another board's ARM processor. The resource mapper will connect the FPGA accelerator to the FMC interboard links between the two boards and connect to the DMA system on the second board to complete the connection. This process happens automatically, without user intervention and without changing the functionality of the original netlist.

The inclusion of a resource mapping program allows the netlist generator to function completely agnostic of the hardware. This hierarchical separation allows for greater flexibility and scalability. For example, if more Zynq boards were added to the cluster, the only program in the system that needs to be updated to the change is the mapper. All other programs are functionally unchanged by the architecture change. Future versions of the system could contain a feedback mechanism to enable load balancing within the cluster. The resource mapper produces an FPGA netlist and the GNU Radio flow graphs that will be run on the ARM cores in the system. The resource mapper employs the tool Tflow [5] to quickly generate bitstreams.

Tflow

To meet the real time requirements of an adaptive radio, the bitstream generation process needed to be accelerated. In a traditional system, this process would involve regenerating an entire

FPGA configuration bitstream, a process involving synthesis, mapping, place and route, and a bit stream generation. To address this issue, Tflow was utilized. Tflow creates two distinct regions for a design, the non-changing “static” portion, and the dynamic “blacktop” region.

The static region contains components that are consistent across all designs, such as interfaces to the ARM processor. Prebuilt macro blocks are placed in the blacktop region as specified by the user. The prebuilt macro blocks are placed and routed beforehand so that after they are placed in the blacktop, the tools only need to route between macro blocks. This use of macro blocks and a static region greatly reduce the bit stream generation time, thereby allowing the system to adapt to a rapidly changing RF environment.

Figure 12 illustrates the default configuration for all Zynq boards in the cluster. The static region consists of the DMA system and the FMC interboard links. Each board has four receive and four transmit DMA interfaces accessible through a Linux device driver. Every board also has four receive FMC ports and three FMC transmit ports. The boards only have three FMC transmit ports as they do not send data back to the data source, and thereby only need a unidirectional link. The static region and dynamic blacktop region are connected through FIFOs at each individual interface.

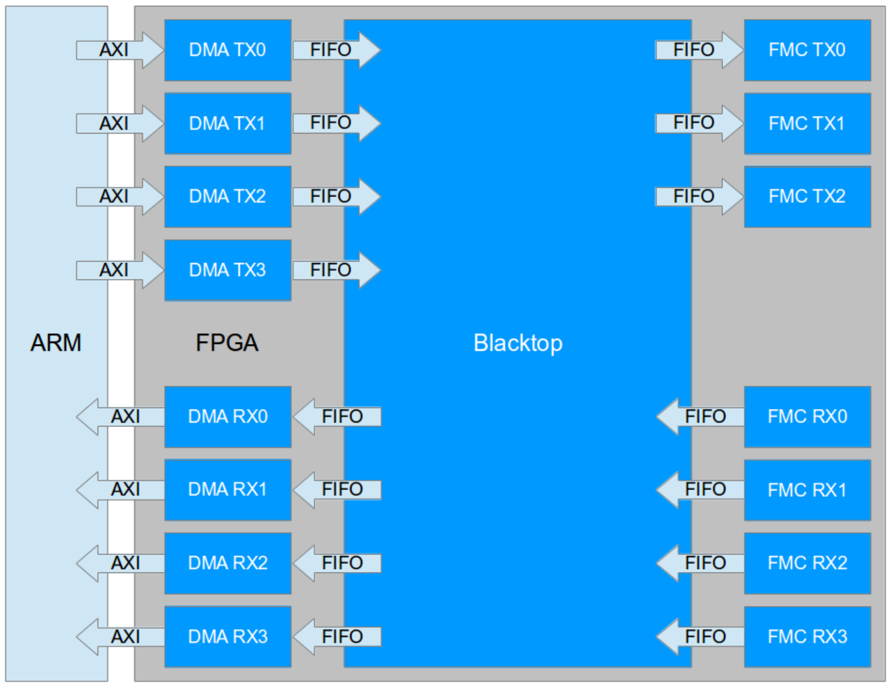


Figure 12 - Tflow regions of Zynq boards

Table 4 shows the FPGA resource utilization for the static region (DMA, FMC and FIFOs). While the static region requires 30% of the available lookup tables (LUTs), it does not use any of the available DSP slices. This is fortunate as the DSP slices can be reserved for accelerators in the blacktop.

Resource	**# of Resources Available**	**# of Resource Used**	**Utilization %**
Look Up Tables	53,200	16,331	30%
Flip Flops	106,400	16,087	15%
Block RAM (36 Kb)	240	25	10%
DSP Slices	220	0	0%

Table 4 - Zynq XC7Z020 FPGA resource utilization

On an x86 processor, it takes approximately 15 minutes to generate the static region. If Xilinx tools were used, each time a new bit file needed to be generated there would be a 15 minute wait. This build time is far too large for use in an adaptive cognitive radio system, where spectrum could change many times in a 15 minute interval. Also, it should be noted that Xilinx does not support ARM processors, so this kind of an embedded system could not exist.

As previously mentioned, Tflow places and routes pre-compiled macro blocks, greatly reducing the time to generate a bit file. On the ARM processor, it takes approximately 90 seconds to generate a bit file. This enables the cognitive radio to adapt to a new environment in 90 seconds. Radio designs without significant FPGA components take approximately 45 seconds to build. By caching previously built designs, reconfiguration time is reduced to a fraction of a second. After receiving an FPGA netlist from the resource mapper, each board in the cluster generates its own bit file using Tflow.

Reconfiguration

After Tflow produces bit files for each of the FPGAs in the system, the final step is to reconfigure the FPGAs. Traditionally, FPGAs are configured with either with a Joint Test Action Group (JTAG) cable or from a read-only memory (ROM). For a typical development system, programming via JTAG is a viable option. However, in an embedded design, a host system might not be present to program the boards. Fortunately, Linaro Linux has a device driver to program the FPGA. FPGAs are configured by writing the binary version of the Tflow generated bit file to the device driver as if it were a file. Furthermore, as the boards are all connected via gigabit Ethernet, each board can be configured remotely through a secure shell (SSH) connection. Another advantage of configuring the FPGAs through the device driver is that it removes the need to have a dedicated JTAG cable for each board in the cluster.

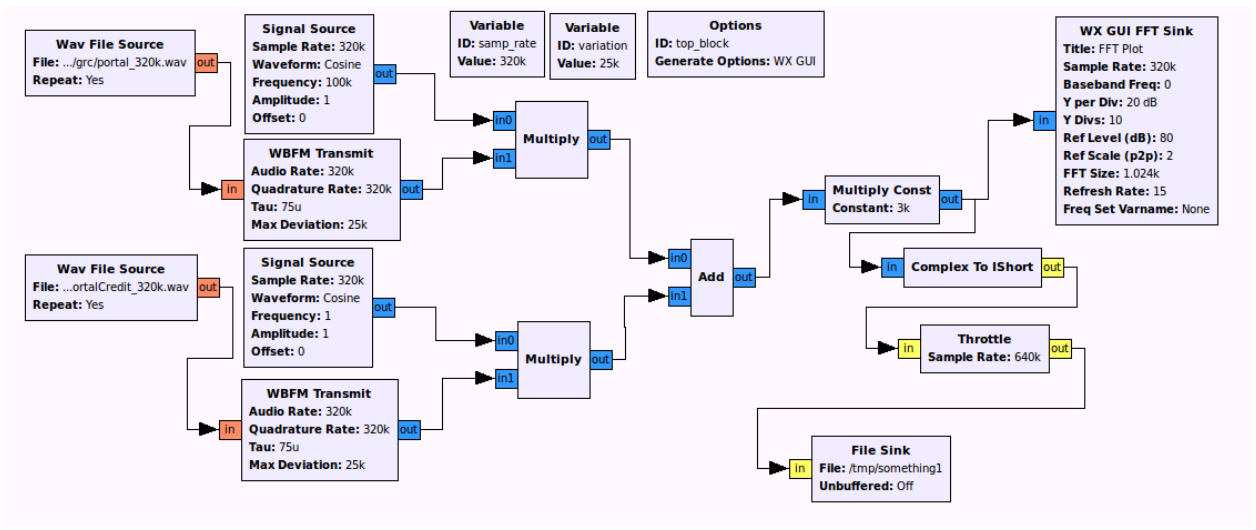
Chapter 5

System Demonstration

In this chapter, the complete cognitive radio system will be demonstrated. Two signals are sent from the data source to the cognitive radio, which then detect the spectrum and associate one of the signals with a known radio configuration. The cognitive radio then constructs the desired radio configuration through the use of GNU Radio software components and hardware accelerators. Tflow is utilized to accelerate bit file generation on each node. Finally, the cluster is reconfigured to deploy a radio receiver and the receiver begins streaming audio to the host for verification.

Frequency Modulation Demonstration

To demonstrate the adaptive nature of the described cognitive radio, the Zedboard data source was configured to transmit two frequency modulated (FM) signals at two different center frequencies. The first signal is transmitted with a center frequency of 1 Hz and the second signal is transmitted with a center frequency of 100 MHz. The signals were generated by first resampling the original audio files to 320 KHz using the Linux command line tool “sox”. Figure 13 shows the GRC flowgraph used to transmit data to the Zynq cluster via the FMC connectors. The “File Sink” seen in Figure 13 connects to the DMA device driver which passes data to the appropriate FMC transmit channel.



Wav File Source
File: .../grc/portal_320k.wav
Repeat: Yes

Signal Source
Sample Rate: 320k
Waveform: Cosine
Frequency: 100k
Amplitude: 1
Offset: 0

Variable
ID: samp_rate
Value: 320k

Variable
ID: variation
Value: 25k

Options
ID: top_block
Generate Options: WX GUI

WX GUI FFT Sink
Title: FFT Plot
Sample Rate: 320k
Baseband Freq: 0
Y per Div: 20 dB
Y Divs: 10
Ref Level (dB): 80
Ref Scale (p2p): 2
FFT Size: 1.024k
Refresh Rate: 15
Freq Set Varname: None

WBFM Transmit
Audio Rate: 320k
Quadrature Rate: 320k
Tau: 75u
Max Deviation: 25k

Multiply

Multiply Const
Constant: 3k

Add

Complex To IShort

Wav File Source
File: ...ortalCredit_320k.wav
Repeat: Yes

Signal Source
Sample Rate: 320k
Waveform: Cosine
Frequency: 1
Amplitude: 1
Offset: 0

Multiply

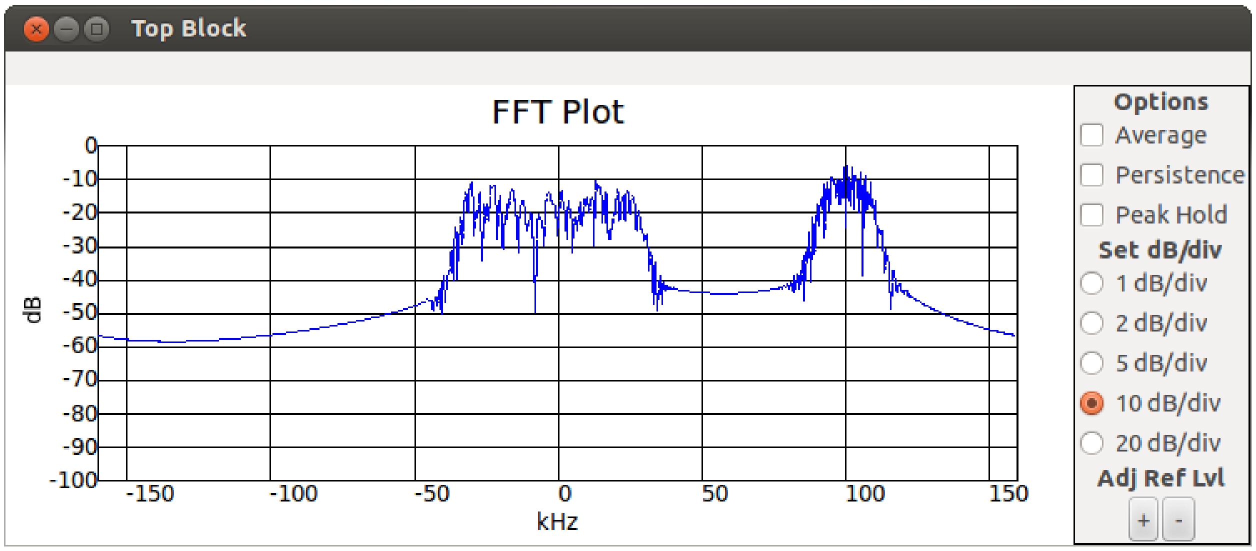
Throttle
Sample Rate: 640k

WBFM Transmit
Audio Rate: 320k
Quadrature Rate: 320k
Tau: 75u
Max Deviation: 25k

File Sink
File: /tmp/something1
Unbuffered: Off

Figure 13 - Zedboard data source GRC flowgraph

The spectrum of the two transmitted signals is shown in Figure 14. The "WX GUI FFT Sink" in the Figure 13 was used to capture a snapshot of the transmitted data.



Options
☐ Average
☐ Persistence
☐ Peak Hold

Set dB/div
1 dB/div
2 dB/div
5 dB/div
[✓ 10 dB/div]
20 dB/div

Figure 14 - Spectrum of Zedboard transmitted data

A host machine is configured to run the GNU Radio flowgraph seen in Figure 15. The flowgraph consists of a user datagram protocol (UDP) sink connected to an audio sink block. This host application functions as a feedback mechanism, which allows the user to hear when the correct radio has been launched.

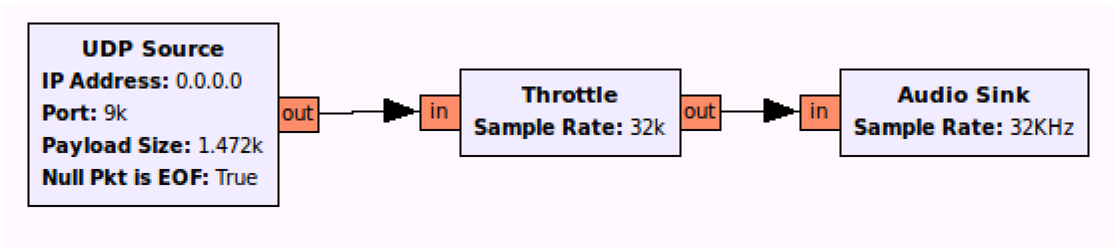


Figure 15 - Host GNU Radio flowgraph

At runtime, the Zynq cluster is configured to contain a hardware FFT block. The FFT block is placed in the dynamic blacktop region of the FPGA, as seen in Figure 16. The exact board that the FFT resides on is determined by the resource mapper at runtime and returned to the energy detector as a parameter. The initial configuration can be thought of as a special case where the desired netlist is provided to the cluster by the user instead of determined by the detected spectrum data. The initial configuration is still run through the mapping and Tflow software, just like all subsequent configurations.

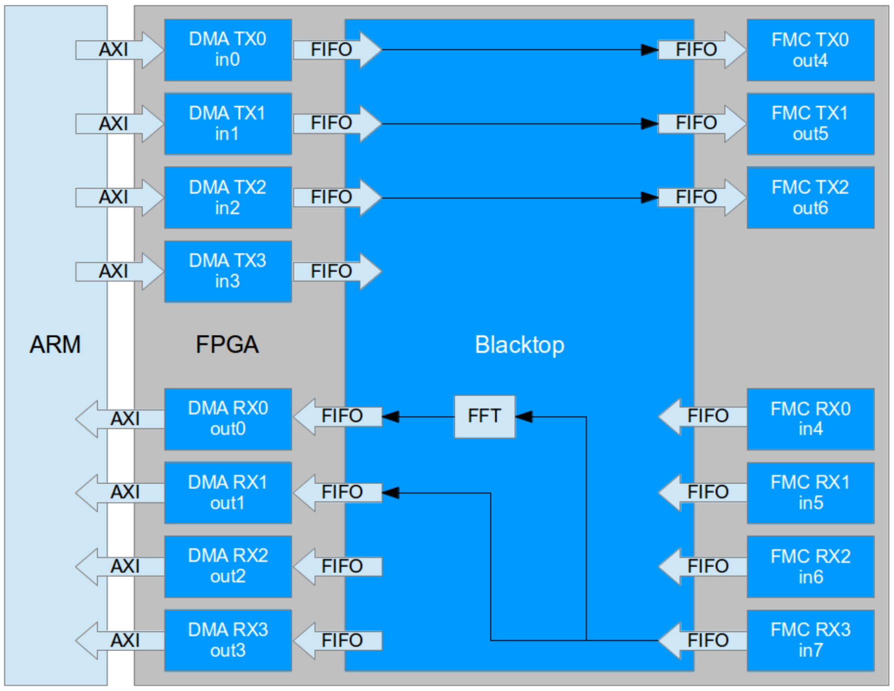


Figure 16 - Initial FPGA configuration

The energy detection software reports any peaks in the RF spectrum to the cognitive engine. Figure 17 shows the received spectrum data from the FFT with the center frequencies and bandwidth of the signals indicated. A horizontal line establishes the minimum threshold a frequency bin must exceed to be consider a signal. Vertical lines bound the peak clusters and denote the bandwidth of the detected signals. Three arrows indicate gaps between peaks that are ignored to ensure that wider signals are not interpreted as several narrow signals.

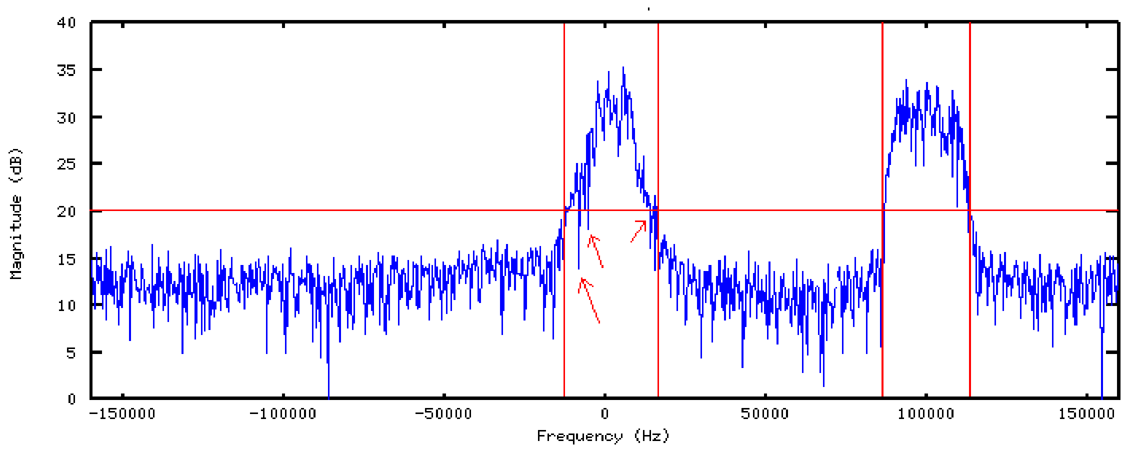


Figure 17 - Frequency domain plot of RF spectrum

Figure 18 shows spectrum information generated by the energy detector for the two signals seen in Figure 17.

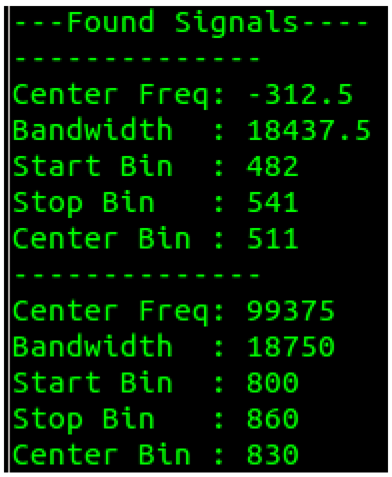

```
---Found Signals----
--------------
Center Freq: -312.5
Bandwidth  : 18437.5
Start Bin  : 482
Stop Bin   : 541
Center Bin : 511
--------------
Center Freq: 99375
Bandwidth  : 18750
Start Bin  : 800
Stop Bin   : 860
Center Bin : 830
```

Figure 18 - Signal data reported from energy detector to cognitive engine

The cognitive engine compares this information with the preloaded signal list. In this case, the cognitive engine is initialized to associate a peak at 100 MHz with a FM radio signal. The cognitive engine then directs the netlist generator to build a netlist containing a 100 MHz FM radio receiver and a UDP source (seen in Figure 15) to communicate with the host. Also in the generated netlist are hardware accelerated FIR filter configured as a low pass filter and a cosine source used to mix the received signal down from 100 MHz to baseband. The resource mapper distributes the netlist across the cluster generating the corresponding Python programs (GRC) for each board. Each node runs Tflow and generates its own FPGA bit file. Finally, the cluster is reconfigured and the host begins playing the demodulated audio it receives via UDP from the Zynq cluster. Figure 19 shows the generated netlist containing an FM demodulator.

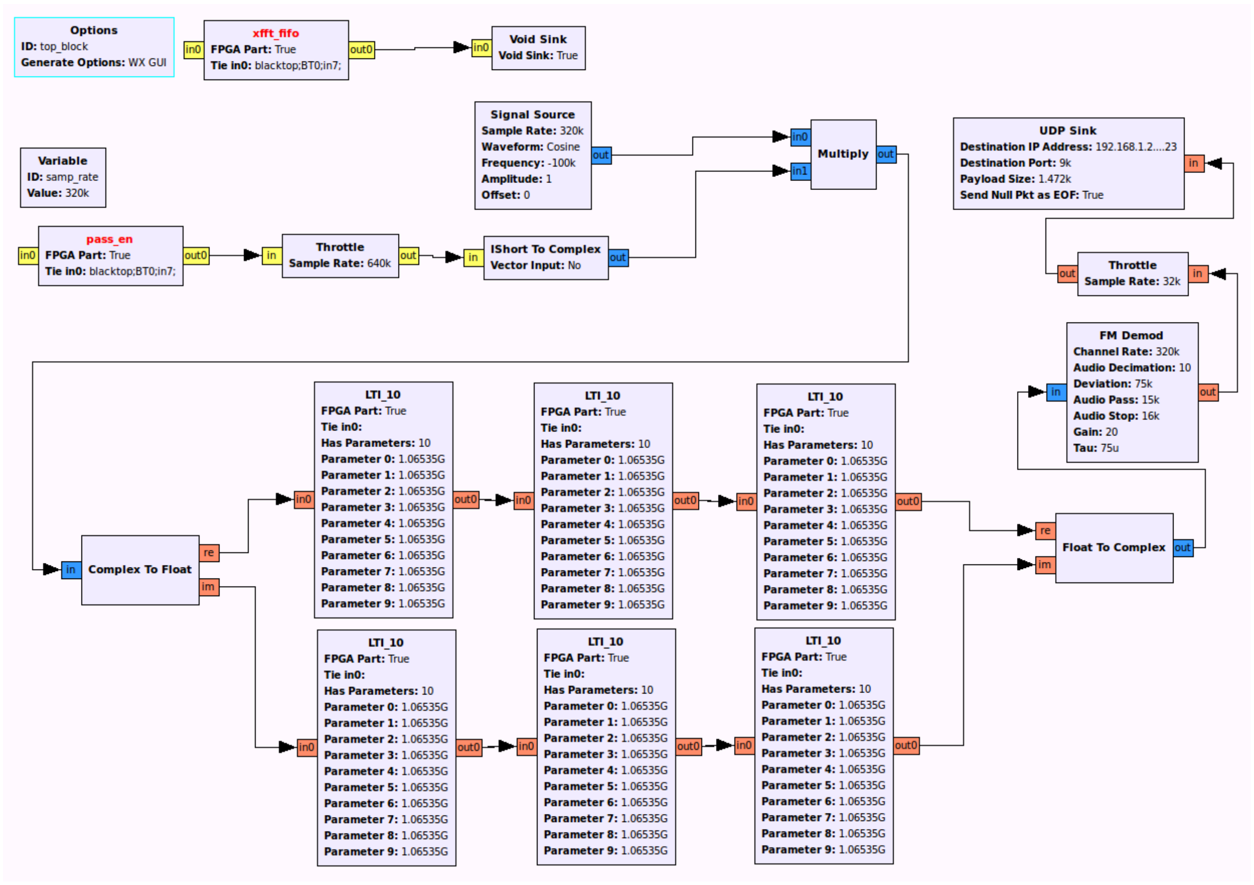


Figure 19 - TFlow generated netlist containing FM radio demodulator

Result Comparison

Table 5 shows a comparison of the cognitive radio described in this thesis with published cognitive radio systems and adaptive radio systems. Compared radio systems were selected for two criteria: self reconfiguration and use a platform with both an FPGA and a general purpose processor.

Name	SDR Framework	Processor	OS	FPGA	Reconfiguration Method
Zynq Cognitive Cluster	GNU Radio	Dual ARM Cortex A9	Linux	(4) Xilinx Artix-7	Tflow + ICAP
Generic Software Framework for Adaptive Applications on FPGAs [13]	IRIS	PowerPC (soft core)	Linux	Virtex II Pro	Xilinx Partial Reconfiguration + ICAP
Autonomous System on a Chip Adaptation through Partial Runtime Reconfiguration [14]	N/A	PowerPC 405	Linux	Xilinx Virtex 4 FX	Xilinx Partial Reconfiguration + ICAP
Partial reconfiguration in the implementation of autonomous radio receivers for space [15]	N/A	Unnamed “Embedded microcontroller”	Unknown	Xilinx Virtex 5 LX50T	Xilinx Partial Reconfiguration + ICAP

Table 5 - Comparison of related works

For all radio systems listed, the Xilinx Internal Configuration Access Port (ICAP) is used to configure the FPGA. With the exception of the cognitive radio platform described in this thesis, all the radio systems in Table 5 use the Xilinx partial reconfiguration flow [16]. The Xilinx partial reconfiguration flow uses static and a dynamic reconfigurable region. Components that can be placed in the reconfigurable region are stored as pre-made partial bit files. Components are generated for specific reconfiguration regions. In order to utilize a component in multiple reconfiguration regions, a separate partial bit file must be generated for each region. Take for example the case where there are four reconfiguration regions and five components. In order to enable all components to be used in any reconfiguration region, a total of 20 partial bit files would need to be generated and stored on the device. In a system with limited disk space, storing large numbers of partial bit files becomes infeasible.

The cognitive radio cluster described in this thesis uses Tflow. Tflow, in contrast with the Xilinx partial reconfiguration flow, uses one large reconfiguration region and generates bit files dynamically at runtime. In this manner, Tflow can generate significantly more combinations of components without storing all possible combinations of parts. By generating the bit file dynamically, Tflow offers more flexibility than the Xilinx partial reconfiguration solution seen in [13][14][15]. Take for example, an applications that dynamically generate netlists. Such an application is fully supported with Tflow, however with the Xilinx partial reconfiguration flow such an application is not realizable.

The additionally flexibility offered by Tflow over pre-generated bit files is at the expense of configuration time. Programming pre-generated files is significantly faster than running the entire Tflow chain for new radio configurations. However, by caching previously built Tflow designs configuration times are comparable. In this manner, Tflow offers flexibility in addition to comparable configuration times as the Xilinx reconfiguration flow for known designs.

The cognitive radio described in this thesis utilizes the GNU Radio framework. Of the radio systems in Table 5, only [13] and the cognitive radio in this thesis use a SDR framework. By leveraging an existing SDR framework, design effort is significantly reduced when creating software defined radio applications. In [13] the general-purpose processor is an PowerPC softcore implemented in a Xilinx Virtex II FPGA. The computational power of such a severely restricted in comparison with the dedicated ARM core found in the system develop in this thesis.

Chapter 6

Conclusion and Future Work

This work culminated in a demonstration of a cognitive radio first sensing its spectrum, then identifying a corresponding radio to deploy and finally reconfiguring itself to begin receiving the detected waveform. Radio components were implemented in both hardware and software communicating via AXI for on chip communication and through FMC connectors for inter board communication. Software components of the radio were drawn from the GNU Radio framework. Additionally, FPGA based radio components were used to supplement the dual core ARM processor within the Zynq. Using standardized interfaces allow radios consisting of hardware and software to be designed as a single GNU Radio flowgraph. A mapping program assigned components to available cluster resources. Tflow was used to accelerate the FPGA bit file generation process.

The cognitive engine used in the system demonstration associated the center frequency of a detected waveform with a radio implementation. Future work could be done to expand the characterization of the detected waveforms to extract additional information. Such information might include baud rate or modulation scheme. Using this information, more complicated associations could be made between the extracted spectral characteristics and the appropriate radio to deploy. Additional work could be done to create more complex, parameterizable radios. The radios used in the system demonstration were made specifically for the sampling rates and center frequencies used in the demonstration.

To enabled the cognitive engine to make more intelligent decisions, information about the performance of currently deployed radios could be gathered. This information would provide the cognitive engine with feedback regarding the performance of the current system configuration. Such information might include statistics about bit error rates (BER) or system load. This information would allow the cognitive engine to function as closed loop, “watchdog” system, making continuous corrections to radio deployments to optimize parameters such as date rate, power efficiency or reception range. Consider the scenario where the platform is configured to receive a waveform with specific baud rate. Next, consider that the baud rate of the original waveform changes. The resulting spike in the BER of the original radio deployment will be reported to the cognitive engine. The cognitive engine can then determine how to reconfigure the system to adapt to the change.

The cognitive engine could be further extended to optimize the system for multiple radio deployments. In the current system, pre-designed radios are manually assigned a resource utilization value. This value attempts to characterize the system resources required to deploy the specific radio. The cognitive engine will try to deploy as many radios as possible (that correspond to detected and verified waveforms). The cognitive engine will continue adding radios to the deployment list until the cummulative system utilization is 100% or there are not any additional radios to deploy. Future improvements could dynamically gather information about the system load for each radio at runtime. This information could be fed back into the cognitive engine to adjust the utilization value. Additional work could also be performed to extract information about current and previous deployed radios to predict the performance of future radios. Such information could be used to make adjustments to radios before they are deployed thereby reducing the reconfiguration time required to achieve optimal radio deployments.

This work establishes a platform for more advanced cognitive radio systems. Future work can operate at a higher level of abstraction, without worrying about low level details such as data transfers, resource allocation and board reconfiguration. Instead, researchers can focus on larger picture concepts such as learning algorithms and behavioral models for the cognitive engine. Learning algorithms could be applied to the cognitive engine to create a software defined radio that not only reacts but adapts to its environment.

Bibliography

[1] J. Mitola, “Software Radios Survey, Critical Evaluation and Future Directions,” *IEEE AES Syst. Mag.*, pp. 25–36, April. 1993.

[2] A. He, K. K. Bae, T. R. Newman, J. Gaeddert, K. Kim, R. Menon, L. Morales-tirado, J. J. Neel, Y. Zhao, J. H. Reed, and W. H. Tranter, “A Survey of Artificial Intelligence for Cognitive Radios,” *IEEE Trans. Veh. Technol.*, vol. 59, no. 4, pp. 1578–1592, 2010.

[3] I. F. Akyildiz, W.-Y. Lee, M. C. Vuran, and S. Mohanty, “NeXt generation/dynamic spectrum access/cognitive radio wireless networks: A survey,” *Computer Networking*, vol. 50, no. 13, pp. 2127–2159, Sep. 2006.

[4] National Instruments, “LabView.” [Online]. Available: http://www.ni.com/labview/.

[5] A. Love, W. Zha, and P. Athanas, “In pursuit of instant gratification for FPGA design,” in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, 2013, pp. 1–8.

[6] R. H. L. Stroop, “Enhancing GNU Radio for Run-Time Assembly of FPGA-Based Accelerators,” 2012.

[7] J. van de Belt, P. D. Sutton, and L. E. Doyle, “Accelerating software radio: Iris on the Zynq SoC,” in *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013, pp. 294–295.

[8] Z. Lin and P. Chow, “ZCluster : A Zynq-based Hadoop Cluster,” pp. 450–453, 2013.

[9] Xillybus, “Xillinux: A Linux distribution for Zedboard, ZyBo, MicroZed and SocKit.” [Online]. Available: http://xillybus.com/xillinux.

[10] Tilera Corporation (San Jose), “TILE PROCESSOR ARCHITECTURE OVERVIEW FOR THE TILEPRO SERIES,” 2013. [Online]. Available: http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf.

[11] Xilinx, Inc (San Jose)“Zynq-7000 Combined Product Table.” [Online]. Available: http://www.xilinx.com/publications/prod_mktg/zynq7000/Zynq-7000-combined-product-table.pdf.

[12] Linaro, “Linaro Ubuntu.” [Online]. Available: http://www.linaro.org/.

[13] S. A. Fahmy, J. Lotze, J. Noguera, L. Doyle, and R. Esser, “Generic Software Framework for Adaptive Applications on FPGAs,” in *Field Programmable Custom Computing Machines, 2009. FCCM ’09. 17th IEEE Symposium on*, 2009, pp. 55–62.

[14] M. French, E. Anderson, and D.-I. Kang, “Autonomous System on a Chip Adaptation through Partial Runtime Reconfiguration,” in *Field-Programmable Custom Computing Machines, 2008. FCCM ’08. 16th International Symposium on*, 2008, pp. 77–86.

[15] G. C. Cardarilli, M. Re, I. Shuli, and L. Simone, “Partial reconfiguration in the implementation of autonomous radio receivers for space,” in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2011 6th International Workshop on*, 2011, pp. 1–6.

[16] Xilinx, “UG702: Partial Reconfiguration User Guide,” 2010. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_1/ug702.pdf.