Hybrid FPGA and GPP Implementation of IEEE 802.15.4 Physical Layer

Jeong-O Jeong

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

Electrical Engineering

Carl B. Dietrich

Jeffrey H. Reed

Peter Athanas

July 30, 2012

Blacksburg, Virginia

Keywords: Software Defined Radio, FPGA, IEEE 802.15.4, ZigBee, USRP N210

Hybrid FPGA and GPP Implementation of IEEE 802.15.4 Physical Layer

Jeong-O Jeong

ABSTRACT

In this thesis, two different cases of hybrid IEEE 802.15.4 PHY (Physical Layer) implementation are explored. The first case is an FPGA implementation of IEEE 802.15.4 PHY on the Xilinx Spartan-3A DSP FPGA of USRP N210. All of the signal processing tasks are performed on the FPGA, while less complex MAC (Media Access Control) layer tasks are performed in GNU Radio on the host. The second case is an implementation of a multi-channel IEEE 802.15.4 receiver. A four-channel channelizer is implemented on the external Virtex 5 FPGA, while the IEEE 802.15.4 receiver is implemented in GNU Radio on the host. The first case demonstrates how spare resources in USRP's FPGA can be used to implement signal processing task while still interfacing with GNU Radio. The second case builds a platform on which a combination of GNU Radio and an external FPGA can be used for signal processing and USRP as an RF source. This thesis lays out the groundwork for more complex wireless protocols to be implemented on any combination of USRP's FPGA, an external FPGA, and GNU Radio.

Acknowledgments

I am grateful to Dr. Carl Dietrich who have guided and supported me throughout my years in graduate school. This work would not have been possible without the help and support from Dr. Dietrich, Dr. Reed, Dr. Athanas, Dr. Gaeddert, and friends from Wireless @ VT and CCM Lab.

Dedication

To my family

Contents

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
ASK	Amplitude-Shift Keying
CIC	Cascaded Integrator-Comb
CLB	Configurable Logic Block
CRC	Cyclic Redundancy Check
CSMA-CA	Carrier Sense Multiple Access Collision Avoidance
DAC	Digital-to-Analog Converter
DFT	Discrete Fourier Transform
DSP	Digital Signal Processor
DSSS	Direct Sequence Spread Spectrum
FFT	Fast Fourier Transform

FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
GPP	General Purpose Processor
HDL	Hardware Description Language
ISM	Industrial, Scientific and Medical
LUT	Look-Up Table
MAC	Media Access Control
MAC	multiply-accumulate
MSK	Minimum-Shift Keying
O-QPSK	Offset Quadrature Phase-Shift Keying
PHR	PHY Header
PHY	Physical Layer
PN	Pseudo-random Noise
PPDU	PHY Protocol Data Unit
PSDU	Physical Layer Service Data Unit
RTL	Register-Transfer Level

SDR	Software Defined Radio
SFD	Start of Frame Delimiter
SHR	Synchronization Header
SOC	System on Chip
SVD	Singular Value Decomposition
UDP	User Datagram Protocol
USRP	Universal Software Radio Peripheral
VITA	VMEbus International Trade Association
VRT	VITA Radio Transport
WPAN	Wireless Personal Area Network

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Software Defined Radio (SDR) attempts to leverage the flexibility of software or reconfigurable hardware to implement flexible radios that can easily switch to different waveforms and standards. This is achieved by migrating most of the signal processing traditionally implemented on hardware onto software. By implementing the signal processing region with software, the matter of reconfiguring the system is simply loading a different image, which can be done even in real-time. The most common platforms on which SDR is implemented are GPP (general purpose processor), DSP (digital signal processor),and FPGA (field-programmable gate array). Each of these platforms has its own advantages and disadvantages. Although GPPs are relatively easy to program, test, and verify, they

are often not well-suited for parallel, signal-processing intensive, or real-time constrained operations. DSPs are optimized for certain signal processing tasks, but they are harder to program than GPPs and not optimized for parallel computations. FPGAs are well-suited for parallel and signal-processing intensive computations, but they are much harder to program and verify. However, many high-level tools such as Simulink and ImpulseC are available to facilitate the programming of FPGA. These tools allow engineers to describe the algorithm at high-level that can be automatically converted to HDL (Hardware Description Language) to be implemented on the FPGA.

As mentioned, GPPs and DSPs are not the best platforms on which signal-processing intensive algorithms can be implemented. Processing power necessary for even one of the simplest wireless protocols such as IEEE 802.15.4 can be too demanding for GPP unless it is a very high-performance GPP. Even with a high-performance GPP, signal processing required for IEEE 802.15.4 consumes most of CPU cycles. It can be seen that wideband wireless protocols such as IEEE 802.11a, which has 20 MHz of bandwidth and complex modulation scheme such as 52-subcarrier OFDM will be too demanding for GPPs. However, FPGAs, with their ability to perform computations in parallel, support high throughput and high sampling rate that GPPs or DSPs are not able to achieve. ASICs are a popular platform for implementing high-performance protocols, but they have very low re-configurability. With its software-like configurability, FPGAs bring significant reduction in NRE (non-recurring engineering) costs. Once a prototype is built and fault is found in the design, FPGAs can be reconfigured with a modified bit-stream and can

be tested again, whereas ASICs need to be re-spun at a very high-cost, often in millions of dollars [3]. Because FPGAs are high-performance like ASICs and flexible like GPPs and DSPs, they are an ideal platform on which prototype wireless protocols can be built and tested. Additionally, the partial reconfiguration ability of FPGA allows it to reconfigure itself in real-time, which unlocks the SDR's promises of concurrent multi-protocol operation.

1.2 Previous Studies

Many implementations of SDR have been done on FPGAs. At Virginia Tech, Charles Irick from Configurable Computing Lab developed an SDR framework which improves upon the GNU Radio framework by allowing an auxiliary Virtex-5 FPGA. In the enhanced GNU Radio framework, a software block representing the auxiliary FPGA controls the dataflow within the software environment. A single high-level software description of GNU Radio and FPGA leaves the mixture of software blocks and the FPGA block transparent to the programmer [4]. More recently, Richard Stroop from the same lab has extended Irick's work to develop a framework called GReasy (GNU Radio Easy). Whereas Irick's work only used a USRP2 as RF front-end and a single auxiliary FPGA, GReasy has successfully worked with not only USRP2 but also a 3.6GSPS ADC as an RF front-end. It has also interfaced with four auxiliary FPGAs for distributed FPGA processing. More importantly, the framework allows rapid reconfiguration of the FPGA by placing and

routing pre-compiled modules. Thus, GReasy presents the user interface where FPGA blocks can be as easily placed and rearranged as software blocks, and the underlying mechanism for compiling the FPGA remains transparent to the user.

In Karve's Master's thesis, the enhanced GNURadio framework developed by Irick was used to implement a ZigBee receiver. The O-QPSK demodulator was implemented on the FPGA. An off-the-shelf ZigBee-compliant solution called XBee was used to transmit packets, which were then received by a ZigBee receiver developed with the framework for verification of interoperability [2].

Implementations of SDR based on other platforms such as GPPs and DSPs are also available. An SDR platform known as SORA is developed on commodity PC architectures. The platform consists of a radio-front end, a radio control board, and a PCIe bus to the CPU cores. Researches have developed full IEEE 802.11 a/b/g PHY and MAC layers on the SORA platform and successfully interoperated with commercial IEEE 802.11 a/b/g devices. It was possible to implement such a complex standard by introducing optimizations such as replacing complex computations with extensive use of LUTs (Look-Up Tables) in L2 cache and use of SIMD (Single Instruction Multiple Data) instruction sets [5].

Other SDR platforms are based on custom reconfigurable hardware. The AsAP2 (Asynchronous Array of Simple Processors) platform from University of California Davis is composed of an array of 164 simple processors and Viterbi and FFT accelerators. Each processor has own instruction and data memory as well as arithmetic and logic unit. Using this platform, they were able to implement a complete IEEE 802.11a baseband receiver

[6].

1.3 Goals

The following list outlines the goals for the thesis.

- To implement IEEE 802.15.4 PHY on Xilinx Spartan 3A-DSP of USRP N210
- To implement a channelizer on an external Virtex 5 FPGA
- To interface GNU Radio and the external Virtex 5 FPGA for hybrid implementations
- To compare performance between GNU Radio and FPGA implementations of IEEE 802.15.4 PHY
- To lay the groundwork for more complex wireless communication protocols and applications to be implemented on FPGA with USRP N210 as an RF front end
- To develop open-source IP cores to be freely used in other SDR projects

1.4 Accomplishments and Contributions

The main accomplishment of this thesis is the implementation of the IEEE 802.15.4 PHY on the FPGA. The FPGA implementation was able to successfully inter-operate with a commercially available, standard-compliant ZigBee module as well as an open-source

GNU Radio implementation. Secondly, the interfacing between USRP N210, an external FPGA and GNU Radio with UHD (Universal Hardware Driver) has been implemented. This enables the hybrid implementation of a waveform where a more complex signal processing task such as channelization is performed on the external FPGA, while a simpler task such as demodulation of an IEEE 802.15.4 packet is performed on GNU Radio. Finally, numerous FPGA blocks developed in the course of this thesis work have been made available to Configurable Computing Lab at Virginia Tech for the GReasy project.

Chapter 2

Background

2.1 Software Defined Radio

The term Software Defined Radio, coined by Joe Mitola in 1991, describes a radio whose physical layer is implemented mostly in software. It is a radio "whose physical layer behavior can be significantly altered through changes to its software" [7]. Common platforms for SDR include General Purpose Processors (GPP), Digital Signal Processors (DSP), Field Programmable Gate Arrays (FPGA), and recently even Graphics Processing Unit (GPU). Traditionally, radios are implemented in hardware, which makes it difficult to modify or upgrade after deployment. However, the software-defined nature of SDRs allows ease of modification and flexibility not found in hardware-defined radios. Because of its flexibility, SDRs can be prepared for "proliferation of wireless standards in the fu-

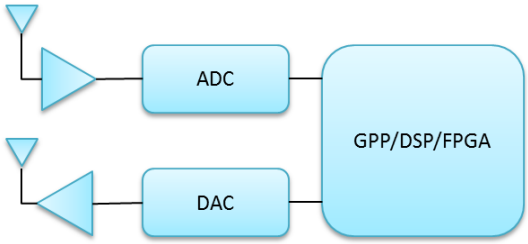

Figure 2.1: Ideal Software Defined Radio

ture" [8]. Due to the flexible and reconfigurable nature of software, SDRs can support multiple air interfaces and multiple modulation schemes as needed.

In order to reach this stage of flexibility, a radio platform close to an ideal SDR model shown in 2.1 is required. An ideal SDR radio would only consist of wide-band antennae that cover the entire RF spectrum, ADC and DAC fast enough to convert that spectrum to digital and analog domains, and processing elements that can process the wideband data. It may include an amplifier for better SNR. All the filtering, up-conversion, and down-conversion are performed in the software-defined processing element. This enables an ideal SDR that is not limited to any band or any modulation scheme.

2.1.1 SDR Platforms

The three most popular SDR platforms are GPP, DSP, and FPGA. Each has its own unique advantages and disadvantages.

GPP (General Purpose Processor)

GPP is a processing unit used for a variety of purposes such as fixed and floating point arithmetic, memory interface, and general input and output. It supports multiple high-level programming languages, and therefore it is the most flexible and easiest to program. However, GPPs generally do not perform computation in parallel and are not optimized for arithmetic operations. For example, the multiply-add operation, the most common operation in signal processing, is not supported in hardware in GPPs.

However, modern microprocessors have started to employ different ways of parallelism to improve DSP and graphics operations. One example is hyperthreading in Intel microprocessors. Hyperthreading allows a single core to act as two logical cores that can execute threads in parallel. Each logical core has its own processor architectural state and shares the execution resources of the physical core. This results in a performance gain of 30 percent when executing multithreaded applications compared to a processor without hyperthreading [9].

Another example of parallelism is the SIMD (single instruction multiple data) technology, known as SSE (Streaming SIMD Extensions) in Intel and 3DNow! in AMD processors. It allows a single instruction to be applied on multiple data simultaneously. This is more efficient than the traditional SISD (single instruction single data) where a single instruction is executed on single data at a time. FFTW, the fastest software implementation of the FFT algorithm, takes advantage of SIMD to achieve its title [10].

The most well-known and established SDR platform for GPP is an open-source software called GNU Radio. It is a software development platform that enables researchers to build software radios from a library of signal processing blocks. While the signal processing blocks are written in C++, the glue logic that connects the blocks is written in Python. Users can also develop their own custom signal processing blocks for their application.

Like FFTW, GNU Radio also takes advantage of SIMD with its VOLK (vector-optimized library of kernels) library. The library consists of vector operations that provide much improvement in performance. For example, a function called *volk_32fc_multiply_aligned16(c, a, b, N)* can perform vector multiplication of two vectors with N items. Without SIMD, the operation would be performed using a standard for-loop that multiplies each vector element, but SIMD allows simultaneous multiplication of the vector elements. Rondeau reports ten percent improvement in speed when using the function in the FFT filters where large number of multiplies may be required [11].

An example of a wireless standard implemented in GNU Radio is the ZigBee PHY implementation by Thomas Schmid from UCLA [12]. The signal processing is done on the general purpose processor with the USRP (Universal Software Radio Peripheral) as the RF front-end. The implementation was verified with a commercially available ZigBee radio compliant with the IEEE 802.15.4 standard. Because of issues such as high latency in GNU Radio, the full protocol stack could not be implemented, but only the physical layer was implemented. The physical layer implemented in this project was 2450 MHz O-QPSK PHY.

Schmid reports that even the high-performance machine they used, a dual Pentium IV, 2.8 GHz with hyperthreading and 4 GB of RAM, could not decode a constant stream of data from the USRP. To measure its performance, the throughput of GNU Radio implementation was measured. At 45 bytes per message, including both the MAC layer payload of 27 bytes and extra bytes at the PHY layer, the GNU Radio implementation could decode slightly above 200 messages per second.

DSP (Digital Signal Processor)

DSP is a microprocessor optimized for mathematical operations, specifically multiply-accumulate (MAC) functions. Unlike GPPs which use Von Nuemann architecture, DSPs use the Harvard architecture. While the Von Nuemann architecture provides a single bus to fetch both instruction and data from memory, the Harvard architecture provides separate buses for instructions and data. This allows instructions and data to be accessed at the same time for faster computation.

Modern DSPs also employ the VLIW (very long instruction word) architecture. VLIW allows the processor to run multiple independent instructions in a single clock cycle, thus increasing parallelism. DSPs with VLIW provides a performance gain of 1.8 to 2.8 times over traditional DSPs without VLIW [13].

The most distinguishing feature of DSPs is the hardware MAC unit. It performs multiply-accumulate operation which forms the basis of essential DSP operations such as FIR fil-

tering, correlations, and FFTs. With specialized MAC blocks, DSPs can perform multiply-accumulate in one or two clock cycles, while it can take multiple clock cycles in GPPs [14].

However, even with dedicated MAC units, data-intensive processes such as Viterbi encoding and decoding are difficult to accelerate. Some DSP chips provide dedicated hardware-based co-processors such as Viterbi-decoder and turbo decoder, but they cannot be customized to specific design needs [15].

Because of special instruction sets and specialized architectural features, DSPs are usually programmed in low level languages such as assembly and C.

FPGA (Field-Programmable Gate Array)

FPGA is a reconfigurable hardware consisting of configurable logic blocks (CLBs) and macro blocks connected via programmable interconnects. Xilinx Spartan-3A DSP FPGA targeted for this thesis consists of four types of macro blocks in addition to CLBs: XtremeDSP DSP48A Slice, Block RAM, Input/Output Blocks, and Digital Clock Manager.

CLBs usually consist of look-up tables, flip-flops, and multiplexers, but they can vary among different FPGA devices. Xilinx Spartan-3A DSP FPGA has four slices in each CLB. A slice consists of two LUTs (Look-Up Tables) , two flip-flops, two multiplexers, and a carry-chain. Although CLBs can be used to implement multipliers and adders, Spartan-3A FPGA provides XtremeDSP DSP48A slices that are dedicated for 18-bit by 18-

bit multiplication and 48-bit accumulation for MAC (multiply-accumulate) operations. The DSP48A slices are ideal for implementing FIR filters which require adder, multiplier, and storage elements. They can be highly pipelined to provide maximum clock frequencies of 250 MHz.

Virtex 5 LX110T, the other FPGA targeted for this thesis, is a more advanced FPGA than Spartan 3A-DSP. Each slice contains four LUTs, four flipflops, multiplexers, and carry logic. The FPGA also has DSP48E slices which supports 25-bit by 18-bit multiplication, 48-bit adder, and accumulator. With maximum pipelining the DSP48E slices can operate at the maximum frequency of 550 MHz [16].

$$y[n] = \sum_{i=0}^{N-1} x[n-i]h[i] \tag{2.1}$$

When performing FIR (Finite Impulse Response) filtering shown in 2.1, DSP48E slices can be arranged in multiple ways to trade-off speed and resource usage. The Single-Multiplier MACC FIR Filter structure uses the least number of DSP48 slices but has the lowest throughput. This structure performs multiply and accumulate on a pair of input sample and filter coefficient at a time. Thus, it takes N clock cycles to produce a single output for a N-tap filter. The Parallel FIR Filter structure uses N DSP48 slices to perform simultaneous multiplication of N coefficients and N respective input samples and accumulate the results. Thus, it takes a single clock cycle to produce a new output sample. Therefore, the Parallel FIR structure has the highest throughput but uses the most num-

ber of DSP48 slices. The Semi-Parallel FIR Filter structure forms a hybrid between the two structures to obtain a higher throughput than the Single-Multiplier structure but uses less slices than the Parallel structure.

In addition to FIR filtering, other signal processing operations lend themselves well to FPGA implementation. For example, taking singular value decomposition (SVD) , shown in 2.2, of a matrix is a common operation in MIMO, radar, or image processing applications.

$$M = U\Sigma V^* \tag{2.2}$$

It is a very computationally intensive process that requires a large number of clock cycles in sequential processors, but the algorithm to produce SVD can be parallelized to be sped up greatly in FPGAs. One well-known systolic array implementation of SVD by Brent can compute SVD in $O(n \log n)$ time using $O(n^2)$ processing elements [17]. AccelChip reports a factor of 50 times increase in speed up with FPGA fixed-point implementation of SVD of an 8x8 matrix compared to floating-point implementation done in a DSP chip TI TMS320C67x [18].

Another advantage of FPGA over other platforms is that it is easy to trade-off between speed and resource usage since the hardware is highly configurable. If the design takes up too much resource, the design can be easily modified to be more resource-efficient at the penalty of decrease in speed. On the other hand, if there is plenty of resource available,

the design can be fully parallelized to achieve maximum speed.

Since it is easily reconfigurable, FPGA can also be a great platform for prototyping a final ASIC design. Once the design is finalized and HDL (Hardware Description Language) is written for FPGA, it is often easy to port to ASIC for final production.

2.1.2 Summary

The three most common SDR platforms have been reviewed. Each platform has its own advantages and disadvantages. GPPs are the easiest to program, test, and verify, but they are often too slow to perform complex computations at high sample rates. With dedicated MAC units, DSPs are better suited for signal processing than GPPs, but they too reach the performance limit as the complexity of the application increases. FPGAs can meet performance requirement needed for high bandwidth applications. However, they are often much harder to program, test, and verify than the other two platforms. Fortunately, recent developments in high-level tools such as Simulink and ImpulseC enable algorithm developers to program FPGAs more easily. Also, tools such as AutoESL's AutoPilot and Synopsys Synphony C Compiler make it easy for DSP software engineers to convert their high-level code in C/C++ to RTL (Register-Transfer-Level). These tools have reportedly been able to achieve comparable resource utilization as manually written RTL code [19]. FPGAs may also be almost as energy efficient as DSPs when considering highly complex signal processing applications. Since a large portion of DSP's circuitry is dedicated to

data transfer, the overall energy consumption per computation of FPGA may be better than that of DSP for some applications [20]. A hybrid approach where complex signal processing is partitioned to FPGA while control-type operation is partitioned to microprocessors maybe the best approach. In this thesis, such hybrid implementations of SDR will be explored.

2.2 ZigBee

2.2.1 Overview

ZigBee is a LR-WPAN (Low Rate-Wireless Personal Area Network) standard commonly used for home control applications and wireless sensor networks. The ZigBee standard defines the application, security, and network layers of the protocol stack, while the physical (PHY) and medium access control (MAC) layers of the standard are based on IEEE 802.15.4. The standard specifies the maximum data rate to be 250kbps and the maximum range to be 100m. Because of its low data rate and simple architecture, it is low cost and consumes less power compared to other WPAN protocols such as Bluetooth. ZigBee devices can last as long as five years on a pair of AA batteries [21].

The network layer of ZigBee supports different types of network topologies such as star, tree, and mesh networks. It supports ad-hoc networking where routes are automatically discovered as new nodes join the network. It has self-healing ability which allows nodes

Table 2.1: ZigBee Channels in 2.4 GHz ISM Band

Channels	11	12	13	14	15	16	17	18
Center Frequencies (MHz)	2405	2410	2415	2420	2425	2430	2435	2440
Channels	19	20	21	22	23	24	25	26
Center Frequencies (MHz)	2445	2450	2455	2460	2465	2470	2475	2480

to discover new routes if intermediary nodes in the route fail. The MAC layer uses CSMA-CA (Carrier Sense Multiple Access Collision Avoidance) to avoid packet collisions among multiple nodes.

This thesis implements the PHY layer as specified in IEEE 802.15.4 and a MAC-like control layer, but leaves out the higher MAC, network, and application layers.

2.2.2 IEEE 802.15.4

The IEEE 802.15 is a working group formed to define Wireless Personal Area Network (WPAN) standards. There are seven task groups in the working group. The ZigBee protocol is based on the IEEE 802.15.4 task group which defines low rate WPAN of 20 kbps, 40 kbps, and 250 kbps. The standard developed by the IEEE 802.15.4 task group allows for 16 channels in the 2.4 GHz ISM band, 10 channels in the 915 MHz band, and one channel in the 868MHz band. The center frequencies used for the 2.4 GHz band are shown in Table 2.1. The primary target applications for the standard are in home control, sensors, interactive toys, smart badges, and remote controls. The standard serves as PHY and MAC layers for the ZigBee standard.

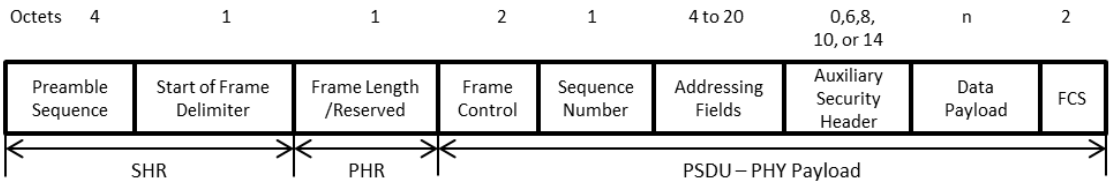

Figure 2.2: IEEE 802.15.4 Data Frame [1]

IEEE 802.15.4 Frame Structure

There are four different types of frames defined at the MAC sub-layer. They include beacon frame, data frame, acknowledgment frame, and MAC command frame. Beacon frames are used to synchronize the nodes in the network for slotted CSMA-CA. Data frames are used for all transfers of data between nodes. Acknowledgment frames are used for confirmation of successful reception of data or MAC command frames. If the transmitter does not receive the acknowledgment frame, it will retransmit. MAC command frames are used for transmitting low-level MAC commands. The data frame shown in Figure 2.2 was implemented in this thesis.

The MAC sub-layer frame is embedded into the PSDU (Physical Layer Service Data Unit) of the PHY layer. The PSDU is prefixed with SHR (synchronization header) and PHR (PHY header) to be transmitted over the air. Within SHR, Preamble Sequence is used by the receiver to detect and synchronize to the received packets. The preamble is simply all zero bits for all PHYs except for the ASK PHY. For the 2.4 GHz O-QPSK , the preamble is 4 octets of zeros, as shown in Table 2.2, equivalent to 8 symbols, and it is 128 us long [1].

The SFD (Start of Frame Delimiter) field indicates the start of PHR. For 2.4 GHz O-QPSK,

Table 2.2: Preamble Field [1]

Bits 0:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.3: SFD Field [1]

Bits 0:	1	2	3	4	5	6	7
1	1	1	0	0	1	0	1

the SFD field is 8 bits long as shown in Table 2.3.

The Frame Length field is a 7 bits long field indicating the total number of octets in the PSDU. The valid frame length values are from 9 to *aMaxPHYPacketSize* of 127. The Frame Length field is followed by a reserved bit.

2450 MHz PHY specifications

The 2450 MHz PHY specification of IEEE 802.15.4 supports data rate of up to 250kb/s. It uses DS-SS (Direct Sequence Spread Spectrum) with O-QPSK (Offset-QPSK) for modulation. Each O-QPSK symbol is one of 16 quasi-orthogonal pseudo-random noise (PN) sequences. Each symbol is 32-chip long and corresponds to one of 16 possible combinations of four information bits. The mapping of 4-bit symbol to 32-chip sequence is shown in Figure 2.4.

Table 2.4: Symbol to Chip Mapping [1]

Data Symbol (b_0 b_1 ... b_3)	Chip values (c_0 c_1 ... c_{31})
0 0 0 0	1 1 0 1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 0 1 1 1 0
1 0 0 0	1 1 1 0 1 1 0 1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 0
0 1 0 0	0 0 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1 0 0 1 0
1 1 0 0	0 0 1 0 0 0 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1
0 0 1 0	0 1 0 1 0 0 1 0 0 0 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 0 0 0 0 1 1
1 0 1 0	0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 0 1 1 1 0 1 1 0 1 1 0 0 1 1 1 0 0
0 1 1 0	1 1 0 0 0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 0 1 1 1 0 1 1 0 1 1 0 0 1
1 1 1 0	1 0 0 1 1 1 0 0 0 0 1 1 0 1 0 1 0 0 1 0 0 0 1 0 1 1 1 0 1 1 0 1
0 0 0 1	1 0 0 0 1 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 1
1 0 0 1	1 0 1 1 1 0 0 0 1 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 1 1 1 0 1 1 1
0 1 0 1	0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 1 1 1
1 1 0 1	0 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0
0 0 1 1	0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 1 0 0 1 0 1 1 0
1 0 1 1	0 1 1 0 0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0 1 0 0 1
0 1 1 1	1 0 0 1 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0 1 1 0 0
1 1 1 1	1 1 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 1 1 0 0 0

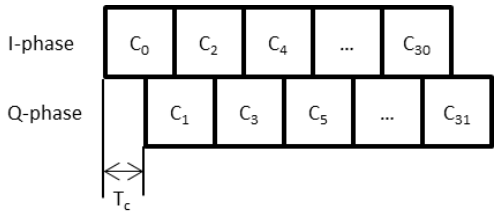

Figure 2.3: O-QPSK Chip Offset [1]

Chip sequences are modulated onto the carrier using O-QPSK with half-sine pulse shaping. With the half-sine pulse shaping, the O-QPSK modulation can be treated as MSK (Minimum-Shift Keying) with a modulation index of h=0.5, which allows for a simple MSK demodulator to be used at the receiver. The biggest advantage of O-QPSK is that relatively non-linear amplifier can be used since the envelope of the signal remains constant.

Figure 2.3 shows how O-QPSK signal is generated from the chip sequences. The even-indexed chips are modulated onto the in-phase carrier, and odd-indexed chips are modulated onto the quadrature-phase carrier. The Q-phase is delayed by T_c with respect to I-phase chips to create O-QPSK from QPSK.

2.3 Polyphase Filter-Bank Channelizer

A polyphase filter-bank channelizer is an efficient signal processing technique used to divide up a wideband spectrum into a number of smaller evenly spaced bands. Its structure consists of a commutator, a polyphase filter bank, and a DFT block as shown in Figure 2.4. The polyphase filterbank consists of M filters created from a lowpass filter where M

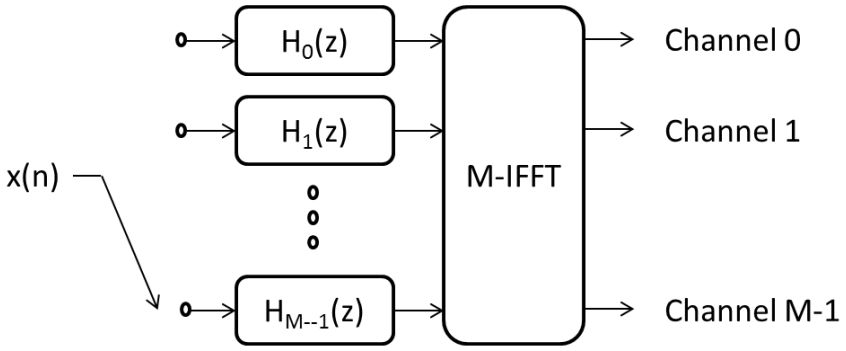

Figure 2.4: Polyphase Channelizer

is the number of channels.

The lowpass filter used to create the polyphase bank can be written as a one-dimensional array of coefficients as the following in the z-domain.

$$H(z) = \sum_{n=0}^{N-1} h[n]z^{-n} \tag{2.3}$$
$$= h[0] + h[1]z^{-1} + h[2]z^{-2} + \ldots + h[N-1]z^{-(N-1)}$$

The coefficients can be rearranged as a two-dimensional array where the number of rows is M as shown in Equation 2.4.

$$\begin{aligned} H(z) = {} & h[0] + h[M+0]z^{-M} + h[2M+0]z^{-2M} + \ldots \\ & + h[1]z^{-1} + h[M+1]z^{-(M+1)} + h[2M+1]z^{-(2M+1)} + \ldots \\ & + h[2]z^{-2} + h[M+2]z^{-(M+2)} + h[2M+2]z^{-(2M+2)} + \ldots \\ & \qquad \ldots \\ & + h[M-1]z^{-(M-1)} + h[2M-1]z^{-(2M-1)} + h[3M-1]z^{-(3M-1)} + \ldots \end{aligned} \tag{2.4}$$

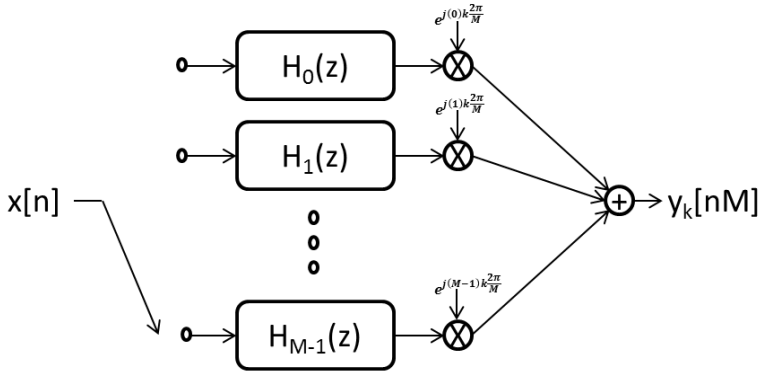

Figure 2.5: Single Channel M-to-1 Resampler

Each row of Equation 2.4 can be grouped together so that $H(z)$ can be re-written in the following way.

$$\begin{aligned} H(z) = H_0(z^M) & \\ + z^{-1}H_1(z^M) & \\ + z^{-2}H_2(z^M) & \\ + \ldots & \\ + z^{-(M-1)}H_{M-1}(z^M) & \end{aligned} \tag{2.5}$$

where $H_0(z^M)$ is the terms in the first row of Equation 2.4, $z^{-1}H_1(z^M)$ is the terms in the second row, and so on.

The M terms in Equation 2.5 correspond to the M filters in the polyphase filter bank. The commutator act as the delay factors in front of the M terms. Given M channels, Figure 2.5 shows the resampler structure where a channel centered at k^{th} center frequency can

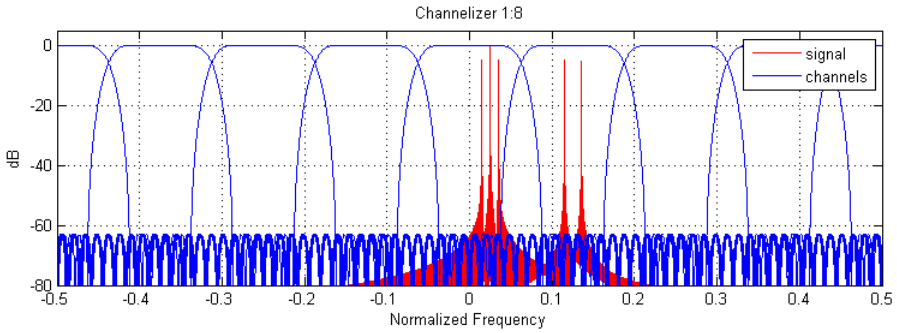

Figure 2.6: Input to Eight-Channel Channelizer

be extracted. The output of the k^{th} channel can be written as Equation 2.6

$$y_k[nM] = \sum_{r=0}^{M-1} y_r[nM] e^{j2\pi \frac{k}{M} r} \tag{2.6}$$

where $y_r[nM]$ is output of r^{th} filter in the filter bank. This process is similar to IDFT as defined in Equation 2.7. The IDFT block effectively calculates Equation 2.6 for $k = 0, 1, ..., M-1$.

$$X[k] = \sum_{n=0}^{M-1} x_n e^{j2\pi \frac{k}{M} n} \tag{2.7}$$

The output of IDFT is then the M channels evenly spaced and sampled at $\frac{Fs}{M}$.

As an illustration, Figure 2.6 shows the positions of eight channels and two input signals occupying two different channels. Figure 2.7 shows the output of each channel. As expected *Channel 0* and *Channel 1* contain the two input signals, while the rest of the channels are empty.

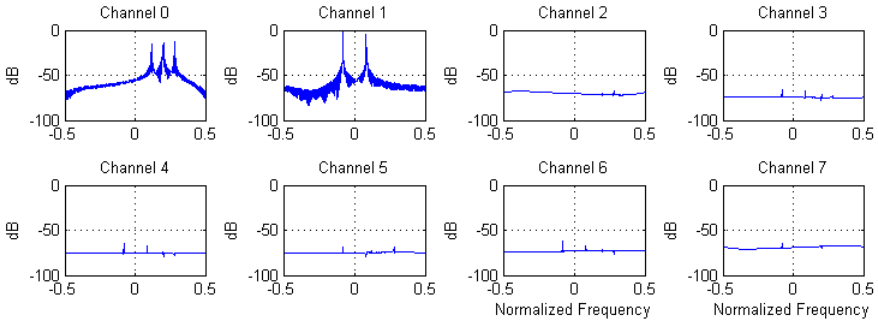

Figure 2.7: Output of Each Channel

Table 2.5: USRP N210 Specification

FPGA	Xilinx Spartan 3A-DSP3400
ADC	14-bits 100 MS/s
DAC	16-bits 400 MS/s
RF Bandwidth	50 MHz of instantaneous RF bandwidth in 8-bit mode 25 MHz of instantaneous RF bandwidth in 16-bit mode
Connectivity	Gigabit Ethernet Interface

2.4 USRP N210

The USRP (Universal Software Radio Peripheral) is a family of software radio platforms developed by Ettus Research LLC. Table 2.5 shows the summary of specifications of USRP N210 [22].

The USRP2, the predecessor of the USRP N210, has almost identical features as the USRP N210, except that it has a smaller Xilinx Spartan 3-2000 FPGA. Table 2.6 compares the FPGAs on USRP N210 and USRP2.

Table 2.6: FPGAs of USRP N210 (3A3400) and USRP2 (3S2000)

FPGA	System Gates	Equivalent Logic Cells	Total Slices	Distributed RAM Bits	Block RAM Bits	DSP48As	Dedicated Multipliers	DCMs	Maximum USER I/O
3A3400	3400K	53,712	23,872	373K	2258K	126	N/A	8	469
3S2000	2000K	46,080	20,480	320K	720K	N/A	40	4	565

2.4.1 Signal Processing in FPGA of USRP N210

The FPGA is configured as a SOC (System On Chip) where different IP cores are connected via the Wishbone bus. Figure 2.8 shows the interconnection between different modules inside the FPGA. The Wishbone interconnect has a 32-bt soft-core processor called ZPU as the master. All other fourteen blocks connected to the Wishbone interconnect are slaves. The slave block of most interest is the *Buffer Pool* block. It routes received samples from *DSP_CORE_RX0* and *DSP_CORE_RX1* to Ethernet MAC for transmission to the host. It also receives frames containing samples to be transmitted over the air from the host and stores into *EXT_FIFO*. The *VITA_TX_CHAIN* then reads the frames from *EXT_FIFO* and strips the VITA headers. Once the headers are removed, samples are sent to DAC for transmission.

Within the *VITA_TX_CHAIN* block interpolation is done so that the sample rate matches the sample rate of the DAC. Figure 2.9 shows how samples from the host are interpolated before being sent to the DAC for transmission. The 32-bit input samples from the host

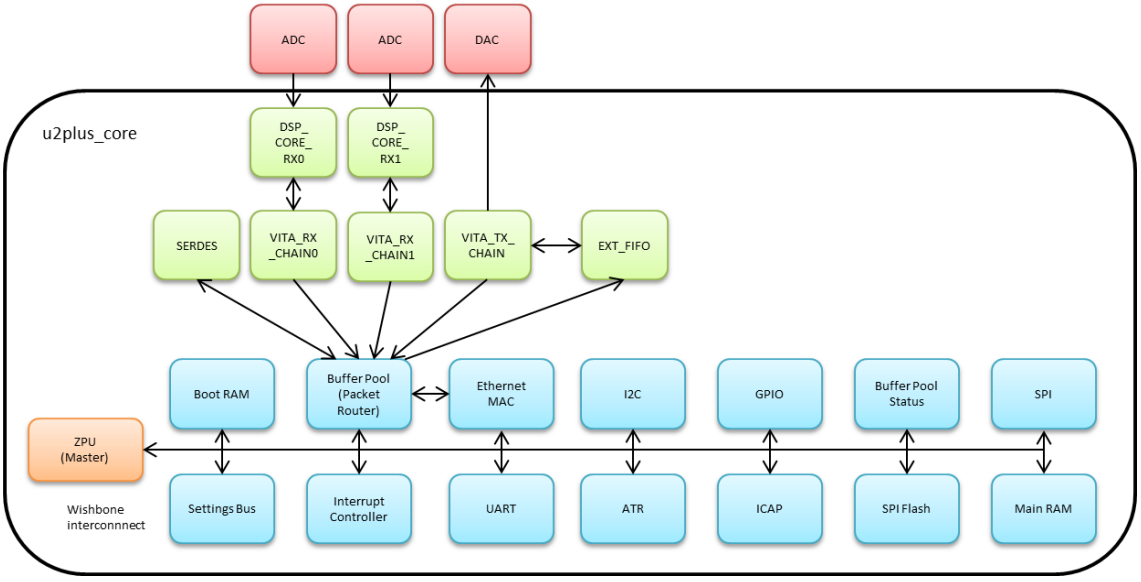

Figure 2.8: SOC in USRP's FPGA

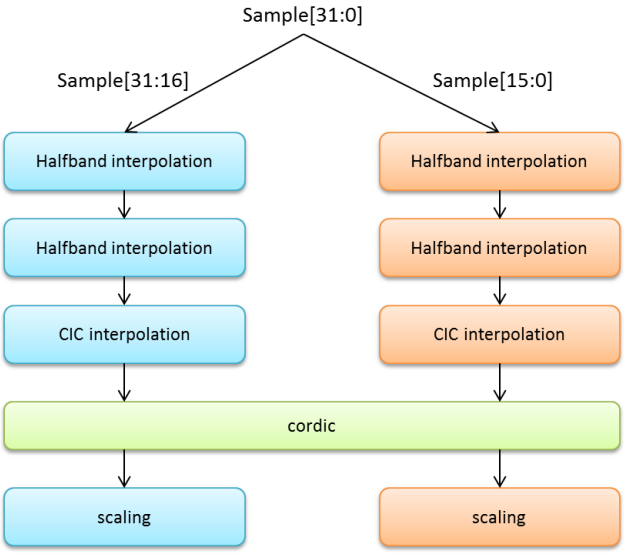

Figure 2.9: Transmit Signal Processing Chain

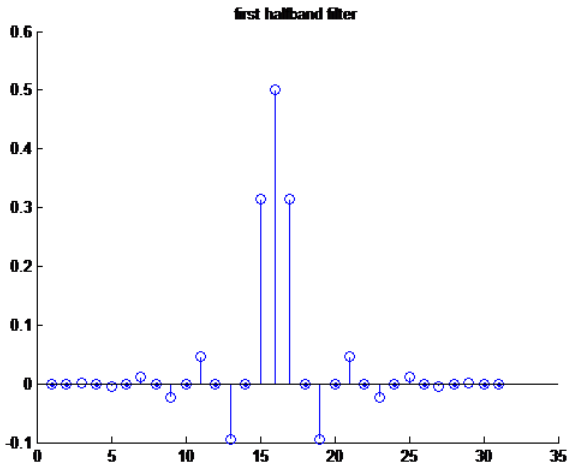

Figure 2.10: Time-Domain Plot of First Half-Band Filter

are divided into 16-bit in-phase samples and 16-bit quadrature-phase samples. The first half-band filter interpolates the samples by a factor of two if the filter is enabled. The second half-band filter interpolates the signal again by another factor of two if enabled. The CIC interpolator can interpolate the signal up to a factor of 128. Thus, the maximum interpolation rate within the FPGA is 512. The 24-stage CORDIC block rotates the signal to give frequency offset before being up-converted by the daughterboard.

The first half-band filter shown in Figure 2.10 has 31 coefficients generated by *myfilt = halfgen4(.7/4,8)* in MATLAB, while the second half-band filter shown in Figure 2.11 only has 7 coefficients generated by *myfilt = halfgen4(.75/8,2)*. The first half-band filter is enabled when the interpolation factor is a multiple of two. Both half-band filters are enabled if the interpolation factor is a multiple of four. Figure 2.12, shows the frequency responses of the two half-band filters.

Figure 2.13 shows the four-stage CIC interpolator used in the USRP N210. The CIC filter

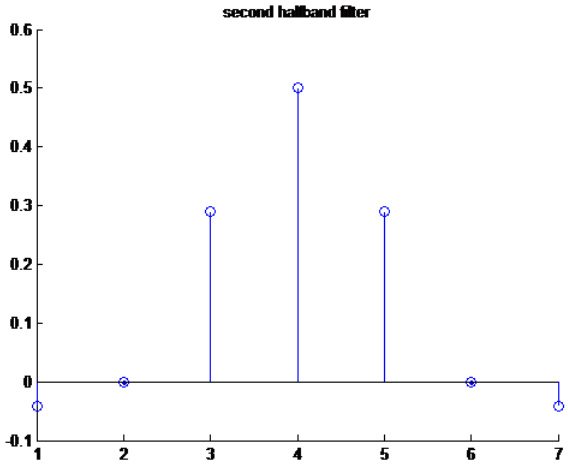

Figure 2.11: Time-Domain Plot of Second Half-Band Filter

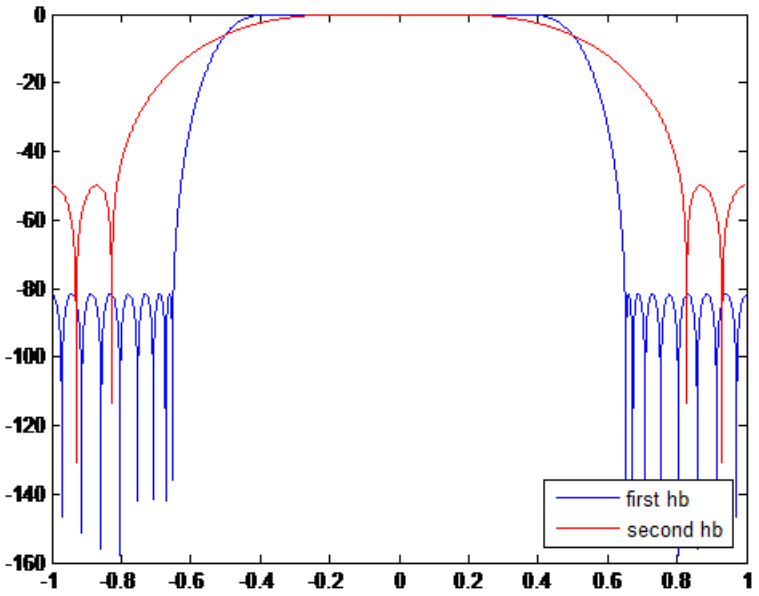

Figure 2.12: Frequency Response of Two Halfband Filters

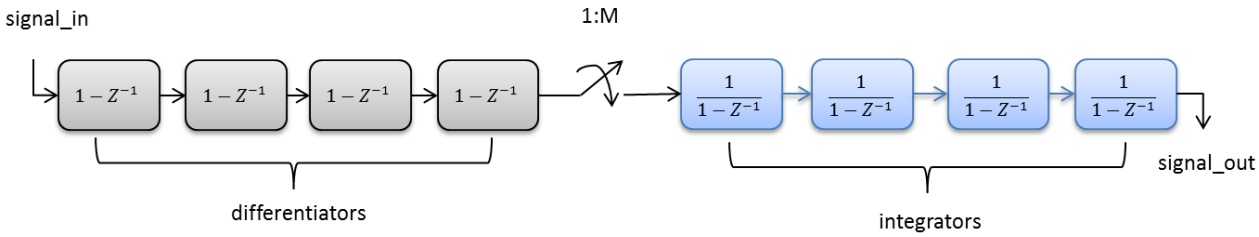

Figure 2.13: CIC Interpolator in USRP N210

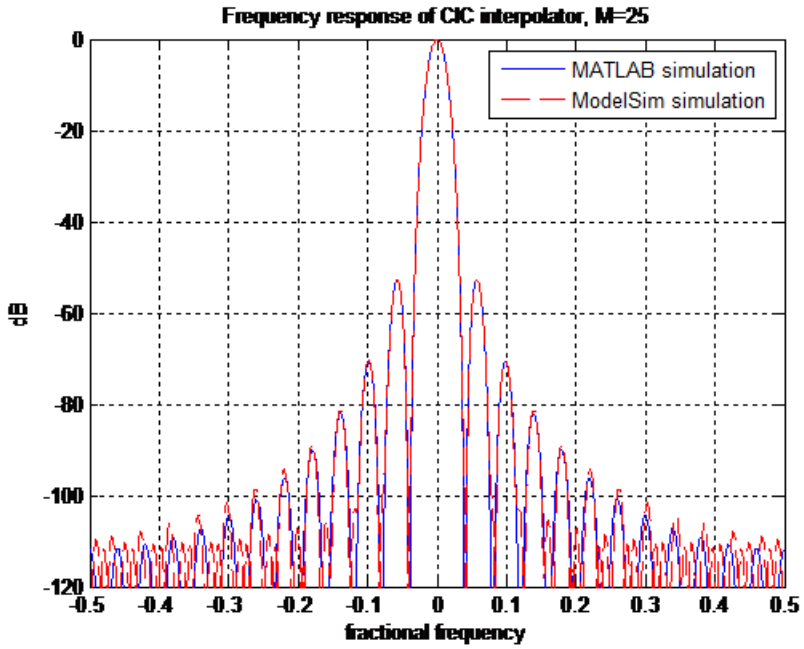

Figure 2.14: Frequency Response of CIC Interpolator

has the Hogenauer structure, where all the differentiators are on one side and all the integrators are on the other side of the resampling switch [23]. The interpolation factor of the CIC in USRP N210 can be set as high as M=128.

For the implementation of IEEE 802.15.4 PHY, the sampling rate of 4 MHz was needed. Since the clock in the FPGA of USRP operates at 100 MHz, the interpolation factor M was set to be 25. Figure 2.14 shows the frequency response of the CIC filter when M=25.

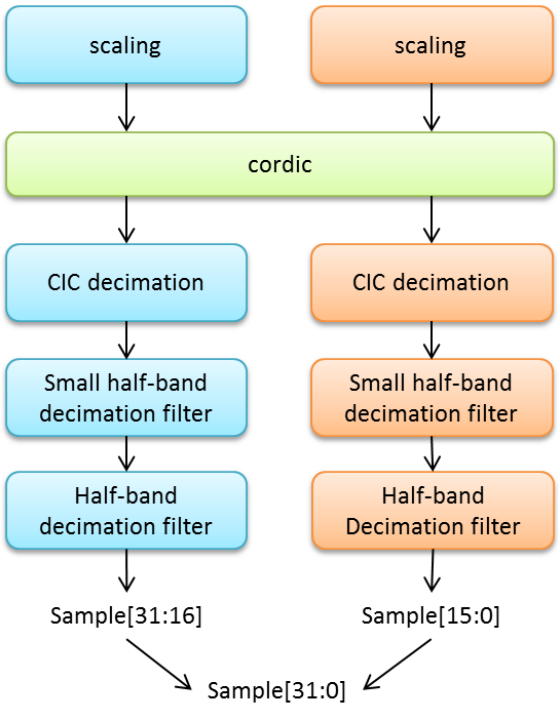

Figure 2.15: Receiver Signal Processing Chain

In the receiver chain, the reverse of the transmitter chain happens. The *DSP_CORE_RX* core in the SOC receives the samples from the ADC. Inside the *DSP_CORE_RX* core, the CORDIC block performs digital down-conversion (DDC). After the DDC is performed, the samples are decimated with the two half-band decimators and the CIC filter. After decimation, the samples are sent to the VRT (VITA Radio Transport) core for packetization. The packets are then sent to the Ethernet MAC to be transported to the host computer.

Chapter 3

Methodology and Implementation

For this thesis, two major signal processing blocks, IEEE 802.15.4 PHY and a polyphase filter-bank channelizer, have been implemented in Spartan 3A-DSP FPGA of USRP N210 and Virtex 5 FPGA, respectively. This section illustrates the steps taken to simulate and implement the blocks.

3.1 IEEE 802.15.4 PHY on FPGA

There are two paths explored in implementing the IEEE 802.15.4 PHY, as shown in Figure 3.1. The first path involves Simulink modeling, while the second one does not. In the first path, the floating-point simulation is performed in MATLAB which is then converted to a Simulink model. At this stage, the Simulink model is still in floating-point. In order to

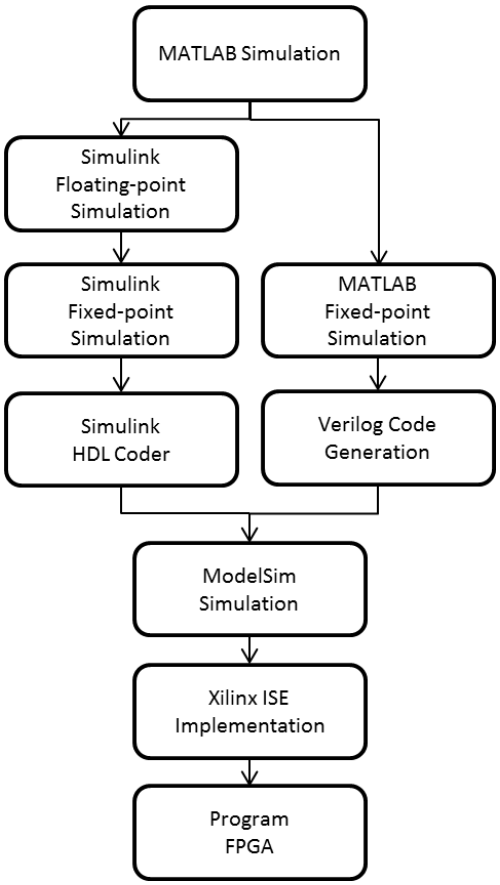

Figure 3.1: Two Paths Explored in FPGA Implementation

generate Verilog code, the model is converted to a fixed-point model. Next, the Simulink HDL Coder is used to automatically generate Verilog or VHDL code from the fixed-point model.

The second path is to perform both floating-point and fixed-point simulations in MATLAB. The Verilog code is then written manually based on the MATLAB simulation, skipping the Simulink model and automatic generation of the Verilog code entirely. Once the ModelSim simulation of the Verilog code is finished, it is compiled and implemented on the FPGA. For the final implementation, the second path was taken.

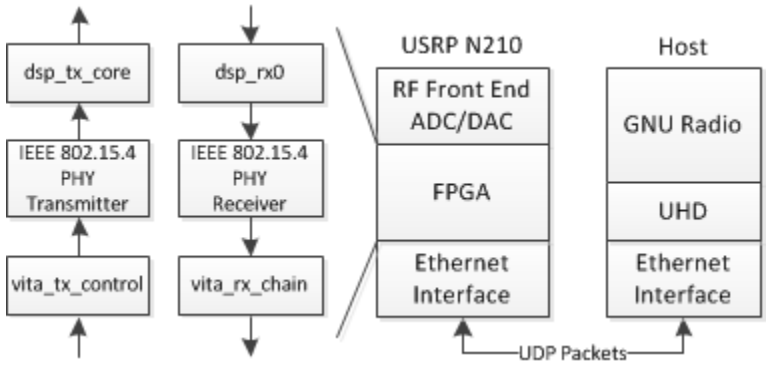

Figure 3.2: Overall Setup of IEEE 802.15.4 PHY on FPGA

3.1.1 Configuration

Figure 3.2 shows the overall setup of the system. In normal receive mode, USRP N210 sends the data samples to GNU Radio after down-conversion and some filtering. However, when the IEEE 802.15.4 PHY receiver is enabled, the receiver resides between the DDC and Ethernet interface and intercepts the samples and processes them before sending the result to the Ethernet interface. Likewise, in transmit mode, USRP N210 normally sends samples from GNU Radio to DUC for transmission. However, when the IEEE 802.15.4 PHY transmitter is enabled, it intercepts the samples from GNU Radio and modulates the samples before sending them to DUC.

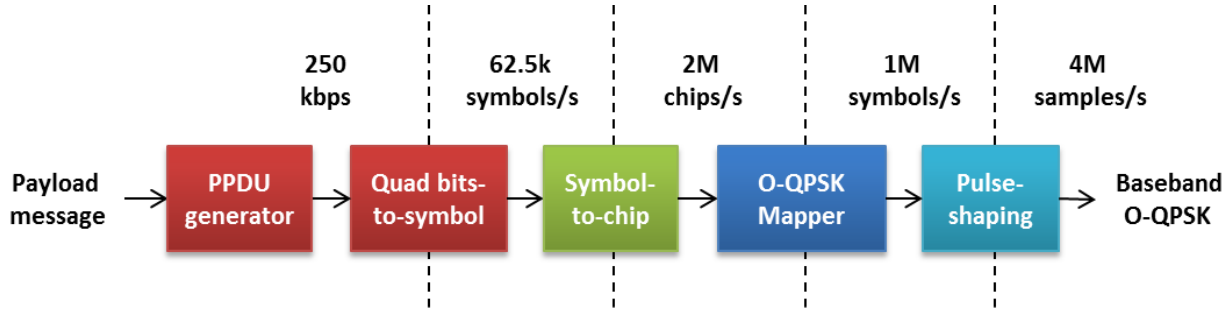

Figure 3.3: Overall Transmitter Chain

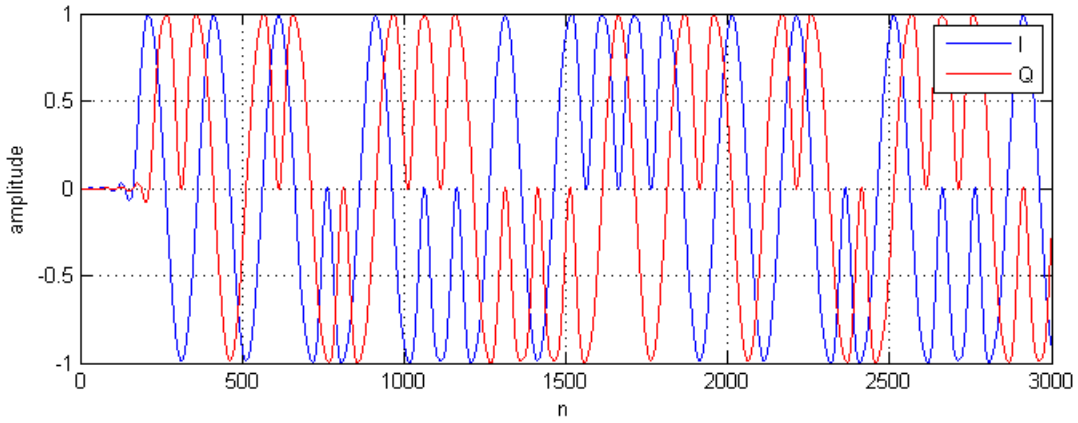

Figure 3.4: Simulation of O-QPSK at Baseband

3.1.2 Transmitter

MATLAB Simulation

The first step in implementation of the IEEE 802.15.4 transmitter is to simulate the signal processing chain, shown in Figure 3.3, in MATLAB. IEEE 802.15.4 frames are generated using the PPDU (PHY Protocol Data Unit) generator function. The PPDU bits are then grouped as quad-bits and mapped to symbols. The symbols are spread to give overall spreading factor of eight. The final output of the transmitter chain is the baseband I and Q as shown in Figure 3.4. The baseband signal is an O-QPSK signal where the quadrature

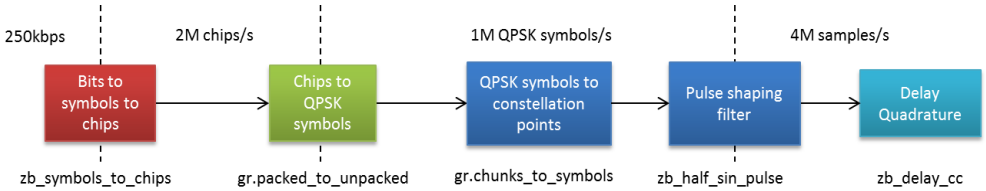

Figure 3.5: Transmitter Chain

component is delayed by half the symbol period.

RTL Implementation/Simulation

After simulation in MATLAB and/or Simulink, the algorithm must be converted to Verilog. The signal chain is modularized so that blocks can be mapped onto the existing GNU Radio implementation of ZigBee Radio [12]. Some of the blocks such as *gr.packed_to_unpacked* and *gr.chunks_to_symbols* are based on the basic blocks provided by the GNU Radio library written in C++. The blocks developed for the thesis can be used in GReasy as basic building blocks for other complex signal processing applications.

Figure 3.5 shows the RTL blocks in the transmitter signal processing chain. The *zb_symbols_to_chips* block breaks the bytes from the incoming payload into two separate four-bit symbols, each of which are then mapped to one of the sixteen chip sequences as was shown in Figure 2.4.

For example, if the current byte is 0x14, it is broken into two four-bit symbols, 0x1 and 0x4. The first symbol 0x1 is mapped to the chip 3986437410_{10}, and the second symbol 0x4 is mapped to the chip 1378802115_{10}. The clock rate increases from 250kbps to 2Mchips/s

at the output.

The *gr.packed_to_unpacked* block breaks the 32-chip long chip sequences into 16 two-chip chunks. For example, the chip 1378802115_{10} ($01010010000101110110110011100000 11_2$), is broken into chunks of 01, 01, 00, 10, 00, 10, 11, 10, 11, 01, 10, 01, 11, 00, 00, and 11. The clock rate is decreased by a factor of two from 2Mchips/s to 1Mchunks/s, or 1 M QPSK symbols/s at the output.

The *gr.chunks_to_symbols* block maps each chunk to a point on the QPSK constellation. Using gray coding, the chunk 00 is mapped to -1-j, 01 to -1+j, 10 to 1-j, and 11 to 1+j. The clock rate stays the same at the output.

The *zb.half_sine_pulse* block performs half-sine pulse-shaping of the QPSK symbols. The pulse-shaping is done by a simple mapping of the constellations to a positive half-sine pulse or a negative half-sine pulse depending on the data. For example, when the in-phase or the quadrature-phase is a '1', it is mapped to the positive pulse. If it is a '-1', it is mapped to a negative half-sine pulse. The pulses used are up-sampled by a factor of four. Therefore, the clock rate is increased by a factor of four at the output to be 4 M samples/s.

Finally, in order to generate O-QPSK signal from the QPSK signal, the quadrature component is delayed by two samples by the *zb.delay_cc block*.

Figure 3.6 shows the I and Q signals generated by the RTL blocks. The signals *bb_i* and *bb_q* generated by the IEEE 802.15.4 transmitter module are taken from the output of the *zb.delay_cc* block. The signals *i_interp* and *q_interp* are interpolated versions of *bb_i* and

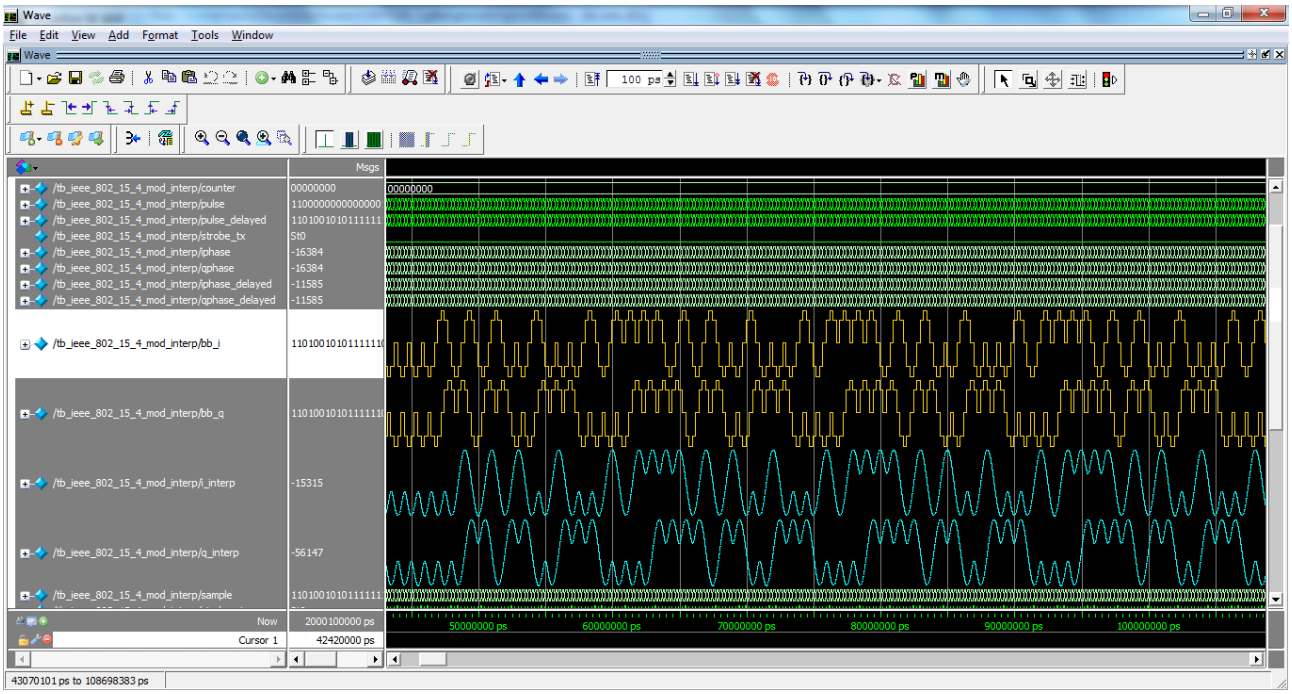

Figure 3.6: ModelSim simulation of transmitted signal

bb_q. The interpolation is done inside the USRP N210's FPGA CIC interpolator blocks. The interpolated signal is up-converted and sent to the DAC for transmission over the air.

FPGA Implementation

Once the transmitter module is thoroughly tested in the RTL simulation stage, it is integrated into the USRP N210's FPGA code. Figure 3.7 shows the modification done to the USRP FPGA code to integrate the IEEE 802.15.4 module. In normal transmit mode, GNU Radio sends a UDP packet containing samples to the USRP. The Ethernet MAC and *packet_router* inside the USRP's FPGA decodes the UDP packet and relays the payload to

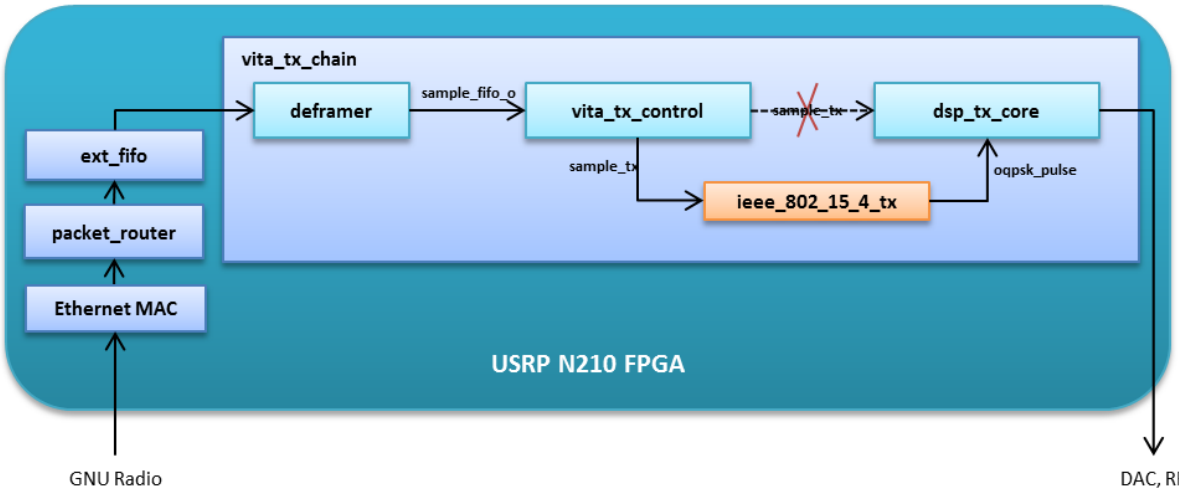

Figure 3.7: Transmitter module inside USRP's FPGA

vita_tx_chain. The VRT headers are then removed by *deframer* and *vita_tx_control*, and the extracted samples are sent to *dsp_tx_core* where they are interpolated and sent to the DAC for transmission. The FPGA code was modified so that the samples from *vita_tx_control* are sent to the IEEE 802.15.4 transmitter instead of *dsp_tx_core*. The output of the IEEE 802.15.4 transmitter is then sent to *dsp_tx_core* for transmission. Using Chipsope, the actual signals inside the FPGA can be probed. Figure 3.8 shows the half-sine waves, generated by the IEEE 802.15.4 module, interpolated by the CIC filter in *dsp_tx_core*.

3.1.3 Receiver

MATLAB Simulation

Like the transmitter, the first step in implementation of the receiver is to simulate the algorithm in MATLAB. The overall receiver signal processing chain is shown in Figure

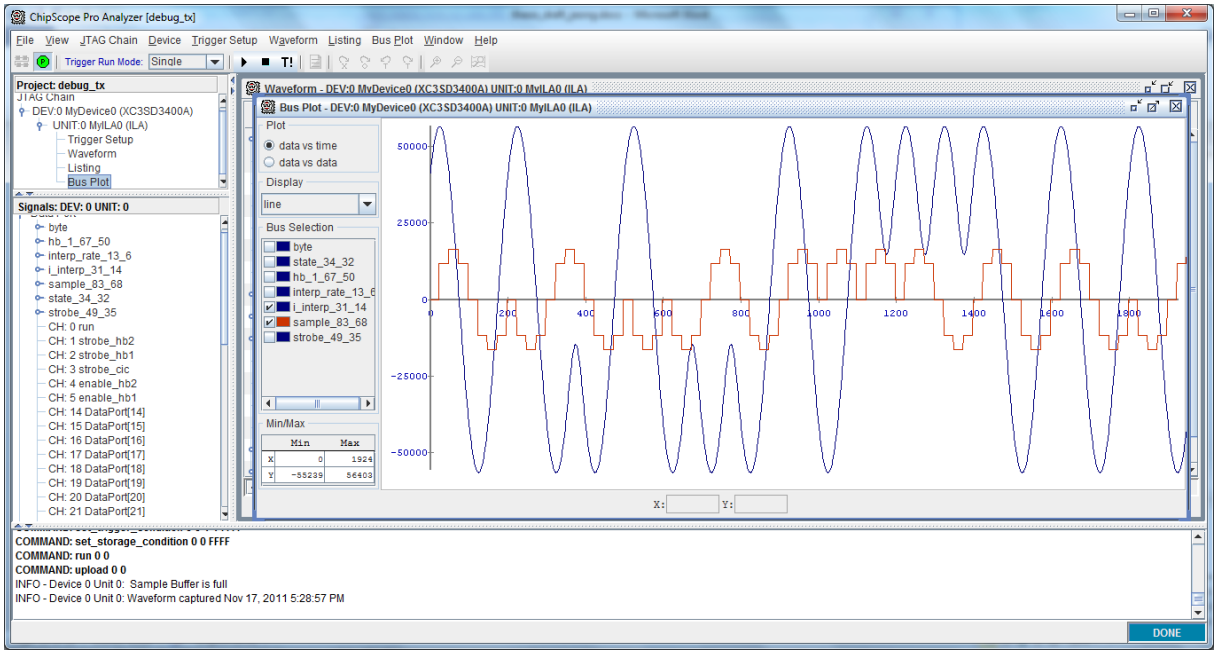

Figure 3.8: Chipscope showing interpolated in-phase waveform of O-QPSK

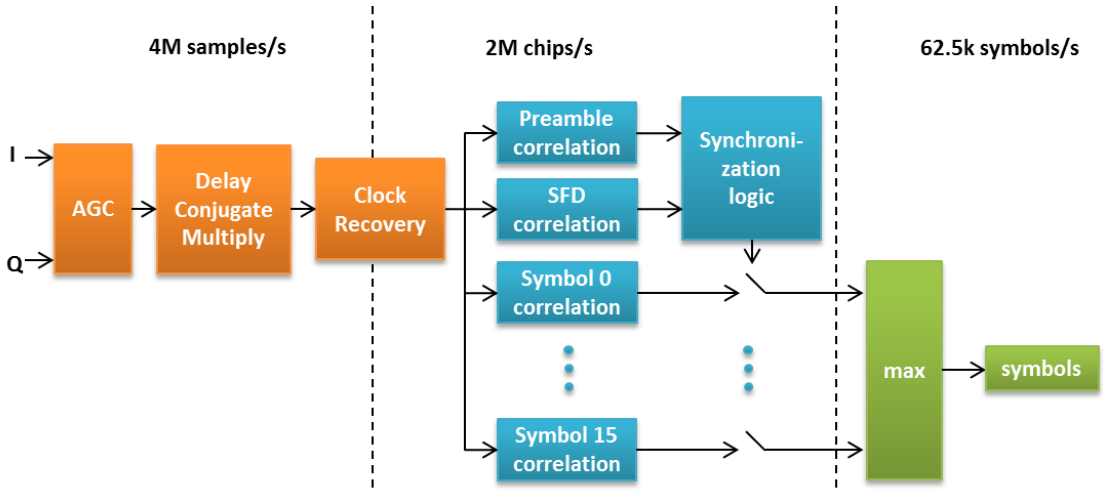

Figure 3.9: Overall receiver algorithm

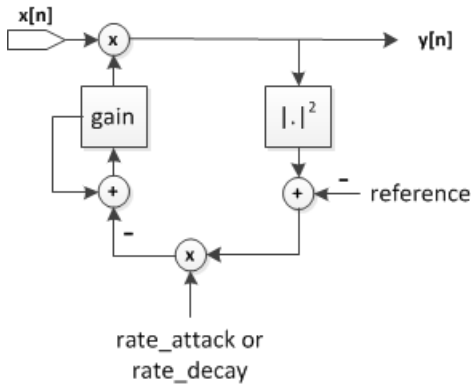

Figure 3.10: AGC Structure

3.9.

Automatic Gain Control

The automatic gain control (AGC) block is needed at the input of the receiver to utilize as much dynamic range as possible. When the magnitude of the input samples is small, the inverse tangent function block in FPGA implementation is not able to produce accurate outputs as shown in the later section. The structure of the AGC block based on the GNU Radio implementation *AGC2* is shown in Figure 3.10.

Demodulation of O-QPSK as MSK

The O-QPSK modulated waveform can be demodulated as MSK when half-sine pulse shaping is used, as is the case with IEEE 802.15.4 [24]. An MSK waveform can be demodulated using the delay-conjugate-multiply shown in Figure 3.11, since the output of the delay-conjugate-multiply block is the phase change between two consecutive sam-

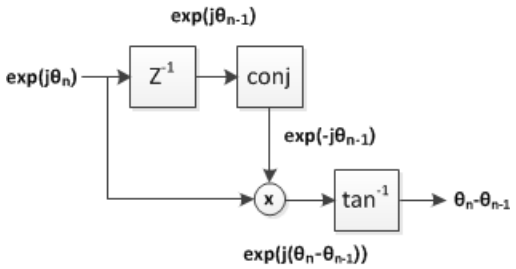

Figure 3.11: Delay-conjugate-multiply

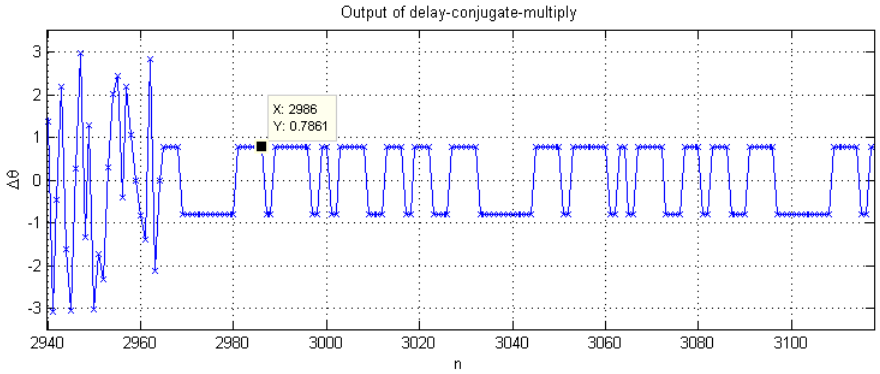

Figure 3.12: Output of delay-conjugate-multiply

ples. The positive phase change shows that $f_c + \frac{1}{4T_b}$ was sent while negative phase change shows that $f_c - \frac{1}{4T_b}$ was sent. The output of delay-conjugate-multiply using simulated input data is shown in Figure 3.12.

Clock Recovery

At the output of the delay-conjugate-multiply block, the sample rate is at 4 MHz. Since the chip rate is at 2M chips/sec, two samples are available for each chip. The clock recovery block based on Mueller and Muller algorithm is used to find the optimal sample instance of the chip [25]. Figure 3.14 shows the chips sampled without clock recovery and the chips sampled with clock recovery. The structure of the clock recovery block is shown in

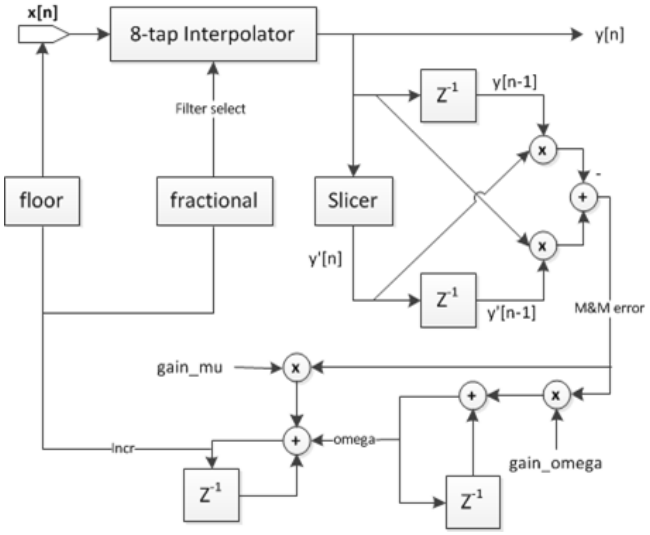

Figure 3.13: Clock Recovery Block Structure

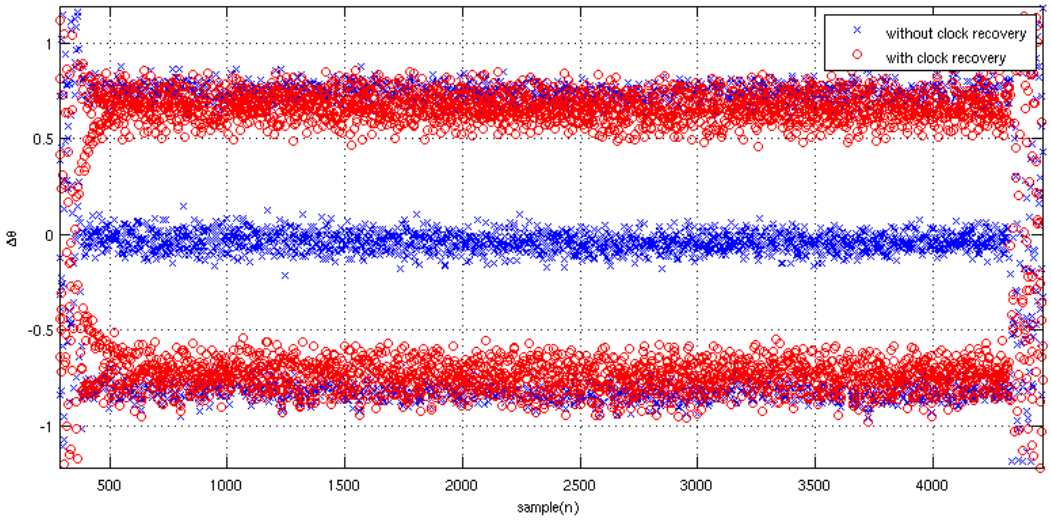

Figure 3.14: Comparison of Sampled Chips With and Without Clock Recovery

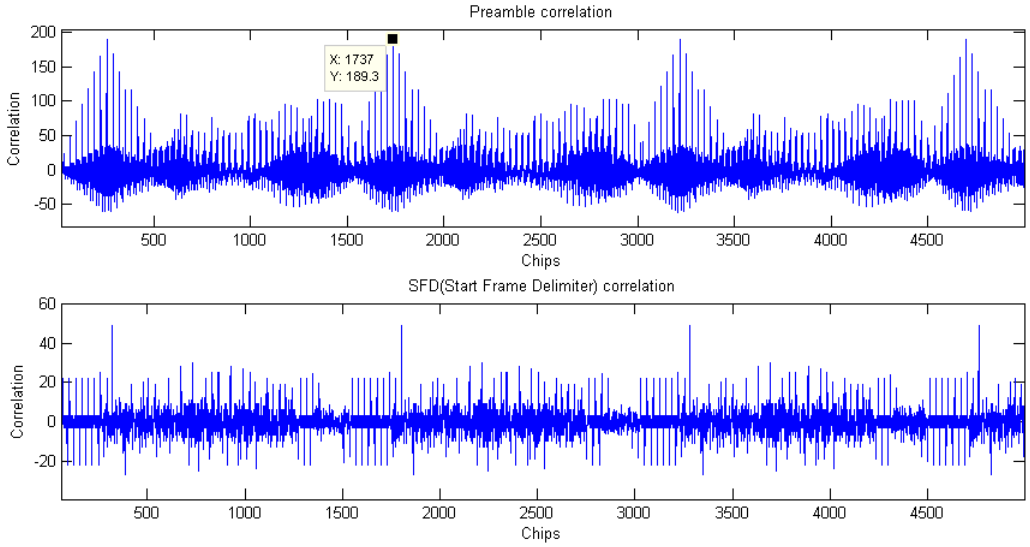

Figure 3.15: Preamble and SFD correlations

Figure 3.13.

Preamble and SFD correlation thresholds

Figure 3.15 shows the preamble and SFD correlations for four consecutive PPDU frames. Once the preamble and SFD correlations are found, they have to be compared to some thresholds to determine if an IEEE 802.15.4 packet was actually present or not. If the correlation values, $\lambda_{preamble}$ and λ_{SFD}, are greater than some thresholds $\tau_{preamble}$ and τ_{SFD}, then it is determined that a valid packet is present and the incoming samples should be demodulated. If the correlation values are lower than the thresholds, it is determined that there is no valid packet present. A simple binary hypothesis testing can be used to

determine the thresholds. Given the hypothesis,

$$H_0 = Packet\ not\ present \tag{3.1}$$

$$H_1 = Packet\ present$$

The thresholds $\tau_{preamble}$ and τ_{SFD} must be such that the false alarm rate $P_{fa} = P[\lambda_{preamble} > \tau_{preamble}|H_0]P[\lambda_{SFD} > \tau_{SFD}|H_0]$ is less than an acceptable level.

The maximum values for the thresholds can be found theoretically by analyzing the properties of MSK waveform. The MSK waveform can be written as the following.

$$s(t) = \sqrt{\frac{2E_b}{T_b}}cos(2\pi f_c t + \theta(t)) \tag{3.2}$$

$$\theta(t) = \theta(0) + \frac{\pi}{2T_b},\ \text{'1' was sent}$$

$$\theta(t) = \theta(0) - \frac{\pi}{2T_b},\ \text{'0' was sent}$$

Depending on whether '1' or '0' was sent, the carrier phase is rotated by $\pm\frac{\pi}{2}$ over the symbol duration T_b. Therefore, when two samples per chip are available, the maximum absolute value of output of the delay-conjugate-multiply is half of $\frac{\pi}{2}$, or $\frac{\pi}{4}$.

Since the maximum absolute value of output of the delay-conjugate-multiply is $\frac{\pi}{4}$, the theoretical maximum value of preamble correlation is $N_{Preamble\ chips}(\frac{\pi}{4}) = 201.06$ and SFD correlation $N_{SFD\ chips}(\frac{\pi}{4}) = 50.27$, where $N_{Preamble\ chips} = 256$ and $N_{Preamble\ chips} = 64$. However, because of noise the correlation values will never reach the theoretical max-

imum values. Therefore, the thresholds of preamble and SFD correlations have to be somewhat lower for the packets to be detected. The actual threshold values used for implementation are determined experimentally as shown in later sections.

Symbol Correlations

Once the SFD is found, the receiver must decide which symbol was sent by the transmitter. The decision of which symbol was sent is made by picking the symbol that gives the highest correlation value. In the GNU Radio implementation, the correlation values are found by first slicing the chips at the output of the delay-conjugate-multiply module, which is similar to hard-decision decoding. In this implementation, the chips are not sliced, but instead the soft-decision values at the output of the delay-conjugate-multiply block is input to the correlators. As shown in Figures 3.16 and 3.17, the performance of the receiver is better with soft decision decoding. The disadvantage of soft-decision decoding is that it takes more processing power. In the hard-decision decoding, the correlation value can be obtained by simply performing an XOR operation of the input chips and the reference chips and then counting the number of bits that are '0'. Whereas in soft-decision decoding, 32-tap FIR filtering must be done for all 16 symbols. However, soft-decision decoding gives better performance at low SNR. This is in line with the result that Viterbi decoders give better results with soft-decision decoding in error-correcting codes [26].

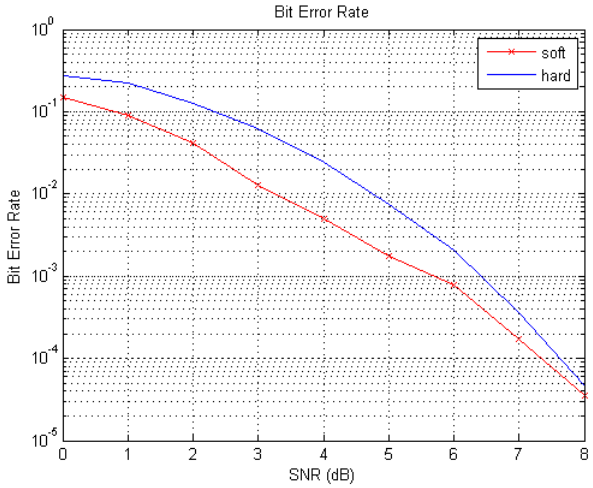

Figure 3.16: BER Simulation of Soft vs. Hard Correlations

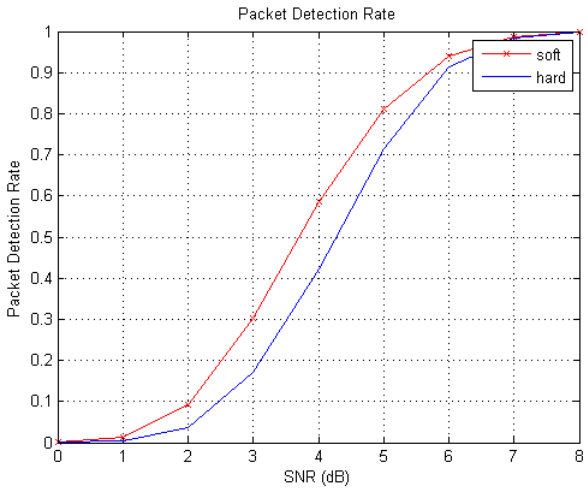

Figure 3.17: Packet Detection Rate Simulation of Soft vs. Hard Correlations

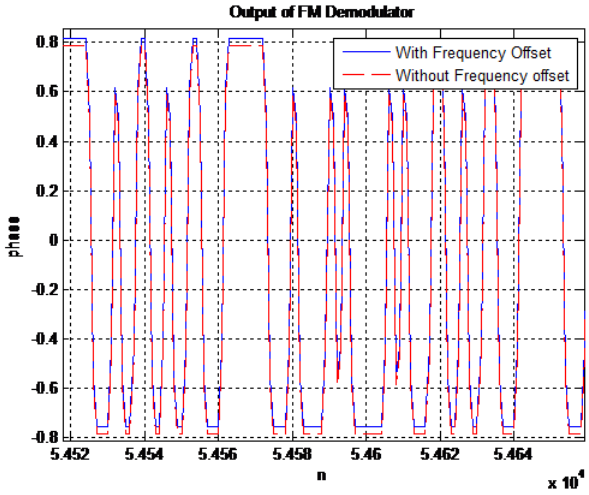

Figure 3.18: Output of delay-conjugate-multiply with frequency offset of 20 KHz

Frequency Offset Estimation

Frequency offset occurs when the local oscillators at the receiver and the transmitter are not completely in sync. This results in a slight frequency shift when the RF signal is converted to baseband. For the O-QPSK case, when the baseband signal is demodulated using the delay-conjugate-multiply method, it results in a slight DC offset as shown in Figure 3.18.

The IEEE 802.15.4 standard uses a similar preamble structure as the IEEE 802.11a standard. The IEEE 802.11a standard has a preamble composed of ten short preambles repeating every 0.8us. Similarly, EEE 802.15.4 has a preamble structure which consists of eight repeating sequence of chips every 16us. Thus, the frequency offset of the IEEE 802.15.4 packets can be estimated in a similar way as is done for the IEEE 802.11a packets [27].

The frequency offset of the received signal can be estimated using the received preamble

at two different sample times.

$$y(t) = x(t)e^{j2\pi f_\Delta t} \tag{3.3}$$

and

$$y(t-T) = x(t-T)e^{j2\pi f_\Delta (t-T)} \tag{3.4}$$

Where $y(t)$ and $y(t-T)$ are received preambles and $x(t)$ and $x(t-T)$ are transmitted preamble. T is the period of repetition in the preamble. The complex sinusoid gives a frequency offset of f_Δ to the transmitted preamble. Since the preamble $x(t)$ repeats with the period T, $y(t-T)$ can be re-written as

$$y(t-T) = x(t)e^{j2\pi f_\Delta (t-T)} \tag{3.5}$$

Multiplying the received signal $y(t-T)$ and the conjugate of $y(t)$ gives the following expressions.

$$\begin{aligned} y^*(t)y(t-T) &= x^*(t)e^{-j2\pi f_\Delta (t)}x(t)e^{j2\pi f_\Delta (t-T)} \\ &= x^*(t)x(t)e^{-j2\pi f_\Delta (t)}e^{j2\pi f_\Delta (t-T)} \\ &= |x(t)|^2 e^{j2\pi f_\Delta (-t+t-T)} \\ &= |x(t)|^2 e^{j2\pi f_\Delta (-T)} \end{aligned} \tag{3.6}$$

Taking the angle of the expression in 3.6 gives the following result.

$$\angle y^*(t)y(t-T) = 2\pi f_\Delta(-T) \tag{3.7}$$

Dividing by an appropriate factor reveals the frequency offset of the received signal.

$$f_\Delta = \frac{2\pi f_\Delta(-T)}{2\pi(-T)} \tag{3.8}$$

Since the range of valid values for the result of the angle function is $(-\pi, \pi]$, the minimum and maximum frequency offset that can be estimated are the following.

$$f_{\Delta min} = \frac{-\pi}{2\pi(T)} = -31.25kHz \tag{3.9}$$
$$f_{\Delta max} = \frac{\pi}{2\pi(T)} = 31.25kHz$$

Figure 3.19 shows the output of the frequency offset estimation. During the preamble, the simulated frequency offset of f_Δ = 20 kHz is correctly estimated. However, as seen in Figure 3.18 the delay-conjugate-multiply demodulation scheme is very robust to the effects of frequency offset. Therefore, no frequency offset correction is needed.

Simulink Simulation

Once the signal processing chain is verified with MATLAB simulations, it is broken down into individual blocks to be implemented in Simulink. The same input test data used

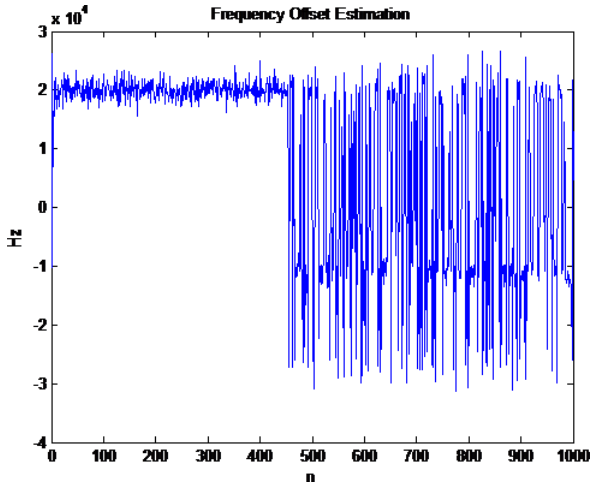

Figure 3.19: Frequency Offset Estimation

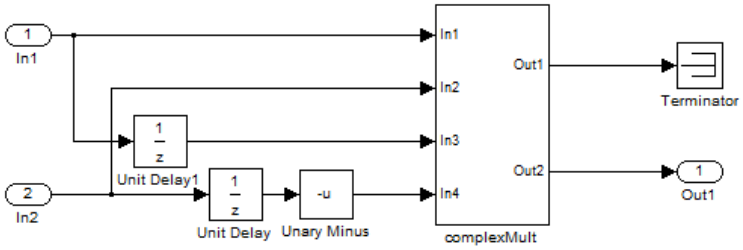

Figure 3.20: Delay-conjugate-multiply Block in Simulink

for MATLAB simulation is used for the Simulink model to see that the results of the two simulations matched. Figure 3.20 and 3.21 show two of the blocks implemented for the Simulink model. In addition to using the library of signal processing blocks available in Simulink, the user can also create custom blocks written in HDL and use wrappers known as *Black Box* for co-simulation of the custom HDL blocks in the Simulink environment. This is a very powerful ability that lets users combine custom HDL blocks with standard Simulink blocks in a design. The user is able to pick and choose which blocks to implement in hand-written HDL and which blocks to implement using just the standard libraries. Furthermore, once the Simulink-only blocks are verified, the user can

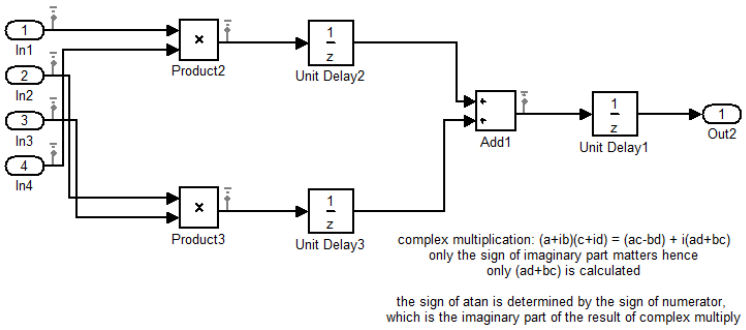

Figure 3.21: Complex Mulitply Block in Simulink

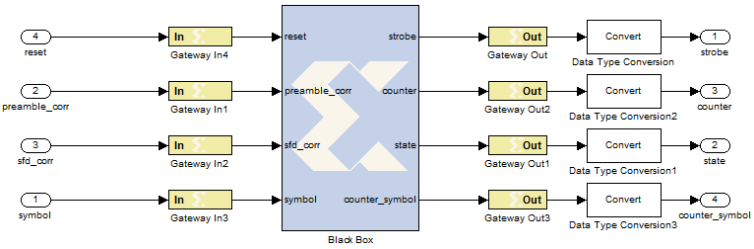

Figure 3.22: Black Box wrapper for a simple MAC processing

replace with the blocks with hand-optimized HDL blocks for better performance and still test the system in a Simulink environment. Figure 3.22 shows an example of a Black Box wrapper for a simple MAC processing. Although Simulink blocks are useful for streaming applications, it is often hard to implement control logic with finite states in Simulink. MathWorks does provide a tool called StateFlow for building state machines, but they are often cumbersome to use. Black Box allows users to implement complicated state machines in HDL rather than Simulink blocks or StateFlow.

Once the algorithm is tested in Simulink, the signals need to be converted to fixed-point format. When the signals are all converted to fixed-point format, the algorithm is tested and verified once again. After the results are verified, the Simulink model is ready for conversion to Verilog or VHDL code. The HDL Coder takes the Simulink model and

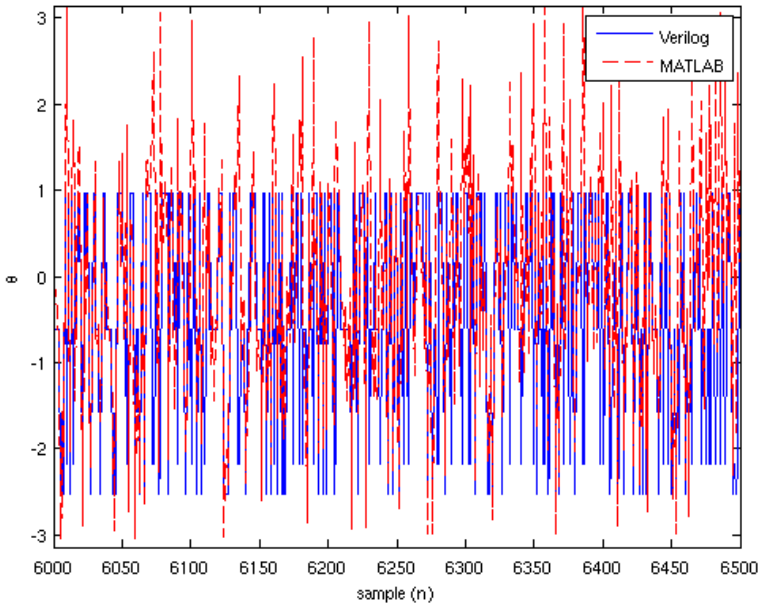

Figure 3.23: Output of CORDIC atan without AGC

converts the blocks into ”bit-true and cycle-accurate” Verilog or VHDL code [28]. The Black Box implementations that are already written in HDL are used as they are written.

RTL Implementation

For RTL implementation, the blocks shown in Figure 3.9 are converted to Verilog modules manually. The AGC block is required to utilize as much dynamic range as possible.

Figure 3.23 shows the effects of limited word-length in fixed-point implementation. When the gain of the USRP is set to a low value, the magnitude of the I and Q samples are small. This leads to limited dynamic range in the received samples. When these values are input to the CORDIC atan module, because of the small dynamic range, the atan block is not able to compute the output accurately. Therefore, the AGC block is needed

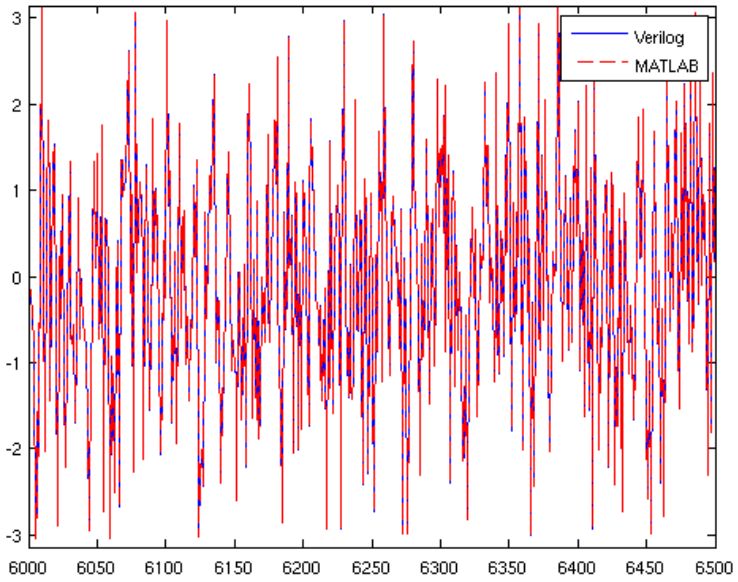

Figure 3.24: Output of CORDIC atan with AGC

at the front of the receiver to maximize the dynamic range of the input samples. Figure 3.24 shows that the output of the atan block matches the ideal MATLAB reference when the AGC block is introduced.

The correlations are performed using the Xilinx FIR cores. For the preamble and SFD correlations, a 256-tap FIR filter and a 64-tap FIR filter are generated using Xilinx CORE Generator. For the symbol correlations, 16 32-tap FIR filters are generated for each of the 16 chip sequences. The inputs to each filter are 16-bit signed soft-decision values. The input clock period for the filters are set to 50 so that the multipliers in the DSP48 slices could be used more efficiently by time-multiplexing. In contrast to the GNU Radio implementation, the FPGA implementation can calculate 16 correlation values simultaneously with soft-decision values because it supports parallelism. However, the GNU Radio im-

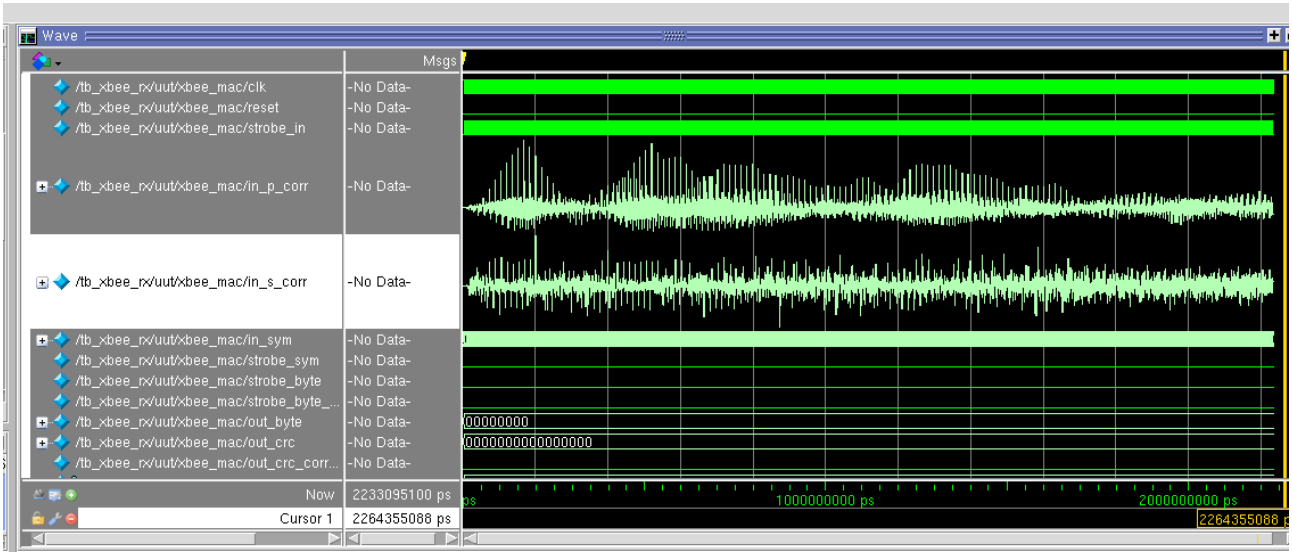

Figure 3.25: ModelSim Simulation Showing Preamble and SFD Corrleations

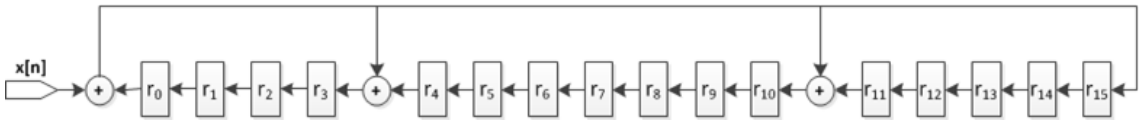

Figure 3.26: CRC-16

plementation must rely on a for-loop to iterate through the 16 correlation computations with hard-decision chips. If the processor is not fast enough to execute the for-loop, an overflow may occur.

Figure 3.25 shows the preamble and SFD correlations in ModelSim. The peaks in the correlations show where the packets are present.

Once the correlation values are computed, the state machine shown in Figure 3.27 searches for a valid packet by comparing the preamble and SFD correlations to the thresholds. If the correlation values are above the thresholds, the state machine decides that a valid packet is present and starts decoding the frame. As it decodes the payload, it generates

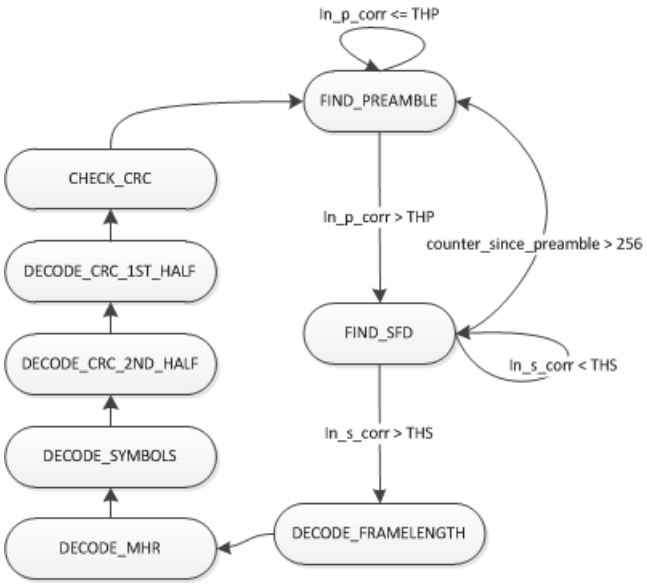

Figure 3.27: State machine for MAC layer

the checksum using a CRC module and compares against the received checksum. If the two checksums are identical, the MAC layer signals that the packet was valid. The decoded payload is sent off to the Ethernet MAC to be processed by GNU Radio. Figure 3.26 shows the structure of the CRC module.

FPGA Implementation

Similar to the transmitter implementation, the IEEE 802.15.4 receiver module is integrated into the USRP FPGA code as shown in Figure 3.28. Normally, the real and complex samples from the *dsp_rx0* core are sent to *vita_rx_chain* directly to be formatted and sent over to the host. However, for the implementation, the samples are intercepted by the IEEE 802.15.4 receiver to be processed. The decoded bytes from the receiver module are sent out to the *vita_rx_chain* core which frames the bytes with VITA headers and sends out

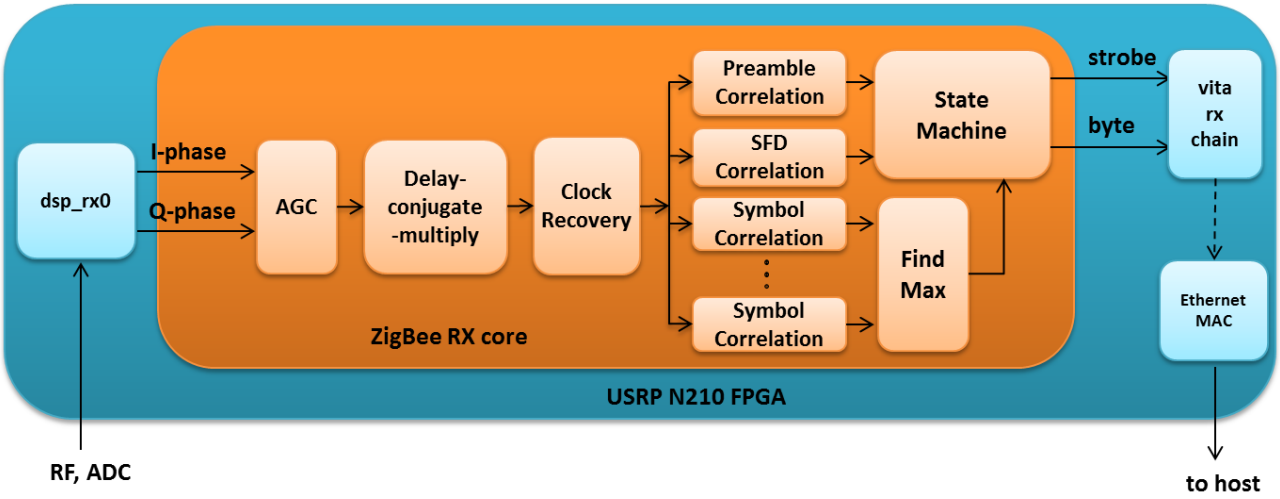

Figure 3.28: Receiver module inside USRP's FPGA

to the Ethernet MAC module.

The data format sent from the FPGA to GNU Radio is normally a 32-bit word consisting of 16-bit I and 16-bit Q data. However, since the data being sent from the FPGA is no longer samples but decoded bytes, the format has to be changed. Table 3.1 shows the new data format sent from FPGA to GNU Radio. The 32-bit data contains not only the decoded byte but also the state of the state machine, a strobe signal for when a valid byte is output, a flag to indicate if the checksum was correct, and debug signals.

GNU Radio Signal Processing Block

In order to interface with the modified FPGA of USRP N210, a custom GNU Radio signal processing block was created. The custom block is able to decode the new 32-bit data format shown in Table 3.1. Figure 3.29 shows the GNU Radio Companion view

Table 3.1: Output data format for USRP

31	30	29	28	27	26	25	24
0	0	out strobe byte	out CRC correct	out symbol			
23	22	21	20	19	18	17	16
out state				1	1	1	1
15	14	13	12	11	10	9	8
debug_signals							
7	6	5	4	3	2	1	0
out_byte							

of the working ZigBee Radio. All of the IEEE 802.15.4 receiver radio logic resides inside the USRP's FPGA. Hence, the user only sees the *UHD: USRP Source* block and the *zigbee_rx_mod* block. The *zigbee_rx_mod* block is a custom block that was created to correctly parse the output data coming from the USRP.

Bypass Mode

In order to let users use the USRP as originally intended, a bypass logic was added to the FPGA. This bypass logic allows the user to select whether to operate the USRP as a IEEE 802.15.4 receiver, or as a normal USRP by bypassing the IEEE 802.15.4 receiver logic inside the FPGA. Figure 3.30 shows the modified *UHD: USRP Source* block where the user is able to select whether to operate the USRP as a ZigBee Radio or as a normal USRP.

In order to switch between the normal mode and IEEE 802.15.4 receiver mode, a register inside the FPGA needs to be toggled from the GNU Radio environment. This is accom-

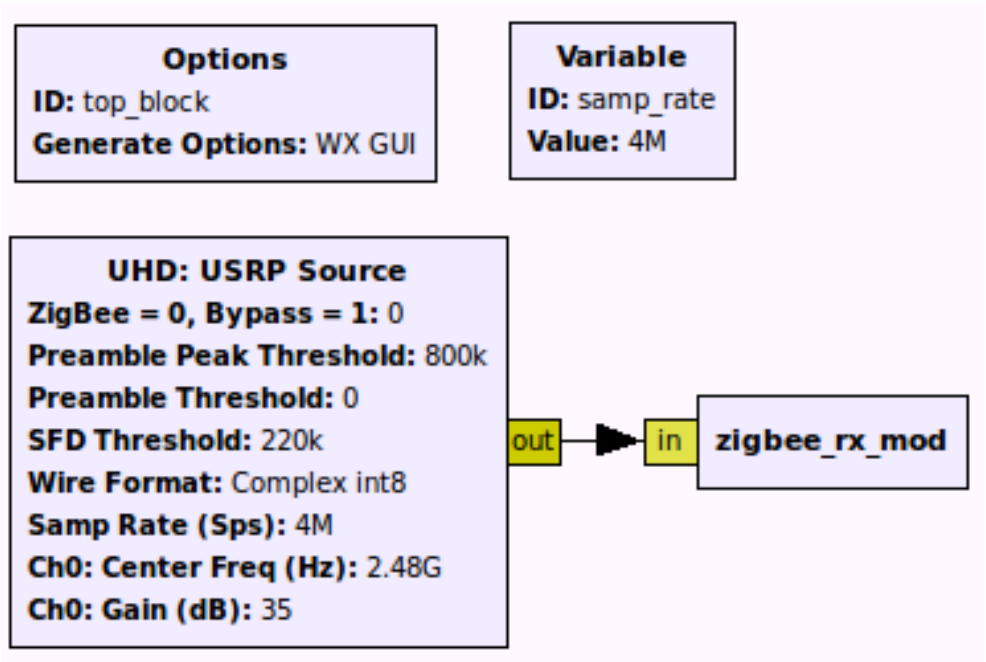

Figure 3.29: GNU Radio companion view of ZigBee RX

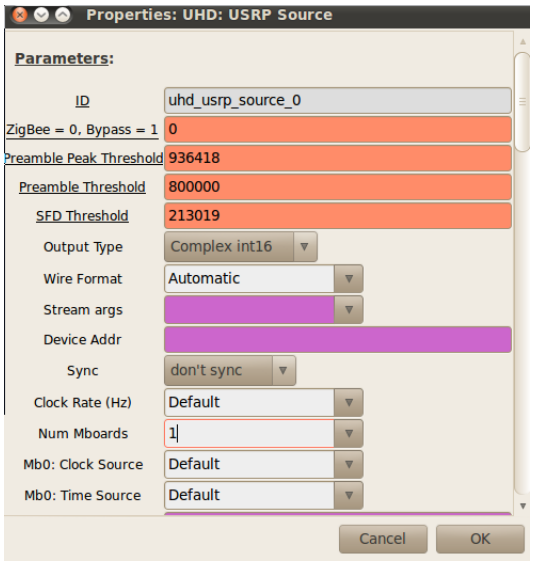

Figure 3.30: Modified UHD USRP Source Block

plished by modifying the whole control chain that spans GNU Radio, UHD, and FPGA. Figure 3.31 shows the series of modifications made to enable changing the register inside the FPGA from a GNU Radio block.

Using the same logic for enabling the bypass mode, the thresholds for preamble and SFD can be controlled also by changing the registers values inside the FPGA. Since usually only a single receive channel *dsp_rx0* on the USRP is used, the registers in *dsp_rx1* were re-routed to control the bypass register and the threshold registers.

3.2 Hybrid Multi-Channel IEEE 802.15.4 Receiver

3.2.1 Configuration

The hybrid implementation of multi-channel IEEE 802.15.4 receiver consists of three major processing entities as shown in Figure 3.32: USRP N210, XUPV5, and GNU Radio. The USRP acts as an RF front-end and a digital down converter. XUPV5 processes high sample rate signal, and GNU Radio processes low sample rate signal. A channelizer resides in XUPV5's Virtex 5 FPGA to split a 20 MHz bandwidth signal into four 5 MHz channels. An energy detector in the same FPGA then detects which channel is occupied by the IEEE 802.15.4 transmitter. The energy detector then tells a 4:1 multiplexer to send only the output of the occupied channel to GNU Radio. The IEEE 802.15.4 receiver in GNU Radio then demodulates the received channel. When the transmitter switches to a

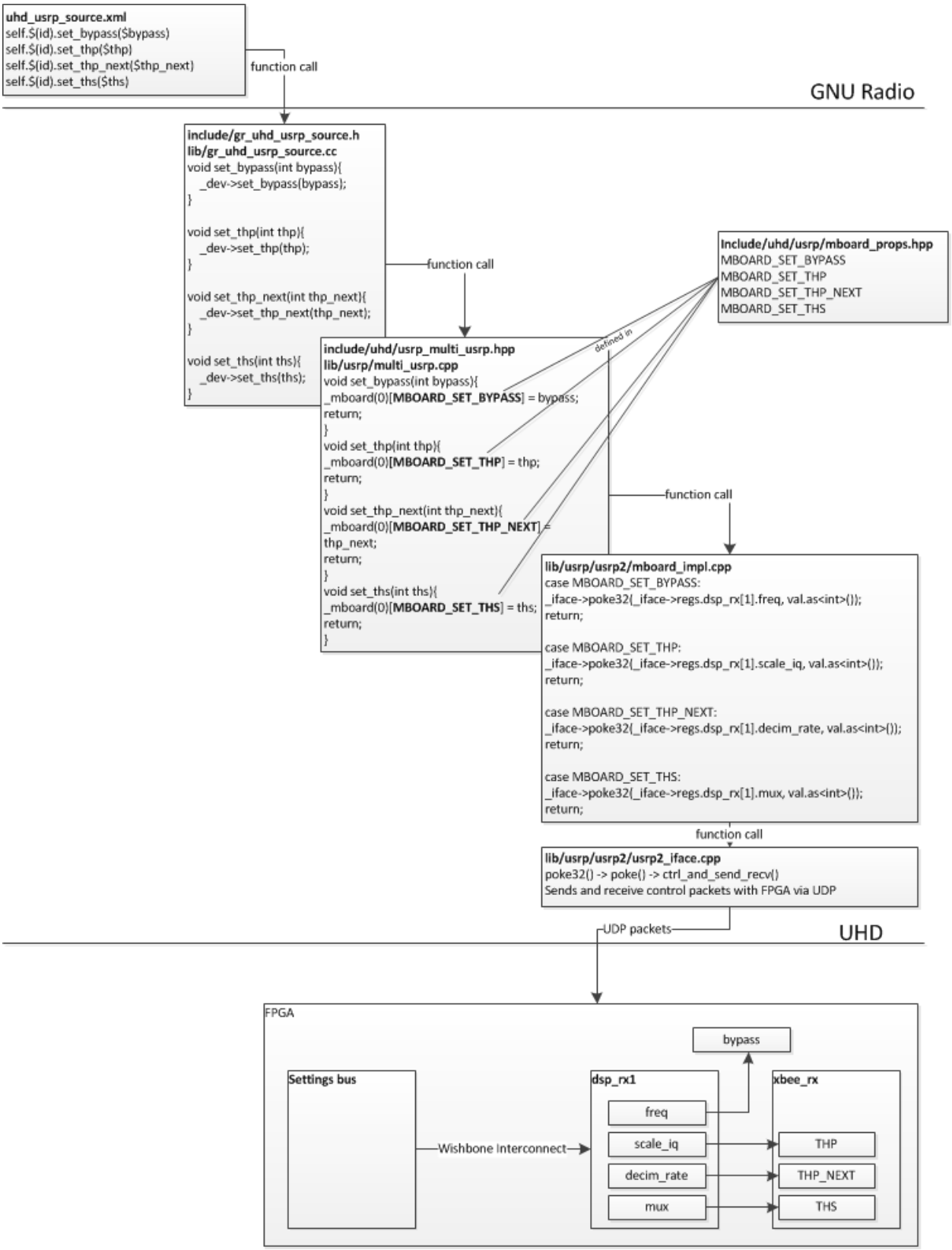

Figure 3.31: Control Chain in Modifying Registers Inside FPGA from GNU Radio

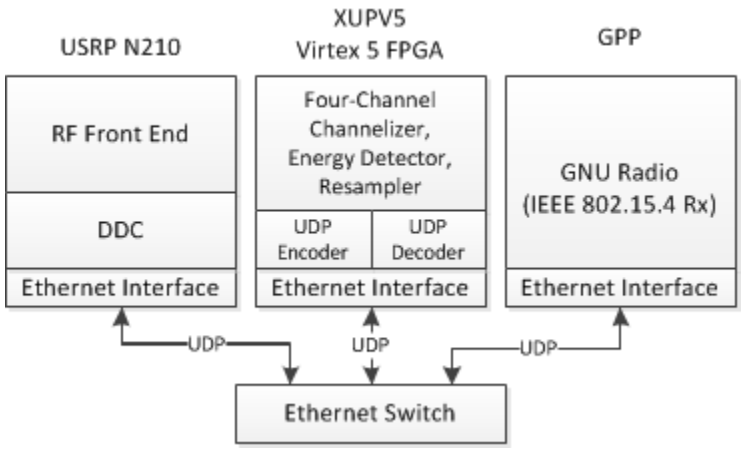

Figure 3.32: System Configuration of Hybrid Multi-Channel IEEE 802.15.4 Receiver

different channel, the energy detector correctly identifies the newly occupied channel and multiplexes the correct channel to GNU Radio.

3.2.2 Channelizer

In order to implement a multi-channel IEEE 802.15.4 receiver, a channelizer is needed. The IEEE 802.15.4 channels are separated by 5 MHz as specified in the standard. The maximum sample rate of USRP N210 is 25 MHz at baseband with 16-bit I and Q samples. Therefore, a maximum of five channels (25 MHz/5 MHz) can be simultaneously channelized. However, a five-channel channelizer requires a five-point IDFT block. In order to utilize the efficient radix-4 FFT algorithm, a four-channel channelizer is implemented instead using a four-point FFT. Thus, the USRP N210 runs at the sampling frequency of 20 MHz instead of 25 MHz.

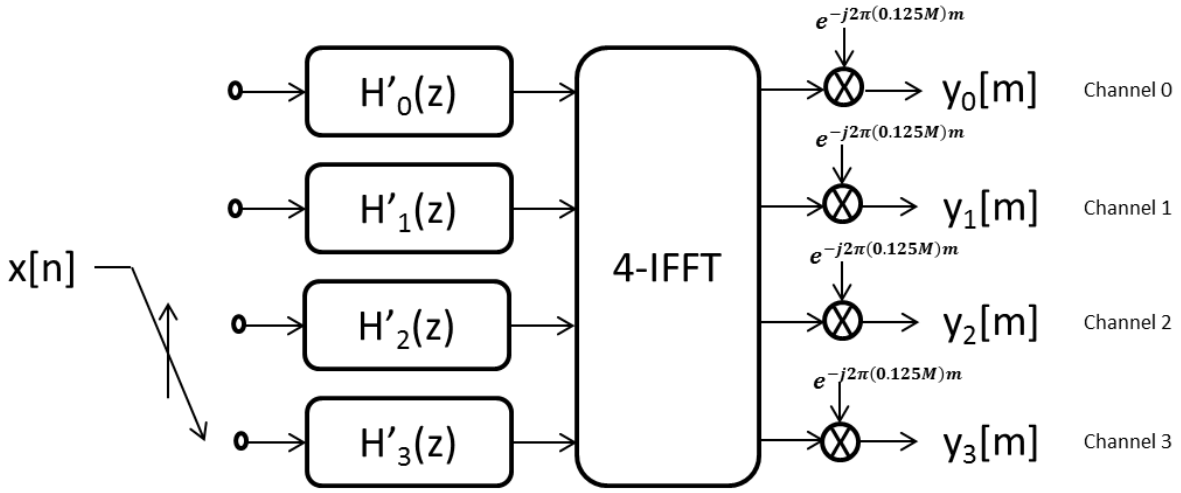

Figure 3.33: Channelizer 4:1

MATLAB Simulation

Figure 3.33 shows the architecture of a four-channel channelizer. The channelizer presented in [23] has the first channel (channel 0) centered at baseband. An example of such channelizer was shown in Figure 2.6 in the earlier section. However such channel positioning makes one half of the edge channel (channel 4 in Figure 2.6) alias to the opposite end of the spectrum. Therefore when the baseband signal from the USRP is channelized, only the seven channels in the middle are useful, since the edge channel contains two halves of opposite ends of the spectrum. In order to prevent this aliasing, the lowpass filter is up-converted to the center frequency of F=0.125 by multiplying with a complex sinusoid so that the center of edge channels will not be at the sampling frequency. Figure 3.34 shows the complex lowpass filter up-converted to F=0.125. Using the complex lowpass filter as a starting point, the channels are now evenly distributed across 20 MHz bandwidth as shown in Figure 3.35 with the edge channel not split between the opposite

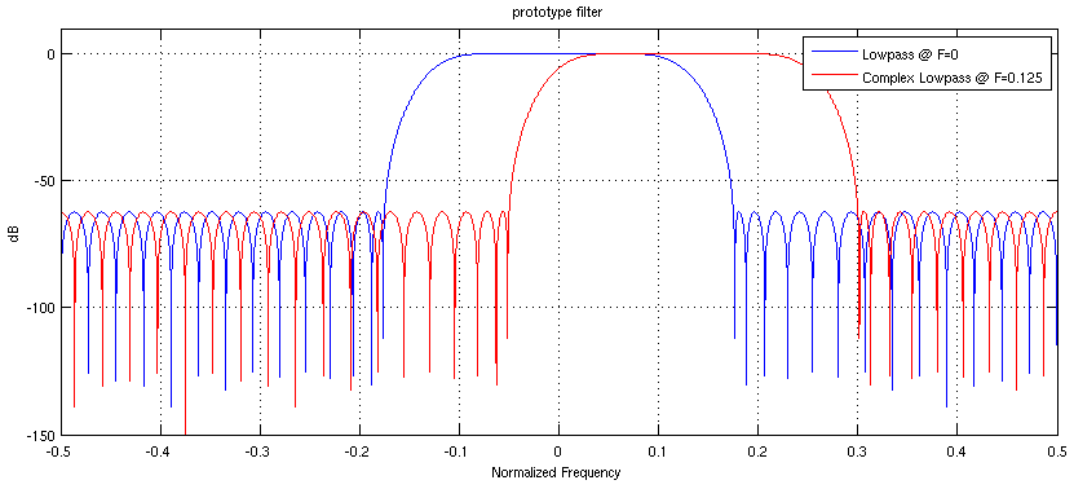

Figure 3.34: Prototype Lowpass Filter

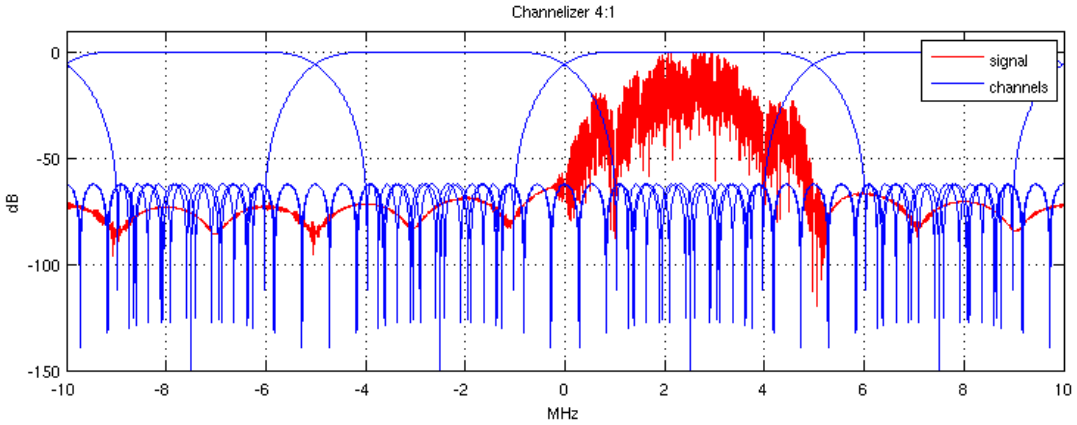

Figure 3.35: Input to Four-Channel Channelizer

ends of the spectrum. The disadvantage of this process is the filter coefficients of the filter bank are now complex. Each channel also needs to be down-converted to baseband at the output of the channelizer, since the prototype lowpass filter is not centered at baseband. Figure 3.36 shows the IEEE 802.15.4 signal residing in *channel 0*.

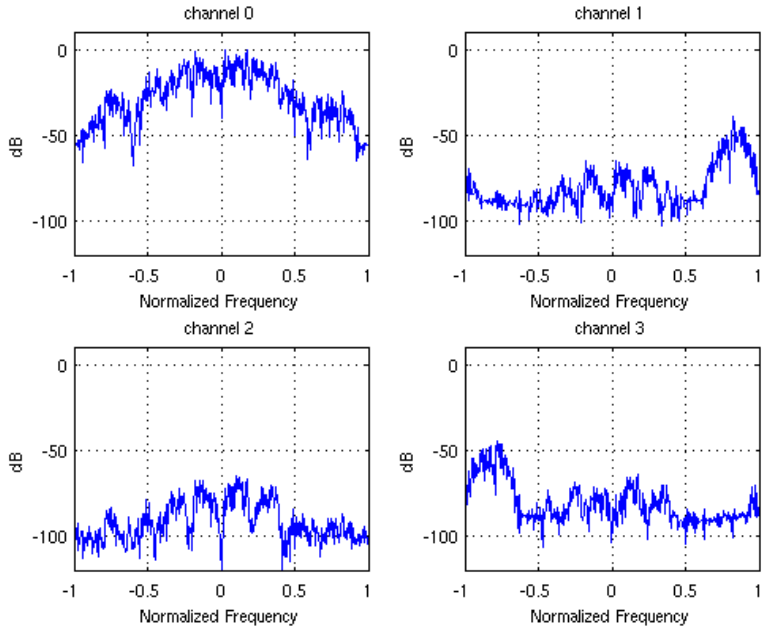

Figure 3.36: Output of Four-Channel Channelizer

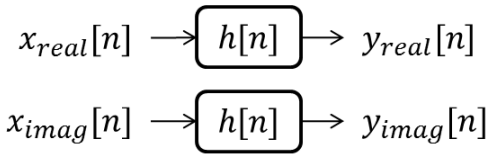

Figure 3.37: Filter with Real Coefficients

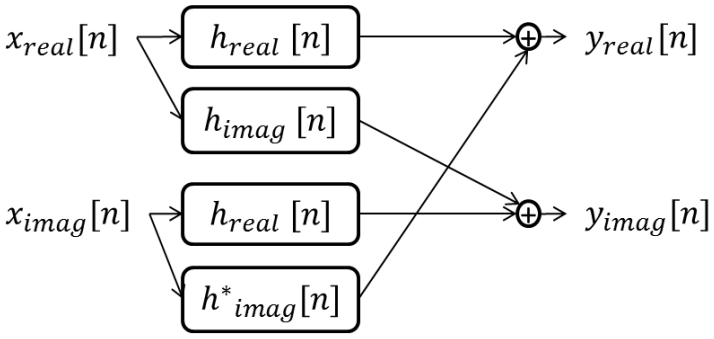

Figure 3.38: Filter with Complex Coefficients

FPGA Implementation

Since the lowpass filter is up-converted to F=0.125, the coefficients of the new lowpass filter are complex-valued. When a filter is real, the i-phase and q-phase parts of the signal can be independently filtered as shown in Figure 3.37. However, when a filter is complex, the i-phase and q-phase have to be both filtered by real and imaginary coefficients of the filter and combined as shown in Figure 3.38. Therefore, complex filters require twice as many resources as real filters.

Once the samples are filtered, they are input to the FFT block. The four-point FFT can be easily built with the radix-4 FFT architecture. Since N=4, only a single radix-4 butterfly is required. In matrix form, four-point DFT can be expressed as Equation 3.10.

$$\begin{bmatrix} X[0] \\ X[1] \\ X[2] \\ X[3] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix} \times \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ x[3] \end{bmatrix} \tag{3.10}$$

The matrix equation can be simplified as Equation 3.11 which only consists of additions and subtractions. Thus, using the radix-4 butterfly, the four-point FFT can be reduced to

a series of simple additions and subtractions.

$$
\begin{aligned}
X(0) &= x[0] + x[1] + x[2] + x[3] \\
&= x_i[0] + x_i[1] + x_i[2] + x_i[3] + j(x_q[0] + x_q[1] + x_q[2] + x_q[3]) \\
X(1) &= x[0] - jx[1] - x[2] + jx[3] \\
&= x_i[0] + x_q[1] - x_i[2] - x_q[3] + j(x_q[0] - x_i[1] - x_q[2] + x_i[3]) \\
X(2) &= x[0] - x[1] + x[2] - x[3] \\
&= x_i[0] - x_i[1] + x_i[2] - x_i[3] + j(x_q[0] - x_q[1] + x_q[2] - x_q[3]) \\
X(3) &= x[0] + jx[1] - x[2] - jx[3] \\
&= x_i[0] - x_q[1] - x_i[2] + x_q[3] + j(x_q[0] + x_i[1] - x_q[2] - x_i[3])
\end{aligned}
\tag{3.11}
$$

After IFFT, each channel needs to be down-converted by $F = 0.125M$ to compensate for the up-converted lowpass filter. The down-conversion is achieved by multiplying each channel with the complex sinusoid $e^{-j2\pi M0.125(m)} = e^{-j2\pi 0.5(m)}$. The complex sinusoid at F=0.5 simplifies to $1, -1, 1, -1, 1....$ Therefore, down-conversion for each channel simplifies to flipping the sign of every other sample.

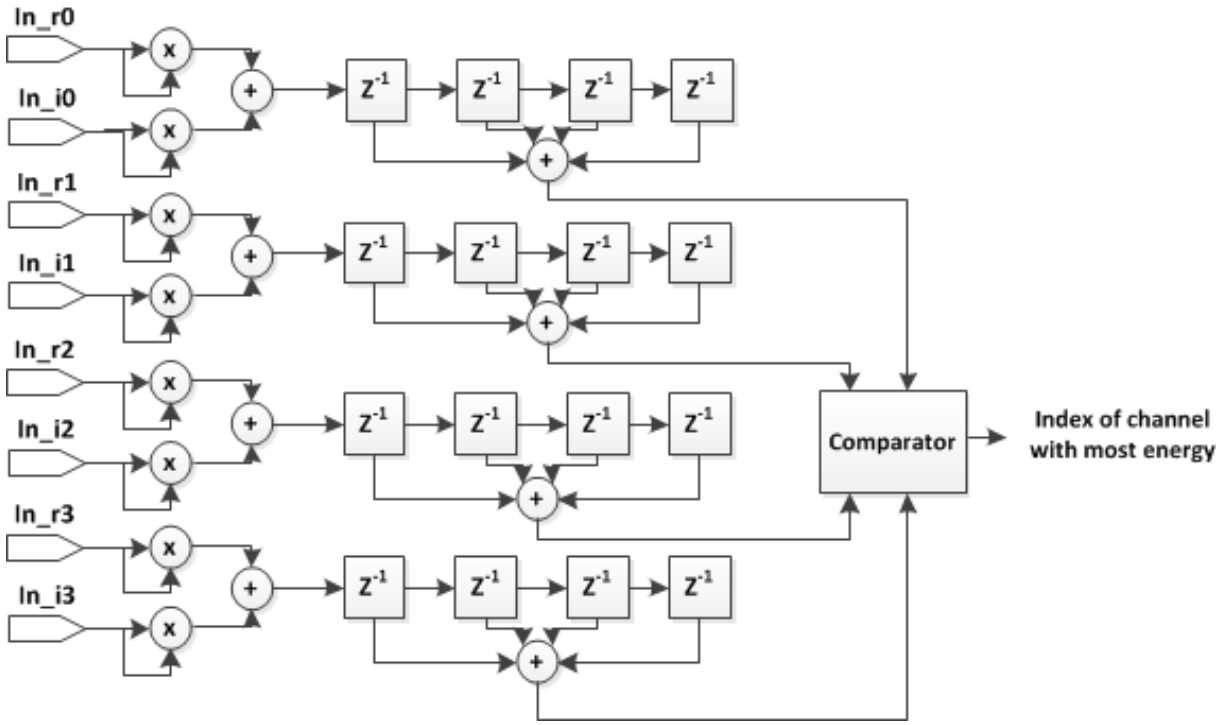

Figure 3.39: Energy Detector

3.2.3 Energy Detector

FPGA Implementation

Figure 3.39 shows the energy detector to detect the channel where signal is present. The average of four consecutive samples of magnitude of each channel is compared, and the channel with most energy is sent through a multiplexer to the resampler.

3.2.4 Resampler 4/5

Since the sample rate of each channel is 5 MHz, a resampler is needed to convert the sample rate to 4 MHz for the demodulator to work properly. A simple polyphase filter-

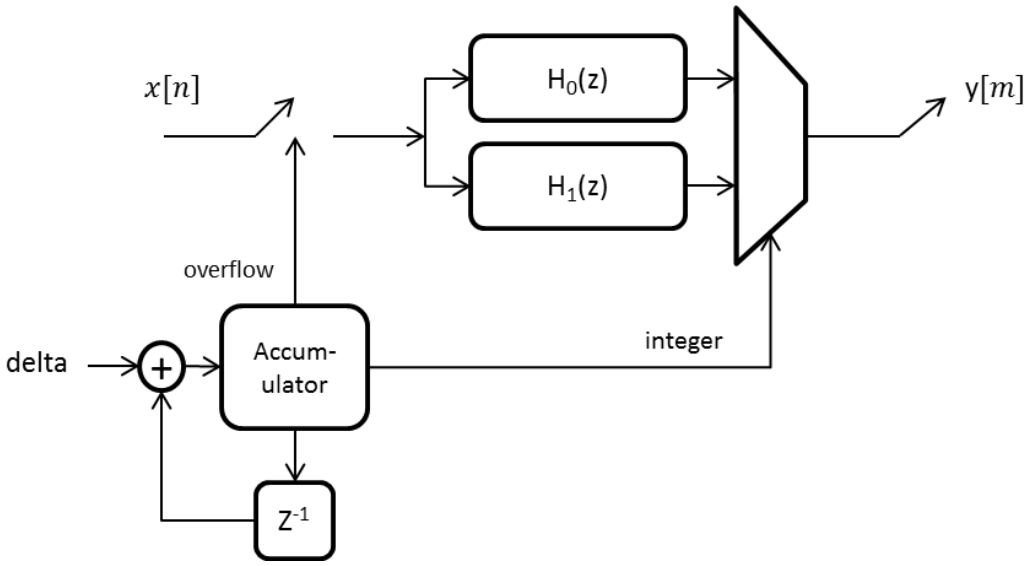

Figure 3.40: Resampler 4/5

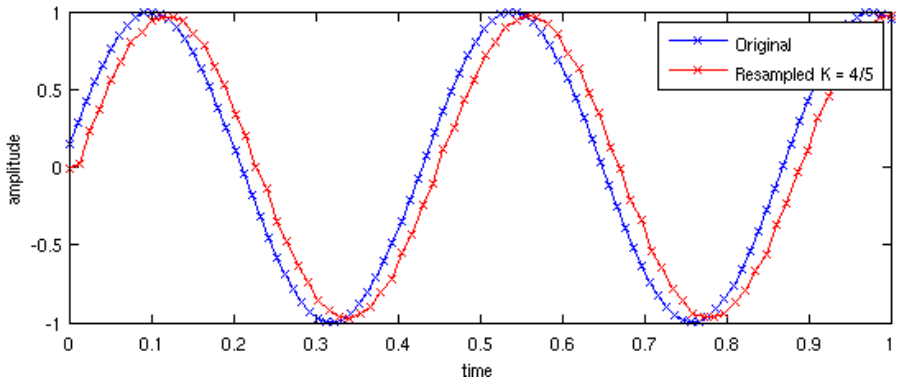

Figure 3.41: Resampler 4/5 Simulation

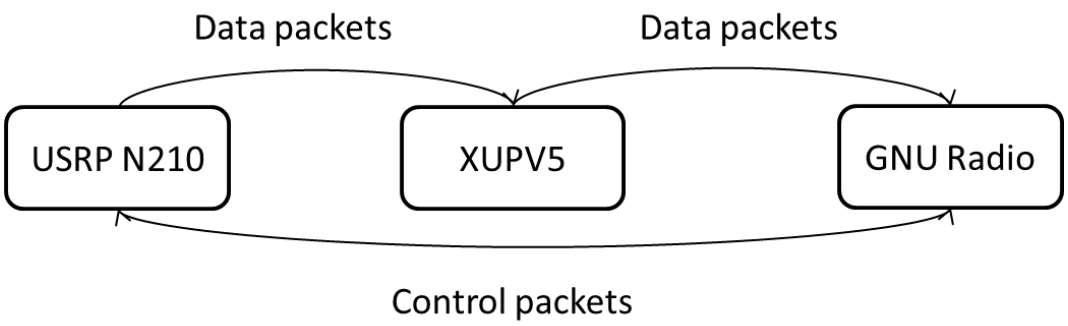

Figure 3.42: Flow of Data and Control Packets

bank resampler shown in Figure 3.40 is used to convert the sample rate from 5 MHz to 4 MHz. Figure 3.41 shows the simulation in MATLAB of resampling the input signal by a factor of 4/5.

3.2.5 Ethernet interface

UHD Modifications

For the external FPGA to process the samples from the USRP N210 before sending the result to GNU Radio, the USRP N210 must be configured to send the samples to the external FPGA instead of GNU Radio. By fixing the MAC destination of the UDP packets to be that of the external FPGA, the sample packets from USRP N210 can be redirected to the external FPGA. Figure 3.42 shows the flow of data and control packets. After processing the samples, XUPV5 sends the processed samples to GNU Radio as if the samples were coming straight from USRP N210. GNU Radio then accepts the samples for further processing.

Simple changes in the firmware of USRP N210 can redirect sample packets to designated MAC address while control packets still communicate with the host.

```
static void setup_network(void){
  //setup ethernet header machine
  /*sr_udp_sm->eth_hdr.mac_dst_0_1 = (fp_mac_addr_dst.addr[0] << 8) |
      fp_mac_addr_dst.addr[1];
  sr_udp_sm->eth_hdr.mac_dst_2_3 = (fp_mac_addr_dst.addr[0] << 8) |
      fp_mac_addr_dst.addr[1];
  sr_udp_sm->eth_hdr.mac_dst_4_5 = (fp_mac_addr_dst.addr[0] << 8) |
      fp_mac_addr_dst.addr[1];*/

  // Fix destination MAC address to be that of the external FPGA
  sr_udp_sm->eth_hdr.mac_dst_0_1 = (0x00 << 8) | 0x0A;
  sr_udp_sm->eth_hdr.mac_dst_2_3 = (0x35 << 8) | 0x00;
  sr_udp_sm->eth_hdr.mac_dst_4_5 = (0x01 << 8) | 0x02;

  sr_udp_sm->eth_hdr.mac_src_0_1 = (fp_mac_addr_src.addr[0] << 8) |
      fp_mac_addr_src.addr[1];
  sr_udp_sm->eth_hdr.mac_src_2_3 = (fp_mac_addr_src.addr[2] << 8) |
      fp_mac_addr_src.addr[3];
  sr_udp_sm->eth_hdr.mac_src_4_5 = (fp_mac_addr_src.addr[4] << 8) |
      fp_mac_addr_src.addr[5];

  ...
  uint32_t dst_ip_addr = fp_socket_dst.addr.addr+2; // Change destination
      address from 192.168.10.1 to 192.168.10.3 (IP of XUPV5)
  ...

  sr_udp_sm->udp_hdr.checksum = UDP_SM_LAST_WORD;
}
```

FPGA

Once the packets arrive at XUPV5 from USRP N210, the FPGA must know how to decode the UDP packets. The PHY and MAC layers of Ethernet are handled by Ethernet PHY and the Xilinx Embedded Tri-Mode Ethernet MAC core. Figure 3.43 shows the Ethernet interface and packet encoders and decoders to handle UDP packets inside XUPV5.

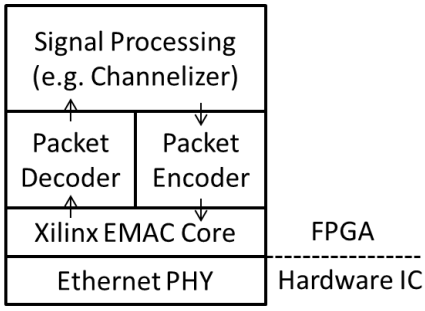

Figure 3.43: Ethernet interface in XUPV5

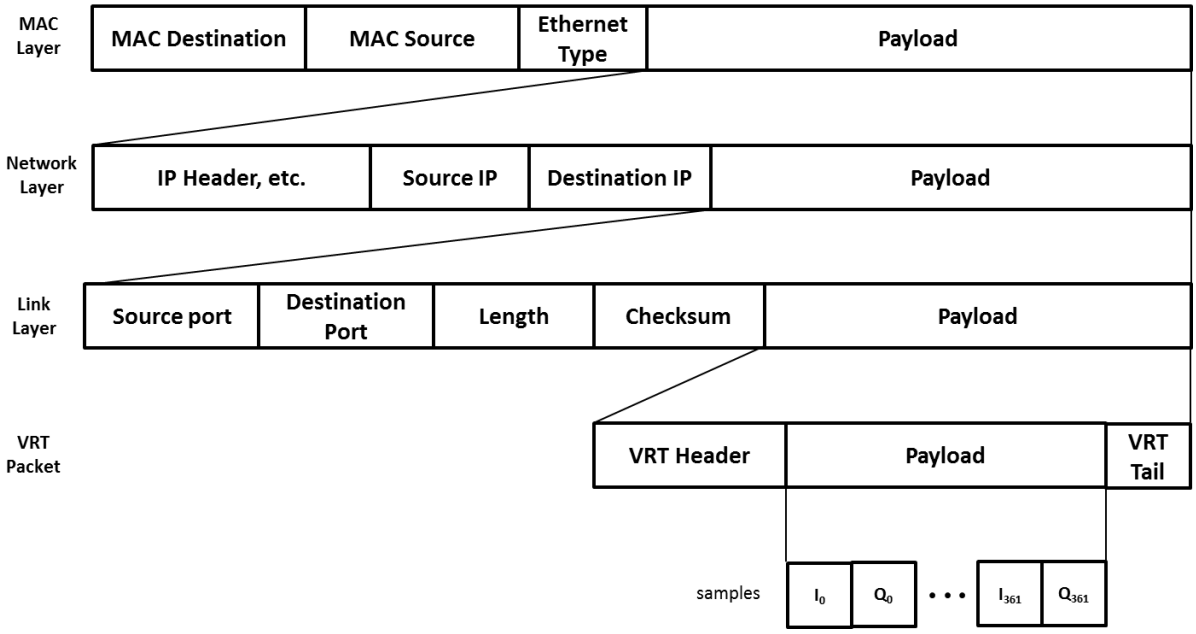

Figure 3.44: Frames at Different OSI Layers

Figure 3.44 shows the structure of UDP packet for transmitting the received samples from USRP N210 to GNU Radio. XUPV5 must be able to extract the samples embedded within the payload of the UDP packet.

When a new packet arrives, the packet decoder module extracts the payload of the UDP packet and strips the VRT header and tail. The extracted samples are then sent to the channelizer module for signal processing. The output of the channelizer is then sent to the packet encoder module which constructs a valid frame that can be read by GNU Radio.

Chapter 4

Results

4.1 Resource Utilization

The tables in this section show the resource utilizations of the FPGA implementations of the IEEE 802.15.4 PHY and the multi-channel receiver. For the IEEE 802.15.4 PHY, the the transmitter is implemented on USRP2, while the receiver is implemented on USRP N210. The receiver is implemented on the USRP N210 instead of USRP2 because the latter does not have enough resources to support signal processing for the receiver algorithm.

4.1.1 Transmitter

Table 4.1 shows the resource utilization of the IEEE 802.15.4 transmitter by itself. No multiplier is used for the transmitter even though the transmitter core has a half-sin pulse

Table 4.1: Device Utilization Summary of ZigBee TX core on Xilinx Spartan 3-2000

Logic Utilization	Used	Available	Utilization
Slice Flip Flops	786	40,960	1%
4 input LUTs	782	40,960	1%
Occupied Slices	820	20,480	4%
MULT 18x18s	0	40	0%
RAMB 16s	0	40	0%

Table 4.2: Device Utilization Summary of ZigBee TX core and USRP core on Xilinx Spartan 3-2000

Logic Utilization	Used	Available	Utilization
Slice Flip Flops	20,055	40,960	48%
4 input LUTs	29,024	40,960	70%
Occupied Slices	17,130	20,480	83%
MULT 18x18s	31	40	77%
RAMB 16s	25	40	63%

shaping filter. It is possible to eliminate the need of multipliers since the input to the filter is always either -1 or 1, and therefore there are only two possible output sequences from the pulse shaping filter: half-sin pulse or inverted half-sin pulse. Depending on the input, the filter can simply output one of the two possible sequences, instead of performing the actual multiply-accumulate operations. Since distributed RAM instead of block RAM is used for FIFO between the signal processing blocks, no RAMB 16s are used either.

Table 4.2 shows the resource utilization when the transmitter module is integrated with the USRP2 FPGA. The 27 multipliers are used by USRP2 for scaling and interpolation operations.

Table 4.3: Device Utilization Summary of ZigBee RX core on Xilinx Spartan 3A-DSP3400

Logic Utilization	Used	Available	Utilization
Slice Flip Flops	6,907	47,744	14%
4 input LUTs	7,372	47,744	7%
DSP48As	39	126	30%
RAM16	47	126	37%

Table 4.4: Device Utilization Summary of ZigBee RX core and USRP core on Xilinx Spartan 3A-DSP3400

Logic Utilization	Used	Available	Utilization
Slice Flip Flops	25,931	47,744	54%
4 input LUTs	35,097	47,744	73%
DSP48As	66	126	52%
RAM16	75	126	59%

4.1.2 Receiver

Table 4.3 shows the resource utilization of the IEEE 802.15.4 receiver by itself. In contrast to the transmitter, the receiver uses 39 multipliers because of the correlation operations done by Xilinx FIR cores and the interpolation operation by the clock recovery module. A lot more slices are used compared to the transmitter because of its higher complexity.

Table 4.4 shows the resource utilization when the receiver module is integrated into USRP N210's FPGA. There are still enough room to implement additional signal processing blocks such as symbol timing synchronization to improve the performance of the receiver. Table 4.5 shows the resource utilization by Karve's receiver implementation [2]. Karve's implementation employs only hard-decision correlation and no clock recovery module,

Table 4.5: Device Utilization Summary of Karve's ZigBee RX core (14-bit) on XUPV5 [2]

Logic Utilization	Used	Available	Utilization
Slice Flip Flops	1,954	69,120	2%
Slice LUTs	3,182	69,120	4%
DSP48Es	10	64	15%
Block RAM	9	148	6%
TEMACs	1	2	50%

Table 4.6: Post-Synthesis Timing Summary

Minimum Period	12.246 ns
Maximum Frequency	81.659 MHz

therefore it uses much less resource, even though it includes the Ethernet MAC module.

The post-synthesis and post-PAR timing summaries in Tables 4.6 and 4.7 show that the timing constraint of 100 MHz system clock is not met. However, using a different version of ISE, update USRP N210 FPGA code, and setting different synthesis, translate, and map properties may enable the tool to meet the timing constraints [29].

Table 4.7: Post-PAR Timing Summary

Minimum Period	16.804 ns
Maximum Frequency	59.510 MHz

Table 4.8: Device Utilization Summary of Channelizer on Xilinx Virtex 5 LX110T

Logic Utilization	Used	Available	Utilization
Slice Registers	3,224	69,120	4%
Slice LUTs	3,410	69,120	4%
Block RAM/FIFO	1	148	1%
DSP48Es	28	64	43%

Table 4.9: Device Utilization Summary of Channelizer with Ethernet MAC on Xilinx Virtex 5 LX110T

Logic Utilization	Used	Available	Utilization
Slice Registers	4,338	69,120	6%
Slice LUTs	4,250	69,120	6%
Block RAM/FIFO	10	148	6%
DSP48Es	28	64	43%

4.1.3 Channelizer

Table 4.8 shows the device utilization of the channelizer module only. It uses a number of DSP48E slices because of the number of filters required in the polyphase filter bank and the resampler module. Table 4.9 shows the device utilization when the channelizer is combined with the Ethernet MAC module. Even with the MAC, sufficient slices are left for further use.

	FPGA RX	GNU Radio RX	Xbee RX	Multi-Channel Rx
FPGA TX	interoperable	interoperable	interoperable	interoperable
GNU Radio TX	interoperable	interoperable	interoperable	interoperable
XBee TX	interoperable	interoperable	interoperable	interoperable

Table 4.10: Interoperability Chart

4.2 Interoperability

Combinations of different receiver and transmitter implementations are used to test interoperability. Table 4.10 shows that all three implementations of IEEE 802.15.4 are interoperable with each other.

Figure 4.1 shows that the commercially available X-Bee module is able to receive the IEEE 802.15.4 packets sent from the FPGA implementations on USRP 2.

4.3 Performance

4.3.1 IEEE 802.15.4 Receiver

The performance of the receiver is first simulated, and the actual bit error rate, packet error rate, and packet detection rate are measured over the air.

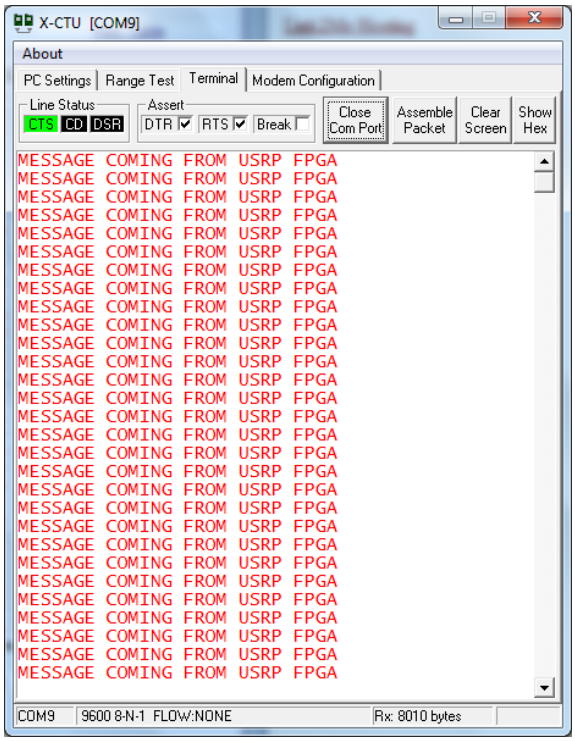

Figure 4.1: Message from FPGA received by Xbee console

Simulation

Figure 4.2 shows the chip error rate of O-QPSK signal. The theoretical curve is simply the bit error rate curve of the O-QPSK modulation. The simulation of O-QPSK demodulation very closely matches the theoretical curve. As expected, differential demodulation using the delay-conjugate-multiply scheme results in poorer performance. However, matched-filtering the input signal with half-sine pulse shape increases the performance of the differential demodulation. The simulation done in Karve's thesis is also shown [2]. All simulations assume perfect synchronization. Figure 4.3 shows the simulated BER curves. The "Coherent Demodulation" curve assumes perfect receiver synchronization. The O-QPSK chips are demodulated coherently and then despread to recover the information

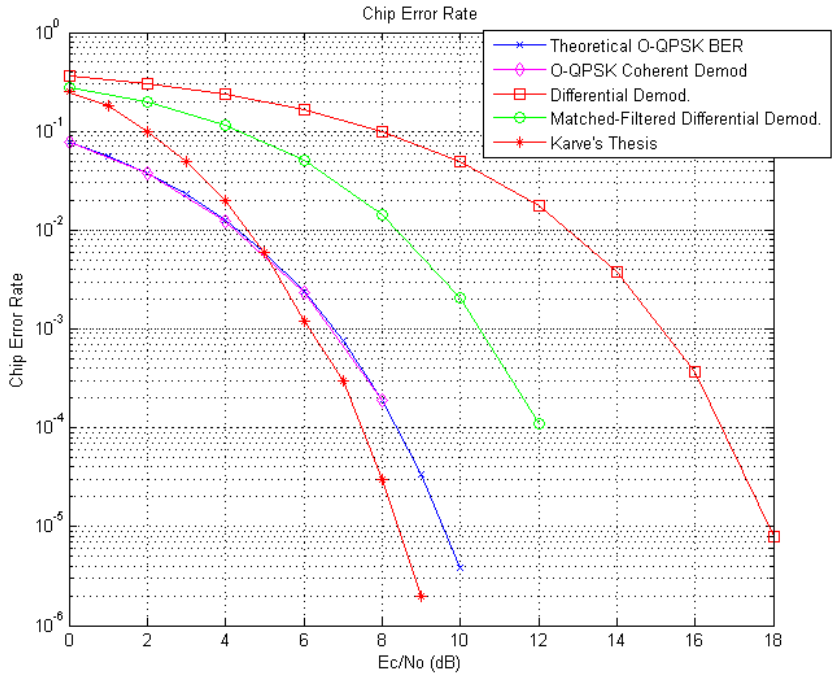

Figure 4.2: Chip Error Rate Simulation

bits. The "Coherent Differential Demodulation" curve assumes perfect receiver synchronization, but the O-QPSK chips are matched-filtered and demodulated differentially with the delay-conjugate-multiply block. The later case is closer to the implementation done for the thesis. As expected, the coherent demodulation of the O-QPSK symbol results in better performance and closely matches the simulation curve, while the differential demodulation results in poorer performance. The theoretical curve in Figure 4.3 is specified in the standard [1] as the following.

$$P_{BER,802.15.4} = \frac{8}{15}\frac{1}{16}\sum_{k=2}^{16} -1^{k}\binom{16}{k}e^{20SINR(\frac{1}{k}-1)} \tag{4.1}$$

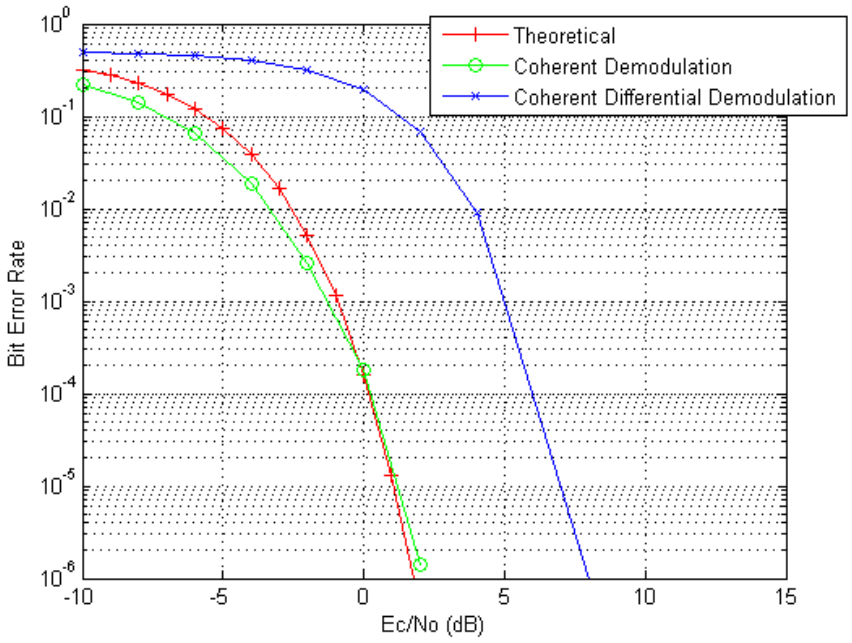

Figure 4.3: BER Simulation

SNR Measurement

The SNR values are calculated in the *bypass* mode, so that the raw received samples can be saved to a file and analyzed.

Two different measurements are done to calculate the SNR value. In the first measurement, only the noise power is measured. The following command is used to collect the samples from the USRP. No packets are sent from the Xbee module, so that only noise samples are collected.

```
1 ./rx_samples_to_file --freq=2480000000 --rate=4000000 --gain=70 --nsamps
    =4000000 --filename=noise_70dB.dat
```

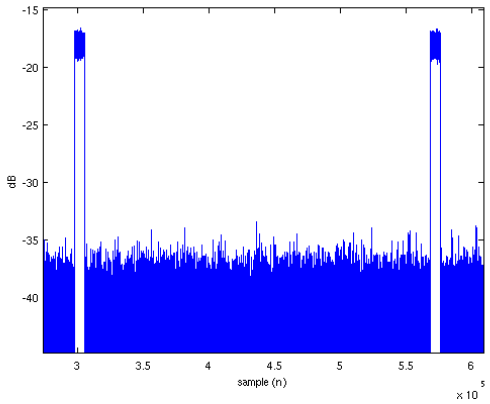

Figure 4.4: Received Packets in Time Domain

In the second measurement, the packets are sent from the XBee module so that the signal of interest as well as the noise samples are captured. The gain on the receiver is set to be the same so that the noise power is the same as in the first measurement.

```
./rx_samples_to_file --freq=2480000000 --rate=4000000 --gain=70 --nsamps
    =4000000 --filename=signal_70dB.dat
```

Figure 4.4 shows one instance of the second measurement, where two packets are clearly seen. The following equation is used to calculate the SNR from the two measurements.

$$\frac{S}{N} = \frac{P_{signal}}{P_{noise}} \tag{4.2}$$

$$\frac{E_b}{N_0} = \frac{S}{N} \tag{4.3}$$

$$E_{total} = \sum_{0}^{N-1} |x[n]|^2 \tag{4.4}$$
$$= P_{signal}N_{signal} + P_{noise}N_{noise}$$

where $N = N_{signal} + N_{noise}$ is the total number of samples, N_{signal} is the number of signal samples, and N_{noise} is the number of noise samples. In the first measurement, since there is no signal present, the equation can be re-written as the following.

$$E_{total} = \sum_{0}^{N-1} |x[n]|^2 \tag{4.5}$$
$$= P_{noise}N_{noise}$$
$$= P_{noise}N$$

Solving for P_{noise}, the noise power can be calculated.

$$P_{noise} = \frac{\sum |x[n]|^2}{N_{noise}} \tag{4.6}$$
$$= \sum |x[n]|^2/N$$

(4.7)

In Equation 4.4, N_{signal} is known because the number of packets and the length of each packet can be controlled. P_{signal} is then the following.

$$P_{signal} = \frac{E_{total} - P_{noise}N_{noise}}{N_{signal}} \tag{4.8}$$

Substituting P_{signal} and P_{noise} into Equation 4.2 gives the SNR.

Packet Detection Rate

The packet detection rate is calculated using the following formula.

$$Packet\ Detection\ Rate = \frac{Number\ of\ Packets\ Detected}{Number\ of\ Packets\ Transmitted} \tag{4.9}$$

Figure 4.5 shows the packet detection rates for the FPGA and GNU Radio implementations. The packet detection rate is higher for the GNU Radio implementation.

Packet Error Rate

The packet error rate is calculated using the following equation.

$$Packet\ Error\ Rate \tag{4.10}$$
$$= \frac{Number\ of\ Packets\ With\ Incorrect\ CRC\ + Number\ of\ Packets\ Not\ Detected}{Number\ of\ Packets\ Transmitted}$$

Figure 4.6 shows the plot of the packet error rates.

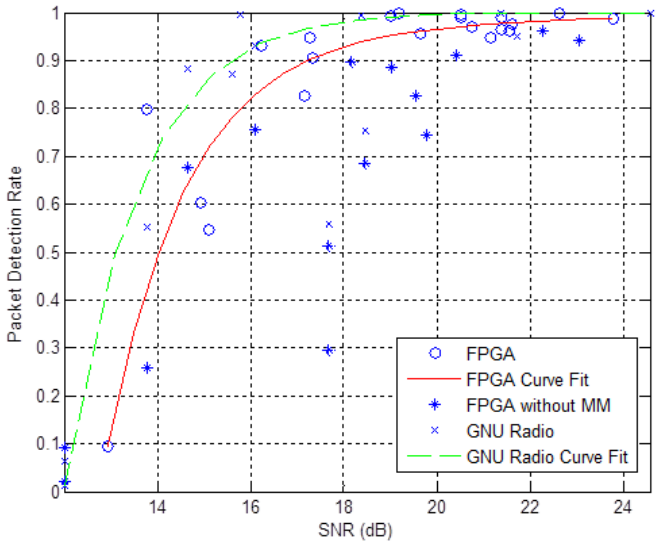

Figure 4.5: Percentages of Packets Detected

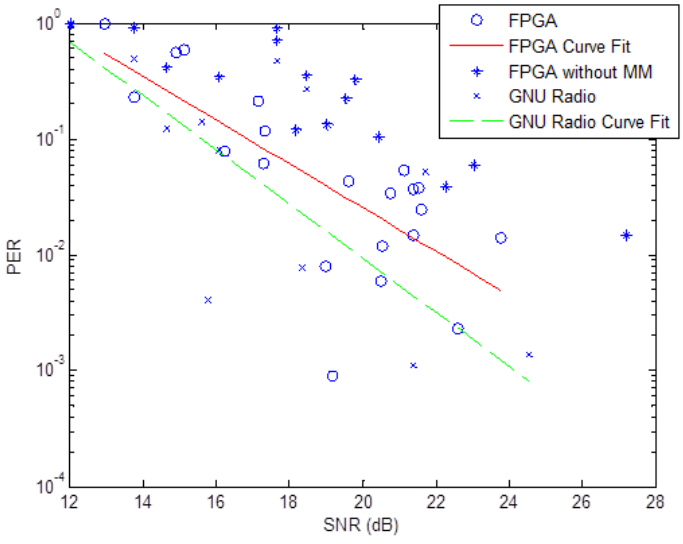

Figure 4.6: Packet Error Rate

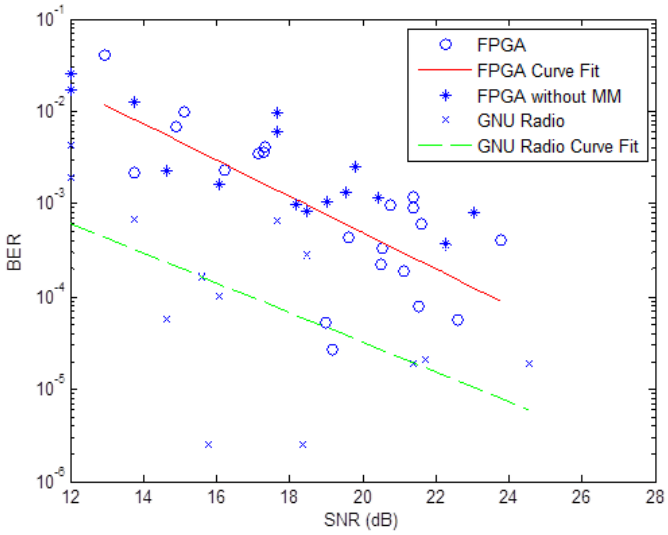

Figure 4.7: Bit Error Rate

Bit Error Rate

The bit error rate is calculated using the following formula. Figure 4.7 shows the bit error rate of GNU Radio and FPGA implementations.

$$Bit\ Error\ Rate = \frac{Number\ of\ Incorrectly\ Decoded\ Bits}{Number\ of\ Bits\ Received} \tag{4.11}$$

In the Packet Detection Rate and Packet Error Rate measurements, the GNU Radio implementation and the FPGA implementation with clock recovery give comparable results, while the FPGA implementation without clock recovery gives the worst result. In the BER measurements, the GNU Radio implementation outperforms both of the FPGA implementations. The FPGA implementation with clock recovery performs better than the one without clock recovery as expected.

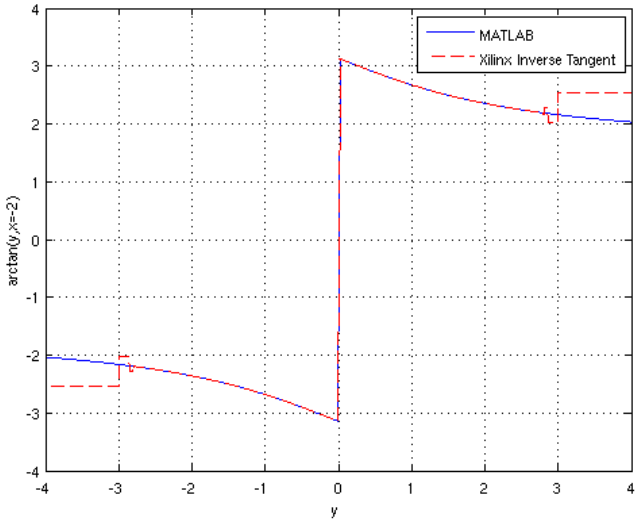

Figure 4.8: Output of Inverse Tangent with Correct Sign

There are multiple reasons why the GNU Radio implementation outperforms the FPGA implementation.

- Algorithmic

  The output of the inverse tangent in the FPGA implementation is intentionally set to zero when it is outside $[-\pi/2,\pi/2]$, with some headroom, to decrease the correlation value when noise is present. The rationale behind this is that when the signal is present, the output of inverse tangent should be within $[-\pi/4,\pi/4]$, and when only noise is present, the output of inverse tangent is outside $[-\pi/4,\pi/4]$. By setting the values outside the range to be zero, the correlation values are lower when only noise is present, thereby decreasing the false alarm rate. However, at low SNR, the output of inverse tangent is outside the range of $[-\pi/2,\pi/2]$ even when the signal is present. Therefore, the algorithm falsely decides that there is only noise present and

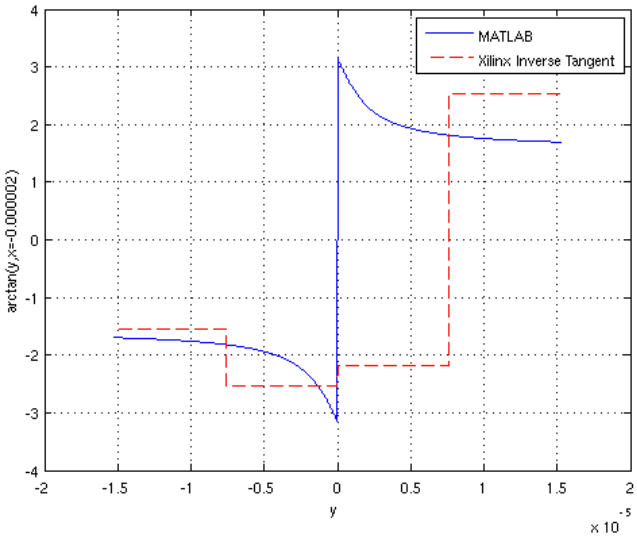

Figure 4.9: Output of Inverse Tangent with Mismatched Sign

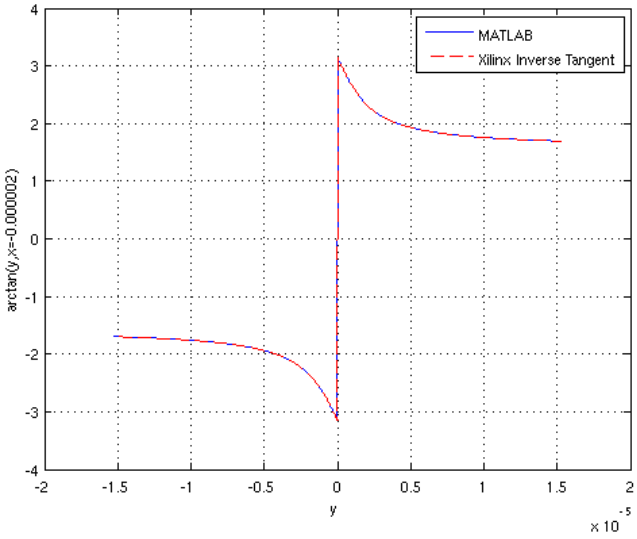

Figure 4.10: Output of Inverse Tangent with Increased CORDIC Iterations

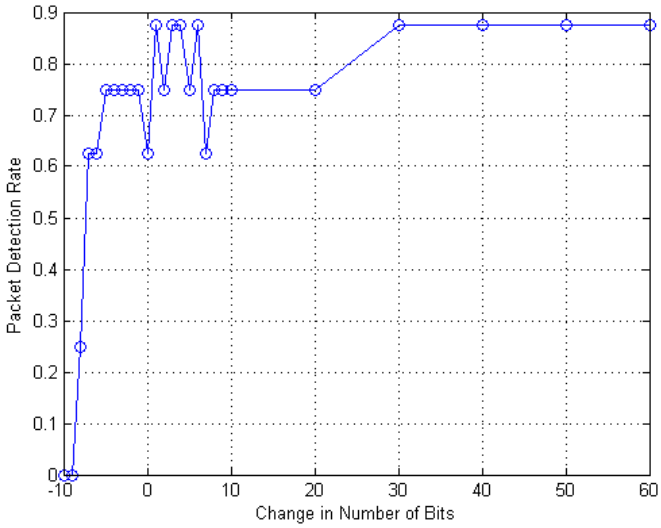

Figure 4.11: Packet Detection Rate with Varying Word Length

sets the output of inverse tangent to zero for some output samples. This decreases the correlation value and degrades the performance.

- Limited word-length

  It is observed that the output of the CORDIC inverse tangent block does not match that of the floating-point inverse tangent function. Figure 4.8 shows the output of inverse tangent when the x value is fixed at $x = -2$ while y is $[-4 : 4]$. This is a benign case where if the y values are big, the output of inverse tangent is not accurate, but the sign of the output is correct. Figure 4.9 shows the malignant case when y values are small, not only do the magnitudes mismatch, but also the signs of the output mistmatch. The wrong sign results in wrong chip decision, which degrades the performance of the clock recovery module and decreases the correlation values. The output of the inverse tangent block can be improved by increasing the

number of CORDIC iterations, however, at the expense of increased latency. Figure 4.10 shows that output values match very closely when the number of CORDIC interations are increased.

The limited word-length of the internal registers of the clock recovery module leads to a poorer performance. This is verified by simulating the clock recovery module with different word lengths. The input to the clock recovery module is the output of the floating-point inverse tangent to make sure that only the clock recovery module affects the performance. When the word lengths of the internal registers are increased, the number of packets detected are increased. However, when the word lengths are decreased, the number of packets detected decreased. Figure 4.11 shows the packet detection rate as the word length is increased or decreased. Since the clock recovery module is very sensitive to noise, slight decrease in word length, actually increases the detection rate for some range, but the detection rate decreases to zero eventually. Similarly, when the word length is increased, the detection rate decreases slightly at one point, but eventually increases and stabilizes at large word length.

4.4 Radio Characteristics

Figure 4.12 shows the power spectral density of the IEEE 802.15.4 waveform transmitted from the FPGA implementation. The IEEE 802.15.4 standard states that the relative power

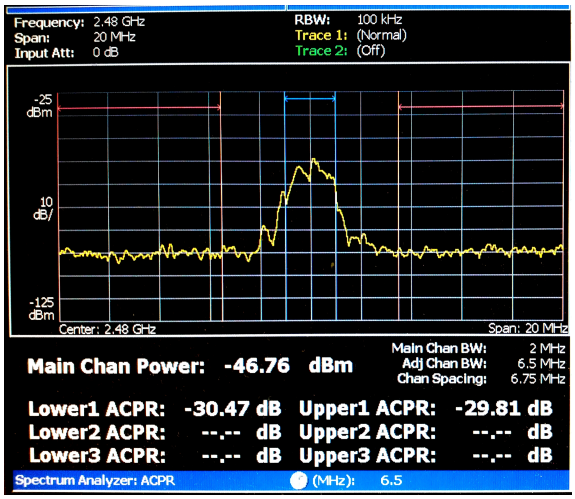

Figure 4.12: Spectral Mask of IEEE 802.15.4 transmitted using USRP N210 FPGA

in frequency $|f - f_c| > 3.5MHz$ shall be less than -20 dBr compared to the average spectrałpower measured within $\pm 1MHz$ of the carrier frequency. The measurement shall be made using a 100 kHz resolution bandwidth. Figure 4.12 shows that power in the upper and lower bands 3.5 MHz away from the center frequencies are -30.47 dB and -29.81 dB, which satisfies the requirement.

The standards states that the transmitted center frequency tolerance shall be $\pm 40ppm = 0.004\%$. Figure 4.13 shows that the frequency deviation is about $\frac{2.480008492\ GHz - 2.48\ GHz}{2.48\ GHz} = 0.0003\%$, which lies within the tolerance limit.

The 99% occupied bandwidth is approximately 2 MHz as shown in Figure 4.14.

Figure 4.15 shows the constellation plot of the O-QPSK signal transmitted by the USRP N210 FPGA implementation. As expected of O-QPSK signals, the constellation points do not travel across the origin.

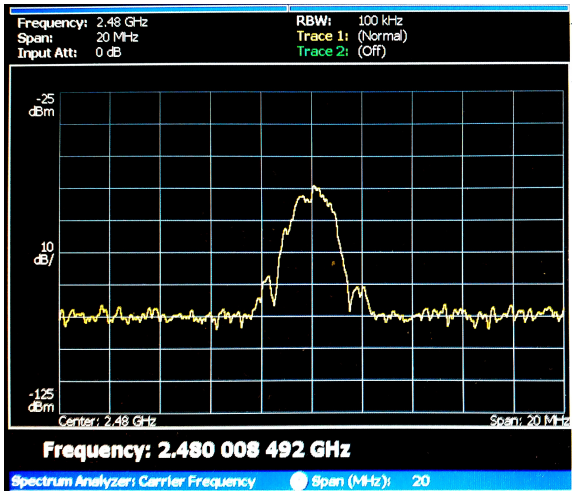

Figure 4.13: Center Frequency of IEEE 802.15.4 Signal from USRP N210 FPGA

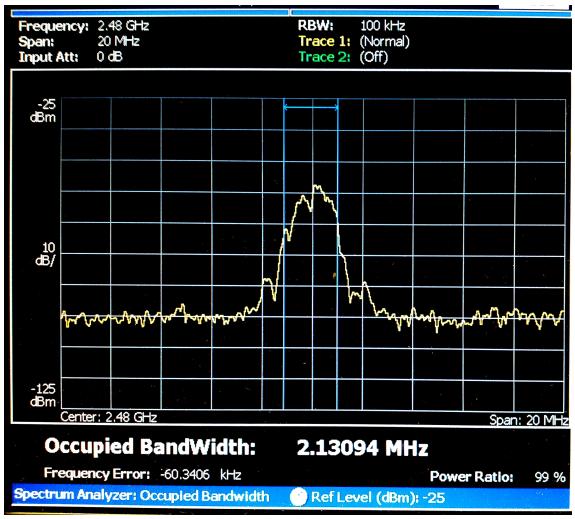

Figure 4.14: Occupied Bandwidth

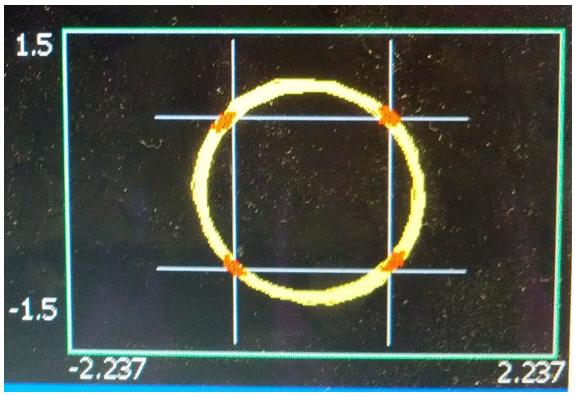

Figure 4.15: O-QPSK Constellation

Chapter 5

Conclusion and Future Work

Two different versions of IEEE 802.15.4 PHY implementations have been explored. The first implementation demonstrates how a signal processing application such as IEEE 802.15.4 PHY can be embedded inside the USRP N210's FPGA. This enables developers to take advantage of the spare resources in the USRP N210's FPGA and delegate complex signal processing tasks from slow GPP to fast FPGA. GNU Radio is now free to do much more processing since FPGA takes care of all the complex tasks.

The second implementation demonstrates the use of an external FPGA in the USRP and GNU Radio environment. By modifying the firmware inside the USRP N210, it is possible to redirect packets from USRP N210 to the external FPGA. The external FPGA then extracts the samples from the UDP packets and performs signal processing on the samples. Because external FPGAs can be much larger than USRP's FPGA, even more complex

tasks can be performed on it. The example implementation of the multi-channel IEEE 802.15.4 demonstrates the viability of the platform by successfully decoding packets from commercially available IEEE 802.15.4 node. If desired, a signal processing chain can be distributed among all three platforms, the USRP's FPGA, an external FPGA, and GNU Radio. This platform is also extensible to additional external FPGAs if desired. The first FPGA can process the samples from USRP and relay to the next FPGA, and so on, until the result reaches GNU Radio.

Future work includes implementing a more complex signal processing applications such as spectrum sensing algorithms or IEEE 802.11a.

Bibliography

[1] *IEEE Std 802.15.4-2006*, IEEE Std.

[2] M. P. Karve, "Evaluation of gnu radio platform enhanced for hardware accelerated radio design," Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2010.

[3] D. Bursky. (2006, May) Designers stretched in asic, fpga tug-of-war. [Online]. Available: http://www.eetimes.com/design/automotive-design/4004080/Designers-stretched-in-ASIC-FPGA-tug-of-war

[4] C. R. Irick, "Enhancing gnu radio for hardware accelerated radio design," Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2010.

[5] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker, "Sora: High performance software radio using general purpose multi-core processors," *NSDI 09: 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.

[6] A. T. Tran, D. N. Truong, and B. M. Baas, "A complete real-time 802.11a baseband receiver implemented on an array of programmable processors," *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, pp. 165–170, 2008.

[7] J. H. Reed, *Software Radio - A Modern Approach to Radio Engineering*. PEARSON Education, 2006.

[8] D. Lau, J. Blackburn, and J. A. Seely, "The use of hardware acceleration in sdr waveforms," Altera Corporation, Tech. Rep., 2005.

[9] Intel. (2003) Intel hyper-threading technology. Intel. [Online]. Available: http://cache-www.intel.com/cd/00/00/01/77/17705_htt_user_guide.pdf

[10] Fftw. http://www.fftw.org/.

[11] T. Rondeau. (2010, 12) Volk: Vector-optimized library of kernels. GNU Radio. [Online]. Available: http://www.trondeau.com/blog/2010/12/11/volk-vector-optimized-library-of-kernels.html

[12] T. Schmid, "Gnu radio 802.15.4 en- and decoding," NESL Department of Electrical Engineering University of California, Los Angeles, Tech. Rep., 2006.

[13] M. A. R. Saghir, P. Chow, and C. G. Lee. (1998) A comparison of traditional and vliw dsp architectures for compiled dsp applications. University of Toronto. [Online]. Available: www.eecg.toronto.edu/~saghir/papers/CASES98.ps

[14] D. Liu, *Embedded DSP Processor Design: Application Specific Instruction Set Processors*. Morgan Kaufmann, 2008.

[15] Altera. (2001) Fpgas provide reconfigurable dsp solutions. [Online]. Available: http://www.altera.com/literature/wp/wp_dsp_fpga.pdf

[16] Xilinx. (2012, January) Virtex-5 fpga xtremedsp design considerations. [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug193.pdf

[17] R. P. Brent, F. T. Luk, and C. V. Loan, "Computation of the singular value decomposition using mesh-connected processors," *Journal of VLSI and Computer Systems*, pp. 242–270, 1985.

[18] AccelChip. Using fpgas to improve the performance of radar, navigation and guidance systems. [Online]. Available: http://www.design-reuse.com/articles/11366/using-fpgas-to-improve-the-performance-of-radar-navigation-and-guidance-systems.html

[19] I. Berkeley Design Technology. (2010) High-level synthesis tools for xilinx fpgas. Berkeley Design Technology, Inc. [Online]. Available: www.bdti.com/MyBDTI/pubs/Xilinx_hlstcp.pdf

[20] ——. (2007, January) Fpgas vs. dsps: A look at the unanswered questions. [Online]. Available: http://www.eetimes.com/design/signal-processing-dsp/4017003/FPGAs-vs-DSPs-A-look-at-the-unanswered-questions/

[21] D. Gislason, *ZigBee Wireless Networking*. Newnes, 2008.

[22] E. Research. Ettus research. [Online]. Available: http://ettus.com/

[23] fredric j harris, *Multirate Signal Processing For Communication Systems*. Prentice Hall, 2004.

[24] J. Notor, A. Caviglia, and G. Levy. (2003) Cmos rfic architectures for ieee 802.15.4 networks. Cadence Design Systems, Inc. 6210 Old Dobbin Lane, Suite 100 Columbia, Maryland 21045, USA. [Online]. Available: http://chipbook.com/databook/Interface/ZIGBEE/CMOSRFICArchforIEEE80215.pdf

[25] H. Meyr, M. Moeneclaey, and S. A. Fechtel, *Digital Communication Receivers*. John Wiley & Sons, 1998.

[26] S. B. Wicker, *Error Control Systems*. Prentice Hall, 1995.

[27] K. Sankar. (2008, March) Frequency offset estimation using 802.11a short preamble. [Online]. Available: http://www.dsplog.com/2008/03/03/frequency-offset-estimation-using-80211a-short-preamble/

[28] MathWorks. Simulink hdl coder. MathWorks. [Online]. Available: http://www.mathworks.com/products/slhdlcoder/

[29] J. Blum. (2012, May) Usrp users mailing list – universal software radio peripheral by ettus research. [Online]. Available: http://comments.gmane.org/gmane.comp.hardware.usrp.e100/3719

Appendix A

Verilog Source Code for IEEE 802.15.4 Receiver on USRP N210's FPGA

A.1 Receiver Top Level

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:              16:41:48 12/17/2011
// Design Name:
// Filename:          xbee_rx.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:       Top file for the IEEE 802.15.4 receiver
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module xbee_rx
(
  input clk,                                // Clock 100 MHz
  input reset,                              // Reset
  input strobe_in,                          // 4 MHz strobe
  input signed [15:0] in_real,              // Input 16-bit real
  input signed [15:0] in_imag,              // Input 16-bit imaginary
  input signed [24:0] in_THP,               // Preamble Threshold
```

```
  input signed [24:0] in_THP_NEXT,          // Secondary Preamble Threshold
  input signed [22:0] in_THS,               // SFD Threshold
  output reg out_strobe_symbol,             // Symbol strobe
  output reg out_strobe_symbol_delayed,     // Delayed symbol strobe
  output reg out_strobe_byte,               // Delayed byte strobe
  output reg out_strobe_byte_pre,           // Byte strobe
  output reg [3:0] out_symbol,              // Decoded symbol
  output reg [7:0] out_byte,                // Decoded byte
  output reg [24:0] out_p_corr,             // Preamble correlation
  output reg [22:0] out_s_corr,             // SFD correlation
  output reg [3:0] out_state,               // MAC state
  output reg out_crc_correct,               // Strobe for correct CRC
  output [511:0] debug                      // Debug signals
);

// maximum index of findmax16
wire [15:0] index_max;

// delay conjugate multiply
wire signed [15:0] out_phase;   // phase output
wire out_phase_ready;           // valid when phase output is ready

// output of fifo between clock recovery and correlation blocks
wire signed [15:0] fifo_avg_out;

// output of fifo between preamble correlation and MAC state machine
wire signed [24:0] fifo_preamble_dout;

// output of coregen_fir_preamble
wire signed [24:0] preamble_dout;
wire preamble_rfd;
wire preamble_rdy;

// output of fifo between sfd correlation and MAC state machine
wire signed [24:0] _fifo_sfd_dout;
wire signed [22:0] fifo_sfd_dout;
wire fifo_sfd_full, fifo_sfd_empty;

// output of coregen_fir_sfd
wire signed [22:0] sfd_dout;
wire sfd_rfd, sfd_rdy;

// output of MAC state machine
wire strobe_sym_from_mac;
wire strobe_byte_from_mac;
wire strobe_byte_pre_from_mac;
wire [3:0] out_state_from_mac;
wire out_crc_correct_from_mac;
wire [7:0] out_byte_from_mac;
wire [15:0] out_crc_from_mac;

```

```
// Debug signals
wire [511:0] debug_mac;

// Generate 2MHz strobe for chips
cic_strober #(.WIDTH(8))   // output 2MHz clk
strober_2(
  .clock(clk),
  .reset(reset),
  .enable(1'b1),
  .rate(2),
  .strobe_fast(strobe_in),
  .strobe_slow(strobe_out_2mhz)
);

// registering output of MAC state machine
always @(posedge clk) begin
  if(reset) begin
    out_state <= 0;
    out_crc_correct <= 0;
    out_byte <= 0;
    out_strobe_symbol <= 0;
    out_strobe_byte <= 0;
    out_strobe_symbol_delayed <= 0;
  end
  else begin
    out_state <= out_state_from_mac;
    out_crc_correct <= out_crc_correct_from_mac;
    out_byte <= out_byte_from_mac;
    out_strobe_symbol <= strobe_sym_from_mac;
    out_strobe_byte_pre <= strobe_byte_pre_from_mac;
    out_strobe_byte <= strobe_byte_from_mac;
    out_strobe_symbol_delayed <= out_strobe_symbol;
  end
end

// Registering symbol output
always @(posedge clk) begin
  if(reset) begin
    out_symbol <= 0;
  end
  else begin
    if(strobe_sym_from_mac)
      out_symbol <= index_max[3:0];
  end
end

// AGC
wire [32:0] out_agc;
wire [20:0] debug_agc;
agc  agc(
  .clk(clk),
```

```
131   .rst(reset),
      .in({in_real, in_imag, strobe_in}), // input
133   .out(out_agc),                      // output
      .debug(debug_agc)                   // debug signals
135 );

137 // Delay-conjugate-multiply
    fmdemod fmdemod (
139   .clk(clk),
      .reset(reset),
141   .ce(out_agc[0]),                    // enable signal = valid from agc output
      .in_real(out_agc[32:17]),           // input real
143   .in_imag(out_agc[16:1]),            // input imag
      .in_real_valid(1'b1),               // input valid
145   .in_imag_valid(1'b1),               // input valid
      .out_phase(out_phase),              // output phase, fractional bits = 13
147   .out_phase_ready(out_phase_ready)   // output phase available
    );
149
    // Clock Recovery
151 wire [32:0] out_mm;
    clock_recovery_mm clock_recovery_mm  (
153   .clk(clk),
      .rst(reset),
155   .in({out_phase, 16'd0, strobe_in}/*out_agc*/),
      .out(out_mm)
157 );

159 // FIFO to store output of clock recovery
    coregen_fifo_512_16 coregen_fifo_avg_out (
161   .clk(clk),                                    // input clk
      .rst(reset),                                  // input rst
163   .din(out_mm[32:17]),                          // input [15 : 0] din
      .wr_en(~fifo_avg_out_full & out_mm[0]),       // input wr_en
165   .rd_en(~fifo_avg_out_empty & preamble_rfd),   // input rd_en
      .dout(fifo_avg_out),                          // output [15 : 0] dout
167   .full(fifo_avg_out_full),                     // output full
      .empty(fifo_avg_out_empty)                    // output empty
169 );

171 // Preamble correlation
    coregen_fir_preamble coregen_fir_preamble (
173   .clk(clk),            // 100MHz
      .rfd(preamble_rfd),   // ready for data
175   .rdy(preamble_rdy),   // output data ready
      .din(fifo_avg_out),   // [15:0]
177   .dout(preamble_dout)  // [24:0]
    );
179
    // FIFO to store preamble correlation
181 coregen_fifo_512_25 coregen_fifo_25_preamble (
```

```
    .clk(clk),                                      // input clk
    .rst(reset),                                    // input rst
    .din(preamble_dout),                            // input [24 : 0] din
    .wr_en(~fifo_preamble_full & preamble_rdy),     // input wr_en
    .rd_en(~fifo_preamble_empty & strobe_out_2mhz),// input rd_en
    .dout(fifo_preamble_dout),                      // output [24 : 0] dout
    .full(fifo_preamble_full),                      // output full
    .empty(fifo_preamble_empty)                     // output empty
);

// SFD correlation
coregen_fir_sfd coregen_fir_sfd (
    .clk(clk),
    .rfd(sfd_rfd),
    .rdy(sfd_rdy),
    .din(fifo_avg_out),
    .dout(sfd_dout) // [22:0]
);

// FIFO to store SFD correlation
assign fifo_sfd_dout = _fifo_sfd_dout[22:0];
coregen_fifo_512_25 coregen_fifo_25_sfd (
    .clk(clk),                                  // input clk
    .rst(reset),                                // input rst
    .din({{2{sfd_dout[22]}},sfd_dout}),         // input [24 : 0] din
    .wr_en(~fifo_sfd_full & sfd_rdy),           // input wr_en
    .rd_en(~fifo_sfd_empty & strobe_out_2mhz),  // input rd_en
    .dout(_fifo_sfd_dout),                      // output [24 : 0] dout
    .full(fifo_sfd_full),                       // output full
    .empty(fifo_sfd_empty)                      // output empty
);

// Symbol correlations
wire fir_bank_sym_out_rfd;
wire signed [21:0] fir_bank_sym_out_0;
wire signed [21:0] fir_bank_sym_out_1;
wire signed [21:0] fir_bank_sym_out_2;
wire signed [21:0] fir_bank_sym_out_3;
wire signed [21:0] fir_bank_sym_out_4;
wire signed [21:0] fir_bank_sym_out_5;
wire signed [21:0] fir_bank_sym_out_6;
wire signed [21:0] fir_bank_sym_out_7;
wire signed [21:0] fir_bank_sym_out_8;
wire signed [21:0] fir_bank_sym_out_9;
wire signed [21:0] fir_bank_sym_out_10;
wire signed [21:0] fir_bank_sym_out_11;
wire signed [21:0] fir_bank_sym_out_12;
wire signed [21:0] fir_bank_sym_out_13;
wire signed [21:0] fir_bank_sym_out_14;
wire signed [21:0] fir_bank_sym_out_15;
```

```
// Symbol Correlations bank
fir_bank_sym fir_bank_sym (
  .clk(clk),
  .reset(reset),
  .strobe_in(strobe_out_2mhz),     // Strobe at input chip rate
  .strobe_out(strobe_out_2mhz),    // Strobe at output correlation rate
  .din(fifo_avg_out),              // Input chips
  .out_rfd(fir_bank_sym_out_rfd), // Ready-for-data of correlation blocks
  .out_0(fir_bank_sym_out_0),      // Correlation value for symbol 0
  .out_1(fir_bank_sym_out_1),      // Correlation value for symbol 1
  .out_2(fir_bank_sym_out_2),      // Correlation value for symbol 2
  .out_3(fir_bank_sym_out_3),      // Correlation value for symbol 3
  .out_4(fir_bank_sym_out_4),      // Correlation value for symbol 4
  .out_5(fir_bank_sym_out_5),      // Correlation value for symbol 5
  .out_6(fir_bank_sym_out_6),      // Correlation value for symbol 6
  .out_7(fir_bank_sym_out_7),      // Correlation value for symbol 7
  .out_8(fir_bank_sym_out_8),      // Correlation value for symbol 8
  .out_9(fir_bank_sym_out_9),      // Correlation value for symbol 9
  .out_10(fir_bank_sym_out_10),    // Correlation value for symbol 10
  .out_11(fir_bank_sym_out_11),    // Correlation value for symbol 11
  .out_12(fir_bank_sym_out_12),    // Correlation value for symbol 12
  .out_13(fir_bank_sym_out_13),    // Correlation value for symbol 13
  .out_14(fir_bank_sym_out_14),    // Correlation value for symbol 14
  .out_15(fir_bank_sym_out_15)     // Correlation value for symbol 15
);

// Findmax 16
findmax16 findmax16 (
  .clk(clk),
  .ce(strobe_out_2mhz),
  .reset(reset),
  .value0(fir_bank_sym_out_0),
  .value1(fir_bank_sym_out_1),
  .value2(fir_bank_sym_out_2),
  .value3(fir_bank_sym_out_3),
  .value4(fir_bank_sym_out_4),
  .value5(fir_bank_sym_out_5),
  .value6(fir_bank_sym_out_6),
  .value7(fir_bank_sym_out_7),
  .value8(fir_bank_sym_out_8),
  .value9(fir_bank_sym_out_9),
  .value10(fir_bank_sym_out_10),
  .value11(fir_bank_sym_out_11),
  .value12(fir_bank_sym_out_12),
  .value13(fir_bank_sym_out_13),
  .value14(fir_bank_sym_out_14),
  .value15(fir_bank_sym_out_15),
  .index_out(index_max),
  .value_out(),
  .passthrough()
);
```

```
285 // MAC
    reg [24:0] delay_p[0:2];
287 reg [22:0] delay_s[0:2];
    reg [24:0] in_p_corr;
289 reg [22:0] in_s_corr;

291 // need to delay preamble and sfd by 4 2mhz clock cycles because of findmax's
        latency
    always @(posedge clk)
293 begin
      if(reset) begin
295     in_p_corr <= 0;
        in_s_corr <= 0;
297     delay_p[0] <= 0;
        delay_p[1] <= 0;
299     delay_p[2] <= 0;
        delay_s[0] <= 0;
301     delay_s[1] <= 0;
        delay_s[2] <= 0;
303   end
      else if(strobe_out_2mhz) begin
305     delay_p[2] <= fifo_preamble_dout;
        delay_p[1] <= delay_p[2];
307     delay_p[0] <= delay_p[1];
        in_p_corr <= delay_p[0];
309     delay_s[2] <= fifo_sfd_dout;
        delay_s[1] <= delay_s[2];
311     delay_s[0] <= delay_s[1];
        in_s_corr <= delay_s[0];
313   end
    end
315
    // MAC state machine
317 xbee_mac xbee_mac (
      .clk(clk),
319   .reset(reset),
      .strobe_in(strobe_out_2mhz),                // 2mhz
321   .in_p_corr(in_p_corr),                      // preamble correlation
      .in_s_corr(in_s_corr),                      // SFD correlation
323   .in_sym(index_max[3:0]),                    // symbol
      .strobe_sym(strobe_sym_from_mac),           // output strobe 62.5khz
325   .strobe_byte(strobe_byte_from_mac),         // output strobe 31.25khz
      .strobe_byte_pre(strobe_byte_pre_from_mac),// output strobe 31.25khz
327   .out_byte(out_byte_from_mac),               // output byte
      .out_crc(out_crc_from_mac),                 // output crc
329   .state(out_state_from_mac),                 // output state
      .out_crc_correct(out_crc_correct_from_mac),// output crc strobe
331   .debug(debug_mac),                          // debug
      .in_THP(in_THP),                            // preamble threshold
333   .in_THP_NEXT(in_THP_NEXT),                  // secondary preamble threshold
```

```
    .in_THS(in_THS)                                 // SFD threshold
);

// Register preamble correlation
always @(posedge clk)
begin
  if(reset)
    out_p_corr <= 0;
  else
    out_p_corr <= fifo_preamble_dout;
end

// Register SFD correlation
always @(posedge clk)
begin
  if(reset)
    out_s_corr <= 0;
  else
    out_s_corr <= fifo_sfd_dout;
end

// Debug signals
assign debug = {debug_agc, in_real, in_imag, strobe_in, out_agc};

endmodule  // xbee_rx
```

src/rx/xbee_rx.v

A.2 Strober

```
//
//  USRP2 - Universal Software Radio Peripheral Mk II
//
//  Copyright (C) 2008 Matt Ettus
//
//  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 2 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, write to the Free Software
//  Foundation, Inc., 51 Franklin Street, Boston, MA  02110-1301  USA
```

```
//

module cic_strober
  #(parameter WIDTH=8)
    ( input clock,
      input reset,
      input enable,
      input [WIDTH-1:0] rate, // Rate should EQUAL to your desired divide
          ratio, no more -1 BS
      input strobe_fast,
      output wire strobe_slow );

   reg [WIDTH-1:0] counter;
   wire      now = (counter==1);
   assign    strobe_slow = now && enable && strobe_fast;

   always @(posedge clock)
     if(reset)
       counter <= 8;
     else if (~enable)
       counter <= rate;
     else if(strobe_fast)
       if(now)
   counter <= rate;
       else
   counter <= counter - 1;

endmodule // cic_strober
```

src/rx/cic_strober.v

A.3 AGC

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:      Wireless @ VT, Virginia Tech CCM Lab
// Author:       Jeong-O Jeong
//
// Date:            17:37:41 05/14/2012
// Design Name:
// Filename:        agc.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:     Automatic Gain Control based on GNU Radio's 'agc2' block
//
// Dependencies:    mult - Xilinx core 16x16 multiplier
//                  sqrt - Xilinx core square root
```

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module agc
#(  parameter REFERENCE = 16'd512,     // 0.5 (10 decimal places)
parameter ATTACK_RATE = 16'd32,      // 1e-3 (15 decimal places)
parameter DECAY_RATE = 16'd32767,       // 0.99 (15 decimal places)
parameter INITIAL_GAIN = 16'd16384,  // 4.0 (12 decimal places)
parameter MAX_GAIN = 21'd262144      // 64.0 (12 decimal places)
)
(
  input clk,
  input rst,
  input [32:0] in,        // 16-bit I, 16-bit Q, valid
  output reg [32:0] out,  // 16-bit I, 16-bit Q, valid
  output [127:0] debug    // debug signal
);

wire signed [31:0] prod_error_rate; // 25 decimal places
reg signed [15:0] error;            // 10 decimal places
reg signed [15:0] rate;             // 15 decimal

// registered inputs
reg signed [15:0] in_i;
reg signed [15:0] in_q;
reg in_valid;

// register inputs
always @(posedge clk) begin
  if(rst) begin
    in_i <= 0;
    in_q <= 0;
  end
  else begin
    if(in[0]) begin
      in_i <= in[32:17];
      in_q <= in[16:1];
    end
    in_valid <= in[0];
  end
end

// set gain
wire signed [20:0] gain; // 21bits, 12 fractional

// pipeline registers for rdy_error_rate = rdy signal from sqrt block
reg rdy_error_rate[0:2];
```

```
// error * rate
wire signed [39:0] prod_error_rate_adjusted;
assign prod_error_rate_adjusted = {{3{prod_error_rate[31]}},prod_error_rate,5'
    d0}; // need to make 25 -> 30 fractional

reg signed [39:0] full_gain; // 30 decimal places

// gain - (error * rate)
wire signed [39:0] diff_gain_prod_error_rate;
assign diff_gain_prod_error_rate = (full_gain - prod_error_rate_adjusted);

// maximum gain
wire signed [39:0] max_gain;
assign max_gain = {MAX_GAIN,18'd0};

always @(posedge clk) begin
  if(rst) begin
    full_gain <= {INITIAL_GAIN,18'd0}; //18 = 30 decimal - 12 decimal places
  end
  else begin
    if(rdy_error_rate[2]) begin // when rate*error signal is ready
      if(diff_gain_prod_error_rate > max_gain)
        full_gain <= max_gain;
      else if(diff_gain_prod_error_rate < 0)
        full_gain <= 1; // small number as per GNU Radio code
      else
        full_gain <= diff_gain_prod_error_rate;
    end
  end
end

// truncate gain to be 21 bits with 12 fractional
assign gain = full_gain[38:18];
wire signed [36:0] out_i;  // 27 decimal places
wire signed [36:0] out_q;  // 27 decimal places

// multiply inputs by gain to produce outputs
mult16x21 mult16x21_out_i (
  .clk(clk),  // input clk
  .a(in_i),   // input [15 : 0] a (15 decimal places)
  .b(gain),   // input [20 : 0] b (12 decimal places)
  .ce(1'b1),  // input ce
  .p(out_i)   // output [36 : 0] p (27 decimal places)
);
mult16x21 mult16x21_out_q (
  .clk(clk),  // input clk
  .a(in_q),   // input [15 : 0] a
  .b(gain),   // input [20 : 0] b
  .ce(1'b1),  // input ce
  .p(out_q)   // output [36 : 0] p
);
```

```
117
reg signed [15:0] out_i_hardlimit; // 11 fractional bits
119 reg signed [15:0] out_q_hardlimit;
parameter ONE_FRAC_27 = 32'd134217727;      // 1, with 27 fractional bits
121 parameter NEG_ONE_FRAC_27 = -32'd134217728; // -1, with 27 fractional bits

123 // limit range of out_i and out_q
always @(posedge clk) begin
125   if(rst)
      out_i_hardlimit <= 0;
127   else if(out_i[31]==1'b0 && out_i > ONE_FRAC_27) // if positive, and greater
        than 0.9999
      out_i_hardlimit <= ONE_FRAC_27[31:16];
129   else if(out_i[31]==1'b1 && out_i < NEG_ONE_FRAC_27) // if negative, and
        smaller than -1
      out_i_hardlimit <= NEG_ONE_FRAC_27[31:16];
131   else
      out_i_hardlimit <= out_i[31:16];
133 end

135 always @(posedge clk) begin
   if(rst)
137     out_q_hardlimit <= 0;
   else if(out_q[31]==1'b0 && out_q > ONE_FRAC_27) // if positive, and greater
        than 0.9999
139     out_q_hardlimit <= ONE_FRAC_27[31:16];
   else if(out_q[31]==1'b1 && out_q < NEG_ONE_FRAC_27) // if negative, and
        smaller than -1
141     out_q_hardlimit <= NEG_ONE_FRAC_27[31:16];
   else
143     out_q_hardlimit <= out_q[31:16];
end
145


147 // pipeline valid signals
reg valid_prod_in_gain[3:0];
149 reg valid_out_square[4:0];

151 always @(posedge clk) begin
   if(rst) begin
153     valid_prod_in_gain[0] <= 0;
     valid_prod_in_gain[1] <= 0;
155     valid_prod_in_gain[2] <= 0;
     valid_prod_in_gain[3] <= 0;
157     valid_out_square[0] <= 0;
     valid_out_square[1] <= 0;
159     valid_out_square[2] <= 0;
     valid_out_square[3] <= 0;
161     valid_out_square[4] <= 0;
   end
163   else begin
```

```
    valid_prod_in_gain[0] <= in_valid;
    valid_prod_in_gain[1] <= valid_prod_in_gain[0];
    valid_prod_in_gain[2] <= valid_prod_in_gain[1];
    valid_prod_in_gain[3] <= valid_prod_in_gain[2];
    valid_out_square[0] <= valid_prod_in_gain[3];
    valid_out_square[1] <= valid_out_square[0];
    valid_out_square[2] <= valid_out_square[1];
    valid_out_square[3] <= valid_out_square[2];
    valid_out_square[4] <= valid_out_square[3];
  end
end

// real*real, imag*imag
wire [31:0] prod_i; // 22 decimal places
wire [31:0] prod_q; // 22 decimal places

mult mult16x16_i (
  .clk(clk), // input clk
  .a(out_i_hardlimit), // input [15 : 0] a
  .b(out_i_hardlimit), // input [15 : 0] b
  .ce(1'b1), // input ce
  .p(prod_i) // output [31 : 0] p
);
mult mult16x16_q (
  .clk(clk), // input clk
  .a(out_q_hardlimit), // input [15 : 0] a
  .b(out_q_hardlimit), // input [15 : 0] b
  .ce(1'b1), // input ce
  .p(prod_q) // output [31 : 0] p
);

parameter ONE_FRAC_15 = 16'd32767;      // 1, with 15 fractional bits
parameter NEG_ONE_FRAC_15 = -16'd32768; //-1, with 15 fractional bits

// limit range of final output
always @(posedge clk) begin
  if(rst)
    out[32:17] <= 0;
  else if(out_i[31]==1'b0 && out_i > ONE_FRAC_27) // if positive, and greater
     than 0.9999
    out[32:17] <= ONE_FRAC_27[27:12];
  else if(out_i[31]==1'b1 && out_i < NEG_ONE_FRAC_27) // if negative, and
     smaller than -1
    out[32:17] <= NEG_ONE_FRAC_27[27:12];
  else
    out[32:17] <= out_i[27:12];
end

// Register output
always @(posedge clk) begin
  if(rst)
```

```
213     out[16:1] <= 0;
    else if(out_q[31]==1'b0 && out_q > ONE_FRAC_27) // if positive, and greater
        than 0.9999
215     out[16:1] <= ONE_FRAC_27[27:12];
    else if(out_q[31]==1'b1 && out_q < NEG_ONE_FRAC_27) // if negative, and
        smaller than -1
217     out[16:1] <= NEG_ONE_FRAC_27[27:12];
    else
219     out[16:1] <= out_q[27:12];
end

221
// Register output valid
223 always @(posedge clk) begin
    if(rst)
225     out[0] <= 0;
    else
227     out[0] <= valid_out_square[4];
end
229
// take sqrt of the sum to get signal magnitude
231 wire signed [32:0] sum;         // 22 decimal places
assign sum=prod_i+prod_q;
233
wire rdy;
235 wire signed [15:0] signal_mag; // 10 decimal places

237 sqrt sqrt(
    .x_in(sum),                  // input [32 : 0] x_in
239 .nd(valid_out_square[4]),     // nd
    .x_out(signal_mag),          // output [15 : 0] x_out
241 .rdy(rdy),                   // output rdy
    .clk(clk),                   // input clk
243 .ce(1'b1)                    // input ce
);
245
// error = magnitude - reference
247 always @(posedge clk) begin
    if(rst) begin
249     error <= 0; // 10 decimal places
    end
251 else begin
      if(rdy) begin
253     error <= signal_mag - REFERENCE;
      end
255 end
end
257
// pick ATTACK or DECAY rate
259 wire signed [20:0] error_adjusted; // match gain = 21 bits, 12 fractional
assign error_adjusted = {{3{error[15]}},error,2'd0};
261
```

```
always @(posedge clk) begin
  if(rst) begin
    rate <= 0;
  end
  else begin
    if(rdy) begin
      if(error_adjusted > gain) // since error has 10 decimal and gain has 12
          decimal
        rate <= ATTACK_RATE;
      else
        rate <= DECAY_RATE;
    end
  end
end

// multiply rate and error
// prod_error_rate 25 fractional bits
mult mult_error_rate (
  .clk(clk), // input clk
  .a(error), // input [15 : 0] a // 10 decimal
  .b(rate), // input [15 : 0] b // 15 decimal
  .ce(1'b1), // input ce
  .p(prod_error_rate) // output [31 : 0]
);

// rdy_error_rate[0] - high when sqrt is calculated
// rdy_error_rate[1] - high when error is calculated
// rdy_error_rate[2] - high when prod_error_rate is calculated (error * rate)
always @(posedge clk) begin
  if(rst) begin
    rdy_error_rate[0] <= 0;
    rdy_error_rate[1] <= 0;
    rdy_error_rate[2] <= 0;
  end
  else begin
    rdy_error_rate[0] <= rdy;
    rdy_error_rate[1] <= rdy_error_rate[0];
    rdy_error_rate[2] <= rdy_error_rate[1];
  end
end

assign debug = gain;
endmodule
```

src/rx/agc.v

A.4 Delay-conjugate-multiply

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:              16:41:48 12/17/2011
// Design Name:
// Filename:          fmdemod.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:       Delay-conjugate-multiply block
//
// Dependencies:      Requires Xilinx CoreGen complex multiply and arctan
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module fmdemod
#(parameter WIDTH=16)
(
  input clk,
  input reset,
  input ce,
  input signed [WIDTH-1:0] in_real,        // Input 16-Bit real
  input signed [WIDTH-1:0] in_imag,        // Input 16-Bit imag
  input in_real_valid,                     // Valid for input
  input in_imag_valid,                     // Valid for input
  output reg signed [WIDTH-1:0] out_phase,// Output phase
  output reg out_phase_ready               // Valid when out_phase is available
);

// Delayed input values and valids
reg signed [WIDTH-1:0] in_real_delayed;
reg signed [WIDTH-1:0] in_imag_delayed;
reg in_real_valid_delayed;
reg in_imag_valid_delayed;

// Output of coregen_complex_mult
wire signed [32:0] out_real;
wire signed [32:0] out_imag;

// Output of coregen_arctan
wire signed [15:0] phase_out;

// Output of FIFO
wire [15:0] dout;
wire empty;
```

```
52 // Delay input
   always @(posedge clk) begin
54   if(reset) begin
       in_real_delayed <= 0;
56     in_imag_delayed <= 0;
       in_real_valid_delayed <= 0;
58     in_imag_valid_delayed <= 0;
     end
60   else if(ce) begin
       in_real_delayed <= in_real;
62     in_imag_delayed <= in_imag;
       in_real_valid_delayed <= in_real_valid;
64     in_imag_valid_delayed <= in_imag_valid;
     end
66 end

68 // complexMult
   // latency 6
70 coregen_complex_mult coregen_complex_mult (
     .clk(clk),                // input clk
72   .ce(ce),                  // input ce
     .ar(in_real),             // input [15 : 0] ar
74   .ai(in_imag),             // input [15 : 0] ai
     .br(in_real_delayed),     // input [15 : 0] br
76   .bi(-in_imag_delayed),    // input [15 : 0] bi
     .pr(out_real),            // ouput [32 : 0] pr
78 .pi(out_imag));           // ouput [32 : 0] pi

80 // inverse tangent core
   // latency : 20
82 coregen_arctan coregen_arctan (
     .clk(clk),                // input clk
84   .ce(ce),                  // input ce
     .x_in(out_real),          // input [32 : 0] x_in
86   .y_in(out_imag),          // input [32 : 0] y_in
     .phase_out(phase_out)     // output [15 : 0] phase_out (fractional bits=13)
88 );

90 // output values
   always @(posedge clk) begin
92   if(reset) begin
       out_phase <= 0;
94     out_phase_ready <= 0;
     end
96   else if(ce) begin
       out_phase <= dout;
98     out_phase_ready <= ~empty;
     end
100 end
```

```
// set to zero if outside [-pi/2:pi/2]
reg signed [15:0] clipped_phase_out;
reg signed [15:0] upperlimit = 16'd12868;   // 12868 = pi/2
reg signed [15:0] lowerlimit = 16'd52668;   // -pi/2

always @(posedge clk) begin
  if(reset) begin
    clipped_phase_out <= 0;
  end
  else if(ce) begin
    // if outside [-pi/2:pi/2] set to zero
    if(phase_out > upperlimit || phase_out < lowerlimit)
      clipped_phase_out <= 16'd0;
    else
      clipped_phase_out <= phase_out;
  end
end

// output fifo
coregen_fifo coregen_fifo (
  .rst(reset),              // input rst
  .wr_clk(clk),             // input wr_clk
  .rd_clk(clk),             // input rd_clk
  .din(clipped_phase_out),  // input [15 : 0] din
  .wr_en(~full & ce),       // input wr_en
  .rd_en(~empty & ce),      // input rd_en
  .dout(dout),              // output [15 : 0] dout
  .full(full),              // output full
  .empty(empty)             // output empty
);

endmodule  // delayConjMult
```

src/rx/fmdemod.v

A.5 Clock Recovery

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:          Wireless @ VT
// Author:           Jeong-O Jeong
//
// Create Date:     11:33:56 07/11/2012
// Design Name:
// Module Name:     clock_recovery_mm
// Project Name:
// Target Devices:
// Tool versions:
```

```
// Description:      Mueller and Muller based clock recovery
//                   Based on GNU Radio digital clock recovery mm block
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module clock_recovery_mm(
  input clk,
  input rst,
  input [32:0] in,      // 16-bit I, 16-bit Q, valid
  output [32:0] out     // 16-bit I, 16-bit Q, valid
);

parameter MU            = 32'd134217728;// 0.5, 28 fractional
parameter GAIN_OMEGA    = 16'd14;       // 2.25e-4 unsigned, 0 integer, 16
    fractional
parameter GAIN_MU       = 16'd1966;     // 0.03, unsigned, 0 integer, 16
    fractional

parameter OMEGA         = 32'd536870912; // 2, unsigned, 28 fractional
parameter MIN_OMEGA     = 32'd536763537; // unsigned, 28 fractional
parameter MAX_OMEGA     = 32'd536978286; // unsinged, 28 fractional

// interpolator ready counter
reg [3:0] rdy_counter;

// count the number of samples in memory
reg [3:0] total_elements;

// counter in state OMEGA
reg [1:0] state_counter_omega;

// State machine
parameter WAIT          = 3'd0;
parameter INTERPOLATE   = 3'd1;
parameter COMPUTE_MM    = 3'd2;
parameter COMPUTE_OMEGA = 3'd3;
parameter COMPUTE_INCR  = 3'd4;
parameter COMPUTE_II    = 3'd5;
(* FSM_ENCODING="SEQUNTIAL", SAFE_IMPLEMENTATION="NO" *)
reg [2:0] state = INTERPOLATE;

// state machine
always@(posedge clk)
  if (rst) begin
    state <= WAIT;
  end
```

```
61    else begin
      case (state)
63      WAIT : begin
          if(total_elements > 10)
65          state <= INTERPOLATE;
          else
67          state <= WAIT;
        end
69      INTERPOLATE : begin
          if(rdy_counter == 7) // interpolation is finished with 8 input samples
              ;
71          // 7, because rdy_counter changes to 8 one clock cycle after the 8th
                output is computed
            state <= COMPUTE_MM;
73        else
            state <= INTERPOLATE;
75      end
        COMPUTE_MM : begin
77        if(1'b1)
            state <= COMPUTE_OMEGA;
79        else
            state <= COMPUTE_MM;
81      end
        COMPUTE_OMEGA : begin
83        if (state_counter_omega == 1)
            state <= COMPUTE_INCR;
85        else
            state <= COMPUTE_OMEGA;
87      end
        COMPUTE_INCR : begin
89        if (1'b1)
            state <= COMPUTE_II;
91        else
            state <= COMPUTE_INCR;
93      end
        COMPUTE_II : begin
95        if (1'b1)
            state <= WAIT;
97        else
            state <= COMPUTE_II;
99      end
      endcase
101   end

103   // input registers
      reg signed [15:0] i_reg;
105   reg valid;

107   // fractional offset
      reg signed [31:0] mu;       // 28 fractional
109   // samples per symbol
```

```
  reg signed [31:0] omega;   // 28 fractional
111 // error
  reg signed [15:0] mm_val;  // signed 4 integer, 12 fractional
113


115 // Interpolator wires
  wire rfd, rdy;
117
  // register inputs (only real value needed)
119 always @(posedge clk) begin
    if(rst) begin
121   i_reg <= 0;
    end
123 else if(in[0]) begin
      i_reg <= in[32:17];
125 end
    valid <= in[0];
127 end

129 // wires needed for 2 port block ram
  reg [3:0] addr_write;
131 reg [3:0] ii;           // pointer to start of read location
  reg [3:0] addr_read;    // actual memory read index
133 wire [15:0] dout_mem;
  reg valid_mem;
135
  reg [31:0] incr; // increment for read index, 28 fractional bit
137
  // total elements counter
139 always @(posedge clk) begin
    if(rst) begin
141   total_elements <= 0;
    end
143 else if(valid) begin
      total_elements <= total_elements + 1;
145 end
    if(state == COMPUTE_II) begin
147   total_elements <= total_elements - incr[31:28];
    end
149 end

151 // write address
  always @(posedge clk) begin
153 if(rst) begin
      addr_write <= 1'b0;
155 end
    // whenever there is a new data increment write addr
157 else if(valid) begin
      addr_write <= addr_write + 1;
159 end
  end
```

```
161
    // prepare for multiplication
163 wire signed [31:0] _gain_mu;
    wire signed [31:0] _gain_omega;
165 wire signed [31:0] _mm_val;
    assign _gain_omega = {16'b0, GAIN_OMEGA};
167 assign _gain_mu = {16'b0, GAIN_MU};
    assign _mm_val = {{16{mm_val[15]}}, mm_val};
169
    reg signed [31:0] prod_gainMu_mmVal; // 28 fractional
171 always @(posedge clk) begin
      if(rst) begin
173     prod_gainMu_mmVal <= 0;
      end
175   else begin
        // 16 fractional * 12 fractional = 28 fractional
177     prod_gainMu_mmVal <= _gain_mu * _mm_val;
      end
179 end

181 // increment for read index, 28 fractional bit
    always @(posedge clk) begin
183   if(rst) begin
        incr <= 0;
185   end
      else if(state == COMPUTE_INCR) begin
187     //        28 +  28   + 28 fractional
        incr <= mu + omega + prod_gainMu_mmVal;
189   end
    end
191
    wire [3:0] incr_int;
193 assign incr_int = incr[31:28];

195 // pointer to start index
    always @(posedge clk) begin
197   if(rst) begin
        ii <= 0;
199   end
      else if(state == COMPUTE_II) begin
201     ii <= ii + incr[31:28];
      end
203 end

205 // read address
    always @(posedge clk) begin
207   if(rst) begin
        addr_read <= 0;
209   end
      else if(state == INTERPOLATE) begin
```

```
211       if(rfd && ~rdy) // using ~rdy because it goes high 8 cycles later since
              first new data
            addr_read <= addr_read + 1'b1;
213     end
        else  // set to ii
215       addr_read <= ii;
    end
217
    // rdy counter
219 always @(posedge clk) begin
      if(rst) begin
221     rdy_counter <= 0;
      end
223   else if(state == INTERPOLATE) begin
        if(rdy)
225       rdy_counter <= rdy_counter + 1;
      end
227   else
        rdy_counter <= 0;
229 end

231 core_mem core_mem(
      // port a
233   .clka(clk), // input clka
      .wea(valid), // input [0 : 0] wea
235   .ena(~rst),
      .addra(addr_write), // input [3 : 0] addra
237   .dina(i_reg), // input [15 : 0] dina
      .douta(), // output [15 : 0] douta
239   // port b
      .clkb(clk), // input clkb
241   .enb(~rst),
      .web(1'b0), // input [0 : 0] web
243   .addrb(addr_read), // input [3 : 0] addrb
      .dinb(16'b0), // input [15 : 0] dinb
245   .doutb(dout_mem) // output [15 : 0] doutb
    );
247
    wire signed [31:0] dout; // fractional 30
249 reg signed [15:0] outputs[0:1]; // 14 fractional bits

251 // store previous and current outputs of the interpolator
    always @(posedge clk) begin
253   if(rst) begin
        outputs[0] <= 0;
255     outputs[1] <= 0;
      end
257   else if(rdy_counter == 7) begin
        outputs[0] <= dout[31:16]; // 14 fractional bits
259     outputs[1] <= outputs[0];
      end
```

```
261   end

263   //assign out = {dout[30:15], 16'b0, rdy_counter == 7};
      assign out = {dout[31:16], 16'b0, rdy_counter == 7};
265
      wire signed [15:0] sample_current; // 14 fractional bits
267   wire signed [15:0] sample_last;
      assign sample_current = outputs[0];
269   assign sample_last =outputs[1];

271   // mm error, 12 fractional bits
      wire signed [15:0] mm_val_0, mm_val_1, mm_val_2, mm_val_3;
273   assign mm_val_0 = -sample_current + sample_last; // 14 fractional
      assign mm_val_1 = sample_current + sample_last;
275   assign mm_val_2 = -sample_current - sample_last;
      assign mm_val_3 = sample_current - sample_last;
277
      // compute all four cases and pick one
279   always @(posedge clk) begin
        if(rst) begin
281       mm_val <= 0;
        end
283     else if(state == COMPUTE_MM) begin
          if(sample_last < 0 && sample_current < 0)
285         mm_val <= {{2{mm_val_0[15]}},mm_val_0[15:2]};
          else if(sample_last > 0 && sample_current < 0)
287         mm_val <= {{2{mm_val_1[15]}},mm_val_1[15:2]};
          else if(sample_last < 0 && sample_current > 0)
289         mm_val <= {{2{mm_val_2[15]}},mm_val_2[15:2]};
          else
291         mm_val <= {{2{mm_val_3[15]}},mm_val_3[15:2]};
        end
293   end

295   // OMEGA state counter
      always @(posedge clk) begin
297     if(rst) begin
          state_counter_omega <= 0;
299     end
        else if(state == COMPUTE_OMEGA) begin
301       state_counter_omega <= state_counter_omega + 1'b1;
        end
303     else
          state_counter_omega <= 0;
305   end

307   reg signed [31:0] prod_gainOmega_mmVal; // 28 fractional
      // GAIN_OMEGA * mm_val
309   always @(posedge clk) begin
        if(rst) begin
311       prod_gainOmega_mmVal <= 0;
```

```
    end
313   else begin
      // 16 fractional * 12 fractional = 28 fractional
315   prod_gainOmega_mmVal <= _gain_omega * _mm_val;
    end
317 end

319 wire [32:0] omega_preclip = omega + prod_gainOmega_mmVal;

321 // omega = samples per symbol, 28 fractional
  always @(posedge clk) begin
323   if(rst) begin
      omega <= OMEGA;
325   end
    else if(state == COMPUTE_OMEGA & state_counter_omega == 1) begin
327   if(omega_preclip > MAX_OMEGA)
        omega <= MAX_OMEGA;
329   else if(omega_preclip < MIN_OMEGA)
        omega <= MIN_OMEGA;
331   else
        omega <= omega_preclip;
333 end
  end
335
  // fractional offset, 28 fractional
337 always @(posedge clk) begin
    if(rst) begin
339   mu <= MU;
    end
341 else if(state == COMPUTE_II) begin
      mu <= {incr[27:0]};// fractional portion of incr
343 end
  end
345
  wire [7:0] filter_sel;
347 wire signed [31:0] round_mu;
  wire signed [31:0] half;
349 assign half = 1'b1<<20;
  assign round_mu = mu + half;
351 assign filter_sel = round_mu[28:21];

353 // regsiter to indicate if the state is interpolate
  reg reg_state_interpolate;
355 always @(posedge clk) begin
    if(rst)
357   reg_state_interpolate <= 0;
    else
359   reg_state_interpolate <= (state == INTERPOLATE);
  end
361
  wire nd;
```

```
363   assign nd = reg_state_interpolate & ~rdy & (state == INTERPOLATE);

365   // 8-tap interpolator
      core_interpolator core_interpolator(
367     .sclr(~reg_state_interpolate),  // input sclr
        .clk(clk),                      // input clk
369     .ce(~rst),                      // input ce
        .nd(nd),                        // input nd
371     .filter_sel(filter_sel),        // input [7 : 0] filter_sel
        .rfd(rfd),                      // ouput rfd
373     .rdy(rdy),                      // ouput rdy
        .din(dout_mem),                 // input [15 : 0] din
375   .dout(dout));                     // ouput [31 : 0] dout

377 endmodule
```

src/rx/clock_recovery_mm.v

A.6 Symbol Correlation

```
1  `timescale 1ns / 1ps
   //////////////////////////////////////////////////////////////////////////////////
3  // Company:        Wireless @ VT
   // Engineer:       Jeong-O Jeong
5  //
   // Create Date:     01:14:33 12/11/2011
7  // Design Name:
   // Module Name:     fir_bank_sym
9  // Project Name:
   // Target Devices:
11 // Tool versions:
   // Description:    Find the symbol correlations of 16 symbols
13 //
   // Dependencies:   CoreGen FIR and FIFO
15 //
   // Revision:
17 // Revision 0.01 - File Created
   // Additional Comments:
19 //
   //////////////////////////////////////////////////////////////////////////////////
21 module fir_bank_sym
   #(parameter WIDTH=16, parameter N=16)
23 (
     input clk,
25   input reset,
     input strobe_in,         // 2Mhz strobe
27   input strobe_out,        // 2MHz strobe
     input [WIDTH-1:0] din,   // input chip
```

```
29    output out_rfd,
      output [21:0] out_0,     // symbol correlation output 0
31    output [21:0] out_1,
      output [21:0] out_2,
33    output [21:0] out_3,
      output [21:0] out_4,
35    output [21:0] out_5,
      output [21:0] out_6,
37    output [21:0] out_7,
      output [21:0] out_8,
39    output [21:0] out_9,
      output [21:0] out_10,
41    output [21:0] out_11,
      output [21:0] out_12,
43    output [21:0] out_13,
      output [21:0] out_14,
45    output [21:0] out_15     // symbol correlation output 15
);
47
genvar i;
49
wire rfd[0:15];
51 wire rdy[0:15];
wire [21:0] dout[0:15];
53 wire fifo_full_sym[0:15];
wire fifo_empty_sym[0:15];
55 wire [24:0] fifo_dout_sym[0:15];

57 assign out_rfd = rfd[0];


59
// output signals
61 assign out_0 = fifo_dout_sym[0][21:0];
assign out_1 = fifo_dout_sym[1][21:0];
63 assign out_2 = fifo_dout_sym[2][21:0];
assign out_3 = fifo_dout_sym[3][21:0];
65 assign out_4 = fifo_dout_sym[4][21:0];
assign out_5 = fifo_dout_sym[5][21:0];
67 assign out_6 = fifo_dout_sym[6][21:0];
assign out_7 = fifo_dout_sym[7][21:0];
69 assign out_8 = fifo_dout_sym[8][21:0];
assign out_9 = fifo_dout_sym[9][21:0];
71 assign out_10 = fifo_dout_sym[10][21:0];
assign out_11 = fifo_dout_sym[11][21:0];
73 assign out_12 = fifo_dout_sym[12][21:0];
assign out_13 = fifo_dout_sym[13][21:0];
75 assign out_14 = fifo_dout_sym[14][21:0];
assign out_15 = fifo_dout_sym[15][21:0];
77
// generate 16 correlation FIR filters
79 generate
```

```
for(i=0;i<N;i=i+1) begin : coregen_fir_sym
  coregen_fir_sym_0 coregen_fir_sym(
    .clk(clk), // input clk
    .filter_sel(i), // input [3 : 0] filter_sel
    .rfd(rfd[i]), // ouput rfd
    .rdy(rdy[i]), // ouput rdy
    .din(din), // input [15 : 0] din
  .dout(dout[i])); // ouput [21 : 0] dout
end
endgenerate

// generate 16 FIFO's
generate
for(i=0;i<N;i=i+1) begin : fifo_sym
  coregen_fifo_512_25 coregen_fifo_25_sym (
    .clk(clk), // input clk
    .rst(reset), // input rst
    .din({{3{dout[i][21]}},dout[i]}), // input [24 : 0] din
    .wr_en(~fifo_full_sym[i] & rdy[i]), // input wr_en
    .rd_en(~fifo_empty_sym[i] & strobe_out), // input rd_en
    .dout(fifo_dout_sym[i]), // output [24 : 0] dout
    .full(fifo_full_sym[i]), // output full
    .empty(fifo_empty_sym[i]) // output empty
  );
end
endgenerate

endmodule
```

src/rx/fir_bank_sym.v

A.7 Find Max 16-input

```
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:              16:39:48 12/17/2011
// Design Name:
// Filename:          findmax.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:       find maximum between two values
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
`timescale 1 ns / 1 ns

module findmax
#(parameter WIDTH=22)
(
  clk,
  ce,
  reset,
  index1,   // index of first element
  value1,   // value of first element
  index2,   // index of second element
  value2,   // value of second element
  index_out,// index of max element
  value_out // value of max element
);

input   clk;
input   ce;
input   reset;
input   [15:0] index1;  // ufix16
input   [15:0] index2;  // ufix16
input   signed [WIDTH-1:0] value1;  // sfix16_En14
input   signed [WIDTH-1:0] value2;  // sfix16_En14

output  reg [15:0] index_out;  // ufix16
output  reg [WIDTH-1:0] value_out;  // sfix16_En14

// output the index of max element
always @(posedge clk)
begin
  if(reset) begin
    index_out = 0;
  end
  else begin
    if(ce)
      index_out = (value1 > value2) ? index1 : index2;
  end
end

// output the value of max element
always @(posedge clk)
begin
  if(reset)
    value_out = 0;
  else
    if(ce)
      value_out = (value1 > value2) ? value1 : value2;
```

```
67 end

69 endmodule  // findmax
```

src/rx/findmax.v

A.8 Find Max 2-input

```
`timescale 1 ns / 1 ns
2 //////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
4 // Author:       Jeong-O Jeong
//
6 // Date:            16:39:59 12/17/2011
// Design Name:
8 // Filename:        findmax16.v
// Project Name:
10 // Target Devices:
// Tool versions:
12 // Description:     find the maximum among 16 different values
//
14 // Dependencies:
//
16 // Revision:
// Revision 0.01 - File Created
18 // Additional Comments:
//
20 //////////////////////////////////////////////////////////////////////////////////
module findmax16
22 #(parameter WIDTH=22)
(
24   input clk,
   input ce,
26   input reset,
   input [WIDTH-1:0] value0,
28   input [WIDTH-1:0] value1,
   input [WIDTH-1:0] value2,
30   input [WIDTH-1:0] value3,
   input [WIDTH-1:0] value4,
32   input [WIDTH-1:0] value5,
   input [WIDTH-1:0] value6,
34   input [WIDTH-1:0] value7,
   input [WIDTH-1:0] value8,
36   input [WIDTH-1:0] value9,
   input [WIDTH-1:0] value10,
38   input [WIDTH-1:0] value11,
   input [WIDTH-1:0] value12,
40   input [WIDTH-1:0] value13,
```

```
    input [WIDTH-1:0] value14,
    input [WIDTH-1:0] value15,
    output [15:0] index_out,
    output signed [WIDTH-1:0] value_out,
    output reg [WIDTH-1:0] passthrough
);

// max index of 8 values when comparing 16 values
wire [15:0] index_1_1;
wire [15:0] index_1_2;
wire [15:0] index_1_3;
wire [15:0] index_1_4;
wire [15:0] index_1_5;
wire [15:0] index_1_6;
wire [15:0] index_1_7;
wire [15:0] index_1_8;

// max index of 4 values when comparing 8 values
wire [15:0] index_2_1;
wire [15:0] index_2_2;
wire [15:0] index_2_3;
wire [15:0] index_2_4;

// max index of 2 values when comparing 4 values
wire [15:0] index_3_1;
wire [15:0] index_3_2;

// max 8 values when comparing 16 values
wire signed[WIDTH-1:0] value_1_1;
wire signed[WIDTH-1:0] value_1_2;
wire signed[WIDTH-1:0] value_1_3;
wire signed[WIDTH-1:0] value_1_4;
wire signed[WIDTH-1:0] value_1_5;
wire signed[WIDTH-1:0] value_1_6;
wire signed[WIDTH-1:0] value_1_7;
wire signed[WIDTH-1:0] value_1_8;

// max 4 values when comparing 8 values
wire signed[WIDTH-1:0] value_2_1;
wire signed[WIDTH-1:0] value_2_2;
wire signed[WIDTH-1:0] value_2_3;
wire signed[WIDTH-1:0] value_2_4;

// max 2 values when comparing 4 values
wire signed[WIDTH-1:0] value_3_1;
wire signed[WIDTH-1:0] value_3_2;

// 1st stage
// value0, value1
findmax findmax1(clk, ce, reset, 16'd0 , value0, 16'd1, value1, index_1_1,
    value_1_1);
```

```
findmax findmax2(clk, ce, reset, 16'd2 , value2, 16'd3, value3, index_1_2,
    value_1_2);
findmax findmax3(clk, ce, reset, 16'd4 , value4, 16'd5, value5, index_1_3,
    value_1_3);
findmax findmax4(clk, ce, reset, 16'd6 , value6, 16'd7, value7, index_1_4,
    value_1_4);
findmax findmax5(clk, ce, reset, 16'd8 , value8, 16'd9, value9, index_1_5,
    value_1_5);
findmax findmax6(clk, ce, reset, 16'd10 ,value10, 16'd11, value11, index_1_6,
    value_1_6);
findmax findmax7(clk, ce, reset, 16'd12 ,value12, 16'd13, value13, index_1_7,
    value_1_7);
findmax findmax8(clk, ce, reset, 16'd14 ,value14, 16'd15, value15, index_1_8,
    value_1_8);

// 2nd stage
findmax findmax9(clk, ce, reset, index_1_1, value_1_1, index_1_2, value_1_2,
    index_2_1, value_2_1);
findmax findmax10(clk, ce, reset, index_1_3, value_1_3, index_1_4, value_1_4,
    index_2_2, value_2_2);
findmax findmax11(clk, ce, reset, index_1_5, value_1_5, index_1_6, value_1_6,
    index_2_3, value_2_3);
findmax findmax12(clk, ce, reset, index_1_7, value_1_7, index_1_8, value_1_8,
    index_2_4, value_2_4);

// 3rd stage
findmax findmax13(clk, ce, reset, index_2_1, value_2_1, index_2_2, value_2_2,
    index_3_1, value_3_1);
findmax findmax14(clk, ce, reset, index_2_3, value_2_3, index_2_4, value_2_4,
    index_3_2, value_3_2);

//4th stage
findmax findmax15(clk, ce, reset, index_3_1, value_3_1, index_3_2, value_3_2,
    index_out, value_out);

// pass through for debug
always @(posedge clk)
begin
  if(reset)
    passthrough <= 0;
  else
    if(ce)
      passthrough <= value0;
end

endmodule  // findmax16
```

src/rx/findmax16.v

A.9 CRC-16

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:              16:39:35 12/17/2011
// Design Name:
// Filename:          crc16.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:       CRC-16 for IEEE 802.15.4
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module crc16
(
  clk,
  ce,
  reset,
  input_bit,     // input bit
  output_crc     // output crc
);

input clk;
input ce;
input reset;
input input_bit;

output reg [15:0] output_crc;

wire s0, s1, s2;

// shift registers
always @(posedge clk or posedge reset) // asynchronous reset
begin
  if(reset) begin
    output_crc <= 0;
  end
  else if(ce) begin
    output_crc[0] <= output_crc[1];
    output_crc[1] <= output_crc[2];
    output_crc[2] <= output_crc[3];
```

```
    output_crc[3] <= s1;
    output_crc[4] <= output_crc[5];
    output_crc[5] <= output_crc[6];
    output_crc[6] <= output_crc[7];
    output_crc[7] <= output_crc[8];
    output_crc[8] <= output_crc[9];
    output_crc[9] <= output_crc[10];
    output_crc[10] <= s2;
    output_crc[11] <= output_crc[12];
    output_crc[12] <= output_crc[13];
    output_crc[13] <= output_crc[14];
    output_crc[14] <= output_crc[15];
    output_crc[15] <= s0;
  end
end

xor(s0,input_bit,output_crc[0]);
xor(s1,s0,output_crc[4]);
xor(s2,s0,output_crc[11]);

endmodule  // crc16
```

src/rx/crc16.v

A.10 MAC State Machine

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:             16:39:59 12/17/2011
// Design Name:
// Filename:         xbee_mac.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:    state machine for detecting and decoding the received
    packet
//                 requires premble and SFD correlation for packet detection
//                 outputs decoded byte and strobe for valid CRC
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
//////////////////////////////////////////////////////////////////////////////////

module xbee_mac
(
  input clk,
  input reset,
  input strobe_in,                  // strobe @ 2mhz
  input signed [24:0] in_p_corr,    // preamble correlation
  input signed [22:0] in_s_corr,    // SFD correlation
  input [3:0] in_sym,               // symbol
  input signed [24:0] in_THP,       // preamble threshold
  input signed [24:0] in_THP_NEXT,  // secondary preamble threshold
  input signed [22:0] in_THS,       // SFD threshold
  output strobe_sym,                // symbol strobe @ 62.5k,
  output reg strobe_byte,           // delayed byte strobe @ 31.25k,
  output strobe_byte_pre,           // byte strobe 31.25khz
  output reg [7:0] out_byte,        // byte output
  output reg [15:0] out_crc,        // recieved crc OTA
  output out_crc_correct,           // high for correct CRC
  output reg [3:0] state,           // MAC FSM state
  output [511:0] debug              // debug port
);


// STATES
localparam FIND_PREAMBLE        = 4'd0; // search for preamble
localparam FIND_PEAK_PREAMBLE   = 4'd1; // search for peak preamble
localparam FIND_SFD             = 4'd2; // found a preamble
localparam DECODE_FRAMELENGTH   = 4'd3; // found a valid SFD
localparam DECODE_MHR           = 4'd4;
localparam DECODE_SYMBOLS       = 4'd5;
localparam DECODE_CRC_1ST_HALF  = 4'd6;
localparam DECODE_CRC_2ND_HALF  = 4'd7;
localparam CHECK_CRC            = 4'd8;

// control signals
reg [15:0] counter_since_preamble_found;  // interval between preamble and SFD
reg [15:0] counter_since_sfd_found;       // interval between SFD and first
   symbol
reg [7:0] framelength;

wire strobe_crc;
wire strobe_preamble;

cic_strober #(.WIDTH(8))   // output 2MHz/32 clk = 62.5k symbols/s
strober_3(
  .clock(clk),
  .reset(reset),
  .enable(state == DECODE_FRAMELENGTH || state == DECODE_MHR || state ==
     DECODE_SYMBOLS  || state == DECODE_CRC_1ST_HALF || state ==
     DECODE_CRC_2ND_HALF || state == CHECK_CRC),
```

```
    .rate(32),
    .strobe_fast(strobe_in),
    .strobe_slow(strobe_sym)
);

cic_strober #(.WIDTH(8))  // output 62.5k sym/s * 4 bit/sym = 250kbit/s
// 250kbits/s = 31.25B/s
strober_4 (
    .clock(clk),
    .reset(reset),
    .enable(state == DECODE_SYMBOLS || state == DECODE_MHR || state ==
        DECODE_CRC_1ST_HALF || state == DECODE_CRC_2ND_HALF || state ==
        CHECK_CRC),
    .rate(2),
    .strobe_fast(strobe_sym),
    .strobe_slow(strobe_byte_pre)
);

cic_strober #(.WIDTH(8))   // strober @ 250khz
strober_5(
    .clock(clk),
    .reset(reset),
    .enable(state == DECODE_MHR || state == DECODE_SYMBOLS || state ==
        DECODE_CRC_1ST_HALF),
    .rate(8),
    .strobe_fast(strobe_in),
    .strobe_slow(strobe_crc)
);

cic_strober #(.WIDTH(8))   // output 2MHz/32 clk = 62.5k for finding peak
    preamble
strober_6(
    .clock(clk),
    .reset(reset),
    .enable(state == FIND_PEAK_PREAMBLE),
    .rate(32),
    .strobe_fast(strobe_in),
    .strobe_slow(strobe_preamble)
);

// byte strobe
always @(posedge clk) begin
    if(reset) begin
        strobe_byte <= 0;
    end
    else begin
        strobe_byte <= strobe_byte_pre;
    end
end

// Preamble and SFD thresholds
```

```
117 wire signed [24:0] THP;
    wire signed [24:0] THP_NEXT;
119 wire signed [22:0] THS;   // 130*(pi/4)*2^13

121 assign THP = in_THP;
    assign THP_NEXT = in_THP_NEXT;
123 assign THS = in_THS;

125 // control signals for FSM
    wire [15:0] decode_symbols_limit;
127 wire [15:0] decode_crc_1st_half_limit;
    wire [15:0] decode_crc_2nd_half_limit;
129 wire [15:0] check_crc_limit;

131 assign decode_symbols_limit = 64 + ((framelength-2)*2 * 32);
    assign decode_crc_1st_half_limit = 64 + 32 + ((framelength-2)*2 * 32);
133 assign decode_crc_2nd_half_limit = 64 + ((framelength)*2 * 32);
    assign check_crc_limit = 64 + ((framelength)*2 * 32) + 64;
135
    //=========================================================================
137 // FSM
    //=========================================================================
139 always @(posedge clk) begin
      if (reset) begin
141     state <= FIND_PREAMBLE;
      end
143   else begin
        if(strobe_in) begin //   2mhz strobe
145       case (state)
            FIND_PREAMBLE:
147           if(in_p_corr > THP) begin
                state <= FIND_SFD;
149           end
              else begin
151             state <= FIND_PREAMBLE;
              end
153
            FIND_SFD:
155           if(in_s_corr > THS)
                state <= DECODE_FRAMELENGTH;
157           else if(counter_since_preamble_found > 16'd256)
                state <= FIND_PREAMBLE;
159           else
                state <= FIND_SFD;
161
            DECODE_FRAMELENGTH:
163           if(counter_since_sfd_found <= 64) // This will always be false at
                 some point
                state <= DECODE_FRAMELENGTH;
165           else
                state <= DECODE_MHR;
```

```
167
          DECODE_MHR:
169         if(counter_since_sfd_found <= 64 + 704 /* =11(bytes)*2(sym/byte)*32(
                chips/sym) */)
              state <= DECODE_MHR;
171         else
              state <= DECODE_SYMBOLS;
173
          DECODE_SYMBOLS:
175         if(counter_since_sfd_found <= decode_symbols_limit)
              state <= DECODE_SYMBOLS;
177         else
              state <= DECODE_CRC_1ST_HALF;
179
        DECODE_CRC_1ST_HALF:
181       if(counter_since_sfd_found <= decode_crc_1st_half_limit)
            state <= DECODE_CRC_1ST_HALF;
183       else
            state <= DECODE_CRC_2ND_HALF;
185
        DECODE_CRC_2ND_HALF:
187       if(counter_since_sfd_found <= decode_crc_2nd_half_limit)
            state <= DECODE_CRC_2ND_HALF;
189       else
            state <= CHECK_CRC;
191
        CHECK_CRC:
193       if(counter_since_sfd_found <= check_crc_limit) begin
            state <= CHECK_CRC;
195       end
          else begin
197         state <= FIND_PREAMBLE;
          end
199     endcase
      end
201 end
end
203
//=========================================================
205 // counters between preamble and SFD states
//=========================================================
207 // start counting if preamble found
always @(posedge clk)
209 begin
  if(reset) begin
211   counter_since_preamble_found <= 9'd0;
  end
213 else if(strobe_in) begin // 2mhz
    begin
215   if(state == FIND_SFD)
        counter_since_preamble_found <= counter_since_preamble_found + 9'd1;
```

```
217       else
            counter_since_preamble_found <= 9'd0;
219     end
      end
221 end

223 // start counting if sfd found
    always @(posedge clk)
225 begin
      if(reset)
227     counter_since_sfd_found <= 0;
      else if(strobe_in) begin // 2mhz
229     if(state == DECODE_FRAMELENGTH || state == DECODE_MHR || state ==
          DECODE_SYMBOLS || state == DECODE_CRC_1ST_HALF || state ==
          DECODE_CRC_2ND_HALF || state == CHECK_CRC)
          counter_since_sfd_found <= counter_since_sfd_found + 1;
231     else
          counter_since_sfd_found <= 0;
233   end
    end
235

    // save framelength
237 always @(posedge clk)
    begin
239   if(reset)
        framelength <= 8'b0111_1111;
241   else if(strobe_sym & state == DECODE_FRAMELENGTH) begin
        framelength[7:4] <= in_sym;
243     framelength[3:0] <= framelength[7:4];
      end
245

      // fix framelength if bigger than max allowed
247   if(framelength >= 8'b0111_1111 && state == DECODE_MHR)
        framelength <= 8'b0111_1111;
249 end

251 // save byte
    always @(posedge clk)
253 begin
      if(reset)
255     out_byte[7:0] <= 0;
      else if(strobe_sym & (state == DECODE_SYMBOLS || state == DECODE_MHR)) begin
257     out_byte[7:4] <= in_sym;
        out_byte[3:0] <= out_byte[7:4];
259   end
    end
261

    // save CRC
263 always @(posedge clk)
    begin
265   if(reset)
```

```
    out_crc[15:0] <= 0;
  else if(strobe_sym & (state == DECODE_CRC_1ST_HALF || state ==
      DECODE_CRC_2ND_HALF)) begin
    out_crc[15:12] <= in_sym;
    out_crc[11:8] <= out_crc[15:12];
    out_crc[7:4] <= out_crc[11:8];
    out_crc[3:0] <= out_crc[7:4];
  end
end

// count between symbols
reg [1:0] counter_between_symbols;

always @(posedge clk)
begin
  if(reset)
    counter_between_symbols <= 0;
  else if((state == DECODE_MHR || state == DECODE_SYMBOLS || state ==
      DECODE_CRC_1ST_HALF)) begin
    if(strobe_crc) begin
      counter_between_symbols <= counter_between_symbols + 1;
    end
  end
end

// register symbol
reg [3:0] persistent_sym;
always @(posedge clk) begin
  if(reset)
    persistent_sym <= 0;
  else if((state == DECODE_MHR || state == DECODE_SYMBOLS)) begin
    if(strobe_sym) begin
      persistent_sym <= in_sym[3:0];
    end
  end
  else if(state == FIND_PREAMBLE || state == FIND_SFD)
    persistent_sym <= 0;
end

// delayed strobe_crc for the multiplexer
reg strobe_crc_delayed;
always @(posedge clk) begin
  if(reset) begin
    strobe_crc_delayed <= 0;
  end
  else
    strobe_crc_delayed <= strobe_crc;
end

reg input_bit;
// multiplexer
```

```
315 always @(counter_between_symbols) begin
    if(reset) begin
317   input_bit <= 0;
    end
319 else if((state == DECODE_MHR || state == DECODE_SYMBOLS || state ==
      DECODE_CRC_1ST_HALF)) begin
    if(strobe_crc_delayed) begin
321   if(counter_between_symbols == 0)
        input_bit <= persistent_sym[0];          // get it from input
323   else if(counter_between_symbols == 1)
        input_bit <= persistent_sym[1];          // get it from
            symbol_persistent
325   else if(counter_between_symbols == 2)
        input_bit <= persistent_sym[2];
327   else if(counter_between_symbols == 3)
        input_bit <= persistent_sym[3];
329 end
    end
331 else
    input_bit <= 0;
333 end

335 // CRC16
    wire [15:0] output_crc;
337 crc16 u_crc16(
    .clk(clk),
339 .ce(strobe_crc_delayed & (state == DECODE_MHR || state == DECODE_SYMBOLS ||
      state == DECODE_CRC_1ST_HALF)),
    .reset(reset || state == FIND_PREAMBLE || state == FIND_SFD),
341 .input_bit(input_bit),
    .output_crc(output_crc)
343 );

345 // strobe if CRC is correct
    assign out_crc_correct = (state == CHECK_CRC && (out_crc == output_crc));
347
    // debug port
349 assign debug = {out_crc,output_crc};

351 endmodule
```

src/rx/xbee_mac.v

Appendix B

Verilog Source Code for IEEE 802.15.4 Transmitter on USRP N210's FPGA

B.1 Top Level Transmitter

```
1  `timescale 1 ns / 1 ns
   //////////////////////////////////////////////////////////////////////////////////
3  // Company:       Wireless @ VT
   // Author:        Jeong-O Jeong
5  //
   // Date:              10/30/2011
7  // Design Name:
   // Filename:          zb_ieee_802_15_4_mod.v
9  // Project Name:
   // Target Devices:
11 // Tool versions:
   // Description:       Top file for IEEE 802.15.4 Transmitter
13 //
   // Dependencies:
15 //
   // Revision:
17 // Revision 0.01 - File Created
   // Additional Comments:
19 //
   //////////////////////////////////////////////////////////////////////////////////
21
   module zb_ieee_802_15_4_mod
23 #(parameter WIDTH=32)
   (
25   input clk,
     input ce,
27   input rst,
     input [7:0] byte,                  // input byte to be modulated
```

```
29    output [WIDTH-1:0] qpsk_pulse,     // baseband qpsk pulse
      output [WIDTH-1:0] oqpsk_pulse,    // baseband oqpsk pulse
31    input strobe_in,                   // input strobe @ 4 MHz
      output strobe_tx,                  // output strobe @ byte rate
33    output strobe_out_qpsk_sym,        // qpsk symbol strobe
      output strobe_out_chips            // chip strobe
35  );


37
  wire [3:0] bitpos;
39  wire [31:0] pulse;
  wire [31:0] pulse_delayed;
41  wire [1:0] chunk;
  wire [31:0] chip;
43  wire [15:0] iphase, qphase, iphase_delayed, qphase_delayed;
  wire [WIDTH-1:0] output_data;
45  wire [WIDTH-1:0] symbol;

47  assign qpsk_pulse = pulse;
  assign oqpsk_pulse = pulse_delayed;
49
  assign iphase = pulse[31:16];
51  assign qphase = pulse[15:0];
  assign iphase_delayed = pulse_delayed[31:16];
53  assign qphase_delayed = pulse_delayed[15:0];

55  wire strobe_out_pulse;
  wire strobe_in_byte;
57
  // FIFO signals
59  wire full_1, empty_1;
  wire full_2, empty_2;
61  wire full_3, empty_3;
  wire full_4, empty_4;
63  wire full_5, empty_5;
  wire full_6, empty_6;
65
  assign strobe_tx = strobe_in_byte;   // strobe @ byte rate
67  assign strobe_out_pulse = strobe_in; // strobe_tx from dsp_tx_core @ 4 MHz

69  // input 4 MHz, output 1 MHz strobe
  cic_strober #(.WIDTH(8))
71  strober_2(
    .clock(clk),
73    .reset(rst),
    .enable(1'b1),
75    .rate(4),
    .strobe_fast(strobe_out_pulse),
77    .strobe_slow(strobe_out_qpsk_sym)
  );
79
```

```
// output strobe @ quad-bit symbol rate
cic_strober #(.WIDTH(8))
strober_3(
  .clock(clk),
  .reset(rst),
  .enable(1'b1),
  .rate(16),
  .strobe_fast(strobe_out_qpsk_sym),
  .strobe_slow(strobe_out_chips)
);

// output strobe @ byte rate
cic_strober #(.WIDTH(8))
strober_4 (
  .clock(clk),
  .reset(rst),
  .enable(1'b1),
  .rate(2),
  .strobe_fast(strobe_out_chips),
  .strobe_slow(strobe_in_byte)
);

// convert bytes to 32-chip sequence symbols
zb_symbols_to_chips_fifo u_zb_symbols_to_chips_fifo
(
  .clk(clk),
  .ce(1'b1),
  .reset(rst),
  .strobe_in(strobe_in_byte),    // strobe @ byte rate
  .strobe_out(strobe_out_chips), // strobe @ quad-bit symbol rate
  .byte(byte),                   // input byte
  .chip(chip),                   // output 32-chip symbol
  .full(full_1),
  .empty(empty_1)
);

// convert chips to 2-bit symbols
gr_packed_to_unpacked_ii_fifo u_gr_packed_to_unpacked_ii_fifo (
  .clk(clk),
  .ce(1'b1),
  .reset(rst),
  .strobe_in(strobe_out_chips),     // strobe @ quad-bit symbol rate
  .strobe_out(strobe_out_qpsk_sym), // strobe @ qpsk symbol rate
  .full(full_2),
  .empty(empty_2),
  .packed(chip),        // input 32-chip symbol
  .unpacked(chunk),       // output 2-bit symbol
  .bitpos(bitpos)
);

// map 2-bit symbols to QPSK constellation
```

```
131 gr_chunks_to_symbols_ic_fifo u_gr_chunks_to_symbols_ic_fifo (
    .clk(clk),  // fastest clk 100 MHz
133 .ce(1'b1),
    .strobe_in(strobe_out_qpsk_sym),  // strobe @ qpsk symbol rate
135 .strobe_out(strobe_out_qpsk_sym), // strobe @ qpsk symbol rate
    .full(full_3),
137 .empty(empty_3),
    .reset(rst),
139 .chunk(chunk),       // input 2-bit symbol
    .symbol(symbol)   // output qpsk constellation symbol
141 );

143 // upsample QPSK symbols by 4
    sp_upsample_cc_fifo #(.L(4))
145 u_sp_upsample_cc_fifo
    (
147 .clk(clk),
    .ce(1'b1),
149 .reset(rst),
    .strobe_in(strobe_out_qpsk_sym),  // strobe @ qpsk symbol rate
151 .strobe_out(strobe_out_pulse),    // strobe @ 4 MHz
    .full(full_4),
153 .empty(empty_4),
    .input_data(symbol),        // complex qpsk symbol
155 .output_data(output_data)          // output upsampled qpsk symbol
    );
157

    // half-sin pulse shaper
159 //zb_half_sin_pulse_no_mult_fifo u_zb_half_sin_pulse_no_mult_fifo
    zb_half_sin_pulse_fifo u_zb_half_sin_pulse_no_mult_fifo
161 (
    .clk(clk),
163 .ce(1'b1),
    .reset(rst),
165 .strobe_in(strobe_out_pulse),     // 4 MHz strobe
    .strobe_out(strobe_out_pulse),    // 4 MHz strobe
167 .full(full_5),
    .empty(empty_5),
169 .symbol(output_data), // 32 bit, 16-bit in-phase, 16-bit q-phase
    .out(pulse)           // output 32 bit I and Q baseband pulse
171 );

173 // delay quadarture to make QPSK to O-QPSK
    zb_delay_cc_fifo u_zb_delay_cc_fifo
175 (
    .clk(clk),
177 .ce(1'b1),
    .reset(rst),
179 .strobe_in(strobe_out_pulse),     // 4 MHz strobe
    .strobe_out(strobe_out_pulse),    // 4 MHz strobe
181 .full(full_6),
```

```
    .empty(empty_6),
    .input_data(pulse),
    .output_data(pulse_delayed)
);

endmodule
```

src/tx/zb_ieee_802_15_4_mod.v

B.2 Symbols to Chips

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:              10/17/2011
// Design Name:
// Filename:          zb_symbols_to_chips_fifo.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:       Map quad-bits to chips
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module zb_symbols_to_chips_fifo
(
  input clk,
  input ce,
  input strobe_in,          // strobe @ byte rate
  input strobe_out,         // strobe @ symbol rate
  input reset,
  input [7:0] byte,         // input byte
  output reg [31:0] chip, // output chip sequence
  output full,              // FIFO full
  output empty              // FIFO empty
);

// store upper and lower 4 bits
wire [3:0] lower;
wire [3:0] upper;

```

```
// signal to switch between upper and lower 4 bits
reg msb_or_lsb;

// FIFO signals
wire [31:0] dout;
wire wr_en, rd_en;
assign wr_en = strobe_in && ~full;
assign rd_en = strobe_in && ~empty;

// fifo to store input
fifo_generator_v6_2 fifo(
  .rst(reset),
  .wr_clk(clk),
  .rd_clk(clk),
  .din(byte),
  .wr_en(wr_en),
  .rd_en(rd_en),
  .dout(dout),
  .full(full),
  .empty(empty)
);

// switch between upper and lower 4 bits
always @(posedge clk) begin
  if(reset) begin
    msb_or_lsb <= 0;
  end
  else if(strobe_in)
    msb_or_lsb <= 0;
  else if(strobe_out) begin
    msb_or_lsb <= ~msb_or_lsb;
  end
end

// split into lower and upper 4 bits
assign lower = dout[3:0];
assign upper = dout[7:4];

// lower 4 bits first
// map 4 bits to chips
always @(msb_or_lsb or lower or upper) begin
  if(reset)
    chip <= 0;
  else begin
    if(~msb_or_lsb) begin
      case(lower)
        0: chip <= 3653456430;
        1: chip <= 3986437410;
        2: chip <= 786023250;
        3: chip <= 585997365;
        4: chip <= 1378802115;
```

```
89          5: chip <= 891481500;
            6: chip <= 3276943065;
91          7: chip <= 2620728045;
            8: chip <= 2358642555;
93          9: chip <= 3100205175;
            10: chip <= 2072811015;
95          11: chip <= 2008598880;
            12: chip <= 125537430;
97          13: chip <= 1618458825;
            14: chip <= 2517072780;
99          15: chip <= 3378542520;
            default: chip <= 0;
101       endcase
        end
103     else begin
          case(upper)
105         0: chip <= 3653456430;
            1: chip <= 3986437410;
107         2: chip <= 786023250;
            3: chip <= 585997365;
109         4: chip <= 1378802115;
            5: chip <= 891481500;
111         6: chip <= 3276943065;
            7: chip <= 2620728045;
113         8: chip <= 2358642555;
            9: chip <= 3100205175;
115         10: chip <= 2072811015;
            11: chip <= 2008598880;
117         12: chip <= 125537430;
            13: chip <= 1618458825;
119         14: chip <= 2517072780;
            15: chip <= 3378542520;
121         default: chip <= 0;
          endcase
123     end
    end
125 end

127 endmodule
```

src/tx/zb_symbols_to_chips_fifo.v

B.3 GNU Radio Packed to Unpacked

```
1 `timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
3 // Company:       Wireless @ VT
// Author:        Jeong-O Jeong
```

```
//
// Date:            09/19/2011
// Design Name:
// Filename:        gr_packed_to_unpacked_ii_fifo.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:     break 32-chip sequence into 16 2-bit symbols
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////

module gr_packed_to_unpacked_ii_fifo
#(parameter BITS_PER_CHUNK=2)
(
  input clk,
  input ce,
  input reset,
  input strobe_in,                              // strobe @ quad-bit symbol rate
  input strobe_out,                             // strobe @ qpsk symbol rate
  output full,                                  // FIFO full
  output empty,                                 // FIFO empty
  input [31:0] packed,                          // 32-chip sequence
  output reg [BITS_PER_CHUNK-1:0] unpacked,     // 2-chip symbol
  output reg [3:0] bitpos                       // 0-15 bi-chip position
);

// output of FIFO (chip sequence)
wire [31:0] dout;

// ASSUME MSB, 2 bits per chunk
// unpacked
always @(bitpos or dout or reset) begin
  if(reset)
    unpacked <= 0;
  else begin
    case(bitpos)
      0: unpacked <= dout[31:30];
      1: unpacked <= dout[29:28];
      2: unpacked <= dout[27:26];
      3: unpacked <= dout[25:24];
      4: unpacked <= dout[23:22];
      5: unpacked <= dout[21:20];
      6: unpacked <= dout[19:18];
      7: unpacked <= dout[17:16];
      8: unpacked <= dout[15:14];
```

```
        9: unpacked <= dout[13:12];
        10: unpacked <= dout[11:10];
        11: unpacked <= dout[9:8];
        12: unpacked <= dout[7:6];
        13: unpacked <= dout[5:4];
        14: unpacked <= dout[3:2];
        15: unpacked <= dout[1:0];
      endcase
    end
end

// FIFO signals
wire wr_en, rd_en;
assign wr_en = strobe_in && ~full;
assign rd_en = strobe_in && ~empty;

// FIFO to store input
fifo_generator_v6_2 fifo(
    .rst(reset),
    .wr_clk(clk),
    .rd_clk(clk),
    .din(packed),
    .wr_en(wr_en),
    .rd_en(rd_en),
    .dout(dout),
    .full(full),
    .empty(empty)
);

// bitpos
// control which two chips out of 32 chips to map to output
always @(posedge clk or posedge reset or posedge strobe_in) begin
    if(reset)
      bitpos <= 0;
    else if(strobe_in)
      bitpos <= 0;
    else begin
      if(strobe_out)
        bitpos <= bitpos + 1;
    end
end

endmodule
```

src/tx/gr_packed_to_unpacked_ii_fifo.v

B.4 GNU Radio Chunks to Symbols

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:              09/19/2011
// Design Name:
// Filename:          gr_chunks_to_symbols_ic_fifo.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:       Convert 2 bits into QPSK symbols
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module gr_chunks_to_symbols_ic_fifo
#(  parameter BITS_PER_CHUNK=2,
parameter WIDTH=32)
(
  input clk,
  input ce,
  input reset,
  input strobe_in,                    // strobe @ qpsk symbol rate
  input strobe_out,                   // strobe @ qpsk symbol rate
  output full,                        // FIFO full
  output empty,                       // FIFO empty
  input [BITS_PER_CHUNK-1:0] chunk,   // 32 bit integer
  output reg [WIDTH-1:0] symbol       // MSB=in-phase, LSB=q-phase
);


// FIFO signals
wire [31:0] dout;
wire wr_en, rd_en;

// ASSUME QPSK
// symbol
// -1 : 16'b1111111111111111
//  1 : 16'b0000000000000001
always @(reset or dout) begin
  if(reset)
    symbol <= 0;
  else begin
    case(dout[1:0])
```

```
      0: symbol <= 32'b1000000000000000_1000000000000000; // -1-1j
      1: symbol <= 32'b1000000000000000_0111111111111111; // -1+1j
      2: symbol <= 32'b0111111111111111_1000000000000000; // 1-1j
      3: symbol <= 32'b0111111111111111_0111111111111111; // 1+1j
    endcase
  end
end

// FIFO to store input
assign wr_en = strobe_in && ~full;
assign rd_en = strobe_in && ~empty;

fifo_generator_v6_2 fifo(
  .rst(reset),
  .wr_clk(clk),
  .rd_clk(clk),
  .din(chunk),
  .wr_en(wr_en),
  .rd_en(rd_en),
  .dout(dout),
  .full(full),
  .empty(empty)
);

endmodule
```

src/tx/gr_chunks_to_symbols_ic_fifo.v

B.5 Upsampler K=4

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////
// Company:      Wireless @ VT
// Author:       Jeong-O Jeong
//
// Date:             10/18/2011
// Design Name:
// Filename:         sp_upsample_cc_fifo
// Project Name:
// Target Devices:
// Tool versions:
// Description:      Upsample by a factor of 4
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

```
//
////////////////////////////////////////////////////////////////////////////////

module sp_upsample_cc_fifo
# ( parameter WIDTH=32,
parameter  L=4        // upsample factor
)
(
  input clk,
  input ce,
  input reset,
  input strobe_in,                  // strobe @ qpsk symbol rate
  input strobe_out,                 // strobe @ 4 MHz
  output full,
  output empty,
  input [WIDTH-1:0] input_data,       // I and Q baseband input
  output reg [WIDTH-1:0] output_data  // upsampled output
);

wire [31:0] dout;
reg [31:0] counter; // 32bit should be big enough

// counter for upsampling
always @(posedge clk or posedge reset) begin
  if(reset) begin
    counter <= 0;
  end
  else if(strobe_in)
    counter <= 0;
  else begin
    if(strobe_out) begin
      if(counter == L-1)
        counter <= 0;
      else
        counter <= counter + 1'b1;
    end
  end
end

// output input sample and insert zeros for the rest
always @(posedge clk or posedge reset) begin
  if(reset)
    output_data <= 0;
  else begin
    if(strobe_out) begin
      if(counter == 0)
        output_data <= dout;
      else
        output_data <= 0;
    end
  end
```

```
end

// FIFO for input
wire wr_en, rd_en;
assign wr_en = strobe_in && ~full;
assign rd_en = strobe_in && ~empty;

fifo_generator_v6_2 fifo(
  .rst(reset),
  .wr_clk(clk),
  .rd_clk(clk),
  .din(input_data),
  .wr_en(wr_en),
  .rd_en(rd_en),
  .dout(dout),
  .full(full),
  .empty(empty)
);

endmodule
```

src/tx/sp_upsample_cc_fifo.v

B.6 Half-Sine Pulse Shaper

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:      Wireless @ VT
// Author:       Jeong-O Jeong
//
// Date:             10/07/2011
// Design Name:
// Filename:         zb_half_sin_pulse_fifo.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:      Half-sine pulse shaper
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module zb_half_sin_pulse_fifo
#(parameter WIDTH=32)
(
```

```
  input clk,
  input ce,
  input reset,
  input strobe_in,              // strobe @ 4 MHz
  input strobe_out,             // strobe @ 4 MHz
  output full,
  output empty,
  input [WIDTH-1:0] symbol,     // 32 bit, 16-bit I, 16-bit Q
  output reg [WIDTH-1:0] out    // 32 bit, 16-bit I, 16-bit Q
);
// Samples per symbol = 4

wire [31:0] dout;

// input signals
wire signed [15:0] iphase = dout[31:16];
wire signed [15:0] qphase = dout[15:0];

// output signals
wire signed [31:0] i_out0, q_out0;
wire signed [31:0] i_out1, q_out1;
wire signed [31:0] i_out2, q_out2;
wire signed [31:0] i_out3, q_out3;

reg signed [15:0] i_z0, i_z1, i_z2, i_z3;
reg signed [15:0] q_z0, q_z1, q_z2, q_z3;

// (i, q) * 0.70710678 = 23170/32768
mult i_mult0 (.clk(clk), .ce(ce), .reset(reset), .a(iphase), .b(16'
    b0101_1010_1000_0010), .out(i_out0));
mult i_mult1 (.clk(clk), .ce(ce), .reset(reset), .a(iphase), .b(16'
    b0111_1111_1111_1111), .out(i_out1));
mult i_mult2 (.clk(clk), .ce(ce), .reset(reset), .a(iphase), .b(16'
    b0101_1010_1000_0010), .out(i_out2));
mult i_mult3 (.clk(clk), .ce(ce), .reset(reset), .a(iphase), .b(16'
    b0000_0000_0000_0000), .out(i_out3));

// (i, q) * 0.70710678 = 23170/32768
mult q_mult0 (.clk(clk), .ce(ce), .reset(reset), .a(qphase), .b(16'
    b0101_1010_1000_0010), .out(q_out0));
mult q_mult1 (.clk(clk), .ce(ce), .reset(reset), .a(qphase), .b(16'
    b0111_1111_1111_1111), .out(q_out1));
mult q_mult2 (.clk(clk), .ce(ce), .reset(reset), .a(qphase), .b(16'
    b0101_1010_1000_0010), .out(q_out2));
mult q_mult3 (.clk(clk), .ce(ce), .reset(reset), .a(qphase), .b(16'
    b0000_0000_0000_0000), .out(q_out3));

// pipeline registers
// simple FIR filter
always @(posedge clk) begin
  if(reset) begin
```

```
67      i_z0 <= 0;
        i_z1 <= 0;
69      i_z2 <= 0;
      end
71    else begin
        if(strobe_out) begin
73        i_z0 <= i_out0[31:16];
          i_z1 <= i_z0 + i_out1[31:16];
75        i_z2 <= i_z1 + i_out2[31:16];
        end
77    end
  end
79
  always @(posedge clk) begin
81    if(reset) begin
        q_z0 <= 0;
83      q_z1 <= 0;
        q_z2 <= 0;
85    end
      else begin
87      if(strobe_out) begin
          q_z0 <= q_out0[31:16];
89        q_z1 <= q_z0 + q_out1[31:16];
          q_z2 <= q_z1 + q_out2[31:16];
91      end
      end
93  end

95  // register output
  always @(posedge clk) begin
97    if(reset) begin
        out <= 0;
99    end
      else begin
101     if(strobe_out) begin
          out <= {i_z2 + i_out3[31:16],  q_z2 + q_out3[31:16]};
103     end
      end
105 end

107 // FIFO for input
  wire wr_en, rd_en;
109 assign wr_en = strobe_in && ~full;
  assign rd_en = strobe_in && ~empty;
111
  fifo_generator_v6_2 fifo(
113   .rst(reset),
      .wr_clk(clk),
115   .rd_clk(clk),
      .din(symbol),
117   .wr_en(wr_en),
```

```
    .rd_en(rd_en),
    .dout(dout),
    .full(full),
    .empty(empty)
);

endmodule
```

src/tx/zb_half_sin_pulse_fifo.v

B.7 Delay Quadrature

```
`timescale 1 ns / 1 ns
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Author:        Jeong-O Jeong
//
// Date:              10/07/2011
// Design Name:
// Filename:          zb_delay_cc_fifo.v
// Project Name:
// Target Devices:
// Tool versions:
// Description:       delay quadrature phase
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module zb_delay_cc_fifo
# ( parameter WIDTH=32  )
(
  input clk,
  input ce,
  input reset,
  input strobe_in,          // 4 MHz strobe
  input strobe_out,         // 4 MHz strobe
  output full,
  output empty,
  input [WIDTH-1:0] input_data,       // input
  output reg [WIDTH-1:0] output_data  // output with delayed quadrature
);

wire [31:0] dout;
```

```
wire [15:0] iphase;
reg [15:0] qphase[0:1];

assign iphase = dout[31:16];

// delay pipeline for quadrature component
always @(posedge clk) begin
  if(reset) begin
    qphase[0] <= 0;
    qphase[1] <= 0;
  end
  else if(strobe_out) begin
    qphase[0] <= dout[15:0];
    qphase[1] <= qphase[0];
  end
end

// register output
always @(posedge clk) begin
  if(reset)
    output_data <= 0;
  else begin
    if(strobe_out) begin
      output_data <= {iphase,qphase[1]};
    end
  end
end

// FIFO for input
wire wr_en, rd_en;
assign wr_en = strobe_in && ~full;
assign rd_en = strobe_in && ~empty;

fifo_generator_v6_2 fifo(
  .rst(reset),
  .wr_clk(clk),
  .rd_clk(clk),
  .din(input_data),
  .wr_en(wr_en),
  .rd_en(rd_en),
  .dout(dout),
  .full(full),
  .empty(empty)
);

endmodule
```

src/tx/zb_delay_cc_fifo.v

Appendix C

Verilog Source Code for Multi-channel IEEE 802.15.4 Receiver

C.1 Top Level Multi-Channel Receiver

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:04:46 05/04/2012
// Design Name:
// Module Name:    multi_channel_xbee
// Project Name:
// Target Devices:
// Tool versions:
// Description:     Top file for 4:1 channelizer, energy detector, and
    resampler
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module multi_channel_xbee
  #(parameter WIDTH=16)
  (
    input clk,                  // 100 MHz
    input ce,
    input rst,
    input [WIDTH-1:0] in_i,     // input I
```

```
28      input [WIDTH-1:0] in_q,      // input Q
        input in_valid,              // 20 MHz
30      output [WIDTH-1:0] out_i,    // output I
        output [WIDTH-1:0] out_q,    // output Q
32      output out_valid,            // 4 MHz
        output [35:0] control
34      );

36    // commutator
      wire signed [WIDTH-1:0] out_i0_commutator;
38    wire signed [WIDTH-1:0] out_i1_commutator;
      wire signed [WIDTH-1:0] out_i2_commutator;
40    wire signed [WIDTH-1:0] out_i3_commutator;
      wire signed [WIDTH-1:0] out_q0_commutator;
42    wire signed [WIDTH-1:0] out_q1_commutator;
      wire signed [WIDTH-1:0] out_q2_commutator;
44    wire signed [WIDTH-1:0] out_q3_commutator;

46    // channelizer
      wire signed [WIDTH-1:0] out_r0;
48    wire signed [WIDTH-1:0] out_r1;
      wire signed [WIDTH-1:0] out_r2;
50    wire signed [WIDTH-1:0] out_r3;
      wire signed [WIDTH-1:0] out_i0;
52    wire signed [WIDTH-1:0] out_i1;
      wire signed [WIDTH-1:0] out_i2;
54    wire signed [WIDTH-1:0] out_i3;
      reg signed [WIDTH-1:0] out_r0_valid;
56    reg signed [WIDTH-1:0] out_r1_valid;
      reg signed [WIDTH-1:0] out_r2_valid;
58    reg signed [WIDTH-1:0] out_r3_valid;
      reg signed [WIDTH-1:0] out_i0_valid;
60    reg signed [WIDTH-1:0] out_i1_valid;
      reg signed [WIDTH-1:0] out_i2_valid;
62    reg signed [WIDTH-1:0] out_i3_valid;

64    // valid signals
      wire out_valid_commutator;
66    wire out_valid_channelizer;

68    // commutator
      commutator_4_1 commutator_4_1 (
70      .clk(clk),
        .ce(ce),
72      .rst(rst),
        .in_valid(in_valid),         // 20 MHz input to commutator
74      .in_i(in_i),
        .in_q(in_q),
76      .out_i0(out_i0_commutator), // 5 MHz output of commutator
        .out_i1(out_i1_commutator),
78      .out_i2(out_i2_commutator),
```

```
    .out_i3(out_i3_commutator),
    .out_q0(out_q0_commutator),
    .out_q1(out_q1_commutator),
    .out_q2(out_q2_commutator),
    .out_q3(out_q3_commutator),
    .out_valid(out_valid_commutator)
  );

  // four-channel channelizer
  channelizer4 channelizer4 (
    .clk(clk),
    .rst(rst),
    .ce(ce),
    .in_valid(in_valid),
    .in_r0(out_i0_commutator),  // input
    .in_r1(out_i1_commutator),
    .in_r2(out_i2_commutator),
    .in_r3(out_i3_commutator),
    .in_i0(out_q0_commutator),
    .in_i1(out_q1_commutator),
    .in_i2(out_q2_commutator),
    .in_i3(out_q3_commutator),
    .out_r0(out_r0),            // output of each channel
    .out_r1(out_r1),
    .out_r2(out_r2),
    .out_r3(out_r3),
    .out_i0(out_i0),
    .out_i1(out_i1),
    .out_i2(out_i2),
    .out_i3(out_i3),
    .out_valid(out_valid_channelizer)
  );

  // register valid output from channelizer
  always @(posedge clk) begin
    if(rst) begin
      out_r0_valid <= 0;
      out_r1_valid <= 0;
      out_r2_valid <= 0;
      out_r3_valid <= 0;
      out_i0_valid <= 0;
      out_i1_valid <= 0;
      out_i2_valid <= 0;
      out_i3_valid <= 0;
    end
    else if(out_valid_channelizer) begin
      out_r0_valid <= out_r0;
      out_r1_valid <= out_r1;
      out_r2_valid <= out_r2;
      out_r3_valid <= out_r3;
      out_i0_valid <= out_i0;
```

```
130         out_i1_valid <= out_i1;
            out_i2_valid <= out_i2;
132         out_i3_valid <= out_i3;
          end
134     end

136     wire [WIDTH-1:0] out_r_energydetector;
        wire [WIDTH-1:0] out_i_energydetector;
138     wire out_valid_energydetector;

140     // energy detector
        energydetector energydetector(
142       .clk(clk),
          .rst(rst),
144       .ce(ce),
          .in_valid(out_valid_channelizer),
146       .in_r0(out_r0),                 // input
          .in_r1(out_r1),
148       .in_r2(out_r2),
          .in_r3(out_r3),
150       .in_i0(out_i0),
          .in_i1(out_i1),
152       .in_i2(out_i2),
          .in_i3(out_i3),
154       .out_r(out_r_energydetector), // output
          .out_i(out_i_energydetector),
156       .out_valid(out_valid_energydetector)
        );
158
        // resampler K=4/5
160     resampler_2_4 resampler_2_4 (
          .clk(clk),
162       .ce(ce),
          .rst(rst),
164       .in_valid(out_valid_energydetector),
          .in_i(out_r_energydetector),  // input
166       .in_q(out_i_energydetector),
          .out_valid(out_valid),        // output
168       .out_i(out_i),
          .out_q(out_q)
170     );

172 endmodule
```

src/multi_channel/multi_channel_xbee.v

C.2 1:4 Commutator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:        Wireless @ VT
// Engineer:       Jeong-O Jeong
//
// Create Date:     16:15:01 05/04/2012
// Design Name:
// Module Name:     commutator_4_1
// Project Name:
// Target Devices:
// Tool versions:
// Description:      Commutator 1:4, used before four-channel channelizer
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module commutator_4_1
#(parameter WIDTH=16)
(
  input clk,
  input ce,
  input rst,
  input in_valid,
  input signed [WIDTH-1:0] in_i,         // input
  input signed [WIDTH-1:0] in_q,
  output reg signed [WIDTH-1:0] out_i0, // output at 1/4 rate
  output reg signed [WIDTH-1:0] out_q0,
  output reg signed [WIDTH-1:0] out_i1,
  output reg signed [WIDTH-1:0] out_q1,
  output reg signed [WIDTH-1:0] out_i2,
  output reg signed [WIDTH-1:0] out_q2,
  output reg signed [WIDTH-1:0] out_i3,
  output reg signed [WIDTH-1:0] out_q3,
  output reg out_valid
);

// commutate input
always @(posedge clk) begin
  if(rst) begin
    out_i0 <= 0;
    out_q0 <= 0;
    out_i1 <= 0;
    out_q1 <= 0;
    out_i2 <= 0;
    out_q2 <= 0;
    out_i3 <= 0;
```

```
    out_q3 <= 0;
  end
  else if(ce & in_valid) begin
    out_i0 <= in_i;
    out_q0 <= in_q;
    out_i1 <= out_i0;
    out_q1 <= out_q0;
    out_i2 <= out_i1;
    out_q2 <= out_q1;
    out_i3 <= out_i2;
    out_q3 <= out_q2;
  end
end

// counter 0-3
reg [1:0] counter;
always @(posedge clk) begin
  if(rst) begin
    counter <= 0;
  end
  else if(ce & in_valid) begin
    counter <= counter + 1'b1;
  end
end

// output valid
always @(posedge clk) begin
  if(rst) begin
    out_valid <= 0;
  end
  else if(ce) begin
    out_valid <= (counter == 0)  & in_valid;
  end
end

endmodule
```

src/multi_channel/commutator_4_1.v

C.3 Four-Channel Channelizer

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless @ VT
// Engineer:      Jeong-O Jeong
//
// Create Date:     15:02:13 04/27/2012
// Design Name:
```

```
// Module Name:      channelizer4
// Project Name:
// Target Devices:
// Tool versions:
// Description:      Four-channel channelizer with complex coefficients
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module channelizer4
#(parameter WIDTH_INPUT=16,
parameter WIDTH_OUTPUT=16)
(
  input clk,
  input rst,
  input ce,
  input in_valid,                              // input
  input signed [WIDTH_INPUT-1:0] in_r0,
  input signed [WIDTH_INPUT-1:0] in_r1,
  input signed [WIDTH_INPUT-1:0] in_r2,
  input signed [WIDTH_INPUT-1:0] in_r3,
  input signed [WIDTH_INPUT-1:0] in_i0,
  input signed [WIDTH_INPUT-1:0] in_i1,
  input signed [WIDTH_INPUT-1:0] in_i2,
  input signed [WIDTH_INPUT-1:0] in_i3,
  output signed [WIDTH_OUTPUT-1:0] out_r0,  // output
  output signed [WIDTH_OUTPUT-1:0] out_r1,
  output signed [WIDTH_OUTPUT-1:0] out_r2,
  output signed [WIDTH_OUTPUT-1:0] out_r3,
  output signed [WIDTH_OUTPUT-1:0] out_i0,
  output signed [WIDTH_OUTPUT-1:0] out_i1,
  output signed [WIDTH_OUTPUT-1:0] out_i2,
  output signed [WIDTH_OUTPUT-1:0] out_i3,
  output out_valid
);


// output of polyphase filter bank
parameter WIDTH_POLY_OUTPUT = 16;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_r0;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_r1;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_r2;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_r3;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_i0;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_i1;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_i2;
wire signed [WIDTH_POLY_OUTPUT-1:0] out_poly_i3;
```

```
wire out_valid_poly;

// output of 4-point FFT
parameter WIDTH_FFT_OUTPUT = 16;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_r0;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_r1;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_r2;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_r3;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_i0;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_i1;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_i2;
wire signed [WIDTH_FFT_OUTPUT-1:0] out_fft_i3;
wire out_valid_fft;

// polyphase filter bank
polyphase4 #(.WIDTH_INPUT(WIDTH_INPUT), .WIDTH_OUTPUT(WIDTH_POLY_OUTPUT))
    polyphase4 (
  .clk(clk),
  .rst(rst),
  .ce(ce),
  .in_valid(in_valid),      // input
  .in_r0(in_r0),
  .in_r1(in_r1),
  .in_r2(in_r2),
  .in_r3(in_r3),
  .in_i0(in_i0),
  .in_i1(in_i1),
  .in_i2(in_i2),
  .in_i3(in_i3),
  .out_r0(out_poly_r0),     // output
  .out_r1(out_poly_r1),
  .out_r2(out_poly_r2),
  .out_r3(out_poly_r3),
  .out_i0(out_poly_i0),
  .out_i1(out_poly_i1),
  .out_i2(out_poly_i2),
  .out_i3(out_poly_i3),
  .out_valid(out_valid_poly)
);

// FFT-4
fft4_complex #(.WIDTH_INPUT(WIDTH_POLY_OUTPUT)) fft4_complex (
  .clk(clk),
  .ce(ce),
  .rst(rst),
  .in_valid(out_valid_poly), // input
  .in_r0(out_poly_r0),
  .in_r1(out_poly_r1),
  .in_r2(out_poly_r2),
  .in_r3(out_poly_r3),
  .in_i0(out_poly_i0),
```

```
    .in_i1(out_poly_i1),
    .in_i2(out_poly_i2),
    .in_i3(out_poly_i3),
    .out_r0(out_fft_r0),        // output
    .out_r1(out_fft_r1),
    .out_r2(out_fft_r2),
    .out_r3(out_fft_r3),
    .out_i0(out_fft_i0),
    .out_i1(out_fft_i1),
    .out_i2(out_fft_i2),
    .out_i3(out_fft_i3),
    .out_valid(out_valid_fft)
);

// complex mixer for downconversion
complex_mixer complex_mixer(
    .clk(clk),
    .ce(ce),
    .rst(rst),
    .in_valid(out_valid_fft), // input
    .in_r0(out_fft_r0),
    .in_r1(out_fft_r1),
    .in_r2(out_fft_r2),
    .in_r3(out_fft_r3),
    .in_i0(out_fft_i0),
    .in_i1(out_fft_i1),
    .in_i2(out_fft_i2),
    .in_i3(out_fft_i3),
    .out_r0(out_r0),            // output
    .out_r1(out_r1),
    .out_r2(out_r2),
    .out_r3(out_r3),
    .out_i0(out_i0),
    .out_i1(out_i1),
    .out_i2(out_i2),
    .out_i3(out_i3),
    .out_valid(out_valid)
);

endmodule
```

src/multi_channel/channelizer4.v

C.4 Polyphase Filter Bank

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:        Wireless @ VT
```

```
// Engineer:       Jeong-O Jeong
//
// Create Date:     10:21:55 04/28/2012
// Design Name:
// Module Name:     polyphase4
// Project Name:
// Target Devices:
// Tool versions:
// Description:      polyphase filterbank with complex coefficients
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module polyphase4
#(parameter WIDTH_INPUT=16,
parameter WIDTH_OUTPUT=16,
parameter WIDTH_FIR_OUTPUT=30)
(
  input clk,
  input rst,
  input ce,
  input in_valid,
  input signed [WIDTH_INPUT-1:0] in_r0,     // inputs that are already
     commutated
  input signed [WIDTH_INPUT-1:0] in_r1,
  input signed [WIDTH_INPUT-1:0] in_r2,
  input signed [WIDTH_INPUT-1:0] in_r3,
  input signed [WIDTH_INPUT-1:0] in_i0,
  input signed [WIDTH_INPUT-1:0] in_i1,
  input signed [WIDTH_INPUT-1:0] in_i2,
  input signed [WIDTH_INPUT-1:0] in_i3,
  output out_valid,
  output signed [WIDTH_OUTPUT-1:0] out_r0,      // outputs of the filters
  output signed [WIDTH_OUTPUT-1:0] out_r1,
  output signed [WIDTH_OUTPUT-1:0] out_r2,
  output signed [WIDTH_OUTPUT-1:0] out_r3,
  output signed [WIDTH_OUTPUT-1:0] out_i0,      // outputs of the filters
  output signed [WIDTH_OUTPUT-1:0] out_i1,
  output signed [WIDTH_OUTPUT-1:0] out_i2,
  output signed [WIDTH_OUTPUT-1:0] out_i3
);

// control signals for FIR filters
wire rdy_r0, rdy_i0;
wire rdy_r1, rdy_i1;
wire rdy_r2, rdy_i2;
wire rdy_r3, rdy_i3;
```

```
54
wire rfd_r0, rfd_i0;
56 wire rfd_r1, rfd_i1;
wire rfd_r2, rfd_i2;
58 wire rfd_r3, rfd_i3;

60 // output of FIR filters
wire signed [WIDTH_FIR_OUTPUT-1:0] dout_r0, dout_i0, dout_ir0, dout_ri0;
62 wire signed [WIDTH_FIR_OUTPUT-1:0] dout_r1, dout_i1, dout_ir1, dout_ri1;
wire signed [WIDTH_FIR_OUTPUT-1:0] dout_r2, dout_i2, dout_ir2, dout_ri2;
64 wire signed [WIDTH_FIR_OUTPUT-1:0] dout_r3, dout_i3, dout_ir3, dout_ri3;

66 // temporary registers for addition and subtraction
reg signed [WIDTH_FIR_OUTPUT:0] tmp_r0, tmp_i0;
68 reg signed [WIDTH_FIR_OUTPUT:0] tmp_r1, tmp_i1;
reg signed [WIDTH_FIR_OUTPUT:0] tmp_r2, tmp_i2;
70 reg signed [WIDTH_FIR_OUTPUT:0] tmp_r3, tmp_i3;
reg tmp_valid;
72
// combine the output of filters this way since the filter coeffients are
    complex
74 always @(posedge clk) begin
  if(rst) begin
76   tmp_r0 <= 0;
    tmp_r1 <= 0;
78   tmp_r2 <= 0;
    tmp_r3 <= 0;
80   tmp_i0 <= 0;
    tmp_i1 <= 0;
82   tmp_i2 <= 0;
    tmp_i3 <= 0;
84   tmp_valid <= 0;
  end
86  else if(ce) begin
    tmp_r0 <= dout_r0-dout_i0;
88   tmp_r1 <= dout_r1-dout_i1;
    tmp_r2 <= dout_r2-dout_i2;
90   tmp_r3 <= dout_r3-dout_i3;
    tmp_i0 <= dout_ri0+dout_ir0;
92   tmp_i1 <= dout_ri1+dout_ir1;
    tmp_i2 <= dout_ri2+dout_ir2;
94   tmp_i3 <= dout_ri3+dout_ir3;
    tmp_valid <= rdy_i3;
96  end
end
98

parameter OFFSET=0;
100
// output
102 assign out_r0 = tmp_r0[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];
```

```
assign out_r1 = tmp_r1[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];
assign out_r2 = tmp_r2[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];
assign out_r3 = tmp_r3[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];

assign out_i0 = tmp_i0[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];
assign out_i1 = tmp_i1[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];
assign out_i2 = tmp_i2[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];
assign out_i3 = tmp_i3[WIDTH_FIR_OUTPUT+1-1-OFFSET:WIDTH_FIR_OUTPUT+1-1-
    WIDTH_OUTPUT+1-OFFSET];

assign out_valid = tmp_valid;
//================================================================
// real coefficients, real inputs
//================================================================
fir_real fir_h0 (
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd0), // input [1 : 0] filter_sel
  .rfd(rfd_r0), // ouput rfd
  .rdy(rdy_r0), // ouput rdy
  .din(in_r0), // input [15 : 0] din_1
.dout(dout_r0)); // ouput [29 : 0] dout_1

fir_real fir_h1 (
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd1), // input [2 : 0] filter_sel
  .rfd(rfd_r1), // ouput rfd
  .rdy(rdy_r1), // ouput rdy
  .din(in_r1), // input [15 : 0] din_1
.dout(dout_r1)); // ouput [29 : 0] dout_1

fir_real fir_h2 (
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd2), // input [2 : 0] filter_sel
  .rfd(rfd_r2), // ouput rfd
  .rdy(rdy_r2), // ouput rdy
  .din(in_r2), // input [15 : 0] din_1
.dout(dout_r2)); // ouput [29 : 0] dout_1

fir_real fir_h3 (
```

```
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd3), // input [2 : 0] filter_sel
  .rfd(rfd_r3), // ouput rfd
  .rdy(rdy_r3), // ouput rdy
  .din(in_r3), // input [15 : 0] din_1
.dout(dout_r3)); // ouput [29 : 0] dout_1

//=================================================================
// imaginary coefficients, imag inputs
//=================================================================
fir_imag fir_imag_h0 (
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd0), // input [2 : 0] filter_sel
  .rfd(rfd_i0), // ouput rfd
  .rdy(rdy_i0), // ouput rdy
  .din(in_i0), // input [15 : 0] din_1
.dout(dout_i0)); // ouput [29 : 0] dout_1

fir_imag fir_imag_h1 (
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd1), // input [2 : 0] filter_sel
  .rfd(rfd_i1), // ouput rfd
  .rdy(rdy_i1), // ouput rdy
  .din(in_i1), // input [15 : 0] din_1
.dout(dout_i1)); // ouput [29 : 0] dout_1

fir_imag fir_imag_h2 (
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd2), // input [2 : 0] filter_sel
  .rfd(rfd_i2), // ouput rfd
  .rdy(rdy_i2), // ouput rdy
  .din(in_i2), // input [15 : 0] din_1
.dout(dout_i2)); // ouput [29 : 0] dout_1

fir_imag fir_imag_h3 (
  .clk(clk), // input clk
  .ce(ce), // input ce
  .nd(in_valid), // input nd
  .filter_sel(2'd3), // input [2 : 0] filter_sel
  .rfd(rfd_i3), // ouput rfd
  .rdy(rdy_i3), // ouput rdy
  .din(in_i3), // input [15 : 0] din_1
.dout(dout_i3)); // ouput [29 : 0] dout_1
```

```
198
199 //=================================================================
200 // imaginary coefficients, real inputs
201 //=================================================================
202 fir_imag fir_ir_h0 (
203   .clk(clk), // input clk
204   .ce(ce), // input ce
205   .nd(in_valid), // input nd
206   .filter_sel(2'd0), // input [2 : 0] filter_sel
207   .rfd(), // ouput rfd
208   .rdy(), // ouput rdy
209   .din(in_r0), // input [15 : 0] din_1
210 .dout(dout_ir0)); // ouput [29 : 0] dout_1
211
212 fir_imag fir_ir_h1 (
213   .clk(clk), // input clk
214   .ce(ce), // input ce
215   .nd(in_valid), // input nd
216   .filter_sel(2'd1), // input [2 : 0] filter_sel
217   .rfd(), // ouput rfd
218   .rdy(), // ouput rdy
219   .din(in_r1), // input [15 : 0] din_1
220 .dout(dout_ir1)); // ouput [29 : 0] dout_1
221
222 fir_imag fir_ir_h2 (
223   .clk(clk), // input clk
224   .ce(ce), // input ce
225   .nd(in_valid), // input nd
226   .filter_sel(2'd2), // input [2 : 0] filter_sel
227   .rfd(), // ouput rfd
228   .rdy(), // ouput rdy
229   .din(in_r2), // input [15 : 0] din_1
230 .dout(dout_ir2)); // ouput [29 : 0] dout_1
231
232 fir_imag fir_ir_h3 (
233   .clk(clk), // input clk
234   .ce(ce), // input ce
235   .nd(in_valid), // input nd
236   .filter_sel(2'd3), // input [2 : 0] filter_sel
237   .rfd(), // ouput rfd
238   .rdy(), // ouput rdy
239   .din(in_r3), // input [15 : 0] din_1
240 .dout(dout_ir3)); // ouput [29 : 0] dout_1
241
242 //=================================================================
243 // real coefficients, imag inputs
244 //=================================================================
245 fir_real fir_ri_h0 (
246   .clk(clk), // input clk
247   .ce(ce), // input ce
248   .nd(in_valid), // input nd
```

```
    .filter_sel(2'd0), // input [1 : 0] filter_sel
250   .rfd(), // ouput rfd
    .rdy(), // ouput rdy
252   .din(in_i0), // input [15 : 0] din_1
.dout(dout_ri0)); // ouput [29 : 0] dout_1
254
fir_real fir_ri_h1 (
256   .clk(clk), // input clk
    .ce(ce), // input ce
258   .nd(in_valid), // input nd
    .filter_sel(2'd1), // input [2 : 0] filter_sel
260   .rfd(), // ouput rfd
    .rdy(), // ouput rdy
262   .din(in_i1), // input [15 : 0] din_1
.dout(dout_ri1)); // ouput [29 : 0] dout_1
264
fir_real fir_ri_h2 (
266   .clk(clk), // input clk
    .ce(ce), // input ce
268   .nd(in_valid), // input nd
    .filter_sel(2'd2), // input [2 : 0] filter_sel
270   .rfd(), // ouput rfd
    .rdy(), // ouput rdy
272   .din(in_i2), // input [15 : 0] din_1
.dout(dout_ri2)); // ouput [29 : 0] dout_1
274
fir_real fir_ri_h3 (
276   .clk(clk), // input clk
    .ce(ce), // input ce
278   .nd(in_valid), // input nd
    .filter_sel(2'd3), // input [2 : 0] filter_sel
280   .rfd(), // ouput rfd
    .rdy(), // ouput rdy
282   .din(in_i3), // input [15 : 0] din_1
.dout(dout_ri3)); // ouput [29 : 0] dout_1
284
endmodule
```

src/multi_channel/polyphase4.v

C.5 Four-point FFT

```
`timescale 1ns / 1ps
2 //////////////////////////////////////////////////////////////////////////////////
// Company:       Wireless@VT
4 // Engineer:      Jeong-O Jeong
//
6 // Create Date:     15:08:58 04/27/2012
```

```
// Design Name:
// Module Name:     fft4_complex
// Project Name:
// Target Devices:
// Tool versions:
// Description:      Complex four-point FFT
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module fft4_complex
#(  parameter WIDTH_INPUT=16,
parameter WIDTH_OUTPUT=16)
(
  input clk,
  input rst,
  input ce,
  input in_valid,
  input signed [WIDTH_INPUT-1:0] in_r0,          // inputs
  input signed [WIDTH_INPUT-1:0] in_r1,
  input signed [WIDTH_INPUT-1:0] in_r2,
  input signed [WIDTH_INPUT-1:0] in_r3,
  input signed [WIDTH_INPUT-1:0] in_i0,
  input signed [WIDTH_INPUT-1:0] in_i1,
  input signed [WIDTH_INPUT-1:0] in_i2,
  input signed [WIDTH_INPUT-1:0] in_i3,
  output reg signed [WIDTH_OUTPUT-1:0] out_r0,  // outputs
  output reg signed [WIDTH_OUTPUT-1:0] out_r1,
  output reg signed [WIDTH_OUTPUT-1:0] out_r2,
  output reg signed [WIDTH_OUTPUT-1:0] out_r3,
  output reg signed [WIDTH_OUTPUT-1:0] out_i0,
  output reg signed [WIDTH_OUTPUT-1:0] out_i1,
  output reg signed [WIDTH_OUTPUT-1:0] out_i2,
  output reg signed [WIDTH_OUTPUT-1:0] out_i3,
  output reg out_valid
);


// Radix 4 FFT reduces to additions and subtactions as the following
wire [18:0] sum_r0 = in_r0 + in_r1 + in_r2 + in_r3;
wire [18:0] sum_r1 = in_r0 + in_i1 - in_r2 - in_i3;
wire [18:0] sum_r2 = in_r0 - in_r1 + in_r2 - in_r3;
wire [18:0] sum_r3 = in_r0 - in_i1 - in_r2 + in_i3;
wire [18:0] sum_r4 = in_i0 + in_i1 + in_i2 + in_i3;
wire [18:0] sum_r5 = in_i0 - in_r1 - in_i2 + in_r3;
wire [18:0] sum_r6 = in_i0 - in_i1 + in_i2 - in_i3;
wire [18:0] sum_r7 = in_i0 + in_r1 - in_i2 - in_r3;
```

```
58
// Register outputs
60 always @(posedge clk) begin
   if(rst) begin
62   out_r0 <= 0;
     out_r1 <= 0;
64   out_r2 <= 0;
     out_r3 <= 0;
66   out_i0 <= 0;
     out_i1 <= 0;
68   out_i2 <= 0;
     out_i3 <= 0;
70   out_valid <= 0;
   end
72 else if(ce) begin
     out_r0 <= sum_r0[18:3];
74   out_r1 <= sum_r1[18:3];
     out_r2 <= sum_r2[18:3];
76   out_r3 <= sum_r3[18:3];
     out_i0 <= sum_r4[18:3];
78   out_i1 <= sum_r5[18:3];
     out_i2 <= sum_r6[18:3];
80   out_i3 <= sum_r7[18:3];
     out_valid <= in_valid;
82 end
end
84
endmodule
```

src/multi_channel/fft4_complex.v

C.6 Complex Mixer

```
1 `timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
3 // Company:        Wireless @ VT
  // Engineer:       Jeong-O Jeong
5 //
  // Create Date:     11:08:59 05/02/2012
7 // Design Name:
  // Module Name:     complex_mixer
9 // Project Name:
  // Target Devices:
11 // Tool versions:
  // Description:      Downconvert input by F=0.5
13 //
  // Dependencies:
15 //
```

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module complex_mixer
#(  parameter WIDTH_INPUT=16,
parameter WIDTH_OUTPUT=16)
(
  input clk,
  input rst,
  input ce,
  input in_valid,                                    // input
  input signed [WIDTH_INPUT-1:0] in_r0,
  input signed [WIDTH_INPUT-1:0] in_r1,
  input signed [WIDTH_INPUT-1:0] in_r2,
  input signed [WIDTH_INPUT-1:0] in_r3,
  input signed [WIDTH_INPUT-1:0] in_i0,
  input signed [WIDTH_INPUT-1:0] in_i1,
  input signed [WIDTH_INPUT-1:0] in_i2,
  input signed [WIDTH_INPUT-1:0] in_i3,
  output reg signed [WIDTH_OUTPUT-1:0] out_r0,  // output
  output reg signed [WIDTH_OUTPUT-1:0] out_r1,
  output reg signed [WIDTH_OUTPUT-1:0] out_r2,
  output reg signed [WIDTH_OUTPUT-1:0] out_r3,
  output reg signed [WIDTH_OUTPUT-1:0] out_i0,
  output reg signed [WIDTH_OUTPUT-1:0] out_i1,
  output reg signed [WIDTH_OUTPUT-1:0] out_i2,
  output reg signed [WIDTH_OUTPUT-1:0] out_i3,
  output reg out_valid
);

// toggle between 0 and 1
reg counter;

always @(posedge clk) begin
  if(rst) begin
    counter <= 0;
  end
  else if(ce & in_valid) begin
    counter <= counter + 1'b1;
  end
end

// Downconversion by F=0.5 is simply flipping the sign at every other sample
always @(posedge clk) begin
  if(rst) begin
    out_r0 <= 0;
    out_r1 <= 0;
    out_r2 <= 0;
    out_r3 <= 0;
```

```
67      out_i0 <= 0;
        out_i1 <= 0;
69      out_i2 <= 0;
        out_i3 <= 0;
71      out_valid <= 0;
    end
73  else if(ce) begin
      out_valid <= in_valid;
75    if(counter) begin
        out_r0 <= in_r0;
77      out_r1 <= in_r1;
        out_r2 <= in_r2;
79      out_r3 <= in_r3;
        out_i0 <= in_i0;
81      out_i1 <= in_i1;
        out_i2 <= in_i2;
83      out_i3 <= in_i3;
      end
85    else begin
        out_r0 <= -in_r0;
87      out_r1 <= -in_r1;
        out_r2 <= -in_r2;
89      out_r3 <= -in_r3;
        out_i0 <= -in_i0;
91      out_i1 <= -in_i1;
        out_i2 <= -in_i2;
93      out_i3 <= -in_i3;
      end
95  end
end
97 endmodule
```

src/multi_channel/complex_mixer.v

C.7 Energy Detector

```
  `timescale 1ns / 1ps
2 //////////////////////////////////////////////////////////////////////////////////
  // Company:          Wireless@VT
4 // Engineer:         Jeong-O Jeong
  //
6 // Create Date:     14:40:59 05/09/2012
  // Design Name:
8 // Module Name:     energydetector
  // Project Name:
10 // Target Devices:
  // Tool versions:
12 // Description:     simple energy detector
```

```
//                  outputs the channel with maximum energy based on 4
    consecutive
//                  samples
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module energydetector
#(parameter WIDTH=16)
(
  input clk,
  input rst,
  input ce,
  input in_valid,                          // input
  input signed [WIDTH-1:0] in_r0,
  input signed [WIDTH-1:0] in_r1,
  input signed [WIDTH-1:0] in_r2,
  input signed [WIDTH-1:0] in_r3,
  input signed [WIDTH-1:0] in_i0,
  input signed [WIDTH-1:0] in_i1,
  input signed [WIDTH-1:0] in_i2,
  input signed [WIDTH-1:0] in_i3,
  output reg signed [WIDTH-1:0] out_r,  // output
  output reg signed [WIDTH-1:0] out_i,
  output reg signed out_valid
);

reg signed [2*WIDTH-1:0] prod_r0, prod_r1, prod_r2, prod_r3;
reg signed [2*WIDTH-1:0] prod_i0, prod_i1, prod_i2, prod_i3;
reg out_valid_prod;

// square each real and imag
always @(posedge clk) begin
  if(rst) begin
    prod_r0 <= 0;
    prod_r1 <= 0;
    prod_r2 <= 0;
    prod_r3 <= 0;
    prod_i0 <= 0;
    prod_i1 <= 0;
    prod_i2 <= 0;
    prod_i3 <= 0;
    out_valid_prod <= 0;
  end
  else if(ce) begin
    if(in_valid) begin
      prod_r0 <= in_r0*in_r0;
```

```
      prod_r1 <= in_r1*in_r1;
      prod_r2 <= in_r2*in_r2;
      prod_r3 <= in_r3*in_r3;
      prod_i0 <= in_i0*in_i0;
      prod_i1 <= in_i1*in_i1;
      prod_i2 <= in_i2*in_i2;
      prod_i3 <= in_i3*in_i3;
    end
    out_valid_prod <= in_valid;
  end
end


// pipeline four consecutive samples
reg signed [2*WIDTH:0] pipeline_0[0:3];
reg signed [2*WIDTH:0] pipeline_1[0:3];
reg signed [2*WIDTH:0] pipeline_2[0:3];
reg signed [2*WIDTH:0] pipeline_3[0:3];

always @(posedge clk) begin
  if(rst) begin
    pipeline_0[0] <= 0;
    pipeline_0[1] <= 0;
    pipeline_0[2] <= 0;
    pipeline_0[3] <= 0;
    pipeline_1[0] <= 0;
    pipeline_1[1] <= 0;
    pipeline_1[2] <= 0;
    pipeline_1[3] <= 0;
    pipeline_2[0] <= 0;
    pipeline_2[1] <= 0;
    pipeline_2[2] <= 0;
    pipeline_2[3] <= 0;
    pipeline_3[0] <= 0;
    pipeline_3[1] <= 0;
    pipeline_3[2] <= 0;
    pipeline_3[3] <= 0;
  end
  else if(ce) begin
    if (out_valid_prod) begin
      pipeline_0[0] <= prod_i0+prod_r0;
      pipeline_0[1] <= pipeline_0[0];
      pipeline_0[2] <= pipeline_0[1];
      pipeline_0[3] <= pipeline_0[2];
      pipeline_1[0] <= prod_i1+prod_r1;
      pipeline_1[1] <= pipeline_1[0];
      pipeline_1[2] <= pipeline_1[1];
      pipeline_1[3] <= pipeline_1[2];
      pipeline_2[0] <= prod_i2+prod_r2;
      pipeline_2[1] <= pipeline_2[0];
      pipeline_2[2] <= pipeline_2[1];
```

```
114         pipeline_2[3] <= pipeline_2[2];
            pipeline_3[0] <= prod_i3+prod_r3;
116         pipeline_3[1] <= pipeline_3[0];
            pipeline_3[2] <= pipeline_3[1];
118         pipeline_3[3] <= pipeline_3[2];
        end
120
    end
122 end


124
    // sum pipeline values
126 reg signed [2*WIDTH+3:0] sum_0, sum_1, sum_2, sum_3;

128 always @(posedge clk) begin
    if(rst) begin
130     sum_0 <= 0;
        sum_1 <= 0;
132     sum_2 <= 0;
        sum_3 <= 0;
134 end
    else if(ce) begin
136     sum_0 <= pipeline_0[0][WIDTH-4:0] + pipeline_0[1][WIDTH-4:0]  + pipeline_0
            [2][WIDTH-4:0]  + pipeline_0[3][WIDTH-4:0];
        sum_1 <= pipeline_1[0][WIDTH-4:0] + pipeline_1[1][WIDTH-4:0]  + pipeline_1
            [2][WIDTH-4:0]  + pipeline_1[3][WIDTH-4:0];
138     sum_2 <= pipeline_2[0][WIDTH-4:0] + pipeline_2[1][WIDTH-4:0]  + pipeline_2
            [2][WIDTH-4:0]  + pipeline_2[3][WIDTH-4:0];
        sum_3 <= pipeline_3[0][WIDTH-4:0] + pipeline_3[1][WIDTH-4:0]  + pipeline_3
            [2][WIDTH-4:0]  + pipeline_3[3][WIDTH-4:0];
140 end
142 end

144 // pick the maximum channel
    always @(posedge clk) begin
    if(rst) begin
146     out_r <= 0;
        out_i <= 0;
148     out_valid <= 0;
    end
150 else if(ce) begin
        if(in_valid) begin
152         if(sum_0 >= sum_1 & sum_0 >= sum_2 & sum_0 >= sum_3) begin
                out_r <= in_r0;
154             out_i <= in_i0;
            end
156         if(sum_1 >= sum_0 & sum_1 >= sum_2 & sum_1 >= sum_3) begin
                out_r <= in_r1;
158             out_i <= in_i1;
            end
160         if(sum_2 >= sum_1 & sum_2 >= sum_0 & sum_2 >= sum_3) begin
```

```
        out_r <= in_r2;
        out_i <= in_i2;
      end
      if(sum_3 >= sum_1 & sum_3 >= sum_2 & sum_3 >= sum_0) begin
        out_r <= in_r3;
        out_i <= in_i3;
      end
    end
    out_valid <= in_valid;
  end
end

endmodule
```

src/multi_channel/energydetector.v

C.8 Resampler 4/5

```
`timescale 1ns / 1ps
//
    //////////////////////////////////////////////////////////////////////////////////

// Company:        Wireless @ VT
// Engineer:       Jeong-O Jeong
//
// Create Date:     13:50:14 05/04/2012
// Design Name:
// Module Name:     resampler_2_4
// Project Name:
// Target Devices:
// Tool versions:
// Description:      Arbitrary resampler, Max interpolation rate = 2
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//
    //////////////////////////////////////////////////////////////////////////////////

module resampler_2_4(
  input clk,
  input ce,
  input rst,
  input in_valid,          // input
  input [15:0] in_i,
```

```
27    input [15:0] in_q,
      output reg out_valid,   // output
29    output reg [15:0] out_i,
      output reg [15:0] out_q
31  );

33  // output of accum_overflow
    wire overflow;
35  wire index;

37  // fifo control
    wire full;
39  wire empty;
    wire valid;
41  wire wr_en = in_valid & ~full;
    wire rd_en = overflow & ~empty;
43  wire [31:0] dout_fifo;

45  // FIFO to store incoming samples
    fifo_resampler fifo_resampler(
47    .clk(clk), // input clk
      .rst(rst), // input rst
49    .din({in_i, in_q}), // input [31 : 0] din
      .wr_en(wr_en), // input wr_en
51    .rd_en(rd_en), // input rd_en
      .dout(dout_fifo), // output [31 : 0] dout
53    .full(full), // output full
      .empty(empty), // output empty
55    .valid(valid) // output valid
    );
57
    // Clock enable
59  reg ce_internal;
    always @(posedge clk) begin
61    if(rst)
        ce_internal <= 0;
63    else
        // If FIFO is empty, stop until FIFO is not empty
65      ce_internal <= ce & ~empty;
    end
67


69  wire rfd0, rdy0;
    wire rfd1, rdy1;
71
    wire [29:0] dout0_i, dout0_q;
73  wire [29:0] dout1_i, dout1_q;

75  // Accumulator and overflow detector
    accum_overflow #(.DELTA(5<<7),  .N_BANK(2<<8), .RATE(10), .FRACTIONAL(8), .
        WIDTH_INDEX(1)) accum_overflow(
```

```
77    // K = Resample rate
      // DELTA = N_BANK/K = 2.5
79    // RATE has to be the same as FIR clock cycles
      // and FIR clock cycles have to be half of input strobe rate?
81    .clk(clk),
      .ce(ce_internal),
83    .rst(rst),
      .overflow(overflow),
85    .index(index)
);
87
// Pick output of the filter bank
89 always @(posedge clk) begin
      if(rst) begin
91      out_valid <= 0;
        out_i <= 0;
93      out_q <= 0;
      end
95    // Output of filter 0
      else if(ce_internal & rdy0 & ~valid & index==0) begin
97      out_valid <= 1'b1;
        out_i <= dout0_i[29:29-15];
99      out_q <= dout0_q[29:29-15];
      end
101   // Output of filter 1
      else if(ce_internal & rdy0 & ~valid & index==1) begin
103     out_valid <= 1'b1;
        out_i <= dout1_i[29:29-15];
105     out_q <= dout1_q[29:29-15];
      end
107   else
        out_valid <= 0;
109 end

111 // Filter banks
fir_resampler_2_4 fir_resampler0(
113   .clk(clk), // input clk
      .ce(ce_internal), // input ce
115   .nd(valid), // input nd
      .filter_sel(1'd0), // input [1 : 0] filter_sel
117   .rfd(rfd0), // ouput rfd
      .rdy(rdy0), // ouput rdy
119   .din_1(dout_fifo[31:16]), // input [15 : 0] din_1
      .din_2(dout_fifo[15:0]), // input [15 : 0] din_2
121   .dout_1(dout0_i), // ouput [29 : 0] dout_1
.dout_2(dout0_q)); // ouput [29 : 0] dout_2
123
fir_resampler_2_4 fir_resampler1(
125   .clk(clk), // input clk
      .ce(ce_internal), // input ce
127   .nd(valid), // input nd
```

```
    .filter_sel(1'd1), // input [1 : 0] filter_sel
    .rfd(rfd1), // ouput rfd
    .rdy(rdy1), // ouput rdy
    .din_1(dout_fifo[31:16]), // input [15 : 0] din_1
    .din_2(dout_fifo[15:0]), // input [15 : 0] din_2
    .dout_1(dout1_i), // ouput [29 : 0] dout_1
.dout_2(dout1_q)); // ouput [29 : 0] dout_2

endmodule
```

src/multi_channel/resampler_2_4.v

C.9 Accumulator and Overflow Detector

```
'timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:        Wireless @ VT
// Engineer:       Jeong-O Jeong
//
// Create Date:     11:26:51 05/03/2012
// Design Name:
// Module Name:     accum_overflow
// Project Name:
// Target Devices:
// Tool versions:
// Description:      accumulator and overflow detector
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module accum_overflow
#(  parameter DELTA = 5,        //delta = n_bank/K = 4/(4/5) = 5;
parameter N_BANK = 4,
parameter RATE = 20,
parameter FRACTIONAL = 0,
parameter WIDTH_INDEX = 2
)
(
  input clk,
  input ce,
  input rst,
  output overflow,                      // high when overflow
  output [WIDTH_INDEX-1:0] index        // index of filter bank
);
```

```
36 // 16 bit delta, 8 bit fractional
   localparam WIDTH_ACCUM = 16;  // 5 for 16 banks, 57 for fractional
38
   reg signed [WIDTH_ACCUM-1:0] accum;
40 wire strobe;
   reg strobe_delay;
42
   // delay strobe
44 always @(posedge clk) begin
     if(rst) begin
46     strobe_delay <= 0;
     end
48   else if(ce) begin
       strobe_delay <= strobe;
50   end
   end
52
   // strober
54 cic_strober_ce #(.WIDTH(8)) cic_strober_5mhz(
     .clock(clk),
56   .reset(rst),
     .ce(ce),
58   .enable(1'b1),
     .rate(RATE),
60   .strobe_fast(1'b1),
     .strobe_slow(strobe)
62 );

64 // accumulator
   always @(posedge clk) begin
66   if(rst) begin
       accum <= 0;
68   end
     else if(ce) begin
70     if(strobe & (accum >= N_BANK)) begin
         accum <= accum - N_BANK;
72     end
       else if(strobe & (accum < N_BANK)) begin
74       accum <= accum + DELTA;
       end
76   end
   end
78
   // overflow detector
80 assign overflow = ce & strobe_delay & (accum >= N_BANK);

82 // choose filter
   assign index = accum[WIDTH_INDEX+FRACTIONAL-1:FRACTIONAL];
84
   endmodule
```

src/multi_channel/accum_overflow.v

Appendix D

C++ Source Code for GNU Radio Blocks

D.1 GNU Radio Transmitter .h and .cc

```
/* -*- c++ -*- */
/*
 * Copyright 2004 Free Software Foundation, Inc.
 *
 * This file is part of GNU Radio
 *
 * GNU Radio 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, or (at your option)
 * any later version.
 *
 * GNU Radio 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 GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

// WARNING: this file is machine generated.  Edits will be over written

#ifndef INCLUDED_ZIGBEE_SIG_SOURCE_C_H
#define INCLUDED_ZIGBEE_SIG_SOURCE_C_H

#include <gr_sync_block.h>

class zigbee_sig_source_c;
```

```
31 typedef boost::shared_ptr<zigbee_sig_source_c> zigbee_sig_source_c_sptr;

33 /*!
 * \brief signal generator with zigbee_complex output.
35 * \ingroup source_blk
 */
37
class zigbee_sig_source_c : public gr_sync_block {
39   friend zigbee_sig_source_c_sptr
   zigbee_make_sig_source_c (double ampl, gr_complex offset);
41
     double          d_ampl;
43     gr_complex    d_offset;
     int             d_cntr;         // counter to keep track of state
45     int             d_N;
     int             d_sent;
47     double          d_factor;

49   zigbee_sig_source_c (double ampl, gr_complex offset);

51  public:
   virtual int work (int noutput_items,
53         gr_vector_const_void_star &input_items,
         gr_vector_void_star &output_items);
55
   // ACCESSORS
57   double amplitude () const { return d_ampl; }
   gr_complex offset () const { return d_offset; }
59
   // MANIPULATORS
61   void set_amplitude (double ampl);
   void set_offset (gr_complex offset);
63   char generate_crc(char msg[], int index);
};
65
zigbee_sig_source_c_sptr
67 zigbee_make_sig_source_c (double ampl, gr_complex offset = 0);

69 #endif
```

src/gr-zigbee/zigbee_sig_source_c.h

```
1 /* -*- c++ -*- */
/*
3  * Copyright 2004,2010 Free Software Foundation, Inc.
 *
5  * This file is part of GNU Radio
 *
7  * GNU Radio is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
9  * the Free Software Foundation; either version 3, or (at your option)
```

```
 * any later version.
 *
 * GNU Radio 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 GNU Radio; see the file COPYING.  If not, write to
 * the Free Software Foundation, Inc., 51 Franklin Street,
 * Boston, MA 02110-1301, USA.
 */

// WARNING: this file is machine generated.  Edits will be over written

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
#include <zigbee_sig_source_c.h>
#include <algorithm>
#include <gr_io_signature.h>
#include <stdexcept>
#include <gr_complex.h>
#include <iostream>
//char msg[]={0x14, 0x61, 0x88, 0x5E, 0x32, 0x33, 0x00, 0x00, 0x00, 0x00
    , 0x00, 'H', 'E', 'L', 'L', 'O', '!', '!', '!'};
//char msg[]={0x61, 0x88, 0xc0, 0x32, 0x33, 0x0, 0x0, 0x0, 0x0, 0xa2, 0x0, 0
    x48, 0x45, 0x4c, 0x4c, 0x4f};
// message to be transmitted
char msg[]={0x41, 0xcc, 0xe5, 0x78, 0x56, 0x22, 0x5a, 0x63, 0x40, 0x00, 0xa2,
    0x13, 0x00, 0x70, 0x5a, 0x63, 0x40, 0x00, 0xa2, 0x13, 0x00, 'M','E','S','S
    ','A','G','E',' ','C','O','M','I','N','G',' ','F','R','O','M',' ','U','S',
    'R','P',' ','F','P','G','A',0x0d};
char chk[]={0x1f, 0xce};//{0xE7, 0x3F};

zigbee_sig_source_c::zigbee_sig_source_c (double ampl, gr_complex offset)
  : gr_sync_block ("sig_source_c",
      gr_make_io_signature (0, 0, 0),
      gr_make_io_signature (1, 1, sizeof (gr_complex))),
  d_ampl (ampl), d_offset (offset)
{
  d_ampl=0;
  d_cntr=0;
  d_N=53;             // length of packet
  d_factor=32768.0;
  d_sent=0;
  chk[0]=generate_crc(msg, 0);
  chk[1]=generate_crc(msg, 1);
  std::cout << "================================" << std::endl;
  std::cout << (unsigned short)(chk[0]) << std::endl;
  std::cout << (unsigned short)(chk[1]) << std::endl;
```

```
  std::cout << "==================================" << std::endl;
}

  zigbee_sig_source_c_sptr
zigbee_make_sig_source_c (double ampl, gr_complex offset)
{
  return gnuradio::get_initial_sptr(new zigbee_sig_source_c (ampl, offset));
}

// Generate checksum
char zigbee_sig_source_c::generate_crc(char msg[], int index)
{
  char checksum[2]={0x00,0x00};
  char FCS[16];
  // initialize FCS vector
  for(int i=0; i<16; i++){
    FCS[i]=0;
  }
  int j=0;
  char input;
  char s0, s1, s2;
  for(int i=0; i<(d_N-2)*8; i++)
  {
    input = 0x01 & ((char)msg[i/8] >> j);
    s0 = input ^ FCS[0];
    s1 = s0 ^ FCS[4];
    s2 = s0 ^ FCS[11];
    FCS[0] = FCS[1];
    FCS[1] = FCS[2];
    FCS[2] = FCS[3];
    FCS[3] = s1;
    FCS[4] = FCS[5];
    FCS[5] = FCS[6];
    FCS[6] = FCS[7];
    FCS[7] = FCS[8];
    FCS[8] = FCS[9];
    FCS[9] = FCS[10];
    FCS[10] = s2;
    FCS[11] = FCS[12];
    FCS[12] = FCS[13];
    FCS[13] = FCS[14];
    FCS[14] = FCS[15];
    FCS[15] = s0;
    j=(j+1) % 8;
  }

  for(int i=0; i<8; i++)
  {
    checksum[0]=checksum[0] | ((FCS[i] << i));
  }
  for(int i=8; i<16; i++)
```

```
107   {
        checksum[1]=checksum[1] | ((FCS[i] << (i-8)));
109   }
      /*std::cout << "================================" << std::endl;
111     std::cout << (unsigned short)(0xFF & checksum[0]) << std::endl;
        std::cout << (unsigned short)(0xFF & checksum[1]) << std::endl;
113     std::cout << "================================" << std::endl;*/
    return checksum[index];
115 }

117   int
    zigbee_sig_source_c::work (int noutput_items,
119     gr_vector_const_void_star &input_items,
        gr_vector_void_star &output_items)
121 {
    gr_complex *optr = (gr_complex *) output_items[0];
123 gr_complex t;

125 t = (gr_complex) d_ampl + d_offset;
    //================================
127 // d_ampl can be between 0 and 255
    //================================
129 std::cout << "noutput_items: " << noutput_items << std::flush << std::endl;
    for (int i = 0; i < noutput_items; i++)
131 {
      // preamble     (4 bytes)
133   // SFD          (1 bytes)
      // framelength  (1 bytes)
135   // payload      (N bytes)
      // FCS          (2 bytes)
137
      if(d_cntr < 4)             // preamble
139     d_ampl = 0;
      else if(d_cntr < 5)        // SFD
141     d_ampl = 167;
      else if(d_cntr < 6)        // framelength
143     d_ampl = d_N;
      else if(d_cntr < 6+d_N-2) // payload
145     d_ampl=msg[d_cntr-6];
      else if(d_cntr < 6+d_N-1) // FCS
147     d_ampl=chk[0];
      else if(d_cntr < 6+d_N)    // FCS
149   {
        d_ampl=chk[1];
151   }

153   // convert char into complex short
      optr[i] = gr_complex(d_ampl/d_factor, 1.0/d_factor);
155
      // reset if finished encoding a packet
157   if(d_cntr >= 6+d_N+2)
```

```
      d_cntr = 0;
159   else
      d_cntr++;
161
  }
163 std::cout << "Generating packets.." << std::flush << std::endl;
  return noutput_items;
165 }

167   void
zigbee_sig_source_c::set_amplitude (double ampl)
169 {
  //  d_ampl = ampl;
171 }

173   void
zigbee_sig_source_c::set_offset (gr_complex offset)
175 {
  d_offset = offset;
177 }
```

src/gr-zigbee/zigbee_sig_source_c.cc

D.2 GNU Radio Receiver .h and .cc

```
#ifndef INCLUDED_ZIGBEE_RX_MOD_SS_H
2 #define INCLUDED_ZIGBEE_RX_MOD_SS_H

4 #include <gr_block.h>
#include <fstream>
6
class zigbee_rx_mod_ss;
8
typedef boost::shared_ptr<zigbee_rx_mod_ss> zigbee_rx_mod_ss_sptr;
10
// public interface for creating new instances
12 // howto_square_ff's constructor is private
zigbee_rx_mod_ss_sptr zigbee_make_rx_mod_ss(size_t itemsize);
14
class zigbee_rx_mod_ss : public gr_block
16 {
  private:
18   // allow howto_make_square_ff to access the private constructor
     // "friend" - once a non-member function is declared as a friend, it can
        acess private data of the class
20   friend zigbee_rx_mod_ss_sptr zigbee_make_rx_mod_ss(size_t itemsize);

22   zigbee_rx_mod_ss(size_t itemsize);  // private constructor
```

```
24        //enum state_t {STATE_FRAMELENGTH, STATE_PAYLOAD, STATE_CRC}
          //int d_state;                        // FRAMELENGTH, PAYLOAD, CRC
26        int d_counter_detected;
          int d_counter64;
28        int d_byte_counter;
          int d_total_crc_correct;
30        int d_total_bits;
          int d_error_bits;
32      std::ofstream d_ofile;
          std::ofstream d_preamble_file;
34
  public:
36    ~zigbee_rx_mod_ss();  // public destructor

38    int general_work (int noutput_items,
              gr_vector_int &ninput_items,
40            gr_vector_const_void_star &input_items,
              gr_vector_void_star &output_items);
42
};
44
#endif /* INCLUDED_HOWTO_SQUARE_FF_H */
```

src/gr-zigbee/zigbee_rx_mod_ss.h

```
#ifdef HAVE_CONFIG_H
2 #include "config.h"
#endif
4
#include <zigbee_rx_mod_ss.h>
6 #include <gr_io_signature.h>
#include <iostream>
8 #include <iomanip>
#include <cstring>
10 #include <stdio.h>
#include <fstream>
12
std::ofstream output_file;
14
zigbee_rx_mod_ss_sptr zigbee_make_rx_mod_ss(size_t itemsize)
16 {
  return zigbee_rx_mod_ss_sptr (new zigbee_rx_mod_ss (itemsize));
18 }

20 static const int MIN_IN = 1;
static const int MAX_IN = 1;
22 static const int MIN_OUT = 0;
static const int MAX_OUT = 0;
24
// Private constructor
```

```
26 zigbee_rx_mod_ss::zigbee_rx_mod_ss(size_t itemsize) : gr_block("
       zigbee_rx_mod_ss",
   gr_make_io_signature (MIN_IN, MAX_IN, itemsize),
28   gr_make_io_signature (MIN_OUT, MAX_OUT, sizeof(short)))
{
30  d_counter64=0;
  d_byte_counter=0;
32  d_total_crc_correct=0;
  d_total_bits=0;
34  d_error_bits=0;
  output_file.open("output_zigbee.txt");
36 }

38 // Virtual destructor
zigbee_rx_mod_ss::~zigbee_rx_mod_ss()
40 {
  output_file.close();
42  //  d_ofile.close();
  //    d_preamble_file.close();
44 }

46 //char message[]="0123456789:;<=>?@ABCDEFGHIJKLMNO";
int LEN=40; // length of transmitted payload
48 // Transmitted message to be compared against the received message for BER
       calculation
//char message[]="0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789:;<=>?
       @ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
50 char message[]="01234567890123456789012345678901234567890123456789";

52 // count the number of bits that are one
int bitcount(char n)
54 {
  int tot=0;
56  int i;
  for (i=1; i<=128; i=i*2){
58    if(n & (0xFF & i))
      ++tot;
60  }
  return tot;
62 }

64 // compare bits
int compare_bits(char received, char ref)
66 {
  return bitcount(0xFF & ((ref^received)));
68 }


70
int zigbee_rx_mod_ss::general_work(int noutput_items,
72   gr_vector_int &ninput_items,
   gr_vector_const_void_star &input_items,
```

```
74      gr_vector_void_star &output_items)
{
76    int MASK = 0x000000FF;
    const char *in = (const char *) input_items[0];
78    short *out = (short *) output_items[0];

80    char *word = (char *)malloc(4);
    int state;
82    int prev_state=0;
    int strobe_byte;
84    char byte;
    int count_byte=0;
86    int crc_correct=0;

88    for(int i=0; i<noutput_items; i++){
      std::memcpy(word, in, 4);
90
      // parse 32-bit input
92      state = (0x0F & word[0]);
      crc_correct = ((0x10 & word[1]) >> 4);
94      strobe_byte = ((0x20 & word[1]) >> 5);
      byte = (0xFF & word[2]);
96
      // reset counter if at DECODE_FRAMELENTH or before
98      if(state < 5)
      {
100       d_byte_counter = 0;
      }
102     // if at DECODE_SYMBOLS state
      if(state == 5 && strobe_byte == 1)
104     {
        if(d_byte_counter == 0)
106       std::cout << "PACKET DETECTED " << ++d_counter64 << std::flush << std
              ::endl;

108       // if passed header information start printng payload
        if(d_byte_counter > 9)
110       {
          std::cout << byte << std::flush;
112       // start couting bits in error
          if(d_byte_counter-10 < LEN){
114         d_error_bits = d_error_bits + compare_bits(byte, message[
                d_byte_counter-10]);
            d_total_bits += 8;
116       }
        }
118     // count the number of bits received
        d_byte_counter++;
120   }

122   // if in CHECK_CRC state
```

```
    if(state == 8 && strobe_byte ==1){
      std::cout << "\nCRC:" << crc_correct << std::endl;
      if(crc_correct == 1){
        d_total_crc_correct++;  // count the number of packets that have
            correct CRC
        // calculate and print BER
        std::cout << "Total CRC: " << d_total_crc_correct << std::flush << std
            ::endl;
        std::cout << "BER : " << (1.0*(double)d_error_bits) / (1.0*(double)
            d_total_bits) << std::flush << std::endl;
        std::cout << "errors : " << (d_error_bits) << std::flush << std::endl;
        std::cout << "total : " << d_total_bits << std::flush << std::endl;
      }
      else{
        std::cout << "Total CRC: " << d_total_crc_correct << std::flush << std
            ::endl;
        std::cout << "BER : " << (1.0*(double)d_error_bits) / (1.0*(double)
            d_total_bits) << std::flush << std::endl;
        std::cout << "errors : " << (d_error_bits) << std::flush << std::endl;
        std::cout << "total : " << d_total_bits << std::flush << std::endl;
      }
    }
    // store previous state
    prev_state = state;
    // increment pointer
    in+=4;
  }
  std::free(word);
  // tell the scheduler how many items were consumed
  consume_each(noutput_items);
  // number of items consumed
  return noutput_items;
}
```

src/gr-zigbee/zigbee_rx_mod_ss.cc