Evaluation of GNU Radio Platform Enhanced for Hardware Accelerated Radio Design

Mrudula P. Karve

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

Peter Athanas, Chair
Claudio DaSilva
Michael Buehrer

December 13, 2010
Blacksburg, Virginia

Keywords: Software defined radio, FPGA, GNUradio, Zigbee receiver, MB-OFDM UWB

Evaluation of GNU Radio Platform Enhanced for Hardware Accelerated Radio Design

Mrudula P. Karve

ABSTRACT

The advent of software radio technology has enabled radio developers to design and implement radios with great ease and flexibility. Software radios are effective in experimentation and development of radio designs. However, they have limitations when it comes to high-speed, high-throughput designs. This limitation can be overcome by introducing a hardware element to the software radio platform. *Enhancing GNU Radio for Hardware Accelerated Radio Design* project implements such a scheme by augmenting an FPGA co-processor to a conventional GNU Radio flow. In this thesis, this novel platform is evaluated in terms of performance of a radio design, as well as hardware and software system requirements. A simple and efficient Zigbee receiver design is presented. Implementation of this receiver is used as a proof-of-concept for the effectiveness and design methodology of the modified GNU Radio. This work also proposes a scheme to extend this idea for design of ultra-wideband radio systems based on multiband-OFDM.

This work received financial support from the Strategic Technology Office (STO) at DARPA.

To my parents...

Acknowledgments

This thesis would not have been possible without the guidance and support of many people. First and foremost, I owe my deepest gratitude to my advisor, Dr. Peter Athanas for supporting me throughout my thesis with his patience and knowledge while allowing me to work in my own way. I am grateful to him for his constant encouragement and giving me an opportunity to work in the CCM lab.

I would like to thank Dr. DaSilva and Dr. Buehrer for graciously agreeing to be a part of my thesis committee and for enriching my learning experience at Virginia Tech. I really enjoyed your classes.

I express my deepest gratitude to Adolfo Recio for taking time to explain to me numerous concepts in communications and DSP and above all, for being a source of inspiration. I am also thankful to all my labmates in the CCM lab for their support on my research and thesis.

I would also like to thank Gautham Chavali for all the discussions, which helped me successfully complete my receiver design.

Thank you Saparya Krishnamoorthy, Vaishnavi Srinivasaraghavan, Kalyani Tipnis, Aditi Chaudhry, Shruti Iyer and Sushrutha Vigraham for being great friends and making my stay at Blacksburg so memorable.

Finally, I would like to thank my father, Prabhakar Karve for encouraging me to take up higher studies and supporting me throughout my way. A heartfelt thanks to my mother, Kanchan Karve and sister, Manasi for their unconditional love and support.

Mrudula Karve
Blacksburg
December 5, 2010

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Great demands for sophisticated wireless connectivity have led to a furiously paced development in the wireless communications technology. This in turn has resulted in rapid emergence of new standards, protocols and techniques in the field of radios. On availability of these new standards, the conventional fixed, hardware-intensive radios are rendered obsolete. Radio manufacturers have reported huge amount of losses due to defective devices [1]. The software-defined radio (SDR) technology has enabled radio-developers to overcome these shortcomings of traditional radios. The reconfigurability or reprogrammability of software radios allows to create future-proof radios, which could be easily upgraded as new standards arise or made defect-free as new bugs are discovered. The biggest disadvantages of software radios are high power consumption, cost and inability to achieve higher data-rates. Nonetheless, numerous endeavors are being undertaken in the development of SDRs by overcoming these drawbacks.

The increasing importance of software radios has triggered several efforts to develop software radio platforms and supporting technologies. Some well known examples of SDR platforms are: WARP (Wireless Open Access Research Platform) developed at the Rice University [4],

OSSIE (Open Source SCA Implementation: Embedded)- a project by the Wireless research group at Virginia Tech [5] and the GNU Radio [6].

GNU Radio is an open source SDR platform that provides signal processing software blocks to rapidly implement software radios on low-cost commodity processors with the help of an external RF-front end. Being open source, the GNU Radio has several advantages in terms of cost, stability and security. A large developer base contributes to fast betterment of the GNU Radio, by a continuous process of testing and debugging the source code and adding new features to it. GNU Radio applications are mainly written using the Python language; while the underlying signal processing components are developed in C++. Users can easily develop radio systems in a simple-to-use, rapid-application-development environment. Thus, a major benefit of system development using GNU Radio is increased user productivity. However, GNU Radio systems primarily use General Purpose Processors (GPP) as their hardware engine, which are not preferred when it comes to real-time signal processing applications. GPPs being slower, other devices such as ASICs, DSP microprocessors and FPGAs are considered more suitable for such applications.

ASICs offer low power consumption but almost zero programmability, where as, DSPs provide good reconfigurability at the expense of being power hungry. FPGAs provide a good trade-off in terms of programmability and power consumption within the different classes of hardware (Figure 1.1). Hence, FPGAs are increasingly becoming a favored candidate for computation-intensive signal processing applications in radios. They have an inherent advantage of high computational power and flexibilty; but high power consumption and large reconfiguration times limit their usefulness in a software radio scenario. Reconfiguration time in a traditional FPGA flow is high due to large build-times in re-synthesizing and loading a new design on the FPGA [1, 2, 3, 10, 25]. Several research efforts have been initiated in order to overcome these drawbacks. FPGA vendors are introducing new devices having lower power requirements and higher computational densities. Significant reduction in reconfiguration-times has been acheived by novel techniques such as partial reconfiguration

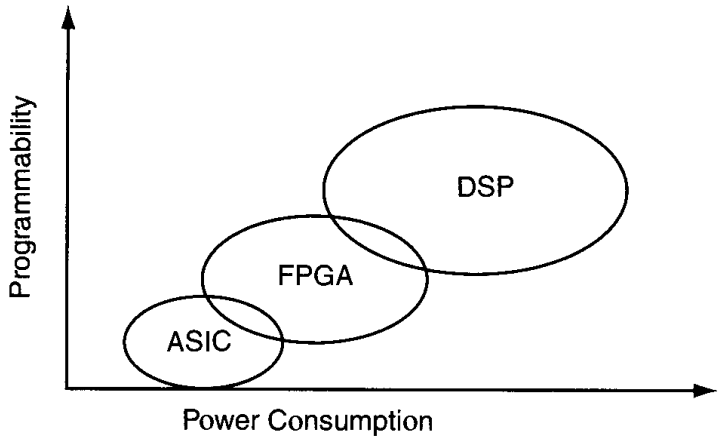


Figure 1.1: Trade-offs in programmability and power consumption for different classes of hardware (figure from [1])

[7, 8, 9]. The AgileHW project developed in the Configurable Computing Machines (CCM) Lab at Virginia Tech is an example of using the partial reconfiguration for increased agility and productivity.

The *Enhancing GNU Radio for Hardware Accelerated Radio Design* project uses a combination of above mentioned technologies such as Agile Hardware and GNU Radio, to create an open, agile, easy-to-use and efficient radio development environment [10]. It provides a robust software radio framework by augmenting the existing GNU Radio platform with an FPGA co-processor. The auxiliary FPGA co-processor resolves the computational power issues of conventional GNU Radio without inhibiting its normal flow. As a result, this enhanced GNU Radio allows more complex and high-speed radio designs without sacrificing agility and productivity.

1.1 Motivation

This modified GNU Radio platform clearly seems to be a promising technique in the field of software defined radios. It enables radio developers to exploit the benefits of both hardware and software domains to their full potential. However, this is a recent development and needs to be investigated further. The platform hasn't been previously tested for implementation of a fully functional radio link. Realizing such a radio design on the platform would provide an insight into design methodology, performance and integrity of the new platform, which provides a good motivation for the work of this thesis.

1.2 Research Contributions

This thesis explores the design methodology and effectiveness of an enhanced GNU Radio platform, by means of implementing a Zigbee receiver design on it. Following are the primary research contributions of this thesis:

1. Implementation and verification of a Zigbee receiver on the modified GNU Radio platform: A simple and efficient modular design of a Zigbee demodulator is presented. Performance of the wireless link is assessed using various experiments and simulations. Design of a more relevant high data rate ultra-wide band (UWB) radio is also discussed.

2. This elementary radio implementation aids in understanding the benefits and challenges of employing the platform. The system is also evaluated and compared with a standard GNU Radio in terms of software and hardware resource utilization.

3. Further research with the platform helps in promoting this new technology. A proof-of-concept radio design raises interest and supports for wider acceptance of this robust, efficient and open-source software radio platform.

1.3 Organization of Thesis

The rest of this thesis is organized as follows. Chapter 2 provides the necessary background information on all the relevant technologies involved in this work. It gives a brief overview of the GNU Radio platform, Agile Hardware project and the Zigbee standard. Chapter 3 outlines all the necessary details of the modified GNU Radio platform enhanced for hardware acceleration. Chapter 4 explains the design and implementation of the Zigbee receiver. Chapter 5 describes a possible solution for realizing a UWB radio system based on multiband OFDM (MB-OFDM) using a software radio platform such as the one involved in this work. Chapter 6 presents implementation results of the receiver and other statistics based on experiments and simulations. Chapter 7 concludes the thesis, along with ideas for future work.

Chapter 2

Background

The primary technologies involved in development of the novel platform used for this work are the GNU Radio and the Agile Hardware project. This chapter highlights the important details about these projects. It also presents an overview on WPAN standards, particularly the Zigbee standard which is used as the radio application for this thesis.

2.1 GNU Radio

The GNU Radio project is a free software toolkit for building and deploying software radio systems. This open-source signal processing package is distributed under the terms of GNU general public license [6]. Eric Blossom and John Gilmore started the GNU Radio in 2001, which is now widely used by universities and the industry for wireless communications research as well as to implement real-time radios. GNU Radio consists of a library of signal processing blocks (implemented in C++) and the glue to tie it all together. Users can easily build a software radio by creating a graph where the vertices are signal processing blocks and the edges represent the data flow between them. These graphs are constructed and run

in Python [6, 10, 11]. The preferred radio front-end for GNU Radio is called Universal Software Radio Peripheral (USRP). USRP is an interface between a host machine and the RF world [12, 13]. The host machine is basically any commercially available computer that runs the software radio script on it. The following subsections elaborate on the USRP, software features of the GNU Radio and implementation of the radio flow graphs on the GNU Radio.

2.1.1 The Universal Software Radio Peripheral

The USRP is a preferred hardware RF front-end solution for the GNU Radio. Being a part of an open-source project, the USRP is an inexpensive device and its board schematics and drivers are freely available. It primarily consists of a motherboard that includes an ADC, a DAC, an FPGA and a controller for the host-machine interface. The FPGA implements up-converters and down-converters. The USRP motherboard also supports four daughterboards (two each for transmit and receive) that implement the RF front ends; hence, the important components of these daughterboards include a mixer and local oscillator ICs. An important feature of the USRP is its flexiblity. It can be configured from the host machine to set required values of various parameters such as ADC/DAC gain factors and decimation/interpolation rates. Also, different daughterboards can be used to support different radio frequency bands. The USRP supports a USB 2.0 interface to connect with the host machine. Another version of USRP, called the USRP2 uses a faster Gigabit Ethernet (GigE) for host machine interface. Figure 2.1 illustrates the data flow between host machine and USRP2 radio front-end at the transmitting and receiving nodes. The USRP2 and host machine use the same GigE interface for the communication of control as well as radio data. The host machine sends configuration commands to the USRP2 to set the required parameters (RF frequency, gain and decimation/interpolation rates). At the transmitter-end, the host machine sends baseband data to the USRP2, which modulates it over the selected carrier frequency and transmits it over the air. In case of the receiver, the USRP2 converts radio frequency data to baseband data, which is used by the software radio on the host machine

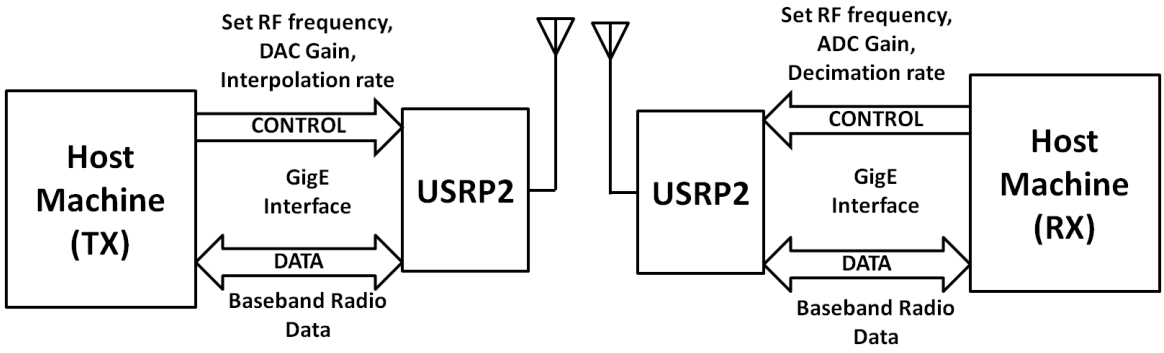


Figure 2.1: GNU Radio Hardware flow.

to demodulate and get back the sent information.

2.1.2 GNU Radio Model

Software functionalities of a GNU Radio system can be run on any standard machine running a Linux, Windows or OSX operating system. The GNU Radio programming has a fairly straightforward model. A Python script is used to create a modular radio design in terms of a flow graph, which represents how the radio data moves through the signal processing functions. A large number of commonly used functional blocks implemented in C++ are provided by the GNU Radio; however, in case new functions are needed then it is the radio developer's responsibility to write those blocks. These blocks can be conceptually considered to process a continuous stream of data flowing from their input to output ports. The attributes of a block include number of its input and output ports as well as the data type handled by each. Most common data types are short, float and complex. Sources and sinks in the graph have either an input or an output port. The sources have only input ports that read from a file or ADC, while the sinks have only ouput ports, which write to a file, digital-to-analog converter (DAC) or a graphical display.

Writing a Python script that implements a required system is reasonably simple and essentially involves connecting all the functional modules in a proper sequence. The GNU Radio also provides a GUI tool called *GNU Radio Companion* to implement these flow graphs [14]. It allows users to create a flow graph using a simple drag-and-drop interface and generates a corresponding Python script. Once the graph is built, simply running the script turns the radio system on. The most important features of GNU Radio are its ease-of-deployment and that it provides almost instant gratification for scripting a radio design. A complicated radio design can be generated without great difficulty using a straight-forward object-oriented Python script [15, 16].

Figure 2.2 shows a simple "Hello world" script for the GNU Radio. It takes in two sine waveforms and connects them to an audio output. In lines 13 and 14 of the script, the sine waveforms with required parameters are generated. The output audio sink is instantiated in line 16. Lines 17 and 18 are use to connect all the nodes of the graph. This shows how the object-oriented nature of Python scripting makes the software radio implementation straight-forward. The nodes or their parameters can be changed when required. This is possible due to the concept of modularity used in the GNU Radio design.

2.2 Agile Hardware

The Agile Hardware project has been an important research effort in the Configurable Computing Machines Lab at Virginia Tech. The central idea of this project is partial reconfiguration, which enables configuration of select portions of an FPGA with help of modifications to the standard FPGA tool-flow. In this environment, there exists a large reconfigurable region on the FPGA on which precompiled logic cores can be placed or removed during runtime [8, 9, 21, 22, 23]. A precompiled logic core is basically a unit that carries out certain computational functionality. The Agile Hardware flow is illustarted in Figure 2.3, which

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

Figure 2.2: A "Hello world" GNU Radio script [15].

depicts a Agile Hardware system in radio scenario. The large reconfigurable region is also referred to as a sandbox and holds the precompiled cores of the functional modules required to create a radio. The rest of the resources on the FPGA device are used for the logic for interaction between the dynamic sandbox region and external devices. Since this part of the FPGA is not enabled for dyanmic reconfiguration, it is known as the static region. In Figure 2.3, the static logic consists of the logic required to connect an external Ethernet interface to the radio realized in the dynamic region. The CDC FIFO (Clock Domain Crossing- First In First Out) blocks represent FIFO blocks used for data flow control between the static and dynamic regions. The clock domain crossing gives the freedom to have different clock rates in the radio design and the external Ethernet-attached device. The encode/decode and Eth Rx/Tx blocks are necessary to implement the Ethernet frame encoding/decoding and other functionalities. The usage of the Agile Hardware can be broken up into two steps:

1. Compile time flow: This is the first step in Agile Hardware flow and is responsible for establishing communication between static and dynamic region and creating partial

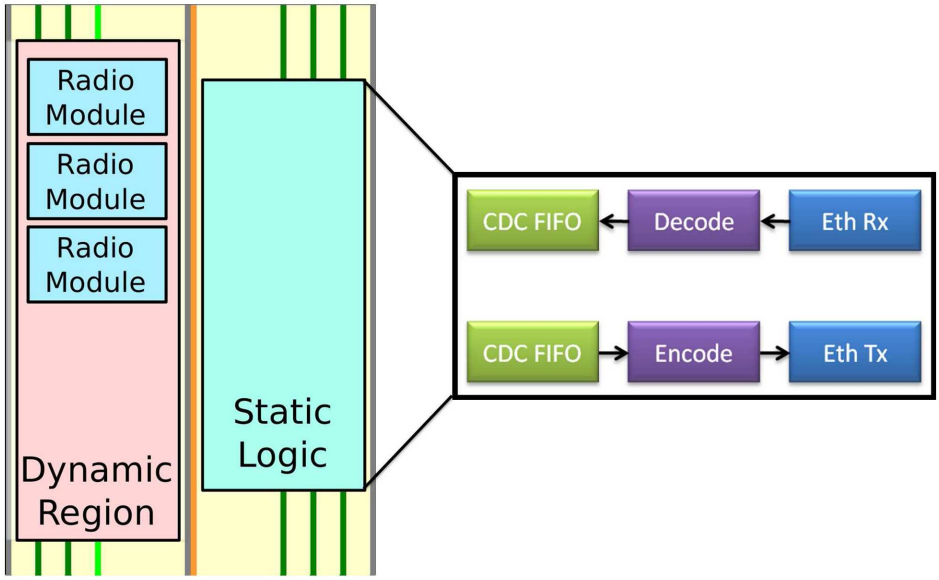


Figure 2.3: Agile Hardware model.

modules. As mentioned before, partial modules are the building blocks of the radio. The static region consists of logic for: interface between the dynamic region and I/O bus, an empty dynamic region wrapper where the partial modules will be placed, clock management functions, data I/O modules and blocks necessary for the partial reconfiguration functionality.

2. Run time flow: This step is responsible for on-the-fly placing and routing of a partial module in the dynamic region and linking it with the static bitstream, so that the static and dynamic logic work together smoothly.

Agile Hardware is an important concept for this work due to following reasons:

- Increased FPGA agility and productivity: In enhanced GNU Radio system, this helps in shifting the computational work load of the radio on FPGA unit without affecting the flexibility.

- Modularity: In a radio scenario, the precompiled logic cores of Agile Hardware would be various radio functionality modules such as scramblers, filters, synchronization loops etc. This is parallel to the concept of having a C++ signal processing components library in the GNU Radio.

2.3 Wireless Personal Area Networks

Wireless Personal Area Network (W-PAN) is a network that wirelessly interconnects devices centered around a person's workspace. This fundamentally means that these networks typically support short to medium range communications. W-PANs find applications in a wide variety of areas apart from interconnecting personal computing and communication devices. Figure 2.4 depicts comparison between most popular W-PAN and W-LAN standards in terms of range of communication and data rates supported. This work uses the Zigbee standard to deploy a proof-of-concept radio, since it offers a simple and efficient design. Design of a more complex, high data rate UWB radio is discussed in Chapter 5. The following subsection provides an overview of the Zigbee standard, including all the details pertaining to the design of its receiver discussed in Chapter 4.

2.3.1 The Zigbee Standard

Zigbee is a set of specifications designed for W-PANs (Wireless Personal Area Networks) using small, low-power digital radios. It is targeted for the applications that need low data rate, low latency and long battery life, such as sensor networks and remote control systems. It generally finds applications in home automation, telecommunication applications, personal and hospital care and commercial building automation. The Zigbee standard is maintained and published collectively by a group of companies, called the Zigbee Alliance [20]. Figure

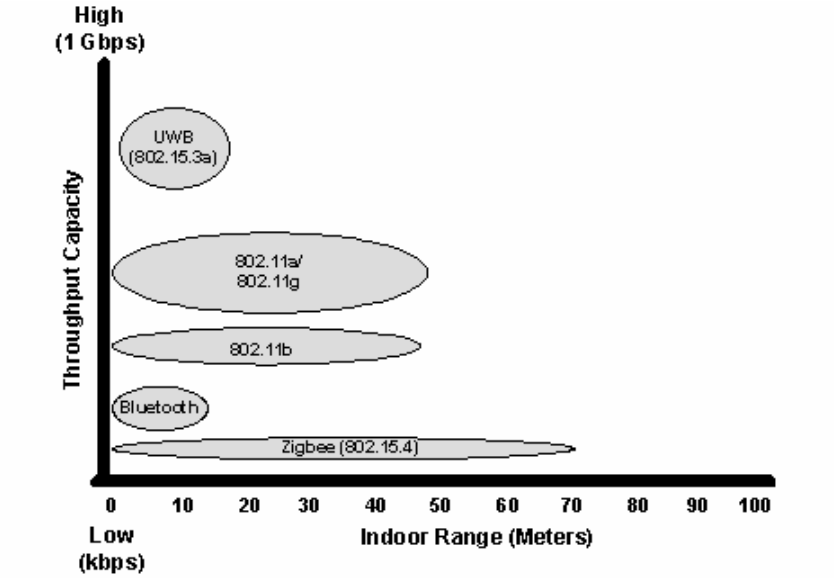


Figure 2.4: Comparison of W-PAN standards (figure from [17]).

2.5 shows the Zigbee protocol stack.

The Zigbee PHY layer is based on the IEEE standard 802.15.4. The PHY layer of 802.15.4 (2450 Mhz) specifies O-QPSK modulation along with half-sine pulse shaping. O-QPSK modulation is a type of quadrature phase shift keying in which the data on the I and Q channels are at an offset of half symbol period from each other. It also employs a Direct Sequence- Spread Spectrum (DS-SS) for better performance in the low-power radio link. Each PHY-layer frame consists of PHY payload, wrapped with the preamble and other control data. The relevant details of the 802.15.4 PHY-layer specifications are presented next.

802.15.4 PHY layer specifications

The IEEE standard 802.15.4 is specified for four PHYs- three at 868/915 MHz employing different modulation/ DS-SS schemes and one at 2450 MHz employing OQPSK modulation.

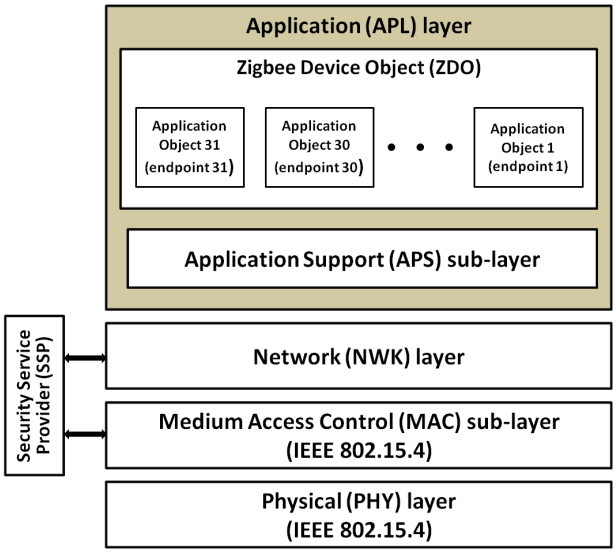


Figure 2.5: Zigbee stack [19].

The PHY of interest for this work is 2450 MHz. The PHY layer frame format and 2450 MHz PHY specifications are described below [18].

The PHY layer data packet is also known as PPDU (PHY Protocol Data Unit). It consists of three basic components-

1. Synchronization Header (SHR): The synchronization header is used for symbol synchronization at the receiver side and consists of the preamble and Start-of-Frame Delimiter (SFD) fields.

   (a) Preamble: is used by the receiver for chip and symbol synchronization of the incoming bit stream. A stream of binary zeros is transmitted along the whole length of preamble. The length of the preamble is different for different PHYs defined in the IEEE 802.15.4 standard. For the 2450 MHz PHY, the length of preamble is specified to be 4 octets (8 symbols).

   (b) SFD: Start-of-frame delimiter marks the end of the synchronizer header (i.e. preamble) and the beginning of the packet. The length of SFD field for the

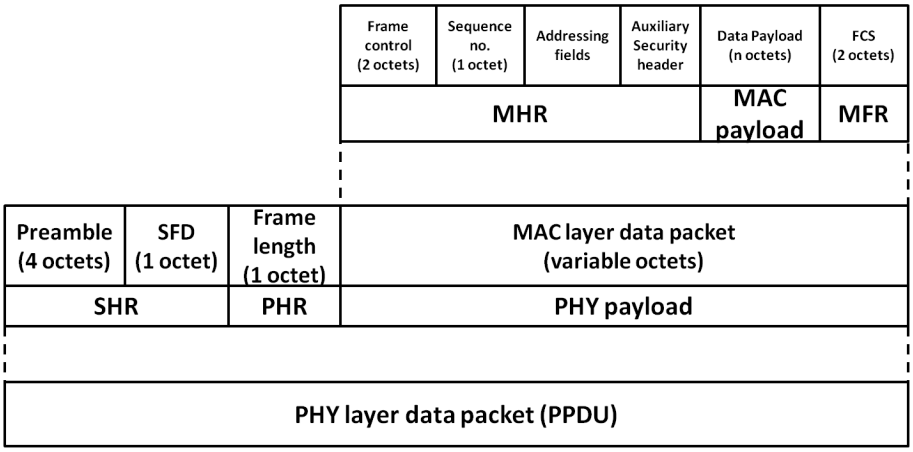


Figure 2.6: IEEE 802.15.4 PHY frame format [18].

2450 MHz PHY is one octet (i.e. 2 symbols). It is defined by a special symbol 11100101 (in binary).

2. PHY Header (PHR): This consists of the frame length field, which specifies the size of the PSDU (PHY payload) in terms of number of octets. The length of this field is 7 bits and the size of PSDU can range between 0 and 127 (maximum possible number of octets contained in a PSDU).

3. PHY payload: The PHY payload basically is same as the MAC sub-layer frame (MPDU). The MPDU (MAC Protocol Data Unit) consists of a MAC header (MHR), MAC Footer (MFR) and the data payload. The MHR and MFR include MAC layer control information fields such as frame control, sequence number, addressing fields, auxiliary security header and the frame check sequence.

The leftmost field of the PPDU is transmitted and received first. In multiple octet fields, least significant octet is transmitted/received first and each octet is transmitted/received LSB first.

The 2450 MHz PHY has a raw data rate of 250 kbps and it uses a 16-ary quasi-orthogonal modulation. The data bits are grouped in 4 bits resulting in 16 different symbols. Each of

these symbols is assigned a 32-bit PN sequence as shown in Table 2.1. The PN sequences are related to each other by circular shifts and/or conjugation. The total chip rate is 2 Mcps; 1Mcps per channel. These chip sequences are then modulated using OQPSK scheme with half-sine pulse shaping. Chapter 4 describes these techniques in more detail.

Table 2.1: Symbol to chip mapping for 802.15.4 [18].

Data symbol (decimal)	Data symbol (binary)	Chip values (c_0 c_1 ... c_{30} c_{31})
0	0000	11011001110000110101001000101110
1	1000	11101101100111000011010100100010
2	0100	00101110110110011100001101010010
3	1100	00100010111011011001110000110101
4	0010	01010010001011101101100111000011
5	1010	00110101001000101110110110011100
6	0110	11000011010100100010111011011001
7	1110	10011100001101010010001011101101
8	0001	10001100100101100000011101111011
9	1001	10111000110010010110000001110111
10	0101	01111011100011001001011000000111
11	1101	01110111101110001100100101100000
12	0011	00000111011110111000110010010110
13	1011	01100000111011110111000110001001
14	0111	10010110000001110111101110001100
15	1111	11001001011000000111011110111000

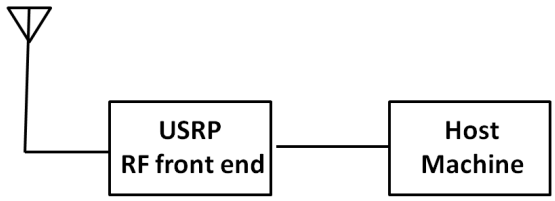
Chapter 3

Enhanced GNU Radio for Hardware Acceleration

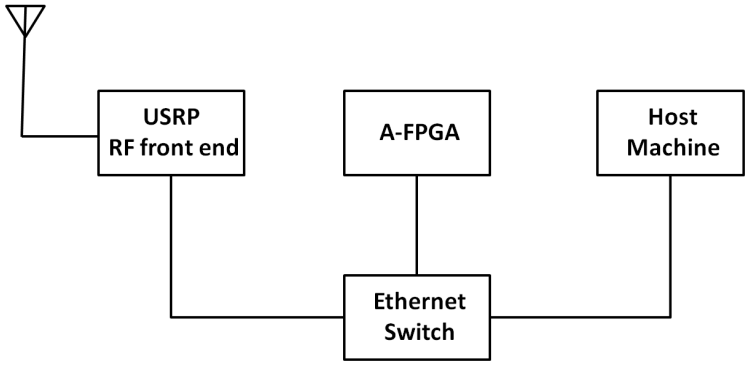
This chapter details the structure and usage model of the modified GNU Radio platform. As mentioned in Chapter 1, the principal concept is to augment the standard GNU Radio for greater computational power so as to allow highly complex radio designs. One of the important factors while designing this improved platform was to retain the ease-of-use of the standard GNU Radio. Hence, most of the modifications made to the GNU Radio are abstracted from the user [10].

3.1 System Layout

The enhanced GNU Radio includes an auxiliary FPGA co-processor (referred to as A-FPGA henceforth) in such a way that a system can take advantage of it only if the user chooses to do so. In this way, the platform maintains all the features of a conventional GNU Radio with an option of hardware acceleration. The following subsections explain how this is done



(a) Hardware layout of standard GNU Radio



(b) Hardware layout of enhanced GNU Radio

Figure 3.1: Hardware layout- Standard versus Enhanced GNU Radio.

by describing hardware set-up and logical layout of the enhanced GNU Radio.

3.1.1 Hardware Set-up

Figure 3.1 shows the connections of hardware devices for standard and modified GNU Radio platforms. In a standard GNU Radio design, there is a direct link for radio data between USRP and the host machine. This link is maintained in the modified GNU Radio via the switch. If a user chooses to include the hardware acceleration co-processor in a design, then the radio data would be diverted to the A-FPGA for the required signal processing on its way between the USRP and host machine. Otherwise, a normal GNU Radio flow would be used only with an extra switch in the path. It has been shown in [10] that adding the switch does affect latency of the system; however, it is not overwhelming. The reason behind using an Ethernet switch as the central node is that GigE is a common interface offered by the

devices/boards used for all the other three nodes. The GigE interface offers a high bandwidth of 1 Gbps for data flow.

Dataflow

On the transmit datapath, the packets from host machine are directed to the A-FPGA in the enhanced GNU Radio, instead of USRP in the standard GNU Radio. This is done by setting the destination MAC address in the packet header to either the A-FPGA or USRP MAC address as desired.

In case of the receive path on a standard GNU Radio, the USRP sends the received radio packets to the same MAC address from where it obtains a request, that is the host machine. In the enhanced GNU Radio, though the host machine sends request for received radio packets, the USRP is supposed to forward them to the A-FPGA for demodulation. To achieve this, a new command `start_rx_stream_A-FPGA` to be included in the top-level Python script, was created in the enhanced GNU Radio. The USRP2 firmware was modified to accomodate this change. This command allows the user to specify a destination MAC address for radio data packets, while using the command in the top-level Python script.

3.1.2 Logical Layout

Logical connections between the host machine, radio front end and the A-FPGA are depicted in Figure 3.2. Using different packet formats and addressing schemes, three bi-directional links are created between the three nodes:

1. Sample channel: Used as data link for transferring modulated samples between the A-FPGA and USRP.

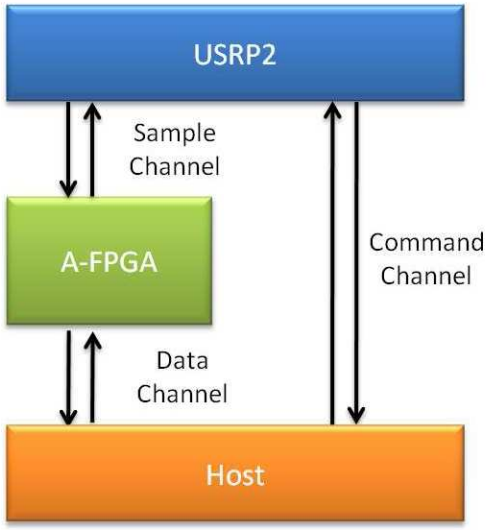


Figure 3.2: Logical layout of Enhanced GNU Radio (figure from [10])

2. Data channel: This link between the host machine and A-FPGA is used to send and receive raw radio data. It can also be used to send configuration commands from host machine to A-FPGA for radio modification using dynamic reconfiguration.

3. Command channel: The command channel is mainly used to configure the USRP by sending commands from host machine. It is also required to support the standard GNU Radio functionalities.

There exists three different packet formats as shown in Figure 3.3 for the three channels. Type A packet is designed for the Data channel. It consists of Ethernet header which includes source and destination MAC addresses and the Ethernet type field. The *type* field is used by the A-FPGA to identify whether a data or configuration packet was sent. In case of data packet, it also uses the *size* field to make sure that it receives the entire length of data that was sent.

Type B packet is used for the Sample channel and is already defined in the standard GNU Radio. It consists of the radio data samples and a USRP2 header which includes an Ethernet

Figure 3.3: Packet formats for different channels (figure from [10])

header in it. This type of packet is used by the GNU Radio to transfer radio samples between the host machine and USRP2. So, the USRP2 is already designed to handle this packet format. The A-FPGA is set up in such a way that it can accept and send the modulated data samples using Type B frame format.

Type C format is used by the host machine to configure and control the USRP2. This format is same as that defined by the GNU Radio for command data between the host machine and USRP2. The packet consists of the USRP2 header, an operation identifier and other data associated with the command. Each type of command has its unique operation identifier.

3.2 System Usage

Figure 3.4 gives an overview of usage model of the system. The first set of commands in the Python script are meant to configure the USRP. Next, all the signal processing blocks are connected to set-up the data path. Finally, the flow graph is wired up together and the script is ready to be executed. The figure also presents how each section of code targets different devices for execution and how different radio modules correspond to the flowgraph built using Python.

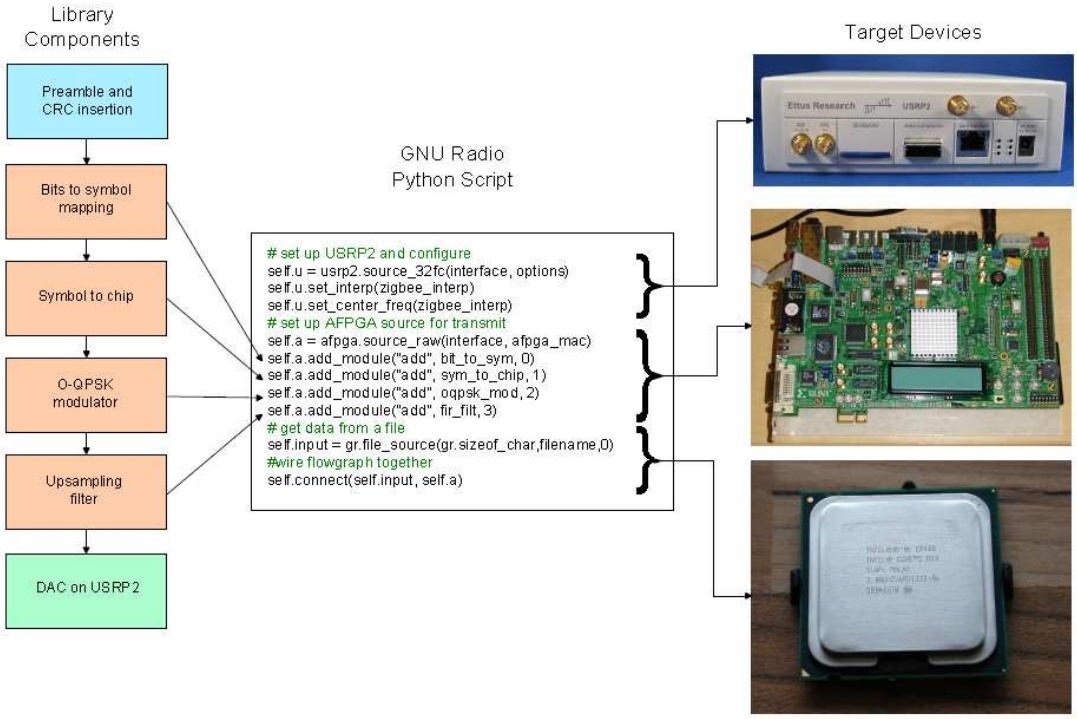


Figure 3.4: System usage (figure from [10])

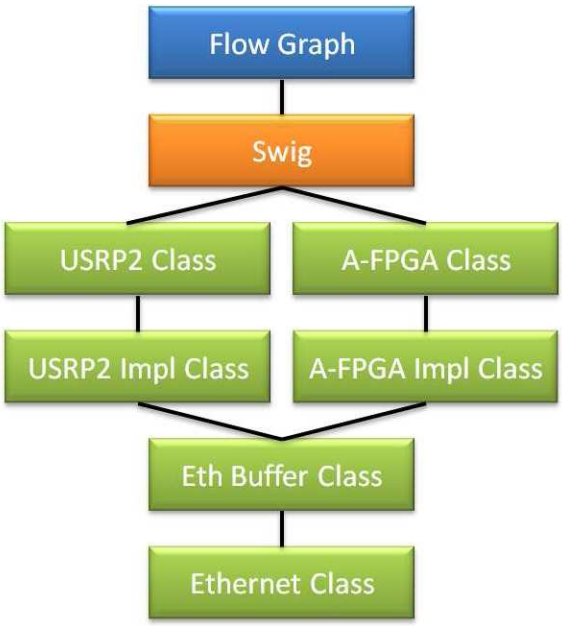


Figure 3.5: Software layout (figure from [10])

3.3 Software Layout

Figure 3.5 shows the software chain for enhanced GNU Radio. The changes were made to standard GNU Radio in such a manner that, the way top level Python script is implemented, remains consistent. Similar to the standard GNU Radio, the Python script includes commands to configure USRP2 first, then add radio modules and lastly wire up all the blocks. The only change is that the added radio modules are on the A-FPGA instead of host machine. Redirection of data takes place in the lower levels of SWIG and C++ implementations. SWIG is an open source project used for connecting the underlying C++ functions to the top level Python code [24]. All the changes that are made follow the coding standards and guidelines of the standard GNU Radio. Thus, the enhanced platform remains compatible with the normal GNU Radio.

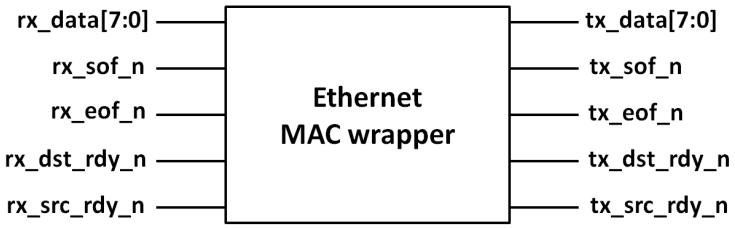


Figure 3.6: Black-box representation of Ethernet module. (figure from [10])

3.4 A-FPGA Hardware Modules

This section describes the hardware modules to be included on the A-FPGA platform that are essential for the operation of enhanced GNU Radio. This includes the HDL implementation for Ethernet, FIFOs for data flow control and ethernet frame encoder/decoder. The HDL modules for Ethernet and FIFOs can be easily generated using the Xilinx Coregen tool.

3.4.1 Ethernet Module

The Ethernet MAC wrapper module produced by Coregen supports a bi-directional Ethernet link and implements a link-layer protocol along with FIFO-based data flow control. Figure 3.6 shows a black-box representation of the Ethernet module. It uses the start-of-frame (`tx_sof_n, rx_sof_n`) and end-of-frame (`tx_eof_n, rx_eof_n`) signals to identify frame boundaries. The Ethernet MAC wrapper also uses the source ready signals (`tx_src_rdy_n, rx_src_rdy_n`) and destination ready signals (`tx_dst_rdy_n, rx_dst_rdy_n`) to control the transfer of incoming and outgoing frames. The frame-data is transferred one byte at a time through the 8-bit wide data port (`tx_data, rx_data`).

3.4.2 Ethernet Frame Encoder/Decoder

The encoder module is necessary to frame the outgoing data from A-FPGA in the correct frame format. If demodulated data is to be sent from a receiver on the A-FPGA to host machine, then it is encoded using Type A frame format. Type B frame format is used for modulated radio samples from a transmitter on A-FPGA going to the USRP2. The corresponding headers are appended to the data by the encoder. The encoder is realized in HDL using a straight-forward state-machine.

The decoder is used to strip the Ethernet header off the incoming packets and present extracted data to further logic stages. After parsing the header information, the decoder confirms validity of the data and forwards it to the proper radio module on the A-FPGA. A Type B frame that comes from the USRP2 is directed to the receiver for demodulation, where-as a Type A frame coming from the host machine is sent to the transmitting radio. The decoder block is implemented using a dual-port-RAM and a control state machine.

3.4.3 Flow Control FIFOs

At the end of both transmit and receive radios, FIFOs are used to queue the data output of the radios. This is done to support different clock rates for the Ethernet MAC and radio circuits on the A-FPGA. The FIFOs used in this work can hold maximum 10 frames of 1500 bytes each. Hence, the platform can support a clock rate ratio (between the radio clock and Ethernet MAC clock) of up to 10, without any loss of data. This ratio between read/write clocks can be increased by using a smaller frame size or alternatively by implementing a bigger FIFO. The Zigbee radio implementation in this work uses a clock rate of 125 MHz for both the radio circuit and the Ethernet MAC. In this case, since the data is read and written to the FIFO at the same rate, there is no possibility of any data loss.

Chapter 4

Zigbee Receiver: Design and Implementation

The Zigbee standard allows for a simple and efficient receiver design, which makes it an ideal proof-of-concept design. This chapter describes the design and HDL implementation of the Zigbee receiver.

4.1 Design of Zigbee Receiver

As mentioned in Chapter 2, Zigbee (2450 MHz) uses OQPSK modulation with half-sine pulse shaping to modulate the chip sequences representing each data symbol. The chips are alternately assigned to I and Q channel data. To get the required offset, Q-phase chips are staggered by half the chip period with respect to the I-phase chips. This is shown in Figure 4.1.

In baseband, this chip sequence undergoes a half-sine pulse shaping. Figure 4.2(a) depicts a

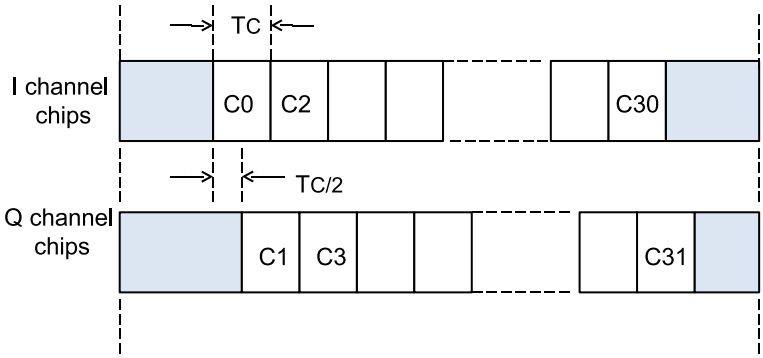


Figure 4.1: OQPSK chip sequence [18].

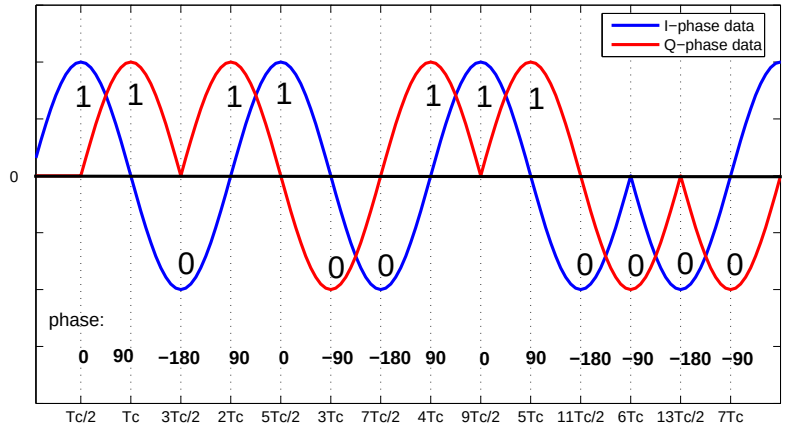
baseband signal for a decimal symbol zero (i.e. all binary 0's). Due to the offset of half-chip duration between I and Q data, the maximum phase change in an OQPSK signal is +/- 90°. This can be seen from Figure 4.2(b).

Commonly, O-QPSK demodulators use a Costas loop circuit for the frequency and phase error correction; along with a timing error adjustment scheme. However, these techniques are computationally demanding. In this thesis, a technique called differential demodulation of O-QPSK is used [26, 27]. This scheme has an advantage of not requiring frequency and phase offset correction. For timing adjustment, the peaks of cross-correlation between the received chips and known spreading signal is used.

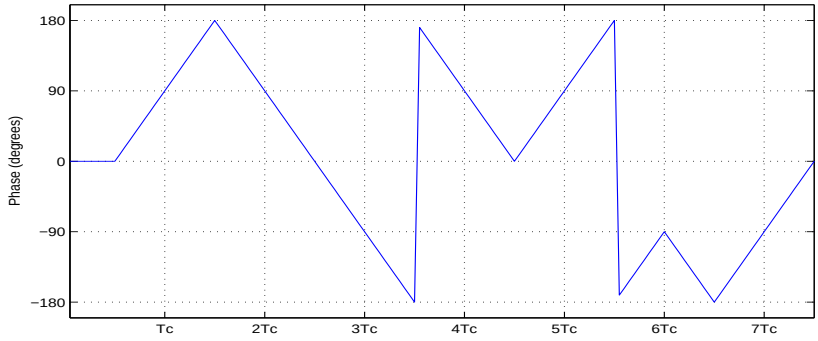
In the differential demodulation method, it is important to look only at the phase change between two consequent samples. Because of this, the absolute phase information is not required. The demodulator basically tracks only the direction of change of phase. This change of phase is expressed by the following equation [26]:

$$\Delta\Theta = \frac{dI}{dt}.sign(Q) - \frac{dQ}{dt}.sign(I)$$

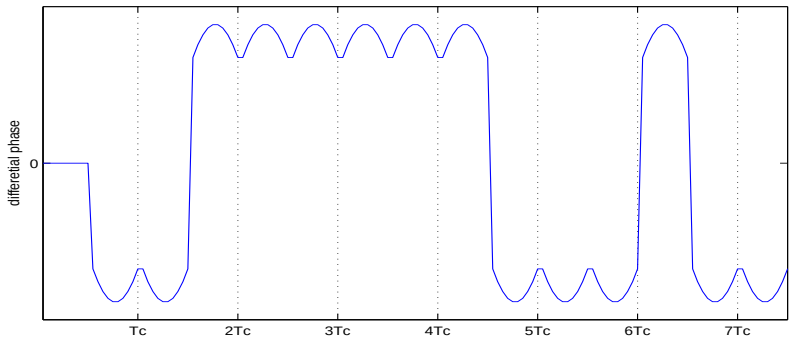
Figure 4.2(c) shows the $\Delta\Theta$ signal for the baseband signal considered in Figure 4.2(a). It can be observed that the $\Delta\Theta$ signal is negative when the phase of the signal increases and vice-versa. Since there are sixteen fixed chip sequences for each of the sixteen symbols, it is easily



(a) Zigbee baseband signal for symbol '0'



(b) Phase of the baseband signal



(c) Differential phase signal

Figure 4.2: OQPSK baseband signal, its phase and differentiated phase

possible to generate the sequence of $\Delta\Theta$ values for each symbol. Table 4.1 [26] contains this new set of spreading codes using the differential phase signal. These new spreading codes are also related to each other by cyclic shifts and conjugation.

After this, the received $\Delta\Theta$ sequence is correlated with the sixteen known sequences. The symbol with highest correlation is picked as received symbol. However, the correlation process depends upon the state of the demodulator. If the demodulator is in the state of waiting for a new frame, then, it keeps looking for a zero symbol (since the frame starts with a preamble that is comprised of eight 'zero' symbols). Thus, the incoming differential phase samples are correlated only with the zero symbol in this state. As soon as it encounters the first zero symbol in preamble, the demodulator locks itself. Now it is not required to monitor the correlation values continuously. The demodulator picks the highest-correlation symbol on every thirty second new sample after this synchronization event. It progresses to reading the frame data after preamble, SFD and frame length field detection. This demodulator state machine diagram is shown in Figure 4.3. Apart from considering correlation values for every thirty-second sample, values obtained for cross-correlations starting one sample before and after, are also taken into account. If the value for a shifted correlation is higher, then it is picked and an adjustment sample is inserted or removed. This is done to achieve timing adjustment in the demodulator circuit.

Following section gives a detailed description of the HDL implementation of this receiver design on an FPGA platform.

4.2 HDL Implementation of a Receiver

As mentioned in Chapter 3, the FPGA gets its input radio data from USRP2 through the Ethernet port. On the FPGA, the input radio data is first unwrapped from the Ethernet frame using the Ethernet frame decoder. This block parses through the Ethernet header and

Table 4.1: Symbol to $\Delta\Theta$ spreading code mapping [26].

Data symbol (decimal)	Data symbol (binary)	$\Delta\Theta$ spreading code
0	0000	00111111000100001010001100100110
1	1000	01100011111100010000101000110010
2	0100	00100110001111110001000010100011
3	1100	00110010011000111111000100001010
4	0010	10100011001001100011111100010000
5	1010	00001010001100100110001111110001
6	0110	00010000101000110010011000111111
7	1110	11110001000010100011001001100011
8	0001	11000000111011110101110011011001
9	1001	10011100000011101111010111001101
10	0101	11011001110000001110111101011100
11	1101	11001101100111000000111011110101
12	0011	01011100110110011100000011101111
13	1011	11110101110011011001110000001110
14	0111	11101111010111001101100111000000
15	1111	00001110111101011100110110011100

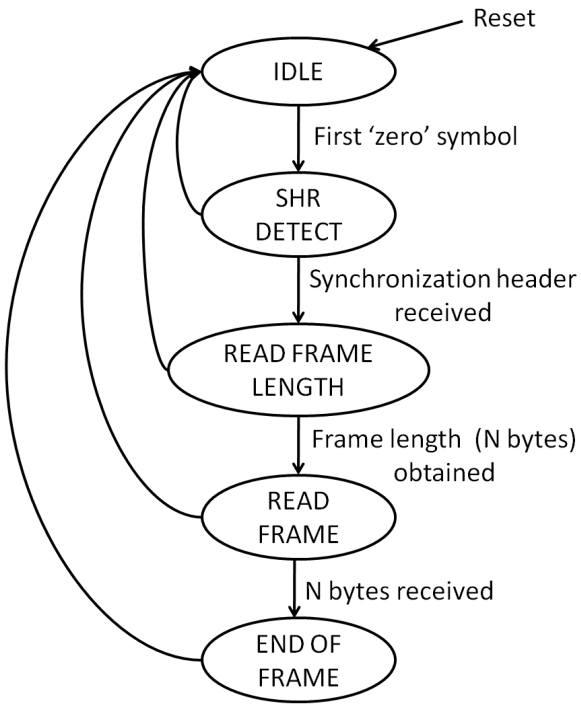


Figure 4.3: 802.15.4 Demodulator state machine.

presents the payload radio data to the next module. This extracted radio data is a stream of alternately interleaved I-phase and Q-phase data samples. The de-interleaved I and Q phase samples are inputs of the demodulator. The demodulator output, that is the received frame is encoded in a proper Ethernet frame format and sent to the host machine via the same Ethernet port. All the logic related to interfacing Ethernet that lies on the A-FPGA was described in Chapter 3. The following section describes the HDL implementation of the Zigbee demodulator design on the A-FPGA.

Figure 4.4 shows a top level block diagram of the demodulator. Each block of the demodulator is described in detail below.

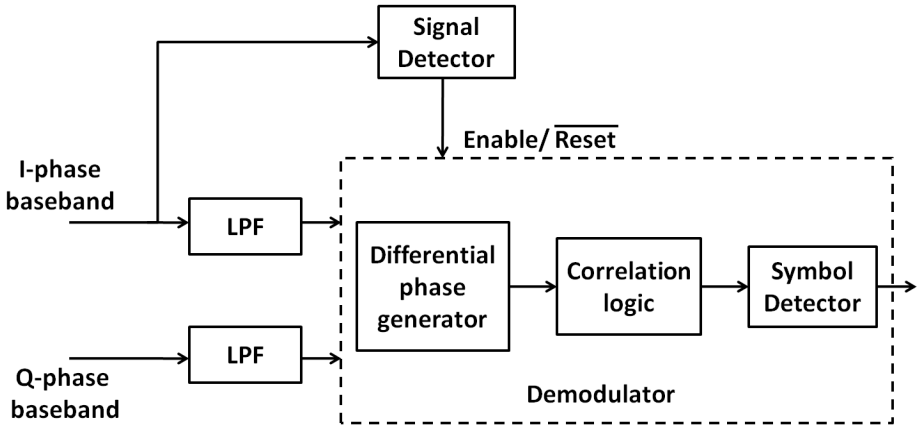


Figure 4.4: Top-level block diagram of demodulator.

4.2.1 Matched Filter

Data on both the channels are passed through matched filters first. These filters have a half-sinusoidal pulse shaped impulse response and they help in enhancing the signal to noise ratio.

4.2.2 Signal detector circuit

The signal detector circuit is used to enable the demodulator upon detection of a Zigbee signal. Since the demodulator needs to synchronize at the start of each frame, it becomes important to reset the demodulator before a new frame begins. The signal detector circuit sets the enable signal when it senses that the amplitude of the received signal is above a preset threshold. This turns the demodulator on, which then starts polling for the synchronization header fields. If the signal level falls below the threshold, then the whole demodulation system is turned off and reset. Figure 4.5 shows the logic implementation of signal detector circuit. The ON/OFF counters are used to avoid false alarms while turning the system on or off. The signal level should remain above or below the threshold for a pre-defined

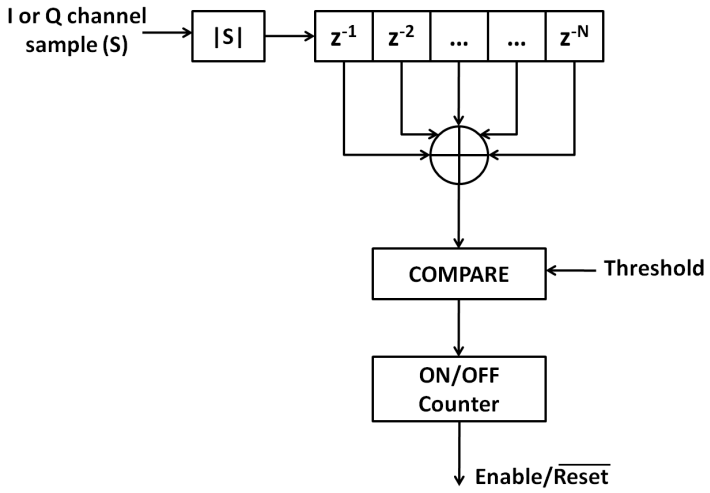


Figure 4.5: Block diagram of signal detector.

number of clock cycles before the enable signal is turned high or low. The threshold value was determined experimentally for this design, based on the average amplitude of the noise as well as the Zigbee signal.

4.2.3 Differential Phase Generator

To produce the $\Delta\Theta$ signal, the differential phase generator block directly follows the mathematical equation that defines it. This is illustrated in Figure 4.6.

4.2.4 Correlation Logic

The correlation block is used to evaluate the cross-correlation value between the received $\Delta\Theta$ signal and the sixteen Zigbee symbols. Since each symbol is represented using a 32-bit wide spreading sequence, every new (1-bit wide) value of the incoming $\Delta\Theta$ signal is pushed

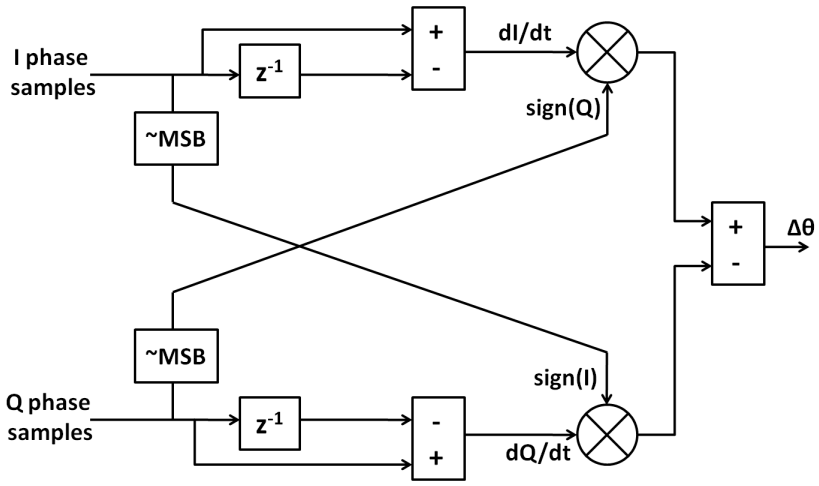


Figure 4.6: Block diagram of differential phase generator.

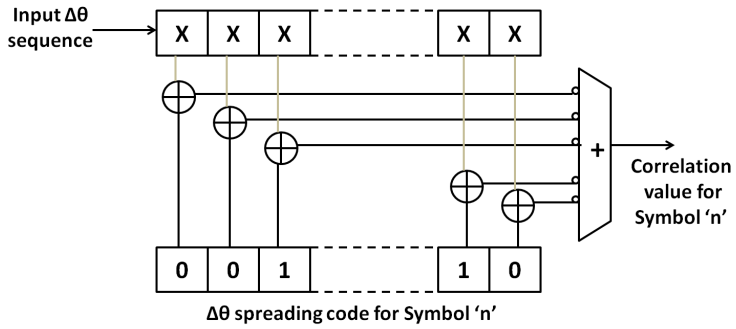


Figure 4.7: Block diagram for correlator logic.

in a 32-bit wide register. The correlator block essentially counts the number of matching bits between a given symbol and the last 32-bits of the received $\Delta\Theta$ signal. As seen from Figure 4.7, this is done by summing up the XNOR output for corresponding individual bits.

4.2.5 Symbol Detector

This is the most important logic module of the design because it implements the timing synchronization as well as chip-to-symbol functionality of the receiver. It directly follows the state machine shown in Figure 4.3 to obtain the symbols in the frame that was sent. Figure 4.8 shows the logic used to determine the sent symbol. The cross-correlation values for all the symbols are compared with a threshold value and the one that exceeds the threshold is picked as the sent symbol. The maximum possible cross-correlation value is 32. A threshold of 24 was chosen to allow for a lower probability of miss for the receiver. Moreover, this does not result in a higher probability of false alarm, since the receiver matches a long sequence of symbols in the synchronization header with the incoming symbols for frame detection.

As mentioned above, when the demodulator state machine is in the "IDLE" state and polling for a new frame to arrive, the circuit keeps looking for a 'zero' symbol. As soon as this event occurs, a 'start of frame' signal is activated in the circuit, which remains ON until the end of the frame. Once the start of frame is detected, the state machine proceeds through the remaining states to obtain the sent frame data after detecting the synchronization header (preamble and start of frame delimiter fields) and the frame length information. The 'start of frame' signal also activates a 5-bit counter that keeps track of the 32 chips and marks the arrival of a new symbol. This counter is reset along with the 'start of frame' signal every time a frame ends. In case no symbol is detected at the end of every thirty second count, then the frame is dropped and state of the state-machine is reset to IDLE.

Instead of looking for the cross-correlation peaks only at every 32nd count of chip-counter, the circuit also considers the correlation peaks a count before and after that. In case a peak is obtained on one of those counts, then the corresponding symbols are accepted as sent symbols and the counter value is adjusted to accomodate this timing error. This is necessary, since there is no other solid timing-synchronization mechanism employed in the receiver. Such a timing-adjustment strategy was possible only due to the DS-SS technique

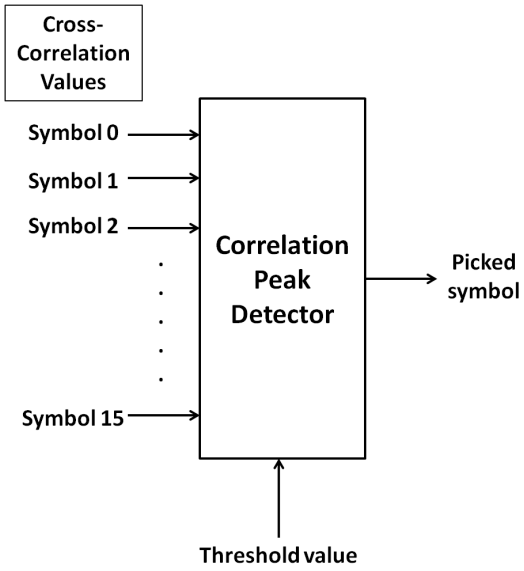


Figure 4.8: Logic to pick symbol with biggest cross-correlation.

employed by Zigbee PHY layer. This saves implementing a complex timing error recovery circuit and keeps the receiver design simple and efficient.

Chapter 5

Design of a UWB System on Enhanced GNU Radio

An important rationale behind enhancing the GNU Radio for hardware acceleration, was to enable radio developers to realize complex, high data rate radio systems on a software radio platform. Thus, it is important to consider a high data rate radio application on the enhanced GNU Radio. This chapter describes design of one such radio system - a Multiband OFDM-based UWB wireless link.

5.1 High Rate W-PANs

As mentioned in Chapter 2, the W-PANs are an important class of wireless standards. The IEEE 802.15.3 was an effort to create a standard for high data rate W-PANs, using a UWB PHY layer [28]. It is meant for applications that need high bandwidth, such as imaging and multi-media. A UWB W-PAN would allow to seamlessly transfer data for these emerging applications within a network of consumer electronics, personal computing peripherals and

mobile devices [17]. Two types of technologies are considered for a UWB-PHY layer: Multi-band Orthogonal Frequency Division Multiplexing (MB-OFDM) and Direct-sequence UWB (DS-UWB) [29].

5.1.1 DS-UWB Systems

The direct sequence UWB is implemented by direct modulation of data into a sequence of impulse-like waveforms which occupy the available ultra-wide bandwidth. Hence, it is also known as carrier-free and impulse-communications [29]. Such a single band system can support multiple users by using time-hopping or direct-sequence spreading techniques. Although, DS-UWB systems offer low implementation costs and simple design, they are not flexible in terms of spectrum-management. Also, making RF and analog circuits and high-speed ADCs to handle such ultra-short pulses is very challenging.

5.1.2 Multiband OFDM Systems

The multiband approach overcomes the drawbacks mentioned in the previous section, since the ultra-wide band is broken into much smaller sub-bands. Since this approach allows processing of data over a narrower bandwidth, it reduces overall design complexity and improves the spectral flexibilty. The underlying OFDM technology also results in an efficient multipath energy capture. This work considers a multiband OFDM UWB design due to these advantages and also because of the related on-going work on OFDM-radio development in the CCM Lab.

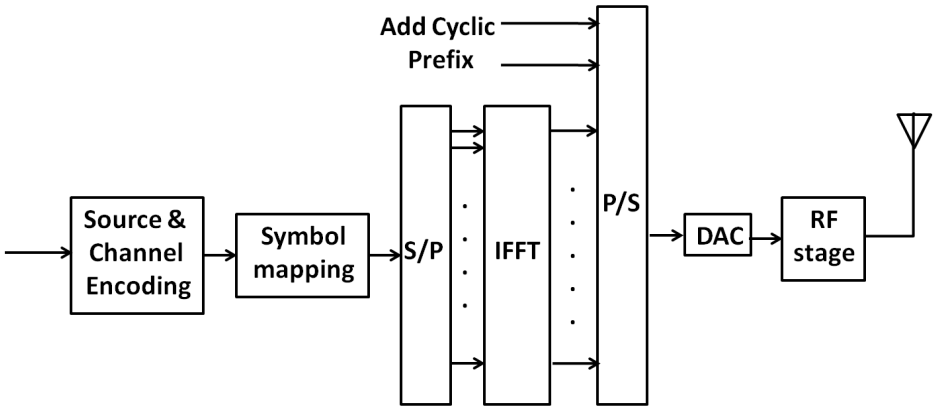
5.2 Proposed UWB Radio Design on Enhanced GNU Radio

The Federal Communications Commission (FCC) defines UWB as any signal that occupies at least 500 MHz of bandwidth within the 7.5 GHz wide spectrum between 3.1 and 10.6 GHz [30, 31]. In a multiband OFDM based UWB system, the UWB spectrum is divided into smaller sub-bands and uses multiple carrier frequencies to transmit information. In such a system, all the sub-bands can be employed simultaneously to achieve a high bit-rate; or several users can be supported by interleaving data over sub-bands across both time and frequency. As the name implies, OFDM technique is employed to modulate data on each of these sub-bands. The following subsection provides fundamentals of the OFDM technique followed by other details of the design.

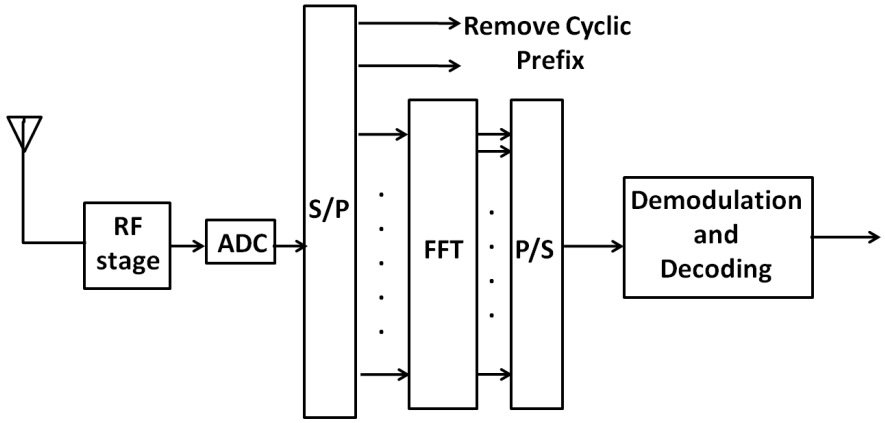
5.2.1 OFDM Fundamentals

OFDM is essentially an efficient frequency division multiplexing scheme that uses a large number of closely spaced orthogonal sub-carriers to transmit data. The most important advantage of OFDM is simple equalization, since each sub-carrier can be viewed as an individual narrow-band link. By the virtue of orthogonality, the spectra of adjacent subcarriers can be allowed to overlap, resulting in a good bandwidth efficiency. OFDM is also known to have a good performance in presence of a fading channel due to frequency diversity.

Figure 5.1 shows an OFDM system. Each narrow sub-carrier is modulated using linear modulation techniques such as BPSK and QPSK. The complex symbols are then frequency-multiplexed on the parallel sub-carriers using I-FFT. Overlapping adjacent sub-carrier spectra is possible by choosing the spacing between sub-carriers such that each sub-channel has a null at other subcarrier frequencies. This can be achieved by selecting tone seperation to



(a) Transmitter



(b) Receiver

Figure 5.1: Transmitter and receiver of an OFDM system (adapted from [29]).

be equal to inverse of signal symbol duration.

Instead of OFDM, Filter Bank Multi-Carrier (FBMC) technology is also considered to be a good candidate as an underlying technology for a multiband UWB radio. Succeeding OFDM, FBMC is an evolving multi-carrier technology that applies filter banks to omit the severe out-of-band leakage of OFDM.

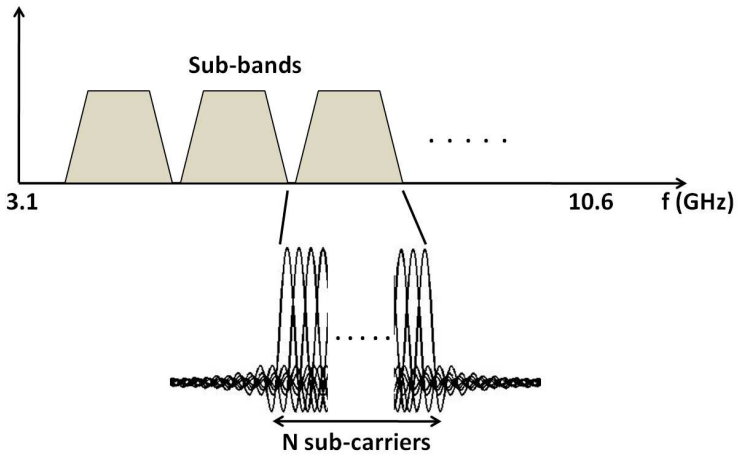


Figure 5.2: UWB multiband OFDM spectrum (figure from [29]).

5.2.2 Multiband System Model

A multiband OFDM system consists of number of sub-bands; where each sub-band in turn employs the OFDM scheme. These sub-bands can be used simultaneously to obtain a single high data rate link or a multiple-access can be enabled using frequency-hopping.

For each sub-band, a cyclic prefix of length T_{CP} as well as guard interval of length T_{GI} is appended. The cyclic prefix helps in mitigating the multi-path effects as well as transforms the multipath linear convolution into a circular convolution. Guard interval is meant to take care of the adjacent channel interference. It helps in relaxing the filter specifications for the adjacent channel rejection filters.

5.2.3 Computational Considerations

An OFDM link 18MHz wide, with 64 channels (each approximately 250 kHz wide) was developed in the CCM Lab at Virginia Tech. This system was developed on a Virtex 4 FPGA. Considering the utilization of critical FPGA resources (such as DSP48s) for this

system and extrapolating those for an UWB system, implementing such a high data rate link seems computationally plausible. Recently developed FPGA families such as the Virtex 7 are highly resouce-rich and would be excellent candidates for a UWB system implementation.

Chapter 6

Experimentation and Results

This chapter demonstrates the results from experiments and analyses of the implemented low-rate radio system. First, the system set-up used to verify the radio functionality is described. The subsequent sections include the results regarding software and hardware resource utilization as well as the radio link performance.

6.1 System Set-up

This section describes the set-up used for the enhanced GNU Radio as well as of the radio link that was used for functional verification of the receiver design.

6.1.1 Enhanced GNU Radio Platform Set-up

The hardware set-up for the enhanced GNU Radio platform is implemented using the following components- a Xilinx Virtex5 (LX110T) board from Digilent called the XUPv5 as

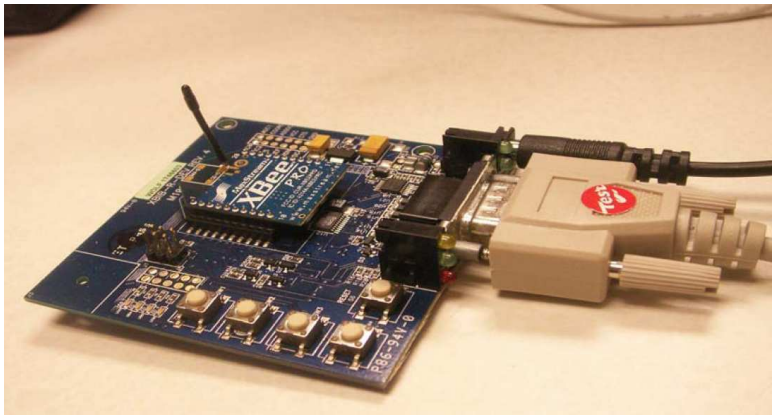


Figure 6.1: Xbee Zigbee transceiver module (Photograph from [25]).

the A-FPGA, the USRP2 radio front-end, a host machine running an Intel Core2Duo E8400 at 3GHz and a TRENDnet TEG-S80g gigabit Ethernet switch.

It is also important to note that the host machine runs Ubuntu 9.04 32-bit using version 3.2.2 of GNU Radio.

6.1.2 Radio Link Set-up

The Zigbee radio receiver is realized on the XC5-VLX110T FPGA device on the A-FPGA platform. All the radio modules described in Chapter 4, were synthesized for the Virtex5 FPGA using Xilinx ISE (Version 11.1) tools.

At the transmitter end, an off-the-shelf embedded Zigbee solution called Xbee is used. It is a product from Digi Solutions based on an ASIC CC2420 from Texas Instruments. The main advantage of Xbee modules is that they are low-cost, low-power and easy-to-use. Digi Solutions also provides a software called X-CTU, which enables users to configure and set up the modules easily. Figure 6.1 shows a photograph of an Xbee module.

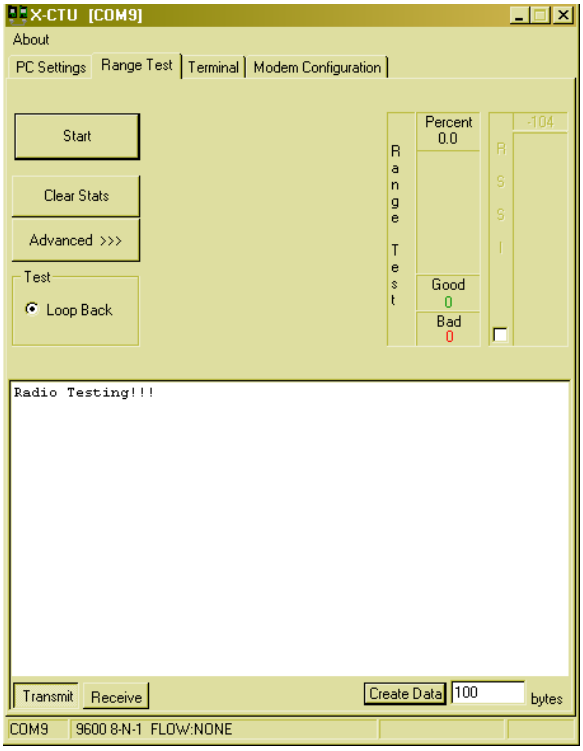


Figure 6.2: Screenshot of X-CTU panel at transmitter.

6.2 Radio Link Performance

This section discusses the performance of the Zigbee radio link developed in this work.

6.2.1 Functional Verification

Functional correctness of the Zigbee receiver was checked by verifying end-to-end communication between the transmitting and receiving nodes. As mentioned earlier, an XBee module was used as the transmitting node. Information frames were sent using X-CTU, a software associated with the XBee. Figure 6.2 shows a screenshot of X-CTU panel set to transmit a string of data.

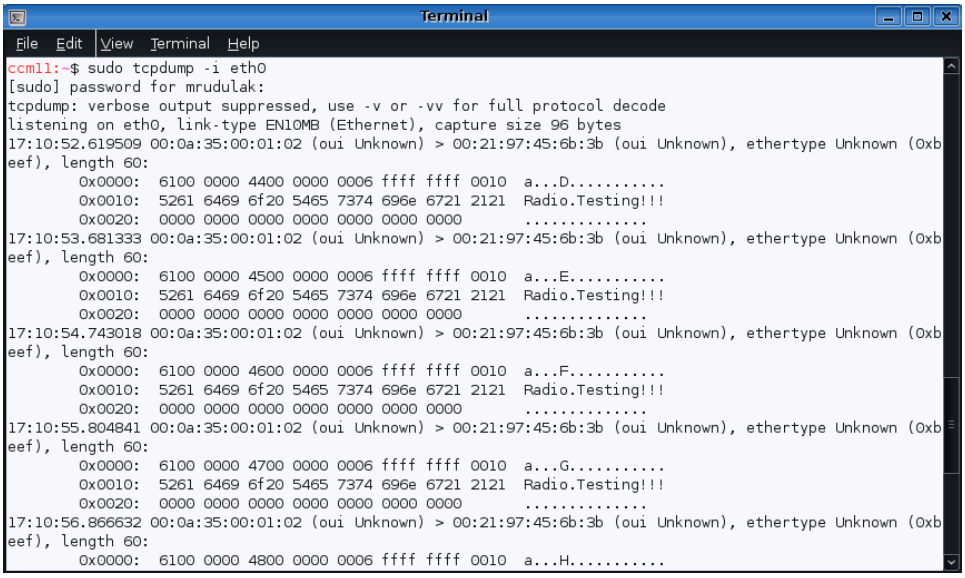


Figure 6.3: Screenshot of a tcpdump at receiver.

At the receiver, a *tcpdump* packet filter was set to capture frames arriving at the Ethernet port. Screenshot of a *tcpdump* at the receiver is captured in Figure 6.3. The figure demonstrates that the receiver demodulates correctly to obtain the same data string that was sent.

6.2.2 Error-rate Performance

The error rate performance of the system was studied using MATLAB simulations. Figure 6.4 shows the chip error rate performance of the differential-phase OQPSK demodulator against the theoretical BER performance of a quadrature-modulated system.

Figure 6.5 presents the overall performance of the Zigbee receiver. The simulated curve was generated for the symbol error rate (SER) of the Zigbee system that uses OQPSK modulation and DS-SS with sixteen quasi-orthogonal codes. The theoretical curve is plotted for the SER of a system that uses 16-ary orthogonal signaling.

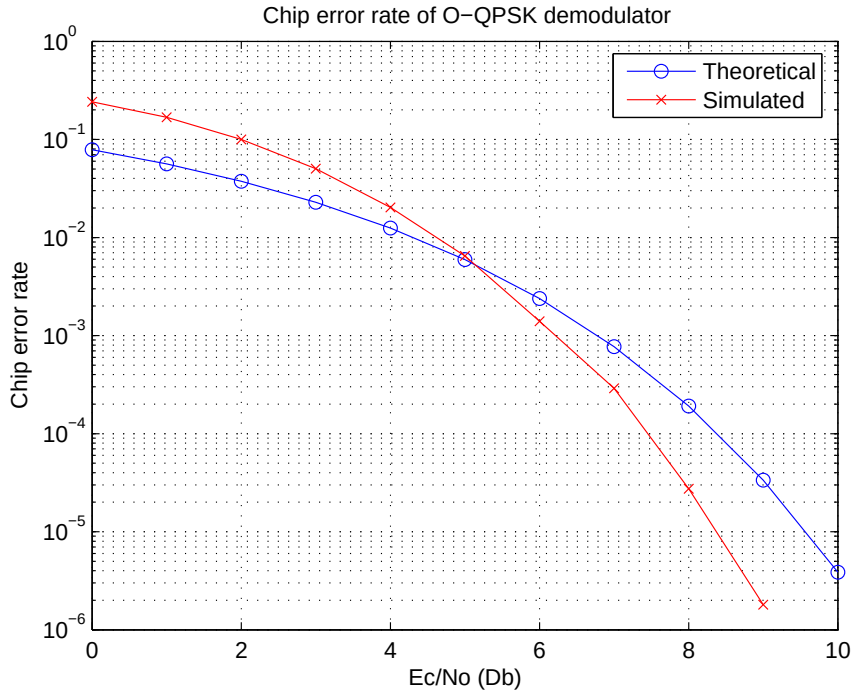


Figure 6.4: Chip error rate of OQPSK demodulator.

6.2.3 Acquisition Performance

The probability of miss for the Zigbee receiver over a range of symbol energy values was simulated. This is depicted in Figure 6.6. In a fully-implemented Zigbee system, frames are re-sent by the transmitter-end in case it fails to get the acknowledgement frames when they are dropped. Hence, a high probability of miss would result in a high overhead traffic.

6.3 Software Resource Utilization

Since enhanced GNU Radio shifts all the radio-related processing to the A-FPGA hardware, it is apparent that it would result in a reduced software resource utilization as compared to the normal GNU Radio. This can be verified by looking at Table 6.1. The table shows

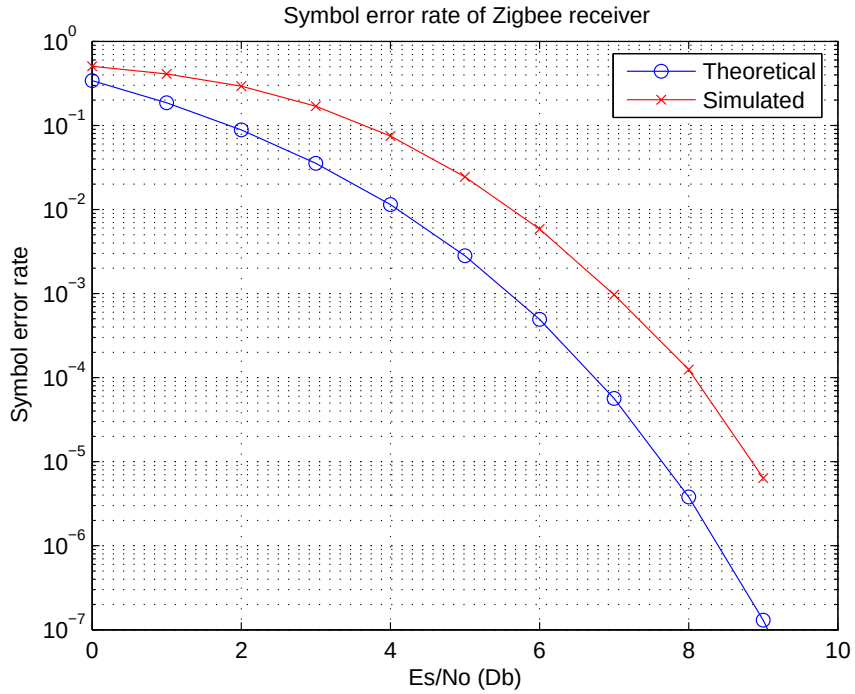


Figure 6.5: Symbol error rate of Zigbee receiver.

average CPU utilization for similar radio applications on both the GNU Radio platforms.

The enhanced GNU Radio was set to run the Zigbee receiver that mainly consists of an OQPSK demodulator. A similar radio receiver that uses a quadrature demodulator was generated and run on the standard GNU Radio. The average CPU utilization for Python scripts running both radios was noted using the *top* command in Linux on the host machine.

Table 6.1: CPU utilization for enhanced versus standard GNU Radio.

	Average CPU Utilization
Enhanced GNU Radio	0%
Standard GNU Radio	16%

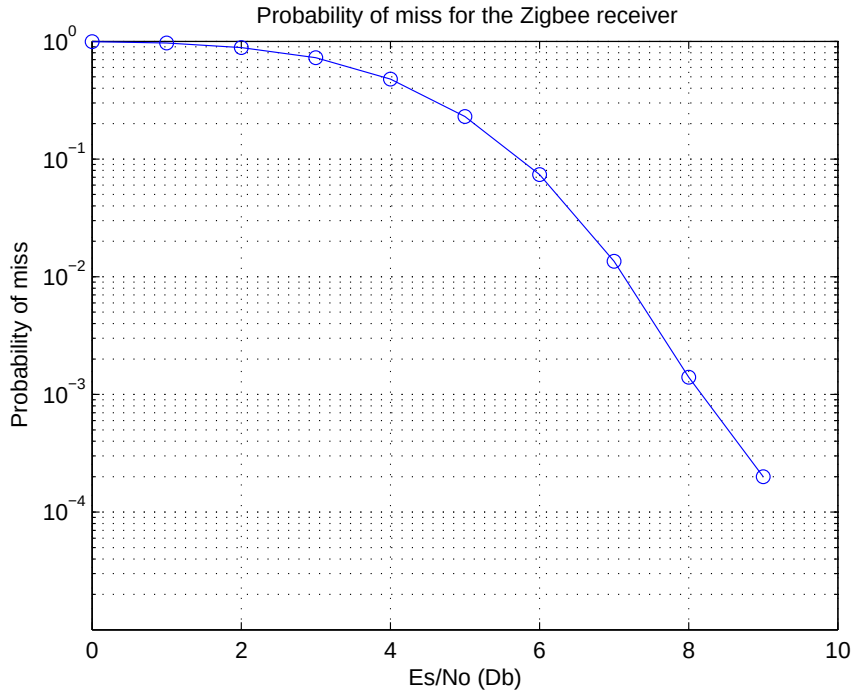


Figure 6.6: Probability of miss of the Zigbee receiver.

6.4 Hardware Resource Utilization

Table 6.2 gives the device utilization summary generated by the Xilinx tools. The entire circuit consists of the radio design and required external interface logic such as the Ethernet MAC wrapper, frame encoder and decoder. The radio design is implemented using a word-length of 14-bits. A 14-bit word-length is suffice to accomodate the peak amplitude of the baseband signal offered by the ADC.

Radio Implementation Using 7-bit Word Length:

Table 6.3 shows summary of device utlization for a design that includes the Zigbee receiver implemented with a fixed-point word length of 7-bits. The device utilization is lower as compared to the 14-bit fixed-point implementation; however, the resource savings are not

Table 6.2: Device utilization summary with 14-bit wide word length.

```
Device utilization summary:

     Number of BUFDSs                  1 out of 8           12%
     Number of BUFGs                   2 out of 32           6%
     Number of BUFRs                   1 out of 32           3%
     Number of DSP48Es                 10 out of 64         15%
     Number of GTP_DUALs               1 out of 8           12%
          Number of LOCed GTP_DUALs    1 out of 1          100%

     Number of TEMACs                  1 out of 2           50%
     Number of BlockRAM                9 out of 148          6%
     Number of bonded IOBs             3 out of 640          1%
          Number of LOCed IOBs         3 out of 3          100%
          Number of bonded IPADs       4 out of 50           8%
          Number of bonded OPADs       2 out of 32           6%

     Number of Slice Registers         1962 out of 69120     2%
          Number used as Flip Flops    1954
          Number used as Latches       8

     Number of Slice LUTs              3182 out of 69120     4%
          Number used as logic         2962 out of 69120     4%
          Number used as Memory        204 out of 17920      1%
```

very large. This can be attributed to the fact that a large part of the overall design consists of interface logic such as the Ethernet MAC wrapper and encoder-decoder.

The performance of both the implementations was compared using the rate of dropped frames (Table 6.4). The rate of dropped frames was evaluated using the radio link set-up described in Section 6.1.2. At the transmitter-end, the X-CTU was set to transmit the frames. At the receiver-end, a *tcpdump* packet filter was set at the Ethernet port of the host machine to capture the frames that were successfully demodulated and received by the receiver on the A-FPGA. The number of dropped frames was calculated by taking the difference between the number of sent and received frames. The ratio of number of frames dropped to the number of frames sent is referred here as the rate of dropped frames (mentioned in Table 6.4). It is apparent that, the rate of dropped frames increases with reduced word-length. In other words, the probability of miss of the receiver is heightened for the 7-bit implementation, deteriorating its performance. With the given decay in performance and limitation to achieve resource savings in the 7-bit implementation, the 14-bit word length implementation would be a more desirable option.

6.5 Comparison with Prior Zigbee Solutions

Table 6.5 presents a comparison of different attributes of Zigbee solution offered by this work with prior Zigbee transceiver products. The XBee module is described in Section 6.1.2. PICDEM Z is another Zigbee commercial product offered by Microchip [32]. PICDEM Z consists of an ASIC MRF24J40 that implements the IEEE 802.15.4 transceiver and a PIC18 microcontroller that implements rest of the Zigbee stack functionalities. The GNU Radio implementation refers to the realization of Zigbee radio on standard GNU Radio that was carried out at UCLA [33]. Enhanced GNU Radio refers to the Zigbee implementation for this work.

Table 6.3: Device utilization summary with 7-bit wide word length.

```
Device utilization summary:

     Number of BUFDSs                  1 out of 8           12%
     Number of BUFGs                   2 out of 32           6%
     Number of BUFRs                   1 out of 32           3%
     Number of GTP_DUALs               1 out of 8           12%
          Number of LOCed GTP_DUALs    1 out of 1          100%

     Number of TEMACs                  1 out of 2           50%
     Number of BlockRAM                9 out of 148          6%
     Number of bonded IOBs             3 out of 640          1%
          Number of LOCed IOBs         3 out of 3          100%
          Number of bonded IPADs       4 out of 50           8%
          Number of bonded OPADs       2 out of 32           6%

     Number of Slice Registers         1643 out of 69120     2%
          Number used as Flip Flops    1635
          Number used as Latches       8

     Number of Slice LUTs              3029 out of 69120     4%
          Number used as logic         2802 out of 69120     4%
          Number used as Memory        204 out of 17920      1%
```

Table 6.4: Probability of miss: 7-bit versus 14-bit implementation.

Word-length	Frames dropped
14-bit	1.31%
7-bit	3.06%

It is important to note that the enhanced GNU Radio and GNU Radio platforms allow implementations of a wide array of radio standards from low-end to high-end, high-rate radio designs, as opposed to fixed implementations offered by XBee and PICDEM Z. However, the high application flexibility in these platforms come at an expense of higher cost and power consumption. Being based on ASICs, XBee and PICDEM Z offer a high power efficiency, low cost as well as compact designs (Figure 1.1). The current versions of standard and enhanced GNU Radio platforms require different devices/boards for the RF front-end, host machine and the A-FPGA. Hence, these platforms have a high cost and bulky systems. Also, these recently developed platforms do not offer ease-of-use, since they need the user to have a higher degree of knowledge of the platform than that required for commercially available products (XBee and PICDEM Z).

Table 6.5: Comparison of Zigbee solutions.

	XBee	PICDEM Z	Enhanced GNU Radio	GNU Radio
Radio application flexibility	Low	Low	High	High
Power efficiency	High	High	Medium	Low
Compactness	High	Medium	Low	Low
Signal processing engine	ASIC	ASIC	FPGA	GPP
Ease-of-use	High	High	Low	Medium
Interface to host	RS-232	USB	GigE	Not required
Cost	$38	$270	$2150	$1400

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The enhanced GNU Radio is an effective SDR platform for development as well as deployment of radio designs. It provides a robust framework that offers flexibility and instant gratification in radio signal processing along with an option to easily include the use of hardware acceleration. This platform also maintains the open source strategy of GNU Radio, allowing for its widespread acceptance and application.

A successful radio implementation using Zigbee standard in this work, provided an insight to radio-development methodologies of the enhanced GNU Radio. Zigbee standard uses OQPSK modulation along with DS-SS technique. This allows it to have very simple and efficient receiver design. This is a very important factor that makes the Zigbee standard a good candidate for a proof-of-concept design. In this work, only the PHY-layer and a few necessary MAC layer features of the standard were deployed. These were sufficient to gauge the effectiveness of enhanced GNU Radio platform. However, deploying the MAC and transport layer is necessary for reliability of a Zigbee link. It is apparent from the acquisition

performance in Chapter 6, that Zigbee PHY-layer design has a fairly high probability of miss, which makes the role of transport layer significant in making a Zigbee link reliable and robust.

Another important conclusion that can be drawn is that the enhanced GNU Radio platform is not efficient in low-speed radio design scenarios. The GPP-based standard GNU Radio can easily handle these slow and small radio designs. Thus, there is no real advantage of the hardware acceleration offered by enhanced GNU Radio for such designs. Also, the enhanced platform provides hardware acceleration at the expense of some software and hardware overheads. In terms of software, adding the switch increases latency of the packets. When it comes to hardware resources, there is always an overhead of logic required to implement the interface (such as Ethernet in the case of this thesis). Hence, the overall design of such radios does not prove to be efficient on the enhanced GNU Radio platform. However, high-speed, complex designs such as the MB-OFDM based UWB radio design discussed in Chapter 5, are expected to be highly efficient on this platform. Such designs would justify the hardware acceleration in the enhanced platform and perform better on it than the standard GNU Radio.

7.2 Future Work

There are several opportunities for future work. Implementing high-rate, complex radio design such as the MB-OFDM UWB would allow to test if the fundamental goal of enhanced GNU Radio was achieved. Present proof-of-concept work helps in verifying the correct functionality, flexibility and integrity of the enhanced platform. However, its real advantage due to hardware acceleration over the standard GNU Radio cannot be gauged. Hence, it is important to realize such a high-speed, high-complexity design on the enhanced GNU Radio.

Another significant advancement would be to fold in the Agile Hardware technology into the platform. Doing this would boost agility of the platform. The enhanced platform was

modified in a way that would allow developers to easily include the Agile Hardware. Also, it is set up in such a manner that adding hardware modules to the design is consistent with the way software modules are added in the standard GNU Radio. In this way, the ease-of-use of GNU Radio is maintained. It would also be interesting to include all the MAC and transport layer functionalities to the current Zigbee design and get a fully-functional Zigbee link working.

Bibliography

[1] J.H. Reed. *Software radio: A modern approach to radio engineering.* Prentice Hall communications engineering and emerging technologies series. Upper Saddle River, NJ: Prentice Hall., 2002.

[2] S. Srikanteswara, R.C. Palat, J.H. Reed, and P. Athanas. An overview of configurable computing machines for software radio handsets. *Communications magazine, IEEE*, 41(7):134–141, 2003.

[3] W.H.W. Tuttlebee. Software-defined radio: facets of a developing technology. *Personal Communications, IEEE*, 6(2):38–44, 1999.

[4] K. Amiri, Yang Sun, P. Murphy, C. Hunter, J.R. Cavallaro, and A. Sabharwal. Warp, a unified wireless network testbed for education and research. In *Microelectronic Systems Education, 2007. MSE '07. IEEE International Conference*, pages 53 –54, June 2007.

[5] Robert Max, Shereef Sayed, Carlos Aguayo, Rekha Menon, Karthik Channak, Chris Vander Valk, Craig Neely, Tom Tsou, Jay Mandeville, and Jeffrey H. Reed. Ossie: Open source sca for researchers. In *SDR Forum Technical Conference, 2004*, 2004.

[6] Eric Blossom. GNU Radio [online]. `http://www.gnuradio.org`, 2010.

[7] Incremental compilation resource center [online]. `http://www.altera.com/support/software/incremental/sof-qts-increment-comp.html`, 2010.

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

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

[10] Charles Irick. Enhancing Gnu Radio for Hardware Accelerated Radio Design. Master's thesis, Virginia Tech, 2010.

[11] Eric Blossom. Gnu radio: tools for exploring the radio frequency spectrum. *Linux J.*, 2004:4–, June 2004.

[12] Ettus Research LLC [online]. `http://www.ettus.com/products`, 2010.

[13] GNU Radio: USRP2 wiki [online]. `http://gnuradio.org/redmine/wiki/1/USRP2`, 2010.

[14] Josh Blum. GNU Radio companion [online]. `http://gnuradio.org/redmine/wiki/1/GNURadioCompanion`, 2010.

[15] Writing Python applications.[online]. `http://gnuradio.org/redmine/wiki/gnuradio/TutorialsWritePythonApplications`, 2009.

[16] How to write a signal processing block.[online]. `http://www.gnu.org/software/gnuradio/doc/howto-write-a-block.html`, 2006.

[17] Ultrawideband: High-speed, short-range technology with far-reaching effects. White paper, September 2004.

[18] IEEE Computer Society. *Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)*, 2006.

[19] Rogelio Reyna Garcia. Understanding the Zigbee stack.

[20] Zigbee alliance.[online]. `http://www.zigbee.org`, 2010.

[21] J. Suris, P. Athanas, and C. Patterson. An efficient run-time router for connecting modules in FPGAs. In *Proceedings of the 2008 International Conference on Field Programmable Logic and Applications, FPL 2008, Heidelberg, Germany*, September 2008.

[22] J. Suris, A. Recio, and P. Athanas. Rapid radio: A framework for human-assistedsignal classification and receiver implementation. In *Proceedings of the 2008 National Aerospace and Electronics Conference, NAECON 2008, Fairborn, Ohio*, December 2007.

[23] J. Suris, A. Recio, and P. Athanas. Enhancing the productivity of radio designers with rapidradio. In *2009 International Conference on ReConFigurable Computing and FPGAs, Cancun, Mexico*, December 2009.

[24] Welcome to swig.[online]. `http://www.swig.org`, 2008.

[25] Chen Zhang. An ECA-Based Zigbee Receiver. Master's thesis, Virginia Tech, 2008.

[26] A. Di Stefano, G. Fiscelli, and C.G. Giaconia. An FPGA-Based Software Defined Radio Platform for the 2.4GHz ISM Band. pages 73 –76, 2006.

[27] Stefan Knauth. Implementation of an IEEE 802.15.4 Transceiver with a Software-defined Radio setup. 2008.

[28] IEEE 802.15. IEEE 802.15 WPAN Task Group 3 (TG3) [online]. `http://www.ieee802.org/15/pub/TG3.html`, November 2010.

[29] W. Pam Siriwongpairat and K. J. Ray Liu. *Ultra-Wideband Communications Systems-Multiband OFDM Approach*. Wiley Series in Telecommunications and Signal Processing, 2008.

[30] Steven K. Jones. The Evolution of Modern UWB Technology: A spectrum management perspective, May 2005.

[31] Ultra-Wideband [online]. `http://en.wikipedia.org/wiki/Ultra-wideband`, 2010.

[32] PICDEM Z Demonstration Kit.[online]. `http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en021925&part=DM163027-2`, 2010.

[33] UCLA Zigbee PHY. [online]. `https://www.cgran.org/wiki/UCLAZigBee`, 2010.

Appendix A

Source Code: Top-level

The source code for the HDL implementation of the Zigbee receiver is included below.

Top Level of the receiver-

```
//////////////////////////////////////////////////////////////////////
//   File name:  top.v
//   Author:  Mrudula Karve
//   Department: CCM Lab, Virginia Tech
//   Email: mrudulak@vt.edu
//   Date: August 2010
//   Description:  Top-level file of the design. It consists of the
//   Ethernet MAC wrapper, CDC FIFOs for flow control and an instance
//   of the Zigbee demodulator.
//   Clock rates for Ethernet and radio = 125 MHz.
//   Assumptions:
//////////////////////////////////////////////////////////////////////
`timescale 1 ps / 1 ps
module top
(
        // SGMII Interface - EMAC0
        TXP_0,
        TXN_0,
        RXP_0,
        RXN_0,
        // SGMII MGT Clock buffer inputs
        MGTCLK_N,
        MGTCLK_P,
        // reset for ethernet phy
        PHY_RESET_0,
        // GTP link status
        GTP_READY,
```

```
        // Asynchronous Reset
        RESET,
);

//------------------------------------------------------------------------
// Port Declarations
//------------------------------------------------------------------------
        // SGMII Interface - EMAC0
        output          TXP_0;
        output          TXN_0;
        input           RXP_0;
        input           RXN_0;
        // SGMII MGT Clock buffer inputs
        input           MGTCLK_N;
        input           MGTCLK_P;
        // reset for ethernet phy
        output          PHY_RESET_0;
        // GTP link status
        output          GTP_READY;
        // Asynchronous Reset
        input           RESET;

//------------------------------------------------------------------------
// Wire and Reg Declarations
//------------------------------------------------------------------------
        // Global asynchronous reset
        wire            reset_i;
        // Local Link Interface Clocking Signal - EMAC0
        wire            usr_clk_eth;
        // Local Link transmitter connections - EMAC0
        wire      [7:0] tx_ll_data_0_i;
        wire            tx_ll_sof_n_0_i;
        wire            tx_ll_eof_n_0_i;
        wire            tx_ll_src_rdy_n_0_i;
        wire            tx_ll_dst_rdy_n_0_i;
        // address swap receiver connections - EMAC0
        wire      [7:0] rx_ll_data_0_i;
        wire            rx_ll_sof_n_0_i;
        wire            rx_ll_eof_n_0_i;
        wire            rx_ll_src_rdy_n_0_i;
        wire            rx_ll_dst_rdy_n_0_i;
        // create a synchronous reset in the local link clock domain
        reg       [5:0] ll_pre_reset_0_i;
        reg             ll_reset_0_i;
        // synthesis attribute ASYNC_REG of tx_pre_reset_0_i is "TRUE";
        // Reset signals from the transceiver
        wire            resetdone_0_i;
        // EMAC0 Clocking signals
        // Transceiver output clock (REFCLKOUT at 125MHz)
        wire            clk125_o;
        // Input 125MHz differential clock for transceiver
        wire            ref_clk;
        // 1.25/12.5/125MHz clock signals for tri-speed SGMII
        wire            client_clk_0_o;
        wire            client_clk_0;
        // GT reset signal
        wire gtreset;
        reg  [3:0] reset_r;
```

```
        // synthesis attribute ASYNC_REG of reset_r is "TRUE";
//-----------------------------------------------------------------------------
// Main Body of Code
//-----------------------------------------------------------------------------
        assign PHY_RESET_0 = ~reset_i;
        IBUF reset_ibuf(.I(RESET), .O(reset_i));

        // Generate the clock input to the GTP
        // ref_clk can be shared between multiple MAC instances.
        IBUFDS clkingen (       .I(MGTCLK_P),
                                .IB(MGTCLK_N),
                                .O(ref_clk));

        // 125MHz from transceiver is routed through a BUFG and
        // input to the MAC wrappers.
        // This clock can be shared between multiple MAC instances.
        BUFG bufg_clk125 (      .I(clk125_o),
                                .O(usr_clk_eth));

        wire radio_clk;
        assign radio_clk = usr_clk_eth;

        // 1.25/12.5/125MHz clock from the MAC is routed through a BUFG and
        // input to the MAC wrappers to clock the client interface.
        BUFG bufg_client_0 (.I(client_clk_0_o), .O(client_clk_0));

//------------------------------------------------------------------
//-- RocketIO PMA reset circuitry
//------------------------------------------------------------------
        always@(posedge reset_i, posedge usr_clk_eth) begin
                if (reset_i == 1'b1)
                        reset_r <= 4'b1111;
                else
                        reset_r <= {reset_r[2:0], reset_i};
        end
        assign gtreset = reset_r[3];
//-----------------------------------------------------------------------
// Instantiate the EMAC Wrapper with LL FIFO
// (v5_emac_v1_6_locallink.v)
//-----------------------------------------------------------------------
        v5_emac_v1_6_locallink v5_emac_ll
        (
        // EMAC0 Clocking
        // 125MHz clock output from transceiver
        .CLK125_OUT                         (clk125_o),
        // 125MHz clock input from BUFG
        .CLK125                             (usr_clk_eth),
        // Tri-speed clock output from EMAC0
        .CLIENT_CLK_OUT_0                   (client_clk_0_o),
        // EMAC0 Tri-speed clock input from BUFG
        .CLIENT_CLK_0                       (client_clk_0),

        // Local link Receiver Interface - EMAC0
        .RX_LL_CLOCK_0                      (usr_clk_eth),
        .RX_LL_RESET_0                      (ll_reset_0_i),
        .RX_LL_DATA_0                       (rx_ll_data_0_i),
        .RX_LL_SOF_N_0                      (rx_ll_sof_n_0_i),
        .RX_LL_EOF_N_0                      (rx_ll_eof_n_0_i),
```

```
.RX_LL_SRC_RDY_N_0                  (rx_ll_src_rdy_n_0_i),
.RX_LL_DST_RDY_N_0                  (rx_ll_dst_rdy_n_0_i),
//    .RX_LL_DST_RDY_N_0                  (1'b0),
.RX_LL_FIFO_STATUS_0                (),

// Unused Receiver signals - EMAC0
.EMAC0CLIENTRXDVLD                  (),
.EMAC0CLIENTRXFRAMEDROP             (),
.EMAC0CLIENTRXSTATS                 (),
.EMAC0CLIENTRXSTATSVLD              (),
.EMAC0CLIENTRXSTATSBYTEVLD          (),

// Local link Transmitter Interface - EMAC0
.TX_LL_CLOCK_0                      (usr_clk_eth),
.TX_LL_RESET_0                      (ll_reset_0_i),
.TX_LL_DATA_0                       (tx_ll_data_0_i),
.TX_LL_SOF_N_0                      (tx_ll_sof_n_0_i),
.TX_LL_EOF_N_0                      (tx_ll_eof_n_0_i),
.TX_LL_SRC_RDY_N_0                  (tx_ll_src_rdy_n_0_i),
.TX_LL_DST_RDY_N_0                  (tx_ll_dst_rdy_n_0_i),

// Unused Transmitter signals - EMAC0
.CLIENTEMAC0TXIFGDELAY              (8'h00),
.EMAC0CLIENTTXSTATS                 (),
.EMAC0CLIENTTXSTATSVLD              (),
.EMAC0CLIENTTXSTATSBYTEVLD          (),

// MAC Control Interface - EMAC0
.CLIENTEMAC0PAUSEREQ                (1'b0),
.CLIENTEMAC0PAUSEVAL                (16'h0000),

//EMAC-MGT link status
.EMAC0CLIENTSYNCACQSTATUS           (GTP_READY),
.EMAC0ANINTERRUPT                   (),

// SGMII Interface - EMAC0
.TXP_0                              (TXP_0),
.TXN_0                              (TXN_0),
.RXP_0                              (RXP_0),
.RXN_0                              (RXN_0),
.PHYAD_0                            (5'b00010),
.RESETDONE_0                        (resetdone_0_i),

// unused transceiver
.TXN_1_UNUSED                       (),
.TXP_1_UNUSED                       (),
.RXN_1_UNUSED                       (1'b1),
.RXP_1_UNUSED                       (1'b0),

// SGMII MGT Clock buffer inputs
.CLK_DS                             (ref_clk),
.GTRESET                            (gtreset),

// Asynchronous Reset Input
.RESET                              (reset_i)
);

wire [7:0] packet_data;
```

```
wire [3:0] decode_flags;
wire       eth_valid;
wire       eth_frame_wr;
wire       fifo_full;
packet_decoder pdc (    .clk          (usr_clk_eth),
                        .rst          (reset_i),
                        .din          (rx_ll_data_0_i),
                        .src_rdy_n    (rx_ll_src_rdy_n_0_i),
                        .sof_n        (rx_ll_sof_n_0_i),
                        .eof_n        (rx_ll_eof_n_0_i),
                        .fifo_full    (1'b0),   //since read & write clocks are same , it won't get full
                        .dst_rdy_n    (rx_ll_dst_rdy_n_0_i),
                        .dout         (packet_data),
                        .decode_flags (decode_flags),
                        .valid        (eth_valid),
                        .frame_wr     (eth_frame_wr));

wire [15:0] fifo_dout;
wire fifo_rd_en;
fifo_cdc_e2r eth2rx (   .rst    (reset_i),
                        .wr_clk (usr_clk_eth),
                        .wr_en  (eth_valid),
                        .din    (packet_data),//[7:0] x 125Mhz
                        .rd_clk (radio_clk),
                        .rd_en  (1'b1),
                        .dout   (fifo_dout),
                        .full   (),
                        .empty  ());

// zigbee rx interface
// this module arranges the I and Q channel data in proper format
wire [15:0] i_data, q_data;
wire i_rdy, q_rdy, data_on;
zb_rx_interface ZRI0 (  .clk(radio_clk),
                                .nd(eth_valid),
                                .in(fifo_dout),
                                .rst(reset_i),
                                .out_i(i_data),
                                .out_q(q_data),
                                .rdy_i(i_rdy),
                                .rdy_q(q_rdy),
                                .rdy(data_on));

//////////////////////////////////////////
// Start of receiver

// Signal detector
wire sig_on;
sig_detect SD0 (        .clk(radio_clk),
                        .rst(reset_i),
                        .in(i_data),
                        .nd(data_on),
                        .sig_on(sig_on));

// pulse-shaping filters
wire [13:0] bb_I, bb_Q;
wire lpf_i_rdy, lpf_q_rdy;
assign bb_I = i_data;
```

```
assign bb_Q = q_data;
assign lpf_i_rdy = i_rdy;
assign lpf_q_rdy = q_rdy;

LPF filter_cos (        .clk(radio_clk),
                        .reset(reset_i),
                        .nd(i_rdy),
                        .data_in(i_data[13:0]),
                        .data_out(bb_I),
                        .rdy(lpf_i_rdy));

LPF filter_sin (        .clk(radio_clk),
                        .reset(reset_i),
                        .nd(q_rdy),
                        .data_in(q_data[13:0]),
                        .data_out(bb_Q),
                        .rdy(lpf_q_rdy));

// demodulator module
wire [15:0] data_out;
wire data_rdy;
wire eof;
zb_radio ZR0  ( .in_i(bb_I),
                .in_q(bb_Q),
                .nd_i(lpf_i_rdy),
                .nd_q(lpf_q_rdy),
                .clk(radio_clk),
                .rst(reset_i),
                .sig_on(sig_on),
                .out(data_out),
                .rdy(data_rdy),
                .eof(eof));

// End of receiver
//////////////////////////////////////////

// receiver fifo
wire [7:0] encoder_din;
wire encoder_rdy;
reg [15:0] rx_frames_in_fifo_cnt;
wire fifo_empty;

wire rx_fifo_empty;
fifo_rx2eth r2e(        .rst(reset_i),
                        .wr_clk(radio_clk),
                        .rd_clk(usr_clk_eth),
                        .din(data_out[7:0]), // Bus [7 : 0]
                        .wr_en(data_rdy),
                        .rd_en(encoder_rdy),
                        .dout(encoder_din), // Bus [7 : 0]
                        .full(),
                        .empty(rx_fifo_empty));

wire rx_enc_frame_rd;
wire rx_frame_wr;
assign rx_frame_wr = eof;

always @(posedge usr_clk_eth) begin
```

```
                if (reset_i) begin
                        rx_frames_in_fifo_cnt <= 16'b0;
                end else
                        case ({rx_frame_wr,rx_enc_frame_rd})
                        2'b00: rx_frames_in_fifo_cnt <= rx_frames_in_fifo_cnt; // when nothing happens
                        2'b10: rx_frames_in_fifo_cnt <= rx_frames_in_fifo_cnt + 16'b1;
                        2'b01: rx_frames_in_fifo_cnt <= rx_frames_in_fifo_cnt - 16'b1;
                        2'b11: rx_frames_in_fifo_cnt <= rx_frames_in_fifo_cnt;
                        endcase
        end

        assign fifo_empty = (rx_frames_in_fifo_cnt == 0);

        rx_packet_encoder rec ( .clk(usr_clk_eth),
                                .rst(reset_i),
                                .din(encoder_din),
                                .dst_rdy_n(tx_ll_dst_rdy_n_0_i),
                                .frames_waiting(~fifo_empty),
                                .rdy(encoder_rdy),
                                .dout(tx_ll_data_0_i),
                                .sof_n(tx_ll_sof_n_0_i),
                                .eof_n(tx_ll_eof_n_0_i),
                                .src_rdy_n(tx_ll_src_rdy_n_0_i),
                                .frame_rd(rx_enc_frame_rd));

        // Create synchronous reset in the transmitter clock domain.
        always @(posedge usr_clk_eth or posedge reset_i) begin
                if (reset_i) begin
                        ll_pre_reset_0_i <= 6'h3F;
                        ll_reset_0_i     <= 1'b1;
                end else begin
                        if (resetdone_0_i) begin
                                ll_pre_reset_0_i[0]   <= 1'b0;
                                ll_pre_reset_0_i[5:1] <= ll_pre_reset_0_i[4:0];
                                ll_reset_0_i          <= ll_pre_reset_0_i[5];
                        end
                end
        end
endmodule
```

Appendix B

Source Code: Zigbee Demodulator Blocks

HDL code for the Zigbee demodulator top-level-

```
//////////////////////////////////////////////////////////////////////
//   File name:  zb_radio.v
//   Author:  Mrudula Karve
//   Department: CCM Lab, Virginia Tech
//   Email: mrudulak@vt.edu
//   Date: April 12, 2010
//   Description:  Zigbee receiver wrapper- this code connects
//   all the logic blocks in the Zigbee demodulator.
//   Clock rate = 125 MHz
//   Assumptions:
//////////////////////////////////////////////////////////////////////

module zb_radio (
                      input signed [13:0] in_i,
                      input signed [13:0] in_q,
                      input nd_i,
                      input nd_q,
                      input clk,
                      input rst,
                      input sig_on,
                      output [15:0] out,
                      output rdy,
                      output eof);

      //////////////////////////////////////////////////////////////////////
      // logic to generate delta_theta signal
      wire signed [15:0] diff_phase;
```

```
        wire diff_phase_rdy;
        delta_phase DP0 (       .clk(clk),
                                .rst(rst),
                                .nd(nd_q),
                                .sig_on(sig_on), //DEBUG
                                .in_i({in_i[13],in_i[13],in_i}),
                                .in_q({in_q[13],in_q[13],in_q}),
                                .rdy(diff_phase_rdy),
                                .out(diff_phase));

        // this module samples the differential phase
        wire chip_phase_out;
        wire chip_phase_out_rdy;
        chip_phase_detect CPD0( .clk(clk),
                                        .rst(rst),
                                        .in(diff_phase),
                                        .nd(diff_phase_rdy),
                                        .sig_on(sig_on),
                                        .out(chip_phase_out),
                                        .rdy(chip_phase_out_rdy));

        // symbol detection
        symb_detect SD0 (       .clk(clk),
                                .rst(rst),
                                .in(chip_phase_out),
                                .nd(chip_phase_out_rdy),
                                .sig_on(sig_on),
                                .out(out),
                                .eof(eof),
                                .rdy(rdy));
        ////////////////////////////////////////////////////////////////////////
endmodule
```

HDL code for the Differential phase generator-

```
//////////////////////////////////////////////////////////////////////
//   File name:  diff_phase_gen.v
//   Author:  Mrudula Karve
//   Department: CCM Lab, Virginia Tech
//   Email: mrudulak@vt.edu
//   Date: August, 2010
//   Description:  This is the first module in the Zigbee demodulator.
//   It outputs the differential phase signal
//   of the complex input Zigbee signal
//   Clock rate = 125 MHz
//   Assumptions:
//////////////////////////////////////////////////////////////////////
module delta_phase (    input clk,
                        input rst,
                        input nd,
                        input sig_on, //DEBUG
                        input signed [15:0] in_i,
                        input signed [15:0] in_q,
                        output rdy,
                        output signed [15:0] out);
        // pipeline for i and q samples
```

```
reg signed [15:0] i1, i2;
reg signed [15:0] q1, q2;
reg pipeline_rdy;
always@(posedge clk) begin
        if (rst) begin
                i1 <= 16'b0;
                i2 <= 16'b0;
                q1 <= 16'b0;
                q2 <= 16'b0;
                pipeline_rdy <= 1'b0;
        end else begin
                if (nd) begin
                        i2 <= i1;
                        i1 <= in_i;
                        q2 <= q1;
                        q1 <= in_q;
                        pipeline_rdy <= 1'b1;
                end else
                        pipeline_rdy <= 1'b0;
        end //else
end //always

wire sign_q, sign_i;
assign sign_q = q2[15];
assign sign_i = i2[15];

// delta_theta = dI/dt.sgn(Q) - dQ/dt.sgn(I)
// diff1 = dI/dt.sgn(Q), diff2 = dQ/dt.sgn(I)
reg signed [15:0] diff1, diff2;
reg diff1_rdy, diff2_rdy;

always @ (posedge clk) begin
        if (rst) begin
                diff1 <= 16'b0;
                diff1_rdy <= 1'b0;
        end else begin
                if (pipeline_rdy) begin
                        if (sign_q) begin
                                diff1 <= (i2-i1) * -16'd1;
                        end else begin
                                diff1 <= (i2-i1);
                        end
                        diff1_rdy <= 1'b1;
                end else
                        diff1_rdy <= 1'b0;
        end //else
end //always

always @ (posedge clk) begin
        if (rst) begin
                diff2 <= 16'b0;
                diff2_rdy <= 1'b0;
        end else begin
                if (pipeline_rdy) begin
                        if (sign_i) begin
                                diff2 <= (q2-q1) * -16'd1;
                        end else begin
                                diff2 <= (q2-q1);
```

```
                        end
                        diff2_rdy <= 1'b1;
                end else
                        diff2_rdy <= 1'b0;
        end //else
    end //always

    assign rdy = diff1_rdy & diff2_rdy;
    assign out = diff1 - diff2;
endmodule
```

HDL code for the $\Delta\Theta$ samples generator-

```
//////////////////////////////////////////////////////////////////////
//   File name:  chip_phase.v
//   Author:  Mrudula Karve
//   Department: CCM Lab, Virginia Tech
//   Email: mrudulak@vt.edu
//   Date: August, 2010
//   Description:  This modules samples the delta_theta signal at every
//   half chip period
//   Clock rate = 125 MHz
//   Assumptions: Sample rate = 10 Msps, i.e. 10 samples
//   per chip OR 5 samples per symbol
//////////////////////////////////////////////////////////////////////
module chip_phase_detect(       input clk,
                                input rst,
                                input signed [15:0] in,
                                input nd,
                                input sig_on,
                                output reg out,
                                output reg rdy);

    // use zero crossing as a timing reference
    reg signed [15:0] reg1, reg2;
    reg sig_locked;
    always@ (posedge clk) begin
            if (rst) begin
                    reg1 <= 16'b0;
                    reg2 <= 16'b0;
                    sig_locked <= 1'b0;
            end else begin
                    if (sig_on) begin
                            if (nd) begin
                                    reg2 <= reg1;
                                    reg1 <= in;
                                    if (reg2[15] != reg1[15])       // zero crossing
                                            sig_locked <= 1'b1;
                            end //if
                    end else
                            sig_locked <= 1'b0;
            end //else
    end //always

    // no. of samples/half-chip period of the complex signal = 5
    // we need to sample on every 5th value
```

```
        reg [2:0] cnt;  //counter for no. of samples
        always @ (posedge clk) begin
                if (rst)
                        cnt <= 3'd2;    // pick the center samples (i.e. between 1...10, we pick 3 and 8)
                else begin
                        if (sig_locked) begin
                                if (nd) begin
                                        if (cnt >= 3'd4) cnt <= 3'b0;
                                        else cnt <= cnt+1;
                                end //if
                        end else
                                cnt <= 3'd2;
                end //else
        end //always

        // sample and assign binary values
        // -ve -> 0 and +ve -> 1
        always @ (posedge clk) begin
                if (rst) begin
                        out <= 1'b0;
                        rdy <= 1'b0;
                end else begin
                        if (nd & sig_locked) begin
                                if (cnt == 3'd4) begin
                                        out <= ~reg2[15];
                                        rdy <= 1'b1;
                                end else
                                        rdy <= 1'b0;
                        end else
                                rdy <= 1'b0;
                end //else
        end //always
endmodule
```

Appendix C

Source Code: Symbol Detector

HDL code for the Symbol detector-

```
//////////////////////////////////////////////////////////////////////
//   File name:  symbol_detect.v
//   Author:  Mrudula Karve
//   Department: CCM Lab, Virginia Tech
//   Email: mrudulak@vt.edu
//   Date: September, 2010
//   Description:  This module implements a demodulator state machine which
//   after detecting the Synchronization fields (preamble and SFD)
//   proceeds to the symbol detection (it reads the frame length field first,
/ /  and then the actual frame)
//   Clock rate = 125 MHz
//   Assumptions:
//////////////////////////////////////////////////////////////////////
module symb_detect(     input clk,
                        input rst,
                        input in,
                        input nd,
                        input sig_on,
                        output [15:0] out,
                        output eof,
                        output reg rdy);

        reg [31:0] chip_in_pipeline;
        reg chip_in_pipeline_rdy;
        always @ (posedge clk) begin
                if (rst) begin
                        chip_in_pipeline <= 32'b0;
                        chip_in_pipeline_rdy <= 1'b0;
                end else
                        if (nd) begin
                                chip_in_pipeline <= {chip_in_pipeline[30:0],in};
```

```
                        chip_in_pipeline_rdy <= 1'b1;
                end else
                        chip_in_pipeline_rdy <= 1'b0;
end //always

// since I & Q channels might interchange, we need two data modes
reg data_mode;  // preamble symb. '0' -> datamode 0;
                // preamble symb. '8' -> datamode 1

wire [31:0] chip_in;
assign chip_in = (data_mode) ? (~chip_in_pipeline) : (chip_in_pipeline);

// sym is the symbol no. with highest correlation
wire [4:0] sym;
highest_correlation HC0 (       .clk(clk),
                                .rst(rst),
                                .in(chip_in),
                                .nd(chip_in_pipeline_rdy),
                                .out(sym));

////////////////////////////////////////////////////////////////////////
// receiver state machine
////////////////////////////////////////////////////////////////////////
// receiver states
localparam IDLE = 3'b000;
localparam PREAMBLE_DETECTED = 3'b001;
localparam SFD_DETECTED = 3'b010;
localparam FRAME_READ = 3'b011;

// delay
reg [5:0] delay_cnt;
always @ (posedge clk) begin
        if (rst)
                delay_cnt <= 6'b0;
        else if (sig_on) begin
                if (chip_in_pipeline_rdy & (delay_cnt < 6'd40))
                        delay_cnt <= delay_cnt + 1'd1;
        end else
                delay_cnt <= 6'b0;
end //always

// counter to keep track of no. of chips
reg SOF; //start of frame
reg signed [6:0] adjust_cnt;
reg signed [6:0] chip_cnt;
always @ (posedge clk) begin
        if (rst | ~SOF)
                chip_cnt <= 7'b0;
        else if (chip_in_pipeline_rdy) begin
                if (chip_cnt >= 7'd31)
                        chip_cnt <= 7'd0 + adjust_cnt;
                else
                        chip_cnt <= chip_cnt + 1 + adjust_cnt;
        end //if
end //always

reg [2:0] receiver_state;
reg [3:0] preamble_cnt; // counter to count 8 preamble symbols
```

```
reg preamble_sym7_rcvd;
reg frame_length_LSB_rcvd;
reg [7:0] frame_length; //in octets
wire [7:0] frame_length_nibble; //in nibbles
reg [7:0] nibble_count;
reg sym_flag;
reg [3:0] nibble_out;
reg nibble_rdy;

assign frame_length_nibble = frame_length * 2;

always @ (posedge clk) begin
        if (rst | ~sig_on) begin
                receiver_state <= IDLE;
                SOF <= 1'b0;
                preamble_cnt <= 4'd0;
                data_mode <= 1'b0; //default
                adjust_cnt <= 7'b0;
                preamble_sym7_rcvd <= 1'b0;
                frame_length_LSB_rcvd <= 1'b0;
                frame_length <= 8'b0;
                nibble_count <= 8'b0;
                nibble_out <= 4'b0;
                nibble_rdy <= 1'b0;
                sym_flag <= 1'b0;
        end else begin
                if ((delay_cnt == 6'd40) & chip_in_pipeline_rdy) begin
                        case (receiver_state)
                                IDLE: begin
                                        if (SOF) begin
                                                if (preamble_cnt == 4'd8) begin
                                                        adjust_cnt <= 7'd0;
                                                        sym_flag <= 1'b0;
                                                        receiver_state <= PREAMBLE_DETECTED;
                                                end else if (sym == 5'd0 & chip_cnt == 7'd30) begin
                                                        preamble_cnt <= preamble_cnt + 1;
                                                        sym_flag <= 1'b1;
                                                        adjust_cnt <= 7'd1;
                                                end else if (sym == 5'd0 & chip_cnt == 7'd31 & ~sym_flag) begin
                                                        preamble_cnt <= preamble_cnt + 1;
                                                        sym_flag <= 1'b1;
                                                        adjust_cnt <= 7'd0;
                                                end else if (sym == 5'd0 & chip_cnt == 7'd0 & ~sym_flag) begin
                                                        preamble_cnt <= preamble_cnt + 1;
                                                        adjust_cnt <= -7'd1;
                                                end else begin
                                                        adjust_cnt <= 7'd0;
                                                        sym_flag <= 1'b0;
                                                end
                                        end else begin
                                                if (sym == 5'd0) begin
                                                        SOF <= 1'b1;
                                                        data_mode <= 1'b0;
                                                        preamble_cnt <= 4'd1;
                                                end else if (sym == 5'd8) begin
                                                        SOF <= 1'b1;
                                                        data_mode <= 1'b1;
                                                        preamble_cnt <= 4'd1;
```

```
                end //if
        end
end
PREAMBLE_DETECTED: begin
        if (preamble_sym7_rcvd) begin
                if (sym == 5'd10 & chip_cnt == 7'd30) begin
                        receiver_state <= SFD_DETECTED;
                        sym_flag <= 1'b1;
                        adjust_cnt <= 7'd1;
                end else if (sym == 5'd10 & chip_cnt == 7'd31 & ~sym_flag) begin
                        receiver_state <= SFD_DETECTED;
                        sym_flag <= 1'b1;
                        adjust_cnt <= 7'd0;
                end else if (sym == 5'd10 & chip_cnt == 7'd0 & ~sym_flag) begin
                        receiver_state <= SFD_DETECTED;
                        adjust_cnt <= -7'd1;
                end else begin
                        adjust_cnt <= 7'd0;
                        sym_flag <= 1'b0;
                end
        end else begin
                if (sym == 5'd7 & chip_cnt == 7'd30) begin
                        preamble_sym7_rcvd <= 1'b1;
                        sym_flag <= 1'b1;
                        adjust_cnt <= 7'd1;
                end else if (sym == 5'd7 & chip_cnt == 7'd31 & ~sym_flag) begin
                        preamble_sym7_rcvd <= 1'b1;
                        sym_flag <= 1'b1;
                        adjust_cnt <= 7'd0;
                end else if (sym == 5'd7 & chip_cnt == 7'd0 & ~sym_flag) begin
                        preamble_sym7_rcvd <= 1'b1;
                        adjust_cnt <= -7'd1;
                end else begin
                        adjust_cnt <= 7'd0;
                        sym_flag <= 1'b0;
                end
        end //else
end
SFD_DETECTED: begin
        if (frame_length_LSB_rcvd) begin
                if (sym != 5'd31 & chip_cnt == 7'd30) begin
                        receiver_state <= FRAME_READ;
                        adjust_cnt <= 7'd1;
                        frame_length[7:4] <= sym[3:0];
                        sym_flag <= 1'b1;
                end else if (sym != 5'd31 & chip_cnt == 7'd31 & ~sym_flag) begin
                        receiver_state <= FRAME_READ;
                        adjust_cnt <= 7'd0;
                        frame_length[7:4] <= sym[3:0];
                        sym_flag <= 1'b1;
                end else if (sym != 5'd31 & chip_cnt == 7'd0 & ~sym_flag) begin
                        receiver_state <= FRAME_READ;
                        adjust_cnt <= -7'd1;
                        frame_length[7:4] <= sym[3:0];
                end else begin
                        adjust_cnt <= 7'd0;
                        sym_flag <= 1'b0;
                end
```

```
        end else begin
                if (sym != 5'd31 & chip_cnt == 7'd30) begin
                        frame_length_LSB_rcvd <= 1'b1;
                        adjust_cnt <= 7'd1;
                        frame_length[3:0] <= sym[3:0];
                        sym_flag <= 1'b1;
                end else if (sym != 5'd31 & chip_cnt == 7'd31 & ~sym_flag) begin
                        frame_length_LSB_rcvd <= 1'b1;
                        adjust_cnt <= 7'd0;
                        frame_length[3:0] <= sym[3:0];
                        sym_flag <= 1'b1;
                end else if (sym != 5'd31 & chip_cnt == 7'd0 & ~sym_flag) begin
                        frame_length_LSB_rcvd <= 1'b1;
                        adjust_cnt <= -7'd1;
                        frame_length[3:0] <= sym[3:0];
                end else begin
                        adjust_cnt <= 7'd0;
                        sym_flag <= 1'b0;
                end
        end //else
end
FRAME_READ: begin
        if (nibble_count <= frame_length_nibble) begin
                if (sym != 5'd31 & chip_cnt == 7'd30) begin
                        adjust_cnt <= 7'd1;
                        sym_flag <= 1'b1;
                        nibble_out <= sym[3:0];
                        nibble_rdy <= 1'b1;
                        nibble_count <= nibble_count + 8'd1;
                end else if (sym != 5'd31 & chip_cnt == 7'd31 & ~sym_flag) begin
                        adjust_cnt <= 7'd0;
                        sym_flag <= 1'b1;
                        nibble_out <= sym[3:0];
                        nibble_rdy <= 1'b1;
                        nibble_count <= nibble_count + 8'd1;
                end else if (sym != 5'd31 & chip_cnt == 7'd0 & ~sym_flag) begin
                        adjust_cnt <= -7'd1;
                        nibble_out <= sym[3:0];
                        nibble_rdy <= 1'b1;
                        nibble_count <= nibble_count + 8'd1;
                end else begin
                        adjust_cnt <= 7'd0;
                        nibble_rdy <= 1'b0;
                        sym_flag <= 1'b0;
                end
        end else begin
                receiver_state <= IDLE;
                nibble_rdy <= 1'b0;
                nibble_out <= 4'b0;
                SOF <= 1'b0;
                preamble_cnt <= 4'd0;
                data_mode <= 1'b0;
                adjust_cnt <= 7'b0;
                preamble_sym7_rcvd <= 1'b0;
                frame_length_LSB_rcvd <= 1'b0;
                frame_length <= 8'b0;
                nibble_count <= 8'b0;
                sym_flag <= 1'b0;
```

```
                                        end //else
                                end
                                default: begin
                                        receiver_state <= IDLE;
                                        SOF <= 1'b0;
                                end
                        endcase
                end //if
        end //else
end //always

reg nibble_rdy_delay;
always @ (posedge clk) begin
        if (rst)
                nibble_rdy_delay <= 1'b0;
        else
                nibble_rdy_delay <= nibble_rdy;
end //always

// n_rdy remains high only for 1 clock cycle
wire n_rdy;
assign n_rdy = nibble_rdy & ~nibble_rdy_delay;

// PHY payload extraction
// we remove all 802.15.4 MAC fields since it's out of our scope for now
wire frame_on;
assign frame_on = (receiver_state == FRAME_READ);

wire [7:0] phy_payload_nibbles;
assign phy_payload_nibbles = (frame_on) ? (frame_length_nibble - 26) : 8'b0;
wire [7:0] size;
assign size = phy_payload_nibbles/2;

reg h_l; // higher/lower nibble select
always @ (posedge clk) begin
        if (rst | ~frame_on) h_l <= 1'b0;
        else if (n_rdy) h_l <= ~h_l;
end //always

// generate eof pulse
wire frame_end;
assign frame_end = (frame_on & nibble_count == frame_length_nibble);
reg frame_end_delay;
always @ (posedge clk) begin
        if (rst) frame_end_delay <= 1'b0;
        else frame_end_delay <= frame_end;
end
assign eof = frame_end & ~frame_end_delay;

reg [7:0] byte_out;
always @ (posedge clk) begin
        if (rst) begin
                byte_out <= 8'b0;
                rdy <= 1'b0;
        end else begin
                if (n_rdy) begin
                        if (nibble_count == 8'd21) begin
                                byte_out <= 16'b0;
```

```
                                rdy <= 1'b1;
                        end else if (nibble_count == 8'd22) begin
                                byte_out <= {4'b0,size};
                                rdy <= 1'b1;
                        end else if ((nibble_count > 8'd22) & (nibble_count <= (frame_length_nibble - 8'd4))) begin
                                if (h_l) begin
                                        byte_out[7:4] <= nibble_out;
                                        rdy <= 1'b1;
                                end else
                                        byte_out[3:0] <= nibble_out;
                        end else
                                byte_out <= 8'b0;
                end else begin
                        //byte_out <= 8'b0;
                        rdy <= 1'b0;
                end
        end //else
    end //always

    assign out = {8'b0, byte_out};
endmodule
```

Appendix D

Source Code: Symbol Detector Modules

This chapter includes all the logic modules used in the symbol detector.

HDL code for correlator-

```
//////////////////////////////////////////////////////////////////////
//   File name:  correlator.v
//   Author:  Mrudula Karve
//   Department: CCM Lab, Virginia Tech
//   Email: mrudulak@vt.edu
//   Date: September, 2010
//   Description:  This module finds the cross-correlation between
//   two 32-bit sequences
//   Clock rate = 125 MHz
//   Assumptions:
//////////////////////////////////////////////////////////////////////
module corr (   input clk,
                input rst,
                input nd,
                input [31:0] in1,
                input [31:0] in2,
                output [5:0] out);

        wire [31:0] xnor_out;
        assign xnor_out = in1 ~^ in2; //bitwise xnor

        assign out =    xnor_out[31] + xnor_out[30] +  xnor_out[29] +  xnor_out[28] +  xnor_out[27] +  xnor_out[26] +  xnor_out[25] +  xnor_out[24] +
                        xnor_out[23] +  xnor_out[22] +  xnor_out[21] +  xnor_out[20] +  xnor_out[19] +  xnor_out[18] +  xnor_out[17] +  xnor_out[16] +
```

```
                    xnor_out[15] +  xnor_out[14] +  xnor_out[13] +  xnor_out[12] +  xnor_out[11] +  xnor_out[10] +  xnor_out[9] +  xnor_out[8] +
                    xnor_out[7] +  xnor_out[6] +  xnor_out[5] +  xnor_out[4] +  xnor_out[3] +  xnor_out[2] +  xnor_out[1] +  xnor_out[0] ;

endmodule
```

HDL code to pick a symbol having highest correlation-

```
//////////////////////////////////////////////////////////////////////
//   File name:  highest_corr.v
//   Author:  Mrudula Karve
//   Department: CCM Lab, Virginia Tech
//   Email: mrudulak@vt.edu
//   Date: September, 2010
//   Description:  This module finds the correlation with each
//   symbol and outputs the symbol with highest correlation value
//   Clock rate = 125 MHz
//   Assumptions:
//////////////////////////////////////////////////////////////////////
module highest_correlation (    input clk,
                                input rst,
                                input [31:0] in, //incoming chips
                                input nd,
                                output [4:0] out);

        // spreading codes for differetial phase signal
        // reference: An FPGA-Based Software Defined Radio Platform for the 2.4GHz ISM Band
        // (authors- Antonio Di Stefano, Giuseppe Fiscelli, Costantino G. Giaconia)
        wire [31:0] sym0, sym1, sym2, sym3, sym4, sym5, sym6, sym7, sym8, sym9, sym10, sym11, sym12, sym13, sym14, sym15;
        assign sym0 = 32'b00111111000100001010001100100110;
        assign sym1 = 32'b01100011111100010000101000110010;
        assign sym2 = 32'b00100110001111110001000010100011;
        assign sym3 = 32'b00110010011000111111000100001010;
        assign sym4 = 32'b10100011001001100011111100010000;
        assign sym5 = 32'b00001010001100100110001111110001;
        assign sym6 = 32'b00010000101000110010011000111111;
        assign sym7 = 32'b11110001000010100011001001100011;
        assign sym8 = 32'b11000000111011110101110011011001;
        assign sym9 = 32'b10011100000011101111010111001101;
        assign sym10 = 32'b11011001110000001110111101011100;
        assign sym11 = 32'b11001101100111000000111011110101;
        assign sym12 = 32'b01011100110110011100000011101111;
        assign sym13 = 32'b11110101110011011001110000001110;
        assign sym14 = 32'b11101111010111001101100111000000;
        assign sym15 = 32'b00001110111101011100110110011100;

        // correlations
        wire [5:0] corr_sym0;
        corr C0 (       .clk(clk),
                        .rst(rst),
                        .nd(nd),
                        .in1(in),
                        .in2(sym0),
                        .out(corr_sym0));

        wire [5:0] corr_sym1;
        corr C1 (       .clk(clk),
```

```
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym1),
                .out(corr_sym1));

wire [5:0] corr_sym2;
corr C2 (       .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym2),
                .out(corr_sym2));

wire [5:0] corr_sym3;
corr C3 (       .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym3),
                .out(corr_sym3));

wire [5:0] corr_sym4;
corr C4 (       .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym4),
                .out(corr_sym4));

wire [5:0] corr_sym5;
corr C5 (       .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym5),
                .out(corr_sym5));

wire [5:0] corr_sym6;
corr C6 (       .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym6),
                .out(corr_sym6));

wire [5:0] corr_sym7;
corr C7 (       .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym7),
                .out(corr_sym7));

wire [5:0] corr_sym8;
corr C8 (       .clk(clk),
                .rst(rst),
                .nd(nd),
```

```
                .in1(in),
                .in2(sym8),
                .out(corr_sym8));

wire [5:0] corr_sym9;
corr C9 (       .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym9),
                .out(corr_sym9));

wire [5:0] corr_sym10;
corr C10 (      .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym10),
                .out(corr_sym10));

wire [5:0] corr_sym11;
corr C11 (      .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym11),
                .out(corr_sym11));

wire [5:0] corr_sym12;
corr C12 (      .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym12),
                .out(corr_sym12));

wire [5:0] corr_sym13;
corr C13 (      .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym13),
                .out(corr_sym13));

wire [5:0] corr_sym14;
corr C14 (      .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym14),
                .out(corr_sym14));

wire [5:0] corr_sym15;
corr C15 (      .clk(clk),
                .rst(rst),
                .nd(nd),
                .in1(in),
                .in2(sym15),
```

```
                    .out(corr_sym15));

        localparam CORRELATION_THRESHOLD = 6'd24;

        // pick the highest correlation value
        assign out =    (corr_sym0 > CORRELATION_THRESHOLD) ? 5'b00000 :
                        (corr_sym1 > CORRELATION_THRESHOLD) ? 5'b00001 :
                        (corr_sym2 > CORRELATION_THRESHOLD) ? 5'b00010 :
                        (corr_sym3 > CORRELATION_THRESHOLD) ? 5'b00011 :
                        (corr_sym4 > CORRELATION_THRESHOLD) ? 5'b00100:
                        (corr_sym5 > CORRELATION_THRESHOLD) ? 5'b00101:
                        (corr_sym6 > CORRELATION_THRESHOLD) ? 5'b00110:
                        (corr_sym7 > CORRELATION_THRESHOLD) ? 5'b00111:
                        (corr_sym8 > CORRELATION_THRESHOLD) ? 5'b01000:
                        (corr_sym9 > CORRELATION_THRESHOLD) ? 5'b01001:
                        (corr_sym10 > CORRELATION_THRESHOLD) ? 5'b01010:
                        (corr_sym11 > CORRELATION_THRESHOLD) ? 5'b01011:
                        (corr_sym12 > CORRELATION_THRESHOLD) ? 5'b01100:
                        (corr_sym13 > CORRELATION_THRESHOLD) ? 5'b01101:
                        (corr_sym14 > CORRELATION_THRESHOLD) ? 5'b01110 :
                        (corr_sym15 > CORRELATION_THRESHOLD) ? 5'b01111: 5'b11111;
endmodule
```