CODIPHY- Composing On-Demand Intelligent PHYsical Layers

by

Aveek Dutta

M.S., University of Colorado Boulder, 2008

B.Tech., University of Kalyani, India, 2002

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer and Energy Engineering

2013

This thesis entitled:
CODIPHY- Composing On-Demand Intelligent PHYsical Layers
written by Aveek Dutta
has been approved for the Department of Electrical, Computer and Energy Engineering

Prof. Dirk Grunwald

Prof. Douglas Sicker

Prof. Tim Brown

Prof. Fabio Somenzi

Dr. Joeseph Mitola III

Date ____________________

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Dutta, Aveek (Ph.D., Electrical Engineering)

CODIPHY- Composing On-Demand Intelligent PHYsical Layers

Thesis directed by Prof. Dirk Grunwald

CODIPHY or Composing On-Demand Intelligent Physical Layers aims to solve two fundamental problems in practical cognitive radio networks: Collaboration between two radio physical layers (PHY) with varying capabilities to agree on a common communication protocol, using an ontology based description of the internal structure of the radio subsystems and secondly, provide a method to compose a functioning radio pipeline from a set of pre-compiled components, using the high-level representation provided by the ontology, to target heterogeneous platforms. CODIPHY isolates the various domains of radio engineering, but still allows sharing of domain knowledge to achieve the common goal of radio adaptation.

CODIPHY goes beyond the concept of *"knobs"* in cognitive radios, to decompose the radio pipeline and then build it back again to implement a different wireless protocol. To automate this process through collaborative learning is the goal of this thesis. Instead of solving the generic problem of collaboration for all waveforms and protocols, we focus on the most common family of waveform used in modern wireless systems and cognitive radio networks, Orthogonal Frequency Division Multiplexing (OFDM), and validate the methodology of CODIPHY by prototype implementations on heterogeneous radio platforms using HDLs and high level programming languages.

CODIPHY is the culmination of experiences gathered from radio prototyping for cognitive radio networks and is greatly influenced by MAC-PHY crosslayer research, where the radio is treated as a mutable entity rather than fixed in function. In this thesis, we present the steps required to realize the concept of CODIPHY, which range from the implementation of cognitive radio prototypes on FPGA to its application in design and evaluation of novel crosslayer protocols. These steps provide valuable insights to the requirements and formulation of CODIPHY. Therefore, CODIPHY is a multi-disciplinary effort that facilitates knowledge sharing among disparate areas of research.

Dedication

To Dola and Adri

Acknowledgements

First of all, I would like to thank my PhD advisor Prof. Dirk Grunwald for his support, advice and guidance throughout my graduate studies.

Would like to thank my research collaborator, Dola Saha for her invaluable contribution in developing and maintaining software drivers for the software defined radio that is one of the foundation pieces of my thesis. Also, like to thank her for collaborating on crosslayer research that has enabled me to propose new architectures for next generation physical layers.

Also, would like to thank all the members of my thesis committee, Prof. Douglas Sicker, Prof. Tim Brown, Prof. Fabio Somenzi and Dr. Joseph Mitola III for their insightful comments that has helped me enrich my research.

Thank you to Prof. Dipankar Raychaudhuri and Ivan Seskar of Rutgers University for their support and cooperation over the years.

And lastly, a big thank you to all of my family members, specially my parents for believing in me and my friends for their support and always standing beside me when required.

Contents

Tables

Figures

Chapter 1

Introduction

Radio Physical (PHY) layer is the interface to the transmission medium and is responsible for reliable communication link. The main components of the PHY layer are the radio front-end, the baseband processor and the interface to the higher layers along with a control framework. The baseband processor is responsible for the core signal processing tasks required to decode a packet. Modern high speed applications require high throughput PHY layers that contain complex Digital Signal Processing (DSP) functions to ensure reliability. The complexity of these DSP subsystems largely depend on the type of waveform supported by a particular PHY.

For a long time, the radio PHY has been shielded from the networking research community, primarily because those are typically built as fixed-function ASICs with very little insight into the internal operations of the underlying processing elements. The Atheros chipset based wireless network cards along with the MadWiFi driver [1] was one of the earliest examples of exposing PHY specific features to be controlled using software drivers. This new feature spawned an entire body of crosslayer research, which focused on optimizing the Medium Access Control (MAC) layer and the PHY layer jointly to improve network functions. This was certainly a new frontier in wireless research. Researchers now have access to certain low-level features like Received Signal Strength Indicator (RSSI), transmit power, data-rate algorithms, etc. [2, 3]. However, it was quickly realized that these functionalities were not enough and the need for a more flexible solution for research was established. GNURadio [4] was the first of its kind that allowed researchers to perform PHY layer processing using software that runs on general purpose processors. This platform further opened up the PHY layer to the researchers. Being a flexible software based processing, GNURadio offered unforeseen gains in network protocols, at least from experiments and testbeds. Since all the signal processing is performed in software, it is much slower compared to the PHY time budgets for common wireless protocols. Therefore, it is not meant to be a replacement of commercial products, but offered a lucrative alternative for research and experimentation.

In order to pursue PHY layer enabled research, a highly programmable PHY that is fast enough to meet protocol time budgets is required. Therefore, an experimental prototype is a primary requirement for the research community that can provide more practical experimental testbeds with realistic results. The need for flexible PHY is further intensified with the advent of Cognitive Radios.

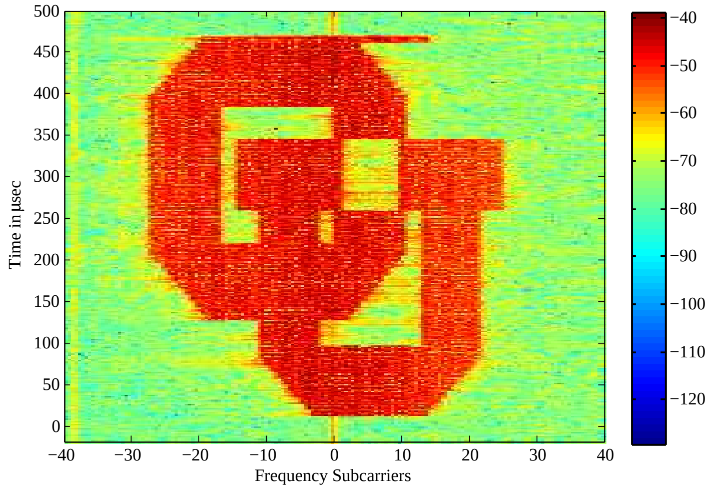


Figure 1.1: Spectrogram captured using a vector signal analyzer showing over-the-air transmission of multiple packets using non-contiguous OFDM from a SDCR transmitter. Packets are transmitted in a way to resemble the University of Colorado at Boulder logo, using non-contiguous subsets of subcarriers ranging between $[-27$ and $+27]$. The maximum possible signal bandwidth is $16.875MHz$.

Cognitive Radio (CR) is an emerging wireless networking technology that is generally characterized as “adapting to an environment” in order to improve network performance. Equally important, most cognitive radios are envisioned to use Dynamic Spectrum Access (DSA) to make use of fractured available spectrum. The use of multiple disjoint spectral bands greatly complicate the tasks the radio encounters — for example, now correlators, used to determine if a signal is in transmission, must look up across multiple bands. Figure 1.1 shows an example of a cognitive radio transmission over multiple sub-bands of varying width. Transmissions like this require unprecedented fine-grained programming at the radio PHY. This further expands the design space of radios that leads to the paradigm of Software Defined Cognitive Radios or SDCR. There are many ways to control and configure such an interface. A common method, which we explored for non-contiguous waveform [5], specifies an interface with different parameters that control specific aspects of the waveform. Systems such as Software Communication Architecture (SCA) provide mechanisms to communicate the waveform configuration of one radio system to another. Thus, it is evident that next generation radio PHY requires many “tuning knobs” to control the various facets of the waveform. Systems based on tuning knobs can implement many different radio waveforms, but they are fundamentally limited and cannot radically change the PHY layer, even on hardware platforms where that is technically possible. Recent research [6, 7, 8] has shown that the performance of modern wireless protocols can be greatly improved by jointly optimizing the MAC and the PHY layer or by changing the way the PHY layer is used. These cross-layer and flexible PHY techniques require adaptation of the PHY, to the changing requirements of the protocol. These new concepts in wireless networking is further facilitated by new waveform technologies like the Orthogonal Frequency Division Multiplexing (OFDM), which in turn influence the design and architecture of the flexible PHY layer.

OFDM [9] is a special type of Multi-Carrier Modulation (MCM), where the serial data is divided into multiple parallel bit streams and are modulated using closely spaced non-interfering frequencies called subcarriers. In OFDM, an Inverse Fast Fourier Transform (IFFT) is used to convert the data carrying sub-carriers to a time domain signal, which is upconverted to the desired carrier frequency. An inverse operation at the receiver using Fast Fourier Transform (FFT) reveal the frequency domain information. Apart from the simple waveform generation and reconstruction, OFDM provides significant advantages over single carrier transmissions like: immunity to multipath distortion, scalability and spectral separation, making it a superior choice for large family of wireless protocols [10]. In cognitive radio networking, each radio is expected to sense the environment for available spectrum and adapt quickly to it without interfering with the incumbent for that carrier frequency. The secondary system should be able to avoid the primary transmission while communicating within its own network in a *spectrum hole*. This kind of networks require sensing capability, and fast adaptation to new frequency band for both transmission and reception. We believe that OFDM is likely to be chosen as the communication substrate in CR networks due to its inherent capability of transmission and reception in variable bandwidth and in multiple chunks of subcarriers called subchannels. Any subcarrier set can be suppressed to form a Non-Contiguous OFDM (NC-OFDM) waveform, which can be utilized to transmit in a spectrum hole, avoiding the primary user. The use of FFT for OFDM also helps in sensing the spectrum, while other adaptation capabilities, like changing the number of subcarriers and subchannels makes OFDM the most appealing medium for communication in CR networks. Since most of the current wireless protocols, like 802.11a/g, 802.16, LTE, DVB, all use OFDM at the physical layer, we believe that OFDM is a likely choice for cognitive radio application as well. This motivates our research in new architectures for OFDM based radio PHY that will allow innovation in future deployments of CR networks.

In this thesis, we present various steps required to evolve an OFDM based prototype radio into a powerful and flexible research platform for next generation wireless networking research and also provide a methodology to synthesize such complex PHY layers on heterogeneous targets. The thesis also addresses the common design challenge of sharing domain specific knowledge among individuals of varied expertise and applies this concept of knowledge sharing in the broader context of CR networking. Architecting intelligent PHYs require multi-disciplinary knowledge. Whether it is the signal processing subsystems, like synchronization, equalization, etc. or the interfaces to the MAC layer, front-end and the control plane, the PHY should be viewed as a "system of systems". We discuss some of the system level challenges involved in architecting a flexible radio PHY and motivate the contributions of this thesis.

Reconfiguring the PHY: Co-existence of multiple standards along with the need for faster time to market, have led hardware manufacturers to lean towards reconfigurable hardware instead of ASICs. However, reconfigurability usually comes at the cost of higher power consumption (FPGAs) or slower processing (CPUs). These practical problems have led manufacturers to resort to either a custom made ASIC or a programmable processor based architecture often involving multicore processors and DSPs, or a heterogeneous architecture involving fast processors and reconfigurable gate arrays. In order to improve efficiency, wireless protocols have severe time constraints built into them. Therefore, while reconfiguring the radio is essential, meeting stringent time constraints, as required by the protocol, is of utmost importance. Also, with advances in silicon technology, the radio design should be able to adapt and optimize across multiple dimensions of speed, reconfigurability, power, etc. Thus, having a method to describe the radio functions independent of the underlying hardware platform is beneficial for casting the final radio hardware into different substrates.

Reconfiguration is also one of the top priorities for experimental prototypes and on-demand reconfiguration is particularly visible in cognitive radio networks. Reconfiguration can be performed at two stages. Basic reconfiguration specifies the operational parameters of the computation kernels, much like input arguments of a function, in programming parlance. This governs the way a particular kernel processes its inputs. Often, the targeted set of protocols have certain common baseband function but with different parameters. For example, the FFT engine for 802.11a/g requires a 64-pt FFT whereas that of 802.16e (WiMax) requires a variable FFT size of 128, 512, 1024 and 2048, supporting different data rates. Similarly, in a cognitive network there is co-existence of heterogeneous devices with different transmission capabilities and the PHY need to adapt its resources to support multiple wireless technologies. The second level of reconfiguration is at a kernel level. Either by replacing specific kernels in the radio pipeline to process a different type of waveform or by re-designing the structure of those kernels at real time from a high level specification. Reconfiguring the radio pipeline is a bigger challenge. While having common baseband kernels make it easier to reuse among various protocols, control over the dataflow between the kernels is also required. A tightly synchronized network-on-chip may provide some run-time reconfiguration of the data flow path but does not provide the full flexibility of software processing. Efficient partitioning the design into software and hardware components while maintaining sufficient flexibility is desired. These decisions should be based on some high level design constraints and not on individual designer's choice.

Another challenge in development of radio PHY is its constant evolution. While every generation of radio design provides support for new and emerging waveforms it also has to adapt with changing silicon technology and board layouts and organization. Migrating from one generation of radio to another has been a constant challenge in the radio design community. It would be beneficial to define a toolflow that would ensure migration - typically code regeneration, allow new constraints on the design, allow for new design partitioning and interconnection of new add-on cards. Often development of each of these individual elements occur in parallel and with little synchrony among disjoint research effort. The key design approach is *"abstraction"* of various layers involved in designing the PHY. Therefore, the architecture of the radio must allow the seamless interconnection of all the components without having hard synchronization logic between the various DSP subsystems of the PHY. In other words the processing pipeline should be insensitive to latency of individual subsystems. This provide flexibility in replacing, adjusting and re-designing the kernels as well as the system level components.

System level design: Any physical radio device cannot perform without its control layer. In wireless network protocols this layer is termed as the MAC layer. The MAC not only controls the flow of wireless packets it is also responsible for controlling and programming the radio to support cognitive behavior. To handle the complexity and ever-changing need for new cognitive support, the radio is controlled by a separate layer called the Radio Link Control (RLC) layer or the Hardware Abstraction Layer (HAL). The primary goal of the RLC is to program the radio, modify and adapt the radio based on the inputs from the MAC layer. From an architectural perspective, one has to determine the interface of the RLC to the MAC as well as to the PHY. Intuitively, having the MAC and RLC close to the hardware makes it faster to respond to a change in the radio environment, but it provides less flexibility in term of adding new features and waveforms to the radio. Software MACs are more flexible and easy to implement and the latency will be governed by the underline protocols for cognitive radio communication. In either case, a programmable gateway interface between the DSP kernels, peripheral cards (RF and ADC/DACs, clocks, etc) and the RLC while maintaining interoperability with a variety of heterogeneous MAC layers is a key design aspect that researchers need to consider.

Any practical radio goes beyond its baseband operation by integrating a plethora of add-on cards and I/O interfaces. The front-end is an important feature of a cognitive radio, as it has to support a wide range of tuning frequency coupled with adaptive filters and oscillators. Modern RF engines require far more control bandwidth than previous generation of fixed function RF pipeline. Multi-radio platforms make things worse as we have to multiplex and manage datastreams between multiple front-ends. High frequency digital interface is also required for wideband application leading to more control information and analog processing. Typically dataflow between the motherboard and the daughter cards is performed over serial interfaces like SPI that limit the bandwidth and data rate. Therefore, defining a standard interface between the baseboard and add-on daughter cards is required which is also independent of the baseband design. Another important add-on feature is the user I/O. Many choices ranging from PCI, Gigabit Ethernet to Thunderbolt can be made available on radio hardware. The radio architecture will have to be receptive of these various types of I/O sockets to ensure a longer lifetime in the market.

Heterogeneity of radio nodes: If all the radio nodes in a network have the same radio architecture, hardware and processing power, any low level PHY adaptation can be achieved by using a common firmware upgrade, usually in the form of a hardware image (FPGA bitfile) or as a software executable. However, we believe that future cognitive radio networks will comprise of heterogeneous radio nodes that coexist harmoniously using their cognitive capabilities. Although high performance purely software based radios already exist [11], they are large, complex and not practical for mobile applications. New systems will combine specialized software processors [12, 13], be based on reconfigurable computing [14] or on emerging "hybrid FPGA" systems that combine general purpose CPU's, SIMD processing and reconfigurable computing [15]. This plethora of platform choices point towards a more radical form of dynamic software radio reconfiguration where a system not previously designed for a particular waveform can be modified for a new waveform. That reconfiguration can occur either when designing a particular waveform (e.g., by a researcher) or during operation.

Heterogeneous computing platform, e.g., the ZYNC platform from Xilinx [15], are often used for deploying SDCRs, primarily to optimize across speed and power consumption. Wireless protocols are very time sensitive and the processing platform has to adhere to the time budget, which in turn dictates the resources required for a particular protocol: e.g., in 802.11a/g a node has to respond to an incoming communication within $16\mu s$. So, running 802.11a/g on a general purpose processor without some form of optimization is not practical as it will take too long to decode a packet. This is by far the most important factor that separates modern radio applications from other applications like radar or medical imaging which also use SDRs for speed and power savings. The inter-frame spacing can be different for other protocols and the radio designer will have to optimize the resources used to meet the time budget. The time budget for future protocols that will be used for cognitive radio networks, are not yet defined and so the design methodology has to be flexible enough to accommodate these on-demand changes. Also, heterogeneous platforms require different languages when it comes to implementation. While a FPGA based system will only accept HDLs, DSPs and Stream processors will require high level languages and compilers. Since optimization is best left to the domain experts, a high level representation of the dataflow between components is very useful for sharing information across these multiple domains of design. High level description of the SDCR PHY is difficult because it also has to be independent of the target platform.

Design and architecture of next generation SDCRs require knowledge and optimization from different fields of research e.g., DSP and digital communication, electronic design automation, compiler and programming languages, formal methods of system modeling and verification to mention a few. Radio design typically begins with the mathematical model of a waveform, from which a system model is derived with processes and kernels that are compiled to either electrical circuits, software programs or a combination of both. SDCRs are composed of many components interwoven into a complex system of systems and goes beyond the boundaries of a particular kernel or interconnection network. Therefore, it is necessary to abstract the kernel and system design from each other, so that domain specific optimizations can be applied at different stages of the radio design. System design involves multiple subsystems such as, the RF front-end, the ADCs and DACs, user I/O and the MAC layer interface. These, subsystems will have their own set of design constraints that needs to be handled at a system level and should be independent of the kernel level optimization. This will ensure interoperability of a vast number of subsystems that can be developed independent of the waveform or the target hardware used to implement the radio.

In order to address these challenges and provide a unified solution, mechanisms are needed to query the configuration of a system in a platform-independent manner and specify the configuration of a dynamically configurable platform. However, the configuration may involve the use of existing hardware blocks, new software or dynamically reconfigurable computing fabrics or all three methods combined. These configurations can either be specified “bottom up” using an imperative specification or “top down” using a more declarative specification. For example, in the domain of databases, early database techniques to find specific data values involved writing programs to search through a vendor-specific database. Those programs were then bound to that particular problem and that particular platform; this corresponds to the way software is currently developed for cognitive radios. Over time, those programs gave way to a relational algebra and a common specification language (SQL) that both simplifies database access and also allows the database system to improve or optimize the actual query.

A similar problem is faced in cognitive radios – how to specify what is needed, but not over-specify the solution to the point where the features or benefits of a specific platform can not be exploited. We believe that this requires a method for hierarchical description. For example, at a high level, a radio PHY design may want to specify a particular common power-detection mechanism or correlator for packet detection. However, if a system involved in the cognitive network does not "know" that specific design, the design specification can be "lowered" or refined to a more concrete form.

In this thesis, we use an ontology specification, a tool of the semantic web, to capture these hierarchical specifications. Ontologies are a method of knowledge representation that are useful in categorizing large amount of objects within a specified domain, in such a way that different agents can learn and understand the concepts in a domain and their relationships. Ontology is useful in defining taxonomies that organize domains like large collection of web pages and large organizations. The goal is two fold: a structured way of representing a hierarchy of components, their properties and the relationships between them and secondly, to be able to extract information about the domain using queries, such that an agent can learn about other agents and collaborate if needed.

In the domain of cognitive radio networks, agents are characterized by the heterogeneous radio nodes that are trying to collaborate with each other by sharing knowledge about the internal structure of their respective PHY layers, which is nothing but a hierarchy of interconnected components. The use of ontologies in cognitive radio has been studied by Kokar et al. [16, 17] and research [18] has shown that ontology can be effectively used by multiple radios to collaboratively decide on a particular access policy. However, the use of ontology to define the internal structure of a radio PHY hasn't been studied before. In this thesis, we compose the radio PHY on-demand from a high level specification that is represented using an ontology.

The multi-disciplinary nature of building SDCRs require a component based design approach. With appropriate domain knowledge, we are able to extract a set of components and aggregates that are repeated throughout the topology of the radio PHY. Components can vary from basic arithmetic elements like adders, subtractors and multipliers to aggregates of these like, complex multiplier, moving average and FFT blocks. SDCRs are essentially a composition of these basic elements and aggregates along with control signal to direct the flow of samples through the pipeline. The advantage of the aggregates is that they can be treated as *black boxes*, which are optimized for a particular application or even for a target hardware: e.g., a FFT unit for FPGAs will have a different set of optimization than that of the FFTW library that runs on a general purpose processor. Therefore, there is clearly a concept of domain specific optimization embedded in the radio design process. The goal is to bridge this gap as much as possible. Languages used for scientific computing like MATLAB, also employ similar design approach, where domain knowledge is shared across users using pre-defined functions with input and output parameters.

In addition, we use the ontology to describe hierarchical representations of equivalent functionality. For example, there are many possible representations of a correlator. At a high-level, we would like to simply specify that a correlator that meets specific properties is needed. However, a receiving system may not have a specific "correlator" object, and multiple instances of correlators may be needed to finally arrive at an implementable system. The ontology describe relationships between components – for example, either an adder or a fused-multiple-adder (FMA) "is a" functional unit and "is a" adder, and components can belong in many such relationships. In addition to specifying "is a" relationship, a rich web of relationships indicating alternate methods and options must be represented; so must the connections between components – these components and aggregates form the basis of the ontology and the relationship between their instances will define the dataflow. Using a structured representation like ontology, CR nodes can query and learn about the radio PHY without prior knowledge of the structure or the waveform it implements. Once the learning process is over, radios can effectively clone each other and agree on a common mode of communication.

For example, radio A can query radio B if it supports a particular modulation format. In the absence of such a modulation in radio B, it can obtain the description for that particular modulation and compose the internal structure (typically by modifying the I/Q mapping format) for the new modulation. Part of this problem can be solved by parametrizing the modulator. However, this approach is infeasible once the complexity of the system increases, such as in OFDM based protocols. Also, this leads to over-provisioning of resources as it is very hard to predict what new functions will be required in the future for harmonious co-existence of heterogeneous CRs. Using an ontology the radios can understand each other and compose newer subsystems using components from their own library or download from a server without involving the designer or the user. As the subsystems get complex and more application specific, this process can be elevated to a DSP kernel level reconfiguration from a structure level because at that point it gets too complex to formally express and reason on those complex internal structures. Not only does the interconnection between the subsystems need to be programmable, the subsystems themselves require changes to accommodate the superior performance obtained from the protocol. Sometimes these changes are minimal, involving addition or deletion of basic arithmetic elements, while in other cases the changes are major and call for building new components or aggregates altogether. Therefore, to support modern protocols, we need to be able to *clone* the radio PHY, rather than build fixed function pipelines that are brittle when exposed to such novel cross-layer implementations. Hence the term CODIPHY - Composing On-Demand Intelligent PHYsical Layers.

Using suitable back-end compilation, the ontology is translated to the preferred implementation language required by the target platform, thus allowing specific optimizations to be targeted during the implementation phase. For example, a radio expressed as an ontology will allow DSP engineers to describe the structure declaratively, while the radio engineer can use that high level relationship to compose the radio using pre-compiled components and aggregates. Automating this process of expressing the PHY and code generation for onward implementation is the goal of CODIPHY. Now machines can process the radio dataflow and use a rich library of building blocks to compose the radio when it is required to modify the physical structure or the dataflow between various components. CODIPHY is the methodology and toolflow that is able to achieve these in an efficient manner and make CR accessible to the broader community of wireless researchers.

Therefore, the contributions of this thesis are broken into four distinct stages of evolution of a radio PHY from a fixed function processing pipeline to a more flexible substrate suited for cognitive radio and crosslayer networking research. Followed by results from utilizing the flexible platform for wireless cross-layer and cognitive radio research. The final contribution is methodology for synthesizing complex PHYs for heterogeneous processing platforms. We enlist the detailed contribution of this thesis as follows:

- Design, implementation and performance evaluation of an OFDM based 802.11a/g PHY on FPGA with emphasis on the receiver algorithms and structures.
- Evolve the fixed function radio prototype into a SDCR. Specify the requirements of various "tuning knobs" and control parameters required to operate as a frequency agile cognitive radio and facilitate MAC-PHY crosslayer protocol design as well.
- Utilize the flexible PHY to design and implement *five* wireless crosslayer techniques for wireless and cognitive networks. Using theoretical concepts of the OFDM waveform, protocols are designed and evaluated by hardware implementation and experimental results.
- Identify the set of components and aggregates that are used to compose the baseband radio to support a family of OFDM waveforms. This includes Wi-Fi, WiMAX, LTE and any cognitive variant using NC-OFDM. This technique can be applied to other waveforms as well, but we choose OFDM as an example for this thesis for its wide applicability in cognitive radios.
- Create an ontology for the CR PHY. The ontology is designed as a hierarchical knowledge base of different representation levels (system, subsystem, specification and dataflow), which can be used by users of varied expertise, MAC and higher layers and other radio agents as well. By making design decisions at different stages of the radio hierarchy we can facilitate a component based design approach so that the radio designer can specify the exact parameters with limited domain knowledge. Once created, the ontology allows reasoning about the structure of the PHY and the dataflow between its components. This allows CR nodes to agree on not only the high level policies but also how to enforce those policies using proper waveforms at the PHY.
- Query the ontology at different levels and infer the hierarchical knowledge about the PHY structures and its capabilities. This allows knowledge sharing using a high level, declarative representation. Users of varied expertise can choose the granularity of design information.
- Synthesize functioning PHY from the high level design representation targeting FPGAs and general purpose processors. Examples of composing various OFDM subsystems from the ontology is also

presented.

In the realm of radio engineering, CODIPHY provides the answer to *"what to build"* rather than *"how to build"* by expressing the radio pipeline as an ontology. Instead of specifying the architecture of the individual components, CODIPHY represents a system, a level above that by specifying the properties and the hierarchy of the components that are required to build the radio. CODIPHY assumes that these components are made available by various domain experts and that the methodology will only assist in composing the radio from its components. This allows for multiple implementations of the same component targeted for different optimizations ensuring better performance across various dimensions of the radio implementation. Using CODIPHY we can express these qualitative attributes that will assist in generating the correct set of optimized components during the implementation phase.

Existing tools for model driven design for DSP systems, like Xilinx System Generator [19] and Ptolemy [20] are specific methods that govern how things are implemented. For example, System Generator generates code for FPGA based implementation while abstracting the designer from the intricacies of HDLs and Ptolemy generates valid schedules for simulating complex systems on multiprocessor architectures. These methods cannot be used for collaborative learning of CRs because neither of them have the data structure or the reasoning framework to be able to express a system at a functional level, which is the core idea of CODIPHY. Both the methods are useful for building fixed function systems which is detrimental to CRs, where the essence is adaptation through collaborative learning. In the context of this thesis, these modeling tools are considered as domain specific tools that can be an integral part of CODIPHY but are kept separated by using a high level relationship of components. Therefore, CODIPHY will provide the framework that makes radio more malleable and promotes domain specific optimizations while sharing domain knowledge between various research communities.

Thesis organization: The concept of CODIPHY is the result of experiences gathered from intersecting areas of research. Whether it is PHY layer prototyping, requiring deep understanding of the signal processing elements or its wide application in crosslayer protocol design and experimental validation, the need to *compose* radio physical layer has been felt at various aspects of wireless research. This thesis is a compilation of all the various steps and experiences that has led to the final realization of CODIPHY.

We start by addressing previous work in the area of radio prototyping, wireless crosslayer research and concepts of knowledge sharing in chapter 2. In chapter 3, we discuss the design and implementation of the various signal processing subsystems of an OFDM receiver. This *"Physical Layer"* is the foundation of this thesis. We extend the basic OFDM PHY to include capabilities for dynamic spectrum access and other cognitive radio processing in chapter 4 thus transforming it to *"Intelligent Physical Layers"*. We present an example implementation of SDCR, which exposes the tunable specifications that caters to the broader community of MAC-PHY crosslayer researchers. In order to show how the PHY is modified to implement crosslayer protocols we present examples of *"On-Demand Intelligent Physical Layers"* in chapter 5, along with design and evaluation of these applications. At a high level, CODIPHY is composed of three parts that combine to form the idea of *"Composing On-Demand Intelligent Physical Layers"*: Chapter 6, presents a hierarchical knowledge representation methodology using an ontology description that describes complex wireless PHY. Secondly, in chapter 7, we discus the process of retrieving design information by querying the ontology at different levels of granularity and show example of collaboration at the PHY and finally in chapter 8 present the steps required to generate synthesizable FPGA code along with executable C code targeting heterogeneous radio hardware. In chapter **??**, we compare CODIPHYto other competing techniques for synthesizing radio PHY and conclude the thesis in chapter 10.

Chapter 2

Related Work

Since CODIPHY focuses on convergence of disparate areas of research, it is important to acknowledge the various research efforts in these domains. We categorize this into three broad areas that are relevant to CODIPHY: 1) Methods of radio design and implementations, 2) MAC-PHY crosslayer protocol design, and 3) Cognition and collaboration at the physical layer. CODIPHY addresses the challenges faced in these domains and overcome the drawbacks to form a unified solution of designing complex PHY while allowing for domain specific improvements and research contributions.

Methods of radio design and implementation: The key aspect of architectures of SDR is having a generic framework which achieves the time budget of a wireless protocol within a portable form factor for mobile devices. Originally, digital radios were implemented using a combination of general purpose DSPs or fixed-function logic implemented as an ASIC or using an FPGA. Over time, specialized processor designs have evolved that are finely tuned for handling a set of wireless protocols. However, these processors are mainly suitable for $3G$ networks, and it's not clear if they are easily adaptable to $4G$ or other emerging radio standards. Alternative architectures, such as the PicoChip [21] processor combines general purpose processors with fixed-function logic designed to provide more efficient solutions to specific tasks (e.g., correlators for determining if a packet is being received). Radio design is driven by researchers from different communities: DSP and digital communication, Electronic Design Automation, Compiler and programming languages, formal modeling of systems. Each of these groups focus on a particular aspect of the radio but often fail to unify the solution into one system, which is the goal for a practical radio implementation.

Wireless protocol processing can be broadly grouped into four categories: (1) Software processing only on general purpose processors, (2) On-chip network based architecture, (3) Multiprocessor architecture and (4) Hybrid architecture - general purpose processors along with dedicated accelerators using reconfigurable gate arrays.

When wireless PHY processing is done entirely in software as in [4] and [22], although it aids in programming by using simple high level programming languages, they often fail to meet the protocol timing requirements for modern wireless protocols such as 802.16 and other cognitive radio protocols like 802.22 because of a combination of I/O throughput and post-processing using commercial CPUs. The SORA platform [11], is an implementation of 802.11 physical layer using general purpose CPUs and a radio control board (primarily used for buffering digital samples). Although most of the transceiver is implemented in software, the system isn't able to support NC-OFDM transmission and reception, which is the basic requirement for cognitive radios. The cache optimizations are specifically done to meet Wi-Fi requirements, and will not work for more complex systems.

Network on Chip (NoC) based processing [23, 24, 25] relies heavily on the performance of the routing algorithm and the efficiency of the common functional unit. Reprogramming such devices can only be done at compile time. Unless, the functional units are made multimode, supporting multiple protocols is a challenge using this form of architecture. Other solutions like the FAUST NoC chip [25] and the Magali NoC [26] provide optimized NoCs based on the GALS paradigm, that facilitates swapping of the kernels but that is also limited by the compile time. Also, these are fixed function radio components held together with inflexible logic interfaces, which is sub-optimal for SDRs.

Multiprocessor architectures are particularly effective for radio processing because it meets the protocol timing requirements in most of the cases. SODA [12] provides a multi-processor architecture using optimized SIMD operations for digital processing, but SODA does not address the requirements for CRs and it is not known if the processor could support non-contiguous OFDM or other 4G protocols. In [21], the authors propose a multiprocessor architecture using several hundred processors. Implementation of radio PHY using highly parallel processors is also shown in [27]. Researchers have also used embedded processor to implement a simple single carrier radio transceiver as in [28].

Processor based architectures are often augmented by dedicated hardware acceleration units for particular algorithms. [29] employs FPGA accelerators for DSP algorithms along with RISC processors, where application specific functions are mapped. These systems often combine dedicated hardware for correlations for signal detection. Software controlled hardware is another form of processing engine that uses software to control certain "knobs" in the hardware to perform multiple tasks. The WARP [30] and KUAR [31] are two such platforms that are capable of certain cognitive radio transmission. Another design approach called partial reconfiguration of FPGA, has been made possible by Xilinx [32], and has been shown to provide run-time reconfigurability for dynamic SDR systems [33] at the cost of high control overhead and resource consumption.

While most of the previous work focuses on architecture of the actual processing engine, very few focus on defining the **requirements** of a true cognitive radio. Therefore, instead of architecting just SDR, future research should be inclined towards the idea of a SDCR. It is important to envision how next generation wireless networks will behave, and the design of the underlying hardware needs to be such that the architecture is ready to embrace any adaptation required.

DSP and digital communication algorithms are usually represented by mathematical equations and joined together to form a processing flow. In order to implement the mathematical structure of a waveform, the equations are translated in the form of a signal flow graph where each **node** represents one or more DSP functions. The graph nodes are parametrized by control information to modify and synchronize the data stream. The data flow graph represents the baseband process of the radio and is mostly composed of forward **edges** with some backward edges for synchronization and iterative loops in the DSP algorithms. The granularity of the data flow model is of importance because that governs the degree of flexibility of the implemented radio. Synchronous data flow based modeling was first introduced for DSP systems in Ptolemy [20]. By analyzing the dataflow graph we can extract a firing schedule for various actors in the graph. Subsequent development of the Dataflow Interchange Format (DIF) [34] has allowed researchers to automate software synthesis from a graph based description of a system [35]. An implementation using DIF for DSP systems has been shown in an example of video coding in [36]. A more recent work in [37] has extended DIF to include topological patterns that are commonly seen in signal processing domain. Although the tool is not exhaustive it shows a that bigger systems can be designed using fundamental building blocks that repeats itself throughout the processing pipeline. This makes it generic enough to process a particular family of waveforms or applications. However, DIF is limited to auto-generating code and valid schedules for processor based systems which may not be the ideal platform for real-time radio applications. This data flow model has interesting analogies to the flow of software programs, particularly in the realm of stream computing and vector processing. Radio samples can be viewed as streams of floating point or integer numbers that undergo the same transformations which forms the input of another set of processes. This data flow is very typical in radio processing. Stream programming has been employed in modern graphics processors that are shown to be useful for DSP application as well [38]. StreamIt [39], is one such stream programming language that identifies data parallelism in the DSP algorithms and schedules processes to different processor cores to achieve fast computation DSP algorithms. StreamIt and its various peers are good choices for implementing radio functions. However, data-dependent computations like loops and iterative operations need to be addressed with care. Since radio processing has fixed time budgets, it limits the use of software processing unless severely optimized for a particular application, which may make it brittle and prone to failure when implementing other protocols. In order to facilitate re-targeting designs on to different substrates, radio processing logic is sometimes represented in a generic form using behavioral or functional descriptor. These functional description can then be used to generate the required code base for a particular platform of computing - FPGAs, Stream processors, DSPs or general purpose CPUs. The Rosetta [40, 41] functional language presents a method of representing DSP systems as purely **functional** programs as opposed to conforming to a particular programming language syntax. However, it is still an emerging language and the support for more complex systems like the ones used in SDCRs is not yet known. The wavefrom description language [42] is another example of a text based language to define computation elements used in DSP algorithms. The design flow does not have to be at that low level, instead it can be used to abstract certain low-level structures from the user and expose them as pre-compiled libraries. The granularity of this abstraction however needs to be determined so that there is a seamless integration of research efforts from multiple disciplines.

Another approach used to re-target programs is by extracting the data flow graph from one programming language and then parse and compile it to produce codes in another syntax. Conversion from stream programs to HDLs has been shown in [43], whereas authors in [44] discuss migrating from DSP to a FPGA platform. FPGAs can also be used as co-processor and the code generation from software binaries can be automated to reduce software bottlenecks. This technique has been applied to video decoding algorithms [45] to improve efficiency. Bluespec [46] is another tool that allows system modeling using functional descriptors and use handshaking mechanisms to interconnect subsystems. But it is limited to RTL implementations on FPGAs or ASICs only. It has been used to implement flexible hardware for wireless crosslayer designs [47]. Therefore, in each of these tool flow, representing a DSP process and its interconnection using a data flow graph is essential and individual compilers can be used to automate the code generation with the proper constraints and data structures.

In terms of hardware implementation, HDLs and their tool flow are the most common method. However, over the years there have been tools from the EDA community that facilitates the hardware design flow by hiding the intricacies of the HDLs from the designer. Tools, such as System Generator from Xilinx and AccelDSP from Matlab, uses a combination of Simulink and Matlab's native compiler to implement synthesizable HDLs. Since radio/DSP processes are flow based, it is easier to represent and understand them in schematic form rather than a code base. Complete design and evaluation platform based on library driven schematics greatly reduce the pain of writing and simulating codes written in HDLs. But, unless designed properly this type of tool flow suffers from hand-coded control and synchronization logic for the dataflow that is often brittle due to the data and control dependency between various subsystems. However, systems can be designed with handshaking mechanism or NoC as along as they can be defined by block diagrams using the library components. These tools also provide clock true simulation environments to verify the design before implementing. Simulating the radio as a system and not just at an algorithmic level is important because unlike serial computation model, multiple radio processes operate in parallel to form a complex network of subsystems. Thus, it is important to consider all these factors for selecting the proper tool flow and compilers to implement future SDCRs.

MAC-PHY crosslayer protocol design: Researchers have long been focused on the layered architecture of the network protocols and were limited to optimizations and improvements in one particular layer at a time. With the advent of new chipset designs from manufacturers like Atheros, researchers were presented with a higher degree of control over how a device can interact with the communication medium. Now, low-level hardware parameters can be changed using device drivers, like the MadWiFi driver [1] that interacts with the physical hardware to optimize some facets of the waveform. This unprecedented control over the PHY allowed researchers to delve into the physical layer as well. Since few parameters are exposed to the user space it offers limited coverage of the entire space of crosslayer design.

GnuRadio [4], on the other hand provides a very flexible framework. Since the entire PHY is implemented in software using a high level language, it is much easier for a programmer to exploit different aspects of the PHY beyond that was available with the Atheros based hardware. Only drawback is that the entire processing is done offline which makes it harder to test real-time performance of these protocols. These two factors have spurred tremendous efforts in MAC-PHY cross layer protocols over the past decade and still continues to make active contributions to the community.

The similarity among different communication protocols is also reflected in their corresponding physical layer. There is an increasing demand to redesign the common processing engines to perform most of the functions in a fast changing environment of cognitive radio. A close look at the current wireless protocols reveal the need to define a more fundamental set of primitives, beyond just the parameters or the functional operation of a particular transceiver subsystem: e.g., instead of having correlators with fixed coefficients, we should have a method to change the coefficients required for a particular protocol. This is typically required in a cognitive radio environment where the available spectrum varies over time that changes the time domain correlation coefficients [48, 49].

With various concurrent wireless protocols in mind, we define a set of basic capabilities, a generic transceiver should have in order to operate as a SDCR. This deconstruction of the radio physical layer is motivated by a substantial amount of prior research and publication in the cognitive radio and wireless networking community. We focus on OFDM based physical layer as this is a promising enabler for future cognitive radio design. We list the various subsystems of the transceiver and highlight the research that require modification of these subsystems:

- SDCR should be able to transmit and receive in any set of subcarriers [50, 6, 51, 52, 7, 53, 8, 54].

Essentially it should support non-contiguous OFDM transmission and reception so that nodes can communicate using fractured spectrum.

- Not only does it need to adapt to changing spectrum availability, the SDCR should be able to change its modulation (e.g: BPSK, QPSK, 16QAM, 64QAM) at a subcarrier level. Also, high throughput wireless PHY layer techniques require advanced modulations such as superposition coding and hierarchical modulation, which require a high degree of programmability in the modulation levels. In [55, 56, 57, 58, 59], we can find requirement of such systems.

- Depending on the availability of spectrum, the SDCR needs to change the FFT size to control the number of subcarriers to be used for the transmission. Also depending on the channel conditions the duration of the cyclic prefix needs to change to combat multipath channel distortions. WIMAX 802.16 [50] and LTE [60] are examples of wireless protocols that directly requires this capability.

- In order to support NC-OFDM transmission, the SDCR receiver needs programmable correlators which can support arbitrary set of correlator coefficients from a pre-defined superset as chosen by the protocol [48, 61]. [53, 62, 63, 64] are such examples that require synchronization of NC-OFDM preambles which changes the correlator coefficients from one packet to another. Detection of signal based on embedded signature in the waveform is an active area of research. [65] is an example of a cyclostationary detector that is commonly used. These examples show that composing structures for new algorithms is required.

- Channel estimation and Equalization is an important signal conditioning step used in the receivers. To accommodate the changing environment, the transmitter selects different set of pilot subcarriers [66, 50, 67] to assist in the equalization at the receiver end. Therefore the pilot locations and their relevant phase is an important information that the receiver needs to have in order to equalize a NC-OFDM signal.

- Error correction module provide key information about channel conditions that lead to more informed decision in higher layer rate control algorithms [68, 69]. Also, research have shown a reconfigurable FEC engine [70] is useful is a multi-protocol environment. Not all protocols used all of the FEC modes so instead of over provisioning one should be able to derive the structure from a high level specification for a particular FEC algorithm.

- A very important aspect of Cognitive Radio is **Sensing** [71] for detection of the primary user a channel, which requires post-processing of the FFT results. No other transmitter or receiver blocks are used in this mode. Recent works on simultaneous transmission and reception [7], [52] show requirement of a simple FFT block at the receiver and a threshold based detection. So, a transceiver should be able to perform only FFT and hand over the results to the **Cognitive Engine** to make decision.
- Other physical layer techniques like MIMO or jamming [72, 73, 3] are also implemented as a part of the PHY and should be addressed in the design specifications. Sometimes custom logic is required beyond what has been mentioned in the literature. Re-routing of signals and adding user control signals is also very common in cognitive radio application. Such cases are to be dealt case by case basis and is not possible to make generic across all possible scenarios. But the tool flow to allow that kind of reconfiguration is currently not available for expressing the physical layer of modern cognitive radio.

All of the above examples are either implemented using a full software stack like GnuRadio or uses the limited functionality of the Atheros based Wi-Fi cards. Unfortunately none of the current methods or solutions mentioned in the literature can support the emerging need for MAC-PHY cross layer protocol designers. The current methodologies are brittle and are not properly abstracted for multi-disciplinary contribution rendering it inefficient in most cases or limited in functionality. Authors in [74, 75] have built a fully functional Wi-Fi ($802.11a/g$) transceiver that provides most of the functions that are addressed above. In [5], authors propose an architecture for modern cognitive radio PHY. However, hand coded controls pose severe impediment while modifying the design to incorporate new features or remove obsolete ones.

Cognition and collaboration at the physical layer: The core of the cognitive radio is the cognition engine. Cognition can be implemented at various aspects of the radio. The main goal of the cognition engine is to learn and adapt to the changing environment. Also, a radio has to be self-aware and if required can be told what to do and how to do it. All of these is typically done by some form of a computing and reasoning machine without the involvement of the user. In [16] the authors layout some practical requirement for the language used to represent the cognitive engine. They claim that the cognitive engine should be able to represent the functions of a radio using an Ontology, which is a declarative way of expressing the relationship of the various radio component. The relationship thus expressed should include the MAC, the PHY and the radio control plane as one unit and not any one of them in isolation. In [18], a simple abstraction of this has been shown by representing the programmable components of the radios as *knobs* and using a reasoning machine to decide on which parameters to change based on the knowledge obtained from the environment. The goal is to maximize certain metric of the link e.g., error rate, data rate, transmit power etc. However, as radios and waveforms become more complex, the number of tunable parameters increase to the extent that parametrizing the variables become nearly impossible. Fixed hardware pipeline and dataflow is no longer optimum and there has to be a way to compose radio structures on-demand. An important aspect of cognition is *understanding your neighbors* and facilitating coexistence in a network topology. An example of radios of different capabilities collaborating to achieve a better communication link has been shown in [76, 77]. The Wireless Innovation Forum has developed the first Ontology based cognitive radio that is capable of transmitting an audio waveform using CDMA technology [78]. A lot of effort in this project has been spent on developing the policy framework that can used to control various *knobs* on the radio pipeline. The pre-cast radio suffers from similar brittleness as an partially programmable ASIC if we need to change one of the components, however small it may be, it requires re-designing the entire pipeline to ensure proper synchrony between components. Therefore, abstractions at different design stages is the key to develop cognitive radio for the longer run.

Chapter 3

Physical Layers: Implementation of OFDM transceiver

One of the goals of CODIPHY is to enable research in wireless networking and cognitive radio using highly programmable radio prototype hardware. Therefore, in order to understand the specifications of such a prototype, we have to focus on the physical layer that is widely used in the wireless networking research community. 802.11a/g, that operates in the unlicensed ISM band of 2.4 GHz has been the focus of research, specially in the MAC layer. In recent years, researchers have also focused in MAC-PHY crosslayer techniques to further improve various aspects of wireless networks. Therefore, OFDM, which is the PHY layer waveform used in 802.11a/g networks, is a plausible choice for prototype platforms. The choice of OFDM as the PHY for prototyping also has larger implication in terms of its broad acceptance for high throughput data communication and cognitive radio. Therefore, once in place, the prototype will not only enable 802.11a/g crosslayer research but also contribute to a broader realm of research.

In this chapter, we present the design and implementation of an OFDM based 802.11a/g physical layer on FPGA. This is the foundation for defining the specifications for an evolved radio PHY for crosslayer and cognitive radio research. This design step also provide insights into the components and building blocks of OFDM base PHY that ultimately leads to *composability* of the radio by using a component based design approach discussed in chapters 6, 7 and 8. A practical prototype also includes other add-on systems like the ADC/DACs and the RF front-end circuits. In order to ensure that the properties of the transmitted waveform is preserved and also to enable testbeds and experimentation, we also include the interface design to program these peripherals. Design and specifications for user interface is also discussed in this chapter.

3.1 Inside the OFDM transceiver

Any waveform used in wireless communication is governed by a mathematical function. Analyzing the functions, one can extract the components that are required to compute it. These components are independent of the implementation and only specify the relation of the input variables to the output variables. This process is similar to specifying a system using a transfer function where the system itself is treated as a black box and the transfer function is sufficient to understand the functionality of the system. Equation 3.1.1 represents the continuous-time baseband representation of the OFDM signal.

$$r(t) = \sum_{k=-N_{ST}/2}^{N_{ST}/2} C_k exp(j2\pi k\delta_f)(t - T_{GUARD}) \tag{3.1.1}$$

Where, C_k represents the coefficients of an inverse Fourier transform and the exponents specify the frequency domain components of the signal. The output signal contains a cyclic prefix equal to duration T_{GUARD} at the beginning of the each OFDM frame. Figure 3.1 shows an example spectrum of an OFDM waveform along with frequency selective fading. The mathematical equation of OFDM is computed using the Inverse Fast Fourier Transform (IFFT) algorithm, that can be efficiently implemented in software as well as in hardware. The coefficients of the IFFT however require multiple stages of processing to introduce error protection. This is typically done in three steps: scrambler, interleaver and forward error correction codes. Each of these steps will have their own mathematical functions and corresponding components. Figure 3.2 shows the high level subsystems in an OFDM transceiver. The receiver of a communication systems is often reciprocal of the transmitter. Therefore, it is useful to analyze the transmitter and receiver together as they are sometimes expressed by the same structure. For example a scrambler and descrambler use the same structure and so does the interleaver. On the other hand a convolution encoder is completely different from its decoder as they use different algorithms to encode and decode. Also, in order to compensate for the channel noise some special purpose signal conditioning techniques are used to improve the signal to noise ratio at the receiver. Equalization, carrier frequency offset correction and digital filters are typically used to reduce imperfections in the signal to improve the error rate of a wireless packet. It is our aim in CODIPHY to extract the components and aggregates of an OFDM transceiver and represent the subsystems using a functional description of the dataflow. In the following sections we discuss the subsystems in the OFDM transceiver and identify the components used to represent the computation flow. These aggregates facilitate the component based design independent of how they are implemented on a platform.

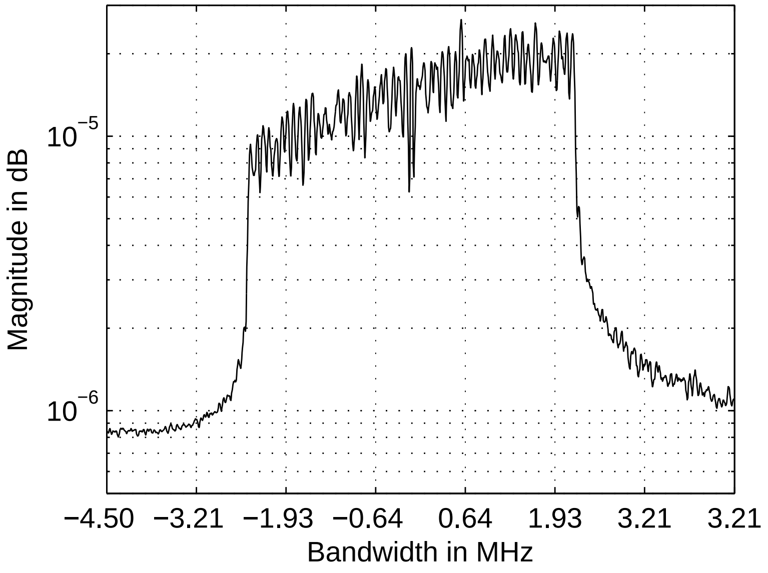


Figure 3.1: OFDM spectrum with frequency selective fading

For this implementation we chose the 802.11a/g [79] physical layer with some changes to the specification. While keeping the subcarrier spacing constant, the FFT/IFFT size has been increased to 256 instead of 64. This allows us to accommodate more number of subcarriers. However, in order to make the system compatible with off-the-shelf 802.11a/g network cards, the baseband is operated at 80MHz, instead of 20MHz to keep the OFDM symbol duration constant. Also to maintain compatibility with WiFi, only 64 out the available 256 subcarriers are used. However, the user is allowed to use all the subcarriers for research purposes and the framework supports that feature. Thus, the prototype provides greater throughput but at the expense of a higher clock frequency for the baseband receiver and higher bandwidth. Since the design operates at a higher clock frequency, the number of samples processed per unit time is four times as compared to conventional 802.11a/g receivers. The design also supports variable OFDM bandwidth by controlling the baseband clock frequency. Different bandwidth also alters the OFDM symbol duration. In the following sections we present the various receiver subsystems and their hardware implementation. The

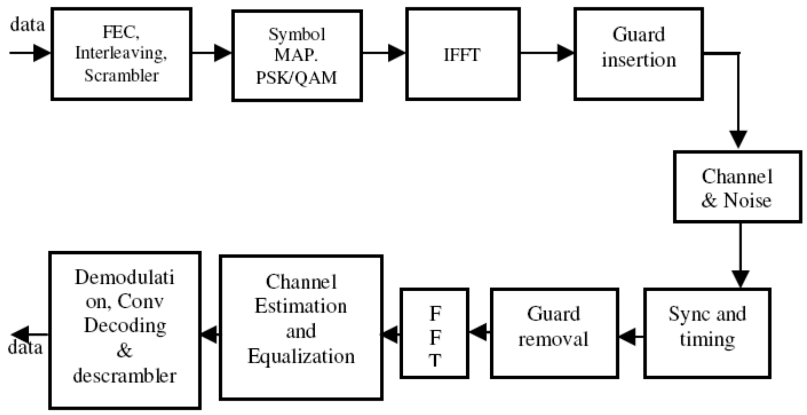

data
FEC, Interleaving, Scrambler
Symbol MAP. PSK/QAM
IFFT
Guard insertion
Channel & Noise
Sync and timing
Guard removal
F F T
Channel Estimation and Equalization
Demodulati on, Conv Decoding & descrambler
data

Figure 3.2: A generic OFDM transceiver

design methodology uses Xilinx System Generator [19] design tool to design hardware synthesizable DSP systems. It also produces a cycle-true, bit-true design that can be verified in the same environment using test signals. The OFDM transmitter is based on prior work that has been developed at University of Colorado Boulder[74].

3.2 Radio Platform and System Level Components

A OFDM transceiver is a combination of a number of systems that are interconnected to operate as one single entity. The baseband processor performs bulk of the signal processing but careful design of other systems like the peripherals, user control and daughter cards are also essential in a complete system. Also, some of these subsystems also has interfaces for programming and control. In this implementation we use a prototype radio board, shown in figure 5.22. The following are the features of the platform:

- The baseboard consists of a Xilinx evaluation kit with a Virtex 5, LX110 FPGA [80].
- The A/D and D/A are connected as daughter card using the expansion slots on the baseboard. The digital board contains Analog Devices AD9862 IC.
- The RF board is capable of transmitting in 2.4GHz and 5 GHz band is also connected to the FPGA using the expansion slots. The main component is the Maxim MAX2829 IC.

- There are several options for user control but in this implementation we use the Ethernet as the main communication interface with the host.

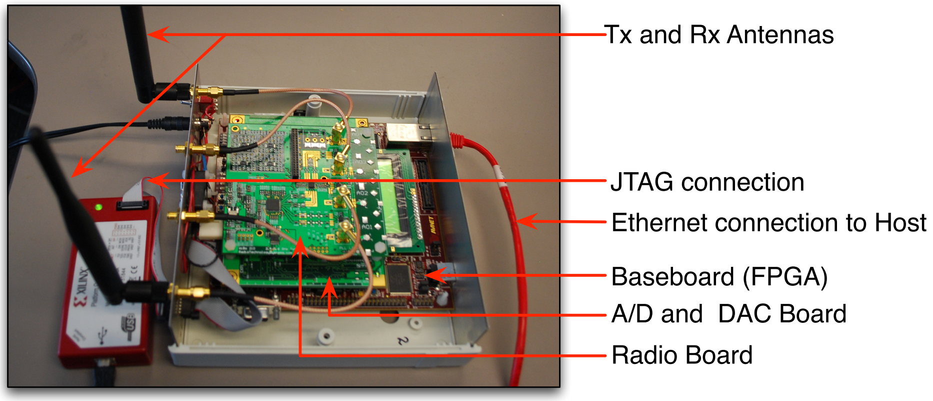


Figure 3.3: Software define radio platform at University of Colorado Boulder

The top-level schematic is shown in 3.4, which contain four distinct clock domains: the baseband operating at 80MHz, Ethernet transceiver operating at 25MHz in half-duplex mode providing a throughput of 100Mbps, a SPI serial interface for programming the RF and the digital boards operating at 16MHz and the output I/Q multiplexing stage to the DACs operating a twice the baseband frequency or 160MHz. Data transfer between these clock domains is handled by asynchronous shared memories and registers in System Generator.

3.3 Ethernet Module

The Ethernet module is a modeled as a simple Ethernet packet detector and parser, that is used to receive control and data packets from the host and also transmit the decoded OFDM packets to the host. The Ethernet receiver and transmitter is shown in figure3.5 and 3.6 respectively. The various command labels and the corresponding address ranges are in Table 3.1. The Ethernet transceiver, consists of the following:

Packet detector: Looks for the Ethernet header and extracts the control information from the payload.

Parser State machine: Used to parse the control information included in the Ethernet payload and send to the baseband processor or the SPI module for programming various parts of the radio. The control bytes are

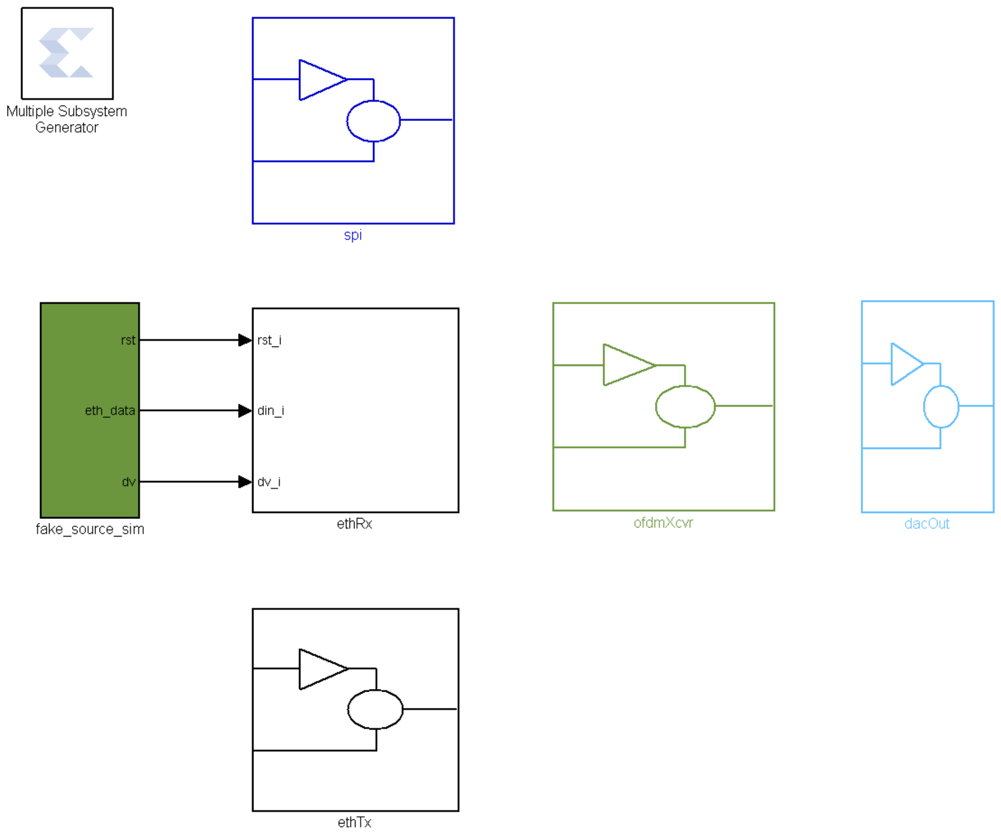


Figure 3.4: Top-level schematic for the OFDM prototype

written to shared registers which is latched onto other clock domains.

Data Buffers: To store the data bytes to be modulated by the transmitter and another buffer is used for the data bytes decoded from the receiver.

Ethernet transmitter: A packetizer to transmit the received data along with values of six status registers after appending 32bit CRC and valid Ethernet header.

3.4 SPI Serial Interface

The digital and RF daughter boards are programmed using a 4 wire serial SPI interface using the Maxim MAX 7301 IC. The main purpose of this module is to serialize the configuration bytes that are received from the Ethernet control frames. It is also used to generate the chip enable signals for the SPI controllers for the digital and the RF daughter cards. The AD9862 provides a 12bit ADC and 14bit DAC

Table 3.1: Radio configuration specifications

Type	**Label**	**Address**	**Format**
Register Update	0xAB	0-12 - Wibo Register 13 - Tx/Rx/ShutDown 14 - Antenna Configuration 15 - SPI Enable Register 16 - Read OFDM Status Register 100 - 164 - ADC/DAC Registers	$[label(1byte), address(1byte), config(2byte)]$
Data Write	0xAC	pkt_cnt[1]	$[label(1byte), pktcnt(1byte), length(2bytes), ofdmdata(variable)]$
Data Read	0xAD	Not Applicable, set to $0x00$[2]	$[label(1byte), 0x00, Rxconfig(2byte)]$
Data Transmit	0xFF	Not Applicable, set to $0x00$[2]	$[label(1byte), 0x00, Txconfig(2byte)]$
SPI Register read	0xAE	100 - 164 - ADC/DAC Registers[3]	$[label(1byte), address(1byte), 0x0000]$

[1]For longer OFDM payload requiring multiple Ethernet packets, [2]Address not required, [3] SPI data read available for AD9862 only

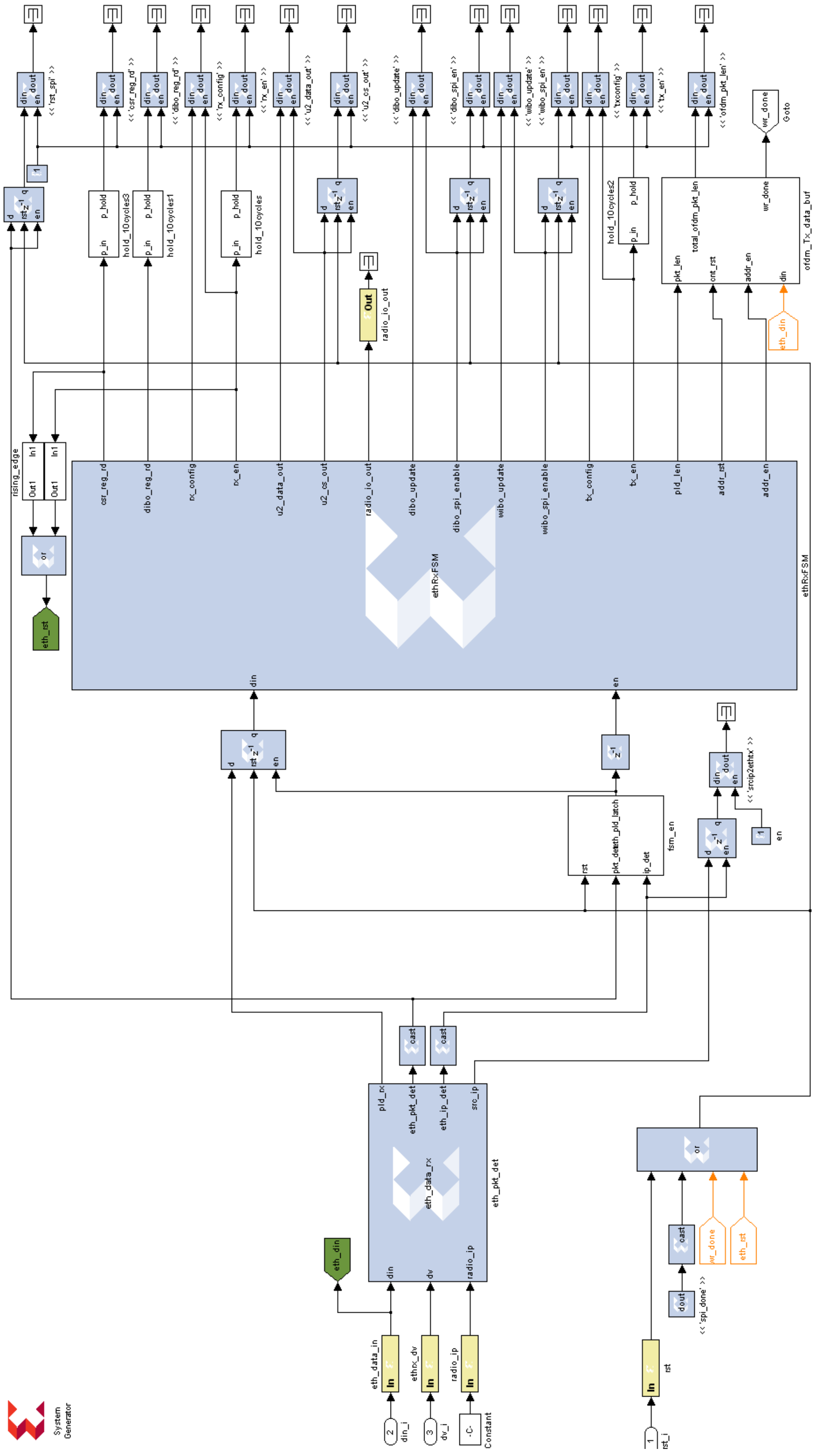


Figure 3.5: Ethernet receiver

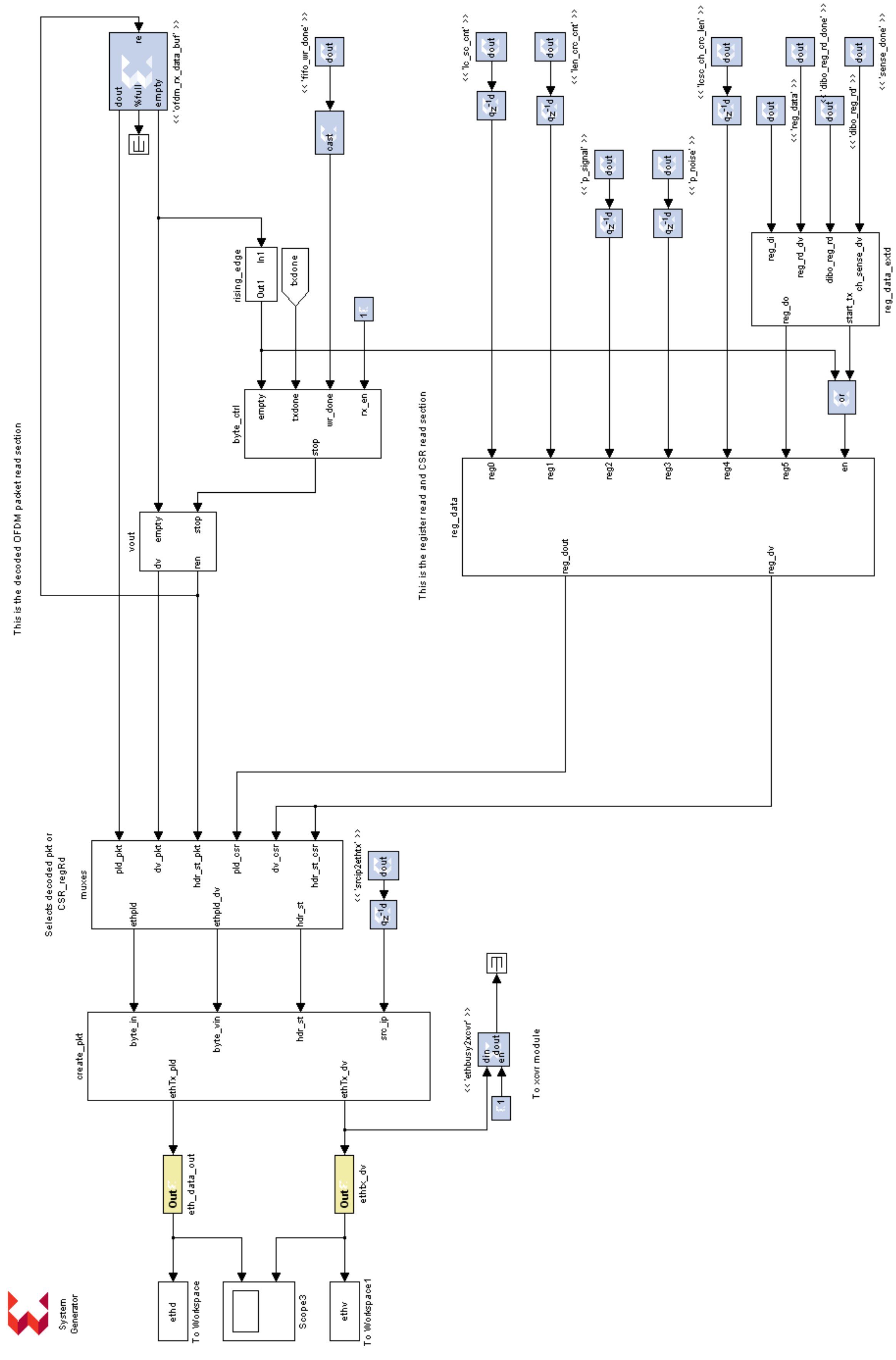


Figure 3.6: Ethernet transmitter

and the various configurations are programmed using 64 control registers. The SPI module is also used to program the Maxim RF IC and also to set the TX/Rx mode of the duplexer. There is also a mechanism to read values from the registers of the AD9862 IC. The serial interface module is shown in figure 3.7

3.5 Baseband Module

The baseband module is the core of the OFDM transceiver. The top-level design is shown in figure3.8 and 3.10. The transmitter contains the sample domain computations in the FPGA as shown in figure 3.9 and is based on previous work [74]. All the subsystems of the receiver has been synthesized from scratch as shown in figure 3.11. The transceiver modules contain two control registers, *tx_config* and *rx_config* to set the mode of operation of the various subsystems. The various operating mode is discussed in chapter 5 where the OFDM transceiver is used in wireless networking research to improve various aspects of wireless and cognitive networks. The output of the transmitter is connected to shared memories of the I/Q multiplexing stage to interleave the I and Q samples to the AD9862 IC.While, the output of the receiver is connected to shared memory interfaces to the Ethernet transmitter module to be transmitted as Ethernet packet to the host. We discuss the design details of receiver subsystems in the following sections.

3.5.1 Packet Detection

The packet detection subsystem utilizes the periodic nature of the short preamble. Every 802.11a/g packet has a short preamble of 10 symbols each 64 samples long. Schmidl and Cox [81] explains a packet detection algorithm which has been used in many implementation. The short preamble symbols are periodic with a period of 64 samples. The upper branch calculates the autocorrelation energy between r(n) and r(n+64)which is given by,

$$C(d) = \sum_{i=0}^{D-1} (r_{d+i}^{*} . r_{d+i+D}) \tag{3.5.1}$$

where D = 64. The lower branch computes the signal energy during that autocorrelation window, given by

$$R(d) = \sum_{i=0}^{D-1} |r_{d+i+D}|^2 \tag{3.5.2}$$

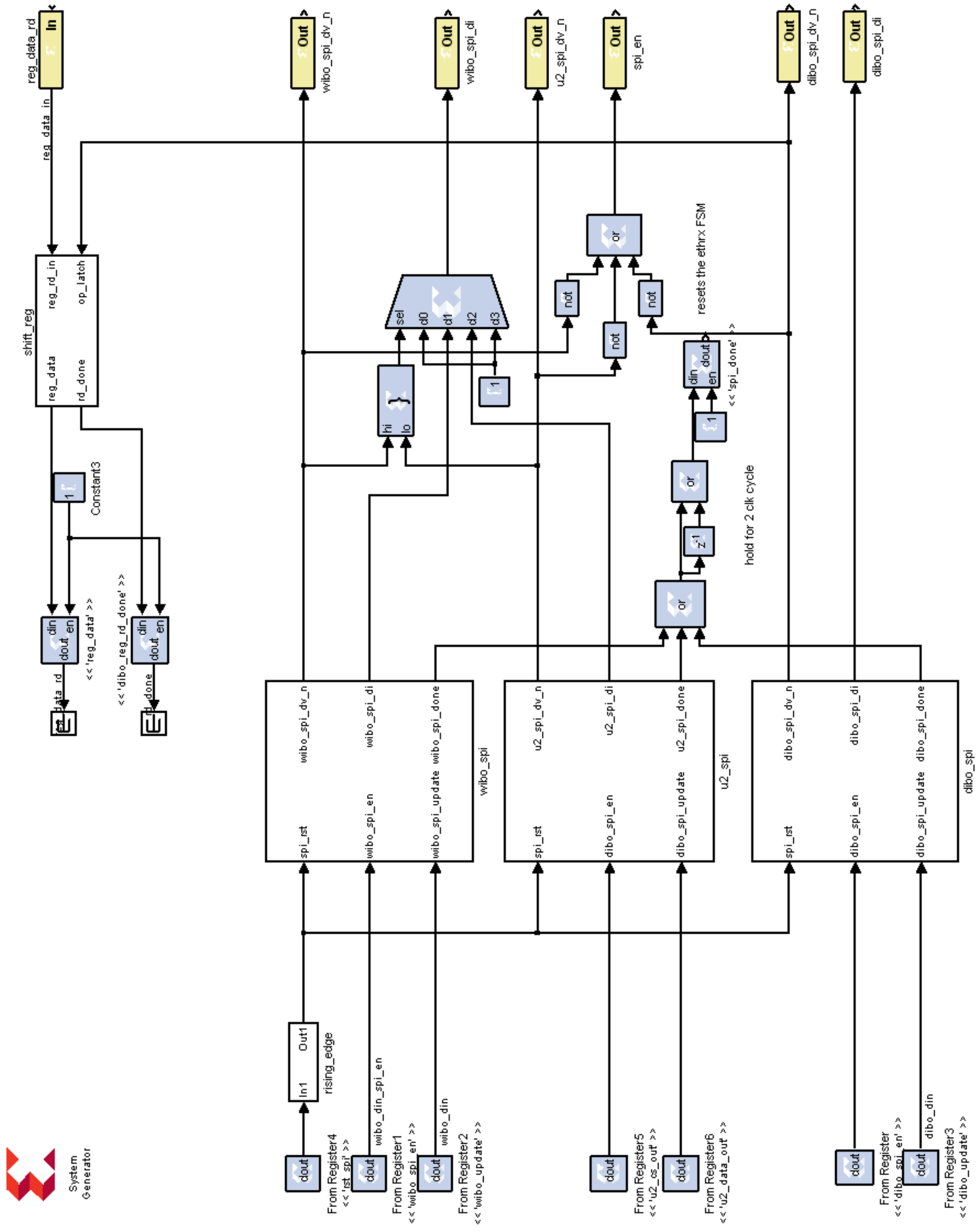


Figure 3.7: Serial SPI interface

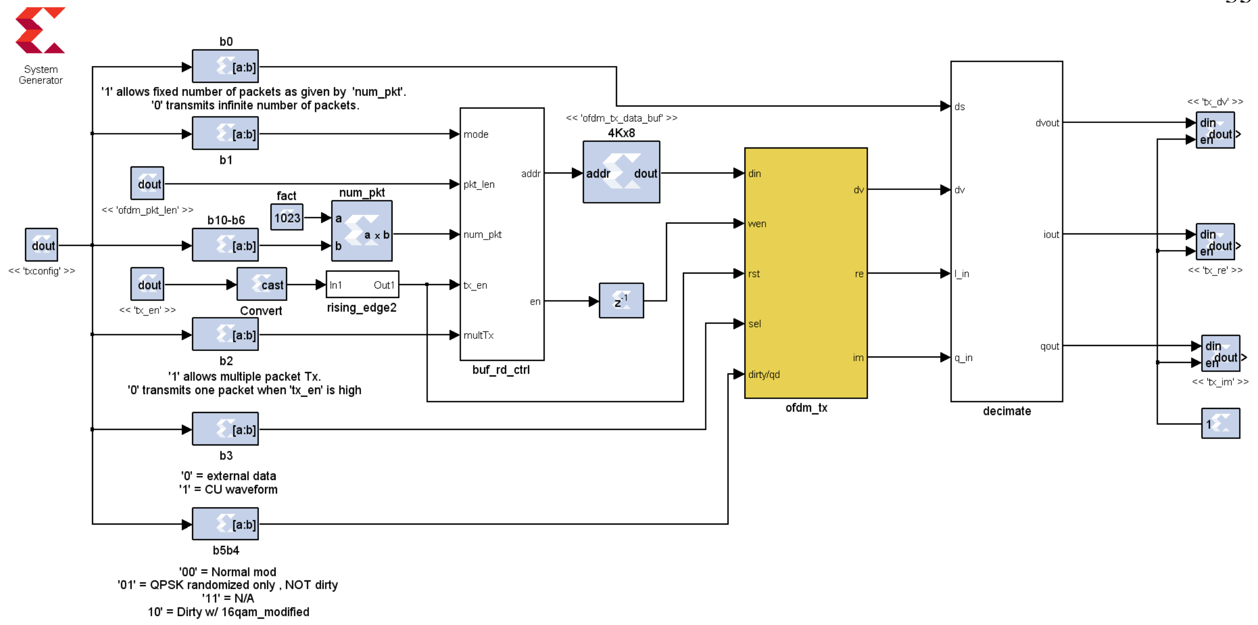


Figure 3.8: OFDM transmitter top-level

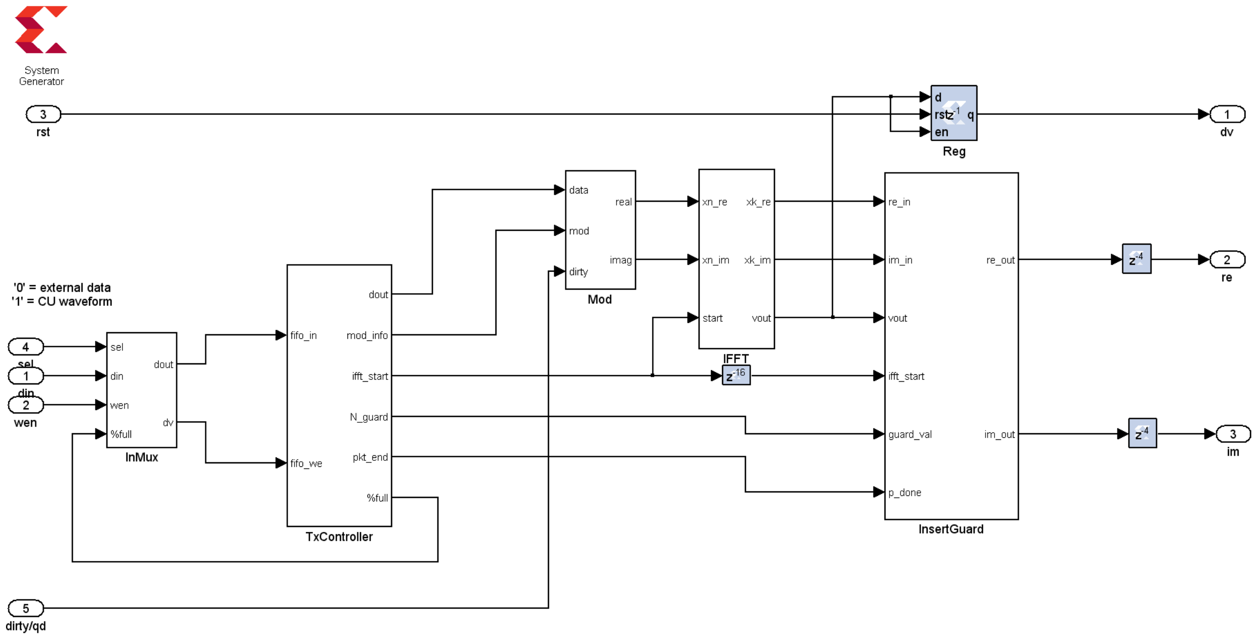


Figure 3.9: OFDM transmitter subsystems

The decision metric is obtained as

$$M(d) = \frac{C(d)}{R(d)} \tag{3.5.3}$$

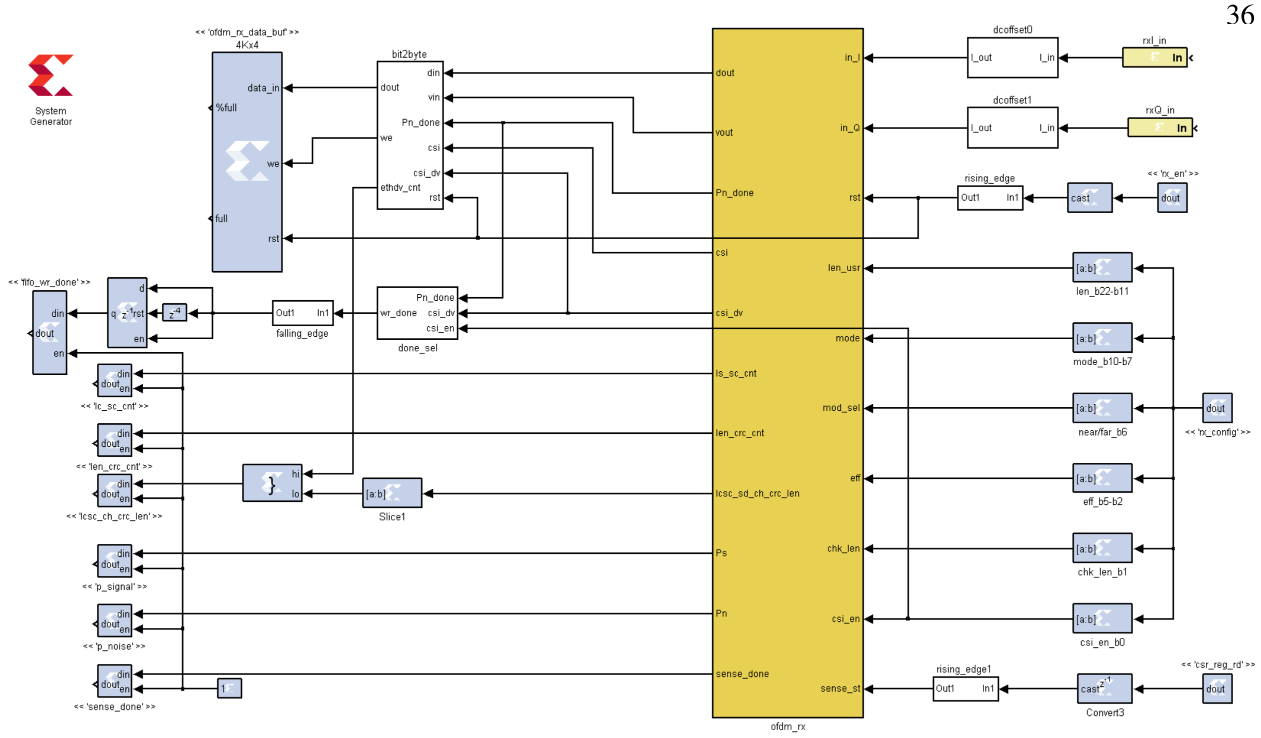


Figure 3.10: OFDM receiver top-level

Both of these correlation contain a sliding window sum unit that is implemented using a single stage Cascaded Integrator Comb (CIC) filter expressed by the iterative eq. (3.5.4) ans implemented as in figure 3.12.

$$y(n) = y(n) + x(n) - x(n - D) \tag{3.5.4}$$

In hardware the CIC is realized using a subtractor and an accumulator after computing the correlation energy. Therefore three CIC units are shown in figure 6.3 along with the two branches of the packet detector as described above. The division in computing the decision metric as given in eq. (3.5.3) is avoided by scaling R(d) and comparing with C(d). Fig. 3.14 shows the output of the packet detector. The plateau shape of the output results from the periodic nature of the short preamble. The autocorrelation energy starts to build up and remains constant after 2 symbol periods. The second trace showing the signal energy during the autocorrelation phase which also builds up and goes constant from the second symbol as all the short preamble symbols have the same energy. In the presence of noise, the plateaus will not be flat but will have noise associated with them. The packet detect module has been shown to detect packets in the presence of noise. The scaling factor(threshold) for the signal energy is determined through experimental measurements

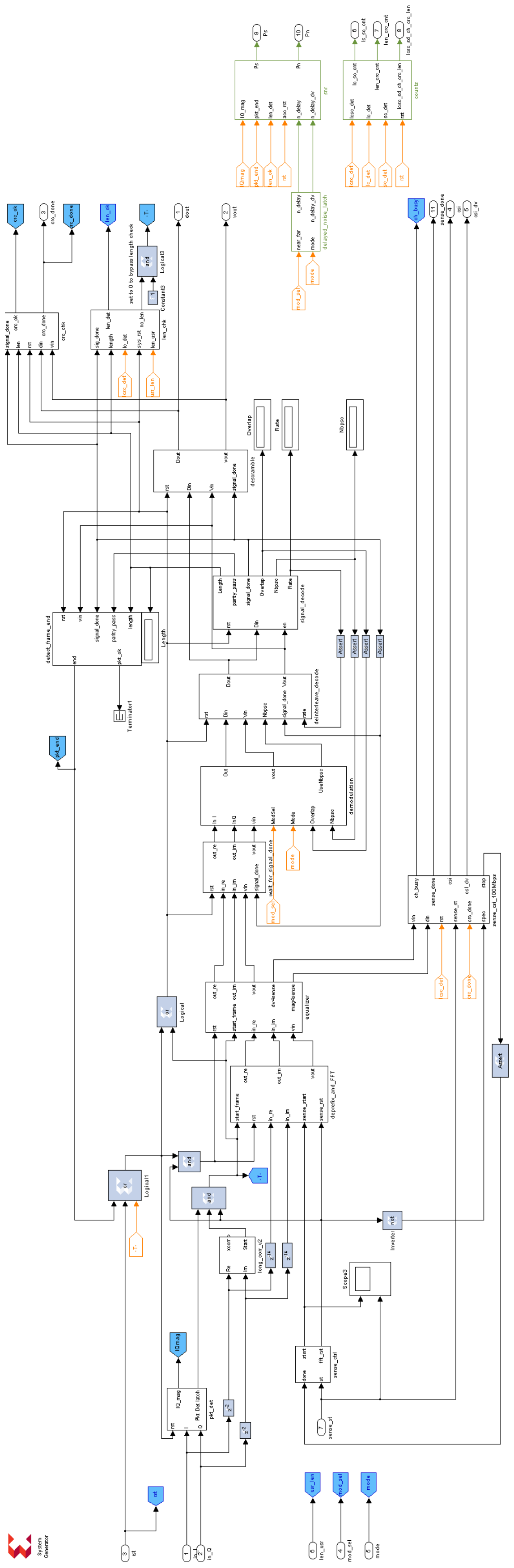


Figure 3.11: OFDM receiver subsystems

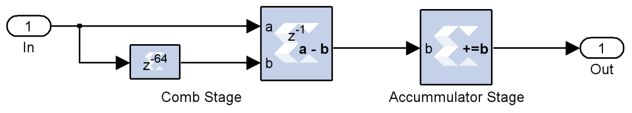


Figure 3.12: Single stage CIC filter

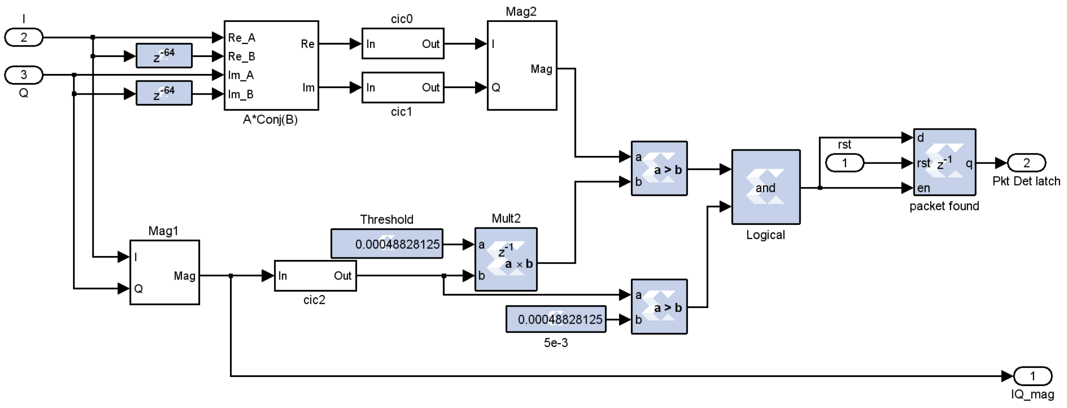


Figure 3.13: OFDM packet detector

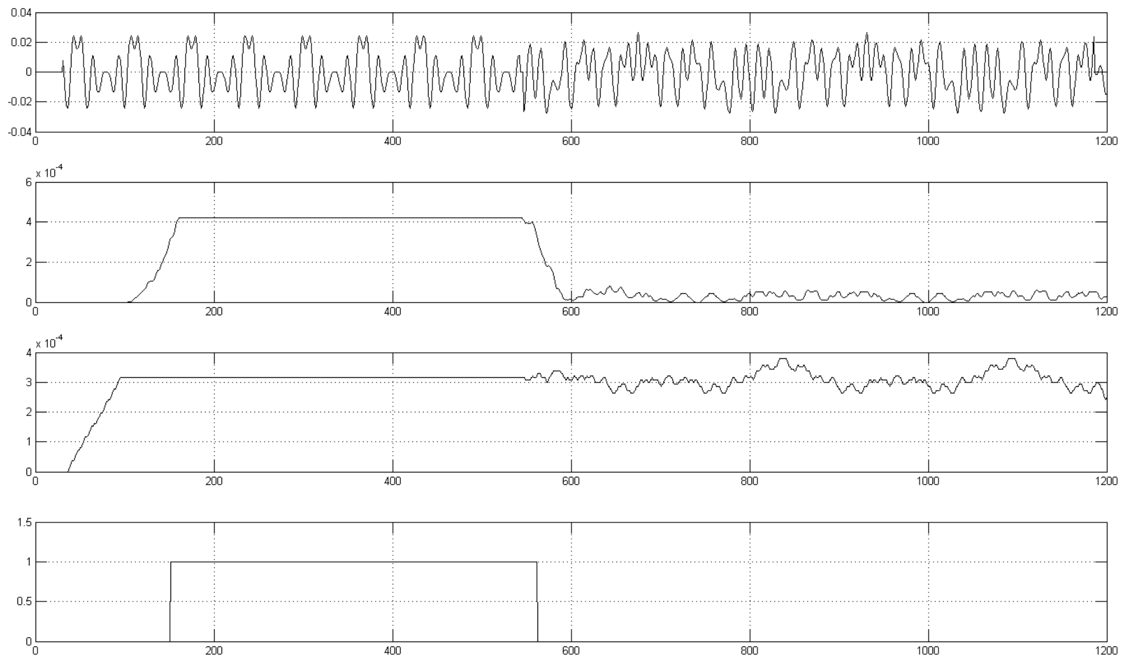


Figure 3.14: Packet detector output

so that it performs over the required signal to noise ratio as seen on normal 802.11a/g channels.

3.5.2 Carrier Frequency Offset Correction

Since the transmitter and the receiver clocks are powered by different crystal oscillator there will always be an offset between the two. At the receiver during down conversion the oscillator will try to tune to the desired center frequency but due to oscillator drifts, let there be an offset of δf. The transmitted signal can be expressed as

$$y_n = s_n e^{-j2\pi f_{tx} n T_s} \tag{3.5.5}$$

after downconversion we get

$$r_n = s_n e^{-j2\pi \delta f n T_s} \quad \text{where} \quad \delta f = f_{rx} \sim f_{tx} \tag{3.5.6}$$

Since the preambles are periodic with periodicity D = 64 sample for short preambles and D = 256, we can use this property to extract the frequency offset δf as follows

$$Z = \sum_{n=1}^{L-1} r_n . r_{n+D}^{*} = e^{-j2\pi \delta f D T_s} \sum_{n=1}^{L-1} |s_n|^2 \tag{3.5.7}$$

L and D are the integration length and periodicity which are different for short and long preambles. The longer the integration time the better is the frequency estimate. Therefore for short preamble, L = 128 and D = 64, where as for long preamble L = 256 and D = 256. The frequency offset can directly computed using the autocorrelation energy in eq. (3.5.7) as follows,

$$\delta f = \frac{1}{2\pi D T_s} \arctan(Z) \tag{3.5.8}$$

Translating the algorithm in the hardware involves the following design block/steps:

- Detect the short preambles to trigger the change of integration time (from L = 128 to L = 256)
- Calculate the frequency offset by calculating the correlation energy and performing the arctan operation.
- A phase accumulator to latch on to the correct phase at the end of the coarse and fine estimates.
- Finally a frequency synthesizer to generate the desired frequency equivalent to the calculated phase offset.

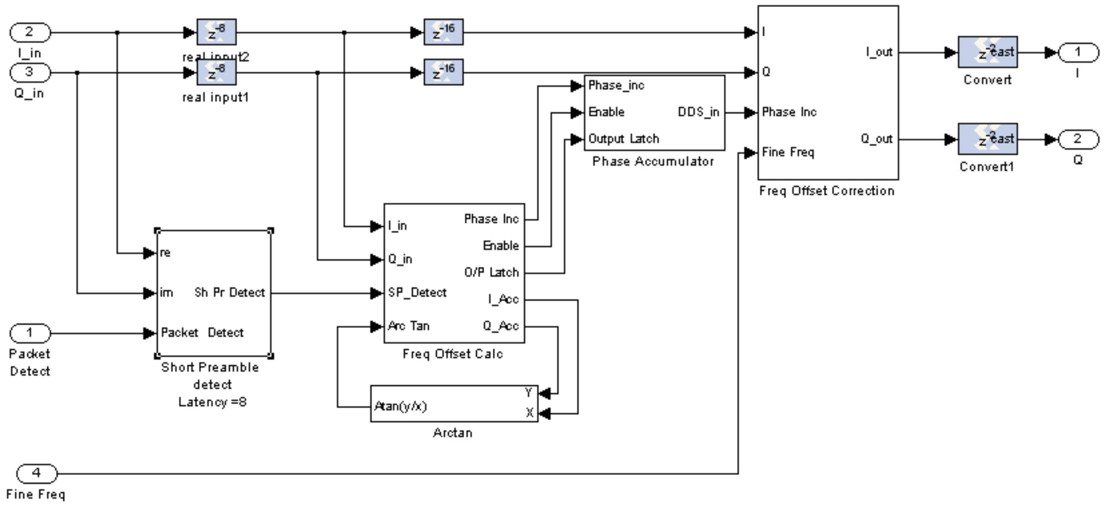


Figure 3.15: Carrier frequency offset block

Fig. 3.15 shows the hardware logic design. Arctan is implemented using the Xilinx Cordic Arctan IP. The short preamble detection is performed by correlating the input signal with a local copy of the short preamble stored at the receiver. The detailed implementation is explained in the next section. Fig. 3.16

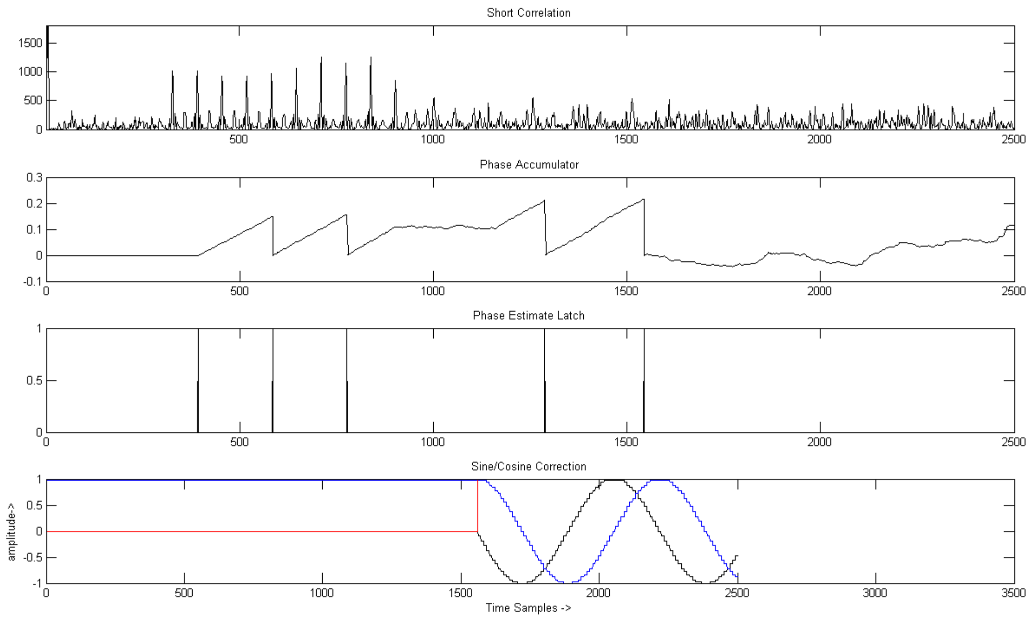


Figure 3.16: Carrier frequency offset correction

shows a typical output of the phase accumulator. The correlation energy is latched during the short and long preambles. The coarse estimate is updated twice during the short preamble and then the estimate is made finer twice at the end of the two long preambles. The phase estimate is finally latched and fed to the Direct Digital Synthesizer(DDS) to produce the offset frequency which is then multiplied with the incoming signal using an I/Q complex multiplier.

3.5.3 Long Correlation and Packet Timing

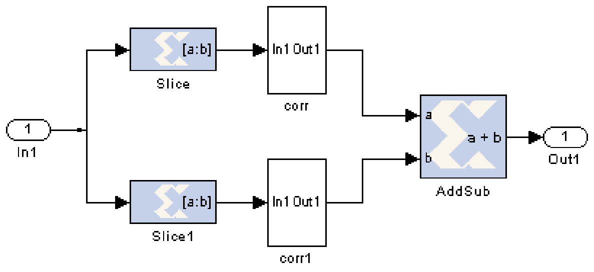
Once a valid packet is detected by the packet detect block and carrier frequency offset has been corrected the long correlator block is used to extract the correct timing for the start of OFDM payload which is defined by the beginning of the signal symbol. This is extremely important for the performance of the receiver because this triggers the FFT block to convert the input signal into frequency domain. The long preamble in 802.11a/g consists of two symbols, each 256 samples long preceded by a 128 sample long cyclic prefix. Correlation is performed with a local copy of the long preamble. Correlation involves 256 sample long multiply-add and one shift operation at every clock. In order to eliminate the use of 256 multipliers, which consumes a lot of FPGA resource, the correlator uses logical operations to perform the correlation as follows:

(1) Use the “sign” bit of the I and Q samples instead of the full precision (12 bits) to eliminate multipliers.

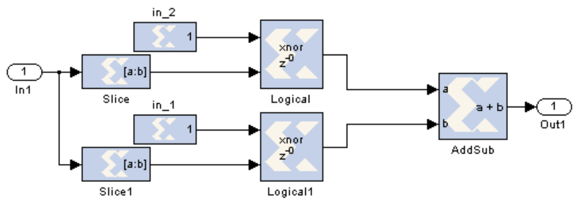
(2) The local copy of the long preamble also consists of the sign of the time domain signal.

(3) Since the objective of correlation is to search for an exact set of samples as stored in the receiver, we use XNOR to compare the similarity between the incoming signal and the local copy of the long preamble. Whenever the sign of the input sample matches to that of the local copy the output is a ’1’ else ’0’. Accumulating the result for 256 logical comparisons gives the correlation energy which is compared to a threshold to trigger the FFT unit.

Fig. 3.17 shows various processing elements involved in the correlator. Figure3.17(a) shows one processing element of the correlator. Each processing element is responsible for correlating two samples of the preamble. Therefore, there are 128 processing elements running in parallel for a 256 bit correlator. Figure3.17(b) shows the logical comparison using *XNOR*. The output of each processing element is accumulated using a two-input adder tree.



(a) A processing element of the correlator



(b) Two-bit correlation using XNOR

Figure 3.17: Long correlation using *sign* bit

The output of the long correlator is shown in figure3.18. The two peaks mark the end of the two long preamble symbols. The second peak is detected and marks the end of the preamble and start of the *signal* symbol. This signal is used to trigger the FFT unit.

3.5.4 De-prefix and FFT

Every OFDM symbol carries a 64 sample long cyclic prefix. This cyclic prefix carries no information and is used to compensate for any multipath effect. Therefore, these 64 samples is removed for every OFDM symbol. The FFT trigger input from the long correlator is used to load data into the FFT unit. The FFT engine used here is the IP block from Xilinx. The FFT engine along with the cyclic prefix removal is shown in fig. 3.19. The counter keeps track of the sample count to be skipped, which is 64 in this case and then triggers the FFT to ensure that the FFT window has the correct samples at the input. The effect of deviations in this timing has been discussed in [82, 83, 84, 85]. Therefore, this timing is of paramount importance to successfully decode an OFDM symbol. The FFT engine is operated in “pipelined streaming mode” and has been optimized for speed. However the latency in the FFT engine is significant compared to the latency of the entire receiver which is discussed in §3.6. The output of the FFT produces the frequency domain representation of the OFDM symbol. The subcarriers are indexed from -128 to 127 for a 256 point

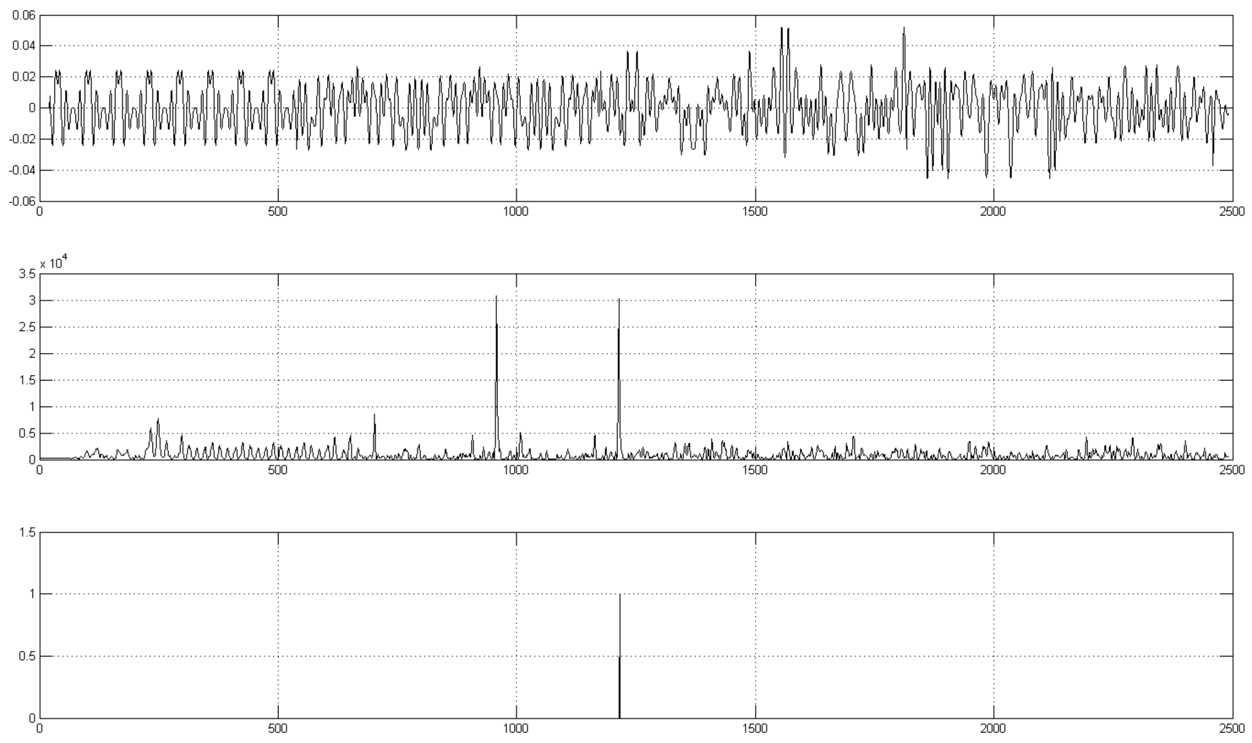


Figure 3.18: Long correlator output

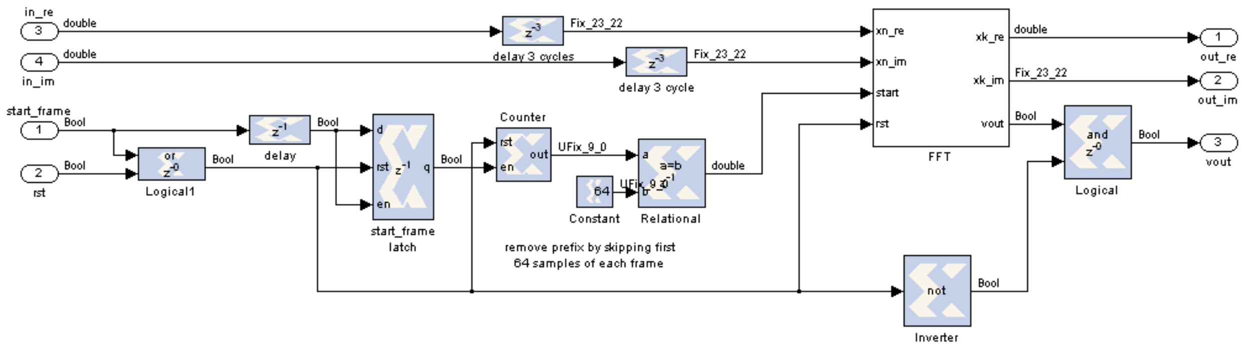


Figure 3.19: De-prefix and FFT subsystem

FFT.

3.5.5 Equalization

The equalizer is an integral part of the receiver. Many equalization algorithms have been discussed in [82, 84, 83, 85, 66, 86, 87]. The equalizer is responsible to correct phase and magnitude errors in the signal introduced by the time-varying, frequency selective wireless channel. A wireless channel is modeled as a sum of a number of delay paths. If h(t) be the time-domain representation of the channel for M different delay paths can be written as,

$$g(t) = \sum_{m=1}^{M} \alpha_m \delta(t - \tau T_s) \tag{3.5.9}$$

where α_m is a complex valued coefficient.

At the receiver a N point DFT is used to transform the data back to frequency domain.

Let $\bar{X} = [X_k]^T$ and $\bar{Y} = [Y_k]^T$ for (k = 0,. . . ,N-1) denote the input data of the IDFT at the transmitter and the output data of the DFT at the receiver respectively. Let $\bar{g} = [g_k]^T$ and $\bar{n} = [n_n]^T$ be the channel impulse response and AWGN respectively. Define the input matrix as $\bar{Y} = diag(\bar{X})$ and the DFT matrix as

$$\bar{\bar{F}} = \begin{bmatrix} W_N^{00} & \dots & W_N^{0(N-1)} \\ \vdots & \ddots & \vdots \\ W_N^{(N-1)0} & \dots & W_N^{(N-1)(N-1)} \end{bmatrix}$$

where $W_N^{ik} = \left(\frac{1}{\sqrt{N}}\right)^{-j2\pi\left(\frac{ik}{N}\right)}$. Also define $\bar{H} = DFT_N(\bar{g}) = \bar{\bar{F}}\bar{g}$ and $\bar{N} = DFT_N(\bar{n}) = \bar{\bar{F}}\bar{n}$. Under the assumption that noise and channel are uncorrelated and in absence of ISI we can write the following,

$$\bar{Y} = DFT_N(IDFT_N(\bar{X}) * \bar{g} + \bar{n}) = \bar{\bar{X}}.\bar{H} + \bar{N}$$

This equation demonstrates that OFDM systems can be viewed as a 2D lattice in time and frequency plane. Pilot subcarriers are inserted according to a pre-determined order which is used to estimate the channel. The channel estimate for the intermediate subcarriers can be obtained by interpolation between the pilots. Pilot arrangement are of two type, [a] Block Type and [b] Comb Type as shown in fig. 3.20.

Pilots are BPSK modulated symbols placed at regular intervals in time-frequency plane. In block type transmission all the N subcarriers are sent as pilots and this pilot OFDM symbols are sent at periodic intervals in time. Whereas in comb type arrangement each OFDM symbol has pilot symbols at periodic frequency bins. The block type arrangement is useful in slow fading channel whereas the comb type is useful in capturing channel characteristics in a fast fading channel. In 802.11a/g comb type pilot arrangements are used and therefore in this work we have implemented the equalization of OFDM symbols involving comb

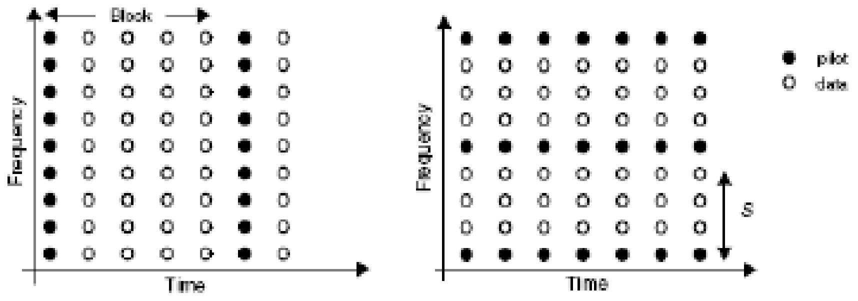


Figure 3.20: Pilot arrangement in OFDM

type pilots. Equalization using pilots is largely dependent on the number of pilots in an OFDM symbol. For 802.11a/g four pilot tones are inserted in subcarriers [-21 -7 7 21] and are used to estimates the channel. Let there be N_p number of pilot in one OFDM symbol and are uniformly inserted in with S subcarriers apart, where $S = 14$. The receiver knows the pilot locations $\bar{P} = [P_k]^T$, (k = 0,. . . ,$N_P - 1$), the pilot values $\bar{X}^P = [P_k]^T$ (k = 0,. . . ,$N_P - 1$) and the received signal $\bar{Y}$. Given these information a Least Squares estimate can be made at the pilot location given by,

$$\hat{H}^p_{LS} = \left[\frac{Y(P_0)}{X^p_0} \dots \frac{Y(P_{N_p-1})}{X^p_{N_p-1}}\right]^T$$

The task here is to estimate the channel condition at the data subcarriers given the LS estimates at the pilot subcarriers, $\hat{H}^p_{LS}$ and the received signal $\bar{Y}$. 1D interpolation techniques can be used to estimate the channel at the data subcarriers. 1D interpolation are of following types,

[a] Linear Interpolation: This is a piece-wise linear interpolation where the channel estimate between two pilots is given by :

$$\hat{H}(kS + t) = \hat{H}^p_{LS}(k) + \left[\hat{H}^p_{LS}(k + 1) - \hat{H}^p_{LS}(k)\right] .\frac{t}{s}, \quad 0 \leq t \leq S$$

Linear interpolation is easy to implement in hardware as it requires one accumulator to perform the task.

[b] Second order interpolation: This technique uses three pilots instead of two as in the linear interpolation case. It is a second order polynomial curve fitting technique used to approximate the channel estimates at the data subcarriers given the information at pilot subcarriers. Linear combinations of three

pilot estimates gives better interpolation results.

$$H_e(k) = H_e(mL + l) \quad 0 \le l \le L \tag{3.5.10}$$

$$= c_1 H_p(m-1) + c_0 H_p(m) + c_{-1} H_p(m+1) \tag{3.5.11}$$

$$where, \left\{ \begin{array}{c} c_1 = \dfrac{\alpha(\alpha-1)}{2} \\ c_0 = -(\alpha-1)(\alpha+1) \\ c_{-1} = \dfrac{\alpha(\alpha+1)}{2} \end{array} \right\}, \quad \alpha = \frac{l}{N}$$

It is to be noted that since 52 subcarriers are used instead of the total available 256, this technique cannot be estimated for subcarriers [0:-7] and [21:26] because during these intervals we do not have three pilots to do a complete non-linear estimation. Resorting to linear interpolation can be a solution in this case. Also because of this discontinuity of data carriers the channel estimate of two adjacent OFDM symbol are independent and as a result we cannot use the last pilot of the previous symbol to do the interpolation of subcarriers [0:-7] of the next symbol. Using 256 subcarrier will allow us to interpolate between two consecutive OFDM symbol and also allow us to use non-linear interpolations as well.

[c] Spline Interpolation: Spline interpolation uses higher order polynomial to obtain the best fit given the pilot carriers. This technique finds the solution of a system of linear equation of the form Y = AX where the matrices have orders in multiples of N (DFT size). As a result this algorithm is hard to implement in hardware due to immense complexity involved in computing the inverse of a matrix.

For hardware implementation, a linear interpolation equalizer have been designed and incorporated in FPGA as shown in fig. 3.21 The interpolation block is shown in fig.3.22. This block is responsible for computing the channel estimates of the data subcarriers using linear interpolation. The constant multiplier multiplies the difference between two pilots by the 1/S = 1/14 and the accumulator accumulates the results for every t$\in$ $[1 : S]$ and adds to the previous pilot. The output provides the channel estimates for all the data subcarriers. Once the channel estimates of the subcarriers are obtained at the output of the interpolator the

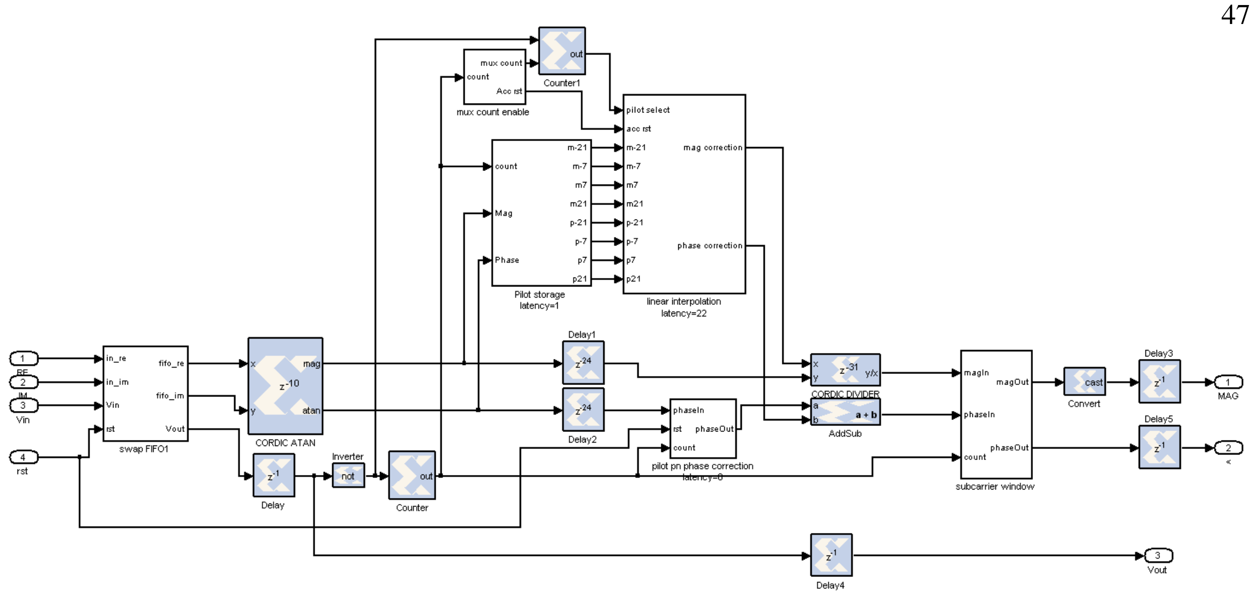


Figure 3.21: Linear interpolation equalizer

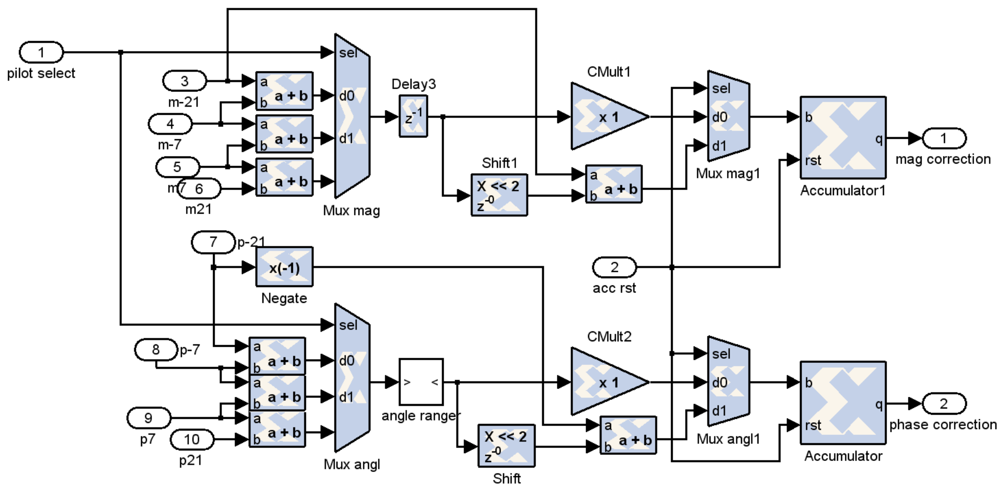


Figure 3.22: Interpolator using channel estimates at pilot locations

received subcarriers $\bar{Y}(k)$ are corrected as follows,

$$angle\left[\hat{X}(k)\right] = angle\left[Y(k)\right] - angle\left[H_e(k)\right] \tag{3.5.12}$$

$$mag\left[\hat{X}(k)\right] = \frac{mag\left[Y(k)\right]}{angle\left[H_e(k)\right]} \tag{3.5.13}$$

Where $H_e(k)$ are the channel estimates obtained from the linear interpolator block. The I and Q signals are computed from the magnitude and angle as follows,

$$I_{corrected} = mag\left[\hat{X}(k)\right].\cos(angle[\hat{X}(k)]) \tag{3.5.14}$$

$$Q_{corrected} = mag\left[\hat{X}(k)\right].\sin(angle[\hat{X}(k)]) \tag{3.5.15}$$

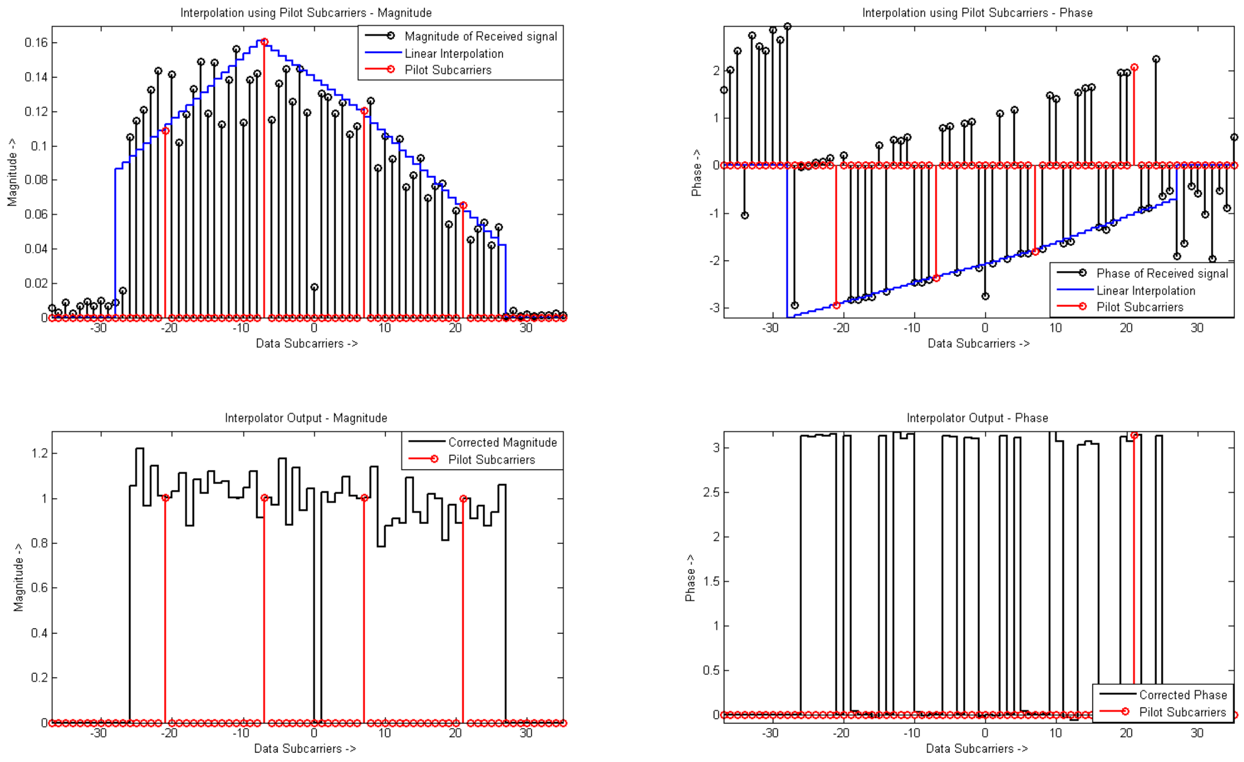
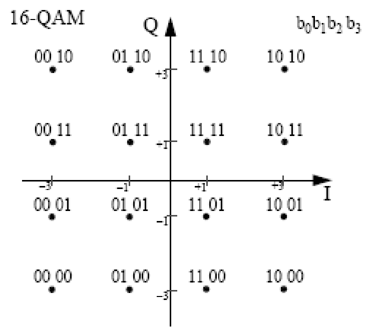


Figure 3.23: Linear interpolation for phase and magnitude correction

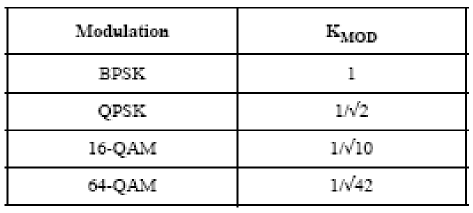
The linear interpolation plots in 3.23 show how the pilots are used to estimate the channel transfer function. Separate interpolation has been done for phase and magnitude. Fig. 3.23 shows the magnitude and phase interpolation of a BPSK modulated OFDM symbol. The red colored samples shows the pilots which has been changed by the time varying frequency selective channel. With the prior knowledge of the pilots we know that the magnitude of the pilots as transmitted is “1” for all pilots and phase of the pilots are $[0, 0, 0, \pi]$. The objective of the equalizer at the receiver is to restore the magnitude and phase of the pilots and assume that the channel in between the pilots is linear, so that the subcarriers can be equalized by linear interpolation. Fig. 3.23 also show that the magnitude and phase of the pilots are restored to their original transmitted values. The uneven magnitude is largely due to the precision of the CORDIC arctan unit and also due to limited precision in the fixed point design. The performance of the equalizer is shown in §3.6 where we present the constellation plots for actual 802.11a/g Wi-Fi packets.

3.5.6 Demodulation

The demodulator follows the equalizer. The equalizer de-rotates the constellation and restores the constellation to its original configuration. The demodulator is a maximum likelihood demodulator which performs threshold test as specified in 802.11a/g specification [79]. Fig. 3.24(a) shows the constellation map for 16QAM modulation. The symbol energy has been normalized using a normalization factor as given in Fig. 3.24(b). Decision boundaries are given by the perpendicular bisector of the line joining the two symbols. This gives the optimum decoder as it minimizes the Euclidean distance between received signal and the nearest constellation point.



(a) 16QAM constellation



Modulation	K_{MOD}
BPSK	1
QPSK	$1/\sqrt{2}$
16-QAM	$1/\sqrt{10}$
64-QAM	$1/\sqrt{42}$

(b) Scaling factors for decision boundaries

Figure 3.24: Decision boundaries

It is to be noted that the right half of the constellation is the same left half, except for the sign bit b_0. Therefore instead of performing the threshold for 16 constellation points we utilize the symmetry in the constellation to demodulate using two thresholds separately on the I and Q channel. The output of the demodulator is always a 6 bit number which is the maximum no. of bits per subcarrier for a 64QAM modulation. The proper number information bits are extracted in the deinterleaver ensuring that only the information bits are extracted at the deinterleaver. Figure 3.25 shows the implementation of the baseband demodulator using Matlab M-Code.

```
function OutData = decodeData(Nbpsc, In_I, In_Q)
    persistent kmod, kmod = xl_state([1.0, 0.707106781186547, ...
            0.316227766016838, 0.154303349962092], {xlSigned,16,15});
    b1 = 0; b2 = 0; b3 = 0; b4 = 0; b5 = 0;
    n_bits = xl_nbits(In_I) - 1;
    msb_i = xfix({xlUnsigned, 1, 0}, xl_slice(In_I,n_bits,n_bits));
    msb_q = xfix({xlUnsigned, 1, 0}, xl_slice(In_Q,n_bits,n_bits));
    b0 = xl_not(msb_i);
    abs_I = In_I;
    abs_Q = In_Q;
    if In_I < 0
        abs_I = -In_I;
    end
    if In_Q < 0
        abs_Q = -In_Q;
    end
    switch Nbpsc
        case 2
            b1 = xl_not(msb_q);
        case 4
            b2 = xl_not(msb_q);
            if abs_I < 2*kmod(2)
                b1 = 1;
            end
            if abs_Q < 2*kmod(2)
                    b3 = 1;
            end
        case 6
            b3 = xl_not(msb_q);
            if abs_I < 4*kmod(3)
                b1 = 1;
            end
            if abs_Q < 4*kmod(3)
                    b4 = 1;
            end
            if abs_I >= 2*kmod(3) && abs_I < 6*kmod(3)
                    b2 = 1;
            end
            if abs_Q >= 2*kmod(3) && abs_Q < 6*kmod(3)
                b5 = 1;
            end
    end
        demod = xl_concat(b5, b4, b3, b2, b1, b0);
    OutData = xfix({xlUnsigned, 6, 0}, demod);
end
```

Figure 3.25: M-Code implementation of the demodulator

The demodulator has a feedback from the signal symbol decoder. The signal symbol contains the information about the modulation and coding rate of the OFDM payload that follows the signal symbol. The parameter called N_{bpsc} has the information of the number of bits per subcarrier, which also specifies the baseband modulation type. N_{bpsc} is used as a control signal select the appropriate decision boundaries and ensure correct demodulation of the bits. Similar techniques have been employed for higher constellations like 64QAM as well, only with the addition that the number of threshold test have been doubled when compared to that of a 16QAM. In this design the demodulator is capable of decoding BPSK, QPSK, 16QAM and 64QAM signals.

3.5.7 De-interleaver

The interleaver is a block interleaver with a block size corresponding to the number of information bits in a single OFDM symbol denoted by N_{CBPS}. The interleaver is defined by a two-step permutation. The first permutation ensures that adjacent coded bits are mapped onto nonadjacent subcarriers. The second ensures that adjacent coded bits are mapped alternately onto less and more significant bits of the constellation and, thereby, long runs of low reliability (LSB) bits are avoided. Like the transmitter the deinterleaving at the receiver is also governed by two permutation. At the receiver let j denote the index of the received signal and i denotes the index after the first interleaving and k denotes the index after the second permutation then the permutation equations are defined as

The first permutation,

$$i = s \times floor\left[\frac{j}{s}\right] + \left[j + floor(16 \times \frac{j}{N_{CBPS}})\right] mod(s) \quad j = 0, 1, \ldots, N_{BPSC} - 1 \tag{3.5.16}$$

$$where \quad s = max\left[\frac{N_{BPSC}}{2}, 1\right] \tag{3.5.17}$$

This permutation is just the inverse of the second permutation at the transmitter. The second permutation is defined as

$$k = 16 \times i - [N_{CBPS} - 1]\, floor\left[16 \times \frac{i}{N_{CBPS}}\right] \quad i = 0, 2, \ldots, N_{CBPS} - 1$$

The implementation is done using M-Code which is a combination of slicers that rearranges the ordering of the bits based on eq.(3.5.16) and eq.(3.5.7) and then concatenated after rearranging. The indexing map can be predetermined and can be hardcoded into the receiver and only needs to be changed when the interleaver at the transmitter changes.

3.5.8 De-puncture

De-puncturing is the process of re-inserting the bits that were stolen at the transmitter to increase the coding rate. The de-puncture block is also implemented using M-Code which inserts the dummy bits at appropriate indices.

3.5.9 Viterbi Decoder

The Viterbi Decoder uses a trellis based decoding. The convolution encoder at the transmitter uses $[133 \quad 171]_8$ as the polynomial to produce a $1/2$ rate convolutional code with constraint length 7. The receiver employs a trellis based maximum likelihood Viterbi decoder which decodes the input bits to obtain the information bits. The trellis length is chosen to be 5 times the constraint length. For this implementation, an open source verilog implementation of the trellis based Viterbi decoder [88] has been used. It has been incorporated into the transceiver design as a *black box* along with proper interface ports to connect to other adjoining blocks. Figure 3.26 shows the complete implementation of the deinterleaver, depuncture and Viterbi decoder using system generator blocks along with the control signals.

3.5.10 De-scrambler

The descrambler is the inverse of the scrambling operation as mentioned in the specification [79]. The same scrambler unit is used to scramble and descramble the data. Scramblers are implemented as shift registers with appropriate taps that are XOR-ed together to give a pseudo-random sequence of a fixed length. The scrambler figure 3.27 shows the scrambler unit

The *signal* symbol is not scrambled whereas all subsequent symbols has to go through the descrambler. Thus a control signal is required called *signal-done* to multiplex between the unscrambled signal

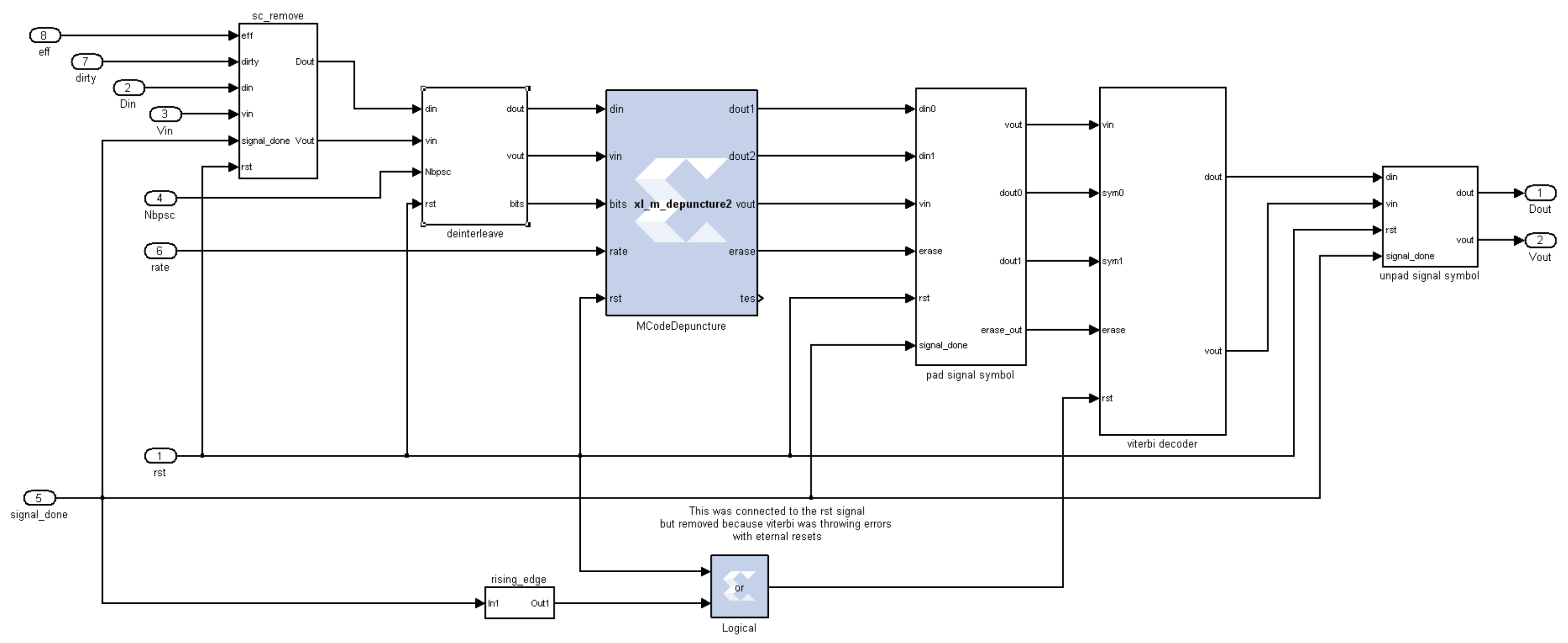


Figure 3.26: De-interleaver, De-puncture and Viterbi decoder

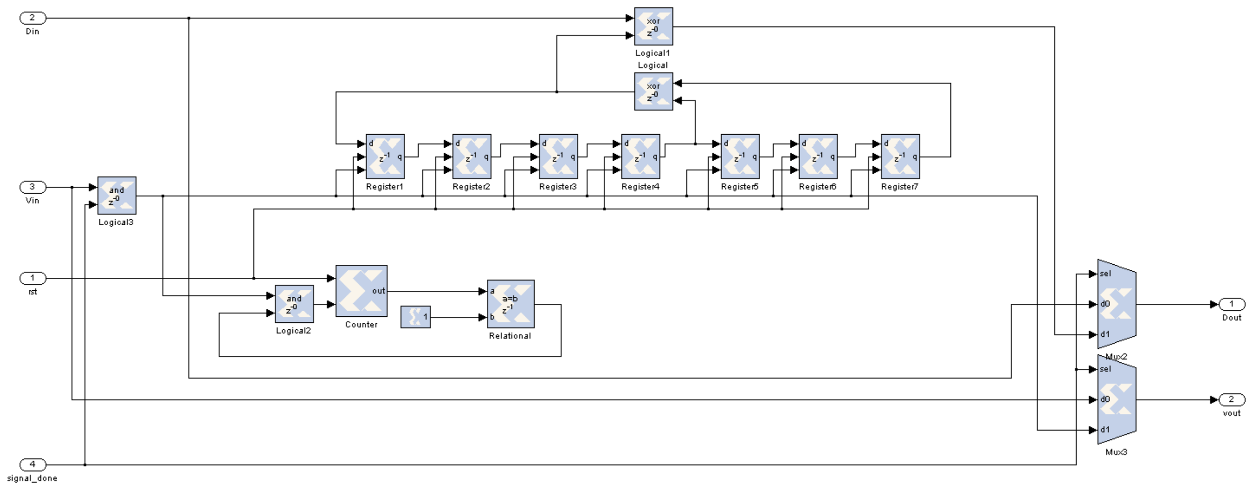


Figure 3.27: De-scrambler

symbol bits and the descrambled bits of the subsequent symbols.

3.5.11 Signal Symbol Decoder

The *signal* symbol is the first symbol in the physical layer payload. The signal symbol contains all the information required to decode the remaining packet. The signal symbol is composed of 24 bits and contains the following fields as specified in [79]: Rate - bits 0 - 3; Reserved bit - bit 4; Length - bits 5 - 16 with bit 16 being the MSB; Parity bit - bit 17; Signal Tail - bits 18 - 23, used to initialize the descrambler at the receiver.

The *signal* symbol decoder is implemented by a 24 stage shift register. The outputs corresponding to the individual fields mentioned above are latched using a suitable control signal. Figure3.28 shows the implementation. The decoder is a set of 24 register units connected in a chain which is controlled by a clock that counts from 0 to 23. At the end of the 23^{rd} count the outputs are latched as shown by the output ports.

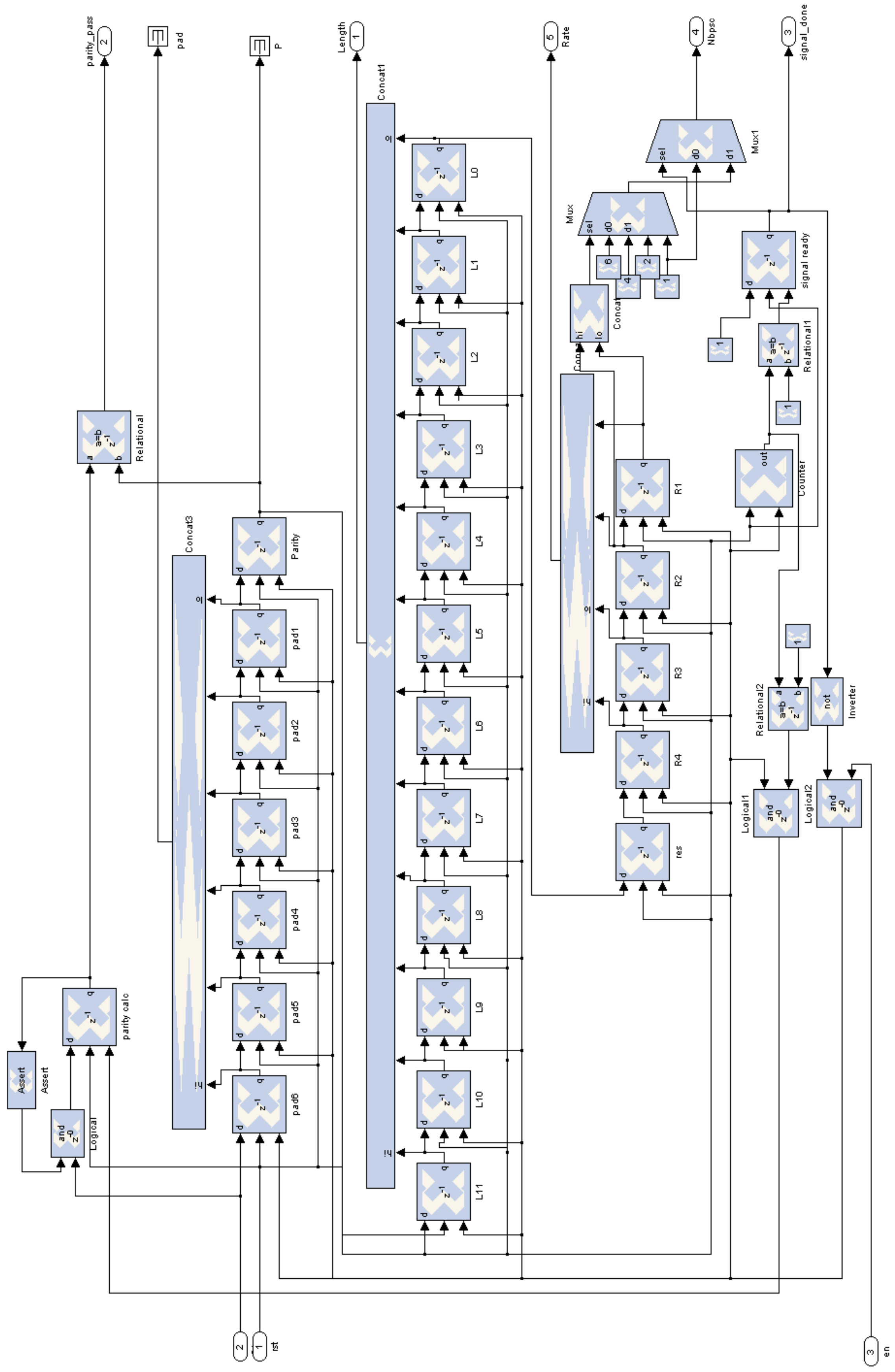


Figure 3.28: Signal symbol decoder

3.6 Performance Evaluation

3.6.1 Hardware Utilization

The OFDM transceiver has been implemented on a Xilinx Virtex V FPGA. The baseband design is done using the Xilinx System Generator design tool. The top-level design consists of clock generation unit for the four clock domains in the transceiver along with instantiation of the OFDM transceiver, the Ethernet transceiver and the SPI serial interface. It also generates the clock signals for the DACs/ADCs. The hardware utilization of the various subsystems in the radio is reported in this section. The baseband subsystem comprising the OFDM transmitter and receiver occupies most of the FPGA resources. The transceiver utilization also included the synchronization logic used to transfer data across the various clock domains in form of asynchronous shared buffers and registers. Table 3.2 shows the utilization for the entire radio system including the user I/O and SPI interface to program the peripheral boards as shown in the top-level schematic in figure 3.4.

Module	Slices	LUTs	BRAM	DSP48s
OFDM Transceiver	10333(59.80)	22704(32.85)	57(44.53)	64(100.00)
OFDM Transmitter	2796(16.18)	5594(8.09)	32(25.00)	15(23.44)
OFDM Receiver	7188(41.60)	16589(24.00)	12(9.38)	48(75.00)
Ethernet Transmitter	243(1.41)	295(0.43)	1(0.78)	0(0.00)
Ethernet Receiver	151(0.87)	270(0.39)	0(0.00)	0(0.00)
SPI Serial Interface	56(0.32)	76(0.11)	0(0.00)	0(0.00)
DAC Output Stage	27(0.16)	39(0.06)	0(0.00)	0(0.00)

Numbers in parenthesis indicate percentage utilization

Table 3.2: Hardware utilization of the complete radio system

The utilization of the transmitter is shown in 8.2, which is based on previous work [74]. The binary domain computations are performed in software and the FPGA is responsible for the sample domain computations only.

The receiver is entirely implemented on the FPGA. The receiver also includes DC offset removal before decoding. Table 8.3. Apart from the core baseband processing subsystems, the receiver includes few utility subsystems: 1) Channel states: This unit provides a quantized output of the FFT result which reflects the frequency domain channel state across the OFDM bandwidth. This information can be used by the

Module	Slices	LUTs	BRAM	DSP48s
InMux	109(0.63)	247(0.36)	3(2.34)	0(0.00)
TxController	120(0.69)	280(0.41)	5(3.91)	0(0.00)
Mod	57(0.33)	97(0.14)	0(0.00)	0(0.00)
IFFT	1349(7.81)	2794(4.04)	4(3.13)	9(14.06)
InsertGuard	245(1.42)	616(0.89)	6(4.69)	0(0.00)

Numbers in parenthesis indicate percentage utilization

Table 3.3: Hardware utilization of the transmitter subsystem

higher layers to design rate adaptation algorithms and obtain finer control over the transmission parameters. 2) Signal to Noise Ratio: In order to use this radio prototype for practical experiments and research, signal to noise ratio of the received signal is very important. 3) MAC CRC checker: This unit helps to check if the packet is received with proper MAC CRC. This helps is reducing latency by eliminating data transfer between the host and the radio which makes it more practical for fast measurements.

Module	Slices	LUTs	BRAM	DSP48s
Packet Detect	536(3.10)	1805(2.61)	0(0.00)	12(18.75)
Correlator	1816(10.51)	2884(4.17)	0(0.00)	2(3.13)
FFT	875(5.06)	1745(2.52)	5(3.91)	30(46.88)
Equalizer[1]	1197(6.93)	3356(4.86)	0(0.00)	4(6.25)
Demodulator	36(0.21)	100(0.14)	0(0.00)	0(0.00)
Decoder[2]	1319(7.63)	2644(3.83)	3(2.34)	0(0.00)
Channel State	98(0.57)	140(0.20)	1(0.78)	0(0.00)
SNR	944(5.46)	3356(4.86)	0(0.00)	0(0.00)
CRC check	114(0.66)	129(0.19)	0(0.00)	0(0.00)

Numbers in parenthesis indicate percentage utilization

[1] Includes channel estimation, [2] Includes De-interleaver and Viterbi decoder

Table 3.4: Hardware utilization of the receiver

3.6.2 Error performance

The OFDM transceiver supports all the data rates as specified in the 802.11 specification. We report the theoretical and experimental packet error rates in table 3.5.

Table 3.5: Throughput and SNR requirements for 802.11a/g data rates

Mod	CR[1]	Data Rate Mbps	SNR(dB)	
			Th[2]	Exp
BPSK	1/2	6	3.0	4.5
BPSK	3/4	9	5.0	6.0
QPSK	1/2	12	6.0	7.0
QPSK	3/4	18	8.0	9.0
16QAM	1/2	24	12.5	13.0
16QAM	3/4	36	17.0	18.0
64QAM	2/3	48	19.5	24.0
64QAM	3/4	54	21.0	26.0

[1]Coding Rate, [2]Theoretical

3.6.3 Latency

Latency in the transceiver is primarily attributed to the processing delays of the various signal processing subsystems in the transceiver. Transceiver latency plays an important role in our implementation. It is required to determine the turnaround time for the receiver at the broadcast node. Usually for any practical transceiver, the minimum time that is required for the MAC/PHY to receive the last symbol of a frame at the air interface process the frame and respond with the first symbol on the air interface of the response frame is of great interest. This includes receiver side PHY layer processing delay + MAC processing delay + Transmitter side processing delay + PCI transfer delay for both Rx and Tx + Front-end radio hardware delay. If we disregard the MAC processing delay and the PCI transfer delay then we can summarize the following:

(1) Receiver side:

Difference between the last symbol received at the air interface to last bit transferred to host = 14.83μsec.

(2) Transmitter side:

Difference between the FIFO read signal to the first analog sample out from the DAC = 11.68 μsec.

(3) Key note: The FFT/IFFT module consumes the bulk of the latency = 7.4 μsec. x 2 (for Tx and Rx) = 14.8μsec.

It is observed that most clock cycles are consumed by the FFT/IFFT unit and other than that the latency is attributed largely to various buffering elements required for proper functioning of the pipeline. in order to further reduce latency we need to use better pipelined cores with faster cycle times. This is purely a limitation of our prototyping hardware, and not of the method–any commercial WiFi chipset is already capable of the processing needed to implement our technique.

Chapter 4

Intelligent Physical Layers: Specification for a SDCR

In this chapter we present an example implementation of a CR that is built as an extension to the 802.11a/g physical layer. In our prior work [75] we have successfully implemented a functioning 802.11a/g physical layer on a FPGA based prototype platform as described in chapter 3, that is fully compatible with commodity Wi-Fi devices. Extending 802.11a/g for cognitive radio application is useful because of the ability to use Wi-Fi like protocols in fractured spectrum using DSA. We can get the high datarate of 802.11a/g while making use of the spectral opportunity provided by lower utilization of the spectrum by certain licensed users, like the television broadcast stations. While transmission in non-contiguous spectrum, often in multiple disjoint chunks spread apart in frequency can be achieved with relative ease, receiving such waveforms greatly complicates the tasks facing the radio — for example: correlators, used to determine if a signal is in transmission, now must look up across multiple bands instead on one continuous range of frequencies [61]. Thus, it is important to take a new look at the radio PHY and assess the requirements for next generation radio architecture. We discusses the challenges involved in designing such radio PHY that is adaptable, fast and easy to reconfigure and backward compatible with legacy systems using OFDM.

The evolution of processing for wireless networking, particularly for **Software Defined Cognitive Radios (SDCRs)**, is approaching a design choice similar to Internet routers which has advanced substantially over time. Network processors used two ways to speed network processing. The first method, exemplified by the Intel IXP processors, was to develop general purpose processors with extensions and multiple specialized packet processing engines. Although those processors are intended for network processing tasks, they can be used for other purposes (e.g. storage processing). Another approach, adopted by systems such as the SiTerra/Vitesse Prism processor accelerate specific steps of the network processing pipeline, such as implementing a specialized route lookup mechanism. Recently, a generalized packet processing approach based on the OpenFlow model has been proposed, leading to a rethinking of forwarding hardware [89]. This model of forwarding uses more general "match logic" (**e.g.** TCAMs) coupled with general purpose processors for populating that hardware. The same trend of using a general purpose processor, followed by specialized hardware and then evolving into some general purpose processor coupled with hardware acceleration is common in many computing domains and reflect both the increase in available silicon for special purpose applications with analysis to determine the "kernel" of specific domains.

Radio processing was originally done on digital hardware that was implemented using a combination of general purpose DSPs or fixed-function logic implemented as an ASIC or using an FPGA. Over time, specialized processor designs have evolved that are finely tuned for handling a set of wireless protocols [12, 11]. However, these processors are mainly suitable for "3G" networks, and it's not clear they are easily adaptable to "4G" or emerging radio standards. Alternative architectures, such as the PicoChip processor[21] and other similar designs like SCA and XiRisc [90, 91] combine general purpose processors with fixed-function logic designed to provide more efficient solutions to specific tasks (**e.g.** correlators for determining if a packet is being received). The most challenging problem in the domain of cognitive radio is supporting non-contiguous spectrum access while remaining backward compatible and these architectures do not provide enough flexibility to adopt this shifting paradigm.

Therefore, we survey some of the techniques that we believe cognitive radios will need to implement and highlight the impact of those techniques on the underlying architecture. Our current implementation is at the point of special purpose functional blocks implemented using an FPGA that form the foundation of $3G$ and $4G$ wireless protocols and are easily reconfigurable at run-time. The contribution of this work is a review of the structure of processing steps needed in cognitive radios rather than a final general purpose design for those processing steps.

4.1 SDCR : Redefining the Radio PHY

The similarity among different communication protocols is also reflected in their corresponding physical layer. There is an increasing demand to redesign the common processing engines to perform most of the functions in a fast changing environment of cognitive radio. A close look at the current wireless protocols reveals that we can define a more fundamental set of operations or primitives, beyond just the the parameters or functional operation of a particular transceiver subsystem: e.g., instead of having correlators with fixed coefficients, we should have a method to feed the coefficients required for a particular packet encoded using a particular protocol. This is typically required in a cognitive radio environment where the available spectrum varies over time and so does the number of available OFDM subcarriers which changes the time domain correlation coefficients [48, 49]. With various concurrent wireless protocols in mind we define a set of fundamental operations, a generic transceiver should have in order to operate as a SDCR. This de-construction of the radio physical layer beyond a multi-mode type operation is motivated by a substantial amount of prior research and publication in the cognitive radio community. We have identified the requirement for a NC-OFDM based SDCR in chapter 2. However, for this example, we intentionally omit the bit-level processing engines such as error correction algorithms, scrambling-descrambling, interleave - deinterleave, because these bit level operations could be done with similar efficiency in software [92, 70, 4] as a part of the higher layer of the SDCR.

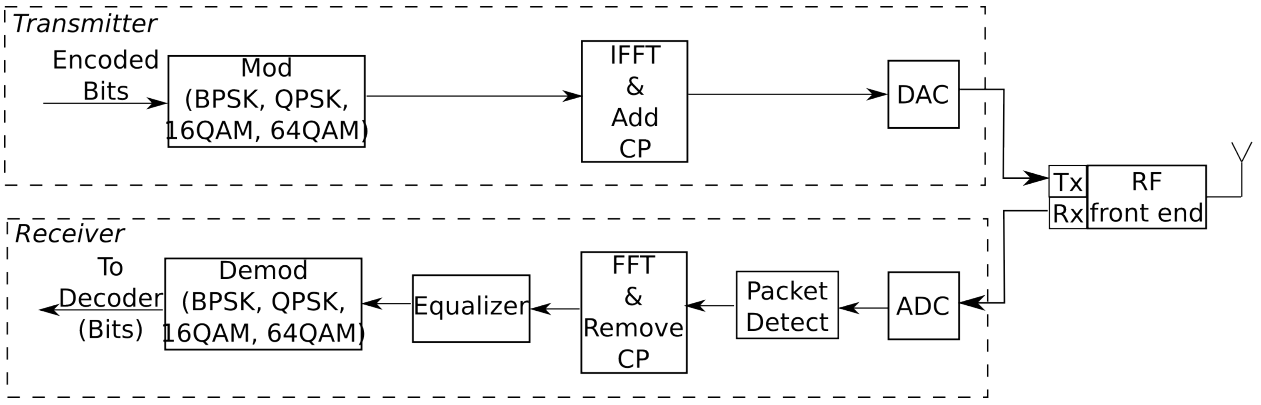


Figure 4.1: OFDM transceiver subsystems

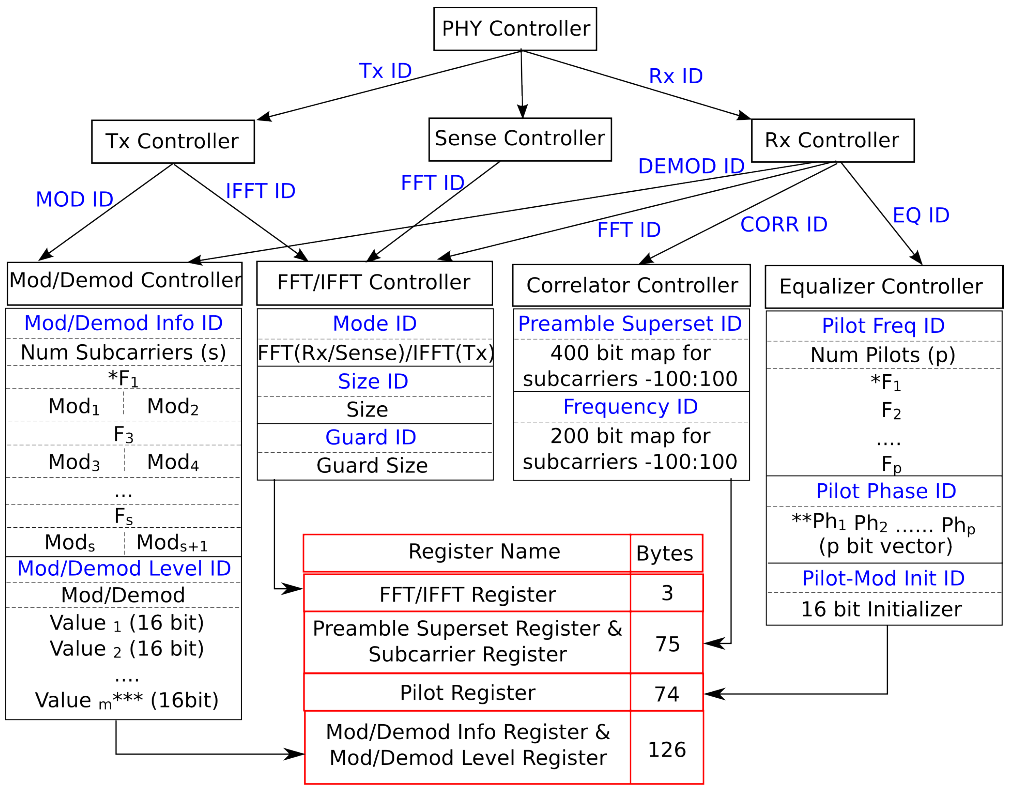
In the following sections we examine the basic OFDM transceiver pipeline, shown in Figure 3.2

and identify the primitives that are fundamental to operate in a cognitive network environment. We build our system level architecture based on the basic Wi-Fi (802.11a/g) transceiver [74, 75]. Once the basic subsystems are in place, we go about to decide on the additional functionality required for a SDCR by adding programmable interfaces and re-designing the subsystems for optimum use. The primitives also allow us to identify a suitable programming interface that can reconfigure the physical layer hardware to adapt to its environment at run-time. This is particularly important in a Cognitive Radio network because with the changing environment, the transmission and reception parameters can change within a short period of time, often in order of microseconds. In this example we limit our discussion to the baseband processing elements called kernels which are able to perform the required functionality without any knowledge of the characteristics of its input samples. Once the **Cognitive Engine** sets the operation parameters for the kernels the transceiver pipeline processes the packets as a pipeline that appears as a black box to the user. It is only the configuration parameters that are required to be modified at runtime while the basic processing pipeline remains unchanged. This makes our proposed design specially suitable for SDCR architecture.

4.1.1 OFDM Transceiver : The Top Level

One of the most important things in a SDCR is programmability and how fast the radio PHY can adapt itself to the changing environment. This requires: 1) barebone kernels broken down to the most fundamental operations, making them more programmable and avoid being always multimode and 2) reconfiguration of the PHY at run-time without going through the compilation process. The system has a hybrid architecture employing an FPGA and software running on the host. The FPGA has all the signal domain processing while the host controls the bit-level processing. Although our representative system is implemented using an FPGA, the goal of this work is to determine an overall **architecture** for SDCR processors.

Figure 4.2 shows a hierarchical design of the control path for an OFDM based SDCR transceiver. The PHY Controller is the interface between the MAC or the **Cognitive Engine** and the underlying hardware. For a typical cognitive radio environment the PHY Controller can be in one of the following states : **Transmit**, **Receive** or **Sense**. As shown in Figure 4.1, the transmit and receive path elements are made functional based on mode selection. In **Sense** mode, only the FFT block is functional. It performs FFTs on incoming



** F_i = Subcarrier Index (-100:100).*
*** Ph_i = Phase of Pilot at subcarrier F_i. Values can be 0(Phase=0) or 1(Phase=π).*
**** The value of m depends on Modulation Type. m=1(BPSK), 2(QPSK), 4(16QAM) or 6(64QAM).*

Figure 4.2: PHY controller

signals and sends the output to the **Cognitive Engine**. In our implementation, the PHY Controller can handle commands from the host or the **Cognitive Engine** in the form of short messages called *control packets*. Every control packet has a kernel ID followed by the number of bytes of message included for that message, followed by a set of parameters to program that particular kernel register. The state machine at the Transmitter(Tx), Receiver(Rx) or Sense Controller level can decode and process any combination of kernel IDs in any order and forwards the information to the respective controller. The respective controllers extract the information and update the hardware registers that are used by each of the subsystems. This facilitates reconfiguration of the kernels within a very short time.

We have designed four controllers, a) FFT/IFFT Controller, b) Mod/Demod Controller, c) Correlator Controller, and d) Equalizer Controller. Since FFT and IFFT have exactly opposite functionality in receiver/sensing and transmitter chain, but similar reconfiguration requirements, we merged these two modules into one FFT/IFFT Controller. Also, the Modulator(Mod) and Demodulator(Demod) components have the same reconfiguration parameters. We combined these together into a single Mod/Demod Controller. Both the FFT/IFFT and mod/demod controllers can handle packets from both Tx and Rx Controllers. Additionally, the FFT/IFFT Controller can parse **control packets** from the Sense Controller as well. A single FFT block is used in the hardware, which is either set in FFT mode for receiving/sensing, or in IFFT mode for transmitting. The "Mode ID" in the FFT/IFFT Controller denotes the mode: Transmit, Receive or Sense. The other two programmable parameters of FFT/IFFT are size and guard, which has been introduced as tunable parameters in this level. FFT/IFFT Controller updates these information in **FFT/IFFT Register**. Mode, size and guard size are the three 8-bit parts of this register, making it a 3-byte register.

The Mod/Demod controller is programmed using a variable length list of configuration information for each subcarrier. An 8-bit subcarrier information (F_i) is inserted, followed by 4-bit modulation type of frequencies F_i and F_{i+1}. The modulation type can be BPSK, QPSK, 16QAM or 64QAM. We use the fourth bit of the modulation type field to denote whether this subcarrier should be modulated or demodulated. Using this information, the cognitive engine can suppress some of the subcarriers and transmit or receive in some others. This design helps to keep the size of the control packet smaller, and thus less time is required to parse the control packet and update the registers. 'Mod/Demod Level ID' is followed by the modulation type (BPSK, QPSK, 16QAM or 64QAM), followed by 16-bit values of the levels of that modulation. The number of preceding values depend on the number of modulation types. Mod/Demod info is stored in **Mod/Demod Info Register**, where 4-bit modulation type is stored for each subcarrier. Since our representative design currently handles 200 subcarriers, this register is 800 bits or 100 bytes. Mod/Demod Level information is stored in **Mod/Demod Level Register**. The maximum number of entries for values depends on the modulation type, which is 1, 2, 4, or 6 for BPSK, QPSK, 16QAM and 64QAM respectively. Hence, a total of $(1 + 2 + 4 + 6) = 13$ 16-bit values can be stored at any time for all the possible modulations, which makes the register size to be $13 \times 16 = 208$ bits, or 26 bytes.

The Correlator Controller handles packets only from the Rx Controller and is used to program the Correlator block of the receiver. The state machine at this controller can parse 'Preamble Superset ID' and 'Frequency ID' messages. We allow the engine to program a superset of preambles of all the frequencies, and then allow only a few to be used at any point in time. In a cognitive radio environment, it is likely that the superset of preambles will change less frequently than the actual frequencies to be used for correlation. This design helps the engine to send fewer bits to reprogram the frequency set by keeping the superset constant. We assume that preamble can be modulated either in BPSK or in QPSK, for which at most 2 bits of information is required per subcarrier. Our design supports 200 frequencies, and so 400 bits of information will suffice to encode the preamble. Hence, preamble superset consists of a 400 bit map for subcarriers $[-100 : 100]$. This superset contains all the frequencies that can possibly be used in a transmission. The Correlator Controller parses these information and updates the **Preamble Superset Register**, which is a 400 bit or 50 byte register. In a cognitive radio domain for a non-contiguous OFDM reception, only a subset of these frequencies will be used, which is given by 'Frequency ID', followed by a 200 bit map of subcarriers $[-100 : 100]$, where 1 denotes that subcarrier will be used in the preamble regeneration. This information is updated in the 200 bit or 25 byte **Subcarrier Register**.

The Equalizer Controller parses control packets from the Rx Controller and generates signals to modify the **Pilot Register**, which is used by the Equalizer block of the receiver chain. The pilot locations are denoted by 'Pilot Frequency ID' and the phase of pilots by 'Pilot Phase ID'. 'Pilot Frequency ID' is followed by number of pilots(p), followed by 8-bit frequency of the p pilots. We consider that phases of the pilots can be either 0 or π, such that we can encode the information in 1 bit. So, 'Pilot Phase ID' is followed by a p-bit vector of the phases of p pilots. Currently, the design can handle at most 32 pilots, compared to 8 pilots in a band of 200 subcarriers in WiMax. Often a Pseudo Random Binary Sequence(PRBS) is generated to modify the pilot phases in each symbol. Hence, an initializer is required to program the PRBS generator. We use a 16-bit initializer for this purpose, denoted by 'Pilot-Mod Init ID', compared to 8-bit initializer in 802.11 and 11-bit initializer for WiMax. Pilot frequency, phase and initializer information are stored together in the **Pilot Register**. This register can store 32 8-bit pilot frequencies, 32-bit pilot phases, and a 16-bit initializer, altogether the size of this register is $(32 \times (8 + 1) + 16) = 304$ bits or 38 bytes.

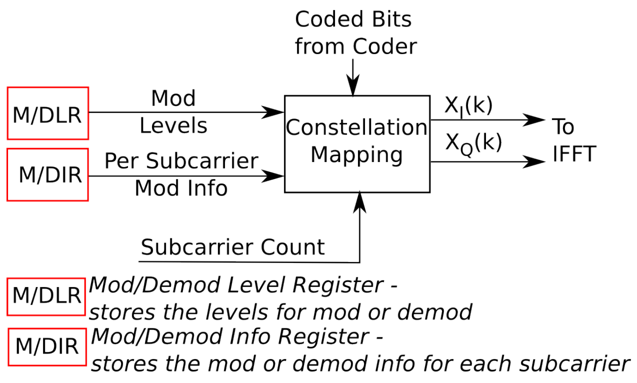

Coded Bits
from Coder
M/DLR
Mod
Levels
Constellation
Mapping
$X_I(k)$
$X_Q(k)$
To
IFFT
M/DIR
Per Subcarrier
Mod Info
Subcarrier Count
M/DLR
Mod/Demod Level Register -
stores the levels for mod or demod
M/DIR
Mod/Demod Info Register -
stores the mod or demod info for each subcarrier

Figure 4.3: Programmable modulator subsystem

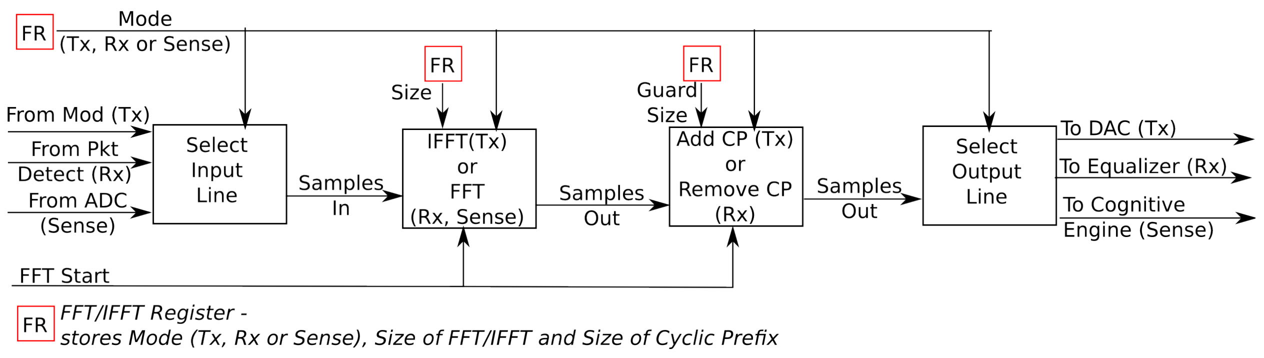


Figure 4.4: Programmable FFT or IFFT subsystem

4.1.2 Transmitter Kernels

The transmitter kernels are Modulator and IFFT with addition of Cyclic Prefix as shown in figure 3.2. The bit-level processing engines, like interleaver, scrambler, and encoder have been implemented in software.

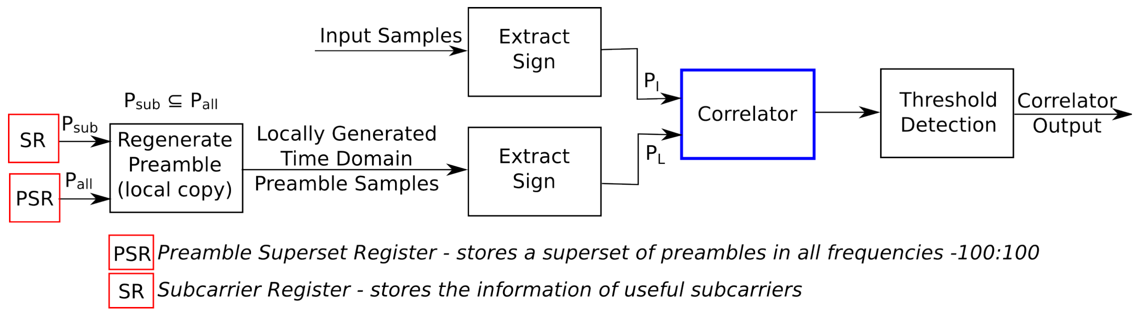
Modulator: The modulator is one of the programmable units implemented in the FPGA, that uses the information in 'Mod/Demod Info Register' and 'Mod/Demod Level Register'. Figure 4.3 shows a block diagram of the programmable modulator. A modulator block intakes coded bits and modulates them into complex samples of I and Q values, depending on the modulation type and modulation level. Advanced communication protocols such as Superposition Coding [55] and Hierarchical Encoding [56] techniques need different levels of modulation; this is implemented by having a "constellation mapping" table that is programmable rather than using fixed values as in a convention transceiver design. The constellation mapping is performed for each subcarrier. So, based on the Subcarrier Count, which is another input of the modulation block, the modulation type for that subcarrier is fetched from the 'Mod/Demod Info Register'. Based on this modulation type, the level for constellation mapping is fetched from 'Mod/Demod Level Register'. Then number of coded bits to be modulated is selected based on the modulation type, which can be 1, 2, 4 or 6 for BPSK, QPSK, 16QAM and 64QAM respectively. Based on these information, the I and Q values are generated by the Constellation Mapping block, which are fed into IFFT. If the 4^{th} bit of modulation type is set to 0 for any subcarrier, it indicates that this subcarrier will not be transmitted, and the Constellation Mapping block outputs 0 values for both I and Q samples.

Inverse Fourier Transform: After the bits are modulated into complex samples, they enter a common programmable FFT/IFFT kernel, used as transmitter, receiver or sensing mode. Figure 4.4 shows a block diagram of this generic platform and in this section, we will discuss how this block can be programmed for transmitter mode. The kernel uses information from 'FFT/IFFT Register' to get the mode. If the mode is set to transmitter, the Modulator output is selected as the input of the IFFT. Also, the FFT/IFFT block is set to IFFT mode. Based on the size information available in the register, the size of IFFT is chosen. Then, IFFT is performed on the modulator output. In transmitter, based on the Guard Size, the Cyclic Prefix is added to the samples. Finally, depending on the mode, output line is selected. For transmitter, samples are sent out to Digital to Analog Converters(DACs) which are connected to the frontend radio.

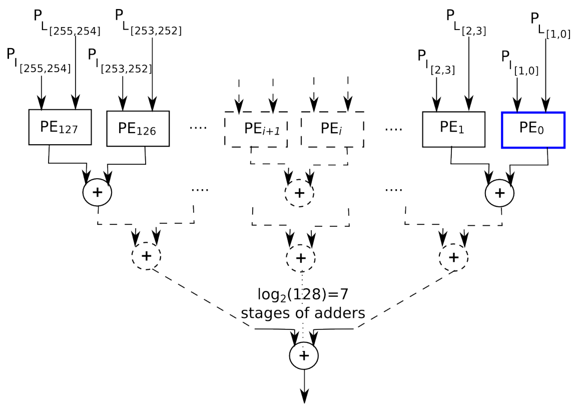
4.1.3 Receiver Kernels

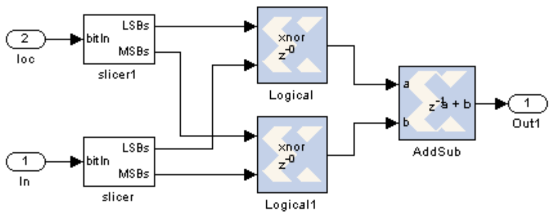
The receiver subsystems as shown in Figure 3.2 comprises of: Synchronizer or Packet Detect, FFT and Guard Removal, Equalizer and Demodulator. Apart from this there are other bit-level processing engines like de-interleaver, FEC decoder, CRC check and de-packetization to extract the information bits in the packet. As in the transmitter we exclude the bit-level processing from our design since they are efficiently performed in software at the host. §4.1.1 describes the control bits required to program all the kernels in the receiver. In this section we delve into the details of how the control bits are used to modify the kernels with changing environment.

Synchronizer and Packet Detect: The Synchronizer or the Packet Detect kernel is the entry point of the


Input Samples
Extract Sign
Extract Sign
P_I
P_L
Correlator
Threshold Detection
Correlator Output
$P_{sub} \subseteq P_{all}$
SR
PSR
P_{sub}
P_{all}
Regenerate Preamble (local copy)
Locally Generated Time Domain Preamble Samples
PSR Preamble Superset Register - stores a superset of preambles in all frequencies -100:100
SR Subcarrier Register - stores the information of useful subcarriers

(a) Functional diagram of correlator





(b) Inner structure of the correlator

(c) Smallest processing element of the correlator

Figure 4.5: Programmable correlator

baseband receiver. The primary job of the Synchronizer is to identify the boundary of a valid OFDM symbol. This is particularly important because it introduces phase noise into the signal which in turn makes the frequency domain decoding erroneous. In an OFDM based system, synchronization is typically done using a time domain correlation which searches for a pre-defined pattern called the preamble. Since in OFDM, the preamble is constructed and encoded in the frequency domain [79, 50], the time domain samples changes significantly with the encoding process, i.e., if the transmitter chooses to suppress a set of subcarriers then those subcarriers are not used to transmit the preamble. This makes the preamble quite different from the conventional preambles in wireless protocols that typically uses all the subcarriers supported by the protocol. Furthermore, the frequency domain data for the preamble varies from one protocol to the other. But fortunately they all employ the same basic technique to acquire synchronization.

Therefore, due to the non-contiguous modulation of the preamble and the variable nature of the frequency domain encoding, it is required to have a programmable Synchronizer. The time-domain correlator employs a running comparison with a local copy of the time-domain samples of the preamble being searched. Typically, the correlator size is of 64 samples. But depending on the sampling frequency and the FFT size this may vary. The basic operation in a correlator is shift-multiply-accumulate for every fixed-point complex sample in the correlator block. As the size of the correlator increases in our design to 256 with increasing FFT size, the number of multipliers and adders increases significantly to make the design too big for FPGA implementations. For a more efficient low-cost implementation we decompose the complex fixed-point correlator to a simple logic operation that eliminates the requirement of expensive multipliers and adders. The correlator has four key components as shown in figure 4.5(a):

- The **sign** bit of the I and Q samples is used instead of the actual values to eliminate large fixed-point (often as large as 16 bit) operation.
- The local copy of the preamble consists of the sign of the time domain signal being correlated with. In conventional designs the local copy is not programmable. To support cognitive radio applications, we need to program the coefficients. The **Preamble Superset Register** contains the complete set of frequency domain preamble for a particular protocol. The **Subcarrier Register** holds the set of subcarriers used for a particular reception. This information is obtained from the subcarrier detection or the spectrum sensing unit which is a standard kernel in a SDCR. Once the subcarrier information is obtained the time domain preamble should be regenerated either by using a look-up table or a dedicated low cost IFFT core. We leave the process of subcarrier detection of the preamble as future work. Once the time domain preamble is generated the **sign** bits of the I and Q components are used as the coefficients of the correlator.
- The correlator has two 256-bit shift registers which feed the coefficients to the core processing elements. Figure 5.9(b) shows the inner structure of the correlator. There are 128 Processing Elements (PE_i), each capable of performing comparison of two samples of the preamble as shown in figure 5.9(c). Since the objective of correlation is to search for the exact set of samples as stored in

the receiver, a simple *XNOR* operation is sufficient to compare the similarity between the incoming samples and the local copy of the preamble. Whenever the sign of the input sample matches that of the local copy, the output is a '1' otherwise '0'. The output of the 256 comparisons is accumulated using a 2 bit adder tree which consists of $log_2(256) - 1$ adder stages for a 256-bit correlation.

- The last stage of the correlator is a threshold based detection of the correlation energy. Once the correlator finds the exact 256 samples the output is a high energy peak, which is used to trigger other subsystems of the receiver to decode the packet. Although the correlator uses 1 bit instead of all the 16 bits of the complex sample, we still find the performance to be satisfactory under varying SNR. We discuss the performance of this kernel in §4.2.

FFT and Equalizer: The FFT kernel in the receiver is triggered by the Synchronizer. Once synchronization is achieved the FFT converts the input I/Q samples back to frequency domain. We use the common FFT/IFFT block as shown in figure 4.4 to perform the FFT and remove the cyclic prefix. Based on the information in 'FFT/IFFT Register', the input line is selected, from where the samples are pushed in to the FFT/IFFT block, which is set in FFT mode. Then the cyclic prefix is removed based on guard size and samples are forwarded to the equalizer in receiver mode.

Once in frequency domain we can decode the information bits after some signal conditioning called equalization. The primary function of the equalizer is to compensate for any phase and amplitude noise that was introduced by the wireless channel. The amount of equalization will vary according to the signal to noise ratio and also on the multipath distortion. OFDM transmission makes the equalization process particularly simple. Pilot subcarriers are included at regular intervals at pre-defined subcarriers which assist in the equalization process. The phase and amplitude of these periodic pilots provide estimates of the channel at those subcarriers which are used to compute the channel estimates of the intermediate subcarriers by performing linear interpolation between two consecutive pilots. The hardware design of such an interpolation based equalizer is shown in [75].

The basic equalizer performs well for the contiguous set of subcarriers where the interpolation is performed for all the subcarriers. A closer look at different protocols reveals that the position of the pilots as

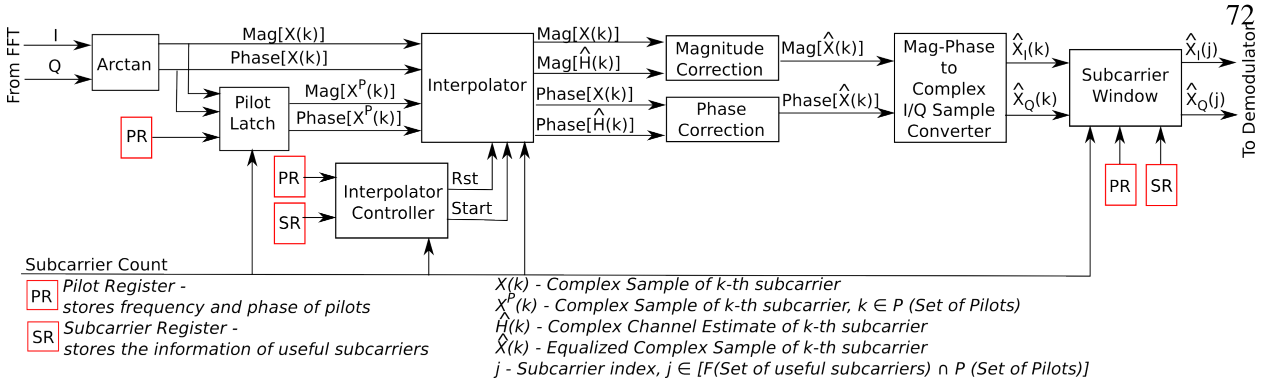

From FFT
I
Q
Arctan
Mag[X(k)]
Phase[X(k)]
Pilot Latch
PR
Mag[X^P(k)]
Phase[X^P(k)]
Interpolator
PR
SR
Interpolator Controller
Rst
Start
Mag[X(k)]
Mag[Ĥ(k)]
Phase[X(k)]
Phase[Ĥ(k)]
Magnitude Correction
Phase Correction
Mag[X̂(k)]
Phase[X̂(k)]
Mag-Phase to Complex I/Q Sample Converter
X̂_I(k)
X̂_Q(k)
Subcarrier Window
X̂_I(j)
X̂_Q(j)
PR
SR
To Demodulator
Subcarrier Count
PR Pilot Register - stores frequency and phase of pilots
SR Subcarrier Register - stores the information of useful subcarriers
X(k) - Complex Sample of k-th subcarrier
X^P(k) - Complex Sample of k-th subcarrier, k ∈ P (Set of Pilots)
Ĥ(k) - Complex Channel Estimate of k-th subcarrier
X̂(k) - Equalized Complex Sample of k-th subcarrier
j - Subcarrier index, j ∈ [F(Set of useful subcarriers) ∩ P (Set of Pilots)]

Figure 4.6: Programmable equalizer

well as their phase and magnitude may not remain constant over time. This requires some of the components of the equalizer to be made programmable. Figure 4.6 shows the structure and programmable registers for the equalizer. The **Pilot Register** holds the information of the pilot phase and magnitude, and **Subcarrier Register** contains the subcarrier information that are used for this particular OFDM symbol. The pilot latch unit stores the channel estimates at the pilot subcarrier as defined by the contents of the **Pilot Register**. Depending on the number of subchannels used and the number of subcarriers in one subchannel, the number of pilots can vary. The channel estimates for the intermediate subcarriers is computed using the estimates at the pilot. But instead of interpolating over the entire bandwidth, only the subcarriers defined by the **Subcarrier Register** is used for the interpolation. In this way we can equalize any arbitrary OFDM transmission which is the essence of a SDCR. We discuss the non-contiguous equalization and its performance in §4.2 with noisy waveforms.

Demodulator: The demodulator uses a simple threshold test to decode the information bits from the equalized I/Q samples [79] as shown in figure 5.23(b). The data rate or the modulation type can vary even for every subcarrier; per subcarrier de-modulation information is a key programming feature for this kernel. Apart from the data rate, the decision boundaries to decide on which constellation was transmitted can also vary. This is required for decoding packets encoded with superposition coding as discussed in §4.1.2. Decision boundaries for each modulation type is obtained from the **Mod/Demod Level Register** and the demodulation type (BPSK, QPSK, 16QAM or 64QAM) for each subcarrier is read from the **Mod/Demod Info**

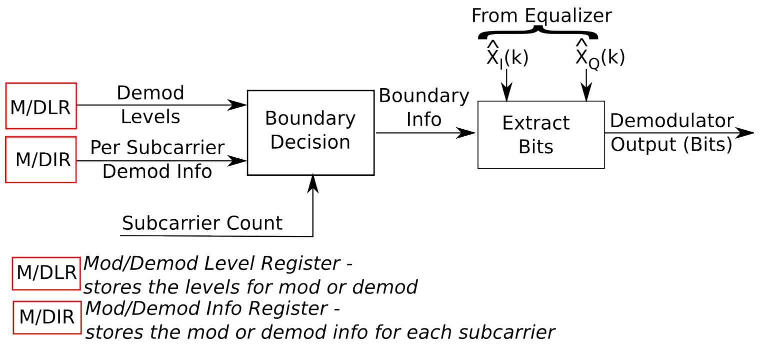

From Equalizer
$\hat{X}_I(k)$
$\hat{X}_Q(k)$
M/DLR
Demod Levels
M/DIR
Per Subcarrier Demod Info
Boundary Decision
Boundary Info
Extract Bits
Demodulator Output (Bits)
Subcarrier Count
M/DLR Mod/Demod Level Register - stores the levels for mod or demod
M/DIR Mod/Demod Info Register - stores the mod or demod info for each subcarrier

Figure 4.7: Programmable demodulator

Register. The subcarrier count is an input to the demodulator to selectively demodulate non-contiguous subcarriers. Based on these inputs, the Boundary Decision module generates the boundary information, which is then used by the Extract Bits module to perform threshold test on the I/Q samples from Equalizer to generate demodulated bits.

4.1.4 Cognitive Sensing

The transceiver can be programmed in the sensing mode, by changing the mode in the 'FFT/IFFT Register'. In this mode, only the FFT/IFFT kernel remains functional as shown in figure 4.4. In OFDM, sensing can be more than detection of carrier power; in most of the cases it is actually the detection of power in each of the subcarriers, which requires performing an FFT on the time domain signal to detect which subcarriers have been transmitted. Spectrum sensing is used by cognitive radio to detect primary users [71]. The final detection mechanism depends on initial FFT results, and may vary significantly from one procedure to another. Hence, we assume that there will be a Cognitive Engine above the PHY to determine the detection based on the FFT values. So, the FFT values are passed on to the Cognitive Engine in sensing mode.

Recent advances in wireless network protocols show that individual subcarriers can be used to transmit higher layer information, like MAC layer acknowledgments [7], or parallel polling [52] or voting mechanisms. This method can be used in any kind of signaling procedure. In these scenarios, simple threshold checks on subcarrier energy reveals whether there has been any transmission in any subcarrier or not. We

Table 4.1: Configurations supported by the SDCR

Kernel	**Supported Configurations**
Subcarriers	256
Guard	1/4, 1/8, 1/16
Correlator	256 samples wide
Data Rate	BPSK, QPSK, 16QAM, 64QAM and superposition coding using these
Pilots	32
FFT size	64, 256, 512, 1024, 2048

have implemented this detection mechanism in our FPGA system as shown in §4.2.

4.2 Implementation and Results

In this section, we describe the details of our implementation and discuss how we program each of the modules in the transceiver to perform as required in the Cognitive Radio Network.

4.2.1 Implementation

We have implemented the programmable SDCR on a Virtex-V FPGA shown in figure 5.22. The design flow consists of Xilinx System Generator: used for designing all the baseband processing kernels and their controllers and VHDL: to interface with the host PC. The current version of the design supports most of the transmission parameters required for a cognitive radio deployment and are listed in table 4.1. The logic consumption details is given in table 5.3.

Table 4.2: Transceiver hardware utilization in Virtex-IV FPGA

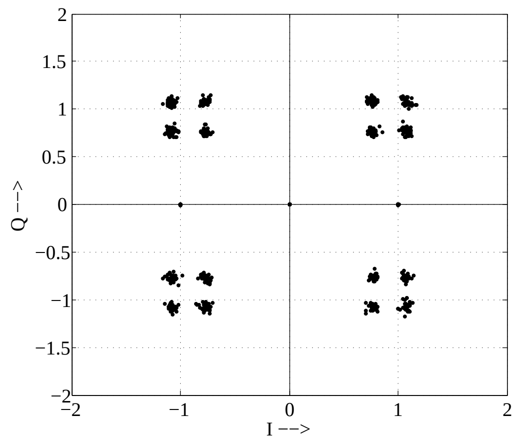
Parameter	**Count / Max**	**Utilization**
Slices	14497 / 15360	94%
Slice Flip Flops	17,644 / 30720	57%
4 input LUTs	20,080 / 30720	65%
FIFO16/RAMB16s	105 / 192	54%
DCM_ADVs	3 / 8	37%
DSP48s	55 / 192	28%

4.2.2 Results

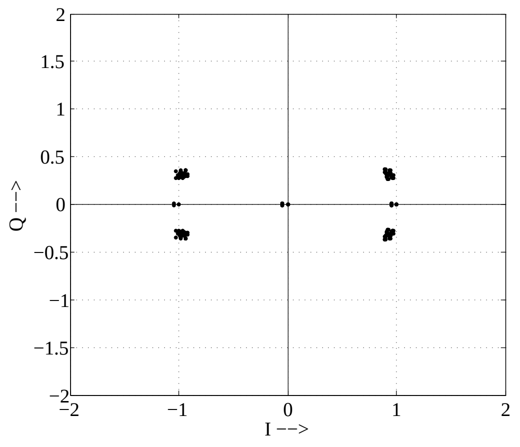
In this section, we demonstrate that our FPGA implementation can be programmed as required and we show the results of reprogramming each of the modules.

Programmable FFT/IFFT: We have tested the transmitter kernels by transmitting packets using non-contiguous OFDM by suppressing different set of subcarriers in each packet. Figure 1.1, shows the spectrogram of the University of Colorado emblem captured by an Agilent Vector Signal Analyzer which is transmitted using non-contiguous OFDM.

Programmable Modulator/Demodulator: Apart from supporting multiple forms of constellations, an SDCR may be required to transmit arbitrary constellations as a part of multi-layered modulations. Figure 4.8 shows the flexibility of the SDCR by transmitting modified constellations using different modulation levels. Figure 5.19(c) is a modified 16QAM constellation where as figure 4.8(b) shows a modified QPSK constellation. This is quite common in using hierarchical modulation and superposition coding. At the receiver the kernels have been tested by transmitting waveforms using our transmitter over a simulated channel multipath channel instantiated in the FPGA. This has been to compensate for a lack of a wide band front-end, which is considered as future work. We captured signals at the demodulation level in the receiver chain to generate figures 5.19(c) and 4.8(b).



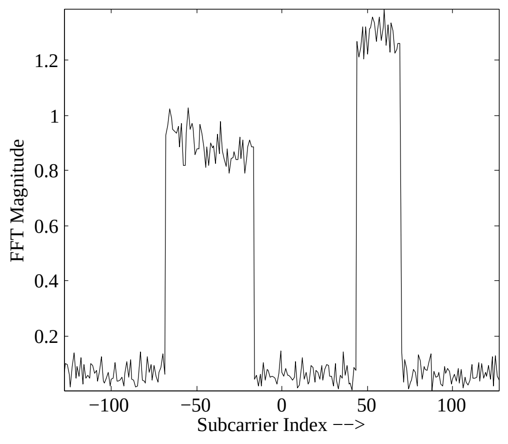
(a) Modified 16QAM



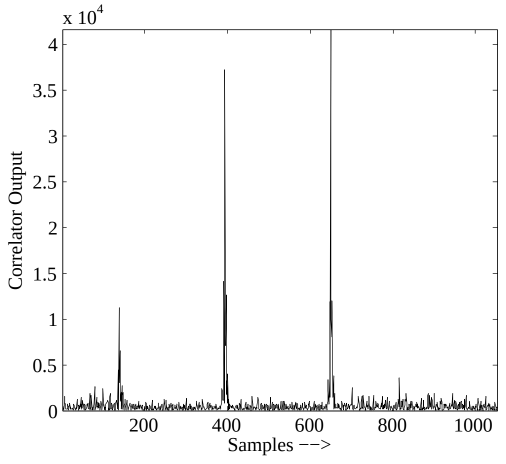
(b) Modified QPSK

Figure 4.8: Programmable constellations using SDCR

Programmable Correlator Figure 5.9 shows the input and output traces of the SDCR correlator which computes correlation on a preamble transmitted in non-contiguous spectrum. Figure 4.9(a) shows that the preamble is transmitted in two subchannels using subcarriers, $[-68 : -17]$ and subcarriers $[44 : 69]$. With prior knowledge of the subcarrier information from the **Cognitive Engine** and the preamble superset, the correlator regenerates the preamble and then performs the time-domain correlation. Figure 4.9(b) shows the correlation energy for the preamble with two high peaks corresponding to the two preamble symbols transmitted at the beginning of the each packet to aid synchronization. The unequal amplitude of the two subchannel is a result of the multipath effect and the additive noise in the received signal at an SNR of approximately $10dB$.



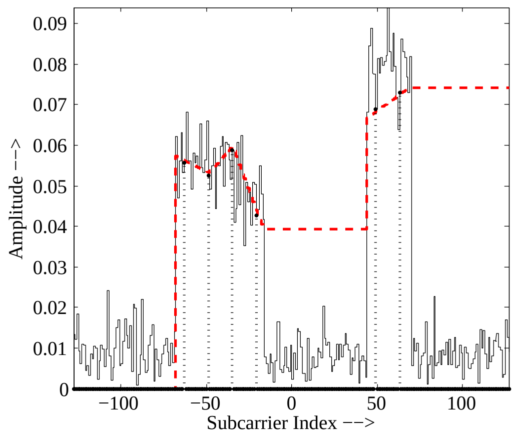
(a) Non-Contiguous spectrum



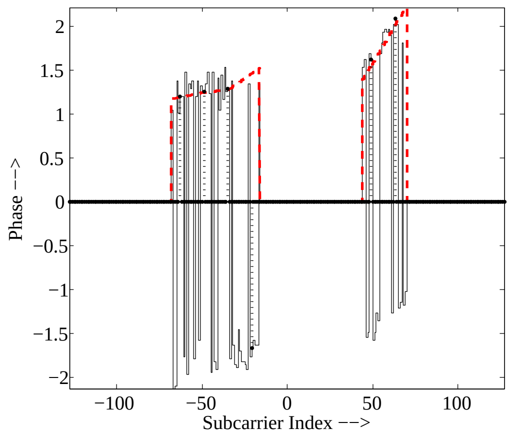
(b) Correlator output

Figure 4.9: Programmable correlator input and output

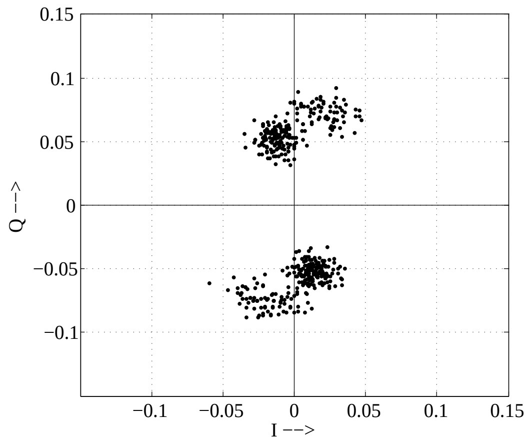
Programmable Equalizer Figure 4.10 shows the performance of the equalizer using the same received signal. The red dotted line in figure 4.10(a) and 4.10(b) shows the interpolation for magnitude and phase respectively over non-adjacent OFDM subcarriers. Figure 4.10(c) shows the unequalized constellation for the same waveform. As we can see that the noise in the phase causes rotation of the constellation by almost $\pi/2$ radians which makes the demodulation quite erroneous. The equalizer de-rotates the constellation after correcting the phase and magnitude of the OFDM subcarriers by channel estimation and pilot-aided linear interpolation as discussed in §4.1.3. The de-rotated constellation is shown in figure 4.10(d) which shows a clean BPSK constellation that can be successfully demodulated without errors. We captured signal from the FPGA at the Equalizer level to generate figure 4.10.



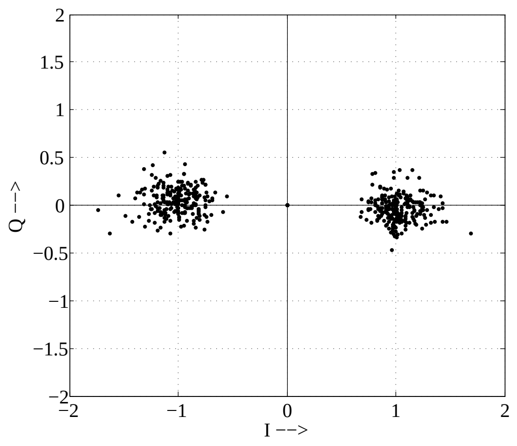
(a) Amplitude Equalization



(b) Phase Equalization



(c) Unequalized BPSK



(d) Equalized BPSK

Figure 4.10: Equalizer performance

Spectrum Sensing: Spectrum sensing is one of the fundamental operation of a cognitive radio. Using OFDM offers the advantage of spectrum sensing without the use of of any new hardware. The Fourier transform unit required to decode OFDM signal can also be used to sense the spectrum for any primary user of the channel. The energy based sensing is a simple mechanism that use a threshold on the FFT output to decide on a **spectrum hole**. Figure 4.9(a) shows an example of FFT output. Using a simple threshold we can clearly identify which part of the spectrum is occupied and which part can be used for the secondary transmission.

4.3 Efficiency and Generality

In this example we aimed at addressing the implementation challenges that are faced in developing next generation wireless network protocols. Using various contemporary research in the field of wireless data communication [6, 48, 55, 56, 62, 53, 6, 7, 52], as examples we present a generalized framework for a SDCR node. Although this implementation is based on FPGA, the design can be easily translated to ASIC implementation and it has sufficient software interface to allow reconfigurability at real time to support a wide variety of waveforms including those employing OFDM based DSA.

We do not find any current research, described in chapter 2, on software defined cognitive radio architecture. However, we assume the software based implementations, capable of processing OFDM signal structure, can be modified easily to perform as a cognitive radio platform. GNURadio and SORA are two such architectures, which we study in detail. GNURadio [4] is a purely software based implementation of SDR platform, and efforts [53] have been made to use it as a software defined cognitive radio. However, large processing delays [93] in the processing pipeline of GNURadio platform is a fundamental limitation in implementing a radio chain in real time. Another recent effort is SORA [11], which implements 802.11 pipeline in software, taking advantage of streamlined processing of multi-core architecture, cache optimized lookup tables and core dedication for specific SDR tasks. SORA meets timing requirements for 802.11, which handles data from relatively small number of OFDM subcarriers (64 in case of 802.11a/g). However, a wider bandwidth cognitive radio may need to process data from 1024 or 2048 subcarriers which might require researchers to redesign SORA, in a way to respond within the timing constraints of a protocol. Compared to these contemporary architectures our framework uses a more generic design approach that is equipped to support future cognitive radio PHY processing at real-time.

In order to make the design generic we have included a simple programming interface using registers and state machines. While the state machines are responsible to generate control signals for various signal processing blocks the signal processing blocks themselves need to have additional resources to interpret those signals and adapt to a changing radio environment. This leads to additional hardware resource. Compared to a basic $802.11a/g$, OFDM based transceiver, the SDCR framework consumes an additional 5.41% of slices and 2.02% of LUTs. This excess hardware usage can be considered negligible given the wide array of waveforms it can support, making it an efficient platform for future wireless technologies. Since OFDM based DSA seems to be the choice for cognitive radio deployments, the architecture presented in this example is considered suitable for the signal processing requirements of such networks.

4.4 Conclusion

In this work we propose an architecture of a *software defined cognitive radio* by defining the specific requirements for every transceiver subsystem to be able to operate in a true cognitive radio network. We aim to deconstruct the conventional OFDM based radio pipeline to include specific programmable features that changes with available spectral resources. Rather than making multimode radio, we aim to treat it as a generic architecture which supports a family of transmission protocols in a cognitive environment. We also propose a programming interface that re-programs the underlying hardware with minimal overhead and with fine-grained control over different operating parameters. It is our belief that this architecture along with a cognitive controller will allow innovations towards future cognitive radio deployments.

Chapter 5

On-Demand Intelligent Physical Layers: Crosslayer protocol design using SDR

In this chapter we discuss few applications of an evolved radio prototype described in chapter 4. This evolved radio prototype has new features that enable a broader realm of networking research. These features, if carefully utilized lead to improvements in various aspects of wireless network. These techniques are categorized a MAC-PHY crosslayer optimization. In this section, we discuss "four" crosslayer techniques utlizing the programmable radio platform and highlight the implementation aspects of these techniques and refer to the publications for evaluation and results.

5.1 SMACK: SMart ACKnowledgment [7]

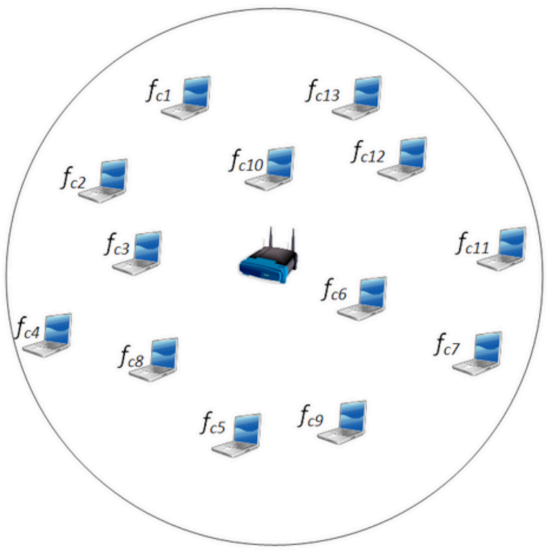
In this work, we focus on speeding **group communication** using **simultaneous transmission and reception**. There are many types of group communications, the most common of which is broadcast or multicast. Conventional infrastructure wireless networks (**e.g.**, a standard WiFi network) usually only use broadcast packets to translate wired broadcasts into wireless packets. The standard 802.11 physical layer doesn't provide a method for determining if a broadcast was delivered; thus such broadcasts are typically transmitted at the lowest modulation rate (in an effort to increase the reliability of reception). Since broadcast messages are exchanged without acknowledgment control frames, there is a limited scope for the source or the access point (AP) to reliably ensure the reception of the message at the host nodes.

In "ad hoc" networks, broadcast messages are used for many purposes. Typical applications include host discovery, network maintenance, route discovery, etc. For example, wireless protocols such as AODV [94] periodically broadcast a routing table to "neighboring nodes" (meaning those that can hear the message). Nodes also periodically transmit “hello” messages to determine if nodes are still reachable. These messages are typically “unicast” messages, because there is no way to safely determine if they’ve been received.

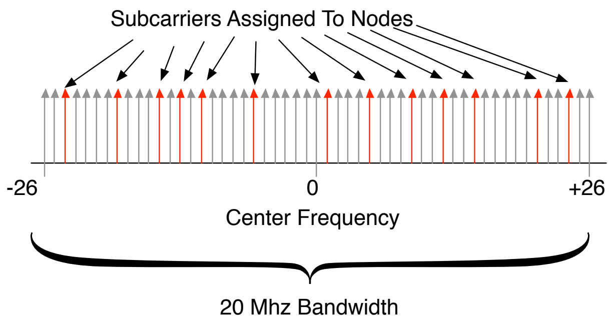
Reliable broadcast messages, “hello” link maintenance messages and many other communications share a common pattern: a message is sent and one or more nodes should “vote” on the transmitted message. For reliable broadcasts, the vote is an acknowledgment that “I have received and can decode the message”. If a node has not received the message, the sender would re-transmit it. Link maintenance messages are almost identical, except that if a formerly “adjacent” node does not receive the message, it is removed from the node neighbors table (with no retransmission). Many other protocols, such as voting protocols, can map to a similar query followed by a yes/no decision from other nodes.

Some of these protocols concerning a single network “link” have an analogous extension to a “network” counterpart. For example, there is considerable work on providing reliable network-wide support for broadcast packets in wireless networks, as well as distributed leader election.

5.1.1 SMACK - Reliable Link Layer Broadcasts



(a) Subcarrier assignment in a network



(b) Non-contiguous OFDM transmission

Figure 5.1: Schematic illustration of ACKs using OFDM

For any reliable broadcast mechanism to be reliable, there must be a clearly defined set of nodes in the network; Figure 5.1(a) shows a single access point and multiple clients. Each client is assigned a unique “membership number”. For our implementation we have chosen the OFDM based physical layer for $802.11a/g$ as the underlying signaling method. Figure 5.1(b) shows a schematic illustration of the properties of the OFDM waveform that are needed. A given bandwidth, such as the 2.4Ghz band used by 802.11g, is subdivided into a number of **subcarriers** around a center frequency; that center frequency is the “channel” to which an 802.11 radio is set.

In 802.11g, 53 subcarriers remain for data modulation. Normally, a single transmitter modulates all subcarriers to send high bandwidth data. In our protocol, since we only need to transmit a “yes” or “no”, we assign subcarriers to individual nodes, as illustrated in Figure 5.1(b); different clients are assigned subcarrier bins labeled as $f_{c1}, f_{c2}, \ldots, f_{cn}$ where n depends on the number of users and the number of subcarriers available. The orthogonality of individual subcarriers allows us to use each of them as separate data carriers for different hosts. Using multicarrier modulation techniques allows the AP to receive ACKs from a greater number of clients in the shortest possible time, dramatically reducing the time to gather reliable acknowledgments for broadcasts. We use the physical layer to combine the responses from the different nodes. Upon receiving a successful broadcast message from the AP the clients use their pre-defined subcarriers to transmit a $'1'$ as an ACK.

To summarize, the protocol has the following steps:

(1) When nodes join the network, the AP assigns each node a unique “membership id”, which is a small integer.

(2) An AP sends the broadcast message using conventional PHY specifications for $802.11a/g$.

(3) On receiving the broadcast message all clients decode the message (if possible).

(4) If a client successfully decodes the message, the client then uses the single orthogonal subcarrier specified by the membership identifier to indicate it has received and decoded the message

(5) The AP receives the composite time domain signal of **all** OFDM subcarriers and performs an FFT

to obtain the frequency domain representation of the signal. After performing demodulation the individual acknowledgments can be recovered. A one in the n^{th} bit position can be mapped as an ACK from one of the N (number of subcarriers) clients.

Due to the conversion between the time domain and frequency domain, relatively tight timing synchronization is needed for the composite additive signal to be decoded at the AP – in other words, all the responding stations must transmit at about the same time; however, that time synchronization is provided by the broadcast message itself.

To understand how much more efficient it is to use physical layer signaling, consider the costs of transmitting a message using the 802.11g PHY that is the basis for our extension. A normal message requires a $20\mu s$ preamble to be transmitted and then, at best assuming the $54Mbps$ modulation rate, each 48×6 bits takes one OFDM symbol time $(4\mu s)$ to transmit. Thus, a 64 byte message, which can't actually even contain the Ethernet addresses in a standard 802.11g packet would take at least $20 + 4 \times 3$ or 32μseconds. After a $16\mu s$ "SIFS" period for a $20MHz$ channel [79], clients would normally respond using a similar message format. Thus, a single response to a standard 802.11g packet would take another $\approx (32 + 16) = 48\mu s$.

By comparison, using physical layer signaling **53** clients can provide a single bit of information within two OFDM symbol periods, or a total of $8\mu s$ or one-sixth the time for a **single** station to respond using standard messages. This means that using the proposed protocol, the time needed for a single station will be reduced by about an order of magnitude; when the number of potential respondents increases, that time is reduced by two orders of magnitude.

5.1.2 Implementing SMACK using SDR

To demonstrate simultaneous reception for reliable acknowledgments we implemented a prototype using a SDR platform. The SDR involves an OFDM transceiver on a Virtex-IV FPGA along with a custom front-end radio as shown in Figure 5.2. The design and implementation has been detailed in [74, 75], which deals with all the signal processing algorithms that have been synthesized into fixed point hardware designs. The platform is capable of transmitting and receiving generic 802.11g as given in physical layer specification

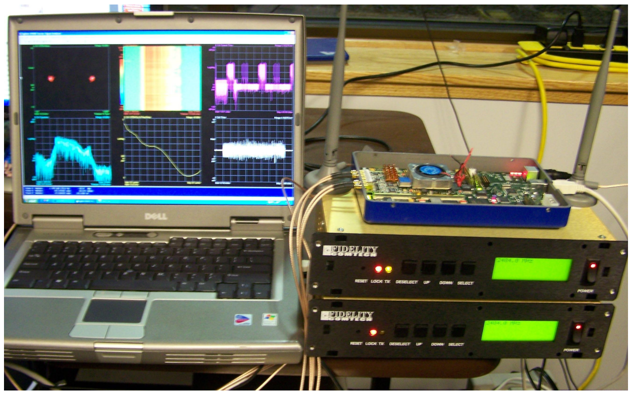


Figure 5.2: Nallatech boards with radios and antennas

[79]. The OFDM transceiver components consist of a custom radio front-end responsible for up/down conversion to/from the 2.4GHz ISM band and a Xilinx ExtremeDSP development kit IV manufactured by Nallatech. The ExtremeDSP board includes either a Virtex IV or a Virtex II FPGA equipped with a PCI/USB interface and two sets of A/D and D/A converters. Gain control is also a part of the radio that can be controlled by software on the host computer.

Transceiver latency plays an important role in our implementation. It is required to determine the turnaround time for the receiver at the broadcast node. Usually for any practical transceiver, the minimum time that is required for the MAC/PHY to receive the last symbol of a frame at the air interface, process the frame and respond with the first symbol on the air interface of the response frame is of great interest. This includes receiver side PHY layer processing delay + MAC processing delay + Transmitter side processing delay + PCI transfer delay for both Rx and Tx + Front-end radio hardware delay. If we disregard the MAC processing delay and the PCI transfer delay then we can summarize the following:

(1) Receiver side:

Difference between the last symbol received at the air interface to last bit transferred to host = 14.83μsec.

(2) Transmitter side:

Difference between the FIFO read signal to the first analog sample out from the DAC = 11.68 μsec.

(3) Key note: The FFT/IFFT module consumes the bulk of the latency = 7.4 μsec. x 2 (for Tx and Rx)

= 14.8μsec.

It is observed that most clock cycles are consumed by the FFT/IFFT unit and other than that the latency is attributed largely to various buffering elements required for proper functioning of the pipeline. in order to further reduce latency we need to use better pipelined cores with faster cycle times. This is purely a limitation of our prototyping hardware, and not of the method – any commercial WiFi chipset is already capable of the processing needed to implement our technique.

The receiver side of the broadcast node comprises of an FFT engine coupled with the energy detection blocks as shown in Figure 5.3. This design can form a part of the standard receiver chain [75] and the mode of operation (depending on if the node is operating as a client of AP) can be easily selected using software controlled registers.

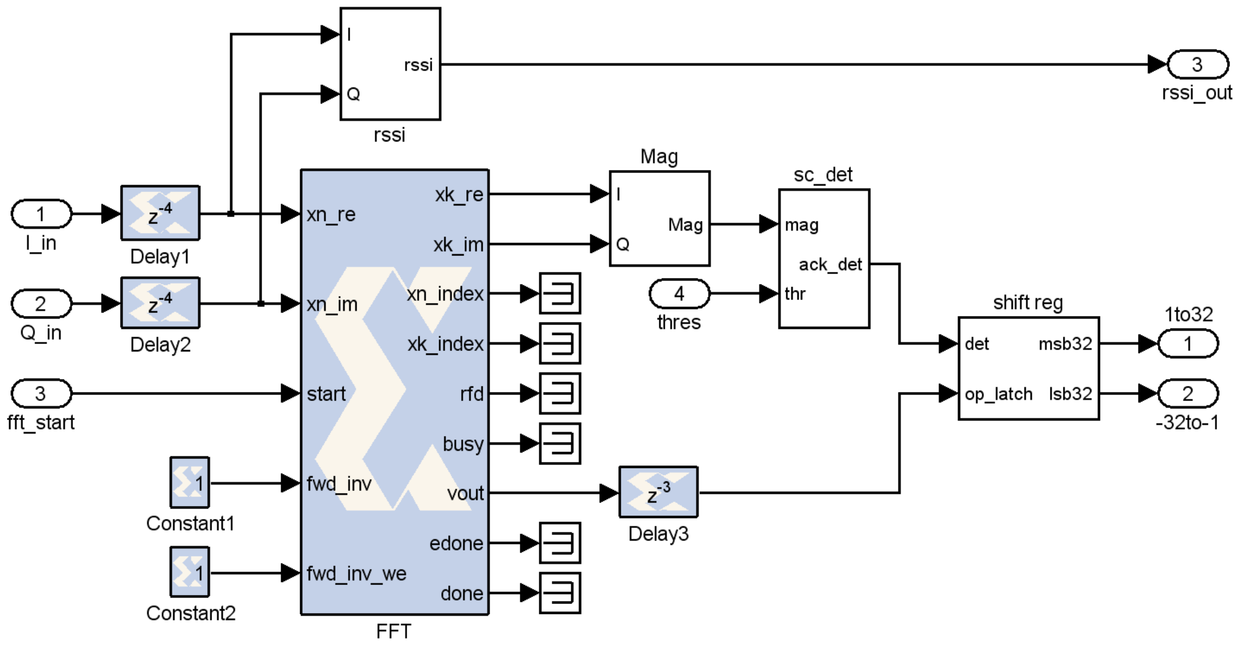


Figure 5.3: Design for the detecting ACK at AP

Triggering the FFT is a key design challenge. Given our hardware design and its inherent latencies, we find that the total time required for an ACK to reach the AP is $(T_{rxlatency} + T_{txlatency}) = 26.51\mu s$. Since ACK transmission control logic is done in hardware, no MAC processing delay or PCI/USB data transfer delays are introduced. In order to accommodate any propagation delays and other eventualities we further add a cushion of $2.49\mu s$ to the above latency. Thus we trigger the FFT exactly $29\mu s$ after the last sample of the broadcast packet transmitted to air interface. This time difference ensures that all the ACK tones from client nodes are available with sufficient energy at the AP to be able to use a simple threshold test to detect them.

The transceiver is operated in the 2.4GHz ISM band with a $20MHz$ bandwidth in order to co-exist with other $802.11a/g$ transmissions. The $20MHz$ spectrum is split into 64 subcarriers including the 0^{th} subcarrier (d.c.). The 0^{th} subcarrier is never used as it will introduce unwanted DC offset at the receiver which has to be removed using suitable algorithms. The output of the energy detector is typically a **bit mask** of 63 subcarriers (excluding the dc subcarrier). This 63 bit mask is read by the MAC layer routine using two software addressable registers. The bit mask for $-ve$ subcarriers are numbered MSB = -32 to LSB = -1 while the $+ve$ subcarriers are numbered MSB = 1 to LSB = 33 (which happens to be always zero as we are using 32 subcarriers). For example, if subcarriers [-26, -16, -6, +6, +11, +16] are being used to transmit ACKs then the bit mask for the $-ve$ frequencies is given by $0x2008020$ and that of the $+ve$ frequency is $0x4210000$. The presence of a '1' in the bit mask indicates that subcarrier index is used to transmit the ACK. Thus a reliable and fairly simple detection of acknowledgment has been accomplished using a software defined radio.

5.1.3 Conclusion

We've shown that by using, rather than fighting against, the properties of the wireless physical media, we can develop robust signaling primitives that are both practical and allow innovative algorithms. We used a signaling method based on OFDM that is easy to understand and implement using reconfigurable hardware. We have also shown that if the signaling mechanism is kept simple, not only does it makes certain network functions, such as reliable broadcasts faster, but can also use simple detection mechanism to extract the required information. These primitives can also be used to implement higher level group communication and signaling protocols as long as the queries require simple "yes/no" answers. The critical insight is that we can combine the results from multiple clients using simultaneous reception in an efficient manner to aid higher protocols to perform more efficiently.

5.2 Blind Synchronization of NC-OFDM [61]

Bandwidth utilization in wireless communication not only refers to the range of radio frequencies used in a particular radio-wave transmission, but also to the duration of useful communication in the occupied bandwidth. Widespread spectrum measurements [95] and analysis [96] have shown the availability of **whitespaces** and enormous capacity that can be used by other users as long as they do not interfere with incumbents of that part of the spectrum. While shared usage of unused parts of licensed spectrum provides opportunity for higher bandwidth utilization, it also opens up a world of new challenges. Often these unused spectrum are available in **chunks** of frequency bands. The secondary user intending to use these chunks will have to transmit over non-adjacent frequencies while avoiding the frequencies used by the primary for that band. The term **Dynamic Spectrum Access (DSA)** has been coined to address the entire family of users that can handle such non-contiguous usage and the devices capable of DSA are termed **cognitive radios**. The cognitive radio not only needs to identify unused parts of the spectrum but also has to provide sustained communication over non-adjacent frequencies.

The availability of **whitespaces** in a cognitive radio network will vary over time with the duty cycle of the primary user. Every time this happens the communicating nodes will have to negotiate for the next set of available frequency bands. This negotiation is typically done by some form of message exchange using a dedicated radio channel, which is also shared by all the users in the network. Evidently, there is an intrinsic contention built into the system, where instead of contending for the communication channel nodes are contending for the control channel. Hence, the benefit of having DSA may be diminished by having a contention based control channel. Therefore, the goal of this work is to provide in-band signalling for channel rendezvous that is fast, simple and makes use of the reconfigurable radio. This work proposes a mechanism of channel migration that is suited for future cognitive radio network employing DSA operating over a wide band of frequencies and that is practical and implementable.

Current radios and spectrum allocation policy are tightly bound to the notion of “channels” that are, partly a historical part of spectrum allocation influenced by practical radio design of the time. The ability to operate over a wide radio spectrum may cause radio designs to look at the application characteristics of data streams when making PHY or MAC decisions. Factors such as power levels, achievable (or needed) modulation rates and the ability to trade total spectrum against each of these makes the decision of where and how to modulate more complex. Networks operating in a wide band need to know how to coordinate the stations within the network. This includes sensing when spectrum is no longer suitable for an application and deciding (and communicating) new resource allocations. Being able to modulate across a large spectrum allows communication to multiple stations simultaneously, but there are a number of PHY and MAC layer issues to resolve because some stations may use non-contiguous spectrum. In short, we envision a network of radio nodes that have one or more heterogeneous radio interfaces each able to flexibly construct signals to fill a radio band chosen from a wide range of bands. The goal is to rethink current radio network protocols to develop new protocols able to fully exploit these capabilities.

Just as different parts of the spectrum greatly influence the propagation of signals, the amount of spectrum used greatly impacts the bandwidth, the potential for interference and the MAC overhead. If different receivers are allocated their own spectrum, contention-based MAC overheads can effectively disappear. Likewise, the ability to allocate arbitrary spectrum amounts to individual flows or receivers means that capacity can be allocated at a much finer granularity than in traditional "channelized" MACs. Channelization is the most common approach to using the spectrum in the cognitive radio domain. However, the presence of a narrowband incumbent (say, a wireless microphone) might render an otherwise wideband channel unusable and lead to under utilization of the spectrum. At the same time, increased narrowing down of channels leads to multiple channels to which a device needs to tune to communicate. Restricted channelization limits the opportunistic exploitation of the spectrum when using a wideband radio. Hence, we propose a generalized concept of channel width that only depends on the bandwidth of the radio front ends on a communicating link and the available spectrum. Without this restriction, the AP can now decide on the optimum bandwidth and tuning frequency that can cater to a group of compatible devices using multiuser techniques like OFDMA, while avoiding the part of the spectrum occupied by an incumbent. The underlying data rate (or resistance to errors) can also be controlled by adjusting the OFDM parameters depending on the radio environment in which the links are communicating. This can be realized by employing distributed blind synchronization techniques for non-contiguous spectrum access as it eliminates any requirement of prior bandwidth negotiation: preliminary results of which are discussed in [97]. Thus, the MAC layer coordinates with the PHY to determine the spectrum management depending on the available spectrum, the radio environment, and the bandwidth requirement of participating clients to optimize the end-to-ends throughput.

Figure 5.4 shows an example of one such heterogeneous network, where the AP is communicating with the clients over different widths of non-contiguous spectrum. This secondary network coexists with a Primary transmission and another secondary network. We envision that the MAC, capable of communicating in this heterogeneous radio environment and achieving high end-to-end throughput will face several interrelated scheduling and allocation decisions. Clearly, using a wide 100-500MHz receiver allows a radio to transmit across a broader spectrum. In most heterodyne radio systems that use a "mixer" to modulate a carrier frequency, demodulation requires that the senders and receivers are on the same *center frequency* f_c. This constrains the selection of frequency bands used by different receivers unless it is possible to cause all receivers to "re-center" their center frequency. Rather than allocate purely contiguous spectrum to a single receiver, it may be more appropriate to use non-contiguous spectrum – this is made possible using the Non-Contiguous Orthogonal Frequency Division Multiplexing (NC-OFDM).

Co-existence of multiple users using the same carrier frequency has been little known until the evolution of multi-carrier communications technologies, like Orthogonal Frequency Division Multiplexing (OFDM) and their subsequent adoption for use in popular wireless protocols. Multi-carrier communication offers the advantage of enhanced spectrum utilization by slicing the bandwidth into closely packed, non-interfering data carriers. Once the bandwidth is available, information bits are placed in those data carriers and an inverse Fourier Transform produces one OFDM symbol of duration $N_{\text{fft}} \times F_s$ seconds (N_{fft} is the width of the Inverse Fourier Transform and F_s is the sampling frequency). At the receiver, a Fourier Transform of this symbol will reveal the information embedded in those data carriers. For the information to be retrieved with no or minimal error, the boundary of an OFDM symbol must be determined with high accuracy: the effect of mis-estimation is discussed in §5.2.1. This accuracy is ensured by the synchronizer in conjunction with other processing units like **packet detect** and **frequency offset correction**.

Synchronization is the process used to detect an incoming packet and also to identify the OFDM symbol boundaries with high accuracy. This is done by employing some form of time-domain correlation

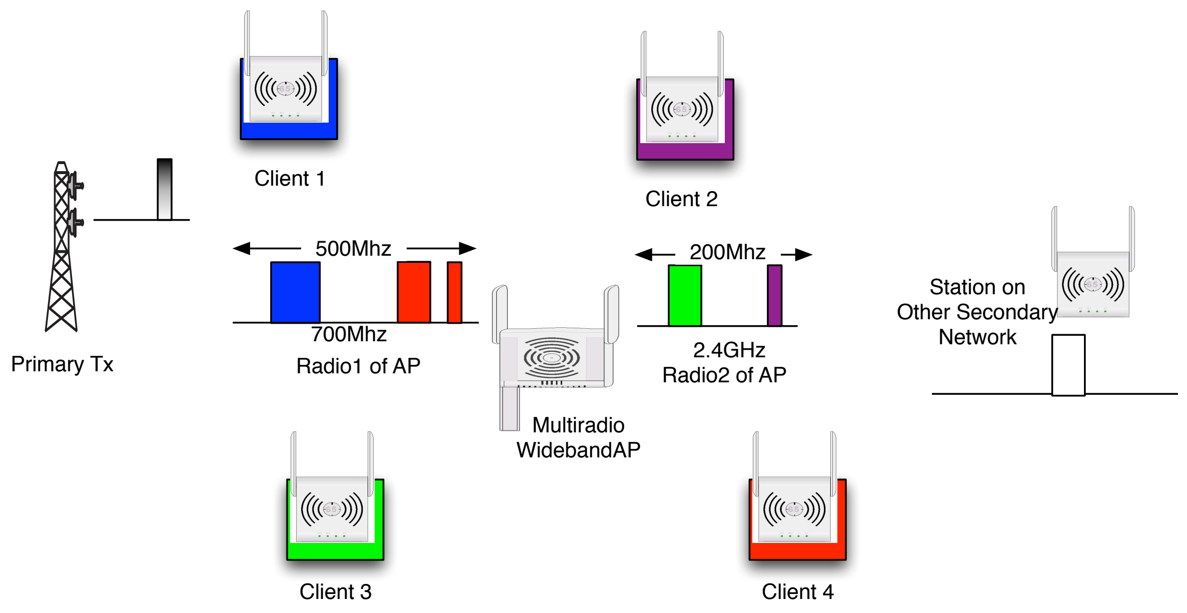


Figure 5.4: Spectrum sharing in wideband cognitive radio network: An example scenario. Secondary network, consisting of an AP and four clients, is coexisting in presence of primary and other secondary networks. The AP has two radios of different bandwidth, tuned to two different center frequencies, and is communicating over variable width NC-OFDM.

with a previously agreed upon sequence of bits or samples at the receiver. Since OFDM waveforms are modulated in frequency domain and then converted to time-domain signals, altering or notching some of the subcarriers as done in NC-OFDM will change the structure of the preamble. Unless otherwise notified, the receiver has no idea of this new sequence being used by the secondary transmitter. Hence, there must be a method the receiver can use to re-establish the correlation properties and ensure proper decoding of the signal and migrate to an empty part of the spectrum within a very short time.

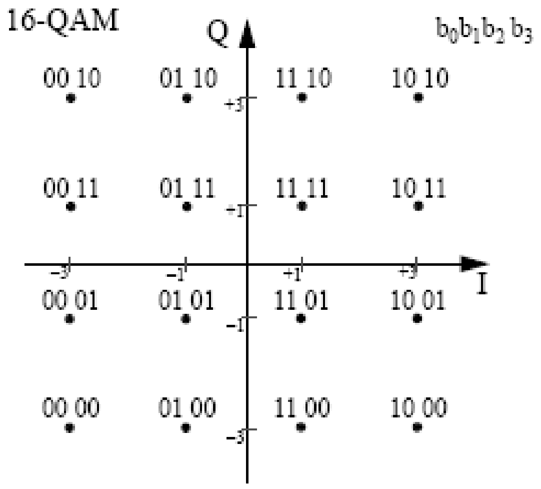
In this research, we propose a distributed synchronization algorithm for NC-OFDM and its hardware implementation. We also present a MAC layer design that uses this synchronization technique to eliminate control channels for rendezvous and simplify channel allocation and network management in a wideband cognitive radio network.

Our Contributions: The key contributions of our research are as follows:

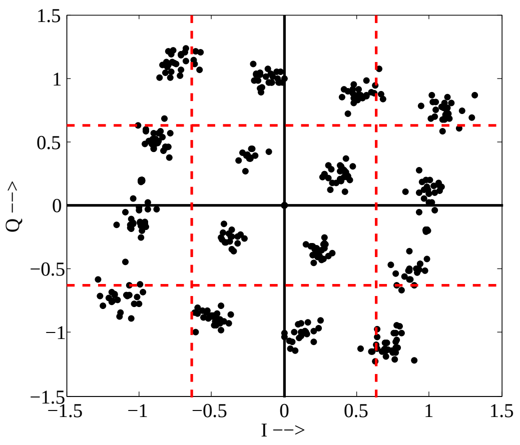
- We propose a distributed algorithm for blind (without prior knowledge of the transmitted frequencies) synchronization for preambles transmitted using NC-OFDM.

- In order to show that our algorithm can be implemented in realistic hardware, we present a programmable correlator structure that can be synthesized for FPGA.
- We analyzed the algorithm under a multi-path channel with varying SNR, yielding considerably better results at low SNR and low bandwidth occupancy compared to existing algorithms.
- Lastly, we propose a MAC framework that utilizes the synchronization technique to make wideband cognitive radio a reality by largely simplifying channel access and its control.

5.2.1 OFDM Symbol Timing



(a) Without Timing Offset



(b) With Timing Offset

Figure 5.5: Effect of symbol timing offset: 16QAM constellation after equalization at SNR = 15.5 with multi-path fading and a timing offset of (-8) samples. The decoding regions are shown by the red dotted lines.

OFDM is a frequency domain modulation technique where the fidelity of the information is highly dependent on the frequency domain properties of the signal, like power spectral density and phase-magnitude variation across the signal bandwidth. OFDM symbols are generated by inverse Fourier Transform on blocks of data to preserve orthogonality between the subcarriers. Multiple blocks are concatenated in time to form a packet. Often the orthogonality is damaged due to multi-path effects of the channel and also by other receiver impairments like incorrect sampling frequency and I/Q offset during down conversion. The block type generation of OFDM calls for a similar decoding method at the receiver to recover the frequency domain modulations. Offset in symbol timing introduces phase noise in the subcarriers and affects phase modulations like BPSK and QPSK. The effect of phase noise has been examined in great detail in previous work [83, 84, 98]. In its simple form, failure to identify the correct symbol boundary results in phase rotation proportional to the number of samples offset from the correct boundary. Figure 5.5 shows an example of the effect of timing offset on demodulation. Figure 5.5(a) is a 16QAM equalized constellation transmitted using a 64 subcarrier OFDM, which has no timing offset. The correct timing ensures that there is no phase noise in the I/Q constellation, and using the red-dotted lines showing the decoding boundaries the bits can be extracted without errors. In contrast, figure 5.5(b) shows the equalized constellation, except the signal block is now offset by 8 samples. Significant phase rotation is seen, which causes the constellation points to cross the decoding boundaries leading to undesired errors. However, there is an allowable offset that will vary based on the amount of the allowable degradation for the constellation, e.g., for BPSK, theoretically, we can allow a rotation of $\pi/2$, although the consequence of such enormous offsets on other processing units are to be considered. Therefore, we can emperically derive that for a low to medium SNR an offset of less than 3 samples can be set as a benchmark for error-free decoding of a non-contiguous OFDM waveform. This benchmark can only be met by an accurate synchronizer in the time-domain which minimizes the average timing offset for an OFDM symbol over a wide range of occupied bandwidth and SNR. A primary transmission will also add interference to a secondary transmission and alter the time domain representation of the preamble used by the secondary nodes. Therefore, we have to consider the effects of primary and secondary occupancies and their corresponding SNR at the secondary receiver. We discuss the synchronization technique in the following section, and discuss performance in §4.2 under realistic channel conditions.

5.2.2 NC-OFDM Synchronizer

Synchronization is performed using a known sequence that is transmitted over multiple OFDM symbols at the beginning of every packet, aptly named the preamble. A correlator is used to compute the

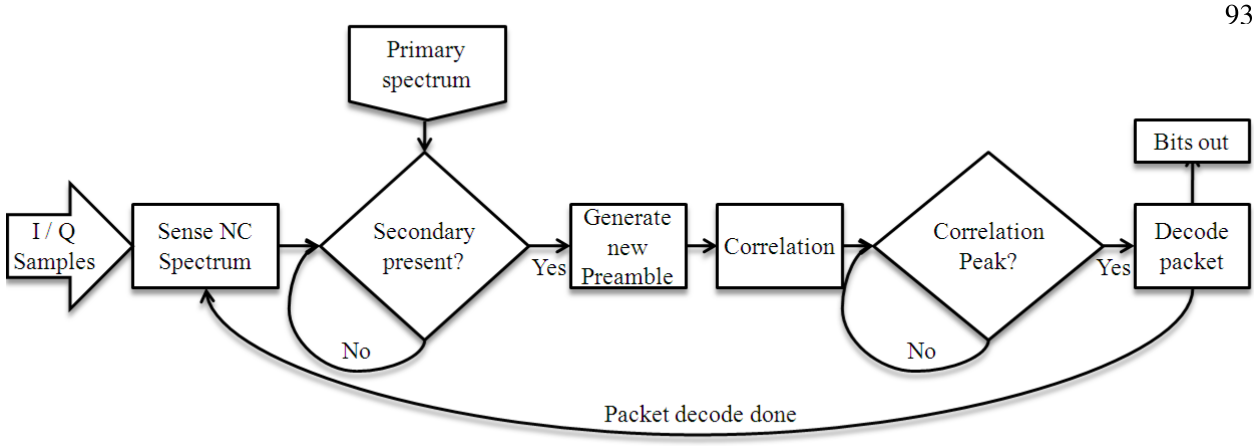


Figure 5.6: NC-OFDM synchronization pipeline

similarity of the incoming sequence of noisy samples to a locally stored copy of the preamble. The uniqueness of the preamble ensures superior correlation at the receiver, which is in turn used to identify the OFDM symbol boundary. Now, when using non-contiguous spectrum, the secondary fails to use the pre-defined preamble, which occupies a wider, and most importantly contiguous band of frequency than what is currently available. While encoding data over non-contiguous bands is easily achieved, synchronizing at the receiver poses a challenge. A receiver, trying to correlate with the pre-defined preamble will fail since the non-contiguous encoding of the signal has changed the time-domain representation of the preamble. This requires the receiver of a cognitive radio signal to be able to adapt to the changing environment. In contrast to the conventional synchronizer, the unit must be intelligent to be able to identify the data carriers used for the preamble, as well as change the parameters of the time-domain correlator to search for the correct OFDM symbol boundary.

This blind NC-OFDM synchronization is done in two steps:

- **Subcarrier Detection:** Since the receiver has no knowledge of the frequencies used by the cognitive transmitter, it has to detect the subcarriers present in the incoming signal by employing spectrum sensing across the entire band. It is assumed that the secondary nodes willing to share the licensed spectrum will have prior knowledge of the location, signal structure and power of the primary. This is a very realistic assumption if the secondary network uses database look-up to identify whitespaces, which has also been adopted as a part of recent spectrum sharing mandates by the FCC [99].

- **Regenerate Preamble and Correlate:** Once the data carriers are detected, a new preamble is generated locally at the receiver using the detected datacarriers only and the correlator is reprogrammed with the new co-efficients corresponding the samples of this preamble. In this way, we can ensure that whatever the incoming sequence is, we can re-establish the correlation based synchronization with high accuracy.

5.2.2.1 Spectrum Detection

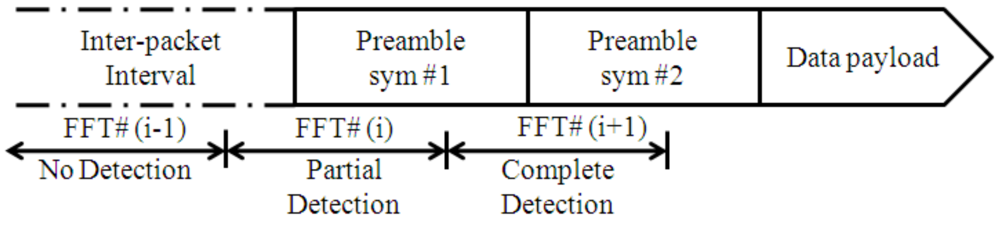


Figure 5.7: Spectrum detection

Since the objective is to perform blind synchronization, i.e., with prior knowledge of the secondary spectrum, a secondary receiver must gather the subcarriers present in the incoming signal. Figure 5.6 shows the receiver pipeline for blind synchronization. Spectrum is sensed by computing FFT on the incoming I/Q samples, and the SNR for each subcarrier is measured. These results are compared to a threshold to decide on the presence of a subcarrier in the signal. This is followed by a decision making process to ascertain the exact width of each non-contiguous band of the secondary, as well as the primary that co-exist in time with the secondary. Figure 5.7 shows a typical packet structure and the corresponding FFT windows. A secondary receiver begins spectrum sensing during the **inter-packet duration** denoted by FFT(i-1). As time progresses, the packet will start to appear in the FFT window as denoted by FFT(i). Depending on the number of samples of the packet that is present in the FFT window, the energy of the OFDM subcarriers will vary. In the subsequent FFT window, FFT(i+1), we see that signal is completely contained within the

FFT window, hence providing an **estimate** of the spectrum. This **estimate** is further processed and strengthened by the decision engine that follows the FFT, and together they form the detection engine. As seen in Figure 5.7, using two preamble symbols at the beginning of the packet will ensure successful detection of the spectrum. Since the synchronization process works as a pipeline, it only stalls when the correlation detects an incoming packet with sufficient accuracy. Therefore, depending on when the signal arrives at the receiver, we can either get good correlation from the sensing results in the i^{th} FFT window or the $(i+1)^{th}$ window.

Decision Engine: Figure 5.8 shows an example spectrum detection using a threshold test that is empiri-

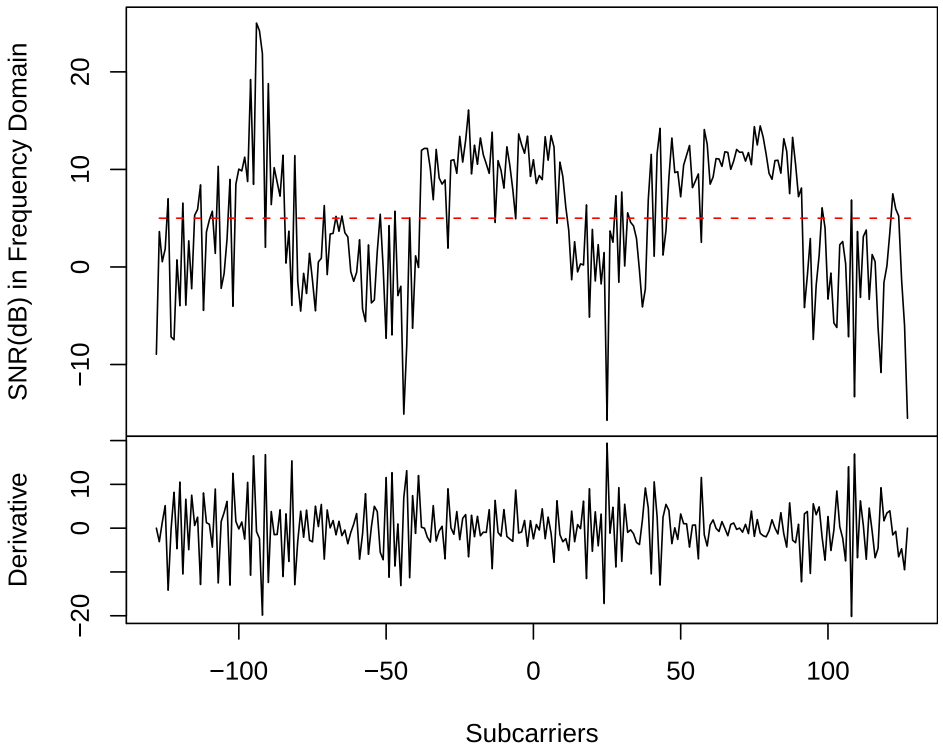


Figure 5.8: NC-OFDM Waveform - Primary occupancy is subcarriers $[-95:-91]$ and for secondary it is $[-38:11], [40:89]$.

cally derived for a particular receiver and channel setting. The spectrum of the wave reveals the subcarrier occupancy of the narrowband primary and the non-contiguous secondary transmission. The spectrum is normalized so that the average noise floor is maintained at $0dB$ so that the threshold can be independent of any fluctuations of the noise floor. This is done by measuring the noise floor during the inter-packet duration using conventional MAC "quiet times" and subtracting it from the FFT output. From the threshold test we extract a "binary mask" of width equal to the number of FFT bins in which the **ones** denote the presence of a subcarrier and **zeros** denote the absence. The mask derived from the input signal contains the primary and secondary transmissions. With prior knowledge of the primary transmission, a **XOR** operation will extract the mask of the secondary transmission only.

When signals from independent sources overlap in time and are received over a multi-path channel, it affects the orthogonality of the OFDM subcarriers and results in inter-carrier interference leading to spurious unwanted fading at the band edges, frequency selective notching within a band and attenuation. In order to make the detection robust and practical, we employ some rules to handle these spurious anomalies in the detection and correct the "binary mask" to accurately represent the secondary:

Rule 1: Remove Outliers –If two or less spurious subcarriers are detected with no carriers on either side - they are deleted from the mask.

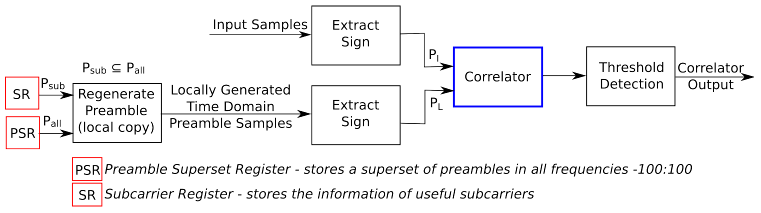
Rule 2: Fill Notches – If two or less nulls are detected in the mask with at least two carriers on either side, then fill those notches as they are considered part of a valid secondary band.

Rule 3: Identify Channel Bounds – Left edge is denoted by five consecutive nulls followed by five consecutive carriers and right edge by five consecutive ones followed by five consecutive zeros, given an inter-band guard distance of at least 10 carriers and minimum 5 carriers required to be considered as a valid secondary band.

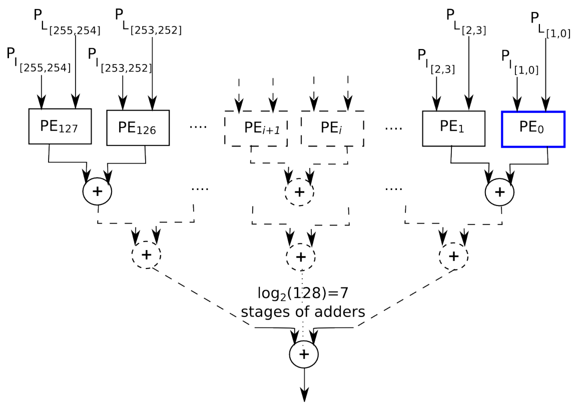
Rule 4: Adjust Band Boundaries – Secondary bands are designed to begin at multiples of five. Therefore, the final mask can be either broadened or squeezed, so that the start is a multiple of five.

Once these corrections are made to the mask, the band detection, and consequently the synchronization, is greatly improved and ensures proper decoding of the packet. We discuss these results in §5.2.4.

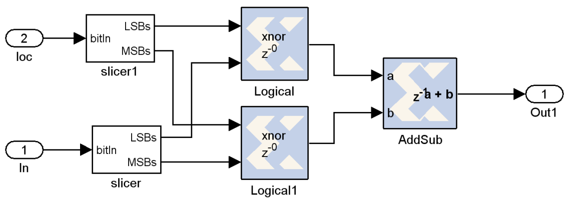
As an alternative to thresholding on instantaneous spectrum measurement, edge detection by thresholding on the first order derivative has been discussed in [53]. However, it is not suitable for instantaneous detection of spectrum by performing a single FFT on the signal, as shown in figure 5.8, and would require prolonged averaging often in the order of tens of OFDM symbols that incurs additional latency in packet detection, which is not optimal in most cases. In comparison, the method proposed here requires only two OFDM symbols and coupled with logic to correct for spurious measurements, make this technique very effective for cognitive radio networks.


Input Samples
Extract Sign
P_I
Correlator
Threshold Detection
Correlator Output
$P_{sub} \subseteq P_{all}$
SR
P_{sub}
PSR
P_{all}
Regenerate Preamble (local copy)
Locally Generated Time Domain Preamble Samples
Extract Sign
P_L
PSR Preamble Superset Register - stores a superset of preambles in all frequencies -100:100
SR Subcarrier Register - stores the information of useful subcarriers

(a) Correlator Functional Diagram



(b) Correlator Inner Structure



(c) Smallest Processing Engine

Figure 5.9: FPGA implementation of a programmable correlator

5.2.2.2 Re-generate Preamble

The detection logic is followed by the actual correlation and subsequent synchronization. The mask computed at the detection stage is used to regenerate the time-domain preamble locally at the receiver. Using the frequency domain representation of a preamble, we select the indices corresponding to the **ones** in the mask while nulling out the other subcarriers as shown in figure 5.6. This results in an accurate estimation of the preamble that is used by the transmitter. Once the time-domain samples of the preamble are regenerated by an Inverse Fourier Transform, the correlator coefficients are initialized with these samples. After initialization, the correlation energy is computed using a buffered version of the incoming packet to include all time samples from the beginning of packet. Once the correlation is successful, the packet decoding is initiated. During this time the spectrum sensing sensing results are disregarded.

5.2.2.3 Update Correlation Coefficients

If the correlation fails to satisfy a pre-defined threshold, then the spectrum sensing unit which operates in pipeline mode, will provide the detection unit with a new spectral measurement every N_{FFT}(FFT size) samples and the correlator is updated with the new mask again to test for the preamble of the next packet. Failure to detect a packet using a threshold can happen when the secondary signal SNR, relative to the primary, is very low and the secondary signal is overwhelmed by the high power primary. Under these circumstances we employ a simple re-transmission based protocol and also increase the guard band between the primary and secondary. We can re-establish the threshold based detection and correlation by this method. However, we find that under normal operating range of SNR and spectral occupancy, the threshold test is able to determine the correct spectrum over a wide range. This is discussed in section §5.2.4.

5.2.3 FPGA Implementation

In order to show that the synchronization algorithm is also suitable for hardware implementation, we propose a lightweight correlator structure that augments an existing OFDM based synchronizer and can be implemented in a FPGA. We base our programmable correlator on designs mentioned in prior works [75, 85], which implements synchronizers in hardware for contiguous OFDM. Figure 5.9(a) shows the structure of a programmable correlation unit that implements the proposed synchronization algorithm. The components of the correlator are as follows:

Acquire Spectrum Information: Spectrum sensing is an integral part of cognitive radios. Depending on the design choice, the spectrum sensing unit can be implemented as a part of the radio processing engine or as a separate unit that is a part of a cognitive engine. In either case, the correlator should have good co-ordination with this unit. The spectrum sensing unit provides the correlator and other key baseband processing units, like the equalizer and demodulator, with the information about the subcarriers used in a particular packet. Therefore, a programmable interface is required between the sensing unit and the baseband processing pipeline. Figure 5.9(a) shows a set of programmable registers that can be accessed by the spectrum sensing unit to update the “binary” mask from the detection unit for the most recent sensing results. The **Subcarrier Register(SR)** holds the mask to be used for regenerating the preamble in the following step.

Regenerate Preamble: Regenerating the preamble is required to establish the time-domain correlation. Usually for a particular network, the complete frequency domain preamble is pre-defined, and spans over all the subcarriers. Since cognitive radio can operate in multiple networks, this preamble can change. In hardware, we define a **Preamble Superset Register (PSR)** that holds the complete frequency domain preamble for the current network. Using the subcarrier map from the **SR** register and the preamble superset from the **PSR** register, the time-domain preamble is regenerated using inverse Fourier Transform. This preamble is used to perform the correlation with the incoming packet to extract the symbol timing.

Correlator Core: A time-domain correlator employs a running comparison of the input I/Q samples with a local copy of the time-domain samples of the preamble being searched for. The basic operation in a correlator is multiply-accumulate-shift for every fixed-point complex sample entering the correlator. Therefore, depending on the number of samples to be matched, the size of the correlator and the number of gates to implement the logic in FPGA increases as it requires more number of adders, multipliers and shift registers. To make the correlator easily implementable, instead of performing wide (14 to 16 bits) fixed-point multiplications we use the **sign bit** of the input I/Q samples and the locally regenerated preamble samples. In doing so, the comparison of two samples reduces to a simple **XNOR** logic operation. Whenever the sign of the input sample matches with that of the local copy, the output is a ‘1’, otherwise ‘0’. Constructing a tree of 2-input adders we can accumulate the result of the comparison for every shift operation within one clock cycle. Figure 5.9(b) shows the adder tree of the correlator for a 256 sample correlator. For a complex input, similar structure is repeated for the I and the Q paths. Figure 5.9(c) shows the **XNOR** processing engine designed using Xilinx System Generator, which performs comparison on two samples. Although the Correlator uses 1 bit instead of the usual 16 bits, we still find the performance to be satisfactory under varying SNR. We discuss the performance of the correlator and its resource consumption in §4.2.

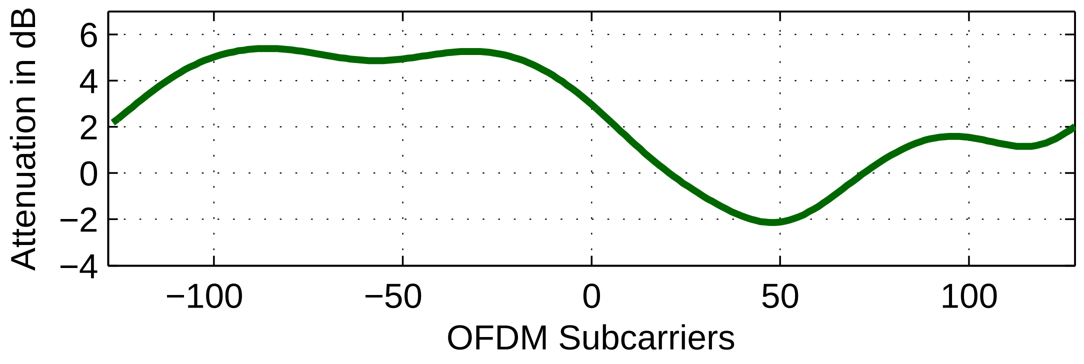
5.2.4 Results and Implementation

In this section, the proposed synchronization method is evaluated by simulations in Matlab under varying channel conditions (SNR) and spectrum occupancy. The objective is to show the performance of the synchronization and subcarrier detection as a function of the primary and secondary parameters. The signal structure and the simulation environment are shown in table 5.1. The channel used in the experiment is derived using a standard channel model built in Matlab (the **stdchan** function), along with additive white Gaussian noise. The chosen multi-path model has a 6 path delay profile for a $802.11a/g$ network. The frequency response and path delay of the channel is shown in Figure 5.10. Since the system (OFDM parameters) is designed on top of a $802.11a/g$ transceiver, so is the channel model chosen to evaluate it. Lastly, each experiment has been repeated for 500 iterations to average out jitters in the results.

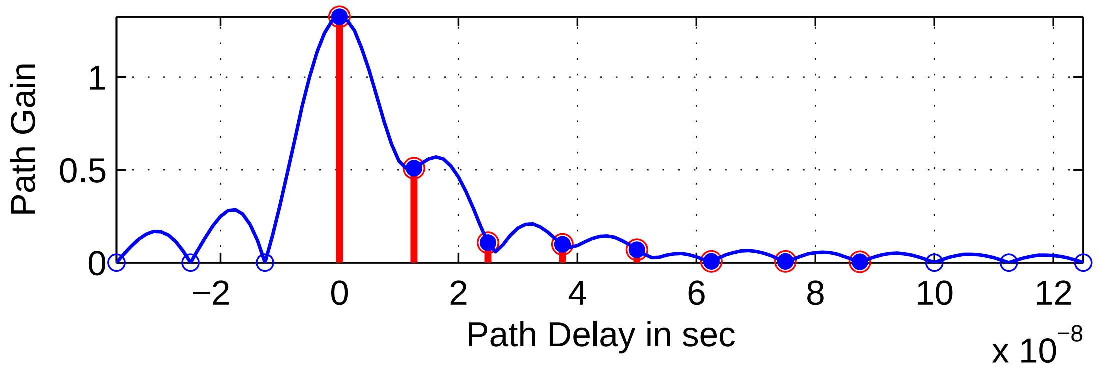
Table 5.1: NC-OFDM parameters

Parameter	**Value/Range**
FFT size (N_{fft})	256 sample
Cyclic Prefix size	64 sample
Sampling frequency (f_s)	$80MHz$
Subcarrier spacing (f_s/N_{fft})	$312.5KHz$
Preamble superset	802.16 [50]
Number of preamble symbols	2 [79]
Subcarriers per band - Secondary	$10 - 100$
Subcarriers per band - Primary	$18 - 72$ (simulates a $6Mhz$ TV band)
Guard subcarriers	10

While evaluating the algorithm we have identified a set of parameters that could vary in a network. Two of these parameters are varied at every analysis to limit the data set to three dimensions and for ease of presentation. Unless otherwise mentioned in the plots, the default values of the parameters are used for the analysis as given in table 5.2. The metric that has been used throughout the analysis is the mean absolute error in samples about the correct symbol boundary and a benchmark of less than 3 has been set for proper decoding of bits from a symbol.



(a) Frequency Response



(b) Path delay

Figure 5.10: Multipath channel response

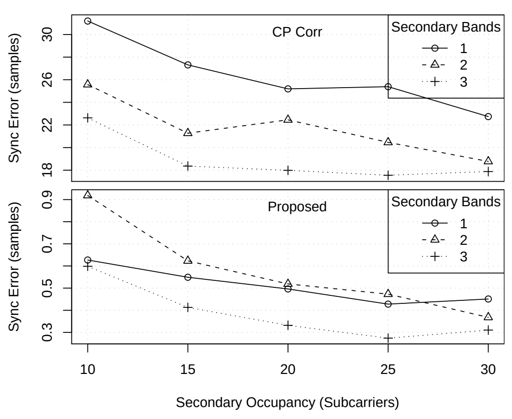
5.2.4.1 Comparison with cyclic prefix based correlation

NC-OFDM Symbol timing using cyclic prefix (CP) is one of the methods found in the literature and the proposed technique is compared with it to measure the improvement. It has been shown that the CP-based algorithm provides a coarse estimation of symbol timing [98] and this coarse timing estimate leads to phase noise and erroneous decoding of the signal constellation. In figure 5.11(a), the error is measured with varying secondary spectrum occupancy, i.e., varying the number of subcarriers per band, as well as varying the number of used by the secondary. Since CP-based timing relies heavily on the length of the cyclic prefix, and also on the similarity of the cyclic prefix to the OFDM symbol, the correlation energy is severely degraded for smaller CP length and lower numbers of non-contiguous subcarriers. Our method shows an improvement of more than $20 - 30$ times over cyclic prefix correlation. Most importantly the

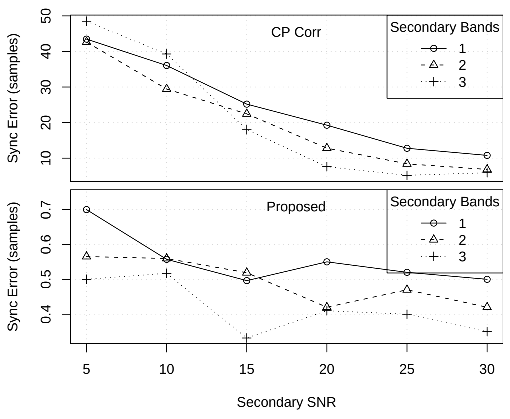
Table 5.2: Default setting for simulation parameters

Parameter	**Default Value**
Primary SNR (SNR_p)	15 dB
Secondary SNR (SNR_s)	15 dB
Number of primary bands ($NBand_p$)	1
Subcarriers per band - primary (NSC_p)	18
Number of secondary bands ($NBand_s$)	1
Subcarriers per band - secondary (NSC_s)	20

mean error is below 1. Figure 5.11(b) shows the error magnitude with varying secondary SNR and number of bands. Mean error is maintained fairly constant at ≤ 1 sample across varying SNR, whereas the CP-based correlation varies from $10 - 50$ samples.



(a) Increasing secondary occupancy

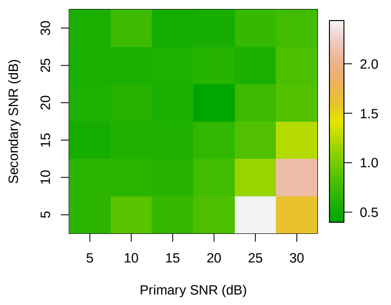


(b) Increasing secondary SNR

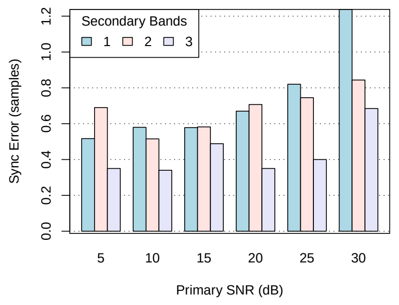
Figure 5.11: Comparison with CP-based correlation

5.2.4.2 Varying primary SNR

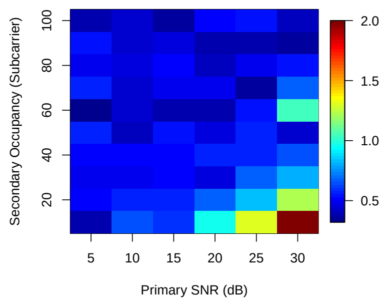
The next set of results show the performance of the technique when the primary SNR is varied. Figure 5.12(a) shows that when the relative SNR of the secondary is higher than that of the primary, the average error is maintained below 1 sample. Only when the primary tends to overwhelm the secondary transmission the error is greater. It is to be noted that when the primary signal is stronger than the secondary by more than $20dB$, it represents a very hostile situation for secondary operation and can be considered as the worst possible case. Figure 5.12(b) shows that when the secondary SNR is held constant, the error decreases with an increase in the number of secondary spectral bands (because correlation is performed across more subcarriers). Similarly, with the number of secondary bands remaining constant, the error decreases with an increase in the bandwidth of the secondary (**i.e.** more subcarriers) as shown in figure 5.12(c). In all the cases, the worst case performance is ≤ 2 samples of synchronization error.



(a) SNR_p vs SNR_s



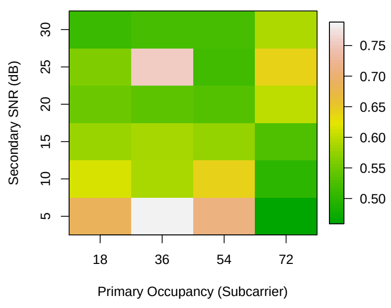
(b) SNR_p vs $NBand_s$



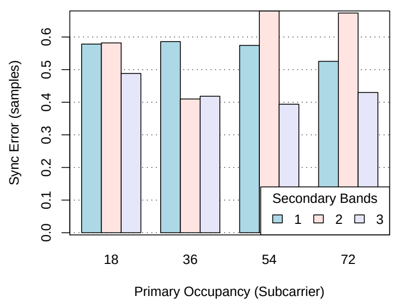
(c) SNR_p vs NSC_s

Figure 5.12: Average synchronization error (in samples) with increasing primary SNR

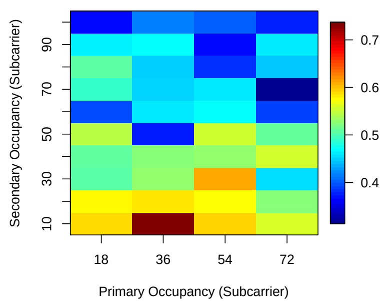
5.2.4.3 Varying primary occupancy



(a) NSC_p vs SNR_s



(b) NSC_p vs $NBand_s$



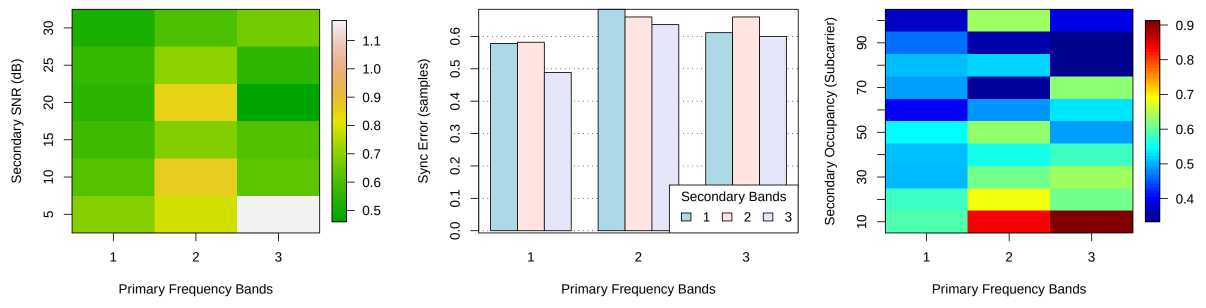
(c) NSC_p vs NSC_s

Figure 5.13: Average synchronization error (in samples) with increasing primary occupancy

As the width of the primary signal increases it tends to produce inter-carrier interference and starts to bleed into adjacent carriers, which could potentially be a secondary band. Figure 5.13(a) and 5.13(c) shows reduction in average error with increase in secondary SNR and occupancy, respectively. This trend is also seen with an increasing number of secondary bands in figure 5.13(b) as the number of secondary subcarriers involved in correlation is more.

5.2.4.4 Varying number of primary bands

In this experiment the effect of multiple primary links in the network on secondary links is studied. In figure 5.14(a), the error magnitude is measured by varying the primary SNR. Understandably, when 3 independent primary links are present the correlation performs worst at low secondary SNR. In contrast, when secondary occupancy increases or more number of disjoint bands are used then the correlation improves as shown in figure 5.14(b). The trend continues in figure 5.14(c) where a wider secondary band gives the best performance while a narrow band secondary is subdued by multiple primary links.



(a) $NBand_p$ vs SNR_s (b) $NBand_p$ vs $NBand_s$ (c) $NBand_p$ vs NSC_s

Figure 5.14: Average synchronization error (in samples) with increasing primary bands

,l

An interesting observation from these experiments is that the synchronization performs well even with errors in estimating the secondary band edges. This is primarily attributed to the good correlation properties of the preamble chosen for these experiments. Hence, the quality of synchronization not only depends on the secondary subcarriers, it also depends on the cross-correlation properties of the preamble. In order to achieve correct band-edge detection in multi-path fading channels, we propose a channel sounding and re-transmission techniquewhich, in conjunction with the preamble properties, make this approach practical for cognitive radio network. The experiments and the results discussed herein, provide a key insight into the trade-offs that can be exploited, while deploying cognitive radio networks. It is the wise selection of transmission parameters that spans across multiple dimensions of SNR, band width and number of non-

Table 5.3: NC-OFDM correlator utilization in Virtex - IV

Parameter	Count / Max	Utilization
Slices	$2900/15360$	18.80%
Slice Flip Flops	$1950/30720$	6.35%
4 input LUTs	$3309/30720$	10.77%
FIFO16/RAMB16s	$4/192$	2.08%

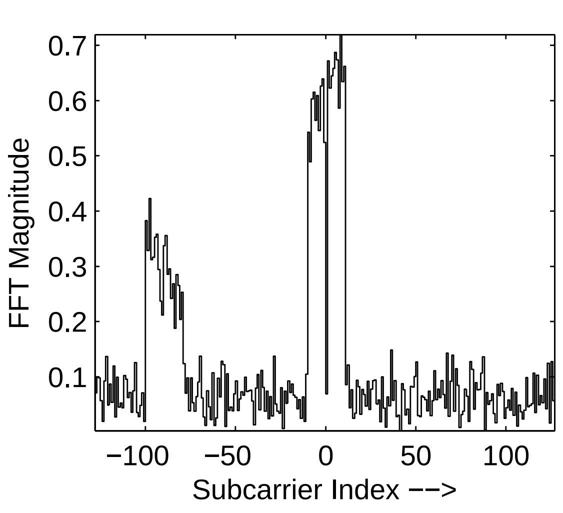
contiguous subcarriers that will provide the optimum performance in a network. We acknowledge that this problem requires further research and this paper is a positive step towards setting the foundation for such future work.

5.2.4.5 Hardware Implementation

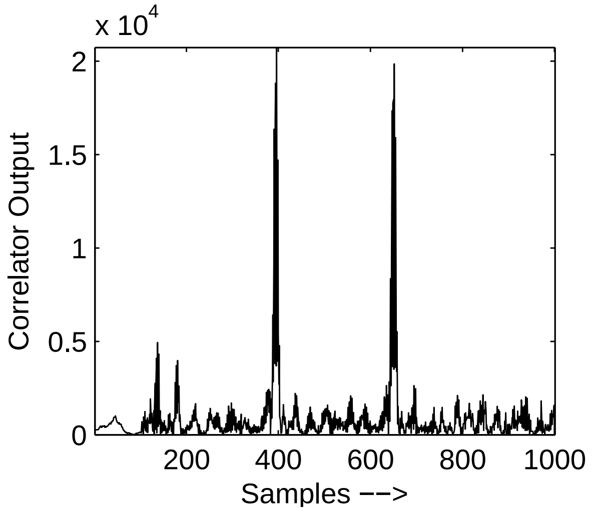
We have implemented a prototype correlator in FPGA and tested it with a simulated channel, due to the unavailability of a wideband front-end. Figure 5.15 shows the input and output traces of the correlator for a NC-OFDM waveform. In figure 5.15(a), the input spectrum occupies subcarriers $[-68 : -17]$ and subcarriers $[44 : 69]$. The spectrum detection unit detects this and programs the correlator registers with the spectral mask to regenerate the preamble. The **sign** bit and **XNOR** based correlator output is shown in figure 5.15(b). Since we have used two preamble symbols to synchronize, we get two distinct peaks that are detected using a simple threshold test. These encouraging results motivate us to perform over-the-air experiments using wideband front-ends, which we leave as a future extension of this work. The design has been synthesized for a Virtex-IV FPGA, and the utilization is given in table 5.3, which includes preamble regeneration using an IFFT core. The correlator design and implementation is a step toward designing realistic cognitive radio hardware and we are actively working in implementing other transceiver components to work as a wideband cognitive radio using NC-OFDM.

5.2.5 Conclusion

In this work we have proposed a set of techniques that can make NC-OFDM transmissions more practical by addressing the synchronization challenges involved. Simple detection technique coupled with programmable radio hardware can lead to fast channel rendezvous between secondary nodes without the



(a) Non-Contiguous spectrum



(b) Correlator output

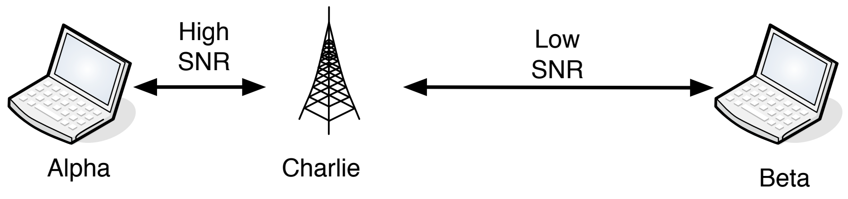
Figure 5.15: Programmable correlator input and output

requirement of a dedicated control channel. We have also validated our approach using realistic simulations that shows the synchronization technique outperforms existing solutions by order of magnitude and is robust against changing channel conditions. While much work remains to be done, this research shows a new approach towards solving one of the most challenging problems in deploying cognitive radio networks.

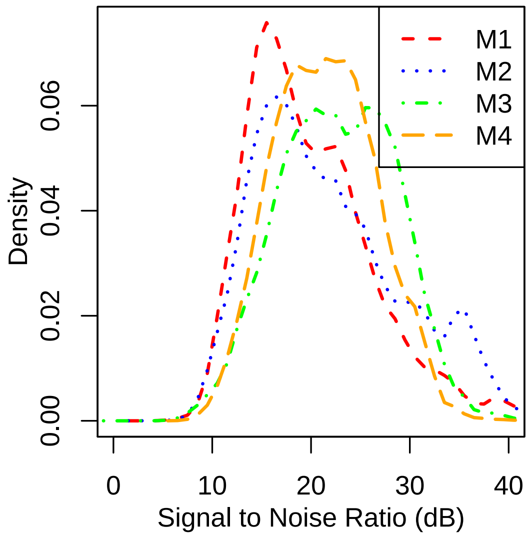
5.3 GRaTIS – Free Bits in the Air [100]

In this work, we show that it is possible to exploit the higher SNR of this primary node to encode another message for a second receiver, increasing the aggregate network bandwidth. In doing so we introduce multiple data rates to provide an even gradation of SNR across a group of receivers – we call this Group Rate Transmission with Intertwined Symbols, or GRaTIS. For example, consider the network organization in figure 5.16(a). If the link between Charlie and Beta has an SNR of 10dB, and the link between Charlie and Alpha has an SNR of 20.5dB, using GRaTIS we can send an 18Mb/s message to one node **and simultaneously send another** 18**Mb/s message to a second node resulting in an effective throughput of** 36**Mb/s**.

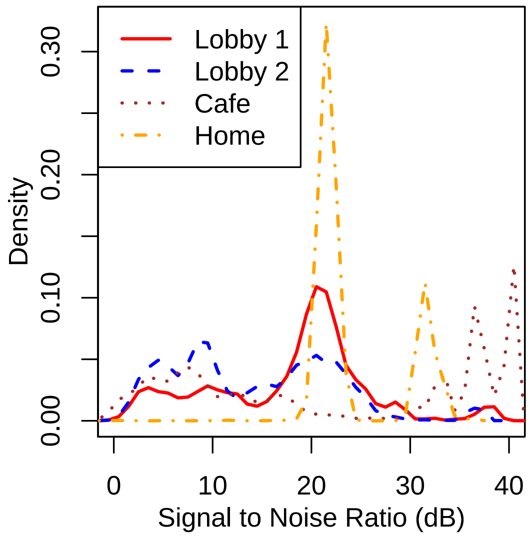
Our method depends on **SNR diversity** between the receivers – in other words, we exploit the fact



(a) Example Of Spatial Rate Diversity



(b) SNR Profile 1



(c) SNR Profile 2

Figure 5.16: Variation of SNR due to spatial diversity in 802.11a/g networks. Profile 1: Measured indoors by 4 packet sniffers at SIGCOMM 2008 [101] Profile 2: Measured indoors in common areas around a university cafe and lobbies and also in home networks.

that most networks have nodes that experience different SNR values. Fortunately, most networks exhibit considerable SNR diversity. Figure 5.16(b) shows the distributions of SNR at a number of locations measured at a SIGCOMM conference in 2008 and figure 5.16(c) shows the same variation, which are measured by us around a university campus. Figure 5.16(c) also shows a SNR profile that is typical to a home network shared by two users with high volume video streaming. The diversity in SNR occurs because of the spatial layout of nodes, room geometries and interference from other sources.

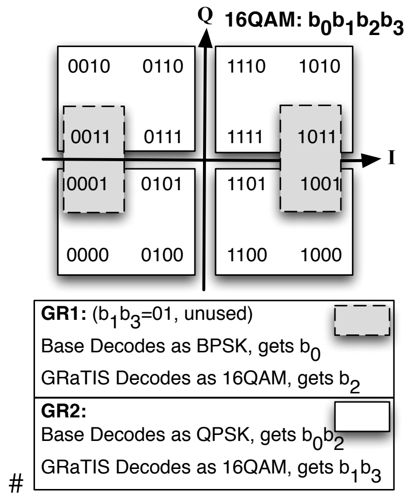
The key contributions of this research as follows:

- We reinterpret the constellations already available for conventional wireless links and provide **group rates**, which result in higher network throughput with no hardware changes.

- We perform a standard analysis of packet error rates for this scheme to ascertain the applicability in real networks.
- We implement GRaTIS on a 802.11a/g compatible software defined radio (SDR) prototype to show that the technique is easy to implement and makes use of existing hardware modulation and demodulation methods. In the prototype, much of the added processing is handled by simple software, rather than complicated fixed-function hardware and DSP algorithms.
- We use the SDR nodes in a testbed setup to measure the SNR requirement for over-the-air transmission of GRaTIS encoded multiuser data packets.
- We apply the results of the testbed to analyze various 802.11a/g traces from SIGCOMM conferences and other small/mid sized wireless data networks to determine how the diverse range of SNR of the receivers can be used to obtain substantial gain in network throughput. This shows the potential performance improvement when GRaTIS is applied on realistic downlink traffic.
- We also show that GRaTIS outperforms a competing method (superposition coding) both in theory and practice. We validate this through experiments conducted in a testbed of SDR nodes using actual over-the-air packet transmissions.

5.3.1 GRaTIS: Free Bits

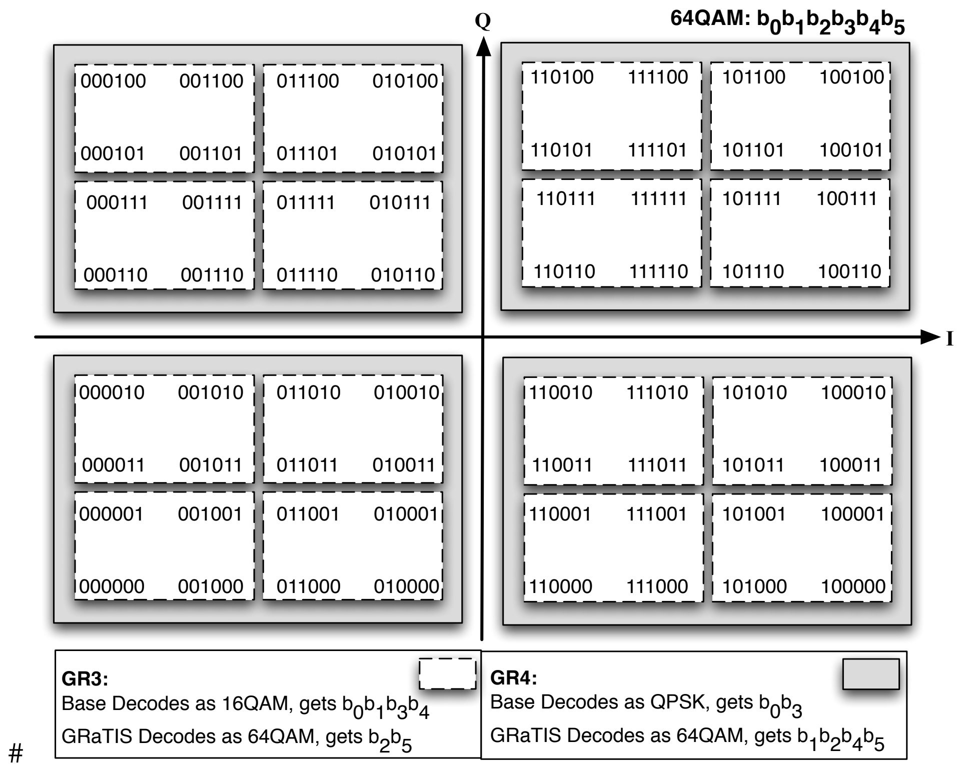
In this section, we describe GRaTIS. Discrete steps in the SNR requirements for each data rate forces the rate adaptation algorithm to fall back to a lower rate even if the node is reachable at a SNR higher than the minimum required. In this scenario, we use GRaTIS to identify two "layers" of constellations within the standard constellations available in 802.11a/g. These two layers are used to map two packets of two different users to form one single packet, such that the time required to transmit the merged packet in GRaTIS is less than the time required to transmit two separate packets in the best achievable data rate of 802.11a/g. The two layers are designed in such a way that one of the layers can be decoded by a legacy decoder, and we call it the **Base Layer**. The second layer is obtained by extracting a few bits after the legacy demodulation system



Q
16QAM: $b_0b_1b_2b_3$
0010 0110 1110 1010
0011 0111 1111 1011
0001 0101 1101 1001
0000 0100 1100 1000
I
GR1: (b_1b_3=01, unused)
Base Decodes as BPSK, gets b_0
GRaTIS Decodes as 16QAM, gets b_2
GR2:
Base Decodes as QPSK, gets b_0b_2
GRaTIS Decodes as 16QAM, gets b_1b_3
#

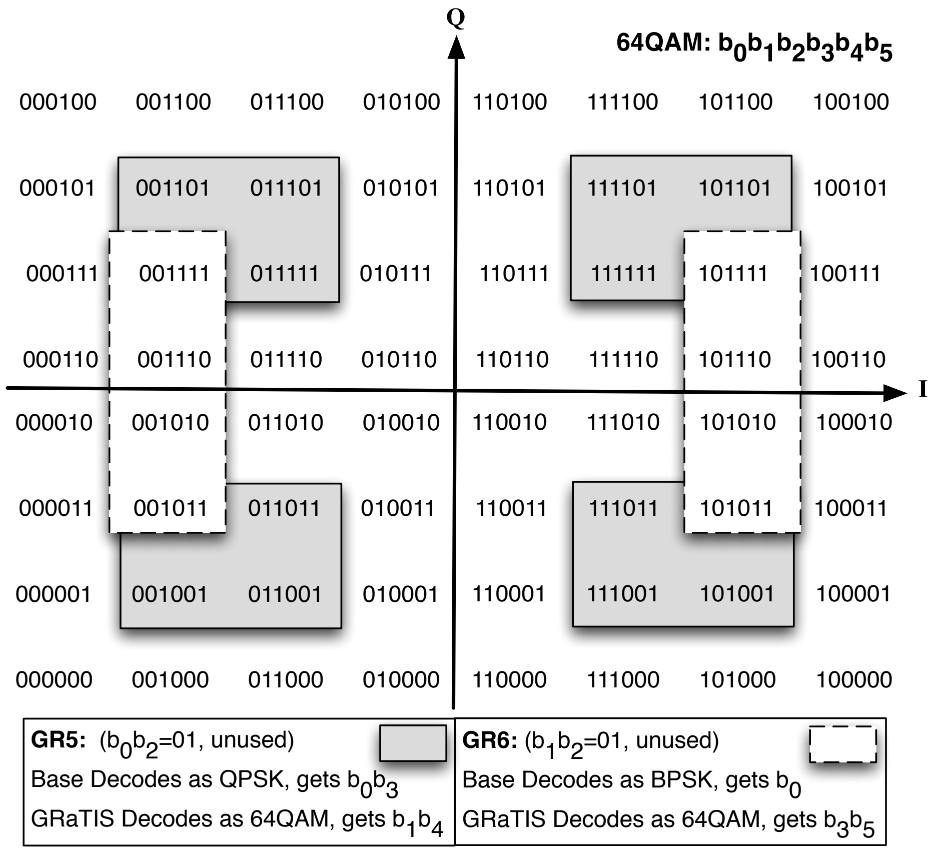
(a) GR1 and GR2

Figure 5.17: Encoding and decoding of GRaTIS derived from standard 802.11a/g constellations. Rectangular regions show the transmitted cluster and the corresponding decision boundary for the base layer.



(b) GR3 and GR4

Figure 5.17: Encoding and decoding of GRaTIS derived from standard 802.11a/g constellations. Rectangular regions show the transmitted cluster and the corresponding decision boundary for the base layer. (continued)



#

(c) GR5 and GR6

Figure 5.17: Encoding and decoding of GRaTIS derived from standard 802.11a/g constellations. Rectangular regions show the transmitted cluster and the corresponding decision boundary for the base layer. (continued)

converts the I/Q samples from the analog domain to the binary domain, and we call it **GRaTIS Layer**. The second packet is transmitted during the transmission of the first packet, without any extra airtime, and comes as free bits to the receiver with a higher SNR – those free bits increase the aggregate throughput of the network.

For example, assume the received SNR of two nodes n_1 and n_2 from a common transmitter are SNR_{n_1} and SNR_{n_2} respectively. Also, there exists a GRaTIS rate, where the SNR requirement for Base and GRaTIS layers are SNR_b and SNR_g respectively, and $SNR_b <= SNR_{n_1}$ and $SNR_g <= SNR_{n_2}$. Consider the best achievable data rate in 802.11a/g are R_{n_1} and R_{n_2}, while that using GRaTIS are R_b and R_g respectively. The common transmitter uses GRaTIS to transmit x bits of data at rate R_b to node n_1, which takes time t_g (total transmission time using GRaTIS). Since in GRaTIS, the GRaTIS layer is transmitted at the same time along with the Base layer, there is no extra time required to transmit the GRaTIS layer. So, the transmitter also transmits y bits of data using rate R_g within the same time t_g. Therefore the total data rate of this transmission is $\frac{(x+y)}{t_g}$.

Now we calculate the achievable data rate if the common transmitter uses 802.11a/g to transmit the same packets. The transmitter uses rate R_{n_1} to transmit the x bits of data to node n_1 in time t_1. After this, it transmits y bits of data at rate R_{n_2} in time t_2. The aggregate data rate for these two transmissions is $\frac{(x+y)}{t_1+t_2}$. The pair of rates R_b and R_g are selected as one of the GRaTIS rates, **iff** $\frac{(x+y)}{t_1+t_2} < \frac{(x+y)}{t_g}$. Or in other words, GRaTIS rates are selected only if there is potential gain in aggregate throughput over the legacy system.

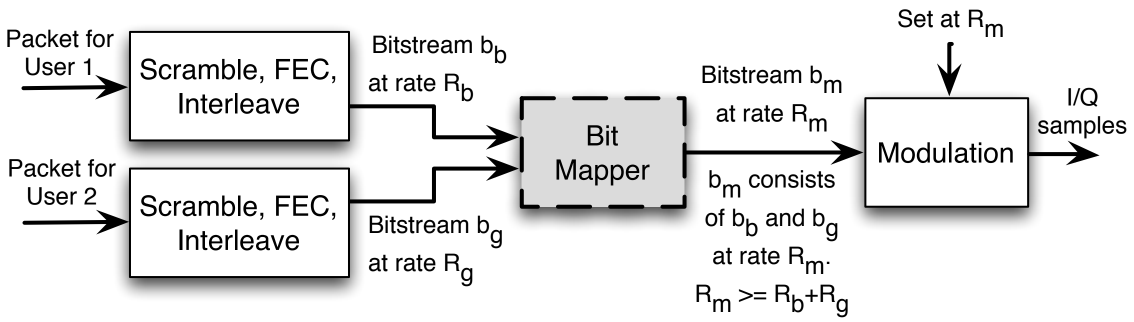
5.3.2 Hardware Implemention of GRaTIS

We introduce **six** distinct GRaTIS rates, or methods of combining packets, as shown in figure 5.17, to increase aggregate throughput of the network. The GRaTIS rates are termed **GR1** through **GR6**.

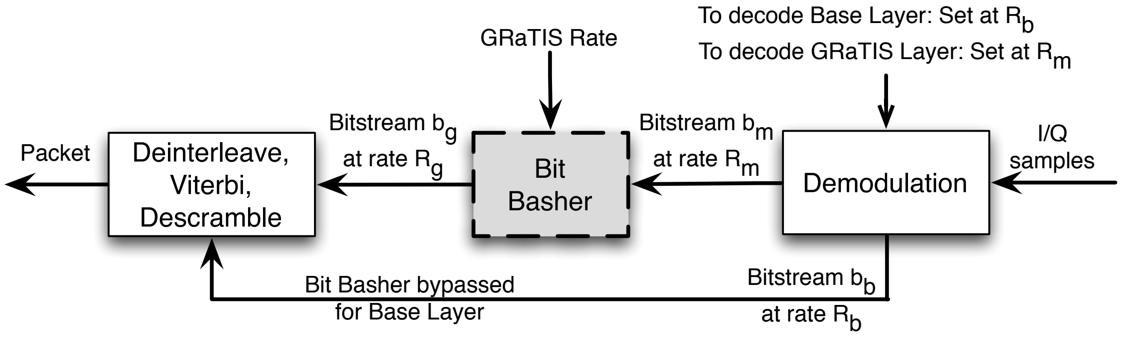
At the transmitter, two packets are encoded independently up to the modulation subsystem as shown in figure 5.18(a). Then the bits of two packets, b_b and b_g, are encoded at rates R_b for the base and R_g for the GRaTIS layer respectively, and combined to form a compound symbol that represents one of the constellation points corresponding to a standard modulation in 802.11a/g, denoted by R_m. This mapping ensures that the compound symbols are mapped only to the I/Q vectors that are part of a selected GRaTIS cluster. In this way, the modulator remains unchanged, as it is fed with the compound bitstream (b_m), and modulation type (R_m) to which it modulates. Since all the packet merging is done at the bit-level it does not require any change in the signal processing pipeline.

Figure 5.18(b) shows the demodulation pipeline for a GRaTIS compatible node. The GRaTIS layer is first decoded as the constellation from which the GRaTIS cluster has been derived (R_m). Then, the bits designated for the GRaTIS layer are extracted using the **GRaTIS rate** information, to form the bit stream for the second packet (b_g) at rate R_g. The rest of the receiver decode pipe remains unchanged.

The theoretical and testbed results are listed in Table 5.4, which shows the SNR requirements for a 2% packet error rate (PER) along with the SNR requirement for the 802.11a/g standard rates. The SNR


Packet for User 1
Scramble, FEC, Interleave
Packet for User 2
Scramble, FEC, Interleave
Bitstream b_b at rate R_b
Bitstream b_g at rate R_g
Bit Mapper
Bitstream b_m at rate R_m
b_m consists of b_b and b_g at rate R_m. $R_m >= R_b+R_g$
Set at R_m
Modulation
I/Q samples

(a) Transmitter Pipeline


Packet
Deinterleave, Viterbi, Descramble
Bitstream b_g at rate R_g
GRaTIS Rate
Bit Basher
Bitstream b_m at rate R_m
To decode Base Layer: Set at R_b
To decode GRaTIS Layer: Set at R_m
Demodulation
I/Q samples
Bit Basher bypassed for Base Layer
Bitstream b_b at rate R_b

(b) Receiver Pipeline

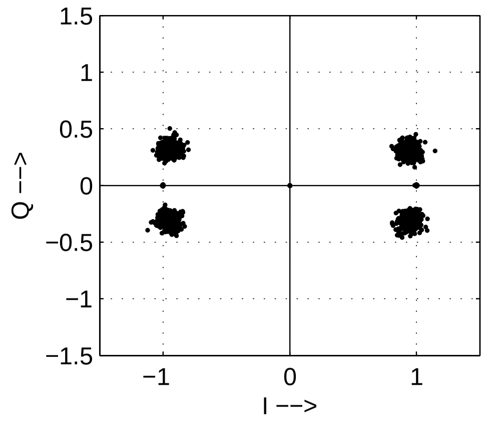
Figure 5.18: Transceiver pipeline for GRaTIS – shaded subsystems show additional processes required for GRaTIS.

- throughput relationship for various group rates are used as a look-up while downlink packets are being considered to be merged. As shown in Table 5.4, the group rates provide a variety of step-down rates while utilizing SNR diversity in the network to increase the aggregate throughput of the network.

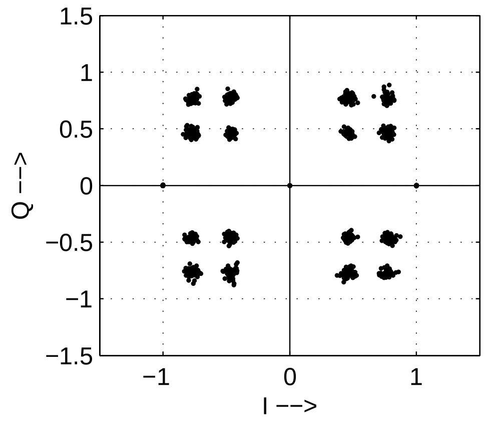
Figure 5.19 shows the various constellations produced by the prototype SDR that support multiuser communication using GRaTIS. Figure 5.19(a) shows the constellation for GR1, which is derived from a 16QAM constellation by not using the other constellation points. Similarly, figure 5.19(b) and 5.19(c) shows a modified 64QAM constellation that provides a combined network throughput of 4 bits/OFDM subcarrier (2 for base and 2 bits for GRaTIS layer) and 3 bit/OFDM subcarrier (1 for base and 2 bits for GRaTIS layer) respectively.

5.3.3 Conclusion

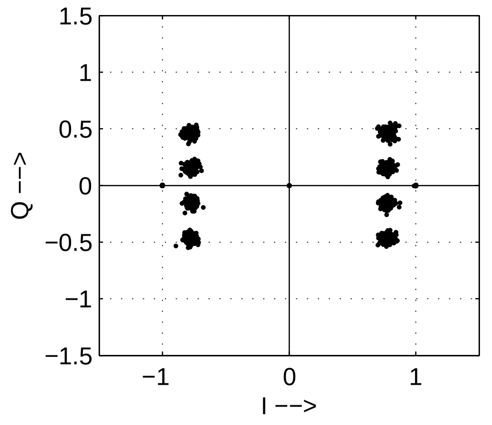
GRaTIS provides an efficient method of simultaneous packet transmission and reception. This in-



(a) GR1



(b) GR5



(c) GR6

Figure 5.19: Various GRaTIS constellations transmitted using the SDR prototype.

Table 5.4: Throughput and SNR requirements for 802.11a/g and CODIPHY rates

Mod	CR[1]	Data Rate (Mbps)		SNR(dB)			
				Base		GRaTIS	
		Link	Grp	Th[2]	Exp	Th[2]	Exp
BPSK	1/2	6	n/a[3]	3.0	4.5	n/a[3]	n/a[3]
BPSK	3/4	9		5.0	6.0		
QPSK	1/2	12		6.0	7.0		
QPSK	3/4	18		8.0	9.0		
16QAM	1/2	24		12.5	13.0		
16QAM	3/4	36		17.0	18.0		
64QAM	2/3	48		19.5	24.0		
64QAM	3/4	54		21.0	26.0		
GR1	1/2	6	12	3.5	5.0	17.0	17.0
GR6	1/2	6	18	4.5	5.0	18.5	21.0
GR5	1/2	12	24	7.5	7.5	18.5	23.0
GR2	1/2	12	24	9.0	10.5	11.5	15.0
GR4	1/2	12	36	15.0	15.0	19.0	23.0
GR3	1/2	24	36	17.0	16.5	20.0	25.0
GR1	3/4	9	18	5.5	n/i[4]	19.5	n/i[4]
GR6	3/4	9	27	7.0		20.5	
GR5	3/4	18	36	10.0		20.5	
GR2	3/4	18	36	13.0		14.5	
GR4	3/4	18	54	19.5		21.5	
GR3	3/4	36	54	20.0		22.0	

[1]Coding Rate, [2]Theoretical, [3]Not Applicable, [4]Not Implemented

creases the network throughput without compromising the throughput of one node while using widespread channel variability to simultaneously transmit an independent packet destined for another node. The GRaTIS packet is indeed extra free bits to the high SNR node, which it would have received after the completion of the first packet using a serialized medium access pattern as in 802.11a/g. We have implemented the protocol in hardware and have shown the ease of implementation if the signal processing is done using a hybrid platform of software and hardware components. The experimental results show several possibilities of use of GRaTIS giving unforeseen gains in throughput in wireless networks. Applying GRaTIS on real-time packet trace analysis reveals that even with a few simple combinations, we can gain significant airtime, which can be further utilized to transmit more packets. Also through our analysis we show that GRaTIS provides more flexibility with better error performance than other contemporary simultaneous packet transmission techniques, making it a suitable candidate for emerging wireless networks.

5.4 Covert Communication through Dirty Constellations [102]

In this work, we explore methods that provide LPD and LPI for high-bandwidth networks. Our method provides a high-bandwidth covert side-channel between multiple radios using a common wireless network. The method is covert because the devices (laptops or smartphones) function as normal devices. Again, the devices “hide in plain sight”. Rather than raising suspicions by exchanging encrypted messages with each other or some centralized server, they appear to be conducting normal network communication (browsing web pages, sending mail, streaming multimedia) when in reality, they are able to communicate undetected. The adversary will face great challenge in discovering the side channel because the covert channel is being transmitted by mobile nodes. Monitoring to locate such nodes would require significant investment or infrastructure, such as monitoring in every coffee shop, bus or public venue where people may be near each other.

The technique uses a common, physical-layer protocol to mask the communication that takes advantage of the hardware imperfections present in commodity hardware, intrinsically noisy channel of wireless communication and receiver diversity. We have implemented this mechanism using software-defined radios, operating in 2.4GHz ISM band, but can also be easily extended to TV whitespaces. Our prototype uses an OFDM waveform. Most consumer electronic devices use OFDM waveforms for high-bandwidth networks (including DVB, DAB, WiFi, WiMAX and LTE), and there are some benefits in “hiding” in such a ubiquitous waveform.

Imperfections in off-the-shelf Network Interface Cards (NICs) [103], coupled with an additive random wireless channel cause the signal to degrade over time and distance. To mask our communication, we “pre-distort” the signal to mimic the normal imperfection of the hardware and Gaussian distortion arising from the channel. This distortion appears as noise to the unobservant receiver, be it the Wi-Fi access point or an adversary. However, a receiver aware of the presence of the signal and its encoding technique can decode the “noise” to reveal the hidden message.

Our motivation for hiding the data in physical layer (analog waveform domain) of common wired and wireless protocols are the following:

- **Hide in plain sight** - Using the physical properties of the transmission medium will allow the covert channel to resemble a common waveform, only distorted by channel noise, or transmitted by a NIC with imperfections.

- **Access to covert channel** - Since the covert channel uses the signal waveform, an adversary is easily abstracted from the covert channel, as opposed to other packet level techniques using higher layers [104]. In our method, the bits of the cover packet are not altered and hence the presence of the covert message is not detected at higher layers, or more specifically in digital domain.

- **Sample collection** - The ubiquitous nature of wireless devices and their localized transmission make it difficult to detect the presence of a covert channel. As opposed to digital contents on the Internet (music, picture, video), which can be accessed from one physical location, acquiring signal waveforms requires hauling expensive, bulky equipment (signal analyzers) to every possible hotspot.

- **Search complexity** - A $500byte$ packet, modulated with QPSK-$1/2$ rate coding, results in $\approx 19KB$ (calculation omitted due to space constraints) of I/Q information. This increases the search space by ≈ 38 times, compared to packet level analysis of a covert channel.

- **Statistically Undetectable** - In higher layer techniques, an adversary can search the header fields (known as unused fields) of a packet stream and find the covert channel [105], whereas in physical layer, the adversary needs to perform several statistical tests on the I/Q samples, which are already tainted by time varying channel noise.

- **Capacity** - Compared to conventional techniques using higher layers, where only a few unused bits of any header field of a packet is used, our technique can easily utilize 10% of the cover signal to transmit covert messages.

These advantages coupled with relative ease of implementation using now popularized software defined radio, makes this technique extremely useful in providing high capacity covert channels.

5.4.1 Dirty Constellation

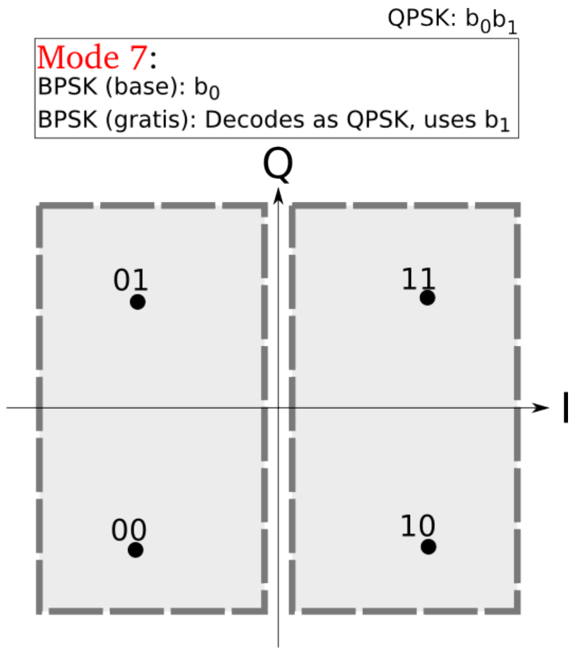
Our method relies on being able to embed one message in another in the wireless channel, but goes well beyond that to then insure that the covert message is undetectable. There are several ways to embed messages by encoding the constellation symbols using bits of two distinct messages [56, 106] but we use a simpler technique that uses existing modulation methods of OFDM.

Using a combination of adaptive modulation and efficient packet sharing using joint constellations we encode the covert channel. If a receiver is aware of our irregular mapping of bits, and it has sufficient SNR for that subcarrier, it is able to decode the covert message while to an uninformed user, the covert constellation points will be treated as random dispersed sample of a low-rate modulation, that reveals an innocuous message.

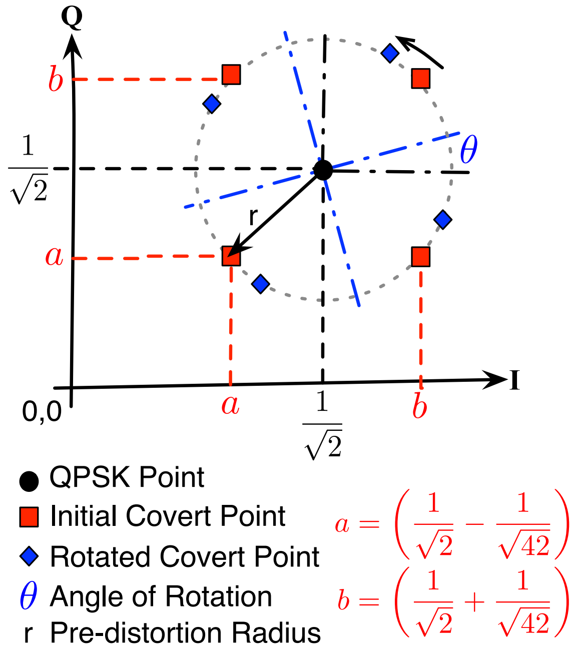
The key to such covert communication using the physical layer of an OFDM based wireless protocol are four fold: **1)** packets containing covert data must be indistinguishable from non-covert packets to all uninformed observers; **2)** the presence of any irregularity in the covert packets has to be kept hidden under rigorous **statistical tests** on the signal; **3)** the covert channel should be non-trivial to replicate, making it secure from spoofing and impersonation; and finally, **4)** it should have high capacity. In this work we satisfy each of these requirements through a set of techniques.

Requirement 1: Identifying a Covert Channel: Our technique relies on encoding "cover packets" that are transmitted at a low rate (BPSK or QPSK) with supplemental information that can be decoded as an additional QPSK signal by an informed receiver. In the examples below, we use QPSK for both the cover and covert channel.

In a QPSK encoding, the constellation points encode two bits of information as shown in Figure 5.20(a). To encode the covert channel, we deflect the placement of the QPSK points. This is similar to having a "covert QPSK" encoding with an origin around the ideal QPSK constellation points of the cover traffic. Figure 5.20(b) corresponds to the upper right quadrant of the cover QPSK constellation shown in Figure 5.20(a). To modulate a subcarrier carrying both the cover and covert message, first the cover constellation point (QPSK) is chosen (as per the cover message stream), specifying the quadrant, followed by



(a) Cover QPSK constellation



Q
I
b
a
$\frac{1}{\sqrt{2}}$
0,0
θ
r
● QPSK Point
🟥 Initial Covert Point
◆ Rotated Covert Point
θ Angle of Rotation
r Pre-distortion Radius
$a = \left(\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{42}}\right)$
$b = \left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{42}}\right)$

(b) Covert constellation

Figure 5.20: Encoding Dirty Constellation

re-mapping that point to one of the four "covert-QPSK" points around the "cover QPSK" point.

Clearly, the goal is to leave the cover message decodable by standard receivers. Only the covert receiver aware of the joint constellation will decode the subcarriers properly and extract the **two** covert bits to form the hidden packet. An adversary will decode at the base rate or the rate for cover message, as specified in the **signal symbol** of the packet; while the covert points will be treated as noisy points. The cover message could be intended for an access point (as part of a web browsing session) while the covert message can be overheard and decoded by a nearby radio. In this way we implement a covert channel while making it appear as completely innocuous to other users receiving the same transmission.

Requirement 2: Low Probability of Detection: How would an adversary detect such communication? As long as the packet can be decoded, a legacy receiver has no way of knowing how signals are being encoded at the core of the physical layer, because conventional packet decoding is performed by identifying the data rates embedded at the beginning of the packet which will always contain the base rate (QPSK) information. However, adversaries using measurement equipment like vector-signal analyzers or software defined radios can extract the digital samples from the radio pipeline at different stages of the signal processing. Therefore, our ultimate goal is to provide very low probability of detection not only at the packet level but also at the signal level.

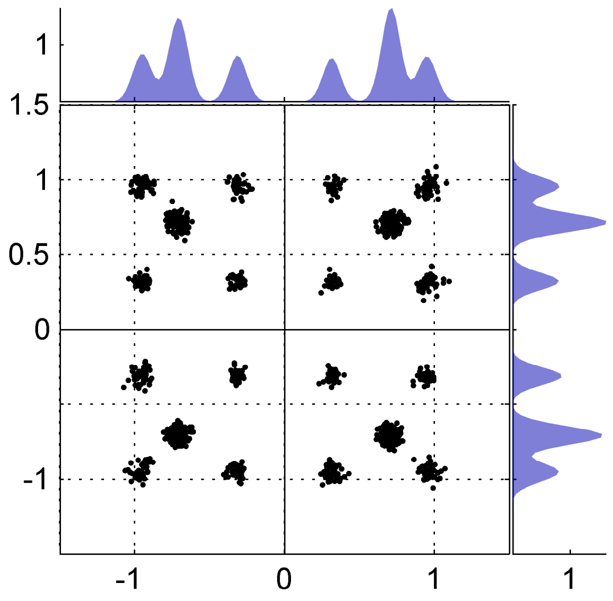


Figure 5.21: Constellation without random pre-distortion of the QPSK points and using existing 16QAM points to map the joint covert constellations.

One simple form of analysis is to look at the equalized I/Q vectors of the jointly encoded packet. The presence of the covert constellation at regular interval will appear as distinct point clouds that will set themselves apart from the cover QPSK point cloud and will reveal the presence of the covert channel, as shown in Figure 5.21.

We solve this problem by changing the I/Q vectors of the covert transmitter in three steps:

Step1: We bring the covert constellation points closer to the ideal QPSK point and re-map the covert constellation points symmetrically around the QPSK points, with a mutual separation of $\frac{2}{\sqrt{42}}$, a distance equal to that of a 64QAM constellation, so that a covert receiver can operate within the operating range of a WiFi receiver.

Step2: We randomize the I/Q vectors of the covert QPSK points with a Gaussian distribution but limit their dispersion to a radius of $\sqrt{\frac{2}{42}}$ as shown in figure 5.20(b). We call this as the **pre-distortion circle**;

pre-distortion of the QPSK signal at the transmitter ensures that the covert constellations are hidden in the cloud of a dispersed (noisy) QPSK point cloud. We introduce imperfections to the transmitted signal in such a way that the average EVM error is equal to or less than 10dB compared to the ideal QPSK constellation points, which is within the limits of hardware anomaly allowed in the IEEE 802.11 standard [79]. Thus, it cannot be ascertained with certainty if the EVM error is due to hardware impairments, channel impairments or intentionally injected distortion.

Step3: To accommodate a higher rate covert channel, e.g., when 50% of the OFDM subcarriers are covert, then at high SNR there is always a finite probability that the covert constellations are visible. To have the covert symbols blend with the pre-distorted QPSK point cloud, the covert symbols are rotated along the circumference of the pre-distortion circle for every subcarrier that is mapped to a covert constellation as shown in Figure 5.20(b). The rotation is performed using a monotonically increasing angle θ; the transmitter and receiver both start with $\theta = 0°$ at the start of the packet and increment θ for each covert subcarrier. In our implementation we use a $15°$ counter-clockwise rotation for the covert points.

These 3 steps allow us to hide the covert channel, even when an adversary has access to the I/Q samples of the packet. The adversary will interpret the point cloud as a noisy version of a valid (albeit noisy) QPSK constellation and would not suspect the presence of a covert communication. This compound constellation involving a covert channel hidden within a cover constellation is termed a **"Dirty Constellation"**. However, in order to avoid raising suspicion by any RF fingerprinting algorithms [103], a QPSK-IM waveform should **always** be used for non-covert transmissions, to avoid sudden changes in the modulation characteristics.

Requirement 3&4: Security and Higher Efficiency: These requirements are considered as an enhancement to the basic scheme of Dirty Constellation. We have implemented 10%, 30% and 50% encoding of subcarriers, as shown in Figure 5.24, yielding up to $9Mbps$ datarate with QPSK modulation and $3/4$ encoding rate. Using higher modulation constellation, e.g., 256-QAM, we can further increase the capacity of the covert channel by encoding more bits per subcarrier. Due to space constraints we leave this as future work.

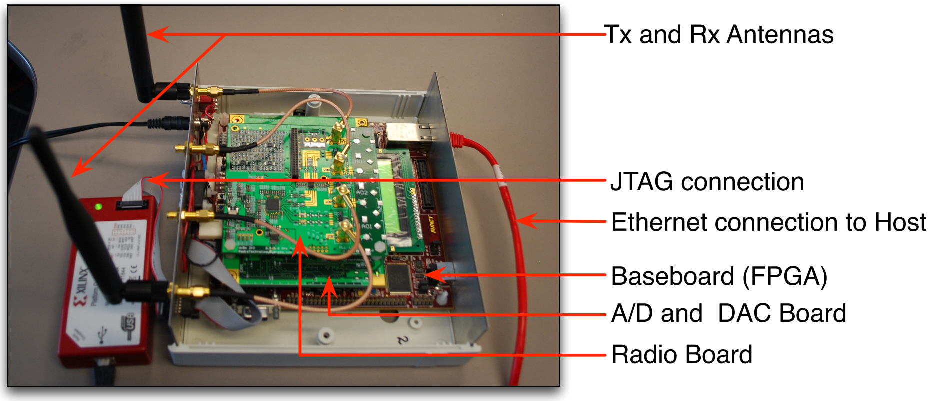
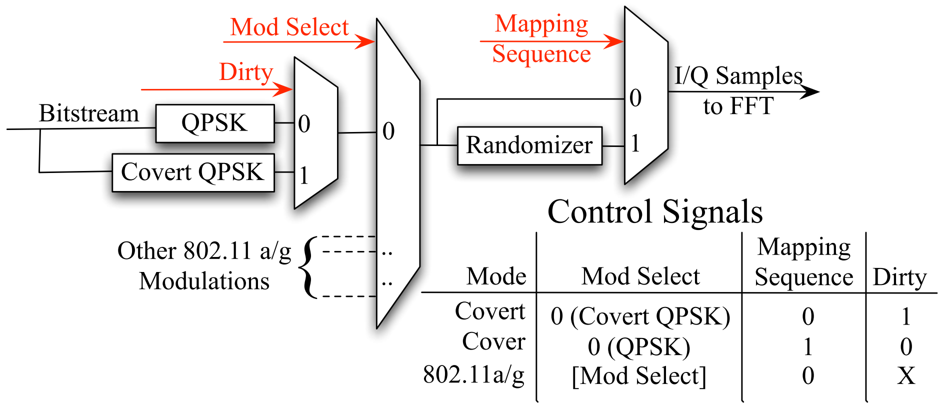


Figure 5.22: SDR prototype using Virtex-V FPGA

5.4.2 Dirty Constellation on SDR

Hiding a message in a randomized modulation constellation requires a programmable modulator and demodulator. Conventional radio 802.11 PHYs modulate all the subcarriers with one type of pre-defined modulation, For this scheme to work, we used a FPGA-based software defined radio platform based on our previous work [74, 75], as shown in figure 5.22, and modified the modulator and demodulator to program each subcarrier with different modulations, adding either noise or covert constellations. Figure 5.23(a) shows the functional diagram of the programmable modulator. The notable parameters in the design are the **dirty** bit and the **mapping sequence** bit which are used to select the appropriate mapping for covert joint constellations and randomize (Gaussian) the cover symbols to engulf the higher order modulation points. The cover and the covert bits are independently packetized as per the 802.11a/g specification and the covert joint symbols are formed by merging the bits of the two packets prior to sending it to the modulator. The merging of packets is performed in software and then fed to the hardware along with the control information to create the Dirty Constellation. The QPSK-IM constellation is generated by using the randomizer unit that emulates an overall modulation error of 10dB, by setting the **dirty** bit to ‘0’ and **mapping sequence** to ‘1’ for all subcarriers.

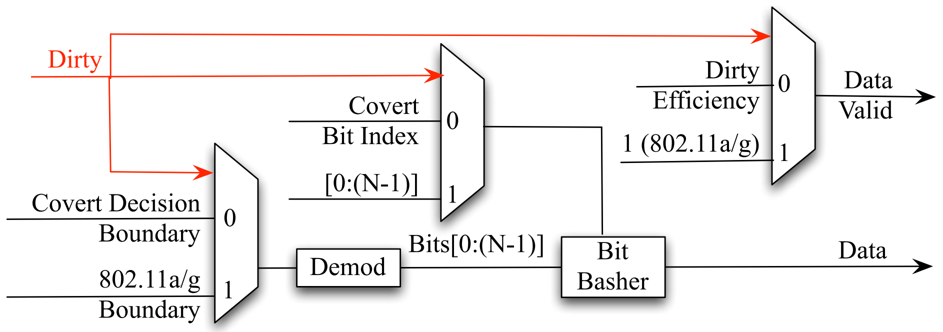
The decoder employs maximum likelihood decoding and uses pre-defined thresholds to decode the constellation. Figure 5.23(b) shows the functional diagram of the demodulator. First the covert receiver demodulates the signal using the covert decision boundaries, 64QAM in this case and then extracts the



Control Signals

Mode	Mod Select	Mapping Sequence	Dirty
Covert	0 (Covert QPSK)	0	1
Cover	0 (QPSK)	1	0
802.11a/g	[Mod Select]	0	X

(a) Transmitter Pipeline



(b) Receiver Pipeline

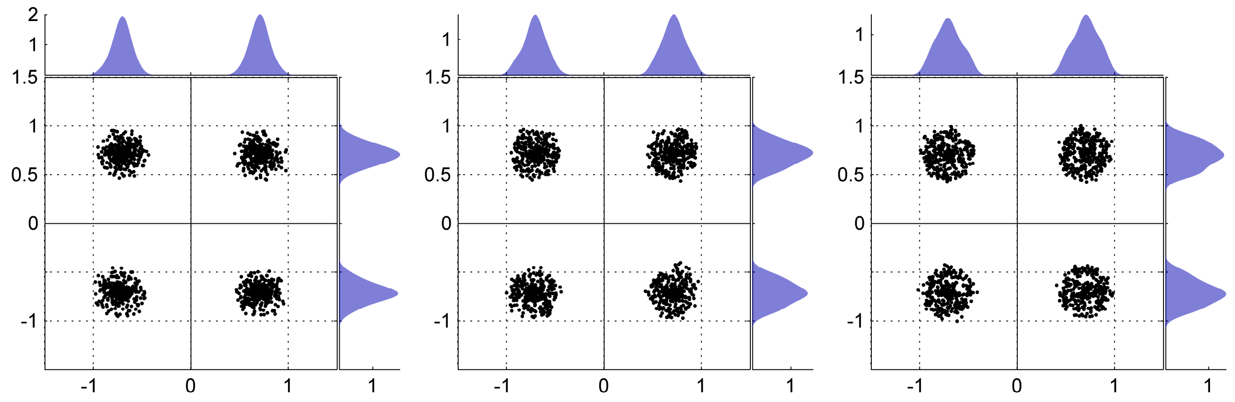
Figure 5.23: Mod/Demodulator for Dirty Constellation

covert bits. Since all subcarriers do not contain the hidden message, the receiver then uses the pre-assigned mapping sequence and its rotation information to filter out the covert subcarriers' information to form the covert packet.

Figure 5.24 shows an example of Dirty Constellation with varying frequency of the covert channel that has been transmitted by the SDR prototype and captured using a VSA. The I and Q histograms alongside the constellation shows the similarity of the distributions and that they are from the family of normal distributions.

5.4.3 Conclusion

In this work, we discussed a technique to implement a covert channel at the physical layer of 802.11a/g wireless protocol. By hiding the covert channel within the perceived noise at the receiver, we can ensure high



(a) Frequency – 10%   (b) Frequency – 30%   (c) Frequency – 50%

Figure 5.24: Examples of over-the-air transmission of Dirty Constellations with varying embedding frequency using the SDR prototype

degree of undetectability. We have implemented the covert communication method using a SDR prototype and present results of a wide variety of statistical tests that confirms the low probability of detection of Dirty Constellation. Higher datarate, very low probability of detection coupled with easy implementation within existing protocol stacks make Dirty Constellation a very successful method to implement covert channels in wireless communication.

Chapter 6

CODIPHY : Part 1 - Hierarchical Knowledge Representation System

The experiences obtained from designing complex and highly programmable physical layers on FPGAs and to enable broader research in the area of wireless networking, enables us to rethink the way these radios are built and operate in a cognitive network. This chapter discusses the techniques to achieve the goal of *composing* radio physical layers from fundamental components based on the current requirements of the wireless environment. Cognitive radios not only operate on various frequencies, they are are also diverse in terms of their capabilities. CODIPHY provides the mechanism for these varied radio nodes to collaborate not only on the high level spectrum policies but also enforce those policies using proper waveform, by specifying ”how to build” the physical layer to process a particular waveform.

Another related problem in the domain of radio engineering is the multi-disciplinary nature of designing radio PHY [107]. In order to make complex physical layer design more tractable by a broader group of researchers, clear abstractions are required at various levels. This also requires a component based design approach that is used to compose the radio pipeline from a set of pre-compiled components, which form the building blocks of the PHY. These components are tied together either by electrical wires in the hardware or form sequential statements to be executed on a processor architecture. The components range from basic arithmetic and logic units to composites of these like complex multipliers, relational unit, counters, etc. Some aggregates are required for specialized task like Fourier transform, forward error correction and filtering. For these complex aggregates it is best to keep them unaltered as they are optimized for a particular implementation by domain experts. SDCRs are essentially a composition of these basic elements and aggregates along with control signal to direct the flow of samples through the pipeline. The advantage of the aggregates is that they can be treated as a black box which are optimized for a particular application or even for a target hardware: e.g., a FFT unit for FPGAs will have a different set of optimization than that of the FFTW library that runs on a general purpose processor. Therefore, there is clearly a concept of domain specific optimization embedded in the radio design process. Languages used for scientific computing like MATLAB, also employ similar design approach where domain knowledge is shared across users using predefined functions with input and output parameters. The users of that function need to understand the relationship between the input and output variables and not worry about how they are processed within the function. Sometimes, these abstractions are also used to improve performance of the algorithm for heterogeneous platforms as well. Figure 6.1 shows the system level view of CODIPHY which consists of the three major components: 1) The hierarchical knowledge representation system (chapter 6) 2) the querying and inferencing engine (chapter 7) and 3) Composing the radio either using precompiled design libraries or code generation from fundamental elements (chapter 8).

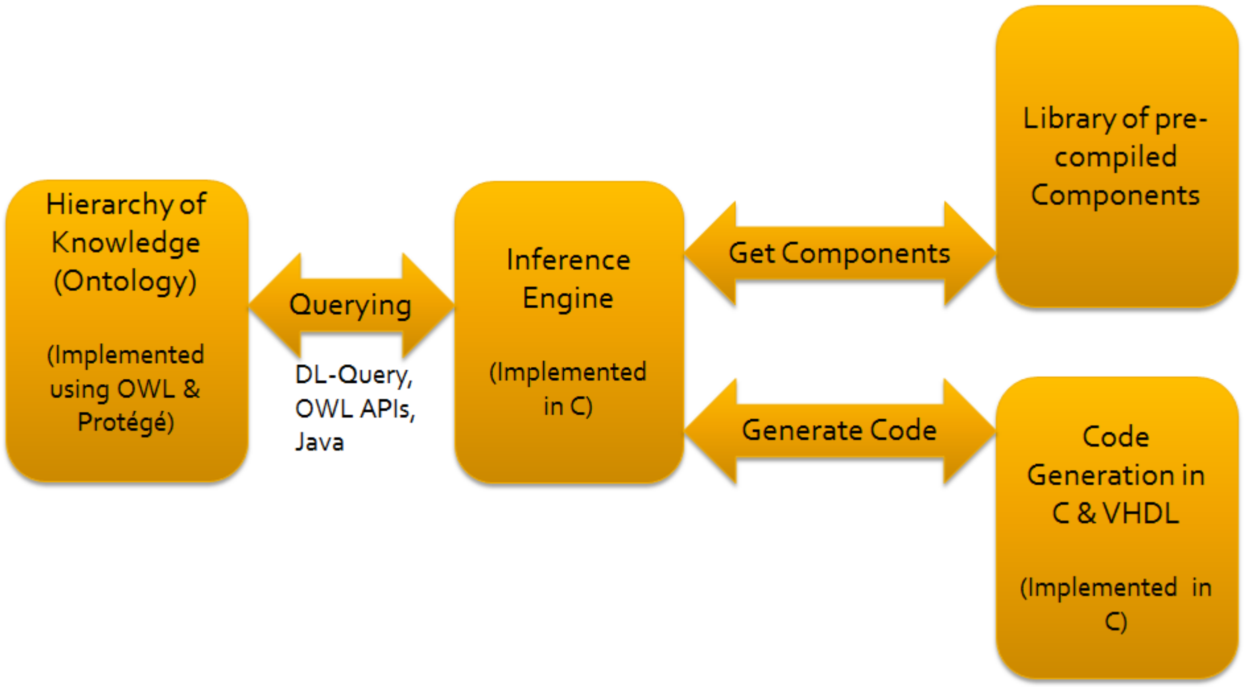

Hierarchy of Knowledge (Ontology)
(Implemented using OWL & Protégé)
Querying
DL-Query, OWL APIs, Java
Inference Engine
(Implemented in C)
Get Components
Generate Code
Library of pre-compiled Components
Code Generation in C & VHDL
(Implemented in C)

Figure 6.1: CODIPHY: A system level view

In this chapter we design a knowledge representation system for an OFDM PHY that is able to provide information about the specifications of the radio at various levels of granularity. Depending upon the specific requirement the radio designer the radio components can be chosen either as a high level aggregates with proper parameters or as a low level dataflow graph. The system allows non-experts to query the knowledge base and decide on which level of granularity to use for a particular implementation. This concept of sharing domain knowledge extends beyond human agents. Cognitive radio agents can now query the knowledge base to acquire knowledge about the components and structures of other radios.

6.1 Knowledge Representation System

The knowledge representation system serves three purposes: 1) Allow researchers and users of varied capabilities to build functioning physical layers by providing information at different level of granularity and details, 2) Serves as the high level definition for a particular radio that is independent of the implementation. It specifies what is required to compose the PHY rather than how to implement those components for a particular processing platform, and 3) Obtain knowledge about other radios by querying a shared knowledge base. Using automated code generation infrastructure, radios will be able to build completely new structures that was not built for that radio. This also facilitates active collaboration of cognitive radios at the physical layer. The levels in the knowledge representation system are as follows:

(1) System level: At this level, we specify a very broad description of the radio from a system perspective. For example, is it a CDMA or an OFDM radio? We also specify the various capabilities of the radio like bandwidth, sampling frequency, tuning range etc. These are required to initiate a collaborative adaptation of PHY using CODIPHY.

(2) Subsystem level: This is still at a high level but now delves deeper into a particular family of radios. For example, what are subsystems of the OFDM radio? Does it have a specific type of correlator? Does it use Ethernet for user I/O, etc.

(3) Specification level: Specifies the parameters of a particular subsystem. What are the inputs and output variables and what are the programmable parameters of a subsystem? These information are typically derived from the mathematical equations that define that subsystem and are synonymous with *tuning knobs* in cognitive radio parlance.

(4) Dataflow level: Dataflow level is the lowest level of expressing a subsystem. At this level the entire subsystem is a graph of interconnected components, which is provided as library of pre-compiled design files that are tied together to form the subsystem.

Each of these levels provide enough information to build a functioning radio but also provide a hierarchical approach that can be exploited by users of varied expertise to build complex radios. The knowledge base allows the user to specify the radio functions at the proper level and build it using a library of component. In this thesis we use choose to represent the physical layer of a OFDM based cognitive radio that is built as an extension to $802.11a/g$ PHY. We present some examples from the radio subsystem to build a hierarchical knowledge representation system and show how to use it to build practical radios.

6.1.1 Specification level

Specification level reveals the programmable parameters and their properties for a particular subsystem. This is useful when an user wants to modify an existing subsystems with a new set of parameters. These parameters are typically obtained from the mathematical equation or specified by the domain expert. The user treats the subsystems as *black box* because the user is limited to the knowledge of the parameters. Equation 6.1.1 shows the correlation operation that is used to detect an incoming packet in $802.11a/g$. With domain knowledge, the parameters for the correlator are identified as: 1) correlator size, 'L' and 2) correlator coefficients, 'y(n)'.

$$c(n) = \sum_{k=0}^{L-1} x(n+k).y^*(n) \tag{6.1.1}$$

Now, At this level, the user will have to provide the values for these two parameters to program the correlator making it flexible. However, the library of components should have a correlator that is designed in such a way to accept these programmable parameters. Otherwise, either it will be a fixed function correlator or more details are required about the correlator to compose it from even more fundamental building blocks. Also, library elements are provided for multiple implementation targets so that heterogeneous devices can use the same specification to build functioning PHY for a specific processing platform. This is an example of sharing domain knowledge while specifying exactly what is required.

6.1.2 Dataflow Level

The dataflow level provides the most detailed information about *how to build* the various subsystems of the PHY. It specifies how the atomic building blocks are connected to each other to compute a particular DSP algorithm. For example, the packet detection unit for OFDM utilizes the periodic nature of the short preamble. The Schmidl and Cox [81] algorithm for packet detection uses the short preamble that is present in every OFDM packets, which are periodic over 64 samples. The algorithm computes two metric on the complex input samples. Equations 6.1.2 and 6.1.3 compute the autocorrelation energy between input signal, r(n) and its delayed version r(n+L), and the signal energy during that autocorrelation window respectively, where $L = 64$ for 802.11a/g.

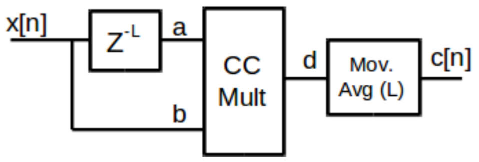
$$c(n) = \sum_{i=0}^{L-1} (r^*_{n+i} . r_{n+i+L}) \tag{6.1.2}$$

$$p(n) = \sum_{i=0}^{L-1} |r_{n+i+L}|^2 \tag{6.1.3}$$

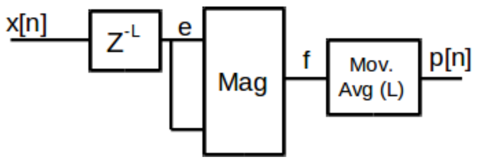
If $c(n) > Kp(n)$ is satisfied then system detects an OFDM packet. The threshold K is a design parameter that is user-defined.

Analyzing eq.6.1.2 and eq.6.1.3 reveal distinct components that compute the metric $c(n)$ and $p(n)$. Since the metric is represented in the discrete domain with n being the index for samples or discrete instance of time, it does not affect the components. In other words, the sample n is represented by clock ticks in a hardware, whereas it is represented as a iterative operation like a *loop* in a software program to apply the same transformation on all the incoming samples. Hence, the notion of time is irrelevant when systems are represented functionally.

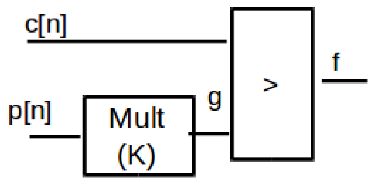
Figure 6.2(a) shows the components for $c(n)$. It suggests that the input signal $r(n)$ is multiplied with the complex conjugate of $r(n)$ that is delayed by L samples. The summation in eq.6.1.2 is indexed over the variable L and is independent of the input sample index n. hence it is a ‘sliding window average’ component that computes a moving average of L values. The flow of the results are designated by variables, a, b and d. Similarly, figure6.2(b) shows the components used to compute $p(n)$. The metric computation requires a comparator that compares the product of k and $p(n)$ with $c(n)$ to produce the binary result f.



(a) Components of c(n)



(b) Components of p(n)



(c) Components of the detection metric

Figure 6.2: Components derived from packet detection unit

It is to be noted that all the components have been identified at a high level and we do not focus on how these components are implemented. For example, the moving average unit can be implemented using a number of structures in hardware, e.g., a single stage comb filter or as a FIR filter with constant coefficients. From a functional point we assume that these implementation specific structure will be provided for composing the radio. Figure 6.3 shows an example implementation of the packet detection unit for a FPGA that has been designed using the System Generator tool [19]. Each of the individual blocks are synthesizable onto FPGA. The complex conjugate multiplier(A*Conj(B)), comb filter (cic1, cic2) and the complex magnitude (Mag0, Mag1) are considered as aggregates that are pre-designed and used as atomic library elements just like the other basic components like the delay, multipliers and comparators.

Using this example we show that any DSP system can be completely represented using a set of components and aggregates which is a direct reflection of the variables and constants provided in the mathematical function. We also generate an implementation from this high level functional description and target different hardware platforms and automating this process is the the ultimate goal of CODIPHY.

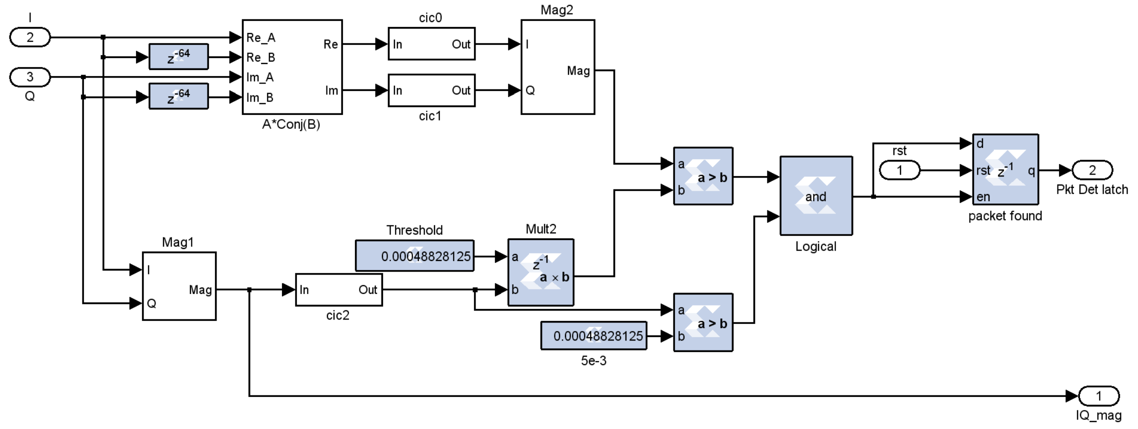


Figure 6.3: Packet Detector using CIC filters

6.2 Ontology of the Radio Physical Layer

In this section, we present the design and architecture of a knowledge representation system using an ontology. Ontology is used to share common knowledge about a particular domain using classes and relationships between them [108], such that it is machine interpretable using logical inferencing. Ontology separates domain knowledge of radio design from operational knowledge of the underlying subsystems but also allow experts to contribute, analyze and improve using domain knowledge. In this thesis, the purpose of the ontology is to capture the hierarchy of the knowledge base. Each level in the knowledge base contain its own set of classifications and relationships. The main components of an ontology are: 1) Classes: Unique concepts in a domain, e.g., the various levels in the knowledge base are distinct classes and so are the various components used to build a radio. 2) Individuals: Instances of these classes are called individuals which form the content and structure of the ontology, and 3) Properties: These define how the instances are related to each other and also their specifications.

The ontology represents a common vocabulary that is used to share domain knowledge among a wide variety of radio designers as well as users. Ontological definition largely depend on the application and is very much guided by the purpose that is to be served by it. Therefore, an ontology is not unique, but has many viable alternatives. In this thesis, we build an ontology to share knowledge about the internal structure and operation of an OFDM physical layer in order to facilitate a component based design by agents of varied expertise and capabilities. We have used the OWL language [109] and the Protégé [110] open source tool to build the ontology.

6.2.1 Taxonomy of the Ontology

In order to use ontology to share knowledge across different domain of radio engineering the terms, structure and semantics should be uniform across the domains. It is important to explicitly assert *"what is the ontology used for?"*, so that the knowledge is interpreted by agents in a meaningful manner and that is consistent. We define a taxonomy for the ontology that focuses on encapsulating the design elements of the physical layer. Although, the ontology can be used to express multiple types of radio, in this thesis we focus on the OFDM physical layer that has been adopted widely for high throughput wireless communications.

OFDM operates with many tunable parameters, but CODIPHY goes beyond "tuning knobs" to include design *blue prints* for new subsystems as well, which can be composed on-demand and implemented on a target radio architecture. We start by classifying the concepts for the radio architecture defined in chapters 3 and 4. Then we define the scope of the ontology in designing generic OFDM physical layers and show specific examples of hierarchical inferencing to compose radio components at different levels of representations. The steps involved in building an ontological classification to represent the hierarchical knowledge base for designing a generic PHY for OFDM waveform are as follows:

(1) Specify the "four" levels of representation. These form the top-level classes in the taxonomy.

(2) Define the sub-classes for each level. Define what information is contained in each level and how it will assist in composing OFDM PHY.

(3) Define instances of these classes, called *Individuals* which are either mapped to pre-compiled design components or specify the various capabilities of the PHY.

(4) Properties that define the relationship between these instances, called *Object Properties* and also specify their attributes, called *Data Properties*

Figure 6.4 shows the taxonomy and classification of the OFDM PHY ontology. The *is-a* relationship

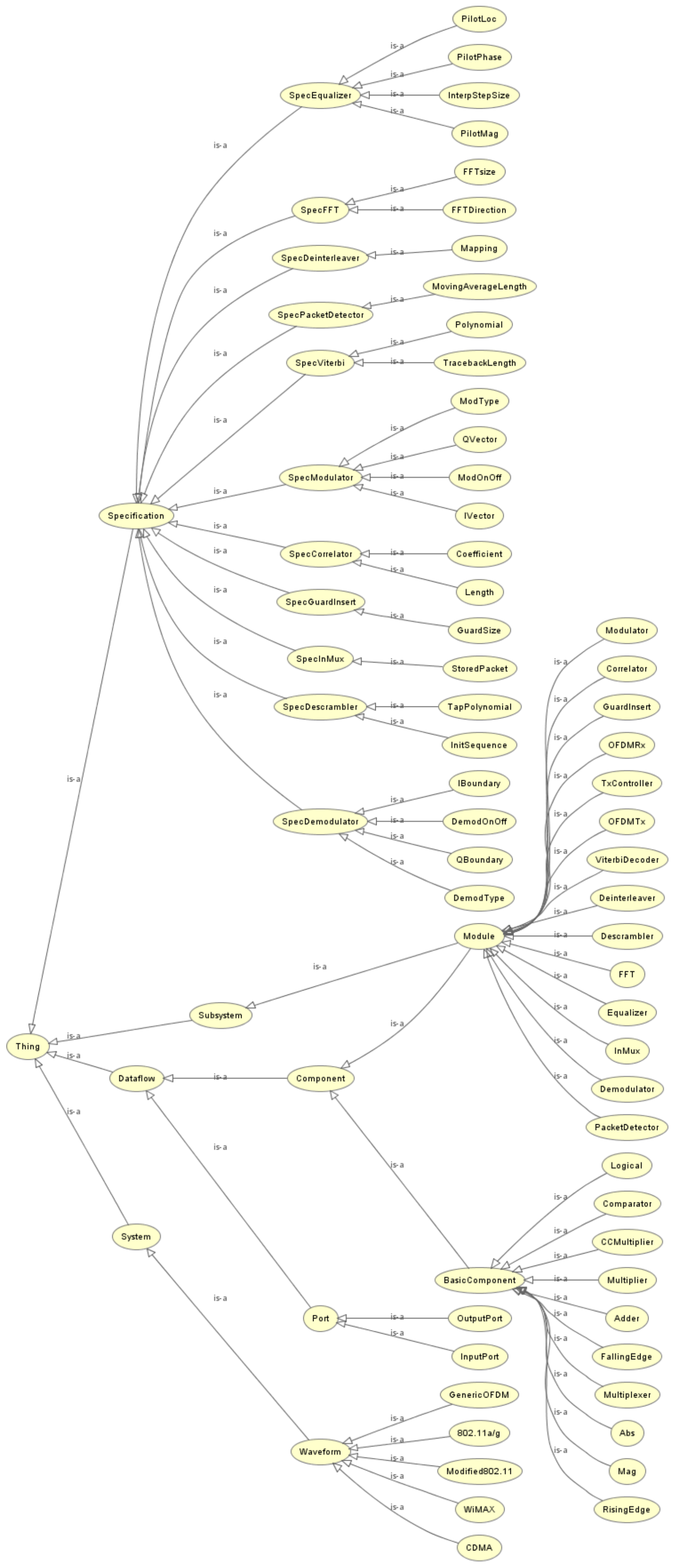


Figure 6.4: Classes in the OFDM PHY ontology

specifies the subclass-superclass hierarchy. The levels of the knowledge base is specified by the four top-level classes: System, Subsystem, Specification and Dataflow. Each of these levels have their own subclasses depending on the type of information obtained from that level.

System Level: The "System" class contain different classes of waveform that is supported by a particular radio platform. This ontology supports 802.11a/g and any modifications to the basic PHY and other OFDM waveforms like WiMax and LTE. Each of these classes have a corresponding instance that specify the library component used to implement that waveform. Instances of a particular waveform also have properties associated with them like bandwidth and tuning range. New properties pertaining to a particular waveform can be added to the ontology if that information is relevant to radio design. However, the information should be limited to system level specification of the PHY only. This is useful for fast prototyping by using pre-compiled implementations from the library of components and synthesize for a particular target.

Subsystem level: The "Subsystem" class contain the various classes of processing units required for decoding a particular waveform. In this ontology, we include all the modules required for the 802.11a/g PHY. These modules are based on our prior implementation and conforms to a specific architecture of the radio. However the ontology can be used to include other architectures as well and the modules corresponding to that will be a part of this class. Each class of component has multiple instances which have different properties and architecture for optimized performance. These instances are related to the instances of classes of different waveforms by the *hasSubsystem* object property.

Specification Level: Every instance of the Module class (subclass of Subsystem Level) are unique implementation of that subsystem. Each implementation will have a set of specifications that define the range of operation. These specifications are analogous the the concept of tuning knobs in cognitive radio that make the PHY programmable with a variety of parameters. The purpose of the Specification class is to organize these specifications for each subsystem. The specification for each subsystem has the convention *Spec¡SubsystemName¿* which has multiple subclasses of unique tuning knobs. The instance of these tuning knobs are properties that are exposed as programmable features during implementation. The instances are related to the instances of the Module class at the Susbsytem level by the *hasSpecification* object property.

Figure 6.5 shows how the System, Subsystem and Specification classes are related. In this example we show the how an instance of the FFT unit is related at different levels of the knowledge base. At the system level let's assume that the knowledge base has an instance of the Waveform class called "modified802.11a/g", which is essentially a 802.11a/g physical layer but with programmable subsystems. The "modified802.11a/g" instance *hasSubsystem* "RangeFFT" (supports a range of FFT sizes) which is an instance (*hasIndividual*) of the FFT class and is classified under the Module class(*hasSubclass*) at the subsystem level. Also at the specification level, SpecFFT class contains the classes for the different specifications for an FFT. In this example, we show that the SpecFFT class has two subclasses, FFTSize and FFTDirection. Each of these have specific instances or unique values, e.g., FFTsize class has instance $[64pt, 256pt, 512pt or 1024pt]$. Now, for particular radio PHY implementation, the range of operation of the RangeFFT instance is given by the object property *hasSpecification* relating it to the instances of the FFTSize class. Therefore, using this example we show how a specific instance of a component is classified in the hierarchy of the knowledge representation system.

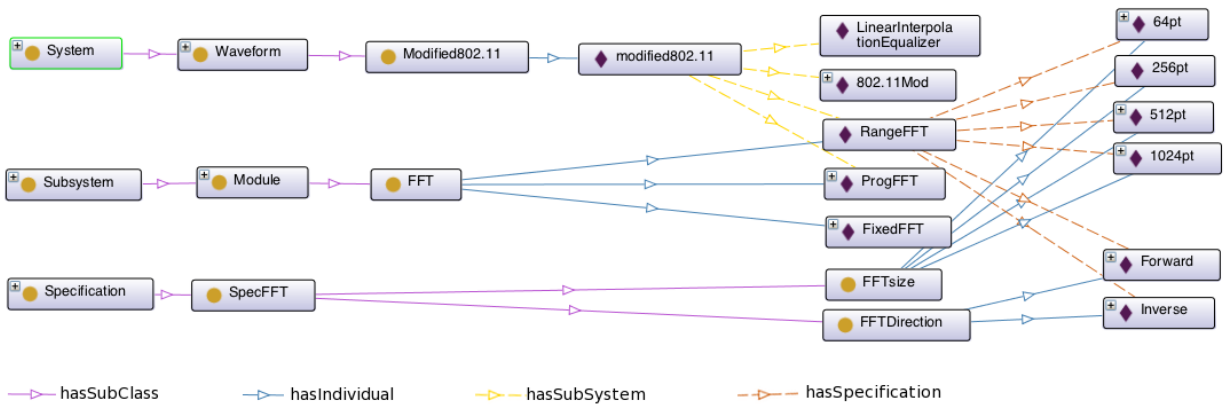


Figure 6.5: Relationship between different level of the Knowledge representation system using object properties

Dataflow Level: Dataflow level is lowest level in the hierarchy and the most complex. The purpose this level of representation is to express cognitive radios beyond the limitations of tuning knobs and allow them to be truly self aware by composing new signal processing structures as required. The top level classes at the Dataflow level are *Component* and *Port*. Component have subclasses 1) BasicComponent: Contain classes of all fundamental processing units defined by the domain experts to implement various DSP functions, and

2) Module: Contain classes of subsystems, which are a built using instances of BasicComponent. Port has two subclasses, InputPort and OutputPort. Each basic component has a set of input and output ports which are instances of these two classes.

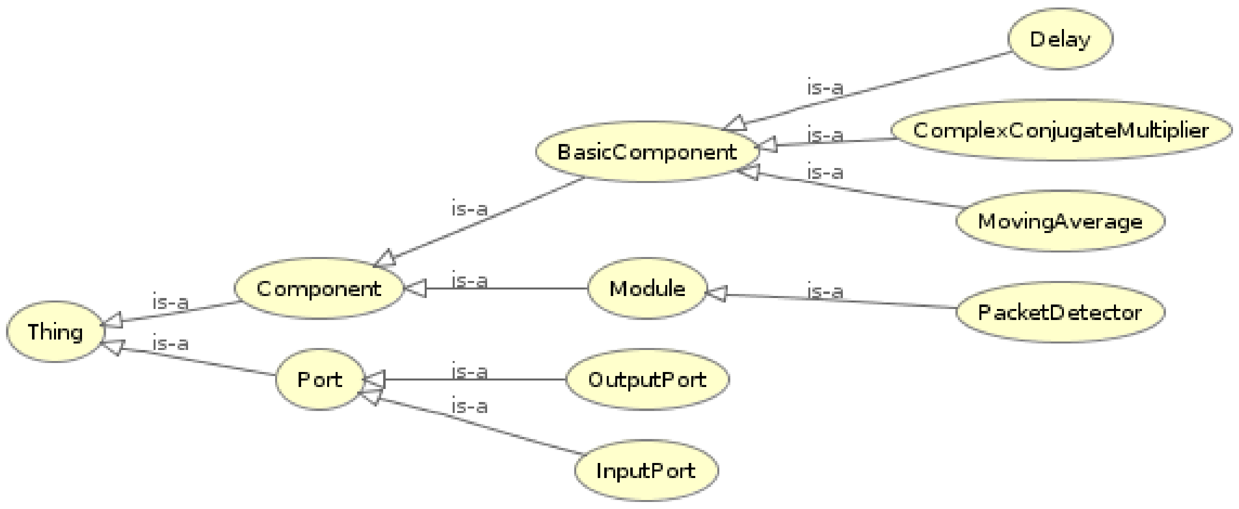


Figure 6.6: Classes in the Packet Detector ontology

Equations 6.1.2 and 6.1.3 represent the mathematical structure of the OFDM packet detector, and figure 6.2 represents the components and the interconnections of the same unit. Figure 6.6 shows the hierarchy of the different classes created for representing the partial Packet Detector specified by eq.6.1.2. The ‘is-a’ relationship denotes the hierarchy of the classes. In this example the PacketDetector is a Module that contains instances of the classes Delay, ComplexConjugateMultiplier and MovingAverage, which are in turn subclasses of BasicComponent. The class hierarchy of CODIPHY is constant for all ontologies that define a radio PHY. However, the specific naming of the instances is left to the designer of the ontology. Thus the instances are in the design space and depend on the person building the ontology while the class taxonomy is the specification that never changes. This is also useful in a component based radio implementation, where the instance of the class BasicComponent is mapped to a pre-defined design file or function that can be stitched together to implement radio. Therefore, the ontology specifies *what* is required to build a radio rather than specifying *how* to build it.

The instances of the classes are related to each other using certain properties, e.g., an instance of the class Delay, *delay1*, has an input port *delay1In* and an output port *delay1Out*, which are instances of the class InputPort and OutputPort respectively. These are in turn subclasses of the class Port. We define the properties that are required to specify the dataflow. These properties are part of the taxonomy and remain constant for all implementations of the ontology. The property *hasInputPort* defines the input ports of an instance of a Component or Module. Also, in order to specify the interconnections of the various instances, we need to specify which port is connected to whom. In particular, there has to be a way to specify how the output port of an instance is connected to the input port of an instance of another class or the same class. Moreover, high level inputs, like that of a module will be also connected to input ports of certain instances, like the $x(n)$ input is connected to one of the inputs of the Delay in figure 6.2(a). The selection of properties and how they relate the instances will depend on how a reasoning engine would classify the ontology to answer queries from the user or other radio agents. In CODIPHY, we deal with those queries only that will allow an external agent to understand what components are present in a module, e.g., Packet Detector and how are they connected to each other, so that if required the querying agent can recreate the packet detector without human intervention. Figure 6.7 shows a partial ontology of the OFDM packet detector and the relationship between various instances of the classes defined in the ontology.

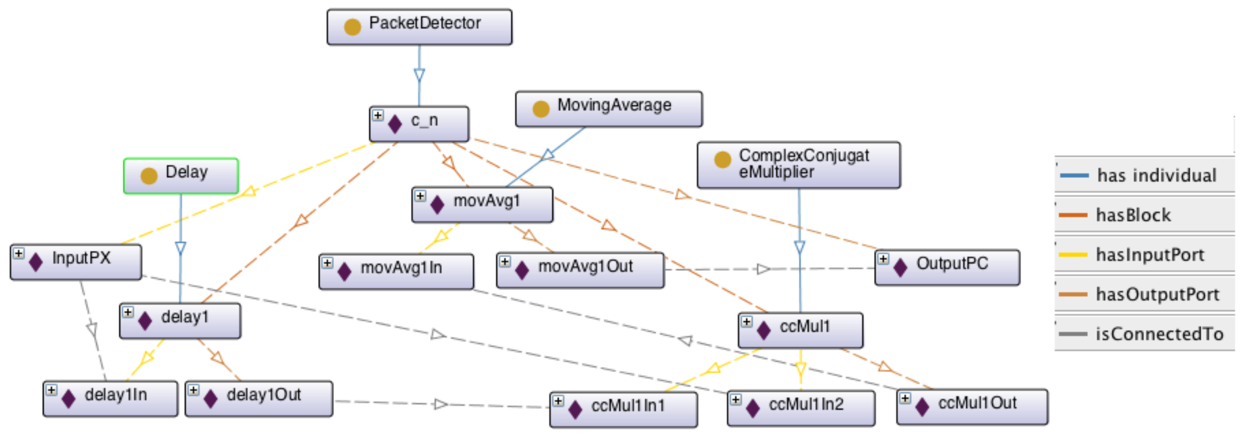


Figure 6.7: Dataflow of the Packet Detector subsystem

The main properties that are used to relate the components are,

- *hasBlock* and *isABlockOf*: These two properties relate an instance of a module with instances of basic components that are contained within that module.

- *hasIndividual*: This property relates instances to their respective classes.
- *hasInputPort* and *isInputPortOf*: Relates the instances of components with instances of their input ports.
- *hasOutputPort* and *isOutputPortOf*: Relates the instances of components with instances of their output ports.
- *isConnectedTo*: Relates various instances of the Port class. e.g., relates the output port to that of an input of another component.

These properties or relationships are defined based on what is the objective of the ontology or otherwise what are the queries that an external agent might ask the knowledge base to learn about the internal structure of a module. This is also a very domain specific requirement. In order to express other facets, e.g., behavioral aspects of a CR, new properties are required which is not included in the scope of this thesis.

6.3 Dataflow representation of OFDM transceiver

Building the ontology is fundamental to making CODIPHY practical. Using domain expertise in hardware design and DSP algorithms, we have identified a set of components and aggregates that are considered *atomic* from a system perspective and are available to the radio design agent as pre-compiled libraries. The dataflow of a subsystem is obtained by analyzing the mathematical equation that specifies a computational flow, transforming the input samples to meaningful output. Complex transceiver design often have model hierarchy to make the design modular. Since ontology is a textual representation, model hierarchy is captured using the *hasBlock* object properties that relates instances of the Module class. The building blocks of that subsystem is related by the property *hasComponent*. Since all subsystems in the hierarchy are members of the Module class, their names should be unique in the ontology. The flat architecture of the ontology entails all instances to have unique names to eliminate ambiguity during the querying process. The designer of the ontology has the liberty to assigned unique names to the individuals but the classes specified in the taxonomy should be consistent across all ontology. This makes the ontology inter-operable across multiple domains and agents.

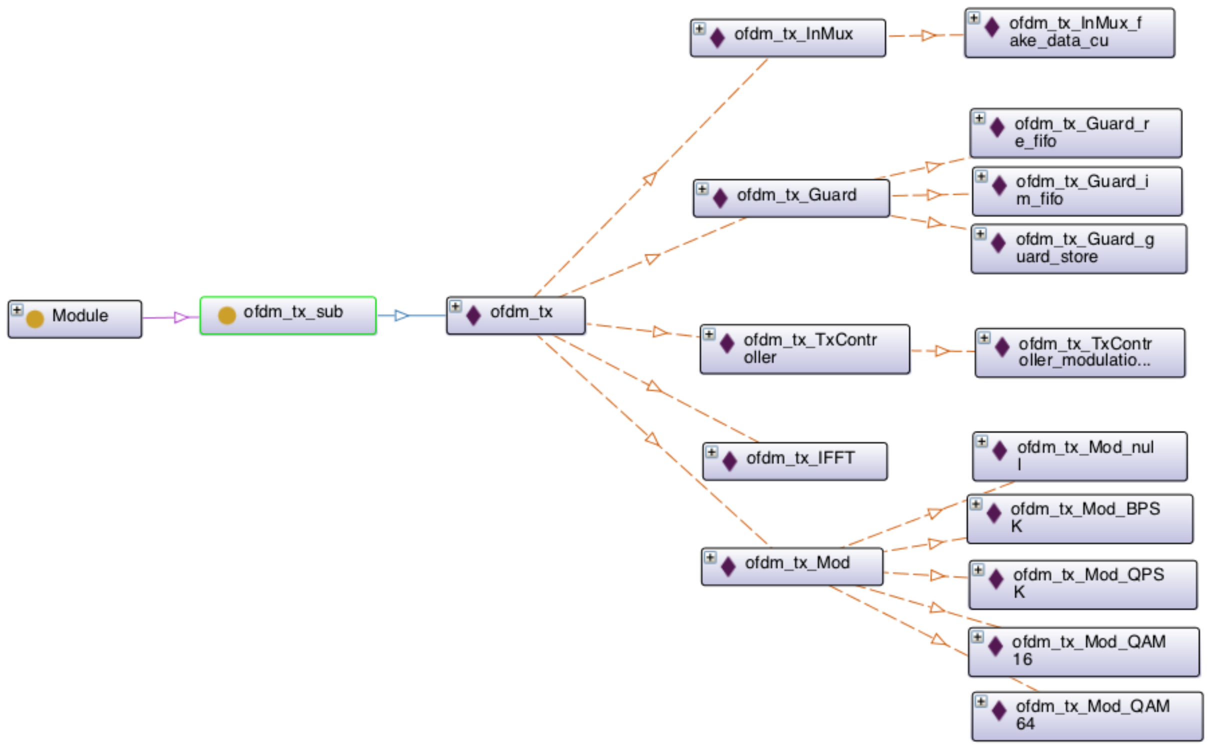


Figure 6.8: Dataflow Hierarchy of OFDM Transmitter

Figures 6.8 and 6.9 show the model hierarchy of the OFDM transmitter and receiver respectively for the radio architecture described in chapters 3 and 4. All instances of the various subsystems belong to the class Module and are related by the *hasBlock* property to capture the model hierarchy. This is an important and a required feature for the querying and code generation steps of CODIPHY discussed in chapters 7 and 8 Table 6.1 and Table 6.2 show the complexity of the ontology for various transmitter and receiver systems respectively.

6.4 Ontology for MAC-PHY Wireless crosslayer research

One of the research domains where radio is an integral part is the wireless MAC-PHY crosslayer research community. A lot of focus in this domain is in utilizing the PHY to innovate higher layer protocols

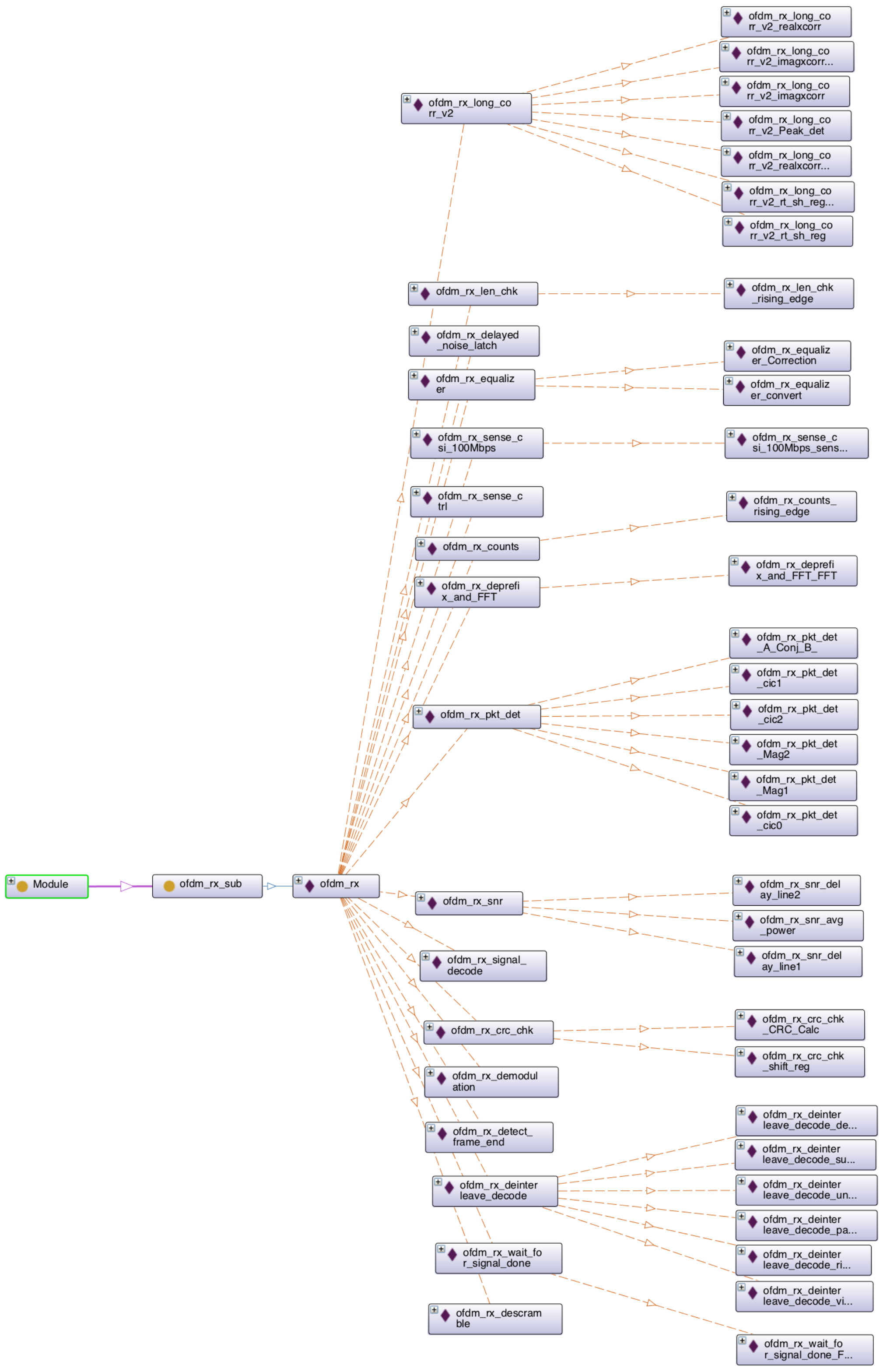

Module
ofdm_rx_sub
ofdm_rx
ofdm_rx_long_corr_v2
ofdm_rx_long_corr_v2_realxcorr
ofdm_rx_long_corr_v2_imagxcorr...
ofdm_rx_long_corr_v2_imagxcorr
ofdm_rx_long_corr_v2_Peak_det
ofdm_rx_long_corr_v2_realxcorr...
ofdm_rx_long_corr_v2_rt_sh_reg...
ofdm_rx_long_corr_v2_rt_sh_reg
ofdm_rx_len_chk
ofdm_rx_len_chk_rising_edge
ofdm_rx_delayed_noise_latch
ofdm_rx_equalizer
ofdm_rx_equalizer_Correction
ofdm_rx_equalizer_convert
ofdm_rx_sense_csi_100Mbps
ofdm_rx_sense_csi_100Mbps_sens...
ofdm_rx_sense_ctrl
ofdm_rx_counts
ofdm_rx_counts_rising_edge
ofdm_rx_deprefix_and_FFT
ofdm_rx_deprefix_and_FFT_FFT
ofdm_rx_pkt_det
ofdm_rx_pkt_det_A_Conj_B_
ofdm_rx_pkt_det_cic1
ofdm_rx_pkt_det_cic2
ofdm_rx_pkt_det_Mag2
ofdm_rx_pkt_det_Mag1
ofdm_rx_pkt_det_cic0
ofdm_rx_snr
ofdm_rx_snr_delay_line2
ofdm_rx_snr_avg_power
ofdm_rx_snr_delay_line1
ofdm_rx_signal_decode
ofdm_rx_crc_chk
ofdm_rx_crc_chk_CRC_Calc
ofdm_rx_crc_chk_shift_reg
ofdm_rx_demodulation
ofdm_rx_detect_frame_end
ofdm_rx_deinterleave_decode
ofdm_rx_deinterleave_decode_de...
ofdm_rx_deinterleave_decode_su...
ofdm_rx_deinterleave_decode_un...
ofdm_rx_deinterleave_decode_pa...
ofdm_rx_deinterleave_decode_ri...
ofdm_rx_deinterleave_decode_vi...
ofdm_rx_wait_for_signal_done
ofdm_rx_wait_for_signal_done_F...
ofdm_rx_descramble

Figure 6.9: Dataflow Hierarchy of OFDM Receiver

Subsystem	Modules	Compo- -nents	Input Ports	Output Ports	Conne- -ctions
TxController[1]	1	29	53	53	55
IFFT[2]	0	7	16	18	12
Modulator	6	63	119	71	102
InsertGuard	3	16	66	32	60
Unique number of Basic Compoents - 23					
Lines of RDF code - 6531					

[1]FSM to parse control and data frames[74]

[2]IFFT treated as atomic unit

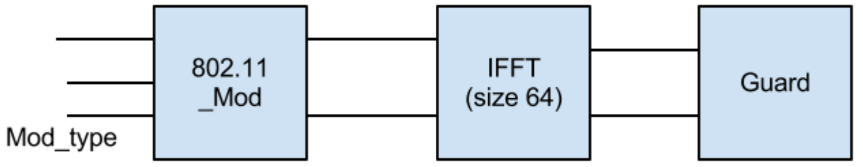
Table 6.1: Ontology of the transmitter

Subsystem	Modules	Compo- -nents	Input Ports	Output Ports	Conne- -ctions
Packet Detect	6	30	65	39	68
Correlator	1024	5152	8252	6178	8240
FFT[1]	1	20	49	34	44
Equalizer	14	137	289	180	299
Demodulator	0	32	67	34	55
Decoder	7	82	179	116	177
CRC Check	6	45	93	57	91
Unique number of Basic Compoents - 30					
Lines of RDF code - 166924					

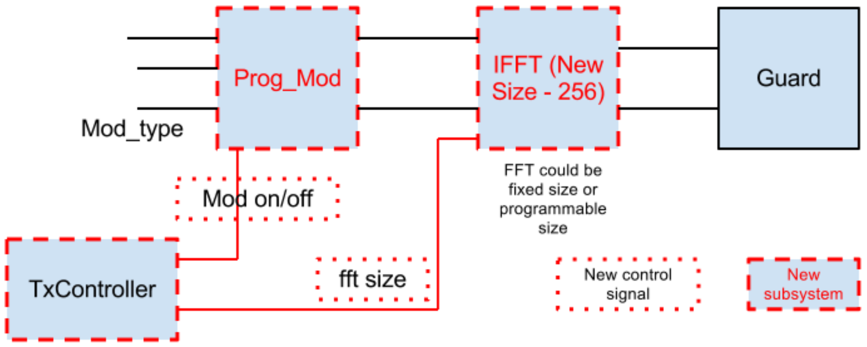
[1]FFT treated as atomic unit

Table 6.2: Ontology of the receiver

that lead to improvement in network performance. Often these techniques require modifying the PHY to implement a new techniques. Sometimes, exposing new tuning knobs are also proposed. Now, these new structures require redesigning of the subsystems which typically requires domain expertise. Therefore, ontology is an effective way to share this design knowledge such that wireless researchers can use these novel structures to implement new protocols. The taxonomy defined in this chapter is used to capture these modifications. We illustrate this by implementing a representative application form the crosslayer research literature [7]. This technique selectively utilizes OFDM subcarriers to send simple messages in a network thus making certain group communications faster. The system is a modified version of the 802.11a/g transceiver along with new structures for some subsystems. In this example we focus on the transmitter modifications and show how these *new* design information is included in the hierarchical knowledge representation system. Figure 6.10(a) shows the transmitter blocks required for modulation and conversion from frequency to time domain (IFFT). Figure 6.10(b) shows the modifications required for the same subsystems to implement the SMACK protocol. The modifications that are included in the ontology using the taxonomy described in §6.2.1 are as follows:



(a) Partial Transmitter for 802.11a/g



(b) Modified Transmitter for SMACK [7]

Figure 6.10: Example MAC-PHY Crosslayer Protocol in wireless networks

System Level: The new PHY thus created is a member of the "Modified802.11" class because the top level subsystems are same as that of 802.11 with different specifications and internal structures. Therefore, it is required to express the underlying subsystems in further details.

Subsystem Level: As shown in figure 6.10(b), the modulator and IFFT are different from that in 802.11a/g. The modulator becomes a member of the Modulator class and may have a name *ProgMod*. The specific instance name is not important for proper understanding of the ontological description. Similarly, the IFFT is also programmable and is a member of the FFT class, say *ProgFFT*. At this level we are classifying the subsystems according to the taxonomy. The specific operating parameters are provided at the next level.

Specification Level: The new modulator, although a member of the class Modulator has very different operating parameters compared to a 802.11a/g modulator. As per the requirement of the protocol, the

ProgMod instance has specification classes: ModOnOff with member *True* and ModType with members *bpsk, qpsk, qam16 an qam64*. These are the tunable parameters for that particular instance of Modulator and are subclasses of the SpecModulator class. Similarly, the instance *ProgFFT* has the specification classes: FFTSize with members *64pt, 128pt and 256pt*.

Dataflow Level: The dataflow of the *ProgMod* is constructed using object properties of the dataflow level described above. The modulator is now conditioned upon the value of the ModOnOff specification which is asserted by a “multiplexer” basic component with two input ports and one select port. The IFFT is treated as a basic component in the taxonomy. The TxController is also modified by adding relevant states and output variables for the new control signals, ModOnOff and IFFT size. Figure 6.11 shows the partial ontology for the SMACK protocol at the Subsystem and the Specification levels.

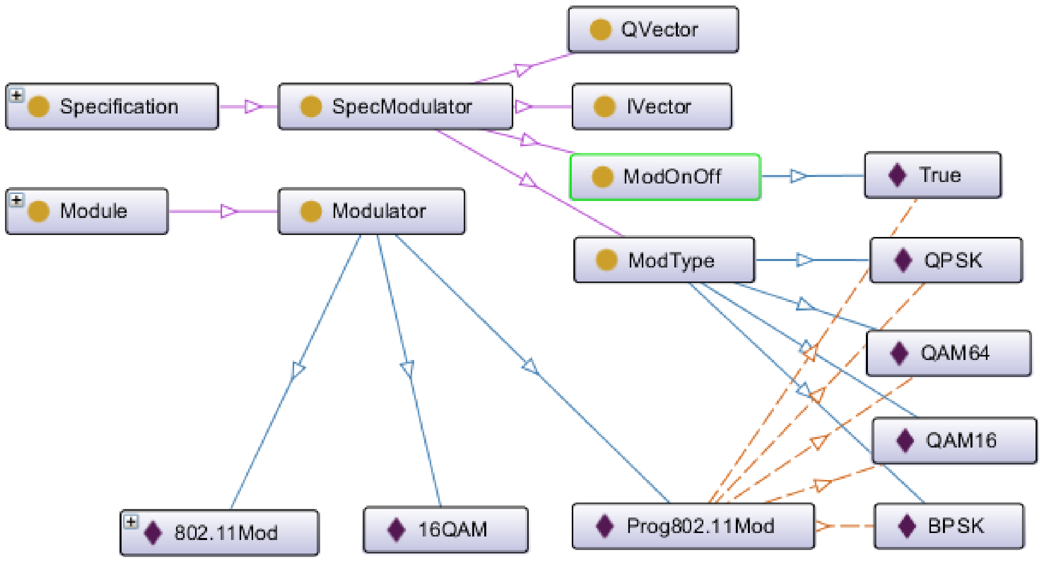


Figure 6.11: Partial Ontology of the SMACK protocol. The Modulator is included in the knowledge base at the Subsystem and the Specification levels

Chapter 7

CODIPHY : Part 2 – Hierarchical Inferencing through Queries

The purpose of using an ontology to express internal structures of a PHY is sharing of common knowledge across various domains. However, in order to understand the knowledge by an agent, human or machine, a mechanism to query the knowledge base is required. In this chapter, we present the method of using queries to achieve two goals: 1) Use hierarchical inference to learn, specify and compose radio subsystems to implement a new PHY structure. 2) Collaboration between two heterogeneous radio nodes at the PHY to agree on common processing subsystems as required by the communication protocol. Both the applications employ the same querying mechanism to achieve two different goals.

CODIPHY shows that on-demand composition of the PHY is possible at different levels depending on the capabilities and the existing knowledge of a radio. This type of collaboration is particularly useful when radio nodes are heterogeneous platforms and also use different crosslayer techniques to improve link performance. CODIPHY can be used in such cases to clone the physical layer of a radio in order to establish a link between the nodes that otherwise would not have been able to communicate. Similar, hierarchical inferencing technique is also useful while building a radio by a designer. Depending on the knowledge level, the designer chooses design elements from any of the four levels of the knowledge base. Every level delves deeper into the details of the radio components and an agent, be it human or cognitive radio, decide at which level of detail is useful for its understanding and implementation on its target platform.

In chapter 6, we described the taxonomy of the ontology that is used to build the knowledge base. In this chapter, we present the method to obtain information from the ontology using structured queries. We use DL-Query and OWL APIs [111] to query the ontology. We describe the queries at different levels and then present an example of collaboration using hierarchical inferencing in §7.5.

7.1 Querying at the System Level

Querying at the System level involves learning about the instance of various members of the Wavefrom class. This tells the querying agent what are the different types of radio PHY supported by the ontology. The purpose of this form of representation is abstraction of design details from non-experts. The instances obtained in the query are mapped to individual implementations used by the querying agent. The availability of these implementations is ensured by domain experts and the composing agent has no knowledge of the internal workings of that system. For example, if a radio node operating in 802.11a/g mode wants to switch to LTE, the PHY implementation for LTE is obtained from the System level for the particular platform of the radio. If the agent wants more information about the internal structure of the radio, querying the Subsystem level is required.

7.2 Querying at the Subsystem Level

At this level, instances of the System level and Subsystem level are related by a pair of Object properties: *hasSubsystem* and its reciprocal *isSubsystemOf*. For example, the ontology described in §6.2, Executing the query: *isSubsystemOf* **value** *802.11a/g*, provide two key information – 1) The classes of different Modules in the 802.11a/g PHY and 2) The instances of each of the Modules. Therefore, at this level the querying agent obtain knowledge about the processing subsystems of a particular radio architecture. Each instance is mapped to corresponding implementation. Conversely, executing the query: *hasSubsystem* **value** *mod* provide two results – 1) The instance name of the system that has *mod* as a subsystem and 2) the class which it belong to at the System level. Therefore, querying can be done top-down or bottom-up depending on the requirement of the composing agent. This knowledge is directly useful to the agent when a subclass of Waveform represents one of the well-known radio PHY like 802.11a/g, LTE or WiMax. This is also useful, if the agent is looking to replace a particular subsystem in its existing processing pipeline. This is a feature of cognitive radio PHY where various subsystems are stitched together to implement a processing pipeline based on the demand posed by the radio environment. This makes radios self-aware at the PHY.

For a cognitive radio and crosslayer implementations, the subsystems are more programmable with various parameters which also depends on how these subsystems are architected. In order to ensure that a subsystem is what the agent is looking for, further querying is required at the specification level.

7.3 Querying at the Specification Level

Once the particular instance is obtained at the Subsystem level, further querying is done to obtain the specifications or the programmable parameters for that instance. For example, a *progMod* instance has Mod-OnOFf as a specification. The instances of the Specification class described in §6.2, are related to instance of the classes at Subsystem level by a pair of object properties: *hasSpecification* and its reciprocal *isSpecificationOf*. This information is obtained by executing the query: *isSpecificationOf* **value** *progMod*, which provides the knowledge that this instance of the Modulator has two programmable parameters: A control to turn off modulation or *modOnOff* and types of modulation supported or *modType*. These specifications complete the knowledge of a Module at the Subsystem level. Assuming that the control plane to program the radio is adaptive to the modification of the parameters, the agent gets the new programmable modulator from the component library and implements in its radio platform. We have defined the specifications for different subsystems in the OFDM PHY in the taxonomy of the ontology in §6.2.

7.4 Querying the Dataflow

The knowledge of the Dataflow is required to compose subsystems from basic components. This is required when the library of component does not contain the implementation of a subsystem for a particular platform. Composing the radio PHY also require automated code generation for multiple targets, which is discussed in chapter 8. We use specific set of queries to learn the internal components and interconnections of a subsystem. The usefulness of representing the physical layer using ontology is in its ability to allow learning about the PHY by a machine, e.g., a CR willing to communicate with the other but is unsure about the capabilities of the other radio. Ontology provides the common language that all CR nodes can use to understand and learn about its own structure and when required, it can let other nodes know about their capabilities. Understanding the radio PHY independent of the platform, operational knowledge or

Algorithm 1: Algorithm to Automate the Query Process at Dataflow Level

```
Input: subSys
/* Module Level Query                                    */
Query: Module;
Result: mods[];
for m in mods[] do
    if subSys == m then
        /* Input Ports of the Subsystem                  */
        Query: isInputPortOf value subSys;
        Result: all input ports of subSys;
        Update: PortIns[subSys].name; Update: PortIns[subSys].dataType;
        /* Output Ports of the Subsystem                 */
        Query: isOutputPortOf value subSys;
        Result: all output ports of subSys;
        Update: PortOuts[subSys].name; Update: PortOuts[subSys].dataType;
        /* Components of the Subsystem                   */
        Query: isABlockOf value subSys;
        Result: all components of subSys;
        Update: Comps.funcName; Update: Comps.vars;
        for c in Comps[] do
            /* Input Ports of each Component             */
            Query: isInputPortOf value c;
            Result: all input ports of component c;
            Update: PortIns[c].name; Update: PortIns[c].dataType;
            /* Output Ports of each Component            */
            Query: isOutputPortOf value c;
            Result: all output ports of component c;
            Update: PortOuts[c].name; Update: PortOuts[c].dataType;
        end
        for inP in PortIns[] do
            /* Connections from all Input Ports          */
            Query: isConnectedTo value inP.name;
            Result: all connections from inP;
            Update: Connects.From = inP; Update: Connects.To;
        end
        for outP in PortOuts[] do
            /* Connections from all Output Ports         */
            Query: isConnectedTo value outP.name;
            Result: all connections from outP;
            Update: Connects.From = outP; Update: Connects.To;
        end
    end
end
```

specific implementation is very beneficial in a heterogeneous environment. With this knowledge distribution mechanism, radios adapt to form homogeneous communication environment.

Algorithm 1 shows the automated querying process at the Dataflow level. First, the subsystem is searched and if it matches the intended subsystem required for collaboration, then the queries are processed for that. At this point, we are interested in knowing the instances of various classes of components in the dataflow ontology. Once the instance of the subsystem (or Module in Taxonomy) is found, we query the input ports, output ports and the components of the subsystem, and update variables **PortIns**, **PortOuts** and **Comps**. This gives us the knowledge of the input and output ports of the subsystem, as well as the components or aggregates contained within the subsystem. Then, for each of the components in the subsystem, we query the input ports and the output ports and update the variables **PortIns** and **PortOuts**. Once the list of all the ports have been generated, we query the connections of each input port and each output port. This gives us the knowledge of how the data flows from one port to another. Through this query process, we also get the information of the data type of each port and the variables of each component, which are required in the code generation phase. The number of queries required to get all the information in dataflow level depend on the size of the subsystem as follows:

$Total\ number\ of\ queries = (1 + Blocks + Components + InputPorts + OutputPorts)$

Hence, the number of queries required for each module can be computed from Tables 6.1 and 6.2. The output of the queries is the intermediate form, from which target-specific code is generated. It is stored in the file system and used by the code generation process.

Continuing with the example of the OFDM packet detector subsystem shown in Figure 6.6 and 6.7, we design a set of queries that will reveal the components present in it and their interconnections. Table 7.4 shows the queries and the corresponding knowledge obtained about the dataflow between various components. Once the knowledge about all the components and their interconnections are obtained, CODIPHY proceeds to the code generation phase that generate an implementation of the design learned through querying.

Query	Class	Instance	Knowledge Acquired
Module	PacketDetector	c_n	Subsystem
isInputPortOf value c_n	InputPort	InputPX	Input Port of the Module.
isOutputPortOf value c_n	OutputPort	OutputPC	Output Port of the Module.
isABlockOf value c_n	Delay ComplexConjugateMultiplier MovingAverage	delay1 ccMul1 movAvg1	Components of the Module.
isInputPortOf value delay1 Repeat this query for each component in Module.	InputPort	delay1In	Input Port of each Component.
isOutputPortOf value delay1 Repeat this query for each component in Module.	OutputPort	delay1Out	Output Port of each Component.
isConnectedTo value ccMul1In1 Repeat this query for all ports in Module.	OutputPort	delay1Out	Connections between ports.

Table 7.1: Querying the dataflow of the Packet Detector subsystem

7.5 Collaboration using Hierarchical Inferencing for Crosslayer Implementations

We illustrate the hierarchical inferencing technique with an example of PHY adaption using the hierarchical knowledge representation framework. We revisit the SMACK crosslayer technique [7] described in 6.4. Let us assume that Radio A and Radio B are two nodes that want to set up a communication link by agreeing on the common PHY as shown in 6.10. Let us also assume that Radio B was operating in 802.11a/g mode and in order to collaborate with Radio B, is using hierarchical inferencing by querying the knowledge base of Radio A.

Table 7.2 shows the queries at different levels. The process starts at the System Level and Radio B infers the class of Waveform used by Radio A which is the instance *modified80211*. We recall that the taxonomy restricts the names of the classes but not the instances. Therefore, Radio B needs to acquire the further knowledge about the instance *modified80211*. At the Subsystem level, Radio B queries for the subsystems in the Radio A's PHY. The subsystems that are different between the two radios are the Modulator(*progMod*) and the inverse FFT which is inferred by Radio B from the query results at the Subsystem level. At this point Radio B, has inferred that it needs a *different* modulator. Radio B queries for Specification of the new modulator instance *progMod* and infers that the modulator has two Specification classes: ModOnOFf and

ModType with instances *true* and *bpsk, qpsk, qam16, qam64* respectively. At this point, Radio B has the complete specification for the modulator required to implement the SMACK technique on its platform. If the component library of Radio B has an implementation of a modulator with the required specifications, it can use that to implement the new PHY structure. If not, then Radio B delves deeper into the Dataflow level to infer what are the components and the interconnections between those so that it can compose the required implementation for its target platform. This Dataflow is obtained by querying the knowledge base of Radio A.

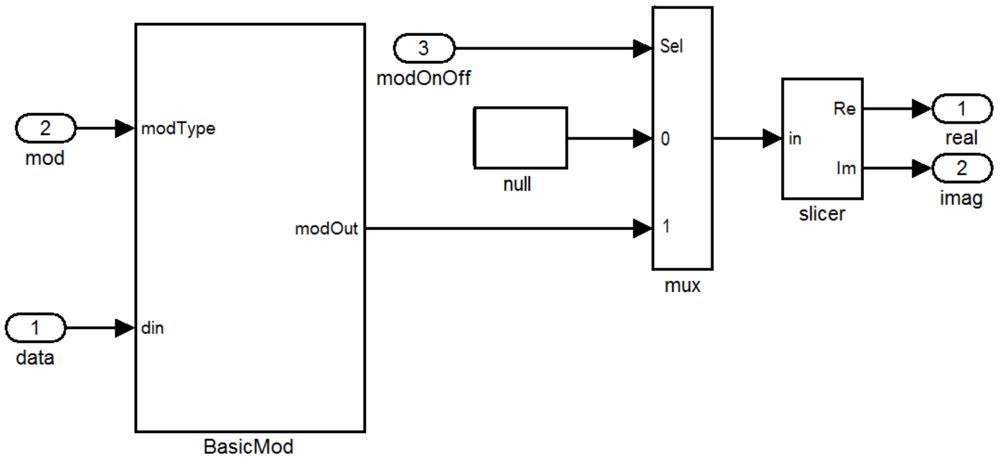


Figure 7.1: Block Diagram of the modulator for SMACK

The Dataflow is obviously the most complex and detailed form of representation of a subsystem. Figure 7.1 shows the internal structure of the *progMod* instance of the Modulator. The *BasicMod* module is the same module that is currently used by Radio B. The additional inputs for “ModOnOFf” which selects between modulated values and “Null” (no modulation) using a multiplexer (Mux). Radio B uses the querying algorithm 1 to determine the instances of BasicComponents, InputPorts, OutputPorts and their relationship by querying using the object property *isConnectedTo*. Figure 7.2 shows the ontology, one level deep into the modulator and Table 7.2 shows the classes of BasicComponents and Ports.

Using this example we show radio nodes can collaborate using common PHY structures which are inferred on-demand according to the requirement of the protocol being used to communicate. The next step towards making CODIPHY practical is to generate code for multiple target using the knowledge learned

Table 7.2: Hierarchical Inferencing using Queries

level	**Query**	**Class**	**Instance**
System	a) Waveform	Modified802.11	
	b) Modified802.11		*modified80211*
Subsystem	*isSubsystemOf* value *modified80211*	progMod_sub	*progMod*
Specification	*isSpecificationOf* value *progMod*	a) ModOnOff	*true*
		b) ModType	*bpsk, qpsk, qam16, qam64*
Dataflow	progMod_sub	Module	progMod
	isInPortOf value progMod		progMod_data, progMod_modOnOff, progMod_modType
	isOutPortOf value progMod		progMod_outReal, progMod_outImag
	isBlockOf value progMod		progMod_slicer, progMod_BasicMod
	isComponentOf value progMod		progMod_mux, progMod_null
	isInPortOf value progMod_slicer		progMod_slicer_In1
	isOutPortOf value progMod_slicer		progMod_slicer_Out1, progMod_slicer_Out2
	isInPortOf value progMod_BasicMod		progMod_BasicMod_In1, progMod_BasicMod_In2
	isOutPortOf value progMod_BasicMod		progMod_BasicMod_Out1
	isInPortOf value progMod_mux		progMod_mux_In1, progMod_mux_In2, progMod_mux_In3
	isOutPortOf value progMod_mux		progMod_mux_Out1
	isInPortOf value progMod_null		progMod_null_In1
	isOutPortOf value progMod_null		progMod_null_Out1
	isConnectedTo value progMod_data		
	isConnectedTo value progMod_modOnOff		
	isConnectedTo value progMod_modType		
	isConnectedTo value progMod_slicer_In1		progMod_mux_Out1
	isConnectedTo value progMod_slicer_Out1		progMod_slicer_Reinterpret_Out1
	isConnectedTo value progMod_slicer_Out2		progMod_slicer_Reinterpret1_Out1
	isConnectedTo value progMod_BasicMod_In1		progMod_modType
	isConnectedTo value progMod_BasicMod_In2		progMod_data
	isConnectedTo value progMod_BasicMod_Out1		progMod_BasicMod_select_modulation_Out1
	isConnectedTo value progMod_mux_In1		progMod_modOnOff
	isConnectedTo value progMod_mux_In2		progMod_null_Out1
	isConnectedTo value progMod_mux_In3		progMod_BasicMod_Out1
	isConnectedTo value progMod_mux_Out1		
	isConnectedTo value progMod_null_In1		
	isConnectedTo value progMod_null_Out1		

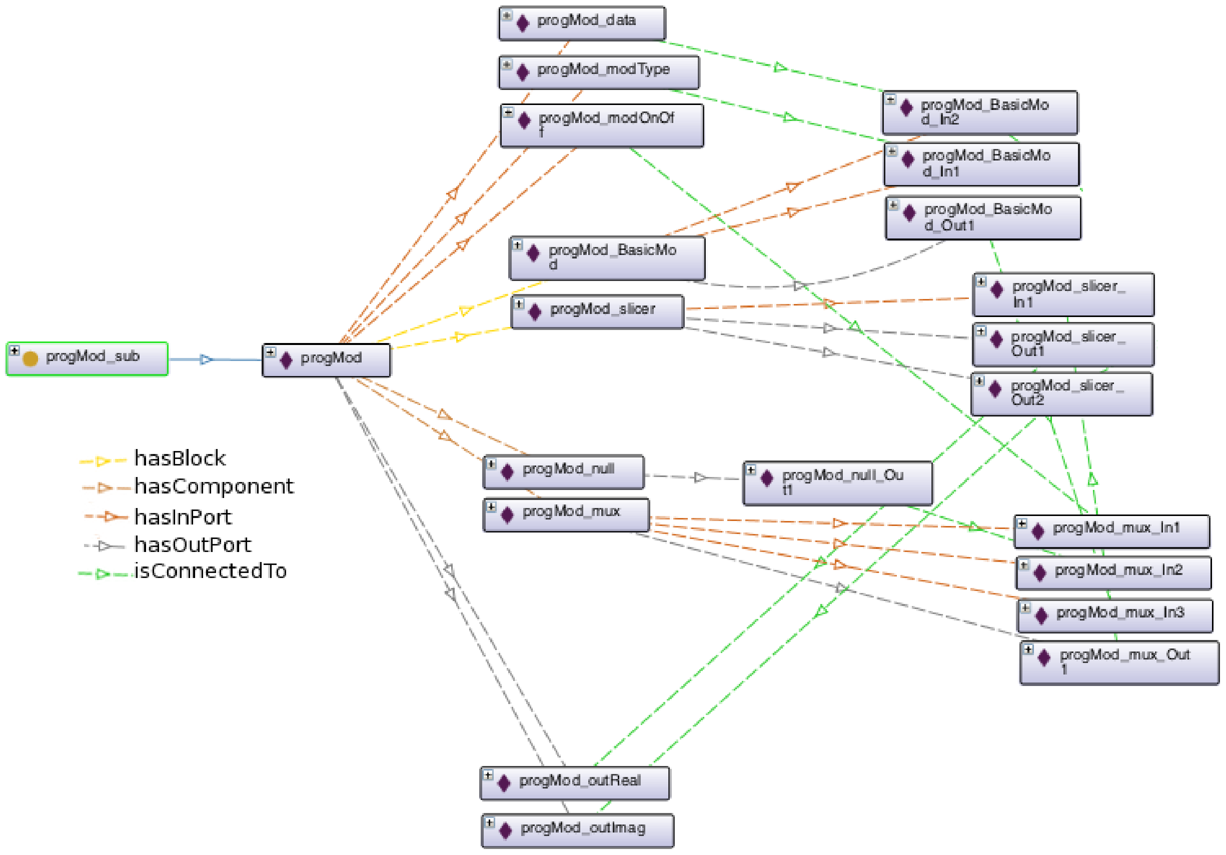


Figure 7.2: Dataflow ontology of the modulator for SMACK

at different levels of the knowledge base. Code generation is required for the dataflow information, while implementations at higher levels are made available as pre-compiled library of component that is readily used for implementing novel PHY.

Chapter 8

CODIPHY : Part 3 – Composing the Radio PHY

In chapters 6 and 7, we have shown a method to construct an ontology to represent a physical layer and to learn the relationship between various components by simple queries. However, composing an operational radio requires many stages of design and optimization and tight coordination of multiple devices and circuits, like the ADC/DAC, RF front-end, I/O interface etc. We believe that the engineering problem of implementing a fully functional radio can be broken down into smaller systems and the baseband is the most important part of it. Ontological description of a system of systems is required to solve the system design problem. In this thesis, we focus on composing the baseband part of the radio only and provide a method to synthesize functioning radio components for multiple targets.

Radio processing can be done on any platform, but often require domain specific optimization, e.g., implementing an FFT engine on a stream processor is completely different from that in a FPGA. Our vision of the methodology of CODIPHY is that the device specific implementation and optimization is not possible to solve from a high level description like an ontology. Instead, it is safe to assume that there will be some abstraction between the domain experts and the composing agent. Domain specific languages allow radio agents to reap the operational benefit of the products created by the language while hiding the functional or implementation details. This is typically done by employing function calls, APIs or pre-compiled quasi-static libraries. We use such an approach to compose the physical layer of a radio from an ontology.

8.1 Composing software executable

A list of components and their connections is obtained by querying the Dataflow level of the ontology as discussed in chapter 7. The result of the queries are used to generate software code, where the components denote function calls to pre-defined functions in the library. Algorithm 2 shows how C code is generated from the information of components, input ports, output ports and their connections. These information is a direct output of the querying process described in algorithm 1. The function declaration for a particular subsystem requires the input ports of the subsystem, and their data type as input arguments of the function. All the input ports and output ports of the subsystem are variables of the function, which will be assigned values later. The sequential processing required in software programs, requires tracking of where the data is available when the code is generated. Initially, the data is available only at the input ports of the subsystem. Then, from the **Connects** list, connections are made to the ports from the input port of the subsystem. Next, if all input ports of a component has data available, then the code for that component is generated, which is a function call from the library of basic components provided by the domain expert. After the code is generated, the data is now available at the output port of that component. This process of assigning connections and then generating the code for the component is repeated until the code for all the components have been generated.

Figure 8.1 shows the code generated for the packet detect unit from the dataflow given in the ontology as shown in figure 6.7, by using the automated querying mechanism described in algorithm 1, and the code generation process described in algorithm 2. It is evident that the generated code is not an optimized version, and requires domain expertise to optimize this implementation. However, the objective of CODIPHY is not to generate optimized code, instead it describes a backbone to represent the domain knowledge in a hierarchical fashion, such that a radio agent can learn and reconfigure its radio with varied parameters. The optimization engine for each hardware type can reside on top of CODIPHY to generate optimized code base for the target platform. We show the code generation process to verify that we can implement the dataflow from the ontology and is functionally correct for a particular signal processing application.

Algorithm 2: Algorithm to Generate C Code from Queries

```
Input: Comps, PortIns, PortOuts, Connects, MotherComp
/* Function Declaration                                                    */
Print PortOuts[MotherComp].dataType; Print PortOuts[MotherComp].name;
for pin in PortIns[MotherComp] do
    Print pin.dataType; Print pin.name;
end
/* Variable Declaration                                                    */
for c in Comps do
    if c not equal to MotherComp then
        for pin in PortIns[c] do
            Print pin.dataType; Print pin.name;
        end
    end
    for pout in PortOuts[c] do
        Print pout.dataType; Print pout.name;
    end
end
/* Initialize Data Ready Port & List of Code Generated Components
   */
for pin in PortIns[MotherComp] do
    dReady.append(pin);
end
cgen=[];
/* Body of the Code                                                        */
while len(cgen) $<$ len(Comps) do
    /* Make connection if data ready                                       */
    for cons in Connects do
        if cons.From in dReady and cons.To not in dReady then
            dReady.append(cons.To);
            Print cons.To, " = ", cons.From , ";" ;
        end
    end
    /* Generate code for a component if data is ready                      */
    for c in Comps do
        if c not in cgen then
            if ($\forall$ pin $\in$ PortIns[c]) in dReady then
                Print PortOuts[c].name, "=", c.funcName, "(", PortIns[c].name, c.vars, ");" ;
            end
            cgen.append(c);
            dReady.append(PortOuts[c]);
            break;
        end
    end
end
/* Return Statement                                                        */
Print "return", PortOuts[MotherComp].name;
```

```
void c_n(int* inputPX, int* outputPC)
{
    int delay1Out;
    Delay(inputPX, 64, &delay1Out);
    int ccMul1Out;
    ComplexConjugateMult(delay1Out, InputPX, &ccMul1Out);
    int movAvg1Out;
    MovingAverage(movAvg1In, 64, &movAvg1Out);
    *OutputPC = movAvg1Out;
}
```

Figure 8.1: Generated C Code for Packet Detector subsystem

8.2 Composing hardware descriptions

The high level description of radio subsystems using a collection of components and their interconnections facilitates the generation of structural HDLs. The set of basic components and aggregates, as defined in the ontology are mapped onto HDL entities while the interconnections between the ports are translated to physical wires. In this implementation, we use the Xilinx System Generator API [112] to programmatically generate synthesizable designs using Matlab function scripts. The APIs use the *xBlock, xSignal, xInport, and xOutport* objects to construct System Generator models. The various *Individuals* of basic components in the ontology are mapped onto the library blocks of System Generator. New library elements have been created for aggregates that are not a part of a standard installation of System Generator, e.g., Complex Multiplier, Magnitude, Moving Average, etc, which shows the that this technique can be generalized for composing any subsystem of a radio and abstractions can be applied to hide implementation details. The code example below, shows the programmatic synthesis of an *adder* with two inputs ‘a’ and ‘b’ and one output ‘s’ with a latency of ‘1’ clock cycle. The library component used in this case is ‘AddSub’.

```
[a, b] = xInport('a', 'b');

s = xOutport('s');

adder = xBlock('AddSub', struct('latency', 1), {a, b}, {s});
```

A Matlab M-function is generated automatically from the query results using a Python script that contains the instantiation of the different blocks and the signals that connect them. The algorithm to synthesize the Matlab script is the same as that for synthesizing C-code using algorithm 2, except for hardware synthesis the order of execution is not necessary as all the components run in parallel and simultaneously for every system clock pulse. This Matlab script is used to generate a System Generator model using APIs. The example below shows the programmatic creation of a System Generator model named 'synth_model' which contain the subsystem 'Mod'.

```
% create and save a new model
new\_system('synth\_model', 'model');
save\_system('synth\_model.mdl');

% create the block
cfg.source = str2func('Mod');
cfg.toplevel = 'synth\_model/Mod';
args = {};
xBlock(cfg, args);
```

The HDL is generated using the compile option in System Generator [112] to produce synthesizable VHDL code that can be implemented using standard tools. We use this method to ensure the correctness of the generated HDL and also provide a platform for users to test the subsystem for functional correctness as well.

8.3 Code generation for OFDM transceiver

The current toolflow of CODIPHY supports automatic code generation in C and VHDL. To generate an executable C code, each of the components in the ontology are mapped onto pre-defined functions with *inPorts* and *outPorts* as the input and output variables respectively. Each function is assigned to a new variable thus making them explicitly independent. It is our belief that further optimization will make this code more efficient by performing static and dynamic analysis. We do not address these optimization and leave that to domain experts. As of now, CODIPHY supports all the components and aggregates required to generate C code for the 802.11a/g transmitter and the library to compose the receiver is under construction. We report the code sizes for various transmitter blocks in Table 8.1. Although this implementation focuses on the sample domain subsystems, CODIPHY is generic enough to include libraries for binary domain systems as well and the methodology is same for that.

Module	Lines of Code
TxController	78
IFFT	21
Modulator	163
InsertGuard	43

Table 8.1: Lines of generated C Code from ontology

The hardware utilization for 802.11a/g largely depends on how the subsystems are designed. Various architectures are available to implement particular DSP subsystems. It is not our goal to build the most optimized system, but rather provide the toolflow to generate synthesizable design from a set of pre-compiled components. The interconnection of these components or the architecture is left to domain experts. We choose the dataflow represented in prior implementations of 802.11a/g in [74, 75]. Table 8.2 and Table 8.3 shows the hardware utilization for transmitter and receiver respectively on a Virtex V (LX110) FPGA.

Module	Slices	LUTs	BRAM	DSP48s
InMux	109(0.63)	247(0.36)	3(2.34)	0(0.00)
TxController	120(0.69)	280(0.41)	5(3.91)	0(0.00)
Mod	57(0.33)	97(0.14)	0(0.00)	0(0.00)
IFFT	1349(7.81)	2794(4.04)	4(3.13)	9(14.06)
InsertGuard	245(1.42)	616(0.89)	6(4.69)	0(0.00)

Numbers in parenthesis indicate percentage utilization

Table 8.2: Hardware Utilization of the synthesized OFDM transmitter subsystems

Module	Slices	LUTs	BRAM	DSP48s
Packet Detect	536(3.10)	1805(2.61)	0(0.00)	12(18.75)
Correlator	1816(10.51)	2884(4.17)	0(0.00)	2(3.13)
FFT	875(5.06)	1745(2.52)	5(3.91)	30(46.88)
Equalizer[1]	1197(6.93)	3356(4.86)	0(0.00)	4(6.25)
Demodulator	36(0.21)	100(0.14)	0(0.00)	0(0.00)
Decoder[2]	1319(7.63)	2644(3.83)	3(2.34)	0(0.00)
CRC check	114(0.66)	129(0.19)	0(0.00)	0(0.00)

Numbers in parenthesis indicate percentage utilization

[1] Includes channel estimation, [2] Includes De-interleaver and Viterbi decoder

Table 8.3: Hardware Utilization of the synthesized OFDM receiver subsystems

8.4 Dataflow based PHY adaptation for Crosslayer Implementations

In §6.4 we described the ontological description of the SMACK crosslayer protocol and in §7.5 we have shared the knowledge using queries, which is used to collaborate at the PHY to implement a common crosslayer technique. Sometimes, pre-fabricated subsystems are not available for a particular implementation. In these cases, the collaborating agent uses the knowledge at the Dataflow level to compose the subsystem from fundamental building blocks for that subsystem. An important requirement of composing radio components for multiple targets is functional correctness. In this section, we present the functional correctness of the code generation process by composing the subsystems required to use the SMACK crosslayer protocol. More generally speaking, it is the programmable modulator that allows selective modulation of the OFDM subcarriers. The SMACK signaling technique is a special case of this generic concept where the nodes use only one subcarrier to send simple binary responses to broadcast queries is wireless networks. Hence, we generate code for a programmable modulator that is used in a variety of applications that employs selective modulation to achieve different goals in wireless networks.

Using the querying process in Table 7.4 and using the System Generator APIs, the programmable modulator, (*ProgMod*), has been synthesized as shown in figure 8.2 and the corresponding M-Code that has been auto-generated from the query results, which in turn is used to synthesize the new subsystem is shown Appendix B. This Sysgen model is compiled using its native compiler to auto-generate VHDL code that is completely synthesizable onto FPGA using standard tools.

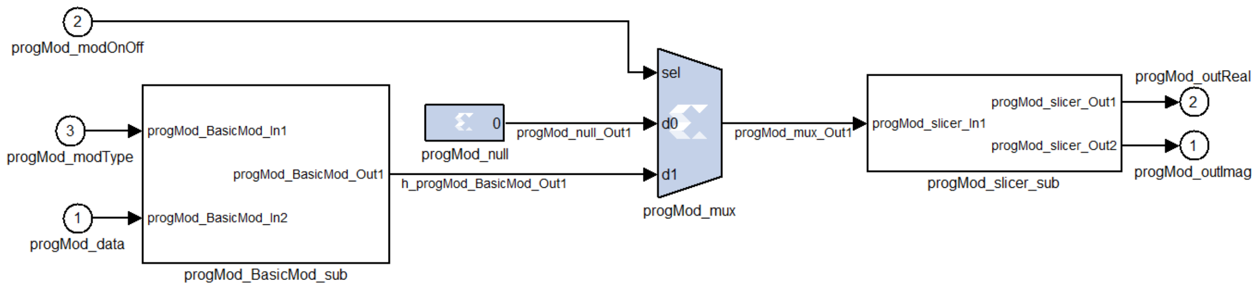
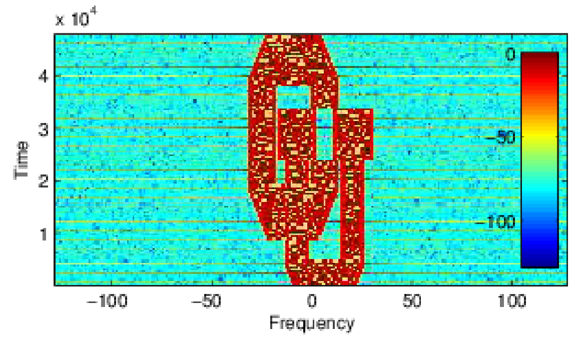


Figure 8.2: *ProgMod* subsystem synthesized from ontology in figure 6.11

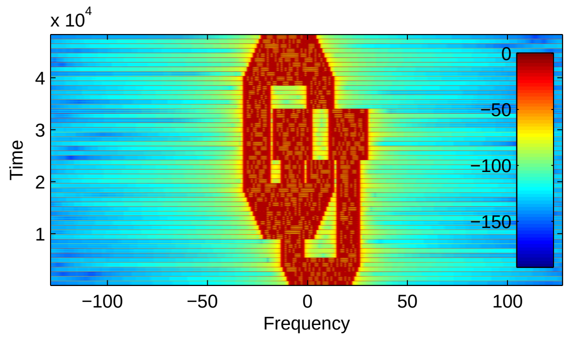
The C code generated from the results obtained from the same query is shown in Appendix C. The current software implementation uses floating point datatypes but fixed-point implementations can be derived by using the precision of various components given in form of *DataProperties* in the ontology. It is to be noted that the naming convention for all the ports (input and output variables of the component functions in software), component instances and the model hierarchy is identical for the two implementations. However, for more optimized performance further improvements can be made which is not within the scope of this thesis.

To show that the code generation phase of CODIPHY is correct, we show an example radio application and measure the accuracy and correctness of the hardware and software implementations. We choose a more complex form of PHY adaptation that requires synthesizing the *ProgMod* module for FPGA as well as software. We revisit the dynamic spectrum access example where certain OFDM subcarriers are nulled to make avoid the primary user of that spectrum. Figure 8.3 shows a comparison of the two implementations. Figure 8.3(a) and 8.3(b) shows the spectrogram for the waveform generated by the hardware and software implementation respectively. For this example, the inverse FFT is also synthesized for two targets along with the programmable modulator *ProgMod* to generate the time-domain signal. The time-domain, real and imaginary components of the transmitted signal (limited to 2000 samples for clarity) from the hardware is shown in figure 8.3(c), 8.3(e) and that from the software implementation is shown in 8.3(d) and 8.3(f) respectively. Finally, to show the correctness between the two implementations, we measure the difference between the two implementations. Figure 8.3(g) shows the absolute difference between the two signals. The difference is of the order of 10^{-5} which is insignificant to change the signal characteristics in time and frequency domain. Hence the two implementations preserve the integrity of the signal while being synthesized from a common high level representation. Figure 8.3(h) shows the spectrogram of the difference signal. The difference in time and frequency is approximately -75 dBm which is very desirable. The uniform noise-like distribution of the spectrogram specifies that the deviation in the two signals is uniform across time and frequency and the implementation does not introduce new impairment in the transmitted signal. The difference is attributed to the quantization error introduced by the fixed-point representation of the digital samples.

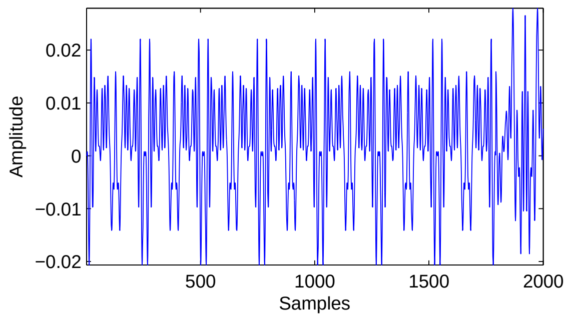
Therefore, using the automatic code generation infrastructure we can synthesize functioning, implementable and correct signal processing subsystems from a common ontological representation.



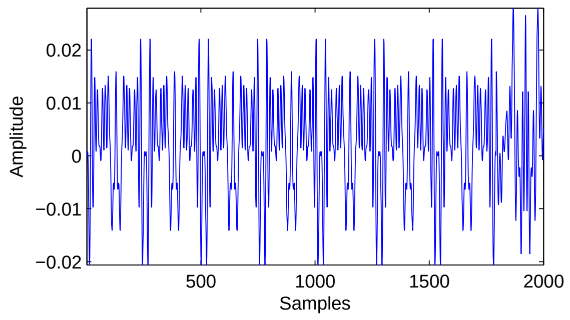
(a) Adaptive modulation using HW *ProgMod*



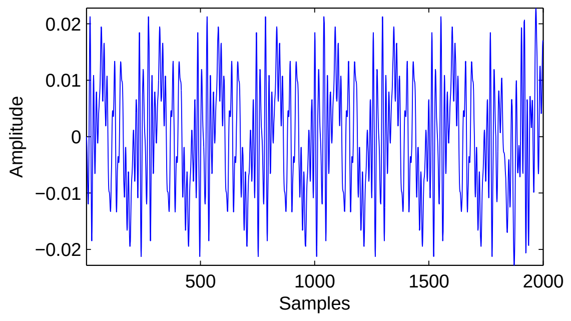
(b) Adaptive modulation using SW *ProgMod*



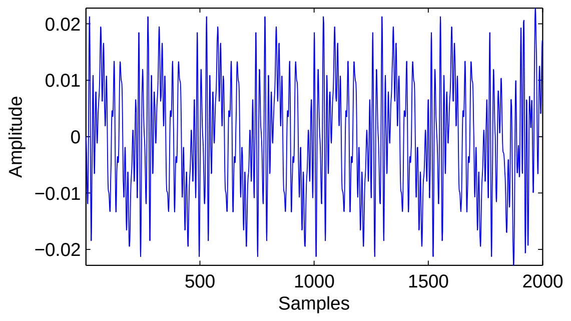
(c) Real component of the signal from HW



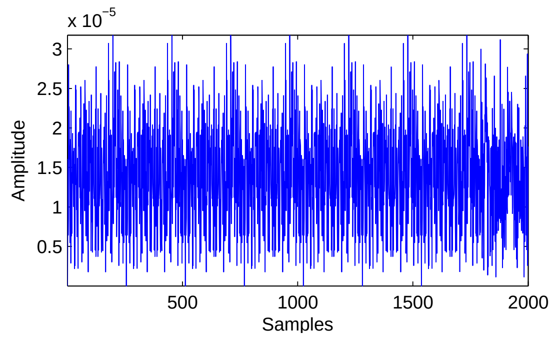
(d) Real component of the signal from SW



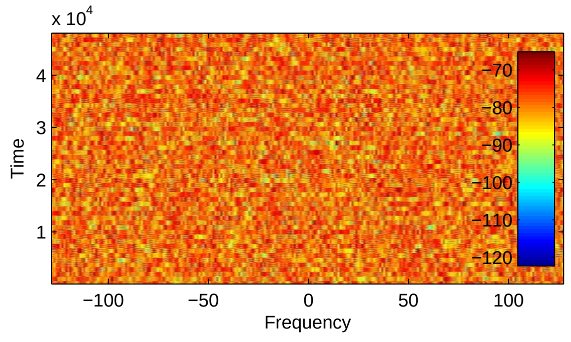
(e) Imaginary component of the signal from HW



(f) Imaginary component of the signal from SW



(g) Absolute difference between HW and SW *ProgMod*



(h) Difference spectrogram between HW and SW *ProgMod*

Figure 8.3: Performance comparison of HW and SW implementations for the *ProgMod* module

Chapter 9

Comparing CODIPHY With Other Related Techniques

The process of synthesizing radio PHY from a high level description is particularly useful in implementation of processing subsystems that was not originally designed for a particular radio node. CODIPHY is the first of its kind that provides a design methodology and framework allowing users, humans or cognitive radio agents, of various capabilities to *compose* functioning subsystems on multiple processing platforms. CODIPHY presents a hierarchical design approach that supercedes existing design methodologies of cognitive radios, which are fundamentally limited to pre-defined *tuning knobs*.

Conventional collaborative techniques [18], use reasoning framework to decide on the optimum parameters for a particular communication link. But the real challenge arises when these parameters are not present in the radio and needs to be synthesized from a set of design specifications. Fixed hardware pipeline is no longer optimum and there has to be a way to compose the radio structures on-demand, based on the requirement of the wireless environment. CODIPHY precisely solves this problem of active radio adaptation at the physical layer that previous techniques does not address.

In the literature, there are several examples of radio description language (RDL) [12, 23, 24, 22, 11], or waveform definition language (WDL) [40, 46] but all of these are tightly coupled to a particular hardware architecture and hence present considerable challenge in migrating a design between different hardware platforms, or collaboration between heterogeneous radio nodes. Clearly, these systems lack a common method of knowledge sharing that can be interpreted independent of the implementation. An approach for synthesizing the MAC has been described by Ansari, et al. [113], by representing the MAC layer components as distinct states connected by event based state transitions. Researchers [114], have also proposed domain specific language for designing MAC and PHY using a component based design, but are limited to the subsystem level only. In contrast, CODIPHY focuses on synthesizing PHY layer subsystems from fundamental building blocks to synthesize *"new"* PHY structures. Domain specific languages like CODIPHY, provide abstractions between various domains of radio engineering but still allows them to share the knowledge between these disparate areas of research.

Therefore, we summarize the novel contributions of CODIPHY as follows:

- Express the composition of a PHY at different levels of granularity that makes radio design accessible to a broader set of users.
- Compose new subsystems that were not built for a particular hardware platform using a high level, declarative representation of the internal structure.
- Share PHY layer information to collaborate using new crosslayer protocols making radios more powerful and agile.
- Provide a code generation framework to implement functioning radio subsystems for multiple hardware platforms (FPGA and general purpose processors)

To the best of our knowledge, CODIPHY is the only methodology that solves these practical constraints of cognitive radio networks and makes co-existence of heterogeneous cognitive radio nodes practical. This is done by incorporating intelligence at the PHY thus making it completely adaptive, malleable and synthesizable based on real-time demands of the radio environment.

Chapter 10

Conclusion

CODIPHY germinates from one of the fundamental requirements for wireless research, which is validation of novel concepts using prototypes and testbeds. The foundation of this thesis is the design and implementation of a prototype radio platform that provides a practical way to design, implement and evaluate high throughput OFDM based MAC-PHY crosslayer protocols. The transceiver has been meticulously tested and benchmarked to perform satisfactorily under normal wireless channel conditions and is also compatible with commodity WiFi devices.

The benefit from such a platform is quantified by its programmable features. We extend the basic transceiver to include *tuning knobs* that are not available in other platforms used for prototyping, hence hindering innovation. The possibilities with this platform expands beyond the standard features that are tied to a particular specification. In doing so, innovation is made possible through practical manifestation of the various research hypotheses.

An important component of this thesis is to utilize the prototype to innovate MAC-PHY crosslayer designs that provide unprecedented gains in various aspects of wireless and cognitive radio networks. By utilizing the programmable features of the platform, we make certain group communications faster in wireless networks using the "SMACK" technique. The "GRaTIS" and the "Dirty Constellation" utilize fine grained control over the modulator and demodulator to implement novel systems and are made practical through various radio experiments. We also contribute to the cognitive radio research by providing improvements by an order of magnitude over existing methods for NC-OFDM synchronization using the programmable correlator of the platform. Thus, the radio platform makes these research concepts more practical and concrete.

Experiences obtained from using the radio prototype for research has enabled us to re-think the way radios are built and cooperate in a heterogeneous environment consisting of radio nodes with varied capabilities. The concept of CODIPHY culminates by defining methods and design infrastructure to *compose* complex radio subsystems from a high-level specification that take advantages of the abstractions in the various domains of radio engineering. We use ontology to design a hierarchical knowledge representation system containing design information at different levels of complexity and granularity. The idea is clearly to make radio design more accessible to a broader community of wireless researchers. We also design a querying mechanism to retrieve these design information in a meaningful way and compose radio subsystems either from pre-compiled libraries or from fundamental components. We also take advantage of the component based design approach to retarget design to multiple implementations. Therefore, the knowledge sharing framework along with the automated code generation for radio systems allow active collaboration of heterogeneous devices at the physical layer by synthesizing subsystems that was not built for a particular radio node, thus making cognitive radios more adaptive, even at the PHY layer as well.

Therefore, this thesis presents unique contributions at various stages of CODIPHY: from PHY layers to making them composable. CODIPHY is a step towards making radio nodes malleable that molds new signal processing subsystems to collaborate with other nodes, not only on high level policies but also implement those in an efficient manner. From enabling broader realm of research in wireless networking to making radio design accessible to users with various expertise, CODIPHY has been instrumental in making these practical. It is our belief that CODIPHY is a promising step towards realizing the vision of practical cognitive radio networks by making radio nodes self-aware and intelligent.

Bibliography

[1] Mad-WiFi Project, “MadWiFi WLAN Driver.” [Online]. Available: http://madwifi-project.org/

[2] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald, “SoftMAC - Flexible Wireless Research Platform,” in Proceedings of the Fourth Workshop on Hot Topics in Networks (HotNets-IV), 2005.

[3] K. C.-J. Lin, N. Kushman, and D. Katabi, “ZipTx,” in Proceedings of the 14th ACM international conference on Mobile computing and networking - MobiCom ’08. New York, New York, USA: ACM Press, Sep. 2008, p. 351.

[4] E. Research, “GnuRadio.” [Online]. Available: http://www.gnu.org/software/gnuradio/

[5] A. Dutta, D. Saha, D. Grunwald, and D. Sicker, “An architecture for software defined cognitive radio,” in Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ser. ANCS ’10. New York, NY, USA: ACM, Sep. 2010.

[6] H. Rahul, N. Kushman, D. Katabi, C. Sodini, and F. Edalat, “Learning to Share: Narrowband-Friendly Wideband Networks,” in SIGCOMM ’08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication. New York, NY, USA: ACM, 2008, pp. 147–158.

[7] A. Dutta, D. Saha, D. Grunwald, and D. Sicker, “SMACK: a SMart ACKnowledgment scheme for broadcast messages in wireless networks,” in SIGCOMM ’09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication, vol. 39, no. 4. New York, NY, USA: ACM, 2009, pp. 15–26.

[8] S. Sen, R. Roy Choudhury, and S. Nelakuditi, “No time to countdown,” in Proceedings of the 17th annual international conference on Mobile computing and networking - MobiCom ’11. New York, New York, USA: ACM Press, Sep. 2011, p. 241.

[9] R. W. Chang, “Orthogonal Frequency Division Multiplexing,” U.S. Patent, Jan. 1970.

[10] H. Mahmoud, T. Yucek, and H. Arslan, “OFDM for cognitive radio: merits and challenges,” Wireless Communications, IEEE, vol. 16, no. 2, pp. 6–15, Apr. 2009.

[11] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker, “Sora: high performance software radio using general purpose multi-core processors,” in NSDI’09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation. Berkeley, CA, USA: USENIX Association, 2009, pp. 75–90.

[12] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A Low-power Architecture For Software Radio," in ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture. Washington, DC, USA: IEEE Computer Society, 2006, pp. 89–101.

[13] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins, "Deterministic parallel processing," Int. J. Parallel Program., vol. 34, no. 4, pp. 323–341, 2006.

[14] P. Murphy, A. Sabharwal, and B. Aazhang, "Design of WARP: A Flexible Wireless Open-Access Research Platform," in Proceedings of EUSIPCO, 2006.

[15] Xilinx, "ZYNC Processing Platform." [Online]. Available: http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm

[16] M. M. Kokar and L. Lechowicz, "Language Issues for Cognitive Radio," Proceedings of the IEEE, vol. 97, no. 4, pp. 689–707, 2009.

[17] K. B. M. Kokar, D. Brady, "Role of Ontologies in Cognitive Radios," in Cognitive Radio Technology, B. A. Fette, Ed. Elsevier Science Publishers B. V., 2009, ch. 13, pp. 401–428.

[18] S. Li, J. Moskal, M. M. Kokar, and D. Brady, "Collaborative Adaptation of Cognitive Radio Parameters Using Ontology and Policy Based Approach," Ph.D. dissertation, Northeastern University, Jul. 2011.

[19] Xilinx, "Xilinx System Generator for DSP User Guide (UG640)," vol. 640, 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/sysgen_user.pdf

[20] J. Eker, J. W. Janneck, E. A. Lee, J. I. E. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity The Ptolemy Approach," vol. 91, no. 1, 2003.

[21] A. Duller, D. Towner, G. Panesar, A. Gray, and W. Robbins, "picoArray Technology: The Tool's Story," in DATE '05: Proceedings of the conference on Design, Automation and Test in Europe. Washington, DC, USA: IEEE Computer Society, 2005, pp. 106–111.

[22] Vanu Inc, "Vanu Software Radio - http://www.vanu.org." [Online]. Available: http://www.vanu.org

[23] M. I. Anwar, S. Virtanen, and J. Isoaho, "A software defined approach for common baseband processing," in J. Syst. Archit., vol. 54, no. 8. New York, NY, USA: Elsevier North-Holland, Inc., 2008, pp. 769–786.

[24] A. Chun, "Key lessons from the scalable communications core: a reconfigurable wireless baseband," Communications Magazine, IEEE, vol. 48, no. 12, pp. 101–109, Dec. 2010.

[25] D. Lattard, E. Beigne, F. Clermidy, Y. Durand, R. Lemaire, P. Vivet, and F. Berens, "A Reconfigurable Baseband Platform Based on an Asynchronous Network-on-Chip," Solid-State Circuits, IEEE Journal of, vol. 43, no. 1, pp. 223–235, 2008.

[26] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn, "MAGALI: A Network-on-Chip based multi-core system-on-chip for MIMO 4G SDR," in IC Design and Technology (ICICDT), 2010 IEEE International Conference on, Jun. 2010, pp. 74–77.

[27] B. A. Dalio and K. A. Shelby, “The Implementation of OFDM Waveforms on an SDR Development Platfor Supporting a Massively Parallel Processor,” in SDR ’09: Proceedings of the Software Defined Radio Technical and Product Exposition, 2009.

[28] C. R. Anderson and E. G. Schaertl, “A Low-Cost Embedded SDR Solution for Prototyping and Experimentation,” in SDR ’09: Proceedings of the Software Defined Radio Technical and Product Exposition, 2009.

[29] A. Lodi, A. Cappelli, M. Bocchi, C. Mucci, M. Innocenti, C. De Bartolomeis, L. Ciccarelli, R. Giansante, A. Deledda, F. Campi, M. Toma, and R. Guerrieri, “{XiSystem}: a {XiRisc} based {SoC} with reconfigurable IO module,” in Solid-State Circuits, IEEE Journal of, vol. 41, no. 1, 2006, pp. 85–96.

[30] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal, “WARP, A Unified Wireless Network Testbed for Education and Research,” in Proceedings of IEEE MSE, 2007.

[31] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. M. Wyglinski, and A. Agah, “KUAR: A Flexible Software-Defined Radio Development Platform,” in New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on, Apr. 2007, pp. 428–439.

[32] “Xilinx Partial Reconfiguration of FPGAs.” [Online]. Available: http://www.xilinx.com/tools/partial-reconfiguration.htm

[33] J. Delorme, J. Martin, A. Nafkha, C. Moy, F. Clermidy, P. Leray, and J. Palicot, “A FPGA partial reconfiguration design approach for cognitive radio based on NoC architecture,” in Circuits and Systems and TAISA Conference, 2008. NEWCAS-TAISA 2008. 2008 Joint 6th International IEEE Northeast Workshop on, Jun. 2008, pp. 355–358.

[34] C. Hsu, F. Keceli, M. Ko, S. Shahparnia, and S. S. Bhattacharyya, “DIF: An interchange format for dataflow-based design tools,” in Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece, July 2004, pp. 423–432.

[35] C. Hsu, M. Ko, and S. S. Bhattacharyya, “Software synthesis from the dataflow interchange format,” in Proceedings of the International Workshop on Software and Compilers for Embedded Systems, Dallas, Texas, September 2005, pp. 37–49.

[36] S. S. Bhattacharyya, J. Eker, J. W. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet, “Overview of the MPEG Reconfigurable Video Coding Framework,” Journal of Signal Processing Systems, vol. 63, no. 2, pp. 251–263, Jul. 2009.

[37] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya, “Topological patterns for scalable representation and analysis of dataflow graphs,” Journal of Signal Processing Systems, vol. 65, no. 2, pp. 229–244, 2011.

[38] M. D. MCCool, “Signal processing and general-purpose computing on GPUs,” Signal Processing Magazine, IEEE, vol. 24, no. 3, pp. 109–114, May 2007.

[39] W. Thies and S. Amarasinghe, “An empirical characterization of stream programs and its implications for language and compiler design,” in Proceedings of the 19th international conference on Parallel

architectures and compilation techniques, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 365–376.

[40] "Rosetta Wikipedia." [Online]. Available: https://wiki.ittc.ku.edu/rosetta_wiki/index.php/Main_Page

[41] G. Kimmell, E. Komp, G. Minden, J. Evans, and P. Alexander, "Synthesizing software defined radio components from Rosetta," 2008 Forum on Specification, Verification and Design Languages, pp. 148–153, Sep. 2008.

[42] D. J. Claypool, T. J. McNevin, W. Liu, and K. M. McNeill, "Automated Software Defined Radio Deployment Using Domain Specific Modeling Languages," 2009 IEEE Mobile WiMAX Symposium, pp. 157–162, Jul. 2009.

[43] F. Plavec, Z. Vranesic, and S. Brown, "Towards compilation of streaming programs into FPGA hardware," in Specification, Verification and Design Languages, 2008. FDL 2008. Forum on, 2008, pp. 67–72.

[44] G. Mittal, D. Zaretsky, X. Tang, and P. Banerjee, "An Overview of a Compiler for Mapping Software Binaries to Hardware," Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 15, no. 11, pp. 1177–1190, 2007.

[45] G. Stitt and F. Vahid, "Hardware/software partitioning of software binaries," in Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ser. ICCAD '02. New York, NY, USA: ACM, 2002, pp. 164–170.

[46] "Bluespec Inc BSV: http://www.bluespec.com." [Online]. Available: http://www.bluespec.com/

[47] M. C. Ng and K. Fleming, "AirBlue : A High-Throughput and Low-Latency Radio Prototyping Platform," in Design Automation and Test in Europe, 2009.

[48] S. Feng, H. Zheng, H. Wang, J. Liu, and P. Zhang, "Preamble design for non-contiguous spectrum usage in cognitive radio networks," in WCNC'09: Proceedings of the 2009 IEEE conference on Wireless Communications & Networking Conference. Piscataway, NJ, USA: IEEE Press, 2009, pp. 705–710.

[49] K. E. Nolan, T. W. Rondeau, P. Sutton, and L. E. Doyle, "Tests and trials of software-defined and cognitive radio in ireland," in SDR Forum Technical Conference and Product Exposition, 2007.

[50] "Part 16: Air Interface for Broadband Wireless Access Systems." [Online]. Available: http://standards.ieee.org/getieee802/download/802.16-2009.pdf

[51] H. Rahul, F. Edalat, D. Katabi, and C. G. Sodini, "Frequency-aware rate adaptation and mac protocols," in Proceedings of the 15th annual international conference on Mobile computing and networking, ser. MobiCom '09. New York, NY, USA: ACM, 2009, pp. 193–204.

[52] D. Saha, A. Dutta, D. Grunwald, and D. Sicker, "PHY Aided MAC - a new paradigm," in INFOCOM 2009, IEEE, 2009, pp. 2986–2990.

[53] L. Yang, W. Hou, L. Cao, B. Y. Zhao, and H. Zheng, "Supporting Demanding Wireless Applications with Frequency-agile Radios," in Proc. of ACM/USENIX NSDI, San Jose, CA, Apr. 2010, p. 5.

[54] S. Rayanchu, V. Shrivastava, S. Banerjee, and R. Chandra, “FLUID : Improving Throughputs in Enterprise Wireless LANs through Flexible Channelization,” in Proceedings of the 17th annual international conference on Mobile computing and networking, 2011.

[55] L. E. Li, R. Alimi, R. Ramjee, J. Shi, Y. Sun, H. Viswanathan, and Y. R. Yang, “Superposition coding for wireless mesh networks,” in MobiCom ’07: Proceedings of the 13th annual ACM international conference on Mobile computing and networking. New York, NY, USA: ACM, 2007, pp. 330–333.

[56] N. Shacham, “Multipoint communication by hierarchically encoded data,” in INFOCOM ’92. Eleventh Annual Joint Conference of the IEEE Computer and Communications Societies, IEEE, May 1992, pp. 2107–2114 vol.3.

[57] D. Halperin, T. Anderson, and D. Wetherall, “Taking the sting out of carrier sense: interference cancellation for wireless LANs,” in Proceedings of the 14th ACM international conference on Mobile computing and networking, ser. MobiCom ’08. New York, NY, USA: ACM, 2008, pp. 339–350.

[58] S. Jakubczak and D. Katabi, “Softcast: one-size-fits-all wireless video,” in Proceedings of the ACM SIGCOMM 2010 conference, ser. SIGCOMM ’10. New York, NY, USA: ACM, 2010, pp. 449–450.

[59] S. Sen, N. Santhapuri, R. R. Choudhury, and S. Nelakuditi, “Accurate: constellation based rate estimation in wireless networks,” in Proceedings of the 7th USENIX conference on Networked systems design and implementation, ser. NSDI’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 12–12.

[60] Overview of the 3GPP Long Term Evolution Physical Layer, Freescale Semiconductors. [Online]. Available: http://www.freescale.com/files/wireless_comm/doc/white_paper/3GPPEVOLUTIONWP.pdf

[61] D. Saha, A. Dutta, D. Grunwald, and D. Sicker, in New Frontiers in Dynamic Spectrum Access Networks (DySPAN), 2011 IEEE Symposium on, 2011, pp. 552–563.

[62] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh, “White space networking with wi-fi like connectivity,” in Proceedings of the ACM SIGCOMM 2009 conference on Data communication, ser. SIGCOMM ’09. New York, NY, USA: ACM, 2009, pp. 27–38.

[63] S. Gollakota and D. Katabi, “Zigzag Decoding: Combating Hidden Terminals in Wireless Networks,” in SIGCOMM ’08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication. New York, NY, USA: ACM, 2008, pp. 159–170.

[64] T. Moscibroda, R. Ch, Y. Wu, S. Sengupta, P. Bahl, and Y. Yuan, “Load-aware spectrum distribution in wireless lans,” in ICNP 08.

[65] P. D. Sutton, B. Ozgul, K. E. Nolan, and L. E. Doyle, “Bandwidth-Adaptive Waveforms for Dynamic Spectrum Access Networks,” in New Frontiers in Dynamic Spectrum Access Networks, 2008. DySPAN 2008. 3rd IEEE Symposium on, 2008, pp. 1–7.

[66] S. Coleri, M. Ergen, A. Puri, and A. Bahai, “Channel estimation techniques based on pilot arrangement in OFDM systems,” in Broadcasting, IEEE Transactions on, vol. 48, pp. 223–229.

[67] D. Halperin, W. Hu, A. Sheth, and D. Wetherall, “Predictable 802.11 packet delivery from wireless channel measurements,” in Proceedings of the ACM SIGCOMM 2010 conference, ser. SIGCOMM ’10. New York, NY, USA: ACM, 2010, pp. 159–170.

[68] M. Vutukuru, H. Balakrishnan, and K. Jamieson, “Cross-layer wireless bit rate adaptation,” in SIGCOMM ’09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication. New York, NY, USA: ACM, 2009, pp. 3–14.

[69] S. Katti, D. Katabi, H. Balakrishnan, and M. Medard, “Symbol-level network coding for wireless mesh networks,” SIGCOMM Comput. Commun. Rev., vol. 38, no. 4, pp. 401–412, 2008.

[70] A. Niktash, H. Parizi, A. H. Kamalizad, and N. Bagherzadeh, “RECFEC: A Reconfigurable FEC Processor for Viterbi, Turbo, Reed-Solomon and LDPC Coding,” in WCNC, 2008, pp. 605–610.

[71] R. Tandra and A. Sahai, “SNR Walls for Feature Detectors,” in New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on, 2007, pp. 559–570.

[72] K. C.-J. Lin, S. Gollakota, and D. Katabi, “Random access heterogeneous MIMO networks,” in Proceedings of the ACM SIGCOMM 2011 conference on SIGCOMM - SIGCOMM ’11, vol. 41, no. 4. New York, New York, USA: ACM Press, Aug. 2011, p. 146.

[73] S. Gollakota and D. Katabi, “Physical layer wireless security made fast and channel independent,” in 2011 Proceedings IEEE INFOCOM. IEEE, Apr. 2011, pp. 1125–1133.

[74] J. Fifield, P. Kasemir, D. Grunwald, and D. Sicker, “Experiences with a platform for frequency agile techniques,” in New Frontiers in Dynamic Spectrum Access Networks (DySPAN), 2011 IEEE Symposium on, 2007.

[75] A. Dutta, J. Fifield, G. Schelle, D. Grunwald, and D. Sicker, “An Intelligent Physical Layer For Cognitive Radio Networks,” in WICON ’08: Proceedings of the 4th international conference on Wireless internet. New York, NY, USA: ACM, 2008.

[76] S. Li, M. M. Kokar, J. Moskal, and L. Pucker, “Now Radios can understand each other: Modeling language for mobility,” FierceWireless: The Wireless Industrial Daily Monitor (White Paper), 21.

[77] M. M. . K. Shujun Li and J. Moskal, “Policy-driven Ontology-based Radio: A Public Safety Use Case,” Proceedings of the Software Defined Radio Technical and Product Exposition Forum, 2008.

[78] “Description of the Cognitive Radio Ontology,” Wireless Innovation Forum, no. September, 2010.

[79] “IEEE Std 802.11-2007, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications.” [Online]. Available: http://standards.ieee.org/

[80] Xilinx, “Xilinx-Virtex-5-LX-Evaluation-Kit.” [Online]. Available: http://www.em.avnet.com/en-us/design/drc/Pages/Xilinx-Virtex-5-LX-Evaluation-Kit.aspx

[81] T. M. Schmidl and D. C. Cox, “Low-overhead, low-complexity [burst] synchronization for OFDM,” in Communications, 1996. ICC 96, Conference Record, Converging Technologies for Tomorrow’s Applications. 1996 IEEE International Conference on, vol. 3, pp. 1301—-1306 vol.3.

[82] C. Dick and F. Harris, “FPGA implementation of an OFDM PHY,” in Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on, vol. 1, 2003, pp. 905—-909Vol.1.

[83] M. Speth, S. Fechtel, G. Fock, and H. Meyr, “Optimum receiver design for wireless broad-band systems using ofdm. I,” Communications, IEEE Transactions on, vol. 47, no. 11, pp. 1668–1677, 1999.

[84] ——, "Optimum receiver design for ofdm-based broadband transmission .II. a case study," Communications, IEEE Transactions on, vol. 49, no. 4, pp. 571–578, 2001.

[85] A. Troya, K. Maharatna, M. Krstic, E. Grass, and R. Kraemer, "OFDM synchronizer implementation for an IEEE 802.11(a) compliant modem," in IASTED International Conference on Wireless and Optical Communication, 2002. IASTED, 2002, pp. 152–157.

[86] M. Wouters, G. Vanwijnsberghe, P. Van Wesemael, T. Huybrechts, and S. Thoen, "Real time implementation on FPGA of an OFDM based wireless LAN modem extended with adaptive loading," in Solid-State Circuits Conference, 2002. ESSCIRC 2002. Proceedings of the 28th European, pp. 531–534.

[87] M. Serra, P. Marti, and J. Carrabina, "Implementation of a channel equalizer for OFDM wireless LANs," in Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on, pp. 232–238.

[88] "Perl Script for Viterbi Decoder -." [Online]. Available: http://viterbi-gen.sourceforge.net/

[89] M. Casado, T. Koponen, D. Moon, and S. Shenker, "Rethinking packet forwarding hardware," in Proc. Seventh ACM SIGCOMM HotNets Workshop, 2008.

[90] A. Chun, E. Tsui, I. Chen, H. Honary, and J. Lin, "Application of the Intel reconfigurable communications architecture to 802.11a, 3G and 4G standards," in Emerging Technologies: Frontiers of Mobile and Wireless Communication, 2004. Proceedings of the IEEE 6th Circuits and Systems Symposium on, vol. 2, 2004, pp. 659–662 Vol.2.

[91] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," Solid-State Circuits, IEEE Journal of, vol. 38, no. 11, pp. 1876–1886, 2003.

[92] L. Rizzo, "On the feasibility of software FEC," Tech. Rep., 1997.

[93] F. Ge, A. Young, T. Brisebois, Q. Chen, and C. W. Bostian, "Software Defined Radio Execution Latency," in in Proc. of Software Defined Radio Technical Conference, 2008.

[94] C. E. Perkins and E. M. Royer, "Adhoc On-Demand Distance Vector Routing," in Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, 1999, pp. 90–100.

[95] Shared Spectrum Company, "Spectrum Occupancy Measurements." [Online]. Available: http://www.sharedspectrum.com/measurements/

[96] K. Harrison, S. M. Mishra, and A. Sahai, "How Much White-Space Capacity Is There?" in New Frontiers in Dynamic Spectrum, 2010 IEEE Symposium on, 2010, pp. 1–10.

[97] A. Dutta, D. Saha, D. Grunwald, and D. Sicker, "Practical Implementation of Blind Synchronization in NC-OFDM based Cognitive Radio Networks," in CORONET. New York, NY, USA: ACM, 2010.

[98] T. Chiueh and T. P, Chapter : Synchronization. John Wiley and Sons (Asia) Pte. Ltd., 2007.

[99] FCC Ruling on Unlicensed operation in TV whitespaces, Federal Communication Commission. [Online]. Available: http://www.fcc.gov/Daily_Releases/Daily_Business/2010/db1025/FCC-10-174A1.pdf

[100] D. Saha, A. Dutta, D. Grunwald, and D. Sicker, "Gratis: Sensing and intelligence for performance in the presence of legacy networks," in Cognitive Radio Oriented Wireless Networks and Communications (CROWNCOM), 2012 7th International ICST Conference on, 2012, pp. 42–47.

[101] A. Schulman, D. Levin, and N. Spring, "CRAWDAD trace set umd/sigcomm2008/pcap (v. 2009-03-02)," Downloaded from http://crawdad.cs.dartmouth.edu/umd/sigcomm2008/pcap, Mar. 2009.

[102] D. Saha, A. Dutta, D. Grunwald, and D. Sicker, "Secret Agent Radio:Covert communication through dirty constellations," in Information Hiding Conference. Springer Berlin Heidelberg, 2012, pp. 160–175.

[103] V. Brik, S. Banerjee, M. Gruteser, and S. Oh, "Wireless device identification with radiometric signatures," in Proceedings of the 14th ACM international conference on Mobile computing and networking, ser. MobiCom '08. New York, NY, USA: ACM, 2008, pp. 116–127.

[104] C. Krätzer, J. Dittmann, A. Lang, and T. Kühne, "WLAN steganography: a first practical review," in MM&S;Sec '06: Proceedings of the 8th workshop on Multimedia and security. New York, NY, USA: ACM, 2006, pp. 17–22.

[105] K. Ahsan and D. Kundur, "Practical Data Hiding in TCP/IP," in Proc. Workshop on Multimedia Security at ACM Multimedia '02, French Riviera, Dec. 2002.

[106] R. K. Ganti, Z. Gong, M. Haenggi, C.-H. Lee, S. Srinivasa, D. Tisza, S. Vanka, and P. Vizi, "Implementation and Experimental Results of Superposition Coding on Software Radio," in 2010 IEEE International Conference on Communications (ICC'10), Cape Town, South Africa, May 2010.

[107] J. Mitola III and H. Man, "Semantics in Cognitive Radio," 2009 IEEE International Conference on Semantic Computing, pp. 261–266, Sep. 2009.

[108] N. F. Noy and D. L. Mcguinness, "Ontology Development 101 : A Guide to Creating Your First Ontology," pp. 1–25, 2000. [Online]. Available: http://www.ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html

[109] W3C, "OWL Web Ontology Language." [Online]. Available: http://www.w3.org/TR/owl-features/

[110] I. P. Conference, N. Drummond, M. Horridge, and H. Knublauch, "Protégé-OWL Tutorial," no. July, pp. 1–41, 2005.

[111] U. of Manchester, "OWL API." [Online]. Available: http://owlapi.sourceforge.net/

[112] Xilinx, "Xilinx System Generator for DSP Reference Guide," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/sysgen_ref.pdfhttp://www.xilinx.com/tools/sysgen.htm

[113] X. Zhang, J. Ansari, and P. Mähönen, "Demo: runtime mac reconfiguration using a meta-compiler assisted toolchain," in Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 277–278.

[114] X. Zhang, J. Ansari, G. Yang, and P. Mahonen, "Trump: Supporting efficient realization of protocols for cognitive radio networks," in New Frontiers in Dynamic Spectrum Access Networks (DySPAN), 2011 IEEE Symposium on, 2011, pp. 476–487.

[115] A. Dutta, D. Saha, D. Grunwald, and D. Sicker, "CODIPHY – Composing On-Demand Intelligent Physical Layers," in Workshop of Software Radio Implementation Forum, to appear. ACM, 2013.

Appendix A

Previously Published Content

Chapter 3 revises previous publication, An Intelligent Physical Layer for Cognitive Radio Networks [75]. Aveek Dutta, Jeffery Fifield, Graham Schelle, Dirk Grunwald, Douglas Sicker in ACM WICON 2008.

Chapter 4 revises previous publication, An Architecture for Software Defined Cognitive Radio [5]. Aveek Dutta, Dola Saha, Dirk Grunwald, Douglas Sicker in ACM/IEEE ANCS 2010.

Chapter 5 revises previous publications,

(1) SMACK : A SMart ACKnowledgment Scheme for Broadcast Messages in Wireless Networks [7]. Aveek Dutta, Dola Saha, Dirk Grunwald, Douglas Sicker in ACM SIGCOMM 2009.

(2) Blind Synchronization for NC-OFDM – When ”Channels” Are Conventions, Not Mandates [61]. Dola Saha, Aveek Dutta, Dirk Grunwald, Douglas Sicker in IEEE DySPAN 2011.

(3) GRaTIS : Free Bits in the Network [100]. Dola Saha, Aveek Dutta, Dirk Grunwald, Douglas Sicker in Transactions of Mobile Computing 2012.

(4) Secret Agent Radio : Covert communication Through Dirty Constellations [102]. Aveek Dutta, Dola Saha, Dirk Grunwald, Douglas Sicker in IEEE Information Hiding 2012.

Chapter 6, 7 and 8 partially revises previous publication, CODIPHY – Composing On-Demand Intelligent Physical Layers [115]. Aveek Dutta, Dola Saha, Dirk Grunwald, Douglas Sicker in ACM Software Radio Implementation Forum 2013.

Appendix B

Autogenerated M-Code for synthesizing *ProgMod* subsystem

Listing B.1: M-Code to synthesize the System Generator model for the programmable modulator

```
function progMod()

    progMod_data = xInport('progMod_data');
    progMod_modOnOff = xInport('progMod_modOnOff');
    progMod_modType = xInport('progMod_modType');
    progMod_outImag = xOutport('progMod_outImag');
    progMod_outReal = xOutport('progMod_outReal');

    progMod_mux_Out1 = xSignal('progMod_mux_Out1');
    progMod_slicer_sub = xBlock(struct('source', @progMod_slicer, 'name', 'progMod_slicer_sub'),
        ...
    {}, ...
    {progMod_mux_Out1}, ...
    {progMod_outReal, progMod_outImag});

    h_progMod_BasicMod_Out1 = xSignal('h_progMod_BasicMod_Out1');
    progMod_BasicMod_sub = xBlock(struct('source', @progMod_BasicMod, 'name', '
        progMod_BasicMod_sub'), ...
    {}, ...
    {progMod_modType, progMod_data}, ...
    {h_progMod_BasicMod_Out1});

    progMod_null_Out1 = xSignal('progMod_null_Out1');
    progMod_mux = xBlock(struct('source', 'Mux', 'name', 'progMod_mux'), ...
    struct('arith_type', 'Signed␣␣(2''s␣comp)', ...
    'bin_pt', 2, ...
    'n_bits', 8), ...
    {progMod_modOnOff, progMod_null_Out1, h_progMod_BasicMod_Out1}, ...
    {progMod_mux_Out1});

    progMod_null_In1 = xSignal('progMod_null_In1');
    progMod_null = xBlock(struct('source', 'Constant', 'name', 'progMod_null'), ...
    struct('bin_pt', 0, ...
    'period', T_xcvr, ...
    'explicit_period', 'on', ...
    'const', 0, ...
    'n_bits', 32, ...
    'arith_type', 'Unsigned'), ...
    {progMod_null_In1}, ...
    {progMod_null_Out1});

    function progMod_slicer()

        progMod_slicer_In1 = xInport('progMod_slicer_In1');
        progMod_slicer_Out1 = xOutport('progMod_slicer_Out1');
        progMod_slicer_Out2 = xOutport('progMod_slicer_Out2');

        progMod_slicer_slice_imag_Out1 = xSignal('progMod_slicer_slice_imag_Out1');
        progMod_slicer_Reinterpret1 = xBlock(struct('source', 'Reinterpret', 'name', '
            progMod_slicer_Reinterpret1'), ...
        struct('force_arith_type', 'on', ...
        'force_bin_pt', 'on', ...
        'bin_pt', 15, ...
        'arith_type', 'Signed__(2''s_comp)'), ...
        {progMod_slicer_slice_imag_Out1}, ...
        {progMod_slicer_Out2});

        progMod_slicer_slice_imag = xBlock(struct('source', 'Slice', 'name', '
            progMod_slicer_slice_imag'), ...
        struct('nbits', 16, ...
        'mode', 'Lower_Bit_Location_+_Width', ...
        'bit1', 16), ...
        {progMod_slicer_In1}, ...
        {progMod_slicer_slice_imag_Out1});

        progMod_slicer_slice_real_Out1 = xSignal('progMod_slicer_slice_real_Out1');
        progMod_slicer_slice_real = xBlock(struct('source', 'Slice', 'name', '
            progMod_slicer_slice_real'), ...
        struct('nbits', 16), ...
        {progMod_slicer_In1}, ...
        {progMod_slicer_slice_real_Out1});

        progMod_slicer_Reinterpret = xBlock(struct('source', 'Reinterpret', 'name', '
            progMod_slicer_Reinterpret'), ...
        struct('force_arith_type', 'on', ...
        'force_bin_pt', 'on', ...
        'bin_pt', 15, ...
        'arith_type', 'Signed__(2''s_comp)'), ...
        {progMod_slicer_slice_real_Out1}, ...
        {progMod_slicer_Out1});

    end

    function progMod_BasicMod()

        progMod_BasicMod_In1 = xInport('progMod_BasicMod_In1');
        progMod_BasicMod_In2 = xInport('progMod_BasicMod_In2');
        progMod_BasicMod_Out1 = xOutport('progMod_BasicMod_Out1');

        progMod_BasicMod_block2_Out1 = xSignal('progMod_BasicMod_block2_Out1');
        h_progMod_BasicMod_QAM16_Out1 = xSignal('h_progMod_BasicMod_QAM16_Out1');
        progMod_BasicMod_QAM16_sub = xBlock(struct('source', @progMod_BasicMod_QAM16, 'name', '
            progMod_BasicMod_QAM16_sub'), ...
        {}, ...
        {progMod_BasicMod_block2_Out1}, ...
        {h_progMod_BasicMod_QAM16_Out1});

        h_progMod_BasicMod_BPSK_Out1 = xSignal('h_progMod_BasicMod_BPSK_Out1');
        progMod_BasicMod_BPSK_sub = xBlock(struct('source', @progMod_BasicMod_BPSK, 'name', '
            progMod_BasicMod_BPSK_sub'), ...
        {}, ...
        {progMod_BasicMod_block2_Out1}, ...
        {h_progMod_BasicMod_BPSK_Out1});

        h_progMod_BasicMod_QAM64_Out1 = xSignal('h_progMod_BasicMod_QAM64_Out1');
        progMod_BasicMod_QAM64_sub = xBlock(struct('source', @progMod_BasicMod_QAM64, 'name', '
            progMod_BasicMod_QAM64_sub'), ...
        {}, ...
        {progMod_BasicMod_block2_Out1}, ...
        {h_progMod_BasicMod_QAM64_Out1});

        progMod_BasicMod_block2 = xBlock(struct('source', 'Slice', 'name', '
            progMod_BasicMod_block2'), ...
        struct('nbits', 7, ...
        'mode', 'Lower␣Bit␣Location␣+␣Width'), ...
        {progMod_BasicMod_In2}, ...
        {progMod_BasicMod_block2_Out1});

        progMod_BasicMod_block1_Out1 = xSignal('progMod_BasicMod_block1_Out1');
        h_progMod_BasicMod_QPSK_Out1 = xSignal('h_progMod_BasicMod_QPSK_Out1');
        progMod_BasicMod_select_modulation = xBlock(struct('source', 'Mux', 'name', '
            progMod_BasicMod_select_modulation'), ...
        struct('inputs', '4', ...
        'arith_type', 'Signed␣␣(2''s␣comp)', ...
        'bin_pt', 2, ...
        'n_bits', 8), ...
        {progMod_BasicMod_block1_Out1, h_progMod_BasicMod_QAM64_Out1,
            h_progMod_BasicMod_QAM16_Out1, h_progMod_BasicMod_QPSK_Out1,
            h_progMod_BasicMod_BPSK_Out1}, ...
        {progMod_BasicMod_Out1});

        progMod_BasicMod_block1 = xBlock(struct('source', 'Slice', 'name', '
            progMod_BasicMod_block1'), ...
        struct('nbits', 2, ...
        'mode', 'Lower␣Bit␣Location␣+␣Width'), ...
        {progMod_BasicMod_In1}, ...
        {progMod_BasicMod_block1_Out1});

```

```
progMod_BasicMod_QPSK_sub = xBlock(struct('source', @progMod_BasicMod_QPSK, 'name', '
     progMod_BasicMod_QPSK_sub'), ...
{}, ...
{progMod_BasicMod_block2_Out1}, ...
{h_progMod_BasicMod_QPSK_Out1});

function progMod_BasicMod_QAM16()

    progMod_BasicMod_QAM16_In1 = xInport('progMod_BasicMod_QAM16_In1');
    progMod_BasicMod_QAM16_Out1 = xOutport('progMod_BasicMod_QAM16_Out1');

    progMod_BasicMod_QAM16_QAM16_sub = xBlock(struct('source',
         @progMod_BasicMod_QAM16_QAM16, 'name', 'progMod_BasicMod_QAM16_QAM16_sub'), ...
    {}, ...
    {progMod_BasicMod_QAM16_In1}, ...
    {progMod_BasicMod_QAM16_Out1});

    function progMod_BasicMod_QAM16_QAM16()

        progMod_BasicMod_QAM16_QAM16_In1 = xInport('progMod_BasicMod_QAM16_QAM16_In1');
        progMod_BasicMod_QAM16_QAM16_Out1 = xOutport('progMod_BasicMod_QAM16_QAM16_Out1')
            ;

        progMod_BasicMod_QAM16_QAM16_Constant1_In1 = xSignal('
            progMod_BasicMod_QAM16_QAM16_Constant1_In1');
        progMod_BasicMod_QAM16_QAM16_Constant1_Out1 = xSignal('
            progMod_BasicMod_QAM16_QAM16_Constant1_Out1');
        progMod_BasicMod_QAM16_QAM16_Constant1 = xBlock(struct('source', 'Constant', '
            name', 'progMod_BasicMod_QAM16_QAM16_Constant1'), ...
        struct('bin_pt', 15, ...
        'const', -3*(1/sqrt(10))*(1/sqrt(13/6)), ...
        'period', T_xcvr, ...
        'explicit_period', 'on'), ...
        {progMod_BasicMod_QAM16_QAM16_Constant1_In1}, ...
        {progMod_BasicMod_QAM16_QAM16_Constant1_Out1});

        progMod_BasicMod_QAM16_QAM16_Constant2_In1 = xSignal('
            progMod_BasicMod_QAM16_QAM16_Constant2_In1');
```

```
progMod_BasicMod_QAM16_QAM16_Constant2_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Constant2_Out1');
progMod_BasicMod_QAM16_QAM16_Constant2 = xBlock(struct('source', 'Constant', '
     name', 'progMod_BasicMod_QAM16_QAM16_Constant2'), ...
struct('bin_pt', 15, ...
'const', -1*(1/sqrt(10))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM16_QAM16_Constant2_In1}, ...
{progMod_BasicMod_QAM16_QAM16_Constant2_Out1});

progMod_BasicMod_QAM16_QAM16_Constant3_In1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Constant3_In1');
progMod_BasicMod_QAM16_QAM16_Constant3_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Constant3_Out1');
progMod_BasicMod_QAM16_QAM16_Constant3 = xBlock(struct('source', 'Constant', '
     name', 'progMod_BasicMod_QAM16_QAM16_Constant3'), ...
struct('bin_pt', 15, ...
'const', 1*(1/sqrt(10))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM16_QAM16_Constant3_In1}, ...
{progMod_BasicMod_QAM16_QAM16_Constant3_Out1});

progMod_BasicMod_QAM16_QAM16_Reinterpret_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Reinterpret_Out1');
progMod_BasicMod_QAM16_QAM16_Reinterpret1_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Reinterpret1_Out1');
progMod_BasicMod_QAM16_QAM16_Concat = xBlock(struct('source', 'Concat', 'name', '
     progMod_BasicMod_QAM16_QAM16_Concat'), ...
[], ...
{progMod_BasicMod_QAM16_QAM16_Reinterpret_Out1,
     progMod_BasicMod_QAM16_QAM16_Reinterpret1_Out1}, ...
{progMod_BasicMod_QAM16_QAM16_Out1});

progMod_BasicMod_QAM16_QAM16_b0b1_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_b0b1_Out1');
```

```
progMod_BasicMod_QAM16_QAM16_b0b1 = xBlock(struct('source', 'Slice', 'name', '
     progMod_BasicMod_QAM16_QAM16_b0b1'), ...
struct('nbits', 2, ...
'mode', 'Lower␣Bit␣Location␣+␣Width', ...
'bit0', 2), ...
{progMod_BasicMod_QAM16_QAM16_In1}, ...
{progMod_BasicMod_QAM16_QAM16_b0b1_Out1});

progMod_BasicMod_QAM16_QAM16_Mux4_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Mux4_Out1');
progMod_BasicMod_QAM16_QAM16_Reinterpret = xBlock(struct('source', 'Reinterpret',
       'name', 'progMod_BasicMod_QAM16_QAM16_Reinterpret'), ...
struct('force_arith_type', 'on', ...
'force_bin_pt', 'on'), ...
{progMod_BasicMod_QAM16_QAM16_Mux4_Out1}, ...
{progMod_BasicMod_QAM16_QAM16_Reinterpret_Out1});

progMod_BasicMod_QAM16_QAM16_Mux1_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Mux1_Out1');
progMod_BasicMod_QAM16_QAM16_Reinterpret1 = xBlock(struct('source', 'Reinterpret'
     , 'name', 'progMod_BasicMod_QAM16_QAM16_Reinterpret1'), ...
struct('force_arith_type', 'on', ...
'force_bin_pt', 'on'), ...
{progMod_BasicMod_QAM16_QAM16_Mux1_Out1}, ...
{progMod_BasicMod_QAM16_QAM16_Reinterpret1_Out1});

progMod_BasicMod_QAM16_QAM16_b2b3_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_b2b3_Out1');
progMod_BasicMod_QAM16_QAM16_Constant4_Out1 = xSignal('
     progMod_BasicMod_QAM16_QAM16_Constant4_Out1');
progMod_BasicMod_QAM16_QAM16_Mux1 = xBlock(struct('source', 'Mux', 'name', '
     progMod_BasicMod_QAM16_QAM16_Mux1'), ...
struct('inputs', '4', ...
'arith_type', 'Signed␣␣(2''s␣comp)', ...
'bin_pt', 2, ...
'n_bits', 8), ...
{progMod_BasicMod_QAM16_QAM16_b2b3_Out1,
     progMod_BasicMod_QAM16_QAM16_Constant1_Out1,
            progMod_BasicMod_QAM16_QAM16_Constant2_Out1 ,
            progMod_BasicMod_QAM16_QAM16_Constant4_Out1 ,
            progMod_BasicMod_QAM16_QAM16_Constant3_Out1 }, ...
        {progMod_BasicMod_QAM16_QAM16_Mux1_Out1 });

        progMod_BasicMod_QAM16_QAM16_Mux4 = xBlock(struct('source', 'Mux', 'name', '
            progMod_BasicMod_QAM16_QAM16_Mux4'), ...
        struct('inputs', '4', ...
        'arith_type', 'Signed␣␣(2''s␣comp)', ...
        'bin_pt', 2, ...
        'n_bits', 8), ...
        {progMod_BasicMod_QAM16_QAM16_b0b1_Out1 ,
            progMod_BasicMod_QAM16_QAM16_Constant1_Out1 ,
            progMod_BasicMod_QAM16_QAM16_Constant2_Out1 ,
            progMod_BasicMod_QAM16_QAM16_Constant4_Out1 ,
            progMod_BasicMod_QAM16_QAM16_Constant3_Out1 }, ...
        {progMod_BasicMod_QAM16_QAM16_Mux4_Out1 });

        progMod_BasicMod_QAM16_QAM16_b2b3 = xBlock(struct('source', 'Slice', 'name', '
            progMod_BasicMod_QAM16_QAM16_b2b3'), ...
        struct('nbits', 2, ...
        'mode', 'Lower␣Bit␣Location␣+␣Width'), ...
        {progMod_BasicMod_QAM16_QAM16_In1 }, ...
        {progMod_BasicMod_QAM16_QAM16_b2b3_Out1 });

        progMod_BasicMod_QAM16_QAM16_Constant4_In1 = xSignal('
            progMod_BasicMod_QAM16_QAM16_Constant4_In1');
        progMod_BasicMod_QAM16_QAM16_Constant4 = xBlock(struct('source', 'Constant', '
            name', 'progMod_BasicMod_QAM16_QAM16_Constant4'), ...
        struct('bin_pt', 15, ...
        'const', 3*(1/sqrt(10))*(1/sqrt(13/6)), ...
        'period', T_xcvr, ...
        'explicit_period', 'on'), ...
        {progMod_BasicMod_QAM16_QAM16_Constant4_In1 }, ...
        {progMod_BasicMod_QAM16_QAM16_Constant4_Out1 });

    end
end
```

```

function progMod_BasicMod_BPSK()

    progMod_BasicMod_BPSK_In1 = xInport('progMod_BasicMod_BPSK_In1');
    progMod_BasicMod_BPSK_Out1 = xOutport('progMod_BasicMod_BPSK_Out1');

    progMod_BasicMod_BPSK_Mux4_Out1 = xSignal('progMod_BasicMod_BPSK_Mux4_Out1');
    progMod_BasicMod_BPSK_Reinterpret_Out1 = xSignal('
        progMod_BasicMod_BPSK_Reinterpret_Out1');
    progMod_BasicMod_BPSK_Reinterpret = xBlock(struct('source', 'Reinterpret', 'name', '
        progMod_BasicMod_BPSK_Reinterpret'), ...
    struct('force_arith_type', 'on', ...
    'force_bin_pt', 'on'), ...
    {progMod_BasicMod_BPSK_Mux4_Out1}, ...
    {progMod_BasicMod_BPSK_Reinterpret_Out1});

    progMod_BasicMod_BPSK_Constant3_In1 = xSignal('progMod_BasicMod_BPSK_Constant3_In1');
    progMod_BasicMod_BPSK_Constant3_Out1 = xSignal('progMod_BasicMod_BPSK_Constant3_Out1'
        );
    progMod_BasicMod_BPSK_Constant3 = xBlock(struct('source', 'Constant', 'name', '
        progMod_BasicMod_BPSK_Constant3'), ...
    struct('bin_pt', 15, ...
    'const', -1*(1/sqrt(13/6)), ...
    'period', T_xcvr, ...
    'explicit_period', 'on'), ...
    {progMod_BasicMod_BPSK_Constant3_In1}, ...
    {progMod_BasicMod_BPSK_Constant3_Out1});

    progMod_BasicMod_BPSK_Constant4_In1 = xSignal('progMod_BasicMod_BPSK_Constant4_In1');
    progMod_BasicMod_BPSK_Constant4_Out1 = xSignal('progMod_BasicMod_BPSK_Constant4_Out1'
        );
    progMod_BasicMod_BPSK_Constant4 = xBlock(struct('source', 'Constant', 'name', '
        progMod_BasicMod_BPSK_Constant4'), ...
    struct('bin_pt', 15, ...
    'period', T_xcvr, ...
    'explicit_period', 'on'), ...
    {progMod_BasicMod_BPSK_Constant4_In1}, ...
    {progMod_BasicMod_BPSK_Constant4_Out1});
```

```

progMod_BasicMod_BPSK_Constant2_In1 = xSignal('progMod_BasicMod_BPSK_Constant2_In1');
progMod_BasicMod_BPSK_Constant2_Out1 = xSignal('progMod_BasicMod_BPSK_Constant2_Out1'
    );
progMod_BasicMod_BPSK_Constant2 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_BPSK_Constant2'), ...
struct('bin_pt', 15, ...
'const', 1*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_BPSK_Constant2_In1}, ...
{progMod_BasicMod_BPSK_Constant2_Out1});

progMod_BasicMod_BPSK_Mux1_Out1 = xSignal('progMod_BasicMod_BPSK_Mux1_Out1');
progMod_BasicMod_BPSK_Reinterpret1_Out1 = xSignal('
    progMod_BasicMod_BPSK_Reinterpret1_Out1');
progMod_BasicMod_BPSK_Reinterpret1 = xBlock(struct('source', 'Reinterpret', 'name', '
    progMod_BasicMod_BPSK_Reinterpret1'), ...
struct('force_arith_type', 'on', ...
'force_bin_pt', 'on'), ...
{progMod_BasicMod_BPSK_Mux1_Out1}, ...
{progMod_BasicMod_BPSK_Reinterpret1_Out1});

progMod_BasicMod_BPSK_Constant1_In1 = xSignal('progMod_BasicMod_BPSK_Constant1_In1');
progMod_BasicMod_BPSK_Constant1_Out1 = xSignal('progMod_BasicMod_BPSK_Constant1_Out1'
    );
progMod_BasicMod_BPSK_Constant1 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_BPSK_Constant1'), ...
struct('bin_pt', 0, ...
'arith_type', 'Unsigned', ...
'const', 0, ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_BPSK_Constant1_In1}, ...
{progMod_BasicMod_BPSK_Constant1_Out1});

progMod_BasicMod_BPSK_Concat1_Out1 = xSignal('progMod_BasicMod_BPSK_Concat1_Out1');
```

```
progMod_BasicMod_BPSK_Concat1 = xBlock(struct('source', 'Concat', 'name', 'progMod_BasicMod_BPSK_Concat1'), ...
[], ...
{progMod_BasicMod_BPSK_Reinterpret1_Out1, progMod_BasicMod_BPSK_Reinterpret1_Out1}, ...
{progMod_BasicMod_BPSK_Concat1_Out1});

progMod_BasicMod_BPSK_Concat_Out1 = xSignal('progMod_BasicMod_BPSK_Concat_Out1');
progMod_BasicMod_BPSK_Concat = xBlock(struct('source', 'Concat', 'name', 'progMod_BasicMod_BPSK_Concat'), ...
[], ...
{progMod_BasicMod_BPSK_Reinterpret_Out1, progMod_BasicMod_BPSK_Constant1_Out1}, ...
{progMod_BasicMod_BPSK_Concat_Out1});

progMod_BasicMod_BPSK_b1_Out1 = xSignal('progMod_BasicMod_BPSK_b1_Out1');
progMod_BasicMod_BPSK_b1 = xBlock(struct('source', 'Slice', 'name', 'progMod_BasicMod_BPSK_b1'), ...
struct('mode', 'Lower␣Bit␣Location␣+␣Width', ...
'bit0', 1), ...
{progMod_BasicMod_BPSK_In1}, ...
{progMod_BasicMod_BPSK_b1_Out1});

progMod_BasicMod_BPSK_b0_Out1 = xSignal('progMod_BasicMod_BPSK_b0_Out1');
progMod_BasicMod_BPSK_b0 = xBlock(struct('source', 'Slice', 'name', 'progMod_BasicMod_BPSK_b0'), ...
struct('mode', 'Lower␣Bit␣Location␣+␣Width'), ...
{progMod_BasicMod_BPSK_In1}, ...
{progMod_BasicMod_BPSK_b0_Out1});

progMod_BasicMod_BPSK_Mux2 = xBlock(struct('source', 'Mux', 'name', 'progMod_BasicMod_BPSK_Mux2'), ...
struct('arith_type', 'Signed␣␣(2''s␣comp)', ...
'bin_pt', 2, ...
'n_bits', 8), ...
{progMod_BasicMod_BPSK_b1_Out1, progMod_BasicMod_BPSK_Concat_Out1, progMod_BasicMod_BPSK_Concat1_Out1}, ...
{progMod_BasicMod_BPSK_Out1});

```

```
        progMod_BasicMod_BPSK_Constant5_Out1 = xSignal('progMod_BasicMod_BPSK_Constant5_Out1'
            );
        progMod_BasicMod_BPSK_Mux1 = xBlock(struct('source', 'Mux', 'name', '
            progMod_BasicMod_BPSK_Mux1'), ...
        struct('arith_type', 'Signed␣␣(2''s␣comp)', ...
        'bin_pt', 2, ...
        'n_bits', 8), ...
        {progMod_BasicMod_BPSK_b0_Out1, progMod_BasicMod_BPSK_Constant5_Out1,
            progMod_BasicMod_BPSK_Constant4_Out1}, ...
        {progMod_BasicMod_BPSK_Mux1_Out1});

        progMod_BasicMod_BPSK_Constant5_In1 = xSignal('progMod_BasicMod_BPSK_Constant5_In1');
        progMod_BasicMod_BPSK_Constant5 = xBlock(struct('source', 'Constant', 'name', '
            progMod_BasicMod_BPSK_Constant5'), ...
        struct('bin_pt', 15, ...
        'const', -1, ...
        'period', T_xcvr, ...
        'explicit_period', 'on'), ...
        {progMod_BasicMod_BPSK_Constant5_In1}, ...
        {progMod_BasicMod_BPSK_Constant5_Out1});

        progMod_BasicMod_BPSK_Mux4 = xBlock(struct('source', 'Mux', 'name', '
            progMod_BasicMod_BPSK_Mux4'), ...
        struct('arith_type', 'Signed␣␣(2''s␣comp)', ...
        'bin_pt', 2, ...
        'n_bits', 8), ...
        {progMod_BasicMod_BPSK_b0_Out1, progMod_BasicMod_BPSK_Constant3_Out1,
            progMod_BasicMod_BPSK_Constant2_Out1}, ...
        {progMod_BasicMod_BPSK_Mux4_Out1});

    end

    function progMod_BasicMod_QAM64()

        progMod_BasicMod_QAM64_In1 = xInport('progMod_BasicMod_QAM64_In1');
        progMod_BasicMod_QAM64_Out1 = xOutport('progMod_BasicMod_QAM64_Out1');

```

```
progMod_BasicMod_QAM64_Constant2_In1 = xSignal('progMod_BasicMod_QAM64_Constant2_In1'
    );
progMod_BasicMod_QAM64_Constant2_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant2_Out1');
progMod_BasicMod_QAM64_Constant2 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant2'), ...
struct('bin_pt', 15, ...
'const', -5*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant2_In1}, ...
{progMod_BasicMod_QAM64_Constant2_Out1});

progMod_BasicMod_QAM64_Mux4_Out1 = xSignal('progMod_BasicMod_QAM64_Mux4_Out1');
progMod_BasicMod_QAM64_Reinterpret1_Out1 = xSignal('
    progMod_BasicMod_QAM64_Reinterpret1_Out1');
progMod_BasicMod_QAM64_Reinterpret1 = xBlock(struct('source', 'Reinterpret', 'name',
    'progMod_BasicMod_QAM64_Reinterpret1'), ...
struct('force_arith_type', 'on', ...
'force_bin_pt', 'on'), ...
{progMod_BasicMod_QAM64_Mux4_Out1}, ...
{progMod_BasicMod_QAM64_Reinterpret1_Out1});

progMod_BasicMod_QAM64_b0b1b2_Out1 = xSignal('progMod_BasicMod_QAM64_b0b1b2_Out1');
progMod_BasicMod_QAM64_Constant1_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant1_Out1');
progMod_BasicMod_QAM64_Constant3_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant3_Out1');
progMod_BasicMod_QAM64_Constant4_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant4_Out1');
progMod_BasicMod_QAM64_Constant5_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant5_Out1');
progMod_BasicMod_QAM64_Constant6_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant6_Out1');
progMod_BasicMod_QAM64_Constant7_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant7_Out1');
progMod_BasicMod_QAM64_Constant8_Out1 = xSignal('
    progMod_BasicMod_QAM64_Constant8_Out1');
```

```
progMod_BasicMod_QAM64_Mux4 = xBlock(struct('source', 'Mux', 'name', '
    progMod_BasicMod_QAM64_Mux4'), ...
struct('inputs', '8', ...
'arith_type', 'Signed␣␣(2''s␣comp)', ...
'bin_pt', 2, ...
'n_bits', 8), ...
{progMod_BasicMod_QAM64_b0b1b2_Out1, progMod_BasicMod_QAM64_Constant1_Out1,
    progMod_BasicMod_QAM64_Constant2_Out1, progMod_BasicMod_QAM64_Constant3_Out1,
    progMod_BasicMod_QAM64_Constant4_Out1, progMod_BasicMod_QAM64_Constant5_Out1,
    progMod_BasicMod_QAM64_Constant6_Out1, progMod_BasicMod_QAM64_Constant7_Out1,
    progMod_BasicMod_QAM64_Constant8_Out1}, ...
{progMod_BasicMod_QAM64_Mux4_Out1});

progMod_BasicMod_QAM64_Constant4_In1 = xSignal('progMod_BasicMod_QAM64_Constant4_In1'
    );
progMod_BasicMod_QAM64_Constant4 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant4'), ...
struct('bin_pt', 15, ...
'const', -3*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant4_In1}, ...
{progMod_BasicMod_QAM64_Constant4_Out1});

progMod_BasicMod_QAM64_Constant5_In1 = xSignal('progMod_BasicMod_QAM64_Constant5_In1'
    );
progMod_BasicMod_QAM64_Constant5 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant5'), ...
struct('bin_pt', 15, ...
'const', 7*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant5_In1}, ...
{progMod_BasicMod_QAM64_Constant5_Out1});

progMod_BasicMod_QAM64_Constant6_In1 = xSignal('progMod_BasicMod_QAM64_Constant6_In1'
    );
```

```
progMod_BasicMod_QAM64_Constant6 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant6'), ...
struct('bin_pt', 15, ...
'const', 5*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant6_In1}, ...
{progMod_BasicMod_QAM64_Constant6_Out1});

progMod_BasicMod_QAM64_Constant7_In1 = xSignal('progMod_BasicMod_QAM64_Constant7_In1'
    );
progMod_BasicMod_QAM64_Constant7 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant7'), ...
struct('bin_pt', 15, ...
'const', 1*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant7_In1}, ...
{progMod_BasicMod_QAM64_Constant7_Out1});

progMod_BasicMod_QAM64_Constant3_In1 = xSignal('progMod_BasicMod_QAM64_Constant3_In1'
    );
progMod_BasicMod_QAM64_Constant3 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant3'), ...
struct('bin_pt', 15, ...
'const', -1*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant3_In1}, ...
{progMod_BasicMod_QAM64_Constant3_Out1});

progMod_BasicMod_QAM64_Constant8_In1 = xSignal('progMod_BasicMod_QAM64_Constant8_In1'
    );
progMod_BasicMod_QAM64_Constant8 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant8'), ...
struct('bin_pt', 15, ...
'const', 3*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant8_In1}, ...
{progMod_BasicMod_QAM64_Constant8_Out1});

progMod_BasicMod_QAM64_Constant1_In1 = xSignal('progMod_BasicMod_QAM64_Constant1_In1'
    );
progMod_BasicMod_QAM64_Constant1 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QAM64_Constant1'), ...
struct('bin_pt', 15, ...
'const', -7*(1/sqrt(42))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QAM64_Constant1_In1}, ...
{progMod_BasicMod_QAM64_Constant1_Out1});

progMod_BasicMod_QAM64_b3b4b5_Out1 = xSignal('progMod_BasicMod_QAM64_b3b4b5_Out1');
progMod_BasicMod_QAM64_Mux1_Out1 = xSignal('progMod_BasicMod_QAM64_Mux1_Out1');
progMod_BasicMod_QAM64_Mux1 = xBlock(struct('source', 'Mux', 'name', '
    progMod_BasicMod_QAM64_Mux1'), ...
struct('inputs', '8', ...
'arith_type', 'Signed␣␣(2''s␣comp)', ...
'bin_pt', 2, ...
'n_bits', 8), ...
{progMod_BasicMod_QAM64_b3b4b5_Out1, progMod_BasicMod_QAM64_Constant1_Out1,
    progMod_BasicMod_QAM64_Constant2_Out1, progMod_BasicMod_QAM64_Constant3_Out1,
    progMod_BasicMod_QAM64_Constant4_Out1, progMod_BasicMod_QAM64_Constant5_Out1,
    progMod_BasicMod_QAM64_Constant6_Out1, progMod_BasicMod_QAM64_Constant7_Out1,
    progMod_BasicMod_QAM64_Constant8_Out1}, ...
{progMod_BasicMod_QAM64_Mux1_Out1});

progMod_BasicMod_QAM64_Reinterpret_Out1 = xSignal('
    progMod_BasicMod_QAM64_Reinterpret_Out1');
progMod_BasicMod_QAM64_Concat = xBlock(struct('source', 'Concat', 'name', '
    progMod_BasicMod_QAM64_Concat'), ...
[], ...
{progMod_BasicMod_QAM64_Reinterpret1_Out1, progMod_BasicMod_QAM64_Reinterpret_Out1},
    ...
{progMod_BasicMod_QAM64_Out1});
```

```

        progMod_BasicMod_QAM64_b3b4b5 = xBlock(struct('source', 'Slice', 'name', '
            progMod_BasicMod_QAM64_b3b4b5'), ...
        struct('nbits', 3, ...
        'mode', 'Lower_Bit_Location_+_Width'), ...
        {progMod_BasicMod_QAM64_In1}, ...
        {progMod_BasicMod_QAM64_b3b4b5_Out1});

        progMod_BasicMod_QAM64_Reinterpret = xBlock(struct('source', 'Reinterpret', 'name', '
            progMod_BasicMod_QAM64_Reinterpret'), ...
        struct('force_arith_type', 'on', ...
        'force_bin_pt', 'on'), ...
        {progMod_BasicMod_QAM64_Mux1_Out1}, ...
        {progMod_BasicMod_QAM64_Reinterpret_Out1});

        progMod_BasicMod_QAM64_b0b1b2 = xBlock(struct('source', 'Slice', 'name', '
            progMod_BasicMod_QAM64_b0b1b2'), ...
        struct('nbits', 3, ...
        'mode', 'Lower_Bit_Location_+_Width', ...
        'bit0', 3), ...
        {progMod_BasicMod_QAM64_In1}, ...
        {progMod_BasicMod_QAM64_b0b1b2_Out1});

    end

    function progMod_BasicMod_QPSK()

        progMod_BasicMod_QPSK_In1 = xInport('progMod_BasicMod_QPSK_In1');
        progMod_BasicMod_QPSK_Out1 = xOutport('progMod_BasicMod_QPSK_Out1');

        progMod_BasicMod_QPSK_b0_Out1 = xSignal('progMod_BasicMod_QPSK_b0_Out1');
        progMod_BasicMod_QPSK_Constant1_Out1 = xSignal('progMod_BasicMod_QPSK_Constant1_Out1'
            );
        progMod_BasicMod_QPSK_Constant2_Out1 = xSignal('progMod_BasicMod_QPSK_Constant2_Out1'
            );
        progMod_BasicMod_QPSK_Mux4_Out1 = xSignal('progMod_BasicMod_QPSK_Mux4_Out1');
        progMod_BasicMod_QPSK_Mux4 = xBlock(struct('source', 'Mux', 'name', '
            progMod_BasicMod_QPSK_Mux4'), ...
```

```
struct('arith_type', 'Signed␣␣(2''s␣comp)', ...
'bin_pt', 2, ...
'n_bits', 8), ...
{progMod_BasicMod_QPSK_b0_Out1, progMod_BasicMod_QPSK_Constant1_Out1,
    progMod_BasicMod_QPSK_Constant2_Out1}, ...
{progMod_BasicMod_QPSK_Mux4_Out1});

progMod_BasicMod_QPSK_b1_Out1 = xSignal('progMod_BasicMod_QPSK_b1_Out1');
progMod_BasicMod_QPSK_Mux1_Out1 = xSignal('progMod_BasicMod_QPSK_Mux1_Out1');
progMod_BasicMod_QPSK_Mux1 = xBlock(struct('source', 'Mux', 'name', '
    progMod_BasicMod_QPSK_Mux1'), ...
struct('arith_type', 'Signed␣␣(2''s␣comp)', ...
'bin_pt', 2, ...
'n_bits', 8), ...
{progMod_BasicMod_QPSK_b1_Out1, progMod_BasicMod_QPSK_Constant1_Out1,
    progMod_BasicMod_QPSK_Constant2_Out1}, ...
{progMod_BasicMod_QPSK_Mux1_Out1});

progMod_BasicMod_QPSK_b1 = xBlock(struct('source', 'Slice', 'name', '
    progMod_BasicMod_QPSK_b1'), ...
struct('mode', 'Lower␣Bit␣Location␣+␣Width'), ...
{progMod_BasicMod_QPSK_In1}, ...
{progMod_BasicMod_QPSK_b1_Out1});

progMod_BasicMod_QPSK_Constant2_In1 = xSignal('progMod_BasicMod_QPSK_Constant2_In1');
progMod_BasicMod_QPSK_Constant2 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QPSK_Constant2'), ...
struct('bin_pt', 15, ...
'const', (1/sqrt(2))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QPSK_Constant2_In1}, ...
{progMod_BasicMod_QPSK_Constant2_Out1});

progMod_BasicMod_QPSK_Constant1_In1 = xSignal('progMod_BasicMod_QPSK_Constant1_In1');
progMod_BasicMod_QPSK_Constant1 = xBlock(struct('source', 'Constant', 'name', '
    progMod_BasicMod_QPSK_Constant1'), ...
struct('bin_pt', 15, ...
'const', -(1/sqrt(2))*(1/sqrt(13/6)), ...
'period', T_xcvr, ...
'explicit_period', 'on'), ...
{progMod_BasicMod_QPSK_Constant1_In1}, ...
{progMod_BasicMod_QPSK_Constant1_Out1});

progMod_BasicMod_QPSK_Reinterpret_Out1 = xSignal('
    progMod_BasicMod_QPSK_Reinterpret_Out1');
progMod_BasicMod_QPSK_Reinterpret = xBlock(struct('source', 'Reinterpret', 'name', '
    progMod_BasicMod_QPSK_Reinterpret'), ...
struct('force_arith_type', 'on', ...
'force_bin_pt', 'on'), ...
{progMod_BasicMod_QPSK_Mux4_Out1}, ...
{progMod_BasicMod_QPSK_Reinterpret_Out1});

progMod_BasicMod_QPSK_b0 = xBlock(struct('source', 'Slice', 'name', '
    progMod_BasicMod_QPSK_b0'), ...
struct('mode', 'Lower␣Bit␣Location␣+␣Width', ...
'bit0', 1), ...
{progMod_BasicMod_QPSK_In1}, ...
{progMod_BasicMod_QPSK_b0_Out1});

progMod_BasicMod_QPSK_Reinterpret1_Out1 = xSignal('
    progMod_BasicMod_QPSK_Reinterpret1_Out1');
progMod_BasicMod_QPSK_Reinterpret1 = xBlock(struct('source', 'Reinterpret', 'name', '
    progMod_BasicMod_QPSK_Reinterpret1'), ...
struct('force_arith_type', 'on', ...
'force_bin_pt', 'on'), ...
{progMod_BasicMod_QPSK_Mux1_Out1}, ...
{progMod_BasicMod_QPSK_Reinterpret1_Out1});

progMod_BasicMod_QPSK_Concat = xBlock(struct('source', 'Concat', 'name', '
    progMod_BasicMod_QPSK_Concat'), ...
[], ...
{progMod_BasicMod_QPSK_Reinterpret_Out1, progMod_BasicMod_QPSK_Reinterpret1_Out1},
    ...
{progMod_BasicMod_QPSK_Out1});

end
end
end
```

Appendix C

Autogenerated C Code for software implemention of the *ProgMod* subsystem

Listing C.1: C code implementation of the programmable modulator

```
void progMod(float* progMod_data_ptr, float* progMod_modOnOff_ptr, float* progMod_modType_ptr,
    float* progMod_outReal, float* progMod_outImag)
{
    int tempInt1, tempInt2, tempInt3;
    float* muxInputs;


    // BasicMod
    float progMod_BasicMod_block2_Out1;
    tempInt1 = (int)*progMod_data_ptr;
    slice(tempInt1, 16, CHOOSE_LSB, 7, 0, &tempInt2);
    //printf("progMod_data_ptr = %d, slicedOut  = %d\n", tempInt1, tempInt2);
    progMod_BasicMod_block2_Out1 = (float)tempInt2;

    // BPSK
    float progMod_BasicMod_BPSK_b0_Out1;
    tempInt1 = (int)progMod_BasicMod_block2_Out1;
    slice(tempInt1, 16, CHOOSE_LSB, 1, 0, &tempInt2);
    progMod_BasicMod_BPSK_b0_Out1 = (float)tempInt2;

    float progMod_BasicMod_BPSK_b1_Out1;
    tempInt1 = (int)progMod_BasicMod_block2_Out1;
    slice(tempInt1, 16, CHOOSE_LSB, 1, 1, &tempInt2);
    progMod_BasicMod_BPSK_b1_Out1 = (float)tempInt2;

    float progMod_BasicMod_BPSK_Constant2_Out1 = 1*(1/sqrt(13.0/6.0));
    float progMod_BasicMod_BPSK_Constant3_Out1 = -1*(1/sqrt(13.0/6.0));

    float progMod_BasicMod_BPSK_Mux4_Out1;
    tempInt1 = (int)progMod_BasicMod_BPSK_b0_Out1;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = progMod_BasicMod_BPSK_Constant3_Out1;
    *(muxInputs+1) = progMod_BasicMod_BPSK_Constant2_Out1;
    mux(tempInt1, muxInputs, &progMod_BasicMod_BPSK_Mux4_Out1);
    free(muxInputs);

    float progMod_BasicMod_BPSK_Constant5_Out1 = -1;
    float progMod_BasicMod_BPSK_Constant4_Out1 = 1;

    float progMod_BasicMod_BPSK_Mux1_Out1;
    tempInt1 = (int)progMod_BasicMod_BPSK_Mux1_Out1;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = progMod_BasicMod_BPSK_Constant5_Out1;
    *(muxInputs+1) = progMod_BasicMod_BPSK_Constant4_Out1;
    mux(tempInt1, muxInputs, &progMod_BasicMod_BPSK_Mux1_Out1);
    free(muxInputs);

    float progMod_BasicMod_BPSK_Mux2_Out1;
    tempInt1 = (int)progMod_BasicMod_BPSK_b1_Out1;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = progMod_BasicMod_BPSK_Mux4_Out1;
    *(muxInputs+1) = progMod_BasicMod_BPSK_Mux1_Out1;
    mux(tempInt1, muxInputs, &progMod_BasicMod_BPSK_Mux2_Out1);
    free(muxInputs);

    float progMod_BasicMod_BPSK_Mux5_Out1;
    tempInt1 = (int)progMod_BasicMod_BPSK_b1_Out1;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = 0;
    *(muxInputs+1) = progMod_BasicMod_BPSK_Mux1_Out1;
    mux(tempInt1, muxInputs, &progMod_BasicMod_BPSK_Mux5_Out1);
    free(muxInputs);

    float progMod_BasicMod_BPSK_Out1 = progMod_BasicMod_BPSK_Mux2_Out1;
    float progMod_BasicMod_BPSK_Out2 = progMod_BasicMod_BPSK_Mux5_Out1;


    // QPSK
    float progMod_BasicMod_QPSK_b0_Out1;
    tempInt1 = (int)progMod_BasicMod_block2_Out1;
    slice(tempInt1, 16, CHOOSE_LSB, 1, 0, &tempInt2);
    progMod_BasicMod_QPSK_b0_Out1 = (float)tempInt2;

    float progMod_BasicMod_QPSK_b1_Out1;
    tempInt1 = (int)progMod_BasicMod_block2_Out1;
    slice(tempInt1, 16, CHOOSE_LSB, 1, 1, &tempInt2);
    progMod_BasicMod_QPSK_b1_Out1 = (float)tempInt2;

    float progMod_BasicMod_QPSK_Constant1_Out1 = -(1/sqrtf(2.0))*(1/sqrtf(13.0/6.0));
    float progMod_BasicMod_QPSK_Constant2_Out1 = (1/sqrtf(2.0))*(1/sqrtf(13.0/6.0));

    float progMod_BasicMod_QPSK_Mux4_Out1;
    tempInt1 = (int)progMod_BasicMod_QPSK_b0_Out1;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = progMod_BasicMod_QPSK_Constant1_Out1;
    *(muxInputs+1) = progMod_BasicMod_QPSK_Constant2_Out1;
    mux(tempInt1, muxInputs, &progMod_BasicMod_QPSK_Mux4_Out1);
    free(muxInputs);

    float progMod_BasicMod_QPSK_Mux1_Out1;
    tempInt1 = (int)progMod_BasicMod_QPSK_b1_Out1;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = progMod_BasicMod_QPSK_Constant1_Out1;
    *(muxInputs+1) = progMod_BasicMod_QPSK_Constant2_Out1;
    mux(tempInt1, muxInputs, &progMod_BasicMod_QPSK_Mux1_Out1);
    free(muxInputs);

    float progMod_BasicMod_QPSK_Out1 = progMod_BasicMod_QPSK_Mux4_Out1;
    float progMod_BasicMod_QPSK_Out2 = progMod_BasicMod_QPSK_Mux1_Out1;


```

```
// QAM16
float progMod_BasicMod_QAM16_QAM16_b0b1_Out1;
tempInt1 = (int)progMod_BasicMod_block2_Out1;
slice(tempInt1, 16, CHOOSE_LSB, 2, 2, &tempInt2);
progMod_BasicMod_QAM16_QAM16_b0b1_Out1 = (float)tempInt2;

float progMod_BasicMod_QAM16_QAM16_b2b3_Out1;
tempInt1 = (int)progMod_BasicMod_block2_Out1;
slice(tempInt1, 16, CHOOSE_LSB, 2, 0, &tempInt2);
progMod_BasicMod_QAM16_QAM16_b2b3_Out1 = (float)tempInt2;

float progMod_BasicMod_QAM16_QAM16_Constant1_Out1 = -3.0*(1.0/sqrtf(10.0))*(1.0/sqrtf
    (13.0/6.0));
float progMod_BasicMod_QAM16_QAM16_Constant2_Out1 = -1.0*(1.0/sqrtf(10.0))*(1.0/sqrtf
    (13.0/6.0));
float progMod_BasicMod_QAM16_QAM16_Constant3_Out1 = 1.0*(1.0/sqrtf(10.0))*(1.0/sqrtf
    (13.0/6.0));
float progMod_BasicMod_QAM16_QAM16_Constant4_Out1 = 3.0*(1.0/sqrtf(10.0))*(1.0/sqrtf
    (13.0/6.0));

float progMod_BasicMod_QAM16_QAM16_Mux1_Out1;
tempInt1 = (int)progMod_BasicMod_QAM16_QAM16_b2b3_Out1;
muxInputs = (float*)malloc(sizeof(float)*4);
*muxInputs = progMod_BasicMod_QAM16_QAM16_Constant1_Out1;
*(muxInputs+1) = progMod_BasicMod_QAM16_QAM16_Constant2_Out1;
*(muxInputs+2) = progMod_BasicMod_QAM16_QAM16_Constant4_Out1;
*(muxInputs+3) = progMod_BasicMod_QAM16_QAM16_Constant3_Out1;
mux(tempInt1, muxInputs, &progMod_BasicMod_QAM16_QAM16_Mux1_Out1);
free(muxInputs);

float progMod_BasicMod_QAM16_QAM16_Mux4;
tempInt1 = (int)progMod_BasicMod_QAM16_QAM16_b0b1_Out1;
muxInputs = (float*)malloc(sizeof(float)*4);
*muxInputs = progMod_BasicMod_QAM16_QAM16_Constant1_Out1;
*(muxInputs+1) = progMod_BasicMod_QAM16_QAM16_Constant2_Out1;
*(muxInputs+2) = progMod_BasicMod_QAM16_QAM16_Constant4_Out1;
*(muxInputs+3) = progMod_BasicMod_QAM16_QAM16_Constant3_Out1;
mux(tempInt1, muxInputs, &progMod_BasicMod_QAM16_QAM16_Mux4);
```

```
free(muxInputs);

float progMod_BasicMod_QAM16_Out1 = progMod_BasicMod_QAM16_QAM16_Mux4;
float progMod_BasicMod_QAM16_Out2 = progMod_BasicMod_QAM16_QAM16_Mux1_Out1;



// QAM64
float progMod_BasicMod_QAM64_b0b1b2_Out1;
tempInt1 = (int)progMod_BasicMod_block2_Out1;
slice(tempInt1, 16, CHOOSE_LSB, 3, 3, &tempInt2);
progMod_BasicMod_QAM64_b0b1b2_Out1 = (float)tempInt2;

float progMod_BasicMod_QAM64_b3b4b5_Out1;
tempInt1 = (int)progMod_BasicMod_block2_Out1;
slice(tempInt1, 16, CHOOSE_LSB, 3, 0, &tempInt2);
progMod_BasicMod_QAM64_b3b4b5_Out1 = (float)tempInt2;

float progMod_BasicMod_QAM64_Constant2_Out1 = -5.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));
float progMod_BasicMod_QAM64_Constant4_Out1 = -3.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));
float progMod_BasicMod_QAM64_Constant5_Out1 = 7.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));
float progMod_BasicMod_QAM64_Constant6_Out1 = 5.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));
float progMod_BasicMod_QAM64_Constant7_Out1 = 1.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));
float progMod_BasicMod_QAM64_Constant3_Out1 = -1.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));
float progMod_BasicMod_QAM64_Constant8_Out1 = 3.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));
float progMod_BasicMod_QAM64_Constant1_Out1 = -7.0*(1.0/sqrtf(42.0))*(1.0/sqrtf(13.0/6.0));

float progMod_BasicMod_QAM64_Mux1_Out1;
tempInt1 = (int)progMod_BasicMod_QAM64_b3b4b5_Out1;
muxInputs = (float*)malloc(sizeof(float)*8);
*muxInputs = progMod_BasicMod_QAM64_Constant1_Out1;
*(muxInputs+1) = progMod_BasicMod_QAM64_Constant2_Out1;
*(muxInputs+2) = progMod_BasicMod_QAM64_Constant3_Out1;
*(muxInputs+3) = progMod_BasicMod_QAM64_Constant4_Out1;
*(muxInputs+4) = progMod_BasicMod_QAM64_Constant5_Out1;
*(muxInputs+5) = progMod_BasicMod_QAM64_Constant6_Out1;
*(muxInputs+6) = progMod_BasicMod_QAM64_Constant7_Out1;
*(muxInputs+7) = progMod_BasicMod_QAM64_Constant8_Out1;
mux(tempInt1, muxInputs, &progMod_BasicMod_QAM64_Mux1_Out1);
```

```
        free(muxInputs);

        float progMod_BasicMod_QAM64_Mux4_Out1;
        tempInt1 = (int)progMod_BasicMod_QAM64_b0b1b2_Out1;
        muxInputs = (float*)malloc(sizeof(float)*8);
        *muxInputs = progMod_BasicMod_QAM64_Constant1_Out1;
        *(muxInputs+1) = progMod_BasicMod_QAM64_Constant2_Out1;
        *(muxInputs+2) = progMod_BasicMod_QAM64_Constant3_Out1;
        *(muxInputs+3) = progMod_BasicMod_QAM64_Constant4_Out1;
        *(muxInputs+4) = progMod_BasicMod_QAM64_Constant5_Out1;
        *(muxInputs+5) = progMod_BasicMod_QAM64_Constant6_Out1;
        *(muxInputs+6) = progMod_BasicMod_QAM64_Constant7_Out1;
        *(muxInputs+7) = progMod_BasicMod_QAM64_Constant8_Out1;
        mux(tempInt1, muxInputs, &progMod_BasicMod_QAM64_Mux4_Out1);
        free(muxInputs);

        float progMod_BasicMod_QAM64_Out1 = progMod_BasicMod_QAM64_Mux4_Out1;
        float progMod_BasicMod_QAM64_Out2 = progMod_BasicMod_QAM64_Mux1_Out1;


        // BasicMod
        float progMod_BasicMod_block1_Out1;
        tempInt1 = (int)*progMod_modType_ptr;
        slice(tempInt1, 16, CHOOSE_LSB, 2, 0, &tempInt2);
        //printf("progMod_data_ptr = %d, slicedOut = %d\n", tempInt1, tempInt2);
        progMod_BasicMod_block1_Out1 = (float)tempInt2;

        float progMod_BasicMod_select_modulation_I_Out1;
        tempInt1 = (int)progMod_BasicMod_block1_Out1;
        muxInputs = (float*)malloc(sizeof(float)*4);
        *muxInputs = progMod_BasicMod_QAM64_Out1;
        *(muxInputs+1) = progMod_BasicMod_QAM16_Out1;
        *(muxInputs+2) = progMod_BasicMod_QPSK_Out1;
        *(muxInputs+3) = progMod_BasicMod_BPSK_Out1;
        mux(tempInt1, muxInputs, &progMod_BasicMod_select_modulation_I_Out1);
        free(muxInputs);

        float progMod_BasicMod_select_modulation_Q_Out1;
```

```
    tempInt1 = (int)progMod_BasicMod_block1_Out1;
    muxInputs = (float*)malloc(sizeof(float)*4);
    *muxInputs = progMod_BasicMod_QAM64_Out2;
    *(muxInputs+1) = progMod_BasicMod_QAM16_Out2;
    *(muxInputs+2) = progMod_BasicMod_QPSK_Out2;
    *(muxInputs+3) = progMod_BasicMod_BPSK_Out2;
    mux(tempInt1, muxInputs, &progMod_BasicMod_select_modulation_Q_Out1);
    free(muxInputs);

    float progMod_BasicMod_Out1 = progMod_BasicMod_select_modulation_I_Out1;
    float progMod_BasicMod_Out2 = progMod_BasicMod_select_modulation_Q_Out1;


    // progMod
    float progMod_null_Out1 = 0;

    float progMod_mux_I_Out1;
    tempInt1 = (int)*progMod_modOnOff_ptr;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = progMod_null_Out1;
    *(muxInputs+1) = progMod_BasicMod_Out1;
    mux(tempInt1, muxInputs, &progMod_mux_I_Out1);
    free(muxInputs);

    float progMod_mux_Q_Out1;
    tempInt1 = (int)*progMod_modOnOff_ptr;
    muxInputs = (float*)malloc(sizeof(float)*2);
    *muxInputs = progMod_null_Out1;
    *(muxInputs+1) = progMod_BasicMod_Out2;
    mux(tempInt1, muxInputs, &progMod_mux_Q_Out1);
    free(muxInputs);

    *progMod_outReal = progMod_mux_I_Out1;
    *progMod_outImag = progMod_mux_Q_Out1;

}
```