PERFORMANCE ANALYSIS OF THE WINC2R PLATFORM

BY SUMIT SATARKAR

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Prof. Wade Trappe

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2009

ABSTRACT OF THE THESIS

Performance Analysis of the WiNC2R Platform

by Sumit Satarkar

Thesis Director: Prof. Wade Trappe

A Cognitive Radio (CR) is an intelligent transceiver device, able to support multiple technologies, dynamic re-configurability, ease of programming and collaboration with other CR devices to improve the communication efficiency. The two key requirements for an efficient CR implementation are flexibility in operation/programming and speed.

WiNC2R (Winlab Network Centric Cognitive Radio) achieves high speed of operation using its hardware platform and flexibility using its software-configurable architecture. The current WiNC2R architecture implements an 802.11a-like OFDM flow. We evaluate the WiNC2R hardware architecture to see the modularity in the architecture, separation of data and control flow and the performance in terms of latency and throughput. To test the system, the Xilinx Bus Functional Model environment, which is designed to test the IBM standard bus-architecture-based hardware systems, is used. We use a simple ALOHA protocol in the MAC layer to communicate between two WiNC2R nodes and evaluate the performance under the best-case scenario, where the performance is only hindered by the architecture itself rather than external conditions like channel state.

The results of our basic experiments showed that for a single OFDM 802.11a-like flow, the Unit Control Modules (UCM) were idle for almost 80% of the total processing time.

We then tested the WiNC2R system to study the effects of changing the frame size. It was seen that the latencies in the WiNC2R transmitter are frame-size dependent while those in the receiver mainly depend on the size of the data in the last chunk rather than the size of the whole frame. We suggest that chunk size should be 2 OFDM symbols, and chunking be moved to MAC layer for better performance. We give analytical estimates of resulting performance improvement. In the next experiment, we describe virtualization in the WiNC2R by adding more flows. We describe the steps to implement the additional flows and estimate maximum number of concurrent flows possible.

In the last analysis, we show the effect of operating clock frequency on the performance. We prove that at 250 MHz operating frequency and 2 OFDM symbols per chunk, the current WiNC2R implementation will be able to satisfy the SIFS criterion.

Acknowledgements

First and foremost, I would like to thank my adviser Prof. Wade Trappe for his constant guidance and support throughout the entire duration of my MS research. He gave me enough freedom to work in the field of my interest while making sure that I was always on the right path. I would also like to thank Prof. Zoran Miljanic for his vision and guidance during the WiNC2R design phase.

This work would not have been possible without Ivan Seskar, Khanh Le and Renu Rajnarayan. Their constant support and enormous patience to deal with many of my doubts and questions have made this work conceptually sound. It was a great experience working with them and because of them, I can confidently say that working on WiNC2R project has been a very good learning experience .

I have met many fantastic people on my way to a graduate degree. My time in Winlab was made memorable by my friends and peers in the Cognitive Radio team Tejaswy, Madhura, Vaidehi, Shalini, Akshay, Prasanthi, Mohit and Onkar. With them there was never a dull moment while working in the lab. Since the beginning of the MS study, I have been fortunate enough to have what I hope to be long-lasting friendships with Amrita, Tanuja and Ayesha.

Last here but foremost in my mind, I would like to thank my parents and my family for their continuous support throughout this journey. I would like to dedicate this work to them.

Table of Contents

Abstract ii

Acknowledgements iv

List of Tables vii

List of Figures viii

1. Introduction 1

2. Survey of Cognitive Radio Platforms 7

2.1. CR Platforms 7

2.1.1. Classification of CR Platforms 7

2.1.2. GNU Radio Platform 9

2.1.3. WARP Platform 11

3. WiNC2R Architecture 13

3.1. WiNC2R Architecture 13

3.1.1. Top Level Modular Architecture 13

3.1.2. Re-configurability in WiNC2R 14

3.1.3. Current WiNC2R Implementation 18

3.1.4. Plug-and-play and Re-configurability Inside PE 20

3.1.5. Architectural Comparison of GNU Radio and WiNC2R 21

4. Testing and Simulations 23

4.1. WiNC2R Test-Bench 23

4.1.1. A Generic Test-Bench to Test Hardware Systems 23

4.1.2. Need for an Automated Test Bench 24

4.1.3. BFM Testing Environment 26
4.1.4. WiNC2R Test-Bench using BFM 28
4.2. Basic Analytical Experiment 30
4.2.1. The Experimental Setup 30
4.2.2. Measurements and Analysis: Data Frame Transfer 32
4.2.2.1. FU MAC Tx Analysis 32
4.2.2.2. FU HDR Tx Analysis 34
4.2.2.3. FU MOD Tx Analysis 38
4.2.2.4. FU IFFT Tx Analysis 39
4.2.2.5. FU SYNC Rx Analysis 39
4.2.2.6. FU MAC Rx Analysis 43
4.2.3. ACK Frame Transfer 44
4.2.4. Summary 44
4.3. Effect of Frame Size on WiNC2R Performance 48
4.3.1. Experimental Setup and Measurements 48
4.3.2. Analysis of TAT and RTT 49
4.3.2.1. TAT Analysis 51
4.3.2.2. TAT Improvement 54
4.3.2.3. RTT Analysis 58
4.3.2.4. Improvement in T_{DATA_Tx} 59
4.4. Supporting Multiple Flows by Time-Sharing of UCM 63
4.4.1. Motivation and Experimental Setup 63
4.4.2. Measurements and Analysis 64
4.5. Effect of Change in Clock Frequency 67

5. Conclusions and Future Work 69

References 72

List of Tables

4.1. Truth Table for Two-Bit-Input Adder 23

4.2. FU MAC Tx Measurements: Fixed Frame Size 33

4.3. FU Sync Rx Preamble Processing Measurements 43

4.4. FU MAC Rx Measurements: Fixed Frame Size 44

4.5. FU Measurements: Fixed Frame Size 45

4.6. Chunk Sizes with 4 Symbols per chunk 53

4.7. T_{MAC_Rx} Measurements: Varying Frame Size 54

4.8. Last Chunk Size Table 56

List of Figures

2.1. The Data and Control Flow in GNU Radio 11

2.2. The Architecture of WARP Platform 12

3.1. Top Level Block Diagram of WiNC2R 13

3.2. Components of the FU: PE and UCM 15

3.3. The Two Versions of TD Table for FU1 16

3.4. The Effect of the Change in TD Table on the Data Flow 17

3.5. FUs Currently Implemented in WiNC2R 18

3.6. The Effect of Chunking on the Frame 19

3.7. The Summary of GNU Radio and WiNC2R Architecture 21

4.1. A Generic Test Bench for Two-bit Input Adder 24

4.2. A Generic Test Bench using BFM 26

4.3. A Sample BFL Script 27

4.4. Interaction between BFL Script and VHDL test-bench 27

4.5. Block Diagram of a BFM Node 28

4.6. Communication between two WiNC2R nodes in BFM simulation 30

4.7. Data Transfer using simple ALOHA MAC Protocol 31

4.8. Data Frame Format 31

4.9. Ack Frame Format 31

4.10. Task Flow for Data Frame Transfer 32

4.11. Components of FU Latencies 32

4.12. PLCP Header Format 35

4.13. Effect of Chunking and Reschedule Period in PE HDR 36

4.14. Processing Latencies in FU HDR: Fixed Frame Size 37

4.15. Comparison of T_{FU_HDR} and T_{IDLE}: Fixed Frame Size 37

4.16. Processing Latencies in FU MOD: Fixed Frame Size 38
4.17. Processing Latencies in FU IFFT: Fixed Frame Size 40
4.18. The Task Flow inside the FU SYNC block 41
4.19. Processing Latencies in FU Sync Rx: Fixed Frame Size 43
4.20. Task Flow for ACK Frame 45
4.21. UCM Latencies per chunk: Fixed Frame Size 46
4.22. Control and Data Overhead in T_{TT} 46
4.23. Data Transfer using simple ALOHA MAC Protocol 48
4.24. RTT Measurement: Varying Frame Size 49
4.25. Throughput Measurement: Varying Frame Size 50
4.26. Measurement of TAT and RTT at GPIO Interfaces of Nodes 50
4.27. Components of TAT 51
4.28. TAT Measurement: Varying Frame Size 52
4.29. Start of Last Chunk Processing w.r.t End of Frame 53
4.30. TAT Measurements: 2 Symbols Per Chunk, Varying Frame Size 56
4.31. TAT with chunking only in FU SYNC 57
4.32. TAT with chunking extended to FU MAC Rx 57
4.33. Components of RTT 58
4.34. RTT Measurements: 4 Symbols Per Chunk, Varying Frame Size 59
4.35. T_{DATA_Tx} with chunking being done in FU HDR 60
4.36. T_{DATA_Tx} with chunking being done in FU MAC Tx 60
4.37. Setup for Two Concurrent Data Flows through WiNC2R 64
4.38. System Latency Components 65
4.39. System Latency Measurement for Multiple Flows 65
4.40. FU MOD Latency Measurement for Multiple Flows 66

Chapter 1

Introduction

A "Cognitive Radio"(CR) is a radio which constantly senses its environment and uses this information to configure its own parameters so as to communicate reliably and with maximum efficiency. The parameters modified can be transmission rate, power, frequency, modulation scheme and any combination of these. To give an example, if a wireless cognitive transceiver senses that the channel on which it is transmitting has improved in quality, it might choose to increase its rate of transmission. If it is communicating with another cognitive transceiver, a higher transmission rate can be agreed upon by both of them when the channel changes state, so as to achieve better communication exchange overall.

Another example might be two people sitting in a cafe with open-access Wi-Fi and exchanging information using their laptops. If the number of customers increases in the cafe, the quality of the communication link will deteriorate and the possibility of disruption of the service will increase. In laptops equipped with cognitive radio cards, the communication can be seamlessly transferred to a say a Bluetooth link or any other suitable technique when the Wi-Fi channel quality deteriorates. If the channel quality improves, the radios could jump back to Wi-Fi which may support higher data rates. In fact, a pair of "ideal" cognitive radios can even negotiate a completely new communication protocol with a specific frame format, medium contention algorithm, modulation and coding scheme etc.

Cognitive radios were first officially proposed as a radio-extension of a Software-Defined Radio (SDR), which can communicate with the network about its own current state and the state required by the network to get the best possible service [1]. A cognitive radio will use the result of this communication to configure the parameters

of the SDR. Today, this definition of cognitive radio has been expanded to include an ideal radio platform which can configure itself to suit any physical and/or medium-access and even upper layer requirements. Such a device will have ability to support adaptive power and rate control, dynamic spectrum sensing and agile frequency hopping at the physical layer along with ability to support various medium access schemes at the higher layers [2].

The chief features of Cognitive Radios are listed below [3].

1. Spectrum Sensing: A cognitive radio can scan a wide spectrum and determine frequencies being used as well as the location of corresponding transceivers. This enables the cognitive radio to determine its own transmission characteristics like operating frequency, signal modulation scheme and transmit signal strength.

2. Spectrum Etiquette Management: This feature includes the regulatory policies which will place constraints on the behavior of cognitive radio. For example, operation of a CR in licensed spectrum be restricted to regulations. These regulations will require the CR to yield the spectrum access to "registered" users for this spectrum. An example of such licensed spectrum is AM and FM radio spectrum. The regulation will require that priority be given to the broadcast radio channels in this spectrum over a CR. A CR is called a secondary user for these spectrum ranges. Another policy can be constraints on configuration parameters like transmit signal strength of the CR.

3. Modular architecture and on-the-go re-configurability: The architecture of the CR should be modular so that adding and removing modules from the radio is easy (Plug-and-play). Each module should be self describing and independent in its chief functionality from the other modules. CRs should provide suitable interface for all such kinds of modules. Also, a CR should be able to change the flow of data between these modules seamlessly while still in operation.

4. Creation of specific user profiles: A CR is able to create a set of configuration settings best suited for a specific application scenario. For example, a CR will create a specific profile for a wireless device whenever the user is at his/her home

by sensing the home network environment specifications and typical usage characteristics.

5. Adaptive algorithms: During the radio operation, CR continues to sense the environment, and will negotiate with the peers in the network to tweak its own operating algorithms to improve its efficiency.

6. Distributed collaboration: Throughout the operation, CRs share their operating characteristics, application requirements and current performance measurements with each other. This data is used to determine the policies to best utilize the network resources.

Because of the above properties CR lends itself to many applications.

1. Public Safety Applications [4]: CR's spectrum sensing and frequency adaptive abilities are useful where guaranteed communication links are a necessity. Public safety personnel and devices always need to be in touch with each other to immediately report any events. CRs will guarantee the best possible link between any two radios in a given environment. Communications from a team working in the most critical part of a burning building will be given priority above those from any other teams' and even those from multiple devices accompanying press reporters. The enhanced capabilities of a cognitive radio node can also provide broadband applications like visual sensing important to security along with narrow band voice communication. Such visual output from public safety personnel's hand-held phone-camera can help more in combating the crisis.

2. Military Applications [5]: Troops of a nation can be distributed in many parts of the world on separate missions. Differences in spectrum allocation policies in those areas and the home country can restrict usage of the communication equipment used by the troops. Also, these differences also translate to the difficulties in communication between troops from different areas. The flexibility in architecture allows the CR to support various communication technologies. Modules supporting the local technologies can immediately be added. Also, during team

operations where constant connectivity can be an issue, the dynamic spectrum allocation capability of the CR can provide improvements in the communication within the troops. Currently, Defense Advanced Research Projects Agency (DARPA) is advancing its Next Generation (XG) program on these lines.

3. Support for heterogeneous networks: Ability of CR to support a multitude of technologies enables it to act as a bridge between two devices/networks based on different communication technologies. CRs can thus operate as gateways and access points and create large extended networks made up of small networks of different kinds.

4. Location dependent services: Using the profiles created for various application scenarios, CRs can provide location dependent services. Based on data gathered when you are home, in office and on the road, a cognitive radio can create separate user profiles which suite your applications in those specific environments.

5. Efficient Spectrum Utilization: Though a cognitive radio can sense a free portion of spectrum and tune to it, it is always a secondary user if the spectrum is licensed. That is, an authorized user for that spectrum will get priority over the cognitive radio. Still, if on such a spectrum there is space available, cognitive radios can communicate. Thus the free portion of a licensed spectrum also gets used. Also, a group of cognitive radio nodes operating in any spectrum can negotiate spectrum access to increase the efficiency. For example, an increased number of customers at a time will be able to use free wireless services in cafes than currently can if the customers are using CRs.

Given the range of features described, and multitude of intended application scenarios, it is not surprising to see that after just a decade since the inception the concept, CR platforms are still in a research and development stage. If we scan the features offered by the CR, we see that the modular architecture and dynamic re-configurability are the most important features when it comes to implementation. Modular architecture implies that radio functionality should be divided into individual entities or blocks, which once configured can perform the requisite operations. Re-configurability implies

that the flow of data between these blocks might not always follow the same path. As this re-configurability is intended on-the-go; i.e. the data flow path can change during the CR operation as well; the architecture needs to support all possible data flow paths at all times. Last but not the least, is the support for plug-and-play for any new service. While the new functionality can be coded as an independent module, for plug-and-play the CR will need to offer a uniform interface to all modules. So, while the different modules can contain implementations of a diverse set of operations inside them, their way of interacting with the CR must be same.

Providing for these features makes the CR architecture much more complex to design than an architecture designed to cater to one specific application/technology. It adds lot of superfluous logic, like control flow management, all of which might not be necessary for each and every application but is there to provide CR capabilities discussed earlier. Because of this, implementation of the CR architecture will need extensive resources in terms of processing power and hardware requirements. Again, as the CR needs to support current technologies as well, it will need to satisfy corresponding operating constraints like Inter-Frame Spacing (IFS) durations in 802.11a/b/g PHY standards. With all the additional baggage, these can be tough to satisfy. Hence, each CR architecture and implementation needs to be evaluated on two points, namely resource requirements and latency, until the architecture reaches maturity.

Winlab Network Centric Cognitive Radio (WiNC2R) [6] platform is a programmable hardware CR platform designed by Wireless Information Network LABoratory (WINLAB) at Rutgers University. WiNC2R implements the radio functionality using various hardware accelerators, which are programmable by software. The software resides in a standard Xilinx soft-core processor called MicroBlaze implemented on Xilinx Virtex-5 FPGA chip. The hardware accelerators are implemented in the FPGA fabric. The accelerators are connected by a standard bus implementation.

WiNC2R provides all the features necessary for a CR platform. The radio level functionality is divided into signal processing applications which are independent of each other, e.g. MAC processing and Modulation. These applications are then assigned to individual blocks called Functional Units (FUs). Inside the FUs, the separation of

control flow and data flow is achieved by provision of a Unit Control Module (UCM) and an independent radio processing block called Processing Engine (PE) which implements the FU core operations. UCM is responsible for configuring and activating the PE, and for the data transfer in between the PEs. The flexibility of data flow is maintained by using two databases which list configuration settings for each operation performed by PE. These databases also describe all possible data paths following each PE. The data flow paths can be configured during the setup and initialization and also during the actual operation. WiNC2R provides a backbone architecture with a uniform interface to all modules, which supports plug-and-play ability. WiNC2R also supports run time re-configurability inside the modules currently implemented.

This work evaluates the architecture of WiNC2R on the ground of re-configurability, and latency. It also presents and classifies some currently available CR architectures. Out of these, two namely, the GNU Radio platform [7] and WARP platform [8] are chosen as representatives of their own class of CR platforms. We compare them with WiNC2R with hopes of identifying a correct direction in CR implementation. Specifically, we prove that while WiNC2R provides the range of re-configurability necessary for CR better than GNU radio and WARP platform, it also pays the price for it in terms of latency.

The rest of this thesis is organized as follows. Section 2 presents a survey of currently available CR architectures. We discuss various features of these platforms with detailed description of the GNU radio and WARP platforms with which we compare the features of WiNC2R. Section 3 will present the WiNC2R architecture and present its implementation of CR features. Section 4 will discuss the experimental setup for our simulations and their results. We will compare these results and compare them with some features of other platforms to draw conclusions in the Section 5.

Chapter 2

Survey of Cognitive Radio Platforms

The aim of this section is to describe a classification of CR platforms with a few examples. Some of the examples mentioned are not cognitive radios per se, but they do posses CR capabilities like re-configurability and flexibility. We describe two platforms, GNU radio and WARP platform, in detail.

2.1 CR Platforms

2.1.1 Classification of CR Platforms

[2] classifies the current CR platforms in four types.

1. Multimodal Hardware-based Platforms

   These platforms represent the most basic type of CRs. To support multiple technologies, these platforms implement modules for each technology separately, for example, separate cards for Wi-Fi and Bluetooth in commercial laptops. The cognitive functionality is limited to activating one or both of them according to the requirements. Apart from support of various technologies, these platforms do not offer any other CR feature like modularity, programmability and plug-and-play. Programmability is limited since each flow (Wi-Fi or Bluetooth) is designed to support one specific application. Also, as number of supported technologies increases, these platforms become inefficient in terms of resource consumption.

2. Portable Software Platforms

   These platforms implement the radio functionalities in the software for programmability. Since, these functions are coded in high-level programming languages, it is easy to add new modules and also the level of programmability is

high. These platforms run as application software on a general purpose or Real-Time Operating System (RTOS) installed on general purpose processors (GPPs). Because of this, these platforms can run on any system which uses the compatible OS. Although most radio processing is done in software, these platforms need RF hardware boards to provide wireless interface. These boards implement the RF functionalities like filtering, up/down conversion and ADC/DAC functions. Implementing these functions in software is difficult. The example of such platforms is GNU Radio and VANU platform. While the use of GPP makes them portable, it also hinders them since the GPP is not tailored to meet the application requirements. Also, performance of these platforms depends not only the GPP used but also on the underlying software (OS) on which these platforms run as applications. Hence, it is often found that lot of times the GPP and the OS become the performance bottlenecks.

3. Re-configurable Hardware Platforms

   This CR platform type includes multiple different platforms which use various combinations of Field Programmable Gate Arrays (FPGAs), DSPs (Digital Signal Processors) and embedded RISC (Reduced Instruction Set Computer) processors. These platforms support one technology at a time but they can be re-configured to support different technologies by downloading a new image into the FPGA. These platforms are essentially reconfigurable because of the presence of FPGA and RISC processors both of which support some degree of programmability. But on-the-go re-configurability is not supported by these platforms since a new image needs to be downloaded to the FPGA to change the functionality of the platform. The programming is done in low-level languages (like hardware development languages) and is, hence, more complex than that in software platforms. Also, the presence of different cores like FPGA and DSP constitutes a heterogeneous environment which is difficult to simulate and test. The example of such platforms is Rice University WARP platform.

4. SoC Programmable Radio Processors

These platforms make up a system-on-chip (SoC) environment using arrays of special purpose processors. Just like the re-configurable platforms, there is no standard architecture and number of processors varies from platform to platform. The functional space and processing power depends upon the number of processors. Programming is done using standard tools and the use of these with high-level language support makes programming easy. The example of such platforms is picoChip devices.

We now look into details of GNU radio and WARP platforms.

2.1.2 GNU Radio Platform

GNU radio (GNU is a recursive acronym, Gnu is Not Unix) is a free software toolkit to build software radios [9]. A software radio is a radio where all radio processing except RF and ADC/DAC is done in high level software languages. GNU radio uses C++ and Python for implementing signal processing applications. It uses C++ to code performance critical functions (many PHY layer applications) and Python for high level organization, policy and non-performance critical blocks. Python is also used to connect together the modules coded in C++. The GNU radio toolkit can be installed and run on any GPP with a compatible (usually Linux-based) OS.

To provide the air-interface and RF functions, GNU Radio is connected to an RF board through USB or Gigabit Ethernet. GNU radio uses standard RF board called Universal Software Radio Peripheral (USRP) [10]. USRP contains ADC/DACs, slots for various types of daughterboards, and an FPGA [11]. The ADC/DACs connect to the FPGA. The FPGA itself usually contains digital up and down converters and spectral shaping filters. The FPGA helps reduce the high bandwidth at the RF interface to data rates supported by USB or Gigabit Ethernet.

GNU Radio provides us with a basic block called GR BLOCK to code our own signal processing blocks(SPBs) [12]. GR BLOCK declares the basic virtual functions and the SPB needs to implement its own instance of these functions to be part of the GNU Radio architecture. For example, the GR BLOCK declares a virtual “general_work”

function. The SPB implments its own instance of this function, where it includes all its processing logic. This function also tells the SPB logic how many input items need to be processed based on output space available. The GR BLOCK also provides functions which are used by the application code to inform the control logic how many input items were consumed and how many output items were generated. To write a SPB, a . cc and a . h file needs to be generated.

The blocks written in C++ are connected in Python together to form a complete communication chain. To make this possible, C++ processing methods need to be visible in Python. The 'Simplified Wrapper and Interface Generator' (SWIG) provides this mechanism. To use SWIG, the user needs to write a . i file for each application. Hence, to create and include a new SPB, three different files, . cc, . h and . i are needed. The instances of C++ methods made visible in python are then connected together to get required functionality. Such a chain is called a flow graph.

To control the data flow between the various C++ blocks, GNU radio provides a C++ block called Scheduler. When the flow graph is created and instantiated in Python, the control is transferred to scheduler. The scheduler creates buffers (each of size 32kB) between each pair of consecutive blocks. The first block processes the data and dumps it into the buffer. The next block then reads the data from this buffer. It is the task of scheduler to call each block and pass it the input data size. After the processing is done each block returns the size of the output data written. The flow of logic in a GNU radio chain is as shown in Figure.

We show here a simple transmitter chain with a framer, CRC calculator, FEC (Forward Error Correction) coder and Modulator implemented as GR BLOCKs in C++. When Python creates and starts the flow graph, scheduler creates the data buffers shown. It then first calls the framer block with the parameter input size. We assume framer reads the input data from a text file. Framer processes the input data and stores it into the data buffer. After the processing is done, it transfers the control back to the scheduler with the size of output data produced. The scheduler then calls the next block in the chain which is CRC calculator. The output size of the framer is used as input size for this block. After CRC block is done, it transfers the control back

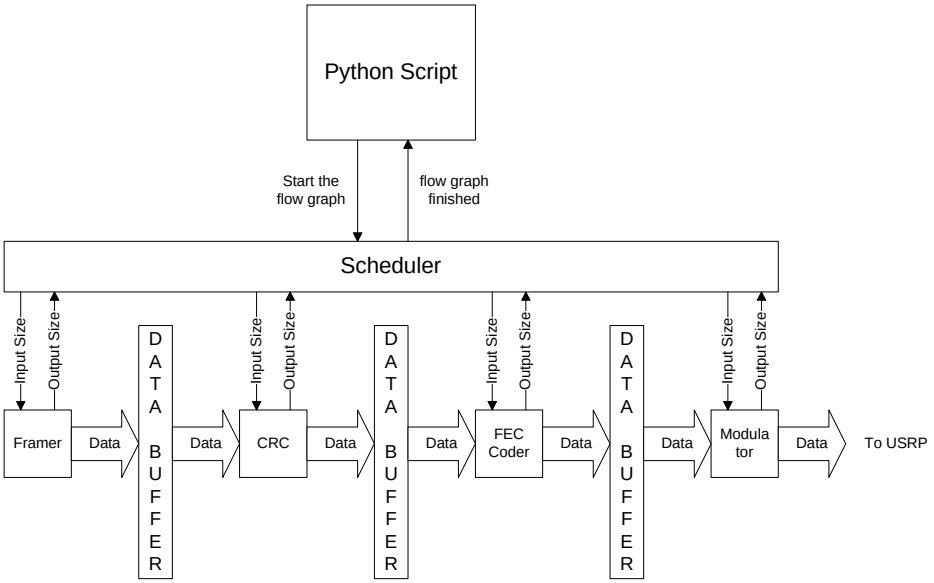

Figure 2.1: The Data and Control Flow in GNU Radio

to scheduler with the size of output data it produced. This process goes on for every block in the flow graph. After the last block is done, the control is transferred back to the python script. If the application wants to send the next packet, it can again start the flow graph.

We now analyze the GNU radio architecture in terms of CR features achieved. GNU radio is modular since it divides the radio processing into separate C++ blocks. Due to use of high-level languages like C++ and Python, programming is comparatively easier than hardware platforms. The connection of the blocks is defined in the python script and this can be changed according to application requirements, hence there is re-configurability. Since, GNU Radio requires that any radio processing block to be coded must be derived from GR BLOCK (or its subclasses), we can say that it achieves uniform interface for all new modules. In Section 3, we compare the architecture of GNU radio with that of WiNC2R in terms of these features.

2.1.3 WARP Platform

Wireless open-Access Research Platform (WARP) is a platform developed at Rice University. This platform implements all the radio functionality in an FPGA and an embedded RISC processor. The FPGA used is Xilinx Virtex-II Pro and the processor is

IBM Power PC (PPC). For detailed architecture description of WARP platform please refer to [13]. WARP implements the time-critical PHY layer functions in the FPGA fabric and uses embedded programming to code higher-level MAC layer functions. The entire structure is connected using IBM standard Processor Local Bus (PLB) as shown in the following figure [14].

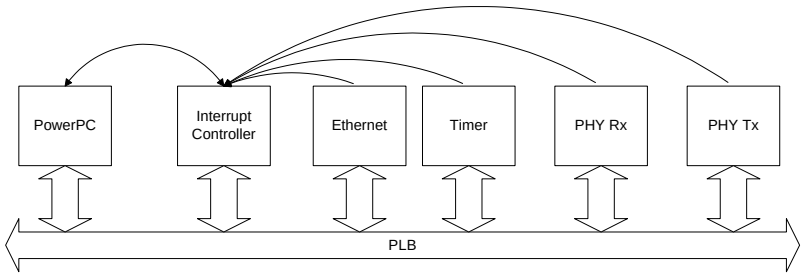

Figure 2.2: The Architecture of WARP Platform

The use of FPGA adds a degree of re-configurability in the WARP board. But it is limited because each time you need to download a new image file to the FPGA to run a different application. Hence, on-the-go re-configurability is not supported. To implement the PHY layer blocks, WARP uses blocks generated using Simulink libraries. We see that apart from the PHY layer re-configurability offered by the use of FPGA and MAC level re-configurability offered by the use of embedded programming in PowerPC, WARP does not provide any other CR features like separation of data and control flow, dynamic selection of one of several possible data-flow paths and modularity. But it achieves better speed in operations due to use of FPGA. We will compare the performance of WARP with that of WiNC2R in Section 4.

Chapter 3

WiNC2R Architecture

The chief architectural requirements of a CR are modularity, plug-and-play and on-the-go re-configurability. In this section, we see how we have achieved these features in WiNC2R through its architecture. The complete platform hardware and software architecture for WiNC2R is described in [15].

3.1 WiNC2R Architecture

3.1.1 Top Level Modular Architecture

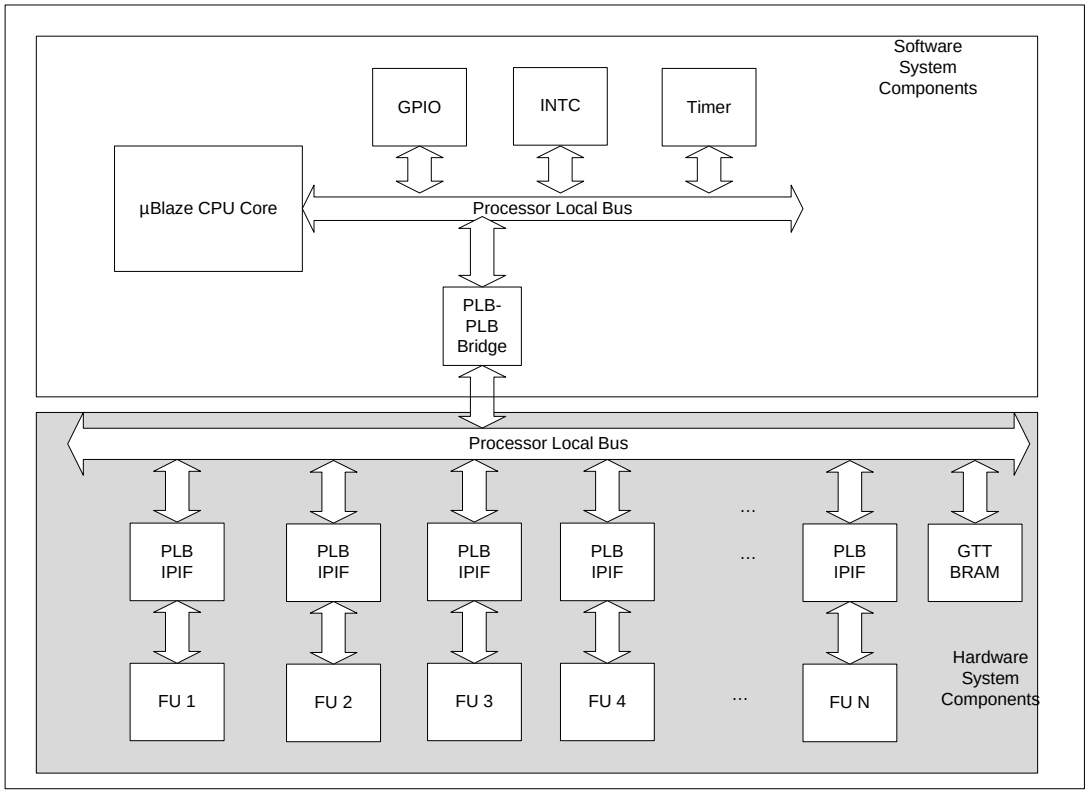

Figure 3.1: Top Level Block Diagram of WiNC2R

In figure 3.1, we show generic top level architectural block diagram of WiNC2R. The architecture of WiNC2R can be divided in two parts shown in separate dashed boxes. The upper box shows the software systems components while the lower one includes the hardware accelerator components. While the actual data processing takes place in the hardware components, the software system is responsible for their configuration and initialization. The software code sits in standard Xilnix soft-core CPU called MicroBlaze. Through this software code, WiNC2R initializes the FUs and configures the data flow between them as per the application requirements. The Microblaze core is connected to its own GPIO ports (General Purpose IO), Timer and Interrupt peripheral modules using an IBM standard bus structure called Processor Local Bus (PLB).

The functionalities of the radio processing are divided into independent blocks called Functional Units (FUs). These are equivalent to SPBs in GNU Radio and serve the same purpose. All FUs are currently implemented using a hardware description language called VHDL (VLSI Hardware Description Language). No two FUs are directly connected; instead all of them communicate using a separate PLB. The two PLBs are connected through a bridge. Having two separate buses helps to reduce capacitive loading on each of them and separates the hardware and software functionality.

Each FU interfaces to the PLB through Xilinx implementation of standard IPIF (Intellectual Property Interface). Since there is no direct interface between any two FUs, the data flow can be configured to be from any FU to any other FU.

3.1.2 Re-configurability in WiNC2R

The processing in FU can be divided into two parts; data processing and control processing. The data processing includes the core radio signal processing functionality while the control processing is the logic WiNC2R adds to achieve CR features.

The block diagram 3.2 describes the corresponding two processing blocks in the WiNC2R FU [16]. The Processing Engine (PE) implements the radio processing functions in the FU. It consists of a Processing Unit (PU) and RMAP (Register Map). Each PU has two block RAMs associated with it called input buffer and output buffer.

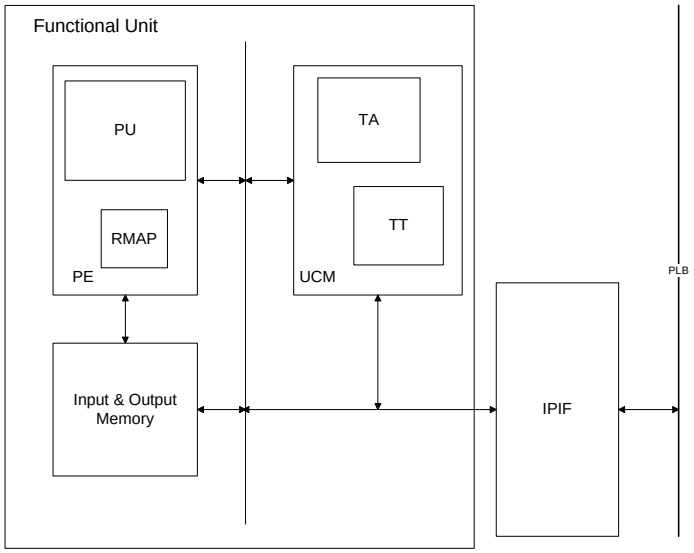

Figure 3.2: Components of the FU: PE and UCM

RMAP is also a RAM, but depending upon the PU requirements it may be implemented as a block RAM or just a set of individual memory locations. The RMAP contains software-configurable control information required for PU processing.

Unit Control Module (UCM) implements the control logic required for re-configurability in WiNC2R [17]. WiNC2R uses a task-based architecture with two types of task-related databases, called Global Task-descriptor Table (GTT) and Task-Descriptor Table (TD Table). Both databases are implemented as block RAMs. GTT is implemented as a block RAM connected to the PLB and separate from all FUs while each FU has its own TD Table. Based on its core functionality, each FU is assigned a set of input tasks and the FU is idle until it receives one of these tasks [18]. There are two types of tasks, data and control. Data task indicate that there is data to be processed present in the input buffer of the PU. For example, data task TxMod tells the Modulator block that there is data in the input buffer that needs to be modulated. This data is transferred to the input buffer of PU before the task is sent. Control tasks impart some useful control information to FUs, for example ChannelIdle/ChannelBusy control tasks tell MAC FU the channel state information which is necessary if MAC uses CSMA (Carrier Sense Multiple Access).

Each FU has its own TD Table that stores all the required information for each input

task for the FU [19]. When the FU receives a data task, the UCM Task Activation block fetches this information from the TD table and processes it. It then forwards the task to PE along with the location and size of input data. The PE processes the data and stores it in the Output Buffer. Once the processing is over, PE relays the location and size of processed data in Output Buffer to the UCM through Next Task Request and Next Task Status signals. The Next Task Status bits indicate the location of the processed data in the output buffer. Depending on the PU, the output data may be stored at more than one location. The Next Task Request signal indicates the UCM Task Termination (TT) block how the output data at locations indicated by status bits is to be processed. The TT processing includes transferring the data to next FU/FUs in the data flow. The NT Request tells the TT to which FU the data is to be sent. The next PE (or FU) in the data flow in determined the information stored in TD table. By updating the information in TD table, software can change the next FU in the data flow path. This way the data flow path is reconfigured on-the-go.

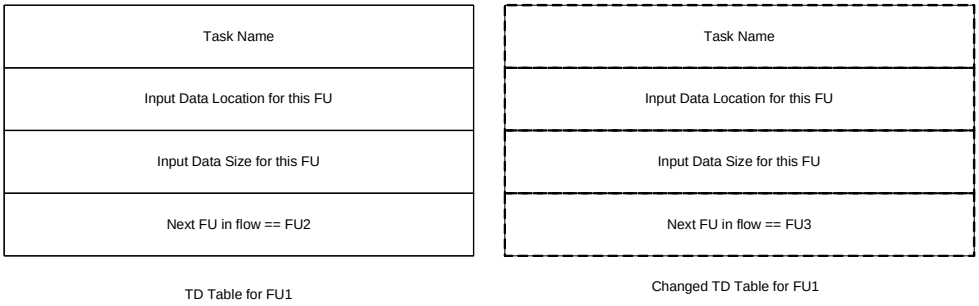

Figure 3.3: The Two Versions of TD Table for FU1

Figure 3.3 and 3.4 show us an example of this. Figure 3.3 shows TD table entry for FU1. It designates the next FU in the data flow as FU2. The TD table for FU 2 (not shown in figure) will have similar entry for FU3. The resulting flow is as shown in Figure 3.4 by solid lines. FU1 receives a task. UCM TA block does activation and passes the task with input data location and size to PE1. PE does task processing (TP) and transfers the NextTaskRequest along with output data location and size. UCM TT block then looks up the next FU for the current data flow and finds out that it is FU2. It then initiates the data transfer between FU 1 Output Memory to FU2 Input Memory and after that is done, generates a task for FU2.

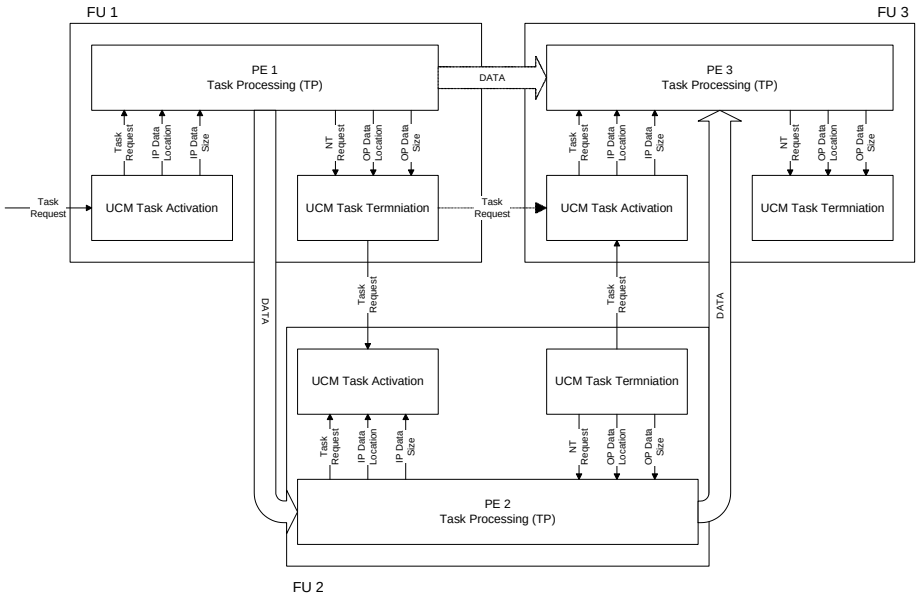

Figure 3.4: The Effect of the Change in TD Table on the Data Flow

Before UCM TT block generates task request for FU2, it needs to look up some information regarding the task request to be generated. This information is specific to FU2 and hence not stored in TD table of FU1. For this information, the UCM accesses Global Task-descriptor Table (GTT). Unlike TD table, there is only single GTT for all FUs and it is implemented in a separate memory than all the FUs as shown in figure. This table contains the information UCM TT block of each FU might need. The details of all these operations are described in [15]. FU2 processing goes through same operational stages (TA, TP and TT) and same happens for FU3.

Now referring again to Figure 3.3, we see in the dashed box an updated TD table entry for FU1. Here the next FU in the data flow is changed to FU3, cutting out FU2 from the flow. The reasons behind this change can be more than one. For example, assume FU1 does MAC processing, FU2 Forward Error Correction coding and FU3 modulation. It may happen that the CR senses that channel quality is good enough to do away with the FEC doing. In that case data will flow from MAC processor (FU1) to Modulator (FU3) directly. To implement this change, software just changes the FU1 TD table next task entry from FU2 to FU3. The resulting data flow is as shown in Figure 3.4 by dotted arrows going from FU1 to FU3.

We also note in Figure 3.4 the clear separation of the data and control flow in the WiNC2R. While the data flows between PEs, the control information flows between the corresponding UCMs. The overall processing in each FU is divided into three non-overlapping stages namely, TA, TP and TT.

3.1.3 Current WiNC2R Implementation

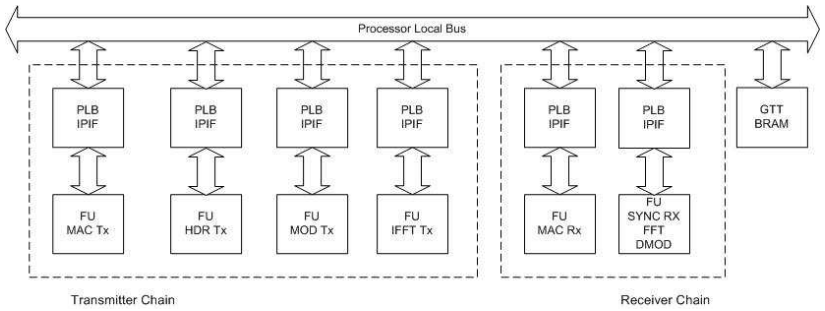

Figure 3.5: FUs Currently Implemented in WiNC2R

The figure shows FUs currently implemented in the WiNC2R. The six FUs are divided into a transmit chain of four FUs and receive chain of two. Following is the brief description of each of them.

1. FU MAC Tx: This FU currently provides two medium access schemes, ALOHA and CSMA-CA back-off. It also uses 802.11a **??** compatible inter-frame spacing (IFS) durations and frame formats.

2. FU HDR Tx: This FU adds 802.11a PLCP Header to the MAC frame. This FU instructs the Tx IFFT to start generating the preamble by means of task request. While processing data, it divides the MAC frame into chunks. Each chunk is processed individually and forwarded to the next FU [20]. The chunking introduces pipelining in the transmitter operation. Since at a time a small portion of a frame is getting processes and stored in the memory, it also reduces the memory requirements.

   The size of the individual chunk is determined by the modulation scheme used. In terms of number of OFDM symbols, each chunk (except the last one) is equal

to four OFDM symbols. FU HDR pads the last chunk to next immediate integral number of OFDM symbols. Since the number of bytes in an OFDM symbol changes with modulation scheme used, the size of each chunk in bytes also changes with modulation scheme. Usually, for a given modulation scheme, the size is same for all chunks except the first and last one. The first chunk size differs since it may contain PLCP header and part of MAC frame. Regardless of the modulation scheme chosen the PLCP header is always modulated using BPSK.

For a given modulation scheme, size of first chunk and size of subsequent chunks is specified in TD Table. The UCM TA block is responsible for doing the chunking operation.

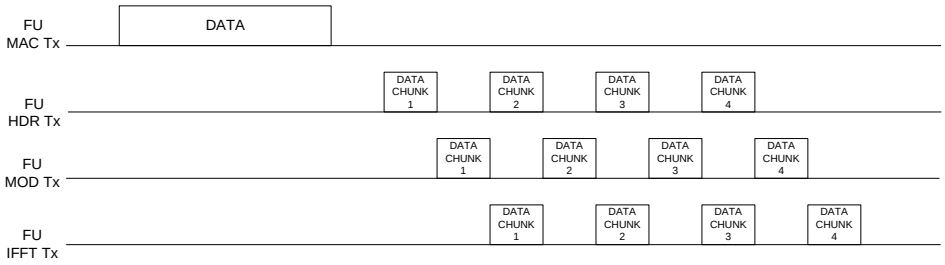

Figure 3.6: The Effect of Chunking on the Frame

The effect of chunking is as shown in Figure 3.6. We see the output of all FUs in the transmit chain.

Current implementation supports BPSK and QPSK modulation schemes. The value of $firstchunksize, chunksize$ is $16, 24$ and $34, 48$ bytes respectively for these.

3. FU MOD Tx: This FU implements various modulation schemes compliant with 802.11a.

4. FU IFFT Tx: This FU implements an IFFT to convert incoming bytes to OFDM symbols. It also generates and adds short and long preamble before the frame.

5. FU MAC Rx: This FU receives the frame and performs various checks. If it finds that frame is meant for it and is error free, then it stores the frame into a separate

RAM and alerts the CPU by means of an interrupt. This FU processing is also 802.11 based.

6. FU SYNC Rx FFT DMOD: This FU is combination of four individual FUs, FU SYNC, FU Rx FFT, FU DMOD and FU CHKR. These were combined to reduce the resource utilization. Before combining them, each FU had its own UCM. By integrating the FUs, we removed three UCMs from the design.

   This FU detects the start of frame, converts it from OFDM symbols to data bytes, demodulates it and assembles all the chunks into a single frame. While doing this, it also calculates checksum over the complete frame and reports the result to MAC Rx.

3.1.4 Plug-and-play and Re-configurability Inside PE

We have seen in previous subsections how we achieve modularity and on-the-go re-configurability for the data flow path amongst the modules. We now see implementation of two other CR features, plug-and-play and re-configurability inside the modules. We refer to block diagram of PE shown in Figure 3.2.

The PE itself is a just a wrapper and does not implement any radio processing logic. As shown in figure, it wraps around two blocks called Processing Unit (PU) and Register Map (RMAP). All radio processing logic is implemented inside the PU. The RMAP is a block of registers (memory locations) required for PU operation. There are registers which can be updated by CPU during the reset stage and they help achieve the re-configurability inside the PE. For example, PE MAC Rx implements 802.11 compliant IFS durations. It uses timers for this purpose. These durations are different for each substandard of 802.11 like a, b and g. Hence, whenever PE MAC Rx wants to wait for a specific IFS duration, it just reads the IFS value from a location in the RMAP. It is the responsibility of CPU to update this location with the correct value for current standard implemented. Hence, PE MAC Rx is reconfigurable for 802.11 a, b and g standard.

The interfaces of PU and RMAP will differ according to a specific application. But

UCM needs a uniform interface to communicate with the PE. The PE wrapper supports this uniform interface. It hides all the PU specific interface ports by mapping them internally to the PE ports. We can thus say that the WiNC2R backbone architecture consists of the software components, the PLB components and the UCM and PE wrapper. All application specific logic is contained in PU and RMAP. Using the PE wrapper, the WiNC2R offers a uniform interface to all the PUs thus achieving plug-and-play.

3.1.5 Architectural Comparison of GNU Radio and WiNC2R

We now see a brief comparison of WiNC2R and GNU Radio architectures.

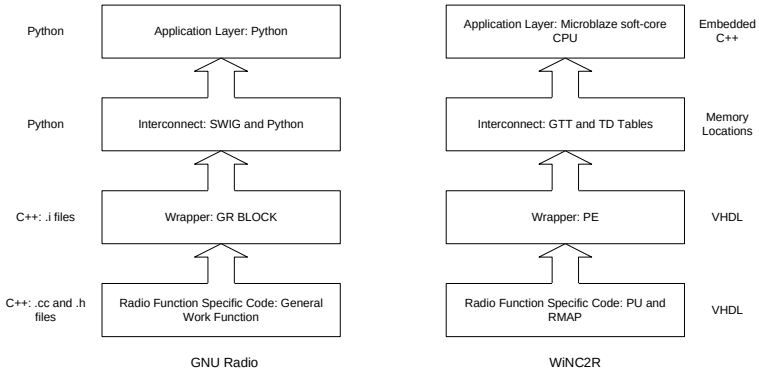

Figure 3.7: The Summary of GNU Radio and WiNC2R Architecture

The Figure 3.7 summarizes the WiNC2R implementation and GNU radio implementation. In GNU Radio platform, the radio processing is implemented in C++ while in WiNC2R we do it in VHDL in PU and RMAP blocks. GNU Radio provides the GR BLOCK interface for plug-and-play while we provide the PE wrapper written in VHDL. The connections between the modules are specified using SWIG and Python in the GNU radio. In WiNC2R these connections are coded in the two databases, GTT and TD tables. The application layer code, from where the payload is written to the radio processing functions, sits in the Python script in GNU radio while we use the Xilinx soft-core embedded CPU Microblaze for the same purpose.

Referring to Figure and Figure 3.4 we see that the control flow is very similar in GNU radio and WiNC2R. In both, there is controller block which activates the radio

processing block with input data size and in case of WiNC2R only, the location. The difference is that in WiNC2R, each radio processing block has its own controller while in GNU radio all the blocks share the same controller. Also, in GNU radio consecutive blocks in the flow graph share the data buffers while in WiNC2R, we have separate buffers for each block. Nevertheless, we see that same degree of programmability and re-configurability has been achieved in both GNU Radio and WiNC2R.

Chapter 4

Testing and Simulations

In this section, we describe the test-bench and simulations used to analyze the performance of the WiNC2R.

4.1 WiNC2R Test-Bench

4.1.1 A Generic Test-Bench to Test Hardware Systems

To simplify the explanation of the test-bench used for evaluating WiNC2R, we first present a generic test-bench model that can be used to test any hardware system [21]. Consider an example of Two-Bit-Input adder as a system that needs to be tested. The complete behavior of the adder can be described by the truth table shown in Table 4.1.

Test Case No.	Input Ports		Output Ports	
	A	B	C	S
1	0	0	0	0
2	0	1	0	1
3	1	0	0	1
4	1	1	1	0

Table 4.1: Truth Table for Two-Bit-Input Adder

There are four possible test cases represented by four AB combinations. The vector AB is called input vector. The output vector of Carry and Sum is denoted by CS. The block diagram of the generic test-bench for this adder is as shown in Figure 4.1.

The system that needs to be tested is called a Design-Under-Test (DUT). Here, the Adder is the DUT. Its input ports are connected to the Driver block while the output ports are connected to the Monitor block. The simulation manager knows all possible

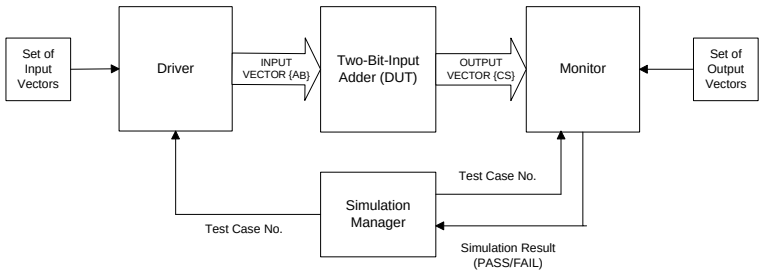

Figure 4.1: A Generic Test Bench for Two-bit Input Adder

test-cases described in Table 4.1. The Driver knows all possible input vectors for each test case i.e. four possible values of set AB. The Monitor knows the corresponding output vectors. The simulation manager determines which test case is to be simulated and provides the test case information to the Driver block. The driver then generates the input sequence, AB, corresponding the test case number and sends it to the DUT. The Monitor is also informed about the test case being executed. The monitor compares the output of the DUT, CS, with the expected output for a given test case number and informs the result to the simulation manager. The manager then takes the decision whether to execute the next test case or to report an error and stop the simulation. The manager can be programmed to carry out hundreds of test cases in a loop and report the collective result back, or it can be programmed to interrupt the simulation after the first error.

4.1.2 Need for an Automated Test Bench

Our goal is to test the performance of whole WinC2R system with respect to latency and throughput. The WiNC2R architecture described in Section 3 can be roughly divided into two parts: software system components and hardware system components as described in Figure 3.1. The software system components include the MicroBlaze and its peripherals like GPIO, timer and Interrupt Controller. The hardware system includes all the FUs and the PLB bus that connects them. Currently, the software implements only the basic functionalities of initializing and configuring the hardware components. We do not yet have a full-fledged software system that has CR features like gathering data from hardware components and making decisions based on them or

negotiating with other CR nodes for efficient spectrum access. The hardware components on the other hand already provide required CR features like uniform interface for every module, separation of control and data flow, and dynamic re-configurability of the data-flow path through the system. We hence conclude that the hardware architecture is ripe and stable enough to be evaluated for performance. Such performance analysis will give useful information about the cost the WiNC2R pays in exchange for all the CR features mentioned above. It will point out the bottlenecks in the architecture; it can also present an analytical model for the same and point out the right direction for future developments. We provide such performance analysis of WiNC2R hardware system in this thesis.

To test just the hardware system, we needed to substitute the software system with the three blocks described in Figure 4.1, namely simulation manager, driver and monitor. Since we use IBM standard PLB bus core in our system, the driver and monitor need to have an interface to the bus core. The driver needs to have all the initialization code for the FUs. For WiNC2R, this stage includes updating the data at thousands of memory locations. The driver also needs to make the system ready for communication by initializing the receiver, so that it starts scanning the spectrum. In the last stage, the driver needs to start data communication by first writing the application data to the MAC transmit block and triggering it by writing the SendFrame task to it. All these operations need to be checked for correct execution. This involves reading all the updated memory locations to confirm the presence of valid data. We need to confirm that all the initializations and activation data is getting written at correct memory locations. The sheer size of these operations in terms of the number of memory writes and reads to be performed convinced us that at least some part, if not all, needed to be automated. Though we could not generate the test data automatically since it was application specific, we could automate the read and write operations. The WiNC2R development team already had used Xilinx cores like Microblaze, Xilinx implementations of the standard IBM PLB bus core and IPIF for each FU. Hence, for the reason of compatibility, ready availability and familiarity, we chose to use the Xilinx Bus Functional Model (BFM) simulation system to test WiNC2R.

4.1.3 BFM Testing Environment

BFM system includes Master, Slave and monitor components for IBM standard core-connect buses [22] [23]. We use the components for PLB bus core. The Bus Functional Language is used to describe the behavior of these components and the Bus Functional Compiler translates a BFL file into executable commands for the BFM components. The block-level diagram of BFM simulation is as shown in Figure 4.2.

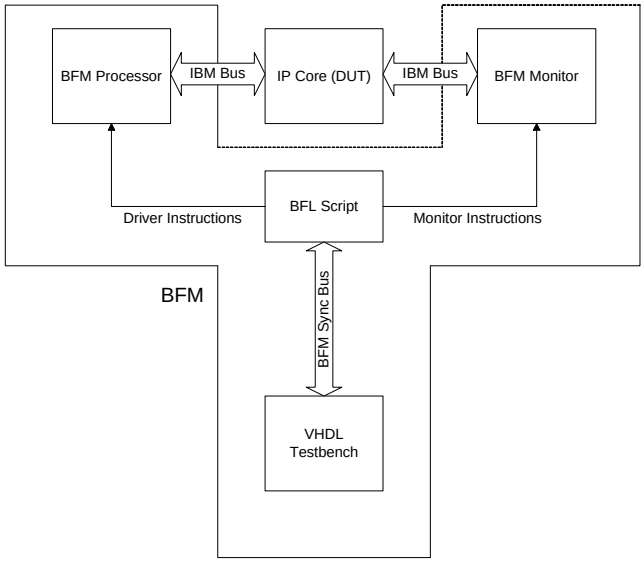

Figure 4.2: A Generic Test Bench using BFM

Figure 4.2 is drawn as an extension of the basic model of Figure 4.1 to facilitate a clear understanding of our test-bench; and can be explained in similar manner. The BFM here emulates the full behavior of a microprocessor. It contains a bus master component which acts as Driver and is called BFM processor. It also contains a monitor component called BFM monitor. The VHDL test-bench acts as simulation manager and control the behavior of both processor and monitor through the BFL script. The BFL file initializes both the processor and monitor devices. It contains series of memory reads and writes. The writes are instructions for processor while reads are instructions for monitor. A sample BFL which initializes both the devices and does read-write operations is shown in Figure 4.3.

The first two lines in the script shown in Figure 4.3 specify the generic definitions. The generics are user-defined and can be as many as required. The next two lines

```
set_alias(ADDR1   = 0xC3621030)
set_alias(DATA1   =  0x01010101)

set_device(path = /bfm_system/bfm_monitor/bfm_monitor/slave, device_type = plb_slave)

set_device(path = /bfm_system/bfm_processor/bfm_processor/master, device_type = plb_master)

mem_update(address = ADDR1, data =  DATA1)
write(addr = ADDR1, size = SINGLE_NORMAL, be = WORD0)

mem_update(address = ADDR1, data =  DATA1)
read(addr = ADDR1, size = SINGLE_NORMAL, be = WORD0, req_delay = 100)
```

Figure 4.3: A Sample BFL Script

initialize the processor and monitor using the set_device command with the full path of device files. The last four lines contain one write and on read operation. The read operation is executed after a 100 clock cycle delay to give the write operation sufficient time to finish.

The VHDL test-bench communicates with the BFL script using the BFM synchronization bus (synch bus) as shown in Figure 4.2. This contains two independent 32-bit unidirectional buses going to and from the BFL script called Synch_Out and Synch_In. The BFL script can be made to wait upon high signal on any pin on its input synch bus. The script in turn can drive any pin on the synch_out bus high. The test-bench can wait for this stimulus to do further operations.

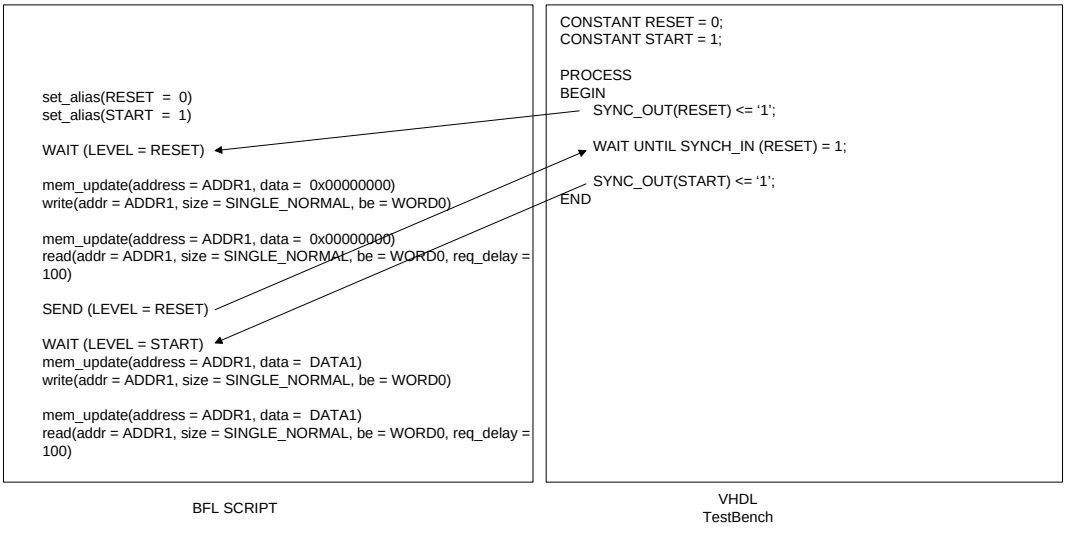

Figure 4.4: Interaction between BFL Script and VHDL test-bench

A simple example of communication between these two entities using synch bus is as shown in Figure 4.4. It shows same code as shown in Figure 4.3, only modified by

addition of synch bus signals. When simulation begins, test-bench makes RESET pin (pin number zero) level high. The BFL script is waiting for this signal and performs reset operation. In this example, the reset operation includes writing zeros to given memory location and confirming that they are correctly written by reading the same memory location (ADDR1) after some delay. The test-bench meanwhile is waiting for the reset processing to be finished. When the script makes its own output pin zero level high, test-bench knows that reset is done and communicates to the script to start normal operations using the START pin.

4.1.4 WiNC2R Test-Bench using BFM

We now describe the system used for testing the WiNC2R. The hardware components of WiNC2R are a collection of various FUs each with its own IPIF to the PLB. Since performance of each FU needs to be tested, we use each FU as a separate DUT in our test-bench. This enables us to use separate simulation manager and to monitor each FU's performance. Each simulation manager has its own synch bus interface with the BFL script.

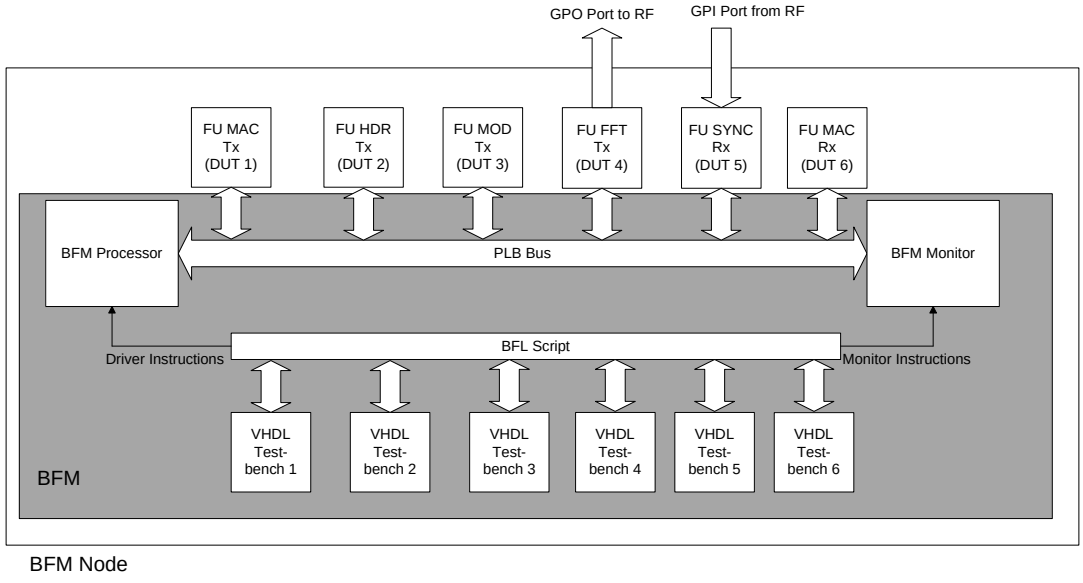

Figure 4.5: Block Diagram of a BFM Node

The top-level diagram of BFM test-bench for WiNC2R is as shown in Figure 4.5. The structure in the figure describes a complete transceiver system as far as simulation

is concerned. The software components of the WiNC2R architecture are replaced by the BFM components. The BFM system (first described in Figure 4.2) is denoted by the inner shaded box. The outer box is referred to as 'BFM Node'. We treat this black-box as a complete communication system. The GPIO(General Purpose IO) interface represents the RF interface of the BFM Node. The output of the transmit chain is sent to RF through GPO port while input from the RF is fed to the synchronizer block in the receive chain of WiNC2R. All the FUs share the same PLB bus. The bus arbiter is not shown in the figure for convenience.

4.2 Basic Analytical Experiment

The aim of this experiment is

- To examine the complete task flow in the transmitter and receiver chain in the WiNC2R
- To examine the concept of chunking and associated configuration parameters
- To analyze the processing latency in each FU for processing the task and find out the UCM and PE processing delays

4.2.1 The Experimental Setup

Figure 4.6 shows the experimental setup for this experiment. Two BFM Nodes(Figure 4.5) are connected through their GPIO interface. The common channel between the two nodes was simulated by ORing the output of the two GPO ports together to form a common output. This output was then looped back to the GPI port of each node. The GPO port of a node is driven low when a node is not transmitting.

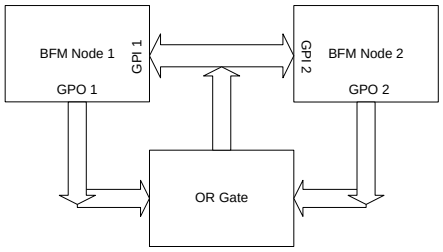

Figure 4.6: Communication between two WiNC2R nodes in BFM simulation

The functional details for the experiments are as follows. ALOHA MAC protocol is simulated on the MAC FUs with the PHY layer blocks providing 802.11a-like OFDM PHY at 5 GHz, 12 Mbps data rate using QPSK Modulation scheme. It is to be noted that the PHY layer implementation is not fully compliant with 802.11a. The format used for MAC frame is 802.11 compatible. Only two frame types are used however, namely DATA and ACK. For all experiments, data transfer takes place as depicted in Figure 4.7.

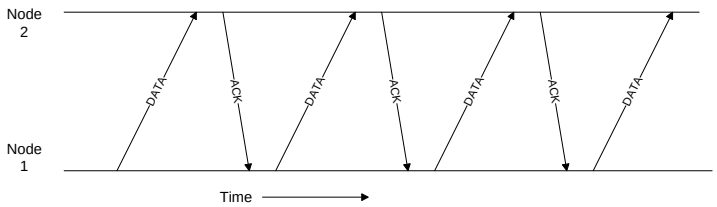

Figure 4.7: Data Transfer using simple ALOHA MAC Protocol

Node 1 initiates the data transfer by sending a frame to Node 2. It then waits for an ACK from Node 2 before sending the next data frame. Note that we do not use Stop-and-Wait ARQ [24]. Instead, node 1 keeps waiting until the ACK is received from Node 2. Since the channel simulated between them is perfect, there are no packet-drops and hence, there is no danger of Node 1 waiting indefinitely for ACK. Also, we assume that Node 1 always has data to send to Node 2. Hence, the data transfer can go indefinitely and is only limited by the simulation time specified in the test-bench. The MAC frame size used for this experiment is 500 bytes including the MAC header, payload and checksum. We use the frame format for the data frame traveling within a BSS(Basic Service Set). The frame format for the data frame is shown in Figure 4.8. The size of the MAC header for such a data frame is always equal to 24 bytes. As seen from the Figure 4.8 checksum size is also fixed at 4 bytes. Thus, the payload for the 500 byte sized frame is 472 bytes long. The ACK frame is always 14 Bytes long. Figure 4.9 shows the frame format for the ACK frame.

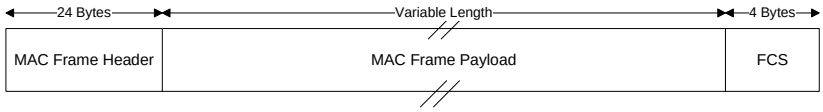

Figure 4.8: Data Frame Format

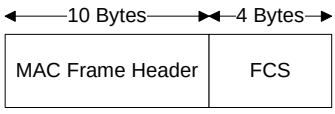

Figure 4.9: Ack Frame Format

4.2.2 Measurements and Analysis: Data Frame Transfer

In this section, we study the complete task and data flow through the WiNC2R FUs for the frame transfer depicted in Figure 4.7. We describe in brief the operations and processing latencies in each FU. The discussion and results from this experiment will make the following experiments easier to understand.

Figure 4.10 shows the task flow in the transmit chain of node 1 and receive chain of node 1 as the data frame progresses through the chain. For all FUs, the UCM task activation processing latency T_{TA}, PE task processing latency T_{TP} and UCM task termination latency T_{TT} are measured as shown in the Figure 4.11. The total processing latency for UCM operations is denoted by T_{UCM}. It is sum of T_{TA} and T_{TP}. We will denote the total FU Latency by T_{FU_NAME} for all FUs, e.g. $T_{FU\_MAC\_Tx}$. It is sum of T_{UCM} and T_{TP}.

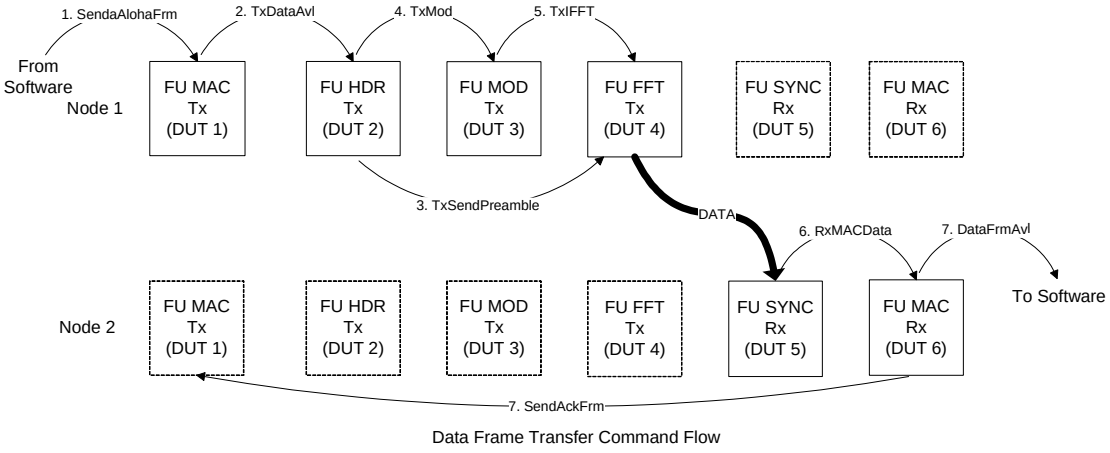

Figure 4.10: Task Flow for Data Frame Transfer

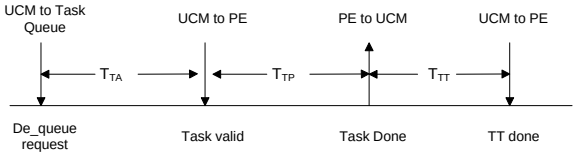

Figure 4.11: Components of FU Latencies

4.2.2.1 FU MAC Tx Analysis

1. Initialization

The experiment begins with the MAC Frame header and the payload data getting written into the Header RAM and Input Buffer of MAC Tx PE respectively. The BFM Processor then writes the descriptor for the 'SendAlohaFrm' task to the task queue of FU MAC Tx.

2. UCM TA Processing

   The UCM TA block fetches the task descriptor from the task queue. It then reads the task information in TD Table and updates the 'InDataPointer' and 'InDataSize'. There is only one input buffer region used for this task. The TA block then sends 'task_valid' signal to PE.

3. PE MAC Tx Processing

   The PE then generates the frame, using the frame header and payload bytes, according to the frame format shown in the Figure 4.8. It also calculates and appends the CRC checksum word at the end. Along with data, PE MAC Tx also generates a 32-bit vector called 'TxVector' and stores it separately from the frame data in a different output buffer region. The details of this vector are not relevant here. The 'TxVector' generation is specific to 802.11 scheme. The task processing ends with task_done signal to UCM.

4. UCM TT Processing

   The UCM TT block then generates the 'TxDataAvl' task for PE HDR. The TT processing involves the transfer of data from two output buffer regions.

5. Processing Latencies

   Table 4.2 shows the processing latency measurements for FU MAC Tx.

T_{TA}	T_{TP}	T_{TT}	$T_{FU\_MAC\_Tx}$	T_{UCM}
μs	μs	μs	μs	μs
0.48	4.08	5.65	10.21	6.13

Table 4.2: FU MAC Tx Measurements: Fixed Frame Size

We note that the T_{UCM} is larger than T_{PE} and T_{TT} latency is the highest component in FU MAC Tx processing latency.

4.2.2.2 FU HDR Tx Analysis

1. UCM TA Processing

   The UCM of FU HDR Tx performs the chunking operation on the frame. This operation is implemented in the UCM TA block. TA block fetches the task 'TxDataAvl' queued in the task queue. It reads the task information in TD table and updates the 'InDataPointer' and 'InDataSize' for buffer in which 'TxVector' is stored. For the data region in input buffer, the pointer and size is updated according to the 'FirstChunkSize' and 'ChunkSize'. Then, the task descriptor for the next chunk is stored in the task queue and at the end, 'task_valid' sent to PE. The process is repeated for each chunk, except the last one. In the TA processing for last chunk, the UCM does not need to write the task for next chunk. The last chunk is decided based upon the size of remaining data to be processed.

   The number of chunks in the 500 Byte MAC frame for current settings is 11. The size of first chunk is 34 Bytes; for intermediate 9 chunks, it is 48 Bytes and the size of the last chunk is remaining 34 Bytes. These chunks are input to PE one by one. The data is divided into chunks as per the Equation 4.1.

   $$500 = 34 + (9 * 48) + 34 \tag{4.1}$$

   When the task for the next chunk is scheduled and stored in the task queue, the start of processing time of that task is calculated using the Reschedule Period. The tasks for every two consecutive chunks are separated by Rescheduling Period. Currently, the value of Reschedule Period is set at 2560 clock cycles, which translates to 25.6 μs for the simulation frequency of 100 MHz.

2. PE HDR Tx Processing

   For the first chunk, PE HDR adds the PLCP (Physical Layer Convergence Protocol) Header before the MAC frame. The format used for the PLCP header is defined as shown in the Figure 4.12. The Rate and Length information is read from the 'TxVector' from the PE MAC Tx.

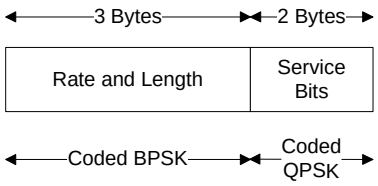

Figure 4.12: PLCP Header Format

The PLCP header format used here is similar to the one specified in 802.11a standard. But the size is different. The format in the standard uses 3 byte field for 'TxVector', but our implementation pads it to 6 bytes. This padding is achieved by duplicating each bit. The reason is that the standard assumes that the PLCP header will be coded using 1/2 rate convolution coder. The WiNC2R does not have a coder in the current implementation. Hence, we compensate for the size expansion by the coder by padding. Doing this is important since the standard requires the 'TxVector' information to be sent as a single OFDM symbol. The service bytes are modulated using the modulation scheme used for the MAC frame. We have used QPSK modulation for the frame. Also for the first chunk, while processing the data, PE HDR also generates 'TxStartPreamble' task for Tx IFFT. This task informs the the Tx IFFT to start processing and storing the preamble. The task processing for the first chunk also includes the TT for this task.

For the intermediate chunks, there is no processing done. PE HDR just forwards them to the Output Buffer. For the last chunk, if the data is not equivalent to integer number of OFDM symbols in size, PE HDR pads zeros at the end to make it so.

The size of the first chunk at the output is 42 Bytes, because of the addition of PLCP header. The size of the intermediate chunks at the output is 48 Bytes, since there is no processing for these. For the last chunk, the size at the output is 36 Bytes, because the PE HDR pads the 34 Byte input chunk by 2 Byte zeros. 36 Byte output chunk translates to 3 OFDM symbols at the IFFT, which was the desired effect.

3. UCM TT Processing

   After each chunk processing, UCM TT generates the 'TxMod' task for FU MOD. The data transfer includes two output buffer regions.

4. Processing Latency

   Figure 4.13 shows the effect of chunking in PE HDR. We see that, for each chunk there is one instance of T_{TA}, T_{TP} and T_{TT}. This is because each chunk is processed as a separate task. We also see the effect of Rescheduling Period which separates the consecutive de-queuing requests from the UCM to the task queue. The time between the 'TT_Done' signal of one chunk and the de-queuing request for the next chunk is called T_{IDLE}. In this time ,the FU HDR UCM is just waiting for the start time of the next chunk.

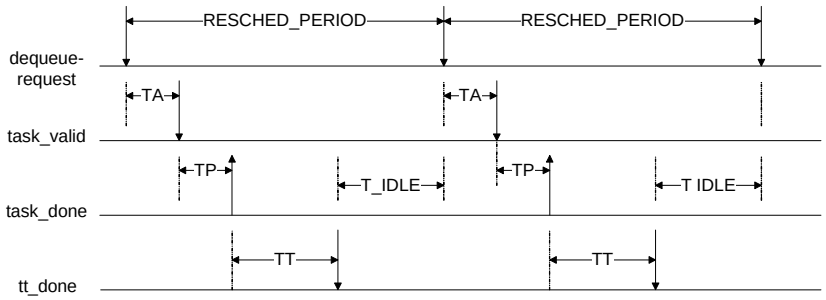

Figure 4.13: Effect of Chunking and Reschedule Period in PE HDR

Figure 4.14 shows the T_{TA}, T_{TP}, and T_{TT} processing latencies for each chunk.

It is seen that for all chunks except the first one, T_{TT} is the dominant part. When the PE HDR receives the task for the first chunk, it generates the task 'TxStartPreamble' for Tx IFFT and simultaneously starts processing data. The TA latency for the first chunk also includes the processing done in the HDR UCM TT block for the TxStartPreamble task. Hence, the T_{TA} value for the first chunk is higher than the rest.

Figure 4.15 shows the total processing latency in FU HDR, denoted by T_{FU_HDR}, and the parameter T_{IDLE}. Note that the T_{IDLE} parameter is zero for the last chunk. T_{FU_HDR} is sum of the components shown in Figure 4.14.

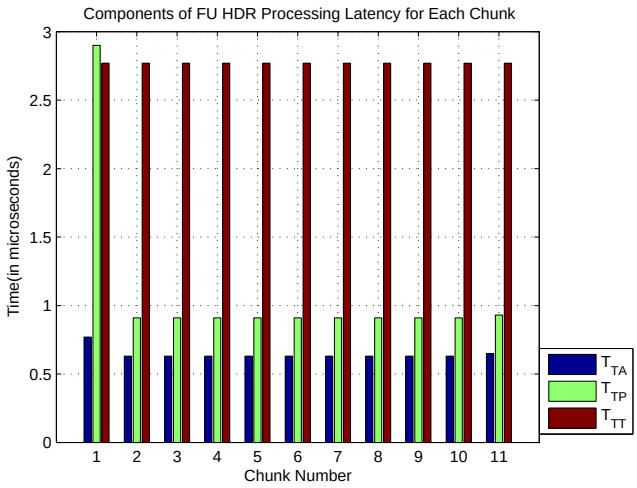

Figure 4.14: Processing Latencies in FU HDR: Fixed Frame Size

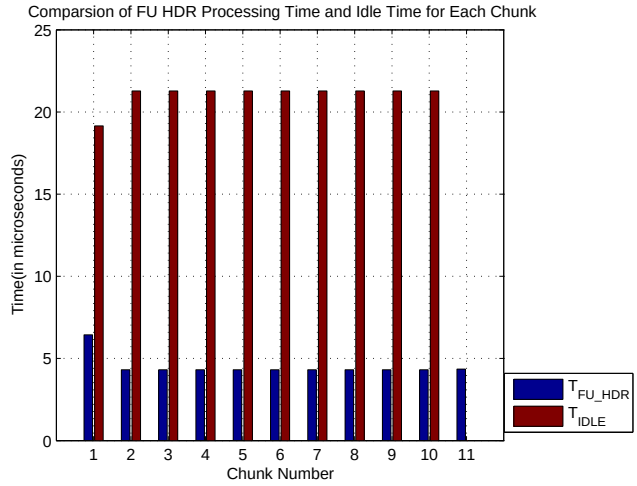

Figure 4.15: Comparison of T_{FU_HDR} and T_{IDLE}: Fixed Frame Size

We see that T_{IDLE} is almost always 3 or 4 times larger than the T_{FU_HDR} because of the Rescheduling Period. The UCM is a complex block and requires lot of resources on the FPGA (7% for each UCM on the currently used Virtex-5 FPGA) and it seems that all these resources are not getting used for almost 80% of the time. It is to be noted that this apparent wastage is not due to UCM architecture but because of the requirement imposed by the 802.11a-standard and corresponding implementation. We suggest a few methods to better utilize T_{IDLE} in later sections.

4.2.2.3 FU MOD Tx Analysis

(a) UCM TA Processing

The TA stage includes fetching the task descriptor from the task queue and updating 'InDataPointer' and 'InDataSize' for two buffer regions.

(b) PE MOD Tx Processing

The most important thing in this FU is that it expands the data by large magnitudes. For QPSK, every pair of 2 bits is converted to a 32-bit word. Hence, the data size multiplication factor is 16. Since the modulator gets the data in small chunks; the required output space is also limited. If the whole 500-byte frame was processed without chunking, the required output space would have been 8000 Bytes. But the chunking minimizes the output memory requirement for each task to maximum 48 * 16 = 768 bytes at a time.

(c) UCM TT Processing For each chunk, 'TxIFFT' task is sent to PE Tx IFFT. There is data in two output buffer regions to be transferred.

(d) Processing Latencies The FU latencies for each chunk are as shown in Figure 4.16. We note that just like previous FUs, the T_{TT} is larger than any other FU latency component.

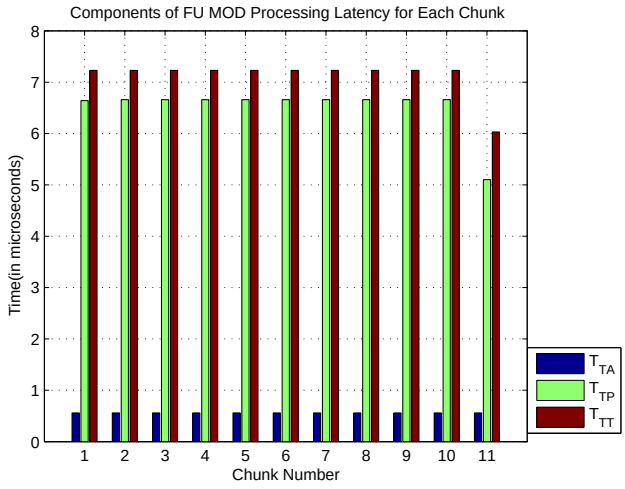

Figure 4.16: Processing Latencies in FU MOD: Fixed Frame Size

4.2.2.4 FU IFFT Tx Analysis

(a) UCM TA Processing

The TA stage includes fetching the task descriptor from the task queue and updating 'InDataPointer' and 'InDataSize' for two buffer regions.

(b) PE IFFT Tx Processing

The IFFT PE converts the incoming data into OFDM symbols and outputs them onto the GPO interface. Hence, there is not data transfer involved in the T_{TT}. The current implementation is 802.11a-specific and is only able to handle single OFDM flow at a time. PE starts outputting the PLCP preamble onto the GPO when it gets the TxStartPreamble task from the PE HDR. The first chunk of the frame should reach the IFFT and should be ready to be outputted before the preamble is over. This timing is made sure by the correct setting of the rescheduling period.

(c) UCM TT Processing

The TT stage includes transferring only the context information required for processing the next chunk.

(d) Processing Latencies

The FU latencies are as shown in Figure 4.17. Here, it is noted that the T_{TA} latencies dominate the other components. This is obvious since both TA and TT operations in FU IFFT contain only control operations and no data transfer.

4.2.2.5 FU SYNC Rx Analysis

(a) UCM TA Processing

The TA stage of FU SYNC Rx is part of the initialization stage in the test script. It consists of fetching the 'RxStartRcvCtl' task descriptor from the task queue. This task processing is not included in the processing latency analysis.

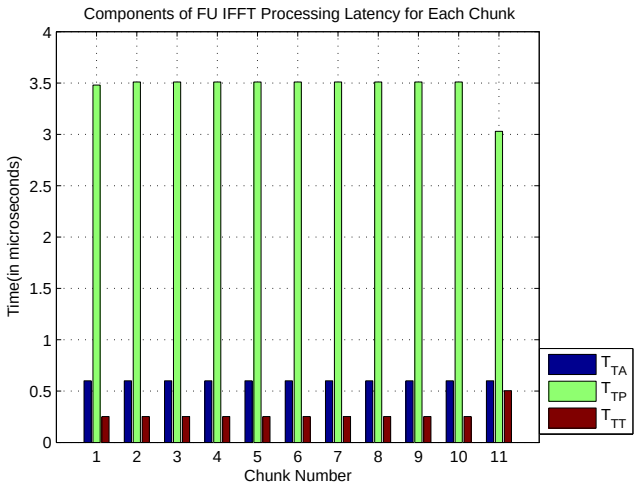

Figure 4.17: Processing Latencies in FU IFFT: Fixed Frame Size

(b) PE SYNC Rx Processing

The FU SYNC is combination of three different PEs, PE Sync-Rx-FFT, PE Demod and PE Checker.(The PE Sync-Rx-FFT is also divided into two parts, synchronizer and Rx-FFT. But these two are not considered separate PEs.) The Sync-Rx-FFT, DMOD and CHKR have been combined because of the resource limitations on the FPGA. If they were separate, each PE would need a separate UCM. Since each UCM requires 7% of the Virtex-5 SX-95 FPGA resources, the combined utilization of three UCMs (one each for Sync-Rx-FFT, DMOD and CHKR) would be 21%. The combined resource requirement of the three PEs is about 41%. Summing up these numbers, we get that the resource consumption of the these three FUs is 62%. Adding to that the utilization of MAC Rx FU, the number becomes close to 70%. This leaves only 30% space on the FPGA for routing the connections among the blocks which is not enough. Since we cannot compromise on the PE logic, we combined the PEs and assigned the combination a single UCM. This saved 14% of the resources and we were able to fit the design on the FPGA.

The Figure 4.18 shows the task flow inside the combined block.

The Sync-Rx-FFT is constantly scanning the GPI port for preamble. When it detects the preamble, it sends the received PCLP header to PE DMOD for

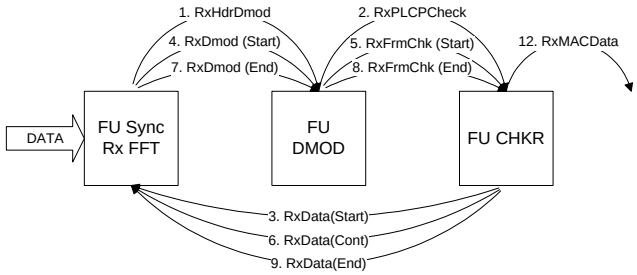

Figure 4.18: The Task Flow inside the FU SYNC block

demodulation. The demodulated data is then sent for Checksum calculation in FU CHKR. If the checksum passes, the CHKR PE demands for the next data by sending the RxData command to the FU Sync. The amount of data demanded depends on the chunk size. Currently, it has been configured to 4 OFDM symbols per chunk which is the same as that in the transmitter. The PE CHKR knows the frame size from the PLCP header. Based on the frame size and the chunk size it calculates the number of chunks in the frame. When the Sync receives the 'RxData' command, it accordingly sends 4 OFDM symbols to the PE DMOD. PE DMOD then demodulates the data and sends it to the PE CHKR for checksum. The PE Chkr then asks for the next chunk and the cycle repeats until the last chunk has been processed by the Chkr. The Chkr then sends the processed frame to MAC Rx.

(c) UCM TT Processing

For each chunk, the UCM transfers the data in the Output Buffer to Input Buffer of MAC Rx. The task descriptor for 'RxMACData' task is only written for the last chunk.

The size of each chunk going to the input of FU MAC Rx is not the same as size of corresponding chunk at the input of PE HDR. The difference between the chunk sizes at the input of PE HDR and those at the output of FU SYNC Rx (or input to FU MAC Rx) can be explained as follows. We have seen that in PE HDR, an 8-Byte long PLCP header is appended at the beginning of the first chunk of the frame. Since, the size of first chunk is 34 Bytes, the output of the first chunk in PE HDR is thus 42 Byte

long. There are two parts in the PLCP header. The 6-Byte information from 'TxVector' is coded with BPSK modulation while the 2-Byte Service Bits field is coded with QPSK modulation. Unlike the PE HDR however, the Sync-Rx-FFT PE does not count the PLCP Header as part of the first chunk. The 'TxVector' is processed before processing the first chunk. The two service bytes are processed with the first chunk but are not sent over. The chunk size currently is set at 48 Bytes. Not counting the service bytes, this means that the Sync should send over 46 Bytes of MAC data in the first chunk. But the current implementation only allows the FU SYNC to write integer number of 32-bit words. Hence, the Sync stores the last two bytes and sends over the first 44 Bytes to DMOD-CHKR. So, the chunking Equation 4.1 becomes now,

$$500 = 44 + (9 * 48) + 24 \tag{4.2}$$

Note the difference in the size of last chunk; it is reduced by 10. Hence, if the last chunk contains less than or equal to 10 bytes in Equation 4.1, the receiver will process one less chunk (but same amount of data) according to Equation 4.2.

Hence, as Equation 4.2 shows, the size of all chunks, except the first and last, is 48 Bytes. The size of the first chunk is 44 Bytes and the size of the last chunk will depend on the frame size. For 500 Byte frame, the last chunk size is 24 Bytes.

(d) Processing Latencies

The processing latencies in the Sync-rx-FFT and in the dmod-chkr are as shown in Table 4.3 and Figure 4.19.

We see here that the processing latencies in the various PEs in FU SYNC dominate the FU Latencies rather than T_{TT} as has been the case in FU MAC Tx, FU HDR and FU MOD. The reason is that since we are using only one UCM for three PEs, the data transfers which would have occurred in the

	μs
Sync Preamble and PLCP Header Processing Time	33.65
Rx FFT Preamble and PLCP Header Processing Time	4.81
Dmod PLCP Header processing time	2.2
Chkr PLCP Header processing time	0.66

Table 4.3: FU Sync Rx Preamble Processing Measurements

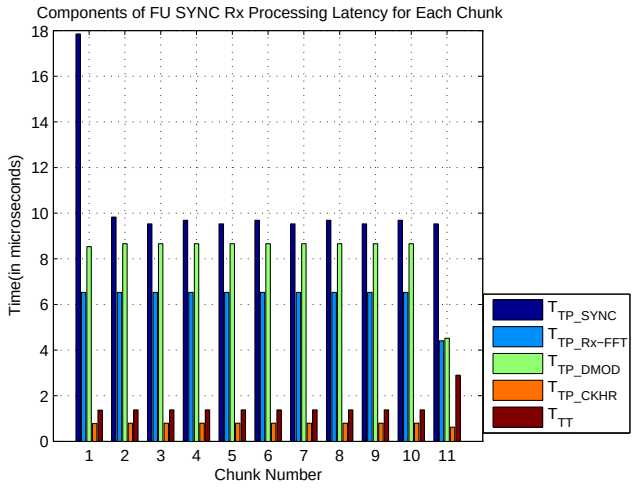

Figure 4.19: Processing Latencies in FU Sync Rx: Fixed Frame Size

intermediate T_{TT} stages are happening internally. The only data transfer that happens through the UCM TT block, is the frame transfer to FU MAC Rx for which the Figure 4.19 shows the processing latency.

4.2.2.6 FU MAC Rx Analysis

(a) UCM TA Processing

The TA stage includes fetching the task descriptor from the task queue and updating 'InDataPointer' and 'InDataSize' for two buffer regions.

(b) PE MAC Rx Processing

The MAC Rx is the last FU in the receiver chain. The last section explained that the Sync-Rx-FFT-DMOD-CHKR FU transfers the data in chunks to FU MAC Rx, the RxMACData task is sent only for the last chunk. Hence, effectively the MAC Rx waits for the complete frame before starting to process it. The MAC Rx PE checks the frame type, intended destination and

the checksum. If the incoming frame is an error-free, type DATA frame intended for itself, the PE generates the task 'SendAckFrm' task for the MAC Tx. It sends the source address in the data frame header to MAC Tx along with this task.

(c) UCM TT Processing

The software reads the data frame from the MAC Rx OBUF, so there is no data frame transfer involved in the MAC Rx TT stage. But, the address of the sender (source) of the incoming frame is sent to FU MAC Tx for transmitting the ACK frame, along with the descriptor of that task.

(d) Processing Latencies

The Table 4.4 shows the processing latencies for this FU. We see that the T_{TT} latency is small because of the absence of data transfer in the TT stage.

T_{TA}	T_{TP}	T_{TT}	T_{UCM}	$T_{FU\_MAC\_Rx}$
μs	μs	μs	μs	μs
0.64	4.84	1.84	2.48	7.32

Table 4.4: FU MAC Rx Measurements: Fixed Frame Size

4.2.3 ACK Frame Transfer

The Figure 4.20 shows the task flow for the Ack frame. We see that the task flow is the same as that for the data frame. The only difference is that since Ack frame is only 14 bytes long, it fits into a single chunk. Since the operations are the same, we do not go into details of operational latencies.

4.2.4 Summary

We now summarize the processing latencies for all FUs and calculate the UCM overhead and PE processing delay for each FU. The results are as shown in the Table 4.5.

The T_{TA} and T_{TT} values shown in Table 4.5 are combined values for all chunks in the case of HDR Tx, MOD Tx and IFFT Tx FUs. We plot the per chunk

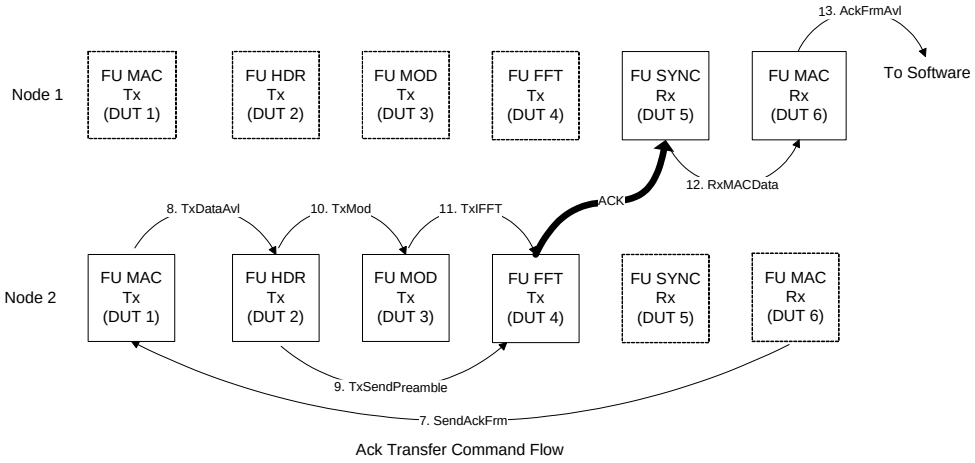

Figure 4.20: Task Flow for ACK Frame

FU	T_{TA} μs	T_{TP} μs	T_{TT} μs	T_{UCM} μs	T_{Total} μs	T_{UCM} %	T_{PE} %
MAC Tx	0.48	4.08	5.65	6.13	10.21	60.039	39.961
HDR Tx	7.09	12.02	30.47	37.56	49.58	75.756	24.244
MOD Tx	6.16	71.68	78.33	84.49	156.17	54.101	45.899
FFT Tx	6.6	38.1	3.024	9.624	47.724	20.166	79.834
SYNC Rx	-	283.39	16.7	16.7	300.09	5.565	94.435
MAC Rx	0.64	4.84	1.84	2.48	7.32	33.880	66.120

Table 4.5: FU Measurements: Fixed Frame Size

T_{TA} and T_{TA} values for these FUs along with T_{TA} and T_{TT} values for MAC FUs in Figure 4.21. We do not include the FU Sync Rx in the analysis since the TA stage is part of the configuration stage.

We see that the per chunk values for the TA stage are almost same across all FUs. This shows that for current implementation, the TA operations are very similar for all PEs. The same is however not true for the T_{TT} values. The T_{TT} values largely depend on the size of the output data being transferred. Also, we again see that except for Tx IFFT FU, the T_{TT} latency is 2 to 7 times greater than the T_{TA} latency. We thus analyze the T_{TT} further.

As seen in Section 3, the TT operations involve data transfer along with some control operations. We now measure the split-up between the control overhead and the data transfer latency in T_{TT} for three FUs which involve data transfer in the TT stage namely FU MAC Tx, FU HDR Tx and FU MOD Tx. The TT

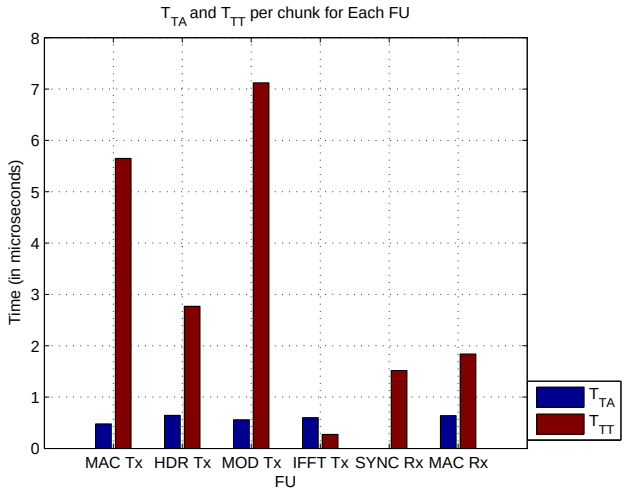

Figure 4.21: UCM Latencies per chunk: Fixed Frame Size

operations in these FUs are identical in nature except the data size. Hence we will be able to study the effect of data size on the T_{TT}.

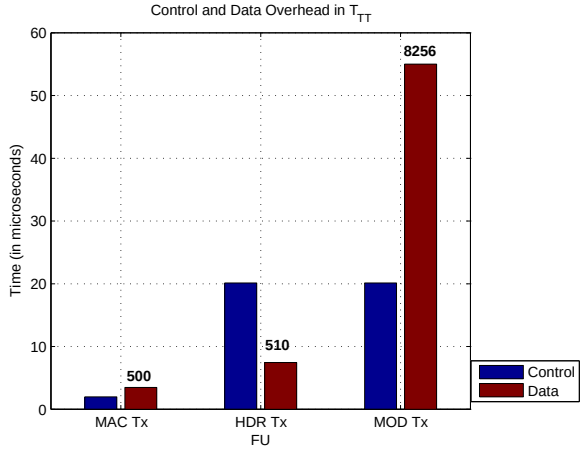

Figure 4.22: Control and Data Overhead in T_{TT}

The Figure 4.22 shows the split up between data and control for these FUs. We also show the size in bytes of the data transfer for each FU above the bars.

We see that though FU MAC Tx and FU HDR Tx generate very similar amount of data at the output, the T_{TT} control overhead is much larger for FU HDR. This is because FU HDR processes and transfers the data in chunks. For each data chunk transfer, there is additional control overhead in TT stage. We see that the control overhead for the FU HDR is same as that of FU MOD Tx which can be

attributed to the fact that they both process and transfer data in same number of chunks. But for the same number of chunks, the data latency is more than 5 times greater for FU MOD. This is because of the large amount of data FU MOD transfers to FU IFFT. We can conclude here that transferring data in smaller chunks is inefficient since it adds up control overhead. But since the chunk size is fixed by modulation, there is no scope for improvement here.

In the next experiment we analyze the effect of change in frame size over the performance of WiNC2R.

4.3 Effect of Frame Size on WiNC2R Performance

4.3.1 Experimental Setup and Measurements

The experimental setup is the same as that for the previous one. The two WiNC2R BFM nodes are connected as shown in the Figure 4.6. The data transfer taking place is also similar to previous experiment. The nodes use ALOHA MAC protocol to communicate. Node 1 sends the data frame and waits for the ACK from node 2. After receiving the ACK, it sends the next data frame. There is no ACK-TIEMOUT period; instead node 1 keeps waiting for the ACK before sending the next frame. Since, the channel between the nodes is perfect there are no packet drops and hence there is no danger of Node 1 waiting indefinitely for the ACK.

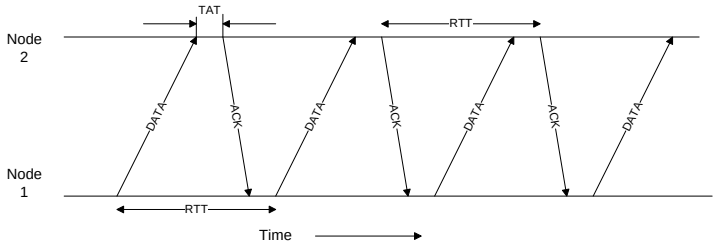

Figure 4.23: Data Transfer using simple ALOHA MAC Protocol

The ALOHA MAC communication is as shown in Figure 4.23. Two parameters Turn-Around-Time (TAT) and Round Trip Time (RTT) are also depicted in the figure. TAT is defined as time taken by the node to turn around from being a receiver to being a transmitter. RTT is defined as time difference between any two data frames or ACK frames. For this experiment, the MAC input frame size is varied from 500 bytes up to 1500 bytes in the steps of 250 bytes. For each frame RTT was measured and throughput was calculated based on the equation 4.3.

$$Throughput = MAC\_Frame\_Size/RTT \tag{4.3}$$

The equation 4.3 measures the MAC-level throughput. The modulation scheme was fixed as QPSK providing the PHY data rate of 12 Mbps according to 802.11a standard. Based on the modulation scheme, the FirstChunkSize was fixed at 34 Bytes

and ChunkSize was fixed at 48 Bytes at the input of PE HDR Tx. At the output of the FU SYNC Rx, these sizes are 44 Bytes and 48 Bytes respectively. The Rescheduling Period was fixed at 2560 clock cycles which translates to 25.6 μs at the current operating frequency of 100 MHz.

Figure 4.24 shows the effect of change in frame size on RTT. We see that though RTT increase linearly with the increase in frame size, the relationship between them is not one-to-one. The green line in the graph shows the frame size to illustrate a one-to-one relationship. We see that as frame size increase the RTT keeps going farther away from this line.

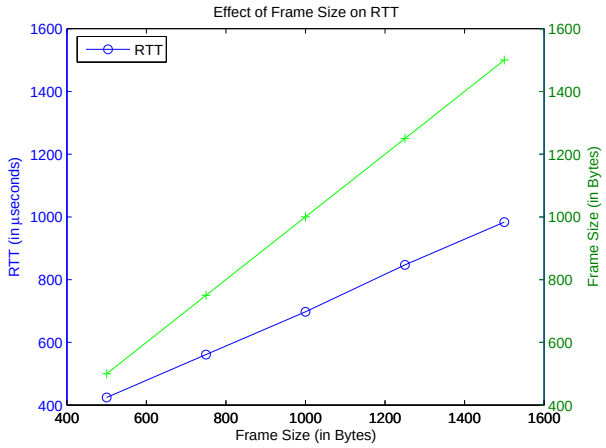

Figure 4.24: RTT Measurement: Varying Frame Size

Figure 4.25 shows the effect on throughput of change in frame size. We see that throughput increases at first but begins to flatten out as the frame size keeps increasing.

The reason the throughput is increasing with the MAC frame size is, that as the MAC frame size increases the percentage time consumed by the frame overhead bits like PLCP Header and PLCP Preamble reduces.

4.3.2 Analysis of TAT and RTT

Figure 4.26 shows how the RTT and TAT were measured.

We can here elaborate more on previous definitions of RTT and TAT. For measurement purposes, TAT was measured at the node 2 GPIO interface as shown in figure. It is defined as time difference between the end of data frame at GPI and the start

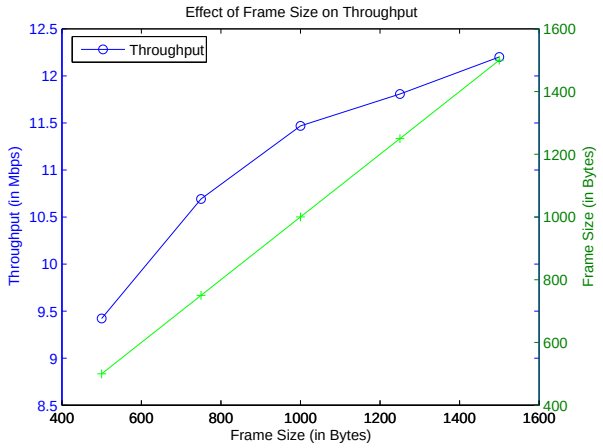

Figure 4.25: Throughput Measurement: Varying Frame Size

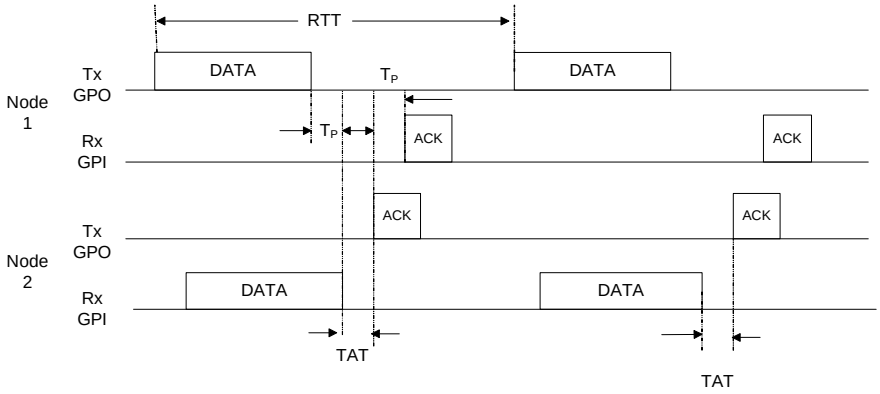

Figure 4.26: Measurement of TAT and RTT at GPIO Interfaces of Nodes

of ACK frame at the GPO interface. The T_P is propagation delay between Tx and Rx. In the simulation environment it is fixed and very small (about 27 clock cycles at clock frequency of 100 MHz). *RTT* is measured at Node 1 interfaces. It is measured as time difference between two consecutive data frames.Since the focus of these experiments is to evaluate the performance of WiNC2R architecture, we have ignored channel propagation delays. Also, we assume a perfect channel with no packet drops. In short we envision a scenario where the performance of WiNC2R is only hindered by its own architecture. In other terms, the performance is affected by the factors inside the radio rather than outside like channel quality and propagation delay. Since we are performing architectural evaluation, we can say that these assumptions are valid for given test cases.

4.3.2.1 TAT Analysis

TAT dictates the amount of time node 1 is willing to wait for an acknowledgment before retransmission. In many systems the ACK timeout is the most stringent timing constraint to be satisfied; e.g. in 802.11 MAC, the ACK timeout denoted by SIFS (Short Inter-Frame Spacing) is the has the smallest value for a time constraint for any PHY layer defined. For 802.11a PHY, it is 16 μs. It is thus important to see how close WiNC2R comes to satisfying this criterion.

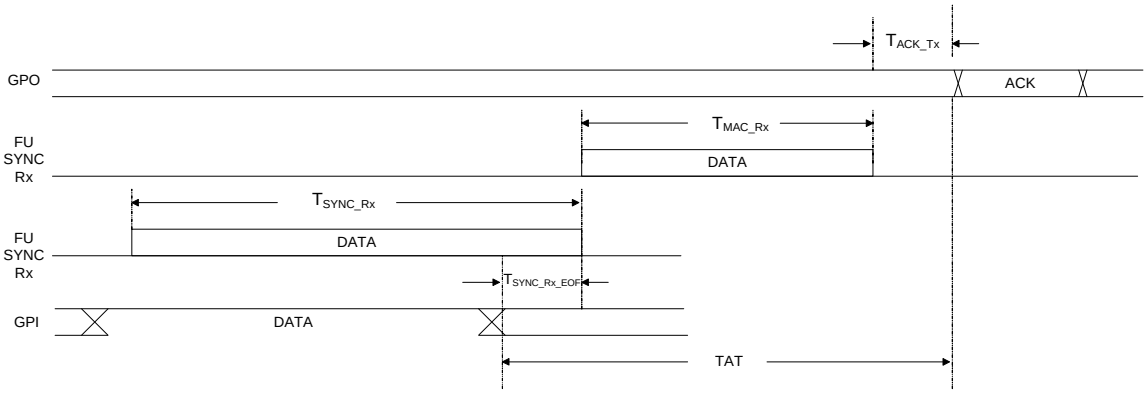

Figure 4.27: Components of TAT

Figure 4.27 shows more details about TAT. We focus on the node 2 GPIO interface and the FUs in the receive chain working behind it. The transmit chain FUs have been bunched together.

As shown in the Figure 4.27, TAT is composed of time required for data frame processing in the receive chain of the node and time required by the first byte of ACK to travel through the transmit chain to reach the GPO interface. The data frame processing in the receive chain can be divided as processing in FU Sync Rx and FU MAC Rx. T_{SYNC_Rx} is used to denoted time required by FU Sync Rx to process the incoming data frame. T_{MAC_Rx} denotes the same for FU MAC Rx and T_{ACK_Tx} denotes the time required by the whole transmit chain to output first byte of ACK frame on the GPO interface.

However, TAT will not include the complete processing time in FU SYNC Rx. The definition of TAT states that it starts only after the end of frame in the receiver. Hence, only the part of processing time of the FU SYNC which occurs after the end of frame

on GPI is included in TAT. We call this time processing time in FU SYNC Rx after End of Frame (EOF), $T_{SYNC\_Rx\_EOF}$. Our equation for TAT is thus,

$$TAT = T_{SYNC\_Rx\_EOF} + T_{MAC\_Rx} + T_{ACK\_Tx} \tag{4.4}$$

Following are the results of our simulations which measured these three components.

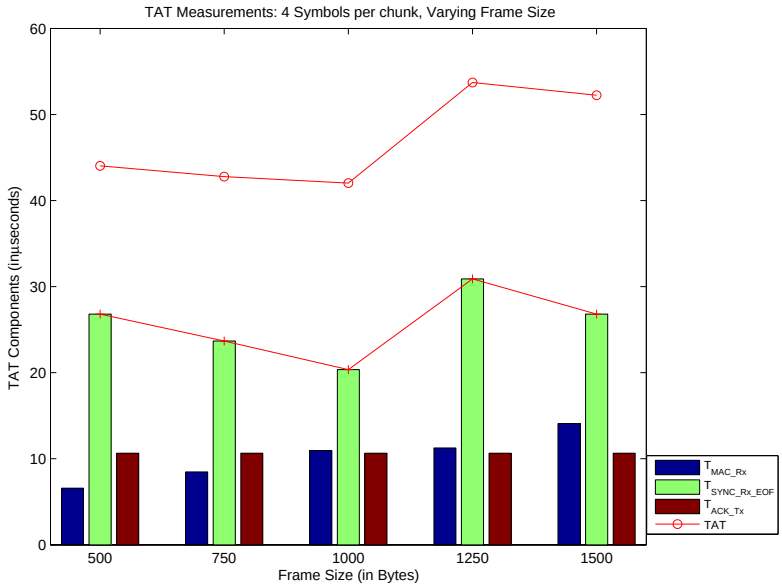

Figure 4.28: TAT Measurement: Varying Frame Size

From the Figure 4.28, it is evident that T_{ACK_Tx} is constant regardless of the data frame size. It is to be noted that irrespective of the data frame size, the ACK frame size is always constant at fourteen bytes. Also, the ACK frame is a single chunk frame and hence goes through the simplest processing the transmit chain could offer. Since, the improvement in individual PEs is out of scope of this thesis, we conclude that there is no scope for improvement in this component.

Referring again to Figure 4.28, we see that TAT closely follows the changes in the $T_{SYNC\_Rx\_EOF}$ component. This component includes the processing latency for the last chunk. In the current setting, four OFDM symbols are processed in each chunk. The last chunk however might not always contain four number of OFDM symbols, since it has the remaining data from the frame. The size of data in the last chunk depends on the frame size and the chunk size. For the current setting, the sizes for the last chunk

vary with the frame size as shown in Table 4.6.

Frame Size (Bytes)	Number of Chunks	First Chunk Size(Bytes)	Chunk Size Size(Bytes)	Last Chunk Size(Bytes)
500	11	44	48	24
750	16	44	48	34
1000	21	44	48	44
1250	27	44	48	6
1500	32	44	48	16

Table 4.6: Chunk Sizes with 4 Symbols per chunk

From Table 4.6 and measurements in Figure 4.28, it can be seen that there is inverse relationship between the last chunk size and $T_{SYNC\_Rx\_EOF}$. Figure 4.29 shows the start of last chunk processing with respect to the end of frame for all frame sizes. For convenience, the time axis starts from the instant the second-to-last chunk processing for the 500 Byte frame starts. Hence, for all frames we see chunk number 11 onwards. Note that we have also counted Preamble and PLCP Header Processing as a chunk in Figure 4.29, though technically it is not so.

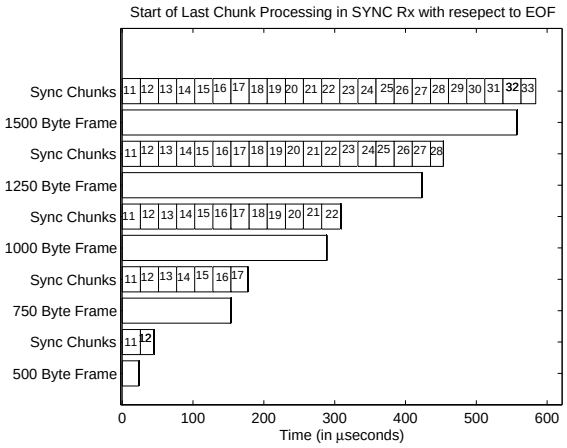

Figure 4.29: Start of Last Chunk Processing w.r.t End of Frame

It is seen that as frame size changes from 500 to 1000, the processing for the last chunk starts earlier. In case of 1250 byte frame however, it starts much later and for 1500 byte frame the starting location is slightly better than the that for 1250 byte frame.

This apparent delay in starting last chunk processing is not because of any factor in

FU SYNC, but the length of the frame. As the last chunk becomes smaller, the frame will earlier on the GPI interface. This is depicted in Figure 4.29. For the 1250 byte frame, which has the smallest last chunk at 6 bytes, the frame ends before the processing in the second-to-last chunk, chunk number 27, has not even finished. The 1000 byte frame, having the largest amount of data in the last chunk at 44 bytes, finishes after the last chunk processing has started. Since we measure the TAT from the end of the frame, the apparent value of the TAT seems larger in case of 1250 byte frame than in the 1000 byte frame.

Referring back to Figure 4.28, the MAC Rx latency, denoted by T_{MAC_Rx}, seems to vary almost linearly with the change in data frame size. We give the details of MAC Rx latency in Table 4.7.

Frame Size	T_{TA}	T_{TP}	T_{TT}	T_{MAC_Rx}
Bytes	μs	μs	μs	μs
500	0.64	4.2	1.84	6.68
750	0.64	6.09	1.84	8.57
1000	0.64	7.95	1.84	10.43
1250	0.64	9.84	1.84	12.32
1500	0.64	11.7	1.84	14.18

Table 4.7: T_{MAC_Rx} Measurements: Varying Frame Size

We see that the TA latency is constant irrespective of the frame size. We saw in Section 3 that the TA operations have nothing to do with the actual data, they are control operations. This explains the independence of TA latency with respect to frame result. Since, the MAC Rx is the last block in the receiver chain, there is no data transfer in MAC Rx TT. Instead, the software in the soft-core CPU reads the frame. Hence, the TT latency is also independent of data frame size. The TP delay increases linearly with frame size. This is also expected since the MAC Rx PE operates on the complete frame.

4.3.2.2 TAT Improvement

1. Improvement in $T_{SYNC\_Rx\_EOF}$

   From Figure 4.28, it is clear that $T_{SYNC\_Rx\_EOF}$ is the largest component in

TAT. The measurements of $T_{SYNC\_Rx\_EOF}$ for different frame size indicate a range between 20 μs to 30 μs. For the 802.11a standard, the SIFS time period is 16 μs. We see that improvement in the performance is a necessity for TAT to conform to SIFS measure. Though we do not deal with improvement inside the PEs, we will try and see if there is any configuration setting which can improve the performance of FU SYNC Rx.

It was seen in TAT analysis that $T_{SYNC\_Rx\_EOF}$ depends upon the processing latency for the last chunk. In Figure 4.29, we saw that it might also depend on the processing latency of the second-to-last chunk as well. The processing latency for each chunk will in turn depend upon the efficiency of the algorithm in FU SYNC Rx and the size of each chunk. We do not deal with improvements in the SYNC algorithm but we study the effect of change in chunk size on the performance.

In the previous experiment, we saw the reason the chunk size in the transmitter was set at 4 OFDM symbols. In the receiver no analysis has been done to decide the most effective chunk size. The default setting was set at 4 just like the transmitter. Since $T_{SYNC\_Rx\_EOF}$ depends on chunk size, we decided to halve the chunk size to 2 OFDM symbols per chunk and see the effect on TAT and its components.

Figure 4.30 shows the effect of changing the frame size on TAT and its components with 2 OFDM symbols per chunk. We also show the TAT and $T_{SYNC\_Rx\_EOF}$ values for the chunk size equal to 4 OFDM symbols, in blue. We notice that $T_{SYNC\_Rx\_EOF}$ s now reduced to be always less than 20 μs. The TAT values are also below 40 μs for all frame sizes, though they are still at least double the SIFS value. It is also noticed that the gain in performance is not the same for all frame sizes.

In Table 4.8, the last chunk sizes are listed with chunk size set to 4 OFDM symbols and 2 OFDM symbols. It is seen that due to change in the chunk size, the distribution of last chunk size is changed. From Table 4.8 and Figure 4.30, we

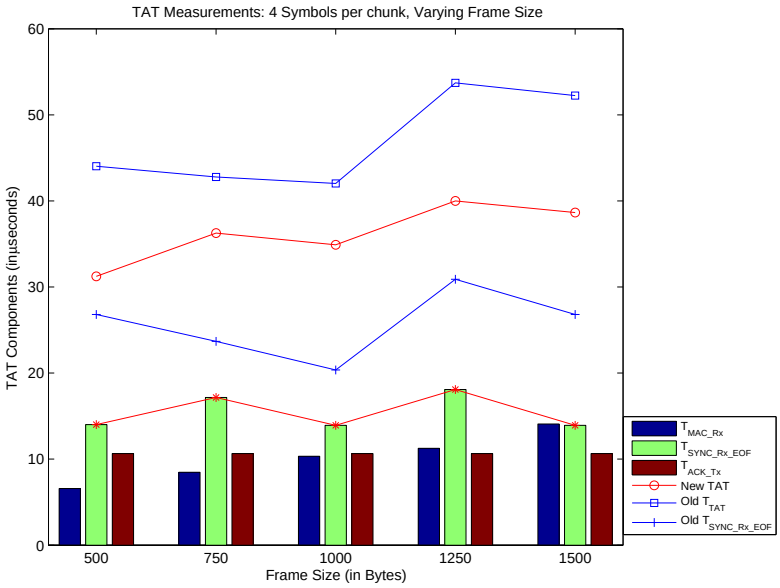

Figure 4.30: TAT Measurements: 2 Symbols Per Chunk, Varying Frame Size

Frame Size	4 Sym per chunk		2 Sym per chunk	
	No. of Chunks	Last Chunk Size	No. of Chunks	Last Chunk Size
Bytes		Bytes		Bytes
500	11	24	21	24
750	16	34	32	10
1000	21	44	42	20
1250	27	6	53	6
1500	32	16	63	16

Table 4.8: Last Chunk Size Table

see that $T_{SYNC\_Rx\_EOF}$ values are still correlate, in the same manner as before, with the last chunk size. That is, $T_{SYNC\_Rx\_EOF}$ decreases as the last chunk size increases.

2. Improvement in T_{MAC_Rx}

   In the TAT analysis, it was seen that the processing latency in the FU MAC Rx depends largely on the T_{TP}. This task processing time in turn depends upon the frame size as was discussed in the basic experiment's analysis. We have already demonstrated how the processing latency can be made independent of the frame size (at least to some degree) in the FU SYNC Rx by the process of chunking. For the initial development, to keep the FU MAC Rx design as simple as possible, we had limited chunking to the FU SYNC. But based on the performance, it is

now proposed that the chunking be extended to FU MAC Rx as well.

Figure 4.31 shows the interaction between FU SYNC Rx and FU MAC Rx in the current implementation. The SYNC Rx transfers the data in chunks to MAC Rx and when all the chunks have been transferred, SYNC triggers the MAC Rx by sending RxMACData task to it.

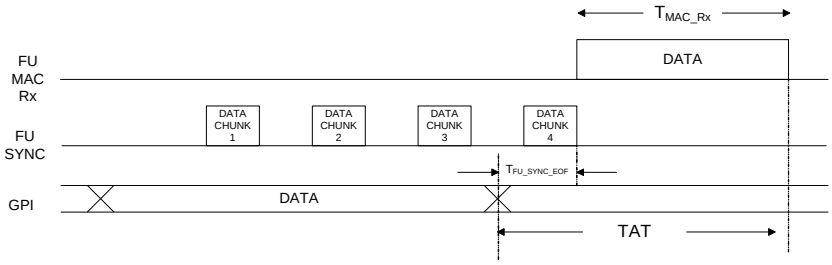

Figure 4.31: TAT with chunking only in FU SYNC

Figure 4.32 shows the interaction between FU SYNC Rx and FU MAC Rx with the chunking extended in MAC Rx. Here also SYNC rx transfers the data in chunks to MAC Rx, but along with the chunks it also activates it with sending the task. As a result FU MAC Rx processes the data in chunks. From the processing latency tables seen in the basic experiment, we can see that compared to FU SYNC, FU MAC rx latencies are small. There will be no situation where the FU SYNC will have to wait until the FU MAC finishes processing. Hence, after the end of frame, only last chunk will need to be processed. Thus, the FU MAC Rx latency contributing to TAT will effectively depend on last chunk size.

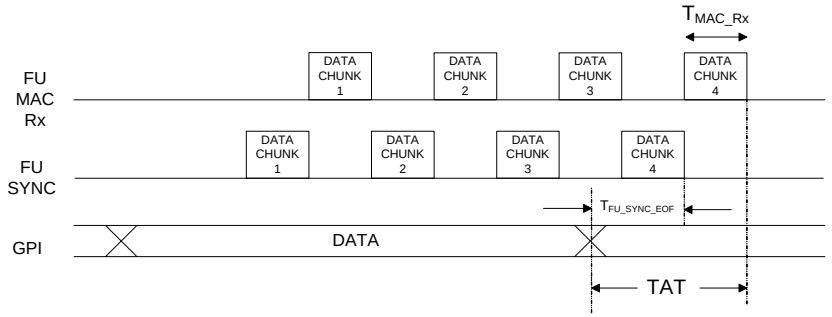

Figure 4.32: TAT with chunking extended to FU MAC Rx

4.3.2.3 RTT Analysis

The Round Trip Time (RTT) is defined as time duration between two successful consecutive data transmissions. We define a successful data transmission as shown in fig. i.e. the frame is received correctly at the receiver of node 2 and ACK is transmitted by node 2 transmitter. This ACK is also received correctly at Node 1 and is interpreted as an OK to send the next frame. The details of RTT are shown in Figure 4.33.

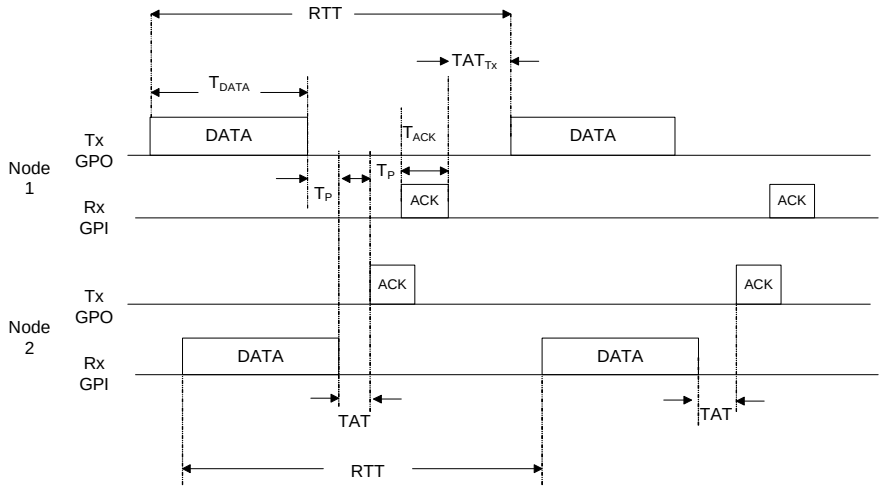

Figure 4.33: Components of RTT

Referring to Figure 4.33 we can have the following equation for RTT. The various terms used in the equation are also explained below.

$$RTT = T_{DATA} + T_P + TAT + T_P + T_{ACK} + TAT_{Tx} + T_{DATA\_Tx} \tag{4.5}$$

Figure 4.34 shows the results from our simulation for all the above parameters.

The results can be explained as follows. T_{DATA} and T_{ACK} only depend upon the respective frame lengths. The data frame size increases linearly from 500 to 1500 but ACK frame size remains constant at 14 bytes. Hence, it can be concluded that T_{DATA} rises linearly with increase in frame size while T_{ACK} remains constant. Thus, The RTT variation with frame size is chiefly because of TAT and T_{DATA_Tx}. The TAT has already been discussed, we thus focus on T_{DATA_Tx}.

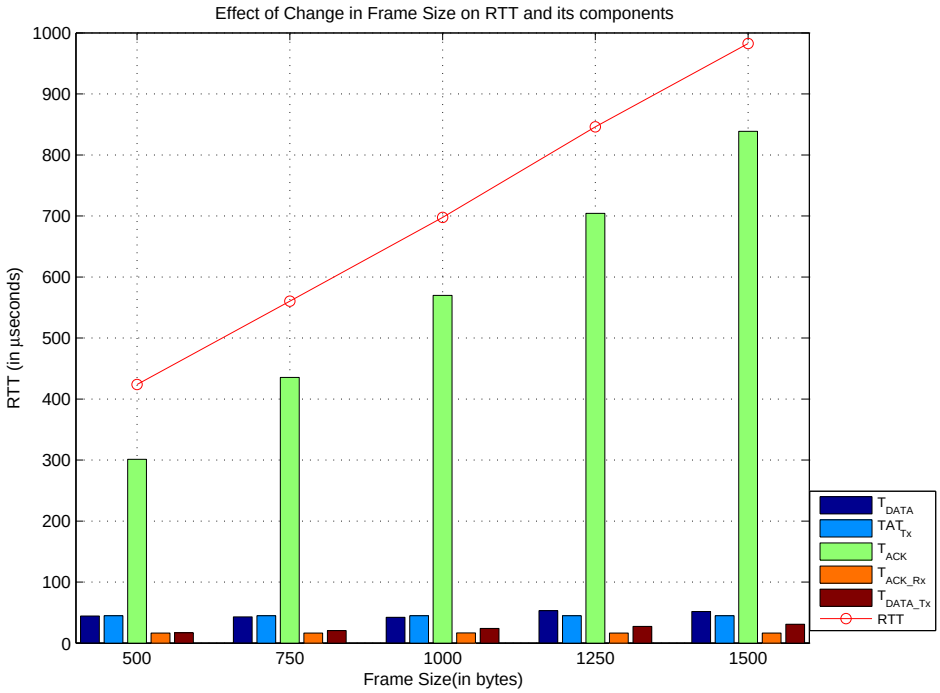

Figure 4.34: RTT Measurements: 4 Symbols Per Chunk, Varying Frame Size

4.3.2.4 Improvement in T_{DATA_Tx}

The two main components of T_{DATA_Tx} are described in Figure 4.35. These are frame processing time in MAC Tx FU and frame processing time in the rest of the transmit chain FUs. The MAC Tx processes the complete frame and sends it to FU HDR. The FU HDR however divides the frame into chunks and processes each chunk separately one after the other. This is done in order to save memory and to introduce parallel processing in the WiNC2R. The number of chunks and size of each chunk depends on the MAC frame size and modulation scheme/bit rate chosen for transmission.

The T_{DATA_Tx} is defined as time taken by the first byte of the data frame to reach FFT GPO interface through the whole transmit chain. Going by the representation of the process shown in Figure 4.35, it is seen that this time will vary directly with frame size. The processing time in MAC Tx FU, denoted by T_{MAC_Tx}, increase proportionally with the increase in frame size. After the MAC Tx, the data frame is divided into chunks. For T_{DATA_Tx} analysis we are only concerned with processing time of the first chunk. The size of this chunk is constant regardless of the MAC frame size. The size of

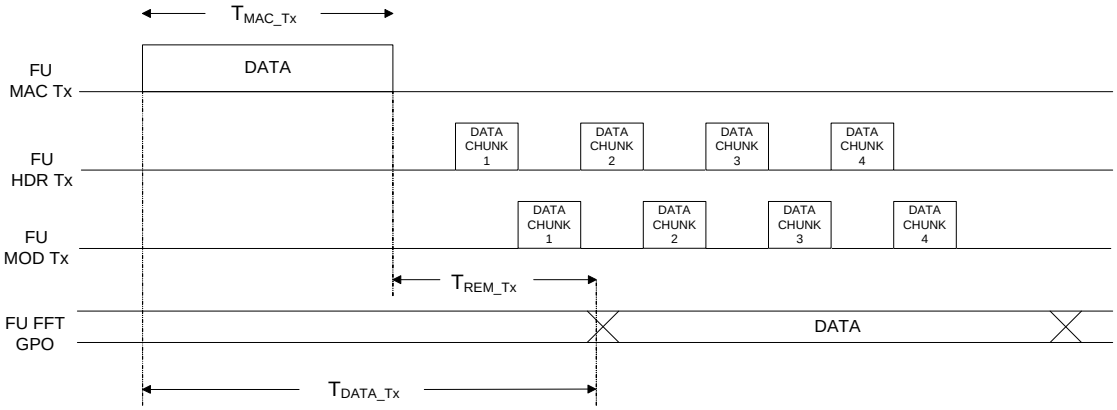

Figure 4.35: T_{DATA_Tx} with chunking being done in FU HDR

first chunk has been fixed by the modulation scheme. Hence, we can say that processing time for the first chunk in the transmit chain, excluding MAC Tx, is constant. This time is denoted by T_{REM_Tx}. The equation for the T_{DATA_Tx} is

$$T_{DATA\_Tx} = T_{MAC\_Tx} + T_{REM\_Tx} \tag{4.6}$$

Since, T_{REM_Tx} is constant, T_{DATA_Tx} changes with T_{MAC_Tx} which in turn changes linearly with frame size. The following scheme is proposed to remove this dependency.

Just as in the case of MAC Rx processing in TAT analysis, it is observed that MAC Tx does not necessarily need to wait until the complete frame has been processed to send the first byte to FU HDR. However, sending a single byte or word over the bus does not make sense. So, we propose that the chunking process, which is currently being done in FU HDR, be moved to FU MAC Tx.

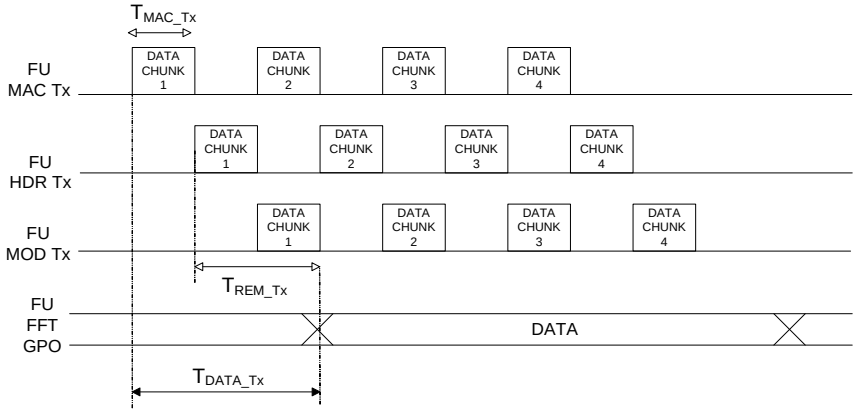

Figure 4.36: T_{DATA_Tx} with chunking being done in FU MAC Tx

Figure 4.36 shows the possible output of this change. Much less time will be required to process a single chunk than that to process a complete frame in MAC Tx FU. Hence, total T_{DATA_Tx} will be reduced. The actual amount of reduction depends on the size chosen for the first chunk. Nonetheless, this technique will help reduce the RTT. But there are difficulties in the implementation. The size of first and subsequent chunk is determined by the modulation scheme. When a chunking task is input to UCM, it looks at these sizes to manipulate the size of input data to the PE. It does to change the actual data. Rather it just modifies the values of the $inputdatalocation, inputdatasize$ tuple passed to PE. For example, when using QPSK modulation scheme, the first chunk size is fixed at 34 bytes while the size of subsequent chunks is fixed at 48 bytes. Assume that a frame 100 bytes long is written starting from location LOC in the input buffer of the PE HDR. For this frame, UCM will send 3 commands, each time with different tuple value namely, $LOC, 34$, $LOC + 34, 48$, and $LOC + 34 + 48, 18$. PE will read corresponding bytes of data and process it.

The problem with doing chunking in MAC Tx is that not all data which is input to MAC Tx is present in input buffer. Some information like MAC header resides in the RMAP of MAC Tx. This header information is attached at the beginning of the frame. The UCM does no have any idea about the RMAPs in the PE. Thus, to any first chunk size given by UCM, MAC Tx is going to add number of bytes equal to size of header. Hence the calculation of first chunk size becomes difficult. It can be said that the first chunk size is only equal to the real value minus the length of the header. For example, again for QPSK, real first chunk size is 34 bytes. If we assume MAC data frame header to be 24 bytes, then the apparent first chunk size for MAC Tx is $34 - 24 = 10$. But this will not work in cases where the real first chunk size is less than the frame header size. For example, in BPSK, the first chunk size is only 16. Also, the length of the MAC frame header might not always be the same even for the same frame type. For example, in 802. 11 MAC the size of MAC header for data frame changes according to ToDS and FromDS bits.

A tentative solution for this problem can be to move the header information from RMAP to Input Buffer. This means the processor will have to write both the header

information and the payload to the input buffer before sending command to the MAC Tx. Due to time constraints, we suggest this as future work.

4.4 Supporting Multiple Flows by Time-Sharing of UCM

4.4.1 Motivation and Experimental Setup

Supporting multiple technologies which might exist in the network at the same time is an important feature for a CR platform. Even more important is the ability to support all these flows at the same time without duplicating the hardware as far as possible. This requires effective time-sharing mechanism in the layer above the hardware and per-flow re-configurability support in the underlying hardware. In this experiment, we demonstrate the ability of current WiNC2R platform to support multiple flows. Since the current implementation supports only 802.11a-like OFDM flow, we use multiple instances of the same OFDM flow.

A secondary motivation behind this experiment is to improve the UCM resource utilization. The analysis in the basic experiment showed us that the FU HDR UCM is idle for almost 80% of the time. We exploit this idle time to multiplex more OFDM flows through the WiNC2R transmit chain. Figure 4.15 showed the comaprison between the T_{IDLE} and T_{FU_HDR}.

We study the effect of adding more flows to the one used in previous experiments. Specifically, we study the change in the FU HDR UCM idle time and the number of flows at which the first flow will break down.

We use BFM tool-set, described in Section 4.1, to create a test-bench using just the WiNC2R transmit chain. To support the addition of more flows, we create instance of the FU IFFT Tx for each added flow. All the flows will be multiplexed throughout the first three FUs, but will split at the output of FU MOD. The duplication of FU Tx IFFT is necessary because the current PE IFFT implementation supports only one OFDM flow at a time. The IFFT output is corrupted if there is an attempt to route more than one flow through it at the same time.

Figure 4.37, shows the task flow setup for two OFDM flows. As shown, a second set of tasks needs to be created to support the additional flow. This is because with each task there is an input data location, input data size set associated. For both the flows, this set has to be different otherwise the data will get overwritten. We create a

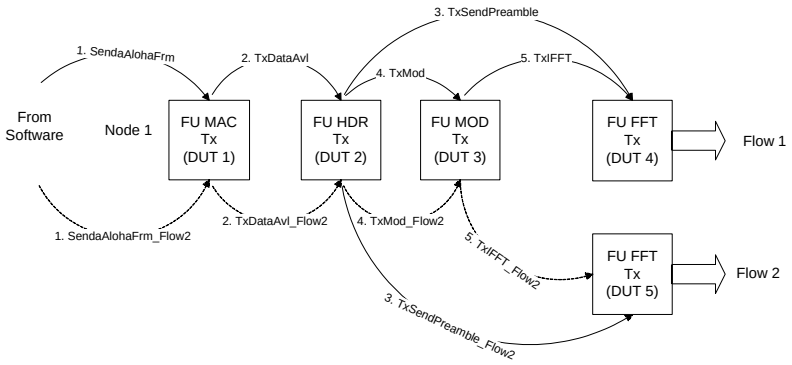

Figure 4.37: Setup for Two Concurrent Data Flows through WiNC2R

duplicate of each task in the first flow until the FU IFFT. All the information in these tasks is the same as that for tasks in first flow, except the input data location and size.

4.4.2 Measurements and Analysis

To consider the effect of additional flows, we first need to consider some details about the working of the PE IFFT block. The PE IFFT block converts the incoming chunked data to a continuous OFDM stream at the output. To achieve this, PE IFFT stores the incoming chunks into a FIFO, which it reads at constant rate. To maintain the continuity in the output stream, once the frame starts, the FIFO has to have some data in it. It cannot be empty. The purpose behind the rescheduling period is to prevent overwriting of FIFO for low system latency cases. The rate at which the data is written into the FIFO depends on the system latency from the input of FU HDR to input of FI IFFT. If the system latency is too large, the FIFO will become empty and the OFDM stream will break. We consider this point as the point where our system breaks down. We denote the system latency for the first flow as T_{sys} and keep on adding flows, until T_{sys} becomes larger than the rescheduling period, at which point the flow 1 output stream will break.

Figure 4.38 shows the T_{sys} and its components.

Figure 4.39 shows the result of our experiment.

The plot shows the average T_{sys} for first flow. The average is calculated using latencies for all chunks except the first and last one. These chunks are special cases. T_{sys}

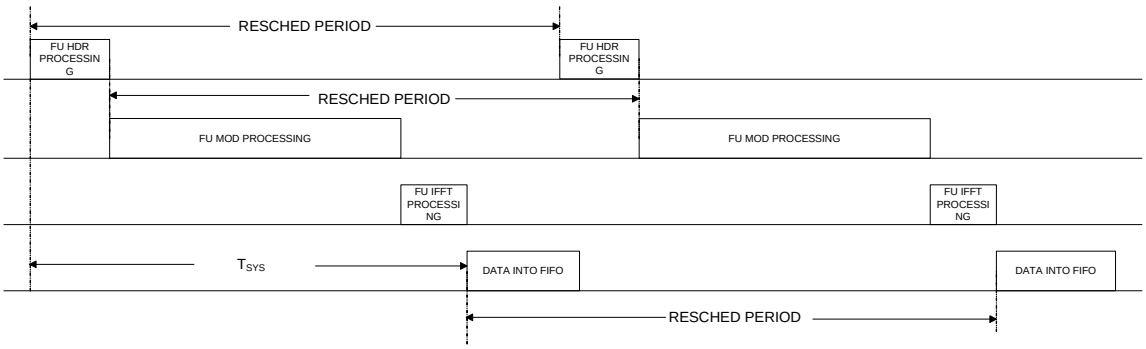

Figure 4.38: System Latency Components

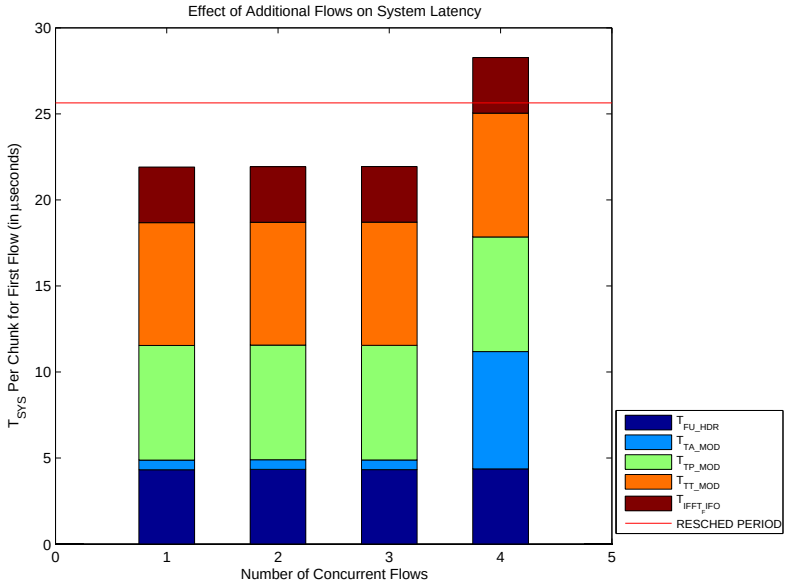

Figure 4.39: System Latency Measurement for Multiple Flows

is divided into 5 components. T_{FU_HDR} is total processing latency for FU HDR. The T_{TA_MOD}, T_{TP_MOD} and T_{TT_MOD} are FU MOD latency components. The T_{FFT_FIFO} is the sum of TA latency for FFT FIFO plus time PE IFFT takes to write the first word to FIFO after receiving the task.

We can see from the plot that up to three concurrent flows the system latencies are about the same. The actual data shows that there are minor variations due to delay in getting access to the bus. But for the fourth flow, the T_{sys} suddenly jumps and crosses the red line denoting Rescheduling period. This indicates that, in case of four concurrent flows, the output stream for flow 1 breaks. It is also seen that the added latency is because of jump in T_{TA_MOD}.

The reason behind this apparent increase in T_{TA_MOD} is that as the number of concurrent flows is increased; the tasks start getting queued up in in FU MOD. The PE can process only one task at a time. The other tasks have to wait in the UCM until PE finishes the current task. For PE MOD, Figure 4.16 shows us that TP and TT latencies are large. Hence, the waiting time is also large. Figure 4.40 shows the processing latencies in the FU MOD for 4 concurrent flows.

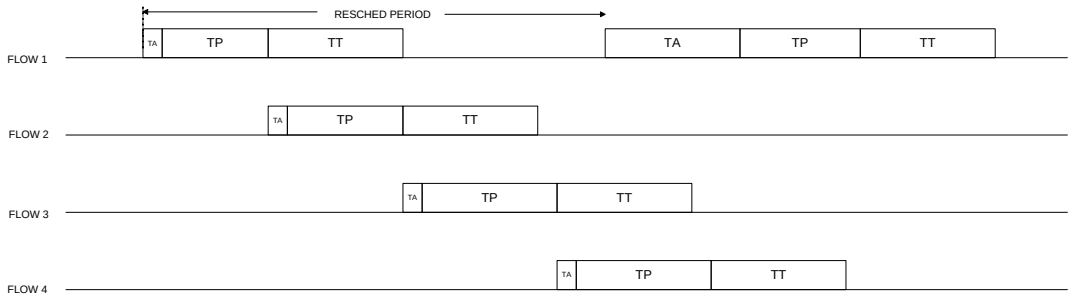

Figure 4.40: FU MOD Latency Measurement for Multiple Flows

It is seen that when the second chunk task for the first flow arrives, UCM is busy processing TT for first chunk of flow 3 and TP for first chunk of flow 4. Until the UCM is done with one of these it has no space to process the incoming flow 1 task. Since, the TP and TT take large amount of time, the flow 1 second chunk task sits in the UCM. Hence the TA latency increases.

We thus conclude that the TP and TT latencies in the FU MOD are bottlenecks when supporting multiple flows.

4.5 Effect of Change in Clock Frequency

In the previous experiments, we evaluated the performance of the WiNC2R platform with different experimental settings. From these experiments, we were able to conclude that for the given operating settings WiNC2R was not able to satisfy the SIFS criterion because of the processing delay in PE SYNC. Also, the number of simultaneous flows the system can support is limited by the processing delay in the PE MOD and the task termination delay for the same PE.

In all these experiments, we had kept the operating frequency, at which the simulation was run, at 100MHz. The reason behind this is that for higher frequencies, the implementation of the receiver chain in the FPGA was not able to meet timing constraints. The clock frequency is thus limited by the resource requirements rather than the architecture. If implemented on an FPGA with larger resources, the design could be run at higher clock frequencies. In this section, we do a simple analysis to estimate the clock frequency at which the WiNC2R can satisfy the SIFS criterion.

From the measurements of Section 4.2, we see that the maxim WiNC2R gives maximum value of TAT, approximately 40 μs, for 1250 byte frame with 2 OFDM symbols per chunk. In terms of number of clock cycles, the TAT value is 4000 clock cycles. To finish this number of clock cycles within the SIFS time (16 μs), the clock frequency should be $4000/16 = 250$MHz.

We thus see that increasing the operating frequency to 250 MHz will be enough for the current WiNC2R implementation to satisfy the SIFS criterion. The requirement to increase the clock frequency is the availability of more resource-rich FPGA. Given the improvements in chip design based on Moore's law, development of such an FPGA can easily be imagined within the next few years.

The increase in clock frequency will also increase the number of simultaneous flows the system can support. As seen in Section 4.3, this number is equal to maximum number of flows at which T_{sys} is less than the rescheduling period. The rescheduling period inversely varies with the rate which FIFO in Tx FFT is read. With increase in clock frequency, this rate will decrease to maintain constant rate of OFDM symbols at the

output. Thus, rescheduling period will increase, increasing the number of simultaneous flows the system can support.

Chapter 5

Conclusions and Future Work

The WiNC2R was developed with an aim to achieve to speed of operation through hardware and flexibility through software. In this work, we studied the performance of the hardware components of WiNC2R on the ground speed and re-configurability. The architectural analysis provided in Section 3 illustrated that WiNC2R is on-the-go per-packet re-configurable platform. It offers modular approach provided with the separation of data and control flow which are important features for any CR platform.

We then conducted experiments to analyze the processing latencies in WiNC2R with an aim to figure out the best performance this architecture can give. The current platform implementation consists only 802.11a-like OFDM flow with slice of 802.11-like MAC operations. WiNC2R assigns the radio signal processing functions to PEs and control functions to corresponding UCMs. Our basic analysis for a simple ALOHA-based frame exchange revealed that for a single-OFDM based flow, the UCMs were idle 80% of the time. It is seen from the measurements that a single UCM could have been enough to support all four FUs in the transmit chain without any adverse effect on performance. Since UCM is complex and resource-intensive block, we believe that this idea of having a common UCM should be investigated further from resource conservation point of view. Analysis should be done so as to how many PEs a UCM can support.

Further analysis of UCM processing latencies indicated that the usage characteristics of the UCM differ for each PE. With a PE like PE MOD, the data transfer to the next PE constituted the dominant component of the UCM processing overhead. At the same time, for PE HDR, the data transfer in small chunks made the control operations overhead look bigger. This suggests that for better performance the architecture needs

to be tailored according to the application requirements. Currently, work is going on a new UCM architecture that will support more than one PE. This will reduce the control overhead in the task termination stage.

The results from the experiment to study the effect of frame size gave us results about the two important parameters RTT and TAT. It was found that TAT does not depend on frame size; but has an inverse relationship with the last chunk size. Also, processing 2 OFDM symbols in a chunk in the receiver reduced TAT. We also suggested that for more improvement, chunking operation should be moved to MAC layer FUs. In spite of these improvements, the TAT was still in the range of $30 - 40\mu$s for all frame sizes used.

The final experiment, tested the WiNC2R transmitter chain for concurrent multiple flows. It was found that for current settings, maximum three 802.11a-like flows can be multiplexed. When the number of concurrent flows goes above three, the large values of task processing and task termination latencies in FU MOD turn out to be the bottleneck.

The operating frequency for all these experiments was limited to 100 MHz. In the last analysis, we saw that this was due to resource constraints and not because of the WiNC2R architecture. If the design is run at 250 MHz, the current implementation will be able to satisfy the SIFS criterion for all frame sizes considered and also increase the number of simultaneous flows it can support. The requirement to increase the clock frequency is the availability of an FPGA with larger amount of resources.

To conclude, we say that the current WiNC2R architecture succeeds in providing important CR features like modularity and ability to support multiple standards. For the operating frequency at 250 MHz and 2 OFDM symbols per chunk, the current WiNC2R implementation will be able to satisfy the SIFS criterion. Since the majority of the TAT is now contributed by the FU SYNC block, and the processing delay of PE MOD limits number of flows the system can support simultaneously, we are working on more sophisticated architectures for PEs. This architecture will be based on commercial RISC cores and will help in improving the performance.

For future analysis, we suggest that more PEs be available so that a full 802.11

flow can be implemented. Different PE implementations might also help in simulating different types of flows in the system.

References

[1] III Mitola, J. and Jr. Maguire, G.Q. Cognitive radio: making software radios more personal. *Personal Communications, IEEE*, 6(4):13–18, Aug 1999.

[2] Zoran Miljanic, Ivan Seskar, Khanh Le, and Dipankar Raychaudhuri. The winlab network centric cognitive radio hardware platform - WiNC2R. In *Cognitive Radio Oriented Wireless Networks and Communications, 2007. CrownCom 2007. 2nd International Conference on*, pages 155–160, Aug. 2007.

[3] Gary Minden Joe Evans and Ed Knightly. Technical document on cognitive radio networks. Discussion papers, U.Kansas, Rice University, September 2006.

[4] Bill Lane. Cognitive radio for public safety. `http://www.fcc.gov/pshs/techtopics/techtopic8.html`.

[5] Sto:the next generation program. `http://www.darpa.mil/sto/smallunitops/xg.html`.

[6] Network centric cognitive radio platform(WiNC2R). `http://www.winlab.rutgers.edu/docs/focus/{W}i{NC}2{R}.html`.

[7] Gnu radio: the gnu software radio. `www.gnu.org/software/gnuradio`.

[8] Rice university: Wireless open-access research platform. `http://warp.rice.edu/`.

[9] Eric Blossom. Exploring gnu radio. `http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html`, November 2004.

[10] The universal software radio peripheral. `http://www.ettus.com/`.

[11] Usrpfaq/intro/fpga. `http://gnuradio.org/trac/wiki/UsrpFAQ/Intro/FPGA`, June 2008.

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

[13] P. Murphy, A. Sabharwal, and B. Aazhang. Design of warp: A flexible wireless open-access research platform. In *Proceedings of EUSIPCO*, 2006.

[14] C. Hunter, J. Camp, P. Murphy, A. Sabharwal, and C. Dick. A flexible framework for wireless medium access protocols. In *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, pages 2046–2050, 29 2006-Nov. 1 2006.

[15] S. Jain. Hardware and software for WiNC2R cognitive radio platform. Master's thesis, Rutgers University, October 2008.

[16] S. Satarkar K. Le, S. Jain and T. Hari. WiNC2R platform functional unit architecture. Architecture Specification Document, October 2008.

[17] Khanh Le. WiNC2R platform unit control module architecture. Architecture Specification Document, May 2008.

[18] Z. Miljanic K. Le and R. Rajnarayan. WiNC2R platform PE command specification. Architecture Specification Document, January 2008.

[19] Shalini Jain and Khanh Le. WiNC2R platform system flow. Architecture Specification Document, March 2008.

[20] Renu Rajnarayan. WiNC2R platform processing engine frame header specification. Architecture Specification Document, March 2009.

[21] S. Iman. *Step by Step Functional Coverge with System-verilog and OVM*. Hansen Brown, 2008.

[22] Xilinx, Inc. *EDK BFM Simulation Tutorial*, July 2006.

[23] Xilinx, Inc. *BFM Simulation in Platform Studio*, 2008.

[24] L. Peterson and B. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann Publishers, 3 edition, 2003.