
Amean Al-Safi

Western Michigan University, ameanshg@yahoo.com

Follow this and additional works at: https://scholarworks.wmich.edu/dissertations

Part of the Electrical and Computer Engineering Commons

Recommended Citation
https://scholarworks.wmich.edu/dissertations/3118
ARCHITECTURE, SIMULATION, AND IMPLEMENTATION OF COMMODITY
COMPUTER COMPONENTS IN SOFTWARE DEFINED RADIO SYSTEMS

by

Amean Al-Safi

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Electrical and Computer Engineering
Western Michigan University
April 2017

Doctoral Committee:

Bradley Bazuin, Ph.D., Chair
Janos Grantner, Ph.D.
John Kapenga, Ph.D.
Radio communications have evolved through an extended history of theoretical and practical component development into modern devices most often envisioned as the ubiquitous smart phones found in almost everyone’s hand on a university campus. During this development, radios have evolved from analog devices operating at low frequencies into nearly all digital processing systems referred to as Software Defined Radio (SDR) operating in frequency bands over 1 Gigahertz. Although specific forms and types of communication are fiercely pursued by commercial communication companies and industry, there remain numerous concepts where further advancement is possible, and applications, possibly less commercially viable, where advancements and improvements may provide tremendous benefit.

In this study, the availability of advanced programmable digital signal processing components for personal computers and digital system design that can be readily incorporated in SDR have been investigated, incorporated and demonstrated. The components involved in the implementations and simulations include personal computers, Graphical Processing Unit (GPU) based graphics cards, Universal Software Radio Peripheral (USRP), Field Programmable Gate Array (FPGA), Raspberry PI, and open source software. Moreover, the most important factors that have been considered in this dissertation are: flexibility, modularity, scalability, and performance.
ACKNOWLEDGEMENTS

Firstly, I would like to express my deepest appreciation to my supervisor and committee chair Dr. Bradley Bazuin, for his excellent guidance, continuous support, motivation, and immense knowledge. His patience, guidance, and knowledge, provide me with an excellent atmosphere for doing research. I could not have imagined having a better supervisor for my Ph.D. study. I also would like to thank my committee members, Dr. Janos Grantner and Dr. John Kapenga for their time, help, comments, and efforts.

I would like also to express my deepest gratitude to all the faculty and staff in the Electrical and Computer Engineering department for all the assistance they have provided me throughout my study at Western Michigan University.

I am very grateful to the Higher Committee of Education Development in Iraq (HCED) for giving me the opportunity to study abroad.

I am heartily thankful to my parents who gave me the moral support I need. I also would like to express my gratitude to my wife for her encouragement and support.

I also place my sense of gratitude to my research colleagues, relatives, friends, and everyone, who is directly or indirectly have supported me during this journey.

Amean Al-Safi
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ............................................................................................................. ii

LIST OF TABLES ........................................................................................................................... vii

LIST OF FIGURES .......................................................................................................................... viii

LIST OF ABBREVIATIONS ............................................................................................................. xiii

CHAPTER I ......................................................................................................................................... 1

BACKGROUND AND MOTIVATION ............................................................................................... 1

1.1 Introduction ............................................................................................................................. 1

1.2 Dissertation Goals ................................................................................................................... 5

1.3 Dissertation Contributions ....................................................................................................... 5

1.4 Dissertation Structure ................................................................................................................. 8

CHAPTER II ..................................................................................................................................... 10

COMMUNICATION SYSTEMS BACKGROUND ............................................................................. 10

2.1 History of Communication Systems ......................................................................................... 10

2.2 Evaluation of Communication Systems .................................................................................... 12

2.2.1 Analog Communication ....................................................................................................... 13

2.2.2 Digital Communication ........................................................................................................ 15

2.3 Implementation of Communication Systems ............................................................................ 16

2.4 Software Defined Radio .......................................................................................................... 22

2.5 SDR Architecture .................................................................................................................... 24

2.6 SDR Signal Processing Stages .................................................................................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.1 Radio Frequency Front-End</td>
<td>25</td>
</tr>
<tr>
<td>2.6.2 Analog to Digital and Digital to Analog Conversion</td>
<td>27</td>
</tr>
<tr>
<td>2.6.3 Digital Signal and Software Processing Choices</td>
<td>28</td>
</tr>
<tr>
<td>2.7 Needs for Communication Systems for Emergency</td>
<td>30</td>
</tr>
<tr>
<td>2.8 Summary</td>
<td>31</td>
</tr>
<tr>
<td>CHAPTER III</td>
<td>32</td>
</tr>
<tr>
<td>GPU BASED IMPLEMENTATION OF A 64- CHANNEL POLYPHASE CHANNELIZER</td>
<td>32</td>
</tr>
<tr>
<td>3.1 First Generation of DSP Based Communication Systems</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Second Generation of DSP Based Communication Systems</td>
<td>36</td>
</tr>
<tr>
<td>3.3 Channelizers</td>
<td>38</td>
</tr>
<tr>
<td>3.4 GPU and CUDA</td>
<td>40</td>
</tr>
<tr>
<td>3.4.1 GPU Architecture</td>
<td>42</td>
</tr>
<tr>
<td>3.4.2 CUDA</td>
<td>43</td>
</tr>
<tr>
<td>3.4.3 Key Concepts in CUDA</td>
<td>45</td>
</tr>
<tr>
<td>3.4.4 Memory Optimizations</td>
<td>48</td>
</tr>
<tr>
<td>3.4.5 Performance Optimizations</td>
<td>50</td>
</tr>
<tr>
<td>3.5 Implementation of Polyphase Channelizer Using GPU</td>
<td>50</td>
</tr>
<tr>
<td>3.5.1 Polyphase Channelizer</td>
<td>50</td>
</tr>
<tr>
<td>3.5.2 Design Options</td>
<td>61</td>
</tr>
<tr>
<td>3.5.3 Implementation Method</td>
<td>65</td>
</tr>
<tr>
<td>3.5.4 Implementation Results</td>
<td>72</td>
</tr>
<tr>
<td>3.6 Summary</td>
<td>73</td>
</tr>
</tbody>
</table>
# Table of Contents

## CHAPTER IV

### SOFTWARE DEFINED COMMUNITY RADIO USING LOW COST RESOURCES 75

#### 4.1 Community Radio ................................................................. 75

- **4.1.1 Community Radio Technical Concept** .................................. 77
- **4.1.2 Types of Modulation** ..................................................... 77
- **4.1.3 CR Transmitter** ............................................................... 79
- **4.1.4 CR Receivers** ................................................................. 80
- **4.1.5 Antenna** .................................................................... 80

#### 4.2 Simulation and Implementation of Communication Systems Using SDR Concepts ..................................................................................... 81

#### 4.3 Proposed Implementation Methods of CR System ........................................... 90

- **4.3.1 Implementation Resources** .............................................. 91
- **4.3.2 Implementation Results** ................................................... 95

#### 4.4 Summary ............................................................................ 98

## CHAPTER V

### FPGA BASED DIGITAL MODULATORS FOR SDR APPLICATIONS ................. 100

#### 5.1 Why use FPGA? .................................................................... 101

#### 5.2 FPGA Programming ................................................................ 101

#### 5.3 Digital Sine Wave .................................................................. 102

- **5.3.1 Taylor Series** ................................................................. 102
- **5.3.2 Direct Digital Synthesizer** .............................................. 103

#### 5.4 FPGA Based Digital Modulator: Review ....................................... 106
### Table of Contents-Continued

5.5  Implementation of Digital Modulators using FPGA .................................. 111
    5.5.1 Implementation of ASK Modulator ............................................. 112
    5.5.2 Implementation of FSK Modulator ............................................. 121
    5.5.3 Implementation of PSK Modulator ............................................. 124
    5.5.4 Implementation of QPSK Modulator ......................................... 130
    5.5.5 Implementation of QAM Modulator ......................................... 135

5.6  Summary .......................................................................................... 142

CHAPTER VI ......................................................................................... 145

CONCLUSION AND FUTURE WORK ..................................................... 145

6.1  Conclusion ....................................................................................... 145
6.2  Future Work ................................................................................. 147
6.3  Summary ....................................................................................... 150

BIBLIOGRAPHY .................................................................................. 151

APPENDICES ....................................................................................... 161
LIST OF TABLES

3.1: Comparison between OpenCL and CUDA .......................................................... 48

4.1: Estimated reach of different transmitter power .................................................. 80

4.2: USRP products ................................................................................................. 92

5.1: AND gate using VHDL and Verilog ................................................................. 102

5.2: The xdc of ASK(OOK) modulators ................................................................. 120

5.3: Device utilization summary of the ASK(OOK) modulator ................................ 121

5.4: Device utilization summary of the FSK modulator ............................................ 124

5.5: Device utilization summary of the PSK modulator ............................................ 129

5.6: Device utilization summary of the QPSK modulator ......................................... 135

5.7: 4QAM symbol representation ........................................................................... 138

5.8: 16QAM symbol representation .......................................................................... 140

5.9: Device utilization summary of the 4QAM modulator ......................................... 142

5.10: Device utilization summary of the 16QAM modulator ..................................... 142
LIST OF FIGURES

1.1: Milestones in the development of radio communications. ........................................... 1
1.2: Radio evolution 1980 - 2010. .......................................................................................... 3
2.1: Block diagram of superheterodyne receiver. ................................................................. 11
2.2: Claude Shannon model. .................................................................................................. 12
2.3: Block diagram of communication system. ........................................................................ 13
2.4: Information signal and its corresponding AM, FM, and PM signals. ......................... 14
2.5: Block diagram of digital communication. ........................................................................ 16
2.6: AM product modulator(a) with schematic diagram of analog multiplier (b). .......... 17
2.7: Transconductance multiplier ............................................................................................ 18
2.8: Square-low modulator circuit (a) with transistor based implementation (b). ........ 19
2.9: Envelope detector circuit. ................................................................................................. 20
2.10: DSP based communication system. ................................................................................ 21
2.11: Block diagram of an SDR based communication system. .............................................. 22
2.12: A simple SDR block diagram with possible ADC/DAC locations ......................... 23
2.13: Superheterodyne RF front end. ....................................................................................... 27
2.14: Direct conversion RF front end. ..................................................................................... 27
2.15: Generic SDR system with possible use of different processors. ............................. 29
3.1: First generation DSP based receiver ................................................................................. 35
3.2: First generation DSP based transmitter. ........................................................................... 36
3.3: Second generation DSP based receiver. ........................................................................... 37
3.4: Second generation DSP based transmitter. ...................................................................... 38
3.5: FDM input signal ............................................................................................................ 39
3.6: Conventional channelizer. .............................................................................................. 40
List of Figures-Continued

3.7: NVIDIA Tesla GPU image................................................................. 41
3.8: CPU vs GPU ................................................................................. 42
3.9: NVIDIA GPUs roadmap................................................................. 42
3.10: Vector addition using C and CUDA ............................................ 44
3.11: CUDA thread execution model and memory hierarchy .............. 47
3.12: Pinned and non-pinned memory ................................................. 49
3.13: Polyphase channelizer ................................................................. 51
3.14: Kth channel of conventional channelizer ................................. 52
3.15: Spectral description of down conversion realized by a complex bandpass filter at a
      multiple of output sample rate, aliased to baseband by output resampling .... 54
3.16: Conceptual digital filter ................................................................. 55
3.17: IDFT based polyphase channelizer ............................................. 58
3.18: Polyphase Channelizer with two circular buffers ....................... 60
3.19: Polyphase channelizer with equal values of M and K ................ 61
3.20: Polyphase channelizer with unequal values of M and K .............. 61
3.21: Input signal with 50 FDM channels ........................................... 62
3.22: Single channel channelizer .......................................................... 63
3.23: Example design .......................................................................... 65
3.24: Input test signal and its spectrum .............................................. 67
3.25: System1 specifications ................................................................. 68
3.26: System2 specifications ................................................................. 69
3.27: Implementation algorithm pseudocode ...................................... 71
3.28: Channelized output signal .......................................................... 72
List of Figures-Continued

4.1: AM, FM, and PM modulated signals .......................................................... 79
4.2: AM modulation in MATLAB SIMULINK .................................................. 81
4.3: AM modulated signal using MATLAB SIMULINK ..................................... 82
4.4: Simulation of AM-DSB transmitter and receiver using GNU Radio .......... 84
4.5: AM-DSB transmitter using GNU Radio ..................................................... 84
4.6: AM-DSB receiver using GNU Radio .......................................................... 85
4.7: Simulation of PSK transmitter and receiver using GNU Radio ................. 86
4.8: Simulation of FSK transmitter and receiver using GNU Radio ................. 86
4.9: Simulation of QPSK transmitter and receiver using GNU Radio ............. 87
4.10: QPSK transmitter using GNU Radio ....................................................... 88
4.11: QPSK receiver using GNU Radio ............................................................ 88
4.12: QPSK modulated signal and its spectrum .............................................. 89
4.13: Received QPSK signal in time domain and its constellation ................... 89
4.14: Community radio transmitter using PC and USRP N210 ......................... 90
4.15: Community radio receiver using RTL-SDR and raspberry PI ................ 91
4.16: USRP (a) Block diagram and (b) B100 Image ....................................... 93
4.17: Image of raspberry PI 2 ........................................................................ 94
4.18: RTL-SDR block diagram ....................................................................... 95
4.19: CR transmitter flow graph ..................................................................... 96
4.20: CR receiver flow graph ......................................................................... 96
4.21: Transmitted signal in time domain ....................................................... 97
4.22: Transmitted signal in frequency domain ............................................... 97
4.23: RSA 3303A real time spectrum analyzer snapshot ................................ 98
List of Figures-Continued

5.1: Block diagram of DDS. ................................................................. 103
5.2: CORDIC Micro-Rotations.......................................................... 106
5.3: ZYBO board. .............................................................................. 112
5.4: ASK modulators. ......................................................................... 113
5.5: ASK modulated signal. ............................................................... 113
5.6: Block diagram of ASK(OOK) modulator implementation in VHDL. .... 114
5.7: Block diagram of ASK(BASK) modulator implementation in VHDL. ........ 115
5.8: Block diagram of 4ASK modulator implementation in VHDL. ................. 116
5.9: Digital sine wave carrier ............................................................. 117
5.10: ASK (OOK) modulation results. .................................................. 117
5.11: ASK (BASK) modulation results. ............................................... 118
5.12: ASK (4ASK) modulation results. ............................................... 118
5.13: FSK modulated signal ............................................................... 122
5.14: Block diagram of BFSK modulator implementation in VHDL. .......... 122
5.15: FSK modulation results in ModelSim. .......................................... 123
5.16: FSK modulation results in MATLAB. .......................................... 123
5.17: PSK modulated signal ............................................................... 125
5.18: BPSK constellation diagram ...................................................... 125
5.19: Block diagram of the BPSK modulator implementation in VHDL. ....... 127
5.20: BPSK implementation results in ModelSim.................................... 128
5.21: The two-generated out-of-phase digital sinusoids............................... 128
5.22: BPSK implementation results in MATLAB................................. 129
5.23: QPSK constellation diagram .................................................... 130
List of Figures-Continued

5.24: QPSK block diagram. ........................................................................................................ 132
5.25: Block diagram of the QPSK modulator implementation in VHDL. ............................. 132
5.26: QPSK implementation results in ModelSim. ................................................................. 133
5.27: The four generated digital sinusoids with 90-degree phase shift. ................................. 134
5.28: QPSK implementation results in MATLAB. ................................................................. 134
5.29: QAM generation. ........................................................................................................... 136
5.30: 4QAM constellation diagram. ...................................................................................... 137
5.31: Block diagram of the 4QAM modulator implementation in VHDL. ........................... 137
5.32: 16QAM constellation diagram. ..................................................................................... 139
5.33: Block diagram of 16QAM modulator implementation in VHDL. ................................. 139
5.34: 4QAM modulation results. ............................................................................................ 141
5.35: 16QAM modulator results. ............................................................................................ 141
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Convertor</td>
</tr>
<tr>
<td>AM</td>
<td>Amplitude Modulation</td>
</tr>
<tr>
<td>AM-DSB</td>
<td>Amplitude Modulation -Double Side Band</td>
</tr>
<tr>
<td>AP</td>
<td>Access Point</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>ASK</td>
<td>Amplitude Shift Keying</td>
</tr>
<tr>
<td>BASK</td>
<td>Binary Amplitude Shift Keying</td>
</tr>
<tr>
<td>BFSK</td>
<td>Binary Frequency Shift Keying</td>
</tr>
<tr>
<td>BPSK</td>
<td>Binary Phase Shift Keying</td>
</tr>
<tr>
<td>BS</td>
<td>Base Station</td>
</tr>
<tr>
<td>CIC</td>
<td>Cascaded Integrator–Comb</td>
</tr>
<tr>
<td>CORDIC</td>
<td>COordinate Rotation DIgital Computer</td>
</tr>
<tr>
<td>CPFSK</td>
<td>Continuous Phase Frequency Shift Keying</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CR</td>
<td>Community Radio</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital to Analog Convertor</td>
</tr>
<tr>
<td>DDFS</td>
<td>Direct Digital Frequency Synthesizer</td>
</tr>
<tr>
<td>DDS</td>
<td>Direct Digital Synthesizer</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete Fourier Transform</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FDM</td>
<td>Frequency–Division-Multiplexed</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FM</td>
<td>Frequency Modulation</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSK</td>
<td>Frequency Shift Keying</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphical Processing Unit</td>
</tr>
<tr>
<td>GPPs</td>
<td>General Purpose Processors</td>
</tr>
</tbody>
</table>
**List of Abbreviations-Continued**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>IDFT</td>
<td>Inverse Discrete Fourier Transform</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IF</td>
<td>Intermediate Frequency</td>
</tr>
<tr>
<td>IOT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual property</td>
</tr>
<tr>
<td>ISM band</td>
<td>Industrial, Scientific, and Medical radio band</td>
</tr>
<tr>
<td>LNA</td>
<td>Low Noise Amplifier</td>
</tr>
<tr>
<td>LTE</td>
<td>Long-Term Evolution</td>
</tr>
<tr>
<td>LUT</td>
<td>Look Up Table</td>
</tr>
<tr>
<td>MSK</td>
<td>Minimum Shift Keying</td>
</tr>
<tr>
<td>MUX</td>
<td>Multiplexer</td>
</tr>
<tr>
<td>NBFM</td>
<td>Narrow Band Frequency Modulation</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency-Division Multiplexing</td>
</tr>
<tr>
<td>OOK</td>
<td>On Off Keying</td>
</tr>
<tr>
<td>PFBC</td>
<td>Polyphase Filter Bank Channelizer</td>
</tr>
<tr>
<td>PM</td>
<td>Phase Modulation</td>
</tr>
<tr>
<td>PSK</td>
<td>Phase Shift Keying</td>
</tr>
<tr>
<td>PWM</td>
<td>Pulse Width Modulation</td>
</tr>
<tr>
<td>QAM</td>
<td>Quadrature Amplitude Modulation</td>
</tr>
<tr>
<td>QPSK</td>
<td>Quadrature Phase Shift Keying</td>
</tr>
<tr>
<td>RF</td>
<td>Radio Frequency</td>
</tr>
<tr>
<td>SDR</td>
<td>Software Defined Radio</td>
</tr>
<tr>
<td>SoC</td>
<td>System on Chip</td>
</tr>
<tr>
<td>SR</td>
<td>Software Radio</td>
</tr>
<tr>
<td>TDM</td>
<td>Time-Division-Multiplexed</td>
</tr>
<tr>
<td>UNESCO</td>
<td>United Nations Educational, Scientific and Cultural Organization</td>
</tr>
</tbody>
</table>
List of Abbreviations-Continued

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USRP</td>
<td>Universal Software Radio Peripheral</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very high speed integrated circuit Hardware Description Language</td>
</tr>
<tr>
<td>VHF</td>
<td>Very High Frequency</td>
</tr>
<tr>
<td>WBFM</td>
<td>Wide Band Frequency Modulation</td>
</tr>
</tbody>
</table>
CHAPTER I
BACKGROUND AND MOTIVATION

1.1 Introduction

Wireless communication systems have seen significant developments from James Clark Maxwell in the nineteenth century up to the present. But there is no doubt that the last few decades have witnessed some tremendous developments as compared with other parts of communication history. The progress that has been made is not only in the theory of communication but also in the way that communication systems are implemented. Generally speaking, the developments that have been achieved can be divided into major historic periods and milestones as shown in Figure 1.1 [1].

Figure 1.1: Milestones in the development of radio communications [1].
The first milestone is Maxwell’s equations. Maxwell’s equations govern the way that electromagnetic signals move from one place to another through free space or any other communication medium. Maxwell was able to present his equations after the discoveries of Orested, Faraday, and other scientists who discovered a connection between the electric and magnetic fields.

The second set of milestones involves the proof of the existence of the electromagnetic waves and the validation of Maxwell’s equations. Significant steps taken during this period were done by Guglielmo Marconi and Alexander Popov, by the virtue of the contributions of other scientists such as Heinrich Hertz, Nikola Tesla, Oliver Lodge, Reginald Fessenden, Amos Dolbear, Mahlon Loomis, Faraday, Orested, and Nathan Stubblefield, and others [2].

The next significant advancements involved the implementation of the first radio, a spark gap radio, wireless telegraphy, and voice transmissions. During the second period of radio development, standardized forms of analog communication were defined, advancements in devices, components and designs improved receiver and transmitter designs and both private and commercial radio stations became commonplace. The analog radio or the analog based implementation radios dominated the communication world for an extended time until the 1980’s, as shown in Figure 1.2, when the first microprocessor and other Digital Signal Processing (DSP) based radios started to appear in radio transceivers which can be considered as the third period in radio development [1].
The DSP based radio systems provide several advantages as compared with the analog based radios, such as flexibility, efficiency, and quality of service. Currently, commercial communication systems are mixed between analog and DSP based where part of the implementation is in analog while the rest is digital. With the presence of software programmable DSP devices performing most signal processing operations, the concept of Software Radio (SR) or Software Defined Radio (SDR) has been introduced and can be considered a fourth period of radio development. Initially developed by contractors for military applications, the terminology and concepts were introduced by Joseph Mitola to a broader audience in the 1990’s [3-5] as a way of implementing part or the entire communication system using reconfigurable devices. Since then and strongly coupled with the advancement and demand for smartphones,
SDR based concepts and systems have increased every year and will continue to do so due to the proliferation of cost effective devices, components, and terminal devices that make up the main part in any SDR based system.

Despite these developments, there remains a range of research areas in communication systems that still needs more study and research, beyond the ubiquitous smartphone. Further concepts and system advancements in SDR in meeting Base Station (BS) or fixed transceiver sites for capacity and flexibility as well as less profitable application where user simplicity with flexibility, minimal cost and size are needed. Just as SDR has embraced programmable DSP, other forms of high speed or even supercomputing systems should be considered as component size decreases and capability increases. Advancements in Graphic Processing Units (GPUs) having numerous homogeneous parallel computing elements have greatly impacted supercomputer architectures and show great promise for communications as well.

Another area is providing emergency communication. Many parts of the world have and continue to experience disasters, both natural and caused by humans, which result in the destruction of basic infrastructure, including communication links and systems. Natural disasters from earthquakes, typhoons, volcanos, tsunamis, and other disasters have regularly occurred throughout history, and may be increasing due to climate change. In addition, there are several man-made disasters where large numbers of people have been displaced and relocated to areas with little or no basic infrastructure, such as power, running water, or basic communication. In order to deal with these kinds of catastrophes, flexible low-cost wireless communication systems can provide essential
information and communication to support recovery until the crisis is over or basic infrastructure can be built or restored. While focused on emergency applications, such systems are also applicable for regions where cellular telephone, Wi-Fi or another communications infrastructure do not exist.

This dissertation presents three projects which can contribute to the development of SDR based communications systems for systems that vary from high throughput and capacity to those required for emergency communications.

1.2 Dissertation Goals

The goal of this dissertation is to investigate, prototype and demonstrate the critical components, processing and system architecture required to implement novel forms of SDR communications. The hardware resources that will be used are: personal computers, GPU, Universal Software Radio Peripheral (USRP) devices, RTL-SDR USB dongles, Raspberry PIs, and Field Programmable Gate Array (FPGA). Software resources to support this work include Xilinx Vivado, ModelSim, Digilent Adept, MATLAB, C, C++, Compute Unified Device Architecture (CUDA), Python, and GNU Radio. Once successful prototype developments and demonstrations of all necessary elements have been achieved, the complete system architecture can be constructed, demonstrated and made available.

1.3 Dissertation Contributions

The contributions of the dissertation have resulted from work on three projects with main goals of establishing powerful and flexible base station, low-cost receiver components to act as low-end PCs and radio receivers, and low-cost minimal component
transmitters for fully functional remote stations or even as transmitting channels for the base station(s).

The first project is the implementation of a 64-channel polyphase channelizer using GPU. A Polyphase Filter Bank Channelizer (PFBC) represents a significant custom, proprietary signal processing element found in many wireless communication system base stations. The successful implementation of such a processing element using PC level computers and compatible GPU parallel processing cards suggests that complete wireless communication system base stations can be constructed using SDR concepts and architectures. The replacement of custom, proprietary wireless base station processing systems with PCs and software can significantly reduce future communication system costs while enhancing flexibility and currently enables low-cost rapid deployment of a communication base station for basic communication needs. The same implemented channelizer can be generalized to any number of channels with only minor software changes.

The second project consists of several smaller SDR projects. The main goal of these projects is to provide low-cost SDR based communication systems that can be used during emergency situations. Nowadays, most of the communication systems that have been used during rescuing missions are hardware-based especially those used by the United Nations. Unfortunately, these kinds of systems are extremely expensive. Besides the higher cost, they need professional engineers to operate, maintain, and update them. We want to establish low-cost SDR based systems as a better solution than the currently used hardware-based ones. One of the several projects in the second part of the
dissertation is the implementation of Community Radio (CR) using cost-effective resources. Community radio is one form of radio broadcasting that has significant use in small communities for educational and social activities. It also has been used by the United Nations during rescue missions. The hardware-based community radios have several issues, such as the high cost, low flexibility, minimal system updates, and require regular maintenance. They also need experts to make them operate. This work seeks to establish low-cost SDR systems as a better solution than the current hardware-based ones. The goal is to define and demonstrate critical elements of the SDR based community radio system that could be developed in high volume, at low cost, and high flexibility for rapid deployment. Both sides of the community radio system are implemented using cost-effective terminals such as USRP, Raspberry PI, RTL-SDR dongle, and antennas. Successful reception of the transmitted signal has been achieved under different scenarios. The idea of building community radio transmitter based on SDR concept has been investigated before but no one ever has investigated the implementation of a complete community radio system using these kinds of resources.

The third project in this dissertation is dealing with the use of FPGAs to build more advanced, flexible, and powerful communications systems for emergencies. The implementation of alternate minimal component, low cost communication system transmitters using FPGA can be used as part of an emergency community radio system. This project focuses on how to make FPGA work as a processing unit to do all the required processing tasks for several digital communication transmitters. If we look at any SDR based systems, we will discover that the hardware terminals are the most expensive part of the systems. But these reconfigurable terminals hardware such as USRP
are responsible for almost all the processing tasks in the system. These terminals usually consist of FPGA, Digital to Analog Convertor (DAC), Analog to Digital Convertor (ADC), and mixers. Our main goal of the third project is to use FPGA with several low cost electronic components to build SDR based communication transmitters and keep the cost as low as possible and avoid the use of any Intellectual property (IP) blocks, which would increase the building cost.

1.4 Dissertation Structure

The rest of the dissertation is organized as follows: chapter II will present an overview of communication systems from the early stages of communication until today. This review is necessary to understand the current existing problems that need to be addressed today and in the future, and the possible solutions that can be used to solve these problems.

Chapter III provides a review of the DSP based communication systems from the first generation where the sampling happens at the baseband to the second generation where the sampling occurs at the Intermediate Frequency (IF) band to the most recent generation which is the polyphase channelizer. After that it illustrates the concept of GPU and its architecture and the most common way of programming GPUs, which is CUDA. Besides this review, it presents its main goal which is the implementation of a 64 - channel rational rate polyphase channelizer using GPU.

Chapter IV presents the implementation of community radio stations using the SDR concept and low cost resources. The chapter describes the SDR system overview, architecture, as well as the signal processing stages within SDR systems. Several SDR
based systems in real time and simulation mode are also presented in this chapter in addition to the main project.

Chapter V is the implementation of different SDR based digital modulators using FPGAs. FPGA is the main part in USRPs which represents the essential component in any SDR based system. Hence this chapter presents several projects that deal with the implementation of digital modulators using SDR concept and FPGA as a processing base.

The last chapter of the dissertation is chapter VI which summarizes the contributions of this research work, provides recommendations to extend it, and directions for possible future work.
CHAPTER II

COMMUNICATION SYSTEMS BACKGROUND

This chapter presents an overview of communication systems from the early stages of communication up to this moment. It provides a useful reference emphasizing the progress of digital signal processing into wireless communication, first as dedicated hardware elements and more recently as software based computational elements. With this background, it helps one understand some of the current challenges and problems that can and need to be addressed now and in the future and the available technology and technical approaches that can solve these issues.

2.1 History of Communication Systems

The story of communication began long time ago when people used traditional methods to share information such as mirrors, fire, torch signaling, and birds [6]. However, what is really considered communication today started after the great achievements of Maxwell in the nineteenth century [7]. After the discovery by Orested and then Faraday of the direct relationship between the electric and magnetic fields, many scientists worldwide tried to determine an appropriate physical formula for this relation [8-9]. Maxwell, relying on himself and the discoveries of Orested, Faraday, Hertz and others, was able to interpret this connection by four equations that govern the movement of any electromagnetic wave through any communication medium; later on, Marconi was able to physically demonstrate the concepts by transmitting and receiving communication signals through free space.
After these discoveries, twentieth century communication systems showed great developments from the first superheterodyne receiver [10] as shown in Figure 2.1, to the concept of information theory described by Claude Shannon [11] (see Figure 2.2), the first use of DSP processors in the implementation of a communication system [12], and finally the contributions of Joseph Mitola, who first presented the concepts of SDR [3-5] and Cognitive Radio [13-14]. Much of the progress was driven by military funding and researchers to support war efforts, particularly WWII. These advances and achievements provided fundamental steps which have made a significant impact on today’s communication systems.

Figure 2.1: Block diagram of superheterodyne receiver.
2.2 Evaluation of Communication Systems

The main task in any communication system is transmitting information signals from one place to another. Figure 2.3 explains, in a simple way, the main parts in any communication system. The first part is the information source, which could be voice, text, video, or any kinds of information. The second part is the transducer, which is responsible for converting information signals into electrical signals [7]. The third part is the transmitter. It is responsible for creating the electric signal which goes through the air using a process known as modulation. The output signal from this part passes through the channel where noise and interference may be added, and both linear and non-linear operations may modify the signal. The receiver part is responsible for getting the electrical signal back and overcoming any channel effects, noise and interference [7]. Finally, the electrical signal can be used to provide at the output the originally transmitted information.

Figure 2.2: Claude Shannon model.
Communication systems can be classified into analog and digital communication systems based on the way the message signal is modulated. They also can be classified into the same categories based on the method they are implemented. The next few sections will describe these systems in detail.

2.2.1 Analog Communication

The old communication systems were analog. At that time, the information signal, the communication medium, and the implementation devices were analog components. The transmitter changes the attributes of the carrier signal based on the message signal through a modulation process. There are three basic kinds of analog modulation; Amplitude Modulation (AM), Frequency Modulation (FM), and Phase Modulation (PM). In AM, the amplitude of the carrier signal is changing based on the information signal while in FM and PM, the frequency and the phase of the carrier signal are changing based
on the information signal respectively. Figure 2.4 shows an information signal with its corresponding AM, FM, and PM signals. AM is considered as continuous modulation while FM and PM are considered as exponential modulation [15]. AM and FM have been used in radio and analog TV broadcasting since the early part of the twentieth century [16]. In fact, analog modulation has been used in all kinds of wireless communication including the first generation cellular telephone and mobile communication systems. Even though analog modulation dominated the world of communication for decades [7,16] the signals that can be recovered are modified or even significantly degraded by the presence of noise in the environment. Proper detection of signals in the presence of noise is one of the reasons that led to the study and development of digital communication.

![Diagram of Message, Carrier, AM, FM, PM signals](image.png)

Figure 2.4: Information signal and its corresponding AM, FM, and PM signals.
2.2.2 Digital Communication

Digital modulation started to be the dominant mode of operation in the last few decades. But before going into the details of digital modulation, there is a very important point that has to be mentioned: for voice and sensor signals in digital communication systems, the front-end part of the system and the transmitted/received signal to/from the medium are still in the analog domain. Based on that, the digital modulation is nothing other than signal conversion and an alternate representation.

Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK), and Phase Shift Keying (PSK) are digital modulation schemes directly related to the analog signals previously described. In these systems, a digitized version of the analog information is transmitted sequentially as symbols and then converted back to analog in the receiver. In current systems, almost all the communication systems are either partially or completely digital in the signal processing and modulation scheme used. Figure 2.5 provides a detailed functional block diagram of a modern digital communication system [17]. If we compare the block diagram of digital communication shown in Figure 2.5 with the block diagram of the basic communication system shown in Figure 2.3, we discover that the former one has a number of additional functional blocks. These blocks are necessary to improve the quality of the communication, secure the communication, or to get the optimum use of the bandwidth. The use of any functional block at the transmitter side means the receiver must have a block that can reverse the work of the transmitter block as shown in Figure 2.5.
2.3 Implementation of Communication Systems

During the early decades of communication systems, the basic passive (resistance, inductance, capacitance, and diode) and active (tubes, diodes, transistor) components are the only way of implementing modulators/transmitters and demodulators/receivers [15]. The properties of the active devices to perform amplification and multiplication and provide oscillation were the key points in this kind of implementation. During this era, the modulation and its implementation method follow continuous time mathematics and are analog. Figure 2.6 is an example of how to implement AM modulation using active and passive components [15]. The main functions in this implementation are the oscillation, multiplication, and summation functions.
Implementing the summation function can be done by connecting the message signal source and the carrier signal source in series or by using an operational amplifier configured as an adder while implementing the product or mixing function is a little harder than the summation one. It can be done by a variety of options. One common way is the variable transconductance multiplier illustrated by Figure 2.7[15]. In this circuit
diagram, the input voltage signal \( v_1 \) is applied to the differential amplifier. The amplifier gain depends on the transconductance of the transistors, which depends on the total emitter current. The other input signal \( v_2 \) works as a controller of the emitter current by means of a voltage to current converter. The resulting output signal is the product of the two input signals \((v_1, v_2)\).

![Transconductance multiplier](image)

Figure 2.7: Transconductance multiplier.

Another implementation option is by using active elements working in the nonlinear mode as shown in Figure 2.8. In this case, the two input signals will be added together before being passed through a nonlinear element, with a strong squaring response along with other terms. Squaring the sum of signals will result in a product term that can be isolated by filtering to provide the desired output [15].
Figure 2.8: Square-low modulator circuit (a) with transistor based implementation (b).

The use of passive and active analog circuitry is the most common way of implementing not only modulators but also demodulators. Figure 2.9 is an envelope detector circuit which is one of the common methods of demodulating AM signals [15].
While these analog components and system implementation have dominated communication systems for many years, advancements in analog-to-digital and digital-to-analog converters and digital components capable of signal processing have gradually replaced most analog components and many analog modulation schemes. This process began with low digitization rate devices at the sensor level, where analog data was being collected and stored. Once data was digitized, the direct transmission of digital information provided improved performance, resolution and reliability based on digital data modulation schemes, supported by analog components. With the continued increases in clock rates and processing capabilities, even those components were slowly replaced by digital devices and even DSPs, leaving analog components to provide amplification, mixing and filtering functions, and little else in the way of modulation and demodulation processes. Current advanced communication systems contain mostly digital components performing digital signal processing algorithms to transmit and receive various forms of digital communications.

The great developments in DSP technology have opened the doors for new methods of implementing communication systems. This kind of implementation saves a
lot of the required hardware, especially for those systems that have multiple channels such as base stations. The use of DSPs in building communication systems has seen a lot of progress (as will be explained in chapter III) from the first generation where DSPs do the baseband processing only to the third and last generation where DSPs do almost all the required processing tasks. Figure 2.10 is a block diagram of an early DSP based communication system.

![Block diagram of an early DSP based communication system.](image)

**Figure 2.10:** DSP based communication system.

Nowadays, there are different DSP options that can be used for this kind of implementation such as FPGAs, GPUs, Application Specific Integrated Circuit (ASIC) devices, Central Processing Unit (CPU), and General Purpose Processors (GPPs) where each one has certain advantages and disadvantages. It is up to the researchers and design engineers to choose which DSP candidate to work with.
After the late 1990s, SDR started to be a common way of implementing communication systems. SDR provides a great flexibility for the designers as compared with other systems. Based on SDR concept and the proliferation of computers and cost effective front ends, different kinds of communication systems can be built. Figure 2.11 is a block diagram of an SDR based communication system.

Figure 2.11: Block diagram of an SDR based communication system.

The next three sections will explain SDR technique in detail since it is the main concept I used in this dissertation to build different kinds of communication using a variety of hardware and software resources.

2.4 Software Defined Radio

The SDR concept; or simply SR; was presented in 1991 by Joseph Mitola as a way of implementing programmable and reconfigurable radio transceivers [3-5]. In the SDR-based system, part of or the entire physical-layer can be implemented using programmable or reconfigurable platforms. The advantages of SDR-based systems became more apparent after the tremendous development in low cost hardware. The
terminal hardware of operators is an essential component in simulating and real time implementing of any communication system. Several research prototypes for SDR platforms have been developed such as WARP board from Rice University [18], the USRP platform from Ettus Research [19], the GENI SDR platform form Rutgers University [20], and the SORA platform from Microsoft [21]. Unfortunately, most of these platforms are more suitable for transceiver prototyping and reconfigurable Access Point (AP) or base station than consumer level devices due to the high cost and significant power consumption. Among these platforms, the USRP is one of the most common. Figure 2.12 is a general block diagram of a SDR based communication system [22].

Figure 2.12: A simple SDR block diagram with possible ADC/DAC locations.

During the last decade of the 20th century, SDR-based systems were not considered commercially viable due to the high cost of the frond ends and low processing capabilities of the available CPUs and DSP devices. Since then, the great developments in Integrated Circuit (IC) technology and the development of software tools have
encouraged many researchers to study, develop and use SDR techniques to build different kinds of communication systems. There are, however, significant differences between implementing communication system using SDR as compared to custom and conventional hardware implementations [23]. In general, SDR-based systems can be efficient and easy to build as compared to other systems.

2.5 SDR Architecture

Implementation of communication systems using SDR concept is similar to other implementation methods despite the differences in the components. Figure 2.11 is a general overview of an SDR based communication system [24]. This system consists of antenna, Radio Frequency (RF) front end hardware, DAC and /or ADC, and DSP processor. The antenna part refers to any kind of antenna from omnidirectional up to adaptive beamforming [23]. The RF front end hardware is responsible for signal mixing or frequency conversion from RF frequency band to IF band in the receiver mode of operation and from IF band to RF frequency band in the transmitter mode of operation. In order to do these tasks, the RF front end has to have a Low Noise Amplifier (LNA) and RF mixer which is responsible for mixing the signal into the required frequency band based on superheterodyne or direct conversion principles. Whether single or dual conversion is used, the SDR mixer generally covers a wider range of frequencies and may have higher noise figure and different filtering than custom radios built for specific applications.

The third part of any SDR based system is the ADCs and DACs. The ADC and DAC conversion processes usually operate in the IF band; reducing the sampling rate
required in the RF band; but the DACs and ADCs used have to be able to support the maximum desired frequency band. Since the sampling rates might change based on the system being implemented, the SDR based system often operates at sample rates higher than the required ones, and then they use digital signal processing front ends to lower (decimate) or increase (interpolate) the sample rate to the required values [23]. The digital front end consists of digital filters, interpolators and decimators, and mixers which are often implemented either on FPGAs or ASICs devices. The remaining signal processing tasks in the SDR systems can be done on different DSP processors such as FPGAs, multi-core CPUs, ASICs and/or GPUs using different software methodologies, tools, and development environments, such as high level programming languages (C/C++, CUDA, MATLAB), middleware, or dedicated software applications like LabVIEW, OpenBTS, and GNU Radio [22-23].

2.6 SDR Signal Processing Stages

As it was illustrated in the previous section and Figure 2.11, the SDR based system consists of several processing stages in the RF, IF, and baseband bands. These processing stages can be explained as follows:

2.6.1 Radio Frequency Front-End

The main tasks of the RF front end are pre-amplification, noise rejection, RF filtering, and frequency up and/or down conversion. During the transmission mode of operation, the RF front end takes the output signal of DACs, up converts it to a higher frequency, amplifies it, reduces the level of its harmonics, and sends it to the antenna [23]. In the receiver mode of operation, the RF front end receives the signal from the
antenna, separates it from the noise, pre-amplifies it using LNA, down converts it to the IF frequency band, and feeds it to ADCs [23]. The entire RF front end processing has been done in analog domain, even if some researchers think they can do it in digital domain [25]. In order to know how good the RF front end is, there are some factors that have to be checked such as [23]:

**Sensitivity:** It is the measurement of the weakest signal the receiver can detect.

**Selectivity:** It is the measurement of the receiver’s capability of detecting the desired signal while rejecting others.

**Dynamic Range:** It is the measurement of the power difference between the strongest and the weakest signal the receiver can detect.

**Stability:** It is the measurement of the changes in the gain and frequency response of the receiver due to the changes in temperature, time, etc.

Today, there are two architectures of RF front end: superheterodyne architecture which is based on Edwin Armstrong receiver and direct conversion architecture which is also known as the homodyne or the zero-IF architecture [23]. The mode of operations looks the same in both architectures but superheterodyne architecture uses two mixing and filtering stages to reach the IF band, while direct conversion architecture does all the processing requirements in single mixing and filtering stage. The superheterodyne architecture is shown in Figure 2.13, while the direct conversion architecture is shown in Figure 2.14.
2.6.2 Analog to Digital and Digital to Analog Conversion

ADCs and DACs play a very crucial role in any SDR system. Their operation directly affects the entire system performance. The placement of ADCs and DACs in the receiver and transmitter will define the signal processing bandwidth, dynamic range, and to a degree the power consumption of the radio system. Getting ADCs and DACs that satisfy all these requirements, especially those related to bandwidth and dynamic range is not easy and has been a limiting factor in the development of digital radios and low cost SDR. Based on Nyquist theory, the sample rate of the ADCs has to be at least two times...
the highest frequency within the signal of interest, but practically, digital signal processing can be greatly simplified if a higher rate is used.

The other important factor that has to be considered in ADCs and DACs’ selection process is the dynamic range. The dynamic range for devices used for communication is directly related to the number of bits the ADCs and DACs convert. The more valid conversion bits they have, the higher the dynamic range can be which means stronger and weaker signals can be simultaneously processed and detected without any problems. Current state of the art ADCs for SDR applications have greater than 3GHz bandwidth, with over 2Gsp/s sampling rate and approximately 77dB spurious free dynamic range and 12-bit resolution [26]. Spurious free dynamic range refers to magnitude between the weakest detectable signal in the presence of noise and the largest full scale signal.

Another factor that might need to be considered when selecting ADCs and DACs is the power consumption. Even if ADCs and DACs power is limited, in mobile or battery power application this can still be a concern.

2.6.3 Digital Signal and Software Processing Choices

One of the biggest advantages of using SDR systems is the wide range of DSP processing hardware available that can be used to do the processing tasks as shown in Figure 2.15 [23]. Each one of the hardware elements shown has its own advantages and disadvantages. In order to select the right processing hardware, there are four factors that have to be considered; flexibility, modularity, scalability, and performance [23]. The flexibility is the measurement of the range of protocols and standards the system can
handle. The modularity indicates how quickly the system can be reconfigured. The scalability refers to the possibility that the systems can be expanded, such as increasing the number of users or channels. The performance reflects the computation power, cost, and some other user-related factors that have to be met in the selection process.

The most common hardware that can be used in real-time SDR based implementation are: ASICs, FPGAs, DSPs, multicore GPPs and GPUs [27]. The variety of hardware options gives us different levels of programming, cost, reconfiguration, time and power consumption. Hence for some applications, FPGAs could be the right hardware candidates, while for others GPUs might be the right ones, especially the ones that need a lot of computation power. There are also some DSP processors that provide multicore support which make them good candidates for certain applications [28].

**Figure 2.15:** Generic SDR system with possible use of different processors.

The current SDR based systems use CPU most of the time; however, there have been a lot of attempts recently to use GPUs as support devices. Unfortunately, working with FPGAs and GPUs is not easy due to some challenges such as the programming software, reconfiguration, and the device architecture. In order to get an optimal
performance, designers have to understand these aspects before working on the problems and their solution [23].

2.7 Needs for Communication Systems for Emergency

Due to climate change and other factors, different parts of the world have seen huge disasters. These kinds of disasters can cause serious damages to infrastructure and human life. Humanitarian aid is the priority in these situations, but catastrophic failure in infrastructure might make it extremely difficult. Failing to deliver assistance on time puts more lives in danger. In order to prevent loss of life, one needs to build a portable and flexible communication systems that can provide temporary communication infrastructure. These kinds of systems are very helpful for rescuers to disperse aid in search and rescue of survivors. As research questions: Is it possible to build a flexible communication system in limited time to assist with disaster relief? Is it possible to build a base station to restore cell phone communication quickly? The answers to these questions must be yes.

In this dissertation, I present different commodity computer components along with their implementation and simulation results using simple commercial off-the-shelf resources such as laptop computers, batteries, antennas, USRP, FPGA, Raspberry PI, and open source software. This dissertation depends heavily on the recent progress in SDR technology, parallel processing, embedded systems, and System on Chip (SoC) devices.

The main and longtime goal is to build communication systems that are completely SDR based such as base station and radio broadcasting using cost effective resources. When building these types of systems, we need to consider other aspects also
such as the size of the system, flexibility, cost, maintenance, system updates, and power consumptions. The next three chapters will present the main research contributions in detail.

2.8 Summary

Wireless communication has seen a lot of developments from the early days of Maxwell’s equations up to this moment where different new systems such as SDR, SoC based, and DSP based communication systems have started to be a common way of implementing communication systems. This chapter presented a general overview of communication systems including the modulation and implementation methods. This overview is necessary to understand what, why, and how the main developments have happened. Addressing these questions will make a big impact not only on today’s communication systems but also the future ones. The chapter also provided a short introduction to the needs for communication systems that can operate during and shortly after catastrophic situations.
CHAPTER III

GPU BASED IMPLEMENTATION OF A 64-CHANNEL POLYPHASE CHANNELIZER

PFBCs are very important signal processing stages in many communication systems involving channel separation and sample rate reduction [29]. The efficient implementation of PFBC can make a significant impact on receiving system architecture and complexity. There are three classes of PFBC: integer rate decimation, rational fraction rate decimation and non-rational fraction rate decimation. While this chapter and results focus on rational fraction rate decimation, all three algorithms have been implemented using hardware based FPGA. The FPGA based implementation involves significant concerns and challenges related to the dynamic range using integer precision computations, coding complexity in either Very high speed integrated circuit Hardware Description Language (VHDL) or Verilog, and the typical need for application-specific implementations. These challenges make FPGA solutions rather inflexible, and their design is relegated to a handful of experts with considerable experience or implementations containing significant vendor-specific intellectual property contents.

This has driven many researchers to look for alternate software and parallel processing based techniques to overcome these challenges. With the evolving performance capability and success of SDR and the real-time computation capabilities of GPU, techniques and hardware to perform PFBC in real time already exist. GPUs have a tremendous computation capability and they can be easily programmed and updated using CUDA programming model and platform [30]. Although communication applications may not involve the high dimensionality or computational structure of many parallel processing algorithms, particularly those involving “big data” problems, using
GPUs as the base of the implemented systems can provide certain gains. For example, there has been research focused on the use of GPUs as a tool for PFBC. Some of this work has demonstrated an entire integer decimating PFBC system such as the work of Harrison et al [29], van der Veldt and Nieuwpoort [31-32], and Kim et al [33-34]. Other researchers have focused on Discrete Fourier Transform (DFT) or Fast Fourier Transform (FFT) architecture such as the work of Govianaraju et al [35], Mitra and Srinivasan [36], Ambuluri [37] and del Mundo et al [38].

In this chapter, the architectural background, GPU architecture and implementation of a software defined, fully parallelized PFBC is presented. This implementation demonstrates the most general form of a filter bank channelizer as compared to the more direct integer decimating form. The implementation was done in CUDA without using any standard CUDA library elements such as the CUFFT library as in other research work. Before going into the implementation detail, there are some concepts that have to be understood, such as DSP based communication systems and GPU programming. The next few sections will explain these concepts as well as some others in detail.

3.1 First Generation of DSP Based Communication Systems

Since the 1980s, with higher rate ADCs and DACs, advances in processor computation rates and the great progress and developments towards SoC technology, FPGA or software-based DSP has become the most common way of prototyping or even implementing communication systems. While Gordon Moore is known for projecting the increase in transistors and integrated circuit complexity in time, he was keenly aware of
the continuous cost reduction and additional capabilities and application that would be made possible [39]. This fact is one of the encouraging reasons to develop DSP theory, algorithms and approaches as a base of future digital processing implementation, even if currently impractical, not only for communication systems but also different kinds of systems that need high computation complexity and power.

One area of the common communication systems that requires a lot of signal processing is wireless base stations. They are examples of radio receivers that must down convert, demodulate, and separate multiple channels at the same time [12]. A first-generation base station architecture that can perform these tasks is shown in Figure 3.1. The architecture consists of multiple analog dual conversion subband receivers. Each signal processing chain is responsible for amplifying, down conversion from RF to IF band, and initial filtering of one channel signal. The output of each IF filter will go through quadrature matched mixers followed by a matched filter at baseband frequency. The output signals from the matched filters are going to be converted by a pair of ADCs into their corresponding digital signals. These digital signals will be processed by the DSP processors at the baseband frequencies. The DSP processor is responsible for the signal demodulation, detection, channel coding, equalization, and synchronization [12].

Figure 3.2 shows the first generation transmitter architecture for a base station. In the transmitter mode of operation, the signal flows in the reverse order of the receiver mode. The transmitter consists of a group of dual conversion sub transmitters. These sub transmitters are responsible for modulation, up conversion, and amplification of multiple signals at the same time. Despite the challenge of the baseband phase and gain imbalance
between the two paths containing the quadrature mixers which may lead to adjacent channel interference, this version of the first DSP based processors was a significant system improvement and reduced the required implementation hardware. It also opened the doors to more advanced and efficient DSP based communication transceivers.

Figure 3.1: First generation DSP based receiver [12].
3.2 Second Generation of DSP Based Communication Systems

With known DSP algorithms to perform quadrature signal processing, including mixing and filtering, it was only a matter of time until ADCs, DACs and computational components performed more of the signal processing for base stations. Figures 3.3 and 3.4 represent the second generation of the DSP based communication receiver and transmitter respectively. In this generation, the conversion from analog to digital and digital to analog occurs at the IF stage instead of at the baseband. The down conversions of the channels are done by a set of digital down converters and a digital low pass filter instead of the analog ones as in the first generation [12]. In this architecture, the RF filtering and amplifications, RF mixer, IF filter, ADCs and DACs are common for all channels. The small number of common components, high rate ADCs and DACs and digital devices to perform signal processing resulted in a huge reduction in the required hardware for the implementation. One of the biggest advantages of the second generation
DSP based transceivers is that by using digital techniques, the processing is identical for every channel as the filters are the same length with the same coefficients which minimizes phase and gain imbalance among the different paths. Another big advantage they have is that by using Finite Impulse Response (FIR) digital filters before or after the mixers a linear phase characteristic is achieved [40]. Therefore, instead of only doing the baseband processing, the second generation of DSP based transceivers were able to do digital mixing and filtering as well.

![Diagram of second generation DSP based receiver](image)

Figure 3.3: Second generation DSP based receiver [12].
This second generation architecture also led to the advancement of digital algorithms for communication and new digital integrated circuits. For multiple narrow band communications signals, the COordinate Rotation DIgital Computer (CORDIC) algorithm for complex mixing and the Cascaded Integrator–Comb (CIC) algorithm [41] for filter-decimation and interpolation-filter operations were developed and directly implemented into integrated circuits. They are now common library elements for both ASIC and FPGA software development systems and implantation. While these techniques work well for narrowband channels, alternate algorithms and approach are required for wider bandwidth channels or simultaneously processing all channels.

3.3 Channelizers

The digital filter bank channelizer is considered as the third generation DSP based communication systems with an initial reference by Bellanger et al in 1976 [42], a refined
description by Crochiere and Rabiner [43], and more recently detailed implementations with examples by Harris et al in 2003[12]. The main goal of the channelizer is to separate and provide independent, sample rate decimated outputs from a defined frequency band containing multiple contiguous, identical bandwidth channels. Described another way, the channelizer takes a Frequency-Division-Multiplexed (FDM) signal as shown in Figure 3.5 and provides filtered and decimated time samples, often in a Time-Division-Multiplexed (TDM) format, for each of the multiplexed channels. There are many applications that use identically spaced frequency channels radio spectra such as FM and AM radios, Very High Frequency (VHF) TV broadcasts, and early cellular telephone signal formats.

![Figure 3.5: FDM input signal.](image)

The signal processing functions performed by a channelizer are tuning, downsampling and rejection of adjacent channel interference and noise. Figure 3.6 is the functional block diagram of a conventional channelizer in which the FDM input signal can be processed by a group of DSP based down converters, filters, and mixers. In the conventional channelizer, each channel has a functional block diagram which makes it able to do the required processing tasks such as down conversion, low pass filtering,
adjacent channel separation, and down sampling. The mathematical derivation and implementation follow in a later section.

Figure 3.6: Conventional channelizer.

### 3.4 GPU and CUDA

GPU is an electronic circuit that was originally designed to accelerate graphics and image processing in a way that makes it very efficient in terms of altering memory and accelerating mathematical operations for the creation of images in a frame buffer in order to be displayed [44]. Beyond images and displays, GPUs have many uses in embedded systems, parallel processing, and computationally intensive applications. The
first GPU was presented by NVIDIA in 1999 which was the GeForce 256. The highly parallel structure of GPUs makes them more efficient than CPUs for certain applications that have a lot of block processing that can be done in parallel. Figure 3.7 is a picture of a Tesla GPU circuit card for a PC manufactured by NVIDIA. There are several companies that produce GPUs but Intel, NVIDIA, and AMD are the most common ones.

Figure 3.7: NVIDIA Tesla GPU image.

The best way to understand the differences between GPUs and CPUs is to look at Figure 3.8, which is the block diagram of CPU as well as GPU. The picture shows that a common CPU consists of a few multiple core processors that may be optimized for sequential or simple multithreaded processing whereas the GPU has thousands of smaller cores that are able to perform significantly more tasks at the same time.
3.4.1 GPU Architecture

Modern GPUs have evolved from fixed function graphics processing to a massive parallel processing with a computation power that exceeds current multicore CPUs. The increased computation rate of a GPU is based on exploiting the degree of parallelism available in certain classes of problems. Since NVIDIA is the best-known company in term of GPUs manufacturing, the NVIDIA GPU roadmap is shown in Figure 3.9.

Figure 3.8: CPU vs GPU.

Figure 3.9: NVIDIA GPUs roadmap.
The architecture of the NVIDIA GPUs has evolved from the Tesla architecture that can process a limited number of instructions with a limited bandwidth and single precision format to the Pascal architecture that can process an enormous number of tasks simultaneously with a wider bandwidth and double precision format. The detailed description of these architectures can be found in the NVIDIA white papers [45-49].

3.4.2 CUDA

CUDA is parallel programming software development system presented by NVIDIA. It enables developers to write code more efficiently and use CUDA-capable GPUs. CUDA software provides direct accesses to the GPU instructions set. It is designed to work with different programming languages such as FORTRAN, C, C++, Python, and MATLAB. Most researchers prefer to use CUDA with C due to familiarity and a fast learning curve.

To explain the difference between the general C code and CUDA code, one can look at Figure 3.10, which explains how to do a simple vector addition in C and CUDA. Here one can see that the CUDA kernel is similar to a C function. There are some differences between writing code in C, compiling and executing it as opposed to writing CUDA code and executing it. The C code is usually executed on a single processor core of the CPU while the CUDA code is executed on several cores of the GPU, thereby providing a speed up factor. Speed up factor refers to the time gained by processing the data in parallel as compared to conventional serial code. As the size of the processed data increases, the speed up factor also usually increases. More details on the key terms in
CUDA, qualifiers, memory allocation, data transfer, and kernel executions can be found in “CUDA By Example Textbook” [50] or CUDA programming guide by NVIDIA [30].

```c
#include <stdlib.h>

#define N 10

int main() {
    int a[N], b[N], c[N];
    // fill the arrays 'a' and 'b' on the CPU
    for(int i=0; i<N; i++) {
        a[i] = -i;  b[i] = i*i;  }
    add( a, b, c );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d \n", a[i], b[i], c[i] ); }
    return 0;
}

void add ( int *a, int *b, int *c ){
    int tid = 0; // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;  // we have one CPU, so we increment by one
    }
}

void __global__ Vector_Addition ( const int *dev_a , const int *dev_b , const int *dev_c){
    unsigned short tid = threadIdx.x ;
    if ( tid < N ) // check the boundary condition for the threads
        dev_c [tid] = dev_a[tid] + dev_b[tid] ;
}

int main (void){
    int Host_a[N], Host_b[N], Host_c[N], *dev_a , *dev_b, *dev_c ;
    cudaMalloc((void **)&dev_a , N*sizeof(int) );
    cudaMalloc((void **)&dev_b , N*sizeof(int) );
    cudaMalloc((void **)&dev_c , N*sizeof(int) );
    for ( int i = 0; i <N ; i++ )
        Host_a[i] = -i ;  Host_b[i] = i*i ;
    cudaMemcpy(dev_a,Host_a , N*sizeof(int) , cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b , Host_b , N*sizeof(int) , cudaMemcpyHostToDevice);
    Vector_Addition <<< 1, N >>> (dev_a , dev_b , dev_c );
    cudaMemcpy(Host_c , dev_c , N*sizeof(int) , cudaMemcpyDeviceToHost);
    for ( int i = 0; i<N ; i++ )
        printf ("%d + %d = %d\n", Host_a[i] , Host_b[i] , Host_c[i] );
    cudaFree (dev_a) ;   cudaFree (dev_b) ;  cudaFree (dev_c ) ;
    return 0 ;}
```

Figure 3.10: Vector addition using C and CUDA.
3.4.3 Key Concepts in CUDA

In order to get the optimum performance of the GPUs using CUDA programming or any other software, there are several concepts that have to be understood. In this section, the most important concepts will be explained. The basic concepts that have been widely used in CUDA are:

**Host**: The term host refers to the CPU side.

**Device**: The term device refers to the GPU side.

**Kernel**: It is a function callable from the host and executable on the device. It corresponds to a C function in CUDA. It is essential to add a qualifier __global__ to make the compiler understand that this is a kernel, not a C function and it is going to be executed on the device.

**Thread**: It is the smallest computing element on the device. The individual threads are referred to by the pointer threadIdx.

**Warp**: The instruction unit manages threads in groups of 32 parallel threads, known as a warp.

**Block**: It is a collection of a group of threads.

**Grid**: It is a group of blocks.

**Global Memory**: It is the largest accessible memory by the thread. Global memory is similar to the random-access memory on the CPU side. Threads can read and write in global memory which is visible to all threads. Accessing global memory is slow compared to other memory architecture.
**Constant Memory**: It is similar to global memory. Threads can read but not write in constant memory. Accessing constant memory is still slow but faster than global memory. Constant and global memory are visible from both the host and the device.

**Shared Memory**: Shared memory is the memory of the block. Each thread within the block can read and write in shared memory. Shared memory is very fast as compared to global and constant memory.

**Local and Register Memory**: Each thread has its own register and local memory. Threads can read and write on these memories very fast.

Besides these memories, there is also the texture memory which is very useful for certain applications like image processing. Figure 3.11 explains the memory hierarchy and some of the key terms in CUDA during thread execution.
The other way of programming GPUs is OpenCL (Open Compute Language) which has been developed by Khronos [51]. It is more general language than CUDA since it works with other general purpose devices and GPUs such as those from AMD, Intel, and others, but the programming complexity is higher than CUDA. Hence, for those who have NVIDIA GPUs, CUDA is a better choice than OpenCL. Table 3.1 is a simple comparison between CUDA and OpenCL. More detail on OpenCL can be found in a variety of textbooks and online resources, such as the khronos.org web site.
Table 3.1: Comparison between OpenCL and CUDA.

<table>
<thead>
<tr>
<th></th>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Language</td>
<td>C</td>
<td>C/C++</td>
</tr>
<tr>
<td>Supported GPUs</td>
<td>AMD, NVIDIA</td>
<td>NVIDIA</td>
</tr>
<tr>
<td>Method of Creating GPU Work</td>
<td>Kernel</td>
<td>Kernel</td>
</tr>
<tr>
<td>Run-time compilation of kernels</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Multiple Kernel Execution</td>
<td>Yes (in certain hardware)</td>
<td>Yes (in certain hardware)</td>
</tr>
<tr>
<td>Execution Across Multiple Components</td>
<td>Yes</td>
<td>Yes – only GPUs</td>
</tr>
<tr>
<td>Need to Optimize for Best Performance</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Coding Complexity</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>

3.4.4 Memory Optimizations

One of the most important things that have to be optimized to get the best performance of the implemented GPU based model is the required memory access. In GPUs’ architectures, there are different kinds of memories. The access time for each one is different. Maximizing the use of the memories that are close to threads such as local, register and shared memory can make a huge impact on the total system performance. On the other side, the more we use slow memories such as global memory, the slower our implemented system will be. In certain circumstances, there is no way to avoid slow memories, such as in moving the data between GPU and CPU over the PCIe interface before and after kernels execution. Saving the intermediate results of kernels executions in the shared memory will reduce the number of times that we need to access the global memory which means less time consumed. For small data size, there are additional
memory aspects to consider; it is recommended to use page locked or pinned memory instead of non-pageable memory. CUDA provides certain instructions to do that such as `cudaHostAlloc()` for memory allocation and `cudaMemcpyAsync()` for data transfer. The main idea behind the pinned memory is that we allocate part of the global memory to be common between the GPU and the CPU which means there is no need to move data from one to another because both of them can access the data at any time. Since a GPU cannot read or write data directly from pageable CPU memory, the default way that CUDA uses to solve this issue is to allocate a temporary page-locked space, then copy the data to this temporary space and then to the GPU. This means the data transfer process is performed in two steps. With the pinned data transfer we can make it a one-step process only as shown in Figure 3.12.

![Pageable and Pinned Data Transfer Diagram](image)

Figure 3.12: Pinned and non-pinned memory.
3.4.5 Performance Optimizations

In order to get the best performance, it is recommended to keep the GPU busy by distributing the load equally among the processors. If one of the GPU processors gets more processing tasks as compared with the remaining ones, it will need more time to complete its job. In this case the performance of the whole system will degrade. Distributing the computation load equally will also hide the memory latency, which also improves the total system performance [50]. For GPUs that have newer architectures, such as Kepler, they support dynamic parallelism, concurrent kernels execution, or data streaming, which can also speed up the performance.

3.5 Implementation of Polyphase Channelizer Using GPU

3.5.1 Polyphase Channelizer

The digital channelizer described in section 3.3 has become widely used in communication systems. By combining the mathematical concepts and digital implementation algorithms, a channelizer is often implemented as a PFBC [12]. In this implementation, a channelizer is constructed from parallel polyphase filter structures that are followed by a DFT as shown in Figure 3.13. The best way to understand the processing steps and diagram is by deriving the mathematical equations at each step.
If we examine the single channel conventional receiver shown in Figure 3.14, we can easily recognize the processing tasks that must be performed. These tasks are: down conversion mixing of the channel of interest to a zero IF or baseband, low pass filtering to reject images from the mixing process and sample rate reduction or decimation. Describing these tasks in term of mathematical equations will give us the first step of describing any N channel system.
If we assume that $K$ represents an equally spaced frequency channels existed in the signal spectrum, the local oscillator can be defined as

$$W_{kn}^n = e^{\frac{j2\pi nk}{K}}$$  \hspace{1cm} (3.1)

Where $k$ is the channel number, $K$ is the number of evenly distributed channels, $n$ is the time sample. Then the expression of the output signal in time domain can be derived as follow:

$$w_k(n) = [x(n) e^{\frac{j2\pi nk}{K}} ] * h(p)$$  \hspace{1cm} (3.2)

Where the mixed product is filtered by $h(p)$ prior to decimation. Writing the convolution shown in eq. (3.2) in term of summation, will result in the following:

$$w_k(n) = \sum_{p=\infty}^{\infty} h(p) x(n-p) e^{\frac{j2\pi(n-p)k}{K}}$$  \hspace{1cm} (3.3)

The signal described in eq. (3.3) is going through the decimator. The output signal of the decimator can be described as:
\begin{align}
y_k(m) = w_k(m,M) &= \sum_{p=-\infty}^{\infty} h(p) x(mM - p) e^{\frac{j2\pi(mM - p)k}{\kappa}} \\
\text{(3.4)}
\end{align}

An equivalent form is

\begin{align}
y_k(m) = w_k(m,M) &= \sum_{p=-\infty}^{\infty} \left[ h(p) e^{\frac{-j2\pi pk}{\kappa}} \right] x(mM - p) e^{\frac{j2\pi mMk}{\kappa}} \\
\text{(3.5)}
\end{align}

For cases where M=K, this can be simplified as

\begin{align}
y_k(m) = w_k(m,M) &= \sum_{p=-\infty}^{\infty} \left[ h(p) e^{\frac{-j2\pi pk}{\kappa}} \right] x(mM - p) \\
\text{(3.6)}
\end{align}

To get a clear picture of what is happening to the input FDM signal to the channelizer based on the equations described before, we can look at Figure 3.15 which explains the spectrum of the input signal after each processing step.
Implementing this single channel stage will consist of several add and multiply operations between the input data samples and the filter coefficients as shown in Figure 3.16. In this case the filter coefficients are assumed to be real, while the input data samples are complex (In-phase $I$ and quadrature components $Q$) due to complex mixing.
Figure 3.16: Conceptual digital filter.

The decimated output of the down conversion can then be performed by simply discarding the unwanted output samples from memory. There is still the task of implementing the complex heterodyne. This can be accomplished by direct digital synthesizer and hardware multipliers or by using CORDIC. When there are $k$ channels, the number of resources required is quite significant. But when the channels are evenly spaced, further simplification can be achieved.

Defining a causal filter as follow:

$$ h(p) = \begin{cases} 
0 & p < 0 \\
\hat{h}(p) & 0 \leq p \leq k\lambda \\
0 & k\lambda \leq p 
\end{cases} \quad (3.7) $$

If we use a change of variable and assume that $p = rk + p$, where $r$ goes from 0 to $\lambda -1$, and $p$ goes from 0 to $K-1$, then the filter can be constructed as a two dimensional matrix in term of $K$ rows and $\lambda$ columns.
This two-dimensional structure of the filter allows the convolution sum to become a double summation in $r$ (columns) and $p$ as follow:

\[
y_m(r) = \sum_{p=0}^{K-1} \sum_{r=0}^{\lambda-1} h(rK + p) x(mM - rK - p) e^{j2 \pi (mM - rK - p)k \over K}
\]

(3.9)

\[
y_m(r) = \sum_{p=0}^{K-1} \sum_{r=0}^{\lambda-1} h(rK + p) x(mM - rK - p) e^{j2 \pi (mM - p)k \over K} e^{j2 \pi rKk \over K}
\]

(3.10)

The last term in eq. (3.8) is equal to one, allowing the equation to be further simplified as:

\[
y_m(r) = \sum_{p=0}^{K-1} e^{j2 \pi (mM - p)k \over K} \sum_{r=0}^{\lambda-1} h(rK + p) x(mM - rK - p)
\]

(3.11)

This defines $K$ unique $\lambda$ length filters that must be multiplied and summed. Using the columns and rows to define the filter coefficients, the $x$ input data can be structured as:

\[
x(mM - rK - p) = x_p(mM - rK) =
\]

\[
\begin{bmatrix}
x(mM) & x(mM - K) & \cdots & x(mM - (\lambda - 1)K) \\
x(mM - 1) & x(mM - K - 1) & \cdots & x(mM - (\lambda - 1)K - 1) \\
\vdots & \vdots & \ddots & \vdots \\
x(mM - K + 1) & x(mM - 2K + 1) & \cdots & x(mM - \lambda K + 1)
\end{bmatrix}
\]

(3.12)
Based on the new two dimensional structures of the signal and the filter coefficients, the output signal can be written as:

\[ y_k(m) = W_{km}^k W_{km}^{k-1} \sum_{p=0}^{K-1} \sum_{r=0}^{K-1} h_p(r) x_p(m.M - rK) \]  

(3.13)

The second multiplication and summation of the K filter outputs can be recognized as the kth output of an inverse DFT. To form all the other k channel outputs, a full inverse DFT can be performed. In summary, the output equation consists of a polyphase filter processing with a filter dimension of \( K \times \lambda \), an inverse DFT, and a post multiplication factor.

The relation between the number of frequencies K (frequency bins) and the decimation rate M plays an important role on the channelizer implementation. This relation can be written as \( a = \frac{K}{M} \). Generally speaking, the original frequency band containing the signals that are going to be demodulated will be divided into K segments. To avoid aliasing in the output signal, the filter, decimation rate and the frequency bins have to be defined appropriately. Assume that K=M then the decimated signal bandwidth will be related to the bin width. To give the decimated signal a bandwidth greater than the bin width, then K has to be greater than M such as K=2M which means the decimated output signal will have a bandwidth equal to twice the frequency bin width. The relation between M and K can be used in eq. (3.12) and eq. (3.13). If we do so and assume a=1 (K=M), then the final phase multiplication will be 1 and eq. (3.13) will be:

\[ y_k(m) = \sum_{p=0}^{K-1} \sum_{r=0}^{K-1} h_p(r) x_p(m-r) \]  

(3.14)
The block diagrams of the PFBC based on eq. (3.16) is shown in Figure 3.17. This is the simplest and most direct form of the PFBC. When the signal frequency spacing and bandwidth allow, it is the preferred architecture.

![Block diagram of the PFBC](image)

**Figure 3.17:** IDFT based polyphase channelizer.

For the more generalized case where M and K are different, then the structure of the filter will stay the same but the structure of the input data matrix signal will be different and no longer from the simple commutator. Now the input signal must be continuously reformed in K length columns, although only M samples are input at one time instance so that:

\[
x(mM - rK - p) =
\begin{bmatrix}
x(mM - 0) & x(mM - K - 0) & \ldots & x(mM - (\lambda - 1)K - 0) \\
x(mM - 1) & x(mM - K - 1) & \ldots & x(mM - (\lambda - 1)K - 1) \\
\vdots & \vdots & \ddots & \vdots \\
x(mM - K + 1) & x(mM - 2K + 1) & \ldots & x(mM - \lambda K + 1)
\end{bmatrix}
\]

(3.18)
In this case the output signal in time domain will be:

\[ y_k(m) = W_k^{mkM} \sum_{p=0}^{K-1} W_k^{-pk} \sum_{r=0}^{j-1} h(rK + p)x(mM - rK - p)W_k^{-rk} \]  \hspace{1cm} (3.19)

We can define the polyphase elements as:

\[ PP_p(m) = \sum_{r=0}^{j-1} h(rK + p)x(mM - rK - p) \hspace{1cm} \text{for } k = 0: K - 1 \]  \hspace{1cm} (3.20)

The data matrix has to be reformed in each iteration where new M samples will be added and the oldest M samples get removed. This can be accomplished with a circular buffer, where the newest set of M samples is continuously written and the data matrix is restructuring “backwards” in time from the new written vector. The output equation after the definition of the polyphase elements can be re-written as:

\[ y_k(m) = \sum_{p=0}^{K-1} W_k^{-pk} W_k^{mkM} [PP_p(m)] \]  \hspace{1cm} (3.21)

The twiddle factor matrix of the DFT is defined as:

\[ W_k^{-nk} = \begin{bmatrix}
W_k^{-0} & W_k^{-0} & \cdots & W_k^{-0} \\
W_k^{-0} & W_k^{-1} & \cdots & W_k^{-1(K-1)} \\
\cdots & \cdots & \cdots & \cdots \\
W_k^{-0} & W_k^{-(K-1)} & \cdots & W_k^{-(K-1)(K-1)}
\end{bmatrix} \]  \hspace{1cm} (3.22)

The DFT is also inherently circular in nature, as each column or row can be extended by restarting at the top (column) or end (row) as appropriate. Realizing that the DFT output must be multiplied by a phase offset based on these phase factors, the twiddle
factor with a complex shift in the output equation can be achieved by circularly shifting the input data prior to performing the DFT as follow:

\[
y_k(m) = \sum_{p=0}^{K-1} W_k^{-(p-mM)k} \ [PP_p(m)]
\] (3.23)

\[
y_k(m) = \sum_{r=-mM}^{K-1} W_k^{-rk} \ [PP_{r+MM}(m)]
\] (3.24)

Generally speaking, if M and K are not equal then we need to use two circular buffers, one before the filter bank and another one after it as shown in Figure 3.18.

![Diagram of Polyphase Channelizer with two circular buffers.](image_url)

Figure 3.18: Polyphase Channelizer with two circular buffers.

The general block diagrams of the polyphase channelizer when M=K and when M and K are different are shown in Figures 3.19 and 3.20 respectively.
3.5.2 Design Options

In the PFBC, the input FDM signal passes through a filter bank to compute polyphasic components that are complex-weighted and combined to form multiple outputs using DFT. The outputs of the DFT represent the individual signals contained in each channel at the desired output sample rate [52]. Based on whether the decimation rate is an integer or rational fraction, circular buffers ahead of the filter banks and/or DFT may be required as explained in the previous section. These buffers have been used for data shuffling to avoid any additional phase shifting or complex multiplication required for non-integer decimation [12, 52]. Based on data shuffling, distributing data among filters, and filtering tasks, there are often different design options available for implementing the signal processing operations of PFBC. However, for most
implementations, a minimum computational approach can be defined that most often results in the most efficient hardware or software implementation.

As a study case, assume that we have a signal containing 50 FDM channels as shown in Figure 3.21. The center to center frequency between these signals is 192 kHz. These signals containing symbols modulated at 128kHz by a square root Nyquist filter with 50% roll off factor. This study case is taken directly from Fredric Harris’ textbook [52] but it can be generalized to any number of channels. Our goal is to channelize all the 50 channels into baseband with an output sample rate of 256 ks/s which is two samples per symbol.

![Figure 3.21: Input signal with 50 FDM channels [12].](image)

The first task in the processing is to choose the transform size which should be greater than the number of channels. The product of the transform size and the spacing between the channels defines the sample rate. These extra channels in the transform will be allocated to the transition bandwidth of the analog anti-aliasing filter. Choosing a larger transform size will reduce the cost of these filters by allowing a wider transition band, but the input sample rate will increase based on eq. (3.25) which is:
\[ f_s = N \Delta f \]  

(3.25)

Where \( N \) is the number of channels, \( f_s \) is the input sample rate, and \( \Delta f \) is the bandwidth plus the transition width of the input FDM channel. More detail on how to select the transform size can be found in Harris textbook [52].

The transform size in this study was selected to be 64 to channelize the 50 channels. Hence the input sample rate will be 64 times 192 kHz which is 12.288 MHz. These samples are complex samples formed from either a digital down conversion and resampling from IF or baseband block conversion. The ratio between the input and output sample rate is the sampling ratio which is 12.288 MHz/192 kHz (48:1). This channelization task can be achieved by a variety of options such as the channelization of each channel alone as shown in Figure 3.22. This block diagram can perform the tasks that we are looking for, but for one channel only. Hence, to channelize all the 50 channels, 50 blocks like this one are needed, which makes the implementation less efficient.

\[ W_M^{nk} \]

Figure 3.22: Single channel channelizer.
Fredric Harris textbook [52] describes five design options to solve the problem. This dissertation will describe design option 5 which is the most efficient one in term of computation and uses the previously defined structure and algorithm. The same study case can be generalized to higher numbers of channels with minor changes only. We do believe that increasing the number of channels and/or data size can improve the acceleration speed up.

3.5.2.1 Detailed Design

In this design, the FDM signal will be channelized with a 48-to-1 down sampler into 64 path polyphase match filter to get a per channel sample rate of 256 kHz, which is the required output sample rate. Two circular buffers are required in this design because the down sampling rate and the number of channels are different. The block diagram of this design is shown in Figure 3.23. This design option matches the derivation previously described. The first circular buffer forms the data matrix described in Eq. 3.18; then there are 64 polyphase filter elements with the outputs feeding the second circular buffer. This buffer alternately shifts by 0, 16, 32, and 48 samples prior to the 64-point inverses DFT.
3.5.3 Implementation Method

Implementation was performed in two steps; first, a full PFBC MATLAB [53] simulation was developed in order to validate the mathematical operation, generate test data, and create desired inputs and outputs; and second, CPU-based and General Purpose Graphical Processing Unit (GPGPU) based implementations of the signal processing elements were generated and tested. The system design parameters of the simulation were selected based on the Harris textbook [52] but may be readily generalized for other applications as well, such as [12]. The text based design parameters call for 50 contiguous signal channels that have a frequency spacing of 192 kHz. The signal plan involves digital symbols being transmitted at 128 kilo-symbols per second. The receiver requires a Nyquist filter with a 50% roll-off factor. The filter bank output rate is 256 kHz or 2 samples per symbol. The selected implementation defines an input sample rate of 12.288 MHz, sufficient for the 9.6 MHz channel signal band and guard bands. For the
testing, a simulated input signal consisting of 36864 samples or approximately 3 msec was used. This test signal, consists of tones distributed in specific channels, is shown in Figure 3.24.

The MATLAB signal processing code was used as a template to generate the CPU based implementation model of the PFBC, where data files were imported and exported between C and MATLAB to transfer the test signal and filter coefficients from MATLAB to C, and after execution, transfer the results back to MATLAB for comparison, validation and visualization purposes. The C model of the PFBC was used for computation rate comparison purposes, and it was also useful for creating GPU based implementations.
For the GPU-based model, two GPU systems with the specifications shown in Figures 3.25 and 3.26, respectively, were used. CUDA was used as the software programming language in the GPU-based model. In order to achieve the best performance in term of computation time, optimal use of fast memories is required.

The GPU model of the PFBC consists of two kernels controlled by the CPU. The first kernel is responsible for computing inner products and circular shift, while the second one is responsible for the DFT. We did not use the available CUFFT library to compute DFT, since simple matrix vector multiplication provides a very high degree of parallelism and possibly a better performance for the small size DFTs.
Device 0: "GeForce GT 520M"
CUDA Driver Version / Runtime Version 6.5 / 6.5
CUDA Capability Major/Minor version number: 2.1
Total amount of global memory: 1024 MBytes (1073741824 bytes)
< 2) Multiprocessors, < 48) CUDA Cores/MP: 96 CUDA Cores
GPU Clock rate: 1930 MHz (1.93 GHz)
Memory Clock rate: 800 MHz
Memory Bus Width: 64-bit
L2 Cache Size: 131072 bytes
Maximum Texture Dimension Size (x,y,z): 1D=(65536), 2D=(65536, 65535), 3D=(2048, 2048, 2048)
Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers
Total amount of constant memory: 65536 bytes
Total amount of shared memory per block: 49152 bytes
Total number of registers available per block: 32768
Warp size: 32
Maximum number of threads per multiprocessor: 1536
Maximum number of threads per block: 1024
Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
Max dimension size of a grid size (x,y,z): (65535, 65535, 65535)
Maximum memory pitch: 2147483647 bytes
Texture alignment: 512 bytes
Concurrent copy and kernel execution: Yes with 1 copy engine(s)
Run time limit on kernels: Yes
Integrated GPU sharing Host Memory: No
Support host page-locked memory mapping: Yes
Alignment requirement for Surfaces: Yes
Device has ECC support: Disabled
CUDA Device Driver Mode (TOC or WDDM): WDDM (Windows Display Driver Model)
Device supports Unified Addressing (UVA): No
Device PCI Bus ID / PCI location ID: 1 / 0
Compute Mode:
< Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5, CUDA Runtime Version = 6.5, NumDevices = 1, Device0 = GeForce GT 520M

Figure 3.25: System 1 specifications.
Figure 3.26: System specifications.

To reduce the time consumed in data transfer, it is better to allocate pinned memory for the data to be channelized and the resulting output signals and perform data shuffling on the CPU side. The way we computed DFT is as follows:

The DFT of any signal $x$ of size $N \times 1$ is:

$$X = W \cdot x$$  \hspace{1cm} (3.26)
where $X$ is $N \times 1$ signal representing the DFT of $x$, and $W$ is the twiddle factor matrix of size $N \times N$. Each element of $W$ matrix can be computed as follow:

$$W_{jk} = e^{-i \cdot 2\pi \cdot j \cdot k / N} \quad (3.27)$$

where $j$, $k$ are the row and column indices respectively, and $i$ is the square root of $-1$. Since any complex number ($y$) consists of real and imaginary part as follow:

$$e^{-i \cdot y} = \cos(y) - i \cdot \sin(y) \quad (3.28)$$

Then the twiddle factor matrix can be separated into two matrices where the first one contains the real part and the second one has the imaginary part of the original $W$ matrix. These matrices will be saved in the constant memory of the device to get fast access since they are needed every time we compute a DFT. The outputs of the filter bank which are going to be the input of the DFT will be saved in shared memory. The implementation algorithm of the GPU-based model is shown in Figure 3.27.

In this algorithm, the $k$ loop represents the main loop which is on the CPU side. Inside this loop, there are the data shuffling process; which will be done on the CPU side also; data movement from host to the device, inner products and circular shift kernel, DFT kernel, and movement of the results from the device to the host. To get a fast performance, we used constant memory to save the filter coefficients and shared memory to save the real and imaginary parts of the results of the inner products before and after circular shift. More details on the CUDA code for the channelizer implementation can be found in appendix A.
The channelized output signal using CUDA and GPU based processing is shown in Figure 3.28 which is similar to the channelized signal using CPU based processing and C or MATLAB software.

for k=0 to Number of data blocks

shuffle the data and get sample signals

copy the sample signals to the device

//channelize kernel begin

i=blockIdx.x*blockDim.x+threadIdx.x

if (i< number of channels)

{[

Shared_Mem_r[i]=\sum_{j=1}^{\text{sub滤波器长度}} filtercoeff[j] \cdot signal_r[j]

Shared_Mem_im[i]=\sum_{j=1}^{\text{sub滤波器长度}} filtercoeff[j] \cdot signal_im[j]

]

do circular shift for Shared_Mem_r and Shared_Mem_im vectors

//channelize kernel end

do DFT for the circulated Shared_Mem_r and Shared_Mem_im

copy the results back to the host

di for k

Figure 3.27: Implementation algorithm pseudocode.
3.5.4 Implementation Results

The reference C model of the PFBC requires an execution time of 2.3 seconds on core i5 laptop computer running windows 7 (64-bits) with 4 Gigabyte memory and 2.3 GHz clock speed when the signal length is 36864 samples and polyphase filter banks contain 64 x 25 or 1600 taps. Processing the same size of data requires approximately 250 and 192 milliseconds on system 1 and 2, respectively, taking into consideration the time spent in data movement back and forth between the host and device. If the movement time is ignored, or can be properly pipelined in processing, then the computation time can be reduced to 190 and 150 milliseconds, respectively. This means that the speed up gain that can be achieved with and without data movement is between
9-12 and 12-16, respectively. As the data block size increases, it is expected that the computational speed up factor will also increase. Currently, there is a big time spent in data movement between host and device, with new upcoming GPU architecture, this time can be reduced significantly.

3.6 Summary

An overview of NVIDIA GPU devices and program was provided. And, the algorithmic derivation of the PFBC has been provided with a specific application example that can be generalized for numerous applications. Prior to the implementation described and presented, the applications concept was defined as the progressive advancement of digital signal processing into wireless communication transceivers and eventually multichannel base station devices.

This chapter presented the specific implementation of a 64-channel rational rate PFBC using two GPU systems. While similar research has been performed in our group, this work has advance the structure and implementation in a number of ways. The implementation was done in CUDA without using any standard CUDA library as in other research work. With further investigation, vector based FFT operations for transformed used in this work, and were shown to be superior to the CUFFT implementation usually used in CUDA for transform sizes expected of most communication applications. The rational fraction PFBC defined and demonstrated as compared with in integer rate PFBC incorporates CUDA based input buffer memory manipulations and sequential circular shifting required prior to performing the FFT. These modifications generalize the implementation for the widest range of future implementations.
The parallel implemented models were compared with an efficient CPU-based one using core i5 laptop computer running windows 7 (64-bits) with 4 Gigabyte memory and 2.3 GHZ clock speed. The comparisons indicate that for this simulation an average speed up of 9-16 times faster can be achieved with addition of GPU processing.

With this work and the result presented, the implementation of a multichannel receiver based on the PFBC using NVIDIA GPU devices has been validated. In addition, the inverse mathematical process used for multichannel synthesis algorithms should also be readily possible.
CHAPTER IV
SOFTWARE DEFINED COMMUNITY RADIO USING LOW COST RESOURCES

Community Radio is the third model of radio broadcasting besides the commercial and public systems. It aims to provide individuals with local news and information that are important to a small community [54]. This chapter presents new methods of implementing community radio transmitter and receiver using cost-effective resources, such as PC, USRP, Raspberry PI, RTL-SDR dongle, and antennas based on SDR concept. The idea of building community radio transmitter based on SDR concept has been investigated before but based on my knowledge, no one ever has investigated the implementation of a complete community radio system using these kinds of resources.

4.1 Community Radio

A community radio station is one that functions in a geographical community, for the community, by the community and about the community. They are nonprofit radio stations which are commonly used to deliver local news, entertainment, and information such as agriculture, health, environment, or social commentary like gender equality to the individuals within the community they serve. Although the term “community” often refers to a particular area or place, it may also imply a group of people with similar ethnicity, economic background, interests and goals. Being a subset of radio broadcasting, community radio shares its space alongside the traditional commercial and public radio stations [54]. What distinguishes this type of radio communication system from other media is that it is owned, operated, and influenced by the people living within
the community. The operators and participants of community radio are most often volunteers who are motivated by community well-being, and not by commercial considerations [55].

In many countries, community radios act as a vehicle for local community groups, agencies, voluntary services, and residents to work together to further societal developments. There is no universal definition of community radio – laws and legal framework vary between countries. Numerous internationally recognized principles promote community radio broadcasting [56]. Despite these advances, there are still challenges with regards to regulation, competition with commercial entities for spectrum usage rights, human resource development and sustainability. Of the mass communication media, community radio is the one that is the most cost-effective, universal, as well as the most flexible and immediate. Therefore, from a technical point-of-view, it is essential for organizers to ensure a simple, low-cost and viable community radio infrastructure. Traditional CR implementation ranges from those that utilize high end commercial equipment costing US$ 23,900 – 140,700 to mobile “suitcase radio” type devices costing US$ 3,000 – 5,000 [56]. These hardware-based implementations offer very little in terms of spectrum flexibility, support for new standards and protocols, enhanced services and simplified operation and maintenance. SDR has many of the features desired and described above and may prove to be a superior approach and solution for future community radio platforms.

Important advantages of SDR based systems are that the software can be easily reconfigured and the same hardware can be used for a variety of protocols, standards and
applications. There have been several projects recently focusing on reducing the cost of implementing community radio and a few on utilizing SDR for community radio. A good example to mention in this regard is the paper presented by Gandhiraj and Soman [58]. The next section will present a community radio communication system, both transmitter and receiver, using SDR as well as cost effective resources such a PC, Raspberry PI, USRP, and RTL-SDR dongle. Such a point to multipoint system employs a PC and USRP as a transmitter and multiple RT-SDR dongle with Raspberry-PI as receiver. This system architecture and implementation employing readily programmable GNU Radio software will reduce not only the total cost but also the power consumed by the system, since the power consumed by Raspberry PI is considerably less than that consumed by the traditional computer. The size of the whole system will be very small as compared with other system implementations.

4.1.1 Community Radio Technical Concept

In comparison with commercial broadcasting, community radio often uses the basic production and transmission equipment appropriate for the size, needs and capability of the community. One can find a diverse range of technologies applied to CR broadcasting, ranging from simple and low-cost to complex and expensive. Major issues that dominate the role of technologies applicable to community radio are:

4.1.2 Types of Modulation

The common modulation scheme in community radio is angle modulation. It is very well-known that there are two kinds of angle modulation: FM and PM. Both are considered as a nonlinear modulation [59]. Angle modulation is widely used in
commercial FM radio broadcasting, NTSC TV audio broadcasting, and other point-to-
point communication systems. For a message signal \( m(t) \), the frequency modulated signal \( u(t) \) will be [59]:

\[
  u(t) = A_c \cos(2\pi f_c t + 2\pi k_f \int_{-\infty}^{t} m(t) \, dt) \tag{4.1}
\]

where \( A_c \) and \( f_c \) are the amplitude and frequency of the carrier signal and \( k_f \) is the frequency deviation. The phase modulated signal is similar to the frequency modulated signal but without integration. Assuming that the same message signal \( m(t) \) is modulated using PM, then the phase modulated signal will be [59]:

\[
  u(t) = A_c \cos(2\pi f_c t + 2\pi k_p m(t)) \tag{4.2}
\]

where \( k_p \) is the phase deviation. FM is more common than PM even if they look almost the same. Figure 4.1 shows AM, FM and PM modulated signals. If the message signal is a sinusoidal signal \( m(t) = a \cos(2\pi f_m t) \), then the frequency modulated signal will be:

\[
  u(t) = A_c \cos(2\pi f_c t + \beta_f \cos(2\pi f_m t)) \tag{4.3}
\]

where \( \beta_f = \frac{k_f a}{f_m} \) is the modulation index. Equation (4.3) can be represented by:

\[
  u(t) = \sum_{n=-\infty}^{\infty} A_c J_n(\beta) \cos((2\pi f_c + n f_m) t) \tag{4.4}
\]

where \( J_n \) is the Bessel function of the first kind of order \( n \). The bandwidth of the modulated signal based on equation (4.4) is unlimited, but the effective bandwidth which contains the significant power of the modulated signal can be estimated using Carson’s rule [59]:

\[
  B_T = 2(\beta_f + 1)W \tag{4.5}
\]
where BT is the bandwidth of the modulated signal while W is the bandwidth of the message signal. Based on the bandwidth used to transmit the signal, there are two kinds of FM modulation, Narrow Band and Wide Band FM (NBFM and WBFM).

![Graph showing AM, FM, and PM modulated signals.](image)

Figure 4.1: AM, FM, and PM modulated signals.

### 4.1.3 CR Transmitter

The transmitter is the core piece of equipment in a CR setup. CR stations typically use low power FM transmitters with geographical coverage from 0 – 35 km. It should be mentioned here that the power of the transmitter is not solely responsible for the coverage range, other factors like efficiency, antenna terrain and atmospheric conditions also play a crucial role. However, a rough estimate of the coverage achieved with different power levels is provided by United Nations Educational, Scientific and Cultural Organization (UNESCO)[55] and is reproduced in Table 4.1. For a small community, the recommended power level is 20-watt. Additional technologies like a linear or power amplifier may be employed to boost the power level of the transmitter. It should be noted
that certain countries like India have limitations on the transmitted power level while others like Australia have no limitations as far as CR is concerned.

Table 4.1: Estimated reach of different transmitter power.

<table>
<thead>
<tr>
<th>Power</th>
<th>Class A</th>
<th>Class B</th>
<th>Class C</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 Watts</td>
<td>0 – 4 km</td>
<td>4 – 6 km</td>
<td>7 – 15 km</td>
</tr>
<tr>
<td>20 Watts</td>
<td>0 – 6 km</td>
<td>6 – 8 km</td>
<td>8 – 15 km</td>
</tr>
<tr>
<td>50 Watts</td>
<td>0 – 8 km</td>
<td>8 – 14 km</td>
<td>14 – 25 km</td>
</tr>
<tr>
<td>100 Watts</td>
<td>0 – 10 km</td>
<td>10 – 20 km</td>
<td>20 – 35 km</td>
</tr>
</tbody>
</table>

4.1.4 CR Receivers

An aspect of community radio that is often overlooked is the receivers. This is due to the fact that traditional CR implementations have fixed signaling format combined with the ubiquity of low-cost FM receivers. If fully reconfigurable SDR transmitters and receivers are investigated, then low-cost, low-power computing platforms like Raspberry PI and USB based SDR modules may be useful and required.

4.1.5 Antenna

Antenna design plays another crucial role in the coverage and quality of community radio implementation. Various antenna configurations and designs could be selected or fabricated; one of the simplest and most common is the folded dipole. Since FM is the preferred method of modulation, the height of the antenna is an important factor that determines signal reach. A primary transmitting antenna of approximately 20 to 30 meters in height fabricated from galvanized iron water pipes will be adequate for a
wide range of scenarios. On the receiver side, a quarter wavelength whip antenna is the most suitable for FM.

4.2 Simulation and Implementation of Communication Systems Using SDR Concepts

Before GNU Radio software and SDR concept existed, the most common way of simulating communication systems was MATLAB SIMULINK [53]. MATLAB SIMULINK gives a clear picture of how communication systems work and how they can be implemented. But unfortunately, this kind of simulation is very close to the theoretical concept than the real-time situation. MATLAB provides a variety of SIMULINK blocks that can be used to simulate almost any kind of communication systems from the simple ones such as AM modulation as shown in Figures 4.2-4.3 to the most complex ones such as Orthogonal Frequency-Division Multiplexing (OFDM).

![Diagram of AM modulation in MATLAB SIMULINK](image)

Figure 4.2: AM modulation in MATLAB SIMULINK.
Figure 4.3: AM modulated signal using MATLAB SIMULINK.

Besides MATLAB SIMULINK, MATLAB coding can also be used to describe all kinds of communication systems. Nowadays, MATLAB toolboxes have seen significant advancement and progress which make the simulation process even easier than before. In addition to MATLAB (coding and SIMULINK), other software such as C, C++, and Fortran have been used as well for the simulation of communication systems.

GNU Radio has a lot of advantages as compared with the simulation software described before. The first one is that GNU Radio is free software which makes it accessible by everyone wherever he/she lives. The other important point is that the simulated system can be readily employed as software based processing for a real-time system by adding an RF front end. With other software, this process may not be as simple, easy and direct.
As an example, to simulate AM modulation and demodulation systems using GNU Radio, the system flow graph shown in Figure 4.4 can be used, which is a simulated AM-Double Side Band (AM-DSB) system. Then, to convert this simulation system into real time AM transmitter and receiver systems, a few things have to be done. First, RF front ends must be added before the channel for the transmitter side and after the channel for the receiver side. The second change is to separate the single simulation to a separate transmitter and receiver. Based on these changes, the transmitter side will look like Figure 4.5 while the receiver side will look like Figure 4.6.
Figure 4.4: Simulation of AM-DSB transmitter and receiver using GNU Radio.

Figure 4.5: AM-DSB transmitter using GNU Radio.
Figure 4.6: AM-DSB receiver using GNU Radio.

For digital communication, GNU Radio has even more powerful blocks that make the simulation and implementation very easy as compared to other software tools. Figures 4.7-4.9 explain the simulation flow graphs of PSK, FSK, and Quadrature Phase Shift Keying (QPSK). To turn these flow graphs into real time transmitters and receivers, the same procedures illustrated for AM-DSB should be followed.
Figure 4.7: Simulation of PSK transmitter and receiver using GNU Radio.

Figure 4.8: Simulation of FSK transmitter and receiver using GNU Radio.
One of the biggest advantages of using GNU Radio as a simulation and implementation tool is the capability it gives us to see what happens to the transmitted signal at each step, even in real time implementation. Let’s assume that we decided to build QPSK transmitter and receiver systems based on the flow graph shown in Figure 4.9. The transmitter and receiver systems will look like Figure 4.10 and 4.11 respectively. The GUI tools available in GNU Radio give us the opportunity to see the transmitted signal in time and frequency domains as shown in Figure 4.12. We also can see the received signal and its constellation as shown in Figure 4.13. There is no other software that provides all these tools for free as GNU Radio.
Figure 4.10: QPSK transmitter using GNU Radio.

Figure 4.11: QPSK receiver using GNU Radio.
Figure 4.12: QPSK modulated signal and its spectrum.

Figure 4.13: Received QPSK signal in time domain and its constellation.
4.3 Proposed Implementation Methods of CR System

The SDR CR system prototyping and implementation method include two main experiments. The first one is the implementation of a CR transmitter using PC and USRP N210. At the same time an SDR based receiver has to be built to receive the signal and verify the operation of the transmitter system. The experimental setup of the CR transmitter which consists of PC and USRP N210 is shown in Figure 4.14, while the receiver system which consists of Raspberry PI and RTL-SDR is shown in Figure 4.15.

The second experiment is the implementation of CR transmitter using USRP B100 and Raspberry PI. The experimental setup of this transmitter is similar to Figure 4.15, but instead of the RTL-SDR dongle, a USRP B100 was used. An SDR based receiver has to be built to validate the functionality of the Raspberry PI transmitter, or we can use RSA 3303A real-time spectrum analyzer to receive the transmitted signal.

Figure 4.14: Community radio transmitter using PC and USRP N210.
4.3.1 Implementation Resources

The descriptions of the main three devices used in these experiments are as follow:

The first device is the USRP. USRPs are a family of radio platforms developed by Ettus Research [60]. All the devices in the USRP family consist of an FPGA with various digital interfaces, ADC, DAC, and an array of RF daughter cards (see Table 4.2). The flexibility these devices provide combined with their relative low cost is why the USRP is widely used as a research platform for wireless communications. The USRP architecture is designed in such a way that most of the resources like signal processing blocks, filters, and resources on its motherboard can be configured, controlled and reprogrammed through software.
Table 4.2: USRP products [60].

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Bandwidth MHz</th>
<th>Freq. range (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>USRP N series</td>
<td>SDR</td>
<td>40</td>
<td>–</td>
</tr>
<tr>
<td>USRP B series</td>
<td>SDR</td>
<td>32</td>
<td>–</td>
</tr>
<tr>
<td>USRP E series</td>
<td>SDR</td>
<td>56</td>
<td>70 – 6000</td>
</tr>
<tr>
<td>SBX</td>
<td>RF card</td>
<td>40 – 120</td>
<td>400 – 4400</td>
</tr>
<tr>
<td>Basic TX/RX</td>
<td>RF card</td>
<td>–</td>
<td>1 – 250</td>
</tr>
<tr>
<td>LF TX/RX</td>
<td>RF card</td>
<td>–</td>
<td>0 – 30</td>
</tr>
</tbody>
</table>

The choice of the appropriate USRP motherboard and daughter card combination depends on the frequency range requirements, cost and processing complexity. A USRP B100 series device with the basic TX/RX card covers the typical FM frequencies used in most countries. Figure 4.16 shows the block diagram and picture of the USRP B100 device. The B and the N series models require a PC with Linux (preferred) or Windows OS and the open source GNU Radio [61] software for implementing the signal processing flow graphs. If a standalone type of system is required, then Ettus provides the embedded E series devices with an integrated RF card and dual-core Arm A9 processor.
The transmitting power of the USRP is about 100 mW [60], hence an external power amplifier is required to boost the transmitted signal power to respectable levels. There are commercially available power amplifiers in the US$ 300 – 500 range that are suitable for performing this function. Combined with the cost of the USRP and the antenna, a simple CR radio station can be set up at the cost of US$ 1500 – 4000.
The second device that is used in the experiments is Raspberry PI 2, as shown in Figure 4.17. It has been used recently for the implementation of a wide range of applications that include both wired and wireless sensor system, Internet of Things (IOT) devices and other examples [62-63]. The great advantage in using a Raspberry PI is the small size, low cost, low power consumption, and LINUX operating system. For developers with a goal of implementing a specific system or application, and where the factors mentioned above are important, then a Raspberry PI is an excellent candidate. In many community radio applications, cost and power consumption are two of the most important factors. Therefore, one of our implementations is based on the Raspberry PI.

Figure 4.17: Image of raspberry PI 2.
The third component in the proposed systems is the RTL-SDR dongle. It is originally a DVB-T tuner but has been utilized as a software defined radio RF front end platform. The block diagram of an RTL-SDR from Nooelec is shown in Figure 4.18.

Figure 4.18: RTL-SDR block diagram.

As explained before, community radio typically involves an FM transmitter/receiver. Therefore, to demonstrate the feasibility of the proposed architecture based on an existing signal format, an FM transmitter was designed using the open source GNU Radio software.

4.3.2 Implementation Results

For the first experiment, a 915 MHz frequency was selected as the frequency of operation, since it is in the ISM (Industrial, Scientific, and Medical radio band), and it meets the specification of the USRP N210. The CR transmitter and receiver flow graphs are shown in Figures 4.19 and 4.20 respectively. The transmitted signal in both time and frequency domain are shown in Figures 4.21 and 4.22 respectively. The transmitted signal was originally an audio signal which was received without errors. For the second experiment, a 40.68 MHz frequency was used as the frequency of operation since it is
also designated for ISM, and it meets the USRP B100 specifications. The GNU Radio
transmitter and receiver were similar to those in the first experiment but with different
frequency of operation. A snapshot of the received signal on the RSA 3303A real time
spectrum analyzer is shown in Figure 4.23.

Figure 4.19: CR transmitter flow graph.

Figure 4.20: CR receiver flow graph.
Figure 4.21: Transmitted signal in time domain.

Figure 4.22: Transmitted signal in frequency domain.
4.4 Summary

This chapter presented an overview of community radio systems as they exist throughout the world, typical operation, signaling formats and dedicated receivers and transmitters used. This work then focused on using SDR techniques and tools to define and demonstrate a fully functional prototype system using readily accessible, low cost resources. Several examples of simulation and implementation of SDR based communication systems suitable for community radio were presented. Multiple SDR implementations of community radio transmitters and simple, low cost receivers were defined and demonstrated. The implementations were successful as the results illustrated. The devices and components used for the implementation of community radio were PC with Linux OS, USRPs, RTL-SDR dongle, and Raspberry PI.
While SDR techniques and hardware have been previously studied, and investigated, this work is the first to define and perform prototype implementation of a complete, viable, low cost SDR community radio communication system. The system architecture is being investigate as a rapidly deployable wireless community radio information system that can be used during natural or man-made disasters where normal communications infrastructure has been destroyed or does not exist. In these instances, the point to multipoint communication using a single PC laptop and USRP to multiple RTL-dongle and Raspberry-Pi receivers with battery or small solar cell power systems can be quickly configured based on government or regional restrictions and distributed to those in need.

As an extension to this system approach, a GPU based multichannel polyphase synthesis implementation, the algorithmic inverse of the PFBC discussed in the previous chapter, could provide multiple simultaneous transmission of either differing information messages or even multiple languages as necessary. The final step in a multichannel; two-way communication system would be the definition of an alternate to the USB-dongle RTL and Raspberry Pi that could both receive and transmitted. Developing a small, low cost transmitter for this application is described in the next chapter.
CHAPTER V

FPGA BASED DIGITAL MODULATORS FOR SDR APPLICATIONS

SDR has become a widely-used method in implementing different kinds of today’s communication systems. The advantages of SDR-based systems have become more apparent after the tremendous development in flexible, low-cost hardware terminals. These reconfigurable terminals’ hardware such as USRP from Ettus Research define the main components required in any SDR-based system. The USRP itself consists of RF receiving and transmitting mixers and filter modules, DAC, ADC, and FPGA. Based on the fact that FPGA does most of the digital signal processing and computer interface work in any SDR-based system, this chapter investigates the possibility of building SDR-based systems minimizing RF electronics and focusing on the FPGA and other low-cost electronics. The idea of building SDR-based components using FPGAs has been investigated in several research papers but almost all of these papers have used either Xilinx System Generator or DSP Builder Tools for the implementation tasks, which mean significant use of IP-Blocks. Therefore; this chapter presents several novel, IP free methods of implementing both simple and more advanced forms of modulators using FPGA.

The FPGA modulators have been defined and prototyped using Digital Zybo Trainer Board [66] that hosts a Xilinx Zyboq-7000 ARM/FPGA device. By Xilinx combining on a single IC an FPGA and ARM processor, future development of a complete Linux OS based computer with digital modulator and demodulator capability becomes possible. Before going to the FPGA and modulator implementation detail, some
concepts have to be explained such as FPGA programming and digital modulators as the next few sections will illustrate.

5.1 Why use FPGA?

FPGA is an integrated circuit consists of a large number of logic gates which provide digital inversion (INV), AND or OR functions. It can be configured by customers and design engineers. FPGA configurations are usually done using Hardware Description Language (HDL) such as Verilog and VHDL. There are several FPGA vendors but the most well-known are Xilinx and Altera, recently purchased by Intel and now known as Intel Programmable Solutions Group.

5.2 FPGA Programming

The main two methods of programming FPGAs are VHDL and Verilog. Both methods are fully supported and capable of designing FPGAs and custom ASICs. There are differences between VHDL and Verilog, but designers can choose either one that is available or preferred. To create a project and implement it on FPGA development board, you need several software tools or a complete design suite, for example Xilinx Vivado [64] and SDSoC [65], Digilent Zybo documentation and resources [66], and Mentor Graphics ModelSim [67] which were readily available to support this work. Table 5.1 shows the difference in defining a simple AND gate using VHDL and Verilog.
5.3 Digital Sine Wave

The main component in any analog or digital modulation is the sine wave carrier since the modulation process depends heavily on the sine wave generation. Even though the main goals of the chapter are the implementation of different kinds of digital communication starting from the simple modulation types such as ASK, PSK, FSK, and ending in more complex types of digital modulation such Quadrature Amplitude Modulation (QAM). The first step in implementing these kinds of modulation is the generation of sine waves. To generate a sine wave in VHDL, there are two common methods:

5.3.1 Taylor Series

It is a way of representing mathematical functions as an infinite summation of terms calculated from the function derivatives at any point. The Taylor series of the sine wave is [68]:

Table 5.1: AND gate using VHDL and Verilog.

<table>
<thead>
<tr>
<th>VHDL</th>
<th>Verilog</th>
</tr>
</thead>
<tbody>
<tr>
<td>library ieee;</td>
<td>module andgate (a, b, y);</td>
</tr>
<tr>
<td>use ieee.std_logic_1164.all;</td>
<td>input a, b;</td>
</tr>
<tr>
<td>entity and Gate is is</td>
<td>output y;</td>
</tr>
<tr>
<td>port( a, b : in std_logic;</td>
<td>assign y = a &amp; b;</td>
</tr>
<tr>
<td>f: out std_logic);</td>
<td>end module;</td>
</tr>
<tr>
<td>end and Gate;</td>
<td></td>
</tr>
<tr>
<td>architecture func of and Gate is</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>f &lt;= a and b;</td>
<td></td>
</tr>
<tr>
<td>end func;</td>
<td></td>
</tr>
</tbody>
</table>

5.3.1 Taylor Series

It is a way of representing mathematical functions as an infinite summation of terms calculated from the function derivatives at any point. The Taylor series of the sine wave is [68]:

\[
\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots
\]
\[
\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots = \sum_{n=1}^{\infty} (-1)^{(n-1)} \frac{x^{(2n-1)}}{(2n-1)!}
\] (5.1)

This method is very easy in terms of the concept but not efficient in terms of hardware implementation. Typically, design engineers try to avoid multiplication instructions during the implementation process as they can consume a lot of hardware resources. Unfortunately, Taylor series has many multiplication terms which is why it is not really an efficient way of computing a digital sine wave.

5.3.2 Direct Digital Synthesizer

Direct Digital Synthesizer (DDS) is a common method that has been used for creating arbitrary waveforms from a single fixed higher frequency reference clock [69]. DDS has several applications in communication systems which include signal generation, sound synthesizers, and modulators. The block diagram of DDS is shown in Figure 5.1.

![Block diagram of DDS](image)

Figure 5.1: Block diagram of DDS.
There are two common methods of generating trigonometric functions based on DDS technique: Look Up Table (LUT) and CORDIC. The following subsections will explain these methods in detail.

5.3.2.1 Look Up Table

In the look up table method, one cycle of the sine wave signal has to be stored in a table, and a phase register accumulator can be used to access these values within the table to provide a periodic waveform. When the value reaches the end of the table, modulo arithmetic is used to circle back to the beginning of the table. As the size of the table gets bigger, the trigonometric wave will be more accurate. Knowing that a circle contains 360 degrees, the phases can be defined and distributed equally among the values that can be represented by the accumulator. To make it simple, consider that we have a two-bit accumulator; then “11” will refer to 360, “00” will refer to angle 90 degree, “01” will refer to 180-degree angle, and “10” will refer to 270-degree angle. This concept is valid for any accumulator size. As the size of the accumulator increases, the step size gets smaller which means more accuracy but higher use of digital hardware.

The sine wave values can be generated in MATLAB, Microsoft Excel or any other software; then the computed values need to be stored in a file readable by VHDL or a memory within the VHDL. After that, users can read these values at any time they want.

5.3.2.2 CORDIC

CORDIC method is based on “a special purpose digital computer for real time airborne computation” [70]. This method is an extension based on the original concept
which has been used widely in implementing different kinds of function generators on FPGAs and ASICs including trigonometric, hyperbolic, linear, and logarithmic functions. CORDIC processors implement different types of functions using a sequence of shift and add operations which reduce the required hardware resources significantly.

The functionality of CORDIC processor can be considered as the digital equivalent of an analog resolver [22]. As in the analog resolver, there are two modes of computing, a rotational mode and a vectoring mode. Starting from initial conditions, CORDIC algorithm uses small planar rotation and vectoring \((r, \theta)\) to compute elementary trigonometric functions. During the rotational mode of operation with the coordinate components of a vector \((X, Y)\) and a \(\theta\) rotation angle, the input vector will be rotated by small predefined steps to obtain a new vector \((X', Y')\) as shown in Figure 5.2 [22]. During the vectoring mode, the length \(r\) and the angle \(\theta\) of the vector \((X, Y)\) with respect to the main x-axis will be computed. For this reason, the input vector is iteratively rotated towards the x-axis so that the y-component approaches zero. The summation of the total micro angles must be equal to the angle \(\theta\), while the value remaining as the x-component corresponds to the length \(r\) of the vector \((X, Y)\). More detail on CORDIC processor and how it can be implemented in VHDL can be found in Nagarjun Marappa thesis [22].
5.4 FPGA Based Digital Modulator: Review

Implementation of digital modulators on FPGA is one of the research areas that have received great attention recently. Most of these researches have focused on the implementation of simple digital modulators on FPGAs such as ASK, FSK, and PSK. Quadri and Tete presented a review of the main research papers in this area with some
implementation examples using Xilinx System Generator [71]. Some of this work will be explained here.

Bhore and Sarde presented an implementation of Binary PSK (BPSK) modulator and demodulator on a SPARTAN-3 FPGA. They used MATLAB SIMULINK to build their entire system and Xilinx System Generator to generate the VHDL implementation models [72]. Binary ASK (BASK), Binary FSK (BFSK), and BPSK modulators were implemented on Altera development and education board using Verilog by Erdogan et al [73]. They used DDS to generate the sine wave carrier which had 1KHz frequency. Their system was implemented directly in Verilog without using Xilinx System Generator tools. BPSK was also implemented on a SPARTAN-3 starter kit board by Popescu et al [74]. MATLAB SIMULINK and Xilinx System Generator were the software tools they used to generate the VHDL implementation module.

Chye et al presented a detailed guideline on how to design and implement a BPSK transmitter on Virtex-4 FPGA development board with DAC in P240 analog module. Many software and design tools such as MATLAB SIMULINK and Xilinx System Generator were used to verify the design output in terms of behavior, functionality, synthesis, timing, and constraints area. The implementation was done in Verilog which was generated by the Xilinx System Generator [75].

Rajaram and Gayathre presented an FPGA based implementation of different digital communication schemes such as QAM, BASK, BFSK, and BPSK. SPARTAN-3 FPGA was used as the base of the implementation [76]. The CORDIC algorithm was used to generate the sine wave carrier. The unique thing about this paper is that
everything was done in VHDL, not in MATLAB or using the Xilinx System Generator as in other papers. Zhuan Ye et al extended the concept of software-defined functionalities to radio frequency with a novel "all-digital transmitter"[77]. The all-digital transmitter uses a Pulse Width Modulation (PWM) method such that the RF signal with binary format can be directly synthesized in the digital domain. The low rate of the signal processing of PWM makes it suitable for implementation using FPGA. By combining a universal QAM modulator and an RF pulse width modulator, they demonstrated a real-time QAM transmitter system with digital RF output providing a realistic signaling format.

Henrik Bostrom built an FPGA based digital stereo FM modulator including the necessary signal processing, such as filtering, waveform generation, and stereo multiplexing in his thesis [78]. The implementation code was written in VHDL and a selection of free IP-Blocks. The thesis focused on the area of efficiency, and a number of suggestions were given to maximize the number of channels that can be modulated using a single FPGA. AM, FM, ASK, FSK, and PSK modulation on Virtex-5 FPGA using MATLAB SIMULINK and Xilinx System Generator were also presented by Mangala and Manikandan [79] while Kislal et al implemented an FM demodulator on an Altera DE0 board [80]. The three main parts of the implementation were: quadrature modulation, mixed demodulation, and FIR filter. A CORDIC algorithm was used for the generation of the trigonometric sine carrier. Adiono et al presented an implementation of FM modulator-demodulator on DE2-70 FPGA board using Verilog and Altera Quartus software [81].
Hatai and Chakrabarti implemented FM modulator and digital phase locked loop FM demodulator on a XC2VP30-7ff896 FPGA [82]. The individual components of the proposed FM modulator and demodulator were optimized in such a way that the overall design is high-speed, area optimized, and low-power consumption. The modulator and demodulator contain an optimized Direct Digital Frequency Synthesizer (DDFS) based on a quarter-wave symmetry sine table technique for generating the carrier signal. They extend their work by building a digital phase locked loop FM receiver in VHDL. As built, the system takes 108.67mW of power executing at 100MHz frequency [83].

The successful implementation of simple communication modulators has opened the door to the investigation of possible implementation of more complicated communication systems such as QPSK and QAM. As examples; Gaikwad et al presented an implementation of QPSK transmitter and receiver system on FPGA [84] while Song and Yao presented an FPGA based QPSK modulator using FPGA. They included an FIR filter to clean up the generated QPSK waveform. They used MATLAB to compute the FIR filter coefficients that were stored in ROM memory. Unfortunately, the paper did not provide enough detail in terms of utilizations or implementation [85].

Thombre and Shah presented an efficient way of implementing QPSK modulator using 16 and 64 address locations ROM memory [86]. They depend on the current and next values of I and Q data to produce the next symbol from the symbols stored in the ROM memory. The concept they used is similar to the Feher QPSK modulation technique [87]. The width of their output symbol is 8-bit.
Kazaz et al presented an efficient way of designing and implementing SDR based QPSK modulator on Altera Cyclone IV FPGA [88]. For the QPSK modulator, they used Altera DSP Builder Tools combined with MATLAB Simulink, ModelSim and Quartus II design tools. As a reconfigurable hardware platform, they used Altera DE2-115 development and education board with ADC/DAC daughter cards.

Elamary et al presented a VHDL based implementation of QPSK modulator on Altera board. They used phase shifters to generate four signals from one input sine wave [89]. The signals were used as inputs to a multiplexer which selects one of them based on the message signal. They compare their system with a simulated model developed in MATLAB. The generated discrete QPSK signal was cleaned up by analog filter before the transmission.

Kolankar and Sakhare presented an efficient implementation method, in terms of power consumption, of a QPSK modulator using the Xilinx System Generator [90]. They designed and tested the whole system in MATLAB SIMULINK, and then used Xilinx System Generator to generate the VHDL implementation of the model. Hardware co-simulation, which is a new feature in MATLAB by virtue of Xilinx System Generator, was used to make sure the implemented system is efficient in terms of performance and hardware implementation.

The use of MATLAB SIMULINK, DSP Builder Tools, and System Generator are common among most of these papers. Unfortunately, the designer doesn’t have complete control of the implementation modules when it were generated using System Generator. Besides that, Xilinx System Generator and DSP Builder Tools use proprietary IP-Blocks
in the implementation processes which allows rapid prototyping but typically at a higher cost. In this work, the use of proprietary or tool vendor IP-Blocks has been avoided. We used MATLAB for system and algorithm development as well as the generation of test vectors. The FPGA design implementation was done using VHDL without using any IP-Blocks.

5.5 Implementation of Digital Modulators using FPGA

The simplest forms of digital communication modulators are ASK, PSK, and FSK. This section presents the implementation of these modules on FPGA. Other more complex schemes such as QPSK and QAM were also implemented as we will see later in this section.

The host development board for the implementation of the digital modulators is a ZYBO board from Digilent [91] which is shown in Figure 5.3. The ZYBO is an entry-level embedded platform which is built around a Xilinx Zynq-7000 (Z-7010) family device. The Z-7010 is based on Xilinx Programmable SoC architecture, integrating a dual-core ARM Cortex-A9 processor with a Xilinx 7-series FPGA. The ZYBO board has a rich set of multimedia and connectivity peripherals, including; video and audio I/O, dual-role USB, on-board memories, Ethernet, and SD slot. Besides these peripherals, it contains six PMOD ports with pins connected to the Xilinx device that can be connected to a range of available devices.
By using the ZYBO board and Xilinx Vivado software, a range of modulators were built as the following subsections will explain.

5.5.1 Implementation of ASK Modulator

ASK is considered as the simplest form of digital modulation which can be generated as shown in Figure 5.4. The ASK modulated signal waveform is shown in Figure 5.5. For the ASK modulator implementation, a 24-bit accumulator working on the rising edge of the clock was used. 256 samples of a one cycle sine wave were stored in LUT. Based on this number of samples, the required bus width to reach them is 8-bit. The eight most significant bits of the 24-bit accumulator were used to select the corresponding sample. If we want a smoother signal, we can increase the size of the LUT and the number of the used bits from the accumulator as well.
The message signal was used as a selector in the Multiplexer (MUX) circuit as shown in Figure 5.6. If the message signal is “1” then the output will be some sort of sinusoidal signal. If the message signal is “0” then the output will be 0. This kind of ASK modulation is also known as On Off Keying (OOK).

The width of the output sine wave was selected to be 16-bit. An 8-bit width works also, but we are looking for a smoother signal for analog conversions and filtering. The message signal that was used in the simulation and implementation is a random binary signal generated in MATLAB and saved in a text file readable by VHDL. This
text file can represent any kind of signal or input message that has been converted into a binary format for transmission, such as audio, video, or computer data.

Figure 5.6: Block diagram of ASK(OOK) modulator implementation in VHDL.

Most people do not consider the ASK explained before as really an ASK modulation. In fact, it is well-known as OOK. In OOK the output modulated signal is zero when the message signal is zero, and sine wave with certain amplitude when the message signal is one.

In ASK, the output modulated signal is a sine wave that takes different discrete amplitudes based on the message signal. Implementing this form of BASK is very similar to the previous one. The only change that we need to do is in the MUX circuit by giving it two input sine waves with different amplitudes and the same frequency. These two sine
waves can be obtained from the same accumulator and LUT. The first signal which refers to the message bit “1” is the generated sine wave itself, while the second signal refers to the message bit “0” is the generated sine wave after multiplying its amplitude by (-1). The best way to do that is by using XOR logic gate and reversing the most significant bit of the accumulator as shown in Figure 5.7.

![Figure 5.7: Block diagram of ASK(BASK) modulator implementation in VHDL.](image)

By the same method used for BASK modulation, higher order ASK (M-array ASK) can be implemented also. As an example of these M-array ASK modulators, we implemented a 4-ASK modulator as shown in Figure 5.8. In this scheme, the data have to be converted from serial to parallel one which will work as a selector in the MUX circuit as explained before. The 4ASK modulator uses two bits to represent its symbols. Hence the MUX circuit will have four input signals as shown in the block diagram of Figure 5.8. The four amplitudes we used for the implementation are -1.5, -0.5, 0.5, and 1.5.
After following the detailed descriptions of the implementation explained above and by using Xilinx Vivado, the OOK modulated signal was obtained using ModelSim software. We found that the signals’ waveforms are not readily observable in ModelSim captured displays, hence we exported the results into MATLAB as text files and plotted them for documentation. The generated digital sine wave using 256-bit LUT is shown in Figure 5.9 while the generated OOK output is shown in Figure 5.10.

For the BASK and 4ASK implementation, the generated BASK signal and 4ASK using VHDL based on the implementation described before are shown in Figures 5.11-5.12, respectively.
Figure 5.9: Digital sine wave carrier.

Figure 5.10: ASK (OOK) modulation results.
Figure 5.11: ASK(BASK) modulation results.

Figure 5.12: ASK(4ASK) modulation results.
To download the Xilinx design on the ZYBO board, there are several steps need to be done. The first one, we need to use the board memory instead of the text file that have the LUT used for the simulation. The other important file that needs to be saved in the memory is the file that contains the input message. The second step that we need to do is to write the .xdc file which explains the used input and output terminals. We couldn’t find a terminal that has 16-bit to use it as an output terminal. To solve this problem, we used the JE pmod output which has 8 pins and used the eight most significant bits of the output result just to see the design board utilization. After following these steps and the .xdc shown in Table 5.2, the project design utilization for the implemented ASK (OOK) module is as shown in Table 5.3.
Table 5.2: The xdc of ASK(OOK) modulators

```plaintext
#Clock signal
#IO_L11P_T1_SRCC_35
set_property PACKAGE_PIN L16 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports clk]
#Pmod Header JE
#IO_L4P_T0_34
set_property PACKAGE_PIN V12 [get_ports ask_amp1[0]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[0]]

#IO_L18N_T2_34
set_property PACKAGE_PIN W16 [get_ports ask_amp1[1]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[1]]

#IO_25_35
set_property PACKAGE_PIN J15 [get_ports ask_amp1[2]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[2]]

#IO_L19P_T3_35
set_property PACKAGE_PIN H15 [get_ports ask_amp1[3]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[3]]

#IO_L3N_T0_DQS_34
set_property PACKAGE_PIN V13 [get_ports ask_amp1[4]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[4]]

#IO_L9N_T1_DQS_34
set_property PACKAGE_PIN U17 [get_ports ask_amp1[5]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[5]]

#IO_L20P_T3_34
set_property PACKAGE_PIN T17 [get_ports ask_amp1[6]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[6]]

#IO_L7N_T1_34
set_property PACKAGE_PIN Y17 [get_ports ask_amp1[7]]
set_property IOSTANDARD LVCMOS33 [get_ports ask_amp1[7]]
```
Table 5.3: Device utilization summary of the ASK(OOK) modulator.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>9</td>
<td>17600</td>
<td>0.05</td>
</tr>
<tr>
<td>FF</td>
<td>9</td>
<td>35200</td>
<td>0.03</td>
</tr>
<tr>
<td>IO</td>
<td>9</td>
<td>100</td>
<td>9.00</td>
</tr>
<tr>
<td>BUFG</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
</tbody>
</table>

5.5.2 Implementation of FSK Modulator

In FSK, the modulated signal consists of two sine waves at different frequencies. Each one represents a certain input message as shown in Figure 5.13. For implementing the BFSK modulator, two accumulators were needed. They are working on the rising edge of the clock but at different speeds in order to give us two sinusoidal signals at different frequencies. Each one has a 24-bit width. The functionality of the MUX circuit in BFSK block diagram shown in Figure 5.14 works the same way as the BASK circuit does. The main difference here is that there are two sinusoidal signals with different frequencies instead of one sine wave signal in the BASK circuit.

The generated BFSK output is shown in Figure 5.15. The BFSK modulated signal was also exported to MATLAB and the resultant signal is shown in Figure 5.16. A very important point has to be mentioned here is that we played with the accumulator step size to get two sinusoidal signals at different frequencies.

The message signal used in the BFSK simulation and implementation is the same random binary signal used in ASK modulation.
Figure 5.13: FSK modulated signal.

Figure 5.14: Block diagram of BFSK modulator implementation in VHDL.
Figure 5.15: FSK modulation results in ModelSim.

Figure 5.16: FSK modulation results in MATLAB.
The device utilization summary of the design after following the same steps explained in the previous section for the design mapping and using a similar .xdc file is shown in Table 5.4.

### Table 5.4: Device utilization summary of the FSK modulator.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>17</td>
<td>17600</td>
<td>0.1</td>
</tr>
<tr>
<td>FF</td>
<td>14</td>
<td>35200</td>
<td>0.04</td>
</tr>
<tr>
<td>BRAM</td>
<td>1</td>
<td>60</td>
<td>1.67</td>
</tr>
<tr>
<td>IO</td>
<td>9</td>
<td>100</td>
<td>9.00</td>
</tr>
<tr>
<td>BUFG</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
</tbody>
</table>

5.5.3 Implementation of PSK Modulator

BPSK; as shown in Figure 5.17; is the simplest form of PSK. It uses two signals with 180-degree phase shift to represent its symbol. Figure 5.18 is the constellation diagram of BPSK. The general equation of BPSK is [7]:

\[
S_n(t) = \sqrt{\frac{2E_s}{T_s}} \cos(2\pi f_c t + (1-n)\pi) \quad n = 0,1
\]  

(5.2)

Where \(E_s\) is the energy per symbol, \(T_s\) is the symbol duration, and \(f_c\) is the carrier frequency. Based on the value of \(n\), two signals will be generated:

\[
S_0(t) = -\sqrt{\frac{2E_s}{T_s}} \cos(2\pi f_c t) \quad \text{when } n = 0
\]  

(5.3)
\[ S_1(t) = \sqrt{\frac{2E}{T_s}} \cos(2\pi f_c t) \quad \text{when } n = 1 \] (5.4)

This means two sinusoidal waveforms have to be generated in order to be able to implement BPSK in VHDL. For this purpose, the DDS technique was used. In the DDS method, a 24-bit accumulator with LUT was used for the sine wave generation. The used
accumulator works on the rising edge of the clock. The 8-most significant bits of the accumulator were used as an address to select the corresponding amplitude of the sinusoidal signal. Since the used address has 8-bit width, the LUT has to have 256 samples values which cover one cycle of the sinusoidal signal. The accumulator and LUT can give us only one signal based on the previous description. Hence we need to find a way to get the other signal which is 180-degree out of phase as compared to the first one. The easiest way to do that is to multiply the first signal by (-1) which corresponds to a 180-degree phase shift. Unfortunately, in VHDL, programmers try to avoid multiplication as possible as they could due to the high resources consumption. The other option, which what I did is to reverse the most significant bit in the accumulator. The best way to do that is by using exclusive or (XOR) logic gate function.

After getting the two out-of-phase sinusoidal signals, a multiplexer was used to generate the output BPSK signal as shown in Figure 5.19. The width of S0, S1, and the BPSK output was selected to be 16-bit. 8-bit width can also be used but we tried to get a smoother signal.
After following the detailed descriptions of the implementation in VHDL, ModelSim and MATLAB were used as tools to visualize the output signals. The message signal that is used in the BPSK simulation and implementation is the same random binary signal used in the previous implementation sections. The BPSK modulated signal in ModelSim is shown in Figure 5.20.

The implementation results were not very clear in ModelSim, so they were exported into MATLAB as text files for both the generated waveforms and the BPSK output signal. The generated sinusoids are shown in Figure 5.21 which was plotted in MATLAB. It is very clear that the generated waves have 180-degree compared to each other. The generated BPSK signal is shown in Figure 5.22. It is clear where the signal reverses its phase based on the incoming message.
Figure 5.20: BPSK implementation results in ModelSim.

Figure 5.21: The two-generated out-of-phase digital sinusoids.
The device utilization summary of the design after following the same steps explained in the ASK implementation section for the design mapping and using a similar xdc file is shown in Table 5.5.

Table 5.5: Device utilization summary of the PSK modulator.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>9</td>
<td>17600</td>
<td>0.05</td>
</tr>
<tr>
<td>FF</td>
<td>9</td>
<td>35200</td>
<td>0.03</td>
</tr>
<tr>
<td>IO</td>
<td>9</td>
<td>100</td>
<td>9.00</td>
</tr>
<tr>
<td>BUFG</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
</tbody>
</table>
5.5.4 Implementation of QPSK Modulator

QPSK, which is also known as quadri PSK, 4-PSK, and 4QAM, uses four different points on the constellation diagram to represent its symbols. Each symbol can be encoded by using two bits. Figure 5.23 is the constellation diagram of QPSK.

![QPSK constellation diagram](image)

Figure 5.23: QPSK constellation diagram.

Mathematically speaking, QPSK can be used to double the data rate as compared to BPSK using the same bandwidth. This means that QPSK can transmit twice the data rate in a given bandwidth as compared to BPSK. The general form of QPSK symbol is [7]:

\[
S_n(t) = \sqrt{\frac{2E}{T_s}} \cos(2\pi f_c t + (2n - 1)\frac{\pi}{4}) \quad n = 0, 1, 2, 3
\]  

(5.5)

Figure 5.24 explains how to generate QPSK signal from a stream of data. Based on the value of \(n\) in equation (3.4), four different signals can be generated. These signals are 90-degree out of phase to each other. Hence, implementation of QPSK modulator
required the generation of four sinusoidal signals which have 90-degree phase shift. In
order to do that, the first signal can be generated as it is described in the previous sections
using the accumulator working on the rising edge of the clock and LUT. The second
signal was generated using the same LUT, but at this time another accumulator working
on the falling edge of the clock was used. The other two signals were obtained by
reversing the most significant bit in the first and second accumulator and using the same
LUT.

After the generation of the four signals, the QPSK modulator can be implemented
as a next step. The incoming binary data which comes from random binary text file has to
be converted from serial to parallel data as it is shown in Figure 5.25. This process can be
easily done in VHDL.

The last step in the implementation is to use a multiplexer which is going to
choose among the four generated sinusoidal signals based on the parallel version of the
message signal as it is shown in the block diagram. S0, S1, S2, S3, and QPSK output
signal were selected to have a 16-bit width for smoother signal.
Figure 5.24: QPSK block diagram.

Figure 5.25: Block diagram of the QPSK modulator implementation in VHDL.
The detailed procedure explained before was followed to generate the QPSK modulated signal. The resultant QPSK signal besides the clock and message signal in ModelSim are shown in Figure 5.26. The four generated sinusoidal waves were exported into MATLAB as text file to check if they meet the specifications we are looking for. It is clear they met all the specifications in terms of the 90-degree phase shift as shown in Figure 5.27. The QPSK output signal was also exported as a text file to plot it in MATLAB which is shown in Figure 5.28. It is not easy to see the ModelSim results; that is why we plot the results in MATLAB also.

Figure 5.26: QPSK implementation results in ModelSim.
Figure 5.27: The four generated digital sinusoids with 90-degree phase shift.

Figure 5.28: QPSK implementation results in MATLAB.
In order to map the design on the ZYBO board, we made a 2-bit counter working based on a slower clock driven from the board main clock. This counter was used as the input message signal to the QPSK system.

The device utilization summary of the design after following the same steps explained in the ASK implementation section for the design mapping and using a similar .xdc file but this time the 2-bit counter was used as the input information signal is shown in Table 5.6.

Table 5.6: Device utilization summary of the QPSK modulator.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>17</td>
<td>17600</td>
<td>0.1</td>
</tr>
<tr>
<td>FF</td>
<td>14</td>
<td>35200</td>
<td>0.04</td>
</tr>
<tr>
<td>IO</td>
<td>9</td>
<td>100</td>
<td>9.00</td>
</tr>
<tr>
<td>BUFG</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
</tbody>
</table>

5.5.5 Implementation of QAM Modulator

In QAM, the amplitude of two carrier waves is changing based on the incoming bits stream. The first carrier wave is a cosine wave while the second one is a sine. These two carriers have the same frequency. QAM is considered as a combination of ASK and PSK since the output modulated signal has amplitude and phase variations. Figure 5.29 is a general block diagram of QAM generator. In QAM, higher spectral efficiency can be achieved but with higher probability of error.
The simplest form of QAM modulation is 4QAM which has the constellation diagram shown in Figure 5.30. In order to implement this kind of modulation, two sinusoidal waves were needed. These two waves have a 90-degree phase difference. The best way to do that in VHDL is by using two accumulators working on the rising edge and falling edge of the clock and one LUT as we did in QPSK implementation.

The block diagram of 4QAM implementation in VHDL is shown in Figure 5.31. The diagram looks similar to our QPSK one with some differences. The main difference here is: we need two carriers only in 4QAM, while in QPSK four carriers were needed. The multiplexer circuit gives four different combinations of the carrier signals based on the incoming information signal. Table 5.7 illustrates the four different multiplexer outputs based on the incoming symbols.
Figure 5.30: 4QAM constellation diagram.

Figure 5.31: Block diagram of the 4QAM modulator implementation in VHDL.
Table 5.7: 4QAM symbol representation

<table>
<thead>
<tr>
<th>Constellation Symbol</th>
<th>Modulated Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>$S_0+S_1$</td>
</tr>
<tr>
<td>01</td>
<td>$S_0-S_1$</td>
</tr>
<tr>
<td>10</td>
<td>$S_1-S_0$</td>
</tr>
<tr>
<td>11</td>
<td>$-S_0-S_1$</td>
</tr>
</tbody>
</table>

The second step in the QAM implementation is implementing 16QAM which has the constellation diagram shown in Figure 5.32. As in 4QAM, two carrier waves were needed for the implementation as illustrated in the block shown in Figure 5.33. The main differences between 16QAM and 4QAM implementation are the changes in the multiplexer and serial to parallel convertor blocks. Now the serial to parallel convertor circuit converts the incoming data into four parallel outputs instead of two as in 4QAM. The multiplexer circuit generates sixteen different combinations of the two input carriers based on the incoming symbols of data as shown in Table 5.8. More details on the VHDL code for 16QAM implementation can be found in appendix B.
Figure 5.32: 16QAM constellation diagram.

Figure 5.33: Block diagram of 16QAM modulator implementation in VHDL.
Table 5.8: 16QAM symbol representation

<table>
<thead>
<tr>
<th>Constellation Symbol</th>
<th>Modulated Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>$S_0+S_1$</td>
</tr>
<tr>
<td>0001</td>
<td>$3S_0+S_1$</td>
</tr>
<tr>
<td>0010</td>
<td>$-S_0+S_1$</td>
</tr>
<tr>
<td>0011</td>
<td>$-3S_0+S_1$</td>
</tr>
<tr>
<td>0100</td>
<td>$S_0+3S_1$</td>
</tr>
<tr>
<td>0101</td>
<td>$3S_0+3S_1$</td>
</tr>
<tr>
<td>0110</td>
<td>$-S_0+3S_1$</td>
</tr>
<tr>
<td>0111</td>
<td>$-3S_0+3S_1$</td>
</tr>
<tr>
<td>1000</td>
<td>$S_0-S_1$</td>
</tr>
<tr>
<td>1001</td>
<td>$3S_0-S_1$</td>
</tr>
<tr>
<td>1010</td>
<td>$-S_0-S_1$</td>
</tr>
<tr>
<td>1011</td>
<td>$-3S_0-S_1$</td>
</tr>
<tr>
<td>1100</td>
<td>$S_0-3S_1$</td>
</tr>
<tr>
<td>1101</td>
<td>$3S_0-3S_1$</td>
</tr>
<tr>
<td>1110</td>
<td>$-S_0-3S_1$</td>
</tr>
<tr>
<td>1111</td>
<td>$-3S_0-3S_1$</td>
</tr>
</tbody>
</table>

The 4QAM modulated signal based on the implemented model is shown in Figure 5.34, while the modulated output signal of the 16QAM module is shown in Figure 5.35. The QAM modulated outputs have some transients at certain instances which means we
might want to include additional signal processing or digital filters to reduce them in the future.

Figure 5.34: 4QAM modulation results.

Figure 5.35: 16QAM modulator results.
The device utilization summary of the 4QAM, and 16QAM design after following the same steps explained in the previous implementation sections for the design mapping and using a similar .xdc file, but this time 2-bit and 4-bit counters are used as the input signals for 4QAM and 16QAM systems are shown in Table 5.9 and 5.10 respectively.

Table 5.9: Device utilization summary of the 4QAM modulator.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>84</td>
<td>17600</td>
<td>0.48</td>
</tr>
<tr>
<td>FF</td>
<td>14</td>
<td>35200</td>
<td>0.04</td>
</tr>
<tr>
<td>IO</td>
<td>9</td>
<td>100</td>
<td>9.00</td>
</tr>
<tr>
<td>BUFG</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
</tbody>
</table>

Table 5.10: Device utilization summary of the 16QAM modulator.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Utilization</th>
<th>Available</th>
<th>Utilization %</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUT</td>
<td>251</td>
<td>17600</td>
<td>1.43</td>
</tr>
<tr>
<td>FF</td>
<td>17</td>
<td>35200</td>
<td>0.05</td>
</tr>
<tr>
<td>IO</td>
<td>9</td>
<td>100</td>
<td>9.00</td>
</tr>
<tr>
<td>BUFG</td>
<td>1</td>
<td>32</td>
<td>3.13</td>
</tr>
</tbody>
</table>

5.6 Summary

This chapter presented novel methods of implementing OOK, BASK, 4-ASK, BFSK, BPSK, QPSK, and QAM digital modulators. It started by explaining the main concept of FPGAs, how they are programmed, and the current implementation methods
of digital modulators using FPGA. Then, it presented a review of the recent research work in the area of FPGA based digital modulators. After that the main goal of the chapter was presented which is the implementation of different kinds of digital communication modulators on FPGA. The technique shown begin with single bit per transmitted symbol systems (OOK, BPSK and 2FSK) which have also been reported, although mostly using IP elements previously. This work then extended the modulation techniques into those capable of supporting multiple bits per symbol (4ASK, 4-QAM and 16-QAM) along with implementation methodologies that could readily extend to M-ASK, M-PSK and M-FSK that is unique to this work and the recent conference publications of this research. The targeted board of the implementation is ZYBO board which has Xilinx Zynq-7000 (Z-7010) FPGA. The implementation was done entirely in VHDL without the help of Xilinx System Generator or DSP Builder Tools which depend on IP-Blocks.

One of the significant challenges in this work is the generation of the digital sine wave carrier(s). In order to do that, digital accumulators representing unsigned phase increments (with the MSB being 180-degree) with LUT table based on DDS method were used. In some modulation schemes, which require more than one carrier, one or two accumulators with a reverse addressing trick were used. For example, the implementation of BPSK modulator, two sinusoidal signals with 180-degree phase shift were generated using a single LUT and without any multiplication. The first signal was obtained using a LUT and accumulator working on the rising edge of the clock based on DDS technique. The second one was obtained by using the same LUT, but after reversing the most significant bit of the accumulator which represents a signal with 180-degree phase shift.
as compared to the first one. For the QPSK modulator, four sinusoidal waves were generated using only one LUT. The first two waves were obtained by using two accumulators working on the rising and the falling edge of the clock which makes them have a 90-degree phase shift. The other waves were obtained from the first two signals by reversing the most significant bit of the two accumulators. In BFSK and QAM, two carriers were generated by using two accumulators and one LUT at different step size for BFSK and based on the rising and falling edges of the clock for QAM.

For higher bit per symbol modulation schemes, integer scaling involving shifts and adds of the phase based sinusoids developed readily allow 4-ASK based on magnitudes scaling of -3, -1, 1 and 3. Similarly 16-QAM uses the same magnitude applied in-phase and quadrature-phase sinusoids. As, a readily available future extensions, 8-ASK and 64-QAM would use -7, -5, -3, -1, 1, 3, 5, and 7 magnitude scaling of the same base sinusoidal signals defined previously. The limitation in this technique involves the resolution of the DAC and the availability of digital or analog filtering. As the magnitude increases, the relative phase accuracy decreases for a DAC with the same number of bits. As the phase accuracy decreases, the spectral purity desired decreases and harmonic distortion and harmonic products result, degrading the transmitted signal.
CHAPTER VI

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The proliferation of cost-effective terminals and multiprocessor devices besides open source software has opened the doors for the implementation of different communication systems using these resources. This kind of implementation enables a wide range of potential applications, selected to either augment or even replace commercial communication equipment. Hence this dissertation presented an implementation of three different projects which aim to work as components of various communications systems, but can more specifically benefit various embodiments of community radio systems.

In the first project, a 64-channel rational rate PFBC was implemented using two GPU systems. The implementation was done in CUDA without using any standard CUDA library as in other research work. The parallel implemented models were compared with an efficient CPU-based one using core i5 laptop computer running windows 7 (64-bits) with 4 Gigabyte memory and 2.3 GHZ clock speed. The comparisons indicate that for this implementation an average speed up of 9-16 times faster can be achieved with the addition of GPU processing. MATLAB software was used to build the complete channelizer system, then the built channelizer was simulated and implemented in C++ and CUDA respectively using visual studio software [92-93].

The second project presented novel, efficient ways of prototyping a complete community radio transmitter and receiver systems using cost-effective resources. The CR
transmitters were demonstrated and validated through FM transmission using a PC, USRP and GNU Radio. A low-cost CR receiver configured using a DVB-T USB dongle and special version of Linux and GNU radio hosted on a Raspberry-PI was also implemented. Successful SDR based point-to-multipoint communications of an FM signal was achieved. Different implementation options of the transmitter/receiver were also investigated using all available lab resources, including cost-effective devices and terminals such as PC, Raspberry PI, USRP, RTL-SDR dongle, real time spectrum analyzer, and antennas [94-95]. While the idea of building a community radio transmitter based on SDR concept has been previously investigated, no one has investigated the implementation of a complete community radio system using these kinds of resources. The advantages are significant in allowing rapid reconfiguration of all components to easily comply with country or regional radio signal regulations and restrictions based on frequency, signal strength, signal bandwidth, or modulation formats.

The third project in this dissertation presented several novel methods of implementing different kinds of digital modulators using FPGA. All structures are IP free without the use of Xilinx System Generator or DSP Builder Tools. The targeted board of the implementation is ZYBO board which is an entry-level embedded platform built around Xilinx Zynq-7000 (Z-7010) family. Successful implementation of different modulators such as OOK, ASK, PSK, FSK, QPSK, and QAM was achieved [96-97]. The implemented systems were also mapped and verified on the ZYBO board. The mapping indicates that a very low utilization was achieved, which means that the same board can be used to build different systems at the same time. The potential greatest extension of
this work involves combining the FPGA designs prototyped with the embedded dual
ARM core processor of the Xilinx Zynq device.

6.2 Future Work

Based on the work that has been completed and the results obtained from the three
projects presented in this dissertation, multiple possibilities exist to further advance the
ideas and concepts already presented and extend this work for more advanced system
implementations. We classified these ideas based on the projects they are related to as the
following paragraphs will explain.

In the research area of the first project which is GPU based implementation of a
64-channel polyphase channelizer, future work will be:

1-To investigate the continuous pipelined processing of the PFBC, different options for
the PFBC, data shuffling hosted on the GPU, and consider different numbers of channels
with different data size. With NVIDIA introduction of Pascal GPU devices, further
increases in computation rates, reduction in computation time and major reductions in
data transfers are anticipated.

2-As a further extension, other wireless communication signal processing operation
typically performed on FPGA or dedicated hardware will be investigated for parallel
software processing and GPGPU implementation. These include the inverse algorithm to
the PFBC involving multiband polyphase signal synthesis, wideband channel
equalization, and OFDM symbol transmission and reception as used in wireless Long-
Term Evolution (LTE) and LTE advanced cellular telephones and IEEE 802.11 WiFi.
3- The first project itself represents the main step of our big project presented in [93]. This means finishing the big project is one of the future research activities that we will continue to work on. These activates include accelerating the channelizer part by utilizing the new Pascal GPU architecture, obtaining an optimum solution for the time spent in data movement between CPU and GPU, and looking for a better connection between GNU Radio and CUDA to reach the real-time processing of the received FDM signal.

4- Once both multichannel PFBC receiving and transmission can be simultaneously hosted on a signal PC with GPU platform, a small scale universal wireless base station can be considered. Such a system could act as a home or small business wireless access point for all types and formats of wireless signals, such as WiFi, cellular telephone, IOT devices, etc. When connected to a fiber optic backbone or service provider, it may also act as a universal cellular telephone base station, supporting any or all required signal formats simultaneously.

Future research based on the second project which is software defined community radio using low cost resources will be:

1- Utilize this community radio architecture to implement other kinds of communication protocols and services. Although FM is often used in existing systems, SDR capability allows other forms of analog and all forms of digital modulation to be considered.

2- Accommodate open source radio automation platforms like Rivendell and demonstrate a complete community radio platform that can be an effective substitute for commercial systems.
3-Implement a community radio transmitter using FPGA based development boards and cost effective terminal devices, continuing to seek lower cost implementations with higher flexibility and re-programmability of hardware analog with the software.

Future research based on the third project which is FPGA based modulators for SDR applications will be:

1-Implement other kinds of digital modulators such as Minimum Shift Keying (MSK), Continuous Phase Frequency Shift Keying (CPFSK) and others.

2- Include hardware programmed VHDL based filters with the modulators to improve signal characteristics and limit distortion or harmonic generations.

3-Investigate the using the Zybo pmod terminals to connect to DAC and ADC and also use USRP RF daughter cards.

4-Future research will also use the dual ARM cores present on Zynq and the other interface resources of the ZYBO board to construct a complete ZYBO SDR transmitter and receiver.

5- We need to complete the design by building complete system transmitters/receivers such as a community radio transmitter. The new complete system will utilize the FPGA based modulators, DACs, low noise power amplifiers, and antennas.

Future research based on combining the various pieces involves a complete community radio emergency communication system. The system would be deployed in the event of a man-made or natural disaster where the communication infrastructure has been destroyed or non-longer exists. A multichannel or single channel transmitting base
station would be required based on elements of project one or two. Small, low cost, low power SDR receivers based on project two would provide all multiple users to hear critical information being broadcast by the base station or even news or music that may bring a sense of normalcy in the emergency. Based on the progress of project three, the remote units could provide two way communications. If the remotes contain the equivalent of a LINUX based PC, digital data and limited computer network traffic may also be available.

6.3 Summary

The world of wireless communication continues to advance, developing every increasing applications and capability. As technology has advanced incorporation of many computer and software programming devices, methods and technique have become available and applied. This work provides further extension in SDR concepts and applications seeking to utilize commercial PC and graphics components to form a wireless base station and finding technologies and devices that can form various embodiments of SDR community. The insights and knowledge gained in this work provide an excellent base for my future career.
BIBLIOGRAPHY


[25]. Cambridge Consultants, “World's first fully digital radio transmitter built purely from microprocessor technology,” available online:


[49]. “NVIDIA Tesla P100: The most advanced datacenter accelerator ever built featuring Pascal GP100, the world’s fastest GPU,” White paper, available online: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.


[55]. L. Tobing, “How to do community radio: A primer for community radio operators,” New Delhi, UNESCO.


[67]. https://www.mentor.com/products/fv/modelsim/


[87]. K. Feher et al.: US Patents 4,567,602; 4,339,724; 4,644,565; 5,491,457; 5,784,402 and Canadian Patents: 1,211,5017; 1,130,871; 1,265,851 and post-patent improvements, including Digcom’s proprietary ‘know-how’ and other US and international patents pending/in applications-in-process. These have been licensed on an equal-opportunity, non-discriminatory fair market value basis for “dual-use” commercial and military FQPSK-GMSK use, make, sale, developments and other use and applications to large, medium size and small corporations and organizations in the USA and internationally by Digcom, Inc.


APPENDICES

Appendix A

#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
#include<windows.h>
#define  PI 3.142857142857143f
#define  test_length 6144*6
#define poly_length 1600
#define Numofchannels 64
#define pcoef_length Numofchannels
#define pcoef_width (poly_length/Numofchannels)
#define DFT_Size 64
#define NumBlocks 128*6
#define blocksize 128  //change the blocksize if there is a difference between the parallel
and serial implementation

__device__ __constant__ float dev_matr[DFT_Size*DFT_Size];
__device__ __constant__ float dev_matim[DFT_Size*DFT_Size];
__device__ __constant__ float dev_poly_filter[poly_length];

__global__ void gpu_flipud_after_lr_withpointMul(float *in2_r,float *in2_im,float
*sum_cs_r,float *sum_cs_im,int length,int width,int k )
{
    int i = blockIdx.x*blockDim.x+threadIdx.x;
    __shared__ float sum_r[pcoef_length];
    __shared__ float sum_im[pcoef_length];

    if(i<length){
        sum_r[i]=dev_poly_filter[i]*in2_r[i]+dev_poly_filter[i+length]*in2_r[i+length]+dev_poly_filter[i+2*length]*in2_r[i+2*length]+dev_poly_filter[i+3*length]*in2_r[i+3*length]+dev_poly_filter[i+4*length]*in2_r[i+4*length]+dev_poly_filter[i+5*length]*in2_r[i+5*length]+dev_poly_filter[i+6*length]*in2_r[i+6*length]+dev_poly_filter[i+7*length]*in2_r[i+7*length]+dev_poly_filter[i+8*length]*in2_r[i+8*length]+dev_poly_filter[i+9*length]*in2_r[i+9*length]+dev_poly_filter[i+10*length]*in2_r[i+10*length]+dev_poly_filter[i+11*length]*in2_r[i+11*length]+dev_poly_filter[i+12*length]*in2_r[i+12*length]+dev_poly_filter[i+13*length]*in2_r[i+13*length]+dev_poly_filter[i+14*length]*in2_r[i+14*length];
    }
}
\begin{verbatim}

r[i+14*length]+dev_poly_filter[i+15*length]*in2_r[i+15*length]+dev_poly_filter[i+16*length]*in2_r[i+16*length]+dev_poly_filter[i+17*length]*in2_r[i+17*length]+dev_poly_filter[i+18*length]*in2_r[i+18*length]+dev_poly_filter[i+19*length]*in2_r[i+19*length]+dev_poly_filter[i+20*length]*in2_r[i+20*length]+dev_poly_filter[i+21*length]*in2_r[i+21*length]+dev_poly_filter[i+22*length]*in2_r[i+22*length]+dev_poly_filter[i+23*length]*in2_r[i+23*length]+dev_poly_filter[i+24*length]*in2_r[i+24*length];


}  //  __syncthreads();

if((i+k)>=length){
    sum_cs_r[i+k-length]=sum_r[i];
    sum_cs_im[i+k-length]=sum_im[i];
}
else{
    sum_cs_r[i+k]=sum_r[i];
    sum_cs_im[i+k]=sum_im[i];
}

__global__ void DFT_GPU(float *vecr,float *vecim, float *outr,float *outim, const int N1){

    int tid=threadIdx.x+blockIdx.x*blockDim.x;
    float sumr=0,sumim=0;
    if(tid<N1){
        for(int i=0; i<N1; i++){

        }

    }

    __syncthreads();

}
\end{verbatim}
sumr += vecr[i]*dev_matr[(i*N1)+tid]-vecim[i]*dev_matim[(i*N1)+tid];
    sumim +=
    vecr[i]*dev_matim[(i*N1)+tid]+vecim[i]*dev_matr[(i*N1)+tid];
}
    outr[tid]=sumr;
    outim[tid]=sumim;
}
}
}

int main()
{
    LARGE_INTEGER frequency;        // ticks per second
    LARGE_INTEGER t1, t2;           // ticks
    double elapsedTime;
    int i=0,j=0;
    float tsg_r[test_length],tsg_im[test_length];

    system("cls");
    //load the test signal (the real and imaginary part of it)
    FILE *file=fopen("Input_test_signal_r.txt","r");

    if(file == NULL)
    { printf("File not found");}
    while (fscanf(file, "%f",&tsg_r[j]) != EOF) {
        j=j+1;
    }

    //close the opened file
    fclose(file);
    FILE *file1=fopen("Input_test_signal_im.txt","r");

    if(file1 == NULL)
    { printf("File not found");}
    j=0;
    while (fscanf(file1, "%f",&tsg_im[j]) != EOF) {
        j=j+1;
    }
    fclose(file1);

    //print the test signal
    // for(j=0; j<test_length; j++)
    //printf("%f+(%f)\n ", tsg_r[j],tsg_im[j]);
//To compute DFT
float
Fmat_r[DFT_Size][DFT_Size], Fmat_im[DFT_Size][DFT_Size], Vec_out_r[DFT_Size],
Vec_out_im[DFT_Size];///
float *dev_Vec_out_r,*dev_Vec_out_im;

for (int i=0;i<DFT_Size;i++){
    for(int j=0;j<DFT_Size;j++){
        Fmat_r[i][j]=cosf(2*PI*i*j/DFT_Size);
        Fmat_im[i][j]=sinf(2*PI*i*j/DFT_Size);
    }
}

for (int i=0;i<DFT_Size;i++){
    Vec_out_r[j]=0;
    Vec_out_im[j]=0;
}

//cudaMalloc((void**)&dev_Fmat_r, sizeof(float)*DFT_Size*DFT_Size);
//cudaMalloc((void**)&dev_Fmat_im, sizeof(float)*DFT_Size*DFT_Size);
cudaMalloc((void**)&dev_Vec_out_r, sizeof(float)*DFT_Size);
cudaMalloc((void**)&dev_Vec_out_im, sizeof(float)*DFT_Size);

//load the filter coeffecients from Matlab
float hpolyphase[poly_length];
    FILE *file2=fopen("Polyfilter_d5.txt","r");
    if(file2 == NULL)
    { printf("File not found");}
    j=0;
    while (fscanf(file2, "%f", &hpolyphase[j]) != EOF) {
        j=j+1;
    }
    fclose(file2);

    //generate the siglin signal which is poly_length*1 vector it starts at zero values
    int M=48,jj,cshiftval,PPfft=pcoef_length;
    float siglin_r[poly_length], siglin_im[poly_length];
    float sum_cs_r[pcoef_length],sum_cs_im[pcoef_length];

    //Dynamic allocation
    int q;
    float *yout_r[pcoef_length];
for (q=0; q<pcoef_length; q++)
    yout_r[q] = (float *)malloc(NumBlocks* sizeof(float));

float *yout_im[pcoef_length];
for (q=0; q<pcoef_length; q++)
    yout_im[q] = (float *)malloc(NumBlocks* sizeof(float));

float *pfout_r[pcoef_length];
for (q=0; q<pcoef_length; q++)
    pfout_r[q] = (float *)malloc(NumBlocks* sizeof(float));

float *pfout_im[pcoef_length];
for (q=0; q<pcoef_length; q++)
    pfout_im[q] = (float *)malloc(NumBlocks* sizeof(float));

float *dev_siglin_r,*dev_siglin_im;
float *dev_sum_cs_r,*dev_sum_cs_im;

cudaMalloc((void **) &dev_sum_cs_r, pcoef_length*sizeof(float));
cudaMalloc((void **) &dev_sum_cs_im, pcoef_length*sizeof(float));

for (i=0;i<poly_length;i++) {
    siglin_r[i]=0;
    siglin_im[i]=0;
}

for (i=0;i<pcoef_length;i++) {
    sum_cs_r[i]=0;
    sum_cs_im[i]=0;
}

cudaMalloc((void **) &dev_siglin_r, poly_length*sizeof(float));
cudaMalloc((void **) &dev_siglin_im, poly_length*sizeof(float));

//copy to the device the DFT real and imaginary matrices
cudaMemcpyToSymbol(dev_matr, &Fmat_r,DFT_Size*DFT_Size*sizeof(float));
cudaMemcpyToSymbol(dev_matim, &Fmat_im,DFT_Size*DFT_Size*sizeof(float));

//copy to the device the initials values of the real and imaginary of the vector you want to compute its DFT

cudaMemcpy(dev_Vec_out_r, Vec_out_r, sizeof(float)*DFT_Size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_Vec_out_im, Vec_out_im, sizeof(float)*DFT_Size, cudaMemcpyHostToDevice);
//copy to the device the initial values of the sums
cudaMemcpy(dev_sum_cs_r, sum_cs_r,
poly_length*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_sum_cs_im, sum_cs_im,
poly_length*sizeof(float), cudaMemcpyHostToDevice);

//copy to the device the filter coefficients
cudaMemcpyToSymbol(dev_poly_filter, &hpolyphase, poly_length*sizeof(float));

//copy to the device the initials values of the signal
cudaMemcpy(dev_siglin_r, siglin_r,
poly_length*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_siglin_im, siglin_im,
poly_length*sizeof(float), cudaMemcpyHostToDevice);

//get ticks per second
QueryPerformanceFrequency(&frequency);
QueryPerformanceCounter(&t1);

for(i=0;i<NumBlocks;i++)
{
    cshiftval=((i*M)%PPfft);
    
    for(j=0;j<(poly_length-M);++j){
        siglin_r[j]=siglin_r[j+M];
        siglin_im[j]=siglin_im[j+M];
    }

    for(j=(poly_length-M);j<poly_length;j++)
    {
        jj=i*M+j-(poly_length-M);
        siglin_r[jj]=tsg_r[jj];
        siglin_im[jj]=tsg_im[jj];
    }

cudaMemcpy(dev_siglin_r, siglin_r,
poly_length*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(dev_siglin_im, siglin_im,
poly_length*sizeof(float), cudaMemcpyHostToDevice);
gpu_flipud_after_lr_withpointMul<<<1,pcoef_length>>>(dev_siglin_r,dev_siglin_im,dev_sum_cs_r,dev_sum_cs_im,pcoef_length,pcoef_width,cshiftval); //((0.7 alone)

DFT_GPU<<<1,DFT_Size>>>(dev_sum_cs_r,dev_sum_cs_im,dev_Vec_out_r,dev_Vec_out_im,DFT_Size);   //((0.667 alone)

cudAMemcpy(sum_cs_r,dev_Vec_out_r, pcoef_length*sizeof(float),cudaMemcpyDeviceToHost);

cudAMemcpy(sum_cs_im,dev_Vec_out_im, pcoef_length*sizeof(float),cudaMemcpyDeviceToHost);

for(j=0;j<pcoef_length;j++){
  yout_r[j][i]=sum_cs_r[j];
  yout_im[j][i]=sum_cs_im[j];
}
//end of th big loop

for(i=0;i<pcoef_length;i++){
  for(j=0;j<NumBlocks;j++){
    pfout_r[i][j]=M*yout_r[i][j];
    pfout_im[i][j]=M*yout_im[i][j];
  }
}
//get the final time
QueryPerformanceCounter(&t2);

elapsedTime = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;

//for real part
FILE *f_r=fopen("out_sig_r_op5_parallel2.txt","w");

for(i=0;i<pcoef_length;i++) {
  for(j=0;j<NumBlocks;j++) {
    fprintf(f_r,"%f ",pfout_r[i][j]);
  }
  fprintf(f_r,"\n");}
fclose(f_r);

//for imaginary part
FILE *f_im=fopen("out_sig_im_op5_parallel2.txt","w");
for(i=0;i<pcoef_length;i++) {
for(j=0;j<NumBlocks;j++) {
    fprintf(f_im,"%f ",pfout_im[i][j]);
}
fprintf(f_im,"\n");}

fclose(f_im);
cudaFree(dev_Vec_out_r);
cudaFree(dev_Vec_out_im);
cudaFree(dev_siglin_r);
cudaFree(dev_siglin_im);
cudaFree(dev_sum_cs_r);
cudaFree(dev_sum_cs_im);

printf("Time elapsed in milli_sec:%1f\n\n",elapsedTime);

system("pause");

return 0;
}

Appendix B
1-QAM Modulator (16QAM)

-- Company:
-- Engineer:
-- Create Date: 09/29/2016 12:02:56 PM
-- Design Name:
-- Module Name: qam_modulator - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
-- use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
-- library UNISIM;
-- use UNISIM.VComponents.all;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;
use STD.TEXTIO.ALL;
entity qam_modulator is
   port(clk : in std_logic;
        message : in std_logic_vector(3 downto 0);
        qam_amp : inout std_logic_vector(15 downto 0));
end qam_modulator;
architecture Behavioral of qam_modulator is
   procedure output_qam_file (amp : in integer) is
      file qam_wave_file : text open append_mode is
         "C:\Users\Amean\Desktop\Dissertation_template\qam_wave_file.txt";
      variable print : line;
      begin
         write(print, amp);
         writeln(qam_wave_file, print);
         file_close(qam_wave_file);
      end output_qam_file;
component ClockPrescaler
  port(
    clk : in STD_LOGIC;
    clk_out : out STD_LOGIC
  );
end component;

component Message_read
  port(
    clk : in std_logic;
    message : inout std_logic);
end component;

component DDS_entity
  port(
    clk : in std_logic;
    sine_wave_amp : inout std_logic_vector(15 downto 0);
    sine_wave_amp2 : inout std_logic_vector(15 downto 0));
end component;

component multiplexer16X1
  port(A,B : in std_logic_vector(15 downto 0);
       sel : std_logic_vector(3 downto 0);
       F : out std_logic_vector(15 downto 0));
end component;

component SerinParout
port(
    clk : in std_logic;
    S_in: in std_logic;
    P_out : out std_logic_vector(3 downto 0));
end component;

signal osc1, osc2 : std_logic_vector(15 downto 0);
    signal message_2 : std_logic_vector(3 downto 0);
    signal message1 : std_logic;
    signal clk_2 : std_logic;

begin
    Freq_division:ClockPrescaler port map(clk,clk_2);
    get_data:Message_read port map(clk_2,message1);

    DDS1 : DDS_entity port map (clk, osc1,osc2);

    STP: SerParout port map(clk,message1, message_2);
    --mux : multiplexer16X1 port map (osc1, osc2, message, qam_amp);
    mux : multiplexer16X1 port map (osc1, osc2, message_2, qam_amp);

    process(clk)
    begin
        if (rising_edge(clk)) then
            output_qam_file(to_integer(signed(qam_amp)));
        end if;
    end process;
end Behavioral;
entity ClockPrescaler is
    port(
        clk : in STD_LOGIC; -- 50 Mhz
        clk_out : out STD_LOGIC
architecture Behavioral of ClockPrescaler is

-- prescaler should be (clock_speed/desired_clock_speed)/2 because you want a rising edge every period where desired_clock_speed=1MHZ

signal prescaler: STD_LOGIC_VECTOR(4 downto 0) := "11001"; -- 25 in binary
signal prescaler_counter: STD_LOGIC_VECTOR(4 downto 0) := (others => '0');
signal newClock : std_logic := '0';

begin

clk_out <= newClock;

countClock: process(clk,newClock)
begin

if rising_edge(clk) then

    prescaler_counter <= prescaler_counter + 1;

    if(prescaler_counter > prescaler) then

        -- Iterate

        newClock <= not newClock;

        prescaler_counter <= (others => '0');

    end if;

end if;

end process;

end Behavioral;

3-MESSAGE_READ
-- Company:
-- Engineer:
-- Create Date: 09/29/2016 12:14:32 PM
-- Design Name:
-- Module Name: Message_read - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;
library STD;
use STD.TEXTIO.ALL;

entity Message_read is
port(
    clk : in std_logic;
    message : inout std_logic);
end Message_read;
architecture Behavioral of Message_read is
signal endoffile : std_logic := '0';
signal linenumber : integer := 1;
begin
--read process
    reading :process
        file infile : text is in
"C:\Users\Amean\Desktop\Dissertation_template\Random_Numbers.txt"; --declare input file
        variable inline : line; --line number declaration
        variable dataread : std_logic;
    begin
        wait until rising_edge(clk);
        if (not endoffile(infile)) then --checking the "END OF FILE" is not reached.
            readline(infile, inline); --reading a line from the file.
            --reading the data from the line and putting it in a real type variable.
            read(inline, dataread);
            message <= dataread; --put the value available in variable in a signal.
        else
            endoffile <= '1'; --set signal to tell end of file read file is reached.
        end if;
    end process reading;
end Behavioral;

4-DDS_entity
-- Company:
-- Engineer:
-- Create Date: 09/29/2016 12:17:42 PM
-- Design Name:
-- Module Name: DDS_entity - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DDS_entity is
  port(
    clk : in std_logic;
    sine_wave_amp : inout std_logic_vector(15 downto 0);
    sine_wave_amp2 : inout std_logic_vector(15 downto 0));
end DDS_entity;
architecture implementation of DDS_entity is

    signal pR_connect : std_logic_vector(23 downto 0) := (others => '0');
    signal pR_connect2 : std_logic_vector(23 downto 0) := (others => '0');

    component phase_generator
        port(
            clk : in std_logic;
            pR : inout std_logic_vector(23 downto 0);
            pR2 : inout std_logic_vector(23 downto 0);
        );
    end component;

    component Phase_To_Wave
        port(pR : in std_logic_vector(23 downto 0);
             sine_wave : inout std_logic_vector(15 downto 0));
    end component;

    component Phase_To_Wave_2
        port(pR : in std_logic_vector(23 downto 0);
             sine_wave : inout std_logic_vector(15 downto 0));
    end component;

    begin
        PG_instance : phase_generator port map (clk, pR_connect,pR_connect2);
        PtWC_instance : Phase_To_Wave port map (pR_connect, sine_wave_amp);
PtWC_instance2 : Phase_To_Wave_2 port map (pR_connect2, sine_wave_amp2);

end implementation;

5-phase_generator

-- Company:
-- Engineer:
-- Create Date: 09/29/2016 12:26:18 PM
-- Design Name:
-- Module Name: phase_generator - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use STD.TEXTIO.ALL;
use IEEE.STD_LOGIC_TEXTIO.ALL;

entity phase_generator is
  port(
    clk : in std_logic;
    pR : inout std_logic_vector(23 downto 0) := (others => '0');
    pR2 : inout std_logic_vector(23 downto 0) := (others => '0'));
end phase_generator;

architecture structural of phase_generator is
begin
  clock_process : process(clk)
    variable pR_buff,pR_buff2 : std_logic_vector(23 downto 0);
    variable pl_integer : integer := 2097152;

    variable pl_vector : std_logic_vector(23 downto 0) :=
    std_logic_vector(to_unsigned(pl_integer, 24));

    begin
      pR_buff := pR;
      pR_buff2 := pR2;

      if rising_edge(clk) then
        pR <= pl_vector + pR_buff;
      else

179
pR2 <= pI_vector + pR_buff2;

end if;

if (to_integer(unsigned('0'&pR)) = 2**24) then
    pR <= (others => '0');
end if;

if (to_integer(unsigned('0'&pR2)) = 2**24) then
    pR2 <= (others => '0');
end if;

end process;
end structural;

6-Phase_To_Wave

-- Company:
-- Engineer:
-- Create Date: 09/29/2016 12:19:52 PM
-- Design Name:
-- Module Name: Phase_To_Wave - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.Numeric_Std.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
library IEEE;
use IEEE.Std_Logic_1164.ALL;
use IEEE.Numeric_Std.ALL;
use IEEE.Std_Logic_Textio.ALL;
use STD.Textio.ALL;
entity Phase_To_Wave is
    port(pR : in std_logic_vector(23 downto 0);
        sine_wave : inout std_logic_vector(15 downto 0) := (others => 'U'));
end Phase_To_Wave;
architecture sine_wave_builder of Phase_To_Wave is
    type table is array (0 to 255) of integer;
    signal addr : integer := 0;
    alias addr_vector : std_logic_vector(7 downto 0) is pR(23 downto 16);
    procedure table_fill (look_up_var : inout table) is
        file LOOK_UP_TABLE : text open read_mode is
            "C:\Users\Amean\Desktop\Dissertation_template\LOOK_UP_TABLE_QAM.txt";
        variable buff : line;
        begin
            while(not (endfile(LOOK_UP_TABLE))) loop
                for i in 0 to 255 loop
                    readline(LOOK_UP_TABLE, buff);
                    read(buff, look_up_var(i));
end loop;
end loop;

file_close(LOOK_UP_TABLE);
end table_fill;

procedure output_sine_file (amp : in integer) is
  file sine_wave_file : text open append_mode is
    "C:\Users\Amean\Desktop\Dissertation_template\sine_wave_file_QAM.txt";
  variable print : line;
begin
  write(print, amp);
  writeln(sine_wave_file, print);
  file_close(sine_wave_file);
end output_sine_file;

begin
  fill_table : process(pR)
  variable look_up : table;
  begin
    table_fill(look_up);
    addr <= to_integer(unsigned(addr_vector));
    sine_wave <= std_logic_vector(to_signed(look_up(addr), 16));
    output_sine_file(to_integer(signed(sine_wave)));
  end process;
  end sine_wave_builder;

7-SerinParout
-- Company:
-- Engineer:
-- Create Date: 09/29/2016 12:07:08 PM
-- Design Name:
-- Module Name: SerinParout - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Dependencies:
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;
-- Uncomment the following library declaration if instantiating
-- any Xilinx leaf cells in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity SerinParout is
port(
    clk : in std_logic;
    S_in: in std_logic;
    P_out : out std_logic_vector(3 downto 0));
end SerinParout;
architecture Behavioral of SerinParout is
    signal temp : std_logic_vector(2 downto 0);
    signal counter: integer:=0;
begin
    process(clk)
    begin
        if (rising_edge(clk)) then
            if (counter<3) then
                temp(counter)<=S_in;
                counter<=counter+1;
            else
                P_out<=S_in&temp;
                counter <=0;
            end if;
        end if;
    end process;
end Behavioral;

8-multiplexer16X1
-- Company:
-- Engineer:
-- Create Date: 09/29/2016 12:48:39 PM
-- Design Name:
-- Module Name: multiplexer16X1 - Behavioral
-- Project Name:
-- Target Devices:
-- Tool Versions:
-- Description:
-- Dependencies:
library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;
entity multiplexer16X1 is
  port(A,B : in std_logic_vector(15 downto 0);
       sel : std_logic_vector(3 downto 0);
       F : out std_logic_vector(15 downto 0));
end multiplexer16X1;

architecture structural of multiplexer16X1 is
signal temp : std_logic_vector(16 downto 0);
signal A_3 : std_logic_vector(16 downto 0);
signal B_3 : std_logic_vector(16 downto 0);
signal A_1 : std_logic_vector(16 downto 0);
signal B_1 : std_logic_vector(16 downto 0);
begin
A_1 <= (A(15) & A);
B_1 <= (B(15) & B);
A_3 <= (A(15) & A) + (A(15) & A) + (A(15) & A);
B_3 <= (B(15) & B) + (B(15) & B) + (B(15) & B);

process(A, B)
begin
  case sel is
    when "0000" => temp <= A_1 + B_1; -- A + B
    when "0001" => temp <= A_3 + B_1; -- 3A + B
    when "0010" => temp <= B_1 - A_1; -- A - B
    when "0011" => temp <= B_1 - A_3; -- 3A - B
    when "0100" => temp <= A_1 + B_3; -- A + 3B
    when "0101" => temp <= A_3 + B_3; -- 3A + 3B
    when "0110" => temp <= B_3 - A_1; -- A + B
    when "0111" => temp <= B_3 - A_3; -- 3A + B
    when "1000" => temp <= A_1 - B_1; -- A - B
    when "1001" => temp <= A_3 - B_1; -- 3A - B
    when "1010" => temp <= "00000000000000000" - A_1 - B_1; -- A - B
    when "1011" => temp <= "00000000000000000" - A_3 - B_1; -- 3A - B
    when "1100" => temp <= A_1 - B_3; -- A - 3B
    when "1101" => temp <= A_3 - B_3; -- 3A - 3B
    when "1110" => temp <= "00000000000000000" - A_1 - B_3; -- A - 3B
    when others => temp <= "00000000000000000" - A_3 - B_3; -- 3A - 3B
  end case;

  F <= temp(15 downto 0);
end process;
end structural;