FPGA Implementation of TROPO-SCATTER MODEM

A Project Report

submitted by

SAI NIKHIL MARAM

ANANTH HARI

in partial fulfilment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

June 2014

THESIS CERTIFICATE

This is to certify that the thesis titled **FPGA IMPLEMENTATION OF TROPO-SCATTER MODEM SUBMITTED TO IIT-M**, submitted by **Sai Nikhil Maram** and **Ananth Hari**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology** , is a bonafide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Nitin Chandrachoodan
Research Guide
Assosciate Professor
Dept. of Electrical Engineering
IIT-Madras, 600036

Place: Chennai

Date: 18th Jume 2014

ACKNOWLEDGEMENTS

We thank our parents for their constant support and motivation. We are grateful to **Dr. Nitin Chandrachoodan** for his guidance throughout the project. His valuable insights helped the project sail smoothly. We also thank our friends Karthikeyan, Amit Salaskar and Srinivas who enriched us with their knowledge and experience. We thank our college **IIT Madras** for providing us with the tools required for our project.

ABSTRACT

KEYWORDS: Schmidl-Cox, In-phase, Quadrature, OFDM, IP core, State Machine

In this project, an OFDM Tropo-scatter Modem used for communication purpose is implemented on hardware. The overview of the complete system is presented followed by the description of the algorithm used for timing estimation, frequency estimation and correction. Various interfaces present between the major components of the Receiver are also elucidated. The implementation details of the system is presented by explaining the data path and control paths of the system separately. The hierarchical level description and implementation of the system is present in the report. The improvements that are achieved while testing the modem are discussed. The system parametrization followed by using the present parameterized system in designing a similar system is also discussed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology, Madras
OFDM	Orthogonal Frequency Division Multiplexing
QPSK	Quadrate Phase shift keying
LDPC	Low Density Parity Check
IP	Intellectual Property
SISO	Single Input Single Output
MIMO	Multi-Input Multi-Output
LLR	Log Likelihood ratio
RAM	Random Access Memory
FIFO	First In First Out
I/O	Input and Output
MSB	Most Significant Bit
LSB	Least Significant Bit
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
SCFDMA	Single Carrier Frequency Division Multiple Access

CHAPTER 1

INTRODUCTION

As more and more people started using the communication equipments, the demand for high data rate increased quickly. With the increase in data rate the distortion of received signal has increased due to multipath signal fading. To overcome this problem, multi-carrier modulation techniques are used.The principal idea of the multi-carrier modulation technique is that a single high data rate signal can be split into many low data stream signals to overcome multipath fading channel. Traditional multi-carrier modulation techniques inefficiently use bandwidth due to employment of guad band. To overcome this, **OFDM** scheme is employed in which data is modulated onto different carriers which are orthogonal to each other.

The project developed is used by Defense Research Development Organisation (**DRDO**) in their communication during emergency. Hence, the channel should tolerate sufficiently high distortion. An efficient OFDM modulation scheme has been employed in order to obtain a reliable communication when all of the other alternatives are exhausted. In this report, we discuss the different stages present in extracting the received data.

1.1 OFDM

Orthogonal Frequency Division Multiplexing is a method of encoding digital data on multiple carrier frequencies. This is widely used in systems where data rate needs to be high. This scheme employs a large number of closely spaced orthogonal sub-carrier signals which are used to transfer data on several parallel data streams. Each data stream (or sub-carrier) is then modulated with any of the different modulation schemes like Quadrature Phase shift keying (**QPSK**), Binary Phase shift keying (**BPSK**), 64-Quadrature Amplitude Modulation (**64-QAM**) at low symbol rate on each stream thus maintaining a data rate equal to that of a single carrier of high data rate over same bandwidth.

The primary advantage of OFDM is that it has a low Inter Symbol Interface because of low symbol rate due to multiple carriers. Channel equalization is easy because of various slowly modulated narrow band signals.

The following sections deal with the modulation scheme and encoding scheme employed in the project.

1.2 Modulation

Information such as audio and video are transmitted from one point to another using radio waves. This is done by modulating an RF signal (or carrier) with information to be transmitted. Modulation is the variation of one or more properties of an RF signal to represent the information being transmitted. The different methods of modulation are

1. Amplitude Modulation : Amplitude of carrier is varied
2. Frequency Modulation : Frequency of carrier is varied
3. Phase Modulation : Phase of the carrier is varied

1.2.1 QPSK Modulation

Phase-shift keying (PSK) is a digital modulation scheme that modulates the phase of a carrier wave. PSK uses a finite number of phases, each assigned a unique pattern of binary digits. Usually, each phase encodes an equal number of bits. Each pattern of bits forms the symbol that is represented by the particular phase. The demodulator, which is designed specifically for the symbol-set used by the modulator, determines the phase of the received signal and maps it back to the symbol it represents, thus recovering the original data.

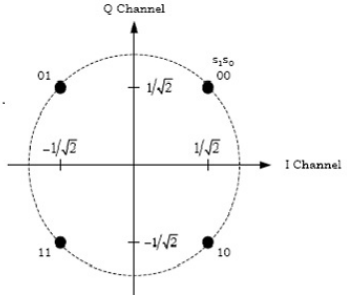

Figure 1.1: QPSK Mapping Constellation

QPSK modulation is the scheme employed in the project. It uses four points on the constellation diagram as in Figure 1.1, equi-spaced around a circle. With four phases, QPSK can encode two bits per symbol as shown in the diagram.QPSK can be used to double the data rate compared with a BPSK system while maintaining the same bandwidth of the signal. Thus it takes two bits as inputs and outputs an **In-phase** and **Quadrature** components. Modulation which is done during transmission is inverted by demodulating with the same mapping in the receiver for further computations.

1.3 Encoding

The general purpose of encoding a system is to make sure that there is no loss in data or to check for its correctness .We have used an encoding scheme that would add parity bits for the given set of data. Hence, the number of bits that are transmitted will be more than the number of message bits. The ratio of number of message bits to the number of data bits transmitted after encoding is said to be **code-rate**.
The encoded message bits when received, are retrieved with the help of the parity bits.

1.3.1 LDPC

LDPC codes are widely used in communication systems for error free transmission. It is a linear block code which is represented by sparse density matrix. It is graphically represented by Tanner Graph.

CHAPTER 2

Specifications and Overview of System

This section details the specification of the system, followed by a brief overview of transmitter and receiver respectively.

2.1 Specifications of the system

Messages are transferred via frames. Each frame consists of a preamble followed by 9 OFDM symbols. Each symbol, including the preamble, consists of **In-phase** and **Quadrature** components.

The **preamble** is used in detecting the start of frame. It has a data length of 64. Preamble helps us in estimating the frequency offset of the sub-carriers occurred due to channel distortion.

The number of symbols in a frame depends on the time interval over which the channel is assumed to be constant and the data transfer rate i.e if the channel is assumed to be constant over certain time, the number of symbols that can be transmitted during this time with specified data rate is the maximum number of symbols that can be accommodated for a given frame. For the frame architecture the number of symbols were decided to be 9 in order to optimize the data rate and not incur a lot of noise.

Each symbol is of length 544 of which 512 are data and the rest 32 corresponds to cyclic prefix.

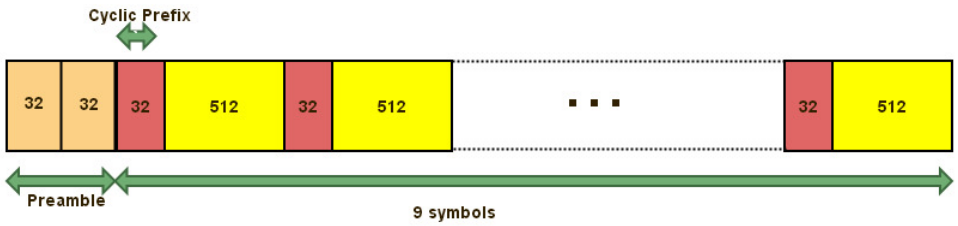

Figure 2.1: Frame Structure

Cyclic Prefix refers to prefixing of a symbol with the data at the end. It is used instead of a guard interval between two symbols to minimize Inter-Symbol Interface (**ISI**) between consecutive symbols. This repetition of end symbol allows us to view the linear convolution as a circular convolution. This allows us to estimate the frequency offset of sub-carriers corresponding to each symbol with a higher degree of accuracy but it has not been implemented in the hardware yet as a rough estimate has proved to be sufficient.

The 512 data values in one symbol correspond to 384 message bits, 46 Pilots and 82 Nulls. **Pilots** and **Nulls** corresponds to specific locations of symbol as decided by the designer which are used in estimating the channel distortion.

Therefore the number of data signals corresponding to each frame is

$$64 + (384+46+82+32)*9 = 4960$$

Thus each frame contains **4960** In-phase and Quadrature components, of which **3456** (384*9) values correspond to message in-phase and Quadrature symbols. These QPSK-modulated symbols correspond to **6912**(3456*2) data bits. Since the code-rate of the given system is $\frac{2}{3}$ the number of actual message bits in each frame is **4608** ($\frac{2}{3}$ *6912).

Therefore, each frame corresponds to **4608** message bits. The data transmission rate is **20 Msymb/s** i.e., 20 Mega data symbols per second.

2.2 Overview of Transmitter

This section deals with data flow associated with transmitter part of Modem.

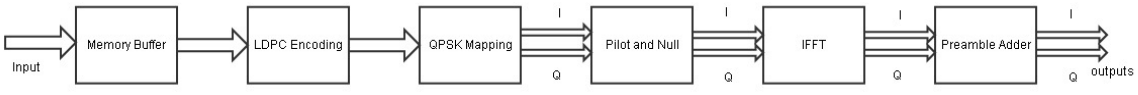

Figure 2.2: Transmitter block diagram

The figure shown above reflects different stages present in transmitter

The input which is a stream of bytes is first stored in a memory buffer. When there is sufficient amount of data to be processed in buffer, it raises a flag signalling to the next

stage that the amount of data has reached threshold value. The encoding stage raises a flag when its free for computation, and when both flags are high the data is transferred from buffer to encoding stage.

Once the data reaches the next state, there will be a **LDPC** encoding done on data which results an increase in bit count. After encoding of data it raises two flags, one for the previous stage stating that it is ready to encode the next set of data and second flag signalling the next stage that data is ready for computation. For each frame the encoding is done in 3 bursts. Each burst corresponds to message signals corresponding to 3 symbols in the frame. After giving the 'done' flag to the next stage and obtaining a ready flag from the next stage(pilot and null) data will be transferred.

The encoded data now will be going through a **QPSK** modulation. The inputs to this stage are **2 bits** per cycle. This is a combination module present in LDPC encoding stage which upon depending input bit values results in giving corresponding **In-phase** and **Quadrature** components as outputs.There will be no flags to this stage as soon as it gets the input it outputs the QPSK symbols. Thus the LDPC stage will give out the QPSK modulated symbols once it receives the flag from the next stage(pilot and null).

The input flag to the present stage (insertion of pilots and nulls) is obtained from encoding stage. The input stream is then stored in a buffer. In this stage we add pilots and nulls to the obtained data depending on the data location value. Upon obtaining flag from next stage(IFFT), depending on the data count value corresponding pilot and null location is added to input data and transferred. The pilot and null stage gives the flag to the previous stage(encoding) once it receives a flag from IFFT stage.

The input start flag for IFFT stage is obtained from encoding stage signalling it to be ready for the data that is present in pilot and null stage. Once the start signal is obtained from encoding stage, the IFFT stage will raise a ready flag to the previous stage(pilot and null). In this stage IFFT(Inverse Fast Fourier Transform) is done on input data which in turn maps the input into different orthogonal sub carriers.A 512 point IFFT is done in the project which is number of data signals corresponding to each symbol. Cyclic Prefix is added by this stage before the data is transferred into next stage. The output of this stage will be a symbol corresponding to a data of 544 consecutive cycles. Once the computation of first symbol in a frame is done it raises a flag for the next stage.

Once the flag from IFFT stage is obtained the Preamble adder stage adds the preamble which is read from a file for the first 64 cycles and then transmits the rest of symbols it gets.

These above mentioned are the different stages that are implemented in transmitter over here. For transmitting it over a channel(for actual communication purposes) there are further stages involved. The digital stream is first up-converted and then it is translated through a DAC(Digital to Analog Converter). The resultant analog signal is then multiplied it with a carrier frequency for transmission.

2.3 Overview of Receiver

This section deals with different stages that are present in Receiver part of Modem. The transmitted signal which is obtained at the receiver end is initially multiplied with the carrier frequency in order to get the signal into base-band, the signal is then fed into ADC(Analog to Digital Converter) and down-converted in order to get the data in required change thus obtaining the Inphase and Quadrant-phase components of the received signal.

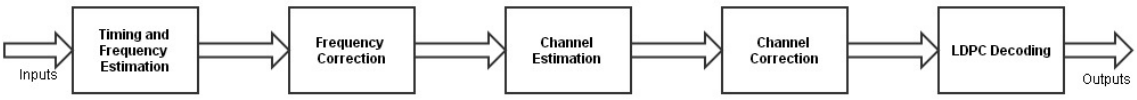

Figure 2.3: Receiver block diagram

The in-phase and quadrature components go through the same stages. Hence we will be discussing different stages present in receiver with respect to either one of the components.

The Input data stream first goes into a stage for Timing and frequency estimation. In this stage we will be checking for the preamble over the received data. We have used **Schmidl-Cox** algorithm for this purpose. Using the algorithm we also estimate the offset in phase. The detection of preamble certifies the start of the frame and raises a flag.

After receiving flag from the Timing and frequency estimation stage, the frequency correction module will be having the offset phase as input and data is multiplied with

the corresponding offset with respect to its sub-carrier index which results in phase corrected data. In this stage we also remove the cyclic prefix attached to each symbol before transmitting to next stage. After the correction of first data signal in the symbol it raises a flag for the next stage.

After receiving flag from the previous stage the channel estimation stage starts working. From the received data it detects the pilot and null values from the known data locations values. It then estimates the offset in the pilot values and raises the flag for the next stage.

As it receives the flag from previous stage the channel correction module will have the offset value as it's input. This offset value is used in obtaining the channel corrected data of the received input and raises a flag for the next stage.

The input to this stage will be the message signals in each symbol after removing cyclic prefix,pilots and nulls in the previous stages. The LDPC stage after receiving the flag from the previous stage waits for a complete symbol in order to decode data which is to negate the encoding present in the transmitter stage. The Inphase and Quadrant phase components after decoding will be then fed into an internal module which demodulates the data in order to negate the effect of QPSK modulation present in the transmitter thus resulting in data bits a output.

The output bits thus obtained are stored into a output buffer when it receives certain threshold as specified by the designer results in message which is transmitted.

In transmitter we have employed handshake type of coordination between each module in order to shift the control signal between modules. The current stage requires a flag from the next stage in order to transfer the data to it. Thus establishing a backward co-ordination in each stage

In the case of receiver we have implemented a central controller which depends on the state it is present and the depending on input flag it goes to the next state raising a flag to the corresponding next state.

2.4 SISO & MIMO

The Receiver can be implemented in two different fashions such as **1. Single Input Single Output(SISO)** : Receiver has a single stream of Inputs coming from a channel. **2. Multiple Input Multiple Output(MIMO)** : Receiver has a multiple stream of inputs coming from channel. This is used mainly in order to mitigate multi-path fading effects. All the input streams go through similar stages and gets combined before going to LDPC stage.

CHAPTER 3

Algorithm and Interface Description

In this chapter, the **Schmidl-Cox** algorithm that is used in timing and frequency estimation in the project and the following stages that are present before the channel estimation shall be discussed. Also, the interfaces present between all the major stages present in the system for both **SISO** and **MIMO** implementations are detailed.

3.1 Schmidl-Cox Algorithm

This algorithm is used in achieving 'Timing and Frequency synchronization'. The preamble that is added in the final stage of transmitter needs to be detected during this stage. It requires both in-phase and quadrature components as the inputs.

The preamble is chosen such that it satisfies a particular property. The preamble consists of two highly correlated sections of equal length. The in-phase component needs to be symmetric about its centre, while the quadrature component needs to be anti-symmetric about the same. Hence, the auto- cross- correlation when performed on these values are strictly monotonous till the end of **correlation depth**, which in this case is 32. Once a transform on these correlation values crosses a certain threshold (set by the designer) and remains constant for a safety period, the preamble is said to be detected. From these computations, the phase offset by which all of the QPSK data symbols have to be rotated is also calculated.

The calculated offset value is used in phase correction. To obtain this we need to multiply the phase offset value with sub-carrier index and the resultant offset value is used in phase rotating the corresponding data with same sub-carrier index. After the phase rotation, the data it is fed into channel estimation module.

3.2 Description of interfaces

Interfaces between consecutive modules are very important to ensure smooth flow of data and to remove large overheads in computation. The following are a few interfaces implemented in the design.

3.2.1 Interface for Channel estimation Module

The outputs of the Frequency Correction module will be one data symbol per clock cycle. But the input to the Channel Estimation block requires a certain delay between two consecutive OFDM symbols. The interface should ensure that it takes inputs from the former in each consecutive cycle and output with delay between each consecutive symbol.

3.2.2 Interface between Channel Estimation and LDPC

The interface between Channel estimation and LDPC is tricky in case of MIMO system. Channel correction Module gives out a 'valid' signal whenever it outputs data thus the LDPC records in a symbol data subject to the 'valid' signal. In MIMO system, to keep things simple, an FSM waits for a small period of time after the arrival of the first 'valid' signal from an Rx chain, then proceeds with the transfer of soft LLRs to the decoder based on number of channels which are active by the end of that period. However, this method outputs garbage values if a preamble is detected in one of the chains because of noise.

3.2.3 Interface between the LDPC decoder and primary outputs

The output of LDPC decoder are 21 bit LLRs (3 concatenated 7 bit LLRs) from each of which 3 message bits can be obtained. The primary outputs are needed to be the same as the primary inputs to the transmitter, which are bytes of messages. Hence, we pack 3 bits of messages from each output LLR into a 24-bit register. When it is filled, we send out a byte of message in each of 3 consecutive cycles.

CHAPTER 4

Overview of Hardware

In this chapter, the hardware specifications and different types of IP cores used in the project are discussed.

4.1 Hardware Specifications

The project is targeted for FPGA implementation. Depending on size of the design, Xilinx's **Virtex-7** board has been chosen for the hardware realization, **XILINX 14.3** version is used as the platform to program it and the Hardware Description Language used is **Verilog**.

4.2 IP cores

This section specifies different type of Intellectual Property (IP) cores used in the project. The input and output specifications to each IP core and different signals that are associated with each one are provided.

Shift Register : The single input can have a variable bit width either signed or unsigned. This module implements the functionality of D register. The depth of the shift register can be chosen by the user.

Normal Multiplier : The two inputs and the output of this IP core can be signed or unsigned and their bit widths can be chosen when instantiating. The latency of output can be set as a variable, depending on the performance/resource requirements. The enable signal can be chosen as per user's choice or it can be always set.

Complex Multiplier : The inputs to this IP core can be either signed or unsigned complex numbers whose bit width can be chosen. Thus it will have 4 inputs

corresponding to real and imaginary parts of the two numbers. The output will be complex product of the inputs which can chosen to be either signed or unsigned whose bit width and choice of bits can be chosen at same time. This can either have a clock as input or not. The latency of output can be set as a variable. The enable signal can be chosen as per users choice or it can be always set. The functionality implemented is

out_r = in1_r * in2_r - in1_i * in2_i

out_i = in1_r * in2_i + in1_i * in2_r

r= real ; i = imaginary

inputs : in1, in2; outputs = out

Accumulator : The single input to this IP core can be either signed or unsigned complex numbers whose bit width can be chosen. It contains the clock, enable, clear as inputs. The output being the accumulated value of input obtained on each clock edge when there is an enable. The output can chosen to be either signed or unsigned whose bit width and choice of bits can be chosen at same time. The functionality is

next_value = Input + current_value

current_value = next_value

CORDIC : The CORDIC IP core operates in two modes

1. Rotation Mode : This mode is used in calculating sine and cosine of given input angle. The input in this mode will be an angle whose bit width is 16. It is represented in 3.13 format which signifies MSB represents signed bit,next 2 bits corresponding to integer and the rest corresponds to fractional value when the angle is represented in radians. The output will be sine and cosine corresponding to input angle of 16 bit wide. It is represented in 2.14 format which signifies MSB as signed bit, next bit corresponding to integer and rest 14 corresponding to fractional value. The accuracy of the output depends on the number of CORDIC iterations or latency of the output. The enable signal can be chosen as per users choice or it can be always set.

2. Vector Mode : This mode is used in calculating the angle from input sine and cosine value. Thus internally it calculates tangent value and then calculates its

arctan value. The input has 2.14 format which signifies MSB as signed bit,next bit as integer value and rest bits for fractional value. The output will be angle corresponding to inputs in 3.13 format. MSB signifies signed bit, next 2 bits represents integer value and rest of the bits represent the fractional value when the angle is represented in radians. The accuracy of the output depends on number of iterations or latency of the output. The enable signal can be chosen as per users choice or it can be always .

Random Access Memory also known as RAM is an IP core used as an storage element. The Input can be data of chosen bit width. There will be an input write read address and a write read enable signal and clock. The size of RAM can be chosen thus deciding number of address bits. When write enable is set depending on write address, the input is written into specific address on clock edge. The output will be data in the location of read address and obtained on clock edge when read enable is high.

First in First Out also known as FIFO is another commonly used storage element. The depth of FIFO is variable. It has data to be fed in, write enable and read enable as inputs. It has output data, empty full as output signals. When the write enable is high the data enters into FIFO on clock edge. When the read enable is high it outputs data in the order it has entered into FIFO. It has an additional empty signal that signifies the FIFO is empty and full signal signifying the data count has crossed certain threshold value as set by user.

DCM is the Digital Clock Manager. This IP core is used in generating the clock of required frequency. The input to this module can either be a differential clock or single ended clock. It can take the input of low frequency and can output the clock with high frequency and vice-versa.

ICON also know as Integrated Controller. This IP core is used to control the interface between the hardware and ILA, VIO IP cores. Instantiating this IP core and giving the control signal to ILA and VIO will ensure a interface between them and hardware.

ILA is the Integrated Logic Analyzer. This IP core is used for analyzing any of the data signals insitialized in the instantiated module. The data values that required for

validation or debugging can be obtained from this IP core after implementation. Each ILA will have 16 trigger ports each of maximum width 256 bits. The depth (number of samples) can be chosen till 131072. But the problem with ILA being we can view the signals only where it is instantiated. The internal signals cannot be verified using ILA. There should be a control signal corresponding to each ILA from an ICON IP core.

VIO is Virtual Input and Output. This IP core is used in controlling the input and output during run-time. Any internal signal cannot be set to users in implementation i.e if a signal is given a particular value after implementation it remains as that particular value but if the signal is given to VIO then the signal can be changed at run-time after implementation. We can also write the required values from VIO onto a file. There is a synchronous input, output and asynchronous input,output for this IP core. The Input to VIO are signals that can be altered by user and outputs from VIO are values that can be written into a file. The bit width of Input and Output can be chosen with a maximum of 256 bits. There will be a control signal corresponding to each VIO from ICON.

CDC also known as Chipscope Definition and Connection File is a source type that is used as a user interface for selecting signals that can be viewed after implementation. This module works after synthesis; we can pick the signals that are required manually. Internal signals can be viewed in this if the design is synthesised with hierarchy. This is far more easier to use than the ILA and ICON IP cores. Both of them result in same thing which is viewing the signals in chipscope.

CHAPTER 5

Algorithm Implementation

The first step in implementing a design is to identify the data path and control path. After this, we need to build the data path with modules containing suitable IP cores and necessary control signals as inputs. Finally we need to setup a control path by building a state machine that guides the control signals. The major part while implementing a system is to check the timing of data which includes checking that data is stable by a clock edge.

In this chapter, different modules present in the project are discussed. The control block that is used in transferring control signals between each module is also detailed. Interface present for channel estimation module.

The receiver has a reset signal (sclr), a clock generated by DCM, the in-phase and the quadrature components as inputs. Initially, the reset signal is set to reset all the modules. The DCM takes the differential clock of 100Mhz from the board and the output clk of frequency 20Mhz is used as the system clock.

The in-phase component is represented as **r** real part and quadrature component is represented as **i** (imaginary) part in the project.

There is a **top_connect** module that is used in connecting different internal modules such as

1. top_fsm
2. Freq_correct
3. Freq_correct_mult
4. Channel_est_fsm(Interface for channel estimation.)

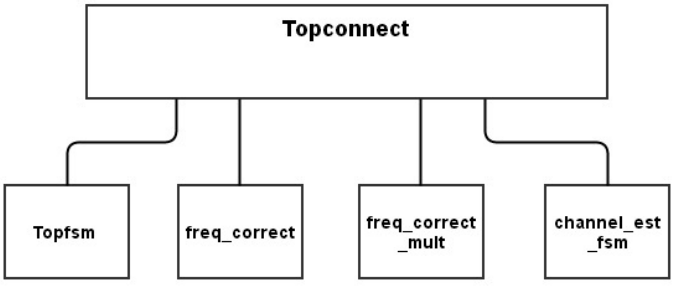

Figure 5.1: Top level description

5.1 Top Finite State Machine

The top_fsm contains the control block that is used in controlling the enable signals to the rest of the modules. There also two modules instantiated in the module named

1. Memstore
2. Topmsandc

The Memstore and Topmsandc are explained here while the control block will be explained in the later part of the chapter.

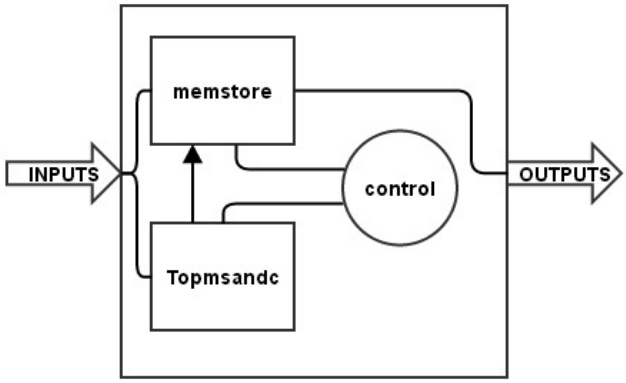

Figure 5.2: Topfsm Block Diagram

5.1.1 Memstore

This is the first stage of the receiver. The Inputs are first fed into a RAM whose write enable is always set. The data goes through several stages later in order to detect the preamble which has some latency. Therefore memory blocks are set up at first to not lose data. The write address keeps incrementing for each clock cycle after the reset signal.

Inputs	Outputs
Received **data_i** and **data_r**	**data_i** and **data_r** removing preamble
Read Enable,clk,sclr	
Preamble location	

Table 5.1: I/O of Memstore module

There is a read enable and preamble location that comes from the state machine (control block), once the read enable is obtained the read address starts from the address which is location value shifted by 32 in order to remove the first cyclic prefix and then read address keeps on incrementing till the enable signal from top_fsm is present. Thus, the next cycle after enable results in giving the first data signal in first symbol of the frame. The size of the memory block depends on the latency of the topmsandc module.

The input which is fed into memstore is also fed into Topmsandc at the same time in order to check for the preamble.

5.1.2 Top Schmidl-Cox module

This module sets up a data path for Schmidl-Cox by instantiating different modules which only contain IP cores. It also contains a module named **comp** which checks for the relation of correlated values that are obtained from the IP cores and raise the preamble flag.

Inputs	Outputs
Received **data_i** and **data_r**	packet
sclr, clk, clr	phase
Threshold	Preamble location

Table 5.2: I/O to Topmsandc

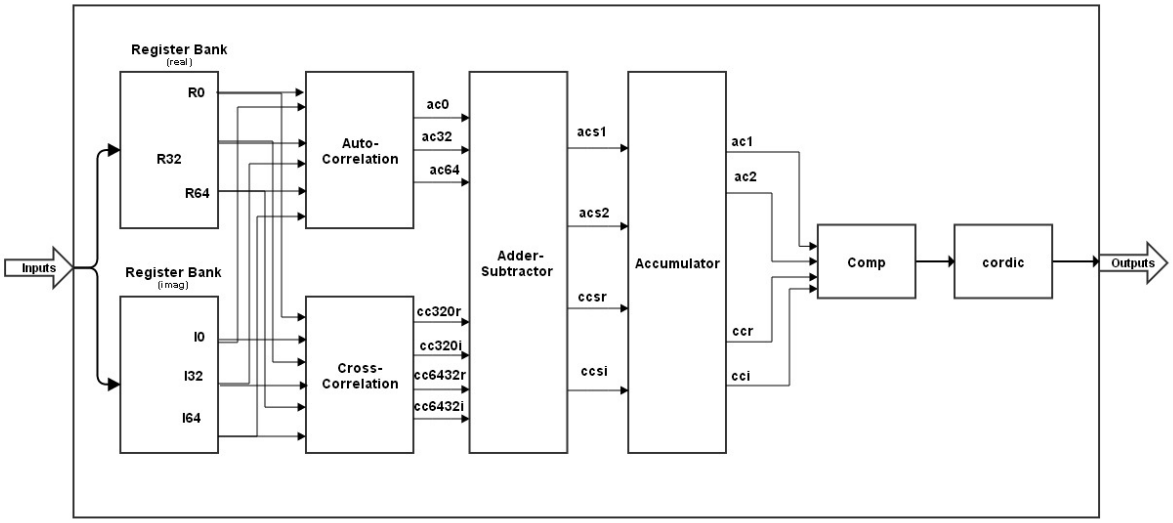

Figure 5.3: Top Schmidl Cox Block Diagram

For calculating the cross and auto correlation values for a correlation depth of 32, we need the present input, the inputs in the 32nd and 64th cycles prior to the present input. For this we can either have a memory storage or a set of 64 shift registers so that we can access the required data.

We have used a register bank module for this purpose which essentially contains two shift registers whose depth is 32. The output of the first shift register is the input to the second shift register. Thus the outputs of each shift register would result in the values which are 32 (w32r, w32i) and 64 (w64r, w64i) cycles prior to the present input (w0r, w0i). We have used register bank for real and imaginary separately. Thus the 6 output values (3 for real and 3 for imaginary) go to the correlator module where correlation values are calculated. Since the correlation depth is 32 the auto and cross correlation should be done over 32 consecutive data values. Since calculating 32 values for each clock cycle results in recomputing the values, we calculate the correlated values for present data and data which is 32 cycle prior to present and subtract them then feed into a accumulator which results in obtaining auto correlation of consecutive 32 cycles starting from present input.

The similar thing is done with data which is 32 and 64 cycles prior to the input. We calculate the correlated values for data which is 32 and 64 cycles prior to present and subtract them then feed into a accumulator which results in obtaining auto correlation of consecutive 32 cycles starting from data which is 32 cycle prior to input. Thus we obtain the auto-correlation values present in two section of the preamble.

We calculate the cross-correlated values for present data, data which is 32 cycle prior to present and for data which is 32 and 64 cycles prior to present and subtract them then feed into a accumulator which results in obtaining cross correlation over the entire preamble.

The above architecture need to be implemented in the hardware in a 'correlator' module. The correlator module calculates auto-correlation and cross correlation of the input data. Five complex multipliers are instantiated; 3 for auto-correlation and 2 for cross correlation. The bit width and choice of bits is chosen as per the C code.

The auto-correlation yields only real values. Thus the outputs of this part are **ac0, ac32, ac64** which are auto correlated values of present data, data which is 32 and 64

cycle prior to present data respectively.

$$ac0 = w0r * w0r + w0i * w0i$$
$$ac32 = w32r * w32r + w32i * w32i$$
$$ac64= w64r * w64r + w64i * w64i$$

For calculating the cross correlation, the inputs to the first complex multiplier are conjugate of present data value and the data which is 32 cycle prior to present giving the cross correlated value. This cross correlated value contains both real and imaginary parts (i.e cc320r, cc320i). The inputs to the second multiplier is conjugate of data which is 32 cycle prior to present data and data which is 64 cycle prior to present data thus resulting in cc6432r, cc6432i.

$$cc320r = w0r * w32r + w0i * w32i$$
$$cc320i = w0r * w32i - w0i * w32r$$
$$cc6432r = w64r * w32r + w64i * w32i$$
$$cc6432i = w32r * w64i - w32i * w64r$$

Now these 7 outputs go to the next module which is to subtract the corresponding values in order to maintain the correlation depth of 32. There will be 4 subtractor IP cores instantiated in this to subtract corresponding correlated values. Thus resulting in two auto correlated (acs1,acs2) values corresponding to two sections of the preamble and a cross correlated value which contain both real and imaginary parts (ccsr,ccsi). Thus the adder-subtractor results in 4 outputs which are

$$acs1 = ac0 - ac32$$
$$acs2 = ac32 - ac64$$
$$ccsr = cc320r - cc6432r$$
$$ccsi = cc320i - cc6432i$$

These values are now fed into accumulator achieving the correlation value over length of 32. These resultant values are now used in checking the condition for the relation to cross the threshold value. The accumulator contains 4 accumulator IP cores instantiated in it. Each accumulator takes single input thus resulting the accumulated value of the input. Thus the accumulator results in 4 outputs corresponding to 4 inputs.

These correlated values are now fed into comp module which checks the relation between these correlated values. This part of the code is just as same as golden C code since this is just combinational stage.

In the comp module it checks whether correlation has crossed certain threshold. This threshold is also the input to it. Once the relation between them crosses the threshold it raises a 'packet' flag stating that preamble is detected and it also gives corresponding cross correlated value at that specific cycle as output. The comp module also gives the preamble location as its output. Here the preamble location should be relative to the address of the data stores in memstore. This situation has been overcome by running the time counter that is in par with the write address corresponding to the memstore (i.e both keep on incrementing from the next cycle after reset).

The output cross correlated values from the comp module is fed into CORDIC IP core which is in vector mode to get the phase offset value. The number of CORDIC iterations is set to 20.

5.1.3 Conclusion to Top finite state machine

In topmsandc at the comp module stage we obtain the packet which is fed to control block and the preamble location which is fed into memstore. But the memstore waits for the enable signal from the control block. The CORDIC IP core gives the phase offset value which is used in next stages. The latency of this stage is 19 (1 for accumulator, 18 for CORDIC). The latency of the multiplier is kept to be 0.

5.1.4 Difficulties in Top finite state machine

1. To achieve bit accuracy with respect to the C code.
2. To achieve synchronization between the location of preamble obtained in Schmidl-Cox module and the corresponding address in memstore module.
3. Taking into account the amount of latency in hardware in order to match the preamble location with that of C code's in comp module.
4. Resetting the Schmidl-Cox module after a preamble is detected so as to clear the buffers before the next frame arrives. This problem is dealt when control block is ex-

plained.

5. The bit level accurate model for CORDIC is not present. Thus to validate the outputs of CORDIC with reference has been a major task.

5.2 Frequency Correction Module

This module is used in calculating the sine and cosine values required for corresponding phase shift. This module contains a multiplier IP core and CORDIC IP core which is in rotation mode.

Inputs	Outputs
Phase	Sine
sclr, clk, clr	Cosine
freq_enable	Corr_enable

Table 5.3: I/O to Freq_correct

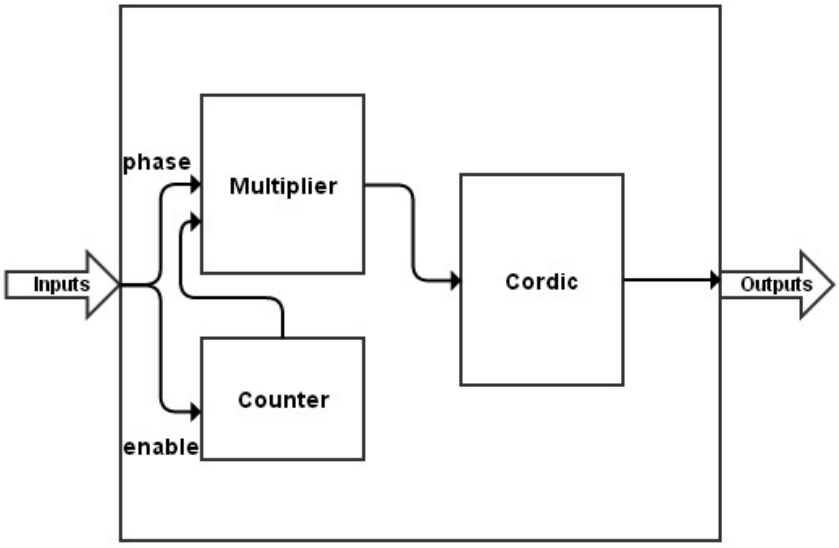

Figure 5.4: Freq_correct_block Diagram

To get the phase offset corresponding to each sub carrier,the phase that is calculated from the Schmidl-Cox should be multiplied with respective sub-carrier index.

The counter that is present will start when the input Freq_enable is set by the control block present in topfsm. Once the enable is obtained the counter increments on each

positive clock edge for clock cycle . The multiplier thus will have the phase as one input and the counter value which increments on every clock cycle as other input. The output of the multiplier thus will output the phase offset corresponding to each sub-carrier.
This phase offset value is then fed into CORDIC which is in rotation mode in order to get the corresponding sine and cosine values. The sine and cosine obtained from CORDIC are shifted accordingly as mentioned in C in order to be in par with it.
The enable to the next stage goes from this module. The correction enable (corr_enable) which is the enable signal to freq_correct_mult stage is the output of this module. This enable is set after obtaining the data from the CORDIC IP core, since the CORDIC IP core will give the output with latency of 18 therefore we need to raise the corr_enable 18 cycles after obtaining the freq_enable. The multiplier is set for latency of 0.

5.3 Frequency correction and multiplication module

This module is used in obtaining the phase corrected value of the input data received. This module contains a Complex multiplier IP core

Inputs	Outputs
data_i ,data_r	data_out_freq_r
sine,cosine	data_out_freq_i
corr_enable	channel_enable
sclr,clk	

Table 5.4: I/O to Freq_correct_mult

The complex multiplier will be enabled whenever there is corr_enable from the previous stage. The sine and cosine obtained from previous stage(freq_correct) is multiplied with data from memstore. The enable to memstore is given by the control block present in topfsm such that the sine and cosine values and data values corresponding to same sub-carrier are given as inputs such that they occur at the same clock edge to the complex multiplier present. The result of the complex multiplier is shifted accordingly as mentioned in C in order to obtain the phase corrected values of the data received.

data_out_freq_r = data_r * cosine - data_i * sine

data_out_freq_i = data_i * cosine + data_r * sine

Since the complex multiplier latency is set to 0, the channel_enable is raised as soon as the corr_enable is raised. Before discussing the Interface for channel estimation we will first discuss the control block present in topfsm.

5.4 Control Block

As discussed above the enable signals to the module are controlled by a single controller which is present in topfsm. We first present different states present and the enable signals to each stage.

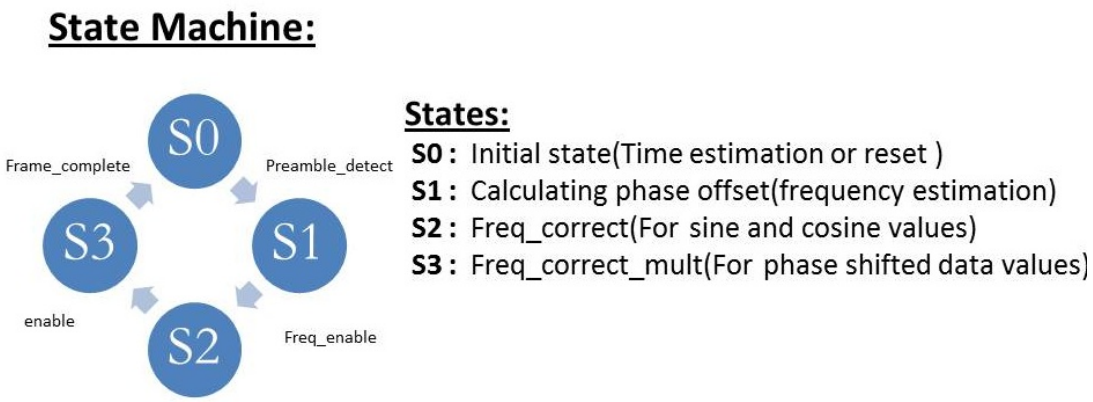

Figure 5.5: Control block of Topfsm

A Melay based state machine is built here i.e the next state depends on present state and the input. Initially, the system is in S0 state, where the system checks for the preamble. In this state the comp, the memstore and topmsandc modules are active since memstore has data feeding into it and the comp module checks for the preamble all the remaining modules wait for an enable signal. Once the preamble is detected it gives a preamble detect signal that changes the state of the state machine. At this stage only preamble is detected and phase it yet to be computed.

When the system is in S1 it sends a clear signal to the modules. This clear signal (clr) is used as a reset signal for the modules, this plays a main role when the preamble is detected again after the first frame is completed since all the modules should go to initial state for this frame. The clear signal also goes to the topmsandc and comp module

thus clearing all the internal values for next frame to be detected. In this stage, the cross correlated values which are fed into comp module which is in vector mode to compute the phase offset value. Since the CORDIC module outputs the phase after 18 cycles it gets the input therefore the system stays in S1 stage for 18 cycles. The count for which the system stays in S1 stage depends on number of CORDIC iterations or latency of the CORDIC. After this 18 cycles it goes to the S2 stage raising cordic_ready which is the enable signal for the freq_correct module(freq_enable).

The system in S2 stage calculates the phase offset corresponding to each sub carrier. As mentioned above in the freq_correct section it takes 18 cycles to compute the corresponding offset after the enable signal. This 18 cycles is based on CORDIC latency which can be varied. Thus after 18 cycles the system which is in S2 state goes to the S3 state raises an enable signal for the memstore. The system in S2 stage (freq_correct) after 18 cycles gives an corr_enable to the freq_correct_mult therefore the control block will give an enable signal to the memstore after 18 cycles such that data from memstore and sine, cosine values from freq_correct arrive at the same time at inputs of freq_correct_mult stage.

The system will be in S3 state till the whole data in frame is phase corrected then it raises a frame done signal and goes to the initial stage.

The channel_est_fsm module present in top connect is the interface for channel estimation module. The phase corrected value will have an enable signal to it. This phase corrected value is fed into the interface for channel estimation.

Module	Area (No. of slice LUTs)	Max. operating frequency (MHz)
Top Finite State Machine	1791	159.7
Freq. correction	1296	421.5

Table 5.5: Hadrware Area and Timiming Specification

CHAPTER 6

Interfaces Implementation

In this chapter we first discuss the different interfaces present in the system :

1. Interface for channel estimation
2. Interface between LDPC decoder and the primary outputs
3. Interface between channel estimation and LDPC present in MIMO

6.1 Interface for Channel Estimation Module

We have a stream of phase-corrected data of the whole frame i.e., OFDM symbols which include cyclic prefix and a flag (channel_enable) that signals the channel estimation stage to start acquiring data. But for the channel estimation module, inputs are required to not be a continuous stream and also cyclic prefix needs to be removed. The channel estimation wants a continuous stream of 512 data corresponding to a symbol and a gap of 64 i.e it requires 64 buffer cycles between two consecutive symbols.The interface module which is described in the next section ensures the above mechanism.

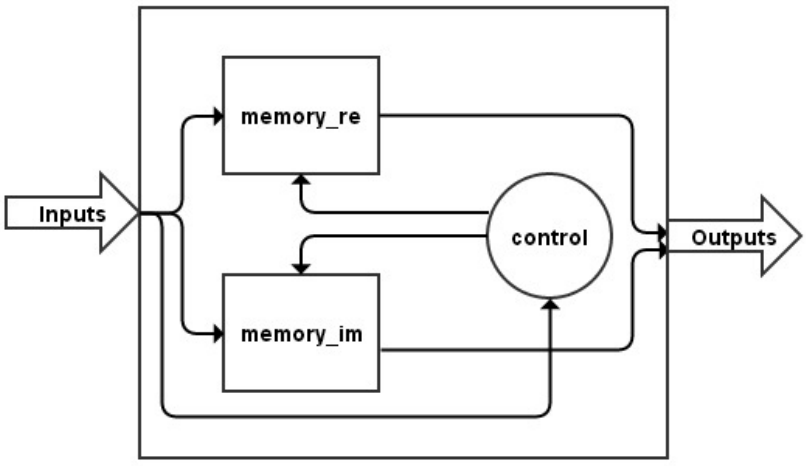

Figure 6.1: Block Diagram of Interface for channel estimation

The interface contains two RAMs and a simple state machine which takes the inputs from the freq_correct_mult stage whenever the enable (channel_enable) is present and outputs the required values in suitable fashion to the channel estimation.

Inputs	Outputs
data_out_freq_r	data_out_final_r
data_out_freq_i	data_out_final_i
channel_enable	rdy
sclr,clk	

Table 6.1: I/O to Interface for channel estimation

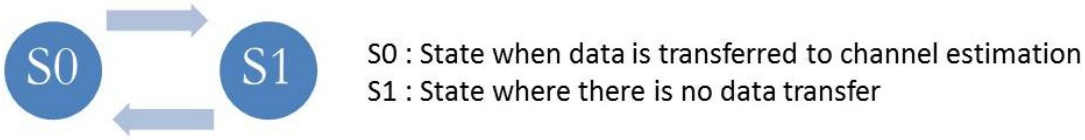

Figure 6.2: Control block for Interface for channel estimation

When the channel_enable is set the write enable signal for the RAM is set thus the input data gets recorded into RAM each clock cycle with input address counter incrementing continuously till the frame end. Now once the channel enable is obtained it waits for 64 cycles and then read enable is enabled. Remember the cyclic prefix of the first symbol is removed as soon as the preamble is detected. Hence there is no need to skip the address by 32 for first symbol.

Once the read enable is enabled the read address is incremented continuously for 512 clock cycles to ensure the data is fed into channel estimation module. The data is transferred with a valid signal. Thus the system is in state S0 where the data is transferred. After transferring 512 data values corresponding to a symbol the system goes into S1 state where it remains ideal for 64 clock cycles. After 64 clock cycles the read address is skipped by 32 in order to remove the cyclic prefix corresponding to the symbol.

There is a catch in this architecture. The system will be active only when the channel enable is set. But for the 9th symbol the channel enable goes low before transmitting 9th symbol to the channel estimation module due to buffering cycles present present between each symbol.

The channel enable is enabled for

512 + 544*8 = 4864 clock cycles

But the duration over which frame data is transferred to channel estimation module is

(512 + 64) * 9 = 5166 clock cycles

To overcome this problem for the 9th symbol even after the channel enable has gone low the read address keeps on incrementing till it completes the 9th symbol and again it goes to the initial state.

The size of the memory should be a minimum of buffering cycles present in the frame which is 64*9 = 576 i.e memory needed to accommodate 576 data signals.

6.2 Interface between LDPC decoder and the primary outputs

The outputs from LDPC decoder are LLRs of bit width 21 of which only three bits are message bits. The LDPC gives 3 LLRs in 3 consecutive cycles remains idle for 1 cycle and again transmits the LLRs. The output from the receiver is expected a single byte per cycle. The interface module present over here takes care of the conversion.

The interface module contains a FIFO IP core for giving the outputs in byte fashion. The module name is final_fifo in the project.

Inputs	Outputs
dout_ldpc	dout
input_valid	output_valid
sclr, clk	

Table 6.2: I/O to Interface for LDPC and Outputs

The output of LDPC decoder are 21 bit LLRs (3 concatenated 7 bit LLRs) from each of which 3 message bits can be obtained. They are sent out of the decoder with a 'valid' signal, which when raised indicates a valid output. The primary outputs are needed

to be the same as the primary inputs to the transmitter, which are bytes of messages. Hence, we pack 3 bits of messages from each output LLR into a 24-bit register. When it is filled, we send out a byte of message in each of 3 consecutive cycles.

An internal counter is present which increments each time the input valid signal is present. In each LLR of width 21, the bit locations of 7, 14 21 correspond to message bits. After retrieving them, the 3 bits are shifted and stored into a 24 bit shift register. Once the counter reaches a value of 8, it signifies that the 24 bit register is fully loaded. So the next step would be unpacking the message bits.

After packing the 8 LLRs into a 24 bit register it raises a 'pack' flag stating that packing has been completed. In the next three clock cycles it unpacks the 24 bit value into three message bytes. It also raises a valid signal to FIFO whenever there is data transferred to FIFO. The data is thus recorded into FIFO, whenever it crosses the threshold value it raises a full flag then the read enable is given to FIFO by an external source.

6.3 Interface between channel estimation and LDPC present in MIMO

The channel estimation module uses pilots in the OFDM symbols to output data symbols corrected for channel distortions. It sends out data subject to a 'valid' signal. The LDPC decoder, with the help of this enable, feeds the data into its corresponding buffers. This is straightforward to implement in case of SISO.

In case of MIMO, the outputs from all the streams need to be combined into a single stream. The following section details the methodology implemented to combine outputs from all the receiver chains into a single stream.

6.3.1 Description of the state machine used in the interface

The channel estimation module gives 384 message signals in a burst and then wait for 192 cycles to give the next burst of data. In case of SISO there is no need for any storage for the interface. The LDPC records data whenever the enable is set to 1 by channel estimation.

In case of MIMO the outputs from the channel estimation is fed into RAM with the write enable being the enable given by channel estimation and write address incremented in each clock cycle till the enable is present. Thus corresponding to each stream there will be 2 RAMs one for real and other for imaginary. The storage unit is present because of the delay that might be present between streams and to send the data in coherence to the decoder.

The architecture that is implemented waits for the first 'valid' from any channel. After that, the state machine waits for a nominal 10 cycles and then check the valid signals corresponding to each stream. The stream which has valid as 1 is data to be considered. After the 10 buffer cycles the read enable to RAM is set to 1 and the read address is incremented for 384 cycles then the read enable is set to 0 and read address stops incrementing for next 192 cycles since there is a buffering gap of 192 cycles between each symbol for the channel estimation output.

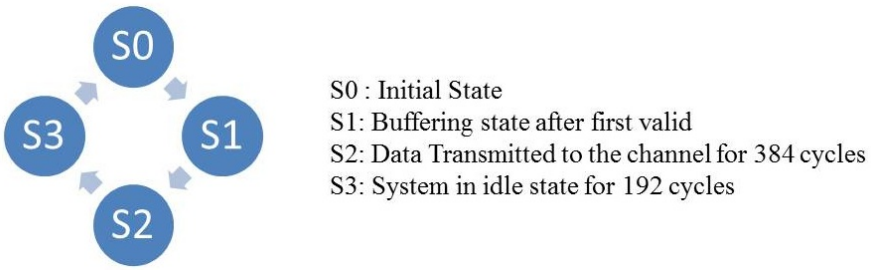

Figure 6.3: Interface for channel estimation and LDPC in MIMO

Initially the system in state S0. The valid signals from all the streams are ORed so that if any of the valid signal becomes 1 then it is set to 1. When this signal becomes 1, the system goes into S1 state. In this state there is a counter present which increments on every cycle, once the count values reaches ten the system goes into S2 stage. In this stage the streams which have valid signal as 1 gets the read enable 1 to the RAM present in that stream. The same read address is given to all the four streams to achieve coherence in the output data from RAM. The data that comes from the RAM is ANDed with the valid signal corresponding to that stream before going into LDPC decoder.

The read address increments for 384 cycles as long as the system is in state S2. After 384 cycles the system goes into state S3 where the read enable is set to 0 and read address stops incrementing. The system will be in this state for 192 cycles and then goes back to state S2. This happens for 9 cycles corresponding to 9 symbols in the

frame and then the system goes to the initial stage.

The streams that need to be considered for the computing can be managed by user by altering rx_valid signal. The valid signals from channels are considered whose rx_valid is set to 1.

The next section gives you a picture of how data from each RAM is combined.

6.3.2 Clipping of Data

In this section we deal with how the data is combined from all the channels. The simple addition of data and shifting it to get a 16 bit output doesn't increase the probability of sign of the bit from the output of the channel estimation, which is the purpose of using antenna diversity.

To overcome this problem the clipping of the data is considered. The clipping module essentially adds all of the data arriving from the channel estimation module and clips the resultant data based on the following snippet:

```
ab = a[15] ^ b[15];
temp = a + b;
out = ab? (a + b) : (a[15]?(temp[15]?(a + b):(16'h8000)):
      (temp[15]? (16'h7fff) : (a + b)));
```

Hera **a** and **b** are the inputs and output is **out** to the clip module which are of 16 bit width. The MSB's of the bits are XORed and stored as ab. The sum of the both is saved in temp.

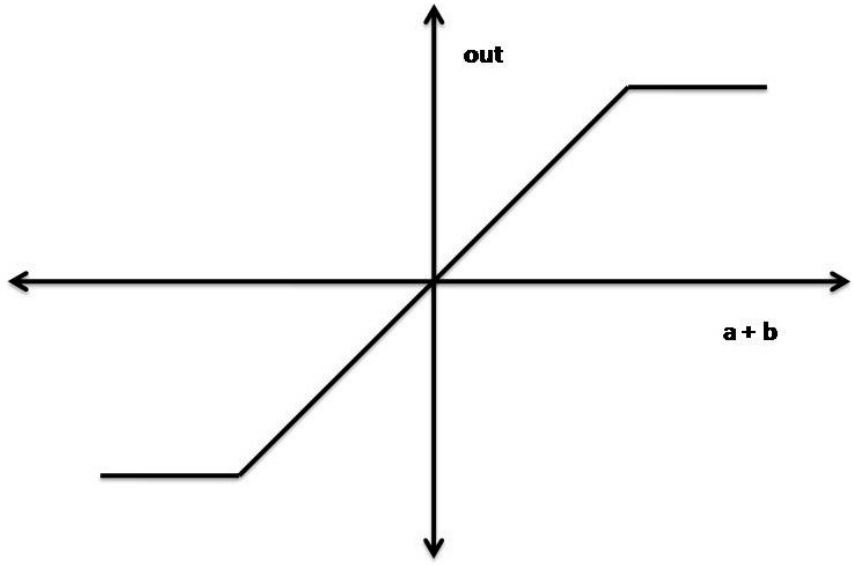

Figure 6.4: Clipping

If ab is 1, then the data coming from both streams have different sign so we simply add them and assign the value to out.

If ab is 0, then the data coming from both streams will have same sign. If MSB of the input is 1 and

1. The MSB of temp is 1 then we simply add the data and assign it to out.
2. The MSB of temp is 0 then it means the data is clipped and we assign the out 16'h8000.

If the MSB of input is 0 and

1. The MSB of temp is 1 then it means the data is clipped and we assign the out 16'h7fff.
2. The MSB of temp is 0 then we simply add the data and assign it to out.

The clip module stated above is used in combining the four inputs to give the output the one going into LDPC stage.

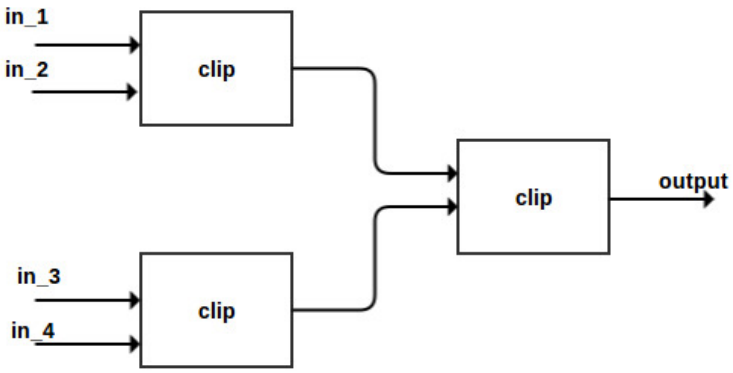

Figure 6.5: Block Diagram for clipping

As shown in the figure above the data from 4 the streams are fed into clipping block. The real and imaginary parts have same architecture. The inputs from 2 streams are given to the clip module thus the output from 2 clipping modules is again fed into other clipping module to obtain the final result. This output is fed into LDPC.

CHAPTER 7

Validation and Improvements

The output of the Verilog code should always be validated with the golden reference (C code) in order to check its credibility. As generating a bit file for this huge project takes a lot of time, we first check the integrity of the system in simulation and then the system is verified on the hardware.

In first section data verification using simulation is discussed and in the next one, we discuss how data is obtained after implementing the design on hardware.

7.1 Simulation

The behavioral simulation was done using **ISIM** software, which is a package provided by Xilinx. This is the environment to check the functionality of the code. The real world conditions like the manner in which data is obtained is duplicated in the test bench which is needed for the simulation.

To start the simulation, the I and Q components are read from two different files by using **readmemh** command. The data that is read from the files is stored into a register value of required length. The data is sent from these registers, one data sample (real and imaginary) per clock cycle.

Certain check points are associated with in the system where the data from the simulated verilog code is checked with the reference code. Since we need to check data for many clock cycles it is not wise to compare each clock cycle data with reference data. Therefore the data that needs to be checked is written into a file by using **fwrite** or **fdisplay** command. The data that is obtained from Verilog and the golden reference C code are verified by writing a **python** script that takes the data from each file and compares it. Thus the data obtained from various check points are compared with corresponding values of C and ensured that they are in agreement.

7.2 Implementation

We have employed three different approaches to get data from the hardware.

1. ILA and ICON
2. VIO and ICON
3. CDC

1. In the first approach, the only data signals that can be viewed on Chipscope are the signals in the module where ILA is instantiated. Therefore all the required signals are made as outputs from the module where it is instantiated. There will be a control signal from ICON to ILA to control the interface with hardware. The data that is to be viewed should present in trigger port. This ILA and ICON will act as an interface between hardware and Chipscope.

2. In the second approach, Chipscope is absent. The data is read and write into a file. There will be a control signal from ICON to VIO to control the interface with hardware. A Tcl script is needed for the interface between the hardware and file read write. This is slightly cumbersome to implement but the advantage is that the inputs can be changed during run-time.

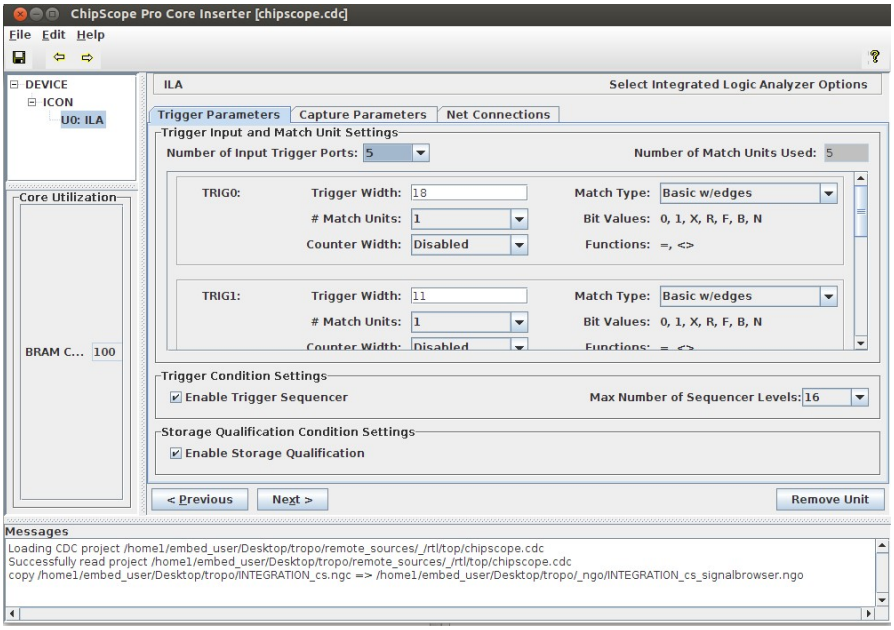

Figure 7.1: Chipscope Definition and Connection File

3. The third approach is the easiest in terms of realization. The CDC (Chipscope Definition and Connection File) core which is instantiated before the logical synthesis

of the design, can be modified only after synthesis since it contains signals which are synthesized. The internal signals can be viewed **only** if the design is synthesised with hierarchy. The signals that are needed to be tracked can be directly selected using a GUI. We can have 16 trigger ports each of width maximum 256 and depth 131072. Different signals can be mapped onto different trigger ports as per users choice. The trigger parameters can be set in the tab corresponding to it. In the capture parameters tab the depth of data is set and finally in the capture parameters tab the signals that need to be captured are set in this tab. Similar to ILA and ICON the signals that are chosen are viewed in Chipscope.

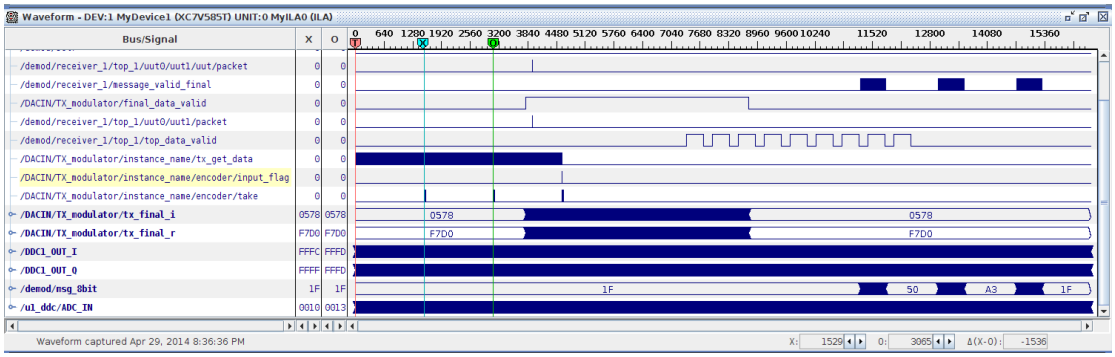

Figure 7.2: Chipscope

The above approaches give you the interface for hardware and Chipscope. So after getting the data into chipscope as shown in the above figure we export the data into a file. Similar to that of the previous method of verification, the data which is fed into a file are checked with reference code using Python.

7.3 Improvements

Initially, after the design was complete, the transmiter and the receiver were tested independently on the FPGA board with success. To simulate the real conditions, it was necessary to test the entire system on board simultaneously. Since there is a single board available the system was tested in a 'looped back' mode i.e., the transmitter and receiver were connected end-to-end and tests were run. The design failed the tests when there was loop-back. On careful observation the Down conversion module that is present in loop back at the receiver end was found to output data values of very low range, which was insufficient for the Schmidl-Cox module to detect a preamble. The

fix was to manually increase their values by considering only relavent shifted bits. This problem was taken care of in the later stages by introducing an AGC (Automatic Gain Controller) so that data would be in required range.

Once SISO design was found to be satisfactory, implementation of MIMO went underway. The MIMO seems to be worked when all the four streams are present but when two of the streams is removed the bits were not decoded in correct fashion. To identify the problem we went through each signal that might be possible for this error and checked with the simulation, but the simulations works fine. After a while it is detected simple addition of data from all the streams and division by 4 will take the range of values that go into LDPC very low and bits couldn't be decoded, after that the clipping module as discussed in the interface between channel estimation and LDPC section was introduced in order to achieve data in the required range irrespective of number of streams that are active.

So after implementing clip module in the system the MIMO was working fine even with only one stream present but there has been a situation when the receiver chains were disconnected and then reconnected. The system was working when all the four streams are present, and even when two of the streams are removed it was doing good, but when all the four streams are reconnected again the decoded message bits were not correct. To identify the problem we have generated bit file multiple times and tested it several times but the problem couldn't be identified. After debugging the signals from start it came clear that the data was not recorded correctly in the RAM that is present in the interface between channel estimation and LDPC. When one of the stream is removed the packet doesn't get detected in that stream so the data is not being written into RAM that is present in this particular stream,hence the write address for this RAM will not be incremented while the write address for RAMs present in other streams are incremented,so when the stream is reconnected the and the frame is detected the write address of all the RAMs present will not have same initial write address value, while the read address will be same for all four streams thus data is being read is wrong Hence to overcome this situation the RAM write address is set to zero after one frame has been fed into it,at the same time the read address from the state machine that is present in the interface will always start with a zero for new frame after getting the valid signal. To achieve above behaviour a counter has been incremented whenever there is a write enable signal for the RAM, once the count value reaches 3456 (384*9) the write address

is set to zero.

7.4 LFSR

The system worked in loop back, on a channel and it was also verified by transmitting over antenna and receiving it. When data was transmitted over Ethernet, the decoded bits were not the same as the message bits sent in. The problem is while transmitting over Ethernet there will be a continuous stream of zeroes padded to the message bits. If there is a continuous stream of zeroes in the message there is a peak at IFFT stage in the transmitter since it considers the stream of zeroes as a dc value. The peak crosses threshold and the system is disturbed.

To overcome this problem we have implemented an **LFSR** (Linear Feedback Shift Register) in the system. LFSR is a shift register whose output bits are a linear function of previous state. There is a shift register present whose initial value is seed. The bits get shifted for each clock cycle. The next input is an XOR of some bits that are present in shift register. The choice of bits that are chosen are determined by considering a coefficients of primitive polynomial which is of order as number of bits in shift register. The output of the shift register can be chosen any set of bits from the register value. The purpose of shift register is that the chance of it giving the output as continuous stream of zeroes is very less and thus mitigating the dc effect.

Since the problem is with the input having a continuous stream of zeroes. The input message bits in transmitter is XORed with LFSR output. In the same fashion the output of the receiver is also XORed with LFSR output. In the transmitter stage we get the message as bytes, there is a valid signal which is associated with this data. The LFSR shifts it register only when this enable is present. Thus LFSR output changes for each clock cycle till the valid is present.The output of LFSR is an 8 bit value of 64 bit register value present. The output of LFSR is now XORed before going into next stage. Thus we have ensured there will be no continuous zeroes in the message for further stages.

The above operation done in the transmitter should be reversed before giving away the output. The LFSR that is used in transmitter should be used here in order to reverse the effect. That is the data Xored with LFSR output in transmitter should also be XORed with same LFSR output value at receiver end in order to get the data back.

There is a small change that needs to be done for LFSR. The message is transmitted via bytes. Thus the first bit entering the transmitter will be LSB of the first byte obtained but at the receiver end when output is packed into bytes it becomes the MSB of first byte. But at the receiver end the output is read from MSB hence the stream of bits will be same at the transmitter or receiver, but when packed into bytes they get reversed,so the output of LFSR present in receiver should be reverse of LFSR output at transmitter,so that the XORing of bits is done correctly.

In the receiver at final stage after packing the LLRs into 8 bit value with a valid signal and feeding it into FIFO, now the data is XORed with LFSR out and fed into FIFO. The LFSR is shifted with respect to valid signal. Hence we receive correct message bits that are transmitted even with a continuous stream of zeroes. Thus the problem with Ethernet was solved.

CHAPTER 8

Parameterization

Parameterizing is essential for any system developed because the new user can use the existing building blocks to build a bigger system, without having to build them from scratch. IP cores should be removed in the project to make sure that the code is compatible with any device.

The following are the signals that are parameterized in the code

1.	Threshold	4.	CORDIC iterations
2.	Correlation Depth	5.	Size of Memory bank
3.	Bit width		

Table 8.1: Parameterization variables

Of these variables only 'threshold' can be changes at run time. The other variables are parametrized in order to use the present modules in a plug-and-play environment in the future. To parametrize variables we need to first parametrize the code since in the IP cores the bit width and number of iterations cannot be changed by changing the parameter and by removing IP cores we can make the code device irrelevant. So,we first discuss how each IP core is parameterized, then the IP cores are removed and replace them with the parameterized code.

The IP cores that are parameterized are

1. Shift Register
2. Accumulator
3. RAM
4. CORDIC in Vector mode
5. CORDIC in Rotation Mode

The Multiplier IP core is just replaced by multiplying two numbers in both normal and complex multiplier.

1. Shift Register : The architecture for shift register should be a continuous registers which are connected with one another.The output of previous register is connected with input of next register. The input of the first register is input to the shift register and output of the last register is output of the shift register. The input bit width and depth of the shift register (number of registers present) can be chosen as per user.

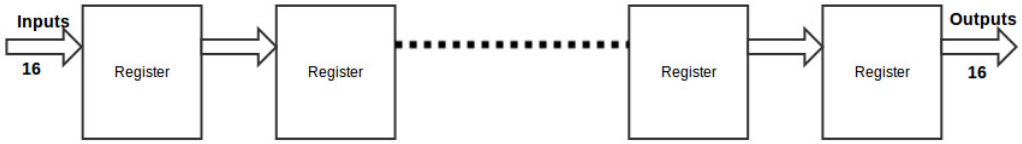

Figure 8.1: Block Diagram for Parameterizing Shift Register

Each register is implemented like a bus of flip-flops, which on receiving a positive edge of clock at one if its inputs, outputs a value equal to its controlling input. We implement read first and write next on series of registers.

Since the depth of Shift registers is high we cannot simply instantiate all manually and connect them, there is command called generate which does this for this.

```
generate
  for(ir=0; ir < shift_value; ir=ir+1)
    begin:shift_r1
     dff dff1(
         .d(dinr[ir]),
         .clk(clk),
         .sclr(sclr),
         .q(dinr[ir+1]));
    end
endgenerate
```

The **dff** is the Register shown in the figure. The shift_value is the depth of the shift register. The generate command generates 'shift_value' number of registers and connects them accordingly thus decreasing the amount of work to be connected. Here just by changing the shift_value we can change the depth of the shift register. Thus the correlation depth can also be changed since the shift register is present for obtaining the values for correlation.

2. Accumulator : The architecture of accumulator is the input value gets added to the previous value thus accumulating the input values. For this we need to have a register to store the previous value and an adder in order to add the previous value and current input.

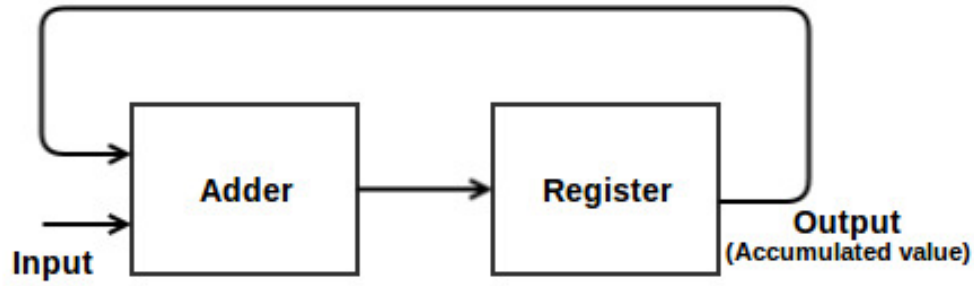

Figure 8.2: Block Diagram for Parameterizing Accumulator

The output at any point time is the accumulated value of inputs till then.

3. RAM : The architecture of RAM is that it should contain a set of registers whose length is the size of RAM and the width of each register is the input bit width. The RAM has clock, reset, write enable, read enable, write address and read address as inputs. The input is fed into register whose location value is equal to write address when write enable is present. The data that is present in the register whose location value is equal to read address is given to output when the read enable is present. Thus reading and writing into RAM is achieved.

The CORDIC is a major part during parameterizing the code. It implements a trigonometric function on the hardware. First we discuss the CORDIC algorithm and how we have managed to get it working. The CORDIC algorithm uses shifting and additions in order to achieve the results. We have employed an LUT (Look Up Table) architecture in CORDIC.

4. CORDIC in Vector Mode : The CORDIC in vector mode is used in calculating the angle from the input sine and cosine values. The cosine corresponds to x-component and sine corresponds to y-component. There is an angle register that accumulates the rotation angle corresponding to each iteration. The angle corresponding to each iteration obtained from LUT. The number of iterations is fixed by user. The angle of rotation depends on iteration number. The bit width of inputs and output is 32.

The initial vector can be obtained from initial cosine and sine values and the initial angle is set to zero. The vector should always be in first quadrant,if the vector is not in first quadrant the vector is moved to first quadrant and the angle rotated is recorded and suitable rotation is done with respect to this angle after obtaining the angle from CORDIC iterations. The initial x and y components should be divided by 1.647 for normalization before it gets fed into CORDIC module. The angle of rotation corresponding to each iteration is present in table from which it is read. The angles are $\tan^{-1}(2^0)$,$\tan^{-1}(2^{-1})$,$\tan^{-1}(2^{-2})$,$\tan^{-1}(2^{-3})$ respectively corresponding to each iteration. Depending on iteration number and corresponding angle to be rotated is chosen.

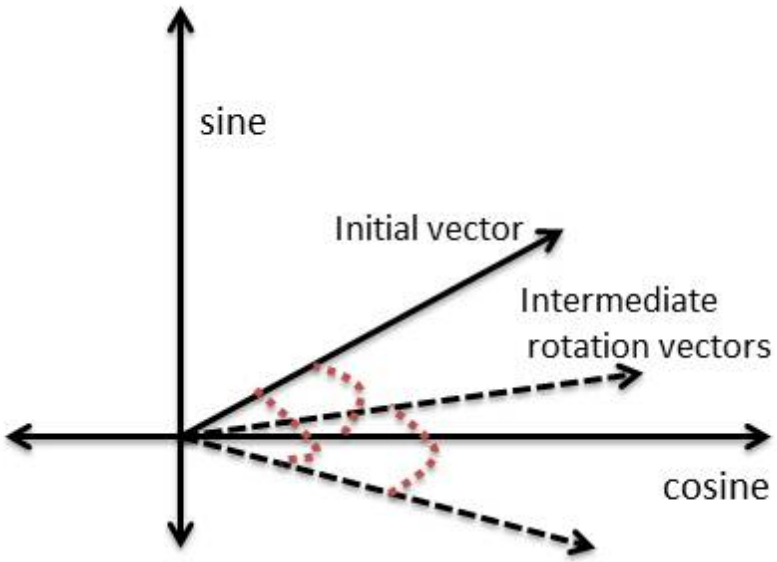

Figure 8.3: CORDIC

The aim of the iteration is to make the y-component close enough to zero on each iteration. This done by rotating the vector. If the vector has a positive angle then the vector is rotated clockwise with angle corresponding to iteration number. If the vector has a negative then the vector is rotated anti-clockwise with the angle corresponding to iteration number. Thus at each step the vector will be close enough to zero. The angle register which is present accumulates the angle corresponding to each iteration and direction of rotation. If the vector is rotated clockwise then the angle corresponding to the iteration is subtracted while the angle is added if the direction is ant-clockwise.

The angles that are chosen for rotation makes sure that the resultant vector obtained after rotating can be merely obtained by shifting and addition.

$$X_{out} = X_{in} * \cos\theta - Y_{in} * \sin\theta$$
$$Y_{out} = X_{in} * \sin\theta + Y_{in} * \cos\theta$$

The above resultant vector boils down to simple shifting with respective to the given rotation angles. if the rotation is in clockwise

$$X_{out} = X_{in} - Y_{in} \gg i$$
$$Y_{out} = Y_{in} + X_{in} \gg i$$

If the rotation is in anti-clockwise

$$X_{out} = X_{in} + Y_{in} \gg i$$
$$Y_{out} = Y_{in} - X_{in} \gg i$$
$$i = \text{CORDIC iteration}$$

The basic module that is present in the CORDIC should take care of this operation. Below shown is the functional block that needs to be implemented in hardware.

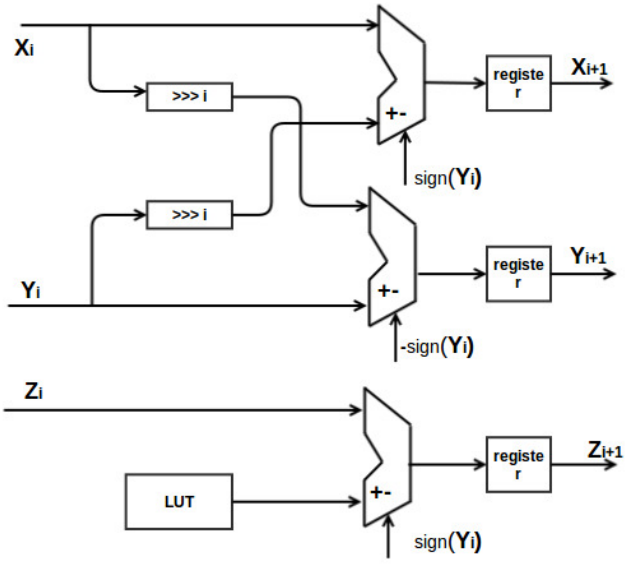

Figure 8.4: CORDIC Functional Diagram in Victor Mode

```
X_shr = X[i] >>> i;
Y_shr = Y[i] >>> i;
X[i+1] <= Y_sign ? X[i] - Y_shr          : X[i] + Y_shr;
Y[i+1] <= Y_sign ? Y[i] + X_shr          : Y[i] - X_shr;
Z[i+1] <= Y_sign ? Z[i] - atan_table[i] : Z[i] + atan_table[i];
```

The Y_sign tells us whether the vector is in first quadrant or fourth quadrant. If it is in first quadrant then the vector is rotated clockwise and if in fourth quadrant it rotates in anti-clockwise direction. The z component gives you the angle.

We have made a pipe-lined CORDIC for this purpose. So the block which has been shown in the above figure is instantiated multiple times. Similar to the shift register generate command is used here in for multiple instantiations.

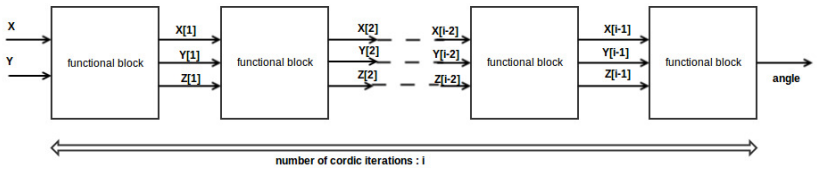

Figure 8.5: CORDIC Block Diagram in Vector mode

Thus after going through multiple stages as chosen by user the z component is the angle value for the input sine and cosine components. If the initial vector is not in first quadrant the angle is rotated sufficiently to compensate the initial vector.

5. CORDIC in Rotation Mode : As in vector mode the basic functionality of CORDIC remains for the rotation mode. There will be rotations corresponding to iterations. The angle of rotation for iterations is same as in vector mode. The input for rotation mode is the angle and outputs are sine and cosine corresponding to x and y components but the main difference is that in vector mode the initial vector will be rotated to make it close enough to zero for each iteration. In rotation mode the initial angle is subtracted with rotation angle corresponding to CORDIC iterations to make it close enough to zero for each iteration. The initial vector will have a x-component of $\frac{1}{1.647}$ and the y-component is 0. The bit width of input and outputs is 32.

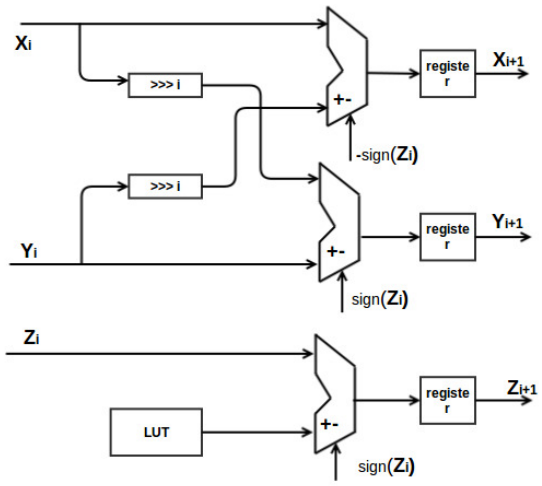

Figure 8.6: CORDIC Functional Diagram for Rotational Mode

Thus for each iteration the angle value is checked,if the angle is positive then vector is rotated in clock wise direction and the corresponding iteration angle is subtracted from the input angle,if the angle is negative then the vector is rotated in anti-clockwise direction and the corresponding iteration angle is added to the input angle. Thus in this case the sign of the angle is checked and the rotation is done accordingly. The functionality implemented above is

```
X_shr = X[i] >>> i;
Y_shr = Y[i] >>> i;
X[i+1] <= Z_sign ? X[i] + Y_shr          : X[i] - Y_shr;
Y[i+1] <= Z_sign ? Y[i] - X_shr          : Y[i] + X_shr;
Z[i+1] <= Z_sign ? Z[i] + atan_table[i] : Z[i] - atan_table[i];
```

We have made a pipe-lined CORDIC for this purpose. So the block which has been shown in the above figure is instantiated multiple times. Similar to the shift register generate command is used here in for multiple instantiations. ROM is replaced by look up table here. The number of iterations depends on number of instantiations.

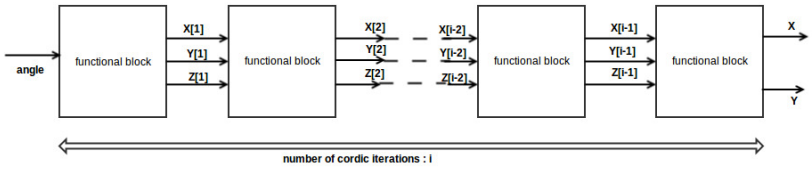

Figure 8.7: CORDIC Block Diagram in Rotation mode

The X and Y components are cosine and sine components respectively which are the outputs of system.

Thus we were able to parametrize all the IP cores present hence parametrizing the data path. Since the shifting of the control signal in the control block depends upon CORDIC iterations. The control block is also parametrized by controlling the iteration value.

CHAPTER 9

SCFDMA

The next part of the project is implementing a similar kind of project. In the previous system we have used multiple orthogonal frequencies(OFDM) to transmit to increase the data rate but this system will have a high Peak to Average Power Ratio which is not favourable in the up-link part of the communication. To overcome this in the new system we have used a single carrier hence it is called Single Carrier Frequency Division Multiple Access (**SCFDMA**). The implementation of SCFDMA has a slight difference with that of the OFDM. Lets first deal with the Specification of system.

9.1 Specification of System

Similar to the OFDM ,the SCFDMA will be transmitting the data via frames. The message consists of In-phase and Quadrature components.

Each frame consists of a preamble,channel estimation pilots with cyclic prefix and 3 symbols with cyclic prefix. The preamble is of length 64 and its cyclic prefix of length 10, which is optional. The channel estimation pilots is 128 and cyclic prefix of it is 32. Each symbol consists of 512 data and a 32 cyclic prefix.

Hence each frame consists of 64+32+128+(512+32)*3 = 1866

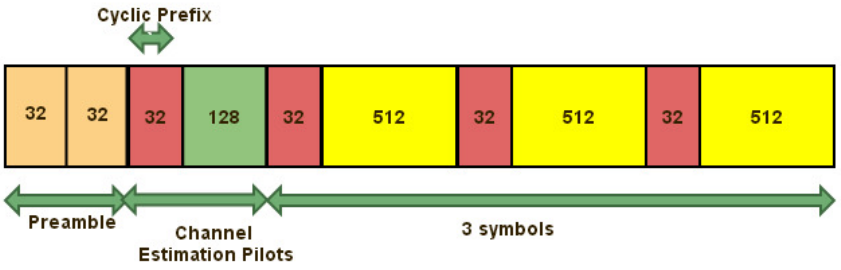

Figure 9.1: Frame structure in SCFDMA

Each symbol consists of 32 pilots of the 512 data signals. Hence the number of

actual data signals corresponding to each component is 1440(480*3). Hence the total actual data signals corresponds to 2880 corresponding to both components.

The number of message bits corresponding to frame is 1536, since the code rate of the system is $\frac{2}{3}$. The number of message signal after encoding is 2304. After QPSK modulation these 2304 message signals translate to 1152 Inphase and Quadrant phase components. This is same as number of message signals corresponding to one LDPC burst present in OFDM. This is maintained in order to use the same encoding block that is present in OFDM.

Since each frame in SCFDMA consists of 1440 In-phase and Quadrant phase components of data. The Inphase and Quadrant phase corresponds to message is 1152, Therefore the rest of 288 junk message signals should be added to the actual message signals.

9.2 Difference between OFDM and SCFMDA

There is an intermediate stage present between encoding and pilot & null which intakes 1152 encoded data and outputs 1440 by adding junk data at the end in order to use the same encoding stage as OFDM.

Since the OFDM consists of multiple frequencies which are orthogonal to each other there is an IFFT IP core which translates the data onto orthogonal frequencies for transmitting and adding a cyclic prefix but SCFDMA has only a single carrier hence no IFFT stage is required and it is replaced by a stage which adds cyclic prefix.

The hand shaking between stages is improved compared to OFDM. The hand shake signal generally should be used to communication between two adjacent stages. But as mentioned in overview of transmitter section of OFDM, there is a handshake start signal from encoding stage to the IFFT stage which is not the adjacent stage to the encoding stage. Once the IFFT stage receives a start signal then it gives ready signal to the pilot and null stage(adjacent stage of encoding). Once the pilot and null stage receives a ready signal from IFFT stage then it gives a ready flag to the adjacent stage. Thus the encoding stage receives a ready signal from the next stage(pilot and null) due to the start signal which it has sent to IFFT stage, this type of handshaking is not good

and it is replaced in SCFDMA by clear handshaking between adjacent stages.

9.3 Overview of Transmitter in SCFDMA

We first give the overview of handshake present between two adjacent stages.

For the present stage there will be a send_in coming from the previous stage stating that it is ready to give data. If the present stage is free then it sends a take_out signal to the previous stage to get the data from it.

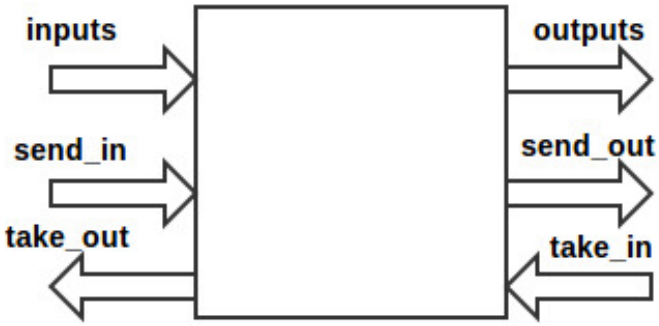

Figure 9.2: Hand-shake between adjacent stages in transmitter

The take_out is given only after it receives a send_in from the previous stage. Once the present stage has data ready to transmit to the next stage then it gives send_out signal to the next stage that data is ready for it to be transmitting, when it receives take_in from the next stage then it transmits the data to the next stage. Thus a harmony is achieved at both transmit and receive ends at each stage. This is other way of managing the control signals between each stages the other being using a centralised controller which raises the control signal for each stage.

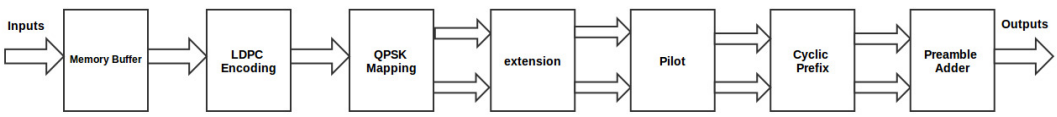

Figure 9.3: Transmitter overview in SCFDMA

The transmitter data path in SCFDMA will be same as in OFDM. There is a buffer initially followed by encoding and map-buffer stage, the handshake between these two

stages is same as the one present in OFDM. The QPSK modulation stage which is present in encoding stage is also retained. Now we will have the QPSK modulated input message data.

After this an extension stage was introduced that takes the input QPSK modulated 1152 symbols and gives output 1440 symbols. The handshake above is maintained between the encoding and extension stage. As soon as it receives send_in from the encoding stage it gives the take_out and in the next cycle it gets the data. Once it gets the data in the next cycle it raises send_out to the next stage signalling the data is ready when it gets take_in from the next stage it transmits data.

After this there is pilot stage which is same as pilot and null of the OFDM, but there are no nulls in OFDM. The handshake is maintained with the previous and next stage. This stage adds the pilot value to the corresponding pilot locations and gives the data.

The next stage being cyclic prefix stage which is replacement to IFFT in OFDM, since IFFT takes care of adding cyclic prefix to each symbol, as IFFT is not present the present module should add the cyclic prefix. As soon as the cyclic prefix is added to the first symbol it raises a flag to the preamble adder which adds the preamble to the input data and sends the data.

9.4 Implementation of Transmitter in SCFDMA

The handshake signals are maintained as described in the previous section In this section we deal with the implementation of individual stages. We have retained the LDPC encoding stage which is present in OFDM for SCFDMA. In the stages we have designed there are two state machines present so that one is used in commuicating with inputs and other for outputs with some inter communication between them.

The encoding stage will encode the incoming data and gives out the 2 bits at a time to QPSk stage thus giving the In-phase and Quadrant phase components. There will be 1152 message symbols output at this point. The new extension stage will be converting the 1152 symbols to 1440 message symbols.

9.4.1 Extension Module

This stage contains two state machines, one for taking the input from previous stage and other for giving the data to next stage. There will be inter communication between these twostate machines. The input to this stage will be a continuous stream of 1152 symbols so once we get the input it is stored into registers.

Inputs	Outputs
QPSK mapped signals of length 1152	QPSK mapped signals of length 1440
in_valid(send_in)	take_a(take_out)
ready_b(take_in)	out_ready(send_out)

Table 9.1: I/O to Extension Module in SCFDMA

This state machine is used for inputs. When the send_in signal from previous stage(encoding) to the present stage is raised which is in initial state S0, goes to S1 state and the extension stage gives take_out flag to the previous stage signalling it to give the data in the next cycle, then from the next cycle we will get data from the previous stage.

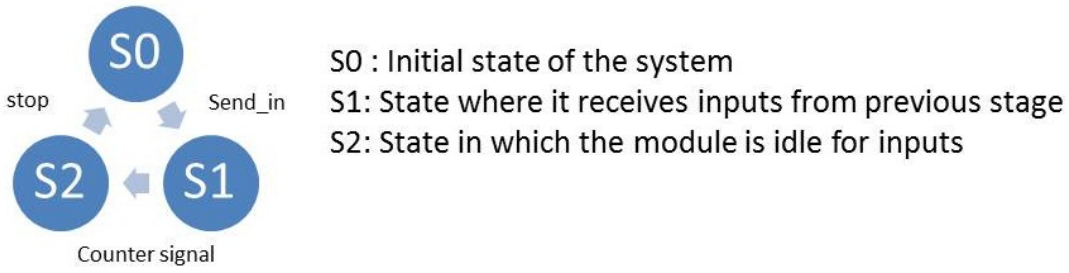

Figure 9.4: Control Block of Inputs of extension module in SCFDMA

After giving take_out signal to the previous stage then we will receive a continuous stream of 1152 QPSk symbols. The next cycle after the present stage gets data it raises a send_out signal to the next stage, which in turn gives a take_in flag whenever it is free. The system is presently in S1 stage, there is an internal counter which counts till 1152 once it reaches that value the system goes into next state S2 where the take_out signal to previous stage is lowered and goes into S2 stage. The system goes to the initial state from S2 state once it gets a stop flag from the other state machine.

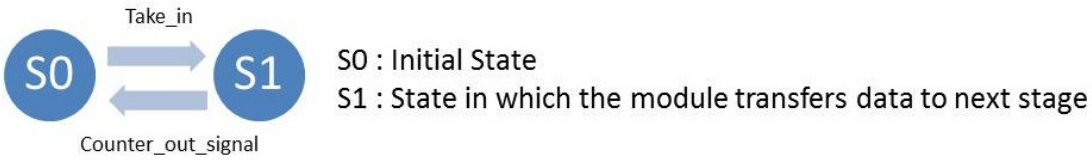

Figure 9.5: Control Block of Outputs of extension module in SCFDMA

This state machine is for outputs. The system is initially in state S0, once it gets a take_in flag from next stge it goes to S1 state. When the system is in state S1 it outputs the data to the next stage. The data which is fed into registers from the inputs are used to give the outputs. There is an internal counter present in state S1 which increments on each clock cycle, the data is given from registers to output till the counter value reaches 1152, after that a constant QPSK symbol is added to the outputs till the counter values reaches 1440. Once the counter value reaches 1440 the state machine goes into initial state S0 and raises a stop flag to the other state machine resetting it to the initial state.

The next stage will be adding the pilots to the the present data forming a symbol of length 512.

9.4.2 Pilot stage

There are some modifications made to the pilot and null stage present in the OFDM to make it work here. In OFDM there is a single state machine for receiving input and transmitting data but in SCFDMA there are two state machine one fore receiving the data and other for transmitting data

Inputs	Outputs
QPSK mapped signals of length 1440	Message signals of complete symbol
in_valid(send_in)	input_ready(take_out)
ifft_valid(take_in)	out_ready(send_out)
clk,reset	

Table 9.2: I/O to Pilot in SCFDMA

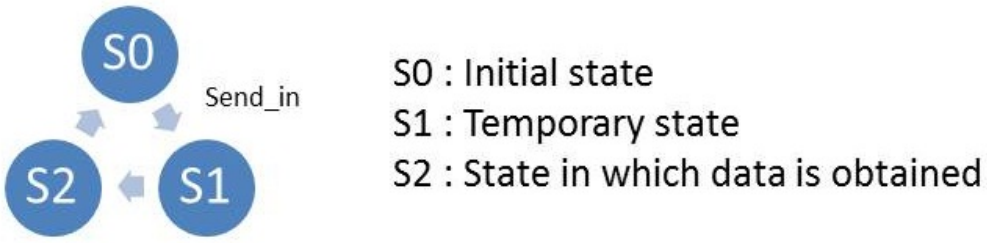

Figure 9.6: Control Block of Inputs of pilot module in SCFDMA

This state machine is for inputs. Initial the system is in state S0, once it gets the send_in flag from previous stage then it goes to a temporary state S1 where it sends a take_out signal to the previous stage and goes into S2 state and raising a flag send_out to the next stage which in turn responds with a take_in flag whenever it is free. Thus the system is S2 state will have inputs which are fed into registers. The inputs are valid when only there is a valid signal is associated with it. When the present stage gives a take_out signal to the previuos stage there is a continuous 1440 data as input to it. Thus we will have a valid signal for 1440 cycles. Once the valid goes low the system goes into intial state S0.

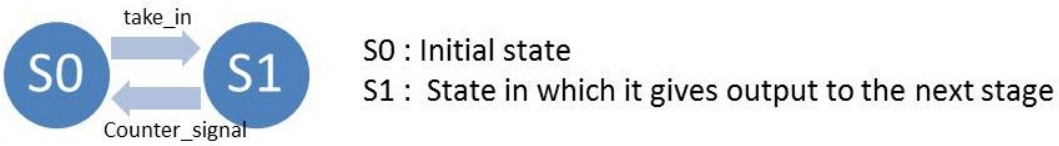

Figure 9.7: Control Block of Outputs of Pilots module in SCFDMA

This state machine is used for outputs. The system is initially in state S0,once it receives a take_in flag from next stage after it sending a send_out flag to next stage the present stage goes into S1 state. In S1 state it outputs the data to the next stage whenever there is a takein signal from the next stage. The outputs are obtained from registers which are fed by inputs. The data from registers are continuously transmitted. An internal counter is maintained if the counter value matches to the pilot location value then the pilot data is transmitted or else the consecutive data is transmitted. Thus the counter runs till 512 and sets to zero for three times owing to the three symbols present in the frame. Thus the 480 data signals corresponding to each symbol is added with 32 pilots to get a symbol of length of 512. This is repeated for three times and the system goes to initial state S0. We get the take_in flag from the next stage in three bursts with

a gap of 32 between each burst. The register size(480) corresponds to length of symbol - number of pilots.

Thus till now we have three symbols each of length 512 corresponding to data. The next stage will be adding cyclic prefix to each symbol.

9.4.3 Cyclic Prefix

In this stage we will have only one state machine since as soon as the data is ready to be transmitted it raises a valid signal to the next stage and sends the data, there is no take_in signal from next stage to the preamble adder stage.

Inputs	Outputs
Message signals of complete symbol	Message signals of complete symbol with cyclic prefix
start(send_in)	rfd(take_out)
clk,reset	done(send_out)

Table 9.3: I/O to Cyclic Prefix in SCFDMA

The basic principle involved is the present stage first gives a take_out signal to the previous stage which will be high for 512 cycles thus we get the data corresponding to one symbol. Once it gets a complete symbol the last 32 data signals is added at first to the present symbol and then transmitted thus putting a cyclic prefic before the symbol. This is done thrice corresponding to the three symbols present in a frame.

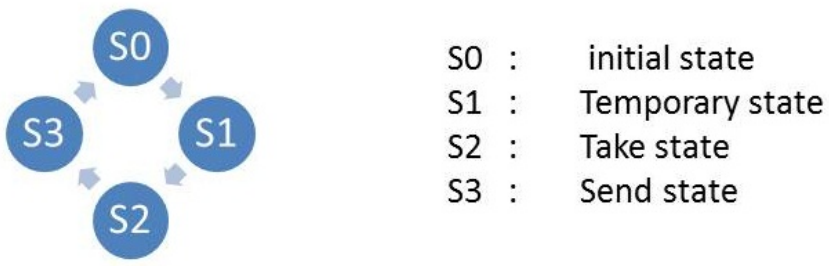

Figure 9.8: Control Block of Cyclic Prefix module in SCFDMA

The S2 state in that state machine is of dual nature i.e if the system is in S2 state it takes data from the previous stage and also gives the data to the next stage.

The system is initially in S0 state when the send_in flag is raised from the previous

stage then it goes into state S1 by raising a take_out flag. In the next cycle it goes to S2 state from S1 state, The system will be in S1 state only for 1 cycle. When the system is in state S2 for the first time it gives take_out flag high for 512 cycles thus obtaining the data corresponding to one symbol which are fed into registers and there is a counter which increments each cycle. Thus the system raises a send_out signal to the next stage and goes to S3 state once the counter reaches 512.

When the system is in state S3 it transmits the data from the register since we need to add the cyclic prefix at first, the data that is present between register address 480 and 512 are transmitted first. Once the cyclic prefix is transmitted it goes to state S2. Now in state S2 it sends the data from the registers with address 0 to 512 in the next 512 cycles and it also obtains the data from the previous stage by raising take_out flag to the previous stage. Thus in these 512 cycles we transmit the data of the first symbol to the next stage and feeds the data corresponding to the second symbol in the registers. After these 512 cycles the systen goes into state S3 to transmit the cyclic prefix corresponding to the second symbol. The same thing happens with the the three symbols. Once the three symbols are transmitted then the system goes into initial state S0. The register size corresponds to length of the symbol(512)

Thus the output from this stage will be a continuous stream of data corresponding to 3 symbols and their cyclic prefix.

9.4.4 Preamble Adder

In this stage the preamble and the pilots corresponding to channel estimation and their cyclic prefix is added to the three symbols that are obtained.

Inputs	Outputs
Message signals of complete symbol with cyclic prefix	Complete frame
start_transmit(send_in)	vd_out(send_out
clk,reset	complete_frame

Table 9.4: I/O to Preamble Adder in SCFDMA

When the cyclic prefix stage sends the send_out signal, from the next cycle the data is present at its input. The input is fed into the registers. Once the send_out signal

from previous stage is obtained the preamble, channel estimation pilots and their cyclic prefix which are present in file are read and transmitted. Once they are transmitted the data stored in registers is sent continuously. The register size (234) should be of length corresponding to length of preamble, channel estimation pilots and their cyclic prefix.

Thus we have obtained a complete frame for transmission which comprises of preamble, its cyclic prefix, channel estimation pilots, its cyclic prefix, 3 SCFDMA symbols and their cyclic prefix.

9.5 Overview of Receiver in SCFDMA

SCFDMA is used mainly in the uplink because of the PAPR problem. Also, the transmitter is relatively easier to implement because of the lack of FFT. The extra computation which is avoided in the transmitter is present in the receiver, which largely prohibits the use of SCFDMA mode in reception. The computation involved in decoding the message in the receiver is explained below.

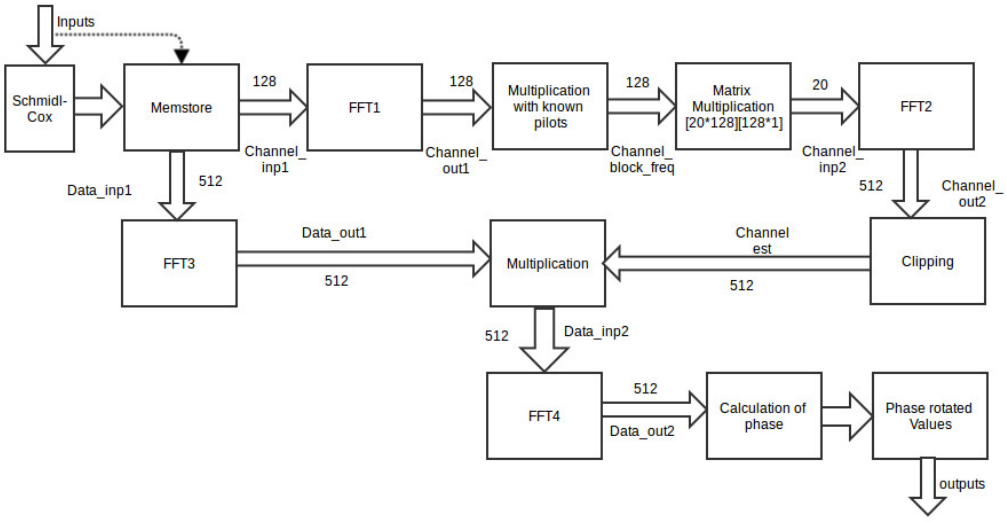

Figure 9.9: Block diagram of the receiver of SCFDMA

The block diagram of the receiver design used in SCFDMA is shown above. It differs from that of its OFDM counterpart in many ways because of the structure of the frame. The channel estimation pilots are not embedded in data. In the frame structure, they are present after the preamble. Since the SCFDMA symbols are not accessed or modified during transforms being applied on the channel estimation pilots, it is made

imperative to add large buffers in 'memstore' module (which is explained in the fifth chapter). The entire receiver design flow is as follows.

The receiver consists of four main stages:

1. **Time synchronization:** Schmidl-Cox algorithm is used to detect preamble and locate the start of channel estimation pilots.
2. **Channel estimation:** Channel estimation block used in SCFDMA is similar to its counterpart used in OFDM. It obtains the channel distortion values that are to multiplied with the data symbols, which have gone through an FFT stage.
3. **Data rotation:** The data samples, after being multiplied with the channel estimation values, go through another FFT stage, and then are rotated with an angle got from the inserted pilots.

It is to be noted that though the data flow in the receiver is correct, the output data at each stage is **not** shifted by any amount to match the C code, as we are not given fixed point code at the time of implementation.

9.6 Implementation details of the receiver in SCFDMA

As already mentioned, implementation of the receiver in SCFDMA is more computationally intensive than OFDM's. But, it was actually coded with more ease because of the reproducibility of the modules written in OFDM. Most of the modules were reused with major changes made to a few state machines. The modules which are reused almost verbatim are not explained in much detail here while newer modules are detailed much more. The modules used in the receiver as follows.

9.6.1 Time synchronization

The method used to detect preamble in SCFDMA is Schmidl-Cox algorithm. It is the same as that of used in OFDM receiver except that parameters like **threshold** and **safety** are different. Also, there is no frequency synchronization involved in the algorithm, at least in the first steps of decoding the symbols. Hence, CORDIC is not used as a part of the algorithm to derotate SCFDMA symbols.

9.6.2 Make_FFT

This module is used in computing FFT of data of a given length. It includes the use of an FFT IP core in which the length of the FFT to be performed is mentioned. This module has clock, reset, start, in_re and in_im as inputs while the output signals are ready, out_re and out_im. Here, 'start' is the flag that stays high for a clock cycle and the I and Q components of the input data arrive consecutively for a time period equivalent to the length of FFT. Thereafter, the IP core does the FFT computation, a 'ready' flag is raised for a cycle which is immediately followed by the output I and Q data for time period equivalent to the FFT length. The figure below shows the timing diagram.

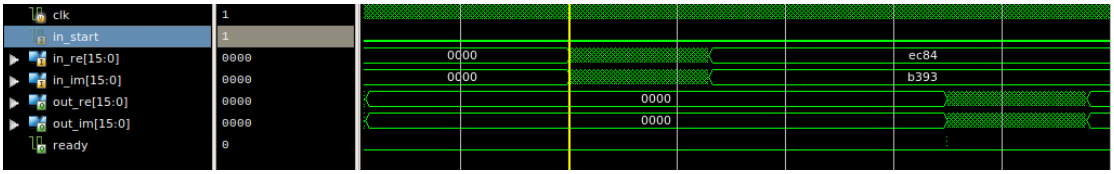

Figure 9.10: Timing diagram of FFT computation

9.6.3 Memstore

This module contains buffers larger than those present in the Memstore module in the receiver of OFDM. Received I Q symbols are continuously streamed into these buffers. Unlike its OFDM counterpart, this module releases data in two different bursts. Once the preamble is detected, these buffers should send the next 128 symbols (which is the length of channel estimation pilots), after the skipping immediate cyclic prefix of length 32. Then, it should wait for another interrupt from the top module, which makes it performs FFT on the SCFDMA data symbols and sends out outputs. It has an FSM which takes care of this process, which is explained below.

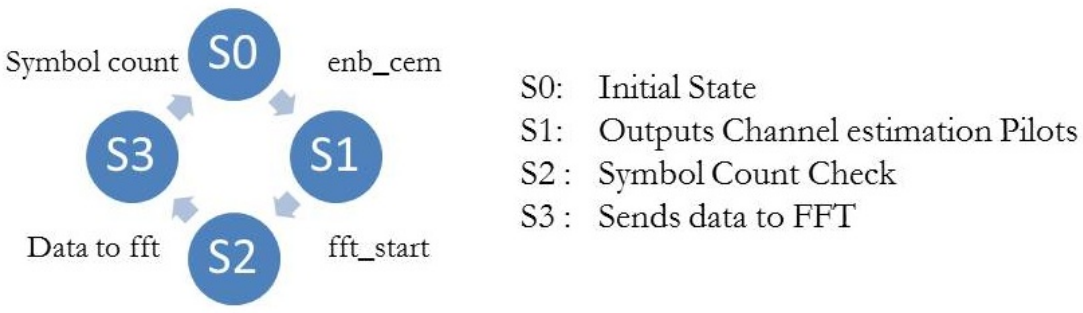

Figure 9.11: Memstore module in SCFDMA

The Mealy state machine has 4 states. Initially, the FSM is in 'st_init', where it waits for an enable from the top_fsm module signalling the detection of a preamble and the finding of the location of the first packet. After it receives the first enable from the top module, the FSM shifts to the next state 'st_send_cem' where it sends channel estimation pilots for a period of 128 cycles (which is the channel estimation length) to the subsequent module. The state machine moves into the next state 'st_send_fft' when the second flag 'fft_start' becomes high for a cycle. Then, each of the 4 SCFDMA symbols is sent to a 'make_fft' module (whose FFT length is 512), whose outputs are sent out in the next state (st_send) while returning to st_send_fft after each symbol is done being transmitted to remove cyclic prefix.

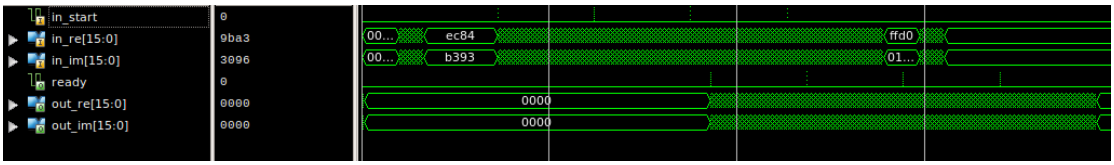

Figure 9.12: Outputs of memstore module in SCFDMA

9.6.4 Channel estimation

Channel estimation in SCFDMA receiver is almost the same as OFDM's. The only exception is the length of the channel estimation pilots is 128 instead of 46 per symbol in OFDM. Also, the preliminary step of pilot extraction does not exist in SCFDMA because only the channel estimation pilots are sent to the channel estimation block. The following lines explain what each of the sub-module does.

It is to be noted that each module inside the channel estimation block follow a single handshake on both of their ports. The manner in which the interface works is already explained in the module make_FFT. A 'start' flag rises at the input for a cycle followed by a burst of data of known length. After the computation is done, the module raises a 'ready' flag followed by a burst of the output data of known length.

The first stage in channel estimation is an FFT of the channel estimation pilots. It makes use of a make_FFT module with length of FFT set as **128**.

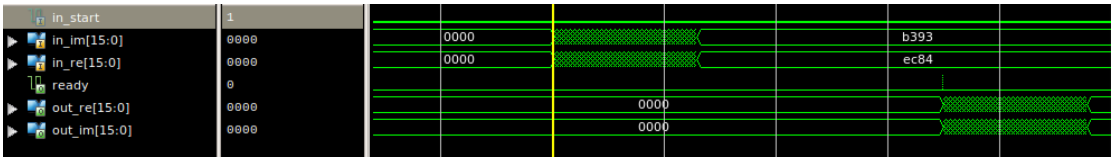

Figure 9.13: Simulation of a 128-point FFT

Then, there is a multiplication of the output of FFT (of length 128) with the known pilot sequence, which outputs a vector of length 128 values.

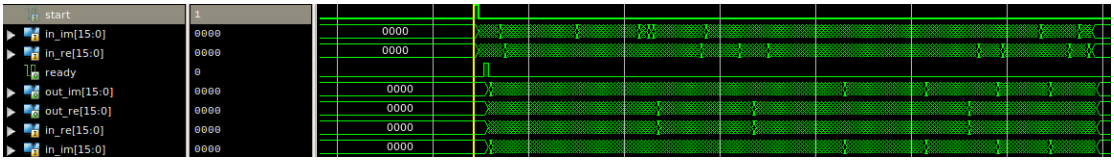

Figure 9.14: Vector - vector multiplication

After the vector-vector multiplication, there is a vector-matrix multiplication of the above vector with the Channel Estimation Matrix (CEM). The number of rows in the CEM is 128, each of whose values are to be multiplied with each of the value of the vector. Thu number of columns of the CEM is **20** which implies that the **channel length** is 20. The output of the vector-matrix multiplication is these accumulated 20 values padded by 492 zeros which make up inputs for a 512 point FFT.

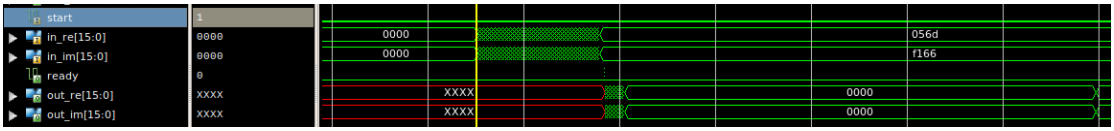

Figure 9.15: Vector - matrix multiplication

The penultimate stage of the channel estimation is a 512 point FFT which takes the 20 channel values padded by zeros to fill up 512 values and outputs a vector of 512 values.

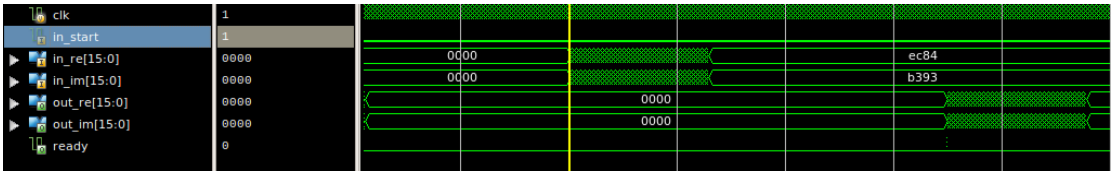

Figure 9.16: The 512 point FFT in channel estimation

9.6.5 Clipping

This is the final stage in channel estimation, where the outputs of the FFT are clipped to 0 if they are below a threshold. This is very easy to implement in software, however, it is considerably tough in hardware. The pseudo-code snippet is as follows :

```
for ( k=0; k<SCFDMA_LENGTH; k++ )
{
    if ( abs ( Channel_est[k] ) < 1/FDE_CLIP_VALUE )
        Channel_est[k] = 0 + j0 ;
    else
        Channel_est[k] = ( Channel_est[k] )
        / ( sqrt(32) * abs(Channel_est[k]) * abs(Channel_est[k]) ) ;
}
```

This includes finding the absolute values of the FFT outputs, finding their square root and the division of each value by shifted version of its absolute value. Since it involves complex transforms, the pseudo-code has been modified for the ease of implementation.

```
for ( k=0; k<SCFDMA_LENGTH; k++ )
{
    p = 1 / (sqrt(32) * abs(Channel_est[k]) * abs(Channel_est[k]));
    if ( p >= ( sqrt(32) * FDE_CLIP_VALUE * FDE_CLIP_VALUE ) )
        Channel_est[k] = 0 + j0 ;
    else
        Channel_est[k] = Channel_est[k] * p ;
}
```

This piece of code does away with any implementation of square root. But, the problem that persists is the division is implemented using an IP core. The latency of this IP core is 20. Hence, the data needs to be buffered for so many cycles before it can be multiplied with **p** (from the code snippet above). Shift registers with depth of 'divide_latency' have been used.

It is to be noted that Verilog does divide operation using '/' operand without any latency. For instance, the following code snippet is perfectly fine in Verilog.

```
c = a/b ;
```

However, the correctness and synthesizability of the code needs to be verified.

9.6.6 Channel_correction

Channel correction involves three main modules which are

1. Multiplication of all 4 SCFDMA symbols which were passed through FFT with the clipped channel estimation values.
2. FFT of the multiplied values.
3. Phase derotation of data

The arrival of the FFT of first data symbol and the clipped channel estimation value have been synchronized thanks to the manipulation of enable signals in the top module. But, before the multiplication begins the FFT values need to be buffered as the 4 FFT bursts need to be multiplied with the same 512-length vector. Hence, a BRAM of length 512 (for both I and Q components) has been implemented which is sufficient. An FSM has been written to send the FFT data into the BRAMs. Also, another FSM is written to minimize latency as well as to prohibit loss of data and to multiply these values accordingly.

After the multiplication of data with channel estimation values, the products are sent to an FFT of length 512, whose outputs aare sent to the next module to derotate data using pilots inserted in them.

9.6.7 Phase derotation of data

Once the SCFDMA symbols and channel estimation values are multiplied, the data values are now to be rotated using pilots that are inserted in them. This is not computationally intensive but the amount of synchronization overhead makes it seem difficult to implement.

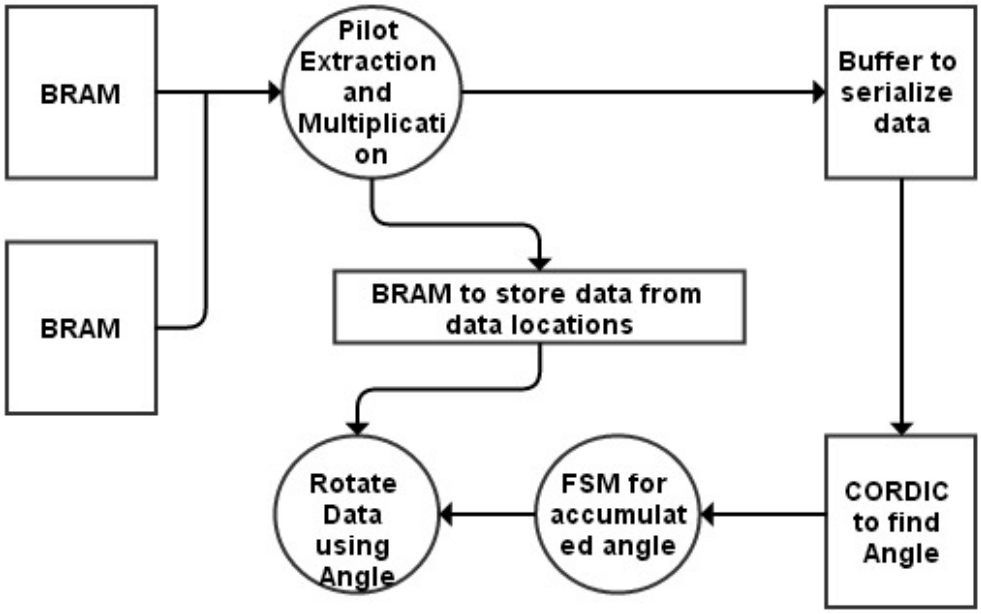

Figure 9.17: Block diagram for data derotation

The first step of the process is to extract received pilots from the respective locations. Since we need to keep count of the 'place' of the symbol value in the frame structure, it would be easier to write the values into a memory block and then read them out again. An FSM has been written to write data to the buffers which are of length 512 (an SCFDMA symbol length), and another to wait for a few cycles to avoid memory conflicts.

Now, pilots need to be extracted. The locations of pilots in all of the SCFDMA symbols are same and they are 32 in number per symbol. The pilot locations are read from a file and so is the known pilot sequence. An FSM is implemented to extract pilots from their locations and also to write the data into another buffer of length (4*480). This is so because there are some transforms applied on these pilots and there is a need to save the useful data of the SCFDMA symbols.

Note: The fact that the pilots are regularly placed is **not** utilized in the code because the location of pilots might change in the future which makes the current code quite generic.

The extracted pilots are multiplied with the known pilot sequence in the above mentioned FSM. Now, an angle has to be extracted from these values which are accumulated over all transformed pilot values in a symbol. Then, the data values of that symbol have

to be derotated using the angle.

For this, there is a need to change these 32 values into a burst where there is data in each consecutive clock cycle. Hence, a module has been written to take these 32 values as they come and then send them out as a burst.

As a burst of 32 transformed pilot values are available, they are sent in serially into a CORDIC module in rotatory mode and the angle by which the data in SCFDMA symbol is to be rotated is accumulated in an FSM.

Then, another CORDIC module is instantiated which rotates the data values with that angle. The above two state machines run for all of SCFDMA symbols as pilots of each symbol are transformed differently, which leads to different angle for each symbol and hence different derotation.

CHAPTER 10

Conclusions

OFDM transceiver is successfully implemented on the hardware. It encapsulates a reliable method to decode ethernet packets for the transmitter to send out messages and to transmit back decoded messages through ethernet. More importantly, the code has been parameterized to the extent that was possible so as to make the code device independent and to use the modules in a plug-and-play environment. As an instance, an SCFDMA transceiver is built using most of the blocks present in the original OFDM code. The data and control flow of receiver is correctly realized but due to lack of fixed point golden reference, the correctness of outputs at each stage are not verified.

REFERENCES

[1] H. Zou, B. McNair and B. Daneshrad, “An integrated OFDM receiver for high-speed mobile data communications,“ Global Telecommunications Conference, IEEE, 2001, pp. 3090-3094.

[2] T. M. Schmidl and D. C. Cox, “Robust Frequency and Timing Estimation for OFDM,“ IEEE Transactions on Communications, Vol. 45, No. 12, 1997, pp. 1613-1621.