HARDWARE IMPLEMENTATION OF RPW MODEM USING ZC702 FPGA PLATFORM AND AD9361 RF TRANSCEIVER

A Project Report

submitted by

AKHILESH BHARATHA

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 2018

THESIS CERTIFICATE

This is to certify that the thesis titled **HARDWARE IMPLEMENTATION OF RPW MODEM USING ZC702 FPGA PLATFORM AND AD9361 RF TRANSCEIVER**, submitted by **AKHILESH BHARATHA**,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.

Prof.Devendra Jalihal	**Prof.Radhakrishna Ganti**
Research Guide	Research Guide
Professor and HOD	Assistant Professor
Dept. of Electrical Engineering	Dept. of Electrical Engineering
IIT-Madras, 600036	IIT-Madras, 600036

Place: Chennai

Date: 1st June, 2018

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to the following people without whom my research work at IIT Madras, would not have been possible.

I am greatly indebted to my guides, Prof.Devendra Jalihal and Prof.Radhakrishna Ganti for sparing their precious time in providing valuable inputs and guidance throughout my thesis work. I would like to thank Prof.Nitin Chandrachoodan for his unvaluable help and for guiding me in the HDL work and sparing time in his busy schedule for my doubts.

I would like to thank my lab mates Mani Krishna and Shiva Prasad for helping me in understanding the tool and softwares required for this project.I would like to thank Arjun for helping me in understanding several communication problems.

My deepest gratitude to my mother and father for their tremendous amount of support, encouragement, patience, and prayers.

ABSTRACT

The project work is a collaboration of Hitachi and IITM.Due to the recent progress in cellular technologies,there has been increase in the innovation with regards to the capacity and data rates.With much anticipated 5G, the focus has shifted to Internet of Things(IoT).It is seen as an opportunity to redefine the network to accommodate a wealth of new and diversely connected devices.Two major areas envisioned in IoT are massive machine to machine communication and ultra reliable communication.The first one is the scenario where thousand of sensor nodes talk to each other at very low data rates and in the second,the ultimate goal is to have robust and authenticated communication irrespective of the environment in which the communication takes place. Wireless M2M(Machine to Machine) communication in the process industry plants is severely affected by slow fading nature of propagation channel and multipath reflections due to presence of metallic obstructions that interact with the electromagnetic waves.Slow fading means if the signal is in deep fade it will continue to remain in deep fade for longer time with higher probability.

An idea of introducing fast fading through polarization angle diversity is presented by DR.Takei of HITACHI.The idea is to send the same data through different polarization angles at the transmitter and receive it in those many polarizations.The signal processing techniques are used at the receiver to decode the signal and achieve diversity.

We have designed and developed a modulator-demodulator (MODEM) based on this technique.Here we discuss briefly the physical layer part of the code,its porting and testing onto the ZC702 FPGA Platform.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Project Background

The project is about development of rotating polarized wave based modem.The concept of rotation polarized wave was brought forward by Dr.Takei of HITACHI. The previous teams have developed the Physical layer of the transmitter and receiver for 1:N protocol.It was further worked upon by the current team.The PHY layer of the N:1 protocol has been completed and demonstrated.The transmitter and receiver were implemented using the ZC702 and AD9361 RF boards.The final implementation is such that the a PHY layer has to run on FPGA and MAC layer is to run on a processor.The current position of the project is that protocol for the N:1 communication has been developed and the physical layer code has been demonstrated.

Many control systems are functioning in industries and reliable communication for such machineries is required.A reliable M to M communication system is required.Wired communication can transmit without much loss of data but those systems are prone to physical damage and security breaches.So a reliable M to M wireless communication system is required.But the wireless systems are prone to slow fading(static) nature of the propagation channel and the communication via wireless systems is not much reliable if the machines are not in line of sight.

Also M2M radio systems in an infrastructure system are surrounded by many electromagnetic scatterers, its communication is in an NLOS situation. Data is transmitted mainly by reflected waves. Because of the many electromagnetic reflectors, there are many transmission paths. And reflected waves which are transmitted by different paths are added at a receiver. If these waves are co-phase, a total received power will increase, and if they are anti-phase, the power will decrease. Because of this, there are some positions where we can't communicate. This is a case of multipath fading. By using polarization angle diversity, one can reduce this multipath fading. When a radio wave is reflected, its polarization angle is changed. Therefore, each polarization angle has its transmission path (or paths). If we can change a polarization angle, we can control multipath fading. This is a basic theory of polarization angle diversity.

1.2 Generating Rotating Polarization Wave

To realize polarization angle diversity, we use a rotating polarized radio wave. Its polarization angle is rotating and time-dependent,this wave is different from a circularly polarized wave. A rotation frequency of a circularly polarized wave is same as its radio frequency. When we use the 860MHz band radio wave, its rotation frequency is also 860MHz. It is too fast to control the polarization angle. On the other hand, our rotating polarized wave is rotating slowly. Because of this low rotation frequency, we can change a polarization angle and control multipath fading.To create a rotating polarized radio wave, we use two radio waves which have different frequencies.

The two radio waves which have different frequency (f1, f2) are mixed and transmitted. In one antenna, they are simply mixed. In the other, they are mixed after 90 degrees delayed. The radio waves from these antennas have two factors. One is a (f1 + f2)/2 frequency radio wave and the other is a (f1 - f2)/2 frequency radio wave. The high frequency factors have same phase between the two antennas, and low frequency factors have 90 degrees difference. So, the radio wave from the antennas is rotating at the low frequency.The two sinusoids with frequencies very close to each other are added together to generate a product of sinusoids with two different frequencies. The two frequencies in the product sinusoid have as one component the average of both the parent frequencies while the other component consists of the difference of two parent frequencies. The average term is close to both the parent frequencies while the difference term is much smaller than the parent frequencies. Generally, the parent frequencies are chosen closer to the desired center/RF frequencies. The average frequency term is the desired center frequency or the frequency of propagation while the difference frequency term is the frequency of rotation. A circularly polarized wave has the frequency of rotation equal to the frequency of propagation.

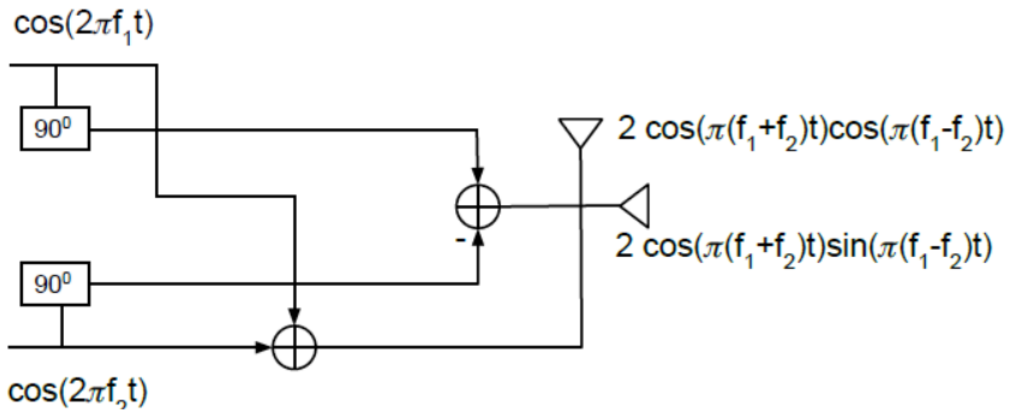

cos(2πf₁t)
90⁰
90⁰
-
2 cos(π(f₁+f₂)t)cos(π(f₁-f₂)t)
2 cos(π(f₁+f₂)t)sin(π(f₁-f₂)t)
cos(2πf₂t)

Figure 1.1: RPW generation

For the ease of implementation, we generate the sequence associated with the rotation frequency at the baseband and pre multiply it with the frame to be transmitted. This frame which is pre multiplied with the rotation sequence is generated by FPGA board, which is given to AD9361 RF module chain to up convert the signal to the desired centre frequency, thus producing the required frequency of rotation and frequency of propagation at the output.

At such high rates, resolving the multi paths at the receiver is not a feasible option. RPW, a competent technique brings about a solution to resolve the multi paths at the receiver. By significantly reducing the frequency of rotation much lower than the baseband sampling rate, the multi paths can be easily resolved. These multi paths are important because, they originate due to the different incident polarization angles. The incident angles of polarization are changed as the wave rotates. But, the number of polarization angles that we resolve at the receiver, depends on the sampling rate and the frequency of rotation. Hence, we must device a rule to fix the frequency of rotation depending on the number of polarization angles required.

Let us consider the sampling frequency to be fs Msps, the number of polarizations required to be resolved be N, over sampling factor be k, sampling time be Ts.

The following steps illustrate the procedure to fix the frequency of rotation.

1. In order to change the incident angles of polarization of a data bit, the bit should occupy the time taken for one complete rotation.

2. It is therefore required to spread the bit to fill the entire rotation period.
3. The spreading length must be equal to the number of incident polarizations needed.
4. After oversampling, the number of samples that should fit in one rotation is N*k.
5. The frequency of rotation, represented by fr, can now be calculated using the inverse of the time taken for the total number of samples required to fit in one rotation as shown in equation fr = fs/(N*k).

The rotating polarization wave can either be generated using the structure shown in the figure 1.1 or by generating the rotation frequency term separately in the baseband and up-converting it to required RF frequency.

1.3 Effectiveness of Rotating Polarization

As the incident polarization angle changes, the obstructions seen by the wave changes and hence the channel experienced by the wave changes. Thus, at each polarization angle, the wave experiences a different channel coefficient, hence making the fading to vary at a much faster rate. This makes the signal to come out of the deep fade scenario very easily.

1.4 Frame Structure

The frame structure used in the rotating polarized wave based modem, consists of the PHY layer and the PHY payload.PHY payload consists of the MAC layer and the MAC payload along with preamble and channel estimation bits.

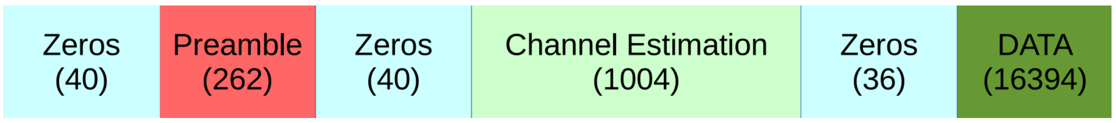


Figure 1.2: Frame Structure

There are forty 0's before the preamble , forty 0's between preamble & channel estimation bits , thirty six 0's between channel estimation bits & Physical payload data.

1.4.1 Preamble Data

A PN sequence, although resembles a random noise, is a deterministic sequence that repeats itself after a certain period. The period can be as large as possible. These sequences are typically used for frame detection/synchronization because of three important properties.

1. On the lines of random noise properties, this sequence has very low correlation with any other sequence in the set, or with the same sequence at a different time offset, or with thermal noise, or with any narrowband interference.
2. In contrast to random noise, PN sequences can be deterministically generated at the transmitter and receiver.
3. This sequence has a very high correlation with the same sequence present in the received signal.

The preamble sequence used for frame synchronization is a 64 length PN sequence.This 64 length Pseudo-Noise sequence bits are oversampled using an oversampling factor of 4.The resultant data is then passed through RRC filter of truncated to length 10. So the entire length of the preamble is 262.
64 bits–>63*4+1=253(oversampling)—>253+9=262(RRC filtered).

1.4.2 Channel Estimation Data

The chip sequence consists of 1000 bits.The sequence of 200 bits is repeated 5 times.The 200 bit sequence is hundred 1's and hundred -1's.The channel Estimation bits are also passed through a 5 point RRC filter an the output sequence is of length 1004.

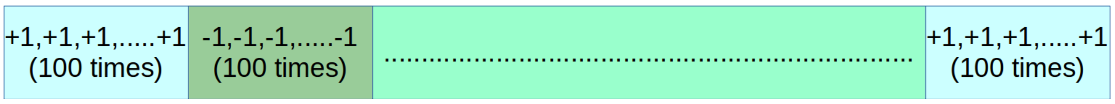


Figure 1.3: Channel estimation data

1.4.3 Spreading sequence

In order to employ polarization angle diversity in our model, the modulated data is spread using a 64 bit chip sequence so that each bit can occupy the time taken by one complete rotation. The chip sequence used is:1,-1,-1,-1,-1,-1,1,1,1,1,1,1,-1,1,-1,1,-1,1,1,-1,-1,1,1,-1,1,1,1,-1,1,1,-1,1,-1,-1,1,-1,-1,1,1,1,-1,-1,-1,1,-1,1,1.

1.4.4 Pulse Shaping (SRRC):

In communications systems, two important requirements of a wireless communications channel demand the use of a pulse shaping filter. These requirements are:

1. Generating bandlimited channels, and
2. Reducing inter symbol interference (ISI) from multi-path signal reflections.

Both requirements can be accomplished by a pulse shaping filter which is applied to each symbol. In band-limited channels, inter-symbol interference (ISI) can be caused by multi-path fading as signals are transmitted over long distances and through various mediums. More specifically, this characteristic of the physical environment causes some symbols to be spread beyond their given time interval. As a result, they can interfere with the following or preceding transmitted symbols. By applying the pulse shaping filter to each symbol that is generated, we are able to reduce channel bandwidth while reducing ISI. The maximum amplitude of the pulse-shaping filter occurs in the middle of the symbol period. In addition, the beginning and ending portions of the symbol period are attenuated. Thus, ISI is reduced by providing a pseudo-guard interval which attenuates signals from multi-path reflections.

Instead of using a single Raised Cosine filter at the transmitter, a square root raised cosine filter is used at both transmitter and receiver. This is to achieve matched filtering which maximizes SNR of the system.In our present design SRRC filter with a roll off factor of 0.5 is chosen to perform pulse shaping at transmitter and receiver.The SRRC filter's oversampling factor of 4.This SRRC filter is truncated to 9 point sequence and the filter coefficients are scaled by a factor of 1.849.

1.4.5 Payload

The data samples consists of the required 64-bits data that need to be transmitted.The 64 bits are encoded by using BPSK,spreaded by 64 length spreading sequence,the spreaded data is upsampled by 4 and the data is further srrc filtered.

1.5 Transmission Of Frame

The entire frame which has preamble, channel estimation bits and PHY payload,is the send by the transmitter.The receiver receives the data and detects the start of the frame by correlating the received data with the preamble sequence.After detecting the start of the frame,channel is estimated and equalization of the data is done to decode the bits.

1.6 Organization Of The Report

In chapter 2,the software used for optimizing the codes,implementing the codes and also hardware used for running the codes are mentioned.

In chapter 3,we will discuss discuss how to optimize the code using the directives options and the reports of the software tools.The codes of transmitter,receiver along with the test-benches will be explained.

In chapter 4,the connection that are used for interfacing different IP's,reports of vivado design and also embedded code that is programmed on FPGA using SDK environment is also discussed.

In chapter 5, we will discuss about the Hardware implementation setup and the problem of frequency offset along with frequency estimation block is discussed.

CHAPTER 2

Introduction to Hardware Platforms ZC702,AD9361 and Software tools Vivado HLS,Vivado,SDK Environment

2.1 Introduction of ZYNQ ZC702 FPGA Platform:

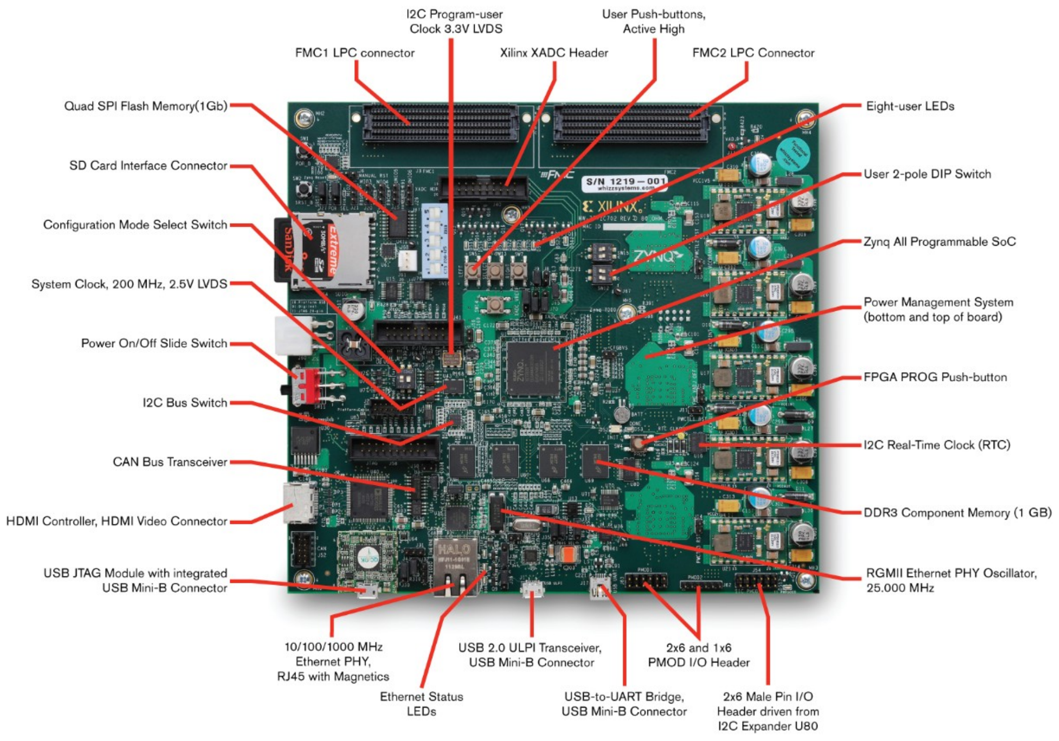
The ZC702 is an evaluation board that acts as a platform for developing hardware descriptions. The ZC702 is comprised of many components and peripheral interfaces common in many embedded applications, such as DDR3-SDRAM for main memory, general purpose input/outputs (GPIO), and universal asynchronous receiver and transmitters (UART), LEDs, and pushbuttons. The ZC702's features can even be extended by peripheral module (PMOD) connectors.

The main components of the zynq board are as follows:

1. **Zynq (ZC702):** The Z-7020 is the System-on-Chip device that combines the processing capabilities of a Dual-core ARM Cortex-A9 processor with programmable logic of an Artix-7 Field Programmable Gate Array (FPGA). The Zynq ARM processor executes the program that is written in SDK environment. The PL(programmable logic area) contains the required design or the required algorithm that is to be executed .

2. **Slide Switches:** These are two sets of slide switches which are set in order to enable JTAG mode and utilize the JTAG Module, which is necessary for programming the ZC702.The JTAG Module is one of the available options for downloading programs, probing, and debugging certain hardware on the ZC702. There are also other options available, but the JTAG approach is the most convenient since it only required a Standard-A to Micro-B USB cable to connect from the JTAG Module to the computer.

3. **USB to UART Bridge Device:** The USB to UART Bridge allows a host computer to connect to the ZC702's UART with a Standard-A to Mini-B USB cable. If the Virtual COM Port (VCP) drivers are also installed on the host computer, the USB-to-UART Bridge will appears as a COM port and thereby enable asynchronous serial communication with the ZC702. A user can then communicate with the ZC702, using software such as HyperTerm or PuTTy.



I2C Program-user Clock 3.3V LVDS
User Push-buttons, Active High
FMC1 LPC connector
Xilinx XADC Header
FMC2 LPC Connector
Quad SPI Flash Memory(1Gb)
Eight-user LEDs
SD Card Interface Connector
User 2-pole DIP Switch
Configuration Mode Select Switch
Zynq All Programmable SoC
System Clock, 200 MHz, 2.5V LVDS
Power Management System (bottom and top of board)
Power On/Off Slide Switch
FPGA PROG Push-button
I2C Bus Switch
I2C Real-Time Clock (RTC)
CAN Bus Transceiver
DDR3 Component Memory (1 GB)
HDMI Controller, HDMI Video Connector
USB JTAG Module with integrated USB Mini-B Connector
RGMII Ethernet PHY Oscillator, 25.000 MHz
10/100/1000 MHz Ethernet PHY, RJ45 with Magnetics
USB 2.0 ULPI Transceiver, USB Mini-B Connector
2x6 and 1x6 PMOD I/O Header
Ethernet Status LEDs
USB-to-UART Bridge, USB Mini-B Connector
2x6 Male Pin I/O Header driven from I2C Expander U80

Figure 2.1: AD9361

The number of available resources on ZYNQ ZC702 platform are as follows:

1. **BRAM_18K**=280.
2. **DSP48E**=220.
3. **Flipflops**=106400.
4. **Look up table**=53200.

2.2 Introduction of AD9361 RF AGILE TRANSCEIVER:

AD9361 is a high performance, highly integrated RF Agile Transceiver. The programmability and wide-band capability make AD9361 ideal for a broad range of transceiver applications. The device combine an RF front end with a flexible mixed-signal baseband section and integrated frequency synthesizers, simplifying design-in by providing a configurable digital interface to a processor. AD9361 operates in the 70 MHz to 6.0 GHz range, all covering most licensed and unlicensed bands. Channel bandwidths on the AD9361 is from 200 kHz to 56 MHz. AD9361 is a 2 Rx, 2 Tx device.

The AD9361 is a highly integrated radio frequency (RF) transceiver capable of being configured for a wide range of applications. The device integrates all RF, mixed signal,and digital blocks necessary to provide all transceiver functions in a single device. Programmability allows this broadband transceiver to be adapted for use with multiple communication standards, including frequency division duplex (FDD) and time division duplex (TDD) systems. This programmability also allows the device to be interfaced to various baseband processors (BBPs) using a single 12-bit parallel data port, dual 12-bit parallel data ports, or a 12-bit low voltage differential signaling (LVDS) interface. The LVDS interface is used on the ADFMCOMMS2-EBZ, AD-FMCOMMS3-EBZ, AD-FMCOMMS4-EBZ and AD-FMCOMMS5-EBZ as the CMOS drive strength is too weak to go through a connector.

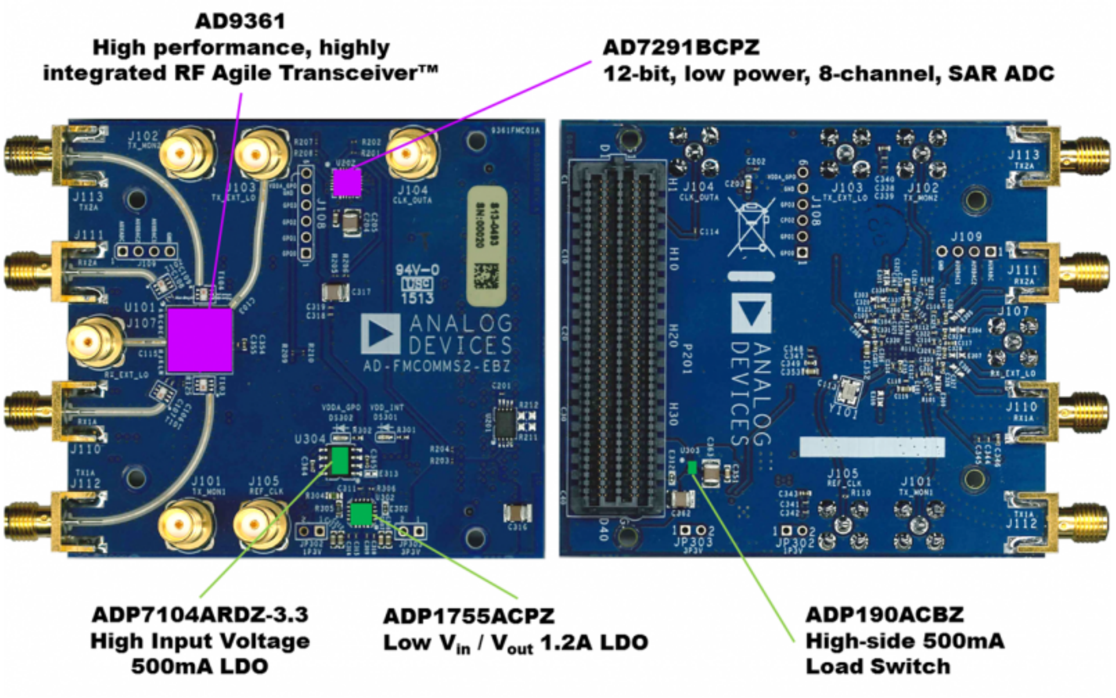

AD9361
High performance, highly integrated RF Agile Transceiver™
AD7291BCPZ
12-bit, low power, 8-channel, SAR ADC
ADP7104ARDZ-3.3
High Input Voltage 500mA LDO
ADP1755ACPZ
Low V_{in} / V_{out} 1.2A LDO
ADP190ACBZ
High-side 500mA Load Switch

Figure 2.2: AD9361

The AD9361 also provide self-calibration and automatic gain control (AGC) systems to maintain a high performance level under varying temperatures and input signal conditions.In addition, the device includes several test modes that allow system designers to insert test tones and create internal loop-back modes that can be used by designers to debug their designs during prototyping and optimize their radio configuration for a specific application.

2.2.1 Transmit Chain:

The transmitter section consists of two identical and independently controlled channels that provide all digital processing, mixed signal, and RF blocks necessary to implement a direct conversion system while sharing a common frequency synthesizer.The TX signal path receives 12-bit 2's complement data in I-Q format from the digital interface, and each channel (I and Q) passes this data through a fully programmable 128-tap FIR filter with interpolation options. The FIR output is sent to a series of additional interpolation filters that provide additional filtering and data rate interpolation prior to reaching the 12-bit DAC.

The FIR filter, and of the three interpolating filters can individually be controlled and bypassed if desired. Each 12-bit DAC has an adjustable sampling rate. The DAC's analog output is passed through two low pass filters (to remove sampling artifacts) prior to the RF mixer. The corner frequency for each low-pass filter is programmable. At this point, the I and Q signals are recombined and modulated on the carrier frequency for transmission to the output stage. The combined signal also passes through analog filters that provide additional band shaping, and then the signal is transmitted to the output amplifier. Each transmit channel provides a wide attenuation adjustment range with fine granularity to help designers optimize signal-to-noise ratio (SNR).

Both the I and the Q paths are schematically identical to each other.Self-calibration circuitry is built into each transmit channel to provide automatic real-time adjustment. The transmitter block also provides a TX monitor block for each channel. This block monitors the transmitter output and routes it back through an unused receiver channel to the BBP for signal monitoring. The TX monitor blocks are available only in TDD mode operation while the receiver is idle.

2.2.2 Receive Chain:

The receiver section contains all blocks necessary to receive RF signals and convert them to digital data that is usable by a BBP. There are two independently controlled channels that can receive signals from different sources, allowing the device to be used in multiple input, multiple output (MIMO) systems while sharing a common frequency synthesizer.

Each channel has three inputs that can be multiplexed to the signal chain, making the AD9361 suitable for use in diversity systems with multiple antenna inputs. The receiver is a direct conversion system that contains a low noise amplifier (LNA), followed by matched in-phase (I) and quadrature (Q) amplifiers, mixers, and band shaping filters that down convert received signals to baseband for digitization. External LNAs can also be interfaced to the device, allowing designers the flexibility to customize the receiver front end for their specific application.

The AD9361 RX signal path passes down-converted signals (I and Q) to the baseband receiver section. The baseband RX signal path is composed of two programmable analog low-pass filters, a 12-bit ADC, and four stages of digital decimating filters. Each of the four decimating filters can be bypassed. The corner frequency for each low-pass filter is programmable. Note that both the I and Q paths are schematically identical to each other.

Gain control is achieved by following a preprogrammed gain index map that distributes gain among the blocks for optimal performance at each level. This can be achieved by enabling the internal AGC in either fast or slow mode or by using manual gain control, allowing the BBP to make the gain adjustments as needed. Additionally, each channel contains independent RSSI measurement capability, dc offset tracking, and all circuitry necessary for self-calibration.

The receivers include 12-bit, sigma-delta ADCs and adjustable sample rates that produce data streams from the received signals. The digitized signals can be conditioned further by a series of decimation filters and a fully programmable 128-tap FIR filter with additional decimation settings. The sample rate of each digital filter block is adjustable by changing decimation factors to produce the desired output data rate.

2.2.3 Rx and Tx Filtering:

In both the receive and transmit chains, there are :

1)Analog low pass filters either to removing sampling artifacts on the Tx side, or to band shaping to reduce adjacent-channel interference on the Rx side.

2)Digital interpolation/decimation filters to up/down convert from the digital baseband rate (64.11MSPS max) to the actual ADC (640MSPS) or DAC (320MSPS) rates.

Irrespective of the implementation (analog or digital) these filters impact the magnitude and phase in the passband. This must be compensated somewhere in the system. It can easily be done (and therefore is normally done) inside the 128 tap FIR filter. The FIR filter is not only used to implement a low pass filter, but also to compensate for the magnitude and phase impacts the analog and digital half band filters create in the baseband area of interest.

These filters depend on sample rates, clocks, and data rates (which sets the digital half band filters), and the RF bandwidth (which sets the analog filters). Loading a filter, and then changing anything in the system, will negatively affect overall baseband performance.

2.2.4 Clocking options:

The AD9361 uses fractional-n phase locked loops (PLLs) to generate the transmitter and receiver local oscillator (LO) frequencies as well as the oscillator (the baseband PLL) used for the data converters, digital filters, and I/O port. These PLLs all require a reference clock input, that includes a digitally controlled crystal oscillator (DCXO) function, an on-chip programmable/variable capacitor. This capacitor can tune the crystal frequency variance out of the system, resulting in a more accurate reference clock from which all other frequency signals are generated. The input for the DCXO can be provided by two different sources:

1)To connect an external oscillator or clock distribution device (such as the AD9548 to the XTALN pin (with the XTALP pin remaining unconnected). If an external oscillator is used, the frequency can vary between 10 MHz and 80 MHz. This is for applications such as wireless base-stations, which require that the reference clock lock to a system master clock.

2)To use a dedicated crystal with a frequency between 19 MHz and 50 MHz connected between the XTALP and XTALN pins. This is typically used in wireless user equipment (UE), which do not typically need to be locked to a master clock but they do need to adjust the LO frequency periodically to maintain connection with a base-station (BTS). The BTS occasionally informs the UE of its frequency error relative to the BTS, and the baseband processor can adjust the reference clock frequency and thus the LO frequency as needed.

The DCXO tuning function can also be used with the on-chip temperature sensor to provide oscillator frequency temperature compensation during normal operation.

2.2.5 AD9361 No-OS SDK API's To Configure Tx and Rx parameters:

1)*struct ad9361_rf_phy *ad9361_init(AD9361_InitParam *init_param):*

Initializes the FMCOMMS2 board. Receives as parameter a structure that contains the AD9361 initial parameters. Returns a structure that contains the AD9361 current state in case of success, negative error code otherwise.

2)*int32_t ad9361_set_rx_rf_gain(struct ad9361_rf_phy *phy, uint8_t ch, int32_t gain_db):*

Sets the receive RF gain for the selected channel. Receives as parameters a structure that contains the AD9361 current state, the desired channel number (0,1) and the RF gain.

3)*int32_t ad9361_set_rx_rf_bandwidth(struct ad9361_rf_phy *phy, uint32_t bandwidth_hz):*

Sets the RF bandwidth. Receives as parameters a structure that contains the AD9361 current state and the desired bandwidth in Hz. Similar function for tx also.

4)*int32_t ad9361_set_rx_sampling_freq(struct ad9361_rf_phy *phy *phy, uint32_t sampling_freq_hz):*

Sets the sampling frequency. Receives as parameters a structure that contains the AD9361 current state and the desired sampling frequency in Hz.Similar function for tx also.

5)*int32_t ad9361_set_rx_gain_control_mode (struct ad9361_rf_phy *phy *phy, uint8_t ch, uint8_t gc_mode):* Sets the gain control mode for the selected channel. Receives as parameters a structure that contains the AD9361 current state, the desired channel (0,1) and the gain control mode (GAIN_MGC, GAIN_FASTATTACK_AGC, GAIN_SLOWATTACK_AGC, GAIN_HYBRID_AGC).

6)*int32_t ad9361_set_tx_attenuation(struct ad9361_rf_phy *phy *phy,uint8_t ch,uint32_t attenuation_mdb):*

Sets the transmit attenuation for the selected channel. Receives as parameters a structure that contains the AD9361 current state, the desired channel number (0, 1) and the attenuation in dB.

7)*int32_t ad9361_set_rx_lo_freq(struct ad9361_rf_phy *phy *phy, uint64_t lo_freq_hz):*

Sets the LO frequency. Receives as parameters a structure that contains the AD9361 current state and the desired LO frequency in Hz.

All functions except ad9361_init() returns 0 in case of success, negative error code otherwise.

2.3 Vivado HLS

The software which has been used to convert c++ code into HDL is VIVADO HLS, which is provided by Xilinx.The Xilinx Vivado High-Level Synthesis (HLS) is a tool that transforms a C specification into a register transfer level (RTL) implementation that can synthesize into a Xilinx field programmable gate array (FPGA). For the c codes to be converted into HDL codes,it has to be written in particular manner which will be discussed later.VIVADO HLS also provides directives like pipelining,unrolling,array partitioning etc which can be used to optimize the code .The software shows the minimum & maximum number of clock cycles taken by the functions in the code and also minimum & maximum amount of time taken by the operations,loops in the c++ code.By analyzing the times taken by different instances of the code, we can know where to optimize the code.VIVADO also shows the amount of hardware which has been used for the HDL and whether the hardware used in within the limit.VIVADO has a option known as "co-simulation" which simulates and runs both the c++ code and also the HDL code by which we can check the accuracy of the HDL code.

High-level synthesis also bridges hardware and software domains, providing the following two primary benefits:

1. Improved productivity for hardware designers (Hardware designers can work at a higher level of abstraction while creating high-performance hardware).
2. Improved system performance for software designers (Software developers can accelerate the computationally intensive parts of their algorithms on a new compilation target, the FPGA).

2.3.1 Design steps in Vivado HLS:

1. First step is to develop algorithm at the C level.

2. Next, we will verify the algorithm at the C level.
3. Next, we will synthesize the C algorithm into an RTL implementation. (Vivado HLS tool can automatically create an RTL implementation of our C algorithm. Vivado HLS will also automatically create data path and control path modules required to implement our algorithm in hardware).
4. Next, we will generate comprehensive reports and analyze the design. Vivado HLS automatically creates synthesis reports to help us understand the performance of the implementation.
5. Next, we verify the RTL implementation.
6. Last step, package the RTL implementation into a selection of IP formats.

2.3.2 Key Benefits of Vivado HLS:

1. Portability of the source code to any hardware which rarely involves restructuring of the code.
2. Data dependencies and the sequence of operation are analyzed by HLS tools to exploit parallelism which can be pipelined further to achieve the desired timing constraints.
3. A pipelined architecture can be inferred from loops which involves less data dependencies between successive iterations.
4. Iterations in a loop can be made parallel by loop unrolling which divides the loops to multiple hardware.

2.3.3 Data types in Vivado HLS:

Floating Point Datatypes:

Vivado HLS supports standard floating type data types of C/C++.But they use lot of resources/Area and also these datatype effect the Latency and throughput of the design.So,we generally use fixed point datatypes.

Fixed Point Datatypes:

The specification of fixed point data type affects the resource utilization, timing performance, and power consumption greatly. But,if word length is Under-specified the compromises accuracy, while over-specifying will leads to increased resources, inflated power consumption, and a sub-optimal maximum clock frequency. So,a trade-off of word length has to be chosen accordingly. Vivado HLS supports both the native C/C++ data types plus the arbitrary precision data types for integer (for C/C++) and fixed point (for C++ only).Arbitrary precision data types are widely used to optimized the area/resources in the required design. They can be used in HLS C program by adding ap_int.h , ap_fixed.h.

1)ap_int.h: These data types are used for integer types of data. A variable range of integer bits can be configured. Data types in this libraries are declared by using the format ap_int<N>.(N bit integer format).Similarly unsigned integers can also be declared as follows ap_uint<N>.

2)ap_fixed.h: These data types are used for fixing decimal types of data. A variable range of bits can be configured and the decimal number can be approximated according to the required precision. Data types in this libraries are declared by using the format ap_fixed<W,I,Q,O,N>.(I bits are for integer and W-I for fractional part).Similarly unsigned decimal can also be declared as follows ap_fixed<W,I,Q,O,N>.

VIVADO HLS also supports the complex data type by including the header file "complex.h".

2.3.4 Optimization directives Vivado HLS:

Pipelining:

In HLS, pipelining refers to the partitioning into sub stages of an arbitrary set of dependent operations. The objective for pipelining is to enable parallel processing, and thereby increase the throughput supported by the design. Pipelining can be applied as a directive in Vivado HLS, at the level of functions and loops.

Latency and Initiation Interval(II):

Latency is defined as the number of clock cycles between applying an input, and achieving the corresponding output. Latency can be viewed at different levels of hierarchy and in the context of loops, latency refers to the completion of all iterations of the loop and the term iteration latency is used when referring to a single iteration. The total latency is equal to the iteration latency, multiplied by the number of iterations of the loop (known as trip count). Latency can also be specified as a design constraint by the user, and the Vivado HLS tool optimizes the design wherever possible to meet the requirement.

The Initiation Interval (II) is the number of clock cycles that separate the acceptance of inputs to the Vivado HLS design. Without applying directives, the initiation interval will be one cycle more than the latency, because the default behaviour of Vivado HLS is to optimize for area, resulting in a serial design. However the use of pipeline directive can reduce the Initiation Interval to much less than the latency of the design. This results in an increase in the area of the design, so there is a trade-off involved. Initiation interval corresponds directly to throughput. Throughput expresses the rate at which data can be passed through the system. The best possible initiation interval is 1, meaning that new input samples can be accepted on every clock cycle, in which case the throughput is equivalent to the clock rate. Higher levels of throughput can be achieved through use of partial loop unrolling, or by replicating a synthesized function.

Loop Optimizations:

Loops are basic constructs of any algorithm and expresses operations that are repetitive in nature. Several loop optimizations can be made using directives, enabling architectural variation exploration with almost no change required in the software code. The various optimizations performed in loops are

1. **Loop Unrolling:** It means that the hardware inferred from the loop body is created N times. Practically it can be less than N, depending on whether data dependency or memory operations are present in the design. Advantage is throughput increase and disadvantage is the increase in hardware resource.
2. **Partially Unrolled:** Only a part of the loop is unrolled.This generally is a trade-

off between rolled and unrolled architecture.

Array Optimization Directives

As arrays represent storage they are synthesized into memories. The memories inferred are mapped to physical resources on the FPGA as Block RAM, or distributed RAM. Various directives are used to map these memories to physical memory resources. Some of the optimizations on arrays are discussed below.

1. **Array Map:** It maps several small arrays to be combined into a single, larger array. Mapping can be either horizontal (arrays are concatenated to form an array with more elements), or vertical (array elements are combined, resulting in an array with longer words).

2. **Array Partition:** It maps the subdivision of a large array into a set of smaller ones. It increase the rate at which memory transactions can take place. In the extreme case, array partitioning will subdivide an array into individual register elements.

3. **Array Reshape:** This allows an array with many elements, each with short words, to be reshaped into an array with fewer, longer words. This directive reduces the number of required memory accesses.

2.3.5 Exporting from Vivado HLS:

Designs can be exported from Vivado HLS to permit easy integration of Vivado HLS IP with other development tools. IPs are exported from HLS to the IP-XACT format, which allows the module to be integrated into a Vivado IP Integrator design. This results in a zip folder residing in the ip sub-folder under impl folder of the respective solution, and represents the IP catalogue package. It also contains API's required to use the IP in Xilinx SDK environment. This format allows easy sharing and distribution of IP across all platforms.

2.3.6 Limitations of Vivado HLS:

1. Dynamic memory allocation.
2. Operating system (OS) operations.
3. Software algorithms are sequential in nature whereas hardware operates concurrently. HLS must map the sequential algorithm onto concurrent hardware.
4. Algorithms are based on pointers, whereas hardware implementations rely on arrays and array references which makes HLS finding it difficult to map to hardware and hence at many times a restructuring of the code is required.Recursive algorithms are very difficult to translate to hardware using HLS.

2.4 VIVADO

The Vivado suite of design is a tool that support all phases of FPGA designs starting from design entry, simulation, synthesis, place and route, bitstream generation, debugging, and verification as well as the development of software targeted for these FPGA.

2.4.1 IP's in Vivado Design Suite:

Intellectual property (IP) cores are fundamental criteria when selecting which FPGA vendor and specific part to choose for a design. IP provides an easy mechanism for incorporating complex logic in your designs, from high-speed gigahertz transceivers (GT's) to digital signal processors (DSPs) as well as soft microprocessors (MicroBlaze) to an embedded ARM system on a chip (SoC).

Xilinx provided IP's have been optimized and tested to work with the FPGA resources including DPS, block RAM, and IO, greatly accelerating design development.

User defined IP's are generated in Vivado HLS tool and the corresponding IP's which are requires for our design can be easily includes by using Vivado design Suite.

In this project use of IP's which are provided by the Vivado design suite to serve various functions such as Controlling speed mismatch between two IP's (Mostly by using FIFO generator block) ,considering few data bits of a data bus for signal processing based operations (xlslice block),controlling operations of the User defined IP's such as TX,Packet detect and RX (using xlconstant) is implemented.

2.4.2 Different Xilinx Vivado IP’s Used :

FIFO Generator IP:

The First in First out (FIFO) IP block is used as a temporary storing element .It has many interfacing option to integrate with the different IP interfaces such as AXI-Memory mapped, AXI-Stream Interface. The FIFO has three modes,they are Native, AXI-Memory mapped,AXI-Stream. There are also implementation options based on the read and write clocks options.

In this project,the Native Interface,Independent clocks with built-in FIFO configuration is considered.In native ports the read mode is a standard FIFO mode,in this mode the user is provided with the data on the cycle after it was requested.The read/write data port can be configured according to the data that needs to be stored.There are additional flag signal which gives many other functionality.

The full,wr_en and empty,rd_en ports are used to interface with the axi-stream interface of AD9361 to read and write samples from ADC and DAC respectively. The different read and write clocks ports are connected to the different IP's (usually AD9361 interface and the user defined IP's) accordingly when the different IP's are operating with different clocks.

In this project, a sampling rate of 10MSPS is chosen as result,effectively clock of AD9361 is 10MHz.The user defined IP's has clock rate of 100MHz. As a result different clock rate FIFO is considered to integrate this two IP's.

Utility vector logic:

This IP is used to implement the basic gate logics such as and,or,not,xor operations. The number of input and output ports can be configured.

In this project,the not gate is used to interface the hls::stream with the FIFO IP block. The empty signal of FIFO is given as input to not gate and the output of not gate is connected to the stream interface's T_valid signal.(this implies that the FIFO when not empty has a valid data that has to be read by the hls::stream interface).

Similarly the full signal of FIFO is given as input to not gate and the output of not gate is connected to the stream interface's T_Ready signal.(this implies that the hls::stream interface should not write any valid data to FIFO). A rst (reset) port is also available,this port is used to clear the FIFO IP.

xlconstant:

his Vivado IP is used to implement a constant value on its output port according to the value that it has been configured.

In this project, the packet detector IP block requires to read the input hls::stream continuously, so the ap_start signal is given high signal.(This configures the user defined block to be in auto restart mode).

ZYNQ processing system IP :

Xilinx provides the Processing System IP Wrapper for the Zynq-7000 to accelerate our design and its configuration for our embedded products. The Processing System IP is the software interface around the Zynq-7000 Processing System. The Zynq-7000 family consists of a system-on-chip (SoC) style integrated processing system (PS) and a Programmable Logic (PL) unit, providing an extensible and flexible SoC solution on a single die.The Processing System IP Wrapper acts as a logic connection between the PS and the PL while assisting you to integrate custom and embedded IPs with the processing system using the Vivado IP integrator.

Key Features and Benefits:

1. Enable/Disable I/O Peripherals (IOP).
2. Enable/Disable AXI I/O ports (AIO).
3. MIO(Multiplexed I/O) Configuration.
4. Extended Multiple Use I/Os (EMIO).

5. DDR Configuration.
6. Security and Isolation Configuration.
7. Interconnect Logic for Vivado IP - PS interface.
8. PL Clocks and Interrupts.

2.4.3 Modules used in Vivado

The modules are the user defined block,these are in hardware descriptive language.In this project two modules are added to the design.

tx_bitslice:

The tx_bitslice verilog module accepts 32-bits data wires and splits it into two 16-bits data wires.This is to given the 12-bit dac the real(16-bit) and imaginary(16-bit) parts of the tx complex 32-bit data.

rxconcate:

The rxconcate verilog module accepts two 16-bit data wires,considers the 12 lower bits in both the wires and write the output data wire by scaling the lower bits by +4. (As the adc is 12 bits only they are considered as valid data).

2.5 Introduction to SDK environment:

Xilinx SDK provides an environment for creating software platforms and applications for Xilinx processor cores.It works with hardware designs generated with Vivado.

The two main terminologies used in Xilinx SDK are hardware platform and software platform. The hardware platform is the embedded hardware design that is created in Vivado and exported in the form of an HDF/XML file through the use of export hardware wizard. Once the hardware platform is identified and imported, we create the software platform. A software platform is a collection of libraries and drivers that form the lowest layer of application software stack. The software applications must run on top of a given software platform, using the provided API's. Therefore before creating and use software applications in SDK, a software platform project must be created. SDK includes the following two software platform types. They are:

1)Standalone OS It is a simple and single-threaded environment that provides basic features like standard input/output and access to processor hardware features. In this project we extensively used this software platform.

2)Xilkernel A simple and lightweight kernel that provides POSIX-style services such as scheduling, threads, synchronization, message passing, and timers.

2.5.1 IP Block Initialization and Status Monitoring Functions in SDK:

There are four standard functions for every Vivado HLS tool IP block:

1)Xtop level function name_Initialize()

2)Xtop level function name_Start()

3)Xtop level function name_IsDone()

4)Xtop level function name_IsIdle()

The required functions in any processor program are:

1)Xtop level function name_Initialize()

2)Xtop level function name_Start()

The initialization function Xtop level function name_Initialize has no effect on the IP block at the hardware level. The purpose of this function is to store identifier information for a specific instance of the example function in hardware. This identifier information is required by all other driver functions to ensure communication with the correct hardware module. The designer can use the same API to communicate with as many fabric instantiations of a hardware accelerator as needed by the application.

The start function, Xtop level function name_Start, pulses the ap_start signal on the target Vivado HLS tool hardware accelerator. This is a 1 clock cycle pulse that initiates the operation of the IP block. Depending on the Vivado HLS tool protocols selected

for the function I/O, the location of XExample_Start in the processor code must be changed to guarantee proper execution of the accelerator. Ports with a raw I/O protocol such as ap_none and ap_stable must be written before the start signal is received by the accelerator. Ports with qualified I/O protocols can be written before or after the XExample_Start function as long as the sequencing of the protocol is respected.
The other two functions generated for the example IP block are:

1)Xtop level function name_IsDone()

2)Xtop level function name_IsIdle()

Both of these functions are optional and allow the processor to monitor the status of the IP block.
Xtop level function name_IsDone() can be used by the processor to check when the IP block started by Xtop level function name_Start has finished execution. This enables polling the IP block for task completion.
Xtop level function name_IsIdle tells the processor if the core is running or not. It is not a deadlock checking function. This function returns false from the time the start function is executed by the processor until the ap_done signal is asserted by the IP block. If the system causes the Vivado HLS tool IP block to deadlock, the *_IsIdle function also returns false.

I/O Read and Write Function:

The number of I/O read and write functions directly corresponds to the number of I/O ports in the IP block.
Consider a function signature of the IP block generated with the Vivado HLS tool:

int example(int A, int B)

This function signature states that there are 2 input ports A and B and 1 output port for the function return value. Also, assume that port A has I/O protocol ap_none and port B has I/O protocol ap_hs. In this case, port A is associated with these driver functions:

XExample_SetA()

XExample_GetA()

The function XExample_SetA writes a value from the processor to port A of the IP block. As a result of using the ap_none protocol on port A, this function must be executed by the processor before the XExample_Start function. All ports utilizing the ap_none protocol are sampled and registered by the IP block at the time ap_start signal is received. Ports using the ap_stable protocol must also be set before the start signal is issued. Furthermore, the value of these ports must remain constant during the execution of the IP block. Ports with the ap_stable I/O protocol are never registered by the Vivado HLS tool IP block.

The purpose of the get function, XExample_GetA, is to verify the value written from the processor to the IP block.

As a result of using a qualified I/O protocol, port B is associated with these driver functions:

XExample_GetB()

XExample_SetB()

XExample_SetB_Vld()

XExample_SetB_Vld function is a direct result of the I/O protocol of port B, tells the IP block when the value of port B is valid and can be registered into the core. The evaluation of the valid signal on port B does not occur until the IP block receives the start signal from the processor. The processor can write the value of B at any time within its program execution. The Vivado HLS tool IP block stalls and waits for the valid signal in port B before continuing operation.

The read functions return the value of a register in the Vivado HLS tool IP block to the processor. In the case of input ports A and B, the read functions are

XExample_GetA()

XExample_GetB()

The purpose of these functions is to read the contents of a register in the Vivado HLS tool IP block. All user defined I/O ports have a read function associated with them. In the case of the function return value, the read function is of the form:

XExample_GetReturn()

The value of XExample_GetReturn is only valid after the done signal has been received from the IP block.

CHAPTER 3

Explanation and Optimization of the codes

3.1 Transmitter

The input bits to the transmitter code are PHY payload bits(which are 0's and 1's). 0's and 1''s are encoded using BPSK.Then the input bits(which are now sequence of -1's and 1's) are multiplied with chip sequence of length 64.

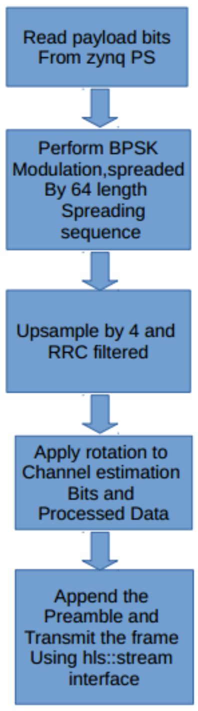

Read payload bits From zynq PS
Perform BPSK Modulation,spreaded By 64 length Spreading sequence
Upsample by 4 and RRC filtered
Apply rotation to Channel estimation Bits and Processed Data
Append the Preamble and Transmit the frame Using hls::stream interface

Figure 3.1: Transmitter Flow Chart

The resultant data is then oversampled with oversampling factor of 4.The oversampled data is then passed through RRC filter.

3.1.1 Transmitter Code Explanation

Transmitter Test Bench Explanation

1. The file which has the input data is opened and then input data is read from that file into an array.

   ```
   ifstream input_test;
   ifstream input_test1;
   input_test.open("test_data.txt");
   ap_uint <1 >out_bc1[56];
   char comma;
   int i=0;
   while (!input_test.eof()) {
     input_test » out_bc1[i] ;
     i++;
   }
   ```

2. The data bits are passed to a RPW_TX function to get a output frame in the form of stream of 32 bits complex data(that is to make RPW modulation scheme).

   ```
   RPW _TX(bits_filtered11,TX_Sig_rot1);
   ```

3. The output is then printed.

   ```
   for(i=0;i<LS;i++)
   {
     cout«i«":"«TX_Sig_rot1[i]«endl;
   }
   ```

Explanation Of The Transmitter HLS source code

RPW_TX

1. `# pragma HLS INTERFACE s_axilite port=out_bc bundle=a`

   (Directive used to declare input axilite interface).

   `# pragma HLS INTERFACE s_axilite port=return bundle=a`

(Directive used to return IP controlling API’s).

```
# pragma HLS INTERFACE axis port=TX_Sig_rot1
```

(Directive for declaring stream interface).

```
# pragma HLS DATA_PACK variable=TX_Sig_rot1
```

(Directive used for concatinating real and imaginary data bits into a single data bus of 32 bits).

2. The input data of 0's and 1's is converted into -1's and 1's.

```
int i,p;
ap_int <2 >Beacon1_CRC[LBC];
ap_int <2 >Beacon_Spread[LBC*spread];
for (i=0;i<LBC;i++)
{
# pragma HLS PIPELINE
if(out_bc[i]==0)
      Beacon1_CRC[i]=-1.0;
else
      Beacon1_CRC[i]=1.0;
}
```

3. The input data(i.e -1's and 1's) is then multiplied with the chip sequence of length 64.

```
for(i=0;i<LBC;i++)
{
   for(p=0;p<64;p++)
   {
      Beacon_Spread[64*i+p]=Beacon1_CRC[i]*chipsequence[p];
   }
}
```

4. The data which is multiplied with chip sequence is then oversampled with a over-sampling factor of 4.

```
ap_int < 2 > Beacon _osf[((LBC*spread-1)*4)+1+16];
int j=0;
for(i=0;i<8;i++)
{
# pragma HLS UNROLL
Beacon_osf[i]=0;
}
for(i=8;i<=(4*(LBC*spread-1)+8);i++)
{
# pragma HLS PIPELINE
if(i%4==0) {
Beacon_osf[i]=Beacon_Spread[j];
j++;
}
else {
Beacon_osf[i]=0;
}
Beacon_osf[4*i+8]=Beacon_Spread[i];
}
```

5. The data which is obtained is then convolved with RRC filter.

```
data_out_t bits_filtered[LD];
data_out_t rrc[9];
rrc[0]=-1.6051438e-01;
rrc[1]=2.3727352e-01;
rrc[2]=8.7536235e-01;
rrc[3]=1.4742286e+00;
rrc[4]=0.8597;
rrc[5]=1.4742286e+00;
rrc[6]=8.7536235e-01;
rrc[7]=2.3727352e-01;
rrc[8]=-1.6051438e-01;
for(i=0;i<LD;i++)
{
```

```
# pragma HLS PIPELINE
# pragma HLS UNROLL factor=4
data_out_t temp=0;
for(p=0;p<=4;p++)
{
temp=temp+(data_out_t(Beacon_osf[8+i-p]+Beacon_osf[i+p])*rrc[p]);
}
bits_filtered[i]=temp;
}
```

6. The preamble data,channel estimation data and the modified data, is then made together into a frame.

```
int n=0;
int r=0;
int l=0,m=0;
for(i=0;i<Ngaurd_F;i++)
{
# pragma HLS PIPELINE
TX_Sig_rot1.write(ComplexData(0,0));
}
for(i=Ngaurd_F;i<LP+Ngaurd_F;i++)
{
# pragma HLS PIPELINE
TX_Sig_rot1.write(ComplexData(PREAMBLE[i-Ngaurd_F],PREAMBLE[i-Ngaurd_F]));
} for(i=Ngaurd_F;i<Ngaurd_CH+LP+Ngaurd_F;i++)
{
# pragma HLS PIPELINE
TX_Sig_rot1.write(ComplexData(0,0));
}
```

7. The remaining frame samples are then multiplied with the rotating sequence which rotates each data except the preamble data.

```
for(i=Ngaurd_F;i<Ngaurd_CH+LP+Ngaurd_F;i++)
{
# pragma HLS PIPELINE
TX_Sig_rot1.write(ComplexData(0,0));
n++;r=n%10;
} for(i=Ngaurd_F;i<2053+Ngaurd_CH+LP+Ngaurd_F;i++)
{
# pragma HLS PIPELINE
TX_Sig_rot1.write(BL[i-(Ngaurd_CH+LP+Ngaurd_F)]*ComplexData(rota[r]));
n++;r=n%10;
} for(i=2053+Ngaurd_F;i<LD+2053+Ngaurd_CH+LP+Ngaurd_F;i++)
{
# pragma HLS PIPELINE
TX_Sig_rot1.write(ComplexData(bits_filtered[i-
(Ngaurd_CH+LP+Ngaurd_F)+2053)],0)*ComplexData(rota[r]));
n++;r=n%10;
}
```

3.2 Receiver

The Samples are received one by one and the start of the frame is found by the correlating the received samples with the preamble sequence.Once the start of the frame is found,then the remaining length of the frame(i.e length of the frame-length of preamble) is stored in a output FIFO buffer.Then from the output FIFO buffer,the channel estimation data is stored in an array and channel is estimated.

y=h*x+w

Where,

y : received channel estimation sequence

x : transmitted channel estimation sequence

h : channel coefficients

w : Noise at receiver

The channel coefficients are estimated by dividing the received signal with the x (because the environment the modem is used is few 100 meters the channel is considered as single-tap wireless channel,w is compensated when we use averaging of channel coefficients).

We get the estimate as follows:

h=(y/x) + (w/x)

From the estimated channel,the data is equalized and then convolved with the RRC filter to find the received data.

Also,the start of the frame is found out by correlating the received samples with the preamble sequence.The cross correlation values are then compared with auto correlation values returned by norm function using stored preamble and received data.The cross and auto correlation value are stored in an array.We find the start of the frame if the value of cross correlation is greater than a product of auto correlation and threshold value,also cross correlation of current value must be greater than all other calculated cross correlation values that found before and after reading the input data samples(we check for few data samples window,here we check if the cross correlation of the read data samples is greater than cross correlation of 9 data samples 4 before and 4 after that input data sample).

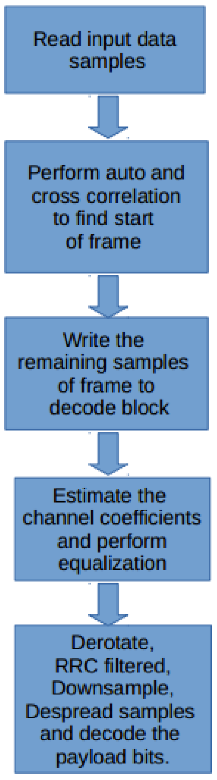

Read input data samples
Perform auto and cross correlation to find start of frame
Write the remaining samples of frame to decode block
Estimate the channel coefficients and perform equalization
Derotate, RRC filtered, Downsample, Despread samples and decode the payload bits.

Figure 3.2: Receiver Flow Chart

3.2.1 Receiver Code Explanation

Receiver Test Bench Explanation

1. The received data is read from a file and then stored in a input FIFO buffer.

```
ifstream input(“rx_in.txt”);
myStream1 input_data(“input_data”), out_data(“out_data”);
data_t8 inp;
data_type8 real_data, imag_data;
ap_int <2 >frame_dcd[64];
ap_int <2 >crc_bits[64];
char comma;
int count = 0, npkts = 0,i=0;
while (!input.eof()) {
   i++;
   input » real_data » comma » imag_data;
   input_data.write(data_t(real_data, imag_data));
}
```

2. The input data is send one by one into the “pckdect” function which detects the start of the frame and then after detecting the start of the frame it pushes the remaining data of the frame(i.e length of the frame-length of the preamble) into an output FIFO buffer.(npts is the number of frames detected in the input data)

```
while (!input_data.empty()) {
   if (pckdect(input_data,out_data)) {
      npkts++;
   }
}
```

3. The output FIFO buffer is then given as an input to the function called “decode”, which takes the remaining data(i.e length of the frame - length of the preamble) from the output buffer and then decode the bits.

```
for(i=0;i<npkts;i++)
{
    decode(out_data,frame_dcd,crc_bits);
}
```

Explanation Of The Functions

pckdect

1. The function "pckdect" accept the input from the FIFO buffer, sample by sample and pushes the data into a shift register called data_store which is of length 262(i.e length of preamble).

   ```
   input_data.read(inp);
   for(i=0;i<LP-1;i++)
   {
   # pragma HLS PIPELINE rewind
   # pragma HLS UNROLL
       data_store[i]=data_store[i+1];
   }
   data_store[LP-1]=inp;
   ```

2. The function pckdect contains three shift register called "val_arr ","num" and prevdata which are of length 9. The correlation between data_store and stored preamble sequence is done.The cross correlation value is then stored in num the auto correlation is stored in val_arr by the norm function.The data_store is shifted by one.

   ```
   norm(data_store,& mag1,& mag_data1);
   for(i=0;i<9-1;i++)
   {
       val_arr[i]=val_arr[i+1];
       prevdata[i]=prevdata[i+1];
   }
   val_arr[W-1]=mag_data1;
   num[W-1]=mag1;
   prevdata[W-1]=inp;
   ```

3. If the the value of the 5th element in val_arr(i.e the middle element in the val_arr) is greater than the remaining elements in the array, val_arr and is also greater than product of threshold and autocorrelation, then the point corresponding to the correlation value of 6th element in the array,val_arr is considered to be the start of the frame(i.e end of the preamble of a new frame) .After detecting the start of the frame,the remaining elements of the frame(i.e length of frame-length of preamble) is pushed into an output FIFO buffer,the the shift registers are cleared.

```
if(num[I]>threshold*val_arr[I] && val_arr[I]>0.125)
{
   flag1=true;
   for(i=0;i<9;i++)
   {
      if(num[5]*val_arr[i]<num[i]*val_arr[5])
      {
         flag1=false;
      }
   }
   if(flag1==true)
   {
      packet=true;
      for(i=0;i<5;i++)
      {
# pragma HLS UNROLL
         out_data.write(prevdata[i+5],0);
      }
      for(i=I;i<17776-LP-40;i++)
      {
# pragma HLS PIPELINE
         input_data.read(inp);
```

```
            out_data.write(inp);
        }
        for(i=0;i<W;i++)
        {
# pragma HLS UNROLL
            val_arr[i]=data_type5(0);
            prevdata[i]=data_t8(0,0);
            num[i]=data_typem(0);
        }
        for(i=0;i<LP;i++)
        {
# pragma HLS PIPELINE rewind
                # pragma HLS UNROLL
                data_store[i]=data_t8(0,0);
        }
        }
        }
return packet;
}
```

decode

1. The decode function takes the output FIFO buffer from the “pckdect” function as the input and reads the channel estimation data from the buffer, divides it with input channel estimation data and stores the result in an array.

   ```
   for(i=0;i<Ngaurd_CH;i++){
   # pragma HLS PIPELINE
   input_data.read(inp);
   }
   ```

2. The middle 780 data samples of the channel_est array is considered,they are derotated and divided by the transmitted data samples(to perform zero forcing), data samples of index 120,130,..170 and 220,230....270 and this sequence of indexes upto 870 is considered and amean values of this 48 samples is assigned as channel_mean_final[0].Simillarly 121,131...271 and this sequence of corresponding indexes mean values are assigned accordingly to the channel_mean_final array to form overall 10 channel coefficients.They are used for channel equilization.

```
decode_label1:for(i=0;i<LC;i++){
# pragma HLS PIPELINE
input_data.read(inp);
if(i>110&&i<890)
channel_est[i]=inp/(channel_rrc[i]*rota[r]);
n++;r=n%10;
}
const data_chnl c3=data_chnl(0.0208333,0);
for(p=0;p<10;p++)
{
# pragma HLS PIPELINE
temp=data_chnl(0,0);
for(i=100;i<900;i=i+100)
  {
     for(j=20;j<80;j=j+10)
        {
        temp=temp+(channel_est[i+j+p]);
         }
  }
channel_mean_final[p]=temp*(c3);
}
```

3. The next input stream data of 36 gaurd bits are read. for(i=0;i<Ngaurd_CH2;i++)

```
{
# pragma HLS PIPELINE
input_data.read(inp);
}
for(j=0;j<=7;j++) {
# pragma HLS PIPELINE
data1[j]=data_dt1(0,0);
}
```

4. The remaining data is read from the buffer, divided with the estimated channel,also with rotation sequence and then stored in an array named data1.

```
j=0;
for(i=8;i<LD+2053+8;i=i+10) {
# pragma HLS PIPELINE
for(j=0;j<10;j++) {
input_data.read(inp);
data1[i+j]=(data_chnl(inp)/(channel_mean_final[j]*data_chnl(rota[j])));
}
}
for(j=LD+2053+8;j<LD+2053+16;j++) {
# pragma HLS PIPELINE
data1[j]=data_dt1(0,0);
}
```

5. The data in the array,“data1” is then convolved with RRC filter and then stored in an array called data1_rrc.The data1 array is also decimated by a factor of 4 and assigned to final_unspread array to compensate for the oversampling factor that has been introduced during Transmission of data samples.This process of Convolution and Undersampling is done in the same loop inorder to reduce the required computions (by reducing multiplications) and also B-RAM memory used(by removing arrays).

```
data_typedt1 rrc[5];
# pragma HLS ARRAY_PARTITION variable=rrc complete dim=1 rrc[0]=-1.6051438e-01;
rrc[1]=2.3727352e-0;
rrc[2]=8.7536235e-01;
```

```
rrc[3]=1.4742286e+00;
rrc[4]=0.8597;
data_typep temr;
int h=0;
for(i=8;i<2053;i=i+4) {
# pragma HLS PIPELINE
# pragma HLS UNROLL factor=4
temr=data_typep(0);
for(p=0;p<=4;p=p+1) {
temr=temr+data_typep((real(data1[i+8-p])+real(data1[i+p]))*(rrc[p]));
}
final_unspread[h]=data_fnsd(temr);
h=h+1;
}
h=512;
for(i=2061;i<LD+2053+8-8;i=i+4) {
# pragma HLS PIPELINE
# pragma HLS UNROLL factor=4
temr=data_typep(0);
for(p=0;p<=4;p=p+1) {
temr=temr+data_typep((real(data1[i+8-p])+real(data1[i+p]))*(rrc[p]));
}
final_unspread[h]=data_fnsd(temr);
h=h+1; }
```

6. The final unspread data is then multiplied with the chipsequence , stored in an array called temp_rx and then every 64 bits of the temp_rx are added to check the real part of the summation. If the real part of the summation is greater than 0,then the bit is decoded as 1 otherwise 0.

```
for(i=0;i<Nbits;i++) {
# pragma HLS PIPELINE
# pragma HLS UNROLL factor=2
pre=data_typep(0);
p=spread*i;
for(j=0;j<64;j++) {
pre=pre+data_typep((final_unspread[p+j])*data_fnsd(chipsequence[j]));    }
if (pre>0) {
    frame_dcd[i]=1;
    crc_bits[i]=1;
  }
  else {
    frame_dcd[i]=-1;
    crc_bits[i]=0;
  }
}
```

3.3 Optimization of codes

Directives present in the VIVADO HLS can be used for the optimization of the code.

3.3.1 Transmitter

In transmitter three major changes have been made for improving latency and Resources used :

1. All the arrays that are used are optimized by declaring them with lower bits(below 16bits).The output stream samples are also made 16-bits as AD9361 can only accept atmost 16bits data as input through one channel.
2. The rotation sequence is pre-generated with a fixed rotation frequency and the samples are stored in an array.So,this eliminated the need for trigonometric functions to be included in our TX.So,as a result the need for generation of cos and sin data is reduced which in turn improved latency and DSP48 resources.We can further make rotation as variable instead of fixed by using CORDIC algorithm for cos ans sine generation.

3. The loop which is used to RRC filter the data samples has been Unrolled by a factor of 4 and the array used is partition is made to improve access to the data array,which has reduced the latency of the code but at the cost of Resources(because of array partition the B-RAM resources are been utilized more also due to loop unrolling more multiplication DSP48 block has been used).

3.3.2 Packet Detector

In Packet Detector algorithm has been modified significantly inorder to reduce its occupency of resources and latency.

1. The PN Sequnce has been changed for 256 length to 64 length (as the number of multiplications that are needed are reduced from 262 to 68 during calculation of under-sampled correlation values).

2. The shift register which are used for storing the cross and auto correlation values are been reduced both in length and also in data width as they are been completely partitioned to get improve latency(by reducing array access time).

3. The loop where the auto and cross correlations are done merged and also the data samples are typecasted to 10 bits data sample inorder to perform lower order correlation thereby reducing the DSP48 and also LUT's required.

4. The data_store array has been array partitioned(cyclic) as they are accessed for computing correlation values.

5. The square root of normalization of cross correlation value with auto correlation value is removed as the square root operation and the division operations are using more DSP48 blocks and increasing the loop latency.Instead of root-normalization operations,the comparision of cross correlation values with the auto correlation is performed,this may have increased some DSP48 and LUT elements but the overall Latency has been reduce significantly.

3.3.3 Receiver

In receiver,three ”for” loops have been optimized for efficiency are :

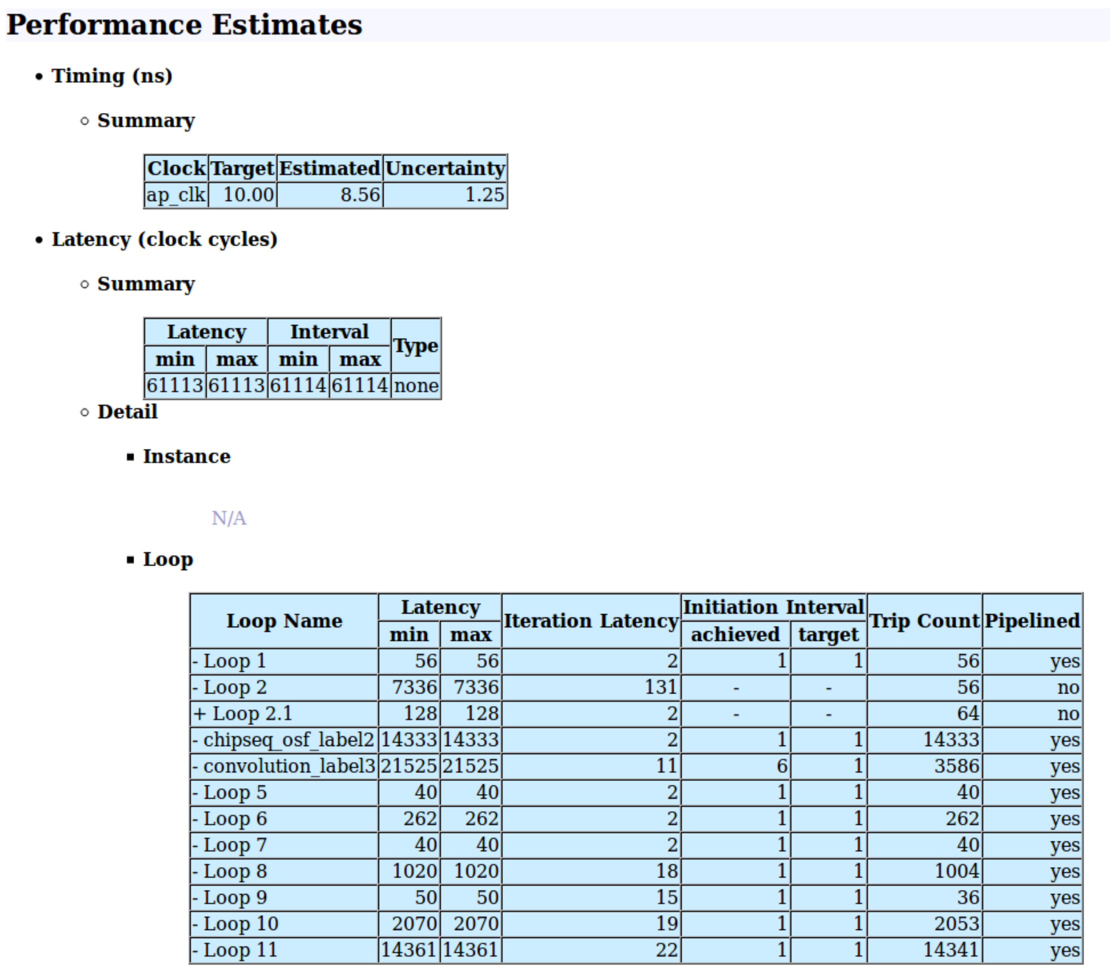
1. The loop for channel estimation are been pipelined to improve latency.
2. The channel_final_ estimation array is partitioned completely to reduce array access time,this improve the latency of the Block.
3. The loop which is used to RRC filter the data samples has been Unrolled by a factor of 4 and the array used is partition is made to improve access to the data array,which has reduced the latency of the code but at the cost of Resources(because of array partition the B-RAM resources are been utilized more also due to loop unrolling more multiplication DSP48 block has been used).
4. The undersampling and the RRC filtering is done in the same loop.This reduces use of additional multiplication.BRAM memory and thus improving both latency and hardware used.

3.4 HLS reports

3.4.1 Transmitter Report

The transmitter IP has an latency of 61113 and Initiation Interval(II) of 61114.By definition of latency,61113 many clock cycle are required to generate the one frame(complete all its operations) and also 61114 cycle to accept the next payload data.The estimated timing clock is 8.56 ns and its meets the target clock.The utilization resources report is shown in the below figure and the transmitter block can be easily fitted onto the Programmable Logic of ZC702.

Performance Estimates



Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.56	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
61113	61113	61114	61114	none

  - **Detail**
    - **Instance**

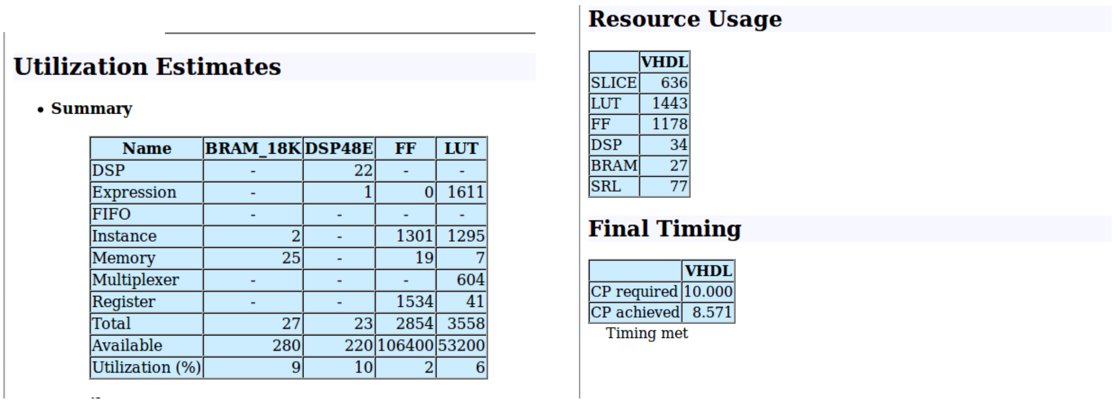
      N/A

    - **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Loop 1	56	56	2	1	1	56	yes
- Loop 2	7336	7336	131	-	-	56	no
+ Loop 2.1	128	128	2	-	-	64	no
- chipseq_osf_label2	14333	14333	2	1	1	14333	yes
- convolution_label3	21525	21525	11	6	1	3586	yes
- Loop 5	40	40	2	1	1	40	yes
- Loop 6	262	262	2	1	1	262	yes
- Loop 7	40	40	2	1	1	40	yes
- Loop 8	1020	1020	18	1	1	1004	yes
- Loop 9	50	50	15	1	1	36	yes
- Loop 10	2070	2070	19	1	1	2053	yes
- Loop 11	14361	14361	22	1	1	14341	yes

Figure 3.3: Transmitter Performance Estimates

Utilization Estimates and Export Report for Transmitter



Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	22	-	-
Expression	-	1	0	1611
FIFO	-	-	-	-
Instance	2	-	1301	1295
Memory	25	-	19	7
Multiplexer	-	-	-	604
Register	-	-	1534	41
Total	27	23	2854	3558
Available	280	220	106400	53200
Utilization (%)	9	10	2	6

Resource Usage

	VHDL
SLICE	636
LUT	1443
FF	1178
DSP	34
BRAM	27
SRL	77

Final Timing

	VHDL
CP required	10.000
CP achieved	8.571

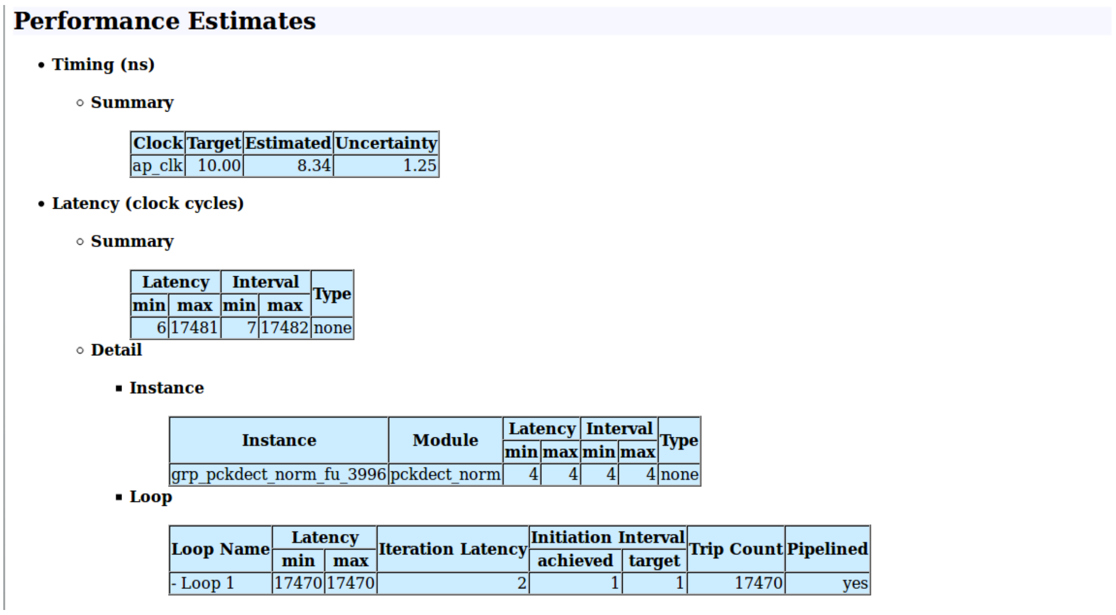
Timing met

Figure 3.4: Transmitter utilization estimates and Export report

3.4.2 Packet detector Report

The Packet detector IP has an Initiation Interval(II) of 7.The packet detector IP has to be in auto-restart mode,as it should accept the samples from the ADC of AD9361 and the samples should be time synchronized and the valid data samples has to be given to the next block serially through stream interface.The samples are read at the rate of 7 clock cycles(II) and the samples to the next block is written at the rate of 21 clock cycle(Latency) this can be observed from the timing diagram.The estimated timing clock is 8.34 ns and its meets the target clock.The utilization resources report is shown in the below figure and the transmitter block can be easily fitted onto the Programmable Logic of ZC702.

Performance Estimates



Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.34	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
6	17481	7	17482	none

  - **Detail**
    - **Instance**

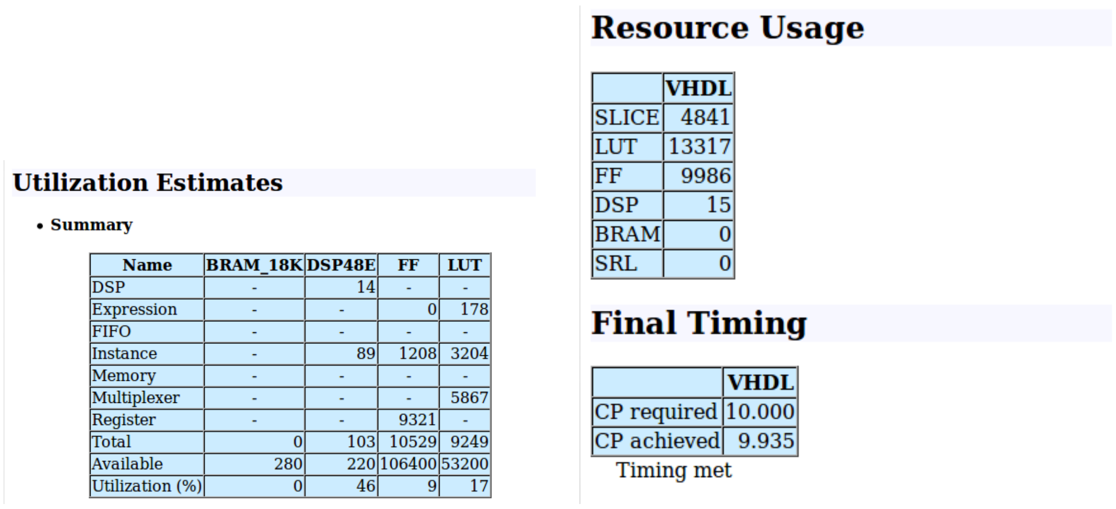
Instance	Module	Latency		Interval		Type
		min	max	min	max	
grp_pckdect_norm_fu_3996	pckdect_norm	4	4	4	4	none

    - **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Loop 1	17470	17470	2	1	1	17470	yes

Figure 3.5: Packet Detector Performance Estimates

Utilization Estimates and Export Report for Packet detector



Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	14	-	-
Expression	-	-	0	178
FIFO	-	-	-	-
Instance	-	89	1208	3204
Memory	-	-	-	-
Multiplexer	-	-	-	5867
Register	-	-	9321	-
Total	0	103	10529	9249
Available	280	220	106400	53200
Utilization (%)	0	46	9	17

Resource Usage

	VHDL
SLICE	4841
LUT	13317
FF	9986
DSP	15
BRAM	0
SRL	0

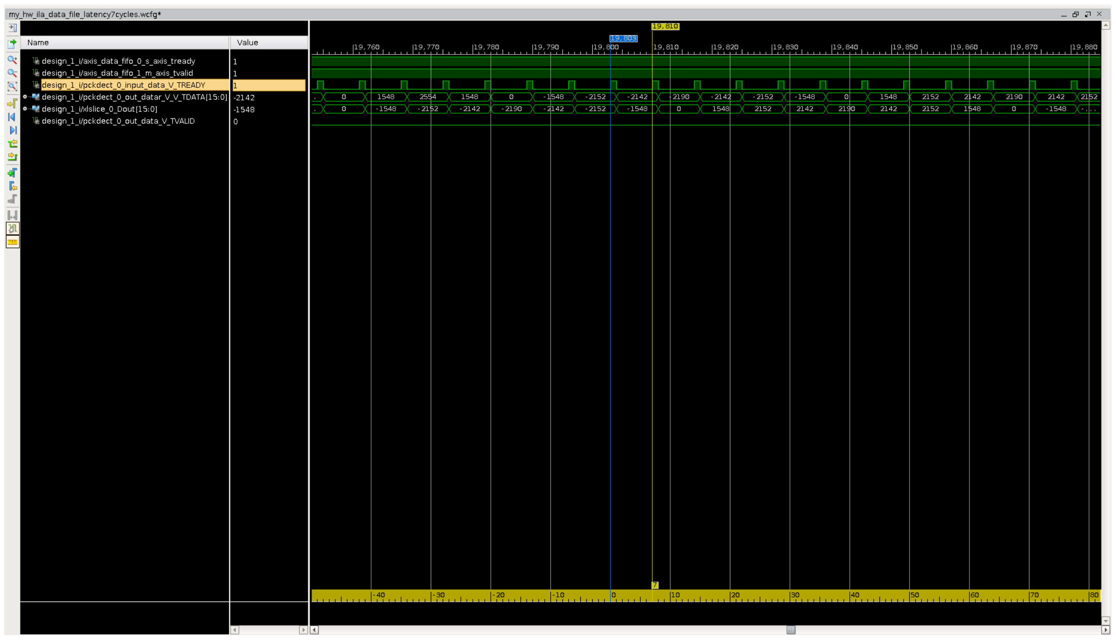
Final Timing

	VHDL
CP required	10.000
CP achieved	9.935

Timing met

Figure 3.6: Packet Detector Utilization Estimates and Export Report

A timing diagram of the stream interface observed using vivado using ILA(Integrated Logic analyzer) is also attached.The input_data_V_Tready signal gives clear info on the latency and Initiation Interval of the packet detector.This signal gives info on when the stream interface is freed inorder to write or read a new samples on to stream interface.The new sample is accepted for every 7 clock cycle,and the sample that is read from the input stream is written to output stream after 21 clock cycles.



my_hw_ila_data_file_latency7cycles.wcfg*

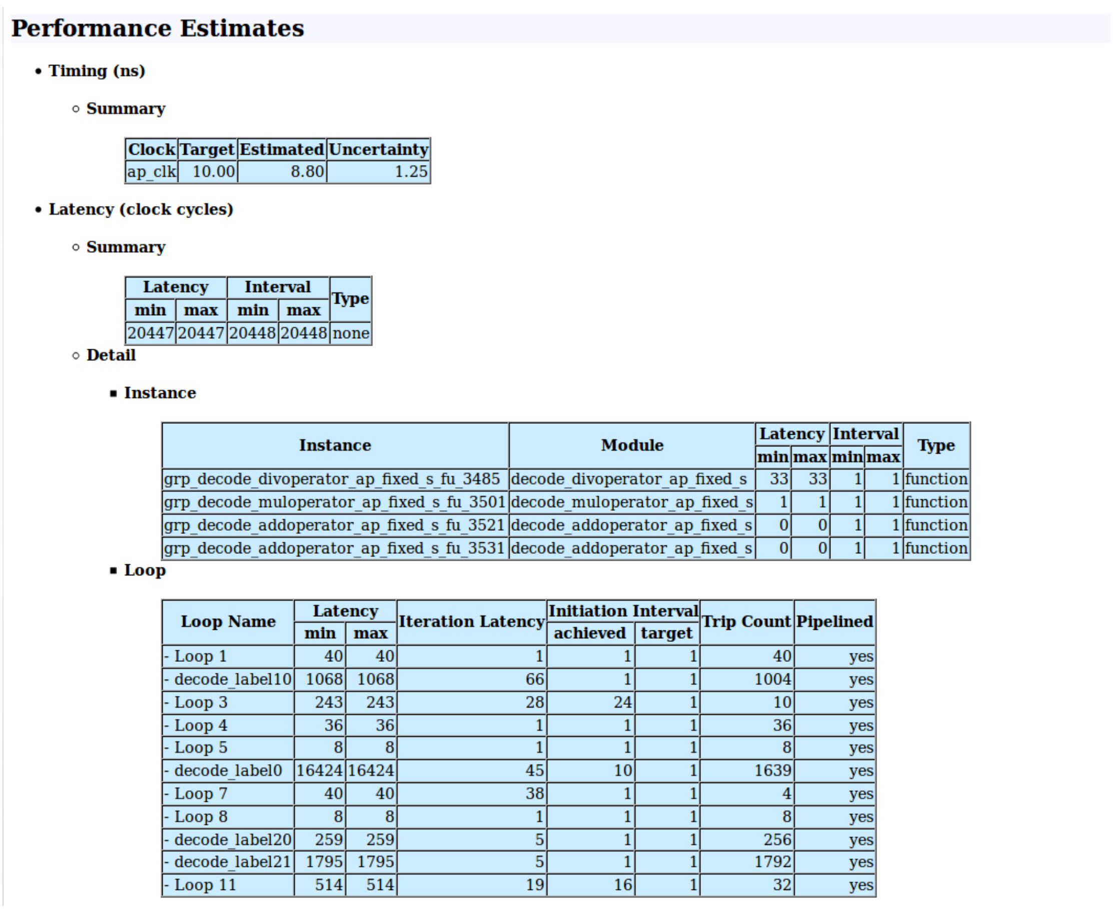
Name	Value
design_1_i/axis_data_fifo_0_s_axis_tready	1
design_1_i/axis_data_fifo_1_m_axis_tvalid	1
design_1_i/pckdect_0_input_data_V_TREADY	1
design_1_i/pckdect_0_out_datar_V_V_TDATA[15:0]	-2142
design_1_i/xlslice_0_Dout[15:0]	-1548
design_1_i/pckdect_0_out_data_V_TVALID	0

Figure 3.7: ILA Waveform

3.4.3 Decode Report

The decode IP has an latency of 20447 and Initiation Interval(II) of 20448.By definition of latency,20447 many clock cycle are required to process one frame and also one extra cycle to decode the frame payload data.The estimated timing clock is 8.80 ns and its meets the target clock.The utilization resources report is shown in the below figure and the decoder block can be fitted onto the Programmable Logic.

Performance Estimates



Performance Estimates

- **Timing (ns)**
  - **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.80	1.25

- **Latency (clock cycles)**
  - **Summary**

Latency		Interval		Type
min	max	min	max	
20447	20447	20448	20448	none

  - **Detail**
    - **Instance**

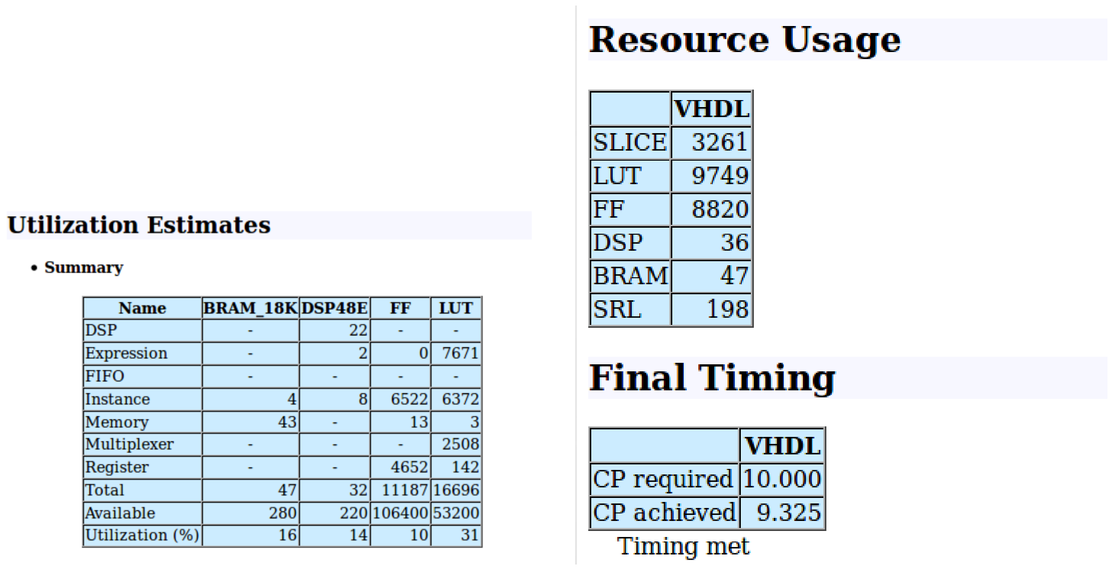
Instance	Module	Latency		Interval		Type
		min	max	min	max	
grp_decode_divoperator_ap_fixed_s_fu_3485	decode_divoperator_ap_fixed_s	33	33	1	1	function
grp_decode_muloperator_ap_fixed_s_fu_3501	decode_muloperator_ap_fixed_s	1	1	1	1	function
grp_decode_addoperator_ap_fixed_s_fu_3521	decode_addoperator_ap_fixed_s	0	0	1	1	function
grp_decode_addoperator_ap_fixed_s_fu_3531	decode_addoperator_ap_fixed_s	0	0	1	1	function

    - **Loop**

Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined
	min	max		achieved	target		
- Loop 1	40	40	1	1	1	40	yes
- decode_label10	1068	1068	66	1	1	1004	yes
- Loop 3	243	243	28	24	1	10	yes
- Loop 4	36	36	1	1	1	36	yes
- Loop 5	8	8	1	1	1	8	yes
- decode_label0	16424	16424	45	10	1	1639	yes
- Loop 7	40	40	38	1	1	4	yes
- Loop 8	8	8	1	1	1	8	yes
- decode_label20	259	259	5	1	1	256	yes
- decode_label21	1795	1795	5	1	1	1792	yes
- Loop 11	514	514	19	16	1	32	yes

Figure 3.8: Decode Performance Estimates

Utilization Estimates and Export Report for Receiver



Utilization Estimates

- **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	22	-	-
Expression	-	2	0	7671
FIFO	-	-	-	-
Instance	4	8	6522	6372
Memory	43	-	13	3
Multiplexer	-	-	-	2508
Register	-	-	4652	142
Total	47	32	11187	16696
Available	280	220	106400	53200
Utilization (%)	16	14	10	31

Resource Usage

	VHDL
SLICE	3261
LUT	9749
FF	8820
DSP	36
BRAM	47
SRL	198

Final Timing

	VHDL
CP required	10.000
CP achieved	9.325

Timing met

Figure 3.9: Receiver Utilization Estimates and Export Report

CHAPTER 4

Vivado IP Connections,Reports and SDK code:

4.1 Vivado Block Diagram and Connections:

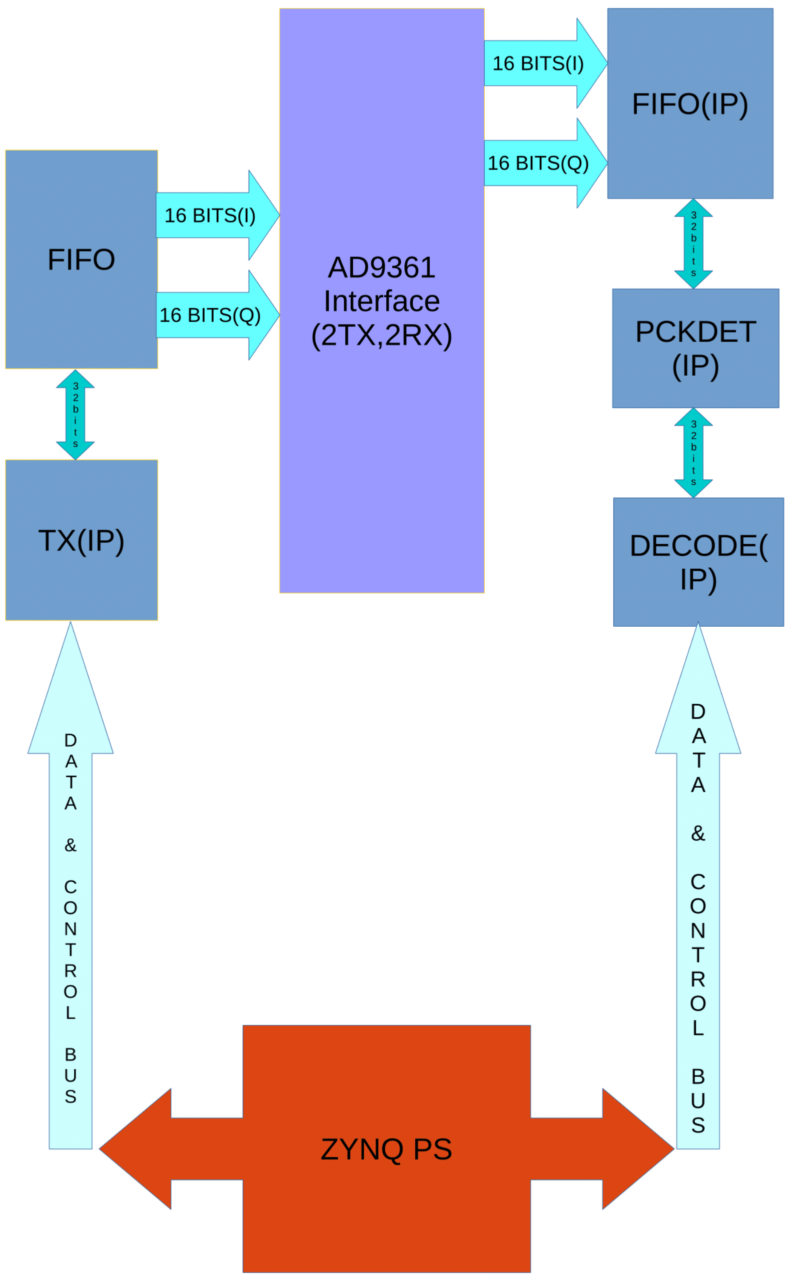


Figure 4.1: Vivado Block Diagram

4.1.1 Transmitter Connections:

The transmitter IP connections are connected as shown below.The Transmitter IP writes the data to the FIFO generator IP.The FIFO generator IP accepts the data until the FIFO is not empty(this is achieved by connecting negation of full signal to the stream Ready signal).The AD9361 DAC reads data from the FIFO(the FIFO read enable is connected to DAC valid signal).The FIFO write clock is from TX IP and read clock is from AD9361 clock.

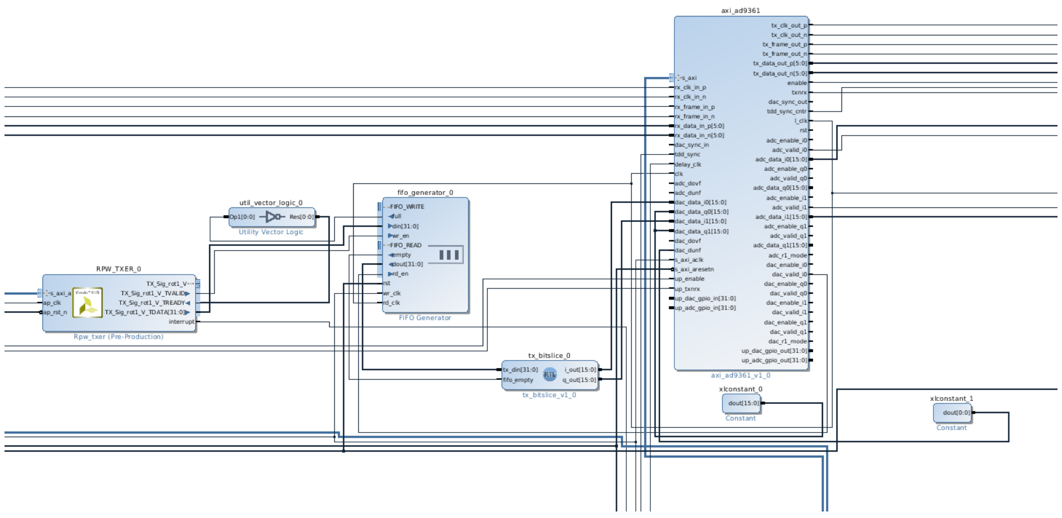


Figure 4.2: Transmitter connections

4.1.2 Receiver Connections:

The packet detector and receiver IP connections are connected as shown below.The AD9361 interface IP writes the data to the FIFO generator IP.The FIFO generator IP accepts the data only when the ADC gives valid data (this is achieved by connecting adc valid of AD9361 to the FIFO write enable).The AD9361 DAC writes data to the FIFO.The FIFO read clock is from RX IP and write clock is from AD9361 clock.The interface between packet detector and RX IP is stream interface.

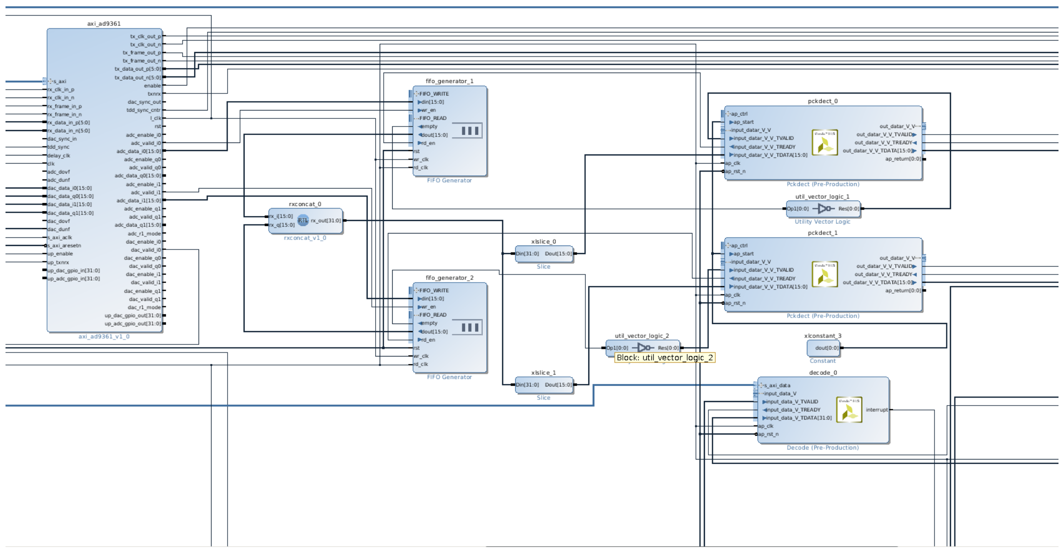
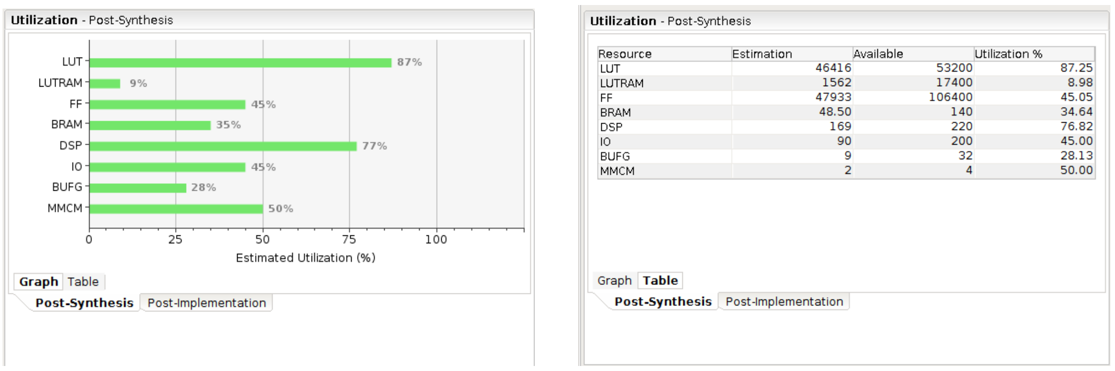


Figure 4.3: Receiver connections

4.2 Vivado Utilization Resource:

The final resources that are used after synthesis and implementation is as follows:

4.2.1 Post Synthesis:

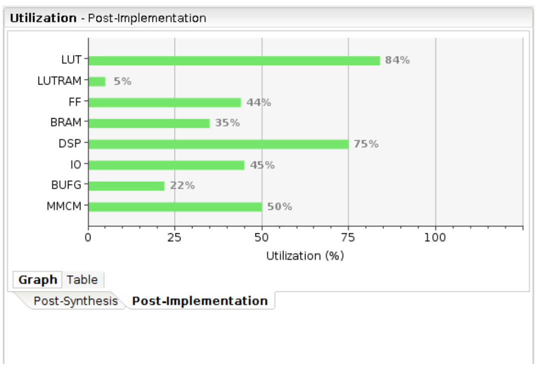


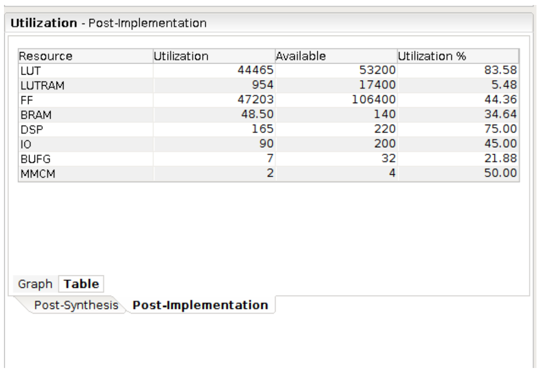
Utilization - Post-Synthesis

Resource	Estimation	Available	Utilization %
LUT	46416	53200	87.25
LUTRAM	1562	17400	8.98
FF	47933	106400	45.05
BRAM	48.50	140	34.64
DSP	169	220	76.82
IO	90	200	45.00
BUFG	9	32	28.13
MMCM	2	4	50.00

Figure 4.4: Vivado Post Synthesis Reports

4.2.2 Post Implementation:




Utilization - Post-Implementation

Resource	Utilization	Available	Utilization %
LUT	44465	53200	83.58
LUTRAM	954	17400	5.48
FF	47203	106400	44.36
BRAM	48.50	140	34.64
DSP	165	220	75.00
IO	90	200	45.00
BUFG	7	32	21.88
MMCM	2	4	50.00

Graph Table
Post-Synthesis Post-Implementation

Figure 4.5: Vivado Post Implementation Reports

4.3 SDK

The Vivado design suite generates a bitstrem file.This is exported into the Xilinx Software Development Kit for programming the ZC702 platform through Uart cable.The SDK toolkit also provide a serial terminal console to observe the data that is returned by the decode ip through USB-JTAG cable.Through SDK the AD9361 parameters for transmission and reception can also be programmed.The SDK tool is also used to pass the data(that needs to be RPW modulated)from ZYNQ to transmitter IP by using axilite interface.(During the final stage of the project the data is not given SDK,but passed from ethernet to transmitter IP through ZYNQ processing System).The continuous transmission of data bits through TX ip and decoding data bits through decode IP using SDK code is given as follows:

4.3.1 SDK codes

1. The ZC702 and AD9361 are initializated.(AD9361 parameter are also configured here).

   ```
   int main() {
   init_platform();
   ad9361_main();
   ```

2. The data for transmitter IP is declared here.

```
int status,sumh,i;
char input_data[56]= {0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,
0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,0,0,0 };
int f[64];
int *fi=(int *)(f);
```

3. The Transmitter IP is initialized and auto-restart is enable here.

```
XRpw_txer doRpw;
XRpw_txer *txptr;
txptr= & doRpw;
XRpw_txer_Config *doRpw_cfg;
status=XRpw_txer_Initialize(txptr,0);
XRpw_txer_EnableAutoRestart(txptr);
```

4. The Decode IP is initialized and auto-restart is enable here.

```
XDecode xdecode;
XDecode* xdecptr =& xdecode;
XDecode_Initialize(xdecptr, 0);
XDecode_EnableAutoRestart(xdecptr);
```

5. The data bits are passed to Transmitter IP.

```
XRpw_txer_Write_out_bc_V_Bytes(txptr,0,input_data,56);
```

6. The Transmitter and Decode IP's are started here.

```
XRpw_txer_Start(txptr);
XDecode_Start(xdecptr);
```

7. We wait for completion of decode IP and the data from decode ip is read through axilite.(Then we clean up the platform and exit the code.)

```
while(1){
```

```
while(!(XDecode_IsDone(xdecptr)));
sumh=XDecode_Read_crc_bits_Words(xdecptr, 0, fi,64);
}
cleanup_platform();
return 0;
}
```

CHAPTER 5

Hardware Implementation Setup and Frequency offset

5.1 Implementation Setup

The following experimental setup as shown in figure is been established.The antenna used is a designed to operate at 860 MHz,it has two dipole antenna that are oriented perpendicular to each other in order to implement the RPW modem.One dipole of the antenna is excited with the TX1 port's signal from the AD9361,the other dipole of the same antenna is excited by TX2 port's signal.At,the receiver end the RX1 port's signal is fed from one of the dipole antennas and the other dipole of same receiver antenna feeds the RX2 port of the AD9361 RF board. The AD9361 boards are connected to the ZC702 platform through fmcomm2 ports of ZC702.The ZC702 boards are programmed through USB-JTAG cable,also the they are monitored through USB-UART cables.

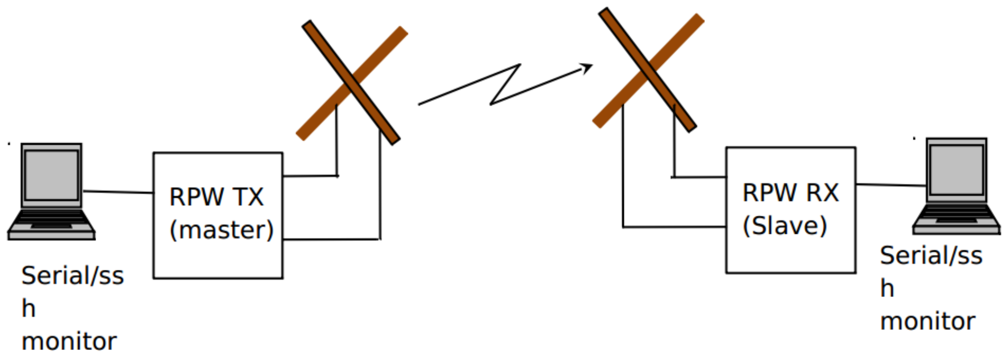


Figure 5.1: Experimental setup with 1 TX and 1 RX RPW nodes with same LO in LOS

The same experiment is repeated by using 1 RX and 2 TX RPW nodes in NLOS with same LO.

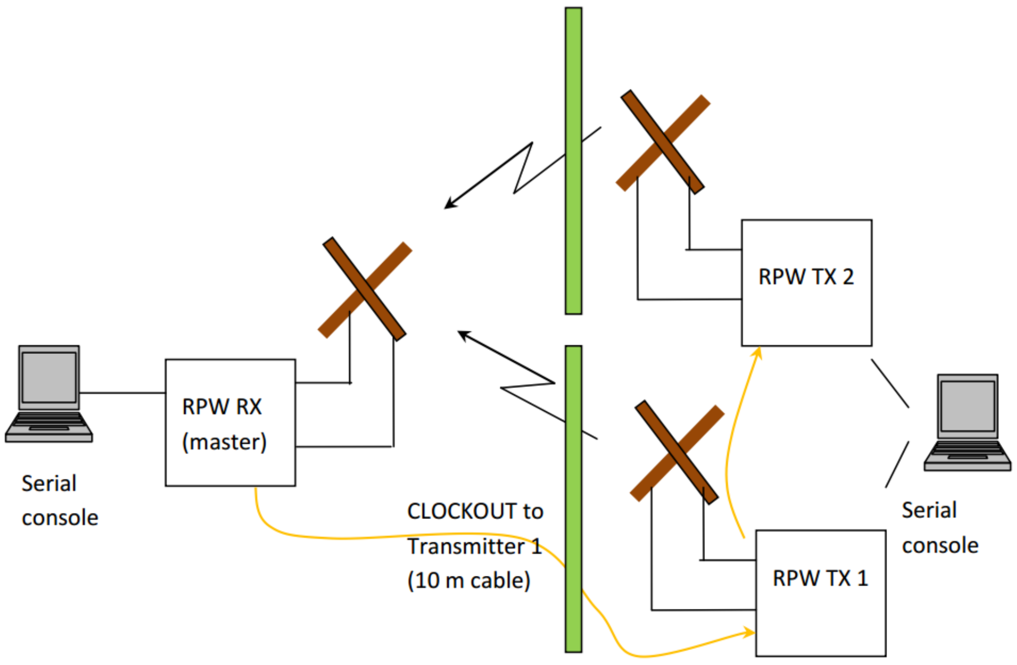


Figure 5.2: Experimental setup with 2 TX and 1 RX RPW nodes with same LO in NLOS

The AD9361 clock-out pin is connected to the clock-in pin of other AD9361,this is to have same clock for TX and RX nodes.

5.2 Frequency offset

Through the project the TX and RX are been frequency synchronized by driving the RF component LO(local oscillator) with the same clock.When the TX and RX are driven by two different LO sources there is a frequency offset between the two LO sources which lead to the problem of frequency offset.This offset has to be estimated and corrected accordingly to decode the bits correctly.

5.2.1 CORDIC

CORDIC(coordinate rotation digital computer) is a hardware-efficient iterative method which uses rotations to calculate a wide range of elementary functions.The fundamental advantage of the CORDIC algorithm is that it only requires addition/subtraction, table look up and bit shift. This makes it particularly suitable for implementation on FPGA's, where the multipliers are a scarce resource. CORDIC can be either used in

1. **Rotation Mode:** The rotation of an input vector by a given angle α .Used in offset correction.

2. **Vectoring Mode:** To calculate inverse tangents(Phase values) of rotation of an input vector V.Used in offset estimation.

The CORDIC algorithm is used because of its low latency and low number of resource usage with accurate output values.

5.2.2 Frequency offset estimation

We transmit a sequence of known single tone complex exponential frequency signal.Then we calculated the frequency of the received signal and the difference between the frequency is the offset that needs to be compensated.The frequency offset is estimated by using two algorithms:

1. CORDIC algorithm is used for calculating the Phase of the received complex signal.(from this frequency is calculated)

   The cordic algorithm initially take the vector (1,0) as input and rotates through 45 degree angle(as argtan(1/2)=45).The cordic algorithm rotates the angle iteratively in step of
   $$argtan(2^{-n})$$
   The vector is rotated with respective to the desired angle we want to achieve,some iterations rotate clockwise direction and some rotates in anticlockwise direction to converge to the desired angle.A look up table for
   $$argtan(2^{-n})$$
   is stored.The K value is fixed based the number of the iteration loops used in the cordic algorithm.

2. The calculated Phase values are fitted to a straight line through Simple linear regression.The slope of this straight line gives the frequency of received signal.A

straight line with slope and intercept is considered and the differentiation w.r.t slope and intercept is performed.

$$\frac{\partial}{\partial m}\sum_{n=1}^{N}(y_n - mx_n - c)^2$$

$$\frac{\partial}{\partial c}\sum_{n=1}^{N}(y_n - mx_n - c)^2$$

The y axis data is phase angle after appropriate compensation from cordic(as the phase is returned in piece-wise continuous form and has to be modified before fitting to a straight line) and x axis is linear scale from 1 to N,the resultant slope after simplification reduces to

$$m = \frac{\sum_{n=1}^{N} 6 * ((2 * n) - N - 1) * (phasevalue_n)}{N^3}$$

The estimated frequency is then given as:

$$Estimatedfrequency = \frac{\sum_{n=1}^{N} 6 * ((2 * n) - N - 1) * (phasevalueatn)}{2 * \pi * 10^{-7} * N^3}$$

Code for Frequency Estimation

1. The read received stream data of cos and sin samples is given as input to CORDIC function.The returned phase value from the CORDIC function is stored in "now" variable and the now values is stored in "prev" variable.

   ```
   prev=now;
   now=angle1;
   int k=0,mid=0,mid1=0;
   ```

2. The phase value of continuous 3000 samples is considered and the piece-wise linear phases returned by CORDIC function is made into continuously varying linear phase by adding the 2* π for every π variations.The compensated phase values are also summed up (with appropriate scaling) in order to find the slope of line(that is fitted to the phase values).

   ```
   if((j>=201 & & j<=3200)) {
   mid=prev-now;mid1=now-prev;
   ```

```
if(mid>51470)k=((ap_int<48>(mid)*83449)»32);now=now+k*51470+51470;
if(mid1>51470)k=((ap_int<48>(mid1)*83449)»32);now=now-k*51470-51470;
accu=accu+now*coeff;
coeff=coeff+2;
}
```

3. The value of the accu variable is scaled by

$$\frac{6}{2 * 3000^3 * 10^{-7}}$$

to get the estimated frequency scaled by π value.In CORDIC π is equivalent to 51470 (integer part of $\pi * 2^{14}$),so the actual sent frequency 1 MHz multiplied with π is equal to 51470000000.When we send positive exponential value the offset is calculated by subtracting the value 51470000000,this result is divide by π(51470) and is stored in slope[0].Similarly if negative exponential value is sent the offset is calculated by adding 51470000000 to the temp variable and divide by 51470 to get the offset value.The division by 51470 and scaling factor of accu consume many DSPs and also has high Initiation Interval and latency values.So,we do the multiplication of 83449 and 4772186 followed by 32 bit shift to perform the division operations (the multiplying numbers are selected accordingly for an appropriate precision).

```
j++;ap_int<64> temp=0;
if(j==3250)
{ temp=((accu*4772186)»32);
temp=temp+(51470000000);
slope[0]=(((temp)*83449)»32);
temp=temp-(102940000000);
slope[1]=(((temp)*83449)»32);
}
```

4. Finally after reading the current frame samples,the variable are reset for estimating the offset for next frame. if(j==18478)

```
{ j=0;temp=0;accu=0;now=0;prev=0;slope[0]=0;slope[1]=0;coeff=-2999;
}
```

CHAPTER 6

Future Work

The RPW Nodes having two different LO has to be implemented.The offset between the two LO's has to be removed accordingly inorder to decode the payload data correctly as in the case of same LO.The problem of the IQ imbalance and DC offset is removed by using the quadrature correction and DC offset correction,the results were not as expected,a baseband algorithm for compensated this IQ,DC offset and frequency correction has to be implemented in-order to decode the payload data.Some of the antenna are not having maximum power transfer at 860 MHz frequency,with the TX spectrum power at -20db and there is an addition loss during transmission and reception(due to antennas),signal reception is effected as the AD9361 has maximum of 70db gain in receiver port.The high gain power amplifier and Low noise amplifier board that is provided by Hitachi has to be integrated in order to improve the reception range.

REFERENCES

1. Xilinx, Vivado Design Suite User Guide High-Level Synthesis (UG902). Xilinx(v2016.2) User Guide, 2016

2. Xilinx, Vivado Design Suite User Guide Implementation (UG904). Xilinx (v2016.2)

3. Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973).User Guide, 2016

4. FIFO Generator v13.1 LogiCORE IP Product Guide (PG057) - Xilinx

5. LogiCORE IP Constant v1.1 Product Brief (PB040) - Xilinx

6. LogiCORE IP Slice v1.0 Product Brief (PB042) - Xilinx

7. AD9361 Reference Manual (Rev. A) - Farnell

8. Zynq-7000 All Programmable SoC ZC702 Evaluation Kit Quick Start Guide (XTP310)

9. ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide UG850 (v1.6) January 3, 2018

10. Xilinx XAPP745 Processor Control of Vivado HLS Designs, Application Note