1. Introduction

1.1 Introduction to GMRT:

The Giant Metrewave Radio Telescope (GMRT), located near Pune in India, is the world's largest array of radio telescopes at meter wavelengths. It is operated by the National Centre for Radio Astrophysics, a part of the Tata Institute of Fundamental Research, Mumbai.

The GMRT contains 30 fully steerable telescopes, each 45 meters in diameter spread over distances of upto 25 km. The design of these antennas is based on the **`SMART'** concept - **S**tretch **M**esh **A**ttached to **R**ope **T**russes. The reflector made of wire rope stretched between metal struts in a parabolic configuration. This configuration works fine as the telescope operates at long wavelengths (21 cm and above). Every antenna has four different receivers mounted at the focus. Figure 1.1 shows one such antenna. Each individual receiver assembly can rotate, enabling the user to select any of them for the observation. GMRT antennas operate in five frequency bands, centered at 153, 233, 327, 610, and 1420 MHz. All these feeds provide dual polarization outputs. In some configurations, dual-frequency observations are also possible.

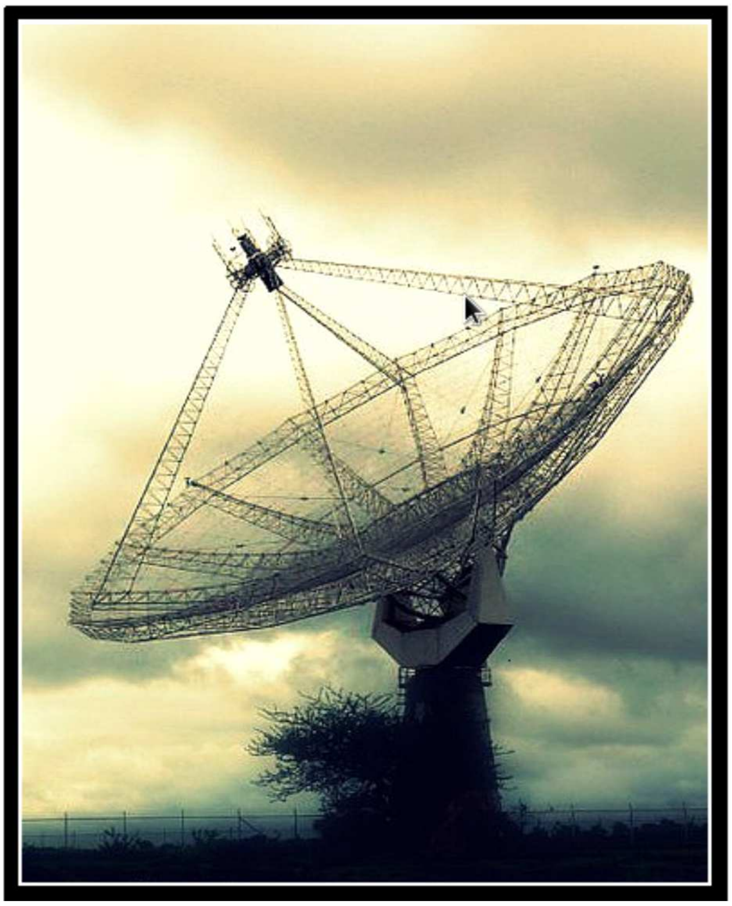
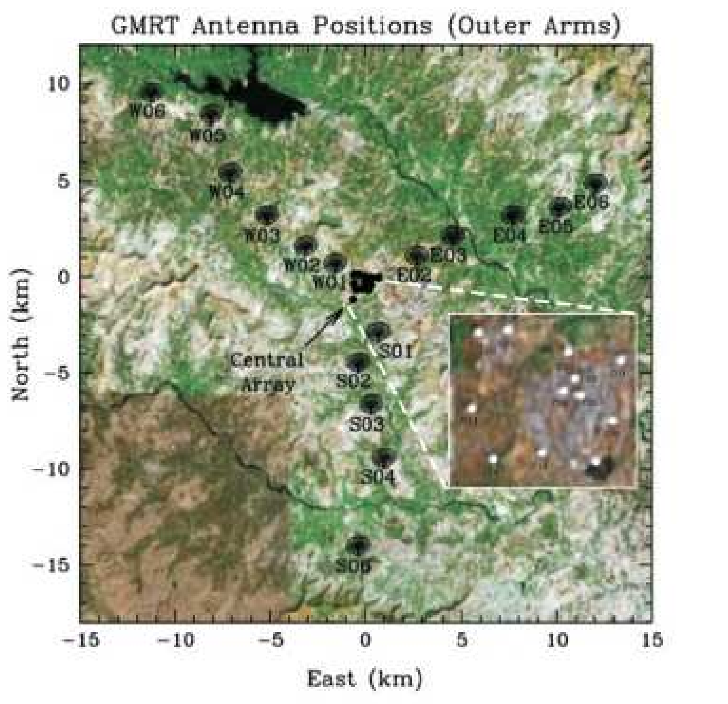


Figure. 1.1 Antenna

Out of the 30 telescopes at GMRT, fourteen telescopes are randomly arranged in the central square of 1 km by 1 km in size. Rest sixteen telescopes are arranged in three arms of a nearly ―'Y'-shaped array each having a length of 14 km from the array centre. The positions of the antennas in the antenna array have been shown in Figure 1.2.



The GMRT array as viewed from space. In the large scale picture the individual antennas are too small to be seen, instead their locations are indicated by symbols. In the blow up of the central square however, the individual antennas can be seen. Figure generated by Philip Lah and Premkumar.

Figure. 1.2 Antenna Array at GMRT

Therefore GMRT can act as an interferometer which uses a technique known as aperture synthesis to make images of radio sources. The multiplication or correlation of radio signals from all the 435 possible pairs of antennas or interferometers over several hours will thus enable radio images of celestial objects to be synthesized with a resolution equivalent to that obtainable with a single gigantic dish 25 kilometer in diameter! The maximum baseline in the array gives the telescope an angular resolution (the smallest angular scale that can be distinguished) of about 1 arc-second, at the frequency of neutral hydrogen. To provide seamless coverage from 100 MHz to 1600 MHz in addition to upgrades to the mechanical and servo control systems to the antenna and an improved high speed telemetry system for controlling the antennas remotely. This needs a major upgrade to the backend electronics, two possible solutions to the backend upgrade are currently being developed – one based on multiple FPGA boards, and second on GPU cluster.

Currently, the GMRT is undergoing an upgrade. As part of the upgrade, the GMRT plans to increase the bandwidth of the GMRT from the present value of 32 MHz to about 400 MHz and also plans to upgrade the digital backend from GSB (GMRT software Backend) to FPGA and GPU based backend.

1.2. Introduction to digital backend:

The digital backend is responsible for digital signal processing of the telescopic data used in interferometer and beamforming modes.
The digital signal is processed through FX Correlator (FX : FFT followed by Multiplier) to generate cross amplitude and phase information between each pair (baseline) among the 30 antennas to give the visibility information.
This data is used in imaging, continuum and many other astronomical observations.

1.3. Introduction to the project:

The Project of implementing and testing incoherent Packetized Beamformer is a part of the upgradation process of GMRT Backend system.
In Radio astronomy, beamforming is a technique which is used to get the pulsar profile. It can be of two types such as, Incoherent beamforming mode and coherent beamforming mode. The incoherent beamformer adds voltage signals from different antennae and computes the basic self term of voltage signals of the two polarizations. This incoherent beamformer for 4 antennae and 2 orthogonal polarizations is implemented on a multiple ROACH-boards (FPGA platform) and tested with proper pulsar source.

1.4. Significance of the project:

Pulsars are weak radio sources, and their individual pulses often do not rise above the background noise, so even with long base line it appears as a point source. Beamforming is the standard signal processing technique for its study to get its profile in higher resolution. Incoherent beamformer exhibits a higher sensitivity by $\sqrt{N}$ times (N= no of antennae). As the voltage signals of different antennae are squared and added, the incoherent beamformer provides vital information of the pulsars. So as a part upgradation process of GMRT backend, incoherent beamformer is implemented on FPGA.

FPGA is chosen as a hardware platform for its re-configurable features and better computing resources with lesser power conservation and higher bandwidth compared to the software based solution.

Within the scope of our project, we need to design the basic hardware and its interfacing utilities and test it with real time sources. So, the 4 antennae and 2 polarizations incoherent beamformer is implemented on multiple Virtex-5 pro FPGA (ROACH-board) to verify the functioning of the incoherent beamforming.

1.5. Aim and Objectives of the project:

The aim of this project is to design and implement incoherent packetized beamformer on multiple ROACH boards (FPGA platform) for 4 antennae 2 polarizations and test the design with Pulsars to get the pulsar profile.
The objectives of the project are:

- Design and implement the incoherent beamformer for 4 antennae and 2 polarizations on multiple FPGA platform (ROACH-board).
- Write scripts for the necessary interfacings of the ROACH-board with host PC.
- Simulation and implementation of design on hardware for verifying design logic.
- Verify the design using sky-test, i.e. testing with signals from radio sources (Pulsar).

1.6. Casper:

The Center for Astronomy Signal Processing and Electronics Research (CASPER) is a global collaboration dedicated to streamlining and simplifying the design flow of radio astronomy instrumentation by promoting design reuse through the development of platform-independent, open-source hardware and software.
The CASPER tool flow is better known as the MSSGE (Matlab/Simulink/System Generator/EDK) or bee xps tool flow. It is the platform for FPGA-based CASPER development and is the interface between several design and implementation environments.

Casper design environment in GMRT that is used during the course of this project use following version of different utility

- Matlab R2008a (v7.6.0)
- Simulink R2008b (v7.2)
- Xilinx System Generator v10.1.3.1386
- Xilinx EDK v11.5
- Xilinx ISE v11.5
- MSSGE libraries

The aim is to couple the real-time streaming performance of application-specific hardware with the design simplicity of general-purpose software. By providing parameterized, platform independent "gateware" libraries that run on reconfigurable, modular hardware building blocks, CASPER abstracts away low-level implementation details and allow astronomers to rapidly design and deploy new instruments.

CASPER instruments use reconfigurable open-source hardware built around Xilinx FPGAs. The GMRT uses Virtex 5 SXT95 based standalone FPGA processing board also called ROACH (Reconfigurable Open Architecture Computing Hardware). Figure 1.3 is an image of one such ROACH board. The ROACH board also has the
Following features:

- A separate PowerPC runs Linux and is used to control the board

• CX4/XAUI/10GbE Networks Interfacing Cards
• ADC2x1000-8: Dual 8-bit, 1000Msps (or single 8-bit 2000Msps), Atmel/e2v AT84AD001B ADC

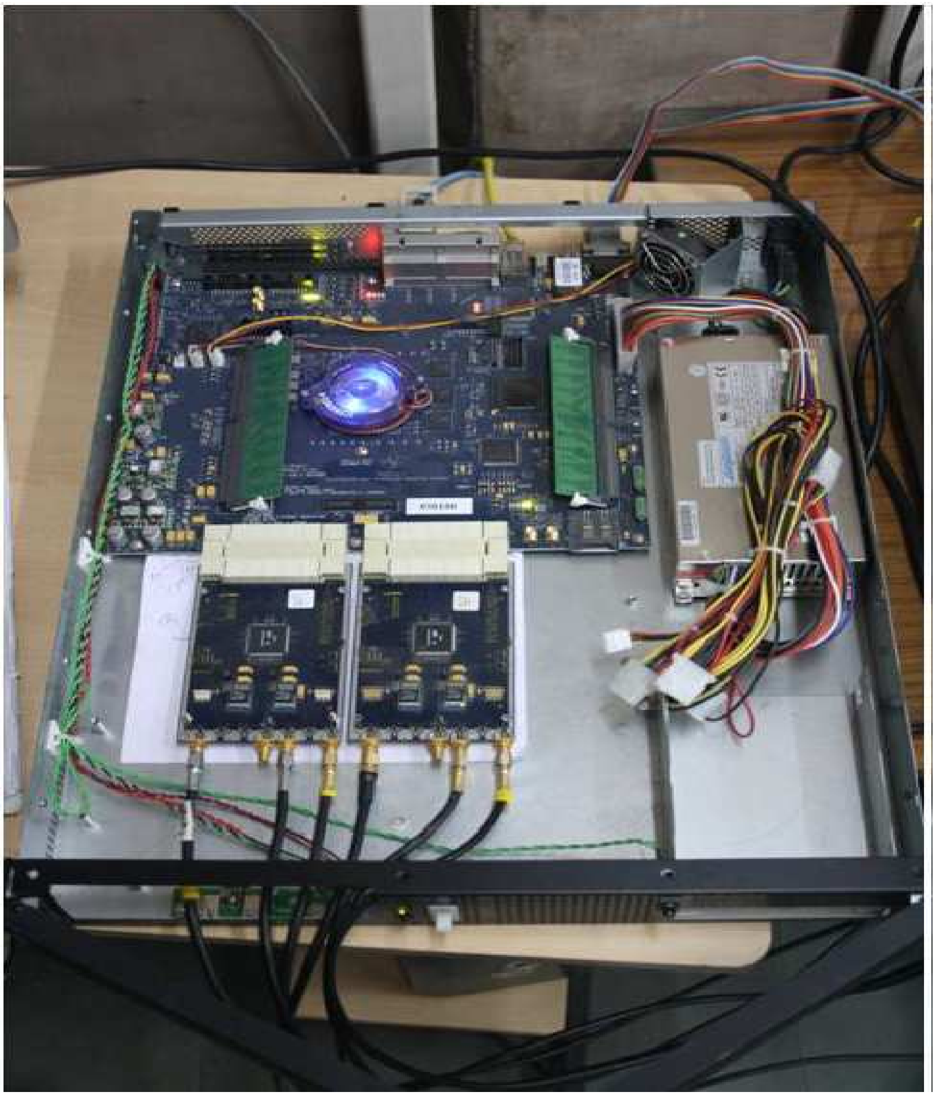


Figure. 1.3 Virtex 5 ROACH board

2. Theoretical concepts

2.1. Interferometry and correlator:

Interferometry is a technique in which waves are superimposed in such a way that one can analyze wave property from residual phase and spectrum. Interferometry makes use of the principle of superposition to combine waves in a way that will cause the result of their combination to have some meaningful pattern that is diagnostic of the original state of the waves. This works because when two waves with the same frequency combine, the resulting pattern is determined by the phase difference between the two waves— waves that are in phase will undergo constructive interference while waves that are out of phase will undergo destructive interference.

A radio interferometer measures the mutual coherence function of the electric field due to a given source brightness distribution in the sky. The antennas of the interferometer convert the electric field into voltages. The mutual coherence function is measured by cross correlating the voltages from each pair of antennas. The measured cross correlation function is also called Visibility. In general it is required to measure the visibility for different frequencies (spectral visibility) to get spectral information for the astronomical source.

The cross correlation between two signals $s_1 t$ and $s_2 t$

$$R_c \tau = < s_1 t \, s_2 t + \tau >$$

Where τ the time delay between the two signals and angle brackets is indicates averaging in time.

According to Wiener-Khinchin theorem which says, the power spectral density (PSD) of a stationary stochastic process is defined to be the FT of its auto-correlation function that is if

$$R_c \tau = < s_1(t) s_2(t+\tau) >$$

then power spectral density function $S_c f$ is

$$S_c(f) = \int_{-\infty}^{\infty} R_c(\tau) e^{-j2\pi f\tau} \, d\tau$$

From the property of Fourier transform we have

$$R_c(0) = < s_1(t) s_2(t) > = \int_{-\infty}^{\infty} S_c(f) \, df$$

2.2. Beamforming- coherent and incoherent:

Pulsars are the weak radio sources, so their individual pulses often do not rise above the background noise level. Beamforming is the basic technique used for their studies. Beamforming is a signal processing technique used in sensor arrays for directional signal transmission or reception. This is achieved by combining elements in the array in such a way that signals at particular angles experience constructive interference while others experience destructive interference. In beamformer, the antennae signals can be added coherently or incoherently.

Incoherent Beamforming:

• In incoherent beamformer, the voltage signals are firstly converted into power spectra. Then the power signals from the N dishes are combined to give the single incoherent beam. As the power spectra of the signals are added, the phase information is lost and no need of phase corrections.
• Root of N improvement in sensitivity.
• Beamwidth of single antenna.
• Application in large scale pulsar search
• The mathematical representation of the incohernt beamformer:

$$B_i = ({V_1}^2 + {V_2}^2)$$

This approach is used in the all the design in the course of this work.

Coherent beamformer:

• Voltage signals from the N dishes are combined to give the single coherent beam. As the voltages are added, it should be in phase with each other to get the resultant coherent signal referred as beam.
• N times improvement in sensitivity
• Beamwidth becomes narrower than the single antenna by nearly 1/N times.

• Application in studies individual known pulsars with its polarimetry studies.
• The mathematical representation of the coherent beamformer

$$B_i = (V_1 + V_2)^2$$

2.3. Pulsar observations requirements:

A pulsar is a rapidly rotating neutron star, highly magnetized which emits electromagnetic radiation beams from its magnetic poles as it rotates. The radiation is visible to us only if one of the poles points toward the earth. This appears to us as a very regular series of pulses with a period beam as low as milliseconds. The compact nature of its emission makes it a point source even for largest baseline on the earth.

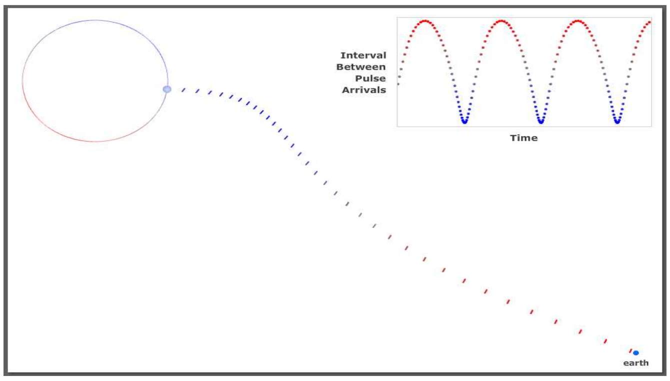


Figure 2.1: Radiation from pulsar

3. Packetized Beamformer Specifications

- Number of antennas: 4
- Polarization: Both polarization
- Number of spectral channels: 512
- Number of F engines: 4
- Number of X-engines: 8
- Number of spectral channels per X-engine:64
- Networks used: 1Gbps, XAUI link and 10 Gb Ethernet.
- Clock Frequency: 800 MHz
- Bandwidth : 400 MHz
- Base integration time: 0.163 milliseconds
- Data rate from 1 X-engine: 27.19 Mbps.
- Data rate from 8 X-engines: 223.2 Mbps.

4. Description of the project work:

4.1. Four Antenna Packetized Beamformer Design:

4 antenna packetized beamformer uses four F-engines and 8 X-engines.
Figure 3.1 shows the function performed by an F-engine and also shows after which stage the signal for beamforming is taped.

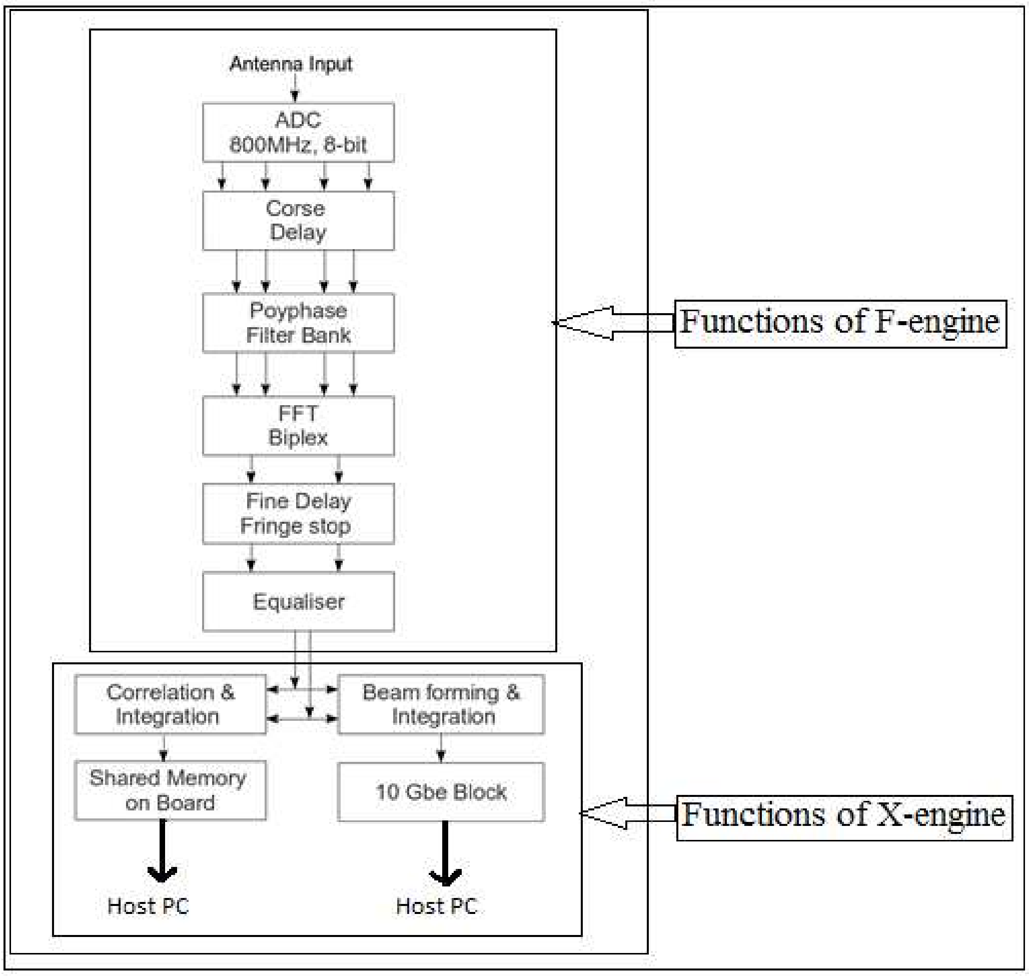


Figure 4.1. Functions performed by an F-engine

Each block mentioned Figure (3.1) is explained in brief below:

1. <u>ADC:</u> The ADCs interfaced to the ROACH board are ADC2x1000-8. They normally operate at 800MHzand give an 8 bit output through 4 channels each operating at 200MHz. This is done as the FPGA operates at 200MHz. In our design, the ADC is running at 400MHz clock frequency.

2. Delay: The radio sources in the sky are in motion over the sky. This differential change in position of the radio sources with respect to the antennas gives some delays. Other than that, the propagation delays from the antennas to the receiver are also considered. The whole delay that need to corrected for proper phasing is divided into two parts:
a. Integral multiple of clock is implemented in course delay block.
b. Fractional delay is implemented in fine delay fringe stop block.
The data rate at the output will be 4 channels of 8 bits at 200MHz.

3. PFB (Polyphase Filter Block) block: The polyphase filter bank implements a hamming window. The PFB is used to reduce spectral leakage and to increase signal to noise ratio. The data rate at the output will be 4 channels each of 18 bits at 200MHz.

4. FFT (Fast Fourier Transform): The FFT block used is FFT Biplex Real 4x (real-sampled biplex FFT). This block computes the real-sampled Fast Fourier Transform using the biplex FFT algorithm to use a complex core to transform two real streams. The data rate of operation at FFT output is 36 bits each at 400MHz. One of the streams gives even channels while the other gives odd channels. Each channel consists of an 18 (fix 18_17 format)bit real part and an 18 bit imaginary part.

5. Fine delay fringe stop Block: Fringe delay appears due to the down conversion of the RF signal to the baseband signal. The delay values are compensated for baseband signal but this give a drift in phase for RF signal. To compensate this drift in phase fringe stop is used. Using fine delay fringe stop block we can apply maximum 1 clock delay.

6. Equalizer block: This block scale down the amplitude of incoming from the channels by a given factor to avoid the over flow during correlation and integration. The scaling factor depends on the integration time and power level of the signal. This block casts the 36 bits input data into 8 bits data so that the bit growth during accumulation does not overflow 32 bits.

7. Beamformer and Integration block: The beamformer block in the design performs the squaring of voltage of a channel and adds it to the square of voltage from other antennas. Its working is explained in detail in section 3.2. The output is all the self-correlated data. This beamformer data is transmitted using 10GbE so that host PC can read the data and store it on a disk for further use.

8. Integration time = (No. of FFT cycle)*(No of FFT point)/(clock frequency)

9. Data rate= packet size/integration time.

4.2 BEAMFORMER SUBSYSTEM FLOWCHART

Figure 4.2 illustrates the signal flow of the beamformer subsystem.

The flow diagram is divided into 2 parts : PART A & PART B

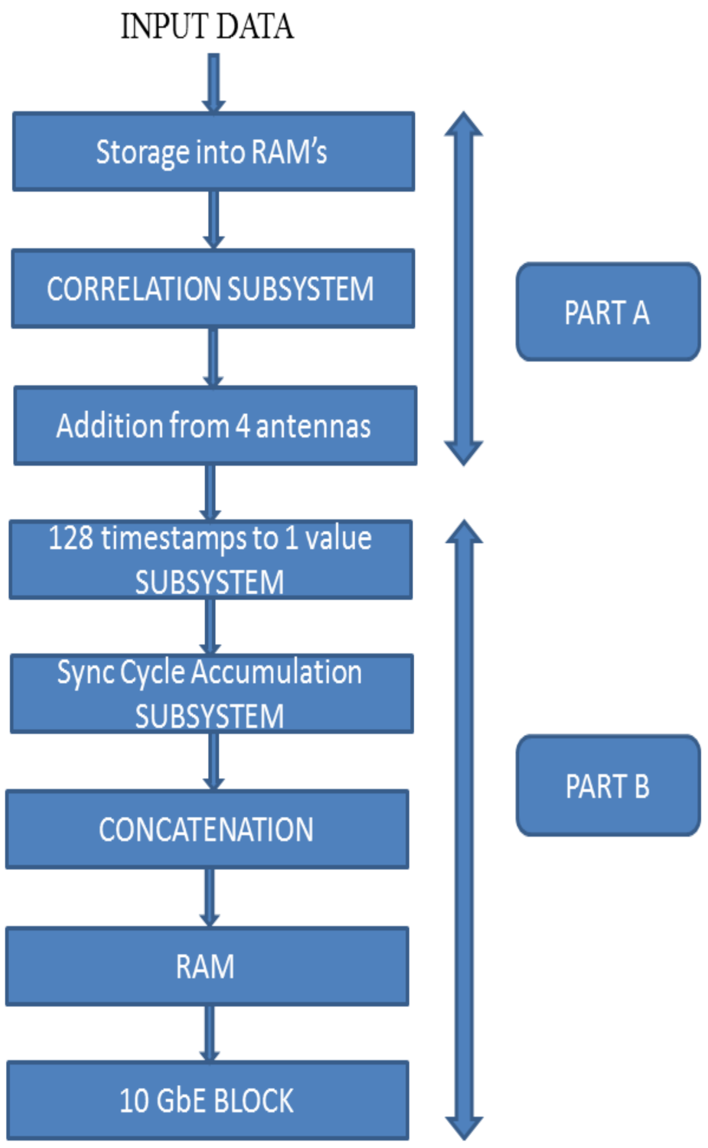


Figure 4.2: Flowchart of Beamformer Subsystem

4.3.BEAMFORMER SUBSYSTEM DOCUMENTATION

Figure 4.3 shows the block diagram of BEAMFORMER_INCOH subsystem:

PART A:

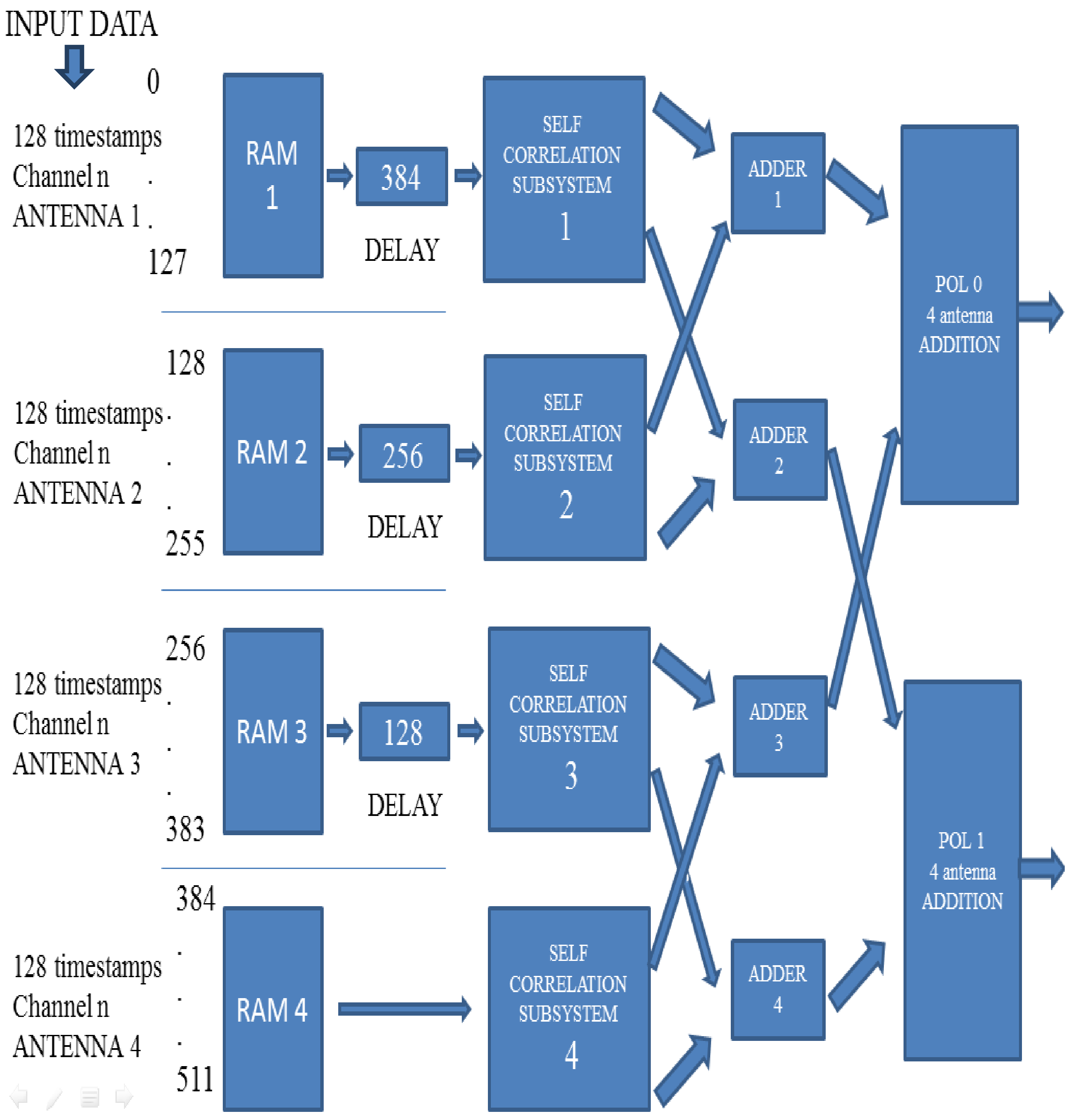


Figure 4.3 Block diagram of Beamformer Subsystem:Part A

BLOCK DIAGRAM

PART B:

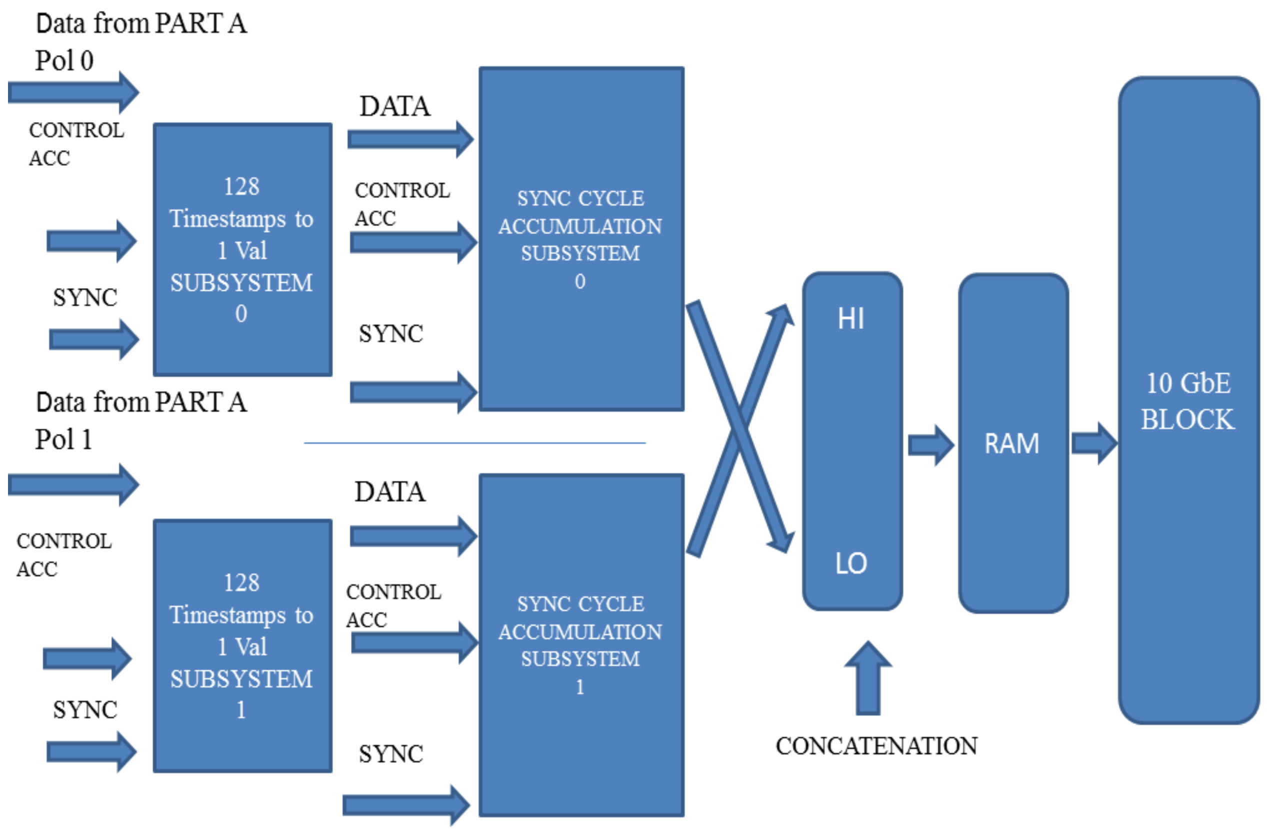


Figure 4.4 Beamformer Subsystem Block Diagram: Part B

4.3.1. INPUT TO THE SUBSYSTEM:

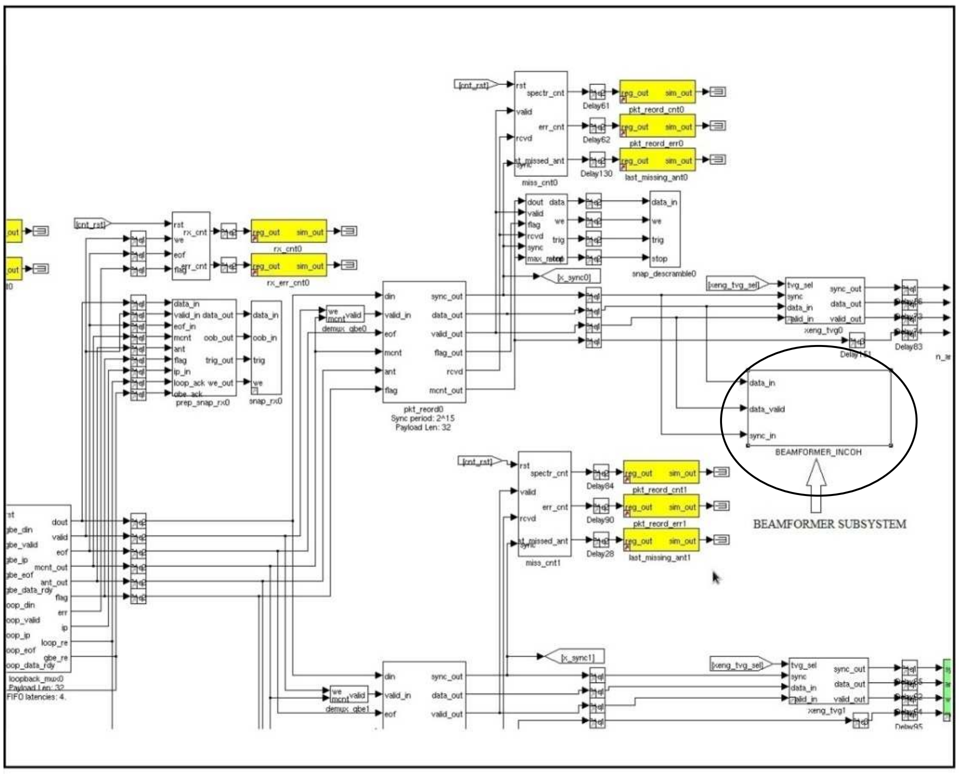


Figure 4.5 Position of the Incoherent Beamformer subsystem in the Packetized Correlator Design

The Beamformer subsystem comes after the Packet reorder block. Packet reorder block is the first part of an X-engine. It functions in the following manner:

The input signal is given to the roach boards acting as the F engine. The signal initially goes through an ADC and then an FFT is taken. The data from 512 channels of the F- engine is passed on to the X- engine.

All 512 channels are not processed by a single X- engine but in fact are distributed among the 8 X-engines of the system. Each X-engine takes responsibility for processing only a certain number of channels. 512 channels distributed among 8 engines implies that each X-engine processes 64 channels individually. This means every 8 th channel is processed by the same X-engine.

For eg. X engine 1: processes Channel 0, Channel 8,Channel 16…………Channel 504.

X engine 2: processes Channel 1, Channel 9, Channel 17…………Channel 505.

X engine 3: processes Channel 2, Channel 10, Channel 18…………Channel 506.

X engine 4: processes Channel 3, Channel 11, Channel19…………Channel 507.

X engine 5: processes Channel 4, Channel 12, Channel 20…………Channel 508.

X engine 6: processes Channel 5, Channel 13, Channel21…………Channel 509.

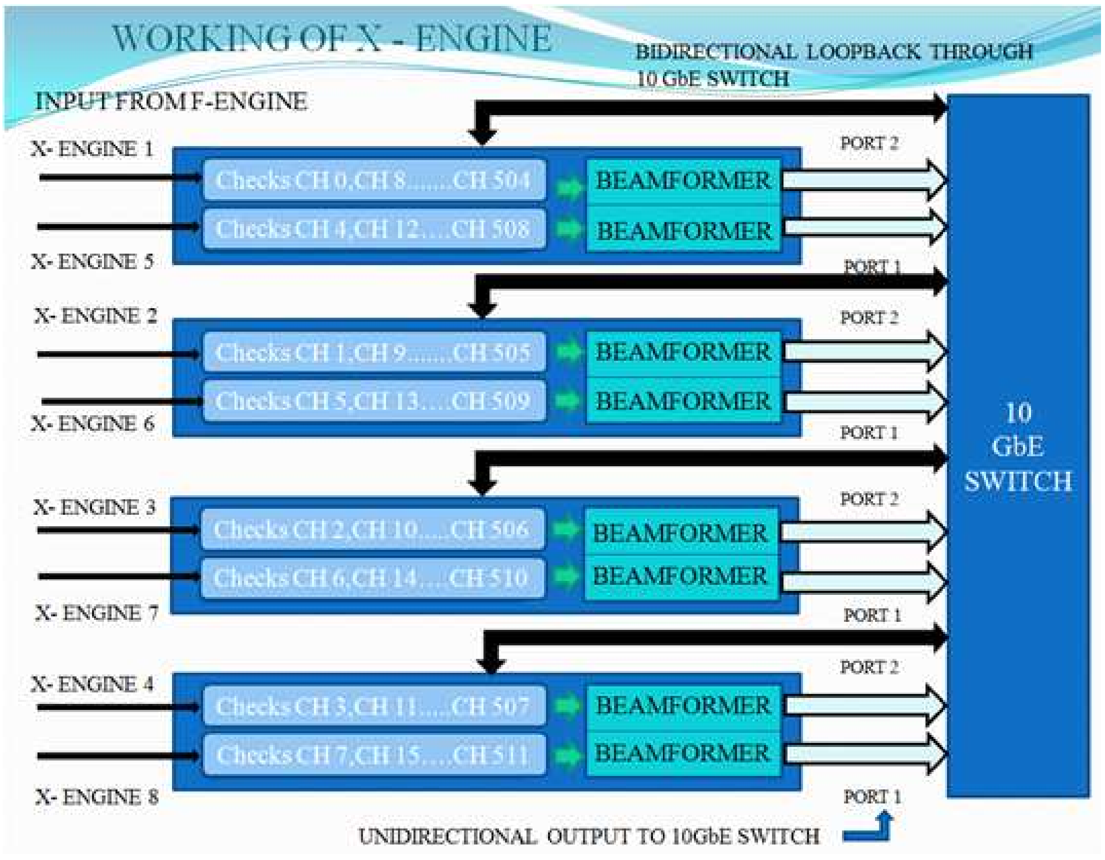
X engine 7: processes Channel 6, Channel 14, Channel 22…………Channel 510.

X engine 8: processes Channel 7, Channel 15, Channel23…………Channel 511.

Each X-engine processes only it's own set of channels irrespective of the antenna that the input is coming from.

There is a checker within each X-engine which checks if the incoming channel belongs to it's own set of channels. If it does belong then the X- engine accepts the data and passes it on to the beamformer subsystem for further processing.

If the incoming channel does not belong to it's own set of channels it does not accept the data and instead sends it to the 10 Gbe switch which routes it to the correct X- engine.



WORKING OF X - ENGINE
BIDIRECTIONAL LOOPBACK THROUGH 10 GbE SWITCH
INPUT FROM F-ENGINE
X- ENGINE 1
Checks CH 0,CH 8.......CH 504
BEAMFORMER
Checks CH 4,CH 12….CH 508
BEAMFORMER
X- ENGINE 5
PORT 2
PORT 1
X- ENGINE 2
Checks CH 1,CH 9.......CH 505
BEAMFORMER
Checks CH 5,CH 13….CH 509
BEAMFORMER
X- ENGINE 6
X- ENGINE 3
Checks CH 2,CH 10.....CH 506
BEAMFORMER
Checks CH 6,CH 14….CH 510
BEAMFORMER
X- ENGINE 7
X- ENGINE 4
Checks CH 3,CH 11.....CH 507
BEAMFORMER
Checks CH 7,CH 15….CH 511
BEAMFORMER
X- ENGINE 8
10 GbE SWITCH
UNIDIRECTIONAL OUTPUT TO 10GbE SWITCH

Figure 4.6 Working of first part of X-engine

The inputs to the beamformer subsystem are the following three signals:

1. Data_valid signal: Boolean signal; when high the incoming data at the input data port is valid.
2. Sync signal: For synchronization between different X-engines.
3. Input data: 16 bit data.
   The input data comes in the following format:

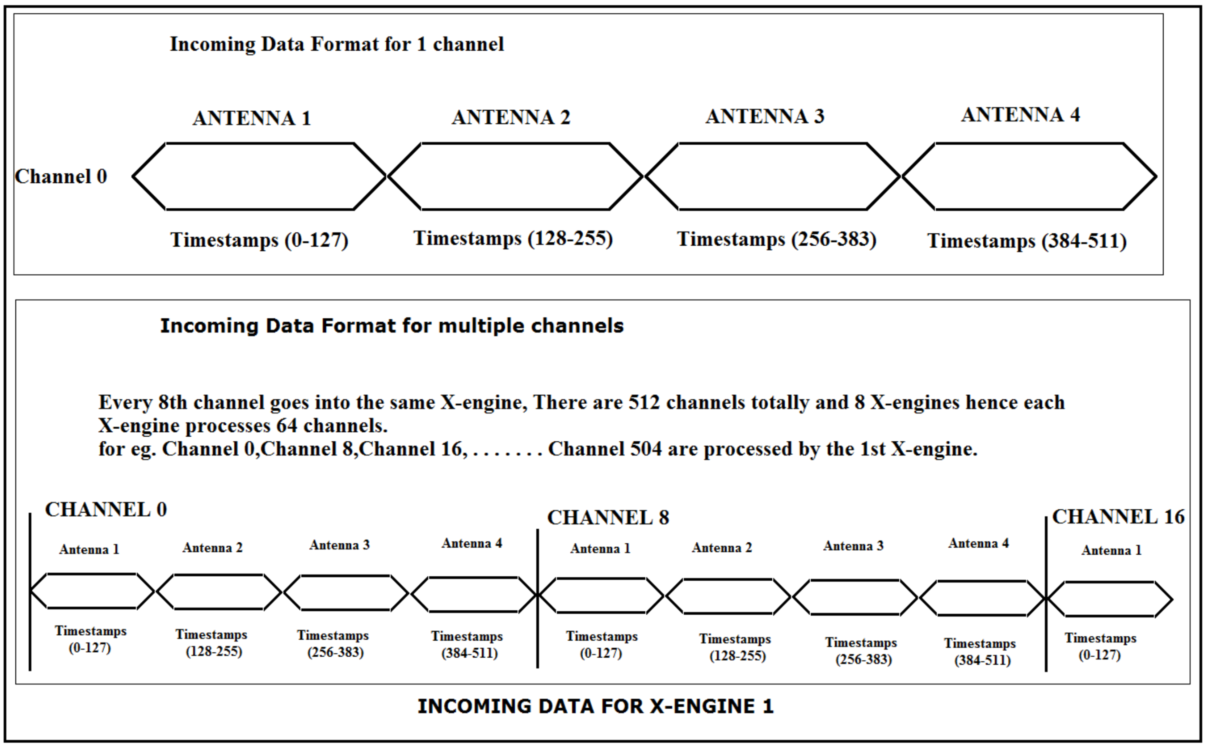

Incoming Data Format for 1 channel
ANTENNA 1
ANTENNA 2
ANTENNA 3
ANTENNA 4
Channel 0
Timestamps (0-127)
Timestamps (128-255)
Timestamps (256-383)
Timestamps (384-511)
Incoming Data Format for multiple channels
Every 8th channel goes into the same X-engine, There are 512 channels totally and 8 X-engines hence each X-engine processes 64 channels.
for eg. Channel 0,Channel 8,Channel 16, ... Channel 504 are processed by the 1st X-engine.
CHANNEL 0
CHANNEL 8
CHANNEL 16
Antenna 1
Antenna 2
Antenna 3
Antenna 4
Timestamps (0-127)
Timestamps (128-255)
Timestamps (256-383)
Timestamps (384-511)
INCOMING DATA FOR X-ENGINE 1

Figure 4.7 Incoming Data format at the input data port for X-engine1

4.3.2. WRITIG THE DATA TO RAM

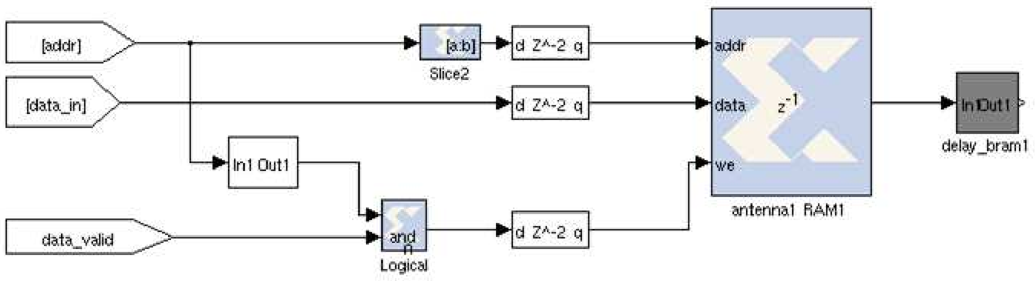


Figure 4.8 Writing Data to the RAM

1. Antenna and corresponding RAM: The incoming data is written to four single port RAMs. Each of these RAMs is 16 bit wide and has 128 address locations. Each one of these RAMs represents the corresponding antenna to which the data belongs. It is as follows.

NUMBER OF ANTENNA FROM WHICH THE DATA IS COMING	THE RAM TO WHICH THE DATA IS STORED
Antenna 1	RAM 1
Antenna 2	RAM 2
Antenna 3	RAM 3
Antenna 4	RAM 4

2. Generating address for RAM: A 9 bit counter gives the address. Only the 7 LSB are used to generate the address for a particular RAM. The 2 MSB are used for generating write enable which is explained in the next point. This counter has reset and enable ports. The counter is reset at every sync and it is enabled only when the data valid signal goes high.

3.Generating the write enable signal for RAM: One RAM has to be selected based on to which antenna the incoming data belongs to. This is done by using the 2 msb. Based on one them output of the selection block for only one BRAM is made high.

VALUE OF TWO MSB OF THE ADDRESS	THE RAM SELECTED
00	RAM 1
01	RAM 2
10	RAM 3
11	RAM 4

Then this output and data_valid are ANDed together and that is given to the write enable of that particular RAM.

4.Input data: The data comes as 128 time stamps for one channel from each antenna. This data is written into the single port RAM whose write enable is high.

4.3.3 DELAY SECTION:

1. Need: As seen earlier (in Section 4.3.1)the format in which the incoming data arrives at the data input port of the beamformer subsystem. The data from antenna 1 arrives first and this is followed by data from antennas 2, 3 and 4 sequentially. There are 128 timestamps of data from each antenna. So data for antenna 2 timestamp 1 comes 128 clock cycles after that of antenna 1 time stamp 1. As the data comes sequentially, data for antenna 3 timestamp 1 comes 256 clock cycles after that of antenna 1 time stamp 1 and data for antenna 4 timestamp 1 comes 384 clock cycles after that of antenna 1 time stamp 1. In order that the data from all the antennas arrives at the same time for the next step of processing these delays are used.

2. Implementation: The timestamps from all antennas are made to arrive with the timestamps from antenna4 by delaying them. In order to provide these delays 3 separate delay blocks are used for antenna 1, 2 and 3. The data from antenna 4 is not delayed.

ANTENNA NUMBER	DELAYED BY
Antenna 1	384
Antenna 2	256
Antenna 3	128

The **delay_bram** block from the CASPER DSP Block set is used here as the delay block

4.3.4. SELF-CORRELATION

This is done separately for each antenna. Therefore we have four correlation subsystems used in the design.

1. Separation of 16 bit input data: The 16 bits of input data contain data for both polarizations. The bottom 8 bits consist of data for polarization 0 and the top 8 bits consist of data for polarization 1. These 8 bits are contain of real and imaginary parts of the data as can be seen from the figure bellow.

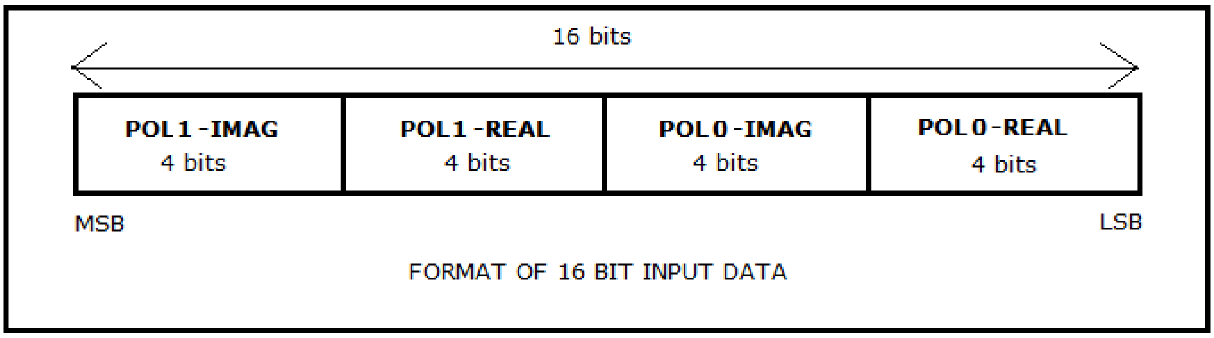


Figure 4.9 Format of input data

The separation of 16 bits data of the 128 time stamps is done sequentially. The 16 bit data is separated using **slice block** from the Xilinx simulink library. The processing of the two polarization has been done separately and in parallel here onwards.

2.Squaring and Adding: As it is an imaginary number, the square of an imaginary number is done as follows:

$(a+ib)(a-ib)^*=a^2+b^{2.}$ We have used the multiplication block **mult** from the Xilinx simulink library for squaring and **AddSub** block from the Xilinx simulink library for addition. For a particular antenna, 128 time stamps are squared and added sequentially.

4.3.5. ADDING DATA FROM ALL 4 ANTENNAS

For each antenna 2 outputs come out of the correlator subsystem. One is for polarization 0 and the other one is for polarization 2. Each of them gives out 128 timestamps of correlated data sequentially.

Now, the data from all the four antennas is added together.**AddSub** block from the Xilinx simulink library for addition. The figure 4.8 illustrates the addition.

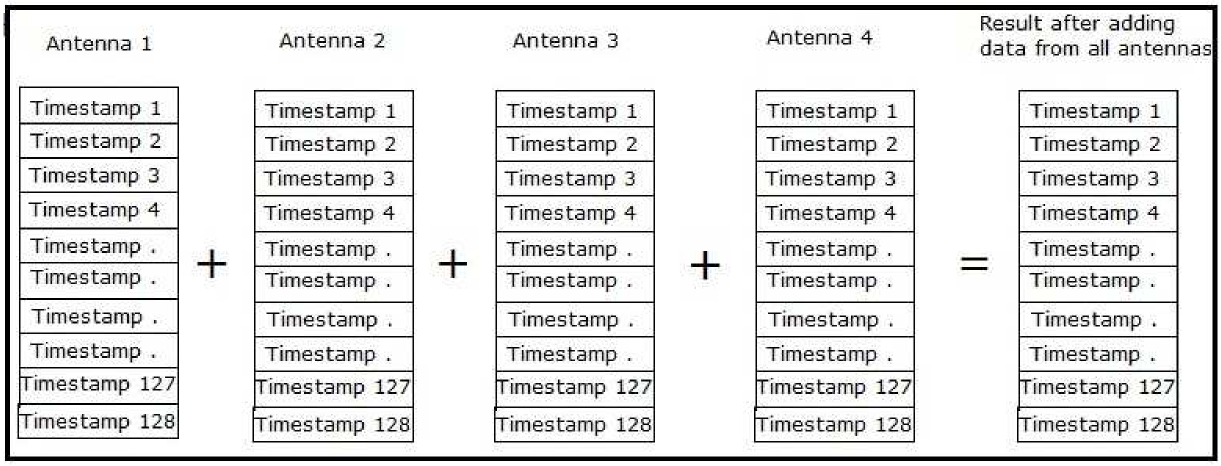


Figure 4.10 Adding data from all antenna

4.3.6. CREATING ONE VALUE OF 128 TIME STAMP VALUES

The subsystem **128 timestamp_to_1_val** subsystem does this operation.

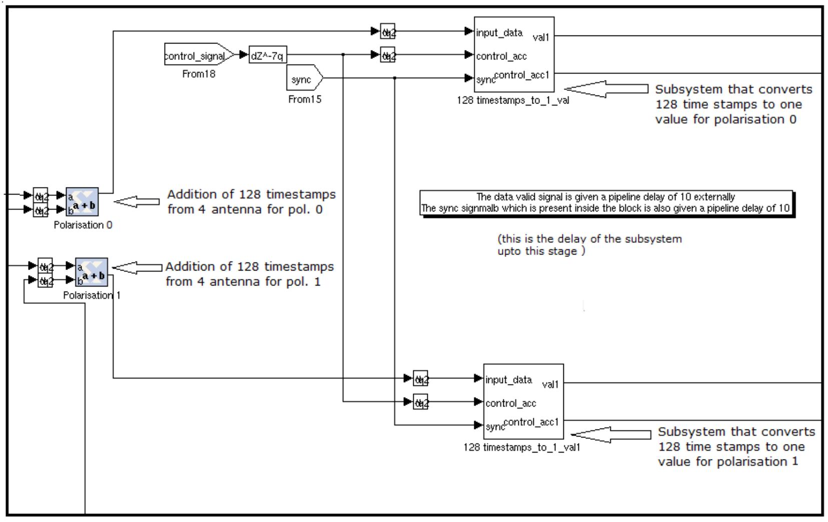

input_data
val1
control_acc
sync
control_acc1
control_signal
From18
dZ^-7q
sync
From15
128 timestamps_to_1_val
Subsystem that converts 128 time stamps to one value for polarisation 0
a
b
a + b
Polarisation 0
Addition of 128 timestamps from 4 antenna for pol. 0
Polarisation 1
Addition of 128 timestamps from 4 antenna for pol. 1
The data valid signal is given a pipeline delay of 10 externally
The sync signmalb which is present inside the block is also given a pipeline delay of 10
(this is the delay of the subsystem upto this stage)
128 timestamps_to_1_val1
Subsystem that converts 128 time stamps to one value for polarisation 1

Figure 4.11 Position of 128 timestamps_to_1_val subsystem

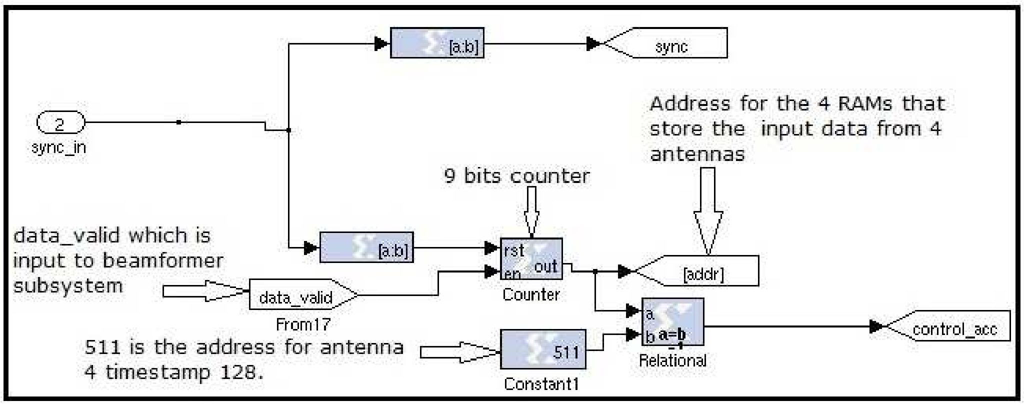
1.<u>Input to this subsystem</u>: The following 3 signals are the input to this system.

1. 128 timestamp data that comes sequentially at the input.
2. The sync signal that comes as an input to the Beamformer subsystem.
3. The control_acc signal generated.

2. <u>Generation of control_acc signal</u>:

This signal has been derived inside the subsystem. This signal goes high every time when timestamp 128 from antenna 4 for every channel arrives at the input. (i.e. when all 512 timestamps which represent 1 channel have arrived at the input.)

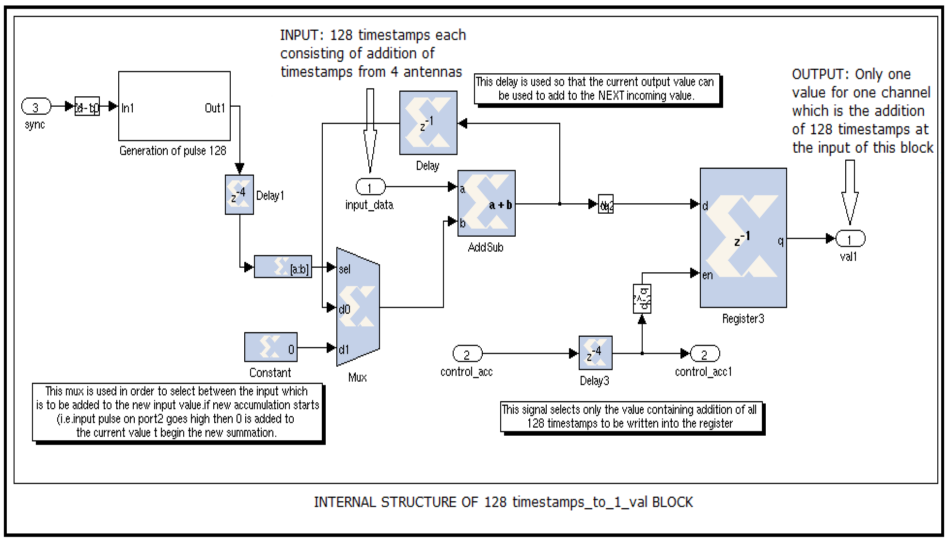
This signal is generated as follows:



[a:b]
sync
2
sync_in
Address for the 4 RAMs that store the input data from 4 antennas
9 bits counter
data_valid which is input to beamformer subsystem
[a:b]
rst
en
out
Counter
[addr]
data_valid
From17
511 is the address for antenna 4 timestamp 128.
511
Constant1
a
b
a=b
Relational
control_acc

Figure 4.12 Generation of control_acc signal

3.<u>Internal Structure of 128 timestamp_to_1_val subsystem:</u>



INPUT: 128 timestamps each consisting of addition of timestamps from 4 antennas
This delay is used so that the current output value can be used to add to the NEXT incoming value.
OUTPUT: Only one value for one channel which is the addition of 128 timestamps at the input of this block
3
sync
In1
Out1
Generation of pulse 128
Delay1
Delay
1
input_data
a + b
AddSub
[a:b]
sel
d0
d1
Mux
0
Constant
d
en
q
Register3
1
val1
2
control_acc
Delay3
2
control_acc1
This mux is used in order to select between the input which is to be added to the new input value.if new accumulation starts (i.e.input pulse on port2 goes high then 0 is added to the current value t begin the new summation.
This signal selects only the value containing addition of all 128 timestamps to be written into the register
INTERNAL STRUCTURE OF 128 timestamps_to_1_val BLOCK

Figure 4.13 Internal structure of 128 timestamps_to_1_val block.

1) Logic Used: The addition of 128 timestamps gives us the value for 1 channel. But next time when 128 timestamps come at the input they belong to a different channel. So a new addition has to begin after 128 time stamps belonging to one channel have been added.

A multiplexer has been used for to serve this purpose. After every 128 clock cycles, 0 is selected as the second input to the adder. First input to the adder is the incoming timestamp data.

If the incoming time stamp data belongs to the same channel then the adders output of previous cycle is used as a second input to the adder.

2) Generation of pulse signal of period 128 clock cycle:

Where is it used: This signal is used as a select signal to the multiplexer **Mux,** which selects between 0 and the previous output of the adder, to be the second input to the adder.

Why period of 128?: A period of 128 is selected as we want to add 128 timestamps.

3) Use of register: At the output of this subsystem we need only one value and that value should be addition of all correlated 128 timestamp values.

At the output of the adder at every clock cycle we have addition of the time stamps. But only at one particular cycle the output of the adder will have the addition of all correlated 128 timestamps. It is this value we desire. Whenever the control_acc signal goes high we have this value at the output of the adder.

Hence, we have connected control_acc signal to the enable port of the register so that only the desired value is passed to the next stages.

Register used here is the **Xilinx Register** from Xilinx Blockset ant the multiplexer used is **Xylinx Bus Multiplexer** from Xilinx Blockset

3.Outputs of this subsystem: The two outputs of this subsystem are the val1 and control _acc1 signal. Val1 is the value that is obtained by adding all the 128 timestamps of that particular channel. i.e. val1 is the channel value. control_acc1 is just the control_acc signal delayed by 4.

4.3.7. Accumulation of Sync cycles:

Sync cycle_accumulation subsystem is used for this purpose. This block is used to integrate multiple number of Sync cycles so as to obtain an averaged value of the incoming data.

The 3 inputs coming into the subsystem are:

1. DATA_INPUT
2. CONTROL_ACC
3. SYNC

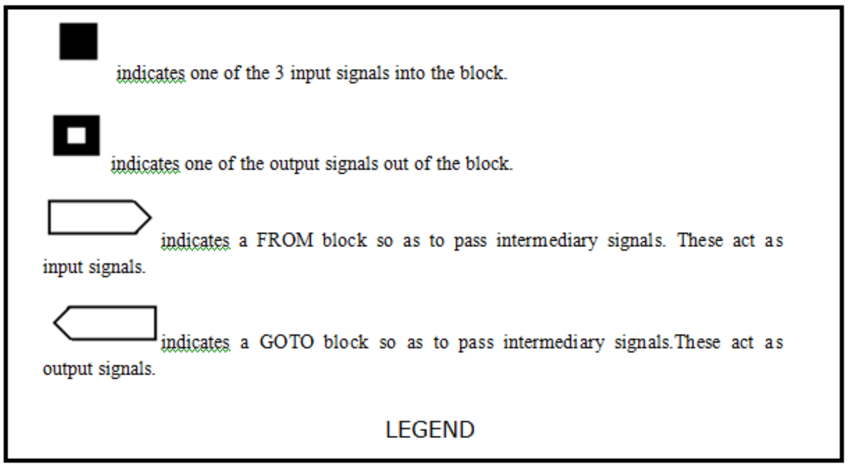
The outputs coming out of the subsystem are:

1. INTEGRATED_CHANNEL_DATA
2. TX_VALID
3. END_OF_FRAME

The position of this subsystem in the design of the beamformer subsystem is as shown in the following diagram.

**Position in the flow of design

We have divided the subsystem into 3 main parts in order to explain the flow of the design through it.



■ indicates one of the 3 input signals into the block.

▣ indicates one of the output signals out of the block.

indicates a FROM block so as to pass intermediary signals. These act as input signals.

indicates a GOTO block so as to pass intermediary signals.These act as output signals.

LEGEND

1. Generation of end of cycle signal:

This part depends on sync signal.

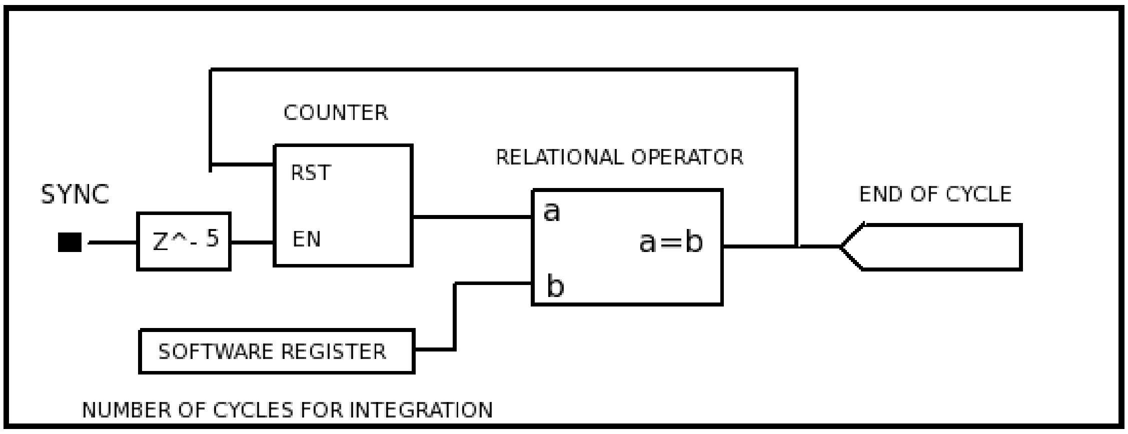


Figure 4.14 Generation of end_of_cycle

A variable "number of cycle for integration" is provided by the user through a software register so as to specify the number of cycles for which we want to integrate the channel data. It is configured through the Python script.

Let us say the value in the software register is given as n.

The Sync signal is used to enable a counter, when the value of this counter equals the "number of cycles for integration" provided in the software register we get a high pulse. This high pulse indicates that n cycles have been integrated. Thus this high pulse signal is called to the "end of cycle" signal. This signal is also used to reset the counter so that the counting for next cycle can begin.

2. Generation of new-accumulation,tx-valid,end of frame and we-accumulator signals:

This part depends on control_acc and the end of cycle signals.

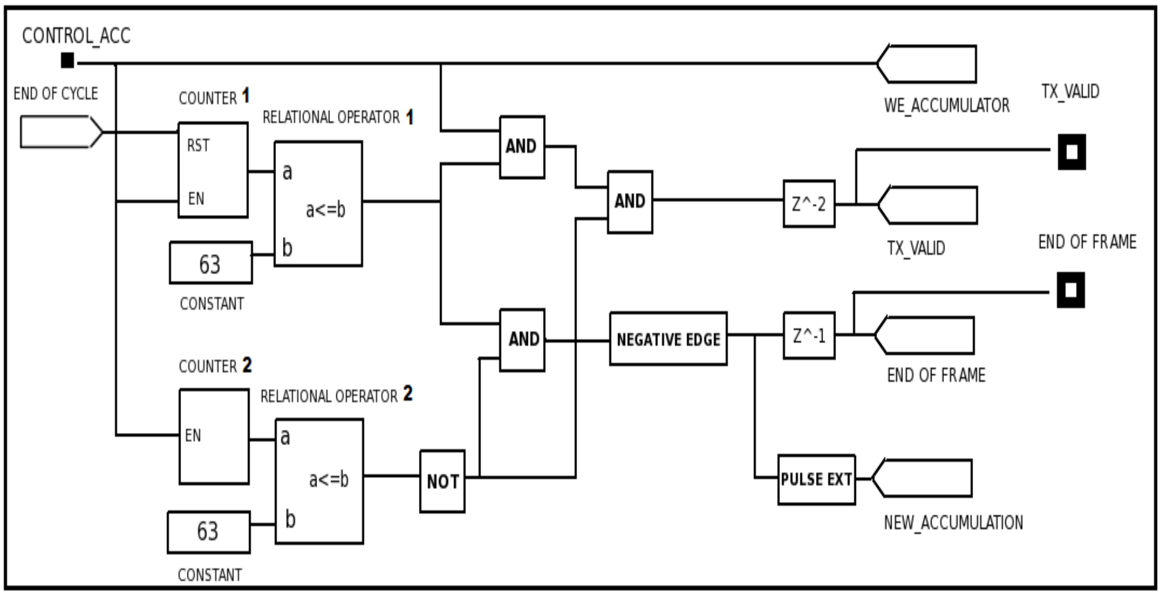


Figure 4.15 Generation of Tx_valid and end of frame

Generation of these signals:

Counter 1 is enabled by the "CONTROL_ACC" signal & reset by the "End of cycle" signal that was obtained in the first system. The output of this counter is given to Relational Operator 1. Relational Operator 1 compares this value with 63, as long as this value is less or equal to 63 we get a high pulse.

Counter 2 has only an Enable & no reset & even this counter is enabled by using the "CONTROL_ACC" signal. The output of this counter is given to Relational Operator 1. Relational Operator 1 compares this value with 63, as long as this value is less or equal to 63 we get a high pulse.

The ouput of Relational Operator 2 is inverted. This inverted output is AND'ed with the output of Relational Operator 1. The output of this AND gate is given to a negative edge block. The output of this block is our "End of Frame signal". The output of this negative edge block is given to a PULSE EXTENDER block. This output of this block is passed ahead to the "New_Accumulation" signal.

The output of Relational Operator 1 is AND'ed with the "CONTROL_ACC" signal and the output of this AND gate is AND'ed with the inverted output of Relational Operator 2 mentioned earlier. This is our "TX_VALID" signal.

Use of these signals:

We-accumulator: This signal is nothing but the control_acc signal that comes into the 2^15 cycle accumulator subsystem. This signal is used as a write enable signal for PORT A and PORT B of the dual port RAM. There is delay difference between enable of port A and port B.

New-accumulation: This signal is the select signal to the Mux before the adder in the part 3 of this subsystem. This signal stays high for the entire duration of cycle 1 of every n cycle integration. After that it stays low till the integration of n cycles is completed.

Tx-valid: The "TX_VALID" signal is very important for transmission over 10GbE. When this signal is high the core of the 10GbE accepts the data into the buffer. So, in our case every time addition of n cycles of channel values of the comes out of the PORTA of the dual port RAM, this signal goes high. That is, it is only in the last cycle that is nth cycle that we have the values that we want to transmit to the 10GbE block.

End of Frame: The "End of Frame" is a very important signal from the point of view of 10GbE. The End of Frame signal should go high when the last data of that particular packet comes to the data input port of the 10GbE. So, in our case when addition of n cycles of channel number 63 data values values comes out of the PORT A of the dual port RAM, this signal goes high That is, it is only in the last cycle, that is nth cycle, that we channel number 63 data comes out of port A this signal goes high

When the "End of Frame" signal is received, the packet of data get transmitted over the Ethernet. "End of frame" signals the transceiver to begin transmitting the buffered frame.

3. Integration of multiple 2^15 cycles.

We have used a Dual port RAM for carrying out the integration of multiple cycles. This dual port RAM is 32 bit wide and has 64 address locations.

This part is further divided into 2 parts.

1) Address Generation for Port A and Port B.

This depends on both SYNC as well as CONTROL_ACC

A Dual Port has 2 output ports A & B and requires 6 signals at it's input.

ADDR_A(Address location for Data in Port A),DIN_A(Input data for Port A), WE_A(When this signal is high the Data pointed by DIN_A is written into the address pointed by ADDR_A), similarly it also possesses ADDR_B, DIN_B, WE_B. These are for writing into Port B.

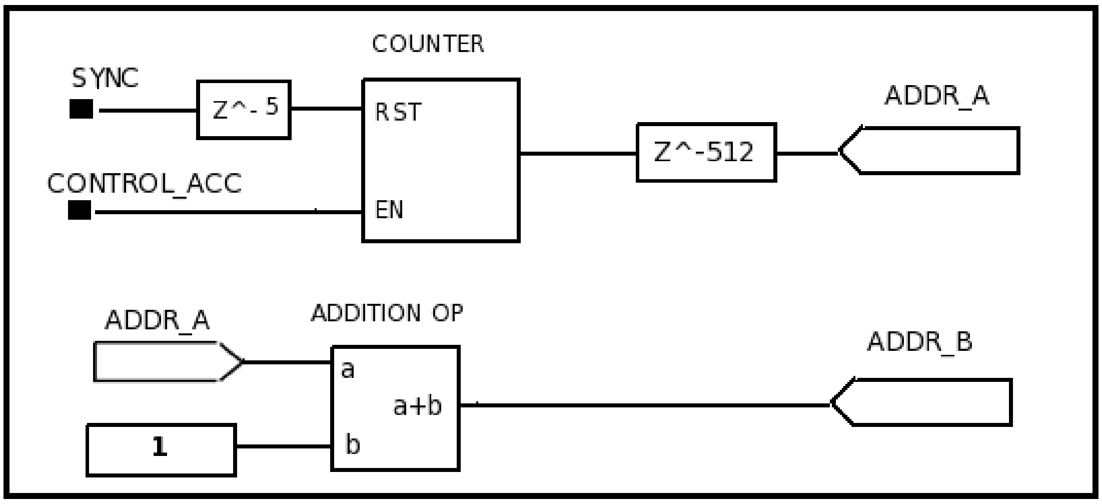


Figure 4.16 Generation of address for dual port RAM.

The ADDR_A is generated by a counter which is reset by SYNC signal & enabled by the "CONTROL_ACC" signal. Delays are adjusted accordingly. ADDR_B is then derived from ADDR_A as shown in the above figure.

The depth of the RAM used is 64.

We require ADDR_B to be differing from ADDR_A by the value 1. When ADDR_A is 0, ADDR_B is 1. When ADDR_A is 1, ADDR_B is 2 and so on. When ADDR_A goes to 63 ADDR_B goes to 0(63+1=64(1000000) in 6 bits (000000)). The addition operation is done by an adder which uses wrap around mode in order to give us the above result. The following table illustrates the same.

ADDRESS OF PORT A (ADDR_A)	ADDRESS OF PORT B (ADDR_B)
0	1
1	2
2	3
.	.
.	.
.	.
63	0

2) **Actual Integration done in dual port RAM:**

This part needs the following signals:
Data Input Signal,we_acc,new_acc,addr_a,addr_b.

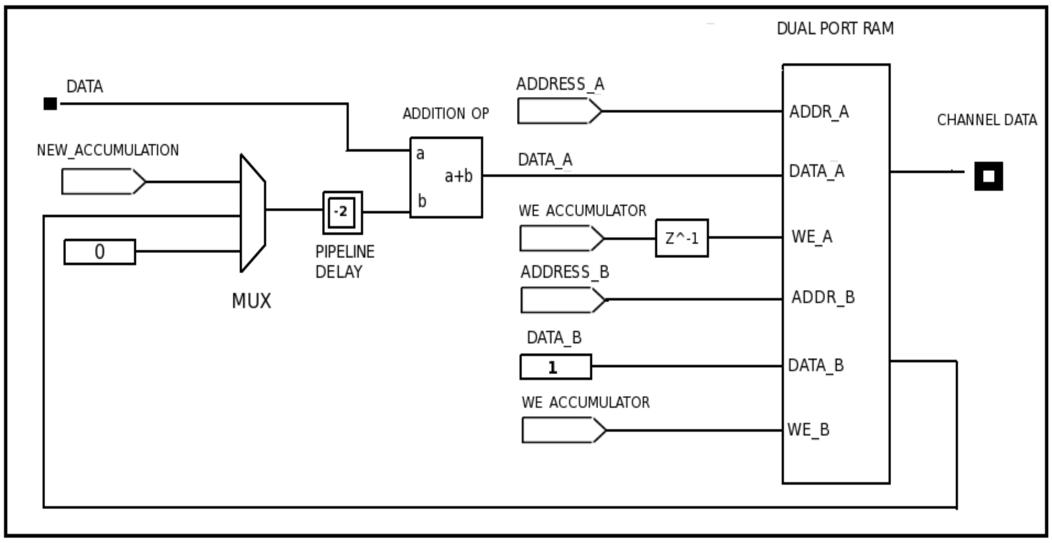


Figure 4.17 Accumulation using Dual Port RAM.

The NEW_ACCUMULATION that was obtained in the 2nd system is used as a Select signal for a MUX to choose between the output of Port B and a constant of Value 0.The logic used for integration is as follows:

Let us consider the integration of first n cycles.

Suppose in Cycle 1 the data for 64 channels comes into the system & in the next cycle i.e. Cycle 2 a completely new set of data for 64 channels came into the system. Subsequently in Cycle 3 the system receives a third set of data for 64 channels and so on till the nth cycle. We want the output of our system to be the addition of these n cycles. This can be accomplished as follows.

At the start of cycle 1 the “NEW_ACCUMULATION” signal selects the constant value 0 (input at port 1 of MUX) and this goes to the second input of adder for the addition operation. The data keeps streaming into the first input of the adder for the addition operation. Since the value at second input is 0 the value at first input of adder gets added by 0 only and hence moves to the output of the Addition operation without a change. This data is directly written to the PORT A of the DUAL PORT RAM. This “NEW_ACCUMULATION” stays high for all the channels for every 1st cycle of integration.

As was defined earlier the address at PORT B of the DUAL PORT RAM differs from address at PORT A of the DUAL PORT RAM by 1. At the end of first cycle addr_b will be pointing to address0 of the dual port RAM and addr_a will be pointing to address 63.

Let us now consider the start of 2nd cycle:

The “NEW_ACCUMULATION” signal is designed that it now selects the value at Port 0 of the MUX. This is actually the output of Port B from the RAM. Earlier addr_B was pointing to address 0 then at the port B output contains the data for channel 0.Now this value moves on to the second input of the addition operation block.While at the second input of the adder we have the 2nd cycles input data for channel 0.These 2 values get added by the Addition Operation Block & moves onto the input at DATA_A.

Thus in general it can be summarized as, the value of a certain channel at the first input of the Addition operation block is it’s value in the 2nd cycle & the value of in 1st cycle comes at second input of the Addition operation block. Now in the 3rd cycle, the addition of the values of the 1st& 2nd cycle get added to the now incoming data of the 3rd cycle, in the nth cycle the value at second input of the adder is the addition of all the previous cycles of that channel and at the first input of the adder the incoming data is the value of that channel in the nth cycle.

This can be done for any number of cycles that the user wants to integrate.

Once this number has been reached, the Output of the Dual Port RAM contains values that are to be sent forward to the Packetization stage.

NOTE: This sync cycle accumulator works perfectly for 1 cycle of integration i.e. base integration. But for greater number of integrations there are some unexpected drops in the output. These have been removed using the logic of the on-board integrator given in the Add-on section(Chapter 9) of this report.

4.3.8. PACKETIZATION STAGE:

1) Requirements of 10GbE block: We have input data in the form of 16 bits which contains information from Polarization 0 and Polarization 1. This data gets split into the respective Polarizations and independent parallel processing takes place till this stage. At the Packetization

stage the data is packetized according to the requirements of the 10GbE NIC. The 10GbE block accepts only a 64 bit wide data stream with user-determined frame breaks.

2) Formation of 64 bits wide data: The data from both polarizations is 32 bit wide. Both polarizations are concatenated to form a width of 64 bits. Then this 64 bits wide data is stored in a Single port RAM before sending it to the 10GbE block. Then 10GbE block wraps this data stream in a UDP frame for transmission. The block used for concatenation is the **Concat** block from Xilinx blockset in the Simulink library.

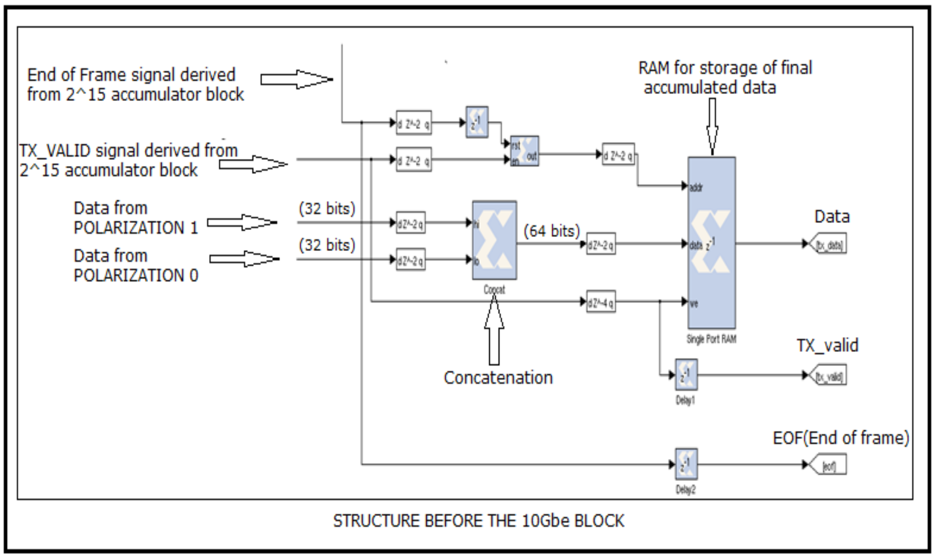

End of Frame signal derived from 2^15 accumulator block
TX_VALID signal derived from 2^15 accumulator block
Data from POLARIZATION 1
Data from POLARIZATION 0
(32 bits)
(32 bits)
(64 bits)
RAM for storage of final accumulated data
Data
TX_valid
EOF(End of frame)
Concatenation
Concat
Single Port RAM
Delay1
Delay2
STRUCTURE BEFORE THE 10Gbe BLOCK

Figure 4.18 Temporary storage before 10GbE block.

3)The 10GbE setup: The 10GbE block requires inputs as data, reset, tx_valid, tx_dest_ip, tx_dest_port ,tx_end_of_frame. Out of these tx_end of frame, tx_valid and tx_data are generated inside the BEAMFORMER_INCOH subsystem.

For the reset, tx_dest_ip and tx_dest_port software registers are present in the design. These software registers are configured via a python script. The following diagram shows the 10GbE setup within the design. The 10GbE block used is the **ten_Gbe_v2** block from the BEE_XPS System Blockset in the Simulink library.

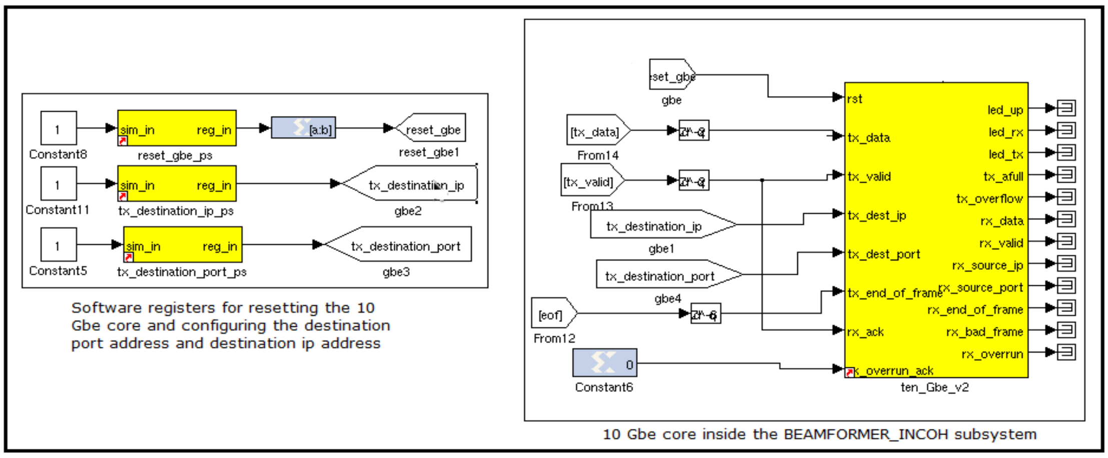


Figure 4.19 10GbE setup in the design.

Figure 4.18 shows the signals that are given to the 10 GbE block and the relationship between them.

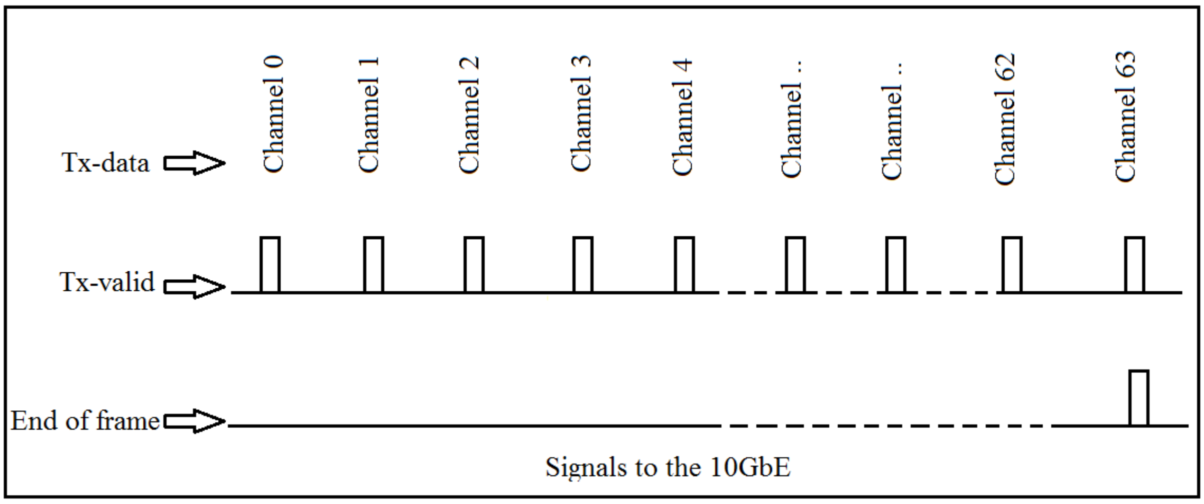


Figure 4.20 Signals to the 10GbE.

4)UDP packet: The 10GbE block sends a out a UDP Packet. Packet Format that is transmitted over 10GbE is as follows:

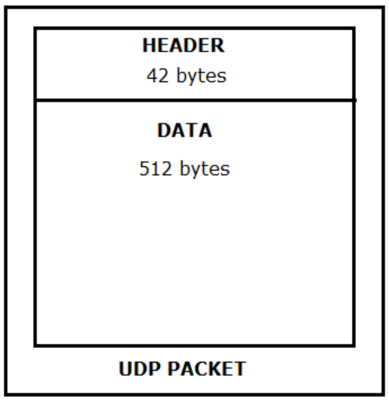


Figure 4.21 UDP packet

And the data in the UDP packet is as shown in figure 4.20 :

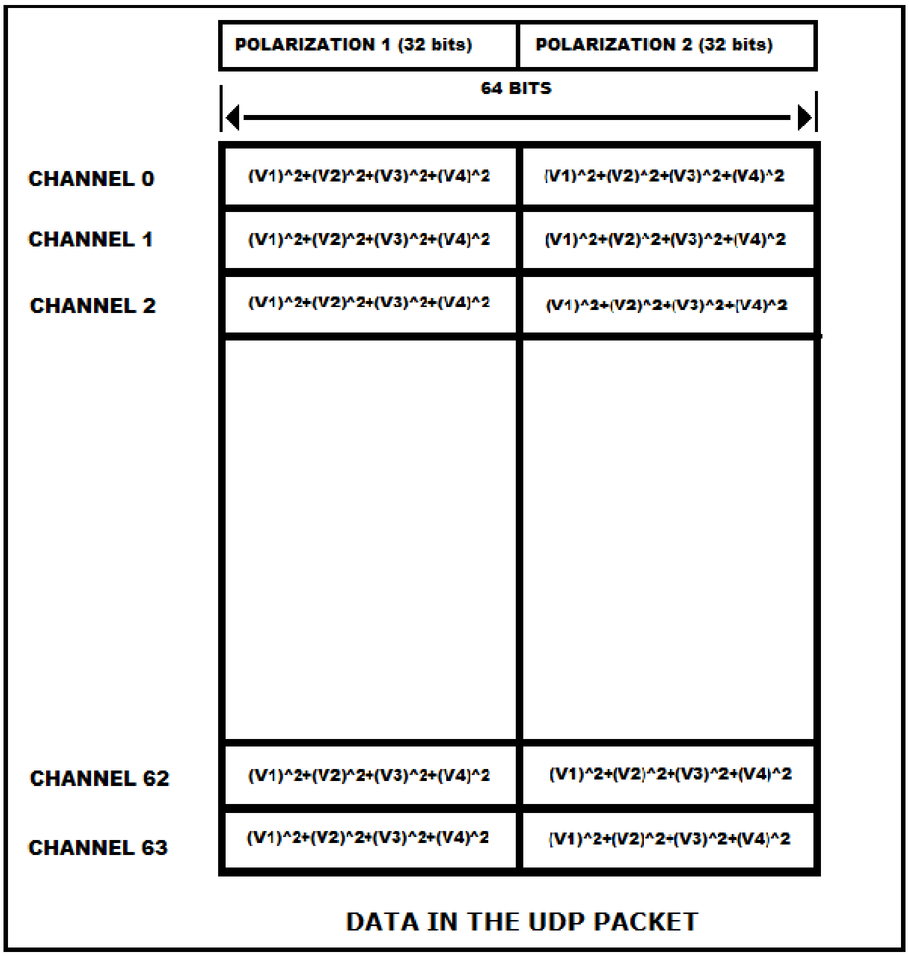


Figure 4.22 Data in the UDP packet.

On a roach board, there are two X-engines; therefore there are two BEAMFORMER_INCOH subsystems on a single roach board. The 10GbE for the upper subsystem gives the output at 10GbE port 2 of the roach board and the lower subsystem gives the output at 10GbE port1 of the roach board.

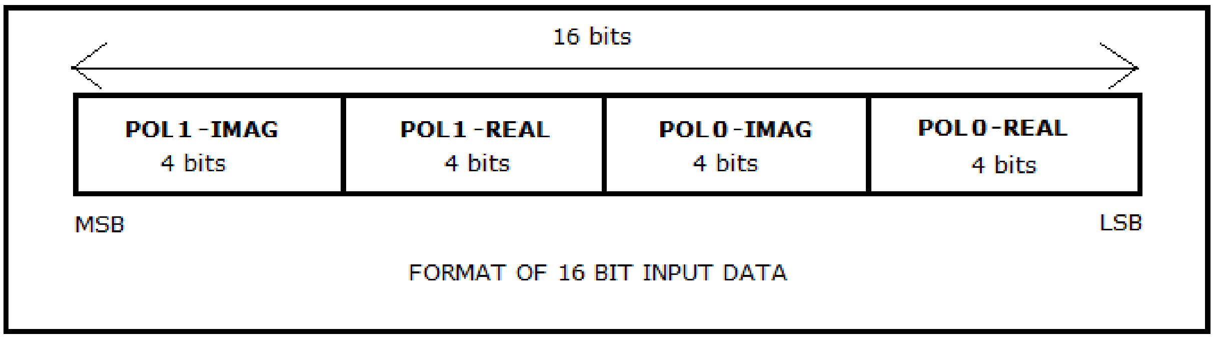
5. Calculations for Packetized Beamformer

5.1 NUMBER OF BITS CALCULATION:

(NOTE: Where ever we say integration it refers to the 2^15 cycle accumulation.)

The following calculation is done for one polarization. The same is true for the next.

1)The 16 bits input data has the following content in it.



We separate the real and imaginary parts of each polarization.

2) Then,

For each timestamp we do the following:

R^2+I^2.

Now the real part consists of 4 bits and the imaginary part consists of 4 bits.

So, the maximum value for each one of them will be:

$R_{max}=1111$, $I_{max}=1111$.

That is $R_{max=}$ $I_{max}=16$.

3) Now we have to square each one of them:

So, it will be: $(R_{max})^2=(I_{max})^2=256$.

Therefore the number of bits required to represent this maximum value will be **8 bits.**

4) Now in the next step we have to add the real part square and the imaginary part square.

So, $((R_{max})^2+(I_{max})^2)=256+256=512$.

Therefore the number of bits required to represent this maximum value will be **9 bits.**

5) In the next stage we have to add all 128 timestamps value and make one value out of them.

So, if all the 128 time stamp values have maximum value then the value of addition will be:

$128*((R_{max})^2+(I_{max})^2)= 128*512=65536$.

Therefore the number of bits required to represent this maximum value will be **16 bits.**

This value represents the value for 1 channel without integration.

Therefore 16 bits would be enough to represent the channel data values without integration.

The same is true for other polarization.

5.2 NUMBER OF INTEGRATION CYCLES:

From the above calculation, we know that the maximum value for 1 channel can be represented in 16 bits.

But we are using 32 bits for representing the channel data for one polarization.

Therefore the maximum number of integrations that can be done are :

$(2^{32}) \div (2^{16}) = 2^{16.}$

Therefore we conclude that the maximum number of integrations that we can do is $2^{16}=65536$.

(NOTE: Following calculations have been done for 1 X-engine.)

5.3 INTEGRATION TIME CALCULATION:

In one sync cycle, we have 128 time samples for each channel are received from 4 antennas. In one sync cycle 64 channels are received by one X engine.

Therefore we need 128*4*64 =32768 clock cycles.

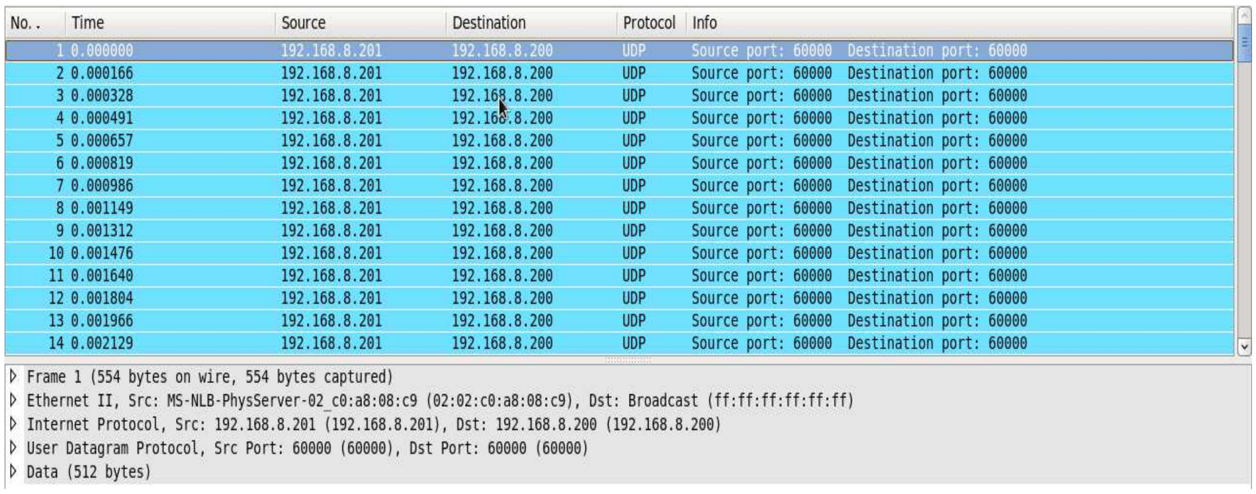
The operating frequency of ROACH board is 200 MHz .Therefore, one clock cycle time period is 5nanosecond.Total time for one sync cycle to be completed = 32768 *5ns =0.163millisecond(163 microseconds.)

If integrate further, then for 10 cycles the time taken would be 163 microsecond *10(1.63 milliseconds.)

These values have been verified with the wireshark software . As a packet is sent out after the specified numbers of integration cycles have been completed.

The following image is a wireshark snapshot for base integration . The leftmost column displays the time at which the packet arrives from the X-engine.

It is 163 micrsecond s*1(0.163milliseconds).



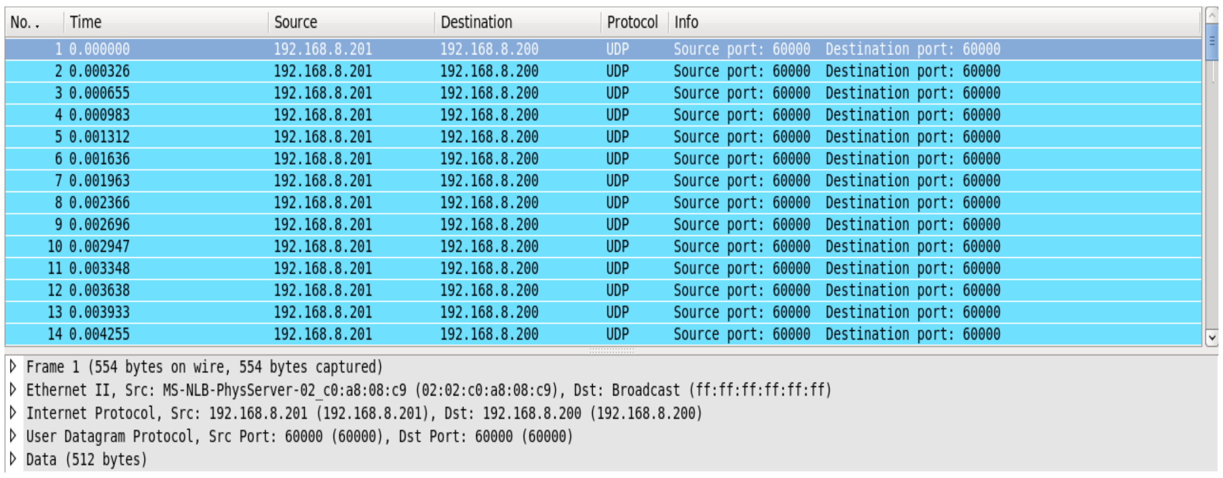
No. .	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
2	0.000166	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
3	0.000328	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
4	0.000491	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
5	0.000657	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
6	0.000819	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
7	0.000986	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
8	0.001149	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
9	0.001312	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
10	0.001476	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
11	0.001640	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
12	0.001804	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
13	0.001966	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
14	0.002129	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000

```
▷ Frame 1 (554 bytes on wire, 554 bytes captured)
▷ Ethernet II, Src: MS-NLB-PhysServer-02_c0:a8:08:c9 (02:02:c0:a8:08:c9), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▷ Internet Protocol, Src: 192.168.8.201 (192.168.8.201), Dst: 192.168.8.200 (192.168.8.200)
▷ User Datagram Protocol, Src Port: 60000 (60000), Dst Port: 60000 (60000)
▷ Data (512 bytes)
```

Figure 5.1 Wireshark snapshot for 1 integration cycle.

The following image is a wireshark snapshot for 2 integration cycles . The leftmost column displays the time at which the packet arrives from the X-engine.

It is 163 micrsecond s*2=0.326milliseconds.



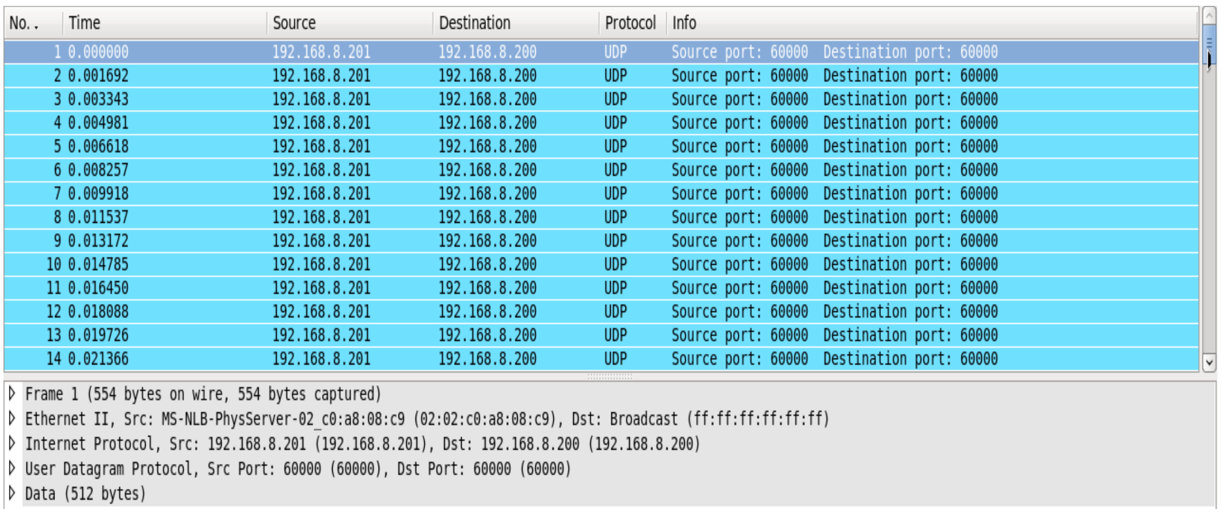
No. .	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
2	0.000326	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
3	0.000655	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
4	0.000983	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
5	0.001312	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
6	0.001636	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
7	0.001963	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
8	0.002366	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
9	0.002696	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
10	0.002947	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
11	0.003348	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
12	0.003638	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
13	0.003933	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000
14	0.004255	192.168.8.201	192.168.8.200	UDP	Source port: 60000 Destination port: 60000

```
▷ Frame 1 (554 bytes on wire, 554 bytes captured)
▷ Ethernet II, Src: MS-NLB-PhysServer-02_c0:a8:08:c9 (02:02:c0:a8:08:c9), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▷ Internet Protocol, Src: 192.168.8.201 (192.168.8.201), Dst: 192.168.8.200 (192.168.8.200)
▷ User Datagram Protocol, Src Port: 60000 (60000), Dst Port: 60000 (60000)
▷ Data (512 bytes)
```

Figure 5.2 Wireshark snapshot for 2 integration cycles.

The following image is a wireshark snapshot for 10 integration cycles . The leftmost column displays the time at which the packet arrives from the X-engine.

It is 163 micrsecond s*10(1.63milliseconds.)



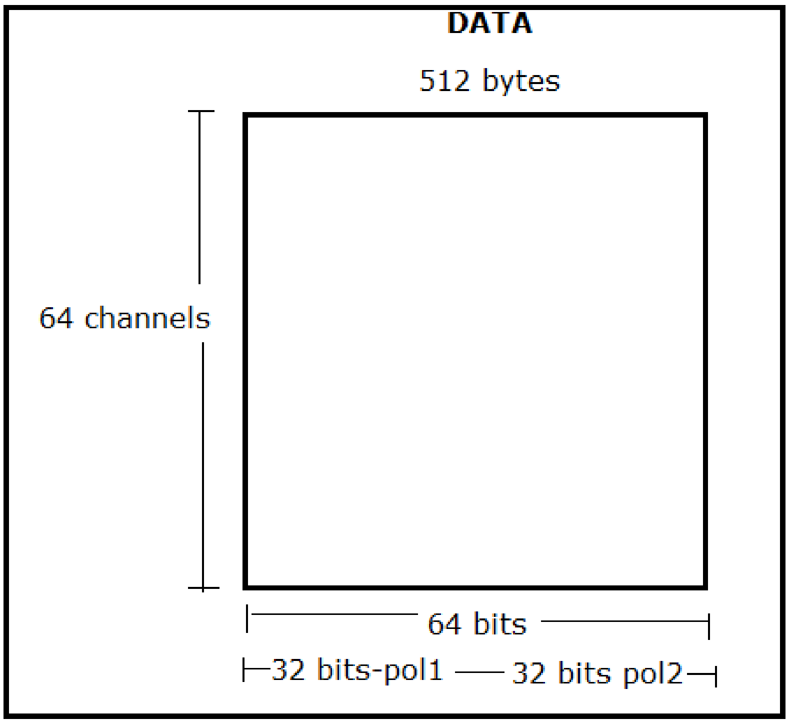
```
No..  Time        Source          Destination     Protocol  Info
   1 0.000000    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   2 0.001692    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   3 0.003343    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   4 0.004981    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   5 0.006618    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   6 0.008257    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   7 0.009918    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   8 0.011537    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
   9 0.013172    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
  10 0.014785    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
  11 0.016450    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
  12 0.018088    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
  13 0.019726    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000
  14 0.021366    192.168.8.201   192.168.8.200   UDP       Source port: 60000  Destination port: 60000

▷ Frame 1 (554 bytes on wire, 554 bytes captured)
▷ Ethernet II, Src: MS-NLB-PhysServer-02_c0:a8:08:c9 (02:02:c0:a8:08:c9), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
▷ Internet Protocol, Src: 192.168.8.201 (192.168.8.201), Dst: 192.168.8.200 (192.168.8.200)
▷ User Datagram Protocol, Src Port: 60000 (60000), Dst Port: 60000 (60000)
▷ Data (512 bytes)
```

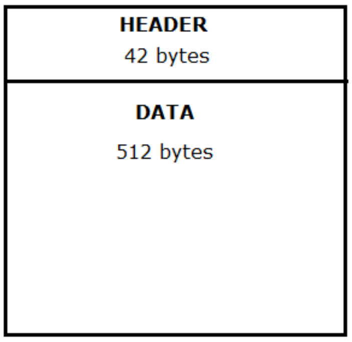
Figure 5.3 Wireshark snapshot for 10 integration cycles

5.4 DATA RATE CALCULATION:

The data that goes into one packet is as follows:



The minimum integration i.e. only 1 cycle of 2^15 is taken into consideration then every 0.163 millisecond (i.e. 163 microsecond) , a packet is transmitted from the X-engine. The data packet that is sent over the 10Gbps has the following format:



There are two options for capturing the data packets via Gulp. One option is to capture with the header and another without header. We will be calculating the data rate for both the options:

1) Without header:

Every 163 microsecond 1 packet of 512 bytes is sent out of one X-engine.

Therefore in 1 second 3.14 Mbytes are transferred.

Data Rate = 3.14 Mbytes * 8 (to convert to bits per second)

=25.13 Mbps.

2) With Header:

Every 163 microsecond 1 packet of 554 bytes is sent out of one X-engine.

Therefore in 1 second 3.398 Mbytes are transferred.

Data Rate = 3.398 Mbytes * 8 (to convert to bits per second)

=27.19 Mbps.

6. Depacketization and Post-Processing

6.1 DEPACKETIZATION:

This stage further consists of two parts.

1. Converting to ASCII: The data packets captured by gulp are in binary format. These are converted into ASCII format. Further Polarization 1 and Polarisation 0 data is separated.
2. Separating into 8 files: Gulp captures packets that are sent over 10Gb Ethernet.8 Different roach Boards are transmitting packets over a single 10Gb Ethernet. Hence these received packets have to be separated depending on the X-engine to which has transmitted that particular packet.

6.2 POST-PROCESSING:

In post processing, the data received from all the 8 X-engines separately, has to be interleaved in a particular order so as to get the entire spectrum of 512 channels. Section 6.3 explains both the Separation into 8 files and interleaving in depth.

6.3 LOGIC USED FOR SEPARATION AND INTERLEAVING:

The Packetized Beamformer design described here has the following specifications:

1. Number of spectral channels: 512
2. Number of X-engines: 8
3. Number of channels processed by each X-engine: 64.

Each X-engine receives data of only those 64 channels which it has programmed to process. After processing this data, each X engine sends out a packet which contains the channel data of these 64 channels. These packets are sent over a single 10GbE connection.But the channels that each X-engine receives are not consecutive. They are in the following format (Table X).

Sr. No.	Channel Numbers for each X-engine:							
	X-engine1	X-engine2	X-engine3	X-engine4	X-engine5	X-engine6	X-engine7	X-engine8
1	0	1	2	3	4	5	6	7
2	8	9	10	11	12	13	14	15
3	16	17	18	19	20	21	22	23
.	..	..	..	..	..	..	..	..
.	..	..	..	..	..	..	..	..
63	496	497	498	499	500	501	502	503
64	504	505	506	507	508	509	510	511

For processing the entire spectrum, we have to interleave the data of the 10GbE packets coming from each X-engine. Interleaving is carried out in 2 steps as follows.

1. The incoming data captured by gulp is separated into 8 files depending on the source IP of the X-engines.
2. The spectral channels from these 8 files have to be arranged serially.

These two things are achieved using the following technique :

1. Separating the data into 8 files (according to the X-engine): The data packets sent over the 10Gbe connection are UDP packets. Each of these packets contains a header of 42 bytes. This header contains the IP address of the source and destination in the 27th to 30th byte respectively.
   The header structure is as follows:

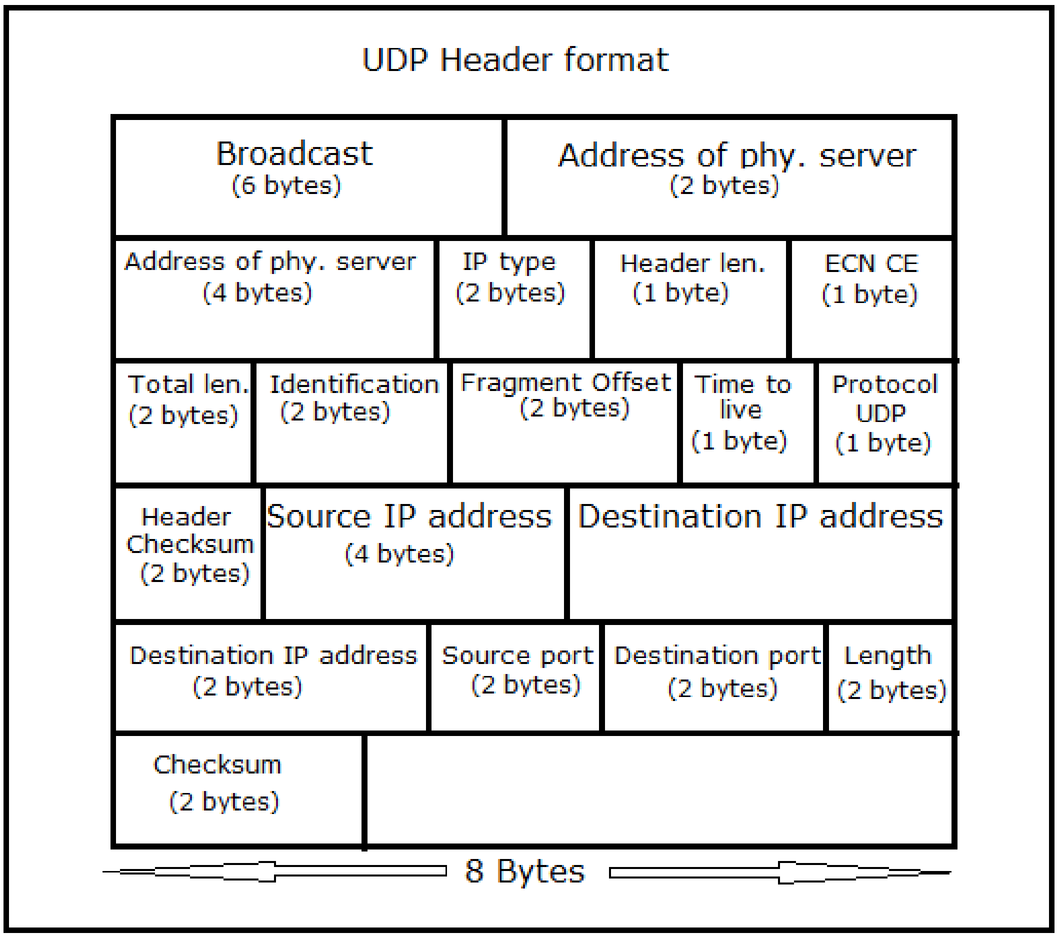


Figure 6.1 UDP Packet Header

Each of the 8 X-engines uses a different IP address. So, the X-engine that is sending a particular packet can be identified on the basis of the source IP present in the packet header sent by it.

Gulp captures the packets along with the header. A utility is developed to extract the source IP from the header, identify it and accordingly select a file in which data has to be written. This utility also converts the packet data into GNU compatible and PMON compatible format. Thus 8 different files each containing data from a particular X-engine are created at the end of this process.

2. <u>Interleaving the 8 files to get a serial output:</u> Then these 8 files are provided to the interleaving code which arranges the channel data sequentially. Figure 6.2 illustrates how channels from 8 different files are arranged in a single file.

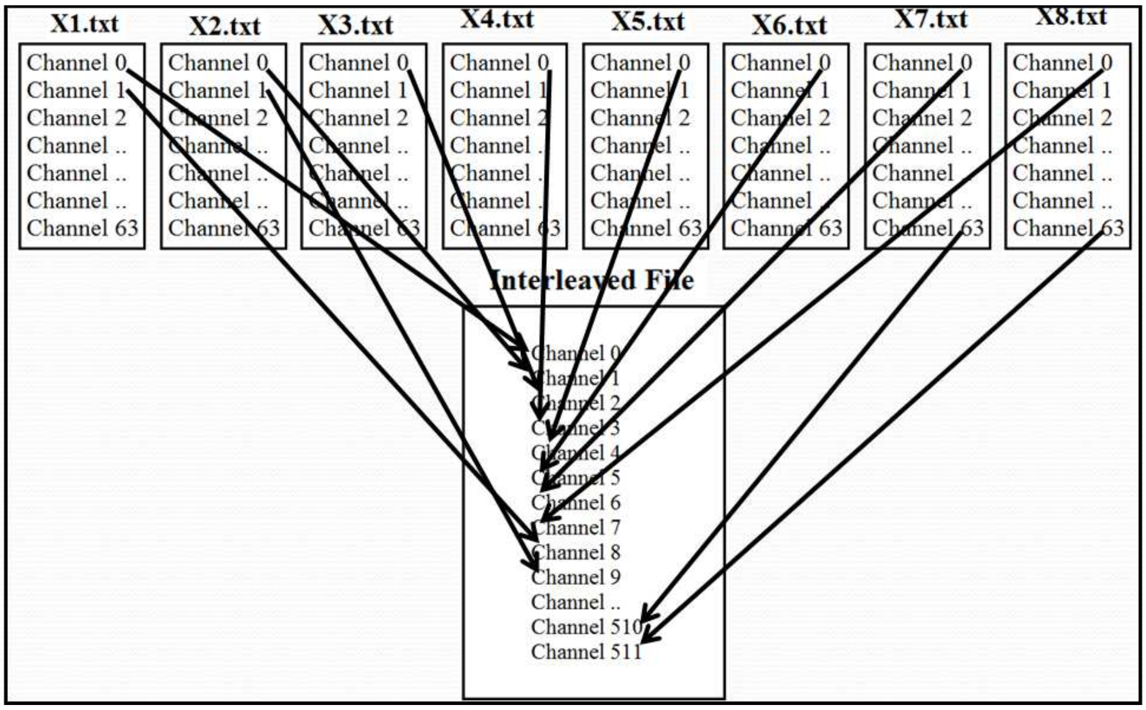


Figure 6.2 Interleaving

In the figure 6.2, X1.txt contains data from X-engine 1, X2.txt contains data from X-engine 2 and so on.

A "packet count" can be an add-on to the system. It will ensure interleaving of time synchronized packets.

3. <u>Ensuring time synchronized interleaving</u>: A packet counter can be transmitted along with the channel data. This is appended to the data packet at the packetization stage (before sending it over the 10 GbE link). This packet counter acts as a time stamp for synchronization of packet transmission from different X-engines. The process of interleaving starts with checking of the packet counter. Only the packets from different

X-engines containing same packet counter will be interleaved together. If even one X-engine’s packet with a specific packet count is not received, then all the packets with that packet count from other X-engines will be discarded.

7. Packetized Beamformer Test Setup

1) **DESIGN SPECIFICATIONS:**
   Number of F-engines: 4
   Number of X-engines: 8
2) **ROACH BOARDS USED:**
   ROACH boards used as F-engine:
   ROACH040241, ROACH040242, ROACH040237, ROACH040246.
   ROACH boards used as X-engine:
   ROACH030167, ROACH030116, ROACH030174, ROACH040235.

   There are two X-engines per ROACH board. The X-engines and the corresponding ROACH boards are as mentioned in the following table:

ROACH BOARD NUMBER	CORRESPONDING X-ENGINES
030167	X1 and X5
030116	X2 and X6
030174	X3 and X7
040235	X2 and X8

Table (Y): ROACH board corresponding to the 8 X-engines.

(Note: Any other ROACH board can be used by providing the name of the desired ROACH board in the config_4ant script. Also make changes in the server_f and server_x accordingly.)

3) **CONNECTIONS TO THE F-ENGINE:**
   F-engine is given four inputs:
   1) Sync
   2) I (Polarisation 0 input)
   3) Clock
   4) Q (Polarisation 1 input)

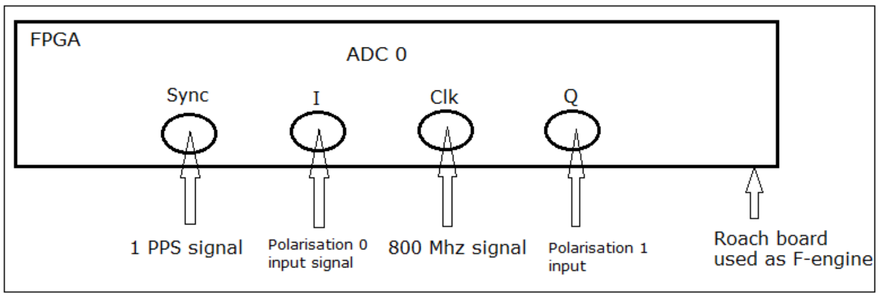


Figure 7.1 Connections to F-engine.

4) 10 GbE PORT CONNECTIONS OF X-ENGINE:

Every ROACH board has 4 10 GbE ports. The connections to them are as shown in the figure7.2:

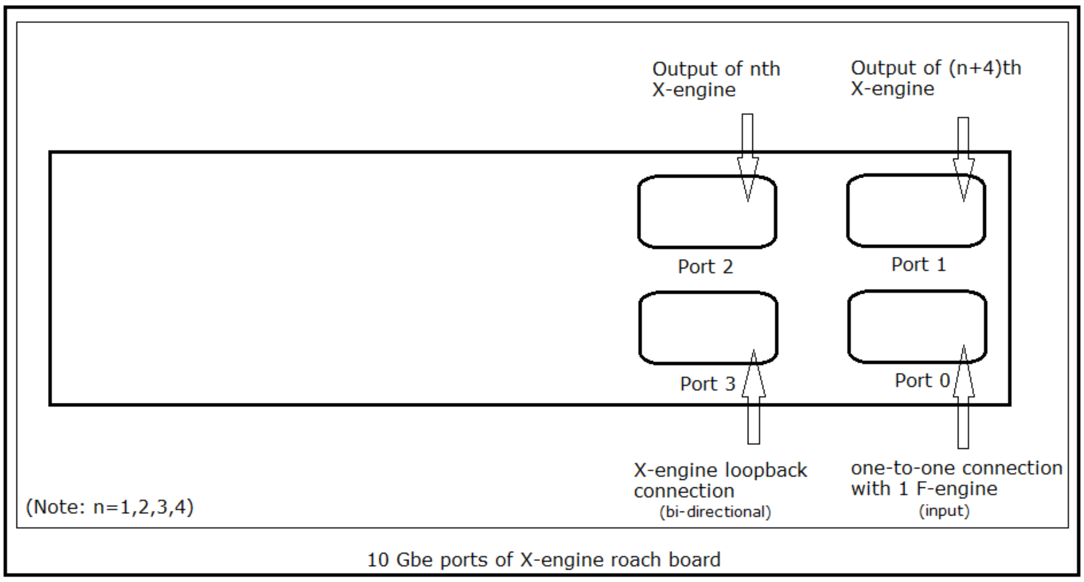

Output of nth X-engine
Output of (n+4)th X-engine
Port 2
Port 1
Port 3
Port 0
X-engine loopback connection (bi-directional)
one-to-one connection with 1 F-engine (input)
(Note: n=1,2,3,4)
10 Gbe ports of X-engine roach board

Figure 7.2 10GbE port connections of X-engine.

5) CONNECTIONS BETWEEN F-ENGINE AND X-ENGINE:

The connections between F-engine and X-engine are as shown in the following diagram:

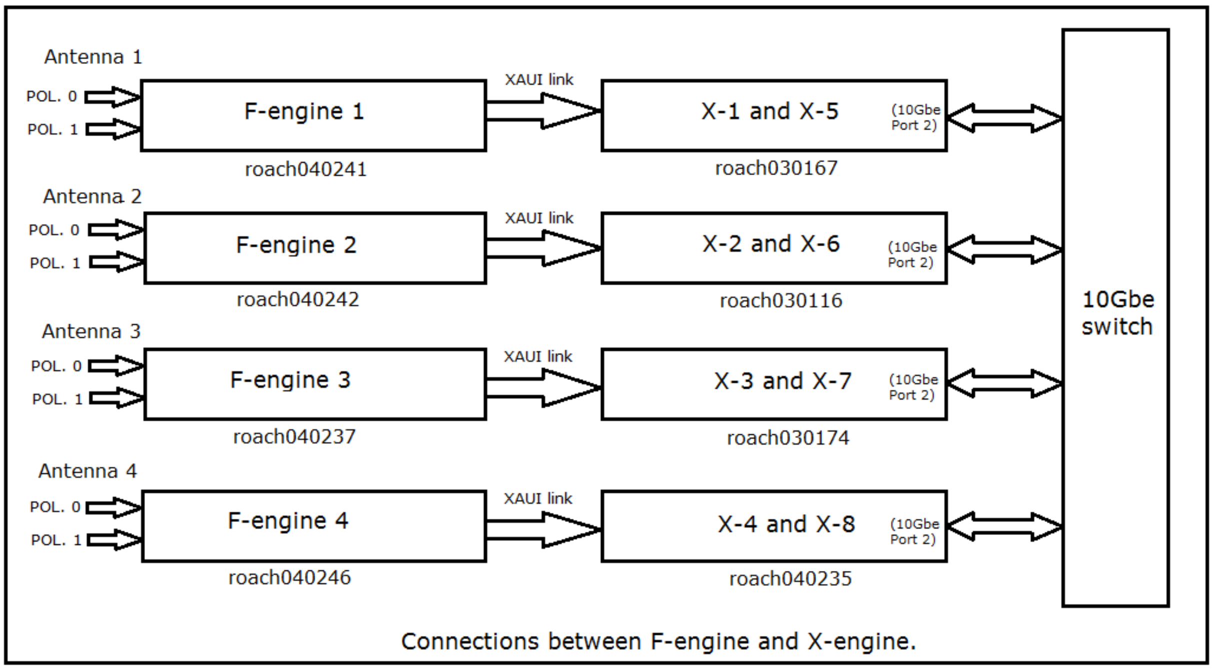
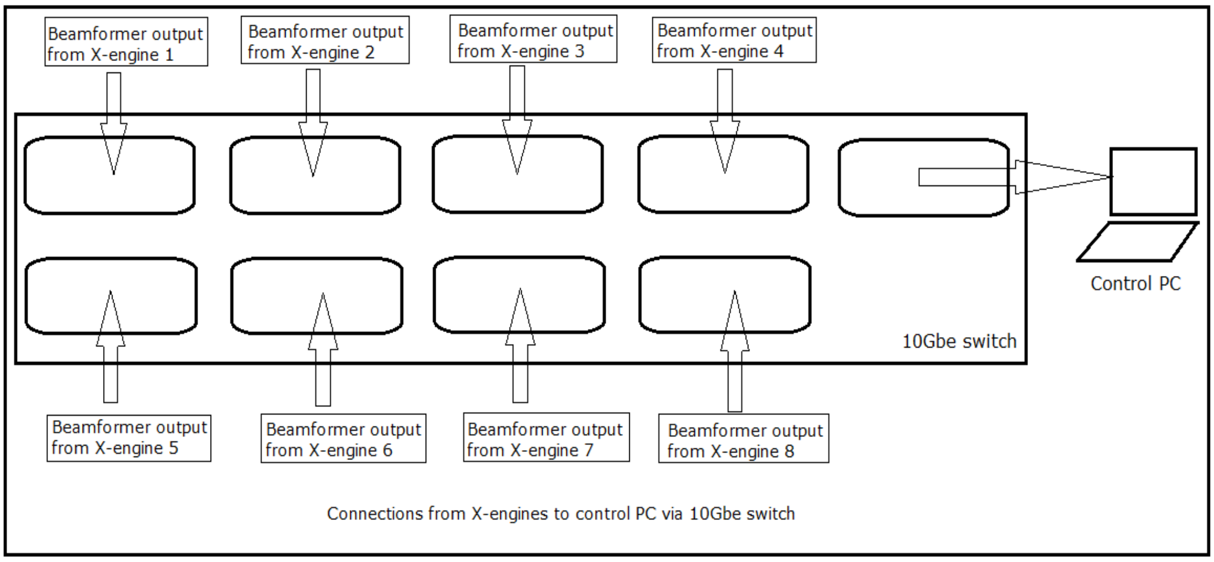


Figure 7.3 Connections between F-engine and X-engine

6) CONNECTIONS FROM X-ENGINE TO CONTROL PC:

The connection from the X-engines to control PC is made via the 10GbE switch. The following diagram shows these connections.



Beamformer output from X-engine 1
Beamformer output from X-engine 2
Beamformer output from X-engine 3
Beamformer output from X-engine 4
Control PC
10Gbe switch
Beamformer output from X-engine 5
Beamformer output from X-engine 6
Beamformer output from X-engine 7
Beamformer output from X-engine 8
Connections from X-engines to control PC via 10Gbe switch

Figure 7.4 Connections from X-engines to control PC via 10 GbE switch.

8. Testing of the Designs and Results

The BEAMFORMER_INCOH subsystem was designed in the MATLAB software using the blocks of CASPER blockset and XYLINX blockset in the simulink library.

As the first step of testing, a 16 bit counter data was given as input to the BEAMFORMER_INCOH subsystem and the results were verified by matching the results with theoreotical calculation. The simulation results are attached below.

8.1 Simulation results

Test parameters for the simulation carried out:

Input: 16 bit counter.

No. of cycles for which 2^15 accumulation is to be carried out: 3.

1) Input: The Figure 8.1 is the output of the 16 bit counter as given to the beamformer subsystem for 3 2^15 cycle. One ramp is considered as 128 timestamp data for all 64 channels in one sync cycle.

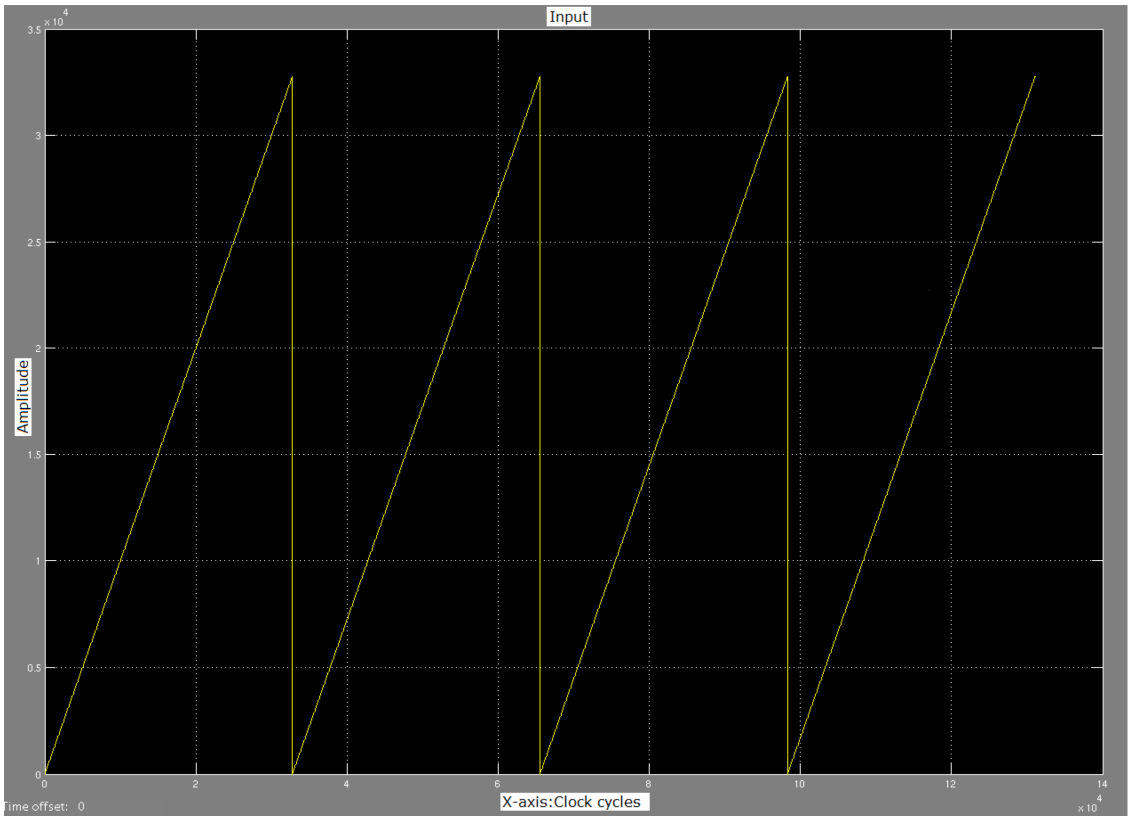


Figure 8.1 Simulation: Counter input

2) Addition of 128 timestamps from all antennas:

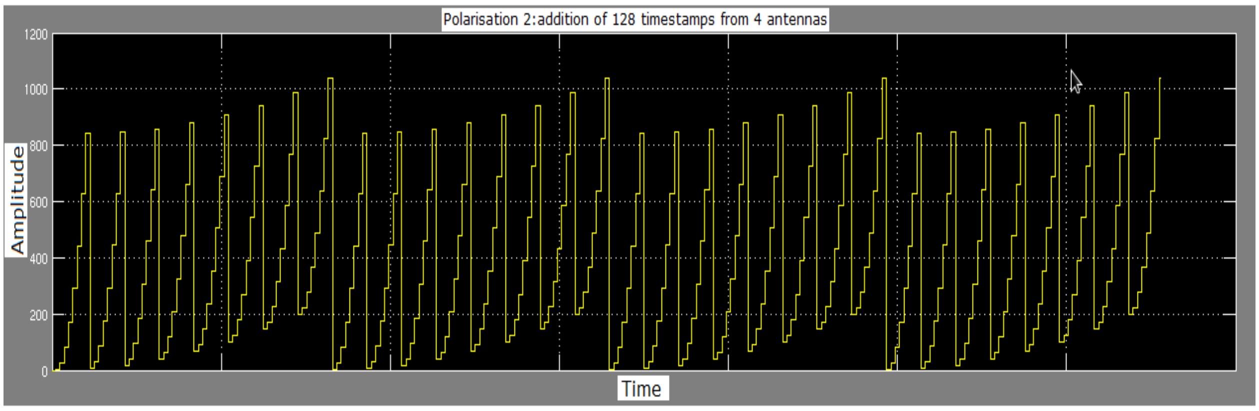


Figure 8.2 Simulation: Addition of 128 timestamps from all antennas.

3) Output of 2^15 accumulator block:

The accumulation takes place as follows. As our design is for accumulating 3 cycles of 2^15, we can see that it adds the 3 cycles of 2^15 and after that it starts new accumulation.

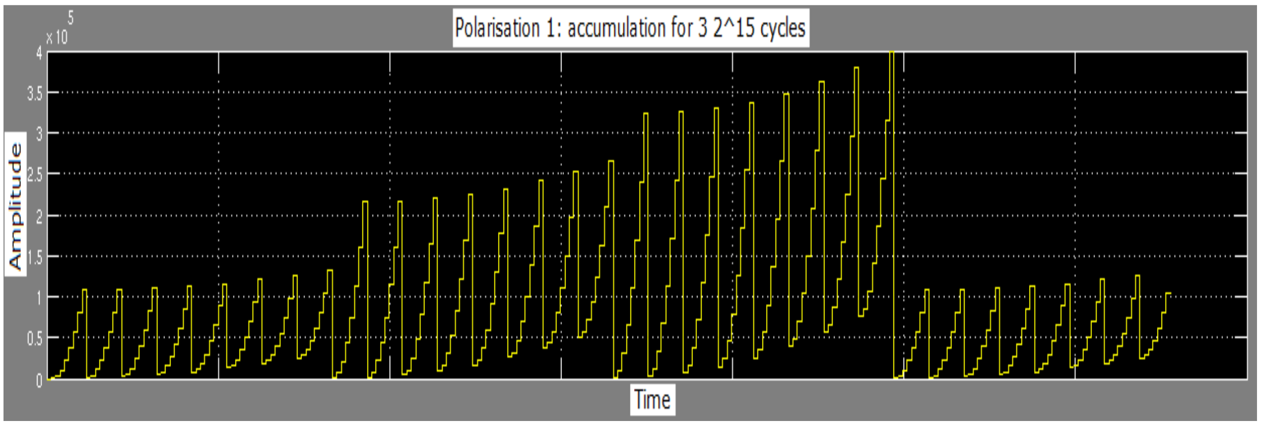


Figure 8.3: Simulation: Output of accumulator block.

4) Generation of data,tx_valid and eof for 10Gbe core:

(Refer Figure 8.4)

1.The 1st graph shows the the ouput data after concatenating the polarisation1 and polarization 0 data. Each signal corresponds to one channel data. They are 64 in number.
2. The 2nd graph shows the data transmission valid signal which is fed to Tx_valid signal of the 10GbE NIC.
3. The 3rd signal shows the end of frame signal. After this signal goes from 1 to 0, the 10GbE NIC acknowledge it as a end of one packet.

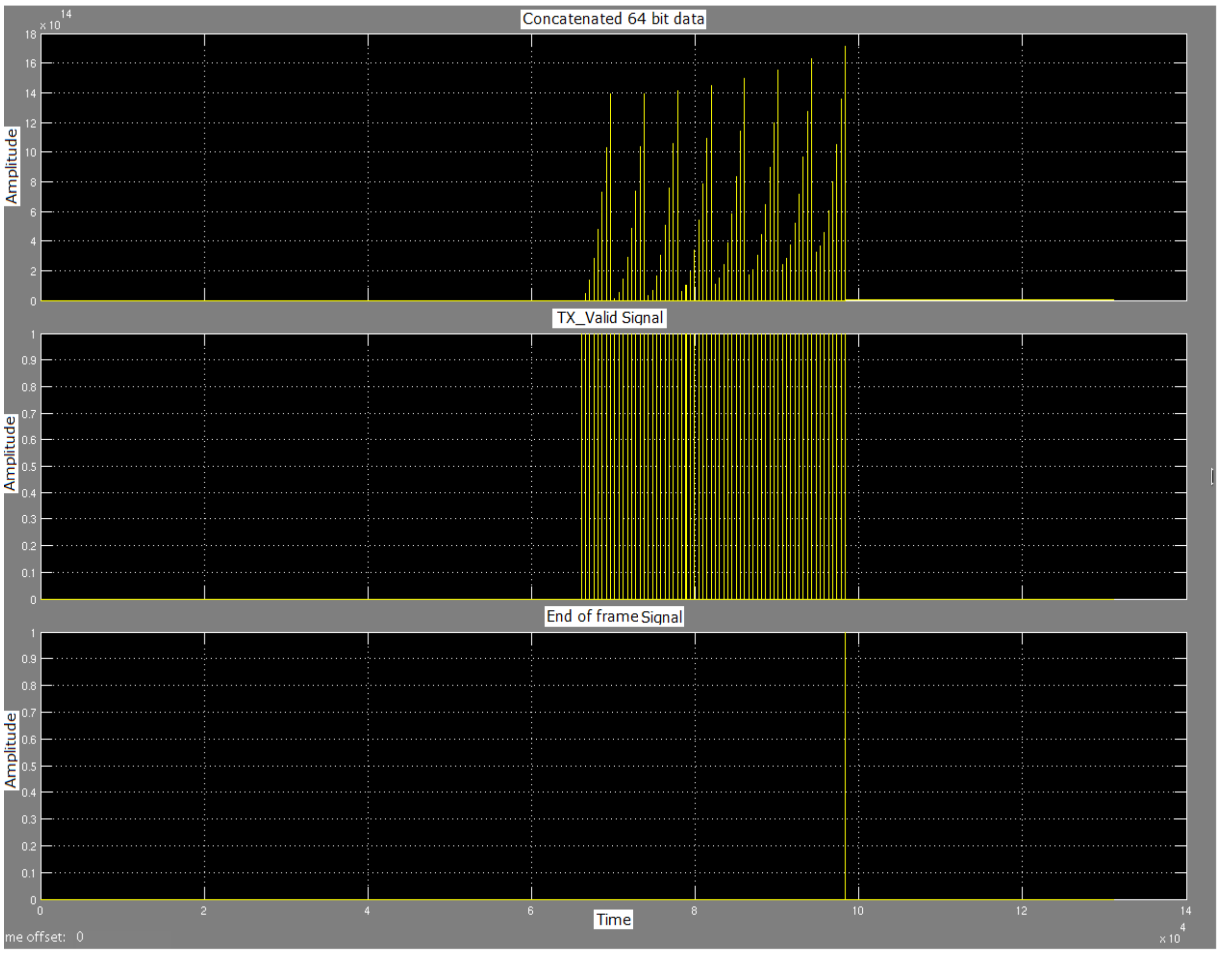


Figure 8.4 Simulation: 10GbE signals.

8.2 Sinewave test result

1.Sine wave test 1

Input: Sine wave at frequency 187.5 MHz

Roach board used: roach030167 used as X-engine.

Expected output:

The frequency 187.5 MHz belongs to channel number 30 of this X-engine. A peak is expected in the output at channel number 30 .

Interpretation of the figure:

X –axis: Channel number

Y-axis: Amplitude

A peak is present at channel number 30. No where else a peak is present. This indicates that input signal contains frequency component corresponding to channel no.30 of this particular X-engine.

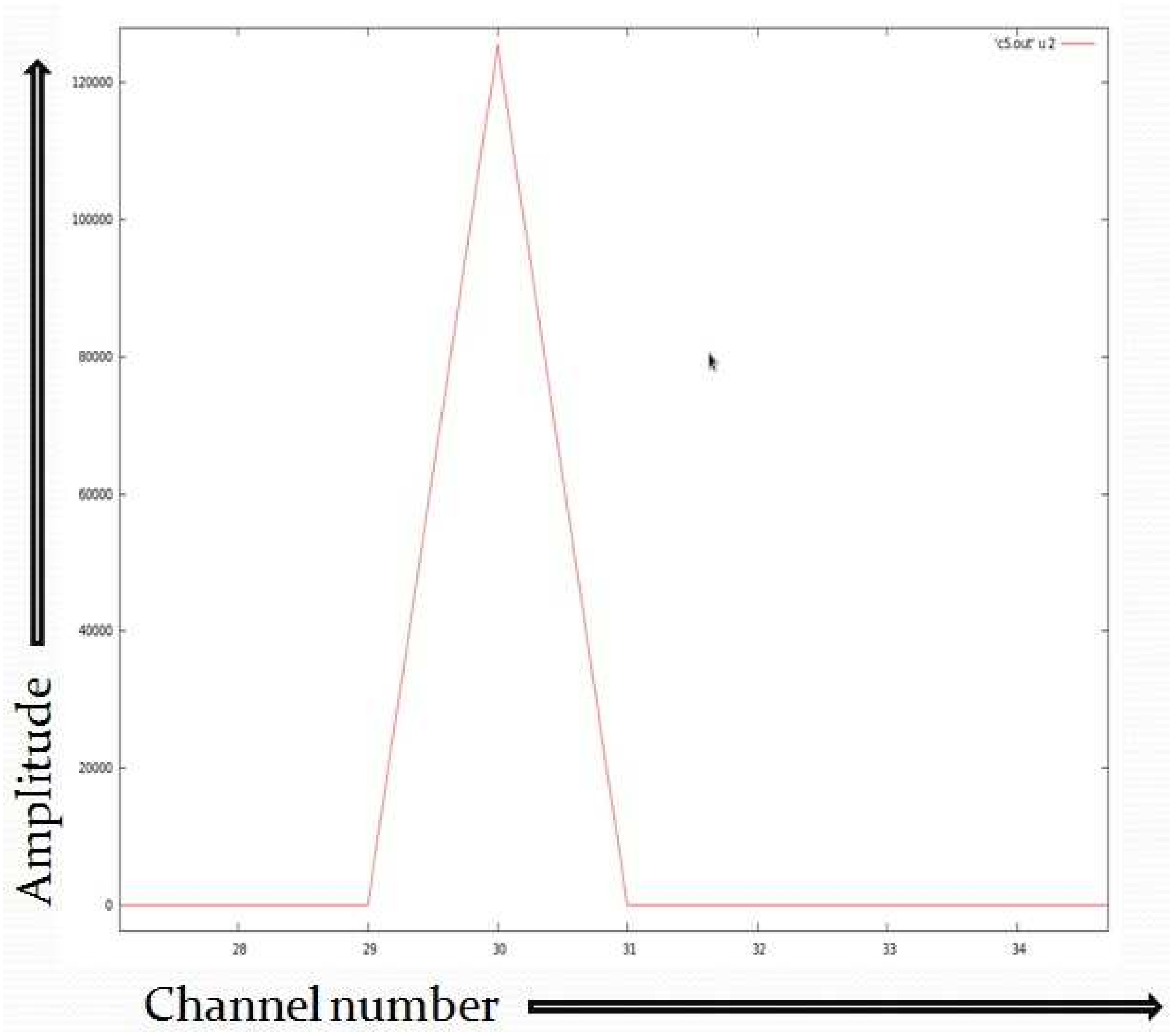


Figure 8.5: Sine wave test result 1

2.Sine wave test 2

Channel 8 in the 512 channel spectrum.

Frequency: 6.25 MHz

X-engine used: 1

X-engine channel no.:1

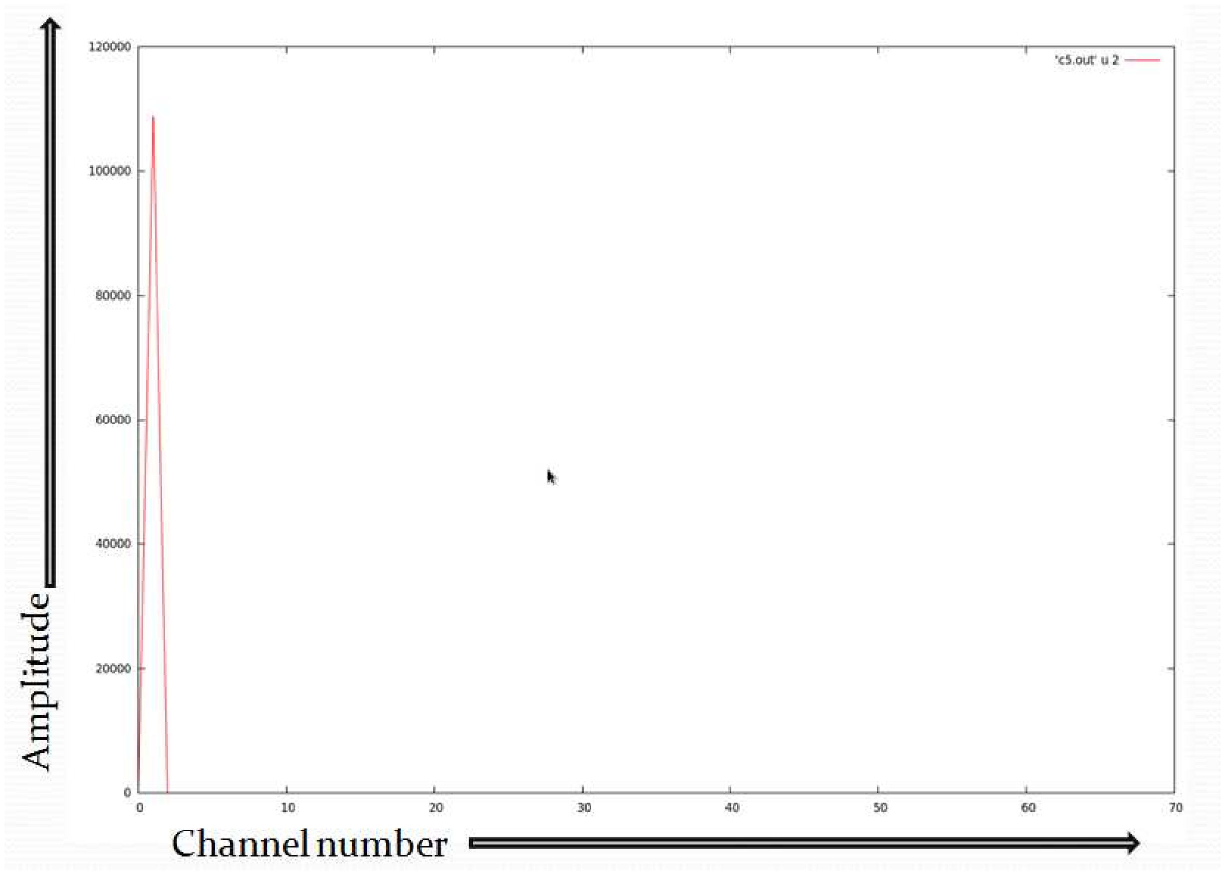


Figure 8.6: Sine wave test result 2

8.3 Interleaved data from 8 X-engines:

TEST : To check the functioning of interleaving code.

Connection:One to one connection between X-engine and control PC

Input: Sine wave frequencies belonging to each X-engines were **given one by one** as an input.

Processing: Data was captured **separately** and interleaved.

Output: Peaks observed at the respective channel numbers.

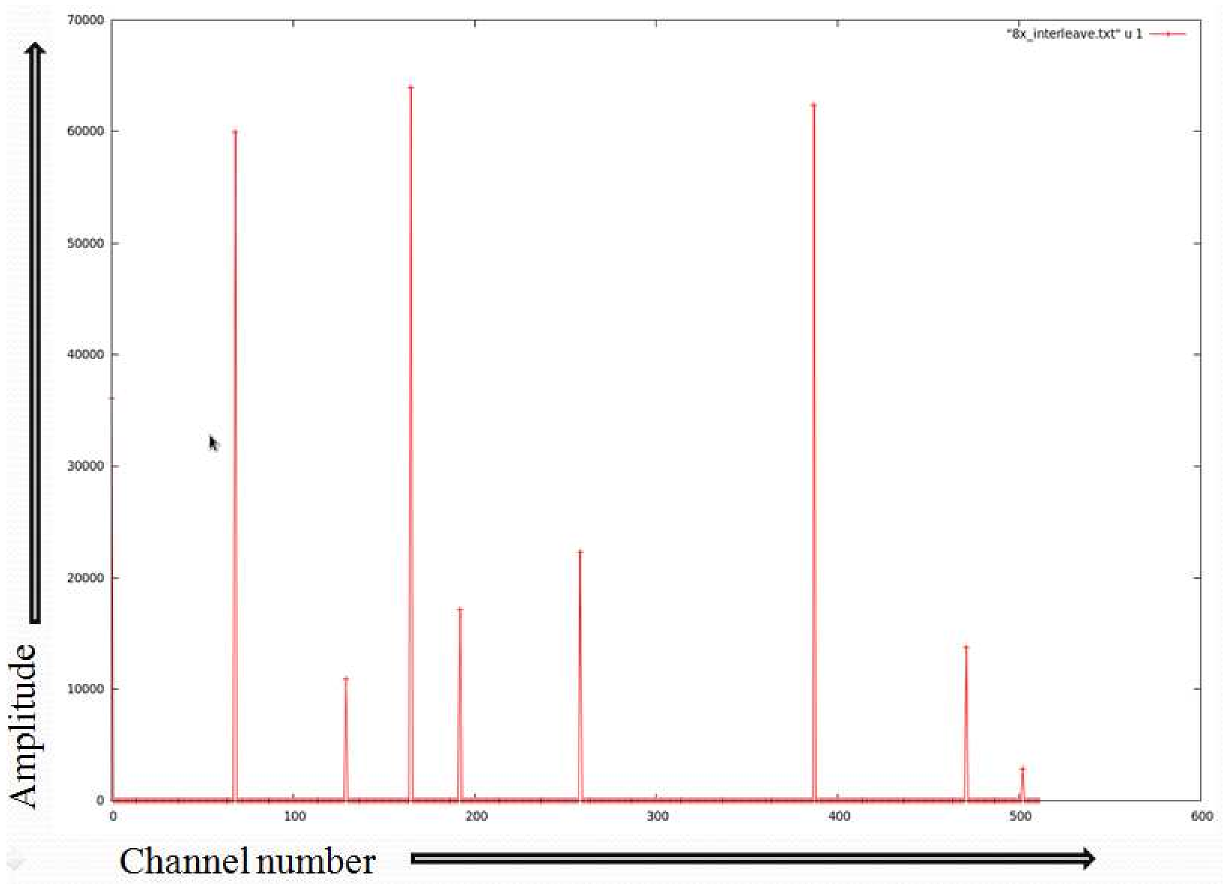


Figure 8.7: Interleaving result for 8 separate files.

8.4 Role played by data_valid

Sine wave without Data_valid:
(Refer figure 8.8)
The input and output both were shifted.
Notice the peak in output.
Input: Frequency:156.25 MHz
Channel input: 25
Output channel:26(shifted)

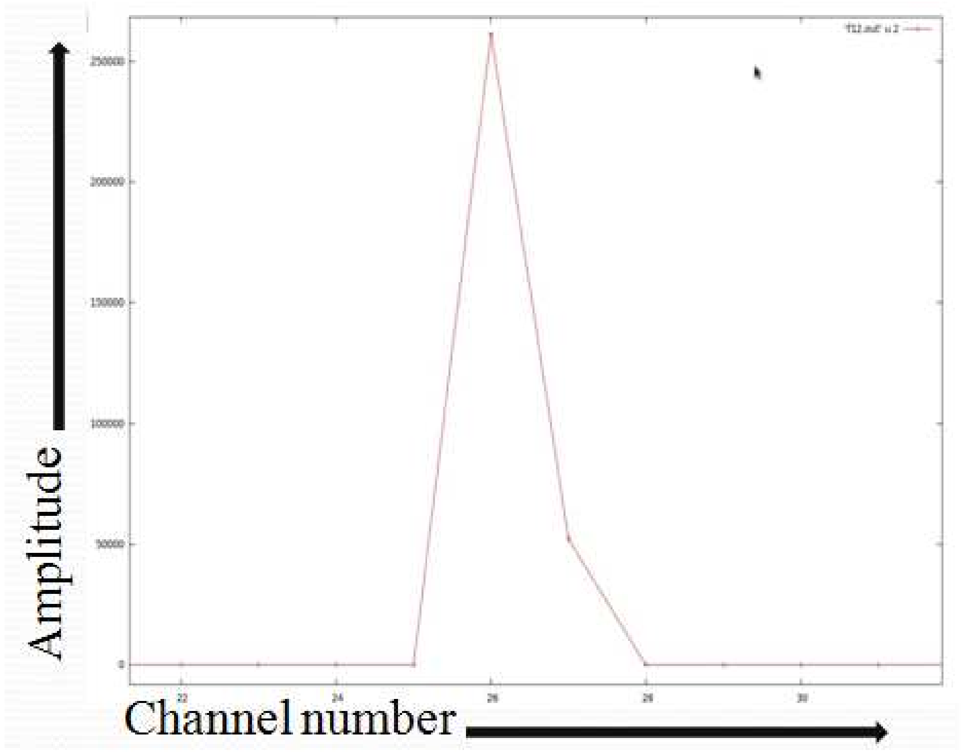


Figure 8.8: Sinewave Output-Data_valid not used.

Sine wave output with Data_valid:
(Refer figure 8.9)

The shifting is eliminated.
Peak at exact location.
Input Frequency:325 MHz
Channel input: 52
Output channel:52 (not shifted)

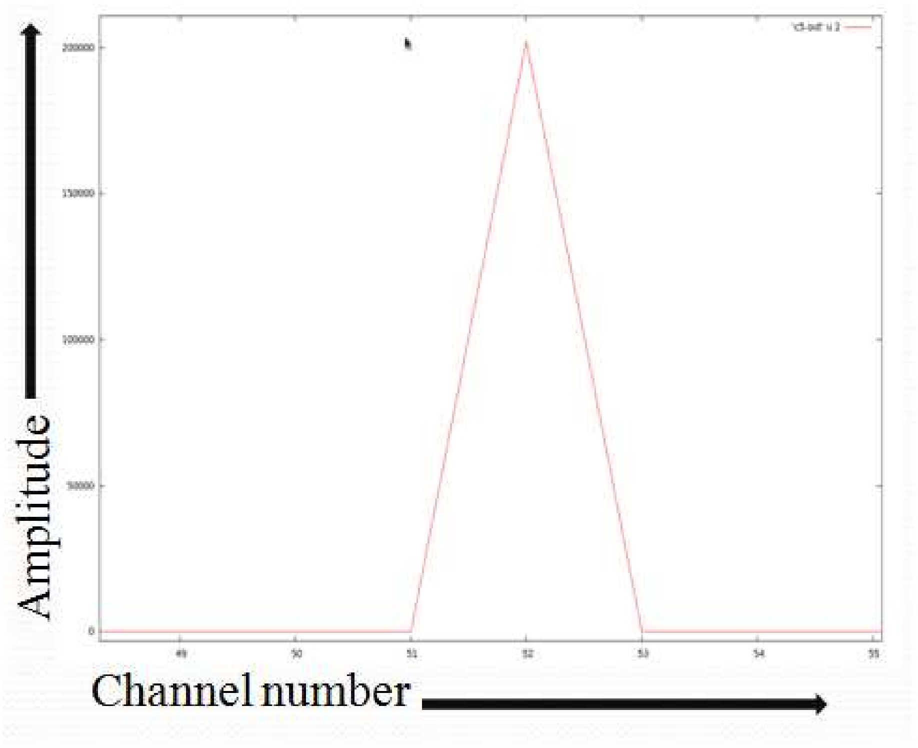


Figure 8.9: Sinewave Output: Data_valid used

8.5 Noise test results

Connections: All 8 X-engines connected to control PC via 10GbE switch

Input: Signal from noise generator passed through a low pass filter of 200 Mhz.

Output: GNU plot of the interleaved data from all X-engines

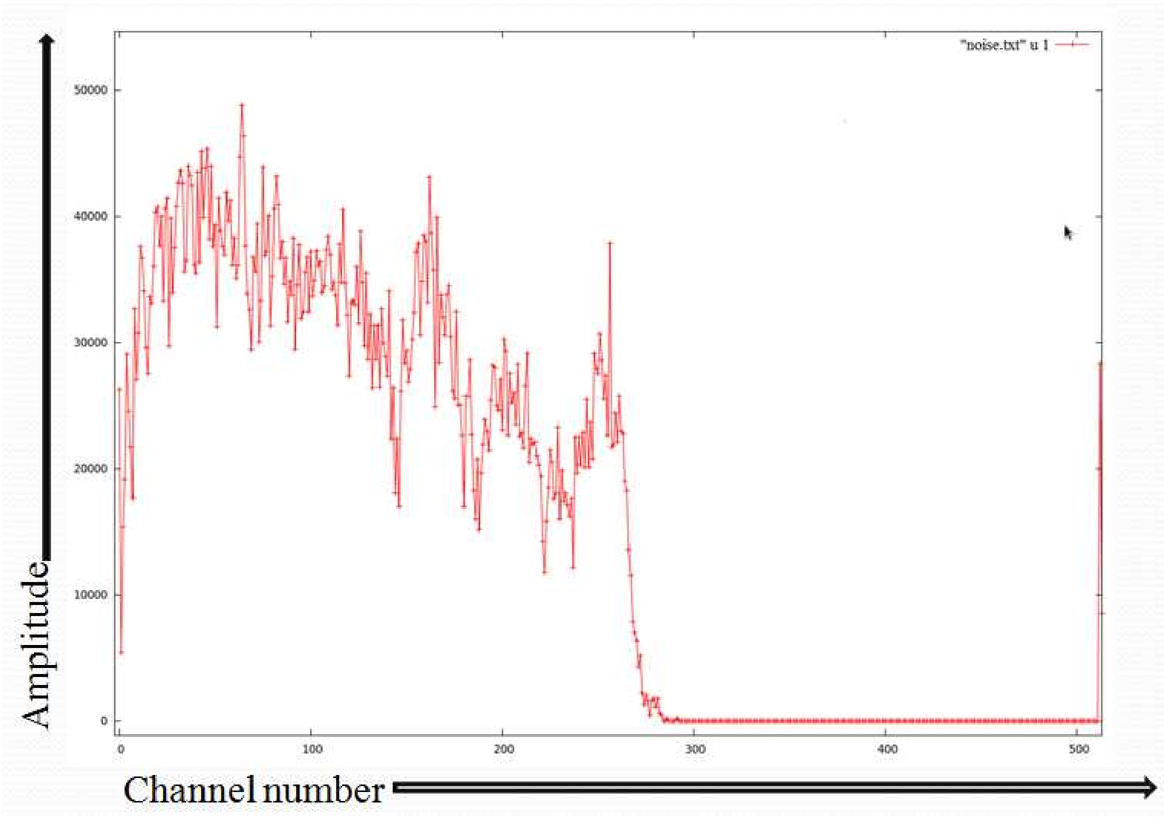


Figure 8.10. Noise Test Result-512 channel spectrum

Comaprison: Output of packetized Correlator Output and packetized Beamformer for Noise test.

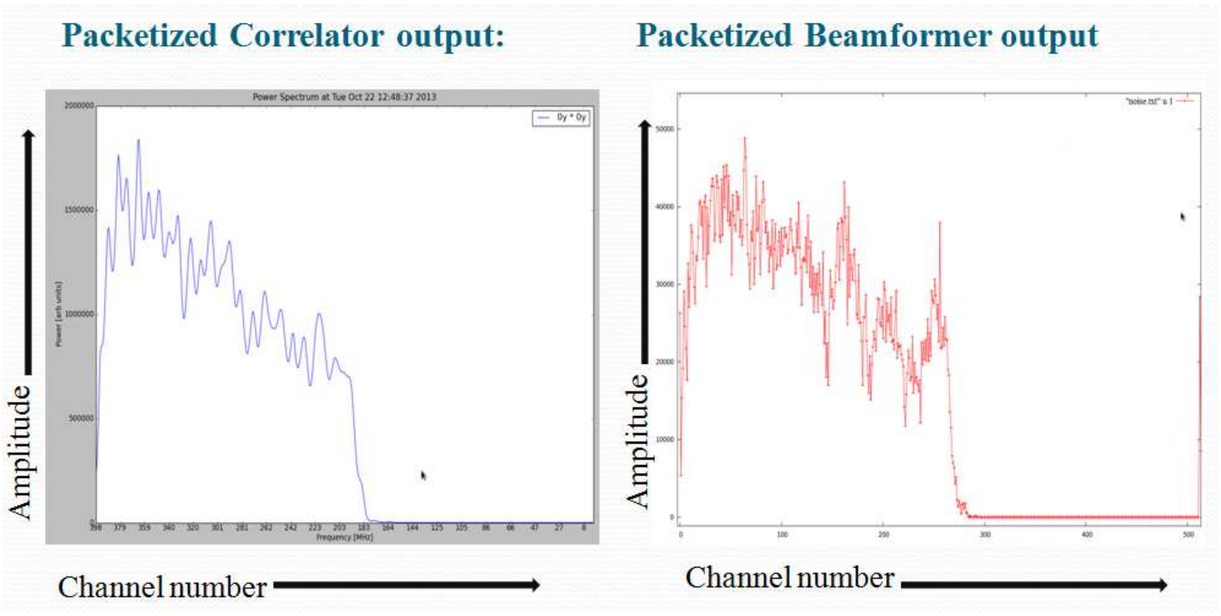


Figure 8.11 Comparison: Packetized Correlator output v/s Packetized Beamformer output

8.6 Improvement in sensitivity with increase in number of antenna

Refer figure 8.12.

The colour code is as follows:

Pink- Noise output for input given to 1 antenna.

Blue- Noise output for input given to 2 antennas.

Green- Noise output for input given to 3 antennas.

Red- Noise output for input given to 4 antennas.

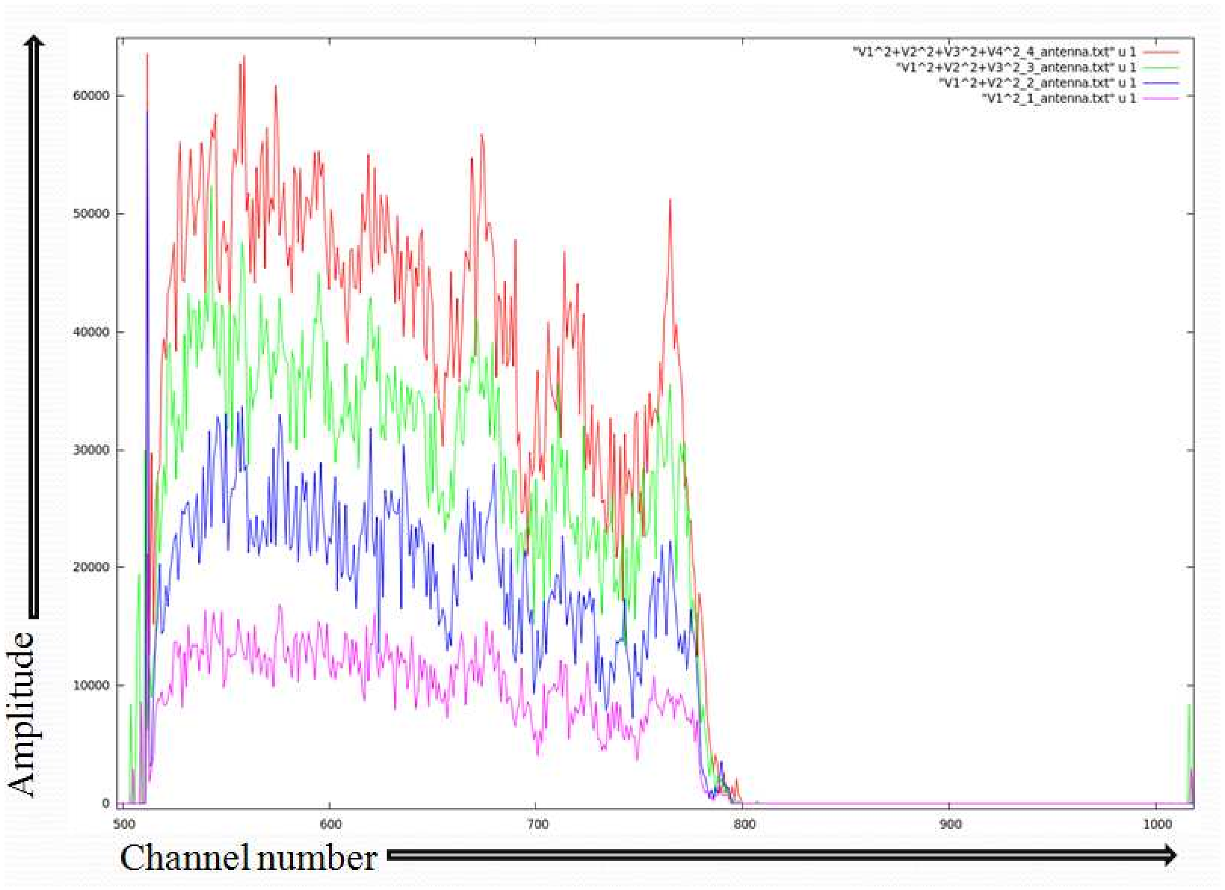


Figure 8.12. Noise test Output for Increasing Number of Antennas.

8.7 Pulsar test:

1.TEST:

- Date of observation: 30th October 2013.
- Pulsar: B0329+54(Period: 714.578196 msec)
- Antennas used: 4 central square antennas used.
- Sampling clock: 800MHz.
- Data acquisition: 5mins
- Integration time: 0.164 millisecond
- Beamformer Bandwidth: 400 MHz
- RF Bandwidth: 32 MHz
- Number of channels:512

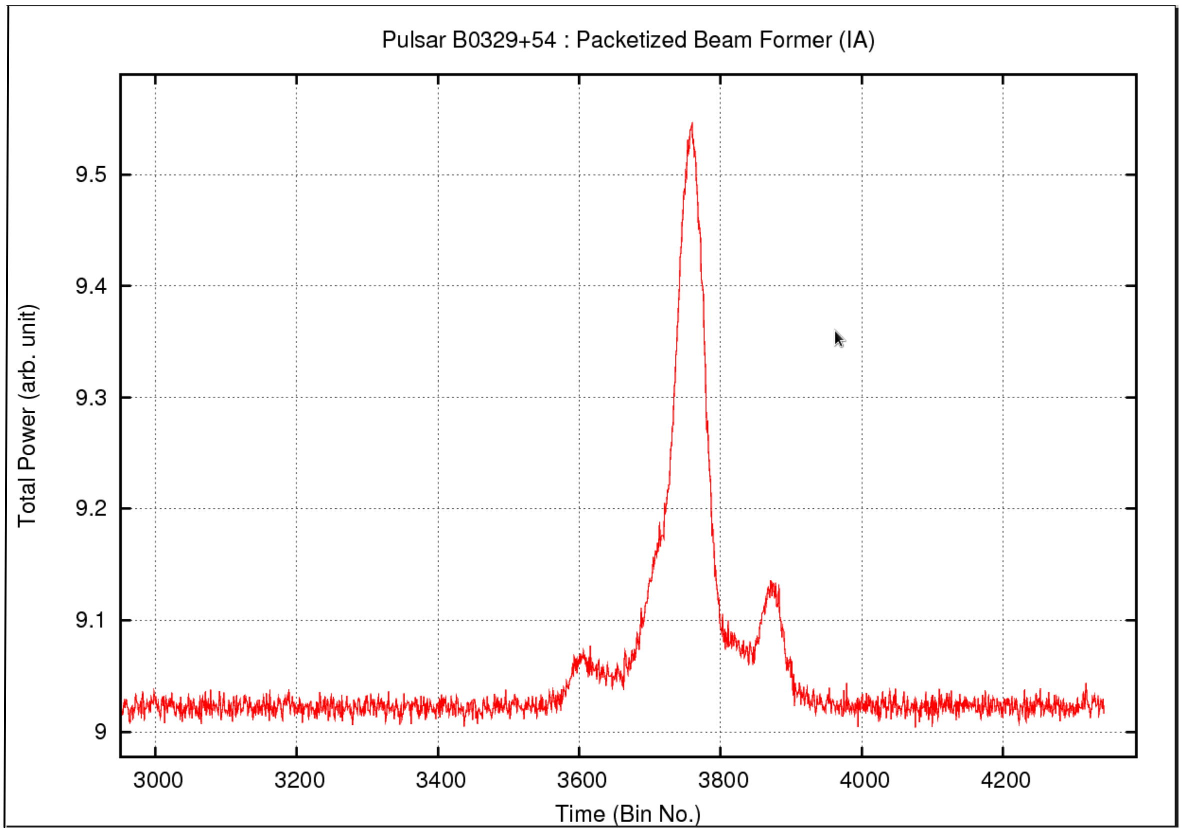


Figure 8.13 PMON Profile for Pulsar B0329+54

COMPARING PULSAR RESULT WITH THEOREOTICAL RESULT: Pulsar B0329+54

GSB Output **EPN Archive**

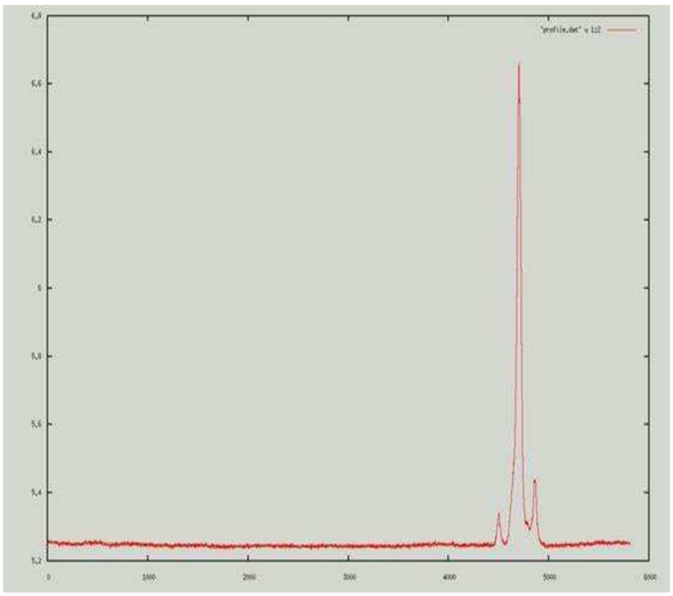


Figure 8.14:GSB output:Pulsar B0329+54

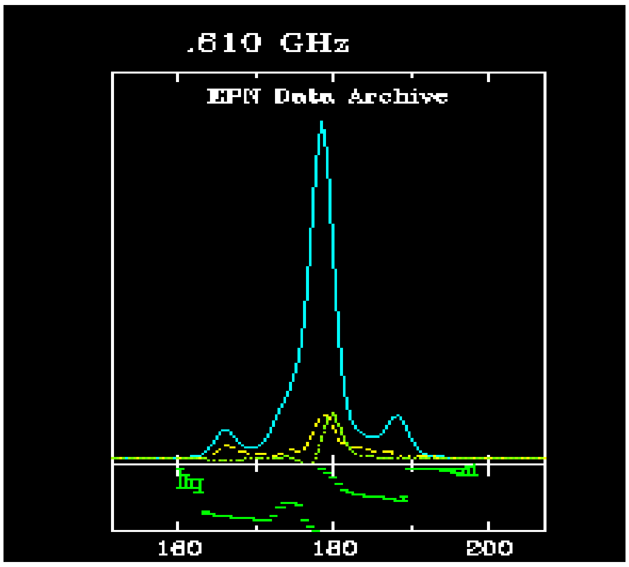
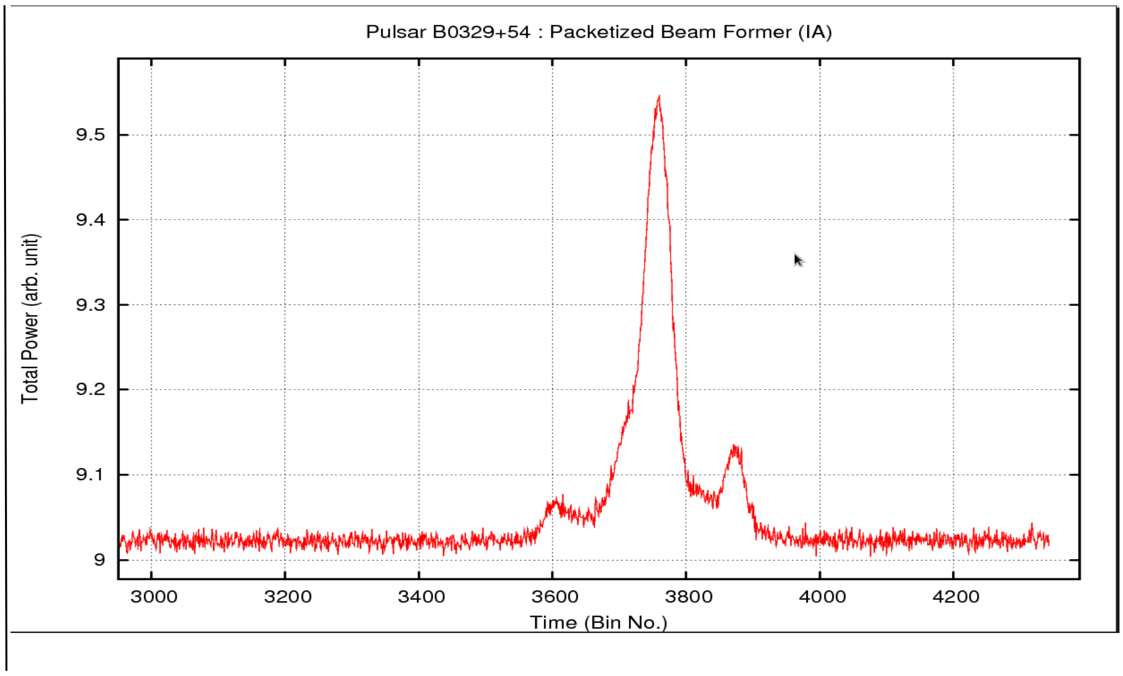


Figure 8.15:EPN Archive:PulsarB0329+54

PACKETIZED BEAMFORMER OUTPUT



2.TEST : Pulsar test at 400Mhz RF Bandwidth.

- Date of observation: 27th November 2013.
- Pulsar: B0329+54
- Antennas used: 4 central square antennas used.
- Sampling clock:800MHz.
- Integration time: 0.164 millisecond
- Beamformer Bandwidth: 400 MHz
- RF Bandwidth: 400 MHz
- Number of channels:512

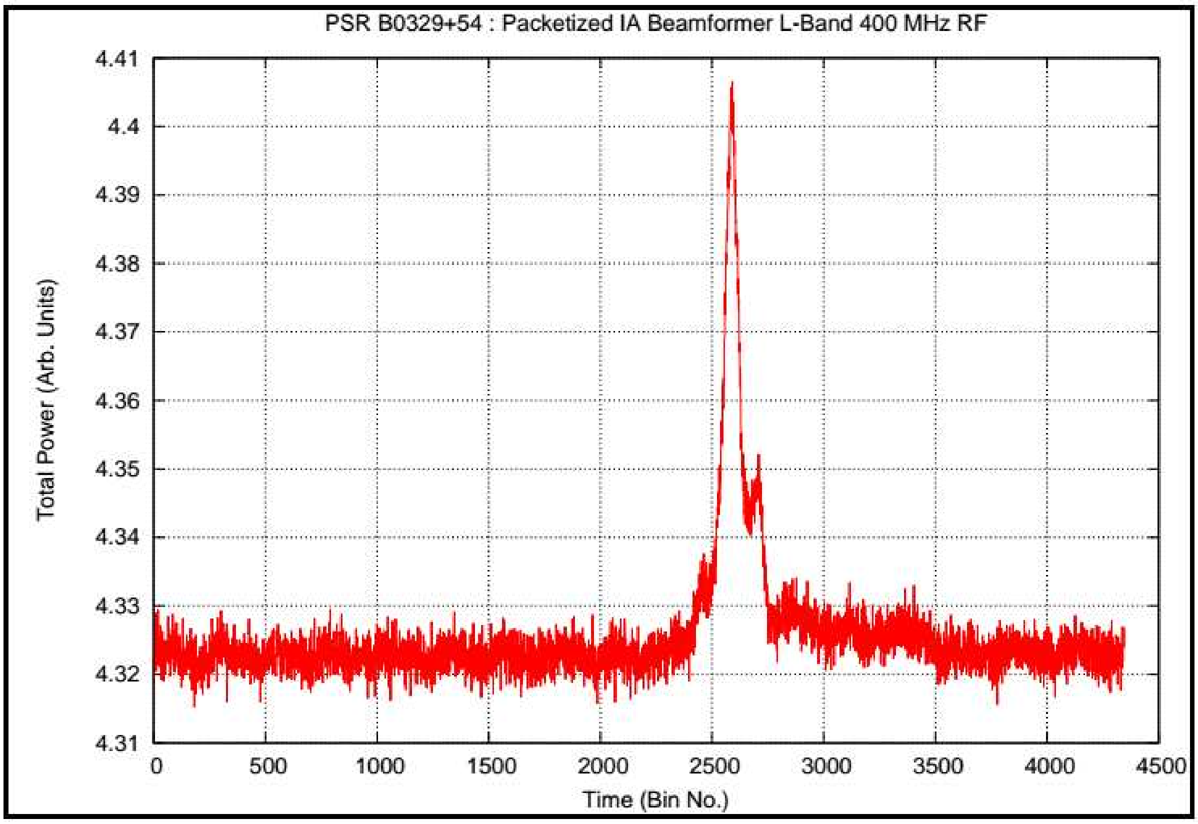


Figure 8.16 PMON Profile of Pulsar B0329+54 at 400MHz R.F. B.W.

This is the first detection through this design at full 400 MHz RF in the L-Band. Local Oscillator at 1450 MHz.

9. Add-on to the Beamformer Subsystem

This section briefly explains the add-ons that have been added to the beamformer subsystem separately.

9.1 Add-On:On-board integrator:

Purpose: Used for multiple sync cycle integration.

Working:

1) Accumulation is done in Dual Port RAM.
2) Three signals are generated: End cycle_minus 1, end_cycle plus 1 and end cycle,end_cycle_minus1_ext.
3) Registers are used to give a continuous high signal till some desired instant. These registers are enabled and reset accordingly.
4) New_ acc gen:Register reset by end_cycle plus 1, enable by end cycle 1
5) Tx_valid: Register reset by end_cycle minus1_ ext, enable by end cycle minus 1,. Output of this register is added with write enable which is input to this subsystem.
6) End of frame: counts 64 tx_valid and goes high on the 64th tx_valid.

Timing Diagram: Figure 9.1 and 9.2 illustrate the timing diagram for accumulation of 3 sync cycles.

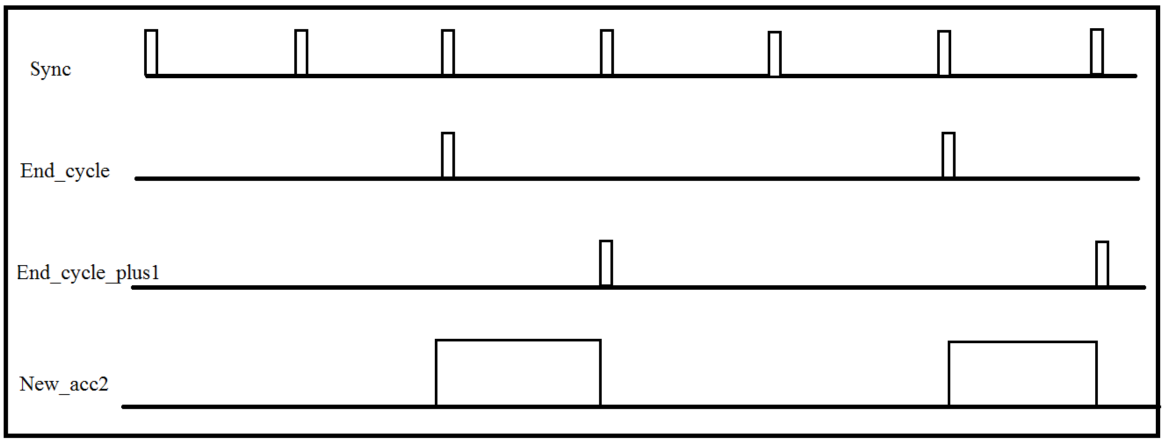


Figure 9.1 Add-on: Generation of new_acc signal

New_acc signal should be high for the first sync cycle of every accumulation as it is the select signal to the MUX that selects the second input to the adder.

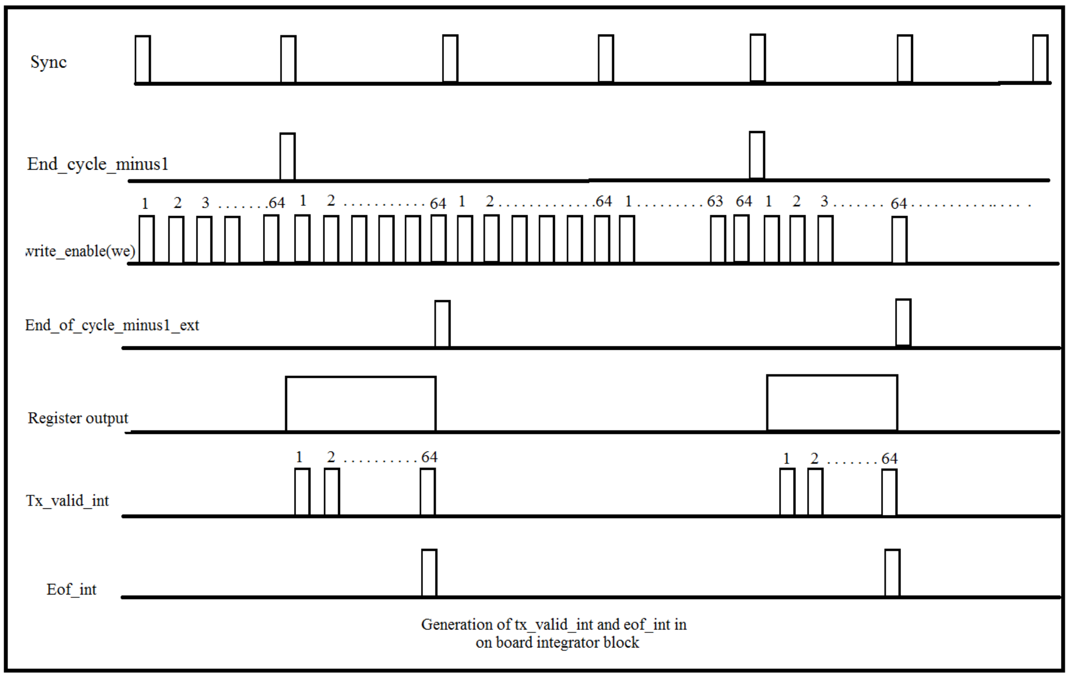


Figure 9.2 Add-on: Generation of Tx-valid and end of frame.

64 tx_valid should come in the last sync cycle of every accumulation and end of frame should go high on every 64th tx_valid.

Status: Design is compiled and tested for each X-engine separately. Results are as expected.

Result: Figure 9.3 shows noise test results for 10 integration cycle(green) and 100 integration cycle(red). The results are only for 1 X-engine.

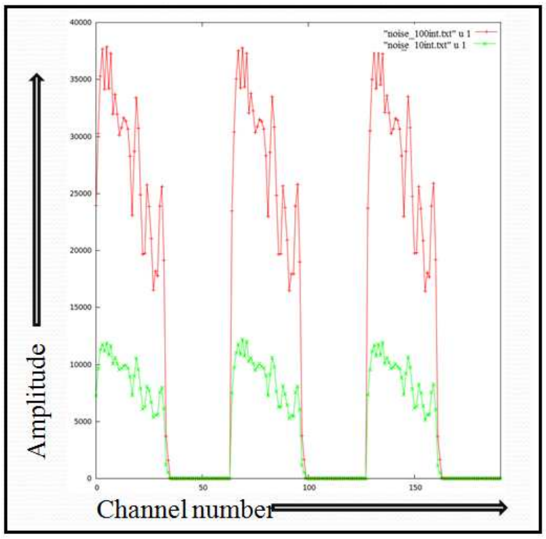


Figure 9.3 Add-on: Multiple Integration Result.

9.2 Add-on:Packet counter:

Purpose: Time synchronized interleaving of packets.

Working: A packet count is added to the data packet after the values for all 64 channels of that X-engine have arrived at the data input of the 10GbE v2 block of that particular X-engine.

Data-packet: It will now be of 520 bytes as 8 byte packet counter is added to the packet.

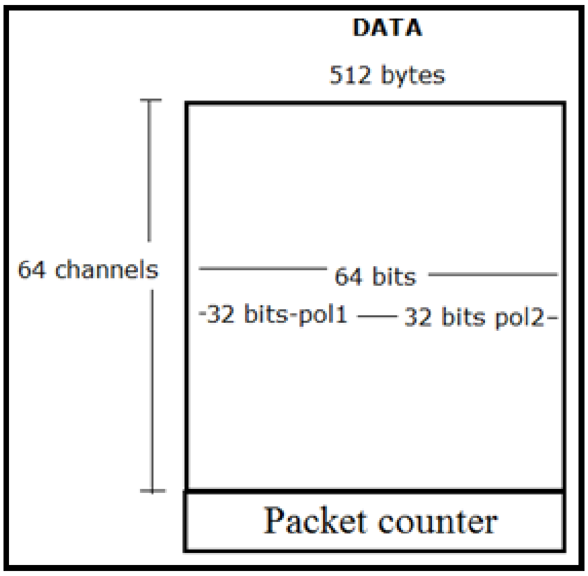


Figure 9.4 Data packet with packet counter

The UDP packet size will be 520 bytes(data)+42 bytes(header) i.e. 562 bytes.

Timing Diagram: Figure 9.5 shows the timing diagram for 10 GbE signals when a packet count is transmitted along with data.

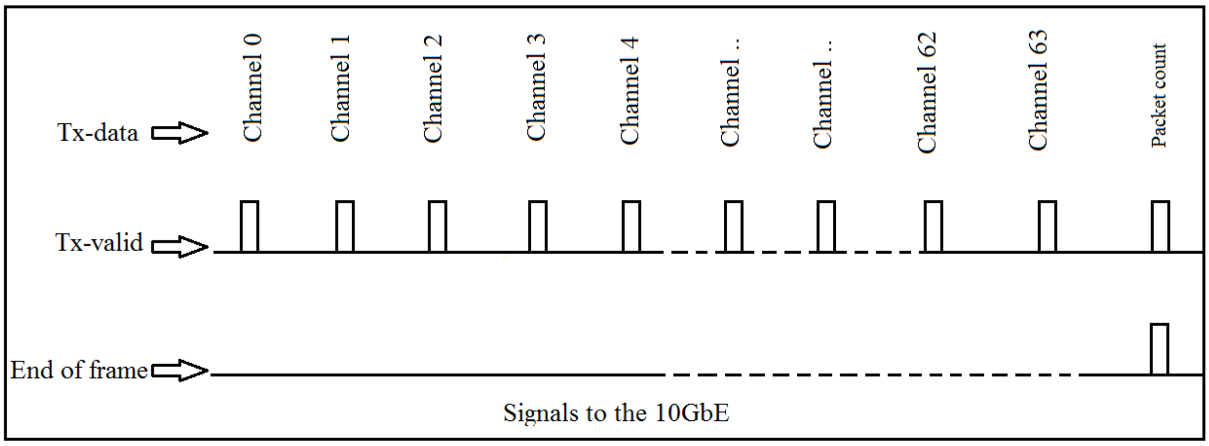


Figure 9.5 Add-on: 10 GbE signals for packet counter

An extra tx_valid is generated for the packet count value and the end of frame goes high with this extra tx_valid. In this case, 65 tx_valids will be there.

Status: Design checked in simulation. Size of data packet checked in wireshark. It is found to be 520 bytes.

10. Future work and recommendations

- Scale the design to 8 antennas
- Scale the design for greater number of channels
- Attempt time synchronized interleaving using packet counter logic. Develop Post-processing scripts for the same.
- Attempt Sync cycle integration for all 8 X-engines together.
- Analysis of relative improvement in SNR as a function of number of antennas.

11. References

1. Jayaram N Chengalur, Yashwant Gupta, K S Dwarkanath, ―*Low Frequency Radio Astronomy*", a note from school on radio astronomy held at NCRA, Pune , 1999.

2.Casper website, *https://casper.berkeley.edu/*

3. Casper tutorials on ROACH board,

a. On Introduction to Simulink, *https://casper.berkeley.edu/wiki/Introduction_to_Simulink*
b. On The 10GbE Interface, *https://casper.berkeley.edu/wiki/Tutorial_10GbE*
c. On the Wideband Pocket Correlator, *https://github.com/casperastro/tutorials_devel/tree/master/workshop_2010/roach_tut4_wideband_poco*

4.Cambodge bist, ― "*Pocket Beamformer on FPGA*" , student project report, NCRA,TIFR

5.Ankur Verma, Subhajit Majumder, ―"*Designing of Coherent Pocket Beamforemer on FPGA*", student project report, NCRA,TIFR

6. Pulsar Period Simulator, *http://astro.unl.edu/classaction/animations/extrasolarplanets/pulsarPeriodSim001.html*

Appendix A

APPENDIX A-1

This is a list of the Frequencies that belong to X-engine1 and X-engine 2.

Each X-engine processes 64 channels.

Column 1 shows the Channel in the respective X-engine out of the 64 channels.

Column 3 shows the Channel number for X-engine 1 out of the total 512 channels.

Column 4 shows the Frequencies accepted by X-engine 1 in MHz.

Column 6 shows the Channel number for X-engine 2 out of the total 512 channels.

Column 7 shows the Frequencies accepted by X-engine 2 in MHz.

Channel		X ENGINE 1	FREQUENCY		X ENGINE 2	FREQUENCY
			(MHZ)			(MHZ)
0		0	0.00		1	0.78125
1		8	6.25		9	7.03125
2		16	12.50		17	13.28125
3		24	18.75		25	19.53125
4		32	25.00		33	25.78125
5		40	31.25		41	32.03125
6		48	37.50		49	38.28125
7		56	43.75		57	44.53125
8		64	50.00		65	50.78125
9		72	56.25		73	57.03125
10		80	62.50		81	63.28125
11		88	68.75		89	69.53125
12		96	75.00		97	75.78125
13		104	81.25		105	82.03125
14		112	87.50		113	88.28125
15		120	93.75		121	94.53125
16		128	100.00		129	100.78125
17		136	106.25		137	107.03125
18		144	112.50		145	113.28125
19		152	118.75		153	119.53125
20		160	125.00		161	125.78125
21		168	131.25		169	132.03125

22		176	137.50		177	138.28125
23		184	143.75		185	144.53125
24		192	150.00		193	150.78125
25		200	156.25		201	157.03125
26		208	162.50		209	163.28125
27		216	168.75		217	169.53125
28		224	175.00		225	175.78125
29		232	181.25		233	182.03125
30		240	187.50		241	188.28125
31		248	193.75		249	194.53125
32		256	200.00		257	200.78125
33		264	206.25		265	207.03125
34		272	212.50		273	213.28125
35		280	218.75		281	219.53125
36		288	225.00		289	225.78125
37		296	231.25		297	232.03125
38		304	237.50		305	238.28125
39		312	243.75		313	244.53125
40		320	250.00		321	250.78125
41		328	256.25		329	257.03125
42		336	262.50		337	263.28125
43		344	268.75		345	269.53125
44		352	275.00		353	275.78125
45		360	281.25		361	282.03125
46		368	287.50		369	288.28125
47		376	293.75		377	294.53125
48		384	300.00		385	300.78125
49		392	306.25		393	307.03125
50		400	312.50		401	313.28125
51		408	318.75		409	319.53125
52		416	325.00		417	325.78125
53		424	331.25		425	332.03125
54		432	337.50		433	338.28125
55		440	343.75		441	344.53125
56		448	350.00		449	350.78125
57		456	356.25		457	357.03125
58		464	362.50		465	363.28125
59		472	368.75		473	369.53125
60		480	375.00		481	375.78125
61		488	381.25		489	382.03125
62		496	387.50		497	388.28125

63		504	393.75		505	394.53125

APPENDIX A-2

This is a list of the Frequencies that belong to X-engine 3 and X-engine 4.

Each X-engine processes 64 channels.

Column 1 shows the Channel in the respective X-engine out of the 64 channels.

Column 3 shows the Channel number for X-engine 3 out of the total 512 channels.

Column 4 shows the Frequencies accepted by X-engine 3 in MHz.

Column 6 shows the Channel number for X-engine 4 out of the total 512 channels.

Column 7 shows the Frequencies accepted by X-engine 4 in MHz.

Channel		X ENGINE 3	FREQUENCY		X ENGINE 4	FREQUENCY
			(MHZ)			(MHZ)
0		2	1.5625		3	2.34375
1		10	7.8125		11	8.59375
2		18	14.0625		19	14.84375
3		26	20.3125		27	21.09375
4		34	26.5625		35	27.34375
5		42	32.8125		43	33.59375
6		50	39.0625		51	39.84375
7		58	45.3125		59	46.09375
8		66	51.5625		67	52.34375
9		74	57.8125		75	58.59375
10		82	64.0625		83	64.84375
11		90	70.3125		91	71.09375
12		98	76.5625		99	77.34375
13		106	82.8125		107	83.59375
14		114	89.0625		115	89.84375
15		122	95.3125		123	96.09375
16		130	101.5625		131	102.34375
17		138	107.8125		139	108.59375
18		146	114.0625		147	114.84375
19		154	120.3125		155	121.09375
20		162	126.5625		163	127.34375
21		170	132.8125		171	133.59375
22		178	139.0625		179	139.84375
23		186	145.3125		187	146.09375

24		194	151.5625		195	152.34375
25		202	157.8125		203	158.59375
26		210	164.0625		211	164.84375
27		218	170.3125		219	171.09375
28		226	176.5625		227	177.34375
29		234	182.8125		235	183.59375
30		242	189.0625		243	189.84375
31		250	195.3125		251	196.09375
32		258	201.5625		259	202.34375
33		266	207.8125		267	208.59375
34		274	214.0625		275	214.84375
35		282	220.3125		283	221.09375
36		290	226.5625		291	227.34375
37		298	232.8125		299	233.59375
38		306	239.0625		307	239.84375
39		314	245.3125		315	246.09375
40		322	251.5625		323	252.34375
41		330	257.8125		331	258.59375
42		338	264.0625		339	264.84375
43		346	270.3125		347	271.09375
44		354	276.5625		355	277.34375
45		362	282.8125		363	283.59375
46		370	289.0625		371	289.84375
47		378	295.3125		379	296.09375
48		386	301.5625		387	302.34375
49		394	307.8125		395	308.59375
50		402	314.0625		403	314.84375
51		410	320.3125		411	321.09375
52		418	326.5625		419	327.34375
53		426	332.8125		427	333.59375
54		434	339.0625		435	339.84375
55		442	345.3125		443	346.09375
56		450	351.5625		451	352.34375
57		458	357.8125		459	358.59375
58		466	364.0625		467	364.84375
59		474	370.3125		475	371.09375
60		482	376.5625		483	377.34375
61		490	382.8125		491	383.59375
62		498	389.0625		499	389.84375
63		506	395.3125		507	396.09375

APPENDIX A-3

This is a list of the Frequencies that belong to X-engine 5 and X-engine 6.

Each X-engine processes 64 channels.

Column 1 shows the Channel in the respective X-engine out of the 64 channels.

Column 3 shows the Channel number for X-engine 5 out of the total 512 channels.

Column 4 shows the Frequencies accepted by X-engine 5 in MHz.

Column 6 shows the Channel number for X-engine 6 out of the total 512 channels.

Column 7 shows the Frequencies accepted by X-engine 6 in MHz.

Channel		X ENGINE 5	FREQUENCY		X ENGINE 6	FREQUENCY
			(MHZ)			(MHZ)
0		4	3.125		5	3.90625
1		12	9.375		13	10.15625
2		20	15.625		21	16.40625
3		28	21.875		29	22.65625
4		36	28.125		37	28.90625
5		44	34.375		45	35.15625
6		52	40.625		53	41.40625
7		60	46.875		61	47.65625
8		68	53.125		69	53.90625
9		76	59.375		77	60.15625
10		84	65.625		85	66.40625
11		92	71.875		93	72.65625
12		100	78.125		101	78.90625
13		108	84.375		109	85.15625
14		116	90.625		117	91.40625
15		124	96.875		125	97.65625
16		132	103.125		133	103.90625
17		140	109.375		141	110.15625
18		148	115.625		149	116.40625
19		156	121.875		157	122.65625
20		164	128.125		165	128.90625
21		172	134.375		173	135.15625
22		180	140.625		181	141.40625
23		188	146.875		189	147.65625

24		196	153.125		197	153.90625
25		204	159.375		205	160.15625
26		212	165.625		213	166.40625
27		220	171.875		221	172.65625
28		228	178.125		229	178.90625
29		236	184.375		237	185.15625
30		244	190.625		245	191.40625
31		252	196.875		253	197.65625
32		260	203.125		261	203.90625
33		268	209.375		269	210.15625
34		276	215.625		277	216.40625
35		284	221.875		285	222.65625
36		292	228.125		293	228.90625
37		300	234.375		301	235.15625
38		308	240.625		309	241.40625
39		316	246.875		317	247.65625
40		324	253.125		325	253.90625
41		332	259.375		333	260.15625
42		340	265.625		341	266.40625
43		348	271.875		349	272.65625
44		356	278.125		357	278.90625
45		364	284.375		365	285.15625
46		372	290.625		373	291.40625
47		380	296.875		381	297.65625
48		388	303.125		389	303.90625
49		396	309.375		397	310.15625
50		404	315.625		405	316.40625
51		412	321.875		413	322.65625
52		420	328.125		421	328.90625
53		428	334.375		429	335.15625
54		436	340.625		437	341.40625
55		444	346.875		445	347.65625
56		452	353.125		453	353.90625
57		460	359.375		461	360.15625
58		468	365.625		469	366.40625
59		476	371.875		477	372.65625
60		484	378.125		485	378.90625
61		492	384.375		493	385.15625
62		500	390.625		501	391.40625
63		508	396.875		509	397.65625

APPENDIX A-4

This is a list of the Frequencies that belong to X-engine 7 and X-engine 8.

Each X-engine processes 64 channels.

Column 1 shows the Channel in the respective X-engine out of the 64 channels.

Column 3 shows the Channel number for X-engine 7 out of the total 512 channels.

Column 4 shows the Frequencies accepted by X-engine 7 in MHz.

Column 6 shows the Channel number for X-engine 8 out of the total 512 channels.

Column 7 shows the Frequencies accepted by X-engine 8 in MHz.

Channel		X ENGINE 7	FREQUENCY		X ENGINE 8	FREQUENCY
			(MHZ)			(MHZ)
0		6	4.68750		7	5.46875
1		14	10.93750		15	11.71875
2		22	17.18750		23	17.96875
3		30	23.43750		31	24.21875
4		38	29.68750		39	30.46875
5		46	35.93750		47	36.71875
6		54	42.18750		55	42.96875
7		62	48.43750		63	49.21875
8		70	54.68750		71	55.46875
9		78	60.93750		79	61.71875
10		86	67.18750		87	67.96875
11		94	73.43750		95	74.21875
12		102	79.68750		103	80.46875
13		110	85.93750		111	86.71875
14		118	92.18750		119	92.96875
15		126	98.43750		127	99.21875
16		134	104.68750		135	105.46875
17		142	110.93750		143	111.71875
18		150	117.18750		151	117.96875
19		158	123.43750		159	124.21875
20		166	129.68750		167	130.46875
21		174	135.93750		175	136.71875
22		182	142.18750		183	142.96875
23		190	148.43750		191	149.21875

24		198	154.68750		199	155.46875
25		206	160.93750		207	161.71875
26		214	167.18750		215	167.96875
27		222	173.43750		223	174.21875
28		230	179.68750		231	180.46875
29		238	185.93750		239	186.71875
30		246	192.18750		247	192.96875
31		254	198.43750		255	199.21875
32		262	204.68750		263	205.46875
33		270	210.93750		271	211.71875
34		278	217.18750		279	217.96875
35		286	223.43750		287	224.21875
36		294	229.68750		295	230.46875
37		302	235.93750		303	236.71875
38		310	242.18750		311	242.96875
39		318	248.43750		319	249.21875
40		326	254.68750		327	255.46875
41		334	260.93750		335	261.71875
42		342	267.18750		343	267.96875
43		350	273.43750		351	274.21875
44		358	279.68750		359	280.46875
45		366	285.93750		367	286.71875
46		374	292.18750		375	292.96875
47		382	298.43750		383	299.21875
48		390	304.68750		391	305.46875
49		398	310.93750		399	311.71875
50		406	317.18750		407	317.96875
51		414	323.43750		415	324.21875
52		422	329.68750		423	330.46875
53		430	335.93750		431	336.71875
54		438	342.18750		439	342.96875
55		446	348.43750		447	349.21875
56		454	354.68750		455	355.46875
57		462	360.93750		463	361.71875
58		470	367.18750		471	367.96875
59		478	373.43750		479	374.21875
60		486	379.68750		487	380.46875
61		494	385.93750		495	386.71875
62		502	392.18750		503	392.96875
63		510	398.43750		511	399.21875

Appendix B

Resource utilization of the Packetized beamformer design.

```
Design Information
------------------
Command Line   : map -ise ../__xps/ise/system.ise -timing -detail -ol high
-xe n
-register_duplication -o system_map.ncd -w -pr b system.ngd system.pcf
Target Device  : xc5vsx95t
Target Package : ff1136
Target Speed   : -1
Mapper Version : virtex5 -- $Revision: 1.51.18.1 $
Mapped Date    : Fri Nov 22 17:44:06 2013



Design Summary
--------------

Design Summary:
Number of errors:      0
Number of warnings: 3233
Slice Logic Utilization:
  Number of Slice Registers:              33,922 out of  58,880   57%
    Number used as Flip Flops:            33,916
    Number used as Latch-thrus:                6
  Number of Slice LUTs:                   32,610 out of  58,880   55%
    Number used as logic:                 28,309 out of  58,880   48%
      Number using O6 output only:        22,088
      Number using O5 output only:         2,898
      Number using O5 and O6:              3,323
    Number used as Memory:                 3,957 out of  24,320   16%
      Number used as Dual Port RAM:          544
        Number using O6 output only:         346
        Number using O5 and O6:              198
      Number used as Shift Register:       3,413
        Number using O6 output only:       3,413
    Number used as exclusive route-thru:     344
  Number of route-thrus:                   3,461
    Number using O6 output only:           3,188
    Number using O5 output only:             234
    Number using O5 and O6:                   39

Slice Logic Distribution:
  Number of occupied Slices:              12,974 out of  14,720   88%
  Number of LUT Flip Flop pairs used:     42,584
    Number with an unused Flip Flop:       8,662 out of  42,584   20%
    Number with an unused LUT:             9,974 out of  42,584   23%
    Number of fully used LUT-FF pairs:    23,948 out of  42,584   56%
    Number of unique control sets:         1,166
    Number of slice register sites lost
      to control set restrictions:          2,489 out of  58,880    4%

  A LUT Flip Flop pair for this architecture represents one LUT paired
with
  one Flip Flop within a slice.  A control set is a unique combination of
  clock, reset, set, and enable signals for a registered element.
  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.
  OVERMAPPING of BRAM resources should be ignored if the design is
  over-mapped for a non-BRAM resource or if placement fails.

IO Utilization:
  Number of bonded IOBs:                      188 out of     640   29%
    Number of LOCed IOBs:                     188 out of     188  100%
    IOB Flip Flops:                           176
    Number of bonded IPADs:                    36 out of      50   72%
    Number of bonded OPADs:                    32 out of      32  100%

Specific Feature Utilization:
  Number of BlockRAM/FIFO:                    173 out of     244   70%
    Number using BlockRAM only:               173
    Total primitives used:
      Number of 36k BlockRAM used:            155
      Number of 18k BlockRAM used:             28
    Total Memory used (KB):                 6,084 out of   8,784   69%
  Number of BUFG/BUFGCTRLs:                    14 out of      32   43%
    Number used as BUFGs:                      14
  Number of IDELAYCTRLs:                        2 out of      22    9%
  Number of BUFDSs:                             2 out of       8   25%
  Number of CRC64s:                             6 out of      16   37%
  Number of DCM_ADVs:                           4 out of      12   33%
  Number of DSP48Es:                          128 out of     640   20%
  Number of GTP_DUALs:                          8 out of       8  100%
  Number of PLL_ADVs:                           2 out of       6   33%

Average Fanout of Non-Clock Nets:               3.07

Peak Memory Usage:  1851 MB
Total REAL time to MAP completion:  15 mins 50 secs
Total CPU time to MAP completion:   15 mins 35 secs

Mapping completed.
```

Appendix C

```
# This is the Initialization python script for the Packetized Beamformer
Design.

#!/usr/bin/python
import katcp, numpy, pylab, time, corr, sys
device_host     = "roach030167" # This board has X-engine 1 and Xengine 5
device_host1    = "roach030116" # This board has X-engine 2 and Xengine 6
device_host2    = "roach030174" # This board has X-engine 3 and Xengine 7
device_host3    = "roach040235" # This board has X-engine 4 and Xengine 8

device_port    = 7147

#dest_ip  =192*(2**24) + 168*(2**16) + 8*(2**8) + 200
dest_ip  =10*(2**24) + 0*(2**16) + 0*(2**8) + 1    #Modified on 25th
Oct2013
fabric_port=60000
#source_ip= 192*(2**24) + 168*(2**16) + 8*(2**8) + 201
source_ip= 10*(2**24) + 0*(2**16) + 0*(2**8) + 13   #Modified on 25th
Oct2013
mac_base=(2<<40) + (2<<32)

# core name of upper 10Gbe
tx_core_name1 = 'BEAMFORMER_INCOH1_ten_Gbe_v2'
# core name of lower 10Gbe
tx_core_name = 'BEAMFORMER_INCOH_ten_Gbe_v2'

# defining my corr for all roach Boards
my_corr =corr.katcp_wrapper.FpgaClient(device_host,device_port)
my_corr1=corr.katcp_wrapper.FpgaClient(device_host1,device_port)
my_corr2=corr.katcp_wrapper.FpgaClient(device_host2,device_port)
my_corr3=corr.katcp_wrapper.FpgaClient(device_host3,device_port)

my_corr4 =corr.katcp_wrapper.FpgaClient(device_host,device_port)
my_corr5=corr.katcp_wrapper.FpgaClient(device_host1,device_port)
my_corr6=corr.katcp_wrapper.FpgaClient(device_host2,device_port)
my_corr7=corr.katcp_wrapper.FpgaClient(device_host3,device_port)

print "beam former"
#checking whether all roach boards are connected
while not (my_corr.is_connected() and my_corr1.is_connected() and
my_corr2.is_connected() and my_corr3.is_connected()):
    pass
#added these lines on 12/12/2012.
print" Successfully Connected to ROACH \n%s\t%s\t%s\t%s\n"
%(device_host,device_host1,device_host2,device_host3)

#writing the number of integration cycles on all roach boards.It should be
same for all 8 x-engines.

my_corr.write_int("no_cycle",1)
```

```
my_corr1.write_int("no_cycle",1)
my_corr2.write_int("no_cycle",1)
my_corr3.write_int("no_cycle",1)



print"Integration time = %f"%(0.163*(10**-3)*10)

print 'Setting Destination IP on transmitter core...',

#writing the destination ip in all software registers on all roach boards.
This is for upper X-engine.
my_corr.write_int("tx_destination_ip_ps_x1",dest_ip)
my_corr1.write_int("tx_destination_ip_ps_x1",dest_ip)
my_corr2.write_int("tx_destination_ip_ps_x1",dest_ip)
my_corr3.write_int("tx_destination_ip_ps_x1",dest_ip)

#writing the destination ip in all software registers on all roach boards.
This is for lower X-engine.
my_corr4.write_int("tx_destination_ip_ps_x2",dest_ip)
my_corr5.write_int("tx_destination_ip_ps_x2",dest_ip)
my_corr6.write_int("tx_destination_ip_ps_x2",dest_ip)
my_corr7.write_int("tx_destination_ip_ps_x2",dest_ip)

print "tx_destination_ip_ps=\n%i\n%i\%i\n%i\n%i\n%i\n%i\n%i\n"
%(my_corr.read_int("tx_destination_ip_ps"),

my_corr1.read_int("tx_destination_ip_ps"),

my_corr2.read_int("tx_destination_ip_ps"),

my_corr3.read_int("tx_destination_ip_ps"),

my_corr4.read_int("tx_destination_ip_ps1"),

my_corr5.read_int("tx_destination_ip_ps1"),

my_corr6.read_int("tx_destination_ip_ps1"),

my_corr7.read_int("tx_destination_ip_ps1"))


print 'Configuring transmitter core...',
sys.stdout.flush()
my_corr.tap_start('tap0',tx_core_name,mac_base+source_ip,source_ip,fabric_
port)
my_corr1.tap_start('tap0',tx_core_name,mac_base+source_ip+1,source_ip+1,fa
bric_port)
my_corr2.tap_start('tap0',tx_core_name,mac_base+source_ip+2,source_ip+2,fa
bric_port)
my_corr3.tap_start('tap0',tx_core_name,mac_base+source_ip+3,source_ip+3,fa
bric_port)
#NOTE: keep the tg tap number different for ten GbE v2 blocks belonging to
the same ROACH board.
```

```
my_corr4.tap_start('tap3',tx_core_name1,mac_base+source_ip+4,source_ip+4,f
abric_port)
my_corr5.tap_start('tap3',tx_core_name1,mac_base+source_ip+5,source_ip+5,f
abric_port)
my_corr6.tap_start('tap3',tx_core_name1,mac_base+source_ip+6,source_ip+6,f
abric_port)
my_corr7.tap_start('tap3',tx_core_name1,mac_base+source_ip+7,source_ip+7,f
abric_port)

print 'done'

print 'Setting-up destination addresses...',
sys.stdout.flush()
my_corr.write_int("tx_destination_ip_ps_x1",dest_ip)
my_corr1.write_int("tx_destination_ip_ps_x1",dest_ip)
my_corr2.write_int("tx_destination_ip_ps_x1",dest_ip)
my_corr3.write_int("tx_destination_ip_ps_x1",dest_ip)

my_corr4.write_int("tx_destination_ip_ps_x2",dest_ip)
my_corr5.write_int("tx_destination_ip_ps_x2",dest_ip)
my_corr6.write_int("tx_destination_ip_ps_x2",dest_ip)
my_corr7.write_int("tx_destination_ip_ps_x2",dest_ip)

#writing the destination port in all software registers of all roach
boards. This is for upper X-engine.
my_corr.write_int('tx_destination_port_ps_x1',fabric_port)
my_corr1.write_int('tx_destination_port_ps_x1',fabric_port)
my_corr2.write_int('tx_destination_port_ps_x1',fabric_port)
my_corr3.write_int('tx_destination_port_ps_x1',fabric_port)

#writing the destination port in all software registers of all roach
boards. This is for lower X-engine.
my_corr4.write_int('tx_destination_port_ps_x2',fabric_port)
my_corr5.write_int('tx_destination_port_ps_x2',fabric_port)
my_corr6.write_int('tx_destination_port_ps_x2',fabric_port)
my_corr7.write_int('tx_destination_port_ps_x2',fabric_port)




print 'done'




#resetting 10Gbe core of upper X-engine of all roach boards
my_corr.write_int("reset_gbe_ps_x1",0)
my_corr1.write_int("reset_gbe_ps_x1",0)
my_corr2.write_int("reset_gbe_ps_x1",0)
my_corr3.write_int("reset_gbe_ps_x1",0)


#resetting 10Gbe core of lower X-engine of all roach boards
my_corr4.write_int("reset_gbe_ps_x2",0)
```

```
my_corr5.write_int("reset_gbe_ps_x2",0)
my_corr6.write_int("reset_gbe_ps_x2",0)
my_corr7.write_int("reset_gbe_ps_x2",0)
#print "reset_gbe_ps= %i" %my_corr.read_int("reset_gbe_ps_x2")


my_corr.stop()
my_corr1.stop()
my_corr2.stop()
my_corr3.stop()
my_corr4.stop()
my_corr5.stop()
my_corr6.stop()
```

Appendix D

```
// This C code converts the data cpatured by Gulp from binary to ASCII//
// while running specify as following: <name of the gulp dumped file>
<pack_size> <scale> <nameof file1> <name of file2> ... <name of file8>
<name of file to store headers>

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stdint.h>
int main(int argc,char* argv[])
{
//unsigned char *buffer;
 long lsize;
int i;
int k,l;

size_t result;
size_t scale;
size_t pack_size;

pack_size = atoi(argv[2]);// input packet size
printf("pack_size is %d\n",pack_size);

scale = atoi(argv[3]);//input scaling factor
// create 8 new empty file with the names n1s.txt n2s.txt ...//
FILE *ha = fopen("n1s.txt","w");// File will conatin data of X-engine1//
FILE *ha1 = fopen("n2s.txt","w");// File will conatin data of X-engine2//
FILE *ha2 = fopen("n3s.txt","w");// File will conatin data of X-engine3//
FILE *ha3 = fopen("n4s.txt","w");// File will conatin data of X-engine4//
FILE *ha4 = fopen("n5s.txt","w");// File will conatin data of X-engine5//
FILE *ha5 = fopen("n6s.txt","w");// File will conatin data of X-engine6//
FILE *ha6 = fopen("n7s.txt","w");// File will conatin data of X-engine7//
FILE *ha7 = fopen("n8s.txt","w");// File will conatin data of X-engine8//

FILE *head = fopen("nhs.txt","w");// File will conatin source and
destination IP of all data packets received//

//fclose(hd);
printf("File Name: %s",argv[1]);// argv[1] contains the name of .dat file
in which gulp packets are dumped.//
FILE *file = fopen( argv[1], "r" );

        /* fopen returns 0, the NULL pointer, on failure */
        if ( file == 0 ){
        fputs ("File error",stderr);
        exit (1);
        }
        else{
        fseek (file , 0 , SEEK_END);
        lsize = ftell (file);
        rewind (file);
        printf("%ld \n",lsize);
 unsigned char* buffer = (unsigned char*) malloc(lsize*sizeof(unsigned
char));
        if (buffer == NULL){
        fputs ("Memory error",stderr);
        exit (2);
        }
        else{

 result = fread(buffer,sizeof(unsigned char),lsize,file);

 printf("fread result   %d\n",result);

for(i=0;i<lsize/pack_size;i++)
{

unsigned long int src_ip,file_select;
unsigned long int temp;

//following for separating source IP and destination IP from the packet
header.//

for(k = 26;k<27;k=k+8)
{


if (k==26){
        unsigned long int src_ip= 0, des_ip= 0; // the source ip and des
ip does not fit in 16 bit so we use long int data type for that.

  // for source ip

        temp = (unsigned long int) buffer[i*pack_size + k];
        temp = temp << 24;
        src_ip += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k + 1];
        temp = temp << 16;
        src_ip += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k + 2];
        temp = temp << 8;
        src_ip += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k + 3];

        src_ip += ( unsigned long int) temp;

      file_select=src_ip;


// for destination ip

        temp = (unsigned long int) buffer[i*pack_size + k+4];
        temp = temp << 24;
        des_ip += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k+5];
        temp = temp << 16;
        des_ip += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k+6];
        temp = temp << 8;
        des_ip += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k+7];

        des_ip += ( unsigned long int) temp;


// to print to the file containing source addresses of received packets

      fprintf(head,"%lu\t%lu\n",des_ip,src_ip);

        }

for(k = 42;k<pack_size;k=k+8) //Data in gulp packet starts at 43rd byte
gulp packet with header. Each data of 8 bytes(64 bits)
{   // Binary to ASCII conversion
     if (k<pack_size){
        unsigned long int pol0 = 0, pol1 = 0,temp_short=0;
        signed short int pol0_scale = 0, pol1_scale = 0;temp_short=0;

// for polarization 0

      temp = (unsigned long int) buffer[i*pack_size + k];
        temp = temp << 24;
        pol0 += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k + 1];
        temp = temp << 16;
        pol0 += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k + 2];
        temp = temp << 8;
        pol0 += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k + 3];
      pol0 += ( unsigned long int) temp;
        if (pol0 > 2147483647)
        {  pol0 = pol0 - 4294967296;
      }
        temp_short=(pol0/scale);
        pol0_scale = (signed short int) temp_short;

//for polarization 1

      temp = (unsigned long int) buffer[i*pack_size + k+4];
        temp = temp << 24;
        pol1 += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k+5];
        temp = temp << 16;
        pol1 += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k+6];
        temp = temp << 8;
        pol1 += (unsigned long int) temp;
        temp = (unsigned long int) buffer[i*pack_size + k+7];

        pol1 += ( unsigned long int) temp;
        if (pol1 > 2147483647)
        {pol1 = pol1 - 4294967296;
         }
         temp_short=(pol1/scale);
         pol1_scale = (signed short int) temp_short;

// the following section selects the output file to which data has to be
written.

if(file_select == 167772173)//167772173= Souce IP of X-engine 1//
// Data of polarisation 1 is printed below. If data of polarisation 0 is
required then repalce the command by fprintf(ha1,%hd\n",pol0_scale);
{ fprintf(ha,"%hd\n",pol1_scale);}
else
        if (file_select == 167772174)//167772174= Souce IP of X-engine 2//
        { fprintf(ha1,"%hd\n",pol1_scale);}
else
        if (file_select == 167772175)//167772175= Souce IP of X-engine 3//
        { fprintf(ha2,"%hd\n",pol1_scale);}
else
        if (file_select == 167772176)//167772176= Souce IP of X-engine 4//
        {fprintf(ha3,"%hd\n",pol1_scale);}

else
        if (file_select == 167772177)//167772177= Souce IP of X-engine 5//
        {fprintf(ha4,"%hd\n",pol1_scale);}

else
        if (file_select == 167772178)//167772178= Souce IP of X-engine 6//
        {fprintf(ha5,"%hd\n",pol1_scale);}
else
        if (file_select == 167772179)//167772179= Souce IP of X-engine 7//
        {fprintf(ha6,"%hd\n",pol1_scale);}
else
        {fprintf(ha7,"%hd\n",pol1_scale);}// print to 8th X-engine's
file.//

                }


        }
}


}


}
   free(buffer);
}
```

```
        fclose(ha);
        fclose(ha1);
        fclose(ha2);
        fclose(ha3);
        fclose(ha4);
        fclose(ha5);
        fclose(ha6);
        fclose(ha7);
        fclose(head);
         fclose(file);

  return 0;

}
```

Appendix E

```
// This C code will interleave all the 8 files. Each file belonged to one
X-engine data//


#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stdint.h>

int main(void)
{


FILE *in1=fopen("n1s.txt", "r");//File containing data from X-engine 1//
FILE *in2=fopen("n2s.txt", "r");//File containing data from X-engine 2//
FILE *in3=fopen("n3s.txt", "r");//File containing data from X-engine 3//
FILE *in4=fopen("n4s.txt", "r");//File containing data from X-engine 4//
FILE *in5=fopen("n5s.txt", "r");//File containing data from X-engine 5//
FILE *in6=fopen("n6s.txt", "r");//File containing data from X-engine 6//
FILE *in7=fopen("n7s.txt", "r");//File containing data from X-engine 7//
FILE *in8=fopen("n8s.txt", "r");//File containing data from X-engine 8//

if ((in1 != NULL) && (in2 != NULL) && (in3 != NULL) && (in4 != NULL) &&
(in5 != NULL) && (in6 != NULL) && (in7 != NULL) && (in8 != NULL))
// This if loop checks that there is atleast 1 packet data in all X-
engines.//
        {
                char line1[BUFSIZ];
                char line2[BUFSIZ];
                char line3[BUFSIZ];
                char line4[BUFSIZ];
                char line5[BUFSIZ];
                char line6[BUFSIZ];
                char line7[BUFSIZ];
                char line8[BUFSIZ];

                while ((fgets(line1, sizeof line1, in1) != NULL) &&
(fgets(line2, sizeof line2, in2) != NULL) && (fgets(line3, sizeof line3,
in3) != NULL) && (fgets(line4, sizeof line4, in4) != NULL) &&
(fgets(line5, sizeof line5, in5) != NULL) && (fgets(line6, sizeof line6,
in6) != NULL) && (fgets(line7, sizeof line7, in7) != NULL) &&
(fgets(line8, sizeof line8, in8) != NULL))
                {
                        char *start1 = line1;
                        char *start2 = line2;
                        char *start3 = line3;
                        char *start4 = line4;
                        char *start5 = line5;
                        char *start6 = line6;
                        char *start7 = line7;
                        char *start8 = line8;
                        signed short int
field1,field2,field3,field4,field5,field6,field7,field8;
                        int n;

                        while ((sscanf(start1, "%hd%n", &field1, &n) == 1)
&& (sscanf(start2, "%hd%n", &field2, &n) == 1) && (sscanf(start3, "%hd%n",
&field3, &n) == 1) && (sscanf(start4, "%hd%n", &field4, &n) == 1) &&
(sscanf(start5, "%hd%n", &field5, &n) == 1) && (sscanf(start6, "%hd%n",
&field6, &n) == 1) && (sscanf(start7, "%hd%n", &field7, &n) == 1) &&
(sscanf(start8, "%hd%n", &field8, &n) == 1))


                        {     // interleaving done here by printing
channels one below the other in proper order//

                                 printf("%hd\n", field1);
                                start1 += n;
                                printf("%hd\n", field2);
                                start2 += n;
                                printf("%hd\n", field3);
                                start3 += n;
                                printf("%hd\n", field4);
                                 start4 += n;
                                printf("%hd\n", field5);
                                start5 += n;
                                 printf("%hd\n", field6);
                                start6 += n;
                                printf("%hd\n", field7);
                                start7 += n;
                                 printf("%hd", field8);
                                start8 += n;

                        }


                }

                fclose(in1);
                fclose(in2);
                fclose(in3);
                fclose(in4);
                fclose(in5);
                fclose(in6);
                fclose(in7);
                fclose(in8);
           fclose(inter_bin);
        }
        return 0;
}
```

Appendix F

```
// This C code will convert ASCII interleaved file into Pmon compatible
format. Pmon compatible format is 16 bit signed binary//

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<stdint.h>

int main(void)
{


FILE *rp=fopen("tst1.txt", "r");// give the name of the File that contains
interleaved ASCII data
FILE *wp=fopen("tst7.raw", "w");//give the name of the File that will
contain interleaved binary data

if ((rp != NULL))
{
        char line1[BUFSIZ];
        while (fgets(line1, sizeof line1, rp) != NULL)
         {
                char *start1 = line1;
                signed short int field1;
                int n;
                 while ((sscanf(start1, "%hd%n", &field1, &n) == 1))
                {



                        fwrite(&field1,sizeof(field1),1,wp); // fwrite
writes binary
                        start1 += n;


                }
        }
                fclose(rp);
}
return 0;
}
```