Postgraduate Studies Programme
in Modern Electronic Technologies

Physics Department
School Of Sciences
University Of Ioannina

Master Thesis

Design of a Readout System for Testing the Electronics of the CMS Experiment Barrel Muon Trigger at the HL-LHC Accelerator at CERN

Spyros Liontos
Registration Number
801

supervisor
Dr. Costas Foudas

Ioannina
January 2024

Acknowledgements

Firstly I would like to thank my supervisor Professor Costas Foudas for introducing and including me to the wonderful and full of possibilities world of CERN. His assistance and kind hearted spirit is deeply appreciated.

A special thanks to Prof. Ioannis Papadopoulos for introducing me to a proper OS that is Linux. A special thanks as well to Prof. Vasilis Christofilakis that introduced me to digital electronics and a thank you to all HEPLAB members for making me feel right at home.

I would like to thank my fellow colleagues Kosmas Adamidis and Yannis Bestintzanos for their excellent cooperation and assistance.

My deepest thanks to my closest friend Stavros, the ideas and conversations with whom I deeply appreciate and value. My warmest thanks to my partner in crime Eleftheria for her loving support and her everlasting patience.

I would like to express my indebted gratitude to all of my family. My brothers Stefanos, Angelos and Stavros, who I am forever grateful to, for all the things I have learned from. My deepest appreciation to my parents for always providing me with all the tools necessary to achieve my goals, especially to my mother that bought me a book about physics when I was a little kid.

Περίληψη

Ο Μεγάλος Επιταχυντής Αδρονίων LHC ειναι ο μεγαλύτερος επιταχυντής στοιχειωδών σωματιδίων στον κόσμο. Βρίσκεται στο ερευνητικό κέντρο του CERN στην Γενεύη της Ελβετίας. Τέσσερις μεγάλοι ανιχνευτές έχουν εγκατασταθεί σε τέσσερα απο τα σημεία που πραγματοποιούνται συγκρούσεις σωματιδίων, ούτως ώστε να συλλέξουν την πληροφορία από τα προϊόντα των συγκρούσεων. Ένας από τους πιο μεγάλους ανιχνευτές είναι ο CMS ανιχνευτής που βρίσκεται στο σημείο 5 του συγκροτήματος. Ο ανιχνευτής απαρτίζεται από ένα υπεραγώγιμο σωληνοειδές μαγνήτη ικανό για την παραγωγή μαγνητικού πεδίου μεγέθους των 4 Tesla αλλά μεταξύ άλλων και από ένα πολυ-σύνθετο σύστημα ανίχνευσης μυονίων.

Κατά το τέλος της τρίτης περιόδου της λειτουργίας του επιταχυντή, το σύστημα θα αναβαθμιστεί ούτως ώστε να αυξηθεί η ονομαστική τιμή της φωτεινότητας του μηχανήματος σε $7.5 \times 10^{34} cm^{-2} s^{-1}$ από την τρέχουσα τιμή $1 \times 10^{34} cm^{-2} s^{-1}$, με σχέδια για τα επόμενα 12 χρόνια να δεκαπλασιάσει την συνολική του συσσώρευση φωτεινότητας στα 4000 fb^{-1}. Ο αναβαθμισμένος επιταχυντής θα ονομάζεται HL-LHC, ο υψηλής φωτεινότητας μεγάλος επιταχυντής αδρονίων. Λόγω της αυξημένης φωτεινότητας ο αριθμός ανελαστικών συγκερούσεων ανά 40MHz θα αυξηθεί περίπου στις 200, από τον τρέχων αριθμό των 50. Για να μπορέσουν να αντέξουν στην αυξημένη ακτινοβολία τα ηλεκτρονικά του συστήματος θα αναβαθμιστούν, καθώς και επίσης για να μπορέσουν να ανταπεξέλθουν στον ανεβασμένο όγκο πληροφορίας που θα προέρχεται από τα προϊόντα των συγκρούσεων πρωτονίων με συνολική ενέργεια κέντρου μάζας στα 14TeV. Οι αναβαθμίσεις συνιστούν την δεύτερη φάση αναβάθμισης του συστήματος Phase-2.

Ο ανιχνευτής του πειράματος CMS θα αναβαθμίσει μερικά από τα συστήματα ανίχνευσης, καθώς και μερικά από τα ηλεκτρονικά συστήματα που τον υποστηρίζουν. Ο ανιχνευτής λαμβάνει σήματα από συγκρούσεις της τάξης των μερικών δεκάδων Petabytes ανά δευτερόλεπτο. Ο όγκος της πληροφορίας καθιστά δύσκολη την αποθήκευση του συνολικού ποσοστού της πληροφορίας αυτής, σε κοινά αποθηκευτικά μέσα όπως αυτά των σκληρών δίσκων. Για τον λόγο αυτόν το σύστημα ανίχνευσης σωματιδίων συμπεριλαμβάνει ένα σύστημα σκανδαλισμού. Το σύστημα σκανδαλισμού απαρτίζεται από το πρώτο επίπεδο σκανδαλισμού Level-1 Trigger, το οποίο λαμβάνει και επεξεργάζεται δεδομένα σε πραγματικό χρόνο απο τον ανιχνευτή σε ειδικά κατευσκευασμένες ηλεκτρονικές κάρτες με FPGA, και κατα δεύτερον από το σύστημα υψηλού επιπέδου σκανδαλισμού High Level Trigger το οποίο απαρτίζεται από κλασικής αρχιτεκτονικής επεξεργαστές. Το πρώτο επίπεδο σκανδαλισμού είναι υπεύθυνο για την μείωση του ρυθμού πληροφορίας από 40MHz σε 750KHz, καθώς και για την μετάδοση της πληροφορίας στο δεύτερο επίπεδο σκανδαλισμού από όπου ο ρυθμός πληροφορίας μειώνεται περαιτέρω στα 7.5KHz όπου και αποθηκεύεται τελικώς σε

κοινά αποθηκευτικά μέσα ούτως ώστε να μελετηθούν τα δεδομένα από ερευνητές σε όλο τον κόσμο.

Το πρώτο επίπεδο σκανδαλισμού, με την χρήση αλγορίθμων μπορεί να προβεί σε ανακατασκευή των τροχιών των σωματιδίων και να αποφανθεί για την σημασία του περιεχομένου της πληροφορίας. Στην συνέχεια σηματοδοτεί όλα τα συστήματα του να μεταδώσουν Read-Out την προσωρινά αποθηκευμένη πληροφορία τους στο σύστημα ανάγνωσης DAQ ούτως ώστε να μεταφερθεί η πληροφορία στο δεύτερο επίπεδο σκανδαλισμού. Στην παρούσα εργασία έχει υλοποιηθεί το Readout σύστημα για τις νέες αναβαθμισμένες κάρτες που θα δέχονται πληροφορία για την ανίχνευση των μυονίων στην περιοχή του βαρελιού. Το σύστημα καθώς βρίσκεται υπό ανάπτυξη, χρησιμοποιεί παράλληλα ηλεκτρονικές κάρτες από το τρέχων σύστημα, καθώς μεταδίδει την αποθηκευμένη πληροφορία από τον ανιχνευτή σε αυτές με την χρήση οπτικών ινών σε υψηλές ταχύτητες. Λόγω της ευρείας χρήσης συστημάτων μετάδοσης πληροφορίας υψηλών ταχυτήτων με οπτικές ίνες, πραγματοποιήθηκε η ανάπτυξη ενός πρωτοκόλλου κατά την μελέτη αυτών.

Abstract

The Large Hadron Collider is the world's largest and most powerful particle accelerator and collider. Four large detectors are placed on the LHC's interaction points in order to capture the data produced by proton-proton collisions. One of the two largest detectors is the CMS detector located at interaction point 5. It features a superconducting solenoid magnet and an elaborate muon system. The LHC is planned to be upgraded to the High Luminosity Large Hadron Collider (HL-LHC) at the end of its third run, Run 3. The new machine will increase its nominal luminosity value from $1 \times 10^{34} cm^{-2} s^{-1}$ to $7.5 \times 10^{34} cm^{-2} s^{-1}$, with a ten fold increase in its total integrated luminosity up to 4000 fb^{-1}. As a result the average number of inelastic collisions per bunch crossing, pileup, will increase from what is currently 50, up to 200. In order to sustain the radiation and the increased data flow the CMS detector is planned to be upgraded, the upgrades called the phase-2 upgrades. This thesis focuses on the readout implementation in FPGA custom made boards for the BMTL1 back end system. The readout mechanism provides the stored data from the detector to a phase-1 system in order to be relayed to the Data Acquisition (DAQ) system. Along with the readout system a comprehensive view of the high speed link protocols used in the firmware, was required. This was achieved by further studying and correctly implementing a newly created high speed protocol at 10 Gbps in a commercially available FPGA board.

Contents

Table Of Contents **vi**

1 Large Hadron Collider **1**

1.1 LHC Injection Chain 2

1.2 LHC Bunch Structure 3

1.3 High Luminosity - LHC 4

2 Compact Muon Solenoid **6**

2.1 Introduction 6

2.2 The CMS coordinate system 7

2.3 CMS Detector 8

2.3.1 The superconducting solenoid magnet 8

2.3.2 Tracker 9

2.3.3 Calorimeters 11

2.3.4 The Muon System 13

2.4 Phase-2 Trigger and DAQ System 17

2.4.1 Level-1 Trigger 18

2.4.2 Data Acquisition and High Level Trigger 22

3 FPGA - Programming Languages **24**

3.1 FPGAs 24

3.2 Programming Languages 26

3.2.1 VHDL 26

3.2.2 Verilog 30

3.2.3 TCL 30

3.2.4 Python 30

3.2.5 Bash Scripting 31

4 MultiGigabit Transceivers **32**

4.1 Multi-gigabit transceiver 32

4.1.1 Introduction 32

4.1.2 PMA - PCS . . . 33
4.1.3 Phase Locked Loop (PLL) - Quads . . . 34
4.1.4 Transmitter Functional Blocks . . . 36
4.1.5 Receiver Functional Blocks . . . 39
4.2 A Synchronous Link Protocol . . . 43
4.2.1 Introduction . . . 43
4.2.2 Protocol Logic Modules . . . 44
4.2.3 Normal Operation . . . 53
4.3 IPBus . . . 59
4.3.1 The IPBus protocol . . . 59
4.3.2 IPBus Firmware . . . 61
4.3.3 IPBus integration with the Synchronous Protocol . . . 62

5 BMTL1 Firmware - Readout System 66
5.1 EMP Framework . . . 66
5.1.1 EMP Framework Structure . . . 66
5.1.2 Empbutler tool . . . 70
5.2 Front-End Link Protocols . . . 70
5.2.1 GBT . . . 70
5.2.2 lpGBT . . . 71
5.3 Back End Link Protocols . . . 72
5.3.1 CSP . . . 72
5.4 Hardware . . . 72
5.4.1 OBDT . . . 72
5.4.2 BMTL1 . . . 74
5.4.3 OCEAN - TM7 . . . 75
5.5 Layout of the USC setup . . . 76
5.6 Readout Firmware . . . 79
5.6.1 Layout Of Readout module . . . 79
5.6.2 Readout Hit module . . . 80
5.6.3 Readout TP module . . . 82
5.6.4 Readout Track module . . . 84
5.7 USC Slice test . . . 85

6 Conclusions 90

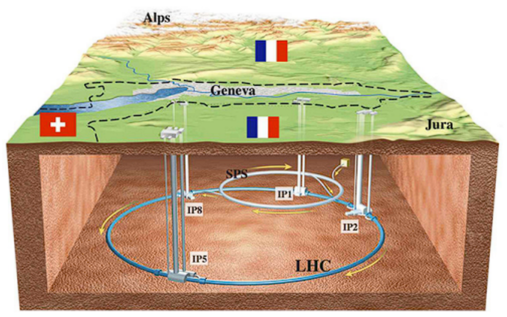
A Appendix A 91
A.1 HDL codes . . . 91
A.2 IPbus code . . . 101

Acronyms 102

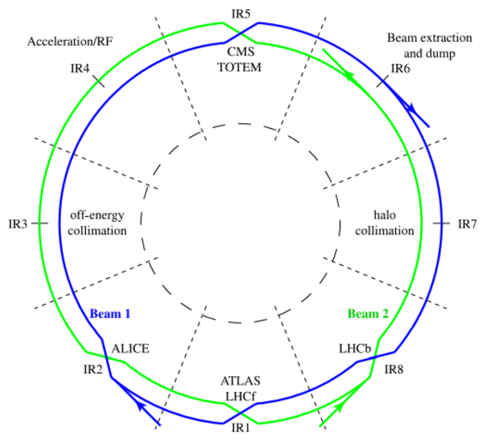
Chapter 1

Large Hadron Collider

The Large Hadron Collider (LHC) is the world's largest and most powerful particle accelerator and collider. It is located at the European Oranization for Nuclear Physics, Conseil Européen pour la Recherche Nucléaire (CERN), in an underground tunnel 100 meters underneath the Franco-Swiss border near Geneva, Switzerland. The LHC consists of a 26.7 kilometer ring of superconducting magnets with a number of accelerating structures to boost the energy of the particles. The LHC is built with two beam-pipes, kept at ultrahigh vacuum, which cross at four interaction points. Particles inside the accelerator form two high-energy beams travelling in opposite directions at a speed close to the speed of light while they are made to collide at those 4 interaction points where the two rings of the



(a)



(b)

Figure 1.1: Underground Location of the LHC collider 1.1a, Schematic layout of the LHC two-beam design and its four Interaction Regions (IR) 1.1b

machine intersect. A beam particle in each ring can reach a maximum energy of 7 TeV which results to a maximum center-of-mass collision energy of 14 TeV.

The LHC has been designed with four experimental insertions at the locations of each of the four interaction points. The CMS and ATLAS high luminosity experiments L $> 10^{34} cm^{-2}s^{-1}$ using proton-proton collisions, the LHCb B-meson experiment requiring medium luminosities L $\propto 10^{32} cm^{-2}s^{-1}$ for proton-proton collisions and the ALICE experiment dedicated for ion collisions, requiring low luminosities L $\propto 10^{29} cm^{-2}s^{-1}$.

The beams are guided around the accelerator by a strong magnetic field maintained by superconducting electromagnets. The coils used for the construction of the magnets are designed to function is a superconductive state, this way ensuring the efficient conduction of electricity without any resistance or energy loss. To achieve this, the magnets need to be cooled to $-271.3°C$, a temperature colder than outer space. For this reason the accelerator is connected to a distribution system of liquid Helium responsible for cooling the magnets [1].

1.1 LHC Injection Chain

Filling the LHC with protons or ions requires a series of pre-accelerators that successively increase their energy before being injected into the main accelerator, as depicted in figure 1.2.

The first system is the 72 meter long Linear ACcelerator 4 (Linac4), located 12 meters below ground. The Linac4 is designed to accelerate negative hydrogen ions to 160 Mev and then inject them into the PSB, through transfer and measurement lines in which the ions are stripped of their two electrons to leave only protons[2].

The Proton Synchrotron Booster (PSB) contains four superimposed rings with a radius of 25 meters, that receive beams of protons from the Linac4, and accelerates them to 2 GeV for injection into the PS [3].

The Proton Synchrotron (PS) with a circumference of 628 meters, accelerates 72 proton bunches up to 26 GeV, each bunch consisting of about 1.15×10^{11} protons [4].

The particles are then inserted to the Super Proton Synchrotron (SPS), which is the second largest machine in CERN's accelerator complex, with a circumference of nearly 7 kilometers. Inside the SPS up to 4 batches of 72 bunches are accelerated to 450 GeV before they are injected to the LHC machine [5].

Once the particles are inside the LHC machine they are accelerated up to 6.5 GeV by receiving an electrical impulse from metallic chambers containing an

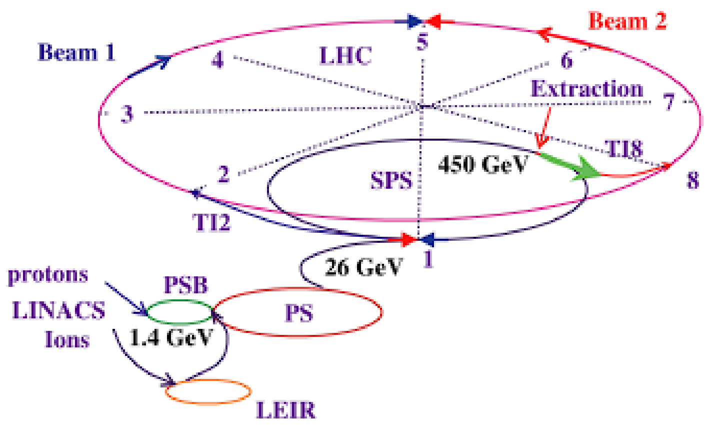


Figure 1.2: LHC injection chain Schematic view

electromagnetic field, known as radiofrequency (RF) cavities. Inside the LHC machine a total of 16 RF cavities, 8 for each of the two rings, are housed in four cylindrical refrigerators called cryomodules, which enable them to work in a superconducting state [6].

1.2 LHC Bunch Structure

The radio frequency (RF) systems provide a longitudinal focusing which constraints the particle motion in the longitudinal phase space to a confined area known as the RF bucket (slots). The machine operates at an RF frequency of 400 MHz, which corresponds to a wavelength of 75 cm or buckets (slots) of 2.5 ns. The LHC's circumference is equal to 35640 RF wavelengths, allowing the same number of buckets (slots). If all available buckets (slots) were filled with particles then collisions would occur at intervals of only 37.5 cm (1.25 ns). This way a more realistic bunch spacing is chosen, equal to one bunch per 10 RF buckets (slots) or 25 ns [1].

The RF system thus specifies the maximum number of bunches to be 3564, 25 ns apart with a frequency of 40.078 MHz, the frequency known as the LHC clock. 3564 bunches form the LHC *orbit*, whereas the orbit's frequency is defined as the time a particle bunch inside the LHC machine completes a full circle, and is equal to 11.2455 KHz.

Not all bunches in an orbit are filled with particles. Batches from the SPS will be injected into the LHC machine by groups of 3, 3 and 4 72-bunch trains, forming through the entire orbit's structure a quasi 4-fold symmetry, as shown in figure 1.3

[7]. As a result only 2808 bunches of the available 3564 are filled with particles, with a notable gap at the end of every orbit of 119 empty bunches. The most notable of all the bunches is the Bunch Crossing 0 (BC0), indicating the start of the orbit's structure as shown in figure 1.3.

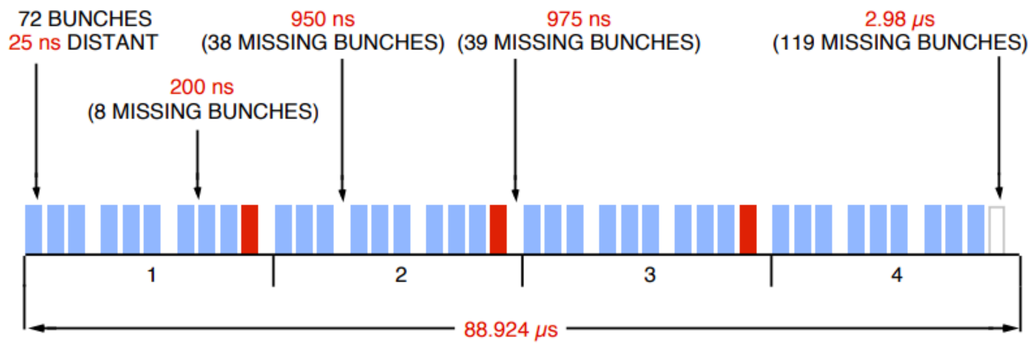


Figure 1.3: LHC bunch structure

1.3 High Luminosity - LHC

The Large Hadron Collider (LHC) was successfully commissioned in 2010 for proton–proton collisions, achieving a center of mass energy of 7 TeV. It operated with 8 TeV center of mass proton collisions from April 2012 until the conclusion of Run 1 in 2013. During Run 2 from 2015 to the end of 2018, the LHC conducted proton collisions at a center of mass energy of 13 TeV, while attaining the nominal design luminosity of $1 \times 10^{34} cm^{-2}s^{-1}$ on 26 June 2016, and a peak luminosity of $2 \times 10^{34} cm^{-2}s^{-1}$ in 2018. Currently at the writing of this thesis at Run 3, the machine operates with at 6.8 TeV center of mass proton collisions.

The remarkable achievements of the LHC have solidified CERN's status as a leading center for high-energy physics on both European and global scales. The discovery of the Higgs boson in 2012 was a major milestone in the history of science by the LHC ATLAS and CMS experiments [8]. However, in order to maintain and extend its discovery potential, a major upgrade has been decided and approved in spring 2016 by the CERN Council, to be implemented after the end of Run 3.

The upgraded High Luminosity Large Hadron Collider (HL-LHC) will aim to extend the machine's operability by another decade, increase its luminosity by a factor of five to $5 \times 10^{34} cm^{-2}s^{-1}$ and increase its integrated luminosity by a factor of 10 at 250 fb^{-1} per year, with the goal of 3000 fb^{-1} in the 12 years or so after the upgrade.

The machine is likely to be pushed to a peak levelled luminosity of $7.5 \times 10^{34} cm^{-2} s^{-1}$ increasing the average pile-up in the detectors up to around 200 from the current value of 50. This luminosity level should enable to collect up to 400 $fb^{-1}/year$, which could yield about 4000 fb^{-1} of total integrated luminosity [9]. With these changes in hand all of the experiments will need to upgrade as well in order to cope effectively with the higher pile-up.

Chapter 2

Compact Muon Solenoid

2.1 Introduction

The CMS detector is one of the two large general-purpose detectors operating at the LHC at CERN [10]. It is located at P5, 100 meters below ground level, outside of the French village of Cessy, between Lake Geneva and the Jura mountains. Proton-Proton collisions occur at the center of the detector, generating throusands of secondary particles. These particles are subsequently distributed in various directions around the interaction point. During Run 3 approximately 50 inelastic collisions (pile-up) occur per bunch crossing (25 ns). For every event the CMS detector identifies the resulting particles' tracks and measures their energy in order to later on re-create the event. The detector successfully led to the discovery of Higgs Boson at its first run [8]. Since then the CMS serves a broad physics program, ranging from the study of the Higgs Mechanism to search for new particles that could make up dark matter.

The CMS detector is depicted in figure 2.1. The detector houses a superconductive magnet that is capable of producing a magnetic field of 4 Tesla. The magnet helps to identify the charge of the particles by bending them in opposite directions, while also allowing the measurement of the particle's momentum by the amount it is bent. The magnet surrounds the following detector systems, starting from the inner most region and moving outwards, the tracker system followed by the Electromagnetic Calorimeter (ECAL) and at last the Hadronic Calorimeter (HCAL) systems. The tracker placed as close as possible to the interaction point records the paths taken by the particles, the ECAL absorbs and measures photons and electons while HCAL absorbs and measures Hadron particles. The muon system is located outside of the magnet. Muons are not stopped by the calorimeters inside the magnet, therefore chambers are placed outside the magnet in order to detect them. The detector has a total weight of 14000 tonnes, with overall dimensions of 21.6 meters of length and 14.6 meters diameter.

The detector was lowered in pieces from the surface at P5, called SX5, and assembled at the Underground eXperimental Cavern (UXC) 100 meters underground. All the electronics at the UXC are radiation hardened, while all the rest of the electronics are located at the Underground Service Cavern (USC) behind a 7 meter concrete wall. The electronics are labeled front-end and back-end electronics respectively. While the information from the front end electronics on the detector is transferred via 90 meter long fiber optic cables to the back end electronics at the service room.

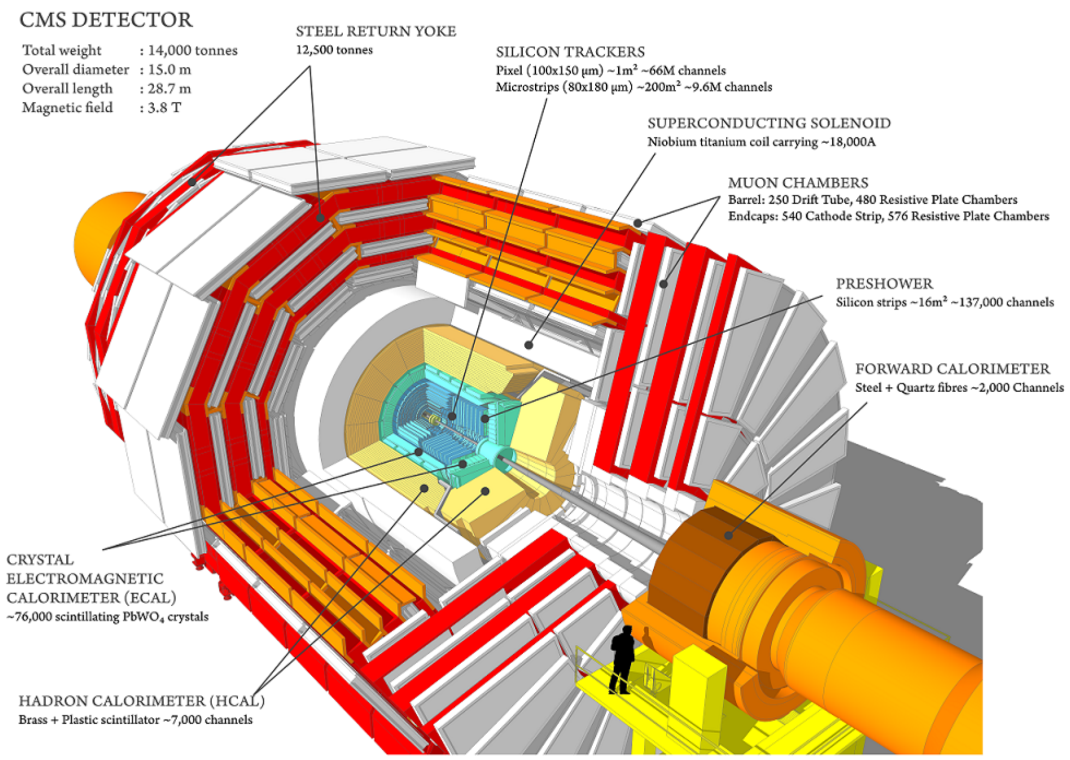

CMS DETECTOR
Total weight : 14,000 tonnes
Overall diameter : 15.0 m
Overall length : 28.7 m
Magnetic field : 3.8 T
STEEL RETURN YOKE
12,500 tonnes
SILICON TRACKERS
Pixel (100x150 μm) ~1m² ~66M channels
Microstrips (80x180 μm) ~200m² ~9.6M channels
SUPERCONDUCTING SOLENOID
Niobium titanium coil carrying ~18,000A
MUON CHAMBERS
Barrel: 250 Drift Tube, 480 Resistive Plate Chambers
Endcaps: 540 Cathode Strip, 576 Resistive Plate Chambers
PRESHOWER
Silicon strips ~16m² ~137,000 channels
FORWARD CALORIMETER
Steel + Quartz fibres ~2,000 Channels
CRYSTAL ELECTROMAGNETIC CALORIMETER (ECAL)
~76,000 scintillating PbWO4 crystals
HADRON CALORIMETER (HCAL)
Brass + Plastic scintillator ~7,000 channels

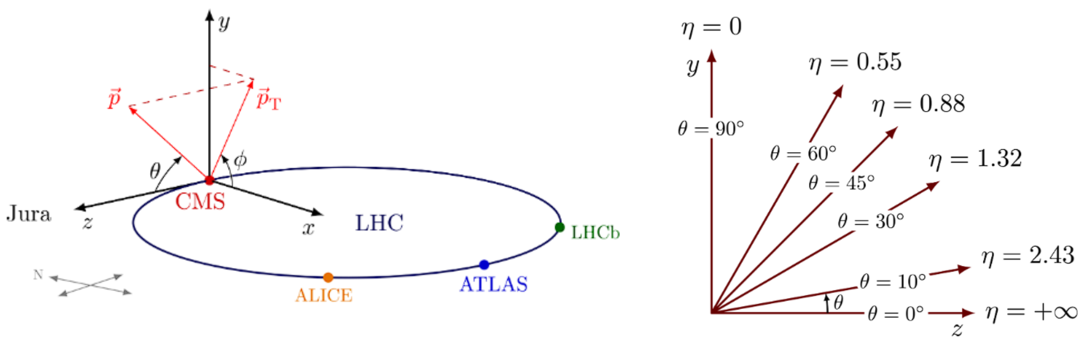
Figure 2.1: Schematic view of the CMS detector

2.2 The CMS coordinate system

The CMS coordinate system places its origin at the center of the collision point, inside the detector. As shown in figure 2.2a the y axis points vertically upwards while the x-axis points radially inwards towards the the center of the LHC. The z-axis points along the beam direction towards the Jura mountains. The azimuthal angle ϕ is measured from the x axis in the x-y plane, the plane providing the transverse view of the detector. While the polar angle θ is measured from the z axis. The CMS coordinate system uses another quantity to describe the angle of a particle relative to the beam axis, called *pseudorapidity*, η, and is defined as:

$$\eta = \ln \tan \frac{\theta}{2}$$

Figure 2.2b shows the relation of the *pseudorapidity* with the polar angle θ.



(a) (b)

Figure 2.2: CMS coordinate system (2.2a), Relation of pseudorapidity η with the polar angle θ (2.2b)

2.3 CMS Detector

2.3.1 The superconducting solenoid magnet

The superconductive Solenoid magnet of CMS is depicted in figure 2.3. It serves the crucial role of bending the trajectories of charged particles, allowing for the identification of the charge of the particle and the measurement of its momentum based on the amount the trajectory is bent. In order to achieve a high bending power the superconductive magnet's field has been designed to reach a nominal value of 4 Tesla. The magnet is formed by a cylindrical coil of 50 km of superconductive cable inside an aluminium sheath, which needs to be cooled down to a temperature of 4 Kelvin. Approximately 18500 amps are drawn by the coil. The magnet has a diameter of 6 meters and a length of 12.5 meters with a total weight of 220 tonnes, making it the largest of its kind. The magnet due to its size and structure, is providing support for the sub-detector systems inside it. The flux of the solenoid's magnetic field is returned by a 12500 tonne steel yoke comprising 5 wheels and 2 endcaps, that forms the bulk of the detector's mass [10].

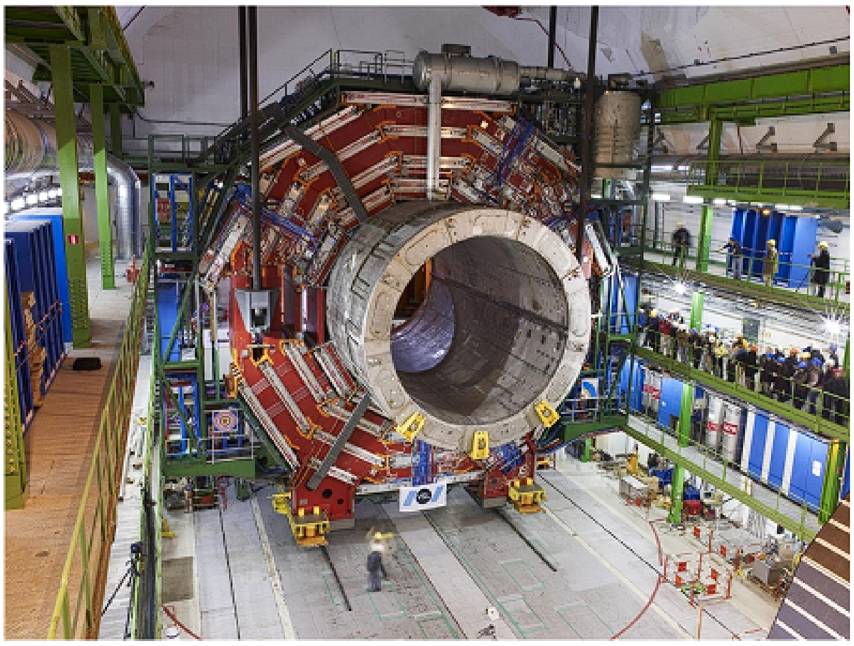
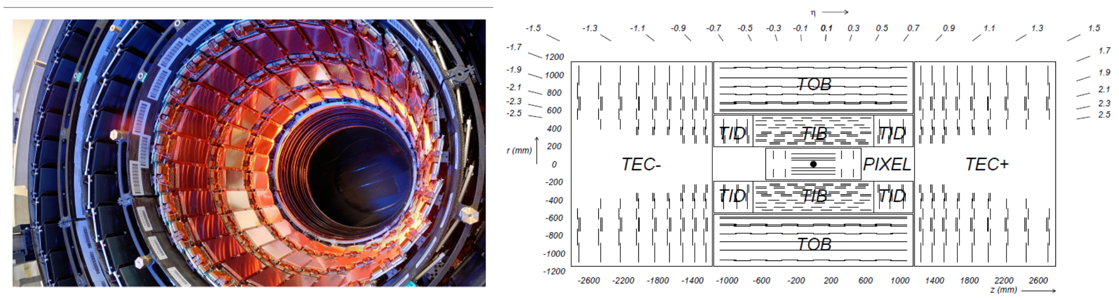


Figure 2.3: The superconducting solenoid magnet

2.3.2 Tracker

The tracking system of CMS is the innermost part of the CMS detector, installed as close as possible to the beam pipe. It is designed to provide a precise and efficient measurement of the trajectories of charged particles emerging from the LHC collisions. On average 1000 charged particles emerge from approximately 50 proton-proton interactions every 25 ns. This requires the tracker system to be a high granularity, fast response and radiation hardened electronic system, all while disturbing the particles as little as possible [10].

The Phase-1 Tracker system shown in figure 2.4a along with its layout in figure 2.4b. It is 5.8 meters wide with a 2.5 meter diameter. The innermost part of the tracker consists of 66 million Silicon Pixels cells, arranged in three cylindrical layers, at radii of 4.4, 7.3 and 10.2 cm surrounding the barrel and two disks on each side. A total of 10 Silicon Strip detectors that surround the Pixel cells are placed in the barrel region between 20cm and 116cm, while 12 disks in total are placed in the endcap regions. The silicon strip tracker has a total of 9.3 million strips, while the tracker's total area of coverage in pseudorapidity ranges from -2.5 to 2.5 as shown in figure 2.4b.



(a) (b)

Figure 2.4: Left: Phase-1 Tracker, Right: Cross section of the Phase-1 tracker

The Phase-2 upgraded tracker will be implemented at the end of Run 3. The system will offer increased radiation hardness in order to be able to cope with the increased HL-LHC particle flux, as well as higher granularity. The systems tracks will also be delivered into the Level-1 Trigger system, contrary to the current setup that only offers them to the High Level Trigger system. The Phase-2 Tracker system will be composed by an Inner Tracker and an Outer Tracker.

The Inner tracker will be equipped with pixel modules. Thin silicon sensors of thickness 100-150 μm segmented into pixel sizes of 25 $\times$ 100 μm^2 or 50 $\times$ 50 μm^2, are designed in order to exhibit the required radiation tolerance and increase the resolution. Four cylindrical layers and eight small plus four large disc-like structures in each forward direction compose the inner tracker as shown in figure 2.5. This way the overall pseudorapidity coverage will extend to $|\eta| \approx 4$ [11].

The Outer tracker will be populated with p_T modules. There are two versions of the p_T module, the 2-strip modules or 2S modules featuring two strip sensors with a length of about 5 cm and the pixel-strip or PS modules featuring a strip along with a macro-pixel sensor with a length of 2.4 cm. PS modules are deployed in the first three layers of the Outer tracker in the radial region of 200-600 mm, while another 3 layers of 2S modules are deployed in the outermost area of the tracker above 600 mm. In the endcaps the modules are arranged in rings with the 2S modules placed at the outermost parts of the rings, shown in figure 2.5. Finally, the outer tracker will be responsible for delivering track data to the Level-1 Trigger system [11].

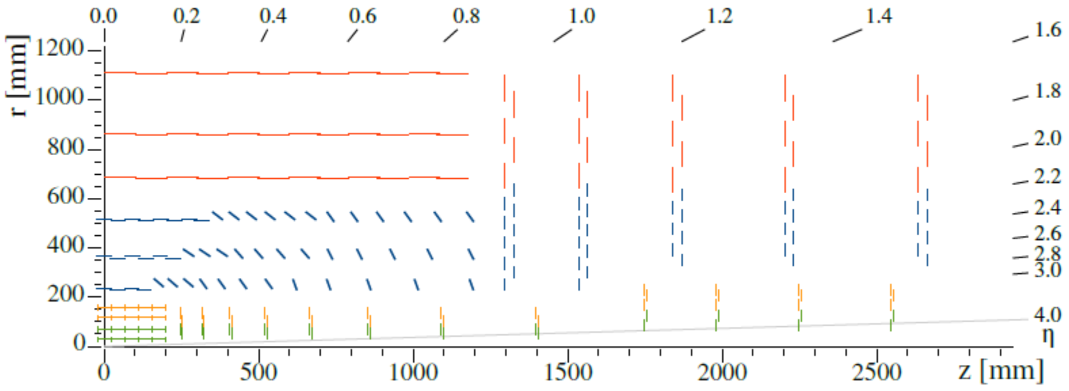


Figure 2.5: Phase-2 Tracker layout in $r - z$ view. The Inner tracker is indicated by green and yellow lines, while the Outer tracker is indicated by blue and red lines

2.3.3 Calorimeters

Two Calorimeter detectors surround the tracker, while being confined inside the solenoid magnet at CMS. The Calorimeters are placed in the following order, the Electromagnetic Calorimeter (ECAL) and Hadronic Calorimeter (HCAL), moving from the center of the detector outwards. There are two parts in each Calorimeters, one for each region they are located. There is the ECAL Barrel (EB), the ECAL Endcap (EE), the HCAL Barrel (HB) and the HCAL Endcap (HE). Both of the endcap calorimeters will be replaced by a new detector the High Granularity Calorimeter (HG-CAL).

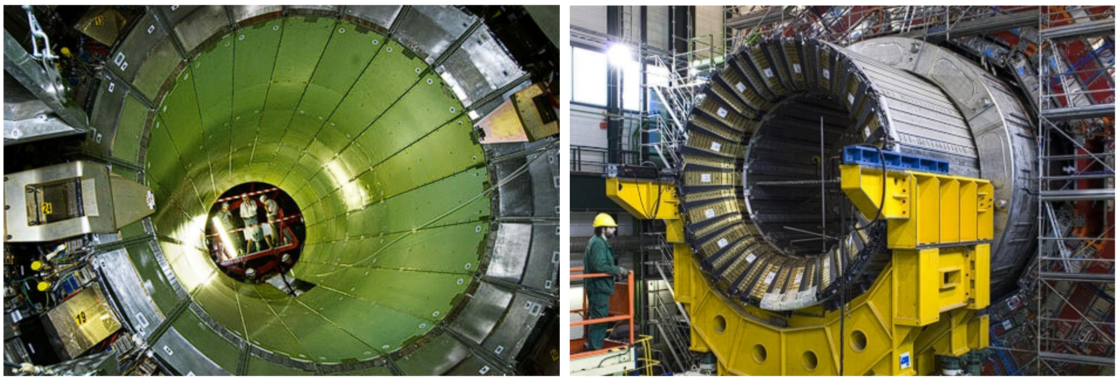
Electromagnetic Calorimeter (ECAL)

The Electromagnetic Calorimeter (ECAL) is a homogeneous crystal calorimeter that provides high precision measurements of photons and electrons. The ECAL Barrel (EB) consists of 61200 lead tungstate $PbWO_4$ crystals while it covers the area of pseudorapidity $|\eta| < 1.479$ The crystals are highly transparent and scintillate when electrons and photons pass though it. The crystals emit a blue-green scintillation light, estimated at 420–430 nm of wavelength, which is captured by Avalanche PhotoDiodess (APDs). The total barrel crystal volume is 8.14 m^3 and its weight is equal to 67.4 tonnes [10]. The ECAL detector is depicted in figure 2.6a.

Hadronic Calorimeter (HCAL)

The Hadronic Calorimeter (HCAL) is a sampling calorimeter covering for the barrel region a pseudorapidity range of $|\eta| < 1.3$. The HCAL Barrel (HB) is radially restricted between the outer extend of the ECAL at $R = 1.77m$ and the inner extent of the superconductive magnet at $R = 2.95m$. HCAL is very important for the measurement of hadrons's energies. Due to the lack of available space, the total amount of material required to absorb the hadronic shower is constrained. Therefore a complementary outer hadron calorimeter is placed outside the solenoid magnet, also called the *tail catcher*.

The HB is divided into two half-barrel sections, the HB- and HB+. Each half-section can be inserted from either end of the barrel. Each half-barrels consist of 18 identical azimuthal wedges, 36 in total for the whole HB. The wedges are constructed out of flat brass absorber plates which are aligned parallel to the beam axis. The plates are bolted together in a staggered geometry, while the innermost and outermost plates are made of stainless steel for structural strength. The plastic scintillator is divided into 16 η sectors. A total amount of 70000 scintillator tiles are grouped intro scintillator tray units, while the first one is installed in front of the first steel plate [10]. A picture of the Hadronic Calorimeter (HCAL) is shown in figure 2.6b.



(a) (b)

Figure 2.6: Left: The Electromagnetic Calorimeter (ECAL), Right: The Hadronic Calorimeter (HCAL)

High Granularity Calorimeter (HG-CAL)

The High Granularity Calorimeter (HG-CAL) will be part of the Phase-2 upgrades for the replacement of ECAL and HCAL endcap calorimeters. The HG-CAL will feature an unprecedented transverse and longitudinal segmentation for both electromagnetic and hadronic compartments, for the pseudorapidity range of $1.4 < \eta < 3.0$. It will be divided in two compartments, the electromagnetic compartment (CE-E) with 28 sampling layers, and the hadronic compartment (CE-H) with 24 sampling layers. A figure of the HG-CAL layout is shown in figure 2.7. The CE-E sections are made out of alternating absorber material, copper or lead, and silicon detectors that are assembled in *casettes*. The CE-H sections will use silicon detector modules similar to those in the CE-E for the first 8 layers and scintillator tiles along with silicon detector for all subsequent layers, assembled in a similar way as CE-E detectors in *casettes*. The CE-H absorber material is made of stainless steel disk of 35 mm thickness for the first 12 layers and 68 mm for the last 12 [12].

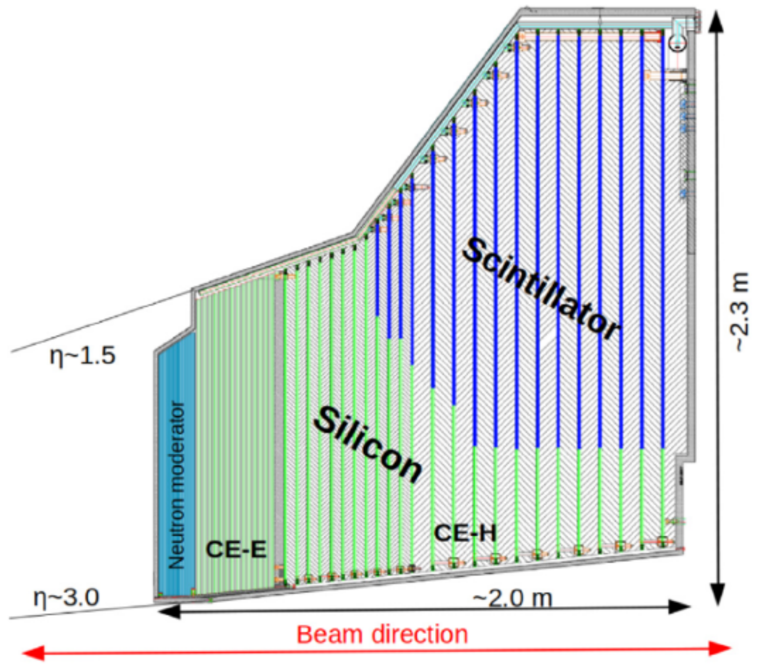


Figure 2.7: Layout of one half of the HG-CAL

2.3.4 The Muon System

As indicated by the detectors middle name, the detection of muons is of great importance to the Compact Muon Solenoid (CMS).The muon system serves three main goals, identifying muons, measuring their momentum and reconstructing their trajectories, eventually enabling the system's trigger mechanisms. In order to perform this, the muon system uses gaseous particle detectors for muon identification, sandwiched between steel layers of the return yoke. The muon system due to the shape of the magnet, is comprised by one barrel section, and two planar endcap regions, consisting of about 25000 m^2 of detector planes. Drift Tube (DT) chambers used in barrel sections cover a range of $|\eta| < 1.2$, Cathode Strip Chambers (CSCs) used in the endcap region cover a range of $0.9 < |\eta| < 2.4$, while both are complemented by Resistive Plate Chambers (RPCs) covering a region of $|\eta| < 1.9$. The muon system is depicted in figure 2.8.

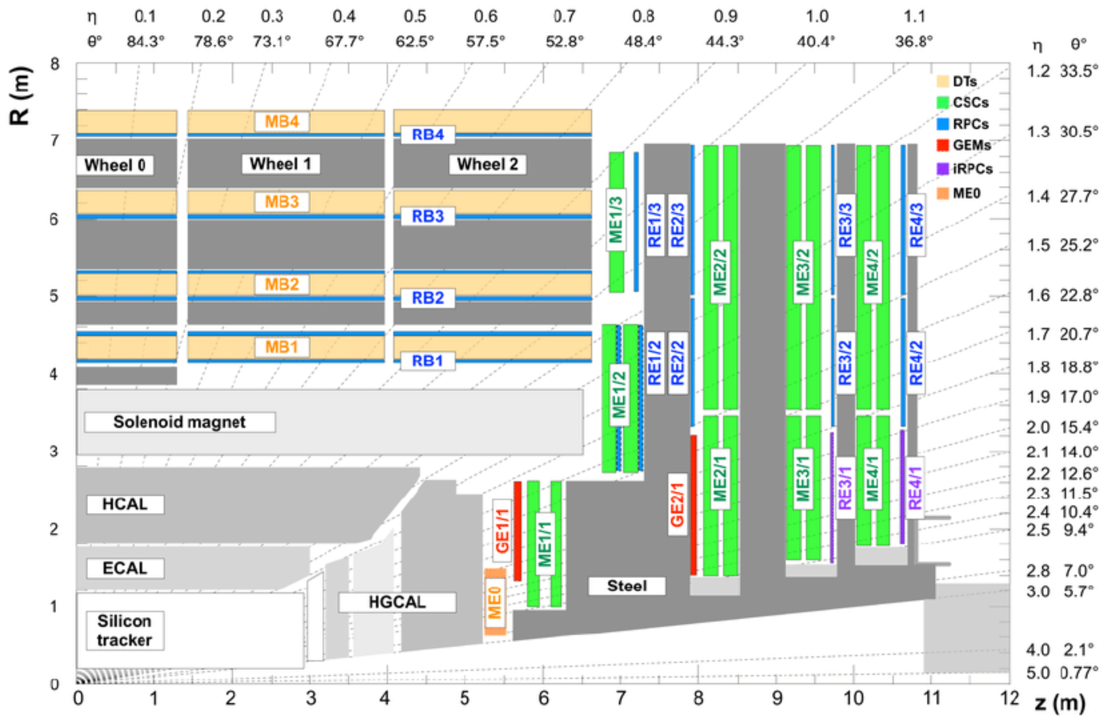


Figure 2.8: r-z view of one quarter of the CMS detector, the muon system is depicted with colors

Drift Tube (DT) system

The Drift Tube (DT) system is located in the barrel section ot the muon system. The DT is divided into five wheels, wheel -2, -1, 0, 1, 2 on the longidutinal view. Each wheel is divided in 12 Sectors on the transverse view as shown in figure 2.9, 60 sectors in total for the whole barrel. The sectors can also be grouped in 12 wedges with each wedge consisting of the 5 Sectors of each of the 5 wheels in the z-axis. Every Sector consists of 4 DT chambers labeled MB1, MB2, MB3, MB4, forming concentric cylinders around the beam axis. The top and bottom Sectors consist of 5 DT chambers due to their increased size. This makes a total of 250 chambers for the whole barrel [10].

The basic element of the DT detector is the drift cell. The cell's cross section dimensions are 13 mm $\times$ 42 mm, with a length up to 2.4 meters. At the center of the cell a 50 μm diameter gold-plated stainless steel anode wire is placed, while cathode and electrode strips are glued on the sides of the cell, shown in figure

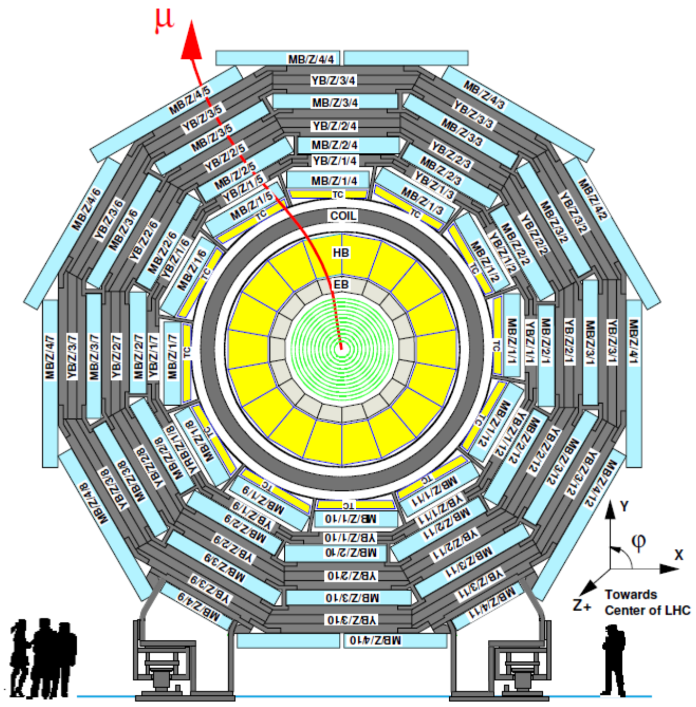
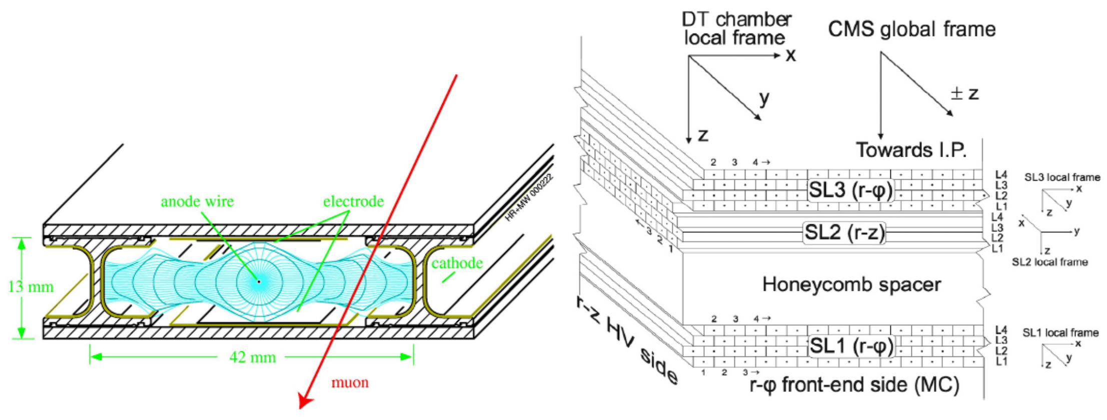


Figure 2.9: Layout of the CMS barrel muon DT chambers in 12 sectors for one of the 5 wheels

2.10a. A high density electric field is created by the anode's high voltage at 3600 V assisted by the electrode strips at 1800 V and the cathode strips at -1200 V. The ionizing gas, a mixture of 85 % Ar and 15 % CO2 which provides a saturated drift velocity of about 54 μm/ns, is used to measure the spatial position of an ionizing particle by causing the ionization of electrons in the gas while producing an avalanche effect. The drift time of the ionized electrons is measured by the electronics, while the maximum drift time is restricted by the maximum drift path of 21 mm to ~390 ns [13]. The barrel muon system contains a total of 172000 Drift Tube (DT) cells.

The Drift Tube (DT) cells are grouped in four half-staggered layers which form one SuperLayer (SL), each SL consisting of 50 to 100 cells. A chamber consists of 3 or 2 SuperLayer (SL), as shown in figure 2.10b. The 2 outer SLs, SL1 and SL3, are placed is such a way that their anode wires are parallel to the beam line, this way providing measurements on the $r-\phi$ plane, while the inner SL, SL2, is placed in way that the anode wires are orthogonal to the beam line, this way providing measurements on the $r-z$ plane or θ angle. The θ measuring SL is not present in the fourth outermost chamber, MB4, of each sector.



(a) (b)

Figure 2.10: Left: Single DT cell, Right: Schematic view of a DT chamber

Cathode Strip Chambers (CSCs)

Cathode Strip Chambers (CSCs) are used for muon detection in the endcap regions of the CMS detector. The system is comprised of trapezoidal chambers, covering 10° or 20° in ϕ direction, installed in four disks at each side of the barrel's endcaps, vertical to the beam axis. There are a total of 540 trapezoidal Cathode Strip Chamber (CSC) modules. The first disk, the one closest to the interaction point, is separated into 3 rings ME1/1, ME1/2, and ME1/3, while the rest of the disks are segmented into 2 rings, as shown in figure 2.8. The rings closer to the beampipe, ME1/1, ME2/1, ME3/1, and ME4/1 are referred to as inner rings, while the others are referred to as outer rings. The largest chambers are about 3.4 meters in length and up to 1.5 meters wide. The CSC system covers an area of pseudorapidity $0.9 < |\eta| < 2.4$, while the area of $0.9 < |\eta| < 1.1$ is covered by a combination of DT system and CSC system, know as the overlap region [14].

The CSCs are multi-wire proportional chambers consisting of 6 anode wire planes, interleaved among 7 copper cathode panels which contain strips that run lengthwise, milled at a constant $\Delta\phi$ width. The six gaps created are filled with gas containing a mixture of 40% Ar + 50% CO_2 + 10% CF_4. The muon coordinates along the wires are obtained by interpolating charges induced on the strips, produced by the ionization of the gas [10].

Resistive Plate Chambers (RPCs)

The Resistive Plate Chambers (RPCs) are gaseous parallel-plate detectors that have been installed at CMS to provide additional spatial resolution along with a more precise time measurement, coparable to that of scintillators, to the muon system. An RPC is capable of tagging an event with a time resolution smaller than the 25 ns between 2 consecutive Bunch Crossings (BXs). This way they are very important in order to identify the relevant BX to which a muon track is associated, all while being in the presence of a high particle rate and background. In total there are 480 Resistive Plate Chamber (RPC) modules located in the barrel and 576 the two endcap regions. In the barrel region there are 4 RPC stations installed in 6 layers forming 6 coaxial cylinders around the beampipe, 2 layers surrounding the first two DT station while the two remaining attached to the inner side of each DT station. The RPCs in the endcaps are placed in four concentric disks around the beampipe [10].

The Resistive Plate Chamber (RPC) module consists of 2 gaps between two resistive plates, *up* and *down* gap, each filled with a mixture of 96% $C_2H_2F_4$, 3.5% i-C_4H_{10} and 0.5% SF_6. Each module is operated in avalanche mode. The plates of the module are coated with graphite and an electrical field is created inside the gaps by applying a high voltage to the plates. When a charged particles passes through the gas, it ionizes it and creates an avalanche effect. The induced charge on the plates is read out by a strip placed between the two plates [10].

As part of the phase-2 upgrades two additional layers of Improved RPC (iRPC) chambers will be installed at disks RE3/1 and RE4/1, as shown in figure 2.8. The new layers will cover the very forward region at ranges of $1.8 < \eta < 2.4$, while the new chambers will feature thicker plates and gas gaps, this way leading to increased sensitivity [14].

Gas Electron Multiplier (GEM) detectors

Gas Electron Multipliers (GEMs) gaseous detectors are a new addition to the muon system as part of the phase-2 upgrades. Three new detectors are planned to be installed in the locations of GE1/1, GE2/1 and ME0, as shown in figure 2.8. The new detectors will cover the very forward reagion of $1.6 < |\eta| < 2.4$ near the beampipe, while the ME0 detector aims to take advantage of the the new inner tracker capabilities in order to extend muon triggering capabilities up to $|\eta| \sim 2.8$ [14].

2.4 Phase-2 Trigger and DAQ System

Proton-Proton collisions at a 40 MHz, at the center of the detector, produce several tens of Petabytes of data every second. This high data stream cannot be directly stored in disks. Therefore in order to reduce the event rate a Trigger system is implemented. The CMS trigger system is an online event selection system that receives data directy from the detector in real time, process them and determines if the data is useful. To determine if the data are useful or not, a trigger menu of algorithms is implemented, which is specified by the physics fields of interest.

There are two stages in the trigger system, the Level 1 Trigger (L1T) system and the High Level Trigger (HLT). The first stage is the L1T, it processes data on every single Bunch Crossing (BX) every 25 ns. In order to process this amount of data, custom made processor board are used with powerfull FPGAs and high speed optical links. The phase-1 Level 1 Trigger (L1T) reduced the event rate down to 100 KHz with a latency of 4 μs, while the phase-2 Level 1 Trigger (L1T) reduces the event rate down to 750 KHz with an increased latency of 12.5 μs. The data is processed by all of the subsystems and when an event if accepted the data stored up to this point in local buffers are read out by the Data Acquisition (DAQ) system, eventually transmitted to the High Level Trigger (HLT). The HLT forms the second stage of the data processing trigger system. It consists of commercial CPUs and GPUs where a more complex higher lever algorithms are able to reduce the event rate even further to 7.5 KHz. At which point the data are stored and subjected to further offline analysis by physicists in institutions all around the world.

2.4.1 Level-1 Trigger

The Level 1 Trigger (L1T) is the first stage of the trigger system. It processes data produced by the collisions at the center of the CMS detector. The Phase-2 Level 1 Trigger (L1T) is based on custom made ATCA boards. It is located in the service room at the Underground Service Cavern (USC) next to the Underground eXperimental Cavern (UXC) cavern where the detector is located.

In figure 2.11 the block diagram of the Level 1 Trigger (L1T) system is depicted. It utilizes a layered architecture processing data coming from the detector in stages. The trigger system receives trigger data from the detector via three different paths, the calorimeter trigger path, the muon trigger path and the track trigger path. The last added as part of the phase-2 upgraded system. At the first stage, data (Hits) from the detector arrive at each path, digitized by the front-end electronics, transmitted via fiber-optic cables stretching up to 90 meters via 10 Gbps links using the lpGBT link protocol to the back-end system at USC. At the second stage the data gathered from the detector are processed by algorithms in order to produce Trigger Primitivess (TPs), which contain useful information such as measurements of energy, ϕ, θ position, quality of measurement and Bunch Crossing (BX) of the Trigger Primitives (TP). At the next stage the Trigger Primitivess (TPs) are forwarded via CMS Standard Protocol (CSP) optical links to each subsystem trigger processing boards in order to reconstruct the particles. The particle data from the three subsystems are then transmitted to the correlator trigger subsystem in order to reconstruct the whole event. Eventually all the data from all subsystems, along with the correlator data, are transmitted to the Global Trigger (GT) subsystem in order to run a menu of algorithms and determine if the data contains meaningful information. If data are found to be of interest then the Global Trigger (GT) produces the Level-1 Accept (L1A) signal and transmits it to all systems, both front end and back end through the Timing and Control Distribution System (TCDS). The data are then read out and transmitted to the High Level Trigger (HLT) system by the Data Acquisition (DAQ) system.

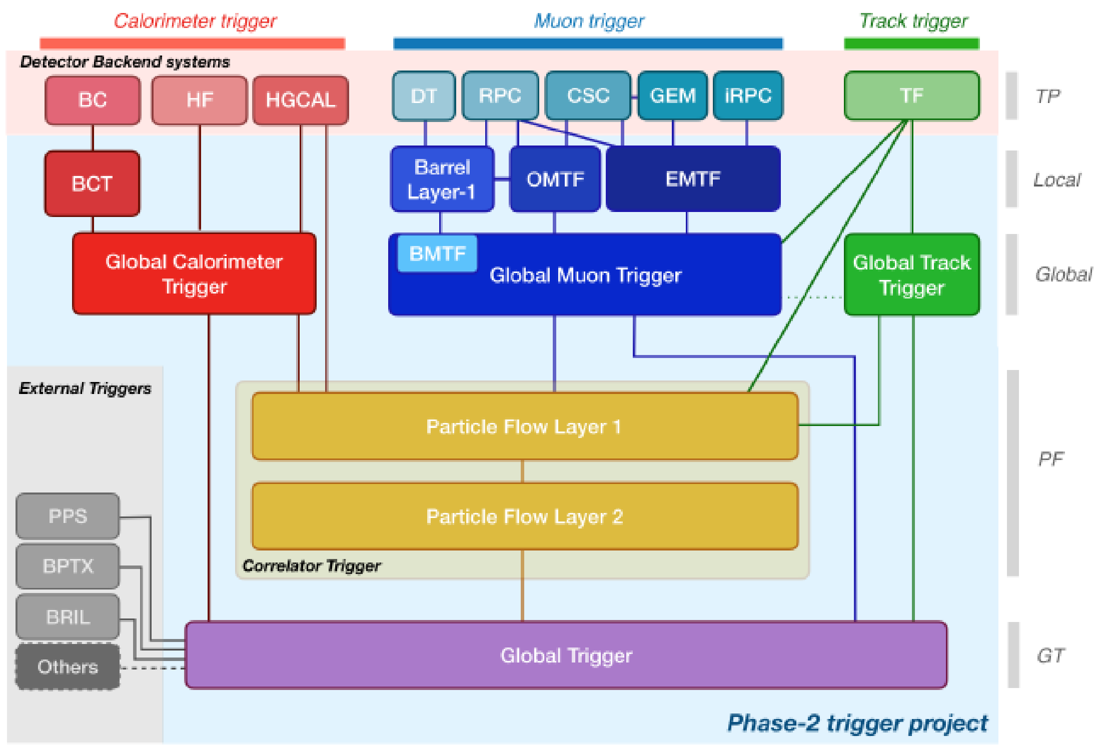


Figure 2.11: Functional diagram of the CMS Level-1 Phase-2 Trigger System

Calorimeter Trigger

A Barrel Calorimeter Trigger (BCT) back end processes the HCAL and ECAL information from the barrel region, while HG-CAL uses its own back end system. Outputs from the BCT, Hadron Forward Calorimeter (HF) primitives from the endcap regions and High Granularity Calorimeter (HG-CAL) trigger primitives are sent to the Global Calorimeter Trigger (GCT). In the GCT calorimeter-only objects are built such as hadronically decaying tau leptons candidates, electron or photon candidates, energy sums and jets. Finally the GCT's output is transmitted both to the Correlator trigger and to the Global Trigger (GT) [15].

Muon Trigger

Due to the muon systems structural form, the muon reconstruction is split into 3 subsystems, the BMTF, the Overlap Muon Track Finder (OMFT) and the Endcap Muon Track Finder (EMTF). The BMTF algorithm is implemented by the GMT boards due to its low utilization footprint.

In the endcap region Trigger primitives are created by front-end electronics receiving hits from CSCs in the region of $1.2 < \eta < 2.4$. In order to be produced, hits that form straight line patterns to at least four layers, are required. They are produced by CSC Trigger Motherboardss (TMBs) and are transmitted to the Endcap Muon Track Finder (EMTF) back end system. RPC detectors installed at regions of $0.9 < \eta < 1.9$ and iRPC detectors extending the overall coverage to $\eta < 2.4$, transmit hits to the EMTF system in order to improve and assist the timing resolution of the generated Trigger primitives. The EMTF then processes all the available Trigger Primitives from the endcap region in order to reconstruct standalone muons. This information is then forwarded to the GMT along with all non zero trigger primitives.

The overlap region covers the range of pseudorapidity $0.9 < \eta < 1.2$. The Overlap Muon Track Finder (OMFT) receives trigger primitives from both the barrel and endcap parts that correspond to this region. It then performs track matching and muon reconstruction, while eventually it transmits all non zero trigger primitives along with the reconstructed muons, to the GMT.

The barrel region consists of 60 sectors containing DT chambers and RPCs as discussed in 2.3.4. Muons passing through the DT cells ionize the gas and in turn produce a signal. Signals from the Drift Tube (DT) cells are digitized by the On detector Board for Drift Tubes (OBDT), forming TDC hits. One OBDT is responsible for transmitting Hits from one SL to the back end boards. Since a categorization of SLs is applied, based on their orientation providing θ or ϕ view measurements, the same categorization is applied to the OBDTs. The TDC hits are transmitted through 10 Gbps lpGBT links to the Barrel Muon Trigger Level-1 (BMTL1) back-end boards. The BMTL1 boards running the AM algorithm produce stubs, or track segments, which contain information about the ϕ or θ coordinates, ϕ or θ bending, BX number, quality of the trigger primitive, among others. RPC Hits are used complementary to the DT Hits in order to improve the robustness and the overall performance of the system, by producing Super Primitives. Two Sectors are able to be processed by one BMTL1 board, the outputs of which are eventually transmitted to the GMT via CSP optical links at 25 Gbps [15].

All data are collected by the Global Muon Trigger (GMT) system, receiving trigger primitives by the BMTL1 and reconstructed muons from the OMFT and EMTF, along with their non zero trigger primitives. The muon reconstruction of the barrel region is performed by the Kalman filter algorithm. The algorithm can also detect slow and displaced muons. The main operation of the GMT is the removal of misreconstructed muons and the removal of duplicate muons that are found on the borders of the aforementioned subsystems. The GMT also receives track data from either the Track Finder (TF) or the Global Track Trigger (GTT) in order generate track matched muons called tracker muons and L1 tracks matched to muon stubs, called tracks plus muon stubs. Due to latency considerations the tracker data will be delivered by the TF. The GMT outputs are delivered to the Correlator Trigger system and the GT system.

Track Trigger

As part of the phase-2 upgrade a Track Trigger path has been added to the L1T system. The track trigger system will process information from the outer tracker area of $|\eta| < 2.4$. Due the increased pileup of around 200, data are processed directly on the outer tracker's p_T modules. The modules perform hit correlations between two closely spaced silicon sensors, this way filtering signals of particles with a p_T above 2 GeV. This way the data rate is reduced by one order of magnitude. The pairs of correlated Hits, called stubs are transferred to the back end Track Finder (TF) system. The stubs are processed by the track finding algorithm, called the *hybrid* algorithm, which first generates seeds, also called *tracklets*. It does so by processing stubs from two adjacent disks that also meet certain criteria for their distance from the beamspot. The *tracklets* produced are thus consistent with particles of $p_T > 2$GeV, $|\eta| < 2.4$ and $|z_0| < 15$ cm. Once the *tracklets* are produced they are projected to other layers or disks, in order to explore for extra matching stubs within a confined window surrounding the projection. A minimum of four stubs are necessary for formation of a L1 track. Once the tracks are formed a Kalman filter algorithm is used to identify the best stub candidates and compute the track's parameters. The track data are then transmitted to the Global Track Triggers (GTTs) which is responsible for the reconstruction of the primary vertex, at which point the data are finally transmitted to the Global Trigger (GT) and Correlator trigger [15].

Correlator Trigger

The Correlator Trigger system receives inputs from all the previous track finder subsystems , GTT, GMT, GCT and HG-CAL. The CT system performs particle identifications and produces higher-level trigger objects by employing more sophisticated algorithms. This is feasible due to the increased latency time of 12.5 μs of the phase-2 system, the inclusion of tracker data in the L1T system in contrast to the phase-1 system and the advanced capabilities of the electronics systems used. The CT system is separated in two layers, *layer-1* and *layer-2*. At *layer-1* the Particle Flow (PF) algorithm is used in order to generate particle flow candidates. The particle flow candidates are constructed from the matching of calorimeter cluster data and L1 tracks, whereas PF muons are formed by matching standalone muons transmitted by the GMT, with L1 tracks by the outer silicon tracker, in which case the PF muon is not reused during the linking phase with the calorimeter clusters. The PUPPI algorithm is also implemented in *layer-1*, which is used to indicate particles originating from the hard scatter or from pileup interactions. Due to latency requirements *layer-1* CT must receive, reconstruct and transmit all data to the *layer-2* CT within 1 μs. At which stage final trigger objects are built by applying further isolation criteria, before finally transmitted to the GT [15].

Global Trigger

The Global Trigger (GT) receives inputs from the CT, GMT, GCT and the GTT system. Inside the GT a large amount of algorithms process the received data, each one describing a distinct event's signature from the trigger menu. The algorithms' outputs are monitored in parallel. Whenever one of the algorithms determines the data contain valuable information about a specific event, a L1A signal is generated along with the trigger type that was produced. The L1A signal is then transmitted to all previous subsystems by the Timing and Control Distribution System (TCDS), initiating the readout to the Data Acquisition (DAQ) system.

2.4.2 Data Acquisition and High Level Trigger

The output rate of the L1T is set to be 750 KHz while the latency of the system is 12.5 μs. Due to bandwidth and resolution considerations the L1T do not always provide the full resolution of data that is produced. The data are thus stored in buffers deep enough, in order to be transmitted after the delay of 12.5 μs in which time the GT will have made a decision of whether to accept or discard the event. Once the systems receive the Level-1 Accept (L1A) signal through the Timing and Control Distribution System (TCDS) system, they deliver the event's data at full resolution to the Data Acquisition (DAQ) system through read out links. The TCDS2 system for the phase-2 upgraded system delivers the Timing Trigger Control (TTC) signals along with the readout buffers' status in order to alert the Trigger Throttling System (TTS) in order to prevent buffer overflow.

Eventually the data are transmitted to the High Level Trigger (HLT) in order to be processed by a large number of CPUs and GPUs. In the High Level Trigger (HLT) a more sophisticated event reconstruction takes place in order to further reduce the event rate to 7.5 KHz and permanently store the event in order to be reconstructed offline and be used for analysis. The HLT runs the CMS Software (CMSSW), which is also used for offline reconstruction, while the average event processing time is 200-300 ms.

Chapter 3

FPGA - Programming Languages

3.1 FPGAs

A Field Programmable Gate Array (FPGA) is a type of Integrated Circuit that can be programmed many times after manufacturing. In contrast to a conventional micro-controller it does not follow the same CPU-oriented architecture, rather it consists of an array of Configurable Logic Blocks and interconnects that can be connected to perform various digital functions. Such logic blocks can be Flip-Flops, Look Up Tables, Digital Signal Processing (DSP) slices, Random Access Memory (RAM) Blocks, high-speed Input/Output (I/0) Serializers/Deserializers Multi-gigabit Transceiver (Transmitter-Receiver) blocks and finally I/O (Input/Output) Pins to be able to receive signals from the outside world, see 3.1. FPGAs were originally developed as a successor to programmable read-only memory (PROM) and programmable logic devices (PLDs) [16], designed to have the option to be programmed in a factory or in the field by a programmer, hence the name "field-programmable". The need to be able to reprogram an integrated circuit arose after it was evident that an application-specific integrated circuits (ASICs) once manufactured, if an error was observed, it was far too costly to redesign and manufacture again. Currently the two biggest manufacturers are XILINX and Altera.

A process in a conventional computer is handled by the CPU. Modern CPUs have many cores and each core can support hyper-threading. That means that whenever a process needs to be handled, one core and more specifically one thread compute the result of the process. This creates a problem when many processes need to be handled, because each has to wait for a thread to be freed and then be handled. FPGAs do not share the same problem, as their logic blocks are parallel in nature. Each process can be handled by a distinct section of the chip and

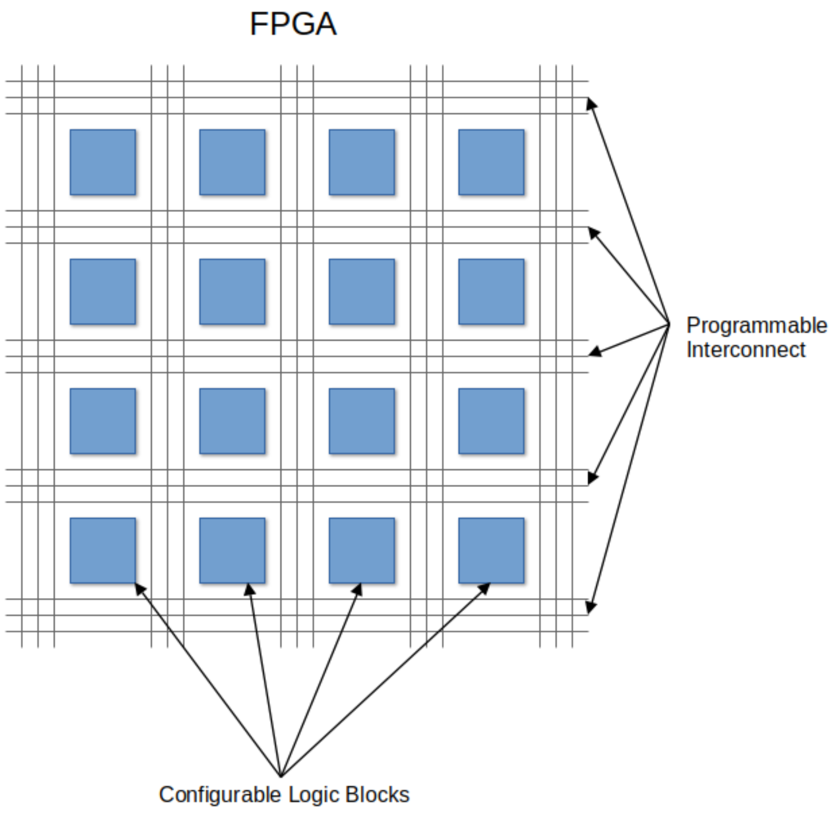


Figure 3.1: Visualization of a modern FPGA, its interconnects and CLBs

specific resources may be allocated for it, without interfering with other processes. Essentially each FPGA can be configured to handle one hundred processes in one cycle whereas a CPU would need one hundred cycles to handle them all. As a result, FPGA can be very efficient when it comes to algorithms.

In modern FPGAs there is an abundance of resources so it is even possible to configure the fpga to recreate a microprocessor. Xilinx offers the MicroBlaze package that allows the user to configure and use a 32-bit/64-bit microprocessor based on the Reduced instruction set computer (RISC) Harvard architecture. Another method of combining an FPGA with a microprocessor is the System On Chip (SOC). A System On Chip is an integrated circuit that integrates most components of a computer. These components typically include a Central Proccessing Unit (CPU), memory interfaces, Input/Output (I/O) devices and secondary storage, all on a single substrate or microchip. These components allow the use of a light-weight operating system (In most cases Linux). There exists a ZYNQ series currently manufactured by Xilinx which includes a SOC along with an FPGA.

3.2 Programming Languages

There exist two types of Programming Languages, Software Programming Languages (SPLs) and Hardware Description Languages (HDLs). Software Programming Languages can be used to create executable software in an operating system such as Linux, Windows, Mac OS to be run by the processor or the microprocessor (μP) of the system, or create a bit-file to be loaded in micro-controller (μC).

Hardware Description Languages (HDLs) on the other hand aims to describe via text a digital electronic circuit. As Hardware is concurrent in its nature, so is the language that describes it. Every statement is executed simultaneously, unlike software programming languages, where each statement is executed sequentially, one instruction at a time. The most common Hardware Description Languages are VHDL and Verilog, this thesis focuses on the first.

3.2.1 VHDL

VHISC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) was developed in 1983 by the US department of Defence in order to document the behaviour of the ASICs, companies included in their equipment. The need to simulate the ASIC from the information of this documentation led to the creation of simulators that could read the VHDL files and show the expected output. So the next step was the development of logic synthesis tools that could read the VHDL files and output a definition of the physical elements of the circuit.

There exist three main versions of the language. The original version of the language was released by Institute of Electrical and Electronics Engineers (IEEE) 1076-1987 [17], the updated version of IEEE 1076-1993 and finally version IEEE 1076-2008. This thesis follows version VHDL-93 and on.

VHDL code is written in a text file. A design in VHDL is described as an *Entity.* The VHDL module is split at the entity's port declaration, its inputs and outputs, and at its architecture. The entity's ports are the same as they would be as Pins in an IC. The architecture describes the behaviour of the module. In 3.2 this is more evident. AVHDL design may consist of many entities, and a very complex system may need to be described. A design is then described with a hierarchy in mind, entities can contain other entities, declared as *Components.* Components can be instantiated in the architecture of an entity and its Port pins can be connected accordingly with a certain Port Map using signals , which represent an electrical wire and/or busses, which represent many parallel electrical wires. Each component instantiated inside another Entity is regarded as a lower level block in the hierarchy of the design. This is analogous to using many ICs

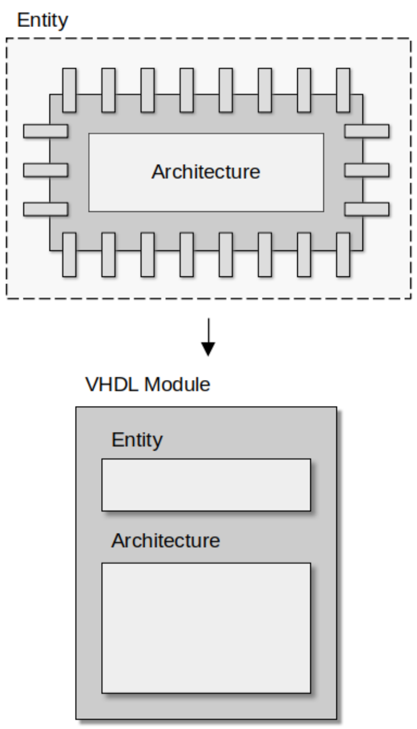
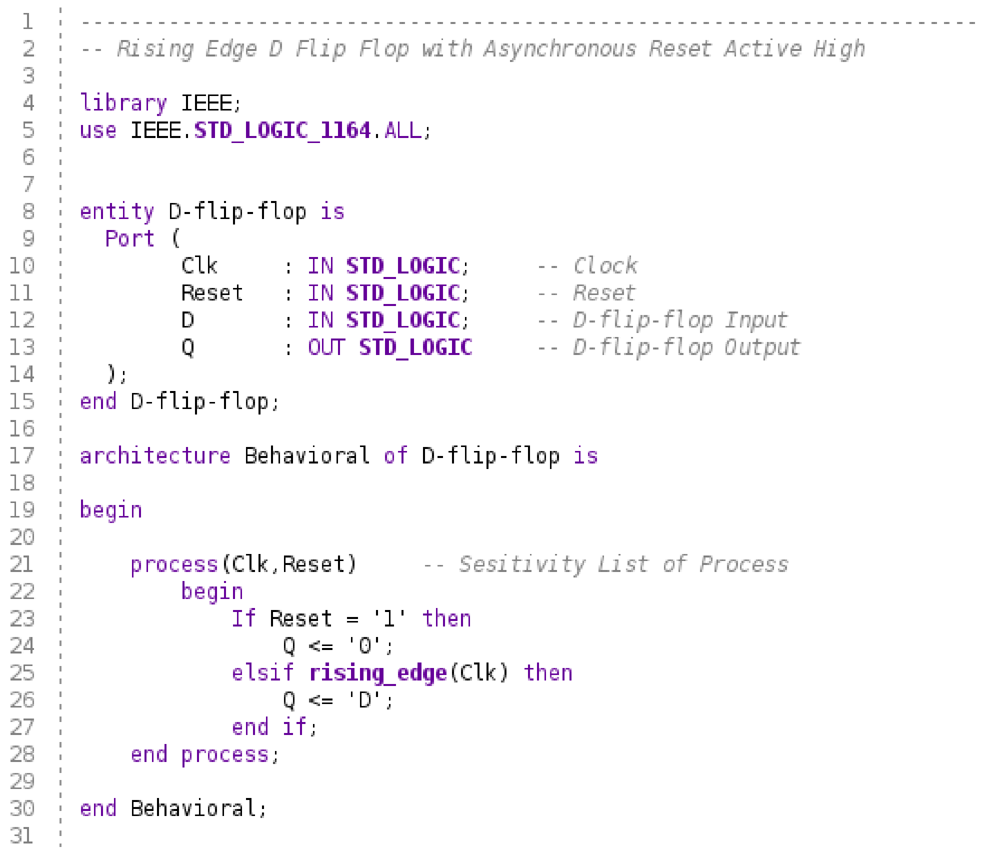


Figure 3.2: Depiction of a VHDL entity as an IC

that are connected together to form an even bigger design.

We can write statements in VHDL that are executed concurrently, essentially everything we will write inside the architecture will be executed concurrently. We use *Processes* to execute statements sequentially every time the value of one of the signals declared in the sensitivity list of the process changes. Combining gates (concurrent logic) and registers (sequential logic) produces a Combinational Logic that describes the function of a circuit.

At figure 3.3 we can see the VHDL code for a rising edge D Flip-Flop with asynchronous reset. In the port declaration we have the input and output ports, and in the architecture we have the process along with its sensitivity list. We can see that the reset is asynchronous because it does not depend at the clock, as when its value changes the process is triggered. This entity could then be instantiated as a component inside the architecture of another and possibly bigger design. That means once we have declared one entity we can use it multiple times without the need to rewrite everything again. This is very usefull as VHDL also allows us to use packages and declare functions, procedures and types in them, in order to use them in different designs. Xilinx, because some designs are common and often



```vhdl
----------------------------------------------------------------------
-- Rising Edge D Flip Flop with Asynchronous Reset Active High

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity D-flip-flop is
  Port (
        Clk     : IN STD_LOGIC;     -- Clock
        Reset   : IN STD_LOGIC;     -- Reset
        D       : IN STD_LOGIC;     -- D-flip-flop Input
        Q       : OUT STD_LOGIC     -- D-flip-flop Output
  );
end D-flip-flop;

architecture Behavioral of D-flip-flop is

begin

    process(Clk,Reset)     -- Sesitivity List of Process
        begin
            If Reset = '1' then
                Q <= '0';
            elsif rising_edge(Clk) then
                Q <= 'D';
            end if;
    end process;

end Behavioral;
```

Figure 3.3: VHDL code for a D Flip-Flop with asynchronous Reset

needed includes some packages as *IP*s (Intellectual Property). The IPs are often configured by the user with minimal difficulty and they contain all the necessary code files. Such IPs could for example be how to set up a FIFO, or perhaps a clocking Wizard that allows us to set up MMCM or PLL to produce a specific clock output.

Vivado Design Suite by Xilinx is a software suite for synthesis and analysis of HDL designs for FPGAs. Initially the VHDL code files are processed by the compiler and the hierarchy of the design is created, linking all components accordingly. Next step is to perform and RTL analysis during which the compiler checks for syntax errors, at this stage the Register-Transfer Level (RTL) design is only expressed as a design abstraction which models the synchronous digital circuit, about to be created, in terms of the flow of digital signals between hardware registers and the logical operations performed on those signals. It is a high level representation of the circuit from which lower-level representations and actual wiring will be derived. We can use behavioural simulation to check with a test

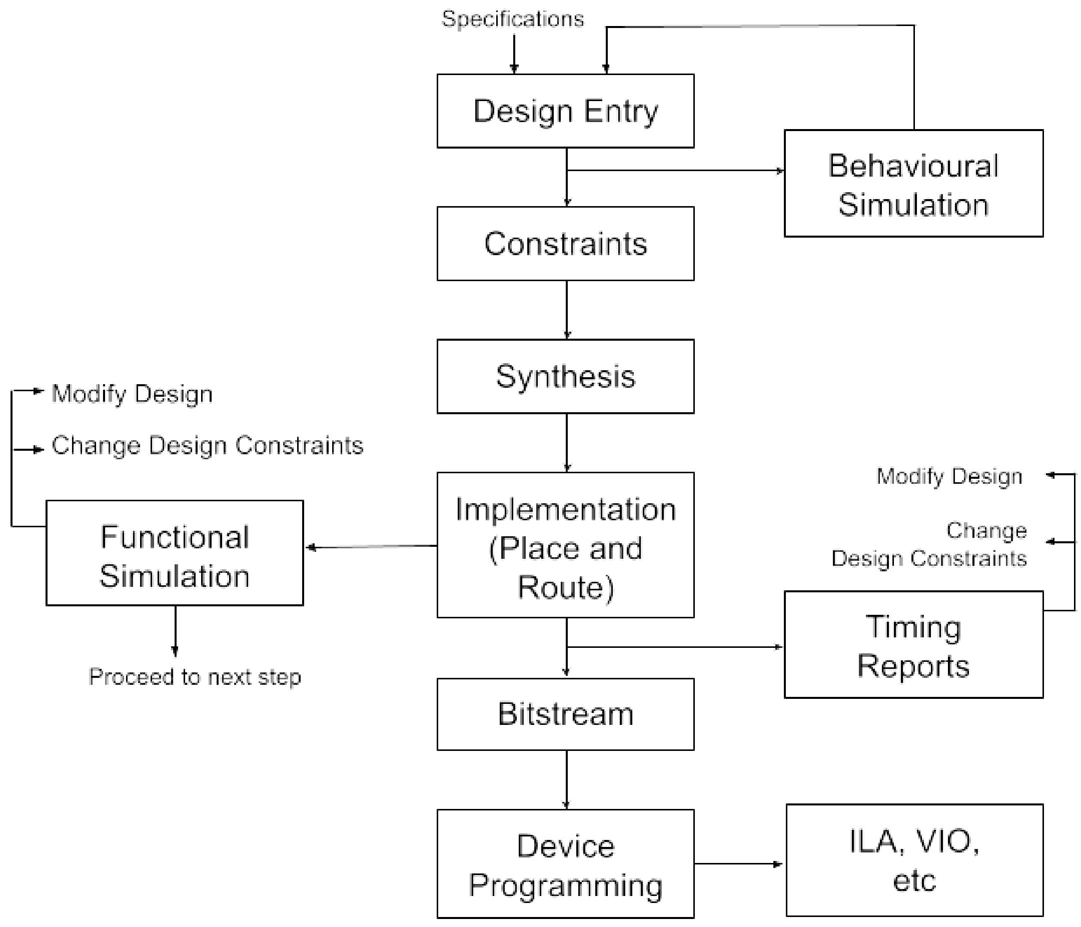


Figure 3.4: Flowchart of VHDL code Development to FPGA programming

bench what is the functionality of our design, and if our specifications are not met we re-design our circuit. The next stage is *Synthesis*, at this level the Register-Transfer Level (RTL) design is transformed into the a gate-level representation, and only after that begins *Implementation*. Both at Synthesis and at Impementation the compiler takes account of any Xilinx Design Constraints (XDC) files the user may have included, but they are specifically required for Implementation in order to connect the I/O Pins of the FPGA and constrain the design accordingly. In Vivado Implementation the following steps are performed:

- Opt Design: Optimizes the logical design to fit onto the target device.
- Place Design: Places the design onto the target device and improves timings.
- Route Design: Routes the design onto the target device.

After Implementation the user can perform a more in-depth Post-Implementation Functional Simulation that can correctly give an estimate on timings, power consumption etc. If timing reports and simulations, all satisfy the specifications then we move on to Bitstream Generation where a .bit file is created in order to programm the FPGA.

Various tools are available in order to debug our design even when it is running on hardware. Such tools are Integrated Logic Analyzers (ILAs), Virtual Input/Outputs (VIOs) and one more that we will use extensively, the IPbus, more to be explained later in this thesis. If we do not observe the expected behaviour then we can modify our design and our constrains accordingly and repeat the process of figure 3.4.

3.2.2 Verilog

Verilog is a Hardware Description Language (HDL) standardized as IEEE 1364 [18] initially developed by Gateway Design Automation, acquired by Cadence Design Systems in 1990. Verilog syntax resembles that of C programming Language, and it is widely adopted by programmers in the US, whereas VHDL is mainly used in Europe. Vivado Design Suite allows us to use both Verilog and VHDL in our designs (Mixed), as long as they can be expressed as components in the other language's syntax.

3.2.3 TCL

Tool Command Language (TCL) is a high level, interpreted programming language with varibales, procedures and control structures. It has been adopted as the standard application programming interface (API) by most EDA vendors. AMD has also adopted TCL as the native programming language for the Vivado Design Suite, as it provides control of the application. access to design objects and create custom reports. Furstermore Vivado uses Xilinx Design Constraintss (XDCs) to specify the design constrains, which is based on a subset of TCL commands availiable in the application [19].

3.2.4 Python

Python is high-level programming language that is a cross-platform programming language, meaning it runs on all the major operating systems, is dynamically typed , meaning the user does not need to specidy the data type of the variable explicitly, is an interpreted Language , meaning that the source code is executed line by line and its design philosophy empasizes on code readability with the use of indentation [20]. In this thesis it is mainly used along with the IPbus tool in order to control and monitor the status of an FPGA.

3.2.5 Bash Scripting

Bash is a Unix shell and command Language developed for the GNU's Not Unix (GNU) Project as a free software replacement for the Bourne shell. Bash is a command processor that operates within a text window where users input commands to perform actions. Bash can also read and execute commands from a file, called a shell script. In a System On Chip (SOC) where a light-weigh operating system such as Linux is present, Bash scripts are used in order to test various functions of the FPGA and often automate others.

Chapter 4

MultiGigabit Transceivers

4.1 Multi-gigabit transceiver

4.1.1 Introduction

FPGAs are used in order to process in parallel large amounts of data. That data needs to be delivered to the chip and possibly transmitted to another by some means. One way would be to use wires in parallel that could transmit and receive the data, but this adds cost and complexity to the system, while we would run out of I/O pins on the chip too. That way Multi-Gigabit Transceivers (MGTs) are used.

A Multi-Gigabit Transceiver (MGT) is a special block inside the FPGA that converts data from parallel interfaces to serial and from serial to parallel on the receiving end. This SerDes (Serializer / Deserializer) block is capable of operating at serial bit rates above 1 Gigabit/sec. MGTs using optical links (with optic fibers cables) exhibit minimal attenuation, and thus the can be used over longer distances, this way reducing the cost compared to the parallel wire scheme, all while delivering the same data throughput.

Some types of transceivers used in the UltraScale architecture of Xilinx FPGAs are GTH, GTY and GTM. All transceivers are arranged in groups of four, known as a transceiver Quad where Each transceiver is a combined transmitter and receiver. Table 4.1 shows various types of transceivers (Ultrascale U and Ultrascale+ UP) along with their maximum Line Data Rate [21]. In this thesis we will focus more on the GTH transceivers.

	KU (Kintex)	KUP (Kintex)	VU (Virtex)	VUP (Virtex)
GTH	16.3	16.3	16.3	-
GTY	16.3	32.75	30.5	32.75
GTM	-	-	-	58.0

Table 4.1: Types of MGTs along with their Maximum Line Data Rate in Gb/s

4.1.2 PMA - PCS

A picture of a channel of an MGT is shown at figure 4.1. The Multi-Gigabit Transceivers (MGTs) is divided in two functional groups, the Physical Medium Attachment (PMA) and the Physical Coding Sublayer (PCS).

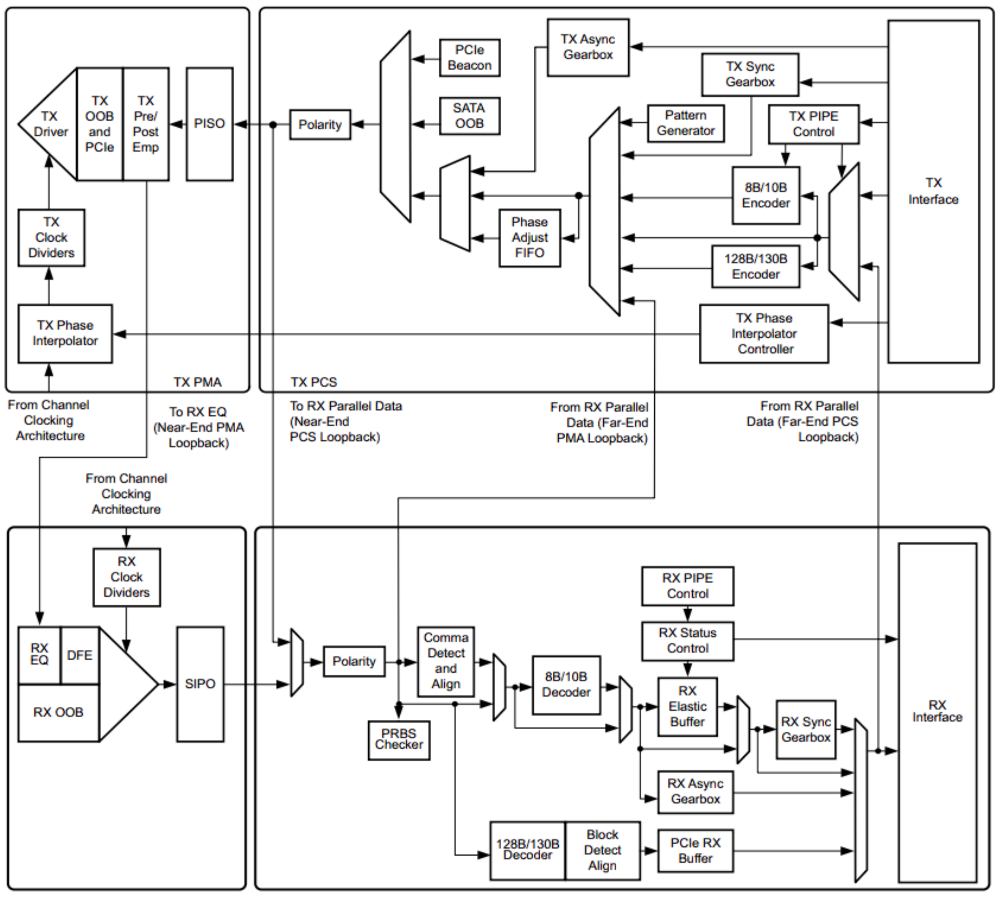


Figure 4.1: MGT channel topology

The PCS block is responsible for data encoding and decoding, scrambling and de-scrambling, Comma detection, byte and word alignment, Pseudorandom Binary Sequence (PRBS) pattern generation and checking and consists of various FIFOs for clock correction and channel bonding. The PMA receives/transmits data in parallel from the PCS and convert them to serial streams. It consists of Phase Locked Loops (PLLs), Serial In Parallel Out (SIPO) and Parallel In Serial Out (PISO) register blocks, signal equalization modules, voltage termination modules and clock recovery blocks.

4.1.3 Phase Locked Loop (PLL) - Quads

In order to be able to convert parallel data to a serial stream, PLLs are used to create a much higher clock. A Phase Locked Loop (PLL) is a closed-loop control system that generates an output signal whose phase is related to the phase of an input signal. It typically consists of a Phase Detector, that compares the phase of the input signal against the feedback output signal, a Loop Filter, a Voltage Controlled Oscillator (VCO), that proportionally controls the frequency and the phase of the oscillator thus producing a periodic signal of a specific frequency and a Feedback Divider (N) that determines the VCO multiplication ratio.

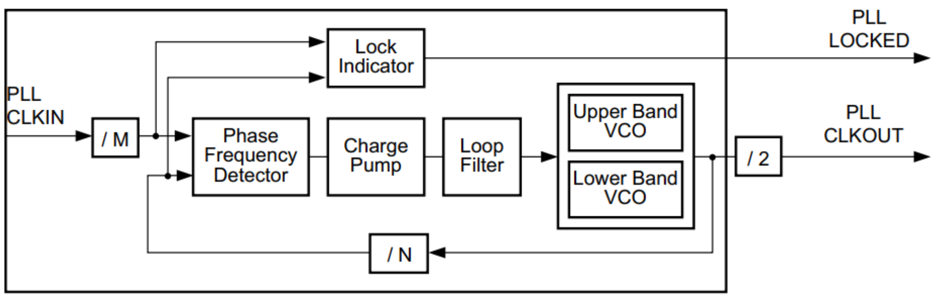


Figure 4.2: GTH QPLL Detail

Figure 4.3 depicts the clustering of four GTH3/4_CHANNEL primitives and one GTH3/4_COMMON primitive that form a *Quad* (Q). The COMMON primitive contains two LC-tank PLLs called the QPLL0 and QPLL1 and only need to be instantiated when a QPLL is in use. Whereas each CHANNEL primitive consists of a Channel Phase Locked Loop (CPLL), a transmitter and a receiver, and needs to be instantiated each time. The clocks produced by the PLLs need a reference clock source, often called the MGTREFCLK. GTH transceiver structure allows two modes of operation, input and output mode. In the input mode we provide a clock source on the dedicated reference clock pins that are used to drive the QPLL or the CPLL, using the IBUFDS3/4_GTE3 buffer primitive. In the output mode

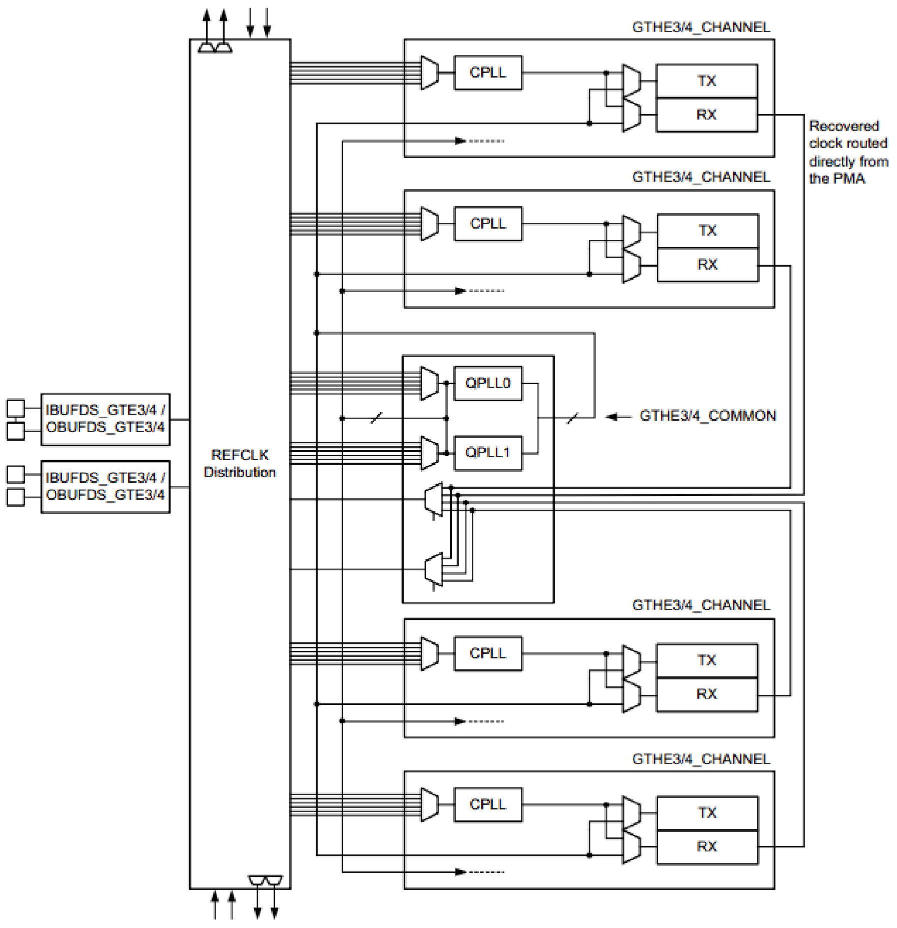


Figure 4.3: GTH Transceiver Quad Configuration

the recovered clock RXRECCLKOUT from any of the four Channels' Receivers from the same Quads can be used with the OBUFDS_GTE3/4 buffer primitive, as shown in figure 4.3.

	VCO Frequency (GHz)
QPLL0	9.8 – 16.375
QPLL1	8.0 – 13.0
CPLL	2.0 - 6.25

Table 4.2: Nominal Operating Range Of QPLL/CPLL

The reference clock for a Quad $Q(n)$ can be sourced from the refence clock of up to two Quads below $Q(n-1)$ and $Q(n-2)$, or from up to two Quads above $Q(n+1)$ and $Q(n+2)$. The selection of QPLL or CPLL depends on desired line rate. The operating line rates for each type of Phase Locked Loop are detailed in Table 4.2.

4.1.4 Transmitter Functional Blocks

Each transceiver includes an independent transmitter. Parallel data flows from the user design into the TX interface, through the PCS and PMA, and then out of the TX driver as a high-speed serial data stream. Figure 4.4 shows the TX block diagram of a transceiver with all its functional blocks along with its clocking network. The most used and important are described below in greater detail.

- **TX interface**
  The TX interface is the gateway to the TX datapath. Data can be written by the user to the TXDATA port of the TX interface on the positive edge of TXUSRCLK2. The width of the port depends on the TX_DATA_WIDTH, with possible combinations being 16,20,32,40,64 and 80 bits. The TXUSRCLK2 frequency is determines by the line rate, the width of the TXDATA port and by the protocol we have chosen (whether 8b/10b is selected). As the data are exported from the TX interface they are inserted with the TXUSRCLK to thePCS via the internal datapath, with the width being configurable between 2-bytes and 4-bytes (Internal Datapath Width). TXUSRCLK and

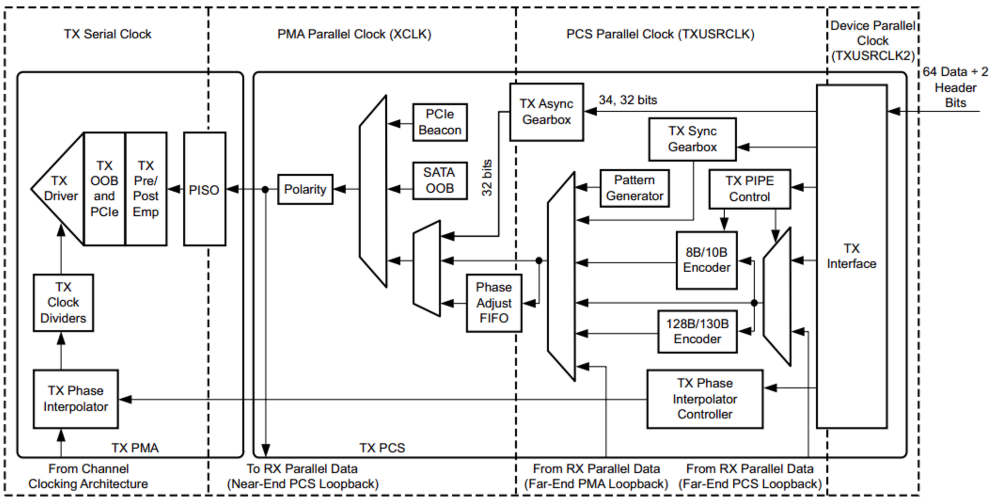


Figure 4.4: GTH Transceiver TX Block Diagram with Clock Domains

  TXUSRCLK2 have a fixed-rate relationship which can be seen at table 4.3, while equation 4.1 shows how to calculate the required rate for TXUSRCLK for all cases except when the asynchronous gearbox is used.

$$TXUSRCLK = \frac{LineRate}{InternalDatapathWidth} \tag{4.1}$$

TXUSRCLK and TXUSRCLK2 must be positive-edge aligned and buffers

Internal Datapath	**TX_DATA_WIDTH**	**Relationship**
2-Byte	16, 20	$F_{TXUSRCLK2} = F_{TXUSRCLK}$
2-Byte	32, 40	$F_{TXUSRCLK2} = F_{TXUSRCLK}/2$
4-Byte	32, 40	$F_{TXUSRCLK2} = F_{TXUSRCLK}$
4-Byte	64, 80	$F_{TXUSRCLK2} = F_{TXUSRCLK}/2$

Table 4.3: TXUSRCLK2 to TXUSRCLK Frequency Relationship

like BUFG_GTs must be instantiated to drive them. Despite operating at different frequencies, the transmitter reference clock MGTREFCLK, TXUSRCLK and TXUSRCLK2 must share the same oscillator as their source. That requires the frequencies of the two clocks to be multiplied or divided versions of the MGTREFCLK [22].

- **TX 8B/10B Encoder**
  The 8b/10b encoding scheme is used by high-speed protocols, designed to achieve DC-balance and bounded disparity in order to allow reasonable clock recovery. The protocol uses an extra of 2 bits in an 8 bit word to encode it, in order to prevent a large amount of consecutive 0s or 1s [23]. The trasmitter has a built in encoder in order to reduce the use of additional device resources.

- **TX Synchronous Gearbox**
  Some other high-speed data rate protocols use the 64b/66b encoding scheme in order to reduce the overhead of the 8b/10b. The TX synchronous gearbox provides support for this encoding, waiting to receive the word, where 64 bits are used for the payload and a total of 2 additional bits for the header, which is responsible for alignment, and control procedures. The potential interfaces are the same as the internal datapath width, 2-byte, 4-byte and 8-byte.

  Figure 4.5 demonstrates the first four clock cycles of Bit and Byte ordering for both the header (2 bits) and the payload (64 bits) of two 64b/66b words, while using a 4-byte (32 bits) internal datapath width. Initially We have available 2 bits from the header of the word and 32 bits from the data.

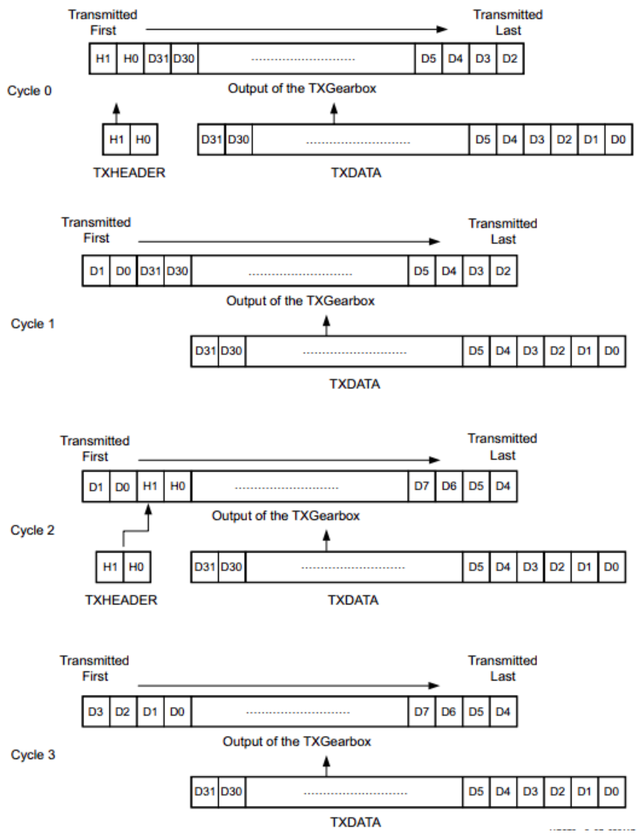


Figure 4.5: TX Synchronous Gearbox Bit and Byte Ordering

On the first clock cycle the 2 bits of header and 30 bits of the word are transmitted. On the second second clock cycle we transmit the 2 bits that were left over and another 30 from the second part of the data. When another 64b/66b word arrives its header (2 bits), the left over 2 data bits from the first word, and 28 bits of data from the current word are transmitted on the third clock cycle. And so on the process continues.

After a few clock cycles we can see that we would begin to lose data. In order to avoid this, for one (or two , depending on the relationship of the internal datapath width and the TX data width) clock cycles of TXSUSRCLK2 the transmission is halted. The TX gearbox has a specific port named TXSEQUENCE, for which the user is responsible to provide an incrementing counter, using the TXUSRCLK2. When 64B/66B encoding is in use, the counter must increment from 0 to 32 and repeat from 0. However the the counter must increment once every two TXUSRCLK2 cycles when TX_DATA_Width and the internal datapath width is the same.

- **TX Buffer**
  In figure 4.4 we can see that the PCS block is split in two clock domains, the TXUSRCLK clock domain and the PMA parallel clock domain XCLK. In order to transmit data the two clocks must have the same frequencies and all phase differences between the two need to be resolved. In order to achieve this, the GTH transmitter includes the TX buffer.

- **TX Buffer Bypass**
  When minimal latency in the design is necessary, the TX buffer needs to be bypassed and the the TX phase alignment circuit is used in order to resolve differences between the XCLK and TXUSRCLK clock domains.

- **TX Polarity Control**
  In the event that TXP and TXN differential traces are swapped by accident on the PCB, the data transmitted are reversed. In order to counter this we invert the parallel data before serialization in order to offset the reversed polarity on the differential pair.

- **TX Fabric Clock Output Control**
  In many instances the user may need to output clocks used to transfer data in parallel inside the transmitter, in order to build additional logic designs in the same clocking region. The TX Clock Divider Control block is divided in two main functions, the serial clock divider control and the parallel clock divider and selector control. Using the parallel clock selector the user can output the desired clock.

- **TX Configurable Driver**
  The TX Configurable Driver is a high-speed current-mode differential output buffer that features differential voltage control and calibrated termination resistors.

4.1.5 Receiver Functional Blocks

Each transceiver also includes an independent Receiver RX. High-speed serial data flows into the PMA of the transceiver, converts to parallel data into the PCS and finally is driven out to the design logic, availiable to be read by the user. Figure 4.6 shows the RX block diagram of a transceiver with all its functional blocks along with its clocking network. The most used and important are described below in greater detail.

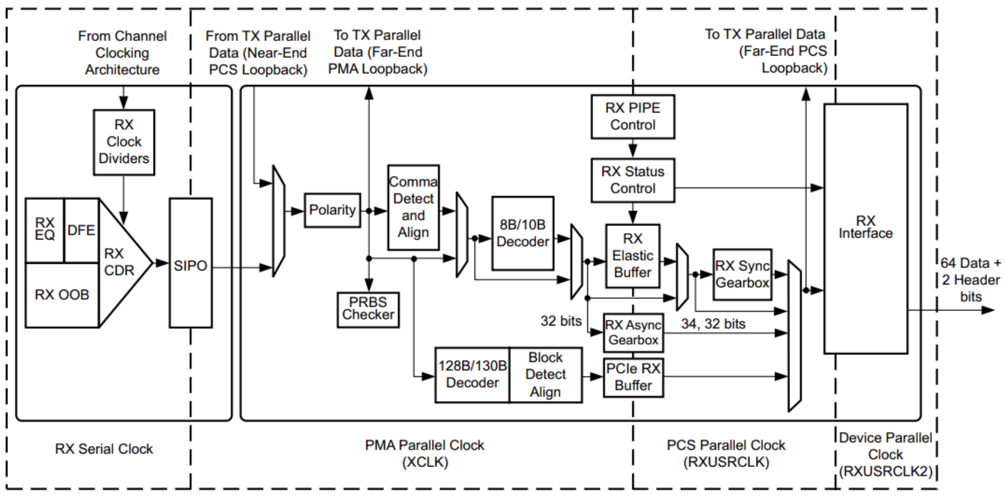


Figure 4.6: GTH Transceiver RX Block Diagram with Clock Domains

- **RX Analog Front End**
  The RX analog front end is a high-speed current-mode input differential buffer which features Configurable RX termination voltage and calibrated termination resistors.

- **RX Equalizer (DFE and LPM)**
  A signal travelling through a transmission medium, is subjected to various effects such as attenuation, distortion and noise. to combat this, there are two types of adaptive filtering available in the receiver, the Low Power Mode (LPM) and the Decision Feedback Equalizer (DFE) mode. LPM focuses on power efficiency and is mainly used on shorter reach and low reflection channel. Whereas DFE is used for longer reach lossier channels, allowing a better compensation of losses by a better adjustment of filter parameters than when using a linear equalizer. DFE can also equalize a channel without amplifying noise and crosstalk between other channels.

- **RX clock data recovery**
  The Clock Data Recovery (CDR) circuit shown in figure 4.7 exists in every channel of a Quad of a GTH transceiver. It is responsible for recovering the clock and data from an incoming serial data stream. Incoming data first go through the equalization stages of the receiver and are then captured by an edge and data sampler, while also transmitted further along the receiver blocks. Both data arriving from both the edge and data samplers are used by the CDR state machine in order to determine the phase of the incoming data stream and to control the Phase Interpolators (PIs). The PLL provides a base clock to the Phase Interpolator that in turn produces fine, evenly

spaced sampling phases which allows the CDR state machine to have a fine phase control. Thus a clock is recovered by the data stream and available to use, as described in section 4.1.3.

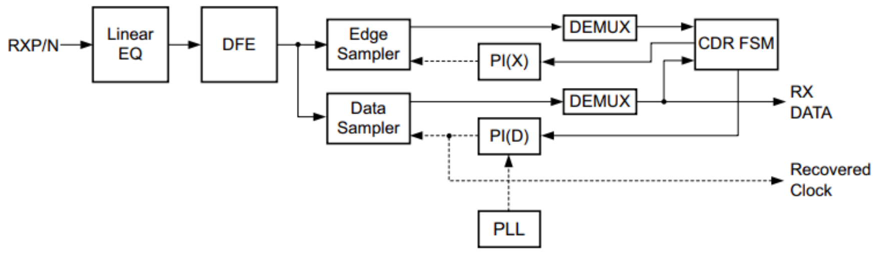


Figure 4.7: Clock Data Recovery Circuit Detail

- **RX Fabric Clock Output Control**
  The RX clock divider control block performs two main functions, serial clock divider control and parallel clock divider and selector control. A desired clock used by the receiver block can be driven out, in order to be used in the design logic by the user.
- **RX Polarity Control**
  In the event RXP and RXN differential traces are swapped by accident on the PCB, then the differential data received by the receiver RX are reversed. In order to counter this we invert the parallel data in the PCS after the SIPO to offset the reversed polarity of the differential pair.
- **RX Byte and Word Alignment**
  In order to convert serial data to parallel data, with specific length words, we need to know where are the boundaries of each word, and align the received data with the data sent by the transmitted. In order to achieve this alignment the transmitter sends a recognizable sequence, usually called a comma. The receiver searches for the comma word in the incoming data and when it is found all of the subsequent serial data are organized, in order to match the words sent by the transmitter.
- **RX 8B/10B Decoder**
  The GTH transceiver has a built in 8b/10b Decoder in the Receiver block. If data are encoded in this scheme then in order to decode them without consuming additional device resources, this block is enabled by the user.
- **RX Elastic Buffer**
  As shown at figure 4.6 the RX datapath has two internal parallel clock domains. The RXUSRCLK parallel clock domain in the PCS and the XCLK

parallel clock domain in the PMA. The two clocks in order to receive data from the PMA to the PCS need sufficiently close rates and all phase differences between the two, resolved. In order to achieve this, the GTH receiver includes the RX Elastic Buffer.

- **RX Buffer Bypass**
  The Elastic Buffer can be bypassed to achieve a lower latency. The RX phase alignment circuit is used to adjust the phase difference between the SIPO parallel clock domain and the RX XCLK domain. The RX XCLK is configured to use RXUSRCLK when using RX phase alignment.

- **RX Synchronous Gearbox**
  When the 64b/66b encoding scheme is used, the RX Synchronous Gearbox can provide support in order to separate the Header from the payload of the word. The RX Synchronous Gearbox receives data from the parallel XCLK PMA clock domain and for the reasons described at 4.1.4, the output data occasionally are invalid. In order to keep track, the RX Synchronous Gearbox provides us with output ports RXHEADERVALID and RXDATAVALID. In contrast to the TX Synchronous Gearbox, the RX Gearbox internally manages all sequencing and deasserts the signals of the valid ports. The RXDATAVALID signal, for instance, is deasserted every 32 cycles for the 4-byte internal datapath width when the RXDATA width is 64 bits, that follows the same sequencing as the TX. The potential RX synchronous gearbox interfaces are 2-bytes, 4-bytes, and 8-bytes.

  Data out of the RX Synchronous Gearbox are not necessarily aligned. The user is responsible for the alignment logic and must utilize the RXGEARBOXSLIP port. The port causes the contents of the gearbox to slip to the next possible alignment. Simultaneously, the user logic design checks for alignment markers or control words that are predefined. If, after a certain amount of clock cycles, no alignment markers are found the user logic reasserts the RXGEARBOXSLIP port and repeats the process until correct alignment is reached.

- **RX Interface**
  The RX interface is the gateway to the RX datapath. Data are received through the RXDATA output port by reading them at the positive edge of the RXUSRCLK2 clock. The output port width is determined by the RX_DATA_WIDTH and can be se to 16,20,32,40,64 or 80 bits. The TX interface is divided in two parallel clock domains, the PCS RXUSRCLK2 clock domain and the RXUSRCLK2 which is the clock that provides the user the data. RXUSRCLK and RXUSRCLK2 have a fixed-rate relationship

which is the same relationship the TXUSRCLK and TXUSRCLK2 have, as shown at table 4.3. The two clocks must be positive-edge aligned, while the equation that is required to calculate the RXUSRCLK frequency is the same as the equation 4.1.

4.2 A Synchronous Link Protocol

4.2.1 Introduction

In the context of learning about the protocols used in optical links at CMS and FPGAs in general, a simple synchronous link protocol was created. All tests and development were done using the AMD Kintex UltraScale FPGA KCU105 Evaluation Kit, with the XCKU040-2FFVA1156E chip. The board contains 20 GTH transceivers (arranged in five Quads), two of which are connected to SFP+ connectors which we can utilize with SFP modules and fiber-optic cables [24].

The protocol implemented is a synchronous link protocol. As depicted at figure 4.8, protocols can be categorized into two types, synchronous and asynchronous. Synchronous link protocols are established when the user produces data at the same clock domain as the MGT's TX interface clock (TXUSRCLK2) operates. When the user's clock and the TXUSRCLK2 are the same, data can flow without fear of data loss or corruption. If the user's clock is not the same as the TXUSRCLK2, then the two frequencies may not be the same and/or the phases of the two clocks may not be aligned. In order to cross clock domains and prevent loss of data or corruption, a dual port FIFO is used with the two clocks connected as input clocks. The frequency of the user clock needs to be of a lower value than the TXUSRCLK2, otherwise the FIFO would receive way more input words than it can output and it would fill up, thus discarding new words and lose data.

The protocol is based on the 64b/66b encoding scheme. This choice is based on the reduced overhead of the 64b/66b encoding in compared to 8b/10b. The overhead of the 64b66b encoding is 2 header bits for every 64 payload bits, which leads to 3.125% overhead. In contrast the overhead of the 8b10b encoding is 20%. When comparing the two we can see that the 64b/66b encoding is more efficient at 96.875% efficiency compared to 8b/10b encoding scheme's efficiency at 80%.

The Line Rate of the Link is designed to be 10.3125 Gb/s. The QPLL of the next quad, quad($n + 1$), is used because of it having a reference clock input connected to a Si570 programmable low-jitter LVDS differential oscillator [24], which is set to 156.25 MHz. In addition the transceiver needs a free-running clock in order to be able to reset itself, which is this case is provided by a system clock

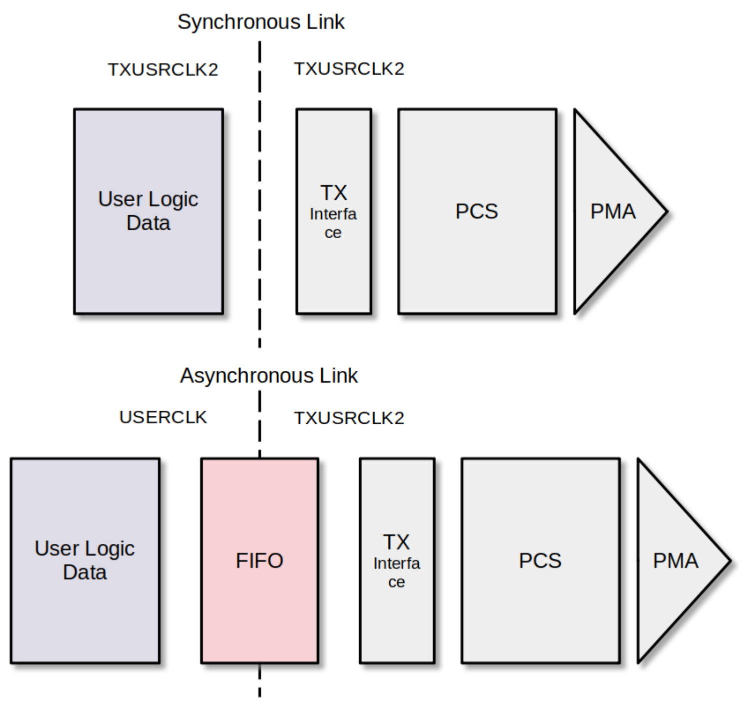


Figure 4.8: Synchronous - Asynchronous Link Protocols with Clock Domains

produced by a Si5335A LVDS Clock Generator, set at 125 MHz. The internal datapath width of both the receiver and the transmitter is set to 4-bytes (32 bits), which requires the condition of 4.3 is met, $F_{T/RXUSRCLK2} = F_{T/RXUSRCLK}/2$. A synchronous gearbox is used and a PCS buffer is enabled at trasmitter and the receiver as well.

4.2.2 Protocol Logic Modules

Figure 4.9 shows a visualization of the protocol as blocks. Each block serves a distinct purpose and each block's purpose is discussed in further detail.

- **Top Module**

  In general we refer to the Top module as the highest module in Hierarchy. It is a module that contains all other blocks and its ports serve as a gateway to the outside world, in the sense that they are constrained by the user within XDC files. That means that the user creates the correct ports, input/output data ports, reset ports, status ports, and ties others to pins on the board, such as clocks, GPIOs , buttons and so on.

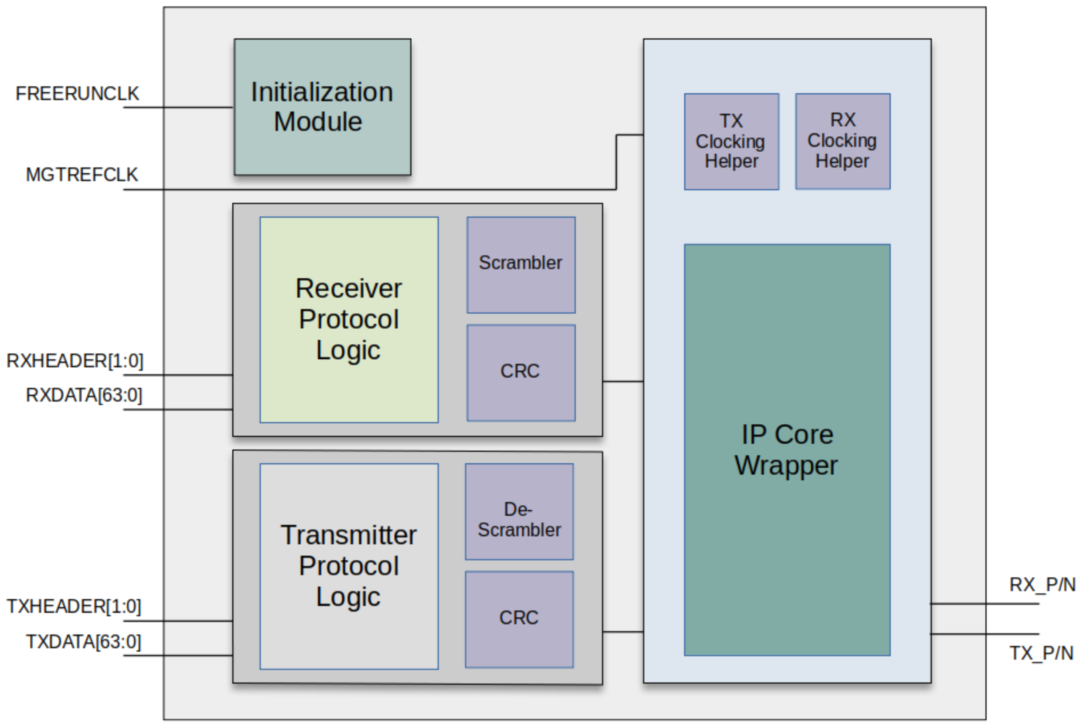


Figure 4.9: Protocol Logic Diagram Blocks

In our design the top module includes all of the blocks depicted in figure 4.9. Its main function is to connect each block with the others using internal signals. Other functions include routing and buffering all the resets, as well as instantiating the IBUFDS buffer primitive as stated in 4.1.3, in order to drive the MGT reference clock.

All reset signals are asynchronous to the user's logic clock domain, which in this case is the TXUSRCLK2 because this is a synchronous link protocol. Reset signals may be asserted by the user via VIO and push buttons, but they may also be triggered by state machine of the protocol, which are using the Free-running clock. Not only resets, but also many other signals used by the user need to cross their clock domain and use some other. This is referred to as Clock Domain Crossing (CDC) and requires some techniques in order to overcome Metastability issues in our design. One technique used to combat this, it to use a Dual Flip Flop Synchronizer fig 4.10, where the metastable signal of the first flip flop is stabilized by the second flip flop, before the user receives the signal. In order for this to work, the signal's origin clock must have a lower frequency than the one the signal crosses. Another way to cross large amounts of data to another clock domain is by using FIFOs, as discussed in the asynchronous protocol.

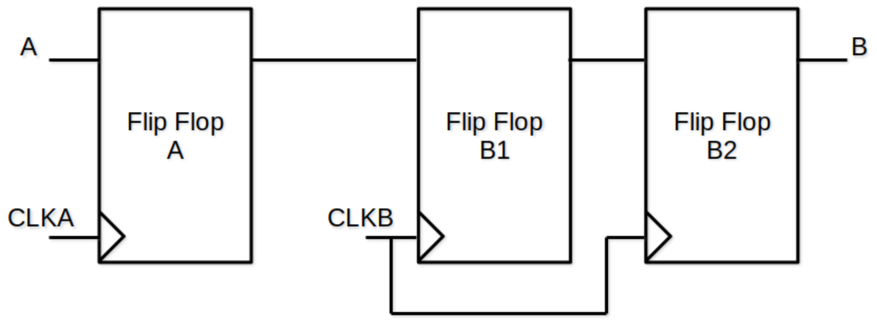


Figure 4.10: CDC with Dual Flip Flop Synchronizer Logic

- **Wrapper Block**

  The wrapper module is provided by the Vivado transceiver's IP. It contains the IP core and two Clocking Network Helper blocks. The two Helper blocks are responsible for creating and buffering the T/RXUSRCLK and T/RXUSRCLK2 from the T/RXOUTCLK port of the master transceiver channel via BUFG_GT buffer primitives, one block for the transmitter and one for the receiver. The relationship of the clocks depends on the settings table 4.3 contains.

- **Reset Procedures**

  The Top module contains all reset procedures the transceiver needs to follow in order to Initialize and configure itself, after user input or system power up. The reset procedure follows the next steps, in the following order:

  1. Reset the associated PLL driving TX/RX.
  2. Reset TX and RX datapaths (PMA and PCS)

  The associated PLL used needs to be reset first. Once the PLL locks we can reset the datapath, with PMA reset first and PCS second. We can categorize the resets in three groups. First group contains signals used to reset the TX transmitter, the associated PLL reset signal and the TX datapath reset signal. Second group contains signals used to reset the associated PLL of the RX receiver and the RX datapath. Third group contains the Reset All signal, responsible for resetting all logic, while it triggers the signals of the other two groups. It is used in order to avoid redundant PLL reset sequences.

- **Initialization Module**

  The Initialization module is provided by the Vivado transceiver's IP. The module contains the state machine shown at figure 4.11 designed to assist and monitor system bring-up. The state machine is activated whenever a Reset All is triggered by the user. It then monitors for timely reset completion and successful TX and RX initialization, retrying resets as necessary. [25]. When Initialization is completed, the machine monitors the RX Data Good indicator and resets whenever link loss occurs, while incrementing a counter used for debugging purposes by the user.

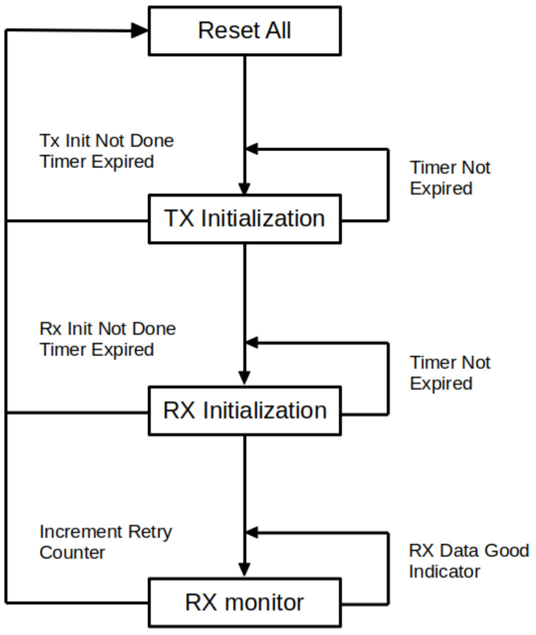


Figure 4.11: Initialization Module State Machine

- **Cyclic Redundancy Check (CRC) module**

  Cyclic Redundancy Check (CRC) is an error-detection code commonly used in digital networks to detect accidental changes to digital data. The CRC technique creates and attaches a certain amount of extra bits, also called a checksum, to the transmitted message. The CRC checksum is calculated based on the remainder of a polynomial division, modulo two, of the message's bits. The receiver can then perform the same calculation and in the event the two checksums do not match we know that there is data corruption [26]. A logic diagram of a serial calculation of CRC-32 using Linear Feedback Shift Registers (LFSRs) is depicted at figure 4.12, where XOR gates are placed in the same positions as the coefficients of the polynomial 4.2. For this protocol the same CRC-32 polynomial is used, as defined in the IEEE 802.3 Ethernet Standard [27]. Because the serial implementation would need 64 clocks cycles in order to calculate the checksum, a parallel implementation is used based on an open source code. This way time spent on calculating the checksum significantly decreases to just one clock cycle.

$$G(x) = x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1 \tag{4.2}$$

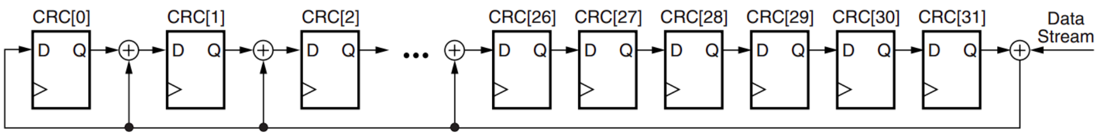


Figure 4.12: Linear Feedback Shift Register Implementation of CRC-32

- **Scrambler - Descrambler block**

  In contrast to the 8B/10b encoding scheme, a separate scrambler block is needed in order to encode the 64b/66b data. This is done in order to achieve DC-Balance, allow receiver reference clock recovery and to maintain a small Run Length. The protocol uses a self-synchronous scrambler, with a 58-bit polynomial provided by Xilinx for 64b/66b encoding. The descrambler block performs the inverse operation in order to return the data to their original form.

- **Transmitter Logic module**

  The TX module contains all logic functions related to the protocol from the transmitter side. In figure 4.13 we can see the module's input and output pins, on the left and right side respectively. A reset all input is connected in order to reset all logic inside the block, while a tx_active pin is monitored for link loss. The txusrclk2 is provided by the clocking helper block in the IP wrapper and all logic is implemented in the same (txusrclk2) domain.

A special, error injection, control pin is connected as a means to test data-loss detection. User data 64-bit words, are clocked with the txusrclk2 and inserted along with a data valid bit. The data valid bit specifies when user data are available to be transmitted.

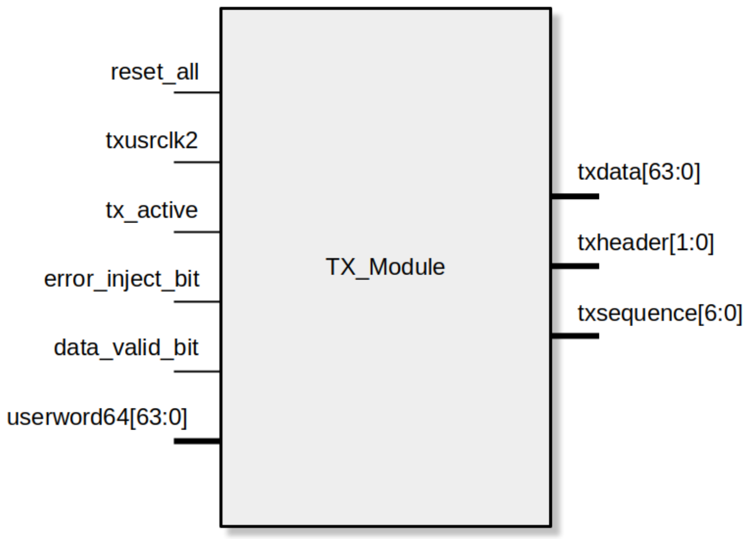


Figure 4.13: Transmitter Logic Block

The protocol produces two types of words, a control word and a data word as shown in figure 4.14. The Header (2 bits) of the 64b/66b word declares what type the word is. Header bits equal to ”01” declare data words while ”10” header bits declare a control word. The protocol does not produce ”11” or ”00” header bits, in order to maintain a small Run Length, when transmitted with the rest of scrambled word. This is done because the header word does not pass through the scrambler block. Whether the word contains data by the user is indicated by the data valid bit and the user is responsible to assert ”HIGH” whenever data are available. The protocol logic then produces a header with ”01” bits, and forwards the message. If the data valid bit is ”LOW” the header bits ”10” are produced and a control word is transmitted. In all instances except one, the control word transmitted is the word ”0x5555555555555555” and it is used for synchronization by the receiver.

Inside the module the txsequence counter that is necessary for the TX synchronous gearbox driven by the txusrclk2, is also implemented and independent by the user inputs. The counter counts up to 32 decimal, at which point

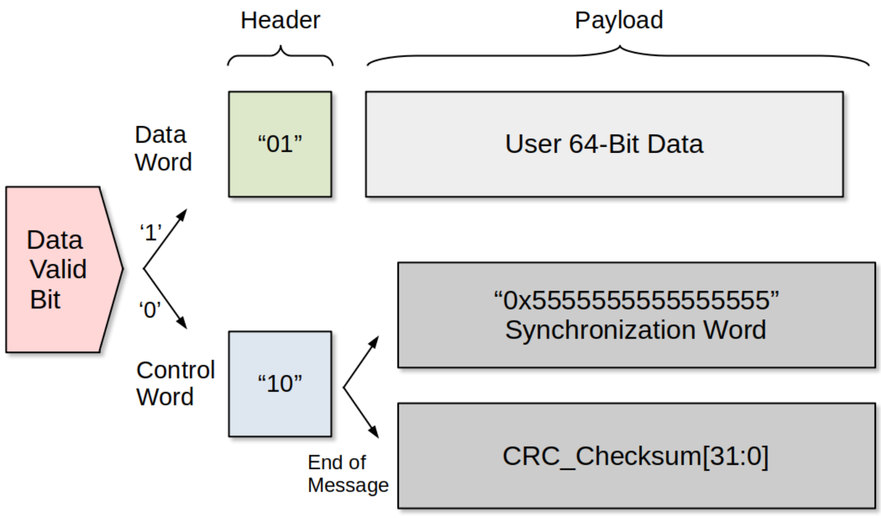


Figure 4.14: Data and Control Words with their corresponding Headers

a stopdata signal stops the flow of new words from being driven out of the block. The counter then resets to zero and starts incrementing again.

The data are passed through the CRC calculator block and the scrambler before exiting the block. The CRC checksum is stored and when no more available data are present the user sets the data valid bit "low". The CRC checksum is then attached, at the next clock cycle, to the end of the message and transmitted last, figure 4.15 . The txdata (64-bit) are driven out by the txusrclk2 and inserted to the TX interface inside the TX wrapper, along with the txheader (2-bits). The tx sequence is driven out as an output bus in order to be used by the TX synchronous gearbox as stated in 4.1.4, but also for user monitor and debug purposes.

The VHDL code for this module is presented at appendix A.1

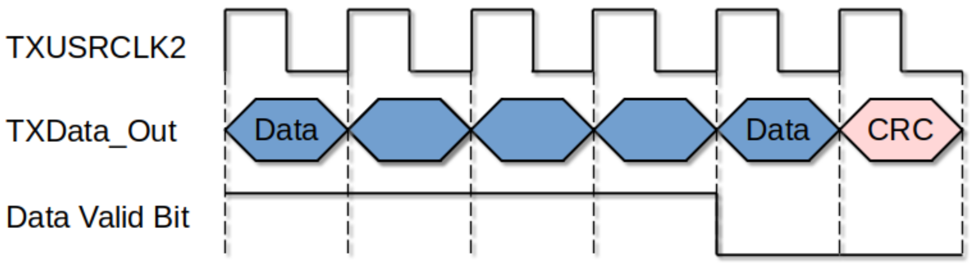


Figure 4.15: CRC Checksum transmission at the end of message

- **Receiver Logic module**

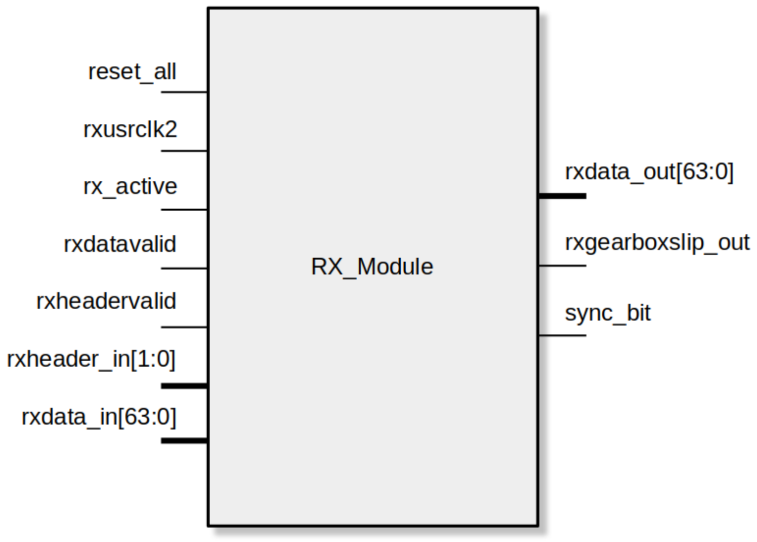


Figure 4.16: Receiver Logic Block

The Receiver module is responsible for data alignment and error-detection. The inputs and outputs pins can be seen at the left and right side at figure 4.16. A reset all pin is present in order to prerform a reset to all logic processes inside the block. when triggered by the user. A rx active pin is present, allowing the receiver to monitor for any link loss and other malfunctions, so as to trigger a reset without user input. A pin is available for clock input, thus connecting the rxusrclk2 as required for the case a synchronous gearbox is enabled.

Rx data 64 bits containing the payload of the word flow for the the RX interface and are clocked into the module with rxusrclk2, The RX header is also provided in the same way, utilizing the rxdata_in[63:0] and rxheader_in[1:0] ports respectively. The receiver synchronous gearbox as stated in 4.1.5 performs all sequencing internally and is thus able to conclude whenever data and their header are valid, while the input ports rxheadervalid and rxdatavalid allow the receiver to monitor this procedure.

Initially all words from the RX interface are scrambled, except the header, so they are passed through the descrambler block to return to their normal form. However, even though the data are not scrambled at this point, there is still no guarantee that the RX gearbox has correctly identified the boundaries for each word upon receipt. This implies that the RX gearbox may deliver a part of a word along with a part of what would be the next word, viewed from the transmitter side. To ensure the receiver aligns its 64 bit frame to the correct boundaries and outputs the same word as sent by the transmitter, the rxgearboxslip is triggered. The rxgearboxslip performs a bitslip and moves the boudaries of the frame by one bit.

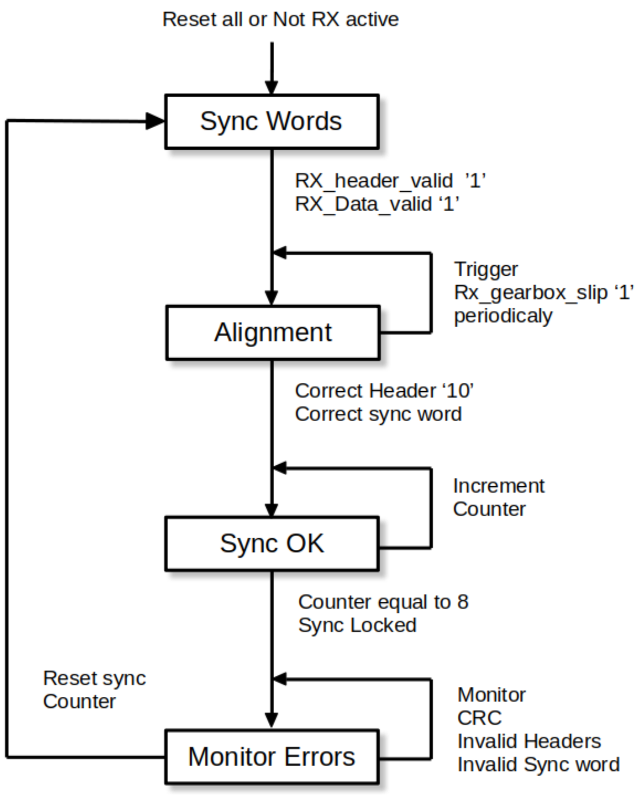


Figure 4.17: Receiver Synchronization Process

The receiver synchronization process is depicted at figure 4.17. After a reset signal, the receiver enters a state of waiting for control words. It then observes signals rxdata_valid and rxheader_valid, and when both are set "HIGH" the receiver first inspects the words for the correct header. Only when the correct header is found, the receiver compares the payload of the word to the predefined word for alignment, "0x5555555555555555". If the header received is an illegal header "11" or "00", or even a data word header, a rxgearboxslip counter is incremented. If the header correctly indicates a control word but the payload is not the predefined word, the rxgearboxslip counter increments again. When the counter reaches 32 decimal, the rxgearboxslip is driven "HIGH", and after one clock cycle set to "LOW" while the counter resets to zero. This is done in order to allow time for the rx gearbox to correctly rearrange its frame, repeating the same process as many times needed until alignment is achieved. When both the header and the synchronization word are correctly read by the receiver then another counter increments. The counter needs to reach a decimal value of 8 for the synchronization OK bit to be set to "HIGH." If an invalid header or a an invalid synchronization with correct header is read then this counter resets to zero. This logic is implemented as a safety measure, making sure 8 correct words are received without any errors in between. The receiver after sync OK bit is asserted "HIGH" is read to receive data words and control words containing the CRC checksum at the end of message, enabling in turn the CRC logic block whenever user data are received. Upon CRC checksum calculation the receiver is able to determine if there was any data-loss, although not making known the quantity of the bit flips. Whenever the Header, the synchronization word or the CRC checksum is different than expected the counter resets to zero, repeating the process again.

The sync OK bit is driven out and used by the Initialization module when the state machine is at RX monitoring stage, but also monitored by the user via various methods. After all steps have been successfully executed the rxdata_out port delivers the message 64-bits to the user, clocked at the rxusrclk2 clock domain.

4.2.3 Normal Operation

A conventional exchange of information, in the normal operation of the protocol is as follows:

1. Transmission of synchronization words.
2. Receiver Alignment.
3. Transmission of data + CRC checksum.
4. Data reception + CRC evaluation.

The creation of an additional module, from now on referred to as the "userside" module, was needed for the purpose of testing the link protocol by following the aforementioned steps. The module provides the appropriate user created words that need to be transmitted. This is performed with the help of an incrementing counter, clocked at txusrclk2. While the counter is below a certain value, the data valid bit is set to "HIGH" as the contents of the counter are transmitted as user words, and when the counter reaches a set value it drives the data valid bit

”LOW”, thus the receiver is able to transmit the synchronization words in those available slots. The userside module also controls some functional pins like reset, reset rx datapath , error injection and loopback.

There are two methods the user can test a link protocol in a single board. The first method is by using an external loopback and the second method is by using an internal loopback. In the KCU105 board as mentioned in 4.2.1, two transceivers are connected to SFP+ pluggable connectors. With an Optical Transceiver module that plugs into the connector, the user can connect an optic fiber cable from the transmitter of the module to the receiver. Figure 4.18 shows the KCU105 board with a single optic fiber cable connected to one transceiver SFP+ modules. The module used can support a maximum data rate of 10.3125 Gb/s over a 300 meter optic fiber cable [28].

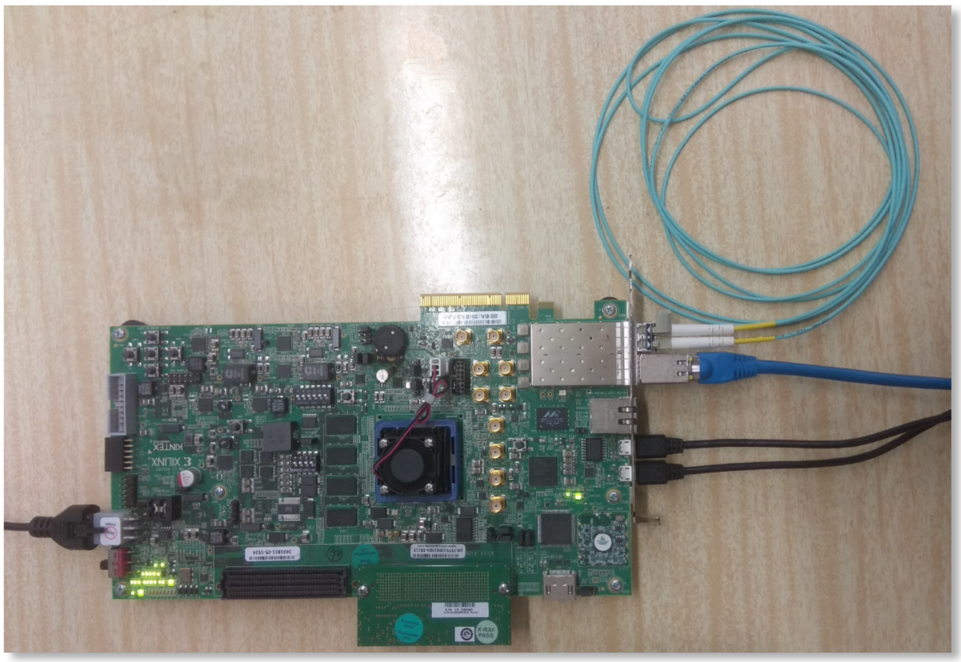


Figure 4.18: KCU105 Board with connected Optic-Fiber cable and RJ45 cable

The second method to establish a loopback between the transmitter and the receiver is based on an internal property of the transceiver, the internal loopback feature. In the internal loopback mode the transceiver datapath is configured in a way which the traffic stream is folded back to the source, as shown in figure 4.19. A 3-bit input port selects which loopback mode the transceiver will follow, as shown in table 4.4.

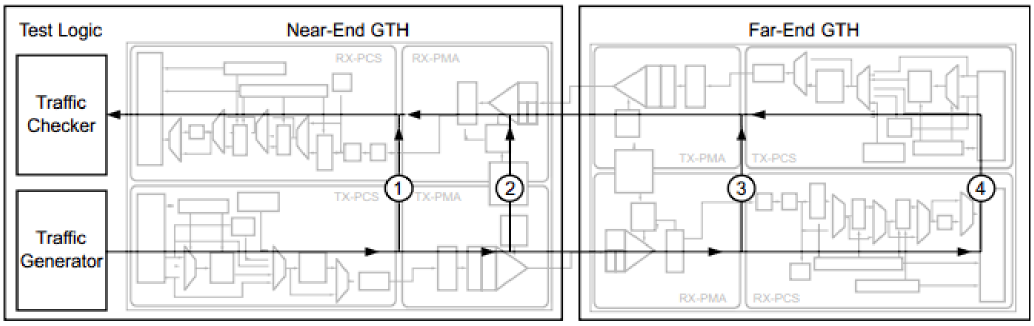


Figure 4.19: Internal Loopback view, with numbered mode paths

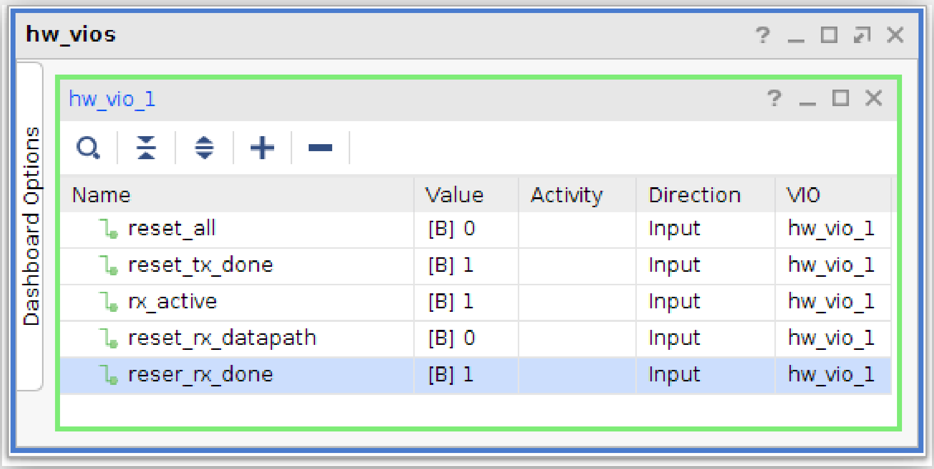
There are two main categories, the Near-End loopback modes where the loopback occurs at the same end where the signal originates and the Far-End loopback modes where the signal travels over the entire communication path. Near-End loopback modes are more focused on internal transceiver functionality, while Far-End loopback modes test the entire communication path.

Port	Path	Description
000	-	Normal Operation
001	1	Near-end PCS Loopback
010	2	Near-end PMA Loopback
011	-	Reserved
100	3	Far-end PMA Loopback
101	-	Reserved
110	4	Far-end PCS Loopback

Table 4.4: GTH transciver Loopback Port Modes

To assess and confirm whether the link protocol is functioning as expected, the deployment of several IP tools is necessary. The three tools used are the Xilinx VIO IP package, the Xilinx ILA IP package and the IPbus protocol.

The VIOs IP package allows the user to monitor and drive internal device signals. In figure 4.20 we can see the signals that are being monitored and their real time values. The user is also able to set a value to the signal, only if the IP is declared properly so and the specific port set as an output port. To use the VIOs IP the user must include its component in the RTL design and manually connect each desired pin to the component's ports [29].



Name	Value	Activity	Direction	VIO
reset_all	[B] 0		Input	hw_vio_1
reset_tx_done	[B] 1		Input	hw_vio_1
rx_active	[B] 1		Input	hw_vio_1
reset_rx_datapath	[B] 0		Input	hw_vio_1
reser_rx_done	[B] 1		Input	hw_vio_1

Figure 4.20: VIOs transceiver internal signals

While using the VIO IP may be acceptable when monitoring relatively stable signals, the same technique cannot be applied for very high frequency signals. The ILA IP package allows users to insert probes into their designs to monitor and capture signals at various points within the design, at system speeds. These signals are connected to the probe inputs and sampled when a trigger occurs, storing all data to on-chip Block RAM (BRAM). Several triggering options allow the user to capture specific events or conditions of the design. After the trigger occurs, the sample buffer is filled and uploaded into the Vivado logic analyzer, where the data are displayed in a waveform window [30]. The user can view the otherwise inaccessible sample of those signals. For the evaluation of the protocol ILA probes were placed at the transmitter module and the receiver module.

Figure 4.21 shows the ILA data for the start of the message at the transmitter module. Initially synchronization words, with txheader "10" and predefined word "0x5555555555555555", are transmitted as shown. The counter pattern created by the userside module then enters the transmitter module, shown as the userword64_in[1]. After a few clock cycles the data are processed and driven out of the module after being scrambled along with the data word header "01", shown by the txdata_out and txheader_out probe respectively, from where they will be inserted to the IP core wrapper.

When user created counter words are no longer available, the transmitter appends the CRC checksum at the end of the message and transmits it with the

[1]Probes with 64 bits only display their MSBs, thus virtual probes are created and displayed with the 32 bit LSBs view, for the user's convenience.

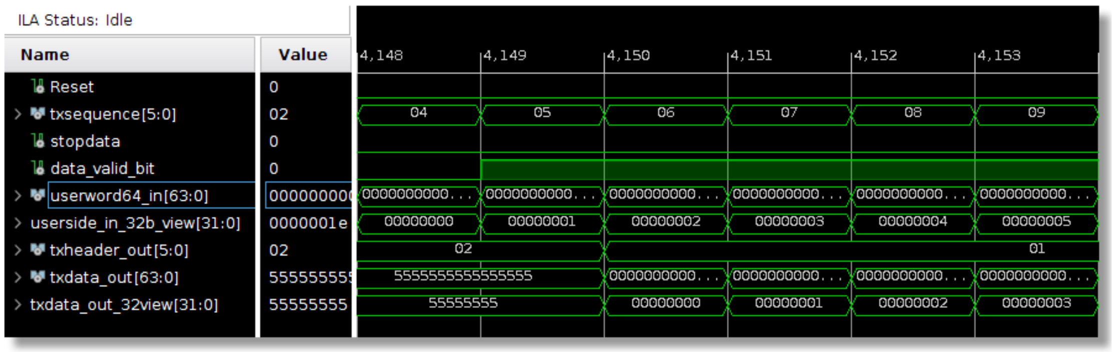


Figure 4.21: TX ILA start of message

header ”10”, thus distinguishing it from the rest of the message sent. We can see the 32 bit checksum attached at the end of the message at the txdata_out probe, shown at figure 4.22.

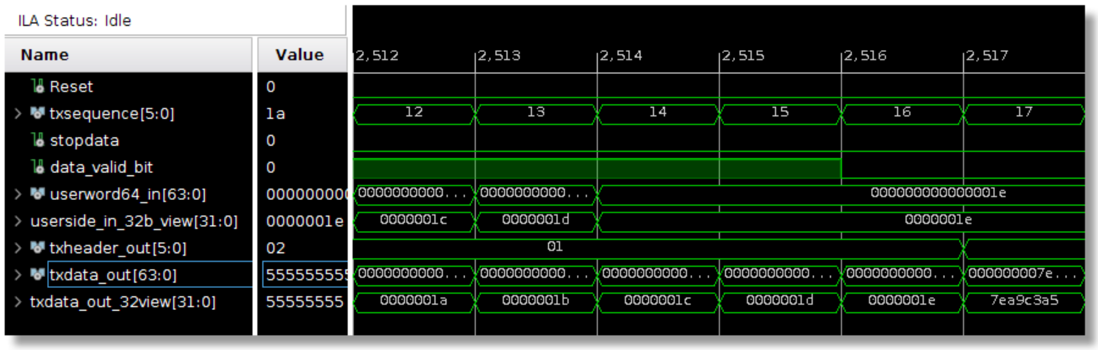
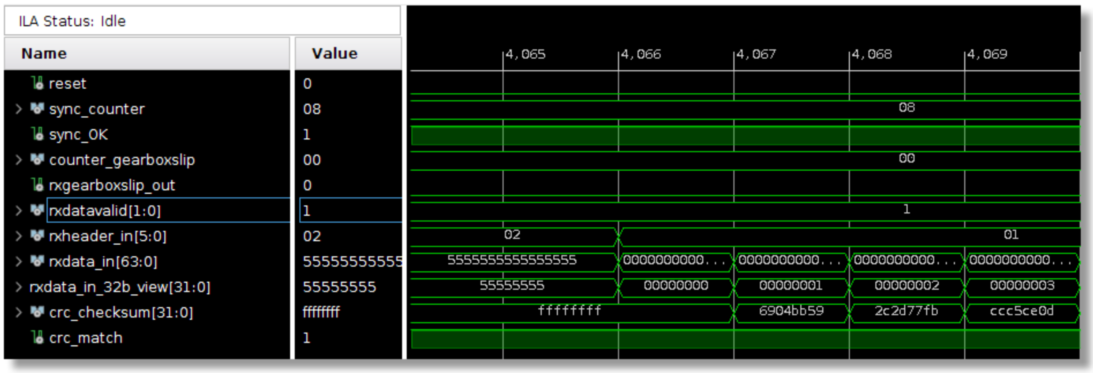


Figure 4.22: TX ILA end of message and CRC Checksum

Data arrive at the receiver side and are inserted at the receiver module in order to be processed before presented to the user. In figure 4.23 the synchronization words appear at the rxdata_in probe, along with their header indicated by the rxheader_in probe. The sync_OK probe indicates that the link alignment has already been achieved. When the header changes to indicate data words, the counter pattern from the userside module can be seen arriving in the correct order, after being de-scrambled. The CRC checksum calculation module also receives the data words and begins generating its own Checksum, which is locally stored, as shown by the crc_checksum probe in figure 4.24. When the last data word is transmitted and the header changes from ”01” to ”10”, the receiver expects the control word containing the CRC Checksum calculated by the transmitter. The receiver then checks its own stored Checksum value with the one received and if

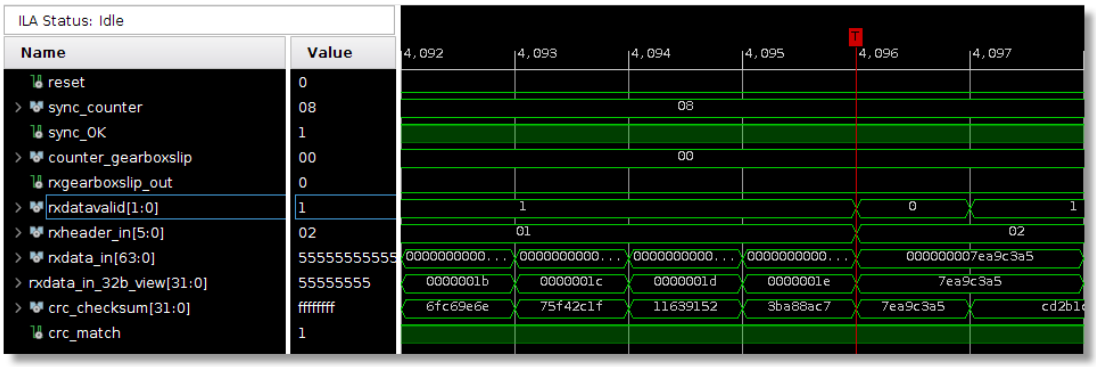


ILA Status: Idle

Name	Value
reset	0
sync_counter	08
sync_OK	1
counter_gearboxslip	00
rxgearboxslip_out	0
rxdatavalid[1:0]	1
rxheader_in[5:0]	02
rxdata_in[63:0]	55555555555
rxdata_in_32b_view[31:0]	55555555
crc_checksum[31:0]	ffffffff
crc_match	1

Figure 4.23: RX ILA start of message

the two are found to be identical then the crc_match signal is driven "HIGH" and there is no data loss. If the two values are found to be different the crc_match signal is driven "LOW" and indicates loss of data, while the signal resets the transceiver as stated in 4.2.2.



ILA Status: Idle

Name	Value
reset	0
sync_counter	08
sync_OK	1
counter_gearboxslip	00
rxgearboxslip_out	0
rxdatavalid[1:0]	1
rxheader_in[5:0]	02
rxdata_in[63:0]	55555555555
rxdata_in_32b_view[31:0]	55555555
crc_checksum[31:0]	ffffffff
crc_match	1

Figure 4.24: RX ILA end of message and CRC evaluation

The last evaluation tool using the IPbus protocol will be discussed in greater detail in the next section.

4.3 IPBus

4.3.1 The IPBus protocol

The last few decades the electronics systems of particle experiments were based on the VMEbus standard. The new eletronics systems are now based on the newer ATCA standards, where specifications incorporate industry standard serial communication technologies such as Gigabit Ethernet, but do not explicitly specify the hardware access protocol for reading and modifying with software applications the memory spaces of the boards from external devices. The control system of an experiment needs to have a reliable and predictable behaviour at all times, since it is responsible for monitoring and debugging all hardware, but also needs to be highly scalable. The necessity of such a protocol led to the creation of the IPbus protocol.

The IPbus protocol is a simple packet-based control protocol for reading and modifying memory-mapped resources within FPGA-based IP-aware hardware devices which have a virtual A32/D32 bus. The protocol assumes the existence of a virtual bus with 32-bit word addressing and 32-bit data transfer, with the 32-bit data width fixed [31]. The protocol can perform the following operations:

- **Read** A read operation of user definable depth. There are two types, an address incrementing read that is used for multiple continuous registers and a non address incrementing read used for ports or FIFOs.
- **Write** A write operation of user definable depth. Categorized in two types, the same as with read operations.
- **Read-Modify-Write bits (RMWbits)** A bit masked write operation, that allows the modification of a selective group of bit within a 32-bit register.
- **Read-Modify-Write sum (RMWsum)** An increment operation, that allows the addition of values to a register.

The User Datagram Protocol (UDP) has been chosen as the transport protocol in order to reduce FPGA resource usage. The IPbus protocol operates in a transactional manner. The client, normally through software, transmits a request to the device for every read, write or RMW operation. In turn the device responds with a message indicating an error code equal to 0 for a successful transaction and in the case of reads includes the returned data. To reduce latency it is possible to combine multiple transactions into a single packet. Following a client request or instruction an packet is formed, containing the header which is the first 32 bits of the packet that specify the packet type and ID fields, followed by the IPbus transactions. The IPbus packet now contained within the payload of the UDP transport protocol packet, which in turn is contained within an IP packet and wrapped within an Ethernet packet, shown in figure 4.25, is transmitted to the device. The packet returned by the hardware device follows the same format, with the exception of the transaction field, containing the relevant response [32].

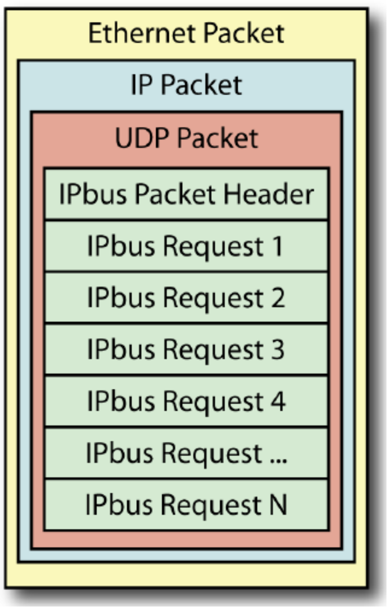


Figure 4.25: IPbus, UDP, IP, Ethernet Packet Structure

A suite of IPbus software and firmware components can be used to create a reliable and scalable control system. The suite consists of the following components [31]:

- **IPbus firmware** The implementation of the IPbus protocol within end-user hardware FPGAs. The firmware module is an implementation of a UDP server in VHDL, as it interprets IPbus transactions on an FPGA. The module has been designed with ease of integration in mind. The user is able to include the module inside the rest of the user logic design, on the same FPGA, while only using resources available inside the FPGA. The module additionally incorporates the echo request/reply functionality from ICMP, a feature used by the Unix *Ping* command.

- **ControlHub** The ControlHub is a software application that establishes a central access point for IPbus control over individual devices. More specifically, it manages simultaneous hardware access from various control applications to one or more devices while it also incorporates the IPbus reliability mechanism for handling UDP packets between itself and the hardware devices. The implementation of the ControlHub is done in Erlang, a general-purpose, concurrent programming language capable of constructing high-availability, fault-tolerant applications. In Erlang the fundamental structural unit is the process. The ControHub then uses a distinct Erlang process for each connected client and IPbus device, this way ensuring the distribution of the workload across multiple CPU cores. Since ControlHub is a software application and no additional resources are needed from the hardware device (FPGA), the TCP transport protocol is used to communicate with the uHAL interface, which uses far superior congestion mitigation and flow control algorithms. The ControlHub can be considered as a network switch that listens for TCP connections on port 10203.

- **uHAL** uHAL is the Hardware Access Library (HAL) that provides the user (client) with C++/Python API for IPbus read, write, RMW transactions. It uses a delayed dispatch model where numerous transactions are queued and concatenated within the payload buffers of the transport layer until either the dispatch method is used or the command queue surpasses the maximum packet size. In order to address each device's register, XML files are created by the user and used, where each node of the XML tree represents a register, a BRAM or a FIFO. One XML file can reference other address files, this way with minimal effort a similar hierarchical structure to the firmware design is created. The uHAL interface to each device is separated into two operation modes. The local-client mode, where the uHAL library interfaces directly with the device over the UDP transport protocol. The remote-client mode, where communication with the hardware devices is performed exclusively via the ControlHub, in order to deploy more reliability measures. The connection file used to indicate the location of the hardware, specifies which type is used.

4.3.2 IPBus Firmware

The IPbus firmware is implemented in VHDL. Several modules can be included in any project in order to provide control over specific user-defined registers inside the design. The most significant modules of the protocol are listed below.

- **IPBus Infra** The IPbus Infrastructure module is responsible for the implementation of the UDP server. This is the core of the IPbus protocol, where UDP packets are received by the client software application. The UDP packet's payload is converted to the IPbus packet and subsequently sent to the IPbus fabric module. The process is reversed whenever a response from the device is performed.

- **IPBus Fabric** The fabric module of the IPbus addresses the correct registers that the received IPbus packet specifies. This is performed with the additional information the decoder file provides. It is responsible to match the IPbus packets with their corresponding registers from the Address Table (XML file), while executing the transactions contained in those packets.

- **IPBus Decoder** The IPbus decoder is a package providing the function called by the fabric module. The function provides the information for assigning the correct address of a register addressed by the client on the software side to the appropriate register on the firmware side. The file is created by a Python script called *gen_ipbus_addr_decode* by reading the address table the XML file contains. This script is included with the IPbus suite. The script produces the appropriate decoders for each node of the table. When creating a project this file needs to be created by the user and included in the design.

- **IPbus Slaves** On the software side each register is declared by the address table. On the firmware side each register is translated to slave modules that the user must include inside the design. There are many types of slave modules, the most common type being the *ipbus_ctrlreg_v* slave, which represents a bank of control / status registers of configurable size, with read access to status register and read-write access to control registers.

A simple point to point local connection between a client PC and a server Hardware FPGA is shown in figure 4.26.

4.3.3 IPBus integration with the Synchronous Protocol

The IPbus protocol was integrated within the Synchronous Protocol project. This was the last evaluation tool used to monitor and control the design. The client PC connects with the board via an RJ45 cable, connected to an SFP to Ethernet module on a transceiver channel on the board, using the 1000BASE-X Gigabit Ethernet Standard.

The necessary modifications were made inside the firmware and the available modules for the instantiation of the IPbus firmware were included. A *ipbus_ctrlreg_v* slave was used in order to control several registers, while also monitoring others, while the *gen_ipbus_addr_decode* script was used in order for the decoder package tobe created and included in the project libraries. The same

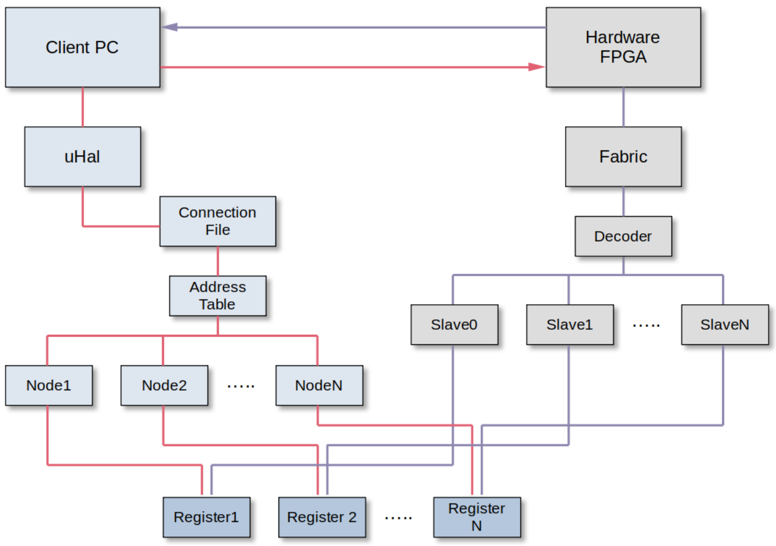


Figure 4.26: Logic Diagram of IPbus protocol communication between client and server

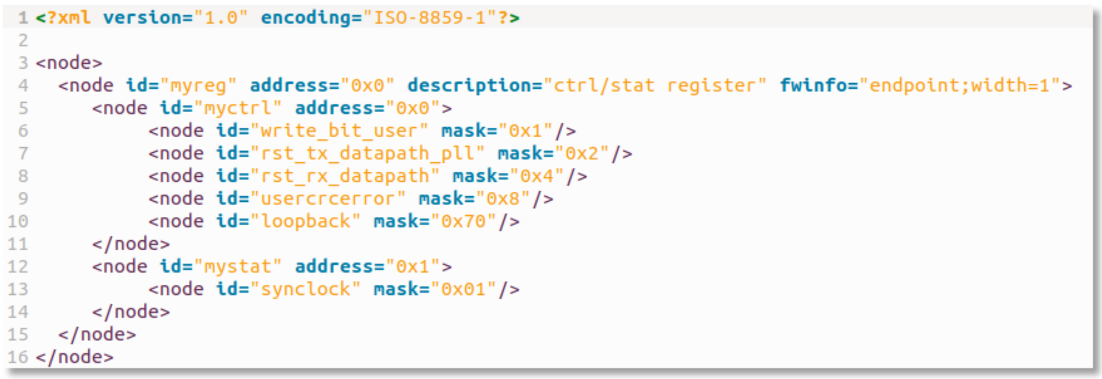
ports that were monitored and controled by the VIO tool, were also connected to the slave's input and output ports, thus able to create a trigger to the signals either from the VIO tool or from the IPbus protocol.

Once the integration of the protocol was completed, the control from the software side was implemented. A connection file was created in order to interface between the Python script and the uHAL interface. The connection file specifies the id of the hardware board the script wishes to talk to, the IP location of the board and the address table file's location, shown in figure 4.27.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<connections>
  <connection id="KCU105board"   uri="ipbusudp-2.0://192.168.200.16:50001"
  address_table="file://ipbus_control.xml" />
</connections>
```

Figure 4.27: IPbus Connection File for the Link Protocol Integration Tests

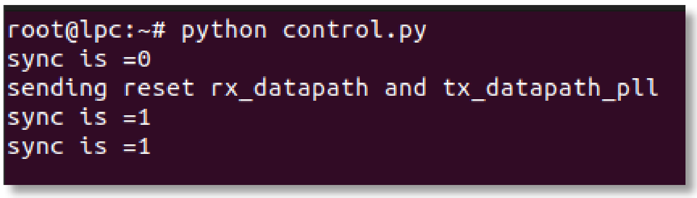
The Address Table used can be seen at figure 4.28. The table specifies the address of the one slave implemented at the firmware side. Since the slave can be used both for control and monitoring status the address table is separated in two nodes, one for control signals and the other for status signals. The addresses of the nodes inside the top node are calculated by adding their value to their parent's node address. The masking performed is done in order to be able to write or read a single individual bit from a single register, even though the IPbus protocol is word-oriented.



```
<?xml version="1.0" encoding="ISO-8859-1"?>

<node>
  <node id="myreg" address="0x0" description="ctrl/stat register" fwinfo="endpoint;width=1">
     <node id="myctrl" address="0x0">
          <node id="write_bit_user" mask="0x1"/>
          <node id="rst_tx_datapath_pll" mask="0x2"/>
          <node id="rst_rx_datapath" mask="0x4"/>
          <node id="usercrcerror" mask="0x8"/>
          <node id="loopback" mask="0x70"/>
     </node>
     <node id="mystat" address="0x1">
          <node id="synclock" mask="0x01"/>
     </node>
  </node>
</node>
```

Figure 4.28: IPbus Address Table File for the Link Protocol Integration Tests

Using the created connection file and address table file a C++ program or a Python script use the uHAL library's methods to read or write value to the registers specified by the nodes of the address table. A simple Python script was used in order to monitor synchronization lock of the link the protocol was implemented, the script can be seen at A.3. In addition the script first reads the current synchronization status and then performs the necessary resets while it then it displays a message to the user about the status of the link, the output of the script is shown at 4.29. In this example script not all register were used.



```
root@lpc:~# python control.py
sync is =0
sending reset rx_datapath and tx_datapath_pll
sync is =1
sync is =1
```

Figure 4.29: Output of Python control.py script inside the client's Terminal

IPbus Build tool (IPBB)

The IPBB tool is a command-line firmware management and project building software tool. The need for such a tool arises when firmware projects for large FPGAs nowadays consist of hundreds of source code files. Components from those source code package files may be created and used in common by different developers. This way the IPBB tool is used in order to correctly set up and build complex firmware projects based on the source code files provided.

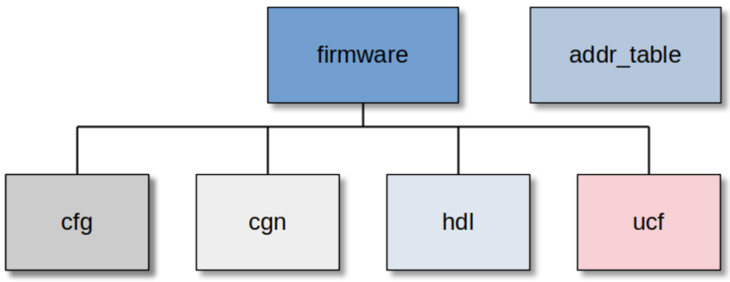


Figure 4.30: IPbus Connection File for the Link Protocol Integration Tests

In order for the IPBB tool to function correctly the user must sort all source code files in a format as the one shown at figure 4.30. The user needs to store every address table file, with the extension .XML, in the *addr_table* folder, this way all address tables may be gathered by the tool by the use of a single command instruction. The rest of the files are distributed at four different folders.

- ***cfg*** folder stores all dependency files necessary for the rest of the files in the groups.
- ***cgn*** folder is used to store all IP package files, ending with the .xci extension.
- ***ucf*** folder stores all constrain files with the .XDC extension, containing all useful TCL instructions for constraining the design.
- ***hdl*** folder contains all VHDL files.

The IPBB tool creates a project by calling the top dependency file. In turn the file specifies each file that is necessary for the project and calls them from the groups. Once all the correct files are included the Vivado Suite must be sourced by the user and then used by the IPBB tool in order to make the project. When the project is build the user can continue to issue IPBB commands in order to run *Synthesis*, *Implementation* and *Bitstream* Generation, or open the project with GUI using the Vivado Suite Environment. The IPBB tool also offers automated generation of the IPbus decoders file from the address tables contained in the project, through a single command (ipbb vivado gendecoders).

The files essential for the implementation of the Synchronous Link Protocol, as well as those utilized by the IPbus protocol, were categorized and placed into the specified groups. Corresponding dependency files were created to accurately reference each required file. That way the project can be successfully built using IPBB commands.

Chapter 5

BMTL1 Firmware - Readout System

5.1 EMP Framework

5.1.1 EMP Framework Structure

The Extensive Modular (data) Processor (EMP) framework is a generic Phase-2 framework developed for the Serenity Carrier Cards, as a continuation of the MP7 framework. The EMP framework contains several top-level designs for different FPGAs and boards. The purpose of the framework is to provide the user with all necessary foundational elements, in order to create an working environment for the implementation of trigger algorithms [33].

Each framework design instantiates an entity called the *emp_payload* and connects its ports to the infrastructure blocks, the control bus (IPbus), and input and output buffers (Links). The framework is configurable by the user, via the the setting of parameters the package *emp_project_decl* offers, this way the framework is able to adapt to the different algorithm designs until all criteria are met. In addition the framework offers Timing Trigger Control (TTC) signals which are necessary for the entire system's operation. THe main blocks of the EMP framework are shown in figure 5.1 and are as follows:

- **TTC**

  The TTC block is responsible for the distribution of the TTC signals. The signals mentioned are the Level-1 Accept signal, the Bunch Crossing 0 (BC0) signal, the LHC 40.078 MHz Clock and the TTC control signals received by

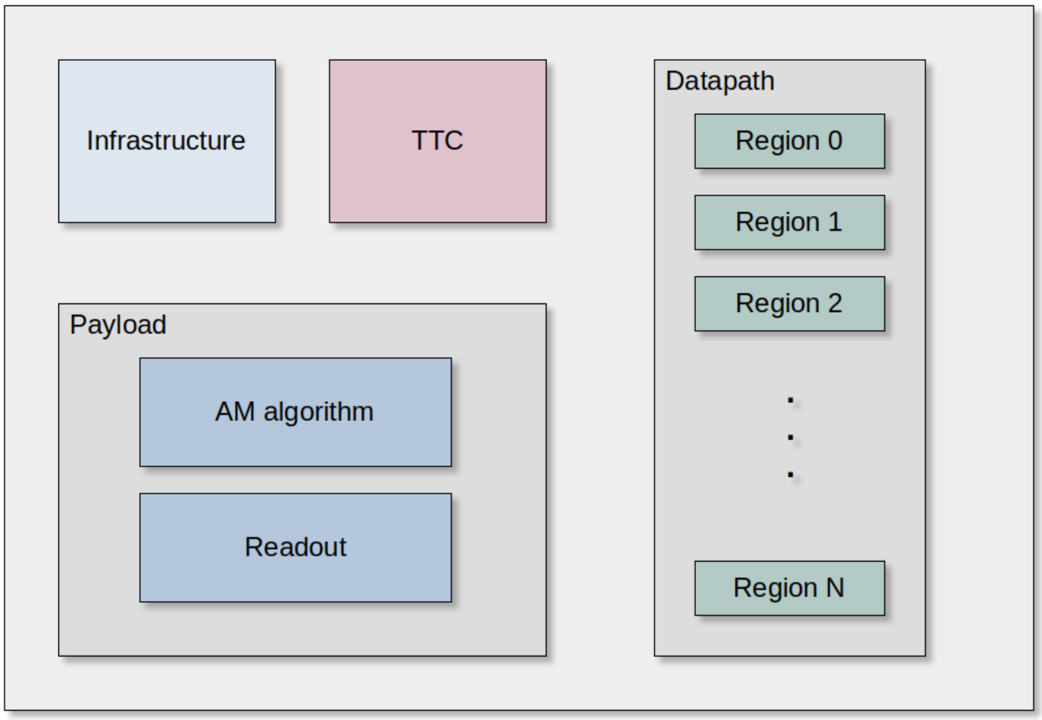


Figure 5.1: EMP Framework Logic Modules

the board. The payload clock is also created with the help of an Advanced Mixed Mode Clock Manager (MMCME4_ADV) primitive, with the LHC clock as the input source clock.

- **Infrastructure**

  The infrastructure block of the framework is responsible for creating all clocks necessary for the IPbus protocol and the TTC blocks. The block contains all logic required for the IPbus implementation. While the communication of the IPbus was performed via Ethernet cable before, on the new ATCA boards for Phase-2 the System On Chip (SOC) included is able to communicate with the Field Programmable Gate Array (FPGA) via PCB trace routes.

- **Datapath**

  The Datapath block contains all link protocols the user can implement on the board. The datapath module interfaces with the Payload module in order to establish bidirectional transmission of data to and from the Multi-Gigabit Transceivers. Inside the module the nessesary TTC signals are distributed along with the IPbus signals and the correct reference clock signals for the MGT PLLs. Depending on the number of Quads on the board, the Datapath module creates the *Region* blocks, where a Region represents a quad instance.

A Region as shown in figure 5.2 contains the following blocks. On every instance of a Region block the TX and RX EMP buffers are present. The buffers can be controlled by the Empbutler tool commands, and have a normal length of 1024 frames and 64 bit width. The Region block is then responsible for creating either one of the *emp_be_mgt* or *emp_fe_mgt* modules, this way differentiating the Front End link protocols by the Back end link protocols. The GBT and lpGBT constitute the front end protocols and the CSP the back end protocol for Phase-2.

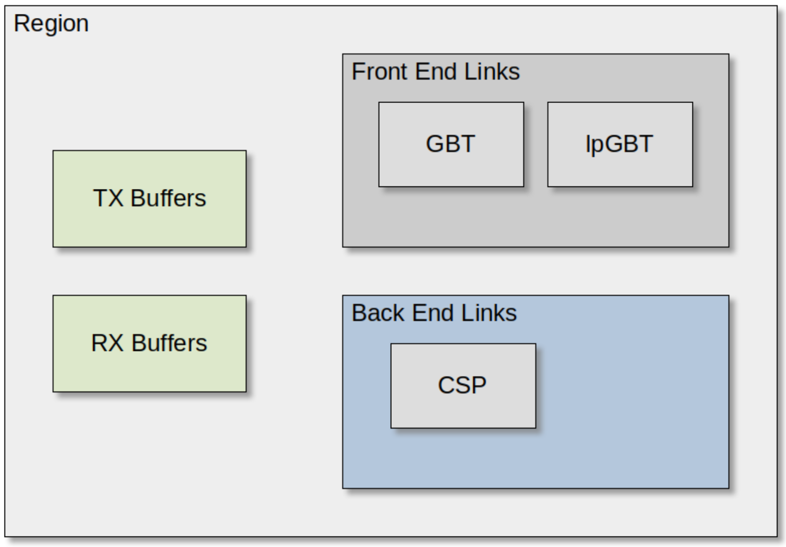


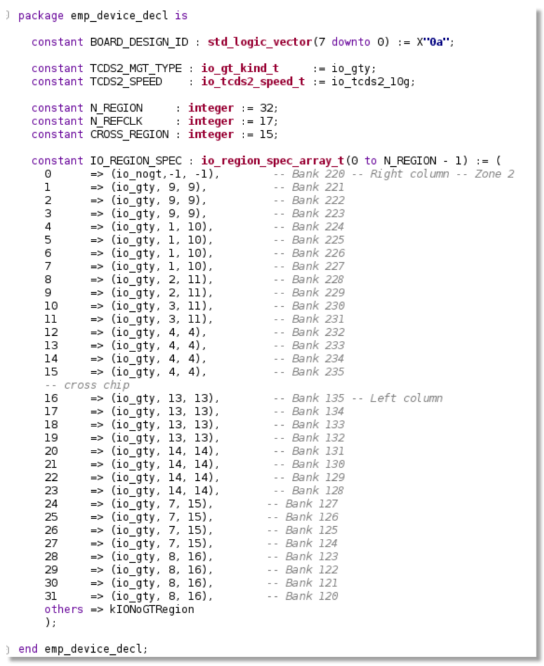
Figure 5.2: Region Block with EMP Buffers and available Link protocols

The bus used to transport the data to and from the payload follows a specific format witch is declared as a record in the framework's package files. The record specifies the following *ldata* type of bus while the *lword* specifies the word format. The *lword* type's fields are as follows:

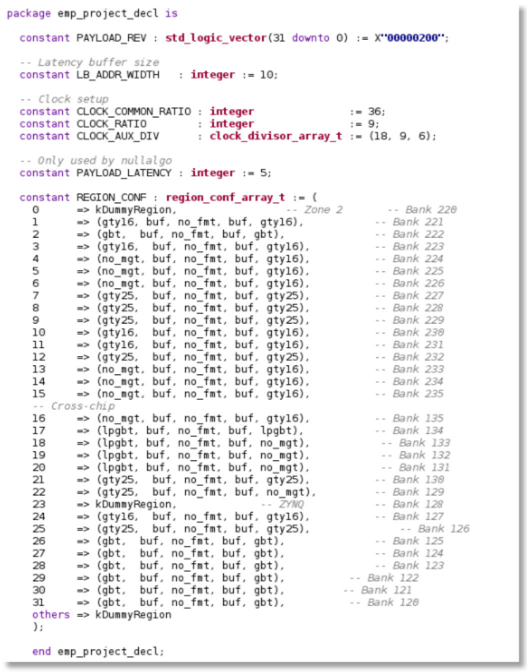
- **Valid** This bit signifies that the data contained within the word is user data and not idle filler words.
- **start** This bit is asserted "High" on the first clock of each packet, containing many words.

- **last** This bit asserts "High" on the last clock of each packet.
- **Strobe** This bit indicates at what phase the data are clocked relative to the LHC clock.
- **data** The data field contains the 64-bit word of the payload data received or transmitted.

The Framework can be configured by the user, by modifying the packages *emp_proj_decl* and *emp_device_decl*, found on the similarly named files.



```vhdl
package emp_device_decl is

  constant BOARD_DESIGN_ID : std_logic_vector(7 downto 0) := X"0a";

  constant TCDS2_MGT_TYPE : io_gt_kind_t      := io_gty;
  constant TCDS2_SPEED    : io_tcds2_speed_t := io_tcds2_10g;

  constant N_REGION     : integer := 32;
  constant N_REFCLK     : integer := 17;
  constant CROSS_REGION : integer := 15;

  constant IO_REGION_SPEC : io_region_spec_array_t(0 to N_REGION - 1) := (
    0      => (io_nogt,-1, -1),          -- Bank 220 -- Right column -- Zone 2
    1      => (io_gty, 9, 9),            -- Bank 221
    2      => (io_gty, 9, 9),            -- Bank 222
    3      => (io_gty, 9, 9),            -- Bank 223
    4      => (io_gty, 1, 10),           -- Bank 224
    5      => (io_gty, 1, 10),           -- Bank 225
    6      => (io_gty, 1, 10),           -- Bank 226
    7      => (io_gty, 1, 10),           -- Bank 227
    8      => (io_gty, 2, 11),           -- Bank 228
    9      => (io_gty, 2, 11),           -- Bank 229
    10     => (io_gty, 3, 11),           -- Bank 230
    11     => (io_gty, 3, 11),           -- Bank 231
    12     => (io_gty, 4, 4),            -- Bank 232
    13     => (io_gty, 4, 4),            -- Bank 233
    14     => (io_gty, 4, 4),            -- Bank 234
    15     => (io_gty, 4, 4),            -- Bank 235
    -- cross chip
    16     => (io_gty, 13, 13),          -- Bank 135 -- Left column
    17     => (io_gty, 13, 13),          -- Bank 134
    18     => (io_gty, 13, 13),          -- Bank 133
    19     => (io_gty, 13, 13),          -- Bank 132
    20     => (io_gty, 14, 14),          -- Bank 131
    21     => (io_gty, 14, 14),          -- Bank 130
    22     => (io_gty, 14, 14),          -- Bank 129
    23     => (io_gty, 14, 14),          -- Bank 128
    24     => (io_gty, 7, 15),          -- Bank 127
    25     => (io_gty, 7, 15),          -- Bank 126
    26     => (io_gty, 7, 15),          -- Bank 125
    27     => (io_gty, 7, 15),          -- Bank 124
    28     => (io_gty, 8, 16),          -- Bank 123
    29     => (io_gty, 8, 16),          -- Bank 122
    30     => (io_gty, 8, 16),          -- Bank 121
    31     => (io_gty, 8, 16),          -- Bank 120
    others => kIONoGTRegion
    );

end emp_device_decl;
```



```vhdl
package emp_project_decl is

  constant PAYLOAD_REV : std_logic_vector(31 downto 0) := X"00000200";

  -- Latency buffer size
  constant LB_ADDR_WIDTH   : integer := 10;

  -- Clock setup
  constant CLOCK_COMMON_RATIO : integer                 := 36;
  constant CLOCK_RATIO        : integer                 := 9;
  constant CLOCK_AUX_DIV      : clock_divisor_array_t := (18, 9, 6);

  -- Only used by nullalgo
  constant PAYLOAD_LATENCY : integer := 5;

  constant REGION_CONF : region_conf_array_t := (
    0      => kDummyRegion,                    -- Zone 2        -- Bank 220
    1      => (gty16, buf, no_fmt, buf, gty16),            -- Bank 221
    2      => (gbt,  buf, no_fmt, buf, gbt),               -- Bank 222
    3      => (gty16,  buf, no_fmt, buf, gty16),           -- Bank 223
    4      => (no_mgt, buf, no_fmt, buf, gty16),           -- Bank 224
    5      => (no_mgt, buf, no_fmt, buf, gty16),           -- Bank 225
    6      => (no_mgt, buf, no_fmt, buf, gty16),           -- Bank 226
    7      => (gty25,  buf, no_fmt, buf, gty25),           -- Bank 227
    8      => (gty25,  buf, no_fmt, buf, gty25),           -- Bank 228
    9      => (gty25,  buf, no_fmt, buf, gty25),           -- Bank 229
    10     => (gty16,  buf, no_fmt, buf, gty16),           -- Bank 230
    11     => (gty16,  buf, no_fmt, buf, gty16),           -- Bank 231
    12     => (gty25,  buf, no_fmt, buf, gty25),           -- Bank 232
    13     => (no_mgt, buf, no_fmt, buf, gty16),           -- Bank 233
    14     => (no_mgt, buf, no_fmt, buf, gty16),           -- Bank 234
    15     => (no_mgt, buf, no_fmt, buf, gty16),           -- Bank 235
    -- Cross-chip
    16     => (no_mgt, buf, no_fmt, buf, gty16),           -- Bank 135
    17     => (lpgbt, buf, no_fmt, buf, lpgbt),            -- Bank 134
    18     => (lpgbt, buf, no_fmt, buf, no_mgt),            -- Bank 133
    19     => (lpgbt, buf, no_fmt, buf, no_mgt),            -- Bank 132
    20     => (lpgbt, buf, no_fmt, buf, no_mgt),            -- Bank 131
    21     => (gty25,  buf, no_fmt, buf, gty25),           -- Bank 130
    22     => (gty25,  buf, no_fmt, buf, no_mgt),          -- Bank 129
    23     => kDummyRegion,               -- ZYNQ          -- Bank 128
    24     => (gty16,  buf, no_fmt, buf, gty16),           -- Bank 127
    25     => (gty25,  buf, no_fmt, buf, gty25),               -- Bank 126
    26     => (gbt,  buf, no_fmt, buf, gbt),               -- Bank 125
    27     => (gbt,  buf, no_fmt, buf, gbt),               -- Bank 124
    28     => (gbt,  buf, no_fmt, buf, gbt),               -- Bank 123
    29     => (gbt,  buf, no_fmt, buf, gbt),           -- Bank 122
    30     => (gbt,  buf, no_fmt, buf, gbt),          -- Bank 121
    31     => (gbt,  buf, no_fmt, buf, gbt),           -- Bank 120
    others => kDummyRegion
    );

  end emp_project_decl;
```

(a) emp_device_decl package (b) emp_proj_decl package

Figure 5.3: The two package files used by the user to configure the Framework

The *emp_device_decl* package can be configured to specify the number of regions and the number of reference clocks provided. Furthermore the type of the MGT can be specified for each transceiver along with the reference clock provided for each Region (Quad). The *emp_proj_decl* package is responsible for setting the Payload's clock ratio to the LHC clock, by setting the *CLOCK_RATIO* constant among other parameters, while also configuring each Region. The user has the option to specify what components are instantiated in each Region. By the *mgt_kind* type the user can select the protocol used for the Receiver and the Transmitter. The most common options are *no_mgt* in order to disable the MGT on that specific region, *gty16* or *gty25* for the CSP back end links, *gbt* or *lpGBT* for the front end links. The EMP buffers can also be disabled or enabled by setting *no_buf* or *buf*, respectively.

5.1.2 Empbutler tool

The Empbutler tool is the main command-line software tool used for controlling and monitoring the EMP framework's TTC control signals, RX/TX buffers and link firmware. The Empbutler can be used to display the board's information and issue the reset commands to bring-up or correctly reset the system. The tool can also be used to control the board's MGTs by addressing each channel and setting them to loopback mode, PRBS mode, DFE LPM mode or completely power them off, while also used to display the MGT's link status for alignment and errors. The Empbutler tool can configure the EMP RX/TX buffers found in every Region instance in the datapath module. The buffers can be configured to load a generated counter, a generated random pattern or data from a file, given that the user has followed the specified EMP buffer data file format. The buffers can also be configured to capture mode and store their contents to output files, displayed in the aforementioned format.

5.2 Front-End Link Protocols

Data from events at the CMS detector at the Underground eXperimental Cavern (UXC), are digitized by the on-detector electronics and need to be transmitted to the Underground Service Cavern (USC) in order to be processed. The data are transferred via fiber-optic cables to the USC. Because of the increased radiation in the UXC the on-detector electronics use radiation tolerant ASICs that implement the protocols used. The protocols themselves need to be able to correct errors that might occur.

5.2.1 GBT

The GigaBit Transceiver (GBT) protocol is the protocol used for transmission of data from the en-detector electronics to the boards at USC, during Phase-1. The protocol is synchronous to the LHC 40.078 MHz clock and operates at line rates of 4.8 Gb/s. In order to combat bit errors that might occur the protocol uses the Reed-Solomon Forward Error Correction (FEC) code. The GBT frame consists of 120-bits in total as shown in figure 5.4. The first LSB 32 bits are used for the

Reed Solomon codes in order for the receiver to be able to correct corrupted data. The 4 MSB bit are used for the Header of the packet and they are responsible for alignment and lock of the link after the receiver correctly reads a number of consecutive frames that contain a valid header. The 80 bits shown at figure 5.4 contain the payload of the frame and transfer the user data. This field is not pre-assigned and can be used for Data Acquisition (DAQ), Timing Trigger Control (TTC) or other Experiment Control (EC) applications. The remaining 4 bits are used for slow control, with two of them reserved for GBT control and the other two user reserved. In the GBT-FPGA implementation the 4 slow control bits can also be used in addition to the payload, that way the estimated user bandwidth is 3.36 Gb/s [34].

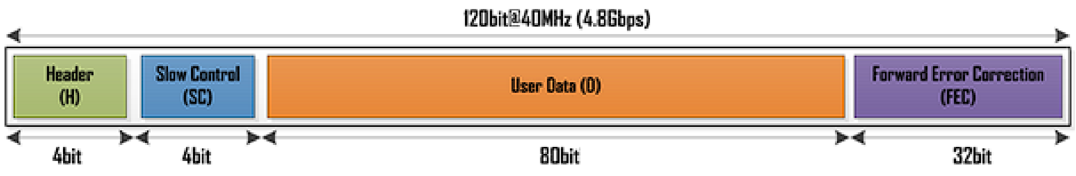


Figure 5.4: GBT frame format

5.2.2 lpGBT

The lpGBT protocol is developed as the natural evolution of the GBT protocol. While the protocol is synchronous to the LHC clock, it is asymmetric to its uplink and downlink path rine rates. The Uplink path being the Front End to Back End path, while the Downlink path being the Back End to Front End path. The protocol also uses Reed-Solomon FEC codes in order to ensure link robustness, while the user may chose between two provided schemes FEC5 or FEC12.

The encoding scheme used for the downlink path is set to FEC12, while the downlink line rate is set to 2.56Gb/s. The downlink frame size is 64 bits including all frame fields, header 4 bit, user data 32 bit, External Control (EC) 2 bits, Internal Control (IC) 2 bits, FEC 24 bits, this way the user total bandwidth provided is 1.28Gb/s.

The uplink datapath line rate can be configured to 10.24Gb/s or 5.12Gb/s, with 256 bit and 128 bit frame sizes respectively. The user is also able to chose between the two encoding FEC schemes, FEC12 or FEC5, depending on the application of the links. This way the uplink datapath can be configure in four different types of operation, as shown with the correct fields in table 5.1 [35].

	5.12G/FEC5	**5.12G/FEC12**	**10.24G/FEC5**	**10.24G/FEC12**
Header	2 bits	2 bits	2 bits	2 bits
IC	2 bits	2 bits	2 bits	2 bits
EC	2 bits	2 bits	2 bits	2 bits
FEC	10 bits	24 bits	20 bits	48 bits
User Data	112 bits	98 bits	230 bits	202 bits

Table 5.1: lpGBT Uplink Frame formats with their Fields

5.3 Back End Link Protocols

5.3.1 CSP

The CMS Standard Protocol (CSP) is used to transmit LHC-synchronous data over asynchronous links between backend boards [33]. The protocol is based on the 64b/67b encoding. The data generated and supplied to the protocol is synchronous to the LHC clock but the reference clock of the links is not synchronous. That means that the reference clock of the links is higher than the clock used to generate the data. The link protocol is able to transmit data at 16Gb/s and 25Gb/s depending on the board specifications. For the 25Gb/s mode the payload frequency used is 360MHz, thus a total of nine 64-bit words are able to be transferred per one LHC clock, while for the 16Gb/s the payload frequency is set to 240MHz and a total of six 64-bit words are able to be transferred per one LHC clock. The protocol consists of two types of words, filler (control) words and data words carrying data to and from the payload. The protocol also contains a CRC mechanism in order to indicate bitflip errors.

5.4 Hardware

In this section the boards used for the slice test which this thesis focuses on, will be briefly discussed along with their primary functions.

5.4.1 OBDT

The On detector Board for Drift Tubes (OBDT) are responsible for the time digitization of the DT signals. On the longitudinal view the Barrel muon detector is divided in five wheels -2, -1, 0, +1, +2. On the transverse view every wheel is divided into 12 Sector. Each Sector consists of 4 DT Chambers named MB1, MB2, MB3 and MB4. Each DT Chamber consists of three SuperLayers (SLs), two of them measuring position and bending angle in the transverse view $(r - \phi)$ and one SL measuring position in the longitudinal view $(r - z)$, while the outer Chamber MB4 is equipped with only two $(r - \phi)$ SLs. A SL consists of DT cells that are grouped in four half-staggered layers [15]. One OBDT board is connected per SL, receiving inputs from 240 DT channels. The number of OBDTs per Sector is shown in table 5.2. The variations on the total number per Sector is due to difference in Chamber sizes.

Sector	Number Of OBDTs
1, 2, 3, 5, 6, 7, 8, 10, 12	14
4	16
9, 11	12

Table 5.2: The number of OBDT boards per Sector

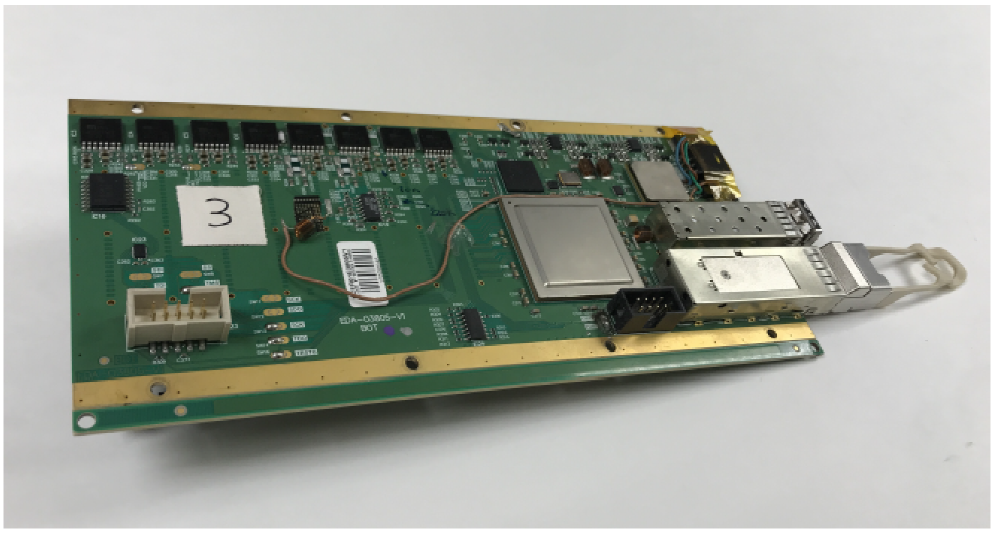


Figure 5.5: The OBDT board

The OBDT generates TDC Hits. The format of a TDC hit is shown in figure 5.6. The frame consists of the channel number of the DT cell, the bunch crossing number and the TDC value.

There are two available versions of the board each using a different ASIC for the link protocol implemented, one version for the GBT and one for the lpGBT.

24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OBDT Channel Number								Bunch Crossing												TDC Value				

Figure 5.6: TDC Hit frame Format

A picture of the OBDT board is shown at figure 5.5.

5.4.2 BMTL1

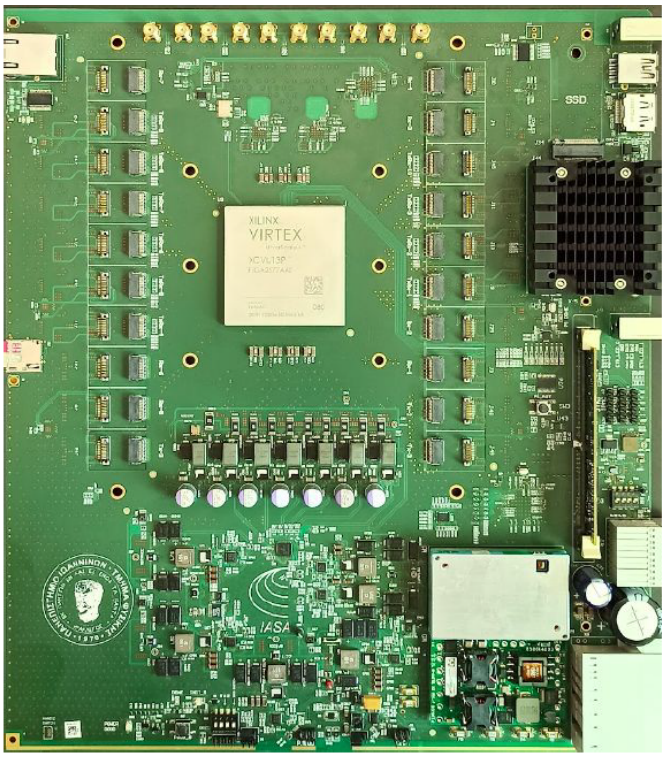


Figure 5.7: The BMTL1 board

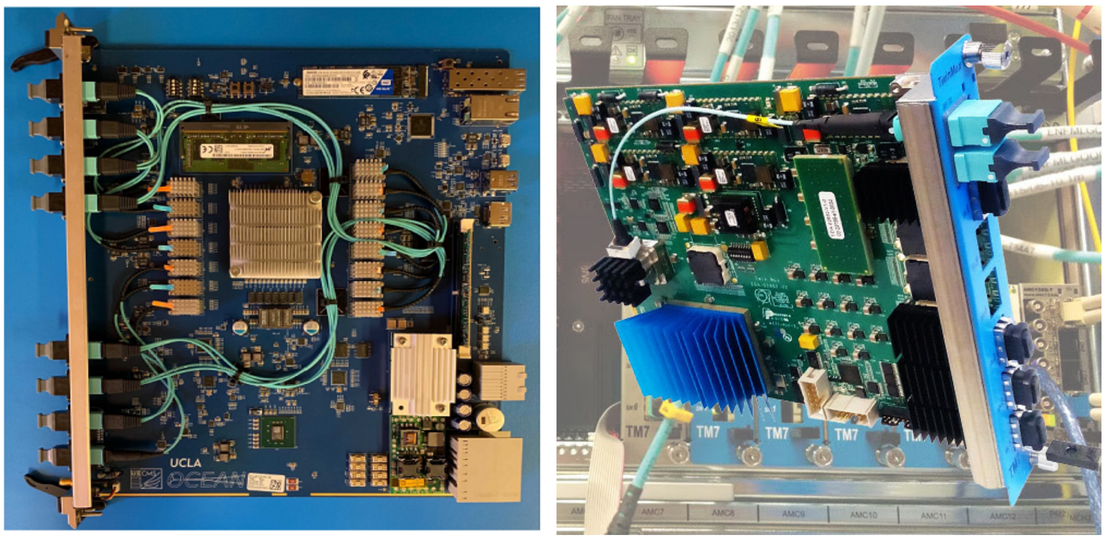
The BMTL1 ATCA board is depicted at figure 5.7. The board receives Drift Tube (DT) Hit data from On detector Board for Drift Tubes (OBDT) boards and Resistive Plate Chamber (RPC) data from Link Boards. The board processes data received by using the AM algorithm, for a total of two Sectors. The AM algorithm running on the firmware of the board produces muon track segments, refer to as Trigger Primitive Generatorss (TPGs) or stubs, from the TDC Hits transmitted by the OBDT board. In each BX the BMTL1 system can create a maximum of 28 stubs per sector, up to 16 stubs in the transverse view - four ϕ stubs per chamber and up to 12 stubs in the longitudinal view four θ stubs for MB1, MB2, MB3

[15]. RPC data can be used complementary to the DT data and produce a Super Primitive. The frame of a TPG is 64-bits wide, and it is transmitted to the Global Muon Trigger (GMT) via 25Gb/s optical links with the CSP protocol.

The BMTL1 board uses a Xilinx XCVU13P FPGA which offers four Super Logic Regions (SLRs), 1.728.000 Look Up Tables (LUTs), 3.456.000 Flip FLops, 12.288 DSP slices, 94.5 Mb Block RAM (BRAM), 360 Mb UltraRAM and 128 GTY transceivers. The optical connectivity of the board is performed via Samtec FireFlies modules, offering 40 RX and TX channels by using 10 x4 bidirectional connectors capable of running at 25Gb/s, 80 RX channels by using 7 x12 RX connectors (one connector is x8) and 36 TX channels by using 3 x12 connectors capable of running at 16Gb/s.

The board is controlled by a ZYNQ Ultrascale+ SOC. Offering an Arm Cortex A-53 Quad core, capable of running Linux. While the module is connected to the FPGA by four GTH MGTs running at 10Gb/s and 20 Low Voltage Differential Signaling (LVDS) pairs [36].

5.4.3 OCEAN - TM7



(a) (b)

Figure 5.8: The OCEAN board at 5.8a and the TM7 board at 5.8b

The Ocean ATCA board is prototype board of the X2O boards the the GMT system will use. The board receives the TPG data from the BMTL1 board using the CSP protocol at 25Gb/s and reconstructs muon candidates using the

Kalman Muon Track Finder (KMTF) algorithm. The board features a ZYNQ Ultrascale+ ZU19EG-2 SOC capable of running Linux, which controls the FPGA and 72 transceivers, 28 GTY capable of running at 28Gb/s and 44 GTH capable of running at 16 Gb/s [37]. The board is shown in figure 5.8a.

The TM7 board is a μTCA Phase-1 Board. It was designed to be used in the trigger (TwinMux) and the readout second-level electronics upgrade. The board offers a Virtex-7 FPGA and six 12-fiber MTP receivers for a maximum of 72 inputs. The board interfaces with the AMC13 board present in the μTCA crate, while the AMC13 receives timing data from the TCDS and delivers readout data to the DAQ system [38]. A picture of the board can be seen in figure 5.8b.

5.5 Layout of the USC setup

At figure 5.9 we can see the setup of the phase-2 upgraded electronics tests for the Barrel Muon Trigger at the USC along with boards from the UXC. In order to validate each board's functionality, multi-board tests must be performed. The setup layout at the writing of this thesis (January 2024) is as follows.

Thirteen OBDTs with the GBT chip are installed in Sector 12 and eight OBDTs with the lpGBT chip are installed in Sector 1, all OBDTs are located in wheel +2 of the CMS detector. DT cells transmit muon pulses to the OBDTs, which in turn convert them to TDC Hit data and transmit them to the USC setup.

At the USC setup two crates are used, one crate includes the Phase-2 ATCA boards BMTL1-OCEAN while the other crate available is a μTCA crate which includes a TM7 and AM13 board along with a Clock Expansion board. At the μTCA crate the AM13 board which is responsible for the distribution of the TCDS signals, generates a local LHC Clock and distributes it through the backplane of the crate. The Clock Expansion board was made in order to expose this clock to SMA connectors, from where the LHC clock is provided to the BMTL1 board which in turn exposes it on SMA connectors itself and the OCEAN board is able to also receive the clock. This clock distribution ensures the system operates at the same clock, using the same clock source and all boards are synchronous to each other. The TM7 board transmits to the BC0 signal via a GBT link to the BMTL1 which in turn propagates the signal to the OCEAN via a CSP link. The BC0 signal is transmitted once every 3564 clocks cycles of the LHC clock which is equivalent to one Orbit. The BC0 signal is critical to the AM and the KMTF algorithms.

The TDC Hit data arrive at the USC and are driven inside the BMTL1 board, where the firmware of the board is correctly set, so that regions with the correct

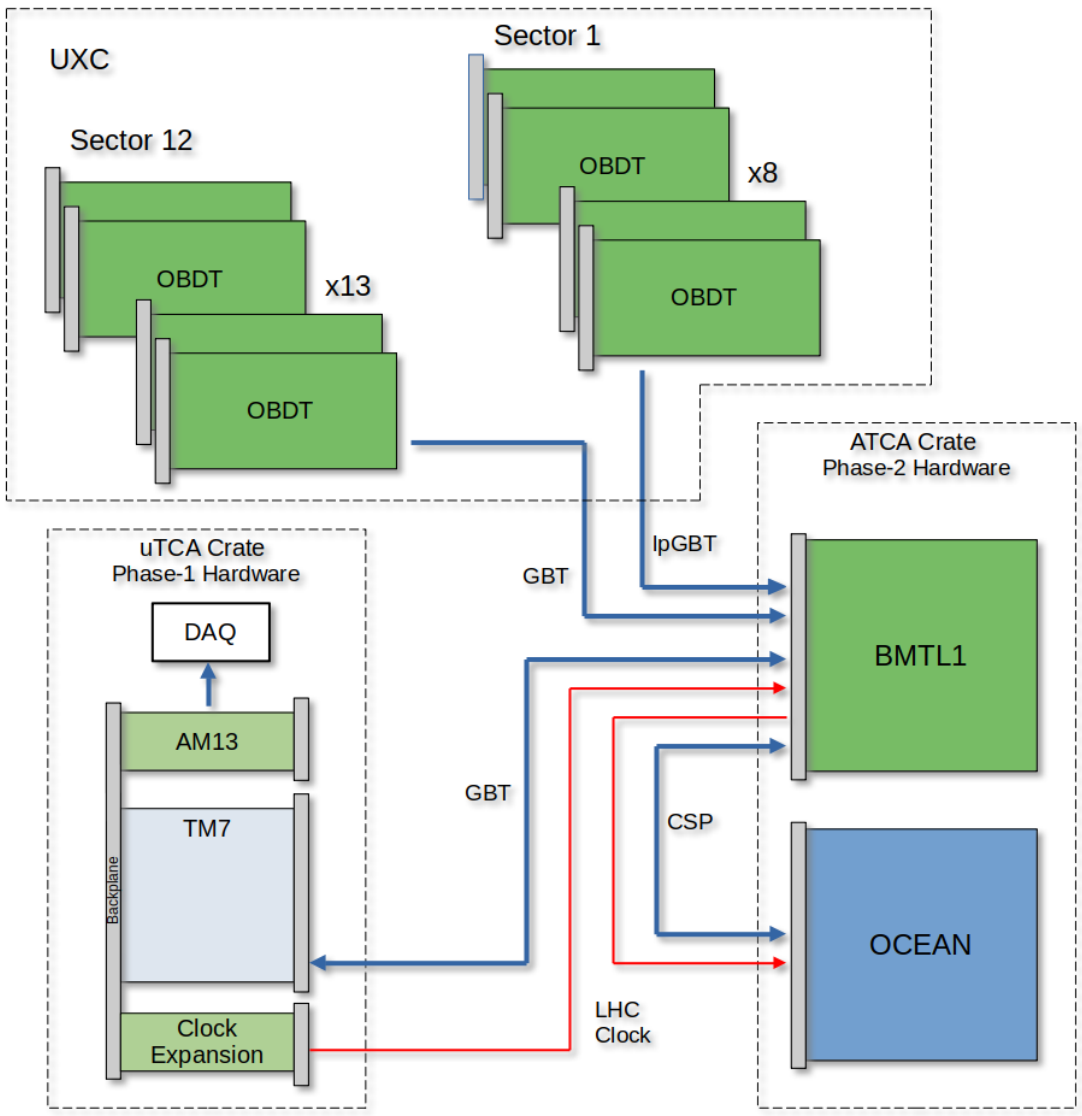


Figure 5.9: USC test setup Layout

protocol are instantiated. The BMTL1 firmware will support two Sector instantiations for the AM algorithm. At the time time of writing this thesis only one Sector of AM algorithm is implemented, rerouting the inputs from OBDTs of one Sector at a time, accordingly. The Hits arriving at the BMTL1 are processed by the AM algorithm and track segments or TPs are produced for four chambers (one Sector) only for the transverse view, ϕ TPs (stubs). The TPs are then transmitted to the OCEAN board in order for the KMTF algorithm to match and reconstruct the muon tracks and produce muon candidates.

These tests are performed in order to observe and validate the data flow (link protocols used) and the algorithms used. All firmware instances offer some form of data monitoring, either from ILAs or by IPbus scripts reading internal registers. In order to be able to gather all data of the system a Readout mechanism needs to be implemented. The proposed scheme is shown in figure 5.10.

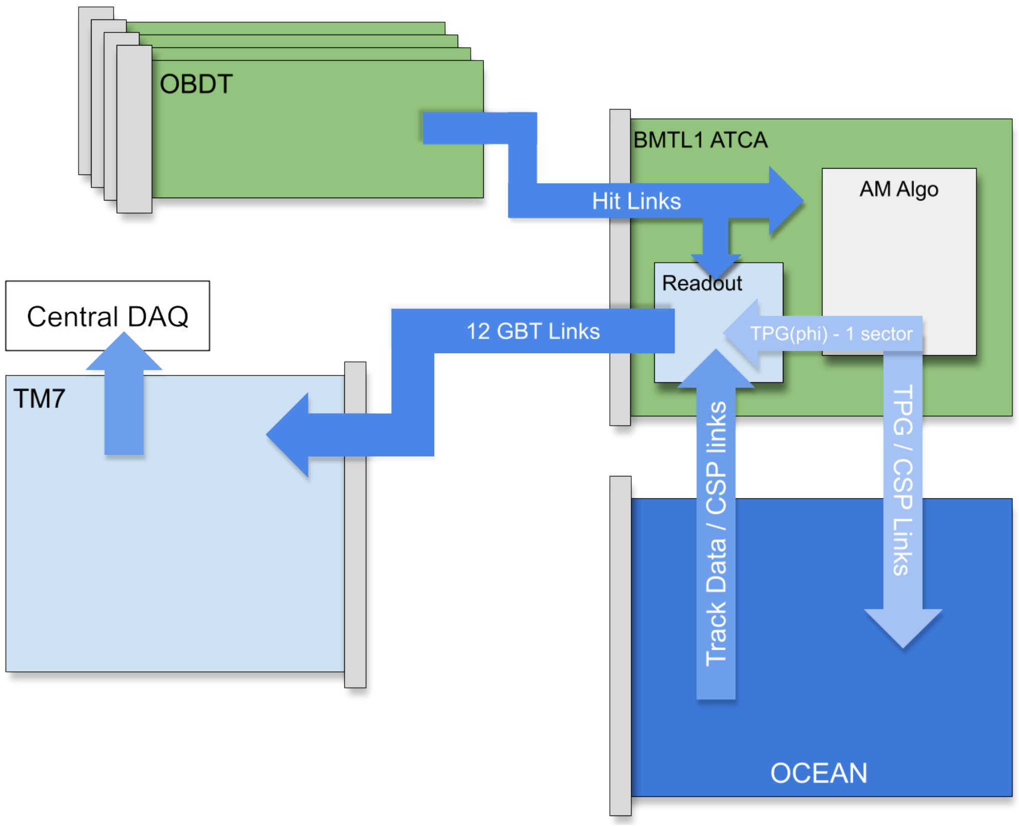


Figure 5.10: Proposed Readout Layout at USC setup

Hit data from the OBDTs, TP data from the AM algorithm and Muon Track data from the KMTF algorithm transmitted back from the OCEAN via CSP links, are collected by the Readout Module inside the BMTL1 firmware. The data are collected and then transmitted to the TM7 Phase-1 board via 12 GBT links, where the TM7 interfaces with the AMC13 board via the μTCA crate backplane and the data are delivered to the DAQ system.

5.6 Readout Firmware

5.6.1 Layout Of Readout module

The BMTL1 firmware implementation follows the EMP framework specifications, that way the payload module contains the algorithm implemented in the board and the datapath module contains all link protocol logic. The Readout module's firmware implementation in the BMTL1 firmware is shown in figure 5.11. The position of the Readout module was chosen to be inside the Payload module in order to easier interface with the AM algorithm's instance, which for the case of one sector is the *dt_sector* module, but also to not disrupt the flow of data with the EMP *ldata* bus type between the payload and the datapath modules.

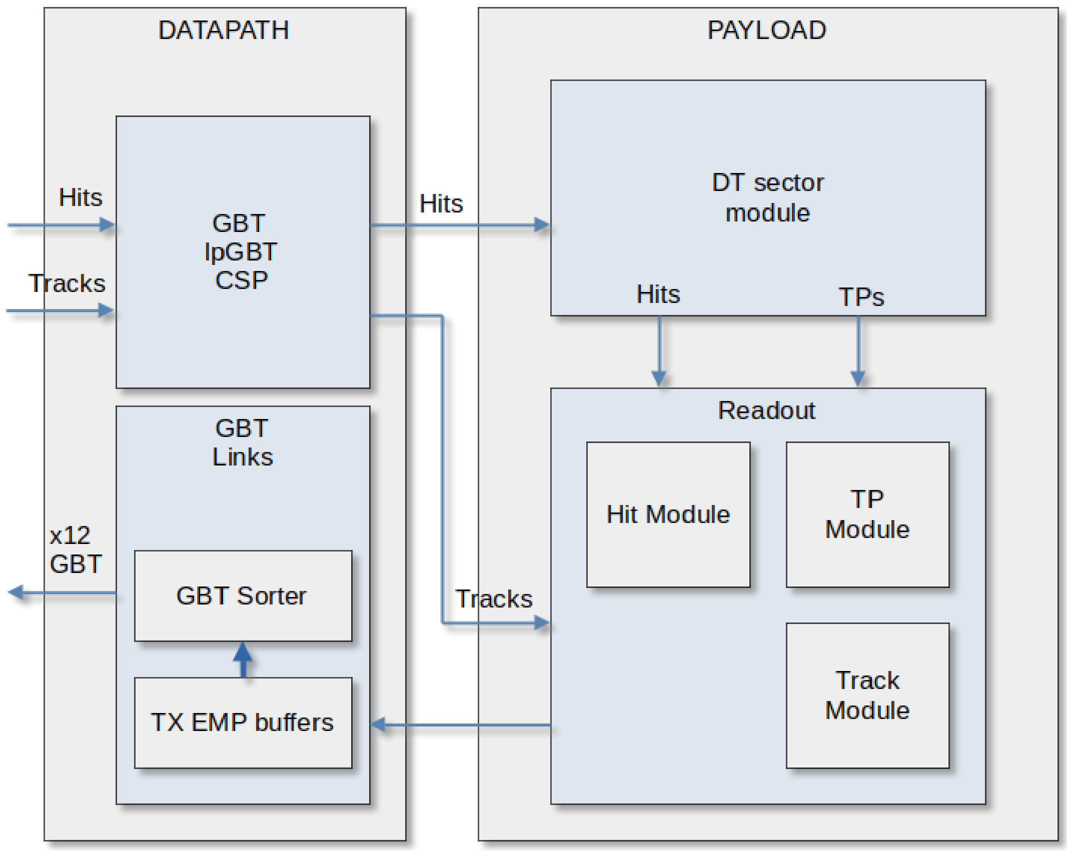


Figure 5.11: Readout Module Diagram in the BMTL1 firmware

TDC Hit data arrive at the GBT or lpGBT RX modules of the datapath block, depending from which Sector the OBDTs belong to. The Hits are then propagated to the payload block and driven inside the *dt_sector* module. The AM algorithm produces Trigger Primitivess (TPs) which are necessary for the KMTF algorithm and thus are transmitted via CSP links to the OCEAN board. The TPs produced along with the TDC Hits are driven out of the *dt_sector* and are inserted to the Readout module. Once inside the top Readout module the Hits enter the Hit module and the TPs enter the TP module. When the KMTF algorithm has reconstructed muon tracks by the TPs that were transmitted, the track data are sent to the BMTL1 and inserted to the datapath RX module of the CSP protocol. The data are then transferred to the payload module and inserted directly to the Readout module where in turn are directed inside the Track module.

All three modules store the information received temporary, in buffers. The data are then driven out of the buffers and directed to the datapath. First the data enter the TX EMP buffers as specified by the EMP framework and then continue to the GBT sorter module of the GBT protocol.

The GBT sorter module exists because the GBT protocol transmits/receives its words on the LHC clock, whereas the rest of the system uses the 360 MHz clock *clkp* with a clock ratio of 9 to the LHC clock. This way the data need to cross from the *clkp* domain to the LHC clock domain. This is easily performed, for the reason that the two clocks are synchronous, given that the *clkp* clock is generated as a multiple of the LHC clock, thus having the same source. The GBT TX sorter receives the data from the EMP buffers and stores them for 9 *clkp* clock cycles or one LHC clock cycle, in local registers and then proceeds to present them at the appropriate GBT protocol's TX ports, in order for the data to be successfully sent.

Inside the sorter module there has been created a multiplexer controlled by an IPbus slave. The multiplexer can be configured to export the data from the EMP buffers, this being the default setting of the multiplexer, or to output a locally generated static word with a counter pattern at its LSB bits. The format of the word is specified as *0xCABABABABABABABAB* & counter[23:0], where the counter is incremented with the LHC clock. This pattern is offered for evaluating the link's normal operation, and the activation of it depends on scripts executed to communicate with the IPbus slave.

Each module used in the Readout module will be explained further in the next sections.

5.6.2 Readout Hit module

In figure 5.12 the data flow of the TDC Hits is shown. The *dt_sector* module drives out an array of vectors of 32 bits, where each element of the array corre-

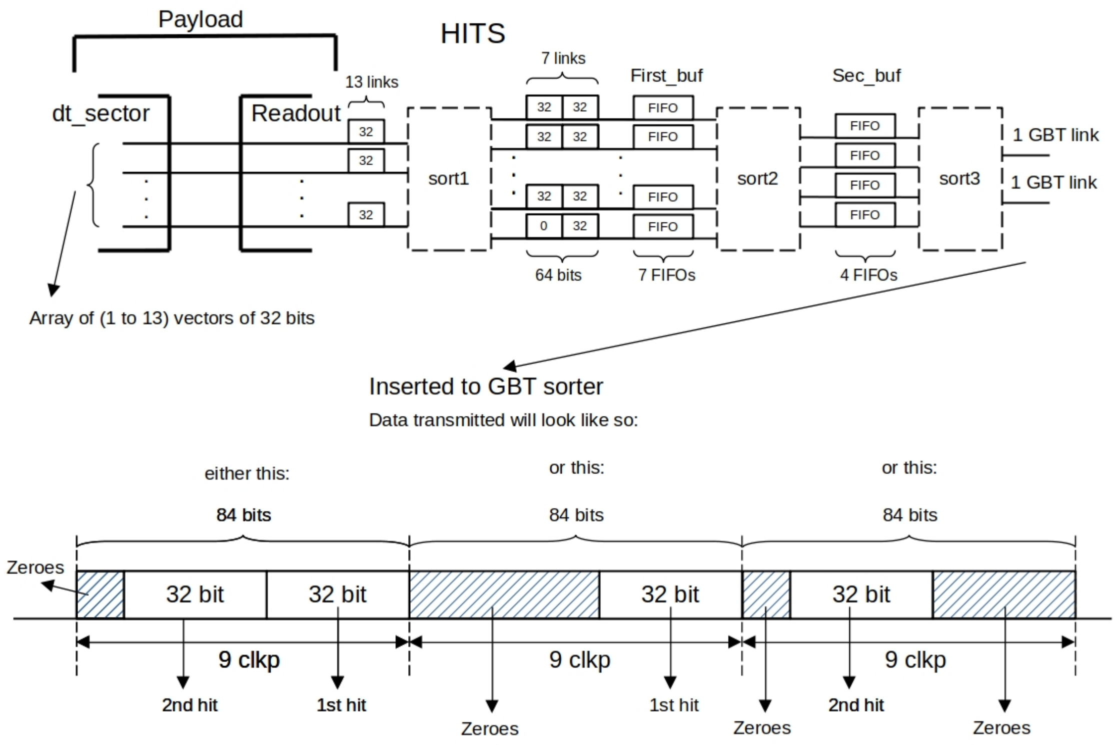


Figure 5.12: Readout Hit module logic Diagram

sponds to the Hits originating from the OBDTs. The data are driven out by the *clkp* clock and are inserted into the top Readout module and sorted to the Hit module of the Readout.

The data pass through their first sorting process, during which two 32 bit data words are combined in order to form a 64 bit word. One word populates the 32 MSB bits while the other word the 32 LSB bits of the 64 bit word. The data form a seven 64-bit array with the 7^{th} element consisting of zeroes and a 32-bit Hit word. Each non zero word of the 64-bit array is inserted to FIFOs.

Each FIFO is using the common clock Block RAM primitive RAMB36E2, which offers a 36-bit wide by 1024-bit deep dual port RAM, configurable to a 72-bit wide by 512 deep dual port RAM [39]. The FIFOs used were configured to be the latter, but using only the 64 LSB bits. Each FIFO offers status indicator output ports, for when the FIFO is empty or full of data, each state indicated by the empty and full ports respectively.

The data pass through their second sorting process to form a four 64-bit array. During the process, the empty ports of each pair of FIFOs are monitored. The aim of the second sorting phase is to descrease the seven 64-bit input array to a four 64-bit output array. Two FIFOs try to merge their outputs to one common lane, while the 7^{th} FIFO has no contest and assumes full control of the lane. The other FIFOs' empty ports are monitored, and depending on which state they are found, the sorting mechanism assumes control of the common lane, accordingly. If one of the two FIFOs is empty while the other one is not, the latter (not-empty) assumes control of the common lane and drives its data out. If both FIFOs are empty, then the lane displays zeroes. If both FIFOs are not empty the control of the lane alternates between the two FIFOs' outputs. One of the FIFOs holds its data while the other drives its data out and the process is reversed on the next clock cycle. This sorting is needed in order to maintain a balance between the two FIFOs, otherwise one of them would be prone to fully filling its available cells and in turn lose data.

Each of the four common lanes comprising a four 64-bit array are fed into 4 separate FIFOs. The data pass through their third and last sorting stage from where the same sorting mechanism is implemented as the one described previously. In summary the four FIFO outputs are sorted to two 64-bit lanes. Each of the two lanes' data are transferred to the datapath module in order to be transferred by a GBT link to the TM7 board, each lane corresponding to one GBT link.

Data are inserted to the GBT sorter as described previously. When a 64 bit contains two Hit words then the GBT sorter will transmit a 84 bit word with the 64 bits of data populating the frames' LSBs positions while the 20 MSB bits of the GBT frame are filled with zeroes. If the 64-bit Hit word contains one 32-bit Hit word at its LSB position, the GBT sorter transmits a 84 bit word with the 52 MSB bits filled with zeroes followed by the 32 bit Hit word. If the 64-bit Hit word contains a 32-bit Hit word at its MSB position then the GBT sorter will transmit a 84 bit word with the 20 MSB bits as zeroes, followed by the 32 bit word with the LSB 32-bits of the 84 bit frame also filled with zeroes.

It is expected that the system will not transmit continuously Hit data but in reality a more sporadic stream of Hit data. The number of the GBT links provided for the transmission of the data to the TM7 board was chosen with this principle in mind, where in total 12 GBT link are connected between the two boards.

5.6.3 Readout TP module

The TPs produced by the AM algorithm are driven out of the *dt_sector* module by the *clkp* clock. Each *dt_sector* instance offers a four 64-bit vector array, each corresponding to the TPs produce by one of the Chambers of the Sector.

The normal TP frame consists of 64-bits of data, but in order to leave room

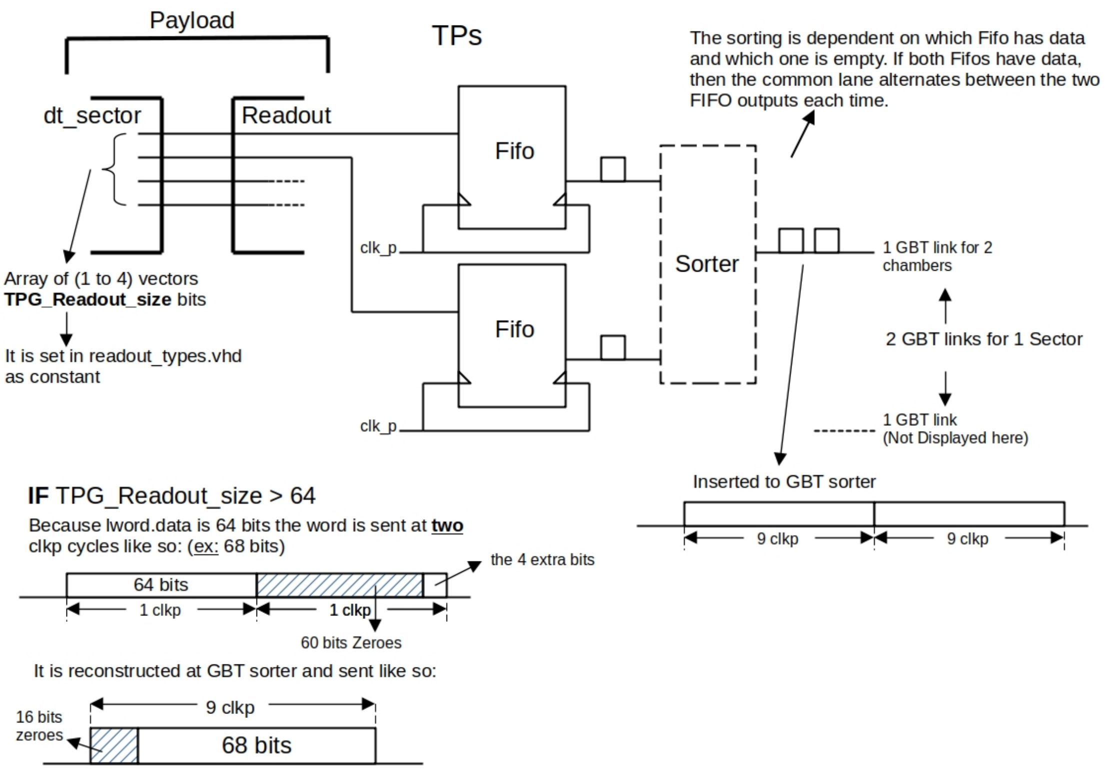

Payload
TPs
dt_sector
Readout
Fifo
Fifo
Sorter
clk_p
clk_p
The sorting is dependent on which Fifo has data and which one is empty. If both Fifos have data, then the common lane alternates between the two FIFO outputs each time.
1 GBT link for 2 chambers
2 GBT links for 1 Sector
1 GBT link (Not Displayed here)
Array of (1 to 4) vectors TPG_Readout_size bits
It is set in readout_types.vhd as constant
Inserted to GBT sorter
9 clkp
9 clkp
IF TPG_Readout_size > 64
Because lword.data is 64 bits the word is sent at two clkp cycles like so: (ex: 68 bits)
64 bits
1 clkp
1 clkp
the 4 extra bits
60 bits Zeroes
It is reconstructed at GBT sorter and sent like so:
16 bits zeroes
9 clkp
68 bits

Figure 5.13: Readout TP module logic Diagram

for expansion as proposed by the AM algorithm developers, a Generic was set in the readout package. The *tpg_readout_size* Generic specifies the bit width of the TP frame by the developer, with no additional configuration, the only limit being the GBT frame width with 84-bits.

In figure 5.13 the process of the TP module is depicted. The TPs are inserted as an array of four vectors. The data are then stored in four FIFOs, using the same primitives with the Hits module. If data are present the word enters the FIFO, whereas zeroes (no data) get discarded. Each pair of the four FIFOs go through the same sorting scheme that was earlier described. This way the output array is decreased to a two 64-bit vector array. In figure 5.13 the process is depicted for a pair of FIFOs and not for all four FIFOs, while the other two not shown follow the same procedure.

Each Readout module's TP output is driven out and transferred to the data-path module, where one GBT link is dedicated for each output. Before data are transmitted, the GBT sorter attaches an additional 20 bits of zeroes at the MSB position of the GBT frame, given that the default 64 bit width is selected. In case the *tpg_readout_size* is set to a higher value than 64, the TP bit width surpasses the *lword*'s bit width. While the *ldata* type bus is the only bus used to transfer payload data to and from the datapath, the TP frame is broken in two parts in order to be transferred. The first part consists of the 64 LSBs fo the TP frame, while the second part consists of the rest of the bits that remained, placed as the LSBs of the *lword*'s 64 bit frame. The TP frame is reconstructed at the GBT sorter. The GBT sorter stores the first part of the frame and wait for the second part at the next *clkp* clock cycle. When all the parts are available the frame with the correct bit width is reconstructed, while the rest of the 84 bits required for the GBT frame are filled with zeroes. In figure 5.13 an example of *tpg_readout_size* set to 68 is depicted.

5.6.4 Readout Track module

Track data are provided to the Readout module via the *ldata* bus from the datapath module. The track data are trasmitted from the OCEAN board via four CSP links at 25Gb/s. Each link contains the exact same information as the other three. In each LHC clock cycle a total of nine 64-bit words can be transferred from one CSP link. The first eight words transmitted from the OCEAN to the BMTL1 in one LHC clock cycle may contain track data, while the 9^{th} word is zeroes.

In figure 5.14 the logic diagram of the Readout track module is depicted. The data are inserted to the Readout module by setting the correct channel number of the *ldata* channel number. The *muon_link* generic is also set at the readout package. Once track data are inside the module, they are stored in a single FIFO. The data are then driven out of the FIFO continuously until the FIFO is empty. The common output lane of the FIFO is split in two lanes. The first lane receives one track word while the other lane receives the following track data word at the *clkp* clock cycle. Essentially this mechanism is a time demultiplexing mechanism, where data are split in two lanes per clock cycle. Before exporting data to one lane again the FIFO must wait another 8 *clkp* clock pulses. This is essential in order to not lose data as the GBT protocol transmits a 84-bit word per 9 *clkp* clock pulses or 1 LHC clock pulse.

The data arrive at the GBT sorter where an additional 20 bit filled with zeroes are attached to the front of the track frame, this way creating the GBT 84 bit frame that is transmitted. Two GBT links are used, one for each lane.

The total amount of BRAM used for the Readout implementation is 16 RAMB-36E2 primitives, each primitive offering 36Kbit BRAM, whereas the system in total offers 94.5 Mbits of BRAM. Each Readout module for one sector uses in total 6 GBT links, whereas the final version for the two sector implementation will use all

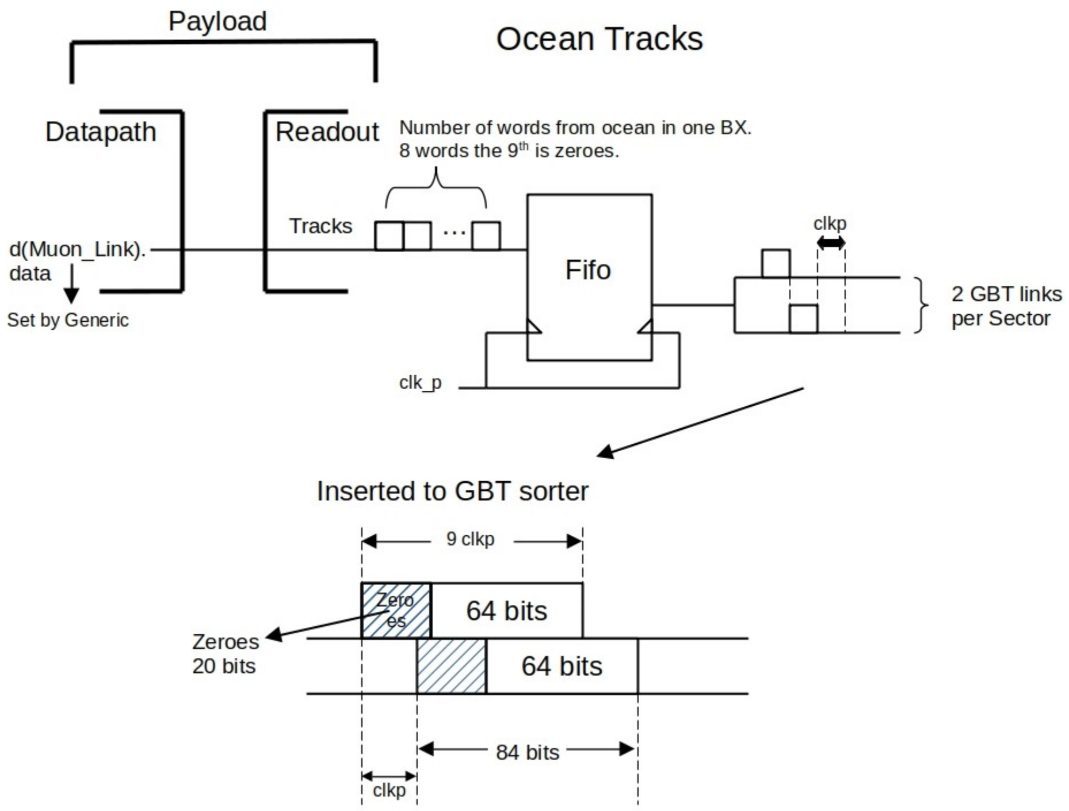


Figure 5.14: Readout Track module logic Diagram

12 available GBT links transmitting to the TM7.

5.7 USC Slice test

The Readout firmware was tested with various methods. The first method being the simulation of the design on software. Once simulation tests were successful then ILAs were added to the design in order to test the implementation on hardware. The DT group had available a test setup at the SX5 surface building at Point 5. The setup included the same boards as the USC setup, with the main difference being only one Chamber present and no TM7 with GBT links implemented. Tests pulses from the OBDT were used, as well as cosmic muons in order to receive Hit data to the BMTL1 board.

At the USC setup a TM7 board was configured to receive 12 GBT links, as stated previously. A portion of the data received by the TM7 can be viewed by a script provided by the DT group. The script enables twelve 4096 deep local buffers for each GBT link to capture mode for a certain, user-defined, amount of time. Once the exposure time is over the script reads and displays the contents of each buffer to the user's terminal, while the user can also chose the amount of words printed on the terminal, in order to avoid printing all 4096 words for each channel.

Trying to validate the data transmitted from one board to the data received by the other board when using Hit data from OBDTs poses a challenge. Hit data are transmitted by the OBDTs, while the user is able to observe them at the BMTL1 board with ILA triggers or scripts. When the user tries to observe data received by the TM7, the script for the local buffers is also executed. Here is the challenge, the user cannot execute the TM7 buffer script fast enough in order to capture the same Hit word that was observed at the BMTL1 board, this is due to the fact that the data are transmitted at ns time scales. Furthermore since a Hit word is independent by the user there is no way for a script monitoring the TM7 data to determine if the data received are in fact the correct data.

A more elaborate plan was proposed in order to validate the system's functionality with the assistance of the GBT buffer script. The layout of the test is depicted in figure 5.15.

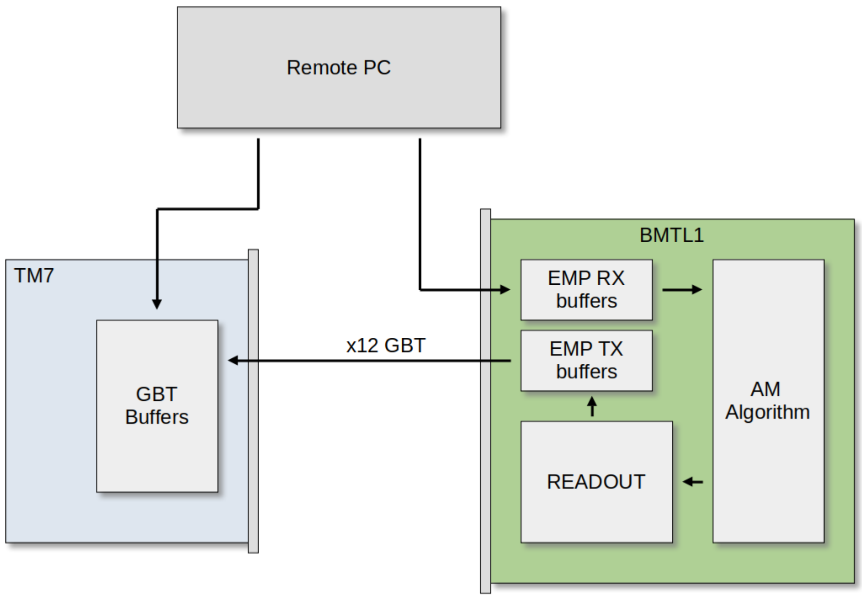


Figure 5.15: BMTL1-TM7 Readout test Layout

Hit data provided by the DT group are loaded in the RX EMP buffers at the datapath module, while the connections from the OBDTs are cut inside the firmware implementation. The EMP buffers are loaded with the user data by the Empbutler command line tool. The Empbutler can issue the *PlayOnce* command when setting the buffers, this way the buffer contents (1024 words) will be presented only once per LHC orbit, to the rest of the system [33]. If this setting is not used, the buffer contents will be read continuously during one LHC orbit. The provided Hits' data format must be modified in order to meet the specific EMP file format. This is performed by a custom C++ program that reads the contents of the original Hit data file and builds the EMP format file. The Receiver RX EMP buffers are loaded in order to simulate exactly the same dataflow from the datapath to the payload's AM algorithm instance.

The Hit data enter the AM algorithm's instance and are slightly altered as described in figure 5.16.

Valid 1bit	**Zeroes** 2bits	**TDC** 5bits	**BX** 12bits	**Ch** 8bits

$$\downarrow$$

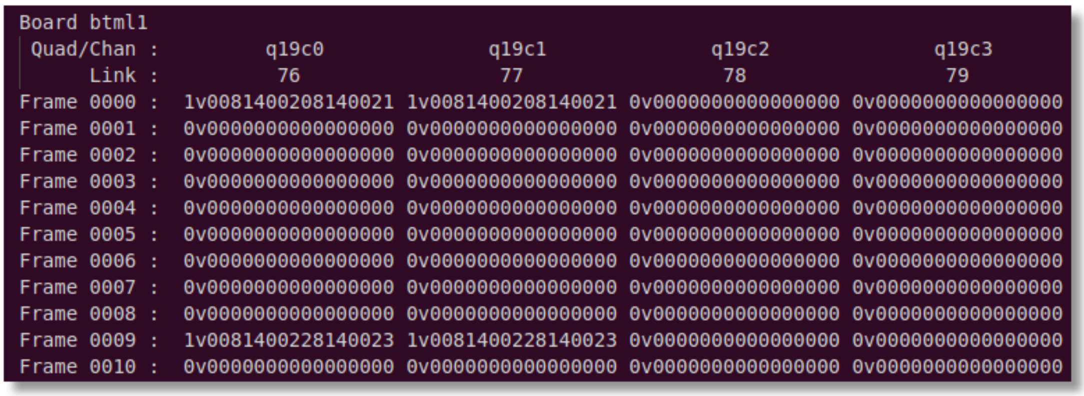
Valid 1bit	**Zeroes** 6bits	**Ch** 8bits	**BX** 12bits	**TDC** 5bits

Figure 5.16: Hit word format as Input and Output from the AM Algorithm

Once the altered Hit data are driven out of the *dt_sector* module, they are inserted into the Readout module. The process inside the Hit Readout module was described in 5.6.2. When the Hit data exit the Readout module they are transferred to the GBT sorter module in order to be transmitted to the TM7 board. At the TM7 board the user can observe the Hit data by calling the buffer script provided by the DT group. Since on every LHC orbit the data arriving are the same, that is until the user loads a different data file to the EMP buffers, the script will capture the Hit contents of multiple orbits, each orbit's Hit contents identical to the others. Given that the buffers on the TM7 board are 4096 cells deep, whereas the EMP buffers are only 1024 cells deep, it is guaranteed that all of the contents of the EMP buffers will be received, at least one orbit's Hit contents. The data can then be reviewed in order to determine if the functionality of the system is correct.

It also possible for the user to execute a Bash script on a remote PC that uses SSH to log in the ZYNQ on the BMTL1 board and execute another Bash script that contains all the Empbutler commands needed to load the Hit contents in the RX EMP buffers. The Bash script executed by the remote PC also uses ssh to log into the machine that controls the TM7 board and issue a command to execute the buffer script.

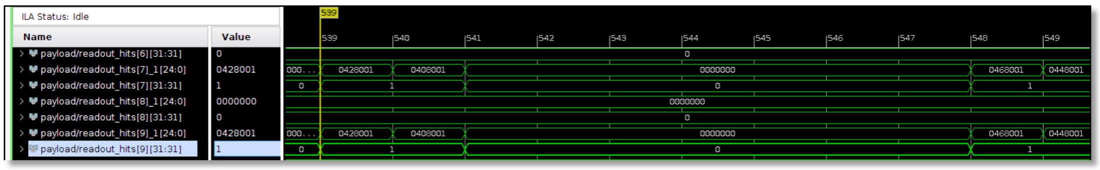
A small sample of the whole process can be shown in the figures 5.17, 5.18, 5.19. The RX EMP buffer is loaded with eight hits as shown in figure 5.17. Each 64 bit word contains two Hit words, whereas each 64 bit word is separated by eight frames of zeroes by the next data word. This is performed in order to correctly simulate the behaviour of lpGBT and GBT protocol that the OBDTs normally use, since each word from both of these protocols is delivered once per LHC clock cycle. The data words are injected to specific region channels, as agreed upon with the DT group developers. For this instance the inputs are mapped to simulate Hits from MB3 (chamber 3) SL1 and SL3.



```
Board btml1
 Quad/Chan :        q19c0              q19c1              q19c2              q19c3
      Link :         76                 77                 78                 79
Frame 0000 :  1v0081400208140021 1v0081400208140021 0v0000000000000000 0v0000000000000000
Frame 0001 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0002 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0003 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0004 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0005 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0006 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0007 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0008 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
Frame 0009 :  1v0081400228140023 1v0081400228140023 0v0000000000000000 0v0000000000000000
Frame 0010 :  0v0000000000000000 0v0000000000000000 0v0000000000000000 0v0000000000000000
```

Figure 5.17: USC test setup Layout

The Hit data arrive at the AM algorithm's module where the are processed. Their new format upon exit is as stated previously at 5.16. For instance the Hit word *0x81400021* is converted to *0x80428001*. In figure 5.18 the eight Hit words with the new format are shown[1], after exiting the AM algorithm's module.



```
ILA Status: Idle
Name                               Value
> payload/readout_hits[6][31:31]    0
> payload/readout_hits[7]_1[24:0]   0428001
> payload/readout_hits[7][31:31]    1
> payload/readout_hits[8]_1[24:0]   0000000
> payload/readout_hits[8][31:31]    0
> payload/readout_hits[9]_1[24:0]   0428001
> payload/readout_hits[9][31:31]    1
```

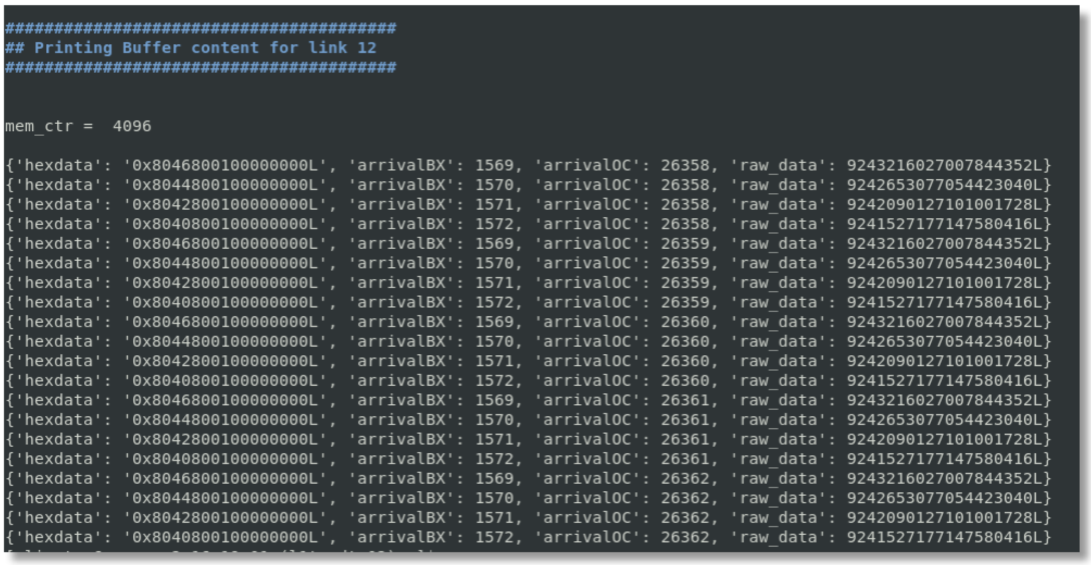
Figure 5.18: USC test setup Layout

Once the data enter the Readout module they are stored and sorted by the Hit module. The Hits enter the module from two separate lanes as shown in figure 5.18. Those lanes are part of the thirteen 32-bit vector array that is connected as an input to the Hit module. Since one lane containing data is the seventh

[1]The 32th bit is displayed separately from the rest of the word

element of the array and the other one the ninth element, the sorting mechanism will output each lane's contents to one GBT link, respectively.

The data are then transmitted to the the TM7 board where they are captured and displayed via the buffer script. In figure 5.8b the data from one of the two links are displayed for many orbits. The orbit count is indicated by the *arrivalOC* variable. In total four Hit words are expected and four Hit words are received per orbit as expected, each arriving at the exact same BXs as every other orbit's Hit data.



```
########################################
## Printing Buffer content for link 12
########################################


mem_ctr =  4096

{'hexdata': '0x8046800100000000L', 'arrivalBX': 1569, 'arrivalOC': 26358, 'raw_data': 9243216027007844352L}
{'hexdata': '0x8044800100000000L', 'arrivalBX': 1570, 'arrivalOC': 26358, 'raw_data': 9242653077054423040L}
{'hexdata': '0x8042800100000000L', 'arrivalBX': 1571, 'arrivalOC': 26358, 'raw_data': 9242090127101001728L}
{'hexdata': '0x8040800100000000L', 'arrivalBX': 1572, 'arrivalOC': 26358, 'raw_data': 9241527177147580416L}
{'hexdata': '0x8046800100000000L', 'arrivalBX': 1569, 'arrivalOC': 26359, 'raw_data': 9243216027007844352L}
{'hexdata': '0x8044800100000000L', 'arrivalBX': 1570, 'arrivalOC': 26359, 'raw_data': 9242653077054423040L}
{'hexdata': '0x8042800100000000L', 'arrivalBX': 1571, 'arrivalOC': 26359, 'raw_data': 9242090127101001728L}
{'hexdata': '0x8040800100000000L', 'arrivalBX': 1572, 'arrivalOC': 26359, 'raw_data': 9241527177147580416L}
{'hexdata': '0x8046800100000000L', 'arrivalBX': 1569, 'arrivalOC': 26360, 'raw_data': 9243216027007844352L}
{'hexdata': '0x8044800100000000L', 'arrivalBX': 1570, 'arrivalOC': 26360, 'raw_data': 9242653077054423040L}
{'hexdata': '0x8042800100000000L', 'arrivalBX': 1571, 'arrivalOC': 26360, 'raw_data': 9242090127101001728L}
{'hexdata': '0x8040800100000000L', 'arrivalBX': 1572, 'arrivalOC': 26360, 'raw_data': 9241527177147580416L}
{'hexdata': '0x8046800100000000L', 'arrivalBX': 1569, 'arrivalOC': 26361, 'raw_data': 9243216027007844352L}
{'hexdata': '0x8044800100000000L', 'arrivalBX': 1570, 'arrivalOC': 26361, 'raw_data': 9242653077054423040L}
{'hexdata': '0x8042800100000000L', 'arrivalBX': 1571, 'arrivalOC': 26361, 'raw_data': 9242090127101001728L}
{'hexdata': '0x8040800100000000L', 'arrivalBX': 1572, 'arrivalOC': 26361, 'raw_data': 9241527177147580416L}
{'hexdata': '0x8046800100000000L', 'arrivalBX': 1569, 'arrivalOC': 26362, 'raw_data': 9243216027007844352L}
{'hexdata': '0x8044800100000000L', 'arrivalBX': 1570, 'arrivalOC': 26362, 'raw_data': 9242653077054423040L}
{'hexdata': '0x8042800100000000L', 'arrivalBX': 1571, 'arrivalOC': 26362, 'raw_data': 9242090127101001728L}
{'hexdata': '0x8040800100000000L', 'arrivalBX': 1572, 'arrivalOC': 26362, 'raw_data': 9241527177147580416L}
```

Figure 5.19: USC test setup Layout

Chapter 6

Conclusions

This thesis details the Readout Design that was implemented on the firmware of the Barrel Muon Trigger Level-1 (BMTL1) board for the Compact Muon Solenoid (CMS) Experiment at CERN's High Luminosity Large Hadron Collider (HL-LHC). The firmware design module that was produced was tested and integrated with the rest of the system's framework.

During the time of the Phase-2 upgrades, the Phase-2 boards, link protocols and algorithms were tested. As the system is not in its final form, many test setups include Phase-1 equipment interfacing with the under development Phase-2 equipment. Multi board slice tests are performed in order to evaluate the Phase-2 upgraded hardware, firmware and software. The implementation of a Readout Design serves as a solution to monitor and evaluate data generated by the algorithms implemented on the Phase-2 boards. The Readout module offers a store and transmit mechanism, while using minimal FPGA resources. This addition to the firmware does not add to the complexity of the framework, nor does it disrupt the normal behaviour of the rest of the system.

As a result of the learning process of the system's framework, a simple synchronous link protocol was created. The protocol was created for educational purposes, aiming to achieve a better understanding of the protocols used in optical links both at the CMS experiment and link protocols in general. The link protocol was designed to be operated by the same tools that the Phase-2 system will use and was evaluated using the appropriate Phase-2 software tools.

Appendix A

Appendix A

A.1 HDL codes

Listing A.1: codefiles/protocol/TX_module.vhd

```vhdl

library IEEE;
use IEEE.STD_LOGIC_1164.all;

use IEEE.NUMERIC_STD.all;
use IEEE.std_logic_unsigned.all;


entity TX_module is
  port(reset_all       : in  std_logic;
       txusrclk2       : in  std_logic;
       tx_active       : in  std_logic;
       txheader_out    : out std_logic_vector(5 downto 0);
       txsequence_out  : out std_logic_vector(6 downto 0);
       txdata_out      : out std_logic_vector(63 downto 0);
       userword64_in   : in  std_logic_vector(63 downto 0);
       data_valid_bit  : in  std_logic;  -----if its 1 its writedata if its 0 its writecontrolword
       crc_error_vio   : in  std_logic
       );
end TX_module;

architecture Behavioral of TX_module is


  signal example_stimulus_reset_sync : std_logic;
  signal example_stimulus_reset_int  : std_logic;

  signal stopdata              : std_logic;
  signal cn                    : std_logic_vector(5 downto 0) := (
      others => '0');
  signal write_sync            : std_logic                    := '1';
  signal txdata_out_reg        : std_logic_vector(63 downto 0);
  signal txheader_out_reg      : std_logic_vector(5 downto 0);
  signal txheader_out_reg_del  : std_logic_vector(5 downto 0);
  signal crcen                 : std_logic;
  signal crc_out               : std_logic_vector(31 downto 0);
  signal txdata_out_scrambled  : std_logic_vector(63 downto 0);
  signal txdata_unscrambled    : std_logic_vector(63 downto 0);
  signal crcreset              : std_logic;
  signal crc_error_bit         : std_logic;
  signal scrambler_en          : std_logic;
  signal data_valid_bit_del    : std_logic;
  signal userword64_in_crc     : std_logic_vector(63 downto 0);
  signal userword64_in_del     : std_logic_vector(63 downto 0);
  signal crc_stored            : std_logic_vector(31 downto 0);
  signal crc_alt               : std_logic;

  component gtwizard_ultrascale_0_example_reset_synchronizer
    port(
      clk_in  : in  std_logic;
      rst_in  : in  std_logic;
      rst_out : out std_logic
      );
  end component;

  component SCRAMBLER_BLOCK
    generic (
      TX_DATA_WIDTH : integer
      );
    port (
      -- User Interface
      UNSCRAMBLED_DATA_IN : in  std_logic_vector(TX_DATA_WIDTH-1
          downto 0);
      SCRAMBLED_DATA_OUT  : out std_logic_vector(TX_DATA_WIDTH-1
          downto 0);
      DATA_VALID_IN       : in  std_logic;
      -- System Interface
      USER_CLK            : in  std_logic;
      SYSTEM_RESET        : in  std_logic
      );
  end component;

  component ucrc_par
    generic (
      POLYNOMIAL : std_logic_vector;
      INIT_VALUE : std_logic_vector;
      DATA_WIDTH : integer range 2 to 256
      );
    port (
      clk_i   : in  std_logic;          -- clock
      rst_i   : in  std_logic;          -- init CRC
      clken_i : in  std_logic;          -- clock enable
      data_i  : in  std_logic_vector(DATA_WIDTH - 1 downto 0);  -- data input
      match_o : out std_logic;          -- CRC match flag
      crc_o   : out std_logic_vector(POLYNOMIAL'length - 1 downto 0));  -- CRC output
  end component;

begin


  example_stimulus_reset_int <= reset_all or (not tx_active);

  reset_synchronizer2inst : gtwizard_ultrascale_0_example_reset_synchronizer
    port map(
      clk_in  => txusrclk2,
      rst_in  => example_stimulus_reset_int,
      rst_out => example_stimulus_reset_sync
      );



  process(txusrclk2)
    variable cnt   : integer := 0;
    variable order : integer := 0;
  begin
    if rising_edge(txusrclk2) then
      cn <= cn + '1';
      if example_stimulus_reset_sync = '1' then
        cn          <= (others => '0');
        write_sync <= '1';
        stopdata   <= '0';
      end if;
      if cn = "011111" then
        stopdata <= '1';
      else
        stopdata <= '0';
      end if;
      if cn = "100000" then
        cn <= (others => '0');
      end if;

      txsequence_out <= '0' & cn;


      if stopdata = '0' then
        if data_valid_bit_del = '1' then
          txdata_out_reg   <= userword64_in_del;  ----sending data
             words
          txheader_out_reg <= "000001";
--                    crcen<='1';
--                    crcreset<='0';
          write_sync       <= '0';
        elsif data_valid_bit_del = '0' then
--                    crcen<='0';
--                    crcreset<='1';
          if write_sync = '0' then
            write_sync <= '1';
            if crc_error_vio = '1' then
              if crc_alt = '1' then
                crc_alt          <= '0';
                txdata_out_reg   <= x"00000000" & crc_stored;
                txheader_out_reg <= "000010";
              else
                txdata_out_reg   <= x"00000000" & crc_error_bit &
                   crc_out(30 downto 0);
                txheader_out_reg <= "000010";
              end if;
            else
              if crc_alt = '1' then
                crc_alt          <= '0';
                txdata_out_reg   <= x"00000000" & crc_stored;
                txheader_out_reg <= "000010";
              else
                txdata_out_reg   <= x"00000000" & crc_out;
                txheader_out_reg <= "000010";
              end if;
            end if;
          else
            txdata_out_reg   <= x"5555555555555555";
            txheader_out_reg <= "000010";
          end if;
        end if;
        if data_valid_bit = '1' then
          crcen             <= '1';
          crcreset          <= '0';
          userword64_in_crc <= userword64_in;
        elsif data_valid_bit = '0' then
          crcen    <= '0';
          crcreset <= '1';
        end if;
        ----delay 1 clock the headers
```

```vhdl
      txheader_out_reg_del <= txheader_out_reg;


      ------data_valid_bit is delayed and we use the delayed one on the logic above.
      ------the not delayed one is used for enabling the crc generator one clock before everything.
      data_valid_bit_del <= data_valid_bit;


      -----the data from the external source (user) are inserted in the crc. and the delayed by one clock cycle data are used for the logic above.
      userword64_in_del <= userword64_in;

    elsif stopdata = '1' then
      crcen <= '0';
      ------special fix...comment out and run it to see the problem.
      if crcreset = '1' and crcen = '0' and write_sync = '0' and txheader_out_reg = "000001" then
        crc_stored <= crc_out;
        crc_alt    <= '1';
      end if;
    end if;
  end if;


end process;
txheader_out       <= txheader_out_reg_del;
txdata_out         <= txdata_out_scrambled;
txdata_unscrambled <= txdata_out_reg;

crcgen : ucrc_par
  generic map(
    POLYNOMIAL => "00000100110000010001110110110111",
    INIT_VALUE => "11111111111111111111111111111111",
    DATA_WIDTH => 64
    )
  port map (
    clk_i   => txusrclk2,             -- clock
    rst_i   => crcreset,              -- init CRC
    clken_i => crcen,                 -- clock enable
    data_i  => userword64_in_crc,     -- data input
    match_o => open,                  -- CRC match flag
    crc_o   => crc_out                -- CRC output
    );

crc_error_bit <= not crc_out(31);
```

```
  scrambler_en  <= not stopdata;

  scrambler : SCRAMBLER_BLOCK
    generic map(
      TX_DATA_WIDTH => 64
      )
    port map(
      -- User Interface
      UNSCRAMBLED_DATA_IN => txdata_unscrambled,
      SCRAMBLED_DATA_OUT  => txdata_out_scrambled,
      DATA_VALID_IN       => scrambler_en,
      -- System Interface
      USER_CLK            => txusrclk2,
      SYSTEM_RESET        => example_stimulus_reset_sync
      );

end Behavioral;
```

Listing A.2: codefiles/protocol/RX_module.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.NUMERIC_STD.all;

entity RX_module is
  port(reset_all          : in  std_logic;
       rxusrclk2          : in  std_logic;
       rx_active          : in  std_logic;
       rxheader_in        : in  std_logic_vector(5 downto 0);
       rxgearboxslip_out  : out std_logic;
       rxdata_in          : in  std_logic_vector(63 downto 0);
       rxdata_out         : out std_logic_vector(63 downto 0);
       sync               : out std_logic;
       rxdatavalid        : in  std_logic_vector(1 downto 0);
       rxheadervalid      : in  std_logic_vector(1 downto 0)
       );
end RX_module;

architecture Behavioral of RX_module is

  signal example_checking_reset_sync : std_logic;
  signal example_checking_reset_int  : std_logic;
  signal counter                     : std_logic_vector(6 downto 0)
     := (others => '0');
  signal syncok                      : std_logic
     := '0';
  signal rxgearboxslip_out_reg       : std_logic
```

```vhdl
    := '0';
signal cnt_reg                       : std_logic_vector(5 downto 0)
    := (others => '0');
signal error_reg                     : std_logic
    := '0';
signal tries                         : std_logic_vector(11 downto 0)
    := (others => '0');
signal rxdata_in_i                   : std_logic_vector(63 downto 0);
signal rxdata_out_reg                : std_logic_vector(63 downto 0);
signal rxheader_in_del               : std_logic_vector(5 downto 0);


component syncbit
  port (clk   : in  std_logic;
        i_in  : in  std_logic;
        o_out : out std_logic
        );
end component;

component DESCRAMBLER_BLOCK
  generic (
    RX_DATA_WIDTH : integer
    );
  port (
    -- User Interface
    SCRAMBLED_DATA_IN     : in  std_logic_vector(RX_DATA_WIDTH-1
        downto 0);
    UNSCRAMBLED_DATA_OUT : out std_logic_vector(RX_DATA_WIDTH-1
        downto 0);
    DATA_VALID_IN         : in  std_logic;
    -- System Interface
    USER_CLK              : in  std_logic;
    SYSTEM_RESET          : in  std_logic
    );
end component;

component ucrc_par
  generic (
    POLYNOMIAL : std_logic_vector;
    INIT_VALUE : std_logic_vector;
    DATA_WIDTH : integer range 2 to 256
    );
  port (
    clk_i   : in  std_logic;          -- clock
    rst_i   : in  std_logic;          -- init CRC
    clken_i : in  std_logic;          -- clock enable
    data_i  : in  std_logic_vector(DATA_WIDTH - 1 downto 0);  --
        data input
    match_o : out std_logic;          -- CRC match flag
```

```
      crc_o   : out std_logic_vector(POLYNOMIAL'length - 1 downto 0))
         ;  -- CRC output
  end component;

  signal crcreset             : std_logic;
  signal crcen                : std_logic;
  signal crc_match            : std_logic;
  signal crc_checksum         : std_logic_vector(31 downto 0);
  signal crc_checksum_stored : std_logic_vector(31 downto 0);
  signal crc_incoming         : std_logic_vector(31 downto 0);
  signal crc_check            : std_logic;

begin

  example_checking_reset_int <= reset_all or (not rx_active);

  reset_synchronizer1inst : syncbit
    port map(
      clk   => rxusrclk2,
      i_in  => example_checking_reset_int,
      o_out => example_checking_reset_sync
      );

  descrambler : DESCRAMBLER_BLOCK
    generic map(
      RX_DATA_WIDTH => 64
      )
    port map(
      -- User Interface
      SCRAMBLED_DATA_IN    => rxdata_in,
      UNSCRAMBLED_DATA_OUT => rxdata_in_i,
      DATA_VALID_IN        => rxdatavalid(0),
      -- System Interface
      USER_CLK             => rxusrclk2,
      SYSTEM_RESET         => example_checking_reset_sync
      );

  crccheck : ucrc_par
    generic map (
      POLYNOMIAL => "00000100110000010001110110110111",
      INIT_VALUE => "11111111111111111111111111111111",
      DATA_WIDTH => 64
      )
    port map (
      clk_i   => rxusrclk2,
      rst_i   => crcreset,
      clken_i => crcen,
      data_i  => rxdata_in_i,
      match_o => open,
      crc_o   => crc_checksum);


  process(rxusrclk2)

  begin

    if rising_edge(rxusrclk2) then
      if example_checking_reset_sync = '1' then
        counter               <= (others => '0');
        rxgearboxslip_out_reg <= '0';
        cnt_reg               <= (others => '0');
        syncok                <= '0';
        error_reg             <= '0';
        counter               <= (others => '0');
      else
        if rxdatavalid = "01" then
          -----header is delayed one clock cycle to be synchronous
              with the unscambled data
          -----it is delayed inside this loop because the descrambler
               is also enabled or disabled via the rxdatavalid.
          -----so we need them to be synchronous.
          rxheader_in_del <= rxheader_in;

          -------control word header
          if rxheader_in_del = "000010" then
            if rxdata_in_i = x"5555555555555555" then
              if counter = "001000" then
                syncok                <= '1';
                rxgearboxslip_out_reg <= '0';
                cnt_reg               <= (others => '0');
              else
                counter <= counter+'1';
                syncok  <= '0';
              end if;
            elsif rxdata_in_i(63 downto 32) = x"00000000" then
              -----maybe if sync='1' then
              crcreset <= '1';
              crcen    <= '0';
              if crc_checksum = rxdata_in_i(31 downto 0) or
                  crc_checksum_stored = rxdata_in_i(31 downto 0) then
                crc_match <= '1';
              else
                counter   <= (others => '0');
                syncok    <= '0';
                crc_match <= '0';
              end if;
            ---end if;
            else
```

```
              syncok  <= '0';
              counter <= (others => '0');
              cnt_reg <= cnt_reg+'1';
              if cnt_reg = "100000" then
                rxgearboxslip_out_reg <= '1';
                cnt_reg               <= (others => '0');
              else
                rxgearboxslip_out_reg <= '0';
              end if;
            end if;
          -------invalid header
          elsif rxheader_in_del = "000000" or rxheader_in_del = "
             000011" then
            cnt_reg <= cnt_reg+'1';
            if cnt_reg = "100000" then
              rxgearboxslip_out_reg <= '1';
              cnt_reg               <= (others => '0');
            else
              rxgearboxslip_out_reg <= '0';
            end if;
          -------data header
          elsif rxheader_in_del = "000001" then
            if syncok = '0' and counter = "0000000" then
              cnt_reg <= cnt_reg+'1';
              if cnt_reg = "100000" then
                rxgearboxslip_out_reg <= '1';
                cnt_reg               <= (others => '0');
              else
                rxgearboxslip_out_reg <= '0';
              end if;
            elsif syncok = '1' then
              rxdata_out_reg <= rxdata_in_i;
            end if;
          end if;
          if rxheader_in = "000001" then
            crcreset <= '0';
            crcen    <= '1';
          end if;
        elsif rxdatavalid = "00" then
          crcen <= '0';
          if rxheader_in_del = "000010" and rxdata_in_i(63 downto 32)
              = x"00000000" then
            crc_checksum_stored <= crc_checksum;
          end if;
        end if;
      end if;
    end if;
  end process;
  rxgearboxslip_out <= rxgearboxslip_out_reg;
  rxdata_out        <= rxdata_out_reg;
end Behavioral;
```

A.2 IPbus code

Listing A.3: codefiles/IPBus/control.py

```
import uhal
import time

manager = uhal.ConnectionManager("file://connection_file.xml")
hw = manager.getDevice("KCU105board")

syncok = hw.getNode("myreg.mystat.synclock").read()
hw.dispatch()

print("sync is =", syncok)

print("sending reset rx_datapath and tx_datapath_pll")

hw.getNode("myreg.myctrl.rst_tx_datapath_pll").write(1)
hw.getNode("myreg.myctrl.rst_rx_datapath").write(1)

time.sleep(3)

hw.getNode("myreg.myctrl.rst_tx_datapath_pll").write(0)
hw.getNode("myreg.myctrl.rst_rx_datapath").write(0)


while(True):
    syncok = hw.getNode("myreg.mystat.synclock").read()
    hw.dispatch()
    print("sync is =", syncok)
    time.sleep(1)
```

Acronyms

μTCA Micro Telecommunications Computing Architecture. 76, 78

2S 2-Strip. 10

ALICE A Large Ion Collider Experiment. 2

AM Analytical Method. 20, 74, 76–80, 82, 83, 87, 88

APD Avalanche PhotoDiodes. 11

API Application Programming Interface. 61

ASIC application-specific integrated circuit. 24, 26, 70, 73

ATCA Advanced Telecommunications Computing Architecture. 18, 59, 67, 74–76

ATLAS A Toroidal LHC Apparatus. 2, 4

BC0 Bunch Crossing 0. 4, 66, 76

BCT Barrel Calorimeter Trigger. 19

BMTF Barrel Muon Track Finder. 20

BMTL1 Barrel Muon Trigger Level-1. v, 20, 21, 74–80, 84–87, 90

BRAM Block RAM. 56, 61, 75, 81, 84

BX Bunch Crossing. 17, 18, 20, 74, 89

CDC Clock Domain Crossing. 45

CDR Clock Data Recovery. 40, 41

CERN Conseil Européen pour la Recherche Nucléaire. 1, 2, 4, 6, 90

CLB Configurable Logic Block. 24

CMS Compact Muon Solenoid. v, 2, 4, 6–9, 11, 13–19, 43, 70, 76, 90

CPLL Channel Phase Locked Loop. 34, 36

CPU Central Proccessing Unit. 18, 23–25, 61

CRC Cyclic Redundancy Check. 47, 48, 50, 53, 57, 72

CSC Cathode Strip Chamber. 13, 16, 20

CSP CMS Standard Protocol. 18, 21, 68, 70, 72, 75, 76, 78, 80, 84

CT Correlator Trigger. 21, 22

DAQ Data Acquisition. v, 18, 19, 22, 71, 76, 78

DFE Decision Feedback Equalizer. 40, 70

DSP Digital Signal Processing. 24, 75

DT Drift Tube. 13–17, 20, 72–76, 85–88

EB ECAL Barrel. 11

ECAL Electromagnetic Calorimeter. 6, 11, 12, 19

EDA Electronic Design Automation. 30

EE ECAL Endcap. 11

EMP Extensive Modular (data) Processor. 66, 68, 70, 79, 80, 86–88

EMTF Endcap Muon Track Finder. 20, 21

FEC Forward Error Correction. 70, 71

FIFO First In First Out. 28, 34, 43, 46, 59, 61, 81–84

FPGA Field Programmable Gate Array. v, 18, 24, 25, 28–32, 43, 59–62, 64, 66, 67, 71, 75, 76, 90

GBT GigaBit Transceiver. 68, 70, 71, 73, 76, 78–80, 82–89

GCT Global Calorimeter Trigger. 19–22

GEM Gas Electron Multiplier. 17

GMT Global Muon Trigger. 20–22, 75

GNU GNU's Not Unix. 31

GPIO General Purpose Input Output. 44

GPU Graphical Processing Unit. 18, 23

GT Global Trigger. 19–22

GTT Global Track Trigger. 21, 22

GUI Graphical User Interface. 65

HAL Hardware Access Library. 61, 63, 64

HB HCAL Barrel. 11, 12

HCAL Hadronic Calorimeter. 6, 11, 12, 19

HDL Hardware Description Language. 26, 28, 30

HE HCAL Endcap. 11

HF Hadron Forward Calorimeter. 19

HG-CAL High Granularity Calorimeter. 11–13, 19, 21

HL-LHC High Luminosity Large Hadron Collider. v, 4, 10, 90

HLT High Level Trigger. 18, 19, 23

IC Integrated Circuit. 26

ICMP Internet Control Message Protocol. 60

IEEE Institute of Electrical and Electronics Engineers. 26, 30, 48

ILA Integrated Logic Analyzer. 30, 55–58, 78, 85, 86

IP Intellectual Property. 28, 46–48, 55, 56, 65

IPBB IPbus Build tool. 64, 65

iRPC Improved RPC. 17, 20

KMTF Kalman Muon Track Finder. 76–78, 80

L1A Level-1 Accept. 19, 22

L1T Level 1 Trigger. 18, 21, 22

LFSR Linear Feedback Shift Register. 48

LHC Large Hadron Collider. v, 1–4, 6, 8, 9, 66, 67, 69–72, 76, 80, 84, 87, 88

LHCb LHC-beauty. 2

Linac4 Linear ACcelerator 4. 2

lpGBT Low Power GigaBit Transceiver. 18, 20, 68, 71, 73, 76, 79, 88

LPM Low Power Mode. 40, 70

LSB Least Significant Bit. 56, 70, 80–82, 84

LUT Look Up Table. 24, 75

LVDS Low Voltage Differential Signaling. 43, 44, 75

MGT Multi-Gigabit Transceiver. 32, 33, 43, 45, 67–70, 75

MMCM Mixed-Mode Clock Manager. 28

MSB Most Significant Bit. 56, 71, 81–83

OBDT On detector Board for Drift Tubes. 20, 72–74, 76–79, 81, 85, 86, 88

OMFT Overlap Muon Track Finder. 20, 21

P5 Interaction Point 5. 6, 7

PCB Printed Circuit Board. 39, 41, 67

PCS Physical Coding Sublayer. 33, 34, 36, 39, 41, 42, 44, 46

PF Particle Flow. 22

PI Phase Interpolator. 40

PISO Parallel In Serial Out. 34

PLD programmable logic device. 24

PLL Phase Locked Loop. 28, 34, 36, 40, 46, 68

PMA Physical Medium Attachment. 33, 34, 36, 39, 42, 46

PRBS Pseudorandom Binary Sequence. 34, 70

PROM programmable read-only memory. 24

PS Pixel-Strip. 10

PS Proton Synchrotron. 2

PSB Proton Synchrotron Booster. 2

PUPPI PileUp Per Particle Identification. 22

QPLL Quad Phase Locked Loop. 34, 36, 43

RAM Random Access Memory. 24

RISC Reduced instruction set computer. 25

RMW Read Modify Write. 59, 61

RPC Resistive Plate Chamber. 14, 16, 17, 20, 74, 75

RTL Register-Transfer Level. 28, 29, 55

SFP Small Form-Factor Pluggable. 43, 54, 62

SIPO Serial In Parallel Out. 34, 41, 42

SL SuperLayer. 15, 20, 73, 88

SLR Super Logic Region. 75

SOC System On Chip. 25, 31, 67, 75, 76

SPL Software Programming Language. 26

SPS Super Proton Synchrotron. 2, 3

TCDS Timing and Control Distribution System. 19, 22, 76

TCL Tool Command Language. 30, 65

TCP Transmission Control Protocol. 61

TDC Time Digital Converter. 20, 73, 74, 76, 79, 80

TF Track Finder. 21

TMB Trigger Motherboards. 20

TP Trigger Primitives. 18, 77, 78, 80, 82–84

TPG Trigger Primitive Generators. 74, 75

TTC Timing Trigger Control. 22, 66, 67, 71

TTS Trigger Throttling System. 23

UDP User Datagram Protocol. 59–61

USC Underground Service Cavern. 7, 18, 70, 76, 78, 85

UXC Underground eXperimental Cavern. 7, 18, 70, 76

VCO Voltage Controlled Oscillator. 34

VHDL VHISC (Very High Speed Integrated Circuit) Hardware Description Language. 26–28, 30, 50, 60, 61, 65

VIO Virtual Input/Output. 30, 45, 55, 56, 63

XDC Xilinx Design Constraints. 29, 30, 44, 65

XML Extensible Markup Language. 61, 62, 65

Glossary

A32/D32 The address bus A32 is used to specify a memory address. The data bus D32 is used to indicate the data. 59

Bash Bash is a Unix shell and command language. (Bourne Again Shell). 31, 87

Empbutler The Empbutler tool is the main command-line software tool used for controlling and monitoring the EMP framework's TTC control signals, RX/TX buffers and link firmware. 68, 70, 87

IPbus The IPbus protocol is a simple packet-based control protocol for reading and modifying memory-mapped resources within FPGAs. 30, 31, 55, 58–68, 78, 80

LC An electric circuit consisting of an inductor, represented by the letter L, and a capacitor, represented by the letter C, connected together. 34

Metastability The state a register can enter where its output has not reached its expected value and can oscillates between '1' and '0'. 45

pile-up the number of events per bunch crossing. 5, 6

Run Length Run length is defined as the number of identical contiguous symbols which appear in a signal stream. 48, 49

SerDes Serializer / Deserializer. 32

SSH SSH (Secure Shell) is a cryptographic network protocol commonly used for remote command-line login. 87

VMEbus VMEbus is a computer bus standard physically based on Eurocard sizes. 59

Bibliography

[1] O. Brüning, H. Burkhardt, and S. Myers. The large hadron collider. *Progress in Particle and Nuclear Physics*, 67(3):705–734, 2012.

[2] CERN Yellow Reports: Monographs. *Linac4 design report*. CERN, 2020.

[3] CERN. The Proton Synchrotron Booster. *CERN records*, 2012-07-18.

[4] CERN. The Proton Synchrotron. *CERN records*, 2012-01-23.

[5] CERN. The Super Proton Synchrotron. *CERN records*, 2012-01-23.

[6] CERN. Radiofrequency cavities. *CERN records*, 2012-09-17.

[7] B G Taylor. Timing distribution at the LHC. *CERN*, 2002.

[8] S. Chatrchyan, V. Khachatryan, A.M. Sirunyan, A. Tumasyan, W. Adam, et al. Observation of a new boson at a mass of 125 gev with the cms experiment at the lhc. *Physics Letters B*, 716(1):30–61, 2012.

[9] O. Aberle, I Béjar Alonso, O Brüning, P Fessia, L Rossi, L Tavian, et al. *High-Luminosity Large Hadron Collider (HL-LHC): Technical design report.* CERN Yellow Reports: Monographs. CERN, Geneva, 2020.

[10] S Chatrchyan, G Hmayakyan, V Khachatryan, A M Sirunyan, R Adolphi, G Anagnostou, et al. The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment. *JINST*, 3:S08004, 2008. Also published by CERN Geneva in 2010.

[11] CMS. The Phase-2 Upgrade of the CMS Tracker. Technical report, CERN, Geneva, 2017.

[12] CMS. The Phase-2 Upgrade of the CMS Endcap Calorimeter. Technical report, CERN, Geneva, 2017.

[13] G. Abbiendi, J. Alcaraz Maestre, A. Álvarez Fernández, B. Álvarez González, N. Amapane, I. Bachiller, L. Barcellan, C. Baldanza, et al. The analytical method algorithm for trigger primitives generation at the lhc drift tubes detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 1049:168103, 2023.

[14] CERN. The Phase-2 Upgrade of the CMS Muon Detectors. Technical report, CERN, Geneva, 2017. This is the final version, approved by the LHCC.

[15] CMS Collaboration. The Phase-2 Upgrade of the CMS Level-1 Trigger. Technical report, CERN, Geneva, 2020. Final version.

[16] Kaitlyn Franz. *History of the FPGA*. Digilent, 2021-06-17.

[17] IEEE. Ieee standard vhdl language reference manual. *IEEE Std 1076-1987*, pages 1–218, 1988.

[18] IEEE. Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[19] Xilinx. *Vivado Design Suite User Guide: Using Tcl Scripting (UG894)*. Xilinx, 2023-11-17.

[20] Eric Matthes. *Python Crash Course, 3rd Edition*. No Starch Press, January 2023.

[21] Xilinx. *UltraScale Architecture and Product Data Sheet: Overview (DS890)*. Xilinx, 2023-07-20.

[22] Xilinx. *UltraScale Architecture GTH Transceivers User Guide (UG576)*. Xilinx, 2021-08-18.

[23] A. X. Widmer and P. A. Franaszek. A dc-balanced, partitioned-block, 8b/10b transmission code. *IBM Journal of Research and Development*, 27(5):440–451, 1983.

[24] Xilinx. *KCU105 Board User Guide v1.10 (UG917)*. Xilinx, 2019-02-06.

[25] Xilinx. *UltraScale FPGAs Transceivers Wizard v1.7 LogiCORE IP Product Guide*. Xilinx, 2019-11-11.

[26] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

[27] Chris Borrelli. *IEEE 802.3 Cyclic Redundancy Check (XAPP209 v1.0)*. Xilinx, 2001-03-23.

[28] Avago. *Optical Transceiver Module AFBR-709SMZ 10GBASE-SR SFP+*. FS, 2023.

[29] Xilinx. *Virtual Input/Output v3.0 Product Guide (PG159)*. Xilinx, 2018-04-04.

[30] Xilinx. *Integrated Logic Analyzer v2.0 Data Sheet (DS875)*. Xilinx, 2012-07-25.

[31] C. Ghabrous Larrea, K. Harder, D. Newbold, D. Sankey, A. Rose, A. Thea, and T. Williams. Ipbus: a flexible ethernet-based control system for xtca hardware. *Journal of Instrumentation*, 10(02):C02019, feb 2015.

[32] Robert Frazier, Greg Iles, Marc Magrans de Abril, Dave Newbold, Andrew Rose, David Sankey, and Tom Williams. *The IPbus Protocol An IP--based control protocol for ATCA*. Cern, 2013-12-19.

[33] EMP Collaboration. *Emp-Fwk Guide Release FW 0.8.1 / SW 0.8.2 Rev2.0*. Cern, May 14th, 2019.

[34] S Baron, J P Cachemiche, F Marin, P Moreira, and C Soos. Implementing the GBT data transmission protocol in FPGAs. *Cern*, 2009.

[35] Paulo Moreira, Sophie Baron, Stefan Biereigel, João Carvalho, and Bram Faes. *lpGBT documentation: release*. Cern, 2022.

[36] I. Bestintzanos, K. Adamidis, I. Euaggelou, C. Fernandez Bedoya, C. Foudas, A. Lymperakis, N. Manthos, A. Navarro, I. Papadopoulos, I. Redondo, S. Sotiropoulos, P. Sphicas, and K. Vellidis. An atca processor for level-1 trigger primitive generation and readout of the cms barrel muon detectors. *Journal of Instrumentation*, 18(02):C02039, feb 2023.

[37] Michail Bachtis. The Ocean and Octopus designs for the Phase-2 upgrade of the CMS L1 muon trigger. TWEPP 2021 Topical Workshop on Electronics for Particle Physics. *CMS*, 2021.

[38] Á. Navarro-Tobar, A. Triossi, C. Fernández-Bedoya, I. Redondo, D. Redondo, J. Sastre, J.M. Cela-Ruiz, and L. Esteban. Phase 1 upgrade of the cms drift tubes read-out system. *Journal of Instrumentation*, 12(03):C03070, mar 2017.

[39] Xilinx. *UltraScale Architecture Libraries Guide (UG974)*. Xilinx, 2021-06-16.