UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia "Galileo Galilei"

Master's Degree in Physics of Data

Firmware development for the CMS DAQ readout cards

Supervisor

Prof. Andrea Triossi

Co-Supervisor

Dr. Christoph Schwick

Candidate

Emanuele Sarte

Anno Accademico 2024/2025

Abstract

The CMS experiment and in particular the DAQ group is preparing a substantial upgrade of the data acquisition system to cope with the data volumes expected from the Phase 2 CMS detector (more than 50 Tb/s). The hardware to read out the sub-detectors back-end cards have been re-designed to read out the back-end boards via custom 25 Gbps optical links and to transfer these data streams with a standard protocol (TCP/IP) to the CMS event building network consisting of commercial network equipment. The monitoring of the correct functioning of these data acquisition cards is essential during data taking. The cards are based on large Xilinx FPGA featuring High Bandwidth Memory (HBM) which is needed to buffer the incoming data stream before it is injected into the commercial TCP/IP network (socket buffers). However, the intention is to use some of the available memory for monitoring purposes by storing histograms or event fragments triggered by specific error conditions. The memory area needs to be accessible from the monitoring firmware in the FPGA and from the software controlling the card to read out the data and present it to the shift crew. The project focuses on the development of firmware components with easy to use interfaces, which will be used by the monitoring firmware and software.

Contents

Chapter 1

Introduction

In the pursuit of fundamental understanding within particle physics, experiments at the Large Hadron Collider contribute significantly to our knowledge of the subatomic realm. The Compact Muon Solenoid experiment, a key component of this research, continuously addresses the challenge of efficiently acquiring and processing large datasets from high-energy particle collisions. The High-Luminosity upgrade of the Large Hadron Collider, expected to commence operation in 2029, will provide a unique opportunity to extend the physics reach of the CMS experiment.

This upgrade of the HL-LHC will allow for precision studies of electroweak symmetry breaking and new physics via quantum loops [1]. Higgs boson couplings will be measured with high precision, and challenging rare Higgs decays, di-Higgs production, and longitudinal vector boson scattering will become accessible. Furthermore, the mass reach for direct searches of exotic particles will be extended, and blind spots can be uncovered. This era may lead to the discovery of particles responsible for dark matter, new resonances, or long-lived particles predicted by various extensions of the Standard Model.

As the LHC prepares for its HL-LHC phase, the Compact Muon Solenoid experiment necessitates a substantial upgrade of its data acquisition system. This upgrade addresses the projected data volumes from the Phase 2 CMS detector, which are expected to exceed 50 terabits per second. To manage this, the hardware for reading out the sub detectors backend cards has been redesigned, incorporating custom 25 Gbps optical links for data readout. These data streams are then aggregated and transferred using a standard TCP/IP protocol to the new CMS DAQ system, which utilizes commercial network equipment.

At the core of this redesigned system are the new DTH-400 cards. These cards are built around large Xilinx FPGAs that include High Bandwidth Memory (HBM). The HBM primary role is to temporarily hold incoming data before it is sent to the TCP/IP network. Currently, however, there are empty sections within the HBM on these FPGAs that are not used for any purpose. The idea is to utilize these otherwise unused memory areas to implement monitoring functionalities, such as histograms or the potential storage of erroneous event fragments for future upgrades.

This thesis focuses on the development of firmware components. These components are designed to create and store histograms for monitoring data, and to make this histograms data accessible from outside the board. Specifically, the DTHistogram project, designed for the DTH-400 DAQ board, enables the creation, updating, and storage of histograms derived from incoming data streams, thus collecting operational statistics directly on the FPGA. This developed firmware

must then be integrated with the main CMS DAQ firmware used for data taking.

The structure of this thesis is as follows: the second chapter provides a comprehensive overview of the Large Hadron Collider and the CMS experiment, detailing the current operational landscape and the imperative upgrades leading to the High-Luminosity LHC era, particularly highlighting the increased demands on data acquisition. The third chapter delves into the conceptual and technological framework of the CMS DAQ system, with a specific focus on its evolution and new requirements for Phase 2 operations. The fourth chapter introduces the fundamental concepts of FPGAs, including their internal memory resources like Block RAM and High Bandwidth Memory , and explains the AXI protocol critical for internal and external communication. The fifth chapter presents the core contribution of this thesis, providing a detailed overview of the DTHistogram project, its objectives, the definition and structure of a histogram within this context, the memory layout within the HBM, and the overall data flow. Finally, the sixth chapter elaborates on the Hardware Description Language design of the DTHistogram, describing the various entities comprising the system, detailing the external communication mechanisms, and introducing the custom protocol developed for intra-entity communication.

Chapter 2

LHC and CMS Upgrades

2.1 The Large Hadron Collider (LHC)

The Large Hadron Collider (LHC)[2], located at CERN, is the world's most powerful superconducting particle accelerator. It is designed to collide two beams of protons (or heavy ions) at extremely high energies, reaching up to 13–14 TeV in the center of mass frame. The LHC is housed in a 27-kilometer underground tunnel, utilizing thousands of powerful superconducting magnets cooled to near absolute zero to guide and focus the particle beams. The primary purpose of the LHC is to conduct fundamental research in particle physics. The LHC's design and operational capabilities allow physicists to probe the fundamental laws of nature at the smallest scales and highest energies.

2.1.1 Current Operations and Performance

The LHC operates in distinct periods called "runs". Following its initial successful operations (Run 1 and Run 2), which led to significant discoveries, the LHC is currently in its Run 3 phase.

The operation of the Large Hadron Collider (LHC) fundamentally relies on the precise manipulation and collision of its particle beams. These beams are not continuous streams of particles but are instead organized into discrete groups called bunches[4]. Each bunch contains a vast number of protons (on the order of 10^{11} protons per bunch) or heavy ions, tightly packed and accelerated to relativistic speeds. These bunches are then arranged into sequences known as bunch trains, which circulate in the LHC ring separated by empty spaces or "gaps." The term "particle packets" can be considered synonymous with "bunches" in this context, referring to these particle groupings.

The collision process occurs at four specific interaction points around the LHC ring, where the counter-rotating proton (or ion) beams are brought to a head-on collision. At these points, the accelerator optics are configured to minimize the beam size, maximizing the collision probability. Due to the high number of bunches and their rapid circulation, these collisions occur at a very high frequency, known as the bunch crossing rate, which is typically 40 MHz. This means that bunches from opposite beams cross each other every 25 nanoseconds. Each collision point hosts one of the four main LHC experiments: ATLAS, CMS, LHCb, and ALICE. Each experiment is designed to detect and record the particles produced in these high-energy interactions.

The operational effectiveness of the LHC is measured by its instantaneous luminosity $\mathcal{L}$, which indicates the rate of collisions, and its integrated luminosity, representing the total number

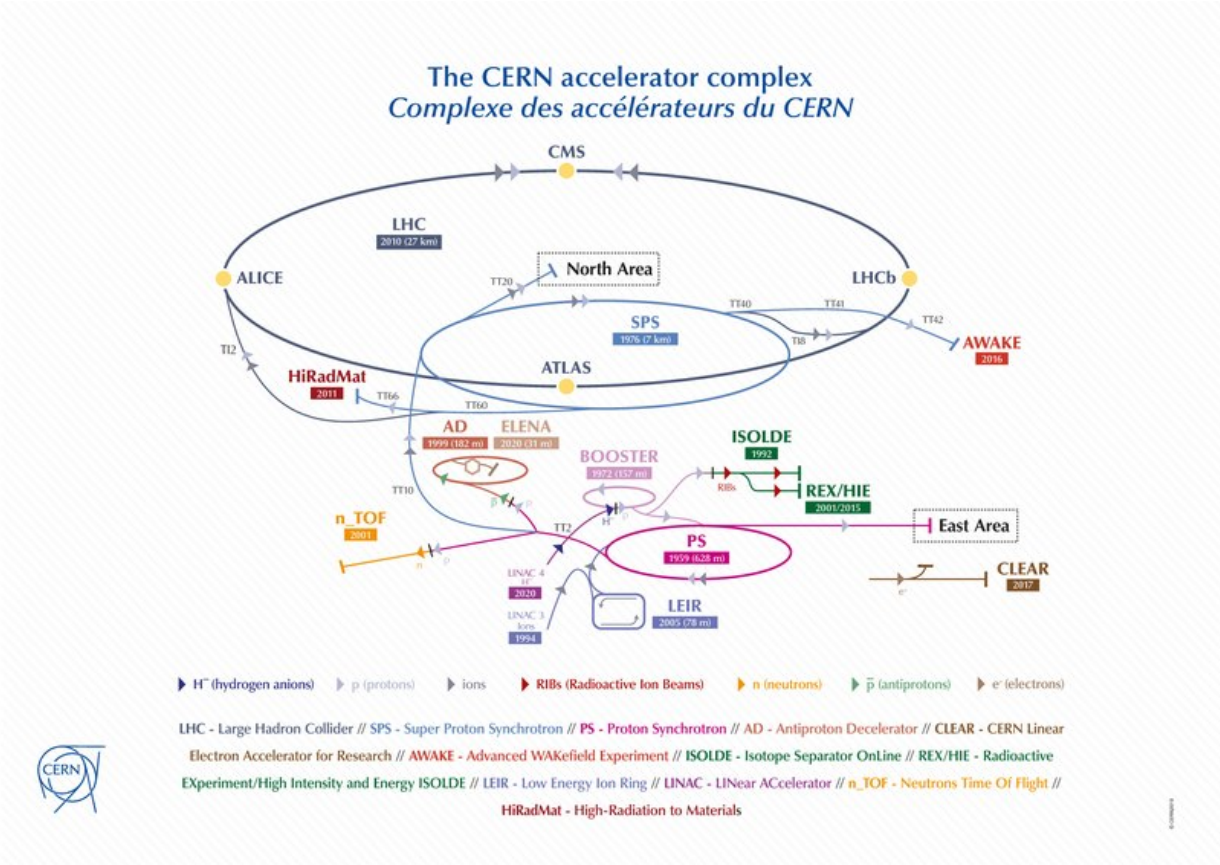

Figure 2.1: The CERN Accelerator Complex [3]

of collisions collected over time. Currently, the average instantaneous luminosity for Run 3 is $\mathcal{L} = 2 \cdot 10^{34}$ cm^{-2} s^{-1}. High instantaneous luminosity is essential for accumulating large datasets, especially for observing rare particle interactions. A direct consequence of an high luminosity is "pile-up", where multiple proton-proton collisions occur within a single beam crossing. In current LHC operations, the average number of pile-up events per bunch crossing $\langle\mu\rangle$ can be in the tens. This environment creates challenges for the experiments, as signals from different collisions can overlap, complicating the precise reconstruction of individual particles and their origins. To handle the immense data volume generated at the collision point (at a rate of 40 MHz), the LHC experiments employ sophisticated multi-level trigger systems. These systems rapidly filter and select only a small fraction of "interesting" events (e.g., thousands per second) for storage and further analysis, while discarding the vast majority of less relevant data.

2.1.2 The High-Luminosity LHC (HL-LHC) Upgrade

The LHC is scheduled for a major enhancement known as the High-Luminosity LHC (HL-LHC) upgrade, with operations expected to begin in the 2030s. The main goal of the HL-LHC is to significantly increase the total integrated luminosity by a factor of 5 to 10 compared to the original LHC design. This translates to achieving peak instantaneous luminosities over 5×10^{34} cm^{-2}s^{-1} and accumulating an integrated luminosity of roughly 3000 fb^{-1} (inverse femtobarns) per experiment over its lifespan. This substantial increase in data volume will enable more precise measurements of known particles, such as the Higgs boson, and greatly enhance the sensitivity to search for very rare processes and new, heavy particles that require larger datasets for their observation.

The HL-LHC will introduce unprecedented operational conditions for the detectors. The most

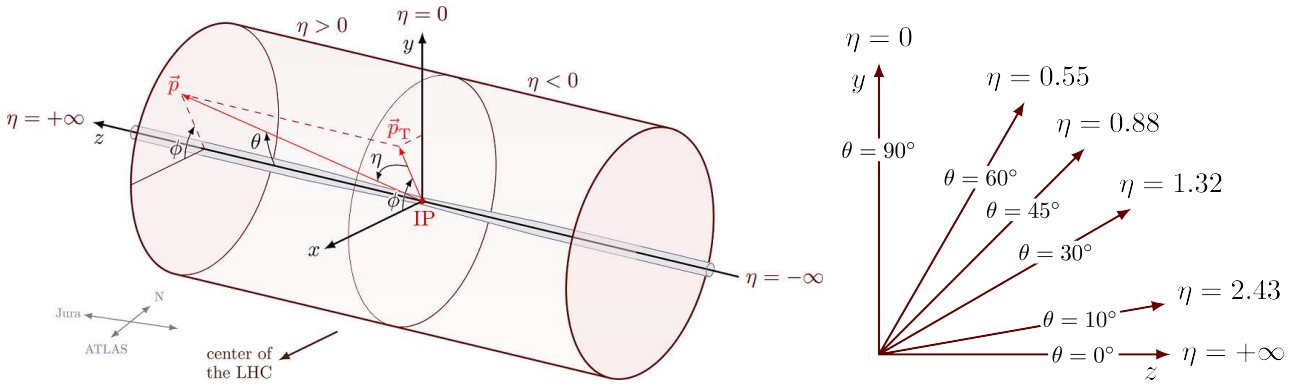

Figure 2.2: (a) Representation of CMS coordinate system. (b) Range of η values. Adapted from [6]

significant challenge will be the extreme level of pile-up, with the average number of interactions per bunch crossing expected to reach $\langle\mu\rangle \approx 200$. This high density of collisions will place immense demands on the experimental apparatus. Detector components must be highly radiation-hard to withstand the increased particle flux and accumulated radiation doses. Additionally, detectors will need higher spatial granularity, faster response times, and improved timing resolution to accurately distinguish between interactions from different primary vertices and mitigate the effects of pile-up. The data acquisition (DAQ) systems will face a massive increase in raw data rates, requiring more advanced bandwidth capabilities. Similarly, the trigger systems will need significant upgrades to maintain their efficiency in selecting relevant physics events in this extremely busy environment, incorporating more complex algorithms and advanced hardware.

2.2 The CMS Experiment

The Compact Muon Solenoid (CMS) experiment is one of the two large general purpose detectors operating at the Large Hadron Collider. Its design is centered around a powerful 3.8 T superconducting solenoid magnet, which provides a strong magnetic field for precise momentum measurement of charged particles. The detector is a multi-layered, cylindrical apparatus approximately 21 meters long and 15 meters in diameter, weighing around 14000 tonnes. Its structure is built concentrically around the beam pipe, with different detector systems optimized for identifying and measuring various types of particles emerging from the collision point. It has relatively small overall dimensions, achieved by placing the hadronic calorimeter inside the solenoid coil.

When it comes to detecting the particles produced from high-energy collisions, it's crucial to define a consistent way to pinpoint their locations. The coordinate system adopted by CMS[5] has its origin centered at the nominal collision point inside the experiment. In this system, the y-axis points vertically upward, and the x-axis points radially inward toward the center of the LHC (as shown in figure 2.2). Consequently, the z-axis points along the beam direction toward the Jura mountains from LHC Point 5. The azimuthal angle ϕ is measured from the x-axis in the xy plane, and the radial coordinate in this plane is denoted by r. The polar angle θ is measured from the z-axis. Pseudorapidity is defined as: $\eta = -\ln\left(\tan\left(\frac{\theta}{2}\right)\right)$. Another important quantity in this context is the transverse momentum: $p_T = \sqrt{p_x^2 + p_y^2}$.

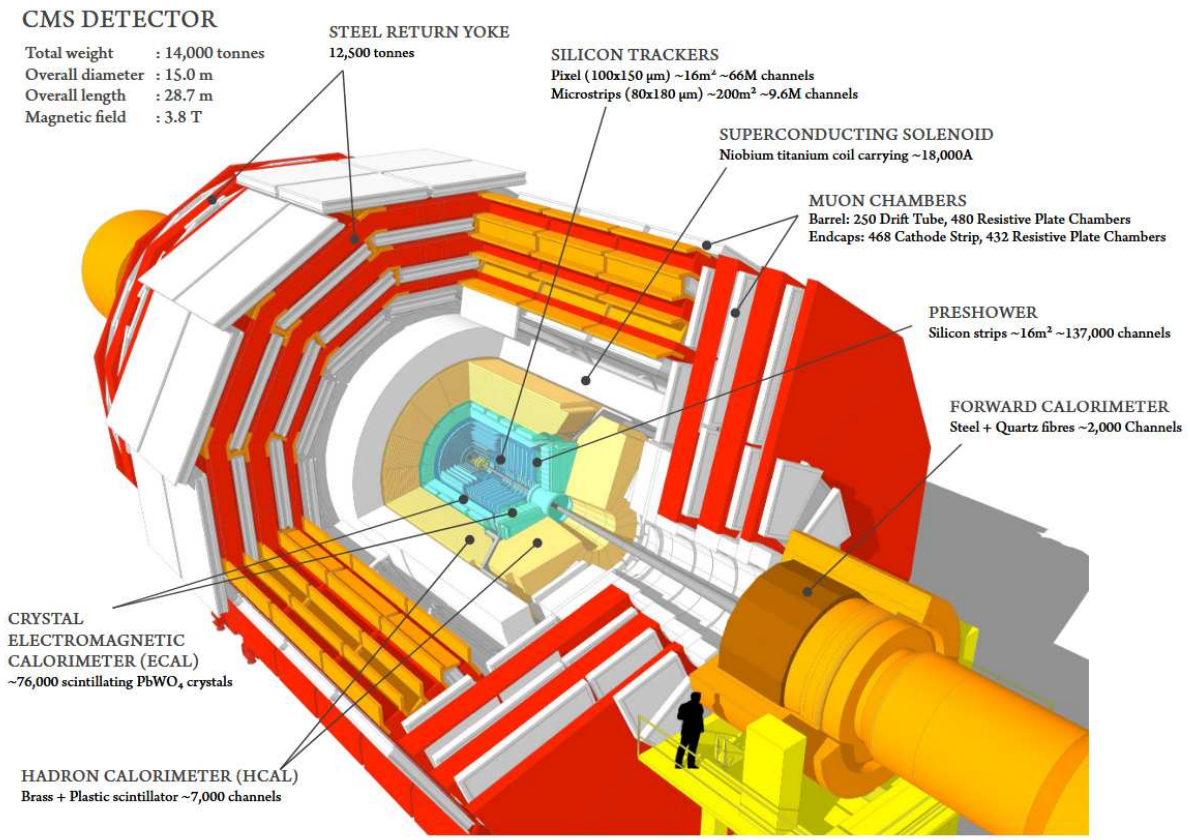

Figure 2.3: Cutaway view model of the CMS experiment. Taken from [7].

2.2.1 Subdetectors

Inner Tracking System

Closest to the interaction point is the Inner Tracking System, which provides precise measurements of charged particle trajectories. It is entirely composed of silicon detectors, offering high spatial resolution and radiation hardness. The tracker operates within the 3.8 T magnetic field of the solenoid, enabling accurate determination of particle momentum by measuring the curvature of their paths. The system is divided into two main parts[8]:

- **Pixel Detector:** The innermost layers consist of silicon pixel detectors, providing the highest granularity and precision for reconstructing primary and secondary vertices. It comprises three barrel layers and two endcap disks on each side, with pixel sizes typically around $100 \times 150\,\mu\text{m}^2$.

- **Strip Tracker:** Surrounding the pixel detector are ten layers of silicon strip detectors, covering a larger volume up to a radius of 1.1 meters and extending up to a pseudorapidity of $|\eta| < 2.5$. The strip tracker provides robust tracking over a wide area and contributes significantly to the overall momentum measurement and track isolation. The entire tracker consists of approximately 200 m^2 of silicon sensors with 9.3 million silicon strip channels and 76 million silicon pixel channels.

The comprehensive tracking system is fundamental for precise momentum measurements, track isolation for lepton identification, and reconstruction of interaction vertices.

Calorimeters

Outside the tracking system are the calorimeters, designed to measure the energy of both electromagnetic and hadronic particles.

- **Electromagnetic Calorimeter (ECAL)[9]:** The ECAL is a high-resolution calorimeter primarily responsible for measuring the energy of electrons and photons. It is a homogeneous calorimeter made of approximately 76000 lead tungstate ($PbWO_4$) crystals. These crystals are chosen for their high density, fast scintillation light decay time (around 25 ns), and radiation hardness. The ECAL is segmented into a barrel region covering $|\eta| < 1.479$ and two endcap regions extending up to $|\eta| < 3.0$. Photodiodes (Avalanche Photodiodes in the barrel and Vacuum Phototriodes in the endcaps) read out the light from the crystals. The ECAL's excellent energy resolution, particularly for high-energy photons and electrons, is crucial for precision measurements of processes.

- **Hadron Calorimeter (HCAL)[10]:** Surrounding the ECAL, the HCAL measures the energy of hadrons (protons, neutrons, pions, etc.) and provides information for jet reconstruction and missing transverse energy measurements. It is a sampling calorimeter, constructed with alternating layers of dense absorber material (brass in the barrel and steel in the endcaps) and scintillating plastic tiles. The HCAL is divided into a barrel, endcap, and forward sections, ensuring nearly full angular coverage up to $|\eta| < 5.2$. The HCAL relies on Hybrid Photodiodes and later Silicon Photomultipliers for light readout. The combined ECAL and HCAL system plays a vital role in reconstructing jets and determining the total missing transverse energy, which is a key signature for undetectable particles like neutrinos or hypothetical dark matter particles.

Muon System

The outermost component of the CMS detector is the Muon System, designed for efficient identification, triggering, and precise momentum measurement of muons. Muons are distinguished by their ability to penetrate significant amounts of material without interacting strongly, allowing them to traverse the inner detector and calorimeters before reaching the dedicated muon chambers. The CMS muon system is integrated within the flux return yoke of the 3.8 T solenoid, which comprises 12 layers of massive steel plates (ranging from 15 cm to 60 cm thick) [11]. The system utilizes three types of gaseous detectors to provide complementary information:

- **Drift Tubes (DTs):** Located in the barrel region ($|\eta| < 1.2$), DT chambers provide high-precision measurements of muon trajectories in the bending plane, crucial for accurate momentum determination. They are robust and reliable for tracking muons in the lower-rate barrel environment.

- **Cathode Strip Chambers (CSCs):** Positioned in the endcap regions ($0.9 < |\eta| < 2.4$), CSCs are designed to operate in high-rate environments and in the presence of strong, non-uniform magnetic fields. They provide excellent spatial and timing resolution, particularly important for muons in the forward direction.

- **Resistive Plate Chambers (RPCs):** These are fast gaseous detectors deployed in both the barrel and endcap regions ($|\eta| < 1.8$). RPCs provide precise timing information, which is critical for associating muon tracks with the correct bunch crossing in high pile-up conditions and for robust triggering.

The redundancy provided by the three different muon detector technologies ensures high efficiency and purity in muon identification and reconstruction across a wide kinematic range.

2.2.2 Trigger and Data Acquisition System

The extreme collision rate at the LHC (40 MHz bunch crossing rate) necessitates a sophisticated multi-level trigger and data acquisition (DAQ) system to manage the enormous data volume. If all collision data were to be recorded, it would vastly exceed storage and processing capabilities. Its core function is to handle the massive data flow generated by the detector, specifically reading out approximately 700 detector backend boards at about 100 kHz. A crucial aspect of this system is its ability to perform event building and distribution with a throughput of around 100 GB/s [12].

The primary role of the trigger system is to reduce the raw event rate to a manageable level (typically around 1 kHz for recording to disk) while preserving as many "interesting" physics events as possible and rejecting the overwhelming background of uninteresting interactions.

The CMS trigger system is organized into two main stages:

- **Level-1 (L1) Trigger:** This is the first stage, implemented entirely in custom-built hardware (FPGAs). It makes a decision on whether to accept an event within a fixed latency of about 4 μs. The L1 Trigger uses coarse-granularity information from the calorimeters and muon detectors to identify signatures of high-energy leptons (electrons, muons), photons, jets, and significant missing transverse energy. It is designed to be highly selective, reducing the 40 MHz bunch crossing rate to an output rate of approximately 100 kHz. Events passing the L1 Trigger are then transferred to the High-Level Trigger system.

- **High-Level Trigger (HLT):** This is the second stage, implemented as a farm of commercial computing nodes running sophisticated software algorithms. The HLT processes the full-granularity data from events accepted by the L1 Trigger. It performs more complex reconstruction and selection algorithms, leveraging detailed detector information, similar to those used in offline analysis but optimized for speed. The HLT further reduces the event rate from approximately 100 kHz down to about 1 kHz, depending on the operational conditions and physics priorities. Events accepted by the HLT are then stored for offline analysis.

The DAQ system is composed of several sophisticated components that facilitate this data flow. The Trigger Throttling System and Trigger Control and Distribution System (TCDS) manage trigger logic and signal distribution. Data is handled by custom electronics modules like FEROL and FEROL-40 boards, which receive detector data and transmit it over high speed Ethernet links. This data then traverses a Data Concentrator Network to Readout Unit (RU) Servers which aggregate data into "superfragments". The Event Builder Network facilitates the assembly of complete events by Builder Unit (BU) Servers. Finally, Filter Unit Servers execute the HLT algorithms, and the Storage and Transfer System (STS) handles the merging, buffering, and transfer of HLT-selected events to CERN's Tier 0 computing center. A block diagram of structure of the CMS DAQ system is shown in figure 2.4.

The evolution of the CMS DAQ system for Run 3 has seen significant technological advancements. Notably, the adoption of 100 Gb/s Ethernet for the Event Builder and the integration of general purpose GPUs for HLT processing have enhanced performance and cost effectiveness. The system is designed to handle an event building rate of 100 kHz with an event size of 1.6 MB at peak luminosity, demonstrating robust performance with a throughput of about 10 GB/s per combined RU/BU node.

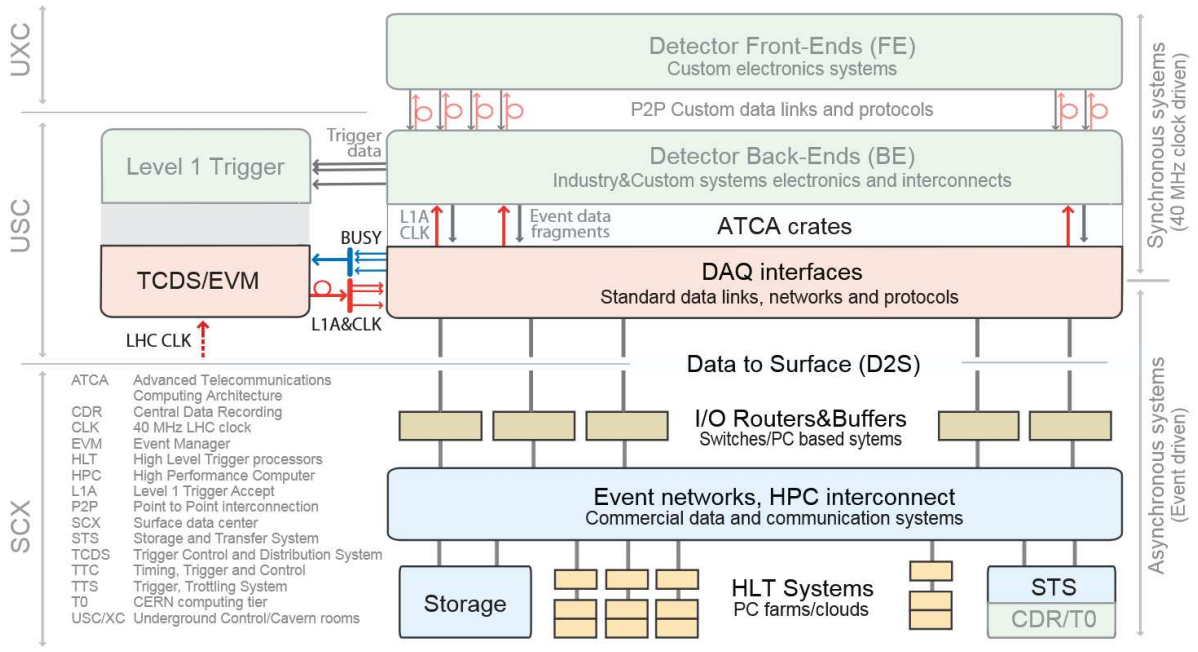

Figure 2.4: Block diagram illustrating the conceptual structure of the CMS DAQ. The detector and the front-end systems are located in the experimental cavern (UXC). The back-end electronics, together with the DAQ interface electronics and the trigger and timing distribution are located in the service cavern (USC), while the rest of the DAQ system is located in the surface data center (SCX).

2.2.3 Event Reconstruction with the CMS Detector

Following the path through the trigger and data acquisition chain, each event is stored, awaiting comprehensive analysis. The initial phase of this analysis is event reconstruction, which involves the precise identification of high level physics objects. This complex task is guided by the Particle Flow (PF) algorithm[13], a method that synthesizes and correlates data from all CMS subdetectors to reconstruct the detailed dynamics of particle interactions. The entire process commences with the decoding of raw detector signals, subsequently moving through stages of pattern recognition, track finding, and iterative fitting to accurately derive these essential high level physics objects.

The reconstruction of charged particle tracks is performed using the Combinatorial Track Finder algorithm[14], which operates on the principles of a Kalman Filter. This process unfolds in three sequential steps: first, the generation of initial "seeds" from a few compatible detector hits; second, the propagation of these initial trajectory estimates layer by layer through the detector, actively searching for additional compatible hits using the Kalman Filter technique; and finally, a precise fit to determine the definitive track parameters. Once tracks are reconstructed, they are clustered to pinpoint both primary interaction vertices and secondary decay vertices. These identified vertices are then ranked based on the quadratic sum of the transverse momenta of the associated daughter tracks.

The CMS detector is engineered for exceptional muon reconstruction performance. This is achieved by combining information derived independently from both the silicon tracker and the dedicated muon chambers. Muons are then categorized into three distinct types: standalone muons (reconstructed solely from muon system hits), tracker muons (tracker tracks extrapolated and matched with muon segments), and global muons (a combined fit of both tracker and standalone tracks)[15]. This comprehensive approach yields muon identification efficiencies exceeding 96%. The transverse momentum resolution for muons is remarkably precise, typically around 1% in the barrel region and 3% in the endcaps for $p_T < 100$ GeV, maintaining a

resolution better than 7% even for muons with p_T up to 1 TeV.

Electrons and photons are primarily reconstructed by clustering their energy deposits within the Electromagnetic Calorimeter [16]. Since electrons interact with the material before the ECAL, radiating bremsstrahlung photons, and photons can convert into electron-positron pairs, a Gaussian Sum Filter tracking algorithm is employed to refine electron tracks, and "super-clusters" are formed to encompass the energy deposited by these radiating particles. Energy resolution for electrons from Z decays with $p_T \approx 45$ GeV typically spans from 1.6% to 5%, generally performing better in the barrel region, while unconverted or late converting photons achieve an impressive resolution of about 1% in the barrel.

Hadronic jets are formed by clustering reconstructed charged and neutral hadrons using the infrared and collinear safe anti-kT algorithm, with a cluster parameter of R=0.4 [17]. The momentum of these jets is carefully corrected for contributions from additional proton-proton collisions (pileup) by discarding tracks from pileup vertices and applying an offset correction. Jet energy corrections are derived from simulations to ensure the measured response aligns with "particle-level" jets, further refined by in site measurements of momentum balance. The jet energy resolution demonstrates significant improvement with increasing transverse momentum, typically reaching 5% at 1 TeV.

Finally, hadronically decaying tau leptons τ_h are identified using the "hadrons plus strips" algorithm[18], which combines specific track and calorimeter energy deposits. To distinguish genuine τ_h decays from backgrounds such as jets originating from quark or gluon hadronization, as well as electrons or muons that might be misidentified as τ, the advanced Deep Tau algorithm is utilized.

2.3 CMS Upgrades for HL-LHC Era

The Phase-2 upgrade involves substantial improvements across various detector subsystems[19]:

- **Tracker System:** The silicon tracker will be completely replaced, featuring an Inner Tracker that is pixel-based and an Outer Tracker with strips and macropixels. This new design enhances radiation resistance and increases granularity, allowing for the reconstruction of 1200 tracks per unit of η. It also has a much reduced material budget, preserving calorimeter resolution. The Phase-2 tracker will contribute to the Level-1 Trigger decision through Outer Tracker p_T modules.

- **High Granularity Calorimeter (HGCAL):** The endcap calorimeter is being entirely replaced with the HGCAL, a 5D calorimeter (position, energy, and time) with coverage in the $1.5 < |\eta| < 3.0$ region, segmented into 47 layers. It uses silicon cells in CE-E and the high radiation part of CE-H, and scintillator cells elsewhere. The HGCAL ReadOut Chip provides fast signal shaping (signal peak in less than 25 ns) and $\tilde{2}5$ ps timing resolution, while being radiation tolerant up to 2 MGy. The endcap calorimeter has to contribute to the L1T decision.

- **MIP Timing Detector (MTD):** A new MTD layer is introduced to measure the production time of minimum ionising particles. One of its main motivations is pileup mitigation, as interactions in a given bunch crossing are spread in longitudinal dimension and time. The MTD will also play a role in new physics searches and heavy ion runs, thanks to its particle identification capability. It is divided into the Barrel Timing Layer ($|\eta| < 1.45$) and the Endcap Timing Layer ($1.6 < |\eta| < 3.0$). The timing resolution for the MTD reaches 30-65 ps in the barrel region, and 35 ps per track in the endcap.

- **Barrel Calorimeters and Muon Detectors:** The electronics systems of the electromagnetic and hadronic barrel calorimeters will undergo upgrades in their electronics systems to address extended latency and higher trigger rate requirements. The muon spectrometers will also undergo Phase-2 upgrades, with existing components receiving electronic upgrades and new muon chambers being built to enhance the detection of forward muons. These include two improved RPCs stations ($1.9 < |\eta| < 2.4$) and three new muon stations based on Gas Electron Multiplier technology.
- **Trigger and Data Acquisition (DAQ) Systems:** A redesigned trigger and DAQ system is being implemented. The experiment will continue using a two-level trigger system. The Phase-2 Level-1 Trigger will be implemented in custom-made electronics and will analyze detector information with a maximum latency of 12.5 μs at a 40 MHz rate, selecting events for further inspection at an output rate of up to 750 kHz. It will receive inputs from the calorimeters, the muon system, and, in a significant change from Phase-1, from the Outer Tracker. The Phase-2 High Level Trigger will be implemented as a series of software algorithms running in a heterogeneous architecture computing farm. The DAQ system will be readout by ≈50000 high speed frontend optical links capable of sustaining up to 60 Tb/s data rate.
- **Beam Radiation, Instrumentation and Luminosity (BRIL) Systems:** The BRIL systems are all integrated within CMS. Their goal is to optimize the protection and lifetime of CMS's subdetectors and to provide real-time luminosity measurements with a precision of 2% online and 1% after offline calibrations. New components include a dedicated data path from the Inner Tracker Endcap Pixel detector for luminosity measurements, new systems based on Outer Tracker p_T modules, and a Fast Beam Conditions Monitor. A pair of systems of neutron monitors are also being developed.

These comprehensive upgrades will enable CMS to sustain high performance under the demanding HL-LHC conditions, thereby maximizing its physics program.

Chapter 3

The CMS Data Acquisition System

The DAQ system is designed to provide a data pathway and time decoupling between the synchronous detector readout and data reduction, the asynchronous selection of interesting events in the High-Level Trigger, local storage at the experiment site, and transfer to Tier-0 for offline analysis. The conceptual structure of the CMS Data Acquisition system remains similar between Phase-1 and the upcoming Phase-2, while the specific technologies, and performance parameters are significantly upgraded for to meet the demands of the High-Luminosity LHC. The subsequent section will elaborate on this common structure. In figure 3.1 is shown the diagram of the current CMS DAQ.

3.1 Overview

The L1 Trigger is a synchronous, hardware based system built with custom electronic boards. Its primary role is to drastically reduce the event rate from the LHC's 40 MHz bunch crossing frequency to a manageable output rate. For the current Phase-1, the maximum L1 accept rate is 100 kHz. However, for the High-Luminosity LHC (HL-LHC) Phase-2, a completely redesigned L1 trigger system will increase this rate up to 750 kHz to maintain signal selection efficiency and enhance the selection of new physics.

The L1 Trigger operates with a low latency, processing information within approximately 4 μs for Phase-1 and 12.5 μs for Phase-2. It utilizes coarse grained information from detector frontend (FE) electronics, which collect, process, and digitize signals from sensors. These frontend electronics are radiation tolerant and insensitive to stray magnetic fields. The backend (BE) electronics, located in a radiation free service cavern, control and synchronize the frontends and receive digitized data, routing relevant portions to the L1 trigger processors. Bidirectional links connect the FE and BE, with downlinks distributing the master clock, L1 accept signals, and fast control signals, while uplinks transport digitized detector data.

Crucially, the L1 Trigger system is designed to be "deadtime-less" under normal conditions. This means that pipelines at every synchronous processing stage are sized to store data for the maximum required latency without losses, ensuring that data is either passed on or dropped.

Despite its deadtime-less design, the system must be protected from buffer overruns due to finite buffer sizes. These overruns can result from problematic detector channels or machine conditions. To prevent this, the synchronous portion of the DAQ includes a Trigger Throttling System (TTS). The TTS collects buffer status from individual BE boards and controls the issuing of L1 accept signals to prevent buffer overflows. If buffers approach their capacity, the

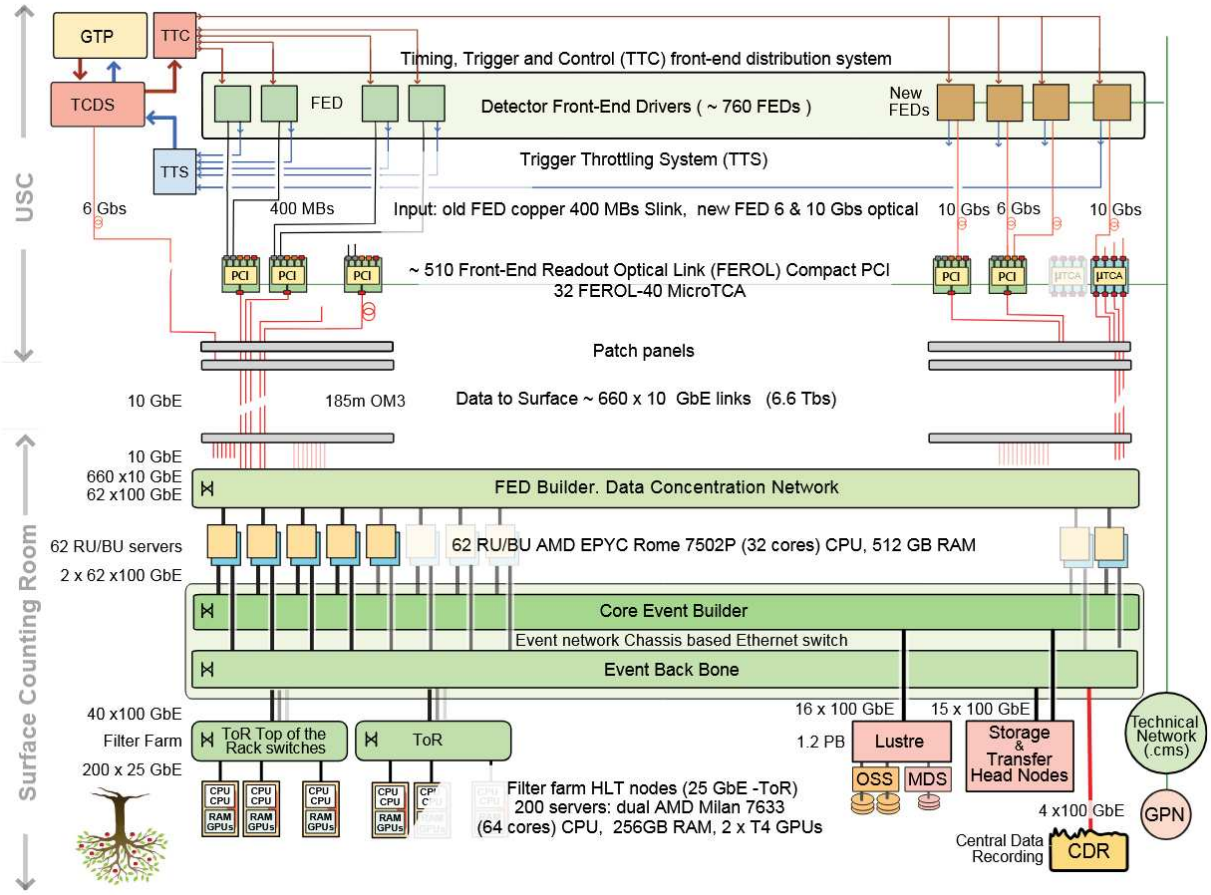

Figure 3.1: Diagram of the Run 3 CMS DAQ system[12].

TTS will temporarily halt the generation of L1 accept signals, introducing a controlled deadtime to allow buffers to clear. In standard Phase-1 operation, the effective deadtime was around 3%, with 1% resulting from preemptive trigger vetos to protect against frontend buffer overflows or calibration sequences. The main goal of this synchronous DAQ portion is to guarantee the collection of all selected data and keep effective deadtime minimal. The Trigger and Timing Control and Distribution System (TCDS) combines the functions of master clock distribution, trigger accept signal distribution, fast control signals, and buffer status collection for TTS. The TCDS also addresses potential desynchronization of detector frontends by issuing fast resynchronization signals, preventing out of sync data from propagating and avoiding the need for full DAQ pipeline draining.

Upon an L1 accept, accepted event data are transferred from each BE board to a DAQ concentrator over asynchronous point to point links with flow control. Data are then concentrated and transported to the surface (Data to Surface, D2S) over high speed links using a standard network protocol into the memory of commercial servers (I/O processors). A lossless protocol is preferred for D2S to guarantee protection against congestion at the destination. The timescale for this step is approximately 1 ms.

A high-performance switched network (Event Network) interconnects the I/O processors, enabling the assembly of all data fragments corresponding to individual events into the memory of a single computer, a process called event building. After events are built, they are stored in a local buffer on the I/O processor until HLT processes can pick them up for analysis. The HLT processing timescale is on the order of 1 second, but a larger latency (up to 1-2 minutes) is necessary to accommodate large service time fluctuations typical of software based selection algorithms, especially during initialization. This larger buffer was implemented in CMS DAQ

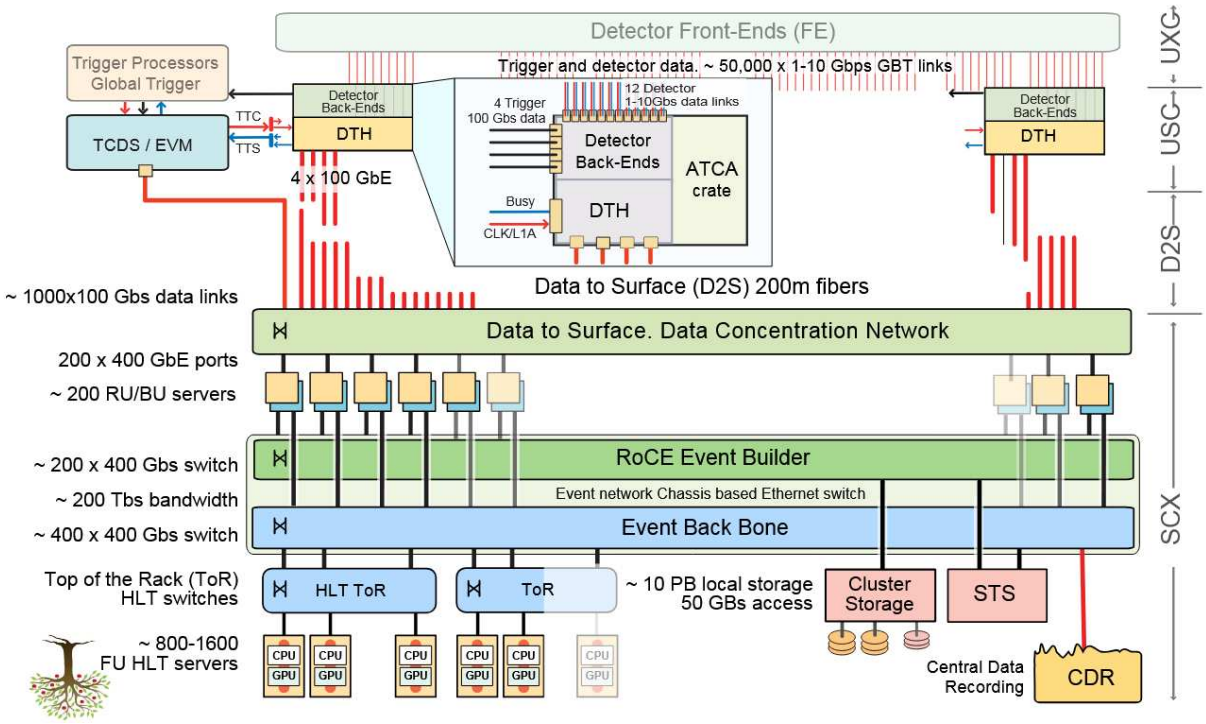

Figure 3.2: Layout of the CMS Phase-2 DAQ. The experiment and front-end electronics are located in the underground experimental cavern (UXC). The back-end electronics crates including the DAQ equipment are located in the underground service cavern (USC). The DAQ links (D2S) connect the USC to the surface complex (SCX) where the DAQ data center is located.

in Run-2 using a memory based filesystem interface. The required size of this event network buffer will increase for Phase-2 due to increased event size and L1 rate.

Finally, accepted events are stored locally at the experiment site before being assembled into larger dataset files for efficient long term storage. This local storage decouples data acquisition from the transfer process, providing enough buffer to absorb transfer speed fluctuations and enable uninterrupted data taking even during transfer link outages. This is achieved using distributed or network storage attached to the event building network. Entire dataset files are then transferred to central computing resources (Tier-0) over long distance links for offline reconstruction.

3.2 DAQ for Phase 2

The following sections will detail the conceptual design and implementation of the CMS Data Acquisition system and the Data to Surface network for Phase-2 of CMS, starting with Run 4 of the LHC [20]. In figure 3.2 is shown the diagram for CMS Phase-2 DAQ.

The CMS data acquisition system transfers data from detector frontends to backend electronics via optical links. The backend electronics serve two purposes: providing data to the Level-1 trigger for a fast decision and supplying data fragments for triggered events to the DAQ system for High Level Trigger selection and storage. Backend boards and frontend electronics must buffer incoming data to accommodate the L1 trigger latency.

Upon an L1 trigger, event data are pushed into the DAQ system. The interface between DAQ and backend systems uses optical links with a custom protocol. Data are received by the D2S boards, where checks are performed, and then aggregated and transferred to the surface data center for event building. A standard network protocol is used from the D2S boards onward, allowing for commercial components in the D2S network.

The Phase-2 Trigger and Timing Control and Distribution System (TCDS2) is the hardware system specifically designed for the low-level control and synchronization of CMS data taking, encompassing all related subsystems. Its core functions include the distribution of a high quality, phase stabilized version of the HL-LHC bunch clock to all CMS subsystems, and the overall synchronization of the CMS experiment to the HL-LHC. Beyond clock distribution, TCDS2 is responsible for handling the distribution of Level-1 physics triggers and generating additional L1As for calibration and testing purposes. It also manages the generation and distribution of timing and synchronization commands, which importantly can carry payload data.

The CMS DAQ system requires every backend board to send a data fragment for each triggered event, and a back pressure mechanism prevents data loss due to buffer overruns. If a component cannot handle the data flow, it signals its sources to pause data transmission. If buffers remain full, the back pressure propagates upstream until backend board buffers are nearly exhausted, at which point the backend board requests the TCDS2 to temporarily stop L1 triggers until the system is ready.

The connection between backend and D2S boards uses point to point links with a custom protocol, including back pressure and retransmit capabilities, to ensure lossless data transfer. The readout link can be implemented in backend FPGAs due to their advanced high speed serial link. Connections to D2S boards must be made from the front panel of boards since ATCA back-plane specifications lack sufficient links.

The D2S system must guarantee delivery of every event fragment for every L1 trigger to the event builder, as all events in CMS analysis require data from all subsystem components. In cases of congestion (e.g., higher than expected L1 trigger rates), the D2S system must be able to throttle the data stream and potentially reduce or pause the L1 trigger to prevent data loss or corruption. The D2S system acts as an interface between the synchronous frontend readout and the asynchronous DAQ.

Detector backends or frontends only buffer data for the L1 latency (12.5 μs). After each L1 trigger, backend boards transfer event data to D2S boards. Since the total bandwidth of incoming backend links can exceed outgoing D2S links, and due to the statistical nature of L1 triggers, D2S boards must temporarily buffer event data without generating back pressure on incoming links. D2S board buffer memory also needs to accommodate interruptions in data extraction from receiving network interface cards. While a realistic simulation for required buffer space is not available, measurements in a test system will confirm the sufficiency of the 256 MB per TCP stream buffer. This is more than five times larger than the successfully used 45 MB buffer in Run-2.

Incoming data streams from subsystem backends will be aggregated to higher bandwidth links at the D2S board output for cost optimization. The D2S system transports data from the underground service cavern to the surface data center, requiring link technology capable of bridging distances up to 200 m. The D2S system design must be robust against hardware failures, allowing quick reconfiguration to work around failing optical links or event building nodes. Due to cost, a fully redundant readout system is not feasible; failing D2S boards must be replaced before data taking resumes.

3.2.1 Data To Surface System

The interface between CMS subsystems and the central DAQ is implemented with two serial links connecting subsystem backend boards to Data to Surface boards. One link, for the TCDS2 system, runs over the ATCA crate backplane and provides a precision clock, sends

Figure 3.3: Image of the DTH-400 board. The CMS DAQ FPGA is under the top black heat sink.

synchronization commands (including L1 triggers), and allows leaf cards to send buffer status information to control trigger emission.

The second interface link connects subsystems to the central DAQ for event data. This link reads out event data fragments from backend boards using a custom point to point protocol called SLinkRocket. For each triggered event, subsystems must send a data fragment using SLinkRocket formatted records, which are prepared by backend FPGAs. Finally, the data, after being aggregated, is sent to the CMS Event Builder on the surface.

Due to varying readout bandwidth requirements among subsystems, two different D2S boards are foreseen. The DTH-400 board (shown in figure 3.3) provides TCDS2 functionality, implements the network switch, and handles data readout with a maximum average throughput of 400 Gb/s. For subsystems needing higher bandwidth, the DAQ-800 board offers an additional 800 Gb/s readout bandwidth by containing two readout blocks equivalent to the DTH-400, but without TCDS functionality or a network switch.

The job of a DTH-400 board is to collect data fragments from backend boards for every L1 trigger, aggregate them into larger data packets, convert them to standard network protocols, interface with TCDS2, and provide a network switch for the ATCA shelf general purpose Ethernet network. To do so, the board has 24 input links transferring SlinkRocket packets at 25 Gb/s each from the backend boards, connected with Firefly connectors. The output links instead consist of 5 QSFP28 connectors transferring data at 100 Gb/s over TCP/IP streams.

The part of the DTH-400 designed to handle the incoming fragments, process them and send them to the Event Builder is called a DAQ Unit. A DAQ Unit is implemented within a Xilinx VU35P FPGA. In order to be able to efficiently use the outgoing bandwidth, the unit needs to

work with sufficiently large data packets to keep the overhead at a negligible level. Therefore the DAQ Unit aggregates the incoming data fragments into larger TCP/IP packets before sending them to the event builder. For Phase-2 it has been decided to aggregate data per LHC orbit. For every SLinkRocket link, data fragments belonging to the same LHC orbit are being aggregated into a larger data packet (orbit packet) before transferring them via TCP/IP links to the Event Builder. At a L1 trigger rate of 750 kHz this results in ≈ 67 events per packet on average, and each packet is produced every $\approx 88\mu$s (one LHC orbit).

Using the standard TCP/IP protocols offers the advantage of utilizing standard network protocols and readily available commercial network equipment, thereby eliminating the need for custom hardware development. Specifically, a lightweight but standard compliance version of the TCP protocol has been implemented into the FPGA to avoid introducing unnecessary complexity.

In cases of congestion on the output link, potentially due to the statistical nature of the L1 trigger, or any issues with subsequent links, the DAQ Unit provides up to 256 MB of buffering for each incoming link. This is achieved by utilizing the high performance HBM memory, which will be discussed in detail in the next chapter.

The DTH-400 also incorporates a Zynq processor. This processor can communicate with the DAQ Unit FPGA and features an Ethernet port for network connectivity. In figure 3.4 is shown a diagram of the DTH-400 board.

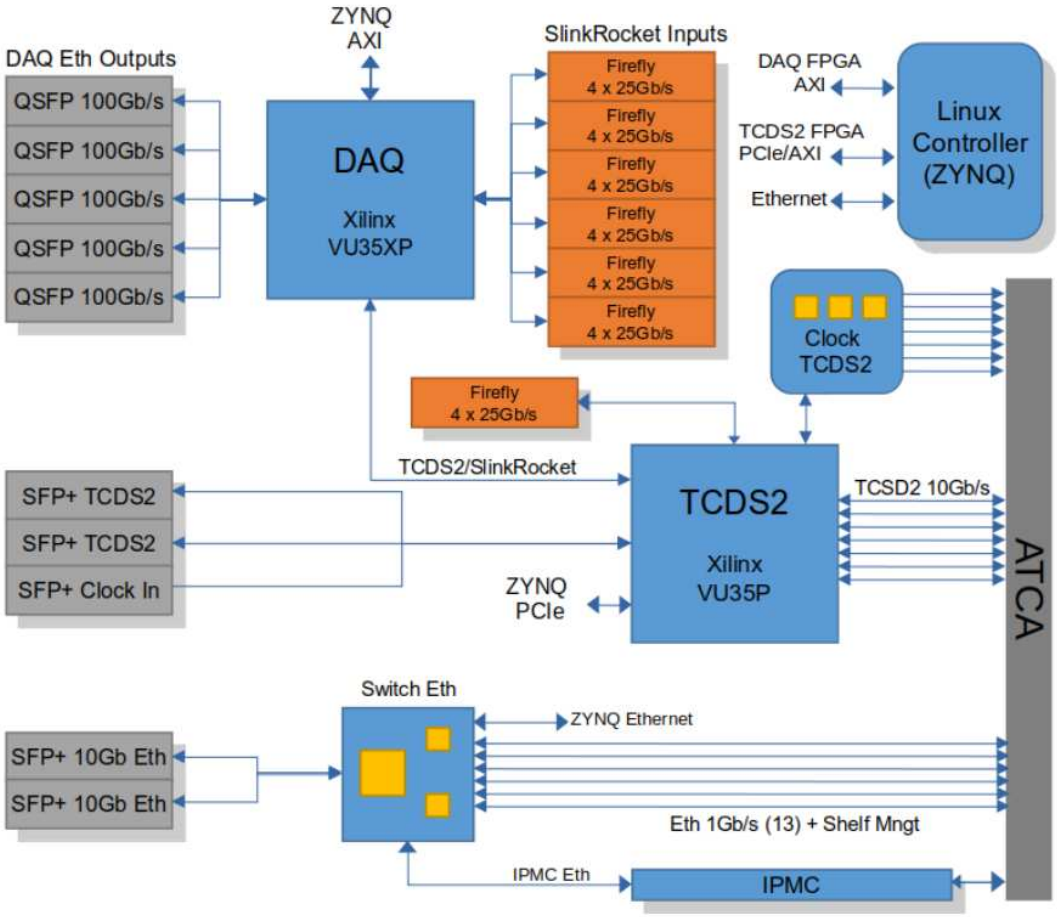

Figure 3.4: Block diagram of the DTH-400 board.

Orbit Aggregation

Data is received by the DAQ Unit (from the backend) through multiple SLinkRocket links. For saving bandwidth these fragments are aggregated per incoming optical link, ensuring that outgoing data packets contain data exclusively from a single SLinkRocket link. The Event Builder directs all data packets for a specific LHC orbit to the same Builder Unit (BU). Subsequently, the BU forwards all packets for that LHC orbit to the designated High Level Trigger node. The initial processing stage within the HLT node involves unpacking the data fragments and assembling individual events before the main HLT processing commences.

The unpacking process necessitates specific metadata within the data. Additional metadata is also beneficial for conducting data consistency checks. To facilitate this, the DAQ Unit generates two distinct headers: the Orbit Header and the Fragment Trailer. The Orbit Header is a data structure preceding each orbit data packet, which is a concatenation of individual SLinkRocket fragments from triggered events within that orbit. Each of these fragments is then followed by a Fragment Trailer, also generated by the DAQ Unit. In figure 3.5 the orbit header is shown, while in figure 3.6 the fragment trailer.

To summarize the data journey within the DTH-400: for each incoming link to the DAQ Unit, all fragments corresponding to a specific LHC orbit are collected. These fragments arrive via the SLinkRocket Protocol and are then aggregated by link and orbit. An orbit packet is subsequently formed, comprising an orbit header, followed by all collected fragments, each succeeded by a fragment trailer. This complete orbit packet is then encapsulated within a TCP/IP packet and transmitted to the Builder Unit by the Data To Surface system.

The fields from the orbit header are the ones used by the monitoring system, which is the subject of this thesis (but any other observable can be used). By filling histograms with the value of specific fields, it's easy to spot some criticality or anomalies.

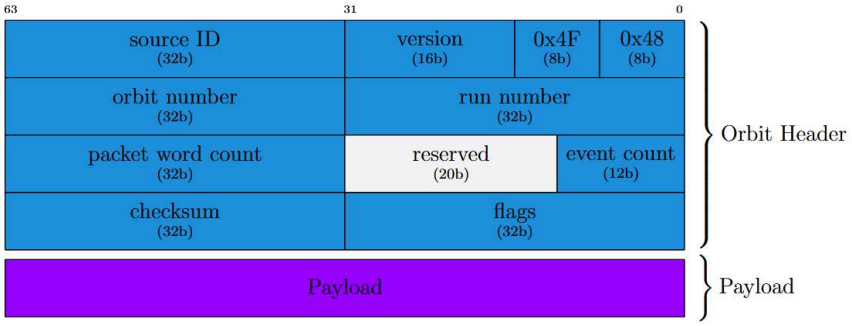

Figure 3.5: The Orbit Header format. This header is placed in front of orbit aggregated data packets formed by the DAQ-Unit. It contains information which is needed to unpack the relevant event data in the HLT nodes as well as meta information which allow to check the consistency of the data.

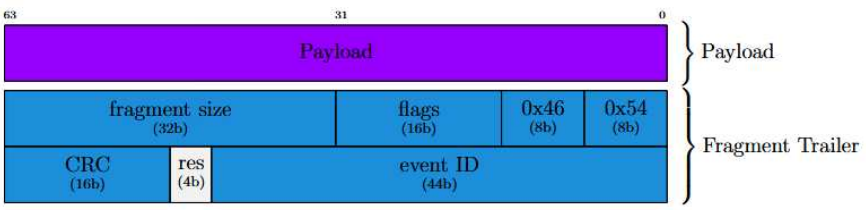

Figure 3.6: he Fragment Trailer format.

The SLinkRocket Protocol

The SLink is a family of custom point to point link protocols used for the transmission of event data fragments from the subsystem backends to the central DAQ system. For Phase-2 the CMS DAQ project provides an SLinkRocket sender core to be used by all subsystem backends. The SLinkRocket sender is an IP core for Xilinx FPGAs, providing an interface similar to a FIFO. A schematic version is shown in figure 3.7. The IP core receives data and a write clock, with a control line indicating valid data, and provides control lines for link readiness and a back-pressure signal. The back pressure signal indicates a full downstream buffer, requiring the subsystem to pause data writing within 16 cycles to avoid data loss. Implementers can choose between two reference clocks and two transfer rates (15.7 Gb/s or 25.8 Gb/s). The underlying low level protocol is a lossless custom protocol over an optical bidirectional link. Data records are divided into smaller packets, acknowledged by the receiver upon successful reception, and CRC checksums verify data integrity. A retransmit mechanism guarantees successful delivery. The link is controlled from the receiver side, sending simple commands to the sender (e.g., initialize link, signal readiness, configure event generator).

In figure 3.8 is shown an SLinkRocket packet, with all its field. Each packet begins with a header and ends with a trailer. In the payload all the data fragments coming from the backend boards are aggregated (separately for each link).

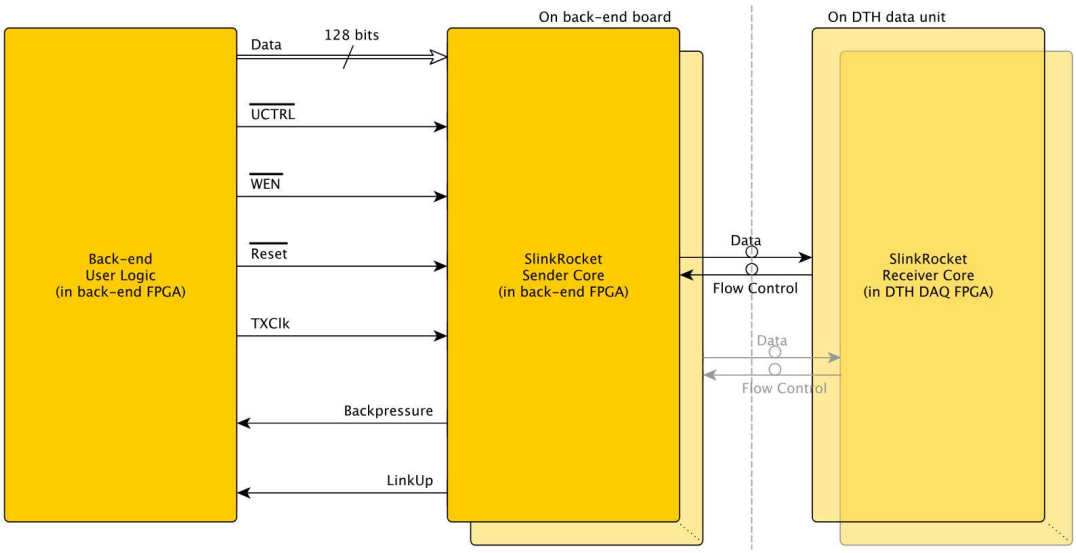

Figure 3.7: Simplified representation of the SLinkRocket sender core user interface and the SLinkRocket connection from a subsystem backend to the DTH. Each backend board can contain multiple SLinkRocket cores, and each firmware core corresponds to one bidirectional optical DAQ link between the back-end board and the DTH.

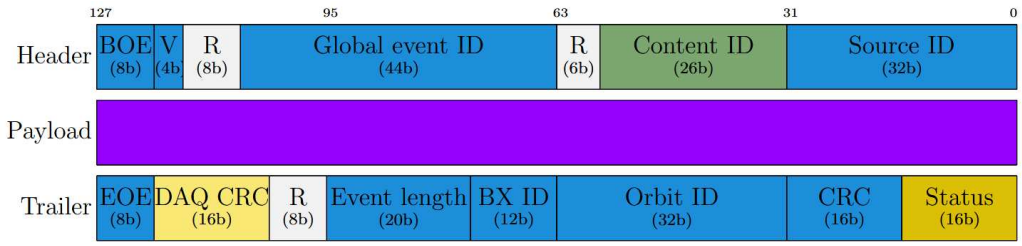

Figure 3.8: The SLinkRocket packet with its fields.

Chapter 4

Field Programmable Gate Arrays

For CMS Phase-2, the new DTH-400 boards in the data acquisition system are responsible for collecting data fragments from subsystem backend boards and can handle a maximal average throughput of 400 Gb/s. A key necessity is buffering, before event data is converted to standard TCP/IP streams and transferred to the surface data center. This buffering is essential to manage statistical fluctuations in Level-1 triggers and to accommodate situations where the total bandwidth of incoming backend links might exceed the outgoing D2S links. Advanced FPGAs such as Xilinx UltraScale+ devices with integrated High Bandwidth Memory are essential to efficiently store the large volumes of data before their aggregation and conversion to TCP/IP streams.

This chapter will first discuss what an FPGA is and its importance in these applications. Following this, two component of the FPGA, the Block RAM and the HBM will be presented, along with an overview of the AXI protocol used for communication with the latter.

4.1 Overview

A Field Programmable Gate Array (FPGA) is a reconfigurable integrated circuit. Unlike fixed function chips or microprocessors that execute software, an FPGA internal logic can be customized post manufacturing to implement any digital circuit. This flexibility makes FPGAs suitable for diverse applications requiring adaptable hardware.

An FPGA architecture usually is composed of Logic Blocks, which are the fundamental units containing programmable lookup tables (LUTs) for implementing boolean functions and flip-flops for data storage. It also includes the routing resources, a programmable interconnect network that links the logic blocks, enabling complex circuit configurations. I/O Blocks interface the FPGA internal logic with external pins, facilitating communication with other devices. Memory Elements, specifically Block RAMs, provide dedicated on chip memory blocks for efficient data storage. Additionally, DSP Slices are specialized hardware blocks optimized for high speed arithmetic operations, common in signal processing applications. Finally, Clock Management circuits are included for precise clock generation and distribution.

Logic blocks, commonly known as Configurable Logic Blocks (CLBs) generally comprises several logical cells and are interconnected by routing channels and surrounded by I/O pads. Each cell usually contains several LUT, D-type flip-flops, multiplexer and some arithmetic gates. For the Xilinx's VU35P FPGA (chosen for the DTH-400 board), there are different version of CLBs, but usually one contains 8 LUT, 16 D-type flip-flops, 1 carry gate, and 3 multiplexers. In figure

4.1 is shown the schematic of a CLB cell. There are more than 1.7M flip-flops and 870K lookup tables inside this FPGA.

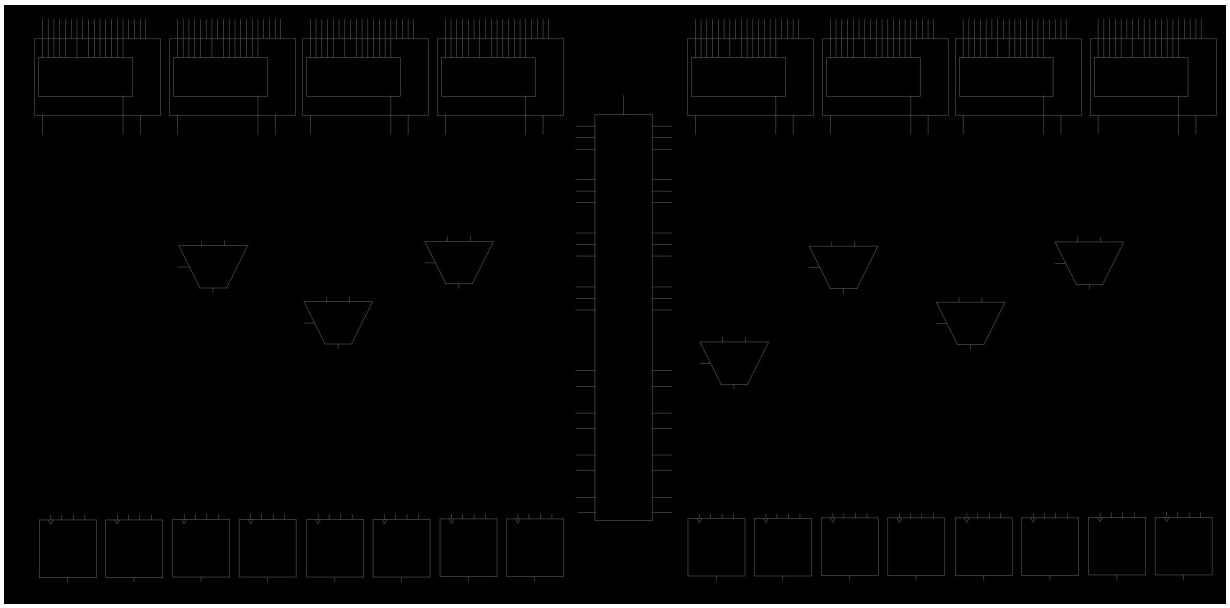

Figure 4.1: One of the possible CLB for the Xilinx's VU35P FPGA. The 8 LUT are the ones on the top, while the 16 D-type flip-flops are on the bottom. Between the two there are the multiplexers and the carry gate (the vertical component).

Along the CLB, there also blocks which include higher level functionality such as multipliers, generic DSP blocks, embedded memories and embedded processors. Compared to building them from logical primitives, having these common functions embedded in the circuit reduces the area required and gives those functions increased performance.

Most of the digital logic within an FPGA operates synchronously, necessitating a stable and precisely distributed clock signal. FPGAs incorporate dedicated global and regional routing networks specifically for clock and reset signals. To provide flexibility in clock generation and management, FPGAs typically include analog components such as Phase-Locked Loops or Delay-Locked Loops. These components are capable of synthesizing new clock frequencies from a reference clock, precise phase shifting, and managing jitter. Complex FPGA designs frequently utilize multiple clocks with varying frequency and phase relationships, each defining a separate clock domain. These clock signals can be sourced externally from an oscillator, generated internally by the FPGA clock management blocks, or recovered directly from incoming data streams. When transferring data between different clock domains, careful design of clock domain crossing circuitry is crucial to mitigate metastability, a condition where a flip-flop output enters an unpredictable state.

Designing a circuit for an FPGA begins with describing the digital logic using a Hardware Description Language (HDL) like VHDL or Verilog, which specifies the circuit behavior and structure. This HDL code then undergoes synthesis, a step where specialized tools translate the high level description into a low level netlist of generic logic gates and connections. Following synthesis, the process moves to mapping, where these generic logic elements are specifically assigned to the available resources within the chosen FPGA architecture, such as lookup tables and flip-flops. Next, during place and route, the mapped logic blocks are physically arranged onto the FPGA silicon fabric, and the routing resources are configured to create all the necessary connections between them. Finally, this physical layout and configuration information is compiled into a binary bitstream file. This bitstream is then loaded onto the FPGA chip, effectively programming its internal switches and lookup tables to form the custom digital

circuit.

FPGAs present significant advantages. Their inherent parallelism allows for the concurrent execution of numerous operations, providing significantly higher throughput and lower latency compared to sequential processing found in microprocessors, making them ideal for data intensive and realtime applications. The core benefit of an FPGA is its reconfigurability: unlike fixed function ASICs, an FPGA hardware design can be entirely modified and reprogrammed even after manufacturing. This enables rapid prototyping, iterative design improvements, and adaptation to changing requirements without incurring the time and cost of new fabrication.

4.1.1 Resourse usage of the CMS DAQ Design

Table 4.1 presents the resource utilization of the CMS DAQ Unit FPGA. Figure 4.2 then illustrates the physical placement of these resources on the FPGA device itself with the color legend. This design does not include the monitoring firmware (the subject of this thesis).

Resource	Utilization	Available	Utilization %
LUT	385835	871680	44.26
LUTRAM	568	403200	0.14
FF	571666	1743360	32.79
BRAM	676	1344	50.30
URAM	156	640	24.38
DSP	10	5952	0.17
IO	157	416	37.74
GT	47	64	73.44
BUFG	104	672	15.48
MMCM	3	8	37.50
PLL	1	16	6.25

Table 4.1: Resource utilization of the current design inside the DAQ Unit FPGA. [21].

4.2 BRAM

Block RAMs (BRAMs) are essential, dedicated memory areas built right into FPGAs. These storage blocks are designed to store data quickly on the chip. Unlike memory made from the general building blocks of the FPGA (distributed RAM), Block RAMs are much better suited for memory tasks. They can hold more data in a smaller space, allow faster access to that data, and use less power. These memory blocks are quite flexible. They can be set up to have different widths and depths. They often support various ways of working, like acting as a single port memory, a dual port memory, or a FIFO buffer. This makes them very useful for many things, such as temporarily holding data, creating lookup tables for quick reference, or storing information for complicated calculations. Because Block RAMs are placed directly on the FPGA chip, they help avoid delays that would happen if data had to travel through many general connections. This direct placement allows for very fast data transfer and predictable access times. Usually the latency is 1 or 2 clock cycles depending on the option chosen.

In case of the VU35P FPGA there are 1344 Block RAM Blocks scattered on the device for a total of 47.3 Mb of dedicated storage, compared to the maximum of 24.6 Mb (using LUTRAM) of the distributed RAM. Each BRAM block can be configured as two independent 18 Kb blocks, or a single 36 Kb block RAM. These blocks can also be configured as FIFO queue, in figure 4.3 are shown the possible combinations of configurations of a 36Kb block of BRAM.

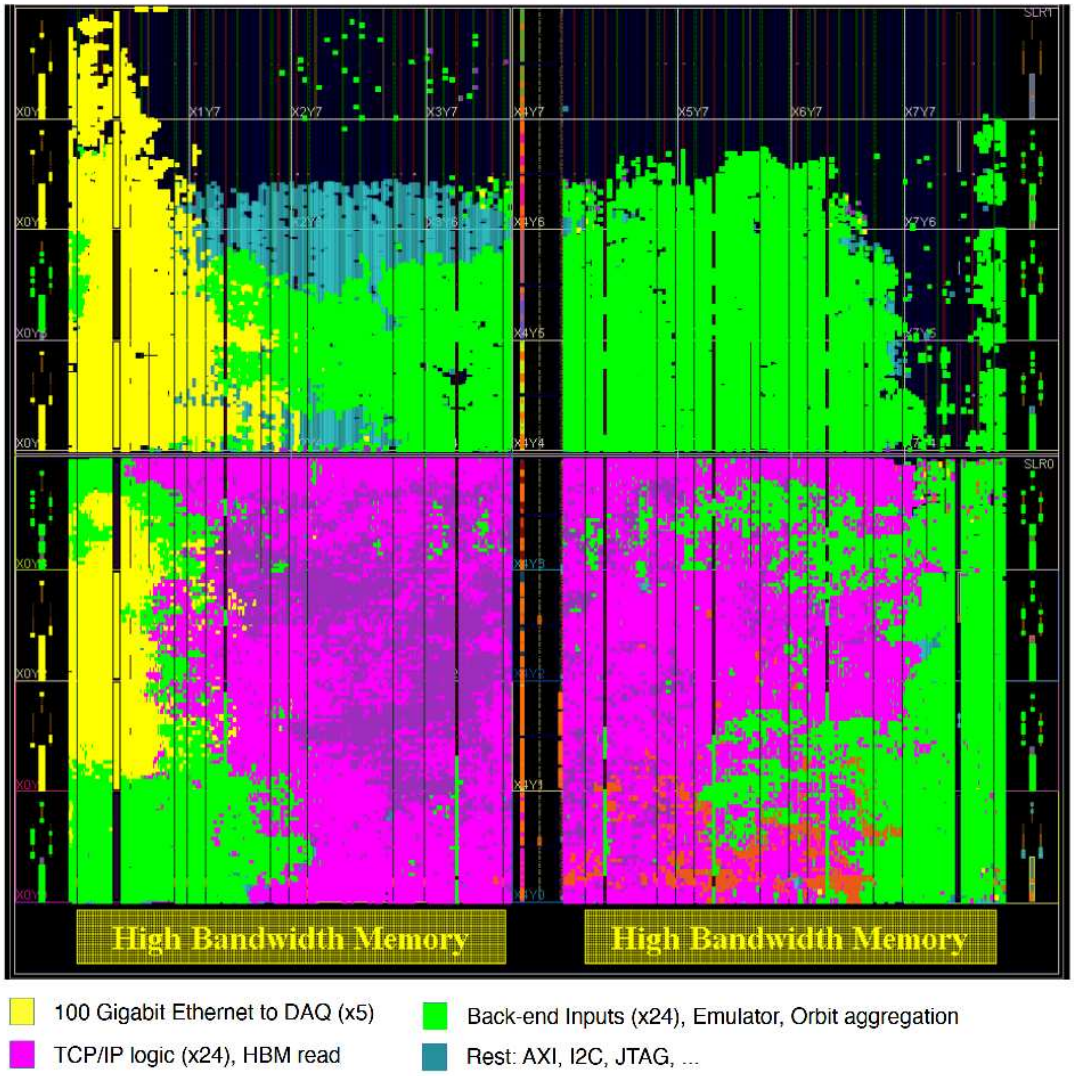

Figure 4.2: Physical placement of the resources on the FPGA [21].

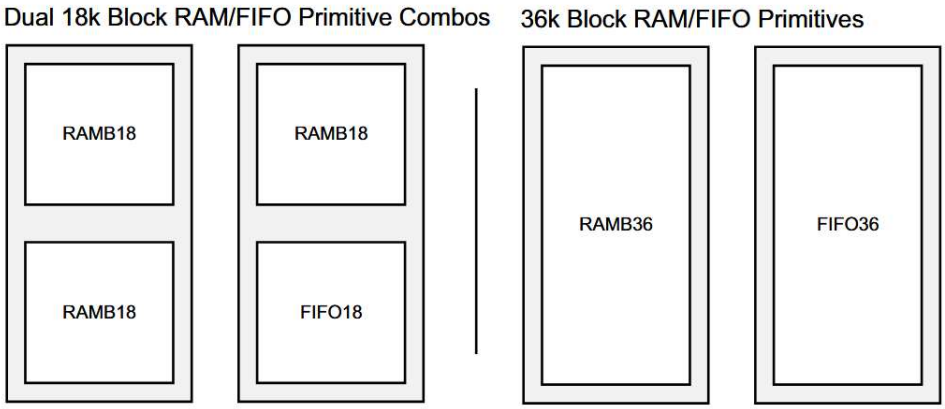

Figure 4.3: Combinations of Block RAM and FIFO for a single 36 Kb BRAM Block.

4.3 HBM

The VU35P FPGA contains 8GB of high performance memory. This memory, called HBM (High Bandwitdh Memory), allows bandwidth up to 400 GB/s while keeping the power consumption low. This storage is crucial to buffer the data for the various TCP/IP output streams in case of congestion or retransmission. This type of memory is generally found only in high-end

FPGAs, where achieving very high performance is a critical requirement.

The HBM inside the VU35P FPGA has 8GB of memory divided in two stacks. Each stack is split into eight independent memory channels. Each channel has two inputs and two outputs, each 256 bits wide, that can be clocked up to 400 MHz simultaneously. Pseudo channel memory access is limited to its own section of the memory (1/16 of the stack capacity or 1/32 of the full memory capacity). In figure 4.4 is shown the HBM configuration. Since only 24 of the 32 pseudo channels are used by the CMS DAQ team (each associated with an incoming optical link), 8 are left unused and can be utilized for other purposes, like monitoring and debugging the incoming data. Indeed, one of the free pseudo channel is used for the monitoring application of this thesis.

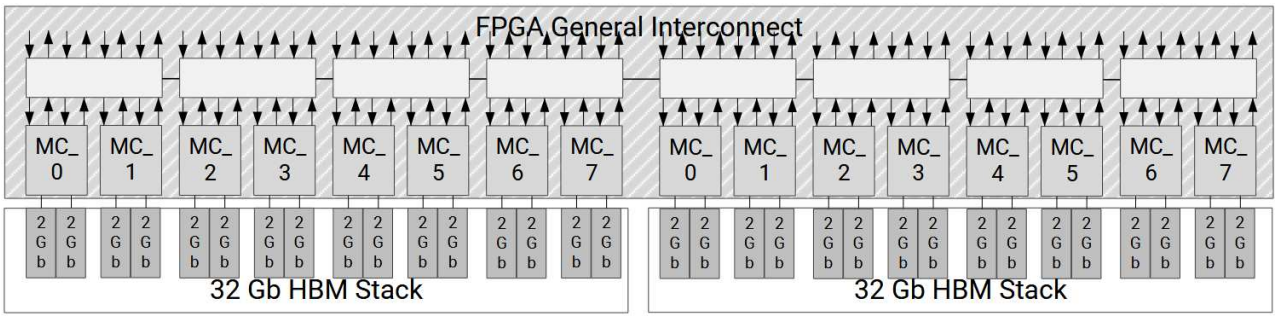

Figure 4.4: HBM Two Stack Configuration. There are 32 pseudo channel of 256MB (32Gb = 4GB and 2Gb = 256MB).

Every interaction with the memory is done with the AXI3 standard protocol. In total there are 32 AXI stacks with a throughput capacity equal to 1/32 of the total HBM bandwidth. Each port can optionally address the entire HBM space (global addressing) to greatly reduce the need for any cross channel routing in general interconnect. Alternatively, non-global addressing (direct addressing) limits an AXI port to the associated pseudo channel with the least latency.

An HBM word is 256 bits wide, and working with smaller chunks for I/O operations is not possible. This is not an issue when reading, as any unwanted bits can simply be discarded. However, writing smaller amounts of data without overwriting existing information can be a problem. Fortunately, the HBM allows specification of exactly which bytes (out of the 32 possible in a 256-bit word) contain valid data. This means writing only to the intended bytes is possible, preventing accidental overwrites across the entire 32-byte word.

The HBM is optimized for high-bandwidth sequential data transfers, particularly long bursts. While exhibiting high access latency, this is offset by its ability to stream data continuously. The overhead associated with AXI protocol setup and of the internal machinery renders this memory suboptimal for random access of small data quantities. Nevertheless, for the scope of this thesis, the HBM is utilized as in the second case, primarily due to the large available memory and the not so strict time requirements of this application.

4.3.1 The AXI Protocol

The Advanced eXtensible Interface protocol[1], facilitates on chip communication for interacting with different systems or IP core like the HBM.

The AXI protocol is transactions-based and defines five independent channels: Read Address (AR), Read Data (R), Write Address (AW), Write Data (W), and Write Response (B). A request channel carries control information that describes the nature of the data to be transferred. This is known as a request. The data is transferred between Manager and Subordinate using either:

[1] https://developer.arm.com/documentation/ihi0022/latest/

- A write data channel to transfer data from the Manager to the Subordinate. In a write transaction, the Subordinate uses the write response channel to signal the completion of the transfer to the Manager.
- A read data channel to transfer data from the Subordinate to the Manager.

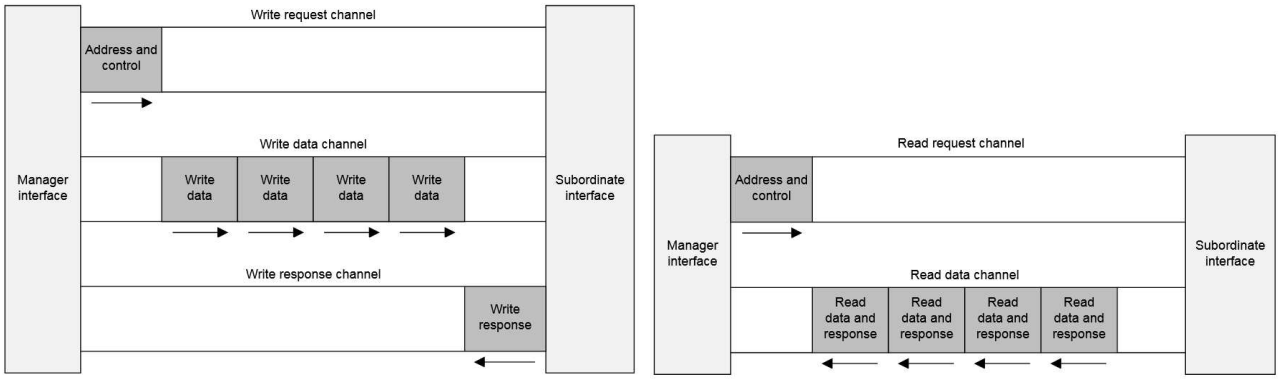

Figure 4.5: On the left a diagram of an AXI write transaction, on the right a diagram of an AXI read transaction.

Each of the five independent channels consists of a set of information signals and VALID and READY signals that provide a two-way handshake mechanism. The information source uses the VALID signal to show when valid address, data, or control information is available on the channel. The destination uses the READY signal to show when it can accept the information. Both the read data channel and the write data channel also include a LAST signal to indicate the transfer of the final data item in a transaction.

For controlling the HBM only a subset of the full AXI protocol is needed.

Write Transaction

To initiate a write transaction, the manager asserts the following signals:

- **AWADDR**: Specifies the memory address to be accessed (it is recommended to be word-aligned).
- **AWSIZE**: Specifies the number of bytes to transfer. For HBM, this must be set to 0b101, indicating 32 bytes.
- **AWLEN**: Specifies the number of transfers within the transaction. The total number of transfer is AWLEN + 1.
- **AWBURST**: Specifies the burst mode. It is typically set to 0b01 for INCR mode, where the address for each subsequent transfer is an increment of the previous transfer address.
- **AWVALID**: Indicates to the subordinate the initiation of a write transaction and the validity of the previous signals.

Whenever the subordinate is ready to accept a write transaction, it signals its readiness to the manager by asserting **AWREADY**. When both **AWVALID** and **AWREADY** are asserted simultaneously, both the manager and the subordinate confirm that a write transaction has started.

Now, the manager must provide the data to be written using the following signals:

- **WDATA**: Carries the data to be written into the memory.
- **WVALID**: Specifies to the subordinate that the **WDATA** signal currently holds valid data.

- **WLAST**: Specifies to the subordinate that the current transfer is the final one.

Whenever the subordinate is ready to accept the data, it communicates its readiness to the manager by asserting **WREADY**. When both **WVALID** and **WREADY** are asserted simultaneously, both the manager and the subordinate acknowledge that the data has been received. This cycle repeats for a total of **AWLEN** + 1 times. During the last transfer, the manager must assert the **WLAST** signal.

Finally, the subordinate must inform the manager of the write transaction's outcome using the following signals:

- **BRESP**: Specifies the result of the write operation.
- **BVALID**: Indicates to the manager that the **BRESP** signal currently carries a valid result.

Whenever the manager is prepared to accept the result, it communicates its readiness to the subordinate by asserting **BREADY**. When both **BVALID** and **BREADY** are asserted simultaneously, both the manager and the subordinate acknowledge that the result of the write transaction has been successfully received. This action concludes a write transaction. A generic diagram for a write transaction is shown in figure 4.6.

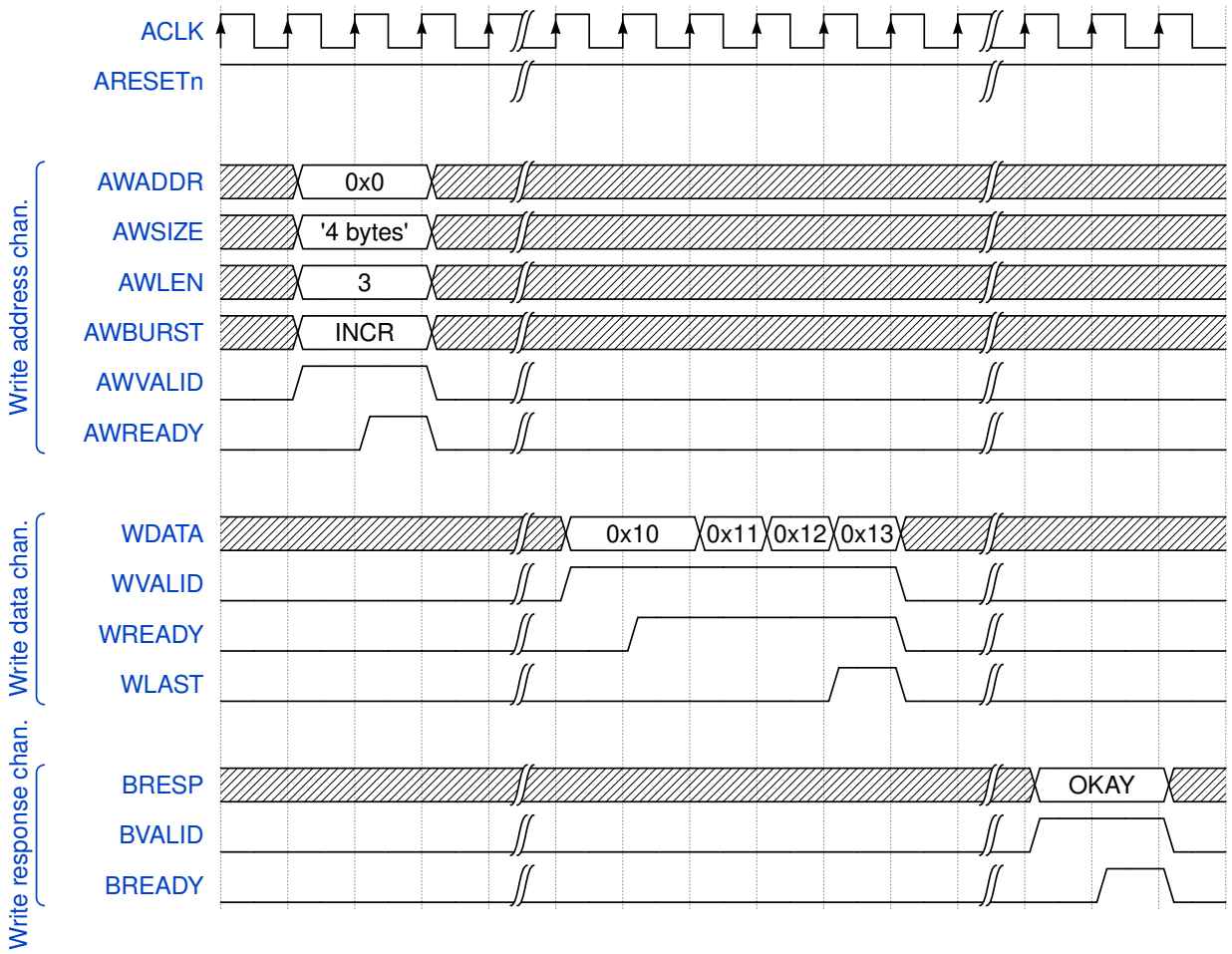

Figure 4.6: Generic example of an AXI write transaction. The manager drives 4 beats of 4 bytes each starting from address 0x0 with INCR type, writing 0x10 for address 0x0, 0x11 for address 0x4, 0x12 for address 0x8 and 0x13 for address 0xc. The subordinate returns 'OKAY' as write response for the whole transaction.

Read Transaction

To initiate a read transaction, the manager asserts the following signals:

- **ARADDR**: Specifies the memory address to be accessed (it is recommended to be word-aligned).
- **ARSIZE**: Specifies the number of bytes to transfer. For HBM, this must be set to 0b101, indicating 32 bytes.
- **ARLEN**: Specifies the number of transfers within the transaction. The total number of transfer is ARLEN + 1.
- **ARBURST**: Specifies the burst mode. It is typically set to 0b01 for INCR mode, where the address for each subsequent transfer is an increment of the previous transfer's address.
- **ARVALID**: Indicates to the subordinate the initiation of a read transaction and the validity of the previous signals.

Whenever the subordinate is ready to accept a read transaction, it signals its readiness to the manager by asserting **ARREADY**. When both **ARVALID** and **ARREADY** are asserted simultaneously, both the manager and the subordinate confirm that a read transaction has started.

Now, the subordinate must provide the data read from the memory using the following signals:

- **RDATA**: Carries the data read from the memory.
- **RRESP**: Specifies the outcome of the read transaction.
- **RVALID**: Specifies to the manager that the **RDATA** and **RRESP** signals currently hold valid data.
- **RLAST**: Specifies to the manager that the current transfer is the final one.

Whenever the manager is ready to accept the data, it communicates its readiness to the subordinate by asserting **RREADY**. When both **RVALID** and **RREADY** are asserted simultaneously, both the manager and the subordinate acknowledge that the data has been received. This cycle repeats for a total of **ARLEN** + 1 times. During the last transfer, the subordinate must assert the **RLAST** signal. This concludes a read transaction. A generic diagram for a read transaction is shown in figure 4.7.

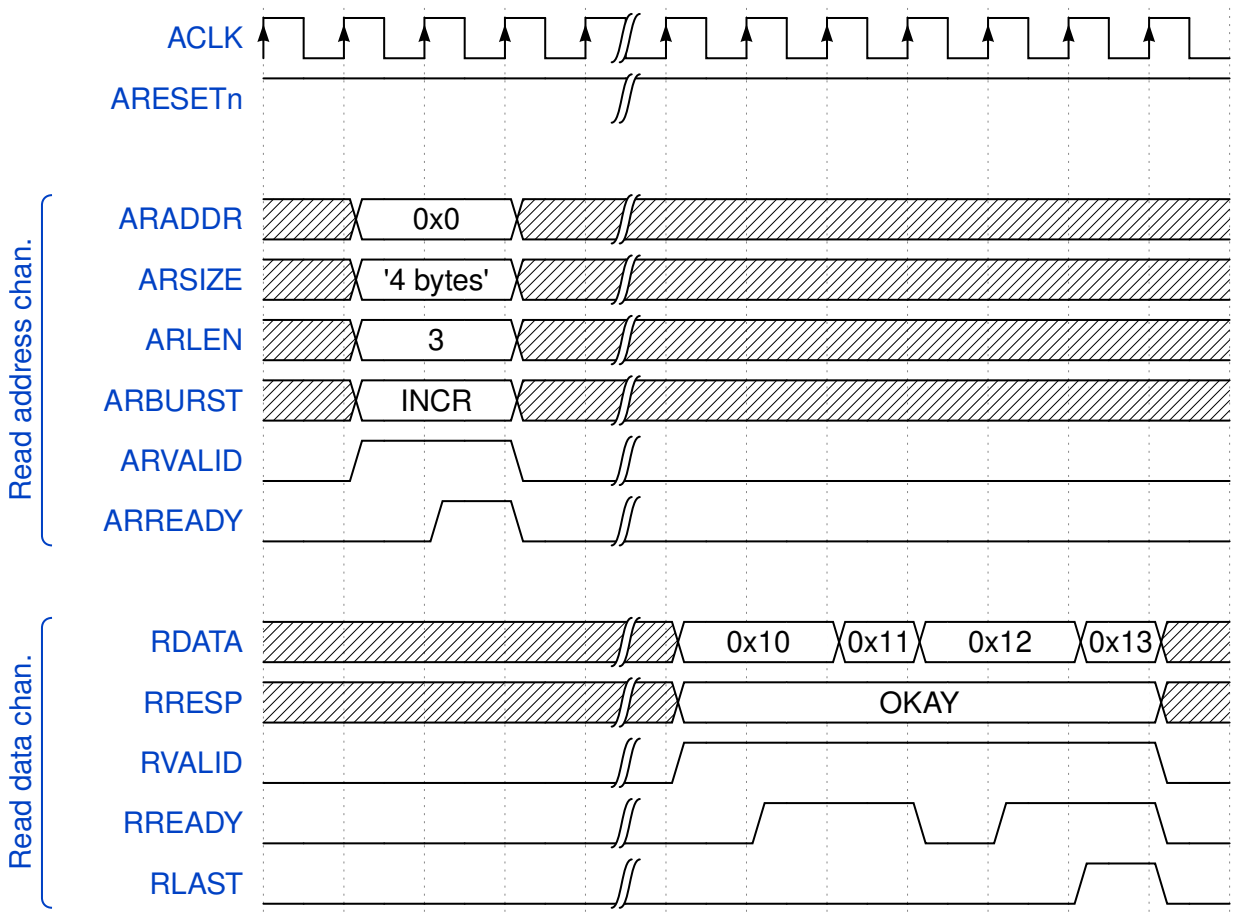

Figure 4.7: Generic example of an AXI read transaction. The manager requests a 4-beats of 4 Bytes each starting from address 0x0 with INCR type. The read data is 0x10 for address 0x0, 0x11 for address 0x4, 0x12 for address 0x8 and 0x13 for address 0xc.

Chapter 5

DTHistogram - Overview

After setting the stage explaining all the technologies and systems, this chapter is about the work done during the internship. The project name is DTHistogram, from the word DTH-400, the new DAQ board, and the word histogram.

5.1 Overview

The DTHistogram firmware enables the creation, updating, and storage of histograms to monitor data streams arriving at the DTH-400 board. Some or all the incoming data are fields from the orbit packets described in section 3.2.1. While the data stored in the histograms typically comprises specific fields from this protocol, any type of data is accepted. To store all the information of the histograms, one of the free subchannel of the HBM is used (256 MB).

As this monitoring is not a critical component of the CMS DAQ, the firmware is designed not to interfere with the existing data flow. Specifically, the DTH-400's existing firmware sends data to DTHistogram for processing without any flow control. A FIFO queue is implemented to buffer potential fluctuations in the incoming data rate. However, if this queue overflows, data is simply dropped. This monitoring system is constructed to avoid slowing down the normal data flow.

Since 24 optical links feed into the DTH-400, histograms are organized and grouped by these links, referred to as "units". For each unit (i.e. optical link), multiple histograms can exist, each associated with different fields within the orbit packet (which aggregates the SLinkRocket packets for a single orbit), or other observables. Data originating from different links (and of different observables) is not aggregated in any way.

When data becomes available, it must be placed into the appropriate bin of the correct histogram. Each histogram has its own unique configuration stored in memory. The system identifies the unit and its data type to retrieve the corresponding histogram configuration. With this configuration, the bin index is calculated, and its counter is incremented by one. This process is performed sequentially for each piece of incoming data; there is no parallel data handling for this specific operation.

The design employs two distinct clocks, establishing two separate clock domains within the system. One clock, `usr_clk`, is given from the interface with the full CMS DAQ design and operates at 100 MHz. This clock is utilized for incoming data paths and in all modules communicating with the Zynq processor, as the chip-to-chip communication protocol operates with this

clock. The second clock, `axi_clk`, runs at 250 MHz and is employed for HBM communication via the AXI protocol and for the majority of the DTHistogram design internal logic.

To retrieve the data of each histograms, change the configurations or perform a reset, a Python library has been developed, which is discussed in appendix B.

The following sections and the following chapter will explain in depth the architecture of the firmware.

5.2 An Histogram

To begin, it's important to define how histograms are characterized within the firmware. Each histogram is defined by three key parameters:

- **Minimum Edge (ME)**: This is the lowest value included in the very first bin of the histogram.
- **Bin Width (BW)**: This specifies the width of a single bin, and this width remains constant for all bins (in that particular histogram).
- **Bins Number (BN)**: This is the total count of bins the histogram will contain.

These three numbers together form an histogram configuration, allowing for the calculation of every bin edge. Specifically, the i-th bin will include values within the interval `[ME + i * BW, ME + (i + 1) * BW - 1]`. For the entire histogram, the minimum value (inclusive) is `ME` and the maximum value (inclusive) is `ME + BW * BN - 1`. Beyond the `BN` standard bins, each histogram also includes two additional bins: an underflow bin and an overflow bin. Values that are lower than `ME` will be placed into the underflow bin, while values equal to or higher than `ME + BW * BN` will be placed into the overflow bin. In table 5.1 are shown the type of the parameters and their ranges. For the number of bins, the range goes up to $2^{12} - 2$ because the underflow and overflow also need to be indexed. In figure 5.1, instead, is shown an example of an histogram with the three parameter composing its configuration.

Parameter	Type	Range
`Minimum Edge (ME)`	`32 bits unsigned integer`	`0` to $2^{32} - 1$
`Bin Width (BW)`	`32 bits unsigned integer`	`1` to $2^{32} - 1$
`Bins Number (BN)`	`12 bits unsigned integer`	`1` to $2^{12} - 2$

Table 5.1: The three parameters composing the configuration of an histogram with their types and ranges.

5.3 Memory Layout

For every histogram, its configuration and all of its bin counters must be stored in the High Bandwidth Memory (explained in Section 4.3). In this context, "storing a bin" specifically refers to storing the counter associated with that bin, which represents the number of times incoming data fell within that bin defined range. One reason HBM is chosen for this storage, over other memory types, is the availability of numerous unused subchannels, each capable of storing up to 256MB. The more crucial reason, however, is that utilizing Block RAM or the scarcer distributed RAM would significantly increase resource usage. This higher usage could complicate the placement of the entire design (which includes the DAQ system) and potentially lead to timing failures. The DTHistogram monitoring firmware is designed to be lightweight and have minimal impact, making the unused storage of the HBM the ideal choice. As shown

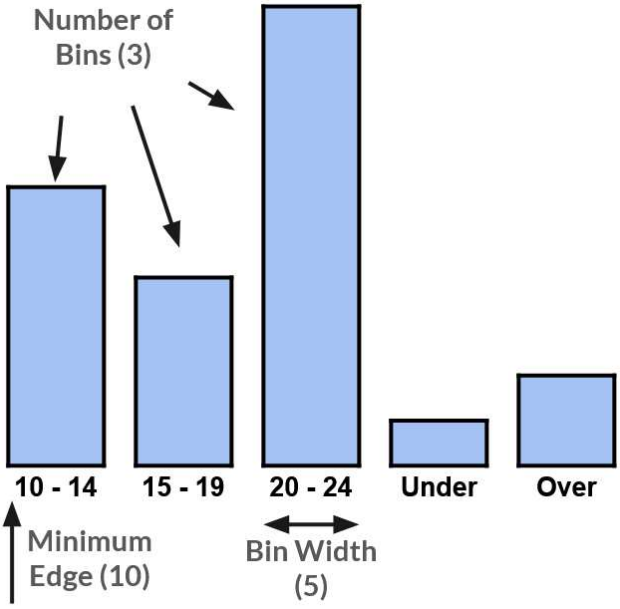

Figure 5.1: An example of an histogram and its configuration.

later, block RAM is utilized to cache the configurations of all histograms, which constitutes only a fraction of the total storage required.

All the information for a single histogram, including its configuration and bin counters, are stored contiguously. Following this, the information for the second histogram is stored, then the third, and so on for all histograms, one after another. For any given histogram, its configuration is stored first, occupying a single HBM word (256 bits or 32 bytes). There is lot of space within a word for the three parameters, leaving most of it empty, as depicted in Figure 5.2.

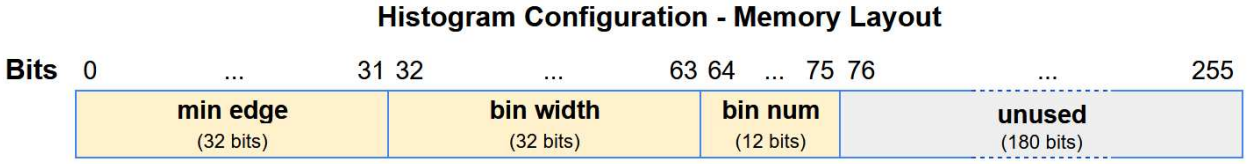

Figure 5.2: The configuration for a single histogram is stored in memory as a single word of 32 bytes. Since only $32 + 32 + 12 = 76$ bits are needed, 180 are unused.

Following the configuration, the bins are stored sequentially, from index 0 to the last one with index $2^{12} - 3$. Regardless of the number of bins specified in the configuration, the maximum possible space for all bins is always allocated. This simplifies memory management significantly, as the configuration and thus the total number of bins can change during runtime, though it is less memory efficient. Each bin counter is a 64-bit unsigned integer, allowing four bins to be stored within a single HBM word. The underflow bin is located at index `BN` and the overflow bin at position `BN + 1`. Should the value of `BN` change, their positions will adjust accordingly. Figure 5.3 illustrates a memory layout for the case where `BN =` 13 (figure a) and for the case where `BN =` $2^{12} - 2$ (figure b). It can be seen that if `BN` is low, a significant amount of space will remain unused.

Considering both its configuration and its bin counters, the total memory utilized for storing a single histogram is:

$$32\ \text{B} + \left(8\ \frac{\text{bytes}}{\text{bin}}\right) \cdot \left(2^{12}\ \text{bin}\right) = 32\ \text{KB} + 32\ \text{B} = 1025\ \text{words} \tag{5.1}$$

Histogram - Memory Layout (Bins Number = 13)

Bytes	0 ... 7	8 ... 15	16 ... 23	24 ... 31
0x400	- unused -	- unused -	- unused -	- unused -
...	...	...	...	...
0x80	Bin 12	Underflow Bin	Overflow Bin	- unused -
0x60	Bin 8	Bin 9	Bin 10	Bin 11
0x40	Bin 4	Bin 5	Bin 6	Bin 7
0x20	Bin 0	Bin 1	Bin 2	Bin 3
0x00	Configuration			

(a)

Histogram - Memory Layout (Bins Number = 2^12 - 2)

Bytes	0 ... 7	8 ... 15	16 ... 23	24 ... 31
0x400	Bin 4092	Bin 4093	Underflow Bin	Overflow Bin
...	...	...	...	...
0x60	Bin 8	Bin 9	Bin 10	Bin 11
0x40	Bin 4	Bin 5	Bin 6	Bin 7
0x20	Bin 0	Bin 1	Bin 2	Bin 3
0x00	Configuration			

(b)

Figure 5.3: Memory layout for two histograms with number of bins 13 and $2^{12}-2$ and respectively. The bins for a single histogram are stored one after another, four for each word. At the end the underflow and the overflow bins are stored. The addressing starts in the bottom left.

where the first 32 bytes are for the configuration and the rest for the bins.

Considering there are 24 units and assuming each unit has around 5 histograms (associated with the different observables), then the total memory used, by rounding to the nearest power of 2, is:

$$\left(\frac{32\text{ KB} + 32\text{ B}}{\text{histogram}}\right) \cdot \left(2^3\ \frac{\text{histogram}}{\text{unit}}\right) \cdot \left(2^5\ \text{unit}\right) = 8200\text{ KB} \approx 8\text{ MB} \tag{5.2}$$

Given that 256MB of memory is available, there is plenty of space remaining. Therefore, even with a significant portion of memory unused, there's no need to achieve higher efficiency at the cost of simplicity. In figure 5.4 the layout of the memory with different histograms is shown.

5.4 Data Flow

This section details the data flow within the firmware, from the initial incoming data to the subsequent increment of histogram bins stored into the HBM.

5.4.1 Initialization

Following a reset or power-on, the firmware must read the configurations of all histograms from memory. As these configurations are written externally, their parameters are not sanitized, meaning some values could be out of bounds. For each histogram, after its configuration is retrieved from memory, each parameter is validated and stored into the Block RAM. If any parameter is found to be out of bounds, a default, valid value will be assigned, but the value

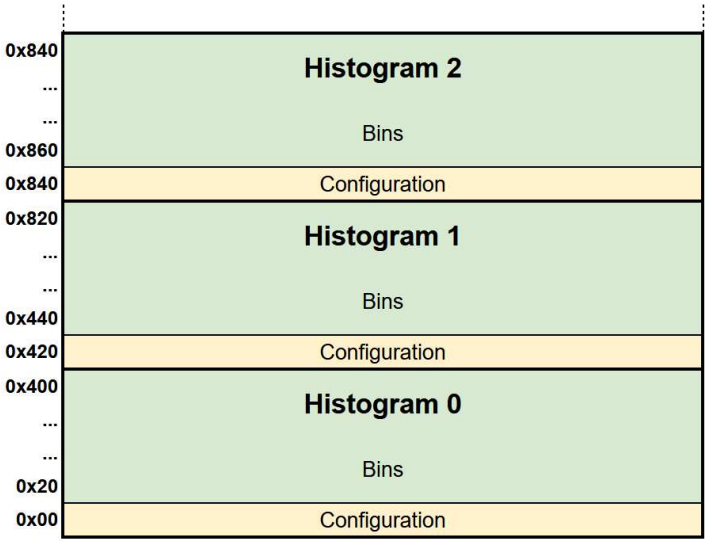

Figure 5.4: Memory layout of different histograms saved into the HBM.

in memory itself will remain unchanged, only the value stored in the Block RAM is updated. The Block RAM is used for this purpose to enable faster storage and retrieval of these values compared to HBM, as detailed in Chapter 4.

In addition to the three configuration parameters, the **Maximum Edge** is also stored in the Block RAM. Its value is calculated as `Maximum Edge = ME + BW * BN`, representing the first value beyond the histogram (right) range, or equivalently, the smallest value within the overflow bin. This value is useful for quickly checking if a piece of data falls outside the histogram defined bounds (together with the minimum edge `ME`). It is computed and stored upfront to avoid recalculating it for every data point.

After reading the configuration and storing it into the BRAM, `BN + 2` bins of the current histogram are set to zero within the HBM, thereby initializing their counters. It is important to note that not all the 2^{12} bins are cleared, only those utilized by the current configuration, as specified by the Bins Number (`BN`) field, are initialized. For example, if `BN = 13` as illustrated in figure 5.3.a, only the first 15 bins are set to zero, since the remaining ones are not used.

These steps are repeated sequentially for all histograms. Once completed, the firmware is ready to process incoming data. Any data that arrives before this initialization is complete, is stored in a FIFO buffer.

5.4.2 Data Processing

Given the 24 distinct input lines, determining the unit associated with a piece of data is straightforward. This is because each input optical line corresponds directly to a unit, which in turn represents a distinct signal within the design. However, the specific histogram within that unit associated with the data is not known. This is why each piece of incoming data has an associated histogram identifier (given together with the data).

When incoming data arrives, it is placed into a unit specific FIFO along with its histogram identifier. There is no flow control at this stage: if the FIFO is full, the data is simply dropped. This is the only point in the architecture where data can be lost. Within a single unit, only one piece of data can arrive per clock cycle. However, data from different units (i.e., different lines) can arrive simultaneously in the same clock cycle, as the described hardware is replicated for each unit. This means there are 24 FIFOs all filling up with data concurrently. A diagram of this design is shown in figure 5.5.

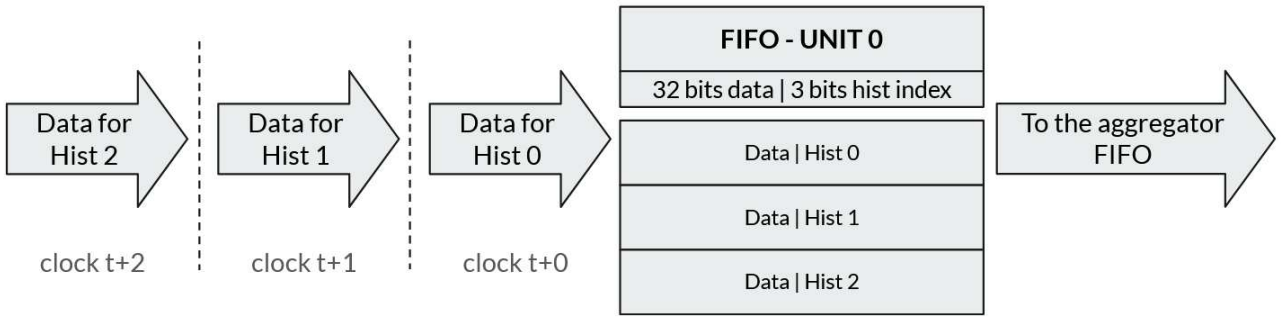

Figure 5.5: Data flow specific to a single unit. The data arrives containing the identifier of the histogram associated, and are both stored in a unit specific FIFO. Here there is no flow control, if the FIFO is full then data is lost.

At each clock cycle, a unit is selected in a round-robin fashion. If data is present in that unit FIFO, it is pulled out and placed into another FIFO, referred to as the "aggregator FIFO" (if is not full). Data from all units converges here, awaiting to be processed into the correct bin of the correct histogram. Even though the data is together, it is not aggregated, the histogram identifier remains, and a new unit identifier is added. This ensures each piece of data can be correctly associated with its right histogram, despite all being within the same FIFO. In this specific case, flow control is implemented, meaning no data is dropped. A diagram of this design is presented in figure 5.6.

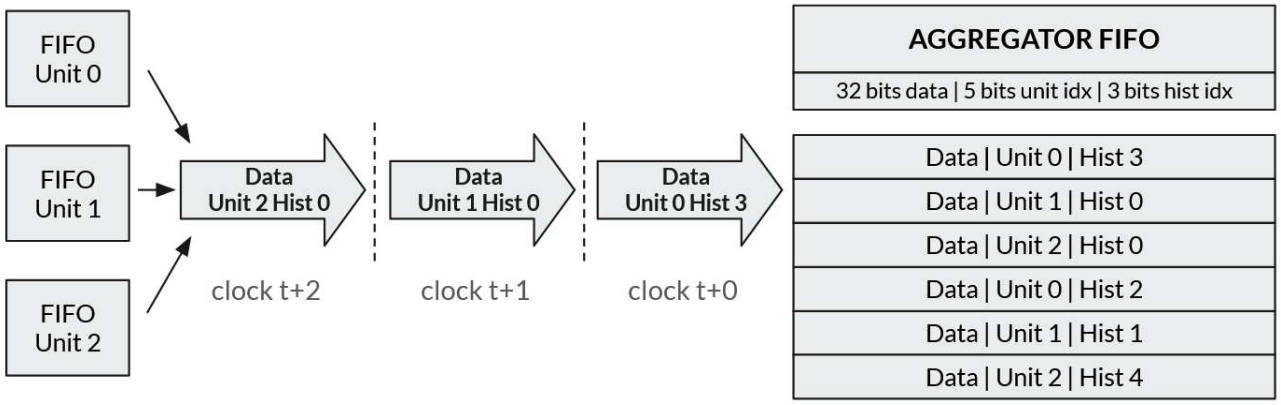

Figure 5.6: Data flow between the unit specific FIFO and the aggregator FIFO. The data is taken out in a round robin way and sent to the aggregator FIFO with a new identifier of the unit. Here there is flow control, no data is dropped.

Data is now pulled one at a time from the aggregator FIFO, along with its unit ID and histogram ID, to be placed into the correct bin of the correct histogram. After extracting a data item from the FIFO, an address is constructed using these two identifiers (which uniquely identify an histogram). This address is then used to index the correct configuration within the BRAM. From the BRAM, the three configuration parameters (minimum edge, bin width, and bins number), along with the maximum edge, are extracted, as shown in figure 5.7.

The data item is first compared against the minimum and maximum edge to determine if it is out of bounds. If it is, the correct bin (either the underflow or the overflow bin) has been identified. If it is not out of bounds, a division must be performed. Specifically, given some data d, the index i of the bin to which d belongs is:

$$i = \left\lfloor \frac{d - (\text{minimum edge})}{(\text{bin width})} \right\rfloor \tag{5.3}$$

Given that divisions are not trivial operations in an FPGA, a custom division algorithm has been

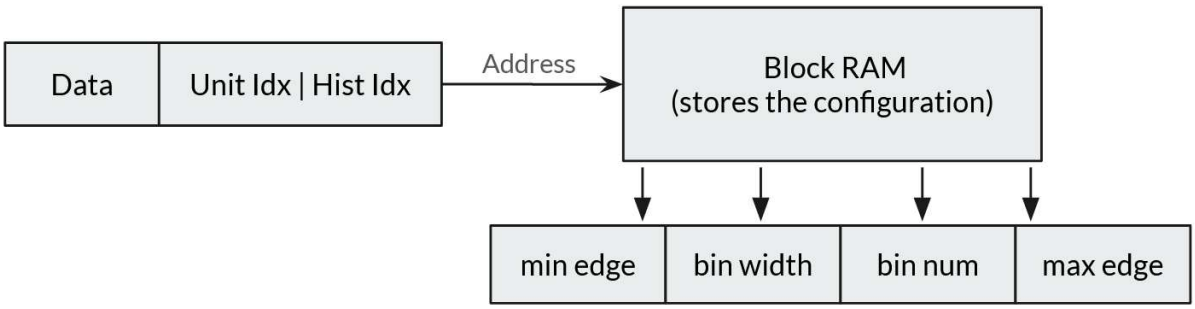

Figure 5.7: By using the unit index and the histogram index, an unique address is composed. This address is then used to retrieve the corresponding configuration (plus the value of max edge) from the block RAM.

implemented. This algorithm leverages the restricted range of the expected result: although the data d, minimum edge, and bin width are all 32-bit numbers, the final result can always be stored within 12 bits. This is because the result represents the bin index, which is always less than 2^{12} (as the data has already been checked to be within the histogram bounds at this stage). By calculating one bit of the result per clock cycle, the final result is ready in 12 clock cycles (plus some constant cycles).

Now that the index of the correct bin has been determined, the actual counter for that bin must be read from the HBM. This counter is then incremented and sent back to be written. Following this step, the data is considered consumed, the system becomes free, and a new data item can be pulled from the aggregator FIFO (if there is any), restarting the entire process.

Communication with external systems is facilitated by the Zynq processor, which interfaces the FPGA with a Python wrapper. Internally, this Python wrapper utilizes the HAL library to communicate with the FPGA. Underneath this, read and write requests are converted into AXI transactions and sent to the FPGA, and vice versa. After establishing an SSH connection to the Zynq, it is possible to retrieve all bin counters, edit configurations, or initiate a reset. In this setup, the FPGA acts as a subordinate to the processor, which serves as a manager, translating requests received from the Python wrapper.

The entity must interface with the HBM to handle these requests. Consequently, the HBM read and write buses (which are separate) are shared between this entity and the entity responsible for updating the bins. A coordinator is positioned in front of the HBM to manage which of these two entities can utilize the read or write bus.

Chapter 6

DTHistogram - HDL Design

To develop the firmware, a Hardware Description Language (HDL) is essential. The HDL, in this case VHDL, precisely describes the circuit behavior and structure. Subsequently, using standard design tools, the HDL code is synthesized into a netlist, which is then placed and routed to create the integrated circuit on the FPGA.

In the following section, without directly presenting the code, the behavior of the entities comprising the DTHistograms will be described, aiming to closely reflect the HDL used for their implementation.

6.1 Entities

This section first provides a brief hierarchical overview of the entities, followed by a more in-depth guide.

- **DTHistogram Entity (DTHE):** it serves to interface the DTHistogram with the full design, gathering all connections, including those to the HBM, to the chip-to-chip interface (explained in section 6.2), the data for the histograms and reset signals.
  - **Histogram Wrapper Entity (HWE):** it contains both the unit FIFOs and the aggregator FIFO, along with the logic responsible for delivering data to the histograms.
    - **Histogram Entity (HE):** it calculates the correct bin from the incoming data, and also handles bin initialization and configuration sanitization. It is memory agnostic, communicating with the entity HME (below) via a custom protocol explained later in section 6.3. It also include the BRAM for caching the configuration.
      - **Division Entity:** It performs divisions between two 32-bit numbers, with the knowledge that the result will always fit within 12 bits.
    - **Histogram Memory Entity (HME):** This entity interfaces with the HBM to perform operations on the stored histograms. Commands are sent by the entity HE via the custom protocol.
  - **HBM Access Control Entity (HACE):** it fulfills requests originating from the Python library using the chip-to-chip interface, enabling operations such as updating configurations and accessing bin counters.

- **AXI Interconnector Entity (AIE):** it shares the bus connected to the HBM between the HE and HACE entities.

To test the DTHistogram design, the complete system with all its components is necessary. However, since developing with a full scale setup is impractical and lacks the external connections, an alternative entity was conceived. This entity, called the **Top Entity**, incorporates components already present in the CMS DAQ Design that should not be replicated within the DTHistogram, such as the HBM and the chip-to-chip interface. It also generates mock data to feed into the histograms.

The entities are now presented using a bottom-up approach, starting with individual, self contained components and progressively moving towards the more intricate entities. It is done this way so the reader can build up understanding incrementally. In appendix A, the schematics and code for the Division entity are presented as an example.

6.1.1 Division Entity

This entity is the one who performs the division to calculate the index of the bin given a piece of data and the bin width, respectively the dividend and the divisor, or the numerator and denominator.

Definition

Port	Mode	Bits	Description
`clk`	`IN`	`1`	Clock
`rstn`	`IN`	`1`	Reset active low
`enable`	`IN`	`1`	Starts the division process
`num`	`IN`	`32`	The dividend or the numerator
`den`	`IN`	`32`	The divisor or the denominator
`res_valid`	`OUT`	`1`	Asserted when the result is valid
`res`	`OUT`	`12`	The value of $\lfloor$`num/den`$\rfloor$

Description

When `enable` is set to 1 for a clock cycle, the value of `num` and `den` are read and stored internally. After the result is calculated, the `res_valid` signal is asserted for one clock cycle and `res` contains the result of the operation. If `rstn` is deasserted then the entity is reset.

The algorithm first checks if `num` is 0. If so, `res` is set to 0 and `res_valid` is asserted, implying that $\frac{0}{n} = 0$ with $n \in [0, 2^{32} - 1]$. Since the denominator, the bin width, cannot be 0, it is always correct. Otherwise, the division proceeds by calculating the bits of the result from the most significant bit (MSB) down to the least significant bit (LSB), with all bits of `res` initially set to 0. The MSB of `res`, corresponding to a value of 2^{11}, is set to 1 if `den` $\cdot\, 2^{11} \leq$ `num`. After that, the MSB-1 bit is calculated, which has a value of 2^{10}. In this case we also need to consider the actual value of `res` and subtract it from the value of `num` after multiplying it by `den`. So the MSB-1 of `res` is set to 1 only if `den` $\cdot\, 2^{10} \leq$ `num` - `res` $\cdot$ `den` and so on. In general:

$$\texttt{res[i]} = \begin{cases} 1 & \texttt{den} \cdot 2^i \leq \texttt{num} - \texttt{res} \cdot \texttt{den} \\ 0 & \text{else} \end{cases} \qquad \text{with } i = 11, 10, ..., 1, 0 \quad \text{(sequentially)}$$

where `res[i]` indicate its $i-th$ bit, with `res` initialized to 0. After `res[0]` is calculated, the `res_valid` is asserted for one clock cycle and `res` contains the results of the operation.

This works because the result cannot overflow `res`.

6.1.2 Histogram Entity

This entity determines the correct bin for a given data sample and increments it. It is also responsible for clearing bins upon startup or after a reset, and for storing the configuration in Block RAM.

Definition

Port	Mode	Bits	Description
`clk`	`IN`	`1`	clock
`rstn`	`IN`	`1`	Reset active low
`hist_rst`	`IN`	`1`	To request a reset of a specific histogram
`hist_rst_idx`	`IN`	`8`	The index of the histogram to reset
`ready`	`OUT`	`1`	Entity ready to accept data
`data_valid`	`IN`	`1`	Valid data on the port `data` and `data_idx`
`data_idx`	`IN`	`8`	The sub histogram index associated with `data`
`data`	`IN`	`32`	The data sample for the histogram
`op_valid`	`OUT`	`1`	Manager operation valid
`op_code`	`OUT`	`3`	Operation code
`op_data`	`OUT`	`32`	Operation data
`op_ready`	`IN`	`1`	Subordinate ready
`op_rvalid`	`IN`	`1`	Response valid
`op_rdata`	`IN`	`32`	Response data

Description

This entity communicates with the HME entity using the custom "OP-Protocol" explained in Section 6.3. It is agnostic about the type of memory used to store histogram information. Its communication is done solely with the `op_XX` signals, interacting with the entity responsible for HBM memory management (the HME entity). Specifically, this entity sends commands specifying operations and the relevant histogram indexes. The HME entity then, using its knowledge of the memory layout, converts these indexes into physical HBM addresses to retrieve or store the information in the correct locations. This entity is unaware of the HME entity internal memory organization, communicating only via these `op_XX` signals. In this exchange, the histogram entity acts as the manager, sending commands to its subordinate, the HME entity.

In case of a reset or after a power up, the entity starts its initialization procedure: for every histogram pull its configuration from memory, sanitize it, store into the BRAM and clear the bin counters.

For every unit, and for every (sub-)histogram of that unit, the index of the final histogram is calculated by simply concatenating these two values together (which is the value of the `data_idx`):

$$\overbrace{01100110}^{\text{histogram index}} = \underbrace{011}_{\text{unit index}} \quad \overbrace{00110}^{\text{sub-hist index}}$$

Using the custom protocol, this entity requests the configuration of a specific histogram in blocks of 32 bytes. The index of the histogram is put in `op_data[23:16]` while the index of the block in `op_data[15:0]`. Then `op_code` is set to `001` and `op_valid` is asserted, and the exchange starts. Once all three configuration parameters are received, their values are validated against defined bounds. If any values are out of bounds, a default is assigned. The next step

involves calculating the maximum edge, which is determined by multiplying the bin width by the number of bins and then adding the minimum edge. This calculated maximum edge should not exceed $2^{32}-1$, as the result is trimmed to the first 32 bits. These four values are then stored in the BRAM, with the histogram index serving as the address. The BRAM itself is 108 bits wide (accommodating three 32-bit values and one 12-bit value) and has a depth of 256 elements, corresponding to the maximum number of histograms. The last initialization step is to clear the bins: `opcode` is set to `111`, `op_data[23:16]` contains the index and `op_data[15:0]` contains the numbers of bins to clear, starting from the bin at index 0. If this is the last histogram the initialization is finished, otherwise the counter is incremented and all this procedure repeated.

The histogram entity is now waiting for histogram data, and `ready` is asserted. When both signals `data_valid` and `ready` are asserted, `data_idx` and `data` are read and buffered internally and `ready` is deasserted. Since `data_idx` contains the index of the histogram, choosing the correct configuration from the BRAM is just a matter of providing that index as the address. The four values stored during initialization are then pulled out of the memory. The incoming data is first checked against its minimum and maximum edges. If it falls outside these bounds, the data is assigned to either the underflow or overflow bin. Otherwise, the divider entity is employed to calculate the precise bin index. This involves subtracting the minimum edge from the data, and then an internal signal initiates the division process with the appropriately configured numerator and denominator, yielding the final bin index. When the correct bin is found, the `op_code` is set to `010`, the histogram index is set into `op_data[23:16]` and the index of the bin is put into `op_data[11:0]`. After the command is sent, no answer is expected and the entity now waits for a new data sample.

Another function of this entity is to reset a single histogram given its index. When `hist_rst` is asserted, the `hist_rst_idx` signal contains the relevant index. Both values are stored internally, awaiting processing. This "single histogram reset" operation functions more like an interrupt than a conventional reset. When the internal signals are set, this entity waits for the current task to complete before handling the single histogram reset by reusing parts of its initialization logic: requesting and validating the configuration, storing it into the BRAM, and then clearing the bins.

6.1.3 Histogram Memory Entity

This entity implements the custom "OP-Protocol" specified in Section 6.3 as the subordinate (the manager is the HE entity). The commands received always specify the index of the histogram on which the operation must be performed. This index is converted to the correct memory address where that histogram is stored.

Definition

Port	Mode	Bits	Description
`clk`	`IN`	1	clock
`rstn`	`IN`	1	reset active low
`op_valid`	`IN`	1	Manager operation valid
`op_code`	`IN`	3	Operation code
`op_data`	`IN`	32	Operation data
`op_ready`	`OUT`	1	Subordinate ready
`op_rvalid`	`OUT`	1	Response valid
`op_rdata`	`OUT`	32	Response data
`AXI_BUS_M`			AXI bus to control one HBM subchannel

Since the DAQ Unit full design already incorporates the HBM IP Core, there is no need to reinstantiate the core with its associated clocks. The `AXI_XX` signals from one free subchannel are connected to this entity. Regarding the AXI protocol used for communication with the HBM, this entity acts as the master, with the HBM acts as its subordinate.

Regarding the `op_` signals, it should be noted that the modes are inverted compared to those of the HE entity.

Description

This entity starts by waiting for an operation. It signals its readiness by asserting `op_ready`. When `op_ready` and `op_valid` are both asserted, `op_data` and `op_code` are read and buffered internally, and the value of `op_ready` is deasserted. The entity is not ready anymore and has an operation to fulfill.

The first step is to extract the index of the histogram to act upon by reading the value of `op_data[23:16]`, then based on the value of `op_code` different operations are performed.

If the value of `op_code` is `001`, a specific block of the configuration must be read from the HBM. The configuration address is calculated using the histogram index, and `AXI_ARADDR` is set. An AXI read transaction is performed, and upon completion, the 256-bit word containing the configuration is retrieved. The specific 32-bit block is then extracted from this full word and put in `op_rdata` (distinct from `op_data`), and `op_rvalid` is asserted for a single clock cycle, thus fulfilling the operation. By design, there is no handshake for the response, requiring the histogram entity to continuously monitor `op_rvalid` for its assertion. The entity then reverts to a ready state, awaiting a new operation.

If the value of `op_code` is `010`, a specific bin must be incremented. This means reading the correct HBM word containing that bin, correctly incrementing the bin's value, and then writing the updated word back to memory. In this scenario, the address must be calculated based on the fact that each HBM word contains four bins. Let b be the index of the bin to increment, found in `op_data[11:0]`. The word containing this bin is located at index $I = b \gg 2$ (which is equivalent to integer division by 4). The remainder of this division, r, specifies the bin's position within that word (r=0,1,2, or 3). After adding I to the base address of the histogram owning the bin, `AXI_ARADDR` is set. An AXI read transaction is then performed, and upon completion, the word containing the desired bin is retrieved. The r-th bin within that word is then incremented, and the updated word is prepared for writing back. By setting `AXI_AWADDR` with the same address and `AXI_WDATA` with the incremented word, an AXI write transaction is performed. Once finished, the entity returns to a ready state, awaiting a new operation; no response is sent in this case.

If the value of `op_code` is `111`, a specified number of bins must be cleared, which means overwriting their stored values with zero. The total number of bins to clear, determined by `op_data[11:0]` + 1 (as per protocol), is then rounded up to the number of HBM words. Let w be the total number of HBM words to erase, calculated as $w = (\texttt{op\_data[11:0]} \gg 2) + 1$, and let q denote the number of words already cleared. For AXI transactions, the HBM supports burst lengths of up to 16, allowing 16 words to be written sequentially. Accordingly, for each burst, `AXI_AWLEN` is set to $\min(15, w - q - 1)$, since `AXI_AWLEN` specifies the number of words minus one in the burst. The `AXI_AWADDR` is set to the current starting address for the burst, `AXI_WDATA` is set to 0, and an AXI write transaction is initiated. Upon completion of each burst, q is incremented by the actual number of words written in that burst (i.e., `AXI_AWLEN` + 1). If $q \geq w$, all required words have been cleared. Otherwise, the process repeats, with `AXI_AWADDR` incremented by the number of words written in the previous burst. Once all bins

are cleared, the entity returns to a ready state, awaiting a new operation. No response is sent in this case.

6.1.4 Histogram Wrapper Entity

This entity encapsulates both the HE and HME entities. It also integrates the 24 unit FIFOs, the aggregator FIFO, and all the necessary logic for transferring data from these FIFOs to the HE entity. Notably, it is the first entity operating across two distinct clock domains: one being the clock utilized by the Unit DAQ design, and the other, an internal clock governing most of this design logic.

Definition

Port	Mode	Bits	Description
`axi_clk`	`IN`	`1`	Clock of the histogram related entities
`wr_clk`	`IN`	`1`	Clock of the incoming data
`rstn`	`IN`	`1`	General reset, active low
`hist_rstn`	`IN`	`1`	Histogram reset only, active low
`single_hist_rst`	`IN`	`1`	To request a reset of a specific histogram
`single_hist_rst_idx`	`IN`	`8`	The index of the histogram to reset
`wr_data_valid`	`IN`	`24`	Specifies which of the 24 input lines has valid data
`wr_data`	`IN`	`[24][32]`	The data of the 24 input lines
`wr_data_idx`	`IN`	`[24][3]`	The index of the sub histogram of the 24 lines
`AXI_BUS_M`			AXI bus to control one HBM subchannel

The value `24` is the number of units, instead the value `3` is the bits to represents the number of histograms per unit (8). The `[X][Y]` notation for the bits column means: the signal is analogous to a matrix, there are X rows and Y columns.

Description

One job of this entity is to handles the incoming data and put them in the unit FIFOs. When the bit i of `wr_data_valid` switches to 1, then if the FIFO of the unit i (`FIFO[i]`) is not full, the `wr_data[i]` is concatenated with `wr_data_idx[i]` and put inside `FIFO[i]`. If it is full then the data is dropped.

Data from the unit FIFOs must be transferred to the aggregator FIFO. Each clock cycle, a rolling counter c is incremented, determining which unit FIFO to check. If `FIFO[c]` is not empty and the aggregator FIFO is not full, data is extracted from `FIFO[c]`, concatenated with its unit index c, and then enqueued into the aggregator FIFO. If any of these conditions are not satisfied, no action is taken, and c is simply incremented. Should the aggregator FIFO be full, `FIFO[c]` lose its turn. The unit FIFOs are 35 bits wide (comprising 32 bits for data and 3 bits for the sub-histogram index) and have a depth of 512 elements. The aggregator FIFO, by contrast, is 40 bits wide (with 35 bits derived from the unit FIFO and 5 bits from the unit index) and also has a depth of 512 elements. The data is now ready to be sent to the histogram entity, but needs to be done gracefully. This involves ensuring the histogram entity is ready to accept data and the aggregator FIFO is not empty. If these conditions are met, the data is dequeued and provided to the histogram entity by asserting the valid data signal for a single clock cycle. Once ready, the histogram entity remains in a ready state until new data is available, thus eliminating the risk of missing the valid signal.

The signal `hist_rstn` is an active low reset used to reinitialize only the histograms, without interrupting any other operation. Specifically the HE entity is reset for hundreds of clock cycles

and is the only entity affected by this reset. The HME entity is not affected because it manages AXI transaction, resetting only one side would cause the transaction to go out of sync. Also there is no strict necessity to reset the latter. This signal originates from an outside request within the Python library and is in the same domain as the `axi_clk`.

Instead the signal `rstn` is an active low reset used to reinitialize all the design. This signal is in the `wr_clk` domain and since it needs to reset all the entities, like the HE and the HME entities, a clock domain crossing is performed with two flip flops to avoid metastability.

Lastly, the signals `single_hist_rst` and `single_hist_rst_idx` are connected to `hist_rst` and `hist_rst_idx` of the histogram entity.

6.1.5 HBM Access Control Entity

This entity handles the requests originated within the Python library, in the Zynq processor. This entity implement the "user interface" explained in Section 6.2.

Definition

Port	Mode	Bits	Description
usr_clk	IN	1	Clock for the communication
axi_clk	IN	1	Clock to access the HBM
rstn	IN	1	Reset active low
usr_func_wr	IN	N	Specify the recipient of the write request
usr_wren	IN	1	Asserted when a write request has arrived
usr_data_wr	IN	64	Data of the write request
usr_func_rd	IN	N	Specify the recipient of the read request
usr_rden	IN	1	Asserted when a read request has arrived
usr_data_rd	OUT	64	Response data of the read request
usr_rd_val	OUT	1	Asserted when the response data is valid
AXI_BUS_M			AXI bus to control one HBM subchannel

Description

This entity handles the I/O requests to the HBM originated from the Zynq processor. Operations like updating the configuration of an histogram or reading its bin counters, are just reading and writing operations to the HBM. The entity is agnostic to who is the sender, it just answer to the signal coming from the user interface.

A read request to the HBM is initiated by writing to a special HBM-read-address. This write request includes as data the desired HBM address and the number, n, of 64-bit words to be read. Afterward, n sequential read requests to the same HBM-read-address are expected. Each response to these read requests delivers one of n words read from the HBM. The value of n ranges from 1 to 64, which means up to 16 HBM words (256 bits) can be read with a single read request, which is also the maximum that can be read in a single AXI transaction.

A write request to the HBM is initiated by writing to a special HBM-write-address. Following this, four sequential write requests to the same control address are expected. Since an HBM word is four 64-bit segments, each of these four requests must contain one quarter of the HBM word (one 64-bit segment). This design approach is simpler compared to the other, primarily because the read channel is utilized significantly more than the write channel; specifically, the configuration is typically written sporadically, whereas all bin counters are read multiple times.

The two FIFOs, used to buffer data for requests, operate across two different clock domains. The internal circuitry of these FIFO IP Cores automatically manages the clock domain crossings. This entity is designed to exploit this convenience as much as possible.

The FIFO responsible for handling read data is asymmetric: its interface to the HBM is 256 bits wide, while the side exposed for user communication is 64 bits wide. The data splitting required for this width difference is handled automatically. This FIFO is initially held in a reset state. A read request to the HBM is initiated when the `usr_wren` signal is asserted and the appropriate bit in `usr_func_wr` is set to 1. At this point, the HBM address is read from `usr_data_wr[32:0]`, and the read length L is determined by `usr_data_wr[38:33]` + 1. Concurrently, a countdown begins to activate the FIFO in three clock cycles. This circuit operates in the `usr_clk` domain, while the one sending AXI transactions to the HBM operates in the `axi_clk` domain. The latter needs to be notified of the new read request to begin the AXI transaction. The arrival of a write request can be inferred by detecting the rising edge of the signal that indicates the FIFO is ready to accept data (write-ready). Specifically, when a request arrives, the FIFO is activated (not in a reset state anymore). This activation causes the write-ready signal in the `axi_clk` domain to transition from low to high, informing the AXI circuit that a new read transaction must commence. As at least three clock cycles have passed, the address and length values stored in internal registers by the other circuit are guaranteed to be stable and can be used without issue. When the AXI transaction is finished, data is read from the HBM in 256-bit words and stored in the FIFO. Now, the reverse problem emerges: the other circuit operating in the `usr_clk` domain needs to be notified that data is available in the FIFO. In this scenario, the signal to monitor is simply the queue empty signal; once this signal deasserts (falls), data can be read from the queue. At this time, the AXI transaction could be still operating and transferring data into the queue. But it can be safely assumed that all L read requests can be made, as the roundtrip time of these far exceeds the HBM worst case latency. When `usr_rden` is asserted together with the correct bit of `usr_func_rd` a read request has arrived. The entity should then send the requested data by using the `usr_data_rd` and `usr_rd_val` signals. When a brand new read request arrives (the one using `usr_wren`), the cycle restarts, with the key difference that the FIFO is reset before the countdown begins (since it would still be active from its previous activation).

For writing data into the HBM, the FIFO is symmetric, with both sides being 64 bits wide. When the `usr_wen` signal is asserted and the appropriate bit in `usr_func_wr` is set to 1, the value of `usr_data_wr` is placed into the FIFO. The very first request contains the address, with subsequent requests containing the data. This implies that the queue should consistently hold an address followed by four pieces of data. Meanwhile if the FIFO is not empty, the AXI circuit (operating in the `axi_clk` domain) pulls out the data while keeping a count. The first item pulled is the address; the others are used to construct the 256-bit wide word. Once the fourth piece of data is pulled out, the transaction begins and the counter is reset. Meanwhile, the other circuit can handle new incoming write requests.

Both FIFOs are 512 elements deep, even though a significantly shallower depth would be sufficient. This is due to the minimum size constraint imposed by the width of the built-in FIFO IP blocks.

6.1.6 AXI Interconnector Entity

Given that the bus connected to the HBM is a shared resource, it requires arbitration between the two requesting entities; this component is responsible for its coordination. Specifically, it manages the bus for the HE and HACE entities. This interconnector operates transparently to the two entities: there is no difference for them to be connected to this or directly to the HBM.

Definition

Port	Mode	Bits	Description
clk	IN	1	Clock
rstn	IN	1	Reset active low
AXI_BUS1_XYZ			AXI subordinate bus 1
AXI_BUS2_XYZ			AXI subordinate bus 2
AXI_HBM_XYZ			AXI bus connection to the HBM

Description

Assuming the previous access was granted to bus 1 and that transaction is concluded. Now, in each clock cycle, bus 2 is checked first if it needs to start a transaction, and then bus 1, or vice versa. Should bus 2 initiate a transaction, it asserts the AXI_BUS2_AXVALID signal (following the AXI protocol) and awaits the assertion of AXI_HBM_AXREADY from the other side. If bus 2 is selected to access the HBM, its bus is then connected to the HBM. If it is not selected, it will maintain the assertion of AXI_BUS2_AXVALID until its turn. The transaction is then carried as there is a direct connection between the two. The conclusion of a transaction can be easily determined by observing the signals: for a read transaction, AXI_HBM_RLAST, AXI_HBM_RVALID, and AXI_BUSX_RREADY are monitored; while for a write transaction, AXI_HBM_BVALID and AXI_BUSX_BREADY. When a transaction ends, this entity repeats the process.

The write and read channels are independent so they can be used simultaneously by two different entities.

6.1.7 DTHistogram Entity

This entity encapsulates all the aforementioned components. Its ports are dedicated to interfacing with the complete CMS DAQ Design. Additionally, it processes specific commands originating from the Python library that are unrelated to HBM operations (as those are handled by the HACE entity). Finally, it manages reset functionalities.

Definition

Port	Mode	Bits	Description
axi_clk	IN	1	Clock for the AXI and histogram related entities
usr_clk	IN	1	Clock for the communication
usr_rst_n	IN	1	Reset active low
usr_func_wr	IN	N	Specify the recipient of the write request
usr_wren	IN	1	Asserted when a write request has arrived
usr_data_wr	IN	64	Data of the write request
usr_func_rd	IN	N	Specify the recipient of the read request
usr_rden	IN	1	Asserted when a read request has arrived
usr_data_rd	OUT	64	Response data of the read request
usr_rd_val	OUT	1	Asserted when the response data is valid
hist_data_valid	IN	24	Specifies which of the 24 input lines has valid data
hist_data	IN	[24][32]	The data of the 24 input lines
hist_data_idx	IN	[24][3]	The index of the sub histogram of the 24 lines
AXI_BUS_M			AXI bus to control one HBM subchannel

Description

The `hist_data_` signals are connected directly to the HWE entity. Same for the `usr_` signals but this entity also use those signals for some special operations: resetting the bins of all the histograms or of only one and providing some debug information. The former is not really performed here, but the request is converted in the specific signals which are connected to the HWE entity (which in turns are connected to the HE entity).

When `usr_wen` is asserted and the specific bit of `usr_func_wr` is set to 1, the value of `usr_data_wr[0]` is examined. If it is 1, this indicates a request to reset only a specific histogram. If it is not 1, this signifies a request to set the reset state (either assert or deassert) for all histograms collectively. In the former case, `usr_data_wr[23:16]` contains the index of the specific histogram to be reset, which then pulses the corresponding signals connected to the HWE entity for one clock cycle. Otherwise, if `usr_data_wr[0]` is not set, the signal responsible for resetting all histograms is set to the value of `usr_data_wr[1]`, implementing a togglable reset. This value is subsequently applied to the relevant signals connected to the HWE entity.

Lastly, if `usr_rst_n` (the main system-wide reset signal) is deasserted (active low), it remains deasserted for 5 clock cycles. Following this, it is propagated to all other entities after being resynchronized into the `axi_clk` domain for certain components.

6.2 Outside Communication

Monitoring a system by collecting statistical data, such as histograms, is important, but the ability to extract this data from the FPGA is crucial. This is addressed in the DTH-400 by the presence of a Zynq processor on the board, which is connected to the CMS DAQ FPGA.

Communication between the FPGA and the Zynq is achieved via a chip to chip interface. On the FPGA side, an IP core handles all encoding and decoding, exposing an AXI interface to the user. An even simpler layer, referred to here as the "user interface", is built on top of this. On the Zynq side, the HAL library[1] interfaces with the low-level device, providing high-level methods to the user, upon which a Python library is built. This data can then be readily accessed because the processor is equipped with an Ethernet port, allowing external users to connect to the Zynq. The chip to chip interface on the FPGA is a unique component within the entire CMS DAQ design; all entities that need to communicate with the Zynq connect to it. The recipient of a request is distinguished by its address. These addresses are simply incremental numbers and do not correspond to any physical memory locations. A simplified diagram of this system is presented in Figure 6.1.

The Zynq processor is the manager while the "user interface" inside the FPGA is the subordinate. No communication can be started from inside this side. To better understand the "user interface" the signal are shown here:

[1] https://gitlab.cern.ch/cmsos/worksuite/-/tree/baseline_sulfur_16/hal.

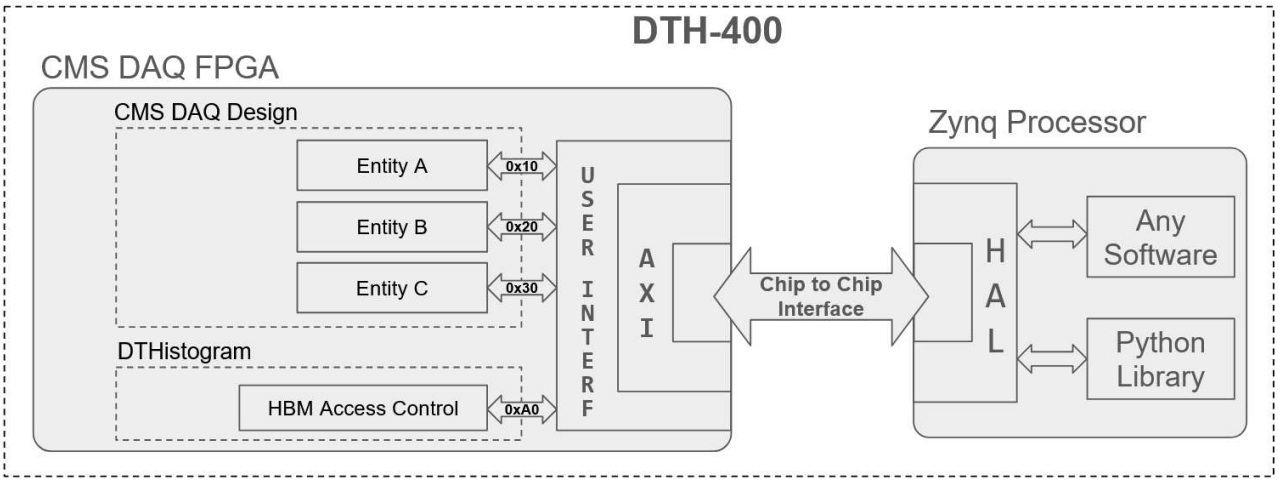

Figure 6.1: Communication diagram from the FPGA to the Zynq Processor and vice versa.

Port	Mode	Bits	Description
`usr_clk`	`IN`	`1`	Clock
`usr_rst_n`	`IN`	`1`	Reset active low
`usr_func_wr`	`IN`	`N`	Specify the recipient of the write request
`usr_wren`	`IN`	`1`	Asserted when a write request has arrived
`usr_data_wr`	`IN`	`64`	Data of the write request
`usr_func_rd`	`IN`	`N`	Specify the recipient of the read request
`usr_rden`	`IN`	`1`	Asserted when a read request has arrived
`usr_data_rd`	`OUT`	`64`	Response data of the read request
`usr_rd_val`	`OUT`	`1`	Asserted when the response data is valid

All signals are characterized by the prefix `usr_`. Possible requests originating from the processor are read and write requests, identified respectively by `rd` and `wr`.

As all entities involved in this communication share the same interface, when a request arrives, the recipient and the type of request are specified by the `usr_data_xx` signal. Each entity has a defined range of possible actions, and all these actions such as writing to the HBM, resetting a specific component, or modifying a register value is assigned a unique number i. When `usr_xxen` is asserted and the i-th bit of `usr_data_xx` is high, the entity corresponding to that number must fulfill the request. Crucially, only one bit of `usr_data_xx` can be set high at any given time, or none at all.

In the case of a read request, `usr_rden` will be asserted concurrently with one bit of `usr_data_rd`. The designated entity should then provide the result inside the `usr_data_rd` signal and assert `usr_rd_val` for a single clock pulse.

Conversely, for a write request, `usr_wren` will be asserted along with one bit of `usr_data_wr`, and the `usr_data_wr` signal will contain valid data. The entity is then expected to perform the requested operation using the received data

6.3 The Custom OP-Protocol

This custom protocol is used to communicate between the HE and the HME entity. In this way the former is agnostic to the memory used and addressed, while the latter is agnostic to the configuration of the histograms.

Port Name	Driven by	Bits	Description
op_valid	Manager	1	Manager operation valid
op_code	Manager	3	Operation code
op_data	Manager	32	Operation data
op_ready	Subordinate	1	Subordinate ready
op_rvalid	Subordinate	1	Response valid
op_rdata	Subordinate	32	Response data

The op_code signal specifies the operation to be performed:

Code	Name
001	Read Configuration Data
010	Increment a Bin
111	Erase Bins

Read Configuration Data

This operation reads 32 bits from an histogram configuration space. The whole configuration of each histogram is divided in blocks of 32 bits. By specifying which histogram and which block to read from the configuration, every parameter of the configuration can be retrieved. The manager needs to wait for a response.

The op_data signal specifies which histogram and which block to read. Instead op_data[2:0] specify which block to get from the configuration of the histogram (8 possible values to index 8 block of 32 bits for a total of 256 bits, a HBM word which is also the configuration space of an histogram). Lastly, op_data[23:16] specify the index of which histogram to read the configuration from. The index is structured the same as the histogram index in the design (the index of the unit followed by the index of its sub histogram). The configuration will be sent back into op_rdata when op_rvalid is asserted.

Increment a Bin

This operation increments a specific bin of an histogram. Which bin and which histogram to increment are specified in op_data[11:0]. Instead op_data[27:16] specify the index of the histogram which owns the bin to increment. The index is structured the same as above. No response is sent in this case.

Erase Bins

This operation erases (sets to 0) a specific amount of bins of an histogram starting from the first bin in memory. How many bins of which histogram to erase are specified in the op_data signal: op_data[11:0] specify how many bins to erase encoded as: num_bin = op_data[11:0] + 1. Setting it to N will cause the bins from index 0 to index N to be erased, for a total of N+1 bins. op_data[23:16] specify the index of which histogram to erase the bins from. The index is structured the same as above. No response is sent in this case.

6.3.1 Protocol Description

When the manager wants to perform a new operation, first it asserts op_valid and drives valid data on op_code and op_data. All these signals must be kept to the same values until op_ready is asserted by the subordinate. The clock cycle where op_valid and op_ready are both asserted, op_code and op_data are read from the subordinate and the operation starts and the handshake ended. If the operation does not require a response, the manager can start a new operation, otherwise the manager waits for the response by looking a the signal

`op_rvalid`. When `op_rvalid` is asserted then `op_rdata` will contain the response. There is no acknowledgment for the response from the manager to the subordinate.

6.4 Final Remarks

6.4.1 Resource Usage

The resource utilization of the DTHistogram design is reported in Table 6.1. It is important to note that the Top Entity, as mentioned in Section 6.1, was included in this calculation. However, since this entity contains some IP cores and logic that will already be part of the final CMS DAQ Design, the figures in the table represent an upper limit of the resource usage. Furthermore, when compared to the full design (Table 4.1), it is evident that this project's resource consumption is minimal.

Resource	Utilization	Available	Utilization %
LUT	4371	871680	0.50
LUTRAM	33	403200	0.01
FF	8079	1743360	0.46
BRAM	32.50	1344	2.42
DSP	4	5952	0.07
IO	3	416	0.72
GT	1	64	1.56
PLL	1	16	6.25

Table 6.1: Resource utilization for the DTHIstogram Design

6.4.2 Processing Time

The maximum data rate this design is able to withstand needs to be determined. Specifically, the average time spent (in clock cycles) by the HE and HME entities between processing two data points should be calculated. These two entities form the core of the design, responsible for categorizing data into the correct bins and incrementing counters in the HBM. The test must be conducted with the aggregator FIFO nearly full, ensuring that the delay between processing two data pieces is solely attributable to the logic and the HBM. It should be noted that the processing time is not consistently uniform for two reasons: firstly, the HBM sometimes requires more time to become ready during an AXI transaction. Secondly, when determining the correct bin, if the data falls out of bounds (into overflow or underflow bins), the division operation is not performed, resulting in a saving of approximately 14 clock cycles.

To perform this check, it is sufficient to observe the interval between two `ready` signals from the HE entity. The `ready` signal is asserted only when the entity has finished categorizing the data into the correct bin. Since data is always ready to be processed, and thus `data_valid` is always asserted by the HWE entity, on the next clock cycle, `ready` is deasserted and new data is ingested. The same principle applies to the `op_ready` signal, which is controlled by the HME entity. This is shown in figure 6.2. By measuring the difference between two `ready` signals, the time (in clock cycles) can be determined and then averaged across all other points.

The average number of clock cycles between two data points is approximately 62, which translates to roughly 248 ns at 250 MHz. This implies that within two LHC orbits (approximately 88 µs), around 355 distinct data samples can be processed. If there are 24 units, the firmware can accommodate up to 14 histograms per unit, which is more than sufficient.

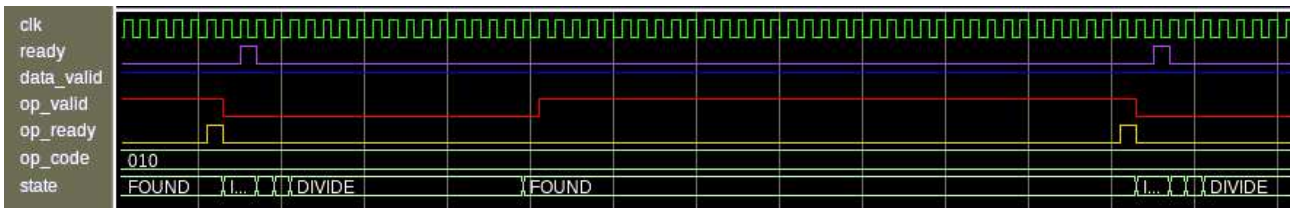

Figure 6.2: Simulation of some signals of the HE entity.

The design was thoroughly tested using the QuestaSim Simulator[2] by writing test benches that covered most possible cases.

[2] https://eda.sw.siemens.com/en-US/ic/questa-one/simulation/questa-one-sim/

Chapter 7

Conclusion

The upgrade of the Large Hadron Collider to High Luminosity necessitated significant hardware upgrades not only for the collider itself but also for the experiments operating on it. This is particularly true for CMS, where a substantial upgrade (Phase 2) is underway to handle this massive influx of new data. The CMS DAQ team developed the new DTH-400 and DAQ-800 boards to manage the immense data flow from the backend boards. As these systems grow in complexity and demand faster, greater resources, robust hardware like high performance FPGAs must be utilized. Even a small percentage of unused resources of these hardware can represent a significant amount in absolute terms. This made possible to implement one, and potentially more in the future, monitoring systems alongside the main design firmware.

The monitoring system, DTHistogram, presented in this thesis, allows for an immediate assessment of erroneous behavior by examining histograms. Users can quickly identify if a particular channel is behaving unusually or if an observable histogram across all units deviates from the norm.

This work can be expanded with new functionalities, such as monitoring data dropping that occurs when the data rate exceeds a certain threshold for an extended period, or if an obscure design bug arises. If a significantly higher number of histograms per unit are required, directly accessing the HBM to increment each individual bin becomes inefficient, necessitating the development of a caching system to mitigate this issue. In some areas, the design could be simplified, as a generic approach was taken in certain cases or to handle scenarios that does not occur.

Appendix A

Division Schematic and Code

In this appendix, the final schematic of the Division Entity is presented. This entity was chosen as an example due to its self contained nature, simplicity, and the fact that its schematic can be displayed on a standard page (is small enough). First, the complete code is provided, then figure A.1 shows the simulation of that code for the division of 437900 by 234.

```
entity uint_divider is
    GENERIC (
        BITS_DATA : natural := 32;
        BITS_RES : natural := 12
    );
    PORT(
        clk         :   IN  STD_LOGIC;
        rstn        :   IN  STD_LOGIC;
        enable      :   IN  STD_LOGIC;
        num         :   IN  unsigned(BITS_DATA-1 downto 0);
        den         :   IN  unsigned(BITS_DATA-1 downto 0);
        res_valid   :   OUT STD_LOGIC := '0';
        res         :   OUT unsigned(BITS_RES-1 downto 0)
    );
end uint_divider;

architecture Behavioral of uint_divider is
    SIGNAL enabled  :   STD_LOGIC := '0';
    SIGNAL num_copy :   UNSIGNED(BITS_DATA-1 downto 0);
    SIGNAL den_copy :   UNSIGNED(BITS_DATA-1 downto 0);
    SIGNAL res_copy :   UNSIGNED(BITS_RES-1 downto 0);
    signal tmp      :   unsigned(BITS_DATA-1 downto 0);
    signal cnt: natural range 0 to BITS_RES-1 := 0;
begin

res <= res_copy;

process(clk, rstn)
begin
    if rstn = '0' then      -- reset
        res_valid <= '0';
        cnt <= 0;
        enabled <= '0';

    elsif rising_edge(clk) then
        res_valid <= '0';

        if enable = '1' then    -- new division request
            cnt <= BITS_RES-1;

            tmp <= (others => '0');
            res_copy <= (others => '0');
            if num = 0 then         -- If the num is 0, then the result is ready
                res_valid <= '1';
                enabled <= '0';
            else                    -- If is not, store the values internally
                num_copy <= num;
                den_copy <= den;
                enabled <= '1';
            end if;

        elsif enabled = '1' then    -- If we are performing a division
            if (num_copy - tmp) >= shift_left(den_copy, cnt) then
                res_copy(cnt) <= '1';
```

```
                tmp <= tmp + shift_left(den_copy, cnt);
            end if;

            if cnt = 0 then          -- We processed the last bit of the result
                res_valid <= '1';
                enabled <= '0';
            else
                cnt <= cnt - 1;
            end if;
        end if;
    end if;
end process;
```

By looking at the image, when the `enable` signal is asserted, in the next clock cycle, `enabled` is asserted, indicating that the entity is now performing a division. The signals `num` and `den` are copied and stored internally in `num_copy` and `den_copy`, while `res_copy` and `tmp` are cleared, and `cnt` is set to 11 (for the eleventh bit). Now, at every clock cycle, the comparison is performed, and `cnt` is decremented. If the comparison is true, `res_copy[cnt]` is set to 1, and `tmp` contains the result of the multiplication between `den_copy` and `cnt`. Specifically, instead of setting `tmp`= `res_copy`·`den_copy` every cycle, which is a costly and inefficient operation, `tmp` is incremented with the value added to `res_copy` at that time ($2^{\texttt{cnt}}$) multiplied by `den`. When `cnt` reaches 0, `enabled` is deasserted, `res_copy` is copied to `res`, and `res_valid` is asserted for one clock cycle.

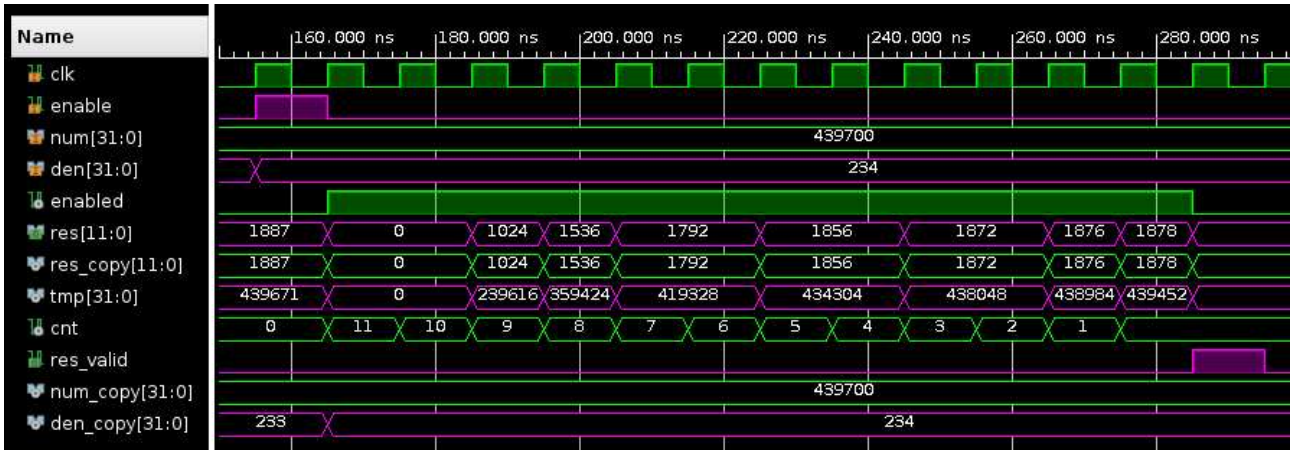

Figure A.1: Simulation of the Division Entity circuit performing the operation 439700 / 234.

Finally, figure A.2 displays the schematic of this entity. This schematic represents the final implementation of the circuit, with all elements mapped to the actual FPGA components (primarily LUTs and flip-flops).

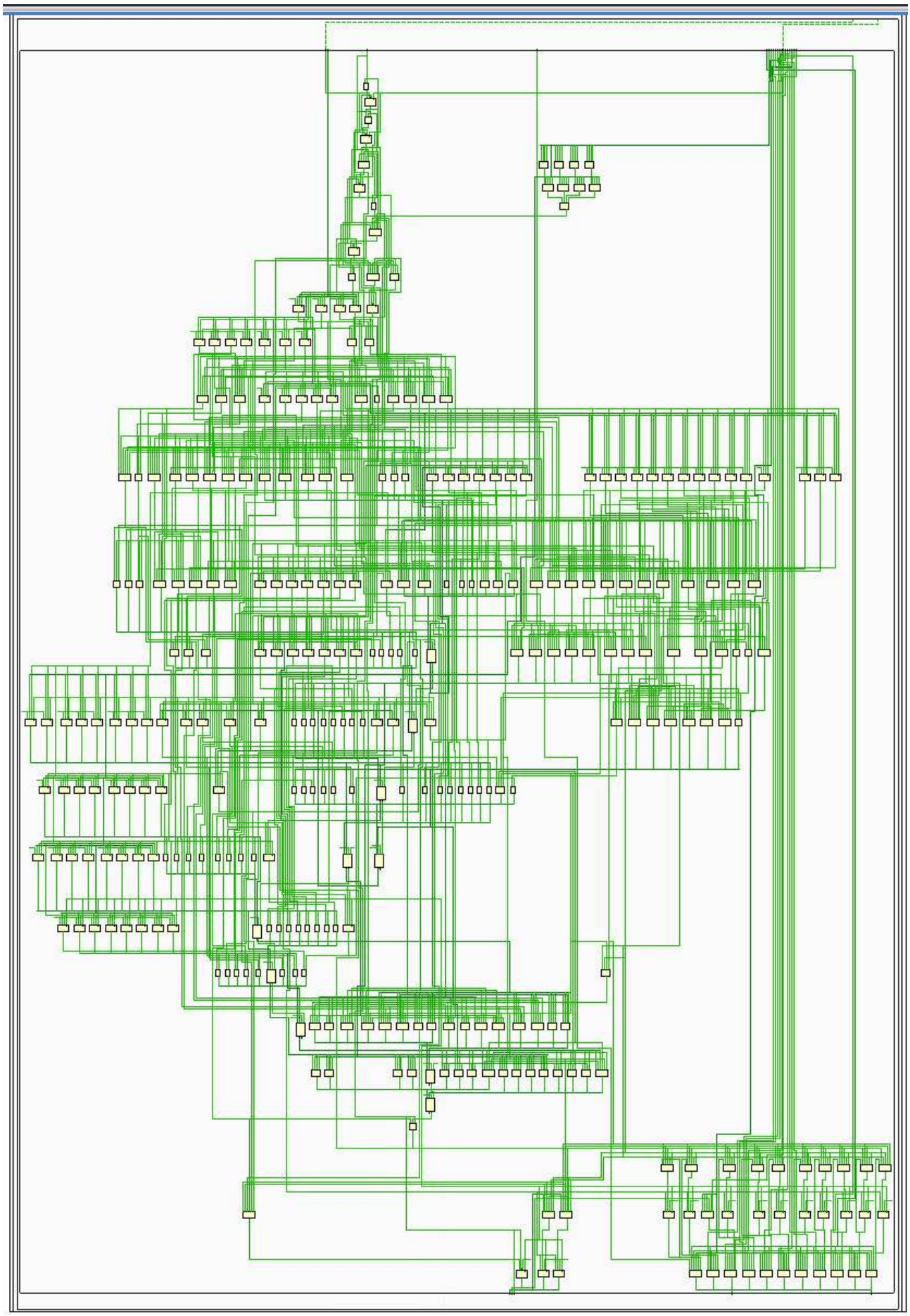

Figure A.2: Schematic of the Division entity. This is the "implemented" schematic, where the digital elements are mapped to the components of the FPGA. The inputs are at the top while the outputs are at the bottom.

Appendix B

Python Library

This appendix describes the functionality of the Python library designed to facilitate interaction with the DTHistogram system on the FPGA, particularly for configuration, data access, and system resets. The library abstracts the underlying hardware communication details, allowing for high level control and monitoring.

```
from dthist import HistManagerDTH
histman = HistManagerDTH(mem_offset=0x20, num_unit=24, hist_per_unit=5)
```

this code imports and instantiate the `HistManagerDTH` object. This object can be configured with the offset of the first histogram, with the total number of units and the number of (sub)histogram per unit.

Then reading the configuration of a specific histogram is just:

```
unit = 3
hist = 6
min_edge, bin_width, bin_num = histman[unit, hist].get_conf()
```

while writing a new configuration is:

```
min_edge = 10
bin_width = 100
bin_num = 20
histman[unit, hist].set_conf(min_edge, bin_width, bin_num)
```

where `histman[unit, hist]` specifies the index of the unit followed by the index of the histogram of that unit.

Instead, reading the bin counters of a specific histogram is done in the following way:

```
unit = 3
hist = 6
bin_counter = histman[unit, hist].get_bins()
```

the library handles the multiple reading required to retrieve all the bins (since, up to 64 words of 64-bit can be retrieved at a single time).

To reset a single histogram (pulse reset):

```
unit = 3
hist = 6
```

```
histman[unit, hist].reset()
```

instead to reset all and only the histograms, without interfering with all the DTHistogram:

```
histman.set_reset(True) # reset activated
histman.set_reset(False) # reset deactivated
```

in this case is a toggle reset, and it will stays like that until changed.

The library also provide two ways to iterate through all the bins:

```
for unit_id, hist_id, hist in histman.items(): # indexes are provided
    print(unit_id, hist_id, hist)

for hist in histman:
    print(hist)
```

Reading and writing to a generic address of the HBM can be done with two functions:

```
from dthist import *
dth = get_dth()
words_list = read_hbm(dth, addr=0x80, nwords=20)

write_hbm(dth, addr=0xA0, data=[123, 456, 789, 001])
```

where `get_dth()` returns an object representing the underlying device (when using `HistManagerDTH` this is done automatically)

Lastly, to fully reset everything:

```
from dthist import *
dth = get_dth()
reset_all(dth)
```

Bibliography

[1] *Snowmass White Paper Contribution: Physics with the Phase-2 ATLAS and CMS Detectors*. Tech. rep. Geneva: CERN, 2022. URL: https://cds.cern.ch/record/2806962.

[2] Oliver Sim Brüning et al. *LHC Design Report*. CERN Yellow Reports: Monographs. Geneva: CERN, 2004. DOI: 10.5170/CERN-2004-003-V-1. URL: https://cds.cern.ch/record/782076.

[3] Ewa Lopienska. "The CERN accelerator complex, layout in 2022. Complexe des accélérateurs du CERN en janvier 2022". In: (2022). General Photo. URL: https://cds.cern.ch/record/2800984.

[4] R Bailey and Paul Collier. *Standard Filling Schemes for Various LHC Operation Modes*. Tech. rep. Geneva: CERN, 2003. URL: https://cds.cern.ch/record/691782.

[5] S Chatrchyan and Hmayakyan. "The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment". In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08004. DOI: 10.1088/1748-0221/3/08/S08004. URL: https://cds.cern.ch/record/1129810.

[6] *TikZ.net - Graphics with TikZ in LaTeX*. URL: https://tikz.net/axis3d_cms/.

[7] Tai Sakuma and Thomas McCauley. *Detector and event visualization with SketchUp at the CMS experiment*. Tech. rep. Comments: 5 pages, 6 figures, Proceedings for CHEP 2013, 20th International Conference on Computing in High Energy and Nuclear Physics. Geneva: CERN, 2014. DOI: 10.1088/1742-6596/513/2/022032. arXiv: 1311.4942. URL: https://cds.cern.ch/record/1626816.

[8] V Karimäki et al. *The CMS tracker system project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: https://cds.cern.ch/record/368412.

[9] *The CMS electromagnetic calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: https://cds.cern.ch/record/349375.

[10] *The CMS hadron calorimeter project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: https://cds.cern.ch/record/357153.

[11] J. G. Layter. *The CMS muon project: Technical Design Report*. Technical design report. CMS. Geneva: CERN, 1997. URL: https://cds.cern.ch/record/343814.

[12] Aram Hayrapetyan, Armen Tumasyan, and Adam. "Development of the CMS detector for the CERN LHC Run 3. Development of the CMS detector for the CERN LHC Run 3". In: *JINST* 19.05 (2024), P05064. DOI: 10.1088/1748-0221/19/05/P05064. arXiv: 2309.05466. URL: https://cds.cern.ch/record/2870088.

[13] Milos Dordevic. "The CMS Particle Flow Algorithm". In: *EPJ Web Conf.* 191 (2018), p. 02016. DOI: 10.1051/epjconf/201819102016. URL: https://cds.cern.ch/record/2678077.

[14] Wolfgang Adam et al. *Track Reconstruction in the CMS tracker*. Tech. rep. Geneva: CERN, 2006. URL: https://cds.cern.ch/record/934067.

[15] A.M. Sirunyan, Armen Tumasyan, and Adam. "Performance of the CMS muon detector and muon reconstruction with proton-proton collisions at $\sqrt{s} = 13$ TeV". In: *JINST* 13.06 (2018), P06015. DOI: 10.1088/1748-0221/13/06/P06015. arXiv: 1804.04528. URL: https://cds.cern.ch/record/2313130.

[16] Albert M Sirunyan, Armen Tumasyan, and Adam. "Electron and photon reconstruction and identification with the CMS experiment at the CERN LHC". In: *JINST* 16.05 (2021), P05014.

DOI: 10.1088/1748-0221/16/05/P05014. arXiv: 2012.06888. URL: https://cds.cern.ch/record/2747266.

[17] Matteo Cacciari, Gavin P. Salam, and Gregory Soyez. "The anti-kt jet clustering algorithm". In: *Journal of High Energy Physics* 2008.04 (Apr. 2008), p. 063. DOI: 10.1088/1126-6708/2008/04/063. URL: https://dx.doi.org/10.1088/1126-6708/2008/04/063.

[18] A.M. Sirunyan et al. "Performance of reconstruction and identification of τ leptons decaying to hadrons and ν_τ in pp collisions at $\sqrt{s} = 13$ TeV". In: *JINST* 13 (2018), P10005. DOI: 10.1088/1748-0221/13/10/P10005. arXiv: 1809.02816. URL: https://cds.cern.ch/record/2637646.

[19] Thiago Rafael Fernandez Perez Tomei. *CMS Upgrades for the High-Luminosity LHC Era*. Tech. rep. 8 pages, 7 figures. Presented at the 12th Large Hadron Collider Physics Conference - LHCP2024, Boston, USA, June 03-07, 2024. Geneva: CERN, 2024. DOI: 10.22323/1.478.0189. arXiv: 2501.03412. URL: https://cds.cern.ch/record/2916169.

[20] CMS Collaboration. *The Phase-2 Upgrade of the CMS Data Acquisition and High Level Trigger*. Tech. rep. This is the final version of the document, approved by the LHCC. Geneva: CERN, 2021. URL: https://cds.cern.ch/record/2759072.

[21] Petr Žejdl. *CMS DAQ Network upgrade for Run4 of HL-LHC*. 2025. URL: https://indico.cern.ch/event/1483219/contributions/6317240/attachments/3007975/5303595/2025_02_DAQ_LHC_CMS_Network.pdf.