UNIVERSITÁ DI PISA
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

# DESIGN AND IMPLEMENTATION OF AN INTEGRATED FULLY DIGITAL TRIGGER AND DATA ACQUISITION SYSTEM FOR HIGH ENERGY PHYSICS EXPERIMENTS

DOCTORAL THESIS

Author
**Elena Pedreschi**

Tutors
**Prof. Luca Fanucci**
**Dott. Franco Spinella**

Reviewers
**Dott. Gianmaria Collazuol**
**Dott. Ing. Angelo Cotta Ramusino**

The Coordinator of the PhD Program
**Prof. Marco Luise**

Pisa, December 2016

XXIX

This thesis is dedicated to Diana, Gianfranco and Francesco

“Everybody is a genius. But if you judge a
fish by its ability to climb a tree, it will
live its whole life believing that it is
stupid.”
A. Einstein

# Acknowledgements

I WISH to send all my gratitude to those who have helped me with suggestions, criticism and comments - to them goes my gratitude, though to me the responsibility for any error contained in this thesis.

I first wish to thank Professor Luca Fanucci, Tutor, and Dottor. Franco Spinella, Co-tutor: without their support and their wise guidance this thesis would not exists.

I would continue on with my external evaluators, the Engineer Angelo Cotta Ramusino (INFN), and Dottor Gianmaria Collazzuol (University of Padua): they patiently read and reviewed my paper, giving advices and timely corrections that made it better.

My special thanks to my colleagues at INFN and CERN - in particular Marco, Franco, Bruno, Jacopo, and Roberto - with whom I shared these years of research and work. Together with them I faced the hard and arduous journey in which I grew as a person and as a researcher.

Finally, I would like to express my gratitude to my parents and to my husband that this work is dedicated.

# Ringraziamenti

DESIDERO ricordare tutti coloro che mi hanno aiutato nella stesura della tesi con suggerimenti, critiche ed osservazioni: a loro va la mia gratitudine, anche se a me spetta la responsabilità per ogni errore contenuto in questa tesi.

Ringrazio anzitutto il  Professor Luca Fanucci, Relatore,  ed il Dottor Franco Spinella,  Co-relatore: senza il loro supporto e la loro guida sapiente questa tesi non esisterebbe.

Proseguo con i miei valutatori esterni, l'Ingegner Angelo Cotta Ramusino (INFN) ed il Dottor Gianmaria Collazzuol (Università di Padova): hanno letto con pazienza e revisionato il mio elaborato fornendomi i consigli e le correzioni puntuali che lo hanno reso migliore.

Un ringraziamento particolare va ai i miei colleghi, dell' INFN e del CERN, con cui ho condiviso questi anni di ricerca e lavoro. Un occhio di riguardo e un grazie speciale vanno a Marco, Franco, Bruno, Jacopo e Roberto insieme a loro ho affrontato un viaggio duro e faticoso in cui sono cresciuta come persona e come ricercatore.

Vorrei infine ringraziare le persone a me più care: i miei genitori e mio marito, a cui questo lavoro è dedicato.

# Summary

THE work reported in this thesis has been performed within the project "Experiment to detect KL Very Rare decays" (KLEVER). KLEVER aims at using powerful programmable systems in the first stages of the data collection and selection process in particle experiments at accelerators, i.e. the use of hardware processors based on Field-Programmable Gate Arrays (FPGAs) and Graphic Processing Units GPUs. The FPGAs are placed at the front-end stage of detectors, immediately after digitization, thus allowing data processing at an earlier stage of the acquisition and trigger chain. We aim also to exploit the use of highly parallelized processors, the GPUs, in order to process data at early selection stages. In recent years GPUs were increasingly used to build high-performing computing systems at reasonable prices, but the growth of their computing power and the reduction of their intrinsic latency is such that they are nowadays suited for real-time application. Both these possibilities entail an effort of integration and adaptation, as these systems were developed for totally different purposes, such as the automotive and the video-games market. In particular the intention is to probe the performances of FPGAs and GPUs processors by building a system acting as an easily updatable test bench of the attainable collection and selection capabilities of large amounts of data. This will allow to evaluate the present technological limits, which in turn represent the most important bottleneck for a high-precision physics experiment studying ultra-rare decays.

My work was focused on the development of the Trigger and Data Acquisition System for the experiment NA62. The NA62 experiment is placed in the CERN North Area in the Super Proton Synchrotron accelerator extraction site and it aims at measuring the Branching Ratio of the ultra-rare decay $K^+ \to \pi^+ \nu\overline{\nu}$ in order to provide a stringent test of the Standard Model. Since the value predicted by the Standard Model is very precise, the measurement of this quantity represents an excellent way to investigate the existence of New Physics, or in case of agreement with the Standard Model(SM) to improve the current knowledge of the $|V_{td}|$ parameter of the CKM matrix. The use of a high-rate kaon beam will result in an event rate of about 15 MHz, so high that it is impossible to store data on disk without a very selective reduction. The experiment use devised three trigger levels, allowing to reduce the data rate fed to the readout PC farm

down to $\sim$10 kHz.

High Energy Physics environment, the historical approach to Trigger and data AcQuisition (TDAQ) system, the state of the art and the integrated fully-digital system approach proposed in this thesis work are described in chapter 1.

In chapter 2 the NA62 experimental setup is described, composed of an upstream part, with detectors used to identify and measure the propriety of the $K^+$ inside the beam, and a downstream part where the decay products are detected.

The first part of this work concerns the hardware and firmware development of the common trigger and data acquisition system for the majority of detectors in NA62. The unified trigger and data acquisition system, where the trigger is integrated inside the DAQ, and allowing a good control of the trigger using the same data available at readout, and a excellent flexibility, is presented in chapter 3.

The second part of the work describes the NA62 L0 standard trigger and the studies performed for a L0 trigger based on GPU. The L0 hardware trigger is described in chapter 4 and the attention is focused on the trigger firmware developed for the RICH detector. The use of GPU in high energy physics, the NA62 GPU trigger and the GPU-RICH firmware are described in chapter 5.

# Sommario

IL il lavoro riportato in questa tesi è stata eseguito nell'ambito del progetto "Experiment to detect KL Very Rare decays "(KLEVER).Obiettivo di KLEVER sono lo studio e la realizzazione di sistemi integrati per l'elaborazione dei dati acquisiti da esperimenti di fisica delle alte energie, basati su processori massicciamente paralleli; KLEVER si propone di esplorare le possibilita' attuali offerte dai processori implementabili in FPGA e dai processori grafici (GPU). Le FPGA sono posizionate vicino ai front end dei rivelatori e ricevono quindi i dati digitalizzati da usare nella selezione degli eventi gia' negli stadi piu' a monte della catena di acquisizione. Le GPU sono processori molto avanzati utilizzati nelle schede grafiche ed negli ultimi anni vengono utilizzati in modo sempre piu' massiccio anche per realizzare sistemi di calcolo di grande potenza a costi contenuti. La crescita continua della loro potenza e la diminuzione dei tempi di latenza permette oggi di considerare tali processori anche per possibili applicazioni in tempo reale negli esperimenti di Fisica delle particelle agli acceleratori. Entrambe queste possibilità comportano uno sforzo di integrazione e adattamento, in quanto questi sistemi sono stati sviluppati per impieghi totalmente diversi, come ad esempio nell'industria automobilistica e nel mercato videogiochi. In particolare si intende sondare le prestazioni dei processori FPGA e GPU costruendo un sistema che agisca come un banco di prova facilmente aggiornabile delle capacità di raccolta e selezione raggiungibili di grandi quantità di dati. Ciò consentirà di impostare gli attuali limiti tecnologici, che a loro volta rappresentano più importante collo di bottiglia per un esperimento di fisica ad alta precisione determinato a studiare decadimenti ultra rari.

Il mio lavoro si è concentrato sullo sviluppo del trigger e del sistema di acquisizione dati per l'esperimento NA62. L'esperimento NA62 è collocato nella North Area del CERN sul sito di estrazione dell'acceleratore Super Proton Synchroton e ha lo scopo di misurare il Branching Ratio del decadimento ultra raro $K^+ \to \pi^+ \nu\overline{\nu}$ per fornire una prova rigorosa del Modello standard. Dal momento che il valore previsto dal modello standard è molto preciso, la misura di questa quantità rappresenta un ottimo modo per indagare l'esistenza di nuova fisica, o in caso di accordo con il Modello Standard (SM) per migliorare le attuali conoscenze del parametro $|V_{td}|$ della matrice CKM. L'uso di un fascio ad alta intensità si traduce in un rate di eventi di circa 15 MHz, valore talmente

elevato che rende impossibile la memorizzazione dei dati su disco senza una riduzione molto selettiva. Tre livelli di trigger sono stati realizzati in modo tale da ridurre a 10 KHz il rate di dati da inviare alla PC farm.

Una introduzione sul mondo della fisica delle alte energie, un cenno a come si sono evoluti i sistemi di trigger e acquisizione dati, una panoramica sullo stato dell'arte e una introduzione al sistema integrato di trigger e acquisizione dati completamente digitale proposto sono descritti nel capitolo 1.

Nel capitolo 2 viene descritto l'apparato sperimentale di NA62, composto da una parte a monte per la identificazione e la misura del $K^+$ nel fascio e una parte a valle dove vengono identificati i prodotti del decadimento.

La prima parte del mio lavoro riguarda il progetto e la realizzazione dell' hardware del sistema di trigger e acquisizione dati comune per la maggior parte dei rivelatori in NA62 e lo sviluppo del firmware ad esso associato. Il sistema integrato di trigger e acquisizione dati , in cui il trigger può utilizzare tutti i dati digitalizzati, è presentato nel capitolo 3.

La seconda parte del mio lavoro descrive il trigger standard di livello 0 (L0) di NA62 e gli studi eseguiti per realizzare un trigger innovativo di livello 0 basato sulle GPU. Il trigger standard L0 è descritto nel capitolo 4. Nello stesso capitolo viene descritto ampiamente il firmware di trigger sviluppato per il rivelatore RICH. L'utilizzo delle GPU in fisica delle alte energie, il trigger basato sulle GPU in NA62 e il firmware GPU-RICH sono descritti nel capitolo 5 .

# Summary of PhD Achievements

## Research Activity

My PhD activity was held at the National Institute of Nuclear Physics (INFN) in Pisa collaborating in the research project KLEVER (*KL Experiment to detect Very Rare decays*). KLEVER aims at using powerful programmable systems in the first stages of the data collection and selection process in particle experiments at accelerators, i.e. the use of hardware processors based on Field-Programmable Gate Arrays (FPGAs) and Graphic Processing Units GPUs. The goal was to develop prototypes in order to investigate the feasibility of data-acquisition and selection systems characterized by a reduced price and a limited obsolescence, as they might easily be updated later on using future technical advancements and without redesigning the system. In particular there was the intention to probe the performances of FPGAs and GPUs processors by building a system acting as an easily updatable test bench of the attainable capabilities of collection and selections of large amounts of data. This will allow to evaluate the present technological limits, which in turn represent the most important bottleneck for a high-precision physics experiment studying ultra rare decays of charged K mesons. The study of ultra rare channels of charged K mesons is currently only performed at CERN in Geneva (Switzerland), by the international collaboration NA62, and as a first application of KLEVER project the Trigger and Data AcQuisition (TDAQ)system for the NA62 experiment was designed. The NA62 TDAQ scheme has two essential differences from the schemes used currently in experiments of this type:

- the digitization is always enabled;
- the same data digitized are the inputs for L0 and L1 levels, as it will be for advanced stages of the trigger and possibly for storage;

## Courses

During my PhD I attended the following training courses:

- Recent Advances in Sensors and Embedded Sys-tems for Automotive ADAS (Autonomous Driver Assistance Systems) - 4 credits

- Academic Writing and Presentation Skills for Engineering PhD students - 4 credits
- RF, Beam Instrumentation and Electrical Engineering for particle accelerators - 4 credits
- Short course on humanoid robots: modeling, planning and control - 5 credits
- Accelerate System Performance with ALTERA SoC - 5 credits
- Programma PHD PLUS 2015: “Creatività, Innovazione, Spirito Imprenditoriale” dell’ Università di Pisa - 8 credits
- Design Technologies for Embedded Multiprocessor Systems-on-Chip - 5 credits
- English for writing and presenting scientific papers - 2 credits
- Biosensors and Biochip: state-of-art and perspectives - 4 credits

## Research Periods in Qualified Institutions

During my PhD I spent part of my research at CERN in Geneva. The months when I was abroad were:

- November 2013;
- July, October, November, December 2014;
- February, March, May, June, July, December 2015;
- April, May, September 2016;

## Project Title and Topology

KEVLER is a research project financed by the Ministry of Education, University and Research (MIUR), with the Decree of 23 October 2012 n. 719 PRIN 2010-2011, Area 2 2010Z5PKWZ Code, and within the sectors Physical Sciences and Engineering and Fundamental constituents of matter.

## Publications

### International Journals

1. E.Pedreschi et Al.: "A high resolution TDC based board for a fully digital trigger and data acquisition system in the NA62 experiment at CERN", IEEE Transaction on Nuclear Science, June 2015.

2. E.Pedreschi et Al.: “A fully digital trigger and data acquisition system for the NA62 kaon factory at the CERN SPS”, IEEE Transaction on Nuclear Science, submitted and under review.

**International Conferences/Workshops with Peer Review**

1. E. Pedreschi: “TDCB and TEL62 status”, TDAQ Working Group Meeting, February 2015.

2. E. Pedreschi et Al.: “The TEL62: a Real time Board for the NA62 Trigger and Data Acquisition. Data Flow and Firmware Design”, Real Time Conference (RT), 2014 19th IEEE NPSS, May 2015.

3. B. Angelucci, E. Pedreschi et Al.: “The FPGA based Trigger and Data Acquisition system for the CERN NA62 experiment”. JINST 9.01 (2014), p. C01055. doi: 10.1088/1748-0221/9/01/C01055.

4. Jacopo Pinzino, E. Pedreschi et Al.: “The CERN NA62 experiment: Trigger and Data Acquisition”, PoS TIPP2014 (2014), p. 200.

5. E. Pedreschi: “TEL62 TDCB hardware status”, NA62 Collaboration meeting 2015, Prague.

**Others**

1. NA62 Collaboration: “2015 NA62 Status Report to the CERN SPSC”, March 2015

2. V. Duk: “Study of the rare decay $K^+ \to \pi^+\nu\overline{\nu}$ at the NA62 experiment”, Proceedings, 18th International Seminar on High Energy Physics (Quarks 2014). url: http://quarks. inr.ac.ru/2014/proceedings/www/p2/Duk.pdf.

3. B. Angelucci et Al.: “Prospects for $K^+ \to \pi^+\nu\overline{\nu}$ Observation at CERN in NA62”, PoS DIS2015 (2015), p. 217.

4. V. Duk et Al.: “LFV and exotics at the NA62 experiment”, J. Phys. Conf. Ser. 556.1 (2014), p. 012067.

5. V. Duk et Al.: “Recent Results From NA62”, Proceedings, 16th Lomonosov Conference on Elementary Particle Physics: Particle Physics at the Year of Centenary of Bruno Pontecorvo. 2015, pp. 309–312.

6. R. Fantechi et Al.: “The NA62 experiment at CERN: status and perspectives”, 12th Conference on Flavor Physics and CP Violation (FPCP 2014) Marseille, France, May 26-30, 2014. 2014. arXiv: 1407.8213 [physics.ins-det]. url: https://inspirehep.net/record/1309159/files/arXiv:1407.8213.pdf. /item V. Kozhuharov et A.: “NA62 experiment at CERN SPS” EPJ Web Conf. 80 (2014), p. 00003.

7. N. Lurkin et Al.: “The NA62 run control”, J. Phys. Conf. Ser. 556.1 (2014), p. 012074.

8. C.Parkinsonet Al.: “Precision tests of the Standard Model with kaon decays at CERN”, HQL2014 (2014), p. 027.

9. M. Pepe et Al.: “Rare and forbidden kaon decays at NA62”, EPJ Web Conf. 95 (2015), p. 03029.

10. A. Romano et Al.: “The kaon identification system in the NA62 experiment at CERN”, Proceedings, 4th International Conference on Advancements in Nuclear Instrumentation Measurement Methods and their Applications (ANIMMA 2015). 2015, pp. 1–8.
11. B. Wrona et Al.: “The kaon identification system in the NA62 experiment at CERN SPS”, JINST 9.12 (2014), p. C12048.

# List of Figures

# List of Tables

# Contents

CHAPTER 1

# Trigger and Data Acquisition system in High Energy Physics

## 1.1 Introduction

Particle physics is the branch of physics that studies the most basic constituents of matter and their interactions. Modern particle physics research is focused on subatomic particles, i.e. particles with dimensions and mass smaller than atoms, including atomic constituents such as electrons, protons and neutrons and particles produced by radiative and scattering processes, such as photons, neutrinos and muons. Since many elementary particles do not occur under normal circumstances in nature, to allow their study they are created and detected by means of high energy collisions of other particles in particle accelerators: therefore, particle physics is often referred to as High Energy Physics (HEP). The purpose of particle physics is to investigate the fundamentals of matter in order to address unanswered key questions about nature and origin of the Universe, such as symmetries in physical laws, the origin of mass, the nature of dark matter and dark energy, possible existence of extra dimensions and so on; the final, ambitious objective would be the creation of a general theoretical model that is able to describe and explain all physical phenomena in an unified and coherent vision.
The main instruments for High Energy Physics are therefore particle accelerators, large and complex machines that produce beams of particles and provide them with the high energies needed for HEP experiments. Accelerators typically employ electric fields to increase kinetic energy of particles and magnetic fields to bend and focus the beam, which is then collided against a fixed target or with another particle beam: the high energy collision produces the new particles and events that must be detected and studied. Beside the use in particle physics, the applications of accelerators nowadays span from industry (e.g. ion implantation in electronic circuits) to medicine (radiotherapy), with

different ranges of energy for the different fields application. Current accelerators for High Energy Physics work in the GeV and TeV energy range (referring to the energy provided to particle beams) and typically treat beams of electrons, hadrons [1] and heavy atomic nuclei; the structure can be linear (LINAC, LINear ACcelerator), with the particle beam traveling from one end to the other and colliding against a fixed target, or circular (cyclotrons, synchrotrons), with the beams traveling repeatedly around a ring, gaining more energy at every loop and colliding with other beams running in opposite direction (see figure 1.1, 1.2 and 1.3)

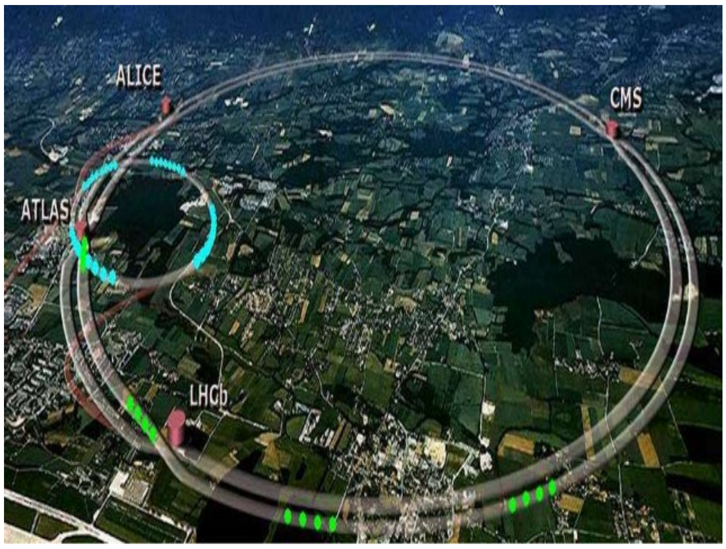


**Figure 1.1:** *Aerial view of the Large Hadron Collider at CERN, that is an example of circular accelerator*

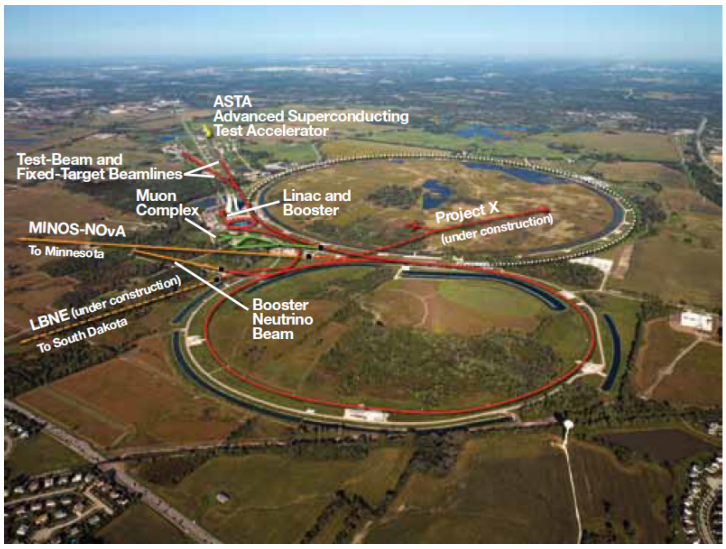


**Figure 1.2:** *Aerial view of the Tevatron at Fermilab, that is an example of circular accelerator*

[1]Hadrons are the subnuclear particles that are subject to the strong force, and are constituted by quarks. The family of hadrons is then divided in two subsets: baryons, comprising neutrons and protons, and mesons, including pions and kaons.

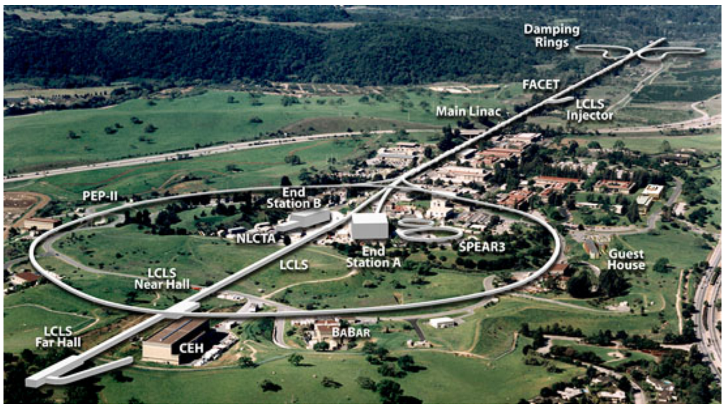


**Figure 1.3:** *Aerial view of the Stanford Linear Accelerator Center (SLAC), an example of LINAC*

The major research centers for High Energy Physics nowadays include, among the others:

- The European Organization for Nuclear Research (CERN), located near Geneva, Switzerland: its main facilities included LEP (Large Electron Positron collider), which was dismantled in 2001 and substituted with LHC (Large Hadron Collider) that is now the world's most energetic accelerator;
- the Fermi National Accelerator Laboratory (Fermilab), located near Chicago, USA;
- he Stanford Linear Accelerator Center (SLAC), located near Palo Alto, USA: it hosts the longest linear accelerator in the world, colliding electrons and positrons;
- the INFN (Istituto Nazionale di Fisica Nuleare) center in Frascati, Italy, that hosts DAFNE (Double Annular ring For Nice Experiments) a circular accelerator for the collision of electrons and positrons.

High Energy Physics experiments thus consist in colliding particle beams in accelerators and studying the results of the collisions by means of particle detectors that surround the interaction point. Particle detectors are devices used to track and identify high-energy particles produced in collisions, also measuring their attributes like momentum, charge and mass. A particle detector is typically made up of different layers of sub-detectors, each specialized in revealing and measuring different particles and properties of particles: normally the innermost layer (i.e. the nearest to the interaction point) is the tracking device, that has the task of revealing the paths of electrically charged particles through the trails they leave behind; in the outer layers calorimeters are typically placed, that measure the energy lost by particles that go through them. To help identify the particles produced in the collisions, the detector usually includes a magnetic field that bends the path of charged particles: from the curvature of the path, it is possible to calculate the particle momentum which helps in identifying its type. Particles with very high momentum travel in almost straight lines, whereas those with low momentum move forward in tight spirals.
To record and analyze events produced by collisions in an experiment, information about particles detected by sensors in the detector are converted into electric signals, which are then collected by dedicated electronic components embedded in the detector

and located in close contact with the sensors themselves: these devices, usually called Front-End (FE) electronics, deal with the proper conditioning of signals (e.g. amplification, shaping, buffering, analog to digital conversion) and their transmission to remote data acquisition systems that perform data analysis and storage. However, some means is needed to reduce the amount of data that must be transferred from the detector to the remote system, that is extremely large in every HEP experiment. In fact, to increase the probability of occurrence of rare, interesting events, the number of interactions per second is made very high (a typical order of magnitude is billions of particle interactions per second); a measure of collision rate is the so-called luminosity, which is usually expressed in $cm^{-2}s^{-1}$ and for a two-beam collider is defined as the number of particles per second in one beam multiplied by the number of collisions per unit area in the other beam at the crossing point. Collision rates of this order of magnitude produce amounts of raw data that range from tens of terabyte to a petabyte per second, which is beyond the possibility of any data acquisition and storage system. Therefore, since the interesting events are a very small fraction of the total, the total amount of data is filtered by means of a trigger system: raw data are temporarily buffered in the FE electronics while a small amount of key information is used by trigger processors (located at various hierarchical level inside the detector and in the remote elaboration center) to perform a fast, approximate calculation and identify significant events: the result of this processing is a trigger signal that is sent back to FE electronics to command a data readout, i.e. the transferring of a selection of the buffered data towards the remote system. This way, the amount of data to be transferred is reduced to rates that can be handled by the readout system (a typical order of magnitude is hundreds of MB/s from each FE device), and only the interesting events are selected.
A typical control and readout system for a HEP experiment can be schematized as in 1.4:

Signals generated by the interaction with sensors of particles produced in the beam collisions are handled by Front-End electronics embedded in the detectors and transferred to remote data acquisition (DAQ) systems, that are placed far away from the experiment area to keep them in an environment that is free from the intensive levels of radiation that are present in the proximities of the interaction point: typically the transfer is carried out by means of electrical links for a first stretch inside the detector, and then through optical links that allow to cover the long distances (hundred of meters) from the experiment area to the remote DAQ system, and provide the large bandwidth needed (up to tens of Gbit/s). A subset of the transferred data is used to perform trigger calculation, and the generated trigger command is sent back to FE electronics along with timing (clock) and control signals by a remote control system, also called TTC (Timing, Trigger and Control) system, that manages the configuration and monitoring processes in the Front-End electronics.
High Energy Physics experiments constitute a very challenging application for electronics, since the equipment must deal with large amounts of data and high data rates, with tight timing and data integrity constraints and operate in an environment that is intrinsically hostile due to the high levels of radiation. Typical requirements for detector electronics and data transmission links in a HEP experiment are:

- radiation hardness: electronic devices and systems must tolerate high levels of

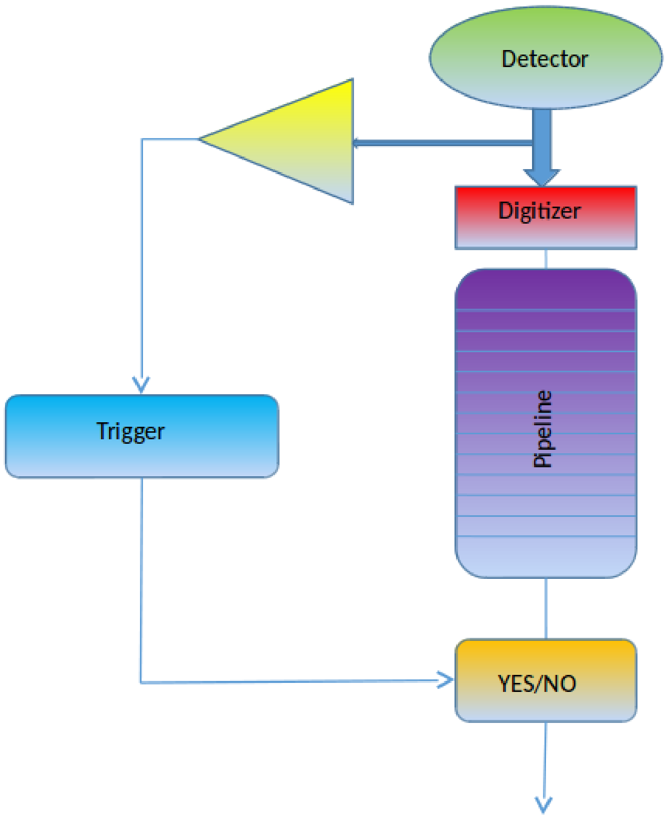


**Figure 1.4:** *Typical architecture of the control and readout system for a HEP experiment*

ionizing radiations and the associated Total Dose Effects (e.g. threshold voltage drift and sub-threshold current increase in MOS devices) and Single Event Effects (e.g. Single Event Upset in flip- flops and SRAM cells);

- small size: the space available for devices and cabling inside a particle detector is usually very limited due to the large amount of different components (readout and control systems, cooling systems, mechanical structures and so on) that must be integrated in a small area around the interaction point; additionally, bulky equipments are undesired because any non-sensor material interferes with the measure by deflecting and absorbing the particles that must be detected (the amount of material surrounding the interaction point, characterized with the radiation thickness of each component/layer, is usually referred to as material budget);

- low power dissipation: due to the high concentration of electronic equipment inside the detector, power density is a major issue because it dictates the cooling requirements; cooling system is a critical aspect in HEP experiments because it complicates the material budget and the mechanical requirements;

- constant trigger latency: the trigger signal must be delivered to all Front-End devices with a fixed and known latency, to command the readout of selected data thus allowing precise reconstruction of significant events;

- capability of handling high data rates: readout electronics must be able to elaborate and transfer large amount of data in a limited time to allow a continuous data flow during the running of the experiment, with minimal loss of information; for the same reason, adequately high bandwidth in electrical and optical links are required;
- data integrity: appropriate methods must be employed in transmission links to protect data against transmission errors, and in storage elements to deal with data corruption due to radiation.

## 1.2 Historical approach to DAQ system

The search for new physics and rare phenomena with low cross sections has demanded progressively higher beam interaction rates and luminosities in order to maintain acceptable event rates and thus collect the required number of events during the lifetime of an experiment. Greater beam energies were also required as experiments targeted particles and interactions that required higher energies. These needs stimulated advances in a number of engineering fields, such as electronics, microwaves, superconductivity, magnet design, vacuum, and cryogenics. This has enabled the construction of modern accelerators. The combination of high luminosity and low cross section for interesting events implies that these events are masked by a large background of uninteresting, but higher cross section, interactions. From the data acquisition and trigger system point of view this implies a number of problems:

- Event reconstruction becomes more complex, as interesting tracks and particles become only a small fraction of the observed events;
- High luminosity can cause multiple events per beam interaction, complicating trigger algorithms and making event reconstruction more difficult
- Channel occupancy in the subdetectors (understood as the percentage of beam interactions in which a channel carries data) increases and so DAQ system throughput increases;
- Radiation levels in the vicinity of the detectors increase, posing radiation-hard or radiation-tolerant requirements for the front-end electronics;

But luminosity and cross section are not the only basic parameters to take into consideration when studying the evolution of HEP experiments from the DAQ system point of view. To understand modern DAQ systems, the ever growing beam interaction rate and detector channel count must also be taken into account. In the late sixties just a hundred data channels were read at low rates (a few hertz), and this meant a single minicomputer could handle the readout. Nowadays, large experiments have millions of channels read at megahertz rates. Today's DAQ systems feature distributed processing, complex trigger systems for event filtering, switch networks, and PC-based computer farms.

### 1.2.1 HEP in the 1960s and 1970s

In a typical HEP experiment of the late sixties and seventies, the front-end electronics were read out by a single minicomputer. This architecture (see figure 1.5) lacked parallelism and only allowed data rates of kilobytes per second. A common trigger signal distributed to all sub-detectors was used for system synchronization and timestamping. Analog signal path delays were equalized with tens of meters of cable per channel. Signals coming from multiwire chamber anodes were amplified and converted into coordinates via time-to-digital converters. Scintillators were connected to counters. Data from each front-end module (event fragment) were read out by a minicomputer at a fixed time after the trigger signal to ensure data availability. Event fragments consisted basically of raw physics data with little or no formatting. Both the read out bus and the front-end electronics used non-standard interconnects and modules, generating confusion and inefficiency.

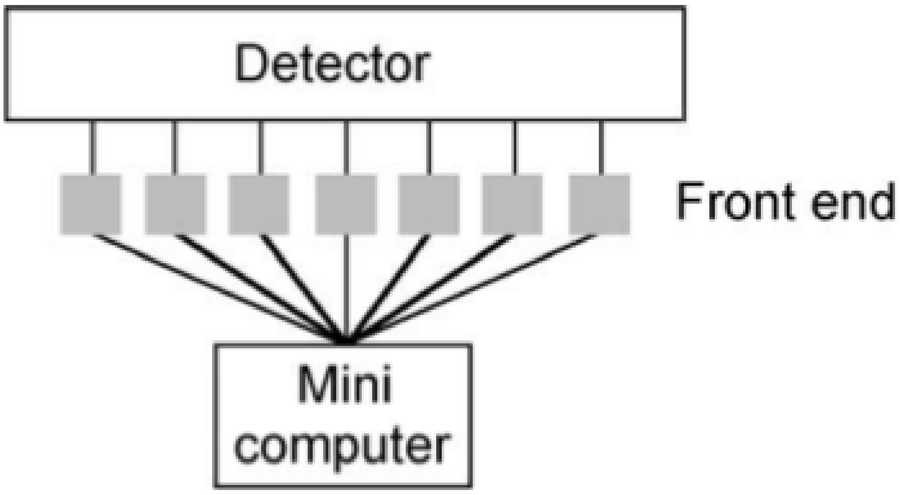


**Figure 1.5:** *DAQ architecture in the seventies*

### 1.2.2 DAQ and trigger systems in the 1980s

The following trends can be observed during the eighties:

- An increase in detector resolution increased the number of channels. The study of rare events made it necessary to increase the event rate to achieve the required statistics. The combination of these two parameters caused an increase in DAQ system bandwidth requirements to several MByte/s;

- Permanent storage bandwidth was around 100 KByte/s, thus requiring a data rate reduction by filtering out non interesting events in the so called trigger systems. Event filtering was carried out in hardware at the front- end level and in software at the main read out computer, leading to multilevel trigger systems. This scheme required more complex data formats in the DAQ system for event timestamping and synchronization;

- The above mentioned increase in sub-detector complexity justified the DAQ being partitioned into several DAQ systems which could work autonomously during development and commissioning phases, as well as for calibration and tests during operation. All the DAQ partitions (which grouped all channels from a sub-detector) could, of course, work together in a single DAQ system;

Typical DAQ architectures had a hierarchical tree-like structure. A good example can be found in the CERN's ALEPH DAQ System. ALEPH was designed with a three-level trigger system. The first two levels were implemented in hardware and reduced the rate from 50 KHz to 500 Hz (Level 1); and down to 10 Hz (Level 2) which is the DAQ rate. Level 3 reduction takes place in the Main Readout Computer and this lowers the rate to 1 Hz; thus matching bandwidth to tape storage capabilities (around 100 KBytes/s).

### 1.2.3 State of the art

During the nineties several ideas and trend were introduced for DAQ and trigger architectures, these were implemented in later years and are currently the base of the running experiments. In particular:

- Collecting data techniques were discussed and push architectures were studied;
- Expensive computing facilities were replaced by PC-based computer farms running Linux, and inter-connected via network technologies such as Ethernet;
- Several experiments, normally located on the same accelerator, shared the same DAQ and trigger infrastructure, to reduce costs and developing time. This is the case of all the LHC experiments ATLAS,CMS,LHCb,and TOTEM [4, 29, 41, 47, 48];

The basic idea for the trigger and DAQ systems is shown in figure 1.6
In detail:

- Digitized data are stored in distributed pipelines, normally located in custom ASIC near the detector. The pipeline depth varies depending on the detector but is normally on the order of hundreds of samples;
- The same custom ASIC is capable of generating a fast subset of the detector signals, significative of the full set. This can be for example a logical AND of the discriminated signals. These fast signals are sent to the low level trigger systems;
- The trigger system receives the subset of data, quickly analyze it and generates the event accept/reject signal;
- This signal is sent back to the ASIC front end, with a fixed delay ($\mu$s) so that the event at the output of the pipeline is related to this same signal;

An example of this particular architecture is currently in use in the LHC experiment TOTEM. I've collaborated at the TOTEM DAQ design several year ago.

#### TOTEM DAQ

TOTEM readout electronics [12] is shown in picture 1.7

The main component is the VFAT2 chip which is a "trigger and tracking front-end ASIC" (250 nm CMOS technology Rad hard). It provides both trigger and data

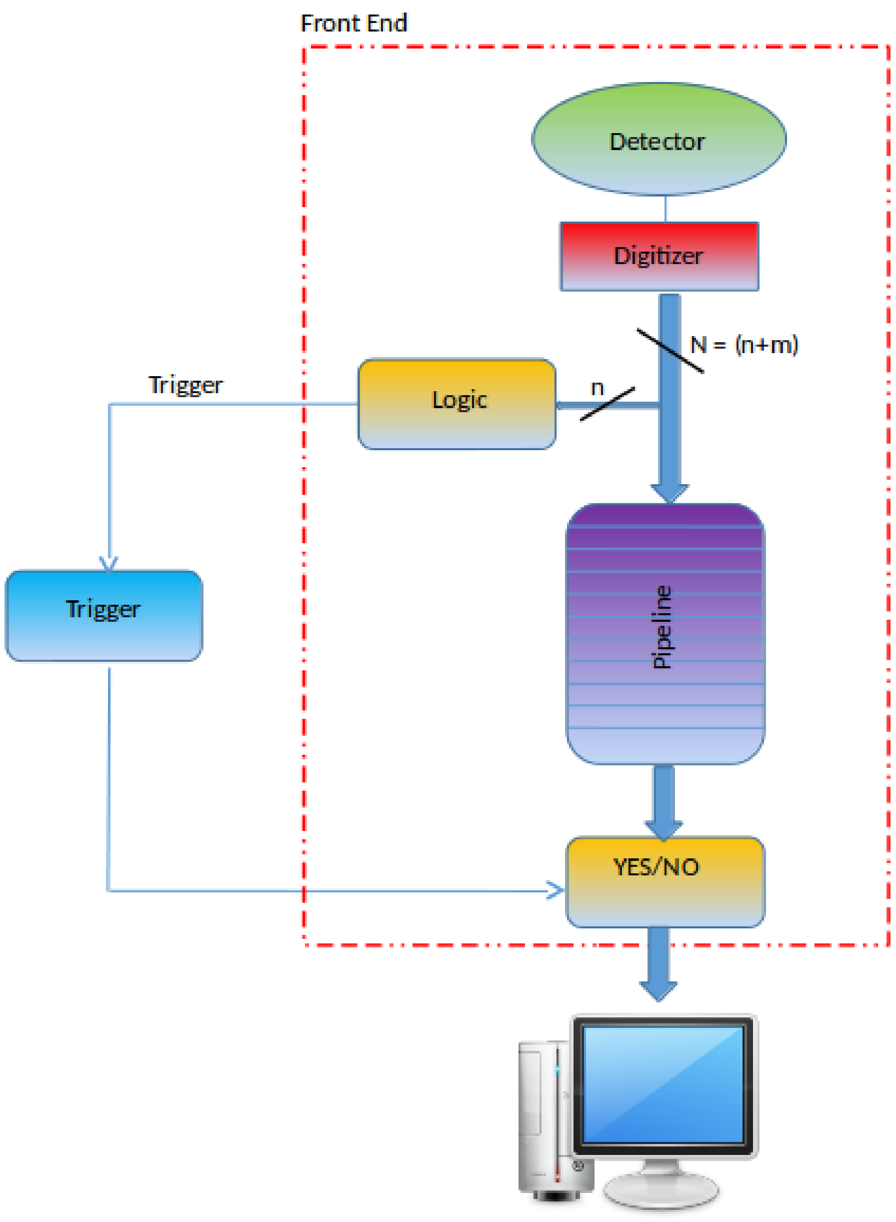


**Figure 1.6:** *Trigger and Data Acquisition architecture for LHC experiments schematic view*

acquisition functions. It has 128 analog input channels each of which are equipped with a very low noise pre-amplifier and shaping stage plus comparator. Signal discrimination on a programmable threshold provides binary “hit” information which passes through a synchronization unit and is then stored within a SRAM (SRAM1) until a trigger is received. The same discriminated signals can be combined in groups of programmable "fast OR", to generate trigger informations. These are immediately sent to the first level trigger systems board, that combine all the received "fast OR" signals to generate the L1 trigger accept signal, which is sent back to the VFAT2 chips, with a fixed latency, so that the corresponding word can be extracted from the SRAM1 and copied to SRAM2

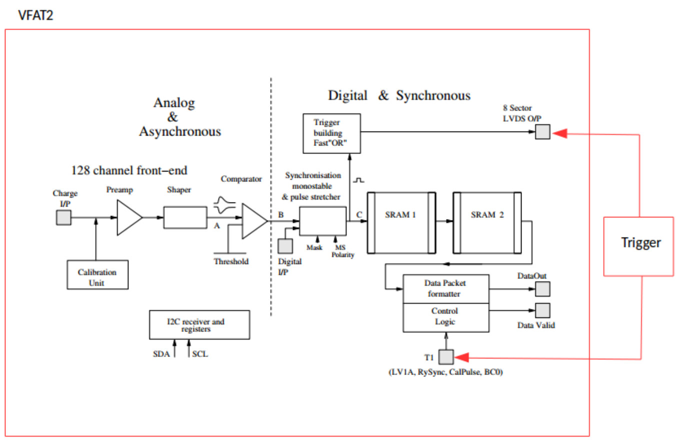


**Figure 1.7:** *Totem readout architecture*

(adding in the meantime a timestamp).

## 1.3 The integrated fully-digital system approach

The idea is to realize an integrated and fully-digital trigger and data acquisition system, instead of a custom distributed system in which only a part of the digitized data can be used to generate the trigger signal. The project aims at using powerful programmable systems in the first stages of the data collection and selection process in particle experiments at accelerators, i.e. the use of hardware processors based on Field-Programmable Gate Arrays (FPGAs) and Graphic Processing Units GPUs. The FPGAs can be placed at the front-end stage of detectors, immediately after digitization, thus allowing data processing at an earlier stage of the acquisition and trigger chain (3).

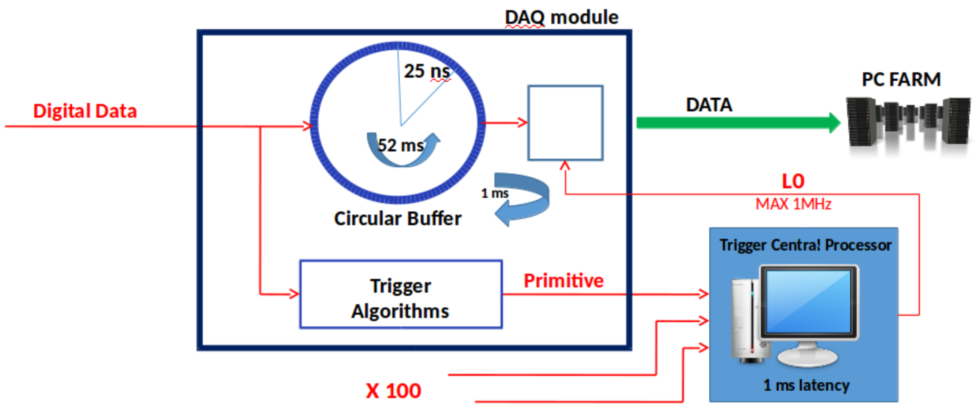


**Figure 1.8:** *Trigger and DAQ integrated system architecture proposed*

TDAQ scheme proposed has two key differences from the schemes used currently in experiments of this type (1.2.3):

- The digitization is always enabled

- The same data digitized are the inputs for L0 and L1 levels
- All digitized data are available and can be analyzed in real time by specific algorithms, and these algorithms can decide whether to accept or reject the event with a fixed latency (ms);
- The trigger algorithms can be more complex and easily reconfigurable (1.2.3);

The electronic modules of the TDAQ are directly part of the trigger system and are designed to allow simultaneous processing of data for acquisition and for trigger purposes, with no dead time introduced by the second. Adequate data buffers must be foreseen for both type of processing. In particular the digitized analog detector signals are stored in a circular buffer (see Fig.1.8). If the trigger logic decided that the event should be accepted the data would be extracted from the circular buffer and sent to a PC-Farm for a further selection and subsequent storage.

The experiments with a relatively large number of channels and high event rates, such as those in High Energy Physics(HEP) are good candidate for this innovative architecture. In fact, the HEP experiments are quite far from reaching the goal of implementing their entire trigger and DAQ system on commodity processors (so-called "trigger-less"), because the size of the required computer farms would be in most cases impractically large. A reliable trigger and data acquisition system and an efficient online selection of candidates represents an important issue for HEP experiment because of the large reduction to be applied on data before tape recording. On the other hand, a loss-less data acquisition system is mandatory to avoid adding artificial detector inefficiencies when vetoing background particles; this last requirement is less common in standard readout and trigger systems. For the above reasons the detectors and the TDAQ systems are integrated in a completely unified digital system where the trigger is structured in a three level system, in order to reduce the event rate from dozen of MHz to tens of kHz. The first level (L0) will be completely hardware based while the other levels (L1 and L2) will be based on software: the L1 decision is taken on single subdetector reconstructed quantities, while the L2 decision is taken on the fully reconstructed event with high resolution.

To demonstrate the feasibility and the reliability of this system the TDAQ for the CERN NA62 experiment was designed, realized and tested.

### 1.3.1 NA62 experiment

The NA62 experiment is located in the CERN North Area SPS extraction site and aims at measuring the Branching Ratio of the ultra-rare decay $K^+ \rightarrow \pi^+\nu\overline{\nu}$, as a highly sensitive test of the standard model (SM), collecting about 100 events in two years of data taking. Since the detection of this process is very challenging due to the smallness of the signal and the presence of a very sizable background, a very low DAQ inefficiency (below ) is a key point of the experiment. The intense flux of NA62 dictates the need for a high-performance triggering and data acquisition system in order to minimize dead time and maximize data collection. A unified trigger and data acquisition (TDAQ) system was designed in order to address such requirements in a simple and cost-effective manner. The system was implemented with a single hardware trigger level (called L0) with a maximum latency time of 1 ms. With an estimated 13 MHz rate of decays in the detector, the L0 hardware trigger maximum rate was chosen to be 1 MHz In NA62 TDAQ the hardware L0 trigger is performed by evaluating conditions on the same complete set of digitized data which is eventually readout. NA62 TDAQ system is rather unique in allowing such fully-digital flexibility on this scale, in which any information available from the detector can be used to be triggered on. This feature is particularly important in view of a constant evolution of trigger. After L0 data is moved to PCs and two further software-based trigger levels are implemented. A central trigger processor will asynchronously match fine-time L0 trigger primitives generated by a few fast sub-detectors, and dispatch a (synchronous) L0 signal to every board through the same system that distributes the clock to all detectors. The L0 trigger primitives are constructed in the same board (TEL62) in which the data are stored to wait for the trigger decision.

CHAPTER 2

# NA62 apparatus overview

The NA62 experiment is placed in the ECN3 zone in the CERN North Area Intensity Facility (see figure 2.1) and it uses the Super Proton Synchrotron accelerator extraction line already used by the NA48 experiment.

The detectors used to detect the kaon decay products are spread along a 170 m long region starting about 100 m downstream of the target. The fiducial decay region is 60 m long and starts 105 m after the target. The largest detectors have an approximately cylindrical shape around the beam axis with a diameter up to about 2 m and down to 10 cm in order to let the very intense flux of undecayed beam particles pass through without affecting the active area.

The experimental setup (see figure 2.3) consists in:

- the Kaon TAGger (KTAG) that uses a Čerenkov Differential counter with Achromatic Ring focus (CEDAR) to identify $K^+$ in the hadron beam;
- the GigaTracKer (GTK), a tracking detector for beam particles upstream the decay region and composed of three silicon micro-pixel stations and a set of achromatic magnet;
- the CHarged ANTIcounter (CHANTI) a set of six stations of plastic scintillator detectors useful for vetoing charged particles generated in the last station of the GTK.
- a photon veto system that guarantees an angular coverage from 0 mrad up to 50 mrad through 12 Large Angle Veto (LAV), a Liquid Krypton electromagnetic calorimeter (LKr), an Inner Ring Calorimeter (IRC) to cover the annular region

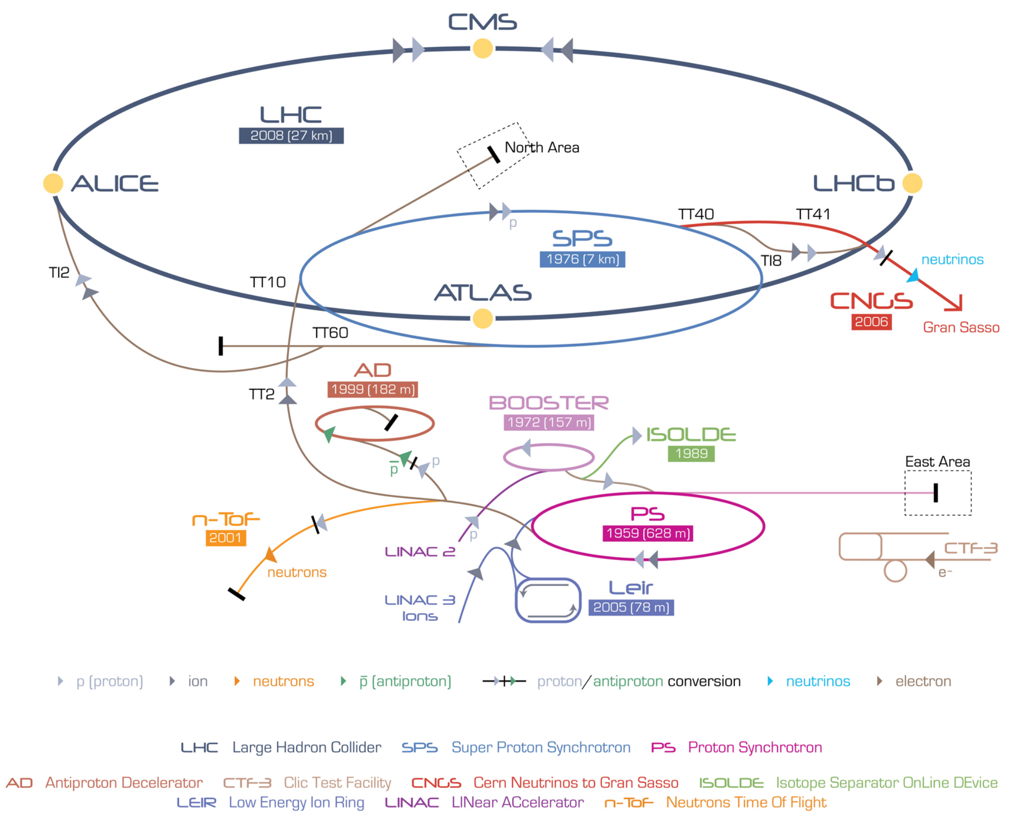


**Figure 2.1:** *Schematic view of the CERN accelerator complex (not to scale). The NA62 experiment is located in the North Area SPS extraction line.*

around the beam and a Small Angle Calorimeter (SAC) located at the end of the experimental hall to cover the small angle region;

- the STRAW magnetic spectrometer for charged particles originating in the decay region;
- a Ring Imaging Čerenkov (RICH) to separate $\pi^+$ from $\mu^+$ in the momentum region between 15 GeV/$c$ to 35 GeV/$c$, which can also measure particle direction and velocity;
- the Charged Hodoscope (CHOD), a segmented plastic scintillator detector conceived for triggering purpose.
- a muon veto system composed of three stations.

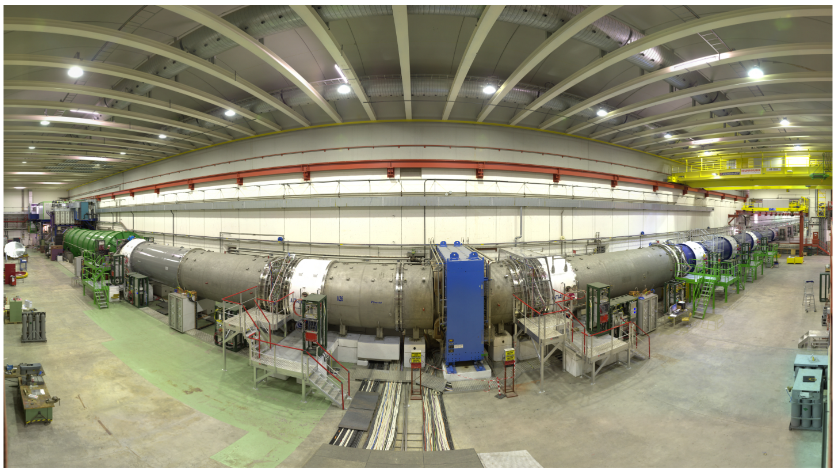


**Figure 2.2:** *Gigapanorama of the NA62 cavern, in which all detectors can be roughly seen.*

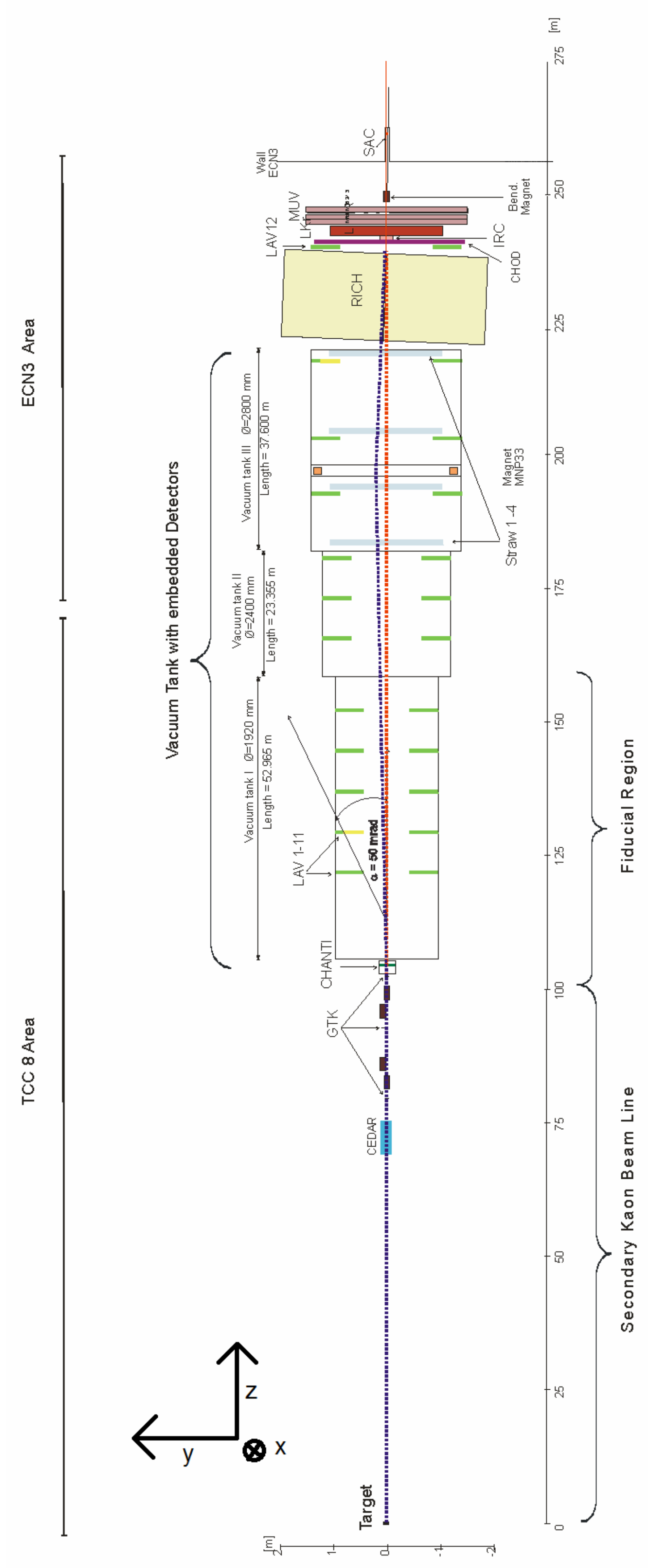


**Figure 2.3:** *Schematic longitudinal view of the NA62 experimental setup.*

## 2.1 The beam line

It is convenient to use a high energy proton beam in order to maximize the production of positive kaons by beam interactions on a beryllium target [33].

The highest kaon production is achieved at $P_K/P_p \simeq 0.35$, where $P_p$ is the central proton beam momentum and $P_K$ is the momentum of the produced kaons. Furthermore, the use of high energy kaons increases the detection efficiency of most sub-detectors. Due to these considerations, a central beam momentum 75 GeV/$c$.

The choice of positive kaons is due to the ratio particles abundances in a hadron beam produced by 400 GeV/$c$ protons [32]:

$$\frac{K^+}{K^-} \simeq 2.1 \tag{2.1}$$

$$\frac{K^+/\pi^+}{K^-/\pi^-} \simeq 1.2 \tag{2.2}$$

Tab. 2.1 shows the different components of the beam.

| Momentum | $75 \pm 0.9$ GeV/$c$ |
|---|---|
| Rate | 750 MHz |
| Composition | $70\%$ $\pi^+$<br>$23\%$ $p^+$<br>$6\%$ $K^+$<br>$1\%$ other |

**Table 2.1:** *NA62 beam composition*

## 2.2 Detectors upstream the decay region

### 2.2.1 KTAG

One disadvantage of high-energy beams is that kaons cannot be efficiently separated from other beam particles. So the detection of kaons before decay is a crucial aspect for the experiment. A detector called Kaon TAGger (KTAG) was built to identify the kaons in the beam and to measure their time with good resolution. The KTAG is a NA62 upgrade of a ChErenkov Differential counter with Achromatic Ring focus (CEDAR) [**?**] [**?**]. A schematic view of a standard CERN SPS CEDAR is shown in figure 2.4.

The CEDAR is a steel vessel of 55.8 cm external (53.4 cm internal) diameter and 4.5 m length, was used at CERN since the early '80s for SPS secondary beam diagnostics, and is designed to identify particles of a given mass by making the detector blind to the Čerenkov light produced by particles of different masses. The Čerenkov angle of the light, emitted by a charged particle traversing a gas of a given refraction index *n*, is a function of the gas pressure, the beam momentum and the mass of the particle. The CEDAR is filled with nitrogen gas ($0.03X_0$), whose refractive index *n* is set by

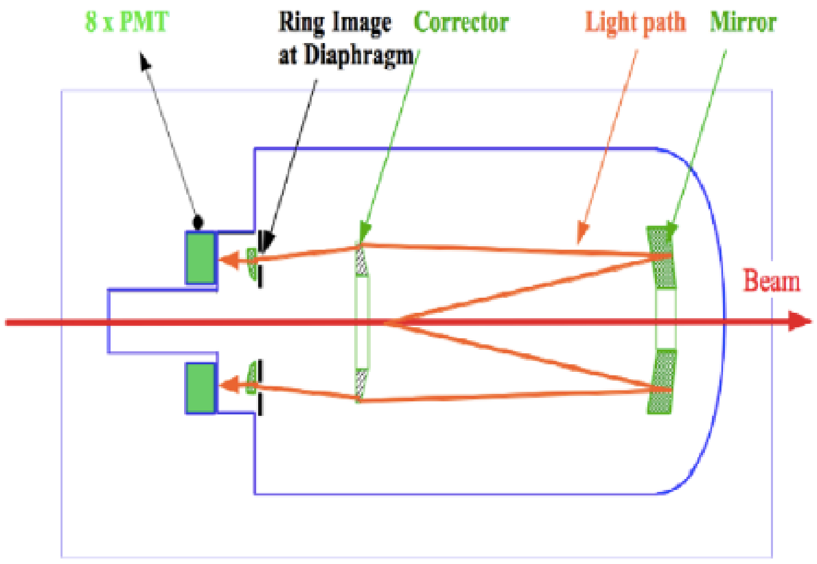


**Figure 2.4:** *Schematic layout of the optical system located inside the CEDAR.*

choosing the gas pressure (see figure **??**), for the kaon mass and the beam momentum (75 GeV/$c$). The CEDAR could be even filled with hydrogen to minimise material on the beam line, and hence reduce multiple Coulomb scattering.

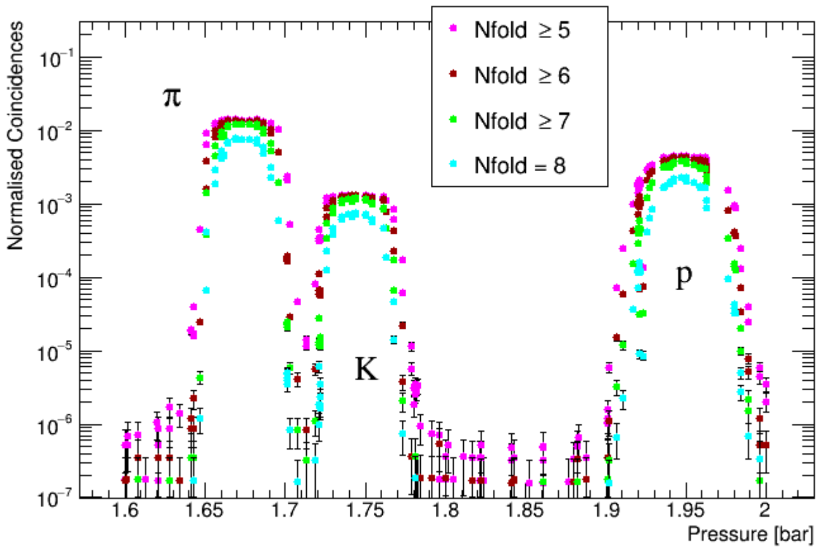


**Figure 2.5:** *Pressure scan, done during the 2015 run, with different requirerments on the number of sectors coincidences: the 1st peak corresponds to the pion peak, the 2nd to the kaon peak, and the 3rd one to the proton peak.*

At the end of the CEDAR vessel, a spherical mirror reflects the Čerenkov light onto a ring-shaped diaphragm of 100 mm radius, located at the vessel entrance. The aperture width of the diaphragm is adjustable to optimize the selection of the kaons. A chromatic corrector lens, designed to match the dispersion curve of the gas, is positioned between the mirror and the diaphragm: it ensures that light of all wavelengths arrives at the same radius on the diaphragm plane. After the diaphragm, 8 photomultipliers are placed behind 8 annular slits to detect the light. Light from other beam components hits the diaphragm plane at a different radius and does not pass through the aperture and in this way it does not contribute to the detector rate.

The upgrade of the CEDAR, the KTAG, was built to cope with the challenging 45 MHz kaon rate and to achieve the required time resolution. The KTAG replaced the original 8 photomultipliers with 384 divided in 8 sectors with an average rate of about 4 MHz on a single PMT (see figure **??**).

The efficiency in kaon tagging required for the KTAG is above 95% with a kaon time resolution of the order of 100 ps, and the pion mis-identification probability has to be below $10^{-3}$. The front-end electronics system is based on 8 boards (one per sector) using 8 ultra fast NINO amplifier/discriminator chips [22], to benefit from the fast photomultiplier response. The CEDAR uses the common TDC-based readout system TDCB+TEL62 described in chapter 3.

### 2.2.2 GTK

The GigaTracKer (GTK) detector provides precise measurement of angle, momentum and time of the crossing particle.

In order to limit hadronic interactions and to preserve the beam divergence, the GTK is composed of three station (3 is the minimum number of station to have a spectrometer) for a total thickness less than $0.5X_0$ [32]. Each station contains $18000$ $300 \times 300$ $\mu$m$^2$ silicon micro-pixels 200 $\mu$m thick, bump-bonded to 10 readout ASIC chips 100 $\mu$m thick. The three stations of the GTK are mounted inside the vacuum tank preceding the decay region, and they are interlaced with 4 achromat magnet pairs as shown in Fig 2.7.

### 2.2.3 CHANTI

The reduction of accidental backgrounds to a level of $10^{-11}$ is a crucial point for the experiment. The purpose of the CHANTI is to detect charged particles due to inelastic interactions between beam and collimator or upstream material at an angle larger than that allowed for the beam as they emerge from the last GigaTracker station. The CHANTI is made of six double-layer stations [32]. Each station is a $30 \times 30$ cm$^2$ square with a $90 \times 50$ mm$^2$ hole to allow the passage of the beam and each layer is composed of 24 (22) scintillator bars aligned to the $x$ axis ($y$ axis). A sketch of the CHANTI is shown in Fig 2.8.

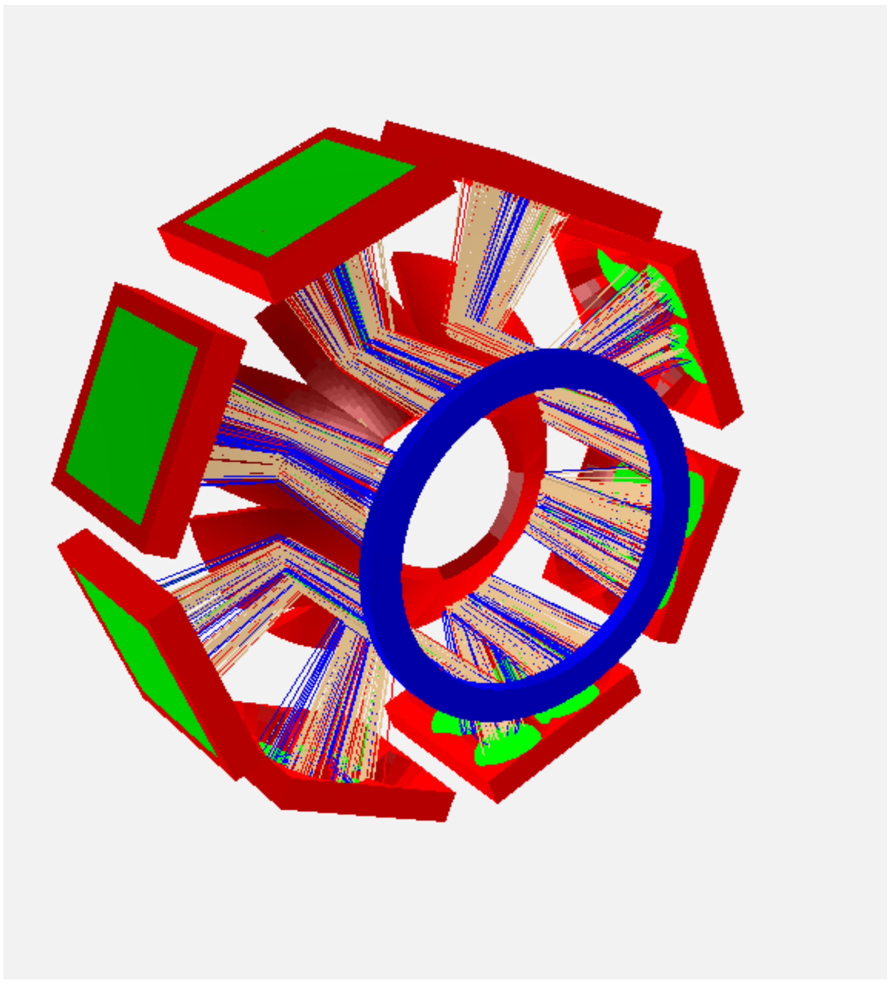


**Figure 2.6:** *A schematic view of KTAG*

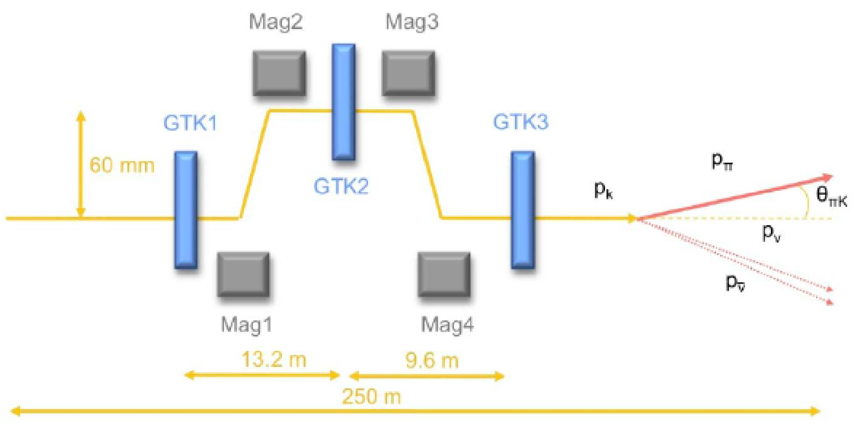


**Figure 2.7:** *Sketch of the Gigatracker stations*

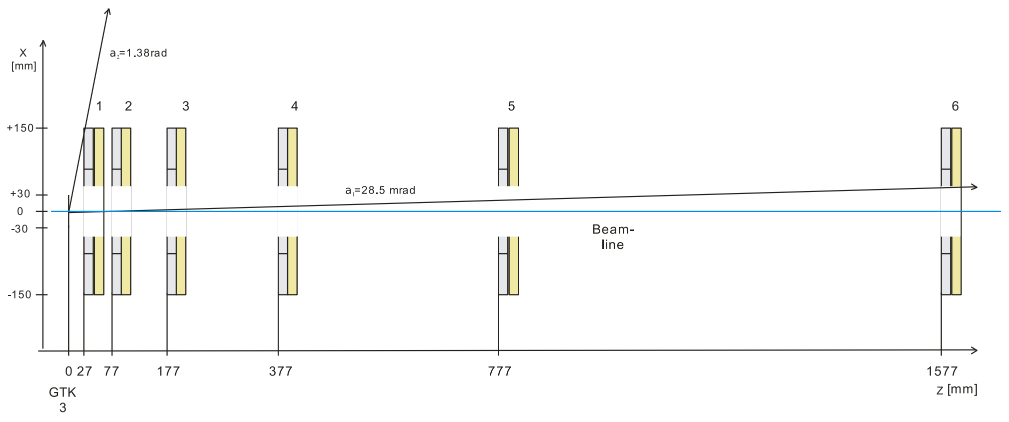


**Figure 2.8:** *Sketch of CHANTI stations on the beam line [32].*

## 2.3 Detectors downstream the decay region

### 2.3.1 Photon veto system

The photon veto system is needed to reduce the background events coming from kaon decays and interactions before the decay region. In order to efficiently reject the photons originating from $K^+ \to \pi^0\pi^+$ a photon veto system was developed, that ensures a rejection inefficiency lower than $10^{-7}$. The photon veto detector cover a 50 mrad angular range around the beam.

The photon veto system is composed by four sub-detectors that cover different angular region between $0 \div 50$ mrad

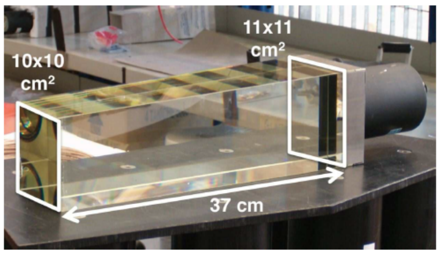
- Large Angle Veto (LAV), cover the angular region between 8.5 and 50 mrad.
- Liquid Krypton Calorimeter (LKr) covers angles between 1 and 8.5 mrad.
- Inner Ring Calorimeter (IRC) and Small Angle Calorimeter (SAC) cover the inner region, from about 0 to 1 mrad.

#### LAV

The Large Angle Vetoes are 12 stations (LAV1-12, see figure 2.9): the first eleven LAV are installed in the NA62 vacuum tank, while the last one is located in air outside of the tank between the RICH and the CHOD.

LAV stations are made of rings of lead-glass blocks (see figure 2.9) that were recovered from electromagnetic calorimeter barrel of the OPAL experiment [**?**]. Each block is a trapezoidal Čerenkov counter exploiting lead-glass[1] as active material; it is read out at one side by a photomultiplier coupled via a 4 cm long cylindrical light guide of the same diameter as the photomultiplier (see figure 2.9). The front and rear faces of the blocks measure about 10 X 10 $\mathrm{cm}^2$ and 11 X 11 $\mathrm{cm}^2$ respectively and the blocks length is 37 cm. A LAV station is made by arranging these blocks around the inside of a segment of vacuum tank: the blocks are aligned radially to form a ring. Multiple

[1]This material is about 75% lead oxide by weight and has a density $\rho$ = 5.5 g $\mathrm{cm}^3$ and a radiation length $X_0$ = 1.50 cm; its index of refraction is $n \approx$ 1.85 at $\lambda$ = 550 nm and $n \approx$ 1.91 at $\lambda$ = 400 nm.



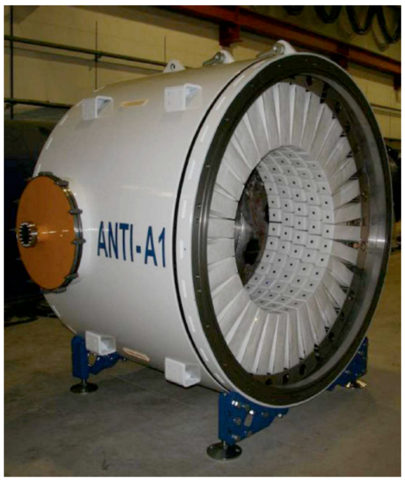


**Figure 2.9:** *A LAV block (left). The LAV1 station, with 32 X 5 lead glass calorimeter blocks (right).*

rings are used in each station to provide total minimum effective depth of 21 radiation lengths for incident particles. The blocks in successive rings are staggered and the rings are spaced longitudinally by about 1 cm.

The LAV provide time and energy measurements using the time-over-threshold (ToT) technique. The time resolution obtained for a single block is

$$\sigma_t = \frac{220\text{ps}}{\sqrt{E(\text{GeV})}} \oplus 140\text{ps}.$$

The front-end electronics is a custom discriminator board (called LAV front-end [3]) which converts the analog signals to low-voltage differential signals (LVDS). These signals sent to the common TDCB+TEL62 system which is used for readout and trigger purpose.

**LKr**

The LKr calorimeter (see figure 2.10) is mainly used as a photon veto in the forward angle region ($1.5\ \text{mrad} < \theta < 8.5\ \text{mrad}$). The LKr is also an important element for the L0 trigger reduction of $K^+ \to \pi^+\pi^0$ decays.

The experiment [**?**] needed an electromagnetic calorimeter with good energy, position and time resolution, precise charge calibration, long-term stability and a fast read-out to study direct CP violation. The experiment chose a liquid Krypton almost homogeneous ionisation chamber to meet these requirements [**?**].

When a photon or an electron enters the calorimeter active volume, it produces an electromagnetic shower via pair production and Bremsstrahlung processes until the energy of the particles falls below the critical energy. The charged particles of the shower can ionise Krypton atoms producing a number of electron-ion pairs proportional to the deposited energy. The produced electrons drift towards the anode where are collected before they can recombine. A liquefied noble gas was chosen to obtain a good resolution and energy linearity, with absence of ageing problems. Furthermore the relative

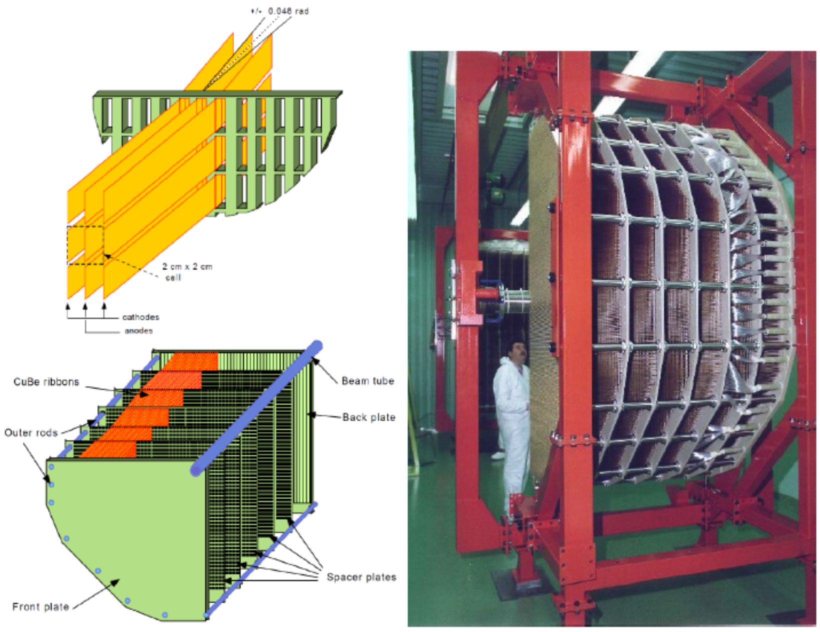


**Figure 2.10:** *The LKr electrode structure and a detail of the LKr cells showing the ribbons structure (left). A picture of the LKr (right).*

short radiation length of the liquid Krypton allows a compact design.

The low boiling temperature of Krypton (120 K) entails that the whole detector has to be kept inside a cryostat: only temperature variations of few per mille are allowed not to have big variations of the drift velocity.

The LKr is read-out in a current-sensitive mode: the initial induced current is proportional to the ionisation generated by the electromagnetic shower and so to the energy of the crossing particle. The signal is sampled and digitized every 25 ns by a flash ADC-based calorimeter readout module (CREAM) [5]. To reduce the number of read cells and the output bandwidth, a zero suppression is applied to the cells with pulse height below a certain threshold.

#### IRC and SAC

The two small-angle veto calorimeters, IRC (see figure **??**) and SAC (see figure **??**), are *"shashlik"* type calorimeters, i.e. detectors made of lead absorber layers with plastic scintillator plates used as active material.
The IRC is placed around the beam line in front of the LKr, and covers the angular region between LKr and the SAC. A dipole magnet bends the beam so that charged particles cannot hit the SAC, the most forward detector in the NA62 setup.

The IRC and SAC read-out uses a CREAM module.

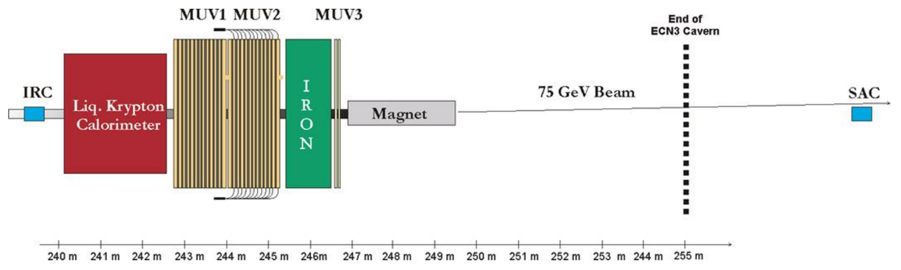


**Figure 2.11:** *IRC and SAC scheme*

### 2.3.2 STRAW

The purpose of the STRAW magnetic spectrometer is to measure the directions and momenta of kaon decay products. The kinematical constraint needed to reject most of the background requires an accurate reconstruction of the secondary particles tracks. The full spectrometer consists of four straw chambers. A dipole magnet, placed between the second and the third chamber, generates a vertical field of 0.36 T, corresponding to a kick of 270 MeV/$c$ along the $x$-axis. Each chamber is composed of four "views" ($x, y, u$ and $v$). Each view is made of 256 straw tubes. Fig. **??** shows the four views of a STRAW chamber.

The read-out electronics was designed in order to cope with an overall particle rate of about 15 MHz and a single straw maximum rate of 700 kHz. The front-end electronics called COVER is placed directly on the detector, sealing the gas volume [13]. A COVER can read-out 16 channels using 2 8-channel CARIOCA chips developed for LHCb [42]. The chip amplifies, shapes and discriminates the current signal induced on the wire chamber electrodes. The discriminator output is sent to the LVDS driver that provides the CARIOCA output signal. The time measurement is performed on an FPGA located directly on the COVER board. Leading and trailing are treated in parallel and the time to digital converter time resolution is of 0.78 ns. After the digitization of the signals and the time measurements, the data are sent to the Straw Read-out Board (SRB). Each SRB can read-out 16 COVERs (256 channels), it selects and packages the data to send it to the PC Farm.

### 2.3.3 RICH

A Ring-Imaging CHerenkov counter (RICH) is the main particle identification detector of NA62; it is employed to obtain $\pi - \mu$ separation and to achieve a precise time measurement of the pion candidate. Because of its good time resolution, it is one of the reference detectors in order to tag the passage of a charged particle and partially reject the multi-track events.

The RICH consists of a 17 m long vessel (see figure 2.13), with 3.8 m diameter, filled with Neon gas at atmospheric pressure (5.6% of a radiation length) and crossed by the beam pipe that allows the beam to pass through.

The internal optics is made of a mosaic of 20 hexagonal spherical mirrors (see figure 2.14a) that reflects and convoys the Čerenkov light towards the upstream part of the detector where the photomultipliers are placed. The mirrors are divided into two

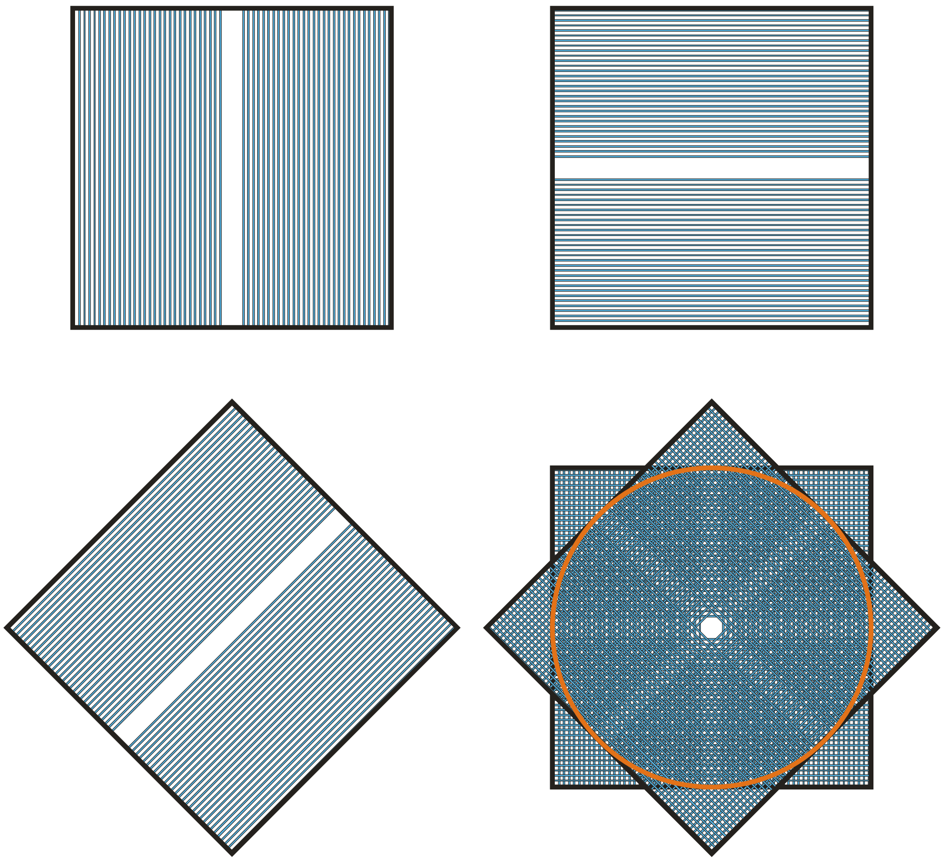


**Figure 2.12:** *The four view of a STRAW chamber [32]. In the bottom right corner the four view are super-imposed.*

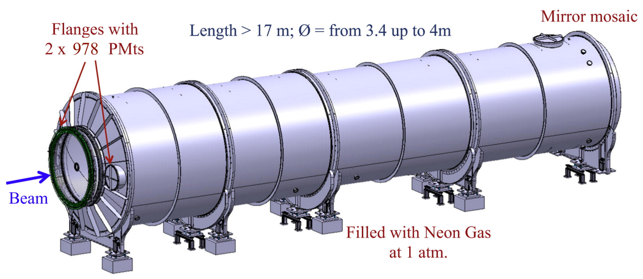
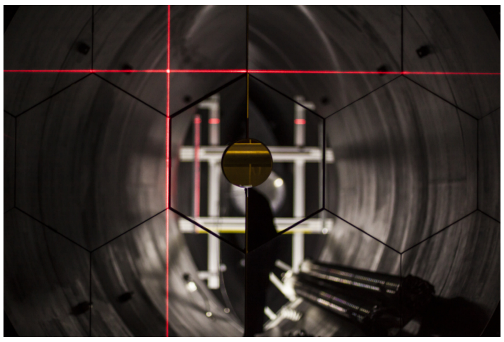


**Figure 2.13:** *A picture of the RICH installed in the NA62 cavern.*

spherical surfaces with centre of curvature respectively on the left and on the right of the beam pipe, in this way the absorption of reflected light by the beam pipe is avoided.

In the the upstream part of the detector, two flanges (see figure 2.14b) host 960 photomultipliers each to collect the Čerenkov light. Each flange has a diameter of about 0.7 m and its centre is at a transverse distance of 1.2 m from the beam pipe axis. The active area of each photomultiplier has a diameter of 8 mm and a Winston cone is used to collect the light from a pixel of 18 mm diameter.

A charged particle, traversing a medium of refraction index $n$, with a velocity $\beta c$


(a) .

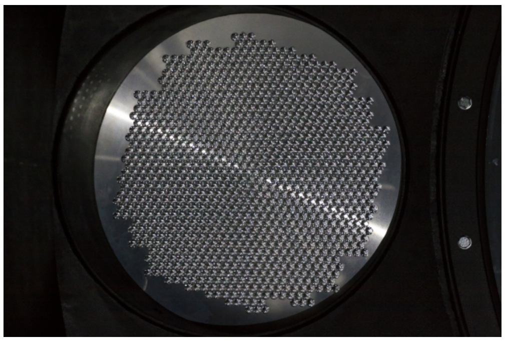

(b) .

**Figure 2.14:** *The RICH mirrors (a), a laser was used to calibrate the mirrors alignment. One photomultiplier flange (b).*

higher than the speed of light in the medium, emits a e.m. radiation at an angle $\cos\theta_c = 1/(n\beta)$. The light cone is reflected by the mirrors toward the photomultipliers placed on the mirror focal plane where the cone image is a ring of radius $r = f\tan\theta_c \sim f\theta_c$ ($f$ is the focal length). In this way the ring radius depends only on the particle velocity: this means that, for a particle of momentum $p$, the radius $r$ depends only on its mass $m$

$$r \approx f\sqrt{\frac{2(n-1)}{n} - \frac{m^2}{np^2}}.$$

Using this information it is possible to obtain a pion-muon separation in the momentum range between 15 GeV/$c$ and 35 GeV/$c$. Figure 2.15 shows the Čerenkov ring radius as a function of momentum (measured by the STRAW spectrometer) obtained with data of 2015 run and without any selection on particle type: electrons, muons, charged pions and scattered charged kaons can be seen.

Cutting on the reconstructed mass, charged pions can be selected and muons can be rejected: with 2015 data a 86% pion efficiency and a 1.3% muon survival probability were measured. It must be noted that in 2015 the RICH mirrors alignment was not optimal and the need for better pion-muon separation was the main reason for detector maintenance carried out during the 2015-2016 winter shutdown.

The measured RICH time resolution was 65 ps and this leads to the choice of the RICH as one of the possible reference positive detectors for the L0 trigger (see section 3.3.1).

The front-end electronics uses the same NINO chip [22] as for the CEDAR to process the photomultiplier signals; the readout of the about 2000 channels is done through the TDCB+TEL62 common system.

### 2.3.4 CHOD

A plastic scintillator hodoscope provides a fast signal to trigger data acquisition on the passage of a charged particle. The CHOD inherited by the NA48 experiment, is com-

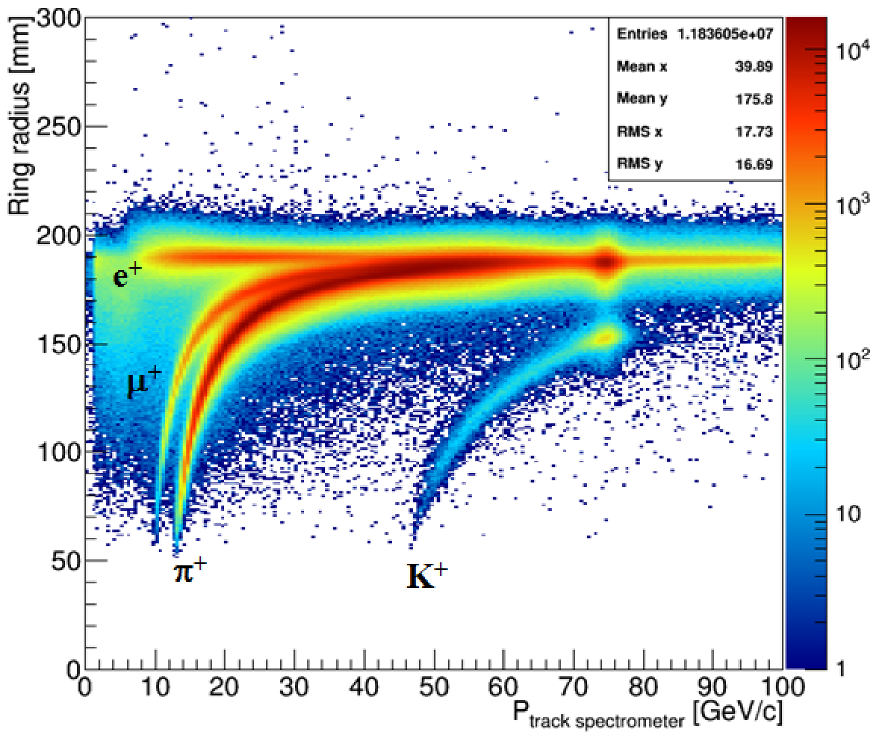


**Figure 2.15:** *Čerenkov ring radius as a function of particle momentum; electrons, muons and charged pions can be seen; charged kaons from the scattered beam can also be seen. Particles with momentum higher than 75 GeV/c are due to background muons from the experiment target. Data obtained in the 2015 run.*

posed of two planes of $64 + 64$ plastic scintillator bars aligned respectively to the $x$ and $y$ directions.

With this time resolution the CHOD is another possible reference positive detector for the main L0 trigger (see section 3.3.1).

The detector uses the LAV front-end electronics [3] and is read-out by the common TDCB+TEL62 system.

### 2.3.5 NEW CHOD

A new charged hodoscope was designed and was used for the first time in the 2016 run together with the old CHOD. The main reason to build the NEW CHOD is the high hit rate at which the long slabs (1 m) of the CHOD are exposed. The intrinsic dead time and the light transit time inside the scintillator are not compatible with the expected overall rate on the detector above 10 MHz.

The NEW CHOD is a two-dimensional array of 152 scintillator tiles (see figure 2.17) installed after the LAV12. In each quadrant, a 30 mm thick plastic scintillator is divided into 38 tiles. The scintillation light is collected and transmitted by 1 mm diameter wavelength shifting fibres to be detected by arrays of 3 X 3 $\text{mm}^2$ silicon photomultipliers (SiPMs) on mother-boards located on the periphery of the detector. A maximum rate of the order of 500 kHz is expected on the tiles close to the beam pipe.

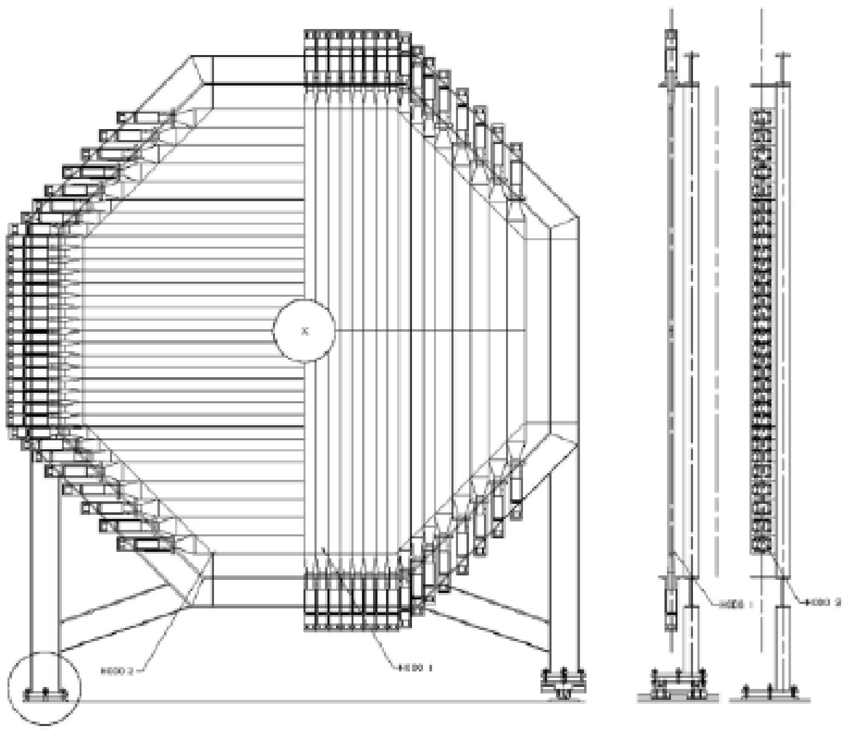


**Figure 2.16:** *Sketch of the CHOD (front and side). One can see the horizontal and vertical planes.*

The signals are shaped using constant fraction discriminators to improve the time resolution, and read out by the common TDCB+TEL62 system.

### 2.3.6 The muon veto system

A further reduction of the $K^+ \to \mu^+\nu_\mu$ background is achieved by the MUon Veto system, composed of three detectors (MUV1, MUV2, and MUV3). The MUV1 and MUV2 are downstream of the LKr calorimeter and work as hadronic calorimeters measuring the deposited energies and the shower shapes of incident particles, the MUV3 is instead located behind a 80 cm thick iron wall and is employed as a fast muon veto in the lowest trigger level (L0) and for offline muon identification (see figure 2.18).

#### MUV1 and MUV2

Both are iron-scintillator sandwich calorimeters; they have 24 (MUV1) and 22 (MUV2) layers of plastic scintillator strips alternated with iron plates. The plates have dimension 2600 X 2600 X 25 $\mathrm{mm}^3$; a central hole of diameter 212 mm allows the beam pipe passage. The scintillator strips of both modules are alternately horizontal and vertical.

The MUV1 consists of 48 X 24 strips (1152 in total) of about 6 cm width. Light is read by two wavelength-shifting fibers per scintillator strip. The fibers of one longi-

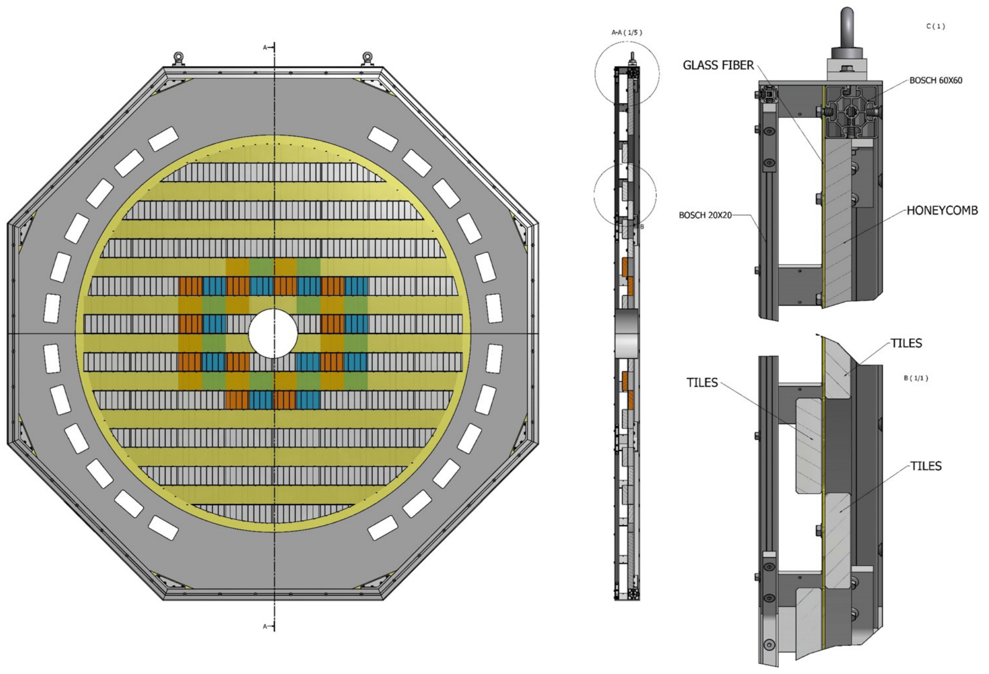


**Figure 2.17:** *Sketch of the NEW CHOD, front and side views.*

tudinal row of scintillators are bundled together to direct the light to one single photomultiplier, therefore no longitudinal segmentation exists.

The MUV2 is composed of 44 strips of about 11 cm width, each one spanning half of the transverse size of the detector. Consecutive strips with identical transverse positions are coupled to the same photomultiplier using plexiglass light-guides.

These two detectors are read-out with the CREAM modules [5] used by the LKr calorimeter.

### MUV3

The MUV3 detector (see figure 2.19) consists of an array of 12 x 12 plastic scintillator tiles, 5 cm thick, with a transverse area of 22 x 22 $\mathrm{cm}^2$. Eight smaller tiles are mounted around the beam pipe to cover the region with a higher rate.

The light produced by traversing charged particles is collected by two photomultipliers positioned about 20 cm downstream. The maximum time jitter between photons produced is below 250 ps, in this way the required time resolution is preserved. The only error in the time measurement could be due to particles traversing the photomultiplier windows: these particles produce Čerenkov photons who arrive earlier than those produced in the scintillators, with typical time differences of about 2 ns. To overcome this problem each scintillator tile is read out by two photomultipliers. The output time of the two photomultiplier signals coincidence, corresponds to the time defined by the photomultiplier which is unaffected by the Cerenkov photons. The time resolution of the MUV3 is below 500 ps, sufficient to keep the random veto probability at a low level.

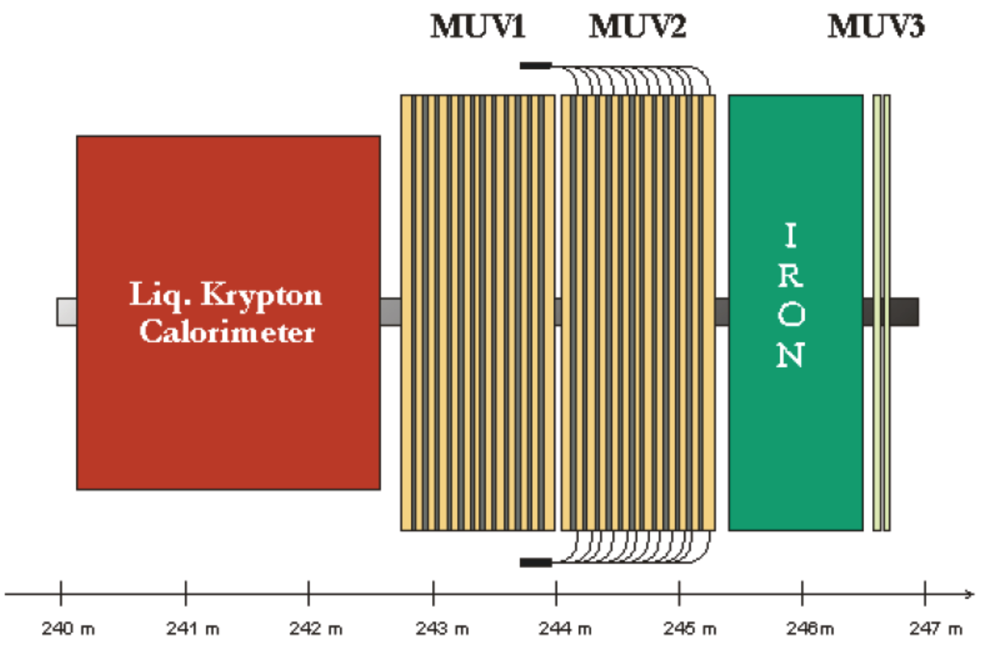
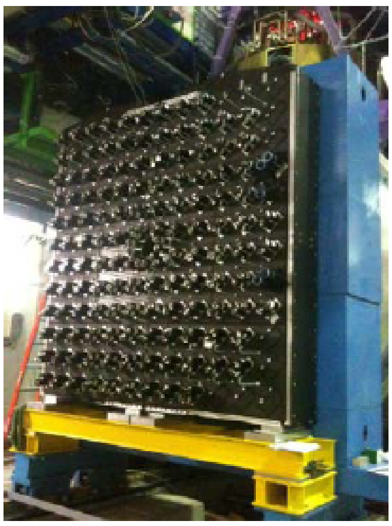
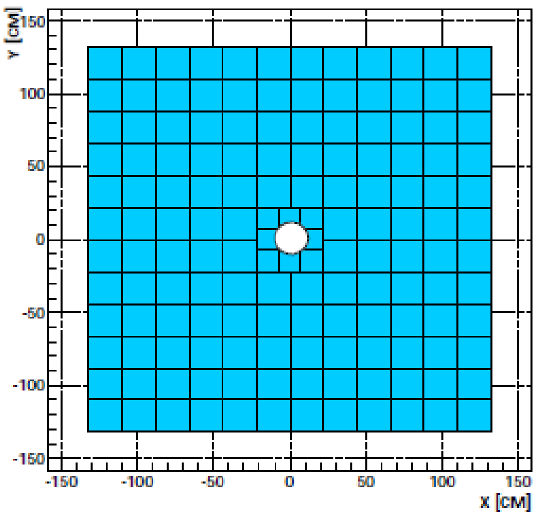


**Figure 2.18:** *Sketch of the Muon Veto System.*

The photomultiplier output signals are sent to constant fraction discriminators (CFD) which are then read-out through the common TDC-based system TDCB+TEL62.


(a)



(b)

**Figure 2.19:** *MUV3 detector picture (a) and layout (b).*

CHAPTER 3

# TDAQ system

## 3.1 Introduction

The high event rate (about 13 MHz) and the number of detectors in NA62 result in a large amount of output data ($\sim$ 30 GB/s) that is difficult to store without filtering. The total number of channels in NA62 is above 90000: considering the detector rate and size of each subdetector packet, the system produces a raw data bandwidth of the order of 2 TB/s. The number of channels and the typical hit rates of the detector are presented in table 3.1. A high performance trigger and data acquisition (TDAQ) system is therefore necessary, which must minimize dead time and maximize data collection rate. To do that, NA62 developed an unified trigger and data acquisition system: trigger is integrated inside the DAQ system, allowing to have a good control of the trigger, that use the same data available at readout, and a excellent flexibility. The trigger system is organised in three levels: the lowest-level trigger (L0) is hardware and is followed by two software high-level triggers (L1 and L2) implemented in a PC farm.

The common clock of the experiment is provided by the Timing, Trigger and Control (TTC) system used in LHC experiments [15], its frequency is close to 40 MHz and is the common unity reference for all time measurements, which are defined by a 32-bit timestamp, with 25 ns LSB (Least Significant Bit), plus 8 bit of fine time with 100 ps LSB covering the duration of an entire SPS spill (which is of the order of 10 s).

## 3.2 The TTC system

The Timing, Trigger and Control (TTC) system [15], developed for the LHC experiments, provides the timing of the experiment with a common, centrally generated, free-running synchronous 40.079 MHz clock. This clock is the unique reference for time measurements of all the experiment. The above frequency is the exact bunch-

| Sub-detector | Total channels | Hit rate (MHz) |
|---|---|---|
| CEDAR | 240 | 50 |
| GTK | 54000 | 2700 |
| LAV | 4992 | 11 |
| CHANTI | 276 | 2 |
| STRAW | 7168 | 240 |
| RICH | 1912 | 11 |
| CHOD | 128 | 35 |
| NEWCHOD | 304 | 45 |
| IRC | 20 | 4.2 |
| LKr | 13248 | 40 |
| MUV | 432 | 30 |
| SAC | 4 | 2.3 |

**Table 3.1:** *Number of channels and typical hit rates of NA62 sub-detectors*

crossing frequency of the LHC but in NA62 the kaon beam is unbunched, and this is just the reference clock frequency of a free running clock. The clock produced by the TTC system is distributed all over the experiment through optical fibers to detectors DAQ and the L0 Trigger Processor (L0TP). The NA62 TTC system (shown in figure 3.1) is composed of a central clock source and many sets of LTU + TTCex modules.

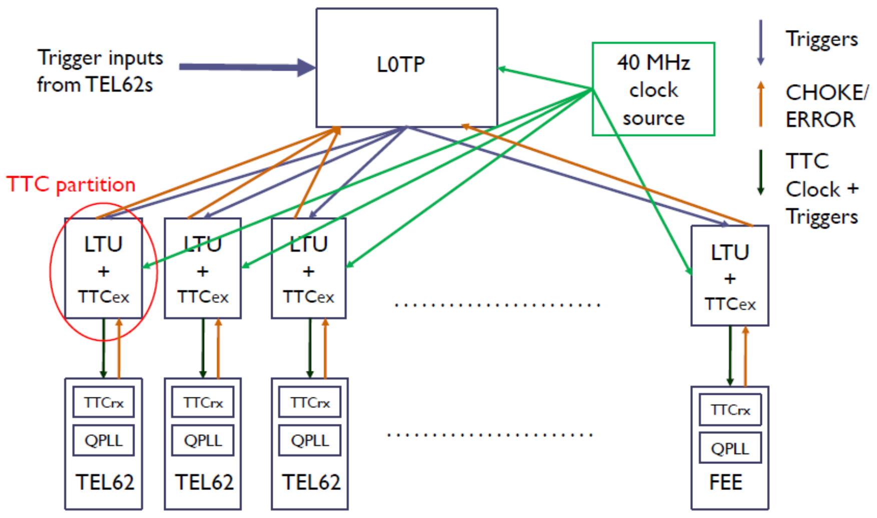


**Figure 3.1:** *The NA62 TTC system.*

These modules encode and send clock and triggers to the readout electronics of the detectors.

A back pressure system of CHOKE and ERROR signals is implemented : they are produced in case of exceedingly high rate or errors by the data acquisition boards and the Local Trigger Unit (LTU) propagates them to the L0TP to momentarily stop L0 triggers.

The LTU [36], a 6U VME module, is an updated version of the local trigger unit designed for the ALICE experiment. An on-board FPGA on the LTU receives the clock from the TTCex and is programmed to perform the following tasks: dispatch triggers received from the L0TP (L0 Trigger Processor, see the next section) to the TTCex for optical encoding, in the form of a trigger signal and a trigger message containing 8 bits; propagate the back pressure CHOKE and ERROR signals to the L0TP; deliver synchronously to the readout systems the Start Of Burst (SOB) and End Of Burst (EOB) signals, received from the SPS, as special trigger messages: these signals respectively start and end the data acquisition operations inside a single burst synchronously to the entire system. In the trigger messages the lowest two LSB bits are reserved to encode SOB and EOB signals, the remaining 6 bits represent the Level 0 trigger type (up to 63 different types).

The TTCex module [46] is a 6U board: it receives the main clock which drives an internal QPLL, and is linked to the LTU to receive trigger signals and trigger information messages. The module provides several optical fibre outputs in which clock, trigger signals and trigger messages are encoded together.

All the subdetectors front-end subsystems are endowed with a TTC interface containing an optoelectronic receiver and a TTCrx chip [21] which decodes the L0 trigger information.

The trigger word can encode several physics triggers belonging to different conditions and subdetectors, as well as some service triggers, e.g. signal useful to synchronize all the detector together, to monitor the the noise or the background in the detectors and the CHOKE/ERROR signals which represent a warning or a problem in the general data acquisition and they says to the read-out board to pause or stop the data acquisition.

## 3.3 The trigger system

The general Trigger and Data AQuisition (TDAQ) structure of a modern high energy experiment is organized on more levels: the detectors data are stored if they satisfy some requirements established by several sequential trigger levels. The NA62 trigger system is structured in 3 levels that have to reduce the event rate from about 13 MHz to some kHz. The number of detectors channels and their high rates led the NA62 collaboration to choose for the first level (called L0) a fully hardware trigger; after the L0 trigger, the sub-detectors transfer the data to a pc farm where the L1 and L2 trigger are implemented. The L1 is based on the information computed by each complete subsystem. The L2 uses assembled and partially reconstructed events with the possibility to use correlations between different sub-detectors. The NA62 trigger hierarchy is shown schematically in figure 3.2.

### 3.3.1 L0 trigger

The L0 trigger is digitally implemented on hardware in the common TEL62 board, it's based mainly on input from CHOD, MUV, RICH, LKr and LAV12, and with the goal of reducing the event rate from about 13 MHz to 1 MHz. The default L0 trigger algorithm consists in requiring a single charged track in the CHOD and RICH, nothing in the MUV3 and in the LAV12, and an amount of energy in the LKr and in the MUV1 com-

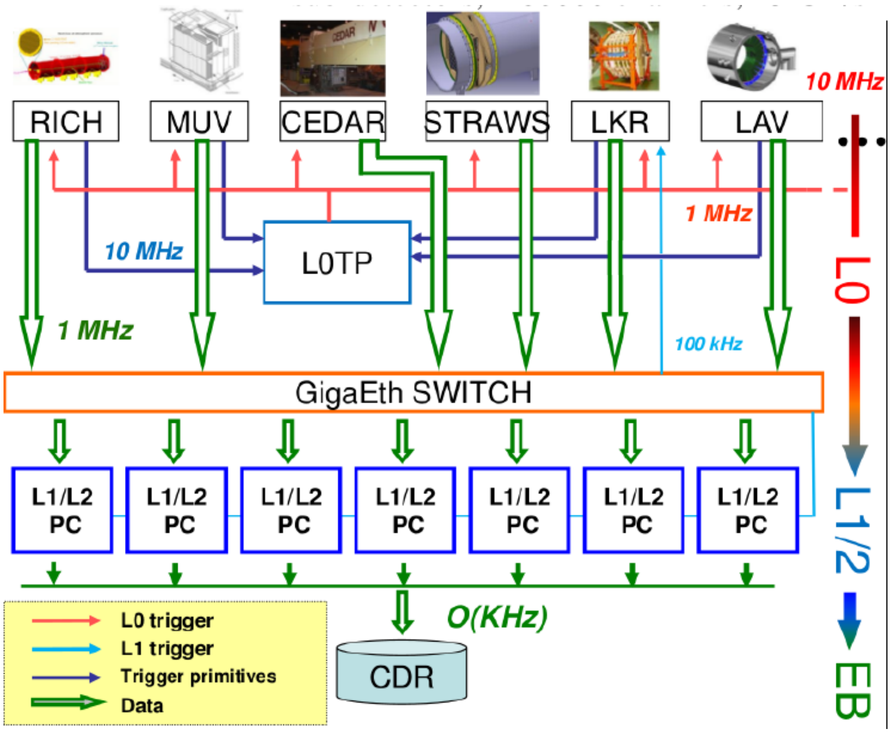


**Figure 3.2:** *Schematic view of NA62 TDAQ system*

patible with a charged pion. Together with this main trigger, other L0 trigger could be implemented to acquire a selection of events useful for different physical goals respect to the branching ration measurement of the $K^+ \to \pi^+\nu\overline{\nu}$ decay. The signals of the CHOD and RICH will allow to tag a charged particle within the detector acceptance, reducing the rate due to K decays downstream of the final collimator. It is possible to use, in addition, the hits multiplicity to select multi-track events. The fast third station of the muon veto (MUV3) is used to reject the high rate of muons due to the major background decay $K^+ \to \mu^+\nu_\mu$ (about 63% of Branching Ratio) and the muon halo components from decays upstream of the final collimator. The LKr and MUV1 are useful to suppress the rate due to the other background decay $K^+ \to \pi^+\pi^0$, by requiring the energy released corresponding to the charged pion electromagnetic shower. An online cluster counting with a time resolution of 1 ns can allow a good rejection and at the same time give useful information for several other sets of physics triggers. The RICH is able to contribute by exploiting hits multiplicity, for the reduction of the background due to multi-track events, and giving a very precise time measurement O(300 ps); we are not able to use the particle identification in the L0 trigger because it requires the correlation with the information from other sub-detectors like the measurement of the particles momentum from the slow magnetic spectrometer. The RICH trigger condition is computed in a single TEL62 board, called *RICH MULTI*, that collect the signal of all the detector super-cells (a super-cell is the digital OR of 8 RICH PMs). The last station of the LAV veto (LAV12) is used to reject at the L0 a part of the events with photons in the final state. The other stations are used inside the L1 trigger condition. It is not possible at the moment to use all the LAV station in the L0, but is under development a system that will allow the firmware communication between the LAV TEL62 board to generate a common trigger condition. The new-CHOD detector is been installed for the 2016 RUN; it could be used, simultaneously, with the old CHOD detector to select single charged tracks. The new-CHOD trigger and read-out efficiency will be tested in the first period of the run.

The electronic boards connected to these sub-detectors inside the trigger system produce L0 trigger primitives with timestamp and fine time to allow the time matching. The L0 trigger primitives are managed by a central L0 Trigger Processor (L0TP). The L0TP used during the 2015 run is a FPGA system based on the Altera®development DE4 board [7] which accommodates a Stratix®IV FPGA [11], plus a TTC [21] interface card and daughter-cards for 8 Gigabit ethernet ports used for trigger primitives reception. The received primitives are stored in RAMs implemented in the FPGA, with a RAM address depending on the primitive time. The timestamps of a reference (positive element) detector are stored in a FIFO, and subsequently read to search for primitive matching in times from other detectors; different trigger masks coexist for different physics goals, and a look up table is implemented inside the FPGA for this multiple matching purpose. After the matching, if the trigger conditions are satisfied the L0TP generates a L0 trigger signal followed by a trigger type word that is dispatched by the TTC system back to all detectors. When the detector boards receive this signal they read-out the data inside a programmable number of time slots (each one of 25 ns) around the trigger time. The data, waiting for the L0 trigger signal, is stored into some memory buffer (on the TEL62 boards is a DDR2 memory).

The maximum L0 trigger latency was set to 1 ms and it is limited by the memory inside the L0TP indeed the read-out electronics of the detectors can currently store data for more time like the DDR2 memory in the TEL62 ($\sim$ 50 ms) and the memory in the readout board used by the LKr ($\sim$ 10 ms).

### 3.3.2 L1 and L2 triggers

The sub-detector data, after the L0 trigger signal, are extracted from memories and sent to the PC farm where the L1 and L2 triggers are implemented. The L1 algorithms check data quality conditions and then requires simple correlations between conditions computed by single sub-detectors. The L1 trigger uses the KTAG to cut the non-kaon component of the beam, the LAV stations to reject events with photon and the CHOD to reduce multi-tracks events. A significant part of the rejection power is obtained using STRAW informations that help to cut events with a decay vertex out of the fiducial region, due to beam particles that are inside the detector acceptance and multi-body decays. The L1 trigger is necessary to achieve an overall trigger rate below 100 kHz. In case of positive L1 decision, a complete event reconstruction at L2 will be done.

The L2 takes care of accepting, for the main trigger, events with only a single identified charged pion. These two software level triggers are both implemented inside the same PCs, so useless data transfers are avoided. All the events accepted by the L2 trigger are finally stored on tape. The available bandwidth for data storage is about 100 MB/s, this limits the final output trigger rate to about 10 KHz.

The NA62 Pisa group is developing a parallel fast online trigger system using commercial graphic processors (GPUs) [17]. This system could support some L1 and L2 trigger algorithms acting as a Level 0.5 trigger between the hardware level and the software levels. Two low latency applications under evaluation in NA62 are the RICH online ring-finding, both for L0.5 and L1, and a L1 tracks reconstruction for the STRAW spectrometer.

## 3.4 Data acquisition system

The high hit rate and the consequent need of a good time resolution to have an efficient selection of the events, led the collaboration to approve the implementation of a TDC-based system for many detectors. A TDC-based system can also provide pulse-height information using a time-over-threshold approach: both leading and trailing edges of pulses should be measured to utilize this method. Consequently most of the detectors of the experiment (CEDAR, CHANTI, LAV, RICH, CHOD, NEWCHOD and MUV3) adopted a TDC and FPGA-based common readout composed of the TEL62 carrier board [24] [14] and the TDCB daughter-card [20]. A part of my PhD work consisted of the firmware development of these two boards; this will be described in sections 3.5.2 and 3.6.1.

Due to specific needs, some detectors chose to adopt a different custom TDC system or an ADC based readout instead and these read-out systems are briefly described in the rest of this section. The readout of the Gigatracker is based on a TDCPix chip that provides, in addition to the hit time, the time-over-threshold self-triggered measurement for 360 channels, the pre-amplification and the discrimination of the signal [37]. The TDCPix time bin size is about 100 ps, the expected rate is about 210 MHits/s and the output is given by four 3.2 Gbit/s serial links. Taking into consideration the propagation of the signal through the chip and the relative time correction the chip time resolution is about 70 ps. The four output serial links send data to a carrier GTK-RO VME 6U board where it is stored waiting for a L0 trigger decision. After the reception of a L0 trigger signal the data inside a 75 ns time windows are sent to the readout PC. There is also on the board a TTC interface to receive triggers from the central L0 Trigger Processor and to send the clock to the TDCpix.

The STRAW spectrometer electronics [13] is based on a 8-channel analogue front-end chip, the CARIOCA chip [42],containing a fast pre-amplifier, semi-Gaussian shaper, a tail cancellation circuitry, base-line restorer and a discriminator. Two CARIOCA chips are integrated in a custom COVER board [44] together with an Altera®Cyclone®III FPGA [6]. The COVER board houses 16 pairs of TDC implemented within the FPGA with de-randomizers and an output link serializer. The full system is composed of 14236 TDCs producing a data rate of order 2 GB/s. The back-end VME 9U Straw Readout Board (SRB) receives data from 16 COVER boards for processing (in FPGAs) and storage in DDR3 memories. A TTC interface is also present, to optically receive the clock and the L0 trigger signal. After receiving a L0 trigger, data are extracted from the buffers and sent to the PC farm.

The LKr, the MUV 1 and MUV2 use a Flash Analog to Digital Converter (FADC) based readout. The readout board is the Calorimeter Readout Module (CREAM) [5] developed by CAEN (see figure 3.3). The CREAM is a VME 6U board able to digitise 32 LKr channels at 40 MHz using four 8-channel, 14-bit ADCs. Four hundred and fourteen such boards are needed to read out all the calorimeter cells (13248). After signal digitalization, samples are stored in a circular buffer built inside a DDR3 module, waiting for the L0 trigger signal. Upon reception of such signal through a custom backplane, data is moved to the L0 buffer, also built in the same DDR3 module, where it waits for a L1 trigger signal. This is one main differences with respect to other detectors: due to the high data rate, the LKr is only read out at the reduced L1 trigger rate (below 100 kHz). When such signal is received, the corresponding data is finally sent to the PC farm. The CREAM module also computes digital sums of $4 \times 4$ channels, called Super-cells, for L0 trigger purposes: two Super-cells are read out by each CREAM, and digital sums are sent every 25 ns to a system based on 36 TEL62s and custom interface mezzanine boards, where LKr L0 trigger primitives, based on energy deposits and cluster identification, are generated.

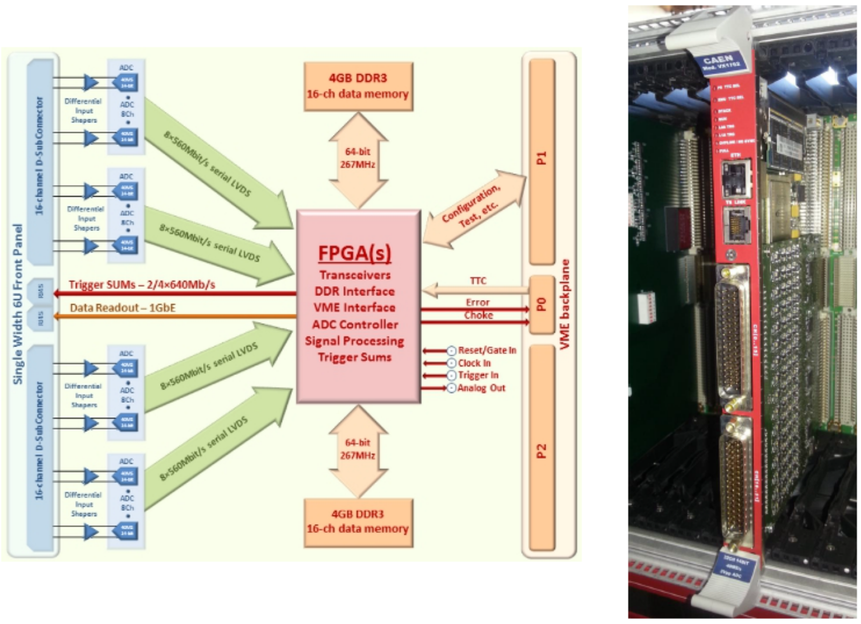


**Figure 3.3:** *Logical scheme and photo of a Calorimeter Readout Module.*

At present, the IRC and SAC small angle photon veto detectors use two different readout system together: the TDCB-TEL62 and the CREAM board systems. Both IRC and SAC have only 4 channels, this number is linked to the transversal dimension of an electromagnetic shower indeed a greater segmentation of these detectors would not have diminished the single channel rate.

## 3.5 The TDC Board

The TDC Board (TDCB) is a high-density (10 layers printed circuit) mezzanine daughter-card for the TEL62 carrier motherboard, designed in Pisa for precision time measurements. The board design (see figure 3.4) was driven by the desire to integrate a high number of channels within the same processing board, in order to ease triggering issues [20].

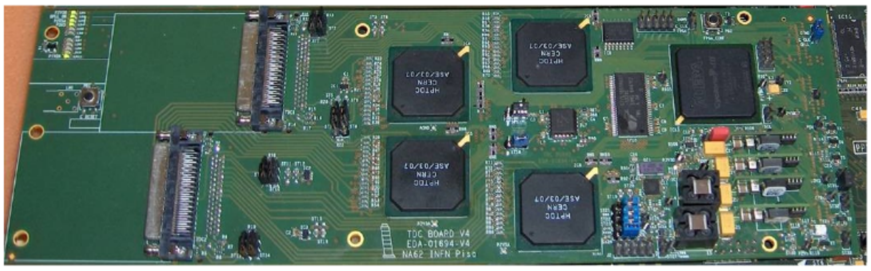


**Figure 3.4:** *The TDC Board.*

The desire for a compact and common electronics and the short distance (order of meters) between subdetectors and readout electronics, with no space constraints, led to the choice of having digitizers on the readout board rather than on the detector thus leaving only analog front-end electronics on each individual subdetector in a potentially higher radiation environment and making all digital electronics common and located on the same boards, at the price of having to transmit analog pulses on the 5 m LVDS cables between the two.

The requirements of a good time resolution and high channel integration led to the choice of the CERN High Performance Time to Digital Converter (HPTDC) [16] as time digitizers (see figure 3.5).

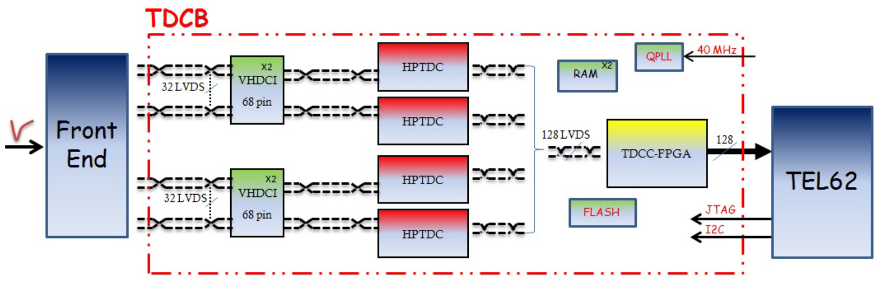


**Figure 3.5:** *Schematic view of the TDCB architecture.*

With a fast front-end electronics providing adequately time-stretched LVDS discriminated pulses, the measurement of both the leading and trailing edge times allows obtaining analog pulse-height information by the time-over-threshold method: HPTDCs can indeed digitize the time of occurrence of both signal edges, provided they are separated by a minimum time (about 7 ns); the resulting time measurements (made with respect to clock edges) can be combined into a single word to reduce the required data bandwidth.

The board houses four 68-pin VHDCI connectors for input signals, each of them delivering 32 LVDS signals to one TDC, with two spare pairs being used to provide additional grounding (one pair) and to allow user-defined back communication from the TDCB to the front-end electronics (one pair). This latter feature can be used to trigger the injection of calibration pulses in the subdetector or calibration patterns in the front-end (as two single-ended lines allowing bidirectional communication, or as a LVDS pair towards the front-end). This choice allows in principle the use of high-performance cables, if required by the intrinsic resolution of a subdetector, as well as cheaper solutions.

The TDCB houses a dedicated Altera®Cyclone®III EP3C120 FPGA [6], named TDC Controller (TDCC-FPGA) which can handle the configuration of the four HPTDCs via JTAG, read the data they collect, and possibly pre-process it. A 2 MB external static RAM block is also available and will be used for online data monitoring purposes and low-level checks on data quality.

The TDCC-FPGA can be configured from an on-board flash memory (Altera®EPCS64 [9]), which can be loaded using an external programmer via an on-board connector or through JTAG, either using a JTAG port on the TEL62 board, or through its Credit Card PC (CCPC). Communication between each TEL62 FPGA and the corresponding TDCC-FPGA on the daughter-board proceeds through a 200-pin connector with 4 independent 32-bit single-ended LVTTL parallel data buses (one for each TDC, running at 40 MHz, for a total bandwidth of about 5 Gbit/s) and a few dedicated lines for synchronous commands and resets. The TDCC-FPGA on the TDC daughter-card can also be accessed from the TEL62 CCPC card via a dedicated I2C connection for slow operations.

The individual TDCs are configured via JTAG, with the TDCC-FPGA acting as the JTAG master: the configuration bits are sent to the TDCC-FPGA from the TEL62 CCPC card via I2C, and are then uploaded to the TDCs. A second working mode allows inserting both the TDCC-FPGA and the four HPTDCs into a global JTAG chain which also includes all the TEL62's devices, and which can be driven by the TEL62 CCPC card.

The contribution of the digitizing system to the time resolution ultimately depends on the random jitter of the reference clock against which the measurement is performed. The 40 MHz clock optically distributed by the TTC is received through the TEL62 and is cleaned by the on-board QPLL [43] to reduce the jitter below 50 ps. This clock signal drives the internal logic and is distributed to each TDCB, where it can be configured to go through more jitter-cleaning stages: these are the internal PLL of the TDCC-FPGA and a second on-board QPLL.

### 3.5.1 HPTDC

The HPTDC [16] can work in an un-triggered mode, in which all available data is delivered at every readout request, or in a trigger-matching mode, in which a readout request follows a trigger pulse, and the TDC only delivers the data which matches in time the trigger occurrence, within some programmable time windows. Trigger-matching mode was implemented to allow HPTDCs to work as front-end buffers, storing data in a buffer (called L1 buffer, but have no relation to the NA62 L1 trigger) while a trigger signal was generated (see figure 3.6); however, in a modern experiment such as NA62, the latency of the lowest trigger level (1 ms) is much longer than the typical time it takes to fill TDCs' buffers (order of tens of $\mu$s in our experiment).

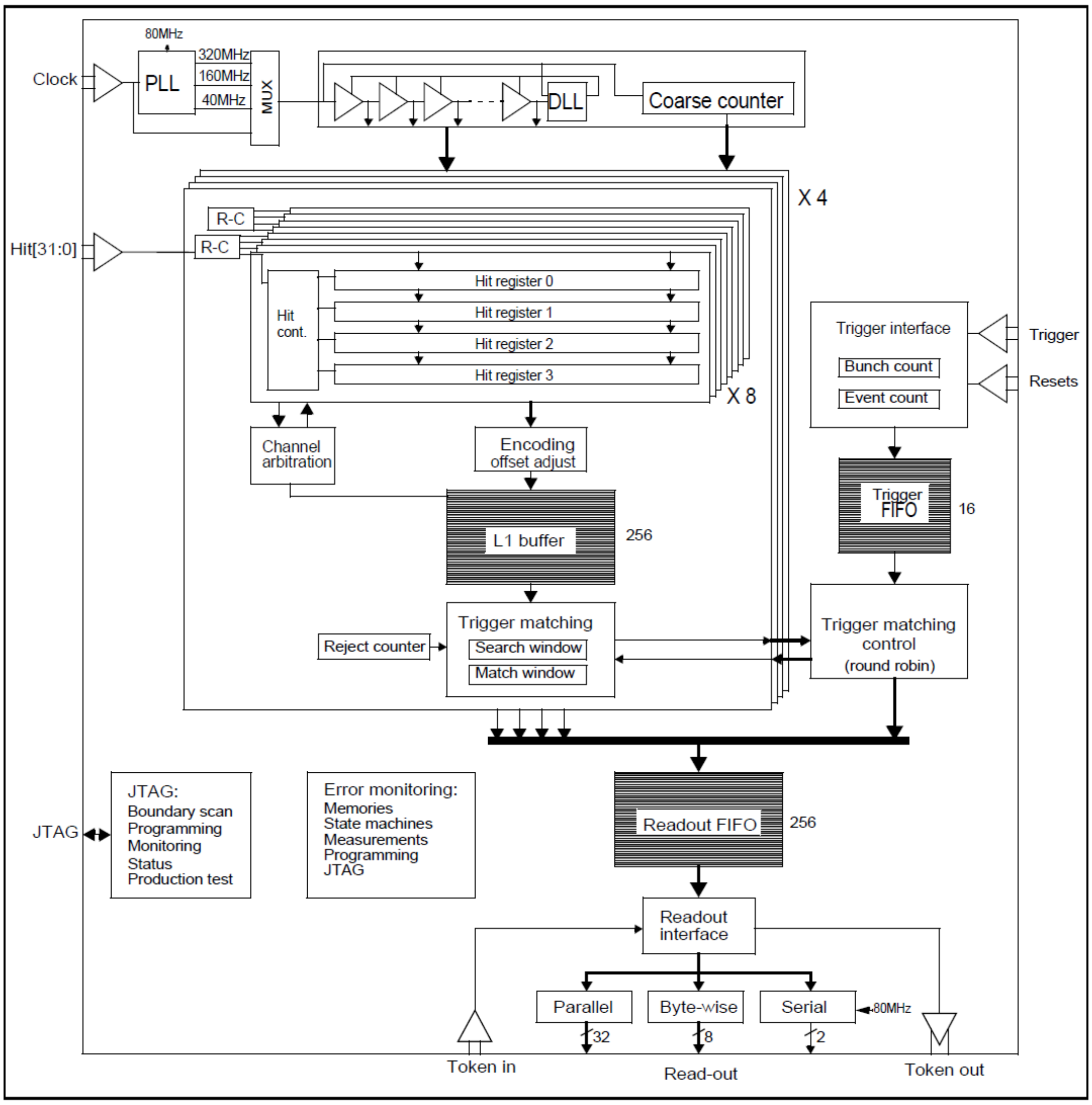


**Figure 3.6:** *Schematic view of the HPTDC architecture. The input for the TDC from the subdetector front-end electronics are at the top left and are called Hit[31:0]. At the bottom there are the link (called in the scheme Read-out) with the TDCC-FPGA.*

The TDCBs therefore normally use TDCs in trigger-matching mode just as a way of obtaining properly time-framed data, but triggers are actually sent to HPTDCs in a continuous periodic stream, with no relation whatsoever to the trigger of the experiment (data storage during trigger latency being actually provided on the carrier TEL62 board with much larger buffers). The time-matching parameters in the TDCs have to be properly set in order to allow readout of all hits which occurred since the previous trigger (time-matching window set equal to trigger period): in this way the TDCs are periodically triggered and readout, receiving all hits corresponding to a time frame of length equal to the triggering period, in a continuous sequence. Since the range of time measurement within the TDC chips is limited (to 51.2 $\mu$s at most), and therefore a roll-over of the TDC time word frequently occurs, only by exploiting this working mode one can be guaranteed (by the TDCs themselves) that all the data are being read.

Each HPTDC provides 32 TDC channels when operated in fully digital mode at 98 ps LSB resolution, with some internal buffering for multi-hit capability and a trigger-matching logic allowing the extraction of hits in selected time windows. Four such chips, for a total of 128 channels, are mounted on each TDCB, resulting in a grand total of 512 TDC channels per fully-equipped TEL62 carrier-board; as an example, the entire RICH detector can be handled by 4 TEL62 boards only, while most small subdetectors only require a single TEL62 board.

### 3.5.2 The TDCC-FPGA firmware

We used the software tools HDL Designer® [27], Modelsim® [28] (from Mentor Graphics®) and Quartus®II [8] (from Altera®) to develop the firmware for the TDC Board's TDCC-FPGA (and for the FPGAs of the TEL62 carrier-board). The main TDCC-FPGA firmware [31] parts and functionalities are shown in a block diagram in figure 3.7.

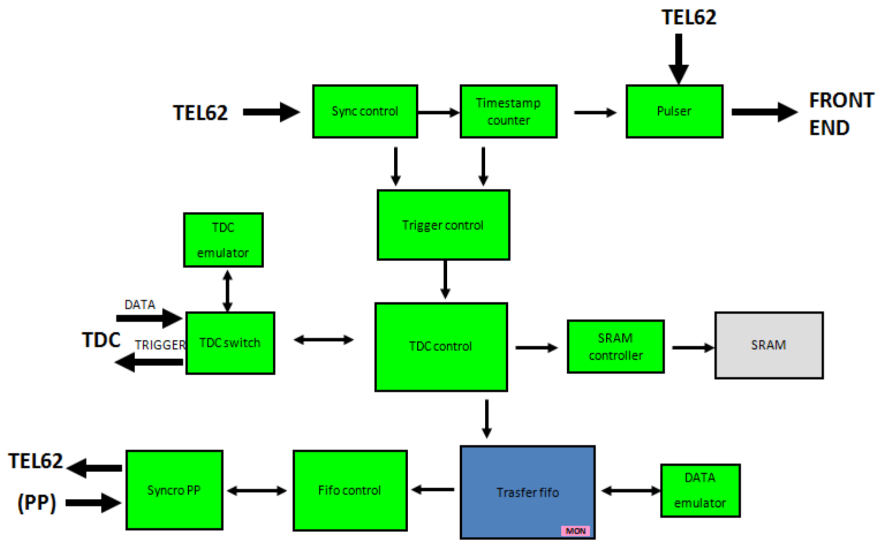


**Figure 3.7:** *Block diagram of the TDCC-FPGA firmware.*

To have a good resolution for the time measurement we have chosen a TDC configuration, in which the chip produces two 32 bit-long words for each signal, one word for the leading and one for the trailing edge. The number of bits dedicated to time measurement are 19 out of 32, the other being used identify the channel (5 bits), the TDC inside the board (4 bits) and the type of data word (4 bits). The possible types of words are 5: leading or trailing edge, error words and TDC frame timestamps and word counters. Inside the FPGA the data are are packed in 6.4 $\mu$s frame, each frame have a header word (TDC frame timestamps) and a trailer word (word counters).

The TDCC-FPGA firmware performs several operations:

- **Packing the data stream from the TDCs.** The TDCC-FPGA receives the data sent by the HPTDCs through a 32-bit parallel block writing protocol. The TDC time measurement rolls over every $51.2\ \mu$s, while the SPS spill is several seconds long. To cope with these different time scales, in the *TDC control* block we add a timestamp to the data in such a way that the TDC time measurement is unambiguously associated to an "absolute" time for the whole length of the spill. This is achieved by periodically triggering the TDCs with a period shorter than the TDC's roll-over (we chose a period of 6.4 $\mu$s) and adding a frame timestamp at the beginning of the data stream associated with each frame. Then at the end of the frame data stream we add a word counter indicating the number of words received by the TDC plus the frame timestamp itself. The roll-over of the frame timestamp (whose least significant bits corresponds to 400 ns) is $2^{28} \times 400\,\mathrm{ns} \approx 107$ s, much longer than the SPS spill duration.

  In the *TDC control* block 2, of the 4 bits that identify the TDC inside the board, are replaced with 2 parity bits. These parity bit are useful to verify if the data were altered (e.g. corruption or bit-flips) during the data flow. A mathematical function returns a 2 bits result for each data word (usually a the XOR of the data word bits); these 2 bits can be checked off-line to test the integrity of the each data word.

- **Configuring TDCs and communicating with the CCPC on the motherboard.** As mentioned before, configuration data is sent to the TDCB from the CCPC on the TEL62 through I2C. An I2C slave controller has been implemented in the firmware, through which internal registers are both written and read. A JTAG master controller is implemented in the TDCC-FPGA to transmit and receive the configuration to and from the TDC chips.

- **HPTDC emulator.** Two different HPTDC emulators are contained in the TDCB firmware: one was developed to send some simple repeating pattern on selected TDC channels (called *single-channel TDC emulator*), the other allows to load some amount of data words from a file into memory and repeatedly send it (called *data emulator*).

  The *data emulator* generates data packets containing a variable number of words per frame as required by the user, sending one data packet for each frame to which the header (timestamp) and trailer (word count) are automatically added; This emulator is directly linked to the TDCC output FIFO (called *transfer FIFO*). The data words are read from four (one for each TDC) 32-bit wide and 1 K deep pattern memories and modified so that the upper bits of the TDC time field do match the current frame timestamp (so the data words will not repeat exactly). The *data emulator* read from the memory and puts in each frame a number of words as read from a 9-bit wide and 1 K deep count memory (0 value in a line of this memory will generate an empty frame with only the header and the trailer). It is even possible to set the *data emulator* to repeat the same sequence continuously. The *data emulator* is used to test parts of the firmware of the TDCB or of the TEL62, using patterns written ad hoc to stress the firmware and the electronics, such ad the TEL62 output links.

- **The on-board RAM.** The data stream can be optionally split and sent to the TDCB's on-board static RAM during the acquisition, to possibly store a fraction of it for reading it and analysing it off-line. This can be useful for debugging or monitoring purposes. The RAM space is equally subdivided among the four TDCs. In the final implementation, it will be possible to choose between filling the RAM with the data from the first part of the spill, the last one or only with the frames that exceed same defined rate.

- **Front-end pulser.** The TDCC-FPGA can drive a spare output LVDS pair of the TDC connector, that allows to trigger the front-end boards for sub-detectors' calibration. Some front-end boards can send data signals to the TDCs in response to that stimulusS: this is useful for debugging purposes and to test the hardware connections. The *Front-end pulser* is driven by a mask register which is compared with a timestamp counter. An alternative working mode generates the output signal when the TDCC receives a special command from the motherboard.

### 3.5.3 TDCB test

Once the TDCB design was finalized various tests were performed in order to verify that the system complies with the experiment's requirements. These tests were performed in the laboratory of Pisa INFN, using a test setup which may reproduce conditions similar to the experiment environment, and at CERN during a test beam [20].

#### Time resolution

An important figure is the intrinsic contribution of the board to the time resolution. It was evaluated in the laboratory by pulsing TDC channels with signals of fixed duration generated by an FPGA-based test board working on the same clock used by the TDCB, and by measuring the time differences between the trailing and leading times of a LVDS pulse with nominal 25 ns duration. In a first test we pulsed 32 TDC channels at a time: we observed that the RMS of the measured pulse width on a single channel was 61 ps (figure 3.8 left) and the average was 25.18 ns on a sample of $10^5$ pulses. In a second test we pulsed simultaneously only 1 every 4 TDC channels (8 in total) with the same signals of the previous case (figure 3.8 right) and found comparable results.

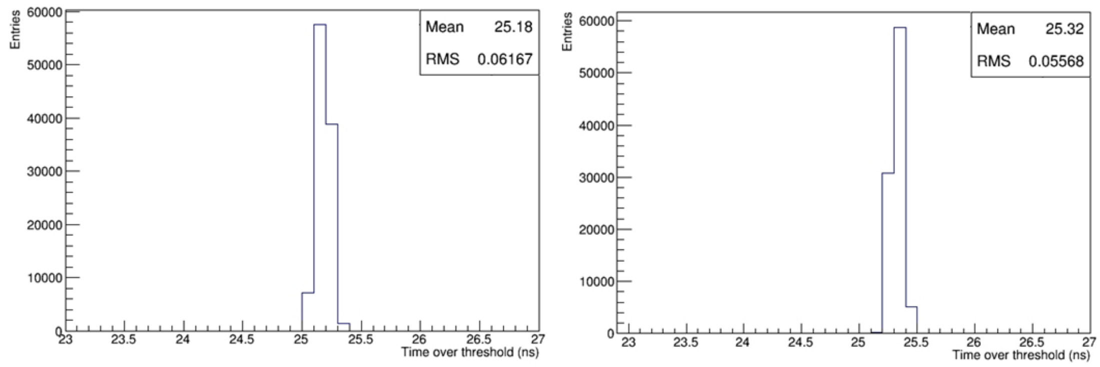


**Figure 3.8:** *Distribution of times-over-threshold for digital 25 ns wide signals as measured by the TDCB, with 32 channels pulsed (left) and 8 channels pulsed (right).*

No spurious hits were detected on channels which were not pulsed, over a sample of $10^7$ events. We checked other possible cross-talk effects by comparing the channel time resolution in the previous two tests: all the 32 TDC channels pulsed with signals of constant width (25 ns) or only 8 TDC channels (1 every 4) pulsed. We didn't observe differences in the time resolution (see figure3.8) when other channels were pulsed, concluding that no significant cross talk effects are present.

#### Channel efficiency

Hit losses are expected at high rates due to the limited amount of buffering present in the HPTDC. In the HPTDC the 32 channels are divided in four independent groups of 8 channels each, and hits are buffered at different stages: per channel, per group of 8 channels, per full chip (respectively Hit registers, L1 buffer and Readout FIFO in figure 3.5); apart from the first buffering stage, the other ones are monitored for overflow conditions.

The data transfer efficiency is defined as the ratio between the number of hits delivered by the system and the number of input pulses. It was measured in the laboratory both using only one single channel (1 of every group of 8 for a total of 4 TDC channels) and using two neighbouring channels (channels 0 and 1 of every group of 8 for a total of 8 TDC channels). We use these configurations because inside the HPTDC every group of 8 channels shared a 256 words deep level 1 buffer [16]. When several hits are waiting to be written inside the level 1 buffer an arbitration between pending requests is performed. Arbitration between channels in the active request queue is done with a simple hard-wired priority (channel 0 highest priority, channel 7 lowest priority). In this way the first channels in each group of 8 are serviced faster than the last ones. At high rates this gives an advantage to the low-numbered high-priority channels which can use their channel de-randomizers more efficiently, while in such conditions lower-priority channels appear like they have smaller de-randomizing capability and therefore slightly higher data losses. The results of the measurement and different channels efficiency are shown in figure 3.9.

No data losses were observed for input rates below 17.5 MHz in the first case (single channel). In the case of two adjacent channels being pulsed, no data losses were observed for input rates below 8.5 MHz per channel; above such value data losses start to appear at slightly different levels depending on the pulsed channel.

#### Test beam results

In November and December 2012 a test run with beam was performed at the CERN SPS. The main measurements performed during of this test run were the response of the detector and the front-end, the rates and efficiencies of the detectors and the TDAQ system, the time and space correlations between the detectors. The setup in the test beam was similar to the experiment, with the secondary kaon beam of 75 GeV/c produced by protons from the SPS. The test was carried out with a low intensity beam (1/50 of nominal rate foreseen for NA62). Figure 3.10 shows the event time difference distribution between the fast and high-resolution detectors CHOD and CEDAR, which had a fitted standard deviation of 410 ps. This value is compatible with the intrinsic detector resolutions, confirming a negligible contribution from the electronics and the read-out board.

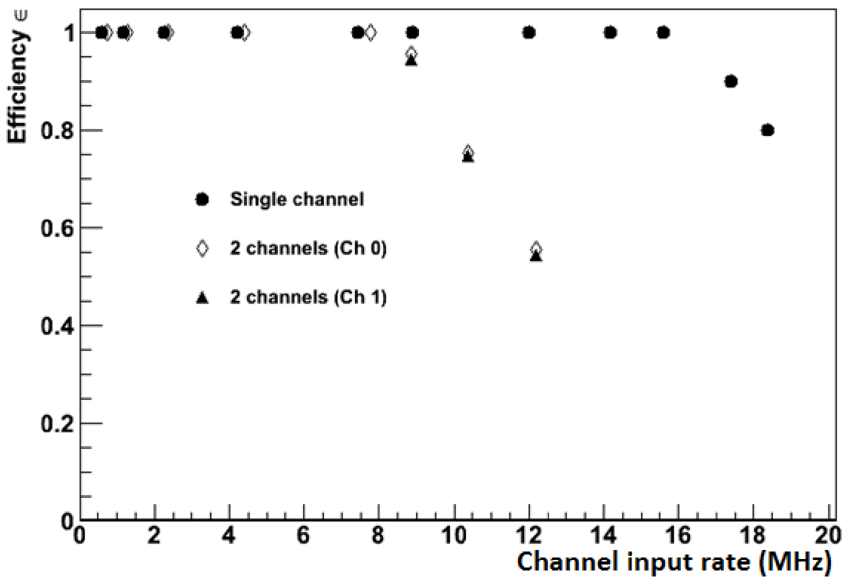


**Figure 3.9:** *TDCB hits efficiency pulsing one and two channels.*

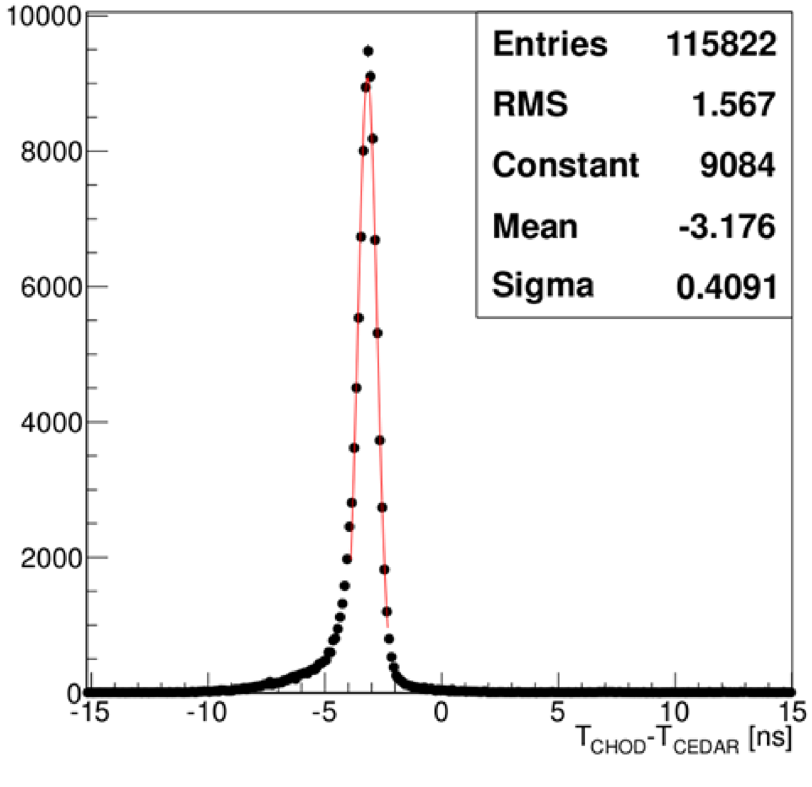


**Figure 3.10:** *Distribution of event time differences between CHOD and CEDAR detectors (2012 test beam).*

**Run 2014**

During the 2014 Run it became evident that, even at low beam intensities (about $\sim 12$ % of the nominal), intensity peaks are present in the beam time structure. These peaks can reach and exceed the DAQ design limit ($\sim 39$ Mword per second per TDC) and the initial version of the firmware was not able to sustain this rate. We measured a firmware limit of $\sim 34$ Mword per second per TDC and when this limit was reached the TDC board was not able to recover smoothly. So we made some improvements to the firmware to reach the design limit and prevent failures when this limit is exceeded. A reshaping of the *TDC control* block allowed to reach the limit of $\sim 39$ Mword per second per TDC and the addition of two new blocks gave the possibility of managing the data frame when the limit is surpassed. These two blocks (which can be enabled via registers) approach the problem in different way:

- **Data limiter** This block limits the number of words written in TDCC output FIFO for a single frame. The maximum value is set by a register; any words (excluding header and trailer) exceeding such value are thrown away, an error word is inserted in the frame and the frame is completed with the correct word count.
- **Data suppressor** When this block is active and one of the input FIFOs storing TDC data becomes almost full (at a pre-defined level), further data from TDCs will be ignored (although TDCs are still read) and an error word is written into the FIFO until the filling of both such input FIFOs falls below a (lower) threshold.

Using either of this two blocks we are able to prevent blocking failures in the TDC Board.

## 3.6 The TEL62 board

The TEL62 board (see figure 3.11) is the common multi-purpose FPGA-based motherboard [24] [14]; it is used in the NA62 experiment both for trigger primitive generation and for data acquisition. It is used by several NA62 detectors with a total number of about 100 installed card.

This board has been developed in Pisa and it is a highly-improved version of the TELL1 board designed by EPFL Lausanne for the LHCb experiment at CERN [25]. The design exhibits a similar overall architecture, but the board is based on much more powerful and modern devices, resulting in more than eight times the computation power and more than twenty times the buffer memory of the original, plus several other improvements in terms of connectivity.

While a large number of TEL62 board are used for the implementation of the calorimetric trigger, we mostly focus here on their use when equipped with TDCB boards.

The TEL62 is a standard 9U Eurocard, with a 16 layers printed circuit with all lines impedance ($50\Omega$) controlled and the clock tree routing done with special care to avoid introducing signal jitter. This motherboard can handle up to 4 mezzanines (like the TDCB), for a total of 512 input channels and houses 5 FPGA of the same type. The 4 *Pre-Processing* (PP) FPGAs are connected to a single *Sync-Link* (SL) FPGA; each PP is directly connected to one of the TDCB mezzanine (see figure 3.12).

Depending on the subdetector, the TDCs can send an amount of data up to some tens of MB/s per channel. Data are organized in packets, each one related to time frames of 6.4 $\mu$s duration. The 4 PP-FPGAs have the role of collecting and merging the data and

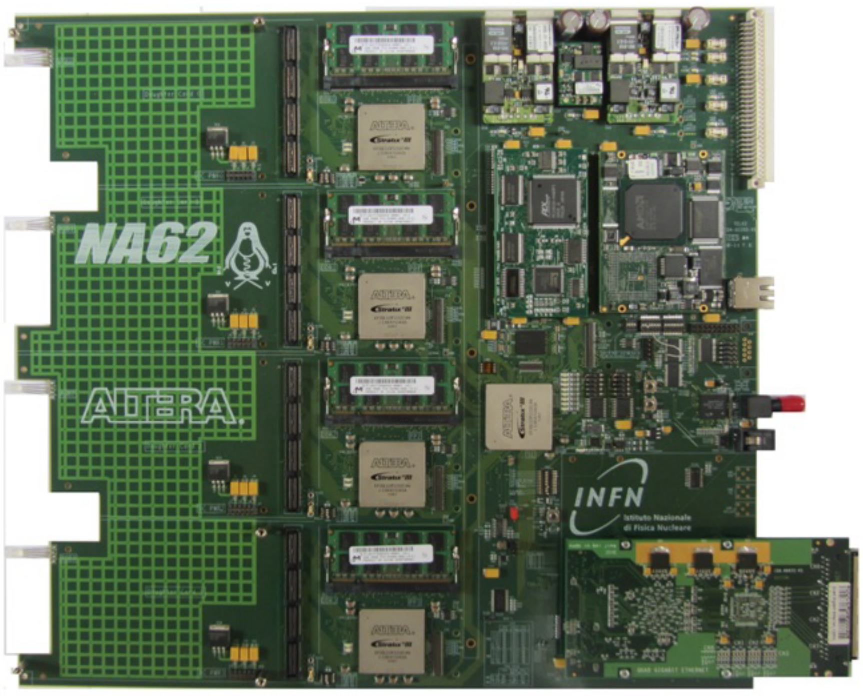


**Figure 3.11:** *The TEL62 motherboard.*

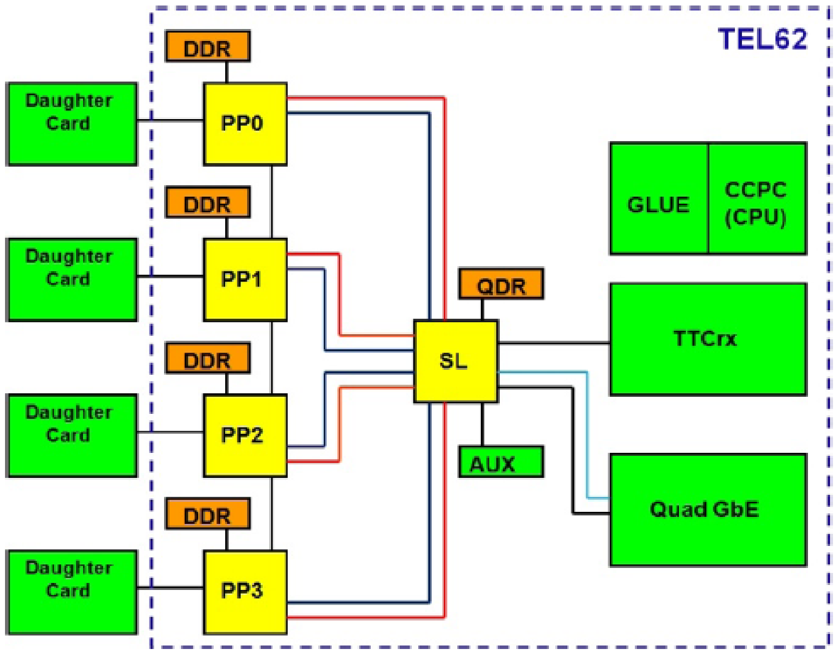


**Figure 3.12:** *Layout of the TEL62 board architecture. FPGAs are shown in yellow, memory buffers in orange, other chip and daughter-card connectors are in green. Lines represent data bus links between devices.*

later organizing them on the fly in a 2 GB DDR2 memory, where each time *slot* corresponds to a single 25 ns time window. Data are stored in the DDR2 memory waiting for a L0 trigger request; at the trigger arrival, the data within a programmable number of 25 ns time windows around the trigger timestamp are read from the DDR2, packed and sent to the SL-FPGA. In the SL-FPGA, the data from the 4 PP are merged, synchronized and stored in a 1 MB QDR SDRAM temporary buffer. Later, the data are extracted for formatting into Ethernet data packets and sent through 4 Gigabit Ethernet links hosted on a custom daughter-card to a computer farm. The board can sustain, by design, 1 MHz of L0 trigger rate for standard-size events.

The fundamental components of the board (see figure 3.13) are shown below:

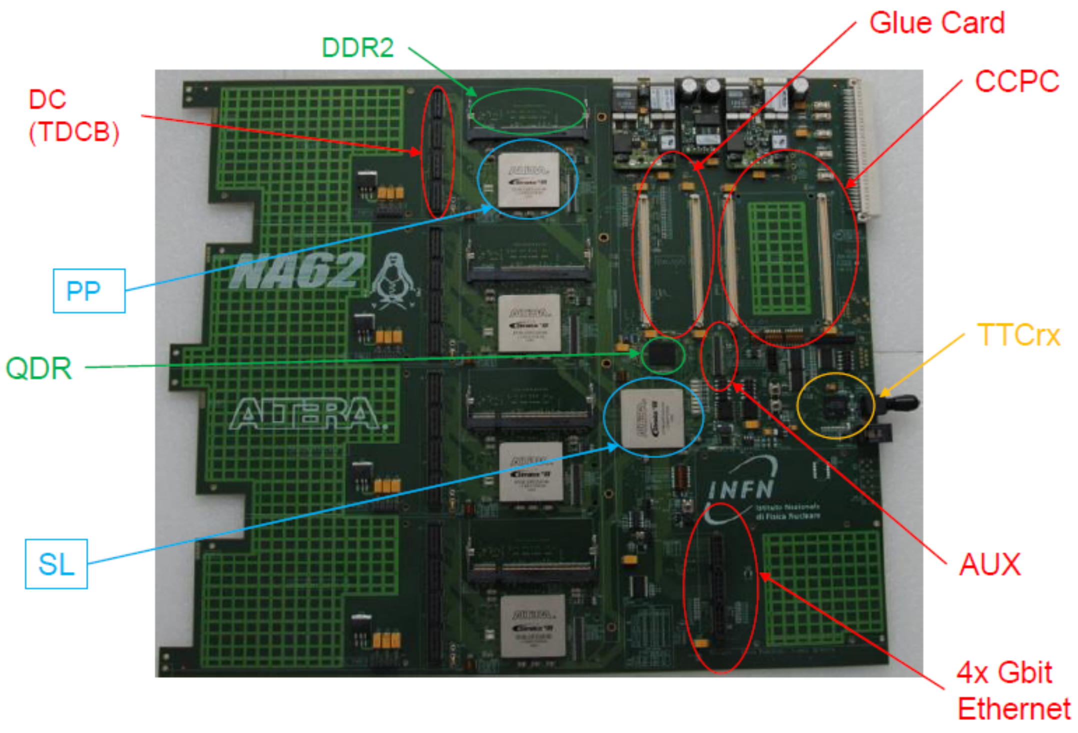


**Figure 3.13:** *Main components of the TEL62.*

- The 4 PP FPGA are Altera®Stratix®III FPGAs EP3SL200F1152 [10], each one containing an embedded memory of 9396 kbit and 198 900 equivalent logic elements. Each PP elaborates the data from the 4 TDCs of one TDCB that is connected through a 200-pin connector, using 4 32-bit wide data buses. Each bus transmits data from one TDC, at 40 MHz for a total bandwidth of about 5 Gbit/s. Each PP is also linked to a DDR2 memory buffer of 2 GB with a 64 bit bus with a clock frequency of 640 MHz, resulting in a memory bandwidth of $\sim$ 41 Gbit/s. Furthermore the 4 FPGA are interconnected in a row (see figure 3.12) with 2 x 16 bit buses (input and output).

- The SL FPGA is also an Altera®Stratix®III FPGAs EP3SL200F1152 [10] with the same hardware characteristics of the PP-FPGA. It is connected to each of the 4 PPs through two independent data and trigger primitives buses, 32 bits wide and operating at a clock frequency of 160 MHz, for a total bandwidth of about 10 Gbit/s per connection. All signal lines are properly terminated and equalized in length to handle this clock speed; moreover we added in the firmware a set of time constraints to guarantee the functionalities of the board. The SL-FPGA is linked to a Quad Data Rate (QDR) Synchronous Pipelined Burst SRAM of 1 MB, read and written at 120 MHz clock frequency and used like temporary buffer in the data flow. The data packets generated in the SL are sent to a custom Gigabit Ethernet (GbE) mezzanine.

- The custom mezzanine is a Quad GbE [35] (the same card used in the TELL1); this card hosts four 1 Gbit Ethernet channels used to send data packets to the PC farm, trigger primitives to the L0 Trigger Processor (L0TP) or data packets to other TEL62s in a daisy chain configuration.

- A standard optical TTC link distributes the main 40 MHz clock and the L0 triggers, which are then decoded by a TTCrx chip [21].

- An on-board QPLL [43] reduces the signal jitter of the input clock to few tens of ps.

- A commercial Credit Card PC (CCPC) [38] is a SM520PC SmartModule produced by Digital-Logic, Inc. running a Scientific Linux operating system mounted from an external CCPC boot server through a dedicated Ethernet link, and a custom input/output interface card named Glue card [23] (same as in the TELL1 board) handle the slow control, the monitoring and the configuration of the TEL62. These two mezzanine cards communicate among them through PCI buses. The Glue card use and distribute to the other devices and connectors three different communication protocols: ECS is a custom parallel bus used to configure or read the FPGAs' internal registers and to control or monitor the board functionalities, I2C is used to control the TDC Board and JTAG allows to configure and program remotely the FPGAs on the TEL62 and on the mezzanines, by loading the firmware on the respective EEPROMs.

- Two 64 Mbit EEPROMs (Altera®EPCS64 [9]), one for the 4 PPs and one for the SL, store the FPGA configurations.

- Dedicated auxiliary connectors with two independent 16-bit buses allow to link several TEL62 in a daisy chain: it could be useful to merge the L0 trigger primitives from different TEL62 of a single detector and to use only one Gigabit link, to send them to the L0TP.

### 3.6.1 The TEL62 firmware

Like for the TDCB, we use HDL Designer®, Modelsim®and Quartus®II to develop the firmware of the PP-FPGAs and the SL-FPGA.

#### The PP-FPGAs

All the 4 PP-FPGAs use the same firmware, which can be divided in 4 parts: the data flow upstream the DDR (inside the red square in figure 3.14), the one downstream the DDR (inside the blue square in figure 3.14), the trigger primitive flow (inside the green square in figure 3.14) and the part related to testing of the FPGA and the connections (inside the brown squares in figure 3.14).

**The upstream data flow.** The PP-FPGA receives the data from four buses linked to the TDCB mezzanine card (one per TDC). Each bus fills a dedicated derandomizer IB (Input Buffer) FIFO with a clock frequency of 40 MHz. The data from the TDCB are packed in 6.4 $\mu$s data frames. The OB (Output Buffer) block merges, at 160 MHz, data of frames with the same timestamp from the four IB buffers; a bigger frame containing all the hits of the daughter-card is created, and it is stored in two copies of the Output Buffer (OB) FIFO, one of them reserved and used for monitoring purposes. Then the data packets are compressed to be stored inside the DDR2 memory. This is one of the most complex part of the firmware, indeed it has been the subject of two important revisions during the evolution of the firmware itself. The difficulty consists in packing the data in order to minimize the number of accesses to the DDR2. Each access requires a relatively large time (3 clock cycles for a write access and 33 clock cycle for a read access) that could slow down the data flow; I have to maximize the time efficiency of DDR2 writing and reading operations because the structure must sustain an input data flow of 160 Mword per second and a trigger rate of 1 MHz. The first version (V1) of the firmware organized the data of each 6.4 $\mu$s frame in fixed-size area related to 25 ns time slots, for a total of 256 slots. Each time slot was reserved a memory amount equal to 128 x 32-bit words, so each frame required 1 Mbit of memory. I implemented two of these memories to avoid interruptions in the data flow: while one memory was written by the Data Organizer, the other was read by the DDR writer.The Data Organizer vhdl code - as an example - is shown in (7) I reserved 128 words per 25 ns time slot, as a compromise between the need to sustain high instantaneous rates (i.e. up to 2.56 GHz of hit) and the available memory in the FPGA (2 Mbit is about a quarter of the total memory). Half of the DDR2 (1 GB) was reserved for memory data storage and half was used for 8-bit counters, that indicated the actual number of words of each time slot. In the second half of the DDR, the counters were organized as 8 DDR locations of 256 bit (16 counters per location) per frame. The use of data counters optimized the DDR reading because only the relevant part of each time slot was read. These numbers defined the maximum latency of the system, that is the time after that data is overwritten (and lost if a L0 trigger did not arrive).

$$\frac{1\,\mathrm{GB}}{1\,\mathrm{Mbit}\,/\,6.4\,\mu\mathrm{s}} = \frac{8\,\mathrm{Gb}}{1\,\mathrm{Mbit}\,/\,6.4\,\mu\mathrm{s}} = 51.2\,\mathrm{ms}$$

At L0 trigger arrival, a module, the *DDR reader*, reads up to three 25 ns slots of data in a single access and packed the data to send them to the SL-FPGA.

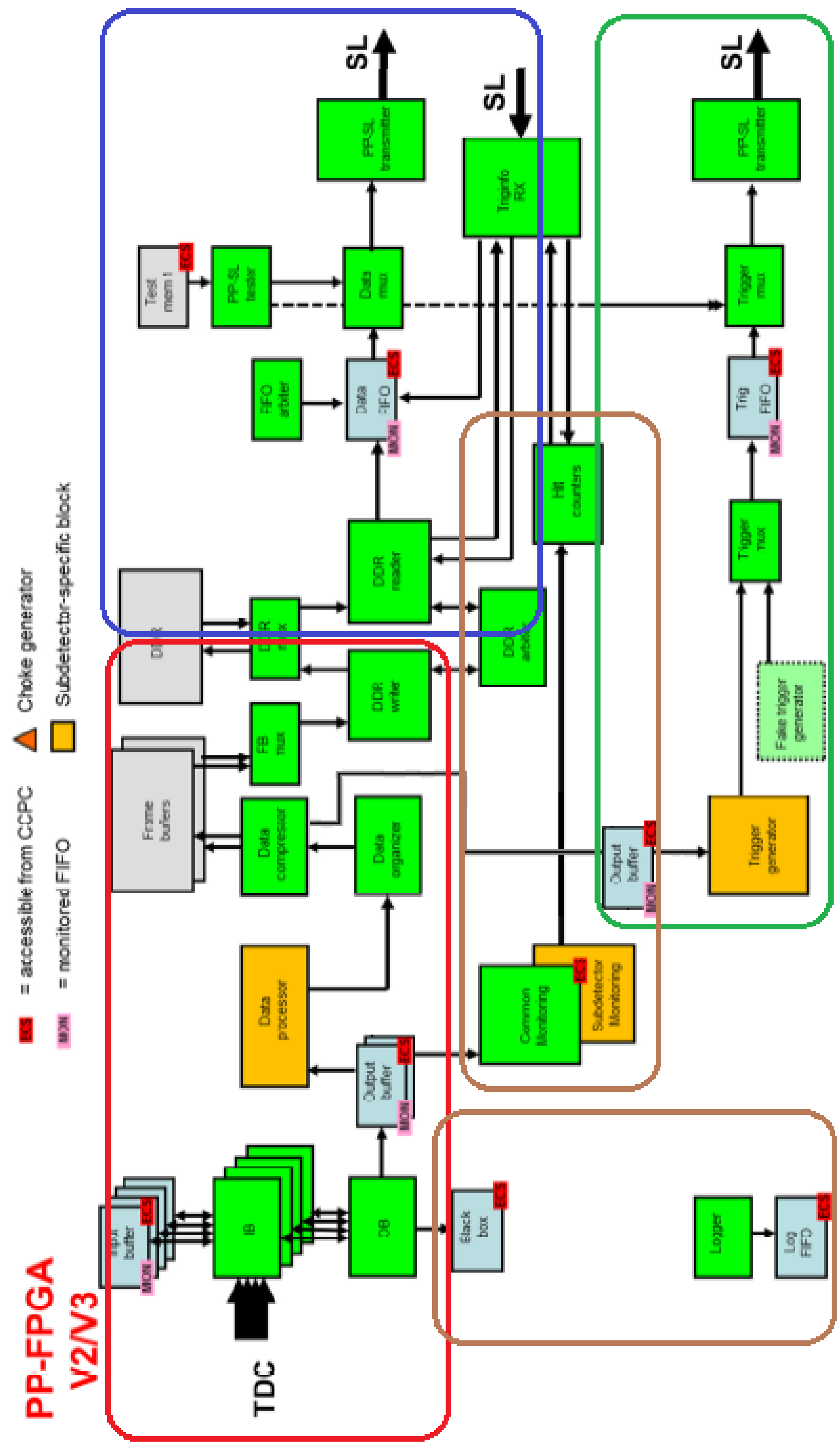


**Figure 3.14:** *Block diagram of the PP-FPGA firmware version 3.*

After some test I realized that this data organization presented some problem at high trigger rate: the *DDR writer* needed 257 accesses (256 time slots + the counter part) to write one frame and the *DDR reader* module had to perform 2 memory accesses (counter + data) for each trigger. The DDR2 access latency for a write operation is 3 clock cycles (the clock period is 6.25 ns) and it needs 1 further clock cycle per DDR line (8 words of a timeslot) to be written. So for each 6.4 $\mu$s frame the total write time was:

$$all\, slots\, empty\ :\ (3\ \times\ 257\ +\ 8\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 5\,\mu\mathrm{s}$$

$$design\, rate\,(160\, MHz\, words)\ :\ (3\ \times\ 257\ +\ 256\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 6.4\,\mu\mathrm{s}$$

$$all\, slots\, completely\, full\ :\ (3\ \times\ 257\ +\ 4096\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 30\,\mu\mathrm{s}$$

The DDR2 access latency for a read operation is 33 clock cycles and it has to read 49 lines (16 line per 3 slot to read the data and 1 line to read the counters). The total read time for a trigger was:

$$empty\, slot\ :\ (33\ +\ 1\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 0.2\,\mu\mathrm{s}$$

$$design\, rate\,(160\, MHz\, word)\ :\ (33\ \times\ 3\ +\ 49\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 0.9\,\mu\mathrm{s}$$

$$slot\, not\, empty\ :\ (33\ \times\ 3\ +\ 49\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 0.9\,\mu\mathrm{s}$$

These values were not compatible with the design trigger rate limit of 1 MHz. Indeed at 1 MHz of trigger rate, the time to write and read an empty frame should be about 6.3 $\mu$s (5 $\mu$s to write an empty frame and 0.2 $\mu$s $\times$ 6.4 trigger in average) that is almost all the available time (6.4 $\mu$s) without considering the DDR refresh time and other operation latencies. The L0 trigger limit measured for this first version was about 700 KHz of triggers with empty events (the limit decreased with non-empty events). Another problem of this firmware version was the limited number of slots that could be read for each trigger, because some detectors (for example the LAV) need more than 3 slots to measure the hit time over threshold.

For these reasons I developed a second version of time DDR data organization. I introduced a new block, the *data compressor* (see figure 3.15), to better pack the data frame.

In this version, each 25 ns time slot is not allocated in a pre-defined space: data was written in adjacent 256 DDR locations with flags pointing to the end of the slot. The memory space allocated for a frame remained the same as in the previous version ( 1 Mbit for 6.4 $\mu$s) but the second area of the DDR2 now contained 8-bit addresses instead of counters. The addresses, related to a time slot, represented the starting address of the next non-empty slot in the DDR2 data area. Figure 3.16 shows an example of the compressor output for a frame with 3 non-empty time slots: slot 1 has 3 data words, slot 2 has 9 words and slot 5 has 2 words; slot 0 is empty so the address of the next non-empty slot is 0, instead for slot 2 the next slot address is 3.

This version resulted in a vast improvement because the *DDR writer* block needed only one access to write all the data part and one to write the address part. So for each 6.4 $\mu$s frame this version had a total write time of:

$$all\, slots\, empty\ :\ (3\ \times\ 2\ +\ 8\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 90\,\mathrm{ns}$$

$$design\, rate\,(160\, MHz\, words)\ :\ (3\ \times\ 2\ +\ 128\ +\ 8\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 900\,\mathrm{ns}$$

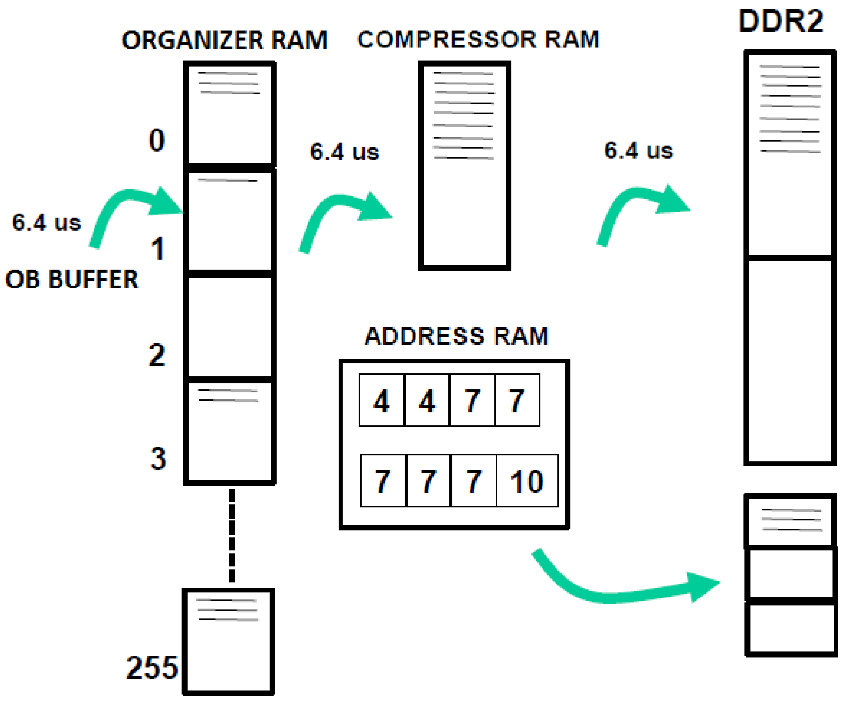
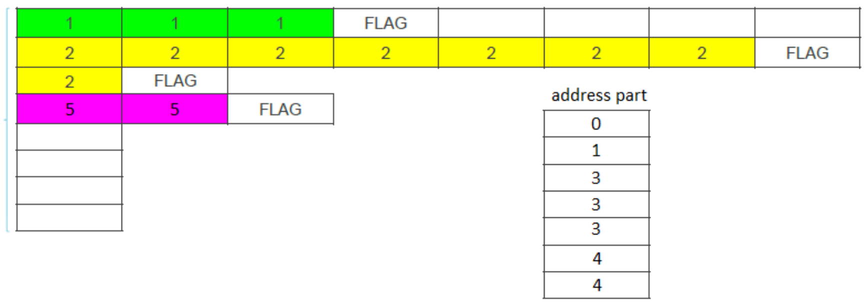


**Figure 3.15:** *Layout of the second version of DDR organization.*



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | FLAG | | | | |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | FLAG |
| 2 | FLAG | | | | | | |
| 5 | 5 | FLAG | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

| address part |
|---|
| 0 |
| 1 |
| 3 |
| 3 |
| 3 |
| 4 |
| 4 |

**Figure 3.16:** *Example of data distribution in the DDR2 for a frame with 3 not empty slots, firmware version 2.*

$$all\, frame\, completely\, full\ :\ (3\ \times\ 2\ +\ 256\ +\ 8\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 1.7\,\mu\mathrm{s}$$

The *DDR reader* part was also improved removing 3 slots limit (in this version the limit was 30 time slots), only 1 access and 2 reads to get all the needed addresses and only one access (in some particular cases 2 accesses) to have all the data words of the time slots triggered. The total read time for a trigger and 3 time slots was:

$$empty\, slot\ :\ (33\ +\ 1\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 0.2\,\mu\mathrm{s}$$

$$design\, rate\, (160\, MHz\, word)\ :\ (33\ \times\ 2\ +\ 5\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 0.4\,\mu\mathrm{s}$$

$$slot\, not\, empty\ :\ (33\ \times\ 2\ +\ 49\,)\ \times\ 6.25\,\mathrm{ns}\ \sim\ 0.7\,\mu\mathrm{s}$$

Version 2 was able to sustain a trigger rate of 1 MHz.

During the 2014 RUN even this version (V2) was shown to be unable to reach the required rate. The PP should be able to sustain the same rate of the TDC Board i.e. $4 \times 39\text{Mword/s} = 156\text{Mword/s}$ while it reached only 116 Mword/s. This high rate could be reached even at intensities below nominal due to the highly non-uniform spill structure (see figure 3.17) that presents a 50 Hz modulation and some high peaks (see figure 3.18).

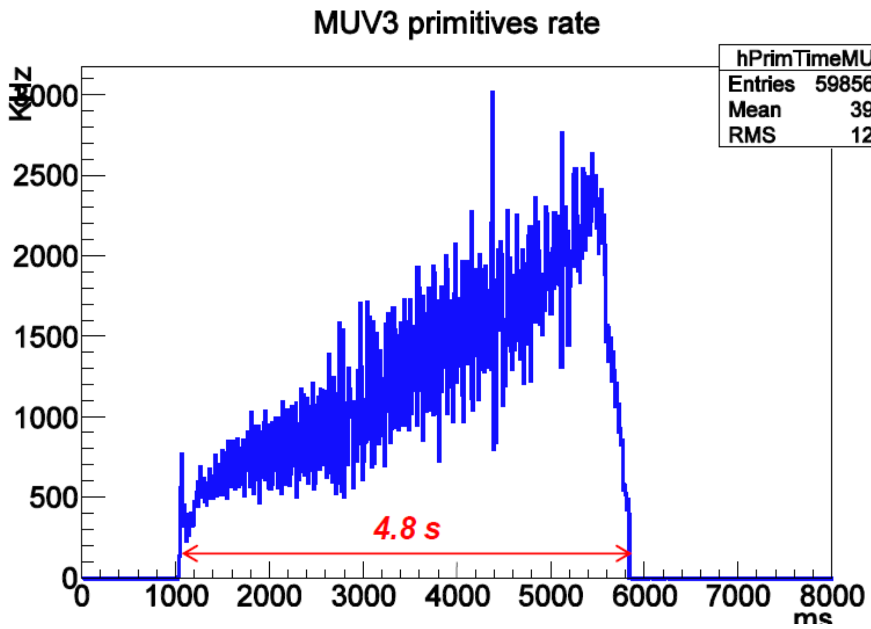


**Figure 3.17:** *Example of burst structure.*

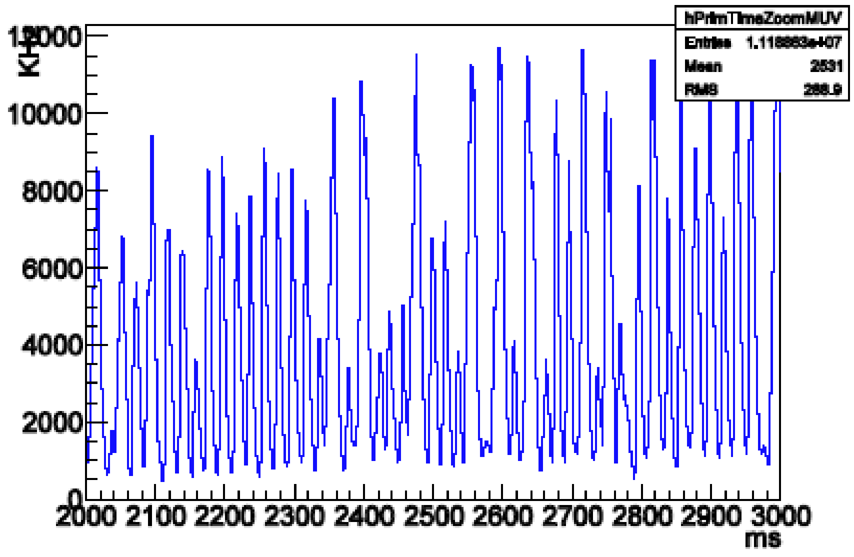


**Figure 3.18:** *Zoom of the burst structure showing the large 50 Hz modulation and the intensity peaks.*

I re-designed again the core firmware of the TEL62 to obtain the (final) V3 version. To stand rate, the data had to be more compressed in the DDR and I had to remove the end-of-slot flag (it wasted 1 clock cycle per each non-empty time slot). The data words of each slot are now one after the other without empty space or flags and in the address word I added 4 bits to specify the position of the slot first data within the row (see figure 3.19). Having reached this compression, it was even possible to increase the capacity of a frame up to 4096 words compared to the 2048 of the V2. The code - as an example - is shown in (7)

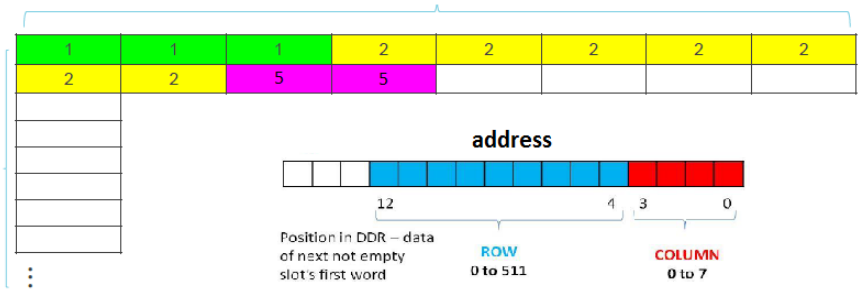


**Figure 3.19:** *Example of data distribution in the DDR2 of a frame with 3 slot not empty, firmware version 3.*

The 2015 Run showed the design rate achievement for this firmware version.

**The downstream data flow.** The firmware part after the DDR2 takes care of elaborating the trigger requests from the SL-FPGA and reading the corresponding data from the DDR2. Trigger requests are received by the *Trigger Receiver* (TrigRX) block that decodes the trigger type and the trigger timestamp. The trigger type can be either physics or a special purpose one. In case of physics triggers (32 possible different types), a request is forwarded to the *DDR Reader* which starts to read a programmed number of 25 ns time slots around the trigger timestamp (the centre of these windows with respect to the trigger time can be also chosen by register). In the version 3, the firmware can extract up to 32 time slots, for a total of 800 ns: such large window is useful to test and study the detector response. The *DDR Reader* vhdl code is complex and it is was obtained through five finite state machines interconnected. The code - as an example - is shown in (7). The DDR data read are stored in the *Data Output FIFO* to be transferred to the SL-FPGA.

**Monitoring and testing tools.** I implemented a lot of debugging and testing tools in the firmware to check its performance during the data acquisition. The CCPC can access the FPGAs via the ECS bus and read the internal registers and the status of FIFOs or memories. An useful tool is the logger system that stores in memory informations about specific error conditions, to be later checked using the CCPC. Time was dedicated to implement and optimize a tool to test the communication between PP-FPGA and TDCC-FPGA or between PP-FPGA and SL-FPGA. It's important to find a perfect coupling between the input and output lines of different FPGA, since a single error can cause some word to be misinterpreted. To test the communication system I implemented random generators using the same seed: the first sends data from the FPGA to another and the second checks the correctness of the arriving data.

**Trigger primitive generation** The compressor module has a second output toward a FIFO, used to provide data to trigger primitive generators. Since the FIFO is written after the organizer module, in such FIFO all the data of a 6.4 $\mu$s frame is organized in a sequence ordered of 25 ns slot. The ordering simplifies the following elaboration because it is easier to work on 25 ns slots of data to obtain more precise clusters (like 3 ns of resolution) or further data sorting. Only detectors involved in the L0 trigger developed this firmware part: RICH, CHOD, new-CHOD, MUV3, LAV12. Some firmware handles channels matching or time slewing correction (see next chapter). After primitive building, these data are sent to the SL-FPGA for further processing.

#### The SL-FPGA

Also the SL-FPGA firmware can be divided in four parts: the trigger and data flow, the trigger primitive generation and the testing tools. Figure 3.20 shows the firmware blocks dedicated to the data flow (red square), the trigger flow (inside the blue squares) and the testing tools (inside the brown squares); figure 3.21 shows the trigger primitive generation flow.

**Trigger and data flow** The SL-FPGA receives the trigger, the start of burst (SOB) and the end of burst signals from the on-board TTCrx chip. The *TTC interface* block handles the communication with the on-board TTCrx chip. It decodes the signals and propagates the result to the remaining parts of the board. In this way at the time of SOB all the blocks are reset and the data acquisition starts, while the EOB signal stops the data acquisition. An internal timer, reset with the SOB signal, produces a timestamp that is assigned to each trigger and is used for debug and monitoring reasons. The trigger requests, from the *TTC interface*, are dispatched towards the PP-FPGAs where the corresponding data are read out. The data belonging to a trigger coming from the four PPs are merged in the SL-FPGA *data merger* block, that works in the same way of the OB block in the PP. The resulting event is stored in the *Event Data FIFO*. Multiple events can be packed in a single Multi-Event Packet (MEP) to optimize the output link bandwidth, limiting the fraction used by the packet header (42 B) rather than the payload. After MEP assembly the packets are temporarily stored in the external 1 Mbit QDR memory, written in a circular way: at each time it contains several readable for debugging purposes. The MEP packet is then extracted from the QDR, formatted in a UDP packet and passed to the *SPI3 interface*. This module takes care of the communication with the GbE output mezzanine that finally sends the packet to the PC farm.

**Trigger primitive generation** The trigger primitives follow a similar path to that of the data. The primitives coming from the PPs are merged by the *primitive merger* block and then they undergo further processing stages like multiplicity counting in the RICH or the cluster identification in the LAV. A communication interface between different TEL62 boards is under development: it could be useful for the generation of trigger primitives for detectors that use more than one TEL62 (like LAV, RICH and KTAG).

The triggers, like the data, are packed in Multi-Trigger Packets (MTPs) which are formatted as UDP packets and sent via dedicated Ethernet ports of the GbE to the L0 Trigger Processor.

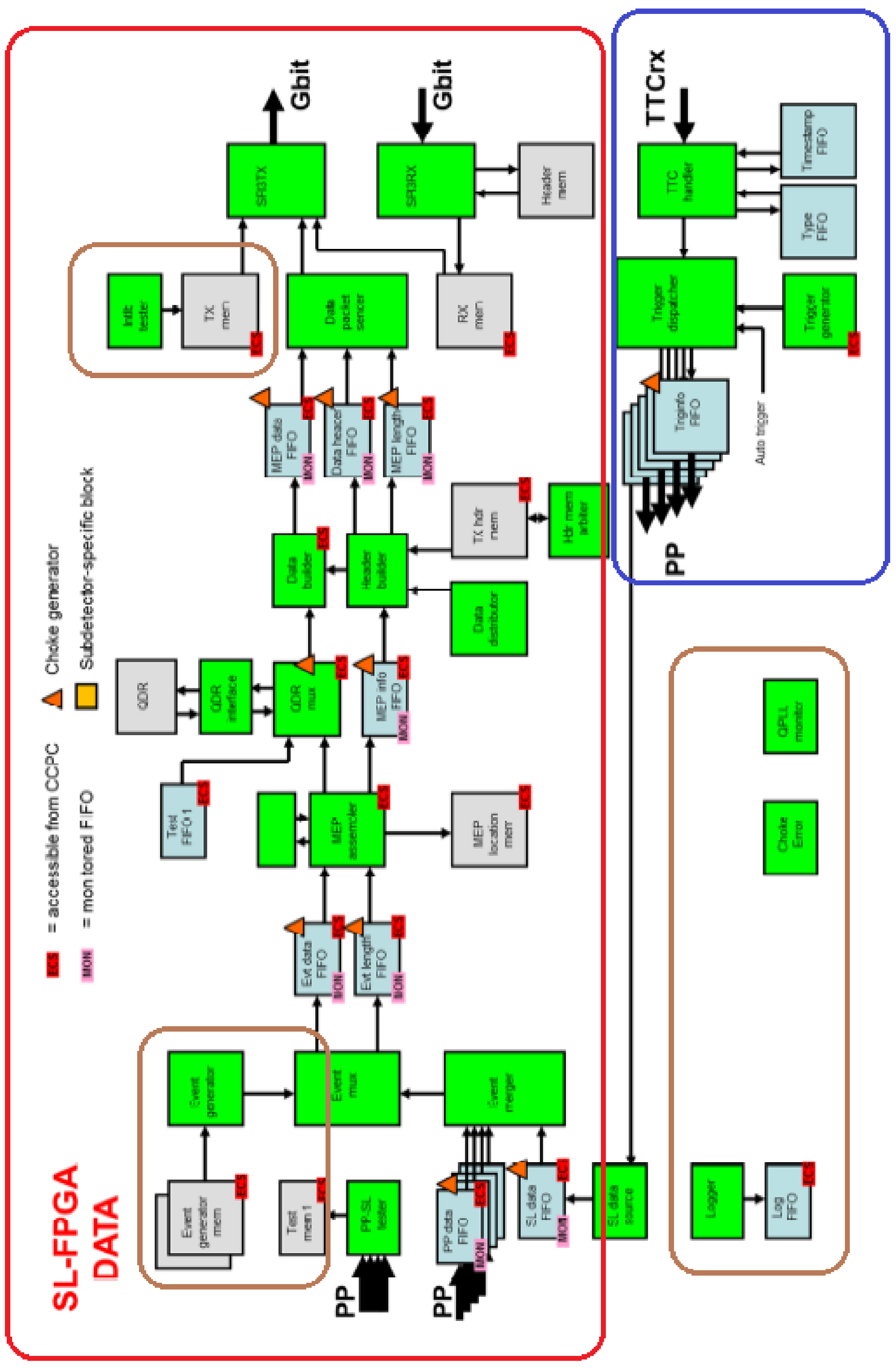


**Figure 3.20:** *Data flow block diagram of the SL-FPGA firmware.*

**Monitoring and testing tools.** The SL-FPGA, like the PP, has a lot of modules dedicated to testing the board interconnections and to monitoring the data acquisition status. It is possible to generate trigger patterns via CC-PC control to test the trigger and the data readout system. A tool in the SL allows to produce fake trigger primitives, overriding the primitive generation in the PP-FPGA, for the development and testing of the L0 Trigger Processor.

### 3.6.2 TEL62 tests

The performances of the TEL62 were tested at the end of 2013 during one of the TDAQ commissioning phases. We tested the output data bandwidth of the board (see figure 3.22) with a standalone data acquisition system, using the TDC emulators (inside the TDCC-FPGA) for data production and the LTU standalone mode[1] as trigger source.

Results were in good agreement with expectations: the small gap between the expected upper rate limit value and the measured values indicates that the firmware could be further optimized even if there is some irreducible limitation introduced by the data formatting and other operations that prevents to reach the theoretical link bandwidth.

Figure 3.23 shows the maximum data payload (that is without including IP and UDP packet headers) for a given trigger rate using 4 Ethernet ports and assuming one data packet for each incoming trigger. The measured values lie on a hyperbola: the product between the trigger rate and the payload size is the effective data output bandwidth.

The TDAQ system (TDCB and TEL62) was extensively tested through some Technical Run, Test Beam and Data Taking (Run). In the last Run the system still showed some rate inefficiency, which is currently under investigation, in the part that selects the Ethernet ports and sends the data packet to the GbE output mezzanine.

The measured time resolution is compatible with the intrinsic detector resolutions, with a negligible contribution from the TDCB. The TEL62 reached the design performance in the DDR2 memory data storage (160 Mword per second) and in the data extraction with a trigger rate up to 1 MHz.

[1]In the standalone mode, the LTU can emulates the L0TP protocol and generates programmable trigger sequences.

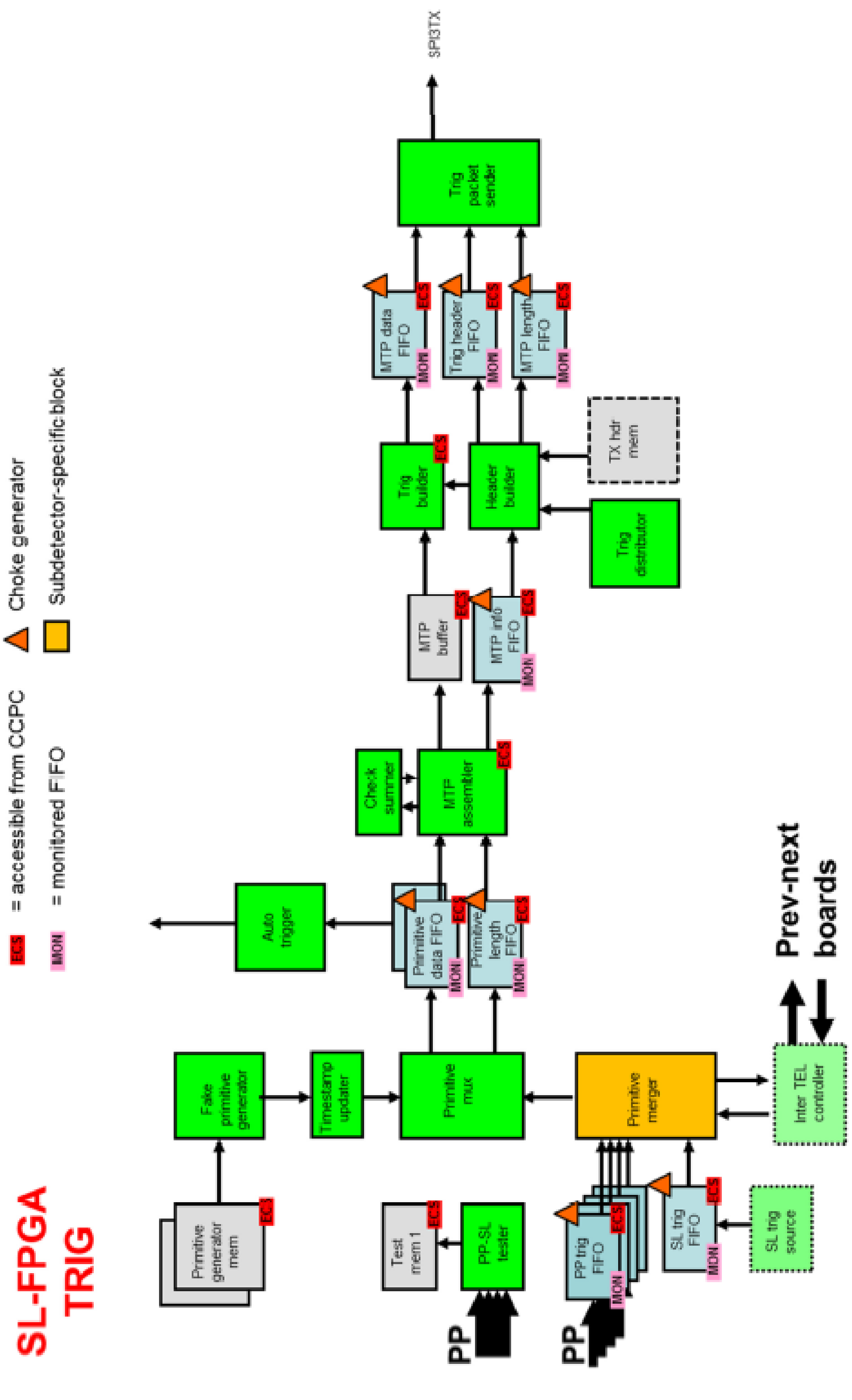


**Figure 3.21:** *Trigger primitive generation flow block diagram of the SL-FPGA firmware.*

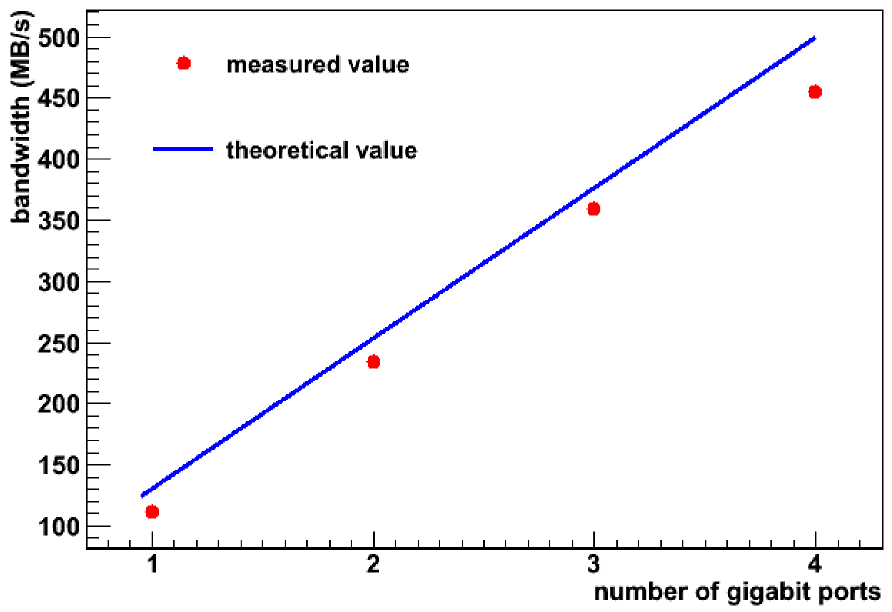


**Figure 3.22:** *TEL62 output bandwidth as function of the number of used Gigabit ports. The blue line indicates the expected upper limit value (1 Gbit/s = 125 MB/s per port), while red points refers to the measured values.*

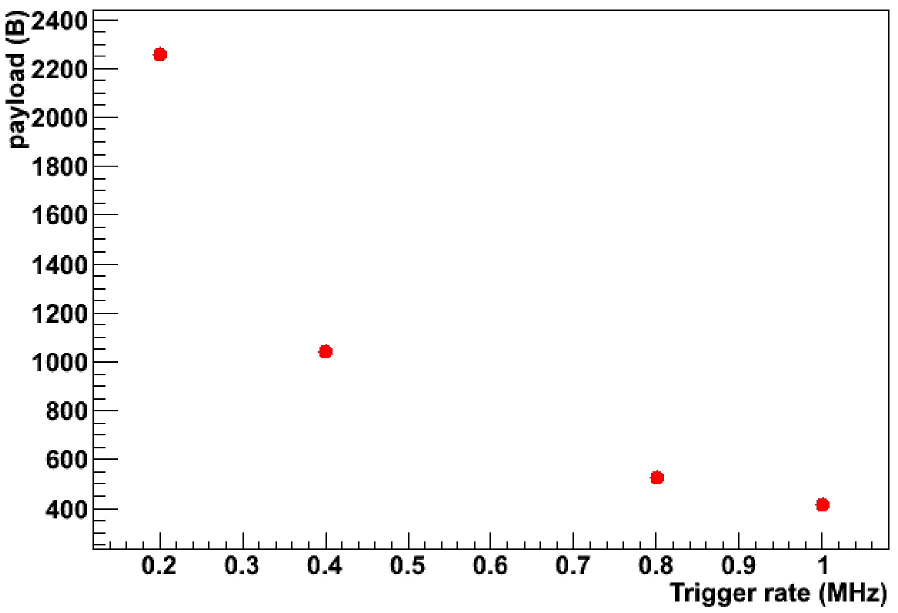


**Figure 3.23:** *TEL62 data packet maximum payload size as function of the trigger rate using 4 output links.*

CHAPTER 4

# The trigger approach in NA62

## 4.1 Introduction

In NA62 the high rate of events and the presence of 12 sub-detectors results in a high output data rate that it is impossible to store on disk without filtering. A multi levels trigger is therefore needed, which should identify the events to be saved and reject the rest. The NA62 trigger is made of three logical levels:

- L0: a hardware trigger, based on the input from few sub-detectors. Rate reduction from 10 MHz to 1 MHz, with a maximum latency of 1 ms;
- L1: a software trigger, based on information computed independently by each sub-detector system. Rate reduction from 1 MHz to 100 kHz;
- L2: a software trigger, based on assembled and partially reconstructed events, in which informations from different from sub-detectors are used. Rate reduction from 100 kHz to about 15 kHz;

## 4.2 L0 hardware trigger

The L0 hardware trigger will be mainly based on input from the CHOD, the MUV3, the LKr, the RICH and the LAV. These detectors will continuously evaluate their incoming data for the fulfillment of certain condition (called *primitives* in TDAQ) and associated time. Trigger primitive will be packed in Multi Trigger Packet Format (MTP) [30] and sent through standard Ethernet links to L0 Trigger Processor (L0TP).

The L0TP time-matches different sub-detectors primitive checking if L0 trigger conditions have been satisfied, in case of positive response, L0TP will issue a L0 trigger.

### 4.2.1 L0 Trigger for the RICH

My work is focused on the L0 trigger firmware for the RICH. Two version of the firmware were developed: one used during the 2015 NA62 data taking (*standard version*) and one used in 2016 NA62 data taking (*NEW RICH version*). Before describing in detail the firmware, I'm going to describe in more detail how the L0 RICH trigger works.

The standard RICH L0 trigger is based on hits multiplicity or SuperCell (digital OR of 8 PMTs) multiplicity.

In case hits multiplicity is used four TEL62 are needed to fully cover the 1952 RICH channels. Each TEL62 receives data only from half flange, Figure 4.1 shows the different area assigned at each board.

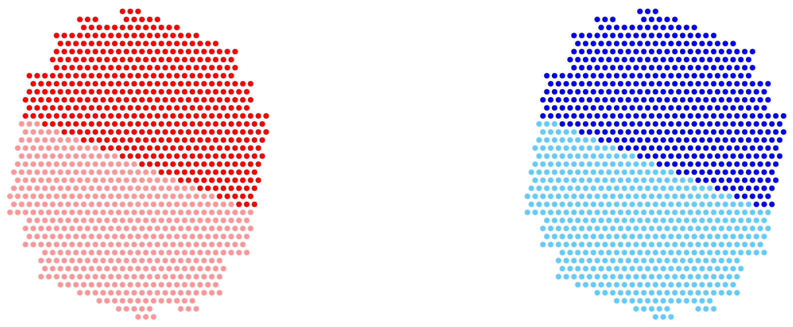


**Figure 4.1:** *The Figure shows how RICH channels are divided between four TEL62s, two dedicated to the Jura side flange (blue ones) and two to the Salève side flange (red ones).*

Instead if SuperCells multiplicity is used, a fifth TEL62 is needed.The fifth board receives the data from RICH SuperCells (digital OR of 8 channels), so only 244 readout channel are needed.

The trigger primitive of the RICH is based on SuperCells multiplicity and encoded in the way described below:

- R1 number of SC hits between $2 \div 9$;
- R2 number of SC hits between $9 \div 33$;
- R3 number of SC hits between $33 \div 59$;
- RS is the single ring trigger and is R1 OR R2;
- RM is the multi ring trigger R2 OR R3;
- MB for minimum bias trigger R1 OR R2 OR R3;

## 4.3 The RICH primitives generator firmware: standard version

The RICH sub-detector primitives generator firmware is used to extract trigger information from the data and send to the L0TP a digest of this information in order to decide if the data are significant or not. A preliminary version of the RICH firmware was developed for the 2015 data taking, in particular the trigger algorithm was based on the clustering of hits coming from the TDCB boards and on the computation of cluster average time and multiplicity. The clustering of the hits is done in the PP-FPGA (3) and then the sub-cluster produced by different PPs are combined inside the SL-FPGA (see figure 4.2). Both time window used for clustering and sub-cluster merging are programmable such as other firmware parameters.

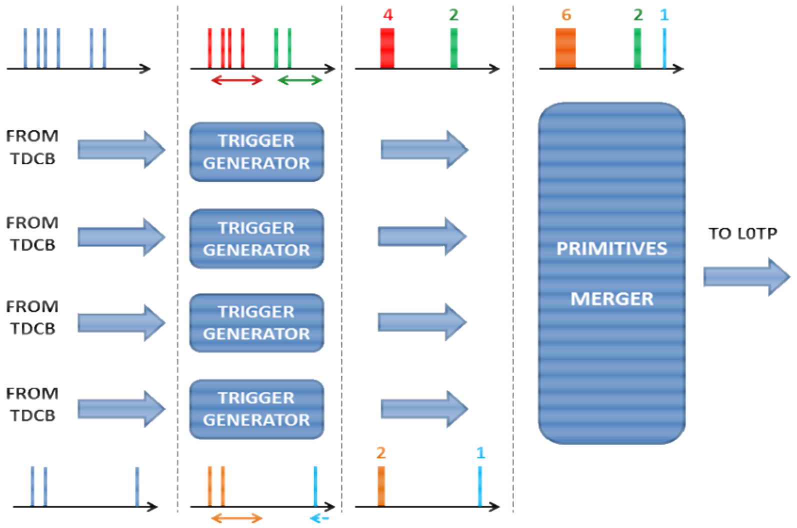


**Figure 4.2:** *RICH firmware scheme.*

### 4.3.1 PP firmware

The firmware developed for the PP-FPGA implements the clustering operation on the hits coming from the TDCBs, the most important are:

- t0 correction: a correction time is added to the hit time in order to take into account the average response time of each channel (see figure 4.4). By applying only positive time correction, all the channel answer time will be aligned to the slowest, however in the primitive builder block another time correction is available to compensate this offset. This correction time is read from a memory accessible via ECS (3) so it is possible to update this corrections at any time. To take into account the possibility for a hit near the data frame edge to slide to the next frame due to the correction, the "current" data frame is buffered and it will be written to the output only after the reception of the entire "next" data frame. In the t0 correction only the leading words are considered while the trailing are dropped. To avoid useless data transmission, empty data frames are dropped in this module (TBM);

- Sorter: due to the algorithm used to read the data from the TDC in the TDCB board, the hits arrive to the PP-FPGA in a disordered way with respect to the arrival time of the signals to the input of the TDCs. To simplify the clustering logic, a sorting of the hits is performed. The sorter module is based on a chain of sorter unit modules, each one of them store an hit in an internal buffer and put on the output the smallest between the stored data and the input one. Due to the daisy chain connection scheme the sorter module has some clock cycle of latency that can be computed as $2 \times n$ with n number of sorter units in the chain. The number of sorter units should be set equal to the maximum number of leading in a data frame (to limit tha latency introduced by this module a reasonable peak of rate should be considered instead of the maximum TEL62 rate);

- Data Converter: this module keeps the clustering logic simple, in fact having the hits divided into data frames, for hits on the edge of a data frame, it is complicated to check the time distance (both timestamp distance and fine time distance should be considered), so the data are converted from this structure to self consistent absolute time by concatenating timestamp and fine time (considering some bit overlapping). The bit width of the data is increased (from 32 to 53 bits), the header and the footer are maintained and, for this two types of words, the additional bit are added right after the word ID (and are all equal to 0);

- Cluster finder: in this module the hits are clustered and either the cluster average time and the multiplicity are written to the output. When the first hit arrives its time is set as the start time of the cluster, subsequent hits are compared with him and if the time distance is less than the clustering time the hit time is added to an accumulator register and the multiplicity counter is increased by one. When a hit with a time “too far” from the cluster start time arrives the cluster average time is computed, and the current hit will be the first of a new cluster. There are two special case to take into account when the cluster average time is computed: one or two hits cluster.If the cluster is composed by one hit than cluster average time is the hit time, while for the two hits cluster the average time is computed by shifting the time sum by one position to the right. For a generic cluster (multiplicity > 2) a division is performed and the latency of this operation is about 24 clock cycle. The data frame header and footer are used in this block only to check data correctness: a timestamp word is expected after a words count word and the number in the words count word is compared with an internal counter to check that all the data words has been received and processed;

- PP output data format (see figure **??**): it is composed by two 32 bit data words: in the first one, the bit from 0 to 7 contains the cluster average fine time with a resolution of 100 ps (full resolution) and the bits 16 to 23 contains the cluster multiplicity, while the second words contains the timestamp of the cluster at a resolution of 25 ns;

### 4.3.2 SL firmware

The clusters coming from different PPs (sub-clusters) are merged into SL firmware. The cluster generated is used to produce the corresponding trigger primitive. The merging operation is done by checking the time distance of the sub- clusters, if two or more

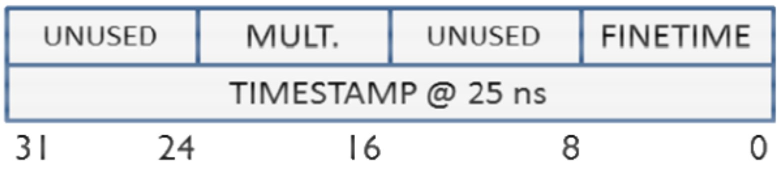


**Figure 4.3:** *PP firmware: t0 corrector block diagram.*

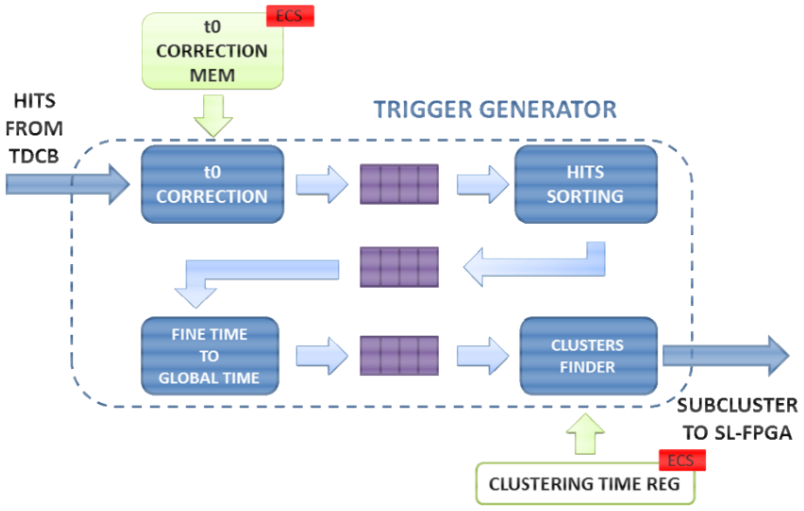


**Figure 4.4:** *PP firmware: trigger data format.*

sub-cluster are closer in time, the sub-clusters multiplicity are summed and the cluster time is computed by applying a weight average operation. The trigger word is then generated from the cluster information (see figure 4.5).

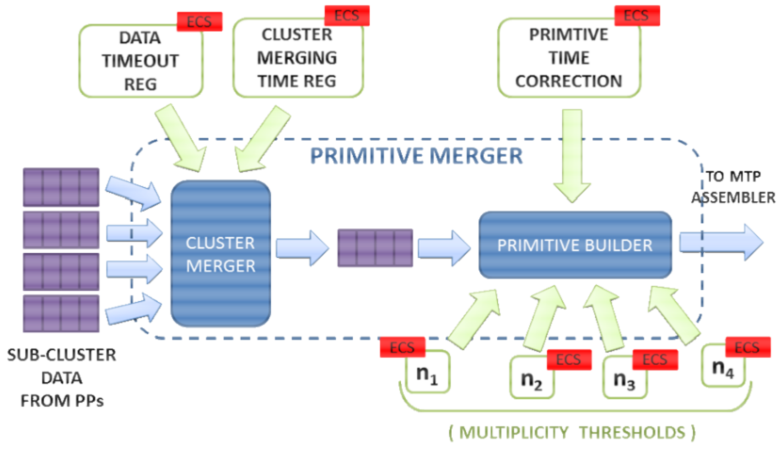


**Figure 4.5:** *SL firmware block scheme.*

In detail:

- Cluster Merger: used to merge the information from the sub-clusters coming from different PPs. To merge two or more sub-cluster, the smallest time among the words read from the PPs is computed. Then, by a comparison of the smallest time with all the others, the decision of which sub-cluster to merge is taken. To merge sub-clusters each average time is first multiplied by the multiplicity and then summed to the others, the result is then divided by the sum of the multiplicity (cluster multiplicity). The cluster information are then written in the output fifo.

- Primitive Builder: takes the cluster information generated by the cluster merger and generates the primitive trigger word; There are two modes of primitive generation: debug mode and trigger primitive mode (see figure 4.6).

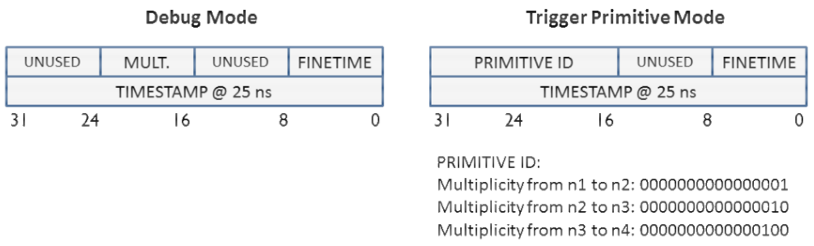


**Figure 4.6:** *SL Rich trigger firmware: primitive generation modes.*

In debug mode the cluster information are copied in the output word; in this way it is possible to read the results of the clustering algorithm and, by knowing the input words, understand if it worked. In primitive trigger mode, a trigger word is generated in the format compliant to the L0TP. In this mode four registers in the ECS contain four multiplicity thresholds (n 1 , n 2 , n 3 , n 4), these thresholds are used to encode the multiplicity information in the primitive ID;

## 4.4 The NEW-RICH firmware

For the 2016 data taking a new version of firmware for the RICH trigger was developed. The aim of the NEW RICH L0 primitive-generating firmware is to produce clusters of groups of hits belonging to the same Cherenkov circle. However, no spatial information is used and only time clusters are produced. This makes this firmware very simple and at the same time very general, making it suitable for more detectors, if necessary. In the PP FPGA, a preliminary clustering is performed and, in the SL, clusters coming from the 4 PPs are merged together. In the final stage of the SL, the average time of the clusters is computed and they are used to produce primitives to be sent to the L0TP (see figure 4.7).

In case of a multi TEL62 setup, that foresees the use of InterTEL boards, only the last TEL62 sends primitives to L0TP, while the others just send cluster data from one to another in a daisy-chain fashion. Time clusters are made of 32-bit data word coded in a common data format called RICH format (see figure 4.8).

In the RICH format, there are two types of words identified by the word ID (bit 31, 30, 15,14):

- Timestamp, identified by bits 0 0 0 1;

- Data word, identified by bits 1 0 1 1

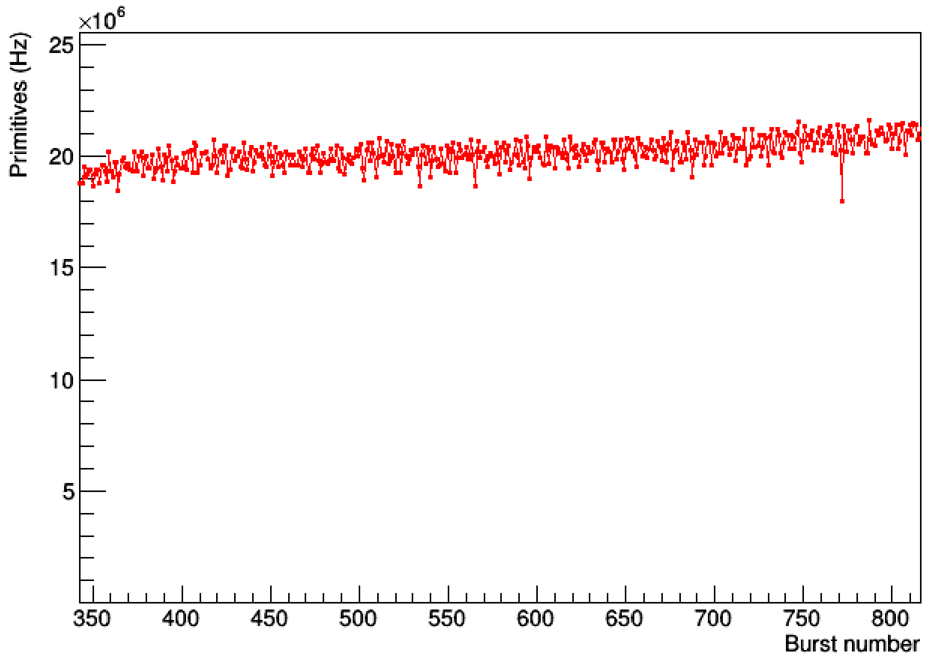
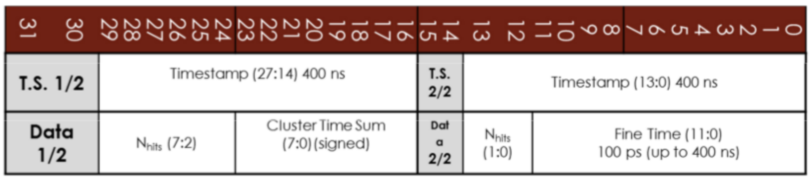


**Figure 4.7:** *NA62 2016 Data Taking: NewRich primitives transmitted to L0TP as a function of the number of the burst received.*



| 31 30 | 29 28 27 26 25 24 | 23 22 21 20 19 18 17 16 | 15 14 | 13 12 | 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|
| **T.S. 1/2** | Timestamp (27:14) 400 ns | | **T.S. 2/2** | Timestamp (13:0) 400 ns | |
| **Data 1/2** | $N_{hits}$ (7:2) | Cluster Time Sum (7:0) (signed) | **Data 2/2** | $N_{hits}$ (1:0) | Fine Time (11:0) 100 ps (up to 400 ns) |

**Figure 4.8:** *RICH format.*

The timestamp is a 28-bit word with 400 ns of resolution. It is sent only if needed, i.e. there will never be a timestamp that is not followed by a data word. Every data word - identifying a time cluster - contains:

- The cluster-seed fine Time (12 bit, 100 ps resolution, up to 400 ns ) called T;
- The cluster hit-multiplicity (8 bit, from 1 to 255 hits), i.e. the Number of hits belonging to that cluster, called N;
- The so-called Sum (8 bit signed) which is the sum of the differences between the cluster-seed time and the hits belonging to the cluster, called S;

Being S signed, on average, its value should remain small even if the cluster is made of a significant number of hits. Every module implemented in the RICH firmware must be able to accept the RICH format as input and output. In this way, modules can be moved around with ease. The RICH data-format gives the possibility to split 32-bit data-words into two 16-bit words to be sent through the InterTEL [1] bus. For this reason, the RICH format uses the peculiar double word ID separated between bits 31,30 and 15,14. A special cluster with N=0, S=0, and T=0xfff is created when the end-of-frame word (starting with 0xb) is received from the TDCs. This is done to ensure a proper PP-SL communication without any timeout mechanism, as explained later on. This word is also referred to as end-of-frame word (or EOF).

### 4.4.1 PP and SL firmware

The new RICH firmware exploits the recent (2016 run) sorting of input data. TDC data are now provided to the primitive-generating firmware in sorted frames of 25 ns. These frames are not separated by special words, on the contrary TDC words simply come partially sorted up the 25 ns bit (9 $^th$ bit). A scheme of the PP and one of the SL firmware is shown in figure 4.9 and figure 4.10 respectively. Internal memories are represented in purple, custom RICH blocks in yellow and ECS registers/FIFOs in red.

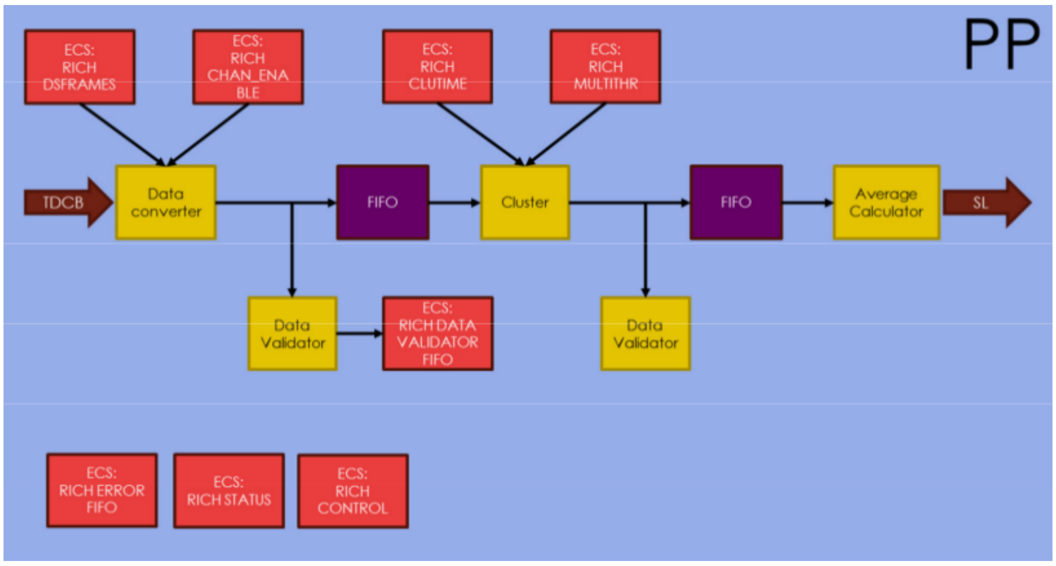


**Figure 4.9:** *NEW RICH PP firmware.*

#### Data Converter

The PP Data Converter (DC) reads TDC data (sorted in frames of 25 ns) from a TDCB fifo and converts them into RICH format. In order to do that it produces absolute time words (40-bit words) merging TDC timestamps and fine times and stores them into an internal FIFO. The FULL signal of this FIFO is connected to bit 8 of the error word, in order to detect a potential error in the data-flow. From these data, the DC produces clusters made by one hit, i.e. with N=1, S=0, and T=TDC time. When it is needed, it produces the 400-ns timestamp word. When the DC creates a timestamp, it loses one clock cycle, otherwise it reads and writes one word per clock cycle. Within a 6.4 $\mu$ s frame there might be at most 16 timestamps so this module is designed to handle a maximum of $1024\breve{}16 = 1008$ words per 6.4 $\mu$ s frame. When the End Of Frame (EOF)

[1]The InterTEL bus is a bus that is able to connect the SL FPGAs of many TEL62s in a daisy-chain fashion. It works at a frequency of 40 MHz with data words of 16 bits. It has never been used so far (April 2016).

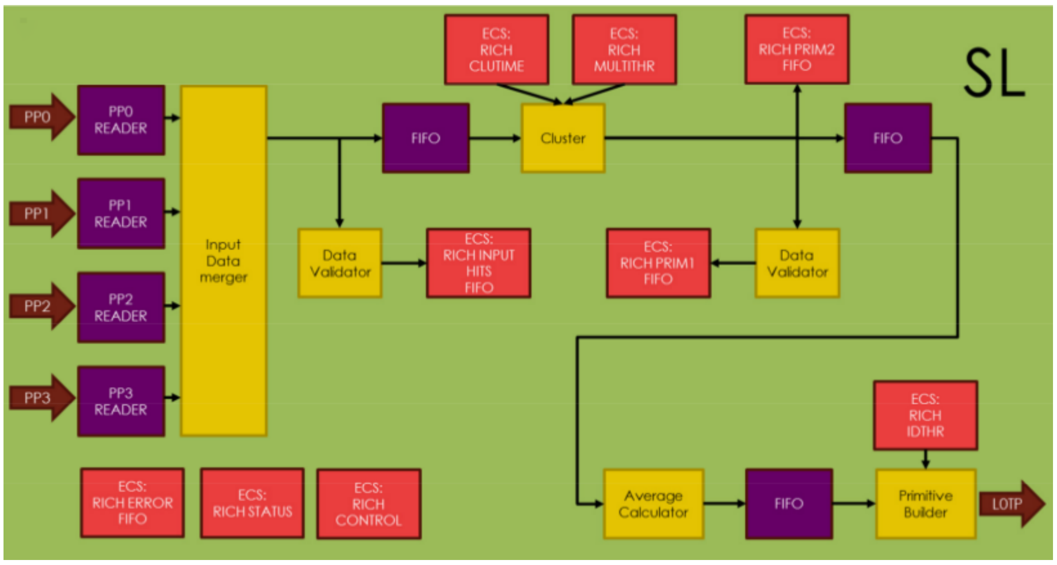


**Figure 4.10:** *NEW RICH SL firmware.*

word arrives from the TDCB board, the DC produces a timestamp and an EOF with the highest cluster seed.

#### Data Validator

The Data Validator module (DV) parasitically validates the data flow between two modules. Thanks to the standard RICH format, it can be plugged anywhere inside the firmware. The performed checks are the following:

- Consistency of the input write-request: the write request should always be '1' or '0'. The module raises an error flag when the write request has other values, e.g. high impedance (this statement it is only true in reference to simulation of the module);
- Consistency of the format: checks if every input word is either a timestamp or a cluster. If not, an error flag is raised;
- Consistency of the data flow: checks if after a timestamp there is always at least one cluster. Raises an error flag when it detects two consecutive timestamps;
- Time of the timestamps always increasing: the 400 ns timestamps must have an increasing time; if the time of a timestamp is equal or lower than the previous timestamp, an error flag is raised;
- Time seed of the clusters always increasing: clusters must have time seed that is equal or greater than the previous cluster. If not, an error flag is raised;

In both the PP and the SL FPGAs, there are two validators: one in input and one after the clustering module. For the PP, the input validator is right downstream of the converter, where the RICH format is used for the first time. It checks whether the data converter is working properly and if the input data are sorted at 25 ns. In the SL, the validator is connected between the data merger and the clustering module, checking the output of the PPs and the proper behavior of the data merger. The output of the DV is also connected to a FIFO readable from ECS. In addition to the error flags, the output word contains also some other information, depending on the error type. The error flags are connected to the same bits as the error word, both in the PP and in the SL FPGAs. In detail, they are:

- Consistency of the input write-request: bit 20 for the cluster validators and bit 15 for the input validators. The error code for the output word is “01000” and all the other bits are 0;
- Consistency of the format: bit 21 for the cluster validators and bit 16 for the input validators. The error code for the output word is “10000” and the other bits report the 27 MSBs of the input word;
- Consistency of the data flow: bit 19 for the cluster validators and bit 14 for the input validators. The error code for the output word is “00100” and the other bits report the 27 LSBs of the input word. This error could happen together with the next one;
- Time of the timestamps always increasing: bit 18 for the cluster validators and bit 13 for the input validators. The error code for the output word is “00010” and the other bits report the 27 LSBs of the input word. This error could happen together with the previous one;
- Time seed of the clusters always increasing: bit 17 for the cluster validators and bit 12 for the input validators. The error code for the output word is “00001” and the other bits report the 23 LSBs of the timestamp and the 4 MSBs of the cluster (in order to know its 25-ns time);

#### Data Merger

The Data Merger’s (DM) goal is to read the 4 input FIFOs of the SL and to merge them into one single FIFO preserving the order at 25 ns level. This task, that may seems apparently trivial, is fairly complex if one considers that the maximum throughput is needed. The DM, indeed, reads and writes one word per clock cycle [2] . It is purely combinatorial in order to be fast enough and waste zero clock cycles. It does not replicate timestamps or end-of-frames words. The data coming out of the module are sorted at the 25 ns level. This is done in order to perform smaller bit comparisons and to reduce path length in the FPGA. The merger is internally composed of sub modules performing binary comparisons, as shown in figure 4.11. The DM starts to operate as soon as both its input FIFOs (IB) are not empty. This cannot be avoided because, being the DM purely combinatorial, it does not have memory of data read in the past. If we imagine that a PP receives a lower rate than the others do, the proper behavior of the DM would be compromised, as one of the two FIFOs would get empty preventing the other to be read out. In the worst-case scenario, the IB belonging to the higher rate PP would get full while the other is still empty. In order to avoid this, as explained before, some dummy data words called end-of-frame words are produced by the DC every 6.4

[2] Actually the reading operation may be even faster. When two timestamps or two end of frames are read, only one is written out so that the throughput is 0.5 in those cases

us, i.e. when an end-of-frame word is received by the TDC. In this way the IBs are never empty for more than 6.4 us. The end-of-frame words contain T=0xfff, i.e. bigger than any other T value that might be present in the other FIFO, making the DM reading the other FIFO first, until the end of the frame is reached.

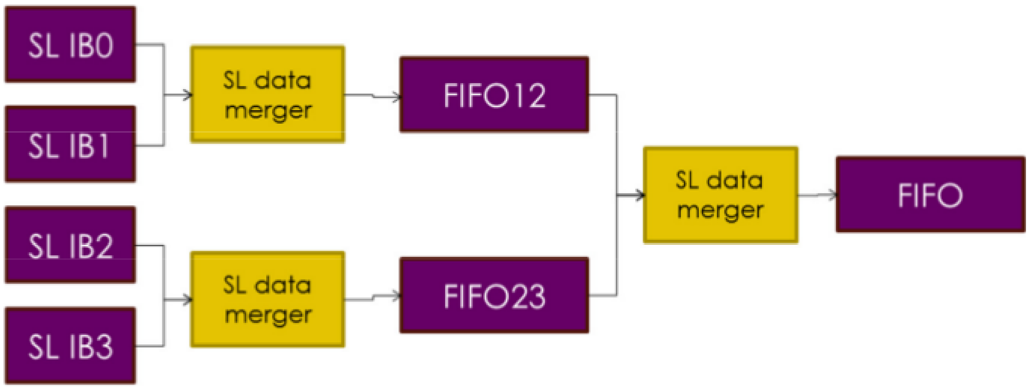


**Figure 4.11:** *Data Merger block diagram.*

### Average Calculator

The Average Calculator (AC), receives and sends clusters in RICH format. It transforms the RICH cluster in the following way:

$$\text{Tnew} = \text{Told} + \frac{Sold}{Nold}$$

$$\text{Nnew} = \text{Nold}$$

$$\text{Snew} = 0$$

By doing this, the new cluster is fully compatible with other clusters (even not averaged),so that one may in principle perform clustering on averaged and non-averaged clusters together, without any complications. The AC is used both in the final part of the PP and in the final part of the SL, before the primitive builder. The AC in the PP is not strictly needed, but it is useful to avoid the overflow in the S register of the clusters, that may occur when two physical adjacent clusters are merged together.

### Primitive Builder

The Primitive Builder (PB) builds the trigger primitives in the standard NA62 MTP format. Primitive ID depends upon multiplicity with thresholds programmable in the same register. Incoming clusters are written into an internal FIFO, containing their 40-bit time and multiplicity. This FIFO could go to the full state, and when this happens, bit 10 of the error word is asserted. As the MTP format requires three words for one primitive, a good fraction of throughput is lost in this stage. Nevertheless, the 66 per cent of 160 MHz is still a big rate with respect to the gigabit Ethernet limit. The module performs some checks to guarantee its proper behavior. At first, checks whether the input data has a valid ID (cluster, timestamp) or not. If not, asserts the bit 11 of the error word of the SL. Input data should not have multiplicity zero, due to the clustering module; if it happens PB asserts bit 9 of the error word.

**Clustering Module**

The RICH Clustering Module (CM) is the most important block of this firmware. It performs cluster of clusters if they are closer than a certain value. The comparison is performed only on the T value (the time seed) while the sum is completely disregarded. During the merging operation of the clusters N and S are also taken into account. The CM is divided in sub-blocks as shown in figure 4.12.

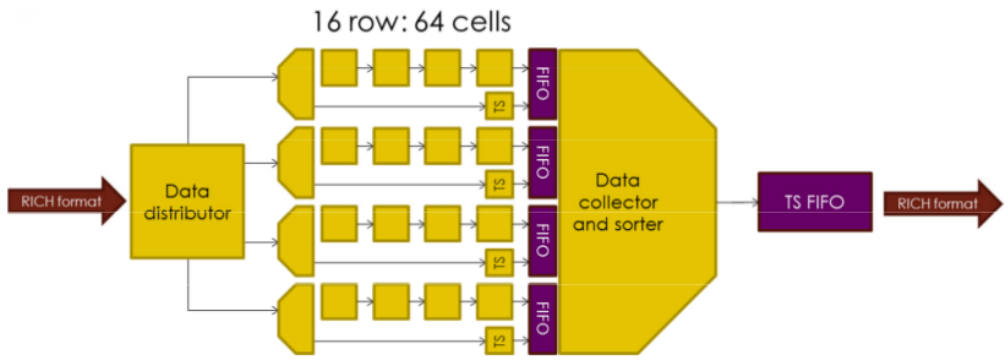


**Figure 4.12:** *Clustering Module block diagram.*

The Data Distributor (DD) handles incoming data (in RICH format) and feeds them into the clustering rows. The clustering rows are 16 rows, each one made of 4 cells and taking in input hits belonging to a specific 25 ns frame. The DD rearranges data into 8 bit 100 ps fine time and 32-bit 25 ns timestamp, which are sent to the time stamp register rather than to the clustering row. In this way, all the operations inside the row are done on 9 bits only (8 bit plus one to handle adjacent time frames, as we will see right in the figure 4.13). In order to handle border effects, the DD must send the hits belonging to clusters split into two adjacent 25-ns frames to the proper row.

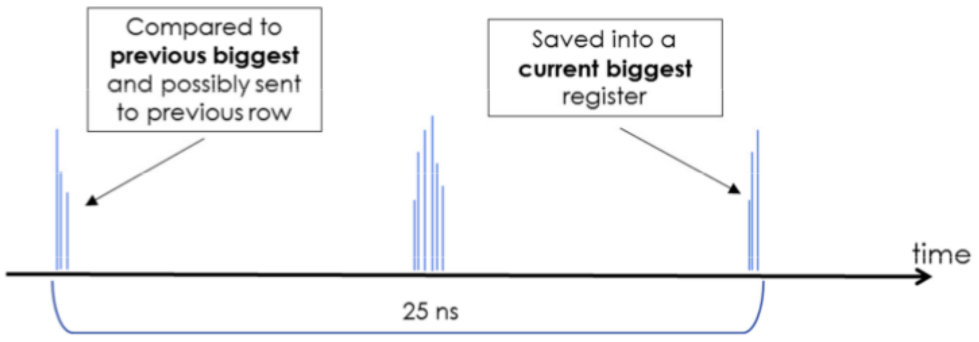

Compared to previous biggest and possibly sent to previous row
Saved into a current biggest register
time
25 ns

**Figure 4.13:** *Data Distributor graphical view.*

To this aim, for each 25-ns frame, the DD stores the value of the greatest time received into a register (current biggest). When it switches to the next row, (only if the two rows are handling frames that are adjacent in time) it compares every time to the previous biggest register. When a time hit is close enough to the previous biggest, with respect to the clustering time, the DD sends it to the previous row instead of to the current. The figure 4.13 and figure 4.14 explain this method with a graphical

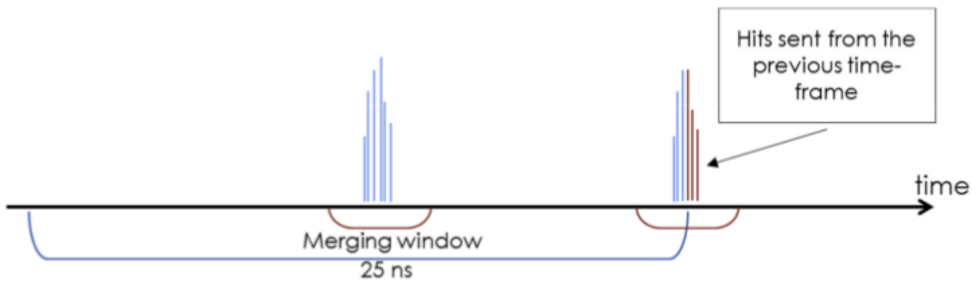


**Figure 4.14:** *Data Distributor graphical view: merger window.*

representation. DD checks if the input words are either a timestamp or a cluster; if they're not one of them, bit 1 of error word is asserted. The core part of the CM is the clustering cell. The comparison of the Ts and the merging of the clusters are performed by the cell. Each cell stores the first T (9 bit) received and forms a cluster that will become the new cluster seed [3] . If the T value of the next clusters received matches the stored T, (within a programmable time window) the cell stores the cluster values for merging. When merging clusters, the cell changes the stored values using the following rules:

S += N new * (T new – T) + S new;

N += N new;

If the incoming T does not match the stored value, the cluster is sent to the next cell that performs the same operation. When a time is not accepted for merging and it is sent to the next cell, if T is bigger than the stored value, an internal position register is increased. In this way, in the end every cell in the row will "know" its position with respect to the others. By construction, the clustering module cannot handle more than 4 clusters per 25-ns frame. All the clusters after the fourth are discarded, while the bit 7 of the error word is asserted. Each cell checks the overflows of the result of the multiplication and of the global correction that has to be applied to the cluster seed. Also N and S fields overflow are checked. Every time an overflow occurs, the corresponding bit is asserted; the bits are, respectively, number 5, 4, 3 and 2 of the error words. When the n$^{th}$ row is filled, the (n-2)$^{th}$ is ready to be emptied. This is done by "flushing" the row. When the flush-mode is enabled, the cells act as a shift register, giving as output all the cluster times and number of events per cluster. The cluster data, plus the position (going from 0 to 3), is now sent into the FIFO at the output of the row. The admitted values of position are 0 to 3, because there are only four words in each row. If the value is greater, bit 6 of the error words (PP and SL) is raised. In order to perform the multiplication, every cell is divided into two independent blocks, connected by a FIFO:

- The matching block, handling the comparison between stored and input time;
- The merger block, handling the calculation involved in the merging, containing a multiplier;

[3]The seed value does not get updated during the clustering operation. It gets updated only in the average calculator. This is due to the fact that a dynamic update of the seed would result in big cost in term of complex operations like multiplications and divisions.

The internal data format is made of:

- Time (8+1 bit): the 9 th bit is used to handle time coming from the following frame;
- Position (3 bit);
- N (8 bit);
- S (8 bit, signed);

The throughput of the clustering module (like any other module in the RICH firmware) is kept at 1 clock cycle. For this reason, 16 clustering rows are needed to take into account the filling time of a row. The emptying operations on a row should be finished before the row needs to be written again. Moreover, two rows are filled “at the same time” to take care of border effect, as previously explained. Once the second row is completed, the first can be read out. 16 rows are enough to compensate for the latency of the cell being given by: (2d + m) where d is the depth of the row and m is the latency of the multiplier. In our case (2*4 + 3 = 11 < 16), so the there will always be a free row to fill. Finally, the data collector (DC) retrieves data from the row FIFOs and sorts them according to the position register. The DC module is divided into three independent parts:

- The first part reads data from the FIFOs, sorting them, and writes the output in absolute time (40 bits) into an internal FIFO FULL signal of this FIFO is connected to the bit 0 of the error word, in order to detect a potential error in the data-flow;
- The second part discards the clusters that have multiplicity outside a predefined range, settable by a dedicated register for both PP and SL. If the multiplicity of the cluster is outside this range, the cluster is discarded, while if it is in the range the whole cluster is written to another internal FIFO;
- The last part of the DC reads the absolute data from the FIFO and formats the output in the RICH format;

## 4.5 L1/L2 Software Trigger

After a positive L0 trigger, all sub-detectors data are moved to PCs for processing. If the L1 trigger condition are fulfilled, each sub-detector sends a L1 trigger primitive to the L1 Trigger Processor PC. The L1 Trigger Processor will match these primitive and issue a L1 decision, at which time the data will be further processed(in case of a positive L1) or discarded (in the case of a negative L1 verdict).

By correlating information between different sub-detectors, the events will be partially reconstructed and made available for the L2 trigger decision. All data satisfying the L2 trigger condition will be saved to disk. While In case L2 conditions are not satisfied, the data will be deleted.

CHAPTER 5

# GPU

## 5.1 Introduction

In recent years the use of commercial graphics cards "Graphic Processing Units" (GPUs) for scientific computing has grown considerably. Although GPUs have been designed for three-dimensional graphics and are produced mainly for the video game market, their parallel structure for data processing and their computing power can be used in several applications in the scientific field (for example in medical physics, astrophysics, quantum mechanics, molecular chemistry, etc. [1]). For these reasons a branch of computing research called General-Purpose Computing on Graphics Processing Units (GPGPU) was developed: it has the goal of using GPUs for computations that require a lot of processing power. GPUs have a different structure compared to the CPU (Central Processing Unit which is the processor of standard computers) with more chip area devoted to the computing unit with respect to flow control or caching, making the device more appropriate to highly parallelizable tasks (see Figure 5.1).

From a comparison between the GPU and CPU (see Figure 5.2) we note that the computing performance of GPUs exceeds by a factor 6 that of CPUs at present.

GPUs could be used to fill the lack of commercial processing devices on which the lowest level trigger of a high-energy experiment can be implemented and to reduce the size of PC farm used for the high level triggers [17].

## 5.2 GPUs in high energy physics

In the past decades High-Energy Physics experiments exploited custom-built processors with the most recent technology for the trigger and data acquisition systems. Lately, the large distribution of mass-market electronic devices changed this trend,

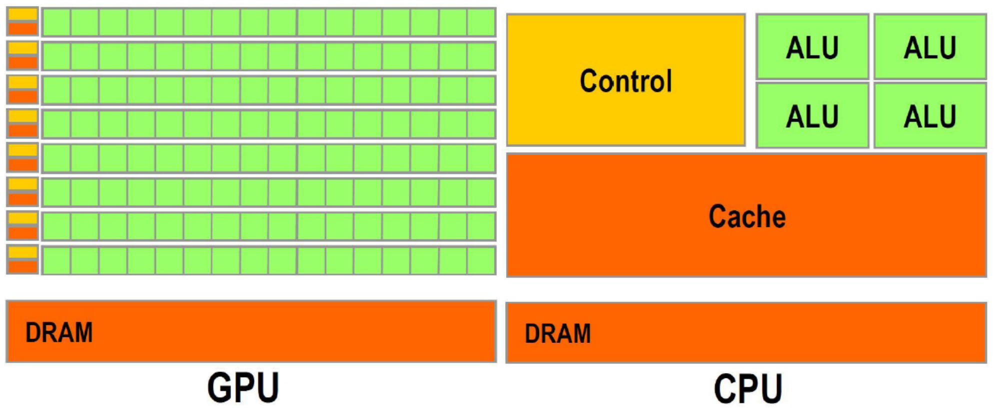


**Figure 5.1:** *GPU and CPU architectures showing the ammount of chip area devoted to the different parts. The largest area (green) GPU chip is devoted to ALU (Arithmetic and Logic Unit) circuits that are fundamental block for computing operation; Flow control (yellow) and caching (red) units occupy a little fraction of the chip area.*

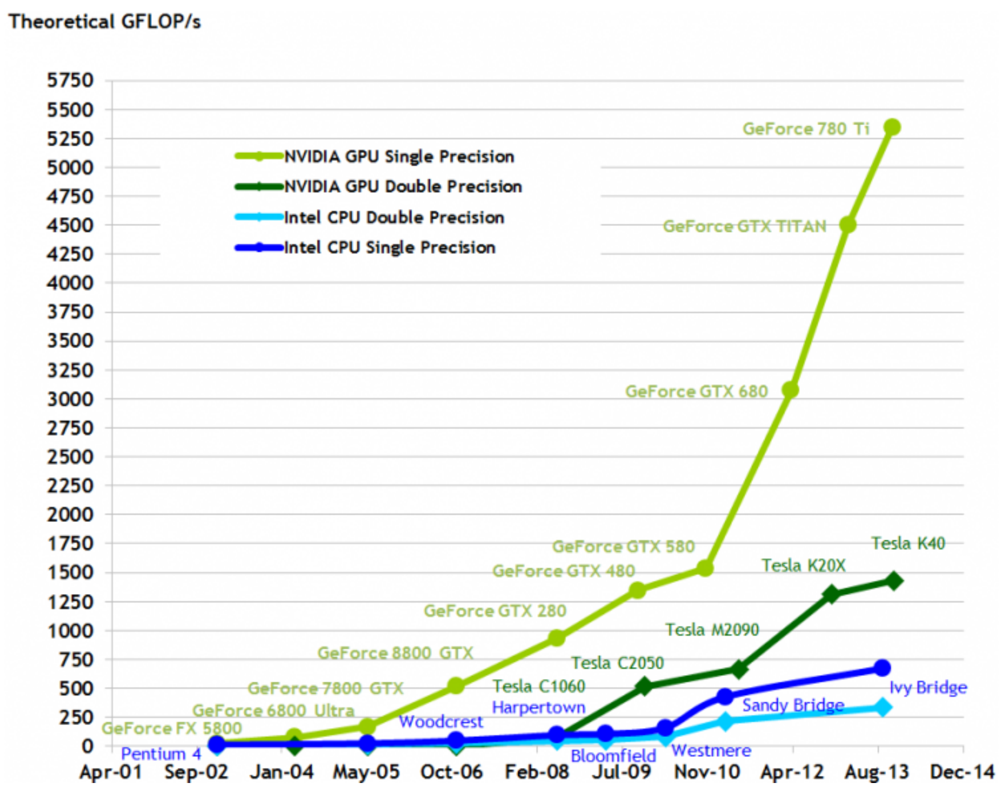


**Figure 5.2:** *Time evolution of floating-Point Operations per Second for CPUs and GPUs in recent years.*

since commercial hardware manufacturers now lead the development of the most performing digital computing devices. Moreover, the development of these perform devices with the latest silicon technology involves costs usually not affordable for research groups.

These facts pushed the scientific community to evaluate commercial solutions instead of custom electronic systems for a growing part of TDAQ system. The obvious advantages of such trend lie in an optimization of the costs, installation and maintenance issues, as well as the possibility of easy upgrade, since more powerful devices become available every year at the same (or lower) price.

The only exception is given by very specific front-end electronics. Indeed until now it was not possible to implement the entire trigger and data acquisition system on a commercial processor for a *triggerless* approach, because the computer farm required would have an impractically large size.

GPUs can fill the missing gap between custom front-end electronics and commercial devices. They can be used to implement low level triggers or to reduce the size and the cost of computer farms used for running high level triggers.

The main problem with this approach is that GPUs are not designed for low latency response, since their target applications have only to deal with video frames at rates usually below a hundred Hz. The first level trigger of a HEP experiment must instead handle event rates even higher than 10 MHz in a maximum latency that is defined by the size of the buffer memories where data are stored. In recent HEP experiments this latency is of the order of $1 \div 10\,\mu$s even if there are some experiments, like NA62, with latencies that reach 1 ms.

Every year the computing power of GPUs increases so fast that their intrinsic time latencies approach the requirements of HEP lowest level triggers. Indeed in the last years the interest of the High-Energy Physics community for these devices grew considerably [18].

The LHC luminosity upgrade foreseen for the next years could profit from GPU developments; the improvement of highly selective algorithms will be essential to obtain a sustainable trigger rate. To exploit these more complex algorithms a large computing performance is needed, which can't be any more obtained by increasing of CPU clock frequency. Indeed until few years ago the CPUs performance improved mostly with the increases of the clock frequency. Now the CPU clock frequency can't be increased due to physical constraints (the CPU power consumption increase with the CPU clock frequency) and increases in computing capabilities can be obtained only by using more cores and processors together. Several experiments are studying algorithms and developing the environment to use GPUs in their high-level trigger.

- The **ATLAS** experiment is studying a GPU implementation for the muon selection [39] and the track reconstruction for its Inner Detector [19] [40]. The muon trigger algorithm is at the moment implemented as a simplified version in the ATLAS level 2 trigger system and it is based on the repeated execution of the same particle trajectory reconstruction. The high computing capability of GPUs will allow to use a more refined muon algorithm for better selection and efficiency. Concerning the track reconstruction algorithm the parallelization of the data preparation step on a GPU reached a speed-up of up to 26x over the serial CPU version and the implementation of a Reference Kalman-Filter on the GPU achieved a speed-up of 16x compared to a single threaded CPU version [40].

- **CMB** (Compressed Baryonic Matter), a future heavy-ion experiment, is studying the implementation of its online First Level Event Selection on a dedicated many-core CPU/GPU cluster. This system should be able to elaborate a huge data quantity of up to 1 TB/s [**?**].

- The **LHCB** experiment in its upgrade phase is developing a trigger-less data acquisition system to read-out all the detector at the bunch-crossing rate of 40 MHz. The events that LHCB stores have a small size (order of 100 kB) so it is relatively easy to use GPUs or multi-core CPUs to process many events in parallel for a real time selection without lose time in the bottleneck due to the transfer speed to the GPUs. Moreover this possible solution might reduce the cost of the High Level Trigger (HLT) Farm. Since the vertex finding and the track reconstruction algorithms are the more time consuming threads running in the HLT, they will be probably the first to be implemented on GPUs. Preliminary results show a speed-up of a factor 3 with respect to CPUs obtained using the GPU for the tracking algorithm respect to the sequential one [45].

- The **CMS** experiment is studying the benefits of using GPUs to reconstruct high energy tracks in the core of high $\mathrm{P_T}$ jets coming from the heavy quarks. Indeed the tracks from B-decays, ad high transverse momentum, become more collimated, reducing the efficiency of the standard track-finding algorithms. The combinatorial complexity of such algorithms could profit from an implementation on GPUs [18].

- The **PANDA** hadronic physics experiment, under construction at FAIR (the Facility for Antiproton and Ion Research in Darmstadt) will not use hardware-based triggering but sophisticated software-based online event triggers. They are investigating three different GPU-based algorithms for the track reconstruction [34]: the Hough transform (a method that allows to detect edges in images), a Riemann Track Finder (which uses the projection of two dimensional hit points onto a Riemann surface) and a Triplet Finder (an algorithm specifically designed for the PANDA straw Tube Tracker that analyses only small subsets of data at time). Using the Triplet Finder algorithm, a cluster of O(100) GPUs seems sufficient for the PANDA trigger system to cope with the expected rates.

## 5.3 NA62 GPU trigger

In a standard trigger system for a high energy physics experiment, the complexity in primitive generation and trigger decision is limited by the time available as defined by latency requirements. Usually in trigger levels with fixed small latency, the trigger primitives are quantities related to multiplicity and hit patterns. The trigger decision is defined with rough conditions, not allowing high rejection factors and selection power.

In many cases the definition of trigger primitives can be reduced to pattern recognition issues. This is the case for charged particle track identification in magnetic spectrometers, trajectories in silicon strip trackers or photon rings in Čerenkov detectors. The RICH detector in the NA62 experiment falls into this last category.

A project is being developed within the NA62 collaboration, which aims to integrate GPUs into the lowest-level trigger for the first time in High Energy Physics [26]. The use of GPUs in such a hard real-time system has not been attempted so far, but it looks like a realistic and challenging possibility. The online use of GPUs would allow the computation of complex trigger primitives at the L0 trigger level, with resolution comparable to offline analysis. NA62 is considered a test bench for the use of GPUs in lowest level trigger. In this work we aim at demonstrating that GPUs can be usefully employed in a low level trigger more than to prove that the computing power available in the present generation of video cards is enough for the NA62 needs. There are two different way to insert GPUs in the NA62 context.

In the first option - the more challenging - the GPUs perform two different works Fig 5.3:

- compute the data received from the TEL62 boards ( see sec. 3.6) of all the detectors contributing high quality primitives to the L0 decision (rings, clusters, tracks)
- substitute the L0TP (see Sec. 3.2), matching the primitives create in the previous first step and issue a trigger decision

Since neither the Host CPU nor the GPU have the necessary precision of 25 ns, for sending the trigger synchronously with the clock experiment an FPGA is needed.

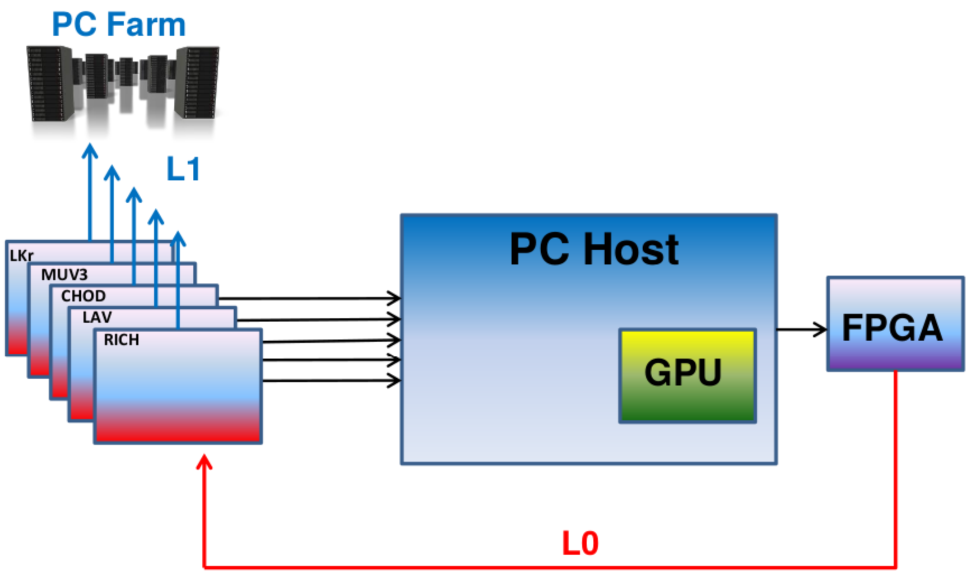


**Figure 5.3:** *The GPU is located inside a Host PC, and receives the primitives from all the detectors participating the L0, to issue a trigger decision, and sends it to the TEL62. According to the trigger decision data will be discarded or sent to L1 PC farm.*

The second option, is illustrated in Fig. 5.4. The GPUs are inserted between the TEL62 boards and the L0TP. In this scenario the work of GPUs is simpler with respect to the one described above. The GPUs receives only the data from the RICH detector, process it and then sends a primitive compatible with the L0TP format, after this the L0TP issues a decision and sends it to the TEL62 boards.

In both cases described above the time passed from when a primitives is sent by the TEL62 to a trigger decision is returned to the boards should be less than the maximum latency of 1 ms, so the computation time performed by the GPUs should be less than 1ms (more specifically should be below 206$\mu$s, see Sec. 5.4).

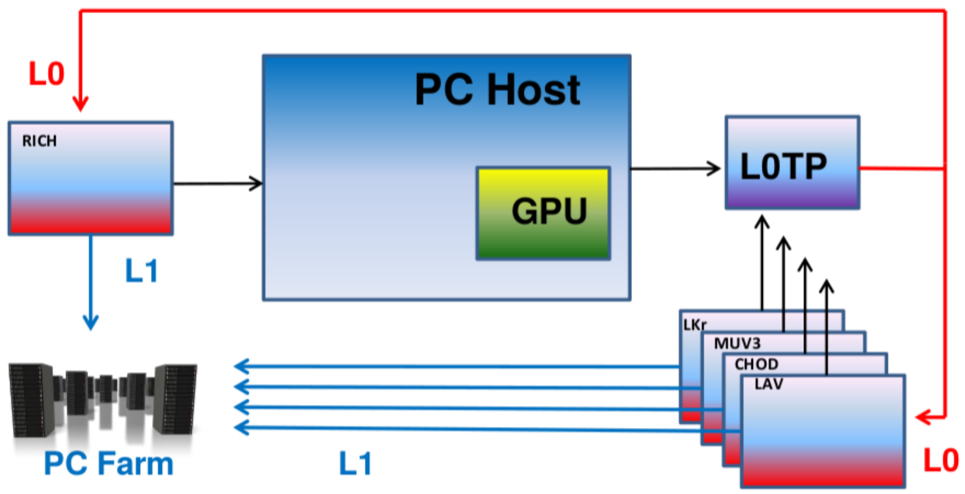


**Figure 5.4:** *The GPU is located inside a Host PC, and receives the data only from the RICH detector, and sends primitives to the L0TP, where will be matched with the ones from the others detectors for issue a L0 trigger decision. According to the trigger decision data will be discarded or sent to L1 PC farm.*

As a first implementation of a low-level L0 trigger on GPUs in NA62, we decided to focus on the fitting of rings on the RICH detector generated from charged particles crossing its volume.

This information can be employed at the trigger level to increase the purity and the rejection power for many triggers of interest. In the standard L0 trigger the RICH information is only used to generate a PMT hit multiplicity but this information is barely connected to the number of rings in the detector and is not very useful.

With a ring fitting one can have a better discrimination of between multi-track and single-track events; extracted parameters might be used in later software trigger levels to perform particle identification with spectrometer data. The input rate to the RICH trigger is expected to be $\sim 11$ MHz with an average hits multiplicity of $\sim 20$ hits for events. The amount of data to be processed is enormous, making this application a good test bench for a first level trigger based on GPUs.
In order to be used in a lowest-level trigger, a ring fitting algorithm needs to be:

**seedless** : it will be fed with raw RICH data with no previous information on the ring position from other detectors;

**fast** : it will run concurrently with the hardware L0 trigger, with a maximum latency of 1 ms (decision making time) and an input event rate about of 10 MHz;

## 5.4 Data input to GPUs

In order to use GPUs in the RICH L0 trigger two main aspects need to be defined:

**Input:** how the data from readout boards are sent to GPUs

**Data format:** how the data sent to GPUs are arranged

### 5.4.1 Input

The data from readout boards need to be transferred in the GPU memory. The copy process need to have a deterministic low latency: the contribution to the total latency has to be low enough in order to respect the requirement of latency $< 1\text{ms}$.

Data on RICH PMT hits are produced within the TEL62 boards (5.5) and they are made available to the GPU trigger system through standard 1Gb/s ethernet links. In the standard way, data are sent to a Network Interface Control (NIC) from the readout boards, then the NIC would copy the data via PCIExpress (PCIe) into the CPU memory and finally data would be copied in the GPU memory to be processed.

This solution has two major problems:

- the multiple copies to write data in GPU memory: $NIC \rightarrow CPU \rightarrow RAM \rightarrow GPU$;
- the non-deterministic latency time, due to CPU running concurrently many different processes.

One approach for addressing the first issue is to speed up the transfer by reducing the multiple copies using a non-standard driver software on the host, such as `PF_RING` [**?**]. The other approach, which addresses both issues , is to avoid the copy to host completely, and this is the one adopted in NA62.

### 5.4.2 NaNet and GPUs

In order to overcome the above limitations an approach was considered in which data are transferred directly to the GPU without action from the host. This is possible because NVIDIA GPUs implement P2P(Peer to Peer)/RDMA (Remote Direct Memory Access) protocol, this means GPUs connected via the same PCIe bus can access to each others' memories without involving the CPU (Fig. 5.5).

One implementation of this is a FPGA-based Network Interface Card with GPUDirect P2P/RDMA capabilities, named NaNet [2] developed by INFN within the project APEnet+. This essentially means that NaNet can copy data directly into GPU memory, because it's seen by the video card as another GPU device and can use the data sharing mechanism between GPUs.

Using this solution data are sent to NaNet, then NaNet copies data directly into GPU memory, without involving the CPU in the process. In this way data are transferred with a low and deterministic latency time as intended [**?**]. Figure 5.6 shows how, for a buffer size smaller than 8KB, the transfer time is below 100 $\mu$s. The 8KB was chosen as the maximum buffer size of the data transmitted by NaNet.

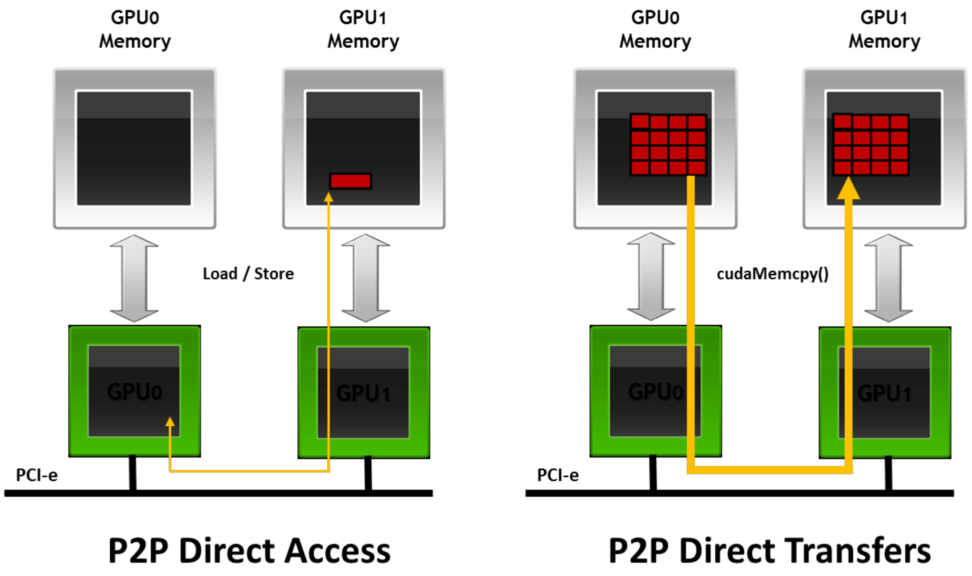


**Figure 5.5:** *NVIDIA GPUDirect Peer-to-Peer (P2P) Communication Between GPUs on the Same PCIe Bus.*

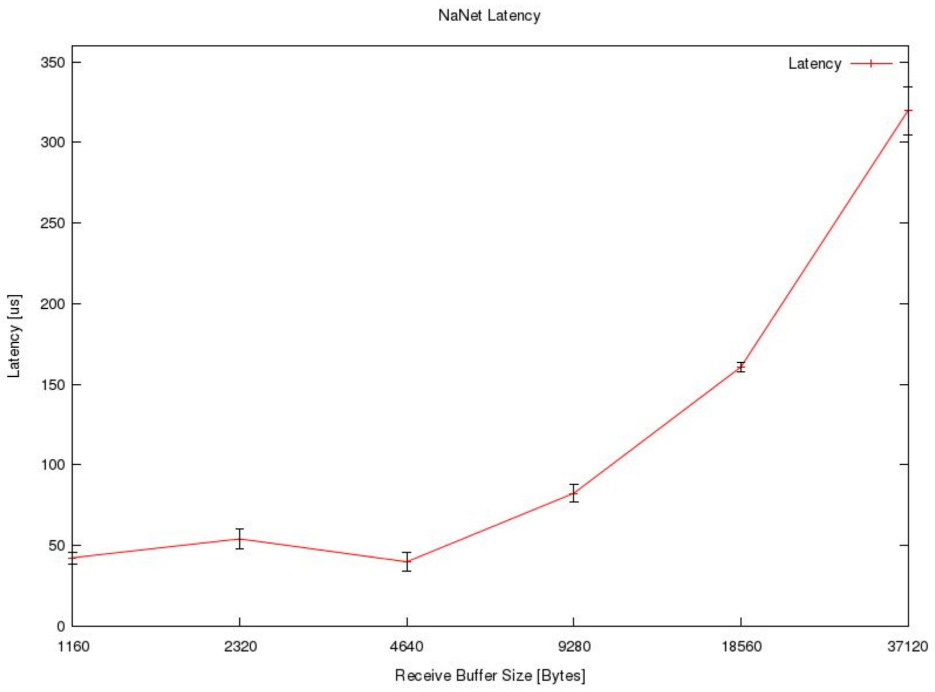


**Figure 5.6:** *The latency time to transfer data from NaNet to GPU memory for different buffer sizes.*

The difference in data copying between NaNet and a standard NIC is shown in Fig.5.7

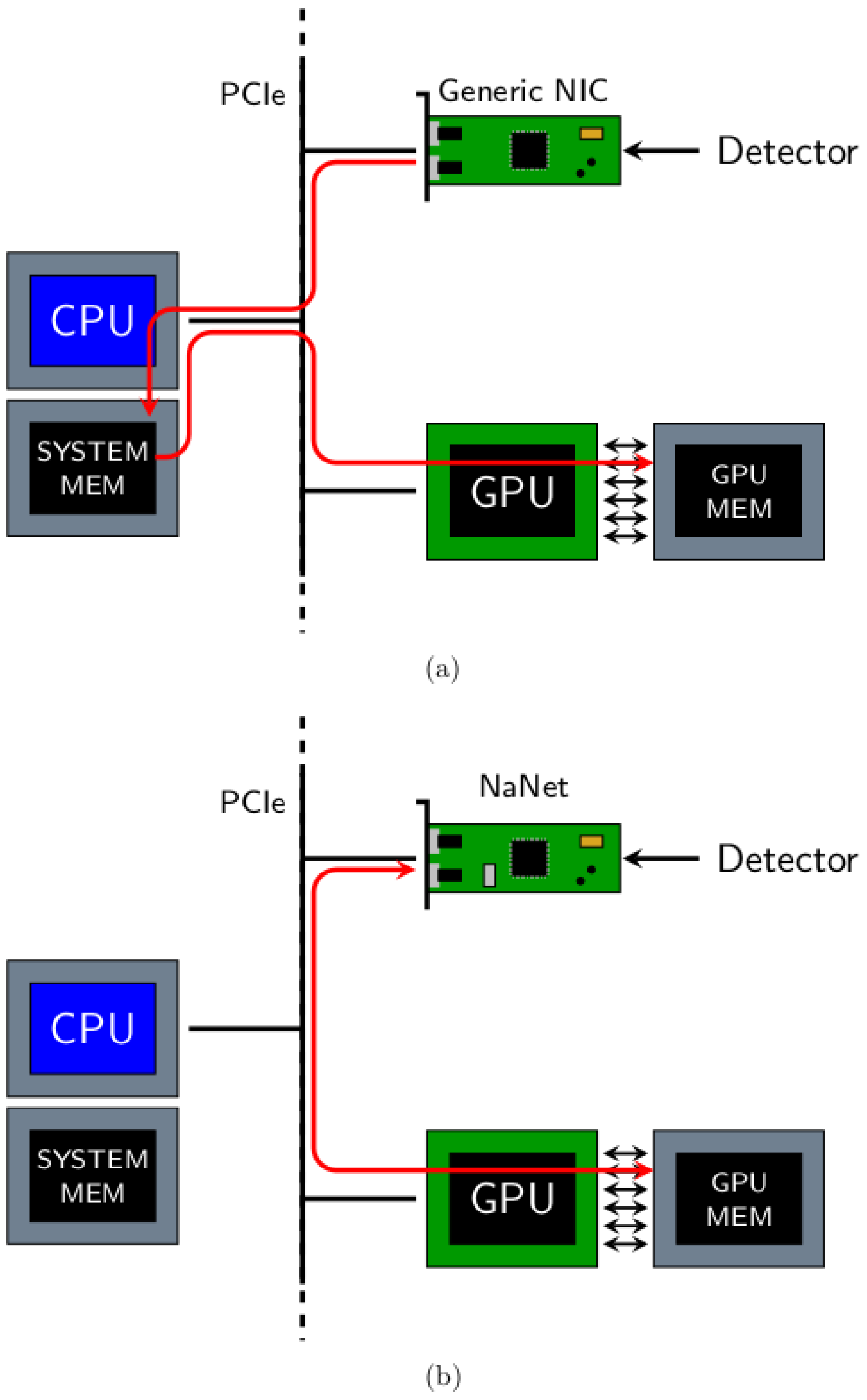


**Figure 5.7:** *Difference in data transfer between a generic NIC 4.3(a) and NaNet 4.3(b).*

## 5.5 GPU-RICH firmware

RICH data are read by on five different TEL62 boards (512 channel per board): the first four boards receive signals from single PMT; the fifth board receives data from the SuperCells (OR of 8 PMTs) and it is the one used for the standard L0 trigger based on FPGA.

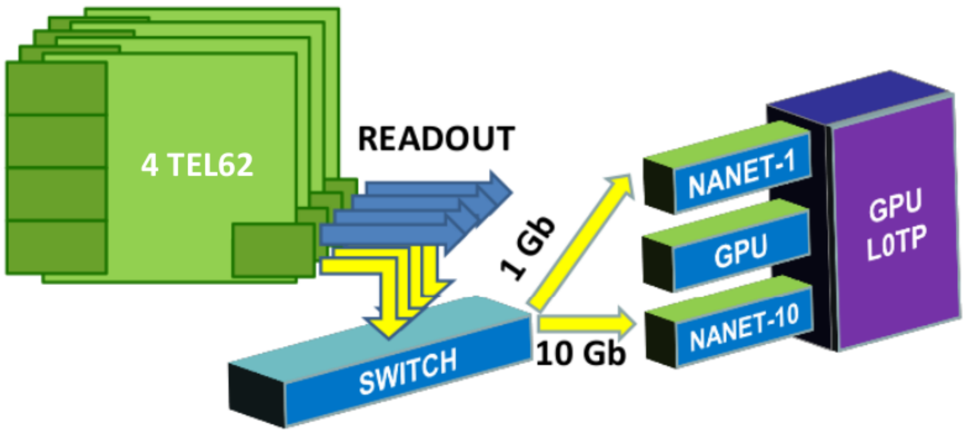


**Figure 5.8:** *Pictorial view of GPU-based Trigger.*

The ring fitting algorithm based on GPUs needs to use the data from the first four boards to have all the information on the individual channel hits (see Fig.5.8). The standard version of RICH firmware (4.3), the same used for RICH data acquisition, is used for this purposes. In the GPU-RICH version the information on hits that have contributed to the primitive is requested. In the PP firmware PP in addition to the transmission of the timestamp and fine-time it is also transmitted the multiplicity of hits and all channels used for the formation of the primitive. The channel information is compressed so a 32 bit word for 3 channels (9 bit each) can be used. The sub-clustering process is the same used in RICH standard and the cluster is formed inside the SL. In fact once data passed in the SL the primitive parts are joined together to form the final primitive. Furthermore in GPU-RICH the multiplicity of partial hits are summed to obtain the total and the channels are merged keeping the same formatting. The plots in Figures 5.9 and 5.10 are the result of the tests performed and indicate how all the produced primitives, are contained inside two frames and the time 0 does not frame dependent.

### 5.5.1 Data preparation for GPU

Each TEL62 board sends to NaNet the individual PMT hit data in a Multi GPU Packet (MGP) format shown in Fig 5.11. This format with PMT IDs coded with 9 bits has been chosen to optimize the bandwidth used by TEL62 boards. The various field of the MGP format are described below

**Source-ID** =0x1C, is the RICH detector identifier

**Source sub-ID** 0x0, 0x1, 0x2, 0x3, is the identifier of the RICH TEL62 board sending the data

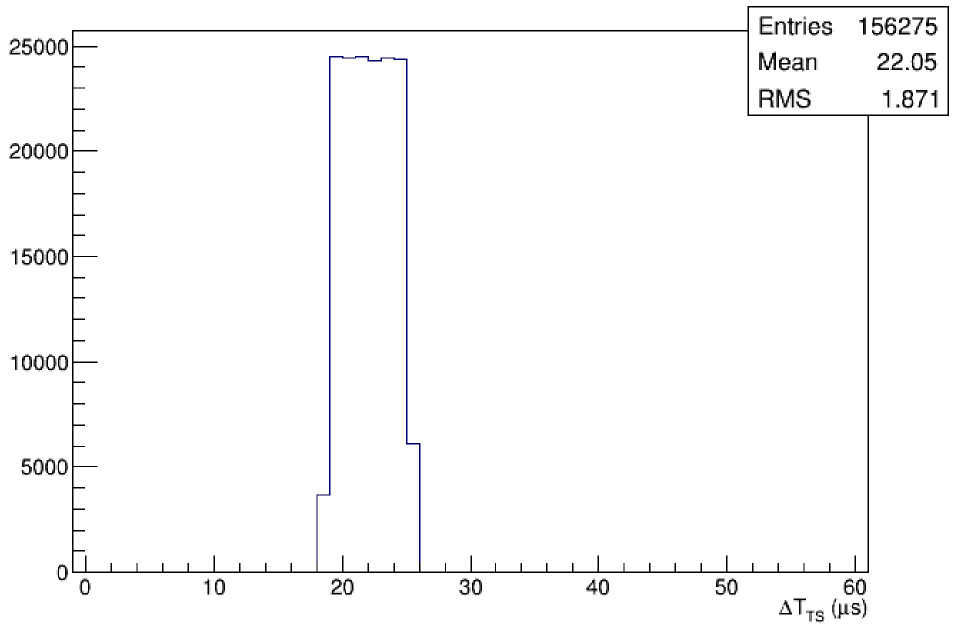


**Figure 5.9:** *difference between the TS of the primitive and the TS of when is was MGP product for a period of 6.4us.*

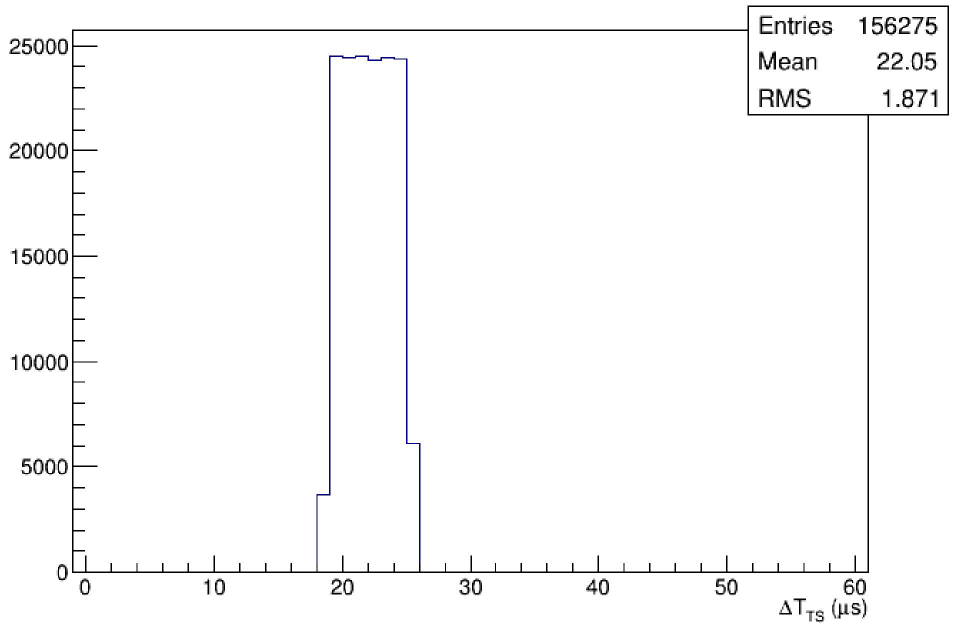


**Figure 5.10:** *difference between the TS of the primitive and the TS of when is was MGP product for a period of 12.8us.*

**Total number of hits** sum of all hits in the MGP(control purpose)

**Counter** progressive number of the MGP (4-bit, wrapping every 16 MGP)

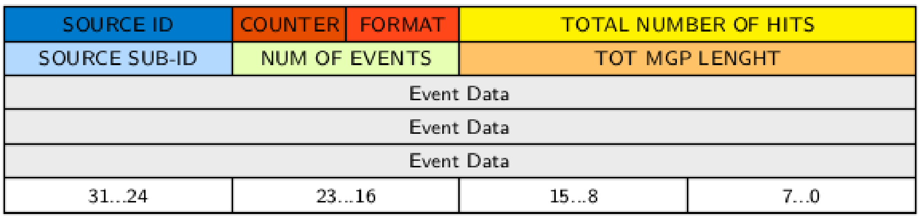
**Number of events** number of events in the MGP

**Event Timestamp** : timestamp of the event with 25 ns LSB

**Event Fine time** : fine time of the event with 100 ps LSB

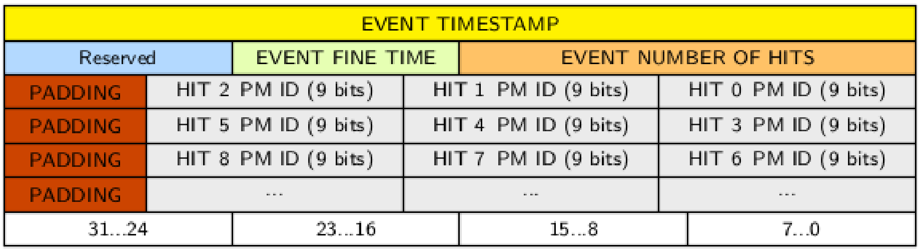
**Event Number of hits** : total number of hits in the event

**Hit ID** : PMT number (9 bit), the full identification number of the RICH PMT is obtained adding in front of the Hit ID the 7 LSB of the source SUB-ID field from the MGP header.



| | | | | |
|---|---|---|---|---|
| SOURCE ID | COUNTER | FORMAT | TOTAL NUMBER OF HITS | |
| SOURCE SUB-ID | NUM OF EVENTS | | TOT MGP LENGHT | |
| Event Data | | | | |
| Event Data | | | | |
| Event Data | | | | |
| 31...24 | 23...16 | | 15...8 | 7...0 |

(a) Header of the MGP.



| | | | | |
|---|---|---|---|---|
| EVENT TIMESTAMP | | | | |
| Reserved | | EVENT FINE TIME | EVENT NUMBER OF HITS | |
| PADDING | HIT 2 PM ID (9 bits) | | HIT 1 PM ID (9 bits) | HIT 0 PM ID (9 bits) |
| PADDING | HIT 5 PM ID (9 bits) | | HIT 4 PM ID (9 bits) | HIT 3 PM ID (9 bits) |
| PADDING | HIT 8 PM ID (9 bits) | | HIT 7 PM ID (9 bits) | HIT 6 PM ID (9 bits) |
| PADDING | ... | | ... | ... |
| 31...24 | | 23...16 | 15...8 | 7...0 |

(b) Event data format of MGP.

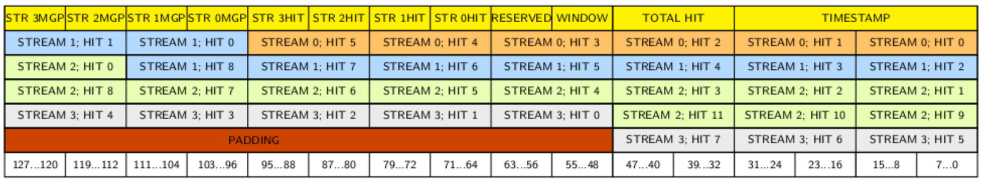
**Figure 5.11:** *The MGP format.*

The first NaNet preprocessing tasks is to merge the data received from the four RICH boards according to the time stamp to make it usable by GPU kernel. NaNet takes the first events sent by each TEL62 board and searches for the one with the smaller Timestamp + Finetime, then it opens a programmable time window around such time (5 Finetime units, $\sim$ 500 ps). All the events with a Timestamp + Finetime in the time window are merged in the same event, stored in a buffer called CLOP (Circular Lists Of Persistent receiving buffers) and sent to GPU for processing. The data are sent to GPU either after a certain programable timeout period (206$\mu$s at this time, the timeout start after the first MGP is arrived from the TEL62 boards) or when a buffer size of 8KB is reached. The NaNet timeout is also the maximum time available to a kernel for processing the events. If during the computation this limits it's exceed repeatedly, data would be overwritten while are read by the kernel, causing in most cases a crash. The only way to prevent the crash is keep the computing time of the kernel below 206$\mu$s.

The data format for each event is shown in Fig. 5.12

The data format called Merged Multi Event GPU Packet(M$^2$EGP) has 128 bit long header containing

- the TIMESTAMP (32bit) of the merged event corresponds to the 24 LSB bits of MGP Timestamp + Finetime of the event with smaller value. Only events with



| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| STR 3MGP | STR 2MGP | STR 1MGP | STR 0MGP | STR 3HIT | STR 2HIT | STR 1HIT | STR 0HIT | RESERVED | WINDOW | TOTAL HIT | | TIMESTAMP | | | |
| STREAM 1; HIT 1 | | STREAM 1; HIT 0 | | STREAM 0; HIT 5 | | STREAM 0; HIT 4 | | STREAM 0; HIT 3 | | STREAM 0; HIT 2 | | STREAM 0; HIT 1 | | STREAM 0; HIT 0 | |
| STREAM 2; HIT 0 | | STREAM 1; HIT 8 | | STREAM 1; HIT 7 | | STREAM 1; HIT 6 | | STREAM 1; HIT 5 | | STREAM 1; HIT 4 | | STREAM 1; HIT 3 | | STREAM 1; HIT 2 | |
| STREAM 2; HIT 8 | | STREAM 2; HIT 7 | | STREAM 2; HIT 6 | | STREAM 2; HIT 5 | | STREAM 2; HIT 4 | | STREAM 2; HIT 3 | | STREAM 2; HIT 2 | | STREAM 2; HIT 1 | |
| STREAM 3; HIT 4 | | STREAM 3; HIT 3 | | STREAM 3; HIT 2 | | STREAM 3; HIT 1 | | STREAM 3; HIT 0 | | STREAM 2; HIT 11 | | STREAM 2; HIT 10 | | STREAM 2; HIT 9 | |
| PADDING | | | | | | | | | | STREAM 3; HIT 7 | | STREAM 3; HIT 6 | | STREAM 3; HIT 5 | |
| 127...120 | 119...112 | 111...104 | 103...96 | 95...88 | 87...80 | 79...72 | 71...64 | 63...56 | 55...48 | 47...40 | 39...32 | 31...24 | 23...16 | 15...8 | 7...0 |

**Figure 5.12:** *The $M^2EGP$ data format.*

a Timestamp + Finetime value within a programmable time window are used for the event

- the WINDOW (8bit) field contains the size of time window used for merging, with 100 ps LSB
- the TOTAL HIT (16bit) field contains the total number of hits of the merged event
- the fields STR X HIT(8 bit) have the information on the hits received from board X
- the fields STR X MGP(8 bit) contains the information on the number of MGP received from board X
- the fields STREAM (TEL62 board) X; HIT Y (16 bit) contain the ChannelID of individual hits

CHAPTER 6

# Conclusions

THE The work reported in this thesis was performed within the project "Experiment to detect KL Very Rare decays" (KLEVER. KLEVER aims at using powerful programmable systems in the first stages of the data collection and selection process in particle experiments at accelerators, i.e. the use of hardware processors based on Field-Programmable Gate Arrays (FPGAs) and Graphic Processing Units GPUs. My work has focused on the development of the Trigger and Data Acquisition System for the experiment NA62 and it covers almost all the aspects of the common Trigger and Data Acquisition of the NA62 experiment that has as main goal the measurement the Branching Ratio of the ultra-rare $K^+ \to \pi^+ \nu\overline{\nu}$ decay, very useful to obtain a stringent test of the Standard Model.

This PhD work began with the design, the development and the testing of the hardware and firmware of common boards of the NA62 TDAQ system: TDCB and TEL62. The TDCB is a daughter-board of the TEL62 and measures the detector hit times. The TEL62 processes and stores the received data in a buffer memory; at the arrival of a L0 trigger request, it extracts the data within a programmable time window around the trigger time to send them to the PC farm. The TEL62s of some detectors also take care of producing the L0 trigger primitives that are merged and processed to generate L0 trigger requests.

In this thesis is described (Chapter 3) the significant contribution given to the developing, the testing and the commissioning of the TDCB and TEL62 hardware and firmware. Since the first Technical Run to the 2016 Run the system that I have developed was tested, and evolved to be compatible with the detector input rate and the beam at growing intensity up to nominal. After three main versions the system composed by TDCB and TEL62 manages to cope with the design rate.

Once the work on the Data Acquisition system was concluded, my research activities were focused on the development of NA62 L0 standard trigger and L0 trigger based on GPU. In particular I have worked on L0 trigger firmware for RICH detector. Two versions of the standard firmware for the RICH detector were developed: one used during the 2015 NA62 data taking (*standard version*) and one used in 2016 NA62 data taking (*NEW RICH version*). The standard L0 firmware trigger is described in chapter 4. About the L0 trigger based on GPU, NA62 is considered test-bench for the use the GPUs in lowest level trigger. In this work we aim at demonstrating that GPUs can be usefully employed in a low level trigger. The studies on the performance of a L0 trigger based on GPU are currently underway. The work I have done during my last year of PhD is explained in Chapter 5.

# CHAPTER 7

# Appendix A

```
--
-- VHDL Architecture common_pp_lib.pp_data_compressor_v3.std
--
-- Created:
--          by - E. Pedreschi and J. Pinzino (UMMO)
--          at - 10:33:23 6/2/2015
--
-- using Mentor Graphics HDL Designer(TM) 2013.1b (Build 2)
--
library altera;
use altera.all;
library altera_mf;
use altera_mf.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library common_TEL62_lib;
use common_TEL62_lib.common_defs.all;
use common_TEL62_lib.memory_map.all;

library common_mgwz_generated;
use common_mgwz_generated.all;

entity pp_data_compressor_v3 is
  port(
    clock              : in    std_logic;
    reset              : in    std_logic;
    reset_error        : in    std_logic;
    soft_reset         : in    std_logic;
    recovery           : in    std_logic;
    enable             : in    std_logic;
    inburst            : in    std_logic;
    inendburst         : in    std_logic;
    freeze             : in    std_logic;
    alive              : out   std_logic;
    done                : out   std_logic;
    limiter_notrig      : in    std_logic;  -- Skip data to primitive FIFO if limiter
        is ON
    limiter_nodata      : in    std_logic;  -- Skip data to DDR writer if limiter is
        ON
    -- From/to organizer
    org_slot_timestamp  : in    std_logic_vector(31 downto 0); -- Must be read and
        changed together with RAM banks
    org_data_counter    : in    std_logic_vector(31 downto 0);
    slot_counters       : in    dataorg_counters_array_type;
    start               : in    std_logic;
    ready               : out   std_logic;
    org_emptyframe      : in    std_logic;
    fb_rdadd            : out   std_logic_vector(14 downto 0);
    fb_rddata           : in    std_logic_vector(31 downto 0);
    next_slot           : in    dataorg_nextslot_array_type;
    -- From/to DDR writer
    ddr_slot_timestamp  : out   std_logic_vector(31 downto 0);
    ddr_wr_start        : inout std_logic;
    ddr_wr_ready        : in    std_logic;
    ddr_wr_emptyframe   : out   std_logic;
    cb_rdadd            : in    std_logic_vector(8 downto 0);
    cb_rddata           : out   std_logic_vector(255 downto 0);
    word_address        : out   address_slot_array_type;
    -- From/to primitive trigger fifo
    prim_fifo_full      : in    std_logic;
    prim_fifo_enable_in : in    std_logic;
    prim_fifo_wdata     : inout std_logic_vector(31 downto 0);
    prim_fifo_wreq      : inout std_logic;
    tdc_warning         : in    std_logic;
    -- Monitoring
    proc_tot            : inout std_logic_vector(31 downto 0);
    proc_num            : inout std_logic_vector(31 downto 0);
    proc_max            : inout std_logic_vector(15 downto 0);
    proc_ovf            : out   std_logic;
    datacmp_frame_time  : out    std_logic_vector(31 downto 0);
    datacmp_frame_words : out    std_logic_vector(15 downto 0);
    -- Error
    error               : out   std_logic;
    comp_err_word       : out   std_logic_vector(31 downto 0);
    -- Logging
    log_permit          : in    std_logic;
    log_level           : in    std_logic_vector(1 downto 0);
    log_data            : out   std_logic_vector(LOG_DATASIZE-1 downto 0);
    log_valid           : inout std_logic;
    log_more            : out   std_logic;
    log_ack             : in    std_logic
  );
end entity pp_data_compressor_v3;

architecture std of pp_data_compressor_v3 is

  type READER_TYPE is (
    IDLE,
    WAIT_REG,
    START_STATE,
    READ_0,
    READ_1,
    WAIT_EMPTY,
    END_FRAME_WAIT_DDR_WRITER
  );
  signal reader_state : READER_TYPE;
  signal reader_alive : std_logic;

  type WRITER_TYPE is (
    IDLE,
    WRITE_EMPTY,
    WRITE_DATA,
   LAST_DATA,
   END_FRAME_WAIT_DDR_WRITER,
   ERR_STATE
);
signal writer_state : WRITER_TYPE;
signal writer_alive : std_logic;

-- Interaction signals
signal write_shift0 : std_logic_vector(1 downto 0);
signal write_shift1 : std_logic_vector(1 downto 0);
--signal writer_ready : std_logic := '0';
signal time_window_shift0 : std_logic_vector(7 downto 0);
signal time_window_shift1 : std_logic_vector(7 downto 0);
signal time_window_shift2 : std_logic_vector(7 downto 0);
signal start_writer0 : std_logic;
signal start_writer1 : std_logic;

-- Arrays
signal word_address_int : address_slot_array_type;
signal word_address0 : address_slot_array_type;
signal word_address1 : address_slot_array_type;

-- Memories
signal bank : std_logic := '0';
signal ab0_address : std_logic_vector(2 downto 0);
signal ab1_address : std_logic_vector(2 downto 0);
signal cb0_address : std_logic_vector(8 downto 0);
signal cb1_address : std_logic_vector(8 downto 0);
signal ab0_rddata_int : std_logic_vector(255 downto 0);
signal ab1_rddata_int : std_logic_vector(255 downto 0);
signal cb0_rddata_int : std_logic_vector(255 downto 0);
signal cb1_rddata_int : std_logic_vector(255 downto 0);
signal cb_wradd : std_logic_vector(8 downto 0);
signal cb_wrdata : std_logic_vector(255 downto 0);
signal cb_wrreq : std_logic;
signal cb0_wrreq : std_logic;
signal cb1_wrreq : std_logic;
signal cb_wrbyteena : std_logic_vector(31 downto 0);
signal cb0_wrbyteena : std_logic_vector(31 downto 0);
signal cb1_wrbyteena : std_logic_vector(31 downto 0);

-- Primitive generator FIFO
signal prim_fifo_wreq_int : std_logic;

-- Timestamp frame
signal slot_timestamp_0 : std_logic_vector(31 downto 0);
signal slot_timestamp_1 : std_logic_vector(31 downto 0);
signal emptyframe_0      : std_logic;
signal emptyframe_1      : std_logic;
signal emptyframe        : std_logic;

-- Reader signals
signal window_counter : std_logic_vector(6 downto 0); -- contatore parole singola
   finestra
signal empty_counter_reader : std_logic_vector(1 downto 0);
signal time_window_0 : std_logic_vector(7 downto 0);
signal next_time_window_0 : std_logic_vector(8 downto 0);
signal time_window_1 : std_logic_vector(7 downto 0);
signal next_time_window_1 : std_logic_vector(8 downto 0);
signal nword_window_0 : std_logic_vector(6 downto 0);
signal nword_window_1 : std_logic_vector(6 downto 0);
signal organizer_data_counter : std_logic_vector(31 downto 0);
--   signal start_int0     : std_logic;
--   signal start_int1     : std_logic;

-- Writer signals
--signal empty_counter_writer : std_logic_vector(2 downto 0);
--signal word_written : std_logic_vector(6 downto 0);
```

```
  signal word_addr : std_logic_vector(12 downto 0);
  signal trailer_word : std_logic_vector(31 downto 0); -- puo' diventare la parola che
       contiene gli errori

  -- Error signals
  signal error_state            : std_logic;
  signal error_overflow         : std_logic;
  signal frame_error_overflow   : std_logic;
  signal error_timeslot         : std_logic;
  signal frame_error_timeslot   : std_logic;
  signal error_datamatch        : std_logic;
  signal frame_error_datamatch  : std_logic;
  signal error_badword          : std_logic;
  signal frame_error_badword    : std_logic;
  signal violation_error        : std_logic;
  signal compressor_counter_error : std_logic;
  signal primitive_fifo_error : std_logic;

  -- Monitoring
  signal proc_cur : std_logic_vector(15 downto 0);
  signal proc_temp : std_logic_vector(17 downto 0);
  signal processing : std_logic;
  signal processing_prev : std_logic;
  signal waiting : std_logic;
  signal first_not_empty : std_logic;
  signal second_not_empty : std_logic;
  signal compressor_timestamp : std_logic_vector(31 downto 0);
  signal compressor_data_counter : std_logic_vector(31 downto 0);


  signal tdc_warning_prim         : std_logic;
  signal tdc_warning_data         : std_logic;

  -- Logger interface
  signal logd           : LOGDATA_TYPE;
  signal logn           : std_logic_vector(LOG2(LOG_MAXWORDS)-1 downto 0);
  signal logit          : std_logic;
  signal logbusy        : std_logic;
  signal logd_rd        : LOGDATA_TYPE;
  signal logn_rd        : std_logic_vector(LOG2(LOG_MAXWORDS)-1 downto 0);
  signal logit_rd       : std_logic;
  signal logd_wr        : LOGDATA_TYPE;
  signal logn_wr        : std_logic_vector(LOG2(LOG_MAXWORDS)-1 downto 0);
  signal logit_wr       : std_logic;
  signal rdbusy, wrbusy : std_logic;

  --tentativo
  signal counters_reg : dataorg_counters_array_type := (others => (others => '0'));
  signal next_slot_reg : dataorg_nextslot_array_type := (others => (others => '0'));

  -- attribute syn_encoding OF READER : TYPE IS "safe";

begin

  alive <= reader_alive and writer_alive;

  prim_fifo_wreq <= prim_fifo_wreq_int and (not prim_fifo_full) and
      prim_fifo_enable_in;

  -- Errors
  error <= error_state or error_overflow or error_timeslot or error_datamatch or
  error_badword or compressor_counter_error or (primitive_fifo_error and
      prim_fifo_enable_in);
  -- Bits to error data word
  violation_error <= frame_error_timeslot or frame_error_badword;
  comp_err_word <= X"0" & "000" & violation_error & frame_error_datamatch &
      frame_error_overflow & "00" & X"00000";
```

```
tdc_warning_prim <= tdc_warning and limiter_notrig;
tdc_warning_data <= tdc_warning and limiter_nodata;

--------------------------------------------------------------------------
-- Frame buffer RAM reader process
--------------------------------------------------------------------------
READER : process(clock)
begin
  if (rising_edge(clock)) then

    if (reset = '1') then
      write_shift0 <= (others => '0');
      write_shift1 <= (others => '0');
      time_window_shift0 <= (others => '0');
      time_window_shift1 <= (others => '0');
      time_window_shift2 <= (others => '0');
      empty_counter_reader <= (others => '0');
      reader_state <= IDLE;
      window_counter <= "0000001";
      start_writer0 <= '0';
      time_window_0 <= (others => '0');
      next_time_window_0 <= (others => '0');
      time_window_1 <= (others => '0');
      next_time_window_1 <= (others => '0');
      nword_window_0 <= (others => '0');
      nword_window_1 <= (others => '0');
      logit_rd <= '0';
      for i in 0 to LOG_MAXWORDS-1 loop
        logd_rd(i) <= (others => '0');
      end loop;
      error_timeslot <= '0';
      frame_error_timeslot <= '0';
      reader_alive <= '0';
      counters_reg <= (others => (others => '0'));
      next_slot_reg <= (others => (others => '0'));
      organizer_data_counter <= (others => '0');

    elsif (reset_error = '1') then
      error_timeslot <= '0';
      frame_error_timeslot <= '0';
      logit_rd <= '0';

    elsif (freeze = '0') then
      logit_rd <= '0';
      write_shift1 <= write_shift0;
      time_window_shift1 <= time_window_shift0;
      time_window_shift2 <= time_window_shift1;
      start_writer1 <= start_writer0;
      reader_alive <= enable;
      counters_reg <= slot_counters;
      next_slot_reg <= next_slot;

      if (enable = '1') then

        case reader_state is

          ----------------------------------------------------------
        when IDLE =>
          ----------------------------------------------------------
          ready <= '1';
          window_counter <= "0000001";
          empty_counter_reader <= (others => '0');
          write_shift0 <= (others => '0');
          write_shift1 <= (others => '0');
          time_window_shift0 <= (others => '0');
          time_window_0 <= (others => '0');
          next_time_window_0 <= (others => '0');
          time_window_1 <= (others => '0');
  next_time_window_1 <= (others => '0');
  start_writer0 <= '0';
  nword_window_0 <= (others => '0');
  nword_window_1 <= (others => '0');
  fb_rdadd <= (others => '0');
  if (start = '1') then
    reader_state <= WAIT_REG;
  else
    reader_state <= IDLE;
  end if;

  -------------------------------------------
when WAIT_REG =>
  -------------------------------------------
  next_time_window_0 <= next_slot_reg(0);
  nword_window_0 <= counters_reg(0);
  organizer_data_counter <= org_data_counter;
  reader_state <= START_STATE;

  -------------------------------------------
when START_STATE =>
  -------------------------------------------
  start_writer0 <= '1';
  emptyframe <= org_emptyframe;
  -- Reset frame error flags
  frame_error_timeslot <= '0';
  if (org_emptyframe = '1') then
    reader_state <= WAIT_EMPTY;
  else
    if (nword_window_0 = "0000000") then
      next_time_window_1 <= next_slot_reg(conv_integer(next_time_window_0(7
          downto 0)));
      time_window_1 <= next_time_window_0(7 downto 0);
      nword_window_1 <= counters_reg(conv_integer(next_time_window_0(7
          downto 0)));
      reader_state <= READ_1;
    else
      reader_state <= READ_0;
    end if;
  end if;

  -------------------------------------------
when WAIT_EMPTY =>
  -------------------------------------------
  ready <= '0';
  write_shift0 <= "11";
  empty_counter_reader <= empty_counter_reader + "01";
  if (empty_counter_reader = "10") then
    reader_state <= END_FRAME_WAIT_DDR_WRITER;
  else
    reader_state <= WAIT_EMPTY;
  end if;

  -------------------------------------------
when READ_0 =>
  -------------------------------------------
  ready <= '0';
  fb_rdadd <= time_window_0 & (window_counter - "0000001");
  time_window_shift0 <= time_window_0;
  if (window_counter = nword_window_0 or nword_window_0 = "0000000") then
    if (next_time_window_0 = "100000000") then
      write_shift0 <= "10";
      reader_state <= END_FRAME_WAIT_DDR_WRITER;
    else
      if(next_time_window_0(7 downto 0) <= time_window_0 ) then  -- Next
          timeslot is earlier than the one just processed
        error_timeslot <= '1';
        frame_error_timeslot <= '1';
```

```
          if (rdbusy = '0') then
            logd_rd(0) <= LOGMSG_DATACMP_TIMESLOT;
            logn_rd     <= conv_std_logic_vector(1,3);
            logit_rd    <= '1';
          end if;
        end if;
        next_time_window_1 <= next_slot_reg(conv_integer(next_time_window_0(7
            downto 0)));
        time_window_1 <= next_time_window_0(7 downto 0);
        nword_window_1 <= counters_reg(conv_integer(next_time_window_0(7
            downto 0)));
        window_counter <= "0000001";
        write_shift0 <= "10";
        reader_state <= READ_1;
      end if;
    else
      window_counter <= window_counter + "0000001";
      write_shift0 <= "01";
      reader_state <= READ_0;
    end if;

  -----------------------------------------
  when READ_1 =>
  -----------------------------------------
    ready <= '0';
    fb_rdadd <= time_window_1 & (window_counter - "0000001");
    time_window_shift0 <= time_window_1;
    if (window_counter = nword_window_1 or nword_window_1 = "0000000") then
      if (next_time_window_1 = "100000000") then
        write_shift0 <= "10";
        reader_state <= END_FRAME_WAIT_DDR_WRITER;
      else
        if (next_time_window_1(7 downto 0) <= time_window_1 ) then  -- Next
            timeslot is earlier than the one just processed
          error_timeslot <= '1';
          frame_error_timeslot <= '1';
          if (rdbusy = '0') then
            logd_rd(0) <= LOGMSG_DATACMP_TIMESLOT;
            logn_rd     <= conv_std_logic_vector(1, 3);
            logit_rd    <= '1';
          end if;
        end if;
        next_time_window_0 <= next_slot_reg(conv_integer(next_time_window_1(7
            downto 0)));
        time_window_0 <= next_time_window_1(7 downto 0);
        nword_window_0 <= counters_reg(conv_integer(next_time_window_1(7
            downto 0)));
        window_counter <= "0000001";
        write_shift0 <= "10";
        reader_state <= READ_0;
      end if;
    else
      window_counter <= window_counter + "0000001";
      write_shift0 <= "01";
      reader_state <= READ_1;
    end if;

  -----------------------------------------
  when END_FRAME_WAIT_DDR_WRITER =>
  -----------------------------------------
    write_shift0 <= "00";
    ready <= '0';
    start_writer0 <= '0';
    if (ddr_wr_ready = '1') then
      --            if (writer_ready = '1') then
      reader_state <= IDLE;
    else
      reader_state <= END_FRAME_WAIT_DDR_WRITER;
```

```
          end if;

        end case;
      end if;
    end if;
  end if;

end process;

----------------------------------------------------------------
-- Frame buffer RAM writer process
----------------------------------------------------------------
WRITER : process(clock)
begin
  if (rising_edge(clock)) then

    if (reset = '1') then
      --empty_counter_writer <= (others => '0');
      word_addr <= (others => '0');
      cb_wradd <= (others => '0');
      cb_wrreq <= '0';
      cb_wrdata <= (others => '0');
      ddr_wr_start <= '0';
      writer_state <= IDLE;
      cb_wrbyteena <= (others => '0');
      slot_timestamp_0 <= (others => '0');
      slot_timestamp_1 <= (others => '0');
      bank <= '0';
      done <= '0';
      writer_alive <= '0';
      logit_wr <= '0';
      for i in 0 to LOG_MAXWORDS-1 loop
        logd_wr(i) <= (others => '0');
      end loop;
      --writer_ready <= '0';
      word_address_int <= (others => (others => '0'));
      error_state <= '0';
      error_overflow <= '0';
      frame_error_overflow <= '0';
      error_datamatch <= '0';
      frame_error_datamatch <= '0';
      error_badword <= '0';
      frame_error_badword <= '0';
      compressor_timestamp <= (others => '0');
      compressor_data_counter <= (others => '0');
      compressor_counter_error <= '0';
      prim_fifo_wreq_int <= '0';
      prim_fifo_wdata <= (others => '0');
      primitive_fifo_error <= '0';

    elsif (reset_error = '1') then
      logit_wr <= '0';
      error_state <= '0';
      error_overflow <= '0';
      frame_error_overflow <= '0';
      error_datamatch <= '0';
      frame_error_datamatch <= '0';
      error_badword <= '0';
      frame_error_badword <= '0';
      primitive_fifo_error <= '0';

    elsif (freeze = '1') then
      logit_wr <= '0';
      cb_wrreq <= '0';
      ddr_wr_start <= '0';

    else
      logit_wr <= '0';
```

```
writer_alive <= enable;

done <= '1';  -- To be changed to indicate that all is finished after EOB...

if (cb_wrreq = '1') then
  compressor_data_counter <= compressor_data_counter + '1';
end if;

if (ddr_wr_start = '1') then
  if (compressor_data_counter = organizer_data_counter) then
    compressor_counter_error <= compressor_counter_error;
  else
    -- Counter mismatch
    compressor_counter_error <= '1';
    if (compressor_counter_error = '0' and wrbusy = '0') then
      logd_wr(0) <= LOGMSG_DATACMP_CNTERR;
      logd_wr(1) <= compressor_data_counter;
      logd_wr(2) <= organizer_data_counter;
      logn_wr    <= conv_std_logic_vector(3, 3);
      logit_wr   <= '1';
    end if;
  end if;
end if;

-- Primitive generator FIFO full
if (prim_fifo_full = '1' and prim_fifo_enable_in = '1') then
  primitive_fifo_error <= '1';
  if (primitive_fifo_error = '0' and wrbusy = '0') then
    logd_wr(0) <= LOGMSG_DATACMP_PRIFIFO;
    logn_wr    <= conv_std_logic_vector(1, 3);
    logit_wr   <= '1';
  end if;
end if;

if (enable = '1') then
  cb_wrreq <= '0';
  ddr_wr_start <= '0';
  prim_fifo_wreq_int <= '0';
  cb_wrbyteena <= (others => '0');  --importante per azzerare gli altri bit
      rispetto a quelli selezionati sul momento

  case writer_state is

    -----------------------------------------
  when IDLE =>
    -----------------------------------------
    -- Reset frame error flags
    frame_error_overflow <= '0';
    frame_error_datamatch <= '0';
    frame_error_badword <= '0';
    --writer_ready <= '0';
    --empty_counter_writer <= (others => '0');
    word_address_int <= (others => (others => '0')) ;
    word_addr <= (others => '0');
    ddr_wr_start <= '0';
    compressor_data_counter <= (others => '0');
    prim_fifo_wdata <= (others => '0');

    if (start_writer1 = '1') then
      compressor_timestamp <= org_slot_timestamp;
      if (bank = '0') then
        slot_timestamp_0 <= org_slot_timestamp;
      else
        slot_timestamp_1 <= org_slot_timestamp;
      end if;
      if(tdc_warning_data = '0') then
        case write_shift1 is
        when "11" =>
```

```
          prim_fifo_wdata <= TDC_FRAMETIME & org_slot_timestamp(31 downto 4);
          prim_fifo_wreq_int <= '1';
          writer_state <= WRITE_EMPTY;
        when "01" =>
          writer_state <= WRITE_DATA;
          prim_fifo_wdata <= TDC_FRAMETIME & org_slot_timestamp(31 downto 4);
          prim_fifo_wreq_int <= '1';
        when "10" =>
          writer_state <= LAST_DATA;
          prim_fifo_wdata <= TDC_FRAMETIME & org_slot_timestamp(31 downto 4);
          prim_fifo_wreq_int <= '1';
        when others =>
          writer_state <= IDLE;
        end case;
      else
        writer_state <= WRITE_EMPTY;
      end if;
    else
      writer_state <= IDLE;
    end if;

  ----------------------------------------------
  when WRITE_EMPTY =>
  ----------------------------------------------
  -- word_address_int <= (others => (others => '0')) ;
    writer_state <= END_FRAME_WAIT_DDR_WRITER;

  ----------------------------------------------
  when WRITE_DATA =>
  ----------------------------------------------
    if (word_addr = X"1000") then
      error_overflow <= '1';
      frame_error_overflow <= '1';
      if (wrbusy = '0') then
        logd_wr(0) <= LOGMSG_DATACMP_OVERFLOW;
        logd_wr(1) <= X"000000" & time_window_shift2;
        logn_wr    <= conv_std_logic_vector(2, 3);
        logit_wr   <= '1';
      end if;
      for i in 0 to 15 loop
        for j in 0 to 15 loop
          if ((i = conv_integer(time_window_shift2(7 downto 4)) and
          j >= conv_integer(time_window_shift2(3 downto 0))) or
          i > conv_integer(time_window_shift2(7 downto 4))) then
          word_address_int(i)(255-(j*16) downto 240-(j*16)) <= X"1000" ;
        end if;
      end loop;
    end loop;
    writer_state <= END_FRAME_WAIT_DDR_WRITER;
  else
    if (fb_rddata(31 downto 28) = TDC_LEADING or fb_rddata(31 downto 28) =
        TDC_TRAILING) then
      word_addr <= word_addr + X"1";
      --cb_wradd <= window_addr + conv_std_logic_vector(conv_integer(
          word_written)/8,8);
      --        cb_wradd <= (word_addr + X"1")(11 downto 3);
      --              if(word_addr(2 downto 0) = "111") then
      --                cb_wradd <= word_addr(11 downto 3) + X"1";
      --              else
      cb_wradd <= word_addr(11 downto 3);
      --              end if;
      cb_wrdata <= fb_rddata&fb_rddata&fb_rddata&fb_rddata&fb_rddata&fb_rddata
          &fb_rddata&fb_rddata; -- byteena writes just one word in the right
          place
      cb_wrreq <= '1';

      case conv_integer(word_addr(2 downto 0)) is
      when 0 => cb_wrbyteena(31 downto 28) <= X"F";
```

```
        when 1 => cb_wrbyteena(27 downto 24) <= X"F";
        when 2 => cb_wrbyteena(23 downto 20) <= X"F";
        when 3 => cb_wrbyteena(19 downto 16) <= X"F";
        when 4 => cb_wrbyteena(15 downto 12) <= X"F";
        when 5 => cb_wrbyteena(11 downto 8)  <= X"F";
        when 6 => cb_wrbyteena(7 downto 4)   <= X"F";
        when 7 => cb_wrbyteena(3 downto 0)   <= X"F";
        when others => cb_wrbyteena <= (others => '0');
        end case;

        if(tdc_warning_prim = '0') then
          prim_fifo_wdata <= fb_rddata;
          prim_fifo_wreq_int <= '1';
        end if;

        -- Check for word not belonging to current timeslot
        if (fb_rddata(15 downto 8) /= time_window_shift2 and fb_rddata(18 downto
             16) /= compressor_timestamp(10 downto 8)) then
          error_datamatch <= '1';
          frame_error_datamatch <= '1';
          if (wrbusy = '0') then
            logd_wr(0) <= LOGMSG_DATACMP_MISMATCH;
            logd_wr(1) <= fb_rddata;
            logd_wr(2) <= X"000000" & time_window_shift2;
            logn_wr    <= conv_std_logic_vector(3, 3);
            logit_wr   <= '1';
          end if;
        end if;

      else
        -- Unknown word
        error_badword <= '1';
        frame_error_badword <= '1';
        if (wrbusy = '0') then
          logd_wr(0) <= LOGMSG_DATACMP_BADWORD;
          logd_wr(1) <= fb_rddata;
          logn_wr    <= conv_std_logic_vector(2, 3);
          logit_wr   <= '1';
        end if;
      end if;

      if (write_shift1 = "01") then
        writer_state <= WRITE_DATA;
      elsif (write_shift1 = "10") then
        writer_state <= LAST_DATA;
      else
        writer_state <= ERR_STATE;
      end if;
    end if;

    ----------------------------------------
  when LAST_DATA =>
    ----------------------------------------
    if (word_addr = X"1000") then
      error_overflow <= '1';
      frame_error_overflow <= '1';
      if (wrbusy = '0') then
        logd_wr(0) <= LOGMSG_DATACMP_OVERFLOW;
        logd_wr(1) <= X"000000" & time_window_shift2;
        logn_wr    <= conv_std_logic_vector(2, 3);
        logit_wr   <= '1';
      end if;
      for i in 0 to 15 loop
        for j in 0 to 15 loop
          if((i = conv_integer(time_window_shift2(7 downto 4)) and
          j >= conv_integer(time_window_shift2(3 downto 0))) or
          i > conv_integer(time_window_shift2(7 downto 4))) then
          word_address_int(i)(255-(j*16) downto 240-(j*16)) <= X"1000" ;
        end if;
      end loop;
    end loop;
    writer_state <= END_FRAME_WAIT_DDR_WRITER;
  else

    word_addr <= word_addr + X"1";
    --cb_wradd <= window_addr + conv_std_logic_vector(conv_integer(word_written)
        /8,8);
    --        cb_wradd <= (word_addr + X"1")(11 downto 3);
    --              if(word_addr(2 downto 0) = "111") then
    --                cb_wradd <= word_addr(11 downto 3) + X"1";
    --              else
    cb_wradd <= word_addr(11 downto 3);
    --              end if;
    cb_wrdata <= fb_rddata&fb_rddata&fb_rddata&fb_rddata&fb_rddata&fb_rddata&
        fb_rddata&fb_rddata; -- byteena writes just one word in the right place
    cb_wrreq <= '1';

    case conv_integer(word_addr(2 downto 0)) is
    when 0 => cb_wrbyteena(31 downto 28) <= X"F";
    when 1 => cb_wrbyteena(27 downto 24) <= X"F";
    when 2 => cb_wrbyteena(23 downto 20) <= X"F";
    when 3 => cb_wrbyteena(19 downto 16) <= X"F";
    when 4 => cb_wrbyteena(15 downto 12) <= X"F";
    when 5 => cb_wrbyteena(11 downto 8)  <= X"F";
    when 6 => cb_wrbyteena(7 downto 4)   <= X"F";
    when 7 => cb_wrbyteena(3 downto 0)   <= X"F";
    when others => cb_wrbyteena <= (others => '0');
    end case;

    if(tdc_warning_prim = '0') then
      prim_fifo_wdata <= fb_rddata;
      prim_fifo_wreq_int <= '1';
    end if;

    -- Check for word not belonging to current timeslot
    if (fb_rddata(15 downto 8) /= time_window_shift2 and fb_rddata(18 downto 16)
         /= compressor_timestamp(10 downto 8)) then
      error_datamatch <= '1';
      frame_error_datamatch <= '1';
      if (wrbusy = '0') then
        logd_wr(0) <= LOGMSG_DATACMP_MISMATCH;
        logd_wr(1) <= fb_rddata;
        logd_wr(2) <= X"000000" & time_window_shift2;
        logn_wr    <= conv_std_logic_vector(3, 3);
        logit_wr   <= '1';
      end if;
    end if;

    for i in 0 to 15 loop
      for j in 0 to 15 loop
        if ((i = conv_integer(time_window_shift2(7 downto 4)) and
        j >= conv_integer(time_window_shift2(3 downto 0))) or
        i > conv_integer(time_window_shift2(7 downto 4))) then
        word_address_int(i)(255-(j*16) downto 240-(j*16)) <= "000" & word_addr
            (12 downto 0) + X"1" ;
      end if;
    end loop;
  end loop;
  if (write_shift1 = "01") then
    writer_state <= WRITE_DATA;
  elsif (write_shift1 = "10") then
    writer_state <= LAST_DATA;
  elsif (write_shift1 = "00") then
    writer_state <= END_FRAME_WAIT_DDR_WRITER;
  else
    writer_state <= ERR_STATE;
```

```
          end if;
        end if;

        ----------------------------------------
      when END_FRAME_WAIT_DDR_WRITER =>
        ----------------------------------------
        if (ddr_wr_ready = '1') then
          prim_fifo_wdata <= TDC_FRAMECOUNT & compressor_data_counter(31 downto 4);
          prim_fifo_wreq_int <= '1';
          ddr_wr_start <= '1';
          --writer_ready <= '1';
          bank <= not bank;
          datacmp_frame_time <= compressor_timestamp;
          datacmp_frame_words <= "000" & word_addr;
          writer_state <= IDLE;
        else
          writer_state <= END_FRAME_WAIT_DDR_WRITER;
        end if;

        ----------------------------------------
      when ERR_STATE =>
        ----------------------------------------
        writer_alive <= '0';
        error_state <= '1';
        if (wrbusy = '0') then
          logd_wr(0) <= LOGMSG_DATACMP_STATE;
          logn_wr     <= conv_std_logic_vector(1, 3);
          logit_wr    <= '1';
        end if;

      end case;
    end if;
  end if;
  end if;

  end process;

----------------------------------------------------------
-- Compressed data buffers instances
----------------------------------------------------------
CB0 : entity common_mgwz_generated.ram_256x512
port map (
  address => cb0_address,
  byteena => cb0_wrbyteena,
  clock   => clock,
  data    => cb_wrdata,
  wren    => cb0_wrreq,
  q       => cb0_rddata_int
);

CB1 : entity common_mgwz_generated.ram_256x512
port map (
  address => cb1_address,
  byteena => cb1_wrbyteena,
  clock   => clock,
  data    => cb_wrdata,
  wren    => cb1_wrreq,
  q       => cb1_rddata_int
);

----------------------------------------------------------
-- Mux and demux for frame buffers and DDR writer
----------------------------------------------------------
COUNTER_PROC : process(clock)
begin
  if (rising_edge(clock)) then
    if (bank = '1') then
      emptyframe_0 <= emptyframe_0;
      emptyframe_1 <= emptyframe;
      word_address0 <= word_address0;
      word_address1 <= word_address_int;
    else
      emptyframe_0 <= emptyframe;
      emptyframe_1 <= emptyframe_1;
      word_address0 <= word_address_int;
      word_address1 <= word_address1;
    end if;
  end if;
end process;

ddr_slot_timestamp <= slot_timestamp_0 when bank = '1' else slot_timestamp_1;
cb0_address <= cb_wradd when bank = '0' else cb_rdadd;
cb1_address <= cb_wradd when bank = '1' else cb_rdadd;
cb_rddata <= cb0_rddata_int when bank = '1' else cb1_rddata_int;
cb0_wrbyteena <= cb_wrbyteena when bank = '0' else (others => '0');
cb1_wrbyteena <= cb_wrbyteena when bank = '1' else (others => '0');
cb0_wrreq <= cb_wrreq when bank = '0' else '0';
cb1_wrreq <= cb_wrreq when bank = '1' else '0';
-- emptyframe_0 <= emptyframe when bank = '0' else emptyframe_0;
-- emptyframe_1 <= emptyframe when bank = '1' else emptyframe_1;
ddr_wr_emptyframe <= emptyframe_0 when bank = '1' else emptyframe_1;
word_address <= word_address0 when bank = '1' else word_address1;
-- word_address0 <= word_address_int when bank = '0' else word_address0;
-- word_address1 <= word_address_int when bank = '1' else word_address1;

-----------------------------------
PROC_LATCH : process(clock)
-----------------------------------
begin
  if (rising_edge(clock)) then
    if (reset = '1') then
      processing <= '0';
      waiting <= '0';
    else
      if (ddr_wr_start = '1') then -- ddr_wr_start is asserted for 1 cycle only
        processing <= '1';
      end if;
      if (processing = '1' and waiting = '0' and ddr_wr_ready = '0') then
        waiting <= '1';
      end if;
      if (waiting = '1' and ddr_wr_ready = '1') then
        processing <= '0';
        waiting <= '0';
      end if;
    end if;
  end if;
end process;

-----------------------------------
PROC_CNT : process(clock)
-----------------------------------
begin
  if (rising_edge(clock)) then
    if (reset = '1') then
      processing_prev <= '0';
      proc_cur  <= (others => '0');
      proc_temp <= (others => '0');
      proc_tot  <= (others => '0');
      proc_max  <= (others => '0');
      proc_num  <= (others => '0');
      proc_ovf  <= '0';

    elsif (reset_error = '1') then
      proc_ovf <= '0';

    elsif (freeze = '0') then
```

```
      proc_cur <= proc_temp(17 downto 2);
      processing_prev <= processing;
      if (processing = '1') then
        if (processing_prev = '0') then
          proc_temp <= "00" & X"0001";
        elsif (proc_cur = X"FFFF" & "11") then
          proc_max <= X"FFFF";
          proc_ovf <= '1';
        else
          proc_temp <= proc_temp+1;
        end if;
      elsif (processing_prev = '1') then
        if (proc_cur > proc_max) then
          proc_max <= proc_cur;
        end if;
        if (proc_tot /= X"FFFFFFFF") then
          proc_tot <= proc_tot+proc_cur;
        else
          proc_ovf <= '1';
        end if;
        if (proc_num /= X"FFFFFFFF") then
          proc_num <= proc_num+1;
        else
          proc_ovf <= '1';
        end if;
      end if;
    end if;
  end if;
end process;

------------------------------
LOGMUX : process(clock)
------------------------------
begin
  if (rising_edge(clock)) then
    if (reset = '1') then
      logd   <= logd_rd;
      logn   <= logn_rd;
      logit  <= '0';
      rdbusy <= '0';
      wrbusy <= '0';
    elsif (rdbusy = '1') then
      if (logit_rd = '0' and logbusy = '0') then
        logit  <= '0';
        rdbusy <= '0';
      end if;
    elsif (wrbusy = '1') then
      if (logit_wr = '0' and logbusy = '0') then
        logit  <= '0';
        wrbusy <= '0';
      end if;
    elsif (logit_rd = '1') then
      logd   <= logd_rd;
      logn   <= logn_rd;
      logit  <= '1';
      rdbusy <= '1';
    elsif (logit_wr = '1') then
      logd   <= logd_wr;
      logn   <= logn_wr;
      logit  <= '1';
      wrbusy <= '1';
    else
      logit  <= '0';
      rdbusy <= '0';
      wrbusy <= '0';
    end if;
  end if;
end process;
```

```
------------------------------------------------------------------------
LOG_INT : entity common_tel62_lib.logger_interface
------------------------------------------------------------------------
port map(
  clk        => clock,
  reset      => reset,
  enable     => log_permit,
  logd       => logd,
  logn       => logn,
  logit      => logit,
  logbusy    => logbusy,
  log_data   => log_data,
  log_valid  => log_valid,
  log_more   => log_more,
  log_ack    => log_ack
);

end architecture std;


--
-- VHDL Architecture common_pp_lib.pp_data_organizer_v3.std
--
-- Created:
--          by - E. Pedreschi and J. Pinzino (KAON02)
--          at - 13:10:09 16/04/2015
--
-- using Mentor Graphics HDL Designer(TM) 2010.3 (Build 21)
--
library altera;
use altera.all;
library altera_mf;
use altera_mf.all;
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library common_TEL62_lib;
use common_TEL62_lib.common_defs.all;
use common_TEL62_lib.memory_map.all;

library common_pp_lib;
use common_pp_lib.all;

library common_mgwz_generated;
use common_mgwz_generated.all;

entity pp_data_organizer_v3 is
  port(
    clock              : in     std_logic;
    reset              : in     std_logic;
    reset_error        : in     std_logic;
    soft_reset         : in     std_logic;
    recovery           : in     std_logic;
    enable             : in     std_logic;
    inburst            : in     std_logic;
    inendburst         : in     std_logic;
    freeze             : in     std_logic;
    alive              : out    std_logic;
    done               : out    std_logic;
    -- From output buffer
    ob_empty           : in     std_logic;
    ob_data            : in     std_logic_vector(31 downto 0);
    ob_rdreq           : inout    std_logic;
    -- From/to compressor (ex ddr writer)
    slot_timestamp     : out    std_logic_vector(31 downto 0);
    slot_counters      : out    dataorg_counters_array_type;
    n_next_slot           : out   dataorg_nextslot_array_type;
    compressor_start      : inout std_logic;
    compressor_ready      : in    std_logic;
    compressor_emptyframe : inout   std_logic;
    fb_rdadd              : in    std_logic_vector(14 downto 0);
    fb_rddata             : out   std_logic_vector(31 downto 0);
    data_counter          : out   std_logic_vector(31 downto 0);
    -- Signals to/from logic (build version register)
    chip_address          : in     std_logic_vector(2 downto 0);
    -- To error handler
    tdc_error_1           : out   std_logic_vector(31 downto 0);
    tdc_error_2           : out   std_logic_vector(31 downto 0);
    tdc_error_3           : out   std_logic_vector(31 downto 0);
    timestamp_err         : out   std_logic_vector(31 downto 0);
    org_err_word          : out   std_logic_vector(31 downto 0);
    tdc_warning           : out   std_logic;
    -- Error
    error                 : out   std_logic;
    -- Logging
    log_permit            : in    std_logic;
    log_level             : in    std_logic_vector(1 downto 0);
    log_data              : out   std_logic_vector(LOG_DATASIZE-1 downto 0);
    log_valid             : inout std_logic;
    log_more              : out   std_logic;
    log_ack               : in    std_logic
  );

end entity pp_data_organizer_v3;

architecture std of pp_data_organizer_v3 is

  type STATE_TYPE is (
    IDLE,
    WAIT_DATA,
    READ_TIMESTAMP,
    READ_DATA,
    WRITE_DATA,
    END_FRAME_WAIT_DDR_WRITER,
    ERROR_STATE
  );
  signal current_state_reader : STATE_TYPE;
  signal current_state_writer : STATE_TYPE;
  signal reader_alive : std_logic;
  signal writer_alive : std_logic;
  signal error_badts : std_logic;
  signal frame_error_badts : std_logic;
  signal error_cntovf : std_logic;
  signal frame_error_cntovf : std_logic;
  signal error_chktdc : std_logic;
  signal frame_error_chktdc : std_logic;
  signal frame_error : std_logic;
  signal ob_empty_int : std_logic;
  signal ob_rdreq_int : std_logic;
  signal ob_rdreq_1 : std_logic;
  signal ob_rdreq_2 : std_logic;
  signal ob_data_reg : std_logic_vector(31 downto 0);
  signal counters_reg : std_logic_vector(6 downto 0);
  --signal counters_temp : std_logic_vector(6 downto 0);
  signal counters0 : dataorg_counters_array_type := (others => (others => '0'));
  signal counters1 : dataorg_counters_array_type := (others => (others => '0'));
  signal counters_int : dataorg_counters_array_type := (others => (others => '0'));
  signal next_slot0 : dataorg_nextslot_array_type := (others => (others => '0'));
  signal next_slot1 : dataorg_nextslot_array_type := (others => (others => '0'));
  signal next_slot_int : dataorg_nextslot_array_type;
  --  signal enable_next:  std_logic;
  --  signal reset_next:  std_logic;
  signal slot_timestamp_0 : std_logic_vector(31 downto 0) := (others => '0');
  signal slot_timestamp_1 : std_logic_vector(31 downto 0) := (others => '0');
  signal data_process : std_logic := '0';
  signal data_process_reg : std_logic := '0';
  signal fb_wradd : std_logic_vector(14 downto 0);
  signal fb_wradd_int : std_logic_vector(14 downto 0);
  signal fb_wrreq : std_logic;
  signal fb0_wrreq : std_logic;
  signal fb1_wrreq : std_logic;
  signal fb_wrdata : std_logic_vector(31 downto 0);
  signal fb_wrdata_int : std_logic_vector(31 downto 0);
  signal fb_bank : std_logic := '0';
  signal fb0_address : std_logic_vector(14 downto 0);
  signal fb1_address : std_logic_vector(14 downto 0);
  signal fb0a_wrreq : std_logic;
  signal fb0b_wrreq : std_logic;
  signal fb1a_wrreq : std_logic;
  signal fb1b_wrreq : std_logic;
  signal fb0a_rddata_int : std_logic_vector(31 downto 0);
  signal fb0b_rddata_int : std_logic_vector(31 downto 0);
  signal fb1a_rddata_int : std_logic_vector(31 downto 0);
  signal fb1b_rddata_int : std_logic_vector(31 downto 0);
  signal fb0_rddata_int : std_logic_vector(31 downto 0);
  signal fb1_rddata_int : std_logic_vector(31 downto 0);
  signal emptyframe : std_logic_vector(1 downto 0) := "11";
  signal rd_wait : std_logic := '0';
  signal mem_error_temp1 : std_logic_vector(31 downto 0);
  signal mem_error_temp2 : std_logic_vector(31 downto 0);
  signal mem_error_temp3 : std_logic_vector(31 downto 0);
  signal tdc_warning_int : std_logic;
  signal fb_writer_ready : std_logic;
  signal ob_data_counter : std_logic_vector(31 downto 0);
  signal ob_otherword_counter : std_logic_vector(31 downto 0);
  signal organizer_data_counter : std_logic_vector(31 downto 0);
  signal org_counter_error : std_logic;
  signal organizer_data_counter_0 : std_logic_vector(31 downto 0);
  signal organizer_data_counter_1 : std_logic_vector(31 downto 0);
  signal previous_slot : std_logic_vector(8 downto 0);
--   signal err_temp: std_logic;
  signal shift0a: std_logic;
  signal shift0b: std_logic;
  signal shift1a: std_logic;
  signal shift1b: std_logic;
  -- Logger interface
  signal badts_log  : std_logic;
  signal cntovf_log : std_logic;
  signal logd : LOGDATA_TYPE;
  signal logn : std_logic_vector(LOG2(LOG_MAXWORDS)-1 downto 0);
  signal logit : std_logic;
  signal logbusy : std_logic;
  signal logd_rd : LOGDATA_TYPE;
  signal logn_rd : std_logic_vector(LOG2(LOG_MAXWORDS)-1 downto 0);
  signal logit_rd : std_logic;
  signal logd_wr : LOGDATA_TYPE;
  signal logn_wr : std_logic_vector(LOG2(LOG_MAXWORDS)-1 downto 0);
  signal logit_wr : std_logic;
  signal rdbusy,wrbusy : std_logic;

begin

  alive <= reader_alive and writer_alive;

  -- Errors
  error <= error_badts or error_cntovf or error_chktdc or org_counter_error;
  -- Bits to error data word
  frame_error <= frame_error_badts or frame_error_chktdc;
  org_err_word <= X"00" & "00" & frame_error & frame_error_cntovf & X"00000";

  ob_rdreq <= ob_rdreq_int and not ob_empty;
```

```
-----------------------------------------------------------------
-- Output buffer fifo reader process
-----------------------------------------------------------------
OB_READER : process(clock)
begin
  if (rising_edge(clock)) then

    reader_alive <= enable;

    if (reset = '1') then
      fb_bank <= '0';
      error_badts <= '0';
      frame_error_badts <= '0';
      ob_rdreq_int <= '0';
      current_state_reader <= IDLE;
      data_process <= '0';
      slot_timestamp_0 <= (others => '0');
      slot_timestamp_1 <= (others => '0');
      rd_wait <= '0';
      done <= '0';
      timestamp_err <= (others => '0');
      reader_alive <= '0';
      logit_rd <= '0';
      for i in 0 to LOG_MAXWORDS-1 loop
        logd_rd(i) <= (others => '0');
      end loop;
      compressor_start <= '0';
      ob_data_counter <= (others => '0');
      ob_otherword_counter <= (others => '0');

    elsif (reset_error = '1') then
      error_badts <= '0';
      frame_error_badts <= '0';
      logit_rd <= '0';

    elsif (freeze = '0') then
      done <= '1'; -- To be changed to indicate that all is finished after EOB...
      logit_rd <= '0';
      compressor_start <= '0';
      if (enable = '1') then
        ob_rdreq_int <= '0';
        ob_empty_int <= ob_empty;
        data_process <= '0';
        compressor_start <= '0';

        case current_state_reader is

        ----------------------------------------
        when IDLE =>
        ----------------------------------------
          ob_otherword_counter <= (others => '0');
          if (ob_empty = '0') then
            ob_rdreq_int <= '1';
            if (ob_rdreq_int = '1') then
              data_process <= '1';
              current_state_reader <= READ_TIMESTAMP;
            else
              current_state_reader <= IDLE;
            end if;
          else
            current_state_reader <= IDLE;
          end if;

        ----------------------------------------
        when READ_TIMESTAMP =>
        ----------------------------------------
          -- Reset frame error flags
  frame_error_badts <= '0';
  if (ob_data(31 downto 28) = TDC_FRAMETIME) then
    data_process <= '1';
    timestamp_err <= ob_data(27 downto 0) & X"0";
    ob_otherword_counter <= X"00000001";
    if (fb_bank = '0') then
      slot_timestamp_0 <= ob_data(27 downto 0) & X"0";
    else
      slot_timestamp_1 <= ob_data(27 downto 0) & X"0";
    end if;
    if (ob_empty = '0') then
      ob_rdreq_int <= '1';
    end if;
    current_state_reader <= READ_DATA;
  else
    error_badts <= '1';
    frame_error_badts <= '1';
    if (rdbusy = '0') then
      logd_rd(0) <= LOGMSG_DATAORG_BADTS;
      logd_rd(1) <= ob_data;
      logn_rd <= conv_std_logic_vector(2,3);
      logit_rd <= '1';
    end if;
    current_state_reader <= ERROR_STATE;
  end if;

  ----------------------------------------
when READ_DATA =>
  ----------------------------------------
  if (ob_data(31 downto 28) = X"6" and ob_rdreq_1 = '1') then
       ob_otherword_counter <= ob_otherword_counter + X"00000001";
   end if;
  if (ob_empty = '0') then
    if (ob_data(31 downto 28) = TDC_FRAMECOUNT) then
      ob_data_counter <= X"0" & ob_data(27 downto 0) - ob_otherword_counter;
      current_state_reader <= END_FRAME_WAIT_DDR_WRITER;
      rd_wait <= '0';
    else
      data_process <= '1';
      ob_rdreq_int <= '1';
      current_state_reader <= READ_DATA;
    end if;
  else
    data_process <= '1';
    current_state_reader <= READ_DATA;
  end if;

  ----------------------------------------
when END_FRAME_WAIT_DDR_WRITER =>
  ----------------------------------------
  if (fb_writer_ready = '1') then
    fb_bank <= not fb_bank;
    tdc_warning <= tdc_warning_int;
    compressor_start <= '1';
    if (ob_rdreq = '0' and ob_data(31 downto 28) /= TDC_FRAMETIME) then
      current_state_reader <= IDLE;
    else
      data_process <= '1';
      if (ob_empty = '0') then
        ob_rdreq_int <= '1';
      end if;
      current_state_reader <= READ_TIMESTAMP;
    end if;
  else
    current_state_reader <= END_FRAME_WAIT_DDR_WRITER;
    if (ob_empty_int = '0' and rd_wait = '0' and ob_data(31 downto 28) /=
        TDC_FRAMETIME) then
      rd_wait <= '1';
```

```
          ob_rdreq_int <= '1';
        end if;
      end if;

      -----------------------------------------
    when ERROR_STATE =>
      -----------------------------------------
      reader_alive <= '0';
      current_state_reader <= ERROR_STATE;
      -- change...

      -----------------------------------------
    when others =>
      -----------------------------------------
      current_state_reader <= IDLE;

    end case;

   end if;
  end if;
 end if;
end process;

----------------------------------------------------------
-- Frame buffer RAM writer process
----------------------------------------------------------
FB_WRITER: process(clock)
begin
 if (rising_edge(clock)) then

   writer_alive <= enable;

   if (reset = '1') then
    fb_wradd <= (others => '0');
    fb_wrreq <= '0';
    fb_wrdata <= (others => '0');
    counters_int <= (others => (others => '0'));
    counters_reg <= (others => '0');
--   counters_temp <= (others => '0');
    emptyframe <= "11";
    current_state_writer <= IDLE;
    error_cntovf <= '0';
    frame_error_cntovf <= '0';
    error_chktdc <= '0';
    frame_error_chktdc <= '0';
    mem_error_temp1 <= (others => '0');
    mem_error_temp2 <= (others => '0');
    mem_error_temp3 <= (others => '0');
    tdc_error_1 <= (others => '0');
    tdc_error_2 <= (others => '0');
    tdc_error_3 <= (others => '0');
    fb_writer_ready <= '0';
    ob_data_reg <= (others => '0');
    data_process_reg <= '0';
    tdc_warning_int <= '0';
    --     err_temp <= '0';
    for i in 0 to 255 loop
      next_slot_int(i) <= "100000000" ;
    end loop;
    logit_wr <= '0';
    for i in 0 to LOG_MAXWORDS-1 loop
      logd_wr(i) <= (others => '0');
    end loop;
    writer_alive <= '0';
    organizer_data_counter <= (others => '0');
    org_counter_error <= '0';
    previous_slot <= (others => '0');
    elsif (reset_error = '1') then
      error_cntovf <= '0';
      frame_error_cntovf <= '0';
      error_chktdc <= '0';
      frame_error_chktdc <= '0';
      logit_wr <= '0';

    elsif (freeze = '1') then
      fb_wrreq <= '0';
      logit_wr <= '0';

    else
      logit_wr <= '0';
      fb_wrdata_int <= fb_wrdata;
      fb_wradd_int <= fb_wradd;
      ob_rdreq_1 <= ob_rdreq;
      ob_rdreq_2 <= ob_rdreq_1;
      ob_data_reg <= ob_data;
      data_process_reg <= data_process;

--          OLD
--          counters_reg <= counters_int(conv_integer(ob_data(15 downto 8)));
--
--          if (ob_rdreq = '1' and (ob_data(31 downto 28) = TDC_LEADING or ob_data(31
      downto 28) = TDC_TRAILING)) then
--            if (counters_int(conv_integer(ob_data(15 downto 8))) = DDR_MAXWORDSPERSLOT
      ) then
--              error_cntovf <= '1';
--              frame_error_cntovf <= '1';
--              if (wrbusy = '0') then
--                logd_wr(0) <= LOGMSG_DATAORG_CNTOVF;
--                --logd_wr(1) <= ob_data_reg;
--                logd_wr(1) <= x"000000" & '0' & counters_temp;
--                logn_wr <= conv_std_logic_vector(2,3);
--                logit_wr <= '1';
--              end if;
--            else
--              counters_int(conv_integer(ob_data(15 downto 8))) <= counters_int(
      conv_integer(ob_data(15 downto 8))) + "0000001";
--              counters_temp <= counters_int(conv_integer(ob_data(15 downto 8))) +
      "0000001";
--            end if;
--          end if;
--        NEW TRIAL

      if (ob_rdreq_1 = '1' and (ob_data(31 downto 28) = TDC_LEADING or ob_data(31
          downto 28) = TDC_TRAILING)) then
        if(ob_data(15 downto 8) = previous_slot(7 downto 0) and previous_slot(8) =
            '1') then
          if (counters_reg /= DDR_MAXWORDSPERSLOT) then
            counters_reg <= counters_reg + "0000001";
          end if;
        else
          counters_reg <= counters_int(conv_integer(ob_data(15 downto 8)));
          previous_slot <= '1' & ob_data(15 downto 8);
        end if;
      end if;

      if (ob_rdreq_2 = '1' and (ob_data_reg(31 downto 28) = TDC_LEADING or
          ob_data_reg(31 downto 28) = TDC_TRAILING)) then
        if (counters_reg = DDR_MAXWORDSPERSLOT) then
          error_cntovf <= '1';
          frame_error_cntovf <= '1';
          if (wrbusy = '0') then
            logd_wr(0) <= LOGMSG_DATAORG_CNTOVF;
            --logd_wr(1) <= ob_data_reg;
            logd_wr(1) <= x"000000" & '0' & counters_reg;
            logn_wr <= conv_std_logic_vector(2,3);
```

```
              logit_wr <= '1';
            end if;
          else
            counters_int(conv_integer(ob_data_reg(15 downto 8))) <= counters_reg + "
                0000001";
          end if;
        end if;

        if (fb_wrreq = '1') then
          organizer_data_counter <= organizer_data_counter + '1';
        end if;

        if (enable = '1') then
          fb_wrreq <= '0';
          --     ppmemfrerr_wreq <= '1';
          --        enable_next <= '0';
          --        reset_next <= '0';


          case current_state_writer is

            ----------------------------------------
          when IDLE =>
            ----------------------------------------
            -- Reset frame error flags
            frame_error_cntovf <= '0';
            frame_error_chktdc <= '0';
            mem_error_temp1 <= (others => '0');
            mem_error_temp2 <= (others => '0');
            mem_error_temp3 <= (others => '0');
            counters_int <= (others => (others => '0'));
            organizer_data_counter <= (others => '0');
        --    counters_temp <= (others => '0');
            counters_reg <= (others => '0');
            previous_slot <= (others => '0');
            tdc_warning_int <= '0';
            if (data_process = '1') then
              current_state_writer <= WAIT_DATA;
            else
              fb_writer_ready <= '1';
              current_state_writer <= IDLE;
            end if;

            ----------------------------------------
          when WAIT_DATA =>
            ----------------------------------------
            for i in 0 to 255 loop
              next_slot_int(i) <= "100000000" ;
            end loop;
            emptyframe(conv_integer(fb_bank)) <= '1';
            fb_writer_ready <= '0';
            current_state_writer <= WRITE_DATA;

            ----------------------------------------
          when WRITE_DATA =>
            ----------------------------------------
            if (data_process_reg = '1') then
              if (ob_rdreq_2 = '1' and (ob_data_reg(31 downto 28) = TDC_LEADING or
                  ob_data_reg(31 downto 28) = TDC_TRAILING)) then
                fb_wradd <= ob_data_reg(15 downto 8) & counters_reg; -- A timeslot can
                     hold at most DDR_MAXWORDSPERSLOT=128 words
                -- First frame buffer

--                if (counters_reg = DDR_MAXWORDSPERSLOT) then
--                  error_cntovf <= '1';
--                  frame_error_cntovf <= '1';
--                  if (wrbusy = '0') then
--                    logd_wr(0) <= LOGMSG_DATAORG_CNTOVF;
--                      logd_wr(1) <= ob_data_reg;
--                      logn_wr <= conv_std_logic_vector(2,3);
--                      logit_wr <= '1';
--                    end if;
--                  else
--                    counters_int(conv_integer(ob_data_reg(15 downto 8))) <=
    counters_reg + "0000001";
--                  end if;

                  if (counters_reg /= DDR_MAXWORDSPERSLOT) then
                  fb_wrdata <= ob_data_reg;
                  fb_wrreq <= '1';
                  end if;

                  for i in 0 to 255 loop
                    if (i < conv_integer(ob_data_reg(15 downto 8)) and next_slot_int(i)
                        > '0' & ob_data_reg(15 downto 8)) then
                      next_slot_int(i) <= '0' & ob_data_reg(15 downto 8) ;
                    end if;
                  end loop;

                  if (emptyframe(conv_integer(fb_bank)) = '1') then -- da levare, non
                      necessario, lasciare solo l'operazione dentro l'if
                    emptyframe(conv_integer(fb_bank)) <= '0';
                  end if;
                elsif (ob_rdreq_2 = '1' and ob_data_reg(31 downto  28) = TDC_ERROR) then
                  if (ob_data_reg(27 downto 26) /= chip_address(1 downto 0)) then
                    error_chktdc <= '1';
                    frame_error_chktdc <= '1';
                    if (wrbusy = '0') then
                      logd_wr(0) <= LOGMSG_DATAORG_ERRWORD;
                      logd_wr(1) <= ob_data_reg;
                      logn_wr <= conv_std_logic_vector(2,3);
                      logit_wr <= '1';
                    end if;
                  end if;
                  if (ob_data_reg(14 downto 0) = ZERO(14 downto 0)) then
                    if (ob_data_reg(16 downto 15) /= X"0") then
                      tdc_warning_int <= '1';
                    end if;
                    if (ob_data_reg(25 downto 24) = X"3") then
                      mem_error_temp3 <= mem_error_temp3 or (ZERO(31 downto 26) &
                          ob_data_reg(16 downto 15) & ZERO(23 downto 0));
                    elsif (ob_data_reg(25 downto 24) = X"2") then
                      mem_error_temp3 <= mem_error_temp3 or (ZERO(31 downto 18) &
                          ob_data_reg(16 downto 15) & ZERO(15 downto 0));
                    elsif (ob_data_reg(25 downto 24) = X"1") then
                      mem_error_temp3 <= mem_error_temp3 or (ZERO(31 downto 10) &
                          ob_data_reg(16 downto 15) & ZERO(7 downto 0));
                    elsif (ob_data_reg(25 downto 24) = X"0") then
                      mem_error_temp3 <= mem_error_temp3 or (ZERO(31 downto 2) &
                          ob_data_reg(16 downto 15));
                    end if;
                  else
                    if (ob_data_reg(25 downto 24) = X"1") then
                      mem_error_temp1 <= mem_error_temp1 or (ZERO(31 downto 30) &
                          ob_data_reg(29 downto 15) & ZERO(14 downto 0));
                    elsif (ob_data_reg(25 downto 24) = X"0") then
                      mem_error_temp1 <= mem_error_temp1 or (ZERO(31 downto 15) &
                          ob_data_reg(14 downto 0));
                    elsif (ob_data_reg(25 downto 24) = X"3") then
                      mem_error_temp2 <= mem_error_temp2 or (ZERO(31 downto 30) &
                          ob_data_reg(29 downto 15) & ZERO(14 downto 0));
                    elsif (ob_data_reg(25 downto 24) = X"2") then
                      mem_error_temp2 <= mem_error_temp2 or (ZERO(31 downto 15) &
                          ob_data_reg(14 downto 0));
                    end if;
                  end if;
```

```
              end if;
              current_state_writer <= WRITE_DATA;
            else
              tdc_error_1 <= mem_error_temp1;
              tdc_error_2 <= mem_error_temp2;
              tdc_error_3 <= mem_error_temp3;
              current_state_writer <= END_FRAME_WAIT_DDR_WRITER;
              if (organizer_data_counter = ob_data_counter) then
                org_counter_error <= org_counter_error;
              else
                org_counter_error <= '1';
                if (wrbusy = '0') then
                  logd_wr(0) <= LOGMSG_DATAORG_CNTERR;
                  logd_wr(1) <= organizer_data_counter;
                  logd_wr(2) <= ob_data_counter;
                  logn_wr <= conv_std_logic_vector(3,3);
                  logit_wr <= '1';
                end if;
              end if;
            end if;

            ----------------------------------------
          when END_FRAME_WAIT_DDR_WRITER =>
            ----------------------------------------
            if (compressor_ready = '1') then
              fb_writer_ready <= '1';
              current_state_writer <= IDLE;
            else
              current_state_writer <= END_FRAME_WAIT_DDR_WRITER;
            end if;

            ----------------------------------------
          when others =>
            ----------------------------------------
            current_state_writer <= IDLE;

          end case;

        end if;
      end if;
    end if;
  end process;

----------------------------------------------------------
-- Frame buffers
----------------------------------------------------------

FB0A : entity common_mgwz_generated.ram_16384x32
port map (
  address => fb0_address(13 downto 0),
  clock   => clock,
  data    => fb_wrdata_int,
  wren    => fb0a_wrreq,
  q       => fb0a_rddata_int
);

FB0B : entity common_mgwz_generated.ram_16384x32
port map (
  address => fb0_address(13 downto 0),
  clock   => clock,
  data    => fb_wrdata_int,
  wren    => fb0b_wrreq,
  q       => fb0b_rddata_int
);

FB1A : entity common_mgwz_generated.ram_16384x32
port map (
  address => fb1_address(13 downto 0),
  clock   => clock,
  data     => fb_wrdata_int ,
  wren     => fb1a_wrreq ,
  q        => fb1a_rddata_int
);

FB1B : entity common_mgwz_generated.ram_16384x32
port map (
  address => fb1_address(13 downto 0),
  clock   => clock ,
  data    => fb_wrdata_int ,
  wren    => fb1b_wrreq ,
  q       => fb1b_rddata_int
);

-------------------------------------------------------------------
-- Mux and demux for frame buffers and DDR writer
-------------------------------------------------------------------
slot_timestamp <= slot_timestamp_0 when fb_bank = '1' else slot_timestamp_1;
slot_counters <= counters0 when fb_bank = '1' else counters1;
n_next_slot <= next_slot0 when fb_bank = '1' else next_slot1;
fb0_address <= fb_wradd_int when fb_bank = '0' else fb_rdadd;
fb1_address <= fb_wradd_int when fb_bank = '1' else fb_rdadd;
fb_rddata <= fb0_rddata_int when fb_bank = '1' else fb1_rddata_int;
fb0_wrreq <= fb_wrreq when fb_bank = '0' else '0';
fb1_wrreq <= fb_wrreq when fb_bank = '1' else '0';
compressor_emptyframe <= emptyframe(1) when fb_bank = '0' else emptyframe(0);
data_counter <= organizer_data_counter_1 when fb_bank = '0' else
    organizer_data_counter_0;

--      counters0 <= counters_int when fb_bank = '0' else counters0;
--      counters1 <= counters_int when fb_bank = '1' else counters1;
--      next_slot0 <= next_slot_int when fb_bank = '0' else next_slot0;
--      next_slot1 <= next_slot_int when fb_bank = '1' else next_slot1;

--      fb0a_wrreq <= fb0_wrreq when fb0_address(14) = '0' else '0';
--      fb0b_wrreq <= fb0_wrreq when fb0_address(14) = '1' else '0';
--      fb1a_wrreq <= fb1_wrreq when fb1_address(14) = '0' else '0';
--      fb1b_wrreq <= fb1_wrreq when fb1_address(14) = '1' else '0';--
fb0_rddata_int <= fb0a_rddata_int when shift0b = '0' else fb0b_rddata_int;
fb1_rddata_int <= fb1a_rddata_int when shift1b = '0' else fb1b_rddata_int;

-------------------------------------------------------------------
-- Counters
-------------------------------------------------------------------
COUNTER_PROC : process(clock)
begin
  if (rising_edge(clock)) then
    if (fb_bank = '1') then
      counters0 <= counters0;
      counters1 <= counters_int;
      next_slot0 <= next_slot0;
      next_slot1 <= next_slot_int;
      organizer_data_counter_0 <= organizer_data_counter_0;
      organizer_data_counter_1 <= organizer_data_counter;
    else
      counters0 <= counters_int;
      counters1 <= counters1;
      next_slot0 <= next_slot_int;
      next_slot1 <= next_slot1;
      organizer_data_counter_0 <= organizer_data_counter;
      organizer_data_counter_1 <= organizer_data_counter_1;
    end if;
  end if;
end process;

ADDR0_PROC : process(clock)
begin
  if (rising_edge(clock)) then
```

```
      if (fb_wradd(14)='1') then
        fb0b_wrreq <= fb0_wrreq;
        fb0a_wrreq <= '0';
      else
        fb0b_wrreq <= '0';
        fb0a_wrreq <= fb0_wrreq;
      end if;
    end if;
  end process;

  ADDR1_PROC : process(clock)
  begin
    if (rising_edge(clock)) then
      if (fb_wradd(14)='1') then
        fb1b_wrreq <= fb1_wrreq;
        fb1a_wrreq <= '0';
      else
        fb1b_wrreq <= '0';
        fb1a_wrreq <= fb1_wrreq;
      end if;
    end if;
  end process;

  ----------------------------------------------------------
  -- Frame buffer RAM writer process
  ----------------------------------------------------------
  FB_SHIFTER: process(clock, reset)
  begin
    if (reset = '1') then
      shift0a <= '0';
      shift0b <= '0';
      shift1a <= '0';
      shift1b <= '0';
    elsif (rising_edge(clock)) then
      shift0a <= fb0_address(14);
      shift0b <= shift0a;
      shift1a <= fb1_address(14);
      shift1b <= shift1a;
    end if;
  end process;

  ----------------------------------------
  LOGMUX : process(clock)
  ----------------------------------------
  begin
    if (rising_edge(clock)) then
      if (reset = '1' or log_permit = '0') then
        logd <= logd_rd;
        logn <= logn_rd;
        logit <= '0';
        rdbusy <= '0';
        wrbusy <= '0';
      elsif (rdbusy = '1') then
        if (logit_rd = '0' and logbusy = '0') then
          logit <= '0';
          rdbusy <= '0';
        end if;
      elsif (wrbusy = '1') then
        if (logit_wr = '0' and logbusy <= '0') then
          logit <= '0';
          wrbusy <= '0';
        end if;
      elsif (logit_rd = '1') then
        logd <= logd_rd;
        logn <= logn_rd;
        logit <= '1';
        rdbusy <= '1';
      elsif (logit_wr = '1') then
```

```
        logd <= logd_wr;
        logn <= logn_wr;
        logit <= '1';
        wrbusy <= '1';
      else
        logd <= (others => (others => '0'));
        logit <= '0';
        rdbusy <= '0';
        wrbusy <= '0';
      end if;
    end if;
  end process;

  --------------------------------------------------------------------
  LOG_INT : entity common_tel62_lib.logger_interface
  --------------------------------------------------------------------
  port map(
    clk        => clock,
    reset      => reset,
    enable     => log_permit,
    logd       => logd,
    logn       => logn,
    logit      => logit,
    logbusy    => logbusy,
    log_data   => log_data,
    log_valid  => log_valid,
    log_more   => log_more,
    log_ack    => log_ack
  );

end architecture std;


--
-- VHDL Architecture common_pp_lib.pp_ddr_reader_v3.std
--
-- Created:
--          by - Elena Pedreschi (NBPEDRESCHI)
--          at - 14:30:51 10/02/2015
--
-- using Mentor Graphics HDL Designer(TM) 2010.3 (Build 21)
--

library altera;
use altera.all;

library altera_mf;
use altera_mf.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_misc.all;

library common_TEL62_lib;
use common_TEL62_lib.common_defs.all;
use common_TEL62_lib.memory_map.all;

library common_pp_lib;
use common_pp_lib.all;

library common_mgwz_generated;
use common_mgwz_generated.all;

entity pp_ddr_reader_v3 is
  port(
    clock                 : in   std_logic;
    reset                 : in   std_logic;
    error_reset           : in  std_logic;
    soft_reset            : in  std_logic;
    recovery              : in  std_logic;
    enable                : in  std_logic;
    inburst               : in  std_logic;
    inendburst            : in  std_logic;
    alive                 : out std_logic;
    done                  : out std_logic;
    freeze                : in  std_logic;
    error                 : out std_logic;
    -- To/From trigger receiver communication (trig_info_rx module)
    first_slot_timestamp  : in  std_logic_vector(31 downto 0);
    number_of_slots       : in  std_logic_vector(5 downto 0);  --max number-of-slots =
        32
    ddr_rd_start          : in  std_logic;
    ddr_rd_done           : out std_logic;
    identifier            : in  std_logic_vector (7 downto 0);
    reader_first_word     : in  std_logic_vector(35 downto 0);
    -- To/From arbiter
    rd_request            : out std_logic;
    rd_grant              : in  std_logic;
    -- To/From DDR
    ddr_rd_avl_read_req   : out std_logic;
    ddr_rd_avl_burstbegin : out std_logic;
    ddr_rd_avl_addr       : out std_logic_vector(25 downto 0);
    avl_rdata             : in  std_logic_vector(255 downto 0);
    ddr_rd_avl_size       : out std_logic_vector(8 downto 0);
    avl_ready             : in  std_logic;
    avl_rdata_valid       : in  std_logic;
    writer_timestamp      : in  std_logic_vector(31 downto 0);
    -- FIFO signals
        ------------------------------------------------------------------------
    ddr_rd_fifo_mon       : out std_logic_vector (9 downto 0);

    -- To/From datafifo_arbiter (new module)
    -- signals from/to data format fifo
        ------------------------------------------------------
    df_rdata          : out    std_logic_vector (35 downto 0);
    df_rdreq          : in     std_logic;
    df_empty          : inout  std_logic;
    -- signals from first word from triginfo_rx and total number of word
      ----------------------
    nw_rdata          : out    std_logic_vector (35 downto 0);
    nw_rdreq          : in     std_logic;
    nw_empty          : inout  std_logic;
    -- Logging
    log_permit        : in     std_logic;
    log_level         : in     std_logic_vector(1 downto 0);
    log_data          : out    std_logic_vector(LOG_DATASIZE-1 downto 0);
    log_valid         : inout  std_logic;
    log_more          : out    std_logic;
    log_ack           : in     std_logic;
    -- Memory error
    ppmemfrerr_raddr  : out    std_logic_vector(PP_MEM_FRERR_ASIZE-1 downto 0);
    ppmemfrerr_rdata  : in     std_logic_vector(PP_MEM_FRERR_DSIZE-1 downto 0)
  );
end entity pp_ddr_reader_v3;

architecture std of pp_ddr_reader_v3 is

  -- Signal def
  type STATE_TYPE is (S0, S0_WAIT1, S0_WAIT,
    S1, S1_WAIT, S1_WAIT0, S1_WAIT1,
    S2, S2_WAIT, S2_WAITDR, S2_WAIT1,
    S3, S3_WAIT, S3_WAIT1, S3_WAIT2, S3_WAIT3,
    S4,
    S5,
    S6, S6_WAIT,
  S7, S7_WAIT,
  S8,
  S9, S9_WAIT,
  S10, S10_WAIT1,
  S11, S11_WAIT,
S12);
signal DDR2RD_STATE              : STATE_TYPE;  -- DDR2 Read
signal DDR2AVLRD_STATE           : STATE_TYPE;  -- DDR2 avalon read
signal DDR2NOWTR_STATE           : STATE_TYPE;  -- DDR2 Number Of Word To Read
    in each slot
signal ERRORS_STATE              : STATE_TYPE;  -- ERRORS (read from RAM error)
signal DF_STATE                  : STATE_TYPE;  -- DATA FORMAT
signal DATA_STATE                : STATE_TYPE;  -- DATA FIFO
-- FSM Start Signals ----------------------------------------------
signal avl_start                 : std_logic;
signal ddr2_reg_ok               : std_logic;  -- DDR2NOWTR FSM start signal
signal error_start               : std_logic;  -- ERROR FSM start signal
-- parity signals
    ---------------------------------------------------------------------
signal dataparityin              : std_logic_vector(31 downto 0);
signal dataparityout             : std_logic;
signal dataparityinnw            : std_logic_vector(31 downto 0);
signal dataparityoutnw           : std_logic;
-- DDR2 Read Fifo FSM Signals ------------------------------------
signal ddr_row_cnt               : std_logic_vector (8 downto 0);  -- number of
    row to read counter
signal ddr2_add_rddata_reg       : std_logic_vector (767 downto 0);  -- 16 feb
    2015
signal firstlinetoread           : std_logic_vector (15 downto 0);
signal firstlinetoread_ff        : std_logic_vector (15 downto 0);
signal numberofwordtoread_ff     : std_logic_vector (15 downto 0);
signal number_of_slots_sf        : std_logic_vector (5 downto 0);
signal numberofwordtoread_sf     : std_logic_vector (15 downto 0);
signal numberofwordtoread        : std_logic_vector (15 downto 0);
signal first_slot_timestamp_sf   : std_logic_vector (31 downto 0);  -- first
    slot timestamp second frame
signal numberofline_int          : std_logic_vector (15 downto 0);  -- number of
     row to read (could be <256 or <512)
signal numberofline_int_ff       : std_logic_vector (15 downto 0);
signal numberofline_int_sf       : std_logic_vector (15 downto 0);
signal numberofline              : std_logic_vector (9 downto 0);  -- number of
    row to read (could be <256 or <512)
signal numberofline_ff           : std_logic_vector (9 downto 0);
signal numberofline_sf           : std_logic_vector (9 downto 0);
signal two_frame                 : std_logic;
signal ff_empty                  : std_logic;
signal sf_empty                  : std_logic;
signal tf_empty                  : std_logic;
signal ff_empty_reg              : std_logic;
signal sf_empty_reg              : std_logic;
signal number_of_slots_ff        : std_logic_vector (5 downto 0);
signal ddr2rd_offset             : std_logic_vector (2 downto 0);  -- first time
     stamp: position of the first word in the first row
signal address_flag              : std_logic_vector (1 downto 0);  -- check DDR
    address near the rollover
--
    ====================================================================================

signal first_slot_timestamp_prec : std_logic_vector (31 downto 0);  -- previous
    time stamp
--
    ====================================================================================

-- DDR2 Number Of Word To Read (DDR2NOWTR) FSM Signals ----------
signal slot_cnt                  : std_logic_vector (5 downto 0);
signal curr_slot_counter         : std_logic_vector (5 downto 0);
signal nes_cnt                   : std_logic_vector (5 downto 0);  -- not empty
    slot counter
```

```
signal notemptyslot                    : std_logic_vector (5 downto 0);  -- number of
    not empty slots
signal first_slot_timestamp_prec_reg : std_logic_vector (31 downto 0);
signal number_of_slots_ff_reg          : std_logic_vector (5 downto 0);
signal number_of_slots_sf_reg          : std_logic_vector (5 downto 0);
signal nowtr_data_int                  : std_logic_vector (15 downto 0);
-- ERROR FSM Signals -------------------------------------------
signal err_line_cnt                    : std_logic_vector (2 downto 0);  -- each frame
     has 4 erroro lines
signal frerr_ok                        : std_logic;
signal totalerror_ff                   : std_logic_vector (2 downto 0);
--signal totalerror_sf                   : std_logic_vector (2 downto 0);
signal totalcnterror                   : std_logic_vector (3 downto 0);
signal cnt_error                       : std_logic_vector (3 downto 0);  -- each frame
     has 4 erroro lines reserved => max 8 total error lines if there are 2 frames
signal header_error_words_reg          : std_logic_vector (31 downto 0);
-- Data Format FSM signals ------------------------------------------------
signal first_slot_timestamp_reg        : std_logic_vector (31 downto 0);
signal number_of_slots_reg             : std_logic_vector (5 downto 0);  -- number of
    slots requested
signal numberofline_reg                : std_logic_vector (9 downto 0);
signal identifier_reg                  : std_logic_vector (7 downto 0);
signal number_of_word_to_read          : std_logic_vector (7 downto 0);
signal number_of_word_to_read_reg      : std_logic_vector (15 downto 0);  -- total
    number of words to read
signal number_of_word_to_read_int      : std_logic_vector (15 downto 0);
signal number_of_word_to_read_cnt      : std_logic_vector (15 downto 0);
signal number_of_word                  : std_logic_vector (15 downto 0);
signal header                          : std_logic_vector (31 downto 0);
signal current_slot_time_check         : std_logic_vector (10 downto 0);
signal current_slot_time_check_error : std_logic                  := ('0');
signal read_line_cnt                   : std_logic_vector (9 downto 0);
signal error_rcnt                      : std_logic_vector(3 downto 0) := "0000";  --
    counter error read
signal error_wcnt                      : std_logic_vector(3 downto 0) := "0001";  --
    counter error write
signal early_timestamp_error           : std_logic                  := '0';

-----------------------------------------------------------------------
--signal avlsize_error_flag              : std_logic                  := '0';  -- if
     asserted an error in the avalon size occurred (avl size can't be 0)
signal errorframe_flag                 : std_logic                  := '0';  -- if
    the error frame is present
-- DDR2 READ (dr) FIFO SIGNALS 2048x256
-----------------------------------------------------------------------
signal dr_wrdata                     : std_logic_vector (255 downto 0);
signal dr_rdreq                      : std_logic;
signal dr_wrreq                      : std_logic;
signal dr_afull                      : std_logic;
signal dr_empty                      : std_logic;
signal dr_rdata                      : std_logic_vector (255 downto 0);
signal dr_usedw                      : std_logic_vector (10 downto 0);
-------------------------------------------------------------------------
--DATA FORMAT FIFO (df) SIGNALS 16384x36
-------------------------------------------------------------------------

signal df_wrdata       : std_logic_vector (35 downto 0);
signal df_wrdatashort : std_logic_vector (31 downto 0);
signal sopeop          : std_logic_vector (1 downto 0);
signal df_wrreq        : std_logic;
signal df_full         : std_logic;
signal df_afull        : std_logic;
signal df_usedw        : std_logic_vector (13 downto 0);
---------------------------------------------------------------------------
--FIRST WORD FROM  TRIGINFO_RX AND TOTAL NUMBER OF WORD FIFO(nw) 32x36
---------------------------------------------------------------------------
signal nw_wrdata       : std_logic_vector (35 downto 0);
signal nw_wrdatashort : std_logic_vector (33 downto 0);
```

```
  signal nw_wrreq        : std_logic;
  signal nw_full         : std_logic;
  signal nw_usedw        : std_logic_vector (4 downto 0);

  ---------------------------------------------------------------------------
  -- NUMBER OF WORD TO READ FOR EACH TIME STAMP REQUESTED FIFO 128x16
  ---------------------------------------------------------------------------
  signal nowtr_wrdata : std_logic_vector (15 downto 0);
  signal nowtr_rdreq  : std_logic;
  signal nowtr_wrreq  : std_logic;
  signal nowtr_afull  : std_logic;
  signal nowtr_rdata  : std_logic_vector (15 downto 0);
  signal nowtr_empty  : std_logic;
  signal nowtr_usedw  : std_logic_vector (6 downto 0);

  ---------------------------------------------------------------------------
  -- NOT EMPTY SLOT FIFO Signals
  signal nes_wrdata : std_logic_vector (5 downto 0);
  signal nes_rdreq  : std_logic;
  signal nes_wrreq  : std_logic;
  signal nes_full   : std_logic;
  signal nes_rdata  : std_logic_vector (5 downto 0);
  signal nes_empty  : std_logic;
  signal nes_usedw  : std_logic_vector (3 downto 0);

  ---------------------------------------------------------------------------
  -- ERROR FIFO 32X36
  ---------------------------------------------------------------------------
  signal error_wrdata : std_logic_vector (31 downto 0);
  signal error_rdreq  : std_logic;
  signal error_wrreq  : std_logic;
  signal error_afull  : std_logic;
  signal error_empty  : std_logic;
  signal error_rdata  : std_logic_vector (31 downto 0);
  signal error_usedw  : std_logic_vector (4 downto 0);

  ---------------------------------------------------------------------------
  -- ERROR HEADER FIFO 16X32
  ---------------------------------------------------------------------------
  signal errorhead_wrdata : std_logic_vector (31 downto 0);
  signal errorhead_rdreq  : std_logic;
  signal errorhead_wrreq  : std_logic;
  signal errorhead_empty  : std_logic;
  signal errorhead_full   : std_logic;
  signal errorhead_rdata  : std_logic_vector (31 downto 0);
  signal errorhead_usedw  : std_logic_vector (3 downto 0);

  ---------------------------------------------------------------------------
  -- LOGGER INTERFACE
  ---------------------------------------------------------------------------
  signal logd    : LOGDATA_TYPE := ((others=> (others=>'0')));
  signal logn    : std_logic_vector(LOG2(LOG_MAXWORDS)-1 downto 0);
  signal logit   : std_logic := ('0');
  signal logbusy : std_logic;

  signal error_int : std_logic := ('0');

  signal temp_wrdata1 : std_logic_vector(15 downto 0) := X"0000";
  signal temp_wrdata2 : std_logic_vector(15 downto 0) := X"0000";

begin

  alive <= '1';

  ddr_rd_fifo_mon <= (dr_afull & dr_empty & nowtr_afull & nowtr_empty & error_afull &
      error_empty & df_afull & df_empty & nw_full & nw_empty);
  done            <= '1';
```

```
df_wrdata       <= '0'&dataparityout&sopeop&df_wrdatashort;
dataparityin    <= df_wrdatashort;

nw_wrdata       <= '0'&dataparityoutnw&nw_wrdatashort;
dataparityinnw  <= nw_wrdatashort(31 downto 0);

error <= error_int;

-------------------------------------------------------------------------
RD_LOGIC: process(clock, reset, error_reset)
-------------------------------------------------------------------------
begin
  if (clock = '1' and clock'event) then
    if ((reset = '1')or (error_reset = '1')) then
      error_int <= '0';
    else
      error_int <= (error_int or current_slot_time_check_error or dr_afull or
          nowtr_afull or error_afull or df_afull or nw_full or early_timestamp_error
          );
      -- error_int <= (error_int or current_slot_time_check_error or
          early_timestamp_error);
    end if;
  end if;
end process RD_LOGIC;

-----------------------------------------------------------------------------------
-- FSM
-----------------------------------------------------------------------------------

----------------------------------------
-- DDR2 READ FSM
----------------------------------------

DDR2RDFSM : process (reset, clock) --read from ddr2 and put the data in a 512x256
    fifo (DDR2RDFIFO) once all the data are copied in the fifo the signal done is
    asserted

variable last_timestamp_req          : integer range 0 to 511 := 0;

begin
  if (reset = '1') then

    last_timestamp_req         := 0;
    ddr2_add_rddata_reg        <= (others => '0');
    rd_request                 <= '0';
    avl_start                  <= '0';
    ddr_rd_avl_addr            <= (others => '0');
    ddr_rd_avl_size            <= (others => '0');
    firstlinetoread            <= (others => '0');
    firstlinetoread_ff         <= (others => '0');
    numberofwordtoread_ff      <= (others => '0');
    number_of_slots_sf         <= (others => '0');
    numberofwordtoread_sf      <= (others => '0');
    numberofwordtoread         <= (others => '0');
    first_slot_timestamp_sf    <= (others => '0');
    dr_wrreq                   <= '0';
    ddr_rd_done                <= '0';
    first_slot_timestamp_prec  <= (others => '0');
    numberofline               <= (others => '0');
    numberofline_ff            <= (others => '0');
    numberofline_sf            <= (others => '0');
    numberofline_int           <= (others => '0');
    numberofline_int_ff        <= (others => '0');
    numberofline_int_sf        <= (others => '0');
    ddr_row_cnt                <= (others => '0');
    ddr2_reg_ok                <= '0';
    number_of_slots_ff         <= (others => '0');
    two_frame                  <= '0';
    ff_empty                   <= '1';
    sf_empty                  <= '1';
    tf_empty                  <= '1';
    ddr2rd_offset             <= (others => '0');
    error_start               <= '0';
    early_timestamp_error     <= '0';
    address_flag              <= "00";

   DDR2RD_STATE <= S0;
  else
    if (clock = '1' and clock'event) then

      ddr2_reg_ok    <= '0';

      case DDR2RD_STATE is
        --
          ------------------------------------------------------------------------------------------


      when S0 =>  -- waits for a start from trig_info rx module (Physics)


        last_timestamp_req          := 0;
        ddr2_add_rddata_reg        <= (others => '0');
        rd_request                 <= '0';
        avl_start                  <= '0';
        ddr_rd_avl_addr            <= (others => '0');
        ddr_rd_avl_size            <= (others => '0');
        firstlinetoread            <= (others => '0');
        firstlinetoread_ff         <= (others => '0');
        numberofwordtoread_ff      <= (others => '0');
        number_of_slots_sf         <= (others => '0');
        numberofwordtoread_sf      <= (others => '0');
        numberofwordtoread         <= (others => '0');
        first_slot_timestamp_sf    <= (others => '0');
        dr_wrreq                   <= '0';
        ddr_rd_done                <= '0';
        first_slot_timestamp_prec  <= (others => '0');
        numberofline               <= (others => '0');
        numberofline_ff            <= (others => '0');
        numberofline_sf            <= (others => '0');
        numberofline_int           <= (others => '0');
        numberofline_int_ff        <= (others => '0');
        numberofline_int_sf        <= (others => '0');
        ddr_row_cnt                <= (others => '0');
        ddr2_reg_ok                <= '0';
        number_of_slots_ff         <= (others => '0');
        two_frame                  <= '0';
        ff_empty                   <= '1';
        sf_empty                   <= '1';
        tf_empty                   <= '1';
        ddr2rd_offset              <= (others => '0');
        error_start                <= '0';
        address_flag               <= "00";

        if (ddr_rd_start = '1' and enable = '1' and freeze = '0') then  -- waiting
            for ddr_rd_start from trig_info module and enable, the freeze signal has
             to be 0

          DDR2RD_STATE                  <= S0_WAIT1;

        end if;
        --
          ------------------------------------------------------------------------------------------------------

      when S0_WAIT1 =>

        if (dr_afull = '0' and nowtr_afull = '0') then  -- if the almost full
            signals are asserted the operation can't start
      rd_request                 <= '1';
      first_slot_timestamp_prec <= conv_std_logic_vector((conv_integer(
          first_slot_timestamp)) - 1, 32);

      if(writer_timestamp < first_slot_timestamp) then
        early_timestamp_error <= '1';
      end if;

      DDR2RD_STATE               <= S0_WAIT;

    end if;
    --
        ------------------------------------------------------------------------------------------

  when S0_WAIT =>

    --  In the address side:
    --  bits (3 downto 0) (200 ns) -> TS position in the row
    --  bits (27 downto 4)         -> row position
    --  bit 32 ="1"                -> DDR address

    -- A row:  16 words of 16 bits

    -- 3 DDR Row for 32 TS ( 256 bits)  => avl_size = 3 (not true near address
        rollover)

    if(rd_grant = '1' and first_slot_timestamp_prec(28 downto 4) = '1'& X"FFFFFF
        " and address_flag = "00") then
      ddr_rd_avl_addr <= '1' & first_slot_timestamp_prec(28 downto 4);
      ddr_rd_avl_size <= "000000001";
      avl_start       <= '1';
      address_flag    <= "01";
      DDR2RD_STATE    <= S1;
    elsif (rd_grant = '1' and first_slot_timestamp_prec(28 downto 4) = '1'& X"
        FFFFFF" and address_flag = "01") then
      ddr_rd_avl_addr <= "10" & X"000000"; -- second time goes to address 0
      ddr_rd_avl_size <= "000000010";
      avl_start       <= '1';
      address_flag    <= "00";
      DDR2RD_STATE    <= S1_WAIT;
    elsif (rd_grant = '1' and first_slot_timestamp_prec(28 downto 4) = '1'& X"
        FFFFFE" and address_flag = "00") then
      ddr_rd_avl_addr <= '1' & first_slot_timestamp_prec(28 downto 4);
      ddr_rd_avl_size <= "000000010";
      avl_start       <= '1';
      address_flag    <= "10";
      DDR2RD_STATE    <= S1;
    elsif (rd_grant = '1' and first_slot_timestamp_prec(28 downto 4) = '1'& X"
        FFFFFE" and address_flag = "10") then
      ddr_rd_avl_addr <= "10" & X"000000"; --second time goes to address 0
      ddr_rd_avl_size <= "000000001";
      avl_start       <= '1';
      address_flag    <= "00";
      DDR2RD_STATE    <= S2;
    elsif (rd_grant = '1') then
      ddr_rd_avl_addr <= '1' & first_slot_timestamp_prec(28 downto 4);
      ddr_rd_avl_size <= "000000011";
      avl_start       <= '1';
      address_flag    <= "00";
      DDR2RD_STATE    <= S1;
    else
      DDR2RD_STATE    <= S0_WAIT;
    end if;

    first_slot_timestamp_sf <= ((first_slot_timestamp(31 downto 8) + X"000001")
        & X"00");
    last_timestamp_req       := conv_integer(first_slot_timestamp(7 downto 0)) +
        conv_integer(number_of_slots) - 1;
  --
    ------------------------------------------------------------------------------------------

when S1 =>  --  request burst read to the DDR2 ADDRESS SECTOR
  avl_start <= '0';
  if ((rd_grant = '1') and (avl_rdata_valid = '1'))then  -- avalon bus granted
      , starts DDR2 read request

    ddr2_add_rddata_reg(767 downto 512) <= avl_rdata; -- 16 address was read
        each address is 16 bit long (256 bit tot)

    if (address_flag = "01") then
      DDR2RD_STATE             <= S0_WAIT;
    else
      DDR2RD_STATE             <= S1_WAIT;
    end if;
  else
    DDR2RD_STATE               <= S1;
  end if;

  --
    ------------------------------------------------------------------------------------------

when S1_WAIT =>               -- waits for bursts been accepted
  avl_start <= '0';
  if ((rd_grant = '1') and (avl_rdata_valid = '1')) then

    ddr2_add_rddata_reg (511 downto 256) <= avl_rdata;

    if (address_flag = "10") then
      DDR2RD_STATE             <= S0_WAIT;
    else
      DDR2RD_STATE             <= S2;
    end if;
  else
    DDR2RD_STATE               <= S1_WAIT;
  end if;
  --
    ------------------------------------------------------------------------------------------------

when S2 =>                    -- waits for bursts been accepted

  avl_start <= '0';
  if ((rd_grant = '1') and (avl_rdata_valid = '1')) then
    ddr2_add_rddata_reg (255 downto 0) <= avl_rdata;
    DDR2RD_STATE                         <= S3;
  end if;

  --
    ------------------------------------------------------------------------------------------

when S3 =>

  error_start    <= '1'; -- start for ERROR FSM
  if last_timestamp_req > 255 then  -- === 2 FRAME
      ================================

    two_frame               <= '1';
    firstlinetoread_ff     <= ddr2_add_rddata_reg ((767 - ((conv_integer(
        first_slot_timestamp_prec(3 downto 0))* 16))) downto
    (752 - ((conv_integer(first_slot_timestamp_prec(3 downto 0))* 16))));

    if (first_slot_timestamp_prec(7 downto 0) > X"EF") then
      -- lastlinetoread_ff        <= (ddr2_add_rddata_reg (527 downto 512)); --
          useful
      --numberofwordtoread_ff = lastlinetoread_ff - firstlinetoread_ff
    numberofwordtoread_ff <= ((ddr2_add_rddata_reg (527 downto 512)) -
    ddr2_add_rddata_reg ((767 - ((conv_integer(first_slot_timestamp_prec(3
        downto 0))* 16))) downto
    (752 - ((conv_integer(first_slot_timestamp_prec(3 downto 0))* 16)))));
  else
    --lastlinetoread_ff         <= (ddr2_add_rddata_reg (271 downto 256)); --
         last line to read in the First Frame (ff)--useful
    numberofwordtoread_ff <= ((ddr2_add_rddata_reg (271 downto 256)) -
    ddr2_add_rddata_reg ((767 - ((conv_integer(first_slot_timestamp_prec(3
        downto 0))* 16))) downto
    (752 - ((conv_integer(first_slot_timestamp_prec(3 downto 0))* 16)))));
  end if;
  ddr2rd_offset          <= ddr2_add_rddata_reg((754 - ((conv_integer(
      first_slot_timestamp_prec(3 downto 0)) * 16))) downto
  (752 - ((conv_integer(first_slot_timestamp_prec(3 downto 0)) * 16))));
  number_of_slots_sf       <= conv_std_logic_vector((last_timestamp_req -255),
      6); --number of slots in the Second Frame
  number_of_slots_ff       <= (number_of_slots - conv_std_logic_vector((
      last_timestamp_req -255), 6)); --number of slots in the First Frame
  if ((first_slot_timestamp_prec(7 downto 0) > X"EF") and (
      last_timestamp_req  > 271)) then --first frame first row/ second frame
       third row
    numberofwordtoread_sf   <= ddr2_add_rddata_reg((255 - ((
        last_timestamp_req -272) * 16)) downto (240 - ((last_timestamp_req
        -272) * 16))); -- number of word to read in the Second Frame =
  elsif ((first_slot_timestamp_prec(7 downto 0) > X"EF") and (
      last_timestamp_req  < 271)) then --first frame first row/ second frame
       second row
    numberofwordtoread_sf   <= ddr2_add_rddata_reg((511 - ((
        last_timestamp_req -256) * 16)) downto (496 - ((last_timestamp_req
        -256) * 16))); -- number of word to read in the Second Frame =

  elsif (first_slot_timestamp_prec(7 downto 0) < X"F0") then   --first frame
       second row/ second frame third row
    numberofwordtoread_sf   <= ddr2_add_rddata_reg((255 - ((
        last_timestamp_req -256) * 16)) downto (240 - ((last_timestamp_req
        -256) * 16))); -- number of word to read in the Second Frame =
  end if;

else
  -- === 1 FRAME ==================================

  number_of_slots_ff <= number_of_slots;
  if (first_slot_timestamp(7 downto 0) = "00000000") then  -- first slot is
      slot 0

    firstlinetoread  <= (others => '0');
    ddr2rd_offset    <= (others => '0');
    numberofwordtoread <= ddr2_add_rddata_reg((767 - (((conv_integer(
        first_slot_timestamp_prec(3 downto 0))+ conv_integer(number_of_slots
        ))*16))) downto

    --
        ----------------------------------------------------------------------------------------------------------------

  else               -- first slot is not slot0
    firstlinetoread  <= ddr2_add_rddata_reg ((767 - ((conv_integer(
        first_slot_timestamp_prec(3 downto 0))* 16))) downto
    (752 - ((conv_integer(first_slot_timestamp_prec(3 downto 0))* 16))));
    numberofwordtoread <= (ddr2_add_rddata_reg((767 - (((conv_integer(
        first_slot_timestamp_prec(3 downto 0))+ conv_integer(number_of_slots
        ))*16))) downto
    (752 - (((conv_integer(first_slot_timestamp_prec(3 downto 0))+
        conv_integer(number_of_slots))*16))))) -
    (ddr2_add_rddata_reg((767 - ((conv_integer(first_slot_timestamp_prec(3
        downto 0))* 16))) downto
    (752 - (((conv_integer(first_slot_timestamp_prec(3 downto 0))* 16))))));
          -- total number of word to read
      ddr2rd_offset    <= ddr2_add_rddata_reg ((754 - ((conv_integer(
          first_slot_timestamp_prec(3 downto 0))* 16))) downto
      (752 - ((conv_integer(first_slot_timestamp_prec(3 downto 0))* 16))));
    end if;
  end if;
  DDR2RD_STATE                    <= S3_WAIT;

  --
      ----------------------------------------------------------------------------------------------------

when S3_WAIT =>

  if (numberofwordtoread = X"0000") then
    numberofline_int <= X"0000";
  else
    numberofline_int <= numberofwordtoread + (X"000" & '0' & ddr2rd_offset)+ X
        "0007";
  end if;
  if (numberofwordtoread_ff = X"0000") then
    numberofline_int_ff <= X"0000";
  else
    numberofline_int_ff <= numberofwordtoread_ff + (X"000" & '0' &
        ddr2rd_offset)+ X"0007";
  end if;
  numberofline_int_sf <= numberofwordtoread_sf+ X"0007";

  DDR2RD_STATE                    <= S3_WAIT2;

  --
      ----------------------------------------------------------------------------------------------------

when S3_WAIT2 =>


  numberofline_ff <= numberofline_int_ff(12 downto 3);
  numberofline_sf <= numberofline_int_sf(12 downto 3);

  if (two_frame = '1') then
    if (numberofline_int_ff(12 downto 3) > "0000000000") then
      ff_empty <= '0';
    else
      ff_empty <= '1';
    end if;
    if (numberofline_int_sf(12 downto 3) > "0000000000") then
      sf_empty <= '0';
    else
      sf_empty <= '1';
    end if;

    numberofwordtoread     <= numberofwordtoread_ff + numberofwordtoread_sf;

    numberofline <= numberofline_int_ff(12 downto 3) + numberofline_int_sf(12
        downto 3);

  else

    if (numberofline_int(12 downto 3) > "0000000000") then
      ff_empty <= '0';
    else
      ff_empty <= '1';
    end if;
    numberofline <= numberofline_int (12 downto 3);

  end if;

  DDR2RD_STATE                    <= S4;

  --
```

```
when S4 =>
  if (numberofline > "0000000000") then
    tf_empty <= '0';
  else
    tf_empty <= '1'; --total frame (tf) empty
  end if;
  ddr2_reg_ok    <= '1'; -- start for DDR2NOWTRFSM (Number Of Word To Read in
      each time stamp requested)
  error_start    <= '0';
  dr_wrreq <= '1';  -- DDR2 READ FIFO write request asserted

  dr_wrdata(255 downto 0) <= ((X"00000000")&(X"00000000")&(X"FFFFFFFF")&(X"
      00000"&identifier)&
  (reader_first_word)&
  (first_slot_timestamp)&
  (X"000000"&"00"&number_of_slots)&
  ("0000" & sf_empty & ff_empty & numberofline & numberofwordtoread));
  if last_timestamp_req > 255 then  -- 2 frame -> more than one access is
      required (one or two for each frame)

    if (numberofline_ff > X"0000") then  -- first frame not empty
      if (numberofline_ff < 256) then -- only one access is required in the
          first frame
        ddr_rd_avl_addr <= ('0' & first_slot_timestamp(23 downto 8) &
            firstlinetoread_ff (11 downto 3));
        ddr_rd_avl_size <= numberofline_ff(8 downto 0);
        avl_start       <= '1';
        DDR2RD_STATE <= S5;
      else                                 -- two accesses are required in the
          first frame
        ddr_rd_avl_addr <= ('0' & first_slot_timestamp(23 downto 8) &
            firstlinetoread_ff (11 downto 3));
        ddr_rd_avl_size <= "100000000";
        avl_start       <= '1';
        DDR2RD_STATE <= S6;
      end if;
    else                                   -- first frame is empty
      if (numberofline_sf > X"0000") then  -- second frame not empty
        if (numberofline_sf < 256) then --only one access is required in the
            second frame
          ddr_rd_avl_addr <= ('0' & (first_slot_timestamp(23 downto 8) + X"
              0001") & "000000000");  --firstlinetoread_sf (11 downto 3));
          ddr_rd_avl_size <= numberofline_sf(8 downto 0);
          avl_start       <= '1';
          DDR2RD_STATE    <= S8;
        else                               -- two accesses are required in the
            second frame
          ddr_rd_avl_addr <= ('0' & (first_slot_timestamp(23 downto 8) + X"
              0001") & "000000000");  --firstlinetoread_sf (11 downto 3));
          ddr_rd_avl_size <= "100000000";
          avl_start       <= '1';
          DDR2RD_STATE <= S9;
        end if;
      else    -- second frame is empty
        DDR2RD_STATE <= S12;
      end if;
    end if;
  else                                 -- 1 frame (one or two access)
    if (numberofline > X"0000") then  -- frame not empty
      if (numberofline < 256) then -- only one access is required
        ddr_rd_avl_addr <= ('0' & first_slot_timestamp(23 downto 8) &
            firstlinetoread (11 downto 3));
        ddr_rd_avl_size <= numberofline(8 downto 0);
        avl_start       <= '1';
        DDR2RD_STATE <= S10;
        else                                  -- two accesses are required in the
            first frame
          ddr_rd_avl_addr <= ('0' & first_slot_timestamp(23 downto 8) &
              firstlinetoread (11 downto 3));
          ddr_rd_avl_size <= "100000000";
          avl_start       <= '1';
          DDR2RD_STATE <= S11;
        end if;
      else                                   --  frame is empty
        DDR2RD_STATE <= S12;
      end if;
    end if;


  when S5 =>  --  request burst read to the DDR2 DATA SECTOR
    avl_start <= '0';
    dr_wrreq  <= '0';
    if (avl_rdata_valid = '1')then  -- avalon bus granted, starts DDR2 read
        request
      dr_wrreq  <= '1';
      dr_wrdata <= avl_rdata;
      ddr_row_cnt <= ddr_row_cnt + '1';
      if (ddr_row_cnt = conv_std_logic_vector((conv_integer(numberofline_ff)) -
          1, 9)) then
        ddr_row_cnt         <= (others => '0');
        DDR2RD_STATE        <= S7;
      end if;
    end if;
  when S6 =>
    avl_start <= '0';
    dr_wrreq  <= '0';
    if (avl_rdata_valid = '1')then  -- avalon bus granted, starts first DDR2
        read request
      dr_wrreq     <= '1';
      dr_wrdata    <= avl_rdata;
      ddr_row_cnt <= ddr_row_cnt + '1';
      if (ddr_row_cnt  = X"FF") then
        DDR2RD_STATE <= S6_WAIT;
        ddr_row_cnt      <= (others => '0');
        ddr_rd_avl_addr <= ('0' & first_slot_timestamp(23 downto 8) & (
            conv_std_logic_vector(conv_integer (firstlinetoread_ff (11 downto 3)
             +256) ,9)));
        ddr_rd_avl_size <= conv_std_logic_vector ((conv_integer (numberofline_ff
            )) - 256, 9);
        avl_start       <= '1';
      end if;
    end if;
  when S6_WAIT =>
    avl_start <= '0';
    dr_wrreq  <= '0';
    if (avl_rdata_valid = '1')then  -- avalon bus granted, starts second DDR2
        read request
      dr_wrreq  <= '1';
      dr_wrdata <= avl_rdata;
      ddr_row_cnt <= ddr_row_cnt + '1';
      if (ddr_row_cnt = conv_std_logic_vector((conv_integer(numberofline_ff)
          -256) - 1, 9)) then
        DDR2RD_STATE     <= S7;
      end if;
    end if;
  when S7 =>
    dr_wrreq  <= '0';
    if (numberofline_sf > X"0000") then  -- second frame not empty
      if (numberofline_sf < 256) then --only one access is required in the
          second frame
        ddr_rd_avl_addr <= ('0' & (first_slot_timestamp(23 downto 8) + X"0001")
            & "000000000"); --firstlinetoread_sf (11 downto 3));
        ddr_rd_avl_size <= numberofline_sf(8 downto 0);
```

```
          avl_start        <= '1';
          DDR2RD_STATE     <= S8;
        else                                   -- two accesses are required in the
            second frame
          ddr_rd_avl_addr <= ('0' & (first_slot_timestamp(23 downto 8) + X"0001")
              & "000000000");  --firstlinetoread_sf (11 downto 3));
          ddr_rd_avl_size <= "100000000";
          avl_start        <= '1';
          DDR2RD_STATE <= S9;
        end if;
      else                                     -- second frame is empty
        DDR2RD_STATE <= S12;
      end if;
    when S8 =>
      avl_start <= '0';
      dr_wrreq  <= '0';
      if (avl_rdata_valid = '1')then  -- avalon bus granted, starts DDR2 read
          request
        dr_wrreq  <= '1';
        dr_wrdata <= avl_rdata;
        ddr_row_cnt <= ddr_row_cnt + '1';
        if (ddr_row_cnt = conv_std_logic_vector((conv_integer(numberofline_sf)) -
            1, 9)) then
          ddr_row_cnt          <= (others => '0');
          DDR2RD_STATE         <= S12;
        end if;
      end if;
    when S9 =>
      avl_start <= '0';
      dr_wrreq  <= '0';
      if (avl_rdata_valid = '1')then  -- avalon bus granted, starts first DDR2
          read request
        dr_wrreq     <= '1';
        dr_wrdata    <= avl_rdata;
        ddr_row_cnt <= ddr_row_cnt + '1';
        if (ddr_row_cnt  = X"FF") then
          DDR2RD_STATE <= S9_WAIT;
          ddr_row_cnt      <= (others => '0');
          ddr_rd_avl_addr <= ('0' & (first_slot_timestamp(23 downto 8) + X"0001")
              & "100000000"); --firstlinetoread_sf (11 downto 3)) + 256;
          ddr_rd_avl_size <= conv_std_logic_vector (conv_integer (numberofline_ff)
               - 256, 9);
          avl_start        <= '1';
        end if;
      end if;
    when S9_WAIT =>
      avl_start <= '0';
      dr_wrreq  <= '0';
      if (avl_rdata_valid = '1')then  -- avalon bus granted, starts second DDR2
          read request
        dr_wrreq  <= '1';
        dr_wrdata <= avl_rdata;
        ddr_row_cnt <= ddr_row_cnt + '1';
        if (ddr_row_cnt = conv_std_logic_vector((conv_integer(numberofline_ff)
            -256) - 1, 9)) then
          ddr_row_cnt          <= (others => '0');
          DDR2RD_STATE         <= S12;
        end if;
      end if;
    when S10 =>
      avl_start <= '0';
      dr_wrreq  <= '0';
      if (avl_rdata_valid = '1')then  -- avalon bus granted, starts DDR2 read
          request
        dr_wrreq  <= '1';
        dr_wrdata <= avl_rdata;
        ddr_row_cnt <= ddr_row_cnt + '1';
        if (ddr_row_cnt   = conv_std_logic_vector((conv_integer(numberofline)) -
              1, 9)) then
            ddr_row_cnt  <= (others => '0');
            DDR2RD_STATE <= S12;
          end if;
        end if;
      when S11 =>
        avl_start <= '0';
        dr_wrreq  <= '0';
        if (avl_rdata_valid = '1')then  -- avalon bus granted, starts first DDR2
            read request
          dr_wrreq    <= '1';
          dr_wrdata   <= avl_rdata;
          ddr_row_cnt <= ddr_row_cnt + '1';
          if (ddr_row_cnt  = X"FF") then
            DDR2RD_STATE <= S11_WAIT;
            ddr_row_cnt      <= (others => '0');
            ddr_rd_avl_addr <= ('0' & first_slot_timestamp(23 downto 8) & (
                conv_std_logic_vector(conv_integer (firstlinetoread (11 downto 3)
                +256) ,9)));
            ddr_rd_avl_size <= conv_std_logic_vector (conv_integer (numberofline) -
                256, 9);
            avl_start        <= '1';
          end if;
        end if;
      when S11_WAIT =>
        avl_start <= '0';
        dr_wrreq  <= '0';
        if (avl_rdata_valid = '1')then  -- avalon bus granted, starts second DDR2
            read request
          dr_wrreq  <= '1';
          dr_wrdata <= avl_rdata;
          ddr_row_cnt <= ddr_row_cnt + '1';
          if (ddr_row_cnt = conv_std_logic_vector((conv_integer(numberofline)-256) -
              1, 9)) then
            ddr_row_cnt         <= (others => '0');
            DDR2RD_STATE <= S12;
          end if;
        end if;
        --
            ------------------------------------------------------------------------

      when S12 =>
        dr_wrreq  <= '0';
        if(DDR2NOWTR_STATE = S0 and ERRORS_STATE = S0) then
          DDR2RD_STATE <= S0;
          ddr_rd_done  <= '1';
        end if;
        --
            ------------------------------------------------------------------------


      when others =>
        DDR2RD_STATE <= S0;

      end case;
    end if;
  end if;
end process DDR2RDFSM;

--
    --------------------------------------------------------------------------


--------------------------------------
-- DDR2 AVALON READ FSM
--------------------------------------
DDR2AVLRDFSM : process (reset, clock)
```

```
begin
  if (reset = '1') then
    ddr_rd_avl_read_req    <= '0';       -- read signal
    ddr_rd_avl_burstbegin  <= '0';       -- start burst request

    DDR2AVLRD_STATE <= S0;
  else
    if (clock = '1' and clock'event) then
      case DDR2AVLRD_STATE is

        --
            ---------------------------------------------------------------------------


        when S0 =>

          if ((rd_grant = '1') and (avl_start = '1')) then  -- avalon bus granted,
              starts DDR2 read request
            ddr_rd_avl_read_req    <= '0';                    -- read signal
            ddr_rd_avl_burstbegin  <= '0';
            DDR2AVLRD_STATE        <= S0_WAIT;
          else
            ddr_rd_avl_read_req    <= '0';                    -- read signal
            ddr_rd_avl_burstbegin  <= '0';
            DDR2AVLRD_STATE        <= S0;
          end if;

        --
            ---------------------------------------------------------------------------


        when S0_WAIT =>  --wait in case of start during avalon not ready

          if ((rd_grant = '1') and (avl_ready = '1')) then  -- avalon bus granted,
              avalon bus ready
            ddr_rd_avl_read_req    <= '1';  -- read signal
            ddr_rd_avl_burstbegin  <= '1';  -- start burst request
            DDR2AVLRD_STATE        <= S1;
          else
            ddr_rd_avl_read_req    <= '0';  -- read signal
            ddr_rd_avl_burstbegin  <= '0';
            DDR2AVLRD_STATE        <= S0_WAIT;
          end if;

        --
            ---------------------------------------------------------------------------


        when S1 =>

          ddr_rd_avl_burstbegin <= '0';  -- remove burst request
          if (avl_ready = '1') then      -- burst request has been accepted
            ddr_rd_avl_read_req <= '0';  -- read signal deasserted
          end if;
          DDR2AVLRD_STATE <= S0;

        --
            ---------------------------------------------------------------------------


        when others =>
          DDR2AVLRD_STATE <= S0;

        end case;

    end if;
  end if;
end process DDR2AVLRDFSM;

--------------------------------------------------------
-- DDR2 Number Of Word To Read FOR EACH TIME STAMP FSM
--------------------------------------------------------
DDR2NOWTRFSM : process (reset, clock)

variable temp          : integer range 0 to 1024   := 0;

begin

  if (reset = '1') then

    slot_cnt      <= (others => '0');
    nes_cnt       <= (others => '0');
    nowtr_wrreq   <= '0';
    nes_wrreq <= '0';
    first_slot_timestamp_prec_reg <= (others => '0');
    nowtr_data_int <= (others => '0');

    DDR2NOWTR_STATE <= S0;

  else
    if (clock = '1' and clock'event) then
      case DDR2NOWTR_STATE is

      when S0 =>
        slot_cnt      <= (others => '0');
        nes_cnt       <= (others => '0');
        nowtr_data_int <= (others => '0');
        nowtr_wrreq   <= '0';
        nes_wrreq <= '0';
        if ((ddr2_reg_ok = '1') and (nowtr_afull = '0')) then
          first_slot_timestamp_prec_reg <= first_slot_timestamp_prec;
          DDR2NOWTR_STATE <= S1;
        end if;

      when S1 =>
        nowtr_wrreq <= '1';
        nowtr_wrdata  <= '0' & X"000" & ddr2rd_offset;  -- It indicates the reading
            start point in the first DDR row

        -- calculation of words to be read for each TS request will be performed in
            the state S2

        if (first_slot_timestamp(7 downto 0) = "00000000") then -- first time stamp
            requested is 0
          nowtr_data_int <= ddr2_add_rddata_reg((751 - (conv_integer(
              first_slot_timestamp_prec_reg(3 downto 0))* 16)) downto
          (736 - (conv_integer(first_slot_timestamp_prec_reg(3 downto 0))* 16)));
        else
          nowtr_data_int <= ddr2_add_rddata_reg((751 - (conv_integer(
              first_slot_timestamp_prec_reg(3 downto 0))*16)) downto
          (736 - (conv_integer(first_slot_timestamp_prec_reg(3 downto 0))*16))) -
          ddr2_add_rddata_reg((767 - (conv_integer(first_slot_timestamp_prec_reg(3
              downto 0))*16)) downto
          (752 - (conv_integer(first_slot_timestamp_prec_reg(3 downto 0))*16)));
          --current address less than the previous
        end if;

        slot_cnt <= slot_cnt + '1';

        if (tf_empty = '0') then
          DDR2NOWTR_STATE <= S2;
        else
          DDR2NOWTR_STATE <= S4;
        end if;
```

```
when S2 =>

  nowtr_wrreq <= '1';
  nowtr_wrdata <= nowtr_data_int;

  if (nowtr_data_int = x"0000") then
    nes_cnt <= nes_cnt ; -- slot empty
  else
    nes_cnt <= nes_cnt + '1'; -- slot not empty counter
  end if;

  if (slot_cnt = number_of_slots_ff) then
    if (two_frame = '1') then
      slot_cnt <= "000001";

      nowtr_data_int <= ddr2_add_rddata_reg((751 - ((conv_integer(
          first_slot_timestamp_prec_reg(3 downto 0))+ conv_integer(
          number_of_slots_ff))*16)) downto
      (736 - ((conv_integer(first_slot_timestamp_prec_reg(3 downto 0))+
          conv_integer(number_of_slots_ff))*16)));

      DDR2NOWTR_STATE <= S3;
    else
      DDR2NOWTR_STATE <= S4;
    end if;
  else

    nowtr_data_int <= ddr2_add_rddata_reg((751 - ((conv_integer(
        first_slot_timestamp_prec_reg(3 downto 0))+ conv_integer(slot_cnt))
        *16)) downto
    (736 - ((conv_integer(first_slot_timestamp_prec_reg(3 downto 0))+
        conv_integer(slot_cnt))*16))) -
    ddr2_add_rddata_reg((767 - ((conv_integer(first_slot_timestamp_prec_reg(3
        downto 0))+ conv_integer(slot_cnt))*16)) downto
    (752 - ((conv_integer(first_slot_timestamp_prec_reg(3 downto 0))+
        conv_integer(slot_cnt))*16)));

    slot_cnt <= slot_cnt + '1';

    DDR2NOWTR_STATE <= S2;
  end if;

  --
    ----------------------------------------------------------------------------------------

when S3 =>
  nowtr_wrreq <= '1';
  nowtr_wrdata <= nowtr_data_int;

  if (nowtr_data_int = X"0000") then
    nes_cnt <= nes_cnt ; -- slot empty
  else
    nes_cnt <= nes_cnt + '1';  -- slot not empty counter
  end if;


  if (slot_cnt = number_of_slots_sf ) then
    DDR2NOWTR_STATE <= S4;
  else

    nowtr_data_int <= ddr2_add_rddata_reg((751 - ((conv_integer(
        first_slot_timestamp_prec_reg(3 downto 0))+ conv_integer(
        number_of_slots_ff)+ conv_integer(slot_cnt))*16)) downto
    (736 - ((conv_integer(first_slot_timestamp_prec_reg(3 downto 0))+
        conv_integer(number_of_slots_ff)+ conv_integer(slot_cnt))*16))) -
    ddr2_add_rddata_reg((767 - ((conv_integer(first_slot_timestamp_prec_reg(3
        downto 0))+ conv_integer(number_of_slots_ff)+ conv_integer(slot_cnt))
        *16)) downto
      (752 - ((conv_integer(first_slot_timestamp_prec_reg(3 downto 0))+
        conv_integer(number_of_slots_ff)+ conv_integer(slot_cnt))*16)));

      slot_cnt <= slot_cnt + '1';

      DDR2NOWTR_STATE <= S3;
    end if;

    --
      ------------------------------------------------------------------------------------------------------------------------

  when S4 =>
    nowtr_wrreq <= '0';
    if (nes_full = '0') then
      nes_wrreq        <= '1';
      nes_wrdata       <= nes_cnt;
      DDR2NOWTR_STATE <= S0;
    end if;

  when others =>
    nowtr_wrreq <= '0';
    DDR2NOWTR_STATE <= S0;

  end case;
  end if;
 end if;
end process DDR2NOWTRFSM;

-- ----------------------------------------------------
-- -- TDCB/PP errors FSM
-- ----------------------------------------------------

ERRORSFSM : process (reset, clock)

begin
  if (reset = '1') then

    err_line_cnt  <= (others => '0');
    frerr_ok      <= '0';
    totalerror_ff <= (others => '0');
    --totalerror_sf <= (others => '0');
    cnt_error     <= (others => '0');
    error_wrreq   <= '0';

    ERRORS_STATE  <= S0;

  else
    if (clock = '1' and clock'event) then
      case ERRORS_STATE is

        --
          ------------------------------------------------------------------------------------------------------

        when S0 =>

          err_line_cnt  <= (others => '0');
          frerr_ok      <= '0';
          totalerror_ff <= (others => '0');
          --totalerror_sf <= (others => '0');
          cnt_error     <= (others => '0');
          error_wrreq   <= '0';
          errorhead_wrreq <= '0';

          if ((error_start = '1') and (error_afull = '0')) then
            ERRORS_STATE <= S1;
          end if;
```

```
--
  -----------------------------------------------------------------------------

when S1 =>
  error_wrreq <= '0';
  if (err_line_cnt = "100") then
    totalerror_ff     <= cnt_error(2 downto 0); -- number of error word in
        first frame
    if two_frame       = '0' then    -- only one frame
      --totalerror_sf <= (others => '0');
      errorhead_wrreq  <= '1';
      errorhead_wrdata <= ((X"000" & (cnt_error + "0001")) & (X"000") & (
          cnt_error)); --number of error words tot & totalerror_sf &
          totalerror_ff
      ERRORS_STATE  <= S0;
    else                          -- two frame
      cnt_error      <= (others => '0');
      err_line_cnt   <= (others => '0');
      ERRORS_STATE   <= S3;
    end if;
  else
    ppmemfrerr_raddr <=  (first_slot_timestamp(15 downto 8) & err_line_cnt(1
        downto 0)); --error memory address is prepared
    err_line_cnt      <= err_line_cnt + '1';
    ERRORS_STATE      <= S1_WAIT;
  end if;
  --
  -----------------------------------------------------------------------------

when S1_WAIT =>
  ERRORS_STATE      <= S1_WAIT0;  -- error memory address is registred
  --
  -----------------------------------------------------------------------------

when S1_WAIT0 =>
  ERRORS_STATE      <= S1_WAIT1;  -- error memory address is registred
  --
  -----------------------------------------------------------------------------

when S1_WAIT1 =>
  frerr_ok          <= or_reduce (ppmemfrerr_rdata (PP_MEM_FRERR_DSIZE-3 downto
       0)); -- if frerr_ok is asserted the error line is not empty
  -- the 2 MSB are not computed in or_reduce
  ERRORS_STATE      <= S2;
  --
  -----------------------------------------------------------------------------

when S2 =>
  if (frerr_ok = '1') then -- if frerr_ok asserted there is an error word
    error_wrreq       <= '1';
    error_wrdata      <= ppmemfrerr_rdata; -- the error word is written in
        error fifo
    cnt_error         <= cnt_error +'1';
  end if;
  ERRORS_STATE      <= S1;
  --
  ---------------------------------------------------------------------------------------

when S3 =>
  error_wrreq <= '0';
  if (err_line_cnt     = "100") then
   --totalerror_sf  <= cnt_error(2 downto 0);
    errorhead_wrreq  <= '1';
    errorhead_wrdata <= ((X"000" & (('0' & totalerror_ff) + cnt_error + "0001"
        )) & ("0000" & cnt_error) & ("00000" & totalerror_ff)); --number of
        error words tot & totalerror_sf(cnt_error) &totalerror_ff
    ERRORS_STATE    <= S0;
  else
```

```
          ppmemfrerr_raddr <=(first_slot_timestamp_sf(15 downto 8) & err_line_cnt(1
              downto 0)); --error memory address is prepared
          err_line_cnt      <= err_line_cnt + '1';
          ERRORS_STATE      <= S3_WAIT;
        end if;
        --
            -----------------------------------------------------------------------------------------

      when S3_WAIT  =>
        ERRORS_STATE     <= S3_WAIT1;  --address registred
        --
            -----------------------------------------------------------------------------------------

      when S3_WAIT1  =>
        ERRORS_STATE     <= S3_WAIT2;  --address registred
        --
            -----------------------------------------------------------------------------------------

      when S3_WAIT2 =>
        frerr_ok           <= or_reduce (ppmemfrerr_rdata (PP_MEM_FRERR_DSIZE-3 downto
            0)); -- if frerr_ok is asserted the error line is not empty
        -- the 2 MSB are not computed in or_reduce
        ERRORS_STATE       <= S3_WAIT3;
        --
            -----------------------------------------------------------------------------------------

      when S3_WAIT3 =>
        if (frerr_ok = '1') then
          error_wrreq      <= '1';
          error_wrdata     <= ppmemfrerr_rdata; -- the error word is written in error
              fifo
          cnt_error        <= cnt_error +'1';
        end if;
        ERRORS_STATE         <= S3;
      when others =>
        ERRORS_STATE     <= S0;
      end case;
    end if;
  end if;
end process ERRORSFSM;

-- ------------------------------------------
--   -- Data Format FSM
--   -----------------------------------------
DFFSM : process (reset, clock)

begin

  if (reset = '1') then
    dr_rdreq                     <= '0';
    error_rdreq                  <= '0';
    errorhead_rdreq              <= '0';
    nowtr_rdreq                  <= '0';
    df_wrreq                     <= '0';
    nw_wrreq                     <= '0';
    identifier_reg               <= (others => '0');
    first_slot_timestamp_reg     <= (others => '0');
    number_of_slots_reg          <= (others => '0');
    numberofline_reg             <= (others => '0');
    number_of_word_to_read_reg   <= (others => '0');
    header                       <= (others => '0');
    number_of_word_to_read       <= (others => '0');
    number_of_word_to_read_int   <= (others => '0');
    number_of_word_to_read_cnt   <= (others => '0');
    curr_slot_counter            <= (others => '0');
    number_of_word               <= (others => '0');
    df_afull                     <= '0';
    current_slot_time_check      <= (others => '0');
    current_slot_time_check_error <= '0';
    error_rcnt                  <= "0000";
    error_wcnt                  <= "0001";
    notemptyslot <= (others => '0');
    read_line_cnt               <= (others => '0');
    totalcnterror <=  (others => '0');
    header_error_words_reg <=  (others => '0');
    ff_empty_reg        <= '1';
    sf_empty_reg        <= '1';
    logit <= '0';

   DF_STATE <= S0;
else
   if (clock = '1' and clock'event) then
     sopeop <= "00";
     logit <= '0';
     if (df_usedw > X"7FE") then
       df_afull <= '1';
     else
       df_afull <= '0';
     end if;

     case DF_STATE is

       --
         ---------------------------------------------------------------------------------


     when S0 =>
       dr_rdreq                    <= '0';
       error_rdreq                 <= '0';
       errorhead_rdreq             <= '0';
       nowtr_rdreq                 <= '0';
       df_wrreq                    <= '0';
       nw_wrreq                    <= '0';
       identifier_reg              <= (others => '0');
       first_slot_timestamp_reg    <= (others => '0');
       number_of_slots_reg         <= (others => '0');
       numberofline_reg            <= (others => '0');
       header                      <= (others => '0');
       number_of_word_to_read      <= (others => '0');
       number_of_word_to_read_int <= (others => '0');
       number_of_word_to_read_cnt <= (others => '0');
       current_slot_time_check     <= (others => '0');
       current_slot_time_check_error <= '0';
       curr_slot_counter           <= (others => '0');
       number_of_word              <= (others => '0');
       error_rcnt                  <= "0000";
       error_wcnt                  <= "0001";
       notemptyslot <= (others => '0');
       read_line_cnt               <= (others => '0');
       totalcnterror <=  (others => '0');
       header_error_words_reg <=  (others => '0');
       ff_empty_reg        <= '1';
       sf_empty_reg        <= '1';

       if (dr_empty = '0' and df_afull = '0' and nw_full = '0' and nes_empty = '0'
           and errorhead_empty = '0') then
         dr_rdreq <= '1';  -- read request for DDR2RD (dr) FIFO asserted
         errorhead_rdreq <= '1';
         read_line_cnt <= "0000000001";
         nowtr_rdreq <= '1'; --read request for NUMBEROFWORDTOREAD (nowtr) FIFO
             asserted
         nes_rdreq <= '1';
         DF_STATE <= S1;
       end if;
       --
         ---------------------------------------------------------------------------------
```

```
when S1 =>
  dr_rdreq <= '0'; -- read request for DDR2RD (dr) FIFO deasserted
  errorhead_rdreq <= '0';
  nowtr_rdreq <= '0';
  nes_rdreq <= '0';
  DF_STATE <= S2;
  --
      ----------------------------------------------------------------------------------------

when S2 =>
  if (nw_full = '0') then
    nw_wrreq          <= '1';
    nw_wrdatashort <= dr_rdata(129 downto 96); -- reader_first_word (event
        number) is written
    identifier_reg              <= dr_rdata(139 downto 132);
    first_slot_timestamp_reg    <= dr_rdata(95 downto 64); -- first slot time
        stamp
    number_of_slots_reg         <= dr_rdata(37 downto 32); -- number of slot
        requested
    sf_empty_reg                <= dr_rdata(27);             -- second frame
        empty
    ff_empty_reg                <= dr_rdata(26);             -- first frame
        empty
    numberofline_reg            <= dr_rdata(25 downto 16); -- number of line
        to read
    number_of_word_to_read_reg <= dr_rdata(15 downto 0); -- total number of
        word to read
    number_of_word <= (number_of_word + nowtr_rdata); -- nowtr_rdata in S2 is
        the number of word inside a row not full ( offset)
    notemptyslot <= nes_rdata;
    totalcnterror <= errorhead_rdata(19 downto 16) - X"1";
    header_error_words_reg <= errorhead_rdata;
    if (errorhead_rdata(19 downto 16) = X"1") then
      errorframe_flag <= '0';
    else
      errorframe_flag <= '1';
    end if;
    DF_STATE         <= S2_WAIT;
  end if;
  --
      ----------------------------------------------------------------------------------------

when S2_WAIT =>
  nw_wrreq                    <= '1';
  if (totalcnterror /= X"0") then
    nw_wrdatashort <= conv_std_logic_vector((1 + conv_integer(notemptyslot) +
        conv_integer(dr_rdata(15 downto 0)) + 1 + conv_integer(totalcnterror))
        ,34);
  else
    nw_wrdatashort <= conv_std_logic_vector((1 + conv_integer(notemptyslot) +
        conv_integer(dr_rdata(15 downto 0))),34);
  end if;

  df_wrreq                    <= '1';
  df_wrdatashort              <= "0000000" & errorframe_flag & "000000" &
      dr_rdata(133 downto 132) & (X"FF") & "00" & notemptyslot; --FPGA header
       (dr_rdata(133 downto 132) is identifier(1 downto 0);
  --(notemptyslot is number of slots with data)
  if (notemptyslot = "000000") then
    DF_STATE <= S2_WAIT1; -- wait state: to watch error memory
  else
    if (ff_empty_reg ='1') then
      dr_rdreq <= '0';
      DF_STATE                    <= S2_WAIT1;
    else
      if (dr_empty = '0') then
        dr_rdreq <= '1';
        read_line_cnt <= read_line_cnt + "0000000001";
```

```
          DF_STATE                    <= S2_WAIT1;
        else
          dr_rdreq <= '0';
          DF_STATE                    <= S2_WAITDR;
        end if;
      end if;
    end if;
    --
        ------------------------------------------------------------------------------

  when S2_WAITDR =>
    df_wrreq <= '0';
    nw_wrreq <= '0';
    if (dr_empty = '0') then
      dr_rdreq <= '1';
      read_line_cnt <= (read_line_cnt + "0000000001");
      DF_STATE                    <= S2_WAIT1;
    else
      dr_rdreq <= '0';
      DF_STATE                    <= S2_WAITDR;
    end if;
    --
        ------------------------------------------------------------------------------

  when S2_WAIT1 =>
    df_wrreq <= '0';
    nw_wrreq <= '0';
    dr_rdreq <= '0';
    current_slot_time_check_error <= '0';
    if  (curr_slot_counter = number_of_slots_reg or notemptyslot = "000000")
        then --finished reading the slot
      if (totalcnterror /= X"0") then
        df_wrreq <= '1';
        df_wrdatashort <= header_error_words_reg; -- (header_error_words_reg <=
            ((X"000" & cnt_error) & ("00000" & totalerror_sf) & ("00000" &
            totalerror_ff))
        error_rdreq <= '1';
        error_rcnt <= (error_rcnt + "0001");
        DF_STATE <= S5;
      else
        DF_STATE <=S7;
      end if;
    else
      nowtr_rdreq <= '1';  -- nowtr FIFO read req asserted (read how many words)
      number_of_word_to_read_cnt <= X"0000"; --reset counter
      DF_STATE                    <= S3;
    end if;
    --
        ------------------------------------------------------------------------------

  when S3 =>
    nowtr_rdreq <= '0';
    DF_STATE                    <= S3_WAIT;
    --
        ------------------------------------------------------------------------------

  when S3_WAIT =>
    number_of_word_to_read      <= nowtr_rdata(7 downto 0);
    if (nowtr_rdata = X"0000") then
      number_of_word_to_read_int <= X"0000";
    else
      number_of_word_to_read_int <= nowtr_rdata - X"0001"; --number of word to
          read in the slot
    end if;
    DF_STATE                    <= S3_WAIT2;
    --
        ------------------------------------------------------------------------------
```

```
when S3_WAIT2 =>

  if (number_of_word_to_read = 0) then    -- the slot is empty
    if ((first_slot_timestamp_reg(7 downto 0) + ("00" & curr_slot_counter)) =
        X"FF") then --special case: begins the next frame
      if(dr_empty = '0' and  sf_empty_reg = '0') then
        number_of_word <= X"0000";
        curr_slot_counter <= (curr_slot_counter + "000001");
        if ((ff_empty_reg = '0') and (number_of_word = X"0000")) then -- It
            has already been accessed one row at the end of a slot,

            --
          -- but has not yet been used
          dr_rdreq <= '0';
        else
          dr_rdreq <= '1';
          read_line_cnt <= (read_line_cnt + "0000000001"); --increase the line
        end if;
        DF_STATE                  <= S2_WAIT1;
      elsif (sf_empty_reg = '1') then
        number_of_word <= X"0000";
        curr_slot_counter <= (curr_slot_counter + "000001");
        DF_STATE                  <= S2_WAIT1;
      else
        DF_STATE                  <= S3_WAIT2;
      end if;
    else
      DF_STATE                  <= S2_WAIT1;
      curr_slot_counter <= (curr_slot_counter + "000001");
    end if;
  else
    df_wrreq <= '1';
    df_wrdatashort <= ((X"00" & number_of_word_to_read) + X"0001") & ((
        first_slot_timestamp_reg(15 downto 0)) + (X"00" & "00" &
        curr_slot_counter)); -- slot header
    curr_slot_counter <= (curr_slot_counter + "000001");
    current_slot_time_check    <= ((first_slot_timestamp_reg(10 downto 0)) + (
        "00000" & curr_slot_counter)); -- LOGGER check sulla timestamp dei
        dati
    DF_STATE                  <= S3_WAIT3;
  end if;

  --
    ------------------------------------------------------------------------------------

when  S3_WAIT3 =>
  if (number_of_word_to_read_cnt = number_of_word_to_read_int) then --
      finished reading the slot
    df_wrreq <= '1';
    df_wrdatashort <= dr_rdata ((255-(32 * conv_integer(number_of_word)))
        downto (224-(32 * conv_integer(number_of_word))));

    ----------START LOGGER
        ------------------------------------------------------------------

    if (dr_rdata ((242-(32 * conv_integer(number_of_word))) downto (232-(32 *
        conv_integer(number_of_word)))) = current_slot_time_check) then
      current_slot_time_check_error <= '0';
    else
      current_slot_time_check_error <= '1';
      if (logbusy = '0') then
        logd(0) <= LOGMSG_DDRRD_TIMEMIS;
        logd(1) <= (first_slot_timestamp_reg + (X"000000" & "00" & (
            curr_slot_counter - "000001")));
        logd(2) <= X"00000" & '0' & current_slot_time_check;
        logd(3) <= X"00000" & '0' & (dr_rdata ((242-(32 * conv_integer(
            number_of_word))) downto (232-(32 * conv_integer(number_of_word)))
            ));
        logn <= conv_std_logic_vector(4,3);
      logit <= '1';
    end if;
  end if;
  ----------END LOGGER
      -------------------------------------------------------------------------

  if ((first_slot_timestamp_reg(7 downto 0) + ("00" & curr_slot_counter)) =
      X"00") then --special case: begins the next frame
    if (dr_empty = '0' and  sf_empty_reg = '0') then
      number_of_word <= X"0000";
      dr_rdreq <= '1';
      read_line_cnt <= (read_line_cnt + "0000000001"); --increase the row la
          riga
      DF_STATE              <= S2_WAIT1;
    elsif (sf_empty_reg = '1') then
      number_of_word <= X"0000";
      DF_STATE              <= S2_WAIT1;
    else
      df_wrreq <= '0';
      DF_STATE              <= S3_WAIT3;
    end if;
  elsif (number_of_word = X"0007") then --if the entire row is been accessed
    if (read_line_cnt = (numberofline_reg + "0000000001")) then  -- if all
        the rows are been accessed
      number_of_word <= X"0000";
      DF_STATE            <= S2_WAIT1;
    elsif (dr_empty = '0') then
      number_of_word <= X"0000";
      dr_rdreq <= '1';
      read_line_cnt <= (read_line_cnt + "0000000001");
      DF_STATE            <= S2_WAIT1;
    else
      df_wrreq <= '0';
      DF_STATE            <= S3_WAIT3;
    end if;
  else
    number_of_word <= (number_of_word + X"0001");
    DF_STATE            <= S2_WAIT1;
  end if;
else  -- the slot is not finished
  df_wrreq <= '1';
  df_wrdatashort <= dr_rdata ((255-(32 * conv_integer(number_of_word)))
      downto (224-(32 * conv_integer(number_of_word))));

  ----------START LOGGER
      -------------------------------------------------------------------------
  if (dr_rdata ((242-(32 * conv_integer(number_of_word))) downto (232-(32 *
      conv_integer(number_of_word)))) = current_slot_time_check) then
    current_slot_time_check_error <= '0';
  else
    current_slot_time_check_error <= '1';
    if (logbusy = '0') then
      logd(0) <= LOGMSG_DDRRD_TIMEMIS;
      logd(1) <= (first_slot_timestamp_reg + (X"000000" & "00" & (
          curr_slot_counter - "000001")));
      logd(2) <= X"00000" & '0' & current_slot_time_check;
      logd(3) <= X"00000" & '0' & (dr_rdata ((242-(32 * conv_integer(
          number_of_word))) downto (232-(32 * conv_integer(number_of_word)))
          ));
      logn <= conv_std_logic_vector(4,3);
      logit <= '1';
    end if;
  end if;
  ----------END LOGGER
      -------------------------------------------------------------------------

  if (number_of_word = X"0007") then --if the entire row is been accessed
    if (dr_empty = '0') then
```

```
          DF_STATE                  <= S4;
          number_of_word_to_read_cnt <= (number_of_word_to_read_cnt + X"0001");
          number_of_word <= X"0000";
          dr_rdreq <= '1';
          read_line_cnt <= (read_line_cnt + "0000000001");
        else
          DF_STATE                  <= S3_WAIT3;
          df_wrreq <= '0';
        end if;
        --              end if;
      else -- --if the entire row is not been accessed
        number_of_word_to_read_cnt <= (number_of_word_to_read_cnt + X"0001");
        number_of_word <= (number_of_word + X"0001");
        DF_STATE                    <= S3_WAIT3;
      end if;
    end if;
    --
    ---------------------------------------------------------------------------------------------

  when S4 =>
    current_slot_time_check_error <= '0';
    dr_rdreq <= '0';
    df_wrreq <= '0';
    DF_STATE                    <= S3_WAIT3;
    --
    ---------------------------------------------------------------------------------------------

  when S5 =>
    if ((error_rcnt) = totalcnterror) then
      error_rdreq <= '0';
    else
      error_rdreq <= '1';
      error_rcnt <= (error_rcnt + "0001");
    end if;
    df_wrreq <='0';
    DF_STATE                    <= S6;
    --
    ---------------------------------------------------------------------------------------------

  when S6 =>
    if ((error_rcnt) = totalcnterror) then
      error_rdreq <= '0';
    else
      error_rdreq <= '1';
      error_rcnt <= (error_rcnt + "0001");
    end if;

    df_wrreq <= '1';
    df_wrdatashort <= error_rdata;
    error_wcnt <= (error_wcnt + "0001");
    if ((error_wcnt) = totalcnterror) then
      DF_STATE <=S7;
    else
      DF_STATE <=S6;
    end if;
    --
    ---------------------------------------------------------------------------------------------

  when S7 =>
    df_wrreq      <= '0';
    if (nw_full = '0') then
      df_wrreq        <= '1';
      df_wrdatashort  <= X"00000000";
      sopeop          <= "01"; -- the end bit is asserted
      DF_STATE        <= S0;
    end if;
    --
    ---------------------------------------------------------------------------------------------
```

```
      when others =>
        DF_STATE <= S0;
      end case;
    end if;
  end if;
end process DFFSM;

--
    ----------------------------------------------------------------------------------------------------


--
    ----------------------------------------------------------------------------------------------------

-- FIFO instances
--
    ----------------------------------------------------------------------------------------------------

--
    ----------------------------------------------------------------------------------------------------

-- "DDR2 READ" (DR) FIFO
------------------------------------------------------------------

DDR2RDFIFO : entity common_mgwz_generated.fifo_2048x256
port map(
  aclr        => reset,
  clock       => clock,
  data        => dr_wrdata,
  rdreq       => dr_rdreq,
  wrreq       => dr_wrreq,
  almost_full => dr_afull,
  empty       => dr_empty,
  q           => dr_rdata,
  usedw       => dr_usedw
);

------------------------------------------------------------------------
--DATA FORMAT FIFO (df)
------------------------------------------------------------------------
DATAFORMATFIFO : entity common_mgwz_generated.fifo_16384x36  -- 4096 words will be
    checked
port map(
  aclr    => reset,
  wrclk   => clock,
  rdclk   => clock,
  data    => df_wrdata,
  rdreq   => df_rdreq,
  wrreq   => df_wrreq,
  q       => df_rdata,
  rdempty => df_empty,
  wrfull  => df_full,
  wrusedw => df_usedw
);
-------------------------------------------------------------------------------
-- TOTAL NUMBER OF WORD FIFO
----------------------------------------------------------------------------------
FIRSTWORDTOTALNUMBEROFWORDFIFO : entity common_mgwz_generated.fifo_32x36
port map
  (
  aclr    => reset,
  wrclk   => clock,
  rdclk   => clock,
  data    => nw_wrdata,
  rdreq   => nw_rdreq,
  wrreq   => nw_wrreq,
  rdempty => nw_empty,
    wrfull   => nw_full,
    q        => nw_rdata,
    wrusedw  => nw_usedw
);

------------------------------------------------------------------------------
-- DDR2 number of word to read in each time stamp requested FIFO
------------------------------------------------------------------------------
NUMBEROFWORDTOREADEACHTIMESTAMPFIFO: entity common_mgwz_generated.fifo_128x16_nw
port map
    (
    aclr                  => reset,
    clock                 => clock,
    data                  => nowtr_wrdata,
    rdreq        => nowtr_rdreq,
    wrreq                 => nowtr_wrreq,
    almost_full    => nowtr_afull,
    empty        => nowtr_empty,
    q                        => nowtr_rdata,
    usedw        => nowtr_usedw
);
------------------------------------------------------------------------------------
-- NOT EMPTY SLOT FIFO
---------------------------------------------------------------------------------
NOTEMPTYSLOTFIFO: entity common_mgwz_generated.fifo_16x6_nw
port map
    (
    aclr                  => reset,
    clock                 => clock,
    data                     => nes_wrdata,
    rdreq        => nes_rdreq,
    wrreq                 => nes_wrreq,
    full          => nes_full,
    empty        => nes_empty,
    q                        => nes_rdata,
    usedw        => nes_usedw
);

-----------------------------------------------------------------------------------
-- ERROR FIFO
----------------------------------------------------------------------------------
ERRORFIFO: entity common_mgwz_generated.fifo_32x32_nw
port map
    (
    aclr                  => reset,
    clock        => clock,
    data                  => error_wrdata,
    rdreq        => error_rdreq,
    wrreq        => error_wrreq,
    almost_full    => error_afull,
    empty                 => error_empty,
    q                        => error_rdata,
    usedw                 => error_usedw
);

-----------------------------------------------------------------------------------
-- ERROR HEADER FIFO
----------------------------------------------------------------------------------
ERRORHEADERFIFO: entity common_mgwz_generated.fifo_16x32_nw
port map
    (
    aclr                => reset,
    clock                 => clock,
    data                  => errorhead_wrdata,
    rdreq                 => errorhead_rdreq,
    wrreq        => errorhead_wrreq,
    empty                 => errorhead_empty,
    full                  => errorhead_full,
  q                          => errorhead_rdata ,
  usedw                  => errorhead_usedw
 );


 ----------------------------------------------------------
 PARITY_GEN : entity common_tel62_lib.parity_gen
 ----------------------------------------------------------
 port map (
   data   => dataparityin ,
   parity => dataparityout
 );
 ----------------------------------------------------------
 PARITY_GEN_2 : entity common_tel62_lib.parity_gen
 ----------------------------------------------------------
 port map (
   data   => dataparityinnw ,
   parity => dataparityoutnw
 );

 ----------------------------------------------------------------
 LOG_INT : entity common_tel62_lib.logger_interface
 ----------------------------------------------------------------
 port map(
   clk          => clock ,
   reset        => reset ,
   enable       => log_permit ,
   logd         => logd ,
   logn         => logn ,
   logit        => logit ,
   logbusy      => logbusy ,
   log_data     => log_data ,
   log_valid    => log_valid ,
   log_more     => log_more ,
   log_ack      => log_ack
 );
end architecture std;
```

# Bibliography

[1] General-purpose computing on graphics processing units.

[2] NaNet overview. `http://apegate.roma1.infn.it/mediawiki/index.php/NaNet_overview`.

[3] A.Antonelli et al. The na62 lav front-end electronics. *JINST*, 7:C01097, 2012.

[4] M Abolins et al. Integration of the trigger and data acquisition systems in atlas. *Journal of Physics: Conference Series*, 119, 2008.

[5] A.Ceccucci et al. The na62 liquid krypton calorimeter's new readout system. *JINST*, 9(01):C01047, 2014.

[6] Altera®. Cyclone iii device handbook.

[7] Altera®. De4 board user manual.

[8] Altera®. Quartus ii software.

[9] Altera®. Serial configuration (epcs) devices datasheet.

[10] Altera®. Stratix iii device handbook.

[11] Altera®. Stratix iv device handbook.

[12] G. Anelli et al. The totem electronics system. *2008 JINST 3 S08007*, pages 205–210, 2007.

[13] N. Azorskiy et al. The na62 spectrometer acquisition system. *Journal of Instrumentation*, 11(02):C02064, 2016.

[14] B.Angelucci et al. Tel62: an integrated trigger and data acquisition board. In *Proceedings, 2011 IEEE Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC 2011)*, pages 823–826, 2011.

[15] S. Baron. *Timing, Trigger and Control (TTC) Systems for the LHC.* Organization, CERN, 2013.

[16] J. Christiansen. Hptdc, high performance time to digital converter, 2004.

[17] G. Collazuol et al. Fast online triggering in high-energy physics experiments using gpus. *Nucl. Instrum. Meth.*, A662:49–54, 2012.

[18] DESY. *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014).* DESY, 2015.

[19] D Emeliyanov and J Howard. Gpu-based tracking algorithms for the atlas high-level trigger. *Journal of Physics: Conference Series*, 396(1):012018, 2012.

[20] E.Pedreschi et al. A high-resolution tdc-based board for a fully digital trigger and data acquisition system in the na62 experiment at cern. *IEEE Trans. Nucl. Sci.*, 62(3):1050–1055, 2015.

[21] J. Christiansen et al. Ttcrx reference manual, 2004.

[22] F.Anghinolfi et al. Nino: An ultra-fast and low-power front-end amplifier/discriminator asic designed for the multigap resistive plate chamber. *Nucl. Instrum. Meth.*, A533:183–187, 2004.

[23] F.Fontanelli et al. Cc-pc gluecard application and user's guide, 2003.

[24] F.Spinella et al. The tel62: A real-time board for the na62 trigger and data acquisition. data flow and firmware design. In *Proceedings, 19th Real Time Conference (RT2014)*, 2014.

[25] G.Haefeli et al. The lhcb daq interface board tell1. *Nucl. Instrum. Meth.*, A560:494–502, 2006.

[26] G.Lamanna et al. Gpus for fast triggering and pattern matching at the cern experiment na62. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 628(1):457 – 460, 2011.

[27] Mentor Graphics®. Hdl designer.

[28] Mentor Graphics®. Modelsim®.

[29] B. Green et al. Atlas trigger/daq robin prototype. *IEEE Trans. Nucl. Sci.*, 51:465–469, 2004.

[30] TDAQ Working group. NA62 data formats. `https://twiki.cern.ch/twiki/pub/NA62/TdaqSystem/DataFormats.pdf`.

[31] The NA62 Pisa group. Tdcb documentation, 2016.

[32] F Hahn, F Ambrosino, A Ceccucci, H Danielsson, N Doble, F Fantechi, A Kluge, C Lazzeroni, M Lenti, G Ruggiero, M Sozzi, P Valente, and R Wanke. NA62: Technical Design Document. Technical Report NA62-10-07, CERN, Geneva, December 2010.

[33] H.Atherton et al. *Precise measurements of particle production by 400 GeV/c protons on beryllium targets*. CERN, 1980.

[34] Andreas Herten. GPU-based Online Tracking for the PANDA Experiment. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014)*, pages 57–63, 2015.

[35] H.Muller et al. Quad gigabit ethernet plug-in card.

[36] P. Jovanovic. Local trigger unit preliminary design review.

[37] A. Kluge, G. Aglieri Rinella, S. Bonacini, P. Jarron, J. Kaplon, M. Morel, M. Noy, L. Perktold, and K. Poltorak. The {TDCpix} readout asic: A 75 ps resolution timing front-end for the {NA62} gigatracker hybrid pixel detector. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 732:511 – 514, 2013. Vienna Conference on Instrumentation 2013.

[38] LHCb. Credit-card pcs as ecs interface.

[39] M.Bauce et al. The gap project: Gpu applications for high level trigger and medical imaging. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014)*, pages 3–8, 2015.

[40] M.Bauce et al. Use of hardware accelerators for atlas computing. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014)*, pages 48–54, 2015.

[41] S. Minutoli et al. The electronics system of the totem t1 telescope. *Nucl. Instrum. Meth.*, A718(3):223–225, 2013.

[42] D. Moraes, W. Bonivento, Nicolas Pelloux, and W. Riegler. The CARIOCA Front End Chip for the LHCb muon chambers. 2003.

[43] P. Moreira. Qpll project, 2005.

[44] P.Lichard et al. Performance evaluation of multiple (32 channels) sub-nanosecond tdc implemented in low-cost fpga. *JINST*, 9:C03013, 2014.

[45] S.Gallorini. Track pattern-recognition on gpgpus in the lhcb experiment. In *Proceedings, GPU Computing in High-Energy Physics (GPUHEP2014)*, pages 38–43, 2015.

[46] B. G. Taylor. Ttc laser transmitter (ttcex, ttctx, ttcmx) user manual.

[47] J. Vermeulen et al. Atlas data flow: The read-out subsystem, results from trigger and data-acquisition system testbed studies and from modeling. *IEEE Trans. Nucl. Sci.*, 53(3):912–917, 2006.

[48] J. Vermeulen et al. The lhcb daq interface board tell1. *Nucl. Instrum. Meth.*, A560:494–502, 2006.