TECHNISCHE UNIVERSITÄT DRESDEN

Development of a Digital Processing Test System for the Phase-I Upgrade of the ATLAS Liquid Argon Calorimeter

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

vorgelegt von

YVES BIANGA

geboren am 03.02.1984 in Gera

der

TECHNISCHEN UNIVERSITÄT DRESDEN
INSTITUT FÜR KERN- UND TEILCHENPHYSIK
FACHRICHTUNG PHYSIK
FAKULTÄT MATHEMATIK UND NATURWISSENSCHAFTEN

2018

Eingereicht am 13. August 2018

1. Gutachter: Prof. Dr. Arno Straessner
2. Gutachter: Prof. Dr. Kai Zuber

Kurzdarstellung

Das European Organization for Nuclear Research (CERN) ist einer der wichtigsten Orte für die Erforschung der Teilchenphysik. Eine Kaskade von Ringbeschleunigern bildet die Basis und liegt den meisten Experimenten zu Grunde. Der größte Beschleunigerring, der LHC, schafft es Teilchen bis zu einer Energie von 7 TeV zu beschleunigen und diese an definierten Kreuzungspunkten kollidieren zu lassen. Der ATLAS Detektor, welcher sich in einem dieser Punkte befindet, ist aus verschiedenen Subdetektoren aufgebaut. Eines davon stellt das Flüssigargon-Kalorimeter (LAr) dar. Die Hauptaufgabe dieses Kalorimeters ist das Vermessen der deponierten Energie von Teilchen. Aufgrund der hohen Kollisionsrate wird eine Vorauswahl über ein mehrstufiges Triggersystem in Echtzeit realisiert.

Um die Luminosität des Beschleunigers weiter zu steigern, ist ab dem Jahr 2019 ein Upgrade des LHC und seiner Detektoren geplant. Die damit steigende Kollisionsrate stellt eine große Herausforderung an das bestehende Triggersystem des Flüssigargon-Kalorimeters dar. Eine verbesserte Auslese- und Triggerelektronik soll die Granularität und die Verarbeitungskapazität des Triggers steigern.

In dieser Masterarbeit werden zwei Testsysteme für die erneuerte erste Triggerstufe des LAr Detektors entwickelt. Durch den fortschreitenden Prozess der Firmwareentwicklung, stellen diese Testsysteme zwei entwicklungsbegleitende Werkzeuge dar. Als erstes wird eine Möglichkeit geschaffen die programmierte Firmware effizienter und dynamischer testen zu können. Im zweiten Teil wird eine Anbindung entwickelt, welche es erlaubt, simulierte Kollisionsdaten des ATLAS in die Firmware einzuspeisen und somit eine realitätsnahe Testumgebung zu schaffen.

Abstract

The CERN is one of the today's most important places for doing research in particle physics. A cascade of ring accelerators forms the basis of the most experiments. The largest accelerator ring, the LHC, manages to accelerate particles up to an energy of 7 TeV and collide them at defined crossing points. The ATLAS detector, which is located in one of these points, is made up of different subdetectors. One of them is the Liquid Argon Calorimeter (LAr). The main task of this calorimeter is to measure the deposited energy of particles. Due to the high collision rate, a preselection is realized in real time via a multi-level trigger system.

In order to increase the luminosity of the accelerator, an upgrade of the LHC and its detectors is scheduled for 2019. The resulting collision rate poses a great challenge to the existing trigger system of the LAr. An improved detector readout and trigger electronics is intended to increase the granularity and processing capacity of the trigger system.

In this master thesis two test systems are developed for the renewed first trigger stage of the LAr detector. Due to the ongoing process of firmware development, these test systems represent two development-accompanying tools. As a first step, a way is created to be able to test the programmed firmware more efficiently and dynamically. In a second step, a connection is developed which allows to feed simulated collision data of the ATLAS into the firmware. This creates a realistic test environment.

Contents

1 Introduction

Exploring the unknown has always been a driving force of humanity and is a mirror of our curiosity. This also applies to particle physics, a relatively new field of physics. Even though the first ideas about the construction of the world have been handed down since antiquity, the foundations of our present view were discovered only at the beginning of the 20th century. Since the introduction of quantum physics and the theory of relativity, all areas of physics are experiencing a renaissance. Particle physics plays a special role due to its extreme parameter ranges. So it goes from the description of the most elementary particles to huge structures such as stars, which we could not understand without particle physics. Likewise, time is closely interwoven with this discipline, ranging from the creation of the universe to the prediction of particle properties.

At the present time, the Standard Model (SM) is one of the most accurate theories of physics. It shows a summary of previously known elemantar particles and differentiates between three types. The hadrons forming quarks, the leptons and the exchange bosons for the interactions. Although the SM already covers a large range of known particle processes, there are still open questions and extensions which need to be checked. For example, models such as Quantum Electrodynamics (QED) or Quantum Chromodynamics (QCD) anticipate massless particles. To solve this discrepancy, a scalar field was postulated in 1964 by Peter Higgs, which was to be found at CERN in 2012.

CERN is the world's largest association of particle physics with more than 10,000 scientists involved and more than 100 participating countries. The largest accelerator ring Large Hadron Collider (LHC) has several crossing points at which proton-proton collisions can be generated. At one of these points the A Toroidal LHC ApparatuS (ATLAS) detector is set up. It is a general purpose particle detector and therefore designed for different research tasks.

As part of the Phase-I upgrade of the LHC, starting in 2019, it is planned to increase the luminosity of the LHC to improve the event rate and by that the statistics of the results. Therefore it is also necessary to adapt the detectors to the increased requirements. The present master thesis refers to the Phase-I upgrade plans of the Liquid Argon Calorimeter of ATLAS. An overview of the current state of development is shown and the development of two test systems is described.

2 Physical Foundations

This chapter gives a summary of the theoretical fundations which are required for this thesis. Section 2.1 describes the fundamental physics by showing the concept of the SM of particle physics. A briefly summary of collider physics can be found in section 2.2.

2.1 The Standard Model

The basis of today's particle physics is the so-called Standard Model which is one of the best approved physical theory. The SM of particle physics is a compilation of essential theories to describe the fundamental structure of the most materials in the universe and the important interactions between them.

It shows a summary of previously known elemantary particles and distinguishes between three types. The hadrons forming quarks, the leptons and the force mediators for interactions between them.

There are six types of quarks named by the flavours: up, down, strange, charm, top, and bottom. All quarks are spin $\frac{1}{2}$ particles and can therefore be classified as fermions. The electrical charge of quarks can be $-\frac{1}{3}$ or $+\frac{2}{3}$ of the elementary charge. To participate in the strong interactions, all quarks also carry an color charge. Up and down quarks carry the lowest masses of all quarks. Through a process of particle decay the heavier quarks rapidly change into up and down quarks. Therefore, up and down quarks are generally stable and the most common in the universe. The four heavier quarks - strange, charm, bottom and top - can only be produced in high energy processes. Collisions in a particle accelerators is one of this processes.

In total, there are also six leptons that can interact with the weak force and the gravitation. If they carry an electric charge, they also interact through the electromagnetic interaction. The electron, myon and tauon are electrically charged by -1 the three associated neutrinos are electrically neutral. As well as the quarks all leptons are fermions and have a spin $\frac{1}{2}$. As shown in picture 2.1, the fermions divided into three generations, the first being the best known.

The third type of elementary particles are bosons which follows the Bose-Einstein statistic. By that the spin has to be an integer. All four gauge bosons - gluon, photon, W- and Z-Boson - are spin 1 particles, the higgs is spin 0. The photon is the best known gauge boson. It mediates the electromagnetic interaction. The other gauge bosons of the Standard Model are the eight gluons of the strong interaction as well as the W bosons and Z bosons of weak nuclear force. The gauge invariance condition requires that all gauge bosons are massless. However, the W and Z bosons have a mass. This is an effect of the Higgs mechanism, which spontaneously breaks the SU (2) × U (1) symmetry of the electroweak interaction. The higgs boson was the last experimentally confirmed particle of the SM of elementary particle physics. It was found in 2012 at CERN.

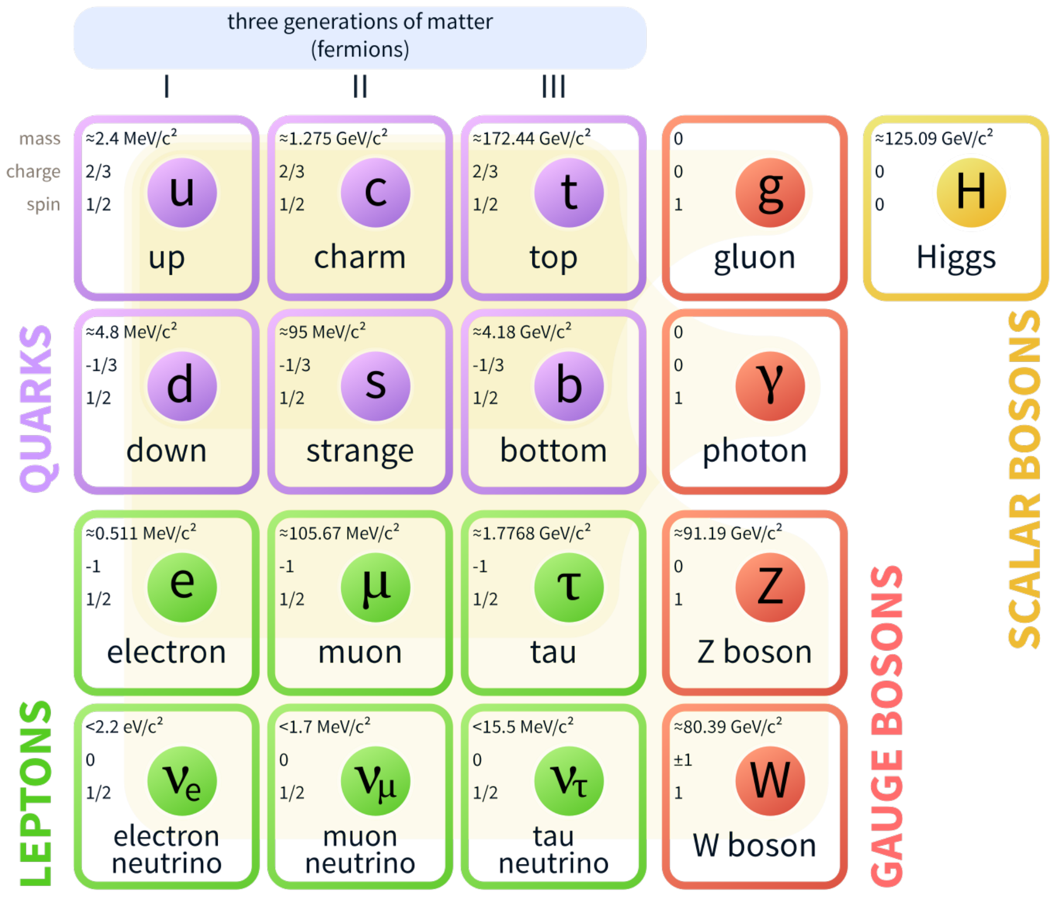

three generations of matter (fermions)
I
II
III
mass
charge
spin
≈2.4 MeV/c² 2/3 1/2 u up
≈1.275 GeV/c² 2/3 1/2 c charm
≈172.44 GeV/c² 2/3 1/2 t top
0 0 1 g gluon
≈125.09 GeV/c² 0 0 H Higgs
≈4.8 MeV/c² -1/3 1/2 d down
≈95 MeV/c² -1/3 1/2 s strange
≈4.18 GeV/c² -1/3 1/2 b bottom
0 0 1 γ photon
≈0.511 MeV/c² -1 1/2 e electron
≈105.67 MeV/c² -1 1/2 μ muon
≈1.7768 GeV/c² -1 1/2 τ tau
≈91.19 GeV/c² 0 1 Z Z boson
<2.2 eV/c² 0 1/2 νe electron neutrino
<1.7 MeV/c² 0 1/2 νμ muon neutrino
<15.5 MeV/c² 0 1/2 ντ tau neutrino
≈80.39 GeV/c² ±1 1 W W boson
QUARKS
LEPTONS
GAUGE BOSONS
SCALAR BOSONS

Figure 2.1: The Standard Model of elementary particles, all known fermions and bosons are systematically ordered according to their properties. [2]

2.2 Collider Physics

One possibility to verify the SM and investigate further physics can be done with a partical collider. We can distinguish between two common collider designs, a linear and a ring collider. Any collider can be characterized by some typically quantities.

The luminosity $\mathscr{L}$ is one typical performance feature of particle collision experiments. As shown in figure 2.1 it indicates how many particles collide per time and area. As shown in equation 2.1 he luminosity is defined about the event rate $\dot{N}$ and the effective cross section σ.

$$\mathscr{L} = \frac{\dot{N}}{\sigma} \tag{2.1}$$

The cross section σ is a measure of the probability that a particular process such as absorption, scattering or a reaction takes place. This is described in the equation 2.2.

$$\sigma = \frac{(\text{Number of events}) \cdot A}{N_A N_B} \tag{2.2}$$

In collisions of two particles with masses m_1 and m_2 and momentums p_1 and p_2 the total energy squared in the center-of-momentum frame ($\vec{p}_1 + \vec{p}_2 = 0$) can be expressed as shown in equation 2.3.

$$s = (p_1 + p_2)^2 = (E_1 + E_2)^2 \tag{2.3}$$

In high energy collisions the beam particles are ultra-relativistic and the momentums are much larger than their masses. The total center-of-momentum energy can thus be approximated as shown in equation 2.4.

$$E_{CM} = \sqrt{s} \approx 2 \cdot E_1 \approx 2 \cdot E_2 \tag{2.4}$$

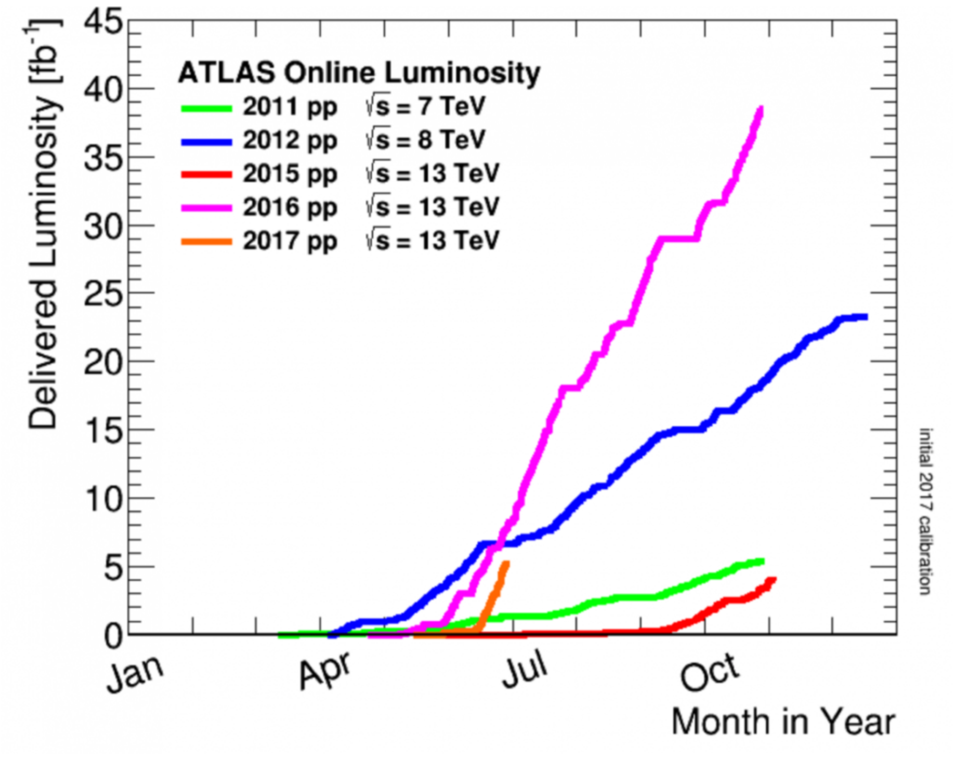


Figure 2.2: The integrated luminosity as a function of time for the various years of LHC operation until 2017. [7]

3 The Experiment

The following chapter explains the necessary basics of the underlying experiment to understand how this work fits into it.

The CERN is one of the today's most important places for doing research in particle physics. Over 10,000 scientists from over 100 countries are associated in this federation. The CERN consists of several accelerator rings with different sizes and so too several detectors which belong to different experiments. Probably the most famous part of the CERN is the LHC accelerator with its seven associated detectors. The biggest two for general purpose are A Toroidal LHC ApparatuS and Compact Muon Solenoid (CMS).

3.1 Large Hardon Collider

The LHC is the largest and most powerful particle collider in the world and was put into operation in 2008. It was build to create the capabilities for testing different theories of particle physics. The main aims are to study where the mass comes from and searching for new particles predicted by supersymmetic theories. Furthermore the study of extra dimenions or generally the exploration of signs from new physics are important tasks.

To manage these tasks the LHC is designed as a synchrotron with a circumference of 26.7 km and can reach high energies up to 14 TeV. A few smaller particle accelerators from older experiments at CERN are used as preaccelerators. This accelerator chain can be used to accelerate protons or heavy ions. The LHC has four crossing points where the ATLAS detector is located in one of them. A fully filled LHC contains up to 2,808 bunches each with 115 billion protons in it. With a bunch collision rate of 40 MHz the LHC can reach a luminosity of 1034 $^{1}/_{\mathrm{cm}^2\mathrm{s}}$. The figure 3.1 shows the whole accelerator chain which is necassary to run the LHC.

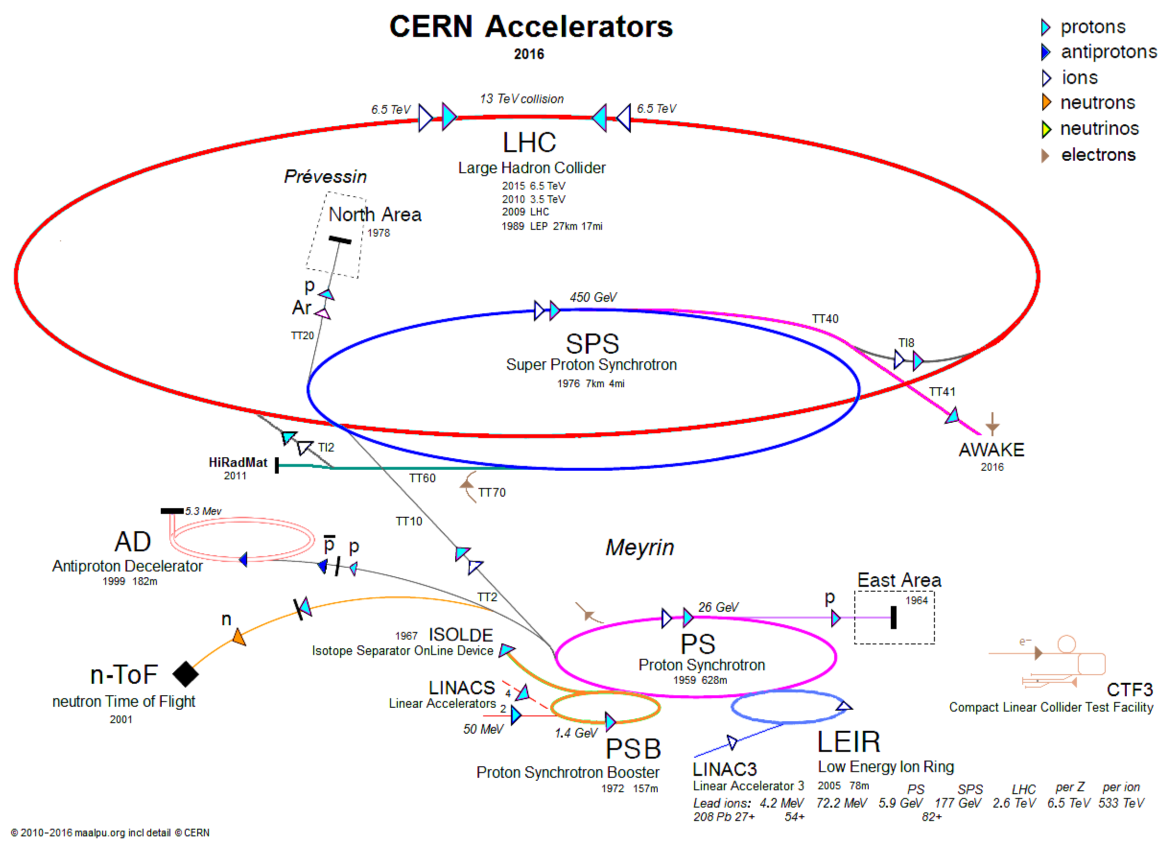


Figure 3.1: All particle accelerators with their compounds available at CERN. [1]

3.2 ATLAS Detector

With the dimension of 44 m in length and 25 m in diameter the ATLAS detector is the biggest particle detector at the LHC. Due to its weight of over 7 t, it is the second heaviest detector at CERN. It is designed as a detector for general-purepose and therefore it has the ability to discover a wide spectrum of physical processes. One of the main aims are the discovery of the Higgs boson and furthermore to discover particle physics beyond the standard model. Particularly it can detect most of the known particles and can measure their masses, momentums, energies, lifetimes, charges and also the their nuclear spins.

To achieve this big scope it is designed in layers made up of detectors and magnets of different types. Figure 3.2 shows the ATLAS detector with it's different layers. The Inner Detector is located close to the particle crossing point and is surrounded by the Solenoid Magnet. Together with the impact of this magnetic field the Inner Detector is responsible to track charged particles from which the particle types and momentums can be determined.

To get the energies of particles the Inner Detector is covered by two different kinds of calorimeters as the next layers. The first is an inner electromagnetic calorimeter and the second an outer hadronic calorimeter. Both kinds are designed as a sampling calorimeter which means to collect energy information by absorbing it. Structurally, these two are splitted slightly differently. The e.m. calorimeter and the endcaps of the hardonic calorimeter were constructed as a LAr calorimeter in contrast to the middle part of the hardonic calorimeter - the Tile calorimeter - which uses iron plates as absorber and plastic scintillating tiles.
As shown in figure 3.2 the outermost detector layer is the Moun Spectrometer. It consists of a magnetic field provided by three toroidal magnets and over 1200 muon tracking and trigger chambers.

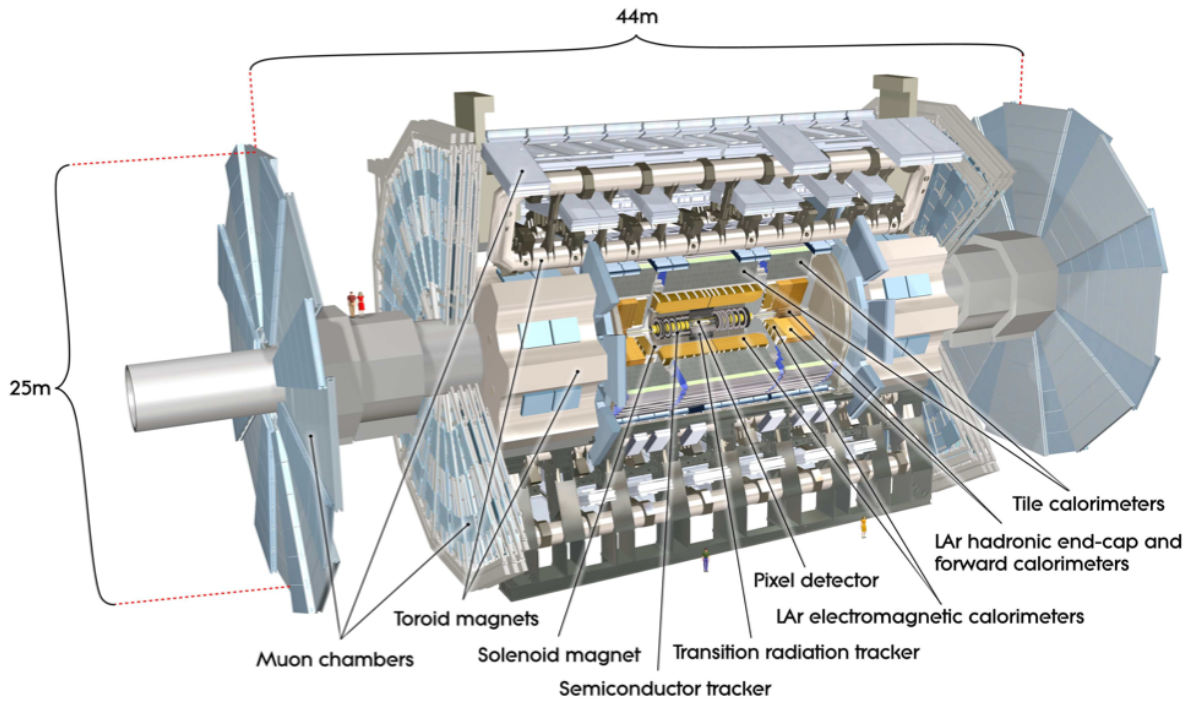

44m
25m
Tile calorimeters
LAr hadronic end-cap and forward calorimeters
Pixel detector
LAr electromagnetic calorimeters
Toroid magnets
Muon chambers
Solenoid magnet
Transition radiation tracker
Semiconductor tracker

Figure 3.2: The ATLAS detector with it's different layers. [3]

A parameter, called pseudorapidity and denoted by η frequently used in colliding beam experiments to express angles with respect to the axis of the colliding beams. It has the value 0 for particle trajectories that are perpendicular to the beam and positive or negative values for those at an angle to the beam. The following equation 3.1 shows it's definition.

$$\eta \equiv -\ln\left[\tan\left(\frac{\theta}{2}\right)\right] = \operatorname{artanh}\left(\frac{p_L}{|\mathbf{p}|}\right) \tag{3.1}$$

3.3 Liquid Argon Calorimeter

The aim of a calorimeter is to measure energies of charged particle and photons. For this, the calorimeter completely stops incoming particles and determines the released energy. The advent of high-energy particles in a calorimeter produces secondary particles, which themselves generate additional particles until the available energy is exhausted. This cascade of secondary particle production is called particle showering.

There are two different types of calorimeters, based on different physical mechanisms. An electromagnetic calorimeter is used to determine the energy of particles that interact essentially via the electromagnetic force. These are electrons and positrons as well as gamma particles and also muons. The mainly used electromagnetic mechanism is an alternating of emitting high energy radiation, called Bremsstrahlung followed by the production of electron-positron pairs. Also the Compton scattering and the photoelectric effect are taking place in electromagnetic calorimeters. In a hadronic calorimeter particles which are mainly subject to the interaction of the strong forces can be detected.

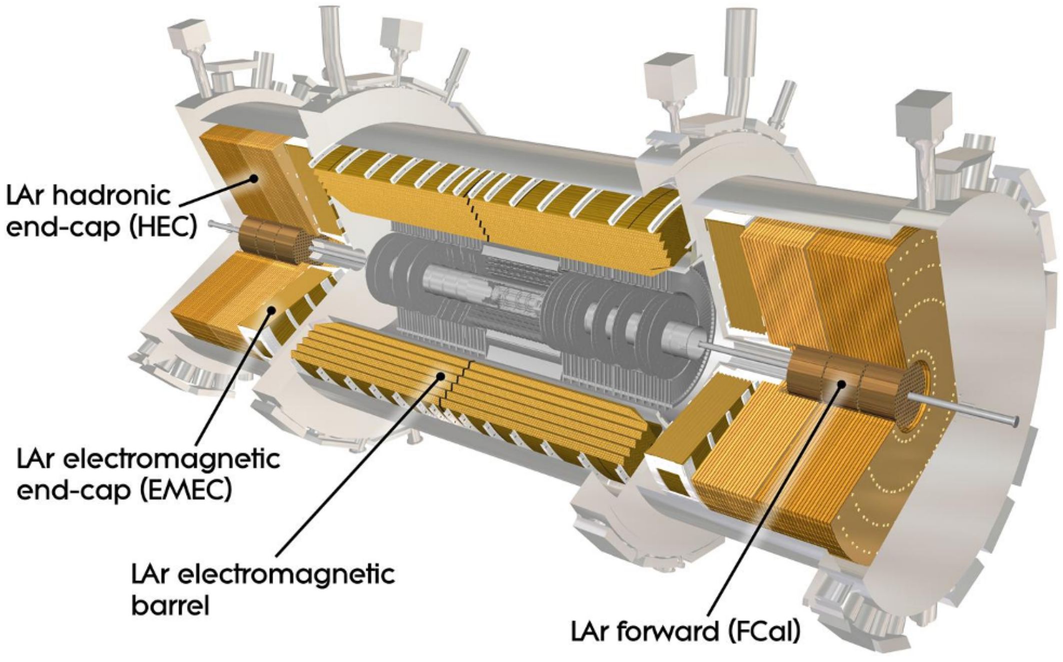

LAr hadronic end-cap (HEC)
LAr electromagnetic end-cap (EMEC)
LAr electromagnetic barrel
LAr forward (FCal)

Figure 3.3: Schematic of the Liquid Argon Calorimeter of ATLAS. [4]

As shown in figure 3.3 the LAr can be strucural divided into four parts. The Electro-Magnetic Barrel (EMB) and the Electro-Magnetic End-Cap (EMEC) make up the electromagnetic calorimeter. The Forward Calorimeter (Fcal) and the Hadronic End-Cap (HEC) belongs to the hadronic calorimeter. As in the Liquid Argon Calorimeter, calorimeters are often built up in layers. In this case, sensitive detection layers alternate with insensitive layers, which serve only for the loss of energy. In these dense absorbers, the interaction processes of the particles take place, which then generate subparticles like electrons, photons, nuclear fragments and hadrons. Afterwards these subparticles are interact with the sensitive layer and can be measured. For the LAr the absorbing layer consists of lead and the active layer is made of liquid Argon. At the border area of the Argon chambers, high voltage electrodes are arranged. If high energy particles passes this chambers the Argon will be ionized and generate an detecalbe electric current at the electrodes. This results in a triangular shaped current pulse as shown in figure 3.4.

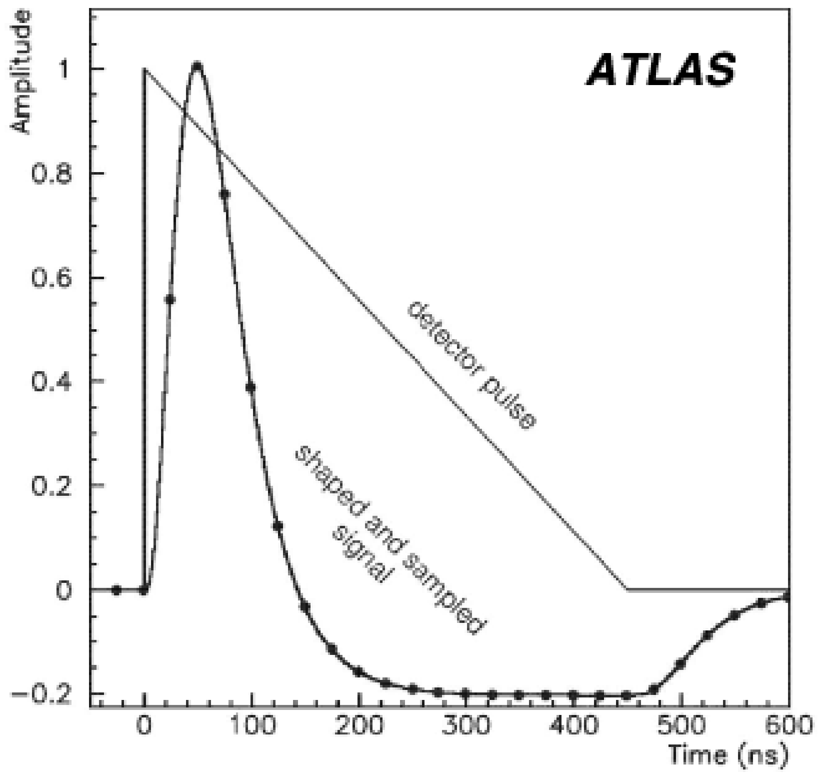


Figure 3.4: The raw and shaped detector signal of the LAr detector cells. [4]

An example setup for $\eta = 0$ of the individual detector planes of LAr-EMB is shown in the figure 3.5. Typically, four successive layers exist. The first is the presampler, followed by the front, middle and back layers. The cross-section of the single cells depends on the layer as seen in figure 3.5. Furthermore, the cross-sectional area changes depending on η and ϕ and thus on the position in the detector.

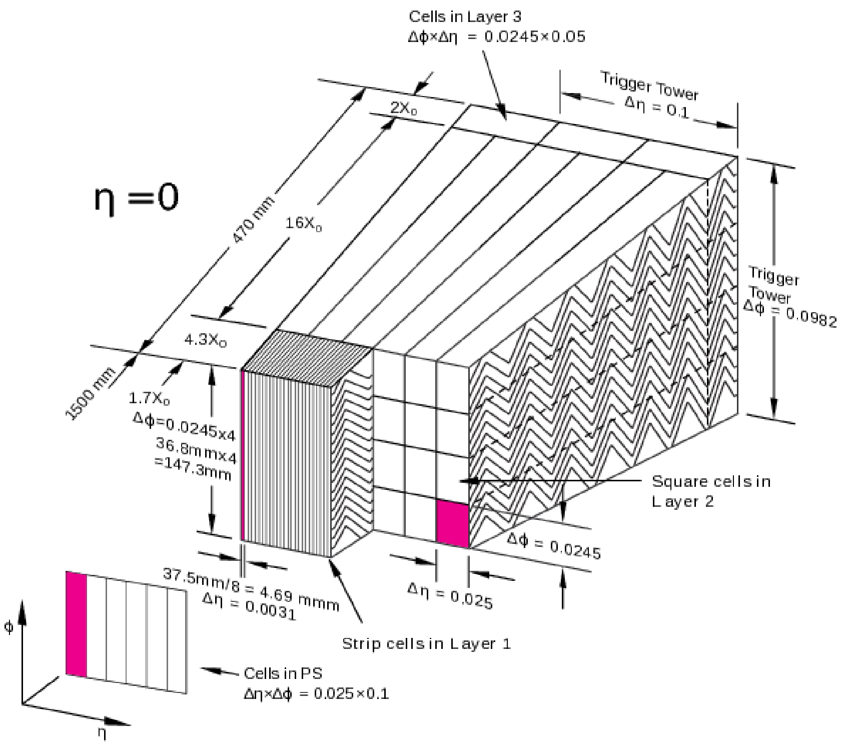


Figure 3.5: Sketch of an EMB module where the different layers are visible. The granularity in eta and phi of the cells of each of the three layers and of the trigger towers is also shown. [14]

3.4 Trigger System

As already mentioned in chapter 3.1 the LHC carries protons which are bunched together into up to 2808 bunches. Each of these bunches contains about 115 billion protons. With a Bunch Crossing (BC) frequency of 40 MHz the ATLAS has a collition rate of nearly 100 MHz. At the highest LHC energies we can expect about 23 inelastic scatterings and 1700 carged particles for each collision at ATLAS. This is equivalent to 1.5 MB per collision or 60 TB/s of raw detector data. The reduction of the data flow to manageable levels is done by a specialized multi-level computing system called the ATLAS Trigger System. The primary goal is to select interessting events to a number which is suitable for the Data Acquisition System (DAQ) with a given storage bandwith of 200 events/s.

In figure 3.6 we can find an overview of the production rates of different physical signatures and can estimate how accurate and fast the Trigger System has to work to find and select the 200 most insteresting events every second.

The ATLAS Trigger System is built in three different levels. Each stage operates at its own processing speed and has a fixed time window to pass the results. In the figure 3.7 the complete trigger chain from the raw detector data to the readout is shown schematically. The LAr Trigger prOcessing MEzzanine (LATOME) project detailed in the next chapter 4 can be assigned to the calorimeter trigger of the Level 1 trigger and corresponds to the figure 3.7 in the upper middle range. It is thus the first trigger stage which performs a processing of the digitized detector signals of LAr.

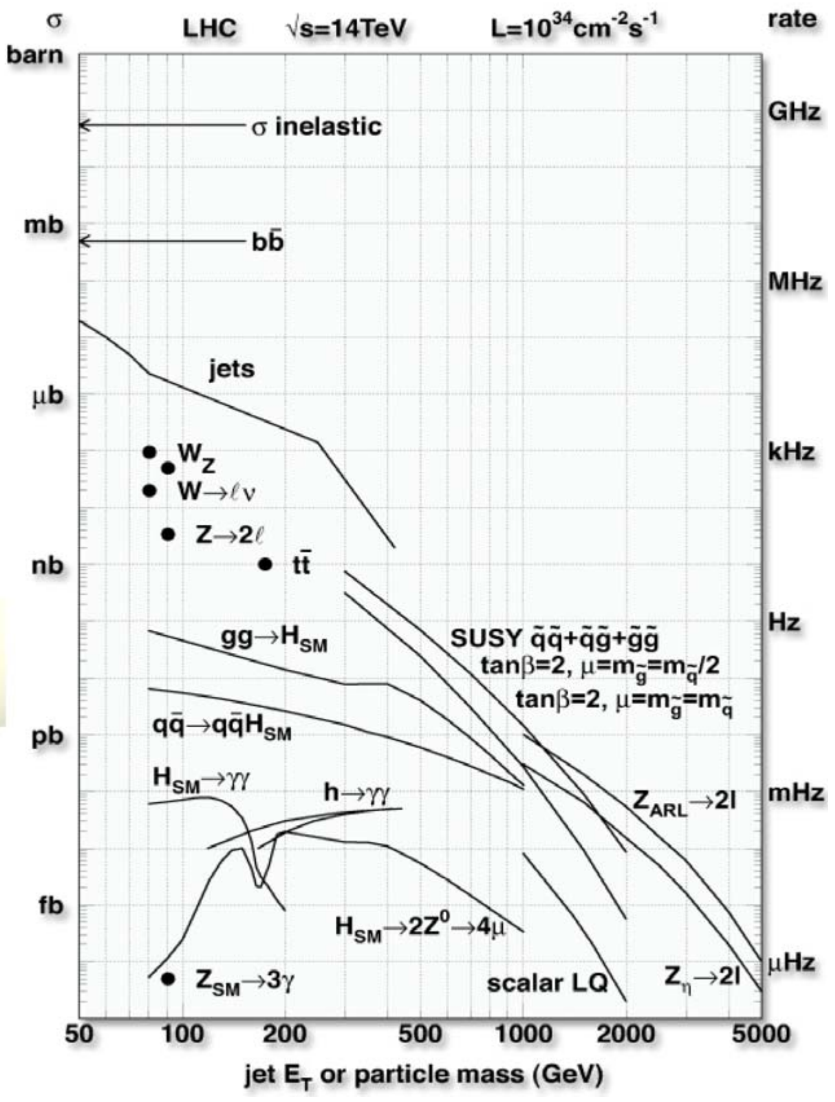


Figure 3.6: Expected event rates of searched physic-signatures [5]

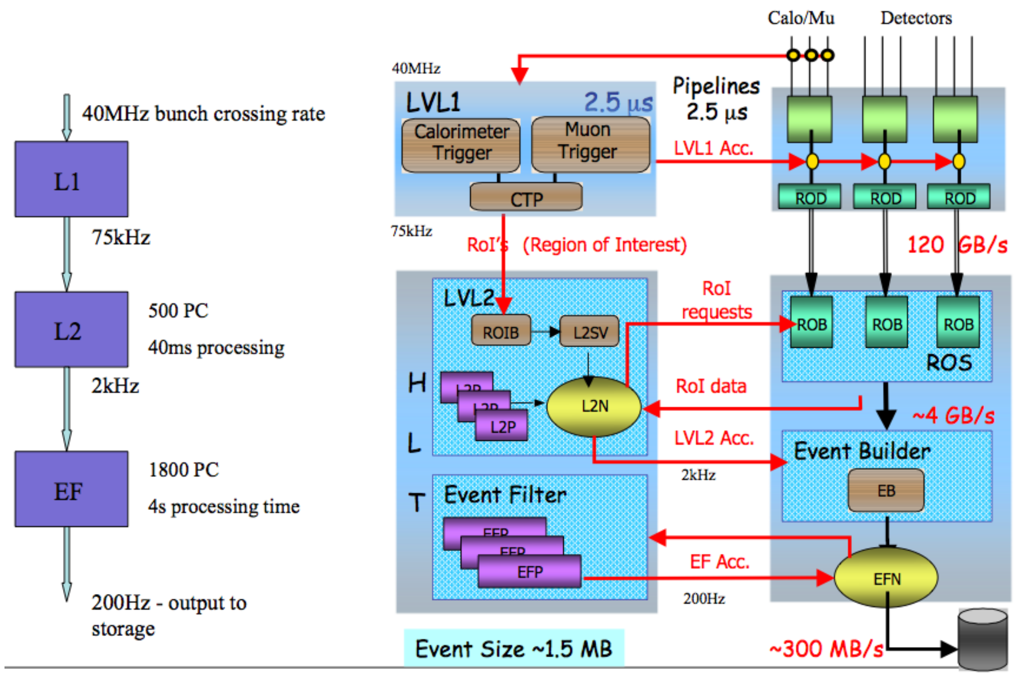


Figure 3.7: Schematic overview and the mode of operation of the ATLAS Trigger System [6]

3.5 Phase-I Upgrade

The LHC is designed to reach a luminosity of $\mathscr{L} = 10^{34}\,\mathrm{cm}^{-2}\mathrm{s}^{-1}$ at a center of mass energy of 14 TeV. Since the beginning at 2009 the LHC approach this design-goal through dedicated upgrades. The upgrades will take place during long shutdowns and are foreseen in the schedule of the LHC. A temporal overview of the upgrade timeline is given in figure 3.8. This overview contains three time ranges marked as Long Shutdown (LS) for the upgrades and the time ranges between them for data taking. The next upcoming upgrade is planned in LS2 and is called Phase-I Upgrade.

The Phase-I Upgrade at LHC aims to decrease the bunch spacing from currently 50 ns to 25 ns and increase the target luminosity $\mathscr{L}$ to $2 \cdot 10^{34}\,\mathrm{cm}^{-2}\mathrm{s}^{-1}$. With this the numbers of interaction per bunch crossing μ will raise from 25 to 60. To be able to process the increased detector activity for every Bunch Crossing, the quality of the trigger selections must rise.

Since it was decided that the ATLAS detector will not be dismantled for this upgrade, the physical hardware of the LAr will be retained. So its physical function of determining the deposited energy by charging Argon ions thus remains but the cell clustering and read-out of the trigger cells and so too the trigger processing will be changed.

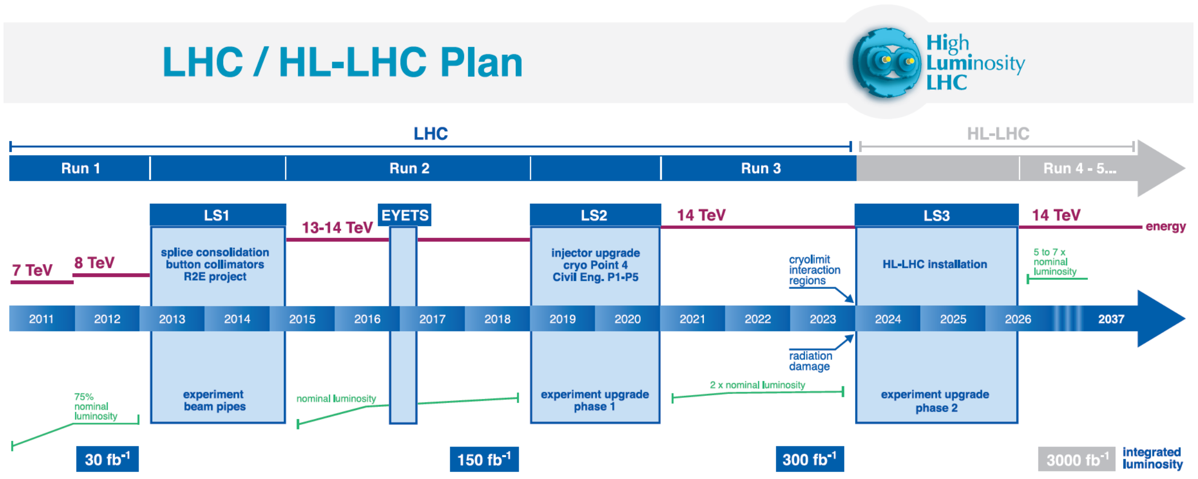


Figure 3.8: Timeline of LHC upgrades and data taking [8]

Up to now a LAr trigger cells, called Trigger Tower (TT), for the Level 1 trigger contains up to 60 single LAr cells. The Phase-I upgrade splits each Trigger Tower into 10 Super Cell (SC) which increase the granularity of the LAr trigger. An example of this new clustering can be found in figure 3.9. The left side shows a Trigger Tower in a segment of $\Delta\eta \times \Delta\phi = 0.1 \times 0.1$ hit by a particle. The same event clustered by SCs can be found on the right side.

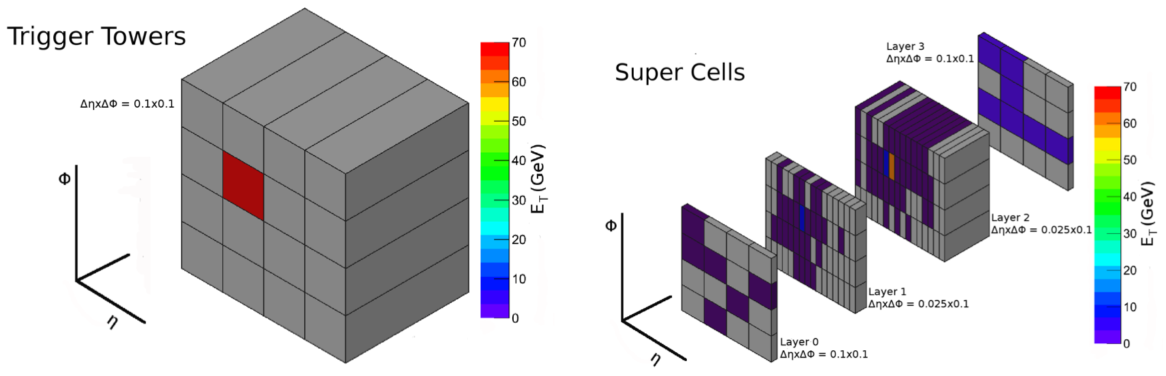


Figure 3.9: Granularity of Trigger Towers (left) and Super Cells of the LAr-EMB (right). [4]

To manage the increased amount of trigger data also the read-out and the processing have to be upgraded. The read-out will be done by a new hardware instance called LAr Trigger Digitizer Board (LTDB) as shown in figure 3.10. These boards belongs to the front-end electronic near the detector and are responsible for digitizing and encoding of the SC data. The first processing of the digitized data is done by the LAr Digital Processing System (LDPS) and take place in the USA15 cavern.

The global task of the LDPS is to calculate the transverse energies deposited in each SC and forward these data to the Trigger Feature Extractors (FEX). To read all 34,000 SCs of the LAr detector every 25 ns, the datarate to be processed is roughly 25 Tbit/s. In order to accomplish such a heavy task, the LDPS is based on 124 high performance Field Programmable Gate Array (FPGA) Advanced Mezzanine Card (AMC) cards.

Figure 3.10 shows the current path of Trigger Tower-data and the upgraded data-path for SC processing.

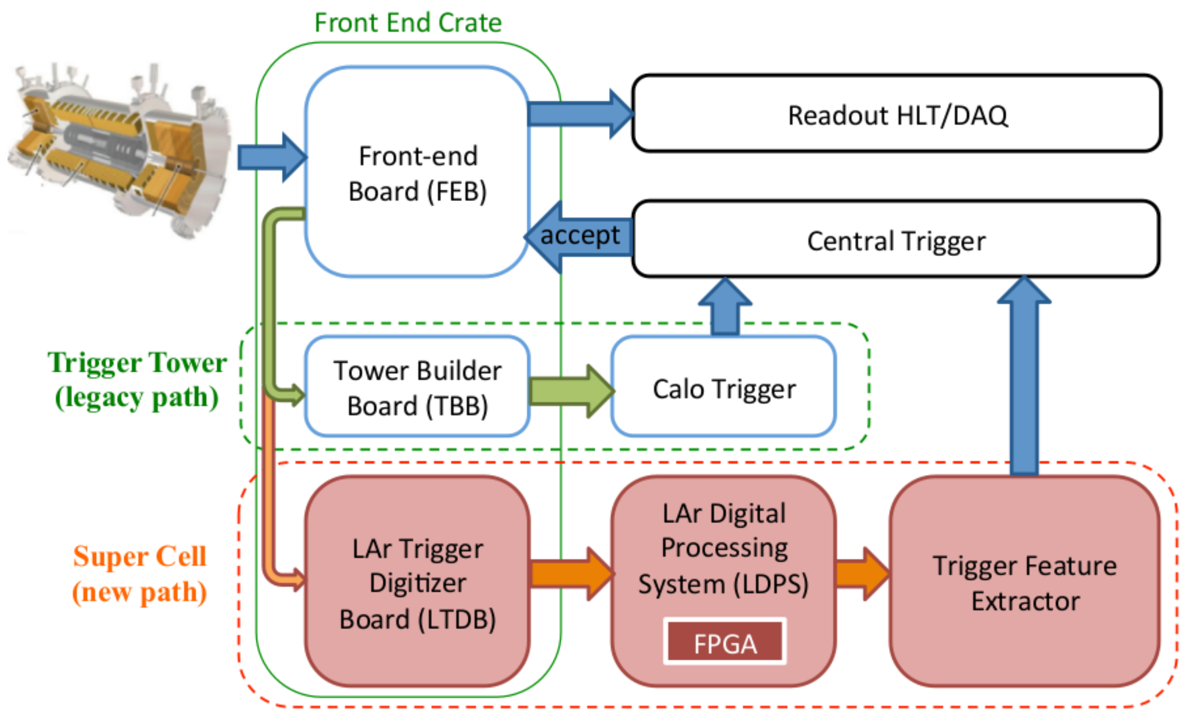

Front End Crate
Front-end Board (FEB)
Readout HLT/DAQ
accept
Central Trigger
Trigger Tower (legacy path)
Tower Builder Board (TBB)
Calo Trigger
Super Cell (new path)
LAr Trigger Digitizer Board (LTDB)
LAr Digital Processing System (LDPS)
FPGA
Trigger Feature Extractor

Figure 3.10: Schematic representation of the trigger paths for TTs and SCs [16]

4 The LATOME Project

4.1 Introduction

As described in section 3.5 the upgrade plans for the Phase-I upgrade of the LHC also require upgrading the LAr calorimeter. To adapt to the described demands of the LAr calorimeter, the trigger resolution and digitization as well as the data processing of trigger cells will be changed. The LATOME project fits in this picture as the hardware and firmware for the data processing and respresent by that a elementary part of the LDPS.

The figure 4.1 gives an overview of the signal paths from the LAr detector up to the Level 1 Trigger. The Front-End Boards on the upper left corner shaped the raw data for single LAr cells and stored them in a Switched Capacitor Array (SCA) waiting for readout triggered by a positive flag from the trigger system. For the trigger path it sends the summed data of this shaped pulses as SC-data to the LTDB where all data will be digitized and encoded. The digitized data will be send through optical links to the LDPS.

The LDPS consists of several Advanced Telecommunications Computing Architecture (ATCA) racks which carries 124 AMC cards for data processing. The LATOME hardware, as shown in section 4.2, represents one of this AMC cards. Each LATOME card receives the digitized data for 320 SC at 40 MHz and processed them in real time. The outgoing data contains electronic Feature Extraction (eFEX), jet Feature Extraction (jFEX) and global Feature Extraction (gFEX) informations and will be passed to the Level 1 Carorimeter Trigger (L1 Calo) and to the Monitoring or TDAQ as well.

As described in the Firmware specification [13] the LATOME has four main requirements. First and foremost, it has to handle the high speed links of the LTDB. As a second it performs a filtering algorithm to reconstruct the $E_T^{SuperCell}$ every 25 ns and identify the Bunch Crossing where the deposited energy was initiated (Bunch Crossing ID (BCid)). The third requirement is to send the calculated energies to the L1 Calo FEXs. Last but not least it has to process and buffer data to be delivered to the TDAQ readout chain and to the monitoring, upon request - Level 1 Accept (L1A) or other.

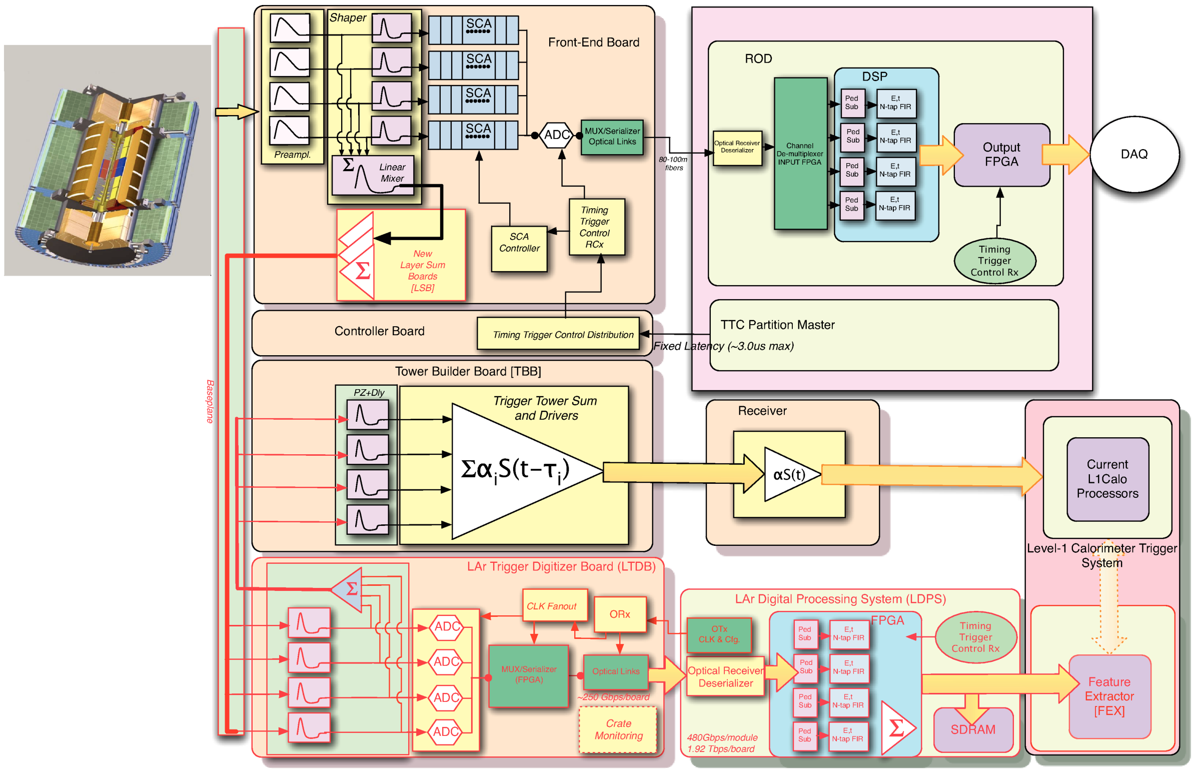

Shaper
Preampl.
SCA
Front-End Board
Linear Mixer
ADC
MUX/Serializer Optical Links
New Layer Sum Boards [LSB]
SCA Controller
Timing Trigger Control RCx
Controller Board
Timing Trigger Control Distribution
Tower Builder Board [TBB]
PZ+Dly
Trigger Tower Sum and Drivers
Baseplane
ROD
DSP
Optical Receiver Deserializer
Channel De-multiplexer INPUT FPGA
Ped Sub
E,t N-tap FIR
Output FPGA
DAQ
Timing Trigger Control Rx
80-100m fibers
TTC Partition Master
Fixed Latency (~3.0us max)
Receiver
Current L1Calo Processors
Level-1 Calorimeter Trigger System
LAr Trigger Digitizer Board (LTDB)
CLK Fanout
ORx
MUX/Serializer (FPGA)
Optical Links
~250 Gbps/board
Crate Monitoring
LAr Digital Processing System (LDPS)
OTx CLK & Cfg.
FPGA
Optical Receiver Deserializer
480Gbps/module 1.92 Tbps/board
SDRAM
Feature Extractor [FEX]

Figure 4.1: Schematic representation of the trigger paths for TT and SCs. [4]

4.2 Hardware

The LATOME hardware, as shown in figure 4.2, is based on a Altera Arria-10 $10AX115R4F40I3SG$ FPGA device. Furthermore four optical transceiver (μPOD) provides the capability to assign the data of 48 optical fibers to the FPGA. Each fiber has a transfer rate up to 12 Gbps. All incomming and outgoing data-streams for one LATOME board are summarized in table 4.1.

ADC data 40 MHz	FEX data 40 MHz	LDPB Monitoring	Data Monitoiring	ATLAS Event-TDAQ	DCS
204 Gbps	$\approx$ 300 Gbps	$\ll$ 1 Gbps	$\simeq$ 20 Gbps	$\simeq$ 1 Gbps	$\ll$ 1 Gbps

Table 4.1: Data flow for one LATOME board. [13]

The Thermal Design Power (TDP) is limited by the used ATCA standard and should not exceed 80W per board. To dissipate heat from the FPGA and transceivers, aluminum heatsinks are installed.

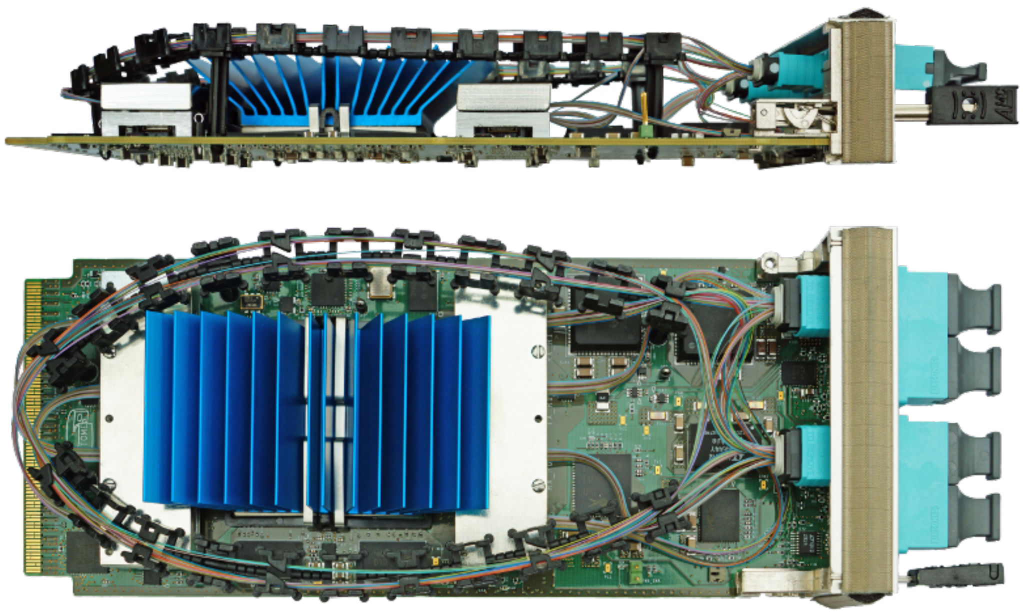


Figure 4.2: Profil- and topdown-view of the LATOME v2 prototyp [13]

4.3 Firmware

The firmware is organized in different high-level modules written in the Hardware Description Languages VHDL and Verilog. All used modules of the LATOME firmware are represeted in the block diagram at figure 4.3.

The firmware is built around the Low Level Interface (LLI) which controls the hardware components such as memory access to the DDR3, the μPODs or access to the 1 GbE ethernet link. The main path of the detector data runs from the LTDB input to the FEX output and has been organised in four logical blocks. The Input Stage (ISTAGE) is responsible for deserializing and demultiplexing of the 12bit ADC data from the LTDB. To take care of the detector geometry and assign associated SCs for the gFEX and jFEX calculation in the proper processing chain, the Configurable Remapping (REMAP) module is used. The User Code (USER) applies the filter to reconstruct $E_T^{SuperCell}$ and determine the BCid. The last block is the Output Summing (OSUM) which makes the proper geometrical sums for gFEX and jFEX. Three other functions are included in the firmware. The TDAQ Monitoring (MON) which organises the transfer of data to ATLAS Trigger and Data Acquisition (TDAQ), the IPBus Controller (IPCTRL) which provides the interface for the LATOME Slow Control system and the Timing, Trigger and Control (TTC) which decodes and provides signals from the TTC system of the LHC.

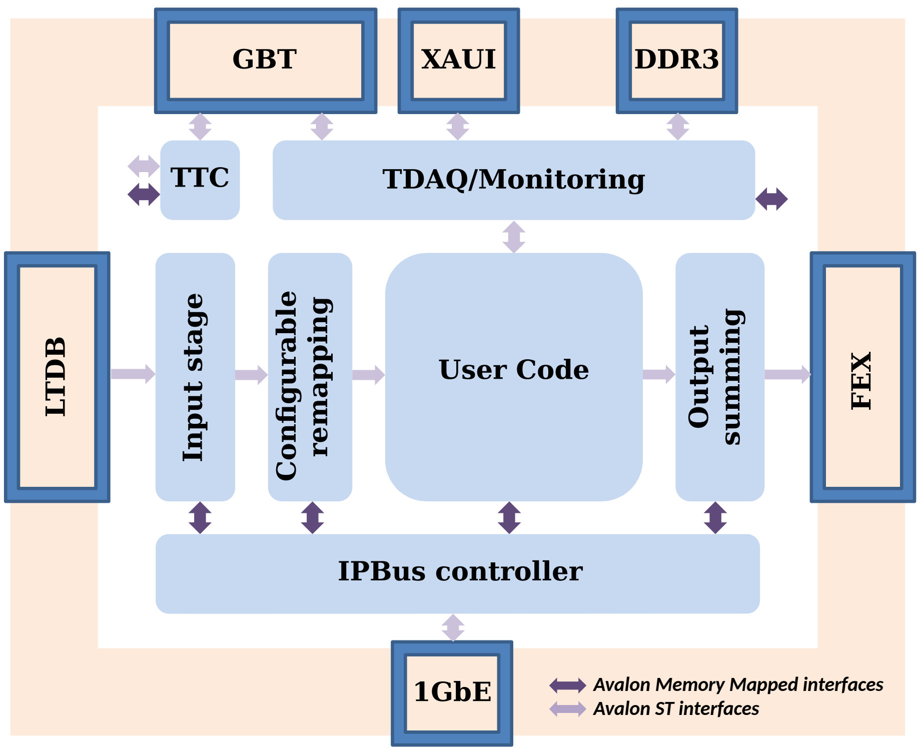


Figure 4.3: Schematic overview of the LATOME Firmware modules - LLI, ISTAGE, REMAP, USER, OSUM, MON, IPCTRL and TTC [13]

The followed chapters 5 and 6 are mainly based on two of this modules. Therefore, they will be discussed in more detail.

4.3.1 IPBus Controller

The task of the IPCTRL is to provide a communication channel between the modules and the outer world. Through this it is possible to load or change configuration parameters or monitoring the states of the modules. The transport mechanism is the IPbus protocol over Ethernet. Due to the low transmission rate, compared to data processing of the LAr detector data, this system is also called Slow Control.

In the following figure 4.4 the data flow of the Slow Control is shown. For a better overview, it is advantageous to consider the data flow at different levels of abstraction.

The physical abstraction represents the hardware level. On the LATOME board itself, there are other controllers besides the ARRIA10 FPGA. The provision of 1 GBit network communication, which is associated with the Slow Control, is realized by a physical layer of the OSI

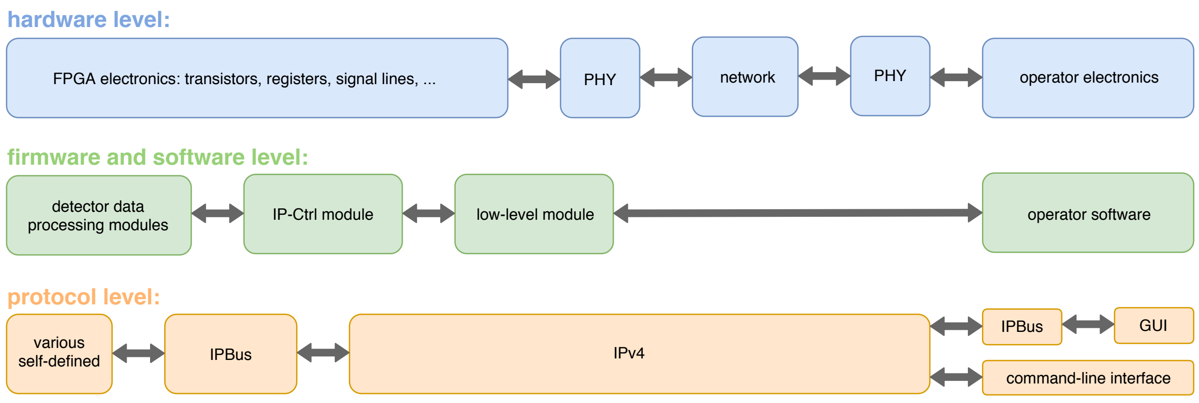


Figure 4.4: Data flow schematic of the Slow Control communication presented in three different abstraction levels - from the point of view of hardware, firmware and protocols.

model (PHY) controller. A potential client that wants to communicate with the LATOME hardware over the network is also connected to it's network card through a PHY controller. The network itself can only consist of one cable or contain many other instances of PHY cascades. Since these are not relevant for the further explanations, the network is summarized as a single block as shown in figure 4.4.

The firmware and software layer in figure 4.4 represents the next level of abstraction. The data flow for controlling a firmware module always runs the same way. The individual firmware modules are coupled in parallel to the IPCTRL module which represents a central instance. The serialized communication to the outer world happens via the LLI module. To control the firmware, various software components are required and allows the operator to access the firmware. They are displayed as an operator software block. As an example of such software, the Serendip GUI is shown in Figure 5.3.

To ensure error-free communication between the individual blocks of firmware and software, communication standards must be defined for the different requirements. This is shown as the third level of abstraction in the figure 4.4 and is called protocol level. The communication between firmware and client is defined by the Internet Protocol version 4 (IPv4) protocol. Due to the inherent properties of IPv4, e.g. routing capability or the detection of transmission errors, it forms a good basis for Slow Control communication. The payload of IPv4 are encoded via the IPBus protocol and is optimized for the demands of detector control.

4.3.2 User Code

An important second module is the User Code(USER). From a physical point of view, the USER represents the main module. It receives the SC data from the Configurable Remapping block and outputs synchronously the reconstructed transverse energy $E_T^{SuperCell}$ as well as the reconstructed BCid and additionally some quality bits to the Output Summing.

As shown in figure 4.5 the user code consists of three blocks. The Filtering Block runs a Finite Impulse Reponse (FIR) filter which converts the 12bit Analog-to-Digital-Converter (ADC) data to transverse energy and its relative phase τ. The Selection Criteria block is responsible to determine the BCid and also to detect irregular pulse shapes, such as saturated pulses. To account the calibration of every SC, the Baseline correction block is used. The Combine Block unites the outputs from the above two blocks and provides $E_T^{SuperCell}$ with the associated BCid as well as the quality information.

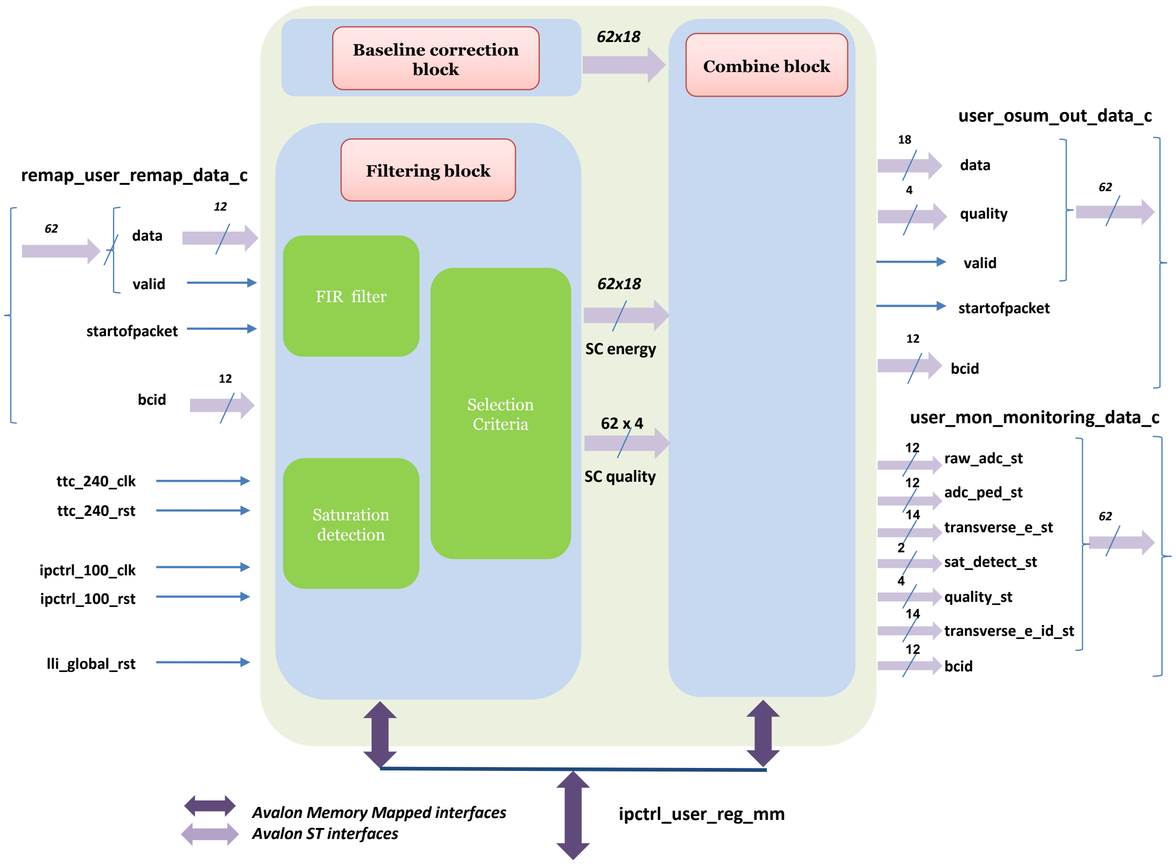


Figure 4.5: A schematic view of the User Code, the different instances of processing are shown. [13]

4.4 Test Benches

Since the LATOME firmware has to be tested, there are several ways to do that. The two shown ways below are fundamentally different and every way has their advantages and disadvantages. As a first option you can compile the firmware and load it on a corresponding FPGA hardware. The advantage here is that the firmware can be tested in real time and under real conditions. The disadvantage is that the complete firmware has to be compiled for each test. This procedure requires a lot of time. To counter this, there is a second option to test the firmware in a simulation. Here, the firmware can be partially emulated on a computer architecture. Emulating saves the time of the compilation process, but runs much slower in execution.

4.4.1 Hardware

There are two possibilities to test the LATOME firmware on hardware. The first option is testing on the development platform of the manufacturer of the FPGA. This is called a Development Kit (Dev-Kit) and contains almost the same FPGA as the later AMC cards. It should be noted that the connection via the LLI module must be adjusted and the full computing capacity is not available, since the FPGA is a development sample. The necessary interfaces, e.g. the optical links for the data path or the network interface for the Slow Control are also present on this card. The figure 4.6 shows an example of such a Dev-Kit card which was available for the tests.

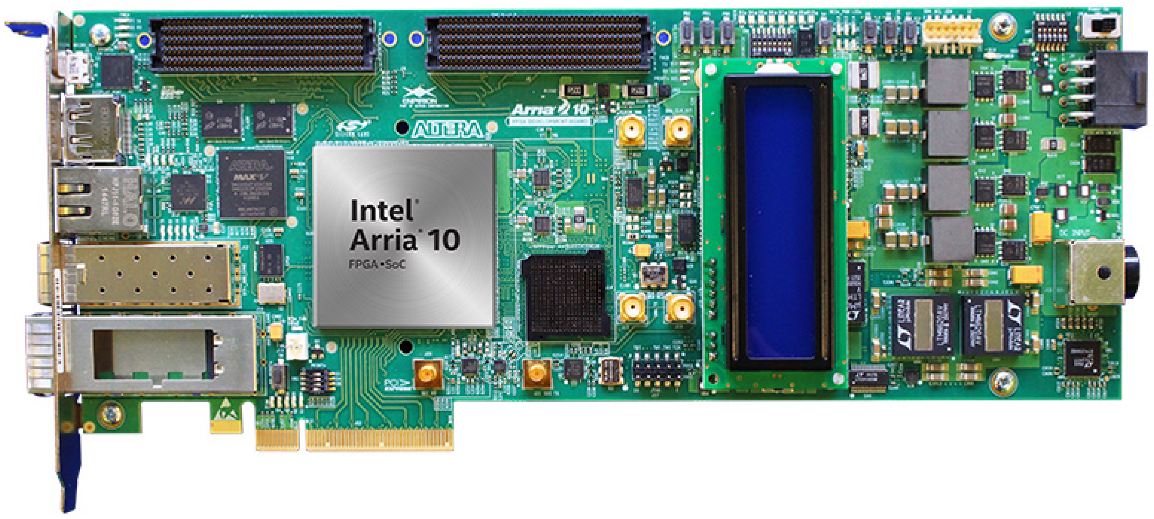


Figure 4.6: An Intel Arria 10 Development Kit card, carrying an Arria 10 FPGA [17]

In addition to the opportunity to use the Dev-Kit, a test platform which contains various prototypes of the LATOME hardware was available at the EMF building at CERN. This could also be used for harware test under real conditions after consultation the other developers.

The advantage of this hardware is to use the target hardware and perform realistic tests. An example of this test setup is shown in the figure 4.7.

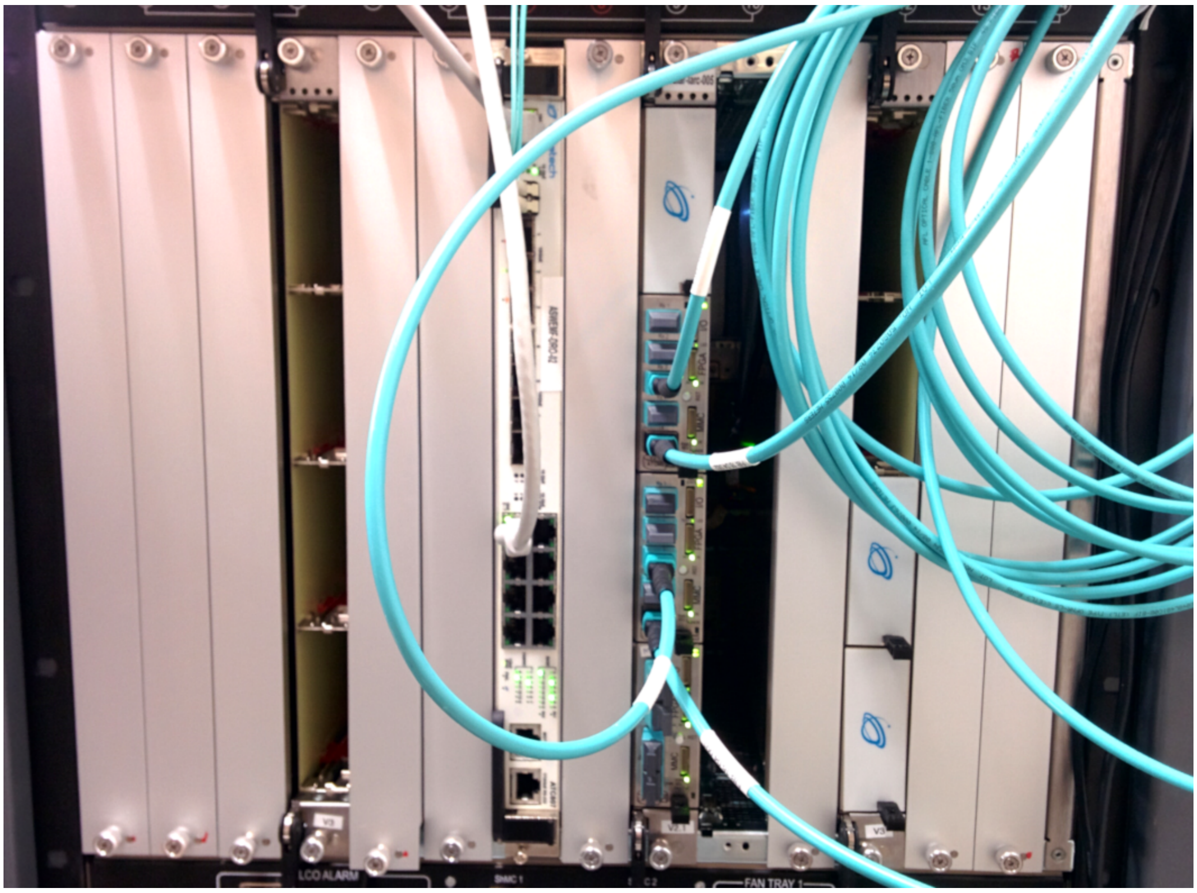


Figure 4.7: A test setup of the prototypes of the LATOME hardware at EMF at CERN

4.4.2 Simulation

The possibility to simulate the firmware is the most frequently used variant to test the firmware or parts of it. Because it does not require any hardware, simulations from each development group can be executed and analyzed on their own computers. This procedure is usually done concomitantly with the development. The FPGA used for the LATOME project requires the Intel Software Quartus for the analysis and synthesis of the HDL code. Once the synthesis has been successfully completed, a simulation environment will be provided via the Questa SIM program. This software contains three essential components which are necessary for the simulation.

The first is the possibility of access to all signals which are described in the HDL source code. In this case, desired signals can be selected and set up with start states. It is also possible to display selected signals graphically over time which makes troubleshooting and debugging much easier.

Another important component is the control of the simulation. By this the simulation can be started or stopped and also configured with different signal trigger conditions for controlling. This is very helpful to be able to find the states you are looking for during the simulation.

A last major component is the possibility of coupling external signals and controlling informations to the simulation. This is done via a C-interface and allows to dynamically control the simulation as shown in the chapter 5.

5 LATOME Virtual Network

5.1 Introduction

As already described in chapter 4.3, there are different levels of data processing in the LATOME project. The data stream with the highest information density is given by the processing chain of the detector data. In this case, the LTDB data arrives at the Input Stage and forms the input to be processed. During processing, the energies are calculated and provided as Feature Extractor (FEX) data to both the trigger level and the monitoring. A much smaller communication stream is the Slow Control. As already explained in chapter 4.3, it is responsible for the administrative access to the involved firmware modules and by that for the parameterization, the monitoring as well as the debugging of all single modules.

Since all firmware modules are subject to a development process, newly implemented functions in the LATOME firmware must also be checked. As already shown in chapters 4.4, there are two different methods for doing this. Because this chapter will mainly be about debugging, both methods are quickly highlighted again under this aspect. The first method - test the firmware on a hardware platform - allows debugging in real time and dynamic intervention in ongoing tests. If malfunctions are detected, the code must be adapted and recompiled and checked again. As the amount of firmware increases, the compile time grows above average, so this type of debugging is very limited. The second option - to simulate the firmware - eliminates some time consuming compilation procedures. These are e.g. the calculation of the electrical FPGA routing or the optimization of the electrical signal transit time. Therefore, the simulation represents a variant of rapid debugging. One of the disadvantages of simulating is the lack of client-side slow-control connectivity. Thus, for each test, the IPv4 packets of the desired control commands must be manually generated and compiled into the simulation. Likewise, the evaluation of the slow-control answers are encoded using different protocols. This makes it difficult to quickly interpret the answers.

In order to eliminate the disadvantages of both methods as much as possible and to combine the advantages, the Virtual Network was developed.

The Virtual Network Handler (VirtNet) is a server-client toolbox which provides direct access to the simulated IP-based control protocol for ATCA (IPBus) controller via real network interfaces on a Linux operating system. This allows to communicate with the simulated firmware like a firmware on physical LATOME hardware and simplify the workflow of IP based communication tests. It also allows to develop IPBus communication tools, even if the firmware is not yet fully developed.

5.2 Structure

As explained in figure 4.4, the data flow of the Slow Control can be viewed at various levels. While simulating the firmware, an associated delimitation can be assigned to each of these levels. This delimitation is shown in the left side of the figure 5.1. The right-hand area of this figure shows the delineation of the levels with respect to the operator of the control software. The connecting element is the VirtNet and can be seen in the middle area. The VirtNet replaces the PHY of the network card on the LATOME hardware with a driver written in Python. It also takes over the functionality of the network and creates a dynamic distributor for routing the bedirectional communication between simulation and operator. In order to ensure autonomous operation of VirtNet, incoming data packets must be analyzed and adapted. For this purpose, special methods were implemented and allow the parsing of the IPv4 protocol.

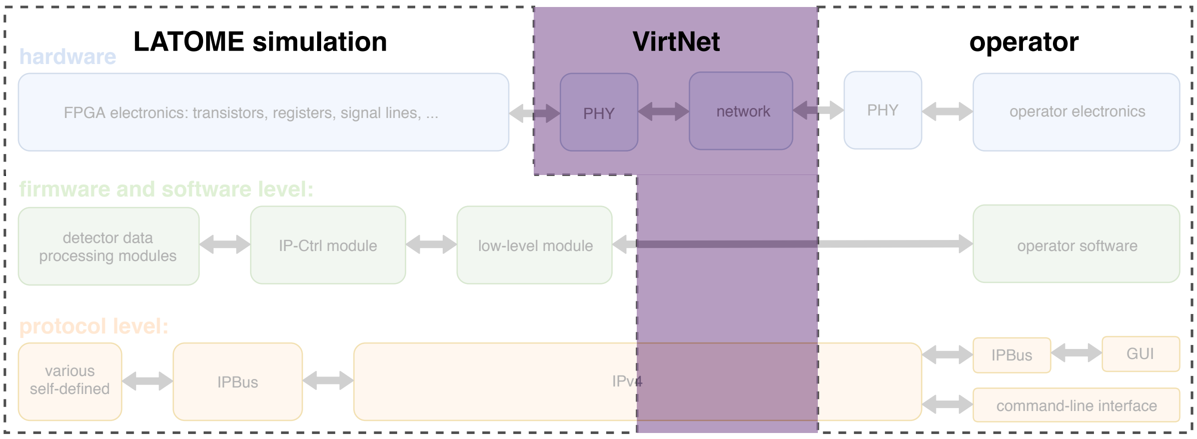


Figure 5.1: Data flow schematic for three different abstraction levels of the Slow Control communication. According to the responsibilities all levels are subdivided for simulation, VirtNet and operator control

The VirtNet is written in Python and can be separated into three important modules.

- **VirtNet daemon**
- **VirtNet simulation**
- **VirtNet controller**

The **VirtNet daemon** is the central communication broker for all incoming and outgoing network traffic from the VirtNet clients and forms the basis. It runs as a Linux daemon in a single instance in the background according to PEP 3143[11]. It can handle multiple simulation sessions in parallel and assign for each simulation the two corresponding clients - VirtNet controller and VirtNet simulation - to each other. For each simulation the VirtNet daemon creates a new virtual network interface (Linux TAP device) with an own class C subnet (e.g. 10.0.0.1 or 10.0.1.1).

The **VirtNet simulation** forms the part which has to be included into the existing simulation to provide access to the VirtNet daemon and by that simulating the PHY of the LATOME hardware. It couples through a coroutine based cosimulation library called cocotb [12] to the C interface of Questa SIM to have dynamic access to all Very High Speed Integrated Circuit Hardware Description Language (VHDL) simulated signals. Furthermore it takes the simulation scheduling and provides a state machine for switching between different simulation states. The provided states are:

- set the simulation on hold and waiting for incoming IPBus requests
- run the simulation for a defined period of time to process routines that are independent of Slow Control communication
- run a defined simulation function
- quit the simulation

The **VirtNet controller** is an interactive client which makes it possible to control and observe the simulation flow. Each simulation requieres an own controller instance. The following shows the control menu of the VirtNet-client.

```
##################################################
TYPE:
 "h" for this help output
 "1" BLOCKING MODE (default)
 "2" UNBLOCKING MODE
 "3-X" UNBLOCK SIMULATION FOR X MICROSECONDS
 "4" CALL BYPASS FUNCTION FROM USER TESTBENCH
 "5" STOP SIMULATION, DROP CONNECTION
 "w" start wireshark
 "q" quit wireshark
 "c" to show current configuration
 "x" to exit the client
FOLLOWED BY ENTER
##################################################
>>> _
```

The listed BLOCKING MODE is the default mode for IPBus communication. If a IPBus request is received the simulation automatically start the processing. After receiving the reponse from the simulated firmware the VirtNet stops the simulation and waiting for further requests. In order to do tests without the need for IPBus communication the UNBLOCKING

MODE was indroduced. The CALL BYPASS FUNCTION option can start a defined coroutine from the VirtNet simulation part. Another important functionality for debuging of the request and responses is provided by the wireshark option. Because a user which doing a simulation usually does not have root privileges on the used computer it is impossible to capture the incoming and outgoing network packages. To still be able to offer the access to the network traffic, an own implementation of the PCAP format was integrated into the VirtNet daemon. If the wireshark option is enabled, the client pc opens the Wireshark program and the daemon pipes the network traffic into it.

The described three instances of the VirtNet can run on a single computer. Due to the high CPU load, powerful servers are often used for the simulation. In such an application it is also possible to execute these three instances on two or three different PCs in the same network. A schematic example of the workflow of VirtNet on different PCs is shown in figure 5.2.

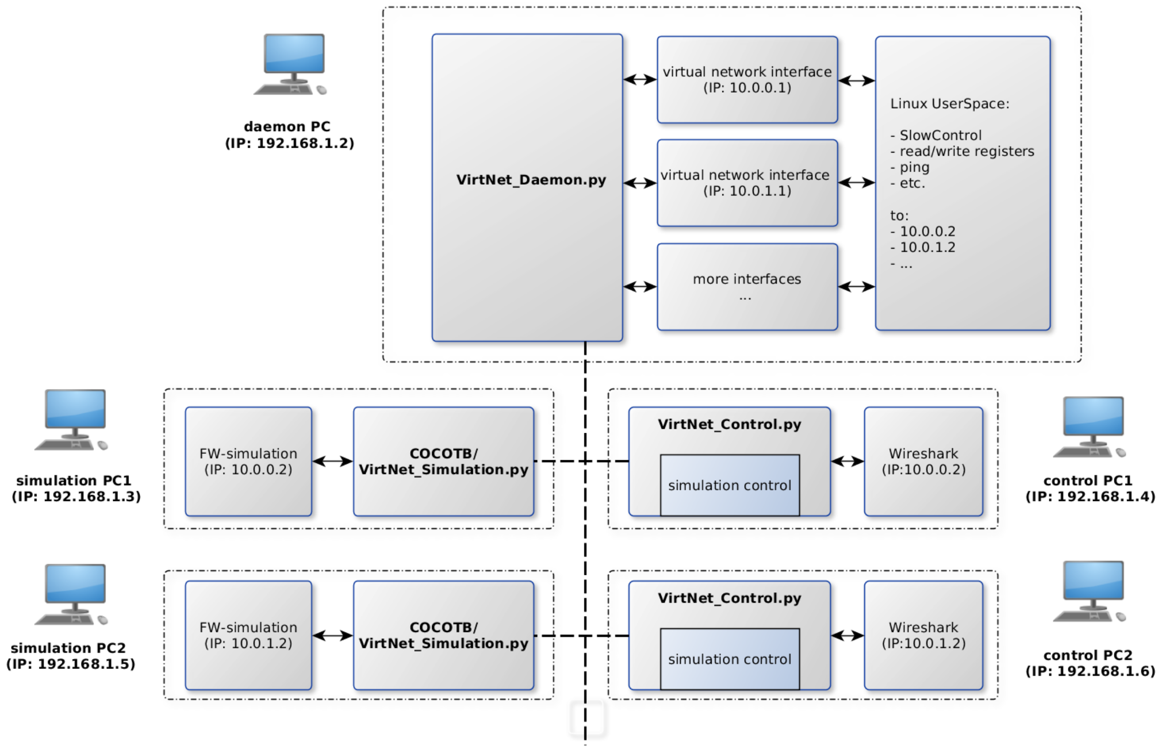


Figure 5.2: Schematic overview of the communication paths of the Virtual Network Handler using dedicated computers

In the upper area you can see the administrative instance of the VirtNet deamon. It can be seen that this provides not only the connection of the other VirtNet instances but also the necessary network interfaces at a Linux OS. A Slow Control communication with one of the shown simulations must be done via this computer. On the left side in the lower area different simulation environments are shown. It is possible to connect up to 255 parallel simulations to the VirtNet deamon. On the right side you can see instances of the VirtNet operator controller. To control the corresponding simulation, it is necessary to start a respective VirtNet controller for each simulation instance.

5.3 Operation and Showcase

The LATOME firmware is developed by several groups. To give them instructions on how to use the VirtNet, a showcase was created. The following steps give a brief overview of how to use the VirtNet.

As a first step a single instance of the VirtNet deamon has to be started.

```
$ cd LATOME/src/ipctrl/tools/VirtNet
$ su
[root]$ ./VirtNet_Daemon.py start
[root]$ exit
```

Before the Questa SIM simulation can be started, a VirtNet controller instance must be created.

```
$ ./VirtNet_Control.py 10.0.0.2
```

To intercept all IP packets transmitted in the Slow Control communication, it is recommended to start Wireshark using the VirtNet controller.

```
>>> w
```

In order to start the simulation, the path of the desired project must be selected and the simulation called via the script framework of the LATOME project.

```
$ cd LATOME/src/ipctrl/test/VirtNet_simulation_showcase
$ make simulation
```

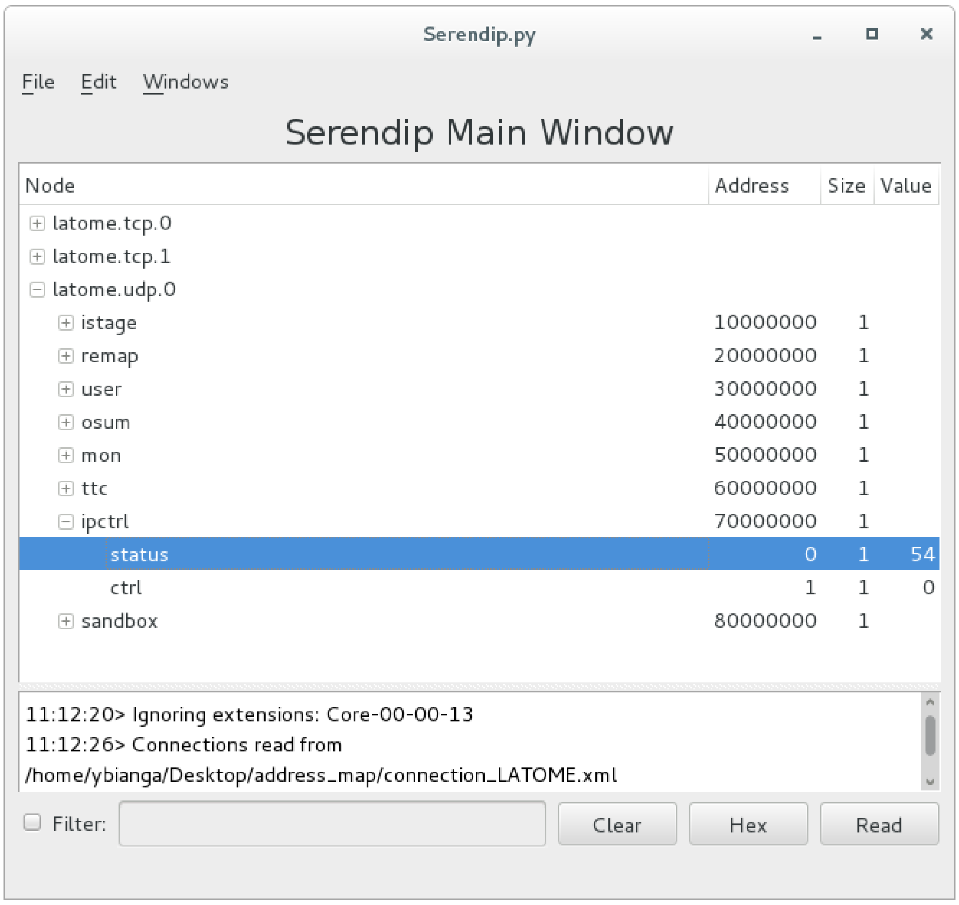
The actual simulation can now be started in the Questa SIM environment.

```
VSIM > run -all
```

After 50 microseconds the firmware is initialized and the simulation is blocking and waiting for Slow Control requests. It is now possible to read or write to the individual registers of each simulated firmware module using a IPBus protocol based software. Simple network tests such as sending a Ping to the simulation are also available.

```
$ ping -c 3 10.0.0.2
```

One of the preferred control software that masters the IPBus Protocol is Serendip. It allows to use a dynamically created list of available registers via the LATOME Framework The figure 5.3 shows the example application of Serendip.



Node	Address	Size	Value
latome.tcp.0			
latome.tcp.1			
latome.udp.0			
istage	10000000	1	
remap	20000000	1	
user	30000000	1	
osum	40000000	1	
mon	50000000	1	
ttc	60000000	1	
ipctrl	70000000	1	
status	0	1	54
ctrl	1	1	0
sandbox	80000000	1	

```
11:12:20> Ignoring extensions: Core-00-00-13
11:12:26> Connections read from
/home/ybianga/Desktop/address_map/connection_LATOME.xml
```

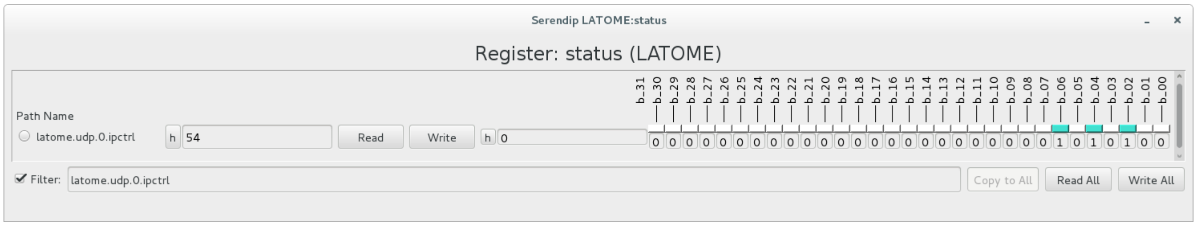
Figure 5.3: The main window of the IPBus based Serendip software

5.4 Results

After the successful connection of the VirtNet to the simulation environment, the current development status of the LATOME firmware modules could be tested via the Slow Control. For this purpose, individual and also automated tests were accomplished.

The first tests involved addressing the IPCTRL module itself to verify the correct operation. For this purpose, a Ping test was performed. The figure 5.5 shows the behavior of the Ping request of the simulated firmware. In the top right corner a Linux console which addresses the simulaion via the TAP network interface is shown. The Wireshark recording at the bottom left area shows next to the two Ping requests and responses also another ARP request which was answered correctly by the firmware. The right part of the figure shows the Signal Tap software and by that the individual signal curves within the simulation using Questa SIM.

For the automated test of the firmware modules via the slow control, scripts were written which covered the entire register space of the LATOME firmware and by that all readable

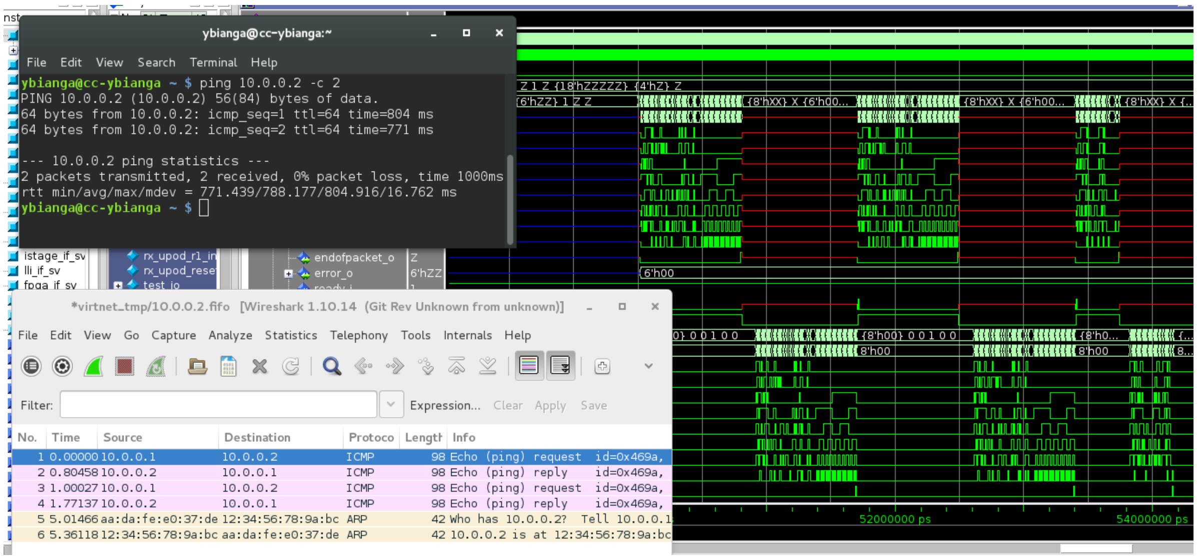


Path Name: latome.udp.0.ipctrl h 54 h 0

b_31	b_30	b_29	b_28	b_27	b_26	b_25	b_24	b_23	b_22	b_21	b_20	b_19	b_18	b_17	b_16	b_15	b_14	b_13	b_12	b_11	b_10	b_09	b_08	b_07	b_06	b_05	b_04	b_03	b_02	b_01	b_00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0

Filter: latome.udp.0.ipctrl

Figure 5.4: Write a register via the Serendip software


ybianga@cc-ybianga:~
File Edit View Search Terminal Help
*virtnet_tmp/10.0.0.2.fifo [Wireshark 1.10.14 (Git Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression...
Clear
Apply
Save
istage_if_sv
lli_if_sv
fpga_if_sv
rx_upod_r1_in
rx_upod_rese
test_io
endofpacket_o
error_o
52000000 ps
54000000 ps

```
ybianga@cc-ybianga ~ $ ping 10.0.0.2 -c 2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=804 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=771 ms

--- 10.0.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 771.439/788.177/804.916/16.762 ms
ybianga@cc-ybianga ~ $
```

```
No. Time    Source            Destination       Protoco Lengtl Info
  1 0.00000 10.0.0.1          10.0.0.2          ICMP     98 Echo (ping) request  id=0x469a,
  2 0.80458 10.0.0.2          10.0.0.1          ICMP     98 Echo (ping) reply    id=0x469a,
  3 1.00027 10.0.0.1          10.0.0.2          ICMP     98 Echo (ping) request  id=0x469a,
  4 1.77137 10.0.0.2          10.0.0.1          ICMP     98 Echo (ping) reply    id=0x469a,
  5 5.01466 aa:da:fe:e0:37:de 12:34:56:78:9a:bc ARP      42 Who has 10.0.0.2?  Tell 10.0.0.1
  6 5.36118 12:34:56:78:9a:bc aa:da:fe:e0:37:de ARP      42 10.0.0.2 is at 12:34:56:78:9a:bc
```

Figure 5.5: An example test using the VirtNet shows the processing of a Ping and ARP request.

and writable registers. After these tests were accomplished, some errors could be identified and forwarded to the responsible development groups.

The register implementation of the ISTAGE, REMAP and OSUM module could mostly be verified. Small errors resulted by incomplete coverage of registers in the data stream configurations by more than 50 processing streams. The IPCTRL implementation of the USER code has been shown to be incomplete in most areas. Since the TTC and also the MON module was not yet available at the time of this test, an automated test could not be performed for this purpose.

6 LATOME Firmware Model

6.1 Introduction

The LATOME firmware is a big project and involving several working groups from different countries. Each single module has clearly defined requirements for its internal tasks and also for the interfacing to the other modules. The development of each module is an ongoing process in which the functionalities are gradually incorporated. To validate these functionalities, all development steps must be checked regularly in the simulation or with hardware tests. To better define the tests, a milestone plan with 12 milestones was developed. This prescribes in which period which areas should be tested. Some of the milestones can be reached in parallel and some are more software or hardware oriented.

The first two milestones relate to the commissioning of the ATCA and LATOME hardware and by that the restoration of the TTC signal and the reading and writing of registers in the firmware.

A far more extensive test is the Milestone 3 (M3). It is intended to test the interaction of most individual modules and also to put the control into operation via the Slow Control. The final goal of M3 is to process detector data which are close to those of the LAr detector in real operation. Mainly, the M3 test require two things. First, it needs well defined data that the firmware can process and second, the processing of the data in all steps must be well known in order to compare them.

To perform these two needs, a LATOME firmware model (FW model) has been introduced. The FW model is a software written in Python that attempts to replicate the complete data processing of all modules in detail. The data flow within each individual module can thus be output at previously defined processing steps and compared with the data flow of the firmware. This allows to define checkpoints at various points in the processing and to compare the firmware model with the processing on hardware. Just like the firmware itself, the FW model is also subject of ongoing development. This is necessary in order to achieve a comparability that accompanies development.

6.2 Structure

The LATOME firmware itself is based on different modules with different tasks. In order to be comparable, this structure was also adopted in the firmware model. For each major firmware module, namely LLI, ISTAGE, REMAP, USER and OSUM, an equivalent software module was coded. As well as in the firmware itself, the software modules have been assembled into a processing chain for detector data. Therefore each single module contains essentially three processing methods. An input method which is used for feeding the module with incomming data. It takes care of the correct data decoding and forwards the data for further processing. In a second step, the incoming data must be processed. This second step is very individual for each module and can take place over several processing stages. A third method is responsible for the correct data encoding and finally return the results. Furthermore there is a central Python instance to associate the inputs and outputs of all modules.

Since the LLI software module has the same decoding as in the firmware, a LTDB module is necessary to dynamically generate a matching input data stream. In the previous version of the FW model, the input data were read in statically via previously generated ADC detector values and transferred to the LLI module. The previous processing chain can be seen in upper part of the figure 6.1. It can also be seen that the USER code exsited only as a transpartent module and ensures that the incoming detector data are passed unfiltered to the OSUM.

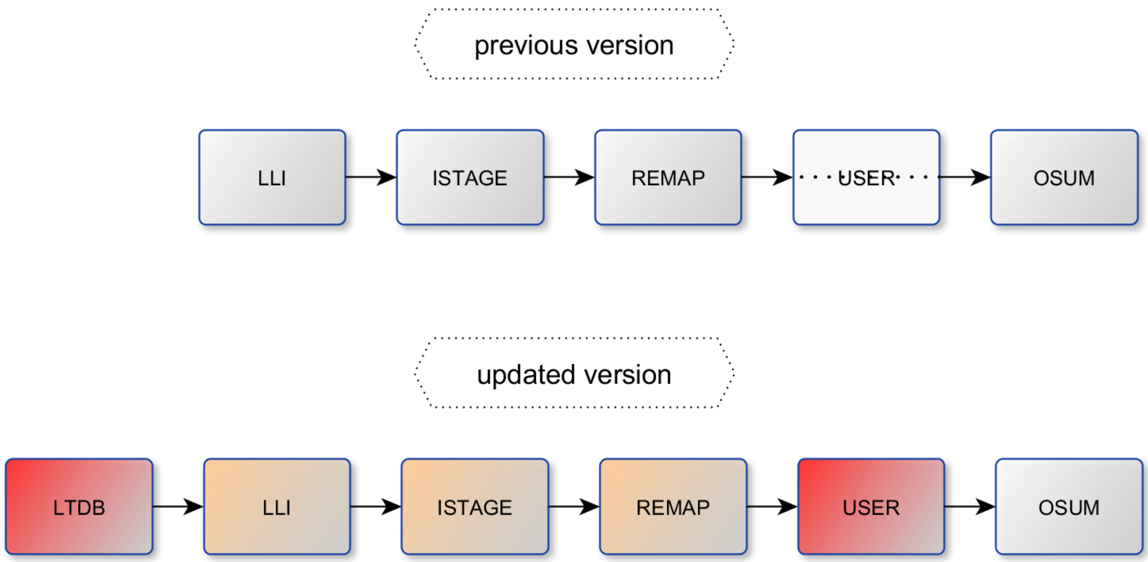


Figure 6.1: The firmware model in the previous state in the upper area, the updated version is shown below.

The bottom part of the figure 6.1 shows the revised FW model version of this work. The FW model has been extended with an LTDB module which can dynamically feed the FW model with different types of detector data. Furthermore, the modules LLI, ISTAGE and REMAP have been updated and some missing parameterizations have been added. This allows the FW model to be configured for different detector areas. Likewise, a Python interfacing to a previously separately developed C++ USER code module was created. In addition to the input and output connection of this C++ module itself, the fully dynamic control of the parameters of this C++ USER module could be implemented.

For all possible hardware configurations each module needs to be propperly configured. The following figure 6.2 shows the various possibilities of parameterizing the FW model. Since the entire LATOME project is controlled by a superordinate framework, there is a predefined project input for different scenarios. The parameters of the project input are largely determined by the detector area to be processed. This includes the number of connected optical fibers between LTDB and LLI as well as the number of outgoing fibers of the OSUM module. The configuration of the reordering in the ISTAGE module and remapping are also derived from this.

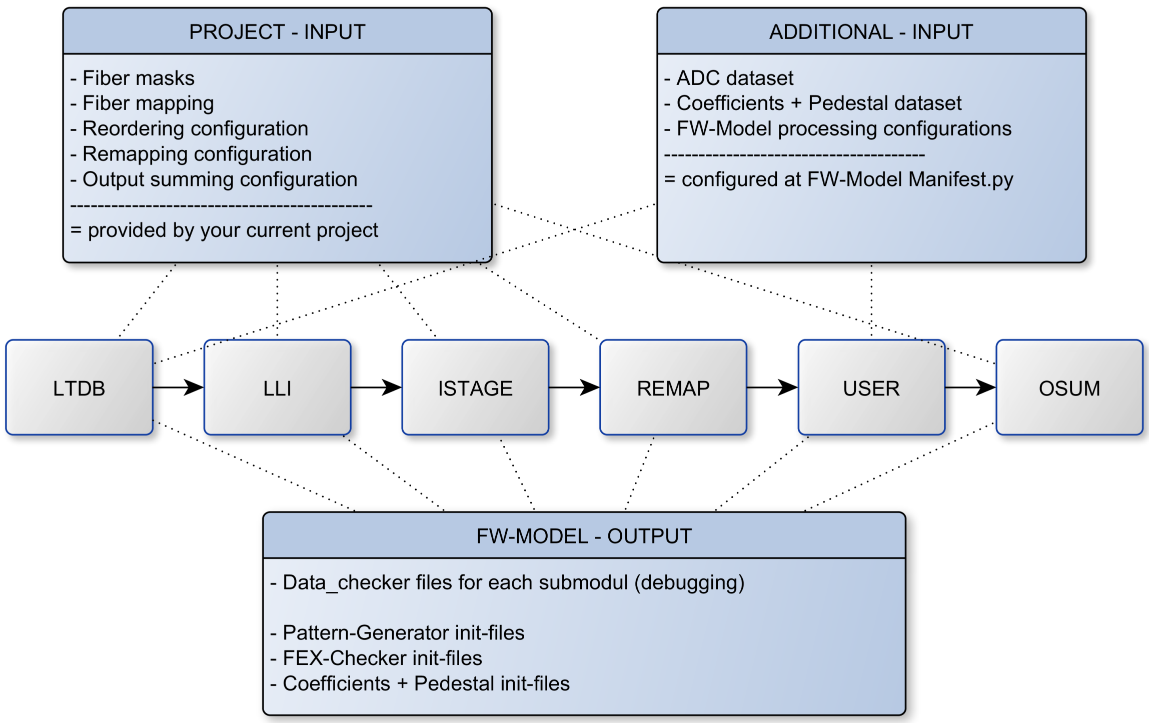

PROJECT - INPUT
- Fiber masks
- Fiber mapping
- Reordering configuration
- Remapping configuration
- Output summing configuration
= provided by your current project
ADDITIONAL - INPUT
- ADC dataset
- Coefficients + Pedestal dataset
- FW-Model processing configurations
= configured at FW-Model Manifest.py
LTDB
LLI
ISTAGE
REMAP
USER
OSUM
FW-MODEL - OUTPUT
- Data_checker files for each submodul (debugging)
- Pattern-Generator init-files
- FEX-Checker init-files
- Coefficients + Pedestal init-files

Figure 6.2: The input and output of the updated LATOME firmware model.

The input that was added by the new LTDB and USER modules is shown as additional input in figure 6.2. By that it can be choosen between different ADC data sets, different coefficients of the detector cells and the mode of filtering via the USER module.

The FW model generates two types of outputs. For development-accompanying debugging, the output can be saved for each individual module and is called data-checker files.

In a further process this allows to compare this modules output directly with the simulation of the firmware or with a test on the hardware. As a second type of output, the FW model allows the creation of initiation files to parameterize the LATOME firmware. Specifically, these are the coefficients and pedestal values of the USER code as well as the detector data to be feed into the LLI. The expected output of each FEX stream at OSUM is also generated.

In order to be able to process physical detector data as described above, the required modules LTDB and USER had to be fully implemented. In the following, these two modules will be described in more detail.

6.2.1 LTDB

As seen in figure 6.3 the LTDB module is responsible to feed different detector data into the FW model. For this there are different generators methods that generate pulses from the detector cells. Since the requirements of the pulses must correlate with the milestones, five different types of pulses are required up to the Milestone M3.

A first pulse pattern has been designed primarily for debugging the firmware. It is named "constant-ID" and is intended to embody the property of containing a unique value at each point in time and in each detector cell. As long as the filter algorithm is deactivated in the USER code, a complete traceability of the data stream can be guaranteed. This is especially important for the investigation of the ISTAGE, REMAP and OSUM module.

If possible errors in the above-mentioned modules can not be found via the "constant-ID" pattern, an attenuated version will be issued. This eliminates the dependencies of the detector cells and thus only contains the time dependency. This is expressed as BCid information and called "constant-BCID". Using this pattern, the mapping can be excellently followed.

In order to be able to understand also the correct filtering of the USER module and thus also the correct application of the filter coefficients, there is another pattern. It is a pattern which repeatedly linearly increments the detector values and thus generates a triangle pulse for each individual detector cell.

Another pattern is the generation of random detector values. Since the expected result of the firmware is known by the FW model, each possible combination can be tested on the developed firmware.

The main test, however, is the testing of realistic detector data. For this purpose, a module was written which imports simulated LAr detector data and can feed the output of the simulation into the FW model. These detector data are simulated by the ATLAS Readout Electronics Upgrade Simulation (AREUS) software and represent physical raw ADC data.

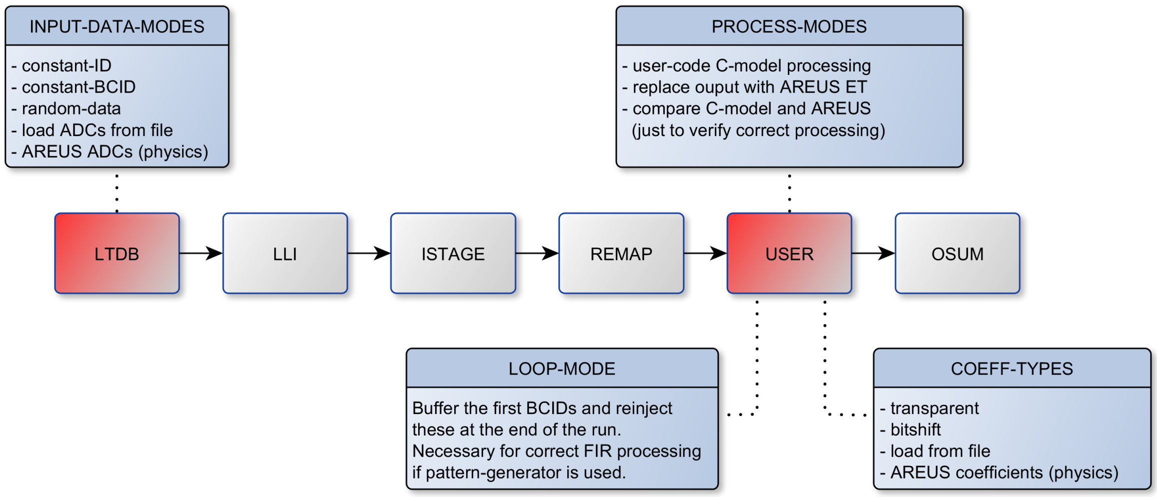

INPUT-DATA-MODES
- constant-ID
- constant-BCID
- random-data
- load ADCs from file
- AREUS ADCs (physics)
PROCESS-MODES
- user-code C-model processing
- replace ouput with AREUS ET
- compare C-model and AREUS
(just to verify correct processing)
LTDB
LLI
ISTAGE
REMAP
USER
OSUM
LOOP-MODE
Buffer the first BCIDs and reinject these at the end of the run. Necessary for correct FIR processing if pattern-generator is used.
COEFF-TYPES
- transparent
- bitshift
- load from file
- AREUS coefficients (physics)

Figure 6.3: Modes of operation of the LATOME FW model.

6.2.2 USER code

The USER module is responsible for filtering of the detector data. On the one hand this is necessary in order to obtain correct energy values for the individual Super Cells and on the other hand to assign the time of arrival of a particle-collision to a BCid.

In addition to the firmware itself, the responsible developers wrote a functionally identical model in C++. Since this model was written in a different programming language than the firmware model, it was necessary to include this C++ model in the FW model. This could be realized via a C-interface of the programming language Python. All input and output streams like ADC values, filter coefficients, quality informations, saturation detection or the transverse energies were tied into the Python model. Just as the data streams, an implementation was created to load the initialization parameters into the model. This allows to specify the coefficients or pedestal for each detector cell. Since the configuration of the developed USER C++ model was static, this was changed so that all necessary parameters can be dynamically feed in the model.

A compilation of some of the most important parameters is shown in table 6.1 below.

name	description
`LHC_CYCLE_NB_BCID`	Bunch Crossing ID to apply the coefficients in the correct order
`DATA_STREAMS_NB`	generate the correct number of filter instances
`TAU_SELECTION_ENABLE`	activate the Tau filtering, necessary to determine the time offset related to the BCid
`HARDPOINT_FILTER_ENABLE`	activate the precision of the later conceived filter
`FILTERING_FIR_FILTER_TAPS_NB`	the filter depth of the FIR filter

Table 6.1: USER code parameter names and descriptions which are passed to the FW model

6.3 AREUS

AREUS [15] is a simulation software that simulates the LAr complete trigger and readout electronics and was specifically designed to investigate the properties of the filter algorithms. In order to precisely investigate the behavior of the filters in the readout electronics of the detector, it is necessary to simulate the detector as detailed as possible.

For this purpose, in a first step, well defined particle hits in the cells of the LAr calorimeter are simulated using GEANT4. The basis for these hits are events from a Monte Carlo simulation. The interaction of the particles with the detector material then contains the information about the deposited energy, the ATLAS coordinates of the hit cell and the BCid. As a next step, the energies of all hits that belongs to a single Bunch Crossing are assigned to associated detector cells. If multiple energies are assigned to one cell, they are added up and will be after that summarized to SCs. After that, these energies are transformed with given analog pulse shapes and a defined noise will also be added. Subsequently, the resulting pulse shapes are subjected to a configurable digitization and forwarded to a filter. The generated output can be used for the desired filter studies.

Since the LATOME firmware is developed as the filter level of this processing chain, the results of the digitization and by that the raw ADC data of the AREUS simulation are ideally suited to feed into the FW model.

Figure 6.4 shows as an example sequence the well defined input from GEANT4 and also the output of the AREUS digitization stage as raw ADC values.

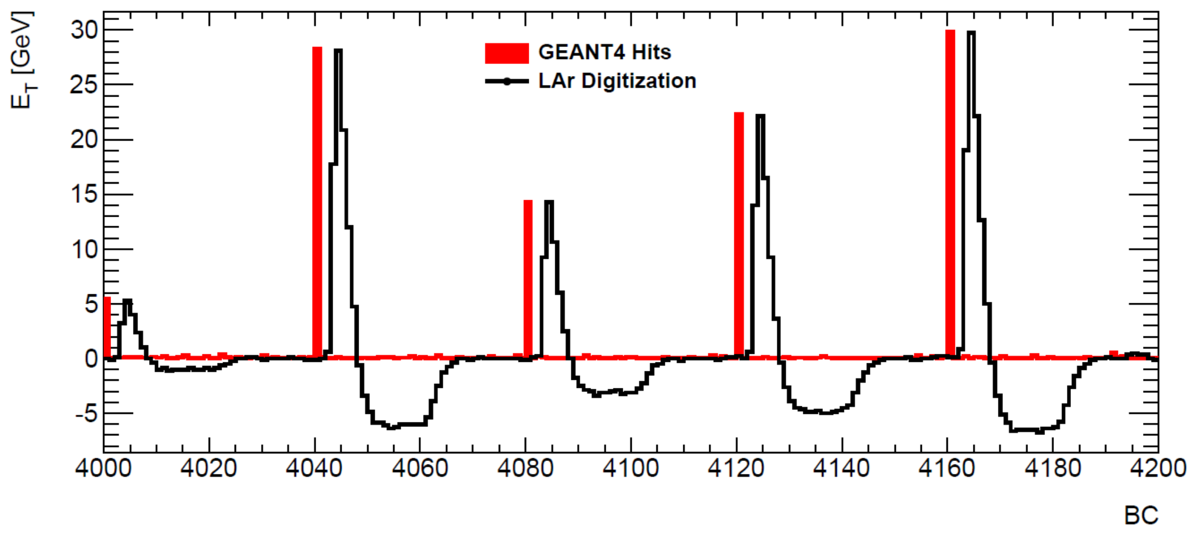


Figure 6.4: Example sequence of the AREUS simulation, GEANT4 Hits forms a defined input, the LAr digitization is shown as a possible output. [15]

6.3.1 Adaptations and physics data sets

In order to use the digitized ADC values in the LATOME firmware model, some adjustments and parameterizations are necessary.

The first thing to do is to decide which physical events and parameters should be used for testing. Since the processing chain is to be examined during the first firmware tests, these parameters play only a minor role. Therefore, simple inelastic electron collisions of two different energy levels were generated in GEANT4 and used for the AREUS simulations. For the low energy level an energy range of 20 to 50 GeV was used. Another record with energies between 1000 and 2000 GeV was also created to check effects such as the saturation detection of the LATOME firmware. In AREUS, these hits were fed with a defined μ distribution at a mean value of $\mu = 80$. A noise of the thermal and electrical influence of the cells and electronics was also added. Spice-simulated data could be used for the pulse shapes of the individual detector cells at different energy levels.

The parameterization of digitization requires a greater attention. To avoid later errors, this must be adapted exactly to the parameters of the LATOME firmware. For this the ADC bitwidth was set to 12 bits with an Least Significant Bit (LSB) of 12.5 MeV as precision. The fix-point arithmetic used in the LATOME firmware also had to be configured. For this purpose, 5 of the 12 bits were used as a decimal fixed point size.

In order to generate the necessary filter coefficients for the LATOME firmware, the FIR filter in AREUS was activated. This also allows to compare the later output of the filtering in the USER module with the FIR output of AREUS. For this the filter depth was set to 4.

The two generated physical data sets were made for 5000 BCids and stored in a ROOT file format. In order to be able to feed the AREUS data sets into the FW model, a ROOT reading and parsing connection to the FW model had to be created. Since AREUS and the firmware use different notation of the SCs, also an SC wrapper had to be implemented and is shown in the appendix B.1.

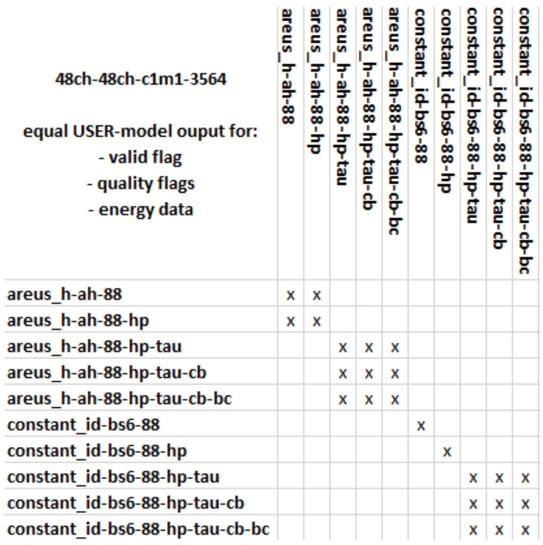
6.4 Investigations and Results

The two hardware test benches that could be used were the two mentioned in chapter 4.4.1. On the one hand the Dev-Kit which could only be operated with 4 channels due to its poorer IO connection and on the other hand the test bench with the LATOME prototypes at the EMF using a 48 channel configuration.

For both test benches all possible combinations of data sets were generated with the FW model. These data sets are composed of the ADC raw data to be fed in, the necessary configuration files of the individual submodules and the expected outputs of the FEX streams. Both, the synthetic data sets such as constant-ID and data sets with physical background from AREUS were varied in the USER code configuration capabilities. In particular, the parameters of the tau selection (tau), the activation of the combine block (cb) and also the baseline correction (bc) are meant. Also the used coefficients have been adjusted.

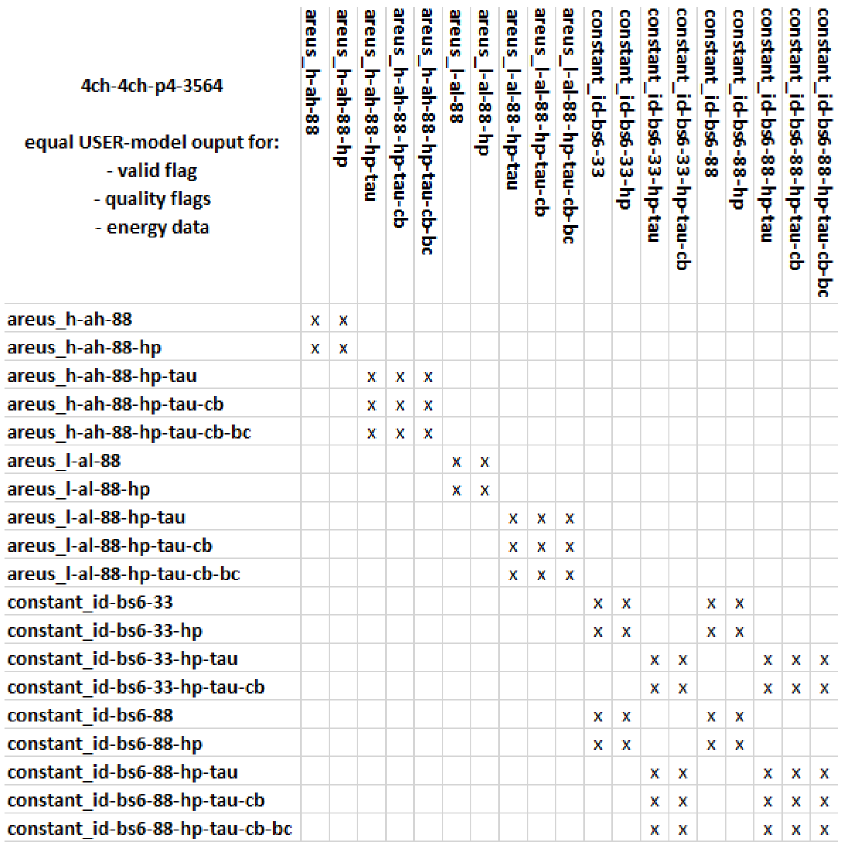
All data sets were tested on hardware, but the initial tests with this data sets led to problems on both test benches and did not yield the desired FEX outputs. The reason were several smaller errors in some firmware modules. Two major bugs found by the tests involved the REMAP and OSUM modules of the firmware. In the remapping there were problems with the routing of the different data streams which are processed in parallel and the missing implementation of the jFEX and gFEX protocol at OSUM.

Another observation concerned the USER code. It could be shown that the planned functions of the combine block and the baseline correction are not functional in the firmware. A comparison of the identical data sets for both test benches can be found in the figures 6.6 and 6.5.



48ch-48ch-c1m1-3564 equal USER-model ouput for: - valid flag - quality flags - energy data	areus_h-ah-88	areus_h-ah-88-hp	areus_h-ah-88-hp-tau	areus_h-ah-88-hp-tau-cb	areus_h-ah-88-hp-tau-cb-bc	constant_id-bs6-88	constant_id-bs6-88-hp	constant_id-bs6-88-hp-tau	constant_id-bs6-88-hp-tau-cb	constant_id-bs6-88-hp-tau-cb-bc
areus_h-ah-88	x	x								
areus_h-ah-88-hp	x	x								
areus_h-ah-88-hp-tau			x	x	x					
areus_h-ah-88-hp-tau-cb			x	x	x					
areus_h-ah-88-hp-tau-cb-bc			x	x	x					
constant_id-bs6-88						x				
constant_id-bs6-88-hp							x			
constant_id-bs6-88-hp-tau								x	x	x
constant_id-bs6-88-hp-tau-cb								x	x	x
constant_id-bs6-88-hp-tau-cb-bc								x	x	x

Figure 6.5: Results of the backtesting at the LATOME prototype cards at EMF. The table shows the indentification of equal outputs of the USER module.



4ch-4ch-p4-3564 equal USER-model ouput for: - valid flag - quality flags - energy data	areus_h-ah-88	areus_h-ah-88-hp	areus_h-ah-88-hp-tau	areus_h-ah-88-hp-tau-cb	areus_h-ah-88-hp-tau-cb-bc	areus_l-al-88	areus_l-al-88-hp	areus_l-al-88-hp-tau	areus_l-al-88-hp-tau-cb	areus_l-al-88-hp-tau-cb-bc	constant_id-bs6-33	constant_id-bs6-33-hp	constant_id-bs6-33-hp-tau	constant_id-bs6-33-hp-tau-cb	constant_id-bs6-88	constant_id-bs6-88-hp	constant_id-bs6-88-hp-tau	constant_id-bs6-88-hp-tau-cb	constant_id-bs6-88-hp-tau-cb-bc
areus_h-ah-88	x	x																	
areus_h-ah-88-hp	x	x																	
areus_h-ah-88-hp-tau			x	x	x														
areus_h-ah-88-hp-tau-cb			x	x	x														
areus_h-ah-88-hp-tau-cb-bc			x	x	x														
areus_l-al-88						x	x												
areus_l-al-88-hp						x	x												
areus_l-al-88-hp-tau								x	x	x									
areus_l-al-88-hp-tau-cb								x	x	x									
areus_l-al-88-hp-tau-cb-bc								x	x	x									
constant_id-bs6-33											x	x			x	x			
constant_id-bs6-33-hp											x	x			x	x			
constant_id-bs6-33-hp-tau													x	x			x	x	x
constant_id-bs6-33-hp-tau-cb													x	x			x	x	x
constant_id-bs6-88											x	x			x	x			
constant_id-bs6-88-hp											x	x			x	x			
constant_id-bs6-88-hp-tau													x	x			x	x	x
constant_id-bs6-88-hp-tau-cb													x	x			x	x	x
constant_id-bs6-88-hp-tau-cb-bc													x	x			x	x	x

Figure 6.6: Results of the backtesting at the Dev-Kit. The table shows the indentification of equal outputs of the USER module.

7 Summary and Outlook

The aim of this work was to develop test systems that support the development process of the LATOME firmware as much as possible. For this purpose, two different test systems were introduced.

The first test system -the Virtual Network Handler - makes debugging execution much more dynamic while simulating the LATOME firmware. Instead of incorporating predefined control commands into the simulation, the simulation control can now be individually selected and fed in as required. This process can save a lot of development time. Simultaneously with the development of the LATOME firmware, it also allows to develop the required IPBus control software in parallel.

Since the VirtNet is essentially a dynamically IPBus router between the outer world and firmware simulations, it could be considered to hand this tool over to the IPBus developers for a basic integration into the IPBus code.

The second test system - the LATOME firmware model - represents a replica of the data processing chain of the LATOME firmware. This model could be supplemented by two further submodules. The introduced LTDB module allows the dynamic generation of different types of detector data. In addition to several synthetic test patterns, it is now also possible to feed physical data from an AREUS simulation of the LAr into the FW model. This makes it possible to test the firmware under realistic conditions. By coupling the separately developed C++ USER model to the FW model, the complete processing chain of the FW model was produced. With these implementations, all requirements for the M3 milestone were met and M3 was completed satisfactorily.

Further work on the FW model is available on the C++ model of the USER code. Since not all planned methods of data processing have yet been implemented, they must be integrated and the FW model adapted accordingly. An example for this is the planned baseline correction. Since not all modules of the LATOME firmware are included in the FW model, there is still a need for further implementations. This concerns above all the two modules TTC and MON.

A LATOME Virtual Network

A.1 Workflow

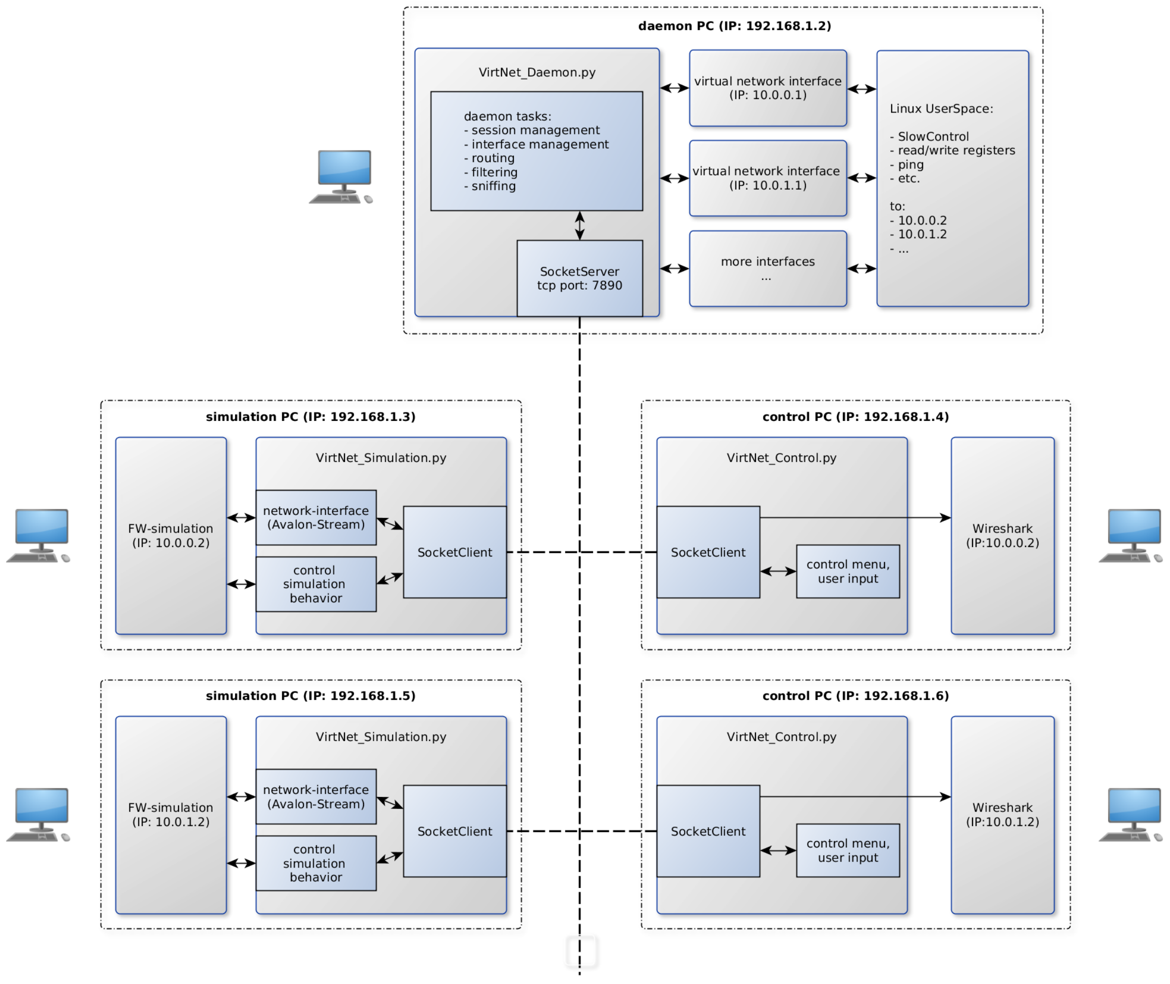


Figure A.1: The data flow of the VirtNet in detail.

A.2 Source Code

A.2.1 VirtNet daemon

```
#!/usr/bin/env python
# −*− coding: utf−8 −*−

"""
−−−VirtualNetworkHandler−Daemon−−−

This is a daemon for handeling virtual networks and routing IP packets.
It can be controlled via VirtNet−Clients.
"""

__author__ = "Yves␣Bianga"
__version__ = "2.0"
__email__ = "yves.bianga@cern.ch"


from pprint import pprint
from ast import literal_eval
import socket
import select
import time
import json
import fcntl
import os
import sys
import struct
import subprocess
import logging
sys.dont_write_bytecode = True
# from VirtNet dependencies
from virtnet_deps.VirtNet_Base import *
from virtnet_deps.daemon_lib import runner


class VirtNet_Adapter(ThreadBase):
    """
    create, configure and close a TAP interface on Linux OS
    """

    def __init__(self, sim_ip):
        self.sim_ip = sim_ip
        self.sim_network_ip = '%s.%s.%s.1' % (sim_ip.split('.')[0], sim_ip.split('.')[1], sim_ip.split('.')[2])
        self.sim_network_dev = 'tap%s' % (sim_ip.replace(".", ""))
        self.callback_reg_warn = True

        if not self._layerInit:
            super(VirtNet_Adapter, self).initLayer()

        if self.__check_device_status():
            self.alive = True
        else:
            self.alive = False
        self.uid = os.getuid()
        self.__callback_func = []
        ThreadBase.__init__(self, name=self.__class__.__name__, service=self.__service__, exit=self.__exit__)

    def __service__(self):
        # this methode is reading the TAP/slowcontrol and forward it to the simulator
        if self.alive:
            self.fd = self.__bring_device_up()
            while self.alive:
```

```
            try:
                packet = os.read(self.fd, 2**16)
                if self.alive:
                    self.__callback(packet)
                else:
                    self.tap.close()
            except:
                self.__exit__()

def __exit__(self):
    self.alive = False
    self.tap.close()
    logger.info('%s␣–␣TAP␣adapter␣(%s,␣%s)␣stopped' % (self.sim_ip, self.sim_network_ip, self.sim_network_dev))

def __bring_device_up(self):
    self.tap = self.__create_tap()
    fd = self.tap.fileno()
    logger.info('%s␣–␣TAP␣adapter␣(%s,␣%s)␣started' % (self.sim_ip, self.sim_network_ip, self.sim_network_dev))
    return fd

def __check_device_status(self):
    if self.sim_network_dev in self.C2S('ifconfig'):
        return False
    else:
        return True

def __create_tap(self):
    TAPSETIFF = 0x400454ca
    TAPSETOWNER = TAPSETIFF + 2
    IFF_TAP = 0x0002
    IFF_NO_PI = 0x1000
    tap = open('/dev/net/tun', 'r+b')
    ifr = struct.pack('16sH', self.sim_network_dev, IFF_TAP | IFF_NO_PI)
    fcntl.ioctl(tap, TAPSETIFF, ifr)
    fcntl.ioctl(tap, TAPSETOWNER, self.uid)

    self.rampup_device()
    self.set_netmask()
    self.check_ipv6()

    self.set_retrans_timeout()
    return tap

def rampup_device(self):
    subprocess.check_call('ifconfig␣%s␣%s␣up' % (self.sim_network_dev, self.sim_network_ip), shell=True)
    time.sleep(1)

def set_netmask(self):
    subprocess.check_call('ifconfig␣%s␣netmask␣%s' % (self.sim_network_dev, '255.255.255.0'), shell=True)
    time.sleep(0.5)

def check_ipv6(self):
    for i in self.C2S('ifconfig␣%s' % self.sim_network_dev).split('\n'):
        if 'inet6␣addr:' in i:
            addr = i.split('inet6␣addr:')[1].split('␣')[1]
            self.remove_ipv6_debian(addr)
        elif 'inet6' in i:
            addr = i.split('inet6')[1].split('␣')[1]
            self.remove_ipv6_centos(addr)

def remove_ipv6_debian(self, addr):
    subprocess.check_call('ifconfig␣%s␣inet6␣del␣%s' % (self.sim_network_dev, addr), shell=True)

def remove_ipv6_centos(self, addr):
    subprocess.check_call('ifconfig␣%s␣inet6␣del␣%s/64' % (self.sim_network_dev, addr), shell=True)
    def set_retrans_timeout(self, timeout=30000):
        # set ARP retransmission timeout
        if self.C2S('whoami').split('\n')[0] == 'root':
            subprocess.check_call('echo␣"%s"␣>␣/proc/sys/net/ipv4/neigh/%s/retrans_time_ms' % (str(timeout), self.
                ↪ sim_network_dev), shell=True)
            logger.info('%s␣−␣set␣retransmission␣timeout␣for␣%␣s␣to␣%s' % (self.sim_ip, self.sim_network_dev, timeout))
        else:
            logger.warn('%s␣−␣running␣without␣root␣permissions,␣can\'t␣set␣retransmission␣timeout␣for␣%s' % (self.sim_ip,
                ↪ self.sim_network_dev))

    def __callback(self, packet):
        if len(self.__callback_func) > 0:
            for i in self.__callback_func:
                i(packet)
        else:
            if self.callback_reg_warn:
                logger.warn('%s␣−␣please␣register␣a␣callback␣to␣the␣TAP␣(%s,␣%s)' % (self.sim_ip, self.sim_network_ip, self.
                    ↪ sim_network_dev))
                self.callback_reg_warn = False

    def register_callback(self, func):
        if func not in self.__callback_func:
            logger.info('%s␣−␣callback␣to␣the␣TAP␣(%s,␣%s)␣registered' % (self.sim_ip, self.sim_network_ip, self.sim_network_dev)
                ↪ )
            self.__callback_func.append(func)

    def remove_callback(self, func):
        if func in self.__callback_func:
            logger.info('%s␣−␣callback␣to␣the␣TAP␣(%s,␣%s)␣removed' % (self.sim_ip, self.sim_network_ip, self.sim_network_dev))
            self.__callback_func.remove(func)

    def send(self, content):
        try:
            os.write(self.fd, str(content))
        except:
            logger.error('%s␣−␣Sending␣to␣SlowControl␣FAILED!' % (self.sim_ip))

    def close_interface(self):
        self.alive = False
        os.system('ping␣−c␣1␣−W␣1␣%s␣&>␣/dev/null␣&' % (self.sim_ip))
        logger.info('%s␣−␣TAP␣adapter␣(%s,␣%s)␣stopped' % (self.sim_ip, self.sim_network_ip, self.sim_network_dev))


class Pcap_Generator(LayerBase):

    def __init__(self):

        if not self._layerInit:
            super(Pcap_Generator, self).initLayer()

        self.pcap_init_done = True

        # global pcap header, fixed format
        global_header = ('d4␣c3␣b2␣a1␣' # magic number
                         '02␣00␣' # major version number
                         '04␣00␣' # File format minor revision (i.e. pcap 2.<4>)
                         '00␣00␣00␣00␣' # GMT to local correction
                         '00␣00␣00␣00␣' # accuracy of timestamps
                         'ff␣ff␣00␣00␣' # max length of captured packets, in octets
                         '01␣00␣00␣00␣') # data link type
        self.pcap_global_header = self.CH(global_header)

        # dynamic pcap header
        timestamp_start = time.time()
        self.timestamp_start_u = int(timestamp_start * 1000000)
        self.timestamp_start_hex = self.int2hex(timestamp_start)
    def __call__(self, layer2_packet):
        dt_now_hex = self.int2hex(int(time.time() * 1000000) - self.timestamp_start_u)
        packet_size = self.int2hex(len(layer2_packet))
        pcap_packet_hex = self.timestamp_start_hex + dt_now_hex + packet_size + packet_size
        if self.pcap_init_done:
            return pcap_packet_hex + self.S2H(layer2_packet)
        else:
            self.pcap_init_done = True
            return self.pcap_global_header + pcap_packet_hex + self.S2H(layer2_packet)

    def int2hex(self, ts_str):
        ts = self.I2H(int(ts_str))
        return str(self.LE(ts) + '00000000')[:8]


class Routing_Packets(ThreadBase):
    """
    bidirectional interfacing of VirtNet and Sockets
    current tasks: routing and filter packets
    """

    def __init__(self, sim_ip, ctrl_id, sim_id, virtnet, send_to_socket):
        self.sim_ip = sim_ip
        self.ctrl_id = ctrl_id
        self.sim_id = sim_id
        self.virtnet = virtnet
        self.send_to_socket = send_to_socket

        if not self._layerInit:
            super(Routing_Packets, self).initLayer()

        self.sim_network_ip = self.virtnet.sim_network_ip
        self.sim_network_dev = self.virtnet.sim_network_dev
        self.ips = [self.sim_ip, self.sim_network_ip]
        self.ips_hex = [self.IP2H(self.sim_ip), self.IP2H(self.sim_network_ip)]

        self.send_to_tap = self.virtnet.send
        self.virtnet.register_callback(self.receive_from_tap)
        self.write_counter = 0

        self.PC = Pcap_Generator()

        logger.info('%s␣-␣router␣started' % (self.sim_ip))
        ThreadBase.__init__(self, name=self.__class__.__name__, service=self.__service__, exit=self.__exit__)

    def __service__(self):
        self.alive = True
        logger.info('%s␣-␣router␣service␣started' % (self.sim_ip))
        while self.alive:
            time.sleep(1)

    def filter_hex(self, hex_msg):
        valid = True
        if hex_msg[24:28] == '0800':
            if hex_msg[60:68] in self.ips_hex:
                pass
            else:
                valid = False
        elif hex_msg[24:28] == '0806':
            pass
        elif hex_msg[24:28] == '0835':
            pass
        else:
            valid = False
        return valid
    def receive_from_tap(self, msg):
        hex_msg = self.S2H(msg)
        if self.filter_hex(hex_msg):
            hex_pcap = self.PC(msg)
            try:
                self.send_to_socket(target=self.ctrl_id, request='pcap_packet', content=hex_pcap)
            except:
                logger.error('%s␣−␣send␣TAP␣message␣to␣Ctrl␣socket␣failed' % (self.sim_ip))
            try:
                self.send_to_socket(target=self.sim_id, request='tap_request', content=hex_msg)
            except:
                logger.error('%s␣−␣send␣TAP␣message␣to␣Sim␣socket␣failed' % (self.sim_ip))

    def receive_from_sim(self, hex_msg):
        if self.filter_hex(hex_msg):
            str_msg = self.H2S(hex_msg)
            hex_pcap = self.PC(str_msg)
            try:
                self.send_to_socket(target=self.ctrl_id, request='pcap_packet', content=hex_pcap)
            except:
                logger.error('%s␣−␣send␣Sim␣message␣to␣Ctrl␣socket␣failed' % (self.sim_ip))
            try:
                self.virtnet.send(str_msg)
            except:
                logger.error('%s␣−␣send␣Sim␣message␣to␣TAP␣failed' % (self.sim_ip))

    def __exit__(self):
        self.alive = False
        logger.info('%s␣−␣router␣stopped' % (self.sim_ip))

    def close_router(self):
        self.virtnet.remove_callback(self.receive_from_tap)
        self.__exit__()


class Session_Management(object):
    """
    collects all clients and looking for matching,
    manage all sessions due to every two matching clients (ctrl and sim),
    contol one router for every session

    example for two matching clients:

    clients:
        {'127.0.0.1:49670': {'client_ip': '10.0.0.2', 'client_type': 'ctrl'},
         '127.0.0.1:49680': {'client_ip': '10.0.0.2', 'client_type': 'sim'}}
    sessions:
        {'10.0.0.2': {'ctrl': '127.0.0.1:49670',
                      'sim': '127.0.0.1:49680',
                      'virtnet': <__main__.VirtNet_Adapter object at 0x18bb490>}}
    router:
        {'10.0.0.2': <__main__.Routing_Packets object at 0x18bb650>}
    """

    def __init__(self, send_to_socket):
        self.send_to_socket = send_to_socket
        self.clients = {}
        self.sim_sessions = {}
        self.router_sessions = {}

    def add_client(self, client_id):
        if client_id not in self.clients.keys():
            self.clients[client_id] = {'client_type': None,
                                       'client_ip': None}
```

```
def remove_client(self, client_id):
    if client_id in self.clients.keys():
        client_ip = self.clients[client_id]['client_ip']
        client_type = self.clients[client_id]['client_type']
        if client_ip in self.sim_sessions.keys():
            if len(self.sim_sessions[client_ip]) == 2 and client_type in self.sim_sessions[client_ip].keys() and self.sim_sessions[
                ↪ client_ip][client_type] == client_id:
                self.virtnet_remove(client_ip)
                del self.sim_sessions[client_ip]
                logger.info('%s␣−␣first␣client␣(%s,␣%s)␣disconnected' % (client_ip, client_id, client_type))
            elif len(self.sim_sessions[client_ip]) == 3 and client_type in self.sim_sessions[client_ip].keys() and self.sim_sessions[
                ↪ client_ip][client_type] == client_id:
                self.router_remove(client_ip)
                del self.sim_sessions[client_ip][self.clients[client_id]['client_type']]
                logger.info('%s␣−␣last␣client␣(%s,␣%s)␣disconnected' % (client_ip, client_id, client_type))
        del self.clients[client_id]

def set_client_type(self, client_id, client_type):
    self.clients[client_id]['client_type'] = client_type

def set_client_ip(self, client_id, client_ip):
    self.clients[client_id]['client_ip'] = client_ip
    accept = False
    return_msg = ''
    if client_ip not in self.sim_sessions.keys():
        subnet_req = client_ip.split('.')[2]
        subnet_check = True
        subnet_used = ''
        for i in self.sim_sessions.keys():
            subnet_used += '␣%s' % i
            subnet = i.split('.')[2]
            if subnet == subnet_req:
                subnet_check = False

        if subnet_check == False:
            accept = False
            return_msg = 'Sorry,␣this␣network␣is␣already␣in␣use.␣(current␣used:%s)' % subnet_used
        else:
            self.sim_sessions[client_ip] = {self.clients[client_id]['client_type']: client_id}
            VNalive = self.virtnet_add(client_ip)
            if VNalive:
                accept = True
                return_msg = 'create␣new␣session␣with␣sim␣IP␣%s' % client_ip
                logger.info('%s␣−␣first␣client␣(%s,␣%s)␣connected' % (client_ip, client_id, self.clients[client_id]['client_type']))
            else:
                del self.sim_sessions[client_ip]
                accept = False
                dev = 'tap%s' % (client_ip.replace(".", ""))
                return_msg = 'Network␣%s␣is␣already␣UP,␣if␣you␣have␣disconneted␣the␣last␣client␣in␣the␣last␣seconds,␣
                    ↪ please␣wait␣a␣few␣more␣seconds␣until␣it␣has␣descended' % (dev)

    elif client_ip in self.sim_sessions.keys():
        if len(self.sim_sessions[client_ip].keys()) == 2:
            if self.clients[client_id]['client_type'] in self.sim_sessions[client_ip].keys():
                return_msg = 'Sorry␣not␣possible,␣%s␣Client␣is␣already␣connected' % self.clients[client_id]['client_type']
                self.remove_client(client_id)
            elif self.clients[client_id]['client_type'] not in self.sim_sessions[client_ip].keys():
                self.sim_sessions[client_ip][self.clients[client_id]['client_type']] = client_id
                accept = True
                return_msg = 'connection␣accepted,␣ready␣to␣rumble'
                logger.info('%s␣−␣second␣client␣(%s,␣%s)␣connected' % (client_ip, client_id, self.clients[client_id]['client_type'
                    ↪ ]))
                self.router_add(client_ip)
        elif len(self.sim_sessions[client_ip].keys()) == 3:
            return_msg = 'Sorry␣not␣possible,␣all␣Clients␣connected'
    return accept, return_msg
    def virtnet_add(self, sim_ip):
        VN = VirtNet_Adapter(sim_ip)
        if VN.alive:
            self.sim_sessions[sim_ip]['virtnet'] = VN
        return VN.alive

    def virtnet_remove(self, sim_ip):
        self.sim_sessions[sim_ip]['virtnet'].close_interface()

    def router_add(self, sim_ip):
        self.router_sessions[sim_ip] = Routing_Packets(sim_ip,
                                                       self.sim_sessions[sim_ip]['ctrl'],
                                                       self.sim_sessions[sim_ip]['sim'],
                                                       self.sim_sessions[sim_ip]['virtnet'],
                                                       self.send_to_socket)

    def router_remove(self, sim_ip):
        self.router_sessions[sim_ip].close_router()
        del self.router_sessions[sim_ip]

    def get_partner_id(self, partner, source):
        client_ip = self.clients[source]['client_ip']
        if partner in self.sim_sessions[client_ip]:
            return self.sim_sessions[client_ip][partner]
        else:
            return None

    def get_session_id(self, source):
        ip = self.clients[source]['client_ip']
        return ip

    def get_router(self, source):
        ip = self.get_session_id(source)
        if ip in self.router_sessions.keys():
            return self.router_sessions[ip]
        else:
            return None

    def send_to_virtnet(self, source, msg):
        router = self.get_router(source)
        if router:
            router.receive_from_sim(msg)

    def print_dicts(self, msg):
        print 20 * '-' + msg + '\n' + 'clients:'
        pprint(self.clients)
        print 'sessions:'
        pprint(self.sim_sessions)
        print 'router:'
        pprint(self.router_sessions)
        print 20 * '-' + '\n'


class SocketServer(ThreadBase):
    """
    Handle bidirectional socket communication.
    """

    def __init__(self, port=7890):
        self.port = port
        ThreadBase.__init__(self, name=self.__class__.__name__, service=self.__service__, exit=self.__exit__)

    def __service__(self):
        self.id = 0
        self.server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.server_socket.bind(("0.0.0.0", self.port))
        self.server_socket.listen(10)

        self.connections = {'server': self.server_socket}

        self.SM = Session_Management(self.send)

        self.alive = True
        while self.alive:
            read_sockets, write_sockets, error_sockets = select.select(self.connections.values(), [], [])
            for sock in read_sockets:
                # init new connection
                if sock == self.server_socket:
                    sockfd, addr = self.server_socket.accept()
                    client_id = str(addr[0]) + ':' + str(addr[1])
                    self.connections[client_id] = sockfd
                    logger.info('Client␣%s␣connected' % (client_id))
                    self.send(target=client_id, request='init_client', content=client_id)
                    continue
                # get data from client
                else:
                    try:
                        data = sock.recv(2**16)
                        msg = literal_eval(data)
                        self.parse_sock_msg(msg)
                    except:
                        if sock in self.connections.values():
                            sock_name = self.connections.keys()[self.connections.values().index(sock)]
                            logger.info('Client␣%s␣disconnected' % (sock_name))
                        self.SM.remove_client(sock_name)
                        del self.connections[sock_name]
                        continue
        self.server_socket.close()

    def __exit__(self):
        self.alive = False

    def parse_sock_msg(self, msg):
        if 'source' in msg.keys() and msg['source'] in self.connections.keys():
            if msg['request'] == 'sim_response':
                self.SM.send_to_virtnet(msg['source'], msg['content'])
            elif msg['request'] == 'ctrl2sim':
                sim_id = self.SM.get_partner_id('sim', msg['source'])
                self.send(target=sim_id, request=msg['request'], content=msg['content'])
            elif msg['request'] == 'set_id':
                self.SM.add_client(msg['content'])
                self.send(target=msg['source'], request='get_type')
            elif msg['request'] == 'set_type':
                self.SM.set_client_type(msg['source'], msg['content'])
                self.send(target=msg['source'], request='get_sim_ip')
            elif msg['request'] == 'set_sim_ip':
                accept, return_msg = self.SM.set_client_ip(msg['source'], msg['content'])
                logger.info('%s␣−␣%s' % (msg['source'], return_msg))
                if accept == True:
                    self.send(target=msg['source'], request='init_done', content=return_msg)
                elif accept == False:
                    self.send(target=msg['source'], request='error', content=return_msg)
            elif msg['request'] == 'get_ctrl_status':
                ctrl_status = self.SM.get_partner_id('ctrl', msg['source'])
                if ctrl_status:
                    self.send(target=msg['source'], request='ctrl_status', content=ctrl_status)

    def send(self, source='server', target=None, request='', content=''):
        if target:
            self.id += 1
            data_out = {'source': source,
                        'target': target,
                        'request': request,
                        'content': content,
                        'id': self.id
                        }
            try:
                self.connections[target].send(json.dumps(data_out))
            except:
                self.connections[target].close()
                del self.connections[target]


class VirtNet_Daemon(ThreadBase):

    def __init__(self):
        self.stdin_path = '/dev/null'
        self.stdout_path = '/dev/tty'
        self.stderr_path = '/dev/tty'
        self.pidfile_path = os.getcwd() + '/virtnet_tmp/VirtNet_Daemon.pid'
        self.pidfile_timeout = 1

    def run(self):
        ThreadBase.init(mode='daemon')
        self.initLayer()
        self.__check_permissions()

        self.SS = SocketServer(port=7890)

        logger.info(50 * '-')
        logger.info('VIRTNET␣DAEMON␣STARTED')
        print "\n␣---␣VirtNet-Daemon␣version␣%s␣(%s)␣started␣---␣\n" % (__version__, __date__)
        while True:
            time.sleep(3600)

    def __check_permissions(self):
        py_version = str(sys.version_info[0]) + '.' + str(sys.version_info[1])
        tools = ['python' + py_version,
                 'ifconfig',
                 'wireshark',
                 'dumpcap']
        paths = []
        for i in tools:
            out = self.C2S('whereis␣-b␣' + i).split('␣')
            if len(out) >= 2:
                for j in out[1:]:
                    if i in j:
                        if '\n' in j:
                            paths.append(j.split('\n')[0])
                        else:
                            paths.append(j)
                        break
            else:
                sys.stderr.write('Please␣install␣%s␣\n' % (i))
                sys.exit(-1)

        if self.C2S('whoami').split('\n')[0] != 'root':
            expected_caps = 'cap_net_admin,cap_net_raw+eip'
            passed = True
            error_msg = ''

            path_getcap = self.C2S('whereis␣-b␣getcap').split('␣')[1].split('\n')[0]
            for i in paths:
                if expected_caps not in self.C2S(path_getcap + '␣' + i):
                    error_msg += 'setcap␣cap_net_raw,cap_net_admin=eip␣%s␣\n' % (i)
                    passed = False
                out = self.C2S('stat␣−c␣\"%a␣%n\"␣' + i)
                if int(out[2]) < 5:
                    out = out[:2] + str(5) + out[3:]
                    error_msg += 'chmod␣%s␣\n' % (out)
                    passed = False
            if not passed:
                sys.stderr.write('\nPlease␣check␣network␣capabilities␣or␣file␣permissions\n\n')
                sys.stderr.write('You␣can␣set␣it␣as␣root␣user:\n')
                sys.stderr.write('%s' % (error_msg))
                sys.exit('\nPress␣Enter␣to␣close')


if __name__ == "__main__":

    VND = VirtNet_Daemon()
    logger = logging.getLogger("VirtNet")
    logger.setLevel(logging.INFO)
    handler = logging.FileHandler(os.getcwd() + '/virtnet_tmp/VirtNet_Daemon.log', mode='w')
    handler.setFormatter(logging.Formatter("%(asctime)s␣−␣%(name)s␣−␣%(levelname)s␣−␣%(message)s"))
    logger.addHandler(handler)

    daemon_runner = runner.DaemonRunner(VND)
    daemon_runner.daemon_context.files_preserve = [handler.stream]
    daemon_runner.do_action()
```

A.2.2 VirtNet simulation binding

```
#!/usr/bin/python
# −*− coding: utf−8 −*−

"""
−−−VirtualNetworkHandler−Simulation−Client−−−

author: Yves Bianga
email: yves.bianga@cern.ch
version: 2.0
"""

from pprint import pprint
from ast import literal_eval
import socket
import select
import sys
import os
import json
import signal
import time
import stat
import Queue
import time
import multiprocessing
sys.dont_write_bytecode = True
from VirtNet_Base import *

# required for simulation flow
from testbench import *
from cocotb import log
from cocotb.triggers import *
from cocotb.clock import Clock
from driver_avalon_latome import AvalonSTPkts as AvalonSTDriver
from monitor_avalon_latome import AvalonSTPkts as AvalonSTMonitor
```

```
class SocketClient(ThreadBase):

    def __init__(self, sim_ip, host):

        self.config = {'client_type': 'sim',
                       'client_id': None,
                       'client_status': None,
                       'sim_ip': sim_ip,
                       'host_ip': host,
                       'host_port': 7890,
                       'ctrl_status': None
                       }

        self.queue = multiprocessing.Queue()

        if not self._layerInit:
            super(SocketClient, self).initLayer()
        ThreadBase.__init__(self, name=self.__class__.__name__, service=self.__service__, exit=self.__exit__)

    def __service__(self):

        try:
            self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.s.settimeout(2)
            self.s.connect((self.config['host_ip'], self.config['host_port']))
        except:
            print 'Unable␣to␣connect␣to␣server'
            sys.exit()

        self.socket_list = [self.s]

        self.alive = True
        while self.alive:
            read_sockets, write_sockets, error_sockets = select.select(self.socket_list, [], [])

            for sock in read_sockets:
                # incoming message from remote server
                if sock == self.s:
                    data = sock.recv(2**16)
                    if not data:
                        print '\nDisconnected␣from␣server'
                        sys.exit()
                    else:
                        msg_list = data.split('}')
                        for i in msg_list:
                            if '{' in i:
                                msg = literal_eval(i + '}')
                                self.parse_sock_msg(msg)

    def __exit__(self):
        self.alive = False

    def parse_sock_msg(self, msg):
        # print 'receive msg from server', msg
        # pprint(msg)
        if 'source' in msg.keys() and msg['source'] == 'server':
            if msg['request'] == 'tap_request':
                self.queue.put('TAP' + self.H2S(msg['content']))
            elif msg['request'] == 'ctrl2sim':
                self.queue.put(msg['content'])
            elif self.config['client_id'] == None and msg['request'] == 'init_client':
                self.config['client_id'] = msg['target']
                self.send(request='set_id', content=self.config['client_id'])
                self.config['client_status'] = 'set_id'
            elif msg['request'] == 'get_type':
                self.send(request='set_type', content=self.config['client_type'])
                self.config['client_status'] = 'set_type'
            elif msg['request'] == 'get_sim_ip':
                self.send(request='set_sim_ip', content=self.config['sim_ip'])
                self.config['client_status'] = 'set_sim_ip'
            elif msg['request'] == 'init_done':
                self.config['client_status'] = 'connected'
            elif msg['request'] == 'ctrl_status':
                self.config['ctrl_status'] = msg['content']
            elif msg['request'] == 'error':
                print msg['content']
                sys.exit()

    def send(self, request=None, content=''):
        if request:
            data_out = {'source': self.config['client_id'],
                        'target': 'server',
                        'request': request,
                        'content': content
                        }
            try:
                self.s.send(json.dumps(data_out))
            except:
                print 'Unable␣to␣communicate␣with␣server'
                self.s.close()
                sys.exit()


class VirtualNetwork(LayerBase):

    def VN_config(self, sim_ip='10.0.0.2', host='0.0.0.0'):

        ThreadBase.init(mode='client')
        if not self._layerInit:
            super(VirtualNetwork, self).initLayer()

        self.SC = SocketClient(sim_ip=sim_ip, host=host)
        self.isAlive = True
        return self.SC.queue.put

    def abort_loop(self):
        self.isAlive = False

    def print_msg(self, msg):
        print 200 * '*' + 2 * '\n' + str(msg) + 2 * '\n' + 200 * '*'

    @cocotb.coroutine
    def check_ctrl_client(self):
        yield FallingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
        self.SC.send(request='get_ctrl_status')
        time.sleep(0.5)
        if not self.SC.config['ctrl_status']:
            self.print_msg('VirtNet␣doesn\'t␣work␣without␣a␣connected␣control−client.␣Please␣start␣the␣VirtNet_Control.py␣
                ↪ now.␣(\"$:␣./VirtNet_Control.py␣%s\")' % (self.SC.config['sim_ip']))
        yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)

    @cocotb.coroutine
    def VN_start(self, init_reset_delay=50e−6):
        """
        configure the firmware
        """
        # 125Mhz clock
        TBClockGenerator(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o, 125000000.0, "lli_ipctrl_gbe_link_c_gbe_125_clk_o:␣125MHz")
        TBWireNot(self.dut.reset_n, self.dut.lli_ipctrl_gbe_link_c_gbe_125_rst_o)

        self.stream_in = AvalonSTDriver(self.dut, "lli_ipctrl_gbe_link_c_rx_data_st", self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
        self.stream_out = AvalonSTMonitor(self.dut, "lli_ipctrl_gbe_link_c_tx_data_st", self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)

        yield Timer((init_reset_delay) / 1e−12) # 50 us, enough time for reset to be effective
        self.dut.lli_ipctrl_gbe_link_c_tx_data_st_ready_o <= 1

        yield Join(cocotb.fork(self.__main_loop()))

    @cocotb.coroutine
    def __main_loop(self):
        """
        simulation use a blocking mainloop as default to communicate with the socket adapter
        """
        yield Join(cocotb.fork(self.check_ctrl_client()))

        self.tap_counter = 0
        self.unblock_timer = 0
        self.unblock_timer_max = 0
        self.bypass_lock = False
        self.print_msg('Start␣loop')
        yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
        while self.isAlive:
            packet = self.SC.queue.get()

            if 'TAP' == packet[:3]:
                self.tap_counter += 1
                print 100 ∗ '#' + '␣request␣␣#%s␣received␣' % str(self.tap_counter)
                self.stream_in.append(packet[3:])
                result = yield self.stream_out.wait_for_recv()
                self.SC.send(request='sim_response', content=self.S2H(result))
                print 100 ∗ '#' + '␣response␣#%s␣transmitted␣' % str(self.tap_counter)
                yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
            elif 'unblocktimer' == packet[:12]:
                self.unblock_timer += 8
                if self.unblock_timer < self.unblock_timer_max:
                    self.SC.queue.put('unblocktimer')
                    yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
            elif 'block' == packet[:5]:
                qsize = int(self.SC.queue.qsize())
                for i in range(qsize):
                    self.SC.queue.get()
            elif 'unblock' == packet[:7]:
                self.SC.queue.put('unblock')
                yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
            elif 'abort' == packet[:5]:
                raise KeyboardInterrupt('␣')
            elif 'bypass' == packet[:6]:
                if self.bypass_lock == False and self.bypass_function:
                    self.bypass_lock = True
                    self.SC.queue.put('unblock')
                    yield Join(cocotb.fork(self.bypass_function()))
                elif self.bypass_lock == True:
                    self.print_msg('BYPASS−DONE')
                    self.bypass_lock = False
                    self.SC.queue.put('block')
                    yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
            elif 'run_' == packet[:4]:
                self.unblock_timer = 0
                self.unblock_timer_max = int(packet[4:]) ∗ 1000
                self.SC.queue.put('unblocktimer')
            else:
                self.print_msg(packet)
                yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)
```

A.2.3 VirtNet control client

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""
---VirtualNetworkHandler-Control-Client---

This is a client to control a cocotb-IPBus simulation by using the VirtualNetworkHandler daemon.

author: Yves Bianga
email: yves.bianga@cern.ch
version: 2.0
"""

from pprint import pprint
from ast import literal_eval
import socket
import select
import sys
import os
import json
import signal
import subprocess
import time
import stat
sys.dont_write_bytecode = True
from virtnet_deps.VirtNet_Base import *


class Cli(object):
    """
    contain all command line output
    """

    def __init__(self):
        self.help()

    def prompt(self):
        sys.stdout.write('\n\n>>>␣')
        sys.stdout.flush()

    def help(self):
        sys.stdout.write('\033[48;0;0;0m\033[92m' +
                         '\n' + 50 * '#' + '\nTYPE:\n' +
                         '␣"' + self.rb('h') + '"␣␣␣␣for␣this␣' + self.rb('h') + 'elp␣output\n' +
                         '␣"' + self.rb('1') + '"␣␣␣␣BLOCKING␣MODE␣(default)\n' +
                         '␣"' + self.rb('2') + '"␣␣␣␣UNBLOCKING␣MODE\n' +
                         '␣"' + self.rb('3-X') + '"␣␣UNBLOCK␣SIMULATION␣FOR␣X␣MICROSECONDS\n' +
                         '␣"' + self.rb('4') + '"␣␣␣␣CALL␣BYPASS␣FUNCTION␣FROM␣USER␣TESTBENCH\n' +
                         '␣"' + self.rb('5') + '"␣␣␣␣STOP␣SIMULATION,␣DROP␣CONNECTION\n' +
                         '␣"' + self.rb('w') + '"␣␣␣␣start␣' + self.rb('w') + 'ireshark\n' +
                         '␣"' + self.rb('q') + '"␣␣␣␣' + self.rb('q') + 'uit␣wireshark\n' +
                         '␣"' + self.rb('c') + '"␣␣␣␣to␣show␣current␣' + self.rb('c') + 'onfiguration\n' +
                         '␣"' + self.rb('x') + '"␣␣␣␣to␣e' + self.rb('x') + 'it␣the␣client\n' +
                         'FOLLOWED␣BY␣ENTER\n' +
                         50 * '#')
        self.prompt()

    def config(self, config):
        sys.stdout.write('\n' +
                         'client␣id:␣' + str(config['client_id']) + '\n' +
                         'client␣type:␣' + str(config['client_type']) + '\n' +
                         'connection␣status:␣' + str(config['client_status']) + '\n' +
                         'simulation␣ip:␣' + str(config['sim_ip']) + '\n' +
                         'server␣ip:␣' + str(config['host_ip']) + '\n' +
                            'server␣port:␣' + str(config['host_port']))
        self.prompt()

    def rb(self, char):
        return '\033[31m\033[1m%s\033[0m\033[48;0;0;0m\033[92m' % char

    def abort(self):
        sys.stdout.write('\nstop␣simulation,␣drop␣sim−connection')
        self.prompt()

    def bypass(self):
        sys.stdout.write('\ncalling␣bypass␣testbench␣from␣user␣testbench')
        self.prompt()

    def stop_blocking(self):
        sys.stdout.write('\nswitch␣to␣unblocking␣mode')
        self.prompt()

    def start_blocking(self):
        sys.stdout.write('\nswitch␣to␣blocking␣mode')
        self.prompt()

    def run(self, us):
        sys.stdout.write('\nunblock␣simulation␣for␣%s␣microseconds' % us)
        self.prompt()

    def wireshark_start(self):
        sys.stdout.write('\nstarting␣wireshark')
        self.prompt()

    def wireshark_stop(self):
        sys.stdout.write('\nstopping␣wireshark')
        self.prompt()

    def other(self, msg):
        sys.stdout.write('\nVirtNet␣says:\n␣%s␣' % (msg))
        self.prompt()

    def nomatch(self, cli):
        sys.stdout.write('\nI␣don\'t␣get␣it␣(\'%s\'),␣try␣again␣or␣type␣\'h\'' % (cli))
        self.prompt()

    def exit(self):
        sys.stdout.write('\nEXIT␣VIRTNET␣CLIENT\n')
        sys.exit(−1)


class Wireshark(ThreadBase):
    """
    start, stop and manage Wireshark GUI instance,
    provide fifo management, a fifo is used as virtual input device for Wireshark
    """

    def __init__(self, dev):
        self.dev = dev

        if not self._layerInit:
            super(Wireshark, self).initLayer()
        ThreadBase.__init__(self, name=self.__class__.__name__, service=self.__service__, exit=self.__exit__)

    def __service__(self):
        self.fifo_write_state = 'off'
        self.wireshark_state = 'off'
        self.wireshark_process = None
        self.alive = True
        while self.alive:
```

```
        if self.fifo_write_state == 'off':
            self.fifo_start()
        if self.wireshark_state == 'on':
            if self.wireshark_process:
                if not self.is_running(self.wireshark_process.pid):
                    self.gui_close()
            time.sleep(0.5)
        else:
            time.sleep(2)

def __exit__(self):
    self.gui_close()
    self.fifo_close()
    self.alive = False

def gui_start(self):
    if self.wireshark_state == 'off':
        self.wireshark_process = subprocess.Popen('wireshark␣−k␣−S␣−i␣%s' % (self.dev), stdout=subprocess.PIPE, shell=True,
              ↪ preexec_fn=os.setsid)
        time.sleep(3)
        self.wireshark_state = 'init'

def gui_close(self):
    if self.wireshark_process:
        os.killpg(os.getpgid(self.wireshark_process.pid), signal.SIGTERM)
    self.wireshark_state = 'off'
    self.wireshark_process = None

def fifo_start(self):

    #subprocess.check_call('mkfifo %s &> /dev/null' % (self.dev), shell=True)
    try:
        subprocess.check_call('mkfifo␣%s␣&>␣/dev/null' % (self.dev), shell=True)
    except:
        pass
    self.fifo = open(self.dev, 'w')
    self.fifo_write_state = 'opened'

def fifo_feed(self, msg):
    try:
        if self.wireshark_state == 'on':
            self.fifo.write(msg)
            self.fifo.flush()
        elif self.wireshark_state == 'init':
            self.wireshark_state = 'on'
            self.fifo.write(self.init_pcap() + msg)
            self.fifo.flush()
        else:
            pass
    except:
        self.fifo_close()

def fifo_close(self):
    self.fifo_write_state = 'off'
    try:
        self.fifo.close()
    except:
        pass
    try:
        subprocess.check_call('rm␣%s' % (self.dev), shell=True)
    except:
        pass

def init_pcap(self):
    global_header = ('d4␣c3␣b2␣a1␣' # magic number
                     '02␣00␣' # major version number
```

```
                        '04␣00␣' # File format minor revision (i.e. pcap 2.<4>)
                        '00␣00␣00␣00␣' # GMT to local correction
                        '00␣00␣00␣00␣' # accuracy of timestamps
                        'ff␣ff␣00␣00␣' # max length of captured packets, in octets
                        '01␣00␣00␣00␣') # data link type
        return self.H2S(self.CH(global_header))

    def is_running(self, pid):
        if os.path.isdir('/proc/{}'.format(pid)):
            return True
        return False

    def exit(self):
        self.__exit__()


class SocketClient(ThreadBase):
    """
    handle connection to VirtNet-Daemon,
    contain and parse I/O logic
    """

    def __init__(self, sim_ip, host):
        self.config = {'client_type': 'ctrl',
                       'client_id': None,
                       'client_status': None,
                       'sim_ip': sim_ip,
                       'host_ip': host,
                       'host_port': 7890
                       }

        if not self._layerInit:
            super(SocketClient, self).initLayer()
        ThreadBase.__init__(self, name=self.__class__.__name__, service=self.__service__, exit=self.__exit__, daemon=False)

    def __service__(self):

        try:
            self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self.s.settimeout(2)
            self.s.connect((self.config['host_ip'], self.config['host_port']))
        except:
            print 'Unable␣to␣connect␣to␣server'
            sys.exit()

        sig_pipe = ThreadBase.get_pipe()

        self.CLI = Cli()
        self.WS = Wireshark('virtnet_tmp/%s.fifo' % (self.config['sim_ip']))

        self.alive = True
        while self.alive:
            read_sockets, write_sockets, error_sockets = select.select([sys.stdin, self.s, sig_pipe], [], [])
            for sock in read_sockets:
                # incoming message from remote server
                if sock == self.s:
                    data = sock.recv(2**16)
                    if not data:
                        print '\nDisconnected␣from␣server'
                        sys.exit()
                    else:
                        msg_list = data.split('}')
                        for i in msg_list:
                            if '{' in i:
                                msg = literal_eval(i + '}')
                                self.parse_sock_msg(msg)
                # user entered a message
                elif sock == 4:
                    self.__exit__()
                else:
                    msg = sys.stdin.readline()
                    self.parse_cli_msg(msg)

def __exit__(self):
    self.alive = False

def parse_sock_msg(self, msg):
    if 'source' in msg.keys() and msg['source'] == 'server':
        # init connection
        if self.config['client_id'] == None and msg['request'] == 'init_client':
            self.config['client_id'] = msg['target']
            self.send(request='set_id', content=self.config['client_id'])
            self.config['client_status'] = 'set_id'
        elif msg['request'] == 'get_type':
            self.send(request='set_type', content=self.config['client_type'])
            self.config['client_status'] = 'set_type'
        elif msg['request'] == 'get_sim_ip':
            self.send(request='set_sim_ip', content=self.config['sim_ip'])
            self.config['client_status'] = 'set_sim_ip'
        elif msg['request'] == 'init_done':
            self.config['client_status'] = 'connected'
        elif msg['request'] == 'print_msg':
            self.CLI.other(msg['content'])
        elif msg['request'] == 'pcap_packet':
            hex_content = msg['content']
            self.WS.fifo_feed(self.H2S(hex_content))
        elif msg['request'] == 'error':
            print msg['content']
            sys.exit()

def parse_cli_msg(self, cli):
    if cli == '\n':
        self.CLI.prompt()
    elif cli == 'h\n':
        self.CLI.help()
    elif cli == '1\n':
        self.CLI.start_blocking()
        self.send(request='ctrl2sim', content='block')
    elif cli == '2\n':
        self.CLI.stop_blocking()
        self.send(request='ctrl2sim', content='unblock')
    elif cli[:2] == '3-':
        try:
            us = int(cli.split('\n')[0][2:])
            self.CLI.run(us)
            self.send(request='ctrl2sim', content='run_%s' % (us))
        except:
            self.CLI.nomatch(cli.split('\n')[0])
    elif cli == '5\n':
        self.CLI.abort()
        self.send(request='ctrl2sim', content='abort')
    elif cli == '4\n':
        self.CLI.bypass()
        self.send(request='ctrl2sim', content='bypass')
    elif cli == 'w\n':
        self.CLI.wireshark_start()
        self.WS.gui_start()
    elif cli == 'q\n':
        self.CLI.wireshark_stop()
        self.WS.gui_close()
    elif cli == 'c\n':
        self.CLI.config(self.config)
        elif cli == 'x\n':
            self.__exit__()
        else:
            self.CLI.nomatch(cli.split('\n')[0])

    def send(self, request=None, content=''):
        if request:
            data_out = {'source': self.config['client_id'],
                        'target': 'server',
                        'request': request,
                        'content': content
                        }
            try:
                self.s.send(json.dumps(data_out))
            except:
                print 'Unable␣to␣communicate␣with␣server'
                self.s.close()
                sys.exit()


if __name__ == "__main__":

    if len(sys.argv) == 2:
        sim_ip = sys.argv[1]
        sim_host = '0.0.0.0'
    elif len(sys.argv) == 3:
        sim_ip = sys.argv[1]
        sim_host = sys.argv[2]
    else:
        sys.stdout.write('\nPlease␣call:␣./VirtNet_Control.py␣\'simulation_IP\'␣(\'simulation_host_IP\')\n\n')
        sys.stdout.flush()
        sys.exit()

    ThreadBase.init(mode='client')
    SC = SocketClient(sim_ip=sim_ip, host=sim_host)
```

A.2.4 VirtNet base classes

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

"""
---VirtualNetworkHandler base-classes---

author: Yves Bianga
email: yves.bianga@cern.ch
version: 2.0
"""

from threading import Thread
import string
import sys
import os
import json
import signal
import subprocess
import atexit
import fcntl
sys.dont_write_bytecode = True


class LayerBase(object):
```

```
"""
all none−specific layer features
"""

_layerInit = False

def initLayer(self, *args, **kwargs):
    if not self._layerInit:
        self._layerInit = True
        self.H2S = self.__hex_to_string
        self.S2H = self.__string_to_hex
        self.I2H = self.__int_to_hex
        self.IP2H = self.__ip_to_hex
        self.H4P = self.__hex_for_pcap
        self.C2S = self.__communicate_to_system
        self.LE = self.__little_endian
        self.CH = self.__concentrate_hex

def __string_to_hex(self, s):
    lst = []
    for ch in s:
        hv = hex(ord(ch)).replace('0x', '')
        if len(hv) == 1:
            hv = '0' + hv
        lst.append(hv)
    return reduce(lambda x, y: x + y, lst)

def __hex_to_string(self, h):
    return h and chr(string.atoi(h[:2], base=16)) + self.__hex_to_string(h[2:]) or ''

def __int_to_hex(self, i):
    return format(int(i), '02x')

def __ip_to_hex(self, ip):
    ip_list = ip.split('.')
    hex_string = ''
    for i in ip_list:
        hex_string += self.__int_to_hex(i)
    return hex_string

def __hex_for_pcap(self, hex_str):
    hex_str_split = [a + b for a, b in zip(hex_str[::2], hex_str[1::2])]
    hex_str_formated = ''
    i, k = 0, 0
    for hex_pair in hex_str_split:
        if i == 0:
            hex_str_formated += '%s0␣' % (hex(k)[2:].zfill(3))
            k += 1
        i += 1
        if i == 16:
            hex_str_formated += hex_pair + '\n'
            i = 0
        else:
            hex_str_formated += hex_pair + '␣'
    return hex_str_formated

def __communicate_to_system(self, cmd):
    proc = subprocess.Popen(cmd, stdout=subprocess.PIPE, shell=True)
    (output, err) = proc.communicate()
    return output.decode('utf−8')

def __little_endian(self, hex_str):
    hex_str = ''.join(hex_str.split('␣'))
    if len(hex_str) % 2:
        hex_str = '0' + hex_str
    hex_str_split = [a + b for a, b in zip(hex_str[::2], hex_str[1::2])]
```

```
        return ''.join(hex_str_split[::-1])

    def __concentrate_hex(self, hex_str):
        return ''.join(hex_str.split('␣'))


class ThreadBase(LayerBase):
    """
    Manage starting and stopping of Threads
    """

    pipe_r, pipe_w = os.pipe()

    @classmethod
    def init(cls, mode=None):
        signal.signal(signal.SIGINT, cls.sigHandler)
        signal.signal(signal.SIGTERM, cls.sigHandler)
        if mode == 'client':
            flags = fcntl.fcntl(cls.pipe_w, fcntl.F_GETFL, 0)
            flags = flags | os.O_NONBLOCK
            fcntl.fcntl(cls.pipe_w, fcntl.F_SETFL, flags)
            signal.set_wakeup_fd(cls.pipe_w)

    @classmethod
    def get_pipe(cls):
        return cls.pipe_r

    @classmethod
    def register(cls, exitfunc):
        atexit.register(exitfunc)

    @classmethod
    def sigHandler(cls, signo, frame):
        sys.exit(0)

    def __init__(self, *args, **kwargs):
        self.__dict__.update(kwargs)
        self.__thread = Thread(target=self.service, args=())
        if 'daemon' in kwargs.keys() and kwargs['daemon'] == False:
            self.__thread.daemon = False
        else:
            self.__thread.daemon = True
        self.__thread.start()
        self.__class__.register(self.exit)
```

A.2.5 VirtNet implementation example

```
#--------------------------------------------------------
#--! @file fpga_user_tb.py
#--! @brief Module fpga user testbench
#--------------------------------------------------------

from cocotb.triggers import *
from cocotb.clock import Clock
from testbench import *
from lli_cst import *
from VirtNet_Simulation import *


class FPGAUserTestbench(VirtualNetwork):
    def __init__(self, dut):
        self.dut = dut
        self.log = SimLog("cocotb.%s" % (self.__class__.__name__))
        self.log.info("Starting␣FPGA␣user␣testbench")

        # MAC address
        self.log.info("Driving␣MAC␣address␣to:␣%s" % (LLI_GBE_BASE_MAC_ADDRESS + AMC_ID_DEFAULT_VALUE))
        self.dut.lli_ipctrl_mac_address_c_mac_address_o <= LLI_GBE_BASE_MAC_ADDRESS + AMC_ID_DEFAULT_VALUE
        self.dut.lli_ipctrl_mac_address_c_valid_o <= 1

        # init VirtNet simulation client
        self.log.info("Starting␣Virtual␣Network")
        self.input = self.VN_config(sim_ip='10.0.0.2', host='0.0.0.0')
        cocotb.fork(self.VN_start(init_reset_delay=50e−6))

    def get_simulation_timeout(self):
        # Keep original timeout
        return −1

    @cocotb.coroutine
    def bypass_function(self):

        self.log.info(5 ∗ 'Bypass␣test␣')
        yield RisingEdge(self.dut.lli_ipctrl_gbe_link_c_gbe_125_clk_o)

        self.input('bypass_done')
```

B LATOME Firmware Model

B.1 Source Code

B.1.1 Parallelized FW-Model instance

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from datetime import datetime
import multiprocessing as mp
import os
from helper_tools import Helper, Log_Pipe
from ltdb_model import LTDB_Model
from lli_model import LLI_Model
from istage_model import ISTAGE_Model
from remap_model import REMAP_Model
from user_model import USER_Model
from osum_model import OSUM_Model
from fpga_cst import *
from ttc_cst import *
from test_cst import *


# Parameters for debugging output for each submodel
DEBUG_IDS = {'dbg_ltdb_sc_name': 'B_8A_F4',
             'dbg_ltdb_stream': 0,
             'dbg_ltdb_sc_id': None,#7,
             'dbg_lli_stream': 0,
             'dbg_lli_sc_id': None,#7,
             'dbg_istage_stream': 0,
             'dbg_istage_sc_id': None,#7,
             'dbg_remap_search_sc_name': 'B_8A_F4',
             'dbg_remap_stream': 4,
             'dbg_remap_sc_id': None,#5,
             'dbg_user_stream': 4,
             'dbg_user_sc_id': None,#5,
             }

# Logging order for initialization of all submodels, necessary in parallel execution
LOG_ORDER = {'LATOME_FW_Model': 0,
             'LTDB_Model': 1,
             'LLI_Model': 2,
             'ISTAGE_Model': 3,
             'REMAP_Model': 4,
             'USER_Model': 5,
             'OSUM_Model': 6,
             }


class LATOME_FW_Model(Helper):

    def __init__(self):
        # Init Helper as parent (=init globals/logging/debugging/printing, manage output path and reset directory)
        Helper.__init__(self, root=True, debug_ids=DEBUG_IDS, log_order=LOG_ORDER)

        # Check output directory
        self.parse_globals('LATOME_FW_MODEL_DATA_PATH')
        if not os.path.isdir(LATOME_FW_MODEL_DATA_PATH):
            os.mkdir(LATOME_FW_MODEL_DATA_PATH)

        self.loop_depth = 0
        if LATOME_FW_MODEL_LOOP_MODE:
            # The delay should be FIR depth - 1
            self.loop_depth =3

        # Switch between parallel or serial FW-Model execution (serial is for debug-mode)
        if LATOME_FW_MODEL_DEBUG_MODE:
            self.__init_serial()
            self.__run_serial()
        else:
            self.__init_parallel()
            self.__run_parallel()


    ##################################
    # From here all parallel execution related methods
    ##################################
    def __init_parallel(self):
        """
        Threaded FW-model initialization
        """
        # OSUM-event to keep FW-Model instance open until the last frame is done
        self.osum_event = mp.Event()
        # LTDB-event for LTDB-only-mode
        self.ltdb_event = mp.Event()

        if LATOME_FW_MODEL_LTDB_ONLY_MODE:
            self.ltdb_thread = mp.Process(target=self.__ltdb_thread)
        else:
            # Init module threads
            self.ltdb_thread = mp.Process(target=self.__ltdb_thread)
            self.lli_thread = mp.Process(target=self.__lli_thread)
            self.istage_thread = mp.Process(target=self.__istage_thread)
            self.remap_thread = mp.Process(target=self.__remap_thread)
            self.user_thread = mp.Process(target=self.__user_thread)
            self.osum_thread = mp.Process(target=self.__osum_thread)
            # Generate queues for frame handling
            queue_list = ['ltdb_to_lli', 'lli_to_istage', 'istage_to_remap', 'remap_to_user', 'user_to_osum']
            for q in queue_list:
                setattr(self, q, mp.Queue())
        self.cout_init_done()

    def __run_parallel(self):
        """
        Threaded FW-model processing
        """
        # Start module threads
        if LATOME_FW_MODEL_LTDB_ONLY_MODE:
            self.ltdb_thread.start()
            # Blocking FW-Model instance
            self.ltdb_event.wait()
        else:
            self.ltdb_thread.start()
            self.lli_thread.start()
            self.istage_thread.start()
            self.remap_thread.start()
            self.user_thread.start()
            self.osum_thread.start()
            # Blocking FW−Model instance
            self.osum_event.wait()
        self.__del__()

    def __ltdb_thread(self):
        # LTDB_Model process
        LTDB = LTDB_Model()
        counter = 0
        while True:
            if LATOME_FW_MODEL_LTDB_ONLY_MODE:
                _ = LTDB.get_frame()
            else:
                self.ltdb_to_lli.put(LTDB.get_frame())
            counter += 1
            if counter == LATOME_FW_MODEL_EXEC_NB_BCID:
                del LTDB
                break
        self.ltdb_event.set()
        self.ltdb_event.clear()

    def __lli_thread(self):
        # LLI_Model process
        LLI = LLI_Model()
        counter = 0
        while True:
            self.lli_to_istage.put(LLI.feed(self.ltdb_to_lli.get()))
            counter += 1
            if counter == LATOME_FW_MODEL_EXEC_NB_BCID:
                del LLI
                break

    def __istage_thread(self):
        # ISTAGE_Model process
        ISTAGE = ISTAGE_Model()
        counter = 0
        while True:
            self.istage_to_remap.put(ISTAGE.feed(self.lli_to_istage.get()))
            counter += 1
            if counter == LATOME_FW_MODEL_EXEC_NB_BCID:
                del ISTAGE
                break

    def __remap_thread(self):
        # REMAP_Model process
        REMAP = REMAP_Model()
        counter = 0
        while True:
            self.remap_to_user.put(REMAP.feed(self.istage_to_remap.get()))
            counter += 1
            if counter == LATOME_FW_MODEL_EXEC_NB_BCID:
                del REMAP
                break

    def __user_thread(self):
        # USER_Model process
        USER = USER_Model(self.loop_depth)
        counter = 0
        delay_inject_counter = 0
        delay_reinject_counter = 0
        while True:
            if LATOME_FW_MODEL_LOOP_MODE:
                if delay_inject_counter < self.loop_depth:
                    # Process the first 3 BCIDs and dump it at user_model
                    _ = USER.feed(self.remap_to_user.get())
                    self.cout('Injected␣BCID␣%s/%s,␣buffered␣at␣USER_model' % (delay_inject_counter+1, self.loop_depth))
                    delay_inject_counter += 1
                elif counter < LATOME_FW_MODEL_EXEC_NB_BCID − self.loop_depth:
                    # Process all remaining BCIDs from LLI to USER and shift the number of BCID with −3 at USER and OSUM
                    self.user_to_osum.put(USER.feed(self.remap_to_user.get()))
                    counter += 1
                elif counter >= LATOME_FW_MODEL_EXEC_NB_BCID − self.loop_depth:
                    # Process the last 3 BCIDs (use the input of the first 3 BCIDs)
                    self.cout('Loop−reinjection␣of␣BCID␣%s/%s' % (delay_reinject_counter+1, self.loop_depth))
                    self.user_to_osum.put(USER.feed(None))
                    counter += 1
                    delay_reinject_counter += 1
            else:
                self.user_to_osum.put(USER.feed(self.remap_to_user.get()))
                counter += 1

            if counter == LATOME_FW_MODEL_EXEC_NB_BCID:
                del USER
                break

    def __osum_thread(self):
        # OSUM_Model process
        OSUM = OSUM_Model()
        counter = 0
        while True:
            _ = OSUM.feed(self.user_to_osum.get())
            self.cout('Finished␣processing␣of␣BCID␣%d/%d' % (counter, LATOME_FW_MODEL_EXEC_NB_BCID−1))
            counter += 1
            if counter == LATOME_FW_MODEL_EXEC_NB_BCID:
                del OSUM
                break
        self.osum_event.set()
        self.osum_event.clear()


    ##################################
    # From here all serial execution related methods for debugging
    ##################################
    def __init_serial(self):
        """
        Serial FW− model initialization
        """
        self.cout_init_done()
        self.LTDB_Model = LTDB_Model()
        self.LLI_Model = LLI_Model()
        self.ISTAGE_Model = ISTAGE_Model()
        self.REMAP_Model = REMAP_Model()
        self.USER_Model = USER_Model(self.loop_depth)
        self.OSUM_Model = OSUM_Model()

    def __run_serial(self):
        """
        Serial FW− model processing
        """
        self.cout('Running␣model')

        frame_index = 0
        delay_inject_counter = 0
        delay_reinject_counter = 0
        while True:

            if LATOME_FW_MODEL_LOOP_MODE:
                if delay_inject_counter < self.loop_depth:
                    # Process the first 3 BCIDs and dump it at user_model
                    ltdb_frame = self.LTDB_Model.get_frame()
                    lli_frame = self.LLI_Model.feed(ltdb_frame)
                    istage_frame = self.ISTAGE_Model.feed(lli_frame)
                    remap_frame = self.REMAP_Model.feed(istage_frame)
                    _ = self.USER_Model.feed(remap_frame)
                    self.cout(’Injected␣BCID␣%s/%s,␣buffered␣at␣USER_model’ % (delay_inject_counter+1, self.loop_depth))
                    delay_inject_counter += 1
                elif frame_index < LATOME_FW_MODEL_EXEC_NB_BCID − self.loop_depth:
                    # Process all remaining BCIDs from LLI to USER and shift the number of BCID with −3 at USER and OSUM
                    ltdb_frame = self.LTDB_Model.get_frame()
                    lli_frame = self.LLI_Model.feed(ltdb_frame)
                    istage_frame = self.ISTAGE_Model.feed(lli_frame)
                    remap_frame = self.REMAP_Model.feed(istage_frame)
                    user_frame = self.USER_Model.feed(remap_frame)
                    osum_frame = self.OSUM_Model.feed(user_frame)
                    self.cout(’Finished␣processing␣of␣BCID␣%d/%d’ % (frame_index, LATOME_FW_MODEL_EXEC_NB_BCID−1))
                    frame_index += 1
                elif frame_index >= LATOME_FW_MODEL_EXEC_NB_BCID − self.loop_depth:
                    # Process the last 3 BCIDs (use the input of the first 3 BCIDs)
                    self.cout(’Loop−reinjection␣of␣BCID␣%s/%s’ % (delay_reinject_counter+1, self.loop_depth))
                    user_frame = self.USER_Model.feed(None)
                    osum_frame = self.OSUM_Model.feed(user_frame)
                    self.cout(’Finished␣processing␣of␣BCID␣%d/%d’ % (frame_index, LATOME_FW_MODEL_EXEC_NB_BCID−1))
                    frame_index += 1
                    delay_reinject_counter += 1

                if frame_index == LATOME_FW_MODEL_EXEC_NB_BCID:
                    break

            else:
                # Get ADC−data from files or other input generators
                ltdb_frame = self.LTDB_Model.get_frame()
                # Get ADC−data from files
                lli_frame = self.LLI_Model.feed(ltdb_frame)
                # Feed the frame into istage
                istage_frame = self.ISTAGE_Model.feed(lli_frame)
                # Feed the frame into remap
                remap_frame = self.REMAP_Model.feed(istage_frame)
                # Get the first ADC−data for FIR at USER_code
                user_frame = self.USER_Model.feed(remap_frame)
                # Feed the frames into ousm
                osum_frame = self.OSUM_Model.feed(user_frame)
                self.cout(’Finished␣processing␣of␣BCID␣%d/%d’ % (frame_index, LATOME_FW_MODEL_EXEC_NB_BCID−1))
                frame_index += 1
                if frame_index == LATOME_FW_MODEL_EXEC_NB_BCID:
                    break

    def __del__(self):
        self.cout(’Deinitialization’)
        self.logpipe.put(’exit’)
        if LATOME_FW_MODEL_ARCHIVE_PATH != ’’:
            folder_name = LATOME_FW_MODEL_ARCHIVE_PATH + ’/’ + str(datetime.strftime(datetime.now(),’%Y−%m−%d_%H−%
                ↪ M−%S’)) + ’_%s_%s_%s’ % (self.LATOME_FW_CONFIGURATION, LATOME_FW_MODEL_EXEC_NB_BCID,
                ↪ LATOME_FW_MODEL_DEBUG_MODE)
            if not os.path.isdir(folder_name):
                os.mkdir(folder_name)
            os.system(’cp␣−rf␣.␣%s’ % (folder_name))


def main():
    model = LATOME_FW_Model()
    del model

if __name__ == "__main__":
    main()
```

B.1.2 AREUS mapper

```
#! /usr/bin/env python

"""
This tool povides simulated LAr physics data (AREUS) for given supercell id.
The python binding of the ROOT framework is required.
"""

__author__: Yves Bianga
__email__: yves.bianga@cern.ch
__version__: 0.1

from pprint import pprint
import sys
import os
import string
try:
    import ROOT
except Exception as e:
    potential_source = ''
    for path in os.environ['PATH'].split(':'):
        if os.path.isdir(path):
            if 'thisroot.sh' in os.listdir(path):
                if potential_source == '':
                    potential_source = ' (found:'
                potential_source += ' ' + path + '/thisroot.sh'
    if potential_source != '':
        potential_source += ')'
    raise Exception("Could not import ROOT library. Please make sure you have set your environment correctly. Usually 
        ↪ you need to source the file thisroot.sh from where you have installed ROOT on your computer%s." % (
        ↪ potential_source))

class System_Output(object):

    def fail(self, msg):
        sys.stderr.write('\n%s\n' % str(msg))
        sys.exit(−1)

    def warn(self, msg):
        sys.stderr.write('\n%s\n' % str(msg))

    def pprint(self, obj):
        pprint(obj)

class Root_Crawler(System_Output):
    """
    is looking for given SC−position in the root file,
    return a generator object if SC−position found
    """

    def __init__(self, config):
        self.config = config
        self.opened_file = []
        self.sc_tree = {}
        self.__get_folder(self.config['path'])

    def __get_folder(self, filename):
        rootfile = ROOT.TFile(filename, 'read')
        if rootfile.IsZombie():
            raise Exception("Cannot open provided ROOT file: %s" % (filename))
        if rootfile:
            self.opened_file.append(rootfile)
        ROOT.gROOT.SetBatch(True)
        for key in rootfile.GetListOfKeys():
            if key.IsFolder():
```

```
            kname = key.GetName()
            self.__check_position(kname)

def __check_position(self, kname):
    try:
        eta = float(kname.split('_')[−1].split('X')[0])
        phi = float(kname.split('_')[−1].split('X')[1])
        region = kname.split('_')[0]
        layer = kname.split('_')[1]
        if region in self.sc_tree.keys():
            if layer in self.sc_tree[region].keys():
                if phi in self.sc_tree[region][layer].keys():
                    if eta in self.sc_tree[region][layer][phi].keys():
                        print 'failed'
                    else:
                        self.sc_tree[region][layer][phi][eta] = kname
                else:
                    self.sc_tree[region][layer][phi] = {}
                    self.__check_position(kname)
            else:
                self.sc_tree[region][layer] = {}
                self.__check_position(kname)
        else:
            self.sc_tree[region] = {}
            self.__check_position(kname)
        return True
    except:
        return False

def find_areus_string(self, req_pos):
    """
    B_8D_B1
    {'eta_low': 0.7,
     'eta_up': 0.8,
     'layer': 'EMBack',
     'phi_low': 0.294,
     'phi_up': 0.393,
     'region': 'EMB',
     'sc_nb': 1}
    'EMB_EMBack_0.75X0.344'
    """
    if req_pos['region'] in self.sc_tree.keys():
        if req_pos['layer'] in self.sc_tree[req_pos['region']].keys():
            phi_found = False
            for phi in self.sc_tree[req_pos['region']][req_pos['layer']].keys():
                if req_pos['phi_low'] < phi <= req_pos['phi_up']:
                    phi_found = True
                    break
            if phi_found == False:
                self.fail('found␣no␣phi␣in␣range␣%s−%s' % (req_pos['phi_low'], req_pos['phi_up']))

            sc_cells = {}
            eta_found = False
            for eta in self.sc_tree[req_pos['region']][req_pos['layer']][phi].keys():
                if req_pos['eta_low'] < eta <= req_pos['eta_up']:
                    eta_found = True
                    sc_cells[eta] = self.sc_tree[req_pos['region']][req_pos['layer']][phi][eta]
            if eta_found == False:
                self.fail('found␣no␣eta␣in␣range␣%s−%s' % (req_pos['eta_low'], req_pos['eta_up']))

            if len(sorted(sc_cells.keys())) <= 4:
                sc_name = sc_cells[sorted(sc_cells.keys())[req_pos['sc_nb']−1]]
                obj = self.__get_keys(sc_name)
                return obj

        else:
                self.fail('layer␣%s␣not␣found' % (req_pos['layer']))
        else:
            self.fail('region␣%s␣not␣found' % (req_pos['region']))
        return None

    def __get_keys(self, kname):
        key_obj = self.opened_file[−1].Get('%s%s' % (kname, self.config['value']))
        RG = Root_Generator(key_obj, self.config['bcid_start'], key_obj.GetNbinsX())
        return RG

    def close_files(self):
        for rootfile in self.opened_file:
            rootfile.Close()

class Root_Generator(System_Output):
    """
    load and handle a single generator for each SC
    """

    def __init__(self, obj, bin_start=None, bin_stop=None):
        self.__obj = obj
        self.__obj_bins = self.__obj.GetNbinsX()
        if bin_stop and self.__obj_bins < bin_stop:
            self.fail('not␣enough␣bins␣available')
        self.__gen = self.__generator()
        if bin_start and self.__obj_bins > bin_start:
            for i in range(bin_start):
                self.__gen.next()
                #print i, self.__gen.next()

    def __call__(self):
        return self.__gen.next()

    def __generator(self):
        for i_bin in range(1, self.__obj_bins+1):
            yield self.__obj[i_bin]

    def get_bins(self):
        return self.__obj_bins


class Pattern_Translator(System_Output):
    """
    translate SC−pattern into eta/phi/SC ranges due to the SC naming convention
    """

    def __init__(self):
                                        #factor, offset, list
        self.a = {'EMB':{ 'phi': [0.098125, 0.0, list(string.ascii_uppercase)[:16]],
                          'eta': [0.1, 0.0, range(1,16)]},
                  'EMEC':{'phi': [0.19625, 0.0, list(string.ascii_uppercase)[:8]],
                          'eta1':[0.1, 1.4, range(1,12)],
                          'eta2':[0.2, 2.5, range(12,16)]},
                  'HEC1':{'phi': [0.098125, 0.0, list(string.ascii_uppercase)[:16]],
                          'eta': [0.1, 1.5, range(2,12)]},
                  'HEC2':{'phi': [0.19625, 0.0, list(string.ascii_uppercase)[:8]],
                          'eta': [0.2, 2.5, range(12,16)]},
                  'FCAL':{'phi': [0.3925, 0.0, list(string.ascii_uppercase)[:16]],
                          'eta1':[0.4, 3.2, range(1,4)],
                          'eta2':[0.5, 4.4, range(4,5)]}
                  }

    def translate(self, pos):
        # triggertower: abs(eta) position
        self.tt_area = pos.split("_")[0]
        if len(pos.split("_")[1]) == 2:
            # triggertower: abs(eta) position (=number)
            self.tt_I_eta = int(pos.split("_")[1][0])
            # triggertower: phi position (=letter)
            self.tt_L_phi = pos.split("_")[1][1]
        elif len(pos.split("_")[1]) == 3:
            # triggertower: abs(eta) position (=number)
            self.tt_I_eta = int(pos.split("_")[1][:2])
            # triggertower: phi position (=letter)
            self.tt_L_phi = pos.split("_")[1][2]
        # supercell calorimeter layer (P,F,M,B)
        self.sc_L = pos.split("_")[2][0]
        # supercell number
        self.sc_x = int(pos.split("_")[2][1])

        position = {'region':'',
                    'layer':'',
                    'sc_nb':0,
                    'phi_low':0,
                    'phi_up':0,
                    'eta_low':0,
                    'eta_up':0
                    }

        if self.tt_area == 'B':
            position['region'] = 'EMB'

        if self.sc_L == 'P':
            position['layer'] = 'EMPresampler'
        elif self.sc_L == 'F':
            position['layer'] = 'EMFront'
        elif self.sc_L == 'M':
            position['layer'] = 'EMMiddle'
        elif self.sc_L == 'B':
            position['layer'] = 'EMBack'

        position['eta_low'], position['eta_up'] = self.get_value(self.tt_I_eta, self.a[position['region']]['eta'])
        position['phi_low'], position['phi_up'] = self.get_value(self.tt_L_phi, self.a[position['region']]['phi'])
        position['sc_nb'] = self.sc_x

        return position

    def get_value(self, ind_value, param):
        ind = param[2].index(ind_value) #=1
        lower_border = param[0]*ind + param[1]
        upper_border = param[0]*(ind+1) + param[1]
        return round(lower_border, 3), round(upper_border, 3)

    def show_mappings(self):
        self.output(a['EMB']['phi'])
        self.output(a['EMB']['eta'])
        self.output(a['EMEC']['phi'])
        self.output(a['EMEC']['eta1'])
        self.output(a['EMEC']['eta2'])
        self.output(a['HEC1']['phi'])
        self.output(a['HEC1']['eta'])
        self.output(a['HEC2']['phi'])
        self.output(a['HEC2']['eta'])
        self.output(a['FCAL']['phi'])
        self.output(a['FCAL']['eta1'])
        self.output(a['FCAL']['eta2'])

    def output(self, b):
        for i in b[2]:
            ind = b[2].index(i)
            lower_border = b[0]*ind + b[1]
            upper_border = b[0]*(ind+1) + b[1]
            print i, round(lower_border, 3), round(upper_border, 3)


class AREUS_mapper(System_Output):
    """
    main class, create and handle a pool of generators for requested SCs
    """

    def __init__(self,parameters):
        self.parameters = parameters
        self.supercells = {}
        self.supercells_bcid = {}
        self.PT = Pattern_Translator()
        self.RC = Root_Crawler(parameters)

    def request(self, pos_id):
        if pos_id in self.supercells.keys():
            self.supercells_bcid[pos_id] += 1
            output = self.parameters['pedestal'] + self.supercells[pos_id]()
        else:
            self.supercells_bcid[pos_id] = 0
            try:
                position = self.PT.translate(pos_id)
                gen_obj = self.RC.find_areus_string(position)
                self.supercells[pos_id] = gen_obj
                output = self.parameters['pedestal'] + self.supercells[pos_id]()
            except:
                output = self.parameters['pedestal']

        if self.parameters['bcid_fake_output']:
            return self.supercells_bcid[pos_id], output
        else:
            return output


def main():

    parameters = {'path':'of_analysis.root',
                  'value':'/0_digitization/digits_out_sequence_cnt',
                  'bcid_start': 1001,
                  'pedestal': 1024
                  }

    AM = AREUS_mapper(parameters)
    while True:
        i = raw_input('')
        print AM.request(i)


if __name__ == "__main__":
    sys.exit(main())
```

B.1.3 LTDB model

```
import os
from pprint import pprint

from helper_tools import Helper
from IntBitVector import *
from loc_ic import *
from mif_file import *
```

```
class LTDB_Model(Helper):

    def __init__(self):
        """
        Parse and deattach globals to keep this module clean, run generator
        """

        # Init Helper as child (=get globals from parent, connect to logging, printing and debugging services)
        Helper.__init__(self)

        # Input/output path/files
        self.input_adc_filename = 'adc_data.*.csv'
        self.ouput_adc_filename = 'adc_data.*.csv'
        self.output_patgen_filename = 'pattern_init.*.mif'

        # BCIDs
        self.parse_globals('LHC_CYCLE_NB_BCID')
        self.parse_globals('LATOME_FW_MODEL_EXEC_NB_BCID', default=self.LHC_CYCLE_NB_BCID)

        # NUMBER OF LTDB INPUT STREAMS
        self.parse_globals('LTDB_INPUT_DATA_STREAMS_NB')

        # NUMBER OF ACTIVE FIBERS / ACTIVE LTDB STREAMS
        self.parse_globals('GENERATE_LTDB_INPUT_ACTIVE_FIBERS', default=hex(int(pow(2, self.
            ↪ LTDB_INPUT_DATA_STREAMS_NB)−1)))

        # MAPPING FILE
        self.parse_globals('GENERATE_LTDB_INPUT_OUTPUT_MAPPING_FILE', default=False)
        self.parse_globals('MAPPING_CONFIGURATION_FILE_PATH')

        # LOC_IC
        self.parse_globals('LOC_IC_FRAME_ADC_SAMPLE_BIT_WIDTH')
        self.parse_globals('LOC_IC_FRAME_NB_ADC_SAMPLES')
        self.parse_globals('LOC_IC_FRAME_WIDTH')

        # PATTERN GENERATOR
        self.parse_globals('PATTERN_GENERATOR_MEMORY_DATA_WIDTH')

        # INPUT DATA TYPE
        self.parse_globals('GENERATE_LTDB_INPUT_DATA_TYPE', default='DATA_CONSTANT_ID')


        # RUNNING GENERATOR
        self.matching_types = ['DATA_CONSTANT_ID', 'DATA_CONSTANT_BCID', 'DATA_RANDOM', 'DATA_AREUS_INPUT', '
            ↪ DATA_FILE_INPUT', 'DATA_RAMPUP']
        if self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[0]:
            self.__gen_method = self.__generate_adc_with_constant_id
            self.__gen_name = 'gen.␣constant−ID'
        elif self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[1]:
            self.parse_globals('BCID_BUS_WIDTH')
            self.__gen_method = self.__generate_adc_with_constant_bcid
            self.__gen_name = 'gen.␣constant−BCID'
        elif self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[2]:
            random.seed()
            self.__gen_method = self.__generate_adc_random
            self.__gen_name = 'gen.␣random'
        elif self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[3]:
            self.parse_globals('LTDB_INPUT_AREUS_FILE_PATH')
            self.parse_globals('LTDB_INPUT_AREUS_FILE_PEDESTAL', default=1024)
            self.parse_globals('LTDB_INPUT_AREUS_BCID_START', default=1001)
            self.__gen_method = self.__generate_adc_with_areus
            self.__gen_name = 'AREUS␣ADC␣read'
        elif self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[4]:
            self.parse_globals('GENERATE_LTDB_INPUT_DATA_FILE_PATH', default='adc')
            self.cout_init('ADC␣input␣files␣location', os.getcwd() + '/' + self.GENERATE_LTDB_INPUT_DATA_FILE_PATH)
            self.__gen_method = self.__generate_adc_from_file
            self.__gen_name = 'read␣file␣input'
        elif self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[5]:
            self.__gen_method = self.__generate_adc_rampup
            self.__gen_name = 'gen.␣ranpup'
        else:
            raise Exception('The␣type␣of␣generated␣data␣must␣be␣one␣of:␣%s' % (str(matching_types)))


        # STARTING INIT MODULE
        self.__init_module()
        self.cout_init_done()

    def __init_module(self):
        """
        Init all related structures.
        """
        # ACTIVE FIBERS MASK
        self.active_fibers = {}
        active_mask = '{:048b}'.format(int(self.GENERATE_LTDB_INPUT_ACTIVE_FIBERS, 16))
        for fiber, active in enumerate(active_mask[::−1]):
            self.active_fibers[fiber] = bool(int(active))
        if sum(self.active_fibers.values()) > self.LTDB_INPUT_DATA_STREAMS_NB:
            raise Exception("There␣are␣only␣%d␣enabled␣fibers.␣You␣cannot␣have␣an␣active␣fiber␣mask␣of␣'0x%x'␣which␣
                ↪ means␣%d␣fibers" % (self.LTDB_INPUT_DATA_STREAMS_NB, self.GENERATE_LTDB_INPUT_ACTIVE_FIBERS,
                ↪ sum(self.active_fibers.values())))

        # GET SC_ID CONFIGURATION
        conf_file = open(self.MAPPING_CONFIGURATION_FILE_PATH, 'r')
        self.fibers_scid = {}
        for line in conf_file:
            if line[:2] == 'IN':
                line_list = line.replace('\n', '').split()
                stream_nb = int(line_list[1][1:]) − 1
                self.fibers_scid[stream_nb] = [None if x == 'GND' else x for x in line_list[4:]]
        conf_file.close()
        for stream_index in range(48):
            if stream_index not in self.fibers_scid.keys():
                self.fibers_scid[stream_index] = 8∗[None]

        # LOAD OUTPUT STREAMS MAPPING
        self.output_streams_mapping = {}
        if self.GENERATE_LTDB_INPUT_OUTPUT_MAPPING_FILE:
            try:
                mapping_io_file = open(self.GENERATE_LTDB_INPUT_OUTPUT_MAPPING_FILE, "r")
            except:
                raise Exception("Cannot␣read␣the␣output␣streams␣mapping␣from␣file:␣'%s'" % (self.
                    ↪ GENERATE_LTDB_INPUT_OUTPUT_MAPPING_FILE))
            for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
                line = mapping_io_file.readline().replace('\n', '').replace('\r', '')
                if line == '':
                    raise Exception("Expecting␣%d␣lines␣in␣the␣%s␣file␣for␣output␣streams␣mapping,␣only␣got␣%d" % (self.
                        ↪ LTDB_INPUT_DATA_STREAMS_NB, self.GENERATE_LTDB_INPUT_OUTPUT_MAPPING_FILE,
                        ↪ stream_index))
                self.output_streams_mapping[stream_index] = int(line)
        else:
            for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
                self.output_streams_mapping[stream_index] = int(stream_index)


        # LOC_IC INSTANCE
        self.loc_ics = []
        for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
            if stream_index == 0:
                self.loc_ics.append(LOC_IC())
            else:
                self.loc_ics.append(LOC_IC(self.loc_ics[0].get_prbs()))
            self.loc_ics[stream_index].resetBCID()
            self.loc_ics[stream_index].resetScrambler()
            self.loc_ics[stream_index].resetIncreasingCounter()

        # Init AREUS−mapper
        if self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[3]:
            from AREUS_mapper import *
            areus_parameters = {'path':self.LTDB_INPUT_AREUS_FILE_PATH,
                                'value': '/0_digitization/digits_out_sequence_cnt',
                                'bcid_start': self.LTDB_INPUT_AREUS_BCID_START,
                                'bcid_fake_output': True,
                                'pedestal': self.LTDB_INPUT_AREUS_FILE_PEDESTAL,
                               }
            self.AM = AREUS_mapper(areus_parameters)

        # Open input files
        if self.GENERATE_LTDB_INPUT_DATA_TYPE == self.matching_types[4]:
            self.adc_data_input_files = []
            adc_fn_split = self.input_adc_filename.split('*')
            for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
                if self.active_fibers[stream_index]:
                    adc_data_file_filename = '%s/%s%d%s' % (self.GENERATE_LTDB_INPUT_DATA_FILE_PATH, adc_fn_split[0],
                         ↪ stream_index, adc_fn_split[1])
                    self.adc_data_input_files.append(open(adc_data_file_filename, 'r'))
                    self.adc_data_input_files[stream_index].readline()
                else:
                    self.adc_data_input_files.append(None)

        # OPEN ADC DATA FILES
        adc_fn_split = self.ouput_adc_filename.split('*')
        self.adc_data_files = []
        for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
            adc_data_file_filename = '%s/%s%d%s' % (self.LATOME_FW_MODEL_OUTPUT_PATH, adc_fn_split[0], stream_index,
                 ↪ adc_fn_split[1])
            self.adc_data_files.append(open(adc_data_file_filename, 'w'))
            header = 'bcid,'
            for adc_index in range(self.LOC_IC_FRAME_NB_ADC_SAMPLES):
                header += str(self.fibers_scid[stream_index][adc_index]) + ','
            self.adc_data_files[−1].write(header[:−1] + '\n')

        # OPEN PATTERN GENERATOR FILES
        patgen_fn_split = self.output_patgen_filename.split('*')
        self.patgen_mif_files = []
        memory_depth = self.LATOME_FW_MODEL_EXEC_NB_BCID*self.LOC_IC_FRAME_NB_ADC_SAMPLES*(self.
             ↪ LOC_IC_FRAME_WIDTH/self.LOC_IC_FRAME_NB_ADC_SAMPLES)/self.
             ↪ PATTERN_GENERATOR_MEMORY_DATA_WIDTH
        memory_depth = int(pow(2, ceil(log(memory_depth, 2))))
        for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
            patgen_mif_filename = '%s/%s%d%s' % (self.LATOME_FW_MODEL_OUTPUT_PATH, patgen_fn_split[0], stream_index,
                 ↪ patgen_fn_split[1])
            self.patgen_mif_files.append(MIFFile(patgen_mif_filename, MIFFile.MODE_WRITE, depth = memory_depth, width = self.
                 ↪ PATTERN_GENERATOR_MEMORY_DATA_WIDTH))

        self.patgen_frames = {}
        self.frame_index = 0



    def run_standalone(self):
        """
        Mainloop for standalone execution
        """
        for bcid in range(self.LATOME_FW_MODEL_EXEC_NB_BCID):
            _ = self.get_frame()
```

```
def get_frame(self):
    """
    Get ADC−data for parallel execution of FW−Model
    """
    ltdb_frame = {'adc_data':{},
                  'sc_id':{}}
    self.patgen_frames[self.frame_index] = {}

    for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
        ltdb_frame['adc_data'][self.output_streams_mapping[stream_index]] = {}
        ltdb_frame['sc_id'][self.output_streams_mapping[stream_index]] = {}
        # ADC VALUES
        adc_in = []
        adc_str = '0x%03x,' % (self.frame_index)
        for supercell_index in range(self.LOC_IC_FRAME_NB_ADC_SAMPLES):
            sc_id = self.fibers_scid[stream_index][supercell_index]
            if sc_id in [None, 'GND']:
                sc_id = None
            ltdb_frame['sc_id'][self.output_streams_mapping[stream_index]][supercell_index] = sc_id
            if self.active_fibers[stream_index]:
                adc_sample = self.__gen_method(self.frame_index, stream_index, supercell_index)
                ltdb_frame['adc_data'][self.output_streams_mapping[stream_index]][supercell_index] = int(adc_sample.intValue())
                # Debugging
                if self.dbg_ltdb_stream > self.LTDB_INPUT_DATA_STREAMS_NB −1:
                    if stream_index == self.LTDB_INPUT_DATA_STREAMS_NB − 1:
                        self.cout_debug('incoming/reordered', ['not␣possible␣to␣fetch␣dbg_ltdb_stream␣=␣', self.
                            ↪ dbg_ltdb_stream, 'LTDB_INPUT_DATA_STREAMS_NB␣=␣', self.
                            ↪ LTDB_INPUT_DATA_STREAMS_NB])
                elif stream_index == self.dbg_ltdb_stream or sc_id == self.dbg_ltdb_sc_name:# and supercell_index == self.
                    ↪ dbg_ltdb_sc_id:
                    self.cout_debug(self.__gen_name, [stream_index,
                                                      supercell_index,
                                                      ltdb_frame['adc_data'][self.output_streams_mapping[stream_index]][
                                                          ↪ supercell_index],
                                                      sc_id,
                                                      ])

            else:
                adc_sample = IntBitVector(size = self.LOC_IC_FRAME_ADC_SAMPLE_BIT_WIDTH)
                ltdb_frame['adc_data'][self.output_streams_mapping[stream_index]][supercell_index] = int(0)
            adc_in.append(adc_sample)
            adc_str += '0x%s,' % (adc_sample.get_bitvector_in_hex())
        self.adc_data_files[stream_index].write(adc_str[:−1] + '\n')
        #self.adc_data_files[self.output_streams_mapping[stream_index]].write(adc_str[:−1] + '\n')

        # PATTERN GENERATOR FRAME
        frame = self.loc_ics[stream_index].createFrame(adc_in, scramble_frame = False, is_active = self.active_fibers[stream_index
            ↪ ])
        self.patgen_frames[self.frame_index][stream_index] = frame

    self.frame_index += 1
    if self.frame_index == self.LATOME_FW_MODEL_EXEC_NB_BCID:
        self.__dump_patgen()
        self.__close_module()
    return ltdb_frame

def __dump_patgen(self):
    """
    Dump pattern generator files at parallel execution of FW−Model
    """
    self.cout('Dump␣pattern␣generator␣values')
    # dump pattern generator values
    for bcid in range(self.LATOME_FW_MODEL_EXEC_NB_BCID):
        for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
            frame = self.patgen_frames[bcid][stream_index]
            if bcid == 0:
                previous_frame = self.patgen_frames[self.LATOME_FW_MODEL_EXEC_NB_BCID−1][stream_index]
            else:
                previous_frame = self.patgen_frames[bcid−1][stream_index]
            # Add previous' frame CRC to this one and cut it in 32 bits chunks
            previous_frame_crc = previous_frame['frame_descrambled'][self.LOC_IC_FRAME_WIDTH−8:self.
                ↪ LOC_IC_FRAME_WIDTH]
            mixed_frame = frame['frame_descrambled'][0:self.LOC_IC_FRAME_WIDTH−8] + previous_frame_crc
            # Cut in 32 bits chunks
            while (len(mixed_frame) >= self.PATTERN_GENERATOR_MEMORY_DATA_WIDTH):
                loc_ic_chunk = mixed_frame[0:self.PATTERN_GENERATOR_MEMORY_DATA_WIDTH]
                mixed_frame = mixed_frame[self.PATTERN_GENERATOR_MEMORY_DATA_WIDTH:len(mixed_frame)]
                self.patgen_mif_files[self.output_streams_mapping[stream_index]].dump(loc_ic_chunk)


def __close_module(self):
    # Close output files
    self.cout('Closing␣files')
    for stream_index in range(self.LTDB_INPUT_DATA_STREAMS_NB):
        if self.active_fibers[stream_index]:
            if self.GENERATE_LTDB_INPUT_DATA_TYPE == 'DATA_FILE_INPUT':
                self.adc_data_input_files[stream_index].close()
            self.adc_data_files[stream_index].close()
            self.patgen_mif_files[stream_index].close()

def __generate_adc_with_constant_id(self, bcid, stream_index, supercell_index):
    # Generate constant values:
    # [13:12] 00
    # [11:6] Fiber index
    # [5:3] Supercell index
    # [2:0] BCID index
    empty_2 = IntBitVector(size = 2)
    fiber_index = IntBitVector(size = 6, intVal = stream_index)
    supercell_index = IntBitVector(size = 3, intVal = supercell_index)
    bcid_3 = IntBitVector(size = 3, intVal = bcid % 8)
    adc_sample = empty_2 + fiber_index + supercell_index + bcid_3
    #value = bcid + (supercell_index << 3) + (stream_index << 6)
    return adc_sample

def __generate_adc_with_constant_bcid(self, bcid, stream_index, supercell_index):
    # Generate constant values:
    # [13:12] 00
    # [11:0] BCID
    empty_2 = IntBitVector(size = 2)
    bcid_12 = IntBitVector(size = self.BCID_BUS_WIDTH, intVal = bcid)
    adc_sample = empty_2 + bcid_12
    return adc_sample

def __generate_adc_random(self, bcid, stream_index, supercell_index):
    adc_sample = IntBitVector(size = self.LOC_IC_FRAME_ADC_SAMPLE_BIT_WIDTH, intVal = random.randrange(pow(2, self.
        ↪ LOC_IC_FRAME_ADC_SAMPLE_BIT_WIDTH)))
    return adc_sample

def __generate_adc_rampup(self, bcid, stream_index, supercell_index):
    adc_sample = IntBitVector(size = self.LOC_IC_FRAME_ADC_SAMPLE_BIT_WIDTH, intVal = (int(bcid)%4)+1)
    return adc_sample

def __generate_adc_with_areus(self, bcid, stream_index, supercell_index):
    # Get AREUS physics data:
    # [13:12] 00
    # [11:0] adc−data
    empty_2 = IntBitVector(size = 2)
    sc_id = self.fibers_scid[stream_index][supercell_index]
    if sc_id not in [None, 'GND']:
        areus_bcid, areus_data = self.AM.request(sc_id)
        """
        areus_data_tmp = 80*(areus_data − 1024)
```

```
            if areus_data_tmp < 0:
                areus_data = areus_data
            elif areus_data_tmp > 3072:
                areus_data = 4095
            else:
                areus_data = areus_data_tmp + 1024
            """
            #areus_data_bit = IntBitVector(size = int(12), intVal = int(100*areus_data) % 4096)
            areus_data_bit = IntBitVector(size = int(12), intVal = int(areus_data))
            if areus_bcid != bcid:
                raise Exception('BCID␣collision␣at␣supercell:␣%s.␣Requested␣BCID:␣%s␣AREUS␣BCID:␣%s.' % (sc_id, bcid,
                    ↪ areus_bcid))
        else:
            areus_data_bit = IntBitVector(size = int(12), intVal = 0)
        return empty_2 + areus_data_bit

    def __generate_adc_from_file(self, bcid, stream_index, supercell_index):
        # Get ADC−data from file
        if supercell_index == 0:
            self.adc_file_line = []
            if self.active_fibers[stream_index]:
                line = self.adc_data_input_files[stream_index].readline().replace('\n','').replace('\r','')
                if len(line) > 0:
                    adc_samples = line.split(',')
                    del adc_samples[0]
                else:
                    raise Exception("Error␣reading␣from␣ADC␣data␣file␣for␣fiber␣%d,␣bcid␣%s" % (stream_index, bcid+1))
            else:
                adc_samples = self.LOC_IC_FRAME_NB_ADC_SAMPLES*['0x0']

            for adc_index in range(self.LOC_IC_FRAME_NB_ADC_SAMPLES):
                self.adc_file_line.append(BitVector(size = self.LOC_IC_FRAME_ADC_SAMPLE_BIT_WIDTH, intVal = int(adc_samples[
                    ↪ adc_index], 16)))

        return self.adc_file_line[supercell_index]
```

B.1.4 User Code model

```
from copy import deepcopy
from math import log, ceil
from ctypes import *

from helper_tools import Helper
from BitVectorL import *
from mif_file import *


class USER_Model(Helper):
    """
    User_model, with binding to the user_code C−model and a AREUS bypass functionality
    """

    def __init__(self):
        """
        Parse and deattach globals to keep this module clean
        """

        # Init Helper as child (=get globals from parent, connect to logging, printing and debugging services)
        Helper.__init__(self)

        # Output files
        self.output_user_filename = 'user_frames.*.txt'
self.output_checker_filename = 'user_data_checker.*.mif'
self.output_coeff_fir_filename = 'coeff_fir_init.*.mif'
self.output_coeff_sat_filename = 'coeff_sat_init.*.mif'

# User_code related parameters
self.parse_globals('LHC_CYCLE_NB_BCID')
self.parse_globals('LATOME_FW_MODEL_EXEC_NB_BCID', default=self.LHC_CYCLE_NB_BCID)

self.parse_globals('USER_CODE_OSUM_DATA_STREAMS_NB')
self.parse_globals('USER_CODE_STREAM_NB_SAMPLES')

self.parse_globals('USER_TAU_SELECTION_ENABLE')
self.parse_globals('USER_COMBINE_BLOCK_ENABLE')
self.parse_globals('USER_HARDPOINT_IN_FIR_FILTER_ENABLE')

self.parse_globals('USER_FILTERING_IN_FIR_FILTER_TAPS_NB')
self.parse_globals('USER_FILTERING_IN_SATURATION_DETECTION_TAPS_NB')

self.parse_globals('USER_ENERGY_WIDTH')
self.parse_globals('USER_ENERGY_TAU_WIDTH')

self.parse_globals('USER_HARDPOINT_IN_FIR_FILTER')
self.parse_globals('USER_HARDPOINT_IN_SATURATION_DETECTION')

self.parse_globals('USER_QUALITY_WIDTH')

self.parse_globals('USER_DATA_CHECKER_VECTOR_BUS_WIDTH')

# Pedestal hardpoint is set in USER-CODE DSP-qsys-configuration and therefore not a USER constant
self.parse_globals('USER_PEDESTAL_HARDPOINT', default=3)
self.parse_globals('USER_FIR_COEFF_BITSHIFT', default=int(self.USER_HARDPOINT_IN_FIR_FILTER - self.
    ↪ USER_PEDESTAL_HARDPOINT))
self.parse_globals('USER_SAT_COEFF_BITSHIFT', default=int(self.USER_HARDPOINT_IN_SATURATION_DETECTION - self.
    ↪ USER_PEDESTAL_HARDPOINT))
#self.parse_globals('USER_FIR_COEFF_BITSHIFT', default=5)
#self.parse_globals('USER_SAT_COEFF_BITSHIFT', default=5)

# If loop-mode is set the OSUM feeding is delayed by 3. This is neseccary to use a FIR depth of currently 4 at USER_model
self.parse_globals('LATOME_FW_MODEL_LOOP_MODE', default=False)
if self.LATOME_FW_MODEL_LOOP_MODE:
    self.parse_globals('LATOME_FW_MODEL_LOOP_DEPTH', default=int(self.USER_FILTERING_IN_FIR_FILTER_TAPS_NB -
        ↪ 1))
    # Generate buffer for System-Test-M3
    self.buffer_remap_frame_m3 = {}

##################################
# Input types for ADC-data
##################################
# Parsing ADC-data input type
self.parse_globals('USER_CODE_OUTPUT_DATA_TYPE', default='C_MODEL')
matching_types = ['C_MODEL', 'AREUS', 'C_MODEL_VS_AREUS']
for out_type in matching_types[:2]:
    setattr(self, 'output_type_' + out_type.lower(), False)

if self.USER_CODE_OUTPUT_DATA_TYPE in [matching_types[0], matching_types[2]]:
    # Process REMAP data with C-model
    self.output_type_c_model = True
    # Binding path to the user_code C-model
    self.parse_globals('LATOME_FW_MODEL_PATH')
    self.user_c_model_path = self.LATOME_FW_MODEL_PATH + 'user_c_model/User_Code.so'
    # Init global pedestal
    self.parse_globals('GENERATE_LTDB_INPUT_DATA_TYPE')
    if self.GENERATE_LTDB_INPUT_DATA_TYPE == 'DATA_AREUS_INPUT':
        self.parse_globals('USER_CODE_GLOBAL_PEDESTAL', default=1024)
    else:
        self.parse_globals('USER_CODE_GLOBAL_PEDESTAL', default=0)
        if self.USER_CODE_OUTPUT_DATA_TYPE in [matching_types[1], matching_types[2]]:
            # Replace REMAP ADC−data with AREUS output for energy
            self.output_type_areus = True
            self.parse_globals('USER_INPUT_AREUS_ET_FILE_PATH')
            self.parse_globals('USER_INPUT_AREUS_TAU_FILE_PATH')
            self.parse_globals('LTDB_INPUT_AREUS_BCID_START', default=1001)
        if self.USER_CODE_OUTPUT_DATA_TYPE == matching_types[2]:
            # Debugging stuff for C−model vs AREUS output
            self.dbg_fir_vs_areus_filename = '0_fw_model_debug_fir_c_vs_areus.txt'
            self.dbg_c_e = {}
            self.dbg_a_e = {}


        ################################
        # Input types for coefficients, only used by C−model
        ################################
        if self.output_type_c_model:
            # Please note: The C−model expected coefficients as 14 bit signed integer (= −8192 < coeff < 8192)
            self.parse_globals('USER_COEFFICIENT_WIDTH')
            # Parsing coefficients input type
            self.parse_globals('USER_CODE_COEFF_TYPE', default='TRANSPARENT')
            if self.USER_CODE_COEFF_TYPE == 'TRANSPARENT':
                # Generate transparent coefficients (= 0/0/0/1 * (2**self.USER_*_COEFF_BITSHIFT))
                pass
            elif self.USER_CODE_COEFF_TYPE == 'BITSHIFT':
                # Generate coefficients to bitshift data (= 0/0/0/2**bitshift * (2**self.USER_*_COEFF_BITSHIFT))
                self.coeff_value_bitshift = 6
            elif self.USER_CODE_COEFF_TYPE == 'RAMPUP':
                # Generate rampup coefficients (= 1/2/3/4 * (2**self.USER_*_COEFF_BITSHIFT))
                pass
            elif self.USER_CODE_COEFF_TYPE == 'AREUS':
                # Load coefficients from AREUS (* (2**self.USER_*_COEFF_BITSHIFT))
                self.parse_globals('USER_INPUT_AREUS_ET_FILE_PATH')
                self.parse_globals('USER_INPUT_AREUS_TAU_FILE_PATH')
            elif self.USER_CODE_COEFF_TYPE == 'FILE':
                # Load coefficients from files
                self.parse_globals('USER_CODE_FIR_FILTER_COEFFICIENTS_INIT_FILE_PATH')
                self.parse_globals('USER_CODE_SATURATION_DETECTION_COEFFICIENTS_INIT_FILE_PATH')
                self.input_fir_filename = 'coefficient_fir_init_file.*.mif'
                self.input_sat_filename = 'coefficient_sat_init_file.*.mif'

        # Init user_code related structures
        self.cout_init_done()
        self.__init_module()

    def __init_module(self):
        """
        Init all related structures.
        """
        # Init frame index
        self.frame_index = −1

        ################################
        # Init C−model if choosen
        ################################
        if self.output_type_c_model:
            # Bind user_code C−Model
            self.user_c_model = CDLL(self.user_c_model_path)

            # Prepare coefficients
            self.coeff_fir = {}
            self.coeff_sat = {}
            for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
                self.coeff_fir[stream_index] = {}
                self.coeff_sat[stream_index] = {}
                for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
```

```
            self.coeff_fir[stream_index][supercell_index] = {}
            self.coeff_sat[stream_index][supercell_index] = {}
            for coeff in range(4):
                self.coeff_fir[stream_index][supercell_index][coeff] = {'float':0.0, 'c-model':0, 'integer':0, 'file-format':hex(0)}
                self.coeff_sat[stream_index][supercell_index][coeff] = {'float':0.0, 'c-model':0, 'integer':0, 'file-format':hex(0)}
# Open coefficient files
self.coeff_fir_mif_files = []
self.coeff_sat_mif_files = []
for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
    coeff_fir_fn = '%s/%s' % (self.LATOME_FW_MODEL_OUTPUT_PATH, self.output_coeff_fir_filename.replace('*', '%02d'
        ↪ % stream_index))
    coeff_sat_fn = '%s/%s' % (self.LATOME_FW_MODEL_OUTPUT_PATH, self.output_coeff_sat_filename.replace('*', '%02
        ↪ d' % stream_index))
    # With filter depth of 4: 8 (= 6 + 2) sample per stream with 4 coefficients.
    depth = (self.USER_CODE_STREAM_NB_SAMPLES + 2) * 4 * 8
    self.coeff_fir_mif_files.append(MIFFile(coeff_fir_fn, MIFFile.MODE_WRITE, depth = depth, width = self.
        ↪ USER_COEFFICIENT_WIDTH))
    self.coeff_sat_mif_files.append(MIFFile(coeff_sat_fn, MIFFile.MODE_WRITE, depth = depth, width = self.
        ↪ USER_COEFFICIENT_WIDTH))

# Generate transparent, bitshift or rampup coefficients
if self.USER_CODE_COEFF_TYPE in ['TRANSPARENT', 'BITSHIFT', 'RAMPUP']:
    self.cout('Generate␣transparent,␣bitshift␣or␣rampup␣coefficients')
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
            for coeff in range(4):
                if self.USER_CODE_COEFF_TYPE == 'TRANSPARENT' and coeff == 0:
                    self.__feed_coefficients(stream_index, supercell_index, coeff, 1.0, 1.0)
                elif self.USER_CODE_COEFF_TYPE == 'BITSHIFT' and coeff == 0:
                    self.__feed_coefficients(stream_index, supercell_index, coeff, 2**self.coeff_value_bitshift, 2**self.
                        ↪ coeff_value_bitshift)
                elif self.USER_CODE_COEFF_TYPE == 'RAMPUP':
                    values = range(1,5)#[::-1]
                    self.__feed_coefficients(stream_index, supercell_index, coeff, values[coeff], values[coeff])
                else:
                    self.__feed_coefficients(stream_index, supercell_index, coeff, 0.0, 0.0)

# Load coefficients from files
elif self.USER_CODE_COEFF_TYPE == 'FILE':
    self.cout('Reading␣FIR␣+␣SAT␣coefficients␣from␣files')
    i = 0
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        # TODO: Implement reading from MIFFile class. At the moment MIFFile class can't parse the mif lines, regex-
            ↪ expression of MIFFile.read() doesn't match.
        coeff_fir_file = open('%s/%s' % (self.USER_CODE_FIR_FILTER_COEFFICIENTS_INIT_FILE_PATH, self.
            ↪ input_fir_filename.replace('*', '%02d' % stream_index)), 'r')
        coeff_sat_file = open('%s/%s' % (self.USER_CODE_SATURATION_DETECTION_COEFFICIENTS_INIT_FILE_PATH,
            ↪ self.input_sat_filename.replace('*', '%02d' % stream_index)), 'r')
        for i in range(10):
            fir_line = coeff_fir_file.readline()
            sat_line = coeff_sat_file.readline()
            if 'BEGIN' in fir_line:
                break
        for coeff in range(4):
            for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
                fir_int = int(coeff_fir_file.readline().replace('\n', '').replace('\r', '').split(';')[0].split('␣')[-1], 16)
                sat_int = int(coeff_sat_file.readline().replace('\n', '').replace('\r', '').split(';')[0].split('␣')[-1], 16)
                fir_float = self.fp_int_to_float_two(self.USER_COEFFICIENT_WIDTH - self.USER_FIR_COEFF_BITSHIFT,
                    ↪ self.USER_FIR_COEFF_BITSHIFT, fir_int)
                sat_float = self.fp_int_to_float_two(self.USER_COEFFICIENT_WIDTH - self.USER_SAT_COEFF_BITSHIFT,
                    ↪ self.USER_SAT_COEFF_BITSHIFT, sat_int)
                self.__feed_coefficients(stream_index, supercell_index, coeff, fir_float, sat_float)
            # Coefficient files are written for 8 samples per stream (= 6 + 2)
            for supercell_index in range(2):
                _ = coeff_fir_file.readline().replace('\n', '').replace('\r', '')
                _ = coeff_sat_file.readline().replace('\n', '').replace('\r', '')
            i += 1
        # Prepare AREUS coefficients dumping while feeding the model
        elif self.USER_CODE_COEFF_TYPE == 'AREUS':
            # get coefficients at the first feed
            areus_et_parameters = {'path':self.USER_INPUT_AREUS_ET_FILE_PATH,
                                   'value': '/1_of/configuration',
                                   'bcid_start': 5,
                                   'bcid_fake_output': False,
                                   'pedestal': 0,
                                  }
            areus_tau_parameters = {'path':self.USER_INPUT_AREUS_TAU_FILE_PATH,
                                    'value': '/1_of_tau/configuration',
                                    'bcid_start': 9,
                                    'bcid_fake_output': False,
                                    'pedestal': 0,
                                   }
            from AREUS_mapper import AREUS_mapper
            self.AM_et_coeff = AREUS_mapper(areus_et_parameters)
            self.AM_tau_coeff = AREUS_mapper(areus_tau_parameters)

    ##################################
    # Init AREUS energy-data output if choosen
    ##################################
    if self.output_type_areus:
        areus_et_parameters = {'path':self.USER_INPUT_AREUS_ET_FILE_PATH,
                               'value': '/1_of/digits_out_sequence_eT',
                               'bcid_start': self.LTDB_INPUT_AREUS_BCID_START,
                               'bcid_fake_output': True,
                               'pedestal': 0,
                              }
        areus_tau_parameters = {'path':self.USER_INPUT_AREUS_TAU_FILE_PATH,
                                'value': '/1_of_tau/digits_out_sequence_eT',
                                'bcid_start': self.LTDB_INPUT_AREUS_BCID_START,
                                'bcid_fake_output': True,
                                'pedestal': 0,
                               }
        from AREUS_mapper import AREUS_mapper
        self.AM_et_data = AREUS_mapper(areus_et_parameters)
        self.AM_tau_data = AREUS_mapper(areus_tau_parameters)

    # Open user frames files
    self.user_frames_files = []
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        user_frames_files_filename = '%s/%s' % (self.LATOME_FW_MODEL_OUTPUT_PATH, self.output_user_filename.replace('*'
            ↪ , '%02d' % stream_index))
        self.user_frames_files.append(open(user_frames_files_filename, 'w'))
        self.user_frames_files[stream_index].write('stream_index,␣,frame_index,frame_index(hex),␣,bcid,bcid(hex),␣,
            ↪ startofpacket,␣,valid,␣,quality,␣,data\n')

    # Open user data checker files
    self.user_data_checker_files = []
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        user_data_checker_files_filename = '%s/%s' % (self.LATOME_FW_MODEL_OUTPUT_PATH, self.output_checker_filename.
            ↪ replace('*', '%d' % stream_index))
        depth = int(pow(2,ceil(log(self.LHC_CYCLE_NB_BCID*self.USER_CODE_STREAM_NB_SAMPLES, 2))))
        self.user_data_checker_files.append(MIFFile(user_data_checker_files_filename, MIFFile.MODE_WRITE, depth = depth,
            ↪ width = self.USER_DATA_CHECKER_VECTOR_BUS_WIDTH))

    # Generate an empty data frame
    user_frame_data = { 'energy_data': [],
                        'energy_data_float': [],
                        'quality': [],
                        'startofpacket': [],
                        'valid': [],
                      }
    for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
        if supercell_index == 0:
            user_frame_data['startofpacket'].append(BitVector(bitstring = '1'))
        else:
            user_frame_data['startofpacket'].append(BitVector(bitstring = '0'))
        user_frame_data['energy_data'].append(BitVector(size = self.USER_ENERGY_WIDTH, intVal = 0))
        user_frame_data['energy_data_float'].append(0.0)
        user_frame_data['quality'].append(BitVector(size = self.USER_QUALITY_WIDTH, intVal = 0))
        user_frame_data['valid'].append(BitVector(bitstring = '0'))

    # Initialize the empty user frame
    self.user_frame_empty = {'data': [], 'bcid': None}
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        self.user_frame_empty['data'].append(deepcopy(user_frame_data))

def __del__(self):
    # Closing files
    self.cout('Closing␣files')
    if len(self.user_frames_files) > 0:
        for stream_index in range(len(self.user_frames_files)):
            self.user_frames_files[stream_index].close()
    if len(self.user_data_checker_files) > 0:
        for stream_index in range(len(self.user_data_checker_files)):
            self.user_data_checker_files[stream_index].close()

def __feed_coefficients(self, stream_index, supercell_index, coeff_nb, fir_coeff_float, sat_coeff_float):
    fir_fixed_point = int(fir_coeff_float*(2**self.USER_FIR_COEFF_BITSHIFT))
    sat_fixed_point = int(sat_coeff_float*(2**self.USER_SAT_COEFF_BITSHIFT))
    fir_twos_complement = self.signed_int_to_two_compl_int(self.USER_COEFFICIENT_WIDTH, fir_fixed_point)
    sat_twos_complement = self.signed_int_to_two_compl_int(self.USER_COEFFICIENT_WIDTH, sat_fixed_point)
    self.coeff_fir[stream_index][supercell_index][coeff_nb]['float'] = fir_coeff_float
    self.coeff_sat[stream_index][supercell_index][coeff_nb]['float'] = sat_coeff_float
    self.coeff_fir[stream_index][supercell_index][coeff_nb]['c-model'] = fir_fixed_point
    self.coeff_sat[stream_index][supercell_index][coeff_nb]['c-model'] = sat_fixed_point
    self.coeff_fir[stream_index][supercell_index][coeff_nb]['integer'] = fir_twos_complement
    self.coeff_sat[stream_index][supercell_index][coeff_nb]['integer'] = sat_twos_complement
    self.coeff_fir[stream_index][supercell_index][coeff_nb]['file-format'] = '0x{0:04x}'.format(fir_twos_complement)
    self.coeff_sat[stream_index][supercell_index][coeff_nb]['file-format'] = '0x{0:04x}'.format(sat_twos_complement)

def __save_coefficients(self):
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        # TODO: check order
        for coeff in range(4)[::-1]:
            for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
                self.coeff_fir_mif_files[stream_index].dump(BitVector(size = self.USER_COEFFICIENT_WIDTH, intVal = self.
                    ↪ coeff_fir[stream_index][supercell_index][coeff]['integer']))
                self.coeff_sat_mif_files[stream_index].dump(BitVector(size = self.USER_COEFFICIENT_WIDTH, intVal = self.
                    ↪ coeff_sat[stream_index][supercell_index][coeff]['integer']))
                # Debugging
                if stream_index == self.dbg_user_stream:# and supercell_index == self.dbg_user_sc_id:
                    self.cout_debug('coefficients␣␣␣', [stream_index,
                                                          supercell_index,
                                                          coeff,
                                                          self.coeff_fir[stream_index][supercell_index][coeff]['float'],
                                                          self.coeff_fir[stream_index][supercell_index][coeff]['c-model'],
                                                          self.coeff_fir[stream_index][supercell_index][coeff]['integer'],
                                                          self.coeff_fir[stream_index][supercell_index][coeff]['file-format'],
                                                          ])
            for supercell_index in range(2):
                self.coeff_fir_mif_files[stream_index].dump(BitVector(size = self.USER_COEFFICIENT_WIDTH, intVal = 0))
                self.coeff_sat_mif_files[stream_index].dump(BitVector(size = self.USER_COEFFICIENT_WIDTH, intVal = 0))

    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
            self.coeff_fir_mif_files[stream_index].dump(BitVector(size = self.USER_COEFFICIENT_WIDTH, intVal = int(self.
                ↪ USER_CODE_GLOBAL_PEDESTAL*(2**self.USER_PEDESTAL_HARDPOINT))))
            self.coeff_sat_mif_files[stream_index].dump(BitVector(size = self.USER_COEFFICIENT_WIDTH, intVal = int(self.
                ↪ USER_CODE_GLOBAL_PEDESTAL*(2**self.USER_PEDESTAL_HARDPOINT))))

        self.coeff_fir_mif_files[stream_index].close()
        self.coeff_sat_mif_files[stream_index].close()

def __get_areus_coefficients(self, remap_frame):
    # GET AREUS COEFFICIENTS IF ACTIVATED (process only at the first feed)
    self.cout('Loading␣AREUS␣coefficients␣(FIR,␣SAT)')
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
            sc_id = remap_frame['data'][stream_index]['sc_id'][supercell_index]
            # TODO: Verify invertation of coefficient order
            for coeff in range(4)[::-1]:
                fir_float = float(self.AM_et_coeff.request(sc_id))
                sat_float = float(self.AM_tau_coeff.request(sc_id))
                self.__feed_coefficients(stream_index, supercell_index, coeff, fir_float, sat_float)

def __generate_c_arrays(self):
    # Prepare input and output c-types to transfer pointers into the C-model
    # C-array sizes
    self.input_c_size = int(self.USER_CODE_OSUM_DATA_STREAMS_NB * self.USER_CODE_STREAM_NB_SAMPLES)
    self.input_c_size_tap = int(self.USER_CODE_OSUM_DATA_STREAMS_NB * self.USER_CODE_STREAM_NB_SAMPLES * self.
       ↪ USER_FILTERING_IN_FIR_FILTER_TAPS_NB)

    # Init all C-arrays
    setattr(self, 'input_c_config', (c_int16*10)())
    setattr(self, 'input_c_config_mode', (c_bool*3)())
    setattr(self, 'input_c_pedestal', (c_int16*self.input_c_size)())
    for c_array in ['adc_samples', 'valid', 'coeff_fir', 'coeff_sat']:
        setattr(self, 'input_c_%s' % (c_array), (c_int16*self.input_c_size_tap)())
    for c_array in ['valid', 'quality']:
        setattr(self, 'output_c_' + c_array, (c_int16*self.input_c_size_tap)())
    for c_array in ['energy_i', 'energy_s', 'energy_cs', 'energy_bcs', 'tau', 'adc_ped']:
        setattr(self, 'output_c_' + c_array, (c_int32*self.input_c_size_tap)())



    # Fill C-Model config
    self.input_c_config[0] = self.USER_CODE_OSUM_DATA_STREAMS_NB
    self.input_c_config[1] = self.USER_FILTERING_IN_FIR_FILTER_TAPS_NB
    self.input_c_config[2] = self.USER_FILTERING_IN_SATURATION_DETECTION_TAPS_NB
    self.input_c_config[3] = self.USER_CODE_STREAM_NB_SAMPLES
    self.input_c_config[4] = self.USER_FILTERING_IN_FIR_FILTER_TAPS_NB * self.USER_CODE_STREAM_NB_SAMPLES
    self.input_c_config[5] = self.USER_ENERGY_WIDTH
    self.input_c_config[6] = self.USER_HARDPOINT_IN_FIR_FILTER
    self.input_c_config[7] = self.USER_ENERGY_TAU_WIDTH
    self.input_c_config[8] = self.USER_HARDPOINT_IN_SATURATION_DETECTION
    self.input_c_config[9] = 16

    self.input_c_config_mode[0] = self.USER_TAU_SELECTION_ENABLE
    self.input_c_config_mode[1] = self.USER_COMBINE_BLOCK_ENABLE
    self.input_c_config_mode[2] = self.USER_HARDPOINT_IN_FIR_FILTER_ENABLE

    # Fill pedestal, ADC, valid and coefficients(+init_dump)
    i = 0
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
            self.input_c_pedestal[i] = int(self.USER_CODE_GLOBAL_PEDESTAL*(2**self.USER_PEDESTAL_HARDPOINT))
            i += 1
    j = 0
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        # TODO: check order
        for coeff in range(4):
            for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
                self.input_c_coeff_fir[j] = self.coeff_fir[stream_index][supercell_index][coeff]['c-model']
                self.input_c_coeff_sat[j] = self.coeff_sat[stream_index][supercell_index][coeff]['c-model']
                self.input_c_adc_samples[j] = self.USER_CODE_GLOBAL_PEDESTAL
                self.input_c_valid[j] = 1
                j += 1

def __dump_user_frame(self, frame_index, frame):
    bcid = frame['bcid']
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
            self.user_frames_files[stream_index].write('%02d,␣,%05d,0x%04x,␣,%04d,0x%s,␣,%s,␣,%s,␣0x%s,␣,0x%s\n' %
                (stream_index,
                 frame_index, frame_index,
                 bcid.intValue(), bcid.get_bitvector_in_hex(),
                 frame['data'][stream_index]['startofpacket'][supercell_index],
                 frame['data'][stream_index]['valid'][supercell_index],
                 frame['data'][stream_index]['quality'][supercell_index].get_bitvector_in_hex(),
                 frame['data'][stream_index]['energy_data'][supercell_index].get_bitvector_in_hex()))

def __dump_user_data_checker(self, frame_index, frame):
    bcid = frame['bcid']
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
            bitstring = str(bcid) + str(frame['data'][stream_index]['startofpacket'][supercell_index]) + str(frame['data'][
                ↪ stream_index]['valid'][supercell_index]) + str(frame['data'][stream_index]['quality'][supercell_index]) + str(
                ↪ frame['data'][stream_index]['energy_data'][supercell_index])
            self.user_data_checker_files[stream_index].dump(BitVector(bitstring = bitstring))

def feed(self, remap_frame):
    """
    Data types of the USER_model should be:
    Input:
        adc_sample : 12 bit unsigned integer
        pedestal : 14 bit = 11 bit integer unsigned + 3 bit fixed point
        coefficients : 14 bit = 9 bit signed integer + 5 bit fixed point (if hardpoint = 8)
        valid : 1 bit unsigned integer

    Output:
        transverse energy : 18 bit signed integer (LSB=12.5MeV with HARDPOINT = 8)
        quality 4 bit = 1 bit saturation flag + 1 bit BCAV flag + 2 bit zeros
        valid : 1 bit unsigned integer
    """

    self.frame_index += 1

    #################################
    # PARSING REMAP FRAME FOR M3 (if loop-mode is activated)
    #################################
    if self.LATOME_FW_MODEL_LOOP_MODE:
        if self.frame_index < self.LATOME_FW_MODEL_LOOP_DEPTH:
            # For the first 3 BCIDs set REMAP BCID to zero (remap_bcid = 0) and buffer remap_frames
            remap_bcid = BitVector(size = 12, intVal = 0)
            self.buffer_remap_frame_m3[self.frame_index] = remap_frame
        elif self.frame_index < self.LATOME_FW_MODEL_EXEC_NB_BCID:
            # For the remaining BCIDs shift REMAP BCID (remap_bcid = bcid - 3)
            remap_bcid = BitVector(size = 12, intVal = (remap_frame['bcid'].intValue() - self.LATOME_FW_MODEL_LOOP_DEPTH)
                ↪ )
        else:
            # For the last 3 BCIDs shift REMAP BCID (remap_bcid = bcid - 3) and use dumped remap_frames
            remap_frame = self.buffer_remap_frame_m3[self.frame_index % self.LATOME_FW_MODEL_EXEC_NB_BCID]
            remap_bcid = BitVector(size = 12, intVal = int(self.frame_index - self.LATOME_FW_MODEL_LOOP_DEPTH))
    else:
        # Get REMAP BCID
        remap_bcid = remap_frame['bcid']

    if len(remap_frame['data']) != self.USER_CODE_OSUM_DATA_STREAMS_NB:
        raise Exception("Incompatible␣number␣of␣streams␣from␣remap␣(%d)␣to␣user␣(%d)" % (len(remap_frame['data']),
            ↪ self.USER_CODE_OSUM_DATA_STREAMS_NB))
```

```
##################################
# INIT C−ARRAYS, GET AREUS COEFFICIENTS IF ACTIVATED (process only at the first feed)
##################################
if self.output_type_c_model and self.frame_index == 0:
    if self.USER_CODE_COEFF_TYPE == 'AREUS':
        self.__get_areus_coefficients(remap_frame)
    self.__generate_c_arrays()
    self.__save_coefficients()

##################################
# PROCESS REMAP DATA WITH C−MODEL
##################################
if self.output_type_c_model:
    # Get REMAP data to prepare user_code C−model execution
    for bcid in range(0, self.input_c_size_tap, self.input_c_size):
        i = 0
        for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
            for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
                if bcid < 3*self.input_c_size:
                    # Shift values from the last 3 BCIDs to the beginning of the C−arrays
                    self.input_c_adc_samples[bcid + i] = self.input_c_adc_samples[bcid + i + self.input_c_size]
                    self.input_c_valid[bcid + i] = self.input_c_valid[bcid + i + self.input_c_size]
                else:
                    # Fill the current REMAP data to the C−arrays at the place for BCID 4
                    self.input_c_adc_samples[bcid + i] = int(remap_frame['data'][stream_index]['adc_data'][supercell_index])
                    self.input_c_valid[bcid + i] = int(remap_frame['data'][stream_index]['valid'][supercell_index])
                i += 1

    # Execute the C−model
    self.user_c_model.main(self.input_c_config, self.input_c_config_mode, self.input_c_adc_samples, self.input_c_valid, self.
        ↪ input_c_pedestal, self.input_c_coeff_fir, self.input_c_coeff_sat, self.input_c_size, self.output_c_valid, self.
        ↪ output_c_quality, self.output_c_energy_i, self.output_c_energy_s, self.output_c_energy_cs, self.
        ↪ output_c_energy_bcs, self.output_c_tau, self.output_c_adc_ped)

    # Transfer C−model output to the user_frame
    user_c_frame = deepcopy(self.user_frame_empty)
    user_c_frame['bcid'] = remap_bcid
    user_areus_frame = None
    i = 0
    for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
        for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
            user_c_frame['data'][stream_index]['valid'][supercell_index] = int(self.output_c_valid[i])
            user_c_frame['data'][stream_index]['quality'][supercell_index] = BitVector(size = self.USER_QUALITY_WIDTH,
                ↪ intVal = int(self.output_c_quality[i]))
            user_c_frame['data'][stream_index]['energy_data'][supercell_index] = BitVector(size = self.USER_ENERGY_WIDTH,
                ↪ intVal = int(self.output_c_energy_i[i]))
            user_c_frame['data'][stream_index]['energy_data_float'][supercell_index] = self.fp_int_to_float_two(18, 0, self.
                ↪ output_c_energy_i[i])

            # Debugging
            if stream_index == self.dbg_user_stream: #and supercell_index == self.dbg_user_sc_id:
                self.cout_debug('incoming␣vs.␣C−model', [stream_index,
                                                supercell_index,
                                                remap_frame['data'][stream_index]['adc_data'][supercell_index].intValue(),
                                                remap_frame['data'][stream_index]['sc_id'][supercell_index],
                                                '␣␣␣/',
                                                user_c_frame['data'][stream_index]['energy_data_float'][supercell_index],
                                                self.output_c_valid[i],
                                                self.output_c_quality[i],
                                                #user_c_frame['data'][stream_index]['quality'][supercell_index][3],
                                                #remap_frame['data'][stream_index]['adc_data'][supercell_index].intValue(),
                                                #remap_frame['data'][stream_index]['valid'][supercell_index],
                                                #self.output_c_energy_i[i],
                                                #self.output_c_energy_s[i],
                                                #self.output_c_energy_cs[i],
                                    #self.output_c_tau[i]
                                    #self.output_c_energy_bcs[i],
                                    #self.output_c_adc_ped[i]]),
                                    ])
            i += 1
        i += 18

    ##################################
    # REPLACE REMAP ADC−DATA WITH AREUS OUTPUT
    ##################################
    if self.output_type_areus:
        user_areus_frame = deepcopy(self.user_frame_empty)
        user_areus_frame['bcid'] = remap_bcid
        # Dumping AREUS energy data
        for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
            for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):

                remap_valid = remap_frame['data'][stream_index]['valid'][supercell_index]
                if remap_valid == '1':
                    # Get AREUS energy data for eT and tau [GeV]
                    remap_sc_id = remap_frame['data'][stream_index]['sc_id'][supercell_index]
                    bcid_et, data_et_gev = self.AM_et_data.request(remap_sc_id)
                    bcid_tau, data_tau_gev = self.AM_tau_data.request(remap_sc_id)
                    if self.LATOME_FW_MODEL_LOOP_MODE:
                        if self.frame_index >= self.LATOME_FW_MODEL_LOOP_DEPTH:
                            if remap_bcid.intValue() != (bcid_et − self.LATOME_FW_MODEL_LOOP_DEPTH) or remap_bcid.
                                ↪ intValue() != (bcid_tau − self.LATOME_FW_MODEL_LOOP_DEPTH):
                                raise Exception('BCID␣collision␣at␣supercell:␣%s.␣Requested␣BCID:␣%s,␣AREUS␣et␣BCID:␣%s,
                                    ↪ ␣AREUS␣tau␣BCID:␣%s' % (remap_sc_id, remap_bcid.intValue(), bcid_et, bcid_tau))
                    else:
                        if remap_bcid.intValue() != bcid_et or remap_bcid.intValue() != bcid_tau:
                            raise Exception('BCID␣collision␣at␣supercell:␣%s.␣Requested␣BCID:␣%s,␣AREUS␣et␣BCID:␣%s,␣
                                ↪ AREUS␣tau␣BCID:␣%s' % (remap_sc_id, remap_bcid.intValue(), bcid_et, bcid_tau))
                    # Scaling + LSB (=12.5MeV)
                    data_et_mev_lsb = round((data_et_gev*1000)/12.5)
                    data_tau_mev_lsb = round((data_tau_gev*1000)/12.5)

                    user_areus_frame['data'][stream_index]['energy_data'][supercell_index] = BitVector(size = self.
                        ↪ USER_ENERGY_WIDTH, intVal = int(self.signed_int_to_two_compl_int(18,data_et_mev_lsb)))
                    user_areus_frame['data'][stream_index]['energy_data_float'][supercell_index] = round(data_et_mev_lsb, 5)

                    # TODO: add quality calculation using AREUS tau data
                    user_areus_frame['data'][stream_index]['quality'][supercell_index] = BitVector(size = self.
                        ↪ USER_QUALITY_WIDTH, intVal = 4)
                    user_areus_frame['data'][stream_index]['valid'][supercell_index] = 1

                else:
                    remap_sc_id = None
                    user_areus_frame['data'][stream_index]['energy_data'][supercell_index] = BitVector(size = self.
                        ↪ USER_ENERGY_WIDTH, intVal = 0)
                    user_areus_frame['data'][stream_index]['quality'][supercell_index] = BitVector(size = self.
                        ↪ USER_QUALITY_WIDTH, intVal = 0)
                    user_areus_frame['data'][stream_index]['valid'][supercell_index] = 0

                # Debugging
                if stream_index == self.dbg_user_stream: #and supercell_index == self.dbg_user_sc_id:
                    self.cout_debug('incoming␣vs.␣AREUS', [stream_index,
                                                  supercell_index,
                                                  remap_frame['data'][stream_index]['adc_data'][supercell_index].intValue
                                                      ↪ (),
                                                  str(remap_sc_id),
                                                  '␣␣␣/',
                                                  user_areus_frame['data'][stream_index]['energy_data_float'][
                                                      ↪ supercell_index],
                                                  user_areus_frame['data'][stream_index]['valid'][supercell_index],
                                                user_areus_frame['data'][stream_index]['quality'][supercell_index].
                                                    ↪ intValue(),
                                                #round(data_tau_gev, 5),
                                                ])

    #################################
    # DUMP AND RETURN
    #################################
    if self.output_type_c_model:
        self.__compare_energy_data(user_c_frame, user_areus_frame, remap_bcid)
        # Dump C−model user_frame
        if self.LATOME_FW_MODEL_LOOP_MODE:
            if self.frame_index < self.LATOME_FW_MODEL_LOOP_DEPTH:
                return None
            else:
                self.__dump_user_frame(self.frame_index − self.LATOME_FW_MODEL_LOOP_DEPTH , user_c_frame)
                self.__dump_user_data_checker(self.frame_index − self.LATOME_FW_MODEL_LOOP_DEPTH, user_c_frame)
                return user_c_frame
        else:
            self.__dump_user_frame(self.frame_index, user_c_frame)
            self.__dump_user_data_checker(self.frame_index, user_c_frame)
            return user_c_frame
    elif self.output_type_areus:
        # Dump AREUS user_frame
        self.__dump_user_frame(self.frame_index, user_areus_frame)
        self.__dump_user_data_checker(self.frame_index, user_areus_frame)
        return user_areus_frame


def __compare_energy_data(self, user_c_frame, user_areus_frame, remap_bcid):
    if self.output_type_areus and self.output_type_c_model:
        # Compare C−model and AREUS, dump and return C−model
        for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
            if stream_index not in self.dbg_c_e.keys():
                self.dbg_c_e[stream_index] = {}
                self.dbg_a_e[stream_index] = {}
            for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
                if supercell_index not in self.dbg_c_e[stream_index].keys():
                    self.dbg_c_e[stream_index][supercell_index] = 'c:{0:3s}␣'.format(str(stream_index))
                    self.dbg_c_e[stream_index][supercell_index] += '{0:3s}␣'.format(str(supercell_index))
                    self.dbg_a_e[stream_index][supercell_index] = 'a:{0:3s}␣'.format(str(stream_index))
                    self.dbg_a_e[stream_index][supercell_index] += '{0:3s}␣'.format(str(supercell_index))
                # Debugging
                if stream_index == self.dbg_user_stream: #and supercell_index == self.dbg_user_sc_id:
                    self.cout_debug('C−model␣vs.␣AREUS', [stream_index,
                                                        supercell_index,
                                                        remap_bcid.intValue(),
                                                        user_c_frame['data'][stream_index]['energy_data_float'][supercell_index],
                                                        user_areus_frame['data'][stream_index]['energy_data_float'][supercell_index],
                                                        #user_c_frame['data'][stream_index]['energy_data'][supercell_index].intValue(),
                                                        #user_areus_frame['data'][stream_index]['energy_data'][supercell_index].intValue(),
                                                        user_c_frame['data'][stream_index]['valid'][supercell_index],
                                                        user_areus_frame['data'][stream_index]['valid'][supercell_index],
                                                        user_c_frame['data'][stream_index]['quality'][supercell_index].intValue(),
                                                        user_areus_frame['data'][stream_index]['quality'][supercell_index].intValue(),
                                                        ])
                c_e = user_c_frame['data'][stream_index]['energy_data_float'][supercell_index]
                a_e = user_areus_frame['data'][stream_index]['energy_data_float'][supercell_index]
                if c_e < 10 and a_e < 10:
                    c_e = 0.0
                    a_e = 0.0
                self.dbg_c_e[stream_index][supercell_index] += '{0:5s}'.format(str(int(c_e)))
                self.dbg_a_e[stream_index][supercell_index] += '{0:5s}'.format(str(int(a_e)))
         # Debugging
        if remap_bcid.intValue() == self.LATOME_FW_MODEL_EXEC_NB_BCID−1:
            a = open('%s/%s' % (self.LATOME_FW_MODEL_OUTPUT_PATH, self.dbg_fir_vs_areus_filename), 'w')
            for stream_index in range(self.USER_CODE_OSUM_DATA_STREAMS_NB):
                for supercell_index in range(self.USER_CODE_STREAM_NB_SAMPLES):
                    a.write('%s\n' % self.dbg_c_e[stream_index][supercell_index])
                    a.write('%s\n' % self.dbg_a_e[stream_index][supercell_index])
            a.close()

def fp_int_to_float_two(self, s_width, f_width, fp_int):
    # Fixed point integer to float converter, "two's complement"
    tot_width = s_width + f_width
    bitstr = bin(fp_int)[2:]
    bitstr = str(tot_width*'0')[:-len(bitstr)] + bitstr
    data = 0.0
    neg = bitstr[0] == "1"
    if neg:
        # bin complement
        tmpBin = list(bitstr)
        for i, c in enumerate(tmpBin):
            if c == "1":
                tmpBin[i] = "0"
            else:
                tmpBin[i] = "1"
        bitstr = ''.join(tmpBin)
        # bin increment
        tmpBin = list(bitstr)
        for i, c in enumerate(tmpBin[::-1]):
            if c == "0":
                tmpBin[-i-1] = "1"
                break
            else:
                tmpBin[-i-1] = "0"
        bitstr = ''.join(tmpBin)
    for i, c in enumerate(bitstr[::-1]):
      if c == "1":
        pow_val = i - f_width
        data = data + (2**pow_val)
    if neg:
      data = -data
    return data

def signed_int_to_two_compl_int(self, width, val_int):
    # signed integer to unsigned two's complement integer converter
    s = bin(int(val_int) & int("1"*width, 2))[2:]
    bin_str = ("{0:0>%s}" % (width)).format(s)
    return int(bin_str, 2)
```

List of Terms and Abbreviations

ADC	Analog-to-Digital-Converter
AMC	Advanced Mezzanine Card
AREUS	ATLAS Readout Electronics Upgrade Simulation
ATCA	Advanced Telecommunications Computing Architecture
ATLAS	A Toroidal LHC ApparatuS
BC	Bunch Crossing
BCid	Bunch Crossing ID
CERN	European Organization for Nuclear Research
CMS	Compact Muon Solenoid
DAQ	Data Acquisition System
eFEX	electronic Feature Extraction
EMB	Electro-Magnetic Barrel
EMEC	Electro-Magnetic End-Cap
Fcal	Forward Calorimeter
FEX	Feature Extractor
FIR	Finite Impulse Reponse
FPGA	Field Programmable Gate Array
GEANT4	Geometry and Tracking
gFEX	global Feature Extraction
GUI	Graphical User Interface
HEC	Hadronic End-Cap
IPBus	IP-based control protocol for ATCA
IPCTRL	IPBus Controller
IPv4	Internet Protocol version 4
ISTAGE	Input Stage
jFEX	jet Feature Extraction
L1 Calo	Level 1 Carorimeter Trigger
L1A	Level 1 Accept
LATOME	LAr Trigger prOcessing MEzzanine

LAr	Liquid Argon Calorimeter
LDPS	LAr Digital Processing System
LHC	Large Hadron Collider
LLI	Low Level Interface
LSB	Least Significant Bit
MON	TDAQ Monitoring
LTDB	LAr Trigger Digitizer Board
OSUM	Output Summing
PHY	physical layer of the OSI model
QCD	Quantum Chromodynamics
QED	Quantum Electrodynamics
REMAP	Configurable Remapping
SC	Super Cell
SM	Standard Model
TDAQ	ATLAS Trigger and Data Acquisition
TT	Trigger Tower
TTC	Timing, Trigger and Control
USER	User Code
VHDL	Very High Speed Integrated Circuit Hardware Description Language
VirtNet	Virtual Network Handler

List of Figures

List of Tables

Bibliography

[1] CERN Accelerators
http://maalpu.org/lhc/LHC.main.htm 7

[2] Standard Model of Elementary Particles
https://en.wikipedia.org/wiki/Elementary_particle 3

[3] Peter Steinberg, ATL-PHYS-PROC-2011-073arXiv,Recent Heavy Ion Results with the ATLAS Detector at the LHC
https://cds.cern.ch/record/1364846/plots 8

[4] Electronics Development for the ATLAS Liquid Argon Calorimeter Trigger and Readout for Future LHC Running. 2016
https://cds.cern.ch/record/2141444/files/ATL-LARG-PROC-2016-001.pdf
9, 10, 14, 17

[5] Stefan Mättig, „The Atlas Trigger“, 2007
https://wwwiexp.desy.de/group/schleper/lehr/UniHHSeminarSS07/maettig_0507.pdf 12

[6] Prof. Dr. S. Bethke, „LHC Experimente – Trigger, Datennahme und Computing“, 2012
https://www.mpp.mpg.de/graduate-de/vorlesungen/WiSe1213/SimonHochenergieTeilchenphysik_WS12/pdf/WS1213-05.pdf 13

[7] CERN records, 2017
http://strangesounds.org/2017/06/cern-record-luminosity/unprecedented-number-particle-collisions-reached.html 5

[8] ATLAS Phase-II Upgrade Scoping Document, 2015
http://cds.cern.ch/record/2055248 14

[9] WalterHopkins, Electronics development for the ATLAS liquid argon calorimeter trigger and readout for future LHC running
https://www.sciencedirect.com/science/article/pii/S0168900216302091

[10] CERN-LHCC-2013-017, ATLAS-TDR-022, Liquid Argon Calorimeter Phase-I upgrade TDR
https://cds.cern.ch/record/1602230?ln=en

[11] PEP 3143 – Standard daemon process library
https://www.python.org/dev/peps/pep-3143/ 27

[12] Cocotb Github repository
https://github.com/potentialventures/cocotb 27

[13] LATOME Firmware specifications, unpublished and under development, 2018
https://gitlab.cern.ch/atlas-lar-be-firmware/LATOME/LATOME/blob/master/LATOME/doc/LAr-LATOME-FW/LAr-LATOME-FW.pdf 16, 17, 18, 19, 21

[14] Performance of the ATLAS Liquid Argon Calorimeter after three years of LHC operation and plans for a future upgrade - ATLAS Collaboration
http://inspirehep.net/record/1240499/plots 11

[15] J. P. Grohs and S. Stärz, ATL-COM-LARG-2013-012, AREUS - ATLAS Readout Electronics Upgrade Simulation, 2013
https://cds.cern.ch/record/1546755 37, 38, xiv

[16] Trigger paths for Trigger Tower and Super Cells
https://twiki.cern.ch/twiki/bin/view/AtlasPublic/LuminosityPublicResultsRun2 15

[17] Arria 10 GX FPGA Development Kit from Intel, 2018
https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/kit-a10-gx-fpga.html 22

Versicherung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer anderen Prüfungsbehörde vorgelegt.

Die vorliegende Masterarbeit wurde am Institut für Kern- und Teilchenphysik der Technischen Universität Dresden unter wissenschaftlicher Betreuung von Prof. Dr. Arno Straessner angefertigt.

Dresden, den 13. August 2018

Yves Bianga, Bachelor of science