Lossy and lossless data compression of data from high energy physics experiments

Verlustbehaftete und verlustlose Datenkomprimierung für Daten von Hochenergiephysik Experimenten

DISSERTATION

to attain the academic degree “Dr. techn.”

submitted by

Dipl.-Ing. Christian Pataurer

at the

Institute of Electronics,
Graz University of Technology

In collaboration with
CERN - European Organization for Nuclear Research

Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Wolfgang Pribyl (TU-Graz/IFE)
Co-Supervisor:	Univ.-Prof. Dipl.-Ing. Dr. Christian Fabjan (ÖAW/HEPHY,TU-Wien)
CERN Supervisor:	Dr. Alessandro Marchioro (CERN/PH-ESE-ME)

Graz, November 2011

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ....
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

....
date (signature)

Abstract

This dissertation describes a data compression system optimized for high energy particle detectors. The aim is to reduce the data in the front-end electronics installed in different kind of particle detectors as used for example along the Large Hadron Collider at CERN. The signals collected and digitized from calorimeters, time projection chambers and in general from all detectors with a large number of sensors produce an extensive amount of data, which need to be reduced. Real-time data compression algorithms applied right at the detector front-end is able to reduce the amount of data to be transmitted and stored as early as possible in the data chain.

Different lossless and lossy compression methods are analyzed regarding their efficiency in compressing data from particle detectors that produce signals amplified and/or shaped to various waveforms. In addition, the complexity of the algorithms is evaluated to determine their suitability for a real-time hardware implementation. From the analyzed methods, a new developed lossless compression method turned out to be the best suitable one for the implementation in high energy physics applications. The detector data are used to search for rare particle physics phenomena, which makes it crucial that the compression method retains the important information in the data with an appropriate accuracy. Considering the importance of not distorting detector data, a lossless compression method was preferred instead of a lossy method.

To go beyond what can be achieved by conventional lossless compression schemes, which are mostly limited by the intrinsic entropy of the underlying data, the proposed compression method makes use of a new scheme where entire vectors of samples are compressed instead of handling the data from the ADCs as individual uncorrelated samples.

Our method works by first approximating the incoming vectors, formed by the digitization of the shaped input waveforms from the detector signals, with a set of digitized reference vectors. This is generally known as vector quantization. To prevent information loss the differences between the incoming vector and the best matching reference vector are retained. These differences are then Huffman encoded to obtain the compression. The performance of the compression method was first evaluated by modeling the algorithm in Matlab and using test-data measured with the time projection chamber in the ALICE experiment at CERN. A compression ratio of 50% has been achieved for this test-data (better as the intrinsic entropy of the test-data of 62%).

For a demonstration of the functionality of the developed compression method in hardware, a digital IP block was realized and modeled using the hardware description language Verilog. The compression algorithm was optimized for the data from a time projection chamber and tested using a FPGA development board applying the same test-data from the ALICE time projection chamber as used previously in Matlab. The implementation achieved almost the same compression ratio.

In this thesis, I show that a data compression of digitized detector data is possible to be realized in the detector front-end electronics very close to the data source by still maintaining the accuracy of the data. The developed and realized lossless compression algorithm achieved a compression ratio of about 50%. The hardware implementation of the algorithm proved its real-time suitability by compressing 10 000 consecutive input signals without introducing dead time. Only an average latency of about 30 clock cycles of 40 MHz has to be accepted.

The designed data compression IP block is available for an implementation in current and future detector front-end electronics either inside FPGAs or inside full custom ASICs. The compression block requires about 2 700 logic slices inside a Virtex-4 FPGA and around 12 200 gates for an ASIC implementation without taking into account the required memory of 7 kbyte.

Kurzfassung

Diese Dissertation beschreibt eine Komprimierungsmethode, welche für Anwendungen in der Hochenergiephysik optimiert ist. Das Ziel dieser Komprimierung ist die Reduktion der Daten, die in der Front-End-Elektronik von Teilchendetektoren entstehen, welche zum Beispiel entlang des Large Hadron Colliders am CERN installiert sind. Die Menge der digitalisierten Signale von Kalorimetern, Zeit-Projektionskammern (TPC) und im Generellen allen Detektoren die eine große Anzahl von Sensoren besitzen, bringt die Anforderungen an die Datenübertragung und Speicherung an ihre Limits. Eine Echtzeit-Datenkomprimierung in der Front-End-Elektronik von Detektoren kann diese Datenmengen frühestmöglich im Datenpfad reduzieren um die Datenübertragung und Speicherung zu vereinfachen.

Verschiedene verlustlose und verlustbehaftete Komprimierungsmethoden werden hinsichtlich ihrer Effizienz untersucht. Die Detektordaten bestehen dabei aus Signalen, die durch die Verstärkung und/oder Signalformung verschiedene digitalisierte lineare Impulsformen ergeben können. Zusätzlich wird die Komplexität der Methoden evaluiert um ihre Einsatzfähigkeit in Echtzeit-Hardware zu erörtern. Die Detektordaten werden benutzt um seltene Phänomene in der Teilchenphysik zu erforschen und das bedingt, dass die Komprimierungsmethode die enthaltenen Informationen mit der nötigen Genauigkeit erhält. Daher wurde eine verlustlose Komprimierung bevorzugt entgegen verlustbehafteten Methoden. Eine neu entwickelte verlustlose Komprimierungsmethode hob sich dabei als die Bestgeeignetste für die Datenkomprimierung in Hochenergiephysik-Anwendungen hervor.

Die vorgeschlagene Komprimierungsmethode bedient sich einer neuen Methodik, welche die digitalen Daten nicht als einzelne unabhängige Abtastwerte sieht, sondern einen Vektor von mehreren Abtastwerten zugleich komprimiert. Dadurch kann eine höhere Effizienz erzielt werden als bei herkömmlichen verlustlosen Methoden, welche meistens durch die intrinsische Entropie limitiert sind.

Die entwickelte Methode vergleicht zuerst die Eingangsvektoren, gebildet von den Abtastwerten der Eingangssignale welche von den Detektor-Sensoren kommen, mit einem Set von Referenzvektoren. Der Index des ähnlichsten Referenzvektors wird ausgesendet. Dies wird im Allgemeinen als Vektor-Quantisierung bezeichnet. Zusätzlich werden die Differenzwerte zwischen dem Eingangsvektor und dem verwendeten Referenzvektor berechnet um einen Informationsverlust zu vermeiden. Diese Differenzwerte werden mit Huffman kodiert um eine Komprimierung zu erreichen. Die Komprimierungsmethode wurde zuerst in Matlab modelliert und mit Testdaten von Messungen mit dem TPC in ALICE getestet. Eine Komprimierungsrate von 50% konnte für diese Testdaten erzielt werden (die Entropie der Testdaten liegt bei 62%).

Um die Funktionalität des entwickelten Komprimierungsverfahrens in Hardware zu demonstrieren wurde ein digitaler IP-Block mit der Hardwarebeschreibungssprache Verilog designt. Der Kompressionsalgorithmus wurde für TPC-Daten optimiert und in einem FPGA Evaluationskit mit denselben Testdaten wie für das Matlab Model getestet. Die Implementierung errichte annähernd dieselbe Kompressionsrate.

In dieser Dissertation konnte ich zeigen, dass eine Datenkomprimierung von digitalisierten Detektordaten nahe der Datenquelle in der Front-End-Elektronik unter Beibehaltung der Datengenauigkeit möglich ist. Der entwickelte und realisierte Komprimierungsalgorithmus erreicht eine Effizienz von rund 50%. Die Hardware Implementierung bewies ihre Echtzeit-Einsetzbarkeit durch die kontinuierliche Komprimierung von 10 000 Eingangsvektoren ohne dabei eine Totzeit zu erzeu-

gen. Lediglich eine Zeitverzögerung von ca. 30 Taktzyklen von 40 MHz muss in Kauf genommen werden.

Für eine FPGA Implementierung des entwickelten Komprimierungs-Blocks in zukünftigen Experimenten werden ca. 2 700 Slices benötigt und für eine Realisierung in einem ASIC benötigt man ca. 12 200 Grundgatter ohne Berücksichtigung des Speichers von 7 kbyte.

Acknowledgements

The author would like to thank everybody who directly or indirectly supported this work. This is more appreciated than many words can say.

Einen besonderen Dank möchte ich meinen Eltern Rosa und Helmut aussprechen die mich während meines gesamten Studiums immer unterstützt haben. Ihr habt mich immer zum Weitermachen motiviert und ohne eure Hilfe hätte ich es sicherlich nicht bis zum Doktorat geschafft. Vielen Dank!

This thesis was performed within the Austrian Doctoral Student Program at CERN and supported by the Austrian Federal Ministry of Science and Research.

List of Abbreviations

AD1	DCAddrDec Address Decoder module
ALICE	A Large Ion Collider Experiment
ALTRO	ALice Tpc Read Out
ASIC	Application-Specific Integrated Circuit
ATLAS	A Toroidal LHC ApparatuS
CDH	Common Data Header
CERN	European Organization for Nuclear Research
Clock40	40 MHz clock signal
Clock80	80 MHz clock signal
CMOS	Complementary Metal Oxide Semiconductor
CMS	Compact Muon Solenoid
DAQ	Data Acquisition
DC	Data Compression block
DC1	DataCompressor module
DcMode	Data Compressor Mode
DCOF	DataCompressor Output Format
DCOP	DataCompressor Output data Package
DCS	Detector Control System interface
DCT	Discrete Cosine Transform
Div1	VQDiv Divider module
DWT	Discrete Wavelet Transform
ECAL	Electromagnetic CALorimeter
FEC	Front-End Card
FEE	Front-End Electronics
FIFO	First In First Out buffer
FIR	Finite Impulse Response filter
FPGA	Field Programmable Gate Array
FWHM	Full Width Half Maximum
HC1	HuffmanCoder module
HCAL	Hadron CALorimeter
HEP	High Energy Physics
HP	High-Pass filter
IIR	Infinite Impulse Response filter
IP	Intellectual Property block
ISE	Integrated Software Environment
ITS	Inner Tracking System
L1	Level 1 trigger
L2	Level 2 trigger
LBG	Yoseph Linde, Andrés Buzo and Robert M. Gray algorithm
LHC	Large Hadron Collider
LHCb	Large Hadron Collider beauty experiment
LHCf	Large Hadron Collider forward experiment
LP	Low-Pass filter

LSB	Least Significant Bit
LUT	Look-Up Table
LVDS	Low Voltage Differential Signal
MEB	Multi-Event Buffer
MF1	MaxFinder module
MSB	Most Significant Bit
MWPC	Multi-Wire Proportional Chamber
NZ1	Normalizer module
OF1	OutputFormater module
PASA	Pre-Amplifier Shaping Amplifier
PMT	Photo Multiplier Tube
PPM	Prediction with Partial string Matching
PSNR	Peak Signal to Noise Ratio
RCU	Readout Control Unit
RLE	Run Length Encoding
RTL	Register Transfer Level
SCA	Switched Capacitor Array
SDD	Silicon Drift Detector
SIU	Source Interface Unit
SUSY	SUper-SYmmetry
TDC	Time-to-Digital Converter
TOF	Time Of Flight detector
TOT	Time Over Threshold
TOTEM	TOTal cross section, Elastic scattering and diffraction dissociation Measurement
TPC	Time Projection Camber
TRD	Transition Radiation Detector
TRT	Transition Radiation Tracker
UART	Universal Asynchronous Receiver Transmitter
VQ1	VectorQuantizer1QSimple module
VQ4	VectorQuantizer4Q module

Contents

1. Introduction

1.1. Why data compression in front-end electronics is a useful tool in particle detectors?

Modern particle detectors are large structures, which contain a vast number of pads and readout channels. Particle detectors are used to collect momentum- and energy-information of particles created by particle collisions, with the aim of performing basic research in high energy physics.

Most of the modern particle detector types are installed along the LHC (Large Hadron Collider) particle accelerator located at CERN (European Organization for Nuclear Research) near Geneva that has been build to perform basic research in High Energy Physics (HEP). The LHC accelerates protons or heavy ions close to the speed of light and collides them in four different points where the four large detector experiments ALICE, ATLAS, LHCb and CMS are located.

In the light of new projects of particle accelerators and their required particle detectors, a study of a data compression method optimized for data from such detectors is developed in my PhD work. The aim of this work is to develop an implementation of a data compression algorithm inside the front-end electronics of future particle detectors to perform a data compression as close as possible to the source of the data (the detector pads) in order to reduce data transfer bandwidth and storage requirements. This could help to handle an increased number of channels in the confined space of future detector experiments.

Data compression is widely used in numerous computing and communication applications, ranging from text to music and video compression, but is currently relatively little used in data acquisition systems for particle physics. This possibly derives from two facts that data in particle physics often resembles random data even over a large number of samples. Random data cannot very effectively be reduced by compression. In addition, data compression methods are often difficult to be implemented in real-time hardware.

Nonetheless, trying to compress the data from particle detectors as much as possible makes sense if one considers the enormous amount of data produced in the vast number of channels of modern detector experiments (e.g. the ATLAS detector experiment at CERN has approximately one hundred million readout channels).

Already in current detector front-ends a data reduction is implemented based on trigger signals, which filter out only the physically interesting events from all the created signals. A lossy compression named zero suppression (based on run-length encoding) is used to separate the interesting data from the baseline data. However, all this effort is not enough to reduce the data to a level where they can be handled easily. Therefore, a further compression method for an implementation in the front-end electronics is developed and presented in this work. This method can provide a further compression of the interesting data by a factor of 2 without introducing additional information loss. The trend of modern detector data production is given in figure 1.1 to show the amount of data produced by recent particle detectors.

The development of the new data compression hardware is based on the requirements extracted from the Time Projection Chamber (TPC) front-end electronics in the ALICE experiment at CERN which I used as an example for a target application. Nevertheless, the data compression method is suitable for all detector front-ends, which digitize signals amplified and filtered (shaped) to various waveforms.

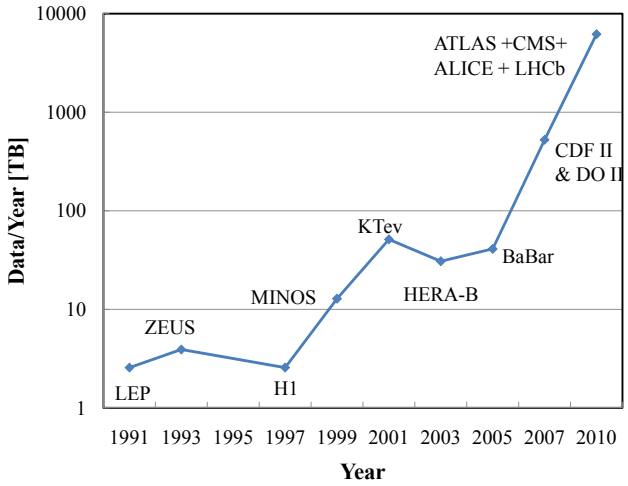


Figure 1.1.: Estimated data production of various particle detector experiments per year

The TPC is the main detector of the ALICE experiment and provides three-dimensional information of the trajectories from particles generated from the collisions. The ALICE TPC is the largest of such detectors built in recent years. It consists of an 88 m^3 cylinder filled with gas as shown in figure 1.2. A more detailed description of CERN, the LHC, ALICE and the TPC is given in chapter 2.

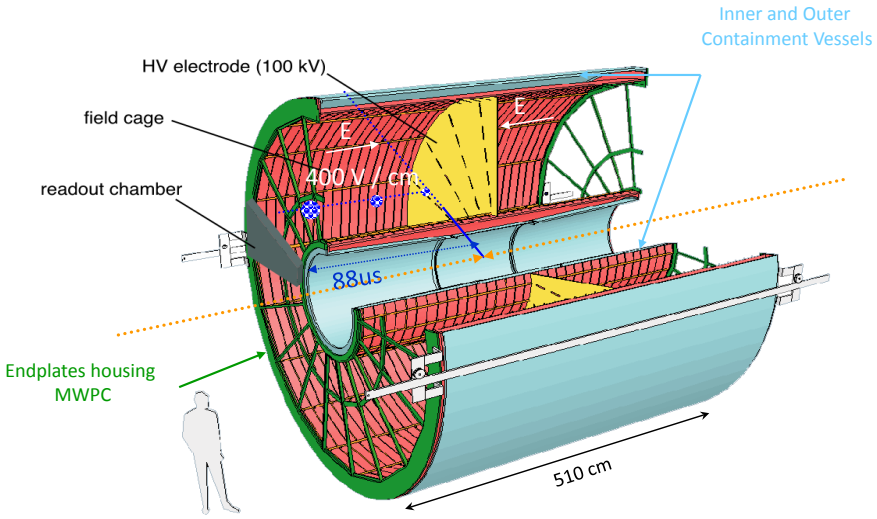


Figure 1.2.: ALICE Time Projection Chamber [1]

In case of ion beam collisions, up to 10 000 particles may traverse the TPC for each bunch crossing. The LHC will produce ion collisions with a frequency of 8 kHz. The rate of interesting events for the ALICE TPC is estimated at around 300 Hz and the event size is estimated at 60 MB after the zero suppression [2, 3]. With these estimations, the maximum data rate for the TPC can be calculated as follows:

$$D_{rate} = 60\,\mathrm{MB} \times 300\,\mathrm{Hz} \approx 20\,\mathrm{GB\,s^{-1}} \tag{1.1}$$

Considering that an experiment runs for years, this amount of data cannot be stored easily. The occupancy is expected to be low in ALICE and zero suppression is used for the first data reduction. This mechanism reduces the amount of data significantly ($\approx$ 80%). Nevertheless, the data rate

in (1.1) is estimated after the zero suppression and is still high. For a further data reduction, the developed data compression method can be used.

The data compression has to be realized in hardware into a confined space inside the detector area and with low power consumption for a reasonable low head production, which gives tight restrictions on the possible complexity of the compression algorithm. In addition, it is important that the loss of accuracy introduced by the compression is low in order to prevent the loss of any valuable measurement information from the physics point of view. The developed algorithm is presented in detail in chapter 4 comparing it to other promising compression methods.

The research work carried out by me in this dissertation is based on the investigation of several compression methods regarding the properties of the underlying digital data from the ALICE TPC and other detector concepts. A development of a compression algorithm with good compression performance and suitability for a real-time hardware implementation has been carried out. I have implemented the algorithm in hardware using a FPGA development board and optimized for a direct implementation in the front-end electronics of the TPC in ALICE. The functionality of the hardware implementation has been tested by me using measured data from the TPC. At the end of this study, I have compared the compression performance of the realized hardware with the predictions extracted from the simulated software model of the compression algorithm. A good match between the achieved real compression efficiency and predicted simulated efficiency was reached. I analyzed the complexity of the implementation and this can be used to estimate the effort and requirements for an implementation of the chosen compression method in future detector front-ends.

1.2. Organization of the document

Throughout this document, two measures are used to evaluate the performance of different compression methods, the compression ratio and the compression factor.

The compression ratio is defined as:

$$compression\ ratio = \frac{size\ of\ output\ stream}{size\ of\ input\ stream} \tag{1.2}$$

The compression ratio represents the ratio between the number of bits in the compressed data and the bits used for representing the original data. If the compression ratio is multiplied with 100 it is given in % which is normally used throughout this document. A compression ratio of 0.6 means therefore the compressed data set occupies 60% of the original data set. A compression ratio above 1 means expansion.

The compression factor is defined as:

$$compression\ factor = \frac{size\ of\ input\ stream}{size\ of\ output\ stream} \tag{1.3}$$

The compression factor is the inverse of the compression ratio and tells how much compression could be achieved. A factor grater then 1 defines compression whereas a compression below 1 represents expansion.

A third measure can be found in the literature, which is the space savings factor. The space savings factor in % is defined as $100 \times (1 - compression\ ratio)$.

The document itself is organized as follows:

The theory background in this document is divided in two parts in order that readers with a pre-knowledge in high energy physics detectors or in data compression can skip the corresponding

chapter.

Chapter 2 introduces the physics, which is important to understand the requirements put on the compression algorithm regarding the different kind of measurements and detector concepts. This chapter serves as well to get the reader familiar with some physical expressions used in this document. Therefore, this chapter includes a brief presentation of CERN and the LHC accelerator project. The larger part of the chapter consists of an introduction to the main kinds of particle detectors used along the LHC. The focus is mainly put on the front-end electronics of the detectors and the signals created in this electronic. At the end of the chapter a short summery presents the most important parameters of the most common output signal found in the different particle detector front-ends and for which the chosen compression method will be optimized.

Chapter 3 gives an introduction into the data compression theory. The most common lossless and lossy data compression methods are presented briefly. The different methods are explained by using small examples. At the end of each presented method a brief discussion of its suitability for the implementation in the detector front-end is given taking into account the signal parameters discussed at the end of chapter 2. A deeper investigation of the most promising methods is then carried out in chapter 4.

In chapter 4 the most promising data compression methods are analyzed in terms of compression efficiency and complexity of their realization in hardware. Different methods are realized in Matlab and tested using measured data from the ALICE TPC. A method is found, which is considered the best suitable for an implementation in the detector front-end electronics by resulting a good compression efficiency without introducing significant information loss.

The implementation of this best suitable compression method is presented in chapter 5. This chapter summarizes the implementation work as well as the properties of the implementation. The RTL Verilog model of the algorithm is presented. The Verilog model is optimized for an implementation of the data compression in the readout control unit of the TPC in ALICE.

The results from the implemented compression method are presented in chapter 6. The simulation of the RTL model with measured data from the ALICE TPC is presented and the simulation results are compared with the theoretical results obtained in chapter 4. The RTL Verilog model is tested for a FPGA implementation by using a development board with a Virtex-4 FPGA. The synthesis results and the hardware test results are presented. A first investigation of a possible implementation in a full custom front-end ASIC in deep submicron 130 nm CMOS IBM technology is summarized. As well a Matlab model of a Decoder for the corresponding data compression algorithm is presented and used to decode the output data of the implementation and to compare it with the original data.

Chapter 7 finalizes the analysis of the data compression method with a research summary and an outlook for possible field of applications of the data compression implementation.

2. High energy physics: Properties of signals from detectors

The topic of this dissertation is the study, evaluation and development of a compression method that can be implemented in the front-end electronics of particle detectors for the reduction of data representing signal waveforms from nuclear sensors. To develop an efficient compression in the hardware of detectors it is crucial that the compression method is optimized for the specific data generated by detector sensors and electronics. An introduction in the high energy physics and the detector concepts of the LHC experiments at CERN are presented to give an idea of the origin and properties of the data to be compressed.

High energy physics is used to study the elementary constituents of matter and the fundamental forces through which the elementary constituents interact among them [4]. The underlying branch of physics studying the elementary particles is called elementary particle physics. Since many elementary particles are not stable and do not exist under normal conditions in nature, particle physics experiments use energetic collisions of other particles to create and study them. The objective of these studies can be very diverse, but in general are focused to understand the fundamental forces and constituents of matter.

CERN, the European Organization for Nuclear Research, is the largest laboratory in the world currently focusing on this type of research. It houses the world's largest particle accelerator called LHC that is used to produce high energy collisions. The LHC consists of a 27 km long ring, which accelerates protons or heavy ions near to the speed of light and collides them. The LHC produces the highest energy proton-proton collisions ever made by humans and it is expected that it will allow new important discoveries in physics [5]. Physicists hope to prove with the LHC the existence of the Higgs boson, which is theorized to be responsible for giving mass to matter. Four huge experiments with complex detectors are built around collision points of the LHC with the objective of performing measurements on the short living particles created by the collisions on the two highly energetic beams.

In this section a short introduction to particle physics will be given. CERN and the LHC will be presented briefly, as well as the different types of particle detectors used along the LHC. The focus is put on the description of the front-end electronics of these particle detectors and the electric signals present in this front-end electronics. In the following, the shapes and properties of these electric signals are important for designing a data compression algorithm, which can be efficiently implemented in the front-end electronics of the detectors.

2.1. High energy physics and CERN

The theoretical understanding of particle physics is currently based on the so called "Standard Model" which provides the most widely accepted classification of the known elementary particles and of their interactions. In the Standard Model 24 elementary particles are distinguished. The Standard Model categorizes them in two different kinds of particles called the "Leptons" and the "Hadrons". Hadrons (from Greek 'adros' meaning 'bulky') are particles composed of quarks. The protons and neutrons building atomic nuclei for example contain three quarks. In total there are six quarks distinguished in the Standard Model called Up, Down, Charm, Strange, Top and Bottom. Leptons (from Greek 'leptos' meaning 'thin') are so called elementary particles that are not made

out of quarks as for example electrons, positrons and muons. [6]

In addition to the elementary particles the Standard Model provides a theoretical background for the fundamental forces which act on the particles. In total four forces are described in the Standard Model; the electromagnetic force, the strong force, the weak force and the gravitation.

The electromagnetic force is together with the gravitation the most well known of the four and it holds the electrons to the nuclei to form atoms and binds atoms to molecules. The force acts for example between the atomic nuclei and the electrons. The carrier of the force is the photon [6].

The gravitation is an attractive force, which acts between all objects with mass. It binds matter in planets and stars, stars in galaxies and us to the earth. The carrier of the force is the graviton. [6]

The strong force binds the quarks together to form protons and neutrons (and other hadrons). This force also binds the protons and neutrons in the atomic nuclei, where it overcomes the enormous electric repulsion between the equally charged protons. The carrier of this force is the gluon. [6]

The weak force underlies natural radioactivity and is essential for the reactions occurring for example during nuclear fusion in the center of stars like our Sun. The carriers are bosons (W^-, W^+, Z^0). [6]

The Standard Model describes all our current understanding of the fundamental particles and forces, but some phenomena recently observed in the universe remain unexplained by this theory. The Standard Model cannot account for dark matter and dark energy, which is supposed to be present in the universe but is not visible to us. The Standard Model also does not explain the origin of mass, why some particles have mass and others have no mass. The search for the Higgs boson and the study of dark matter, dark energy, as well as the investigation of the properties of quarks and gluons are among the main objectives for the LHC experiment at CERN.

2.1.1. CERN History and numbers

CERN is located at the border between Switzerland and France near Geneva. CERN provides facilities (accelerators) to perform fundamental physics research for studying the basic constituents of matter. At the end of the Second World War, a visionary idea of a handful of scientists was to create a European atomic physics laboratory to unify the forces of the national institutes and to perform research in a large scale. Raoul Dautry, Pierre Auger and Lew Kowarski in France, Edoardo Amaldi in Italy and Niels Bohr in Denmark were among these pioneers. At the beginning, 12 Member States ratified the founding of CERN: Belgium, Denmark, France, the Federal Republic of Germany, Greece, Italy, the Netherlands, Norway, Sweden, Switzerland, the United Kingdom and Yugoslavia. In 1954, the European Organization for Nuclear Research, CERN, came officially into being. The CERN's first particle accelerator the 600 MeV SynchroCyclotron (SC), was built in 1957, and it provided particle beams for first particle and nuclear physics experiments. [7]

Since then the laboratory, has grown to be the largest particle physics laboratory in the world and one of the world's largest and most respected centers for scientific research [7]. At present, CERN has 20 member states, and around 6500 scientists are working at CERN from over 500 Universities in more than 80 countries [8].

2.1.2. The LHC

The Large Hadron Collider is a 27 km long accelerator ring, which is placed in an underground tunnel. The tunnel with a mean depth of 100 m was already built for the predecessor accelerator LEP (Large Electron-Positron). The LEP was dismantled in 2001 to give space to the LHC. Superconducting dipole magnets bend the accelerated particles in a circular track. Along the 27 km circumference 1232 superconducting dipole magnets are placed, which are cooled down to 1.9 K and provide a magnetic field of up to 8 T. Two beam pipes are surrounded by the magnets in which

two proton beams are traveling in a high vacuum in opposite directions. At four points along the accelerator ring, the particle beams can be brought to collide. Detectors are placed around these collision points to detect the secondary particles generated by the collisions. In the two beam pipes, protons as well as heavy ions can be accelerated to nearly the speed of light (99.9999991%c). The LHC is designed to accelerate protons to a maximum energy of 7 TeV per beam, while the heavy lead ions can reach an energy of 575 TeV. The collision energy is given by the sum of the energies of the two colliding beams. The collisions of protons with 14 TeV will recreate the conditions which are supposed to have happened a few moments ($\approx 10^{-12}$ s) after the Big Bang. In this way, physicists hope to discover how the Universe evolved and probably solve some mysteries, which are not yet fully described by the Standard Model. [6]

There are six experiments placed along the LHC, which house the different particle detectors. They are called: ALICE, ATLAS, CMS, LHCb, LHCf, TOTEM.

The two experiments ATLAS (A Toroidal LHC ApparatuS) and CMS (Compact Muon Solenoid) contain general-purpose detectors to cover the widest range of physics at LHC, mainly targeting in the discovery of the Higgs boson, searching for supersymmetry and extra dimensions [6].

ALICE (A Large Ion Collider Experiment) is specialized in studying the quark-gluon plasma produced in the heavy ion collisions to understand better the properties of quarks [6].

The main objective of the LHCb (Large Hadron Collider beauty) experiment is to study the slight asymmetry between matter and antimatter present in interactions of particles containing beauty quarks [6]. This could provide an answer to the question why the Universe is made out of the matter we observe and what happened with the antimatter as well created in the Big Bang.

The LHCf (Large Hadron Collider forward) experiments will be used to measure particles produced very close to the direction of the beams to test models which are used to estimate the primary energy of ultra high-energy cosmic rays.

The TOTEM (TOTal cross section, Elastic scattering and diffraction dissociation Measurement) experiment targets to measure the effective size of the protons in the LHC.

The main kinds of particle detectors used in the different experiments will be described in the following.

2.2. Particle detectors

The different experiments along the LHC use similar detector schemes even though they focus on different physics measurements. In this chapter, the detector structure of the ALICE experiment is given as an example for the composition of the experiments at LHC and is shown in figure 2.1. The main detectors of ALICE are presented together with some additional detector concepts from the other three large experiments (ATLAS, CMS, LHCb).

Most of the detectors of ALICE are installed inside a large solenoid magnet, which provides a high magnetic field. Charged particles emerging the collisions and traversing the detectors are deviated by the magnetic field forcing them on a curved path. The curvature of the particle trajectory is used to determine the charge polarity and the momentum (product of mass and velocity) of the particles, which are important for the particle identification. Different detection methods are used to track and identify different kinds of particles. In the following, some of these detector methods are described briefly and the signals which are produced in the detector front-end electronics, are discussed.

2.2.1. Silicon based detectors

Silicon based detectors, e.g. the silicon pixel detector and the silicon microstrip detector, are used in all experiments as the innermost detectors, which are closest to the collision point and surround the beam pipe. The inner detector of the ALICE experiment ITS (Inner Tracking System) contains

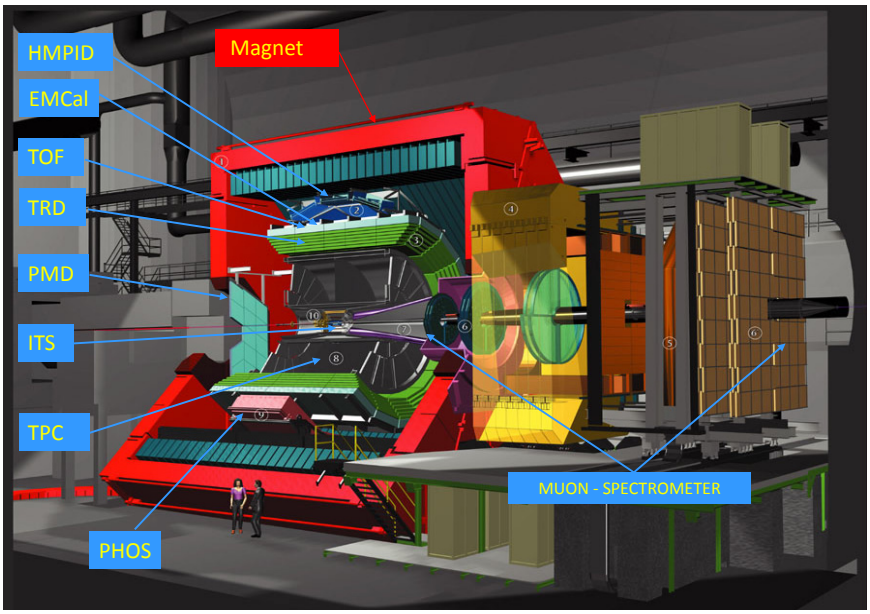


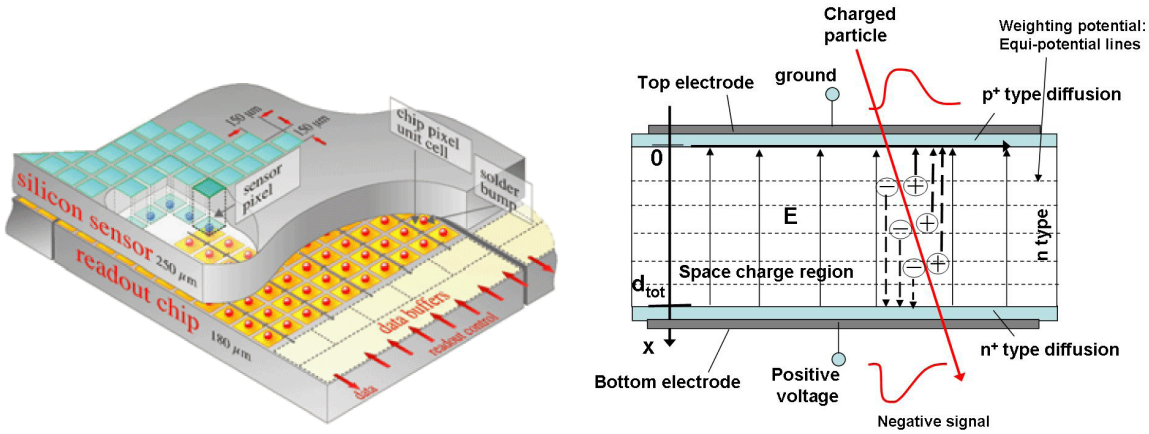
Figure 2.1.: The ALICE experiment with the detectors: HMPID (High Momentum Particle Identification detector), EMCal (ElectroMagnetic Calorimeter), TOF (Time-Of-Flight detector), TRD (Transition Radiation Detector), PMD (Photon Multiplicity Detector), ITS (Inner Tracking System), TPC (Time Projection Chamber), PHOS (PHOton Spectrometer), Muon detectors

in addition a silicon drift detector. The advantage of silicon detectors is that they provide a high granularity due to small sizes of the detecting pixels. Silicon detectors produce short signals, which allows them to handle very high particle rates [9]. This makes the silicon detectors the best choice for the innermost detection as close as possible to the collision point where a high number of particles emerge from the collisions and a high precision is required to track them. Several layers of these detectors are used to track the paths of charged particles covering a high range of particle energies. Therefore, in combination with the detectors farther away from the collision point an exact reconstruction of the particle trajectories and their origin can be achieved. Interesting events can be identified by detecting short-lived particles, which after their creation in the primary collision decay already inside the beam pipe in secondary particles. To identify these short-lived particles it is crucial that the first detectors detect the secondary particles precisely to reconstruct their trajectories and to find the common point of origin displaced from the primary collision point to determine the primary particle [9]. The silicon detectors near the collision point are subject the high radiation generated by the secondary particles. This radiation damages silicon sensors and components and reduces their lifetime. The relatively high cost of silicon detectors makes them not feasible to cover large detection areas forcing physicists to use other detectors at large external radii.

Silicon pixel detector

Silicon pixel detectors provide a detection of charged particles in two dimensions with a high resolution in space and time. Each pixel has a size of a few tenths or hundreds of micrometers (microns) per side. A pixel corresponds to a reverse biased diode and a pixel detector consists of a matrix of several such pixels as shown in figure 2.2(a). The working principle of a pixel is shown in figure 2.2(b). The reverse biased p-n junction forming the diode creates a large depletion region (space charge region). A charged particle passes through this depletion region liberates electrons from the n-doped silicon creating electron-hole pairs. Each silicon pixel contains a top and bottom electrode supplied with a high reverse biasing voltage (in the order of 100 V), which

forms an electric field inside the depletion region. The created charges drift in the electric field towards the electrodes and their movement induces a signal in the electrodes. The electrodes are then connected to a silicon front-end ASIC chip, which reads out the induced signal. The connection between the pixel matrix structure and the front-end chip can be realized for example by bump bounding techniques as shown in figure 2.2(a). The width of the silicon pixel structure



(a) Hybrid Pixelelement [10] (b) One silicon pixel element [11]

Figure 2.2.: Principle and composition of a silicon pixel element

is a few 100 µm. A pixel, which is traversed by a charged particle, provides the information of its position in two dimensions, together with the time information when the particle hit occurred (time stamp). This information is used to reconstruct the particle trajectory by installing several layers of pixel detectors around the beam pipe providing a few space points of each trajectory. In this way, a three dimensional reconstruction of the particle tracks emerging the collision point is obtained. In addition, the number of generated electron-hole pairs is proportional to the energy loss of the particle, which can be determined by integrating the induced current in the electrodes. This information is used to identify the detected particle. An example of a particle track through the ATLAS inner tracker structure is shown as an example in figure 2.3. The signal which is induced in the electrode of a pixel element is composed by a fast rise time created by the fast moving electrons, which are present in the depletion region for less than 10 ns before they are collected at the electrode. The decay time of the signal is dominated by the slower motion of the positive charged holes, which drift to the opposite electrode having a collection time of around 25 ns. The first stage of the front-end electronics is an analogue current integrator, which integrates the input signal to translate the induced signal charge into a voltage level. The signal amplitude can be small which requires also an amplification operation of the integrator. Therefore, in high-energy physics this step is called sometimes pre-amplification. The second stage is a differentiator (high-pass filter) which brings the integrated voltage level back to ground (defines the decay-time of the output signal). To increase the signal-to-noise ratio a third stage can be added consisting of low-pass filters to limit the output signal bandwidth. This bandwidth limitation will cut the noise outside the signal spectrum but it also increases the signal duration, which can be problematic for pile-up effects regarding later arriving signals. The last two stages are named often as signal shaping stage. The output of the shaper is an analogue signal that will have in most cases a semi-Gaussian shape. An example of the ALICE pixel detector front-end electronics is shown in figure 2.4 together with the output signal after the analogue preamplifier-shaper. This front-end electronic can cope with both positive and negative induced charges depending on which electrode it is connected. [9]

Upon traversal of a pixel by a charged particle, the analogue output signal of the shaper can then be processed in different ways. A comparator can be used to detect when the output signal

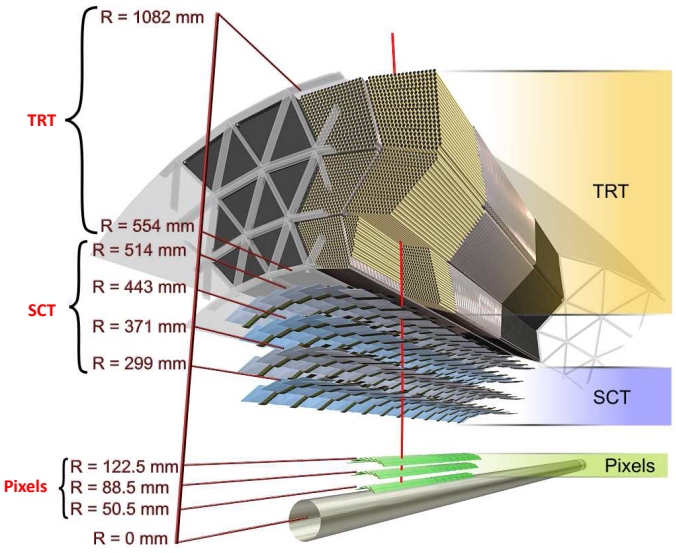
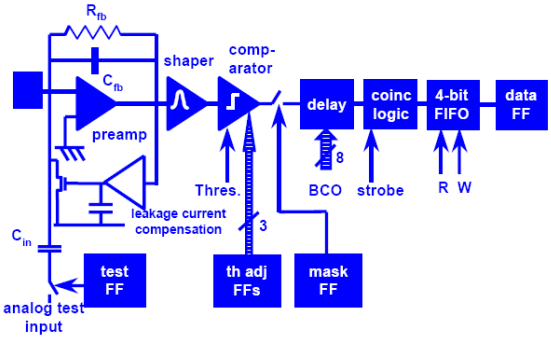


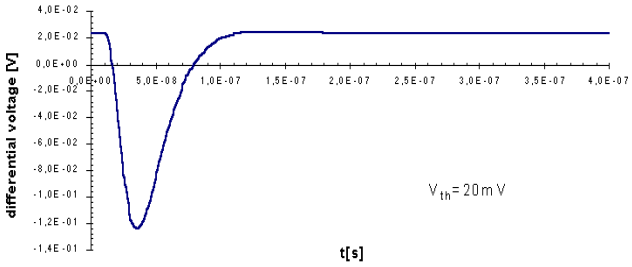
Figure 2.3.: The figure shows the inner detector layers of the ATLAS experiment and a horizontal charged particle track exiting the beryllium beam-pipe, in which the accelerated particles travel and collide. The several layers of pixel sensors, silicon microstrip sensors (SCT) and the transition radiation tracker (TRT), which surrounding the beam pipe at different radii provide the space points for the particle tracks passing through them [12]

rises above a defined threshold. The comparator output is typically stored in a memory waiting for the arrival of an external trigger. To identify unambiguously the event in time, a time stamp when this hit occurred is stored as well. This gives the information, which pixels are hit at which time from the particles and can be used for tracking their path. If in addition the amplitude of the signal should be retained as well, a possibility is to digitize the shaper output signal using an ADC. In order to obtain the amplitude, the signal can be reconstructed offline using a fitting of the samples. The amount of charge induced in a pixel element can also be measured using the so called Time Over Threshold (TOT) method. This consist in measuring the time during which a signal remains higher than a predefined threshold and gives an approximate relation to the amplitude of the signal.

Not all signals caused by a particle detection of all collisions can be read out because that would lead to a huge amount of data coming from the high number of channels (in ALICE $\approx$ 10 million pixels) and the high collision rate of the LHC of 40 MHz. Therefore, a trigger system decides if the data from a collision event should be retrieved or rejected.



(a) ALICE pixel readout mixed signal chip (ALICE1) [13]



(b) Output signal of the shaper [14]

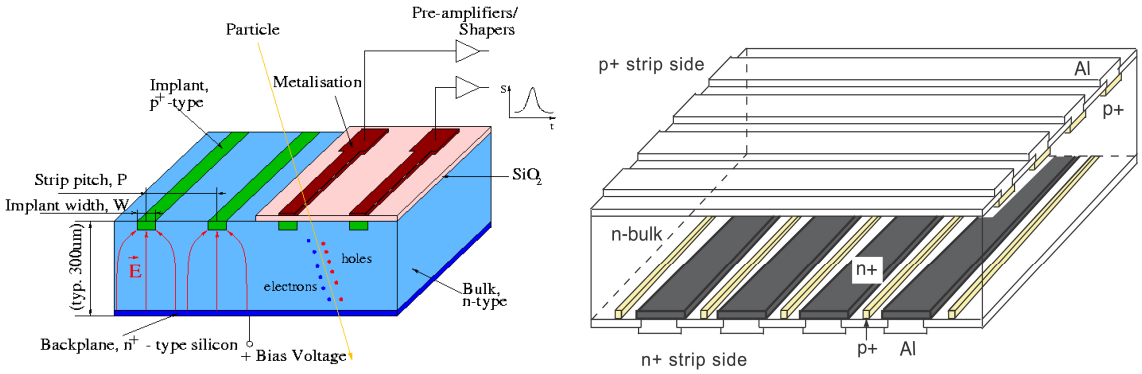
Figure 2.4.: Block diagram of the front-end chip for the ALICE pixel detector and the LHCb RICH detector. The signal after the shaper is shown to the right.

In the ALICE experiment, several detectors e.g. Transition Radiation Detector (TRD), silicon detectors etc. can provide information to the trigger system. For example if the TRD detects an interesting event, a level 0 trigger (L0) is created. A pre-trigger activates the TRD after each collision (<900 ns) to collect the data for the L0-trigger. The L0-trigger is sent out after around 1.2 µs. The TRD performs some quick analysis on the detected signals to decide if the event is interesting and fulfills some criteria. In the case that the event could be interesting a level 1 (L1) trigger is generated in 6.5 µs. Some further online data analysis is performed to decide if the collected data from the event are read out or discarded. A level 2 (L2) trigger arrives at the detectors around 100 µs after the L1-trigger deciding to read out the data or throw them away. The detectors (e.g. TPC) use the three level of triggers to control their signal processing and readout of data.

The ALICE pixel detector front-end contains a discriminator and a digital delay line to delay the input signal until the L1-trigger arrives. The discriminator (comparator) digitizes the input signal sending out a logic high signal if the signal is above the threshold. Then the pixel hit data proceed through the digital delay line for 6.5 µs. If the L1-trigger arrives within this delay, the pixel-hit data are stored in a multi-event buffer inside the front-end electronics. The data are rejected if the L1-trigger does not arrive. The multi-event buffer (4-bit FIFO) is capable to store up to four hit events. After arrival of a L2-trigger, the multi-event buffers of the pixel chips are read out starting from the oldest stored event.

Silicon microstrip detector

The silicon microstrip detector or simply strip detector is similar to the pixel detector. In the experiments, the strip detectors surround the beam pipe at a larger radius from the collision point than the pixel detectors and cover a larger area. The charged particle density is smaller than for the pixel detectors (for ALICE <1 particle/cm^2) but the larger detection area needs a different concept to limit the number of readout channels. The main difference between silicon pixel detectors and silicon strip detectors is that a strip provides a high spatial resolution in one direction whereas a pixel gives a high resolution in two dimensions. Figure 2.5(a) shows the principle of a silicon strip detector. The free charges created in the depletion region by a traversing charged particle,



(a) Principle of a silicon strip detector [15]

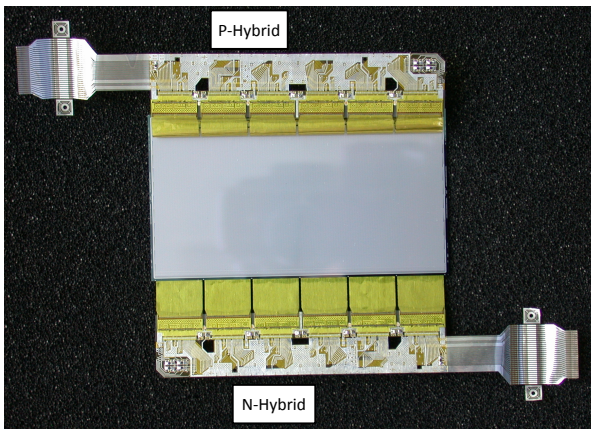
(b) Double sided silicon strip detector with orthogonal strips [16]

Figure 2.5.: Silicon strip detector element single sided for a one dimensional space resolution and double sided for two dimensional space resolution

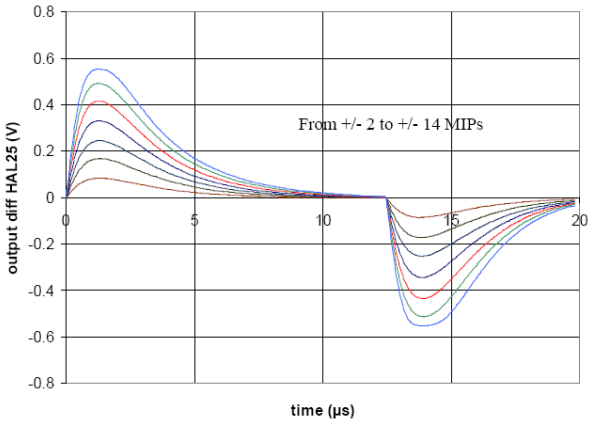
are drifting in the electric field towards the electrodes. The top electrode has a strip form. The induced signal in the strip electrodes is detected by the front-end electronics, which has similar requirements as for the pixel detectors. A front-end electronics channel contains a pre-amplifier and a shaper. The effect of charge sharing between neighboring strips caused by a particle traversing

the sensor at an angle can be used to obtain a space resolution better than the strip pitch. To obtain a space resolution in direction of the microstrips a double-sided strip detector (so called stereo-module) can be constructed with two microstrip elements glued together back to back resulting in two strip planes arranged at an angle. The principle of double sided strip detector elements is shown in figure 2.5(b).

In the ALICE experiment around 1700 double sided silicon strip modules are installed leading to 2.5 million readout channels. A picture of such a strip module is shown in figure 2.6(a).



(a) ALICE Silicon Strip Module



(b) Alice silicon strip detector front-end electronics, pre-amplifier-shaper output signals of positive and negative charge induction [17]

Figure 2.6.: ALICE silicon strip detector module and analogue output signal of the front-end amplifier-shaper circuit

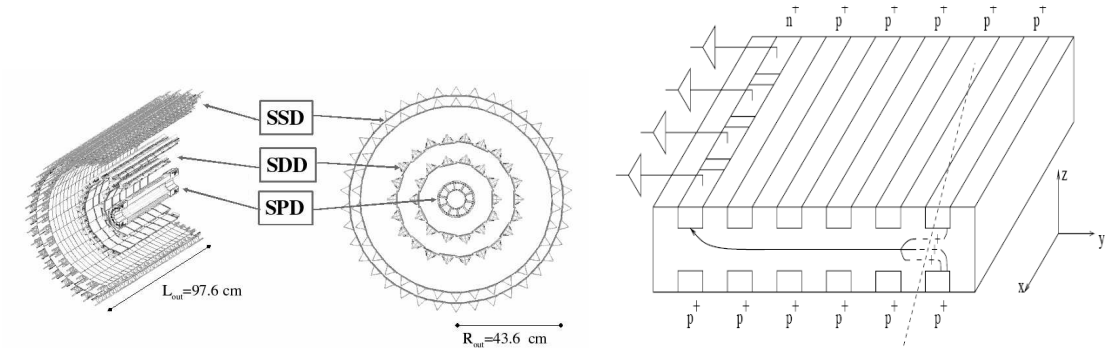
The front-end electronics consist in most cases of pre-amplifier, shaper, discriminator, delay elements and output buffers similar to the pixel detector. The output signal of the shaper is again a semi-Gaussian waveform. This semi-Gaussian output signals are shown in 2.6(b) with positive and negative polarity depending on the strip plane of the double sided strip element.

Silicon drift detector

In the ALICE inner tracking system, a third type of silicon detector is installed containing silicon drift sensors. The silicon drift detector is located between the silicon pixel detector and the silicon strip detector as sketched in figure 2.7(a). The principle of a silicon drift detector is shown in figure 2.7(b).

A silicon drift detector is used to detect charged particles passing through the sensor and determine the position in two dimensions. The charged particle liberates electrons in the depletion region (high resistive n-type wafer). The p+ doped strips on both sides create an electrostatic field, which forces the free electrons in the center of the depletion region. An additional electric field is applied between a cathode on one side of the detector element and an anode region at the opposite side. This electric field forces the liberated electrons in the middle of the depletion region to drift to the anode region. The anode region contains several separated electrodes to improve space resolution in x direction. The electrons drift with constant drift velocity from their creation point to the anodes and the drift time is used to determine the position of the particle trajectory in y direction. The front-end electronics is connected to the anodes and performs the amplification and shaping of the signal.

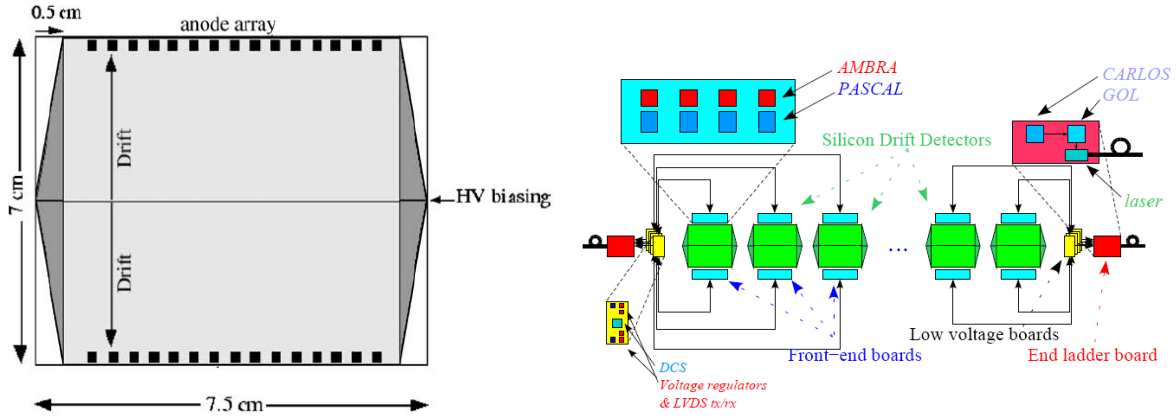
The architecture of an ALICE silicon drift detector module is shown in figure 2.8(a) and consists of two sensor elements separated by a high voltage cathode. Each element has 256 anodes connected to the front-end electronics. The ALICE silicon drift detector has in total around 133 000



(a) ALICE inner tracking system: Silicon Pixel Detector (SPD), Silicon Drift Detector (SDD), Silicon Strip Detector (SSD) [18]

(b) Silicon drift detector element [19]

Figure 2.7.: ALICE silicon drift detector position in the inner tracking system and working principle of a detector element



(a) Silicon Drift Detector Module

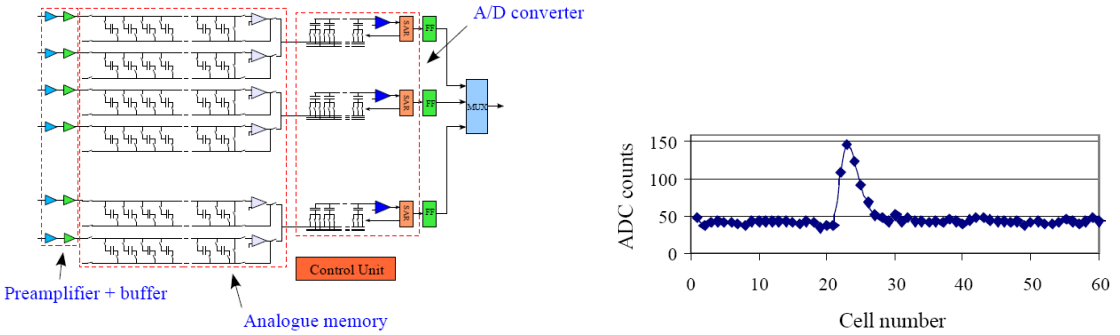
(b) Silicon Drift detector Readout System

Figure 2.8.: ALICE silicon drift detector module and readout system containing PASCAL, ALBRA, CARLOS

channels (anodes). The front-end electronics is composed of three full custom chips (ASIC) which perform the amplification, shaping, digitization, compression and buffering of the input signals from each anode. The front-end electronics is shown in figure 2.8(b). The first chip called PASCAL contains 64 channels, each one containing an analogue amplifier and shaper producing a semi-Gaussian waveform. The analogue signal is then stored in an analogue memory composed of 256 cells to account for the L0-trigger latency (1.2 µs) and the maximum drift time (5 µs). The analogue memory is frozen upon arrival of a L0-trigger accept signal and the digitization of the analogue value is started. A successive approximation analog to digital converter (SAR-ADC) with 10 bit resolution and a clock of 40 MHz digitizes the signal stored in the analogue memory. The structure of the PASCAL chip is shown in figure 2.9(a).

The digital signal is then send to the next chip called AMBRA, which performs a baseline correction and a data reduction by a non-linear compression of the 10-bit samples from PASCAL to 8-bit words. The 8-bit words are stored in one of sixteen on board RAM memories. An output signal and the digitized sample values are shown in figure 2.9(b). [20]

When the full content of the analog memory is read out and digitized the AMBRA chip starts transferring the data from the RAM memory to a third chip called CARLOS. During the slow speed transfer from AMBRA to CARLOS, the PASCAL chip can continue the data acquisition and new digitized data are stored in another RAM memory inside the AMBRA chip. If the L1-trigger does



(a) Silicon Drift detector front-end chip PASCAL (b) Silicon Drift detector digitized output signal

Figure 2.9.: ALICE silicon drift detector PASCAL ASIC and its output signal

not arrive during the digitization phase (around $320\mu s$) the process is aborted and the memory in the AMBRA chip is cleared. The data are discarded as well if a L2-trigger reject signal arrives. The CARLOS chip performs a further data compression by performing a zero suppression before the data are transferred outside the experimental area.

2.2.2. Calorimeter

Calorimeters are devices used in particle physics experiments to measure the energy of particles created in the collisions. When traversing heavy materials, charged particles are progressively slowed down by electromagnetic interactions with the atoms of the absorbing materials. In this process, electrons and photons are mainly created (sometime nuclear interactions are also generated). The photons in turn are mostly converted into electron-hole pairs. A measurement of the amount of free electrons provides an estimate of the energy belonging to the primary particle. The basic objective of measuring energy of particles in calorimeters is simply achieved by interspersing layers of absorbers and layers of detecting materials in a way as to measure the energy of all secondary particles produced. The free electrons induce a signal in electrodes, whereas light sensing devices detect the photons if these are produced with wavelength roughly close to visible light. The fact that calorimeters cause the detected particles to stop requires that they be positioned in the outer layers of the experiments. ATLAS and CMS use calorimeters as their main detecting devices, but also ALICE and LHCb make use of calorimeters. There are mainly two different types of calorimeters used in the experiments: the electromagnetic calorimeter and the hadronic calorimeter. These two types are described in the following.

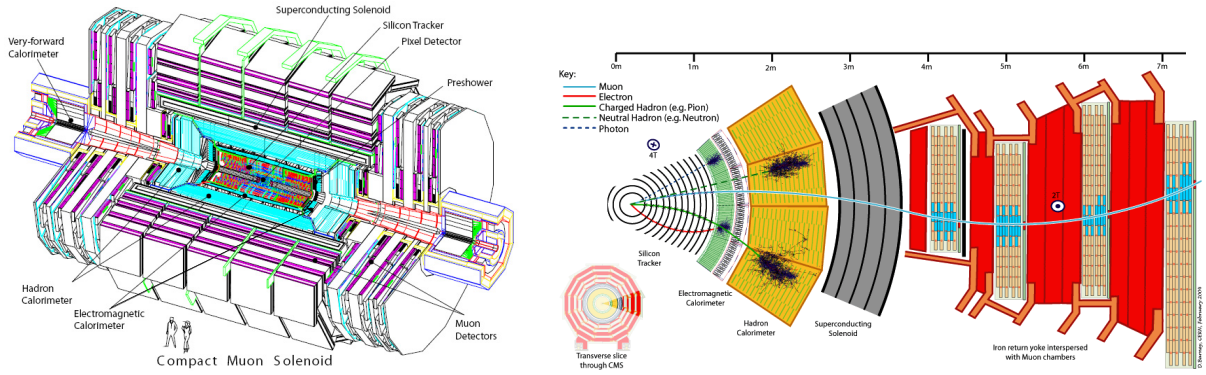
Electromagnetic Calorimeter

The electromagnetic calorimeter (ECAL) is used to determine the energy of electrons and photons created by the collision. These are interesting particles for the discovery of the Higgs bosons. To detect the energy of the high energetic electrons or photons the electromagnetic calorimeter consists of alternating layers of absorber material and active material such as scintillators or liquids. When the electrons and photons are passing through the absorber material, they interact with the atoms of the material either by producing electron-positron pairs or through bremsstrahlung producing secondary photons. This leads to a particle shower in the ECAL layers of low-energy electrons, positrons and photons. The primary high-energy electron or photon passes through several layers of the ECAL until eventually it has lost all its energy and stops as shown in figure 2.10(b).

The ECAL of the ATLAS experiment consists of accordion shaped absorber plates made of lead and stainless steel. The active material between the absorber plates is liquid argon [12]. A

copper grid is placed in the liquid argon, which is supplied with high voltage to create an electric field (drift region). The shower of secondary low-energy particles ionizes the liquid argon and the movement of the liberated electrons and ions in the drift region, induce a signal in the grid that then is read out.

The CMS electromagnetic calorimeter uses lead tungstate crystals. A small amount of oxygen is added inside the crystalline structure of the lead tungstate, which makes the crystals highly transparent. A high-energy electron or photon interacts with the heavy nuclei of the crystal and produces a shower of secondary low-energy electrons, positrons and photons. The energy of passing by secondary particles excites the electrons in the atoms of the crystal that when relaxing emit photons. These photons are detected by photo detector devices, which transform the light into an electrical signal [21]. The position of the ECAL inside the CMS barrel is shown in figure 2.10(a) and is located immediately after the inner silicon tracker and before the hadronic calorimeter. ECAL end-cap wheels are placed on both sides of the silicon tracker. The PHOS detector and the EMCal in the ALICE experiment use also lead tungstate scintillators and their position can be seen in figure 2.1.



(a) CMS structure (b) CMS slice

Figure 2.10.: CMS experiment detector structure and radial slice showing the particle path of different particles

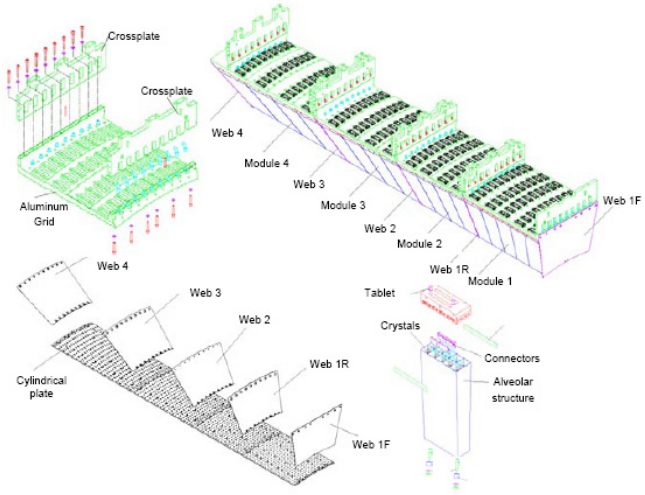
To distinguish the high-energy electrons from the high-energy photons the track of the particle shower in the ECAL is followed back and combined with the tracks detected inside the inner tracker. If the silicon detectors have detected a particle track, which matches with the track of the particle shower, the detected particle is a high-energy electron, because high-energy photons have no charge and leave no track in the silicon detectors.

The composition of an ECAL module is shown in figure 2.11 together with an image of the modules front view. The image in figure 2.12(a) shows half an end-cap ECAL with the scintillator crystals.

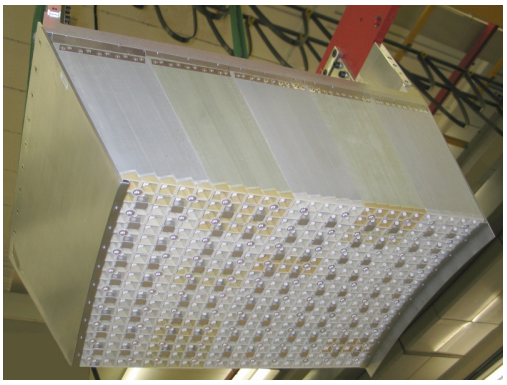
The detection of the light inside the barrel ECAL is performed by avalanche photo diodes (APD), while for the end-cap ECAL vacuum photo triodes (VPT) are used. Two crystals one with the APD and the other with the VPT are shown in figure 2.12(b). The photo detectors APD and VPT transform the scintillator light into electric signals, which are then processed in the front-end electronics.

A front-end electronics channel of the CMS ECAL consists of a pre-amplifier and shaper, three additional amplifiers with different gain, three ADC's, a digital pipeline with 256 words, an event buffer, a trigger generator and a gigabit optical link. The different blocks are shown in figure 2.13.

The signal from the APD or VPT is first amplified and shaped resulting in a semi-Gaussian shape. An additional set of three single-ended to differential amplifiers with gains 1, 6 and 12 are used to best amplify the signal to the full scale of the ADC's. The shaping and amplification units



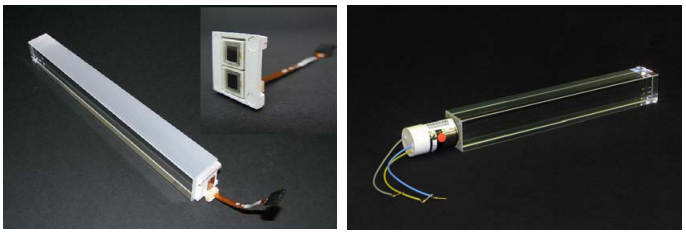
(a) Module layout of CMS barrel ECAL



(b) Picture of the front view of the barrel ECAL of CMS

Figure 2.11.: The barrel electromagnetic calorimeter of the CMS experiment containing the lead tungstate crystals [22]

(a) Picture of half of the end-cap ECAL in the CMS



(b) Two lead tungstate crystals: left with APD, right with VPT

Figure 2.12.: The end-cap electromagnetic calorimeter of the CMS experiment and two lead tungstate crystals with photo elements [22]

are combined in an analogue 0.25 µm ASIC called MGPA (Multi Gain Pre-Amplifier). The three amplifier outputs of the MGPA are differentially connected to three ADC's with 12-bit resolution and 40 MHz sampling frequency. The highest not saturated output signal of the three ADC's is detected and the 12-bit words are sent out combined with two bits to indicate the used gain. A LVDS (Low Voltage Differential Signal) chip adapts the differential outputs of the ADC's to single ended signals. These blocks are realized in a very front-end card (VFE) connected to 5 crystals. The single ended signals of the VFE cards are transferred to an additional front-end card (FE) where the digital data are stored in a 256-word deep dual port memory to buffer them for the level-1 trigger latency. A chip called FENIX performs a summation of the energy of the 5 crystals and sends the information to the off-detector trigger concentration card, which controls the L1-trigger creation. If a L1-trigger indicates an interesting event, the data stored in the 256-word buffer are transferred to a multi-event buffer before they are readout via a gigabit optical hybrid link upon a L2-trigger accept signal.

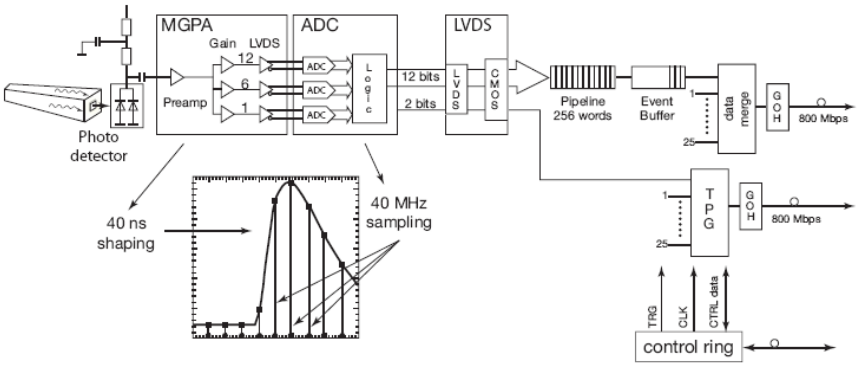


Figure 2.13.: Block diagram of the ECAL front-end electronics in the CMS experiment [22]

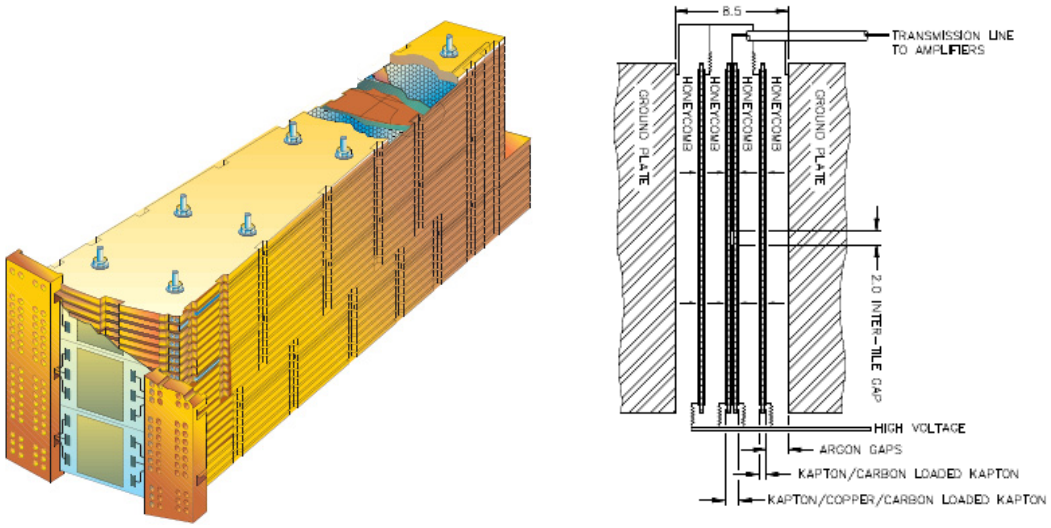
Hadronic Calorimeter

The hadronic calorimeters (HCAL) are used in the experiments to detect hadrons which are particles made out of quarks e.g. protons, neutrons, kaons and pions. The hadronic calorimeters are positioned after the electromagnetic calorimeters (seen from the interaction point) as shown in figure 2.10(a) for the CMS detector. After the high-energy electrons and photons are stopped in the electromagnetic calorimeters, the heavier hadrons are stopped in the hadronic calorimeter (HCAL). The concept and structure of the hadronic calorimeters are similar to the ECAL. An absorber material slows down the hadron particles from the collision until they stop. When the hadrons pass through several layers of absorber material, they lose energy by elastic and inelastic scattering between the particle and the nucleons of the calorimeter material until they release their remaining energy by ionization or nuclear absorption. The loss of energy of the primary hadron particle leads to the production of secondary particles that create a particle shower in the HCAL.

In the CMS experiment, the hadronic calorimeter is composed of alternating layers of absorber material (brass or steal) and layers of active material so called scintillators (plastic). As the particle shower produced by the primary hadron develops through the HCAL, the particles pass through the scintillator material and cause this material to emit blue-violet light [23]. This blue-violet light is absorbed by wavelength-shifting fibers installed in the scintillators and transferred to green light. The green light can be transported with clear optical cables to readout boxes and the light of a full sector of the hadronic calorimeter is summed up optically by bundling the cables. This allows detecting the total light produced by one particle shower, which corresponds to the energy of the primary hadron. In the readout box photo-sensing devices, e.g. photo-multiplier tubes or hybrid photodiodes are transferring the incoming photons in electrical signals, which then can be processed by the front-end electronics. The CMS front-end electronics of the hadronic calorimeters, which are based on scintillators is composed by a charge-integrating ADC chip called QIE (Charge-Integrator and Encoder). The analogue signal from the hybrid photodiode (HPD) is integrated by four capacitors that are connected in turn to the input of the QIE each one having a time constant of 25 ns. The integrated charge from the capacitors is converted then to a digital value of 7-bit resolution with a non-linear scale. The digital values from the QIE outputs are combined with some monitoring information to 32-bit words and send with 40 MHz via a gigabit optical link to the counting room.

Steel absorbers and plastic scintillators as active medium are installed as well in the central hadronic calorimeter of the ATLAS experiment. Wavelength-shifting fibers absorb the light produced in the scintillators and shift it to the green spectrum (longer wavelength). Photomultiplier tubes detect and convert the green light into electrical signals. The Hadronic End-cap Calorimeters (HEC) and the forward calorimeters of ATLAS are based on copper absorber material and liquid argon as active medium between the copper plates similar to the ECAL. The particle shower ionizes

the liquid argon. Electrode plates are inserted in the liquid argon and supplied with high voltages to form an electric field and a drift region. The electrons and ions drift towards the electrodes and their movement induces a signal in the electrodes, which is then readout from the front-end electronics connected to the grounded electrode. A schematic view of a hadronic end-cap calorimeter module and the readout structure are shown in figure 2.14.

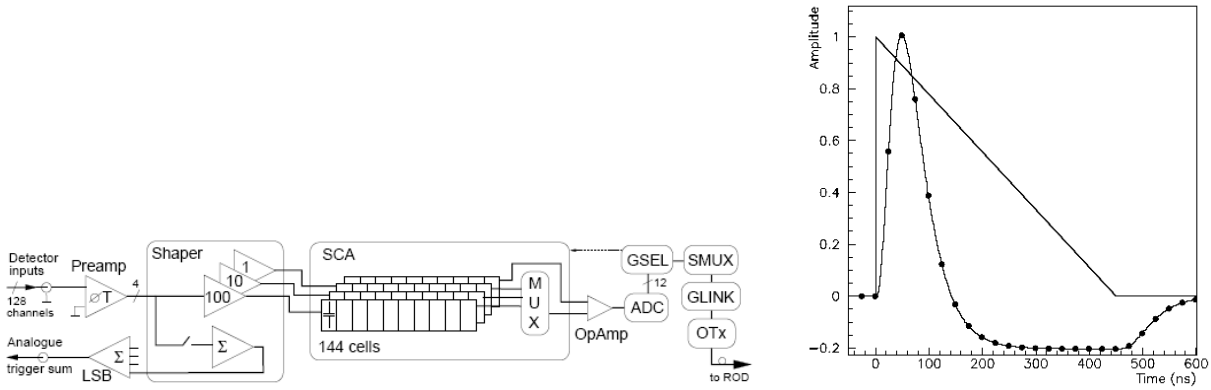


(a) Schematic view of the HEC module of ATLAS

(b) Schematic of the readout structure in the liquid argon gap of the HEC module

Figure 2.14.: ATLAS hadronic end-cap calorimeter (HEC) module [12]

The front-end electronic of the ATLAS hadronic calorimeters based on liquid argon is identical to the front-end electronics used for the ECALs of ATLAS. This front-end electronic is composed of a pre-amplifier and shaper with three different linear gain scales, analogue memory (SCA-switched capacitor array), and 12-bit ADCs to digitize the buffered analogue signals if a L1-trigger arrives. The digital values are then sent via optical links to the counting room. A schematic of the front-end electronic for liquid argon calorimeters is shown in figure 2.15, together with the shaped output signal of the front-end boards.



(a) Block diagram of the front-end board architecture

(b) Triangular input signal and shaped output signal of the front-end board

Figure 2.15.: Front-end electronic block diagram and input-output signals of the ATLAS liquid argon calorimeters [12]

The front-end electronics of the scintillator based hadronic calorimeter in the ATLAS experiment is composed by a shaper and amplifier board, which is located in the photomultiplier tube (PMT) boxes and gives the analogue signal from the PMT a unipolar semi-Gaussian shape. This board is called 3-in-1 card [24]. In addition, the 3-in-1 card uses charge-averaging amplifiers to provide a signal for calibration and monitoring and analogue signals for the L1-trigger handling. Two amplified and shaped signals with high and low gain amplification are then differentially sent to a digitizer board. The digitizer board contains a 10 bit ADC with 40 MHz sampling frequency. The digital data are saved in a pipeline memory for the latency of the L1-trigger signal. If a L1-trigger signal indicates an interesting event, the data in the pipeline memory are read out selecting the high or low gain data according to the value of the maximum samples. The selected data are stored in the output memory that can store data from 36 events. The data from the output buffer are sent to the off-detector readout driver by an optical interface card, which adds a cyclic redundancy check and transmits the data via an optical link to the counting room. A block diagram of the front-end electronics is shown in figure 2.16.

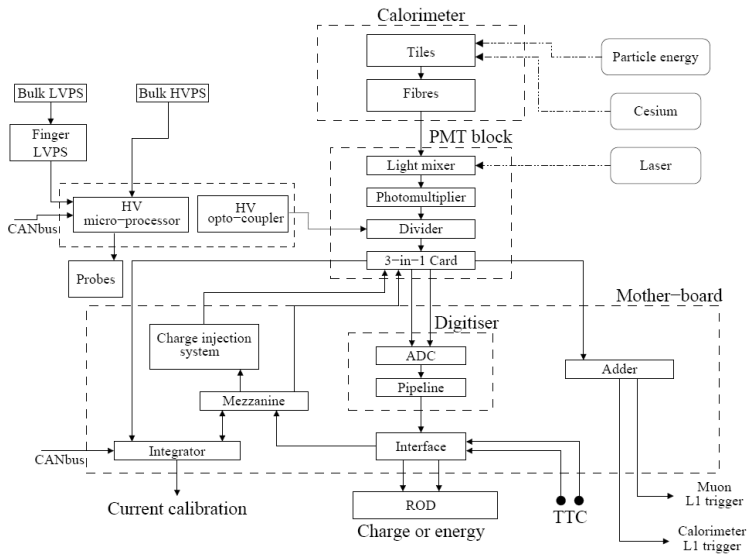


Figure 2.16.: Block diagram of the readout electronic of the ATLAS scintillator based hadronic calorimeter [12]

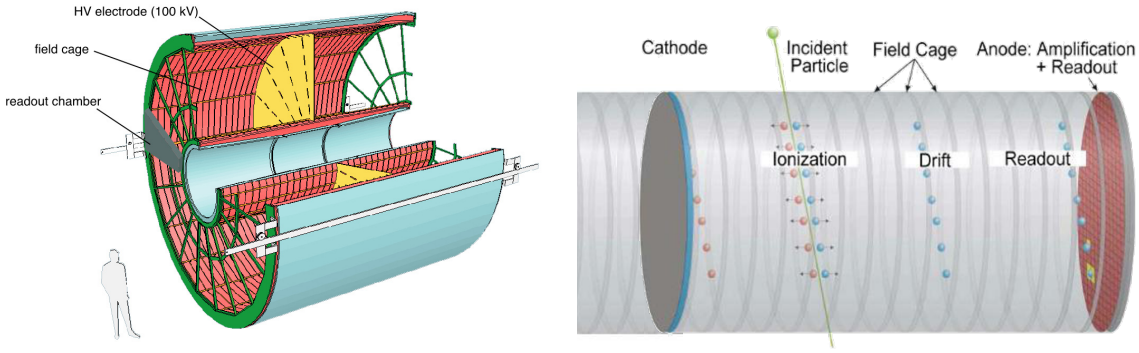
The LHCb experiment contains also a hadronic calorimeter made of iron absorbers and scintillators as active material. Wavelength-shifting fibers and PMTs are used to transfer the scintillating light in electrical signals. The electrical signal is shaped to a semi-Gaussian shape and digitized by a 12-bit ADC sampling at 40 MHz.

The ALICE experiment has a small hadronic calorimeter as forward detector called zero degree calorimeter (ZDC). This calorimeter is made of quartz fibers as active material and tungsten alloy or brass as absorber material. Particles passing through the absorber layers produce Cherenkov radiation in the quartz fibers. PMTs are detecting the light from the quartz fibers. The analogue signals from the PMTs are transferred via long coaxial cables to the counting rooms behind the detector cavern where the analogue signals are digitized and processed.

2.2.3. TPC detector

A Time Projection Chamber (TPC) composes the central detector of the ALICE experiment. The TPC consists of a cylindrical vessel filled with gas and it is used to provide a 3-dimensional image of the trajectory of charged particles emerging the interaction point. Two cylinders are forming

a 88 m^3 gas volume around the beam pipe as shown in figure 2.17(a). The diameter of the inner cylinder is 1.14 m and the outer cylinder has a diameter of 5.56 m. The gas filled cylinder volume



(a) Schematics of the TPC in the ALICE experiment [1]

(b) Working principle of the ALICE TPC [25]

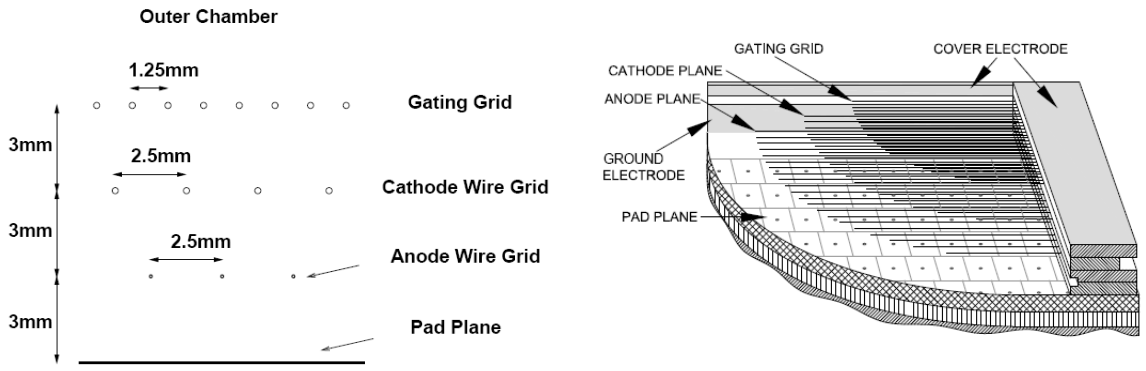
Figure 2.17.: Time Projection Chamber (TPC) of the ALICE experiment

is divided by a central electrode into two drift regions. The central electrode is biased to a high voltage (-100 kV) to provide a high electric field between the central electrode and the end-caps (grounded). The end-caps at both ends of the detector house the readout chambers, which contain Multi-Wire Proportional Chambers (MWPC) and the readout pads.

The working principle of the TPC is presented in figure 2.17(b). A charged particle, which emerges from the interaction point, traverses the gas volume and ionizes the gas. The liberated electrons in the gas volume are drifting towards the endplates guided by the high homogenous electric field (400 V cm^{-1}) whereas the ions are collected at the central electrode. In the MWPC at the end-caps, an avalanche process multiplies the drifting electrons. This amplification leads to a measurable electrical signal in the detector pads. The position of the pads which detect the signals provides an x-y resolution (min. pad size 4 mm$\times$7.5 mm) of the particle trajectory. The third coordinate (z) is obtained by measuring the drift time of the liberated electrons from the trajectory until the end-caps. Therefore, a 3-dimentional image of the charged particle trajectory is obtained. For a good resolution in z, the electric field inside the gas volume has to be highly homogenous providing a constant drift velocity of the electrons in the overall drift zone. A magnetic field of 0.5 T is applied to the TPC by the large solenoid magnet in the ALICE experiment (see 2.1). This magnetic field causes a deviation of the trajectory of charged particles. According to this deviation, the momentum of the charged particle is calculated. The track and momentum information in combination with the measurement of the specific energy loss dE/dx provides good particle identification. The induced charge in the detector pads is proportional to the specific energy loss of the ionizing particle. This is measured by the amplitude of the electrical signal (voltage) generated in the front-end electronics connected to the pad.

The MWPC in the readout chamber consists of three layers of wire grids above the pad plane as shown in figure 2.18. Above the pads, there is an anode wire grid and above the anode wires, a cathode wire grid is placed with the same pitch. Between this cathode wires and the anode wires a high electric field is formed, which chausses an avalanche initiated by an electron entering from the drift region. A gating grid above the cathode wires prevents that created ions from the avalanche enter the drift region, which would cause severe distortions to the drift (electric) field for following events. The gate also blocks electrons from the drift region to enter the MWPC if no valid trigger signal is received. The electrons and ions created in the avalanche process move to the anode wires and cathode wires respectively and induce a signal in the underneath positioned detector pads. The TPC in ALICE contains in total around 560 000 such readout pads.

The front-end electronics is connected via Kapton cables to the detector pads. The induced

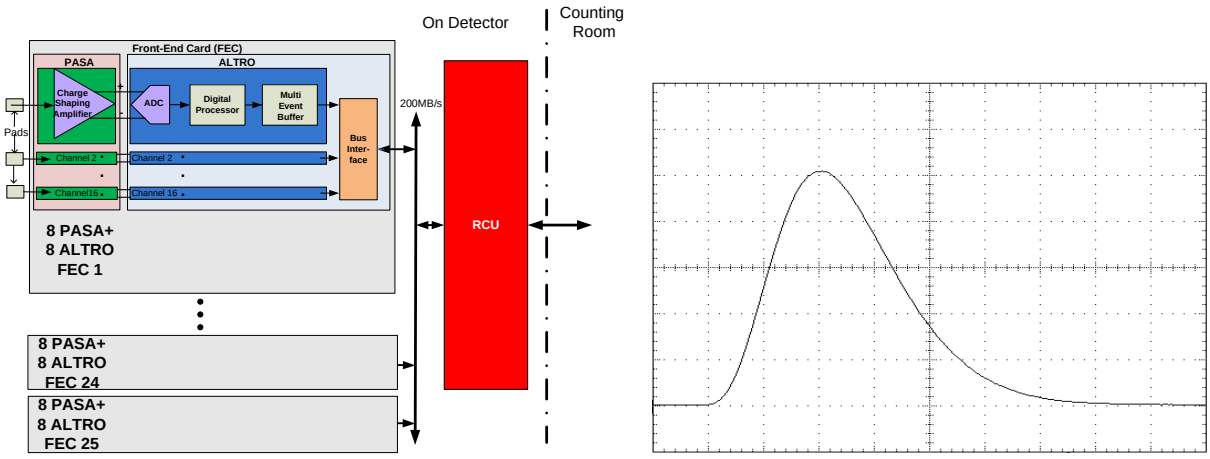


(a) Geometry of the MWPC wires in the outer part of the readout chamber [2]

(b) Cross section through a readout camber [3]

Figure 2.18.: Multi-wire proportional chamber (MWPC) inside the readout chamber of the ALICE TPC

electrical signal of each detector pad is first amplified and shaped by an analogue Pre-Amplifier and Shaper ASIC called PASA. The PASA chip contains 16 channels directly connected to 16 pads. Each channel integrates and amplifies the induced charge and send out an equivalent voltage signal with a semi-Gaussian shape and shaping time of around 190 ns (equal to Full Width Half Maximum (FWHM) of impulse response). The impulse response of the PASA is shown in figure 2.19(b). The PASA is connected differentially to a chip called ALice Tpc Read Out (ALTRO), which contains as well 16 channels. Each channel of ALTRO contains an ADC with 10-bit resolution and a digital signal processing unit. The ADC samples the semi-Gaussian shape with a frequency of 10 MHz. The digitized signal is processed then by the signal processing unit, which contains two baseline correction filters and a tail cancelation filter. The full signal chain of the front-end electronic of the TPC is sketched in figure 2.19(a).



(a) Block diagram of the front-end electronics

(b) Impulse response of the PASA having a semi-Gaussian shape 4th order (horiz. 100 ns/div, vert. 200 mV/div) [26]

Figure 2.19.: Front-end electronics of the TPC in the ALICE experiment and the shaped output signal of the PASA connected to the detector pads

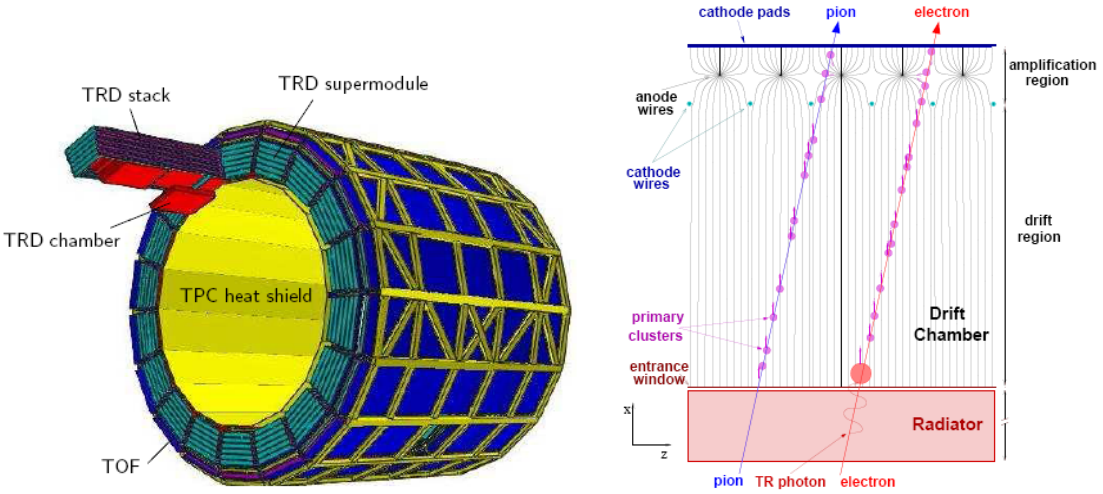
The signal-processing unit will correct the digital signal for baseline distortions and reduce the long tail caused by the slow moving ions of the avalanche process to perform then a data reduction using zero suppression. The zero suppression is performed on the digital signal with corrected, stable baseline by applying a threshold and saving only the sample values rising above the threshold level combined with a time stamp for each waveform cluster to preserve the arrival

time information (z resolution of trajectory). The zero suppressed digital samples together with the time stamp are saved in Multi-Event Buffer (MEB) memories if a L1-trigger signal has arrived.

A FPGA based control board called Readout Control Unit (RCU) is handling the data readout and is connected to up to 25 front-end cards each card containing 8 PASA and 8 ALTRO chips. If the RCU receives, a L2-trigger signal for accepting the event it starts reading out the MEB from each channel one after one and building a defined data format for transmitting the data via optical links to the counting room.

2.2.4. Transition Radiation Detector

The Transition Radiation Detector (TRD) of the ALICE experiment is used to provide electron identification for momenta higher than 1 GeV/c where the energy loss in the TPC is not high enough to provide a good tracking. The TRD is installed on the outside of the TPC below the Time-Of-Flight detector as can be seen in figure 2.1. The TRD surrounds the TPC and is made of 540 individual readout detector modules arranged in 18 super modules. Each super module is build by 30 readout modules. The layout of the TRD is shown in figure 2.20(a). Each readout



(a) Schematic view of the TRD layout [18] (b) Cross-section view of a detector module [18]

Figure 2.20.: Transition Radiation Detector of the ALICE experiment

module is made of a radiator with a gas filed drift chamber and a MWPC above it as can be seen in figure 2.20(b).

The radiator consists of polypropylene fibers in a carbon fiber-laminated Rohacell casing of 48 mm thickness [18]. The above drift region of 30 mm is filled with Xenon gas. The MWPC at the end of the drift region creates an amplification of the electrons by an avalanche process equal to the MWPC of the TPC. High energetic particles traversing the drift volume are ionizing the gas producing free electrons, which drift towards the MWPC guided by a high electric field ($0.7\,\mathrm{kV\,cm^{-1}}$). Particles with energies exceeding the threshold for transition radiation production e.g. high-energy electrons create in addition X-ray photons in the radiator. These X-ray photons are converted in the Xenon gas into additional free electrons, which increase the amplitude of the signal in the readout pads. The liberated electrons in the gas drift towards the MWPC where they are amplified through an avalanche process between the cathode and anode wires. The charges created in the avalanche induce a signal in the segmented cathode pads. The transition radiation is important to differentiate the high-energy electrons from the pions, which are not causing transition radiation. The detector front-end electronic is connected to the cathode pads and the schematic overview is shown in figure 2.21.

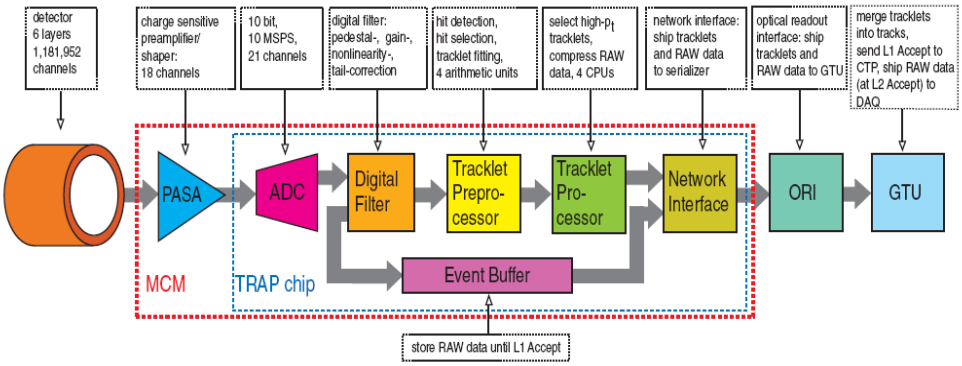
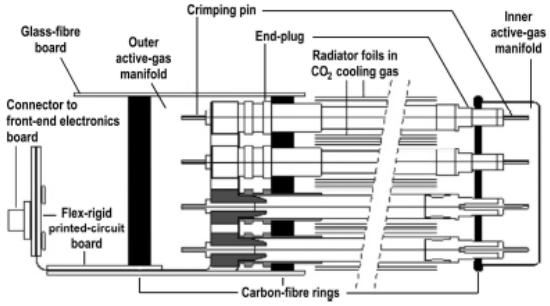


Figure 2.21.: Block diagram of the front-end electronics of the TRD in the ALICE experiment [18]

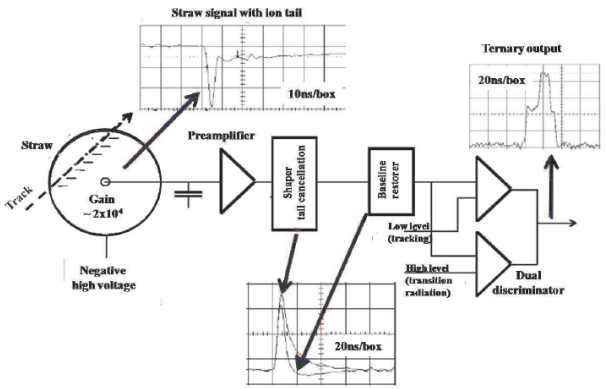
The front-end electronics contains as well a pre-amplifier and shaper ASIC called PASA as in the TPC front-end. This charge sensitive pre-amplifier and shaper transforms the induced signal in the cathode pad into a semi-Gaussian shaped voltage signal with 120 ns shaping time [18]. A 10-bit ADC samples the analogue shaped signal with 10 MHz and a signal processor made of digital filters performs a baseline correction and tail cancelation equal to the TPC front-end electronics. The filtered digital signal is once fed to an event buffer and second to a processing unit (Tracklet Preprocessor and Processor). The processing unit calculates the inclination of a track in the bending direction and the total charge along the track. This information allows identifying interesting particles (high transverse momentum). The short drift region compared to the TPC leads to a fast signal creation in the detector pads (drift time 2 µs) and to fast information about interesting events, which is used to created the L1-trigger information. If the L1-trigger indicates an interesting event, the data stored in the event buffer are serialized by the Optical Readout Interface (ORI) and send via optical fibers to the Global Tracking Unit (GTU) in the counting room. The GTU sends then the data to the DAQ upon a L2-trigger accept signal arrives.

The ATLAS experiment contains also a transition radiation detector, which is called Transition Radiation Tracker (TRT) and is installed in the inner tracking system (see figure 2.3). The TRT consists of small straw tubes (d=4 mm), which are filled with Xenon gas and contain a sensing wire in the center. The TRT is build of 52 544 straw tubes in the barrel region and 122 880 straw tubes in the end-cap wheels. Between the tubes, polypropylene fibers (barrel) and foils (end-caps) are installed and used as radiator material. Charged particles from the collision point, which traverse the straw tubes, are ionizing the gas in the tubes. The X-ray photons created in the radiator material by high-energy particles (e.g. electrons) are transferred in the Xenon gas inside the tubes into additional free electrons. These liberated electrons are drifting towards the sensing wires guided by a high electric field applied between the tube wall and the sensing wire. The small diameter of the straws and the isolation of the sensing wires within individual gas volume lead to a high detection rate. The schematic view of a TRT end-cap module is shown in figure 2.22(a).

The front-end electronic of the TRT is connected to the sensing wires and consist of an analogue amplifier and shaper chip called ASDBLR and a second ASIC for the data handling. The schematic of the ASDBLR chip together with the signal forms at different points in the signal chain are shown in figure 2.22(b). The shaper produces a semi-Gaussian waveform. To digitize the signal two discriminators are used one with a low threshold and the other one with a high threshold. With the two threshold levels, it can be distinguished if the signal is produced by a high-energy particle that causes a transition radiation or by a particle that does not cause transition radiation. A subsequent second ASIC performs a drift-time measurement, digital pipelining of the discriminator signals for the L1-trigger latency and an output buffering for the L2-trigger latency. This ASIC is connected via a 40 Mbit s^{-1} optical interface to the off-detector electronics.



(a) Schematic view of a TRT module [12]

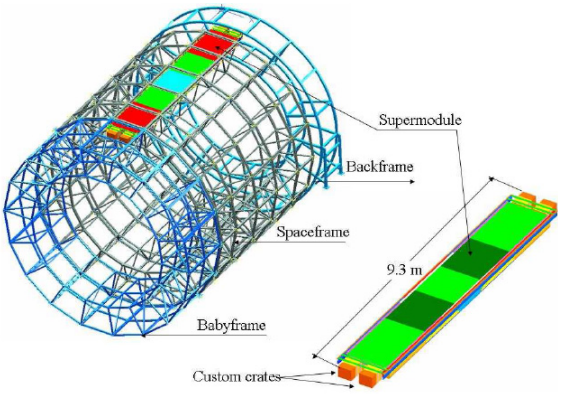


(b) Schematics of the ASDBLR TRT front-end electronic ASIC [12]

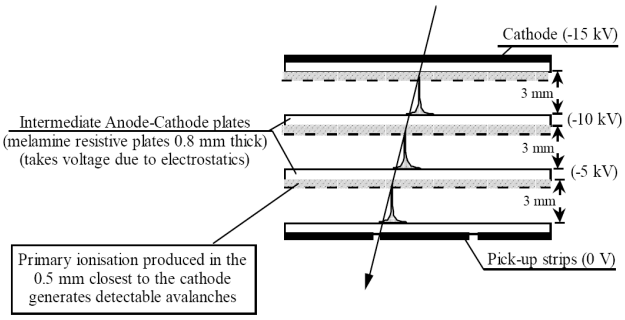
Figure 2.22.: Transition Radiation Tracker module and front-end of the ATLAS experiment

2.2.5. Time-Of-Flight detector

The Time-of-Flight detector (TOF) of the ALICE experiment is positioned on the outside of the TRD as shown in figure 2.1. The TOF consists of 18 super modules surrounding the TRD as shown in figure 2.23(a). Each super module contains 5 modules which are build of Multi-gap Resistive-Plate Chamber (MRPC). A MRPC consists of a small gap ($\approx$ 10 mm) filled with gas and a high uniform electric field is applied to this gap. To increase the detection performance multiple layers of anode-cathode plates are inserted in the gas volume, which create multiple intermediate gaps and a higher signal in the readout electrodes. The structure of a MRPC is shown in figure 2.23(b).



(a) TOF super module and the ALICE spaceframe [18]



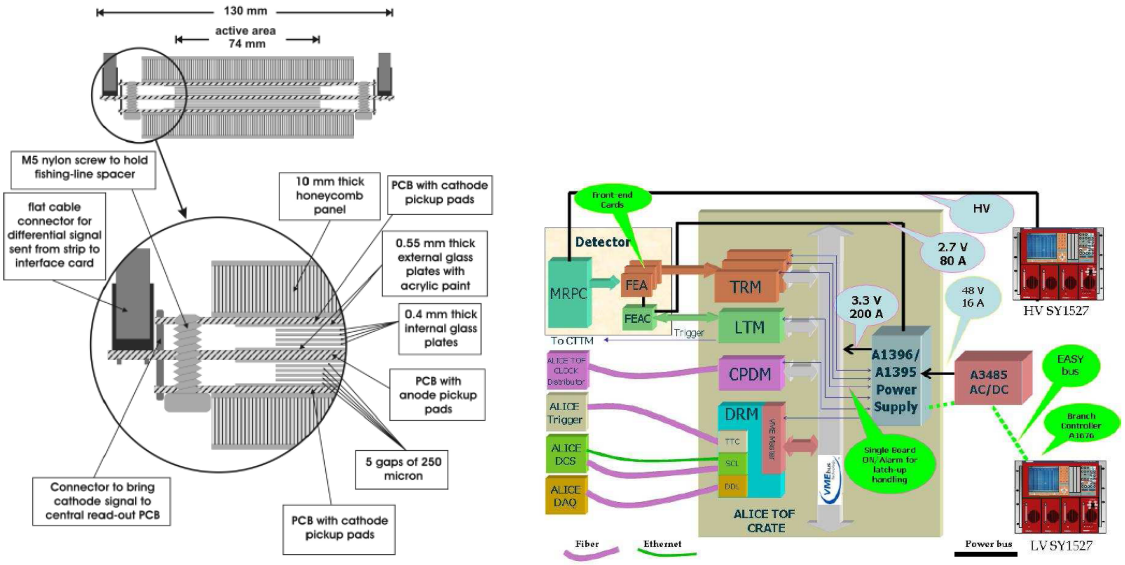
(b) Principle of a MRPC [27]

Figure 2.23.: ALICE Time-of-Flight detector (TOF) super module and Multi-gap Resistive Plate Chamber (MRPC) working principle

A charged particle, which traverses the gas volume, ionizes the gas and the liberated electrons immediately start an avalanche process in the gas gaps caused by the high electric field uniformly applied to the gas volume. The signal, which is induced in the pick-up electrodes, is the sum of the avalanches in the different gaps.

The advantages of a Multi-gap Resistive-Plate Chamber is that unlike other gaseous detectors there is no long drift time associated with the movement of electrons to a region of high electric field. The time jitter of this detector is related only to the fluctuation of the avalanche process. The induced signal contains no long ion tail since the gaps where the avalanches take place are small having short drift times. The charge spectrum is not of an exponential shape but has a well separated peak from zero [18]. The schematics of a MRPC strip cross-section is shown in figure

2.24(a) and contains two 5-gap gas chambers mounted back to back with two rows of 48 pick-up pads. In total, the TOF of the ALICE experiment contains 157 248 such readout pads. The intermediate electrode plats in the gas volume are electrically floating and capacitive coupled to the external plates.



(a) Schematic cross section of a 10-gap double-stack MRPC strip [18]

(b) Schematic of the TOF front-end electronics [18]

Figure 2.24.: ALICE Time-Of-Flight detector (TOF) Multi-gap Resistive Plate Chamber (MRPC) strip and the front-end electronics connected to the pick-up pads of the strips

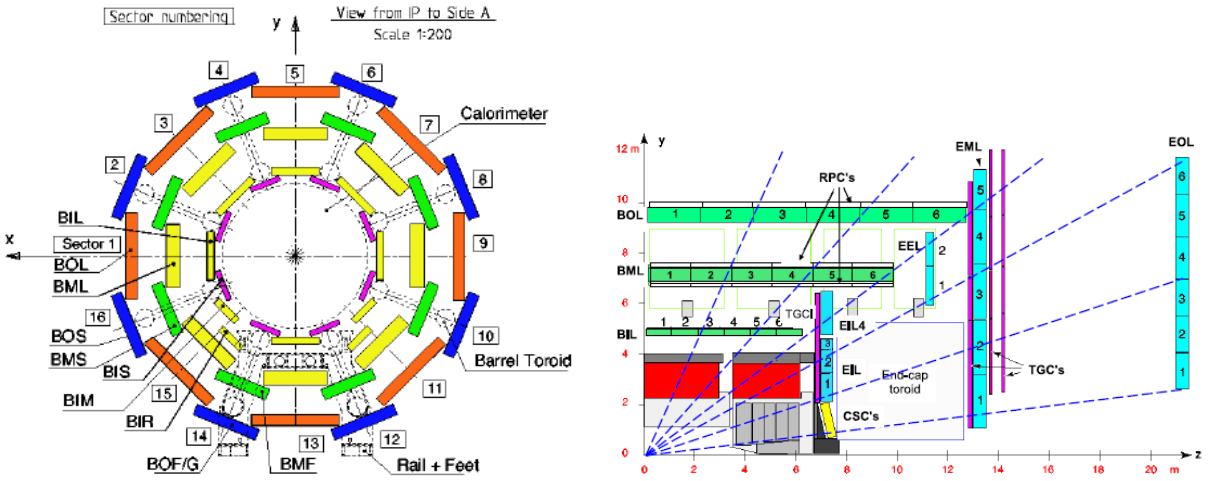
The front-end electronic connected to the readout pads consists of a front-end analogue ASIC called NINO which contains an input amplifier and a comparator both using differential signals, which provide information of the hit time (leading edge) and of the Time-Over-Threshold (TOT). Three NINO chips are mounted on a Front-End Analogue (FEA) card each chip contains 8 channels connected to 8 readout pads. A schematic of the TOF front-end electronic is shown in figure 2.24(b).

The signal from the FEA is send to a TDC (Time-to-Digital Converter) Readout Module (TRM) card, which contains high performance TDCs (24.4 ps resolution) to digitize the hit time of charged particles traversing the MRPC. A second card the Data Readout Module (DRM) reads and encodes the data from the TRM's and sends the data via optical links to the DAQ upon a trigger signal arrival (L1-trigger and L2-trigger). Two more cards are included in the custom crate of the TOF; the Local Trigger Module (LTM) and the Clock and Pulser Distribution Module (CPDM). The CPDM provides a high precision clock signal to the TRM, DRM and LTM.

2.2.6. Muon detectors

The outer part of the ATLAS and CMS experiment is covered by the muon spectrometer. These muon detectors are used to detect charged particles, which have energies high enough to pass the inner detectors and calorimeters. The muon spectrometer identifies muons and measures their momenta. Muons can be used to prove the existence of the Higgs boson and can revile some new information about the SUper-SYmmetry (SUSY) theory. The muon spectrometer is also designed to provide trigger information to the experiment for interesting events. ATLAS and CMS contain muon detectors in the barrel region and in several end-cap wheels. CMS uses three different types of muon detectors: Drift Tubes chambers (DT), Cathode-Strip Chambers (CSC) and Resistive Plate Chambers (RPC). The ATLAS experiment uses as well these types of detectors and contains

in addition Thin Gap Chambers (TGC) in the end-cap region. Drift tubes and CSC provide a precise momentum measurement. The fast RPC and TGC detectors are used as trigger generators, because they are able to provide track information within a few tens of nanoseconds after the crossing of a charged particle [12]. A cross-section of the ATLAS muon system perpendicular to the beam axis and in plane of the beam axis are shown in figure 2.25.



(a) Cross-section perpendicular to the beam axis [12]

(b) Cross-section of the muon system in a plane of the beam axis showing the four different muon detector positions. The numerated boxes contain the MDT's. [12]

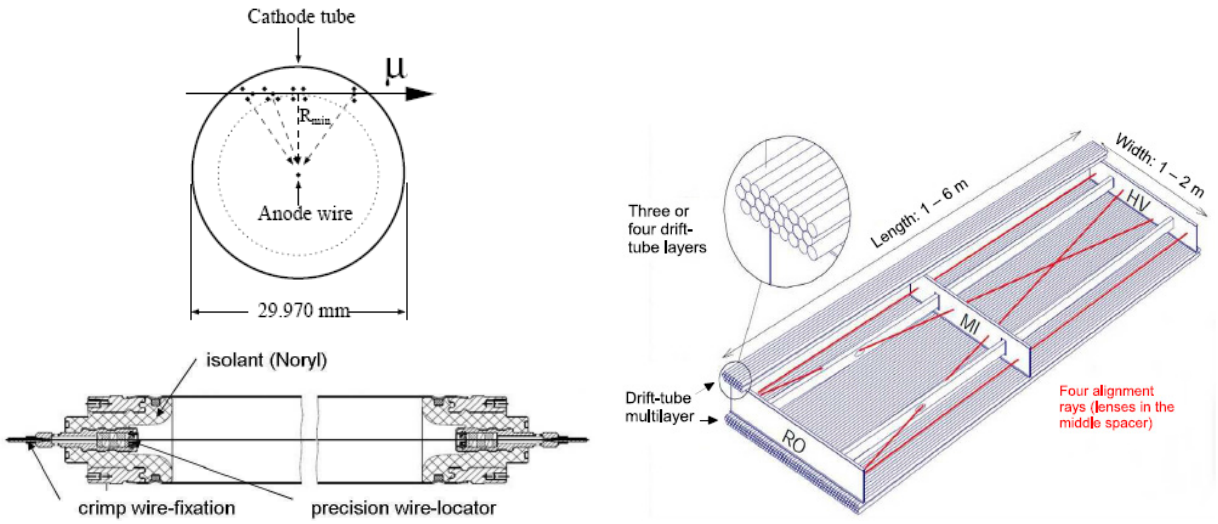
Figure 2.25.: Layout of the ATLAS muon system composed by Monitored Drift Tubes (MDT), Cathode-Strip Chambers (CSC), Resistive Plate Chambers (RPC) and Thin Gap Chambers (TGC)

Drift Tube chambers

The drift tube chambers are used as muon detectors in the barrel region of CMS and ATLAS and in the end-cap region of ATLAS. A drift tube chamber is composed by aluminum tubes with a diameter of around 30 mm. The tubes are filed with gas and held under pressure. In the center of the tube a gold plated tungsten-rhenium wire with a diameter of 50 µm is placed and charged to 3080 V. Charged particles emerging the collision point and traversing the gas-filled tube ionize the gas liberating electrons. The electrons drift towards the central wire guided by an electric field created between the tube wall and the central anode wire. The cross-section of such a drift tube in front and longitudinal view can be seen in figure 2.26(a). These drift tubes are similar to the once in the transition radiation detector in the ATLAS inner tracker but with a much larger diameter and without radiator material between the tubes. Several tubes are mounted together to form a drift tube chamber shown in figure 2.26(b). The number of tubes per chamber and the length of the tubes vary according to the position of the chamber in the experiment. In total, the ATLAS experiment contains 354 384 tubes each one connected to a front-end electronic channel.

The drift tube front-end electronics of ATLAS consists of an Amplifier/Shaper/Discriminator ASIC called (ASD), a Time-to-Digital Converter (TDC) chip and a Chamber Service Module (CSM). The block diagram of the front-end is shown in figure 2.27(a).

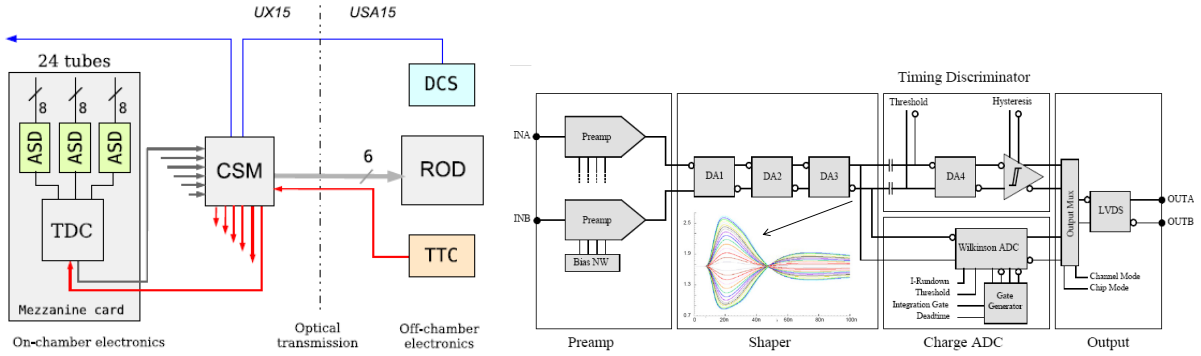
The ASD will amplify and shape the signals creating bipolar semi-Gaussian shapes. Then a threshold is applied to the shaped analogue signal by the discriminator (comparator) to produce a digital signal, which indicates a particle hit. The arrival times of the leading and trailing edges of the input signal are detected to obtain the TOT information. The schematic of the ASD and the shaper output is shown in figure 2.27(b). In addition, the analogue shaper output is fed also into an ADC, which will measure and digitize the signal charge in a defined time window (integrated charge) to determine the maximum signal height [28].



(a) Cross-section front view (top) and longitudinal view (bottom) of a Monitored Drift Tube [12]

(b) Mechanical structure of a MDT chamber with drift tubes on top and bottom [12]

Figure 2.26.: Monitored Drift Tube (MDT) detector of the ATLAS experiment



(a) Schematic of the MDT readout electronics [12]

(b) Block diagram of the MDT front-end ASD chip [28]

Figure 2.27.: Monitored Drift Tube (MDT) front-end electronics and shaper output signal

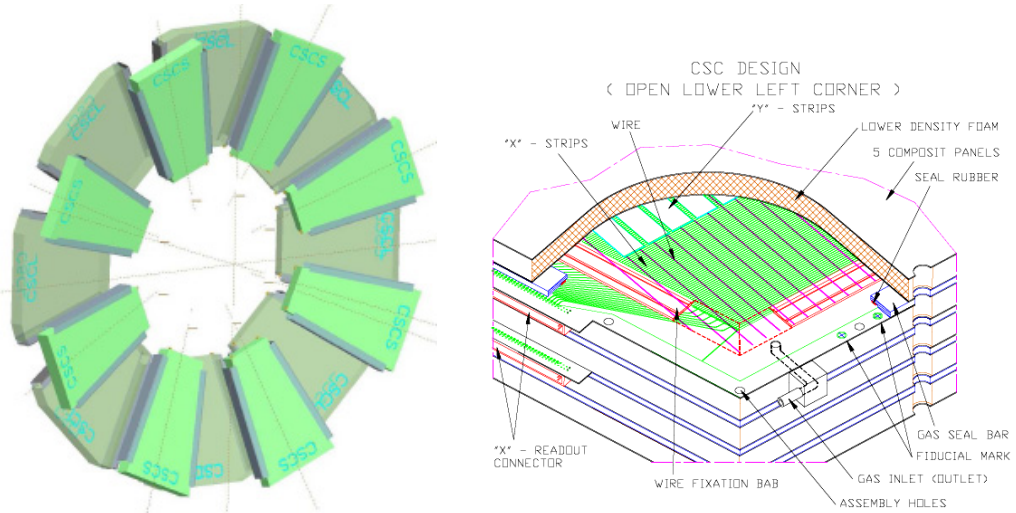
The binary differential signal from the ASD is send to the TDC chip which performs a time-to-digital conversion of the leading and trailing edge times with a resolution of 0.78 ns. The digital time information is stored in a buffer memory inside the TDC chip together with a flag bit to distinguish between leading and trailing edge, a beam-crossing identifier and a tube identifier.

The TDC data in the buffer are sent to the Chamber Service Module (CSM), which packages the data and sent them via optical link to the off-detector Readout Drivers (ROD) in the service cavern.

Cathode-strip chambers

Cathode-strip chambers are used in the end-cap region close to the collision point instead of MDT's, because they support better the higher rate of particles (radiation) traversing the detector area (counting rate $> 150\,\mathrm{Hz\,cm^{-1}}$). Cathode-strip chambers (CSC) combine high spatial and time resolution and provide double track points resolution [12]. The CSC chambers are based on MWPCs, similar to the ones used in the TPC and TRD of the ALICE experiment. The layout of the CSC end-caps of ATLAS is shown in figure 2.28(a).

The MWPC is composed by a gas filed chamber with anode wires in the center running in radial direction of the end-cap wheel. Both sides of the chamber contain cathode strips. These



(a) Layout of the CSC end-cap detector [12]

(b) Structure of the CSC module [12]

Figure 2.28.: Cathode-strip chamber of the ATALS experiment with two cathode strip layers in x and y direction

strips are oriented at one side parallel to the central anode wires and at the other side perpendicular to the wires. The structure of a CSC module is shown in figure 2.28(b). The charged particle traversing the chamber ionizes the gas and the liberated electrons are causing an avalanche in the high electric field between the anode wires and the cathode strips. The free charges drift towards the electrodes and induce a charge in the cathode strips. Between two strips connected to the front-end electronics, there are two intermediate strips, which are electrically floating and capacitively coupled to the readout strips. This capacitive coupled strips provide an additional charge interpolation, which increases the precision of the position measurement of the particle tracks [12].

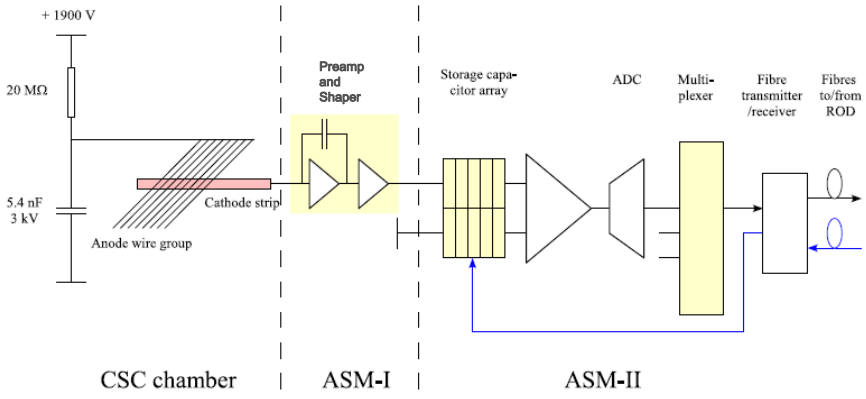


Figure 2.29.: Schematics of the CSC front-end electronics of the ATLAS experiment [12]

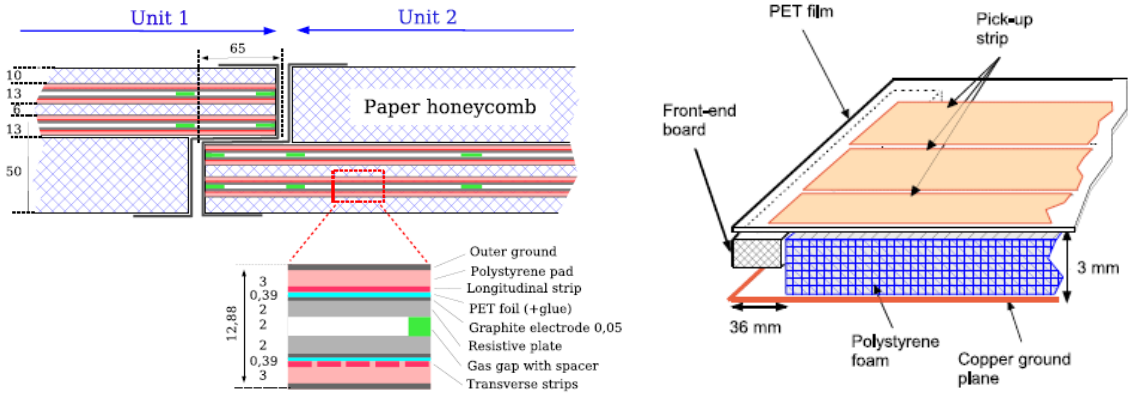
The front-end electronics connected to the cathode strips of CSC in ATLAS consists of an Amplifier Storage Module (ASM) containing an amplifier, shaper, an analogue storage and an ADC as shown in figure 2.29. The amplifier and shaper stage gives the induced signal in the cathode strip a bipolar semi-Gaussian shape. The shaped signal is sampled at a rate of around 40 MHz and stored in a Switched Capacitor Array (SCA), which works as analogue memory. The analogue memory is used to store the input signal for the L1-trigger latency. If a L1-trigger accept signal arrives indicating an interesting event the analogue data in the SCA are digitized by a 12-bit ADC, multiplexed, and transferred via optical fibers to the off-detector ROD's (Readout Driver).

The ROD performs a zero suppression and event data formatting and sends out the data to the DAQ system.

Resistive Plate Chambers

Three concentric cylinders of Resistive Plate Chambers (RPC) serving as trigger stations are installed in the ATLAS barrel region as shown in figure 2.25. The trigger chambers provide fast information on muon tracks, their multiplicity and approximate energy range, which are used to perform a L1-trigger decision [12]. The RPC chambers are mounted on top and/or bottom of the middle and outer layers of the MDT's.

A RPC chamber consists of two parallel resistive electrode-plates with a 2 mm gap between them, which is filled with gas similar to the TOF detector of ALCIE. A high electric field (about $4.9\,\mathrm{kV\,mm^{-1}}$) is applied to the gas volume through the two resistive plates, which are made of plastic laminate and graphite electrodes. Charged particles traversing the gap ionize the gas and the ionization electrons cause avalanches due to the high electric field. The electrons and ions from the avalanche drift in the electric field towards the electrodes and induce a signal in metallic strips mounted on the outer face of the resistive plates. A cross-section of a resistive plate chamber is shown in figure 2.30(a) together with the structure of a RPC module consisting of two units with two chambers. The front-end electronics is connected to the metallic strips, which are mounted longitudinal on top of the chamber and transversal on the bottom of the chamber. This orthogonality of the strips provides a resolution of the particle track in 2-dimensions.



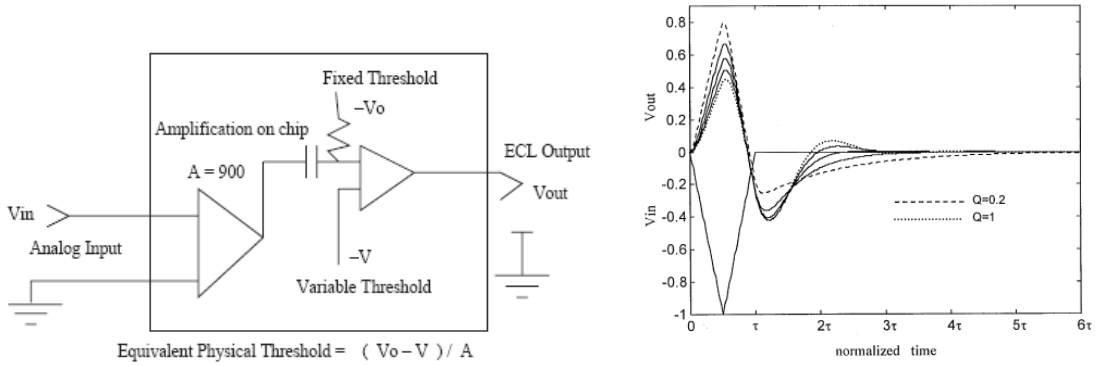
(a) Structure of the resistive plate chamber module composed by two units. Each unit has two RPC's. The cross-section of an RPC is shown at the bottom [12]

(b) Layout of the RPC readout strip plane [12]

Figure 2.30.: Resistive plate chamber (RPC) muon detector in the ATLAS barrel region

The front-end electronic board is mounted directly along the edges of the RPC and soldered to the metallic readout strips as illustrated in figure 2.30(b). The front-end electronics consists of a three-stage shaping amplifier followed by a comparator as shown in figure 2.31(a). The shaping amplifier produces a bipolar semi-Gaussian output signal as shown in figure 2.31(b).

The shaped signal is then compared to a threshold by the comparator (discriminator) and if the input signal raises above the threshold a digital impulse is send out by the discriminator preserving the arrival time of the input signal. The signals of all three layers of the RPC muon trigger in the barrel region are combined and used to obtain a L1-trigger decision.



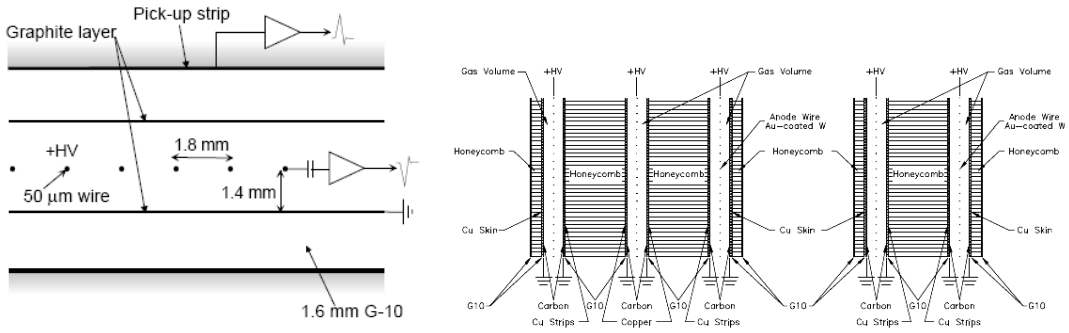
(a) Schematics of the front-end electronics [29]

(b) Shaped output signal of the Amplifier/Shaper stage [30]

Figure 2.31.: Front-end electronic of the RPC in the ATLAS experiment

Thin gap chambers

In the end-cap regions of the ATLAS muon detection system Thin Gap Chambers (TGC) are used as trigger stations instead of RPC's because of the high particle rate in this end-cap regions. The position of the TGC's is shown in figure 2.25. The TGC is a MWPC similar to the one of the TPC and TRD in the ALICE experiment, but with the characteristic that the distance between the wire and the cathode plane is smaller than the wire-to-wire distance. A small wire-to-wire distance and a high electric field around the wires leads to a very good time resolution of the particle tracks [12]. The structure of a TGC is shown in 2.32(a). There are two types of TGC modules in the end-cap wheels of the ATLAS experiment one with three TGC's (Triplet) and one with two TGC's (doublet) as shown in figure 2.32(b).



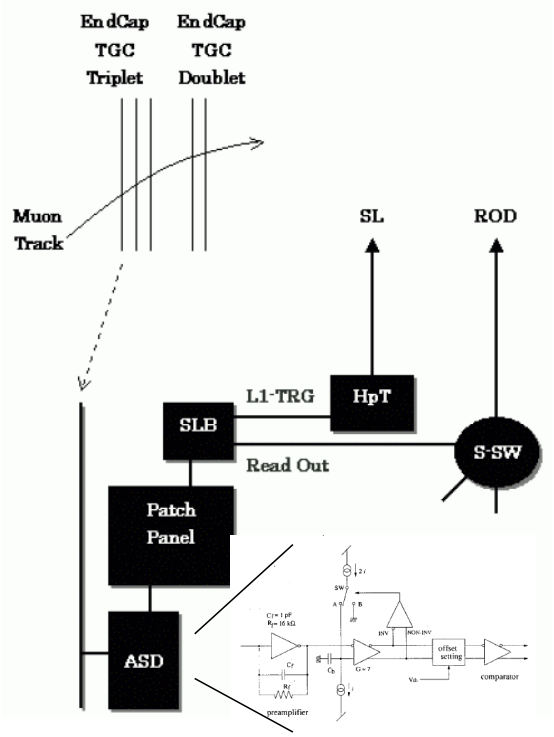
(a) Structure of a Tin Gap Chamber [12]

(b) Cross-section of a TGC triplet and doublet module. The triplet has three wire layers but only two strip layers. The doublet has two wire and strip layers [12]

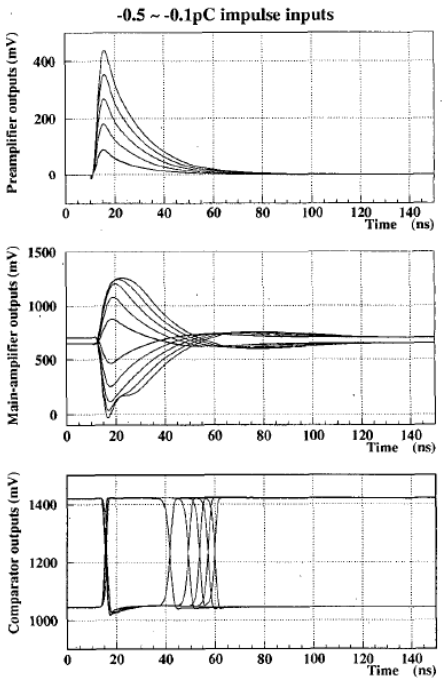
Figure 2.32.: Thin-Gap Chambers (TGC) used in the muon system of the end-cap region of the ATLAS experiment

A charged particle ionizes the gas in the chamber and an avalanche process starts, which induces a signal in the anode wires and in the cathode strips running perpendicular to the wires, which gives a 2-D resolution.

The front-end electronic connected to the strips and the wires consist of an Amplifier-Shaper-Discriminator chip (ASD) and is shown in figure 2.33(a). The ASD chip performs the amplification and shaping of the induced signal, which results in a bipolar semi-Gaussian shape. Then the semi-Gaussian shaped signal is digitized by a comparator with an adjustable threshold to set an adequate level for the wire signals and for the cathode strip signals. The output signals of the



(a) Front-end electronics of the TGC [31], [32]



(b) Output signals of the pre-amplifier, the main-amplifier with shaper and the discriminator (comparator) [32]

Figure 2.33.: Front-end electronics of the Thin Gap Chambers (TGC) of the ATLAS experiment

pre-amplifier and main-amplifier as well as the comparator output of the discriminator are shown in figure 2.33(b).

The digital output signals of the ASD's are sent to a Patch-Panel (PP) ASIC which performs a time-alignment and synchronization to the beam-crossing frequency [12]. The PP ASIC is mounted on the back of the chamber close to the ASD's. A SLave Board (SLB) mounted on the back of the chamber is taking the signals of the PP ASICS and performs a coincidence decision as well as a data formatting and buffering for the L1-trigger latency. The SLB combines the three signals from the triplet wires to perform a 2-ot-of-3 coincidence decision for improving the rejection of fake trigger. For the duplets, the SLB performs a 3-out-of-4 coincidence decision using the two wire layer signals and the two cathode strip layer signals. A Hi-p_T (high momentum) and Star Switch Control (HSC) crate which is mounted at the outer rim of the chamber sends the readout data via the Star-Switch (S-SW) to the off-detector readout drivers.

2.3. Summary

Different detector concepts are explained in this chapter and the structure of their front-end electronics is presented. The front-end electronics contain a pre-amplifier and shaper stage, which transforms the induced charge signal from the detector pads in a voltage signal. This stage consist of an integrator to translate the induced chare in a voltage signal. The integrator follows a differentiator, which brings the integrator output voltage back to ground. Then low pass filters are used to limit the bandwidth of the voltage signal. The output voltage from this stage has in most cases a semi-Gaussian shape that can be represented mathematically by a gamma function 4th order. In some detector front-ends the semi-Gaussian waveforms are bipolar and in others they are unipolar as shown in figure 2.34. The amplitude of the semi-Gaussian waveform is proportional to the

induced charge in the detector pads and therefore also related to the energy-loss of the charged particles in the detector, which emerge the collision point. Some discussed particle detectors in this chapter use ADCs to digitize the semi-Gaussian shaped voltage signal to preserve the amplitude information and therefore the energy-loss information of the detected particle. To reduce data most of the detectors use a zero suppression, which eliminates the baseline data before and after an interesting signal and retains only the sample values of the interesting waveform. Nevertheless also the zero suppressed data amount is high and therefore an additional digital data compression algorithm should be found to further reduce this kind of digital data. The amount of data saved per semi-Gaussian signal depends on the number of samples and their range of values. The range of the sample values is represented by the ADC resolution in the front-end electronics. The number of samples per semi-Gaussian waveform is given by the shaping time of the pre-amplifier/shaper stage and the sampling rate of the ADC and depends on the duration of the waveform. The signal duration depends on the detector structure, the charge creation of the detected particle and the avalanche process in the detector amplification region.

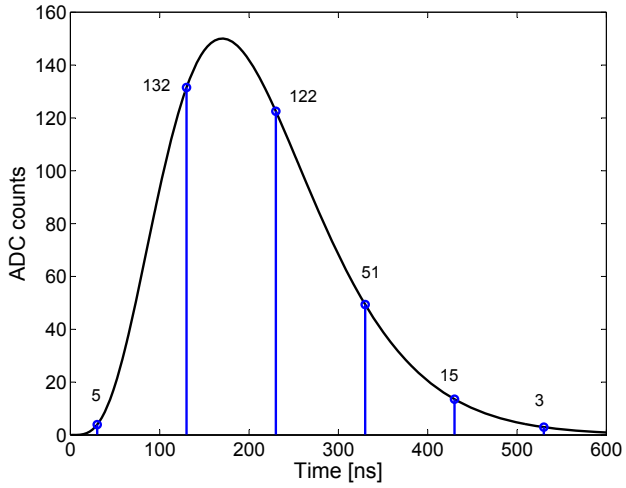


Figure 2.34.: Semi-Gaussian signal with the digitized sample values (sampling rate 10 MHz)

Summing up the properties of the digital data, which have to be compressed by the searched compression method, it will be consider in the following that they consist of digitized semi-Gaussian waveforms with varying amplitudes and signal durations. The variation in amplitude leads to large variations in the sample values requiring a certain bit-resolution (in most cases 10 bit). The variation in signal duration leads to varying numbers of samples per waveform. The samples are normally stored in memories in the front-end electronics to buffer them for a L1-trigger latency. This allows the data compression algorithm either to compress the samples individually or to compress vectors containing all samples of one waveform.

3. Data compression

Data compression is well known from software applications for home computers to reduce disc space occupation and to reduce data size for transmission via internet, email or facsimile. The most popular compression tools in this field are WinZip, GNUzip and compression standards for audio files like mp3 and image files like jpeg or others. In telecommunication systems, data compression is mainly used in source coding components to increase the amount of data, which can be transferred over a communication channel in a certain time.

The objective of this work is the reduction of the size of data transfer and data storage in applications for high energy physics experiments. Detectors in such experiments produce a huge amount of data that have to be transferred from the detectors to the counting rooms, where computer-farms perform first analysis of the data. Then the data collected from the detectors have to be stored and distributed to different Institutes all over the world for further investigating the data and extracting relevant physics information. To reduce the amount of data as early as possible in the data chain, e.g. already in the front-end electronics, a data compression algorithm is developed. An important attribute of this algorithm is that it should be possible to implement it in real-time using relative simple hardware.

Different compression methods are presented in this chapter and the block diagram in figure 3.1 shows a breakdown of them. Each of the different compression schemas is evaluated in terms of its usability for the underlying kind of data.

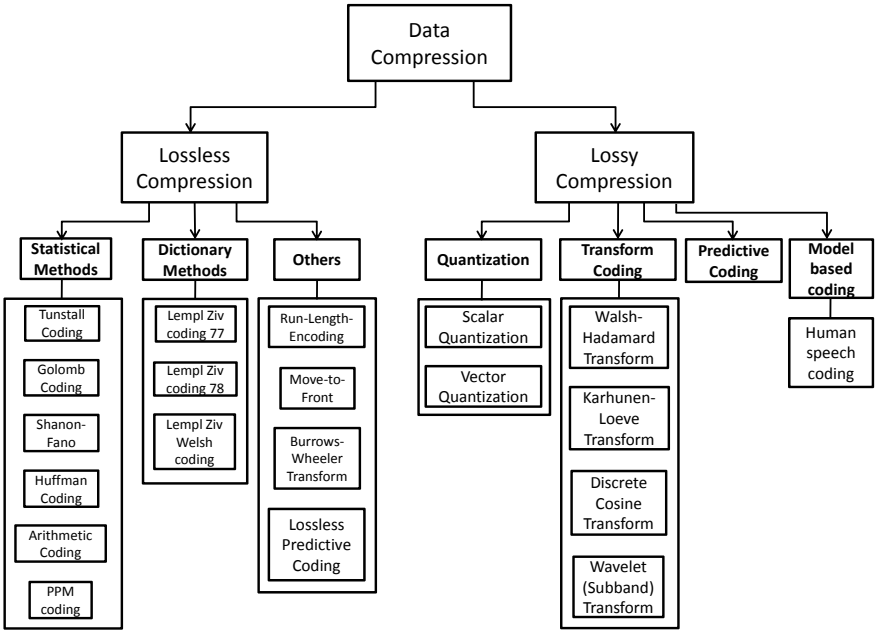


Figure 3.1.: Breakdown of the lossless and lossy compression methods presented in this chapter.

A detailed discussion and comparison of the most promising methods is then given in chapter 4 based on their performance on example data from real measurements carried out with the TPC of the ALICE experiment.

The descriptions of the compression methods in here are elaborated using the references [33–36].

3.1. Lossless compression

Lossless compression methods attempt to reduce as much as possible the redundancy of the input data. Therefore, the original data are analyzed and characterized to find encoding schemes which represent the same information using a minimum of redundancy. The reduction of redundancy caused by the compression reduces the error robustness of the code and makes data transmission more problematic. Some of these encoding methods are specialized to certain types of data coming from defined data sources others can be used on every kind of input data (general-purpose methods) The encoding of the data has to be performed in a way that the decoder can perfectly reconstruct the original data. Some of the information used for the encoding have to be transferred to the decoder, while others are predefined and are already known to both the encoder and the decoder. For lossless data compression three main concepts can be defined:

The frequency based methods (statistical methods), which in a first step analyze the frequencies of the different symbols in the input data and then compress the input data using these probabilities. Otherwise, a statistical model for the specified data source can be used to obtain the probability values of the different input symbols. Variable length codewords are then assigned to the different symbols, according to the probabilities, in such a way that frequently occurring symbols (high probability) produce short output codes and unlikely occurring symbols produce longer codewords. In this way the variable length codes can reduce the average number of bits per symbol and compress the underlying data set.

The dictionary-based methods create a dictionary containing blocks of input data like words or phrases. The encoder searches for entries in the dictionary, which match the input data and sends out the position of the best matching entry. Since the original data block is substituted with just the position number compression can be achieved. The decoder has to have the same dictionary or be able to construct the same dictionary from the (de)compressed data.

Furthermore, there are also some other methods that do not fit in one of the two previous categories which for example use substitution of blocks of input data with special characters or count the number of consecutive equal symbols like run-length-encoding.

Some of the most popular lossless compression methods are described in the following.

3.1.1. Run-Length encoding

Run-length encoding (RLE) is a lossless data compression method, which is used for example in facsimile transmission. The idea is simple; Always when several consecutive identical symbols S in the input bit stream are appearing the run-length encoder replaces these run of symbols with a run-length-encoding pair nS where n is the count, i.e. the number of consecutive equal symbols. If for example there are five symbols a in the input stream $xaaaaaz$ the run length encoder sends out $x\ 5a\ z$, which compresses the bit stream from 7 symbols to only 4 symbols [33]. If the input data can contain also numbers and not only letters a problem of RLE arises in how to distinguish the count number of a RLE pair from a normal occurrence of a number in the input data. There are different solutions to this problem which are described in the following.

One possibility is to precede every pair of count and symbol (nS) with an escape character, which is a symbol that cannot appear in the input stream. If an escape character, e.g. the @ symbol, is used, then the bit stream above is compressed as $x\ @5a\ z$. With this escape character the decoder has no problem anymore to distinguish between numbers contained in the input data and the count number of a RLE pair. The drawback is that an additional symbol has to be added to the compressed data, which reduces the compression efficiency. Therefore, 5 symbols have to be

used now to represent the 7 input symbols instead of 4. That means also that only runs of three and more consecutive equal symbols should be substituted by the RLE pair, whereas three consecutive symbols lead to no compression. Runs of four and more characters are resulting in compression for this case. In normal text files, it is quite unlikely to find many of four consecutive equal characters. In mathematical texts, the numbers can have four or more consecutive equal characters and RLE could lead to good compression. Better results could be archived in image compression, where runs of pixels with equal color values can occur.

If all possible characters or bit combinations could appear in the input data, no special character (escape character) can be identified. Another method to distinguish a RLE pair from the input symbols is required. Such a method is used in modems and is known as MNP5 (Microcom Network Protocol). The MNP5 method defines a RLE pair by writing three consecutive equal symbols followed by a count number on the output data. If the encoder identifies a run of three or more equal symbols it writes three equal symbols followed by the number of total, consecutive equal symbols (whereas a count of 0 means only three equal symbols are present). The decoder knows that if it receives three equal symbols the following symbol is the count number. The disadvantage of this method is that a run of three consecutive equal symbols leads to four characters in the output data and therefore to expansion. A run of four equal symbols do still not produce compression. To achieve a compression with this method five or more consecutive symbols in the input data have to occur.

A third possibility is to use one additional bit for each symbol to indicate whether it is a count of a RLE pair, or a symbol of the input stream. If for example the pixels of a grayscale image have a maximum value of 128 then each pixel can be represented with 8 bit (1 byte) where the most significant bit (MSB) represents a flag bit. This flag bit indicates if the byte is a grayscale value or a count.

A variant of this is to not add the additional bit to each symbol but to group them together to an additional word. For example if each pixel of a grayscale image is represented by a byte then after 8 byte written on the output data an additional byte is added, which contains the 8 flag bits to classify the preceding 8 byte. In this example, the additional byte increases the encoded output bit stream by 12.5%.

Another possibility is not to precede RLE pairs with a special character, but precede a run of different symbols by the number of different symbols until the next RLE pair. The different input symbols between two RLE pairs can be identified by preceding them with a negative number, whereas the absolute value represents how much consecutive different symbols are following. A negative number in the output bit stream like -4 indicates that the following 4 symbols are not RLE encoded, they are normal different input symbols and after them the next RLE pair follows. This is advantageous in cases where many RLE pairs are present and between them only a few different symbols.

In bitmap images of only black and white pixels, for example in facsimiles of text documents, the encoder and decoder agree whether each row starts with a white pixel or a black pixel. The image is then encoded row by row and just the counts of consecutive white pixels and black pixels are written on the output stream. If for example the encoder writes 17, 1, 55 to the output file and the agreement says that each row starts with a white pixel, it means that the row starts with 17 consecutive white pixels followed by one black pixel and then again 55 white pixels occur. If a row starts with black pixels, the encoder writes 0 to indicate that no white pixel is at the beginning of the row and then the number of the black pixels which are at the beginning. As the decoder knows from the agreement that each row stars with white pixels it can perfectly decode the rows. To split the encoded bit stream in the correct number of pixels per row the encoder has to add the size of the image to the output data. A good encoder should be able to scan images by row, by column or in a zigzag way. If for example, an image contains a large number of vertical lines but this image is scanned and compressed row by row, the compression efficiency is very low. However,

scanning this image column wise would result in a good compression. The encoder should scan and compress the image in all three ways and then decide which scan method produces the best compression and retain the corresponding compressed data.

To increase the efficiency of RLE the input data can also first be modified in a way that not only the absolute values of the symbols are encoded but differences to a reference value. For example in sensor networks, a temperature sensor could collect the temperature every hour. Instead of sending all the measured temperature values, only the first temperature could be send followed by the differences between the first temperature value and the successive measured temperatures. Successive measured temperatures normally differ not by much and the resulting differences should have small values. The encoder could have a limit for the differences so that if the differences are below this limit they are send out, otherwise the absolute measured temperature values are send. The use of the differences has the advantage that since their values should be small they need fewer bits to be represented and they are concentrated in a smaller range, which could then lead to more runs of equal values and increase the performance of RLE. To distinguish between absolute values and difference values some of the methods described before can be used. In image compression neighboring pixels are often not differing by much and the use of differences instead of absolute values can lead to improved compression performance of compression methods.

Another compression method similar to RLE is based on the idea of substituting frequently used words or common combinations of letters (digram) in text files with symbols, which cannot occur in the input data or with a special character and a signature of the digram. In English texts for example the combination th is really common and this could be substituted with another ASCII character which is not used in English writings. In computer programs common words like $printf$ could be substituted with a special character such as @ followed by a signature like p resulting in $printf(a)$ being substituted by $@p(a)$.

Suitability of RLE for the implementation of a detector data compression:

+ Advantageous is that a hardware implementation is easy to realize requiring mainly a counter
– Disadvantageous is that the expected performance is low for data containing semi-Gaussian waveforms because consecutive equal sample values are not likely due to this signal shape. The baseline between interesting signals is already RLE by the so-called zero suppression, which is explained in chapter 4 in more detail.
– Better performance can be expected by first calculating the differences between consecutive samples of the input waveform and then RLE the differences. However, the semi-Gaussian shaped input waveforms have different amplitudes and no regular shapes, that leads not to many consecutive differences being equal.
– Disadvantageous is also that the semi-Gaussian input waveforms have normally not a high number of samples. Compressing several semi-Gaussian waveforms at once is also not so promising since they can have different amplitudes.

3.1.2. Move-to-Front encoding

The idea of this method is to update the alphabet in the dictionary in a way that frequently occurring symbols are located near to the front positions of the dictionary. Each symbol is encoded by the position number where it appears in the alphabet. Frequent symbols should get small position numbers indicating a position near the front of the alphabet. If for example the alphabet is containing the symbols $A = h, e, r, ...$ and the next symbol to encode is e it gets represented by the position number 1 because one symbol precedes e in the alphabet. Before the next input symbol gets encoded the encoder updates the alphabet by moving the actually encoded symbol e in front of the alphabet so that now $A = e, h, r, ...$ is used the next time. If the next input symbol is again an e this gets encoded this time as 0.

The advantage of getting small position numbers is related to a possible combination of this method with Huffman coding or arithmetic coding. In the case of using Huffman coding the encoder will assign the position numbers variable size codewords in a way that the small numbers get assigned short codewords and larger position numbers get assigned long codewords. This can result for example in the following codes for the position numbers [33]:

$0 \rightarrow 0; \quad 1 \rightarrow 10; \quad 2 \rightarrow 110; \quad 3 \rightarrow 1110; \quad 4 \rightarrow 111110; \quad 5 \rightarrow 111110$

By using the move-to-front method, local frequencies of the symbols are exploited to compress concentrations of identical symbols in the input stream. In input streams with a good concentration property, the move-to-front method performs better than the normal Huffman coding alone. An advantage is also that the encoder has not to analyze first the entire input data to extract the frequencies of the different symbols for building the Huffman codewords. The Huffman codewords are defined previously and the move-to-front method performs the correct assignment of the codewords according to the local frequencies. The method has a drawback if symbols are appearing frequently in the input stream but not in consecutive order, then the move-to-front method creates poorer results than by other methods, which first analyze the frequencies of the entire input data. This behavior is shown in a small example for better illustration of the presented method [33].

The alphabet used in this example is containing the following symbols I, M, S, W. The two strings $SWISSMISS$ and $MISSSWISS$ containing these characters in different order are encoded and shown in table 3.1. In addition, the changing alphabet is shown and both strings are compared with the position numbers in column (b) resulting by not applying the move-to-front method to the alphabet. If the average of the position numbers is calculated for the first string

Table 3.1.: Move-to-front method for $SWISSMISS$ and $MISSSWISS$

	IMSW			IMSW			IMSW	
S	SIMW	2	S	IMSW	2	M	MISW	1
W	WSIM	3	W	IMSW	3	I	IMSW	1
I	IWSM	2	I	IMSW	0	S	SIMW	2
S	SIWM	2	S	IMSW	2	S	SIMW	0
S	SIWM	0	S	IMSW	2	S	SIMW	0
M	MSIW	3	M	IMSW	1	W	WSIM	3
I	IMSW	2	I	IMSW	0	I	IWSM	2
S	SIMW	2	S	IMSW	2	S	SIWM	2
S	SIMW	0	S	IMSW	2	S	SIWM	0
	(a)			(b)			(c)	

$SWISSMISS$ in column (a) it results in 1.77. The second column (b), which shows the same string as in (a) but encoded without updating the alphabet results in an average of 1.55. The reorganized string $MISSSWISS$, in column (c), results an average of 1.22. This shows that the first string in column (a) has a higher average and therefore larger position numbers due to the move-to-front method compared to (b) without the move-to-front. This is related to the fact that the concentration property of this string is lower having only, two times two identically consecutive characters. The string in column (c), containing the same characters but in a different order, performs better by using the move-to-front method then without updating the alphabet. The second string has a higher concentration property having one run of three consecutive identically characters and another run of two consecutive identically characters. To increase the concentration property of the input data a method can be used before performing the move-to-front called the Burrows-Wheeler transform, which is explained in the next section.

Some variants of the move-to-front method are shortly explained as followes.

1. Move-ahead-k. The current input symbol in the alphabet A is moved ahead only k positions and not all the way to the front. The value k is defined by the user. This decreases the performance of input streams with a high concentration property but it can work better for input streams with low concentration property. If k is chosen $k = n$, where n is the number of elements in the alphabet this equals to the move-to-front method. The case $k = 1$ requires only a swap of the actual input symbol in the alphabet with the one preceding its position. This is simple and produces less complexity for an implementation.

2. Wait-c-and-move. In this method, the elements in the alphabet are not moved after each encoding of a symbol. An element in the alphabet is only moved to the front after it appeared c times in the input stream. The c times of appearance of a symbol have not necessarily to be seen consecutively. This reduces the times the alphabet has to be updated and is therefore advantageous in implementations where updating the alphabet is slow.

3. Move-words-to-front. This method can be used for example in text compressions. The alphabet contains not just single characters but entire words. Each input word is encoded by the position of it in the alphabet, which results in a great compression. The entire input word is then moved in the alphabet to the front. The problem is that an alphabet containing entire words can become large requiring a lot of memory space in the encoder and decoder. These large memory requirements are in most hardware implementations not realizable. A solution to this problem is that the encoder and decoder will start with an empty alphabet. The encoder reads a word from the input stream and if the word is not already contained in the alphabet, it adds the word to the alphabet in the next free position. Then the position of the new word is send out followed by the word itself. The decoder reads the position number and detects that this position is empty in its alphabet and it knows that the following symbols are representing a new word terminated by the space character or the end-of-text. The decoder adds the new word to the empty position in its alphabet. If an already contained word in the alphabet is appearing at the input, only the position number is sent out from the encoder. If the memory space reserved for the alphabet is full, the entire alphabet can be deleted and everything restarts again, or only the last entry is deleted if a new word appears in the input stream. In both cases it has to be defined a way to tell the decoder to perform the chosen action.

Suitability of Move-to-Front for the implementation of a detector data compression:

+ Advantageous is that a hardware implementation is easy to realize requiring only shifts of memory words or pointer systems.
– Disadvantageous is that the memory accesses to perform the several memory shifts for each input sample require some clock cycles which makes it difficult to realize a real-time implementation.
+ Advantageous is that the code tables for entropy coding are easy to build, not requiring knowing the frequencies of the different sample values.
– Disadvantageous is that this method uses the concentration properties of the sample values, which is normally not high since the semi-Gaussian waveforms have different amplitudes and consecutive samples have mostly quite different values.
+ This method could be used in combination with other methods to increase a bit the performance of the entropy coding but the additional effort for the implementation has to be taken into account.

3.1.3. Burrows-Wheeler Transform

Burrows-Wheeler coding or Burrows-Wheeler transform is a context based method, which works on sequences of input symbols. The Burrows-Wheeler transform (BWT) do not produce any compression by itself. The BWT prepares a sequence of input symbols so that additionally used com-

pression methods like run-length-encoding and move-to-front coding in combination with Huffman coding result in good compression performances. The BWT is a general-purpose method (universal coding method) that works on every kind of input data. As discussed in the previous section the problem of the move-to-front method arises when equal symbols are present in the data, but they are not next to each other, i.e. the concentration property of the data is not high. The BWT reorganizes the input sequence so that equal symbols are concentrated in a region, which gives a much higher concentration property and an increased efficiency of the move-to-front method. Run-length encoding can also be used prior to the move-to-front method to gain compression efficiency.

The BWT decoder performs not exactly the reverse operation of the encoder, which means that after knowing how the encoder works it is not immediately evident to see how the decoder works. First, I will explain how the encoder works and then I will show how the decoder executes.

1. The first step of the BWT encoder is to create the permutations of the input sequence with length N. This is done by cyclic shifting the input sequence by one character to get all $N-1$ cyclically shifted variants (permutations) of the input sequence. To illustrate the creation of the permutations a short example is used consisting of the sequence $swiss\_miss$. The cyclic shifts can be seen in table 3.2(a).

2. The second step is to sort the created cyclic shifts and the original sequence, in alphabetic (lexicographic) order. The sorted list is shown in table 3.2(b).

3. The last characters of the sorted cyclic shifts are used to form the output sequence L, which is send to the decoder. The sequence L swm_siisss consists of the same symbols as the original input sequence but in a different order. L is a permutation of the original input sequence with the property that after sorting the cyclic shifts L will have concentrations of equal symbols. Compressing now L instead of the original sequence by using move-to-front eventually in combination with run-length encoding and then Huffman coding can yield in a better efficiency. In addition to L also the index of the position of the original input sequence in the sorted list is send out denoted by $I = 8$ (starting counting from 0). These two information are enough to reconstruct the original sequence.

Table 3.2.: Permutations of the input sequence swiss_miss

(a) Cyclic shift of the sequence

swiss_miss
wiss_misss
iss_misssw
ss_missswi
s_missswis
_missswiss
missswiss_
issswiss_m
ssswiss_mi
sswiss_mis

(b) Sorted cyclic shift

_missswis**s**
iss_misss**w**
issswiss_**m**
missswiss_
s_missswi**s**
ss_missswi
ssswiss_m**i**
sswiss_mi**s**
swiss_mis**s**
wiss_miss**s**

Decoding the received sequence L to obtain the original sorted input sequence works as follows:

1. After the decompression, the sequence L is obtained and the decoder starts creating an additional sequence F by sorting L in alphabetic order. This sequence F corresponds to the first characters of the sorted list in table 3.2(b).

2. Then another vector is created denoted by T, which helps to resort L to get back the original sequence. To get T the two sequences F and L are compared and for each character in F the corresponding character in L is searched and its position number in L is entered in T starting counting by 0.

3. The second received information I is then used to know at which position of T the decoder has to start performing the decoding. The decoding steps are now explained by using a small pseudo code execution for the resulting vectors: I=8; L=$[swm\_siisss]$; F=$[\_iimssssw]$; T=$[3, 5, 6, 2, 0, 4, 7, 8, 9, 1]$.
$k = T[I]$
$D[0] = L[k]$
for j = 1 to N-1
{
$k = T[k]$
$D[j] = L[k]$
}
The decoded sequence is represented by the vector D, which corresponds to the original input sequence.

To get good concentration properties and good compression efficiency the length of the sequences, which are encoded individually, should be some thousand symbols. This shows already the disadvantage of the BWT, which cannot compress the input symbols on the fly and needs large buffers to store a sequence before being able to process it.

Suitability of Burrows-Wheeler transform for the implementation of a detector data compression:

+ Advantageous is that with the Burrows-Wheeler transform the concentration property of the samples can be significantly improved, which makes the move-to-front method and the RLE method suitable for this kind of data. This can work only well if several semi-Gaussian waveforms are grouped together and the Burrows-Wheeler transform is executed on this group.
– Disadvantageous is that the number of waveforms per group should be high to get a high efficiency. This on the other hand requires a lot of memory space in hardware to buffer the group of waveforms and the permutations.
– Disadvantageous is that the Burrows-Wheeler transform will not help a lot if only single semi-Gaussian waveforms are transformed since not a lot of samples in one waveform are likely to be equal.
+ Advantageous is that the realization of the Burrows-Wheeler transform in hardware requires only shift registers to get the permutations and a sorting algorithm.

3.1.4. Tunstall code

The Tunstall code is a variable-length to fixed-length coding method, which encodes a variable number of input symbols with codewords of fixed length. As described in section 3.1.7 prefix codes can be used for variable length codewords to allow the decoder to separate the different codewords without the use of boundary symbols. Variable length codewords are used in a way that frequent symbols are assigned short codewords and not so common symbols are encoded with longer codewords. Depending on the statistics of the input data the average number of bits can be less than by using fixed length codewords, what leads to a compression. The disadvantage is that variable length codewords are difficult to work with especially in hardware implementations, because big buffers are required at the encoder- and the decoder side. The handling of fixed length codewords is much easier and the Tunstall code offers this advantage by still providing the compression performance of variable-length codewords.

Instead of assigning variable-length codewords to the different symbols, the Tunstall code assigns a variable number of symbols to fixed length codewords. The simplest way to explain this method is by showing an example:

The example consists of the already familiar four symbols S, I, M and W ($N = 4$) having the

probabilities 0.6, 0.2, 0.1 and 0.1 respectively. The number of bits for the different Tunstall codes has to be defined and is in this example $n = 3\,\mathrm{bit}$. Then the Tunstall tree can be build starting by the root to which the four symbols are connected as children, as it is shown in figure 3.2(a). Then the child with the highest probability is searched resulting in S and turned into a root of a

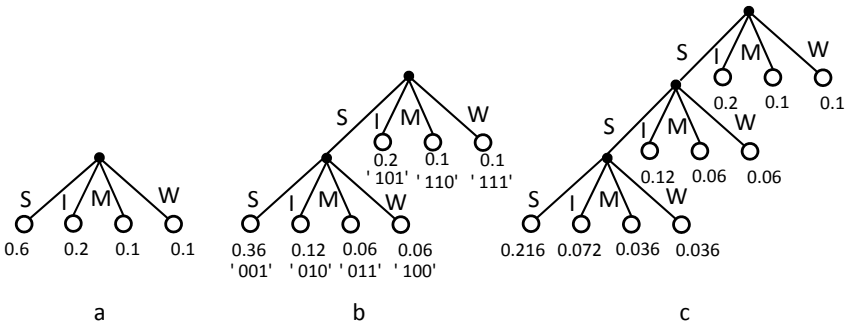


Figure 3.2.: Example of constructing a Tunstall tree

subtree which contains again the four children S, I, M and W. The probability of the root S is divided according to the probabilities of the four children resulting in 0.36 $(0.6 * 0.6)$ for SS, 0.12 $(0.6 * 0.2)$ for SI and 0.06 $(0.6 * 0.1)$ for SM and SW. This step has added $N - 1 = 3$ leaves to the tree by removing the leaf with the highest frequency and adding the N symbols. This action will be executed iteratively as long as the number of leafs of the tree do not exceed the number of possible Tunstall codes defined by n. For this small example $n = 3$ is sufficient which results in maximum $2^n = 8$ leaves and therefore the Tunstall tree in figure 3.2(b) is used containing 7 leaves. In figure 3.2(c) also the next iteration is shown, which results in a tree with 10 leaves and is therefore only relevant for higher values of n. Each leaf of the tree 3.2(b) receives a fixed length codeword of 3 bit. Following the tree from the root to its leaves, defines the symbol sequences corresponding to each Tunstall code. The encoder follows the tree according to the input symbols until it reaches a leaf, sends out the Tunstall code related to this leaf and restarts from the root for the next input symbols. The input sequence $SwissMissSwiss$ is encoded as the following output bit stream: $100, 101, 001, 110, 101, 001, 100, 101, 001$.

This results in 27 bit instead of 28 bit (N=4 symbols need 2 bit/symbol and therefore in a slight compression. In real implementations, the Tunstall tree will be much higher as in this example to get a good compression performance. This on the other hand causes the problem that a high tree will need a large memory in the encoder and the decoder. Therefore, a good compromise between compression performance and memory space has to be found.

Suitability of Tunstall coding for the implementation of a detector data compression:

+ Advantageous is that the Tunstall coding exploits the probabilities of the sample values by producing fixed length codewords.
+ Advantageous is that the fixed length codewords are easier to handle in hardware than variable length codewords. Especially concatenating the fixed length coderwords for sending them sequentially is simple.
– Disadvantageous is that saving the Tunsall codebook requires a large memory space in hardware especially if several levels of the Tunnstal tree are used to achieve a good compression ratio. The codebook consists of the codewords and the corresponding sample values.

3.1.5. Golomb code

The Golomb code is a prefix code, which can be used to encode a run-length encoding (RLE) of consecutive equal symbols in the input stream when the occurrence of the different RLE is not

known in advanced. The Golomb code produces parameterized prefix codes which depend on the probability p and have infinite possible number of codewords.
To explain how the different codewords are calculated a small example will help. In 3.1 a bit stream is shown containing 55 zeros and 15 ones.

$$0000001000011000000000101000000000000111000100001000000100000000100111 \quad (3.1)$$

This leads to the probability of zeros of $p = 55/(55 + 15) = 0.79\%$. The 15 run lengths of consecutive zeros in the bit stream above are 6,4,0,9,1,12,0,0,3,4,6,8,2,0,0. The median of this run length is 3. The median m of a sequence of numbers is defined as the number which halves the sequence in the middle so that about half the numbers are smaller than m and about half the numbers are greater or equal to m. After sorting the run lengths in ascending order, resulting 0,0,0,0,0,1,2,3,4,4,6,6,8,9,12 it can be seen that the number 3 is the median. For example, the run length values could be encoded using Huffman coding but in a system where the number of different appearing RLE cannot be predicted, it can be difficult to guaranty that the Huffman codebook contains codewords for all possibly appearing run length values. Therefore, in such cases the Golomb code can be used, which produces also prefix codewords.

To construct the Golomb code m has to be chosen or calculated or predicted. Then three important parameters have to be calculated as shown in 3.2.

$$q = \left\lfloor \frac{n}{m} \right\rfloor, \qquad r = n - q \times m, \qquad c = \lceil log_2(m) \rceil \quad (3.2)$$

c corresponds to the number of bits needed to represent m, q represents the quotient of the run length number n divided by the median m and r gives the reminder of n/m. The brackets $\lceil\ \rceil$ and $\lfloor\ \rfloor$ indicate that the result is rounded to the nearest integer towards ∞ and towards $-\infty$, respectively. The Golomb code for each run length is calculated in two steps:

First, the quotient q is calculated and encoded in unary code. The unary code represents q as the number of leading ones followed by a zero.

Second, the reminder is calculated and added to the unary code in one of two different ways.

The possible resulting reminders are the integer numbers from 0 to $m - 1$. The first $2^c - m$ values of the reminder are represented in binary with $c - 1$ bit each. The rest is represented as the binary value $r + 2^c - m$ with c bit which assigns the biggest reminder c '1'. This is only the case if m is not a power of 2, otherwise all the reminder values get represented in binary with c bit. The example above results in the following Golomb code for the first run length $n = 6$:
$q = 2,\ r = 0,\ and\ c = 2$ which gives the unary code '110'. The reminder is 0, which is less than $2^c - m = 2^2 - 3 = 1$ and therefore it is represented by $c - 1 = 1\,\mathrm{bit}$ resulting '0'. The Golomb code for $m = 3$ and run length $n = 6$ results from the concatenation of the unary code and the representation of the reminder giving the codeword '110|0'.

For the second run length number $n = 4$ the resulting parameters are $q = 1$, $r = 1$ and $c = 2$. The unary code is '10'. The reminder is equal to $2^2 - 3 = 1$, which means that $c = 2\,\mathrm{bit}$ are used to represent it. The two resulting bits given by $r + 2^c - m = 1 + 1 = 2$ are '10'. The resulting Golomb code for $n = 4$ is therefore '10|10'.

To decode the Golomb codes the decoder has to perform the following steps:

The decoder starts reading the bit stream from the beginning and counts the number of 1's preceding the first 0. This represents the quotient and is named hear with A.

If m is a power of two it corresponds to the simpler case and the decoder just calculates c out of m like in 3.2 and then takes the $c + 1$ bit following the 1's of A. This $c + 1$ bit following A are denoted by R. The encoded number of run length n is then reconstructed by calculating $m \times A + R$. The code length is $A + 1 + c$ and these number of bits are then removed from the bit stream and the decoder starts again counting the 1's preceding the next '0'.

If m is not a power of 2 the decoder starts removing the 1's (giving A) preceding the first '0' which also is removed and then takes the following $c - 1$ bit. This $c - 1$ bit are denoted by R.

Then the decoder checks if $R < 2^c - m$. If that is the case the reconstruction of n is calculated by $m \times A + R$. The length of the code is $A + 1 + (c - 1)$ (the A 1's + the '0' following them + the c-1 bit of R). After the A 1's and the following '0' are already removed from the bit stream, also the $c - 1$ bit have to be removed and the decoder starts again counting 1's.

If $R \geq 2^c - m$ the decoder reconstructs n by calculating $A \times m + R' - (2^c - m)$. R' is the concatenation of the bits of R and the bit following R because the reminder is represented by c bit (length of the code is $A + 1 + c$). After the decoder already removed A and the following '0', it has to remove also the bits of R' from the bit stream before it can restart counting the leading 1's of the next code. To clarify better the steps of the decoder, three short examples are given in the following.

The first example uses $m = 4$ which is a power of 2 and the first part of the incoming bit stream is '11001xxx...'.

The decoder starts by calculating c, which results $c = \lceil log_2(m) \rceil = 2$. Then the leading 1's are counted resulting $A = 2$. The following $c + 1 = 3$ bit are giving $R =$ '001' which equals $R = 1$. The value of the encoded run-length results now in $n = m \times A + R = 4 \times 2 + 1 = 9$. The length of the codeword is $A + c + 1 = 2 + 2 + 1 = 5$ and this number of bits have to be removed from the bit stream before the decoder restarts counting the next leading 1's.

The second example is related to the example used for the encoding process and uses $m = 3$, which is not a power of 2. That gives $c = \lceil log_2(m) \rceil = 2$. The first bits of the encoded bit stream are '11001010xxx...'. The first ones preceding the '0' are giving $A = 2$. Then the decoder removes these two 1's and the following '0' from the bit stream. After that the decoder has to look at the $c - 1 = 1$ bit which build R and result in $R = 0$. The decoder checks whether $R < 2^c - m = 4 - 3 = 1$, which is the case and therefore the decoded run-length value results in $n = m \times A + R = 3 \times 2 + 0 = 6$. The length of the codeword is $A + 1 + (c - 1) = 2 + 1 + (2 - 1) = 4$ and after the A 1's and the following '0' have been already removed from the bit stream, the fourth bit of R the '0' has also to be removed. Then the decoder can restart counting the next leading 1's.

The remaining bit stream '1010xxx...' from the second example is used in the third example by keeping the same median $m = 3$. Again the decoder counts the leading 1's resulting $A = 1$ and removes the $'1'$ and the following '0'. Then it looks at the $c - 1 = 1$ bit following A and see that the bit $R = 1$ is $\geq 2^c - m = 1$. In this case the decoder takes also the next bit following R which is '0' and concatenates it with R to $R' =$ '10' $= 2$. The reconstructed run-length value now results in $n = m \times A + R' - (2^c - m) = 3 \times 1 + 2 - (4 - 3) = 4$. The length of the codeword is $A + 1 + c = 1 + 1 + 2 = 4$ and after the leading 1's and the following '0' have been already removed from the bit stream also the following bits of R' are removed before the decoder restarts counting the next 1's.

The Golomb code is defined by the value of the median m and this value can be calculated in order to get the best prefix codes. The equation for m in 3.3 contains the probability p of the most common bit value in the input bit stream, either '1' or '0'.

$$m = \left\lceil -\frac{log_2(1+p)}{log_2 p} \right\rceil \tag{3.3}$$

Glomb codes result good performances for input data which have a probability distribution of the form $p(i) = q^i(1 - q)$. For the use of this method in real-time applications the problem arises that p is not known in advance and therefore m cannot be calculated. To be able to use the Golomb code in real-time applications an adaptive algorithm can be used that varies m according to the input data which have been encoded so far. In this case the best value of m cannot be used from the beginning, but it will evolve during the encoding. The evolution of m can be performed in changing the value of m just in powers of 2 to facilitate the implementation, although this can further decrease the quality of the adaptive Golomb code.

Another method is to estimate m. The probability p can be estimated using representative data from the data source or use a statistical model.

Suitability of Golomb coding for the implementation of a detector data compression:

+ Advantageous is that the Golomb codes can be used in combination with the Burrows-Wheeler transform and the RLE to get better compression results. With the Golomb codes, the RLE can be easer variable length encoded instead of using entropy encoding methods with predefined code tables.
+ Advantageous is that the implementation is not so difficult requiring one division, one multiplication and some summation or subtraction.
– Disadvantageous is that the compression efficiency can be not so high if only short RLEs are produced. The right choice of the median has to be estimated beforehand by analyzing representative data.

3.1.6. Shannon Fano coding

The Shannon-Fano coding is similar to the Huffman coding described in 3.1.7 and uses the probabilities of the different symbols in a given data set to compress the data set. Symbols that are more often contained in the data set, i.e. have a higher frequency and therefore a higher probability, get assigned short codewords. Sparse symbols with low frequency and low probability are assigned long codewords. The average number of bits per symbol for the entire data set can be calculated and can result in fewer bits as by encoding all symbols with equal length codewords.

The difference between Huffman coding and Shannon-Fano coding consist in the way the optimal codewords for the symbols are found regarding their probabilities. The Shannon-Fano method builds the codewords top to bottom that means from the most significant bit (MSB) to the least significant bit (LSB) whereas the Huffman method builds a tree from bottom up and creates the codewords starting from the LSB.

To create the best variable-size codewords for the symbols the first step is to derive the frequencies of the symbols, which means the amount how often each symbol is contained in the data set. The probabilities of the symbols can be calculated by dividing the frequencies by the total number n of symbols in the data set $p = f/n$. The symbols and their probabilities are then arranged in a list with descending order of the probabilities.

The second step is to divide the list in two subsets in a way that both subsets have the same or almost the same sum of probabilities. Then all symbols in one subset are assigned a bit with value ‘0’. The symbols in the other subset are assigned a bit with value ‘1’.

Next, the second step is repeated by dividing the two subsets into two parts with the same or almost the same sum of probabilities. Again a bit ‘0’ is added to one of the parts concatenated with the bit already assigned in the previous iteration. The symbols in the second part are assigned an additional bit with value ‘1’. This dividing of the subsets is continued for all the subsets until a subset contains only one or two symbols anymore. One of the two symbols is assigned an additional bit ‘0’ and the other an additional bit ‘1’ and therewith the codewords of this two symbols are defined. The described process continues for the other subsets until all symbols have received their complete codewords.

To better clarify the steps described above again a short example of a data set containing four different symbols is shown. The data set contains in total 9 elements $n = 9$ out of this four symbols resulting in the phrase $SwissMiss$. The frequencies of the symbols are given in table 3.3(a) and the probabilities are calculated.

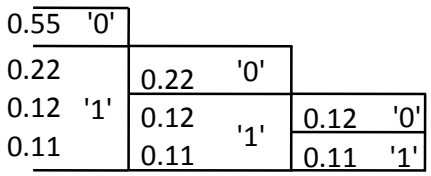
The output bitstream of the encoding of $SwissMiss$ using the Shannon-Fano coding gives the following codewords:
$0, 111, 10, 0, 0, 110, 10, 0, 0$
The output bitstream consists of 15 bit, which is less than by using fixed length codewords. For fixed length codewords 2 bit per symbols are needed to code 4 different symbols, which results in

Table 3.3.: Creating the codewords using Shannon-Fano coding

(a) Symbol-frequencies and probabilities

Symbol	Freq.	Prob. (p)
S	5	0.55
I	2	0.22
M	1	0.12
W	1	0.11

(b) Dividing in subintervals



(c) Codewords of the symbols

Symbol	p	Code
S	0.55	0
I	0.22	10
M	0.12	110
W	0.11	111

18 bit for the phrase $SwissMiss$. The average number of bits per symbols for the Shannon-Fano codes is
$(5 \times 1\,\text{bit} + 2 \times 2\,\text{bit} + 1 \times 3\,\text{bit} + 1 \times 3\,\text{bit})\,/9 = 1.6\,\text{bit/symbol}$.

Suitability of Shannon Fano coding for the implementation of a detector data compression:

+ Advantageous is that the Shannon Fano coding is a entropy coding method which uses the probabilities of the sample values independent in which order the values occur to assign them variable length codewords. A good compression can be expected without the need of a good concentration property.
– Disadvantageous is that the Shannon Fano coding produces the code tree from top to dawn starting with the most significant bit. This can lead to not optimal codewords for the samples. The Huffman coding is preferable because it produces always optimal codewords and therefore has equal or better efficiency.

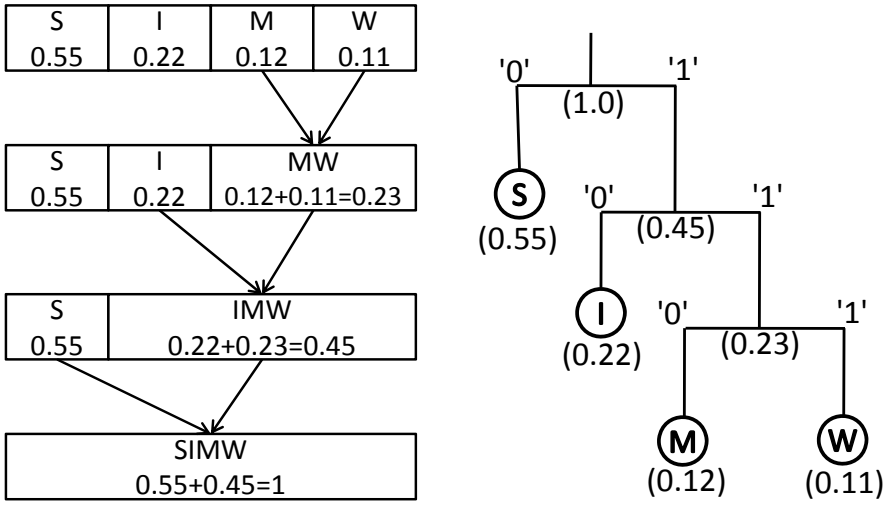
3.1.7. Huffman coding

The Huffman coding method was developed in 1952 by David A. Huffman during his PhD. It is similar to the Shannon- Fano coding described in section 3.1.6 with the difference that in the Huffman coding the codewords are constructed starting from the LSB up to the MSB (i.e. bottom up). Therefore, the Huffman coding method produces in general better codes.

To construct the codewords a Huffman tree is build. Before the tree can be build the frequencies of the different symbols in the data set which has to be compressed, have to be counted and the probabilities have to be calculated. Then the Huffman tree is constructed by writng the different symbols and their probabilities (or frequencies) in a list in descending order of the probabilities. In the next step, the two symbols with the lowest probability values are searched and replaced with a new auxiliary symbol, which is the combination of the two replaced symbols and gets their sum of probabilities. The auxiliary symbol is added to the Huffman tree as the parent node of the two symbols containing them as leaves. Then the step is repeated with the next two symbols in the list, which have the smallest probabilities. The two found symbols and their parent (a new auxiliary symbol) are added to the tree and the tree grows. When the new parent reaches the probability 1, it becomes the root of the tree and the tree is completed. In the list remains only one auxiliary symbols with probability 1, this is the sum of all the probabilities of the different symbols.

A Huffman tree that is built from bottom up can now be used to determine the codewords for each symbol. To use the tree to define the codewords one has to decide that left branches of the tree are adding a bit of value ‘0’ to the codeword and right branches add a ‘1’ to the codeword or vice versa. For getting the full codeword for a symbol, the tree has to be followed from the root to the leaf, which corresponds to the symbol and the bits have to be recoded according which branches ware taken. After all the codewords of the different symbols are defined, the data set can be encoded by replacing the symbols with their Huffman codewords. The probability distribution

of the symbols in the data set defines the quality of the compression. The construction of the Huffman tree and the resulting codewords are shown in figure 3.3 by using the same short example as in 3.1.6 for the Shannon-Fano coding. The usually used phrase $SwissMiss$ throughout this chapter is encoded.



(a) List of Symbols and Probabilities (b) Huffman tree

Figure 3.3.: Construction of the Huffman tree

The resulting bit stream for $SwissMiss$ using the Huffman codewords results in:
$0, 111, 10, 0, 0, 110, 10, 0, 0$
The Huffman tree produces the same results, as by using the Shannon-Fano coding but in some cases the construction of the tree in bottom up manner as Huffman tree leads even to better results. That is the reason that the Huffman coding is much more used than the Shannon-Fano coding. There are some properties of this Huffman tree, which will be discussed in details as follows:

- Huffman codewords are not unique
- Optimality of the Huffman tree
- Huffman codewords are prefix free
- Height of a Huffman tree

If there are more than two symbols or auxiliary symbols in the list with the same probabilities, different combinations can be used to form the new auxiliary symbol, which means that the Huffman tree and therefore the Huffman codewords are not unique. Nevertheless, all possible Huffman trees for the same given probabilities have to result in the same average number of bits per symbol. The different possibilities in constructing the Huffman tree and codewords are relevant only if the encoded data stream has to be sent over a communication line because larger variations in codeword lengths require larger buffers to guaranty a continuous data handling.

The Huffman codewords, which are produced by building a Huffman tree fulfill the properties of a prefix free code. The prefix free codes are codes where no codeword can be the prefix of another codeword. If for example a codeword is ‘01’ no other codeword can start with the bits ‘01xxx’. With this property, each codeword is uniquely distinguishable in a bit stream. This means that the resulting Huffman codewords from the encoder can be joined together to a bit stream without using any boundary symbols like start and stop bits. The decoder starts reading bit by bit from the incoming bit stream and follows the Huffman tree from the root down to the leaves according to the values of the bits. If the decoder reaches a leaf it outputs the symbol which corresponds to this leaf and jumps back to the root before continuing reading the next bits from the bit stream.

The height of the Huffman tree gives the length of the longest Huffman codeword, which is important for the determination of the size of the memory to store the Huffman codewords in

encoder and decoder. Furthermore for designing the buffer, the maximum length of the Huffman codewords is important and to calculate the amount of extra bits which have to be send from the encoder (as side information) to the decoder in order to transmit the used Huffman tree.

Canonical Huffman Codes

Canonical Huffman codes are codes, which are build by a Huffman tree but with special construction rules. They are simple to use and can be of benefit when fast decoding is required. The canonical Huffman codes are also useful in circumstances where the alphabet (number of different symbols) is large.

The first step is to determine the length of the required Huffman codewords for the different symbols. For a large alphabet, the construction of a Huffman tree to determine the length can be not possible because it would require too large memory space. Another possibility to determine the length of the needed Huffman codewords is to use a heap. In this case the required memory space is $2 \times n$ where n is the number of different symbols. For a description of the construction of a heap, see [33].

After the code length for the different symbols is known all possible lengths from 1 to the maximum required length are written in a list as it can be seen in table 3.4(b). In the second row, the number of needed codewords for each length is inserted. The third row contains the integer number at which each set of equal codeword length starts. The canonical codes are built in a way that each set of codewords with equal length starts at a calculated integer value and increments for each following codeword by 1 as it can be seen in table 3.4(a). The start value for each set of

Table 3.4.: Canonical Huffman coding

(a) Canonical Huffman Codes

1:	011	9:	01000
2:	100	10:	000000
3:	101	11:	000001
4:	110	12:	000010
5:	00100	13:	000011
6:	00101	14:	000100
7:	00110	15:	000101
8:	00111	16:	000110

(b) Codeword length an start value

length:	1	2	3	4	5	6
numel:	0	0	4	0	5	7
First:	2	4	3	5	4	0

equal length is calculated in a way that prefix codes are guaranteed. The third row in table 3.4(b) contains the first integer value from which each set of equal codeword length starts. To determine the start value for each set the following equation is used:

$$First_l = \left\lceil \frac{First_{l+1} + numel_{l+1}}{2} \right\rceil \tag{3.4}$$

According to that equation the 5-bit codewords start with integer value 4 and contain all values till 8. The 6-bit codewords start at 0 till 6. The biggest 6-bit codeword with value 6 has the five most significant bits representing the integer value 3. Since the 5-bit codewords start at value 4, they cannot be prefixes of the 6-bit codewords. This is valid for all codewords.

The decoder can now easily identify the length of the codeword by reading bit by bit the incoming data stream. The decoder starts reading the first bit of the data stream and compares the value with the value of $First_1$. If the value is below the value of $First_1$ then the decoder continues reading the second bit and checks if the first two bits denoted by V are smaller than $First_2$. If this is the case, the decoder continues reading the following bits. If the value of the l bits read up

to now denoted by V exceed $First_l$ then the length of the codeword is l and the decoded symbol for this codeword can be found by subtracting the value of $First_l$ from the value of V. The result gives the position in the set of codewords with length l counted from the first codeword in this set. By adding the number of codewords shorter than l it gives the position of the codeword in the whole codebook shown in table 3.4(a) and the symbol is found.

Adaptive Huffman Coding

Normally the frequencies of the different symbols of the data set to be compressed are not known in advanced. The encoder can compress the data in two steps by reading in the data set two times. In the first step, the encoder reads in the symbols of the data set and counts the frequencies of the different symbols. Then the encoder builds the Huffman tree and defines the best Huffman codewords for the different symbols. In the second step, the encoder reads the data set again and encodes each symbol with the corresponding Huffman codeword created in the previous step. This approach is slow and the created Huffman tree has to be added to the output data so that the decoder knows how to decode the compressed data. The approach with the two steps is also not useable if the input data are a data stream, which has to be processed on the fly. Otherwise, a huge memory space has to be used to save the whole data set first. Therefore, often the Huffman coding method is implemented in an adaptive way.

The principle of the adaptive Huffman coding is that the encoder starts with an empty tree. The first symbol, which is read in, is just send out in its uncompressed form. The symbol is added to the Huffman tree and a Huffman codeword is assigned to the symbol. The next symbols are read in and as long as they are not already included in the Huffman tree they are send out uncompressed and added to the tree. If a symbol is already present in the Huffman tree, the corresponding Huffman codeword is send out and the frequency value of the symbols is incremented. The Huffman tree is updated and the next symbol is read in. To distinguish between variable length Huffman codewords and uncompressed symbols a special variable length codeword an escape symbol precedes each uncompressed send symbol. This special codeword has also to be included from the beginning in the Huffman tree and follows the changes when the Huffman tree is updated. The decoder starts as well whit an empty Huffman tree and mirrors the actions of the encoder. The decoder reads in the first word, which is the special codeword (escape symbol) indicating that the following bits are presenting the first uncompressed symbol. The decoder then adds the escape symbol and the uncompressed symbol to the Huffman tree. The decoder continues reading in the bits from the bit stream and following the Huffman three from the top to the bottom and as long as it ends by the escape symbol, it reads the following bits belonging to the uncompressed symbol and adds the symbols to the Huffman tree following the same rules as the encoder. If the decoder ends up at a leaf, which is not the escape symbol, it outputs the symbol corresponding to the received Huffman codeword. The decoder has decoded the received Huffman codeword. The decoder then increases the frequency of the decoded symbol and updates the Huffman tree in the same way as the encoder does it. With this adaptive method of the Huffman coding, some compression performance gets lost especially at the beginning as the symbols are not compressed and the escape symbol has to be added to the output stream. On the other hand, no Huffman tree has to be send to the decoder because the decoder constructs the Huffman tree from the received data during decompression by following the same rules as the encoder.

If the source of the data to be compress, is known in advanced and a probability model can be created so that adequate Huffman codewords can be estimated a Huffman tree can be predefined. The encoder and decoder are preset with this Huffman tree so that already the first symbols of the data set can be encoded properly. The encoder and decoder can then update the Huffman tree during execution to adjust it to the real input data and to increase the compression performance. The frequency values predefined for the different symbols should be as small as possible so that the

actual input symbols at the encoder have a maximum impact on the reorganization of the Huffman tree and the best Huffman codewords are found as fast as possible for the underlying data set.

Suitability of Huffman coding for the implementation of a detector data compression:

+ Advantageous is that the Huffman coding is an entropy coding method which uses the probabilities of the sample values to assign variable length codewords independent of concentration properties of the samples. A good compression of the samples close to their entropy can be expected.
+ Advantageous is that the Huffman coding builds the code tree from bottom up and produces always optimal codeword length for the probabilities of the samples.
+ Advantageous is also that the produced codewords are prefix free which allows them to be concatenated without boundary symbols. The codeword table can be calculated before the encoding of the data by using representative detector data from previous measurements.
+ Advantageous is that the codeword table size is at most as large as the number of different sample values. It is expected to be smaller as for Tunstall coding.
+ Advantageous is that the implementation is easy by using only a memory to store the code table in a way that it is seen as Look-Up table using the sample values as addresses. The corresponding addressed Huffman codeword is reed out and concatenated.
– Disadvantageous is that each sample is assigned a codeword with an integer number of bits, which can produce a compression efficiency not as good as the entropy of the data.
– Disadvantageous is that by wrongly predefining the code table with representative data low compression efficiency can occur. Adaptive Huffman coding could reduce this problem but this increases the implementation effort.

3.1.8. Arithmetic coding

Arithmetic coding is an entropy based coding method, which uses the probability of the different symbols in a given data set to compress this data set. In contrary to the above described Huffman coding, which codes symbol by symbol assigning a codeword to each symbol, the arithmetic coding creates only one output codeword for a sequence of input symbols. Ideally, arithmetic coding creates one output word for the entire data set. Huffman coding is limited to assign integer numbers of bits to each symbol. Therefore with Huffman coding, the theoretical compression limit given by the entropy is reached only if the probabilities of the symbols are negative powers of two. Since arithmetic coding encodes a set of symbols by using a mathematical calculated number it can reach the entropy limit better then Huffman coding.

The method of arithmetic coding bases on the idea of narrowing a defined interval by using the probability values of the incoming symbols. The encoder starts dividing the given interval (e.g. [0,1)) according to the probabilities of the different symbols and assigns each subinterval the corresponding symbol name. Then the subinterval is selected to which the first incoming symbol is assigned and this subinterval is defined as the new interval for the next iteration. The new interval is again divided according to the probabilities of the different symbols and the resulting subintervals are assigned to the corresponding symbol names. The second input symbol selects then the corresponding subinterval, which becomes the new interval for the next step and so on. These steps are executed until all symbols are read from the input stream, which should be encoded together. The output word of the encoder is then any number contained in the final subinterval defined by the last input symbol. To represent a narrower interval more bits are required because a more precise floating-point number has to be used for the upper and lower limit of the subinterval. To achieve a compression the symbols with a high probability are narrowing the interval less as symbols with low probability and therefore fewer bits have to be used to represent the resulting subinterval.

The decoder receives the bit stream representing the number send by the encoder, which is contained in the final subinterval. The decoder too has to know the probabilities of the different symbols and has to divide the defined interval in the same way as the encoder dose it. Then the decoder can search the subinterval in which the received number is contained and the corresponding symbol for this subinterval is the first decoded symbol from the data set. Then the decoder too selects the resulting subinterval as the new interval and divides this again in subintervals according to the probabilities of the symbols. The decoder therefore executes the narrowing of the interval in the same way as the encoder. By repeating these steps, it can decode all symbols from the compressed data set by only using the received number and the probabilities of the different symbols. For a better illustration of this method a short examples is shown as follows:

The example consists of the usual four symbols I, M, S, W forming the sequence $SwissMiss$. The different symbols and their probabilities, as well as the defined ranges are shown in table 3.5. The initial range is chosen to be [0,1) so that all the range limits are below 1 and the floating point

Table 3.5.: Arithmetic Coding example: symbols and their probabilities

Symbol	Probability	Range
I	0.22	[0.00,0.22)
M	0.12	[0.22, 0.34)
S	0.55	[0.34, 0.89)
W	0.11	[0.89, 1.00)

numbers can be represented as integer numbers after the decimal point, skipping the leading 0. The notation [a,b) used above means that the range represents all real numbers between a and b including a but not including b. The notation expresses that the range is closed at the bottom (a) but open at the top (b).

The encoding of the input sequence $SwissMiss$ is described subsequently. The figure 3.4 shows the evolution of the interval which is narrowed according to the symbols in the input sequence and their probabilities. 3.4. The interval [0,1) is divided in the subintervals according to

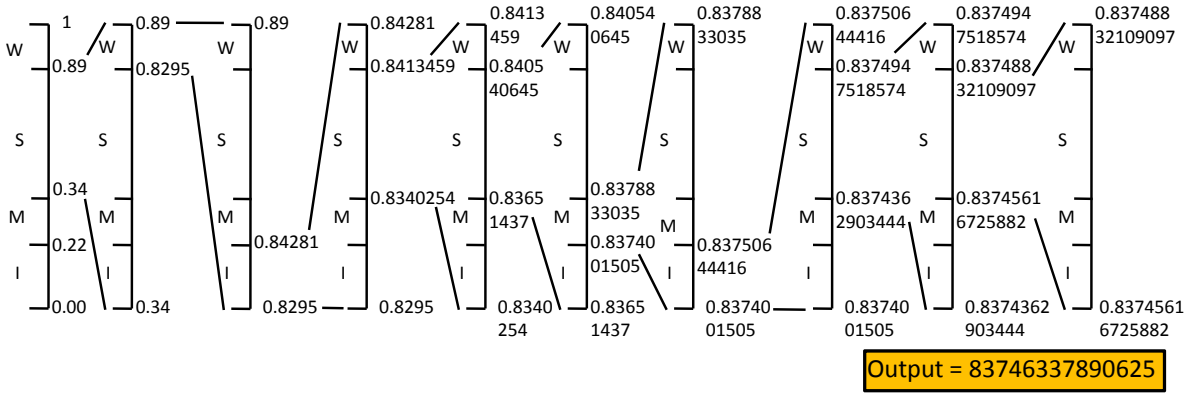


Figure 3.4.: Arithmetic Coding: narrowing of the interval

the probabilities of the different symbols given in table 3.5. The ranges of the intervals can be seen in the table as well as in the figure. Then the first input symbol S is used to select the corresponding subinterval. The boundaries of this selected subinterval become the limits of the new interval used in the next iteration. The new interval is again divided in subintervals according to the percentage of the probabilities. The order of the subintervals is not important but the decoder has to be able to divide the intervals in the same way. Therefore, in this example the subintervals are sorted in

alphabetic order of their symbol names from bottom up. To make the figure better readable only the boundaries of the intervals and the limits of the subinterval for the next symbol are shown. The next input symbol w selects the first subinterval, which is used in the next iteration as the new interval. The floating-point number has to get more precise from iteration to iteration to be able to represent the boundaries of the subintervals, which are narrowed more and more from step to step. The important property of this compression is that symbols with higher probability narrow the intervals less. Therefore, the increase of precision of the floating-point numbers, which have to be used to represent the intervals, is less. When all input symbols are used to narrow the initial interval [0,1) the resulting narrow interval has a range 0.83748832109097-0.83745616725882. To represent the full sequence of input symbols any number contained in this final range can be send to the decoder. To represent the range the minimum number of bits needed is calculated by the $-log_2(intervalsize) = -log_2(0.00003215383215)$ resulting in 15 bit. For this example the chosen floating-point number in the final interval is 0.83746337890625 which is represented by the binary number 0.110101100110010. The encoder send only the binary values after the floating point to the decoder since the leading 0 is obvious because the initial interval has been chosen to be [0,1) and therefore all the floating-point numbers of the subintervals are below 1.

The decoder needs only this number and the table 3.5 to reconstruct the input sequence. The decoder also has to respect the rule of sorting the subintervals from bottom up in alphabetic order what is the definition for this example. The way the decoder reconstructs the input symbols out of the received number is similar to the encoder with the difference that it uses the received number to select the right subintervals and outputs the symbol names corresponding to the selected subintervals. In figure 3.5 the evolution of the interval for the decoding process is shown. The

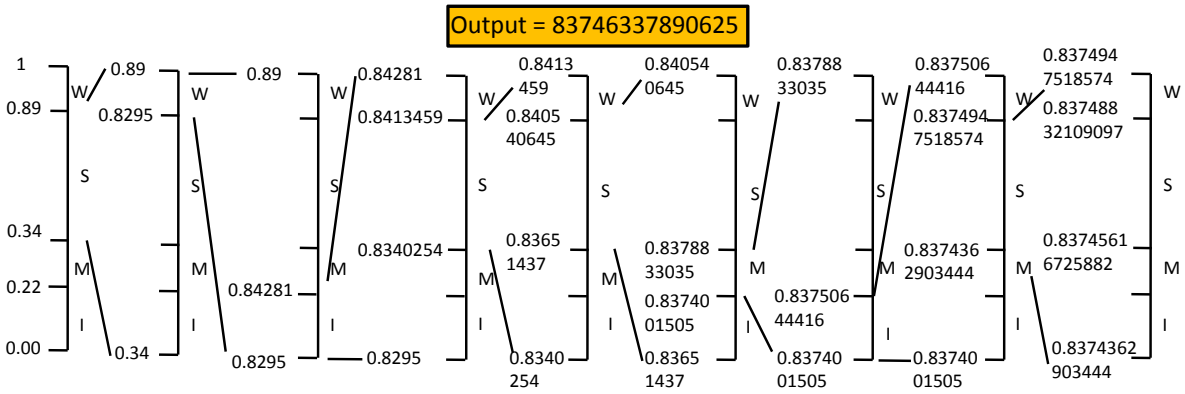


Figure 3.5.: Arithmetic Decoding: narrowing of the interval

decoder divides the interval [0,1) in the same way as the encoder according to the probabilities of the different symbols in the table 3.5. Then the subinterval is search which contains the received number 0.83746337890625 resulting from the received 15 bit. The subinterval related to the symbol S contains the input number and the first reconstruct symbol is S. Then like in the encoder, the found subinterval is used as the new interval, which is again divided into subintervals according to the probabilities. The decoder searches again the subinterval, which contains the received input number. This time the subinterval corresponding to the symbol w is containing the number and the next reconstructed symbol is w. The decoder repeats the described steps until the full sequence $SwissMiss$ of the symbols is reconstructed. The achieved result by arithmetic coding leads to the same number of bits needed to encode the same sequence of symbols as by using Huffman coding. An advantage of the arithmetic coding is that its complexity is independent of the length of the sequence to encode which makes it possible to encode much longer sequences as the one used in the short example and therefore the efficiency can get better as by using Huffman coding. The mean number of bits per symbol for arithmetic coding will approach the entropy when the number

of symbols in the sequence to encode grows towards infinity.

For a real implementation of the arithmetic coding algorithm, one problem arises that the precision of the floating-point number can get very high but the representation of this floating point values is always limited to the number of bits used to represent them by the implementation. One property of the arithmetic coding which can help to solve this problem in a real implementation is the fact that if the first digits after the decimal point of the upper limit and the lower limit of the range become equal than they will never change again in the next iterations. In this case, the digits, which are equal, can be shifted out of the register holding the upper and lower limit. To shift out the equal digits the upper limit gets filled up on the other end with 9's and the lower limit gets shifted in 0's. The exact procedure is explained in the following using the previous example.

The two variables for the lower and higher limit of the interval are called $IntHigh$ and $IntLow$ and the variables for the resulting subinterval, which becomes the new interval are $NewHigh$ and $NewLow$. The length of these variables is defined to be a 4 digits integer which need 14 bit to be represented. The initial interval is [0,1) which leads to the initial values of $IntHigh = 9999$ and $IntLow = 0000$ assuming that both values represent the digits after the decimal point. The representation in this case of the upper limit 1 with 0.9999 is allowed, since there is now real value, which could be contained between 1 and 0.9999 if all numbers are limited to 4 digits.

The first step is to calculate the limits of the subinterval corresponding to the first input symbol, which becomes the new interval by using the following formula:

$$NewHigh = IntLow + Range \times HighRange(x) - 1 \quad (3.5)$$

$$NewLow = IntLow + Range \times LowRange(x) \quad (3.6)$$

$$Range = IntHigh - IntLow + 1 \quad (3.7)$$

The variables $HighRange$ and $LowRange$ represent the upper and lower limit of the ranges of the different symbols according to their probabilities which are given in table 3.5. The first input symbol is S which leads to the following results:

$$\begin{aligned} Range &= 9999 - 0 + 1 = 10000 \\ NewHigh &= 0 + 10000 \times 0.89 - 1 = 8899 \\ NewLow &= 0 + 10000 \times 0.34 = 3400 \end{aligned}$$

The next input symbol is w and gain the formulas 3.5-3.7 are used to calculate the limits for the subinterval corresponding to it. The values for $IntHigh$ and $IntLow$ are now the previous calculated values of $NewHigh = 8899$ and $NewLow = 3400$. The next values of $NewHigh$ and $NewLow$ are resulting as follows:

$$\begin{aligned} Range &= 8899 - 3400 + 1 = 5500 \\ NewHigh &= 3400 + 5500 \times 1 - 1 = 8899 \\ NewLow &= 3400 + 5500 \times 0.89 = 8295 \end{aligned}$$

From the results it can be seen that the first digit from the left has the same value 8 in both variable $NewHigh$ and $NewLow$. Therefore, this value 8 will not change anymore in the following step of arithmetic encoding as it can be seen also in figure 3.4. Since the value will not change anymore it will be contained also in the final output value of the encoder and therefore one can already add this value to the output stream and shift it out of the two variables $NewHigh$ and $NewLow$ to prevent the problem of limited representation in real applications. The value 8 is add to the output variable $ArithmCOut = 8$ and the digits in the two variables $NewHigh$ and $NewLow$ are shifted to the left by shifting in from the right a 9 for the $NewHigh$ and a 0 for the $NewLow$. The shifted values for the variables $NewHigh$ and $NewLow$ are resulting in the values $IntHigh = 8999$ and

$IntLow = 2950$ and are used in the next iteration for the input symbol I.

$$\begin{aligned} Range &= 8999 - 2950 + 1 = 6050 \\ NewHigh &= 2950 + 6050 \times 0.22 - 1 = 4280 \\ NewLow &= 2950 + 6050 \times 0 = 2950 \end{aligned}$$

The results show that this time the first digit of $NewHigh$ and $NewLow$ is not equal and therefore no shift of the variables will be done and no additional value is add to the output stream. For the next step the variables $IntHigh$ and $IntLow$ get the values of $NewHigh = 4280$ and $NewLow = 2950$ without any change.

A problem can arise if the values of the two variables $NewHigh$ and $NewLow$ are getting close to each other, but the first digits differ in each step. This can cause the problem that the limited precision of the two variables is reached by the variables and the algorithm above continues forever producing an incorrect result. To prevent this problem a rescaling of the variables should be done as sun as there is the indication that they can get to close to each other. If the content of the register is seen as binary numbers as it is the case in real implementations then rescaling will be done if the most significant bits are getting '10' for $NewHigh$ and '01' for $NewLow$. The first bit of $NewHigh$ and $NewLow$ is shifted out and the second most significant bits are inverted. A counter variable $count = 1$ is set to indicate that one rescaling has been performed. If in the next step, the most significant bits of the two variables are again '10' for $NewHigh$ and '01' for $NewLow$ another rescaling is done and the counter variable is incremented to $count = 2$. Several rescaling steps might be executed until the first bits of the two variables $NewHigh$ and $NewLow$ becomes equal and can be shifted out. Before adding the equal most significant bit to the output as many bits with the inverted value of the most significant bit are added to the output $ArithmCOut$ as the value $count$ results followed by the most significant bit itself. [36]

Adaptive arithmetic coding

Adaptive arithmetic coding can be used in applications where it is not possible to construct the probability model of the different symbols before the encoder starts encoding. In this case, the encoder starts with an empty model where all the symbols have the same probability or with a probability model for the known source. Like in adaptive Huffman coding the encoder creates the correct probabilities of the symbols while encoding the input data. The important step where the arithmetic coder needs the probabilities of the symbols is when calculating the boundaries for the subinterval $NewHigh$ and $NewLow$ of the actual input symbol like described above. However, in this step, it is not important in which order the different symbols are and which actual probability values they have as long as the encoder and the decoder use at each step the same table.

The adaptive arithmetic encoder reads in the first symbol and encodes it using the equal probabilities or the probability model. Then the frequency value of the actual symbol is incremented and the probability table is updated and resorted if needed. The decoder reads in the codeword, decodes it using the actual table and then adds the resulting decoded symbol to the table by incrementing the corresponding frequency value and if necessary resorting the table. Both the encoder and decoder have to resort the symbols in the same way. The important rule is that the encoder has to update the probability table only after encoding the input symbol. This is important because the decoder has now knowledge about the actual symbol, before it decodes the received codeword and only than the decoder can update the probability table accordingly. After each input symbol, the probability table is updated and it evolves closer to the ideal model for the underlying data set. The implementation of the arithmetic coding is the same for adaptive and not adaptive coding apart from the updating of the probability table.

Suitability of arithmetic coding for the implementation of a detector data compression:

+ Advantageous is that the arithmetic coding is an entropy coding method, which uses the probabilities of the sample values. A good compression of the samples close to their entropy can be expected.
+ Advantageous is that the arithmetic coding encodes several samples at once which reduces the problem of assigning integer number of bits and therefore it can produce compression performances very close to the entropy even better than Huffman coding.
+ Advantageous is also that the probability model can be defined using representative detector data from previous measurements.
– Disadvantageous is that the realization of arithmetic coding requires implementing some arithmetic operations as multipliers, dividers, adders and subtractors. This makes the implementation more complex as Huffman coding.
– Disadvantageous is that by wrongly predefined the probability model from representative data low compression efficiencies can occur. Adaptive arithmetic coding could reduce this problem but this increases even more the implementation afford by updating continuously the probability model.

3.1.9. PPM

The abbreviation PPM stands for "Prediction with Partial string Matching". The PPM method bases on an arithmetic coding but the way it assigns the different symbols of the alphabet is different. The arithmetic coding calculates the frequencies of the different symbols based on the number each symbol appears in the input data. Either, this is done by analyzing the full data set before the compression step or by counting the frequencies of the different symbols by the time they are encoded and decoded.

In the PPM method, not the frequency of each symbol is used but the probability of its appearance, taking into account the preceding symbols arrived at the encoder. This method is a statistical compression method, which uses prediction by looking at a context instead of the frequency of the symbols. This method tries to predict the next input symbols by looking at a defined number of already received symbols. The encoder reads the actual input symbol and looks how often in the past the symbol already appeared preceded by a string of N characters, which is called the order-N context C. This determines the probability of the input symbol regarding C. The probability is then used to perform an adaptive arithmetic coding as described in section 3.1.8. By looking for example at typical English texts the probability of encountering a h is about 5% but the probability of h is much high (about 30%) if it is known that the previous input symbol was t, since the digram th is common in English. By predicting the symbols, a better compression can be achieved with the drawback of a higher complexity in constructing a probability model. To illustrate better the concept of PPM a short example is given using English texts. Let us assume that the string the has been seen 100 times so far in the input text. From this 100 times it has been followed 42 times by the space character $\sqcup$, 21 times by r, 16 times by s, 14 times by n, 5 times by m and 2 times by i.

The probabilities of the different letters following the context $C = the$ are summarized in the table 3.6. If for example the next input symbol is r then the probability 21% of r following the order-3 context the is given to the adaptive arithmetic encoder to encode this symbol. Then the table is updated by increasing the count of the followed by r and the probabilities are recalculated.

If for example the next input symbol would have been a which is not contained in the table, because the context the followed by the character a has been not yet seen in the input text. This would mean that the probability of a according to the order-3 context would be 0% but the arithmetic encoder cannot handle a symbol with 0%. The solution of the PPM encoder to this is to shorten the context and to see if the shorter string appeared already in the input data. The order-3 context is shortened to the order-2 context he. Again the encoder searches if the context he has been seen already followed by the character a. As it can be seen in table 3.6 the probability of a

Table 3.6.: Order-3 and Order-2 context of the as example of English text and there counts and probabilities

Order-3 context	Counts	Probability	Order-2 context	Counts	Probability
the $\rightarrow$ ␣	42	42%	he $\rightarrow$ ␣	64	32%
the $\rightarrow r$	21	21%	he $\rightarrow r$	40	20%
the $\rightarrow s$	16	16%	he $\rightarrow a$	32	16%
the $\rightarrow n$	14	14%	he $\rightarrow s$	20	10%
the $\rightarrow m$	5	5%	he $\rightarrow n$	16	8%
the $\rightarrow i$	2	2%	he $\rightarrow e$	12	6%
			he $\rightarrow l$	8	4%
			he $\rightarrow m$	6	3%
			he $\rightarrow i$	2	1%

following he is 16%. This can now be encoded by the arithmetic encoder.

What, if the actual input symbol is the character x. Even the order-2 context has been not seen so far followed by the character x. The encoder reduces again the context to order-1 and searches if e has been seen already followed by x. If this is the case the probability of symbol x following e is used for the arithmetic coding.

Otherwise, the PPM encoder reduces again the context order to 0, which means it searches if the character x itself already appeared in the input data without taking into account any context. If x already appeared in the input, the probability is calculated out of the frequency of x divided by the total amount of received symbols. This corresponds to the normal approach of arithmetic coding by using the frequencies of the symbols.

At the end it can also happen that the input symbol x is seen the first time in the input data. In this case the PPM encoder reduces the order to -1 and sends a probability of $1/alphabet$ for the symbol to the arithmetic coder, where the alphabet contains all possible different symbols which can appear in the input data.

One aspect, which has to be discussed, is the way the PPM encoder tells the decoder when to reduce the order of the context. The decoder has no knowledge of the next symbol appearing in the input data, before it decodes the compressed symbol. Therefore, the decoder cannot predict that for the next symbol it has to reduce the context and use another column in the table containing the right probability model. To solve this problem the encoder has to add an escape symbol to the encoded data. The escape symbol is a special character what is not part of the normal alphabet of the input data. The escape symbol has to be included in the probability table by both the encoder and the decoder. The way the escape symbol is added to the probability model is shown on behalf of the example $swissMiss$ already used in the previous discussed methods.

The order-2, order-1 and order-0 context of the string $swissMiss$ are shown in the table 3.7. One way to add the escape symbol is to extend each context by this escape character (@). In the table, each group of the contexts is shown separately with the additional escape symbol. The probability of the escape symbol depends on the number of different characters following the corresponding context. For example the context $C = is$ has been seen twice in the past and both times it has been followed by the character s. Therefore, in this group the symbol s receives a probability of 2/3 and the escape symbol @ has the probability 1/3. On the other hand the context ss has been seen also two times, but one time it was followed by the character m and the second time by the ␣ character. The escape symbols @ gets the count two in this group because two different combinations are contained in this context. The idea of assigning the probabilities of the escape symbol in this way is; If a context is seen several times and it is always followed by the same character the probability is high that the next time the context is seen it is again followed by the same character. Therefore, the encoder should not have to send often the escape symbol for

Table 3.7.: Context counts order 2, 1, 0 and probabilities of $swissMiss$ as example of including the escape character

Order-2	Counts	Prob.	Order-1	Counts	Prob.	Order-0	Counts	Prob.
sw $\rightarrow i$	1	1/2	s $\rightarrow w$	1	1/9	s	5	5/15
@	1	1/2	s $\rightarrow s$	2	2/9	w	1	1/15
			s $\rightarrow M$	1	1/9	i	2	2/15
wi $\rightarrow s$	1	1/2	s $\rightarrow \sqcup$	1	1/6	M	1	1/15
@	1	1/2	@	4	4/9	$\sqcup$	1	1/15
						@	5	5/15
is $\rightarrow s$	2	2/3	w $\rightarrow i$	1	1/2			
@	1	1/3	@	1	1/2			
ss $\rightarrow M$	1	1/4	i $\rightarrow s$	2	2/3			
ss $\rightarrow \sqcup$	1	1/4	@	1	1/3			
@	2	2/4						
			M $\rightarrow i$	1	1/2			
sM $\rightarrow i$	1	1/2	@	1	1/2			
@	1	1/2						
Mi $\rightarrow s$	1	1/2						
@	1	1/2						

this context and therefore the probability can be low.

On the other hand if a context is seen several times but is always followed by a different character the chance is high that the next time this context is followed again by a new character. The encoder has to use the escape symbol @ to switch to a lower order context. Since the escape symbol in this case is used more frequently, it gets a higher probability to narrow the interval of the arithmetic coder not too much causing many bits to represent the narrow interval. The probability of the escape symbol is also updated continuously as input symbols arrive.

There are also other ways to determine the probability assigned to the escape symbol. The one described above is known as PPMC.

Another method is called PPMA, which assigns the escape symbol in a group of context always the count 1. If the total number of counts in the group is n not including the count of the escape symbol, then the escape symbol gets the probability of $1/(n+1)$. The rest of the symbols in the group get their original probability of x/n, where x is the count value of the corresponding symbol following the context of the group.

A third method has the name PPMB and is similar to PPMC with the distinction that it assigns a probability to a symbol, which follows the corresponding context only after the symbol is seen twice. The way this is realized is that the probability of each symbol is calculates as $(x-1)/n$. If the count of the symbol following the context C is $x = 1$, then this symbol has probability $((1-1)/n = 0)$. In this case, the escape symbol is used. The reason for this constrained is that a symbol which is seen twice following the context is more reliable to happen again in future, the "believe" in the prediction is strong.

Another method called PPMP (P stands for Poisson) assigns each symbol a probability under the assumption that the appearance of the symbol follows a Poisson distribution. The expectation value (average) for the Poisson distribution is retrieved from the already received symbols. By using the Poisson distribution, it is tried to simulate the probability of the symbol for the entire input data (including future data) based on the already processed input data. With this method the

expected number of new symbols following the context C can be calculated by 3.8, where t_1 is the number of symbols that appeared only once so far, t_2 is the number of symbols that appeared twice and so on.

$$P = t_1\frac{1}{n} - t_2\frac{1}{n^2} + t_3\frac{1}{n^3} - ... \tag{3.8}$$

This expression represents the probability that the next symbol is a new symbol, which means this probability is assigned to the escape character.
A version of the PPMP method is the PPMX (the X stands for approximate) method which uses only the first term $\frac{t_1}{n}$ of the expression 3.8. If $t_1 = 0$, this method world assign a probability of 0 to the escape symbol which cannot be handled by the arithmetic coding and therefore it is modified to PPMXC which in this case uses the same method as described for PPMC.

One improvement of the PPM method can be achieved by exclusion. The probabilities of the different symbols in a context after a shift down from a higher order context can be increased by excluding the symbols already analyzed in the higher order context. If we look at the table 3.7 and assume that the next input symbol is w and the context is actually the order-2 ss the encoder cannot find ss followed by w. Therefore, the encoder switches down to order-1 context. In the order-1 context s the encoder finds the case of s followed by w which has the probability 1/8. However, the fact that first the encoder search the w in order-2 tells that the two cases of order-2 ss followed by $\llcorner$ and M can be excluded in order-1, otherwise the encoder would have already found the next symbol in order-2. That means that the order-1 case of s can be reduced to three possibilities and therefore the next symbol w which is found in order-1 following s gets the probability 1/6. With this exclusion the probabilities increase after a shift down of the context order and therefore the arithmetic coder narrows the interval less which leads to less bits and a higher compression. The disadvantage of the exclusion is that it increases the complexity due to the symbols, which have to be remembered to be excluded and the probabilities, which have to be recalculated more often.

Suitability of PPM coding for the implementation of a detector data compression:

+ Advantageous is that the PPM method is used in combination with arithmetic coding and uses the probabilities of the samples for a good compression.
+ Advantageous is that with the PPM method not only the probabilities of the samples can be used but also their correlations according to the defined waveform shape. This can lead to better compression efficiencies as arithmetic coding itself even better than the entropy of the data.
– Disadvantageous is that the constructing of the probability models is much more complicated as for arithmetic coding alone. This increases significantly the complexity by requiring much more memory space as in arithmetic coding for the various possible probability models and requires updating continuously the model.
– Disadvantageous is that with the complex probability models a wrongly predefinition from representative data is much likelier as for the simple probability model of arithmetic coding. This can lead to bad compression performances.
– Disadvantageous is also the additional required escape symbol and the difficult handling of the escape symbol.

3.1.10. Lempel-Ziv

The compression method LZ77 was developed from the two researchers Abraham Lempel and Jacob Ziv in the 1970s. This compression method bases not on statistical properties of the input data as previous mentioned methods, but is a dictionary based method. The performance of the statistical compression methods is related to the accuracy of a statistical model representing the probability distribution of the symbols in the input data. A dictionary based compression method

does not look at the probabilities of the symbols; it just selects strings of symbols like words in text documents and encodes them using a token, which tells their position in a dictionary. Therefore, this dictionary based compression methods are called universal coding methods because they work on all kind of data independent of any probability model. The universal compression methods from Lempel and Ziv are the most popular used methods for file compression and are included e.g. in compression methods like compress, zip, gzip, png, gif. The way a dictionary-based encoder works is by using a dictionary in which a set of symbol-combinations (strings) are saved and sequences of input symbols are compared to the entries in the dictionary. If a corresponding string is found in the dictionary, the encoder adds the index to the output data. The index tells the decoder where to find the encoded string in the dictionary. Therefore, the decoder in a dictionary-based method has not to perform the exact opposite of the encoder like in statistical methods, but only has to use the index to find the right string in the dictionary. The dictionary-based decoder is much simpler as the encoder because it has not to divide the input bit stream in strings and has not to find a match of the strings in the dictionary.

A simple example of this method is to use the English dictionary and searching the words of an English text in this dictionary. If a word is found, the index of it in the dictionary is added to the output stream, otherwise the word itself is written on the output.

In addition, they use strings and not single symbols what makes them often more efficient than statistical methods. The statement that compressing strings yield better efficiency than compressing single symbols, can be explained as follows:

In statistical methods like Huffman coding described in 3.1.7 the efficiency of the compression increases the more the probabilities of the different symbols differ from each other. The probability of strings differ more and more the longer the strings gets. Short strings are more likely to appear more often in the input data as longer strings. The prove that the probabilities of strings varies more than the probabilities of the single symbols can be given by the short example shown in the table 3.8 and table 3.9.

Table 3.8.: Probabilities and Huffman Codes for a Two-Symbol Alphabet $P_{a_1} = 0.8$ and $P_{a_2} = 0.2$

String	Probability	Code	String	Probability	Code
a_1a_1	$0.8 \times 0.8 = 0.64$	0	$a_1a_1a_1$	$0.8 \times 0.8 \times 0.8 = 0.512$	0
a_1a_2	$0.8 \times 0.2 = 0.16$	11	$a_1a_1a_2$	$0.8 \times 0.8 \times 0.2 = 0.128$	100
a_2a_1	$0.2 \times 0.8 = 0.16$	100	$a_1a_2a_1$	$0.8 \times 0.2 \times 0.8 = 0.128$	101
a_2a_2	$0.2 \times 0.2 = 0.04$	101	$a_1a_2a_2$	$0.8 \times 0.2 \times 0.2 = 0.032$	11100
			$a_2a_1a_1$	$0.2 \times 0.8 \times 0.8 = 0.128$	110
			$a_2a_1a_2$	$0.2 \times 0.8 \times 0.2 = 0.032$	11101
			$a_2a_2a_1$	$0.2 \times 0.2 \times 0.8 = 0.032$	11110
			$a_2a_2a_2$	$0.2 \times 0.2 \times 0.2 = 0.008$	11111

Table 3.9.: Probability variance and average code size

String size	Variance of Probability	Average Code size
1 Symbol	0.6	1
2 Symbols	0.78	0.78
3 Symbols	0.792	0.728

The resulting variances of the probabilities for the three cases of single symbols, strings of two symbols and strings of three symbols are shown in the table 3.9 and the strings result higher values.

The variance of the probabilities reflects the deviation of the individual probability values from the average value. The higher the variance the more the probability values varies from the average value. The calculation of the variance of the probabilities is given in 3.9.

$$v = \sum |p_i - m| = |0.64 - 0.25| + |0.16 - 0.25| + |0.16 - 0.25| + |0.04 - 0.25| = 0.78 \quad (3.9)$$

In the equation 3.9, m represents the mean value of the probabilities and p_i are the probability values of the symbols or strings.

If one would use variable length codewords like Huffman coding the strings with their higher variance world yield in a better codeword length distribution and a smaller average number of bits per string than by encoding the individual symbols.

The big disadvantage of the dictionary based compression methods using strings instead of individual symbols is that the number of different strings (different symbol combinations) is much higher, than the number of different single symbols which requires a huge dictionary size. To store all possible strings a large memory space is needed for the dictionary and the search for a matching string in the dictionary is slowed down because of the large number of entries to check.

A possible solution to this problem is to use a sliding window as dictionary and not creating the full dictionary of all possible strings as it is used in the LZ77 compression method.

LZ77

The method developed by Lempel and Ziv in 1977 called LZ77 uses a part of the already received input stream as the dictionary to encode the next input symbol. The encoder shifts the received input symbols in a buffer from the right, which is called the window. The window is divided in two parts; the part to the left is the dictionary, which is followed by the part to the right, which is the lock-ahead buffer. The dictionary part contains the last already encoded input symbols and the look-ahead buffer is filled with the next input symbols, which have to be compressed. The encoder now looks if it can find the first leftmost character of the look-ahead buffer inside the dictionary part of the window, scanning the dictionary from right to left. If a match of a number of characters (string) larger than 2 is found somewhere in the dictionary the encoder sends out a token containing three entries;

The first number represents the offset (number of characters) from the end of the dictionary until the first character of the found match.

The second value of the token is the length of the found matched string, the number of matching characters.

The third entry of the token is the next symbol in the look-ahead buffer following the string, which has been found in the dictionary.

This third entry of the token is important in the case where no match can be found in the dictionary. In this case, the encoder sends out a token, which has an offset value and a length value of 0 and as the third entry the character which couldn't be found in the dictionary. This is especially the case at the beginning of the encoding, where the dictionary contains no or only a few symbols. If the encoder finds more than one match in the dictionary, the one, which contains the most characters, is used. In the case where more equally long matches are found, the last one is used, so that the encoder does not need to keep track of the found match, which simplifies the algorithm.

After the encoder encoded one matching string of the look-ahead buffer, the window has to be shifted to the right for the length of the encoded string. The look-ahead buffer is filled up with new characters to encode. The actually encoded characters are shifted into the dictionary part and the leftmost characters, the oldest ones are shifted out of the dictionary at the left side. This described compression method with the sliding window is illustrated in a small example to understand the algorithm better.

Before, one aspect of the LZ77 has to be discussed that is the choice of the length of the dictionary and the look-ahead buffer. The longer the dictionary is, the more characters are contained and the higher is the chance to find matches for the new input symbols. A long dictionary on the other hand creates the need of more bits to represent the offset value. These bits, which have to be used for the offset, are contained in every token created by the encoder.

The length of the look-ahead buffer defines the maximum length of a matching string, which can be found in the dictionary. Even if the matched string in the dictionary would be longer, the encoder could not see this, because it cannot see the new input symbols, which are not already shifted into the look-ahead buffer. A too short look-ahead buffer would split up long matching strings in different tokens and therefore decrease the compression efficiency. Typically, in text compression the length of the dictionary is a few thousand characters, which results in 10 bit-12 bit for the offset value. The length of the look-ahead buffer is only a few tenths or hundreds of characters long, needing a few bits (4 bit-8 bit) for the length value in the tokens.

An example of the LZ77 compression method is shown below containing the sliding window with the two parts, the dictionary (left) and the look-ahead buffer (right).

Bilbo's favorite, was young | Frodo Baggins. When Bilbo was_ninety- | nine he adopted Frodo | as

The encoder looks at the first character, the leftmost in the look-ahead buffer, n and searches this in the dictionary. The encoder finds four n in the dictionary and looks now which of these are followed by the second character in the look-ahead buffer, i. The second n in the dictionary from right, which has the offset value 7, is followed by i. The encoder continues to investigate how many characters following the ni match as well in the dictionary. The next two characters ne following ni also match in the dictionary, so the string $nine$ is found in the dictionary with an offset of 7. The encoder adds the token (7,4, ␣) to the output data, and shifts the window five characters to the right. The third entry of the token is the space character ␣, which follows the matching string $nine$ in the look-ahead buffer. After the shift of the window of five characters is the first word in the look-ahead buffer he and the string $nine$␣ is shifted into the dictionary, while $Frodo$ is shifted out of the dictionary on the left side.

This shows another deficiency of the LZ77 method. When the encoder starts looking at the word $Frodo$ in the look-ahead buffer this word is already shifted out of the dictionary and the encoder will not find a match anymore in the dictionary. This word $Frodo$ is encoded by using several tokens instead of one single token. The LZ77 assumes that pattern in the text occur close together which fails if words are common in the text but uniformly distributed. To overcome this, common words should be store in a separate buffer.

The decoder of LZ77 is much simpler; it has not to execute a search operation. It only needs a buffer to store the dictionary. The decoder reads in a token and uses the offset to point to the first character in the dictionary. Then it outputs as many characters as the second value of the token tells, starting from the one it points too. At the end, it adds the character, which is contained as the third entry in the token. Then the string, which is just send out by the decoder, is also shifted into the dictionary buffer from the right to update the dictionary for the next token.

A variant of the LZ77 called LZSS that was developed by Storer and Szymanski in 1982. This algorithm improves the LZ77 in tree aspects:

1. It uses a circular queue for the look-ahead buffer.
2. To fasten up the search in the dictionary, a binary tree is used to organize the dictionary.
3. The tokens are reduced to two entries instead of three.

A circular queue is a basic data structure, which uses two pointers to point to the start and the end of the string, which is searched. These two pointers are moved instead of shifting all the characters in the buffer. If a string in the look-ahead buffer is extended to find a longer match in the dictionary only the end pointer is moved.

A binary tree is a tree where the left children B of a parent node A is smaller than A, and the right children C is greater than A (B<A<C). For text compression, smaller and greater strings are defined lexicographically, that means the string, which precedes another one in the alphabetic order, is smaller. This binary tree organization of the dictionary fastens up the search step of LZ77. The height of a binary tree defines the maximum steps to find a string in the dictionary. The height of a binary tree is given by $log_2(n)$, where n is the number of elements in the tree.

The third difference of LZSS to LZ77 is that the tokens contain only two values, the offset value and the length value of the matched strings. The third entry, the character following the matched string, which is used to encode also non-matching characters, is not used in LZSS. Instead of using the third element in LZSS the characters which are not found in the dictionary are send out one by one with their original uncompressed codes and are distinguished to the tokens by a preceding flag bit. That means that all words in the output stream contain a preceding flag bit, which tells if the word is part of a token or an uncompressed character. This gives two outputs token: for found matches the token ('1',o,l) is send, for a non matching character the token ('0',c) is send.

LZ78

Another variant of dictionary based compression method was published again by Lempel and Ziv in 1978 with the name LZ78. This method does not use a sliding window, but a dictionary, which contains previously seen strings. The size of the memory space limits the dictionary but it has to be sufficiently large because no entries are ever deleted from the dictionary. The encoder searches the input string in the dictionary and if it finds the string then a token is added to the output stream containing two words. The first word of the token is a pointer (address) which gives the position of the matched string in the dictionary. The second word represents the not matching character, which follows the matched string in the input data. Then the encoder saves the matched string followed by the following input character as a new string in the next free position of the dictionary. The tokens do not contain the length of the strings since this is already defined by the entry in the dictionary. At the beginning, the encoder starts with an empty dictionary containing only a null token at address 0. The first token send by the encoder is this null token together with the first input character C_1 $(0, C_1)$. The C_1 character is then added to the dictionary at position 1. The next character C_2 is searched in the dictionary and since there is only C_1 contained yet C_2 is compared to C_1. If the two characters are not the same the encoder uses again the null token $(0, C_2)$, and writes C_2 in the dictionary at position 2.

If the two characters C_1 and C_2 are equal the encoder reads in the next character C_3 from the input stream and concatenates it with C_2 to form the string C_2C_3. It searches the string in the dictionary, but there is only C_1 no further match is found. The encoder then sends out the token $(1, C_3)$, where 1 is the pointer to C_1 since C_2 and C_1 are matching. The character C_3 is added as well to the token since it is the next symbol which caused the search to fail. Then the encoder adds the new string C_2C_3 to the dictionary at position 2. For a better understanding the dictionary entries and the tokens are shown in table 3.10 for the example string "$When\ Bilbo\ was\_ninety - nine$". As it can be seen in the table, the first 10 characters of the phrase are simply added to the dictionary and the null tokens are used to send them out. This show how fast the dictionary can grow if no matches are found. In these cases, the method creates expansion rater then compression because to each input character the null token has to be added which means that they are represented by two words instead of one.

After the first ten individual character the following space character $\sqcup$ has been seen already before and is found in the dictionary at position 5. The encoder concatenates the next character w to the space character to form the string $\sqcup w$. This string is not found anywhere in the dictionary, so the encoder creates the token $(5, w)$ and adds the string $\sqcup w$ to the dictionary at position 11. An expansion is also in this case most likely because the pointer value will need more bits than a

Table 3.10.: LZ78 example for the string "$When\ Bilbo\ was\_ninety - nine$"

Dictionary		Token	Dictionary		Token	Dictionary		Token
Pointer	Entry	Pointer+C	Pointer	Entry	Pointer+C	Pointer	Entry	Pointer+C
0	null		8	l	$(0, l)$	16	ne	$(4, e)$
1	W	$(0, W)$	9	b	$(0, b)$	17	t	$(0, t)$
2	h	$(0, h)$	10	o	$(0, o)$	18	y	$(0, y)$
3	e	$(0, e)$	11	$\llcorner w$	$(5, w)$	19	$-$	$(0, -)$
4	n	$(0, n)$	12	a	$(0, a)$	20	nin	$(15, n)$
5	$\llcorner$	$(0, \llcorner)$	13	s	$(0, s)$			
6	B	$(0, B)$	14	_	$(0, \_)$			
7	i	$(0, i)$	15	ni	$(4, i)$			

character to permit a large dictionary for more matches in later steps.

At the end the string ni is found in the table at position 15. The encoder creates the token $(15, n)$ and adds the string nin to the dictionary. Now a compression can be achieved since tree characters are represented by one token. The more input symbols are read, the more matches can be found and the longer are the strings in the dictionary, which increases the efficiency of the compression. On the other hand the more the dictionary grows the more memory space is needed and the search for matching strings becomes slower. A good way to organize the dictionary and speed up the search process is to use a tree structure.

A binary tree like the one described in LZSS would not be the best choice, therefore a tree structure called $trie$ is used in LZ78 [33]. The tree starts with the null token as the root. Then all the individual characters requesting the null token are added as children to the root. If strings of two characters are build the second character is add as a child of the first character which already is the child of the null token. In the next level, the last characters of 3-character strings are added at their parents, which are the second character and so on. The trie for the example in table 3.10 is shown in figure 3.6.

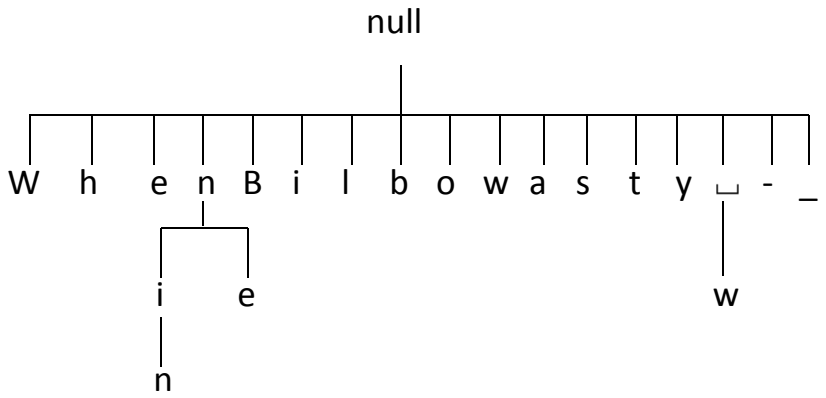


Figure 3.6.: Trie for LZ78 example "$When\ Bilbo\ was\_ninety - nine$"

One problem of LZ78 arises when the memory space of the dictionary is full. The simplest solution is to keep the dictionary unchanged until the end of the compression, but this can reduce the performance of the compression quite a lot.

Another possibility is to delete the entire dictionary and restart from the beginning with the assumption that further symbols benefit more from the new data than from the old one. This is the case if the data can be divided in blocks, which are uncorrelated. The third solution is to delete the least used entries in the dictionary, but this creates the problem of how to track the use of the

entries. Unfortunately, there is no good, simple algorithm to find the right entries to delete.

The decoder of LZ78 has to mirror the steps of the encoder and build up the same dictionary, which makes it more complex than the decoder for LZ77 does.

One frequently use variant of the LZ78 method was developed by Terry Welch in 1984 and named LZW. This method uses tokens, which contain just the pointer value. The first entries of the dictionary are initialized to the different characters of the alphabet. In this way, the encoder will always find a match of the next character in the dictionary and the tokens contain no following character but only the pointer to the matched strings or single characters. The encoder concatenates the incoming characters to a string I as long as it finds the string in the dictionary. If the next added character C_n causes the string IC_n not to be found anywhere in the dictionary the encoder sends out the token with the pointer to the position of the last match of I. The string IC_n is then added to the dictionary and the encoder starts concatenating the next input characters to C_n to form a new string I.

The LZW decoder reads the token containing the pointer to the string I. It extracts the string I from the dictionary and sends it out. Then the string I is stored until the next token arrives. The next pointer points to the next string J and the decoder extracts J from the dictionary. Then it sends out J and stores the string I followed by the first character of J in the dictionary as a new entry. The first character of J is C_n which caused the encoder to fail the search of IC_n.

Suitability of Lempel-Ziv coding for the implementation of a detector data compression:

+ Advantageous is that the Lempel-Ziv coding is a dictionary based method which requires no probability model. The problem of correctly building a probability model for the data to be compressed as for entropy coding methods is not necessary for Lempel-Ziv coding.
+ Advantageous is that neighboring channels of detectors often register waveforms with similar amplitudes, which can increase the length of matching sample values found in the dictionary.
– Disadvantageous is that the amplitude of the waveforms can vary in a large range, which can make it difficult to find long matching sequences of consecutive samples in the dictionary. The possible variation of consecutive sample values is quite large.
– Disadvantageous is that for a good compression performance it is important that the dictionary can store several combinations of sample values, which requires a large memory space in hardware.
– Disadvantageous is that the search for matches in the dictionary costs time, which makes it problematic for a real-time implementation.

3.2. Lossy compression

In addition to the lossless compression methods described in section 3.1, several lossy compression methods are presented now in this section. The lossy compression methods try not only to reduce the redundancy of the input data, but they also reduce the data by rejecting irrelevant information. In lossy compression, the properties of the expected data, which have to be compressed, are evaluated and a decision is taken which information is indispensable for the application and has to be retained and which part of the data can be rejected by accepting some reduction of accuracy of the reconstructed data (error). The irrelevant information is information, which is not important to the user of the data. For example in image compression, mostly the user is a person, which looks at the image. The sensitivity of human eyes is limited. Therefore, some loss of information in digital image representation might be tolerated, because human eyes cannot recognize the decreased quality of the image caused by this information loss.

The rejection of parts of the data by the compression results in an irrevocable loss of information, but this yields in general in higher compression efficiency as with lossless compression

methods could be achieved. The difficulty is to find methods, which are capable to separate the important information of the input data from the unimportant (irrelevant) one in a way that a high compression rate is achieved by maintaining a tolerable loss of data.

In the following, the most important different lossy compression methods are presented.

3.2.1. Scalar and Vector Quantization

The by far best known quantization method is the **scalar quantization** used for example to digitize analogue signals. The representation of a time-continuous analogue signal with a time-discrete digital signal by sampling the analogue signal in defined time intervals can be seen also as lossy data reduction. The analogue signal, which contains information in every point in time, is quantized to a digital signal, which retains the information in only certain (discrete) points in time. This reduces the amount of data by losing the analogue information between two discrete points in time.

The scalar quantization can also be used for example in image compression. A simple way to compress an image which consists for example of 8 bit pixels can be quantized to 4 bit by cutting of the least significant four bits. This results in a fixed compression ratio of 1/2, which is known in advanced. The drawback is that the possible 256 different grayscale or color values are reduced to just 16 different values. This creates a high distortion and can cause bands of different colors with sharp transitions, which are annoying to the user. For example, a row of grayscale pixels can consist of the following 12 values.

$$\begin{aligned} Row_1 &= 215,\ 214,\ 213,\ 211,\ 210,\ 209,\ 207,\ 206,\ 205,\ 204,\ 203,\ 202 \\ Row_{bin} &= 11010111,\ 11010110,\ 11010101,\ 11010011,\ 11010010,\ 11010001,\ ...\, 11001010 \end{aligned}$$

After the quantization only the 4 MSBs of each pixel are retained which yield in the first 6 pixel being represented by $1101_b =$ and the last 6 pixels being represented by 1100_b.

The decoder adds to each of the 4 bit input words 4 LSBs with value 0. This results in 12 pixel value where the first six have value $1101000_b = 208$ and the last six have value $11000000_b = 192$. From a row where the pixel values change smoothly after the quantization, the row has a sharp change in the middle from 208 to 192. If the following rows result similar behavior the image is divided in two grayscale bands with a sharp transition in between.

A variant of the scalar quantization, which tries to reduce the effect, is called Improved Grayscale Quantization (IGS). The principle of this method is to reduce the pixel representation to 4 bit not by just cutting the least significant bits, but by adding randomness depending on the neighboring pixels.

The first pixel is still quantized by just cutting the 4 LSBs. For the next pixels, the encoder calculates an intermediate 8 bit value R by summing the 8 bit of the actual pixel P and the 4 LSBs of R (at the beginning R is initialized to 0). The 4 MSBs of R are then added to the output stream. One special case is when P has a value $1111xxxx_b$, then P is just copied in R. The table 3.11 shows the encoding of the last 6 pixels of the previous example.

Often scalar quantization is used in lossy compression in combination with other data transforming methods or predictive methods discussed later in this section. The input data e.g. a block of an image are first transformed in another domain by one of the described transform methods in section 3.2.2 and then the resulting transform coefficients are scalar quantized to achieve compression.

Another well-known quantization method is the **vector quantization**. The vector quantization do not quantizes each input value separately as the scalar quantization does, but groups a defined number of input values to an input vector and compares each input vector with reference vectors stored in a codebook. Vector quantization is also used in lossy images compression. An image is

Table 3.11.: IGS method on 6 grayscale pixels

Pixel	Value	R	Compressed output
1	1100 1111	0000 0000	1100
2	1100 1110	1100 1110	1100
3	1100 1101	1101 1011	1101
4	1100 1100	1101 0111	1101
5	1100 1011	1101 0010	1101
6	1100 1010	1100 1100	1100

divided in blocks of for example 4×4 or 8×8 pixels and each block forms an input vector. The vector quantizer has several reference vectors stored in a codebook and each input vector is compared to these reference vectors to find the best matching one. Then only the index of the best matching reference vector is added to the output data.

The decoder has just to read the incoming indices from the compressed stream and use them to address the corresponding reference vectors in the codebook. Then the indexed reference vector, which represents an uncompressed block of pixels, is sent out. While the encoder has to perform a comparison between the input vectors and the reference vectors, the decoder has only to use the received index to readout the corresponding reference vector from the codebook. The decoder is much simpler because it does not mirror the steps of the encoder and therefore vector quantization is an asymmetric compression method. The compression efficiency is known in advanced and given by $I/(block\ size)$. The variable I is the number of bits used to represented all the indices of the codebook and it is divided by the bits contained in each image block ($block\ size$) which represents an input vector.

If for example each block contains 2×2 pixel values, then each input vector consist of 4 pixels. If each pixel is represented by only one bit like in a bi-level image (normally a black and white image) then the block size is 4 bit and the possible number of different input vectors is $2^4 = 16$. If the codebook now contains all the 16 possible combinations of a block of pixels as the reference vectors, the encoder will find the reference vector, which exactly matches the input vector. In this case, the decoder will find the reference vector and reconstruct perfectly the input vector with no information loss. On the other hand since the codebook contains all the possible combinations of the input vectors which are 16 the index to select each of the 16 reference vectors has also to be represented by 4 bit, which then yield in no compression $I = 4;\ pixels/block = 4;\ bits/pixel = 1;\ I/(block\ size) = 4\,\text{bit}/4\,\text{bit} = 1$. Vector quantization used in compression makes only sense if not all the possible reference vectors are stored in the codebook and therefore the index can be represented with less bits as the input vectors. This is also more realistic since for example by using a block size of 2×2 pixels and each pixel is represented by 8 bit like in grayscale images the number of different reference vectors world already result in $2^{2\times2\times8} \approx 4.3\,billion$. The reduced set of reference vectors can cause that an input vector is not exactly identical to a reference vector in the codebook. Then, the encoder selects the best matching reference vector to represent the input vector. This causes an error in the reconstruction of the image, since the real input vectors are not all identical to the used reference vectors, which can be seen also as loss of information (i.e. lossy compression). A intuitive example of vector quantization used in image compression is shown in figure 3.7.

The input vectors (image blocks) containing 4 pixels and are compared with the reference vectors in the codebook to find the best index value. The decompressed blocks are shown on the bottom of the figure. One can see that not all pixels are reconstructed with the correct original colors but the overall impression is quite similar.

One aspect of the vector quantizer, which has still to be discussed, is the way that the encoder

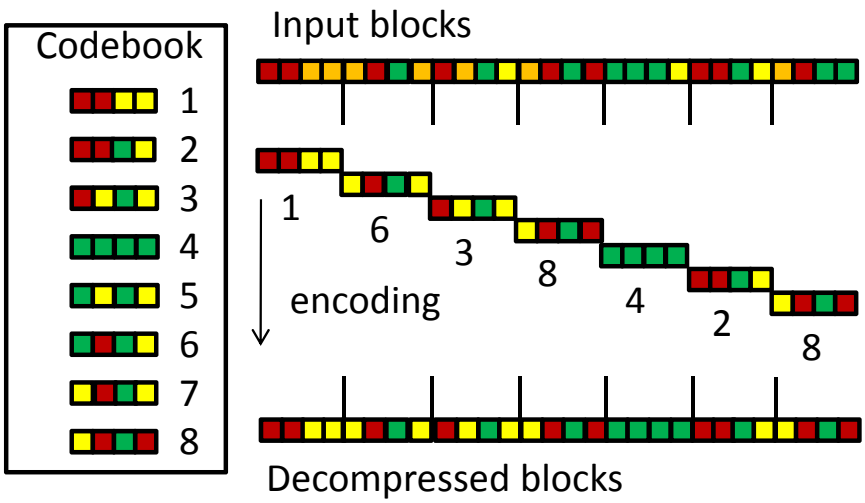


Figure 3.7.: Intuitive vector quantization of an image block

finds the best matching reference vector in the codebook. Three measures are presented in the following, which can be used to find the best matching reference vector:

Let $B = (b_1, b_2, ..., b_n)$ denote the input vector and $C = (c_1, c_2, ..., c_n)$ be a reference vector. The first measure is the so-called "distance" d between the input vector B and a reference vector C. The distance is defined as:

$$d_1(B, C) = \sum_{i=1}^{n} |b_i - c_i| \tag{3.10}$$

The distance d(B,C) for $n = 3$ represents the distance between the two three-dimensional vectors B and C when moving along the coordinates axes. The second measure is the Euclidean (straight line) distance defined as:

$$d_2(B, C) = \sum_{i=1}^{n} (b_i - c_i)^2 \tag{3.11}$$

The third measure gives the maximum difference between the components of the two vectors B and C and is defined as:

$$d_3(B, C) = \max_{i=1}^{n} |b_i - c_i| \tag{3.12}$$

One of these measures can now be used to classify the degree of similarity between the input vector and a reference vector. This comparison has to be calculated for all the reference vectors in the codebook and this for each input vector. If the codebook contains k reference vectors, then this means that the encoder has to calculate one of the distance measures above k times for each input vector. The reference vector, which results in a minimum of the used measure, is then selected and the corresponding index is added to the output data.

A variant of the vector quantization is called tree-structured vector quantization (TSVQ), and this method tries to fasten up the search process for the right reference vector. This method will find the right reference vector in $log_2 k$ steps by using the individual bits of the pixels in the input vector to locate the different block areas in space and to exclude some reference vectors in this way. The more the area is confined the more reference vectors can be excluded. The search algorithm takes bit per bit of the pixels starting by the MSB and uses them to find the corresponding path through a tree at which leafs the reference vectors are listed. For example when each image block consists of two 8 bit pixels this input vector can be seen as a point (x,y) in a two dimensional space. The two dimensional space can be divided in four regions numbered with the bits '00', '01', '10', '11'. The most significant bit of both pixels in the input vector is then used to determine in which region

the input vector is present. Then this region is again split into four sub-regions with the numbering '00', '01', '10', '11' and the second MSBs its of the two pixel-values are used to allocate the input vector in one of these four sub-regions. This is continued for all the bit pairs of the input vector and the sub-region resulting at the end corresponds to a reference vector in the codebook. This works with every number of pixels per input vector, but for orders higher than three it cannot be visualized anymore in space. [33]

A key function of the vector quantization is the way the codebook is constructed, which has to ensure a good compression by maintaining a small distortion in the reconstruction of the image. Yoseph Linde, Andrés Buzo and Robert M. Gray developed an algorithm (called LBG), which is often used in vector quantization for image and audio compression. The algorithm starts with an initial codebook and then modifies this codebook to adapt it to the actual image. The codebook has to be included in the compressed output data but since it is optimized to the image, the compression ratio can be significant. The first step is to select a threshold ϵ which defines when the codebook is optimized enough to use it for the encoding.

The encoder starts to divide the image in the different blocks B_i. Then it groups the blocks by finding all blocks B_m which are closer to a reference vector C_i than to any other reference vector. All found blocks B_m which are closest to C_i are forming the group P_i. The boundaries which surround each group P_i and are at half way between the reference vector belonging to P_i and the neighboring reference vectors are giving the Voronoi region for P_i. Each input vector positioned inside the Voronoi region is closest to the reference vector belonging to the corresponding Voronoi region.

After all blocks are partitioned into groups P_i, the encoder calculates the distortion D_i for all groups P_i which are not empty. The average of the distances calculated between each block B_m in the group P_i and the reference vector C_i related to this group gives the distortion D_i. The average distortion D is then calculated out of all D_i.

The distortion for the actual iteration k denoted by D^k and the distortion of the previous iteration D^{k-1} are used to decide if another iteration is needed by solving equation 3.13.

$$\left(D^{k-1} - D^k\right) / D^k \leq \epsilon \tag{3.13}$$

If the equation results true, no further iteration is needed and the actual codebook is used to perform the vector quantization on the image. The actual codebook has to be included in the output data.

If a further iteration is needed then the encoder updates the codebook by calculating the average of all the input vectors B_m in one group P_i and replaces the reference vector C_i by this average vector. After this, the blocks B_i of the image are partitioned again into the groups P_i and new partitions can occur after the reference vectors are updated. The average distortion is again calculated and should now be smaller than the one from the previous iteration $k-1$. The equation 3.13 is used again to decide if a further iteration has to be performed. An example of this algorithm is shown in [33].

Suitability of Quantization for the implementation of a detector data compression:

+ Advantageous is that the scalar quantization can be used in all lossy compression methods for a fast and easy reduction of the range or representation of resulting numbers.
+ Advantageous in terms of vector quantization is that samples of one waveform can be quantized together. A good compression performance can be achieved by using small codebook sizes.
+ Advantageous is that the realization in hardware is simple requiring a memory as look-up table for the reference vectors and some summations and comparisons for finding the best matching reference vector to the input waveform (e.g. Manhattan distance).
+ Advantageous is also that the size of the output words is fixed and known in advance depending on the size of the codebook. If the input waveform would have a fixed number of

samples, the compression efficiency is also known in advance.
- Disadvantageous is that the introduced distortion by using small codebook sizes is high since only a few reference vectors are available to represent the input waveforms.
- Disadvantageous is also that for a reasonably low distortion a large codebook has to be used with reference vectors representing waveforms with different amplitudes and number of samples. This increases the area of the implementation requiring large memories.
- Disadvantageous is that if the input waveforms have variable numbers of samples an alignment step has to be performed to align the reference vectors and input waveforms that increases complexity and requires additional information (about the alignment) to be transferred.

3.2.2. Transforms

A transform used for data compression is a mathematical calculation like Fourier transform, which is executed by grouping the input data into matrices or vectors and multiplying those with a transform matrix to weight the input symbols.

The idea to use a transform method in data compression is that the transform should decorrelate the symbols in the input data and concentrate the energy in the first resulting transform coefficients. The energy of a vector or matrix is defined as the sum of the squares of the elements $a^2 + b^2 + c^2 + d^2 + ...$ If the energy is concentrated in the first coefficients, they hold the important information and have large values, whereas the following coefficients have small and not so important values. Compression can be achieved by heavily quantizing the small unimportant coefficients and lightly quantizing the important coefficients or even not changing them.

The transform coefficients can also be seen as the frequency components of the input data, where the frequency represents the information of how strong elements of the input data are changing between them. The first transform coefficient represents the DC component with frequency 0, whereas the following coefficients correspond to AC components of the input data with increasing frequencies. The concept of frequencies of the input data can be illustrated by figure 3.8.

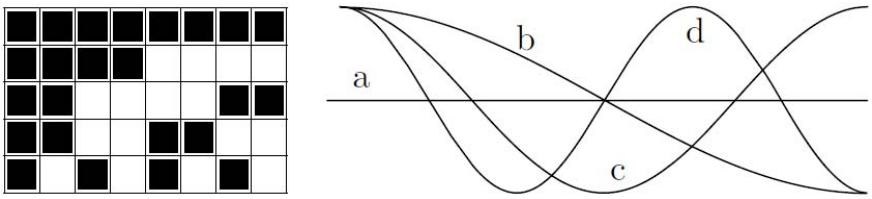
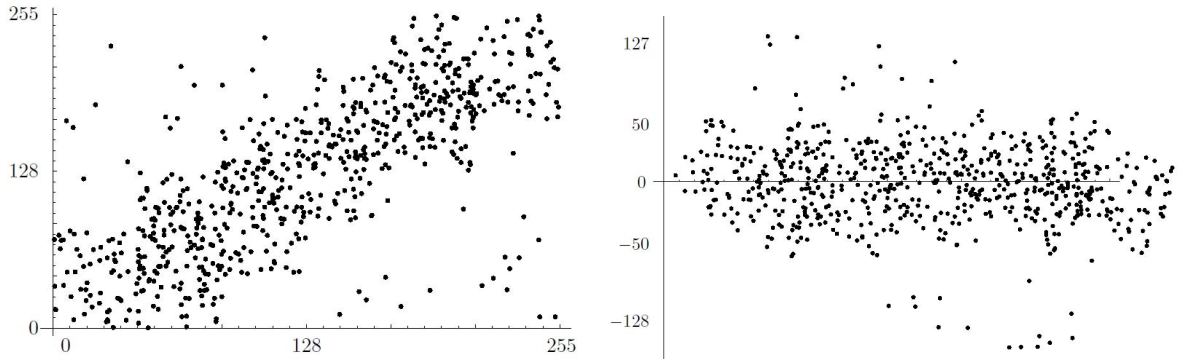


Figure 3.8.: Bi-level image frequencies [33]

The pixels in the first row are uniform, they do not change and therefore they have frequency "a" = 0 shown as a straight horizontal line in figure 3.8. The second row changes one time the pixel values and therefore the frequency is low as shown by line "b", which crosses one time the 0-line. The pixels of the next row are changing twice the color and therefore the frequency is higher and the line "c" crosses twice the 0-line. The frequency gets continuously higher for the last two rows.

To visualize the use of a transform for compression an example is presented based on 128×128 grayscale image with one byte per pixel. This image is known as the Lena image in the data compression community and is one of several test images to analyze and compare compression methods. The image is grouped in vectors of two pixels each. These input vectors can be shown as dots in a 2-diemnsianl space (x, y), where all the odd-numbered pixels are representing the x coordinates and the even numbered pixel are the y coordinates. A natural image has the property that adjacent pixels have similar or equal values and therefore the most points in the 2-dimentional representation are lying on the $45°$-line $x = y$ as it can be seen in figure 3.9(a). This is representing

the correlation between the pixels. With a transform all points can now be rotated $45°$ clockwise so



(a) 2-pixel input vectors in 2D (b) Rotated input vectors

Figure 3.9.: 2D representation of the input vectors containing two pixels of the grayscale Lena image [33]

that the most points are coming to lie on the x-axis as it can be seen in figure 3.9(b). The rotation is done mathematically by multiplying each input vector with the rotation matrix R as given in equation 3.14.

$$(x^*, y^*) = (x, y)\, R = (x, y) \begin{pmatrix} \cos 45° & -\sin 45° \\ \sin 45° & \cos 45° \end{pmatrix} = (x, y)\, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \tag{3.14}$$

Most points have after the rotation a y-component concentrated around $y = 0$. The values of the x-component are slightly increased after the transform. Therefore, the energies of the y-components are transferred to the x-components. The reduction in correlation between the pixels before and after the transform can be retrieved by comparing the cross-correlation. The cross-correlation is defined as the sum $\sum_{i=1}^{n} x_i y_i$ of the input vectors (x_i, y_i). The resulting rotated (x^*, y^*) can be quantized separately and are representing the compressed data. [33]

The decoder reconstructs the input vectors by taking the received transform coefficients (x^*, y^*) and multiplying them with the inverse of the transform matrix as given in equation 3.15.

$$(x, y) = (x^*, y^*)\, R^{-1} = (x^*, y^*)\, R^T = (x^*, y^*)\, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \tag{3.15}$$

The transform matrix R is not the only possible transform matrix which can be used in lossy compression. In the following different transform methods are discussed and some properties of a good transform matrix are presented.

A transform matrix which can lead to good lossy compression should:

1. Reduce the redundancy of the input data by decorrelating the elements.
2. Identify the less important components of the input data to quantize them strongly.

To achieve the first aim the elements of the transform matrix, the weights w_{ij} should be positive and negative. To ensure that the first transform coefficient c_1 becomes the most information and the other coefficients are getting small, the first row of the transform matrix w_{1j} should have only positive elements. The other rows of the transform matrix should have equal numbers of positive and negative weights, so the other coefficients become much smaller than the first one.

The second aim is to separate the important and the unimportant components of the input data. Therefore, the resulting transform coefficients should be related to different frequency-components of the input data. The first coefficient should represent the DC component with frequency 0 and the following coefficients should represent components related to increasing frequencies. To achieve this relation the weights of the transform matrix should change their sign according to the frequency components. The first row w_{1j} is related to the DC component and therefore no sign-change is present, all elements are positive. The second row w_{2j} should have only one sign change, that

can be achieved by selecting the first half elements of the row as positive values and the second half as negative values. The third row of the transform matrix should have two sign changes, the fourth row three and so on. A possible transform matrix showing this properties is given in 3.16. This matrix is orthogonal, which means that the inverse of the matrix and the transpose are equal $W^{-1} = W^T$. This is important to guarantee that the transform can be reversed perfectly resulting in the original input matrix if no quantization would be performed on the coefficients.

$$W = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} \tag{3.16}$$

Other possible transform methods are described in the following:

Walsh-Hadamard Transform

The transform method of Joseph L. Walsh and Jacques S. Hadamard (WHT) is fast and easy to implement because it uses only additions, subtractions and right shifts. Unfortunately, the compression efficiency is not very high, which is the reason why it is not used frequently. The WHT is a generalized Fourier transform. The transformation and inverse transformation is given by equation 3.17 and equation 3.18, respectively.

$$\begin{aligned} H(u,v) &= \sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)g(x,y,u,v) \\ H(u,v) &= \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)(-1)^{\sum_{i=0}^{n-1}[b_i(x)p_i(u)+b_i(y)p_i(v)]} \end{aligned} \tag{3.17}$$

$$\begin{aligned} f(x,y) &= \sum_{u=0}^{N-1}\sum_{v=0}^{N-1} H(u,v)h(x,y,u,v) \\ f(x,y) &= \frac{1}{N}\sum_{u=0}^{N-1}\sum_{v=0}^{N-1} H(u,v)(-1)^{\sum_{i=0}^{n-1}[b_i(x)p_i(u)+b_i(y)p_i(v)]} \end{aligned} \tag{3.18}$$

The variables b_i represent the single bits of the binary representation of the integers u, v, x and y. The term p_i is defined by the bits of the indices u and v as following:

$$\begin{aligned} p_0(u) &= b_{n-1}(u), \\ p_1(u) &= b_{n-1}(u) + b_{n-2}(u), \\ p_2(u) &= b_{n-2}(u) + b_{n-3}(u), \\ &\vdots \\ p_{n-1}(u) &= b_1(u) + b_0(u) \end{aligned} \tag{3.19}$$

The results of the transform, $H\,(u,v)$ are the coefficients of the WHT. The functions $g(x,y,u,v)$ and $h(x,y,u,v)$ are equal and represent the weighting matrix with elements of $+1$ and -1 and the matrix is divided by $\sqrt{N}$. The value of N represents the row and column length of the input data block $N \times N$, and N must be a power of 2. Since $N = 2^n$ is a power of two the division for $N > 2$ can be performed by a right-shift of $n - 1$ positions.

For each coefficient, the elements of an input block are multiplied by $+1$ or -1 and all resulting values of each input block are summed up and divided by $\sqrt{N}$ to normalize them. Then the

resulting coefficients can be quantized and add to the compressed stream. A resulting Walsh-Hadamard transform matrix for N=2 and N=4 is shown in 3.20 and as well the manner to construct Walsh-Hadamard matrixes for higher N. The normalization of the transform matrix by $\sqrt{N}$ is needed that the magnitude of the row vectors becomes 1 so that an orthonormal matrix is the result.

$$H_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad H_{2N} = \begin{pmatrix} H_N & H_N \\ H_N & -H_N \end{pmatrix} \quad H_4 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{3.20}$$

Suitability of Walsh-Hadamard transform for the implementation of a detector data compression:

+ Advantageous is that the WHT is executed fast by multiplying an input vector with the transform matrix. A real-time implementation is possible.
+ Advantageous is also that the WHT is easy to implement requiring only a few additions subtractions and shift operations.
– Disadvantageous is that the lossy compression efficiency is low compared to other transform methods. The decorrelation property of the WHT is not too good.

Karhunen-Loève Transform

The Karhunen-Loève transform creates a transform matrix out of the data set, which has to be compressed. This optimal transform matrix is then used in a second stage to transform the data set for compression. This method has the best efficiency in terms of energy compaction and decorrelation because it is optimized to the underlying input data. The drawback is that the calculated transform matrix has to be added to the compressed data in order that the decoder is able to perform the inverse transform to reconstruct the original data. The second problem is that the calculation of the transform matrix is not trivial and no fast implementation is developed so fare, which makes this method less than ideal for practical implementations.

To calculate the transform matrix the input data set is first grouped to input vectors of a defined length. In image compression a input vector $b^{(i)}$ contains the pixels of one block. The image to compress can for example be divided into k blocks, each of 8×8 pixels, which results in $n = 64$ pixels per input vector and k such input vectors.

Then the average vector $\overline{b}$ of the k input vectors $b^{(i)}$ with length n can be calculated as $\overline{b} = \sum_{i=1}^{k} b^{(i)}/k$.

A new set of vectors $v^{(i)}$ is calculated, which has an average of 0 by subtracting the average vector $\overline{b}$ from each input vector by $v^{(i)} = b^{(i)} - \overline{b}$. This new vectors are combined to the matrix V having n rows and k columns.

The searched transform matrix is denoted by A. This matrix (A) is multiplied by V to form the weighting matrix $W = A \cdot V$. The rows of W represent the n coefficient vectors $c^{(j)},\ j = 1, 2, ..., n$.

The matrix product $W \cdot W^T$ represents in their diagonal elements the variances of the coefficient vectors. The off-diagonal elements are the covariances of the coefficients. To decorrelate the coefficients, in the Karhunen-Loève transform the covariances have to become 0. That means that the matrix product $W \cdot W^T$ has to result in a diagonal matrix with all the diagonal elements being the eigenvalues $(\lambda_1, \lambda_2, ..., \lambda_n)$ of $V \times V^T$ and the off-diagonal elements have to be 0. This leads to the equation 3.21 from which the elements of the transform matrix A can be calculated for the

given input data (image).

$$W \cdot W^T = (AV) \cdot (AV)^T = A(V \cdot V^T)A^T = \begin{pmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ 0 & 0 & \lambda_3 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \lambda_n \end{pmatrix} \tag{3.21}$$

The product matrix $V \cdot V^T$ is symmetric and its eigenvectors are orthogonal. Therefore, the rows of A have to become normalized, which means that their magnitude has to result 1 and then the matrix become a orthonormal matrix.

Suitability of Karhunen-Loève transform for the implementation of a detector data compression:

+ Advantageous is that this transform method is optimized for the input data to compress. This transform produces a perfect decorrelation of the input data and the highest compression performance from all the transform methods.
– Disadvantageous is that the arithmetic to calculate the transform matrix out of the input data is difficult to implement. No fast execution for real-time implementation is evident at this time.
– Disadvantageous is also that for calculating the transform matrix for many input waveforms, they have to be buffered in large memories together with the transform matrix elements.
– Disadvantageous is also that the calculated transform matrix has to be included in the output data in order to send it to the decoder. This decreases the compression performance significantly.

Discrete Cosine Transform

The discrete cosine transform (DCT) is one of the most popular transform methods in lossy compression used principally in images and audio compression. The transform in one dimension for a vector p_t is given by the equation 3.22.

$$\begin{aligned} G_f &= \sqrt{\frac{2}{n}} C_f \sum_{t=0}^{n-1} p_t \cos\left[\frac{(2t+1)f\pi}{2n}\right] \\ C_f &= \begin{cases} \frac{1}{\sqrt{2}}, & f = 0, \\ 1, & f > 0, \end{cases} \quad for\ f = 0, 1, ..., n-1. \end{aligned} \tag{3.22}$$

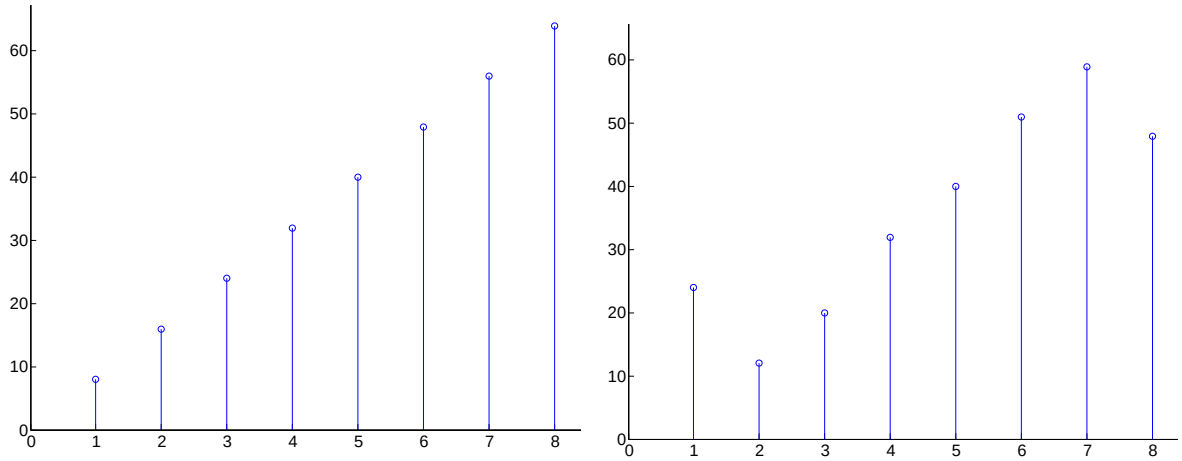
The idea of the DCT is that after the transform the resulting coefficients G_f are decorrelated and the energy is concentrated in the first few coefficients. The first coefficients, which represent the low frequency components of the input data should have large values, whereas the remaining coefficients representing the high frequency components, will have small values close to 0 or even 0. The small values will be quantized strongly, even down to 0 because the high frequency components normally are less important giving only the details of the input signal. This is true for input data where the elements are correlated between them. For a strong correlation of the input data, the DCT can achieve results as good as the Karhunen-Loève transform. Otherwise, if the correlation is low the coefficients are all becoming large. To perform the quantization the first coefficients, which hold the important low frequency information, can be quantized for example to the nearest integer and the rest of the coefficients can be quantized to 0. The quantized coefficients can then be further compressed by using lossless compression methods like Huffman coding or RLE since many coefficients are 0.

The decoder performs the inverse discrete cosine transform (IDCT) after decompressing the quantized coefficients, to reconstruct the input data. The inverse transform is given in equation 3.23.

$$p_t = \sqrt{\frac{2}{n}} \sum_{j=0}^{n-1} C_j G_j \cos\left[\frac{(2t+1)j\pi}{2n}\right], \quad for\ t = 0, 1, ..., n-1 \tag{3.23}$$

The DCT splits the input signals in their different frequency components and therefore it can be compared with the discrete Fourier transform (DFT). The DFT is used for example in telecommunications engineering to transform a modulation signal from the time domain to the frequency domain. The difference between DCT and DFT is that the Fourier transform tends to produce periodical signals while DCT works for non-periodic signals, which can be illustrate with the following example:

The input data consist of the vector $(8, 16, 24, 32, 40, 48, 56, 64)$ and are displayed in figure 3.10(a). The DCT results in the coefficients $(102, -52, 0, -5, 0, -2, 0, 0.4)$. These coefficients are quantized to $(102, -52, 0, -5, 0, 0, 0, 0)$. The inverse DCT on the quantized coefficients yield in the reconstructed input vector $(8, 15, 24, 32, 40, 48, 57, 64)$. The reconstructed input vector is almost identical to the original input data by having only the first 4 coefficients not quantized. When the input vector is transformed by using the DFT the resulting coefficients are $(36, 10, 10, 6, 6, 4, 4, 4)$ as described in [33]. These coefficients are now quantized in the same way as before for the DCT, resulting in $(36, 10, 10, 6, 0, 0, 0, 0)$. After the inverse DFT on the quantized coefficients the reconstructed input vector results in $(24, 12, 20, 32, 40, 51, 59, 48)$, which is illustrated in figure 3.10(b). The tendency of the Fourier transform to produce periodic signals can be seen.



(a) Input vector for DCT and DFT

(b) Reconstructed vector after DFT

Figure 3.10.: 1-dimensional discrete cosine transform showing tendency for periodic results

One property of the DCT coefficients is that they are real numbers with positive and negative signs even if the input data consist of only non-negative integer values. If the DCT coefficients would not be quantized, the original data could be reconstructed perfectly with a small error related to the limited precision of the representation of the real numbers. The use of the DCT in lossless compression is not considered, because the compression performance due to the smaller coefficient values is marginal and the problem of representing real numbers is not trivial in implementations. For the lossy compression, the DCT results in good performances if the coefficients are quantized properly.

In image compression the DCT is used in two dimensions by performing a one dimensional DCT first to the rows and then to the columns of an image block or vice versa. The mathematical

way to perform this is given in equation 3.24.

$$G_{ij} = \sqrt{\frac{2}{n}}\sqrt{\frac{2}{m}} C_i C_j \sum_{x=0}^{n-1}\sum_{y=0}^{m-1} p_{xy} \cos\left[\frac{(2x+1)i\pi}{2n}\right] \cos\left[\frac{(2y+1)j\pi}{2m}\right] \tag{3.24}$$

The indizes i and j move in the ranges $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$, respectively. The inverse DCT in 2D is given by equation 3.25.

$$p_{xy} = \sqrt{\frac{2}{n}}\sqrt{\frac{2}{m}} \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} C_i C_j G_{ij} \cos\left[\frac{(2x+1)i\pi}{2n}\right] \cos\left[\frac{(2y+1)j\pi}{2m}\right] \tag{3.25}$$

The encoder first divides the image into k blocks of normally 8×8 pixels. If the image rows are not exactly divisible by 8, the bottom row is duplicated as many times as needed to complete 8 rows. The same applies to the columns of the image. Then the DCT is performed on each block resulting a vector of 64 coefficients $w_j^{(i)}$ (where $j = 0, 1, ..., 63$ and $i = 1, 2, ..., k$). The k coefficient vectors become the rows of the matrix W.

$$W = \begin{bmatrix} w_0^{(1)} & w_1^{(1)} & w_2^{(1)} & \dots & w_{63}^{(1)} \\ w_0^{(2)} & w_1^{(2)} & w_2^{(2)} & \dots & w_{63}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_0^{(k)} & w_1^{(k)} & w_2^{(k)} & \dots & w_{63}^{(k)} \end{bmatrix} \tag{3.26}$$

Then each column of W is quantized separately so that the first column which represents the DC components, is quantized slightly and the later ones are stronger quantized. The resulting quantized matrix (denoted by Q) is then encoded by using for example variable-size codes (e.g. Huffman coding).

The decoder receives the variable size codes and decodes them. It saves the decoded values as the elements of a matrix which corresponds to the quantized matrix Q. Then it performs the inverse DCT on each row of Q to reconstruct the image resulting in some slight distortion caused by the quantization. An example is shown in [33].

The equation 3.22 can also be written in matrix notation. Therefore, a order $n = 3$ is chosen, which corresponds to a representation in a 3-dimansional space. For the simplicity, in the first steps the normalization factors $\sqrt{2/3}$ and C_f are ignored, which leads to the following equation:

$$\begin{bmatrix} G_0 \\ G_1 \\ G_2 \end{bmatrix} = \begin{bmatrix} \cos(0) & \cos(0) & \cos(0) \\ \cos\left(\frac{\pi}{6}\right) & \cos\left(\frac{3\pi}{6}\right) & \cos\left(\frac{5\pi}{6}\right) \\ \cos\left(2\frac{\pi}{6}\right) & \cos\left(2\frac{3\pi}{6}\right) & \cos\left(2\frac{5\pi}{6}\right) \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \end{bmatrix} = D \cdot p \tag{3.27}$$

The matrix D is the transform matrix for the DCT and it is an orthogonal matrix ($D^{-1} = D^T$). The matrix D contains the following values:

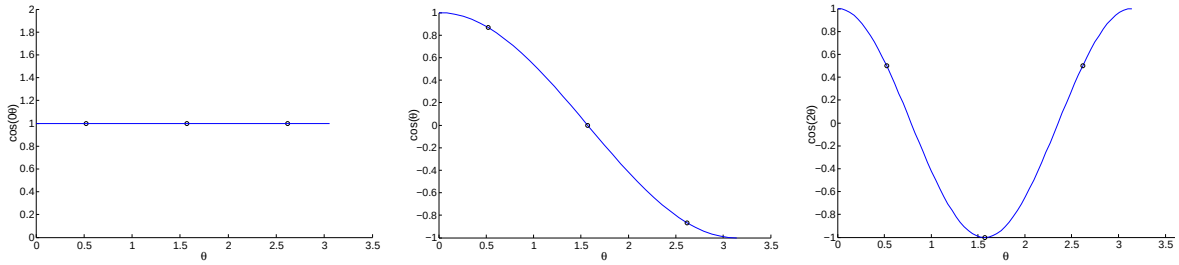
$$D = \begin{bmatrix} 1 & 1 & 1 \\ 0.866 & 0 & -0.866 \\ 0.5 & -1 & 0.5 \end{bmatrix}$$

The particular choice of the angles makes the vectors ($rows$) of the matrix orthogonal but they are not orthonormal. To get orthonormal vectors their magnitude has to be one, which means they have to be normalized. This normalization is done by multiplying the elements of D with the normalization factors $\sqrt{2/3}$ and C_f as given in equation 3.22. This results the orthonormal matrix M:

$$M = \begin{bmatrix} 0.5774 & 0.5774 & 0.5774 \\ 0.7071 & 0 & -0.7071 \\ 0.4082 & -0.8165 & 0.0.4082 \end{bmatrix} \tag{3.28}$$

The vectors (rows) of the matrix M starting from the first on top, have increasing frequencies. The frequencies of the three rows are shown in figure 3.11. This property allows extracting the frequencies contained in the input data and concentrating the energy (important information) in the first resulting transform-coefficients.

The orthonormal property of the matrix M leads to the decorrelation of the input data by returning decorrelated transform coefficients, which are also related to the different frequencies in the input data. Another, property of orthonormal matrices is that the inverse of the matrix is also its transpose, what makes it possible to reconstruct the input vector (v) by multiplying the coefficients with M^T. If $M \cdot v = c$ then v can be reconstructed from c by $M^T \cdot c = M^{-1} \cdot c = v$.



(a) DC frequency (b) Low frequency (c) High frequency

Figure 3.11.: Increasing frequencies of the vectors of the DCT matrix

Suitability of discrete cosine transform for the implementation of a detector data compression:

+ Advantageous is that the DCT method produces a good decorrelation of the input data by extracting the different frequency components from the input waveforms.
+ Advantageous is that according to the good decorrelation an easy and efficient quantization of the high frequency coefficients can be achieved which can yield in a good compression performance with acceptable low distortion of the reconstruction.
+ Advantageous is that the transform matrix elements consist only of a few different values, which makes it easier to store the transform matrix.
+ Advantageous is also that the number of resulting transform coefficients is equal to the number of samples in the input waveforms, which makes it easy to handle variable waveform lengths (variable elements per input vector).
– Disadvantageous is that the multiplication of the input vectors with the transform matrix is not so easy to implement.
– Disadvantageous is also that the DCT results in positive and negative floating-point coefficients even if the sample values at the input are pure positive integers.

Wavelet Transform (Subband coding)

In the previous section the transform coding is described, which uses orthogonal basis functions to decompose the input signal in different frequency components. Another method to decompose the input signal is the wavelet transform, which splits the input signal in subbands with dedicated frequency bands.

The name wavelet is related to the requirements, which such functions have to fulfill. Among these requirements is that a wavelet has to integrate to zero [33]. That means that the area the wavelet curve encloses above the x-axis it also has to have bellow the x-axis, which justifies the first part of the name ”wave”. A second requirement is that the wavelet function has to be localized in space (or time), which means that it has to be limited in a certain range and result zero outside this range. This non-infinitely continuing wave therefore is named with the diminutive ”wavelet”

to define it as a small part of a wave. The wavelet transform can be compared with the Fourier transform. The Fourier transform is used to translate a signal from the time domain to the frequency domain. The Fourier transform returns the frequency components, which are present in the input signal and the amplitude of these frequency components. The disadvantage is that the Fourier transform decomposes the input signal in sums of sine and cosine functions, which periodically continue infinitely and therefore, are not localized. This results in a good resolution of the signal in the frequency domain but gives no time information while representing the signal in the frequency domain. After the Fourier transform of a signal, we know all the frequencies contained in the signal and there amplitudes but we do not know at which time each frequency component appears in the analyzed signal. The Fourier transform and its inverse give us either a good resolution in time and zero resolution in frequency or a good resolution in frequency but zero resolution in time. The wavelet transform uses localized functions to transform an input signal in the frequency domain, it uses a window, which slides over the input signal and gives us the frequency components seen in this window. The window is shifted in time to analyze the different parts of the input signal separately, but it is also scalable to see the overall behavior of the signal as well as the detailed aspects. One fact is that it is not possible to achieve a perfect resolution of the signal in time and frequency at the same time, which is described by the uncertainty principle known from particle physics. The better the resolution in frequency is the lower the resolution in time can be and vice versa.

To compute the wavelet transform a wavelet function has to be defined which is denoted by the "mother wavelet". To execute the transform the inner product of the input signal and the mother wavelet is calculated. As described above the mother wavelet has to fulfill the following conditions:

1. The total area under the curve of the function is zero, i.e.

$$\int_{-\infty}^{\infty} \psi(t)dt = 0. \tag{3.29}$$

2. The total area of $|\psi(t)|^2$ is finite, i.e.

$$\int_{-\infty}^{\infty} |\psi(t)|^2 \, dt < \infty. \tag{3.30}$$

This means that 1. the mother wavelet has to be a wave that oscillates above and below the t(x)-axis and that the total area below this wave is zero. The 2. condition implies that the energy of the function is finite, which means that the function is localized in some finite interval and is zero outside this interval. The continuous wavelet transform is than given by:

$$W(a,b) = \int_{-\infty}^{\infty} f(t)\psi_{a,b}(t)dt. \tag{3.31}$$

$$\psi_{a,b}(t) = \frac{1}{\sqrt{|a|}}\psi\left(\frac{t-b}{a}\right) \tag{3.32}$$

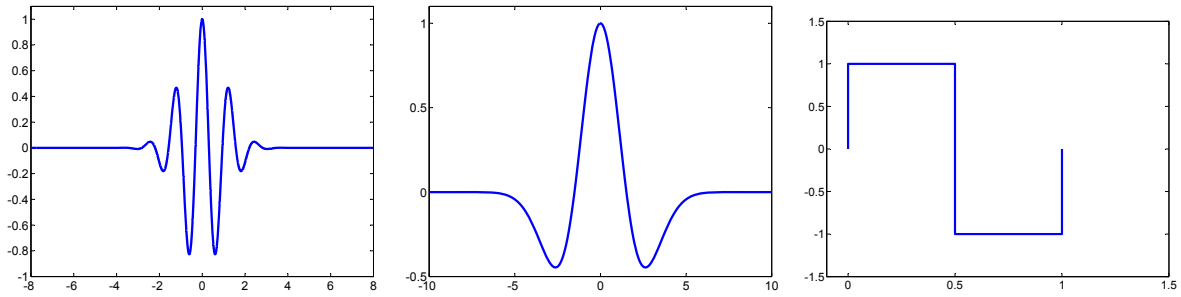
The mother wavelet ψ is scaled by the factor a and shifted by the translation factor b. The quantity $1/\sqrt{|a|}$ is a normalization factor that ensures that the energy of $\psi(t)$ remains independent of a and b [33].

The inverse wavelet transform is given by:

$$f(t) = \frac{1}{C}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \frac{1}{|a|^2} W(a,b)\psi_{a,b}(t)da \; db \tag{3.33}$$

where the quantity C is defined as:

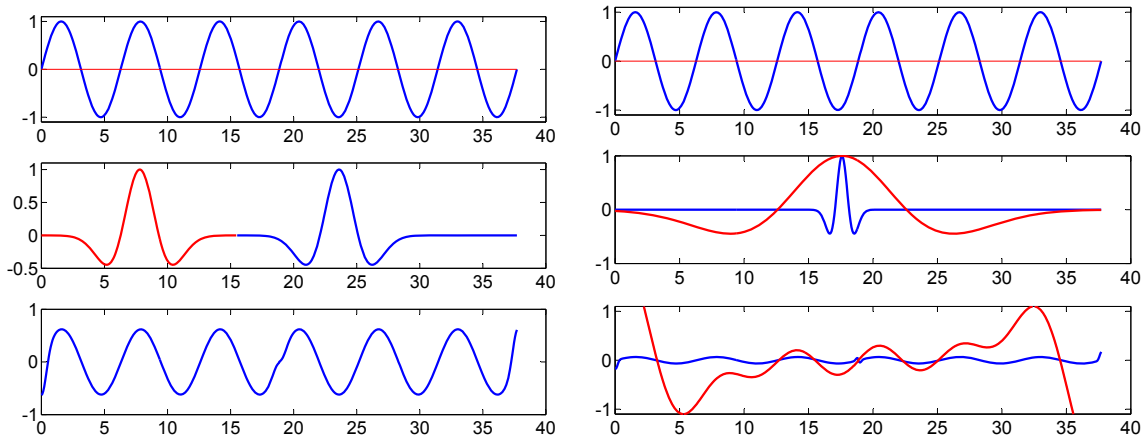
$$C = \int_{\infty}^{\infty} \frac{|\psi(\omega)|^2}{\omega} d\omega. \tag{3.34}$$



(a) Morlet Wavelet (b) Mexican Hat Wavelet (c) Haar Wavelet

Figure 3.12.: Three examples of mother wavelet functions

In figure 3.12 three examples of a mother wavelet are illustrated. The way the continuous wavelet works can be explained by the example of transforming a pure sine wave. The used mother wavelet is the ”Mexican Hat”. In figure 3.13(a) the sine wave is shown in the first graph. The second graph shows the Mexican Hat mother wavelet and a shifted (translated) version of the Mexican Hat function. The third graph shows the resulting wavelet coefficients. The resulting coefficient values



(a) Continuous Wavelet decomposition using Mexican Hat (b) Scaled Mexican Hat results

Figure 3.13.: Continuous Wavelet decomposition of a pure sine wave

are received by continuously shifting the Mexican Hat function from left to right (along the time axis) and calculating the inner product of the Mexican Hat and the sine wave. The first Mexican Hat is located at a point where the sine wave has a maximum and therefore it has a good match with the sine function. The resulting wavelet coefficient has a maximum at this position. The second, translated Mexican Hat is positioned where the sine wave has a minimum. At this point the wavelet and the sine wave are mirrored which leads to a large negative transform coefficient. In between the coefficients drop from positive to zero to negative values.

If the wavelet is scaled the frequency of the wavelet and the sine wave are not anymore similar and the resulting coefficients get smaller. This can be seen in figure 3.13(b). The Mexican Hat function is either stretched to cover several periods of the sine wave or shrunk to be much narrower than one cycle of the sine wave. If these scaled functions are translated from left to right, the resulting coefficients are much smaller than in the previous case. The larger the scaling is the closer the coefficients become to a constant value.

In data compression, the input signals are in most of the cases discrete signals e.g. obtained by sampling audio signals. Therefore, the use of the wavelet transform in data compression is executed in the discrete form. To explain how the discrete wavelet transform works the simplest

mother wavelet is used, which is the Haar wavelet shown in figure 3.12(c). To illustrate the use of the Haar wavelet in data compression a single row of pixel values of a grayscale image is used. The vector containing the pixel values has a length of n, where n should be a power of 2. If the number of pixels is not a power of 2 the vector can be extended with 0's, which are then cut again at the decoder. Let us use now an example vector of 8 pixel values being $(1, 2, 3, 4, 5, 6, 7, 8)$ [33].

The first step is to calculate the averages between two adjacent pixels resulting in the four values $(1 + 2)/2 = 3/2, (3 + 4)/2 = 7/2, (5 + 6)/2 = 11/2, (7 + 8)/2 = 15/2$. Transmitting these four values is not enough to reconstruct the original pixels. In addition the differences between adjacent pixels are calculated which result in the four values $(1 - 2)/2 = -1/2, (3 - 4)/2 = -1/2, (5 - 6)/2 = -1/2, (7 - 8)/2 = -1/2$. The resulting vector has the values $(3/2, 7/2, 11/2, 15/2, -1/2, -1/2, -1/2, -1/2)$. The first four elements, the averages, are called the smooth coefficients and the last four values, the differences are called the detail coefficients. With these eight values, the original pixel values can now be reconstructed. If the adjacent pixel values are correlated, the smooth coefficients will resemble the original pixel values and the differences will be small values. The small values can now be compressed efficiently by using RLE or Move-to-Front and Huffman coding. This algorithm can be applied iteratively by calculating the averages and differences of the remaining smooth coefficients. After the second iteration the vector contains the values $(10/4, 26/4, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2)$. The next and final iteration gives the final output vector $(36/8, -16/8, -4/4, -4/4, -1/2, -1/2, -1/2, -1/2)$. This vector is the output of the Haar wavelet transform of the original input vector. The original pixel values can perfectly be reconstructed with an iteratively reverse algorithm and the small difference (detail) coefficients help to compress the resulting output vector.

To achieve an orthonormal Haar Transform the output vector should be normalized by dividing the coefficients with the square root of the resolution. The resolution is defined as the number of remaining smooth coefficients after each iteration. The normalized output vector is given by:

$$\left(\frac{36/8}{\sqrt{2^0}}, \frac{-16/8}{\sqrt{2^0}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-4/4}{\sqrt{2^1}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}, \frac{-1/2}{\sqrt{2^2}}\right) \tag{3.35}$$

The wavelet transformation of a 2D-image can be performed by applying the iterative algorithm first to each row of the image matrix, which results in the first column containing all averages and the other columns contain the differences. Then the iterative algorithm is applied column-vice to the resulting values, which gives one average value at the top-left corner. The rest of the top row is the average of the differences and all other values are differences. This approach is called standard decomposition [33].

A second approach is to apply each iteration of the wavelet transform alternating to the rows and columns. The first step is to calculate the averages and differences for all rows and then for all columns. These results in the averages in the first halve of the rows and upper half of the columns (top-left quarter) and the rest are differences. In the second iteration, the averages and differences are calculated only for the top-left quarter, and so one. This approach is called pyramid decomposition [33].

The calculation of the wavelet transform can also be executed by a matrix approach. The calculation of the averages and differences for the Haar wavelet transform can be done using a transform

matrix. The transform matrix for the first iteration is given by A_1.

$$A_1 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix}, \quad A_1 \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} = \begin{pmatrix} 3/2 \\ 7/2 \\ 11/2 \\ 15/2 \\ -1/2 \\ -1/2 \\ -1/2 \\ -1/2 \end{pmatrix} \tag{3.36}$$

For the second iteration the transform matrix results in A_2 and the third iteration is performed with A_3:

$$A_2 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad A_3 = \begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.37}$$

The Haar wavelet transform can be calculated by multiplying the transform matrices for each iteration resulting the Haar wavelet transform matrix $W = A_3 \cdot (A_2 \cdot A_1)$. Then to calculate the transform coefficients the input vector containing the pixel values is multiplied with the overall transform matrix W.

$$W \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} = \begin{pmatrix} \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} \\ \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & -\frac{1}{2} \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} = \begin{pmatrix} 36/8 \\ -16/8 \\ -4/4 \\ -4/4 \\ -1/2 \\ -1/2 \\ -1/2 \\ -1/2 \end{pmatrix} \tag{3.38}$$

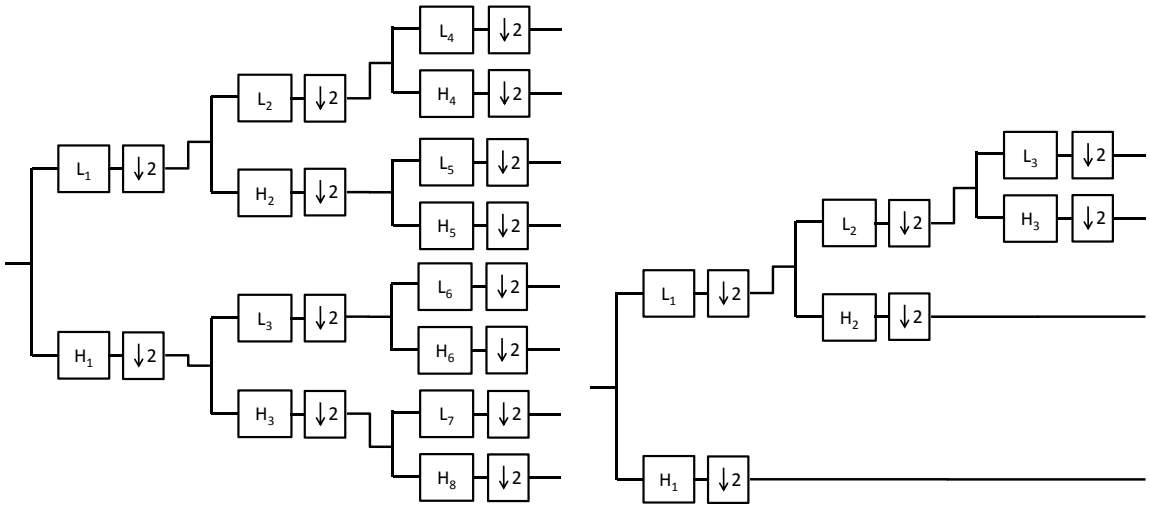
To perform the Haar transform on a 2-D image, W is first applied to the columns of the pixel matrix and then to the resulting rows. Mathematically this can be written as follows:

$$I_{tr} = \left(W \left(W \cdot I\right)^T\right)^T = W \cdot I \cdot W^T \tag{3.39}$$

In order to receive a orthonormal transform matrix for the Haar wavelet the used division by 2 in the transform matrixes A_1, A_2, A_3 have to be exchanged by $1/\sqrt{2}$.

The implementation of wavelet transform can be done by using so called filter banks. Considering the example of the Haar wavelet transform, the average and difference coefficients (smooth and detail) can be obtained by using low-pass and high-pass linear filters. The building of the averages of the input samples can be compared with a linear low-pass filtering of the input signal, while the differences are obtained by linear high-pass filtering. Each filter output produces n values where n is the length of the input vector, which results in twice as much output values than input samples. The Nyquist criteria says that sampling a continuous signal which has a finite bandwidth with a rate slightly more than double the maximum contained frequency allows us to

reconstruct the original signal by interpolating between the samples. The finite bandwidth of the signal source (such as microphone, speaker) is given by the limited respond speed of the source and prevents that the signal can change faster as 2 times the maximum frequency. The decomposition of the input signal by using a low-pass and a high-pass filter results in two output signals, which have half the input bandwidth. Since the output bandwidth of each linear filter is half the input bandwidth, we only need half the output samples of both to reconstruct the original input samples. Therefore, after the filters the outputs are down sampled by a factor of 2 retaining only the even numbered output values. In the previous discussion of the wavelet transform is shown that only four averages and four differences are needed to reconstruct the eight input pixel values. The decomposition of the input signal into two subbands by using a low-pass filter and a high-pass filer is called subband coding. The iterative execution of the Haar transform can be achieved by consecutively connecting several low-pass and high-pass filters as well as down sampling devices. Two different approaches are shown in figure 3.14 [35]. For the low-pass and high-pass filters,



(a) Flat filter bank (b) Dyadic filter bank

Figure 3.14.: Subband decomposition using 2 different filter bank concepts

liner finite impulse response (FIR) filters are preferred. The FIR filter coefficients for the Haar wavelet result for the low-pass filter in $h(0) = 1/2, h(1) = 1/2$ and for the high-pass filter in $h(0) = 1/2, h(1) = -1/2$. The disadvantage of the Haar wavelet decomposition using the FIR filters for example in image compression is that artifacts, such as ringing or aliasing, are introduced caused by the not ideal filters and the down sampling. To reduce this problem better FIR filters with higher tab order are searched. Some well known mother wavelet functions which result in better FIR filters are the Coifman wavelet, the Daubechies wavelet, Beylkin wavelet and more [33].

Suitability of discrete wavelet transform for the implementation of a detector data compression:

+ Advantageous is that the DWT method produces a good decorrelation of the input data by consecutively splitting the input waveforms (vectors) in two frequency bands.
+ Advantageous is that the low frequency band contains the averages, which still preserve some properties of the input waveform. This can be used for preselecting some data at the decoder to find some signals with certain properties without having to decode all the received data.
+ Advantageous is that the transform is easy to implement using FIR filter banks.
+ Advantageous is that pipelined filter banks allow a perfect real-time implantation.
– Disadvantageous is that a right mother-wavelet has to be defined for the underlying input data to obtain a good compression efficiency with low distortion.
– Disadvantageous is also that it is more complicated to find the coefficients, which can be

quantized strongly since the high frequency coefficients are not all at the end of the resulting coefficient vectors.
- Disadvantageous is that for the DWT the number of samples per input waveform has to be a power of two. This can decrease the compression performance since input waveforms can have variable number of samples and need to be extended to input vectors with a power of two numbers of elements.

3.2.3. Predictive coding

Another lossy compression method, which is used e.g. in image and audio coding is based on predictive coding. Some kinds of data to be compressed contain a large correlation between neighboring samples. That means that for example neighboring pixels in natural images are likely to have similar or equal color values, or consecutive (in time) samples of audio signals are likely to have similar amplitudes. To use this correlation the first idea what comes in mind is to encode the differences between neighboring samples or pixels instead of encoding the original values. The differences of strongly correlated input data should be presented in a much smaller range as the original values and this requires fewer bits to present them. In addition, if a quantization is performed on the differences the error introduced will be much smaller then by quantizing directly the original samples. The compression efficiency is depending on how close the value of the actual sample x_n is to the preceding sample x_{n-1} and therefore how small the difference gets. An even better compression performance can be achieved by not just using the preceding sample and calculating the difference but by using more than one preceding sample to predict the value of the actual sample x_n and then calculating the difference between the predicted value $\widehat{x}_n$ and the original value. In this way, not only the similarity of neighboring samples is used but also the trend in the signal, which corresponds to the higher order correlation.

A general predictive coder uses N signal samples back in time or previous encoded neighboring pixels to make a prediction of the actual signal sample and then encoding the difference between the predicted and the original value [35].

The decoder will use the same N previous reconstructed samples to do the same prediction and then use the received difference to correct the prediction and to obtain the original value.

If the difference between the predicted value and the original value is quantized, as it is the case for lossy compression then the encoder has to use the reconstructed values for the prediction, that is the encoder has to mimic the decoder. This is important because the decoder has only the information about the previous reconstructed samples containing the quantization error and not of the exact original values. Therefore, the decoder can predict the value of the actual sample only based on the previous reconstructed samples and the encoder has to use the same information what is available to the decoder, which is the reconstructed value including the quantization error and not the original values. This important fact is illustrated in figure 3.15.

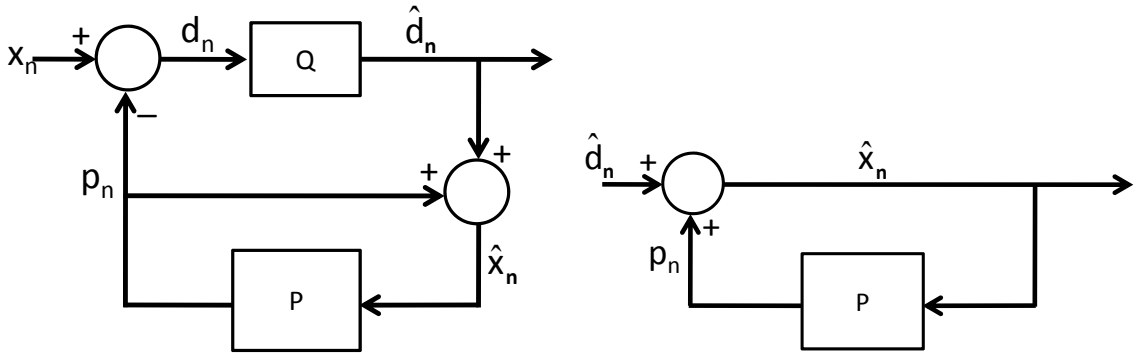
There are different functions to estimate (predict) the value of the actual sample. In the following the use of linear predicators is explained which bases on a linear combination of previous sample values and is considered as a auto regressive (AR) model of the source [35].

$$p_n = a_1\widehat{x}_{n-1} + a_2\widehat{x}_{n-2} + ... + a_N\widehat{x}_{n-N} = \sum_{i=1}^{N} a_i\widehat{x}_{n-i} \tag{3.40}$$

The prediction error $d_n = \widehat{x}_n - p_n$ is quantized and transmitted. Both the coder and decoder then reconstruct $\widehat{d}_n$ and $\widehat{x}_n = p_n + \widehat{d}_n$ which is used in the next iteration to predict the next input sample.

The predictor coefficients a_i should be chosen in a way to minimize the distortion D.

$$D = \frac{1}{N}\sum_{n=1}^{N}(x_n - \widehat{x}_n)^2 = \frac{1}{N}\sum_{n=1}^{N}\left(d_n - \widehat{d}_n\right)^2 \tag{3.41}$$



(a) Predictive Encoder (b) Predictive Decoder

Figure 3.15.: Blockdiagram of the predictive coding method

The quantization makes it difficult to calculate the predictor coefficients, but if one assumes fine quantization which means that the number of quantization levels is large, the approximation $\widehat{x}_n \approx x_n$ can be done [35]. With this assumption, the variance of the prediction error is given by:

$$\sigma_d^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - p_n)^2 = \frac{1}{N}\sum_{n=1}^{N}\left(x_n - \sum_{i=1}^{N} a_i \widehat{x}_{n-i}\right)^2 \approx \frac{1}{N}\sum_{n=1}^{N}\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right)^2 \tag{3.42}$$

To calculate the coefficients which give the minimum distortion D the variance of the prediction error is differentiated with respect to each a_i and the resulting equation is set to 0. This gives N equations for the N predictor coefficients as in 3.43.

$$\frac{\partial}{\partial a_j}\sigma_d^2 = -2\frac{1}{N}\sum_{n=1}^{N}\left[\left(x_n - \sum_{i=1}^{N} a_i x_{n-i}\right)\cdot x_{n-j}\right] = 0 \tag{3.43}$$

The predictor coefficients can be calculated also by using matrix notation and the auto correlation function 3.44.

$$R_{XX}(k) = \frac{1}{n-k}\sum_{i=1}^{n-k} x_i \cdot x_{i+k} \tag{3.44}$$

$$\mathbf{R} = \begin{pmatrix} R_{XX}(0) & R_{XX}(1) & \dots & R_{XX}(N-1) \\ R_{XX}(1) & R_{XX}(0) & \dots & R_{XX}(N-2) \\ \vdots & \vdots & \ddots & \vdots \\ R_{XX}(N-1) & R_{XX}(N-2) & \dots & R_{XX}(0) \end{pmatrix} \tag{3.45}$$

The predictor coefficients can now be calculated as:

$$\mathbf{RA} = \mathbf{P} \Rightarrow \mathbf{A} = \mathbf{R}^{-1}\mathbf{P} \tag{3.46}$$

Where **P** contains the resulting predicted values which should be the original values and **A** is the vector of the predictor coefficients and they are given by:

$$\mathbf{P} = \begin{pmatrix} R_{XX}(1) \\ R_{XX}(2) \\ \vdots \\ R_{XX}(N) \end{pmatrix}, \quad \mathbf{A} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{pmatrix} \tag{3.47}$$

Suitability of predictive coding for the implementation of a detector data compression:

+ Advantageous is that predictive coding can perfectly use the correlation of the samples according to the predefined signal shape. A good compression performance can be obtained by quantizing the differences between predicted values and original sample values.
+ Advantageous is also that a simple calculation of only the differences between consecutive samples can be used in combination with other compression methods to increase their compression efficiency. With the differences the correlation is already exploit to some extend in a simple way.
– Disadvantageous is that a more complex prediction using more than one preceding sample is difficult to implement in hardware.
– Disadvantageous is also that by using the quantization for increasing the compression performance a decoding has to be performed already in the implementation of the encoder to predict following samples value in the same way as the decoder does.

3.2.4. Model based coding

In model based coding a parametric function is searched, which can model the underlying data source. To compress the data, the resulting model parameters for the input signals are sent to the decoder instead of encoding the input signal directly. This principal is also known as analysis/synthesis coding and is used for example in human speech coding.

The encoder uses the defined model for the given source and performs an analysis of the input signal to estimate the model parameters corresponding to the input signal. These model parameters are then send to the decoder [35].

The decoder uses the received parameters to control the same model for the underlying source and performs a synthesis to recreate the input signal.

A variant of this method is called analysis by synthesis coding, where the encoder also performs a decoding (synthesis) step and tries to find the model parameters that give the closest decoded signal to the original signal. The encoder optimizes the parameter with respect to the decoded signal before they get transmitted [35].

Model based coding will work well when a good model of the source can be found which represents a narrow class of signals. For example, the way that human speech sounds are produced has been studied extensively and a good model for this is developed. In contrary, for audio coding of music it is much more difficult to define a model since the spectrum of how music can be produced is broad. As an example of model based coding the human speech coding is described in the following.

The sounds for speech are produced by forcing air through the vocal cords in the larynx. If the vocal cords are tense, they vibrate and generate tones and overtones (voiced sound). If the vocal cords are relaxed, a noise-like sound is produced (unvoiced sound) [35]. This can be modeled by a switch that switches between a noise source and a signal source. The signal source produces a signal train that can have variable or fixed amplitudes at constant intervals that correspond to the pitch period of the input signal. The rest of the vocal tract of humans can be modeled by a linear filter.

The input speech signal is split in short segments. For each segment, the filter coefficients and the switch position (voiced/unvoiced sound) are estimated and for a voiced segment, a signal train is defined according to the pitch period. Then the switch position, the filter coefficients and if required the defined signal train are transmitted to the decoder. The blocks of the simple speech model are shown in figure 3.16.

The decoder uses the received filter parameters and the switch position to recreate the sounds for the spoken phrases. If voiced sounds are received, the decoder uses the received signal train information to recreate the input signal for the filter. The more parameters are used for the model the less the recreated voice sounds artificial. On the other hand, the more parameters have to be

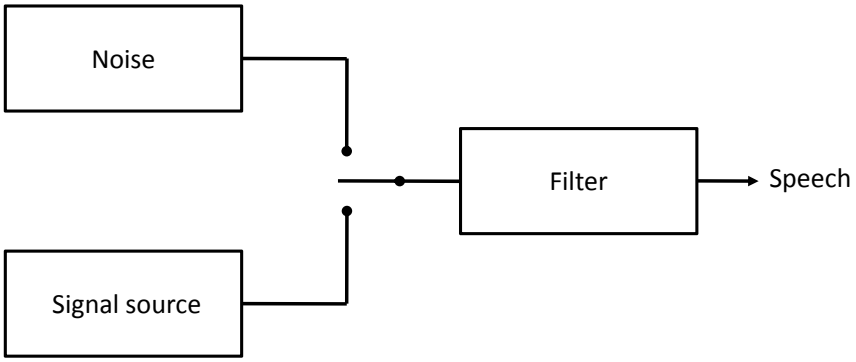


Figure 3.16.: Model for human speech model [35]

transferred, the lower the compression efficiency gets.

For other kind of sources, other models have to be found which are capable to produce signals similar to the real input signals and which can be described with a few model parameters to result in a good compression.

Suitability of model based coding for the implementation of a detector data compression:

+ Advantageous is that for the high energy physics not the single sample values are important, only the amplitude of the input waveforms and the time stamps. If a model can be created which extracts these tow information of the received waveforms with an accuracy, which is high enough for the aimed physics research, a very high compression performance could be achieved.
+ Advantageous is also that the off detector analysis of the detector data already would have the important information. For the time being a complicated fitting of the sample values is done offline to reconstruct the original amplitude and time of the corresponding analogue signal before the digitization.
– Disadvantageous is that this complicated fitting cannot be performed in a hardware implementation because it is to complex. A simpler model has to be found.
– Disadvantageous is that by using a simpler model a good accuracy of the extraction of the important information cannot be guaranteed. Considering different effects on the input waveforms (e.g. pile-up effects) can cause wrong parameters. A mechanism has to be implemented in hardware to check back the quality of the extracted parameters, which increases the complexity of the implementation. If the quality is low, the corresponding input waveform has to be sent out uncompressed.

4. Algorithm for data compression in particle detectors

After a short inside in the data compression theory is given in chapter 3 in this chapter a deeper analysis of the promising compression methods is discussed using measured data from the ALICE TPC front-end electronics. The best-suited compression algorithm found in this chapter is then used to develop an implementable real-time hardware solution that is optimized for the TPC front-end electronics in ALICE. All promising compression methods are discussed and compared in this aspect. Thereby a distinction between lossless and lossy compression methods is applied. The constellation of the data from the ALICE TPC consist of digitized samples from pre-amplified and shaped semi-Gaussian waveforms that are produced in the front-end electronics as already discussed in chapter 3. The investigated compression methods in this chapter are executed in Matlab on these TPC data to retrieve their compression efficiency and compare them. The same data are used as well for testing the developed real-time hardware implementation as presented later in chapter 6.

The chapter starts with a discussion of the characteristics of the underlying TPC data set. Then a closer look first to the promising lossless compression methods is given and then some lossy methods are discussed. The algorithms are analyzed focusing on the requirements for the underlying data set and the possible implementation in real-time front-end electronics. The lossy methods are also carefully investigated and compared for their introduction of distortion in the reconstructed data caused by the information loss. Therefore, the Peak-Signal-to-Noise Ratio ($PSNR$) is used as a measure of the introduced distortion. A first estimate of the effect of the distortion on the physical relevant information is given by fitting the original input waveforms and the reconstructed waveforms and comparing the amplitudes and peak times.

4.1. Characteristics of the digitized detector data from the ALICE TPC

The signal from the ALICE TPC pads is pre-amplified and shaped by the PASA chip. The PASA consists of an integrator stage, which integrates the charge induced in the detector pad and converts the induced current into a voltage. A second stage of a differentiator brings the voltage back to ground and a filter bank limits the spectrum of the voltage signal. This results in a semi-Gaussian shaped output voltage signal with a fast rise time of a few nanoseconds and a slower decay time. The fast rise time is correlated to the fast moving electrons from the avalanche process in the MWPCs, whereas the slower decay time is related to the slow moving ions of the avalanche process. The movement of the electrons and ions created by the avalanches induces the charge in the detector pads.

The primary electrons that are liberated by the traversing charged particles in the TPC gas volume and start the avalanches in the MWPC, are not only spread in space but also in time. Therefore is the induced signal in a pad spread over 400 ns to 500 ns in time. The shape of a cluster in the detector plain is given by a Gaussian distribution in space and time. The folding of the impulse response of the PASA with the induced signal results in a semi-Gaussian shaped voltage signal with around 500 ns-600 ns duration as shown in figure 4.1.

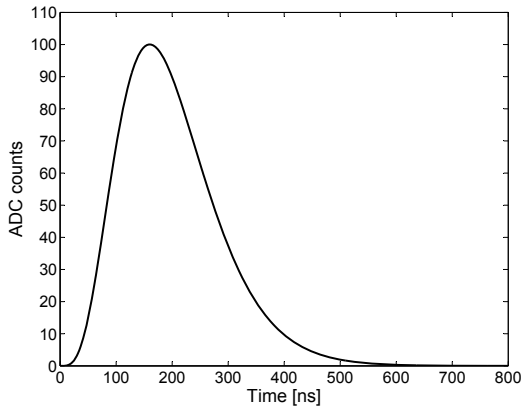


Figure 4.1.: Example of a PASA output signal with semi-Gaussian shape

The voltage output signal of the PASA is then digitized by an ADC in the ALTRO chip which samples the signal with 10 MHz resulting in around 5 to 6 samples. By a high occupancy of the TPC detector, it can happen that one pad records two semi-Gaussian waveforms within a short time distance and the tail of the first waveform adds up to the second one. This leads to so called pile-up effects and the tow waveforms are seen as one long signal by the following digital processing block.

The digital processing unit inside the ALTRO chip prepares the signal for a zero suppression to reduce the data volume. At the end, the zero suppressed data are stored in multi-event buffers from where they can be readout and compressed further from the implementation of the compression method of choice.

4.1.1. Zero suppression

The digital processing unit consists of two baseline corrections and a tail cancelation. After the digital processing unit has corrected the baseline and the pile-up possibility is minimized, zero suppression can be performed by applying a threshold level. If a sample rises above the threshold, the start of a waveform is indicated and all samples above the threshold are saved until a sample falls again below the threshold. The zero suppression concept is illustrated in figure 4.2.

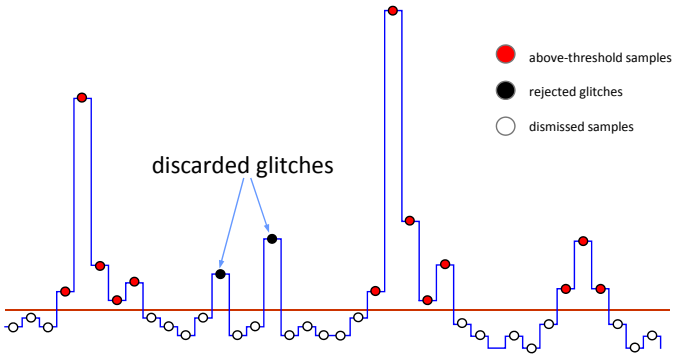


Figure 4.2.: Principle of the zero suppression

Single samples above the threshold are not saved because they are considered as glitches. A time stamp is added to each saved waveform to preserve the arrival time information, which is important to get the z-component of the particle trajectory. The compression of the time stamp is not considered in this work.

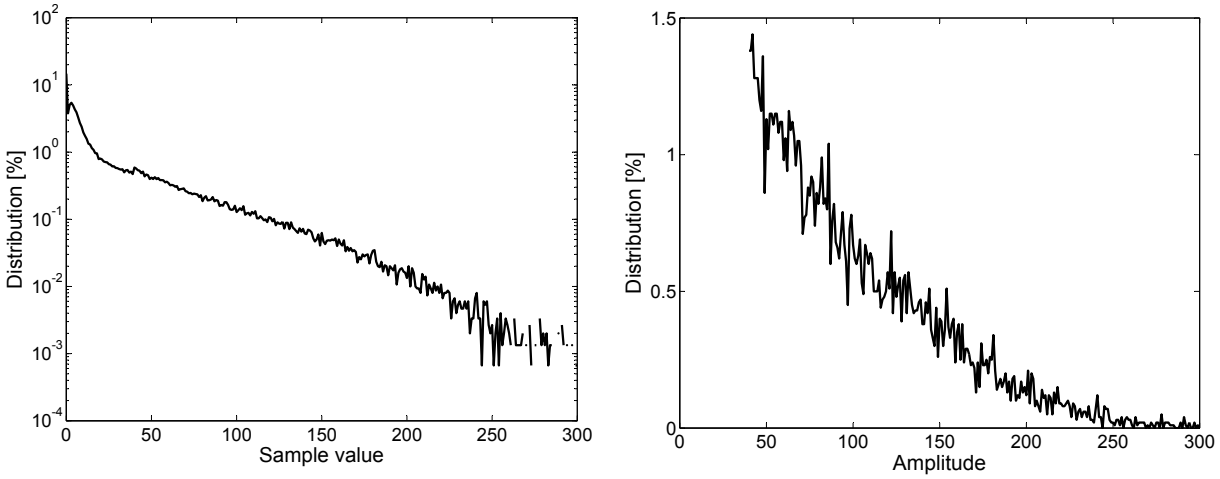
4.1.2. Test-data characteristics

The following compression methods are modeled and tested according to the data from a measured data set of the ALICE TPC. This data set contains 10 000 waveforms recorded from cosmic ray measurements realized in 2006. The front-end electronic was configured to performed the statistical baseline correction, whereas the tail cancelation filter and the baseline correction with the moving average filter are deactivated. The measured data are recorded without zero suppression. From the recorded data around 10 000 detected semi-Gaussian waveforms from different detector pads are combined to form a Matlab test-data matrix which than is used to investigate the different compression methods. Since the data are not zero suppressed in Matlab a zero suppression with a low threshold is simulated giving waveforms with a constant length of 15 samples. The waveforms are saved in the matrix with the samples in increasing sampling time order and with their maximum samples positioned all in the same column. A illustration of the matrix is shown in 4.1.

$$\begin{pmatrix} S1_{P1} & \dots & S5_{P1} & S6_{P1}(Max) & S7_{P1} & \dots & S15_{P1} \\ 8 & \dots & 31 & 71 & 67 & \dots & 10 \\ 7 & \dots & 47 & 50 & 36 & \dots & 9 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ S1_{P10000} & \dots & S5_{P10000} & S6_{P10000}(Max) & S7_{P10000} & \dots & S15_{P10000} \end{pmatrix} \tag{4.1}$$

The symbol $S1_{P1}$ stands for the first sample ($S1$) of the first input waveform ($P1$) in the test-data matrix. The second and third row of the matrix 4.1 shows real sample values of the waveforms from the measurement. The waveforms in the test-matrix have different amplitudes and different widths at half maximum (FWHM). Some waveforms contain even pile-up effects. I have chosen a number of 15 samples for the signal widths regarding the specifications of future front-end electronics, which will most likely sample the analogue signals with 40 MHz, which is four times faster as the actual 10 MHz-ADC does. An average semi-Gaussian waveform with actually around 3 to 5 samples will result in 12-20 samples with the higher sampling and assuming the same signal duration. Therefore, I use 15 samples per waveform as mean signal duration in these first investigations of the compression algorithms.

The probabilities of the samples and the distribution of the amplitude of the input waveforms in the data set are represented in figure 4.3.



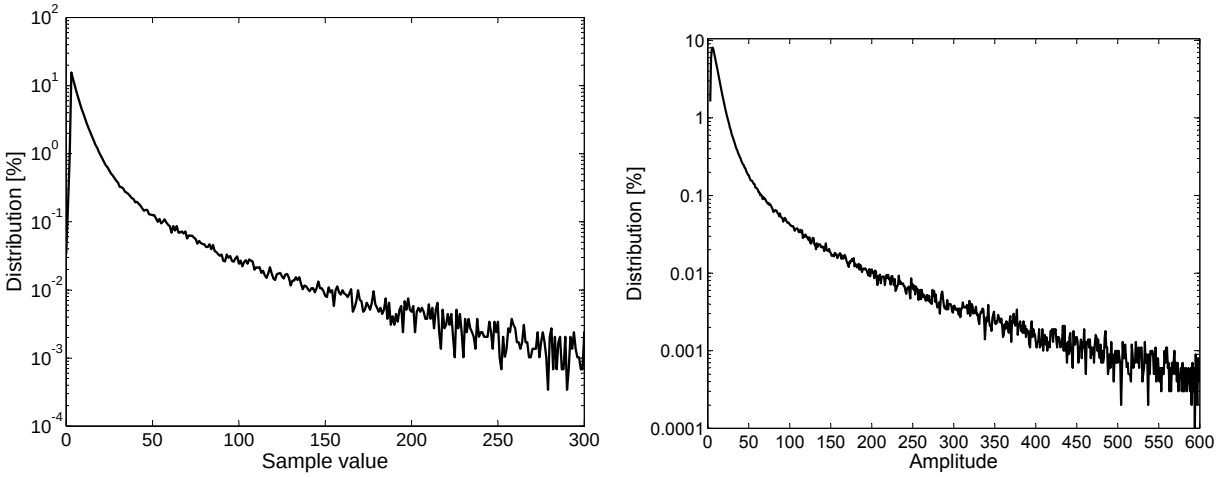
(a) Distribution of the sample values in the test-data

(b) Distribution of the amplitude of the waveforms in the test-data

Figure 4.3.: Distribution of the single sample values and of the amplitudes of the semi-Gaussian waveforms in the test-data matrix used for the investigation of the compression methods in Matlab

A few sample values are seen above 300 ADC counts but are cut out in the graph for a better visualization.

First analysis of more recently recorded data by the ALICE TPC front-end from real proton-proton collisions in the LHC in 2010 result in an amplitude distribution shown in figure 4.4. The distribution of the sample values of the waveforms are shown as well. The measured data from these proton-proton collisions is already zero suppressed. It can be seen that the distribution of the waveform amplitudes and the sample values in the test-data are similar to the one of the obtained real collision data from 2010.



(a) Distribution of the sample values

(b) Distribution of the amplitudes of the waveforms

Figure 4.4.: Distribution of the sample values and amplitudes of real zero suppressed proton-proton collision waveforms measured from the TPC in ALICE in 2010

The properties of the data set can be summarized as follows:

- Correlation of consecutive samples in a waveform according to the defined signal shape
- Different amplitudes of the waveforms (i.e. samples) in the range of 10-411
- The sample values are represented by 10 bit which gives a maximum range of 1024 possible values
- All sample values are positive
- Different widths of the waveforms at half maximum (FWHM)
- Some waveforms with pile-up effect are present as well in the test-data
- Different probabilities of the sample values

4.1.3. Categorizing the data compression methods

Two different approaches can be distinguished in compressing the data from the TPC. The data can be compressed either locally on a channel-by-channel (pad-by-pad) basis or globally by compressing full clusters, which are extended over several pads and over time. The clusters are produced by the primary electrons from the TPC drift region, which start the avalanche processes that create clouds of electrons and ions close to the detector pads. The charge cloud has a Gaussian distribution and extends over several pads inducing signals in these pads as it can be seen in figure 4.5. The important information from this cloud is the position in the pad plain and the time to reconstruct the particle trajectories as well as the amplitude, which is related to the energy loss of the primary particle.

An investigation of data compression methods based on the global cluster compression can be found in [37]. The efficient compression methods are based on models to extract the important physical information from the clusters instead of transmitting every single signal from the pads.

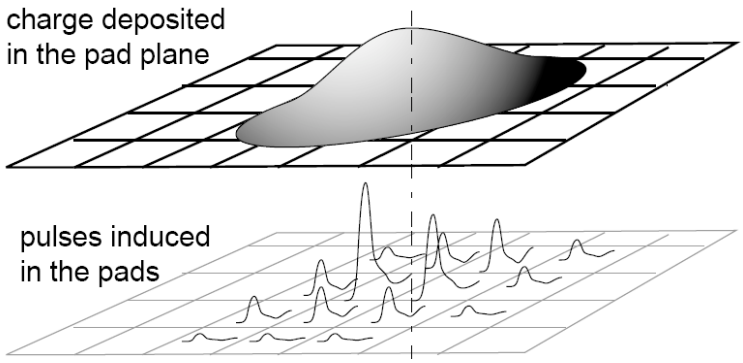


Figure 4.5.: Schematic of the charge cloud from the avalanche process in the MWPC of the ALICE TPC (top) together with the schematics of the induced signals in the detector pads underneath (bottom)

In [37] a compression efficiency of around 85%-90% is stated with a mentioned feasible loss of information.

In this work, I focus on the local channel-by-channel based compression approach since the aim is to implement the compression method in the front-end electronics. Each front-end chip is connected to only a limited number of pads (e.g. 16 or 64) and has only access to data from these limited number of channels. To implement global data compression methods in the front-end an inter-chip communication would be required to access channels in neighboring chips to guarantee the processing of full clusters, which can extend over more than the pads connected to one front-end chip. An inter-chip communication will increase the complexity of the front-end electronics and is not implemented yet in the TPC front-end. First investigations are started for a future realization of this inter-chip communication but for the time being it is more evident to focus on local compression approaches for the implementation of a compression in the front-end hardware.

In the following, the different compression methods, which are promising for a local compression approach, are discussed.

4.2. Lossless compression methods

Several lossless compression methods are investigated in order to find an algorithm that optimally compresses the digitized, zero suppressed data from a single channel in the detector front-end electronics. In the ALICE TPC front-end, the data are saved in the multi-event buffers of the ALTRO chips. The data compressor can read out the data from these buffers and compress them before they are sent out of the detector. Since the data from the detector contain important information for the HEP research and the zero suppression already causes some information loss, especially lossless compression methods are preferable to prevent additional information loss. Therefore, special emphasis is put on lossless compressions in this chapter. The compression methods are analyzed in terms of their suitability for compressing the underlying data, their compression efficiency and their feasibility for a real-time implementation in hardware.

Dictionary based methods like Lempel-Ziv coding require a large memory for the dictionary to achieve a good compression performance and therefore are considered to be too demanding in terms of resource/area requirements for a hardware implementation (resources in case of using FPGAs and area in case of using ASICs). More promising lossless methods are investigation in Matlab and are described in the following.

The raw data reduction using zero suppression can also be seen as a kind of **RLE** (run-length-

encoding) method. The samples above the threshold are saved in the MEB unchanged and all samples below the threshold are considered to have a value of 0. The number of samples with value 0 (= samples below the threshold) between two valid waveforms (samples above the threshold) give the run-length value. Therefore, the data can be saved in the MEBs consisting of the run-length number of zeros preceding the original sample values above the threshold, which are belonging to a recorded waveform. In the context of the zero suppression method, the rung-length value is named as time stamp and preserves the information of the time the waveform is seen at the detector pad. Since the zero suppression performs already a RLE a further used of RLE on the waveform samples seems to be not promising also considering that consecutive sample values in one semi-Gaussian waveform are normally not equal. A **Burrows-Wheeler Transform** (BWT) of a group of several semi-Gaussian signals could increase the probability of having several consecutive equal sample values which could make a RLE more efficient. First tests executed on the test-data in Matlab showed that a compression can be obtained by using the BWT in combination with the RLE and followed by Huffman coding. The BWT is performed on blocks of four individual semi-Gaussian waveforms from the test-data matrix in Matlab. Then the L vector from the BWT of the sample values of these four waveforms is run-length encoded and the run-length values are Huffman encoded together with the corresponding sample values of the run-length pair (nS). The index values from the BWT are as well Huffman encoded. The output data consist therefore of the Huffman coded BWT indices, the Huffman coded RLE sample values and the Huffman coded RLE run values for the L. This resulted in a compression ratio of 69%. A direct Huffman coding of the original sample values as presented later in this chapter in the diagram in figure 4.13 results in a better compression with less effort making this method not useful.

The choice of the block sized for the BWT is made taking into account a possible hardware implementation with limited amount of memory space. A block of four waveforms with 15 samples contains in total 60 sample values of 10 bit. Since the BWT requires calculating and temporally storing all possible permutations of these blocks the memory size has to be chosen in order to store 60^2 samples of 10 bit resulting in a required size of 4.4 kB. In addition to this memory also the Huffman code tables for the RLE and the index have to be stored resulting in a memory requirement larger than 7 kB.

Another compression algorithm using the BWT is known as bzip2. The bzip2 performs a Burrows-Wheeler transform followed by a move-to-front encoding and then a Huffman coding.

The **Move-to-Front** encoding method can be used to exploit local frequencies of symbols (consecutive equal symbols). Consecutive sample values in the case of the TPC are normally not equal to each other since they are belonging to a semi-Gaussian waveform and each recorded semi-Gaussian signal can have a different width and amplitude. The TPC data offer not a good concentration property making the move-to-front method by itself not very efficient. In combination with the BWT as in bzip2 the efficiency of the move-to-front method can be increased. The move-to-front method is tested in Matlab on the L of the BWT of the test-data (BWT block of 4 waveforms). A memory is initialized with all possible sample values from the test-data and then the move-to-front method sent out the indices of the sample values in L found in this memory by updating the memory according to the move-to-front rule. The indices from the move-to-front encoding and the one from the BWT are Huffman encoded. The bzip2 method with BWT, move-to-front encoding and Huffman coding resulted in a compression ratio of 65%. This is better as by using the BWT together with the RLE, but still worse than the direct Huffman encoding of the original samples values as presented later (see figure 4.13). The memory requirements are similar to the BWT with the RLE.

An additional RLE on the move-to-front indices from the bzip2 approach before the Huffman encoding showed a worsening of the compression efficiency compared to the bzip2 approach.

This first investigation showed that a more promising compression approach is to exploit the overall frequencies of sample values in the entire data by entropy coding methods as the Huff-

man encoding and the arithmetic encoding, instead of exploiting local frequencies. The overall frequencies are advantageous because several samples distributed in the entire test-data can have equal values resulting in a high frequency (probability). The frequencies of the different sample values can be estimated by using representative data of previous measurements to obtain the probability model.

This probability model is then used by the so-called statistical methods (entropy coding methods) to encode each sample in the data set with a variable length codeword regarding its probability. The theoretical limit to which the statistical methods can compress a data set is given by the Shannon entropy. The Shannon entropy gives the average uncorrelated amount of information contained in the symbols of the data set and is defined as:

$$H = -\sum_{i=1}^{n} p_i \times log_2(p_i) \tag{4.2}$$

The probability of a symbol i is represented by p_i in the formula. The term $log_2(p_i)$ represents the theoretical best codeword length for the symbol i. The entropy for the test-data used in Matlab to investigate the compression methods results in:

$$H = 6.24\,\mathrm{bit/symbol}$$

Regarding the statistical methods mentioned in 3.1 the **Tunstall coding** would be advantageous in terms of a simple hardware implementation, because it produces fixed length codewords, which are easier to handle in hardware as variable length codewords. The problem is that for a good performance the Tunstall coding method needs to create a large code tree. The sample values can have theoretically 1024 (10 bit) different values. A simple Tunstall tree with four levels would require already a codebook of 4093 entries with 12 bit codewords and will be not enough for a good compression performance. The larger the codebook is the more memory is required in the hardware implementation, which increases significantly the resource/area requirements and the power consumption of the data compression block. Other statistical methods as Huffman coding require a codebook of maximum 1024 entries with a maximum codeword length of 22 bit (as it will be shown in the following).

The Huffman coding is similar to the **Shannon-Fano coding** but it generates always optimal codes. Therefore, the Huffman coding is preferable to the Shannon-Fano coding because it produces equal or better compression performances. The most promising statistical compression methods are therefore the Huffman coding and the arithmetic coding, which are easily applicable on the sample values of the waveforms in the test-data and provide a feasible complexity for a hardware realization. These two compression methods are discussed in detail in the following.

4.2.1. Simple Huffman coding

As stated before, the entropy of the used test-data results in average 6.24 bit/symbol. The sample values are represented with a 10 bit resolution from the ADC in the ALTRO chip. This gives a compression ratio of 62.4%, which can be maximally achieved by statistical compression methods using the probabilities of the sample values.

To perform a Huffman coding of the samples of the 10 000 semi-Gaussian waveforms in the test-data matrix in Matlab first the Huffman codewords for the different samples have to be defined. The first step is to calculate the probabilities of the different sample values in the test-data matrix. To calculate the probability of a sample value the frequency is determined, which represents the number of times the value is contained in the test-data matrix. The frequency value is then divided by the total number of sample values in the test-data matrix as shown in equation 4.3:

$$p_i = \frac{Freq_i}{15 \times 10000} \tag{4.3}$$

The probabilities of the different sample values are combined to a probability model. In a hardware implementation, this probability model can be calculated either inside the front-end electronics or offline by using representative data from previous measurements.

Using the probabilities (or the frequencies), the Huffman tree can be build, which gives the ideal codeword lengths for the different samples in the test-data matrix. For example the most common sample has a frequency of 9283 and a probability of 6.19% and gets assigned a codeword of 3 bit.

If the codewords are determined, the samples in the test-data matrix are encoded one by one with the corresponding codeword and the single bits are saved from the developed Matlab function in a text file. The Huffman compression of the underlying test-data resulted in an output data file containing 939 896 bit. This gives a compression ratio of:

$$\begin{aligned} comp.\,ratio &= \frac{size\,of\,output\,stream}{size\,of\,input\,stream} \\ &= \frac{939\,896\,\text{bit}}{10\,000\,\text{waveforms} \times 15\,\text{samples} \times 10\,\text{bit}} \times 100 = \frac{939\,896}{1\,500\,000} \times 100 \approx 62.7\% \end{aligned} \tag{4.4}$$

This compression ratio is close to the Shannon entropy using only the probabilities of the different sample values and not exploiting the relation between neighboring samples. There is a dependence of neighboring samples of a waveform regarding the signal shape and this shape is known because the pre-amplifier and shaper chip PASA in the front-end of the ALICE TPC defines it. This known dependency of neighboring samples introduced by the waveform shape can be considered by a compression method to improve the compression efficiency further than the entropy limit of this kind of data.

The first approach to use the dependency of neighboring samples is to calculate the differences between them. Similar waveforms should produce similar differences between consecutive samples. An example of resulting differences of a waveform from the test-data matrix is shown in figure 4.6.

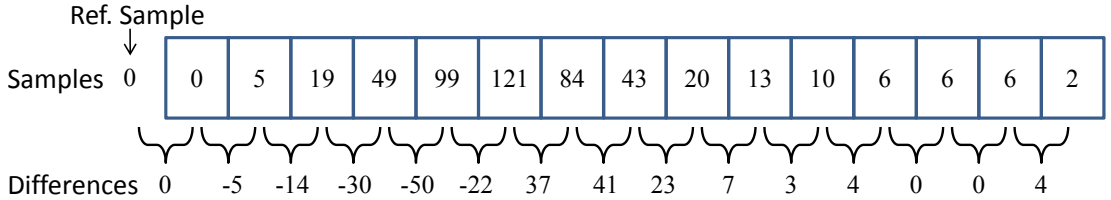


Figure 4.6.: Example of the difference values between neighboring samples of a waveform from the test-data matrix

The first sample of each waveform is compared to a reference sample of value 0. This is required to preserve the original first sample value that is used by the decoder as the starting value to reconstruct the original waveform from the differences. The calculation of the entropy of the differences of the test-data results in average 5.87 bit/symbol (58.7%). Huffman encoding the difference values instead of the original sample values of the test-data matrix results in a compression ratio of 59%. The compression of the differences results in a better compression as by encoding the original sample values, because it makes use of the dependencies of the samples according to the signal shape.

Another approach to use the correlation between the samples is to use several different Huffman codebooks. For the first sample of each waveform, the general codebook is used based on the statistical model. For the following samples, several different codebooks are developed based on the value of the preceding sample. For example, the second sample can be encoded by selecting

one out of four different codebooks. The four codebooks are obtained by dividing the range of the first sample (0...1023) in four parts and optimizing the probabilities of the second sample value according to these ranges. Then the corresponding codebook for encoding the second sample value is chosen according to the range in which the first sample is contained. If for example the first sample has a value contained in the first range the probabilities of possible values of the second sample are different then if the first sample would be contained in the second range. The multiple statistical models can also be used for an arithmetic coding instead of Huffman coding. This encoding scheme with entropy coding based on multiple statistical models (codebooks) is described in [38] and [39]. The encoding scheme named as Temporal Correlation (TC) in the paper of [38] achieves a compression ratio of 54% for the sample values as stated in table II of the paper. This compression ratio resulted from using 20 different codebooks (code tables), which makes it quite complex for a hardware implementation requiring a huge memory space to store all these codebooks and is therefore not considered here.

An even better lossless method, which uses the dependencies of the sample values based on lossless vector quantization is presented later in this chapter.

4.2.2. Simple Arithmetic coding

A second well-known and good qualified lossless compression method, which bases on a statistical model, is the arithmetic coding. In general, arithmetic coding can achieve a compression efficiency equal or better than Huffman coding. By calculating the entropy of a data set, the probability of a symbol is used to find the optimal codeword length for this symbol, which results not necessary in an integer value. The Huffman coding can assign each symbol a codeword with only an integer number of bits, which results in a compression ratio higher than the entropy. The property of the arithmetic coding to encode a stream of several symbols with one codeword reduces this problem and can result in a compression ratio closer to the entropy. The samples in the test-data matrix are encoded by arithmetic coding all in one floating-point number using the same statistical model as for the Huffman coding. In a real application, the waveforms generated in one event in the detector can be encoded in one large floating-point number. A large number of bits are required to represent this floating-point number with the required precision. The high precision for the floating-point number is crucial in order to guarantee that the arithmetic decoder is able to reconstruct all samples of the encoded waveforms. In case of encoding all the test-data in one floating-point number the required number of bits is 938 605 bit. This results in a compression ratio of:

$$compression\,ratio = \frac{938\,605\,\text{bit}}{10\,000\,\text{waveforms} \times 15\,\text{samples} \times 10\,\text{bit}} \times 100 = \frac{938\,605}{1\,500\,000} \times 100 \approx 62.6\% \tag{4.5}$$

The arithmetic coding requires 1291 bit less than the Huffman coding to compress the test-data. This is closer to the entropy but no significant improvement of the compression ratio is achieved compared to the Huffman coding.

The compression of the differences of neighboring samples inside each waveform results as well in a similar compression performance as for Huffman coding with a compression ratio of 59% and 3560 bit less.

The lower complexity of the Huffman coding and the similar compression performance with the arithmetic coding, makes the Huffman coding preferable for a hardware implementation. Arithmetic coding requires some arithmetic operations as multiplications, divisions, subtractions as also stated in [39] that require quite some resources/area to be realized in hardware. In addition, the implementation of the arithmetic encoder can access only to data from a limited number of pads connected to the corresponding front-end unit what can decrease the compression performance. The Huffman coding requires only a memory to store the Huffman codebook if the codebook is

calculated off-line by using representative data. The received sample values are used as address to readout the corresponding Huffman codewords from the memory, which can be seen as Look-Up-Table (LUT).

The **PPM** method can be seen as an arithmetic coding with multiple statistical models from which one is selected based on the previous sample value as described for the Huffman coding with multiple codebooks. The complexity of the PPM with several probability models is considered too high for an actual hardware implementation.

4.2.3. Lossless Vector Quantization

A method that makes use of the correlation of the samples from a waveform is presented next. The method bases on a vector quantization. The vector quantization is normally known as lossy compression method and is similar to the scalar quantization beside the fact that a set of several input symbols are quantized together whereas for the scalar quantization each input symbol is quantized individually. To perform the vector quantization on the test-data all the samples of one semi-Gaussian waveform are considered as the elements of one input vector. For the quantization, each input vector is compared to several reference vectors stored in a codebook and the best matching reference vector is searched. The index of the best matching reference vector is then sent out. The decoder uses this index to find the corresponding reference vector in the codebook and uses it for the reconstruction of the original input waveform.

The problem is that even though this is the best matching reference vector it is not guaranteed that it is equal to the input vector. The difference between the input vector and the reference vector is seen as an error in the reconstruction of the original input data. The fact that the shape of the input waveforms is known makes it easier to define reference vectors, which match well with the input vectors. This can be expressed as the use of the correlation of the data to reduce the information loss. Nevertheless, to obtain small errors between the input vector and the reference vector a large codebook has to be created, which takes into account the different amplitudes of the input waveforms as well as other effects. On the other hand, the larger the codebook is the more bits have to be used for the index, which decreases the compression ratio. The memory required for storing the codebook in hardware increases with larger codebook size and this increases drastically resource/area requirements for a hardware implementation.

Another possibility to reduce the error is to calculate the difference between each element (sample of the waveform) of the input vector and the chosen best matching reference vector and send out these differences in addition to the index (as side-information). The additional data from the difference values reduce drastically the compression ratio. The differences can be scalar quantized before they are sent to limit the additional side-information by still keeping the introduced error small.

Otherwise, the differences can be sent unchanged which would reduce the reconstruction error to zero and make the vector quantization lossless. The advantage is that normally the differences have smaller values as the original sample values and can be represented using fewer bits. In addition, the differences can be entropy coded e.g. Huffman coded to reduce the side-information. The gain of using a vector quantization in combination with the calculation of the differences (deltas) which then are Huffman coded is that the differences normally are small values with high probabilities, which increases significantly the efficiency of the Huffman coding compared to directly Huffman code the original samples. This higher efficiency regarding the Huffman coding is only reduced slightly because the index for each reference vector has to be sent as well. A drawback of using the vector quantization in combination with Huffman coding is that in additional to the memory for the Huffman codebook a second large memory has to be provided to store the reference vectors. The performance of this method based on vector quantization and entropy coding of the differences (residuals) is presented in [40]. A compression ratio of 64% for the lossless

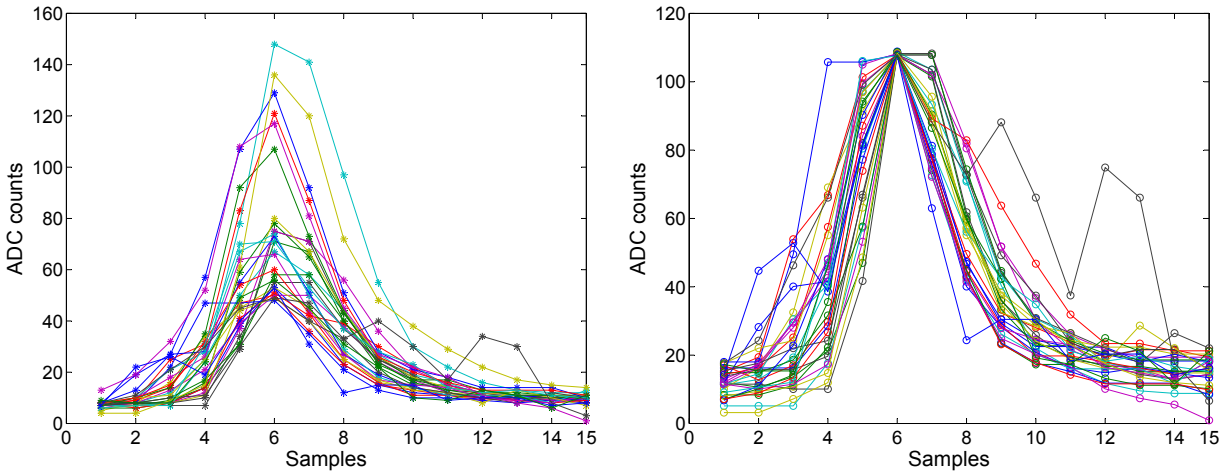
version with a codebook of 256 reference vector could be achieved for simulated TPC data from ALICE and original sample values with a resolution of 8 bit (instead of 10 bit as for the measured TPC data used in this work). The quantization of the differences before entropy coding resulted in 48% with some small loss of information as stated in [40]. Another method for a lossless vector quantization with similar results and better properties for a hardware implementation is developed and presented next.

This method uses a reduced set of maximum four reference vectors in the codebook. To avoid defining several different reference vectors for a good match to input waveforms with different amplitudes the input waveforms are first normalized in amplitude. Then their sample values are compared to the reduced set of reference vectors with equal maxima.

To perform this normalization the maximum sample value of each input waveform is searched. Then a pre-defined normalization value is divided by the maximum sample value to calculate the normalization factor as given in 4.6.

$$NormFactor = \frac{NormValue}{Max.\ Sample} \tag{4.6}$$

To obtain the normalized input waveform every sample is multiplied with this normalization factor. Figure 4.7 shows the original sample values of some input waveforms from the test-data and the normalized sample values of the input waveforms.



(a) Original sample values

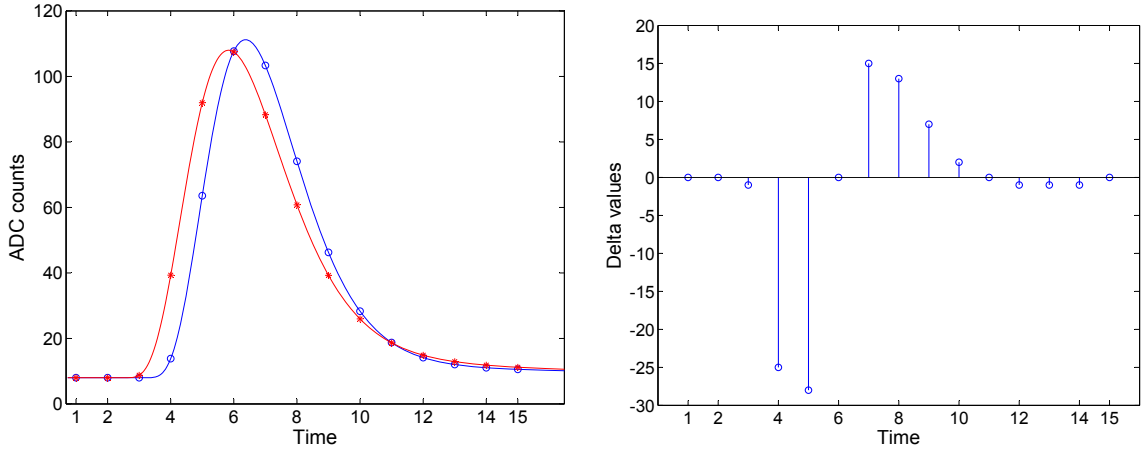
(b) Normalized sample values

Figure 4.7.: Samples of first 30 input waveforms in the test-data matrix in original form and the normalized sample values

The calculated normalization factor for each input waveform has to be added as side-information to the output data, in order to reconstruct the original sample values and the original waveform amplitudes at the decoder. Then the lossless vector quantization is performed on the normalized input waveforms.

After the normalization of the input waveform, the resulting input vector is compared to the reference vectors in the codebook and the best matching reference vector is searched. Since the amplitude of the normalized input waveforms corresponds to the maximum of the reference vectors, the quality of the matching depends only on the different shapes of the input waveforms. Different reference vectors can be calculated to cover different effects on the input waveforms e.g. different signal durations (FWHM) or pile-up effects. After the best matching reference vector is found the corresponding index is add to the output data and the differences to the normalized input waveform (in the following named as deltas) are calculated. The figure 4.8 shows a normalized

input waveform, a reference waveform created from the elements of a reference vector, and the resulting delta values.



(a) Normalized input waveform (blue o) vs. reference waveform (red *)

(b) Delta values for corresponding example

Figure 4.8.: Example of a normalized input waveform and a reference waveform constructed from the elements of a reference vector. The resulting delta values at the right have to be transmitted to prevent information loss.

To realize the normalization step in hardware an integer divider and a multiplier are needed in the implementation, which require some resources/area depending on their resolution. On the other hand the memory space for the reference vector codebook is drastically reduced (e.g. from 256 to 4) by still obtaining small differences between input vectors and the chosen reference vectors.

The hardware realization of the normalization step can introduce some error in the reconstruction of the original sample values, because the division and multiplication implementation have a limited precision. The precision of the divider and multiplier in hardware depends on the used number of bits to represent their results, which determines the resource/area requirements as well as clock speed limitations for the implementation. The truncation of the normalization factor and the multiplier results to a defined number of bits can cause rounding errors in the normalized and reconstructed sample values. To make the transmitted number of bits for side-information of the normalization independent from the choice of resolution for diver and multiplier the maximum sample value per waveform can be included in the output data instead of the normalization factor. In this way, the additional side-information is fixed to the resolution of the samples of the input waveforms and the decoder can recalculate the normalization factor from the received maximum sample in the same way as the encoder. The resolution of the multiplication and division step in the encoder and decoder can be chosen independent from the defined transmission format and can be adjusted to the requirements of the corresponding target application.

The algorithm intended for the implementation represents the elements of the reference vectors by integer values and after the normalization step, the normalized samples are as well rounded to integer values. These introduce already a small round-off error and therefore no further quantization of the delta values is done. The delta values are directly encoded using Huffman coding. The analysis of the encoding of the delta values showed that the performance of using arithmetic coding or Huffman coding is almost equal resulting in the same compression ratio. Therefore, the Huffman coding, which is simpler to realize in hardware is chosen as entropy coding method for the delta values.

The steps of the developed lossless vector quantization algorithm are summarized as follows:

- Normalization of the input waveforms
- Vector quantization: finding the best matching reference vector

- Delta calculation: calculating the differences between the normalized sample values of the input waveform and the elements of the best matching reference vector
- Huffman coding of the resulting delta values (no further scalar quantization)

The content of the output data including the side-information and all pre-information, which are necessary for the decoder, can be summarized as follows:

Transmission data:

- Maximum sample value of each input waveform, which is used to calculate the normalization factor at the encoder and decoder
- Index of chosen reference vector (can be omitted if only one reference vector is contained in the codebook)
- Huffman coded delta values

Predefined data:

- The $NormValue$ to which each input waveform is normalized has to be known to encoder and decoder
- The codebook, which contains the reference vectors for the vector quantization has to be predefined and used by encoder and decoder
- The position of the maximum sample in the reference vector is used to alien properly the normalized input vector with the reference vector for the delta calculation
- The codebook for the Huffman coding which contains the Huffman codewords and their lengths has to be used by encoder and decoder

Creation of the codebook for the vector quantization

To find the reference vectors for the codebook of the vector quantization an analysis of the test-data is performed. The first approach is to define only one reference vector for all the test-data and analyze the achievable compression performance. The easiest way to define a reference vector is to calculate the mean sample values of the input waveforms. Therefore, first each waveform in the test-data matrix has to be normalized. The $NormValue$ to which the input waveforms are normalized is obtained by calculating the mean of the maximum samples of all waveforms. Then the normalization factors are calculated by dividing the $NormValue$ with the maximum sample values and multiplying each sample of an input waveform with the corresponding normalization factor. The resulting normalized waveforms are saved in a matrix called NormVectors taking care that the maximum samples are all in the same column of the matrix as it is the case also for the test-data matrix. In this way the waveforms are aligned as it can be seen in 4.7.

$$\begin{pmatrix} NS1_{W1} & \dots & NS5_{W1} & NS6_{W1}(Max) & NS7_{W1} & \dots & NS15_{W1} \\ 12 & \dots & 47 & 108 & 102 & \dots & 15 \\ 15 & \dots & 102 & 108 & 78 & \dots & 19 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ NS1_{W10000} & \dots & NS5_{W10000} & NS6_{W10000}(Max) & NS7_{W10000} & \dots & NS15_{W10000} \end{pmatrix} \tag{4.7}$$

The symbol $NS1_{W1}$ stands for the first normalized sample ($NS1$) of the first waveform ($W1$) in the NormVectors matrix. The second and third row of the matrix 4.7 shows the values of the normalized samples of the second and third waveforms normalized to 108. The reference vector can be obtained by calculating the mean of the sample values in each column. The samples in a column are representing all samples of the 10 000 waveforms in the same position aligned with the maximum sample. Each normalized input vector of the NormVectors matrix can be aligned with the reference vector by using the position of the maximum element of both and the delta values can be obtained by calculating the differences between the elements in the corresponding positions

of both. A detailed description how this alignment is realized in hardware for zero suppressed waveforms with different lengths is explained in chapter 5.

Another approach to find an ideal reference vector is to perform an up-sampling of the waveforms in the test-data. This up-sampling can be done by fitting the samples of the waveforms with a semi-Gaussian function. From the fitted oversampled waveforms an average waveform is calculated, which then is again down-sampled to obtain the elements of the reference vector. In this way, maybe a more suitable reference vector is found by using representative data from previous measurements. The fit-command in Matlab called "lsqcurvefit" is used to perform a data fit of the samples of each input waveform. The fit-function which is used for the fitting with lsqcurvefit bases on a gamma-4 function as given in 4.8 [41].

$$f(t,\tau) = t_{tp}\, n!\, e^n \left(n - \frac{t_{tp}}{\tau}\right)^{-(n+1)} \left(e^{-t/\tau} - e^{-nt/t_{tp}} \sum_{m=0}^{n} \frac{1}{m!} \left[\left(n - \frac{t_{tp}}{\tau}\right) \frac{t}{t_{tp}}\right]^m\right) \tag{4.8}$$

The variable t_{tp} stands for the peaking time of the signal and n is the order of the function which is 4 for a gamma-4 function that produces a semi-Gaussian shaped signal. The used fit function is given by several exponential terms of $f(t,\tau)$ as shown in equation 4.9. The more exponential terms are used the better the function can model special effects in the signal, like for example a small after-pulse or undershoot on the tail of the semi-Gaussian signal created from slow moving ions (third term). For the fitting of the samples in the test-data with a semi-Gaussian shape two exponentials showed to be enough (no after-pulsing is considered). Two time constants are used in the two exponentials being τ_1 and τ_2. The constant τ_1 is dedicated to the signal part induced by the fast moving electrons in the MWPC, whereas τ_2 is dedicated to the signal part induced by the slower ion movements. The variable t is replaced in the equation 4.10 with $t - t_0$ to account for a time shift of the rising edges of some signals.

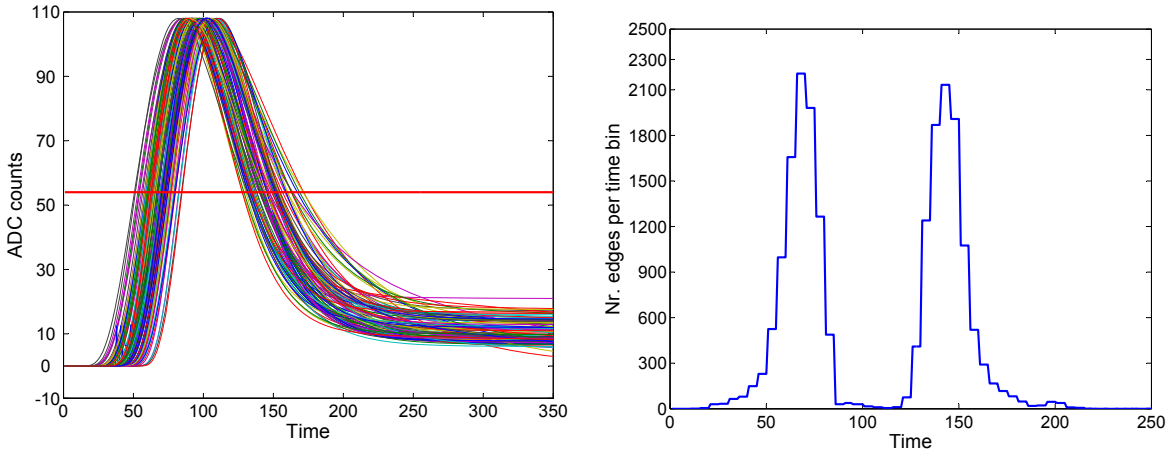
$$F_{fit} = A_1 f(t,\tau_1) + A_2(f(t,\tau_2)) \tag{4.9}$$

$$\begin{aligned} = {} & A_1 \times \left\{ t_{tp}\, 4!\, e^4 \left(4 - \frac{t_{tp}}{\tau_1}\right)^{-(4+1)} \left(e^{-(t-t_0)/\tau_1} - e^{-4(t-t_0)/t_{tp}} \left(\left[\left(4 - \frac{t_{tp}}{\tau_1}\right) \frac{t-t_0}{t_{tp}}\right]^0 + \right.\right.\right. \\ & \left[\left(4 - \frac{t_{tp}}{\tau_1}\right) \frac{t-t_0}{t_{tp}}\right]^1 + \frac{1}{2}\left[\left(4 - \frac{t_{tp}}{\tau_1}\right) \frac{t-t_0}{t_{tp}}\right]^2 + \frac{1}{6}\left[\left(4 - \frac{t_{tp}}{\tau_1}\right) \frac{t-t_0}{t_{tp}}\right]^3 + \\ & \left.\left.\left.\frac{1}{24}\left[\left(4 - \frac{t_{tp}}{\tau_1}\right) \frac{t-t_0}{t_{tp}}\right]^4\right)\right)\right\} + \\ & A_2 \times \left\{ t_{tp}\, 4!\, e^4 \left(4 - \frac{t_{tp}}{\tau_2}\right)^{-(4+1)} \left(e^{-(t-t_0)/\tau_2} - e^{-4(t-t_0)/t_{tp}} \left(\left[\left(4 - \frac{t_{tp}}{\tau_2}\right) \frac{t-t_0}{t_{tp}}\right]^0 + \right.\right.\right. \\ & \left[\left(4 - \frac{t_{tp}}{\tau_2}\right) \frac{t-t_0}{t_{tp}}\right]^1 + \frac{1}{2}\left[\left(4 - \frac{t_{tp}}{\tau_2}\right) \frac{t-t_0}{t_{tp}}\right]^2 + \frac{1}{6}\left[\left(4 - \frac{t_{tp}}{\tau_2}\right) \frac{t-t_0}{t_{tp}}\right]^3 + \\ & \left.\left.\left.\frac{1}{24}\left[\left(4 - \frac{t_{tp}}{\tau_2}\right) \frac{t-t_0}{t_{tp}}\right]^4\right)\right)\right\} \end{aligned} \tag{4.10}$$

To perform a fit the right starting values have to be given to the variables t_0, A_1, A_2, τ_1 and τ_2. In general the initial values are chosen as $t_0 = 800\,\text{ns}$, $A_1 = 100\,\text{ADC counts}$, $A_2 = 10\,\text{ADC counts}$, $\tau_1 = 100\,\text{ns}$ and $\tau_2 = 1000\,\text{ns}$. The peaking time t_{tp} was set to 224 ns. For some input waveforms different sets of initial values have to be used mainly changing the t_0. [42]

In the case, the calculated reference vector build by using the fitted input waveforms resulted in an almost equal compression performance then by using just the mean values of the input samples.

Further analysis of the fitted input waveforms are used for defining additional reference vectors and extending the codebook to increase the compression performance. A histogram from the distribution of the rising edges and falling edges of the fitted waveforms showed that there is a certain time variance in the sampling position of the signals. The oversampled and normalized input waveforms are shown in figure 4.9(a). The histogram in figure 4.9(b) shows the distribution of the points where the rising and falling edges of the waveforms crossed the red line which is at half the maximum.



(a) Fitted, oversampled, normalized input waveforms

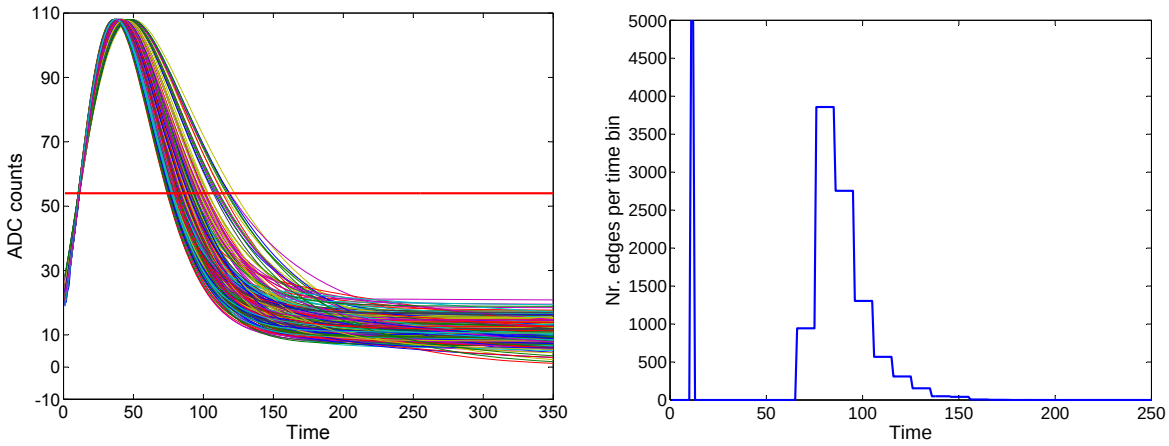
(b) Histogram of the distribution of the rising and falling edges

Figure 4.9.: The fitted input waveforms are oversampled and normalized. The distribution of the points where the rising and falling edges of the waveforms cross the red line is shown in the histogram. The red line is positioned at half maximum.

From the histogram, it can be seen that the rising and falling edges are widespread. The sampling of the analogue shaped input signal from the ADC causes this time variance because the sampling is not synchronized with the arrival time of the signals at the detector pads. The 10 MHz ADC clock is received from the RCU unit that derives it from the 40 MHz LHC clock using a frequency divider. The LHC clock is a general clock, which is not synchronized with the drift time of the electrons in the gas volume of the ALICE TPC. This variance in the sampling time causes a mismatching of the input vectors containing the normalized input waveforms with the reference vectors, which produces larger delta values. The larger delta values limit the compression performance of the lossless vector quantization.

A second analysis with the fitted normalized waveforms aligned in time on their rising edges (as illustrated in 4.10(a)) revealed that also the signal widths (FWHM) are not constant among the input waveforms. A second histogram shown in 4.10(b) on the time aligned waveforms shows that even if the rising edges are aligned, the falling edges are still distributed in a large range. This variation in the signal widths can be explained by different time spreads of the primary electrons depending on their drift distance in the gas volume causing an avalanche in the MWPC of different duration.

The distribution of the falling edges shows four main ranges of the signal widths. This discovery is used to define four reference vectors to match waveforms with different FWHMs consistent with these four ranges. The reference vectors are calculated by using the indices of the fitted normalized waveforms contained in each of the defined ranges. The indices are used to segment the test-data matrix in the corresponding waveforms belonging to each range and calculating the mean values of the samples of the waveforms in each segment. The mean values represent the



(a) Time aligned, fitted, normalized input waveforms

(b) Histogram of the distribution of the rising and falling edges

Figure 4.10.: The fitted input waveforms are oversampled and normalized and time aligned on their rising edge. The distribution of the points where the rising and falling edges cross the red line is shown in the histogram. The red line is positioned at half the maximum.

elements of the reference vector dedicated to the corresponding range. In this way, a first set of four reference vectors is obtained. The reference vectors are then optimized using the LBG algorithm. Each input vector is compared to all the four reference vectors and the delta values are calculated. To determine the best matching reference vector the distance between the input vector and the reference vector is calculated. There are different norms, which can be used to calculate the distances, the most common ones are the Manhattan distance L^1 and the Euclidian distance L^2, which are explained in section 3.2.1. The Manhattan distance is the simpler one to realize in hardware and used here for the LBG optimization. The L^1 norm is given by:

$$d_1(In_{Vec}, Ref_{Vec}) = \sum_{i=1}^{n} |In_i - Ref_i| \tag{4.11}$$

The variables In_i and Ref_i represent the values of the input vector and the reference vector, respectively. After the distances between the input vector and the four reference vectors are calculated the minimum of the distances is searched and the input vector is included in the set dedicated to the reference vector with the minimum distance. After all 10 000 input vectors in the test-data are grouped in the four sets a new set of reference vectors is defined by calculating the mean of all the input vectors in each set. After the new reference vectors are calculated, a next iteration of the LBG optimization is performed until the optimal reference vectors are found. The reference vectors resulting from the test-data are shown in figure 4.11.

The optimized four reference vectors are then used to perform a lossless vector quantization with delta calculation and Huffman coding. Each input vector containing a normalized input waveform is compared to the four reference vectors and four sets of delta values are calculated. The Manhattan distance is calculated from each set of delta values by summing up the absolute delta values in the set. The minimum of the four resulting Manhattan distances is determined and the best matching reference vector is found. The index of this reference vector is added to the output data. The corresponding set of delta values, which resulted the minimum distance is then send to the Huffman coder and the Huffman codewords for the delta values are concatenated to an output bitstream.

The Huffman codebook is optimized for the resulting delta values from the test-data. If the correct maximum sample of each input waveform is found and the normalization is performed correctly the maximum resulting delta value between the input vector and the best matching reference

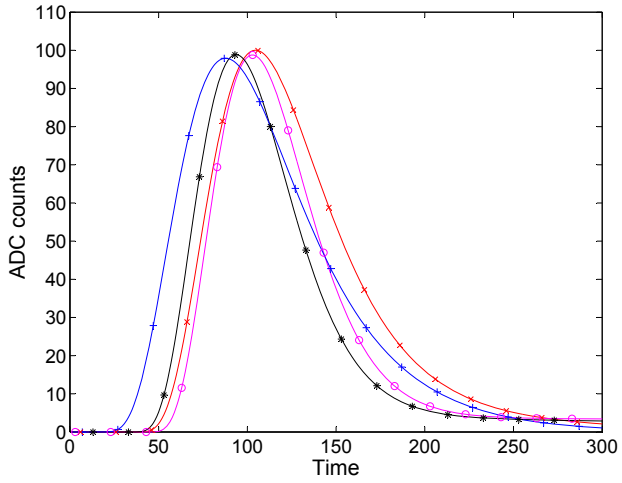


Figure 4.11.: The four reference waveforms obtained by semi-Gaussian fitting the elements of the reference vectors.

vector can be at most $\pm NormValue$. This maximum delta value can occur for example when an input vector contains an normalized input waveform with pile-up effect resulting in a second peak from a second semi-Gaussian signal on the tail of the first signal. In this case, the position of the second peak falls in the region of the tail of the reference waveform, which is close to the baseline or already 0 giving a delta value as large as the second peak value that can correspond to the $NormValue$. The maximum possible delta defines the size of the Huffman codebook being $2 \times NormValue + 1$ codewords to cover all possible delta values. For safety reasons a larger Huffman codebook can be used including a larger range of delta values in case some wrong maximum samples are causing normalized input waveforms with higher peaks then the $NormValue$. A larger Huffman codebook needs more memory space and can lower the compression efficiency because the codeword length of the delta values change.

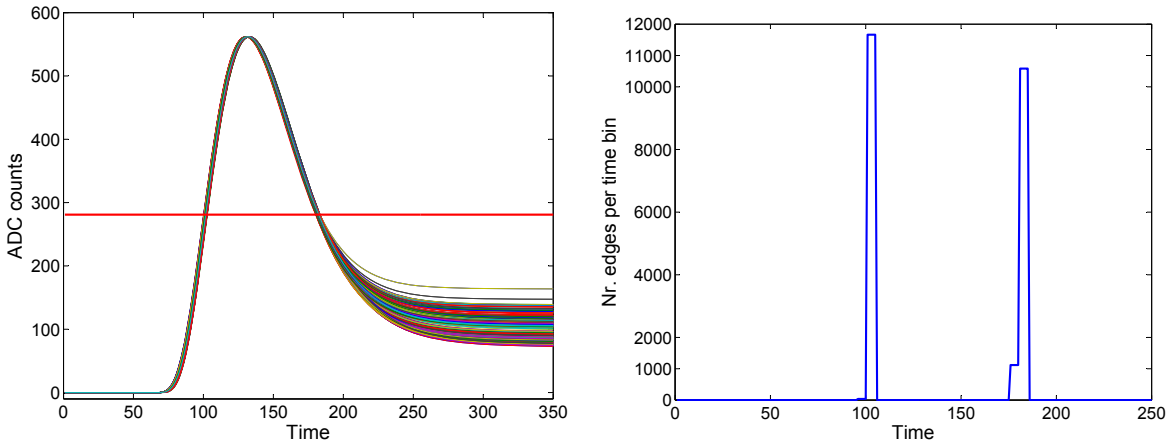
An enlarging of the vector quantization codebook with more than four reference vectors has been investigated as well. It showed that using more reference vectors for the test-data decreased not significantly the delta values and since more bits have to be used for the index of the reference vectors, the compression efficiency decreased. That leads to the assumption that four reference vectors that represent waveforms with different FWHM are optimal for the underlying test-data. The fact that the reference vectors are predefined by using representative data and then downloaded to a memory in the hardware implementation allows updating the codebook at every time to optimize the compression.

A compression ratio of 50% has been achieved with the four optimal reference vectors and the optimized Huffman codebook for the resulting deltas.

It can be said that the major limitation for the compression efficiency is related to finding good reference vectors for an optimal matching with the input vectors. Two effects worsen the matching are the time variance of the sampling of the input waveforms and the variance in the signal widths (FWHM).

A second set of data has been analyzed coming from test measurements of the ECAL detector in the CMS experiment. The advantage of this set of data is that the sampling of the detector signals is synchronized with their arrival time, which results in minor variations between the input waveforms. The normalized input waveforms have as well much more uniformity in their signal widths as it can be seen in figure 4.12.

The histogram in figure 4.12(b) shows that the rising edges and most of the falling edges are



(a) Fitted, normalized ECAL waveforms

(b) Histogram of the distribution of the rising and falling edges

Figure 4.12.: Fitted input waveforms from ECAL data that are oversampled and normalized. The distribution of the points where the rising and falling edges cross the red line is shown in the histogram. The red line is positioned at half the maximum.

crossing the red line all in the same time bin. This shows that the waveforms from the ECAL are aligned in time and have low variation in their widths. These properties are advantageous for the efficiency of the lossless vector quantization method because an adequate reference vector can be found and the input vectors of the normalized input waveforms from ECAL are matching well with the reference vectors.

The ECAL data have been scaled for a comparison with the TPC data in a way that the sample values are contained in a similar range and have similar entropy. A compression ratio of 49% has been achieved on the ECAL data using only one reference vector. That is better as the achieved 50% of the TPC data using four reference vectors.

4.2.4. Comparison of different lossless compression performance

The resulting compression ratios of the different analyzed lossless methods are summarized in the following. The results are obtained by compressing the test-data matrix using Matlab models of the different compression methods. The most interesting compression ratios are compared in the diagram in figure 4.13. The results obtained by the Burrows-Wheeler transform based compression methods, PPM and other methods giving worst compression performances as entropy coding methods are not considered as suitable for an implementation and are not listed in the diagram.

The diagram shows the percentage of the size of the compressed output data compared to the size of the uncompressed test-data. The entropy of the test-data is shown in addition.

The compression ratios of a direct compression of the sample values with arithmetic coding (Simple Arithmetic coding) and Huffman coding (Simple Huffman coding) can be compared and show that they are equal and close to the entropy. A Huffman compression or arithmetic compression of the differences between the sample values of each waveform result in a better compression ratio as by the direct compression of the sample values. This proves that a significant improvement could be achieved by exploiting the correlation between neighboring samples.

The best compression ratios are achieved with the developed lossless vector quantization method, which fist normalizes the input waveforms. The exploitation of the knowledge of the signal shape in this compression method gives the best performance using the correlation of the samples. It can be seen that using four different reference vectors could improve the compression efficiency compared to the version that uses only one reference vector. No further improvement has been discovered by using more than four reference vectors. No significant difference in performance

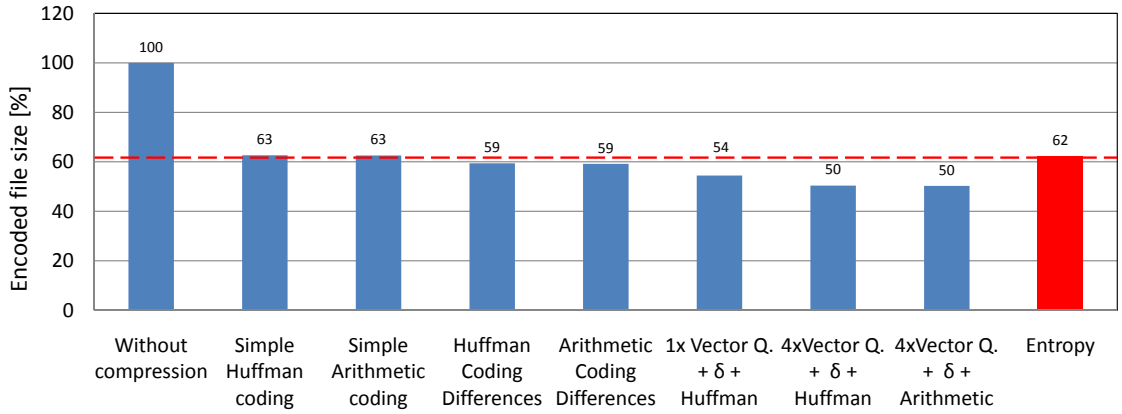


Figure 4.13.: Comparison of the compression ratios resulting from the analyzed lossless methods compressing the test-data from the ALICE TPC

between Huffman coding and arithmetic coding has been seen.

The performance of the lossless vector quantization on the more uniform ECAL data is illustrated in the diagram in figure 4.14.

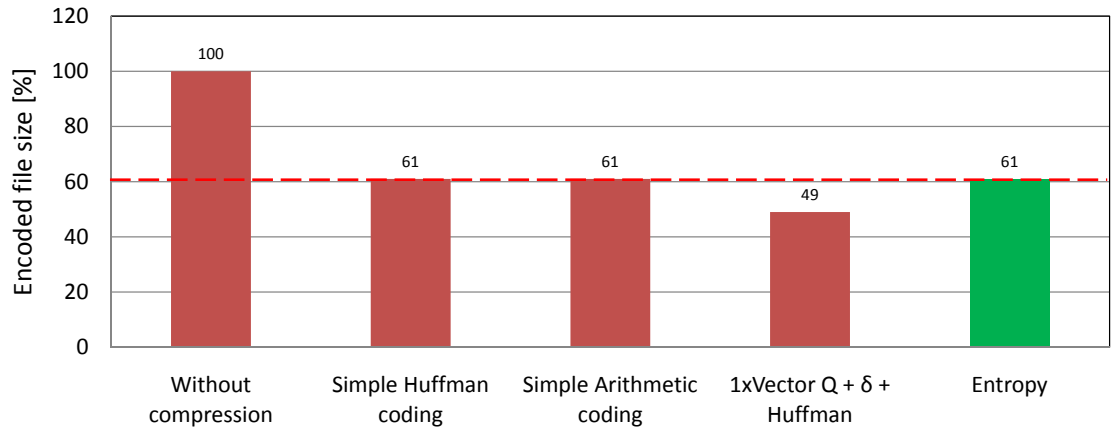


Figure 4.14.: Compression performance of the lossless vector quantization on ECAL test-data from CMS

The ECAL data show fewer variations on the signal widths (FWHM), which leads to a good compression efficiency by using the simpler version of the lossless vector quantization with only one reference vector.

The good performance of the lossless vector quantization and the good properties for hardware implementations makes it the first choice for a lossless data compression realization in the hardware of the front-end electronics of particle detectors as for the TPC in ALICE. The memory requirement of the lossless vector quantization algorithm is given mainly by tree memories:

One memory is used as buffer for the input samples of each waveform while the normalization is performed and needs a sizes of a few words of 10 bit (e.g. 64 words$\times$10 bit).

A second memory is required for the LUT of the Huffman codebook, which has to store the Huffman codewords and codeword lengths for all possible delta values. The range of possible delta values depends on the $NormValue$. The maximum Huffman codeword length seen for the test-data in the Matlab tests resulted in 17 bit. This together with 5 bit for representing the codeword length gives a word length of 22 bit per LUT memory word. Assuming a maximum range of input words for the Huffman coder of 1024 (corresponds to the range of 10 bit input sample values) this results in a maximum memory size for the LUT of 1024 words$\times$22 bit.

The third memory is used for the vector quantization codebook. For storing up to four reference vectors, which are estimated to have a maximum length of 64 elements requires a memory size of 256 words $\times$ 10 bit.

In total this gives an estimated memory requirement of 3.2 kByte. This is less then normally required for lossless compression methods, which are based on a large dictionary for a good compression efficiency as bzip2 or deflate. A first estimation of the complexity for a hardware implementation can be summarizes as follows:

The most complex part is the normalization of the input waveforms, which requires mainly some comparators for the maximum finding, an integer divider and a multiplier for the calculation of the normalization factor and the normalized samples. The vector quantization and delta calculation needs some adders (respectively subtractors) and comparators. The concatenation of the Huffman codewords can be realized with shift registers and multiplexers.

The lossless vector quantization algorithm is advantageous in terms of introduced distortion compared to lossy compression methods discussed in the following. The fact that a lossy compression is already performed in the front-end by the zero suppression and that the estimation of the effect of additional information loss on the physical parameters of the detectors is difficult for the time being, the lossless compression is the method of choice for the hardware implementation realized in this PhD work. The distortion of the samples according to the limited precision of the divider and multiplier in hardware used for the normalization is discussed after the presentation of the implementation of the algorithm in chapter 6 to include some properties and requirements of the realized implementation especially regarding the choice of the resolution of the divider and multiplier.

Nevertheless, the problematic discussion on the amount of information loss, which can be tolerated by the physics experiments an investigation of some interesting lossy compression methods are discussed in the following.

4.3. Lossy compression methods

Lossy compression methods can be more efficient in compressing certain kinds of data as lossless compression methods. In this work a compression method is searched, which provides a good performance especially on particle detector data not requiring to be a general-purpose method. Therefore, lossy compression methods can maybe use specific properties of these data to obtain a better compression ratio as the lossless methods, by having a tolerable loss of information. The detector data are used to discover new physics, which makes it crucial that no information gets lost that can reveal this new phenomena. The most important criteria of the lossy compression methods are their information loss.

Already in previous projects and documents several lossy compression methods have been discussed, which are optimized for the data of the ALICE TPC. These projects focused mainly on software solutions based on global compression of the data already sent off detector for a high-level trigger processing and data storage. Further information on this global lossy compression methods can be found in [37, 43].

In this work only a local, channel-by-channel based compression is considered, which allows a compact implementation in the front-end electronics on detector. In case of the TPC of the ALICE experiment, the digitized shaped semi-Gaussian waveforms are representing the signals in the readout channels that should be compressed. The important information contained in these waveforms is the amplitude or energy (i.e. sum of the sample values of a waveform) and the time when they have been recorded. This information should not be significantly distorted after the decompression. Some previously investigated local compression methods for a lossy compression of the TPC data can be found in [39, 40, 44]. In the following, a few more algorithms are discussed

and tested with the same test-data matrix in Matlab as used for the lossless methods.

4.3.1. Discrete cosine transform

One of the most used lossy compression methods is the discrete cosine transform for example included in the JPEG image coding. The description of the DCT method is given in section 3.2.2. The advantage of the DCT on the TPC data is that the low frequency components of the semi-Gaussian signal are concentrated in the first few resulting coefficients and can be separated from the not so important high frequency components that are small values and can be quantized. The resulting coefficients can then be Huffman coded before they are sent out. With the inverse DCT, the decoder can reconstruct the original sample values with some distortion caused by the quantization of the high frequency components.

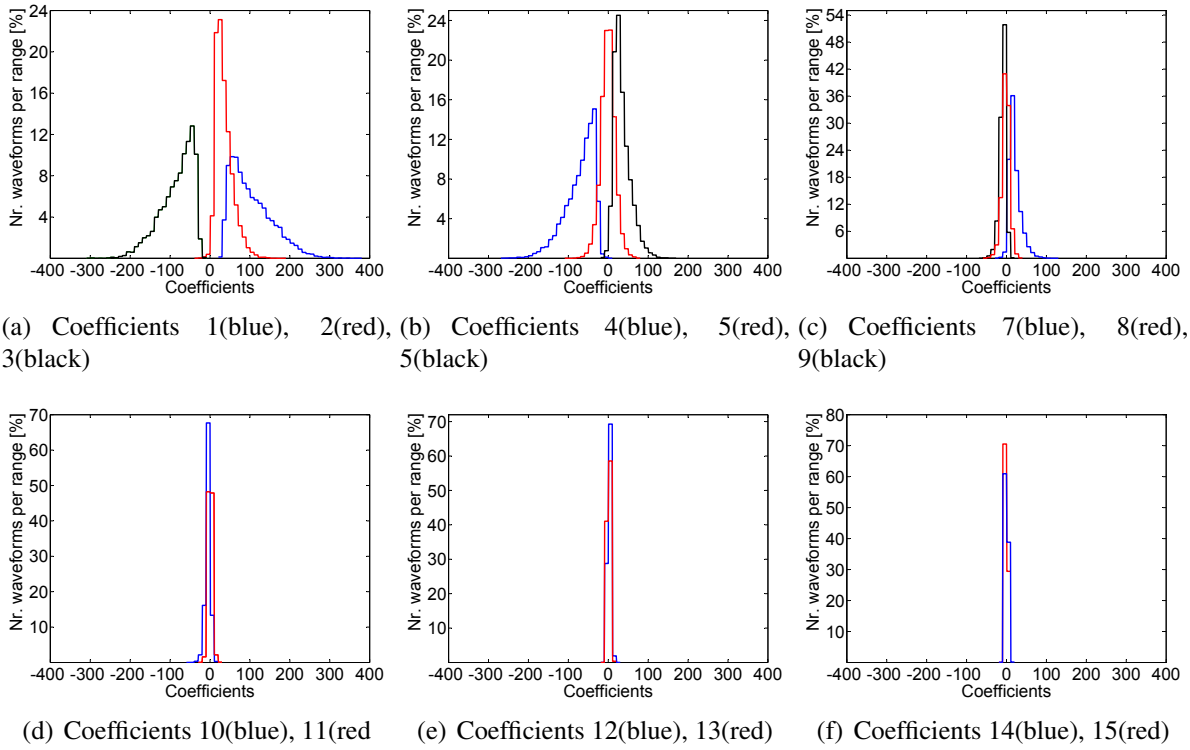
To investigate the efficiency of a DCT based compression methods different variants are realized in Matlab. First, a DCT is performed on the test-data waveforms to calculate the DCT coefficients. Then several methods are investigated to quantize and manipulate the coefficients to compress them efficiently with Huffman coding. The first manipulation of the coefficients is to round them to the nearest integer, since the DCT returns real numbers. The number of coefficients in the resulting DCT vectors is equal to the number of samples of the according input vectors representing the semi-Gaussian waveforms in the test-data. Some of these DCT vectors resulting from input vectors are shown in the matrix 4.12 that represents a part of the DCT output matrix of the 10 000 input vectors containing the test-data waveforms.

$$\begin{pmatrix} 69 & 14 & -33 & -30 & 0 & 20 & 13 & -2 & -7 & -5 & 0 & 2 & 1 & 1 & 0 \\ 88 & 14 & -55 & -36 & 15 & 31 & 8 & -14 & -10 & 1 & 6 & 5 & -1 & -4 & -2 \\ 79 & 24 & -28 & -35 & -12 & 6 & 7 & 0 & -4 & -1 & 1 & 3 & 4 & 2 & 1 \\ 96 & 20 & -56 & -39 & 5 & 19 & 10 & -2 & -3 & -2 & -1 & 1 & 2 & -1 & -1 \\ 191 & 77 & -98 & -120 & -37 & 37 & 36 & -2 & -25 & -14 & 3 & 12 & 7 & -1 & -3 \\ 226 & 94 & -112 & -129 & -36 & 25 & 16 & -10 & -11 & 3 & 12 & 7 & 0 & -6 & -6 \\ 71 & 10 & -39 & -26 & 10 & 24 & 7 & -7 & -10 & -3 & 2 & 3 & 3 & -1 & -2 \\ 100 & 19 & -68 & -48 & 19 & 42 & 11 & -18 & -14 & 1 & 7 & 4 & -1 & -3 & -2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \tag{4.12}$$

It can be seen in the matrix that the absolute values of the first coefficients to the left of the resulting DCT vectors (rows of the matrix) are higher than the absolute values to the right. The first coefficients are representing the low frequencies, i.e. if for example the waveforms are shifted by a constant value representing a DC offset only the first coefficient of each row changes. To analyze the resulting coefficients the distributions of their values according to their positions in the DCT vectors (column-by-column of the coefficients matrix) are calculated and shown in figure 4.15.

It can be seen that the values of the first nine coefficients of the coefficient vectors are varying in a larger range as the six coefficients at the end of the vectors.

If one would transmit these DCT coefficients encoding them with Huffman the original sample values could be reconstructed by producing only a very small distortion of the original values caused by the rounding of the coefficients. The drawback is that the compression ratio is 68%, which is even worst then by encoding directly the original sample values with Huffman. This can be explained by the fact that the overall coefficients are signed values distributed in a larger range as the original sample values and they are now decorrelated. The advantage of using DCT is based on its ability to produce decorrelated coefficients, which then can be quantized individually in a way that gives an acceptable low distortion of the reconstructed data by producing high compression efficiency.



(a) Coefficients 1(blue), 2(red), 3(black)

(b) Coefficients 4(blue), 5(red), 5(black)

(c) Coefficients 7(blue), 8(red), 9(black)

(d) Coefficients 10(blue), 11(red

(e) Coefficients 12(blue), 13(red)

(f) Coefficients 14(blue), 15(red)

Figure 4.15.: Histogram of the distribution of the DCT coefficients according to their position in the resulting vector per input vector.

Regarding the distributions of the coefficient values shown in figure 4.15 the first idea is to quantize strongly the last six coefficients of each DCT vector, whereas just rounding the first coefficients. The last six coefficients are distributed in a small range around 0. Therefore, an obvious quantization of the last six coefficients per input vector is performed by forcing them to 0. The compression ratio after Huffman coding of the coefficients from all 10 000 input vectors with the quantized last six coefficients to 0 results in 56%. In the domain of lossy compression, a measure called Peak-Signal-to-Noise Ratio ($PSNR$) is often used to compare the quality of the reconstructed data according to the information loss among different compression methods and quantization methods [33]. The $PSNR$ uses the Root Mean Squared Error ($RMSE$) between the original sample data (S_i) and the reconstructed data (R_i) and is defined in 4.14:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(S_i - R_i)^2} \tag{4.13}$$

$$PSNR = 20\,log_{10}\left(\frac{max_i\,|S_i|}{RMSE}\right) = 20\,log_{10}\left(\frac{2^{10}}{RMSE}\right) \tag{4.14}$$

The term $max_i\,|S_i|$ represents the maximum possible sample value which here is given by the resolution of the ADC being 10 bit for the ALICE TPC resulting in $2^{10} = 1024$. The $PSNR$ uses the logarithm and results in a decibel (dB) value. The resulting value gives a quantitative measure of the error introduced by the lossy compression with quantization but cannot qualitatively determine the effect of this error on the relevant information of the data. For example in image compression the $PSNR$ can be used to compare different compression methods or different quantization algorithms but it cannot qualitatively tell how the error affects the impression of the reconstructed image on an observer. In this document the $PSNR$ is used to compare DCT results with different quantization methods and with other lossy compression methods discussed in the following.

For the DCT of the original sample values and the quantization of the coefficients by rounding them to the nearest integer a $PSNR = 71\,\mathrm{dB}$ is obtained. For the case where a quantization of the last six coefficients to 0 is performed and the remaining ones are rounded to the nearest integer the $PSNR$ resulted in 51 dB.

To reduce further the range of the remaining not quantized coefficients they can be subtracted from a pre-defined reference vector. The reference vector is created by calculating the mean value of the coefficients in each column of the resulting coefficients matrix. The differences between the reference vector and each coefficient vector are calculated. The reference vector and a coefficient vector are shown in figure 4.16 together with the resulting differences (delta values). This is similar to the previously explained lossless vector quantization using one reference vector.

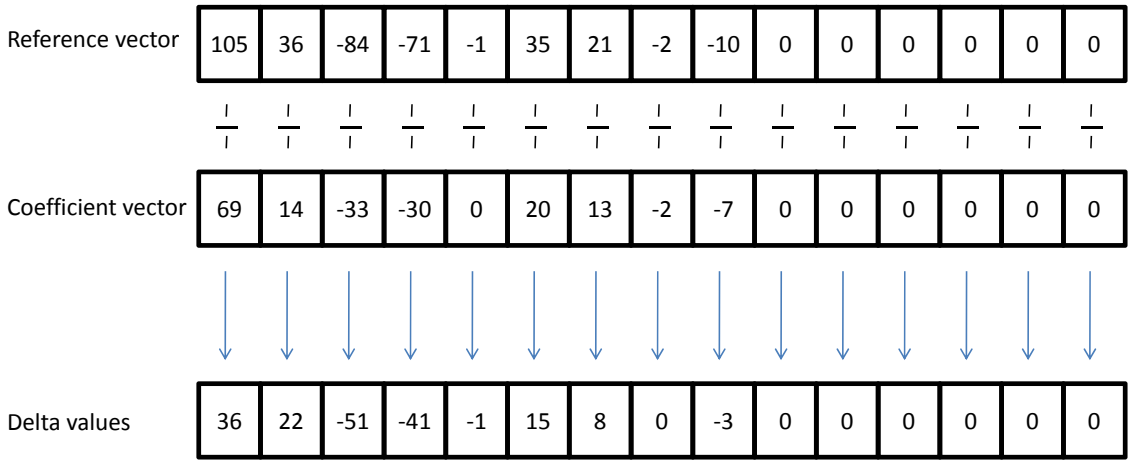


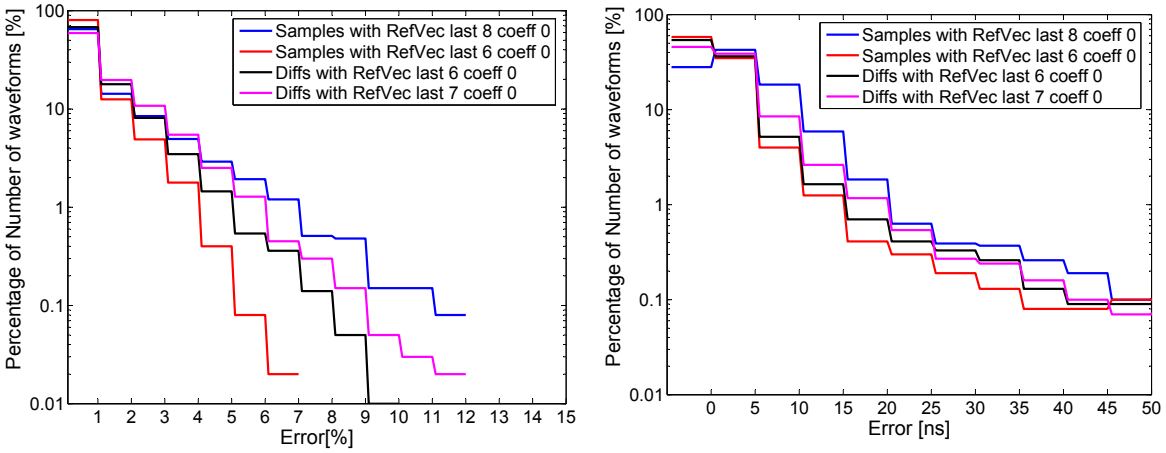
Figure 4.16.: Reference vector calculated from the mean of the coefficient vectors. One of the coefficient vectors is shown below the reference vector with the last six coefficients quantized to 0. On the bottom, the resulting delta values are given obtained from the subtraction of the coefficient vector from the reference vector

The resulting deltas between the reference vector and the coefficient vectors are then Huffman encoded. The resulting compression ratio is 50% and the $PSNR$ results in 52 dB.

If the last seven deltas per coefficient vector are quantized to 0 a compression ratio of 46% is achieved giving a $PSNR$ of 50 dB. A quantization of the last 8 deltas results in a compression ratio of 42% and a $PSNR = 48\,\mathrm{dB}$.

To determine which effect the quantization has on the reconstruction of the important information of amplitude and time stamp of each input waveform the errors in these two parameters are investigated. To calculate the error the original sample values of each input waveform are fitted using the "lsqcurvefit" operation in Matlab (see section 4.2.3). The reconstructed sample values of each waveform after the quantization and inverse DCT with the reference vector are fitted with the same function. Then the difference in amplitude and time stamp of each fitted input waveform and reconstructed waveform is calculated. The amplitude error is normalized to the amplitude of the fitted input waveforms and is given in percent. The time error is expressed in nanoseconds. The histogram of the distribution of the amplitude error and time error for different quantization cases of the DCT with reference vector are shown in figure 4.17.

The error in amplitude of the reconstructed sample values after the DCT, the quantization of the coefficients and the comparison with the reference vector is shown in 4.17(a) for the two cases of quantizing the last six coefficients to 0 (red line) and the last 8 coefficients to 0 (blue line). The last six quantized coefficients version results in around 80% of the waveforms with errors less than 1% of the original amplitude. In contrary, it can be seen that the error is significantly increased for the quantization of the last 8 coefficients to 0, having only around 68% of the reconstructed waveforms with an error less than 1%. The red line also stops by much lower error values as the blue line. This



(a) Amplitude Error (b) Time Error

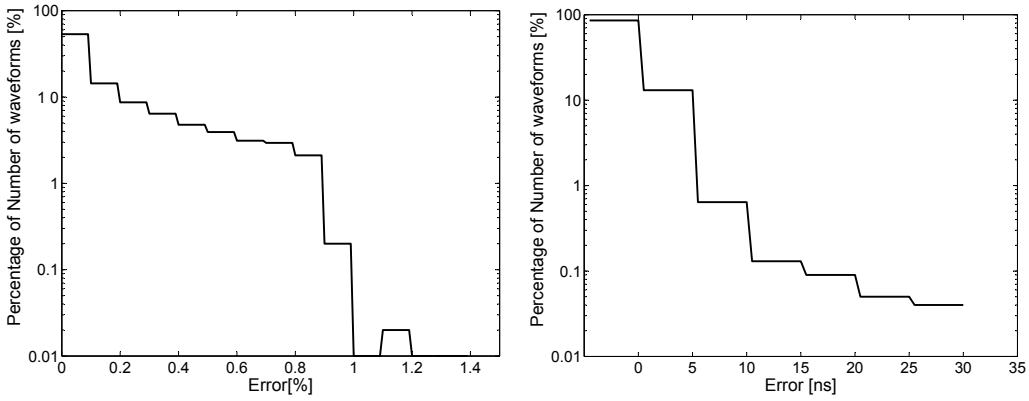
Figure 4.17.: Histogram of the error in amplitude (left) and time (right) between the original waveforms and the reconstructed waveforms from the DCT. The deviation in amplitude is given in % of the original amplitude and the error in time is given in nanoseconds. The number of waveforms corresponding to the error is given in % of the total analyzed waveforms

gives an idea of the error introduced in the important parameters caused by the quantization of the coefficients showing that quantizing the last 8 coefficients to 0 still preserves a good accuracy for more than 60% of the reconstructed waveforms and a seen maximum error of 12%. In a further consideration of this method, the limit that would be acceptable from the physics point of view has to be investigated to find the right trade-off between compression ratio and introduced error. At this moment, this limit is not known and is under investigation by analyzing the actual detector data from real collision in the LHC.

In addition two other error distributions are shown in figure 4.17 retrieved from the concept of not performing a DCT on the original sample values but on the differences between neighboring samples of each input waveform as already analyzed for the lossless compression methods. First, the differences between consecutive samples of one waveform are calculated and built the new input vector and then this new input vector is transformed with the DCT. A new reference vector is calculated from the new coefficients. The rounded coefficients vectors are subtracted from the new reference vector and the resulting values are then quantized and Huffman coded. For the first investigations, again the last six coefficients are quantized to 0. The error introduced from this quantization causes a distortion in the reconstructed differences. When the sample values are reconstructed from the differences the error in one difference value is extended to the consecutive samples of the reconstructed input waveform, which increases the distortion compared to the DCT of the original sample values. The $PSNR$ of the DCT of the differences with quantization of the last 6 coefficients to 0 results in 51 dB (1 dB less as for the DCT of the samples). On the other hand the compression efficiency is significantly increase resulting in a compression ratio of 43%. The error introduced in the amplitude and time stamp of the input waveform is shown by the black line in figure 4.17 and is worst than for the DCT of the samples represented by the red line. In addition the error of the DCT of the differences with the quantization of the last 7 coefficients to 0 is shown in magenta in figure 4.17 and is worst then the error introduced by the DCT of the samples quantizing the last 8 coefficients. The compression ratio of the DCT of the differences with the quantization of the last 7 coefficients is 39%.

At the end, a DCT is investigated performed on the Delta values of the lossless vector quantization presented above, before they are Huffman encoded to see the possible improvement of the compression efficiently. The coefficients of the DCT of the delta values are rounded and then Huffman encoded without further quantization. The rounding of the DCT coefficients introduces

an additional error to the already existing error from the limited precision of the normalization step. The curves in figure 4.18 show the effect of the introduced error on the important parameters am-



(a) Amplitude Error of the DCT of the Deltas (b) Time Error of the DCT of the Deltas

Figure 4.18.: Histogram of the error in amplitude (left) and time (right) between the original waveforms and the reconstructed waveforms from the DCT of the deltas from the lossless vector quantization method. The error in amplitude is given in % of the original amplitude and the error in time is given in nanoseconds. The number of waveforms corresponding to the error is given in % of the total analyzed waveforms.

plitude and time stamp and can be compared to the results shown in chapter 6 for the realized and implemented lossless vector quantization without DCT. Only a few percent of the reconstructed waveforms have amplitude errors slightly above 1%. The time error is also very small with more than 80% of the waveforms having no error in the time stamp.

The improvement in compression efficiency is by 3% resulting in a compression ratio of 47% instead of 50% as achieved by the lossless vector quantization without DCT. The $PSNR$ is 66.6 dB which is around 3 dB lower than for the $PSNR$ of the lossless vector quantization without DCT of the deltas resulting in 70 dB.

At the end a diagram is shown in figure 4.19 which summarizes the compression ratios of the different versions of the DCT. The $PSNR$ values for the according DCT methods are shown in figure 4.20.

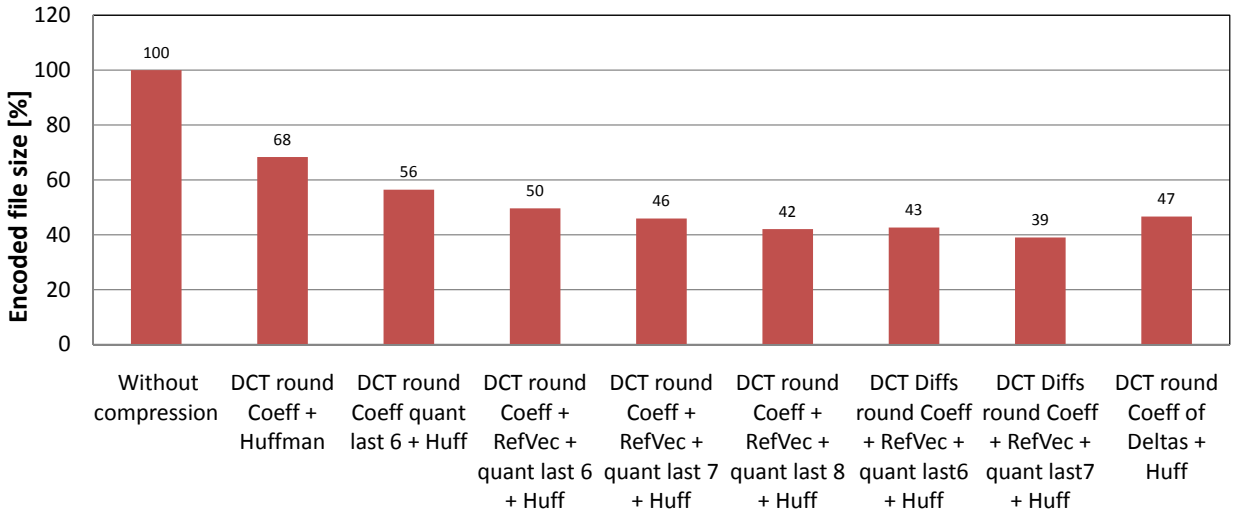


Figure 4.19.: Comparison of the compression ratios resulting from the various DCT methods performed on the test-data from the ALICE TPC

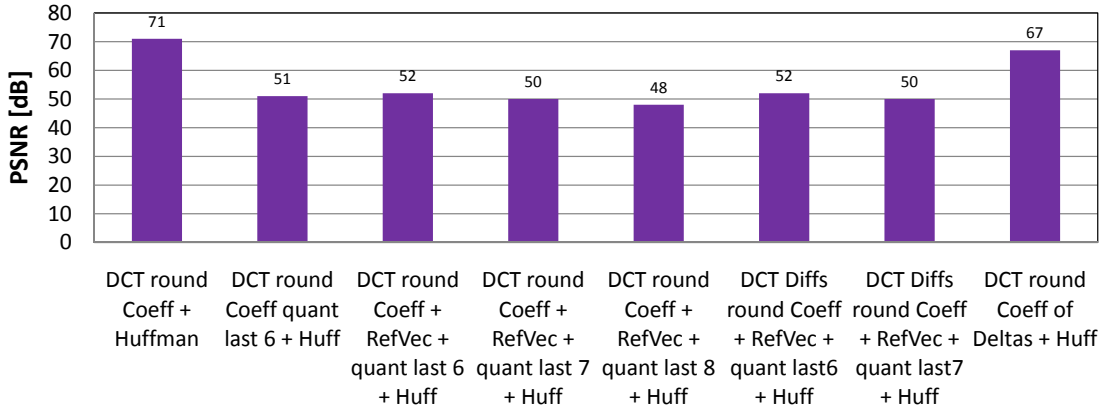


Figure 4.20.: Comparison of the peak signal to noise ratios (PSNR) resulting from introduced distortion of the various DCT methods performed on the test-data from the ALICE TPC

With the DCT methods a compression ratio down to 39% could be achieved, which is better than by the lossless compression methods but the introduced error has to be considered.

4.3.2. Wavelet transform

Another well-known transform method used in lossy data compression is the wavelet transform. The wavelet transform is used for example in image compression based on JPEG 2000. The difference of the wavelet transform compared to the DCT is that it splits the input signal in a coarse information part and a detailed information part and not into the single frequency components as the DCT does. The coarse information can be seen as a representation of the input signal with a lower resolution by calculating the averages between neighboring samples. The coarse information preserves still the shape of the input signal as it can be seen in figure 4.21.

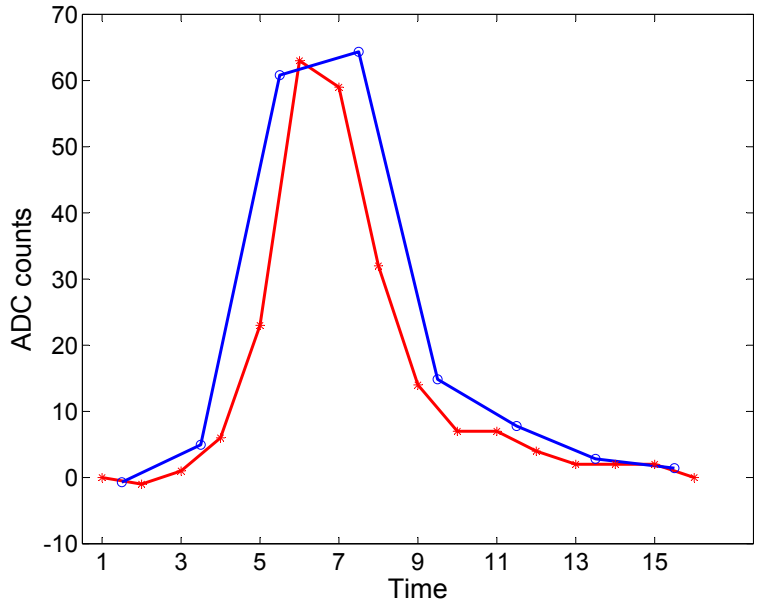


Figure 4.21.: Samples of an input waveform (in red *) versus the coarse information coefficients of the DWT (in blue o)

The detailed information is then used to reconstruct the original signal form the coarse representation and is composed by the differences between neighboring samples. If the input signal

has a strong correlation, the resulting detailed coefficients have small values, which can be quantized stronger. On the other hand, the coarse information can be used to perform first analyses to classify and select certain signals before decompressing them totally. An execution of the wavelet transform can be realized in hardware by using the Discrete Wavelet Transform (DWT) in matrix form or with filter banks. The simplest DWT is obtained from the Haar-wavelet as mother-wavelet and can be implemented by using two FIR-filters. One filter is a low-pass filter (LP) with the coefficients $h(0) = 1/2$ and $h(1) = 1/2$ used to calculated the coarse information coefficients (averages). The second filter is a high-pass filter (HP) with the coefficients $h(0) = 1/2$ and $h(0) = -1/2$ that gives the detailed information coefficients (differences). In this way, the input signal is split into two subbands containing low frequencies and high frequencies, respectively. Each of the two FIR filters produce as many output coefficients as number of sample values in the input signal, i.e. if the input signal contains 16 samples the two FIR filters produce together 32 coefficients. According to the fact that the bandwidth of the output signal of each filter is half the input bandwidth since we divided the input signal into two subbands and regarding the Nyquist criteria, only half the output coefficients are needed to reconstruct the input signal. A down-sampling block is positioned after each FIR to discard every second coefficient value in order to obtain equal number of coefficients as elements in the input vector given by the input samples per waveform. Some resulting coefficient vectors of a Haar-wavelet transform of the test-data are given in the matrix 4.15.

$$\begin{pmatrix} 11 & 16 & 64 & 45 & 19 & 16 & 13 & 12 & 0 & -4 & -11 & 12 & 2 & 0 & 0 & 1 \\ 11 & 16 & 72 & 76 & 26 & 19 & 14 & 13 & 1 & -4 & -28 & 19 & 5 & 2 & 0 & 1 \\ 11 & 40 & 69 & 42 & 20 & 14 & 14 & 12 & -1 & -4 & -2 & 9 & 1 & 1 & 0 & 1 \\ 11 & 26 & 83 & 72 & 33 & 19 & 13 & 13 & -1 & -9 & -12 & 10 & 6 & 4 & 1 & 1 \\ 16 & 88 & 215 & 110 & 39 & 25 & 21 & 14 & -11 & -18 & -24 & 36 & 5 & 3 & 1 & 3 \\ 21 & 127 & 217 & 144 & 49 & 29 & 21 & 16 & -13 & -32 & -25 & 40 & 8 & 2 & 1 & 4 \\ 11 & 11 & 59 & 54 & 23 & 16 & 14 & 11 & 0 & 0 & -19 & 11 & 3 & 0 & 1 & 0 \\ 11 & 15 & 90 & 92 & 28 & 18 & 14 & 13 & 0 & -4 & -35 & 27 & 5 & 1 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \end{pmatrix} \tag{4.15}$$

From the 16 coefficients, the first half are representing the coarse information and the other half the detailed information. The detailed information coefficients can be quantized to obtain a good compression without reducing too much the quality of the reconstruction of the signal. The coarse information, which is similar to the input signal can be transformed again using recursively the DWT. A second set of FIR filters (LP+HP) and down-sampling blocks can be connected to the output of the first low-pass filter with down-sampling block (see figure 3.14(b)). The second step produces four coarse coefficients and four detailed coefficients. The detailed coefficients are combined with the already previously calculated eight detailed coefficients and the new coarse coefficients are again DWT transformed in a next iteration. This can be executed until only one coarse information coefficient remains per input vector and the rest are detailed coefficients, which can be quantized to obtain a good compression performance. For reconstructing the original sample values, the inverse DWT has to be performed iteratively in the reverse order on the quantized coefficients.

In Matlab first tests are performed using the DWT with the Haar-wavelet in matrix form applied on the test-data matrix to investigate the compression performance and the distortion using the $PSNR$ measure. The results can be compared with the previously discussed DCT. The DWT requires that the input vectors containing the sample values of the input waveforms have a length that is a power of two. Therefore, an extension of the input vectors containing the test-data waveforms, from 15 samples to 16 values was done by adding a 0 value at the end.

The Haar wavelet transform matrix is extended to a 16×16 matrix according to the rules

described in section 3.2.2. The transform matrix elements are calculated to perform the four iterations of the wavelet transform at ones resulting in only one coarse coefficient per input vector. The remaining coefficients are the detailed values from the different resolutions (iteration steps). The product between the transform matrix and each input vector (row) of the test-data matrix is calculated resulting in the 16 transform coefficients. These coefficients are rounded to the nearest integer and can then be quantized with different methods. By Huffman coding the rounded coefficients without farther quantization, a compression ratio of 72% has been achieved with a $PSNR$ of 70 dB. The compression efficiency is lower as for the DCT (68%) because the input vectors had to be extended by one element for the power of two length requirement. The $PSNR$ on the other hand is almost equal. The extension of the input vectors is tried to be compensated by quantizing the last coefficient of each output vector to 0, which results in a compression ratio of 69% which is closer to the one from the DCT. The $PSNR$ of the DWT with the quantization of the last coefficient is 63 dB and is worst compared to the DCT. Another quantization method searches all coefficients that have absolute values of 1 and quantizes them to 0. In this case the compression ratio stays at 69% but the $PSNR$ is better resulting in 66 dB.

A calculation of a reference vector from the DWT transform coefficients can also be performed similar to the DCT resulting in the following values:

$$102,\ 68,\ -93,\ 15,\ -19,\ 24,\ 12,\ 3,\ -1,\ -16,\ -19,\ 23,\ 8,\ 2,\ 1,\ 2$$

Each DWT coefficient vector can then be subtracted from the reference vector before the resulting deltas are quantized and Huffman coded. The compression ratio using a reference vector results in 62% with the same $PSNR = 70\,\mathrm{dB}$ as by transforming directly the sample values. If the differences (deltas) between the reference vector and the transform coefficients are quantized before the Huffman encoding the compression efficiency can be increased further. The quantization of the deltas with absolute values of 1 to 0 and quantizing as well the last delta value from each vector to 0 results in a compression ratio of 56% and a $PSNR$ of 64 dB.

A quantization of the last seven deltas to 0 including the added 16th coefficient results in a compression ratio of 46% and a $PSNR = 45\,\mathrm{dB}$. This can be compared with the DCT with reference vector and quantization of the last 6 coefficients and shows that the DWT gives a better compression performance but a worst $PSNR$.

An equal compression ratio of 46% can be achieved by transforming the differences between neighboring input samples of one input waveform instead of transforming the sample values directly. From the resulting transform coefficients, a new reference vector is calculated. The differences between each coefficient vector and the new reference vector are calculated and all differences with absolute value of 1 are quantized to 0 as well as the last coefficient. The Huffman coded quantized differences are resulting a compression ratio of 46% but a $PSNR$ of 57 dB is much better as for the previous method with quantizing the last 7 deltas. This can be compared with the DCT of the differences between sample values and quantizing the last 6 coefficients. The DCT shows a lower compression ratio but also a lower $PSNR$.

The following diagram in figure 4.22 summarizes the compression ratios achieved by the DWT with the Haar-wavelet and can be compared to the results from the DCT and the lossless compression methods.

The distortion introduced in the reconstruction of the input waveforms caused by the different quantization methods of the transform coefficients is summarized in figure 4.23.

A second wavelet transform is being analyzed using the Daubechies 4 wavelet with eight filter coefficients for the LP-filter and HP-filter. This wavelet has a shape closer to the one of the semi-Gaussian waveform assuming therefore a better compression ratio with lower distortion. The achieved results can be compared to the DWT with the Haar wavelet to see if an improvement could be obtained by changing the mother-wavelet.

By transforming the input vectors of 16 samples with the Daubechies 4 wavelet (db4) and

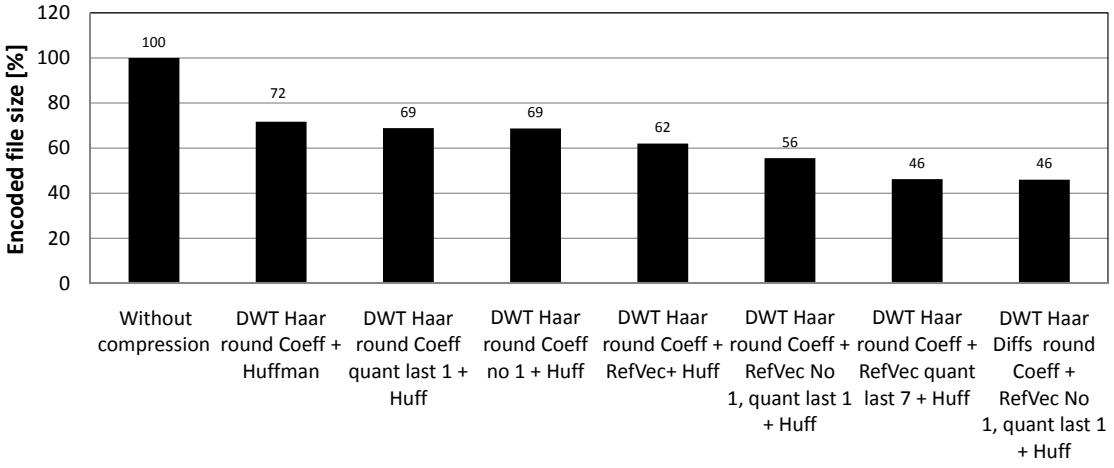


Figure 4.22.: Comparison of the compression ratios resulting from the various DWT methods with the Haar wavelet performed on the test-data from the ALICE TPC extended to 16 elements per input vector

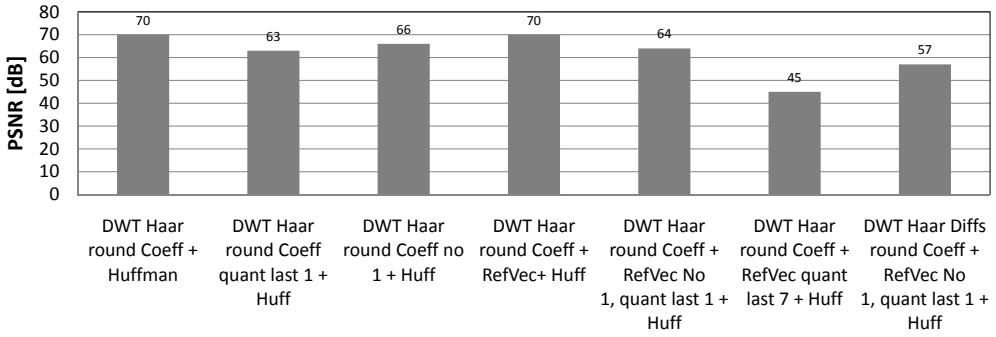


Figure 4.23.: Comparison of the peak signal to noise ratios (PSNR) resulting from the introduced distortion of the various DWT methods with Haar wavelet performed on the test-data from the TPC

rounding the resulting 16 coefficients to the nearest integer a compression ratio of 69% has been achieved. This compression ratio is much closer to the one from the DCT even with the extension of the input vectors to 16 samples as by using the Haar wavelet. The resulting $PSNR = 71\,\mathrm{dB}$ is also equal to the DCT transform.

A quantization of the coefficients with absolute value 1 to 0 and of the last coefficient of the resulting transform vectors to 0 gives a compression ratio after Huffman coding of 62% for the db4-wavelet. This is 4% better as for the Haar wavelet by producing an equal $PSNR$ of 62 dB.

The calculation of a reference vector from the transform coefficients of the db4 and using it to obtain the differences (deltas) between the coefficient vectors and the reference vector produces a compression ratio of 53% when all deltas with absolute values of 1 are quantized to 0 and the last delta value per vector is quantized to 0. This is better than the 56% of the Haar wavelet by a bit lower $PSNR$ of 62 dB (for Haar $PSNR = 64\,\mathrm{dB}$).

A quantization of the last 7 delta values per coefficient vector results in a compression ratio of 48% and a $PSNR$ of =49 dB. The compression ratio is a bit higher than for the Haar wavelet but the $PSNR$ is a bit lower.

The transformation of the differences between sample values of the input waveforms using the db4 wavelet results as well in a lower compression ratio as for the Haar wavelet (48% for db4; 46% for Haar) and also the $PSNR = 55\,\mathrm{dB}$ is lower.

The compression ratios for the DWT with the db4 wavelet are summarized in figure 4.24 and

the resulting $PSNRs$ are shown in figure 4.25.

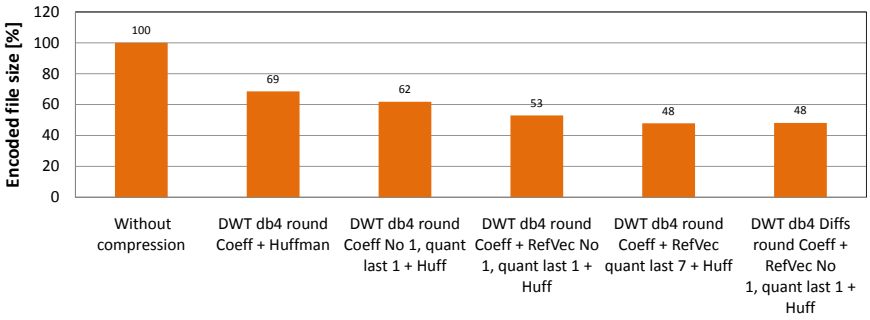


Figure 4.24.: Comparison of the compression ratios resulting from the various DWT methods with the Daubechies 4 wavelet performed on the test-data from the TPC extended to 16 elements per input vector

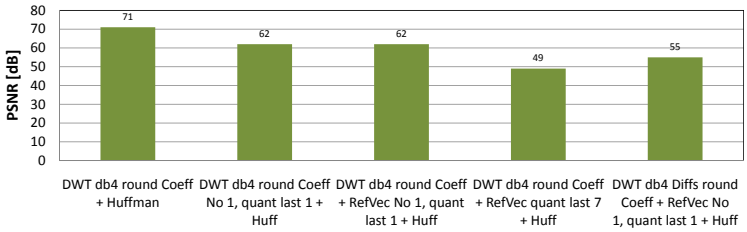


Figure 4.25.: Comparison of the peak signal to noise ratios (PSNR) resulting from the introduced distortion of the various DWT methods with Daubechies 4 wavelet performed on the test-data from the TPC

The analysis of the wavelet transform on the test-data matrix showed that it produces similar compression results as the DCT with in most cases better $PSNR$ values, which shows a reduced distortion in the reconstructed waveforms. For the DWT more caution has to be put in the quantization of the resulting transform coefficients because the coefficients with the small values are not all concentrated in the end of the resulting coefficient vectors as for the DCT.

The comparison of two different mother-wavelets, the Haar wavelet and the Daubecies 4 wavelet, showed that the right choice of the wavelet can improve the compression efficiency by keeping the same $PSNR$ or improving the $PSNR$. Another advantage of the wavelet transform is that not all coefficient values have to be decoded to investigate some simple properties of an input signal for a possible pre-selection of data before performing a total decoding for the analysis. For example after the iteratively executed wavelet-transform, the only remaining coefficient with the coarse information represents the mean value of the samples of a input waveform, which represents the energy of the signal.

A realization of the wavelet-transform in hardware requires FIR filters and down-sampling blocks, which are common blocks in digital signal processing circuits. The complexity of the implementation is defined by the number of taps per FIR-filter (given by the underlying mother-wavelet) and the number of required iterations.

The biggest disadvantage of the wavelet-transform is the requirement that the input vectors have to have a number of elements, which are a power of two. Since the input vector have various different lengths after the zero suppression in the Alice TPC this criteria can cause sometimes lower compression efficiencies due to a required extension of the input vectors.

Both transform methods, the DCT and DWT can produce higher compression efficiencies as the lossless methods if some information loss can be tolerated. A deeper investigation of the

acceptable distortion and a related optimization of the quantization of the transform coefficients has to be carried out for each target application before these methods can be implemented and used efficiently.

4.3.3. Model based coding

Model based coding is a lossy compression method, which is used for example in speech coding and is promising as well for an efficient use on detector data. The important information from the semi-Gaussian waveform is the amplitude and the time (time stamp) when the signal has been recorded. The difficulty is to find out the real amplitude and time stamp of the analogue signals from the few digitized samples. The digitization of the analogue signal through the ADC in the front-end chip has a sampling rate that is not high enough to guarantee a precise measurement of the amplitude and time by just taking the maximum sample value and position. Since the sampling time of the input waveforms is not synchronized with their arrival, it is not known how close the maximum sample is to the real amplitude of the analogue signal. This implies that the time between the maximum sample and the real peak of the waveform is not constant which makes a simple extraction of the amplitude and time stamp impossible. In the actual offline data analysis, the samples of a detected waveform are fitted with a semi-Gaussian fit function to reconstruct the original amplitude and time stamp. This fitting of the waveforms requires an exponential function (e.g. as given in equation 4.9), which is too complex to perform in hardware.

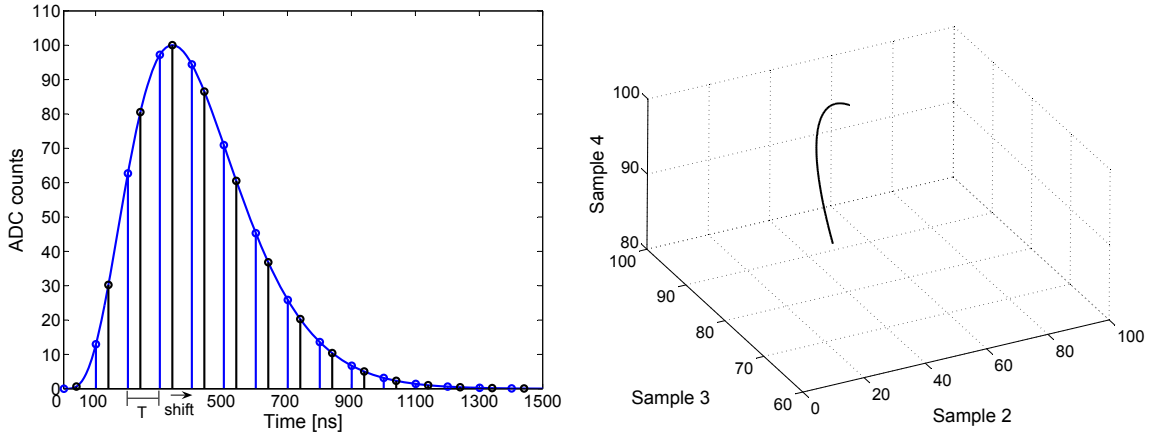
The idea of the model based coding is to find a simple model, which allows a parameter extraction for the important parameters from the received sample values, in order to send out only these parameters with the required precision instead of sending the individual sample values. This reduces the data of an input waveform to two values instead of the corresponding number of samples. The key-point of this compression method is to find a model, which is simple enough to be implemented in real-time hardware with a reasonable resource/area requirement and power consumption but which is good enough to extract the parameters with a high precision.

One method to extract the important parameters from a few samples of an input waveform is presented in [44]. This model bases on two simple linear polynomial equations weighting the sample. One equation is used for the amplitude extraction and the second one for the time stamp determination. The realization of the polynomial equations in hardware is simple requiring a few additions and multiplications like for FIR-filters. To determine the optimal constant elements (coefficients) of the two polynomials presented in the following equations 4.16 and 4.18 and in [44], the samples of the input waveforms are represented in the sample space where they form a representative curve. For the test-data matrix, the input waveforms have 15 samples. In the 15-diminsional sample space, each input waveform is represented by a point. If we take an ideal semi-Gaussian waveform sampled 15 times as shown in figure 4.26(a), the position in the sample space varies according to the sampling times depending on the shift in respect of the peak of the waveform. The waveform in figure 4.26(a) is sampled with a sampling period of 100 ns and 20 sets of samples are created by shifting the sampling time in 5 ns steps within a sampling period. The different points in a 3-dimensional sample space using the three samples at position 2, 3 and 4 of each set describe the representative curve shown in figure 4.26(b) that can be seen as a section of the 15-dimensional sample space.

It can be seen that the illustrated representative curve in 3-dimensons lies in one plane of the 3-D space. A plane curve can be represented by the polynomial in equation 4.16 as stated in [44].

$$\sum_i a_i s_i = v \tag{4.16}$$

For a precise description of the representative curve several sets of samples s_i from the ideal semi-Gaussian waveform with different time shifts are used. All sets of samples s_i^k and the constant



(a) Ideal semi-Gaussian waveform sampled at two different time shifts

(b) Representative curve in 3D

Figure 4.26.: An ideal semi-Gaussian waveform sampled at shifting sampling times of 5 ns in the range of [0,T] with a sampling period T of 100 ns. The resulting sets of samples are illustrated as points in a sample space. A 3-dimensional section of the sample space is shown using the samples 2, 3 and 4.

coefficients a_i have to result the same amplitude of the ideal semi-Gaussian waveform represented by v. From this constrained the calculation of the coefficients a_i can be performed in Matlab using the matrix notation in equation 4.17.

$$\begin{pmatrix} s_1^1 & s_2^1 & s_3^1 & s_4^1 & \dots & s_{15}^1 \\ s_1^2 & s_2^2 & s_3^2 & s_4^2 & \dots & s_{15}^2 \\ s_1^3 & s_2^3 & s_3^3 & s_4^3 & \dots & s_{15}^3 \\ s_1^4 & s_2^4 & s_3^4 & s_4^4 & \dots & s_{15}^4 \\ s_1^5 & s_2^5 & s_3^5 & s_4^5 & \dots & s_{15}^5 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ \vdots \\ a_{15} \end{bmatrix} = \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ \vdots \end{bmatrix} \tag{4.17}$$

If now an input vector containing the samples of an input waveform is multiplied with the resulting coefficients a_i it should result the corresponding real amplitude value. The coefficients are weighting the sample values in order to obtain the real amplitude value.

For the time stamp, a similar polynomial equation is used with a different set of coefficients giving the time shift of the sampling time of the semi-Gaussian waveform. The polynomial equation is as follows:

$$\sum_i b_i s_i^k = t_k = t_0 + k\tau \qquad k = 0, ..., N-1 \tag{4.18}$$

To determine the time stamp first the input waveforms have to be normalized in amplitude to a defined $NormValue$. The coefficients have to be calculated in order to result together with the normalized samples the time shift t_k. Again the matrix notation is used to calculate the coefficients as given in equation 4.19.

$$\begin{pmatrix} Ns_1^1 & Ns_2^1 & Ns_3^1 & Ns_4^1 & \dots & Ns_{15}^1 \\ Ns_1^2 & Ns_2^2 & Ns_3^2 & Ns_4^2 & \dots & Ns_{15}^2 \\ Ns_1^3 & Ns_2^3 & Ns_3^3 & Ns_4^3 & \dots & Ns_{15}^3 \\ Ns_1^4 & Ns_2^4 & Ns_3^4 & Ns_4^4 & \dots & Ns_{15}^4 \\ Ns_1^5 & Ns_2^5 & Ns_3^5 & Ns_4^5 & \dots & Ns_{15}^5 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \end{pmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \\ b_{15} \end{bmatrix} = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \\ \vdots \end{bmatrix} \tag{4.19}$$

The normalized sample values with different time shifts are named Ns_i^k in the matrix and the time shifts are reduces to the multiplying constant k of the time units τ. The real-time shift is composed from the offset time t_0 and the time shift τ. The offset time t_0 represents the time when the charged particles cross the detector until the related signals are seen at the pads and sampled. To this t_0 the number k of time units τ is added which represents the time difference from the maximum taken sample by the ADC and the real peak time of the recorded input waveform. The offset time is already determined by the zero suppression, which means that only the time shift $k\tau$ has to be extracted from the input signals. The multiplication of the samples of a normalized input waveform with the coefficients b_i should result the real-time shift $k\tau$ between the sampling of the waveform and the peak time. The resolution of the time extraction depends on the time unit τ and is here 5 ns.

First tests are carried out using ideal semi-Gaussian waveforms scaled to different amplitudes and sampled at different time shift as shown in figure 4.27. The tests showed a good performance of the model based coding method using the parameter extraction described above.

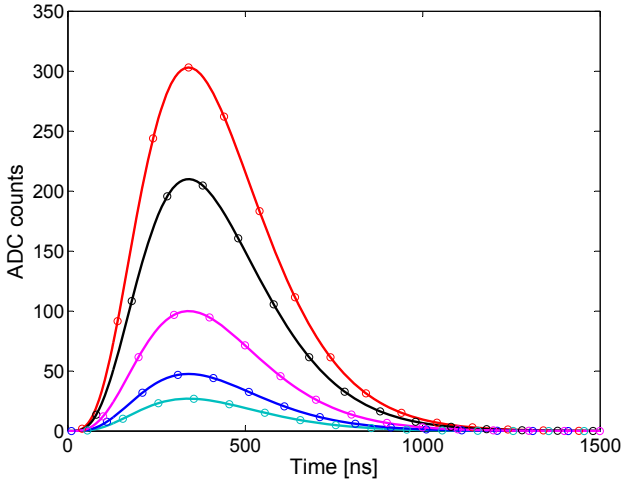
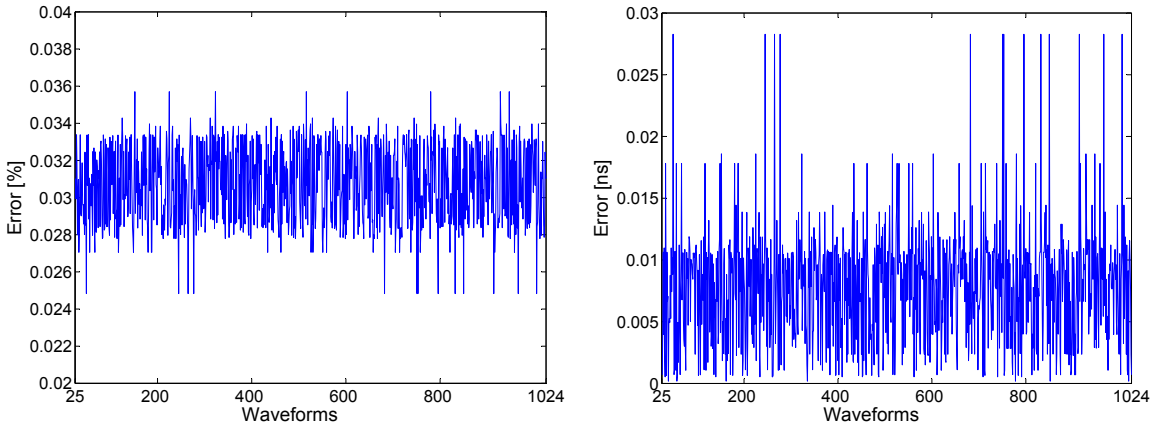


Figure 4.27.: Some ideal semi-Gaussian waveforms with different amplitudes and sampled at different times

First the amplitudes of the semi-Gaussian waveforms are extracted from the 15 samples by using the coefficients a_i. The waveforms are sampled with a sampling period of 100 ns and a random number of steps of 5 ns in the range of [0,100]. The obtained extracted amplitudes are then compared to the real amplitudes of the semi-Gaussian waveforms and the absolute differences are calculated. The differences are compared to the real amplitudes of the corresponding waveforms and shown in % in figure 4.28(a).

Then the time shifts are calculated using the coefficients b_i. First, the samples from the semi-Gaussian waveforms are normalized using the extracted amplitude values and multiplying each sample of the corresponding waveform with the factor $100/ExtAMP_i$. The input waveforms are normalized to the defined amplitude of 100, which is equal to the amplitude of the semi-Gaussian waveforms used to calculate the coefficients b_i. Then the normalized samples of each waveform are multiplied with the coefficients b_i and summed up. The resulting time shifts t_k are compared to the random generated time shifts of the sampling time. The absolute differences are presented in nanoseconds in figure 4.28(b).

It can be seen that the errors in amplitude and time are very small compared to the original amplitude or the sampling periods. This would make the model based method perfectly suited for a parameter extraction in the front-end electronics obtaining a high compression efficiency. For example if one uses 17 bit to represent the amplitude and 7 bit to represent the k of the time shift unit $k\tau$ (gives a resolution of better than 1 ns for 100 ns sampling period) one needs to sent 24 bit



(a) Amplitude error of ideal semi-Gaussian signal (b) Time stamp error of ideal semi-Gaussian signal

Figure 4.28.: Error in amplitude and time of the parameter extraction from 1000 ideal semi-Gaussian signals with different amplitudes and sampling times. The amplitude error is given in percent of the differences normalized to the original amplitudes of the waveforms. The error in time is given in nanoseconds.

per input waveform out of the detector. For the underlying example, the input waveforms have 15 samples with 10 bit per sample which results in the following compression ratio:

$$Compression\ Ratio = \frac{17\,\text{bit} + 7\,\text{bit}}{10\,\text{bit} \times 15} \times 100 = \frac{24\,\text{bit}}{150\,\text{bit}} \times 100 = 16\% \tag{4.20}$$

The compression ratio depends on the required (chosen) resolution of the two parameters and the number of samples per input waveform, but it is independent from the probabilities of the samples. Nevertheless, one aspect has to be taken into account that the real digitized signals in the front-end electronics are not ideal semi-Gaussian waveforms but they contain noise. The samples of the input waveforms are distorted by different sources in the detector structure (e.g. gas fluctuation distortion, field distortion, drift time uncertainties etc.) and in the front-end electronics (e.g. limited bandwidth of filters, rounding errors from limited precisions in the digital processing unit etc.). Therefore, the method has to be evaluated on real data to see how much the noise from the samples affects the accuracy of the reconstructed parameters.

The parameter extraction is executed in Matlab using the test-data matrix, which already has been used for testing the previous compression concepts. The coefficients a_i and b_i are calculated in order to optimize them for this test-data. To determine the coefficients a_i, the waveforms in the test-data matrix are fitted to find out the original amplitude represented by the maximum of each fitted waveform. These maximum values are then building the vector v of the equation 4.16. Next, the pseudoinverse matrix is calculated out of the test-data matrix. This pseudoinverse matrix is multiplied with the vector v to obtain the coefficients a_i. The use of the pseudoinverse matrix produces small coefficient values so that the effect of the noise of the sample values is reduced on the coefficients (see [45]). The values of the coefficient vector a are as follows:

$$\begin{aligned} a \;=\; & [-0.02, -0.01, -0.07, -0.04, 0.22, 0.67, 0.22, 0.06, -0.01, -0.08, -0.02, -0.01, 0.02, \\ & -0.05, -0.04] \end{aligned}$$

The resulting coefficient vector a is then multiplied to each row of the test-data matrix containing the samples of each input waveform and the amplitude of the waveforms is extracted. The extracted amplitude values from all the 10 000 waveforms in the test-data matrix are compared with the original amplitudes from the fitted waveforms and the difference (error) is calculated.

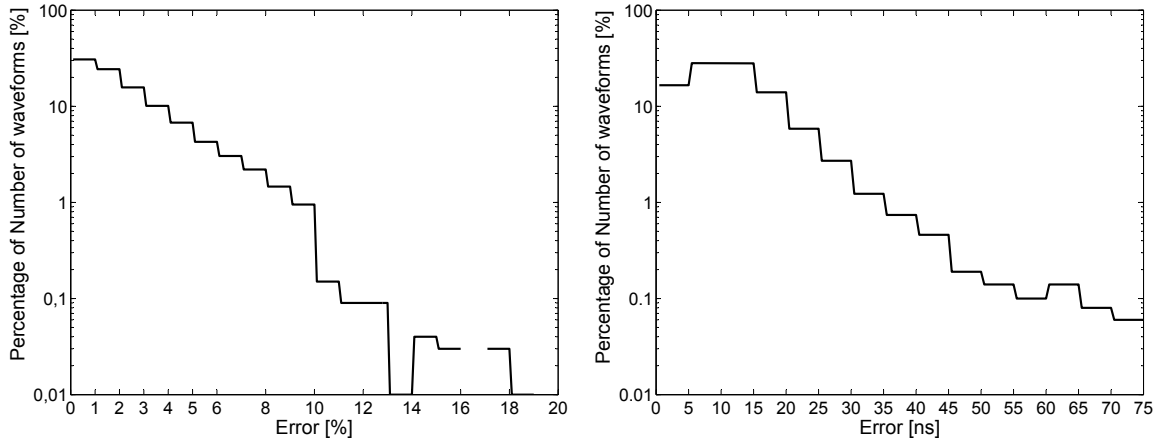
After the extraction of the amplitudes, the time stamps can be extracted. First, the original amplitudes of the fitted waveforms are used to calculate the normalization factor ($f_i = 100/amp_i$) for

each input waveform. Then the samples of the waveforms in the test-data matrix are normalized using the corresponding factors and a second pseudoinverse matrix is calculated from the normalized test-data matrix. To obtain the coefficients b_i the pseudoinverse matrix is multiplied with the vector t_k, which contains the sample positions of the maximum samples from the fitted waveforms. The resulting values for the coefficient vector b are as follows.

$$b = [0.22, -0.11, -0.21, -0.06, -0.12, 0.15, 0.10, 0.22, -0.03, 0.09, -0.02, -0.18, 0.23, -0.29, -0.17]$$

To extract the time stamps out of the waveforms in the test-data matrix, each row of samples of the test-data matrix is first normalized using the extracted amplitudes from the previous step. Then the normalized waveforms are multiplied with the resulting coefficient vector b to extract the time stamps. The extracted time stamps are then compared with the original position of the peak of the fitted waveforms and the difference (error) is calculated.

The resulting error for the extracted amplitudes and time stamps of the 10 000 test-data waveforms are evaluated and shown in figure 4.29.



(a) Amplitude error using real measured data

(b) Time error using real measured data

Figure 4.29.: Error in amplitude and time of the parameter extraction from 10 000 real measured waveforms from the test-data used in the previous compression methods. The amplitude error is given in percent of the differences normalized to the original amplitudes of the waveforms. The error in time is given in nanoseconds.

The error is comparable to the previously presented lossy compression methods. More than 90% of the 10 000 extracted amplitudes have an error below 5%. A bit more than half of the 10 000 waveforms have an error below 2%. The error of the reconstruction of the time stamp is a bit higher because the extraction has to use the already extracted erroneous amplitudes to normalize the input waveforms. More than 90% of the 10 000 extracted time stamps have an error less than 25 ns (sampling time is 100 ns). Around 45% of the input waveforms have an error below 10 ns. The introduced error from the model based method is higher as by other lossy compression methods but the achievable compression ratio is significantly better.

For the hardware implementation two FIR filter could be used with a latch at the output to extract the parameter values in the right moments when all samples of an input waveform are shifted to the correct position in the FIR filters. The normalization logic previously presented is needed to scale the input waveforms using the extracted amplitudes before the time stamps can be extracted.

To guarantee the quality of the reconstructed important information using this method a quality check could be performed in the hardware on the extracted parameters before they are send out.

An ideal semi-Gaussian waveform can be saved in an internal memory with a sampling time of τ that is used in the time stamp extraction process. Then the extracted amplitude from the samples of an input waveform is used to scale the stored semi-Gaussian signal to the corresponding height. The extracted time stamp determines which samples of the ideal semi-Gaussian signal have to be taken considering the original sampling rate of the input waveforms (e.g. 10 MHz). The selected samples of the ideal semi-Gaussian signal are then compared to the original samples of the input waveforms and the residuals are calculated. The absolute values of these residuals are summed up and compared to a defined threshold value to qualify the precision of the extracted parameters. If the sum of the absolute residuals is below the defined threshold, the two extracted parameters are sent out; otherwise, the original samples of the input waveform are transmitted in uncompressed form. In this way, a good lossy compression can be achieved by guaranteeing a quality of the parameter extraction that is acceptable. This check has the disadvantage that a memory is required to store the oversampled semi-Gaussian signal in addition to the FIR filters. By using a τ of 20 times smaller than the sampling period of the input waveforms and assuming that the input waveforms have at most 15 samples the memory has to be able to store 300 samples of 10 bit. If the τ is selected to be smaller a better resolution of the time stamp extraction is achieved which might reduce the error but this will increase the required memory space.

5. Implementation of the algorithm in a FPGA

After a suitable compression algorithm for particle detector front-ends is found and evaluated in chapter 4 an implementation of this algorithm is presented in this chapter. The implementation of the lossless vector quantization method bases on normalization of the input signals, vector quantization, delta calculation and Huffman coding. A project for the ALICE Time Projection Chamber front-end electronics foresees the design of a new mixed signal full custom ASIC in $130\mu m$ technology with the name S-ALTRO, which will combine the analogue pre-amplifier and shaper chip PASA and the digital chip ALTRO. In the course of this project the R&D of a data compression method is foreseen with the aim to implement an additional data compression in the new mixed signal front-end ASIC. For a first prove of concept of the here developed compression algorithm an implementation in a Field Programmable Gate Array (FPGA) is carried out. The presented implementation in this chapter is targeted on a FPGA, which is installed in the Readout Control Unit (RCU) of the ALICE TPC front-end electronics that gives the possibility for a future integration in the real detector environment. The RCU is sitting behind the front-end cards, which house the PASA and ALTRO chips and it controls the data taking from the ALTRO chips as well as preparing the data for sending them off-detector to the counting room. The design is realized in the hardware description language Verilog. The developed Verilog code can be used as well for a later ASIC implementation in the new front-end chip to move the data compression closer to the data source (detector pads). This chapter presents the specifications, the parts and the challenges of the data compression implementation.

5.1. Specification of the RCU Data Compression implementation

The lossless compression algorithm described in chapter 4 consists of three main parts: Normalizer, Vector Quantizer, Delta Calculator and Huffman Coder.

The data compression block containing these three parts is designed to meet the requirements of an implementation in the RCU FPGA. To understand the requirements first the data path through the front-end electronics of the TPC in ALICE from the detector pads until the data compression block inside the RCU has been investigated. A block diagram showing the components along this data path can be seen in figure 5.1.

The different components of the front-end electronics are described in the following:

The signal induced in the detector pads is first amplified and shaped by the PASA. The PASA is an analogue ASIC containing 16 channels connected to 16 detector pads [26]. The first stage of a PASA channel is an integrator, which integrates the current induced in the detector pad to obtain a charge-equivalent output voltage. Then a differentiator performs a pole-zero cancellation to prepare the signal for the last stage made of two low-pass filters second order. The two low-pass filters are seen as shaping amplifiers. The output signal of the PASA after the filters has a semi-Gaussian shape 4^{th} order as presented in chapter 2 and 4. The semi-Gaussian signal is then send to the ALTRO chip. The ALTRO chip contains as well 16 channels [47, 48]. Each channel has a 10-bit ADC sampling at 10 MHz to digitize the analogue signal. The digital signal is processed by the

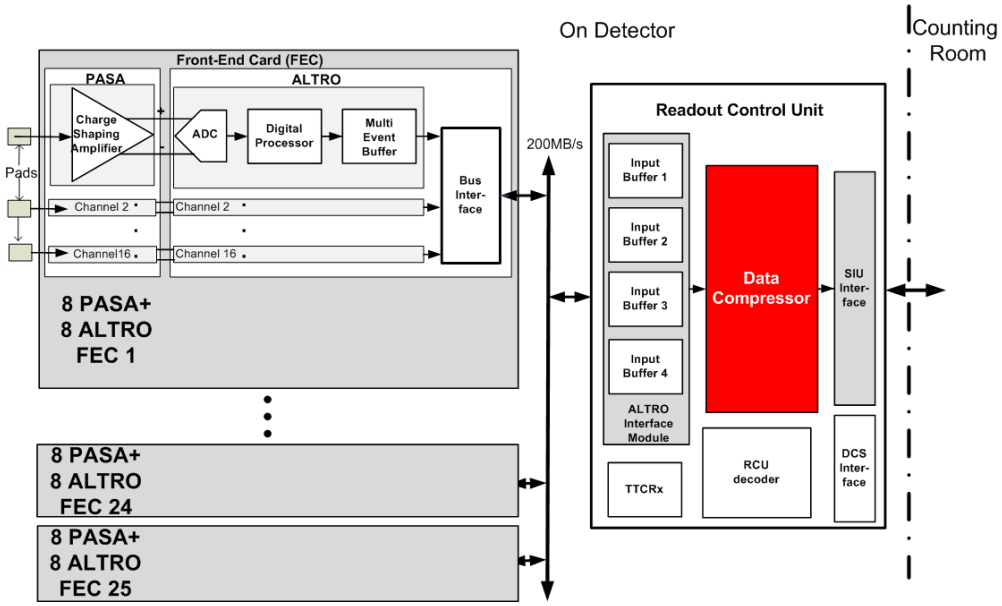


Figure 5.1.: Block diagram of the front-end electronics of the TPC in the ALICE experiment (modified from [46])

digital processor in each ALTRO channel. The digital processor performs first a baseline correction based on pedestal values stored in a memory. The pedestal values are subtracted from the digitized input signal to correct the baseline for systematic errors and prepare the signal for the following tail cancelation filter. The tail cancelation is performed by a 3-stage Infinite Impulse Response (IIR) filter used to cancel the long signal tail caused by the slow moving ions from the avalanche process in the MWPC. This long ion tail is problematic for pile-up effects on following input waveforms, which would render the zero suppression inefficient and cause an error in the signal amplitude. The last filter of the digital processor is a moving average filter to perform the second baseline correction, which reduces non-systematic perturbations. After the filtering of the input signal the zero suppression is performed, which can be considered as a first lossy data compression after the digitization of the input signal. The zero suppression uses a defined threshold to distinguish useful information from non important data (zero data). The zero suppression is explained in figure 5.2.

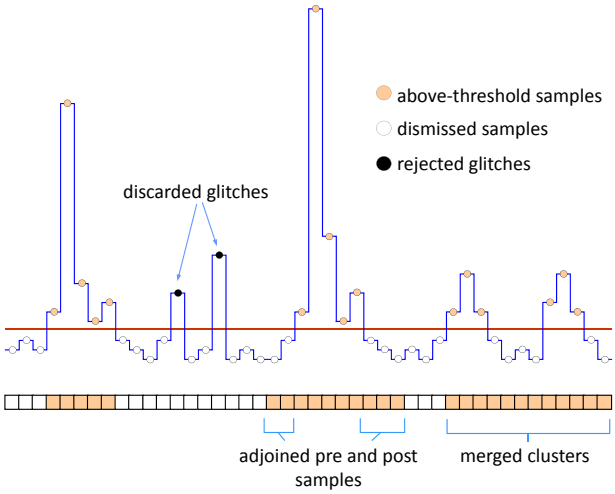
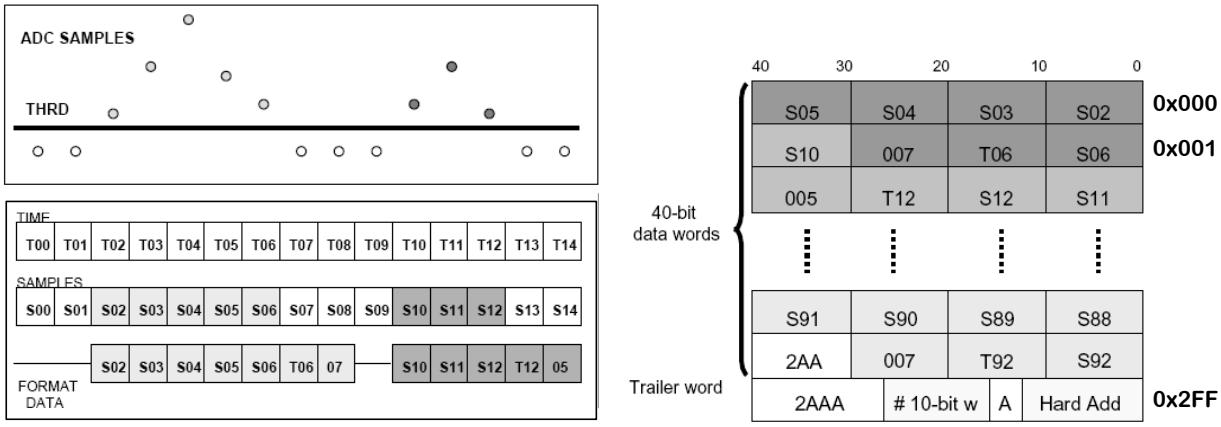


Figure 5.2.: Zero Suppression in the ALTRO chip [3]

When a sample is found above the threshold, it is considered as the start of a waveform. To-

gether with the following samples above the threshold it is stored in the Multi-Event Buffer (MEB) until a sample is received with a value below the threshold. This sample is considered as the end of a waveform. A number of pre- and post-samples can be defined to be stored in the MEB together with the detected waveform. If only a single sample (or less than 3 samples) raise above the threshold it is considered as a glitch and nothing is saved. Each cluster (group of samples ideally considered as one waveform) is combined with a time stamp, which preserves the arrival time information of the waveform. If a pile-up effect occurs, one cluster contains more than one waveform. If a L1-trigger arrived the zero suppressed data are stored in a MEB using the format shown in figure 5.3.



(a) Cluster data formation including samples, time stamps (T06) and cluster width (number of words per cluster e.g. 07) [2]

(b) ALTRO output data format stored in the MEB [47]

Figure 5.3.: Zero suppressed output data of the ALTRO stored in the MEB with the ALTRO channel header (Trailer word) and read out by the RCU

The data in the MEB of the ALTRO channels are then readout by the RCU upon a L2-trigger accept signal. The data transfer to the RCU is performed with 40 MHz and a word length of 40 bit. This means that up to 4 sample values are received every 25 ns at the RCU. Inside the RCU FPGA, there are four buffer memories to save the data from the ALTROs.

The Data Compression block (DC-block) will read the data from the internal buffer memories inside the RCU and compress them. The DC-block has then to bring the compressed data in a defined format and to store them in an output buffer inside the Source Interface Unit (SIU) of the RCU. The command to the DC-block for starting reading an internal buffer is given from the RCU by setting the corresponding $data\_ready$ signal. This signal contains 4 bit, each one is set high after the RCU has saved all data from one ALTRO channel MEB in the corresponding internal buffer. The RCU then can read in the next ALTRO channel MEB and save the data in the next free internal buffer, while the data compressor is reading and processing the data of the current buffer. The data compressor will perform the compression of the single waveforms (clusters) and build the output format. After the formatted compressed data are stored in the SIU output memory they can be read out and transferred via optical links to the counting room (off-detector).

For the new version of a TPC mixed signal front-end chip with the name S-ALTRO, a clock frequency of 40 MHz and a readout frequency of 80 MHz is planned. In order to prepare the data compression for a future implementation in this S-ALTRO, two clock input signals with frequencies of 40 MHz and 80 MHz are used inside the DC-block.

The description of the actual TPC front-end electronics leads to the following conditions that the design of the data compression block has to fulfill:

- The input bandwidth is $40\,\mathrm{bit} \times 40\,\mathrm{MHz} = 1.6\,\mathrm{Gbit/s} \approx 191\,\mathrm{Mbyte/s}$
- Controlling the reading of the four internal buffers of the RCU

- Compressing the data in real-time by processing four sample values per clock cycle to cope with the 191 Mbyte/s
- Formatting the compressed data according to the RCU output format
- Controlling the storage of the formatted data in the SIU output buffer
- Two input clocks one with 40 MHz and one with 80 MHz

A top level block diagram of the implementation of the DC-block in the RCU FPGA environment is shown in figure 5.4. The different parts of the realized data compression block are described in the following.

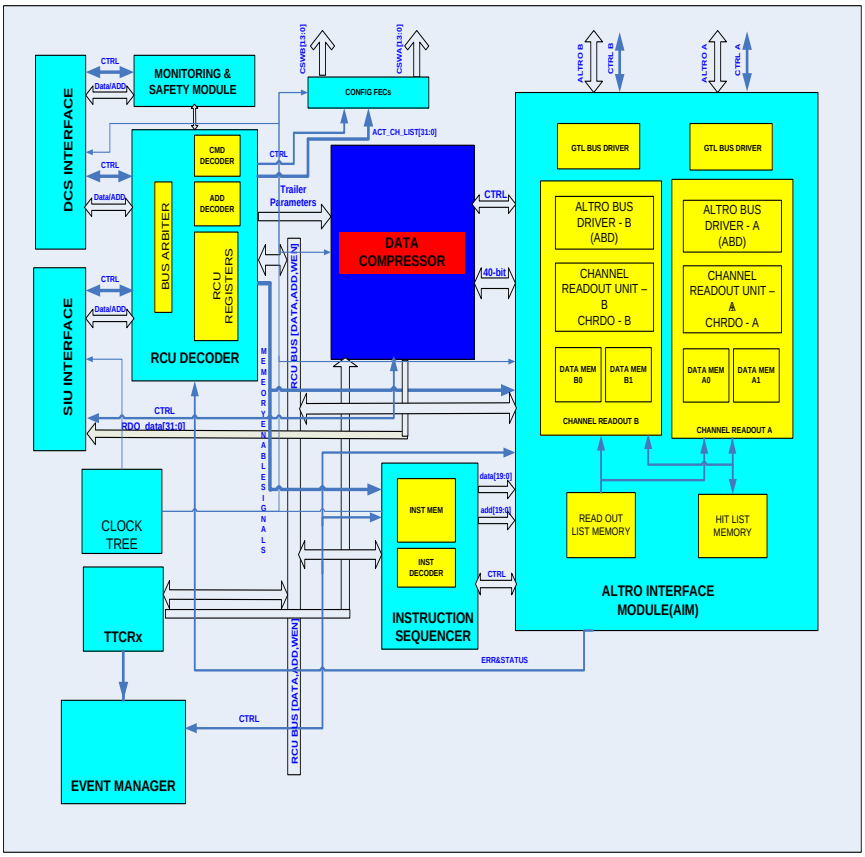


Figure 5.4.: Block diagram of the RCU and data compression block interface [49]

5.2. Overview of the data compression block

According to the specifications and the algorithm described in chapter 4 a first conceptual block diagram is presented in figure 5.5 as an overview for the created RTL Verilog model. The functions of the parts inside the data compression block are explained shortly, before each part is discussed more in detail in the following sections.

The different parts are realized in individual Verilog modules, which are instantiated in a top module called DataCompressor. This DataCompressor module represents the entire data compression block and in the following "DataCompressor" is used to refer to the implementation of the compression algorithm. The DataCompressor uses the two clock signals, which are provided by the RCU. The 40 MHz input clock is called $Clock40$ and is the primary clock signal used widely in the implementation. A second fast input clock signal of 80 MHz is called $Clock80$ and is used mainly in the OutputFormatter module for the readout of the data. A pipelined divider inside the Normalizer module uses as well the fast 80 MHz clock.

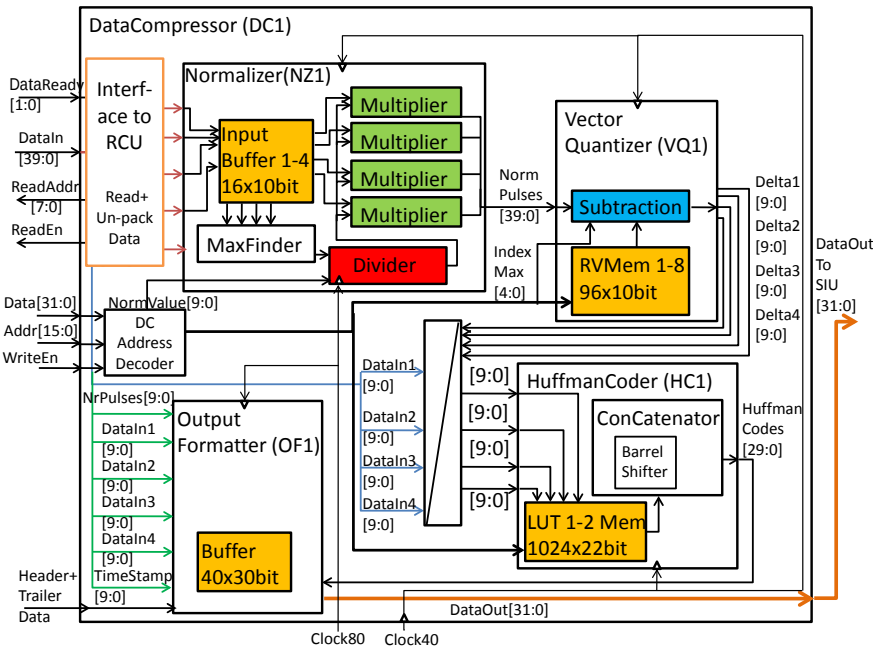


Figure 5.5.: Conceptual block diagram of the RTL Verilog model for the DataCompressor

The DataCompressor top module (DC1) handles the communication with the remaining RCU logic. A state machine in the DC1 module controls the readout of the data from the internal buffers of the RCU. The data from the internal buffers, which are formatted according to the ALTRO data format, are split up in the individual components. The sample values are sent together with the waveform width (number of samples per waveform) to the Normalizer module called Normalizer (NZ1).

The Normalizer stores the sample values of the input waveform in a buffer memory while it starts searching for the maximum sample value of the waveform. If the maximum sample is detected the normalization factor is calculated using a pipelined integer divider. After the Divider module returns the calculated normalization factor, the sample values of the corresponding waveform are read out from the Normalizer internal buffer and multiplied with this factor. In this way, a normalized input waveform is obtained. Four multipliers are working in parallel to normalize four sample values per $Clock40$ cycle. This is important to cope with the input bandwidth of four 10-bit input sample words per 40 MHz. The Normalizer will send each $Clock40$ cycle up to four normalized samples to the VectorQuantizer module. In addition, the position of the maximum sample in each input waveform is sent to the VectorQuantizer to perform a time alignment between the input vector containing the samples of the normalized input waveform and the reference vector.

The DataCompressor top module contains two different VectorQuantizer modules, which can be instantiated; one called VectorQuantizer1QSimple (VQ1), which uses only one reference vector and a second one called VectorQuantizer4Q (VQ4), which uses four reference vectors. According to the available resources in the target FPGA one of the two VectorQuantizer modules can be used.

The VQ1 contains two dual-port memories, which are programmed prior to the measurements with the elements of the reference vectors. The index of the maximum sample of a received normalized waveform is used to address the reference vector memory and to read out the corresponding reference vector elements, which are aligned with the normalized samples from the input vectors. Then the differences of the reference vector elements (reference samples) and the normalized samples are calculated and the resulting Delta values are forwarded to the HuffmanCoder. Two dual-port memories have to be used for storing the reference samples twice to allow reading out four reference samples per $Clock40$ cycle and therefore being able to process four normalized samples in parallel. The VQ4 uses four different reference vectors to obtain a better compression

efficiency, which requires four times more memory as for VQ1 to save them. The index of the maximum sample is as well used to align the normalized input samples to the for reference vectors. Then four sets of Delta values are calculated in parallel using the input vector and each of the four reference vectors. To decide which of the four sets of Delta values give the best compression efficiency the sum of the absolute values of each set is calculated and the set with the minimum sum is sent out to the HuffmanCoder. The index indicating the selected set is sent to the OutputFormatter module to be included in the output data.

The Huffman coding module is called HuffmanCoder (HC1) and it consists of two dual-port memories, which serve as Look-Up Tables (LUT1 and LUT2) and a ConCatenator module. The two dual-port memories are programmed prior to operation with the Huffman codebook, which is calculated in Matlab using representative data. The received Delta values are used as address for the two memories and the corresponding Huffman codewords and codeword lengths are read out and sent to the ConCatenator module. The two dual-port memories containing the same codebook and are required to process four Delta values per $Clock40$ cycle. The HuffmanCoder can also be used to compress directly the original input samples if the corresponding Mode is set and the corresponding codebook is saved in the two memories. The ConCatenator called ConCatenator60 (C1) uses barrel shifters to concatenate the variable length Huffman codewords to an output bit stream. The output bit stream is then sent to the OutputFormatter in blocks of 60 bit.

The data-formatting module is called OutputFormatter (OF1) and it combines the compressed data with header and trailer information from the RCU to a defined output package. The OutputFormatter formats the information in 32-bit words where the two most significant bits (MSB) are flag bits to distinguish the different kind of package words. Each data package for an event starts with a common data header and ends with a trailer.

The implementation of the data compression block is designed to support three modes of operation.

The general mode, which is selected by setting the $DcMode$ register to '11', performs the described data compression algorithm of chapter 4. In this mode all the above described modules, Normalizer, VectorQuantizer, HuffmanCoder and OutputFormatter are in use to obtain the best compression efficiency.

The second mode of operation, which is selected by setting the $DcMode$ register to '01' uses only the HuffmanCoder and the OutputFormatter inside the DataCompressor top module. A direct compression of the input samples using only Huffman coding is performed. The resulting concatenated Huffman codewords are then send to the OutputFormatter and used as the payload for the data package. Important is to program the LUT memories of the HuffmanCoder with the correct Huffman codebook. The compression efficiency will be lower than by using the full algorithm as shown in chapter 4, but the execution of this compression is simpler and faster (lower latency). The modules Normalizer and VectorQuantizer are not used in this mode of operation.

The third mode of operation is selected by setting the $DcMode$ register to '00' and is not performing any compression on the input data. The sample values of the RCU internal buffers are directly send to the OutputFormatter where they are included uncompressed in the data package.

The following sections describe the realization of the individual Verilog modules in more detail.

5.3. Data Compressor top module

The **DataCompressor** module (DC1) is the top module of the design, which is connected to the other blocks of the RCU. Before the DataCompressor can start working, it has to be initialized; the Huffman codebook and reference vectors have to be programmed into the dedicated memories. A 16-bit wide address bus is used to address the corresponding memories and registers. A 32-bit wide bidirectional data bus is used inside the RCU and to program the data compression block.

The DataCompressor internally uses a 22-bit wide bidirectional bus to distribute the data to the various memories and registers. With this bidirectional bus, it is as well possible to read back the programmed data for testing purposes or to check the mode of operation and configurations. The RCU itself uses the Detector Control System interface (DCS) to receive instructions from the counting room and to provide control information. To access the programmable memories and registers the input signal $WriteEn$ has to be set to '1' for writing or '0' for reading. The correct address has to be set at the address bus $rcu\_address$ and then the data can be written or read via the data bus called rcu_prog_data.

Inside the DataCompressor, a module called DCAddrDec (AD1) is performing the address decoding. This module has to decode the addresses received from the RCU and set the corresponding enable signals and addresses to select the corresponding memory or register. The composition of the received address from the RCU is as follows:

Bits 15-13	**Bits 12-10**	**Bits 9:0**
For DC fixed to '011'	Instruction Set	Address to select the words of the memory or register

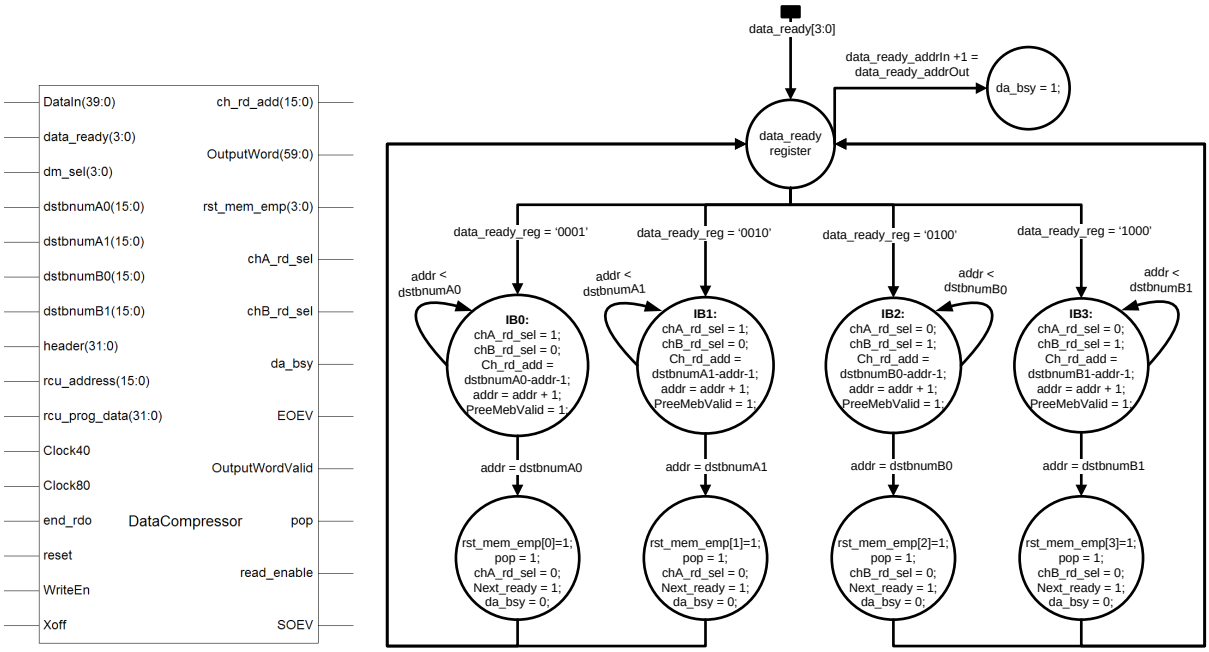
The three MSBs '011' indicate that the address is dedicated to the data compression block. The next three bits are selecting the programmable device. The values '000'-'010' of the bits 10 to 12 are reserved to address the two Huffman LUTs for storing or reading the codebook ('000' enables LUT1, '001' enables LUT2, '010' enables both). If the bits are set to '011' the registers for the $DcMode$, $NormValue$, $NormValueIndex$ and $FristSecond$ can be programmed. The values '100'-'111' are used to program the four reference vectors. The remaining 10 bits of the address word (0 to 9) are used to access the individual words in the memories or to select the corresponding register. The resulting address space dedicated to program the data compression block is listed in table 5.1.

Table 5.1.: Adress space for the parameter registers and programmable memories

Dc Registers:				
Name	**Address**	**Size**	**Access**	**Description**
NormValue	0x6C00	1x10	W/R	Defines the max sample value of the Reference Vectors. Default:100
NormValueIndex	0x6C01	1x6	W/R	Defines index of the max sample in the Ref. Vectors. Default: 31
DcMode	0x6C02	1x2	W/R	Data Compressor working mode. Default: 3
FirstSecond	0x6C03	1x1	W/R	To switch between 2 equal Mems
Dc Memories:				
Name	**Address**	**Size**	**Access**	**Description**
Huffmem	0x6000-0x63FF	22x1024	W/R	LUT for the Huffman Codewords
RVmem1	0x7000-0x703F	10x64	W/R	Memory for the Ref. Vector1. Used for 1Q&4Q.
RVmem 2	0x7400-0x743F	10x64	W/R	Memory for the Ref. Vector2. Used only for 4Q.
RVmem 3	0x7800-0x783F	10x64	W/R	Memory for the Ref. Vector3. Used only for 4Q
RVmem 4	0x7C00-0x7C3F	10x64	W/R	Memory for the Ref. Vector4. Used only for 4Q.

After the data compression block is programmed and initialized, it is ready to take data and

compress them. The DataCompressor module contains a state machine, which controls the reading of the four internal buffers of the RCU. The diagram of the state machine is shown in figure 5.6(b). After the RCU has finished reading one MEB content from a channel of one ALTRO chip and has stored the data in one of the four internal buffers, it sets a ready signal for the corresponding buffer. A 4-bit input signal called $data\_ready$ indicates which internal buffer is filled and ready to be readout from the DataCompressor. The value of the $data\_ready$ signal is stored in a four word deep $data\_readyReg$ buffer and the state machine starts the readout operation. The $data\_ready$ signal is readout from the $data\_readyReg$ buffer to know which internal buffer is ready to be readout next. The buffering of the $data\_ready$ signal is important in case the RCU sends already a new ready signal while the DataCompressor is still reading out the previous internal buffer. The corresponding input word $dstbnumX$ belonging to the internal buffer which is ready is used by the state machine to know the address of the last word of the data packet saved in the internal buffer. From this address, the DataCompressor starts reading the internal buffer. The first word which is read in is the trailer of the ALTRO format (see 5.3(b)). At each $Clock40$ cycle the address output word ch_rd_add is decremented by one to read in the next word from the internal buffer until address=0 is reached. If the address=0 is reached the pop signal is send to the RCU and the reset command for the corresponding internal buffer is set in the $rst\_mem\_emp[x]$ output. This releases the corresponding internal buffer and it can be refilled by the RCU. The DataCompressor busy signal da_bsy is set when the RCU has filled all the remaining internal buffers before the DataCompressor can release one buffer. This prevents that the RCU starts overwriting the actual buffer before it is fully read out by the DataCompressor. The DataCompressor module is also handling the unpacking of the data according to the ALTRO data format show in figure 5.3.



(a) Input (left) and output (right) pins of the DataCompressor module

(b) State machine of the Data Compressor interface to the RCU

Figure 5.6.: DataCompressor module and input interface state machine

The trailer word of the ALTRO format contains a 12-bit address, which tells the ALTRO chip ID and the channel number (indicates the detector pad) from which the data originates. This word is directly sent to the OutputFormatter to be included in the output data packet. A second word of 10 bit in the trailer gives the number of all 10-bit words (samples, time stamps, and cluster widths) contained in the payload. This word is used to calculate the position of the first cluster

width word in the first 40-bit word of the payload (the second word read in). To calculate this position the number of 10-bit words is subtracted from the received $dstbnumX$ values decreased by one and multiplied by four (multiplication realized by shifting 2 position to the left) resulting in $(dstbnumX - 1) << 2 - \#10\_bit\_w$. Another state machine is splitting the payload in cluster width words (Nr. samples), time stamp words and sample words starting from the first cluster width word at the calculated position (in figure 5.3(b) it is 007). The cluster width indicates the number of words belonging to one zero suppressed cluster (ideally one waveform). The cluster width words and time stamp words are sent to the OutputFormatter to be included in the waveform headers of the output data format. If the mode of operation is set to $DcMode$='11', the sample words together with the number of samples ($Nr.Sample = ClusterWidth-2$) for each waveform are sent to the Normalizer. In $DcMode$='01' the samples are directly sent to the HuffmanCoder and in $DcMode$='00' they are given to the OutputFormatter.

The input and output signals of the DataCompressor module are shown in figure 5.6(a) and described in table 5.2.

Table 5.2.: Input and Output signal list of the DataCompressor

Signal	**Polarity**	**Description**
rcu_prog_data [31:0]	Bidirect	Data bus to access the Huffman LUT, Ref. Vectors etc.
$WriteEn$	In	'1' for Write/ '0' for read of Huffman LUT, Ref. Vectors,...
$rcu\_address$ [15:0]	In	Address to the memories of Huffman LUT, Ref. Vectors,...
$reset$	In	Asynchronous reset signal
$Clock40$	In	40 MHz Clock input
$Clock80$	In	80 MHz Clock input
end_rdo	In	End readout. High when last MEB data is saved in RCU
$Xoff$	In	Output FIFO in SIU is full (pause data sending to SIU)
$header$ [31:0]	In	Common Data Header (CDH) from TTRx
$dstbnumA0$[15:0]	In	Start address for reading the 1. internal buffer of the RCU
$dstbnumA1$[15:0]	In	Start address for reading the 2. internal buffer of the RCU
$dstbnumB0$[15:0]	In	Start address for reading the 3. internal buffer of the RCU
$dstbnumB1$[15:0]	In	Start address for reading the 4. internal buffer of the RCU
dm_sel [1:0]	In	Shows which internal memory actually gets filled
$data\_ready$ [3:0]	In	Indicates which internal memory is ready to read out
$DataIn$ [39:0]	In	Input data from the read RCU internal memory
$read\_enable$	Out	Starts the readout if the TTRx for the CDH
ch_rd_add [15:0]	Out	Selects address to be read from the internal memory
chA_rd_sel	Out	Select the access to data memory in branch A
chB_rd_sel	Out	Select the access to data memory in branch B
Pop	Out	Indicates that the memory is read out and can be rewritten
rst_mem_emp[3:0]	Out	Clears the corresponding memory
da_bsy	Out	Indicates that the data compressor is busy
$SOEV$	Out	Indicates a write to the SIU from the data compressor
$EOEV$	Out	Indicates the end of the writing to the SIU
$OutputWord$[59:0]	Out	Output words from the Data Compressor to the SIU
$OutputWordValid$	Out	Validates the output words from the Data compressor

5.4. Normalizer

The **Normalizer** module (NZ1) performs a normalization of the input waveforms to the amplitude defined by the value in the $NormValue$ register. This register is programmed prior to operation via the rcu_prog_data bus. The first operation of the Normalizer is to find the maximum sample of each input waveform. This is performed with the MaxFinder module (MF1). Then the normalization factor has to be calculated, which requires an integer divider realized in the VQDiv module (Div1). The divider needs a few $Clock40$ cycles to perform the calculation ($NormValue/MaxSample$). This makes it necessary to buffer the received input samples in the meantime. Four dual-port memories are available inside the Normalizer for the buffering. The dual-port memories are used in a cyclic manner with one port for writing and one for reading to allow the writing into the memories and the reading from the memories simultaneously. Four memories are used in parallel to be able to write and read four sample values per $Clock40$ cycle. After the normalization factor is calculated for an input waveform the corresponding samples are read out from the buffers and multiplied with the normalization factor by using four multipliers in parallel.

The Normalizer is designed to handle input waveforms with more than 3 samples and at most 32 samples. The signal from the ALICE TPC is considered to have a maximum duration of around 600 ns caused by the time spread of the induced signal in the pads and the shaping time of the PASA chip (around 190 ns). The ADC in the ALTRO of the TPC has a sampling frequency of 10 MHz, which leads to maximum 6 samples per waveform (if no pile-up effect occurred). In the future, it is planned to implement an ADC sampling with 40 MHz inside the new TPC front-end chip S-ALTRO, which results in max. 24 samples for a similar waveform duration. Therefore, for the actual implementation of the Normalizer a maximum waveform length of 32 samples is considered. Bunches having more than 32 samples are considered to contain multiple waveforms due to pile-up effects combined to one cluster by the zero suppression. In this case, the comparison with a reference vectors will result in large Delta values and therefore in an inefficient compression of the data. Consequently, waveforms with more than 32 samples are not compressed in the full compression mode ($DcMode$='11'), but they are included uncompressed (in their original form) in the output data. In case waveforms have only three or less samples (less than 3 should be filtered out by the zero suppression) they are also included in original form in the output data because the effort to compress them is higher as the expected compression efficiency.

5.4.1. Max Finder

As soon as the input samples arrive at the Normalizer the **MaxFinder** module starts searching for the maximum sample. The MaxFinder uses combinational logic to compare the four input samples, which arrive at every $Clock40$ cycle and determines the maximum value of the four. It compares the maximum value with the one found in the previous $Clock40$ cycle that is stored in a Temp register. If the value in the Temp register is larger than the new maximum value the Temp register value is sent out representing the maximum of the current waveform and the Temp register is cleared. Otherwise, the Temp register is updated with the actual maximum sample value for the next $Clock40$ cycle. In addition, the position of the maximum sample has to be calculated. This is performed by detecting the position of the maximum sample in each clock cycle. Until the maximum sample is found, the samples of a waveform received per $Clock40$ cycle are counted. If the maximum is found, the counted samples of the previous clock cycles are added to the position of the maximum sample in the actual clock cycle. A block diagram of the combinational logic of the MaxFinder is shown in figure 5.7.

The index of the maximum sample of the waveform is sent to the VectorQuantizer where it is used to align the input vector with the reference vectors. The value of the maximum sample is

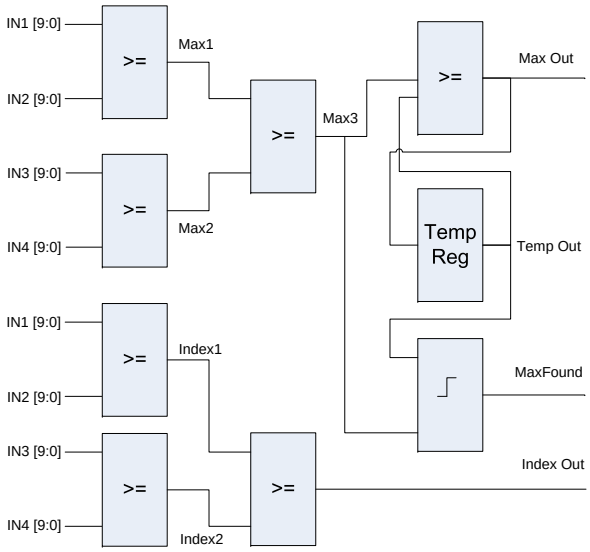


Figure 5.7.: Block diagram of the MaxFinder module, which uses combinational logic to search for the maximum sample value and the position of the maximum sample in the input waveforms

forwarded to the integer divider module for the calculation of the normalization factor. In addition, it is sent to the OutputFormatter to be included in the output data for the re-normalization process in the decoder.

5.4.2. Pipelined Integer-Divider

To calculate the normalization factor for each input waveform the $NormValue$ has to be divided by the maximum sample value of the waveforms.

$$NormFactor = \frac{NormValue}{MaxSample} \tag{5.1}$$

The $NormValue$ is programmed prior to operation in the dedicated register via the programming interface of the RCU and DataCompressor. This value represents the amplitude to which the input waveforms have to be normalized and it should be close or equal to the maximum of the reference vectors. To perform the division a pipelined integer divider is realized. This integer divider is contained in the **VQDiv** module and uses the fast 80 MHz clock signal $Clock80$. The divider is realized in five pipelined stages, each stage processes 3 bit of the numerator using combinational logic. The division concept is shown in figure 5.8 using an example. The divider consists of essentially three comparators, three subtractors and three shift-registers. At each step, one bit of the numerator is shifted into the reminder from the right and then the reminder is compared with the divisor. If the reminder is smaller than the divisor, a '0' is added to the output word and the next numerator bit is shifted into the reminder. Otherwise, a '1' is added to the output word and the difference between the reminder and the divisor is calculated. This difference represents the new reminder and the next bit of the numerator is shifted into the new reminder from the right (CLK Cycle 4 and 5 in figure 5.8). Three comparator and subtractor have to be used sequentially to process 3 bit per $Clock80$ cycle of the numerator. By using more comparator and subtractor sequentially the logic will require more time than one period of the $Clock80$ cycle which causes a timing violation. In case the 40 MHz clock ($Clock40$) is used, more comparator and subtractor can process more bits of the numerator in the longer clock period, which reduces the number of pipeline stages. On the other hand, more area is required for the increased logic without obtaining an overall faster calculation time. Therefore, the 80 MHz clock signal is chosen for the divider to

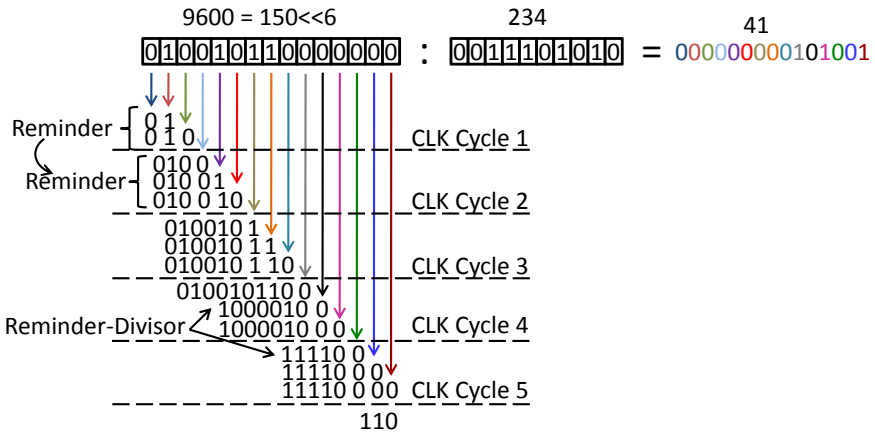


Figure 5.8.: Pipelined integer divider with five pipeline stages

save logic. Nonetheless, the Register Transfer Level (RTL) Verilog code for the divider module is written using parameter statements so that a fast and easy conversion of the number of pipeline stages can be performed. The RTL Verilog code of the Divider module is included in the appendix C.

To obtain a high precision of the normalization factor (given by the division result) the programmed $NormValue$ (numerator) is shifted 6 positions to the left after programming to multiply it with 64 (2^6). In the implementation a 15-bit register is used to store the shifted $NormValue$. The shifted $NormValue$ is then used in VQDiv and divided by the 10-bit maximum sample value. The resulting normalization factor is truncated to 10-bit and multiplied to all the samples of the waveform. The results of the multipliers are stored in 20-bit registers and are then shifted back 6 positions to the right to perform a division by 64. This is needed to compensate for the previous left shift in order to get the correct normalized samples of each waveform. This can be expressed mathematically by:

$$NormS_{Out} = \frac{NormValue}{MaxValue} \times Sample_{In} = \frac{NormValue \times 2^6}{MaxValue} \times Sample_{In} \times \frac{1}{2^6} \quad (5.2)$$

By shifting the $NormValue$ the truncation of the results from the divider and the multipliers is affecting less the precision of the resulting normalized samples. Most of the FPGA families targeted for the implementation have embedded multipliers, which are used in parallel to perform the four multiplications in one $Clock40$ cycle. The normalized sample values of each waveform are sent in blocks of four samples per $Clock40$ cycle to the VectorQuantizer together with the index of the maximum sample.

5.5. Vector Quantizer

The data compression algorithm presented in chapter 4 foresees that the input vectors containing the normalized waveforms are compared with reference vectors, which are stored prior to operation in on chip memories. The difference between an input vector and the best matching reference vector is calculated and results in Delta values that are send to the decoder together with the index of the chosen reference vector. For the implementation of the algorithm in the FPGA, two different versions of the VectorQuantizer are produced; one version using only one reference vector and the second version using four reference vectors.

The VectorQuantizer module with the name **VectorQunatizer1QSimple** contains two dual-port memories to store only one reference vector. The input vectors are compared to this reference vector and the Delta values are calculated and sent to the HuffmanCoder.

The second module named **VectorQuantizer4Q** contains eight memories to store up to four different reference vectors. Each input vector is compared to the four reference vectors and four sets of Delta values are calculated. The set with the lowest sum of the absolute Delta values is then send out.

Each reference vector is programmed into two dual-port memories running at 40 MHz, which allows to read four reference samples a time and to compare them to four normalized samples arriving in one $Clock40$ cycle.

An important step in both VectorQuantizer modules is the alignment of the input vectors containing the normalized waveforms with the reference vectors in a way that the maximum elements of the two are at equal positions and are compared to get the Delta. The remaining vector elements of the input vector are then compared accordingly to the corresponding elements of the reference vector. The positions of the maximum sample of the incoming normalized waveform and the reference vectors are required to both VectorQuantizer modules for the alignment. The index of the maximum sample in the normalized waveform $MaxIndex$ is hand over from the Normalizer. The index of the maximum element in the reference vector is programmed prior to operation in the register called $NormValueIndex$. The received $MaxIndex$ is subtracted from the $NormValueIndex$ and the difference represents the position of the first relevant element in the reference vector, which has to be compared to the first normalized sample in the input vector. Starting from this position, the required following number of reference samples are read out from the memories of the reference vectors and compared to the normalized samples. The alignment of one reference vector and one input vector is shown in figure 5.9.

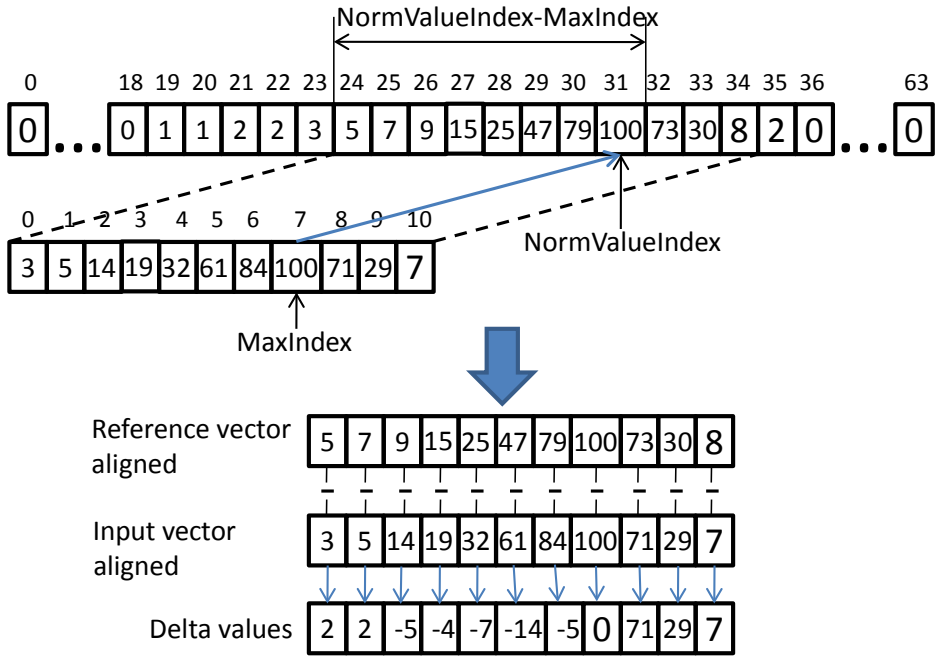


Figure 5.9.: Alignment of the input vector, containing the normalized samples of a waveform and one reference vector. On the bottom, the Delta calculation from the aligned samples is shown.

The VectorQuantizer1QSimple module uses four subtractors working in parallel to calculate four Delta value per $Clock40$ cycle. The resulting Delta values are sent directly to the Huffman-Coder.

In the VectorQuantizer4Q module, 16 subtractors are used in parallel because the normalized input waveform has to be compared to four different reference vectors. After the alignment, four samples of the received normalized waveform are compared with four elements of each reference vector resulting in 4 sets of Delta values. These Delta values are buffered in four buffers dedicated

to the four reference vectors until all Delta values per normalized waveform are calculated. Then the sum of the absolute Delta values in each buffer is calculated (Manhattan distance). The four summation results are compared to identify the buffer, containing the smallest absolute Deltas. The content of the buffer with the smallest Delta values is then sent to the HuffmanCoder. Each buffer is associated with a 2-bit index indicating the used reference vector. This index is sent to the OutputFormatter to be included in the output data. The decoder uses this index to identify the right reference vector for the reconstruction of the normalized input waveform.

Another important information for the decoder is the maximum sample used to re-normalize the input waveform. In addition, also the position of the maximum sample used for the alignment in the VectorQuantizer is important for the decoder in order to execute the same alignment between the selected reference vector and the received Delta values.

One option is to include an additional word in the output data containing the index of the maximum samples in the input waveforms. This will increase the amount of data that have to be transferred to the decoder and reduces the compression performance. Nevertheless, this option is foreseen in the implementation and can be selected by setting the $DcMode$ to '10'.

The preferred variant for the implementation is to include the information in the Delta values itself (which is implemented in mode $DcMode$='11'). This method bases on a few assumptions made on the data produced by the TPC. After the normalization, the amplitude of the normalized input waveform and the maximum of the reference vector should be equal and therefore the Delta value corresponding to the maximum sample should be 0. It can happen that the Delta value for the maximum sample is not 0 because a truncation error is introduced by the limited precision of the divider and the multiplier logic. In this case, the Delta value is anyway forced to 0 in the VectorQuantizer module, regarding the fact that the original maximum value is included in the output data and anyway sent to the decoder. In most cases, the samples of a waveform following the maximum sample (representing the rising edge of the input waveform according to 5.3) result in Delta values different than 0 caused by a not perfect match of the normalized waveform and the reference vector. This property can be used to define that the last 0 Delta value received for each waveform represents the position of the maximum sample, which the decoder uses for the alignment. In case there are single Delta values being 0 after the maximum sample position they are changed to 1 introducing a small error in the reconstruction but this is seldom and negligible after a fitting process of the waveforms is performed in the offline analysis, as can be seen later in chapter 6. The HuffmanCoder codebook used to compress the Delta values assigns to the Deltas with value 0 and 1 a codeword with the same length of 3 bit and therefore the output data volume after Huffman coding remain unaffected by the change of the 0 Delta values.

5.6. Huffman Coder

The **HuffmanCoder** module performs a codification of the input data using variable length codewords. The codewords are created by building a Huffman tree using representative data from previous measurements and then these Huffman codewords are stored prior to operation in on-chip memories. The HuffmanCoder module receives input data either from the VectorQuantizer operating in mode $DcMode$='11' (Delta values) or from the RCU internal buffer through the interface in the DataCompressor in mode $DcMode$='01' (original input data). Depending on the input data, the correct Huffman codewords (codebook) have to be loaded in two dual-port memories inside the HuffmanCoder module. This codewords are defined by performing a Huffman tree on representative data outside the detector (offline). Two dual-port memories are used to read out four Huffman codewords per $Clock40$ cycle, which allows to process four input words in parallel. The data arriving at the HuffmanCoder are used to address the memories that can be seen as Look-Up Tables (LUT). In $DcMode$='01' the original input samples can directly be used as addresses. In

$DcMode$='11' the Delta values are signed value and therefore they are shifted into the positive domain to be used as addresses for the memories. A fixed value of 512 is added to the Delta values, which requires that the codeword for a Delta value of 0 is stored at address $0x200_h$ (512_d) in the memories. Huffman codewords for negative Delta values are saved at preceding addresses, whereas Huffman codewords of positive Delta values are placed at higher addresses. Each entry in the memories consists of a 22-bit word, where the five MSBs are representing the codeword length and the remaining bits contain the codeword bits filled up with zeros. This scheme which is used for the organization of the Huffman codewords in the memories is illustrated in table 5.3. The addressed words in the memories are readout and sent to the ConCatenator60 module.

Table 5.3.: Look-up table for the Huffman codewords and their length in the two dual-port HuffmanCoder memories

	22 ... 17	0 ...
(Delta -512) 0x000	Length -512	Codeword -512
⋮	⋮	⋮
(Delta -2) 0x1FE	Length -2	Codeword -2
(Delta -1) 0x1FF	Length -1	Codeword -1
(Delta 0) 0x200	Length 0	Codeword 0
(Delta 1) 0x201	Length 1	Codeword 1
(Delta 2) 0x202	Length 2	Codeword 2
⋮	⋮	⋮
(Delta 511) 0x3FF	Length 511	Codeword 511

5.6.1. Concatenator

The **ConCatenator60** module concatenates the received Huffman codewords (up to 4 per $Clock40$ cycle), which have variable length (variable number of bits) to an output bit stream. This bit stream is split in 30-bit words and included as the payload in the output data package of the DataCompressor. The ConCatenator60 uses the $Clock40$ signal for the internal logic and sends the output bit stream in blocks of 60-bit words to the OutputFormatter. The OutputFormatter then split up the 60-bit words in two 30-bit words using the 80 MHz clock and add two identifier bits before they are included in the output data package. The ConCatenator60 is designed to concatenate all Huffman codewords resulting from data acquired during one TPC event and readout by the RCU from all the MEBs in the connected ATLRO chips. This allows achieving a good information density in the output package by minimizing meaningless zeros used to fill-up incomplete 30-bit words. To concatenate variable length Huffman codewords several barrel shifters are used in a pipelined structure.

At each $Clock40$ cycle up to four Huffman codewords arrive at the ConCatenator60 module. Two barrel shifter are working in parallel each one concatenating two of the four input Huffman words using their codeword length value. The resulting concatenated codewords are stored in two 34-bit registers called $ConcRegA$ and $ConcRegB$. In the following $Clock40$ cycle these two register contents are concatenated from a third barrel shifter and stored in the 68-bit register called $ConcRegAB$. The relevant bits in the $ConcRegAB$ register are then concatenated to the Huffman codewords from previous input data stored in a $Temp$ register by using a fourth barrel shifter. If the resulting concatenated output word from the fourth barrel shifter has a length of 60 or more relevant bits, the ConCatenator60 sends out the 60 most significant bits. Another barrel shifter that works in left shift mode is used then to store the remaining bits into the $Temp$ register. In case the

resulting concatenation output contains less than 60 relevant bits all bits are shifted to the $Temp$ register and no valid output word is sent in this clock cycle.

The combinational logic function of a barrel shifter allows shifting an input word by a variable number of positions in one clock cycle. The barrel shifters use the length of one input word (number relevant bits) to right shift the second input word accordingly. A following multiplexer network guides then the relevant bits of the two input words to the output register. If for example the first input word of a barrel shifter contains 3 relevant bits then the second input word is shifted 3 positions to the right. The following multiplexer network will connect the first 3 bit of the first word to the output register. The remaining bits of the output register are connected to the barrel shifter output starting from position 4 (first bit of second input word). The barrel shifter pipeline structure including this example is shown in figure 5.10.

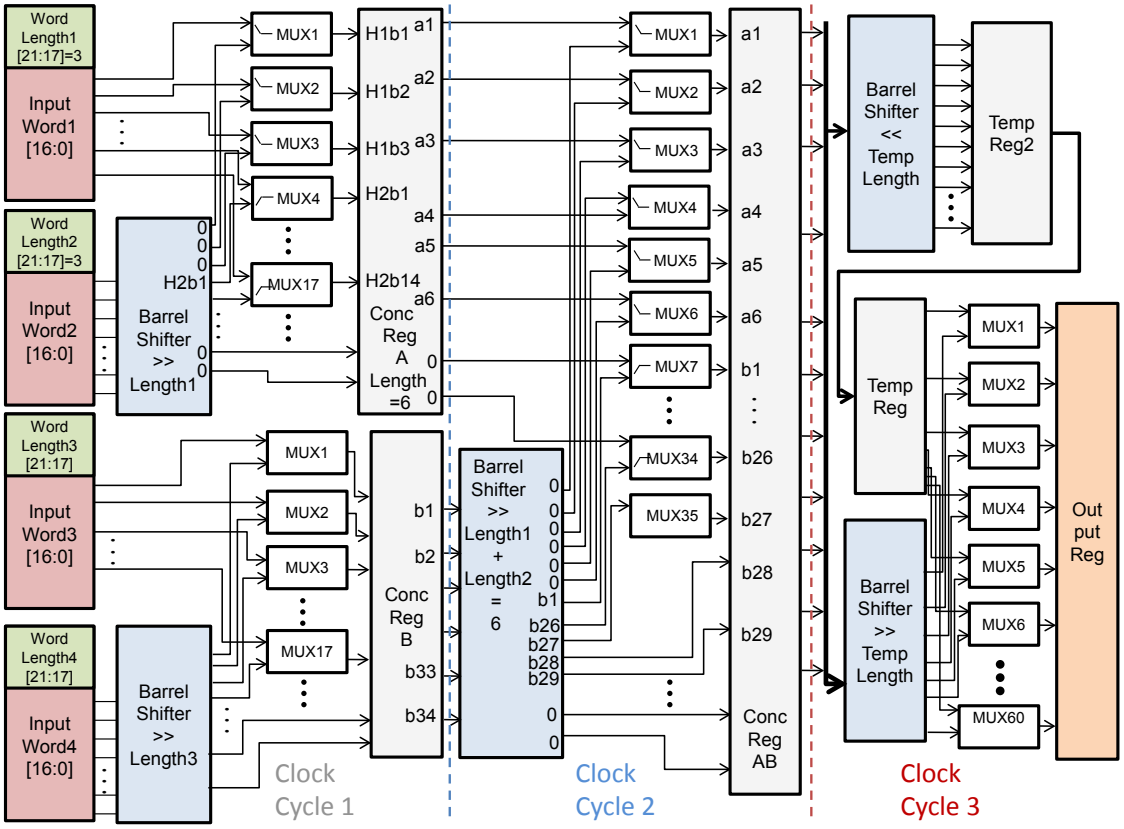


Figure 5.10.: Block diagram of the ConCatenator60 structure with the three pipeline stages of barrel shifters

In case a stop concatenating signal ($StopConCat$) arrives at the ConCatenator60 the last received Huffman coderwords are concatenated and at the end the content of the $Temp$ register is sent out which contains the last Huffman codeword bits. If the number of the last relevant bits in the $Temp$ register is larger than 30 a signal called $LastBitsMore30$ is set high. This tells the OutputFormatter to include all 60 bit of the last ConCatenater60 output word in the output data package. If this signal is low, the OutputFormatter includes only the 30 MSBs from the last ConCatenater60 output word in the package. There are two cases where the $StopConCat$ signal is set to interrupt the concatenation:

First, all input data from the MEBs in the connected ALTRO chips belonging to one event are processed.

Second, a too long or too short input waveform is detected by the Normalizer in the full compression mode ($DcMode$='11'). In this case, the ConCatenator60 finishes concatenating the Huffman coded Delta values belonging to the last good waveform to allow then the OutputFormatter to include the original, uncompressed samples of the too long or too short waveforms into the

DataCompressor output data package. The ConCatenator60 restarts concatenating when the Huffman codewords of Deltas from new accepted waveforms arrive.

5.7. Output Formatter

The **OutputFormatter** module is included in the DataCompressor to prepare the resulting output data for sending them off detector by using a format similar to actual one of the RCU in the ALICE TPC. The module controls as well the writing of the formatted output data into the SIU memory of the RCU. As soon as the first $data\_ready$ signal starts the data compression, the OutputFormatter starts reading the Common Data Header (CDH) information from the Trigger Receiver module (TTRx) contained in the RCU. The CDH consists of 8 words to 32-bit containing the version number of the format, the trigger information event ID and error message codes. The CDH defined in [50] for the RCU is read in and add unchanged to the DataCompressor Output data Package (DCOP).

After the CDH, the payload is included in the DCOP consisting of the ALTRO channel headers, waveform headers and the compressed data. The two MSBs of the payload words are used to distinguish between the different components of the payload. The trailer words are added at the end of the DCOP and distinguished from the payload by the two MSBs. The defined DataCompressor Output Format (DCOF) is shown in table 5.4(a).

Table 5.4.: Data Compressor Output Format (DCOF) including Common Data Header and Trailer

(a) Data Compressor Output format

Address	32 0	
0x0000 ••• 0x0007	Comon Data Header (CDH)	
0x0008	01	ALTRO channel header 0
0x0009	11	NrSamples01 [29:20], TimeStamp01 [19:10], NormFactor01 [9:0]
0x0010	00	Concatenated Huffman Codewords bit stream: word-0 [29:0]
0x0011	00	Concatenated Huffman Codewords bit stream: word-1 [29:0]
0x0012	00	Concatenated Huffman Codewords bit stream: word-2 [29:0]
0x0013	11	NrSamples02 [29:20], TimeStamp02 [19:10], NormFactor02 [9:0]
0x0014	11	NrSamples03 [29:20], TimeStamp03 [19:10], NormFactor03 [9:0]
0x0015	00	Concatenated Huffman Codewords bit stream: word-4 [29:0]
0x0016	00	Concatenated Huffman Codewords bit stream: word-5 [29:0]
0x0017	00	Concatenated Huffman Codewords bit stream: word-6 [29:0]
0x0018	01	ALTRO channel header 1
0x0019	11	NrSamples11 [29:20], TimeStamp11 [19:10], NormFactor11 [9:0]
0x0020	11	NrSamples12 [29:20], TimeStamp12 [19:10], NormFactor12 [9:0]
0x0021	00	Concatenated Huffman Codewords bit stream: word-6 [29:0]
•••	01	ALTRO channel header M
+n+N	11	NrSamplesMn [29:20], TimeStampMn [19:10], NormFactorMn [9:0]
+n+N+1	00	Concatenated Huffman Codewords bit stream: word-N-1 [29:0]
+n+N+2	00	Concatenated Huffman Codewords bit stream: word-N [29:0]
+n+N+3	10	RCU Trailer: First word
•••	10	RCU Trailer: optional words
Last addr	10	RCU Trailer: Last word

(b) Modified Trailer from the RCU [50] including DataCompressor configuration word

31 30	29 26	25 0		
Word ID	**Parameter**		**Value**	
10	Pay Load Length	0000	Number of 32 bit words	
10	Error Register 1	0001	Error Registers for Branch A and B	
10	Error Register 2	0010	Read out Errors	
10	Error Register 3	0011	Number of Altro Trailer Errors	
10	Act FEC A	0100	Active Front End Cards for Branch A	
10	Act FEC B	0101	Active Front End Cards for Branch B	
10	RDO CONFIG 1	0110	Readout configuration Register1 of Altro	
10	RDO CONFIG 2	0111	Readout configuration Register2 of Altro	
10	DC CONFIG 1	1000	DcMode	NormValue / NormValue Index
10	DC CONFIG 2	1001	Reference Vector Set ID	
10	DC CONFIG 3	1010	Huffman Codebook ID	
10	RCU ID	1011	RCU FW version	RCU address

The ALTRO channel header contains the information about the channel address and the block length number of 10-bit words contained in the ALTRO data package, which is read in by the RCU from the MEB of each channel and saved in the internal buffer. An error bit is added to the ATRO channel header to indicate a mismatch between the read words of a channel data package and the stated number of 10-bit words in this package. Each ALTRO channel header can be identified by the two MSBs set to '01'.

The waveform header consists of the three 10-bit words NrSamples, TimeStamp, MaxValue and the two MSBs set to '11'. MaxValue represents the value of the maximum sample of each input waveform and is hand over from the Normalizer to the OutputFormatter. This value is important for the decoder to calculate the $NormFactor$ and to re-normalize the decompressed waveform. The TimeStamp of each input waveform is obtained from the ALTRO data package in the RCU

internal buffers and included unchanged in the DCOP. The NrSamples word tells the number of samples belonging to the input waveform. These NrSamples words are important for the decoder to split correctly the Huffman bitstream and to combine the reconstructed normalized samples to the corresponding waveforms.

The concatenated Huffman codewords are included in the payload in 30-bit words combined with two MSBs set to '00'. The 60-bit words coming from the ConCatenator60 are synchronous with the $Clock40$ and have to be divided by the OutputFormatter into two 30-bit words using the 80 MHz clock. For a future implementation in the new TPC front-end chip S-ALTRO the 80 MHz clock is intended to be used as well for the readout in order to combine the 30-bit words with the two MSBs and sending them out. In the actual RCU version, the SIU can be accessed only with 40 MHz, therefore the 30-bit words are buffered in a small memory inside the OutputFormatter and send with 40 MHz to the SIU. This is possible because the compression using variable length Huffman codewords produces not every $Clock40$ cycle a full output word of 60 bit.

The ALTRO channel header and waveform header information are buffered as well. They are sent out at any time there is no valid compression word from the HuffmanCoder to be sent. The decoder separates the header words from the compressed bit stream by using the two MSBs. This scrambling scheme of the payload reduces the overall time required to send the DCOP by requires smaller buffers. The scrambling scheme requires assuring the correct order of channel header and waveform headers regarding their input sequence to assign correctly in the decoder the reconstructed waveforms to the individual channels in the corresponding ALTRO chips.

At the end, the trailer words are added to the output data package. This trailer contains the number of 32-bit words included in the package, which is added by the OutputFormatter. The trailer information received from the RCU contains error messages, and configuration information, which are included as well in the DCOP. This information can be extended by the configuration information of the DataCompressor containing the $DcMode$, $NormValue$ and $NormValueIndex$ information. In addition, identifiers can be included to track the programmed reference vectors set and Huffman codebook. At the end, the trailer terminates with a word containing the RCU Firmware version and RCU address. All trailer words are identified by the two MSBs set to '10'. The trailer for the DCOF is shown in table 5.4(b).

5.8. Test module for testing the DataCompressor

To test the written RTL Verilog model inside a FPGA a development board from Xilinx is used named ML401 containing a Virtex-4 FPGA [51]. The FPGA contains in total 10 752 slices of 24 192 logic cells. Block RAM memories of 1296 Kb are available inside the Virtex-4 for the DataCompressor module and to store the programming data of Huffman LUTs and reference vectors. The test pattern data will also be stored in a block RAM memory inside the FPGA from where they can be readout by the DataCompressor. In addition the FPGA contains 48 DSP blocks with 18 bit $\times$ 18 bit multipliers from which four multiplier are used in the Normalizer. To test the DataCompressor a top level module called **DataCompression2ClockExt_Top** is created, which handles the programming of the DataCompressor at the initialization phase and the communication with the DataCompressor module in replacement of the real RCU. Memories inside the Virtex-4 are pre-initialized with the codebook, reference vectors and the test pattern data during the programming of the FPGA. The $Clock80$ signal for the DataCompressor is provided by a crystal oscillator mounted in the user-clock socket of the development board. The 40 MHz signal for the $Clock40$ is created by a clock divider logic realized inside the DataCompression2ClockExt_Top module. The module uses three input signals connected to buttons on the development board to control the testing procedure. One button is used to start the initialization of the DataCompressor. This button gives the command for DataCompression2ClockExt_Top module to read the data

from the pre-initialized RAM memories and programs the DataCompressor via the rcu_prog_data bus. After the initialization phase, a second button has to be pressed to start the data compression. The DataCompression2ClockExt_Top module sets one of the $data\_ready$ signal high and provides a RAM memory that is pre-initialized with the test pattern data from where the DataCompressor reads in the data. The test pattern data are stored in the ALTRO format in the RAM memory, which simulates a RCU internal buffer. The DataCompressor performs the compression on the test patter data according to the programmed mode of operation. The OutputFormatter stores the output words in a memory provided by the DataCompression2ClockExt_Top module, which simulates the SIU output buffer of the RCU. The DataCompression2ClockExt_Top module contains a module of a UART interface, which is used to read out the data in the output memory (SIU) via the serial port of the development board. The DataCompression2ClockExt_Top module starts sending the data via the UART to the PC after the pop signal is set from the DataCompressor indicating the end of the compression. A third button is used to reset the DataCompression2ClockExt_Top module and the DataCompressor.

The Huffman codebook and the reference vectors for the initialization of the DataCompressor are calculated in Matlab using measured data from the ALICE TPC measuring cosmic rays. These data are used as well to create the input test pattern data for the DataCompressor. The formatting of the input data according to the ALTRO format is performed as well in Matlab. In addition, a model of the DataCompressor is created in Matlab to compare the results coming from the simulations and implementation of the DataCompressor RTL code for correctness.

The test results of the DataCompressor module in the different modes of operation are summarized in the next section.

6. Implementation Test Results

In this chapter the test results are summarized which are obtained by the Verilog simulations and hardware tests of the data compression unit that is implemented in the Virtex-4 FPGA. First, the functionality of the implementation is tested and then the compression performance is evaluated using the test-data obtained from the measured data from the TPC in ALICE. These test-data have been already used in the Matlab simulations. For testing the functionality, a dedicated Matlab model is created to emulate exactly the steps of the hardware implementation. The results from the dedicated Matlab model are compared with the results from the Verilog simulation of the implementation. At the end the functionality is tested as well by decompressing the bitstream sent out from the data compression hardware implementation in the Virtex-4 and comparing it with the original test-data.

The obtained compression performance of the implementation is compared with the theoretical results from the data compression Matlab simulation of the algorithm presented in chapter 4. The introduced distortion by the Normalizer is discussed.

In addition to the compression performance, also the complexity of the implementation is discussed in this chapter by analyzing the resource requirements inside the FPGA. The latency introduced by the compression block is investigated in order to prove the usability in real-time applications.

At the end of the chapter, a layout of the synthesized Verilog code for a deep-submicron ASIC implementation is presented to discuss the requirements for an implementation of the DataCompressor in a full custom ASIC as the S-ALTRO chip. More and more detector front-end electronics is realized using full custom ASICs to optimize the electronics for the specific applications and to reduce area/power requirements.

The organization of this chapter is divided in four parts:

First, the test setup and procedure of the DataCompressor is presented together with the test pattern data, which are used in the tests.

Then the simulation results of the RTL Verilog model are presented, which are carried out using NC-Sim from Cadence and the Integrated Software Environment tool (ISE) from Xilinx.

Afterwards, the implementation of the DataCompressor in the Virtex-4 FPGA of the development board is tested.

At the end, the full custom layout is presented and discussed.

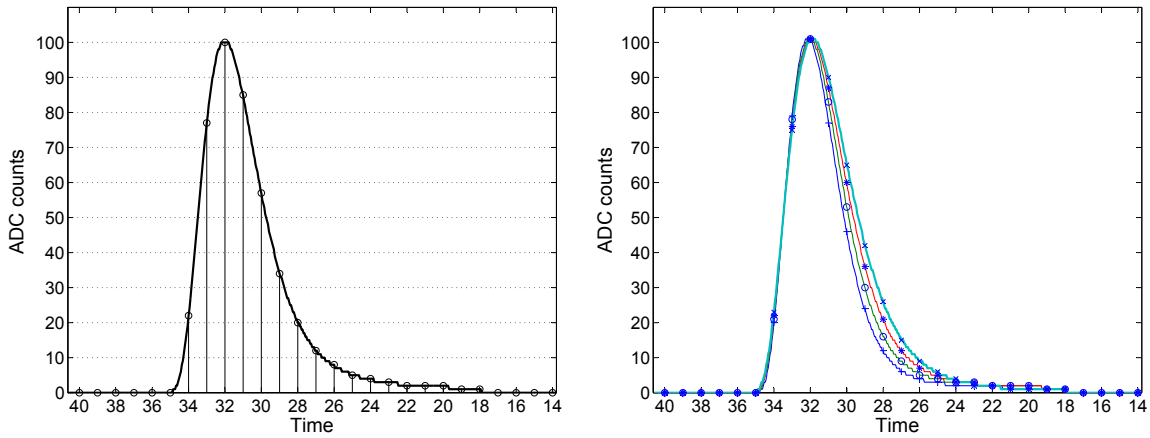
6.1. Test setup and test-data

The DataCompressor is tested first in the full compression mode by setting the $DcMode$ to '11'. Then, it is tested in the Huffman only mode by setting the $DcMode$ to '01'.

In the full compression mode, the VectorQuantizer1QSimple module is instantiated inside the DataCompressor top module to test the DataCompressor version, which uses only one reference vector (1Q version). The full compression mode using all four reference vectors (4Q version) is tested subsequently by instantiating the VectorQuantizer4Q module instead of the VectorQuantizer1QSimple module.

All reference vectors have a maximum element with value of 100 ADC counts and therefore the $NormValue$ register for the normalization is programmed to 100. This $NormValue$ and the reference vectors itself are determined in Matlab by calculating the average of the input waveforms in

the measured data set from the ALICE TPC and optimizing the reference vectors according to the LBG algorithm. The reference vectors are stored in the memories of the DataCompressor in a way that their maximum element is saved at position 32, which requires that the $NormValueIndex$ register is initialized with 32. This is important for the correct alignment of the reference vectors with the input vectors (normalized input waveforms). The used reference vectors for the 1Q version and the 4Q version are fitted with the semi-Gaussian function and shown together with their elements(samples values) in figure 6.1.



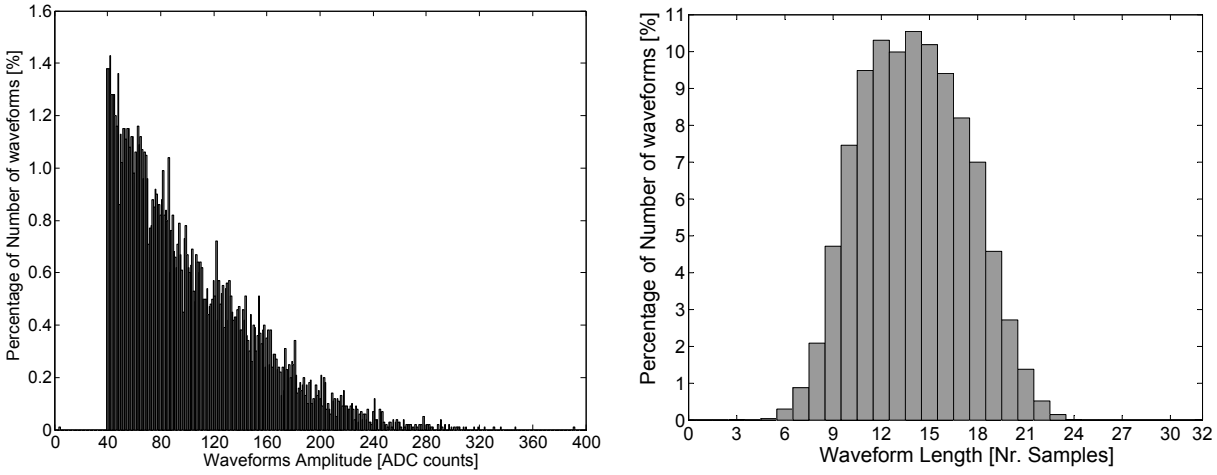
(a) Fitted reference vector for the 1Q version (b) The four fitted reference vectors for the 4Q version

Figure 6.1.: The reference vectors fitted with the semi-Gaussian function and their elements used in the full compression mode of the DataCompressor in versions 1Q and 4Q

The x-axes numbering represents the position numbers of the elements (samples) as saved in the memories. The samples from 1 to 14 and from 40 to 64 have a value of 0 and are cut out in the figures to improve the readability. The y-axis numbering represents the ADC counts from the ALTRO chip with 10-bit resolution. For the alignment of the reference vectors with the input vectors, the maximum compressible waveform length of 32 samples is considered, which lead to the defined length of the reference vectors to be 64 (2×32) elements with the maximum at 32. This guarantees the possibility to align the input vector with the reference vector independent of the position of the maximum sample in the input vector.

The test pattern for testing the DataCompressor consists of waveforms from the measured data set from the ALICE TPC measuring cosmic rays. The test pattern data from the cosmic ray run of the ALICE TPC in 2006 are formatted according to the ALTRO format shown in 5.3 and contain 10 002 waveforms (140 311 samples). The time stamps related to the waveforms in the test pattern data contain not the original values but continuous numbers for improving the waveform identification during the tests. The time stamps are not compressed in this work but included unchanged in the output data and therefore their exact value is not important for the tests of the DataCompressor. The waveforms in the test pattern are zero suppressed and have variable numbers of samples. The waveform lengths in the test pattern data are chosen according to the assumed analogue signal duration (around 600 ns) and a sampling frequency of 40 MHz(according to the new S-ALTRO project specifications). The test pattern contains also a waveform with a too large number of samples (86) and another one with only 3 samples. These two waveforms cannot be compressed in the full compression mode and are used to test the correct embedding of them in uncompressed form in the output bitstream of the DataCompressor. The distributions of the waveform lengths and amplitudes in the test pattern data are shown in figure 6.2.

The Huffman codebooks for the 1Q version and the 4Q version as well as for the Huffman Only mode are calculated in Matlab using the test pattern data and are given in appendix A.



(a) Histogram of the maximum sample values of the waveforms

(b) Histogram of the waveform lengths

Figure 6.2.: Histogram of the amplitudes and waveform lengths (number of samples) in the test pattern data

6.2. Simulation results

Two signal generators are used for the simulation, the NC-Sim (Simvision) from Cadence and the ISE tool from Xilinx. With these two simulators, the functionality of the developed RTL Verilog model is simulated and tested. At the beginning, the signals are tested and analyzed manually, but to check the compression of the large number of waveforms in the test pattern data, several contents of registers are dumped during simulation into text files. Afterwards, these files are imported into an Excel sheet, where the contents are compared with results obtained by the Matlab model of the DataCompressor.

The Matlab model is executed on the same test pattern data that are used for the simulations of the RTL code described above. The test pattern data are organized in Matlab in a matrix in which each row contains the samples of one waveform. Then the maximum sample of each waveform is searched in Matlab and the normalization factors are calculated. To model in Matlab the shift of the $NormValue$ 6 positions to the left the $NormValue$ is multiplied by 2^6. In the same way the right shift after the multiplication of the samples with the normalization factor is performed by dividing the results by 2^6. The resulting normalized waveforms are then organized in rows of a new matrix so that they all have their maximum sample in column 32. In this way, the alignment of the normalized waveforms with the reference vectors can be performed easily in Matlab. Afterwards, the Delta value sets can be calculated by subtracting the reference vectors from each input vector. A Matlab implementation of a Huffman coding algorithm is used to encode the Delta values. With the same Huffman algorithm the codebook, which is stored in the LUT memories inside the DataCompressor implementation is create. Important output values of the Matlab model resulting from the different steps of the data compression algorithm (e.g. max values, normalized sample values, Delta values etc.) are copied into the Excel sheets. The Excel sheets are then used for the comparison of the results from the Matlab model with the ones obtained by the simulations of the RTL Verilog model and by the hardware tests. An exact match of the results from both models shows the correct functionality of the DataCompressor implementation on the test pattern data.

In the following, the simulation tests are presented by using some screen shots from the signal generator and from Excel. The test procedure is organized according to the data flow in the DataCompressor:

- Checking the correct reading and unfolding of the ALTRO data packages from the RCU

internal buffer memories
- Checking the MaxFinder and Divider results
- Checking the Normalizer output
- Checking the Delta values from the two versions of the VectorQuantizer
- Checking the Huffman Coder for the correct addressing, reading and concatenating of the Huffman codewords

The first test is dedicated to the correct reading and unfolding of the ATLRO data package from the RCU internal buffer. It has to be guaranteed that the splitting of the data in samples, time stamps and cluster widths is performed correctly for all input waveforms. A small section of the analysis of the relevant signals using the signal generator implemented in the ISE tool is shown in figure 6.3.

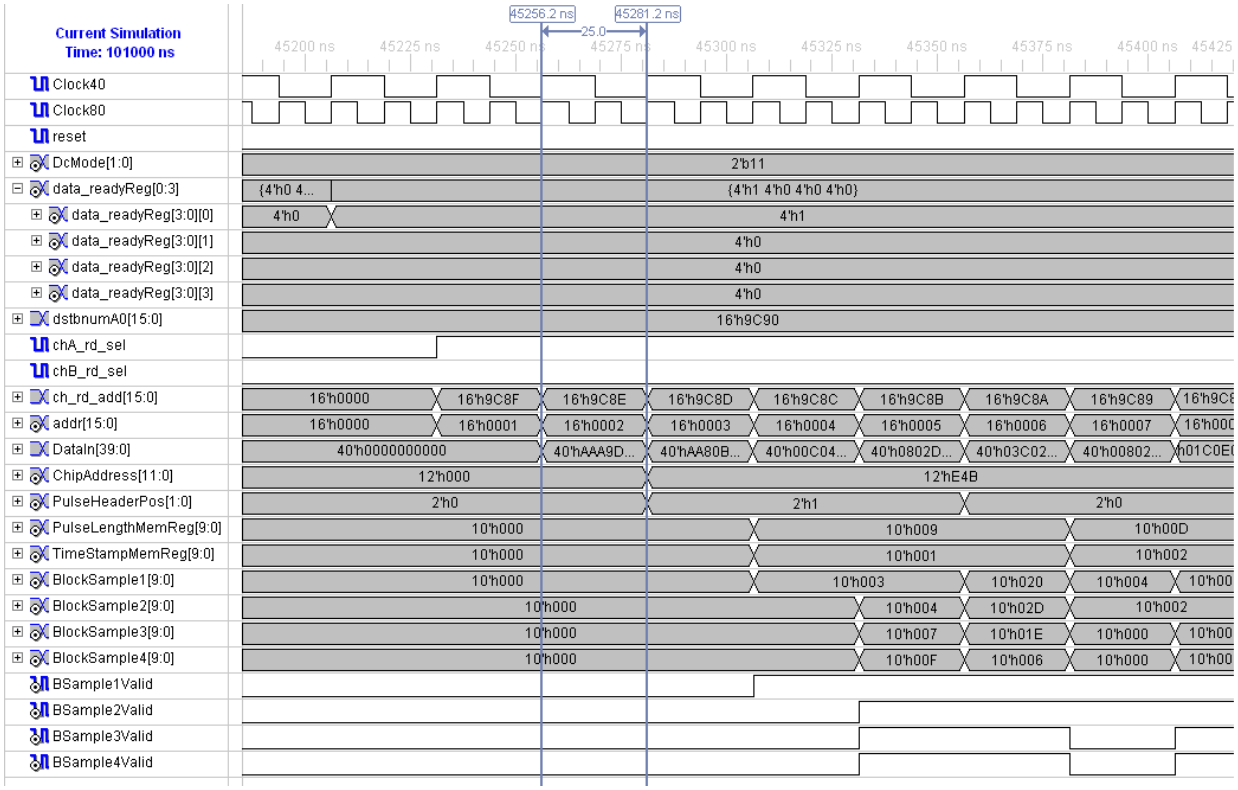


Figure 6.3.: Signal diagram showing the reading and unfolding of the input data from the RCU internal buffers (test patter)

After the $data\_readyReg$ signal changes, the ch_A_rd_sel goes high and the ch_rd_add starts decreasing from the value received by $dstbnumA0$. The addressed words from the internal RCU buffer are read in through the $DataIn$ bus and are split into their components. The first word contains the $Nr10BitWords$ and the $ChipAddress$. The OutputFormatter adds the $ChipAddress$ to the output data package. The $Nr10BitWords$ is used in combination with the $dstbnumA0$ to calculate the first waveform header position named in the Verilog model as $PulseHeaderPos$. The first $PulseHeaderPos$ value gives the position of the first cluster width word, which is saved in the $PulseLengthMemReg$ register. The following 10-bit word represents the first time stamp and is stored in $TimeStampMemReg$. These two words and the following cluster width words and corresponding time stamp words are buffered in the dedicated RAM memories until the OutputFormatter adds them to the output data package. The cluster width words are also used to calculate the corresponding next $PulseHeaderPos$. The sample values of the waveforms are transferred to the $BlockSample1$-$BlockSample4$ registers. The signals $BlockSample1Valid$-$BlockSample4Valid$ indicate which of the $BlockSample$ registers contain valid sample values

per $Clock40$ cycle. The $BlockSample$ values are sent to the Normalizer together with the cluster width value in the $PulseLengthMemReg$. It can be seen that sample values are saved in the $BlockSample$ register each $Clock40$ cycle, without any dead time. This is normally the case unless there are waveforms with a too large number of samples that cannot be compressed and require an introduction of dead time. Due to the fact that four samples are received contemporary while only three samples can be included uncompressed in the output data, after every third $Clock40$ cycle the reading of the RCU internal buffer is interrupted for one cycle. Additional dead time can be introduced by setting the da_bsy signal high, which can be useful e.g. when buffers in the DataCompressor tent to overflow. This is not implemented in the actual version because the buffers are designed (sized) to cope with a constant input data stream of the full test pattern data (10 002 waveforms with 140 311 samples).

To be able to test the correct unfolding of the 10 002 waveforms in the test pattern data the manually evaluation through the waveform simulator is not adequate and therefore the method with the Excel sheets is used. The values of the $BlockSample$ registers are dumped during simulation according to the $BlockSampleValid$ high signals in a text file. This text file is imported into an Excel sheet and the values are compared with the results obtained from the Matlab model copied in the same Excel sheet (original concatenated samples from the test pattern data). A cut-out of the Excel sheet is shown in table 6.1.

Table 6.1.: A cut-out of the Excel sheet used to check the correct unfolding of the ALTRO data package in the Data Compressor compared with the Matlab model

	A	B	C	D	E	F	G	H	I	J	K
1											
2	Input_Data	Input_Data	Check Equal		ClusterWidths	ClusterWidths	Check Equal		TimeStamps	TimeStamps	Check Equal
3	Stream Matlab	Stream Verilog			Matlab	Verilog			Matlab	Verilog	
4	3	3	TRUE		9	9	TRUE		1	1	TRUE
5	3	3	TRUE		13	13	TRUE		2	2	TRUE
6	4	4	TRUE		9	9	TRUE		3	3	TRUE
7	7	7	TRUE		11	11	TRUE		4	4	TRUE
8	15	15	TRUE		10	10	TRUE		5	5	TRUE
9	32	32	TRUE		13	13	TRUE		6	6	TRUE
10	45	45	TRUE		9	9	TRUE		7	7	TRUE
11	30	30	TRUE		9	9	TRUE		8	8	TRUE
12	6	6	TRUE		12	12	TRUE		9	9	TRUE
13	4	4	TRUE		12	12	TRUE		10	10	TRUE
14	2	2	TRUE		86	86	TRUE		761	761	TRUE
15	2	2	TRUE		13	13	TRUE		11	11	TRUE
16	2	2	TRUE		10	10	TRUE		12	12	TRUE
17	4	4	TRUE		12	12	TRUE		13	13	TRUE
18	7	7	TRUE		11	11	TRUE		14	14	TRUE
19	7	7	TRUE		13	13	TRUE		15	15	TRUE
20	14	14	TRUE		13	13	TRUE		16	16	TRUE
21	32	32	TRUE		10	10	TRUE		17	17	TRUE
22	59	59	TRUE		12	12	TRUE		18	18	TRUE
23	63	63	TRUE		11	11	TRUE		19	19	TRUE
24	23	23	TRUE		10	10	TRUE		20	20	TRUE
25	6	6	TRUE		10	10	TRUE		21	21	TRUE
26	3	3	TRUE		3	3	TRUE		1002	1002	TRUE
27	5	5	TRUE		14	14	TRUE		22	22	TRUE
28	7	7	TRUE		17	17	TRUE		23	23	TRUE
29	15	15	TRUE		15	15	TRUE		24	24	TRUE
30	28	28	TRUE		13	13	TRUE		25	25	TRUE
31	42	42	TRUE		12	12	TRUE		26	26	TRUE
32	39	39	TRUE		15	15	TRUE		27	27	TRUE
33	23	23	TRUE		12	12	TRUE		28	28	TRUE
34	17	17	TRUE		9	9	TRUE		29	29	TRUE
35	2	2	TRUE		7	7	TRUE		30	30	TRUE
36	3	3	TRUE		15	15	TRUE		31	31	TRUE
37	8	8	TRUE		9	9	TRUE		32	32	TRUE

Input | CheckMmax | MaxIndex | NormFactor | NormPulses | Delta1Q | 1QOut | 1QOutBadPulse | 32BitOut

In Excel the command “exact” compares two cells and returns TRUE if their content is exactly the same or FALSE if they are not equal. With the filter function in Excel the rows can be found,

which contain a FALSE check result. If all comparisons return TRUE the Verilog code can be considered to work correctly. The 10 002 cluster width words and time stamp words are as well compared with the ones obtained by the Matlab model in the Excel sheet.

The next tests are made to investigate the correct execution of the MaxFinder and Divider in the Normalizer module. To analyze the functionality in the time domain again the ISE signal generator is used and a cut-out of the signal diagram is shown in figure 6.4.

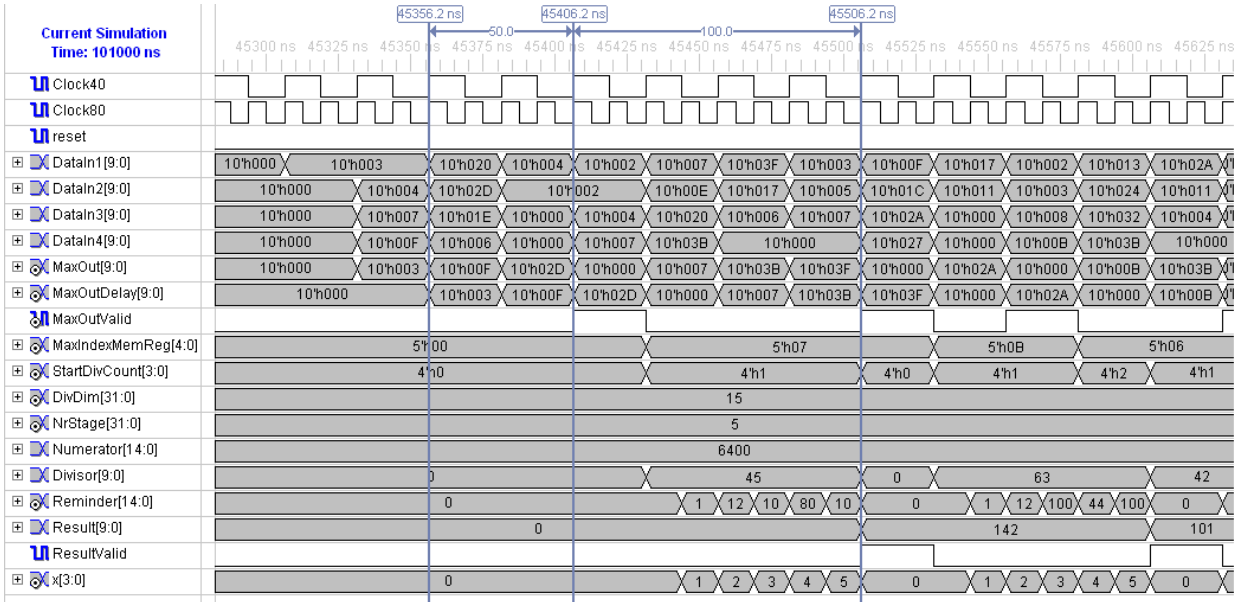


Figure 6.4.: Signal diagram showing the MaxFinder and Divider time requirement

The MaxFinder output is saved in the $MaxOut$ register which shows the maximum value of the four received samples per $Clock40$ cycle. The MaxFinder logic needs one $Clock40$ cycle to determine the $MaxOut$ values so that the value corresponds to the maximum of the samples seen in the previous cycle. If the maximum value in the $MaxOut$ register is detected as the maximum of the entire actual waveform the $MaxOutValid$ signal goes high. For the identification of the $MaxOut$ value as the maximum of the waveform the MaxFinder needs to wait for the following $MaxOut$ value which is lower than the actual one. Therefore, the $MaxOutValid$ signal is delayed for one $Clock40$ cycle in respect to the $MaxOut$ value. To compensate for this delay, the $MaxOut$ value is as well delayed for one $Clock40$ cycle by copying it into the $MaxOutDelay$ register which than is aligned with the $MaxOutValid$ signal. To check that the maximum values for all 10 000 waveforms (not including the too long and to short waveform) are found correctly the values in the $MaxOutDelay$ register are dumped in a text file each time the $MaxOutValid$ signal is high. The file is imported into an Excel sheet and checked against the maximum values given by the Matlab model. The register $MaxIndexMemReg$ contains the position of the found maximum sample of the received waveform and is checked as well in the Excel sheet. The comparison with the Matlab model showed that all maximum values are found correctly and the $MaxIndexMemReg$ values are calculated correctly.

The $MaxOutValid$ high signal starts the divider, which uses the $MaxOutDelay$ values as the divisor. The numerator value is given by the $NormValue$ (100) shifted 6 positions to the left (6400). The divider performs the division in six $Clock80$ cycles which is defined by the $NrStage$ parameter (5 pipelined stages) and can be seen by the changes of the $Reminder$ register value and the x value. After the five stages, the result is stored in the register $Result$. A high level of the signal $ResultValid$ shows that the divider output is ready and can be used for the normalization. The value in the $Result$ register and the high level of the $ResultValid$ signal are held unchanged for

two $Clock80$ cycles to guarantee the synchronization with the remaining logic running at 40 MHz. The $Result$ value is written in a file each time a $ResultValid$ high signal is detected to compare it in Excel with the Matlab model results. All calculated normalization factors (divider results) from the VQDiv module turned out to be correct. Figure 6.4 shows the latency between the input of the maximum sample until the normalization factor (divider result) is calculated consisting of 6 $Clock40$ cycles. The Verilog code of the Divider has been tested as well as a standalone module for an eventual use in other projects. The high level of the $ResultValid$ signal starts the multiplication of the sample values with the calculated normalization factor and the shift back of 6 positions to obtain the normalized samples. All 140 222 normalized samples from the 10 000 waveforms are written in a file and compared in Excel with the normalized samples from the Matlab model to approved the functionality of the Normalizer.

After that, the Delta calculation in the VectorQuantizer is tested. First, the correct initialization of the reference vector memories is checked manually with the ISE simulator displaying the memory contents. Then the correct performance of the VectorQuantizer module for the 1Q version and the 4Q version is simulated and the calculated $DeltaW0$-$DeltaW3$ values are dumped into a file each time the $DeltaW0Valid$-$DeltaW3Valid$ signals are high. These are checked then in Excel against the Delta values calculated with the Matlab model. Two tests are made, first instantiating the VectorQuantizer1QSimple module to check the $DeltaW$ values using only one reference vector. The second test is made with the VectorQuantizer4Q module being instantiated to check the $DeltaW$ values resulting by using four reference vectors. In the 4Q version, the summation of the $DeltaW$ values for each input waveform is checked as well to ensure the correctness of the best matching reference vector selection. At the end the index, which is sent out to preserve the information of the chosen reference vector per input vector is checked in an Excel sheet and turned out to be consistent with the Matlab model.

Then the Huffman coding of the Delta values is tested. At the beginning, the correct initialization of the Huffman codebook in the LUT memories is checked manually in the ISE waveform simulator. The readout of the Huffman codewords according to the Delta values and the concatenation can be seen in the signal diagram screenshot which is presented in figure 6.5.

To use the Delta values as address for the LUT in the memories the value 512 is added which results in the values seen in the registers $HuffIn1$-$HuffIn4$. In case the HuffmanOnly mode is selected ($DcMode$='01') the inputs $HuffIn1$-$HuffIn4$ receive the original samples of the test pattern data as read from the RCU internal buffers. The addressed words of the LUT are read out from the memories and transferred to the ConCatenator module. The signals $HuffW1AValid$-$HuffW2BValid$ indicate which Huffman words (HuffW1A-HuffW2B) have to be concatenated. The correct readout of the Huffman codewords is checked by random visual inspection in the ISE waveform simulator.

The ConCatenator60 concatenates the first two Huffman words and stores them in $ConcRegA$. The second two Huffman words are concatenated and stored in $ConcRegB$ contemporaneously. The 5 MSBs of the Huffman codewords HuffW1A-HuffW2B are representing the Huffman word length and only the remaining relevant bits of the Huffman codewords can be seen concatenated in the $ConcRegA$ and $ConcRegB$ register. These two registers are concatenated then in the next $Clock40$ cycle and stored in $ConcRegAB$. In the following $Clock40$ cycle the relevant bits of the $ConcRegAB$ are added to the bits of previous received Huffman codewords saved in the $Temp$ register. The new concatenated word is stored in the $HuffmanWord$ register. If 60 bit are concatenated and stored in the $HuffmanWord$ the $HuffmanValid$ signal goes high. This indicates that the bits in the $HuffmanWord$ register can be added to the DCOP by the OutputFormatter.

To check if all Huffman codewords are concatenated correctly the content of $HuffmanWord$ is written in a file each time the $HuffmanValid$ signal is high. The file is read into an Excel

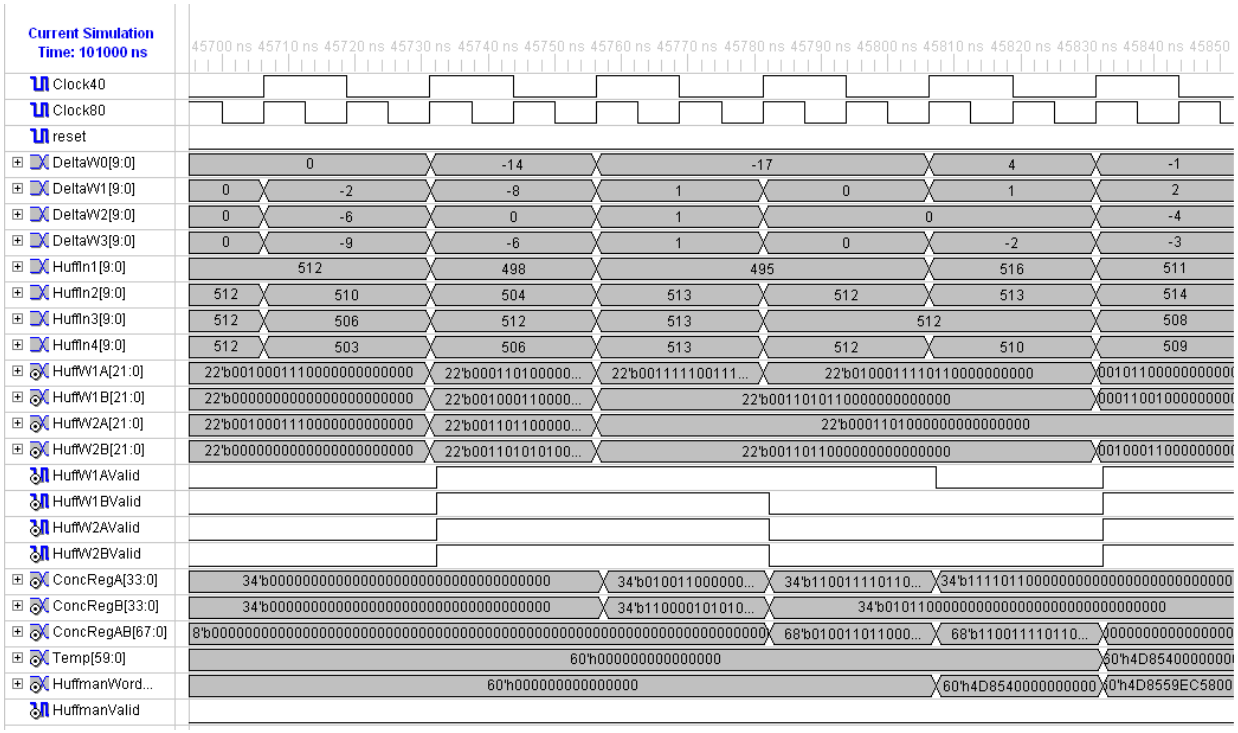


Figure 6.5.: Signal diagram of the Concatenation of the Huffman codewords corresponding to the Delta values

sheet and compared with the resulting Huffman codewords from the Matlab model.

To investigate the latency introduced by the DataCompressor a last signal diagram is shown in figure 6.6. The cut-out of the signal diagram shows the time delays from the first received input samples to the different steps of the DataCompressor until the first bits are stored in the $HuffmanWord$ (output bitstream). The latency between input and output is not constant and the shown time delays have to be considered as an example. The latency between the receptions of the first samples of an input waveform and the $MaxOutValid$ high signal depends on the waveform length and the position of the maximum sample in the waveform. Therefore, this latency depends on the signal shape and is not constant. The latency between the detection of the maximum sample ($MaxOutValid$=high) and the divider result is constant and results in 4 $Clock40$ cycles. As well 4 $Clock40$ cycles of latencies arise between the divider result and the signal named with $NormPulseReady$ indicating the end of the normalization of a waveform. Also between the $NormPulseReady$ and the output of the calculated Delta values the latency corresponds to 4 $Clock40$ cycles as shown in figure 6.6. The HuffmanCoder adds a fixed latency of 4 $Clock40$ cycles from the receiving of a Delta value until the corresponding Huffman codeword bits for this Delta value are added to the output bitstream in the $HuffmanWord$. The time delay until enough Huffman codewords are concatenated to result in a valid 60 bit $HuffmanWord$ is also variable and depends on the length of the Huffman codewords related to the Delta values.

The overall latency of the DataCompressor from the input samples to the Huffman codeword outputs cannot be given as a fixed value but is estimated to be in average 30 $Clock40$ cycles for the 1Q version.

The 4Q version has 2 $Clock40$ cycles longer latency resulting in 32 $Clock40$ cycles because the VectorQuantizer4Q requires extra time to determine the best matching reference vector.

The latency for the HuffmanOnly mode is given in average by 10 $Clock40$ cycles.

These average latencies are calculated using the test pattern data described above and can

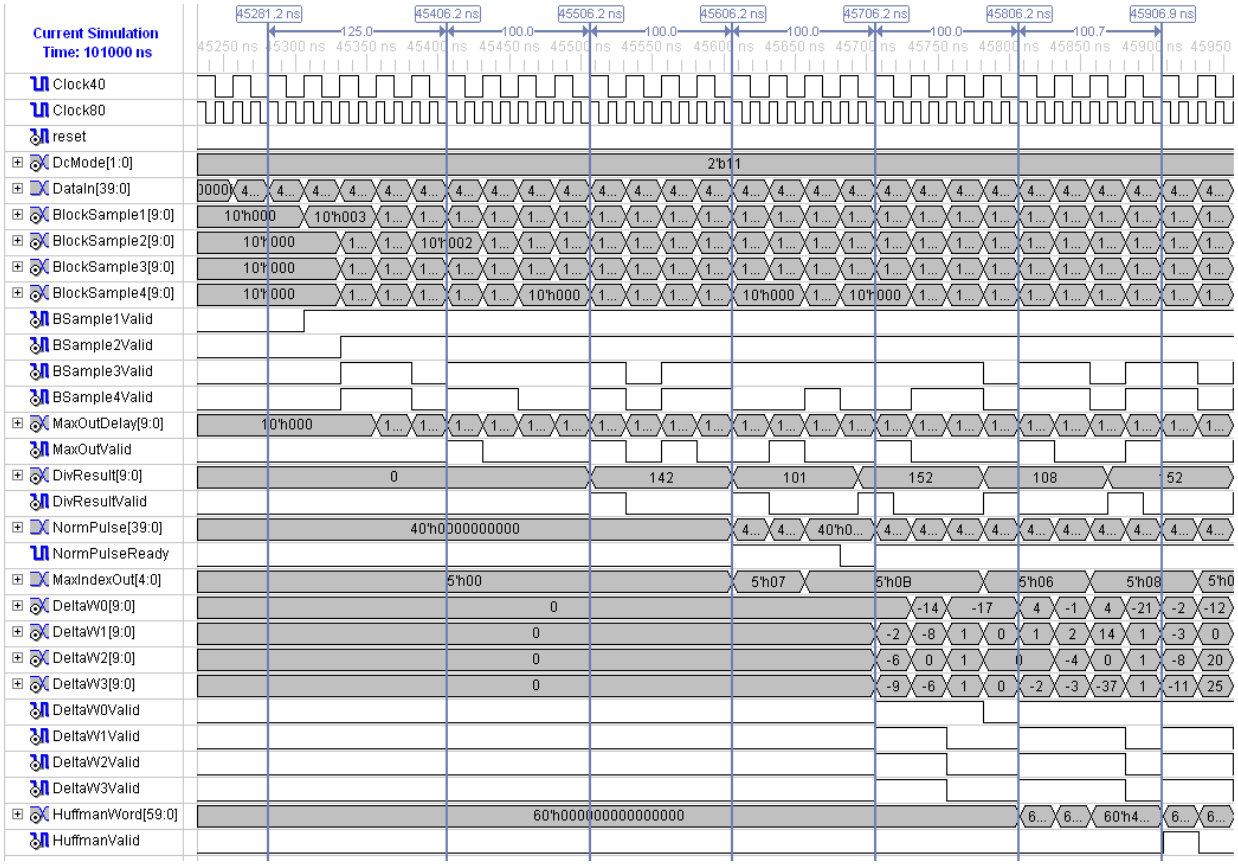


Figure 6.6.: Signal diagram used to investigate the latency of the DataCompressor

change for other input data.

6.2.1. Final simulation check using the Decompressor

The final check of the correct execution of the DataCompressor described in Verilog is performed by dumping the resulting output data package created by the OutputFormatter in a file. This file is imported in Matlab and a model of a **Decompressor** programmed in Matlab (given in appendix B) starts unfolding and decompressing the DCOP. The Decompressor is decoding and reconstructing the unfolded Huffman bitstream in the payload to obtain the original waveform samples. These samples can then be compared with the original data of the test pattern.

For the reconstruction of the original data, the Decompressor first needs to decode the Huffman bitstream contained in the payload of the DCOP. Therefore, it uses the same Huffman codebook as loaded in the DataCompressor prior to operations. The bits contained in the Huffman bitstream are compared to the Huffman codewords in the codebook and the corresponding Delta values are reconstructed. The decoder processes the bitstream sequentially. The $NrSamples$ words for every waveform that are contained in the waveform headers of the DCOP are used to group the obtained Delta values in rows of a matrix corresponding to the waveform lengths. If a waveform header in the DCOP contains a NrSamples higher than 32 or less than 4 the Decoder knows that the following bits in the bitstream are representing not compressed data and have to be split in groups of 10 bit. These groups are representing the original samples of the corresponding waveform and are written directly in the original data matrix. After all samples of 10 bit are saved regarding to the NrSamples, the Decompressor continues with the Huffman decoding of the Delta values. For the reconstruction of the normalized waveforms, the Decompressor needs to know which reference vector has been used by the DataCompressor for each input vector. For the 1Q version, only one

reference vector exists and the Decompressor uses this one for all Delta values. In the 4Q version, the Decompressor uses the indices sent by the DataCompressor to identify the correct reference vector for each row of Delta values. To align the reference vector with each row of the Delta value matrix the Decompressor searches the first 0 value scanning the Delta values rows from the right (last received 0 Delta value for each input waveform). The position of the first 0 from the right tells the position of the maximum sample in this input waveform. The aligned Delta values are summed up to the elements of the reference vector. The resulting normalized waveforms are saved in rows of a matrix. The Decompressor uses the received $MaxValues$ contained in the waveform headers of the DCOP to calculate the normalization factor for each normalized waveform in the same way as the DataCompressor did. Therefore the Decompressor needs to know the $NormValue$ which was programmed into the DataCompressor prior to operation (being 100 in our case). Each row of the normalized waveform matrix is then multiplied with the corresponding normalization factor to reconstruct the original sample values of the test pattern data. The reconstructed samples by the Decompressor can be compared to the original samples of the test pattern data. The comparison showed that most of the samples are equal. From all reconstructed samples only around 16% differ from the original values, which can be explained by an error introduced from the limited precision of the divider and the multipliers used in the Normalizer and by the manipulation of some Delta values for the alignment information. Less than 0.25% of the reconstructed samples have an error of more than ± 1 ADC count. The percentage of the erroneous samples is not differing significantly between the 1Q version and the 4Q version.

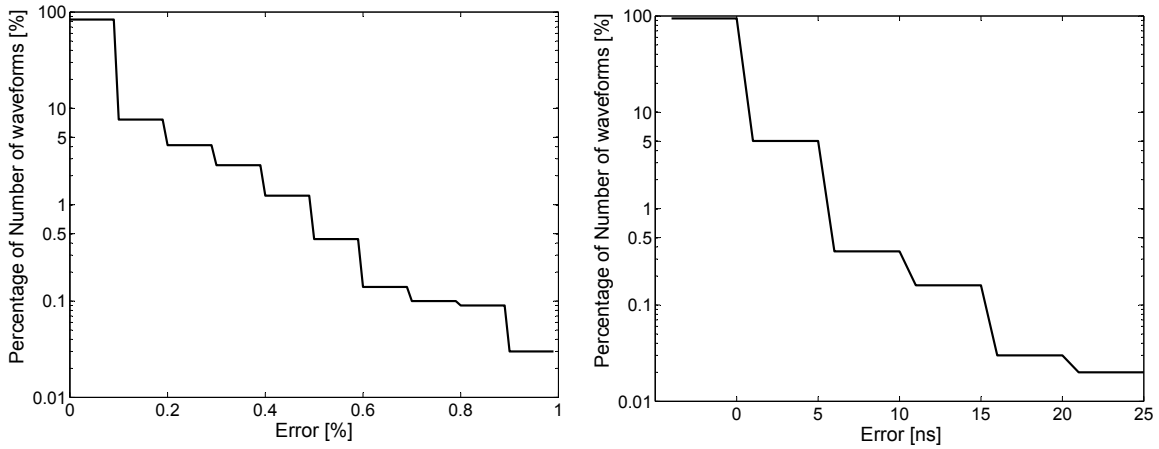
To investigate the effect of this error on the important waveform parameters (amplitude and peak time) the original waveforms and the reconstructed waveforms is fitted in Matlab by using the gamma-4 function and the "curvefit" command. Each original input waveform is compared to the corresponding reconstructed waveform and the differences in amplitude and peak time are calculated. The distribution of the difference in amplitude normalized to the original amplitude of the waveforms is shown in % in figure 6.7(a). The distribution of the error in time of the position of the maximum sample values of the waveform is shown in nanoseconds in figure 6.7(b). The error distribution histograms shown in 6.7 result from the analysis of all 10 002 waveforms contained in the test pattern. Around 85% of the reconstructed waveforms have an error in amplitude less than 0.1%. The remaining waveforms have an error above 0.1%, among them only a few reconstructed waveforms have errors up to 1%. No reconstructed waveform was seen with an error in amplitude higher than 1%. The error in time is even smaller resulting in around 95% of the reconstructed waveforms having no time error. Around 5% of the waveforms show errors of up to 5 ns. Only a few reconstructed waveforms have errors above 5 ns (at most 25 ns). Considering other sources of imprecision in the front-end electronics (ADC, digital signal processor), these errors can be tolerated easily. An increase of the divider and multipliers resolution would farther reduce the error by the cost of additional hardware. In addition, more effort in controlling and improving of the fit results could reduce some obtained error.

After the functionality of the DataCompressor is approved and the introduced error by the limited precision of the logic is analyzed and considered to be not critical the compression performance is discussed.

The compression performance achieved by the DataCompressor realization in Verilog is calculated. The resulting performance of the DataCompressor in the different modes of operation is presented in the diagram in figure 6.8.

As it can be seen from the diagram the results obtained by compressing the 10 002 waveforms of the test pattern data are close to the theoretical results obtained by the Matlab investigation of the algorithm presented in chapter 4. This proves that a data compression based on the developed algorithm can be realized in hardware providing the expected compression performance.

To ensure the functioning of the DataCompressor in hardware an implementation of the RTL



(a) Amplitude Error

(b) Time Error

Figure 6.7.: Histogram of the error in amplitude (left) and time (right) between the reconstructed waveforms and the original waveforms. The error in amplitude is given in % of the original amplitude and the error in time is given in nanoseconds. The number of waveforms corresponding to the error is given in % of the total analyzed waveforms

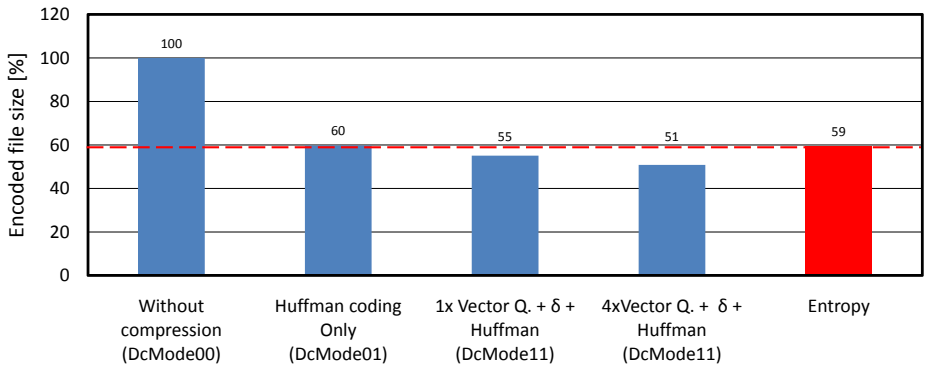


Figure 6.8.: Compression performance of implementation on zero suppressed data

Verilog model is performed inside a FPGA. The results of the hardware tests are presented in the following.

6.3. Hardware results

To investigate if the written Verilog code is synthesizable and works as expected also in hardware it is implemented in a FPGA development board from Xilinx [51]. The development board with the name ML401 contains a Virtex-4 LX25 FPGA as shown in the image in figure 6.9.

The synthesis and Place&Route of the RTL Verilog model are executed in the Xilinx ISE tool version 10.1.03. The settings for the Xilinx Synthesis Technology (XST) tool in ISE are set to the default values except for the following changes:

— Optimization goal: AREA
— Optimization effort: High
— FSM Encoding Algorithm: Gray

The synthesis is set to optimize the area of the DataCompressor module in order to minimize the required resources in the FPGA. Both the 1Q version and the 4Q version of the DataCompressor in combination with the DataCompression2ClockExt_Top module are synthesized and tested with

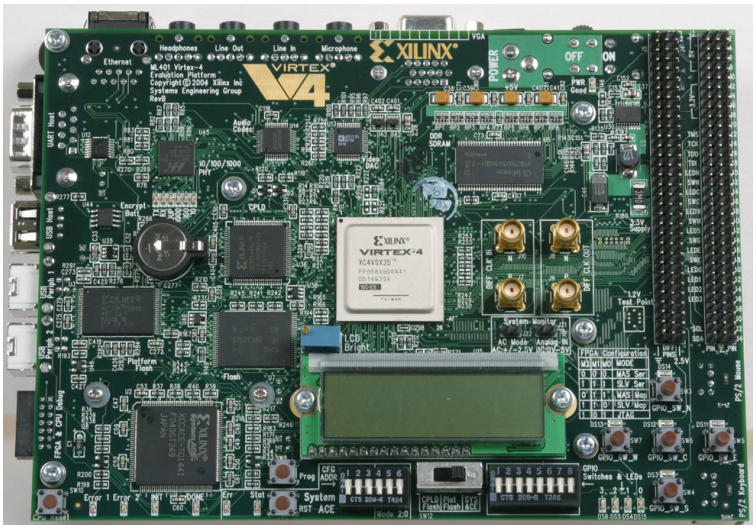


Figure 6.9.: FPGA development platform from Xilinx called ML401 containing a Virtex-4 [51]

the Virtex-4 FPGA. In addition, the DataCompressor is also tested in the Huffman Only mode (DcMode='01'). The DataCompression2ClockExt_Top module is required to control the data flow to and from the DataCompressor module so that the DataCompressor functionality can be tested. A summary of the synthesis report is shown in figure 6.10 and in figure 6.11.

Figure 6.11 shows the device utilization summary for the 1Q version and for the 4Q version. The device utilization for the 4Q version is much higher as for the 1Q version because the VectorQuantizer4Q module is not optimized in terms of area requirements for the actual design. For the use in future applications some registers in the module can be exchanged with block RAM memories to reduce significantly the area. For the moment I optimized only the area of the 1Q version to prepare it for an eventual implementation in the Virtex-2P FPGA of the actual RCU in the TPC of ALICE to show the operability. More information about a first implementation in the RCU of the ALICE experiment is given in chapter 7. Nevertheless, figure 6.11(b) shows that the functionality of the 4Q version can be tested with the development board using the resources of the Vitrex-4 FPGA.

To better understand the resource requirements a synthesis of the DataCompressor as a standalone module is performed and the device utilization summaries for the 1Q version and the 4Q version are given in figure 6.12.

The main difference in the resource requirements between the standalone version and the previous combined version can be seen in the usage of the block RAM memories. The combination with the DataCompression2ClockExt_Top for testing purposes uses more RAM memories to store the initialization data and the test pattern data, which are sent to the DataCompressor. The output data of the DataCompressor are saved in a RAM memory, from where the data are sent to the PC via the UART serial interface. As well, the number of used IO buffers is high but this is from minor importance because the DataCompressor is intended to be implemented in a larger system and internally connected to logic, not to input and output pins. The timing analysis summary of the synthesis report is shown in figure 6.13(a) and results from estimations of the synthesizer taking into account the delays of the logic.

The timing constraints for the $Clock40$ and $Clock80$ signals are the same for all the synthesis performed for the Virtex-4. This estimations for the timing delay are giving a maximum frequency that can be applied to the $Clock40$ and $Clock80$ signals. The critical path for the timing (path with the maximum delay) is inside the Divider which required the pipelined structure of it as explained above. The maximum usable frequencies are higher as the specified frequencies for the DataCompressor, which allows to implement the DataCompressor with the foreseen clock frequen-

```
=========================================================================
*                     Synthesis Options Summary                         *
=========================================================================
---- Source Parameters
Input File Name                    : "DataCompression2ClockExt_Top.prj"
Input Format                       : mixed
Ignore Synthesis Constraint File   : NO

---- Target Parameters
Output File Name                   : "DataCompression2ClockExt_Top"
Output Format                      : NGC
Target Device                      : xc4vlx25-12-ff668

---- Source Options
Top Module Name                    : DataCompression2ClockExt_Top
Automatic FSM Extraction           : YES
FSM Encoding Algorithm             : Gray
Safe Implementation                : No
FSM Style                          : lut
RAM Extraction                     : Yes
RAM Style                          : Auto
ROM Extraction                     : Yes
Mux Style                          : Auto
Decoder Extraction                 : YES
Priority Encoder Extraction        : YES
Shift Register Extraction          : YES
Logical Shifter Extraction         : YES
XOR Collapsing                     : YES
ROM Style                          : Auto
Mux Extraction                     : YES
Resource Sharing                   : YES
Asynchronous To Synchronous        : NO
Use DSP Block                      : auto
Automatic Register Balancing       : No

---- Target Options
Add IO Buffers                     : YES
Global Maximum Fanout              : 500
Add Generic Clock Buffer(BUFG)     : 32
Number of Regional Clock Buffers   : 24
Register Duplication               : YES
Slice Packing                      : YES
Optimize Instantiated Primitives   : NO
Use Clock Enable                   : Auto
Use Synchronous Set                : Auto
Use Synchronous Reset              : Auto
Pack IO Registers into IOBs        : auto
Equivalent register Removal        : YES
```

(a) First part of the settings

```
---- General Options
Optimization Goal                  : Area
Optimization Effort                : 2
Power Reduction                    : NO
Library Search Order               : DataCompression2
Keep Hierarchy                     : NO
Netlist Hierarchy                  : as_optimized
RTL Output                         : Yes
Global Optimization                : AllClockNets
Read Cores                         : YES
Write Timing Constraints           : NO
Cross Clock Analysis               : NO
Hierarchy Separator                : /
Bus Delimiter                      : <>
Case Specifier                     : maintain
Slice Utilization Ratio            : 100
BRAM Utilization Ratio             : 100
DSP48 Utilization Ratio            : 100
Verilog 2001                       : YES
Auto BRAM Packing                  : NO
Slice Utilization Ratio Delta      : 5
```

(b) Second part of the settings

Figure 6.10.: Summary of the settings for the synthesis of the DataCompressor module in combination with the DataCompression2ClockExt_Top module

cies in the Virtex-4. According to investigations for an implementation of the DataCompressor in the Virtex-2P FPGA of the ALICE RCU the synthesis shows that the maximum clock frequencies for this device are closer to the specified frequencies. Nevertheless, the estimated maximum frequencies show that they have still some margin regarding the specified clock frequencies for the DataCompressor. The number of foreseen pipeline stages for the divider are suitable for a future implementation of the DataCompressor in the RCU.

After the synthesis is performed, the remaining design implementation steps are executed in ISE until the Place&Route model is obtained and a programming file is created. The ISE default settings are used for these steps. A Post-Place&Route Simulation model is created as well to perform a sign-off simulation of the DataCompressor design [52]. For this model the option -ism is included in the command-line, which generates a netlist Verilog file where the Xilinx primitive models are included. In this way, the netlist generated after the Place&Route using logic primitives can be directly simulated with the ISE simulator or with the Cadence NC-Sim simulator.

In addition, a user constraint file is used in the project to include timing constraints, which are then checked in the Post-Place&Route simulation. The timing constraints are set to the clock frequencies of 40 MHz and 80 MHz. The simulation results of the Post-Place&Route netlist model are checked again by dumping some register contents in text files and comparing them in Excel against the results from the RTL Verilog model and the Matlab model. All tests confirmed that the Post-Place&Route model gives equal results as the behavioral model and the Matlab model. The timing checks resulted in a warning for a possible timing constrained violation in the UART module. Since the UART has been already tested with the development board and showed to work correctly for sending the output memory data to the PC, this warning was ignored because

Number of Slices:	2981	out of	10752	27%
Number of Slice Flip Flops:	2258	out of	21504	10%
Number of 4 input LUTs:	5370	out of	21504	24%
Number used as logic:	5330			
Number used as Shift registers:	40			
Number of IOs:	7			
Number of bonded IOBs:	7	out of	448	1%
IOB Flip Flops:	1			
Number of FIFO16/RAMB16s:	70	out of	72	97%
Number used as RAMB16s:	70			
Number of GCLKs:	2	out of	32	6%
Number of DSP48s:	4	out of	48	8%

(a) Device utilization summary for the 1Q version

Number of Slices:	10683	out of	10752	99%
Number of Slice Flip Flops:	5211	out of	21504	24%
Number of 4 input LUTs:	19480	out of	21504	90%
Number used as logic:	19228			
Number used as Shift registers:	40			
Number used as RAMs:	212			
Number of IOs:	7			
Number of bonded IOBs:	7	out of	448	1%
IOB Flip Flops:	1			
Number of FIFO16/RAMB16s:	72	out of	72	100%
Number used as RAMB16s:	72			
Number of GCLKs:	2	out of	32	6%
Number of DSP48s:	4	out of	48	8%

(b) Device utilization summary for the 4Q version

Figure 6.11.: The synthesis report device utilization summaries for the 1Q version and the 4Q version of the DataCompressor module in combination with the DataCompression2ClockExt_Top module

Number of Slices:	2686	out of	10752	24%
Number of Slice Flip Flops:	2018	out of	21504	9%
Number of 4 input LUTs:	4826	out of	21504	22%
Number used as logic:	4786			
Number used as Shift registers:	40			
Number of IOs:	286			
Number of bonded IOBs:	284	out of	448	63%
IOB Flip Flops:	58			
Number of FIFO16/RAMB16s:	18	out of	72	25%
Number used as RAMB16s:	18			
Number of GCLKs:	2	out of	32	6%
Number of DSP48s:	4	out of	48	8%

(a) Device utilization summary for the 1Q version

Number of Slices:	10197	out of	10752	94%
Number of Slice Flip Flops:	4905	out of	21504	22%
Number of 4 input LUTs:	18642	out of	21504	86%
Number used as logic:	18602			
Number used as Shift registers:	40			
Number of IOs:	286			
Number of bonded IOBs:	284	out of	448	63%
IOB Flip Flops:	58			
Number of FIFO16/RAMB16s:	24	out of	72	33%
Number used as RAMB16s:	24			
Number of GCLKs:	2	out of	32	6%
Number of DSP48s:	4	out of	48	8%

(b) Device utilization summary for the 4Q version

Figure 6.12.: The synthesis reports device utilization summaries for the 1Q version and the 4Q version of DataCompressor module as a standalone unit

it is not related to the DataCompressor itself. The timing constraints are met as well for the DataCompressor stand-alone simulation with the Virtex-2P FPGA for the RCU.

To test the DataCompressor module in a hardware environment, the generated programming file from the ISE tool is downloaded to the Virtex-4 FPGA of the development board. The clock signal for the Clcok80 is generated by a crystal oscillator from SaRonix mounted in the custom clock socket on the backside of the development board delivering a stable 80 MHz signal. The 40 MHz signal for the $Clock40$ is generated in the DataCompression2ClockExt_Top module by using a clock divider logic. Three buttons on the development board are used to control the test process. The names and positions of the FPGA pins connected to the user clock socked and the pins connected to the buttons can be found in the schematics of the ML401 development board available from the Xilinx homepage [53]. The button SW_S is used to initiate a reset and is connected to the $reset$ signal of the DataCompressor. The button SW_E starts the initialization phase in which

```
Minimum period: 8.592ns (Maximum Frequency: 116.388MHz)
Minimum input arrival time before clock: 8.282ns
Maximum output required time after clock: 6.874ns
Maximum combinational path delay: 5.382ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================
Timing constraint: Default period analysis for Clock 'Clock40'
Clock period: 8.592ns (frequency: 116.388MHz)
Total number of paths / destination ports: 583879 / 4075
---------------------------------------------------------

=========================================================
Timing constraint: Default period analysis for Clock 'Clock80'
Clock period: 6.848ns (frequency: 146.024MHz)
Total number of paths / destination ports: 645726 / 191
---------------------------------------------------------
```

(a) Timing analysis results for the 1Q version of the DataCompressor module using a Virtex-4

```
Minimum period: 12.974ns (Maximum Frequency: 77.076MHz)
Minimum input arrival time before clock: 12.751ns
Maximum output required time after clock: 6.305ns
Maximum combinational path delay: 5.571ns

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================
Timing constraint: Default period analysis for Clock 'Clock40'
Clock period: 12.974ns (frequency: 77.076MHz)
Total number of paths / destination ports: 590739 / 4057
---------------------------------------------------------

=========================================================
Timing constraint: Default period analysis for Clock 'Clock80'
Clock period: 10.409ns (frequency: 96.068MHz)
Total number of paths / destination ports: 645726 / 191
---------------------------------------------------------
```

(b) Timing analysis results for the 1Q version of the DataCompressor module using a Virtex-2P

Figure 6.13.: Timing analysis results for the DataCompressor module

the DataCompressor is programmed via the bidirectional Data bus with the Huffman codebook and the reference vectors data pre-initialized in RAM memories inside the FPGA. The codebook and reference vector elements are pre-initialized in RAM memories using initialization files during the programming phase of the FPGA. The button SW_C starts the data compression by setting the $data\_ready$ signal high. The DataCompressor starts reading the test pattern data from a RAM memory, which represents a RCU internal buffer. The test pattern data of 4000 waveforms (limited by the available RAM space) are pre-initialized in two RAM memories of the FPGA during the programming phase. The output data coming from the SOB_Din output bus of the DataCompressor are stored in a RAM memory each time the output signal SOB_we is high. This corresponds to the saving of the output data in the SIU output buffer of the RCU. If all test pattern data in one RAM memory are processed and compressed the pop signal goes high for one $Clock40$ cycle and the DataCompressor starts reading the next (second) RAM memory. After also the data in the second RAM are processed and compressed the data in the output memory are automatically sent via the UART interface to the serial input port (e.g. COM1) of the PC. A provisional trailer word is added to indicate the end of the words in the output memory and to ensure that all output data are received at the PC.

The received data can be visualized with a terminal program on the PC as shown in the screenshot in figure 6.14.

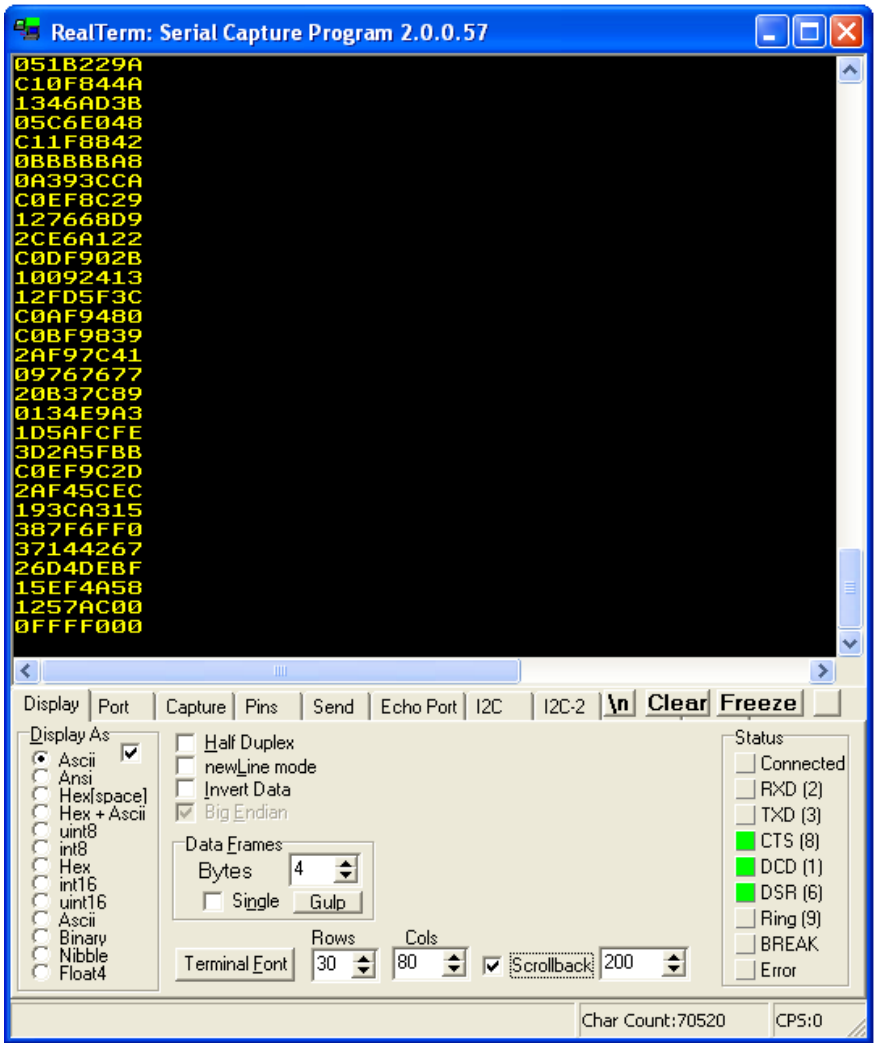


Figure 6.14.: Terminal program used to display the output of the DataCompressor sent via the UART to the PC

For the first testing of the interface, the terminal program is useful. To investigate the correctness of all output words from the DataCompressor send to the PC, the terminal program is not suitable and therefore a small C-program is written that reads the data received at the COM port and dumps them in a text file. The text file can be imported in Matlab and the Decompressor model can be used to decode and reconstruct the original test pattern data. The text file can also be imported in Excel to check if the output words are equal to the words generated by the simulation of the Verilog model. The tests showed that the output words produced by the hardware implementation of the DataCompressor in the Virtex-4 match exactly the simulation results. The output bus can easily be redirected inside the DataCompressor module to send out intermediate data from the Normalizer module, VectorQuantizer module or HuffmanCoder module, which allows checking the individual functionality of the different modules in hardware and performing a stepwise verification. The correct initialization of the LUT memories and the other memories inside the DataCompressor are checked by using the bidirectional rcu_prog_data bus, which allows to read-back the data from the selected memories inside the DataCompressor and to store these data as well in the output RAM memory from where it is sent to the PC. The DataCompression2ClockExt_Top module controls the execution of this read-back process. After all simulation tests and hardware tests are completed successfully for the DataCompressor module a first investigation of a possible implementation of the module in a future ASIC project is presented in the following.

6.4. Analysis for an Implementation in an ASIC

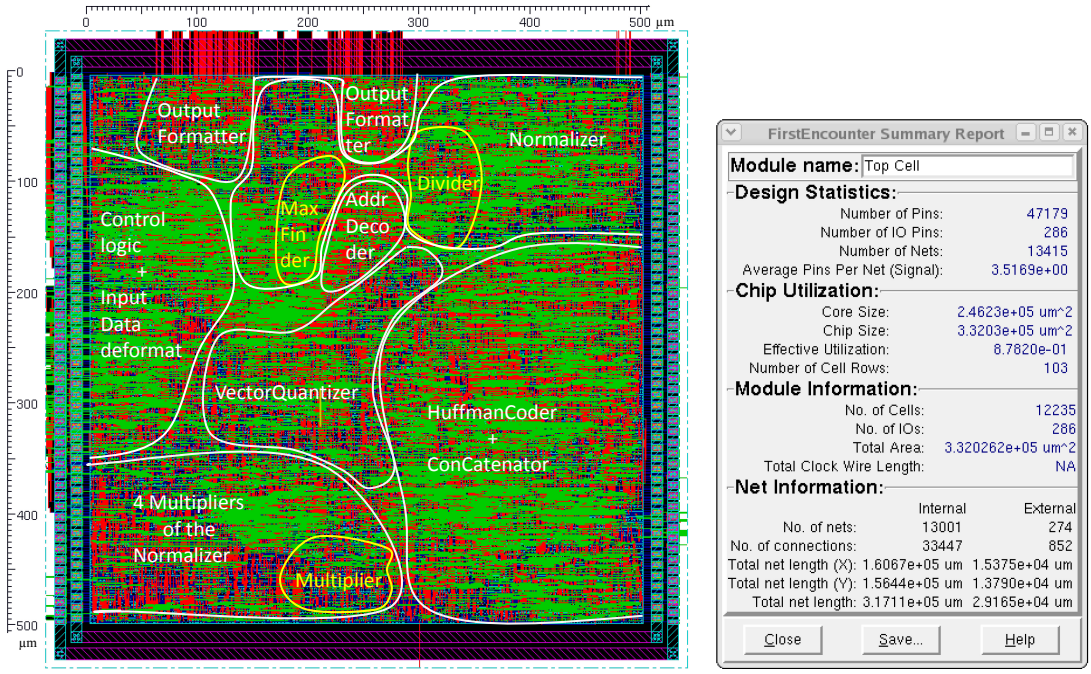
The DataCompressor development is included in a project which targets the development of a new mixed signal ASIC for the front-end electronics of the TPC in the ALICE experiment (S-ALTRO). In order to investigate the requirements for an implementation of the developed DataCompressor module in such a deep-submicron front-end chip the RTL Verilog model is synthesized (RTL Compiler). A Place&Route is performed using the Cadence Encounter tool version 8.1.

An Intellectual Property (IP) block for the DataCompressor logic is generated without including RAM memories. The used technology for the IP block is the IBM 130 nm 8RF-DM CMOS technology with DM metal stack and Process Design Kit (PDK) version 1.6 [54]. This technology is used as well for a first prototype development of the new TPC front-end ASIC S-ALTRO. The created layout for the DataCompressor IP block is shown in figure 6.15(a) and some parts of the logic are highlighted. The summary of the area requirements for the DataCompressor is shown in figure 6.15(b).

The required number of logic gates (cells) for the DataCompressor results in 12 235 gates. The estimated area of the IP block (core size) is around 496 µm×496 µm. The required memories that have to be added in a future implementation of the DataCompressor inside the ASIC are summarized in table 6.2.

The memory requirements can be reduced by almost a factor of two if their clocking is changed to run with the 80 MHz clock instead of the actually used 40 MHz clock. The 40 MHz clock is used in the actual implementation because the memories in the RCU work as well at this frequency. In addition, an optimization of some memory sizes (buffer sizes) can be performed for specific target applications.

The required logic can also be reduced if for example the input bandwidth is reduced to 10 bit per 40 MHz which is possible in a direct implementation in the front-end AISC by taking the digital data directly after the digital processor and before the formatting process. The reduction of the input bandwidth requires less parallelism in the DataCompressor. For example, the actually four multipliers in the Normalizer working in parallel can be reduced to one. The disadvantage of the reduction of input bandwidth is an increase in latency and overall compression time.



(a) IP-block Layout (b) Design Summary

Figure 6.15.: DataCompressor IP block Layout and Design Summary

Table 6.2.: List of memories and their sizes in the DataCompressor

Memory Name	Word Length [bit]	Depth [words]	Total space [bit]
TimeStampMem	10	64	640
PulseLengthMem	10	64	640
PulseLengthMem2	10	64	640
MaxValueMem	10	64	640
InputBuffer1	10	64	640
InputBuffer2	10	64	640
InputBuffer3	10	64	640
InputBuffer4	10	64	640
MaxIndexMem	5	32	160
NFactorMem	10	32	320
RVmem11	10	64	640
RVmem12	10	64	640
RVmem21(only 4Q version)	10	64	640
RVmem22(only 4Q version)	10	64	640
RVmem31(only 4Q version)	10	64	640
RVmem32(only 4Q version)	10	64	640
RVmem41(only 4Q version)	10	64	640
RVmem42(only 4Q version)	10	64	640
Huffmem LUT1	22	1024	22528
Huffmem LUT2	22	1024	22528
HuffBufferMem	30	128	3840
LongShortPulseMem	30	32	960
Total Memory requirements 1Q version:			56736 (6.9 kB)
Total Memory requirements 4Q version:			60576 (7.4 kB)

7. Conclusion and possible fields of application

In this doctoral thesis, a data compression method is developed with the aim to implement it in the front-end electronics of particle detectors used in high energy physics applications. A new developed lossless compression algorithm is presented and optimized for digital detector data that have been obtained from the TPC in the ALICE experiment. This algorithm is than implemented in hardware.

An introduction to the value of a implementation of a data compression in particle detector front-end electronics is given in chapter 1. The question "Why data compression in front-end electronics is a useful tool in particle detectors?" is answered first. The ALICE TPC detector is introduced that is used as an example application for the implementation of the data compression in the front-end electronics throughout this thesis. Then the two important measures of evaluating and comparing the efficiency of compression methods is explained.

First investigations on the properties and shape of digitized detector signals appearing in the front-end electronic is given in chapter 2. Different kinds of particle detectors are investigated and their front-end electronics and signals are present. The most common digitized front-end signal shape of these detectors, which is compressed is shown at the end of this chapter.

A discussion and evaluation of the most known compression methods are given in chapter 3. The different lossless and lossy compression methods are explained and an evaluation for their suitability of compressing the underlying detector signal data as well as their suitability for a real-time hardware implementation are given.

The best suited compression methods are then analyzed and compared carefully in chapter 4. The different lossless and lossy compression methods are modeled and used to compress a example set of measured data from the ALICE TPC to understand their efficiency for them. Their compression efficiency has been compared and an estimation of their complexity is given in order to find the best-suited compression algorithm for a hardware implementation. A new developed compression method with the name "lossless vector quantization" is chosen for the hardware implementation.

In chapter 5 the realization of the lossless vector quantization method in a FPGA based system is given. The compression method is modeled in the hardware description language Verilog and a detailed description of the different parts of this implementation is presented. The critical aspects and solutions for this hardware implementation are pointed out. The hardware implementation is optimized regarding the requirements of a possible implementation in the RCU FPGA of the ALICE TPC front-end electronics.

The results obtained from simulations and hardware tests of the implementations are discussed in chapter 6. The developed hardware solution is tested with the same set of measured data from the ALICE TPC as already used for the analysis and comparison of the different compression methods. Firstly, the simulation results of the Verilog code are presented and a proof of the functionality of the implementation on the TPC data is given. Then the hardware tests of the implementation are shown which have been carried out using a FPGA development board containing a Virtex-4 chip. At the end of this chapter a first investigation of the requirements for a hardware implementation in an ASIC chip is given.

To conclude the thesis a summary of the obtained research results is given in the following including a detailed description of the authors contribution to this research. At the end a short

outlook of possible fields of applications for the developed compression unit are presented.

7.1. Research Summary

The research carried out in this document comprised the investigation of the commonly known compression methods regarding their usability and efficiency for compressing digitized detector data, which are already reduced strongly by a zero suppression. The underlying detector data for this research have been obtained by measurements with the ALICE TPC. The data are zero suppressed and consist of samples belonging to waveforms with a known shape introduced by the analogue amplifiers and/or shapers in the detector front-end. It could be shown that the signals from the ALICE TPC similarly appear also in various other types of particle detector concepts allowing to use the development data compression unit in a wide range of particle detector experiments. A developed lossless compression algorithm based on a modified lossy vector quantization has been selected as the best suited method for the compression of these detector data and for a hardware implementation. The lossless vector quantization algorithm offers the highest compression efficiency for the corresponding detector data among all lossless methods. With this method, the zero suppressed data could be further reduced by a factor of 2. This allows to relax significantly the requirements for data transfer and data storage earliest in the data chain already inside the front-end electronics of particle detectors.

Because of the low distortion introduced on the important information of the detector data by this compression method, it is preferred instead of using one of the as well analyzed lossy compression methods. The lossless vector quantization produces a negligible distortion of the data in the hardware implementation caused by limited precisions of some required arithmetic logic for a normalization of the input waveforms. The lossy compression methods on the other hand produce a higher distortion by systematic data loss in order to obtained a higher compression ratio.

The implemented algorithm consists of a Normalizer, Vector Quantizer and Huffman encoder. The Normalizer normalizes the incoming waveforms to a predefined amplitude, which is equal to the maximum element of the reference vectors saved in an on-chip memory. The Vector Quantizer then compares the normalized samples of the input waveforms that form an input vector with different reference vectors and searches for the best matching reference vector. The differences (Deltas) between the best matching reference vector and the input vector are calculated and then Huffman encoded.

The data that are sent out of the detector front-end consists of the waveform lengths, the time stamps (both deriving from the zero suppression), the normalization factors (which in the actual implementation are represented by the maximum sample values of the input waveforms), the indices of the chosen reference vectors and the Huffman encoded Deltas. The decoder reconstructs Delta values from the received Huffman words and restores the normalized sample values by adding the Deltas to the elements of the corresponding reference vector selected by the received index. Then the decoder recalculates the normalization factor using the received maximum sample value to re-normalize the original sample values of the input waveform out of the normalized samples. Both the Data Compressor and the Decompressor have to be loaded prior to operation with the same reference vector set, normalization value and Huffman codebook which are obtained by analyzing representative data from previous measurements.

The compression algorithm has been tested first using a Matlab model and then the implementation of the algorithm has been evaluated using a simulator and signal generator for HDL codes. Afterwards, the developed implementation has been tested for its functionality and real-time behavior in hardware by using a FPGA development board. The results obtained from the implementation are very close to the results from the Matlab model.

The hardware implementation showed its ability for a real-time application by consecutively

compressing 10 000 waveforms without causing any wait cycle running at 40 MHz. The implementation has been realized in a pipeline structure which causes a latency of around 30 clock cycles of 40 MHz. Since the algorithm executes on entire waveforms the latency is not constant and depends on the length of the input waveforms, which after the zero suppression are composed by varying number of samples. The actual RCU implementation without data compression on the other hand requires every third clock cycle one wait cycle stopping reading data from the MEBs to cope with the different input (40 bit) to output (30 bit) word length (input and output frequency is 40 MHz). The implementation of the data compression allows the continuous reading of the MEBs without introducing wait cycles.

Two versions of the data compression block are realized, one using only one reference vector and the second using four reference vectors. To determine the minimum required resources of the data compression block in hardware the version with one reference vector is implemented in the Virtex-4 FPGA of the development board. This 1Q version requires around 24% of the slices and 25% of the block RAMs. The version with 4 reference vectors is implemented as well in the Virtex-4 to investigate the best achievable compression performance for the used data and to prove also the functionality of this version. The timing analysis showed that the used clock frequencies of 40 MHz main clock and 80 MHz fast clock are perfectly suited for both versions and can be used as well for an implementation in the smaller Virtex-2P FPGA of the current RCU installed in the ALICE TPC.

In addition to the resource requirements of the data compression implementation in FPGAs, also the area requirements of an implementation of the data compression unit in a full custom ASIC are presented. The layout of the data compression IP block has been realized in the IBM 130 nm technology with DM metal stack. It shows that the 1Q version would fit in around 500 µm $\times$ 500 µm area and requires around 12 200 logic gates (without counting the required memory of around 7 kB)

This proves that an implementation of the developed compression method in future particle detector front-ends is possible and can obtain a good data reduction very close to the detector pads. The data compression in the front-end offers the possibility to relax the limits of data transfer bandwidth and data storage for future experiments, by still maintaining the required data integrity and measurement accuracy.

7.1.1. Main contributions of the author

The author claims to have made the following contributions to the described research activity:

- I could prove that a further data reduction of zero suppressed digitized detector data can be obtained by implementing a data compression unit in the front-end electronics of detectors, without introducing relevant distortions on the measurements. A data reduction of 50% has been achieved by this implementation. To the best of my knowledge this has been not yet achieved inside the front-end electronic of detectors on zero suppressed data. This helps to relax significantly the data transfer and storage requirements.
- At the beginning of my research activities I have studied different types of particle detectors to investigate the form, parameters and information content of the digitized detector data produced in the front-end electronics. For most detector types, including the TPC, I identified a signal form corresponding to a digitized semi-Gaussian waveform that represented the basic data for which I developed my data compression implementation.
- During my research activities I have compared different compression methods to find and develop the best-suited method for compressing the corresponding detector data. Comparing the compression efficiencies of the different methods, I could show that the best-suited method is the new developed lossless vector quantization composed by the normalization, delta calculation and Huffman coding.

- I have optimized the concept of lossless vector quantization for the compression of the detector data from the ALICE TPC. The key point of this method is the normalization of the input waveforms before performing the vector quantization in order to reduce the required number of reference vectors and therefore the memory space in the hardware to store this reference vectors.
- I have realized a hardware implementation block performing this lossless vector quantization using Verilog and implementing it in a FPGA development board. I tested the data compression in hardware and I could prove its functionality and real-time performance. The implementation achieved the expected compression ratio of a factor of 2 obtained by the Matlab model. In addition, a fasten-up of the readout has been achieved compared to the actually implemented RCU logic in the ALICE TPC.
- During the hardware realization of the data compression unit I implemented an integer divider that is used to normalize the input waveforms. This divider has to perform the calculation in a fast way in order to keep the buffers for the input samples small. To run the divider with the 80 MHz input clock I have realized the divider in a pipelined structure with 3 stages.
- To send the information about the normalization factor resulting from the division I decided to use the original value of the maximum sample instead of the calculated normalization factor itself. This requires that the decoder has to recalculate the normalization factor using the received maximum value. On the other hand, this keeps the number of used bits for transmitting the information independent from the used precision of the implemented diver and multipliers. Therefore, more flexibility is achieved for future applications in terms of optimizing arithmetic precision versus acceptable distortion and area/recourse requirements without having to change the output data format.
- Another important aspect of the data compression implementation is the alignment of the input vectors with the reference vectors and the transfer of this alignment information to the decoder. Therefore, I have used the position of the maximum sample of the input waveforms to align them with the reference vectors. I have embedded the information of the position of the maximum sample in the values of the resulting Deltas. This concept allows transmitting the alignment information without adding additional bits to the output data, with the expense of introducing some small additional distortions to the detector data. I could show that these small distortions have negligible impact on the important information of the data (amplitudes and time stamps).
- For the transmission of the output data of the data compressor I defined a new output format based on the current RCU format. The difference to the actual RCU output format consists in the scrambling of the compressed bitstream words with the waveform headers in order to optimize the output package for the unoccupied bus time between the sporadic obtained compressed Huffman words. To distinguish the different components of the output package the two MSBs are used which already exist in the current RCU format.
- To investigate the possibility of implementing the data compression unit also in further ASIC projects I created an IP block in 130 nm CMOS technology and extracted the area requirements and the number of required logic gates.
- During my investigations of the different compression methods I could show that the best lossy compression method is the model based method for the underlying detector data. I have created a model based on the idea of the parameter extraction published in [44] and I could show that a good extraction of amplitudes and time stamps from the ALICE TPC waveforms can be performed with this model by obtaining in most of the cases small distortions. A hardware realization of this parameter extraction can be performed by implementing tow FIR filter. A concept of proving the quality of the parameter extraction in the hardware has still to be developed in order to guarantee the required measurement accuracy.

- To improve the efficiency of the lossy compression methods based on DCT and DWT I combined them with the concept of using a reference vector and delta calculation. The resulting coefficient vectors are compared to a reference vector and then the differences (deltas) are quantized and Huffman encoded. This improved the performance of the transform methods.

7.2. Outlook

The designed data compression algorithm named lossless vector quantization showed its functionality in a hardware implementation and its ability to compress the detector data by around 50%. There are some future projects listed in the following where the data compression IP block could be implemented to facilitate the data transfer and storage in new detectors with large amounts of readout channels. The smaller the front-end electronic can be designed by using for example new full custom ASICs and new FPGA devices (with much more resources) the more readout pads can be used in the limited space of the detector area. This could allow reducing the size of the readout pads providing a higher resolution of the particle tracking which on the other hand produce more readout channels and more data that have to be handled. The data compression block could be a good option to handle these data.

7.2.1. SALTRO

An actual ongoing project is aiming to design a new front-end chip for the ALICE TPC and other detectors, which combines the analogue font-end electronics of the PASA and the digital front-end electronics of the ALTRO chip. A first prototype of this mixed signal ASIC called S-ALTRO has been produced and is currently under test. As well the data compression IP block could be implemented in this new front-end ASIC in order to move the data compression even closer to the detector pads. The plans for the S-ALTRO foresee also the investigation of a data link between neighboring chips to allow an inter-chip communication in order to increase the performance of the zero suppression by a 3D zero suppression [55]. This inter-chip communication could also be used to increase the compression performance of the implemented data compression method. Neighboring channels can measure contemporary singles created by the same cluster of induced charges produced from the avalanche process in the amplification region of the TPC end caps (MWPC). The signals in neighboring channels from the same cluster are quite similar with a predictable variation in amplitude according to a Gaussian distribution. That means that the digitized signal from one channel could be used as a reference vector for compressing the signal of a neighboring channel. This would maybe increase the compression efficiency and reduce the complexity of the lossless vector quantization.

The layout of the data compression block, which is presented in chapter 6 is designed in the same technology as the actual prototype of the mixed-signal ASIC S-ALTRO. This would allow an easy implementation in a next generation of the S-ALTRO. The area of the data compression IP can be reduced further in the new ASIC by using less parallelization of the data paths changing the input data format at the data compressor.

7.2.2. ALICE TPC upgrade

The resource requirements of the version with one reference vector showed that an implementation of the data compression block in the Virtex-2P FPGA of the RCU actually installed in the TPC front-end inside the ALCIE experiment requires half of the resources. A first investigation of a combination of the data compression block with a reduced version of the actual RCU logic showed that a slight exceed of the available recourses is encountered for the Virtex-2P as shown in figure 7.1. A further effort in reducing the complexity of the required RCU logic and the data

compressor logic could result in a possible implementation of the data compressor inside the actual RCU unit inside the ALICE TPC. This would allow testing the data compressor already inside the real environment of TPC front-end electronics in the ALICE experiment.

An upgrade of the LHC and the experiments is actually scheduled for the year 2015. In this period, also the ALICE experiment will be upgraded and possibly some parts of the TPC readout electronics can be exchanged [56].

If in the planned upgrade for the ALICE TPC also the RCU FPGA can be exchanged to a newer, larger FPGA like the Virtex-4, a permanent implementation of the data compression block in the ALICE TPC is possible by maintaining the full functionality of the RCU.

```
Selected Device : 2vp7ff672-7

 Number of Slices:                    5851  out of   4928   118% (*)
 Number of Slice Flip Flops:          4108  out of   9856    41%
 Number of 4 input LUTs:              9947  out of   9856   100% (*)
    Number used as logic:             9364
    Number used as Shift registers:     43
    Number used as RAMs:               540
 Number of IOs:                        254
 Number of bonded IOBs:                 93  out of    396    23%
    IOB Flip Flops:                     41
 Number of TBUFs:                      220  out of   2464     8%
 Number of BRAMs:                       44  out of     44   100%
 Number of MULT18X18s:                   4  out of     44     9%
 Number of GCLKs:                        7  out of     16    43%
 Number of DCMs:                         3  out of      4    75%

WARNING:Xst:1336 -  (*) More than 100% of Device resources are used
```

Figure 7.1.: The synthesis report device utilization summary for the DataCompressor combined with the necessary RCU logic in the Virtex-2P FPGA installed in the RCU of the TPC.

7.2.3. CLIC

First studies for a new accelerator project called “CLIC” are triggered by CERN, which could be the follower of the LHC. The concept of a Compact LIner Collider (CLIC) is based on a 48 km linear accelerator probably be built somewhere in the neighborhood of CERN. The collider will provide two accelerated beams, one of electrons and one of positrons, which then collide in the middle of the structure where the detectors are placed (interaction point). To accelerate the electrons and positrons the CLIC uses a new concept of two beams, a drive beam that injects energy in a main beam, which then is brought to collision [57]. The concept is shown in figure 7.2.

The CLIC is planned to provide collisions of elementary particles with energies up to 5 TeV (nominal 3 TeV). The previously most powerful electron-positron accelerator was the LEP with a collision energy of 209 GeV build at CERN (in the tunnel where now the LHC is placed).

The detector experiment placed at the interaction point will contain a variety of different kinds of detectors. The detector concept for the CLIC is carried out together with the collaboration for the ILC project under the Linear Collider Detector (LCD) collaboration. In tree of the four detector concepts a TPC is included, which would allow the use of the actual data compression block in the front-end electronics. The detector concepts are presented within the ILC project description in the following.

7.2.4. ILC

A second research study is initiated for a future linear collider called “ILC”. The International Linear Collider (ILC) project is based on two particle beams, one of electrons and the other one

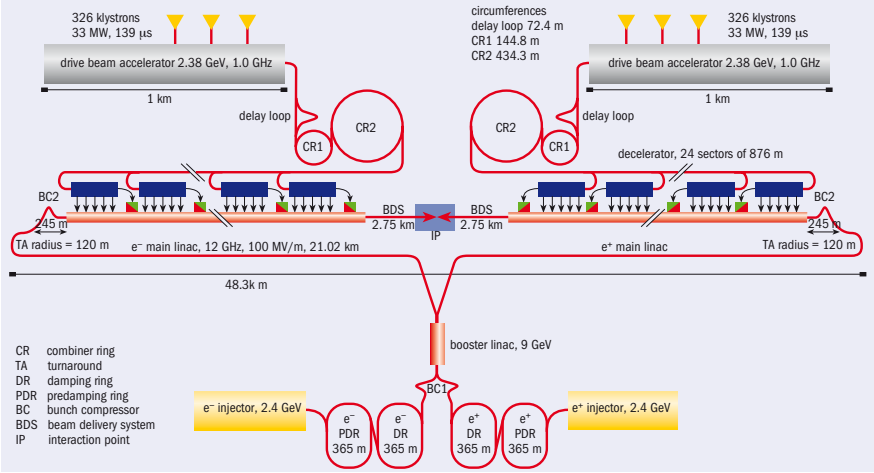


Figure 7.2.: Compact Linear Collider Concept. New accelerator project intended to be the follower of the LHC at CERN. Nominal collision energy will be 3 TeV. [57]

of positrons brought to collision in the middle of the structure as shown in figure 7.3. The ILC will be around 31 km long and produce a collision energy of 500 GeV with a possible upgrade to 1 TeV. The accelerator uses well-known technologies and is shorter as the CLIC but provides also less collision energy. The results obtained by the LHC will show which collision energy is needed for the new targeted particle physics research.

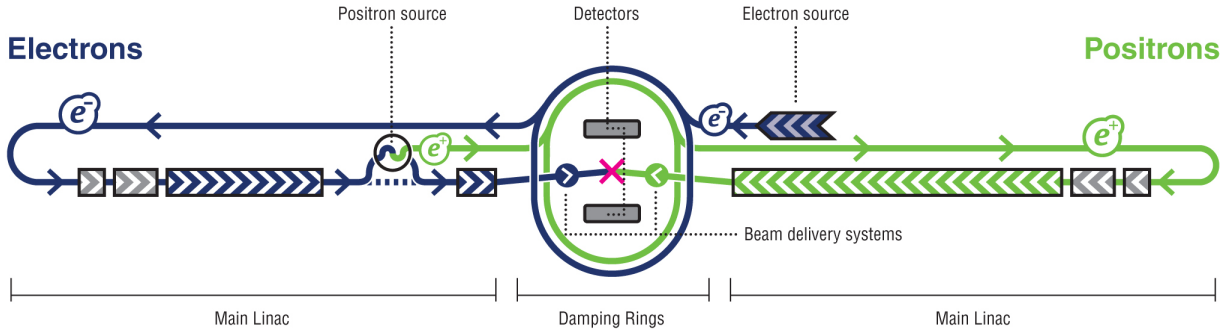


Figure 7.3.: The international linear collider project.

For the detectors of the ILC (or as well for CLIC) there are four concepts developed and described in the reference design report [58].

The first is the **SiD** detector concept. This detector experiment is composed of only silicon trackers and calorimeters. There are no gaseous detectors such as TPCs in this concept. This is probably the most expensive concept because of the high costs of the calorimeters. They should be designed as compact as possible to save costs.

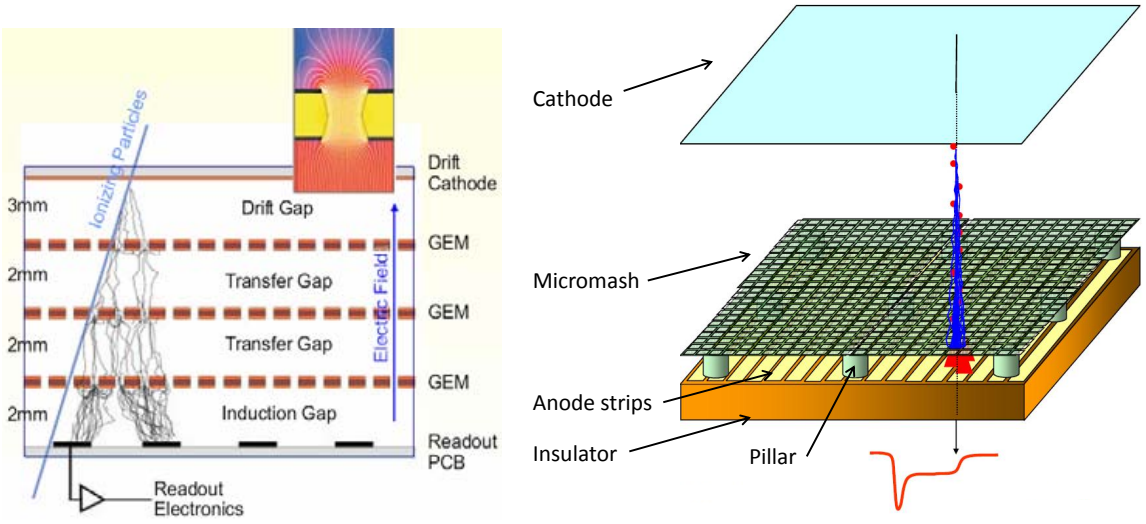
The second concept is the LDC (Large Detector Concept). This detector concept has as its central component a TPC. A vertex detector is between the beam pipe and the TPC and on the outside of the TPC, there are electromagnetic and hadronic calorimeters placed. The TPC has a inner radius of 32 cm and an outer radius of 168 cm with a length of 273 cm. A 4 T magnet produces the required magnetic field.

The third proposal is the **GLD** (Global Large Detector) concept. This concept has as well a TPC as main tracker component. The TPC is even larger as the one of LDC with an inner radius of 40 cm and an outer radius of 200 cm. A vertex detector is paced between the beam pipe and the TPC. The TPC is surrounded of electromagnetic, hadronic and forward calorimeters. A 3 T magnet produces the surrounding magnetic field.

The **4th** concept is quite different from the other tree. It uses compensated dual readout calorimeters instead of the particle flow calorimeters used in the other ones. As well, this concept has a TPC as central component. The TPC probably would have an inner radius of 20 cm and an outer radius of 140 cm. The magnetic field is planned to be 3.5 T.

Since a TPC is included in three of the four detector concepts, it is possible that the data compression IP block optimized for TPC data could provide its usability into the new detector experiment of the ILC or CLIC. However, also in ECALs and other kinds of detectors the data compression block could be implemented with an optimization rework for the respective data properties.

The new TPC studies investigate new methods to amplify the ionized charge in the detector drift volume. Two possible alternatives to the MWPC used actually in the ALCIE TPC are under investigation named GEM (Gas Electron Multiplier) and MICROMEGAS (MICRO MEsh GAseous Structure). These new concepts are shown in figure 7.4. They are promising in terms of increased position resolution for the new TPCs to get a better momentum measurement of the primary particles.



(a) Schematic structure of a triple GEM [26]

(b) Schematic structure of the Micromegas [59]

Figure 7.4.: Structural concept of GEM and Micromegas

The advantage of the new concepts in comparison to the MWPC is that the signals in the detector pads get induced mainly by electrons, which give a fast signal and prevent almost the existents of long ion tails in the signals caused by induction from ions crated in the avalanche effects [60]. Ion tails cause pile-up problems in consecutive signals from the same pad. A prevention of these pile-up effects is also advantageous for the data compression with lossless vector quantization since the strange waveform shapes from pile-ups increase the Deltas and therefore reduce the compression efficiency.

7.2.5. PANDA

Another new experiment is planned at DESY in Darmstadt called PANDA (anit-Proton ANnihilation at DArmstadt). This detector experiment will be installed in the FAIR accelerator complex, which accelerates anti-protons and ions. The detector concept of PANDA includes as well GEMs and ECALs in which the digital compression block could be implemented in the front-end electronics. A comparison of some important parameters between the discussed future experiments is given in table 7.1.

An important aspect for the data compression block is the cluster width, which is affected by the signal duration and therefore determines the number of samples per waveform. The cluster

Table 7.1.: Comparison of some important parameters between the actual ALCIE TPC and future detector experiments containing GEMs or MICROMEGAS. [61]

	ILC (ILD)	CLIC (ILD)	FAIR (PANDA)	LHC (ALICE)
drift volume (m^3)	40 (L = 4.6m, Φ = 3.6m)		0.7 (L = 82cm, Φ = 150cm)	90
Acceptance	(up to cosθ = 0.98) × 2π		(cosθ = 0.98) ×2π	(cosθ =0.94) ×2π
drift field (V/cm)	200		400	400
magnetic field (T)	3.5		2	0.5
drift velocity (cm/μs)	5-10		2.7	2.6
max drift time	46 μs		56 μs	92 μs
longit. cluster width (ns)	100 (LOI) - 400 (dip angle)		60 - ?	124 – 400
endcap readout	MPGD (GEM, MicroMegas)		GEM	MWPC
gain (mult. region)	~10^3		~ 2-5×10^3	~10^4
pad size (mm^2)	1×5		2×2	4×7.5
nr. channels	2×10^6		0.8×10^5	5.7 × 10^5
dynamic range	8-10 bits		8-bit	10 bit
beam pulse duration (ns)	10^6	156 - 177	continuous	continuous
beam repetition rate (Hz)	5	50	-	-
trigger/event rate (Hz)	5	50	20M	1k
occupancy	<0.2%	<2%	2% - 8%	<10%
material budget	~0.15X_0 endcaps in z		~0.15X_0 endcaps in z	not relevant
trigger	beam pulse		none	event (collision)

with for ILC and CLIC detector experiments is predicted similar to the ALICE experiment. The dynamic range stays probably the same and the number of readout channels is as well large for the future detectors listed in the table. Therefore, the implementation of the data compression block would give advantages in handling the readout data. An implementation in the new SALTRO front-end ASIC would provide an easy, efficient and compact system for the detector pad readout. The time will show in which form data compression will find the way into future particle detector electronics.

List of Publications

Articles in international journals

- **C. Patauner**, A. Marchioro, S. Bonacini, A. U. Rehman and W. Pribyl, "A Lossless Data Compression System for a Real-Time Application in HEP Data Acquisition", *IEEE Transaction on Nuclear Science*, vol. 42, no. 17, pp. 1002–1003, June 2011.

Articles in international conference proceedings

- **C. Patauner**, A. Marchioro and W. Pribyl, "A Lossless Data Compression Method for an Application in High Energy Physics", *Proceedings of the 2009 IEEE PRIME Conference*, pp. 806–809, July 2009.

- **C. Patauner**, A. Marchioro, S. Bonacini, A. U. Rehman and W. Pribyl, "A Lossless Data Compression System for a Real-Time Application in HEP Data Acquisition ", *Proceedings of the 2010 IEEE Real-Time Conference*, pp. 162–165, June 2010.

Bibliography

[1] "The ALICE Time Projection Chamber (TPC)," CERN, Geneva, Switzerland, 2009. [Online]. Available: http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2_TPC.html

[2] "Time Projection Chamber ALICE - Technical Design Report 7," CERN, Geneva, Switzerland, Tech. Rep. LHCC 2000-001, Jan 2000.

[3] J. Alme *et al.*, "The ALICE TPC, a large 3-dimensional tracking device with fast readout for ultra-high multiplicity events," Tech. Rep. arXiv:1001.1950, Jan 2010.

[4] A. Fayyazuddin and M. Riazuddin, *A modern introduction to particle physics*, 2nd ed. Singapore: World Scientific, 2000.

[5] C. P. Office, "LHC research programme gets underway," *CERN Press Release*, no. PR07.10, Mar. 2010. [Online]. Available: http://public.web.cern.ch/press/pressreleases/Releases2010/PR07.10E.html

[6] C. Lefevre, "LHC: the guide (English version)," Feb 2009. [Online]. Available: http://cdsweb.cern.ch/record/1165534/files/CERN-Brochure-2009-003-Eng.pdf

[7] "CERN in a nutshell," CERN, Geneva, Switzerland, 2010. [Online]. Available: http://public.web.cern.ch/public/en/About/About-en.html

[8] "Origins: CERN: People," Exploratorium, San Francisco, CA, 2009. [Online]. Available: http://www.exploratorium.edu/origins/cern/people/index.html

[9] H. Spieler, *Semiconductor Detector Systems*. Oxford University Press, USA, 2005.

[10] "CMS Tracker: Pixels," CERN, Geneva, Switzerland, 2011. [Online]. Available: http://cms.web.cern.ch/cms/Detector/Tracker/Pixels.html

[11] M. Despeisse, G. Anelli, and C. Gontrand, "Etude et caractérisation d'un capteur en silicium amorphe hydrogéné déposé sur circuit intégré pour la détection de particules et de rayonnements," Ph.D. dissertation, INSA Lyon, Lyon, France, 2006.

[12] G. Aad *et al.*, "The ATLAS experiment at the CERN Large Hadron Collider," *Journal of Instrumentation*, vol. 3, 2008.

[13] "Technical Design Report of the Inner Tracking System (ITS)," CERN, Geneva, Switzerland, Tech. Rep. LHCC 99-12, Jun 1999.

[14] R. Dinapoli *et al.*, "A front-end for silicon pixel detectors in ALICE and LHCb," *Nucl. Instrum. Methods Phys. Res., A*, vol. 461, no. 1-3, pp. 492–495, 2001.

[15] "CMS Silicon Strip Detector: Principle of Operation," INFN, Florence, Italy, 2001, image [GIF]. [Online]. Available: http://hep.fi.infn.it/CMS/sensors/Silicon_Detector.gif

[16] T. Shin'ichiro Takeda, S. Watanabe, H. Tajima, T. Tanaka, K. Nakazawa, and Y. Fukazawa, "Double-sided silicon strip detector for x-ray imaging," *SPIE Newsroom*, 2008, iSAS/JAXA. [Online]. Available: http://www.spie.org/x20060.xml

[17] C. Hu-Guo *et al.*, "The HAL25 Front-end chip for the ALICE Silicon Strip Detectors." Stockholm, Sweden: 7th Workshop on Electronics for LHC Experiments, Sep. 2001, pp. 76–80.

[18] K. Aamodt *et al.*, "The ALICE experiment at the CERN LHC," *Journal of Instrumentation*, 2008.

[19] D. Falchieri, M. Basile, E. Gandolfi, and G. Venturi, "Hardware Implementation of Data Compression Algorithms in the ALICE Experiment," Ph.D. dissertation, Università degli studi di Bologna, Bologna, 2001.

[20] A. Rivetti *et al.*, "The front-end system of the silicon drift detectors of ALICE," *Nucl. Instrum. Methods Phys. Res., A*, vol. 541, no. 1-2, pp. 267–273, 2005.

[21] "CMS: Electromagnetic calorimeter (ECAL)," CERN, Geneva, Switzerland, 2011. [Online]. Available: http://cms.web.cern.ch/cms/Detector/ECAL/

[22] S. Chatrchyan *et al.*, "The CMS experiment at the CERN LHC," *Journal of Instrumentation*, vol. 3, no. 8, 2008.

[23] "CMS: Hadron Calorimeter (HCAL)," CERN, Geneva, Switzerland, 2011. [Online]. Available: http://cms.web.cern.ch/cms/Detector/HCAL/

[24] K. Anderson, T. Del Prete, E. Fullana, J. Huston, C. Roda, and R. Stanek, "TileCal: The Hadronic Section of the Central ATLAS Calorimeter," *At the leading edge: the ATLAS and CMS LHC experiments*, pp. 233–258, 2009.

[25] O. Schäfer, "Working Principle of a TPC," DESY: LCTPC, 2009. [Online]. Available: http://www.lctpc.org/e8/e57671/

[26] G. Trampitsch, "Design and characterization of an analogue amplifier for the readout of micro-pattern gaseous detectors," Ph.D. dissertation, TU Graz, Graz, Austria, 2007.

[27] E. Cerron Zeballos *et al.*, "A new type of resistive plate chamber: the multigap RPC," *Nucl. Instrum. Methods Phys. Res., A*, vol. 374, no. 1, pp. 132–135, 1996.

[28] C. Posch, E. Hazen, and J. Oliver, *MDT-ASD, CMOS front-end for ATLAS MDT, rev. version 2.1*, Geneva, Sep 2007.

[29] G. Aielli *et al.*, "Test of ATLAS RPCs Front-End electronics," *Nucl. Instrum. Methods Phys. Res., A*, vol. 508, no. 1-2, pp. 189–193, 2003.

[30] F. Giannini, E. Limiti, G. Orengo, and R. Cardarelli, "An 8 channel GaAs IC front-end discriminator for RPC detectors," *Nucl. Instrum. Methods Phys. Res., A*, vol. 432, no. 2-3, pp. 440–449, 1999.

[31] H. Kano *et al.*, "Custom chips developed for the trigger/readout system of the ATLAS endcap muon chambers." Krakow, Poland: Sixth Workshop on Electronics for LHC Experiments, Sep. 2000.

[32] O. Sasaki and M. Yoshida, "ASD IC for the thin gap chambers in the LHC ATLAS Experiment," *IEEE Transactions on Nuclear Science VOL. 46*, Dec. 1999.

[33] D. Salomon, *Data compression: The Complete Reference*, 4th ed. London: Springer, 2007.

[34] K. Sayood, *Introduction to data compression*, 3rd ed., ser. Morgan Kaufmann Series in Multimedia Information and Systems. Burlington, MA: Elsevier, 2005.

[35] H. Nautsch, "Image and Audio Coding - Lecture notes (TSBK06)," Linköping University, Linköping, Sweden, 2010. [Online]. Available: http://www.bk.isy.liu.se/en/courses/tsbk06/material/

[36] H. Nautsch, "Data Compression - Lecture notes (TSBK08)," Linköping University, Linköping, Sweden, 2010. [Online]. Available: http://www.bk.isy.liu.se/en/courses/tsbk08/material.html

[37] A. Vestbo, "Pattern Recognition and Data Compression for the ALICE High Level Trigger," Ph.D. dissertation, Bergen Univ., Bergen, 2004.

[38] M. Ivanov, A. Nicolaucig, and A. Krechtchouk, "Data compression using correlations and stochastic processes in the ALICE Time Projection chamber." La Jolla, CA: 2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP 03), Mar. 2003, p. THLT002.

[39] A. Nicolaucig, M. Mattavelli, and S. Carrato, "Compression of TPC Data in the ALICE Experiment," *Nucl. Instrum. Methods Phys. Res., A*, vol. 487, no. CERN-ALI-2001-051. CERN-ALICE-PUB-2001-51. 3, pp. 542–556. 38 p, Oct 2001.

[40] J. Berger *et al.*, "TPC Data Compression," *Nucl. Instrum. Methods Phys. Res., A*, vol. 489, no. 1-3, pp. 406–421, 2002.

[41] W. Blum, W. Riegler, and L. Rolandi, *Particle Detection with Drift Chambers*. Springer, 2008, chapter 6.

[42] S. Rossegger, B. Schnizer, W. Riegler, and L. Betev, "Simulation and Calibration of the ALICE TPC including innovative Space Charge Calculations," PhD thesis, Graz, University of Technology, Graz, 2009, presented on 30 Oct 2009.

[43] A. Nicolaucig, M. Ivanov, and M. Mattavelli, "Lossy compression of TPC data and trajectory tracking efficiency for the ALICE experiment," *Nucl. Instrum. Methods Phys. Res., A*, vol. 500, pp. 412–420, 2003.

[44] V. Buzuloiu, M. Malciu, V. Popescu, and C. Vertan, "Optimal recovery of signal parameters from a few samples: one-and two-dimensional applications," *Optical Engineering*, vol. 35, p. 1576, 1996.

[45] K. D. Kammeyer and V. K., *MATLAB in der Nachrichtentechnik*. J. Schlembach Fachverlag, 2001.

[46] E. G. Gracia, "SALTRO Review meeting - Data Processor," CERN, May 2009. [Online]. Available: http://indico.cern.ch/getFile.py/access?contribId=5&resId=1&materialId=slides&confId=58839

[47] B. Mota and D. Mlynek, "Time-Domain Signal Processing Algorithms and their Implementation in the ALTRO chip for the ALICE TPC," Ph.D. dissertation, EPFL, Geneva, Switzerland, 2003.

[48] *ALICE TPC Readout Chip User Manual*, CERN, Geneva, Switzerland, Jun. 2002, draft 0.2. [Online]. Available: http://ep-ed-alice-tpc.web.cern.ch/ep%2Ded%2Dalice%2Dtpc/doc/ALTRO_CHIP/UserManual_draft_02.pdf

[49] A. U. Rehman, "Data Compressor Interface to Integrate in RCU," Sep. 2009, draft, Internal note.

[50] *RCU Firmware User Manual*, 2nd ed., CERN, Geneva, Switzerland, Aug. 2010. [Online]. Available: http://ep-ed-alice-tpc.web.cern.ch/ep-ed-alice-tpc/doc/RCU/DOC/RCUFirmwar2.1.pdf

[51] "Virtex-4 Ml401 Evaluation Platform," XILINX Inc., 2010. [Online]. Available: http://www.xilinx.com/products/devkits/HW-V4-ML401-UNI-G.htm

[52] M. Wilmott, "Advanced Digital Physical Implementation flow," IDESA, Belgium, 2008.

[53] "ML401/ML402/ML403 Schematics," XILINX Inc., Aug. 2004. [Online]. Available: http://www.xilinx.com/support/documentation/boards_and_kits/ml401_2_3_schematics.pdf

[54] *CMOS8RF (CMRF8SF) Design Manual*, IBM, Mar. 2009, v 1.6.0.0.

[55] P. Aspell *et al.*, "S-ALTRO Specification," April 2009, internal note.

[56] P. Giubellino, "ALICE upgrate plans," CERN, July 2010. [Online]. Available: http://indico.cern.ch/getFile.py/access?contribId=2&resId=1&materialId=slides&confId=99603

[57] H. Braun, J. P. Delahaye, A. De Roeck, and G. Geschonke, "CLIC here for the future," *CERN Courier*, vol. 48 issue 7, September 2008.

[58] T. Behnke *et al.*, "International Linear Collider Rference Design Report Volume 4: DETECTORS," ILC, Tech. Rep., 2007. [Online]. Available: http://www.linearcollider.org/about/Publications/Reference-Design-Report

[59] P. Colas, "Practical operation of Micromegas dectectors," CEA/Irfu, Saclay, February 2009. [Online]. Available: http://indico.cern.ch/conferenceDisplay.py?confId=52462

[60] "A Time Projection Chamber with Gas Electron Multipliers for the TESLA Detector," University Karlsruhe. [Online]. Available: http://www-ekp.physik.uni-karlsruhe.de/~ilctpc/GEM-TPC/gem-tpc.html

[61] L. Musa, "SALTRO Review meeting - Introduction," CERN, May 2009. [Online]. Available: http://indico.cern.ch/getFile.py/access?contribId=9&resId=0&materialId=slides&confId=58839

A. Code Tables

In the following the codebook tables used for the compression of the ALICE TPC test-data matrix with the data compression implementation are given. Three different Huffman codebooks are listed for the 1Q version, the 4Q version and the HuffmanOnly mode.

The entries in the codebook are given in the columns named Code in hexadecimal form. The input values of the Huffman coder corresponding to the codes are given in the columns Delta or Input. The hexadecimal values of the codes represent the binary words of 22 bit of the Huffman codeword length and the Huffman codeword. The five most significant bits of each code represent the codeword length the following bits are the Hoffman codeword filled up with 0 to the 22 bit.

For example, the code for the Delta value 0 of the 1Q version is 68000_h. This gives the binary number 0001101000000000000000_b. The five MSBs are 00011_b which gives a codeword length of 3 bit. The next three bit 010_b therefore represent the Huffman codeword for the Delta value 0. The remaining bits of value 0 are used to fill up the binary word to the memory word size of 22 bit. The Code for the Delta value 1 of the 1Q version is 64000_h representing the binary word 0001100100000000000000_b. The 5 MSBs give the same codeword length of 3 and the corresponding Huffman codeword for the Delta value 1 is 001_b. This shows that the codeword of the Delta values 0 and 1 have equal length which was used in the data compressor for the alignment concept of the input vectors with the reference vectors to obtain the best compression ratio. The problematic 0 Delta values for this alignment are changed to the value 1 which not increases the resulting encoded bitstream.

This property is also true for the 4Q version with the difference that the codewords for the Delta values 0 and 1 are 2 bit long. The maximum considered Delta values are ± 100 because the normalization value for the test-data is set to 100. The remaining codes for Deltas smaller than -100 and lager +100 are not shown in the tables because they have all a hexadecimal value of 000000_h. The codebook table for the HuffmanOnly mode shows only the codes for input values from 0 to 222. The full codebook is too large to show and its importance is not high, since the HuffmanOnly mode of the data compressor is not the principally mode used within this research.

Table A.1.: The Huffman Codebook for the 1Q version of the data compressor

Delta	Code	Delta	Code	Delta	Code	Delta	Code	Delta	Code
-512	000000	-61	23FFE4	-20	11E200	21	11E800	62	1DFF68
⋮	⋮	-60	23FFE5	-19	11E400	22	11E000	63	1BFE90
-100	21FFB0	-59	23FFE6	-18	11EA00	23	11DE00	64	1BFE70
-99	23FFBE	-58	1FFF90	-17	11EC00	24	11D400	65	1DFF50
-98	23FFBF	-57	23FFE7	-16	F8C00	25	11DA00	66	1DFF70
-97	23FFC0	-56	23FFE8	-15	F9800	26	13F300	67	1DFF78
-96	23FFC1	-55	23FFE9	-14	F9C00	27	15F880	68	1DFF58
-95	23FFC2	-54	21FFB2	-13	FA400	28	13EE00	69	1DFF20
-94	23FFC3	-53	21FFB4	-12	FB000	29	15F780	70	1FFFA8
-93	23FFC4	-52	23FFEA	-11	FC000	30	15F700	71	1DFF28
-92	23FFC5	-51	1FFF94	-10	FCC00	31	15F800	72	1FFFAC
-91	23FFC6	-50	1DFF60	-9	D5000	32	15F600	73	1FFF9C
-90	23FFC7	-49	1FFF98	-8	D6000	33	15F680	74	23FFEB
-89	23FFC8	-48	1DFF38	-7	D7000	34	17FC00	75	1DFF30
-88	23FFC9	-47	1DFF40	-6	D8000	35	15F500	76	21FFB6
-87	23FFCA	-46	1FFFA4	-5	B1000	36	15F480	77	23FFEC
-86	23FFCB	-45	1BFEB0	-4	B2000	37	17FC40	78	23FFED
-85	23FFCC	-44	1DFF48	-3	B3000	38	15F400	79	21FFB8
-84	23FFCD	-43	1DFF80	-2	8C000	39	17FB00	80	23FFEE
-83	23FFCE	-42	1BFE80	-1	8E000	40	17FBC0	81	23FFEF
-82	23FFCF	-41	1BFED0	0	68000	41	17FB40	82	23FFF0
-81	23FFD0	-40	1BFEF0	1	64000	42	17FAC0	83	23FFF1
-80	23FFD1	-39	19FCC0	2	60000	43	17FA00	84	23FFF2
-79	23FFD2	-38	19FC80	3	B4000	44	17FA40	85	21FFBA
-78	23FFD3	-37	19FCE0	4	B0000	45	17F980	86	1FFFA0
-77	23FFD4	-36	19FE00	5	D7800	46	19FE40	87	21FFBC
-76	23FFD5	-35	17F900	6	D6800	47	19FE20	88	23FFF3
-75	23FFD6	-34	19FDC0	7	D5800	48	19FD00	89	23FFF4
-74	23FFD7	-33	19FDE0	8	FD000	49	19FD80	90	23FFF5
-73	23FFD8	-32	17F940	9	FC800	50	19FD40	91	23FFF6
-72	23FFD9	-31	17F9C0	10	FC400	51	19FDA0	92	23FFF7
-71	23FFDA	-30	17FA80	11	FBC00	52	19FCA0	93	23FFF8
-70	23FFDB	-29	17FB80	12	FB800	53	19FD60	94	23FFF9
-69	23FFDC	-28	15F580	13	FA000	54	19FD20	95	23FFFA
-68	23FFDD	-27	13EF00	14	FB400	55	1BFF10	96	23FFFB
-67	23FFDE	-26	13F000	15	FA800	56	1BFF00	97	23FFFC
-66	23FFDF	-25	13F200	16	FAC00	57	1BFEA0	98	23FFFD
-65	23FFE0	-24	13F100	17	F9400	58	1BFEE0	99	23FFFE
-64	23FFE1	-23	11D600	18	F9000	59	1BFEC0	100	23FFFF
-63	23FFE2	-22	11D800	19	F8800	60	1BFE60	⋮	⋮
-62	23FFE3	-21	11DC00	20	11E600	61	1DFF88	511	0000000

Table A.2.: The Huffman Codebook for the 4Q version of the data compressor

Delta	Code	Delta	Code	Delta	Code	Delta	Code	Delta	Code
-512	000000	-61	23FFDE	-20	13F100	21	13EF00	62	21FFB0
⋮	⋮	-60	23FFDF	-19	13F200	22	13F000	63	1FFF90
-100	23FFB8	-59	23FFE0	-18	11D600	23	13EB00	64	21FFB2
-99	23FFB9	-58	23FFE1	-17	11D800	24	13ED00	65	1FFF94
-98	23FFBA	-57	23FFE2	-16	11DE00	25	15F900	66	1FFF98
-97	23FFBB	-56	1FFF70	-15	11E000	26	15F780	67	1FFF9C
-96	23FFBC	-55	23FFE3	-14	11E200	27	15F800	68	1FFF7C
-95	23FFBD	-54	21FFAE	-13	11E800	28	15F600	69	1FFF80
-94	23FFBE	-53	1FFF74	-12	F9800	29	15F580	70	23FFE5
-93	23FFBF	-52	23FFE4	-11	FA400	30	15F300	71	1FFF84
-92	23FFC0	-51	1DFF40	-10	FB800	31	15F480	72	21FFB4
-91	23FFC1	-50	1FFF88	-9	FB400	32	17FCC0	73	23FFE6
-90	23FFC2	-49	21FFAA	-8	FBC00	33	17FAC0	74	23FFE7
-89	23FFC3	-48	1FFFA0	-7	FC800	34	17FBC0	75	21FFAC
-88	23FFC4	-47	1FFFA4	-6	FD000	35	17FB40	76	23FFE8
-87	23FFC5	-46	1DFF48	-5	D7000	36	17FB00	77	23FFE9
-86	23FFC6	-45	1BFE60	-4	D7800	37	17FA00	78	23FFEA
-85	23FFC7	-44	1DFF50	-3	D8800	38	19FDE0	79	23FFEB
-84	23FFC8	-43	1BFE90	-2	B5000	39	19FE40	80	23FFEC
-83	23FFC9	-42	1BFEA0	-1	90000	40	19FDC0	81	23FFFF
-82	23FFCA	-41	19FD20	0	48000	41	19FE00	82	21FFB6
-81	23FFCB	-40	1BFEC0	1	40000	42	19FD60	83	23FFED
-80	23FFCC	-39	19FD00	2	92000	43	19FD80	84	23FFEE
-79	23FFCD	-38	19FE20	3	B4000	44	19FDA0	85	23FFEF
-78	23FFCE	-37	17F980	4	D9000	45	1BFF00	86	23FFF0
-77	23FFCF	-36	17FA40	5	D8000	46	1BFF20	87	23FFF1
-76	23FFD0	-35	17F9C0	6	D6800	47	19FD40	88	23FFF2
-75	23FFD1	-34	17FA80	7	D6000	48	1BFF10	89	23FFF3
-74	23FFD2	-33	17FC00	8	FCC00	49	1BFED0	90	23FFF4
-73	23FFD3	-32	17FB80	9	FC400	50	1BFEE0	91	23FFF5
-72	23FFD4	-31	17FC40	10	FC000	51	1DFF60	92	23FFF6
-71	23FFD5	-30	17FC80	11	FAC00	52	1BFEF0	93	23FFF7
-70	23FFD6	-29	15F400	12	FB000	53	1DFF68	94	23FFF8
-69	23FFD7	-28	15F380	13	FA800	54	1BFE70	95	23FFF9
-68	23FFD8	-27	15F500	14	F9C00	55	1BFEB0	96	23FFFA
-67	23FFD9	-26	15F680	15	FA000	56	1BFE80	97	23FFFB
-66	23FFDA	-25	15F700	16	11E600	57	1DFF58	98	23FFFC
-65	23FFDB	-24	15F880	17	11E400	58	1DFF30	99	23FFFD
-64	23FFDC	-23	13EC00	18	11DA00	59	1FFF8C	100	23FFFE
-63	21FFA8	-22	13EA00	19	11DC00	60	1FFF78	⋮	⋮
-62	23FFDD	-21	13EE00	20	11D400	61	1DFF38	511	0000000

Table A.3.: The Huffman Codebook for the HuffmanOnly mode of the data compressor

Delta	Code	Delta	Code	Delta	Code	Delta	Code	Delta	Code
0	21FF54	45	11B400	90	13C400	135	17F8C0	180	19FCC0
1	23FF86	46	11A400	91	13C700	136	17F7C0	181	19FCE0
2	64000	47	11A800	92	13C600	137	15E200	182	19FB20
3	60000	48	11A600	93	13CA00	138	17F880	183	19FA20
4	8A000	49	119A00	94	13C900	139	15E100	184	19FA00
5	88000	50	11A000	95	13C800	140	17F780	185	19FB40
6	AF000	51	119C00	96	13C100	141	17F740	186	1BFE30
7	AE000	52	11A200	97	15F180	142	17F940	187	19FAA0
8	AD000	53	119800	98	13C000	143	17F900	188	19FA40
9	AC000	54	119E00	99	13C500	144	17F840	189	1BFDE0
10	D2800	55	119400	100	15EF00	145	17F680	190	19FA60
11	D2000	56	119600	101	15F080	146	17F440	191	19F9A0
12	D1800	57	119200	102	13C200	147	17F5C0	192	1BFE50
13	D1000	58	118E00	103	15F100	148	17F700	193	19FAC0
14	D0800	59	119000	104	13C300	149	17F300	194	1BFD70
15	D0000	60	118C00	105	15ED00	150	17F800	195	1BFE00
16	F7800	61	118800	106	15EE00	151	17F400	196	19F980
17	F7400	62	118A00	107	15EE80	152	17F380	197	19F9C0
18	F7000	63	118600	108	15EC80	153	17F480	198	1BFE40
19	F6800	64	118200	109	15EF80	154	17F500	199	1BFE10
20	F6C00	65	118400	110	15ED80	155	17F4C0	200	1BFDC0
21	F6400	66	13DF00	111	15F000	156	17F600	201	19FA80
22	F6000	67	117C00	112	15EA00	157	17F540	202	1BFD40
23	F5C00	68	117E00	113	15E980	158	17F6C0	203	19F9E0
24	F5800	69	118000	114	15EC00	159	17F2C0	204	1BFE20
25	F5400	70	13DE00	115	15EA80	160	17F640	205	1BFD50
26	F5000	71	13DD00	116	15E800	161	17F3C0	206	1BFD10
27	F4800	72	13DB00	117	15E880	162	17F340	207	1BFD20
28	F4C00	73	13DC00	118	15E680	163	17F580	208	1DFED0
29	F4000	74	13D800	119	15EB80	164	19FC60	209	1BFDF0
30	F3C00	75	13DA00	120	15EB00	165	17F280	210	1BFDB0
31	F3400	76	13D900	121	15E780	166	19FCA0	211	1BFDD0
32	11BE00	77	13D700	122	15E900	167	17F240	212	1DFEC8
33	F3000	78	13D300	123	15E300	168	19FC80	213	1BFD80
34	11B000	79	13D600	124	15E580	169	17F200	214	1BFD30
35	11B800	80	13CC00	125	15E400	170	19FC40	215	1DFED8
36	11BC00	81	13D200	126	15E380	171	19FB00	216	1BFDA0
37	11AE00	82	13D500	127	15E700	172	19FB80	217	1BFD00
38	11B200	83	13CE00	128	15E180	173	19FB60	218	1BFD60
39	11AA00	84	13CF00	129	15E480	174	19FC00	219	1DFEE0
40	F4400	85	13D100	130	15E500	175	19FBA0	220	1BFD90
41	F3800	86	13D400	131	15E000	176	19FBC0	221	1DFEB0
42	11BA00	87	13CB00	132	15E600	177	19FC20	222	1DFEE8
43	11B600	88	13D000	133	15E280	178	19FAE0		⋮
44	11AC00	89	13CD00	134	15E080	179	19FBE0	1023	0000000

B. Matlab codes

In this section the code of the two Matlab functions for the decompressor is given. These functions are used to decompress the output data of the DataCompressor unit.

```
% ****************************************************************
%  Huffman_decoder30bit.m
%
% Christian Patauner
% 04.03.2010: Performs the decompression of the output data of
%             the DataCompressor by decompressing the Huffman
%             bitstream reconstruction of the NormSamples from
%             the Deltas and the reference vector.
%
% - 01.08.2011:  Add Comments
%
% ****************************************************************
% Variable and functions
% ****************************************************************
% InputMatrix = Matrix which contains the symbols to code
% Codebook = Is the codetable for decoding the Huffman bitstream
% Input_words30bit = Are the input Huffman coded bitstream words
% ClusterWidth = Is a vector which contains the length (number of
%                samples) of each compressed input waveform
%                extracted from the output data package.
% MaxValues =    Is a vector which contains the maximum sample
%                values of the compressed waveforms extracted
%                from the output data package.
% NrP =          Is a value representing the total number of
%                compressed waveforms extracted from the output
%                data package.
% Q =            Is a vector (only one reference vector) or
%                matrix containing the reference vectors.
% NormValue =    Is the programmed NormValue to which the input
%                waveforms are normalized.
% MaxIndex =     Is the value giving the Index of the maximum
%                value in the reference vectors.
% RefVector_Index = Is a vector containing the indexes which
%                   define the used reference vector for each
%                   input waveform.
%
% ReconstructPulses = Is the output containing the decompressed
%                     waveforms (and can be compared to the
% original input waveform)
% Origin_Delta = Contains the decompressed Delta values out of
%   the Huffman bitstream.
```

```
%
% Used Functions: Decompressor_vector_q.m; Hex2bitStream.m
%
% ***************************************************************

function [ReconstructPulses,Origin_Delta] = Huffman_decoder30bit...
         (Codebook, Input_words30bit, ...
          ClusterWidth, MaxValues, NrP,...
          Q, NormValue, MaxIndex,...
          RefVector_Index)

%*************************************************************
% Initialization of variables
%*************************************************************
index_max = length (Codebook(:,5));
i = 1;
x = 1;
j = 1;
k = 1;
n = 1;
m = 1;
NoCompressData = 0;
index_new = [1:index_max];

%*************************************************************
% Create output bit stream from hexadecimal 30 bit input words
%*************************************************************
for l = 1:numel(Input_words30bit)
   Input_bin(l,:) = Hex2bitStream (Input_words30bit(l),30);
end
  Input_bin2 = Input_bin(:,end:-1:1);

%*************************************************************
% Decodes Huffman bitstream to obtain the Delta values
%*************************************************************
while (m <= numel(Input_words30bit))
  if (NoCompressData == 0)
    while (i<=30)
    symbol = Input_bin2(m,i);
    index_old = index_new;
    index = find (Codebook(index_old,4+x)==symbol);% find Code
    index_new = index_old(index);
    if (numel (index)>1)
        x = x +1 ;
    else
        %*********************************************************
        % when a valid codeword is found representing a decoded
        % Delta
        %*********************************************************
        Origin_Delta(n,j) = Codebook(index_new,1);
```

```
if (numel(ClusterWidth)>=n && j == ClusterWidth(n))
% when enough deltas are decoded to form the new waveform
    if exist('RefVector_Index') ~= 0  && numel(Q)>1
        %*************************************************
        % if more than one reference vector was used the
        % following is used to recalculate NormSamples
        % from the Deltas and the original sample values
        % by renormalizing
        %*************************************************
        [ReconstructPulses(n,1:64)] = ...
         Decompressor_vector_q (Origin_Delta(n,:)',...
             MaxValues(k), ClusterWidth(n), Q, ...
             NormValue, MaxIndex, RefVector_Index(k));
    elseif (numel(Q)>1) % if only one reference vector
        [ReconstructPulses(n,1:64)] = ...
         Decompressor_vector_q (Origin_Delta(n,:)',...
         MaxValues(k), ClusterWidth(n), Q,...
         NormValue, MaxIndex);
    end
    n = n + 1; % counts for the next input waveform
 end
    % when the whished number of decompressed pulses is
    % archived stops executing by breaking the loop
    if (n==NrP+1) break; end

    k = k + 1; % counts only compressed waveforms (not too
               % long or too short uncompressed clusters)
    j = 0;
    if (numel(ClusterWidth)<n) % if not enough deltas are
           % decoded for the corresponding waveform length
        x = 1;
        j = j +1;
        index_new = [1:index_max];
        break;
    end
    %*****************************************************
    % Indicates that a cluster with too long or too short
    % length is received uncompressed. Sets
    % NoCompressData one and breaks loop to jump in the
    % NoCompressData routine
    %*****************************************************
  if ((ClusterWidth(n)>32||ClusterWidth(n)<4)&&numel(Q)>1)
      NoCompressData = 1;
      break;
    end

end
x = 1;
j = j +1;
index_new = [1:index_max];
```

```
    end


    i=i+1;
  end
    i = 1;
    m = m + 1;
  end

  if (numel(Q)<=1)
    ReconstructPulses = Origin_Delta;
  end
 %************************************************************
 % Routine for the handling of uncompressed waveforms which are
 % too long or too short
 %************************************************************
  if (NoCompressData == 1)
   j = 1;
   x = 1;
   p = 1;
   index_new = [1:index_max];
   while(p<ClusterWidth(n)) % counts and converts the following 10
                            % bit sample words
   ReconstructPulses(n,p:p+2)=[binvec2dec(Input_bin2(m,10:-1:1)),...
                               binvec2dec(Input_bin2(m,20:-1:11)),...
                               binvec2dec(Input_bin2(m,30:-1:21))];
     p = p + 3;
     m = m + 1;
   end
   n = n + 1;
   if (ClusterWidth(n)>3&&ClusterWidth(n)<33) % if a new compressed
                                              % waveform arrives
      NoCompressData = 0;
   end
   if (n==NrP+1) break; end
  end
end

% ********************************************************************
%  UnComp_quanttiz_vector.m
%
% Christian Patauner
% 17.10.2009: Uncompressing the Delta values from the vector
%             quantization to reconstruct the original normalized
%             pulses and then renormalizing them to reconstruct
%             the original waveforms
%
% - 01.08.2011:  Add Comments
%
% ********************************************************************
```

```
% Variables and functions
% ************************************************************
% Input_Delta = Matrix with the Delta values Huffman decompressed
% Q =           Reference vectors
% NormFac =     Normalization factor to renormalize the samples
%
% Orig_waveforms = Resulting original waveforms (reconstructed and
%                  renormalized
% ReconstructWaveforms = Aligned and rounded Orig_waveforms
% Norm_WF_re  =    Resulting reconstructed normalized waveforms
%                  using the Delta values and the Q vector
%
% no custom functions used
%
% ************************************************************

function [ReconstructWaveforms, Norm_WF_re, Orig_waveforms] =...
          Decompressor_vector_q (Input_DeltaStream, ...
          WaveformsMax, WaveformsLength, Q, NormValue, ...
    NormValueIndex, RefVecIndex)

%************************************************************
% Initialization of variables
%************************************************************
n = numel(WaveformsMax); % number of contained waveforms in the
                         % Input data
LengthSum = 0;
ReconstructWaveforms = zeros(1,64);

%************************************************************
% sorting the Received Deltas in the corresponding waveforms
% clusters and searching the first 0 Delta in the rising edge
% for the alignment index
%************************************************************
for j = 1 : numel(WaveformsMax)
DeltaShaped(j,1:WaveformsLength(j)) = ...
   Input_DeltaStream(LengthSum+1:LengthSum+WaveformsLength(j));
LengthSum = LengthSum + WaveformsLength(j);
MaxIndexTemp = find (DeltaShaped(j,1:WaveformsLength(j)) == 0);
MaxIndex_Rec(j,:) = MaxIndexTemp(end);
end

%************************************************************
% Calculates the IndexDifference and aligning the Deltas of the
% waveforms for adding them to the corresponding reference vector
%************************************************************
IndexDiff = NormValueIndex -MaxIndex_Rec;
for i = 1:numel(WaveformsMax)
Delta_Aligned(:,i) = [zeros(1,IndexDiff(i)),DeltaShaped(i,...
                     1:WaveformsLength(i)),zeros(1,64-...
```

```
                    IndexDiff(i)-WaveformsLength(i))];
end

Delta_Aligned(32,:) = ...
round((floor((NormValue*2^6)./WaveformsMax).*WaveformsMax)./2^6)...
  -100;

% Calculating the normalization factor out of the maximum samples
NormFac = floor((NormValue*2^6)./WaveformsMax);

%************************************************************
% Reconstructs the normalized sample values by adding the
% Deltas to the corresponding reference vectors per waveform
% and renormalizing the normalized samples to the original
% samples by multiplying them with the calculated NormFactor
%************************************************************
for i = 1:n
    if exist('RefVecIndex') ~= 0
      Norm_WF_re(:,i) = Delta_Aligned(:,i)+Q(RefVecIndex(i),:)';
    else
      Norm_WF_re(:,i) = Delta_Aligned(:,i)+Q';
    end
      Norm_pu_1Q(:,i) = Norm_WF_re(:,i).*2^6;
    Orig_waveforms(i,:) = (Norm_pu_1Q(:,i)./NormFac(i));
end

% Rounds the obtained renormalized samples to get the
% integer values of the original sample values per waveform
ReconstWaveforms = round(Orig_waveforms);

%************************************************************
% Reorganizing the samples of the input waveform in a matrix
% to compare them with the original test-data matrix
%************************************************************
for i = 1: numel(WaveformsMax)
ReconstWaveforms(i,IndexDiff(i)+WaveformsLength(i)+1:64) = 0;
end
for i = 1: numel(WaveformsMax)
ReconstWaveforms(i,1:IndexDiff(i)) = 0;
end
for i = 1: numel(WaveformsMax)
ReconstructWaveforms(i,1:WaveformsLength(i))= ...
      ReconstWaveforms(i,IndexDiff(i)+1:IndexDiff(i) +...
            WaveformsLength(i));
end
```

C. Verilog Code

In this section the Verilog code of the behavioral model of the Integer Divider is given. This model is written using two parameters to define the number pipelined stages and the number of bits of the Enumerator processed in each stage. In this way, the code can be adapted fast for different clock frequencies and application requirements.

```
/*************************************************************
            Divider module for the Normalizer                *
*************************************************************
                                                             *
file:         VQDiv.v                                        *
authors:      Christian Patauner                             *
creation:     12.07.09                                       *
                                                             *
description: Module containing a function to divide two integer *
values. The divider is needed to calculate the normalization    *
factor to normalize the input waveforms. The function performs a*
"DivDim/NrStage" bit division in several pipeline stages        *
according to the set parameters "DivDim" and "NrStage".         *
                                                             *
Included Modules: No other modules are included              *
                                                             *
 - 17.07.09: using a for loop suggested by Dr. Sandro Bonacini  *
 - 26.08.09: Add comments                                    *
 - 10.10.11: Add comments                                    *
*************************************************************/

//`timescale 10 ps / 1 ps // for a time based simulation

module VQDiv (Result, ResultValid, DivEnd, Numerator, Divisor,
  StartDiv, Clock80, reset);

/*************************************************************
Definition of parameters to adjust the number of pipelined stages
*************************************************************/
parameter DivDim = 15;
parameter NrStage = 5;

/*************************************************************
 Definition of the input and output signals
*************************************************************/
output [9:0] Result;
output ResultValid;
output DivEnd;
input [DivDim-1:0] Numerator;
```

```
input [9:0] Divisor;
input StartDiv;
input Clock80;
input reset;

/*****************************************************************
 Definition of the registers
*****************************************************************/
reg [3:0] x;
reg [DivDim/NrStage-1:0] Numerator2;
reg [DivDim-1:0] Reminder;
reg [9:0] ResultTemp;
reg [9:0] Result;
reg ResultValid, ResultValidEnd;
reg DivEnd, StopDiv;


/*****************************************************************
 Function atomdiv which compares the Divisor with the Reminder
*****************************************************************/
function [DivDim:0] atomdiv;

input [DivDim-1:0] Reminder;
input [DivDim-1:0] Divisor;
reg [DivDim-1:0] Reminder;
reg [DivDim-1:0] Divisor;

/* returns one output bit indicating if the Reminder is greater
   than the Divisor or not; the second part of the output is
   the difference between Reminder and Divisor already shifted
   one position to the right */
  atomdiv = {Reminder>=Divisor,(Reminder-Divisor)<<1};

endfunction


/*****************************************************************
 Function DIV which performs the division in a single stage
*****************************************************************/
function [DivDim*2-1:0] DIV;
  input [DivDim/NrStage-1:0] DivE;
  input [DivDim-1:0] DivD;
  input [DivDim-1:0] Reminder;
  input [DivDim-1:0] ResultTemp;
  reg [DivDim/NrStage-1:0] DivE;
  reg [DivDim-1:0] DivD;
  reg [DivDim-1:0] Reminder2;
  reg [DivDim-1:0] Reminder3;
  reg [DivDim-1:0] Result2;
  reg [4:0] i;
```

```
  begin
   Reminder2 = Reminder;
   Result2 = ResultTemp;

// for loop executing the number of bit times executed per stage
   for (i = 0; i < DivDim/NrStage; i = i + 1) begin

// calls atomdiv to perform the division of the corresp. bits
    {Result2[0],Reminder3} = atomdiv (Reminder2, DivD);

// differentiates if the Reminder is greater than the Divisor or
// not in order to use the new reminder or the old one
    if (Result2[0] == 1) begin
     Reminder2 = {Reminder3[DivDim-1:1],DivE[DivDim/NrStage-1-i]};
    end
    else begin
     Reminder2 = {Reminder2[DivDim-2:0],DivE[DivDim/NrStage-1-i]};
    end
    Result2 = Result2<<1; // shifts the Result register one bit to
       // the right to include the new result bit
   end // end for
   DIV = {Result2,Reminder2};
   end
endfunction

/*****************************************************************
Procedural block: positive clock edge triggered and asynch. reset
*****************************************************************/
always @ (posedge Clock80 or posedge reset) begin
 if (reset == 1) begin
   x <=#1 0;
   DivEnd <=#1 0;
   Reminder <=#1 0;
   ResultTemp <=#1 0;
   Numerator2 = 0;
   Result <=#1 0;
   ResultValid <=#1 0;
   ResultValidEnd <=#1 0;
   StopDiv <=#1 0;
 end
 else begin
 if (StartDiv == 1) begin
   if (x < NrStage) begin
    case (x) // state machine for handing over the corresponding
  // bits of the enumerator to the function in each stage
   5'd0: Numerator2 = Numerator[DivDim-1:DivDim-DivDim/NrStage];
   5'd1: Numerator2 = Numerator[DivDim-1-DivDim/NrStage:DivDim-
                      DivDim/NrStage*2];
   5'd2: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*2:DivDim-
                      DivDim/NrStage*3];
```

```
   5'd3: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*3:DivDim-
                        DivDim/NrStage*4];
   5'd4: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*4:DivDim-
                        DivDim/NrStage*5];
//  not used stages are commented
// 5'd5: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*5:DivDim-
                        DivDim/NrStage*6];
// 5'd6: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*6:DivDim-
                        DivDim/NrStage*7];
// 5'd7: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*7:DivDim-
                        DivDim/NrStage*8];
// 5'd8: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*8:DivDim-
                        DivDim/NrStage*9];
// 5'd9: Numerator2 = Numerator[DivDim-1-DivDim/NrStage*8:DivDim-
                        DivDim/NrStage*10];
    default: Numerator2 = 0;
  endcase
  x <=#1 x+1; // switch to next stage
  // save intermediate results in ResultTemp register and update
  // Reminder
 {ResultTemp,Reminder}<=#1 DIV(Numerator2, Divisor, Reminder,
                               ResultTemp);

  // indicates the end of the Division if all stages are executed
    if (x == NrStage-1) begin
      DivEnd <=#1 1;
    end
   end
 end

 // Transfers the ResultTemp value to the output register and
 // updating the last bit of the Result
 if(DivEnd == 1 && StopDiv == 0) begin
   ResultValid <=#1 1;
   Result <=#1 ResultTemp;
   Result[0] <=#1 (Reminder)>=Divisor;
   x <=#1 0;
   //DivEnd <=#1 0;
   StopDiv <=#1 1;
   Reminder <=#1 0;
   ResultTemp <=#1 0;
 end

  // strobe indicating the end of the calculation
  if (StopDiv == 1) begin
    StopDiv <=#1 0;
DivEnd <=#1 0;
  end

 // strobe indicating that the Result is valid
```

```
 if (ResultValid == 1) begin
   ResultValidEnd <=#1 1;
   //ResultValid <=#1 0;
 end

 if (ResultValidEnd == 1) begin
    ResultValidEnd <=#1 0;
ResultValid <=#1 0;
 end

 end    // else if (reset==1) -> reset = 0
end     // always @ (posedge Clock80 or posedge reset)

endmodule
```