GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Fakultät für
Physik

Bachelor's Thesis

Entwicklung neuer Datenübertragungsmethoden für das Auslesesystem des ATLAS Pixel Detektors

Development of new data transmission methods for the read-out system of the ATLAS Pixel Detector

prepared by

Benjamin von Ardenne

from Dresden

at the 2.Physikalisches Institut

Thesis period:	12th April 2010 until 19th July 2010
Supervisor:	Dipl.-Phys. Nina Krieger
First Referee:	PD Dr. Jörn Grosse-Knetter
Second Referee:	Professor Dr. Arnulf Quadt
Thesis Number:	II. Physik-UniGö-BSc-2010/06

Abstract

The ATLAS Pixel Detector, as part of the Large Hadron Collider, will produce large amounts of data. Thus a new data encoding methods called 8b10b will be used in the future to fascilitate error monitoring to conserve the transmission hardware. For that reason, a similar read-out chain in laboratory will be prepared to be able to decode 8b10b signals. A pixel module emulator with an 8b10b encoded data output is developed and utilized to test the system. In the end, the modifications are validated with various error and delay measurements.

Contents

1 Introduction

The idea that all matter is composed of elementary particles dates back to the 6th century BC when people started investigating the philosophical doctrine of atomism and the nature of elementary particles. These early ideas were founded in abstract, philosophical reasoning rather than experimentation and empirical observation. With his work on stoichiometry, John Dalton posed the hypothesis in 1808 that each element of nature was composed of a single, unique type of particle that he called atom, after the Greek word atomos ("indivisible"). Near the end of the 19th century, physicists discovered that atoms were not, in fact, the fundamental particles of nature, but conglomerates of even smaller particles. The finding of the proton in 1919 by Ernest Rutherford and the neutron in 1931 by James Chadwick made people believe that they had found the smallest existing particles.

However, ongoing studies of particles with accelerators and experiments concerning cosmic rays finally revealed a large variety of smaller particles that was also referred to as the "particle zoo". With the formulation of the *Standard Model* during the 1970s, physicists explained this large number of particles as combinations of a small number of fundamental particles using symmetry groups. This Standard Model states that our universe is comprised of 3 fundamental groups of particles:. quarks, leptons and bosons. The latter are responsible for the mediation of forces between particles. Up until today, almost all parameter predictions by the Standard Model (SM), such as masses and decay modes, could be successfully measured and verified. One last piece is missing however. The SM introduces the *Higgs field* which, in theory, is responsible for the mass-generation of all particles. By self-interaction it can produce the *Higgs particle* which is still to be found in modern particle physics and has motivated the most recent experiments.

Despite the success of the Standard Model, a lot of scientists are convinced that it needs some extensions at higher energies. Modifications propose the existence of a super-symmetric partner for every particle or additional space dimensions. For further verification of the Standard Model as well as the search for new particles

or physical behavior, particles are accelerated to high energies and brought to collision. The Large Hadron Collider (LHC) in Genf is currently the world's largest and highest-energy particle accelerator of that kind. But as crucial as the accelerator itself are the instruments to detect the generated particles. Thus collisions at the LHC are precisely measured by four separate experiments resulting in an enormous amount of data. The storage space needed by these experiments will be approximately 15 petabytes (15 million gigabytes) annually – enough to fill more than 1.7 million dual-layer DVDs a year.

Given these large quantities of data being produced by the detectors, errors in the data transmission process are unavoidable. Due to the radiation being produced in the collisions and the distance between the detectors and the read-out hardware, bit flips may occur and lead to wrong experimental data. In order to make measured data more reliable, an established transmission methods called 8b10b encoding was proposed for the communication within the detector. The encoding would allow for easy error detection in the transmitted bit streams between the detector hardware and the off-detector read-out system and conserve the transmission hardware.

The ATLAS Pixel Detector as part of the ATLAS experiment will introduce the 8b10b encoding technique in the Insertable-B-Layer Upgrade. For testing purpose almost identical read-out systems were built in a laboratory environment to facilitate the implementation and testing of these new technologies. For this bachelor thesis, the test system will be prepared for the 8b10b encoding technique. Additionally an emulator of a pixel module has to be written and equipped with a 8b10b encoding unit in order to test the new transmission method in the existing read-out chain. In the end the system will be validated with a large number of events in order to check the functioning under realistic conditions.

Chapter 2 will start with a more detailed report on the LHC experiment, focusing on the ATLAS Pixel Detector as a hybrid silicon detector. Further information will be given of how the experimental read-out chain is rebuilt in laboratory to facilitate the testing of the new read-out chain. In chapter 3 the FPGA technology will be introduced along with the VHDL design language. They are the most important tools in the data read out process. Chapter 4 will give an overview of 8b10b encoding standard that is widely used in the communication industry. Finally a documentation about the modifications that were done on the laboratory test system can be found in Chapter 5, followed by a thorough validation of the newly developed system.

2 The LHC and the ATLAS Experiment

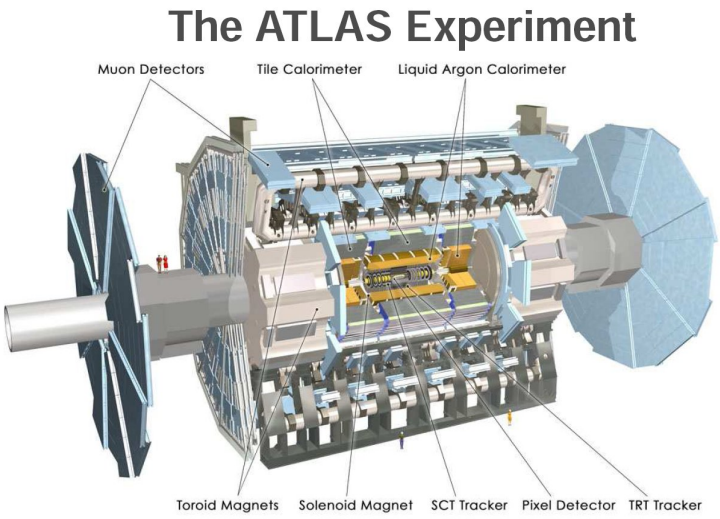


Figure 2.1: Schematic overview of the ATLAS experiment with the Muon Chamber, the Calorimeters and the Inner Detector.

The LHC began operation in September 2008. It lies in a tunnel 27 kilometers in circumference, as much as 175 meters beneath the Franco-Swiss border near Geneva, Switzerland. In the final design, proton packages will collide with a center of mass energy of up to 14 TeV at a luminosity of $10^{34} \mathrm{c} m^{-2} s^{-1}$. These packages will circulate the ring with up to $1.1 \cdot 10^{11}$ particles being only 25 ns apart. There are 4 collision points with experiments built around them to study the collisions.

The ATLAS experiment (see figure 2.1) is one of two main-purpose particle detectors at the LHC. Collisions take place in the center of the construction and are measured in all directions which is why it is called a 4π-detector. It has a total length of 45m which makes it one of the largest scientific experiments in the world.

The outermost parts of the detector are the muon chambers which are placed in

a 0.5 T toroidal magnetic field. The next layer consists of the hadronic and electromagnetic calorimeter that can measure the energy of single particles and jets. The "Inner Detector" has been designed for precise tracking and vertex reconstructing of charged particles and is the focus of this thesis. It comprises of a transition radiation tracker, a silicon strip detector and a silicon pixel detector. The bending of the trajectories of the charged particles is done by a 2 T solenoid magnetic field to facilitate momentum measurement.

2.1 The ATLAS Pixel Detector

As the innermost part the Pixel Detector is subdivided into three barrel layers at radii of 50.5 mm, 88.5 mm and 122.5 mm along with three disks on each side for the forward direction at a distance of 49.5 cm, 58 cm, and 65 cm from the interaction point[1]. It has a size of approximately 1.4 m which results in detecting at least three hits per traversing particle with a pseudo-rapidity[1] of $|\eta| < 2.5$. Since the Pixel Detector is very close to the collision point it faces special requirements. Due to the high particle flux the components were designed to have a very good radiation hardness. Furthermore the read-out system of this layer has been designed for high transmission rates to be able to transmit the recorded data off the detector as fast as possible to avoid an overflow of the module hit buffers.

The actual pixel components of the detector consist of 1744 identical sensor-chip-hybrid modules which have a total of 80 million pixels. The pixel sensors consist of oxygenated silicon n-type bulk material with n^+ pixel implants. Each module has 47,232 of those pixels which can be individually readout by 16 front-end (FE) chips. The FE chip is designed to digitize the charge signal received from the sensor pads directly on the module. It contains 2880 individual charge sensitive analog circuits with a digital read-out that operates at the global clock speed of 40 MHz. The circuit contains a fast preamplifier, a DC-coupled second amplifier and a differential discriminator. The preamplifier integrates the induced charge of the sensor pads using a feedback capacitor which is constantly discharged by a feedback current. The discriminator compares the input signal with an adjustable threshold value. If the signal is higher than this threshold, the output is a logical one, otherwise it is logical zero. The actual measurement value is the time, for which the signal has been above the threshold value (time over threshold, ToT). This time is proportional to

[1] spatial coordinate describing the angle of a particle relative to the beam axis

the amount of electrons that have been created by the ionization of the traversing charged particle in the silicon material.

A module control chip (MCC) is responsible for the interaction between the module's FE chips and the further readout chain. It combines the individual events from the FE chips and distributes trigger and command signals. The connection to the off-site read-out machinery is being established via opto-links. The data is serialized on the modules and then being transmitted via one or two data lines, depending on the data transmission speed selected.

2.2 The Read-out Chain

Not every collision in the detector will lead to physically interesting events. To avoid the acquisition of useless data, the concept of event triggering is being used to read out only promising events. Trigger decisions can be made by various components in the detector, like the muon chamber for example. In the case of a trigger, a special signal - the Level1 (LV1) trigger - is sent to the individual pixel modules. Following this trigger, the MCC chip is responsible to format the hits of all the FE chips to one single event. These packets are then sent to the off-detector electronics with a speed of 40 Mb/s, 80Mb/s or 160 Mb/s depending on the module layer. In order to assign these event packages to the correct collision, the modules and the off-detector hardware both have two important counters. The first counter keeps tracks of the number of Level1 triggers, that have been sent. This Level1 ID (LV1ID) will be sent once for every event and contains the Level1 triggering ID of the event. The second counter is the bunch crossing ID (BCID) that is being increased every 25 ns or 40MHz. It is used to link an event to an actually moment in time. Both LV1ID and BCID can be reset and hence resynchronized by command signals sent by the off-detector hardware.

The receiving end of the MCC signal is the BOC (Back-of-Crate Card). The optical signal is detected by PiN[2] diodes which convert it into an electrical one. The BOC provides the complete timing functionality for the pixel detector. Each BOC forwards the received data to a ROD [3]. This device is mainly responsible for generating command bit-streams for the modules and reading out the data. After data collection the ROD is responsible for building a common event and sending it to

[2]Diode with an extra wide, lightly doped 'near' intrinsic semiconductor region between a p-type semiconductor and an n-type semiconductor region.

[3]Read-Out Driver

the ATLAS DAQ[4]-System via another optical S-Link on the BOC. Besides that the ROD also offers histogramming functionality and analysis capabilities using DSPs[5]. These information can be used during data taking for monitoring or for calibration purposes such as measurement and adjustment of the FE threshold settings.

2.3 ATLAS Upgrade - The Insertable B-Layer

The continuous radiation dose received by the individual pixel modules in the detector, especially those situated extremely close to the collision point, causes damage to the sensor material. This leads to a decreased signal strength of time which is why the innermost layer will be replaced within the IBL[6] project[2]. With a radius of about 37mm pixel modules will be even closer to the interaction point and render higher resolution. The upgrade poses new requirements for the sensors and the electronics due to higher radiation and luminosity. It is planned to increase the read-out speed to a maximum of 160Mbit/s for each channel while the global read-out clock remains at 40MHz. Simultaneously some new features are going to be implemented such as the 8b10b encoding that will be explained in chapter 4. For this purpose a new version of the FE chip, the FE-I4, is under development and will replace the current FE-I3 chip. At the beginning of July 2010 the final design for the FE-I4 has been submitted. One module will only contain 2 large FE-chips with 26 880 pixels that are smaller than before[3]. The active area of the sensor modules will increase from approximately 75% to 90%. The MCC chip does not contain digital functionality anymore and its functionality is replaced by the FE circuitry. To achieve higher data rates, the incoming 40MHz clock will be multiplied and phase shifted in the chip to get the desired 160MHz clock. The same has to be done on the receiving end at the BOC although it is being considered to provide a 80MHz clock via VME.

The test system, that is explained in the following section, needs to be adapted to these changes of the FE architecture. The Master DSP[7] code needs will be modified to understand the new FE-I4 data protocol. As part of this thesis, the eBOC (see section 2.4.1) needs to be prepared for the 8b10b decoding of the data signals.

[4]Data Acquisition
[5]Digital Signal Processors
[6]Insertable B-Layer
[7]Digital Signal Processor

2.4 The Read-Out Chain in Laboratory

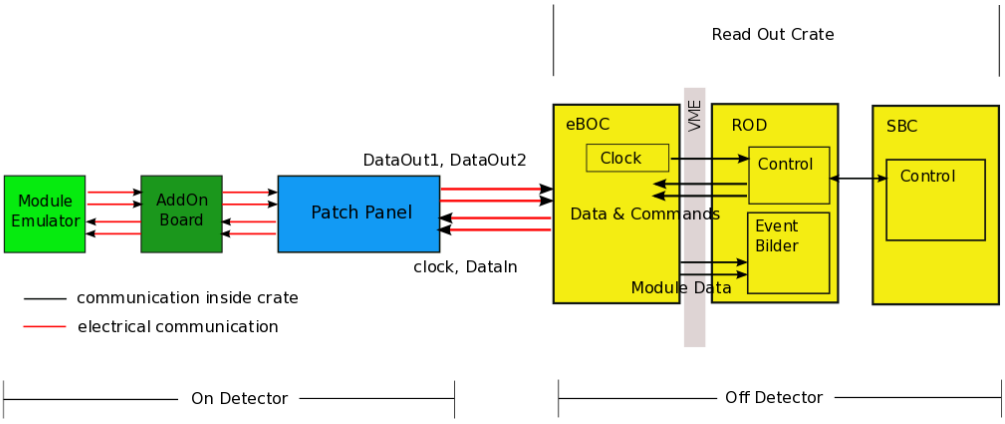


Figure 2.2: Scheme of read-out chain in laboratory with a module emulator that is connected to the system via an add-on board. Instead of the emulator, a normal pixel module can also be tested.

The optical data transmission needs a lot of tuning to work reliably. One has to monitor and recalibrate several voltage values to successfully run all the optical components. For pixel module testing purposes the optical BOC is too sensitive to changes of its environment and an optical data transmission is not required due to the small distance between modules and crate. Since optical data transmission is not needed for testing purposes, a test system has been developed that replaces all optical components and transmits the data electrically. Thus the optical BOC is replaced by an electrical version, the eBOC. Every eBOC is connected, like the optical equivalent, one-to-one to a Read-Out Driver. This version of the BOC is also responsible for generating the clock signal with the help of an oscillating crystal. The main purpose is to send the command signals from the ROD to the modules and forward the received data to the ROD. To replace the optical interface between eBOC and modules, a Patch Panel (PP0) with an FPGA has been developed. Instead of converting the module signals from and into optical ones, the PP0 forwards the individual signals and acts as an electrical signal bridge.

The electrical read out chain is more reliable and robust e.g. to changes of the room temperature. It proves also to be easier to set up than optical systems.

2.4.1 The Electrical Back of Crate Card

The eBOC is equipped with an Altera Flex6000 FPGA unit (see chapter 3.1). Channels are available for up to 28 modules at 40 Mbit/s, 14 modules at 80 Mbit/s and 7 modules at 160Mbit/s. In the upper right part of the board, the outgoing channels are for direct transmission of clock and command signals to the pixel modules (see figure 2.3). The lower half of the module holds the receiver data streams. The incoming data streams are routed into the FPGA chip, where they can be split, rerouted and sent to the ROD with 40MBit/s for further processing. This first design of the eBOC has been supplied by Berkeley National Laboratories, California. A second revision along with the full support of different data transmission speeds has been set up for the laboratory by Mathias George in his diploma thesis[2].

The eBOC sends the 40MHz clock on several channels through the same wiring, the data signal is being transmitted. On the PP0 board, that is placed in between the eBOC and the modules, the clock signal is being retransmitted to the eBOC. This makes it easy for the receiving parts of the eBOC to compensate for time delay in the data signal due to cable length by simply using the also shifted clock to receive the data.

All routing activies are being done by the Altera FPGA[4]. It manages all data streams from the ROD to the modules, from the modules to the ROD, and all clock signals. More details about the mode of operation of the FPGA can be found in the revised code implementation of the eBOC in section 5.2.

The FPGA does not have its own memory to store chip configurations. A PROM[8] is placed next to the chip. Every system-start-up, the FPGA automatically loads its configuration from the PROM and keeps this configuration until the reset signal is sent to the FPGA or it is powered down. Alternatively the chip can programmed via a serial cable and the Altera Quartus Programming Software[4].

[8]Programmable Read-Only Memory

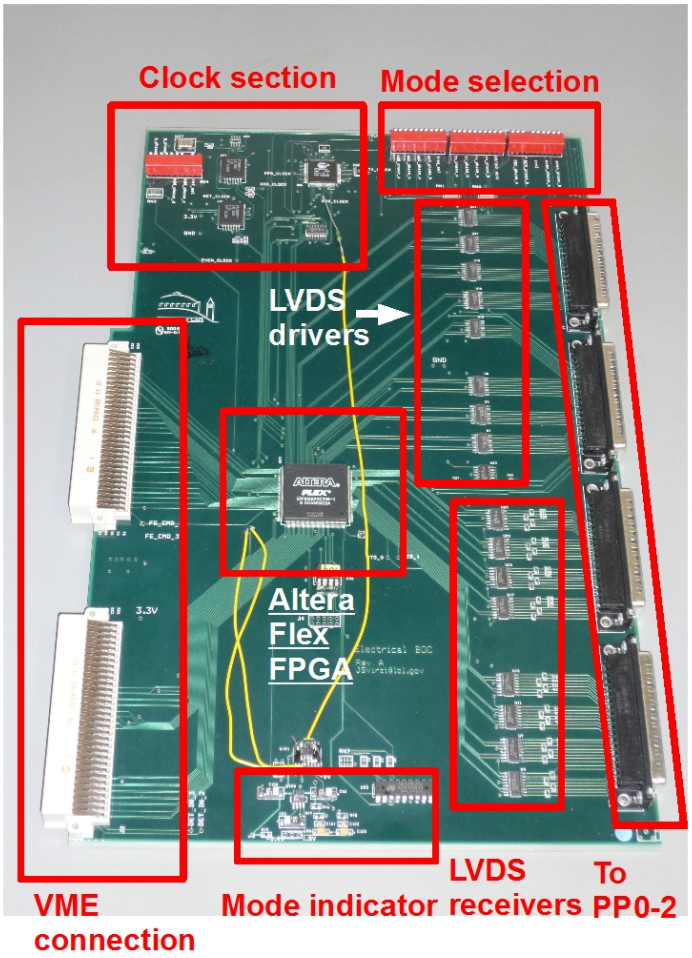


Figure 2.3: Picture of the eBOC with its functional sections.

All electrical data communications are done with differential signals via LVDS*[5]*[9] drivers. Normally a binary signal is transmitted via a single line. The logic zero is equivalent to a 0 Volt signal, also called "low" while the logic one is a 3.3 Volt signal, which is also referred to as "high". Definitions of logic one and logic zero might vary from system to system due to certain design issues or the nature of the transmitted signals. However for long distance electrical communication, these basic communication protocols are not suitable due to the large capacity of the wires. The LVDS system uses two wires named *n (negative)* and *p (positive)*. The information is encoded in the voltage difference between these two wires which is

[9]Low-voltage differential signaling

created by a comparably small charge of 3.5 mA[3] that is injected into one of the two wires (depending on the information being transmitted). The differential voltage is only about 350 mV high and leads to a decreased power consumption of the system. Besides that LVDS is designed[10] for a possible speed of over 1.9 GBit/s which makes it suitable for fast data transmission systems. The maximum expected speeds at IBL will be 160 MBit/s and thus lies within current specifications of the LVDS transmission.

[10] ANSI/TIA/EIA-644-A (published in 2001) standard defines LVDS specifications

3 Hardware and Software

3.1 Field Programmable Gate Arrays (FPGA)

Figure 3.1: Picture of a modern Altera Stratix IV FPGA embedded in a development board.

In the early days of the micro-electronic development, new signal processing tasks demanded for specially engineered chips that were designed for a single purpose only. These application-specific integrated circuit (ASIC) chips fulfill their task fast and efficiently but their development is lengthy and expensive. Meanwhile a new technology has emerged that greatly facilitates the development of new electric signal processing systems - the Field Programmable Gate Arrays (FPGA)[4, 6, 7].

Instead of a predefined logic structure, they contain large arrays of programmable logic components called "logic blocks", and a hierarchy of reconfigurable interconnections that allow these blocks to be "wired together" - somewhat like an on-chip programmable breadboard. These connections can achieve tasks ranging from simple counters to highly complex structures like microprocessors.

The FPGA routing is done via programmable switch boxes. Each wiring segment only connects two logic blocks. In order to establish longer paths, the switches can be turned on to built up complex paths. For higher speed interconnects such as

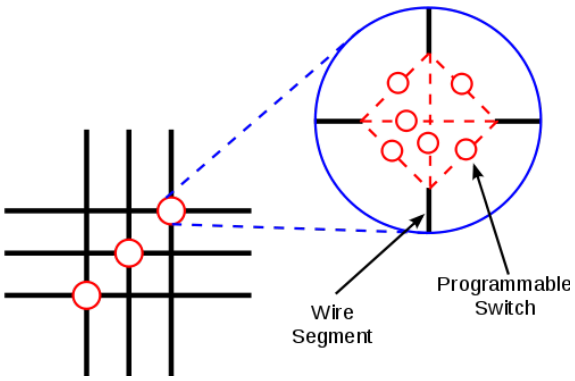


Figure 3.2: The switch box topology of modern FPGAs.

clock channels, modern FPGA chips offer special routing lines that span multiple logic blocks and are optimized for timing related signals. The switch box itself (as can be seen in figure 3.2) can route any one of the wires to another adjacent channel segment, illustrated by the two red circles within the switchbox rhombus. To allow for the complex interconnectivity that is needed by several FPGA designs, modern FPGAs contain up to 12 different wiring layers of this kind on a single chip.

3.1.1 History

The FPGA industry sprouted from programmable read-only memory (PROM) and programmable logic devices (PLDs)[8]. The first devices of that kind were invented by Motorola, Texas Instruments and National Semiconductor in the early 1970s. They contained a plane with AND-gates and another plane with OR-gates that could be programmable interconnected. Simple summation and multiplication circuits could be built with them but they were slow and too expensive. The need for higher logic capacity and more inputs led to the idea of linking several logic units, that each contained programmable logic, to a large array. Following this idea, Xilinx Co-Founders, Ross Freeman and Bernard Vonderschmitt, invented the first commercially viable field programmable gate array in 1985 – the XC2064. It had 64 configurable logic blocks with programmable interconnects between them.

Over the years, new wiring techniques such as the antifuse-technology[6] were developed to increase logic density and circuit speed. Modern FPGAs like the Xilinx Virtex-6 family contain more than 800.000 logic blocks and are equipped with a large battery of features such as on-chip SRAM[1], interface blocks for PCI-

[1]Static random access memory

Express[2] or Ethernet and hard wired signal processing units.

3.1.2 Functionality

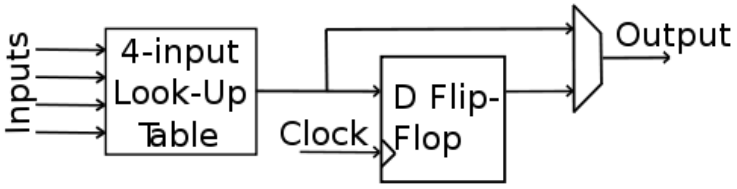
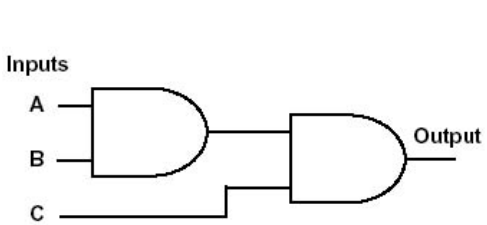


Figure 3.3: Simple logic block of an FPGA.

In order to define the behavior of an FPGA, one has to provide a schematic design or a language description of the design which is implemented through a hardware description language (HDL) (see section 3.2). This design needs to be synthesized into a structure that can be realized with the available logic blocks in the FPGA. Certain design constraints can be applied at this stage of development such as linking certain ports to external connections. FPGAs have very strong testing and simulation capabilities that allow the designer to check the functionality at every step in the development process. After translation the FPGA design into a logic structure, a binary file with a suited description is being generated that can be transmitted to the FPGA. With the help of the binary data, the FPGA will create a wired logic structure that fulfills the desired functionality.

The main element of every FPGA is the configurable logic block (CLB) that consists of a n-input look-up table (LUT) and a D-type flip-flop (DFF) (see figure 3.3).

[2]Peripheral Component Interconnect Express



A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Figure 3.4: Example of a 3-input Look-Up Table.

Look-Up Table (LUT) The look-up table is usually comprised of n binary input channels and a single binary output line. The n inputs can form 2^n different values and act as a bit address for a small SRAM unit that can be programmed. Depending on the input address, the output will take a defined value. Look-up tables can be used to replace computational tasks with pre-saved results in the SRAM. By setting the look-up table to specific values, one can easily synthesize logic structures like AND, OR, NAND and XOR gates. Figure 3.4 shows a simple 3 input LUT that works as a 3-AND gate, whereupon the output will only change to 1, if all 3 inputs are 1 as well. Besides from logic operations, LUTs are also used to store results of common mathematical calculations such as trigonometric functions. In a sine-table, e.g. the sine-values are precalculated and saved in discrete steps, according to the resolution needed for the application. Since the calculation of sine-values takes time that is sometimes not available, this is useful when it comes to high speed signal processing tasks. Instead of calculating the sine value, the function argument is rounded to the nearest table value and simply read out.

D Flip-Flop The D Fip-Fop[9] is a kind of primitive memory cell or delay line and allows for sequential structures in the design. The D in the name (delay) derives from the fact, that the output Q is a delayed copy of the input D (see figure 3.5). At the moment of a positive edge at the clock pin, the device is designed to take the input D signal and set it to the output Q signal. Thus, the input is delayed by a maximum of one clock cycle to the output. After the rising edge of the clock, a change in the input signal does not influence the value of the output signal.

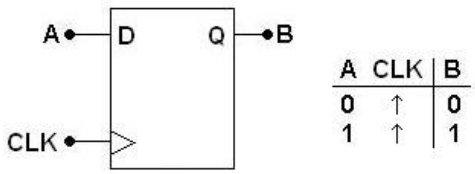


Figure 3.5: Essential part of a FPGA, the D-Flipflop with the input D, the output Q and the clock signal CLK. The ↑ marks the rising edge of the clock signal.

Other common elements of an FPGA:

Input/Output Blocks These physical pins/pads serve as the communication interface between the logic structure and peripherals devices, such as digital signal processors (DSPs), analog-to-digital converters, USB and Ethernet ports or other FPGA units. Usually there are special input pads for clock channels. The clock is provided by external oscillators with the frequency of the designers choice. Within the FPGA, these clock channels are routed on special paths that minimize the signal delay and improve the timing of the designs.

Memory Complex FPGA applications often ask for a fast and sufficiently large memory access. While external memory blocks can be easily accessed via the IO-ports, vendors have moved to integrated versions of SRAM of up to 36 KB[7].

Arithmetic Logic Units (ALU) For digital signal processing tasks FPGAs sometimes offer units within the logic arrays that can multiply two binary numbers very fast and efficiently. This is useful in instrumentation and control systems where measured data needs to be analyzed on the fly.

Hard-Cores Many system on a chip (SoC) applications, such as cell phones or signal processing hardware, demand for recurring features such as micro-controllers or Ethernet interfaces. Hard-cores are hardwired components within the FPGA that are optimized for certain functionality and thus require less logic gates. They usually offer higher speed with the cost of less flexibility. Each vendor offers its unique set of hard-cores with the most common ones being communication interfaces (Wireless, Bluetooth, Ethernet, USB), memory interfaces for industry standard blocks, decoding/encoding units or signal processing cores (Fast Fourier Transformations, digital filters of n-th order).

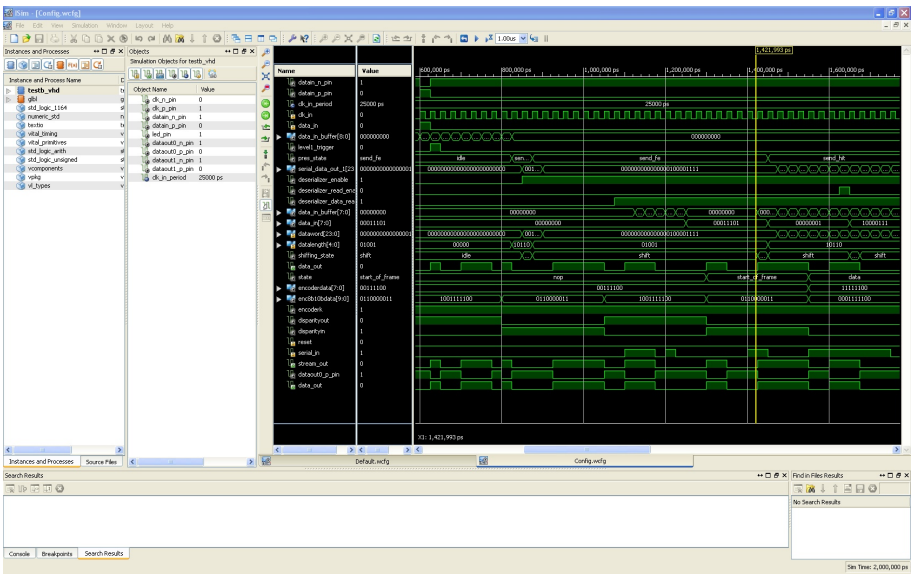


Figure 3.6: Screenshot of the Xilinx ISim Simulation tool with an overview on the existing signals (left) in the design and the time response of the observed signals (right).

3.1.3 Development Tools

Every FPGA vendor is responsible to supply the design tools for their logic circuits. Different FPGA architectures demand for different synthesizing algorithms that convert the HDL code into a real softwired circuit. This results in a variety of hardware dependent software development tools that cannot be unified like in higher-abstraction language development. In this thesis, the Xilinx ISE 9-11 IDE[3] and the Altera Quartus 8 IDE were used respectively for Xilinx and Altera FPGAs. Both tools support the 3 major design languages VHDL (see section 3.2), verilog and AHDL.

The programs can combine different VHDL entities into one project and allow for various configurations, such as optimization level and power usage constraints. Another major task of those tools is to incorporate the normally abstract and hardware independent logic designs into the existing circuitry. A pin planning tool allows for the correct connections between the ports of the design's logic circuit and the physical output pads of the FPGA chip. Futhermore the user is notified if a design does not fit into the FPGA chip, or if certain implementations do not function on the desired hardware.

[3]Integrated development environment

Instead of directly programming the design to the hardware and testing it, the development software offer a variety of simulation tools. Each logic can be embedded in a *test bench* that can simulate a time varying input for the circuitry. The simulator (see figure 3.6) offers a visualization of the signal change over time due to a user specified input. Problems like corrupt signals or wrong interconnections can be spotted much easier then in hardware testing. The simulator is the most important debugging tool in the early development stage. Modern simulators even take time delays of the signal into account and offer a realistic remodeling of the actual hardware. They are one of the reasons why FPGAs are very popular in prototyping nowadays.

3.1.4 Advantages/Disadvantages and Applications

FPGAs are favored for a number of reasons. The development time of electronic systems is much lower then with comparable ASICs. Basic prototyping can be done by some simple wiring and programming instead of intensive chip design. After the FPGA has been shipped, the functionality can still be altered or enhanced by reprogramming the chip.

Synthesized components work in parallel which is often faster than sequential designs with equal functionality. The hard-cores provided by the FPGA vendors further increase the productivity of the development process.

On the other hand, if a product requires large quantities of chips, the use of ASICs is still less expensive and preferable. Unfortunately current FPGAs can only work with lower clock frequencies of up to 1.5 GHz (20-500 MHz are common) in comparison to digital ASICS with more than 3GHz clock speed. The logic density in these chips is also usually 10 times smaller then the one of ASICS and they require more power for the same functionality.

Nowadays, the applications of FPGAs include digital signal processing like in the ATLAS project (see chapter 2.4), automotive embedded systems, medical imaging, data encryption and mobile technology. They also find applications in any area or algorithm that can make use of the massive parallelism offered by their architecture. These areas include code breaking of passwords or wireless keys with brute-force methods and cryptographic algorithms that can be up to 50 times faster than x86 CPUs.[4]

[4]Pico E-14 (Virtex-4 FX60) vs. 2.16 GHz Intel Duo WPA-PSK decryption: http://openciphers.sourceforge.net/oc/wpa.php

3.2 The VHDL Design Language

VHDL[5][10][11] is one of two major languages that are used to define the behavior of embedded hardware such as FPGAs. It was originally developed by the US Department of Defense in order to document the functionality of the ASICs that supplier companies were including in equipment. The other common hardware description language is called Verilog and is equally used. The development approach is different than in structural or functional programming languages such as Java or C. While CPU-based higher-abstraction programming languages produce byte code, that represents a sequential list of commands that are executed by the underlying hardware, VHDL produces a large interconnection of logic blocks that is driven by a clock signal.

VHDL is based on "*entities*" which can be thought of as little black boxes with input and output ports[12]. The designer´s responsibility is to declare the properties of these ports like name or size. After doing so, one can continue to define its behavior by using almost any kind of structural statements like *if, else, for, while* and even functions. To achieve a certain amount of complexity, these entities can be connected in a modular way to achieve the desired design. The soft-wiring of the circuits allow for large logic and signal processing blocks that work in parallel. Many of these entities can receive and send signals simultaneously which needs to be considered in the design in order to prevent timing problems. In comparison to sequential designs, it is harder for programers to trace where data is at which time of the execution process. As a substitue for *function calls* in sequential languages, *enable signals* are intensely used to activate entities and to control the flow of the data in the circuit.

3.2.1 Basic VHDL Elements

The following list holds a description of the most common language elements that are used throughout the design:

entity The *entity* describes the interface of a design with other VHDL-blocks. It is comprised of a name and a number of ports with input or output attributes.

architecture The *architecture* holds the description of the functionality of the corresponding entity. It can be comprised of various processes.

[5] very high speed integrated circuit hardware description language

process A *process* is a digital mapping of input signal to output signal influenced by logical decisions. Processes can be of combinatorial or clocked nature. In the first approach, changes in the input signals instantly result in the change of the output signal. Processes that are clocked will be only executed on the edge of a clock signal, allowing for time sensitive applications.

signal *Signals* can be one or more binary states that can be set by logical decisions or processes.

The following example shows the implementation of a simple 4 input, 1 output multiplexer to demonstrate the most important characteristics of the VHDL language ([11], see p. 33):

```
entity MUX4x1 is
   port (       CLK:     in std_logic;
                         SELECT: in bit_vector(1 downto 0);
                         INPUT:  in bit_vector(3 downto 0);
                         RESET:  in std_logic;

                         OUTPUT: out std_logic
                         SELECTED_TWO: out std_logic;
                         );
end MUX4x1;

architecture BEHAVIOR of MUX4x1 is
begin
        MUXPROC:         process(CLK, SELECT, INPUT)
        begin
                if RESET = '1' then
                        OUTPUT <= '0';
                elsif rising_edge(CLK) then
                        case S is
                                when "00" => Y <= E(0);
                                when "01" => Y <= E(1);
                                when "10" => Y <= E(2);
                                when "11" => Y <= E(3);
                        end case;
                end if;
        end process MUXPROC;

        SELECTED_TWO <= '1' when SELECT = "10" else '0';
end BEHAVIOR;
```

This 4 input multiplexer routes one of the inputs to the single output line according to the state of the 2 select channels. The output can be also reset to zero by raising the reset line. The multiplexer's 4 input channels are combined under a common name in a *bit vector* called E. The vector can be accessed like an array in C via an indexing operation with parentheses (i). VHDL main data types are bits and vectors of bits instead of more complex data types such as integer, double or strings in high abstraction languages. A VHDL bit is called *std_logic* and a bit vector is *std_logic_vector*. To take into account, that different systems either work with MSB[6] first and others with LSB [7] first, you can define an array to be ascending with *std_logic_vector(0 to n)* or descending with *std_logic_vector(n downto 0)*. Data is being stored in so called *signals* instead of variables. Signals can be assigned with values by the "<=" operator. These assignment can be coupled with certain conditions. In the example above, the *SELECTED_TWO* signal will only be assigned '1' if the *SELECT* input is indeed 2 or in binary "10". In other cases, the signal will be '0'.

Another important aspect of the example is the fact, that *SELECTED_TWO* will change immediately if a change in *SELECT* occurs. The *OUTPUT* signal on the contrary will only be altered on the rising edge of the incoming clock because its alteration is embedded in a clock driven process called *MUXPROC*. Signal assignments within a process will come effective only after the process has been fully executed. This is due to the parallelism meaning that all assignments in the process will be synthesized to a circuit that represents the logic and change immediately. It is also the reason why signals are not called variables. VHDL offers the capability of variables in processes but their use is buggy, mostly limited to simulations and normally not recommended by the FPGA vendors.

The multiplexer example given above can now be instantiated in any other design in order to route multiple input signals through a single output line. A basic version of a multiplexer is used in the old eBOC implementation to route the incoming data lines from the pixel modules to the ROD. According to data speed settings, either all channels would be forwarded, or only chosen ones.

[6] most significant bit
[7] least significant bit

4 8b10b Encoding

Modern signal transmission systems, such as the ATLAS read-out chain, require a certain monitoring mechanism to give a high level of confidence that the transmitted data can be correctly reconstructed at the receiving end. The so called 8b10b code is one type of encoding for serial data transmission which has become an industry wide standard. Almost all common high speed data protocols such as PCI Express, SerialATA, DVI and HDMI, USB3 and Gigabit Ethernet make use of 8b10b encoding to monitor data transmissions.

The 8b10b (en)coding scheme is designed to fulfill certain requirements that are made for the data stream. First and foremost, the 8b10b code satisfies the DC balance requirement. This means that the absolute difference between the logic ones and zeros in a long stream of data bits stays within 2. This improves the precision of the system because it is easier for the communication partners to stay synchronized. To implement this requirement, the 8b10b code provides a special way to wrap 8 bit raw data blocks into 10 bit code words where the 10 bit code words are chosen to maximize the number of binary data transitions (change from 1 to 0 and reversely).

The "*running disparity*" (RD) plays an important part in the 8b10b encoding. It is the cumulative sum of the disparity of all previously transmitted data blocks, where the disparity P is defined as the difference between the 1's and 0's $P = N_1 - N_0$ in a fixed size transmission block. The RD starts with a value of -1. As soon as a 10 bit data word has been transmitted, its disparity is added to the running disparity. The main idea of the scheme is that the 8b10b code is a mapping of a 2^8 (256) space to a 2^{10} (1024) space that introduces an additional degree of freedom. Thus only those 10 bit bytes are chosen, that have a disparity of +2, 0 or -2. The coding now poses the constraint that the *running disparity* must be either -1 or +1. This implies that a code word with a disparity of +2 can only be sent if the RD is -1 and a code word with a disparity of -2 will be only sent if the RD is +1. Code words with zero disparity can be sent regardless of the current RD.

The runnding disparity is an important monitoring tool for the receiver. If any disparity values for the received 10 bit values occur that are not conform with the 8b10b specifications, it is a strong indication for a transmission error. The RD is used in section 5.2 to detect errors in the data stream from the pixel modules to the rest of the read-out chain.

Besides the *DC balance* in the long run, 8b10b codes are chosen to ensure for no more than 4 consecutive 1s or 0s in the normal data stream. More binary transitions equal a higher signal frequency which is why this is good for data transmission through channels with a high-pass characteristic, such as Ethernet's transformer-coupled unshielded twisted pair cable or optical receivers using automatic gain control. In case of the ATLAS Pixel Detector, the photo diodes, that transmit the data from the modules to the off-detector hardware, require such a constant binary transision to work better.

For the encoding, the 8 bit byte is divided into a 3 bit and a 5 bit part which are then encoded into 4 bit and 6 bit sequences. Some of the 256 possible 8 bit words can be encoded in two different ways to make sure that the RD stays within +/- 1.

As an example for the code mapping, all possible 3 bit data values are listed in column *HGF* of table 4.1 together with their corresponding valid 4 bit codes in *fghj*.

input		RD = −1	RD = +1
	HGF	fghj	
D.x.0	000	1011	0100
D.x.1	001	1001	
D.x.2	010	0101	
D.x.3	011	1100	0011
D.x.4	100	1101	0010
D.x.5	101	1010	
D.x.6	110	0110	
D.x.P7	111	1110	0001
D.x.A7	111	0111	1000

Table 4.1: The 3b/4b encoding table.

The *data symbols* are labeled as D.x.y with x (from 0–31) representing the 5 bit part and y (from 0-7) representing the 3 bit part. In the convention, "A" and "a" refer to the least significant bit (LSB) and “H” and “j” refer the most significant bit (MSB). In data transmission, the 10 bit codes are sent with their least significant bit first: a → b → c → d → e → i → f → g → h → j in comparison to raw data

transmission in which the most significant bit is shifted out first. The naming of the 10 bit digits (*abcdei fghj*) is rather arbitrary and arose from the naming convention in the original US patent[13].

It can be seen in table 4.1 that there are 2 possible codes for data symbols with $y = 7$. The primary (D.x.P7) code is as the default code word. However it is possible to produce more then 4 consecutive 0's or 1's when combining D.x.P7 with its 6 bit counterpart. Therefore the alternate (D.x.A7) encoding must be chosen when $x = 17$, $x = 18$, and $x = 20$ at $RD = -1$ or when $x = 11$, $x = 13$ and $x = 14$ at $RD = +1$.

In all other cases, the D.x.A7 code is to be avoided because it can produce a 10 bit word that is either a *control symbol* (see section 4.1) or a data symbol that, in combination with other 10 bit symbols, might lead to a misaligned comma sequence in the data stream.

The paper on "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code" by X.Widmer et al describes a methods to implement 8b10b encoding via combinatorial circuits. In this thesis, an encoding unit from Laura Gonella, Bonn University and a decoding unit from *opencores.org* were used. Their implementations are close to circuits described in Widmer´s paper.

Generally 8b10b encoding decreases the effective data transmission speed. Since an 8 bit information is transmitted via a 10 bit word, the true data transmission speed S_{true} is

$$S_{true} = \frac{8}{10} S_{original}$$

IBL plans a maximum data transmission speed of 160 Mbit/s which would be a true speed of only 128 Mbit/s.

4.1 Control Symbols

Standards using the 8b10b (en)coding also define up to 12 special 10 bit symbols (or control characters) that do not have an equivalent 8 bit word and can be sent in place of a data symbol. They are used to indicate start-of-frame, end-of-frame, idle, skip or other link-related conditions. Since they do not contain data, they are mainly used to control the data flow. To distinguish them from normal data symbols, these control characters are referred to as K.x.y. Control symbols are usually sent less often than data symbols which is why they are allowed to have up to six consecutive 1's or 0's. The encoding unit, which is used in this thesis, has a special control signal

that can be used to choose whether the data input is encoded into the corresponding data word (k = '0') or control character (k = '1').

Listing 4.1: Entity declaration of the 8b10b encoding unit from Laura Gonella.

```
component encode_8b10b is
    port(
                data_in : in std_logic_vector(7 downto 0) ;
                control_word : in std_logic ;

                data_out : out std_logic_vector(9 downto 0) ;
                disparity : out std_logic
            );
end component;
```

Within these control symbols, K.28.1, K.28.5, and K.28.7 usually take a special meaning as "comma symbols". These are used for synchronization in the data stream[1]. If K.28.7 is not used in the protocol, the unique comma sequences 0011111 or 1100000 cannot be found at any bit position within any combination of normal codes and can be used to detect bit errors. A combination of multiple succeeding K.28.7 codes is not allowable because it would result in undetectable misaligned comma symbols. The K.28.7 is also often used because it is the only 10 bit word that cannot be the result of a single bit error in the data stream.

input		RD = -1	RD = +1
	HGF EDCBA	abcdei fghj	abcdei fghj
K.28.0	000 11100	001111 0100	110000 1011
K.28.1	**001 11100**	**001111 1001**	**110000 0110**
K.28.2	010 11100	001111 0101	110000 1010
K.28.3	011 11100	001111 0011	110000 1100
K.28.4	100 11100	001111 0010	110000 1101
K.28.5	**101 11100**	**001111 1010**	**110000 0101**
K.28.6	110 11100	001111 0110	110000 1001
K.28.7	**111 11100**	**001111 1000**	**110000 0111**
K.23.7	111 10111	111010 1000	000101 0111
K.27.7	111 11011	110110 1000	001001 0111
K.29.7	111 11101	101110 1000	010001 0111
K.30.7	111 11110	011110 1000	100001 0111

Table 4.2: List of control symbols.

[1]Detecting the alignment of the 8b/10b codes within the bit-stream.

The FE-I4 Data Output Protocol for IBL proposes the use of the 3 comma symbols as Start of Frame (K.28.7) and End of Frame(K.28.5) delimiters as well as Idle words (K.28.1)[14]. Each event, sent out by the FE-I4 chip, will be wrapped in with these Start of Frame (SOF) and End of Frame (EOF) words[15]. The encoding unit will also use the SOF word for a correct data alignment. The framing of the data with the SOF and EOF word introduces 20 bit of overhead for each frame. In comparison to the an event with 100 hits, which would be 2750 bits for the hit data, the 20 bit overhead of the framing is negligible.

5 Modifications of the Laboratory Read-Out System

5.1 The MCC-Module Emulator (FE-I3)

In section 2.2 the read-out chain in laboratory has been discussed. The goal of this thesis is to implement the 8b10b decoding mechanism on the eBOC. To be able to test the modifications of the eBOC (see section 5.2) a MCC module emulator is needed, that follows the FE-I3 specification but also implements 8b10b encoding. With this emulator the 8b10b decoding can be tested in the working FE-I3 environment of the laboratory (ROD Master DPS and read-out software). The following section describes the newly developed FE-I3 MCC emulator with its 8b10b encoding unit.

5.1.1 MCC to ROD Data Protocol

The MCC module emulator strictly follows the MCC-to-ROD data format that is specified in the ATLAS MCC-I2.1 Specifications[16]. The format is designed for a serial data communication and defines certain bit patterns that one needs to know to understand the data correctly. For this thesis, only the basic aspects of the event format will be explained. The emulator is designed to generate arbitrary pixel hit events but does not react on configurational requests.

The data format of the event generated by the MCC has been defined by making certain considerations. As one can see in the two examples of figure 5.1 each event starts with a header (11101) which also acts as synchronization sequence for the read-out chain. The header is followed by a unique Level 1 ID (LVID), that marks which Level 1 Trigger the event belongs to. This ID has a maximum value of 15 before it is reset which is why it is only unique in a certain span of time. In spite of that the interval is enough for the read-out chain to distinguish the individual

events.

Events on the MCC are ordered and sent out by their LV1 arrival time. Every event is completely transmitted before the following event will be considered for transmission. Within an event, all the hits are grouped together, that belong to a Level 1 trigger. The Bunch Crossing ID follows the LV1ID and defines which clock cycle the event belongs to.

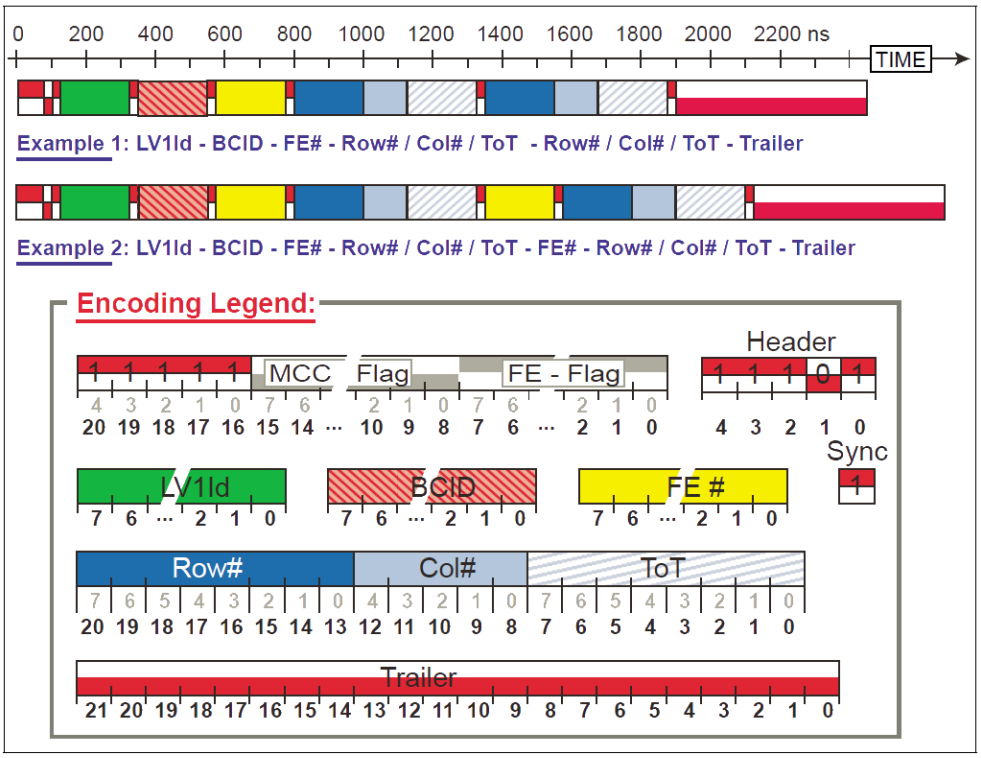


Figure 5.1: Event data format at the MCC output. Example 1 shows a one hit event. Example 2 demonstrates the format under multiple hits.[16]

Within the event, hits are sorted and grouped by FE chips. This permits data compression because for several hits on a single FE chip, the FE ID has to be sent only once. For the read-out chain, the full event length is not known until the whole event is transmitted. There is no event description header that informs the chain about its length. Thus, event hit data must be parsed one after another and the end of an event must be recognized by the content of the data.

During the implementation of the MCC specification, data compression was considered to be important. Due to the physical limitations, the number of transmission cables need to be kept as small as possible and still transmit the data to the off-detector hardware as fast as possible. The protocol is chosen to transmit only as

much bits as needed to communicate the important experimental data. Pixel hits are comprised of 4 or 3 components. They usually start with a 8 bit long FE-ID although the FE-ID can only range from 0 to 15. The first four bits of the FE-ID sequence (1110) again act as a synchronization pattern and do not contain data. After the FE-ID, the pixel position is transmitted as a pair of row and column values followed by the measured ToT value. The maximum number of hits allowed per FE chip is 112 due to the fact that the MCC has 128 word FIFO's inside each Receiver and 16 words are reserved for End-of-Event (EoE) words.

Another aspect in the MCC specification is the recovery from transmission errors. Any error in the event data should not influence the transmission of the subsequent events. That is why a unique Trailer has been chosen to terminate an event and allow for the distinction of succeeding events. It consists of 22 zeros along with a starting Sync bit (bit set to '1') which is a combination cannot be found in the event data. The Trailer is used in section 5.3 to extract single events from the ROD FIFO.

All data parts of an event with multiple hits can be found in table 5.1.

Keyword	*Bit Size*	*Low Value*	*High Value*	*Description*
LVID	8	0000 0000	1111 1111	First 4 bits:skipped Level 1 Triggers (0-15). Second 4 bits: Level 1 Trigger ID (0-15)
BCID	8	0000 0000	1111 1111	Bunch Crossing ID: Ranges from 0-255. MCC increments BCID every 40MHz clock cycle. BCID will be reset upon BCID data signal.
MCC-FE#	8	1110 0000	1110 1111	FE module number ranging from 0-15.
ROW#	8	0000 0000	1101 1111	Row number from 0 - 239[1].
COL#	5	0 0000	1 0111	Column number from 0 - 23.
TOT	8	0000 0000	1111 1111	Time over Threshold with 256 distinctive values.
Trailer	22	1 00 0000 0000 0000 0000 0000		Marks the end of an event.

Table 5.1: Length and possible values of the keywords in the event syntax used by the MCC.[16]

5.1.2 MCC to ROD Physical Layer Protocol

The MCC to ROD (via BOC) interface consists of 4 different channels. One channel distributes the *clock* (*CLK*) from the BOC to the module. Another channel is called the *Data In (DTI)* channel and transmits trigger and configuration data from the ROD to the module. The MCC can respond on two different lines called *Data Out 0* and *Data Out 1 (DTO0* and *DTO1*). Different data speeds are available and influence which channel will be used in which way.

In 40Mbit/s mode new data is being sent on both channels simultaneously on the rising edge of the 40MHz clock signal. Furthermore there are two possible 80 Mbit/s modes. One that transmits the data in 40 MHz cycles in which 2 bits, bit 1 on DTO0 and bit 2 on DTO1, are being transmitted on both lines simultaneously. The other methods sends data on one line at both the rising and the falling edge of the clock. This will also give a 80Mbit/s data speed. The same can be done on both lines with different data resulting in 160Mbit/s. For testing purpose, only the 40Mbit/s mode was used in this thesis. All implementations should work with higher speed modes without problems. All transmission channels are realized as LVDS channels (see section 2.4.1).

5.1.3 Software Implementation of the MCC

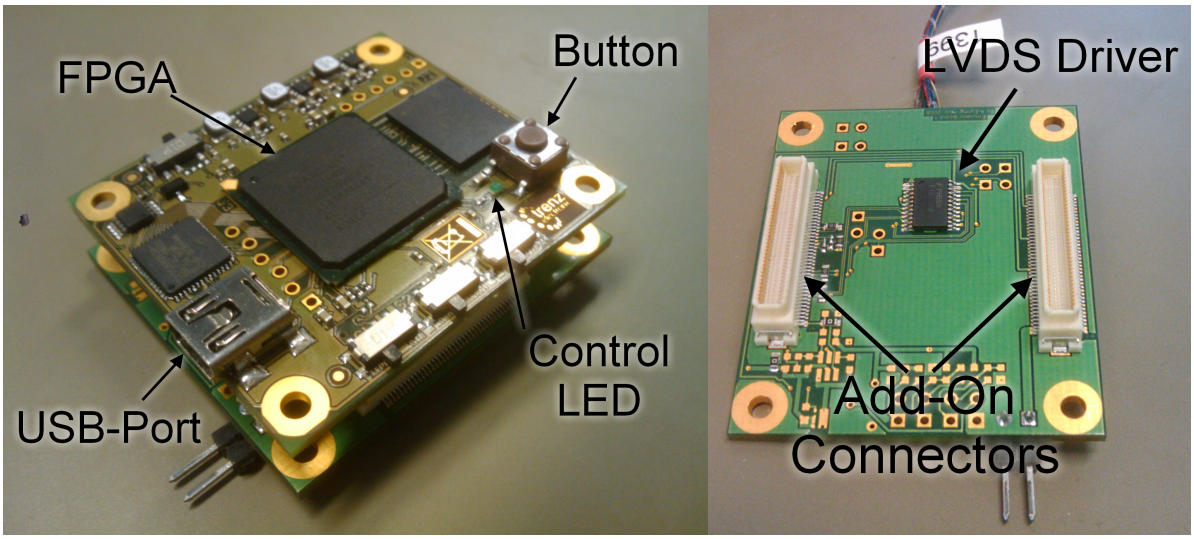


Figure 5.2: The Trenz Electronic TE-0300-01 Board with a Spartan-3E FPGA (left) and the Add-On Board on the Back (right). The Add-On Board contains a LVDS driver that is clearly visible.

In order to emulate the behavior of the MCC chip, a FPGA development board from Trenz Electronic with a Xilinx Spartan-3E was equipped with an Add-On board[2] that could be connected via a Type 0 cable to the rest of the read out chain. The Type 0 cable is a special ATLAS design. A picture of the FPGA board and the Add-On board can be seen in figure 5.2. Xilinx provides the ISE development environment for their FPGA families (see section 3.1.3) that was used to write the VHDL code, route and synthesize the design and program it to the FPGA.

The following designs make extensive use of two very common structures in digital computing: the FIFO and the (De)-Serializer.

FIFO A FIFO[3] describes a data storage method in which the element is read out first, that has been first written to the unit (see figure 5.3). It can be compared to a queue in real life where a person is served-first, that comes-first and the next has to wait until the first is finished. The opposite of this queue is the LIFO that is also known as a stack. FIFOs are heavily used in devices such as computer mouses or keyboards where those events are prioritized that are triggered first.

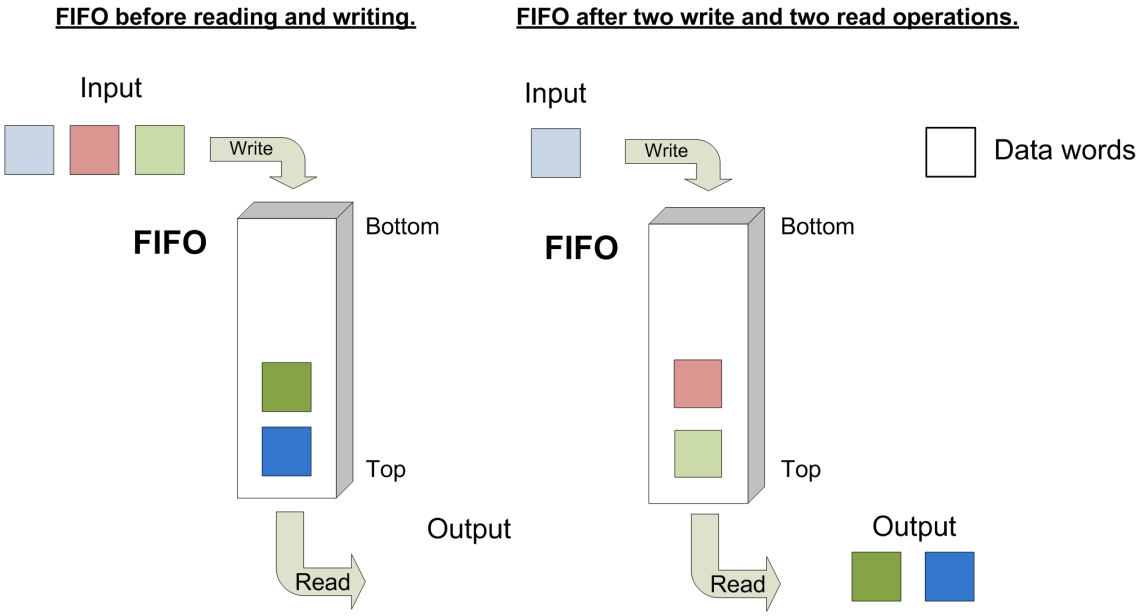


Figure 5.3: Scheme of the basic functionality of a FIFO before and after 2 write and read operations. Each colored block represents a data word.

FIFOs are characterized by the data width and the amount of data words they

[2]Provided by Bonn University (A.Eyring)[17]
[3]First In, First Out

can store. In comparison to random acess memory (RAM) blocks, the user does not have to think about the address where to store the data. On the other hand, it is impossible to access data members that are not at the beginning of the data queue in the memory. FIFOs are a good choice when it comes to storing sequential data gathered from a serial data stream.

The following listing contains the entity definition of an 8 bit FIFO used in the module emulator:

```
component Fifo8
        port (
        clk: IN std_logic;
        reset: IN std_logic;
        data_in: IN std_logic_VECTOR(7 downto 0);
        write_enable: IN std_logic;
        read_enable: IN std_logic;

        data_out: OUT std_logic_VECTOR(7 downto 0);
        is_full: OUT std_logic;
        is_empty: OUT std_logic);
end component;
```

The read and write speed of the FIFO is limited by clock speed used for the entity. Each clock cycle, data can be either written to the FIFO, read from the FIFO, or both. The data at *data_in* will be written to the internal memory when the *write_enable* signal is '1' at the rising edge of the clock (*clk).* In the same way, *data_out* will contain the next element of the FIFO every time the *read_enable* signal is high at the rising edge of the clock. The ability to read and write at the same time is helpful because reader and writer do not have to communicate about this issue. Two additional status lines signal if the FIFO is empty or full.

In this project FIFOs are mainly used to store incoming serial data for buffering and further processing. Since FIFOs are a very common way to store data, FPGA vendors deliver hard cores of these structures. Most FPGAs nowadays have additional SRAM space for this exact purpose. The saving of the data thus does not have be implemented with logic structures but can be facilitated with the SRAM blocks. If larger FIFOs are required, the use of external memory blocks is recommended. An interface has then to be provided in order to make it possible for the FIFO logic to store the data in the RAM. In the following designs, most of the FIFOs are used to buffer complete events. Depending on the event length, the size of this data can exceed over 3 kB (considering events with more than 1000 hits). The MCC emulator

FPGA can easily create FIFOs of such length but the Altera Flex FPGA on the eBOC can handle only up to 50 bytes. In the future the eBOC FPGA will have to be replaced by a more modern unit with integrated SRAM to allow for large event buffering.

(De)Serializer Deserializers and serializes are the interfaces between streams of bits and fully accessible data blocks. A serializer takes a given input data word - usually more than one bit - and shifts it out on a single data line. The frequency of this shifting process is determined by a clock. The serializer is always used, when a large amount of data has to be transmitted via a limited count of data lines. On the other end of the line, the deserializer reconverts the data stream into a full data word and stores it. For this process, the deserializer needs to know the transmission clock to receive the data bits correctly.

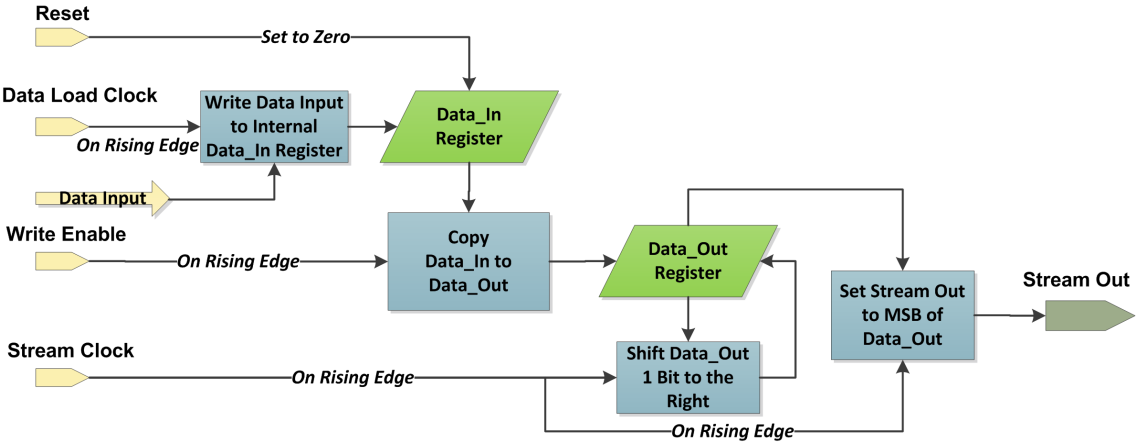


Figure 5.4: Basic concept of the internal behavior of a serializer.

Several problems occur when data is transmitted on a single line. The most important part is the data alignment. The receiver needs to know the start of a certain bit sequence, when deserializing it into data words with fixed length. The correct data alignment needs to be set by a communication protocol between sender and receiver and normally consists of a unique bit sequence that can only be found in this marker and not in the data. Encoding techniques such as the 8b10b encoding make this alignment easier.

Figure 5.4 shows the basic architecture of a serializer. Usually the data input (*Data In*) for the serializer is buffered inside the entity in a register (*Data_In Register*) whenever the *Data Load Clock* rises. This prevents the serialization to be

corrupted when the data input signal changes. Independently from the load signal, the *Write Enable* signal starts the serialization process by copying the *Data_In Register* to the internal *Data_Out Register*. This register will be shifted 1 bit to the right, each time the *Stream Clock* has a rising edge and the serializer's output signal will be set to the MSB of *Data_Out*. A more complex implementation of the serializer used in this thesis does not contain a single data register but a complete FIFO. This allows the user to feed several data words into the serializer before starting the streaming.

5.1.4 The Final Design

The physical communication layer between the module and the ROD offers two output data lines that are both used in this design. For testing purposes, the *DTO0* channel will contain the encoded data signal and the *DTO1* channel will have the raw data signal. This will allow to easily detect errors in the encoding system later on in section 5.3.

The MCC module emulator is a basic state machine that starts in an idle mode (see figure 5.5). Upon idle mode, the emulator will send a '0' signal on the raw data line and the NOP [4] K.28.1 idle word on the encoded data channel. The incoming command bit signal from the ROD is being shifted into a 5 bit register which activates a Level 1 Trigger signal, when it contains the LV1 sequence "11101" (*Level 1 Detector*). In that case, an activation signal is sent to the *FE-I3 Event Emulator*[5] which then produces a raw data stream with a FE-I3 compliant event containing a configurable number of hits. The *Event Emulator* is capable of buffering multiple Level 1 Trigger which means that any trigger that arrives during event sending will still be processed after that event is finished. The entity declaration of the *Event Emulator* contains a couple of configuration channels as well as the output links and control signals:

Listing 5.1: Entity declaration of the pixel simulator.

```
COMPONENT event_emulator
    PORT(
        clk_in:                 IN std_logic;
        rst_n_in:               IN std_logic;
        config_in_1:            IN std_logic_vector(31 downto 0);
```

[4] No Operation
[5] originally written by Daniel Dobos (CERN) and Jens Dopke (Universität Wuppertal)

```
        config_in_2:            IN std_logic_vector(31 downto 0);
        level1_trigger:         IN std_logic;
        BCR:                    IN std_logic;
        ECR:                    IN std_logic;
        inlinks:                IN std_logic_vector(3 downto 0);
        outlinks:               OUT std_logic_vector(3 downto 0);
        sending_event:          OUT std_logic
        );
    END COMPONENT;
```

Configurations include the variation of the *number of hits per event*, the amounts of hits sent, if a Level 1 Trigger is received, and a couple of MCC and FE flags that have not been used in the design.

To signal the sending of a new event, the entity has been modified to offer a special notifier, the *sending_event* signal. It tells the *8b10b Frame Builder* to leave IDLE mode and to send a SOF word (K.28.7). At the same time, the *Deserializer* starts reading the *Fake Event Data Stream* provided by the *FE-I3 Event Emulator* and stores it into a FIFO. The original data format of the FE-I3 is not formatted into 8 bit packages. Thus the *Deserializer* simply cuts the arriving data stream into 8 bit parts. The first part of the event data stream might contain only zeros, because the simulator raises the *Sending Event Signal* some bits before it actually starts transmitting data. This makes sure that the ongoing data processors catch the complete event.

After sending the SOF word, the *Frame Builder* reads out the *Deserializer's FIFO*, encodes the data in the *8b10b Encoder* and streams out the encoded data via the *Serializer*. The 8 bit data packages saved by the *Deserializer* are smaller then the 10 bit data packages produced by the *8b10b Encoder*. Since the 8 bit raw data and the 10 bit encoded data words are both read in and out with a clock speed of 40 MHz, the *8b10b Frame Builder* can instantly (after the raise of the *Sending Event Signal*) begin to encode and serialize the incoming raw event data. This is because the data cannot be read out faster then it is written to the *Deserializer FIFO*.

As soon as the *Sending Event Signal* has gone to zero, the *Deserializer* finishes saving the last 8 bit sequence from the *Event Data Stream*. The *8b10b Frame Builder* reads out the *Deserializer FIFO* until it is empty and terminates the frame with an EOF (K.28.5) word. After that the *Emulator State Machine* returns to IDLE mode and sends NOP K.28.1 word with alternating disparity - waiting for the next Level 1 Trigger.

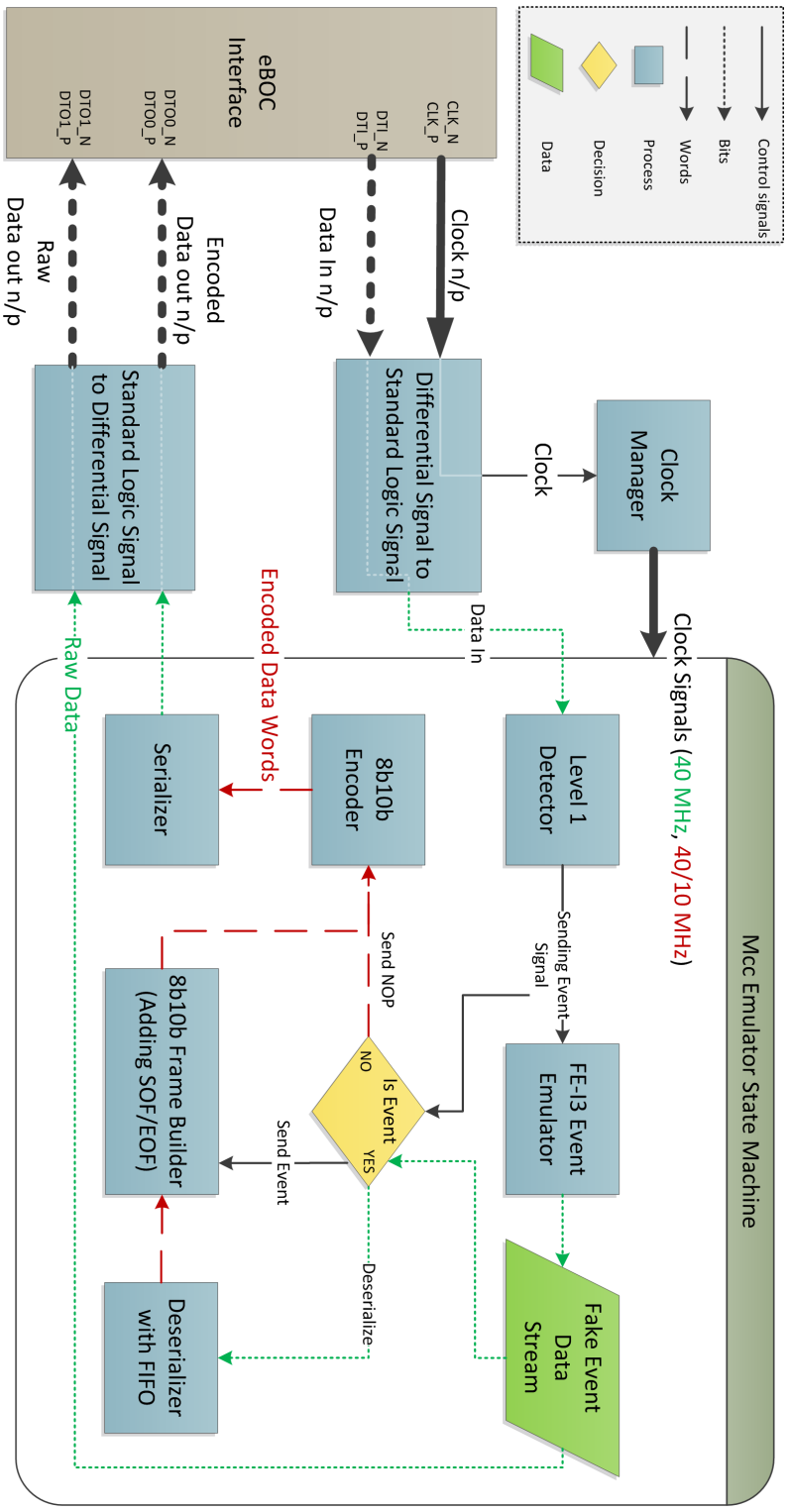


Figure 5.5: The schematic functionality of the FE-I3 module emulator unit.

5.2 Modifications of the eBOC

With the introduction of the 8b10b encoding scheme, the eBOC has to be designed to be able to decode the 8b10b signal from the modules, convert it into the original 8 bit data stream and send this off to the ROD. The 8b10b implementation of the FE-I3 module emulator is the same, that is used in the upcoming FE-I4 specification. The main problem with the encoding scheme is, that the 10 bit to 8 bit decoding introduces 2 redundant clock cycles. For every 10 bit word that is received and forwarded as a 8 bit word, at the same clock cycle, the 2 last bits after the 8 bit word has been shifted out do not contain data. The ROD on the other hand expects the data to arrive without any gaps. Consequently the event data has to be stored in a buffer (FIFO) before being sent out to the ROD.

The final design of the eBOC modifications can be found in figure 5.6. In its initial configuration the eBOC forwards the 32 FE Command Channels coming from the ROD to the Modules. In the laboratory test system, the eBOC also provides the 40 MHz clock signal for all the modules via an onboard oscillating crystal. Without the decoding unit, the eBOC receives the *32 Data In Channels* and simply forwards them to the ROD. For the decoding to take place, the *Encoded Data Stream* of the designated module is routed into the *Decoding Unit*. Each encoded module would need its separate encoding unit in the future. The incoming data stream is constantly shifted into a 10 bit register. For the correct data alignment, the stream is searched for the Start of Frame (SOF) word, marking the beginning of a new event. In this case, all following 10 bit words are being deserialized, decoded and stored in an *Event Data FIFO.* This FIFO currently is 8 bit wide and 32 words deep which is relativly small in comparison to the expected event length of far more than 100 words (for a 50 hit event). The limitation results from the small amount of logic gates in the Altera Flex FPGA which does not allow for a larger storage. The maximum possible event length with this setup is discussed in the validation section 5.3.

Parallel to storing the events to the FIFO, a counter in the *Stream Analyser* is marking the number of event words, that have been received. If the End of Frame (EOF) word has been received, the total *Event Length* will be written into a separate FIFO (*Event Length FIFO*). Furthermore the deserialization will be stopped. Whenever the *Event Length FIFO* contains data, the *Serializer* knows that a new event has been completely deserialized and decoded and is ready for sending to the

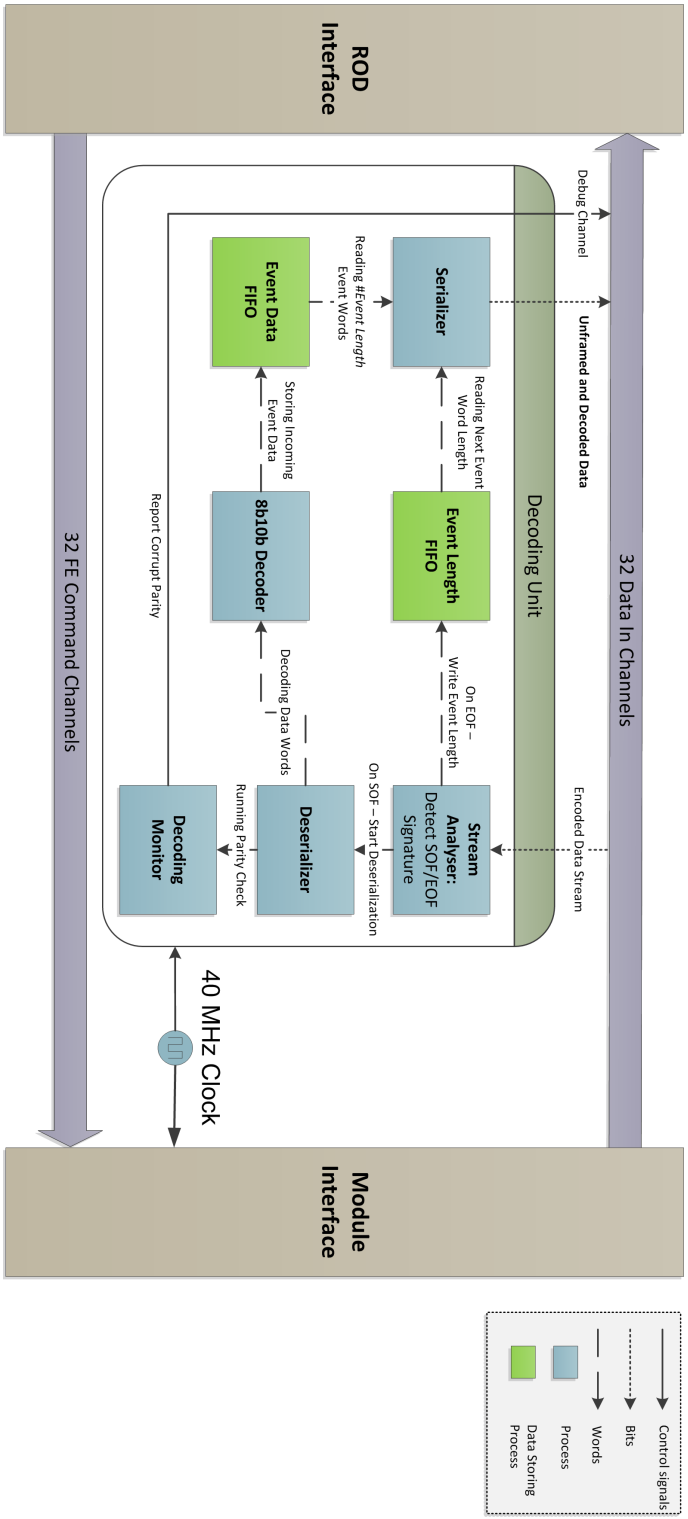


Figure 5.6: Schematic of the eBOC and the decoding unit.

ROD. The *Event Length* is read out and used to read the exact amount of data words from the *Event Data FIFO* one after another. These raw data words are fed back into the data stream for the ROD via the *Serializer.*

Since the Open-Source implementation of the *8b10b Decoder* in use does not contain data monitoring capabilities, the *Decoding Monitor* was implemented to watch the arriving 10 bit data words and checks for the correct *Running Diparity (RD)*. Whenever an error in the parity occurs, the *Debug Channel* will be raised for one clock cycle. The *Debug Channel* replaces one of the input links on the ROD and serves for data monitoring. This method was implemented for debugging reasons and will probably replaced in the new eBOC design with a more sophisticated data monitoring technique.

5.3 Validation

To validate the functioning of the encoding and decoding chain, an event error rate measurement has been performed. The debugging tool called "MccErrorChecker" has been developed for this purpose. It is based on the "Single Events Generator", which is part of PixLib[6], but was rewritten for most parts to repeatedly send Level1 Triggers to the module and to read out the received data from the ROD FIFOs. The event data arrives on two sepparate channels, one with raw data and the other one with the encoded data.

After decoding, the content on these two data streams should be the same but slightly shifted. The program analyzes the FIFO contents and extracts single events by looking for sequences of 1's ending with more than 22 0's which is the length of the data Trailer. These isolated events are cross checked with their equivalent events on the other data channel. An event has been successfully transmitted when these event bit streams are exactly the same.

The event error rate can be defined as:

$$\epsilon_{error} = \frac{N_{errors}}{N_{events}}$$

with the total number of errors N_{errors} and the total number of transmitted events N_{events}. Several runs have been made to check the events. The results can be found in table 5.2.

[6]C++ interface library for the communication between a Linux based computer and the ROD.

Run #	N_{errors}	N_{events}	ϵ_{error} Upper Limit 90%CL
1	0	23900	$9.63 \cdot 10^{-5}$
2	0	27000	$8.53 \cdot 10^{-5}$
3	0	1260000	$1.83 \cdot 10^{-6}$

Table 5.2: Results of the event error rate measurement for 3 different runs with variable event count.

The upper limit of ϵ_{error} can be calculated with the Baysian method. With ϵ being the probability that an error occurs, the number of checked events n and the number of erors in these events k, one gets the following binomial distribution[2]:

$$\begin{aligned} p(\epsilon|n,k) &= \binom{n}{k} \cdot \epsilon^k \cdot (1-\epsilon)^{n-k} \\ &= \frac{n!}{k!(n-k)!} \epsilon^k \cdot (1-\epsilon)^{n-k}. \end{aligned}$$

The number of event errors in this test turned out to be 0, which means that $k = 0$. The probability is thus:

$$\begin{aligned} p(\epsilon|n,k) &= \frac{n!}{0! \cdot n!} \epsilon^0 \cdot (1-\epsilon)^n \\ &= (1-\epsilon)^n . \end{aligned}$$

After full normalization the probability density function can be written as:

$$\tilde{p}(\epsilon|n,k=0) = \frac{p(k=n|\epsilon,n) \cdot p_0(\epsilon)}{\int_0^1 \mathrm{d}\epsilon \cdot p(k=n|\epsilon,n) \cdot p_0(\epsilon)} = (n+1) \cdot (1-\epsilon)^n .$$

The factor $p_0(\epsilon)$ is the "a priori" probability, which contains the previous knowledge about the distribution[18]. In this case it is not nescessary to include such an information which is why the value is constant and does not influence the normalization integral.

The 90% upper limit on ϵ can be calculated as

$$\int_0^{\epsilon_{error}} \mathrm{d}\epsilon' \cdot \tilde{p}(\epsilon'|n,k=0) = -\frac{n+1}{n+1} \cdot (1-\epsilon')^{n+1} \,|_0^{\epsilon_{error}} = 1 - (1-\epsilon_{error})^{n+1} = 0.9$$

Solving for the event error rate upper limit, the result is

$$\epsilon_{error} = 1 - \sqrt[n+1]{0.1}$$

The resulting upper limit of the event error rate ϵ_{error} in table 5.2 represents the probability with which an error occurs at a confidence level of 90%. The measurements show that the 8b10b decoding system seems to work quite well and does not introduce transmission errors, even in over one million events. The calculated upper limits correspond to this observation with values down to 10^{-6}. If runs with more events would been made, this number could be decreased even further. Thus as a result, errors in the encoding mechanism are most unlikely which is a good validation of the design.

During the test runs, event errors occasionally occurred due to loose contacts which is why several test runs have been made. Those corrupt measurements were not counted in. After fixing these problem, the system ran stable over more than 3 days. However, for longer runs, one should remove the programming cable of the eBOC because voltage variability in the cable could lead to an unintentional reset of the eBOC FPGA.

5.3.1 Realistic Event Triggering

To test the system under realistic conditions, a more sophisticated testing scheme has been developed to model the random nature of the experiment. Instead of constantly firing triggers with the same time difference in between, the triggering time differences follow a Poisson distribution:

$$P(\Delta t; \tau) = \frac{1}{\tau} \exp\left(-\Delta t/\tau\right)$$

where Δt is the difference between two trigger events and τ is the average trigger time difference. The distribution is already normalized. The minimum trigger time difference Δt_{min} is given by a test program[7] that cannot be faster then 200 ms between events. This time is mainly limited by the time needed to read out the FIFOs on the ROD and the interface between the read-out computer and the ROD. To determine the average trigger time difference τ we say that we want to be able

[7] *MccRawViewer* with built-in time measurements.

to trigger at least 99% with this value, meaning that:

$$\begin{aligned} \int_{\Delta t_{min}}^{\infty} P(\Delta t)\mathrm{d}\Delta t &= 0.99 \\ &= \exp\left(-\Delta t_{min}/\tau\right) \end{aligned}$$

The correct value for τ would be:

$$\begin{aligned} \tau &= -\frac{\Delta t_{min}}{\ln\left(0.99\right)} \\ &= 19899.8\,\mathrm{ms} \\ &\approx 20\,\mathrm{s} \end{aligned}$$

This average trigger time difference τ is a parameter of the minimal time Δt_{min}, the ROD FIFOs can be read out, and thus is only valid for the laboratory test system. Still the poisson shaped distribution of these times can be compared to the real ATLAS trigger time differences.

The common C++ random number generators usually cannot create values with a certain distribution but offer evenly distributed numbers on the interval $[0:1]$. The goal is to get Poisson distributed numbers from these evenly distributed ones. By using the *transformation method[19],* these even distributions can be transformed into any other probability density function as followed.

Consider a random variable X defined by the density $p_X(x)$. We can easily define a new random variable Y:

$$Y = f(X)$$

We are now interested in the density function $p_Y(y)$. Obviously the function f maps the interval $\mathrm{d}x$ to

$$\mathrm{d}y = \frac{\mathrm{d}f}{\mathrm{d}x}\mathrm{d}x$$

The probability to find X in $[x, x+\mathrm{d}x]$ is the same as to find Y in $[y, y+\mathrm{d}y]$, suppose the function f is unique. It can be written as

$$p_Y(y = f(x))\mathrm{d}y = p_X(x)\mathrm{d}x$$

or

$$p_Y(y = f(x)) = p_X(x) \left| \frac{\mathrm{d}f}{\mathrm{d}x} \right|^{-1} . \tag{5.1}$$

The initial simple distribution can be distributed by

$$p_X(x) = \begin{cases} 1 & for\ x\,\epsilon\,[0:1] \\ 0 & else \end{cases}$$

and our desired distribution is (with $y = \Delta t$)

$$p_{\Delta T}(\Delta t) = \frac{1}{\tau} \exp\left(-\Delta t/\tau\right)$$

To determine f, we use the fact that an inverse function f^{-1} exists:

$$x = f^{-1}(\Delta t)$$

No we can rewrite equation 5.1 with $\left| \frac{\mathrm{d}f}{\mathrm{d}x} \right|^{-1} = \left| \frac{\mathrm{d}f^{-1}}{\mathrm{d}\Delta t} \right|$ as

$$p_{\Delta T}(\Delta t) = \left| \frac{\mathrm{d}f^{-1}}{\mathrm{d}\Delta t} \right| p_X(x = f^{-1}(\Delta t)).$$

It follows that

$$\frac{\mathrm{d}f^{-1}}{\mathrm{d}\Delta t} = \frac{1}{\tau} \exp\left(-\Delta t/\tau\right)$$

and after integrating we arrive at

$$f^{-1}(\Delta t) = \exp(-\Delta t/\tau).$$

After taking the inverse of the function, the result is

$$\Delta t = f(x) = -\tau \ln\left(x\right).$$

If we are interested in Poisson distributed random numbers, we simply take evenly distributed numbers from the interval $0 < x \leq 1$ and put them into $\Delta t = -\tau \ln\left(x\right)$ to get a new time difference between to events that represents a realistic time interval. This algorithm does only work if the inverse function f^{-1} is a simple analytical function. If this is not the case, one should use rejection sampling[19] which was developed by John von Neumann.

In the test run, 22953 events with trigger delay times were generated and tested. The delay times n_i were histogrammed and normalized with the number of events N_{events}:

$$\begin{aligned} p_i &= \frac{n_i}{N_{events}} \\ \Delta p_i &= \sqrt{\left(\frac{\partial n_i}{N_{events}}\right)^2 \Delta n_i} = \frac{1}{N}\Delta n_i = \frac{\sqrt{n_i}}{N_{events}} \end{aligned}$$

To compare the normalized histogram with the initial distribution one has to consider that

$$p_{i,theoretical} = \int_{\Delta t_i}^{\Delta t_i + \Delta t_{bin}} \frac{1}{\tau} \exp\left(-\Delta t/\tau\right) \approx \Delta t_{bin} \cdot \frac{1}{\tau} \exp\left(-\Delta t/\tau\right)$$

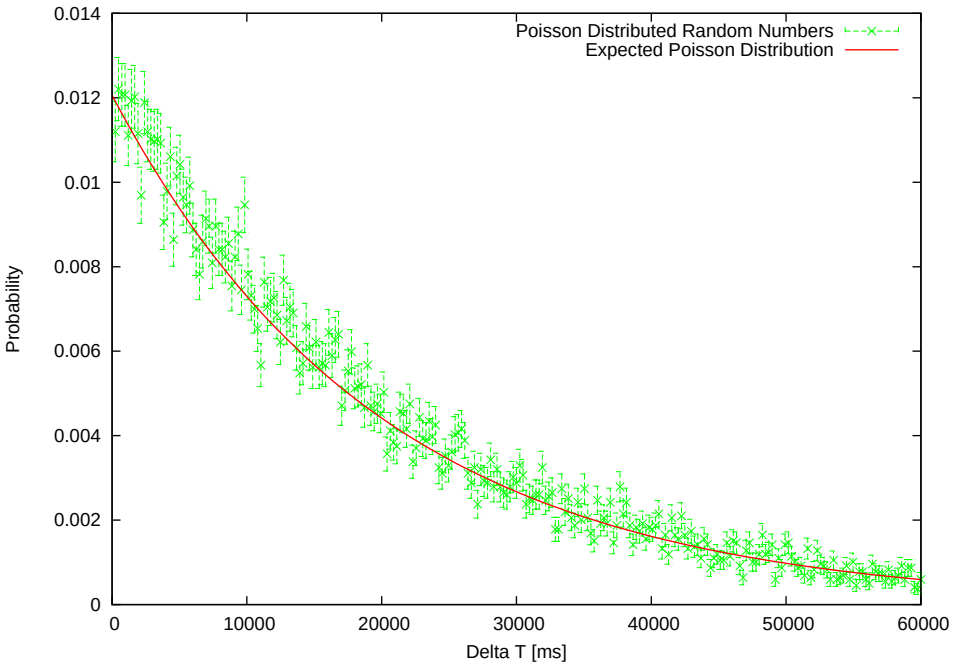


Figure 5.7: Plot of randomly chosen trigger spans according to the Poisson distribution. This plot was created with a 250 bins and 22953 random numbers.

The current run was histogrammed with 250 bins (approximately $N_{events}/100$) and a range of 0 to 60000 ms which resulted in $\Delta t_{bin} = 240\,\text{ms}$. The result can be found in figure 5.7. Almost all bin heights coincide with the expected poisson distributed probability (within the error Δp_i).

The random trigger event checker was implemented in another modification of the SingleEventGenerator called *MccErrorChecker_Poisson*. In the run, no error ocurred, resulting in an event error rate ϵ_{error} of $1.00 \cdot 10^{-4}$. The poisson distributed triggering test showed that the system is ready to work under real conditions without problems.

5.3.2 Data Delay

In the last part, the data delay that is being introduced due to the encoding and decoding process is evaluated and quantified. To determine the delay, one has to consider that an event is fully buffered inside the eBOC before it is forwarded to the ROD. The module emulator introduces a 2 bit delay per event word, due to the encoding and streaming of the 10 bit words. On the decoder side the encoded word is decoded and stored into an 8 bit FIFO. This introduces another 8 bit delay resulting in a total of 10 bit delay per 8 bit event word.

In theory, the total bit difference between the raw data and the encoded data would be

$$\Delta bit = 10\,\text{bit} \cdot N_{Event\,Words} + 10\,\text{bit} + const.$$

with the additioanl 10 bit overhead by the SOF word and the constant offset factor that is produced by the way, the MCC module emulator encodes the raw data via FIFOs. To verify that the bit delay increases with a linear factor and to determine the maximum possible event length (~ hit count) with the current design, the emulator has been equipped with the functionality to increase the number of hits per event each time the button of the development board is pressed (after 15 hits, the number is reset to 0). For each hit count, 2000 events were triggered, received and the bit difference between the raw data and the decoded data in the ROD was measured.

A first result of the measurement is, that a maximum of 9 hits per event for a fixed FE-chip ID can be transmitted so far with the current setting. This limitation arises from the limited storage space in the eBOC FIFO of only 32 byte (see section 5.2) that stores the incomming event data. The expected maximum event length

thus would be

$$\begin{aligned} N_{max,expected} &= Floor\left(\frac{S_{FIFO} - S_{Head}}{BPH}\right) \\ &= 10\,\text{events} \end{aligned}$$

with the size of the FIFO $S_{FIFO} = 32\,\text{byte}$, the size of the header/LV1D/BID/FE-ID + sync bits (32 bit) $S_{Head} = 4\,\text{byte}$ and the bytes per hit (Row#, Col#, ToT# = 22 bit) $BPH = 2.75\,\text{byte}$. As discussed in section 5.1.4 the encoding might catch some zeros before the actual event resulting in a slightly larger event header. This would decrease the expected maximum event length below 10 and makes it equal to the measured $N_{max} = 9$.

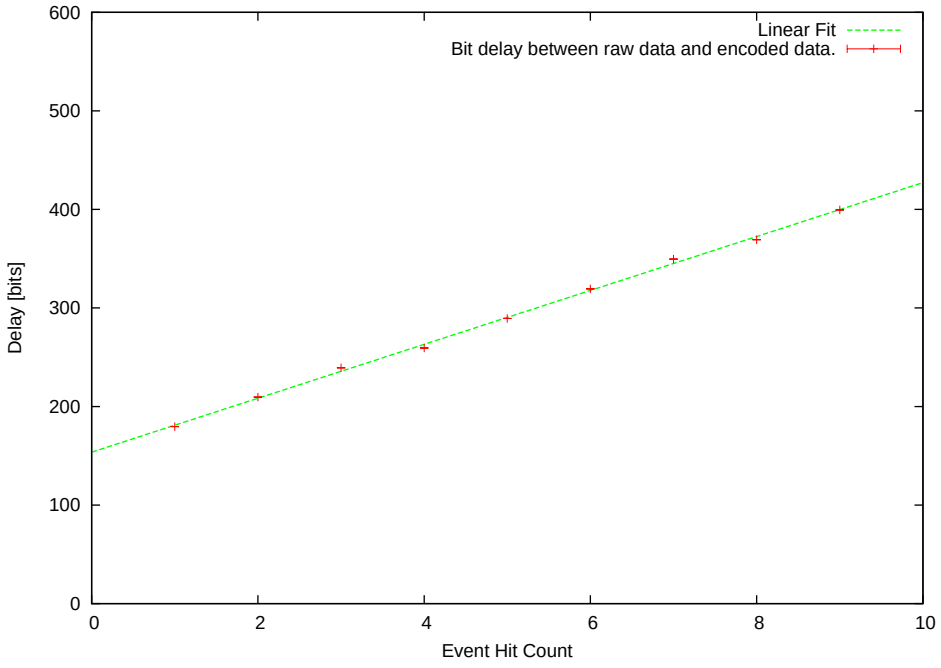


Figure 5.8: A plot of the measured bit delay for variable hit counts per event together with a linear fit of the data.

In figure 5.8 the averaged bit delays for the first 9 hit counts are plotted and fitted with a linear function. The standard deviation of these bit delays are not larger than 0.03% and therefore barely noticable in the plot. Given the $BPH = 2.75\,\text{byte}$ and the expected 10 bit delay for each event word, the theoretical bit delay increase per

event word is

$$m_{bit,expected} = 10\frac{\text{bit}}{\text{byte}} \cdot 2.75\frac{\text{byte}}{\text{hit}} = 27.5\,\frac{\text{bit}}{\text{hit}}$$

The slope of the linear regression in figure 5.8 m_{bit} is the measured average increase in the bit delay for an additional hit in the event:

$$m_{bit} = 27.3 \pm 0.3\,\frac{\text{bit}}{\text{hit}}$$

The expected bit delay for an additional hit and the measured bit delay are the same within the error thus confirming the theoretical expectations.

Given the average occupancy of 1.44 hits per B-layer module in the normal LHC experimentation phase, the bit delay would be approximately 40 bit which is negligible. For a calibration scan, events with up to 1440 hits per module can be generated. This would result in a bit delay of more than 39 kbit. Thus events would arrive 982μs later at the eBOC at a data transmission speed of 40MBit\s. This is still relatively short in comparison to the minimum trigger time difference of 200 ms but should be kept in mind in future designs of the scan and calibration software.

6 Summary and Outlook

6.1 Summary

In preparation for the Insertable B-Layer Project with the newly developed FE-I4 chip, the laboratory test system needed to be prepared for the 8b10b encoding method. For this purpose, a MCC module emulator with the specification of the FE-I3 chip was developed to facilitate the testing in the current laboratory test system which is working with the FE-I3 chip only. The emulator was realized with an FPGA development board and an additional Add-On for the interface between the device and the rest of the read-out chain. The design of the emulator was both tested in the simulation environment of the FPGA vendor and the actual read-out chain. In the end, the emulator is a fairly complex state machine that can generate both raw and 8b10b encoded event data. The system responds to a Level1 Trigger from the ROD , processes it and usess it as a start signal for event generation. The final design is a robust implementation of the emulator with the most basic features of a pixel module and a couple of configurable options like variable event length. In the future, parts of the code might be reused in a FE-I4 module emulator.

The eBOC modifications followed the implementation of the module emulator. After the emulator was fully tested with raw data, the eBOC was prepared for the decoding of 8b10b data packages. Unfortunately the Altera Flex FPGA posed some design limitation such as small event length. In the future these problems will be fixed in a complete redesign of the eBOC and the use of a modern FPGA unit with more logic blocks.

The 8b10 decoding unit was developed as a ”plug and play” module for the eBOC in which the unit would be put in between the incoming channel of the eBOC and the outgoing channel to the ROD. The most important features of the 8b10b encoding, like the bit alignment detection, the running disparity error monitoring and the event buffering have been successfully implemented.

Various tests were made to ensure the functionality of the system under realistic

and fast trigger conditions. No errors were detected in over 1 million events which validates the quality of the 8b10b decoding unit on the eBOC. In the end of the validation process, the delay introduced by the decoding system was quantified by various measurements with different event sizes. The resulted delay equates to the theoretical delay that has been derived from the hardware design itself and is considered to be negligible.

6.2 Outlook

In the future, the 8b10b implementation for the eBOC might be migrated to FPGAs on the ROD because it would facilitate the storing of the events. If the eBOC keeps decoding the arriving data stream, event buffering needs to be considered in the redesign of the eBOC board. The buffering unit (e.g. a SRAM block) will have to have enough space to store at least one complete event. As a rough estimatation (maximum of 53.760 pixel per module, 3 bytes per pixel) a single module would require at least 162 kB SRAM space if all pixel would be read out in a sole event.

As a preparation for IBL and the FE-I4 chip, the data rate of the module to eBOC communication still needs to be raised to 160 Mbit/s and tested.

Concerning the module emulator, the design could be improved in the future to react on configurational requests to make the emulator more realistic. If the buffering capabilites of the eBOC are implemented, the event error rate measurements should be repeated for events with a much higher hit count. For that the emulator would need another modifaction. Also the content of these hits, such as tot value, could be varied in order to create streams with almost every possible bit sequence.

Bibliography

[1] J. Grosse-Knetter. Vertex Measurment at a Large Hardron Collider. Professorial Dissertation, 2008. BONN-IR-2008-04.

[2] M. George. Implementation of an Electrical Read Out System for Multi-Module Labratory Tests of the ATLAS Pixel Detector. 2009. II.Physik-Uni-Goe-Dipl-2009/03.

[3] M. Barbero, editor. *FE-I4 - The New ATLAS Pixel Chip for Upgraded LHC Luminosities*, volume ATL-UPGRADE-SLIDE-2009-319. Bonn University, 2010.

[4] Altera Corporation. *www.altera.com*, Quartus 8 Documentation.

[5] Institute of Electrical and Electronics Engineers. *www.ieee.org*.

[6] J. Rose S. Brown. Architecture of FPGAs and CPLDs: A Tutorial. page `http://www.eecg.toronto.edu/~jayar/pubs/brown/survey.pdf`, 2000.

[7] FPGA Solutions from XILINX. *www.xilinx.com*, Spartan 3E Datasheet, Spartan 3AN Datasheet, ISE 9-11 Documentation, ISim VHDL Design Simulator Documentation.

[8] I. Bolsens. FPGA, a history of interconnect. *CTO, Xilinx*, 2005.

[9] Montgomery Phister. *Logical design of digital computers*. Wiley, 1958.

[10] P. J. Ashenden. *The Designer's Guide to VHDL*. M. Kaufmann, 2008.

[11] B. Shwarz J. Reichardt. *VHDL-Synthese*. Oldenbour, 2007.

[12] A. Maeder. *VHDL Kompakt*. Department of Computer Sciences, 2008.

[13] Y.-C. Wang A. Hsu. 8b/10b Encoder providing one of pair of noncomplementary, opposite disparity codes responsive to running disparity and selected commands. *United States Patent*, Feb. 21, 1992.

[14] FEI4 Data Output Protocol for IBL ver. 2. IBL internal proposal, 2009.

[15] P. A. Franaszek Al X. Widmer. A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code. *IBM Journal of Research and Development*, 27:440, 1983.

[16] R. Beccherle and G. Darbo. MCC-I2.1 Specifications. *MCC Design Group*, Data Format:`http://www.ge.infn.it/ATLAS/Electronics/MCC--I2/Spec--I2.1/Receiver.pdf`, 2004.

[17] A. Eyring. Module Emulator AddOnBoard Schematics.

[18] K. Kröninger. private communication.

[19] M. Mueller T. Pruschke, R. Kree. Einfuehrung in die Rechnerbedienung und Programmierung in den Naturwissenschaften. Script for lecture.

Acknowledgments

I would like to thank Prof. Dr. Arnulf Quadt and PD Dr. Jörn Grosse-Knetter for offering me this thesis and supervising me from the beginning of my study. A special thanks goes to Nina Krieger who repeatably helped me to debug the electrical read out system and find errors of various kinds. Also I would like to thank Mathias George for his help with various questions about the test system in general.

Another thanks goes to Julia Rieger, my office mate, who supported me mentally and motivated me in countless hours of debugging.

In the end I would like to thank all the members of the II. Institute of Physics for the enjoyable and friendly atmosphere and the very good time I had.

Erklärung nach §13(8) der Prüfungsordnung für den Bachelor-Studiengang Physik und den Master-Studiengang Physik an der Universität Göttingen:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und alle Stellen, die wörtlich oder sinngemäSS aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe.
Darüberhinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, im Rahmen einer nichtbestandenen Prüfung an dieser oder einer anderen Hochschule eingereicht wurde.

Göttingen, den July 18, 2010

(Benjamin von Ardenne)