SystemC simulation of the future SAMPA ASIC for use in the ALICE Experiment in Run 3

Consectetur adipisicing elit, sed do tempor incididunt ut labore et dolore magna aliqua

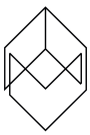
Håvard Rustad Olsen

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

June 2015



HØGSKOLEN
I BERGEN

Acknowledgements

Hvard Helstrup, Johan Alme, Dieter, Arild, Christian, (Damian).

Contents

List of Figures

List of Tables

Listings

Acronyms

ALICE A Large Ion Collider Experiment. 10–16, 20, 27

ALTRO ALTRO ASIC. 16, 18, 28, 31

ASIC Application Specific Integrated Circuits. 16, 18, 27–29

BT Binary Tree. 34

C++ A object-oriented programming language.. 20–22, 39

CERN European Organization for Nuclear Research. 10–12

CRU Common Readout Unit. 18, 27, 31, 36

FEC Front-End Card. 17, 18, 27, 28, 35

FIFO First-In-First-Out. 21–23, 28, 30, 35, 39

FPGA Field-Programmable Gate Array. 31

GBTx Giga Bit Transceiver. 18, 27, 28, 31, 36

GEM Gas Electron Multiplier. 17

LHC Large Hadron Collider. 10, 12, 13, 16

MWPC Multi Wire Proportional Chamber. 17

OOP Object-Oriented Programming. 39

Priority Queue Datastructure which sorts elements based on a priority(numerical value). 34

QCD Quantum Chromodynamics. 14

Random Number Generator Computational device designed to generate numbers that lack a pattern. 49, 52

RCU Readout Control Unit. 16, 18

SAMPA SAMPA ASIC. 17, 18, 27–31, 35–40, 42, 44, 48–50, 56

SystemC A simulation library building on C++. 20–23, 26, 27, 38, 39, 49, 55, 56

TeV Tera Electron Volt. 13

TPC Time Projection Chamber. 10, 13, 15, 17, 27, 28

Verilog A Hardware description language. 26

VHDL A Hardware description language. 26

Zero suppression Suppression schema/algorithm. 32–34, 36, 52, 54

Chapter 1

Introduction

This chapter will cover the motivation, as well as the scope and goal of this report.

1.1 Motivation

The Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN) is the world's largest particle accelerator, hosting multiple ongoing experiments. After a run period of more than 3 years, the LHC will be shut down from 2018 until 2021.[8] The purpose of this shutdown is to do maintenance on various equipment in the LHC, as well as significant upgrades to the different detectors, one of which is the detector for the A Large Ion Collider Experiment (ALICE). ALICE consists of multiple sub-detectors, which combined collect an enormous amount of data. This amount is expected to increase after the shutdown period as the interaction rate of the LHC will increase. Due to the increase in data output, the ALICE collaboration is seeking to upgrade and enhance the detector capabilities.[9] This includes a partial redesign of the readout electronics, upgrades to multiple sub-detectors and additional hardware upgrades.

The Time Projection Chamber (TPC) is the ALICE detector's main subdetector for tracking and identifying particles. A starting design for the new TPC readout electronics has been made, and the different components are currently being developed. As this is still being worked on, many questions about the different components are yet to be answered. Are the current specifications sufficient to handle the expected increase in output from the detector? Do they have the necessary bandwidth to be able to send the data with minimal sample loss. Are the buffer memory enough to handle the

traffic. Is it possible to optimize the current solution in any way?

The previous paragraph provides motivation for us to find a reliable way of determining a sufficient design for the readout electronics, while being both time and cost efficient. One strategy for solving this problem, which will be further explored in this thesis is creating a simulation of the system. Doing a simulation requires designing a accurate representation of the readout electronics, and creating a testbench where it is possible to configure and run multiple tests.

1.2 Research Question and thesis goal

Given the motivation and introduction given in section 1.1 the research question for this thesis becomes:

Is it possible to design and implement a simulation which directly represent the readout electronics, and in doing so will it have an optimizing effect?

Further explained, the main tasks of this thesis will be to create a computer model of the readout electronics main components, and run multiple simulations on it. Experimenting with different configurations in order to find bottlenecks, faulty design or areas of improvement. The experiments should be logged, and the results will be presented in an organized fashion.

1.3 Report structure

Chapter 2 will give the reader the background information to be able to understand the different academic and scientific terms used, as well as some information about the context of the report. This includes information about CERN, the ALICE experiment and the physics most relevant to the thesis. It will discuss the current readout electronics as well as the proposed upgrade. Chapter 3 is going further into the problem discussed in this report, initial plans on solving the problem, and information about the tools used. Chapter 4 will talk about the implementation of the simulation, what problems occurred along the way, and the chosen solution. The chapter will go into the design, as well as code snippets from the implementation. With the information given in chapter 4, chapter 5 will discuss the results of the different simulation runs, and evaluate the solution. Chapter 6 will conclude the thesis with some closing words, and work that can be done in the future.

Chapter 2

Background

This chapter will give the reader the background needed to set the rest of the thesis in context.

2.1 CERN

CERN is a European research and scientific organization based out of Geneva near the Franco-Swiss border[10]. CERN is a collaboration between 21 countries with a member staff of over 2500, and more than 12000 associates and apprentices. The organization was founded in 1954 and has since then been the birthplace of many major scientific discoveries. These are not limited to discoveries in the field of physics, but includes the creation of the World Wide Web[11]. Currently the biggest project at CERN is the LHC particle accelerator, which serves as the foundation for multiple experiments in the field of particle physics.

2.2 The Large Hadron Collider

Starting up on 10 September 2008, LHC is the latest construct added to CERN's particle accelerator complex[12]. It consist of a 27 kilometer underground ring of superconducting magnets which boost the energy of the particles travelling inside the collider. The collider contains two adjacent parallel high-energy particle beams. These beams consist of protons extracted from standard hydrogen atoms by stripping them of electrons. Along the collider ring there are four intersection points where collisions occur. Each point corresponds to the location of a particle detector - ATLAS, ALICE, CMS and LHCb. The particle detectors are each built and operated by large collaborations, with thousands of scientists from different institutes around

the world. The beams travel at close to the speed of light and are guided by magnetic fields, which are created and maintained by superconducting electromagnets. Superconducting meaning that it is in a state where it can most efficiently conduct electricity, without resistance or energy loss. Achieving this state requires cooling the magnets to -271.3° C , which is done by the distribution of liquid helium. The layout of the LHC ring as well as its four collision points can be seen in Figure 2.1.

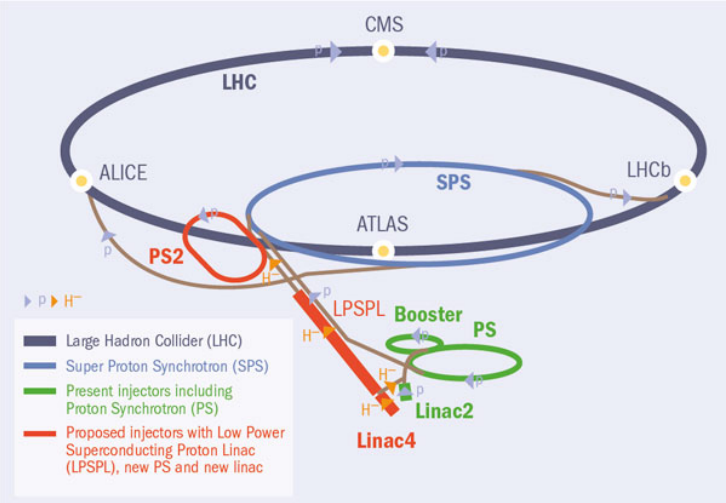


Figure 2.1: The Large Hadron Collider [1]

The beams travelling inside the LHC reach an energy-peak of 7 Tera Electron Volt (TeV), which means that on impact with each other the collision reach an energy of 14 TeV[13]. During a normal run of the collider there will be about 600 million particle collisions per second during a period of 10 hours. This leads to a huge amount of data for each of the detectors to read out. ALICE is the detector which produce the most data per collision, with a design value of about 1.25 GB/s written to permanent storage. The high amount of data per collision is produced primarily by the TPC sub-detector, which records a high number of points per track, and has a low momentum threshold. Detectors like ATLAS and CMS are designed with a higher momentum threshold, but can cope with significantly higher collision rates than ALICE. ALICE is designed for the study of heavy ion reactions, where particle correlations at low momentum is an important measure. The number of tracks correlating with momentum is exponentially declining. This means that a lot of tracks which doesn't get registered in ATLAS, produces data in ALICE.

2.3 ALICE

2.3.1 Introduction

ALICE is designed as a heavy-ion detector, which means it studies collisions between heavy nuclei of high energy[14]. The experiments is run with two different particle collision systems, lead-lead(Pb-Pb) and lead-proton(Pb-p) Both systems produce a extreme amount of temperature and density. They produce different, but equally interesting results. Pb-Pb collisions create Hot Nuclear Matter, while Pb-p create Cold Nuclear Matter. The explanations for these types of matter is beyond the scope of this thesis and will not be discussed further. The high temperature and density is necessary to produce a phase of matter called quark-gluon plasma.

2.3.2 Quark-gluon plasma

Shortly after the Big Bang, the universe was filled with a extremely hot cluster of all kinds of different particles moving around at near the speed of light[15]. Most of these particles were quarks, fundamental building blocks for matter, and gluons which ties quarks together in order to form heavier particles. Normally quarks and gluons are very strictly tied together, but in the conditions of extreme temperature and density as in the time shortly after the Big Bang, they are allowed to move freely in an extended volume called quark-gluon plasma. The existence of quark-gluon plasma and its properties is one of the key issues in Quantum Chromodynamics (QCD). The ALICE collaboration studies this, observing how it behaves.

2.3.3 The detector setup

The detector weight is about 10,000 ton, it is 26 m long, 16 m wide, and 16 m high[16]. It consists of 18 sub-detectors, each with its own set of tasks regarding tracking and identifying particles. This large number of sub-detectors are needed in order to get the full picture of the complex system which is being studied(i.e different types of particles and the correlations between them). Most of the detector is embedded in a magnetic field, created by a large solenoid magnet, which makes particles formed in collision bend according to their charge, and behave differently relative to their momentum. High momentum equals near straight lines while low momentum makes the particles move in spiral-like tracks. During lead to lead collisions the collision rate peaks at 8 kHz(Where Hz is defined as number of events per second). The number of recorded events is smaller in practice because the ALICE

detector uses a triggered readout, which only triggers on head-on(central) collisions. The maximum readout rate of the current ALICE detector is 500 Hz, which is more than enough to track central collisions. Figure 2.2 shows a cross section of the detector as it is today with the red solenoid magnet, and all sub-detectors labeled.

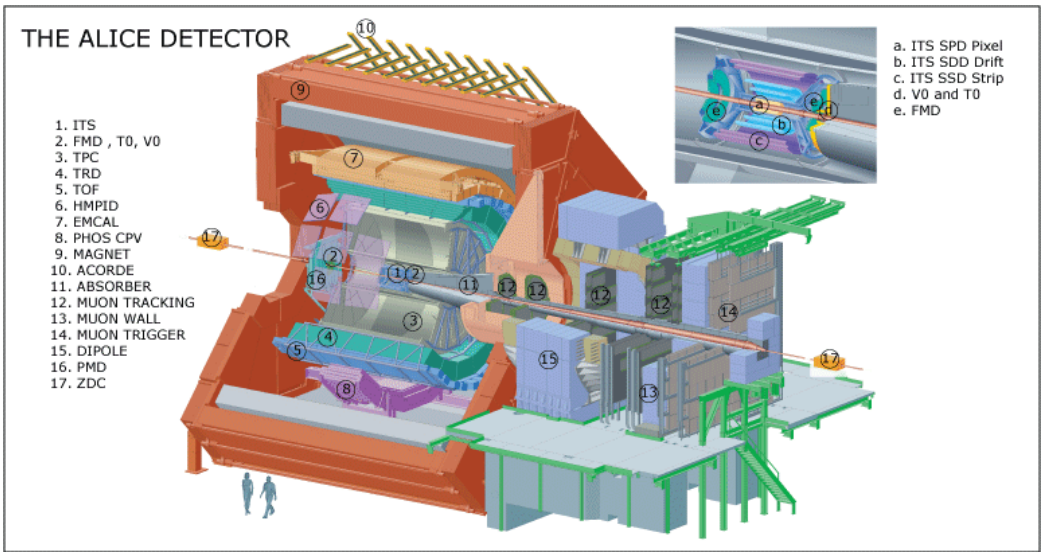


Figure 2.2: The ALICE detector [2]

2.4 The TPC detector

2.4.1 Intro

One of the most important sub-detectors, and the one that is relevant for this thesis is the TPC detector. Located at the center of the ALICE detector it is among the first entry points when gathering data from a particle collision. It is a $88m^3$ cylinder filled with gas. The gas works as a detection medium, which means that charged particles from a collision crossing will ionize the gas atoms, freeing electrons that move towards the end plates of the detector. The readout is done by specially designed readout chambers, which are capable of handling the high amount of data produced in heavy-ion collisions.

2.4.2 Readout electronics

Signals from the readout chambers are passed along to the front-end readout electronics, which today consist of 4356 ALTRO Application Specific Integrated Circuits (ASIC) chips[17]. ASIC is the term used for specially customized chips, rather than chips with a more general-purpose use[18]. The ALTRO chip is made up of 16 asynchronous channels that digitize, process and compress the analogue signals from the readout chambers. It operates on a so called triggered readout mode. In short when ALTRO receives the first trigger, it stores the following data stream into memory, holding on to it until it is ready to pass on the data. The front-end electronics are able to readout data at a speed of up to 300 MB/s.

The ALTRO chip sends the digitized signals further down the readout chain to the Readout Control Unit (RCU), where it is further processed and shipped to and stored in the online systems. The schematics is shown in Figure 2.3.

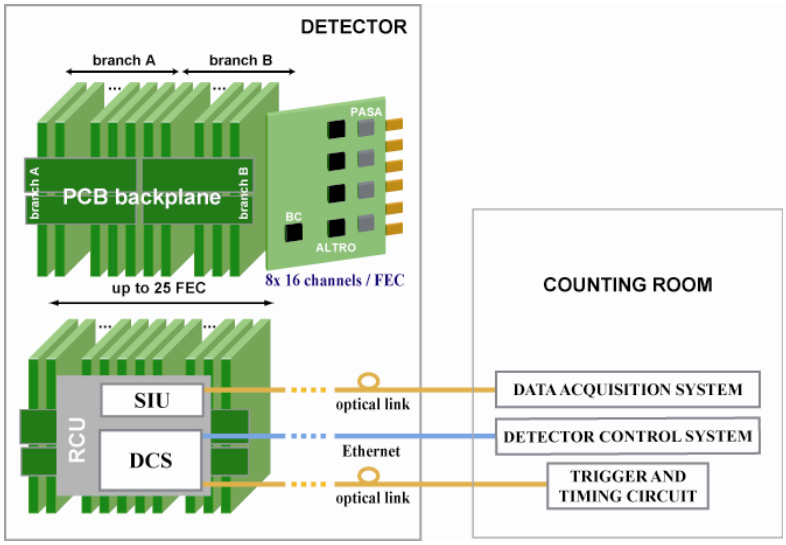


Figure 2.3: Readout schematics for the current TPC detector [3]

2.5 Long Shutdown 2

As mentioned in 1.1 the LHC ring will be shut down for about 3 years, starting 2018. During that time the ALICE detector will undergo an extensive upgrade. The upgrade strategy for ALICE is based on the expected

increase in collision rate to 50 kHz, and will now track every collision. Essentially this comes down to a increase by a factor of 100, compared to what is achievable today.

To be able to handle the increase in collision rate the TPC will receive upgrades to both its readout chambers, and front-end readout electronics. The current Multi Wire Proportional Chamber (MWPC) based read-out chambers will be replaced by Gas Electron Multiplier (GEM) detectors, which has a much higher readout rate capability. Signals will be passed from the new readout chambers to the Front-End Card (FEC) via a readout pad structure similar to the one presently used, where each pad is mapped to a SAMPA channel. There are multiple pad structures depending on its location on the detector, but the difference in structure is not relevant for this thesis. What is relevant however is that more data is expected from low pad numbers, an example of a pad structure is shown in Figure 2.4.

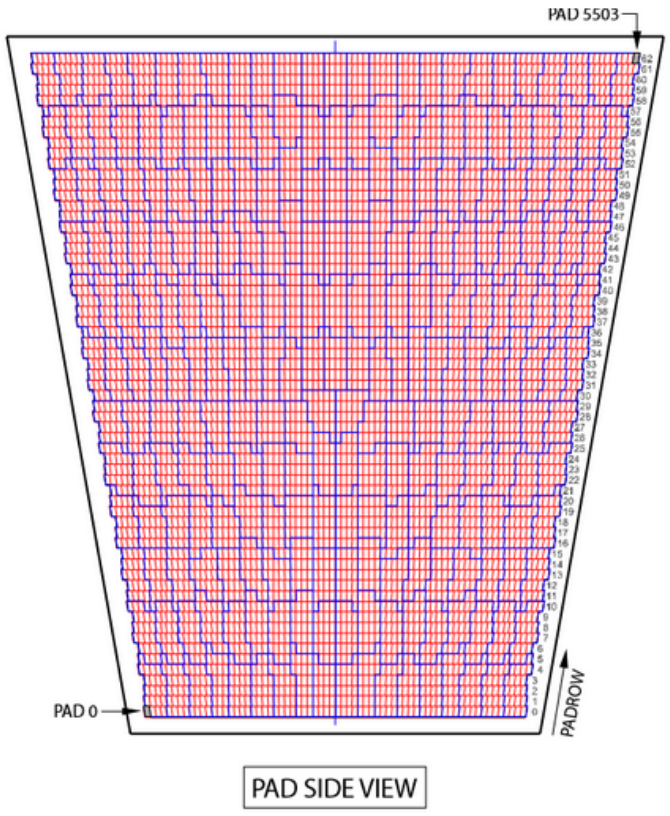


Figure 2.4: Pad structure of an Inner Readout Chamber(IROC)[4]

The entry point in the FEC is the new custom-made ASIC, the SAMPA, which will replace the ALTRO chip[5]. The SAMPA chip is capable of processing signals asynchronously in 32 individual channels, each channel is directly connected to a single pad. They are further on digitized and concurrently transferred to the Giga Bit Transceiver (GBTx), which enhances the signal strength and transmits them via multiple optical fiber links to the Common Readout Unit (CRU). The CRU can be thought of as the new RCU and serves as an interface to the online systems. The data flow from the detector, and a working schematics can be seen in Figure 2.5. Chapter 4 will go into more detail about the readout electronics in the context of our simulation.

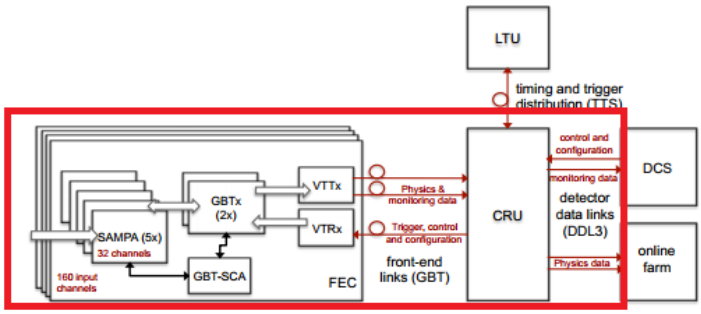


Figure 2.5: Schematics of the readout electronics [3]

Chapter 3

Simulations

SystemC, Starting design of the simulation, plans for implementation and test runs

3.1 Simulation Theory

3.1.1 Theory

A simulation can be seen as the imitation of a real-world system and its operations over time. This requires a model representation of the system which is accurate enough to conduct experiments on, which produce real-like results. The model should include key characteristics, specifications and functions of the selected system, but in a simplified fashion. A simulation model can take many forms as it can be used in different contexts ranging from physical object such as electrical circuits, bridges, and even entire cities to abstract systems like a mathematical equation or a scientific experiment[19].

As the model represent the system itself, the simulation represents its operations over a set period of time. The simulation is normally conducted in a controlled environment that makes it possible to observe, monitor and log results. To achieve efficient experiments using a simulation, it should be easy to change its parameters with respect to what is being tested.

There are many benefits of simulating a system instead of creating and test the real thing. A simulation will in most cases be very time efficient, you can conduct the same kinds of experiments on the system in a much shorter time compared to the real thing. This means that more information about the systems behavior and its limitations can be gathered in less time, which in

turn can result in a better final product. Creating the real-world system can often be very expensive, which may limit the amount of prototypes or test-products that are possible to create. Therefore using results of a simulation to fine tune the specifications before starting to produce prototypes will cut unnecessary development costs by a significant margin.

Taking the upgrade of the readout electronics for the ALICE detector as an example to further address this point one can see the usefulness of not having to create multiple custom hardware components, all with different purposed specification. In regards to the readout electronics, another important point is that the proposed designs might already function properly, but there is always room for improvement. Finding out that the design doesn't need as much memory, or less optic fiber cables can impact the overall production costs. One way to efficiently and accurately simulate hardware components is by creating a virtual computer simulation.

3.1.2 Computer Simulations

Using computers to do simulations becomes more and more useful because of their incredible computational power, and ability to produce fast results. This is important as simulations often become quite complex, both in regards to computational complexity and level of difficulty to understand and further work with. Therefore it can be wise to use existing tools to help make the process easier. There is an array of different tools that can be used to various kinds of simulations. They vary from complete frameworks, with graphical user interfaces to tools which help programmers write there own simulation programs. The later requires of course the most work, but will most often end with the better results as you can tailor your simulation on a lower level than with a complete framework. A programming tool that is made for creating simulations is the SystemC library, which will be discussed in the following section.

3.2 SystemC

Explain how SystemC works, what benefits and downsides

3.2.1 Background

SystemC is a system design library based on C++[20]. It provides an interface to easily create a software model that represents a hardware architecture,

and together with standard C++ development tools it is possible to quickly build a full scale simulation. Following the standards of C++, SystemC is built to be easy to understand for both software and hardware developers, resulting in clearer cooperation between them while developing the hardware design. The SystemC library provides an object-oriented approach to model design, where a single C++ class represents a model. This makes it easy to separate concerns between the different models in your simulation.

When simulating a hardware system there is a couple of key points to be aware of, firstly you need to be able to handle hardware timing, clock cycles, and synchronisation. One of the benefits of SystemC is that it takes care of all of this, again taking advantage of the object-oriented nature of C++ to extend its capabilities through `classes`. Here is some of the other features SystemC provides, with emphasis on the ones needed to understand code snippets shown in this thesis.

- **Modules**
  - Container `class` representing a hardware model.
- **Processes**
  - In short, processes are methods inside a module which describe the module functionality.
- **Ports**
  - Ports represent the input and output points of a module, they can be connected to other modules through Channels. When you declare a port in a simulation, it is required to specify if the port is an input, output or bidirectional port. This is done by specifying a channel interface for the port. Example of a port using a input First-In-First-Out (FIFO) interface:

    ```
    sc_port<sc_fifo_in_if>
    ```

    .
- **Channels**

  - Channels are the wires connecting two Ports. SystemC comes with three predefined channels: FIFO, mutex, and semaphore. It is possible to configure custom channels, but in most cases it is not necessary.

- **Signals**
  - Signals represent data sent between modules via ports. They can be arbitrary data types like `bool` or `int`, but also user defined types.

- Rich set of data types
  - SystemC supports all data types defined in C++ as well as multiple custom types.

- Clocks
  - SystemC comes with clocks, which can be seen as timekeepers of the system during a simulation.

3.2.2 Small example

To get a basic understanding of how a SystemC simulation looks like, it is useful to see it in action. The following Figure 3.1 and Listings 3.1-3.3 make up a very trivial example with only 2 modules; a `Producer` and a `Consumer`. The `Producer` will increase a counter every clock cycle, and send a `bool` value based if the count is an even number, and send this value to the `Consumer`, which registers how many times the `Producer` counted an even number. The example uses a FIFO channel, connected between an output port on the `Producer`, and an input port on the `Consumer`.

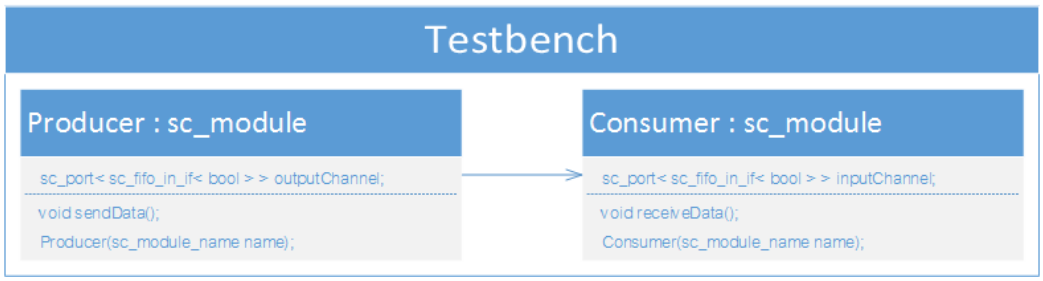


Figure 3.1: Basic SystemC example

```
1  SC_HAS_PROCESS(Producer); //macro to indicate that the module
       has process
2
3  //Constructor with name of module as parameter
4  Producer::Producer(sc_module_name name) : sc_module(name){
5    SC_THREAD(sendData); //Registrer the sendData thread
6  }
7
8  //Thread which runs until the simulation is over.
9  //Clock frequency: 100 Mhz; 1 / 10 ^ 7 = 10 nanoseconds
10 void Producer::sendData(){
11
12   bool signal = false; //signal value
13   int count = 0; // count variable
14
15   while(true){ //infinite loop
16
17     if(!(count % 2)){ // if count is even, signal = true
18       signal = true;
19     }
20
21     outputChannel->nb_write(signal); //write signal to output
           channel
22
23     signal = false; //reset signal
24     count++; //increase count
25     wait(10, SC_NS); //End of a clock cycle, wait 10 nanoseconds
26   }
27 }
```

Listing 3.1: Producer module.

```
1  SC_HAS_PROCESS(Consumer); //macro to indicate that the Module
       has 1 or more processes
2
3  //Constructor with name of module as parameter
4  Consumer::Consumer(sc_module_name name) : sc_module(name){
5    SC_THREAD(receiveData); //Registrer the receiveData thread
6
7  }
8
9  //Thread which runs until the simulation is over.
10 //Clock frequency: 100 Mhz; 1 / 10 ^ 7 = 10 nanoseconds.
11 void Consumer::receiveData(){
12
13   int numberOfEvens = 0; // counts number of evens
14   bool receivedSignal = false; //received signal variable
15
16   while(numberOfEvens < 10){ //stop lopp when received 10 evens
17
18     if(inputChannel->nb_read(receivedSignal)){ //receiving
           signal; nb_read returns true if signal is read.
19       if(receivedSignal){
20         numberOfEvens++; // if signal is true, count was even.
21       }
22     }
23     wait(10, SC_NS); //End of a clock cycle, wait 10 nanoseconds
24   }
25   sc_stop(); //Force stop simulation.
26 }
```

Listing 3.2: Consumer module.

```
1  int sc_main(int argc, char* argv[]) {
2
3    Producer producer("Producer");
4    Consumer consumer("Consumer");
5
6    sc_fifo<bool> channel(20); //(First-In-First-Out) channel with
         depth of 20.
7
8    //Connecting Producer-Consumer channel.
9    producer.outputChannel = channel;
10   consumer.inputChannel = channel;
11
12   sc_start(); //Alternative: sc_start(30, SC_NS) - Specified
        simulation lenght.
13
14   return 0;
15 }
```

Listing 3.3: Simulation test-bench.

SystemC can be used to create very low level hardware descriptions and models, and can interface directly with hardware description languages like VHDL and Verilog. This is one way to create a simulation, and the models will be very accurately represented by doing so. The other way is to have a high level of abstraction, leaving out the unimportant details and focus solely on the expected problem areas. There are benefits and drawbacks for both ways, but sticking to a high abstraction level can in complex cases make it a lot easier to work with the model design and allows you to focus on the important parts.

Chapter 4

Problem Description

Explain the model, introduce the problem

The previous chapters has briefly introduced the problems of this thesis, relevant background information and looked at tools and the method of solving them. Essentially it boils down to creating a model based on the schematic of the TPC readout electronics, run multiple simulations, testing different parameters for the involved components. Until now there has only been an introduction level description of the different components that is being included in the simulation model. This chapter will go deeper into them, giving detailed information about their design parameters, and how the ALICE experiment data is handled by them. Not going too far into the task of implementing this in a SystemC environment, but focus on the different problem areas, what is required in order to solve them and what goals to achieve.

4.1 Model Design

Different design patterns, and plans for the electronics

The hardware design which is being simulated is already briefly shown in Figure 2.5. The proposed schematic shown there consists of 12 FEC cards for every CRU. Each FEC consists of 5 SAMPA and 2 GBTx ASICs, with the CRU being connected to them via 24 optical links. Out of the 3 main chips, the SAMPA and the CRU are the most interesting as they are still being developed and testing them can give a lot of valuable feedback. The GBTx is a completed component, so even though it is part of readout electronic being simulated, it will only be a very shallow abstraction of it. This means

that it will remain as an empty module whose objective will be to just pass along received data to the correct output links. One important note about the GBTx input and output links. Each GBTx has 10 input e-links, each with a transfer rate of 320 Mbit/s, giving an effective input speed of 3.2 Gbit/s per GBTx. The output is 1 optical fiber link with a speed of 3.2 Gbit/s, giving the GBTx the same input and output speed. This is the reason letting data flow directly through the GBTx in the simulation is possible. The next sections will go into details about the more important components.

4.1.1 SAMPA

The SAMPA ASIC is based on the work from its predecessor, the ALTRO. Just like the ALTRO it will be the first step for signals being tracked in the TPC detector. The signals will be processed, compressed, digitized, and temporarily stored in the SAMPAs memory before it is passed along. The SAMPA has 32 integrated channels, which separately and asynchronously process the analog signals coming from the detector[3]. Each channel has a readout speed of 10 bit on a 10 MHz clock, which combined results in 3.2 Gbit/s. The channels also have their own FIFO buffer memory where signals coming in are stored as they wait to be sent along. The most efficient size for these buffers are one of the things the simulations will hopefully provide. The output links for the SAMPA chip consists of 4 e-links connecting them to the GBTx. Each e-link has as said in the previous section a speed of 320 Mbit/s, which sums up to 1.28 Gb/s[5]. The e-links are connected to 4 readout buffers on the SAMPA that reads from the channel buffers and transports the data to the e-links. The readout buffers reads from 8 channels each. Since each SAMPA and GBTx has a specific number of output and input links, there are only certain setups which are desirable. This is why the proposed schematic uses 5 SAMPA and 2 GBTx chips for each FEC. That setup gives exactly 20 output links from the SAMPA chips, and 20 input links on the GBTx chips.

As the ALTRO, the SAMPA can be run in triggered readout mode, but in addition it can be run continuously. Being able to read out continuously is a necessary upgrade to handle the increased data load coming from the detector. During continuous mode the data acquisition is uninterruptable, meaning that there is no pause between reading two consecutive events from the detector. The difference it makes compared to triggered mode can be seen in Figure 4.1. Every event, from now on referred to as time frames, is 1024 clock cycles long, and all 32 channels of the SAMPA use the same time frame. This means that every 1024 clock cycle a 1024 long time window is

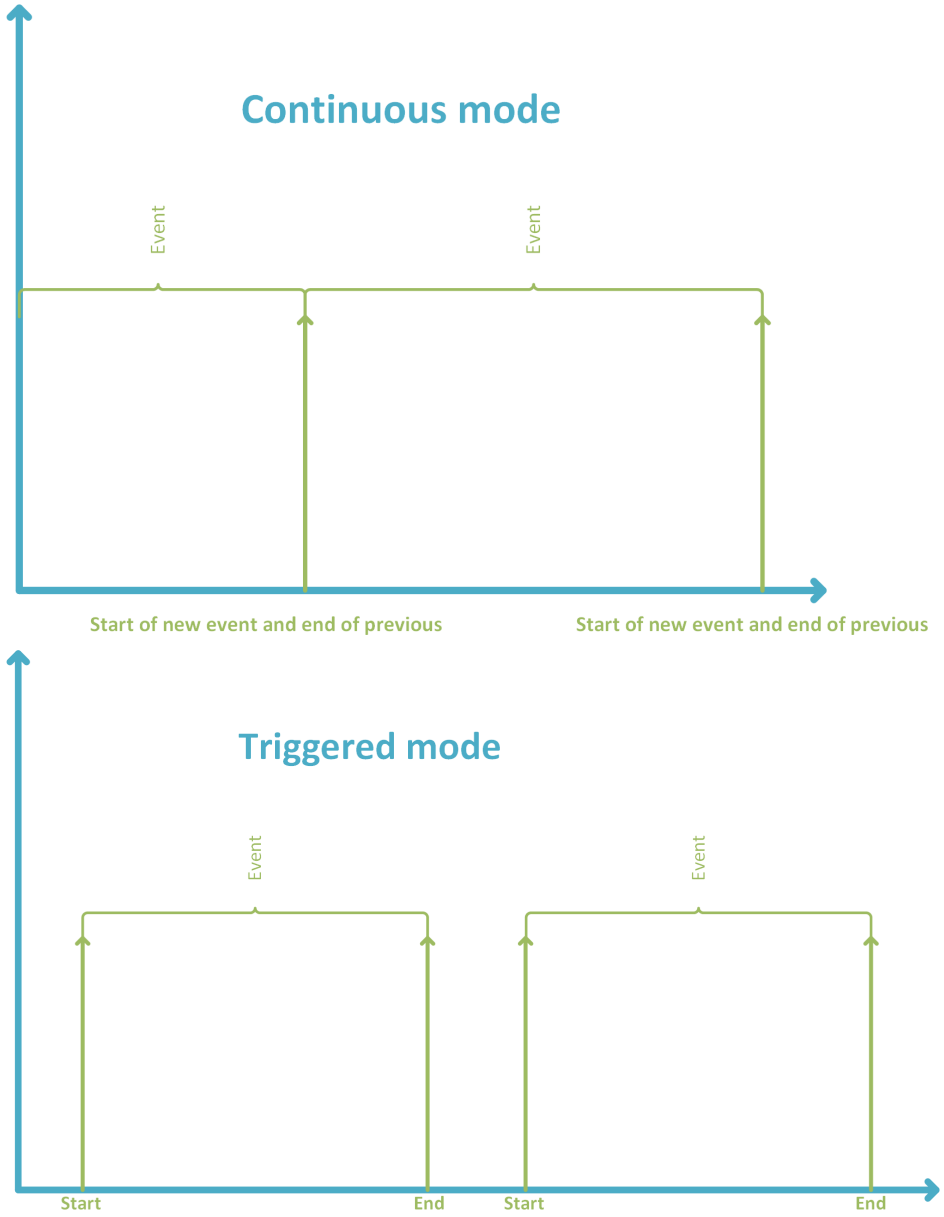


Figure 4.1: Continuous vs Triggered mode

initiated for all 32 channels, meaning they can readout 10 bit data samples 1024 times during this window. A synchronization input allows multiple SAMPA ASICs to align their time frames with respect to each others.[5]

The SAMPA creates data packets from the data assembled from each time frame. Consisting of a header of fixed size 50 bit, followed by a list of 10 bit

samples, created from a single time frame. Even though a time frame consists of 1024 clock cycles, in practice a maximum of 1022 samples are received each time. This is due to the fact that 2 * 10 bit words are required to represent cluster size (size of consecutive samples) and a timestamp. The headers are stored in their own FIFO buffers, separate for each channel, much like the sample buffers. Figure 4.2 shows the structure and format of the packets.

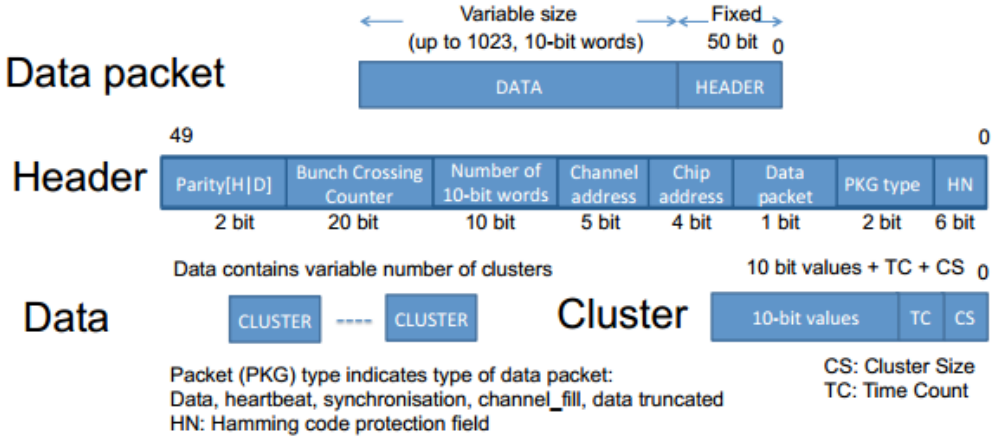


Figure 4.2: Data packet format [5]

The header consists of information regarding the data, such as address for the channel and chip, number of data words in the time frame and packet type. The packet type is used as a marker to see if anything out of the ordinary has happened to the data. This can be if there is no samples in the time frame, causing the packet type to just become a channel fill packet. It can indicate if the stream of data was cut short because the FIFO buffer was full, causing buffer overflow. In case of buffer overflow all data for the particular time frame are discarded and the empty packet is sent with type overflow. Overflow can cause a lot of data to get discarded if the SAMPA can't empty the buffers fast enough, this can happen if the buffers don't have enough space. As the input rate is 3.2 Gbit/s and the readout speed is 1.28 Gbit/s, the SAMPA can receive up to 2.5 times more data per second than it can pass along. This is why the FIFO buffers are necessary, and finding a size which is sufficient, without giving overflow is crucial.

There have been done some calculations on how much data will actually be received from the detector at any given time. It is estimated that on average over all channels for every SAMPA there is around 30% occupancy. This

means that on a global average there is 30% data in every given time frame. Some channels may be full while others are empty, and some may have 40%, but on average there is 30%, which means 306 samples out of 1022 for every time frame. Taking this into account when calculating the input speed of the SAMPA gives 960 Mbit/s which the design should be able to handle without any buffer overflow. Even though there is an estimated average occupancy there can still be some channels which time frame after time frame gets a lot more then that, so how much can the design handle? This is some of the question the simulation will give answers to.

4.1.2 CRU

The CRU serves as an interface between electronics directly on the detector and the online computing systems. It is based on high performance Field-Programmable Gate Array (FPGA) processors, with optical fiber used as input and output [5]. The CRU is somewhat out of the scope of the thesis, and will be regarded in the same fashion as the GBTx. How the CRU is implemented in our design model has no effect on the tests which are going to be performed on the SAMPA and its channels. It is discussed in the thesis work of Damian K Wejnerowski, who is simulating the CRU and inspecting it in great detail.

4.2 Signal processing in the SAMPA

The SAMPA chips will receive and process a huge amount of data, both relevant signals and background noise. In section 4.1.1 we talked about occupancy and amount of samples in each time frame. The estimated amount of 30% refers to relevant samples, removing or compressing the background noise. Seeing as it will always be some interference in the background, there will always come samples with data, and gathering all will be a waste of time and space that could be used on the actual collision data in the detector. Figure 4.3 shows 2 actual events collected from the 2 different ALTRO channels, the events will look similar after the upgrade and we can use this as a starting point. The x-axis expresses the current time bin within a time frame from 0 to 1021. Here one can see that every sample in the time frame has some value most with 48-52, as well as certain peaks here and there. Those peaks or pulses are what is interesting, everything else is considered noise and should be removed. In order for any compression schema or method of reducing noise to be valid it needs to have a compression factor above 2.5 for the average amount of data being processed. The compression fac-

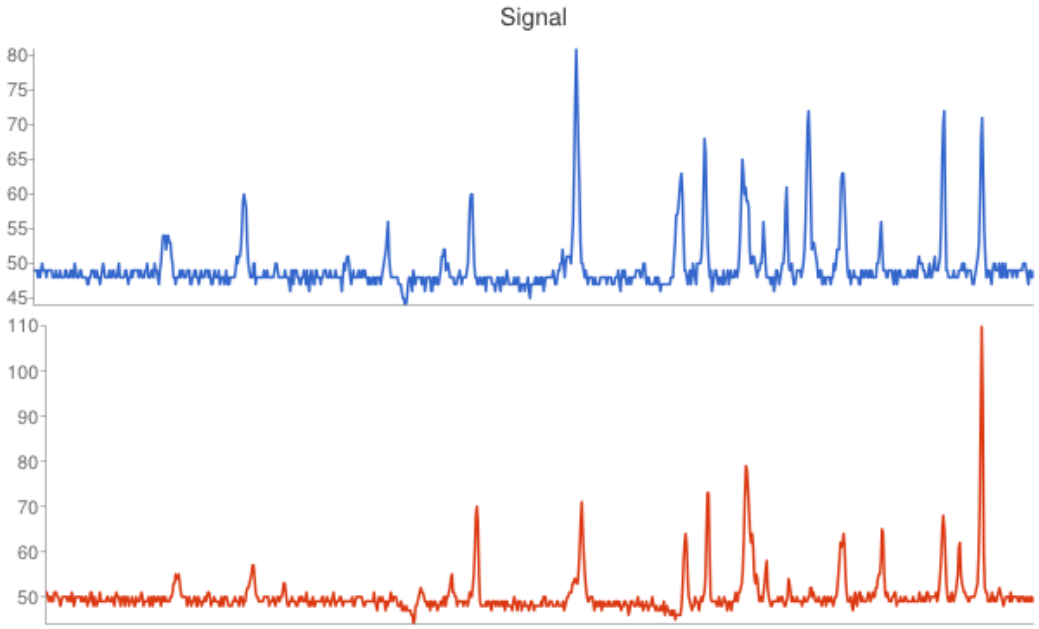


Figure 4.3: Two signals from a previous experiment

tor will be the number of bits in a time frame before compressing compared to after. `factor = (bits before compression / bits after)`. There are a number of ways to reduce the amount of noise, and/or compress the data to a manageable size. What has been used with the current setup and is also discussed to use in in the upgraded setup is Zero suppression.

4.2.1 Zero suppression

Zero suppression is the process of removing insignificant values below a set threshold or baseline.[21]. Applying this in order to remove the background noise without discarding any important samples, a baseline for the Zero suppression must be established. The problem with this is that the baseline may shift, in the case of our 2 example time frames the first one has a visibly lower baseline by 1 or 2. In the upgrade plans described in [5], it is specified how the signal processing will take place. It works by looking at consecutive signals with value over the set threshold, confirming that the peak is indeed a real pulse. The term real pulse refers to a sequence of signals over the threshold with more than one signal, standalone values over the threshold will be discarded. The difference is displayed in Figure 4.4.

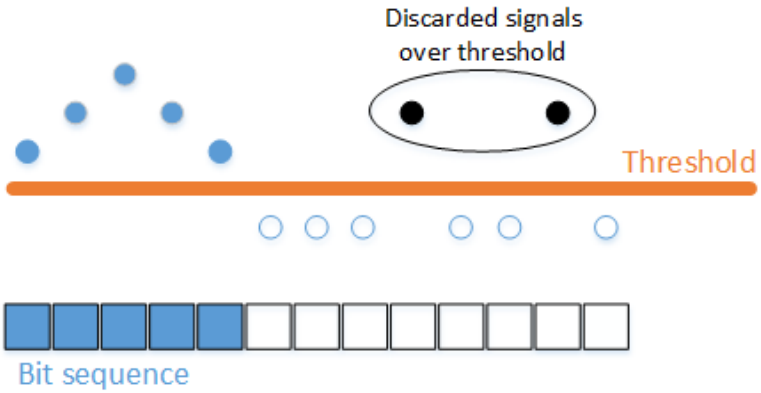


Figure 4.4: Difference between a valid and invalid signal sequence.

Because of the fact that Zero suppression removes signals from various places in a time frame, the data losses its temporal positioning. Therefore every real pulse must be tagged with a time stamp and a word representing the number of words in the pulse. Since for every pulse we add two words, if two consecutive pulses are closer than three words they are merged and counted as one (Figure 4.5).

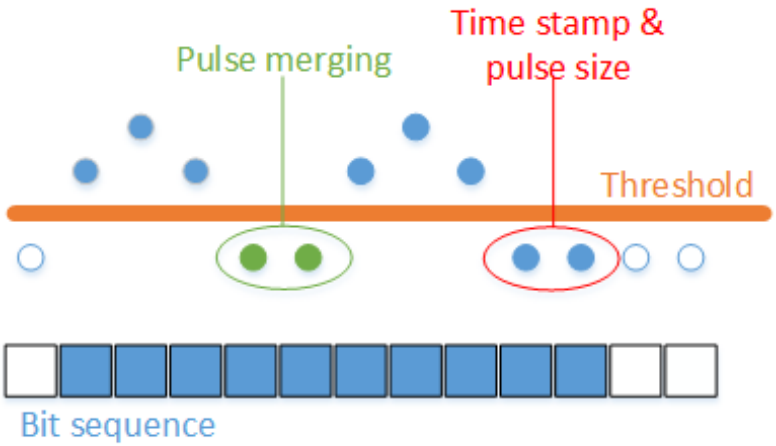


Figure 4.5: Merging of two pulses and the storing of extra pulse information.

In some later discussion regarding the upgrade there has been questions if the described method is insufficient. The theory behind the discussion is that the baseline will shift too much to be able to do efficient Zero suppression without loosing important samples in the process. Another argument against

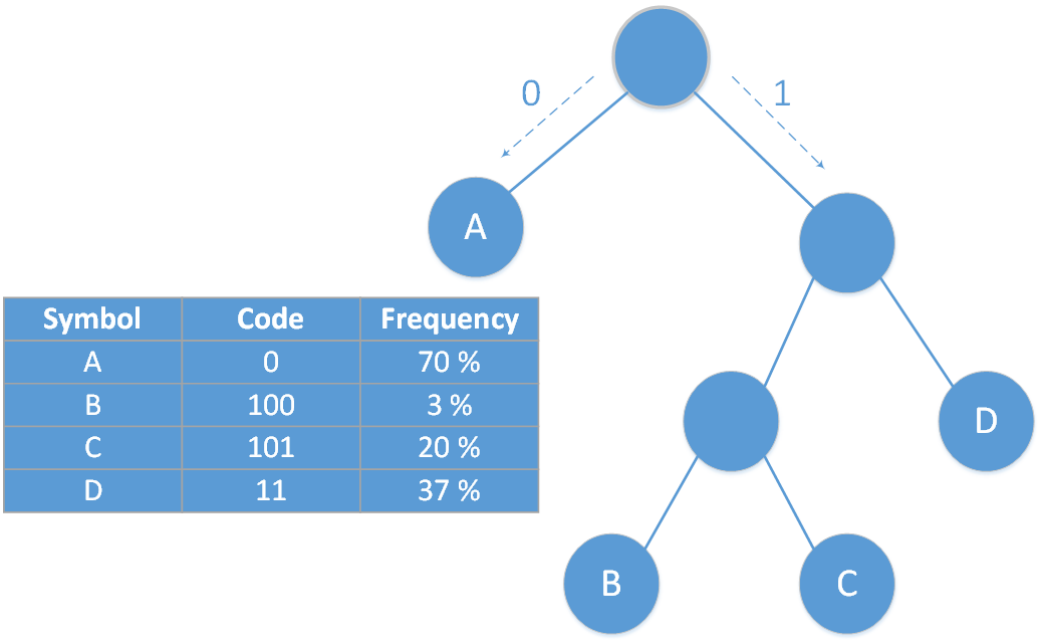
Zero suppression is that with time frames with larger occupancies (40%++) the compression factor is drastically reduced and will not be good enough. This is because time frames with higher occupancy will have more signal pulses, and pulses will be closer together, meaning that more pulses will be merged rather than discarded. This encourage finding another way of processing the signals. One proposed method is to use Huffman coding on the signal values.

4.2.2 Huffman Coding

Huffman is a method used to achieve data compression[22]. It works by assigning binary codes to symbols in order to reduce the number of bits used to encode the symbol. By looking at the frequency of appearance for every symbol used one can produce a frequency table sorted by most frequent. One thing to note is that since the binary codes is of variable length, they may not all be uniquely decipherable. For instance, if the codewords looks like the following: {0,01,11,001}, the code 0 is a prefix to 001. This is solved by using the right data structure to store the codes, the one most used is a *full* Binary Tree (BT). A *full* BT is a tree where every node either has zero or two child nodes. The symbols are then generated by the path from the root to a leaf node, where left and right indicates 0 or 1. Figure 4.6 shows an example of a Huffman tree using made up frequencies for the letters A to D. Here you can see the advantage of sorting by frequency, since the most frequent symbol A only needs one bit to store. Creating the Huffman tree can be implemented using the following pseudo-code algorithm:

```
1  //Input: An array f[1..n] of frequencies
2  //Output: An encoding tree with n leaves
3  //let H be a Priority Queue of integers, ordered by f
4  function Huffman(f) {
5    for(int i = 1; i <= n; i++){
6      H.insert(i);
7    }
8    for(int k = n+1; k <= 2n - 1; k++){
9      i = H.deletemin();
10     j = H.deletemin();
11     //Create a node numbered k with children i,j
12     f[k] = f[i] + f[j];
13     H.insert(k);
14   }
15
16 }
```

Listing 4.1: Huffman algorithm [7]



Symbol	Code	Frequency
A	0	70 %
B	100	3 %
C	101	20 %
D	11	37 %

Figure 4.6: Huffman tree with four symbols.

Need to have Dieter look over this paragraph In the context of compressing data coming from the detector there is one particular foreseen complication. First of all, generating the Huffman tree needs values from the detector, so how do one create a tree with high compression factor without knowing this? One answer to this is to generate a tree using existing data from previous experiments, but update the tree when receiving new data. This gives us an uncertain compression factor in the beginning, but it will become better over time. Because of a shifting baseline encoding the signal values directly may lead to a large Huffman tree, and the best tree for one channel may not be the same for another. It is inefficient to create a separate tree for each channel, as there will be 160 channels for every FEC. A possible solution to this is to encode the derivative of each signal in a time frame compared to the previous value. In other words, for every signal n you store the value: *signal(n) - signal(n - 1)*. Doing so takes away the problem caused by shift in the baseline as it only stores the difference between two signals. This method requires that the first value of every time frame is stored somewhere(maybe the header of a SAMPA packet) in order to decode it later on.

The way the FIFO buffers for each SAMPA channel works is that it stores up to 10 bits in parallel for each slot. This means that compressing 10 bit samples into smaller sizes will still take up 10 bit of space in the buffers.

However reading the data from the buffer will be faster as there is less data to read.

4.3 Designing the simulation model

With all of the information regarding the different components already specified, creating a simulation model should be more than feasible. There will be in total 3 main modules part of the simulation: the SAMPA, GBTx and CRU, but focusing heavily on the SAMPA. In addition to the different modules there is need for a module which can be tasked with producing and/or distributing sample data to the simulation. This module will contain all methods of sending samples to the different SAMPA channels and in doing so start the entire simulation process. The tasks, objectives and goals that this all boils down to is summarized in the list below.

- **Tasks**
  - Designing a model which is accurate, simple and customizable.
  - Creating a data generator module which can send data to the simulation, both synthetic and real.
  - Create a simulation test bench that allows for quick changes in order to run multiple simulations.
  - Run different stress tests on the system, find out where it breaks and why.
  - Run focused simulations on the SAMPA channel buffers.
  - Run simulations which compares Zero suppression and Huffman encoding.
  - Gather, and compile the simulation data into a readable and understandable format.
  - Verify that the simulation results is comparable to what is expected, and calculated beforehand.
- **Goals**
  - With a verified simulation model, we have a created a strong argument that the results are valid.
  - Find out how much SAMPA buffer space is needed.
  - Conclude the compression factor of both Zero suppression and Huffman encoding.

– Verify the overall design of the SAMPA chip, and use the results to come with a recommendation on possible changes.

4.4 Workflow

Approaching this project, one must assume that there will be many uncertainties along the way. Trying to simulate behaviour of an electronic system based solely on its early schematics, while others are working on the design in different areas will undoubtedly lead to many changes in the simulation model. Another characteristic concerning this project is that it requires a lot of work before one can start to see any results, but after completing a satisfying model the results should be easy to obtain without many changes to the simulation program. Splitting the work into different phases, first a longer period of only working on the model, implementing the aspects that are known, and making the model ready to run simulations on. When the base model is complete, an iterative process can start. Simulate for a specific scenario, gather results from the simulation, compile it into a readable format, verify the correctness of the results, in the case they are not legitimate, make adjustments before running new simulations in the same scenario. Customize the simulation parameters and tweak the model for different scenarios, and do the same as before. This way any changes in requirements, or changes to the model can be handled in a separate iteration. Working like this will result in a large period with no speakable results, but this will towards the end be very beneficial.

Chapter 5

Solution implementation

Code snippets, Incremental implementation stages and the final implementation, (before and after huffman), using real data vs random. Implementing fluxiation into the simulation

5.1 Implementing the model in SystemC

5.1.1 The SAMPA module

As the focus of study in this project, the implementation of the SAMPA is the most important piece to the simulation. The overall structure to the SAMPA consists of 32 channels, with a input port for each channel, and in total 4 serial outputs which reads data from the channel buffers. There are a couple of things to think about when translating this design into code.

1. What SystemC channel should the input and output ports use.

    - The requirements for the SAMPA I/O ports is that everything comes in the correct order, and on a specific clock cycle. SystemC comes with the channel type `sc_fifo`, it contains both read and write methods, depending on what channel interface is implemented. So for our one directional design this should work perfectly. The clock cycle is not tied to the ports specifically and will be handled separately.

2. What data structure to use for the channel buffers.

    - When choosing a data structure one need to think about what the purpose of it is, what operations are being done on it, and so forth.

The essential attributes the structure must have are: *Insert* items to the back, *Read/Remove* items from the front, dynamical storage space, and the structure should be a linear one-dimensional sequential storage. On first glance using a FIFO like structure sounds like the best way to go. However in addition to the essential attributes it may be needed to be able to remove and read from the back of the buffer. This is because in the simulation it can be used to grab statistical data from the buffer, and reading from the back will not have any impact of the simulation result, but can make the buffers more versatile. C++ has many different data structures to choose from, all depending on the need for it. In Table 5.1 three different C++ data structures are being evaluated: `vector`, `list` and `queue`. From this table and the requirements of what is needed from the buffer structure, it becomes clear that the `list` container has all the attributes needed, as well as performing equally or better than the rest in the different operations.

3. Handling the clock frequency.

   - SystemC will handle the clock frequency for us, the only thing to note is that SystemC uses pauses in the threads as a way to simulate the clock cycles. In other words, one perform the actions for 1 clock cycle, than the wait statement, and repeat. This means that the frequencies need to be converted to a time delay. An example of such a conversion is shown in listing 3.2.

One possible setup for the SAMPA deceleration is shown in listing 5.1. This takes everything we need into account, storing the channel header and data buffer in an array, declaring input ports, and the output elinks. In addition it declares a single thread for receiving signals, and four threads representing the serial outs reading from the channel buffers. Lets now look at how the implementation for receiving signals can look like. In listing 5.2 the thread logic is implemented, and here some weaknesses with the SAMPA module becomes clear. Having to iterate over every channel every timebin can become costly and hard to maintain when the code becomes more complex. In the code shown there is already a flaw, if one of the channel buffers has `overflow`, none of the other buffers will receive data. The `overflow` variable needs to be stored as a array as well to be able to know what channel it represented. The same problem will occur for every value that is unique for each channel. A principle of Object-Oriented Programming (OOP) is

Table 5.1: Data structure comparison[23], [24], [25].

	Time			
Operation	**Vector**	**List**	**Queue**	**Remarks**
Add back	O(1)	O(1)	O(1)	Constant time for all containers.
Add front	O(n)	O(1)	X	Vector does not have a direct method for adding to front. Queue cant do that at all.
Access back	O(1)	O(1)	O(1)	Constant time for all containers.
Access front	O(1)	O(1)	O(1)	Constant time for all containers.
Remove front	O(n+m)	O(1)	O(1)	Vector erase is linear to number of deleted elements + number of elements after last deleted item (moving).
Remove back	O(1)	O(1)	X	Queue does not have a method for doing this.
Size of container	O(1)	O(1)	O(1)	Constant time for all containers.

single-responsibility, meaning that every class/object should be responsibly for one piece of functionality. One solution to the previous problem following this principle can be to create a `Channel` sub-module inside of the SAMPA module. The channel module will contain logic for receiving signals, as this happens for every channel, while the SAMPA will contain the serial outs, which are accessing data from the buffers.

```
1  class SAMPA : public sc_module {
2  public:
3
4    //I/O Ports.
5    sc_port< sc_fifo_in_if< Sample > >  inputPorts[
         NUMBER_OF_CHANNELS];
6    sc_port< sc_fifo_out_if< Packet > > eLinks[NUMBER_OF_ELINKS];
7
8    //4 async serial out threads
9    void serialOut0(void);
10   void serialOut1(void);
11   void serialOut2(void);
12   void serialOut3(void);
13
14   //Routing method, serial outs read from correct buffer.
15   int processData(int serialOut);
16
17   //Thread which receives samples for every channel.
18   void receiveSamples(void);
19   SAMPA(sc_module_name name);
20
21 private:
22   int Addr; //Hardware Address
23   list<Sample> dataBuffers[NUMBER_OF_CHANNELS]; //Sample FIFOs
24   list<Packet> headerBuffers[NUMBER_OF_CHANNELS]; //SAMPA Header
         FIFOs
25 };
```

Listing 5.1: SAMPA - First iteration.

```
1  void SAMPA::receiveSamples(){
2    Sample sample;
3    int currentTimebin = 0;
4    float waitTime = 100.0; // 10bit 10 MHz clock: 1 / 10 ^ 6 =
       100 ns.
5    bool overflow = false;
6    while(true){
7      for(int i = 0; i < NUMBER_OF_CHANNELS; i++){
8
9        if(inputPorts[i].nb_read(sample)){
10         if(dataBuffers[i].size < MAX_BUFFER_SIZE
11           && !overflow){
12             dataBuffers[i].push_back(sample);
13             currentTimebin++;
14           } else {
15             overflow = true;
16           }
17         }
18         if(currentTimebin == MAX_NUMBER_OF_TIMEBINS){ //End of
             timeframe
19
20           //Creating a header packet and adding to headerBuffer.
21           Packet packet(timeFrame, i, dataBuffers.size(),
                overflow);
22           headerBuffers[i].push_back(packet);
23         }
24       }
25       wait(waitTime, SC_NS);
26     }
27   }
```

Listing 5.2: Receive thread.

The changes made to the SAMPA and `Channel` modules is shown in listing 5.3 and 5.4. The SAMPA now has an array of `Channel` modules, a new method called `initChannel()` that initiate the channels, and connects them to the correct input port.

```
1  class SAMPA : public sc_module {
2  public:
3
4    //I/O Ports.
5    sc_port< sc_fifo_in_if< Sample > >  inputPorts[
         NUMBER_OF_CHANNELS];
6    sc_port< sc_fifo_out_if< Packet > > eLinks[NUMBER_OF_ELINKS];
7
8    //Channels
9    Channel *channels[SAMPA_NUMBER_INPUT_PORTS];
10
11   //Initialize channels
12   void initChannels(void);
13
14   //4 async serial out threads
15   void serialOut0(void);
16   void serialOut1(void);
17   void serialOut2(void);
18   void serialOut3(void);
19
20   //Routing method, serial outs read from correct buffer.
21   int processData(int serialOut);
22   SAMPA(sc_module_name name);
23
24 private:
25   int Addr; //Hardware Address
26 };
```

Listing 5.3: SAMPA - Second iteration.

```
1  class Channel : public sc_module {
2  public:
3
4    //Ports between the DataGenerator and the Channel
5    sc_port< sc_fifo_in_if< Sample > > inputPort;
6
7    //Data and Header buffers
8    list<Sample> dataBuffer;
9    list<Packet> headerBuffer;
10
11   //Main SystemC Thread.
12   void receiveData();
13   //Getter and Setter methods
14     .......
15   //End
16   Channel(sc_module_name name);
17
18 private:
19   //Can specify which pad the channel comes from.
20   int Pad;
21   int PadRow;
22   int Addr;
23   int SampaAddr;
24 };
```

Listing 5.4: Channel module.

The `Channel` module now has the `receiveData()` thread, and its own data/header buffer. Having it structured like this makes it possible to add `Channel` specific variables or methods, without disturbing the SAMPA as a whole. The implementation of the receive thread is now more simple in terms of complexity, and is exclusive for each `Channel`. Listing 5.5 shows the basic thread structure, excluding any data compression or processing. It continuously receives samples and adds them to the buffer, unless the buffer reach its maximum size.

```
while(true){

    //Read from datagenerator
    if(port_DG_to_CHANNEL->nb_read(sample)){
      numberOfClockCycles++; //timebin

      //Check if max buffer size is reached.
      if(dataBuffer.size() + sample.size > constants::
          CHANNEL_DATA_BUFFER_SIZE){
        overflow = true;
      }

      //Add sample to buffer if there is no overflow.
      if(!overflow){
        numberOfSamples++;
        addSampleToBuffer(sample, numberOfClockCycles);
      }

    }
    //When we reach the end of a timeWindow we send the header
        packet to its buffer and starts a new window
    if(numberOfClockCycles == constants::
        NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW ){

      //Remove samples added earlier in timeframe if overflow
      if(overflow){
        for (int i = 0; i < numberOfSamples; ++i)
        {
          dataBuffer.pop_back();

        }
        numberOfSamples = 0;
      }
      //Create header packet, and add to header buffer.
      Packet header(currentTimeWindow, this->getAddr(),
          numberOfSamples, overflow, 1, currentOccupancy);
      header.sampaChipId = this->getSampaAddr();
      headerBuffer.push(header);
      //Clean up temp variables
    }
}
```

Listing 5.5: Receive samples.

The basic implementation of reading from the buffer is somewhat more complicated than receiving data. This is because the real reading procedure is more complex in itself. In listing 5.6 the procedure is shown, along with

one of four equal serial out threads. The procedure loops through the eight channels for the given serial out, getting the correct channel, and reading out the entire timeframe. One thing to notice here is that the actual data from the buffers aren't passed along further. This is because we only care about the time it takes to transfer them, which is being calculated in the `wait` statement. The header packet is sent as it contains important information which can be useful later on.

```
1  void SAMPA::processData(int serialOut){
2
3    //Go through all channels for specific serialout
4    for(int i = 0; i < constants::CHANNELS_PER_E_LINK; i++){
5      float waitTime = 0.0;
6      //Find channel
7      int channelId = i + (serialOut*constants::
           CHANNELS_PER_E_LINK);
8      Channel *channel = channels[channelId];
9
10     //Start reading after first timeframe is complete
11     if(channel->isReadable()){
12       //find header
13       if(!channel->headerBuffer.empty()){
14
15         Packet header = channel->headerBuffer.front();
16         channel->headerBuffer.pop();
17
18         //Read from databuffer, but check for overflow.
19         if(!header.overflow || header.numberOfSamples > 0){
20
21           for(int j = 0; j < header.numberOfSamples; j++){
22             if(!channel->dataBuffer.empty()){
23
24               channel->dataBuffer.pop_front();
25             }
26           }
27         }
28         //Simulate number of clock cycles it took to read the
               timeframe.
29         waitTime = (5 + header.numberOfSamples); //50bit header
               + 10 bit samples
30         porter_SAMPA_to_GBT[serialOut]->nb_write(header);
31
32         wait((constants::SAMPA_OUTPUT_WAIT_TIME * waitTime),
               SC_NS);
33       }
34     }
35   }
36 }
37
38 void SAMPA::serialOut0(){
39   //1 of 4 serial out threads
40   while(true){
41     //wait small amount of time to create cycle.
42     wait(1, SC_NS);
43     processData(0); //0-3
44   }
45 }
```

Listing 5.6: Reading data from the SAMPA buffers.

Viser implementasjon av huffman og zero suppression her.

5.1.2 The DataGenerator module

The simulation model is scoped to the readout electronics, which doesn't handle the creation of signals. This means that the simulation needs a module which can create and distribute data, both simulated and real to the SAMPA modules. The module needs to be able to continuously send samples based on what is desired for a specific simulation scenario. It is important that all the different methods for sending samples, do so in the same fundamental way. The rest of the simulation model should not need to change for each different scenario, but instead the data generator should make sure that it sends samples in the correct format, regardless of the simulation type.

With the expected behaviour given in the previous paragraph, the bounds and requirements of the data generator can be established. The module will continually be updated, and new functionality will be added as new simulation scenarios (surface?). In order for this to be efficient, the module needs to be easily extended, without causing any disturbance to the rest of the module. Sending samples should follow the specifications for the actual hardware, emulating the connection between the readout chambers and the SAMPA asic. That includes the clock frequency and making sure every channel receives data asynchronously.

Basic data generating functions??

Achieving the same format for every simulation sample is done by using a custom class which the link between the data generator and the `Channels` are expecting as input. This means that every time the data generator sends a signal it uses this class. The format of the `Sample` class is discussed and shown in section 5.1.3. Creating a single function that has the core functionality of data generator, which all simulation scenarios use as base could have been a good way to remove some of the boilerplate code needed, but without knowing all types of simulations from the start this could lead to unwanted restrictions later on. Because of this, for every different way of creating/distributing data, there is a completely different function. The benefits of implementing it like this is that the functions doesn't depend on each other and can be separately updated or improved, which allows for easier development as the module becomes larger and more complex.

To be able to quickly switch between different functions when running a different simulation, the module contains only a single SystemC thread, where we can choose the correct function based on the simulation type. The `sink_tread()` function can be seen in 5.7. The testbench is explained in section **??**, but one of its purposes is storing global variables which are used around the different modules. One example of this is shown in 5.7 with the use of the `DG_SIMULATION_TYPE` variable. The different functions for distributing data is also shown.

```
1  void DataGenerator::sink_thread(void){
2    if(constants::DG_SIMULTION_TYPE == 1){
3      standardSink();
4    } else if(constants::DG_SIMULTION_TYPE == 2){
5      incrementingOccupancySink();
6    } else if(constants::DG_SIMULTION_TYPE == 3){
7      alternatingOccupancySink();
8    } else if(constants::DG_SIMULTION_TYPE == 4){
9      sendBlackEvents();
10   } else {
11     sendGaussianDistribution();
12   }
13 }
```

Listing 5.7: Data generator SystemC thread.

They have been implemented one by one in different stages of the development phase. The first three: `standardSink()`, `incrementingOccupancySink()`, and `alternatingOccupancySink()` are functions created in the early stages, before there was any need to implement compression schemes in the simulation model. They all assume that the data being generated are already compressed, and can be sent through the simulation without any data processing. The shear amount of samples is the important factor with these functions, relying on a flat occupancy value to determine if a sample is sent. The chosen occupancy(specified as percent(0-100)) is compared against a number generated by a Random Number Generator. If the number is lower then the occupancy the sample is sent, else it will send an empty sample instead. In other words, a sample is sent a fixed percent of the time, specified by the occupancy. The `standardSink()` implementation is shown in 5.8. The double loop seen here is just about the same for all the five functions, where the outer one counts the number of time frames and the inner goes through all SAMPA channels present. The wait statement is called after the inner loop, which is equivalent to sending samples to all channels in the same clock cycle. This function is a good starting point in order to test the rest of the simulation model, and to verify that the model can be trusted. Since it

relays on a flat occupancy it is expected that most channels will get similar data, and there will be little to no fluctuations. Real experiment data will not be as flat, and the occupancy will differ from timeframe to timeframe.

```
1  void DataGenerator::standardSink(){
2    int64_t packetCounter = 1;
3    int currentSample = 0;
4    int currentTimeWindow = 1;
5    RandomGenerator randomGenerator;
6    //While we still have timewindows to send
7    while(currentTimeWindow <= constants::
         NUMBER_TIME_WINDOWS_TO_SIMULATE)
8    {
9      //Loop each channel
10     for(int i = 0; i < constants::NUMBER_OF_SAMPA_CHIPS *
           constants::SAMPA_NUMBER_INPUT_PORTS; i++)
11     {
12
13       if(randomGenerator.generate(0, 100) <= constants::
             DG_OCCUPANCY){
14         //Send real sample
15       } else {
16         //Send empty sample
17       }
18     }
19     currentSample++;
20     //Increments timeWindow
21     if(currentSample == constants::
           NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW )//1021 samples
22     {
23       currentTimeWindow++;
24       currentSample = 0;
25     }
26     wait((constants::DG_WAIT_TIME), SC_NS);
27   }
28 }
```

Listing 5.8: Data generator SystemC thread.

The limitations of the `standardSink()` function was the motivation behind the `incrementingOccupancySink()` and the `alternatingOccupancySink()`. As their names suggest they implement more diversity into the simulation with the ability to increase, or alternate the data occupancy as time goes. This creates more possibilities to test the limits of the model, especially the SAMPA buffers. The code for these functions is similar to the `standardSink()`, and the differences is shown in **??** and **??**.

```
1  //Standard sink
2  if(currentSample == constants::
       NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW )
3  {
4    currentTimeWindow++;
5    currentSample = 0;
6  }
7
8  //Increasing occupancy sink
9  if(currentSample == constants::
       NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW ){
10   //How often do we increase the occupancy?
11   if(currentTimeWindow % (constants::
         NUMBER_TIME_WINDOWS_TO_SIMULATE / constants::
         TIME_WINDOW_OCCUPANCY_SPLIT) == 0){
12     //Increase occupancy.
13     occupancy = occupancy + constants::
           TIME_WINDOW_OCCUPANCY_SPLIT;
14   }
15   currentTimeWindow++;
16   currentSample = 0;
17 }
```

Listing 5.9: Data generator SystemC thread.

```
1
2  //Standard sink
3  if(randomGenerator.generate(0, 100) <= constants::DG_OCCUPANCY){
4    //Send real sample
5  } else {
6    //Send empty sample
7  }
8
9  //Alternating occupancy sink
10 if(randomGenerator.generate(0, 100) <= occupancyPoints[
       currentTimeWindow - 1]){
11   //Send real sample
12 } else {
13   //Send empty sample
14 }
```

Listing 5.10: Data generator SystemC thread.

The `incrementingOccupancy()` will provide initial results on how much occupancy the system will be able to handle, while the `alternatingOccupancySing()` will test its ability to handle various amount of data distributed over time. This is an improvement from the `standardSink()`, but the functions does

still not care about how the data is shaped or distributed inside a timeframe. To get more accurate results from the simulation there is a need for data which more closely resembles actual experiment data. The implementation so far doesn't worry about the value of the samples, but the values are required in order to test Zero suppression and Huffman encoding.

Creating Normally distributed samples

Her trenger litt hjelp formulere meg. En del ting vi har snakket med dieter om jeg ikke vet om jeg finner noen kilde p osv..Trenger den distribusjons grafen til Dieter.

Analysis done on the results from RUN 1??(Need source) has shown that the shape of the incoming samples can be somewhat predicted. This is shown in (NEED FIGURE HERE), where the x axis is the position of a single pad on the actual detector. The closer to origin, the closer to the center of the detector. The pads closer to the center has generally higher occupancy than the outer ones, which makes them the worst case, and the most interesting for use in the simulation. FIGURE shows that the average occupancy for the inner pads is 28%, in 10% of all timeframes the occupancy is 44% and in 1% of all timeframes the occupancy is 74%. Based on this a normal distribution of occupancies can be created for a more realistic simulation. A normal distribution is a mathematical/statistical distribution of values which depend on a central value (the mean value) and a stretch factor (the standard deviation)[26]. An example of a normal distribution is shown in Figure 5.1.

Using a normally distributed Random Number Generator a set of occupancy values can be created. For every new timeframe in the simulation, a random value from the set is picked as the current occupancy. The size of the set is determined by the number of timeframes in the simulation. The mean value for the distribution is already known being 28%, but the standard deviation is not obvious and needs to be calculated. Using the statistics given in FIGURE the deviation can be calculated. In a set x of 100 values, 1 value will be 74, 10 will be 44 and the rest will on average be 28. First of the *varians* must be calculated by the following formula: $\frac{\sum_{i=1}^{100}(x_i - mean)^2}{100}$ Take the square root of the *varians* and the result is the standard deviation. How this is implemented programmatically is shown in 5.11 along with creating the final set of occupancies.

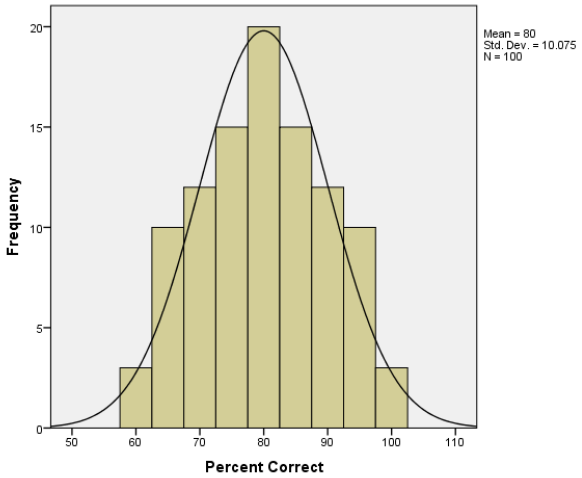


Figure 5.1: Normal distribution. [6]

```
1  std::vector<int> DataGenerator::getOccupancy(){
2    double mean = 28.0;
3    double sum = 0.0;
4    double array[100];
5
6    array[0] = 74;
7    for(int i = 1; i <= 10; i++){
8      array[i] = 44;
9    }
10   //+-10 for a wider distribution.
11   for(int i = 11; i < 100; i++){
12     array[i] = generator.generate(mean-10, mean+10);
13   }
14   for(int i = 0; i < 100; i++)
15     sum += pow(array[i] - mean, 2.0);
16
17   double varians = sum/100;
18   double deviation = sqrt(varians);
19
20   std::default_random_engine gen(seed);
21   std::normal_distribution<double> dist(mean, deviation);
22   std::vector<int> result;
23
24   //create a vector of occupancies based on the normal
        distribution.
25   for(int i = 0; i < constants::NUMBER_TIME_WINDOWS_TO_SIMULATE;
         i++){
26     int occ = (int)dist(gen);
27     if(occ <= 0)
28       result.push_back(1);
29     else
30       result.push_back(occ);
31   }
32   return result;
33 }
```

Listing 5.11: Data generator SystemC thread.

After testing the distribution out the conclusion was that the distribution became very narrow towards the mean. By using values from $mean - 10$ to $mean + 10$ gave a bigger deviation, and as a result a wider distribution. The difference by increasing the deviation is shown in Figure 5.3.

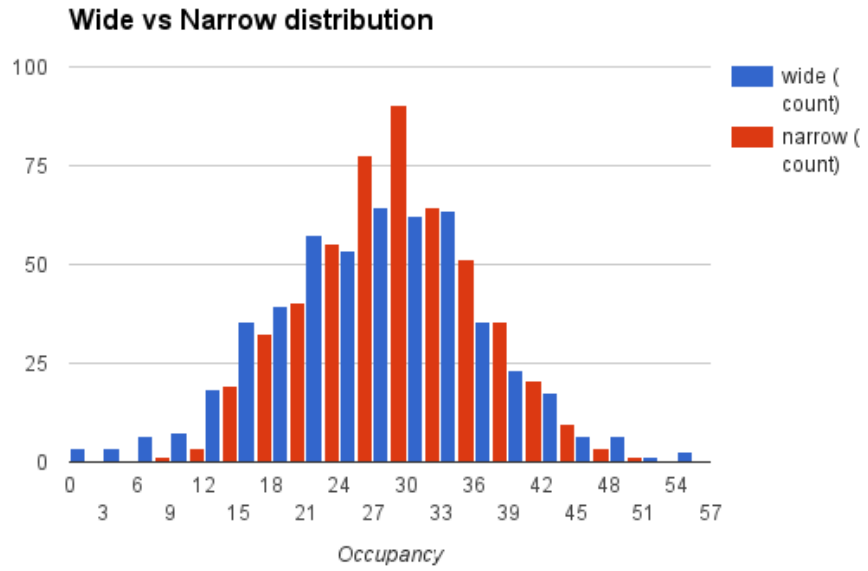


Figure 5.2: Difference in the normal distribution.

To be able to test the Zero suppression the samples need to have a shape which allows for this. The Zero suppression works best with data where the peaks are wider, but further apart from each other. So to test it in its worst case the samples within a timeframe needs to have as many peaks as possible, split evenly over it. A peak will be of width 3, and the occupancy for the timeframe determines the space between two peaks. An example using 28% occupancy is displayed in **??**. Even though the data still doesn't look like the real thing, in the case of testing the Zero suppression it should sufficient since it only cares if the sample value is zero or above.

A process for determining the space between each peak is the next thing to look at. The space can be calculated by first finding the number of samples with value 0, and divide it by the number of peaks in the timeframe. Since both the occupancy and the number of samples in a timeframe is know, the math becomes very straight forward. The solution used is shown in 5.12.

Figure 5.3: FEIL FIGUR!

```
1 double DataGenerator::calcSpace(int occ){
2   double numberOfSamples = (constants::
        NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW * occ) / 100.0;
3   double numberOfPeaks = numberOfSamples / 3.0;
4   double numberOfEmptyTimebins = constants::
        NUMBER_OF_SAMPLES_IN_EACH_TIME_WINDOW - numberOfSamples;
5
6   return (numberOfEmptyTimebins / numberOfPeaks);
7 }
```

Listing 5.12: Calculating the space between two peaks in a timeframe.

With a completed set of occupancies and the necessary functions to help shape the data the only thing left is to implement the sink function. It uses the same loop structure as the three previous sink functions, but in addition before every timeframe it picks the next occupancy from the set and calculate the space between the peaks. *Her mangler jeg litt utfylding, kode og litt drfting om mlet bak denne typen simulering.*

"Real data"

The last type of simulation will use realistic datasets as input. Two types of datasets has been available for use: Experiment data collected from RUN 1, and synthetic data created by researchers which matches the expected data from RUN 3. In addition to this, we want to use the data from RUN 1 and create pileup data. Pileup occurs when a channel gets overlapping data from multiple neighbouring channels in the same timeframe *Riktig?*. In the worst case the pileup can happen five times, i.e samples from five neighbouring

channels are causing interference. By overlapping five timeframes from the RUN 1 data, a third set for use in the simulation is made available.

The tasks required to implement this boils down to the following list.

- Define a common data structure to store the different data.
- Create readout functions for the different datasets.
- Implement the sink function.

Kode for det forklart over. Hva nsker vi oppn med bruke realistiske data.

5.1.3 Signal classes

SystemC allows the creation of custom `classes` to be used as data type when transferring data between modules. There are some requirements when doing so that needs to be fulfilled for it to work. SystemC requires you to define several methods which is vital for the read/write methods of a SystemC channel. The read/write methods involve copying the custom data type. Because of this it requires the definition of the assignment operator(`operator=()`). In addition the output streaming(`ostream& operator<<()`) method is required.

The simulation needs two different data types, these have been briefly shown in the previous code listings. The `Sample` and the `Packet` classes. `Sample` represents a single 10-bit signal, storing information about what timeframe the sample belongs to, the signal value itself, and other statistical variables. Representing the SAMPA header is the `Packet` class, it stores the relevant values selected from its documentation. This include the timeframe, channel id, sampa id, number of samples and whether there was overflow in its timeframe. Source code for the `Packet` class is shown in listing 5.13. The `Sample` class is implemented in similar fashion and will not be displayed in the report.

```
1  class Packet
2  {
3  public:
4
5    int timeFrame;
6    int channelId;
7    int sampaChipId;
8    int numberOfSamples;
9    bool overflow;
10   int sampleId;
11   int occupancy;
12   Packet(int _timeFrame, int _channelId, int _numberOfSamples,
         bool _overflow, int _sampleId, int _occupancy);
13   Packet();
14
15   inline friend std::ostream& operator << ( std::ostream &os,
         Packet const &packet )
16   {
17     os << "Packet: time frame: " << packet.timeFrame << ",
          sampaId: " << packet.sampaChipId << ", channelId: " <<
          packet.channelId << ", number of samples: " << packet.
          numberOfSamples;
18
19     return os;
20   };
21   Packet& operator = (const Packet& _packet);
22 };
```

Listing 5.13: Custom data type - The SAMPA header.

5.1.4 Connecting the modules together

As seen in **??** connecting the modules together is done in the `sc_main` method. *Mangler mer utdypende her.*

5.2 Creating a customizable testbench

Dette omtaler egentlig bare en fil med et namespace der alle variabler for simuleringen er. Kanskje ikke verd et eget del kapittel? Hvor skal jeg sette det inn da?

5.3 Data gathering

Creating Struct object in the sampa and channel modules to gather information, which can be written to a graph file later on

Chapter 6

Evaluation and results

Running the tests, results from different tests, Evaluating the final product

6.1 Simulation results

6.1.1 Initial test scenarios

6.1.2 First substantial simulations

6.1.3 Full simulation

6.1.4 Zero Suppression - preliminary results

6.1.5 Zero Suppression - extended results

6.1.6 Huffman results

Chapter 7

Conclusion and Future work

Conclude the thesis, talk about the impact it has and its usefulness in future planing of the front end electronics.

Bibliography

[1] Image of the LHC. http://slhcpp.web.cern.ch/SLHCPP/images/courier_article1.jpg. Accessed: 2015-02-17.

[2] The ALICE detector. http://alicematters.web.cern.ch/sites/alicematters.web.cern.ch/files/images/ALICE_paper_diag.jpg. Accessed: 2015-02-19.

[3] Upgrade of the ALICE Time Projection Chamber - Techincal Design Report. http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2_TPC.html. Accessed: 2015-02-13.

[4] C. Lippmann. Example of a GEM pad structure. private communication.

[5] Upgrade of the Readout & Trigger System - Techincal Design Report. http://cds.cern.ch/record/1603472/files/ALICE-TDR-015.pdf. Accessed: 2015-02-13.

[6] Example of a normal distribution. http://study.com/cimages/multimages/16/normal_distribution2.PNG. Accessed: 2015-04-30.

[7] Dasgupta Papadimitriou and Vazirani. *Algorithms*. Alan R. Apt, 2008.

[8] Long Shutdown 2 @ LHC. https://indico.cern.ch/event/315665/session/7/contribution/37/material/paper/1.pdf. Accessed: 2015-01-09.

[9] Werner Riegler. The ALICE Upgrade plans - Article. http://ph-news.web.cern.ch/content/alice-upgrade-plans. Accessed: 2015-01-12.

[10] CERN - Article. http://home.web.cern.ch/about. Accessed: 2015-01-12.

[11] The birth of the web - Article. http://home.web.cern.ch/about. Accessed: 2015-01-12.

[12] The Large Hadron Collider - Article. http://home.web.cern.ch/topics/large-hadron-collider. Accessed: 2014-11-14.

[13] The Large Hadron Collider - Brochure. http://cds.cern.ch/record/1165534/files/CERN-Brochure-2009-003-Eng.pdf. Accessed: 2015-01-16.

[14] The ALICE experiment - Homepage. http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2Experiment-en.html. Accessed: 2015-01-17.

[15] Quark-Gluon plasma - Article. http://home.web.cern.ch/about/physics/heavy-ions-and-quark-gluon-plasma. Accessed: 2015-01-18.

[16] The ALICE experiment - Article. http://home.web.cern.ch/about/experiments/alice. Accessed: 2015-01-17.

[17] ALTRO - Article. http://aliceinfo.cern.ch/Public/en/Chapter2/Chap2_TPC.html. Accessed: 2015-02-13.

[18] ASIC - Definition. http://www.radio-electronics.com/info/data/semicond/asic/asic.php. Accessed: 2015-02-13.

[19] Jerry Banks. *Discrete-event System Simulation.* Upper Saddle River, NJ: Prentice Hall, 2001.

[20] Bunton Black, Donovan and Keist. *SystemC: From the Ground Up.* Springer Science+Business Media, second edition, 2010.

[21] Daintith and Wright. zero suppression. "http://www.oxfordreference.com/10.1093/acref/9780199234004.001.0001/acref-9780199234004-e-5900". Accessed: 2015-02-24.

[22] Ince. Huffman coding. "http://www.oxfordreference.com/10.1093/acref/9780191744150.001.0001/acref-9780191744150-e-1565". Accessed: 2015-02-25.

[23] cplusplus.com. Vector, c++ container - Documentation. http://www.cplusplus.com/reference/vector/vector/. Accessed: 2015-03-17.

[24] cplusplus.com. List, c++ container - Documentation. `http://www.cplusplus.com/reference/list/list`. Accessed: 2015-03-17.

[25] cplusplus.com. Queue, c++ container - Documentation. `http://www.cplusplus.com/reference/queue/queue/`. Accessed: 2015-03-17.

[26] Roger L Casella, George; Berger. *Statistical Inference*. Duxbury, second edition, 2001.