Bachelorthesis
Physics

RWTH Aachen University

Physics Institute III B

Digital Rate Regulation at the Cooler Synchrotron COSY

Lefan Zhang

Aachen, September 2021

Eigenhändigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

(Ort, Datum)	(Lefan Zhang)

Diese Arbeit wurde betreut von:

1. Prüfer:	**Prof. Dr. Jörg Pretz**	(RWTH Aachen)
2. Prüfer:	**Prof. Dr. Oliver Pooth**	(RWTH Aachen)

Sie wurde angefertigt im Institut für Kernphysik (IKP-4) der
Forschungszentrum Jülich GmbH

Acknowledgement

I would like to thank the people involved into my bachelor thesis. First I want to thank Ilja Bekman for helping me throughout my work, especially with the technical questions. Secondly I want to thank Prof. Jörg Pretz not only for correcting my thesis but also for helping me with the related physical questions. I would also like to thank the people at the IKP-4 institute for being very helpful with COSY related questions.

Abstract

Particle accelerators are an important tool in modern particle physics. Accelerators can be used in different collision experiments to gain deeper knowledge about the behaviour of elementary particles. Operating such accelerators demands a wide range of precise data processing systems.
One of the tasks in a collision experiment is the target rate regulation, responsible for regulating the particle rate hitting a fixed target. At the Cooler Synchrotron in Jülich the current device responsible for this is the so-called "Schneiderbox", an analogue counting and regulating device.
The task of this bachelor thesis is to replace the analogue "Schneiderbox" with a Field Programmable Gate Array (FPGA). Using Hardware Description Language (HDL), the rate regulation is digitized, which makes it possible to access the regulation system remotely and makes automatic rate regulation possible.

Contents

List of Figures iv

List of Tables iv

1. Introduction 1

2. Physical Background and Set Up 3
2.1. Scattering Experiments and Target Rate Regulation 3
2.2. Fourier Transform and White Noise 5
2.3. Cooler Synchrotron 5
2.3.1. Replacing the Analog Regulation System 6
2.4. FPGA Basics 7
2.4.1. The Red Pitaya Board 7
2.4.2. Verilog 7

3. Implemented Design 9
3.1. Fundamental Design 9
3.2. Counting Module 10
3.3. Amplitude Adjusting Module 12
3.4. The 'generate' Function 15

4. Testing and Simulations 17
4.1. Simulations with Vivado Testbench 17
4.1.1. Simulations at 30.000 Signals per Second 17
4.1.2. Simulations at 200.000 Signals per Second 18
4.1.3. Simulations at 80.000 Signals per Second 19
4.2. Tests with Frequency Generator 21
4.3. Tests with Discriminator 24

5. Summary and Outlook 27

Bibliography 29

A. Appendix A 1
A.0.1. Verilog Code for Counting Module . . . . . A 1
A.0.2. Verilog Code for Amplitude Adjustment Module . . . . . A 10
A.0.3. Verilog Code for Verilog Testbench Source . . . . . A 22

List of Figures

2.1. Rutherford's scattering experiment set up [4] 3
2.2. Scattering experiment set up 4
2.3. Power spectrum density of an arbitrary band limited white noise signal 5
2.4. Layout plan of the COSY accelerator [7] 6
2.5. The Red Pitaya board [11] 7

3.1. Flowchart of the FPGA modules 10
3.2. Positive flank detection with a 4 bit shift register 11
3.3. Sketch of the adjusting step function 14

4.1. Simulation results for a target rate of 30.000 signals per second 18
4.2. Simulation results for a target rate of 200.000 signals per second 19
4.3. Test results with a constant signal rate of 150.0000 signals per second . 21
4.4. Test results with a constant signal rate of 2000 signals per second . . . 21
4.5. Set up with the signal generator 22
4.6. Test at 400khz, the upper plot describes the behaviour of the amplitude register, the lower plot describes the difference between measured signal rate and the desired rate 23
4.7. Test at 7Mhz, the upper plot describes the behaviour of the amplitude register, the lower plot describes the difference between measured signal rate and the desired rate 23
4.8. Set up at the discriminator 24
4.9. Test results without generator 25
4.10. Test results with generator 25

List of Tables

3.1. Port listing for the whole design 9
3.2. Scale for various msb differences between desired rate and signal rate deviation in decimal notation 13

4.1. Simulation conditions for simulation at low target rates 17
4.2. Simulation conditions for simulation at high target rates 18

4.3. Simulation conditions for simulation with one failing signal 19
4.4. Results of the test series with reduced target rate 20
4.5. Simulation conditions 20

1. Introduction

Over a hundred years ago Ernest Rutherford marked the beginning of modern nuclear physics with his famous scattering experiment. By scattering alpha particles against a thin gold sheet, he proved that atoms have a dense center with a positive charge. Nuclear physics has advanced a lot since then, the accelerators have improved significantly and the elaborated designs are more complicated, but they still follow Rutherford's concept of two colliding particles. [10]
These accelerators still deliver new insights about elementary particles and help us understand physics on a very small scale. In order to achieve very precise and reproducible results, particle accelerators need a wide spectrum of components that work with a high precision and process data in real time. [9]
At the Forschungszentrum Jülich (FZJ) the Institute for Nuclear Physics (Institut für Kernphysik short IKP) operates the Cooler Synchrotron (COSY) accelerator and storage ring. COSY is able to accelerate and store the protons and deuterons pre-accelerated by the Jüoverllich Light Ion Cyclotron (JULIC). [8] The storage ring uses a wide range of real-time data acquisition and processing systems. One of this systems is the so-called "Schneiderbox". It is a beam regulation system responsible for controlling the particle rate hitting the target in a fixed target experiment.
The goal of this thesis is to replace the old regulating system with a modern digital regulation system using the Field Programmable Gate Array (FPGA) on a Red Pitaya single-board. The digital implementation should be able to adjust the particle beam automatically and allow for remote manual configuration.
This thesis contains a short description of the experimental set up, the current rate regulation system, a short introduction to FPGAs and the hardware description language Verilog. It explains the overall idea of the digital rate regulation system. An in-depth discussion about the problems of implementing the system on an FPGA board and how they are solved are given in the thesis. The thesis also includes a short discussion about the advantages, disadvantages of the developed digital rate regulation, possible ways to improve the system and alternative approaches.

2. Physical Background and Set Up

This chapter is a brief summary of the physical background and sets up knowledge needed for the target rate regulation. This includes a basic summary of scattering experiments and white noise signals. It also includes the current rate regulation set up at the COSY accelerator and a brief introduction to FPGA technology and the corresponding programming language.

2.1. Scattering Experiments and Target Rate Regulation

Rutherford's scattering experiment is one of the most influential experiments in the field of physics. The experiment uses a radioactive source emitting α particles. The emitted α particles are used as a particle beam and collide with a thin layer of gold. The set up can be seen in Fig.2.1.

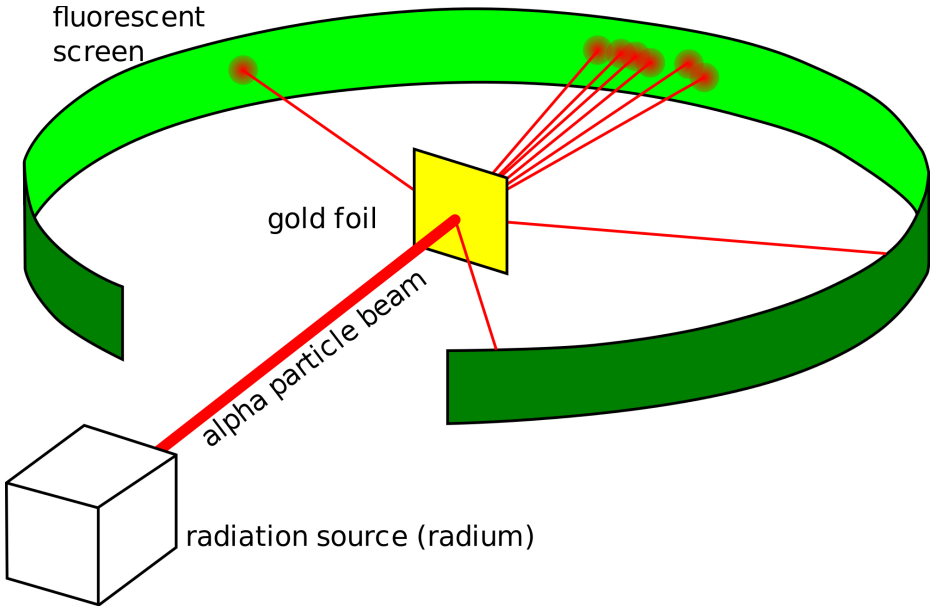

Figure 2.1.: Rutherford's scattering experiment set up [4]

This set up was used by Rutherford to come to the conclusion, that most of the space occupied by the gold atoms is empty and the positive charge is concentrated around a dense center, by evaluating the angle distribution of the transmitted and reflected α particles.

Modern particle accelerators have developed a lot since then. Nowadays the radioactive source has been replaced by particle accelerators, but still they work similar to Rutherford's set up. The accelerated particles are used to collide with atoms or other particles. A distinction is made between fixed target scattering and collider experiments. The fixed target experiment takes a beam of particles with known energy E and momentum p and scatters them at bulk matter, while the collider experiment scatters two particle beams [5]. The relevant scattering experiment at the COSY accelerator is a fixed target experiment. First the particle beam is widened by a kicker with an oscillating electric field. The scattering target is placed behind the widened beam. In order to know the approximate particle rate hitting the target there are 4 detectors behind the target, up, down, left and right. The whole set up is shown in Fig.2.2. The number of particles hitting the target can be adjusted through adjustments on the capacitor voltage. Subject of this thesis is to develop an FPGA system with the capabilities of automatic voltage adjustments, in order to achieve a constant target rate.

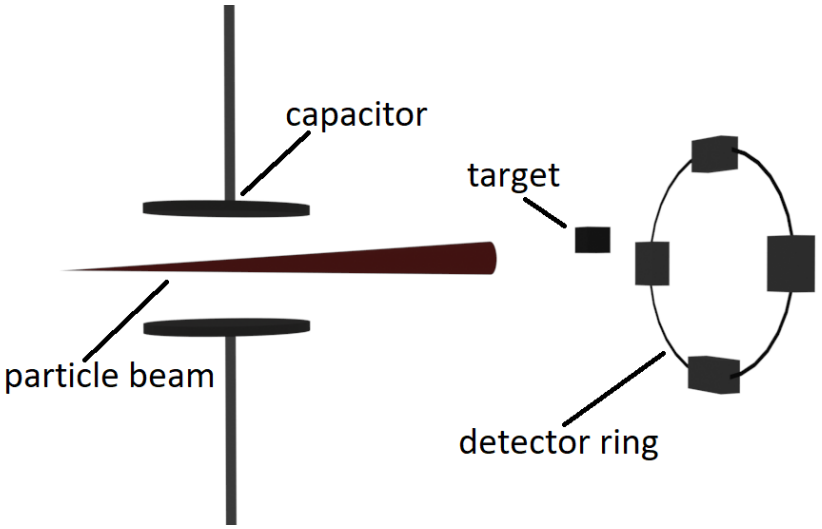

Figure 2.2.: Scattering experiment set up

The purpose of the target rate regulation is to control the rate of the particles hitting the target by evaluating the detector signals and adjusting the amplitude of the capacitor signal.
A higher amplitude at the capacitor leads to the beam being widened more, which then leads to more particles hitting the target.
The detector signals have underlying statistical errors, primarily due to quantum mechanical effects. The total signal count therefore underlies counting statistics, which can be approximated by a Poisson distribution. Therefore the statistical error for k signal counts is $\sqrt{k}$.
With this information, it makes sense to evaluate the detector signals over a long time, in order to achieve a high signal count, which would then result in a low relative

statistical error. However evaluating the signals over a long period of time would mean that the response time of the target rate regulation system is very slow.

2.2. Fourier Transform and White Noise

The signal at the capacitor, that is responsible for widening the beam, is a band limited white noise signal. So it makes sense to briefly talk about Fourier transforms and white noise generation.
The Fourier transform is the mathematical transformation of a time-domain waveform into a frequency-domain representation or vice versa [1]. The time domain shows the signal as a function of time. The frequency domain represents the wave as a continuous spectral function of periodic waves. The Fourier transform of signal $f(t)$ from the time domain into frequency domain is calculated by

$$F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{\mathbb{R}} f(t) e^{-i\omega t} dt \tag{2.1}$$

White noise is a random signal with a constant power spectrum density, meaning that the frequency domain function is constant. A power density spectrum that is constant in $[f_0, f_1]$ and zero elsewhere is called a band limited white noise signal. The power density spectrum of a band limited white noise is shown in Fig.2.3.

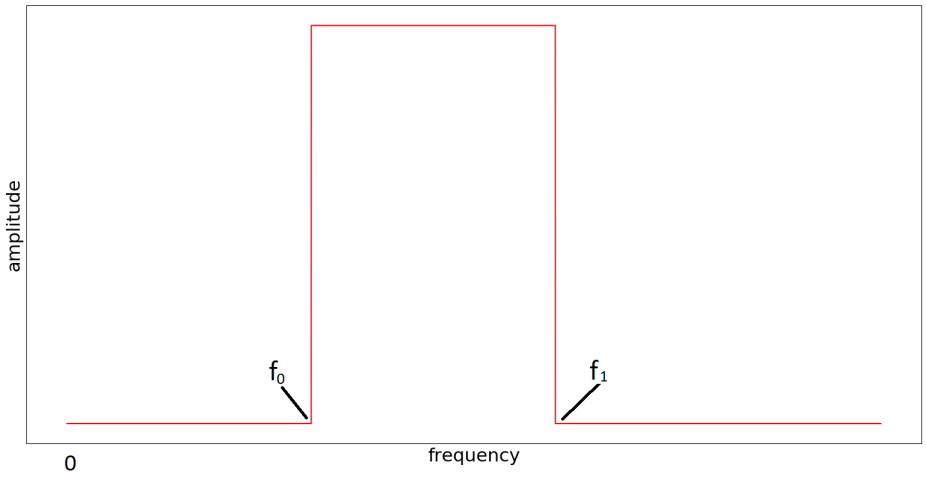

Figure 2.3.: Power spectrum density of an arbitrary band limited white noise signal

2.3. Cooler Synchrotron

This subsection introduces the Cooler Synchrtron COSY, where the design developed in this thesis will be implemented.

The accelerator has a total circumference of 183.47m with two arcs connected by two 40m long linear sections. [9] The Jülich Light Ion Cyclotron is connected to the COSY ring for preacceleration and it has an extraction line for external beam experiments. COSY is used in the fields of hadron, particle and nuclear physics. Designed for experiments in the medium energy range, it offers unpolarized and polarized proton and deuteron beams in the momentum range from 200 MeV/c to 3.7 GeV/c. A special feature is that it offers two different cooling types: electron and stochastic cooling. [3]
A layout plan of COSY is shown in fig.2.4.

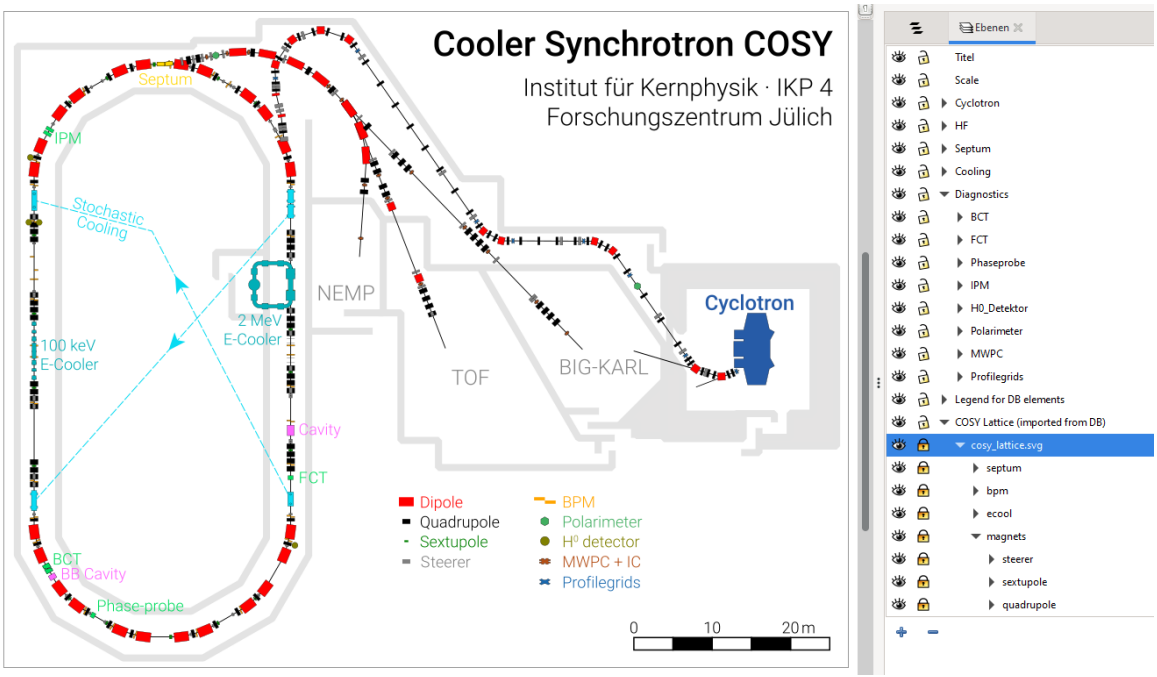

Figure 2.4.: Layout plan of the COSY accelerator [7]

2.3.1. Replacing the Analog Regulation System

As elaborated above, target rate control is very important for scattering experiments. The current system responsible for this at the COSY accelerator is the so-called "Schneiderbox". The signal pulses from the detectors first go through a discriminator that transforms the pulses into ≈ 80ns wide logic signals. The 4 logic signals go through an analog rate meter which can be set up for different integration times between 0.3s to 3s. The measured signal rate then gets converted to a voltage amplitude according to a preset proportionality.
This preset proportionality needs to be adjusted regularly by hand. This is where the digital rate regulation system is supposed to come in. The goal is to implement a system with remote capabilities and automatic rate regulation. A modern FPGA board provides a suitable base for designing a system, that can fullfill these demands.

2.4. FPGA Basics

The FPGA board is the most important hardware part for the digital rate regulation system. FPGAs or field programmable gate arrays are built around an array of logic blocks embedded in a grid of logical interconnections with I/O blocks on the edges for interaction with outside appliances [2, p.5], which allows us to implement unique hardware solutions without developing custom chips. This saves money and developing time for small projects [2, p.1].
The programmable interconnections are a set of wires in the grid of logic blocks that can be used to create arbitrary logic networks by connecting them. Each block consists of programmable logic functions implemented as a 4-6 bit configurable look-up table [2, p.6]. Memory is either embedded in the logic blocks or as external DDR memory. The embedded memory can be implemented as discrete registers, shift registers, distributed RAM or block RAM [2, p.7].

2.4.1. The Red Pitaya Board

The FPGA board used for the digital rate regulation is a Red Pitaya board equipped with a STEMLab 125-14 FPGA . The board itself is a mixture of FPGA and conventional CPU based computer which runs on a Linux OS loaded onto an SD-Card. The Red Pitaya, that will be used, is shown in Fig.2.5.

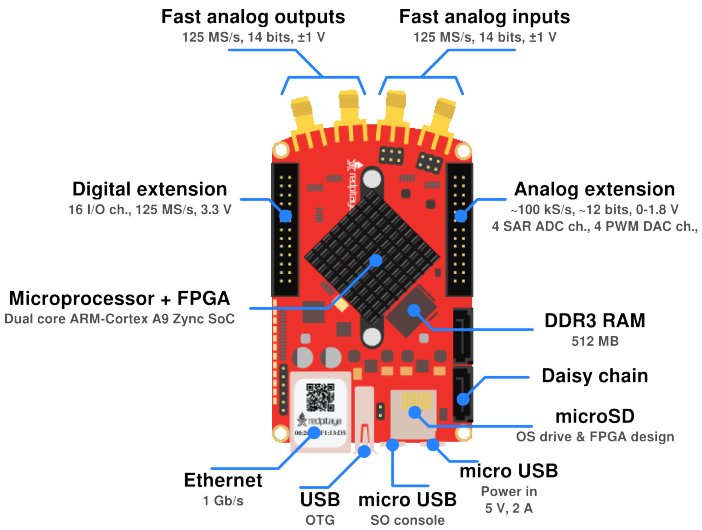

Figure 2.5.: The Red Pitaya board [11]

2.4.2. Verilog

Verilog is the hardware description language (HDL) used for the FPGA programming. HDLs are similar to conventional programming languages with the difference

that HDLs can be used to create digital circuits. HDL programs are also called designs, in order to differentiate between conventional programing languages. When the circuit generator is able to create a gate circuit out of the source code the model is called a synthesizable model, otherwise its called unsynthesizable. The most common reason for a model to be unsynthesizable is that the digital circuit described is too complicated and cannot be automatically written into a realistic circuit. Verilog is very similar to C in syntax [6, p.5ff].
While programming in Verilog is very similar to programming in C there are some constraints that need to be considered, when the design is supposed to be synthesizeable. The main difference is that the operations are all bit by bit, meaning that complicated divisions and floating point numbers can result in a design, that is too complicated to be synthesized. Due to the physical limitations of the FPGA board, ressources like register storage are scarce compared to traditional programming languages.

3. Implemented Design

Creating an FPGA design, that is able to replace manual input reliably comes with a few challenges. The main problem is that the regulation system needs to be able to deal with a desired rate scale of 10^4 and 10^5 particles per second and has to consider edge cases to avoid over regulating the particle beam. This chapter discusses the implemented design, the potential scenarios that were considered and how the design deals with them.

3.1. Fundamental Design

The digital target regulation is split into 2 different modules, which both can be found in the Appendix. The first module counts the number of pulses during the given integration time. It checks the detector rate of each detector and is able to correct for eventual detector errors (further explanation in 3.2). The second module takes the pulse count from the first module and compares it to a given target value and adjusts the output relative to the difference between target value and measured pulses. A program flowchart can be found in Fig.3.1 The IOs are listed in Table 3.1.

port name	**bit size**	**comment**
inputs		
clock	1	125Mhz clock provided by the Red Pitaya
reset	1	reset signal provided by the Red Pitaya
signals 1-4	1	detector signals after discrimination
integration time	32	integration time in multiples of clock cycles
detector threshold	32	minimum amounts of detected particles in given time
mode control	1	switch to choose between counting modes
target/desired rate	32	desired signal count from the detectors
outputs		
amplitude	14	noise amplitude
signal count	32	amount of signals during int time

Table 3.1.: Port listing for the whole design

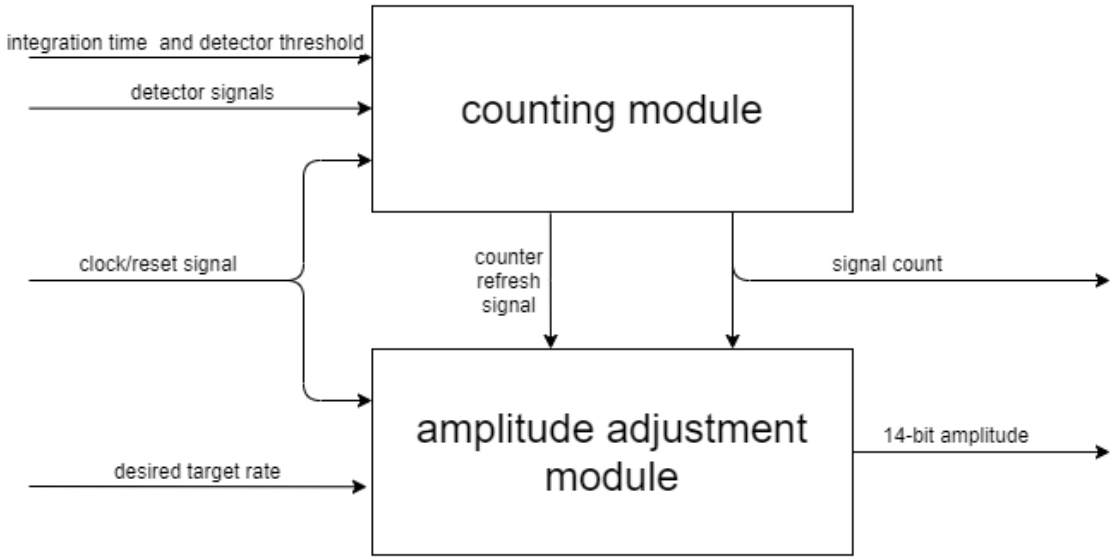

Figure 3.1.: Flowchart of the FPGA modules

3.2. Counting Module

This module is responsible for counting the signals in a given integration time and providing the total number of measured particles. The design has two modes to count the input signals. The first counts every signal separately and adds them all together after the integration time has passed while the second bundles up all the signals into one wire counting the signals on the single wire. The program can check for potential detector errors and adjust accordingly.

FPGAs are not able to directly detect signal changes and can only periodically check the state of a signal input. The detector signals from the discriminator are logical one bit signals with the logical one representing a particle hitting the detector. The signal length is approximately 80ns.

The clock provided by the Red Pitaya has a frequency of 125MHz, meaning that using this clock the design would check the signal inputs every 8ns. This is the first problem. The design would check every 8ns and count multiple signals for one 80ns long signal. In order to avoid multiple counts, the design creates a wire responsible for detecting positive flanks on the input signal. This is done by using a 4-bit shift register on the signal input. When a positive flank occurs the shift register will have the bits set to 1-1-1-0 with 0 being the last bit on the shift register. This can also be seen in Fig.3.2. It does cause a delay of 3 clock cycles in detecting the incoming signal, but this is irrelevant for the counting process.

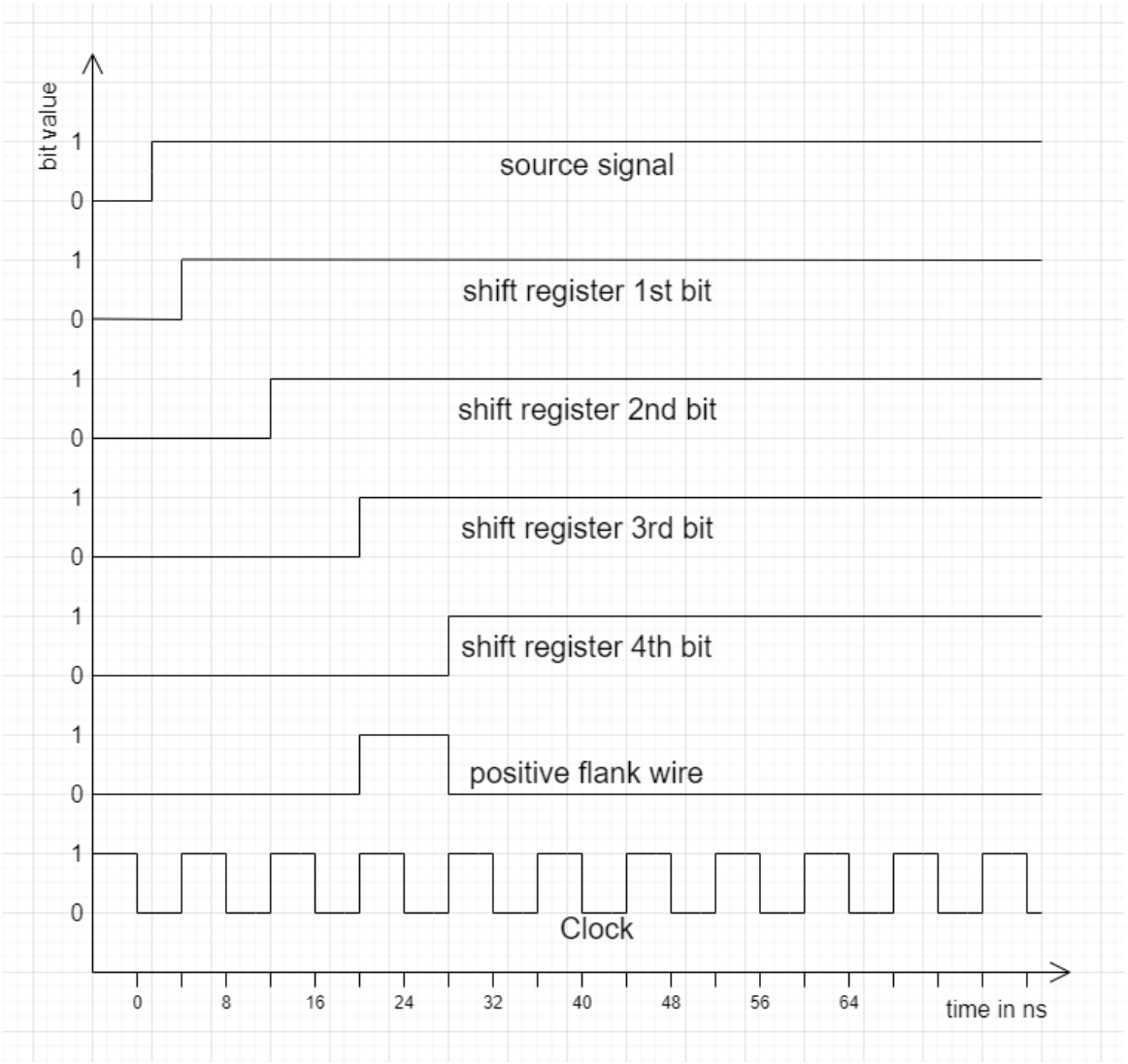

Figure 3.2.: Positive flank detection with a 4 bit shift register

The clock is also used to figure out when the integration time has passed. The design creates a 32-bit register for a clock counter, which simply counts the amount of cycles the clock has been through, it checks if the clock counter is equal to the input *int_time*. If it is equal to the integration time the clock counter is reset and signals the rest of the design that the integration time has passed through a clock reset register.
The counting algorithm has two separate counting modes. It can either count each signal on its own and add them up at the end of the integration time or bundle the 4 signals into one combined signal and count the combined signal. This combined signal is made by setting a wire, which is set to one whenever one of the signal inputs is one.
The second mode loses, accuracy because overlapping signals from detectors can appear to be only one signal, however the loss in accuracy should be negligibly small.

Assume that each detector has the same signal rate f and a signal length of τ = 80ns. With the assumption of independent random signals, a conservative worst case overlap rate f_o of two signal inputs overlapping can be calculated by

$$f_o = 2 \cdot \tau \cdot f^2 \quad . \tag{3.1}$$

This rate describes a worst case scenario of how often a signal is lost due to overlaps. Each of the 4 signal inputs could overlap with one of the other inputs which results in 6 different possible overlapping cases. So the total overlap rate f_{total} is given by

$$f_{total} = 6 \cdot f_o = 12 \cdot \tau \cdot f^2 \quad . \tag{3.2}$$

The expected signal rate is somewhere in the range between 10^4 and 10^5, while τ is on a scale of 10^{-8}s. So the total overlap rate would be in the range of 10^0 and 10^2 s^{-1}. This idealized view would at worst result in ≈ 100 missed signals per second. This missed signal rate is negligible compared to the expected statistical errors discussed in the previous chapter.
With every clock cycle the program first checks for the counter reset register. If it signals that the the integration time has ended, the design then checks if the signal count is above the threshold. If it is, the counter value gets stored in a storage register and the counter resets. If it is not the storage value is set to zero and the fail switch register is set to one signalling a defective signal input.
At the end the design checks the mode control input and assigns the signal count of the chosen counting mode to the signal count output. It also checks the fail switches of each detector, if one signal has not enough counts, the design throws out the opposing detector count too and just doubles the amount of the remaining two signals. This is done because multiplying by two is very easy in bit notation. The error state of each signal input can be accessed through an implemented bug fixing register output.

3.3. Amplitude Adjusting Module

This module takes the output of the counting module and calculates the difference between the signal count and the desired value. The output of the module is the amplitude of the regulating white noise and gets adjusted relative to the difference of the particle count to the desired value. One important property is that the module can detect certain signal problems and does not over adjust the amplitude.
Every time the signal count is updated the design calculates the absolute difference between the signal count and the desired rate. One of the main problems is that the rates can vary from a 30.000 particles per second to rates in the range of $10^5 s^{-1}$. An absolute difference of for example 10.000 could be negligible for a high signal rate while very significant for a low signal rate. The logical conclusion would be to work with relative differences, but FPGA designs cannot handle complicated division calculations very well, making it unreliable to just calculate the relative differences. However FPGA designs allow for an easy way to divide by two. The numbers are all

stored in a binary notation, so removing the least significant bit (lsb) is the same as dividing by two and discarding the remainder. This means that the index of the most significant bit (msb) is also the number of times one can divide by two and discard the remainder without going below one. It is basically the same as finding out the order of magnitude of a number written in a decimal system(10^d), but in a binary notation (2^d). In the decimal system the difference between the order of magnitude of two numbers can give information about the relative difference between two numbers. Similar to this, the difference in msb of two binary numbers can also give information about the relative difference between two numbers.

Suppose $x > y$ with $x, y \in \mathbb{N}$ and $n = \text{msb}_x - \text{msb}_y$, then:

$$y \propto \frac{1}{2^n} x \quad . \tag{3.3}$$

The design uses this method to approximate the relative difference between signal and desired rate by taking the difference between the msb of the desired rate and the msb of the absolute difference. Table 3.2 shows the scale of this method.

difference in msb	approximate scale of signal rate/desired rate
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
....	....
n	$\frac{1}{2^n}$

Table 3.2.: Scale for various msb differences between desired rate and signal rate deviation in decimal notation

This results in a step function for the amplitude correction. An example function is plotted in Fig.3.3. This method has the problem of number neighbours, that are on the verge of an msb increment. In a decimal system an example would be 999 and 1000, both are only separated by one but have different orders of magnitude. The same is true for numbers like 16 and 15 in binary notation. However this should not be a major problem for the rate regulation design, because if the difference between signal count and desired value is near the desired value the design has to adjust a lot anyway and it would at worst result in a lag of one integration time for the regulation to get the rate back into the target territory.

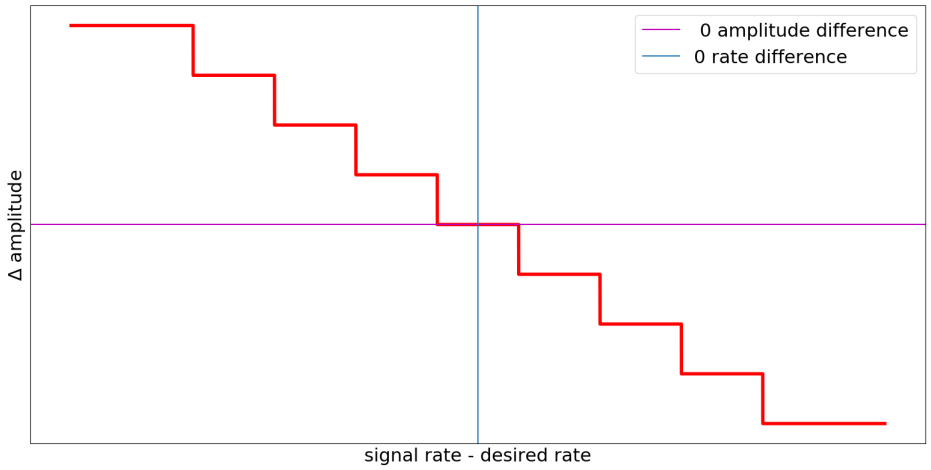

Figure 3.3.: Sketch of the adjusting step function

The rate regulation design has a confidence interval. If the signal rate is in this interval around the desired rate, the design stops adjusting the amplitude and accepts the signal rate as right. A confidence interval is needed because the signal rate underlies statistical fluctuations. Without it the program would oscillate around the target amplitude by constantly adjusting in different directions, which in turn would cause for unwanted frequencies to show up in the band limited white noise signal.
This problem is solved by ignoring rate differences with $\Delta\text{msb} > 4$, where $\Delta\text{msb} = \text{msb}_{target} - \text{msb}_{dif}$. The biggest possible confidence interval can then be calculated by:

Suppose that $x, y, z \in \mathbb{N}$ and $d \in \mathbb{N}$ with x being the smallest number with msb= d, y being the largest $(d-5)$-bit number and $z = y + 1$, then (in binary notation):

$$x = \underbrace{100000000....}_{\text{d}}$$
$$y = 00000\underbrace{1111....}_{\text{d-5}}$$
$$z = 0000\underbrace{10000....}_{\text{d-4}}$$

Multiplying a binary number by two or dividing a binary number by two is a simple bit shift, then:

$$z = \frac{1}{2^4}x \implies y < \frac{1}{2^4}x \tag{3.4}$$

This means that the biggest possible confidence interval is at ± 6.25 percent of the target value. The confidence interval can be adjusted to be smaller by choosing integration times that will result in target counts that are not on the verge of changing msb.
The output amplitude of the module is limited to a number between 0 and 16383 (the highest 14-bit number). However the design has a self set lower and upper boundary of 1000 and 15900 in order to stop itself from overflowing and over regulating.
The confidence interval and the amplitude boundaries can be changed in the firmware by adjusting the corresponding parameters.

3.4. The 'generate' Function

The Red Pitaya comes with a few preinstalled functions written in C. One of them is the generate command which is used to generate predefined periodic signals . The beam regulation requires a band limited white noise signal. In order to accommodate that, the generate function was extended with the "arbitrary" signal setting. This setting allows the user to upload an arbitrary waveform in a .csv file.
The generate function stores the wave parameters like amplitude and frequency in registers. Changing the entries in these registers will also change the properties of the output signal. The developed design changes the amplitude register of the generate function with the amplitude output.

4. Testing and Simulations

This chapter discusses the results of tests and simulations run on the design. These tests and simulations are supposed to simulate real world cases as close as possible and test the limits of the design by testing the response to input, that is different from the expected input .

4.1. Simulations with Vivado Testbench

The HDL IDE Vivado Design Suite offers a simulation tool. The only thing needed is a testbench source, which simulates the input signals and reads the output. The main advantage of the simulation is, that it is very easy to customize and the ability to monitor every register closely makes bug fixing easier. The disadvantage is, that the simulation can only simulate very idealized cases.
The main goal of the simulations is to verify the functionality of the design in the expected signal rate range and that the implemented features like detector failure function properly. In order to achieve this series of simulations are done with different conditions.

4.1.1. Simulations at 30.000 Signals per Second

This simulation is supposed to ensure the functionality of the rate regulation at the lower particle range by setting a target rate of 30.000 signals per second. A series of simulations is done by simulating a time frame of one integration time with a range of signal rates. At the end the difference between signal rate and target rate and the amplitude output is recorded for analysis. The testing conditions can be found in Table 4.1.

target rate	30000 s^{-1}
signal rate	10000-50000 s^{-1}
integration time	0.25s , 2.5s

Table 4.1.: Simulation conditions for simulation at low target rates

Simulating 2 s on the design can take a relatively long time. That is the reason why the simulation for this time frame is done only a few times to show that it shares the same functionality as the simulations at 0.25s. The results are shown in Fig.4.1

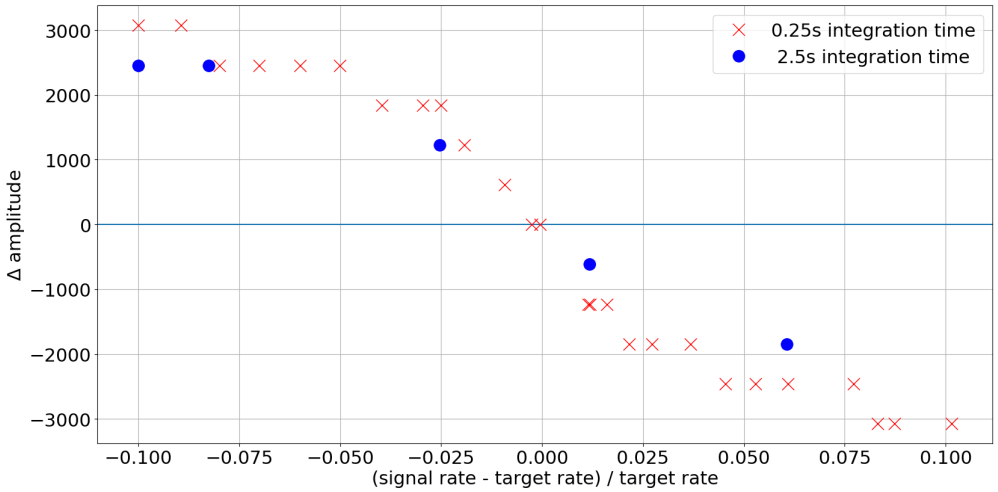

Figure 4.1.: Simulation results for a target rate of 30.000 signals per second

Overall one can observe that the design regulates the amplitude according to the expected step function. The step function for the long integration time seems to have different step intervals. This is probably due to the difference in bit representation of the numbers, as elaborated in the previous chapter.

4.1.2. Simulations at 200.000 Signals per Second

The purpose of this simulation series is similar to above but this time its to prove the functionality of the design at high frequency rates. The simulation procedure is similar to the simulation at 30.000 signals per second. The simulation conditions can be found in Table 4.2.

target rate	200000 s^{-1}
signal rate	50000-350000 s^{-1}
integration time	0.25s , 2.5s

Table 4.2.: Simulation conditions for simulation at high target rates

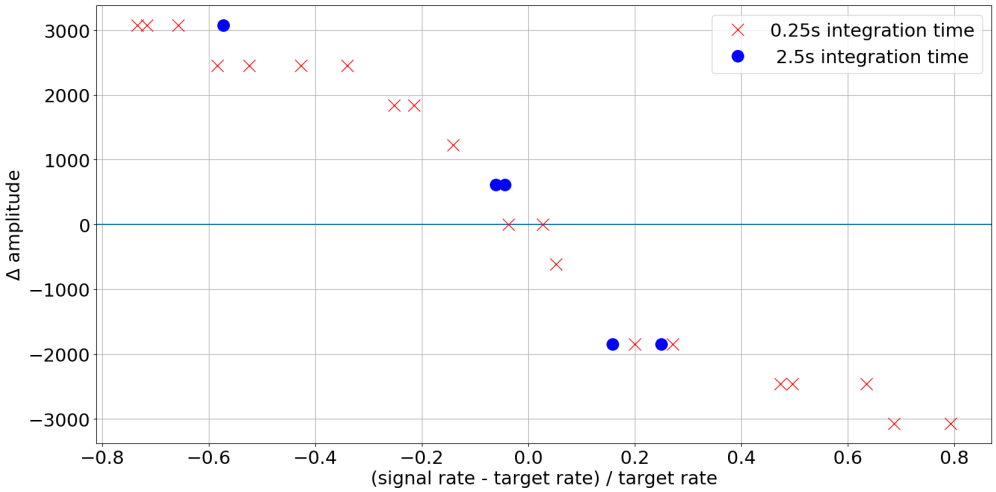

Figure 4.2.: Simulation results for a target rate of 200.000 signals per second

The simulation results are shown in Fig.4.2. Similar to the results above the simulations show the expected step function with different step width for different integration times.

4.1.3. Simulations at 80.000 Signals per Second

The simulations at a target rate of 80.000 signals per second are done to verify the functionality of the design in special conditions and that the implemented features are working as planned. Multiple special cases were considered in the simulation tests. The first case is one of the input signals falling below a certain threshold value. The design is supposed to detected the detector signal rate being too low and adjust the signal count accordingly. In order to verify this feature, four simulations are done. With each simulation one signal input rate is significantly reduced. The simulation conditions can be found in Table 4.3

detector count threshold	500 s^{-1}
signal rate per detector	80000 s^{-1}
integration time	0.25s

Table 4.3.: Simulation conditions for simulation with one failing signal

	simulation 1	simulation 2	simulation 3	simulation 4
signal count 1	4	4563	4563	4563
signal count 2	4648	5	5186	5114
signal count 3	5165	5165	53	5422
signal count 4	5800	6082	6082	19
output counter	21930	22494	19498	19354

Table 4.4.: Results of the test series with reduced target rate

The results in Table 4.4 show that the design is able to detect the low signal rate on one source and adjust the output counter accordingly, by also ignoring the signal rate of the opposite source and doubling the the signal count of the remaining signal counts.

The next feature that needs to be verified is that the design stops regulating the amplitude when there is no change and stops itself from adjusting the amplitude register at preset boundary thresholds. This is important, in order to stop effects like bit overflows. To test this functionality the simulated signal sources are either much lower or much higher than the expected signal rate and they do not change throughout the whole simulation process. The testing conditions are shown in Table 4.5.

target rate	80000 s^{-1}
signal rate	2000 , 150000 s^{-1}
integration time	0.25s
simulation duration	3s

Table 4.5.: Simulation conditions

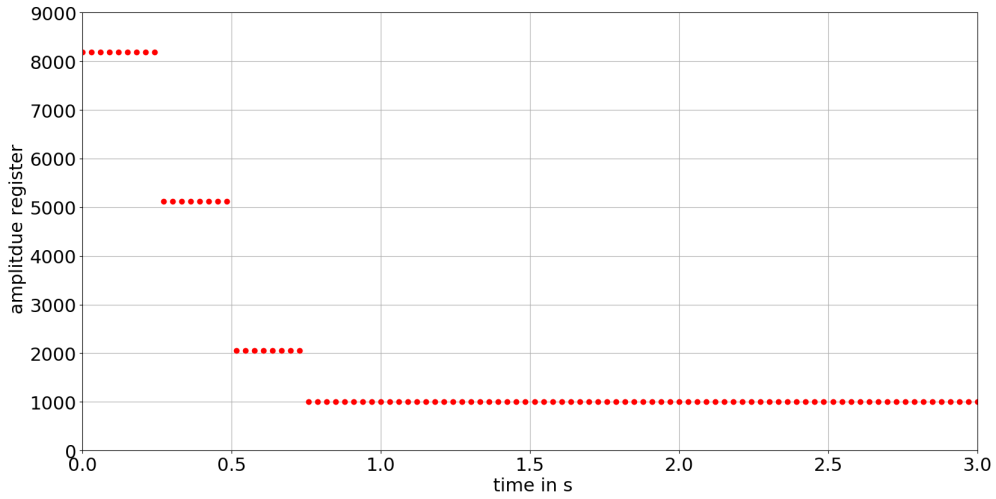

Figure 4.3.: Test results with a constant signal rate of 150.0000 signals per second

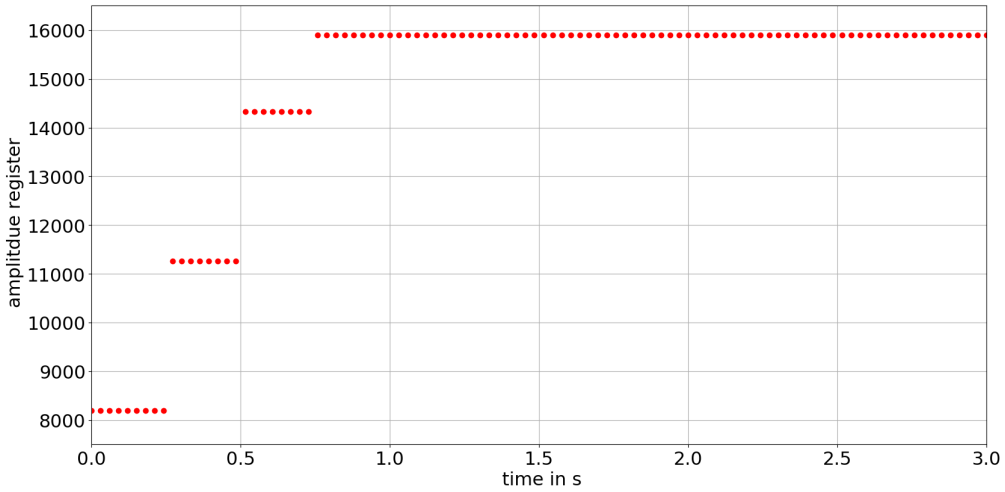

Figure 4.4.: Test results with a constant signal rate of 2000 signals per second

The simulation results are shown in Fig.4.3 and Fig.4.4. The design stops adjusting the amplitude register at the threshold values of 1000 and 15900. This response was expected as it stops the design from over regulating.

4.2. Tests with Frequency Generator

This test is done with the help of a signal generator, which allows us to test the design with real signals but still have a good amount of control over signal rate and signal

width. The main goal of this test series is to verify the design's functionality on a real FPGA board with real signals and the correct regulating response to signal changes. The set up is very simple. The generator has a signal output which is split by T connectors in 4 signal outputs. These signal outputs are connected to the Red Pitaya with LEMO cables. The set up is shown in Fig.4.5.

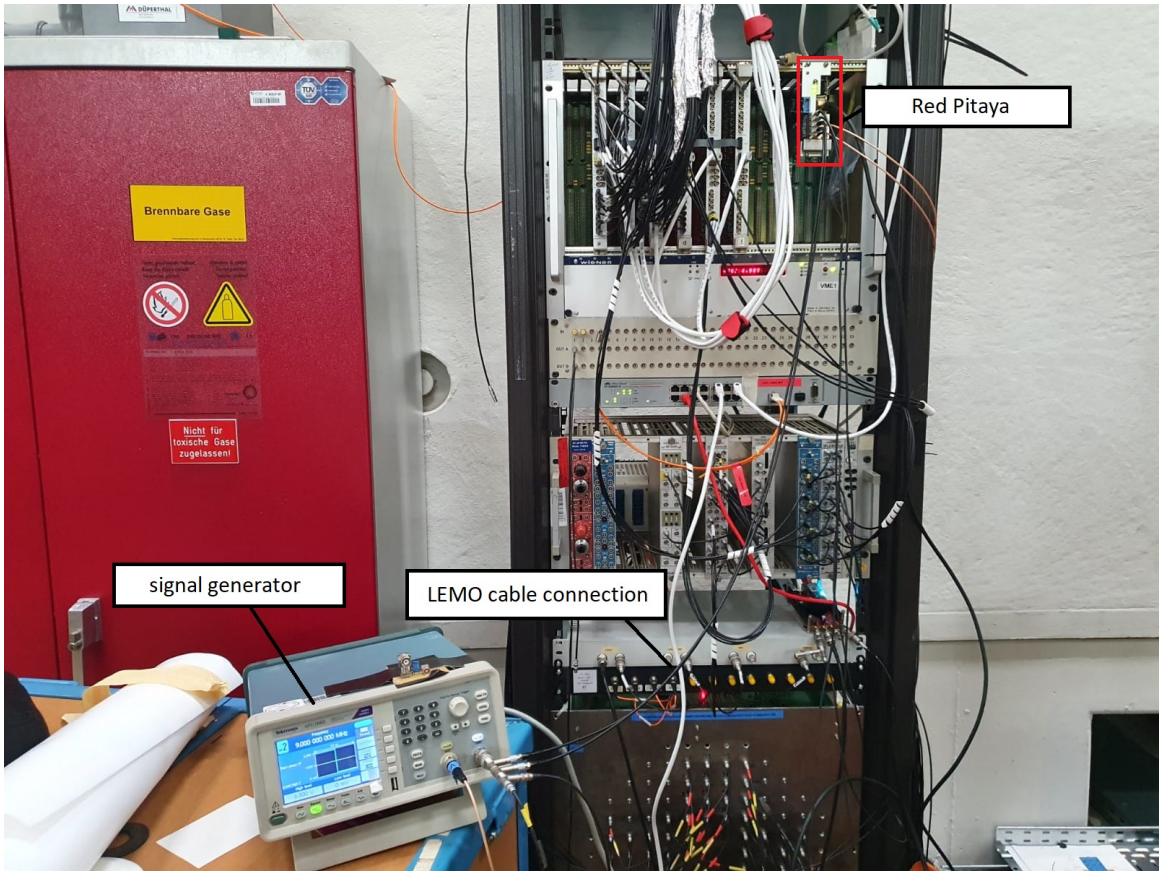

Figure 4.5.: Set up with the signal generator

The first test is done with a generator frequency of 7MHz. The generated frequency is much higher than the expected signal frequency from the detector ring. But due to constraints on the signal generator this frequency is needed to simulate the correct detector signal width of approximately 80ns. The second test is done with a signal rate of 400kHz, which is supposed to simulate the detector rate.

For both tests the generator is set to the right frequency and the Red Pitaya is given the integration time of approximately 2.1s and the target signal count, calculated from signal rate and integration time. Afterwards the generator frequency is changed in both directions. The amplitude register and signal count register is read every 6 seconds. The results for the 7Mhz test is shown in Fig.4.7 and the results for the 400kHz test is shown in Fig.4.6.

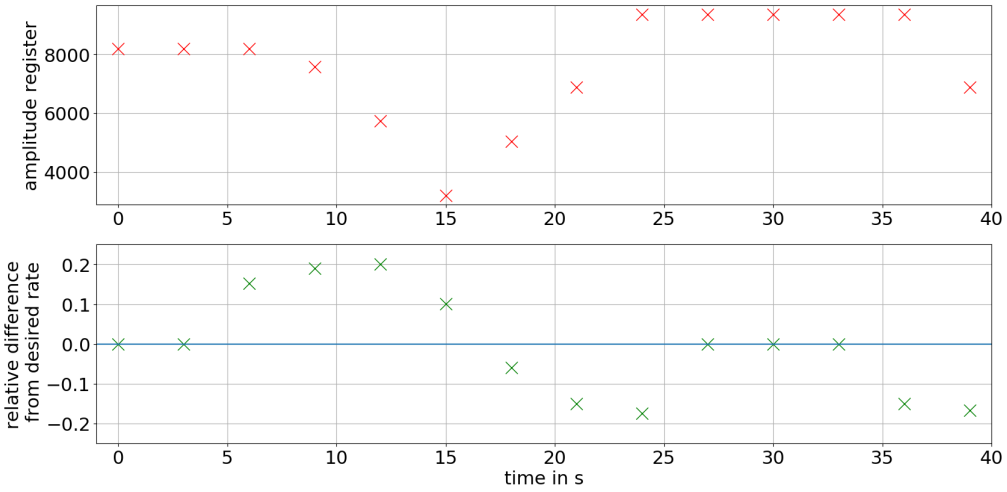

Figure 4.6.: Test at 400khz, the upper plot describes the behaviour of the amplitude register, the lower plot describes the difference between measured signal rate and the desired rate

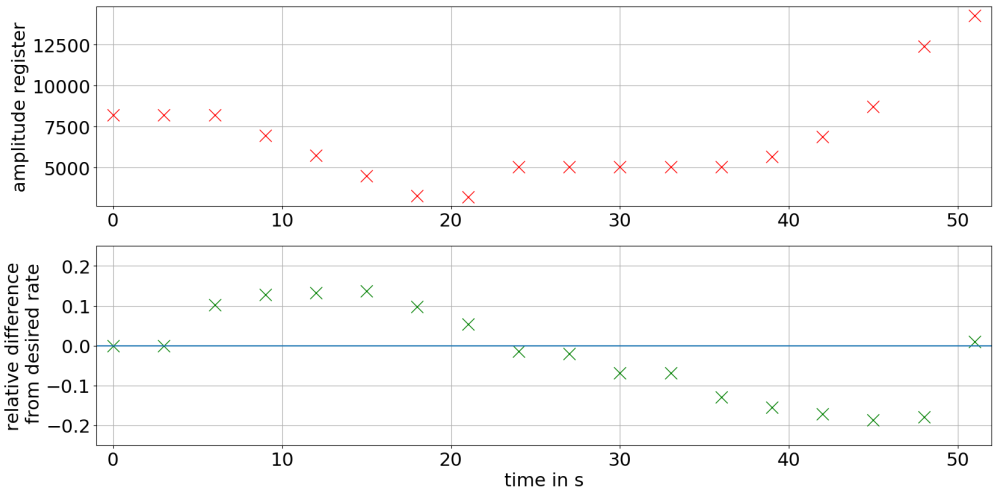

Figure 4.7.: Test at 7Mhz, the upper plot describes the behaviour of the amplitude register, the lower plot describes the difference between measured signal rate and the desired rate

The tests show the expected amplitude regulation behaviour. The design stops adjusting the amplitude register when the signal count stays in a certain confidence interval around the target count. However it seems, that the design lags behind one integration time while adjusting.

4.3. Tests with Discriminator

The goal of this test is to verify the ability of the design to properly detect the discriminator signal inputs. During the time of writing the thesis, COSY was not operating, so the only possible way to get signal inputs was with background noise from the detectors, which can have multiple different sources like cosmic rays, and by connecting the detector with the discriminator. The set up is shown in Fig.4.8.

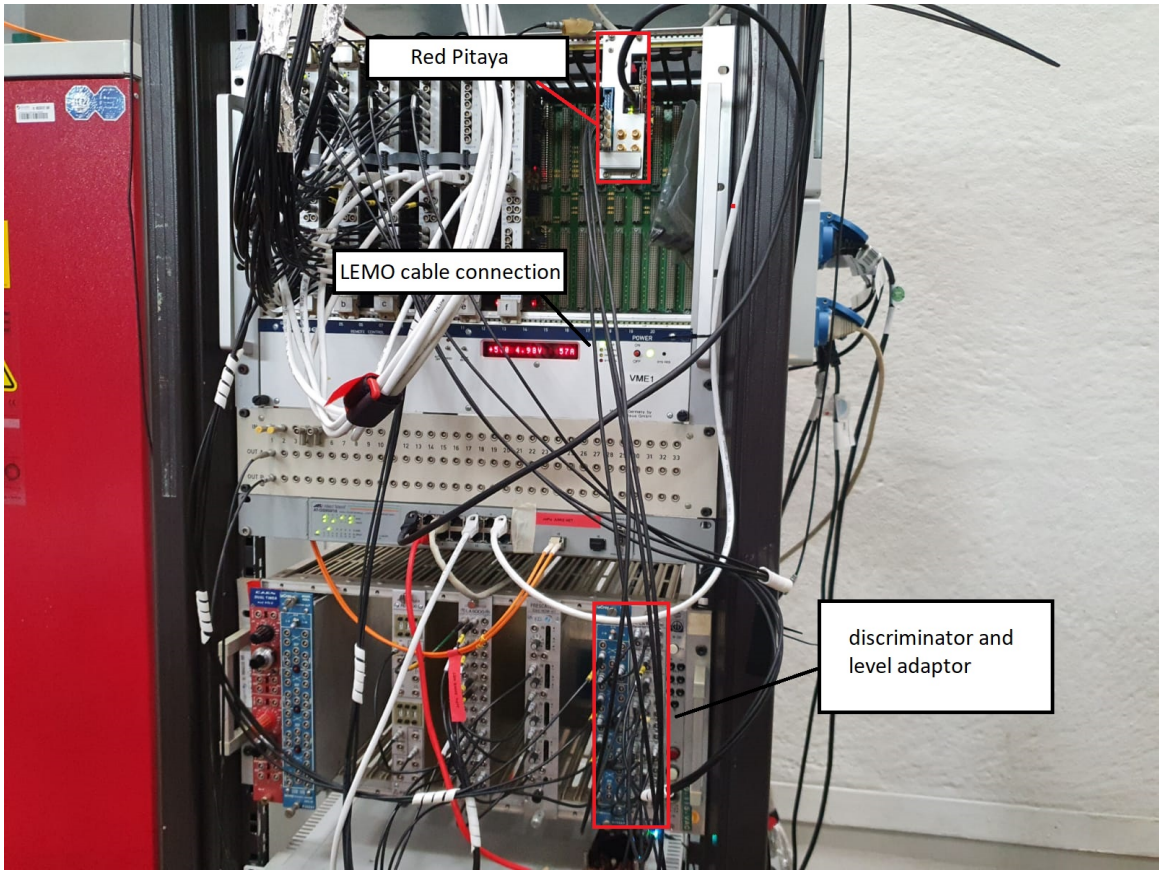

Figure 4.8.: Set up at the discriminator

Two tests are done with and without generator. Due to the low rate of noise captured by the detectors a long integration time of 5s is chosen. The generator is set to a frequency of 20kHz. The results of these tests are shown in Fig.4.9 and Fig.4.10.

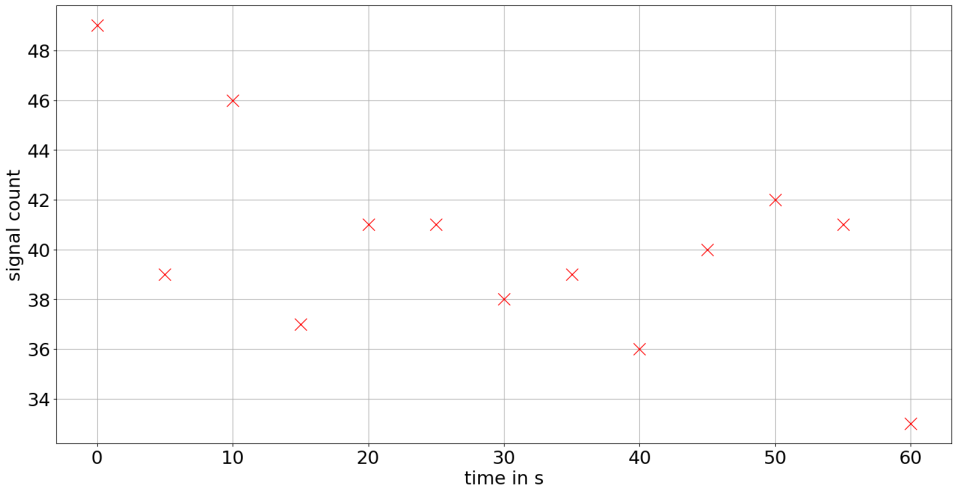

Figure 4.9.: Test results without generator

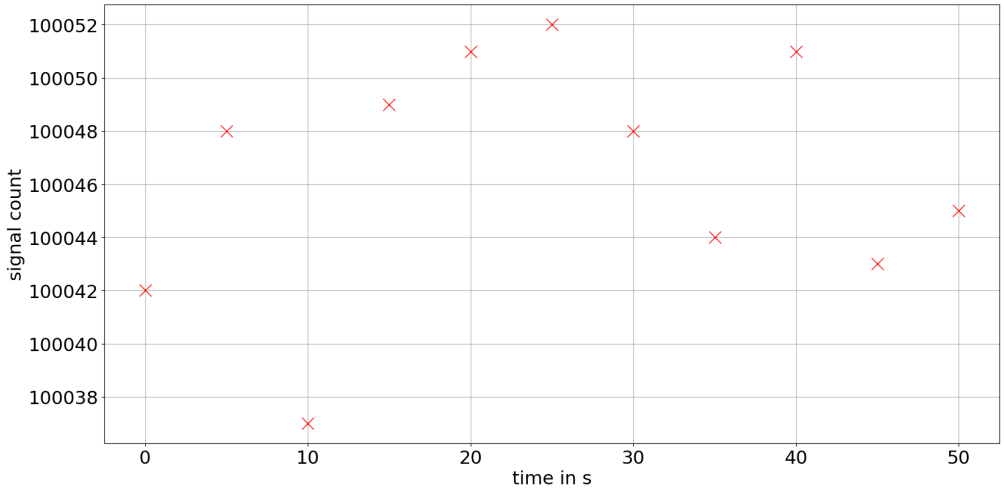

Figure 4.10.: Test results with generator

The results are as expected. The background noise picked up by the detectors seems to be distributed randomly. The results with generator also suggest that the design is able to count every incoming signal, because a signal frequency of 20kHz over 5s would result in 100.000 signals counted.

5. Summary and Outlook

The last chapter summarizes the overall results of the thesis and provides an outlook for further improvements. It also includes a short discussion about possible different approaches.
Goal of this thesis was the development of a new rate regulation system. The result of the thesis is a digital rate regulation system running on an FPGA board.
The tests and simulations performed in this thesis show that the developed design is able to appropriately respond to signal rate changes and the different special cases, which were considered. The developed design also allows for easy remote access for manual tweaking once it is set up. Due to the modular structure it is relatively easy to adjust or split the system if needed.
However due to the lack of tests with real beam signals it is not clear if further tweaks are needed. Possible problems could be that the provided amplitude adjustment is not exact enough. It is also not entirely clear how well the amplitude adjustment works with the white noise signal generator.
There are two other approaches, that could be considered for replacing the current target rate regulation. The first one would be to use a CPU based programming language like C to process the signals and adjust the beam. This approach has the benefit that mathematical operations like division are not a problem to handle allowing for adjustment through a mathematical function based on the relative difference between target rate and signal rate. However this approach requires deeper knowledge of how CPUs prioritize calculations and processes.
The second approach would have been replacing the current analogue rate regulation system with a digital version of itself. This would have the benefit of being similar to the present system and would be safer to implement in the regulation circuit. This would result in only minor advantages compared to the old system, as one could do everything remotely. But the regulation itself would not be automatic.
Overall, the developed design can be used as a base structure for further development and improvements of the target rate regulation system at the COSY accelerator.

Bibliography

[1] "Fourier Transform". In: *Encyclopedia of Neuroscience*. Ed. by Marc D. Binder, Nobutaka Hirokawa, and Uwe Windhorst. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1626–1626. ISBN: 978-3-540-29678-2. DOI: 10.1007/978-3-540-29678-2_1833. URL: https://doi.org/10.1007/978-3-540-29678-2_1833.

[2] S. Churiwala. *Designing with Xilinx® FPGAs*. Springer, 2017.

[3] *COSY (Cooler Synchrotron) - Helmholtz Gemeinschaft*. URL: https://www.helmholtz.de/forschungsinfrastrukturen/beschleuniger/cosy-cooler-synchrotron/.

[4] *Experimental Evidence for the Structure of the Atom - George Sivulka*. URL: http://large.stanford.edu/courses/2017/ph241/sivulka2/.

[5] Brigitte Falkenburg. "Scattering Experiments". In: *Compendium of Quantum Physics*. Ed. by Daniel Greenberger, Klaus Hentschel, and Friedel Weinert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 676–681. ISBN: 978-3-540-70626-7. DOI: 10.1007/978-3-540-70626-7_191. URL: https://doi.org/10.1007/978-3-540-70626-7_191.

[6] Harald Flügel. *FPGA-Design mit Verilog*. Oldenbourg Wissenschaftsverlag, 2011.

[7] *IKP intern GitLab - GitLab*. URL: https://gitlab.cce.kfa-juelich.de/.

[8] *Institut für Kernphysik - JULIC*. URL: https://www.fz-juelich.de/ikp/ikp-4/DE/Leistungen/_doc/Beschleunigerbetrieb/betriebJULIC_node.html.

[9] Taline Kehlenbach. "Real Time Integration of MHz Signal using a Fast Programmable Gate Array". Bachelor Thesis. RWTH Aachen III. Physikalisches Institut B, 2020.

[10] Hans Paetz gen. Schieck. *Atome, Kerne, Quarks - Alles begann mit Rutherford*. Springer Spektrum, 2019.

[11] *The Red Pitaya Board*. URL: https://marceluda.github.io/rp_lock-in_pid/TheApp/RedPitaya_board/ (visited on 08/21/2021).

A. Appendix

A.0.1. Verilog Code for Counting Module

```
module drr_counter(
counter_reset, //out: couner reset / int time over signal
ctrl_sw , // in: switch to detemine which counting method to use
 clk_i  ,// clock
 signal1 ,  // in : signals ins
 signal2 ,
 signal3 ,
 signal4 ,
 rstn_i , // RP reset
 target_value ,
  int_time , // in: integration time
  counter, //out: signal count
  out_debug , //out : debug info
  threshold, //in: signal count threshold that each detector needs to surpa
debug, //out debug
clk_cnt //out debug info about clk cnt

) ;

// input explanation: ctrl_sw is for switching between 4 seperate counters
//threshold is the minimal amount of particles that should be counted by eac
//clk_i and rstn_i are the RP intern clock and reset signal
//signal X is the input of detector X
// int_time integration time

//output:
//out is the output for the register that controls the amp
//counter reset is the reset signal that int_time is over and counters got
//counter is the total count of signals

input clk_i;
input[31:0] threshold;
input ctrl_sw;

input signal1 ;
```

```
input signal2 ;
input signal3 ;
input signal4 ;

input rstn_i ;

input [31:0] int_time;

input [31:0] target_value;



output[31:0] counter;
output[31:0] clk_cnt;
output[31:0] debug;

output counter_reset;

output[13:0] out_debug;









wire reset;

reg fail_sw1 , fail_sw2 , fail_sw3 , fail_sw4;
// fail switches 1 = detector signals are to low

wire signal; // signal wire for or combination for counting method

wire rstn_i ;

wire clk_i;

wire [31:0] int_time; //

reg[31:0] counter;
```

```
reg[31:0] clk_count;

reg counter_reset;

reg[31:0] counter1;
reg[31:0] counter2; //signal counter for each detector
reg[31:0] counter3;
reg[31:0] counter4;

reg[31:0] counter1_unclip; // temp coiunter storage for when int_time is ov
reg[31:0] counter2_unclip;
reg[31:0] counter3_unclip;
reg[31:0] counter4_unclip;


reg[4-1:0] signal1_shift;
reg[4-1:0] signal2_shift; // shift register for edge detection (for more in
, fig: positive flank detection
reg[4-1:0] signal3_shift;
reg[4-1:0] signal4_shift;

reg signal1_edge ;
reg signal2_edge ; //edge detetction register see above
reg signal3_edge ;
reg signal4_edge ;


assign reset = rstn_i;


//assign signal1_edge = signal1_shift[0] & ~signal1_shift[3];
//assign signal2_edge = signal2_shift[0] & ~signal2_shift[3];
//assign signal3_edge = signal3_shift[0] & ~signal3_shift[3];
//assign signal4_edge = signal4_shift[0] & ~signal4_shift[3];

assign signal1_edge = (signal1_shift == 4'b0111);
assign signal2_edge = (signal2_shift == 4'b0111);
assign signal3_edge = (signal3_shift == 4'b0111); // edge detection above
assign signal4_edge = (signal4_shift == 4'b0111);

wire signal_or;
wire signal_or1;//or combination of signals for counting method 2
wire signal_or2;
```

```
reg[31:0] clock_cylces;

assign signal_or1 = (signal1_shift[3] | signal2_shift[3]); // or
assign signal_or2 = (signal3_shift[3] | signal4_shift[3]);

assign signal_or = (signal_or1 | signal_or2);

reg[4-1:0] signal_or_shift;
reg signal_or_edge;

assign signal_or_edge = signal_or_shift == 4'b0111; //edge detection li

reg[31:0] debug;



initial begin
        counter_reset <= 0;
        counter <= 0; //initalize some values
        fail_sw1 = 0;
        fail_sw2 = 0;
        fail_sw3 = 0;
        fail_sw4 = 0;
        clock_cylces =  31'h1312D0;
end



always @(posedge clk_i) begin

    if (reset == 1) begin

        clk_count <= 0 ;
        counter_reset <= 0;  //reset values

    end


    else if (int_time == clk_count) begin

        clk_count <= 0; //int time over reset counter
        counter_reset <= 1 ;
```

```
    end


    else begin

        clk_count <= clk_count + 1;
//count clock cycles
        counter_reset <= 0;

    end


    signal1_shift <= {signal1_shift[2],signal1_shift[1],signal1_shift[0],sig
    signal2_shift <= {signal2_shift[2],signal2_shift[1],signal2_shift[0],sig
    signal3_shift <= {signal3_shift[2],signal3_shift[1],signal3_shift[0],sig
    signal4_shift <= {signal4_shift[2],signal4_shift[1],signal4_shift[0],sig
    signal_or_shift <= {signal_or_shift[2],signal_or_shift[1],signal_or_shi

end

always @(posedge clk_i ) begin

if ( reset == 1'b1) begin

        counter1 <= 0;
                        //reset values

end

if (counter_reset) begin

    if (counter1 > threshold) begin
        counter1_unclip <= counter1;
//if above theshold accept count
        fail_sw1 <= 0;
        end

    else begin
        counter1_unclip <= 0;
        fail_sw1 <= 1;
        end

    counter1 <= 0;
```

```
    end

if (signal1_edge) begin

counter1 <= counter1 + 1;  //count each signal edge

end
end

always @(posedge clk_i ) begin

if ( reset == 1'b1) begin  //same as above.

        counter2 <= 0;


end

if (counter_reset) begin

    if (counter2 > threshold) begin
        counter2_unclip <= counter2;
        fail_sw2 <= 0;
        end

    else begin
        counter2_unclip <= 0;
        fail_sw2 <= 1;
        end

    counter2 <= 0;

    end

if (signal2_edge) begin

counter2 <= counter2 + 1;

end
end


always @(posedge clk_i ) begin          //same as above
```

```
if ( reset == 1'b1) begin

        counter3 <= 0;


end

if (counter_reset) begin

    if (counter3 > threshold) begin
        counter3_unclip <= counter3;
        fail_sw3 <= 0;
        end

    else begin
        counter3_unclip <= 0;
        fail_sw3 <= 1;
        end

    counter3 <= 0;

    end

if (signal3_edge) begin

counter3 <= counter3 + 1;

end
end



always @(posedge clk_i ) begin       //same as above

if ( reset == 1'b1) begin

        counter4 <= 0;


end

if (counter_reset) begin

    if (counter4 > threshold) begin
```

```
        counter4_unclip <= counter4;
        fail_sw4 <= 0;
        end

    else begin
        counter4_unclip <= 0;
        fail_sw4 <= 1;
        end

    counter4 <= 0;

    end

if (signal4_edge) begin

counter4 <= counter4 + 1;

end
end

reg[31:0] counter_or_running;
reg[31:0] counter_or;

always@(posedge clk_i) begin    //similar as above just with the or sig

        if (reset == 1 ) begin

                counter_or_running <= 0;

        end else begin
                if (counter_reset) begin

                        counter_or_running <= 0 ;
                        counter_or <= counter_or_running;

                end else if (signal_or_edge) begin

                        counter_or_running <= counter_or_running + 1;

                end
        end
end

always@(posedge clk_i) begin        //assign counting algo to out
```

```
if (reset == 1) begin

    counter <=target_value;

end else begin if (ctrl_sw == 0) begin
        counter <= counter_or;
end
if (ctrl_sw == 1 ) begin

    if (fail_sw1 || fail_sw2) begin

        counter <= 2* (counter4_unclip + counter3_unclip);

    end
    else if (fail_sw3 || fail_sw4) begin

        counter <= 2* (counter1_unclip + counter2_unclip);

    end
    else begin

           counter <= counter1_unclip + counter2_unclip + counter3_unclip +

        end

end
end
end

//assign to out for debugging purposes....


assign clk_cnt = clk_count;
assign debug[0] = signal1;
assign debug[1] = signal2;
assign debug[2] = signal3;
assign debug[3] = signal4;
assign debug[4] = signal1_edge;          //debug info
assign debug[5] = signal2_edge;
assign debug[6] = signal3_edge;
assign debug[7] = signal4_edge;
assign debug[8] = signal_or1;
assign debug[9] = signal_or2;
```

```
assign debug[10] = signal_or;
assign debug[11] = signal_or_edge;
assign debug[12] = counter_reset;
assign debug[13] = counter1[0];
assign debug[14] = counter1_unclip[0];
assign debug[16] = fail_sw1;
assign debug[17] = fail_sw2;
assign debug[18] = fail_sw3;
assign debug[19] = fail_sw4;
assign debug[20] = reset;
assign debug[31] = 1;




endmodule
```

A.0.2. Verilog Code for Amplitude Adjustment Module

```
module drr_voltage(target_value , // target value for signal counts inp
counter , //signal coun input from the counting module
out,//out: adjusment of the amplitude register
rstn_i,// in: reset provided by RP
clk_i, // in: clk providede by rp
error, //debug info
counter_reset , // in: reset signal of the counter module
debug2 // out: various debug info out put
);


//////////////////////////

input counter_reset;

input clk_i;

input rstn_i;

input [31:0] counter;

input [31:0] target_value;
```

```
output [13:0] out;

output error;

output[ 31: 0 ] debug2;




//////////////////

reg overflow_error;

reg[31:0] dif;  //dif of signal rate and target rate

reg[13:0] out_temp; //temp. storage of amp adjustment

reg [13:0] out1;//register that gets changed accordingly

reg[31:0] counter_temp; //legacy I believe it runs without now

reg[15:0] rel_dif; // msb dif from dif and target rate

reg[15:0] msb_t; //msb of target rate

reg[15:0] msb_r; // msb of diff

reg[23:0] counter_reset_shift; //shift register for the counter reset

reg[13:0] step; //step size of one adjustment step
reg[13:0] upper; // upper boundary for out
reg[13:0] lower; // lower boundary
reg out_err; //legacy debug can be used if wanted as output

reg[13:0] storage; // storage of how many steps in adjustments are needed
```

```
//////////////////////////

always @(posedge clk_i) begin



if ( counter_reset) begin  // make differenc once the int time is over
    if (target_value > counter) begin
        dif <= target_value - counter ;
    end
    if (target_value < counter ) begin
        dif <= counter - target_value;
    end
    if (target_value == counter ) begin
        dif <= 0;
    end
end


counter_reset_shift[0] <= counter_reset;
counter_reset_shift[1] <= counter_reset_shift[0];
counter_reset_shift[2] <= counter_reset_shift[1];
counter_reset_shift[3] <= counter_reset_shift[2];//
24 Bit shift register of the counter reset signal
counter_reset_shift[4] <= counter_reset_shift[3];
counter_reset_shift[5] <= counter_reset_shift[4];
counter_reset_shift[6] <= counter_reset_shift[5];
counter_reset_shift[7] <= counter_reset_shift[6];
```

```
counter_reset_shift[8] <= counter_reset_shift[7];
counter_reset_shift[9] <= counter_reset_shift[8];
counter_reset_shift[10] <= counter_reset_shift[9];
counter_reset_shift[11] <= counter_reset_shift[10];
counter_reset_shift[12] <= counter_reset_shift[11];
counter_reset_shift[13] <= counter_reset_shift[12];
counter_reset_shift[14] <= counter_reset_shift[13];
counter_reset_shift[15] <= counter_reset_shift[14];
counter_reset_shift[16] <= counter_reset_shift[15];
counter_reset_shift[17] <= counter_reset_shift[16];
counter_reset_shift[18] <= counter_reset_shift[17];
counter_reset_shift[19] <= counter_reset_shift[18];
counter_reset_shift[20] <= counter_reset_shift[19];
counter_reset_shift[21] <= counter_reset_shift[20];
counter_reset_shift[22] <= counter_reset_shift[21];
counter_reset_shift[23] <= counter_reset_shift[22];



end



////////////////////////////////


always @(posedge clk_i) begin //determine msb of target r and diff r

if (counter_reset ) begin

    if (target_value[23]) begin

        msb_t <= 16'd24;
    end

    else if (target_value[22]) begin

            msb_t <= 16'd23;

    end
```

```
else if (target_value[21]) begin
          msb_t <= 16'd22;
end


else if (target_value[20]) begin

        msb_t <= 16'd21;
end

else if (target_value[19]) begin

        msb_t <= 16'd20;
end

else if (target_value[18]) begin

        msb_t <= 16'd19;
end
else if (target_value[17]) begin

        msb_t <= 16'd18;
end
else if (target_value[16]) begin

        msb_t <= 16'd17;
end
else if (target_value[15]) begin

        msb_t <= 16'd16;
end
else if (target_value[14]) begin

        msb_t <= 16'd15;
end

else if (target_value[13]) begin

        msb_t <= 16'd14;
end

else if (target_value[12]) begin

        msb_t <= 16'd13;
```

```
end

else if (target_value[11]) begin

        msb_t <= 16'd12;
end

else if (target_value[10]) begin

        msb_t <= 16'd11;
end

else if (target_value[9]) begin

        msb_t <= 16'd10;
end

else if (target_value[8]) begin

        msb_t <= 16'd9;
end



// else msb_t = 0

if (dif[23]) begin
    msb_r <= 16'd24;
    end
else if (dif[22]) begin

    msb_r <= 16'd23;

    end

else if (dif[21]) begin

    msb_r <= 16'd22;

    end

else if (dif[20]) begin

    msb_r <= 16'd21;
```

```
        end
    else if (dif[19]) begin
        msb_r <= 16'd20;
        end
    else if (dif[18]) begin
        msb_r <= 16'd19;
        end
    else if (dif[17]) begin
        msb_r <= 16'd18;
        end
    else if (dif[16]) begin
        msb_r <= 16'd17;
        end
    else if (dif[15]) begin
        msb_r <= 16'd16;
        end
    else if (dif[14]) begin
        msb_r <= 16'd15;
        end
    else if (dif[13]) begin
        msb_r <= 16'd14;
        end
```

```
else if (dif[12]) begin
    msb_r <= 16'd13;
    end
else if (dif[11]) begin
    msb_r <= 16'd12;
    end
else if (dif[10]) begin
    msb_r <= 16'd11;
    end
else if (dif[9]) begin
    msb_r <= 16'd10;
    end
else if (dif[8]) begin
    msb_r <= 16'd9;
    end
else if (dif[7]) begin
    msb_r <= 16'd8;
    end
else if (dif[6]) begin
    msb_r <= 16'd7;
    end
else if (dif[5]) begin
```

```
            msb_r <= 16'd6;
            end
        else if (dif[4]) begin
            msb_r <= 16'd5;
            end
        else if (dif[3]) begin
            msb_r <= 16'd4;
            end
        else if (dif[2]) begin
             msb_r <= 16'd3;
             end
        else if (dif[1]) begin
             msb_r <= 16'd2;
             end
        else if (dif[0]) begin
             msb_r <= 16'd1;
             end
        else begin
            msb_r <= 0;
            end

end
end
```

```
//////////////


assign rel_dif = msb_t - msb_r ; //relative differnece



//initalize some values
initial begin
step = 14'd614;
storage = 14'd5;
lower = 14'd1000;
upper = 14'd15900;
end

always @(posedge clk_i) begin

    if (rstn_i == 1) begin

        out1 <= 14'h2000; //set to 2000 hex when reset

    end

    if(dif > target_value) begin //debug info
        out_err <= 1;
    end

    else begin
        out_err <= 0;
    end

    if (counter_reset_shift[19]) begin
//determine steps that needs to taken for adjustment

        storage <= 14'd5 - rel_dif;

    end

    if (counter_reset_shift[20]) begin
```

```
        if (storage > 14'd5 ) begin
            storage <= 14'd0;
        end
    end

    if (counter_reset_shift[21]) begin //calc temp out
        if (counter < target_value) begin
            out_temp <= out1 + ( step * storage) ;
        end
        if(counter > target_value) begin
            out_temp <= out1 - ( step * storage) ;
        end
    end
if (counter_reset_shift[22]) begin // determine if temp out is in bound

    if(counter > target_value) begin
        if (out_temp > upper ) begin
            out1 <= lower;
        end

        if (out_temp < lower ) begin
            out1 <= lower;
        end
        else begin
```

```
            out1 <= out_temp;
        end

    end

    else if(counter < target_value) begin

        if (out_temp > upper ) begin

            out1 <= upper;

        end

        if (out_temp < lower ) begin

                out1 <= upper;

        end

        else begin

            out1 <= out_temp;

        end


     end

end




end



assign out = out1 ; // assign to output
```

```
assign error = out_err;//debug info

wire[31:0] debug_value;//debug info
assign debug_value = target_value;
assign debug2 = dif;
```

```
endmodule
```

A.0.3. Verilog Code for Verilog Testbench Source

```
module tb_drr_main();
reg signal1, signal2 , signal3, signal4 , reset; // signal is the signa
reg [31:0] int_time ; //integration time given as a 4 bit number that s
wire [31:0] counter; //signal counter
reg ctrl_sw;
```

```
reg[31:0] threshhold;
wire[31:0] clk_count;

reg clk_i ; //wire for the internal clock , that is used to get a int time

wire [13:0] out; //output

reg [31:0] value ;//compare value. the value of counter is compared to this

reg reset_m;
wire counter_reset;

drr_counter DUT(
.signal1(signal1),
.signal2(signal2),
.signal3(signal3),
.signal4(signal4),
.rstn_i(reset),
.int_time(int_time),
.counter(counter),
.clk_i(clk_i) ,
.out_debug(out),
.threshold(threshhold),
.ctrl_sw(ctrl_sw),
.clk_cnt(clk_count),
.counter_reset(counter_reset)
);

drr_voltage DUT2(
.counter_reset(counter_reset),
.clk_i(clk_i),
.rstn_i(reset),
.target_value(value),
.counter(counter)
);
```

```
////

initial begin //initialize
reset_m = 0;
ctrl_sw = 1 ;
value = 32'd19000;
clk_i = 0;
signal1 = 0    ;
signal2 = 0    ;
signal3 = 0    ;
signal4 = 0    ;
int_time = 32'd31250000;
threshhold = 32'd500;
#1 reset = 0 ;

#20 reset = 1;
#8 reset = 0;


end

always begin

#4 clk_i = ~clk_i; //clock , the clock input later comes from the redpi


end

always begin
#54733100 signal1 = ~signal1;  //creates a signal that turns on every 4
#80 signal1 = ~signal1;
end
always begin
#53700 signal2 = ~signal2;
#80 signal2 = ~signal2;
end
always begin
#48321 signal3 = ~signal3;
#80 signal3 = ~signal3;
end
always begin
#43022 signal4 = ~signal4;
#80 signal4 = ~signal4;
end
```

```
endmodule
```