Bachelor Thesis
Physics

RWTH Aachen University

Physics Institute III B

Real Time Integration of a MHz Signal using a Fast Programmable Gate Array

Taline Kehlenbach

Aachen, May 25, 2020

Eigenhändigkeitserklärung

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

(Ort, Datum)	(Taline Kehlenbach)

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) des Forschungszentrums Jülich GmbH

Abstract

Real-time beam diagnostic systems play a significant role in particle accelerator physics. Digital processing of signals is often part of the diagnostic systems. At Cooler Synchrotron (COSY) at Forschungszentrum Jülich the alternating part of the beam current is measured with a Fast Current Transformer (FCT).
The signal processing from the FCT is done with a Field Programmable Gate Array (FPGA) board. Here, an algorithm for real time integration of the alternating part of the beam current for detected particle bunch at COSY is presented. This algorithm has been constructed to specifically meet the requirements of the signal and will be implemented in the already existing structure of the signal processing using FPGA at COSY. The task is to construct an algorithm in Hardware Description Language (HDL) that can be used to integrate over the described signal and handle the challenges posed by the high orbital frequency up to 2MHz of the beam bunches, which affect the baseline detection of the beam current. Furthermore the possibility of multiple distinct beam bunches inside the synchrotron at the same time needed to be considered.
The developed algorithm serves as a foundation for further improvements of the integration process and is therefore fairly minimal and separated in several modules to simplify changes in the future. Finally, possible areas of improvement for the designed algorithm are pointed out and discussed.

Contents

List of Acronyms iii

List of Symbols iv

List of Figures vi

List of Tables vii

1. Introduction 1

2. Theory and Setup 3

2.1. The Cooler Synchrotron . . . 3
2.2. Principle of Fast Current Transformers . . . 3
2.2.1. Beam current measurement at COSY . . . 7
2.3. Basics of FPGA technology . . . 8
2.3.1. The Red Pitaya Board . . . 8
2.3.2. Programming with Verilog . . . 9

3. Implemented Design 13

3.1. Overall module . . . 13
3.2. Exponential moving average . . . 15
3.2.1. Analysis of the algorithm for the Exponential Moving Average (EMA) . . . 15
3.2.2. Verilog module for the EMA . . . 17
3.3. Peak detector . . . 19
3.4. Simple moving average . . . 19
3.4.1. Analysis of the algorithm for the Simple Moving Average (SMA) 19
3.4.2. Verilog module for the SMA . . . 22
3.5. Baseline detector . . . 23
3.6. Integrator . . . 23
3.7. Simulation and testing . . . 24
3.7.1. Signal simulation . . . 24
3.7.2. Simulation results . . . 25
3.7.3. Implementation and testing on hardware . . . 29

4. Results and Outlook 33

Bibliography **35**

A. Appendix **A 1**
A.1. Verilog modules . . . . . . A 1
A.1.1. Structural module new_integration.v . . . . . . A 1
A.1.2. EMA filter module ema.v . . . . . . A 3
A.1.3. Peak detection module peak_detector.v . . . . . . A 4
A.1.4. SMA filter module sma.v . . . . . . A 5
A.1.5. Baseline detection module baseline_detector.v . . . . . . A 7
A.1.6. Integration module integration.v . . . . . . A 8
A.2. Bash script . . . . . . A 10

List of Acronyms

API Application Programming Interface

COSY Cooler Synchrotron

EMA Exponential Moving Average

FCT Fast Current Transformer

FPGA Field Programmable Gate Array

HDL Hardware Description Language

IKP Nuclear Physics Institute (Institut für Kernphysik)

JULIC Jülich Light Ion Cyclotron

LUT Lookup Table

RF Radio Frequency

SMA Simple Moving Average

List of Symbols

Symbol	Physical quantity	Unit	See chapter
B	Magnetic field	T	2
C_2	Stray capacitances in secondary circuit	F	2
H	Transfer function		3
I_2	Current in secondary circuit of transformer	A	2
I_2	Secondary current in FCT	A	2
I_b	Beam current	A	2
L_1	Magnetic inductance of coil in primary circuit	H	2
L_2	Magnetic inductance of coil in secondary circuit	H	2
N	Maximum time index		3
N_1	Number of turns of primary transformer circuit		2
N_2	Number of turns of secondary transformer circuit		2
R_2	Ohmic resistance in secondary circuit of transformer	Ω	2
T_s	Sampling interval	s	3
U_2	Voltage induced in secondary transformer circuit	V	2
V_r	Voltrange of ADC connector	V	3
V_{in}	Voltage input of ADC connector	V	3
Z	Impedance of FCT	Ω	2
Δt	Time step for integration	s	3
Ω	Normalized angular frequency		3
Φ_1	Magnetic flux caused by beam current	Wb	2
Φ_2	Magnetic flux caused by current in secondary transformer circuit	Wb	2
Φ_t	Total magnetic flux inside FCT torus	Wb	2
α	Weighting factor for EMA		3
δ	Dirac impulse		3
λ_N	Number of particles per unit length	A	2
μ	Magnetic permeability	H/m	2

Symbol	**Physical quantity**	**Unit**	See chapter
ω	Angular frequency	1/s	2
ω_{high}	Higher cutoff frequency	1/s	2
ω_{low}	Lower cutoff frequency	1/s	2
τ	Time constant of FCT	s	2
τ_{droop}	Droop time of FCT	s	2
τ_{rise}	Rise time of FCT	s	2
a	Inner radius of FCT torus	m	2
b	Lowest signal input (baseline)		3
b	Outer radius of FCT torus	m	2
d	Dumping width for EMA		3
f_i	Frequency of the input signal		3
h	Width of FCT torus	m	2
h	Impulse response		3
i_14	14-bit integer		3
k	Number of averaged values for SMA		3
n	Time index		3
$n_{bunches}$	Number of bunches detected by the integration module		3
$n_{skipped}$	Number of skipped bunches in between integrated bunches		3
q	Particle charge	C	2
r	Radius	m	2
s	Laplace variable	1/s	2
t	Time	s	2
t_{int}	Runtime of integration module	s	3
v	Particle velocity	m/s	2
x	Time-discrete signal		3
y	Filtered signal		3

List of Figures

2.1. Layout of the accelerator complex Cooler Synchrotron (COSY) 4
2.2. a) Transfer impedance of the FCT with dependence of the angular frequency ω on a logarithmic scale and marked bandwidth b) Response of the FCT to a step-function 6
2.3. a) Gaussian bunch shape (blue graph), b) Sine-like bunch shape (blue graph), c) Overlapping bunch shape (blue graph) 8
2.4. Redpitaya STEMLab hardware [10] 9

3.1. Block diagram of the overall module 14
3.2. Impulse response of EMA for various weighting factors 16
3.3. Frequency response of EMA for $\alpha = 1/64$ 18
3.4. Block diagram of EMA (red: recursive paths) 19
3.5. Impulse response of SMA for $k = 8$ 21
3.6. Frequency response of SMA for $k = 4$ 21
3.7. Various filtered versions of a simulated ADC input using both the EMA and SMA 22
3.8. Signal generated with Pyhton script for three different signal shapes . . 25
3.9. Simulation results for sinusoidal input, SMA with $k = 3$ and no skipped bunches (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter) 26
3.10. Simulation results for gauss shaped input, SMA with $k = 3$ and no skipped bunches (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter) 27
3.11. Simulation results for signal shape with two peaks, SMA with $k = 3$ and no skipped bunches (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter) 27

3.12. Simulation results right after reset signal for sinusoidal input, SMA with $k = 3$ and no skipped bunches, visible deviation of the baseline value in row 5 and lower end results for integration in row 6 for the first signal period (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter) 28
3.13. Simulation results for signal shape with two peaks, SMA with $k = 3$ and every second bunch skipped (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter) 29
3.14. Bunch counter output from tests on the Redpitaya board showing linear behavior as expected 31

List of Tables

2.1. FCT specifications [8] 7
2.2. STEMLab 125-14 hardware [10] 9
2.3. STEMLab 125-14 specifications for Radio Frequency (RF) connectors [10] 9

3.1. Different signals and settings used for testing 31
3.2. Averaged testing results for different signals with their percentage-wise deviation form the expected value 32

1. Introduction

Ever since the early 20th century scattering experiments have been an integral part of particle physics to study the building blocks of the universe. Nowadays these experiments have developed into particle accelerators which are able to accelerate and collide different particles at very high energies.
Since these tasks demand all accelerator components to function with high precision, diagnostic systems for the parameters of the particle beam are essential to ensure the functionality of any particle accelerator. At the Nuclear Physics Institute (Institut für Kernphysik) (IKP) and more specifically COSY at Forschungszentrum Jülich there are many data acquisition systems and real-time processing systems that ensure the performance of the facility. One of these important beam measurements is the beam current, since it carrys information regarding the longitudinal charge distribution of the beam [1, p. 3]. At COSY, the beam current is measured with a FCT. The real time signal processing is accomplished with Field Programmable Gate Array (FPGA) technology.
In this thesis, a way to implement a bunch-wise integration over the detected beam current for FPGAs is presented. The resulting data can give further insight into the amount of particles per beam bunch. In order for the algorithm to be useful, it has to be:

- applicable to the Red Pitaya STEMLab 125-14 measurement board,
- able to periodically skip particle bunches in the beam in the case of multiple distinct particle bunches inside the ring,
- integrate over one particle bunch in real-time

The thesis covers the basic setup of FPGAs and the measurement process for beam currents. Additionally, the fundamentals of FPGAs and Hardware Description Languages (HDL) and the hardware used at COSY are introduced. Consequentially, the layout of the algorithm will be explained and discussed regarding its strengths and weaknesses and possible room for improvement.

2. Theory and Setup

In this chapter the fundamental knowledge about the different parts of signal generation and acquisition leading up to the implemented design is elucidated. This includes the basic setup of the Cooler Synchrotron and the physical principles behind the Fast Current Transformer to explain the typical output signal of the device. Afterwards the focus will lie on the signal processing electronics that are based on FPGA technology and the programming software to configure said electronics.

2.1. The Cooler Synchrotron

Since the results of this thesis are to be applied at the COSY operated by the IKP, a few details about the facility are introduced as context.
The particle accelerator and storage ring COSY is used in fundamental research in the fields of hadron, particle and nuclear physics [2, p.2]. The ring has a circumference of 183.47 m with two 40 m long linear sections. It is connected to the Jülich Light Ion Cyclotron (JULIC) for preacceleration [3, p.7] and an extraction beam line leading to external experiments. A layout of the accelerator complex can be seen in Figure 2.1.
At COSY, one is able to accelerate polarized and unpolarized proton and deuteron beams in the momentum range between 300 MeV/c and 3.65 GeV/c [3, p.6]. A fairly unique feature of COSY are the cooling units: COSY offers electron as well as stochastic cooling [2, p.2].
The FCT can be found in the straight section after the injection beamline [5]. Since the developed algorithm processes the FCT signal, the setup and functionality of a FCT are elucidated in the following section.

2.2. Principle of Fast Current Transformers

In order to measure the beam current at COSY a FCT is used. Therefore, a basic explanation of the functionality of FCTs is needed to put the programming, that is to be discussed, into physical context.
Since the FCT only relies on the electromagnetic field emitted by the particle beam, it is a non-destructive method to measure the beam current [1, p.2]. Basically the FCT works with similar principles as a loaded current transformer.

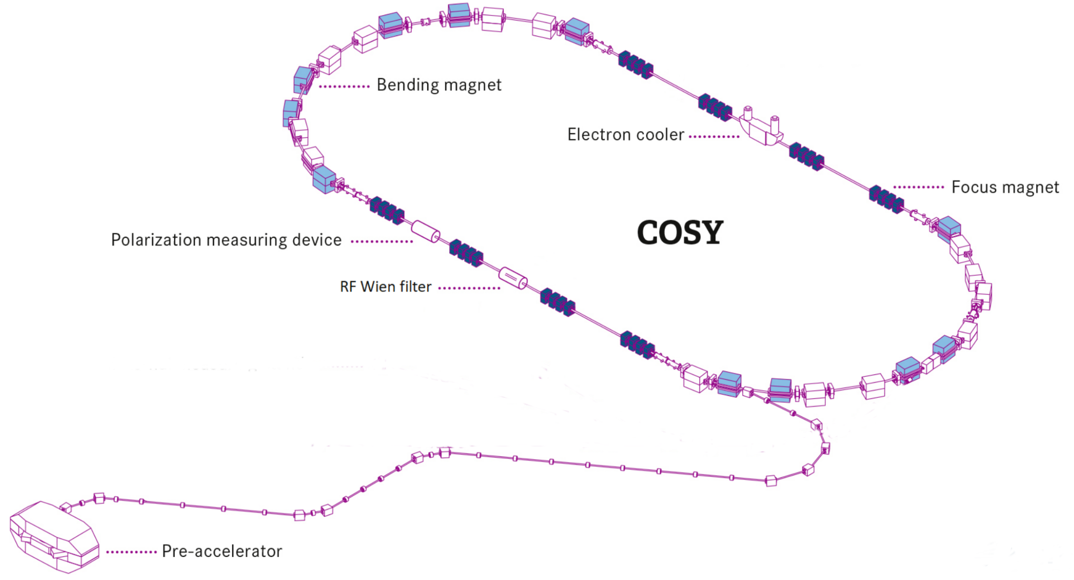


Figure 2.1.: Layout of the accelerator complex COSY

The circular beam current I_b is the primary winding of the transformer and can be described as

$$I_b = q \cdot \lambda_N \cdot v \tag{2.1}$$

with q the particle charge, λ_N the number of particles per unit length and v the velocity of particles [1, p.3]. The secondary winding is connected to the beam current by a highly permeable torus to guide the field lines and to increase the inductance L [7, p.11]. The beam current creates a magnetic field $B(r)$ according to the Biot-Savart law

$$B(r) = \frac{\mu \cdot I_b}{2\pi r} \tag{2.2}$$

with r the distance from the beam and I_b beam current. Accordingly, the magnetic flux Φ_1 caused by the beam current inside the torus with width h, the inner radius from the particle beam a and the outer radius b is given by the equation

$$\Phi_1 = \int \vec{B} \cdot d\vec{A} = \mu \frac{h\, N_1\, I_b}{2\pi} \ln \frac{a}{b} \tag{2.3}$$

with $N_1 = 1$ number of turns of the primary winding (i.e. the particle beam). Any change of the beam current results in a change of the total magnetic flux Φ_t which induces a voltage U_2 inside the secondary winding according to Faraday's law of induction:

$$U_2 = -N_2 \frac{d\Phi_t}{dt} \tag{2.4}$$

with N_2 the number of turns of the secondary circuit. Because of the electric load of the secondary circuit, a current I_2 is able to flow:

$$I_2 = U_2/R_2 = -\frac{N_2}{R_2}\frac{d\Phi_t}{dt} \tag{2.5}$$

with R_2 the ohmic resistance of the secondary circuit. This current results in a magnetic flux

$$\Phi_2 = \int \vec{B} \cdot d\vec{A} = \mu \frac{h\, N_2\, I_2}{2\pi} \ln \frac{a}{b} \,. \tag{2.6}$$

The total flux is then given by

$$\Phi_t = \Phi_1 + \Phi_2 = \frac{L_1\, I_1}{N_1} + \frac{L_2\, I_2}{N_2} \tag{2.7}$$

with $L_1 = \mu \frac{h\, N_1^2}{2\pi} \ln \frac{a}{b}$ the inductance of the primary winding and $L_2 = \mu \frac{h\, N_2^2}{2\pi} \ln \frac{a}{b}$ the one of the secondary winding. The relation between the beam current and the current in the secondary circuit is typically derived using the Laplace-transformation on equations 2.5 and 2.7:

$$\Phi_t = L_1\, I_1 + \frac{L_2\, I_2}{N_2} \tag{2.8}$$

$$I_2 = -s \frac{N_2\, \Phi_t}{R_2} \tag{2.9}$$

The Laplace variable here is $s = i\omega$ and $N_1 = 1$ has already been equated. The solution of these equation leads to

$$I_2 = \frac{s\,\tau}{1 + s\,\tau} \frac{I_b}{N_2} \tag{2.10}$$

with $\tau = L_2/R_2$. Typically the resistance R_2 is very small compared to the inductivity L_2 [7, p.11] so that equation 2.10 can be simplified as

$$I_2 \approx I_b/N_2 \tag{2.11}$$

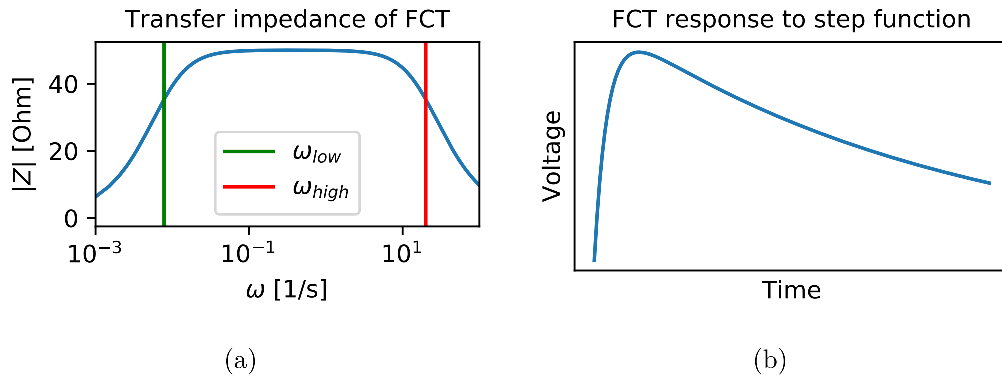
which is the formula for the ideal current transformer. Since instead of the current of the secondary circuit typically the voltage U_2 with $U_2 = \frac{R_2}{N_2} \cdot I_b$ is measured, the value $\frac{R_2}{N_2}$ is referred to as the sensitivity of the transformer. [1, pp.10-11]
To examine the frequency response of the FCT, some stray capacitances C_2 between the turns of the coil and the cables of the secondary circuit in general have to be taken into account. Together with the ohmic resistance and the inductivity of the coil this leads to the following impedance of the circuit:

$$\frac{1}{Z} = \frac{1}{i\omega L_2} + \frac{1}{R_2} + i\omega C_2 \,. \tag{2.12}$$

Typically the Voltage over this impedance is measured as the output value of the FCT. Equation 2.12 shows, that the output of the FCT is influenced by the angular frequency of excitation. The possible cases are:

- $\omega \ll R_2/L_2$
  In this case the impedance becomes $Z \rightarrow 1/(i\omega L_2)$ which clarifies again that no direct currents can be measured.
- $\omega \gg 1/(R_2 C_2)$
  Here, the impedance can be approximated by $Z \rightarrow 1/(i\omega C_2)$ meaning the current is mainly influenced by the capacitances and in return leads to a low voltage output.
- $R_2/L_2 \ll \omega \ll 1/(R_2 C_2)$
  This case is also known as the working region of the FCT for which $Z \approx R$ is true which leads to optimal measurement conditions for the beam current.

The bandwidth of the working region from the lower cutoff frequency $\omega_{low} = R_2/L_2$ to the upper cutoff frequency $\omega_{high} = 1/(R_2 C_2)$ is shown in Figure 2.2. These cutoff frequencies are connected to the rise and droop times of the FCT which are characteristic for the time response of the device. Given the example of a simple step function as a signal, the output of the FCT would rise proportional to $1 - e^{-t/\tau_{rise}}$ with the rise time $\tau_{rise} = 1/\omega_{high} = R_2 \cdot C_2$. Afterwards the signal would drop proportional to $e^{-t/\tau_{droop}}$ with $\tau_{droop} = 1/\omega_{low} = L_2/R_2$ since the direct current component of the step function is cannot be consistently measured by the FCT. [7, pp.12-13]
The modeled response to a step function is also shown in Figure 2.2.



(a) (b)

Figure 2.2.: a) Transfer impedance of the FCT with dependence of the angular frequency ω on a logarithmic scale and marked bandwidth b) Response of the FCT to a step-function

In conclusion, the output of the FCT is dependent on a number of properties of the circuit. Firstly, the amount of turns N_2 has a direct effect on the sensitivity R_2/N_2

from equation 2.11. A high number of turns will lead to a low output amplitude which is to be avoided. To achieve a long droop time, the inductance L_2 should be large, which is difficult to achieve via a large amount of turns since that would interfere with the last point. Therefore, the permeability can be increased by choosing the right material for the torus of the transformer. Lastly, the rise time can be minimized by keeping the stray capacitances small. [7, pp.14-15]
Another interference with the measurement of the FCT is the wall current that is induced into the conductive beam pipe and would cancel the magnetic field created by the beam current since the wall current is the opposite image of the beam current. The solution is to guide the wall current around the torus of the FCT by installing a highly permeable metallic shield around the whole device and putting a ceramic gap in the beam pipe, where the FCT is installed. This way the wall current has almost no impact on the voltage measured by the transformer. [7, p.15]

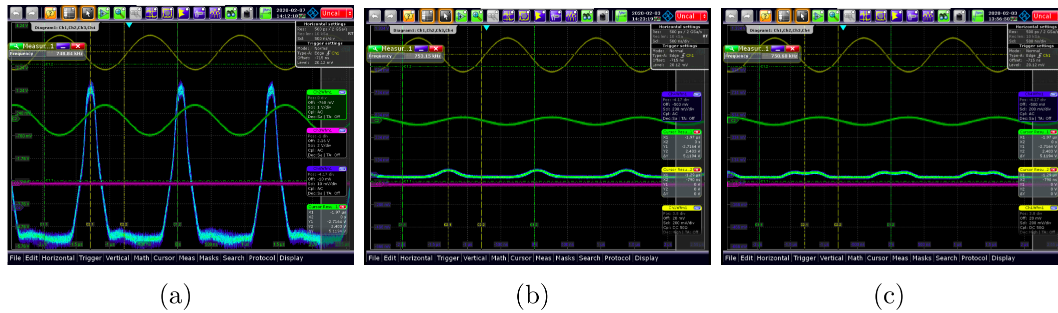
2.2.1. Beam current measurement at COSY

The FCT installed at COSY is manufactured by bergoz instrumentation and has the properties shown in Table 2.1. The torus consists of a CoFe amorphous alloy.

Table 2.1.: FCT specifications [8]

Turns	5	Units
Sensitivity	5	V/A
Rise time (typical)	500	ps
Droop time	5	μs
Upper cutoff frequency (typ.)	700	MHz
Lower cutoff frequency	ł32	kHz
Outer radius	15.24	cm
Inner radius	14.76	cm
Length	4	cm

The output signal, i.e. the respective bunch shapes of the particle beam, depends on the operation mode of the accelerator, e.g. beam cooling etc. The bunches that are detected at COSY are mostly Gaussian or comparable to a sine wave. But also a signal with two "overlapping" peaks is possible. Examples for these bunch shapes are given in Figure 2.3. With contrast to the beam current shown in these figures, the length of the beam bunch compared to the period of circulation at COSY is too large, so that there is no significant downtime in between two bunches, which has to be considered in the algorithm. The frequency of the bunches inside the synchrotron and therefore the frequency of the beam current signal ranges from $\approx$ 500kHz right after an injection up to 2MHz.



(a) (b) (c)

Figure 2.3.: a) Gaussian bunch shape (blue graph), b) Sine-like bunch shape (blue graph), c) Overlapping bunch shape (blue graph)

2.3. Basics of FPGA technology

The output of the FCT is transferred to a STEMLab 125-14 unit and subsequently the FPGA board as the main component of the unit. Therefore, the following paragraph will provide a concise introduction into FPGA technology.
A Field Programmable Gate Array (FPGA) can be used in a vast number of applications since it is highly customizable and offers the benefit of parallelization of processes [9, p.10]. The reason behind this is its structure. FPGAs are basically arrays of programmable logic blocks embedded in a then again programmable interconnect. At the edges of this so called "logic fabric", programmable I/O blocks can be found to interface the fabric signals with external appliances [9, p.5]. These three are the main components of a FPGA.
The programmable interconnect is formed by wires that can be connected to any two logic blocks leading to arbitrary logic networks to be designed by the user. Embedded into this set of wires are the programmable logic blocks. These consist of one or more logic function in the form of a Lookup Table (LUT) and other components like carry chains for high-performance adders and registers that can all be configured by the user. [9, p.6]

2.3.1. The Red Pitaya Board

The FPGA board that will be used for this thesis is a Red Pitaya or more specifically a STEMLab 125-14. It is a FPGA-based open-source project that functions as a versatile measuring and controlling device to replace more expensive laboratory equipment.
The hardware specifications of the board are listed in Table 2.2. A Linux operating system can be installed on a Micro SD card and run on the system processor of the board. Next to the FPGA board of the Red Pitaya, the Radio Frequency (RF) connectors have a high relevance for the discussed application, since the signal from the FCT is connected to the ADC-input port of the Red Pitaya. The most important parameters to be considered during the design implementation are the ADC resolution

of 14 bit and the sample rate of 125 MHz. The other parameters are shown in Table 2.3.

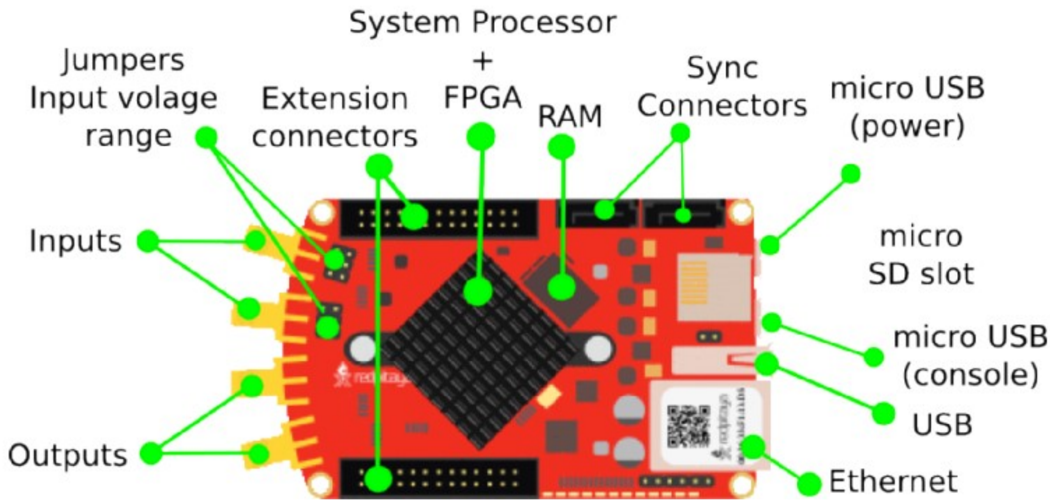


Figure 2.4.: Redpitaya STEMLab hardware [10]

Table 2.2.: STEMLab 125-14 hardware [10]

Processor	Processor DUAL CORE ARM CORTEX A9
FPGA	FPGA Xilinx Zynq 7010 SOC
RAM	512MB (4Gb)
System memory	Micro SD up to 32GB
Power consumption	5V, 2A max

Table 2.3.: STEMLab 125-14 specifications for RF connectors [10]

RF input channels	2
Sample rate	125 MHz
ADC resolution	14 bit
Input impedance	1MΩ
Full scale voltage range	±1V (LV) ±20 (HV)
Absolute max. input voltage range	30V

2.3.2. Programming with Verilog

The configuration of FPGA boards can be accomplished with Hardware Description Languages (HDLs) such as VHDL or Verilog, which apart from some syntax conventions are very similar. The software architecture for this project is built in Verilog and the software used for editing and implementing is Vivado 2016.4.

A data type in Verilog can take on four values:

- The value "0" is a logic zero or a false condition.
- A logic one or a true condition is given by the value "1".
- The value "x" or "X" is taken on by uninitialized variables.
- Lastly, the value "z" or "Z" depicts a high impedance value for tri-state elements.

The two variable types that can take on this set of values are "net data types" which model interconnections between components and "variable data types" that can hold values until their next assignment.
The most common net data type is called "wire", i.e. a simple connection. There are other net data types that are of no concern for the required task. A wire variable needs to be driven in order to lead to a sensible output. It does not store data over time.
Opposed to this are the variable data types. The most common type here is called "reg" (short for register), which models logic storage.
The discussed data types are mostly one bit in length by default. In order to handle signals with more than one bit, Verilog supports vectors, i.e. one-dimensional arrays of elements. The ADC input of the Red Pitaya board for example is 14 bit long.
In order to include these data types in a verilog design they have to be part of a Verilog module. Any Verilog design is stored in a single file ("example.v"). Inside this file, the description of the design is given inside a module. This module typically consists of a port list and port definitions declaring, the port type "input", "output" or "inout" and the data type as discussed above and the port name. Afterwards the module contains the declarations and operations necessary to perform the task required by the user. It is common, to create lower-level modules inside a top-module, to create the desired design and structure the code in a hierarchy.[11, p.14 ff.]
The operators used in Verilog are mostly very similar to the Operators used in the programming language C. However, the keyword "assign" can be used to define a continuous assignment to model combinational logic or time-independent logic. The right-hand side of such an assignment serves as the input for the left-hand side and any change to the input will result in an update of the left-hand side. Several successive continuous assignments in Verilog will result in separate logic circuits and therefore in a concurrent execution of the assign-statements.[11, p.23]
Furthermore, Verilog allows for procedural assignments that are triggered by an event. The possible procedural blocks are "initial" and "always". The inital block is not synthesizable and therefore only used for behavioral simulations. The statements inside an inital block are executed at one time at the beginning of a simulation. Opposed to this, the always block is executed forever (or the duration of the simulation). A procedural block is often followed by a sensitivity list that state the possible trigger for their execution, e.g. a clock signal.

Inside a procedural block two types of assignments can be used: blocking and non-blocking assignments. A blocking assignment is simply denoted by a "=" symbol and leads to a immediate assignment of the right-hand side to the left-hand side, before the next statement is executed. However, with a non-blocking assignment (denoted by "=>") the assignment of update of the left-hand side is delayed until the next blocking assignment occurs while the right-hand side value is cached. This assignment can be used to model sequential logic, where the output depends on the input history. [11, p.65 ff]
To implement a design, the written code is put through a design flow. The Xilinx design flow consists of the following steps:
Firstly, the design entry is naturally defined by the creation of a project with various modules and constraints. This step is followed by the design synthesis that outputs a simple netlist listing all connections between the different logic gates etc. This netlist is then mapped onto the specific device during the design implementation. Finally, a bitstream can be generated that is then used to program the FPGA board. Additionally, Verilog provides several verification tools in between the described steps to simulate the several stages of the desig. [12]
In this project mostly behavioral simulations are used to test the design. These simulations require a typical Verilog module called a testbench, that calls the module that is to be tested and executes various input conditions for that module [13, p.127].

3. Implemented Design

In this chapter, the various modules designed for the defined task and their interdependencies are described and discussed. Furthermore, the implemented algorithms in each module will be explained. The Verilog code for the modules can be found in section A.1 of the appendix. Finally the process of simulating and testing the designed modules will be explained shortly and the testing results will be discussed.

3.1. Overall module

The module described in this section, serves as the top module in the hierarchy of the integration algorithm. It is supposed to be purely structural, i.e. only serves to instantiate and interconnect sub-modules. However, some behavioral statements such as signal assignments etc. are still found in this module for testing purposes and potential debugging until the full functionality outside of simulations is confirmed.
The module inputs are the clock signal generated by the FPGA to time the procedural blocks, a reset signal, the actual input from the ADC connector and two parameters that can be modified by the user to vary the impact of the filter on the signal and to skip incoming bunches if necessary.
Inside the module, five sub-modules are instantiated. Two of these modules are filters. One is only for lightly filtering out noise of the signal to achieve a more exact value for the minimum of the signal or the baseline in this case. This filter operates according to the Simple Moving Average (SMA) which will be elaborated later on. The impact of the filter on the raw signal can be determined by the parameter mentioned previously. The other module is heavily filtering the input signal to facilitate the detection of noise-independent minima in the signal later on. In this case, the filter used is the Exponential Moving Average (EMA). The filter modules output a filtered version of the input signal each. The output of the SMA-module serves as the input to the module for baseline detection, while the output of the EMA-module is connected to the input port of the peak detection-module. The peak detector outputs a start/stop-signal with each negative peak detected in the output of the EMA-module. This is essential to the integration of the input signal, since it provides a start and stop signal for processes in two other modules. Firstly, the baseline detector has to correct the baseline of the input signal with the beginning of each new bunch, since the ADC input usually has an arbitrary baseline and the minimum of the FCT-signal typically lies near zero. Secondly, the integration has to begin and end with each new bunch. Therefore, the output of the peak detector is connected to two separate modules. However, the output of the baseline detector has to be connected to the integration

as well to shift the before mentioned arbitrary baseline back to zero. Finally, the last module integrates the raw input data and averages over a few integration results. The integration can be skipped for a specific amount of bunches by setting the respective input variable, mentioned in the beginning of this section, to the desired number of integration cycles to be skipped. The average integral is the output of the overall module. To monitor the functionality of the module on the FPGA, it also outputs the baseline value from the baseline detector and a counter, to see how many bunches have been integrated. Additionally, the clock signal is connected to every sub-module as an input to trigger the procedural assignments inside the modules. The reset signal is connected to the sub-modules in the same way. The interconnections of the modules are visualized in Figure 3.1.
Once a module is triggered by the clock signal, it performs the specific tasks in that module and generates an output, as soon as it is done. This means, that the integration module could be working on the current ADC input signal, while the EMA-module might be a few clock signals behind. This leads to a shift of the integration interval, so it does not line up perfectly with the beginning and ending of one beam bunch. Although this does not heavily affect the output of the module in the behavioral simulation and the conducted hardware tests, timing and race conditions for the sub-modules should be considered in further research.

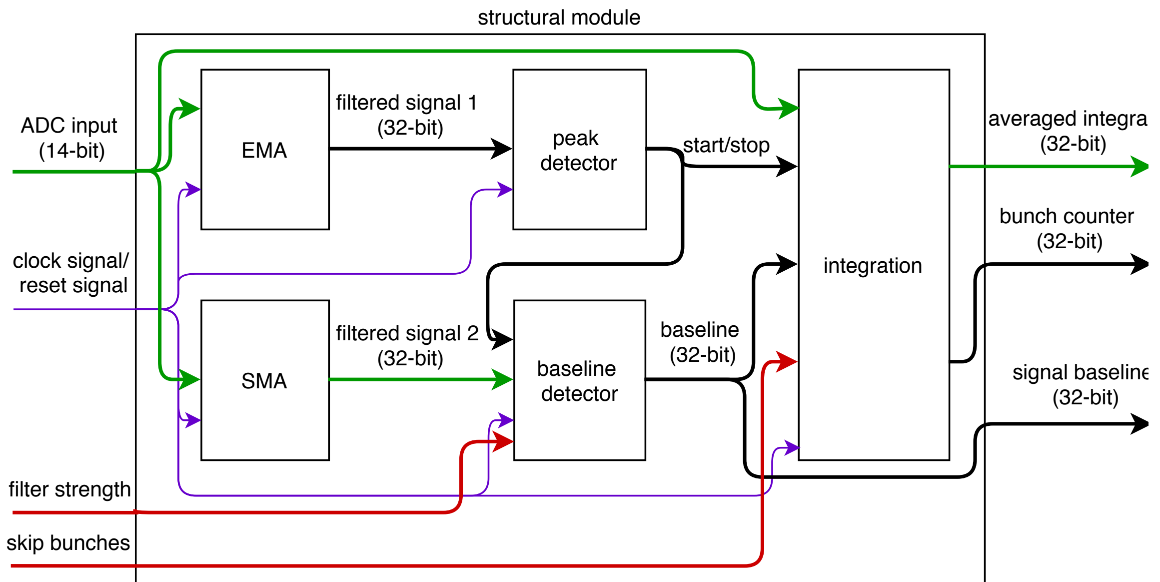


Figure 3.1.: Block diagram of the overall module

3.2. Exponential moving average

The first sub-module to be described has three input signals: the 14-bit long ADC input, the clock signal and the reset signal. The output is a 32-bit filtered version of the ADC input-signal. The behavioral elements contain a filter based on an EMA which is applied to discrete-time systems [14, pp.28-29].

3.2.1. Analysis of the algorithm for the EMA

The algorithm serves as a digital low-pass filter to simplify the subsequent algorithm necessary for the peak detection-module.
The input of this filter is the ADC signal $x[n]$ with $n \in \mathbb{N}_0$ that is sampled in intervals of $T_s = 8$ns. For example a basic sine-function with frequency f can be represented as a digital signal with

$$x[n] = \sin(2\pi \cdot T_s \cdot n), \qquad \Omega = 2\pi \cdot T_s[14, \text{p.30}]. \tag{3.1}$$

The parameter n is the time index and Ω is the dimensionless angular frequency. Compared to a continuous signal, the time t and time index n are connected by $t = n \cdot T_s$. The recursive formula of the filter is

$$y[n] = \alpha x[n] + (1 - \alpha)y[n-1]. \tag{3.2}$$

Here, $x[n]$ is the current state of the system, $y[n]$ is the current filtered state, $y[n-1]$ is the previous filtered state and $\alpha \in (0, 1]$ is the weighting factor (no filtering for $\alpha = 1$) and in part the cutoff frequency. The weighting factor is chosen out of negative powers of two to simplify the identification of the bit-shift operators which will be introduced in the following. When computing the recursive formula 3.2, one can identify the exponential behavior of the filter, since the weighting factor of the previous filter states decreases exponentially:

$$y[n] = \alpha \textstyle\sum_{k=0}^{n} (1 - \alpha)^k x[n-k]. \tag{3.3}$$

This leads to the most recent input value to have the most significance when calculating the filtered state.
To analyze and describe the EMA, the impulse response, frequency response and phase response will be discussed.
The impulse response is the response of the filter to a Dirac impulse

$$\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & n \neq 0 \end{cases}. \tag{3.4}$$

Applied to equation 3.3 this leads to the impulse response

$$h[n] = \alpha(1 - \alpha)^n. \tag{3.5}$$

The impulse response in Figure 3.2 visualizes the exponential behavior of the filter for different weighting factors. It can be seen, that this filter lowers the signal amplitude. Therefore, its output is not suitable as input for the baseline detector-module.

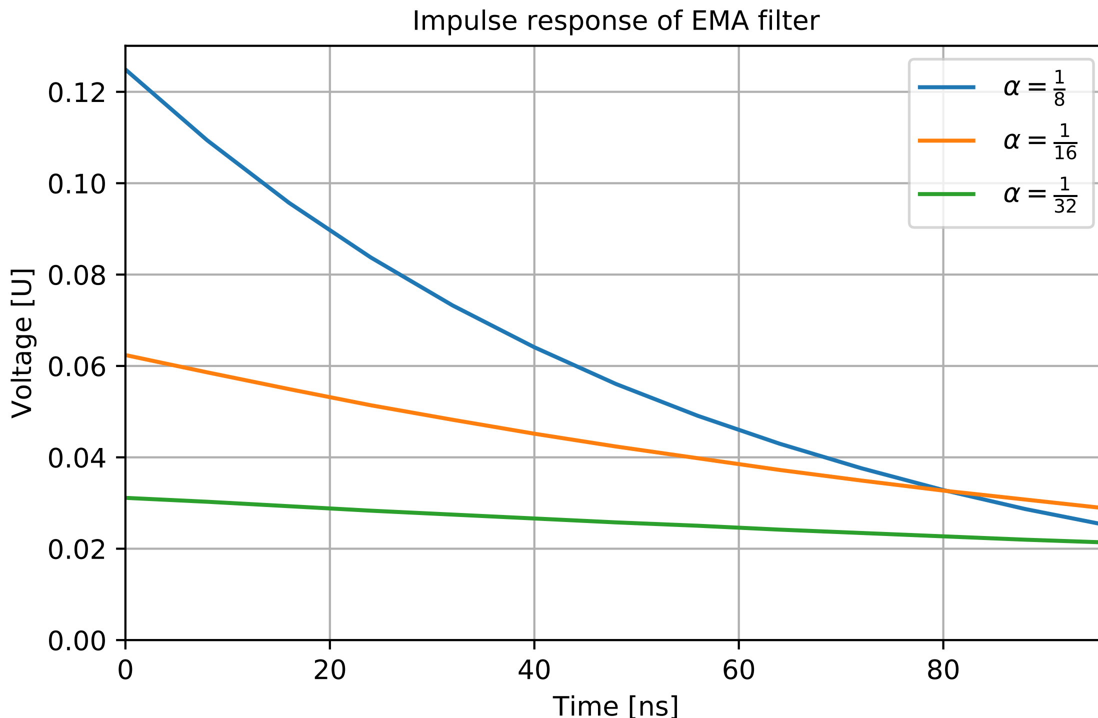


Figure 3.2.: Impulse response of EMA for various weighting factors

To calculate the frequency response of the EMA, the transfer function $H(z)$ of the filter is necessary. Luckily, the Z-transformation of the impulse response equals the transfer function [14, pp.152, 218-219]: As a result, the Z-transformation of the recursive formula 3.2 is

$$H(z) = \sum_{n=0}^{\infty} h[n] z^{-n} \quad z \in \mathbb{Z} \tag{3.6}$$

$$= \sum_{n=0}^{\infty} \alpha (1-\alpha)^n z^{-n} \tag{3.7}$$

$$= \frac{\alpha}{1-(1-\alpha)\, z^{-1}}. \tag{3.8}$$

As one can see, the transfer function has a pole at $z = 1 - \alpha$. This impacts the frequency response of the system.
The frequency response of the filter is then given by

$$H(\mathrm{e}^{i\cdot\Omega}) = \frac{\alpha}{1-(1-\alpha)\, \mathrm{e}^{-i\cdot\Omega}} \tag{3.9}$$

with the dimensionless frequency $\Omega = \omega T_s$, since it models the transfer behavior of the filter for signals with arbitrary frequency Ω, amplitude one and phase shift zero [14, p.152]. The modulus of the frequency response

$$|H(\mathrm{e}^{i\cdot\Omega})| = \frac{\alpha}{\sqrt{1-2(1-\alpha)\cos\omega + (1-\alpha)^2}} \tag{3.10}$$

is the amplitude response for different frequencies. It is typically given in decibels

$$-\log_{10}(|H(\mathrm{e}^{i\cdot\Omega})|)\,\mathrm{dB}. \tag{3.11}$$

The cutoff frequency is connected to this value being the frequency, for which the amplitude of the filtered signal is half the amplitude of the original signal [14, pp.149-150]. This corresponds to approximately −3dB. For the implemented filter, a weighting factor of $\alpha = 1/64$ with cutoff frequency ≈0.31MHz is chosen. The reason for this low cutoff frequency compared to the beam bunch frequency (see subsection 2.2.1) is the high sensitivity of the beam detection-module. Noise should be almost completely filtered out, while the resulting low amplitude of the filtered signal does not affect the beam detection. However, for future tests of the algorithm, that are not part of this thesis, a re-evaluation of the weighting factor should be considered. One should also notice, that the beam detection will likely work best, for higher bunch frequencies.
The phase response is given by the argument of equation 3.9 and shows, how the phase of a signal is shifted regarding the input frequency. The combination of graphs for the modulus and the argument of the complex frequency response is called a Bode plot, which is shown in Figure 3.3. One can see, that the phase response is non-linear, which leads to a distortion of the signal [14, pp.149-150]. This does however not interfere with the desired detection of peaks of the filtered signal, since general upward and downward trends of the signal are still recognizable for the chosen weighting factor.

3.2.2. Verilog module for the EMA

Next, the Verilog module for the EMA is to be discussed.
Like all the sub-modules, this module mainly consists of an always block which begins with the behavioral code in case the reset signal is triggered. This part basically sets all necessary parameters to zero. Afterwards the actual EMA is implemented. The interesting part here is that the recursive formula 3.2 consists of an addition and a subtraction but also a multiplication with a decimal number, which is not time-efficient compared to the other two operations and would require floating-point arithmetic. Now the definition of α as a negative power bears the opportunity for a simpler option. The weighting factor will be defined as $\alpha = 1/2^d$ with the dumping width d. The multiplication of a number by a negative power of two is equal to a bit shift of said number by the exponent. Therefore, if equation 3.2 is multiplied by 2^d (i.e. "shifted" to the left by d bits), then the multiplications remaining are

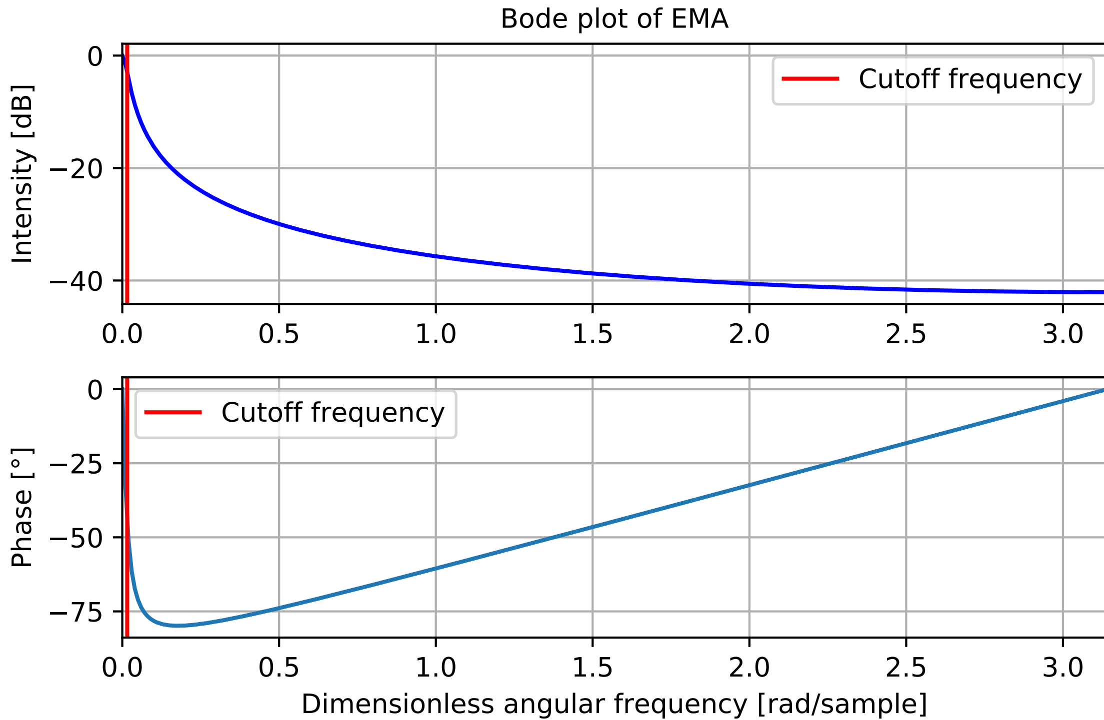


Figure 3.3.: Frequency response of EMA for $\alpha = 1/64$

multiplications by positive integers:

$$y[n] = y[n-1] - \alpha y[n-1] + \alpha x[n] \quad | \cdot 2^d \tag{3.12}$$

$$2^d \cdot y[n] = 2^d \cdot y[n-1] - y[n-1] + x[n]. \tag{3.13}$$

This equation can be easily implemented without floating-point arithmetic. However, it is important, that as a last step, the result is scaled back to its actual value by shifting it d bits to the right before the variable is connected to the output of the module. This process is also shown in Figure 3.4. To avoid an overflow of the variables, when implementing a larger scale of the original recursive formula, one has to account for at least d extra bits in every variable related to the algorithm inside the module. For the 14-bit ADC-input and $\alpha = 1/2^6$, this means that, the variables need to be at least 20 bits long.

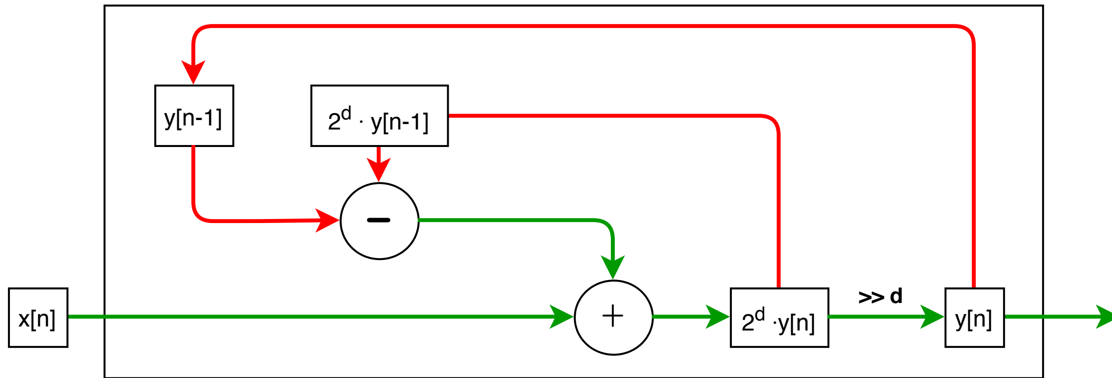


Figure 3.4.: Block diagram of EMA (red: recursive paths)

3.3. Peak detector

Directly connected to the EMA-module, the Peak Detector uses the output of the previous module to signal the beginning and ending of a detected beam bunch. Here, the filtered signal from the EMA-module should be specifically mentioned as an input. Other inputs are, again, the clock signal and the reset signal. The output of this module is a 1-bit signal that serves as a start or stop signal for the baseline detector and the integration module.
Since the algorithm used is fairly simple, no block diagram is provided for this section. Basically, the algorithm detects, whether the input signal is currently rising or falling and sets the start/stop-signal to one, when the edge switches from falling to rising. Otherwise the output is zero. Therefore every local negative peak leads to output one. Since the previously discussed low-pass filter is very effective, a simple algorithm like this can work in this context. However, it is prone to errors such as the detection of noise peaks that are not canceled by the filter. A more stable option could be working with a threshold which proved to be difficult, since the signal amplitude changes depending of the state of the accelerator, i.e. beam cooling or no beam cooling etc.
Therefore,further research into more stable options could improve this module later on.

3.4. Simple moving average

In this section, the second filter will be discussed.

3.4.1. Analysis of the algorithm for the SMA

In contrast to the Exponential Moving Average, the Simple Moving Average (SMA) is a finite impulse response-filter. That means, that there are no recursive elements. The noteworthy input ports are the ADC signal and a variable to declare the strength

of the filter.
The principle behind the SMA is the arithmetic mean of a shift register. The entries of this shift register are signal values from $x[n]$ to $x[n-k-1]$ with k as the length of the shift register. This is summarized in the following equation:

$$y[n] = \frac{1}{k} \sum_{j=0}^{k-1} x[n-j]. \tag{3.14}$$

Similarly to the weighting factor of the EMA, k has to be a power of two, to simplify the calculation of the mean using bit shifting operators.
The impulse response of this filter clearly shows its finiteness:

$$h[n] = \frac{1}{k} \sum_{j=0}^{k-1} \delta[n-j]. \tag{3.15}$$

The graph can be seen in Figure 3.5.
Moving into the discussion of the filter's frequency response, the transfer function is given by

$$\begin{aligned} H(z) &= \sum_{n=0}^{\infty} h[n] z^{-n} \quad z \in \mathbb{Z} && (3.16)\\ &= \sum_{n=0}^{\infty} \frac{1}{k} \sum_{j=0}^{k-1} \delta[n-j] z^{-n} && (3.17)\\ &= \frac{1}{k} \sum_{n=0}^{k-1} z^{-n} && (3.18)\\ &= \frac{1}{k} \cdot \frac{1-z^{-k}}{1-z^{-1}}. && (3.19) \end{aligned}$$

The transfer function has a pole at $z = 0$ and zeros at $z = \mathrm{e}^{i2\pi n/k}$ with $n \in [1, k-1]$. Analogous to the EMA this leads to the Bode plot shown in Figure 3.6. Since the phase response is mostly linear with phase jumps at the zeros of the transfer function, the filtered signal is less distorted than with the use of the EMA.

The reason for using this filter additionally to the EMA is that it enables a much more exact way of filtering noise especially regarding the distortion of the signal amplitude. Since the objective of this filter is to get a more exact baseline value that is not systematically shifted to a lower value by noise, the subsistence of the amplitude is relevant. In Figure 3.7 it is shown, how a simulated signal is filtered with the EMA for the smallest weighting factors compared to filtering with the SMA for the smallest parameters k. One can see, that both filters shift the signal amplitude to a lower value, as expected. However, the SMA distorts the original signal less and enables a finer adjustment of the filter for the possible filter parameters. As a result, the SMA is more suitable for the subsequent baseline detection.

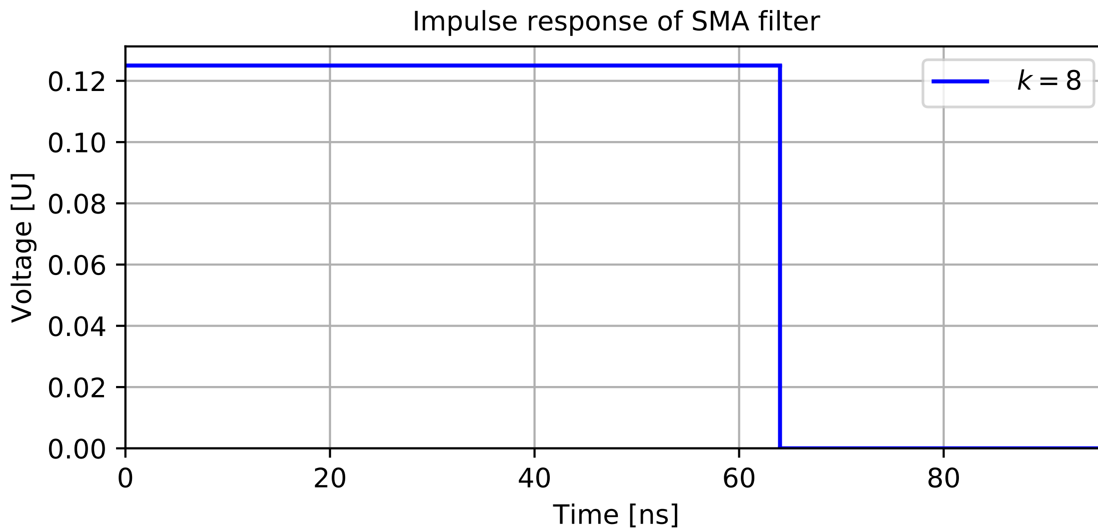


Figure 3.5.: Impulse response of SMA for $k = 8$

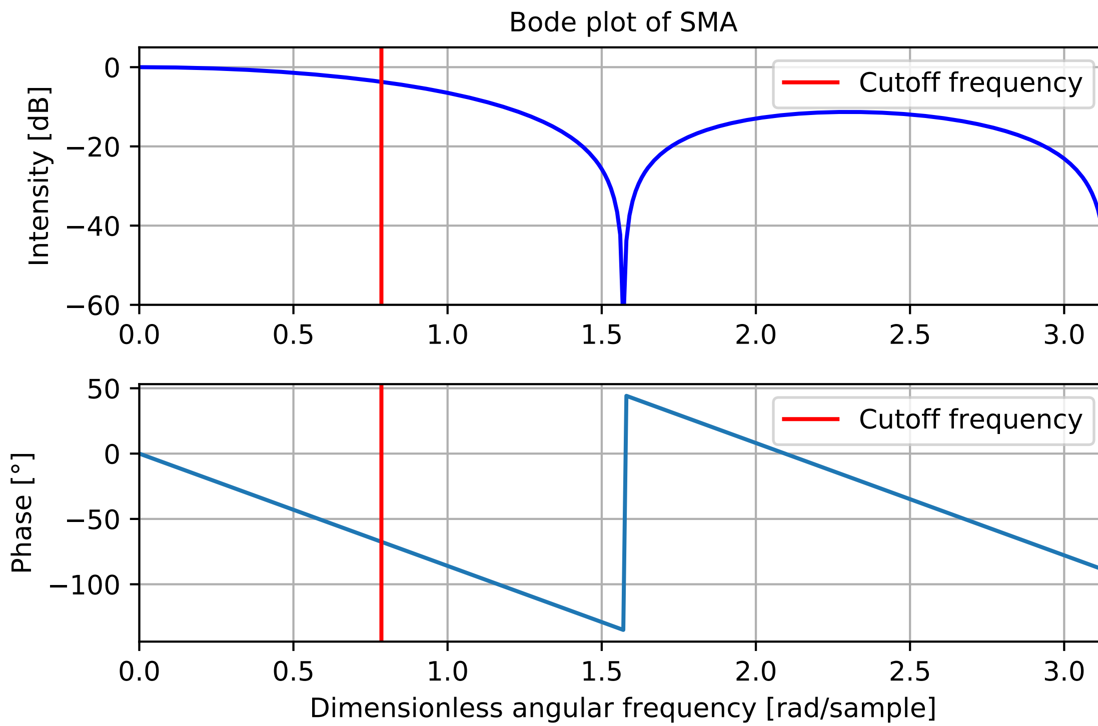


Figure 3.6.: Frequency response of SMA for $k = 4$

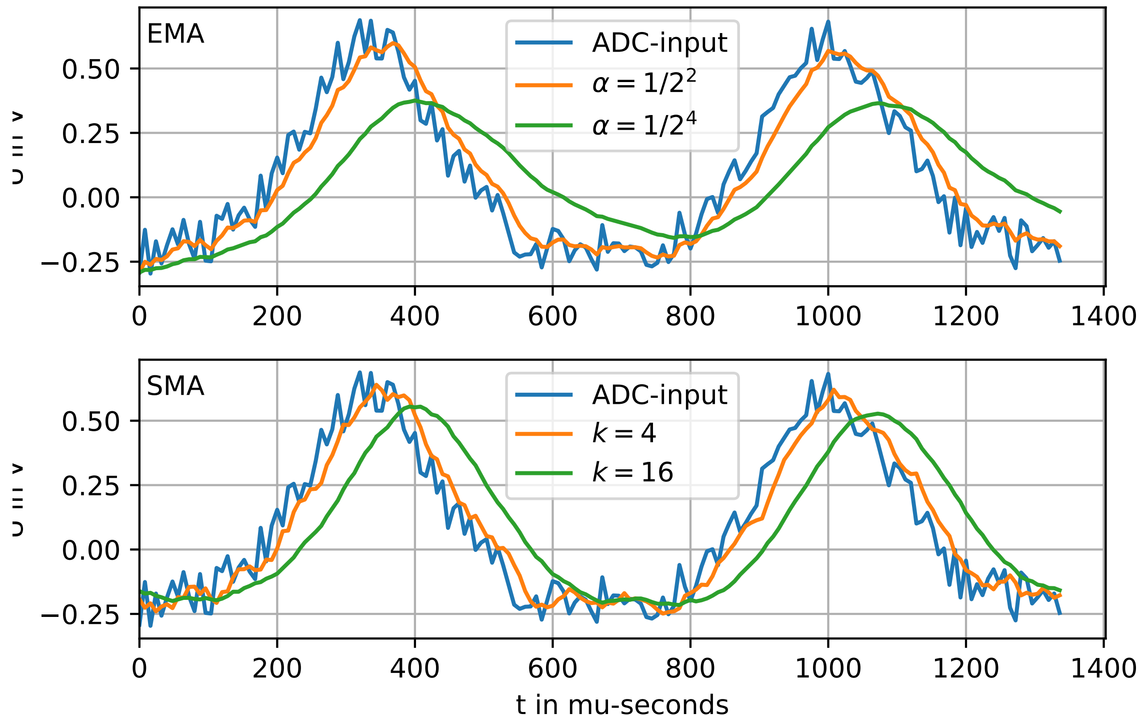


Figure 3.7.: Various filtered versions of a simulated ADC input using both the EMA and SMA

3.4.2. Verilog module for the SMA

The Verilog module consists of two always blocks. The first one contains the declaration and update of a shift register, which is implemented by using non-blocking assignments. This register contains the 32 most recent input signals. Since the shift register can save 32 inputs, the parameter k for the calculation of the mean can be 32,16,8,4 or 2, since k should be a power of two. The calculation of the mean is handled in a separate always block to avoid errors when mixing non-blocking and blocking assignments. Here, the sum of the variables inside the shift register is calculated. However, the shift register has a fixed length of 32 variables. But depending on the user input, one might only want to average the most recent eight or four ADC-input signals. Therefore, the sums over the first 2,4,8,16 and 32 register entries are calculated separately. Then the desired sum is chosen, depending on the user input. This sum is then shifted by the amount of bits given by the user input signal. Again, the advantages of substituting a division with a bit shift are used. This value then serves as the overall output of the module, which is connected to the baseline detector. Since this implementation takes up more resources on the FPGA, the EMA is more efficient, where severe filters are necessary, i.e. the peak detection.

3.5. Baseline detector

This module serves the purpose of defining a new baseline of the ADC signal since the baseline of the raw input signal is arbitrary while it should be near the lowest point of the signal. In a noisy signal the minimum is systematically shifted to a lower value due to noise minima. In order to reduce this shift of the minimum, the input for this module is the signal that is filtered using a SMA. The strength of the filter can be adjusted according to the amount of noise. Over the course of the operating time of the accelerator, the baseline of the ADC signal can shift slightly, which affects the value of the integral later on. As a result, the baseline, i.e. the global minimum of the signal, is evaluated separately for every beam bunch. Therefore, this module is connected to the output of the peak detector which signals the ending of one beam bunch and the beginning of the next one.
The module simply works by comparing current input of the filtered signal to the latest assumed minimum. If the current input is smaller than that assumed minimum, the minimum is replaced by that input. When the peak detector signals the beginning of a new bunch, the current minimum is set as the output of the module and afterwards the cycle begins anew with the current filtered input as the new assumed minimum. This way, the algorithm finds global minima of the filtered ADC signal in the interval for one beam bunch.
To get feedback on the functionality of this module while it is running on the Redpitaya board, the baseline value is also connected to an output port of the overall module. This way, the detected baseline of the signal can be compared to the one of the test signal by monitoring the respective registers of the board.

3.6. Integrator

The Integrator is the Verilog module that joins the results from all previous sub-modules to integrate the ADC signal. This module has the most input ports out of all other sub-modules Firstly, the obligatory clock and reset signal, then the raw ADC input, the start/stop-signal from the peak detector, the baseline signal from the baseline detector and the skip signal to skip over beam bunches as required. The output is an averaged integration result. For tests of the module on the Redpitaya board, a bunch counter, that is not relevant for the functionality of the module, is added as an output.
The algorithm used to integrate the signal follows the rectangle method. With this method, the overall time interval of the integral, i.e. the period length of the beam bunch, is divided into smaller intervals Δt each the length of the sampling time of 8ns. The arbitrary baseline of the signal is corrected by subtracting the baseline value b from the raw input. Since all modules are triggered by the same clock signal, the baseline value that is used to correct the integral is the one from the previous beam bunch. However, the baseline shift is supposed to be a lot slower than the actual beam current so using the baseline value of the last full beam bunch when calculating

the integral of the current beam bunch does not pose a problem. The area in between the current and the last ADC input is approximated by a rectangle the height of the current ADC input. The formula for this method can be written as

$$\int_{t_0}^{t_1} x(t)dx \approx \Delta t \sum_n^N (x[n] - b) \tag{3.20}$$

with the start and end time of the beam bunch t_0 and t_1, the ADC signal input $x[n]$, the baseline value b, the time index n and the maximum time index N for which $t_1 = N \cdot \Delta t$ is true. For the Redpitaya, where the signal is sampled at 125MHz, the time step Δt equals 8ns.

In addition to the integration itself, there are two other conditional blocks in place. One is to skip as many bunches as are given by the user (default is zero), the other is to track how many bunches have been integrated and output an average of the last eight integration results, once eight bunches have been integrated. This last step serves to decimate the data that needs to be saved, since the integration result of a single beam bunch will not be relevant for later data analysis.

The number of integration results to be averaged should be adjusted, depending on the read-out frequency of the output. For example, if the whole module has a runtime of t_{int} without averaging the end result and the output is only read with a frequency of $1/(100 \cdot t_{int})$, the number of results to be averaged should, in this case, be as close to 100 as possible. This would lead to more integration results having an influence on the read-out value. But since the timing conditions of the module are not part of this thesis, the average of eight integration results is chosen, to improve the readability of the simulated output of the integration module in section 3.7.2.

3.7. Simulation and testing

In this section, the simulation and testing results of the overall module are shown.

3.7.1. Signal simulation

To recreate a signal similar to the one of the actual FCT at COSY, a python program that outputs Verilog code to simulate various signals is used.

This program takes the desired amplitude, period length, signal form, noise ratio etc. as inputs and calculates the respective ADC input, in which such a signal could result. To do this, the actual volt range V_r of the ADC connector of -1V to 1V has to be scaled to fit the range of a 14-bit signed integer. The equation to convert the Voltage input V_{in} to the respective 14-bit integer i_{14} is given by

$$i_{14} = \frac{2^{14}}{V_r} \cdot V_{in} \quad [2,\ \text{p.28}]. \tag{3.21}$$

Some examples for possible signals are shown in Figure 3.8. The program delivers a file with one period of the desired signal as Verilog code that can be copied into the

testbench for the integration module to simulate the module. The simulation results are shown in the next section.

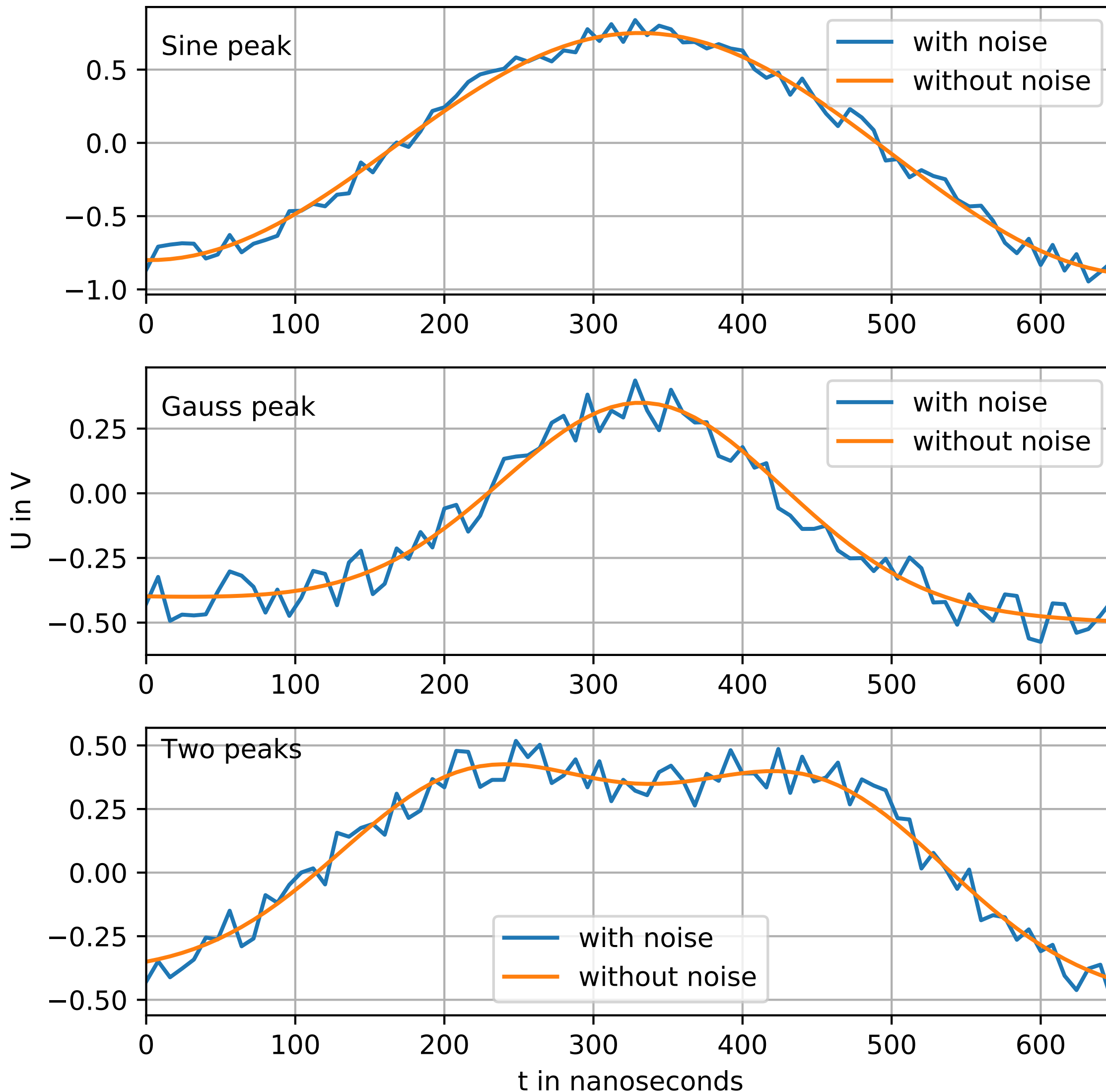


Figure 3.8.: Signal generated with Pyhton script for three different signal shapes

3.7.2. Simulation results

To verify the functionality of the module without having to generate a bitstream for the FPGA, which is time consuming and does not enable an easy way to monitor the variables of the module like simulation does. For the simulation of the integration module, the simulation tool from the Vivado Design Suite has been used.
Figures 3.9, 3.10 and 3.11 show the simulation results for the three signal shapes. In

these figures, the two graphs at the top represents the reset signal and the simulated ADC input. Under that are, in the following order, the Exponential Moving Average, the output of the peak detector, the Simple Moving Average and the output of the baseline detector. The graph below that shows the current value of one integral while it is calculated which is only monitored during simulation. The last two graphs are the averaged integration result and the bunch counter that is used to monitor the module when it is tested on the FPGA. The signals are set to have a frequency of 1.5MHz and noise in the range of $\pm 0.2V$. The SMA filter averages over the last eight ADC inputs and every bunch is to be integrated.
In all three figures, it can be observed that the graph for the current integral value follows the expected behavior of the integral of a sine wave or the other signal shapes which are offset to have their minimum at zero. Furthermore, the averaged integral is only updated to a non-zero result after the first eight bunches, since only then, the average is written to the output. However, the scaling of the graphs is not shown in the figures. For example, for Figure 3.11 the minimum of the raw signal is −4226 (−0.52V). As expected, the SMA has a higher minimum at −3083 (−0.38V). But the highest minimum is found for the EMA at 192 (0.02V). Here, it becomes clear again that the output from the EMA module is not suitable for baseline detection.

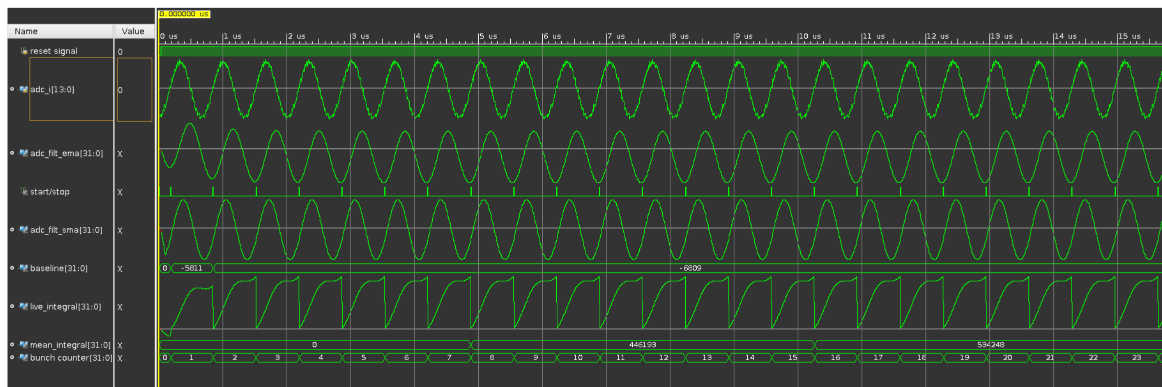


Figure 3.9.: Simulation results for sinusoidal input, SMA with $k = 3$ and no skipped bunches (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter)

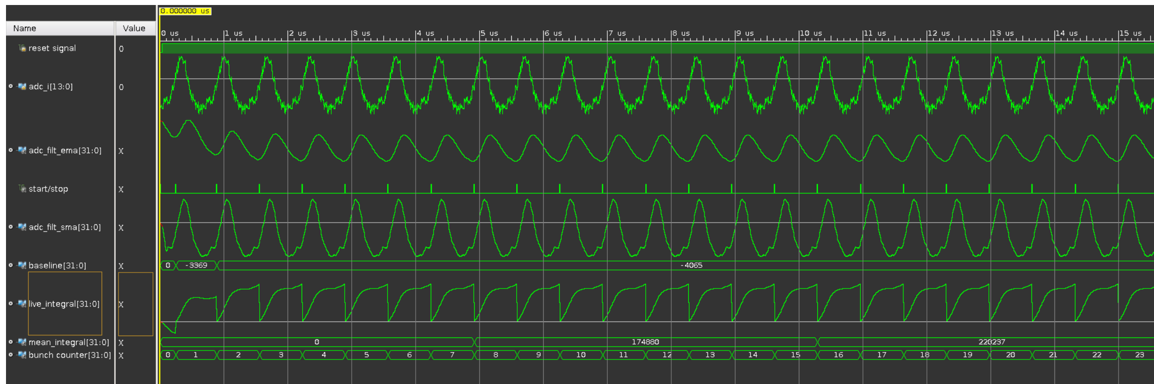


Figure 3.10.: Simulation results for gauss shaped input, SMA with $k = 3$ and no skipped bunches (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter)

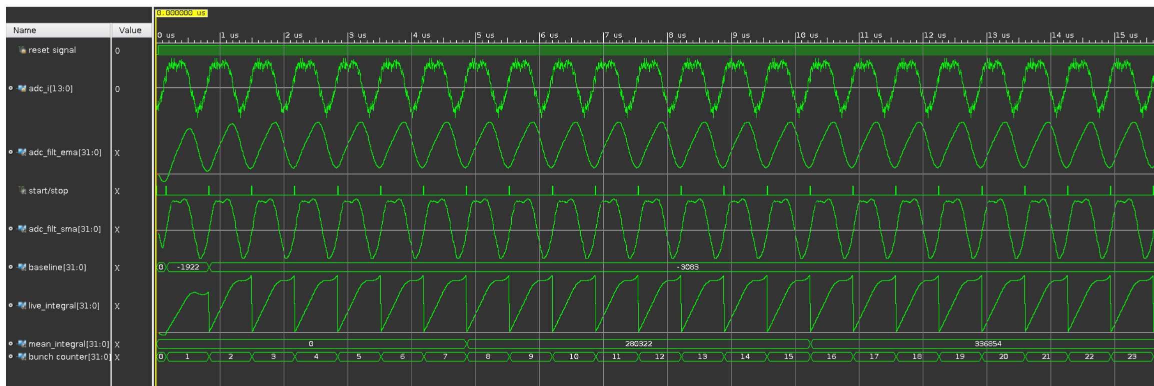


Figure 3.11.: Simulation results for signal shape with two peaks, SMA with $k = 3$ and no skipped bunches (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter)

On top of that, all three simulation results differ from the expected behavior at the beginning of the simulation, right after the reset signal is triggered. A magnified version of that can be seen in Figure 3.12. Here one can see that specifically the baseline detector and the EMA need several clock cycles to arrive at the desired output. For the EMA the reason is that with a weighting factor of 1/64 the algorithm needs at least 64 clock cycles à 8ns to adjust to the current signal input. For the baseline detector the first baseline value is the minimum signal input up to the first detected bunch. This is not a whole bunch and therefore the baseline is not correct. This results in a different correction of the integration result. However, after a few beam bunches, the output has adjusted to the actual input signal.

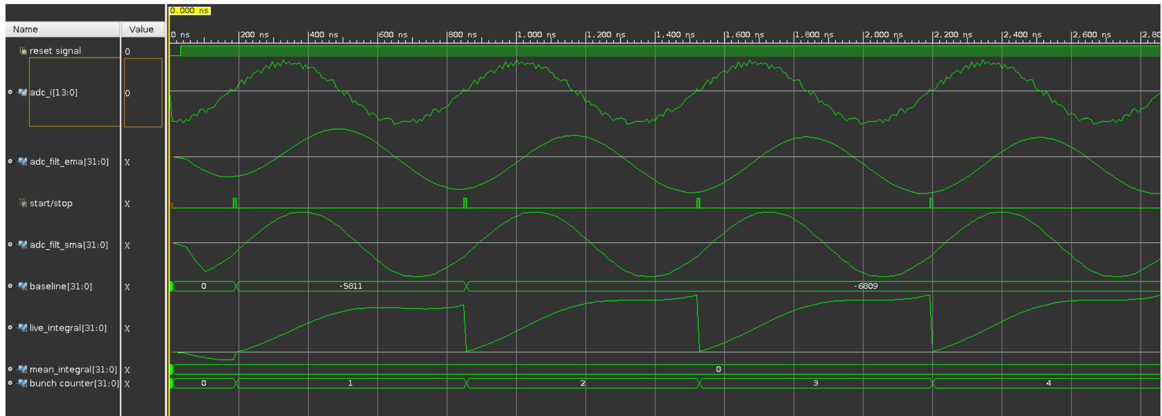


Figure 3.12.: Simulation results right after reset signal for sinusoidal input, SMA with $k = 3$ and no skipped bunches, visible deviation of the baseline value in row 5 and lower end results for integration in row 6 for the first signal period (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter)

The simulation for an integration, that should skip every second beam bunch is depicted in Figure 3.13. One can clearly see, how only every second bunch is integrated and also the bunch counter is only incremented for every second bunch.
In conclusion, the simulations show, that the module can handle the signal noise, find the arbitrary baseline and can skip bunches if needed. Therefore, the design can be implemented into the whole design architecture of the pre-existing top-module which the integration module is a part of. Afterwards, the top-module with all the sub-modules including the one for integration is synthesized and implemented so that a bitstream can be generated. This bitstream can be loaded onto the Redpitaya board for tests of the FPGA configuration on actual hardware. This will be discussed in the next section.

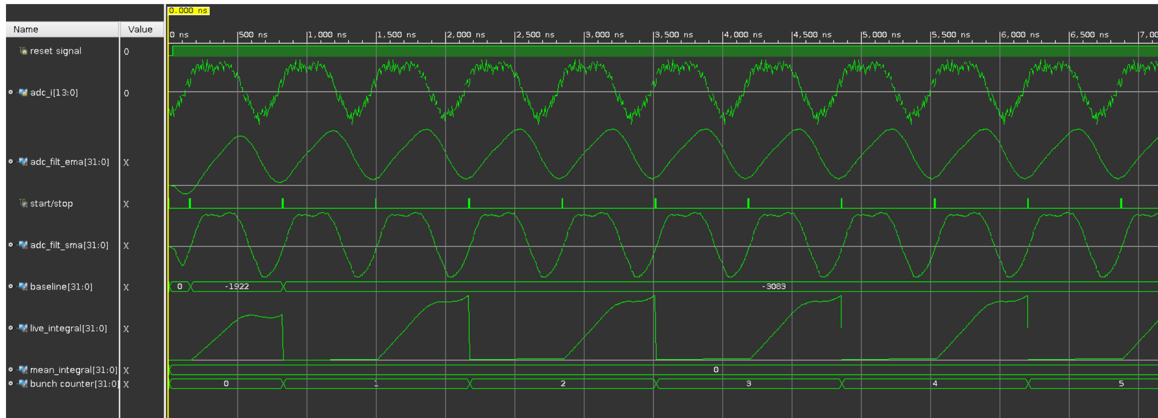


Figure 3.13.: Simulation results for signal shape with two peaks, SMA with $k = 3$ and every second bunch skipped (row 1: reset signal, row 2: ADC input signal, row 3: EMA output, row 4: peak detector output, row 5: SMA output, row 6: baseline detector output, row 7: current value of one integral while it is calculated, row 8: averaged integration result, row 9: bunch counter)

3.7.3. Implementation and testing on hardware

In this section, the progress of loading the bitstream on the FPGA and the results are explained.
To test the FPGA configuration on the Redpitaya board, the following setup is used. The Redpitaya board described in section 2.3.1 is connected to a laptop using a LAN cable. An SMA cable is used to connect the DAC output A to the ADC input A of the board. A 8GB Micro SD card with the latest stable image for STEMlab 125-14 written onto it is inserted into the respective slot and the power adapter is connected to the Redpitaya and a power source.
Via the established Ethernet connection the .bin-file which contains the bitstream for the top-module can be copied onto the SD card and used to configure the FPGA.
To generate the signal to test the FPGA configuration, the signal generation utility of the Redpitaya board is used. This allows the user to generate a simple sinusoidal signal with adjustable amplitude, period length and output channel (A or B). However, the addition of a simulated noise to this signal is not possible. The Application Programming Interface (API) has been adjusted so that the inputs and outputs of the integration module can be accessed from certain addresses by using the `monitor` command. Therefore, a small shell script that is saved on the SD card can monitor these addresses and write the values in a separate file. This file then contains the timestamp of the value, the averaged integral, beam count and baseline value at that time as well as the user inputs for skipping bunches and the smoothing factor for the SMA. Since only the ADC channel A is connected to the generated signal, only the data for channel A will be analyzed.
The reason, why only the relatively constant or linear rising parameters are monitored,

is that the monitor utility of the Redpitaya can read a register with approximately 30Hz which is too small to get meaningful information out of a signal with frequencies up to 2MHz. The values that are suitable to give feedback on the functionality of the module are, firstly, the baseline value and integral result which should be near constant for a signal with constant amplitude. Secondly, the bunch counter should rise in a linear fashion and with a slope equal to the frequency of the input signal.
The tested wave forms are listed in Table 3.1 and the testing results are listed in Table 3.2. For every signal type 50 samples of registers for channel A have been read by the shell script. The 50 monitored values for the baseline and the integral have been averaged, while the bunch count has been subjected to a linear fit with the slope being the frequency of the input signal f_i divided by the number of skipped bunches $n_{skipped}$ in between the integrated bunches plus one:

$$n_{bunches} = \frac{f_i}{n_{skipped} + 1} \cdot t \tag{3.22}$$

The calculated frequency of the signal, is shown in Table 3.2 and can be compared to the configured input frequency. The linear behavior of the beam count can also be seen in Figure 3.14. The overall results show, that the algorithm works for the chosen input signals as the baseline, frequency and integration result for each test are close to the expected values. Especially the detected baseline for SMA-length 4 and amplitude 0.9 show a maximum deviation of 0.3%. Only for tests 5, 8 and 9, the detected baseline deviates up to 5% from the actual value since either the SMA-length is increased, resulting in heavier filtering of the input, or the signal amplitude is decreased, so any form of noise that could be introduced from the SMA cable has a larger impact on the baseline detection. Since the baseline value is used to correct the integration result, these effects can also be noticed in the integration results. Furthermore, the expected influence of a lower signal frequency that is closer to the cutoff frequency of the EMA on the integration result can be seen, as a slight increase in deviation of the integration result form the expected value.
Overall, the module seems to be working as demanded. However, tests with significantly more noisy signals (simulated or technically induced noise) need to be conducted, to thoroughly test the filters and the whole module. Unfortunately, that will not be part of this thesis.

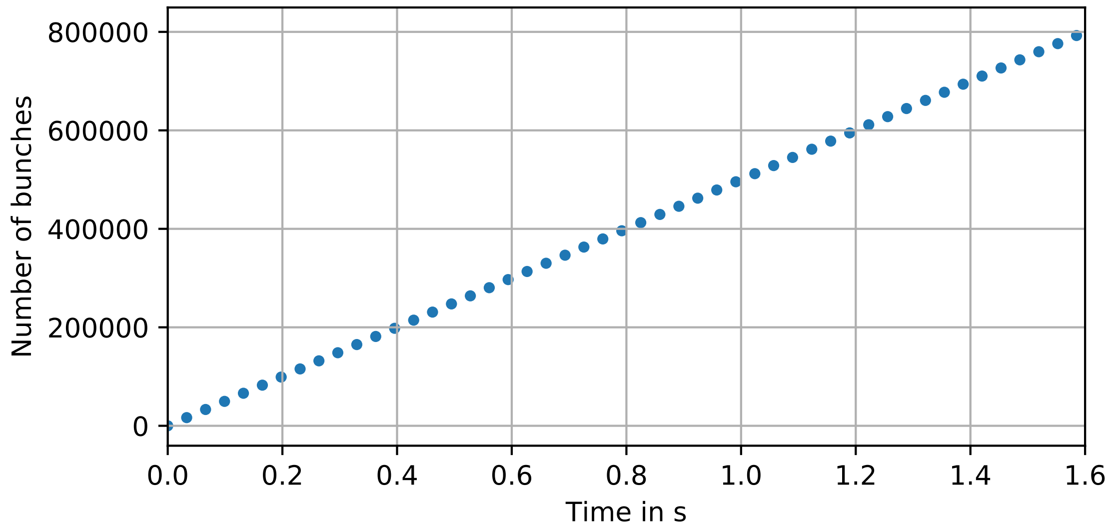


Figure 3.14.: Bunch counter output from tests on the Redpitaya board showing linear behavior as expected

Table 3.1.: Different signals and settings used for testing

Test	Amplitude [V]	Frequency [MHz]	Integral [V·ns]	Skipping	SMA-length
1	0.9	0.5	1800	0	4
2	0.9	1.0	900	0	4
3	0.9	1.5	600	0	4
4	0.9	2.0	450	0	4
5	0.5	1.5	333	0	4
6	0.9	1.5	600	3	4
7	0.9	1.5	600	6	4
8	0.9	1.5	600	0	8
9	0.9	1.5	600	0	16

Table 3.2.: Averaged testing results for different signals with their percentage-wise deviation form the expected value

Test	Baseline [V]	Dev. [%]	Frequency [MHz]	Dev. [%]	Integral [V·ns]	Dev. [%]
1	−0.8972	0.31	0.50001	0.002	1711.06	4.94
2	−0.9009	0.1	0.99999	0.001	859.32	4.52
3	−0.9019	0.22	1.50003	0.002	573.19	4.47
4	−0.9005	0.06	1.99999	0.0	429.57	4.54
5	−0.519	3.79	1.50003	0.002	317.72	4.68
6	−0.9008	0.09	1.50005	0.003	572.91	4.52
7	−0.9007	0.08	1.50002	0.001	573.08	4.49
8	−0.8963	0.41	1.49975	0.017	568.38	5.27
9	−0.8538	5.13	1.49996	0.003	541.19	9.8

4. Results and Outlook

In this last chapter the results of this thesis will be summarized and possible adjustments will be discussed. This will be followed by a short outlook on the usefulness of the developed algorithm.

To sum up the developed module, the required features, i.e. being able to integrate a signal in MHz-range for each period and dealing with missing down times and noise of the signal, have been considered while designing the algorithm. The simulations show, that the module should also be able to handle noise. However, this still needs to be verified with testing on the actual FPGA since with the current setup the code can only be tested with signals with nearly no noise. The benefit of the used algorithm is, that by integrating over the raw ADC input signal, possible distortions of the signal from the added filters do not influence the integration result. Furthermore the EMA that is used, requires very few memory space. Additionally, it is quite simple to replace a sub-module in the case of a superior algorithm, e.g. another digital filter or a more elaborate peak detector, being developed, since the module follows a clear hierarchy defined by the structural module explained in section 3.1.

However, there are still aspects that might need adjustments as for a full validation of the module, more elaborated tests on hardware are required. For example, the weighting factor of the EMA might need to be increased, if the noise filtering does not prove itself sufficient. Also, the weighting factor could be changed into an input parameter for the user similar to the SMA-length. That would enable a more flexible adjustment of the filter to the noise-to-signal ratio. Regarding the timing of the module, the FPGA timing for logic operations has to be checked and the possible occurrence of race conditions has to be considered.

This module provides real time insight into the number of particles in one beam bunch at COSY as required by the task of this thesis. The different sub-modules developed for this thesis can also be used in different contexts, when handling other noisy signals for example that are processed with a FPGA similar to the one used. The module as a whole (with all sub-modules together) is likely only useful for signals similar to the one this thesis is aimed at, since e.g. for a signal with longer downtime there are simpler methods to detect a baseline.

As a whole, the module designed for this thesis can be seen as a basis for further development that can be elaborated and improved as necessary.

Bibliography

[1] R.C. Webber. “Charged particle beam current monitoring tutorial”. In: *The 6th workshop on beam instrumentation.* (Vancouver, British Columbia (Canada)). Ed. by George H. MacKenzie, Bill Rawnsley, Jana Thomson. Vol. 333. AIP, May 1994.

[2] Mathis Beyß. “Detection and Analysis of Recombination Rates during Electron Cooling at COSY”. Master Thesis. RWTH Aachen III. Physikalisches Institut B, 2019.

[3] P. Lenisa, F. Rathmann, L. Barion et al. “Low-energy spin-physics experiments with polarized beams and targets at the COSY storage ring”. In: *EPJ Techniques and Instrumentation* 6.2 (2019). DOI: `https://doi.org/10.1140/epjti/s40485-019-0051-y`.

[4] H. Stockhorst, U. Bechstedt, J. Dietrich, R. Maier, S. Martin, D. Prasuhn, A. Schnase, H.Schneider, R. Tölle. “The cooler synchrotron COSY facility”. In: *Proceedings of the 1997 Particle Accelerator Conference.* (Vancouver, British Columbia (Canada)). Ed. by M. Comyn, M.K. Craddock, M. Reiser, J. Thomson. Vol. 1. IEEE, May 1997.

[5] Y. Valdau, L. Eltcov, S. Trusov, S. Mikirtytchiants. “Development of High Resolution Beam Current Measurement System for COSY-Jülich”. In: *Proceedings, 5th International Beam Instrumentation Conference (IBIC 2016) : Barcelona, Spain, September 11-15, 2016.* (Barcelona, Spain). Ed. by Isidre Costa et al. JACoW, 2017. DOI: `10.18429/JACoW-IBIC2016-TUPG41`.

[6] A. Lehrach. “Beam- and Spin Dynamics for Hadron Storage Rings”. In: *Microscopy and Microanalysis* 21 (June 2015), pp. 24–28. DOI: `10.1017/S1431927615013082`.

[7] Peter Forck. *Lecture Notes on Beam Instrumentation and Diagnostics.* 2011.

[8] *Fast Current Transformer User’s Manual.* Bergoz Instrumentation.

[9] “State-of-the-Art Programmable Logic”. In: *Designing with Xilinx ® FPGAs.* Ed. by S. Churiwala. Springer International Publishing Switzerland, 2017, pp. 1–15.

[10] *Red Pitaya Documentation - 3.1.1. STEMlab boards comparison.* 2017. URL: `https://redpitaya.readthedocs.io/en/latest/developerGuide/125-10/vs.html` (visited on 04/06/2020).

[11] *Quick Start Guide to Verilog.* Springer Nature Switzerland, 2019.

[12] *FPGA Design Flow Overview*. 2008. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx10/isehelp/ise_c_fpga_design_flow_overview.htm (visited on 04/06/2020).

[13] "Simulation". In: *Designing with Xilinx® FPGAs*. Ed. by S. Churiwala. Springer International Publishing Switzerland, 2017, pp. 127–140.

[14] *Digitale Signalverarbeitung mit MATLAB*. Springer Vieweg, 2019.

A. Appendix

A.1. Verilog modules

A.1.1. Structural module `new_integration.v`

Listing A.1: test listing

```
module new_integration(
      input                             adc_clk_i,
      input                             adc_rstn_i,
      input   signed  [14-1:0]          adc_i,              //ADC
         input
      input           [32-1:0]          skip,               //number
         of bunches to periodically skip (when multiple
         beams are measured)
      input           [32-1:0]          pow,
      output  signed  [32-1:0]          integral_o,
         //integral output for each time step- simulation
         only
      output  signed  [32-1:0]          mean_integral_o,
         //averaged integral output
      //output                              dect_o,
         //auxiliary - simulation only
      output          [32-1:0]          bc,  // for testing
      output  signed  [32-1:0]          baseline_o    //auxiliary
         - simulation and testing only
      //output  signed  [32-1:0]          adc_filt_ema_o,
         //auxiliary - smoothed signal for peak detection -
         simulation only
      //output  signed  [32-1:0]          adc_filt_sma_o
         //auxiliary - smoothed signal for baseline
         detection - simulation only

   );

   localparam int=32;
   wire     signed [32-1:0]       filt_ema;
   wire     signed [32-1:0]       filt_sma;
```

```
wire                              count_dect;
wire     signed   [32-1:0]        baseline;
wire     signed   [32-1:0]        adc_i_32_bit;

assign adc_i_32_bit = adc_i;

ema ema1(
    adc_clk_i,     //in
    adc_rstn_i,    //in
    adc_i_32_bit,           //in
    filt_ema       //out
    );

peak_detector p1(
    adc_clk_i,     //in
    adc_rstn_i,    //in
    filt_ema,      //in
    //bc,           //out
    count_dect     //out
    );

sma sma1(
    adc_clk_i,     //in
    adc_rstn_i,    //in
    adc_i_32_bit,           //in
    //sma_delay,    //in
    pow,           //in
    filt_sma       //out
    );

baseline_detector b1(
    adc_clk_i,     //in
    adc_rstn_i,    //in
    filt_sma,      //in
    count_dect,    //in
    baseline       //out
    );

integration i1(
     adc_clk_i,    //in
     adc_rstn_i,   //in
     adc_i_32_bit,          //in
     baseline,     //in
     count_dect,   //in
```

```
        skip,           //in
        //integral_o,    //Simulation only - out
        mean_integral_o, //out
        bc               //out
        );
    assign baseline_o=baseline;

    /*
    //Simulation only
    assign adc_filt_ema_o=filt_ema;
    assign adc_filt_sma_o=filt_sma;
    assign dect_o = count_dect;
    */

endmodule
```

A.1.2. EMA filter module `ema.v`

Listing A.2: test listing

```
module ema(
        input                    adc_clk_i,  //adc clock
        input                    adc_rstn_i, //reset or
           initialization
        input  signed  [32-1:0] adc_i,      //raw input data
        output signed  [32-1:0] adc_filt    //Smoothed
           output data
    );

    localparam width=32;
    reg     signed  [width-1:0]  filt;
    reg     signed  [width-1:0]  filt_state;

    always@(posedge adc_clk_i) begin
        if (adc_rstn_i == 1'b0) begin
            filt            <=   32'h0;
            filt_state      <=   32'h0;
        end else begin
            filt_state      <=   filt_state + adc_i - filt;
            filt            <=   filt_state >>> 6;
        end
    end

    assign adc_filt = filt;
```

```
endmodule
```

A.1.3. Peak detection module `peak_detector.v`

Listing A.3: test listing

```
module  peak_detector(
        input                           adc_clk_i,  //adc clock
        input                           adc_rstn_i, //reset or
           initialization
        input   signed  [32-1:0]        adc_i,      //filtered
           input data from ema module
        output                          count_dect  //Signal for
           when a pulse is detected
    );

    reg     signed  [32-1:0]    peak_p;
    reg     signed  [32-1:0]    peak_n;
    reg                         dect;
    reg             [32-1:0]    counter;
    reg                         arm;        //rising or
       falling arm

    localparam  RISING  = 1'b1;
    localparam  FALLING = 1'b0;

    always@(posedge adc_clk_i) begin
        if (adc_rstn_i==1'b0) begin
            peak_p          <=  32'h0;
            peak_n          <=  32'h0;
            //counter         <=  -32'h0;
            dect            <=  1'b0;
            arm             <=  1'b0;
        end else begin
            case(arm)
                RISING:
                    //search for largest value
                    if (adc_i >= peak_p) begin
                        peak_p          <=  adc_i;
                        dect            <=  1'b0;
                    end else begin
                        arm             <=  FALLING;
                        peak_n          <=  peak_p;
                    end
                FALLING:
```

```
                    //search for smallest value
                    if (adc_i <= peak_n) begin
                        peak_n          <=   adc_i;
                        dect            <=   1'b0;
                    end else begin
                        dect            <=   1'b1;
                           //signal peak detection when
                           negative peak is found
                        //counter          <=   counter +
                           32'h1;
                        arm             <=   RISING;
                        peak_p          <=   peak_n;
                    end
            endcase
        end
    end

    assign count_dect = dect;

endmodule
```

A.1.4. SMA filter module `sma.v`

Listing A.4: test listing

```
module sma(
        input                       adc_clk_i,   //adc clock
        input                       adc_rstn_i,  //reset or
           initialization
        input  signed  [32-1:0] adc_i,       //raw input data
        input          [32-1:0] pow,         //length of
           shift register for smoothing in powers of two
        output signed  [32-1:0] adc_filt     //Smoothed
           output data
    );

    reg    signed  [32-1:0] shift_reg [32-1:0];
    reg    signed  [32-1:0] filt_32;
    reg    signed  [32-1:0] filt_16;
    reg    signed  [32-1:0] filt_8;
    reg    signed  [32-1:0] filt_4;
    reg    signed  [32-1:0] filt_2;
    reg    signed  [32-1:0] filt;
    integer                 i = 0;
    integer                 j = 0;
```

```
always@(posedge adc_clk_i) begin
    if (adc_rstn_i == 1'b0) begin
        for (i=0; i < 32; i = i+1)
            shift_reg[i]    <=  32'h0;
    end else begin
        shift_reg[0]     <=    adc_i ;
        for (i=0; i < 32; i = i+1)
            shift_reg[i+1] <=    shift_reg[i];
    end
end

always@(posedge adc_clk_i) begin
    if (adc_rstn_i == 1'b0) begin
        filt_32        =    32'h0;
        filt_16        =    32'h0;
        filt_8         =    32'h0;
        filt_4         =    32'h0;
        filt_2         =    32'h0;
        filt           =    32'h0;
    end else begin
        filt_32            =    32'h0;
        filt_16            =    32'h0;
        filt_8             =    32'h0;
        filt_4             =    32'h0;
        filt_2             =    32'h0;
        for (j=0; j < 32; j = j+1)
            filt_32 = filt_32 + shift_reg[j];
        for (j=0; j < 16; j = j+1)
            filt_16 = filt_16 + shift_reg[j];
        for (j=0; j < 8; j = j+1)
            filt_8  = filt_8 + shift_reg[j];
        for (j=0; j < 4; j = j+1)
            filt_4  = filt_4 + shift_reg[j];
        for (j=0; j < 2; j = j+1)
            filt_2  = filt_2 + shift_reg[j];
        if (pow == 32'd5) begin
            filt = filt_32 >>> pow;
        end else if (pow == 32'd4) begin
            filt = filt_16 >>> pow;
        end else if (pow == 32'd3) begin
            filt = filt_8 >>> pow;
        end else if (pow == 32'd2) begin
            filt = filt_4 >>> pow;
```

```
            end else if (pow == 32'd1) begin
                filt = filt_2 >>> pow;
            end
        end
    end

    assign adc_filt = filt;

endmodule
```

A.1.5. Baseline detection module `baseline_detector.v`

Listing A.5: test listing

```
module baseline_detector(
        input                           adc_clk_i,  // ADC clock
        input                           adc_rstn_i, // ADC reset
           - active low
        input   signed  [32-1:0]        filt,       // ADC input
           data filtered with SMA
        input                           dect,       // pulse
           detected
        output  signed  [32-1:0]        baseline    // Baseline
           output value
    );

    reg     signed  [32-1:0]    base_1;         //storage
       for preliminary baseline
    reg     signed  [32-1:0]    base_2;         //final
       baseline value

    always@(posedge adc_clk_i) begin
        if (adc_rstn_i == 1'b0) begin
            base_1          <=  32'h0;
            base_2          <=  32'h0;
        end else begin
            if (dect == 1'b0) begin
                if (filt < base_1) begin    //search for
                   smallest value in filtered signal
                    base_1      <=  filt;
                end
            end else begin      //if Peak is found, ouput
               baseline value and reset preliminary baseline
               value
                base_2      <=  base_1;
```

```
                base_1        <=    filt;
            end
        end
    end

    assign baseline   =     base_2;

endmodule
```

A.1.6. Integration module `integration.v`

Listing A.6: test listing

```
module integration(
        input                           adc_clk_i,
        input                           adc_rstn_i,
        input   signed  [32-1:0]        adc_i,
           //ADC input
        input   signed  [32-1:0]        baseline,
           //baseline value (from baseline detector)
        input                           dect,
           //bunch detected (from peak detector)
        input           [32-1:0]        skip,
           //number of bunches to skip between measurements
        //output   signed  [32-1:0]     integral,
           //auxiliary integral output after each clock
           (simulation only)
        output  signed  [32-1:0]        mean_integral_o,
           //Integral output
        output          [32-1:0]        pulse_cnt
           //Pulse counter - for testing
    );

    localparam int=32;
    reg     signed  [int-1:0]       single_integral;
    reg     signed  [int-1:0]       mean_integral;
    reg     signed  [int-1:0]       mean_integral_fin;
    //reg      signed  [32-1:0]       per_count;
    reg             [3-1:0]         integrated; //counter for
       integrals registered (after 8 integrated bunches the
       averaged integral is given out)
    reg             [32-1:0]        skipped;    //counter for
       skipped bunches
    reg             [32-1:0]        counter;
   // reg                           last_MSB;
```

```
// assign bl_correct = adc_i - baseline;

 always@(posedge adc_clk_i) begin
     if (adc_rstn_i == 1'b0) begin //reset all registers
        to zero
         single_integral     =   -32'h0;
         mean_integral       =   -32'h0;
         mean_integral_fin   =   -32'h0;
         //per_count           =   -32'h0;
         integrated          =   3'b000;
         skipped             =   32'b0;
         counter             =   32'h0;
     end else begin
         if (skipped == skip) begin  //check if required
            bunches have been skipped
             if (dect == 1'b1) begin     //write integral
                and start new integral when peak is
                detected
                 single_integral =   single_integral;
                                     //divide final
                    integral value for the averaged
                    integral
                 counter         =   counter + 1;
                 skipped         =   32'b0;

                    //restart counter for skipped bunches
                 if (integrated < 3'b111) begin
                    //averaging condition
                     //mean_integral_fin   =   -32'h0;
                        //to make simulation output
                        easier to read
                     mean_integral       =
                        mean_integral + single_integral;
                     integrated          =   integrated +
                        3'b001;
                 end else begin
                     mean_integral_fin   =
                        (mean_integral + single_integral)
                        >>> 3;//ouput of final, averaged
                        integral value
                     mean_integral       =   -32'h0;
                                                 //restart
                        averaging
```

```
                        integrated          =    3'b000;
                    end
                    single_integral     =    adc_i - baseline;
                end else begin              //ongoing
                    integral calculation when no peak is
                    detected
                    single_integral     =    single_integral
                        + adc_i - baseline;    //integration
                        with baseline correction
                end
            end else begin              //skip current bunch
                if (dect == 1'b1) begin      //increase
                    skipped bunch counter
                    skipped             =    skipped + 1;
                end
            end

        end
    end

    //assign integral = single_integral; //Simulation only
    assign mean_integral_o = mean_integral_fin;
    assign pulse_cnt = counter;

endmodule
```

A.2. Bash script

Listing A.7: test listing

```
#!/bin/bash

read -p "Insert number of data points: " len

touch ../tmp/local_file
>../tmp/local_file

for ((i=0;i<len;i++))
do
        echo "$(date +%S%N) $(monitor 0x40101150) $(monitor
            0x40101158) $(monitor 0x40101160) $(monitor
            0x40101168) $(monitor 0x4010116C)" >>
            ../tmp/local_file
done
```