Development of a trigger system based on FPGA logic

by

Henrik Matschat

Bachelor thesis submitted in

July 2016

to the
Faculty of Mathematics, Computer Science and Natural Science
Department of Physics
at
RWTH Aachen University
Thesis supervisor:
Prof. Dr. rer. nat. Jörg Pretz

Contents

1. Introduction

This thesis describes the programming and testing of a trigger system that is designed to run on a field-programmable gate array, often abbreviated as FPGA. The trigger system uses the logical blocks available on the FPGA to evaluate signals from a particle detector and generates an output signal if certain conditions are met. The output signal contains information about the characteristics of the particle penetration. The trigger is specifically designed for the WASA detector at the **co**oler **sy**nchrotron and storage ring COSY in Juelich.

Currently, the WASA detector is prepared for upcoming experiments that aim at building a polarimeter database for pC and dC scattering[1]. The plan is to use the trigger system to evaluate the trajectory of scattered particles by categorizing them into four directions and select which events shall be saved. The database is then built by using the results from the offline analysis.

The following chapters describe the design of the trigger system in compliance with the WASA[2] detector by giving an overview of the detector layout and how the signals from the detector are evaluated in order to generate a corresponding trigger signal. Different functions of the trigger system are described and the processing of the signals is illustrated. A FIFO, a special data buffer that stores signals from different clock pulses, is implemented to allow the user to set up individual delays in the evaluation process.

Computer simulations of the trigger system are used to assure that the signals are evaluated and processed as desired. Eventually, the trigger system is transferred to the FPGA and multiple tests are performed that ascertain the correct functionality of the FPGA trigger system.

[1] proton-carbon respectively deuteron-carbon
[2] Wide Angle Shower Apparatus

2. Theoretical and experimental background

2.1. Particle accelerators and storage rings

One way to cause and observe particle scattering and decays is the acceleration of particles. This acceleration can be achieved in various ways. Two constructions are used in the focus of this thesis. One is the cyclotron and the other one the synchrotron. Therefore, a short overview of these methods to accelerate particles shall be given at this point.

2.1.1. Cyclotron

A cyclotron consists of a cylindrical vacuum chamber in the x-y-plane between a magnet that creates a static magnetic field in z-direction. The cylinder is separated into two equal halves with a gap in between as shown in fig. 2.1. Between the two halves a radio frequency voltage is applied. The positive ions are inserted in the middle between the two halves and the electric field causes them to move towards the negative half. Additionally, the magnetic field causes movement along a semi-cycle inside the halves, which are made of metal so that the electric field does not influence the particle trajectory.

By equating the Lorentz force and the centripetal force one finds that the radius of the semi-cycle amounts to

$$r = \frac{mv}{qB} \tag{2.1}$$

and the frequency is

$$\omega = \frac{2\pi}{t} = \frac{qb}{m}. \tag{2.2}$$

By setting the frequency of the electric field U_E between the two halves to match the cyclotron frequency 2.2, it is possible to switch the polarity of the electric field simultaneously with the particle leaving one of the two halves. Therefore, the frequency of the electric field has to be

$$f_c = \frac{\omega}{2\pi} = \frac{qb}{2\pi m}. \tag{2.3}$$

This configuration has the effect that the particles will be accelerated whenever they cross the gap between the two half-cylinders. The acceleration will increase the radius of the semi-cycles and, therefore, the trajectory forms a spiral. At some point, however, the radius reaches the rim of the half-cylinders. At this point the particles leave the cyclotron and can be used in the further experimental setup. In the non-relativistic case the energy gain is

$$E_{kin} = \frac{1}{2} \cdot m \cdot v^2 = \frac{q^2}{2m} \cdot (r_{max} \cdot B)^2 \tag{2.4}$$

where r_{max} is the largest possible radius the particles can reach inside the cyclotron. When considering higher energies and therefore relativistic effects, a deviation between the frequency of the electric field and the frequency with which the particles arrive at the gap between the half-cylinders occurs. To avoid this problem, the frequency of the electric field is constantly reduced to match the reduced frequency of the particles [1].

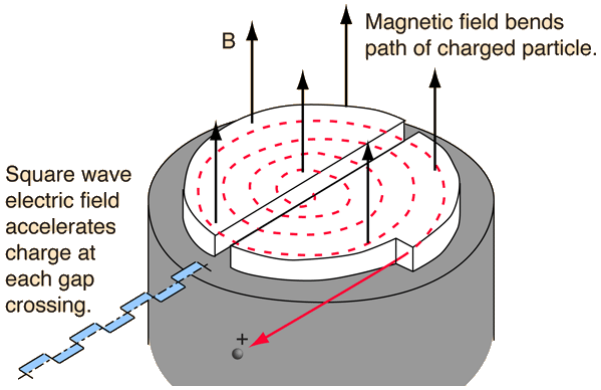


Figure 2.1.: A cyclotron with its spiral particle trajectory (dotted line) and the magnetic field (black arrows) as well as pulses representing the change in the polarization of the electric field. [2]

2.1.2. Synchrotron

Another way to accelerate particles is the synchrotron. As shown before, with a cyclotron it is necessary to increase the magnetic field or the radius in order to reach higher energies. A different concept is used in a synchrotron. The particles are injected into the synchrotron tangentially to the trajectory, on which they will later circulate with a given velocity. Resting particles can not be accelerated with a synchrotron due to its structure and technical restrictions, among others hysteresis. The arcs, as shown in fig. 2.2, house magnets which direct the particle beam. Similar to the cyclotron, electric fields are applied between at least one of the arcs to accelerate the particles. Additionally, quadrupole magnets can be used to focus the beams. To ensure that the particles will stay on track while they are accelerated, the magnetic field strength is adjusted according to the energy of the particles [3]. Using formula 2.1 with relativistic energies, it can be shown that a particle with a momentum of $p = mv$ moving along a circular path will need a radius of

$$r = \frac{mv}{qB} = \frac{m_0 c}{qB} \cdot \sqrt{\left(\frac{E_{kin}}{m_0 c^2}\right)^2 + 2\frac{E_{kin}}{m_0 c^2}} = \frac{E_{kin}}{qBc} \cdot \sqrt{1 + 2\frac{m_0 c^2}{E_{kin}}} \tag{2.5}$$

to stay on track. For relativistic energies, where $E_{kin} \gg m_0 c^2$, the approximation

$$r \approx \frac{E_{kin}}{qBc} \tag{2.6}$$

is suitable to describe the radius of the particle trajectory. Taking into consideration that the radius of the synchrotron as well as the charge of the particles shall be constant, it can easily be found that the magnetic field strength has to be proportional to the energy of the particle:

$$r = const. = \frac{E_{kin}}{qBc} \rightarrow E_{kin} \sim B. \tag{2.7}$$

Therefore, the magnetic field and the energy have to be synchronized to ensure that particles with increasing velocities do not leave the ring. The radio frequency f_{acc} of the acceleration voltage also requires an adjustment depending on the velocity of the particles. These two dependencies lead to a relation between the frequency and the magnetic field:

$$f = \frac{k}{2\pi\sqrt{\left(\frac{r}{c}\right)^2 + \left(\frac{m_0}{qrB}\right)^2}}, \tag{2.8}$$

where k is harmonic number of the frequency. Thus, a synchronization between the frequency of the electric field and the magnetic field strength is required [1].

Let U be the acceleration voltage and ϕ the phase of the electric field, the energy gain of the particles is described by:

$$\Delta E = q \cdot U \cdot \sin(\phi) - \Delta E_{loss}. \tag{2.9}$$

Taking into consideration the financial and technical limitations, synchrotron radiation limits the achievable energy to about 100 GeV for electrons[1], while for protons and other heavier particles energies of more than 1000 GeV can be achieved[2] [3].

2.1.3. Fixed target scattering for high precision physics

In order to generate scattering and decays, two major experimental setups can be used. The particle beam can be directed onto a fixed target or two particle beams can be collided head-on with each other. The fixed target brings along the advantage that the target can be chosen from a wide range of materials. Furthermore, the thickness of the target can be adjusted more flexible. The luminosity of fixed target experiments is a product of the thickness of the target and the beam intensity and can therefore reach high values. The energies available in such experiments is, however, much lower than in a colliding beam experiment, because a lot of the energy of the beam is lost in the movement of the center-of-mass frame.

The center-of-mass energy is defined as

$$\sqrt{s} = \sqrt{\left(\begin{pmatrix}E_1\\ \vec{p}_1\end{pmatrix} + \begin{pmatrix}E_2\\ \vec{p}_2\end{pmatrix}\right)^2}. \tag{2.10}$$

[1] Reached by the LEP
[2] for instance with the LHC

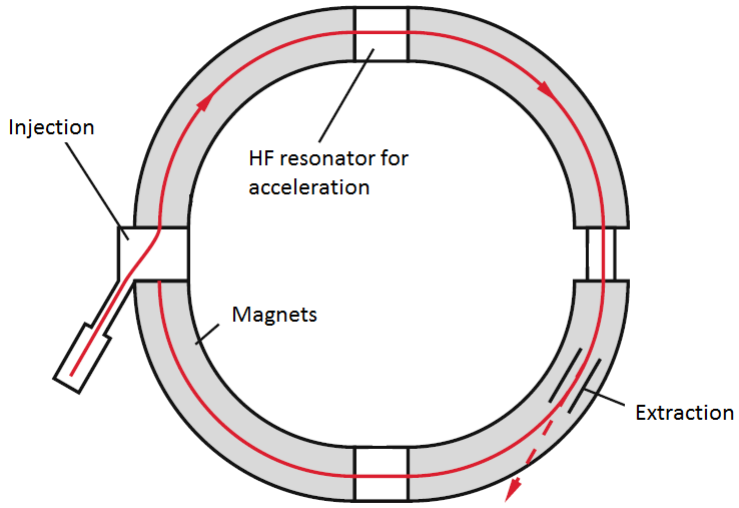


Figure 2.2.: The layout of a synchrotron with four magnets, the injection at the left-hand side as well as an indicated ejection at the lower right arc [1, p. 75]

In case of colliding beams, where both beams have the same energy E and collide head-on, the center-of-mass energy amounts to

$$\sqrt{s} = \sqrt{4 \cdot E^2} = 2E, \tag{2.11}$$

whereas the energy using a fixed target with a mass of m is

$$\sqrt{s} = \sqrt{\left(\begin{pmatrix} E \\ \vec{p} \end{pmatrix} + \begin{pmatrix} m \\ \vec{0} \end{pmatrix}\right)} = \sqrt{\begin{pmatrix} E+m \\ \vec{p} \end{pmatrix}} = \sqrt{2mE + 2m^2}. \tag{2.12}$$

If the mass is neglected, the center-of-mass energy of a fixed target experiment is

$$\sqrt{s} = \sqrt{2mE}. \tag{2.13}$$

An essential part of the energy goes into moving forward the particles that resulted from the impact. For the experiments at hand, it is reasonable to use fixed targets.

2.2. WASA at COSY

A trigger can be used to compare signals from, in this case, particle detectors to predefined conditions and generate an output signal if the conditions are met. The output signal can in turn be used to run further processes in the evaluation of the signals. By using the trigger, accidental or unwanted signals can be sorted out and only the desired signals will be evaluated. The name *trigger* refers to the function of starting, i.e. triggering, further steps in the evaluation process if the conditions are met.

The trigger at-hand shall be used as part of experiments with the WASA detector at the **CO**oler **SY**nchrotron and storage ring COSY in Juelich, shown in fig. 2.3. The particle beams produced in COSY consist of deuterons or protons. The cyclotron JULIC serves as a first accelerator and injector for the particles into the storage ring. An ion source supplies JULIC with H^- and D^-. The cyclotron accelerates the protons and deuterons to an energy of 0.3 GeV/c respectively 0.54 GeV/c.

Further acceleration of the proton and deuteron beams takes place in the storage ring, where the beams are guided by dipole magnets in the curves and focused by quadrupole magnets. Additionally, corrector magnets, which are small dipols, arranged vertically as well as horizontally, are used to adjust the position of the beam. The beams inside the storage ring can be polarized or unpolarized and their momentum ranges from 0.3 GeV/c to 3.7 GeV/c. The cooling of the beams in phase-space is done by way of electron cooling at injection energy and stochastic cooling at high energies, which start at a momentum of the particle beam of at least 1.5 GeV/c [5]. The electron cooling is based on merging the accelerated particle beam with an electron beam that consists of preferably mono-energetic electrons which match the average speed of the particle beam. In the coordinate system in which the average speed of the accelerated particle beam is zero, this mingling of the two beams equates a system of two gases, namely the electron gas and either the proton or the deuteron gas. In this system, the temperature of the particles only increases with the masses. Therefore, the proton or deuteron gas is cooled by the electrons [6].

The stochastic cooling is done by reading a particle's motion with a so-called "pickup" and compensating the dissipation force of a particle with a "kicker" that moves the particle back to the middle of the beam. It is referred to as stochastic cooling as different particles are affected by this process every time and only on average the entire beam will be cooled down.[7].

WASA, short for Wide Angle Shower Apparatus, originally comprised a pellet-target system as well as two detector parts for measurements. At this point, only the forward part from the original WASA detector will be used. The forward part is designed to measure charged target-recoil particles as well as scattered projectiles whereas the central part was used to measure meson decay products [9]. The trigger will be based upon the layout of the forward detector and will use its signals to generate an output if certain trigger conditions are met.

Major studies conducted with the WASA detector are already completed and the detector shall now be used to search for electric dipole moments. For this new task, it is necessary to study the trajectory of scattered particles. After the particles scattered at the target, the spin of the particles causes a movement into different directions. Taking this polarization into consideration, the cross section for the scattered particles at the WASA detector can be written as

$$\sigma(\phi, \theta) = \sigma_0(\theta) \cdot (1 + P \cdot A(\phi, \theta)), \tag{2.14}$$

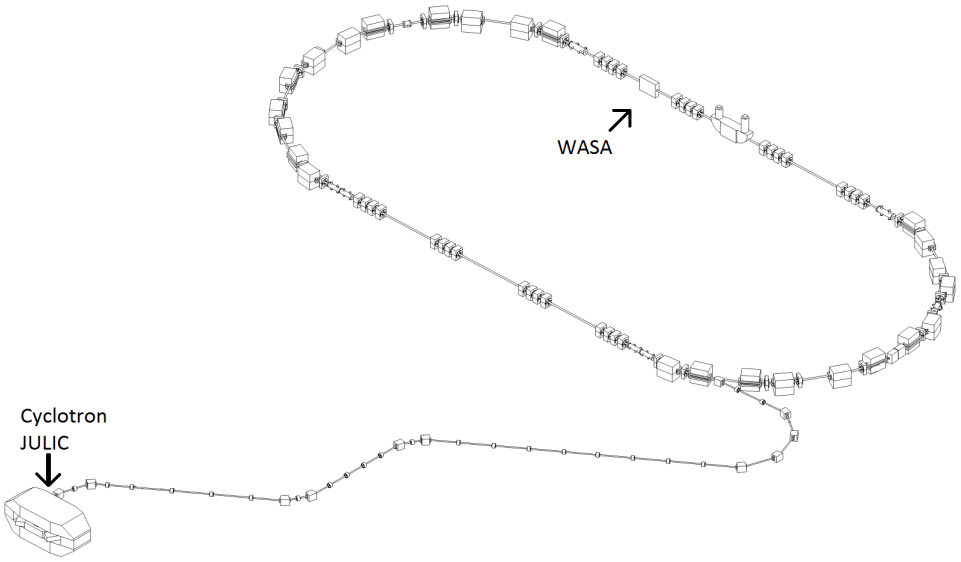


Figure 2.3.: Floorplan of the COSY complex, showing the Cyclotron JULIC, where the poliarized and unpolarized beams originate, and the WASA detector. [8, p. 4]

where σ_0 is the unpolarized cross section, P is the polarization and A is the analyzing power, which depends on the used target, the energy of the beam and the accelerated particles, in this case either protons or deuterons. As $A \propto \cos(\phi)$ for particles with spin $\frac{1}{2}$, where only the vector polarization has to be considered, 2.14 can be simplified to

$$\sigma(\phi, \theta) = \sigma_0(\theta) \cdot (1 + P \cdot A(\theta) \cdot \cos(\phi)). \tag{2.15}$$

When it comes to deuterons, the tensor polarization also has to be considered. As shown in fig. 2.4, dividing the detector into four quarters allows to differentiate between four major directions the particle can scatter towards. Using this division, the ratio between, for instance, the left and the right quarter is given by

$$\frac{n_R - n_L}{n_R + n_L} \propto \frac{\sigma_0(1 + P \cdot A_\theta) - \sigma_0(1 - P \cdot A_\theta)}{\sigma_0(1 + P \cdot A_\theta) + \sigma_0(1 - P \cdot A_\theta)} = P^\uparrow \cdot A_\theta(\phi = 0, \pi). \tag{2.16}$$

Thus, it is reasonable to base the trigger on this quartering of the detector and categorize events according to the quarter they occurred in. By measuring the polarizations and different analyzing powers, a polarimeter database will be built that can be used to produce Monte Carlo simulations of detector responses and estimate systematic error effects in any EDM ring polarimeter [11].

The detector comprises multiple scintillators which are used to detect particles. A scintillator consists of a luminescent material that will emit photons when struck by a particle. The amount of the emitted photons depends on the energy of the absorbed particle. By

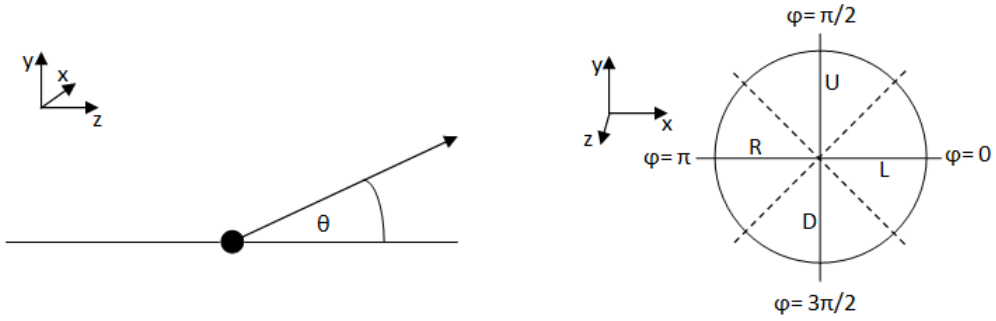


Figure 2.4.: A rough illustration of the scattering and the division into four quarters later used to design the trigger system. The detector is quartered into **L**eft, **R**ight, **U**p and **D**own. The z-axis is pointing towards the beam direction.

using a photomultiplier tube, the photons can be absorbed at the photocathode and, as part of the photoelectric effect, electrons will be emitted that are multiplied inside the tube. This allows to measure an electric signal at the anode [1]. Fig. 2.5 shows a sketch of a scintillator and a photomultiplier. Before the signals from the detector are evaluated by the FPGA trigger, a discriminator is used to digitize the signals. The discriminator

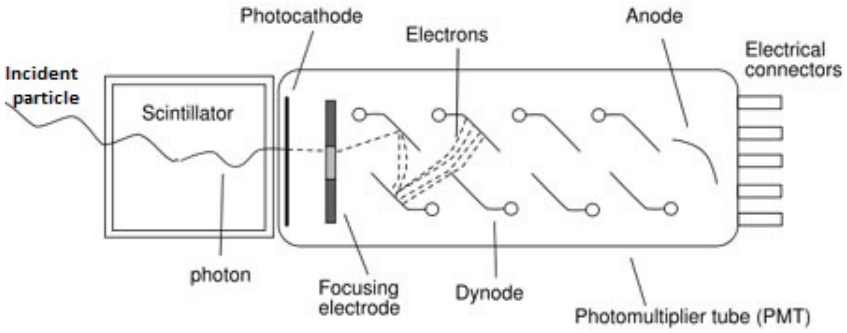


Figure 2.5.: The basic layout of a scintillator and a photomultiplier. The photomultiplier includes multiple dynodes that are used to multiply the electrons as well as a focusing electrode that is used to direct the electons onto the first dynode. [10]

will generate an electric pulse if the signal from the photomultiplier exceeds a predetermined amplitude. The width of the electric pulse can be set by the user. In this way, the discriminator will generate a normalized pulse whenever a particle with a certain energy was detected by one of the scintillators. Thus, the logical connectives of the trigger will be based on the evaluation of truth values in the form of *0* for no particle detected and *1* for particle detection. The width of the digitized signals should at least match the periodicity of the clock of the FPGA board to ensure that all signals will be processed correctly. An overview of the forward detector is given in fig. 2.6

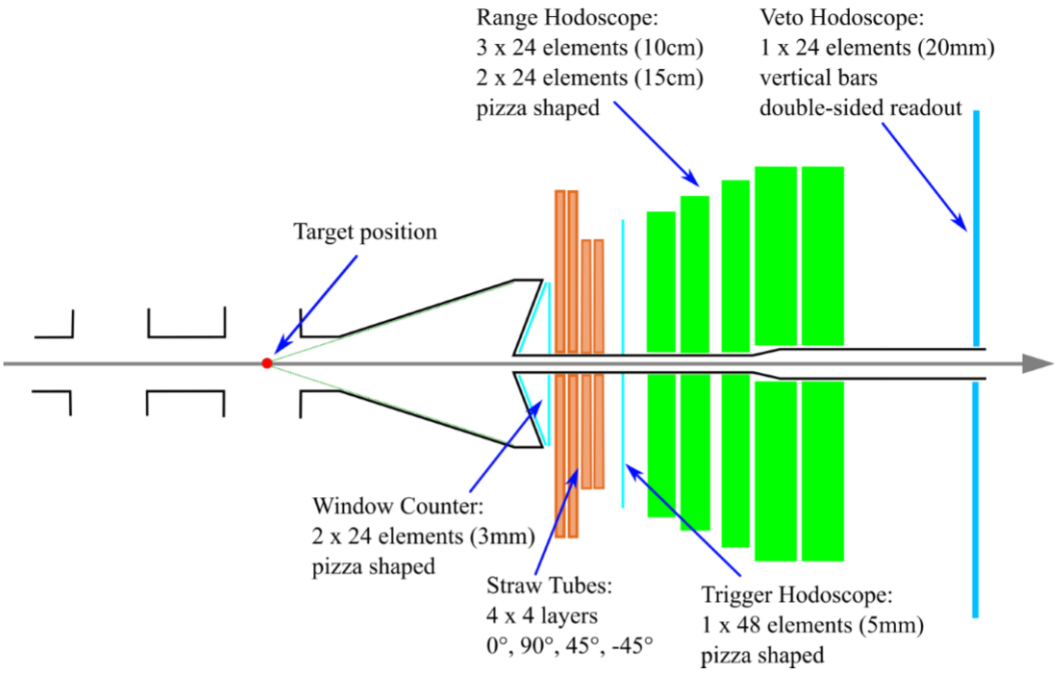


Figure 2.6.: The forward detector of the WASA with its different layers as well as the target crosses at the left-hand side. [11, p. 3]

2.3. Layout of the forward detector

The forward detector consists of multiple radially symmetrically arranged layers of plastic scintillators in combination with four layers of straw tubes. In order to generate a trigger signal, up to 216 scintillators of the forward detector are evaluated. Those scintillators are located in three different areas of the detector. The forward detector covers a scattering angle between 3° and 17° [9].

2.3.1. Window counter

The first two layers shape the window counter. This is the closest part of the detector to the target position and comprises 24 scintillators per layer. The two layers of scintillators are rotated in an angle of $\Delta\Phi_{FWC} = 7.5°$ with respect to each other which will later become important when the FPGA evaluates whether all necessary trigger conditions are met. Each layer of the window counter has a thickness of $d_{FWC} = 3\,\text{mm}$. A rough sketch of the window counter can be found in fig. 2.7. The layer closer to the target position is henceforth referred to as FWC1 while the rear layer is referred to as FWC2. The rotation between the two layers is also depicted in this sketch.

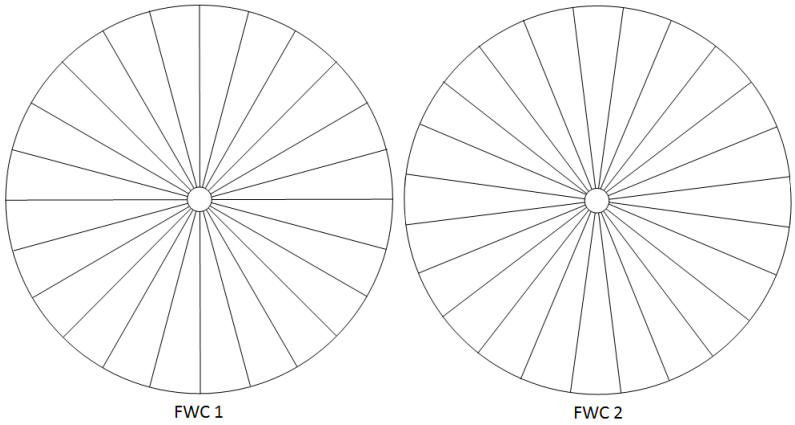


Figure 2.7.: The two window counters that comprise 24 scintillators each and are rotated 7.5° relatively to each other.

2.3.2. Trigger hodoscope

The window counter is followed by four layers of straw tubes that will, however, not be used for the trigger. The straw tubes are used for a precise determination of the scattering angle. The next layer that will count towards the trigger is the so-called trigger hodoscope. The trigger hodoscope consists of 48 scintillators and is $d_{FTH} = 5\,\text{mm}$ thick. With 48 elements[3], the trigger hodoscope has the highest granularity of all layers evaluated by the FPGA. Similar to the window counter, the scintillators of the trigger hodoscope, henceforth abbreviated as FTH, are also arranged in a rotationally symmetric shape. The size of each element of the FTH equals the size of the intersection of two elements of the FWC1 and FWC2. That means that each element covers an angle of $\Phi = 7.5°$.

2.3.3. Range hodoscope

The next part of the detector is the range hodoscope. It comprises five layers of 24 scintillators each. There is no rotation between the scintillators of the different layers, wherefore the layout of each layer looks alike. The shape matches the shape of the FWC1 as shown in fig. 2.9. The range hodoscope will primarily be used to determine the penetration depth of a particle, as the position of a particle can more precisely be ascertained by the FTH or overlapping elements of the FWC1 and FWC2.

While the first three layers of the range hodoscope are each $d_{RH,front} = 10\,\text{cm}$ thick, the two layers at the back-end have a thickness of $d_{RH,back} = 15\,\text{cm}$ each. Aside from its use as part of the trigger, the range hodoscope is used to identify particles and determine their energy by evaluating the pattern of deposited energy in the different detector planes.

[3]The term element refers to any one of the scintillators on the layer. The rotational symmetry of each layer prompts to call the layer "pizza shaped".

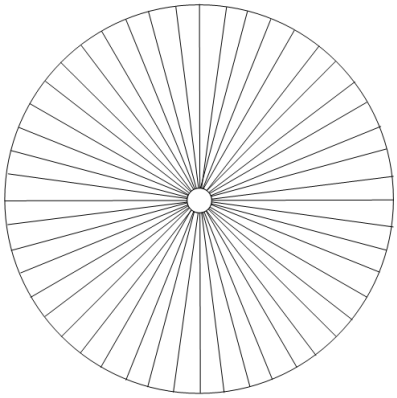


Figure 2.8.: A sketch of the trigger hodoscope. The trigger hodoscope comprises twice as many scintillators as each of the window counters.

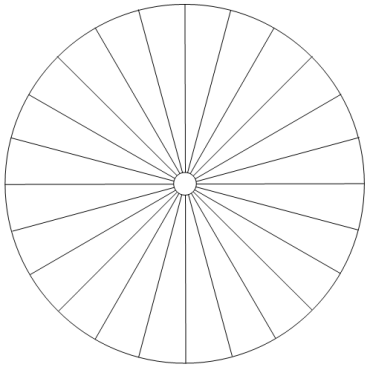


Figure 2.9.: The five layers of the range hodoscope all match this layout, which is equivalent to the first window counter.

2.3.4. Additional parts

The last part of the detector is the veto hodoscope. This hodoscope consists of 24 vertical bars. The signals from the veto hodoscope will, as is the case for the straw tubes, not be considered for the trigger signal generation. The intended use of the veto hodoscope is to reject or select particles punching through the range hodoscope.

The layout as described above constitutes the current draft for the forward detector. The FPGA trigger will entirely be based on the configuration described above, wherefore the original configuration of the WASA detector will not be discussed at this point.

3. Technical Components

A major requirement for the trigger is the ability to process and evaluate all necessary signals from the forward detector and generate an output signal if certain scintillators show a particle passage. These signals are evaluated by passing logical connectives and, if predefined conditions, referred to as trigger conditions, are met, the trigger will in turn send an output signal that can be used to start the process that will store the data on disk. A device that fulfills these requirements particularly well is a so-called FPGA, short for Field-Programmable Gate Array, a semi-conductor device that not only allows to be reprogrammed after manufacturing to the desired application or functionality requirements, but can also be partially reconfigured while unaffected parts of the FPGA keep running. In order to be able to evaluate the signals from the detector, the FPGA brings along "a matrix of configurable logic blocks (CLBs) connected via programmable interconnects" [12].

3.1. Technical specifications and functions of the FPGA

The device used to build the trigger is a V1495 by CAEN S.p.A., which is a general purpose VME board that includes a user customisable FPGA Unit[1].

3.1.1. "User" and "Bridge" FPGA

The "User" FPGA manages the front I/O channels and can be run with a custom firmware. The FIFO depth is adjusted via the VME interface, which means without having to remove the board from the experimental setup or turning off the crate. The centerpiece of the FPGA trigger, the logical connectives to evaluate the signals and compare them to the trigger conditions, will be retained on the board, but different parameters can be freely adjusted at any time. Aside from the "User" FPGA, another FPGA is also included on the board. The so-called "Bridge" FPGA[2] is used for the VME interface and communicates with the "User" FPGA via a proprietary local bus. The "User" FPGA is set up as a slave of the "Bridge" FPGA and, therefore, the "Bridge" FPGA manages the programming via VME of the "USER" FPGA. An overview of the entire board including the "Bridge" and "User" FPGA is given in fig. 3.2.

3.1.2. Input channels and division of the detector

The board has 64 permanent input channels, that handle LVDS, ECL and PECL[3] signals, and 32 output channels that send LVDS signals. The input channels are split into two

[1] Cyclone EP1C20F400C6N
[2] Cyclone EP1C6Q2408N
[3] Standards for logic signals: LVDS: Low Voltage Differential Signal, ECL: Emitter Coupled Logic, PECL: Positive Emitter Coupled Logic

32 pin ports, referred to as port A and port B. The output is referred to as port C. By using up to three additional mezzanine boards, another 96 channels are added. These 96 channels are evenly split into three ports of 32 pins each, referred to as ports D, E and F. These three ports can be used as input as well as output ports. For this design, the ports D and E will be used as ECL input ports and port F will be used as an ECL output port. The mezzanine boards are available for ECL, PECL, LVDS, NIM as well as TTL signals[4]. Another two channels can be used for NIM and TTL signals only, referred to as port G. Therefore, the board provides a total of 194 channels [13]. This means that a single board can not cover the entire forward detector, which requires in total 216 input channels. To make sure that all signals of the scintillators can be evaluated, it is necessary to use two boards. Those two boards will run independently of each other. The division of the detector will be done by splitting each of the eight evaluated layers into two parts along the same axis. This is demonstrated in fig. 3.1. As shown in the graphic, the detector is divided into two equally large halves. Each layer of the range hodoscope is divided exactly like the FWC 1.

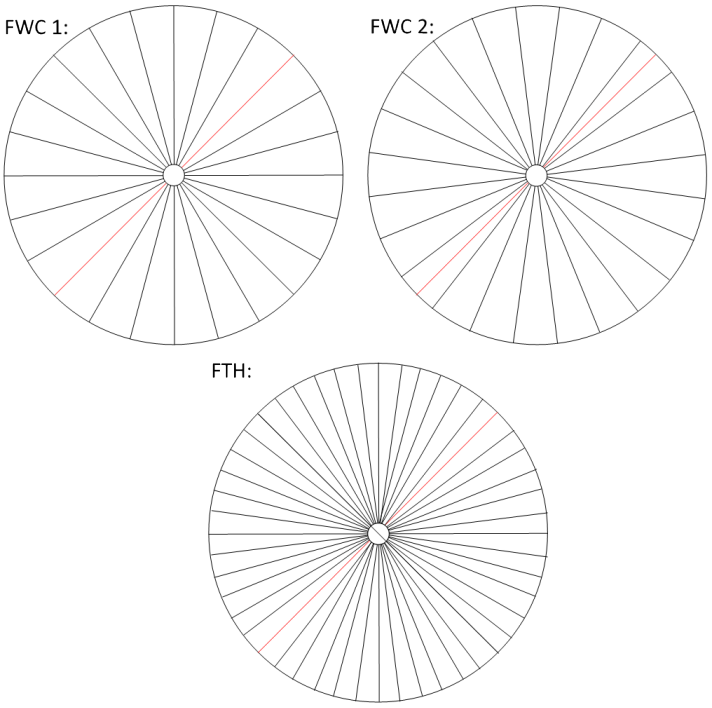


Figure 3.1.: The window counter and the trigger hodoscope divided into two halves by a red line, representing the division of the detector for the two FPGAs.

By dividing each layer in half and with respect to the amount of scintillators per layer as described in section 2.3, 108 input channels have to be covered per FPGA. Another 15 input channels come from the two closest scintillators at the boundaries that are located in the otherwise not covered half of each FPGA. This means that at both borders of

[4]TTL: Transistor-Transistor Logic, NIM: Nuclear Instrumentation Module

each half one scintillator of the other half will be evaluated by both FPGAs to ensure that the detector will entirely be covered even if an overlap of, for instance, one element of the FWC1 from the upper-left half and one boundary element[5] of the FWC2 from the lower-right half occurs. All in all, the amount of input channels needed to evaluate all signals can be reduced to 123 per FPGA, as the calculation above shows, when two FPGAs are used.

3.1.3. Clock

The build-in clock of the board runs at a frequency of $f_{FPGA} = 40\,\text{MHz}$, which corresponds to a periodicity of $t = 25\,\text{ns}$. This means that the digitized signals from the discriminator need to have a width of at least $t = 25\,\text{ns}$ to ensure that a signal is not send during the insensitive time of the FPGA and vanishes before the next clock pulse starts. This, however, can easily be achieved as the signal width coming from the discriminator can individually be adjusted by the user. Choosing a proper width guarantees that all signals will be registered and evaluated. When it comes to the expected event rate, the frequency of 40 MHz is also by far high enough to ensure that all events will be evaluated as even the highest event rates at COSY with the best possible luminosity never exceeded $2 \cdot 10^6$ events per second. For the planned measurements the event rate is expected to be smaller than 10^6 events per second.

3.2. Software

To write and later on simulate code for the FPGA certain software tools are necessary. In this chapter, the used language, VHDL, as well as the used programs to build and simulate the design are described.

3.2.1. VHDL

The coding language used to program the FPGA is called VHDL, short for *VHSIC*[6] *Hardware Description Language*, a hardware description language that is nowadays commonly used in Europe. VHDL gives the user the opportunity to describe how a design shall be structured into sub-designs and how these sub-designs shall be connected. The function of a design can be specified by using a language that resembles common programming languages, particularly Ada. An upside of VHDL is the opportunity to run simulations of the program before it is integrated into the hardware. This allows the user to test and adjust the program without having to build a hardware prototype. Special simulation tools allow the user to construct so-called testbenches, with which input signals can be emulated in order to check the correct processing of signals by the design [14].

The design is separated into so-called *entities*. They contain a list of signals and constants that will be used within the particular entity. Each entity is able to process a set of signals, which are for their part divided between inputs and outputs. It is also possible to declare inouts and buffers. Those signals can not only be read but also written to

[5] "boundary element" refers to the two elements that overlap the separating line, as shown in 3.1

[6] Very High Speed Integrated Circuits, a U.S. government program from the 1980s.

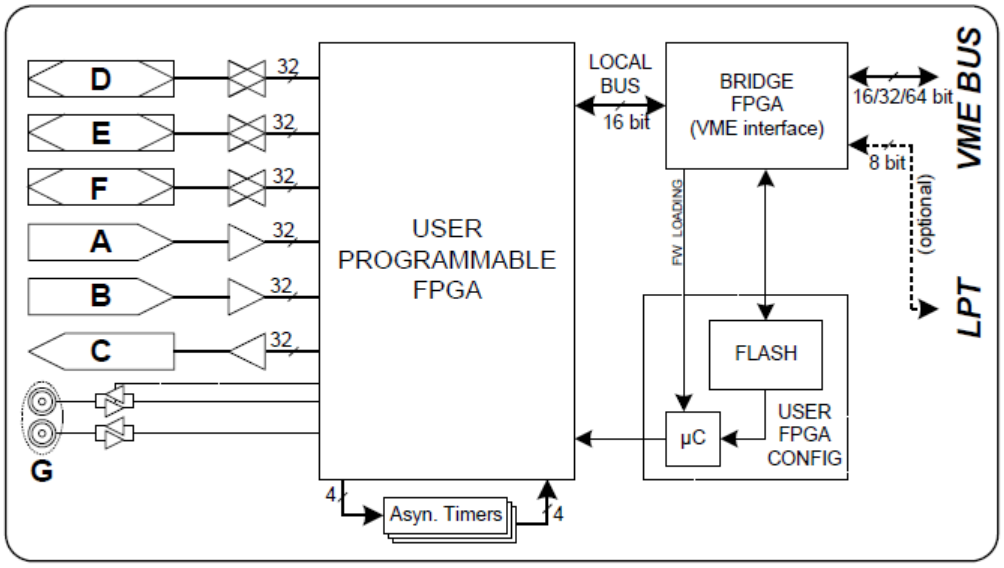


Figure 3.2.: A block diagram showing the V1495 General Purpose VME Board including the input ports A and B, the output port C as well as the three inout ports D, E and F, which can be added by using mezzanine boards. The two additional ports for NIM and TTL signals, referred to as port G, are located at the lower left side of the board. The "USER" FPGA is connected to the ports and via a local bus to the "Bridge" FPGA, located at the upper right side. A VME bus is used for the communication with the FPGA. [13, p. 8]

at the same time. All of those signals are categorized as *ports*. They are declared in a dedicated segment at the beginning of an entity. Similarly, constants are declared in a segment called *generic*. It is possible to call and execute an entity from another entity. In this case, it is necessary to tell the design which ports from the mother entity shall constitute which ports of the called entity. This step is called *port mapping*.

Within an entity one has to instantiate the architecture. Inside the architecture of an entity further signals, variables and constants can be declared, which, however, need to be sent to a designated output port to have them available at an output of the FPGA. Additionally, the declarations made at this point are only available inside the architecture. The main part of the architecture contains concurrent statements that will instruct how the signals will be processed. VHDL provides a large range of Boolean operators to construct logical connectives, which can process the input signals and generate output signals. The logic synthesis as part of the compilation of the design is also able to detect logical connectives between signals, even if they are not expressed by conventional operators, and tries to build the optimal hardware implementation, using, for instance, as few transistors as possible. In order to keep the architecture of the entity neat, so-called *procedures* can be declared, which are similar to functions in programming languages. Within such procedures, another set of declarations has to be incorporated. These declarations have to cover all variables that will be used within the procedure. Similar to

the architecture, the procedure includes a main part in which sequential statements are made, telling the hardware how to process the signals. Procedures can be called from the architecture by stating the name and defining all variables that shall be passed to the procedure, similar to the way another entity is called from within an entity.

3.2.2. Quartus II 11.0

The software used to design the trigger is *Quartus II 11.0 Web Edition*, a design software for programmable logic devices. The producer *Altera Corporation* used to be a company specialized in the production of programmable logic devices. In 2015 *Altera* was acquired by *Intel*. The software *Quartus II* not only allows to write VHDL code and compile it, but also brings along useful tools for the production of a design. An integrated RTL[7] viewer provides the option to generate a model of the digital circuit, which shows a schematic realization of the logical connectives by the use of hardware registers. Similar tools like the *Chip Planer* or the *TimeQuest Timing Analyzer* allow the user to check how the in- and outputs of the design will be arranged on the device and how it will perform in regards to processing time, delays and suchlike. The compiler comes along with an analyzer, synthesizer, fitter, assembler as well as multiple other tools to evaluate the design and its performance. *Quartus II* also features links to simulation tools which can be used to test the design. Hence, the design can easily be loaded into a simulation tool, like Modelsim-Altera.

3.2.3. Modelsim-Altera

To make sure that the design works as planned and all signals and logical connectives are correctly processed, one can simulate the design in special simulation tools, like Modelsim-Altera. Modelsim-Altera is as the name suggests also developed by Altera and can simulate HDL designs written in Quartus II and other programs. By writing so-called *testbenches* it is possible to simulate incoming signals and check whether the signals created within the architecture match the expectation. All signals, variables and constants declared in the design are listed in Modelsim and can be set to specific values. It is possible to create waves and periodic signals to simulate a change of the signals. A screenshot of the software is shown in fig. 3.3

In regards to the FPGA trigger, the ports containing the incoming signals from the detector would be set to display a particle strike and would then be used to check whether the logical connectives are correctly processed and, if all necessary conditions are met, whether the output signal shows the desired values. This makes it possible to test different configurations of signals and ensure that all configurations that should trigger a signal are indeed properly processed. The generated output can be examined in regards to the correct processing of signals. Furthermore, this approach makes it possible to check whether signals which do not meet the conditions will not accidentally trigger an output signal. It is also possible to set a periodicity, the width of the signals and integrate delays between the different signals.

[7] Register-transfer level

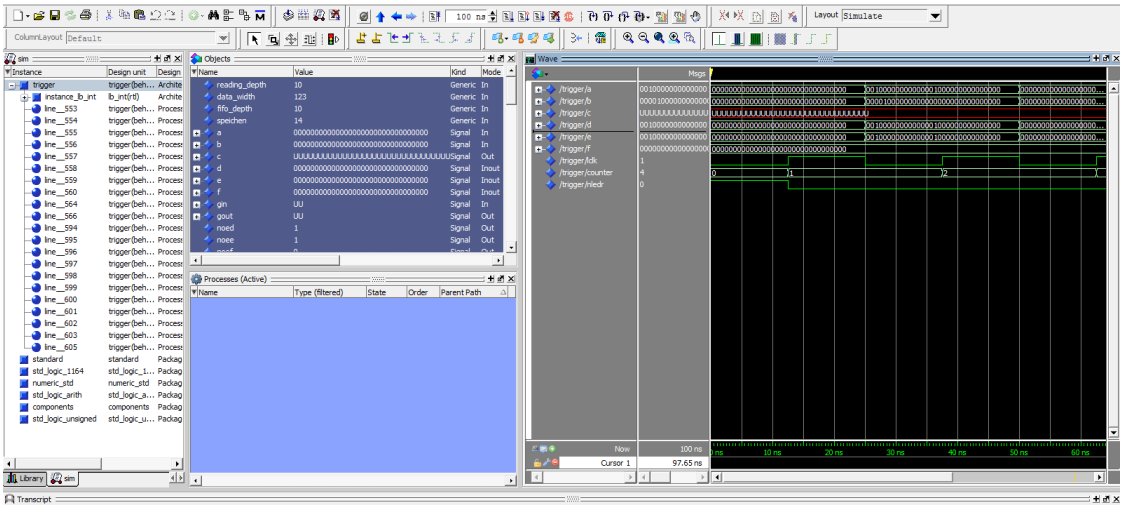


Figure 3.3.: The screenshot shows the main window of Modelsim. It includes signals similar to those that will later be used to simulate the trigger inside the frame at the right-hand side. The middle frame titled "Objects" includes a list of all signals, variables and constants used in the design.

To make sure that every single signal does not have to be entered individually, the user can import entire files, which contain values for all signals included in a design. To write such a file for the trigger at hand, it is reasonable to write a tool that can generate signals based on the trigger design. This makes it possible for the user to display the signals, as the bits used in Modelsim are not in the least descriptive of which layers of the detectors are struck. Thus, a simple tool is implemented in the scripting language *Autohotkey*. It includes a graphical representation of the WASA forward detector. Different configurations can be set up and will graphically be displayed in the tool. Once the desired configuration of signals is set up, the configuration is saved to a file and then imported into Modelsim, where the simulation can be run. Aside from basic signals, the tool also provides the opportunity to generate delays between the signals arriving at different layers of the detector and adjust which entries of the FIFO shall be read. A screenshot of the tool can be found in fig. 3.4.

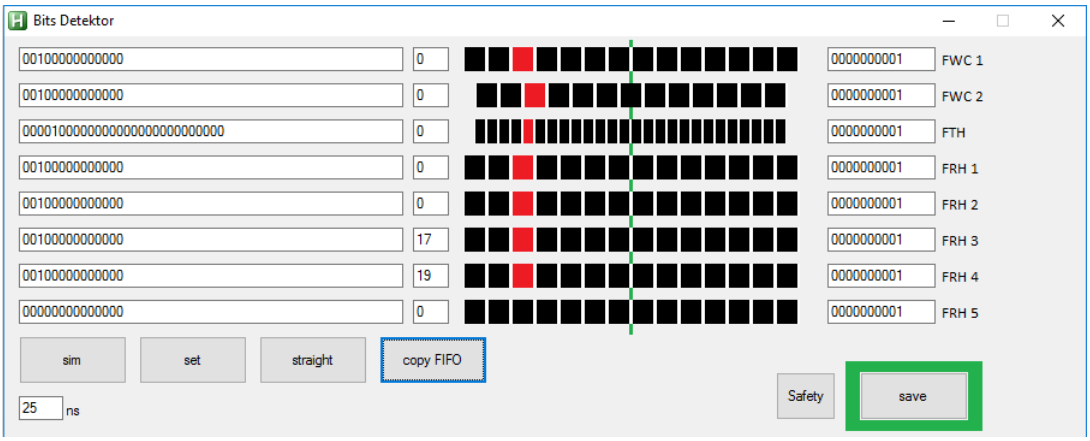


Figure 3.4.: The tool written in *Autohotkey* which includes, from left to right, the bits sent to Modelsim representing incoming signals from the detector, delays between the layers, a graphical representation of the detector and a list that allows the user to set which entries of the FIFO shall be evaluated.

4. Functions of the FPGA Trigger

The trigger is supposed to send an output signal containing information about the events that are registered in the detector, like the position of the particle and the penetration depth, if certain trigger conditions are met. The signals coming from the detector are stored in a FIFO[1] before they are evaluated. The signals will be divided into four quarters, representing the left, right, upper and lower part of the detector. As the signals coming from the detector are already separated into the upper left and lower right part, due to the use of two FPGAs, it is only necessary to split the remaining 123 input signals of each FPGA into two halves, as illustrated in fig. 4.1. This division of the detector into four quarters allows to examine the polarization for online evaluation of the asymmetries and the analyzing powers, as described in section 2.2.

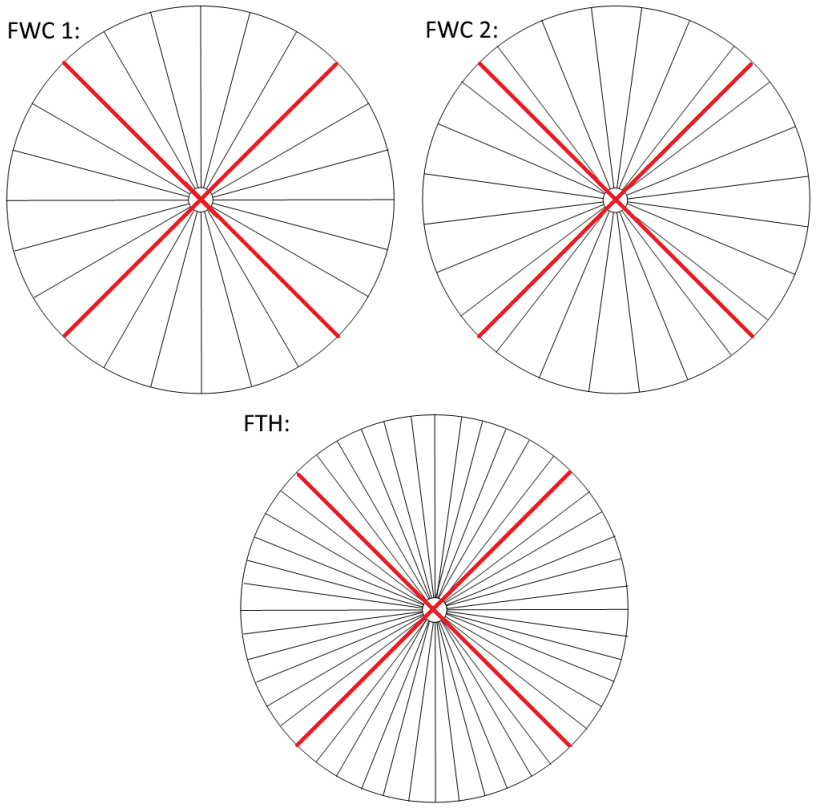


Figure 4.1.: The quartering of the detector. The division always takes place along the same axes and also applies for the five range hodoscopes.

[1] First in first out, a type of data buffer described in section. 4.1

The signals from the detector are send to the ports A, B, D and E of the VME board, all of which are ECL inputs. The ports D and E are made available by adding mezzanine boards to the VME board. The output is send to port F, which is also added to the VME board by using a mezzanine board. Each port consists of 32 pins, wherefore the 123 signals from the detector have to be apportioned between the 4 input ports. As can be seen in the following excerpt, the signals from the two window counters are send to port A, the signals from the trigger hodoscope are send to port B, the signals from the first four range hodoscopes are send to ports D and E and the fifth range hodoscope is split between the remaining pins of the four ports. The term *Speichen*[2] represents the amount of scintillators forming the first window counter.

```
APort        <= not A(0 to Speichen-1);
BPort         <= not A(Speichen to 2*Speichen-2);
CPort         <= not B(0 to 2*(Speichen-1)-1);
DPort         <= not D(0 to Speichen-1);
EPort         <= not D(Speichen to 2*Speichen-1);
FPort         <= not E(0 to Speichen-1);
GPort         <= not E(Speichen to 2*Speichen-1);
HPort(0 to 4)         <= not A(2*Speichen-1 to 31);
HPort (5 to 10)  <= not B(2*(Speichen-1) to 31);
HPort (11 to 13) <= not D(2*Speichen to 2*Speichen+2);
```

The eight layers of the detector are referred to as A to H, meaning that signals starting with an A are associated with the first window counter and signals starting with a H are associated with the fifth range hodoscope. Since the incoming signals are inverted by the FPGA, meaning that a pulse is registered as a logical 0 whereas no pulse is considered to be a logical 1, the logical complement is built to use a pulse as a logical 1. These values are then stored in the FIFO before the trigger conditions are checked and, if all necessary conditions are met, the output signal is generated.

4.1. FIFO

To make sure that delays between the different layers will be covered by the trigger, a FIFO that stores the signals from all eight layers of the detector per clock pulse is implemented. Due to the versatility of the FPGA, thousands of signals can potentially be stored in the FIFO. The current firmware is designed to store signals from ten clock pulses in the FIFO, which amounts to an interval of $T_{FIFO} = 250\,\text{ns}$, meaning that over a span of 250 ns signals will be kept available for evaluation. Per clock pulse the 123 signals registered at the input ports of the board are added to the end of a vector called *Data* that can carry 1230 logical values. These newly added values replace the oldest 123 values, meaning the values from the clock pulse 250 ns ago:

```
Data <= Data(DATA_WIDTH to DATA_WIDTH*FIFO_DEPTH-1)&APort&BPort&CPort&DPort
&EPort&FPort&GPort&HPort; -- new signals are added to FIFO
```

[2] *Speichen* is a constant used in the design to adjust the amount of scintillators the FWC1 comprises. Every layer is based on this amount of scintillators, meaning that increasing *Speichen* by one would also add a scintillator to the FWC2 and so on.

This feature of the trigger does not only allow the user to compensate for the time it takes particles to travel through the detector, but also compensate for cable delays in case the experimental setup does not include means to compensate such delays before the signals arrive at the FPGA. Furthermore, the FIFO can be used to implement delays between the registration of signals at the input ports and the generation of the output signal.

The possibility to communicate with the FPGA using registers allows the user to choose which entries of the FIFO shall be evaluated per clock pulse. A ten bit logical vector[3] is used to tell the FPGA which entries are to be used in the evaluation process. Every one of the ten bits represents an entry in the FIFO, beginning with the oldest entry associated with the first bit and ending with the youngest entry associated with the tenth bit. For instance, the following vector would mean that the FIFO shall use the first two youngest as well as the fourth youngest entries to determine whether the trigger hodoscope detected a particle that will count towards the trigger conditions[4]

```
Cread <= "0000001011";.
```

Using this feature the selection of signals is completely adjustable for all eight layers of the detector. The entries used for the evaluation between the different layers can vary at will. If at least one of the entries selected for evaluation contains a logical 1, the trigger will count it as an registered event.

The table 4.1 shows which register addresses are used to adjust which entries of the FIFO shall be used to generate a trigger signal. The listed register addresses are referred to the base address of the board, which means that the they have to be added to the board base address [13].

Detector layer	Register
FWC1	1010
FWC2	1014
FTH	1018
FRH1	101C
FRH2	1020
FRH3	1024
FRH4	1028
FRH5	102C

Table 4.1.: The table shows which register addresses are linked to which layers of the detector when it comes to configuring the FIFO.

[3] meaning that it can either contain a logical 1 or 0, referred to in vhdl as std_logic or std_logic_vector (standard logic)

[4] This is just an example used to explain the adjustment of the FIFO and does not represent a suitable setup for the experiments conducted in COSY.

4.2. Trigger conditions

The most basic condition that has to be met is an overlap of the scintillators of the first two window counters a particle penetrates. As the window counters consist of 24 scintillators each and the two layers are shifted $\Delta\Phi_{FWC} = 7.5°$, a particle that registers on overlapping elements of both layers hits an intersection of $\Phi_{int} = 7.5°$. After the particle struck a scintillator of the FWC1 (1), it can either hit the scintillator to the left (2a) or the right (2b), as shown in fig. 4.2.

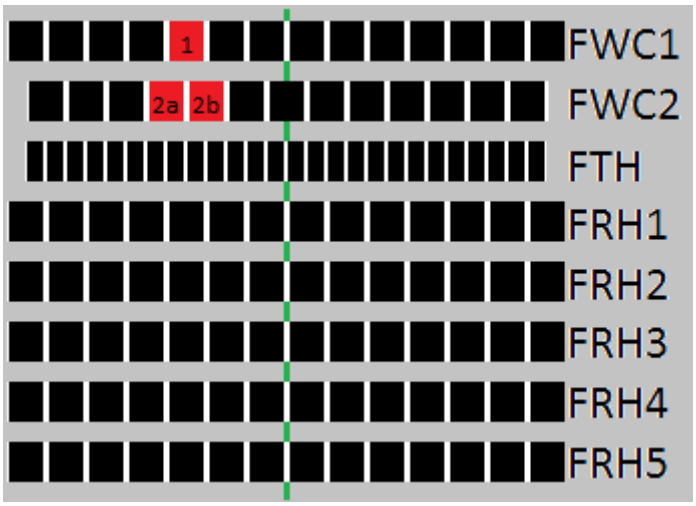


Figure 4.2.: The particle hits the FWC1 and can either hit the scintillator to the left (2a) or to the right (2b), building an intersection of $\Phi_{int} = 7.5°$.

As soon as this condition is met, an output signal is generated. The calculated particle position, which is part of the output signal, is based on the intersection Φ_{int}. From this point on, it is possible to tell which quarter of the detector has been hit. If the particle travels even further and reaches the FTH, the output signal will be based upon the position determined by the FTH, as this layer has the highest granularity. In case of a straight penetration, it is non-relevant whether the position is based on the overlap of the two window counters or the struck element of the FTH, as the two-times higher granularity of the FTH ensures that every single one of the possible 48 intersections between the two window counters has a corresponding element of the FTH. However, a particle drift within one element to the left or two the right of the intersection of the window counters shall also count as a proper event and therefore cause an output signal. This feature allows a particle drift of up to $\Phi_{drift} = 15°$ to also be evaluated and cause a trigger signal. If such an event occurs, the position determined by the FTH will determine which quarter of the detector will be associated with the penetration. This is specifically important at the boundaries of the four quarters, as the element of the FTH registering a particle could be located in another quarter as the penetrated intersection of the window counters. Larger deviations than those described above will not count towards the trigger conditions and therefore not cause a trigger output. The last five layers, the range hodoscopes, will not influence the determined particle position as only

straight penetrations through the range hodoscopes will be counted towards the penetration depth. Struck scintillators from the range hodoscopes will only be considered if they are located behind the struck layer of the trigger hodoscope or the intersection of the window counters. A straight penetration as well as a two penetrations with particle drifts are displayed in the Autohotkey tool in fig. 4.3.

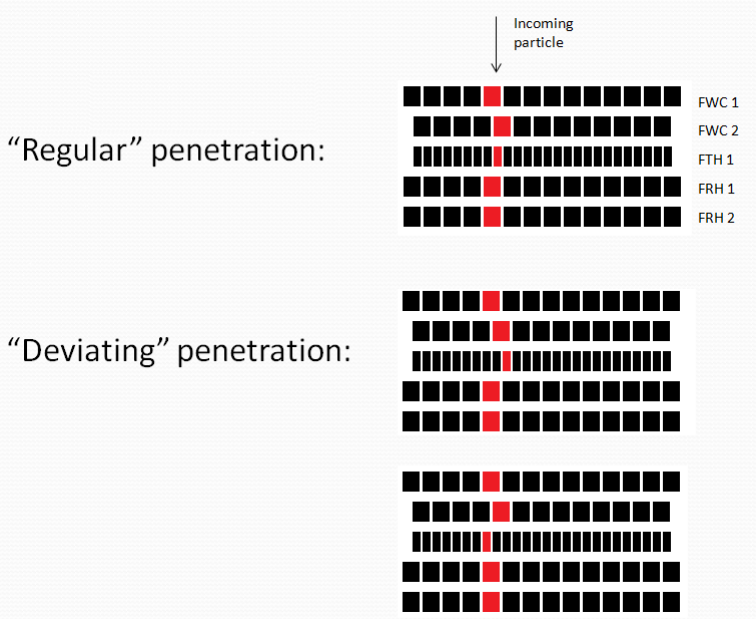


Figure 4.3.: Three particle penetrations are shown that will all cause a trigger signal. As the trigger hodoscope has been struck within $\Phi_{drift} = 15°$ of the intersection of the window counters, the position will always be determined by the struck element of the trigger hodoscope. The last three layers of the range hodoscope are not displayed in order to shorten the picture.

4.3. Output signals

As soon as overlapping elements of the two window counters register a particle penetration, an output signal is generated. The output signal contains 18 bits and is sent to port F of the VME board. The first two bits represent the *Quarter*[5], the next eight bits show how deep the particle traveled into the detector (*Depth*) and the last eight bit show the *Maximal Penetration Depth*. The *Quarter* is determined as described in 4.2. Each of the two FPGAs shows which half of the respective FPGA has been struck. A logical 1 as the first bit means that the left half was penetrated while a logical 1 as the second bit means that the right half was penetrated.[6] Of course, it is also possible that both halves are

[5] which of the four quarters, as shown in fig. 4.1, was hit
[6] In this case left and right half refers to the separation as shown in fig. 3.4.

struck at the same time. The next eight bits, the *Depth*, show how far the penetration reached into the detector. If a trigger signal is sent, at least the first two of these eight bits have to be set to 1 as the trigger will only sent a signal if the two window counters detect a particle. As soon as the trigger hodoscope is reached, the third bit will be set to 1 as well. The five range hodoscopes will be set to 1 according to the penetration depth. The code where the penetration depth is calculated is part of the main examination of the signals and can be found in the appendix. Once the *Depth* is examined, a procedure called *maximal depth* will calculate the last layer that was penetrated by the particle and set the bit representing this layer to 1 while the seven remaining bits are set to 0. This signal can be used in the further evaluation of the experiment, in case only the maximal depth is required instead of an eight bit vector showing all penetrated layers. The 18 signals are sent to the first 18 pins of the port F and can then be passed on to the further readout setup. A scheme of port F is shown in table 4.2.

Pin	Information	Segment
01	Quarter 1	Quarter
02	Quarter 2	
03	Hit FWC1	Depth
04	Hit FWC2	
05	Hit FTH	
06	Hit FRH1	
07	Hit FRH2	
08	Hit FRH3	
09	Hit FRH4	
10	Hit FRH5	
11	Max. depth is FWC1	Maximal penetration depth
12	Max. depth is FWC2	
13	Max. depth is FTH	
14	Max. depth is FRH1	
15	Max. depth is FRH2	
16	Max. depth is FRH3	
17	Max. depth is FRH4	
18	Max. depth is FRH5	

Table 4.2.: The first 18 pins of the output F. The remaining pins (19 to 32) of the output F are not used.

5. Simulations

Using a simulation tool like Modelsim-Altera allows to test the design at various stages to make sure that the code works as expected. Therefore, multiple simulations are run in the process of designing the trigger to check whether newly added features actually work. At this point, a couple of simulations run with the final version of the trigger design, before the firmware is transferred to the FPGA, are shown and explained in order to demonstrate the functionality of the design.

All simulations are run using signals from the *Autohotkey* tool as shown in fig. 3.4. For the different particle penetrations, delays and FIFO setups, a screenshot of the tool as well as the simulated signals in Modelsim-Altera are included. By testing the design this way, the functionality can be checked without having to connect actual cables for countless different configurations. To ensure that the transferred firmware nevertheless runs on the actual board, some configurations are also tested with real signals. Those test can be found in section 6.

The first simulation includes a straight penetration through the third element[1] of the two window counters and the five range hodoscopes. The trigger hodoscope is struck right behind the intersection of the two window counters. There are no delays and the FIFO is set to use only the latest signals. The result is shown in fig. 5.1. Larger versions of the screenshots from the simulations are included in the appendix B.

[1] The numbering be clockwise

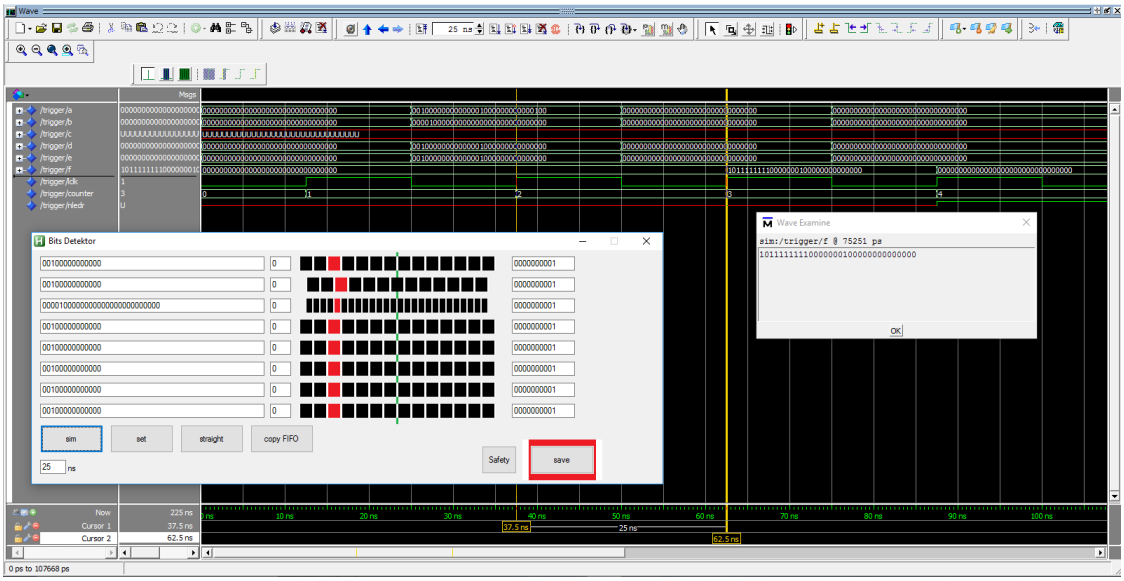


Figure 5.1.: A straight penetration without any delays, the box at the right-hand side shows the output F: *1011111111000000010000000000000*.

As can be seen in the screenshot, the output signal is sent 37.5 ns after the input signals are first sent and 25 ns after the input signals are stored in the FIFO at the next clock pulse. This means that the output not only shows the correct quarter, namely the left side of the detector half, as well as the correct penetration depth of eight layers, but also that the FIFO is set up correctly to read the latest entry for all eight layers.

A second simulation depicts a scenario in which the particle hits overlapping scintillators of the window counters right in the middle of the detector half, but the hit scintillator from the FTH deviates one element to the right from the intersection of the window counters. Furthermore, the first three range hodoscopes also show a penetration but an increasing delay is set up that requires an adjustment of the FIFO. To make sure that all six layers that show a penetration are evaluated at the same time, the FIFO is set up to put out the second youngest entries from the first three layers and the youngest entries from the first three range hodoscopes. As the position output is based on the location determined by the FTH, and the deviating element of the FTH is located in the right half, the output should display that the particle hit the right half of the detector half. The signals are shown in fig. 5.2 and the results can be found in fig. 5.3.

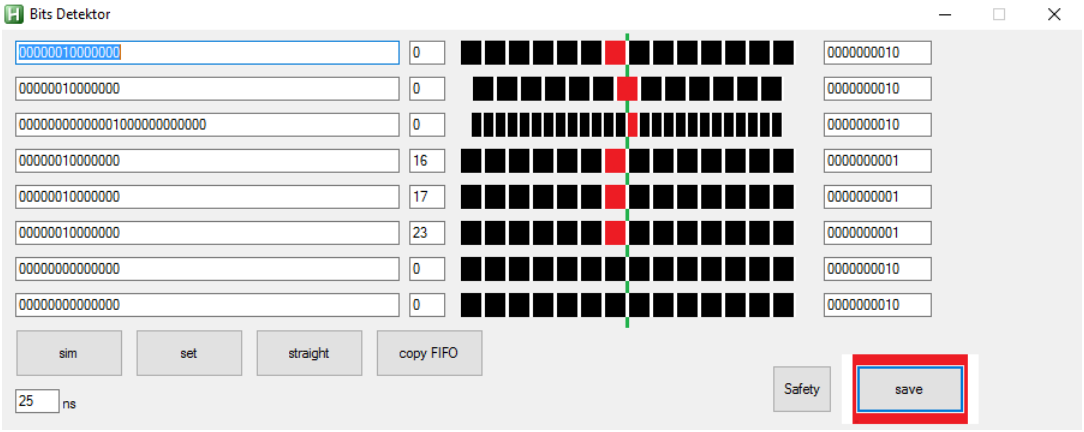


Figure 5.2.: Configuration for a penetration with multiple delays and a deviation in the FTH.

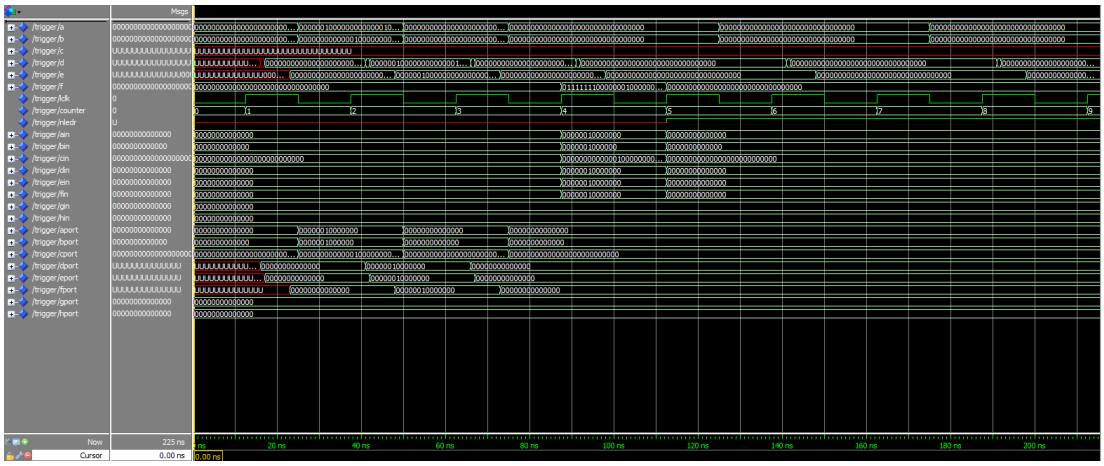


Figure 5.3.: Signals from a particle penetration with multiple delays and a deviation in the FTH. The output F: *011111110000000100* (first 18 bits) matches the expectations.

As can be seen in the screenshot, the particle is registered in the right half and struck the first six layers. This matches the configuration of input signals and shows that the FIFO correctly compensates the delay between the different layers.

In the next configuration the particle strikes straight through overlapping scintillators of the window counters, a scintillator of the FTH right behind this overlap and a scintillator of the FRH1. There is a delay of 225 ns between the penetration of the first two layers and the last two hit layers. The FIFO is set up to evaluate the two oldest entries for the first two layers and the two youngest entries for the last two layers. The particle hit the scintillator of the FWC2 that reaches into both halves of the detector but the struck scintillator of the FTH is located in the right side, wherefore the output of the position should display *01*. The configuration can be seen in fig. 5.4 and the signals from the

simulation are included in appendix B[2].

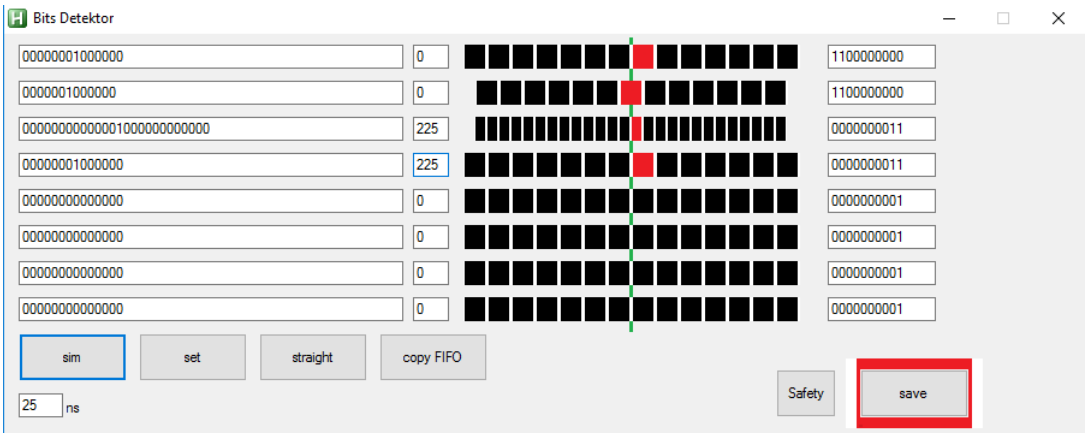


Figure 5.4.: Configuration for a penetration with a long delay. The output shows *011100000001000000* for the clock pulse starting at 262.5 ns and *011111000000010000* for the next clock pulse.

The first output signal shows a penetration of only the first two layers and is caused by the configuration of the FIFO, which is told to use the oldest entries of the first two layers after 200 ns and the youngest signals of the last two layers as they arrive as well as for one more clock pulse after they arrived. Thus, it is ensured that the penetration of all four layers will be evaluated but the second signal for the penetration of the first two layers is also sent to the output port for the above-mentioned reasons. Such a configuration can be used to ensure that a particle penetration that reaches the range hodoscope generates a trigger signal despite long delays between the registration of the penetration at different layers.

The last simulation shows a scenario in which two particles hit the detector half but the penetration depth varies between the two particles. There are no delays in this scenario and the FIFO is set up to evaluate the latest signals. In both halves the struck element of the FTH deviates from the intersection of the two window counters. In case of the left half, the deviation amounts to only one element, wherefore it still counts towards the trigger conditions, but the deviation in the right half exceeds the 15° which are tolerated, as the struck scintillator deviates two elements from the intersection, and accordingly this penetration is only considered to have reached the two window counters. The settings are shown in fig. 5.5 and a screenshot can be found in fig. 5.6[3].

[2]The output shown in the screenshot is too small if placed within the text
[3]A larger version of the screenshot can also be found in the appendix B

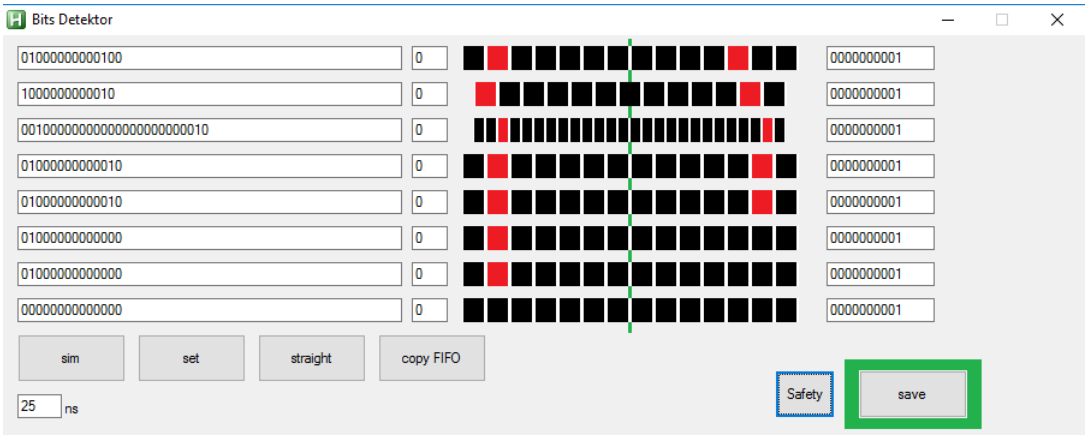


Figure 5.5.: Configuration for two particles with different penetration depths.

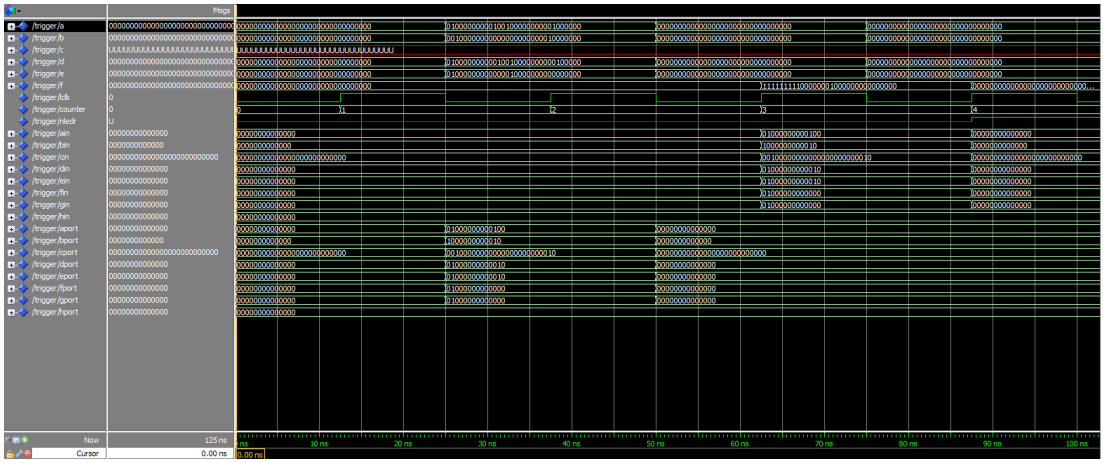


Figure 5.6.: Signals for two particles with different penetration depths. The output shows *1111111110000010*, which means that both halves have been hit and the maximal penetration depth is determined by the left particle which traveled all the way to the fourth range hodoscope.

All in all, the simulations show the expected results in all four cases. Further simulations with similar configurations were run and in all cases the output matches the particle penetration set up in the *Autohotkey* tool.

6. Tests with the developed firmware

6.1. Devices and wiring

After the firmware was transferred to the FPGA, further tests are run to check the functionality of the trigger. For these tests a *Dual Gate Generator* is used to send a pulse to a coincidence level used as FAN-IN and FAN-OUT, where the signals are forwarded to a NIM-to-ECL converter, from where the ECL signals are sent to the ports of the FPGA. Only a couple of ports are actually connected with cables from the converter and the cables have to be reconnected in order to change the pretended particle penetration. Permanent input signals are achieved by using the complement *or-output* of the coincidence level which is not connected to any input signals and therefore permanently sends an output signal. The setup is shown in fig. 6.1.

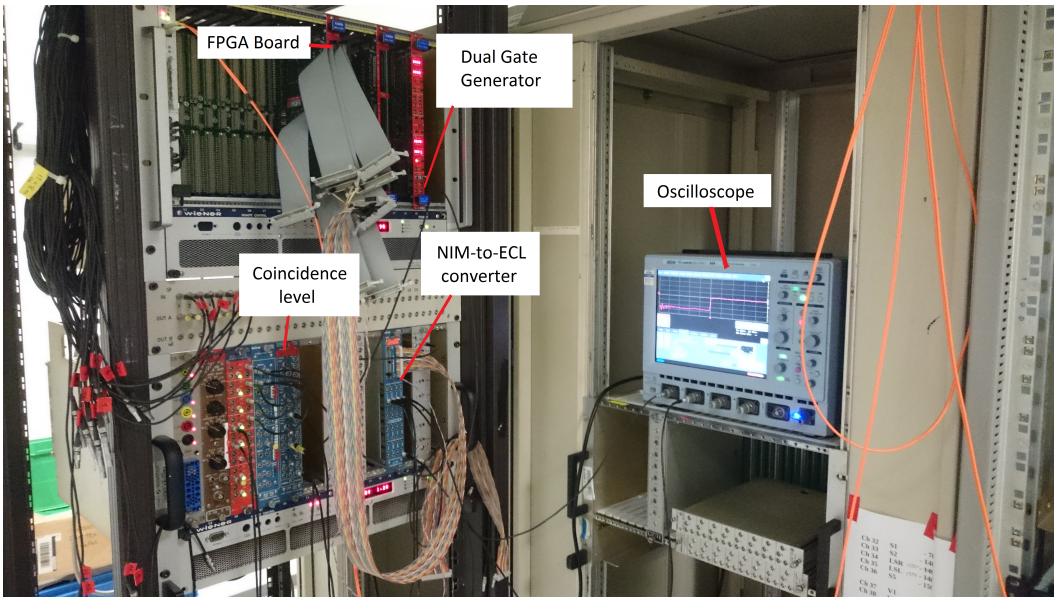


Figure 6.1.: The devices and wiring used to test the trigger.

A simpler test in which the two LEDs of the FPGA board are used to display a particle strike reaching an appointed layer of the detector is also run, but due to the lack of demonstrative output, except for a flashing LED, the results from those tests are not included at this point. These tests are used to check whether a particle penetration up to the first range hodoscope is correctly evaluated and the output signal contained a logical 1 for the respective quarter of the detector as well as for the first four values representing

the penetration depth. In this way, it is also possible to check whether a deviation of the FTH is correctly evaluated, which is the case for all tested configurations.

6.2. Tests with a single output signal

Fig. 6.2 shows the incoming signals used for the first couple of tests.

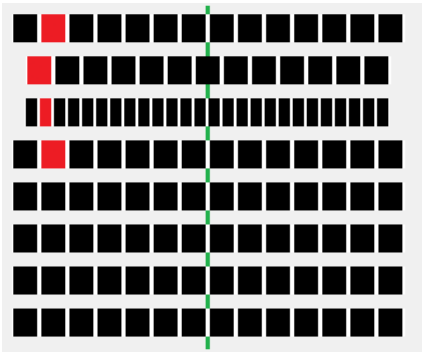


Figure 6.2.: The signals used for the first tests of the transferred firmware as they would look like in the WASA detector.

The following figures show signals displayed on an oscilloscope used to measure the functionality of the FIFO and delays in the signal processing. The yellow signal is the signal that is send to the input pin that will be connected to the second scintillator of the first range hodoscope. The red signal is the fifth output pin of the trigger, which is the third value of the eight bit output that represents the penetration depth and therefore shows whether the particle reached the trigger hodoscope. The setup is also shown in fig. 6.3. By using this configuration a pulse can be send to the trigger hodoscope and the trigger should then send an output signal to the oscilloscope. As a first test the FIFO is configured to use only the latest signals from all layers. Fig. 6.4 shows the signals as they are displayed on the oscilloscope.

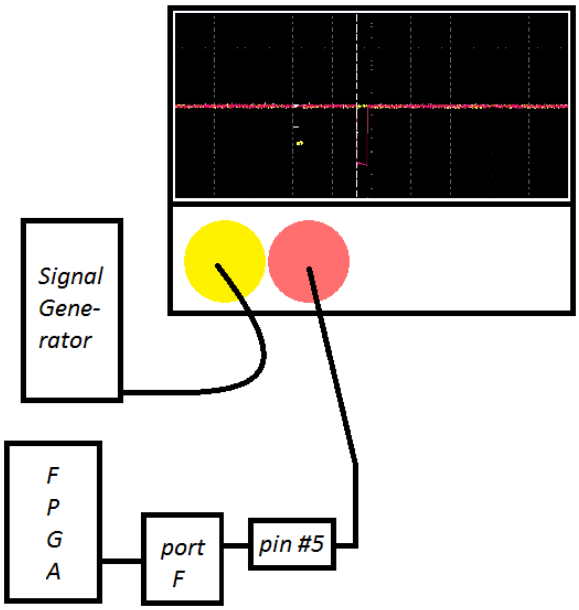


Figure 6.3.: The wiring used to run the first couple of tests.

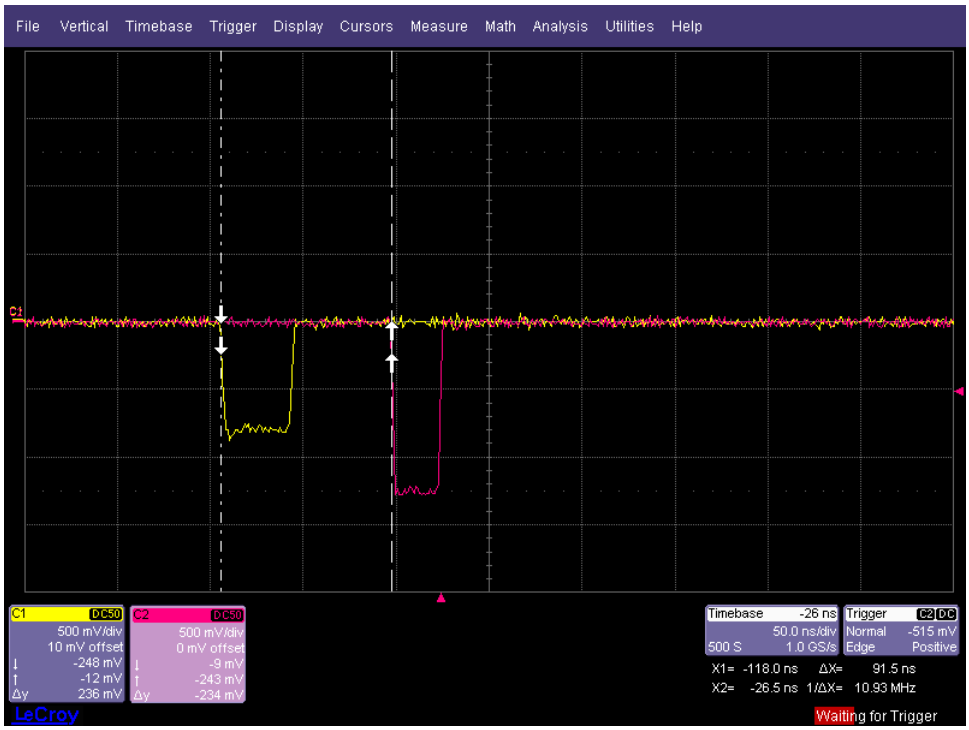


Figure 6.4.: Signals from a test using the latest signals stored in the FIFO. The yellow signal represents the signals sent to the FPGA and the red signal is the fifth pin of the output F, as illustrated in fig. 6.3.

One can see that the trigger signal is sent $t_1 = 91.5\,\text{ns}$ after the incoming signal. Up to $t_{jit,1} = 25\,\text{ns}$ of t_1 can derive from the jitter between the time the signal is sent and the time the next clock pulse starts. The remaining delay derives from the time it takes the FPGA to evaluate the signals and generate the output as well as cable delays. This means that the latency in this measurement is somewhere between $t_{lat,min} = 66.5\,\text{ns}$ and $t_{lat,max} = 91.5\,\text{ns}$. Although the jitter does not cause any problems for the planned experiments, it could be reduced even more by using an *OR* coincidence between the trigger signal and the signal from the first hodoscope.

For a second test, using the same signals, the FIFO is set up to use only the oldest entry. This means that the signal is sent with a delay of $t_{delay} = 225\,\text{ns}$. Fig. 6.5 shows the signals displayed on the oscilloscope. The oscilloscope shows a delay of $t_{10} = 310\,\text{ns}$. $t_{delay} = 225\,\text{ns}$ derive from the delay due to the FIFO setup and the remaining $t_{rem} = 85\,\text{ns}$ derive from the jitter and the processing time. Depending on how large the jitter is, the latency for this measurement is between $t_{lat,min} = 60\,\text{ns}$ and $t_{lat,max} = 85\,\text{ns}$. This test shows that the readout of the FIFO can successfully be adjusted and the generated delay matches the expected duration.

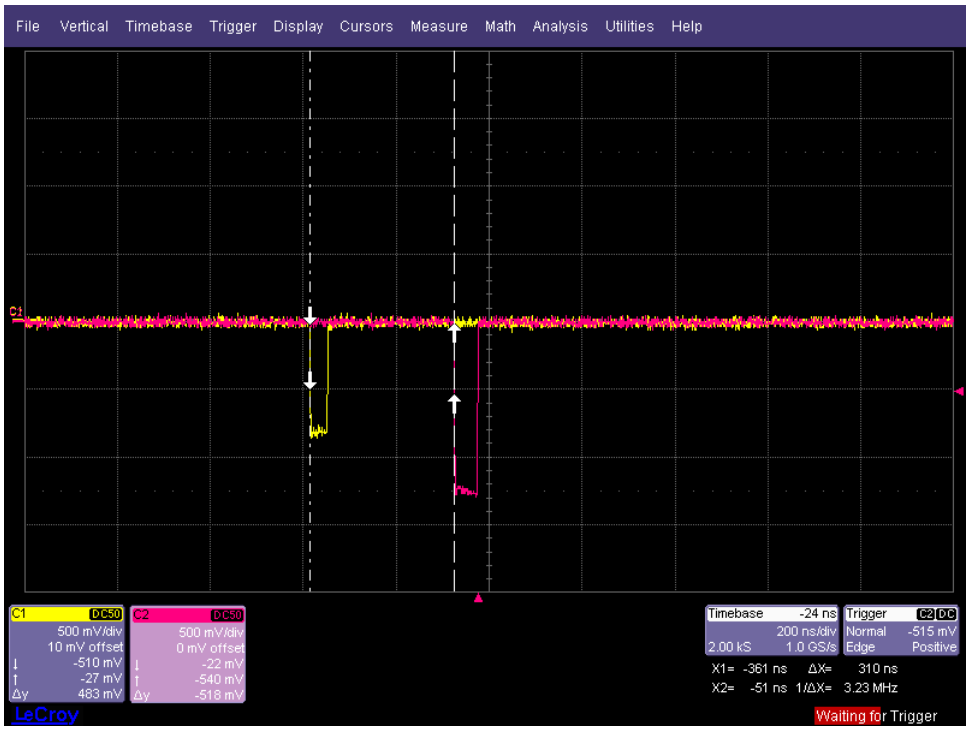


Figure 6.5.: Signals from a test using the oldest signals stored in the FIFO. The yellow and red signals again correspond to the configuration shown in fig. 6.3.

In a third test the same signals are used but the FIFO is configured to evaluate all ten entries for each layer. This means that the output signal should have a width of $t_{FIFO} = 250\,\text{ns}$. The fig. 6.6 shows the resulting signals and the width of the red signal, representing the output of the FPGA, amounts to $t_{output} = 251\,\text{ns}$, which matches the expected width perfectly[1]. The minimal deviation can originate from the distortion of the output signal, which does not match a perfect square-wave signal.

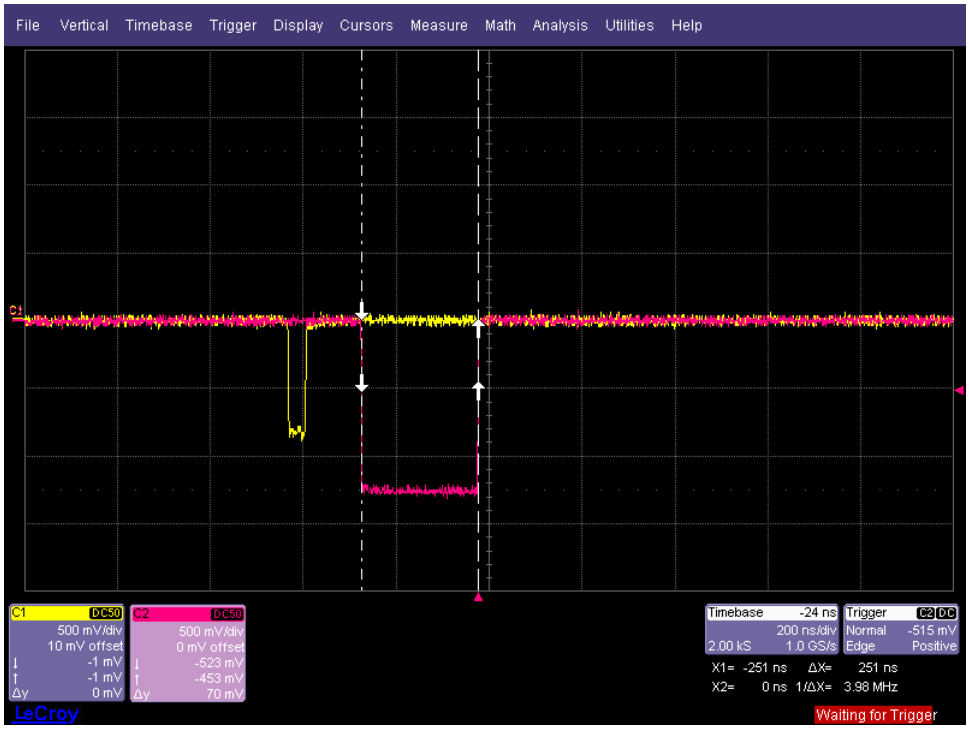


Figure 6.6.: Signals from a test using the entire set of signals stored in the FIFO. The markers show that the width of the signal matches the expected 250 ns perfectly.

[1] Resolution: $\frac{1}{250} = 0.4\%$

Additionally, the persistence mode of the oscilloscope is activated and the *Gate Generator* is adjusted to send periodical signals for an expanded period of time. This makes it possible to see the overlap of numerous signals and the occurring jitter. As fig. 6.7 shows, the time between the arrival of the incoming signal at the ports of the FPGA and the start of the next clock pulse fluctuates and causes a jitter of up to 25 ns.

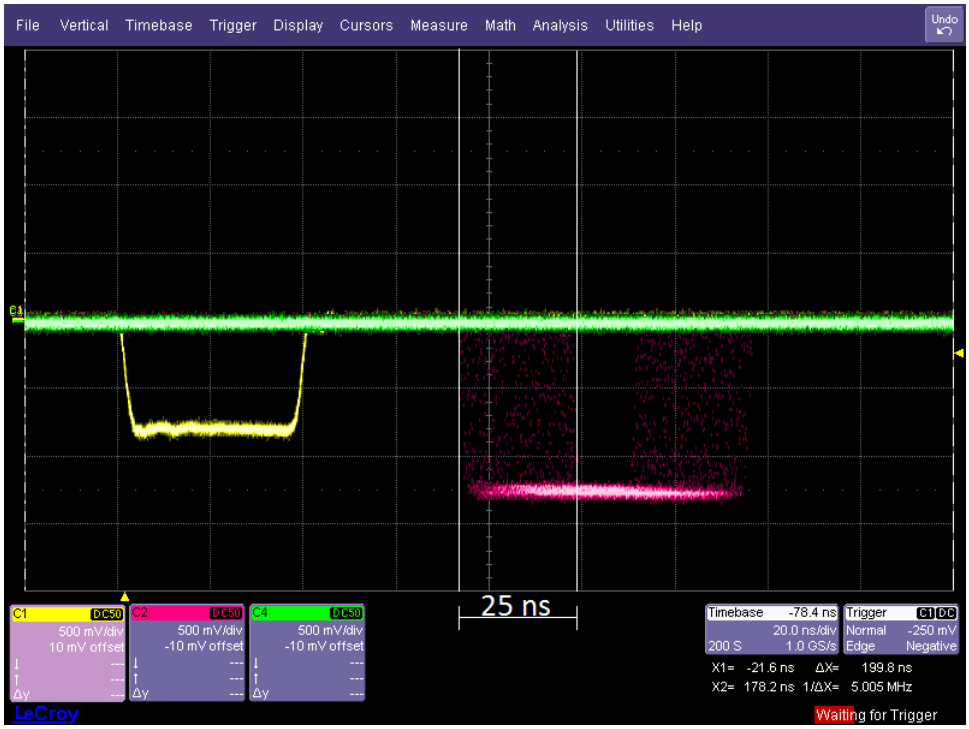


Figure 6.7.: Multiple trigger signals recorded by using the persistence mode of the oscilloscope. The vertical white lines demonstrate that the jitter is limited to 25 ns

6.3. Tests using the FTH and FRH2

After these first tests are run and the output signals satisfied the expectations, the input signals are expanded to the first and second range hodoscope and relocated to the other quarter covered by this FPGA to make sure that this part of the detector is also correctly evaluated. Additionally to the pin showing whether the penetration reaches the trigger hodoscope, a second output pin, namely the one showing a penetration of the second range hodoscope, is connected and displayed on the oscilloscope, as shown in fig. 6.8.

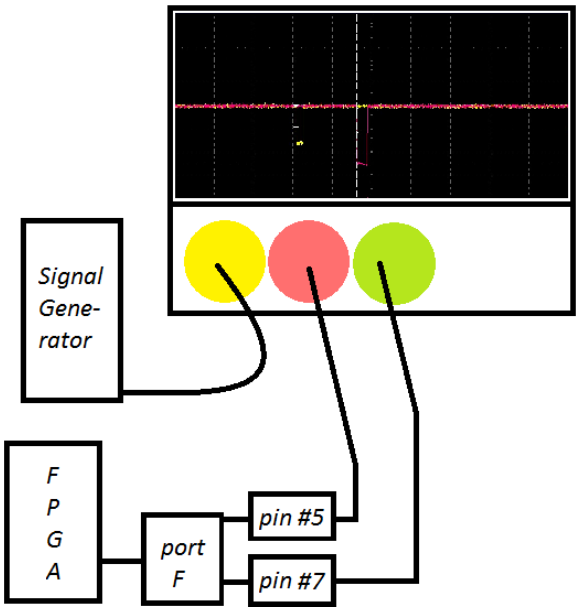


Figure 6.8.: The wiring used for the next tests. A second output signal was added to test further functions of the trigger system.

After the rewiring, a couple of configurations of the FIFO are tested with this new setup. In a first test, the FIFO is configured to use the lastest signals from the trigger hodoscope and the range hodoscope so that the output signals should match each other. The results are shown in fig. 6.9 and correspond to the expectation that both signals overlap.

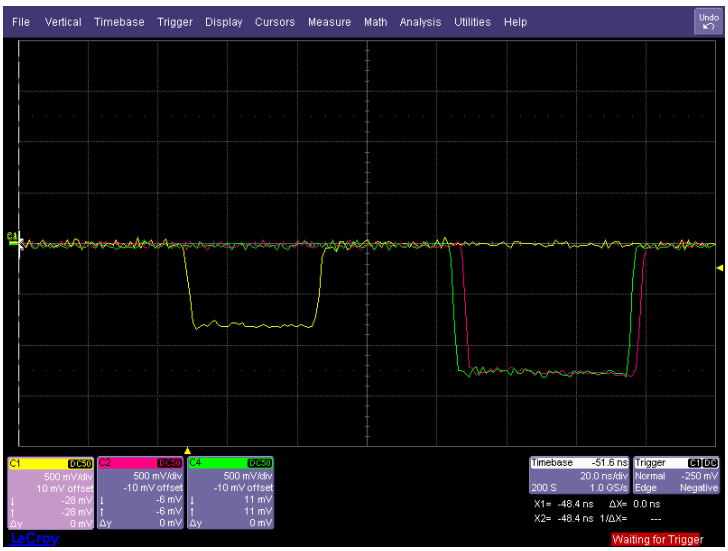


Figure 6.9.: Signals from a test showing the penetration of the trigger as well as the second range hodoscope. The three signals correspond to the configuration shown in fig. 6.8.

It is obvious that the output represents the chosen FIFO and input configurations. To check whether the FIFO correctly distinguishes between the different layers of the detector, the FIFO is then adjusted to evaluate the latest entry from the FRH2 but the two latest entries from the FTH. This means that the signal from the FTH should be twice as wide as the signal from the FRH2. The results can be found in fig. 6.10. They show the signals from the new configuration as expected.

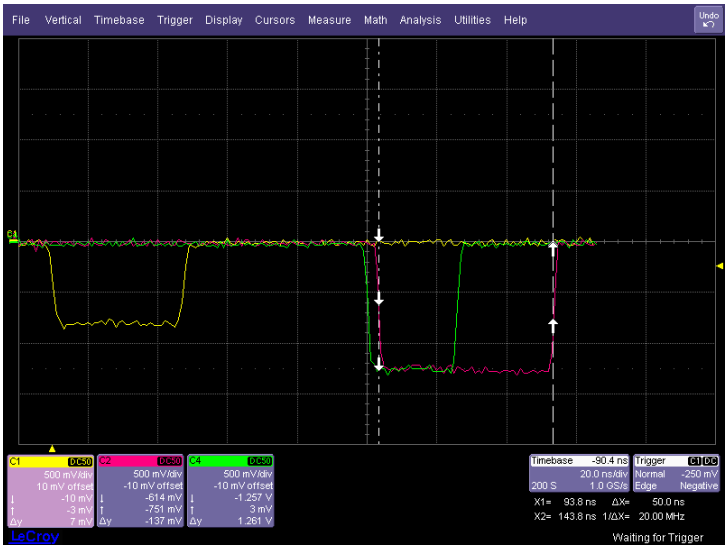


Figure 6.10.: Signals from a test showing the penetration of the FTH as well as the FRH2 with different widths.

After the width of the output signal was successfully changed in the previous tests, a last test is run that used the same signals from the FTH and FRH2 but now the FIFO was set up to use all ten entries from the FTH and only the first two respectively last two entries from the FRH2. This means that a signal with a width of $t_{FRH2} = 50\,\text{ns}$ should be located at the beginning respectively the end of a signal with a width of $t_{FTH} = 250\,\text{ns}$ from the FTH. The figures 6.11 and 6.12 show the output and in both cases the FPGA evaluated the signals correctly.

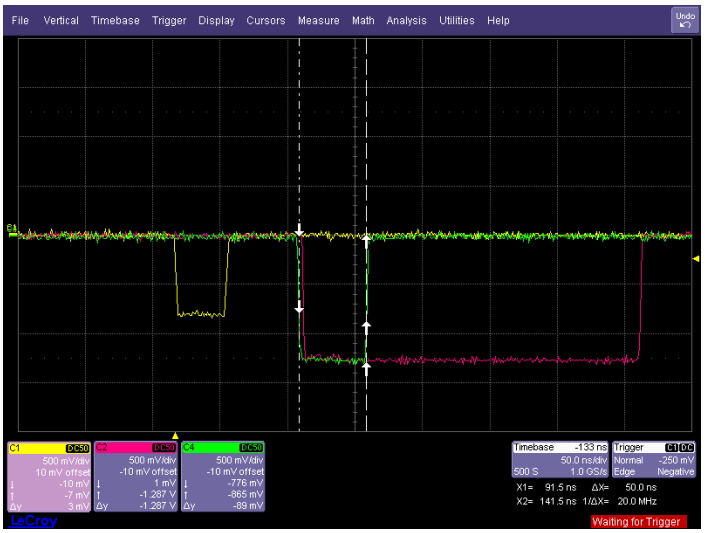


Figure 6.11.: Signals from a test showing the penetration of the FTH as well as the FRH2 where the FIFO is set up to use the two youngest signals from the FRH2.

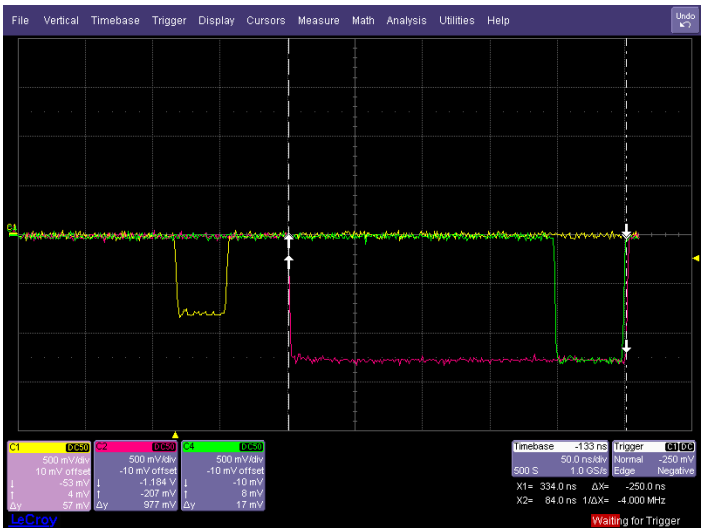


Figure 6.12.: Signals from a test showing the penetration of the FTH as well as the FRH2 where the FIFO is set up to use the two oldest signals from the FRH2.

6.4. Tests using the first three range hodoscopes

Next, the pretended particle penetration is expended to reach all the way to the third range hodoscope and the output of the three struck range hodoscopes is displayed on the oscilloscope. In this case, the red signals corresponds to the signal from the *Depth* output of the FRH1, the green signal corresponds to the output of the FRH2 and the blue signal to the FRH3. In a first, rather simple test, the FIFO is set to read the latest signals from all layers of the detector.

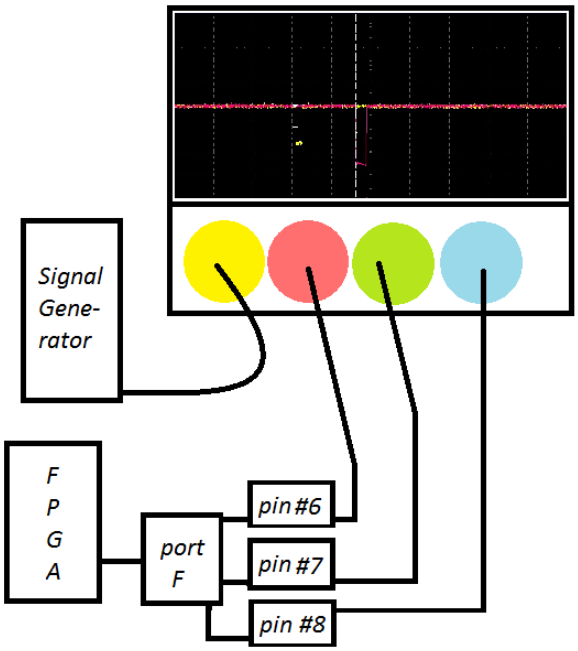


Figure 6.13.: The wiring used for the last couple of tests with four signals displayed on the oscilloscope. The three signals from the FPGA represent the first three range hodoscopes.

The output can be found in fig. 6.14. As expected the signals from the three range hodoscopes overlap each other and the width of every signal amounts to 25 ns.

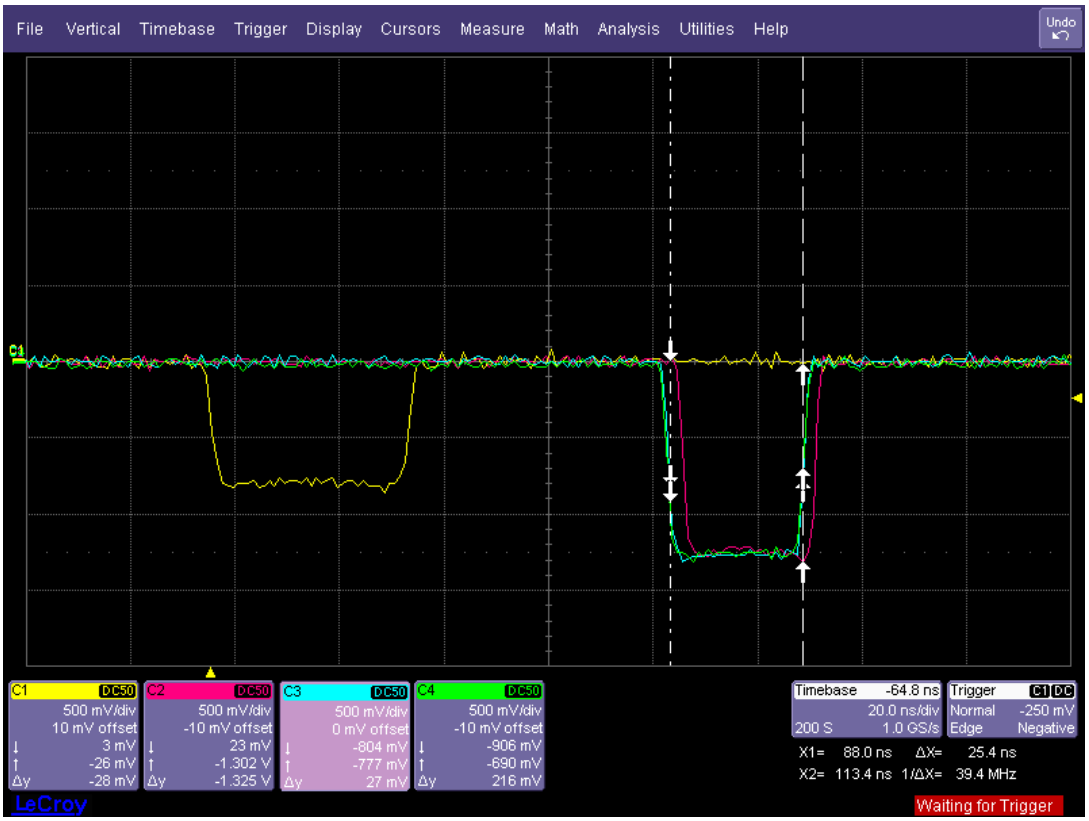


Figure 6.14.: Signals from a test showing the penetration of the first three range hodoscopes.

Additionally, the FIFO is reconfigured to evaluate the three youngest entries of the first four layers and the second and third youngest entry of the fifth layer, the FRH2, as well as the third youngest entry of the FRH3. The output should therefore show three differently wide pulses, where the width decreases the deeper the particle traveled into the detector. The output is shown in fig. 6.15. As can be seen, the signal of the FRH1 covers a width of 75 ns, the signal of the FRH2 covers the later 50 ns and the signal from the FRH3 covers only the last 25 ns. This shows that the signals from the range hodoscopes are evaluated without any problems and even more difficult settings from the FIFO are processed correctly.

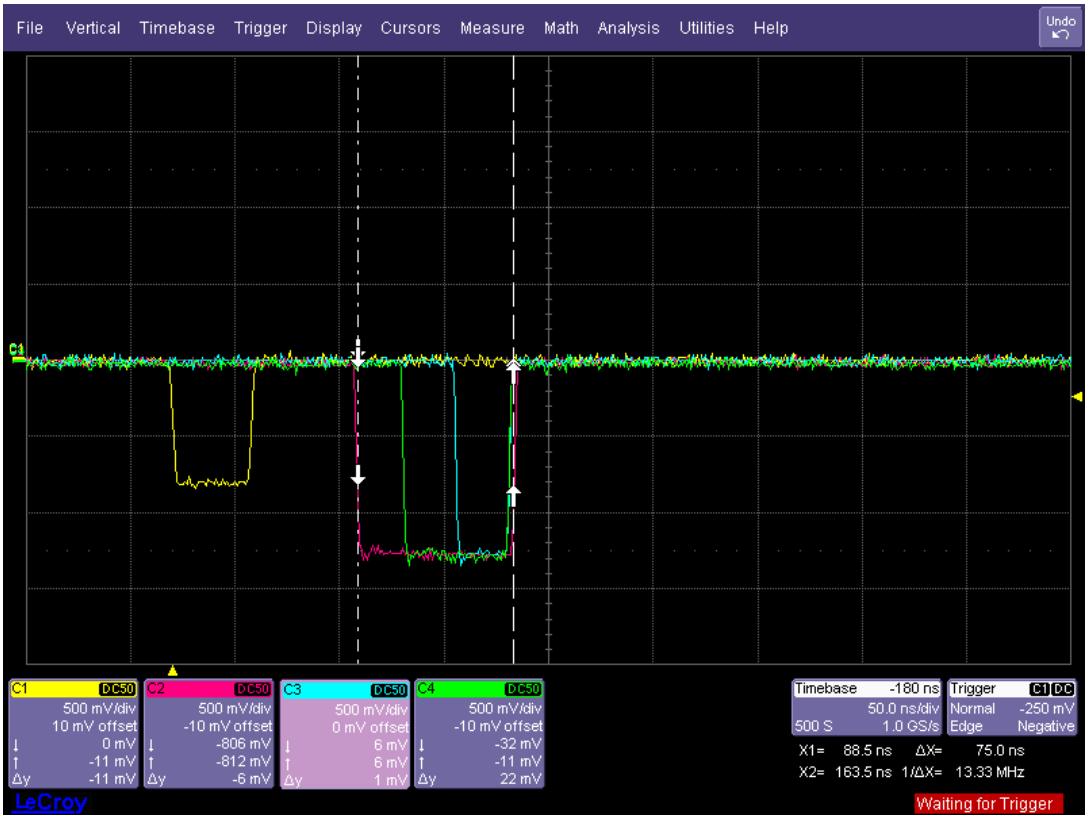


Figure 6.15.: Signals from a test showing the penetration of the first three range hodoscopes with increasingly smaller signal widths.

Finally, the jitter that occurs for this configuration of incoming signals and the last FIFO setup is also recorded. The fig. 6.16 shows the results. The procedure corresponds with the above-mentioned procedure to record the jitter. The jitter of each of the three incoming signals can be seen with the expected offset of 25 ns between the three layers, due to the increasingly smaller signal width. As is expected, the right slope of all three signals overlaps, corresponding to the way the FIFO is set up. The offset of 25 ns between the signals demonstrates the jitter of up to $t_{jit,1} = 25\,\text{ns}$ between the three channels as the space between the afterglows of the FRH1 and FRH3 is entirely covered by the afterglow of the FRH2.

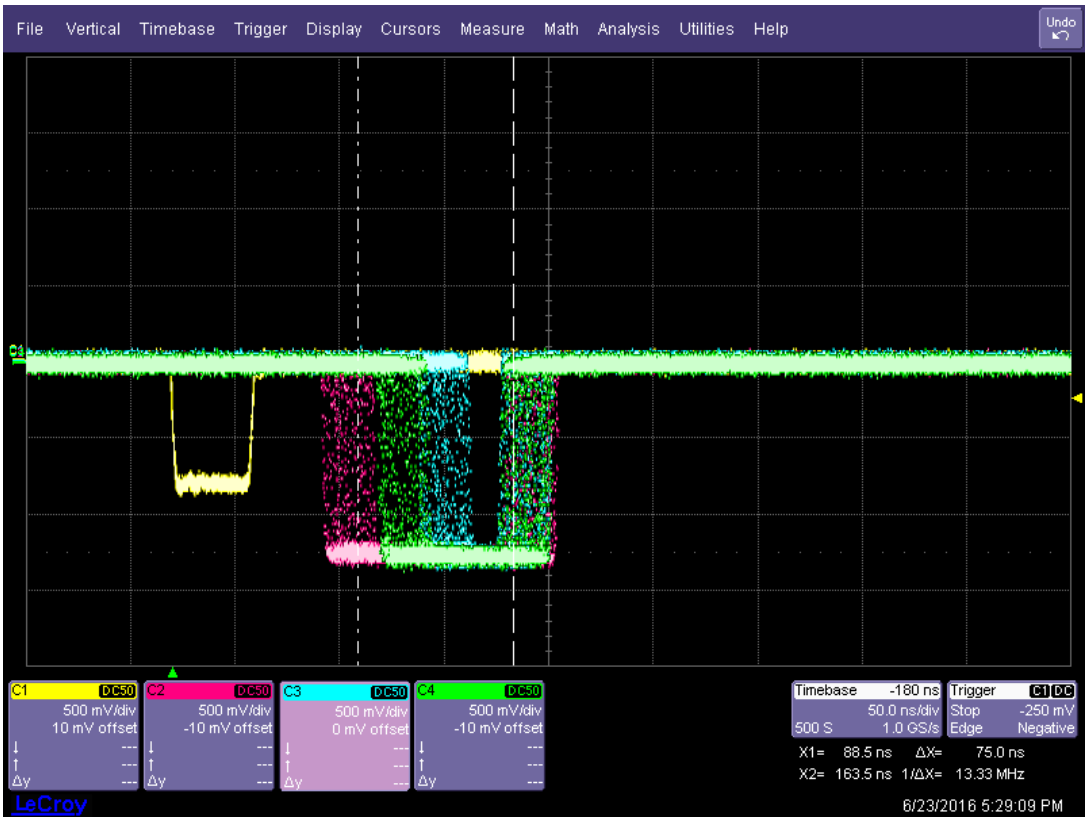


Figure 6.16.: Display of the jitter for the configuration used in fig. 6.15. Each signal has a jitter of up to 25 ns. The three 25 ns intervals are offset by one clock pulse per layer.

All the tests match the expectations and in combination with the simulations run in Modelsim-Altera no malfunctions of the FPGA trigger are found. Thus, the logical connectives as well as the implemented FIFO seem to work without any problems.

7. Conclusion

The goal of this work was to design a trigger for the WASA detector at COSY. A FPGA was used to implement the logical connectives necessary to evaluate the signals from the detector. The trigger will be used for the search for electric dipole moments, wherefore its design was specifically based on the necessary quartering of the detector.

All proposed functions were successfully implemented in the trigger system. The trigger is based on the WASA detector and each FPGA will evaluate one of the two halves of the detector. As planned, the trigger system can determine the position of the particle penetration and the output signal will show which quarter of the detector was struck. The eight layers will also be examined with respect to the penetration depth. Two eight bit vectors will include information on how deep the particles traveled into the detector. The FIFO allows completely individual adjustments for each layer of the detector. Signals from up to ten clock pulses can be evaluated at the same time and, as the simulations and tests with actual signals showed, the FIFO will indeed process all signals correctly and the adjustment via an external computer works fluently.

Multiple possible scenarios were simulated using Modelsim-Altera by setting up particle penetrations and the results matched the expectations. Tests with the transferred firmware were run using actual electronic signals and the output was observed by using the included LEDs of the VME board as well as an oscilloscope. The different configurations of incoming signals were correctly processed.

Unfortunately, the trigger system could not be tested as part of the detector it was designed for, the WASA detector in the cooler synchrotron COSY, as no beam time was schedule for the period of this three month project. However, since the simulations and test signals were based on the detector layout, the trigger system should also work in the WASA detector.

All in all, the ideas behind the trigger system were successfully translated into a hardware design, which was then transferred to the FPGA and tested without experiencing any problems.

The trigger can now be used to search for electric dipole moments. With the integrated conditions, events can be categorized and stored for the offline analysis. It is also possible to trigger further online analysis of the signals. A third FPGA was already ordered and will be used to evaluate the signals even further.

Bibliography

[1] Wolfgang Demtröder, *Experimentalphysik 4, Kern-, Teilchen- und Astrophysik*, 3rd ed., 2010.

[2] C. Rod Nave, *Cyclotron*, 2012.

[3] Lutz Feld, *Teilchen- und Astrophysik: Teilchenbeschleuniger*, Aachen, 2015.

[4] William Brooks, *Fixed-Target Electron Accelerators*, 2001.

[5] Fabian Hinder, *Upgrade of the readout electronics for the EDDA-Polarimeter at the storage ring COSY*, Aachen, October 14th, 2013.

[6] Gersh Itskovich Budker, *An effective method of damping particle oscillations in proton and antiproton storage rings*, May, 1967.

[7] John Marriner, *Stochastic Cooling Overview*, August 11th, 2003

[8] Paul Goslawski et al., *High precision beam momentum determination in a synchrotron using a spin-resonance method*, December 28th, 2013.

[9] James Ritman, *Proposal for the Wide Angle Shower Apparatus (WASA) at COSY-Jülich "WASA at COSY"*, Jülich, October 5th, 2004.

[10] *Research Fundamentals - What are scintillator materials?*, `http://web.stanford.edu/group/scintillators/scintillators.html`, (accessed: 7/05/2016) Leland Stanford Junior University.

[11] Volker Hejny, Irakli Keshelashvili, Edward J. Stephenson, *Proposal EDM Polarimeter Database for Deuterons and Protons*, Jülich, 2015.

[12] Xilinx Inc., *Field Programmable Gate Array (FPGA). What is an FPGA?*, `http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm`, (accessed: 6/7/2016).

[13] CAEN Electronic Instrumentation, *Technical Information Manual*, revision n. 16, June 15th, 2015.

[14] Peter J. Ashenden, *The VHDL Cookbook*, 1st ed., Adelaide, July, 1990.

List of Figures

List of Tables

Appendices

A. VHDL code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.components.all;

entity Trigger is
        Generic (
                constant Reading_Depth : positive := 10;                -- SELECT HOW
                 ↪  MANY ENTRIES OF THE FIFO CAN BE EVALUATED PER CLOCK PULSE
                constant DATA_WIDTH    : positive := 123;         -- SELECT HOW MANY
                 ↪  VALUES ARE SAVED IN THE FIFO PER ENTRY (=amount of input pins)
                constant FIFO_DEPTH          : positive := 10;                  -- THE
                 ↪  AMOUNT OF ENTRIES IN THE FIFO
                constant Speichen      : positive := 14                 -- AMOUNT OF
                 ↪  SLICES THAT ARE EVALUATED PER FPGA FROM THE FWC 1
        );
        Port (
    -- Front Panel Ports
    A        : IN     std_logic_vector (0 to 31);
     ↪                                              -- In A (32 x LVDS/ECL) INVERSE
    B        : IN     std_logic_vector (0 to 31);
     ↪                                                      -- In B (32 x
     ↪  LVDS/ECL) INVERSE
    C        : OUT    std_logic_vector (0 to
     ↪  31);                                                -- Out C (32 x LVDS)
     ↪  INVERSE
    D        : INOUT  std_logic_vector (0 to 31);
     ↪                                                      -- In/Out D (I/O
     ↪  Expansion) INVERSE
    E        : INOUT  std_logic_vector (0 to 31);
     ↪                                                 -- In/Out E (I/O Expansion)
     ↪  INVERSE
    F        : INOUT  std_logic_vector (0 to 31) := (others => '0');         -- In/Out F
     ↪  (I/O Expansion) INVERSE
    GIN      : IN     std_logic_vector ( 1 DOWNTO 0);
     ↪                                          -- In G - LEMO (2 x NIM/TTL)
    GOUT     : OUT    std_logic_vector ( 1 DOWNTO 0);
     ↪                                          -- Out G - LEMO (2 x NIM/TTL)
    -- Port Output Enable (0=Output, 1=Input)
    nOED     : OUT    std_logic :='1';                        -- Output Enable Port D
     ↪  (only for A395D)
    nOEE     : OUT    std_logic :='1';                        -- Output Enable Port E
     ↪  (only for A395D)
    nOEF     : OUT    std_logic :='0';                        -- Output Enable Port F
     ↪  (only for A395D)
    nOEG     : OUT    std_logic :='1';                        -- Output Enable Port G
    -- Port Level Select (0=NIM, 1=TTL)
    SELD     : OUT    std_logic;                      -- Output Level Select Port D
     ↪  (only for A395D)
```

```
    SELE     : OUT    std_logic;                        -- Output Level Select Port E
     ↪  (only for A395D)
    SELF     : OUT    std_logic;                        -- Output Level Select Port F
     ↪  (only for A395D)
    SELG     : OUT    std_logic := '0';                 -- Output Level Select Port G

    -- Expansion Mezzanine Identifier:
    -- 000 : A395A (32 x IN LVDS/ECL)
    -- 001 : A395B (32 x OUT LVDS)
    -- 010 : A395C (32 x OUT ECL)
    -- 011 : A395D (8  x IN/OUT NIM/TTL)
    IDD      : IN     std_logic_vector (2 DOWNTO 0);   -- Slot D
    IDE      : IN     std_logic_vector (2 DOWNTO 0);   -- Slot E
    IDF      : IN     std_logic_vector (2 DOWNTO 0);   -- Slot F

    -- Delay Lines
    -- 0:1 => PDL (Programmable Delay Line): Step = 0.25ns / FSR = 64ns
    -- 2:3 => FDL (Free Running Delay Line with fixed delay)
    PULSE    : IN     std_logic_vector (3 DOWNTO 0);   -- Output of the delay line (0:1
     ↪  => PDL; 2:3 => FDL)
    nSTART   : OUT    std_logic_vector (3 DOWNTO 2);   -- Start of FDL (active low)
    START    : OUT    std_logic_vector (1 DOWNTO 0);   -- Input of PDL (active high)
    DDLY     : INOUT  std_logic_vector (7 DOWNTO 0);   -- R/W Data for the PDL
    WR_DLY0  : OUT    std_logic;                        -- Write signal for the PDL0
    WR_DLY1  : OUT    std_logic;                        -- Write signal for the PDL1
    DIRDDLY  : OUT    std_logic;                        -- Direction of PDL data (0 =>
     ↪  Read Dip Switches)
                                                        --                       (1 =>
                                                         ↪  Write from FPGA)
    nOEDDLY0 : OUT    std_logic;                        -- Output Enable for PDL0 (active
     ↪  low)
    nOEDDLY1 : OUT    std_logic;                        -- Output Enable for PDL1 (active
     ↪  low)

    -- LED drivers
    nLEDG    : OUT    std_logic;                                -- Green (active low)
    nLEDR    : OUT    std_logic;            -- Red   (active low)

    -- Spare
    -- SPARE    : INOUT  std_logic_vector (11 DOWNTO 0);

    -- Local Bus in/out signals
    nLBRES     : IN     std_logic;
    nBLAST     : IN     std_logic;
    WnR        : IN     std_logic;
    nADS       : IN     std_logic;
    LCLK       : IN     std_logic;
    nREADY     : OUT    std_logic;
    nINT       : OUT    std_logic;
    LAD        : INOUT  std_logic_vector (15 DOWNTO 0)
        );
end Trigger;


architecture behavioral of Trigger is
```

```
-----------------------------------------------------------
--SIGNALS FOR EACH LAYER OF SCINTILLATORS ARE GENERATED--
-----------------------------------------------------------
signal APort : std_logic_vector (0 to Speichen-1);
signal BPort : std_logic_vector (0 to Speichen-2);
signal CPort : std_logic_vector (0 to 2*(Speichen-1)-1);
signal DPort : std_logic_vector (0 to Speichen-1);
signal EPort : std_logic_vector (0 to Speichen-1);
signal FPort : std_logic_vector (0 to Speichen-1);
signal GPort : std_logic_vector (0 to Speichen-1);
signal HPort : std_logic_vector (0 to Speichen-1);
-------------------------------------------------------------
--SIGNALS FOR PENNETRATION DEPTH AND LOCATION ARE GENERATED--
-------------------------------------------------------------
shared variable Quarter : std_logic_vector (0 to 1);
shared variable Output  : std_logic_vector (0 to 2*(Speichen-1)-1);
shared variable Depth   : std_logic_vector (0 to 7);
shared variable MPD          : std_logic_vector (0 to 7);

-------------------------------------------------------------
-----------------USED REGISTERS ARE DECLARED-----------------
-------------------------------------------------------------

signal REG_R6                                        : std_logic_vector(31 downto 0) :=
 ↪  (others => 'Z');

signal REG_RW1                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW2                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW3                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW4                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW5                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW6                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW7                             : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW8                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');
signal REG_RW9                              : std_logic_vector(31 downto 0) := (others =>
 ↪  'Z');


-------------------------------------------------------------
-----FIFO READING DEPTH FOR THE DIFFERENT LAYERS ARE SET-----
-------------------------------------------------------------

--signal Aread                          : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";
--signal Bread                            : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";
--signal Cread                            : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";
```

```
--signal Dread                          : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";
--signal Eread                          : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";
--signal Fread                          : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";
--signal Gread                          : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";
--signal Hread                          : std_logic_vector (0 to Reading_Depth-1)  :=
 ↪  "1000000000";


signal Aread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');
signal Bread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');
signal Cread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');
signal Dread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');
signal Eread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');
signal Fread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');
signal Gread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');
signal Hread  : std_logic_vector (Reading_Depth-1 downto 0)  := (others => 'Z');


------------------------------------------------------------
----SIGNALS FOR FIFO OUTPUT ARE GENERATED FOR ALL LAYERS-----
------------------------------------------------------------
shared variable Ain     : std_logic_vector(0 to Speichen-1)               :=(others =>
 ↪  '0');
shared variable Bin     : std_logic_vector (0 to Speichen-2)              :=(others =>
 ↪  '0');
shared variable Cin     : std_logic_vector (0 to 2*(Speichen-1)-1)        :=(others =>
 ↪  '0');
shared variable Din,Ein,Fin,Gina,Hin : std_logic_vector (0 to Speichen-1)  :=(others =>
 ↪  '0');

signal Data                             : std_logic_vector (0 to (FIFO_DEPTH -
 ↪  1)*DATA_WIDTH+(DATA_WIDTH-1)) := (others => '0'); --FIFO Memory is generated
shared variable counter : natural
 ↪                                                                          := 0; --
 ↪  Counter that increases with each clock pulse
signal LED                              : natural; -- Counter that increases with each
 ↪  clock pulse to show how long LED is turned on
signal nLEDRV                           : std_logic := '0'; -- Signal that will link the
 ↪  output channel of the LED with the procedures that evaluate the incoming signals


---------------------------------------------------------------------
-----PROCEDURE THAT WILL CALCULATE THE MAXIMAL PENETRATION DEPTH-----
---------------------------------------------------------------------
procedure maximal_depth
        (Depth         : in std_logic_vector;
         MPD               : out std_logic_vector (0 to 7)) is
begin
        MPD  := (others => '0');
        for i in 0 to 7 loop
                if (Depth(i)='1') then
                        MPD := (i => '1', others => '0');
```

```
                end if;
        end loop;
end maximal_depth;


-------------------------------------------------------------------------------------------
-----PROCEDURES THAT WILL CALCULATE THE PENETRATION DEPTH PER LAYER CONSIDERING
 ↪  DRIFTS----
-------------------------------------------------------------------------------------------
procedure depth_straight
        (Ain                                : in std_logic_vector (0 to Speichen-1);
         Bin                                : in std_logic_vector (0 to Speichen-2);
         Cin                                : in std_logic_vector (0 to
          ↪  2*(Speichen-1)-1);
         Din                                : in std_logic_vector (0 to Speichen-1);
         Ein                                : in std_logic_vector (0 to Speichen-1);
         Fin                                : in std_logic_vector (0 to Speichen-1);
         Gina                                : in std_logic_vector (0 to Speichen-1);
         Hin                                : in std_logic_vector (0 to Speichen-1);
         i                                 : in natural;
         Depth                          : out std_logic_vector (0 to 7)) is
begin
        Depth(3 to 7) := (3 to 7 => '0');
        if (Din(i)=Ain(i)) then -- Further layers are being evaluated to find the
         ↪  maximal penetration depth: FRH 1
                if (Ain(i)=Ein(i)) then -- Further layers are being evaluated to find
                 ↪  the maximal penetration depth: FRH 2
                        if (Ain(i)=Fin(i)) then -- Further layers are being evaluated to
                         ↪  find the maximal penetration depth: FRH 3
                                if (Ain(i)=Gina(i)) then -- Further layers are being
                                 ↪  evaluated to find the maximal penetration depth:
                                 ↪  FRH 4
                                        if (Ain(i)=Hin(i)) then -- Further layers are
                                         ↪  being evaluated to find the maximal
                                         ↪  penetration depth: FRH 5
                                                Depth := (others => '1');
                                        else
                                                Depth(0 to 6) := (0 to 6 => '1');
                                        end if;
                                else
                                        Depth(0 to 5) := (0 to 5 => '1');
                                end if;
                        else
                                Depth(0 to 4) := (0 to 4 => '1');
                        end if;
                else
                        Depth(0 to 3) := (0 to 3 => '1');
                end if;
        else
                Depth(0 to 2) := (0 to 2 => '1');
        end if;
end depth_straight;



procedure depth_shift_low
```

```
        (Ain                                : in std_logic_vector (0 to Speichen-1);
         Bin                                : in std_logic_vector (0 to Speichen-2);
         Cin                                : in std_logic_vector (0 to
          ↪  2*(Speichen-1)-1);
         Din                                : in std_logic_vector (0 to Speichen-1);
         Ein                                : in std_logic_vector (0 to Speichen-1);
         Fin                                : in std_logic_vector (0 to Speichen-1);
         Gina                                : in std_logic_vector (0 to Speichen-1);
         Hin                                : in std_logic_vector (0 to Speichen-1);
         i                                 : in natural;
         Depth                           : out std_logic_vector (0 to 7)) is
begin
        Depth(3 to 7) := (3 to 7 => '0');
        if (Din(i)=Ain(i)) or (Din(i-1)=Ain(i)) then -- see above
                if (Ain(i)=Ein(i)) or (Ain(i)=Ein(i-1)) then
                        if (Ain(i)=Fin(i)) or (Ain(i)=Fin(i-1)) then
                                 if (Ain(i)=Gina(i)) or (Ain(i)=Gina(i-1)) then
                                        if (Ain(i)=Hin(i)) or (Ain(i)=Hin(i-1)) then
                                                Depth := (others => '1');
                                        else
                                                Depth(0 to 6) := (0 to 6 => '1');
                                        end if;
                                else
                                        Depth(0 to 5) := (0 to 5 => '1');
                                end if;
                        else
                                Depth(0 to 4) := (0 to 4 => '1');
                        end if;
                else
                        Depth(0 to 3) := (0 to 3 => '1');
                end if;
        else
                Depth(0 to 2) := (0 to 2 => '1');
        end if;
end depth_shift_low;



procedure depth_shift_high
        (Ain                                : in std_logic_vector (0 to Speichen-1);
         Bin                                : in std_logic_vector (0 to Speichen-2);
         Cin                                : in std_logic_vector (0 to
          ↪  2*(Speichen-1)-1);
         Din                                : in std_logic_vector (0 to Speichen-1);
         Ein                                : in std_logic_vector (0 to Speichen-1);
         Fin                                : in std_logic_vector (0 to Speichen-1);
         Gina                                : in std_logic_vector (0 to Speichen-1);
         Hin                                : in std_logic_vector (0 to Speichen-1);
         i                                 : in natural;
         Depth                           : out std_logic_vector (0 to 7)) is
begin
        Depth(3 to 7) := (3 to 7 => '0');
        if (Din(i)=Ain(i)) or (Din(i+1)=Ain(i)) then -- see above
                if (Ain(i)=Ein(i)) or (Ain(i)=Ein(i+1)) then
                        if (Ain(i)=Fin(i)) or (Ain(i)=Fin(i+1)) then
                                if (Ain(i)=Gina(i)) or (Ain(i)=Gina(i+1)) then
```

```
                                        if (Ain(i)=Hin(i)) or (Ain(i)=Hin(i+1)) then
                                                Depth := (others => '1');
                                        else
                                                Depth(0 to 6) := (0 to 6 => '1');
                                        end if;
                                else
                                        Depth(0 to 5) := (0 to 5 => '1');
                                end if;
                        else
                                Depth(0 to 4) := (0 to 4 => '1');
                        end if;
                else
                        Depth(0 to 3) := (0 to 3 => '1');
                end if;
        else
                Depth(0 to 2) := (0 to 2 => '1');
        end if;
end depth_shift_high;

-------------------------------------------------------------------------
-----PROCEDURE THAT WILL PROCESS AND GENERATE THE OUTPUT OF THE FIFO-----
-------------------------------------------------------------------------
procedure FIFO
        (Reading_Depth : in positive;
         DATA_WIDTH         : in positive;
         Data                                : in std_logic_vector;
         Aread                          : in std_logic_vector;
         Bread                          : in std_logic_vector;
         Cread                          : in std_logic_vector;
         Dread                          : in std_logic_vector;
         Eread                          : in std_logic_vector;
         Fread                          : in std_logic_vector;
         Gread                          : in std_logic_vector;
         Hread                          : in std_logic_vector;
         Ain                                 : out std_logic_vector (0 to Speichen-1);
         Bin                                 : out std_logic_vector (0 to Speichen-2);
         Cin                                 : out std_logic_vector (0 to
          ↪  2*(Speichen-1)-1);
         Din                                 : out std_logic_vector (0 to Speichen-1);
         Ein                                 : out std_logic_vector (0 to Speichen-1);
         Fin                                 : out std_logic_vector (0 to Speichen-1);
         Gina                                 : out std_logic_vector (0 to Speichen-1);
         Hin                                 : out std_logic_vector (0 to Speichen-1)) is
begin
        Ain  := (others => '0');
        Bin  := (others => '0');
        Cin  := (others => '0');
        Din  := (others => '0');
        Ein  := (others => '0');
        Fin  := (others => '0');
        Gina := (others => '0');
        Hin  := (others => '0');
        for i in 0 to (Reading_Depth-1) loop
         ↪                                                                                --
         ↪  FIFO is evaluted according to chosen READING DEPTH
```

```
for k in 0 to 7 loop
 ↪                                                                    --
 ↪  loop for all 8 scintillator layers
       for j in 0 to Speichen-1
        ↪  loop
        ↪  loop for each slice per layer
              if (k=0) then
               ↪
               ↪  FWC 1 is evaluated
                     if (Data(j+i*DATA_WIDTH)='1') and
                      ↪  (Aread(9-i)='1') then -- IF MEMORY SLOT IS
                      ↪  ACTIVATED FOR READING IN AREAD...
                           Ain(j) := '1';
                            ↪
                            ↪  ...VALUE IS WRITTEN INTO AIN
                     end if;
              elsif (k=1) then -- FWC 2 is evaluated
                     if (j<Speichen-1) then
                           if (Data(j+i*DATA_WIDTH+Speichen*k)='1')
                            ↪  and (Bread(9-i)='1') then -- same
                            ↪  procedure as with FWC 1
                                 Bin(j) := '1';
                           end if;
                     end if;
              elsif (k=2) then -- FTH 1 is evaluated (SAME PROCEDURE
               ↪  AS FOR THE FWC 1)
                     if (j<Speichen-1) then
                           if
                            ↪  (Data(2*j+i*DATA_WIDTH+Speichen*k)='1')
                            ↪  and (Cread(9-i)='1') then
                                 Cin(2*j+1) := '1';
                           elsif
                            ↪  (Data(2*j+i*DATA_WIDTH+Speichen*k-1)='1')
                            ↪  and (Cread(9-i)='1') then
                                 Cin(2*j) := '1';
                           end if;
                     end if;
              else -- FRH 1-5 are evaluated (SAME PROCEDURE AS FOR THE
               ↪  FWC 1)
                     if (Data(j+i*DATA_WIDTH+Speichen*(k+1)-3)='1')
                      ↪  then
                           if (k=3) and (Dread(9-i)='1') then
                                 Din(j) := '1';
                           elsif (k=4) and (Eread(9-i)='1') then
                                 Ein(j) := '1';
                           elsif (k=5) and (Fread(9-i)='1') then
                                 Fin(j) := '1';
                           elsif (k=6) and (Gread(9-i)='1') then
                                 Gina(j) := '1';
                           elsif (k=7) and (Hread(9-i)='1') then
                                 Hin(j) := '1';
                           end if;
                     end if;
              end if;
       end loop;
end loop;
```

```
        end loop;

end FIFO;



begin

-------------------------------------------------------------
---------COMMUNICATION BETWEEN USER FPGA AND THE VME---------
-------------------------------------------------------------


Aread  <= REG_RW1(Reading_Depth-1 downto 0); -- Values from Register 1 are written into
 ↪  Aread - FIFO setup
Bread  <= REG_RW2(Reading_Depth-1 downto 0); -- Values from Register 2 are written into
 ↪  Bread - FIFO setup
Cread  <= REG_RW4(Reading_Depth-1 downto 0); -- Values from Register 3 are written into
 ↪  Cread - FIFO setup
Dread  <= REG_RW5(Reading_Depth-1 downto 0); -- Values from Register 4 are written into
 ↪  Dread - FIFO setup
Eread  <= REG_RW6(Reading_Depth-1 downto 0); -- Values from Register 5 are written into
 ↪  Eread - FIFO setup
Fread  <= REG_RW7(Reading_Depth-1 downto 0); -- Values from Register 6 are written into
 ↪  Fread - FIFO setup
Gread  <= REG_RW8(Reading_Depth-1 downto 0); -- Values from Register 7 are written into
 ↪  Gread - FIFO setup
Hread  <= REG_RW9(Reading_Depth-1 downto 0); -- Values from Register 8 are written into
 ↪  Hread - FIFO setup

-- REG_R6  -->  |    ------ 24 bit ------    --4bit--4bit-|
-- REG_R6  -->  | ... obligatory '0'  ...    |  0  |  0  |
REG_R6(3 downto 0)   <= "0001";  -- Firmware release
--REG_R6(7 downto 4) <= conv_std_logic_vector(1, 4);  -- Demo number
REG_R6(31 downto 4)  <= (others => '0');

instance_LB_INT: LB_INT
        port map (
                        -- Local Bus in/out signals
                        nLBRES      => nLBRES,
                        nBLAST      => nBLAST,
                        WnR         => WnR,
                        nADS        => nADS,
                        LCLK        => LCLK,
                        nREADY      => nREADY,
                        nINT        => nINT,
                        LAD         => LAD,

                        -- Internal Registers
                        REG_R6     => REG_R6,

                        REG_RW1    => REG_RW1,
                        REG_RW2    => REG_RW2,
                        REG_RW3    => REG_RW3,
                        REG_RW4    => REG_RW4,
                        REG_RW5    => REG_RW5,
```

```
                        REG_RW6     => REG_RW6,
                        REG_RW7     => REG_RW7,
                        REG_RW8     => REG_RW8,
                        REG_RW9     => REG_RW9
                );

APort                                       <= not A(0 to Speichen-1);
 ↪                  -- INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FWC
 ↪  1
BPort                                   <= not A(Speichen to 2*Speichen-2);
 ↪                  -- INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FWC
 ↪  2
CPort                                   <= not B(0 to 2*(Speichen-1)-1);
 ↪                  -- INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FTH
 ↪  1
DPort                                   <= not D(0 to Speichen-1);
 ↪                  -- INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FRH
 ↪  1
EPort                                   <= not D(Speichen to 2*Speichen-1);
 ↪                  -- INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FRH
 ↪  2
FPort                                   <= not E(0 to Speichen-1);
 ↪                  -- INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FRH
 ↪  3
GPort                                   <= not E(Speichen to 2*Speichen-1);
 ↪                  -- INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FRH
 ↪  4
HPort(0 to 4)                   <= not A(2*Speichen-1 to
 ↪  31);                                        -- INPUT SIGNALS ARE SPLIT ACCORDING TO
 ↪  LAYER OF SCINTILLATORS: FRH 5 Pins 1 to 5
HPort (5 to 10)           <= not B(2*(Speichen-1) to 31);                           --
 ↪  INPUT SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FRH 5 Pins 6 to 11
HPort (11 to 13)          <= not D(2*Speichen to 2*Speichen+2);                -- INPUT
 ↪  SIGNALS ARE SPLIT ACCORDING TO LAYER OF SCINTILLATORS: FRH 5 Pins 12 to 14

        process begin
                wait until rising_edge(LCLK);

                Data <= Data(DATA_WIDTH to
                 ↪  DATA_WIDTH*FIFO_DEPTH-1)&APort&BPort&CPort&DPort&EPort&FPort&GPort&HPort;
                 ↪  -- new signals are added to FIFO



                FIFO(Reading_Depth, DATA_WIDTH, Data, Aread, Bread, Cread, Dread, Eread,
                 ↪  Fread, Gread, Hread, Ain, Bin, Cin, Din, Ein, Fin, Gina, Hin); --
                 ↪  FIFO is read out


                if (Counter > LED+20000000) then -- RED LED is turned off after flashing
                 ↪  for half a second
                        nLEDR  <= '1';
                end if;

                if (Ain(0 to Speichen-1)=(Ain'range => '0')) or (Bin(0 to
                 ↪  Speichen-2)=(Bin'range => '0')) then -- If particle doesn't reach
                 ↪  FWC 2, no event shall be registered
```

```
        Depth         := (others => '0');   -- In the above-mentioned
         ↪  case, the depth is set to 0...
        Quarter       := (others => '0'); -- ...and the Position is
         ↪  not evaluated
    else -- B TRIGGERED
        Depth         := (others => '0');   -- In the above-mentioned
         ↪  case, the depth is set to 0...
        Quarter       := (others => '0');
        for i in 0 to Speichen-1 loop -- Loop for each of the 14
         ↪  evaluated scintillators of FWC 1
            if (i=0) then -- particle is registered on the edge of
             ↪  the layer
                --lower(nLEDRV, LED, counter, Ain, Bin, Cin,
                 ↪  Din, Ein, Fin, Gina, Hin, Quarter, Output,
                 ↪  Depth);
                if (Ain(i)='1') and (Bin(i)='1') then -- The
                 ↪  position of the particle on FWC 1 & FWC 2
                 ↪  matches
                    nLEDR   <= '0'; -- RED LED is turned on
                    LED <= counter;
                    Quarter(0) := '1';
                    if (Cin(2*i)='1') then -- Particle
                     ↪  struck FTH 1 on lower slice(ON THE
                     ↪  SAME LEVEL AS THE STRUCK SLICE OF
                     ↪  THE FWC 1)
                        Output(2*i) := '1'; -- Output
                         ↪  signal is set according to
                         ↪  the evaluated position
                        depth_straight(Ain, Bin, Cin,
                         ↪  Din, Ein, Fin, Gina, Hin,
                         ↪  i, Depth);

                    elsif (Cin(2*i+1)='1') then -- particle
                     ↪  struck FTH 1 on upper slice (ONE
                     ↪  SLICE ABOVE THE STRUCK SLICE OF THE
                     ↪  FWC 1)
                        Output(2*i+1) := '1'; --
                         ↪  Penetration depth is set to
                         ↪  display that particle hit
                         ↪  at least FWC 1 & 2 & FTH 1
                        Depth(0 to 1) := (0 to 1 =>
                         ↪  '1'); -- Penetration depth
                         ↪  is set to display that
                         ↪  particle hit at least FWC 1
                         ↪  & 2 & FTH 1
                        depth_shift_high(Ain, Bin, Cin,
                         ↪  Din, Ein, Fin, Gina, Hin,
                         ↪  i, Depth);

                    else -- Particle did not reach FTH 1
                        Output(2*i) := '1'; --
                         ↪  Positional Output is set
                        Depth(0 to 1) := (0 to 1 =>
                         ↪  '1'); -- Penetration depth
                         ↪  is set to display that
                         ↪  particle hit FWC 1 & 2
```

```
                end if;
            end if;
        elsif (0<i) and (i<(Speichen-1)) then    -- PARTICLE IS
         ↪  REGISTERED SOMEWHERE IN THE MIDDLE OF THE LAYER
            if (Ain(i)='1') and (Bin(i)='1') then -- The
             ↪  particle went straight through FWC 1&2 by
             ↪  striking the upper overlapping slice of the
             ↪  FWC 2
                nLEDR   <= '0'; -- RED LED is turned on
                LED <= counter;
                if (Cin(2*i)='1') then              --
                 ↪  The particle went straight through
                 ↪  FWC 1&2 and FTH 1
                    if (2*i<Speichen-1) then
                        Quarter(0) := '1';
                    else
                        Quarter(1) := '1';
                    end if;
                    Output(2*i) := '1';
                    depth_straight(Ain, Bin, Cin,
                     ↪  Din, Ein, Fin, Gina, Hin,
                     ↪  i, Depth);

                elsif (Cin(2*i-1)='1') then -- The
                 ↪  particle went through FWC 1&2 as
                 ↪  welll as FTH 1 but the struck slice
                 ↪  on the FTH 1 deviates +1 towards
                 ↪  the lower edge
                    if (2*i-1<Speichen-1) then
                        Quarter(0) := '1';
                    else
                        Quarter(1) := '1';
                    end if;
                    Output(2*i-1) := '1';
                    Depth(0 to 1) := (0 to 1 =>
                     ↪  '1');
                    depth_shift_low(Ain, Bin, Cin,
                     ↪  Din, Ein, Fin, Gina, Hin,
                     ↪  i, Depth);

                elsif (Cin(2*i+1)='1') then -- The
                 ↪  particle went through FWC 1&2 as
                 ↪  welll as FTH 1 but the struck slice
                 ↪  on the FTH 1 deviates +1 towards
                 ↪  the upper edge
                    if (2*i+1<Speichen-1) then
                        Quarter(0) := '1';
                    else
                        Quarter(1) := '1';
                    end if;
                    Output(2*i+1) := '1';
                    Depth(0 to 1) := (0 to 1 =>
                     ↪  '1');
                    depth_shift_high(Ain, Bin, Cin,
                     ↪  Din, Ein, Fin, Gina, Hin,
                     ↪  i, Depth);
```

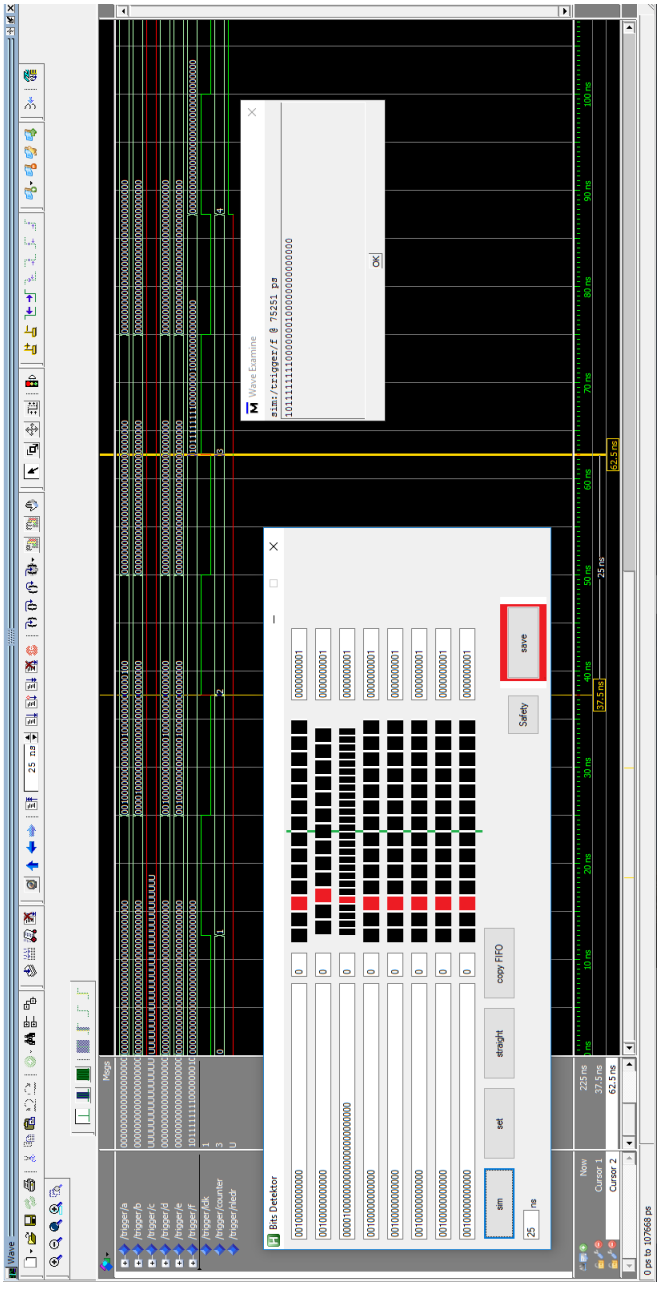
```
                else -- The particle went straight
                 ↪  through FWC 1&2 but did not reach
                 ↪  FTH 1
                        if (i<Speichen/2) then
                                Quarter(0) := '1';
                        else
                                Quarter(1) := '1';
                        end if;
                        Depth(0 to 1) := (0 to 1 =>
                         ↪  '1');
                end if;
        end if;
        if (Ain(i)='1') and (Bin(i-1)='1') then -- The
         ↪  particle went straight through FWC 1&2 by
         ↪  striking the lower overlapping slice of the
         ↪  FWC 2
                nLEDR   <= '0'; -- RED LED is turned on
                LED <= counter;
                if (Cin(2*i)='1') then
                 ↪
                 ↪  -- Evaulation process is equal to
                 ↪  the above-mentioned case only
                 ↪  shifted one slice down on the FWC 2
                 ↪  and FTH 1
                        if (2*i<Speichen-1) then
                                Quarter(0) := '1';
                        else
                                Quarter(1) := '1';
                        end if;
                        Output(2*i) := '1';
                        depth_straight(Ain, Bin, Cin,
                         ↪  Din, Ein, Fin, Gina, Hin,
                         ↪  i, Depth);

                elsif (Cin(2*i-1)='1') then -- see above
                        if (2*i-1<Speichen-1) then
                                Quarter(0) := '1';
                        else
                                Quarter(1) := '1';
                        end if;
                        Output(2*i-1) := '1';
                        Depth(0 to 1) := (0 to 1 =>
                         ↪  '1');
                        depth_straight(Ain, Bin, Cin,
                         ↪  Din, Ein, Fin, Gina, Hin,
                         ↪  i, Depth);

                elsif (Cin(2*i-2)='1') then -- see above
                        if (2*i-2<Speichen-1) then
                                Quarter(0) := '1';
                        else
                                Quarter(1) := '1';
                        end if;
                        Output(2*i-2) := '1';
                        Depth(0 to 1) := (0 to 1 =>
                         ↪  '1');
```

```
                                        depth_shift_low(Ain, Bin, Cin,
                                         ↪  Din, Ein, Fin, Gina, Hin,
                                         ↪  i, Depth);

                                    else
                                        if (i<Speichen/2) then
                                            Quarter(0) := '1';
                                        else
                                            Quarter(1) := '1';
                                        end if;
                                        Depth(0 to 1) := (0 to 1 =>
                                         ↪  '1');
                                    end if;
                                end if;
                            else
                             ↪  -- PARTICLE IS REGISTERED ON THE OTHER EDGE OF THE
                             ↪  LAYER
                                --upper(nLEDRV, LED, counter, Ain, Bin, Cin,
                                 ↪  Din, Ein, Fin, Gina, Hin, Quarter, Output,
                                 ↪  Depth);
                                if (Ain(i)='1') and (Bin(i-1)='1') then --
                                 ↪  Particle struck overlapping slices of the
                                 ↪  FWC 1&2 on the edge
                                    nLEDR   <= '0'; -- RED LED is turned on
                                    LED <= counter;
                                    Quarter(1) := '1';
                                    if (Cin(2*i-1)='1') then            --
                                     ↪  particle went straight through
                                     ↪  overlapping slices of FWC 1&2 and
                                     ↪  FTH 1
                                        Output(2*i-1) := '1';
                                        depth_straight(Ain, Bin, Cin,
                                         ↪  Din, Ein, Fin, Gina, Hin,
                                         ↪  i, Depth);

                                    elsif (Cin(2*i-2)='1') then
                                     ↪                          -- Struck
                                     ↪  slice of FTH 1 deviates -1 from the
                                     ↪  overlapping slices of FWC 1&2
                                        Depth(0 to 1) := (0 to 1 =>
                                         ↪  '1');
                                        Output(2*i-2) := '1';
                                        depth_shift_low(Ain, Bin, Cin,
                                         ↪  Din, Ein, Fin, Gina, Hin,
                                         ↪  i, Depth);

                                    else -- FTH 1 was not struck
                                        Output(2*i-1) := '1';
                                        Depth(0 to 1) := (0 to 1 =>
                                         ↪  '1');
                                    end if;
                                end if;
                            end if;
                        end loop;
                    end if;
                    if (Cin=(Cin'range => '0')) then                        --Depth from FTH
                     ↪  onwards is set to 0 if no particle was registered by the FTH
```

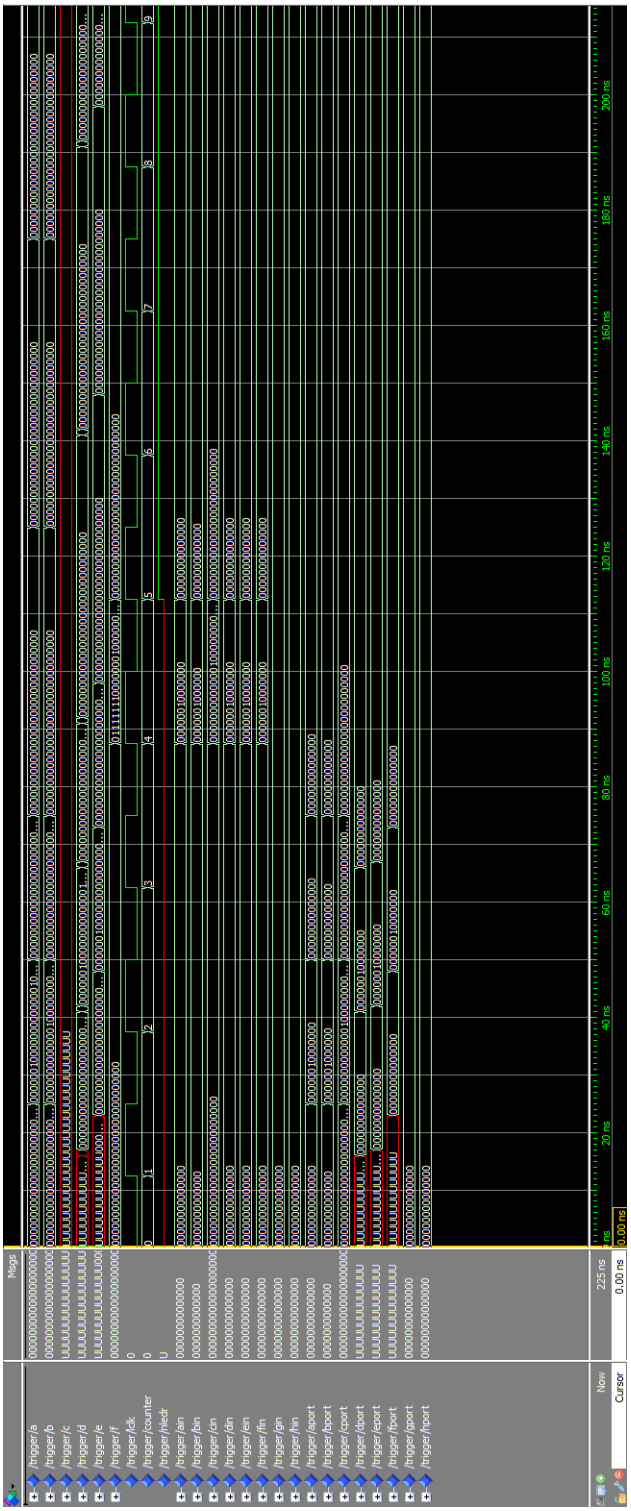
```
                Depth(2 to 7) := (2 to 7 => '0');
            elsif (Din=(Din'range => '0')) then                --Depth from the
             ↪  respective FRH onwards is set to 0 if no particle was regsitered by
             ↪  the respective FRH
                Depth(3 to 7) := (3 to 7 => '0');
            elsif (Ein=(Ein'range => '0')) then
                Depth(4 to 7) := (4 to 7 => '0');
            elsif (Fin=(Fin'range => '0')) then
                Depth(5 to 7) := (5 to 7 => '0');
            elsif (Gin=(Gin'range => '0')) then
                Depth(6 to 7) := (6 to 7 => '0');
            elsif (Hin=(Hin'range => '0')) then
                Depth(7) := '0';
            end if;
            counter := counter + 1;                                    -- counter
             ↪  increasesx with each clock pulse
            maximal_depth(Depth, MPD);       -- Maximal penetration depth is
             ↪  calculated
            F(0 to 17) <= Quarter&Depth&MPD; -- Output signal for FPGA Port is set
        end process;
end behavioral;
```

B. Screenshots from the simulations

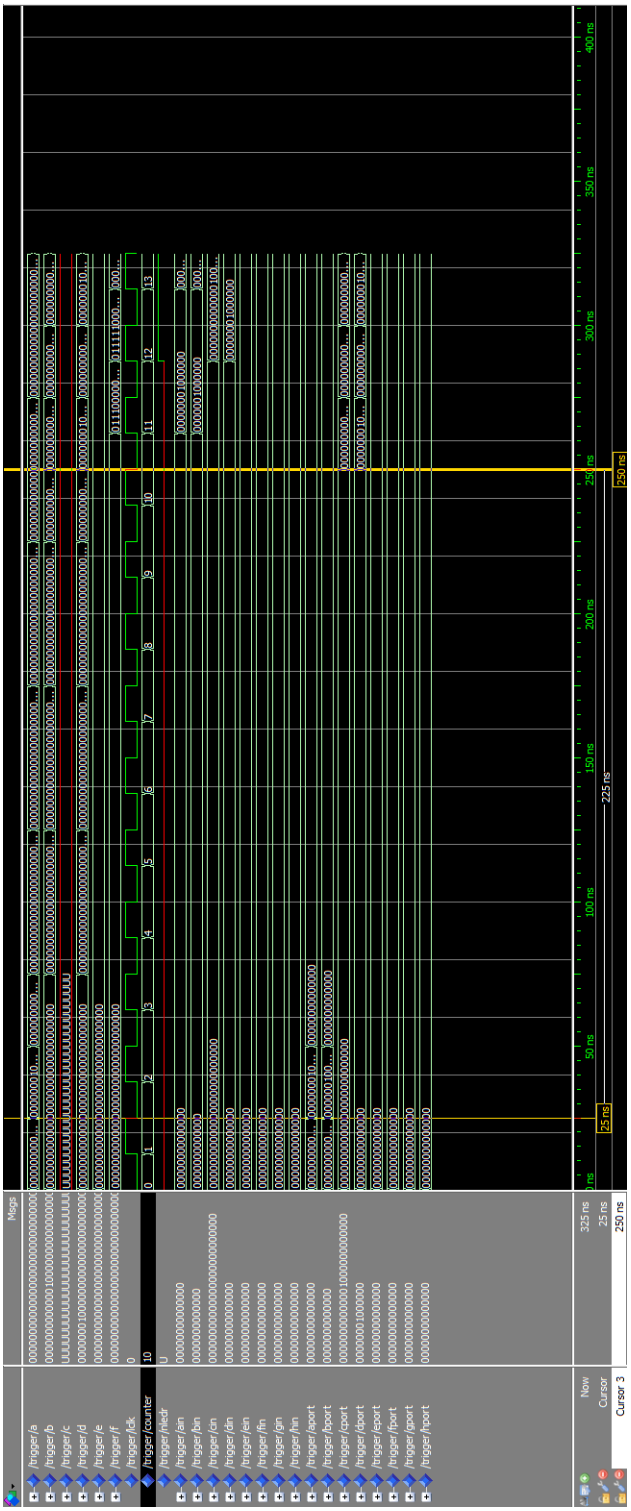
Simulation 1:



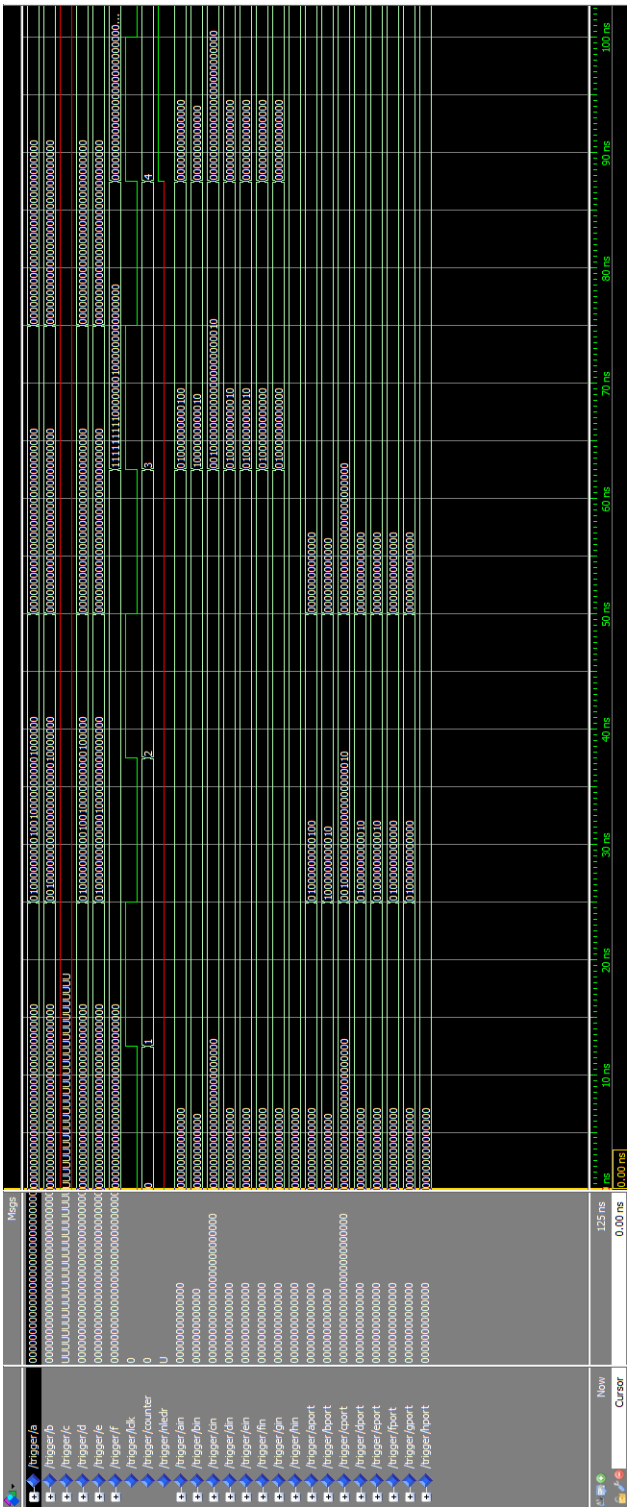
Simulation 2:



Simulation 3:



Simulation 4:



C. Symbols and constants

r	radius	m
t	time	s
m	mass	kg
v	velocity	m/s
q	charge	C
B	magnetic field	T
ω	angular frequency	rad/s
E_{kin}	kinetic energy	GeV/c
U	voltage	V
c	speed of light	299,792,458 m/s

D. Acknowledgments

I wish to express my sincere thanks to Prof. Dr. Jörg Pretz for giving me the opportunity to work on this thesis as well as for his support and advice throughout the entire project.

I would also like to show gratitude to Dr. Volker Hejny who gave me an understanding of COSY, the WASA detector and all necessary technical devices used in the course of this project. I also want to give thanks to him for his expert input when it comes to the features of the trigger system and all of his additional support.

I wish to thank Fabian Hinder for his many explanations of the FPGA and all of his help during this project as well as for reading and improving this thesis.

Furthermore, I would like to thank Fabian Trinkel for also reading and enhancing my thesis.

Last but not least, I wish to thank all the men and women who work at the IKP for their kindness and the good atmosphere. I really enjoyed working with every single one of you.

E. Statutory declaration

Henrik Matschat (Matrikelnummer: 330778)

I declare that I have authored this bachelor thesis independently and without any unauthorized support. I have not used any other than the declared sources and resources. In case that this thesis is also submitted electronically, I declare that the printed and the digital version are completely identical. This thesis has not been submitted to any department before.

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmt. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, 12. Juli 2016

_____

Henrik Matschat