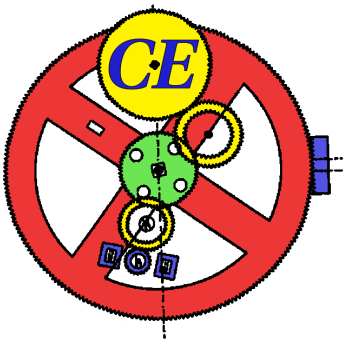

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Reconfigurable Trigger Logic for Electronic Instrumentation in Space Applications.

Mihai Lefter

CE-MS-2010-20

Abstract

Future space missions have to rely on advanced, smart and very light payloads in order to explore the solar system within a reasonable cost envelope. For this reason, efforts are made to obtain higher levels of integration that can reduce costs and allow the presence of more and more instruments on board of small spacecrafts. With the advent of radiation hardened FPGAs, the use of reprogrammable hardware in space is no longer an issue. This opens new perspectives to space electronics. System-on-Chip (SoC) design methodologies for future highly-integrated devices are actively promoted by space agencies. In this thesis we focus on FPGA based SoC architectures for space electronics instrumentation, targeting time related issues. In this line of reasoning we proposed and developed a highly customizable trigger logic block able to reject background events with the highest possible efficiency and to accurately timestamp the accepted ones. Its features include programmable coincidence window, input ordering, and for every input in particular the possibility to choose different states and to program the delay. The trigger logic block is designed as an AMBA IP core and it can be interfaced with many SoC libraries. For testing purposes we have programmed an AMBA based SoC architecture including a LEON3 on-chip processor and a minimal selection of IP cores from the GRLIB library on a Xilinx XC3S1500 FPGA. The trigger logic IP together with another IP developed for testing reasons were clocked at 100 MHz, while the rest of the system was running at 40 MHz. An average dead time of 1.5 μs was obtained, corresponding to an events frequency of 0.65 MHz. Based on our experimental results we can conclude that the proposed trigger logic approach can potentially successfully function in space applications. In extent to the trigger logic IP design, we have further performed research on the current SpaceWire time-codes, in an attempt to improve the inter-module time distribution accuracy. Several methods were proposed to reach synchronization in the order of nanoseconds, as opposed to the current microseconds synchronization, with little changes over the current SpaceWire standard.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Reconfigurable Trigger Logic for Electronic Instrumentation in Space Applications.

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Mihai Lefter
born in Brasov, Romania

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Reconfigurable Trigger Logic for Electronic Instrumentation in Space Applications.

by Mihai Lefter

Abstract

Future space missions have to rely on advanced, smart and very light payloads in order to explore the solar system within a reasonable cost envelope. For this reason, efforts are made to obtain higher levels of integration that can reduce costs and allow the presence of more and more instruments on board of small spacecrafts. With the advent of radiation hardened FPGAs, the use of reprogrammable hardware in space is no longer an issue. This opens new perspectives to space electronics. System-on-Chip (SoC) design methodologies for future highly-integrated devices are actively promoted by space agencies. In this thesis we focus on FPGA based SoC architectures for space electronics instrumentation, targeting time related issues. In this line of reasoning we proposed and developed a highly customizable trigger logic block able to reject background events with the highest possible efficiency and to accurately timestamp the accepted ones. Its features include programmable coincidence window, input ordering, and for every input in particular the possibility to choose different states and to program the delay. The trigger logic block is designed as an AMBA IP core and it can be interfaced with many SoC libraries. For testing purposes we have programmed an AMBA based SoC architecture including a LEON3 on-chip processor and a minimal selection of IP cores from the GRLIB library on a Xilinx XC3S1500 FPGA. The trigger logic IP together with another IP developed for testing reasons were clocked at 100 MHz, while the rest of the system was running at 40 MHz. An average dead time of 1.5 μs was obtained, corresponding to an events frequency of 0.65 MHz. Based on our experimental results we can conclude that the proposed trigger logic approach can potentially successfully function in space applications. In extent to the trigger logic IP design, we have further performed research on the current SpaceWire time-codes, in an attempt to improve the inter-module time distribution accuracy. Several methods were proposed to reach synchronization in the order of nanoseconds, as opposed to the current microseconds synchronization, with little changes over the current SpaceWire standard.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-20

Committee Members :

Advisor:	Sorin D. Cotofana, CE, TU Delft
Advisor:	Dimitris Lampridis, cosine research BV
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft

Member: Stephan Wong, CE, TU Delft

To my grandparents.

Contents

List of Figures

List of Tables

Acknowledgements

This thesis is the result of nine months of work that was carried out as partial fulfillment for the Master of Science in Computer Engineering. During all these months I have spent most of my time at cosine Research BV in Leiden. I consider this a fruitful experience since I could benefit from a lot of guidance and I learnt to look at problems from different angles.

I would like to thank in particular to Dimitris Lampridis, my on-site supervisor. Dimitris put a lot of time and effort into transmitting parts of his knowledge to me. He was always there to help me whenever I had a problem. Special thanks go to Chris who was a great partner for discussions, not always work related, but from which I have learned many things. Also I thank Scott, Eric and my other colleagues from cosine for creating a pleasant working atmosphere.

My gratitude goes to my professor and advisor Sorin Cotofana, for all the support he offered me, for granting me the opportunity to be involved in this project, for his patience and his valuable advice. He has a subtle way of helping me clear my thoughts every time I get lost in details.

Many thanks to Catalin, Bogdan, Vlad, Alex and Mottaqiallah for their help and friendship, as well as to many other friends whose names I will not list here, but who should know who they are.

I am where I am now because of the support, continued efforts and sacrifices of my parents, for which I am grateful.

A special place is reserved to Iulia, for sharing with me all the burdons and joys over the past years, and many to come.

Mihai Lefter
Delft, The Netherlands
July 4, 2010

1 Introduction

Space has always been considered the ultimate matter of research for humanity. Since ancient times scientists have observed the heavens and wondered about the nature of the objects seen in the sky. However it was not until last century that space was finally reached and space research entered a new era. Only after achieving spaceflights it was possible to understand many objects and phenomena which are better observed from a space perspective.

Current space exploration requires lighter payloads in an effort to reduce costs and at the same time to be able to include more and more instruments on-board of small spacecrafts [23]. The process of reducing a payload spans many domains, out of which electronics plays a major role. Systems-on-a-Chip (SoC) architectures are a key component in this context, since they allow very high integration levels required by modern space electronics.

SoC technology allows the integration of different components of a computer as well as other electronic devices into a single integrated circuit - a chip. This enables substantial reductions of power and area resources, which are crucial in space design. Generally speaking, SoC can be easily implemented on a Field Programmable Gate Array (FPGA) as well as manufactured directly into an Application Specific Integrated Circuit (ASIC).

In recent years FPGAs have been introduced to space applications and are now used on a large scale, especially for small space projects. Featuring high flexibility combined with high performance and complexity, FPGAs start nowadays to be preferred over ASICs. Todays main reason is their low cost, as the manufacturing price for an ASIC is still very high, especially to produce small quantities that are required in space applications. However there are already clear indications that their reprogrammability attribute will become the most important reason in the future. With the current satellites lifetimes reaching far beyond 10 years, and different software standards changing few times in this interval, being able to reprogram what is in space becomes a stringent issue.

Even though FPGAs have been flying in space for more than a decade, during all this time they were programmed only once, before being launched. The reason behind this is that in contrast to an ASIC, the configuration of an FPGA is stored in an SRAM, which is sensitive to Single Event Upsets (SEU). There is ongoing research aiming to improve FPGAs capabilities in high radiation environments in order to be able to reconfigure them in space. For more information on this topic we redirect the interested reader to [18], [17] and [13].

The work performed for this thesis addresses FPGA-based SoC architectures utilization in space electronics, with a main focus on time related issues. Section 1.1 gives an introduction and a motivation for the current work. In Section 1.2 the main research goals are presented, while Section 1.3 closes the chapter, giving an overview of

the organization of the remainder of this thesis.

1.1 Problem Statement

The amount of data generated during a space mission increases as the number of instruments present on-board of a spacecraft becomes larger. As this happens, it is difficult for the down to Earth link bandwidth to support the transfer of all this information. Hence there is a current need to perform on-line data analysis on-board and transfer only the meaningful information down to Earth, where it can be later on processed.

Time plays an essential role in data analysis techniques. When information from multiple detectors/instruments needs to be correlated, timing accuracy is crucial. Physicists need to know what events took place exactly at the same time as well as the time interval between certain events. One of the mostly utilized methods to distribute time across the spacecraft is based on a special mechanism, called time-codes. This is specific to the SpaceWire protocol, a communication standard which is worldwide accepted by space agencies, and assures at the moment a time resolution accuracy in the order of microseconds. However, this starts to be no longer sufficient for current experiments which require nanoseconds resolution accuracy to enable a better interpretation of the results.

Another aspect which is really important regarding space instruments is that from the total amount of data that are read, only a small part is actually relevant, the majority being only noise. Therefore, an important problem that must be solved is to increase the number of detected useful events per time unit, while suppressing background events. This should preferably happen in real-time. The solution is to detect the moment in time when information becomes relevant and to generate a trigger to signalize that valuable data can be recorded.

Trigger systems are uilized in many domains but one branch in which they are predominant and studies have been performed is high-energy physics (HEP). Many fast triggering/coincidence devices are used in HEP to perform preliminary selection and filtering of information. However, in contrast to HEP experiments performed on Earth, where almost unlimited resources are available, in space designers have to deal with many constraints, regarding materials, power, space, etc.

1.2 Thesis Objectives

From the context stated above, time proves to be a key element in experiments performed in space. The way in which the time accuracy problem is solved is very important in the current need to miniaturize space instruments. FPGA-based specialized trigger and data acquisition systems play a major role in this process.

In view of that the main thesis goals are:

- Design and develop a customizable AMBA based IP core to be used in real-time space data analysis. Based on the user configuration the core has to act as a trigger logic device able to reject background events with the highest possible efficiency. Additionally, it should also be able to accurately time-stamp the accepted events.

- Integrate the developed trigger IP core in an SoC architecture together with a LEON3 processor and other required IPs for testing purposes and study the instrument dead time.
- Study SpaceWire specific distributed time protocol synchronization based on time-codes and propose improvements in order to achieve a better accuracy.

1.3 Thesis Outline

The organization of the thesis is structured in the following manner. In Chapter 2 the relevant background theory related to space instruments is presented. In the same chapter some short information regarding related work performed in time-analysis techniques is also introduced. The main focus is on experiments which require trigger logic. In Chapter 3 the architecture and hardware implementation of the proposed AMBA peripheral IP is presented, as well as its functionality. Next, Chapter 4 focuses on the tests performed on the hardware implementation and the results that were obtained. Chapter 5 targets time accuracy across SpaceWire networks. In the first part the current time distribution is presented, while in the second part possible time improvements are introduced. The thesis ends with our conclusions, presented in Chapter 6, which also includes other final remarks related to lessons learnt, confronting problems, and future research directions.

2 Background & Related Work

This chapter can be regarded as a short presentation of the current space instrumentation, having as main target space electronics and data analysis techniques. However, we focus only on aspects that are related to the topic of the thesis.

In the first part FPGA based space instruments are explained. We state though that this should not be considered as the general case for all space instruments. Next we briefly describe how instruments are interconnected in a spacecraft via their SpaceWire interfaces. As an example of an FPGA based space instrument the Multifunctional Particle Spectrometer (MPS) is presented in detail.

In the final part of the chapter an introduction to data analysis techniques for high-energy physics is included, which addresses trigger logic systems. This can be seen as related work for this thesis.

2.1 FPGA Based Space Instruments

Instruments are the key components of almost all space missions. They monitor the environment in which the spacecraft travels, observe different celestial bodies and gather intelligence information. The data which are recorded by instruments are provided to scientists and analysts on the ground for further analysis.

A modern instrument consists of several detectors and the required read-out and data analysis electronics. These detectors are combined based on their resource needs, shared functionality, and mechanical considerations. All the components are wrapped in a mechanical enclosure and with the help of a certain interface the instrument is able to receive commands and to deliver measurement results.

A design example of a space instrument is presented in Figure 2.1. The sensors measure a physical quantity and convert it into an electrical signal. This signal can be already digitized, and used in this form by the rest of the electronic components, or it can be analogue. In this latter case it is required an analogue preprocessing chain that conditions and digitizes the signal. Special devices like an amplifier and an analog-to-digital converter (ADC) need to be present to perform this operation.

All the digital interfaces from the sensors are connected to a local FPGA. Here a number of modules able to control and readout the sensors are arranged in an SoC architecture with one or more soft CPUs and several specialized hardware blocks. There are several soft processor options available, e.g., LEON3, Microblaze, OpenRISC, etc., which can be fully implemented using logic synthesis. All the hardware blocks are interconnected together via a specific architecture bus, which is in close relation with the chosen processor.

Most of the FPGAs store their configuration data in SRAM memories. For this reason they require some external boot devices. There are also SRAM-based FPGAs

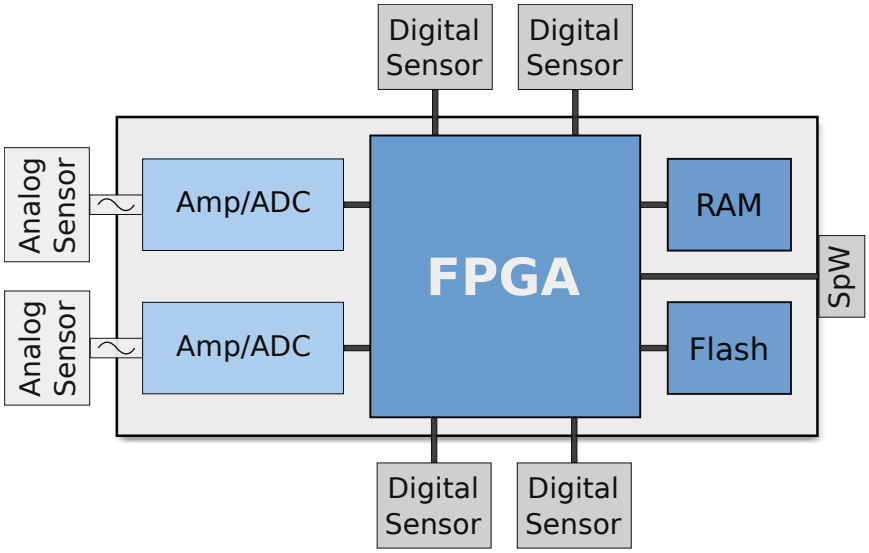

Figure 2.1: Example design of a space instrument including sensors and electronics

with integrated flash. In this case their configuration is stored in the internal flash memory and when they are powered-up the SRAM is configured. Some other type of FPGAs are based on Flash memory only and do not contain any configuration SRAM. Such an example is the ProASIC3 family from Actel.

Different embedded software can be run on the soft CPU to control the behavior of the instrument. Depending on the application requirements a separation has to be done between the part of the algorithm running on the soft CPU and the part implemented in specialized hardware. Finding the most appropriate partition is subject of a design space exploration (DSE) process, which in general can be rather complex and tedious. The utilization of SoC architectures including reconfigurable hardware ease the DSE as it offers great flexibility to the designer.

An important subcomponent of the instrument is the SpaceWire interface port. Within a spacecraft several instruments can be interconnected via their SpaceWire interfaces, forming a SpaceWire network of devices. Figure 2.2 presents a view of the possible top-level architecture of the spacecraft. Different network components as SpW routers can be also present. More detailed information regarding SpaceWire are introduced in Chapter 5.

The core of the top-level architecture is the main digital processing unit (DPU) through which all the instruments are controlled. The collected data are correlated by the DPU and the significant information is stored to be later processed or transmitted to the ground, if necessary. Also the DPU can alert the other spacecraft components in case of emergency, based on the information it receives from certain instruments.

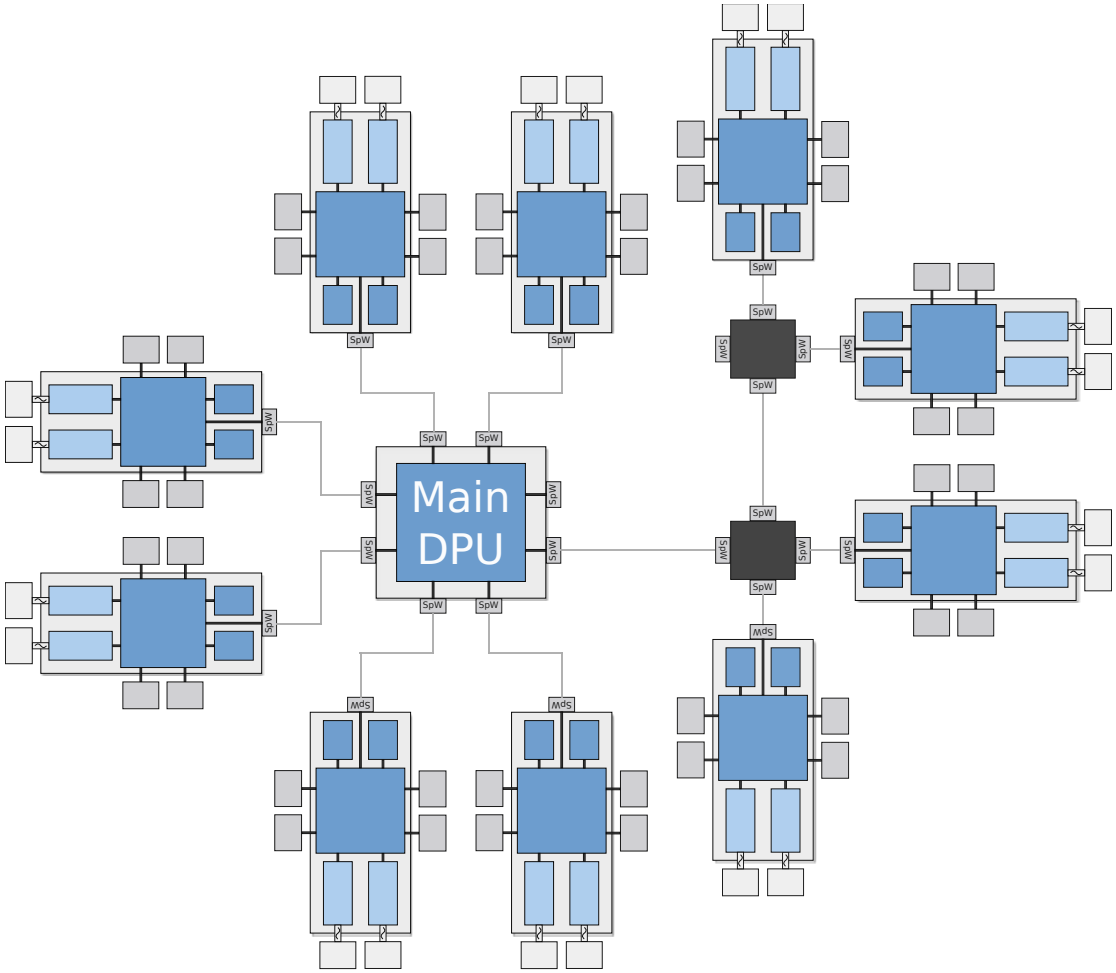

Figure 2.2: Example of a top-level space architecture including multiple instruments

2.2 The Multifunctional Particle Spectrometer

The Multifunctional Particle Spectrometer (MPS) is a miniaturized energetic particle spectrometer and a general radiation monitor. It can be utilized for planetary exploration, as well as for small satellites. It can discriminate between four types of particles and at the same time it can determine the incident angle. The MPS provides scientific information and acts also as a spacecraft protection system.

The base line design of the MPS, depicted in Figure 2.3, consists of a tracker and a scintillator. The tracker has two 40x40 mm^2 silicon pixel sensors with a thickness of about 300 μm. The distance between the two sensors is 10 mm and the angle of incidence for charged particles which traverse both sensors can be reconstructed with an accuracy better then 10°.

The number of silicon pixel sensors that are present in the tracker can vary from experiment to experiment, but it is important to be at least two in order to reconstruct the incidence angle. This operation is performed as follows. Every layer is composed of an

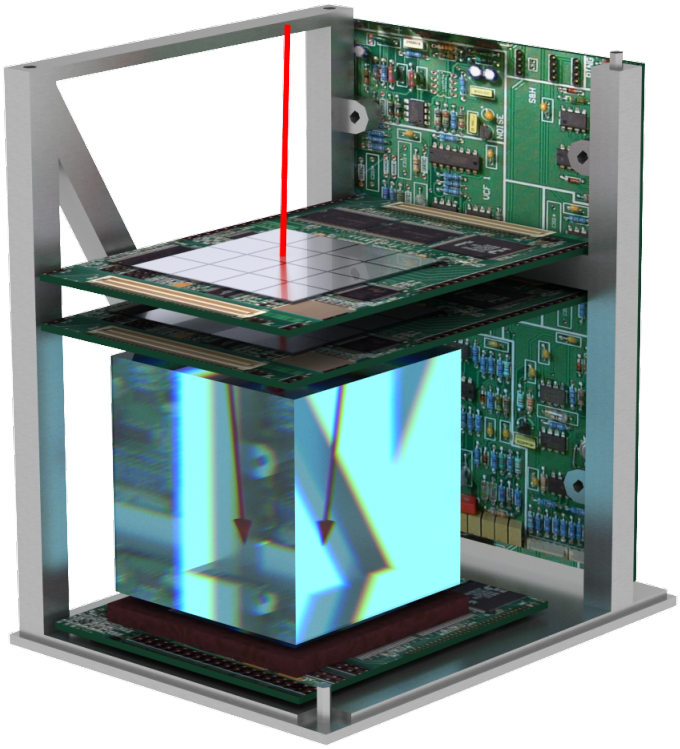

Figure 2.3: Base line design of the Multifunctional Particle Spectrometer [2]

8x8 array of sensitive diodes. When the particle crosses the first layer, the corresponding indexes in the array are determined. The same happens in the next layer. These indexes form the coordinates based on which the direction of the particle can be determined.

The scintillator is an inorganic alkali halide CsI(TI) crystal with the dimensions 34x34x34 mm^3, being optically coupled to a large area photo-diode. The MPS provides identification of electrons, protons, ions, and gammas by the ΔE versus E method [29], where the former value is obtained by the tracker and the latter from the scintillator.

The read out system of the MPS is a combination between an FPGA based SoC architecture and multi-channel ASICs. On the FPGA custom hardware blocks together with off-the-shelf IP cores and embedded software compose the digital processing system. For the scintillator a custom IP core implements a digital pulse height detection system based on trapezoidal filtering [24].

More detailed information regarding the MPS can be found in: [27], [2].

2.3 Trigger Logic Systems

Experiments performed in HEP are different from those in other areas in that detectors need to read data event by event, measurements being essentially instantaneous. These

events involve particles and physical interactions that researchers are studying, being of great interest when they occur.

Due to the large amount of information, out of which only a small part is really important, experiments are selective in what they read out. And once such an event is selected for read out the detector enters in a so called busy time, or dead time. For this reason, the way in which events are chosen is decisive so that good events are not lost. Instruments need to have an indicator of the correct time to read out, typically with a precision in the order of nanoseconds. Triggers were introduced as a mechanism to solve these problems.

Trigger systems are used at all levels of HEP experiments and are designed together with detector systems in order to match the requirements of the experiment. To reduce dead time and to be able to include complex decisions, modern trigger systems are typically constructed in a hierarchy as suggested in Figure 2.4. Lower trigger levels reject events based on obvious and simple criteria, while higher levels of trigger implement more complex analysis as they have more time available for processing. Higher levels cannot undo a reject of a lower level as they operate only on a subset of events selected by the lower levels [26] [16].

Different electronic components are found at each level of a trigger system. Nevertheless this varies from experiment to experiment. At the down-most level only simple logical operations are possible by modules that are usually built according to the Nuclear Instrumental Module standard (NIM), developed especially for particle and nuclear physics experiments. More complex operations require a higher level of integration and can be implemented by using integrated circuits as FPGAs, PALs, RAMS, CAMs, etc. At higher levels Digital Signal Processors (DSPs) are found and at the very end the readout chain data enter a computer via electronic buses (VME, PCI) or via networks (SCI, ATM).

One of the greatest experiments performed at the moment takes place at CERN (European Organization for Nuclear Research) in Switzerland. The Large Hadron Collider (LHC) is the world's largest and highest-energy particle accelerator [9]. At the LHC four major detectors will be operating. Two of them, ATLAS and CMS, are general purpose particle detectors designed to investigate the largest range of physics possible. The other two are medium-size experiments, ALICE and LHCb, which have specialized detectors for analyzing the LHC collisions in relation to specific phenomena.

All LHC experiments use multi-level trigger systems. The low-level trigger is implemented using fast electronics while the high-level trigger system is implemented in software running on large computer farms. More information on trigger systems at LHC: [20], [28] and [11].

2.3.1 Integration of FPGAs in Trigger Logic Systems

FPGA technology started to be utilized in trigger systems in the 1990s. At that time FPGA circuits had a very low logic capacity at relatively high costs, and were mostly embedded in trigger modules where a dynamic reconfiguration of algorithms was needed. Lately, the developmental prospects of FPGAs suggested the feasibility of replacing ASIC circuits with FPGAs. This major trend concerned the majority of HEP experiments

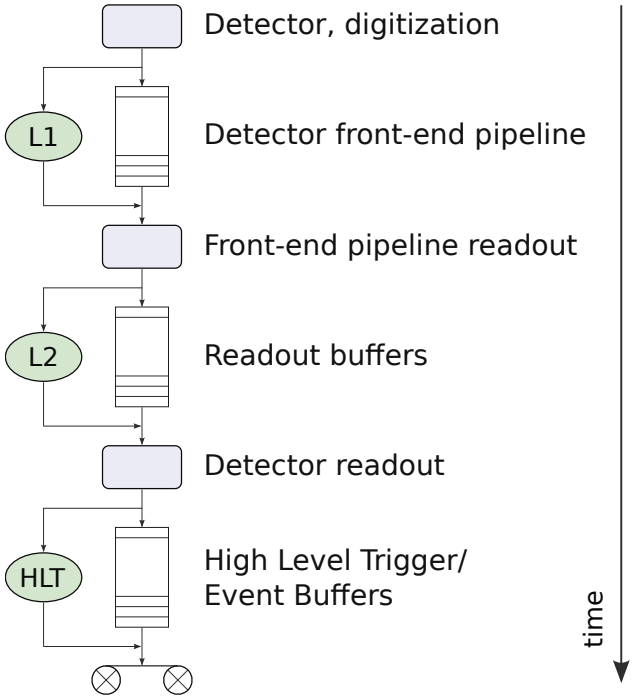

Figure 2.4: A generic trigger hierarchy [26]

(LHC, TEVATRON [4], HERA [8], BELLE [1]) and is an ongoing process [31].

In [21] a programmable trigger logic module (TRILOMO) implemented in FPGA is presented. A modified version of it is used in the NA58 (COMPASS) experiment at CERN. It accepts 100 MHz input rates while up to 16 trigger input signals can be combined logically for fast trigger decision. A single slot 6U VME has been designed and assembled having a Spartan II FPGA as a central element for trigger logic and NIM inputs/outputs. The functionality of the FPGA can be modified via the VME bus with the help of a CPLD which realizes the access. Within the VME architecture the board acts as a VME slave module.

Another akin example is the trigger module developed for the BELLE experiment [22]. The board accommodates 144 input signals and provides 24 output signals. All the signals are obtained from an electro-magnetic calorimeter. It functions as a 9U VME module through which the FPGA can be reprogrammed. The trigger latency obtained is 50 ns.

Similar approaches, where on the FPGA only simple logic operations are performed, are popular in almost all experiments as part of the low trigger level systems. However more complicated trigger decisions can also be taken directly at the low level. An example is [19], where a dedicated trigger generation processor was developed for the U-70 accelerator at IHEP (Institute for High Energy Physics). The experiment requires that arithmetic operations have to be performed, based on which the trigger decision is taken. The trigger processor is implemented on an Altera FPGA, has 64 channel inputs and the trigger generation time is 150 ns.

2.4 Conclusion

We have presented so far some background information related to the topic of the thesis. First we argued that space research requires data analysis to be performed in the same manner as in HEP experiments on the ground, where information from several detectors is analyzed and trigger logic systems are used to select good events and suppress those reactions which are not interesting.

Subsequently we demonstrated that FPGAs have been introduced in many applications and domains ([25]), as well as in space and HEP experiments and presented some information regarding trigger systems, closely linked to the main goal of this thesis.

This concludes our discussion on the background knowledge and related work. In the next chapter we introduce our trigger logic IP core and present its implementation in detail. To the best of our knowledge, our trigger IP proposed is unique in that it proposes a single-chip solution, based on a SoC architecture approach. It is highly reconfigurable and it suits different space applications as it can be easily connected to many of the available soft processors through its AMBA bus interface.

A Reconfigurable IP Core for SoC Trigger Logic Systems

3

This chapter focuses on the main goal of this thesis, presenting the IP core that was developed. We begin with a description of the general trigger system in which the block will be included, as well as its role there. In the next section all the possible functionalities of the block are introduced. Based on that the actual hardware implementation took place, which is presented in the following section. In the last part of the chapter we describe the action flow that is required to design and to use the entire trigger system.

3.1 General Description

In Figure 2.1 an FPGA based space instrument, with several detectors, some readout electronics, and a SoC architecture present inside the FPGA, is depicted. In Figure 3.1 we recall only the trigger logic part, including also the proposed IP block. For every detector the flow is more or less the same: the data are read, digitized (if in analogue) and a first-level arithmetic trigger is assigned. This can be perceived also as a low trigger level, for every detector a tick being produced whenever an event is read. It can be implemented for example as a filter with a certain threshold above which the event becomes valid.

All the first-level triggers provide inputs to the Coincidence Block (CB). This custom IP acts as a high level trigger and, based on the configuration that is set by the user, it generates a final trigger of the system. This signal is produced only if the first-level trigger signals are being received according to the specified rules, and within a certain time interval, called coincidence time window. A time-stamping of the events also takes place, as the block records the time when these conditions are met and the trigger assigned.

The rules based on which the final trigger occurs are derived from the user configuration which can specify for example the length of the coincidence window, from what trigger to expect valid inputs, the operation mode of the system, etc. All this functional information is presented in detail in the next section. We note here that throughout this thesis we also refer to the final trigger of the system as 'tick_out', and to the first-level triggers as sensor triggers.

3.2 Functional Overview

3.2.1 User Configuration

The following section describes the customizable elements of the CB. As already stated before, the way in which the user configures the CB has a direct impact on the 'tick_out' generation rules, so a clear understanding of this information is mandatory. Some of

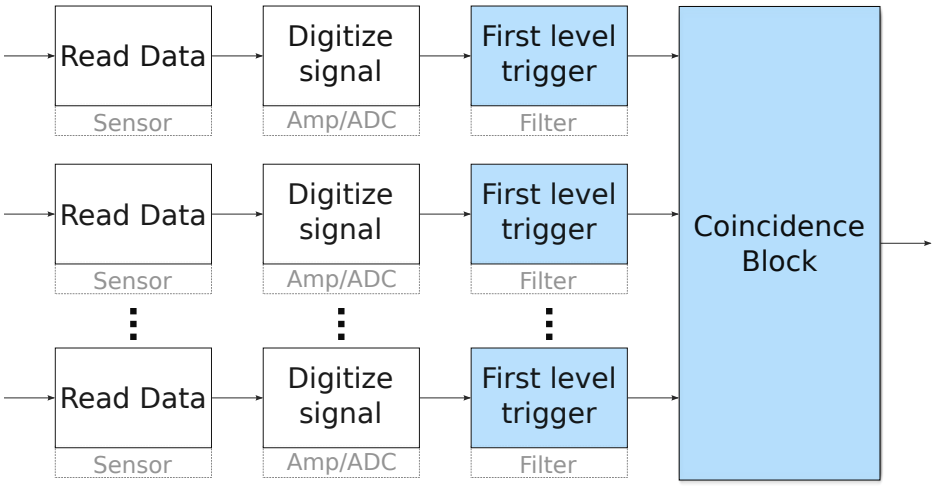

Figure 3.1: Trigger System

the configuration actions can be performed at runtime while certain elements need to be configured before the component is synthesized. This delimitation is specified later.

3.2.1.1 Coincidence window

Coincidences are of a high importance in HEP as they provide essential experimental information. The coincidence window is the time interval in which the rules that were set have to be fulfilled. It starts with a certain tick received from one of the first level triggers, as specified by the user, and it ends either when its time expires or if the final trigger is asserted.

The user is able to program the coincidence window of the CB in steps equal to the clock period. If we use for its representation a total of 8 bits, 256 possible values can be selected. If the clock frequency of the CB is 100 MHz the minimum value of the coincidence window is 10 ns and the maximum 2.56 μs. In order to ensure that the final trigger is assigned correctly, and that measurements are done simultaneously, the coincidence window has to be selected properly.

3.2.1.2 Sensor states

By the state of the sensor we refer to as whether we wait for the first-level trigger component associated with the sensor to actually trigger or not during the coincidence window. This is equivalent to its internal generated signal being in a predefined value (for example '1' or '0' logic), associated with the trigger. This value is considered every clock cycle, so if it is '1' for two consecutive cycles, and '1' equals to a trigger, then it means that two triggers were generated for two separate valid events, one after the other.

The way in which sensors should behave can be programmed and the user can place them in one of the following states:

- **Coincidence** - the sensor should trigger only once during a coincidence time window.

- **Anti-coincidence** - the sensor should not trigger at all during a coincidence time window.

- **Coincidence-check** - the sensor does not affect the coincidence window nor the generation of the tick_out signal, but the user is interested to see whether the sensor has triggered or not in a valid coincidence window.

A total of 2 bits are necessary for every sensor in order to be able to specify its state. One bit is required to specify whether the trigger is active high or low.

3.2.1.3 Sensor ordering

The main operation mode of the CB is determined by the order in which first level triggers are considered. Together with the state of each sensor, the sensor ordering influences the start of the coincidence window. The need to introduce an order for the first level triggers occurrence appeared as for certain experiments it is important to discard coincidence windows in which certain sensors trigger before other.

Regarding sensor ordering and starting/resetting of the coincidence window time counter, the user can choose between one of the following operation modes:

1. **No ordering (NO)** at all, and the coincidence window starts when a trigger is received from any of the sensors that were put in the coincidence state. Multiple sensor triggers are also allowed. The coincidence window ends after its time expires and it is reset when a sensor that had already triggered, triggers again.

2. **No ordering with starting sensor (NOWSS)** - similar to the first operation mode, except that the coincidence window starts when a specific sensor triggers. The sensor that starts the time counting for the coincidence window has to be put into the coincidence state. Several sensors can trigger together with the starting sensor, but they should also be in the 'coincidence' state. The coincidence window ends after its time expires or if a sensor that had already triggered triggers again and the sensor is not the starting one. The coincidence window is reset when the starting sensor triggers again.

3. **Specific order (SO)**, in which the user has to program for each sensor the position in which it is considered valid. The sensors have to trigger in the specified order, but consecutive sensors can trigger together if the current expected sensor to trigger is among them. The sensor on the first position is the starting sensor for the coincidence window. The coincidence window is reset when the starting sensor triggers again. It ends when one of the sensors that have already triggered in the current coincidence window triggers again, except for the starting one, when an out of order sensor triggers, or when its time ends.

In the above description we focused only on those sensors that were considered set by the user in the coincidence state. We have to add to the mentioned rules the fact that the anti-coincidence sensors can end the coincidence window at any time if one of them

Table 3.1: Start/reset/end of coincidence window based on operation mode and sensor state

	No Ordering	**No ordering with starting sensor**	**Specific Order**
Start	- When one or multiple sensors trigger, irrespective to any order.	- When the specified starting sensor triggers. Other sensors can trigger as well together with the starting one.	- When the sensor placed on the first position triggers. Consecutive sensors including the starting one can trigger as well.
Reset	- When a coincidence sensor that already triggered triggers again.	- When the specified starting sensor triggers again.	- When the sensor placed on the first position triggers again.
End	- When its time expires; - When an anti-coincidence sensor triggers.	- When its time expires; - When a coincidence sensor that already triggered triggers again (but it is not the starting sensor); - When an anti-coincidence sensor triggers.	- When its time expires; - When a sensor that already triggered triggers again (but it is not the starting sensor); - When an out of order sensor triggers; - When an anti-coincidence sensor triggers.

triggers. A sensor put in the anti-coincidence state does not affect the start or reset of the coincidence window.

Table 3.1 summarizes the rules based on which the coincidence window starts/resets/ends according to the order specified by the user. We can see that the coincidence window can end normally, when its time expires, or can be interrupted, when certain rules are broken. In the following we refer to the first type as full coincidence window and to the second type as interrupted coincidence window.

To distinguish between the different operation modes 2 bits are required. In addition to that, for the NOWSS operation mode a number of $log_2(\#sensors)$ bits are required to specify the starting sensor, and for the SO operation mode a total of $log_2(\#sensors)$ bits are required for each sensor.

3.2.1.4 Trigger delay appearance

In some experiments it is known for sure that some triggers can be received with a certain delay. To cope with that we introduced the possibility to specify certain delays for every sensor in particular. As the trigger of the delayed sensor can not be sent back in time, the triggers of the other sensors need to be moved forward. In case that there are more delayed sensors with different values, some calculations have to be performed by the user to compute the delay of every sensor.

A single bit is used to specify if a delay should be considered or not. We have chosen the same number of bits (8 bits) as for the coincidence window to represent the delay. In total 9 bits are required for every sensor.

3.2.2 Generation of 'tick_out'

As we have presented in the previous subsection the user configuration of the CB, it is possible now to derive the conditions based on which the final trigger ('tick_out') is asserted.

A tick_out signal is generated when all the sensors behaved according to their state and specified order within a coincidence window. In case of the coincidence sensors, it is required that their associated trigger value is active only for one clock cycle. If all the required sensors have triggered in the specified order, the tick_out signal is immediately asserted in the next cycle after the conditions were fulfilled, and the coincidence window ends, even if its time did not expire. A coincidence window is considered valid if a 'tick_out' signal is generated during its time interval.

In Figure 3.2 a 'tick_out' generation example is presented. There are six sensors: s1, s3, s4, and s6 are put on the coincidence state, s2 is put on anti-coincidence and s5 on don't care. The coincidence window is set to 6 cycles (equivalent to 60 ns for this example), and the sensor ordering is specific order (for simplicity being from s1 to s6).

When a trigger from s1 is received, the coincidence window is started. Next trigger that is received is from s5, in the second cycle, which is out of the order but as its state is don't care, it does not influence the coincidence window. We have no event in the third cycle. In the fourth cycle of the coincidence window, both s3 and s4 trigger. This is according to the rules, as s3 is the next trigger expected, and s4 is allowed to trigger at the same time. In the fifth cycle the remaining expected trigger appears, from s6, which signalizes that the final trigger can be generated. This happens in the next cycle. We can notice that s2 did not trigger at all during the coincidence window. As s2 is put in the anti-coincidence state, a trigger from it would have ended the coincidence window.

3.2.3 Pre/post-synthesis Configuration

Pre-synthesis configuration is performed by the use of VHDL generics/constants. One thing that the user is able to specify in this way is the number of sensor triggers that the CB component has as inputs. As this value is in direct relation with other user configuration parameters the actual $log_2(\#sensors)$ value has to be specified as well by the use of generics/constants. For the case we considered the minimum number of sensors is 2 and the maximum is 32.

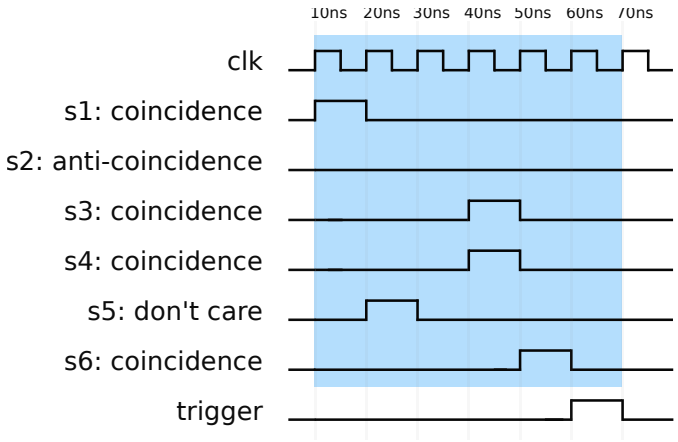

Figure 3.2: Trigger Example

Also by the use of generics/constants, the user can indicate the number of bits used to represent the following configuration parameters:

- Coincidence window - minimum value 8, maximum 16;
- Delay appearance - equal value as for the coincidence window;
- Local time counter - maximum value of 32.

A change of a generic/constant value requires that the component is re-synthesized.

Post-synthesis configuration refers to the real-time configuration, in which the user programs the parameters with their required value. The range is determined by the number of bits specified using generics. The following information can be specified at run-time:

- The value of the coincidence window;
- The state of each sensor;
- The sensor ordering - including the operation mode, the starting sensor and the position for each sensor;
- The delay appearance for each sensor.

3.2.4 I/O specification

The CB is implemented as an AMBA slave. The entire user configuration is performed exclusively through the AMBA bus, together with the output delivery of the block, leaving just the sensor trigger inputs to be connected outside the bus.

Inputs:

- Specific AMBA input signals - used to configure the CB. The user configuration information is stored in registers that can be accessed through the bus at certain addresses. One register is used to control the CB. When this register indicates the activation of the component, a local counter is start to track of the time.
- Sensor triggers - 1 bit each, connected directly from the first-level trigger IP cores. Their logic value should be kept on their associated trigger value an amount of time that is greater or equal to twice the period of the CB clock.

Outputs:

- Specific AMBA output signals.
  - The tick_out signal, when asserted, will set a bit from a specific register to logic '1'.
  - The information regarding the way in which sensors behaved can be read from an output register. A value of '1' on the i^{th} bit of the register means that the i^{th} sensor from the order has triggered. A value of 0 signifies that the i^{th} sensor did not trigger.
  - The value of the local time counter is available also in an output register.
- Hold signals, used to pus the sensors on hold until their value is read.

3.3 Hardware Implementation

This section presents in detail the hardware implementation of the CB. A general overview of the SoC architecture implemented on the FPGA is depicted in Figure 3.3, where the CB is included together with the first level triggers, being connected to the AMBA bus. We notice that the first level triggers are directly connected to the CB, outside the bus. At the same time the CB hold signals are also connected outside the bus to the first level triggers.

The CB can be integrated in a variety of SoC architectures, not only GRLIB based as in the figure. However the representation is similar, and includes a soft processor and certain architecture specific hardware blocks. We do not go into details regarding the entire GRLIB architecture in this section, as it is presented in detail in the next chapter. Here we only describe the implementation of the CB in isolation from the rest of the architecture.

3.3.1 Architectural Overview

A structural overview of the CB is depicted in Figure 3.4. We can see that there are two main paths that split the CB in two main components. One is the ambapath, through which the direct connection to the AMBA bus is assured. In this way the component can be configured and controlled. A set of memory mapped registers can be accessed

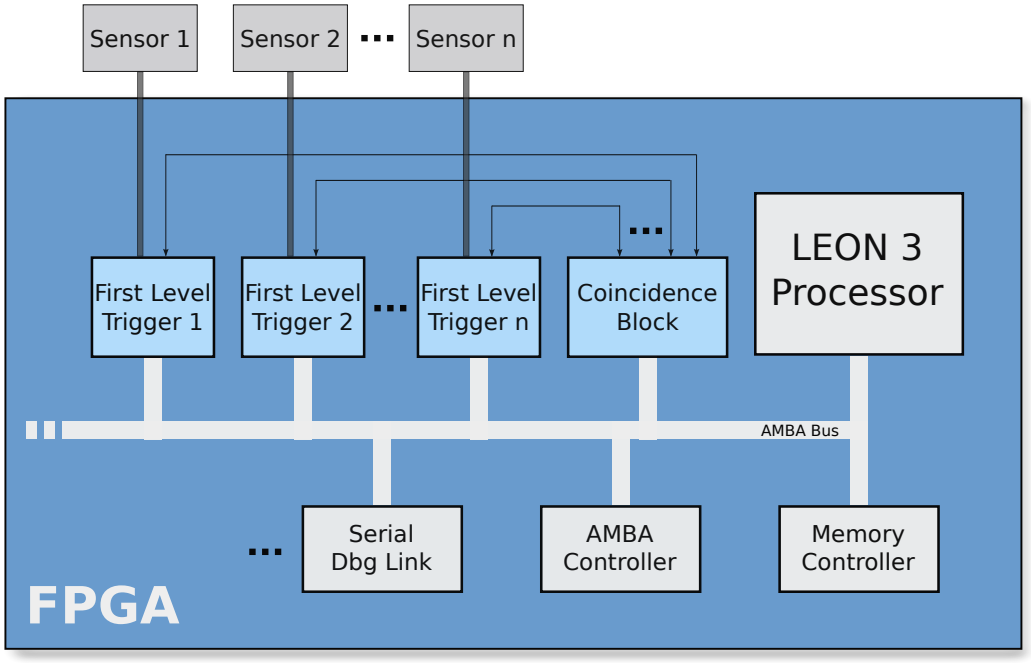

Figure 3.3: SoC block diagram

over the bus by the embedded software that runs on the processor. The clock driven signal of this component works at the same frequency with the one of the AMBA bus.

The second component - coincidencepath - has as external inputs the triggers received from the sensors. Based on the control information received from the first component, and on the trigger values, it generates the tick_out signal. As this happens the ambapath is announced to signal that to the processor. As a higher processing speed is required the second component works at a different clock frequency.

The chosen design implementation allows a total separation between computation and communication. It is easier in this way to integrate the coincidencepath with another bus protocol that might be needed by a different architecture, or even to use it alone outside any SoC design, if the required user configuration is assured.

3.3.2 AMBA Control Component

The main role of the this component (ambapath) is to form the communication link between the computational part (coincidencepath) and the software control application which runs on the on-chip processor.

A set of 32 bit memory-mapped registers are accessed over the AMBA bus. The read and write operations are being facilitated by the AMBA Wrapper, which decodes the specific I/O bus signals. Currently the AMBA Wrapper is designed to communicate over the AMBA Advanced Peripheral Bus (APB). The component can be adapted to other types of AMBA protocol buses by performing changes only at the AMBA Wrapper.

The registers are used to store the user configuration, introduced in the previous section, and to control the component. One register is used to store the operation mode

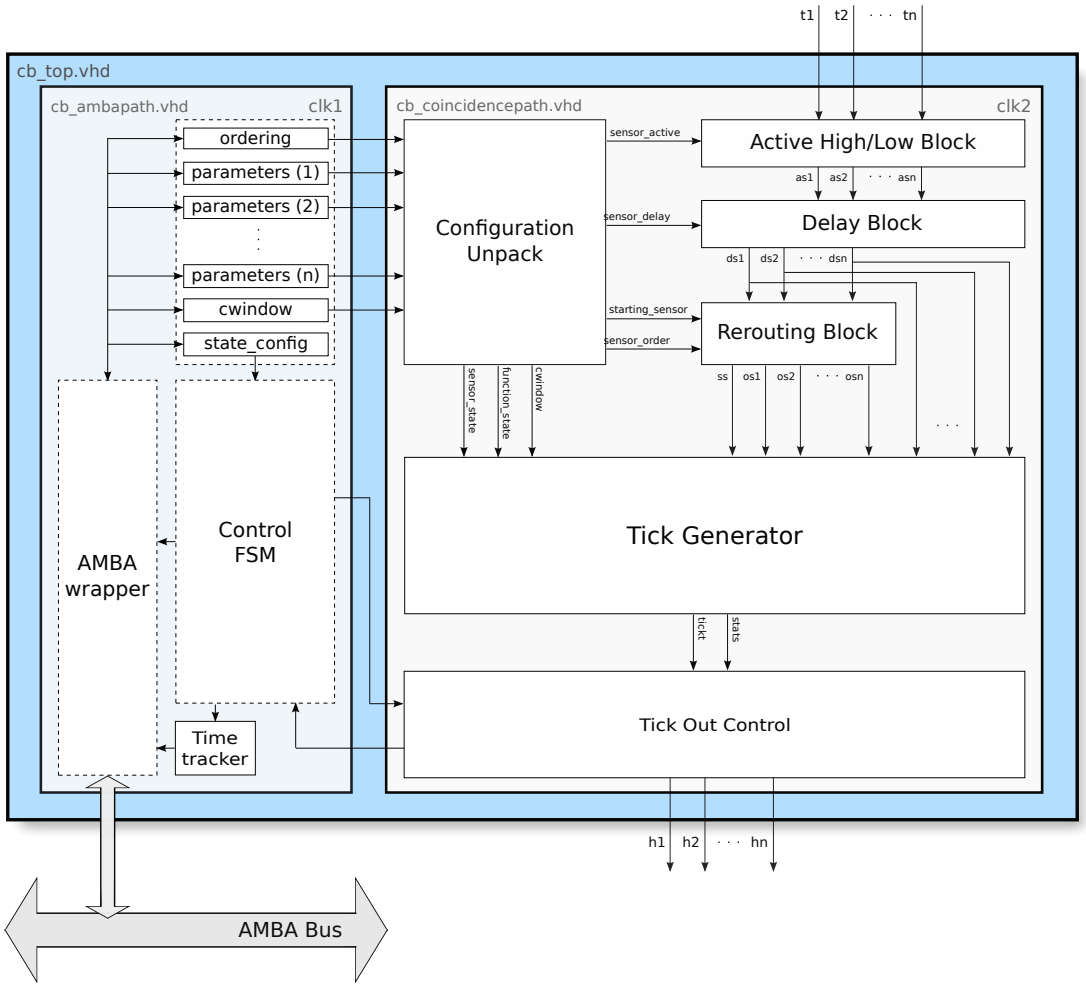

Figure 3.4: Structural overview of CB

together with the position of the starting sensor. Next, for every sensor in particular there is a register that holds its state, position order, delay, and active type. The value of the coincidence window is stored in a separate register. In the end there is another register used to control the component and to signalize that a trigger has been generated. In Appendix A the structure of all registers is explained in detail.

On the component there is also a local FSM included, which is in charge of the entire local operation. It is driven by the control register. When it receives the start operation it goes into the running state, where it waits for a trigger to be generated. If this happens it informs the processor and waits for its response to restart the trigger generation.

The last block present inside the ambapath is the Time Tracker, a local counter used to retrieve the time when triggers are generated. Its value represents the number of clock cycles that passed since it was reset, which is the moment when the FSM went into the running state.

3.3.3 Computational Component

The computational component (coincidencepath) generates the 'tick_out' signal based on the specific rules that are set by the user through configuration. The process begins by unpacking this configuration information, stored in the registers, and creating internal logic signals. This is operation is performed in the Configuration Unpack block.

The preprocessing operation can start now. First the triggers received from the sensors that are active low are reversed by the Active High/Low block. Next, the resulted local signals are delayed one by one with the number of cycles specified in the configuration. This operation takes place in the Delay block. The last preprocessing action takes place in the Rerouting block, where the triggers are put on their specified position. Here also the local starting sensor signal is selected from the triggers.

As a result of all these operations the Tick Generator receives as inputs the original ordered delayed triggers, the delayed rerouted triggers, and the starting trigger. All these signals are required by the three main operation modes of the component.

3.3.3.1 Tick Generator Block

The Tick Generator is in charge of detecting valid coincidence windows and generating the tick out signal. The algorithm behind the Tick Generator was derived from table 3.1. For every operation mode in which the coincidence block can work (NO, NOWSS and SO) there will be three signals that will mark the start, reset or end of the coincidence window.

The value of these signals is based on the input received from the sensors together with the rules that the inputs have to obey for every operation mode in particular. As a result of these rules we will have different implementations for the start/reset/end signals for every operation mode.

The structural view of the Tick Generator can be seen in Figure 3.5. Based on the states in which the sensors were configured certain masks are derived, action performed in the Masks Generator sub-block. These masks together with the sensor triggers are used in the Local Signals Generator sub-block to compute the required input signals for the NO/NOWSS/SO Signals Generators. Here the start/reset/end signals are computed for every operation mode separate.

A local state machine is present to control the whole process and to determine when a valid coincidence window occurs. The start/reset/end signals are multiplexed as reset input to a local counter, based on the operation mode in which the system is in. If all the required sensor triggered and the output value of the counter is less or equal to the coincidence window, then the 'tick_out' signal is generated. The FSM also initializes and updates the reset masks required for the NO/NOWSS/SO Signals Generators.

Together with the final trigger signal, a status which indicates the sensors from which a trigger was received is also outputted.

In the following we will present how the start/reset/end signals are generated for every operation mode in particular.

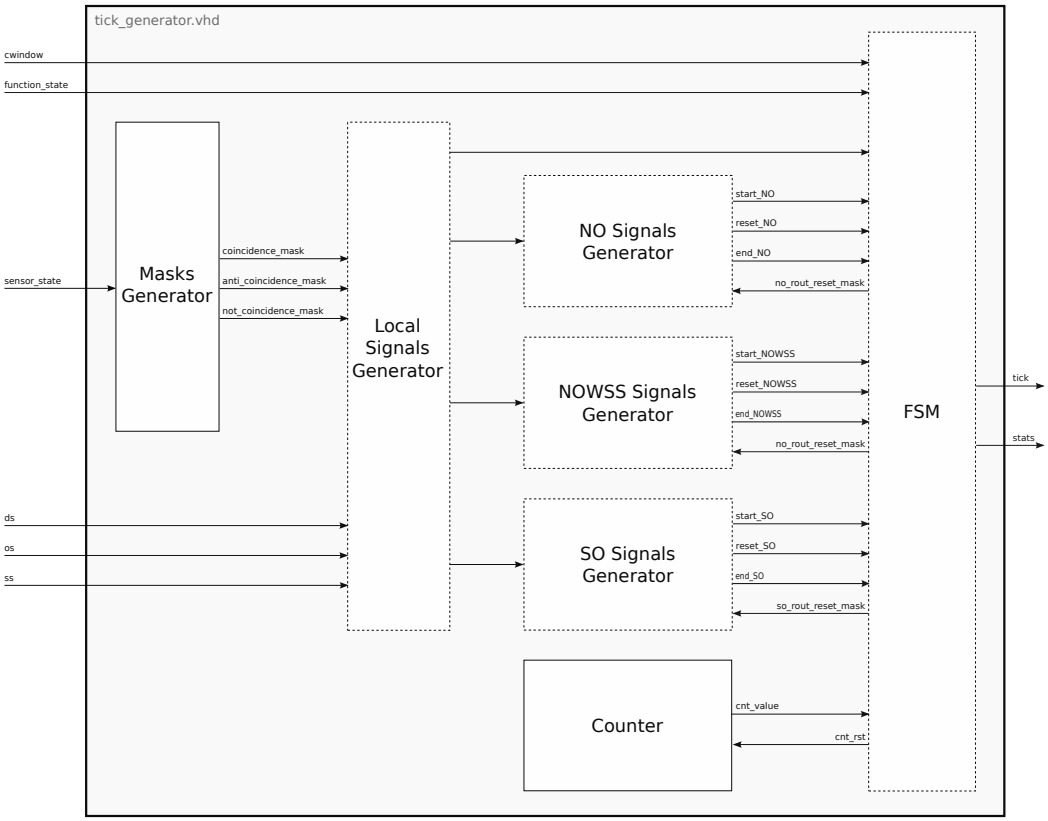

Figure 3.5: Structural overview of Tick Generator

NO - start

The start of the coincidence window takes place when any of the first level trigger is received. To determine this moment the coincidence inputs are first isolated with the help of a coincidence mask, which has '1' for every sensor that is in the coincidence state and '0' for the rest. The coincidence inputs are determined by performing a logical AND between the coincidence mask and the sensor triggers. If after this operation there are bits equal to '1' (determined by performing a logical OR between all the coincidence inputs) the start signal is assigned.

NO - reset

The coincidence window is reset when sensors that have already triggered trigger again. A reset mask that keeps track of the sensors that have already triggered is compared with the coincidence input in order to see whether the coincidence window has to be reset. The mask is initialized when the start signal is assigned. After that the mask is currently updated until the end of the coincidence window occurs, or it is reinitialized if the reset signal is assigned. In the end the reset signal is determined by performing the logical OR between all the bits of the reset mask.

NO - end

The end of the coincidence window in this state is marked when an anti-coincidence sensor triggers. In a similar way as determining the start signal, we need to create first

an anti-coincidence mask, which has '1' for every sensor that is in the anti-coincidence state and '0' for the rest. By applying logical AND between the anti-coincidence mask and the sensor triggers, and after that by applying logical OR between all the bits of the result, the anti-coincidence end signal is obtained. In this state, this is also the end signal. This is a common ending also for the other two operation modes, NOWSS and SO.

NOWSS - start/reset

The only difference between the NOWSS state and the NO state is that now appears a starting sensor which can start or reset the coincidence window. As a result the start/reset signal is exactly this starting sensor.

NOWSS - end

The end signal of the NOWSS operation mode should be asserted when one or more sensors that already triggered trigger again, but the starting sensor is not among them, or when one of the anti-coincidence sensor triggers. The first part is similar to the reset signal of the NO operation mode, and is treated in a similar manner.

A reset mask is initialized once the start signal was received and it is updated with every clock cycle unless the coincidence window ends or it is interrupted. In the beginning we need to AND the current mask with the coincidence input. By applying logical OR between all the bits of this result we know when a sensor that already triggered triggered again. We have to make a logical AND between this last signal and the inverse of the starting sensor, to be sure that the coincidence window is not reset. In the end a final OR with the anti coincidence end signal will signalize the end of the coincidence window.

SO - start/reset

For the SO operation mode the order in which sensors trigger is important. We have to keep in mind that consecutive sensors can trigger at the same time if the sensor that it is expected to trigger is among them. Also sensors that are put in the dont care state can trigger at the same time, as they do not influence the coincidence window.

The first step is to create the inverse of the coincidence mask and OR it with the ordered sensor triggers. Next we have to compute the leading 1 value of the previous result and OR it again with the inversed coincidence mask. By applying logical XOR between this last result and the ordered sensor triggers, we can determine if an out of order sensor that is in the coincidence state has triggered, by performing a simple OR operation between all the resulted bits. In this way a sensor that is in the dont care state but out of order can trigger with any effect, but if a coincidence sensor which is out of order triggers, an output of value 1 will appear, signalizing the fault. If this final output is 0, but we have a valid event (operation performed simultaneously by OR-ing all the sensor triggers) and at the same time the value of the anti coincidence end signal is 0, then the start/reset signal can be assigned.

SO - end

For the end signal of the SO operation mode a similar approach is used. Once the coincidence window has started a record of the sensors that already triggered is kept. This plays the role of the inversed coincidence mask from the start/reset algorithm. This mask is OR-ed with the ordered sensor triggers and then the leading 1 value of the result is determined. By applying logical XOR between this result and the ordered sensor

triggers we can see if an out of order sensor has triggered, which can cause the end of the coincidence window.

Another case when the coincidence window ends is when a sensor that already triggered triggers again and it is not the sensor placed on the first position. To determine this situation we have to AND the mask of the sensors that have already triggered with the current sensor triggers. If now by OR-ing all the bits of the previous result a value of 1 appears, and the start/reset signal was not assigned, the coincidence window has to be interrupted again.

The last case when the end signal is assigned is when the anti coincidence end signal is '1'.

3.4 Trigger System

This chapter would not have been complete without some information regarding the trigger system. We have presented so far the hardware implementation of the CB, but the entire trigger system is in fact a combination between hardware and software. The CB generates the final trigger of the trigger system, but through the software that runs on the CPU the block is controlled and configured in order to suit the experiment.

The design of the trigger system involves several stages. In the first phase of the design the number of the first level triggers is determined. Based on this value and on some other information that is known about the experiment the other CB pre-synthesis variables can be specified (see section 3.2.3). Next the rest of the IPs are selected and the SoC architecture is known. After that the bitstream is generated and the FPGA can be programmed.

The role of the software begins once the SoC architecture is placed on the FPGA. First, the parameters of the first level triggers and of the CB have to be configured, by setting their specific registers. After that the start signal can be assigned. The software then waits for a CB trigger to be received. When this happens it will check what sensor triggers have been received in order to visit their associated IPs to retrieve the actual data. In the end the CB will be restarted and the operation will start all over again.

3.5 Conclusion

In this chapter we have presented in detail our high level trigger IP core. Based on the functional requirements, introduced in the first part of the chapter, the design of the block emerged. We have split the component into two main parts. One assures communication with the AMBA bus, while the other one generates the trigger signal. Even though we focused mainly on the hardware implementation, we specify that the final trigger system is a combination between hardware and software.

To conclude, we state that our developed high level trigger IP core provides plenty of configuration opportunities which allow the user to generate complex trigger rules in order to suit the experiment requirements.

In the next chapter we present the setups that were created for verification and testing purposes.

Experimental Setup and Results

4

In the previous chapter the Coincidence Block was described in detail, introducing both its functionalities as well as its hardware design. In this chapter we present the experimental setup that was developed for testing the IP behavior in real situations. For this we focused the Multifunctional Particle Spectrometer, introduced in chapter 2, in which the Coincidence Block can be easily integrated. However, before reaching the testing stage, several other aspects need to be introduced and explained.

One of these aspects regards the GRLIB library IP cores, as well as the processor used to assemble the SoC architecture. Information about this is included in the first section of this chapter. The development board together with the FPGA and other hardware/software tools that were used are presented in the following two sections, 4.2 and 4.3. We continue with the design verification process, presented in Section 4.4. The testing setup including some information regarding pseudo random generators used in the simulation follows in Section 4.5. In the end of this chapter, Section 4.6, the obtained results are discussed.

4.1 GRLIB IP Library

The GRLIB IP Library is provided under the GNU GPL license by Aeroflex Gaisler [5], by which it was developed and is currently maintained. It supports different CAD tools and target technologies and it is vendor independent. The library consists of an integrated set of IP cores that are designed for SoC development. These IP cores are reusable and centered around the AMBA Bus on-chip interconnection protocol.

The library includes cores for AMBA AHB/APB control, the LEON3 SPARC general purpose processor, 32-bit PC133 SDRAM memory controller, 32-bit PCI bridge with DMA, 10/100/1000 Mbit Ethernet MAC, 8/16/32-bit PROM and SRAM controller, 16/32/64-bit DDR/DDR2 controllers, USB-2.0 host and device controllers, CAN controller, TAP controller, SPI, I2C, ATA, UART with FIFO, modular timer unit, interrupt controller, and a 32-bit GPIO port. Memory and pad generators are available for Virage, Xilinx, UMC, Atmel, Altera, Actel, and Lattice.

For our SoC implementation we use the following IP cores: LEON3 processor, AHB bus controller, memory controller, a debug support unit for the LEON3 over a serial port, and the AHB/APB bridge. The CB is connected to the APB bus together with the other IPs that were developed for verification/testing purposes, as well as the UART port.

4.1.1 Plug & Play Capability

GRLIB is organized around VHDL libraries. For every major IP (or vendor IP) a unique library name is assigned. In case of a vendor IP, several IP cores can be found in one library. Each library typically contains a number of packages where the exported IP cores are declared and their specific set of interfaces are defined. By using separate libraries, name clashes between IP cores are avoided. At the same time unnecessary implementation details are hidden from the end user.

Extension of GRLIB can be performed by adding new developed IP cores to an existing library, or by adding first a new library into which the new IP cores are included together with their associated packages. An IP core with an AMBA interface can be easily adapted to fit into GRLIB, especially if the AMBA signals are declared as standard IEEE-1164 signals. If this is the case, then each signal has to be assigned to the corresponding field of the AMBA record types which is declared in the GRLIB.

Once the desired configuration has been planned, GRLIB provides a plug&play method to be used in order to configure and connect the selected IP cores, without being necessary to make any global changes. The plug&play information consists of three items: a unique IP core ID, AHB/APB memory mapping, and used interrupt vector. This information is sent as a constant vector to the bus arbiter/decoder where it is mapped on a small read-only area in the top of the address space.

4.1.2 The LEON3 Processor

The Leon3 processor is a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture. It is based on the Harvard architecture, implementing a 7-stage pipeline with separate instruction and data cache buses, with hardware units developed for MUL, MAC, and DIV instructions. An optional IEEE-754 FPU with support for single- and double-precision floating point operations can be attached. The cache system supports multi-set caches with up to 4 ways per cache, 256 kbyte per way.

A new version (LEON4) has been released recently. Is is an evolution of LEON3 with improved performance thanks to wider internal buses, modified pipeline and support for a Level-2 cache. As this appeared at the end of our work, we were not able to use it.

The processor can be utilized in uniprocessor as well as multiprocessor systems. Up to 16 CPU cores can be implemented in asymmetric multiprocessing (AMP) or synchronous multiprocessing (SMP) configurations. According to the manufacturer the processor reaches 150 MHz on an FPGA, and 800 MHz on a 65nm ASIC technology.

The LEON3 is highly configurable and particularly suitable for SoC designs. It is interfaced using the AMBA AHB bus protocol and supports the IP core plug & play method provided by GRLIB.

For our implementation we have used a minimal instance of the processor, including the Debug Support Unit (DSU) which offers access via the serial port to all on-chip registers and memory. We can inspect in this way the trace buffers of both executed instructions and AMBA bus traffic.

A non-free fault-tolerant (FT) version that offers Single Event Upset (SEU) immunity with no timing penalty compared to the non-FT version is also available. This makes LEON3 an attractive solution for SoC based space projects.

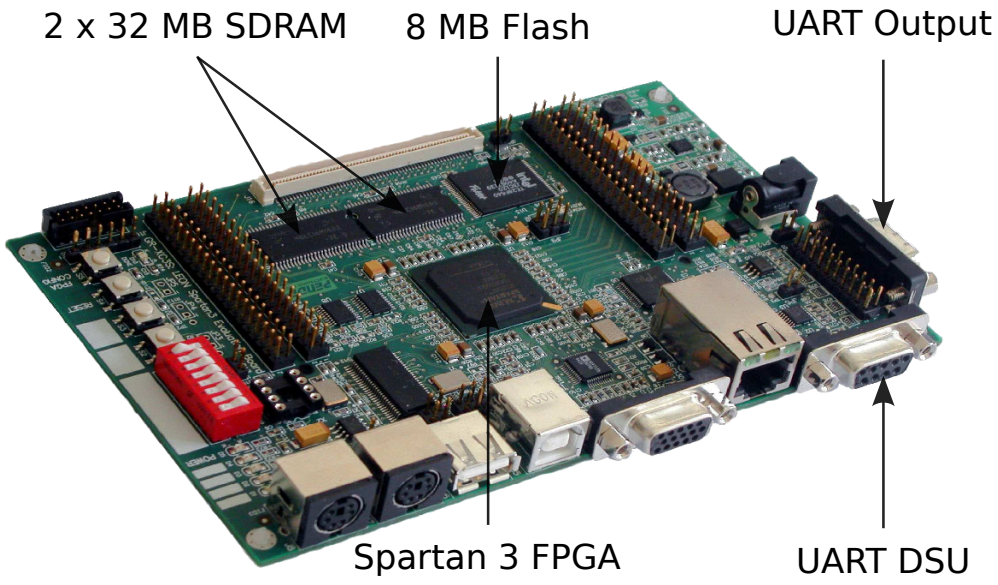

Figure 4.1: Development board

4.2 Development Board

For testing purposes we have used the GR-XC3S development board, designed by Pender Electronics GmbH [10]. The GR-XC3S board is compact, low-cost, targeted especially for the evaluation of LEON3/GRLIB processor systems. The complete design, which includes the LEON3 processor, the required GRLIB IP cores, together with the developed Coincidence Block and the testing IPs, are programmed on the FPGA that is present on board.

In Figure 4.1 the GR-XC3S board can be seen. Its core is represented by the 1.5 million gate XC3S1500 Spartan 3 FPGA designed by Xilinx [12]. The on-board memory is positioned just above the FPGA. It consists of 64MB SDRAM as well as 8MB flash memory. On the bottom-right corner the two serial (1Mbaud RS232) ports available are located. One is used for the Debug Support Unit and the other one for interfacing with the host PC. The FPGA is clocked by a 50MHz crystal that is part of the development board.

Besides the parts that were mentioned above, the board features also an Ethernet 10/100 Mbit MAC and PHY, 24 bit video DAC, USB PHY Controller, and PS2 mouse and keyboard interfaces. All these components were not used in our implementation, which increased the total available area of the FPGA.

4.3 Development Tools

So far we have only discussed hardware related information. This section focuses on the software tools that were used for development and testing. A general rule based on

which we selected the software was that it should be freely available and if possible also open-source.

In order to simulate our design we have used GHDL [6], a complete VHDL simulator that uses GCC technology and implements the VHDL language according to IEEE standards. After GHDL compiles the VHDL files it will directly create binaries or executable images. This is one of the best form for testbenches. A binary can also create Value Change Dump ASCII based format files, or GHDL specific GHW dump format files. These files can be visually inspected with a waveform viewer. We have used for that GTKWave [7], which has the ability to read both GHDL format dump files, as well as many others. GHDL and GTKWave are both open-source software, released under GPL.

To synthesize the complete SoC design and to program the FPGA with the generated bitstream we have used scripts that are provided by GRLIB. These scripts allow to run under the command line Xilinx synthesis and place&route tools. In order to do that the ISE WebPack Software Design was needed, which can be downloaded from the Xilinx website. The ISE WebPack is free but not open-source. Except for the tools already mentioned, it includes a variety of other tools that are useful for a designer. One such tool is the FPGA editor which allows the visualization of the placement and routing of the synthesised design. Another tool, the timing analyzer, helps to debug timing errors.

To compile the embedded C code we have used the LEON Bare-C Cross Compiler (BCC), offered by Gaisler Research. BCC allows cross-compilation of single and multi-treaded C and C++ applications created for LEON systems. It offers also a small tool to pack the compiled executable into a PROM file that can be uploaded to the on-board flash memory.

Another tool that we used, also provided by Gaisler Research, is GRMON - a debug monitor for LEON processors. It communicates with the LEON DSU and it allows access to all registers and memory, instruction and AMBA trace buffers. It can be used also to download and execute applications, insert breakpoints, remote connection to GDB and flash programming. GRMON is not free, but an evaluation version is available.

4.4 Design Verification

In order to verify our design we have created test-benches for every top-level component and subcomponent in particular. Pre-synthesis, post-synthesis, and post-map simulations were performed using GTKWave as a viewer.

As the simulation results were good, for the next step we have developed a Verification IP to be integrated together with the CB. With a minimal selection of GRLIB IP cores that are mandatory, the architecture was implemented on the FPGA. Some important internal signals were routed to the external ports of the FPGA where we have connected a logic analyzer to check their behavior. This operation is similar to the simulation check performed with GTKWave during the previous step. An overview of the verification setup is presented in Figure 4.2.

The verification IP is designed in such a way that various test sequences can be loaded and run. Tests are created by specifying first the total number of events to be generated. After that for every event in particular two values must be specified:

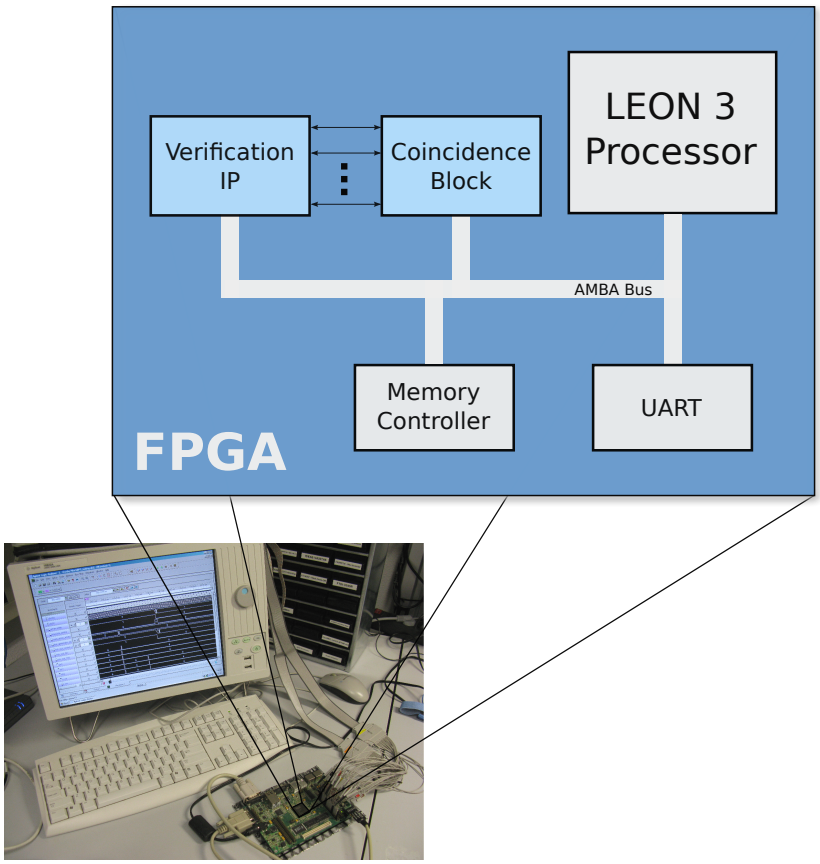

Figure 4.2: Verification setup

- The sensors that are active only one cycle, having logic value '1';
- The number of empty cycles that will follow, in which no sensor has logic value '1'.

We can see that by using the verification IP all the functionalities of the CB can be verified. In this way different verification tests were developed for every operation mode in particular. The coincidence window was varied and all the start/reset/end signals were checked. As part of this verification process the embedded software that runs on LEON3 to control the AMBA peripherals as well as the verification process had to be also developed.

4.5 Testing Setup

We have presented so far the verification process. In order to validate our CB design and to test its functionalities, we have created a testing setup in which we have included the CB into the MPS. An overview of the entire system is presented in Figure 4.3. The analogue data obtained from the silicon pixel trackers and from the scintillator are digitized and becomes input to some custom IP filters which act as first level triggers. These triggers are the inputs based on which the final trigger is asserted by the CB.

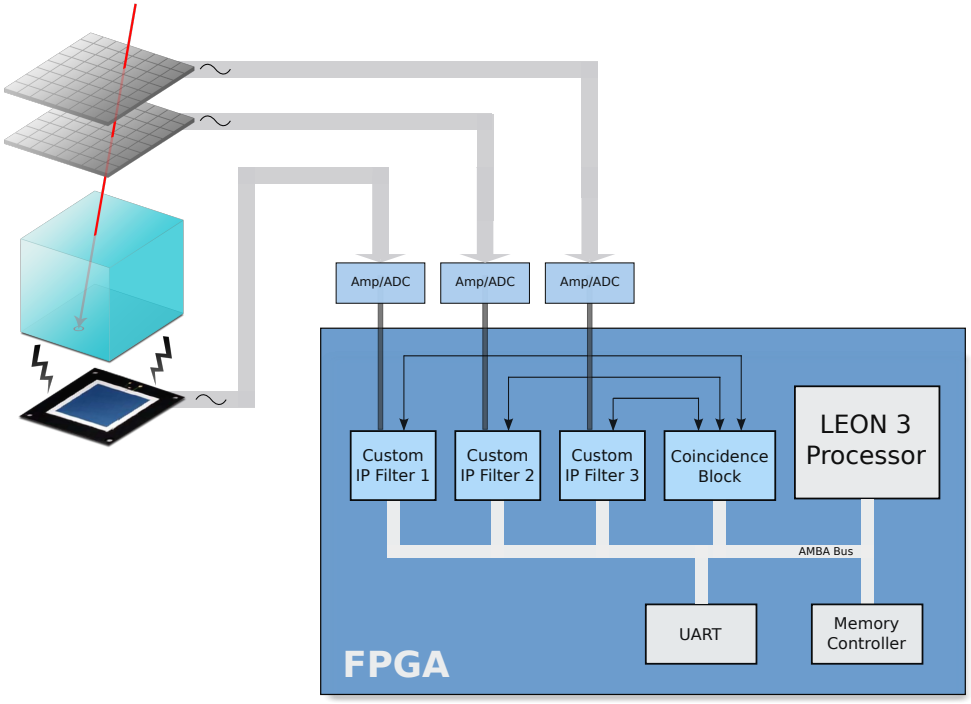

Figure 4.3: MPS read out system including CB

We can derive several functions that the CB can have on the MPS. The basic is to tell when an event took place at the same time in the scintillator as well as in the silicon pixel trackers. To do this all the first level triggers should be put in the coincidence state.

Another role is to determine when one of the top layers is not working. This can be performed by turning its respective sensor trigger state to anti-coincidence and observe how the others behave. If there are plenty of hits in the layers that are placed on a lower position it means that there is a problem with that specific layer that was put on anti-coincidence. The coincidence check state can be used as well for this operation.

In all the cases the CB helps in selecting only the valuable information from what is read. It also time stamp it to be of any scientific use and to be able to correlate it with other instruments.

4.5.1 MPS Testing Generation IP

As the MPS is still in a preliminary phase we were not able to perform real tests including all the necessary equipment. Therefore we created the MPS Testing Generation (MPSTG) IP core that simulates the input behavior for the CB by creating first level triggers.

A simplified structural overview of the MPSTG is presented in Figure 4.4. We can see that the MPSTG is connected to the AMBA bus (AMBA APB) in the same way as the CB. There is an AMBA Wrapper that takes care of this connection, which acts also

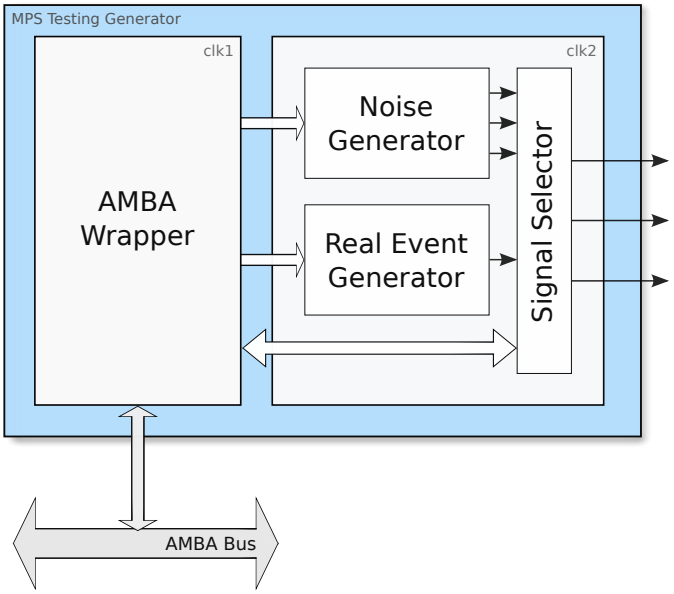

Figure 4.4: Structural overview of the MPs testing generation IP

as a control block. Inside the AMBA Wrapper a set of memory mapped registers are available to store the configuration. The clock frequency of this block is the one used also for the AMBA bus.

On the right side of the figure we have the block that generates the triggers. It works at a clock frequency of 100 MHz, same as the coincidence path block of the CB, which generates the final trigger. To simulate closely the real behavior we have split the signal generation phase in two parts. One block creates the real events while the other one creates noise. When a real event is generated the signal selector block will set it automatically as the final output. If no real event is generated, but there is a noise event, then the noise value is set as output. If there is no noise nor real event generated the output of the block contains only '0' logic values.

4.5.2 Noise/Real Events Generation

To simulate as much as possible the environment in which the MPS should detect radiations we have chosen to generate the events in a random manner. For this we have used linear feedback shift registers (LFSRs), as they provide an accessible way to generate pseudo-random values in hardware.

An LFSR is a shift register which uses feedback to modify itself at each rising edge of the clock. The feedback causes the value in the shift register to cycle through a set of unique values, starting from the initial one which is called the seed. The bit positions that affect the generation of the next state, being used in the feedback function, are called taps. An LFSR with a well chosen feedback function can produce a sequence of bits which appear random and which has a very long cycle.

One of the mostly used types of LFSR is the Fibonacci (called also many-to-one

or standard). In a Fibonacci LFSR all the bits are shifted one position to the right unchanged and the left-most bit value (the input bit) is determined from the taps by performing an exclusive-OR/NOR (XOR/XNOR) logic operation.

A maximum-length LFSR cycles through all the possible $2^n - 1$ states within the shift register, producing an m-sequence. We specify that a state with all ones is illegal for an XNOR based LFSR. In the same way a state with all zeros is illegal for an XOR based LFSR. This is considered illegal because the counter remains in a locked-up state.

In our experiment we have used three maximum length 16 bit XNOR Fibonacci LFSRs to produce the noise events. For all of them the used taps were 16, 15, 12 and 4, while the seed was different in order to achieve better randomness. To generate the real events only one maximum length 8 bit XNOR Fibonacci LFSR was used, with the taps sequence 8, 6, 5 and 4.

The generation of a noise or a of a real event takes place only when the value of their associated LFSR goes above a certain threshold. In this way some control over the way in which events are generated is achieved.

4.6 Results

The experiments that we have done can be split in two main parts. In the beginning we have performed simulations to see if the CB behaves as expected, and also to check the maximum performance that can be obtained. For this, both the CB and the MPSTG were included into one top level component that simulated the behavior of the entire assembly. We specify that in this case the dead time was caused only by the CB, since the dummy model for the processor reacting immediately after a new final trigger was asserted.

For the second part of the experiment we have programed the FPGA that resides on the development board with the minimal required configuration, presented in the introduction of this chapter. The complete SoC architecture interfaces through the serial port with the PC. This communication is a simple task as the system is configured to redirect all the communication to the serial port, while on the PC side a terminal application with logging capabilities is used to retrieve the transmitted values. Appendix B contains an example of such a C code which we have developed.

An experiment test consisted in varying the threshold of the noise generator within its possible limits, while keeping the real events generator threshold constant. Measurements were then taken for different values of the real events generator threshold. The execution time for each test was 655.35 μs (equivalent to 65535 clock cycles at 100 MHz). During this time interval the 16 bit LFSR used for the noise generator completes one cycle, while the 8 bit LFSR used for the real events generator completes approximately 256 cycles.

We present the results obtained for different values of the real events generator threshold, first for the simulation and after that directly from the FPGA. The operation mode of the CB was no order, and all the sensors were put in the coincidence state.

4.6.1 Simulation Results

In Figure 4.5 the threshold value for the real events generator was set to 250. Therefore the computed frequency of appearance of the real ticks is 1.96 MHz, signaled on the plot by the green line. With the blue color the trigger frequency of appearance is represented. Please note that in the following plots we refer to the frequency of appearance as frequency.

Coming back to Figure 4.5, we can see that if there is little noise, the trigger frequency meets the real events one. As the noise level increases there is a bigger chance that these events will meet the rules and the trigger will be asserted. This actually happens, and it can be seen also in the plot.

In Figure 4.6 we have lowered the threshold value for the real events generator, setting it to 232. By doing this the actual frequency of the real events was increased to 9.02 MHz. We can see that with zero noise this value is not equaled by the trigger frequency, the reason being the dead time. If however the noise frequency is increased then the two lines intersect at some point, after which the trigger rises even more. This is because the noise events meet the rules and are interpreted as good events.

If we increase the real events frequency to 18.44, by setting the real events generator threshold value to 208, the trigger frequency will not meet it even with the total possible noise. This can be seen in Figure 4.7. The maximum trigger frequency that can be obtained is around 16.5 MHz. The dead time in simulation equals to 60 ns (6 clock cycles).

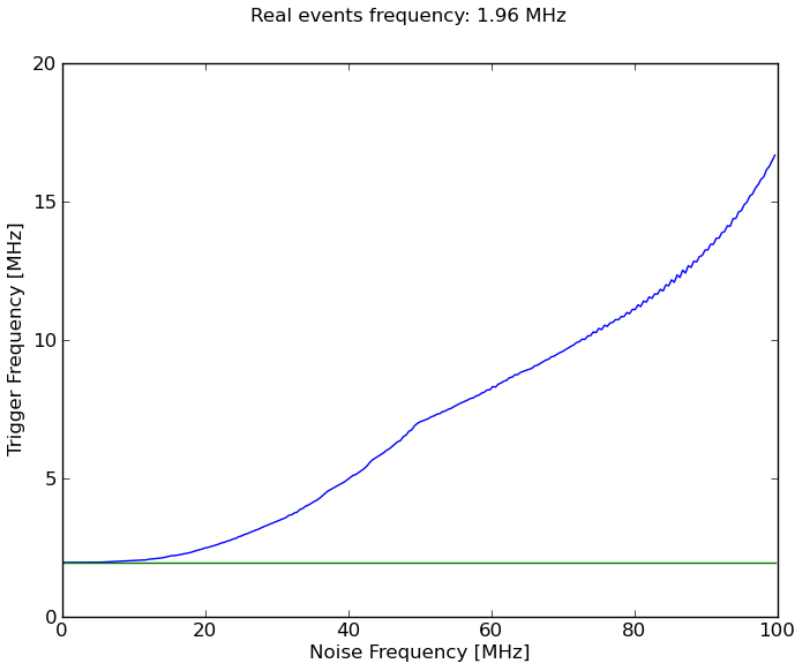

Figure 4.5: Simulation results - real events generator threshold value of 250

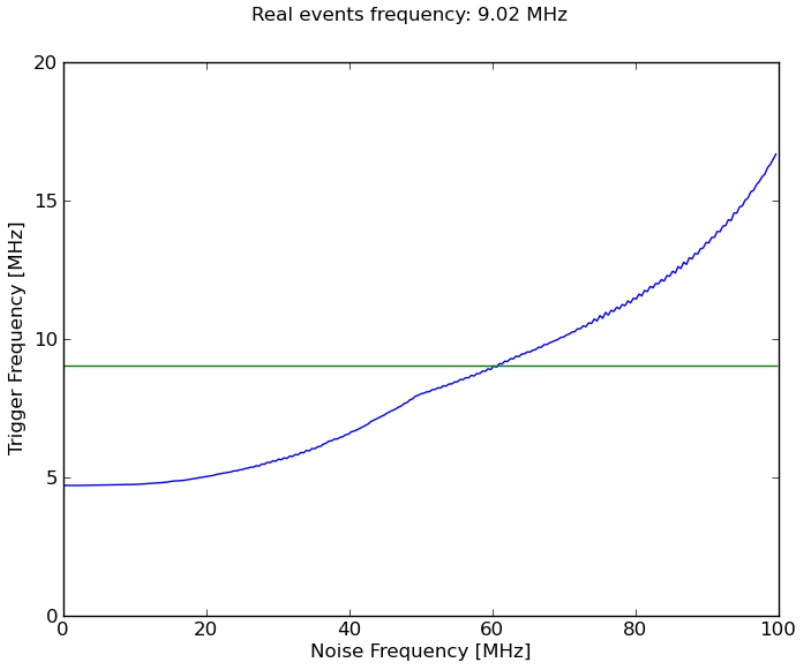

Figure 4.6: Simulation results - real events generator threshold value of 232

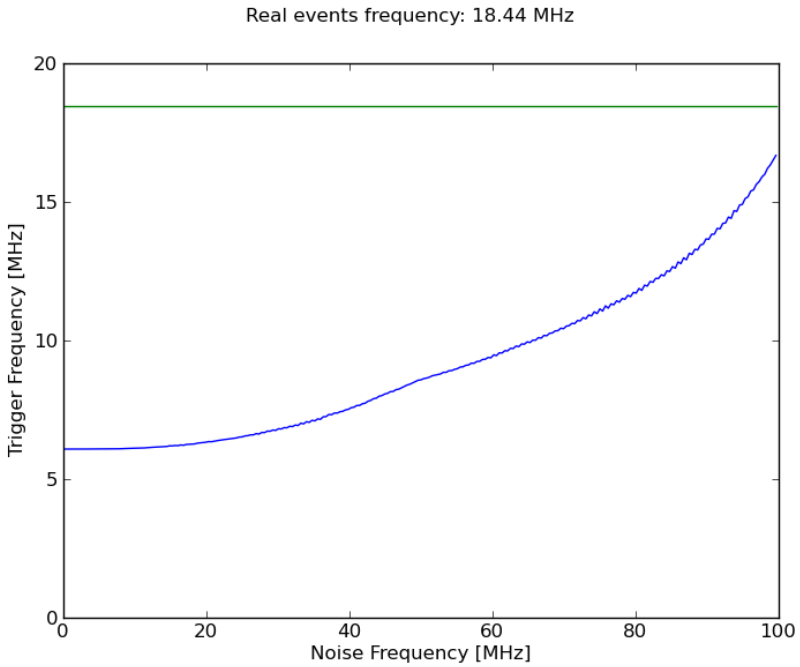

Figure 4.7: Simulation results - real events generator threshold value of 208

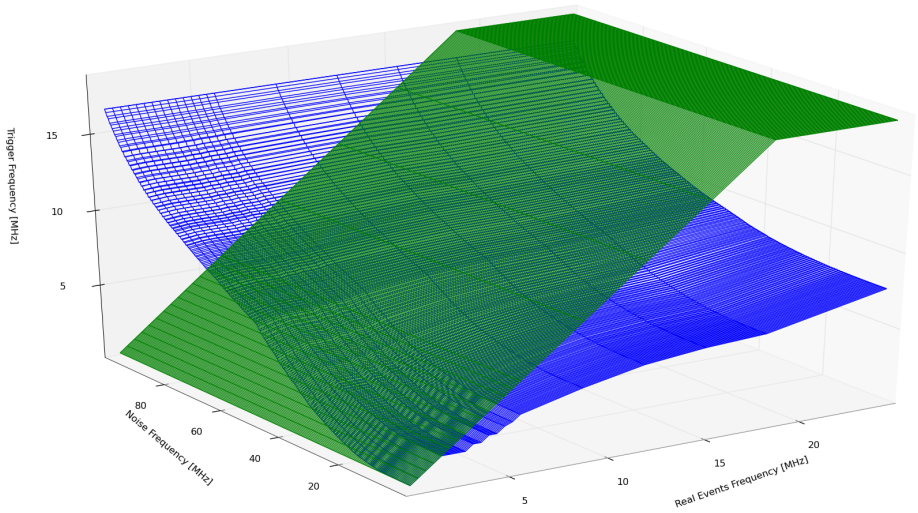

Figure 4.8: Simulation results all together

All the results obtained in simulation are presented in Figure 4.8. We can see better here that as we increase the real events generator threshold value, after some point the dead time appears. This affects the trigger frequency and the CB cannot trigger for all the real events.

We mention that in this 3D plot we have 'cut' the rising part of the real events frequency. This would have had continued its linear ascent.

4.6.2 FPGA Results

For the FPGA the results regarding the dead time were totally different. In Figure 4.9 the threshold value of the real events generator was set to its maximum possible value, 254. We recall that for our LFSR implementation the seed must not be all '1' (255 in decimal) as this is a locked-up state. We can see that when there is no noise the trigger frequency meets the one for the real events generator.

Already from this picture we can derive that the maximum frequency for the trigger is around 0.65 MHz. This is confirmed in Figure 4.10, where we have used a threshold value of 253 for the real events generator. Even with all the possible noise the green line is not reached by the blue one, which represents the trigger frequency.

All the FPGA results are presented in the 3D plot, Figure 4.11. We have again cut the green part for a better view, as it would have gone to high. As it can be seen from the plot, the dead time of the instrument is around 1.50 μs in a real situation. This has different causes, but the main part of it is due to the processor. We mention that in our situation the AMBA bus was free. If some other IPs are present this dead time might increase due to the bus traffic collisions.

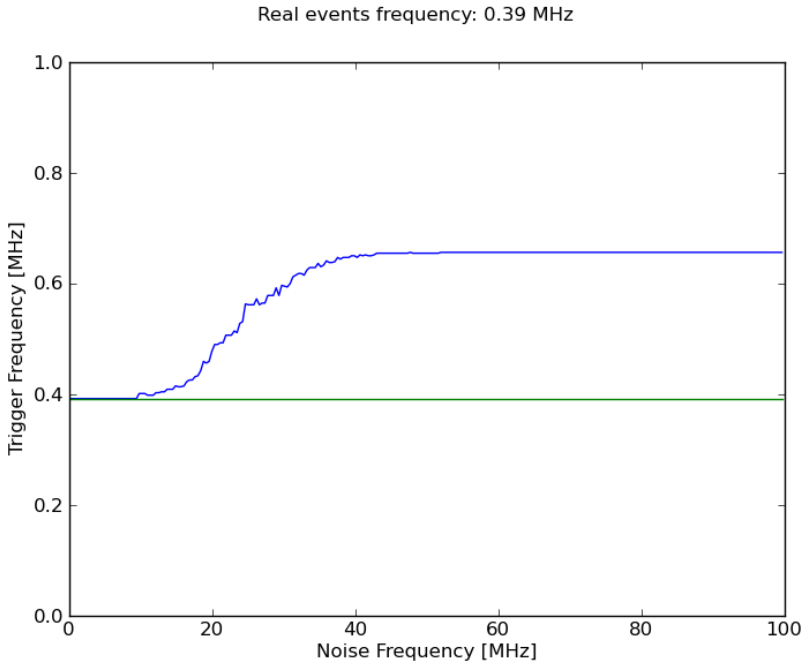

Figure 4.9: FPGA results - real events generator threshold value of 254

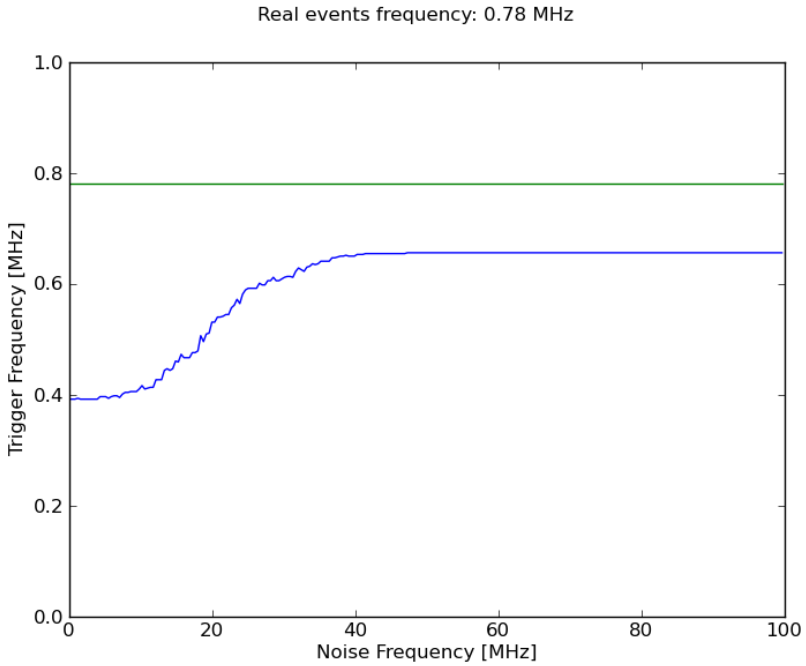

Figure 4.10: FPGA results - real events generator threshold value of 253

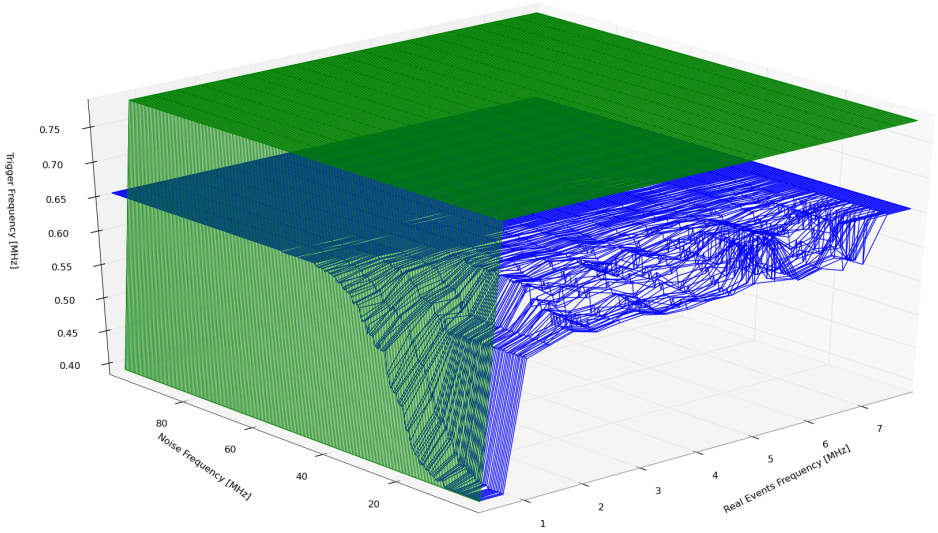

Figure 4.11: FPGA results all together

4.7 Conclusion

In this chapter we have investigated in detail the behavior of our trigger logic IP core. As a first step, a verification setup was created. Once this process proved successful, we have developed a realistic testing setup. Our IP core was integrated into the MPS, a space instrument that requires trigger logic. In order to check the performance of our component, we have generated both noise and real values in a manner that allowed us to have control over the final outcome of the events. Results show that the behavior of our proposed trigger design meets our expectations and can be successfully integrated in space instruments.

So far we have targeted only the intra-module part of the thesis. In the next chapter we address the Space Wire time-codes mechanism as part of the inter-module timing accuracy.

5 Spacewire Extension

The final part of the project involves a study of the current inter-module time distribution accuracy, focusing on SpaceWire networks. In section 5.1 a short overview of SpaceWire is given. Section 5.2 presents the time-codes, followed by the time accuracy that can be obtained (Section 5.3) with the present SpaceWire standard. Next the possible improvements are introduced, in Section 5.4. In the end, Section 5.5 discusses the changes that are required by different SpaceWire components to be able to implement those improvements.

5.1 Introduction to Spacewire

Sensors, processing-units, down link telemetry sub-systems and other electronic equipment used in a spacecraft today are easily interconnected via SpaceWire networks. Within such a network, serial, high-speed, bi-directional, full-duplex links and packet switching wormhole routing routers are used. Developed first by ESA based on the IEEE 1355 standard, SpaceWire nowadays is used worldwide by NASA, JAXA, and other space organizations.

An important aspect of a SpaceWire network is that a specific topology is carefully planned ahead and no further changes might appear. In contrast for example with any computer LAN where nodes can be connected or disconnected at any time and the topology of the network is in a current change.

Communication across a SpaceWire network is realized by sending control characters and data characters. By linking the ESC and the FCT control characters the NULL control code is formed, used to keep the link active and to support link disconnect detection. By linking the ESC control character with a data character the Time-Code control code is obtained. SpaceWire uses time-codes in order to distribute time information across a SpaceWire system.

Time-codes should not be used to increment a time-counter at the receiving nodes, with the expectation that the counting value corresponds to the system time. The reason behind is that a missing time-code results in time discrepancy. Instead, each node should have a local time-counter that is periodically updated whenever a time-code has been received.

Time-codes are generated by only one node of the network, the time master, that takes care of the global time management. Routers broadcast the received time-codes to all their ports in order to distribute the global time to all the nodes of the network. In this way a global synchronization in the order of microseconds is achieved. Unfortunately this is not suitable for many today's applications. If we consider for example SpaceWire being used for sensor readout synchronization it is important to know that certain events took place exactly at the same time. This requires time resolution in the order of few

nanoseconds to be able to use the recorded data for any scientific studies.

Time-codes delays are associated with skew and jitter. The skew delay is constant at each router and consists of the time in which the time-code traverses the router. In this the cable delays and the processing time is included. The jitter delay varies and appears because even though the time-codes have highest priority they still have to wait for the transmission of the current character to end.

The total sum of skew and jitter delays depends also on the speed and size of the network. If we consider the maximum possible speed, defined by the standard at 400 Mb/s, and a network with just two nodes connected, the total delay associated with skew and jitter varies between 25 and 60 nanoseconds. Thus, we can see that by just increasing the speed until it reaches the maximum is not a way of reducing delays to meet the requirements. Some possible improvements are presented next, that aim to achieve nanosecond synchronization irrespective of the network speed. The solutions are based on reducing skew and jitter delays.

5.2 Time-codes

In the following we introduce time-codes as they are presented in the ECSS-E-ST-50-12C/2008 standard [3]. The time control code is formed from an ESC followed by a single data character. From all the four bits of the ESC character three are always set to logic '1'. The remaining one is the parity check bit (P from Figure 5.1). The least significant bits of the data character, T0-T5 hold the time value. The most significant bits (T6, T7) contain two control flags for future general use (they do not have a specific function for time distribution and shall be both set to zero [3] 7.7.h).

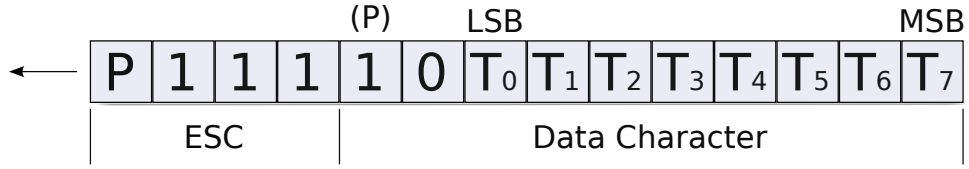

Figure 5.1: Time-code character

5.2.1 Sending Time-codes

The TICK_IN signal is used to request the transmission of the time-code value, TIME_IN. Only one node has an active TICK_IN, namely the time master. All the other nodes keep their TICK_IN signal not asserted.

When TICK_IN is asserted the TIME_IN value is sent immediately after the transmission of the current character ends. Time-codes are sent only in the Run state.

5.2.2 Receiving Time-codes

The receiver determines that the current character is a time-code.

Each node and router contains a 6 bit time-counter. The value that is received has to be one more than the value stored in the counter to be declared valid. If this happens the TICK_OUT signal is asserted.

The value of the counter is updated if the received value is different than the currently stored value. The reason behind is to deal with link errors. In case one or more time-codes were missed the router will be able to recover the correct time. This might happen with the second time-code received after link recovering.

The TIME_OUT value is propagated up to the application (in a node) or to all the output ports (in case of a router) only if it is valid.

TICK_OUT is asserted only if the link interface is in the Run state.

5.3 Current Timing Accuracy

We start with the accuracy determination from the time master, where the time-code is generated. According to the standard, after the TICK_IN signal is asserted, the time master starts transmitting the time-code value in the first possible moment, after the current character is transmitted. After the time-code leaves the time-master it has to traverse a number of routers until it reaches the destination (every end node). At each router, the time-code might have to wait again for the current character to be transmitted. We can see that the delay can increase each time a router is passed. This results in jitter delay, determined by the number of bits that the time-code has to wait before being transmitted.

Once the transmission of the time-code has began, a certain amount of time has to pass until it arrives at the receiving link. This time represents the skew delay, composed by the time required for all the 14 bits to travel, eventual cable delays, and the time in which the time-code is processed to be available for use.

As each transmitter might wait for a data character to be transmitted (10 bits) before being able to send the time-code there can be a maximum time jitter of:

$$T_{jitter} = 10 * N/R = 10 * N * T_{bit}$$

As each time-code has a total of 14 bits, this results a time skew of:

$$T_{skew} = 14 * N/R = 14 * N * T_{bit} + T_{cable} + T_{processing}$$

The total delay can be expressed as:

$$Delay = T_{skew} + T_{jitter}$$

Where:

- N - total number of links that have to be traversed;
- R - average link operating rate;
- T_{bit} - bit transmission time;
- T_{cable} - cable delays;

Table 5.1: Influence of the link speed and network size over the skew delay

		# Links							
		1	2	3	4	5	6	7	8
Speed Mb/s	10	1.400	2.800	4.200	5.600	7.000	8.400	9.800	11.200
	25	0.560	1.120	1.680	2.240	2.800	3.360	3.920	4.480
	50	0.280	0.560	0.840	1.120	1.400	1.680	1.960	2.240
	100	0.140	0.280	0.420	0.560	0.700	0.840	0.980	1.120
	200	0.070	0.140	0.210	0.280	0.350	0.420	0.490	0.560
	300	0.047	0.093	0.140	0.187	0.233	0.280	0.327	0.373
	400	0.035	0.070	0.105	0.140	0.175	0.210	0.245	0.280
	3000	0.007	0.013	0.020	0.026	0.033	0.039	0.046	0.065

Table 5.2: Influence of the link speed and network size over the jitter delay

		# Links							
		1	2	3	4	5	6	7	8
Speed Mb/s	10	1.000	2.000	3.000	4.000	5.000	6.000	7.000	8.000
	25	0.400	0.800	1.200	1.600	2.000	2.400	2.800	3.200
	50	0.200	0.400	0.600	0.800	1.400	1.200	1.600	1.600
	100	0.100	0.200	0.300	0.400	0.500	0.600	0.700	0.800
	200	0.050	0.100	0.150	0.200	0.250	0.300	0.350	0.400
	300	0.033	0.067	0.100	0.133	0.167	0.200	0.233	0.267
	400	0.025	0.050	0.075	0.100	0.125	0.150	0.175	0.200
	3000	0.003	0.007	0.010	0.013	0.017	0.020	0.023	0.027

- $T_{processing}$ - time required to process the time-code.

The accuracy with which a time-code is distributed depends also on:

- the size of the network (total number of links a time-code has to traverse);
- speed rate of the network.

From Tables 5.1 and 5.2 we can see that at a maximum link speed of 400 Mb/s a delay between 35-60 ns appears at each link that is traversed. This is due to the fact that to the skew delay of 35 ns a jitter delay that can reach a maximum of 25 ns can be added. To achieve a maximum delay of 10 ns per link a speed rate of 3000 Mb/s is required. This is unrealistic for today's technology. Instead of increasing the speed it is better to find a different solution to reduce these delays.

5.3.1 Delay Variation

The skew delay is constant at each router and is equal to 14 bit times. This is important if we consider network topologies where from the time master to node Ai time-codes pass the same number of routers whatever the path is. In this case each destination node can expect a constant skew delay, which equals the number of routers passed multiplied with

14 bit times. We assume also that the link rate is known. In contrast with the skew, the jitter delay is variable and can be between 0 to 10 bit times at each router.

5.4 Possible Timing Improvements

The SpaceWire standard at the moment is quite restrictive and does not offer any solution to the jitter and skew delays. Due to the constant value of the skew, network topologies that reduce it can be built without any change to the standard. This is not the case for the jitter as its value varies. In the following sections three methods of improving the accuracy are be presented. We note however they all require additions/modifications to the existing standard.

5.4.1 Method 1

One solution is to use a second time-code to transmit the jitter and skew delays. This second time-code has the control flag bits of specific values, different than '00', and should be sent immediately after the first time-code. In this way jitter and skew delays are being propagated to the receiving nodes. As the final nodes receives the delay represented in number of bits they have to transform that value to get the actual time delay in order of seconds.

By using this method, normal standard compliant devices can be used together with time aware standard compliant devices. The difference is that the second time-code is ignored by the first ones, which will consider it as not valid because of the value of the control bits being different than '00'. The time aware standard compliant devices can use the second time-code to compute the delay.

The two control bits of the second time-code can be used together to form a code. In this way, 6 bits can be made available to transmit the delay. As with 6 bits 64 values can be represented, and each router can add a maximum of 10 bits jitter delay, a total of 6 routers can be traversed. This is the case when only the jitter delay is transmitted. There is also the possibility to use only one control bit to signal the second time-code. A total of 7 bits can be used now, which doubles the number of possible traversed routers to 12. Again, if we transmit only the jitter delay.

For certain network topologies it might be required to transmit also the skew delay. In this case the skew can be recorded as the number of routers that are traversed.

This method, using a second time-code with the control bits 01 was proposed in [30] for the Simbol-X mission [15]. Four bits of this second time-code are used to transmit the jitter delay. At the receiving part this value is used to get a constant and known delay between the TICK_IN assertion and the TICK_OUT negative edge.

5.4.2 Method 2

Another solution is to use the control bits of the time-code to signal that another data character containing the delay follows.

This method is similar to the first one but with the difference that normal standard compliant devices cannot be used with the time aware ones.

A total of 8 bits can be used to transfer the delay associated with jitter and skew. The number of possible routers is 25 if only the jitter is transmitted, and 10 if we add also the skew.

5.4.3 Method 3

The basic idea of this method is to make the value of the jitter delay constant. As the longest possible character that can be transmitted is 10 bits long, a constant jitter delay can be obtained by sending the time-code always 10 cycles after the TICK_IN signal has been received [14]. A paused delay is introduced by holding both data and strobe lines to zero after the current character has been transmitted until those 10 cycles pass. In total this pause should not be more than the link disconnect timeout window of 850 ns. This requires the link to be running at 12.5 Mb/s or faster. For lower bit rates it is necessary to transmit a NULL if the number of pause cycles is less or equal to two. The minimum bit rate in this way will be reduced to 2.5 Mb/s.

Using this method for network topologies in which the number of routers that are traversed in order to reach a specific node is constant, the total delay time is also constant, and nodes can be programmed with its value. For network topologies where different paths are possible (with different lengths) from the master node to the same end node, one of the first two methods is still required to send the value of the delay. This value can be represented in number of traversed routers.

5.4.4 Bit Usage

All the possible bit usage combinations are presented in the following tables. The total number of bits that can be used was split first between the skew and the jitter, assuming that the jitter part counts the bits and the skew part counts the number of traversed routers. If the skew delay is counted also in bits then the total sum of jitter and skew can be transferred together. We note in here that for static network topologies only the jitter bits need to be transferred.

Table 5.3: Number of possible routers using 6 bits from the time-code

Jitter		Skew		# total possible routers
# bits	# possible routers	# bits	# possible routers	
5	3	1	2	2
4	1	2	4	1
If we add the jitter and skew at each node (64 bits) will result a max of 2 routers If we transmit only the jitter will result a max of 6 routers				

The number of routers that can be present in the network was computed by dividing the maximum range representation to the maximum delay at each link. We can see that it is better to represent both the jitter and the skew in number of bits in order to use more routers. It is best to transfer only the jitter bits to use the largest number of routers.

Table 5.4: Number of possible routers using 7 bits from the time-code

Jitter		Skew		# total possible routers
# bits	# possible routers	# bits	# possible routers	
6	6	1	2	2
5	3	2	4	3
4	1	3	8	1
If we add the jitter and skew at each node (128 bits) will result a max of 5 routers If we transmit only the jitter will result a max of 12 routers				

Table 5.5: Number of possible routers using 8 bits to transmit delay

Jitter		Skew		# total possible routers
# bits	# possible routers	# bits	# possible routers	
6	6	2	4	4
5	3	3	8	3
4	1	4	16	1
If we add the jitter and skew at each node (256 bits) will result a max of 10 routers If we transmit only the jitter will result a max of 25 routers				

5.4.5 Bandwidth Addition

We can divide the bandwidth by the frequency with which the TICK_IN signal is generated. Between two TICK_IN assertions a number of bits remain in which data information is being sent. This value is influenced by the link rate and the TICK_IN frequency.

5.5 Changes Required

In the following we present what actions each component has to perform and the changes required to the current implementations. As the first two methods are similar and consist of sending the delay after the transmission of the time-code the actions that each component has to perform are also similar from the functional point of view.

All the required changes should be implemented mainly in the transmitter and receiver blocks. Signals have to be driven out to the host interface at the receiving nodes and besides the TIME_OUT signal the delay has to be added. The received delay has to be transferred to the transmitter in routers, where the addition with the current one is performed. At the transmitter, as soon as the TICK_IN signal is received a counter has to be started to compute the delay which will be added to the current one received from the receiver.

5.5.1 Time Master

Once a TICK_IN command has been received the transmitter has to start counting the jitter bits. The time-code will be sent as soon as possible. The delay (either as second

time-code or data character) has to be the next character sent by the time master after the time-code.

The time master traverses the following action flow:

- receive TICK_IN and increment time counter;
- wait for the current character to be transmitted; count delay;
- send time-code;
- send delay.

5.5.2 Router

For the first method, after the first time-code has been received, the second time-code has to be identified based on the control bits. If the first-time code was valid and the decision to broadcast it was taken, the router has to count the jitter bits on each of its ports. While the first time-code is transmitted, the router has time to add his delay to the received one. Even if no second time-code was received, the router still transmits the delay right after the first time-code.

For the second method, the only difference is that the router knows that a delay follows after it has received the time-code with the specific control bits.

The router traverses the following action flow:

- receive time-code;
- check if time-code is valid and start receiving delay;
- wait for the current character to be transmitted and count jitter bits;
- send time-code; update delay;
- send delay.

5.5.3 Receiving Node

At the receiving node once a delay has been decoded it has to be updated. Then it is made available to the host interface only if the first time-code was valid.

The receiving node traverses the following action flow:

- receive time-code;
- check if time-code is valid and start receiving delay;
- update delay and assert TICK_OUT.

5.5.4 Method 3 Additions

For the third method, at the transmitter, it is important to know the delay bits when the TICK_IN signal is received. This can be accomplished by setting a down-counter to 10 every time a character starts transmitting. With each bit that is transmitted the value of the counter is decreased. When the TICK_IN arrives the value of the counter represents the number of pause bits that need to be added.

If the stored value of the counter is more or equal to 8 and the link speed is less than 12.5 Mb/s a NULL character shall be transmitted.

5.6 Conclusion

Current SpaceWire standard assures time information distribution with microseconds accuracy. This represents a drawback since intra-module events are timestamped with a time resolution in the order of nanoseconds. Furthermore the delay over SpaceWire networks does not have a constant value. In this chapter we have studied the SpaceWire timing synchronization to find possibilities for improvements. We have proposed three methods that can reduce the delays associated with skew and jitter in order to achieve nanoseconds accuracy. The changes required in order to implement these methods for each network component were also presented.

A summary of the work including general conclusions and future research directions is given in the next chapter.

6 Conclusion

The work performed for this thesis is related to time synchronization and data acquisition systems in space instrumentation. We targeted a SoC architecture in order to assure that it fits in the current space instrumentation miniaturization trend. Our SoC implementation was focused on FPGA based instruments as they offer the possibility to reconfigure the hardware that flies in space.

As outlined in the introduction, there are different motivations for this thesis, one of them being the current need to perform data analysis on-board. This is required because the slow down link from spacecraft to Earth does not cope with the amount of data that has to be transferred. As the major part of all these data is actually composed by noise, there is an obvious need to discard it so that only meaningful information is processed on-line. Form what remains it would be at the same time important to keep only the data that is relevant to the research that is performed. Therefore, we proposed the usage of trigger systems as a solution to these problems.

During the first part of the document we have described the principles, the design and the implementation of an AMBA based IP core to be used in trigger logic systems. The block is highly customizable and acts as a higher level trigger in a hierarchical trigger system. It accepts a number of low level triggers as inputs and based on the configuration that is set it asserts the final trigger. The moment in time when this operation is performed is also made available in order to be able to reconstruct the environment in detail.

From the tests that we have performed an average dead time of 1.5 μs was obtained. From here we conclude that the maximum frequency with which our IP core generates a trigger, on the current hardware implementation, is 0.65 MHz. This value can be increased by using a faster FPGA.

Even though we have included our designed trigger IP block only in one instrument for testing purposes, we state that there are plenty of other instruments in which it can fit easily.

Our approach highlights the potential benefits of using reconfigurable hardware in space. SoC have a major role in space electronics as they allow higher level of integration, by compacting multiple functions into a single chip. They increase the re-usability of the design while at the same time decrease the development time. This leads to an overall cost reduction.

The scientific value of a multi-functional instrument increases if one can correlate the results from one detector with the results from another one. For this purpose we have studied the SpaceWire mechanism used to distribute time across the network, by means of time codes. Possible solutions to improve the current mechanism were presented, in an effort to reach nanoseconds timing accuracy. As SpaceWire networks have static topologies it is possible to eliminate most of the jitter and skew delays, leaving only the

processing and cable delays. Nevertheless some of the methods require additions/modifications to the current SpaceWire standard.

We conclude that all of the thesis objectives were covered. We continue to present next some problems that we encountered during our work and in the last section of this thesis we propose some future developments.

6.1 Problems Encountered and Lessons Learned

One important problem that we had encountered appeared because we did not use any revision control software. As many times we had to go back to previous versions it was difficult to find the one we were looking for from the multitude of older versions. Therefore, we recommend the usage of any revision control software available, like for example Git, CVS or SVN.

We have tried to maintain the VHDL code as clean and generic as possible. For this we avoided the addition of manufacturer specific macros up to a certain limit. The only solution to this would require some standardization process, where the manufacturers would agree on a common way of providing the underlying resources.

The addition of new IP cores, or combining cores from different sources is not a straight process neither. Sometimes the designer has to go deep into the code of some cores which is written by others to track compatibility issues. GRLIB offers some automated tools but in order to include external cores modifications has to be performed over some specific files.

Another problem which unfortunately has no solution at the moment is the total time that is required for synthesis, mapping, placement and routing, in order to generate a new bitstream. This limits the designer to test only a few changes per day, if we take into considerations also the time required to implement those changes.

6.2 Recommendations for Future Work

The outcome of our work is promising but there is always space for improvements. The following section proposes some directions for future research.

One such direction regards the dead time, which comes from the AMBA communication and from the processor. We have performed some basic study but we do not know how the presence of other IP cores would influence the value of the dead time. Furthermore we have only used the APB bus. A more detailed study is needed to determine whether some dead time reduction appears if the the bus protocol is changed, or if the software that runs on the processor is modified.

The current method that is used to signal to the processor when the final trigger is asserted can be also improved by using interrupts, instead of just raising a bit of a certain register. Maybe this method will have also some impact over the dead time, but some other disadvantages might arise as well.

Another improvement for our IP would be to give feedback to the user on how to adjust the size of the coincidence window, to sense if the coincidence window is too big or too small. In order to achieve this, some statistical information need to be processed,

which has to be performed in hardware. Nevertheless, providing such guidance to the user would be useful as in certain experiments it might be difficult to make the proper setting of the coincidence window size.

Regarding SpaceWire, we have performed only a theoretical study that indicates that there is space for timing accuracy improvements. Our study suggests that nanoseconds accuracy can be obtained. It is however necessary to check the portability of these results to real life scenarios. Some hardware implementations for different SpaceWire cores must be modified for that and some networks need to be created and checked. This is important for intermodule timing accuracy.

Bibliography

[1] *BELLE at KEKB*, [Online]. Available: http://belle.kek.jp/.

[2] *cosine Research*, [Online]. Available: http://www.cosine.nl/.

[3] *ECSS-E-50-12C (31 July 2008) SpaceWire engineering Links, nodes, routers and networks.*

[4] *Fermi National Accelerator Laboratory - TEVATRON department*, [Online]. Available: http://www-bdnew.fnal.gov/tevatron/.

[5] *Gaisler Research GRLIB Library and LEON3 Synthesizable Processor*, [Online]. Available: http://www.gaisler.com/.

[6] *GHDL, a Complete VHDL Simulator*, [Online]. Available: http://ghdl.free.fr/.

[7] *GTKWave viewer*, [Online]. Available: http://gtkwave.sourceforge.net/.

[8] *HERA Accelerator*, [Online]. Available: http://adweb.desy.de/mpy/hera/.

[9] *LHC - The Large Hadron Collider*, [Online]. Available: http://lhc.web.cern.ch/lhc/.

[10] *Pender Electronic Design*, [Online]. Available: http://www.pender.ch/.

[11] *Timing, Trigger and Control (TTC) Systems for the LHC*, [Online]. Available: http://ttc.web.cern.ch/TTC/.

[12] *Xilinx, DS099 - Spartan-3 FPGA Family: Complete Data Sheet*, [Online]. Available: http://direct.xilinx.com/bvdocs/publications/ds099.pdf.

[13] M. Alderighi, *Mitigation of SEUs Affecting Configuration Memory and Reconfiguration Logic in VIRTEX II FPGAs*, Tech. report, IASF-INAF, 2009.

[14] B. M. Cook, *Reducing spacewire time-code jitter.*, International SpaceWire Seminar, ESTEC Noordwijk, The Netherlands (2003).

[15] P. Ferrando and al., *Simbol-x: mission overview.*, SPIE, 2006, p. 6266.

[16] R. Fruhwirth, M. Regler, R. K. Bock, H.Grote, and D. Notz, *Data analysis techniques for high-energy physics*, 2 ed., Cambridge University Press, 2000.

[17] S. Habinc, *Functional triple modular redundancy*, Tech. report, Gaisler Research, 2002.

[18] ———, *Suitability of reprogrammable fpgas in space applications*, Tech. report, Gaisler Research, 2002.

[19] N. S. Ivanova, Yu. D. Karpekov, V. A. Senko, and V. I. Yakimchuk, *A dedicated high-speed trigger generation processor for selecting single particle decay events using coordinate signals from hodoscopes of scintillation counters*, Instruments and Experimental Techniques **51** (2008), 381–391.

[20] Manfred Jeitler, *Trigger systems at LHC experiments*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **598** (2009), no. 1, 305 – 311, Instrumentation for Collding Beam Physics - Proceedings of the 10th International Conference on Instrumentation for Colliding Beam Physics.

[21] F. Karstens and S. Trippel, *Programmable Trigger Logic Unit Based on FPGA Technology*, IEEE Trans.Nucl.Sci.52:1192-1195,2005 (2005).

[22] H. J. Kim, S. K. Kim, S. H. Lee, T. W. Hur, C. H. Kim, F. Wang, I. C. Park, Hee-Jong Kim, B. G. Cheon, and E. Won, *A fast programmable trigger for isolated cluster counting in the belle experiment*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **457** (2001), no. 3, 634 – 639.

[23] S. Kraft, J. Moorhouse, A. L. Mieremet, M. Collon, J. Montella, M. Beijersbergen, J. Harris, M. L. van den Berg, A. Atzei, A. Lyngvi, D. Renton, C. Erd, and P. Falkner, *Study of highly integrated payload architectures for future planetary missions*, vol. 5570, SPIE, 2004, pp. 133–144.

[24] D. Lampridis, *High speed reconfigurable computation for electronic instrumentation in space applications*, (2007).

[25] P. H. W. Leong, *Recent trends in fpga architectures and applications*, Proceedings of IEEE International Symposium on Electronic Design, Test, and Applications, Jan. 2008, pp. 137-141.

[26] Volker Lindenstruth and Ivan Kisel, *Overview of trigger systems*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **535** (2004), no. 1-2, 48 – 56, Proceedings of the 10th International Vienna Conference on Instrumentation.

[27] Erik Maddox, Alex Palacios, Dimitris Lampridis, Stefan Kraft, Alan Owens, Dana Tomuta, and Reint Ostendorf, *Development of a multifunctional particle spectrometer for space radiation imaging*, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **591** (2008), no. 1, 121 – 124, Radiation Imaging Detectors 2007 - Proceedings of the 9th International Workshop on Radiation Imaging Detectors.

[28] Thilo Pauly and the Atlas Collaboration, *The ATLAS level-1 central trigger system in operation*, Journal of Physics: Conference Series **219** (2010), no. 2, 022017.

[29] D.G. Perry and L.P. Remsberg, *Particle identification with very thin transmission detectors*, Nuclear Instruments and Methods **135** (1976), no. 1, 103 – 109.

[30] F. Pinsard and C. Cara, *High resolution time synchronization over spacewire links.*, Aerospace Conference (2008).

[31] Krzysztof T Pozniak, *FPGA-based, specialized trigger and data acquisition systems for High-Energy Physics experiments*, Measurement Science and Technology **21** (2010), no. 6, 062002.

Abbreviations

ADC	-	Analog-to-digital converter
AHB	-	Advanced High-performance Bus
ALICE	-	A Large Ion Collider Experiment
AMBA	-	Advanced Microcontroller Bus Architecture
APB	-	Advanced System Bus
ASIC	-	Application Specific Integrated Circuit
ATLAS	-	A Toroidal LHC ApparatuS
BCC	-	Bare-C Cross Compiler
CB	-	Coincidence Block
CPU	-	Central Processing Unit
CMS	-	Compact Muon Solenoid
DCM	-	Digital Clock Manager
DPU	-	Digital Processing Unit
DSP	-	Digital Signal Processors
DSU	-	Debug Support Unit
FSM	-	Finite State Machine
FIFO	-	First In First Out
FPGA	-	Field Programmable Gate Array
GCC	-	GNU Compiler Collection
HEP	-	High Energy Physics
SEU	-	Single Event Upset
IP	-	Intelectual Property
LFSR	-	Linear Feedback Shift Register
LHC	-	Large Hadron Collider
MPS	-	Multifunctional Particle Spectrometer
MPSTG	-	MPS Testing Generator
NIM	-	Nuclear Instruemntal Module
NO	-	No Ordering
NOWSS	-	No Ordering With Starting Sensor
SO	-	Specific Order
SoC	-	System-on-a-Chip
SpW	-	SpaceWire
UART	-	Universal Asynchronous Receiver/Transmitter
VHDL	-	VHSIC Hardware Description Language

List of registers

A

This is a list of all the registers implemented and used in the Coincidence Block.

A.1 Coincidence window register

Table A.1: Coincidence window register, offset 00001H

field	**Reserved**	**cwindow**
bits	31 to cw	cw-1 to 0
access	-	R/W

This register is used to store the size of the coincidence window. The number of bits used for this representation is specified before synthesis in a package.

A.2 Parameter[i] register

Table A.2: Parameter[i] register, offset for i=0 at 00110H

field	**Reserved**	**Delay**	**Order**	**State**	**Active H/L**
bits	31 to l2+1	l2 to l1+1	l1 to 3	2 to 1	0
access	-	R/W	R/W	R/W	R/W

For every sensor there is a register like this allocated, in which the configuration is stored. The value of l1 is determined from the value of $\log_2$(nrofsensors). The value of l2 is determined from cw.

[**Active H/L**] Specifies if the input is active high or low.

[**State**] Represents the sensor state, in the following manner:

Coincidence = 00
Anti-coincidence = 01
Coincidence check = 10

[**Order**] Specifies the position of the sensor when the operation mode is specific order.

[**Delay**] Specifies the number of cycles with which the sensor trigger will be delayed. From the first bit it can be determined if it should be considered or not.

A.3 Operation mode register

Table A.3: Operation mode register register, offset 00010H

field	**Reserved**	**Starting sensor**	**State**
bits	31 to l1+1	l1 to 2	1 to 0
access	-	R/W	R/W

As the name suggests this register holds the operation mode of the Coincidence Block.

[**State**] Represents the operation mode, in the following manner:

No order = 00
No order with starting sensor = 01
Specific order = 10

[**Starting sensor**] Holds the index value of the starting sensor.

Sample Embedded C Code

B

```
#define B_t_status_conf  0x80000500
#define B_t_status       0x80000504
#define B_t_ra_tick_val  0x80000508
#define B_t_ra_start_val 0x80000514
#define B_t_re_tick_val  0x80000520

#define B_cb_status_conf 0x80000400
#define B_cb_cwindow     0x80000404
#define B_cb_ordering    0x80000408
#define B_cb_status      0x80000414
#define B_cb_parameters  0x80000440

volatile unsigned int *t_status_conf  = (unsigned int *)B_t_status_conf;
volatile unsigned int *t_status       = (unsigned int *)B_t_status;
volatile unsigned int *t_ra_tick_val  = (unsigned int *)B_t_ra_tick_val;
volatile unsigned int *t_ra_start_val = (unsigned int *)B_t_ra_start_val;
volatile unsigned int *t_re_tick_val  = (unsigned int *)B_t_re_tick_val;

volatile unsigned int *cb_status_conf = (unsigned int *)B_cb_status_conf;
volatile unsigned int *cb_cwindow     = (unsigned int *)B_cb_cwindow;
volatile unsigned int *cb_ordering    = (unsigned int *)B_cb_ordering;
volatile unsigned int *cb_status      = (unsigned int *)B_cb_status;
volatile unsigned int *cb_parameters  = (unsigned int *)B_cb_parameters;

void load_test (unsigned int ra_threshold, unsigned int re_threshold){
  // resetting the testing component
  t_status_conf[0] = 2;
  t_status_conf[0] = 0;

  // setting the testing parameters
  t_ra_tick_val[0]  = ra_threshold;
  t_ra_tick_val[1]  = ra_threshold;
  t_ra_tick_val[2]  = ra_threshold;
  t_ra_start_val[0] = 0x0000;
  t_ra_start_val[1] = 0xF007;
  t_ra_start_val[2] = 0xFC07;
  t_re_tick_val[0]  = re_threshold;

  // resetting the CB
  cb_status_conf[0] = 2;
  cb_status_conf[0] = 0;

  // setting the CB parameters
  cb_ordering[0]    = 2;
  cb_cwindow[0]     = 1;
```

```
  cb_parameters[0] = 1;
  cb_parameters[1] = 9;
  cb_parameters[2] = 17;
}

void run_single_test (void){
  volatile int nr = 0;
  volatile unsigned int l_status, rout_tick, l_t_status, l_t_done;

  // empty data cache
  swift_flush_cache_all();
  l_t_status = t_status[0];
  l_t_done   = (l_t_status >> 4) & 1;

  // start the testing
  l_t_done          = 0;
  cb_status_conf[0] = 1;
  t_status_conf[0]  = 1;

  // determine the total number of triggers
  while (l_t_done != 1){
    // checking for a trigger
    rout_tick = (cb_status[0] >> 8) & 1;
    if (rout_tick == 1){
      nr++;
      // acknowledge that the trigger was read
      cb_status_conf[0] = 3;
      cb_status_conf[0] = 1;
    }
    l_t_done   = (t_status[0] >> 4) & 1;
  }

  // output the total number of triggers
  printf("\n%d", nr);

  cb_status_conf[0] = 2;
  cb_status_conf[0] = 0;

  t_status_conf[0] = 2;
  t_status_conf[0] = 0;
}

void run_tests(void) {
  volatile int i;
  volatile unsigned int re_threshold;

  printf("\n real threshold : ");
  scanf("%u", &re_threshold);

  for (i = 0; i<= 65535; i++){
    load_test(i, re_threshold);
    run_single_test();
  }
}
```