Master's Thesis

A Timing System Application using White Rabbit

Alexander Aulin Söderqvist
Niklas Claesson

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, January 2014.

LUND
UNIVERSITY

Master Thesis:
A Timing System Application using White Rabbit

Author:
Alexander Aulin Söderqvist
Niklas Claesson

Advisor:
Rok Tavčar

Examiner:
Joachim Rodrigues

Faculty of Engineering
Lund University

January 2014

Department of Electrical and Information Technology
Faculty of Engineering, LTH
Lund University
Box 118
SE–221 00 LUND
SWEDEN

This thesis is set in Adobe Garamond 12pt and Google Roboto,
with the LaTeX document preparation system.

Printed in Sweden
E-huset, Lund, 2014.

Abstract

In this work, two synchronization layers for timing systems in large experimental physics control systems were studied. White Rabbit (WR), which is an emerging standard, is compared against the well-established event-based approach. Several typical timing system services have successfully been implemented on an FPGA to explore WR's concepts and architecture, which is fundamentally different from an event-based one. The requirements for the implemented prototype were decided based on typical requirements of current accelerator projects and with regard to other parameters, such as scalability and commercial availability. The proposed design methodology and prototype demonstrate one way of deploying WR in future accelerator projects.

Acknowledgements

We wish to thank Associate Professor Joachim Rodrigues for finding this master thesis project, pushing us to apply and assisting us to ensure its completion.

We also wish to thank Cosylab, the accelerator team and in particular our advisors Rok Tavčar and Rok Štefanič for giving us valuable feedback on the implemented architecture and our conference submissions. Everyone at Cosylab have been more than friendly, made us feel like home and a part of the team. Furthermore, it was very rewarding to do a poster presentation regarding this work on the 14th International Conference on Accelerator & Large Experimental Physics Control Systems in San Francisco, California, USA.

We also acknowledge all the work put into the White Rabbit project by all the contributors, without them the project would not exist. Contributions have been committed by several organizations, such as CERN, GSI and companies, such as Seven Solutions. But, since it is an open source project there may have been contributions by individuals not associated with any organizations as well.

This is an interesting field of science and it is our hope that the collaboration between the faculty and Cosylab grows, so that more students will have the great experience of enjoying, working and living in Ljubljana.

Table of Contents

List of Figures

List of Tables

Abbreviations

ATOE	Absolute Time Of Execution
CERN	Organisation européenne pour la recherche nucléaire (European Organization for Nuclear Research)
CPU	Central Processing Unit
DIT	Digital Input Timestamper
DOC	Digital Output Controller
FAIR	Facility for Antiproton and Ion Research
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
FIFO	First in first out queue
GPS	Global Positioning System
GMT	General Machine Timing System
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
ITU	International Telegraph Union
ITU-T	ITU - Telecommunication Standardization Sector
LAN	Local Area Network
MRF	Micro-Research Finland
NTP	Network Time Protocol
NIC	Network Interface Card
PTP	Precision Time Protocol
RAM	Random access memory
RTOS	Real-time Operating System

RTOE	Relative Time Of Execution
SI	Le Système international d'unités (International System of Units)
TAI	Temps atomique international (International Atomic Time)
TMR	Timing Message Receiver
WR	White Rabbit
PLL	Phase-Locked loop
VCO	Voltage controlled oscillator

Chapter 1

Introduction

As materials scientists try to understand how nano-scale objects, for example molecules, look and behave, they require more specialized and powerful tools. This is one of the reasons to build the European Spallation Source (ESS). It will be the accelerator with most intense proton beam in the world at 5 MW [1], letting academia and industry investigate science with unbeatable results. One of the largest competing facilities is the Spallation Neutron Source (SNS), in Oak Ridge, Tennessee, USA, which is specified to deliver a beam of 1.4 MW [2].

Figure 1.1: *Overview of the ESS complex. Credit: ESS.*

Particle accelerators are also used in cancer treatment. It is still in a research phase, the scientists are evaluating which particles and energy levels are best suited [3]. Both proton and heavy ion therapy looks promising because of the "Bragg peak" phenomenon, which gives the physicians higher precision and less damage is dealt to healthy tissue around the tumour. It can therefore be used to treat cancers close to vital organs. Since it is still a young concept, more long term studies will have to be made to confirm that it raises the overall survival and the life quality.

1.1 Control System

In order to reach the specified maximum performance, a sophisticated control system is needed. This system need to be able to trigger accelerating and steering elements in a way so that the particles are able to obtain the required speed, which often is close to the speed of light. This is achieved with a combination of techniques and subsystems. There are predetermined execution sequences that are defined during commissioning and then there are extremely fast local control loops for correction of the beam. Overall the components have to work together like an orchestra. If a single crucial component fails, the whole machine is emergency shutdown.

Figure 1.2 shows the multitude of devices within a typical accelerator. The particles start in the linear accelerator to the left. They are then further accelerated in the circular synchrotron part, and in the end delivered to one of the four end stations to the right. Each device has to act at a specific time in order to reach fundamental requirements such as specific energy levels and speed of particles.

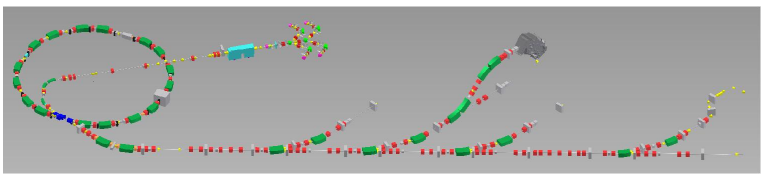


Figure 1.2: *Devices that need a control system. Credit: MedAustron.*

Some noticeable subsystems of the control system are the timing system, the machine protection system and the personnel protection system. The control system also needs logging systems, databases for collected data and graphical user interfaces for the operators. All aspects are usually extreme: huge bandwidth, copious amount of data, high frequencies, elaborate synchronization and many computer networks.

Usually all these subsystems have their own networks to interfere less, but to keep costs down, it is beneficial if some of them are able to share network. Until now it has been more or less impossible for the timing network to share its fibres with any other because they have often used non-standard protocols.

1.2 Timing System

A lot of devices used in an accelerator is timing sensitive and require elaborate synchronization. The system that handles this synchronization is called the

timing system and it is an important part of the control system. Different devices have different demand on precision of the synchronization and it also varies between facilities. Typically, they have to act with a precision in the range of μs, ns or ps.

Accelerators come in many different sizes and shapes and achieve acceleration in various ways. Linear accelerators, for example, shoot pulses of particles while circular accelerators have a period time. Due to this fact, all timing critical devices around an accelerator have to be triggered sequentially. To achieve this, the predetermined sequences consists of a set of actions, each notifying devices of when and what to carry out. Large accelerators can contain multiple linear and circular parts, each having unique sequences which need to be carried out. They often run simultaneously and when transferring particles between the different parts, the sequences also depend on each other.

As timing sensitive equipment is becoming more common, there is a desire to avoid reinventing the wheel. Progress is being made on unifying timing systems by making common hardware platforms. These platforms need to be compatible with a lot of existing backbones to unite hardware from different vendors. To enable this, the convention is to include configurable hardware, such as field-programmable gate arrays (FPGA), which provide the timing system designer with flexibility to adapt the platform to specific requirements.

In current facilities it is common to send the same timing signal over fanned out fibre to all the devices, therefore making it real-time and deterministic. The drawbacks with this method are that you require equal delay to all devices and that you use the whole network as a single connection. These two factors make it inconvenient to use the network for data transfers; you cannot send data between two arbitrary nodes. Also, since there is no feedback loop, it is difficult to compensate for signal delay variations due to environmental changes.

The most timing critical projects are synchrotron accelerators and laser facilities. They can require that the timing system is synchronized down to tens of picoseconds. Then there are projects accelerating larger particles, which require synchronization to a nanosecond. There are even completely unrelated projects like antenna or detector arrays that also benefit from a timing system with high accuracy, since they do distributed time stamping of scientific measurements.

Amongst the large science projects there are a few which are exceptionally large. They are constructed using multiple accelerators and accumulators. In these huge facilities, scalability is also an essential aspect of the timing system.

Usually, timing systems have unique, machine-specific, requirements and are therefore developed completely or partly in-house, without any thought on standardization. This has made it difficult to collaborate and share knowledge

between organizations in the accelerator community. The same functionality has often been implemented in slightly different ways using either in-house developed hardware or commercial of-the-shelf customizable hardware.

The major problem with in-house developed systems are that they can contain poorly documented proprietary hardware, protocols and software, making it problematic for one party to extend and improve the work of someone else. All parties are often required to sign non-disclosure agreements to collaborate and share information.

1.3 White Rabbit

White Rabbit [4] is the first attempt to solve the timing critical challenges using only open standards, open software and open hardware. The project's initial goal was to be the basis for a replacement of the current timing systems at CERN, but now it has the possibility to become the *de facto* standard platform for timing systems. The projects logotype can be seen in Figure 1.3.

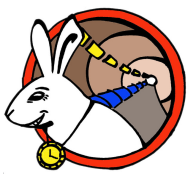


Figure 1.3: *The White Rabbit logotype.*

It has an unprecedented requirement of scalability because its requirements are distilled from all the systems it replaces. It will replace thousands of nodes with up to several kilometres distance in between. This gives it its most noticeable characteristics, namely, automatic delay compensation for cable length. It is also understandable that Ethernet (1000BASE-BX10) was chosen as physical layer to make the system more generic than the current offerings.

1.3.1 Open Hardware

White Rabbit is unique in its approach to hardware, since everything developed is available for free through an on-line repository [5]. This allows a new form of collaboration were the developers can focus their contribution to where they are most proficient. This is meant to be beneficial for smaller companies with narrower expertise, because it lowers the initial investments.

Open hardware is still a rare concept. Companies that develop hardware are reluctant to share their work because they need a return on investment. This, however, does not apply to publicly funded institutions, which usually want to transfer their gained knowledge back to the public. Therefore, they can reap the benefits from open source without the drawbacks. There are, for example, already several manufacturers producing White Rabbit reference hardware, competing with price and quality.

It is also a bit unexplored territory regarding patents and the law, which is preventing acceptance. If a medical device, for example, would malfunction and harm patients, it is not clear how bears the responsibility. Hardware companies conventionally patent as much as possible and open sourcing makes it easier for the patent holders to find unintentionally infringing technologies.

1.3.2 Standards

Complying with standards has shown to be very beneficial. Because the network uses standard Ethernet it is possible to use a variety of tools to analyse the traffic. By settling with precision time protocol (PTP) as synchronization protocol, it is possible to be compliant with non-White-Rabbit hardware, only degrading that link to regular PTP accuracy. White Rabbit has proved to be one of the best PTP implementations on several PTP compatibility meetings.

Complying with standards also has its drawbacks. Packaging everything in Ethernet frames adds overhead, which makes White Rabbit unsuitable for some use cases. If really low latency links are required White Rabbit cannot be used, due to the delays introduced in overhead and routing.

1.4 Scope

In this work, a prototype timing system has been implemented to understand and explore the potential of White Rabbit. White Rabbit is a new state-of-the-art way of synchronizing time and clock on multiple FPGAs to sub-nanosecond accuracy.

The requirements for the prototype were devised after studying requirements of current big science projects in development and considering the hardware available. The prototype should also exploit White Rabbit to show its strengths and weaknesses.

1.5 Outline

Chapter 1

In chapter 1, an introduction to timing systems and an overview of current systems is given. The motivation behind White Rabbit is also explained.

Chapter 2

In chapter 2, the theory required to understand the report is explained, including synchronization and a general description of White Rabbit. Then the report proceeds by explaining what a timing system is and how it is defined. A more thorough overview of current timing systems is also here.

Chapter 3

The primary focus of the work is explained in chapter 3, were the implemented timing system prototype is explained in detail.

Chapter 4

In chapter 4, the results and verification of the timing system prototype are presented.

Chapter 5

Finally the report is concluded with discussion and future work in this chapter.

Chapter 2
Background

This chapter goes through definitions together with theoretical background. It also covers existing methods of synchronization in timing systems.

2.1 Definitions

In this section fundamental definitions necessary to understand synchronization will be explained.

2.1.1 Clock versus Time

The difference between clock and time is essential to timing systems. Clock is, in this report, primarily an electrical signal with a given frequency. Two clocks with the same frequency will have a phase difference.

Claiming that multiple digital devices have the same clock means that their clocks have the same frequency and that the phase difference is small enough to meet the requirements. Even if they have the same clock, they still need some way of deciding which rising edge is equivalent everywhere. Therefore time needs to be synchronized, i.e., they need to agree on how long time has passed since a defined moment.

Time is represented as seconds. Clock cycles are counted in between seconds to get better granularity. Every device therefore has to keep track of two values, seconds and clock cycles.

2.1.2 International Atomic Time

International Atomic Time (TAI) is a standard way of representing time defined by the standards institute SI. It is calculated as a mean of about 200 atomic clocks around the world. TAI is possible to obtain with extreme precision using high-end GPS equipment; therefore it is possible to synchronize different sites throughout the world down to tens of nanoseconds [6].

2.1.3 Accuracy and Precision

Typical requirements on synchronized distributed outputs are the alignment amongst the nodes. This is measured using accuracy and precision, see Figure 2.1, where accuracy is the mean offset from the reference signal and precision is the jitter, or deviation, of the offset.

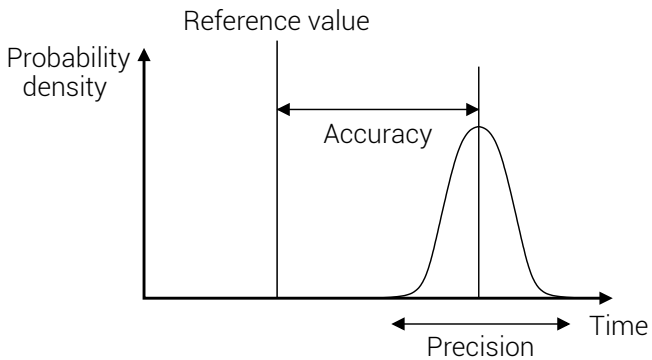


Figure 2.1: *Accuracy and precision.*

2.1.4 Determinism

Determinism is a philosophical concept, where every action produces a known reaction and the systems input, output, and states always are known. If a system is fully deterministic, nothing is left to chance and everything behaves like expected.

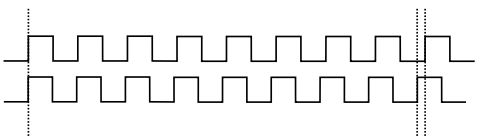
This is desirable in digital designs, because then actions are predictable and repeatable. The complexity in many systems is minimized to achieve better determinism. Instead of implementing algorithms in software, hardware implementations are used. Instead of a complicated network protocol, only predetermined patterns are sent. But, this also makes the systems more specialized and less generic.

2.2 Synchronization

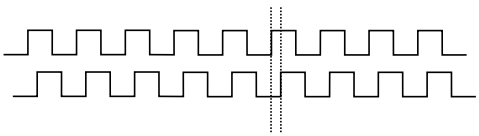
Two different approaches to synchronization in timing systems have been identified [7, 8]. Both use phased-locked loops and 8b/10b encoding for clock recovery, which are essential techniques in synchronization. The first one, which is more common, is called event-based and only synchronizes clock, whereas White Rabbit also synchronizes time. This section will go through the different methods that are used to achieve synchronization in regard to both clock and time.

2.2.1 Synchronizing clocks

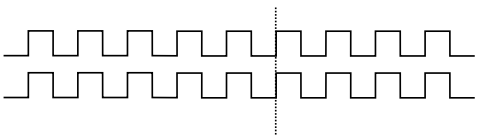
Synchronization of clocks requires two steps, which can be seen in Figure 2.2. The first process is syntonization, where the clocks are adjusted to the same frequency. The second part is the measurement and alignment of the phases.



(a) *Clock drift is resolved by syntonization.*



(b) *Phase offset is resolved by phase alignment.*



(c) *Synchronized!*

Figure 2.2: *Synchronization is achieved through two processes, syntonization (a) and phase alignment (b).*

2.2.2 Phased-Locked Loop

A phase-locked loop (PLL) consists of a phase & frequency detector (PFD), a loop filter and a voltage controlled oscillator (VCO), see Figure 2.3. Its purpose

is to lock the phase of the generated output frequency (F_{out}) to the phase of an input frequency (F_{ref}). The implementation of a PLL differs a lot depending on application. They often include scale factors to increase or decrease the frequency. It can accomplish both syntonization and phase alignment. The following basic explanation of the concept is enough to understand synchronization.

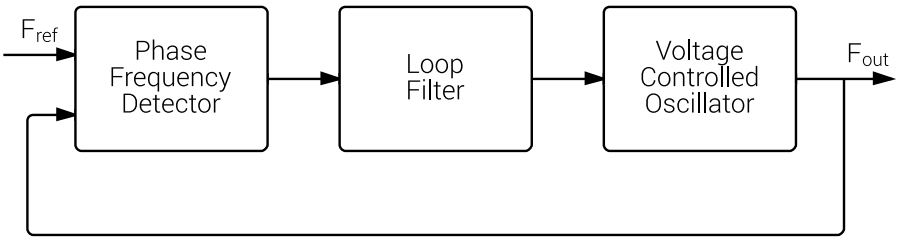


Figure 2.3: *Block diagram of a general phase-locked loop.*

The PFD's purpose is to generate a control signal for the VCO. The loop filter keeps the system stable when, for example, changes in F_{ref} occur and on start up.

Using PLLs it is possible to produce a lot of different frequencies. This is often used in integrated circuits to multiply, divide and/or shift the phase of the input clock.

2.2.3 Syntonization with fibre

One common way of syntonizing slaves in regard to the master is to use the 8b/10b encoding [9]. This encoding ensures that there are enough transitions to correctly recover the clock by never letting there be more than 5 consecutive ones or zeroes. In Figure 2.4 a block diagram shows how it is connected. Data is transmitted serially over an optical link using 1.25 GHz. Every 8 bit data is converted to one of two 10 bit symbols, see Table 2.1. Some of the remaining 10 bit symbols are used for link maintenance.

Data	Symbol (+)	Symbol (−)
000 00000	100111 0100	011000 1011
000 00001	011101 0100	100010 1011

Table 2.1: *Excerpt from 8b/10b conversion table.*

The 10 bit symbols differ in number of ones. The scheme selects one of the 10 bit symbols to ensure that there is no DC offset, i.e., same number of zeroes and ones over a longer period of time. The *comma* maintenance symbol allow the

process to align the data stream, therefore it cannot be present anywhere else in the stream.

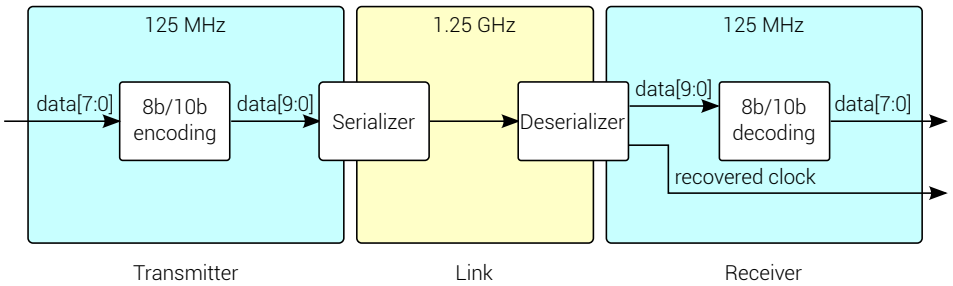


Figure 2.4: *Block diagram of clock recovery.*

2.2.4 Phase detection

To compensate for phase offsets a digital equivalent of a dual-mixer time difference (DMTD) is used in White Rabbit [10]. The DMTD is a system that compares two clocks using a third, slightly out of frequency, clock. Imagine two clocks ($a(t) = \cos(2\pi t F_{\mathrm{clk}} + \Phi_{\mathrm{a}})$ and $b(t) = \cos(2\pi t F_{\mathrm{clk}} + \Phi_{\mathrm{b}})$) with the same frequency, F_{clk}, and amplitude. The third clock will have frequency F_{offset}, $c(t) = \cos(2\pi t F_{\mathrm{offset}} + \Phi_{\mathrm{c}})$. Then both clocks are multiplied with the third as:

$$\begin{aligned} a(t) \cdot c(t) &= \cos(2\pi t F_{\mathrm{clk}} + \Phi_{\mathrm{a}}) \cdot \cos(2\pi t F_{\mathrm{offset}} + \Phi_{\mathrm{c}}) \\ &= \frac{1}{2}(\cos(2\pi t (F_{\mathrm{clk}} + F_{\mathrm{offset}}) + \Phi_{\mathrm{a}} + \Phi_{\mathrm{c}}) + \\ &+ \cos(2\pi t (F_{\mathrm{clk}} - F_{\mathrm{offset}}) + \Phi_{\mathrm{a}} - \Phi_{\mathrm{c}})) \end{aligned}$$

The results of both multiplications are two clocks, one with high and one with low frequency. By low-pass filtering the result, it is then possible to study the low frequency signals by counting pulse length using F_{clk}. If the offset clock is very close to the input clock, better accuracy is achieved. Doing this with two clocks in parallel enables phase detection with a counter because the mixing only affects the frequency and not the phase.

In White Rabbit the circuit is implemented with digital equivalents to the analog components, for example, registers are used as mixers.

2.2.5 Event-Based Synchronization

This synchronization approach is called event-based, since the protocol for communication between the master and receiver nodes use identifiers called events. All the receivers receive exactly the same signal at the exact same time. Therefore there is no need to synchronize time. Because of syntonization, all the receivers have the same clock and since they all have the same delay, they all have the same phase difference to the master, which is derivable from cable length.

The most straight-forward way of achieving syntonization is to recover the master's clock with a PLL in the receiver nodes FPGA. Because all receivers have the same phase, they are synchronized, but not with the master. The clock can be recovered with high precision by using optical serial transmission running at 10 times higher frequency than the FPGA. The optical transmission also has the positive side effect that they are less sensitive to interference and radiation.

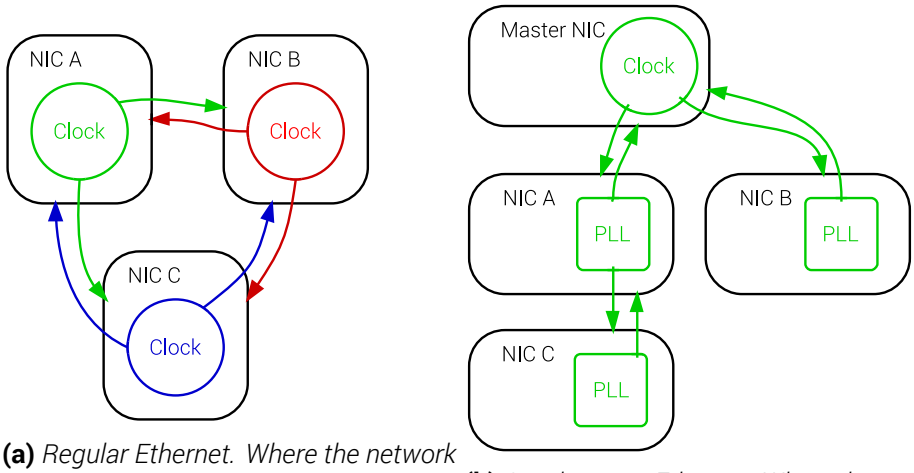
There are multiple proprietary solutions using this technique. Some vendors allow customization of firmware and in certain cases the vendors are even compatible.

2.2.6 Synchronous Ethernet

Synchronous Ethernet (SyncE) is a specification for frequency transfer over Ethernet, standardized by International Telecommunication Unions (ITU) standardization unit Telecommunication Standardization Sector (ITU-T). It is essentially the same as event-based synchronization, since it synchronizes on the physical layer.

The requirements listed in SyncE are, amongst other things, that each node should be clocked with a traceable reference clock. This is achieved via syntonization on the lowest layer and it is therefore independent of the network load. The clock is traceable since every node uses the recovered clock for transmission to other nodes. Regular Ethernet also recovers the clock, but only for the incoming transmission, limiting the synchronization to the first layer of connected nodes. SyncE therefore requires a common reference clock, which leads to a hierarchical tree network with a primary reference clock at the top.

The difference between using a free running oscillator and a recovered clock as transmission clock is shown in Figure 2.5. In Figure 2.5a the network interface cards (NIC) do not use the recovered clock as transmission clock, therefore they are not synchronous. On the other hand, in Figure 2.5b, the NICs use the recovered clock as transmission clock, hence the clock is the same in all nodes and they are synchronous.



(a) *Regular Ethernet. Where the network interface cards are using free running oscillators as transmit clocks.*

(b) *Synchronous Ethernet. Where the network interface cards are using recovered clocks as transmit clocks.*

Figure 2.5: *The different colours symbolize different transmit clocks that are used in regular and synchronous Ethernet.*

2.2.7 Time Synchronization

Another approach, different to the event-based, is to synchronize time in all the nodes in the network. There are several existing Ethernet protocols which do this. The most common are the Network Time Protocol (NTP) and the Precision Time Protocol (PTP). Both synchronize time by calculating the offset between the synchronizing node and a reference node.

NTP is designed for use over the Internet. It is therefore possible to implement completely in software and the synchronization is initiated by a client, which contacts one of several publicly accessible NTP servers. PTP, on the other hand, is designed for use on a segment of a LAN. The synchronization is initiated by a master and it will continuously make sure that all the slaves are synchronized. PTP also requires specialized hardware, especially better clocks, for precise time stamping.

NTP accomplishes accuracy from a couple of microseconds on a LAN and typically tens of milliseconds when used over Internet. PTP can synchronize time in the order of hundreds of nano seconds or even better depending on hardware.

Errors in NTP and PTP arise from multiple sources, for example, precision in time stamping, buffering, congestion, routing and different time stamping

strategies.

For further reading about time synchronization and especially NTP see [11].

2.2.8 Precision Time Protocol

Precision Time Protocol (PTP) (IEEE 1588-2002[12]) is designed to synchronize clocks with high precision on a segment of a LAN instead of over the Internet as NTP. It can, for example, be used for industrial control systems where microsecond accuracy is good enough.

PTP synchronizes time by exchanging timestamps back and forth between the master and the slave, see Figure 2.6.

The goal of the algorithm is to calculate the time offset (θ) between the master's clock and the slave's clock and the round-trip delay (δ) back and forth between the nodes. The first message, *Sync*, is sent from the master to the slave and is time stamped both on transmission and reception. The following message, *Follow Up*, contains t_1. The slave sends a *Delay Request* message which also is time stamped at both ends. Finally the master sends a *Delay Response* carrying t_4. t_1 and t_4 are captured by the master's clock and t_2 and t_3 are captured by the slave's clock. After this procedure the slave has acquired all the four timestamps.

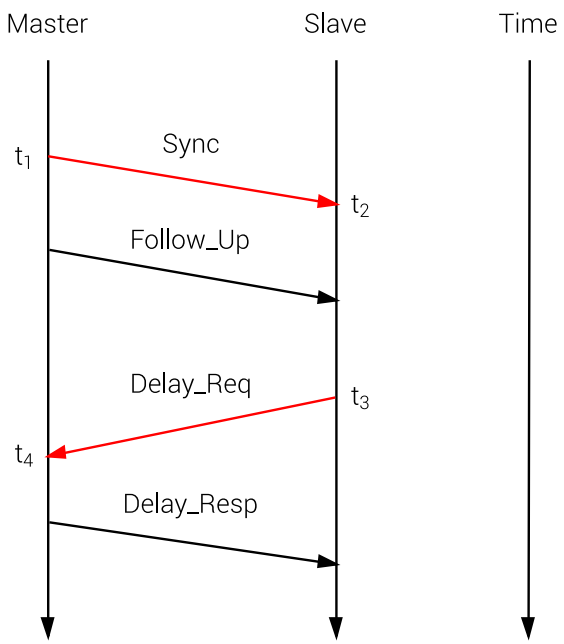


Figure 2.6: *PTP time synchronization process.*

The round-trip delay and clock offset is calculated as:

$$\delta = (t_4 - t_1) - (t_3 - t_2) \tag{2.1a}$$

$$\theta = \frac{1}{2}[(t_2 - t_1) + (t_3 - t_4)] \tag{2.1b}$$

The latest version, PTPv2 (IEEE 1588-2008[13]), improves the accuracy, but is not backward compatible and delivers sub-microsecond accuracy. This is still not good enough for particle accelerators, which require higher accuracy.

The precision of the algorithm is mostly affected by the precision of the timestamps. High-end PTP equipment therefore implements timestamps in hardware as close to the transmission medium as possible.

2.3 Synchronization Layer

To create a fully functional timing system it is preferable to use commercial off-the-shelf products instead of putting effort into designing custom hardware. This hardware should enable synchronous action and is called synchronization layer. Every timing system project is unique and therefore different kind of customization need to be applied. It is important to understand how the synchronization layers work and what their limitations are to make the right decision of which one to use.

This chapter presents both MRF and White Rabbit as synchronization layers. They use FPGAs, which enables customization, and are commercially available.

2.4 Timing System Model

The timing system's purpose is to provide services to the control system, see Figure 2.7. Synchronization through the network is needed to implement them. The synchronization layer provides basic synchronization capabilities, which enables the design of certain timing system services.

The most basic timing system is a network that consists of a timing master and several timing receivers [14]. The timing system network can be divided into sub-networks depending on the complexity of the requirements. Each sub-network has a local timing master and the total number of timing receivers can be hundreds or even thousands.

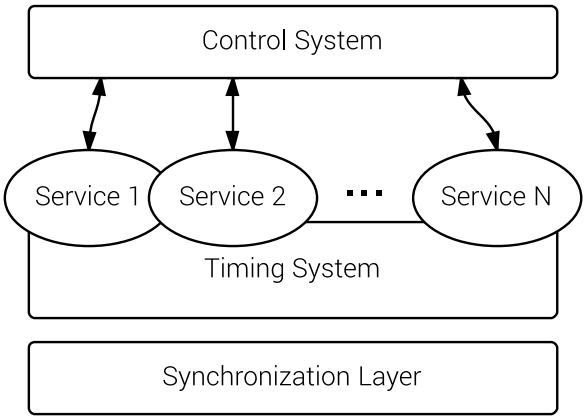


Figure 2.7: *The timing system is the link between the control system and the synchronization layer.*

The main purpose of the timing system is to provide services to the control system. Requirements on timing systems for accelerators are often tough and require real-time applications. The requirements are different for all machines, which mean that there is not always a commercial off-the-shelf product available that fits the needs.

This section will go through concepts based on current conventions. These concepts are heavily inspired by current timing systems based on the MRF synchronization layer and the ongoing General Machine Timing System (GMT) at Facility for Antiproton and Ion Research [15].

2.4.1 Timing Master

The timing master is a node in a timing system. It dictates to every underlying node what to do and, more importantly, when to do it. The timing master is responsible for several crucial tasks, some of which are: keeping the timing receivers synchronous, synchronizing the network to external triggers and emitting sequences of instructions.

The timing receivers are synchronized in different ways depending on the synchronization layer. To dictate the timing receivers, the timing master has access to sequences, which it plays over the timing network. The sequences consists of a set of trigger instructions, see Table 2.2. Depending on the size of the machine the scheduling problem grows or shrinks. There can also be interdependencies between machines when transferring bunches of particles, which further complicates it.

0	Trigger instruction 1
1	Trigger instruction 2
2	Trigger instruction 3
...	...
N	Trigger instruction N+1

Table 2.2: *A sequence consists of multiple trigger instructions which are played over the timing network.*

2.4.2 Timing Receiver

The timing receivers are specialized hardware devices that have the ability to synchronize with the timing master. Timing receivers are spread out over the facility and positioned near the front end devices. They are usually not completely stand alone and are often placed in a host called front end controller together with other necessary equipment. They can have a range of different interfaces for controlling the front end devices. From the simplest, in form of digital– and/or analog outputs and/or inputs, to more sophisticated clock and function generators.

In addition to synchronize with the timing master, they also receive trigger instructions. The trigger instructions must provide the necessary data to the receivers of when and what to carry out.

2.4.3 Timing System Services

The functionality that the timing system provides to the control system can be seen as a set of services. These services range from fundamental, like triggering of front end devices and time stamping inputs, to more complex functionality.

Trigger and time stamping services is the minimum required of a timing system and will be presented in this section. Since high level services are derived from machine unique requirements, they are out of scope for this thesis.

A minimal example of a simple timing system can be seen in Figure 2.8. It illustrates the timing system and its interface to the front end devices. The interface between the timing system and control system can be seen in Figure 2.9.

Real-Time triggering

An accelerator consists of hundreds or thousands of distributed front end devices that need to be controlled in real-time with high resolution. Precisely timed

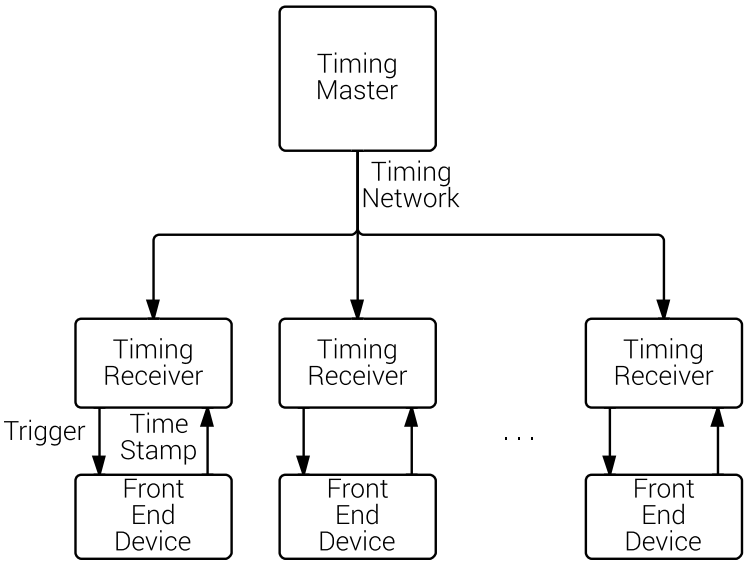


Figure 2.8: *The timing system consists of at least one timing master and multiple timing receivers. The timing master plays the sequence on the timing network to the timing receivers. Each timing receiver is connected to a front end device which it triggers and/or receives events from to timestamp.*

outputs are required around the machine to do so satisfyingly. This output functionality is the most prominent feature in the timing system and is located in the timing receivers closest to the front end devices. The outputs on the timing receivers are configured individually in advance by the control system to act differently on each trigger instruction received.

The configuration of what to carry out on each trigger instruction is dependent on the timing receiver's output functionality. On a single digital output, for example, there are at least the possibility to define a delay after which the output start and length of the output. With more sophisticated functionalities come different kinds of configuration.

Timestamp external events

It is important to know the status of equipment around the accelerator to diagnose functionality and to detect potential malfunctions. Logging events from the front end devices during run-time is another fundamental functionality for the timing system. Therefore the timing receivers need to be able to timestamp external events. The timestamps are then made available to control system, see Figure 2.9.

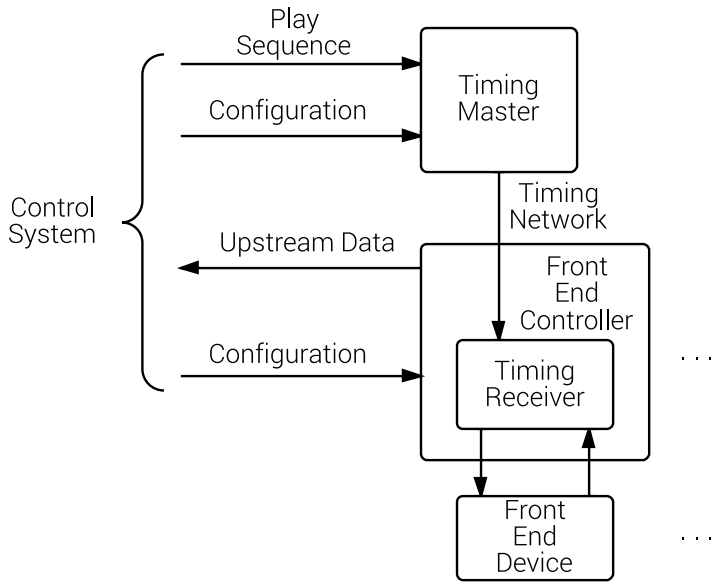


Figure 2.9: *The timing system is configured by the control system. The control system informs the timing master on when and what sequence to play over the timing network.*

2.5 Micro-Research Finland

Micro-Research Finland [7] (MRF) is an event-based system and has similar specification to SyncE, except that it does not use Ethernet. This means that the system is synchronous, in other words, that every node runs on the extracted clock from the fibre. Phase alignment is guaranteed by using the same length fibre or with manual delay compensation.

Everything in the master and receivers is implemented in hardware and only configuration is done from software to ensure easier verifiable determinism.

2.5.1 As Synchronization Layer

MRF uses a custom 2-byte protocol where 1 byte is sent every clock cycle, as can be seen in Figure 2.10. The first byte defines the event and the second byte is called payload. Each event received by the Timing Receiver is immediately translated through an event table and carried out. The payload can be used for custom configuration.

The event and custom data can be seen as its interfaces. If a timing system is developed, you are restricted to communicating with all the receivers at the same

time and only using this event code and custom data.

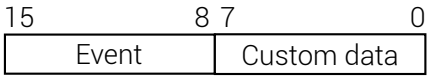


Figure 2.10: *MRF 2-byte protocol.*

Primarily, MRF is used for transmission in one direction, to avoid any risk of transmission collisions, but it is also possible to transmit upstream on a separate link.

An MRF network is a strict hierarchical tree structure seen to the distribution of events. Events are generated by a single node at the top and transmitted downwards through fan-outs [16]. The fan-out merely converts the optical signal to an electrical signal and routes it to 10 optical output transmitters. This enables high precision and low latency distribution of the signal since no data processing is being done.

There are uplink capabilities by using so called concentrators [17]. The concentrators have to prioritize between the uplinks, which makes it less deterministic than the fan out but with a maximum latency, around 200 ns, depending on set-up.

The performance limitations of MRF lie in the difference in delay between the nodes. In reality this depends on many factors not only fibre length or inserted delays. The effect from varying temperature on fibres and transceivers has significant impact on propagation speed. Hence degrading performance if the temperature varies in the facility where the system is installed.

2.5.2 As Timing System

MRF does not only provide a synchronization layer platform, but a complete timing system [7]. It has a firmware available for the on-board FPGAs, which provides a configurable timing system. It includes the fundamental functionality of real-time triggering and time stamping which will be explained.

The top node, corresponding to the timing master in the model that generates the events is called Event Generator [18]. The underlying nodes that are connected through fan-outs are called Event Receivers [19], coherent with the timing receiver definition. A minimal timing system with MRF can be seen in Figure 2.11.

The event generator has to 2 sequencers, each containing 2048 event. One sequence definition can be seen in Table 2.3. The event consists of a 32 bit timestamp and an 8 bit event code. The timestamp declares when the event

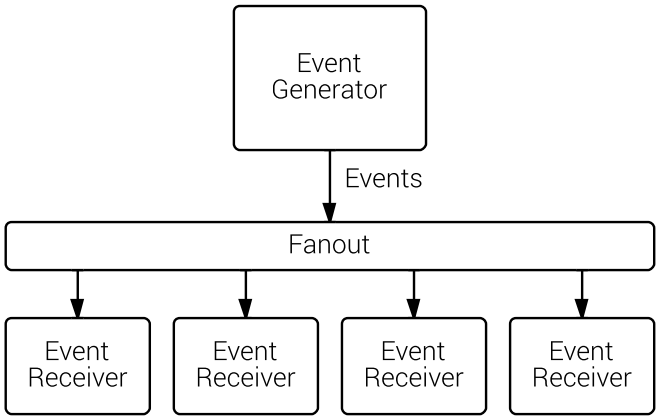


Figure 2.11: *An MRF network at its minimum consists of one event generator and a fan-out to the timing receivers.*

code is emitted on the network and is relative to the start of the sequence. The sequencer can be triggered via software, TTL inputs, AC mains synchronization or by configurable counters.

Event	32 bits	8 bits
0	Timestamp	Event code
1	Timestamp	Event code
...	...	...
2047	Timestamp	Event code

Table 2.3: *An MRF sequence consist of 2048 events, each define a timestamp and event code.*

There are $2^8 = 256$ events, whereof 246 are user definable. If no event is scheduled to be emitted, the null event is automatically emitted.

The event receiver has a number of configurable pulse generators; the amount depends on firmware and hardware version, and an event FIFO for timestamps. To enable these functionalities it has 2 individually configurable event RAMs, but only one is active in a given moment. Each RAM has 256 addresses with 128 bits words. Every time an event is received it is immediately mapped to the event RAM's address lines. The data is then outputted from the active RAM to configure the internal functions, including the pulse generators and time stamping functionality.

The pulse generators are controlled immediately upon a received event, see Figure 2.12, with set and reset from the event RAM, where set activates the pulse and reset deactivates the pulse. It can also be controlled from internal counters,

which define the delay until the pulse is activated and width of the pulse. These counters have to be preloaded from software running on the front end controller and are activated with the trigger signal from the event RAM. The counters count downwards with a configurable rate from a prescaler.

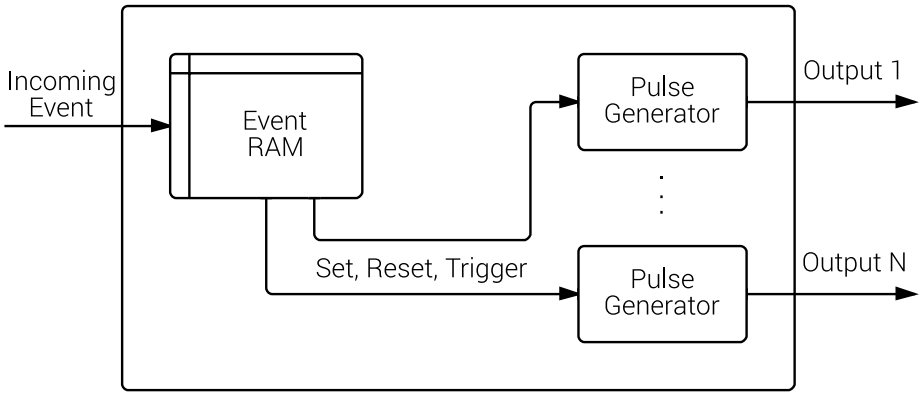


Figure 2.12: *The event receiver maps incoming event to its event RAM immediately and the data output configures the pulse generators.*

The system provides a common time base and each timing receiver has an event FIFO with room for 511 time stamped events. Each event received by the event receivers can be configured, in the event RAM, to be saved in the event FIFO. The receiver has inputs which can be configured to generate an event locally. Hence, these two functionalities can be combined to time stamp external events. The event FIFO is readable from the front end controller.

Subsystems may be required in big or complex timing systems. In MRF this can be achieved by cascading two subsystems with a simple system on top, which is done at, for example, KEKB in Japan [20]. The event generator can start a sequence on TTL inputs, by putting a single event generator connected directly to a timing receiver this can be used to trigger several event generators, see figure 2.13.

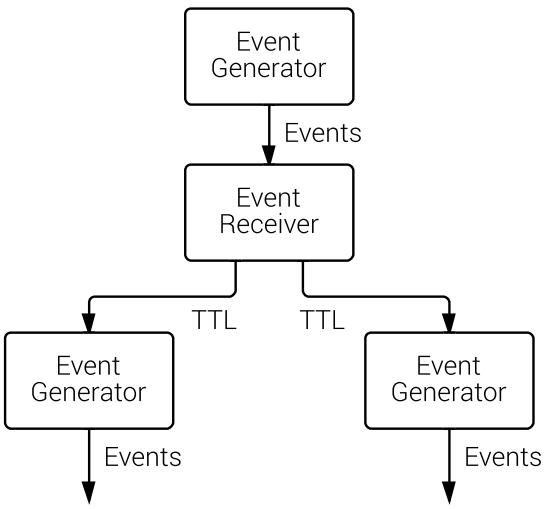


Figure 2.13: *One event generator together with one timing receiver can be used to trigger underlying event generators to build subsystems.*

2.6 White Rabbit

The White Rabbit project is an open-source hardware platform specifying an improvement to IEEE-1588 (PTP) which is called White Rabbit Precision Time Protocol (WRPTP). It also fulfils the specification of Synchronous Ethernet (SyncE) as given by ITU-T.

WRPTP improves PTP with several key features and solves most of the accuracy issues. It does this by, for example, having very deterministic time stamping and, in addition to time, also synchronizing the clock. The clock is synchronized to achieve sub-clock-period accuracy, which is impossible with regular PTP. Using a period of 8 ns gives sub-nanosecond accuracy. Other inaccuracies which occur due to buffers and router delays are removed by restricting the network topology.

2.6.1 As Synchronization Layer

Existing hardware implementations of WRPTP uses Ethernet on optical fibre, 1000BASE-BX10, which enables ranges up to 10 km. Each node has its own phase and frequency locked oscillator, which is synchronized with the clock extracted from the fibre. The phase difference between the extracted clock and the oscillator is continuously measured and compensated for. The continuous synchronization, inherited from PTP, compensates for run-time variations of the propagation delays between nodes. The complete White Rabbit PTP implementation is built into the hardware and hidden from the host CPU. White Rabbit aims to be part of the next PTP revision. For specifics on White Rabbit's synchronization see the specification [21].

The network hierarchy is limited by the distribution of the reference time, which is done by the clock master. In White Rabbit the clock master is a multi-port node with the possibility to synchronize its time with a GPS signal or an atomic clock. The clock master then performs the necessary synchronization with the underlying nodes (switches or receivers).

The performance of White Rabbit varies depending on the set-up. An advantage is that it compensates for fibre delay and therefore also the varying temperature on the fibre. It aims to have sub-nanosecond accuracy which it achieves with different results depending on set-up. Performance tests done in climate chambers proofs that this works [22]. The current performance limitation comes partly from delay variations, caused by change of temperature, in the White Rabbit nodes themselves [23].

2.6.2 The General Machine Timing System for FAIR and GSI

The General Machine Timing System for FAIR and GSI (GMT) [24] is still in development but the timing master and the timing receivers are well advanced. It will use the White Rabbit platform and similar name conventions as the prototype in this report.

In all White Rabbit timing systems it is easy to create subsystems, because the White Rabbit switch features the IEEE 802.1Q Virtual Bridged Local Area Networks standard [25]. It lets the user split the physical network into arbitrary virtual networks in software, limiting interference between subsystems.

Timing Master

The GMT timing master [26, 27] will consist of a clock master, a data master and a management master.

The clock master will be a device connected to a primary clock source, such as a GPS, and will be the reference for clock and time.

The data master makes use of a high end CPU for complexity and a high end FPGA for timing sensitive tasks. On the FPGA there will be multiple soft cores, each taking care of one subsystem of the accelerator. The messages they emit will be aggregated in *control messages* and transmitted using Etherbone [28].

Lastly, the management master will take care of the network configuration using common Ethernet services, such as DHCP, RSTP and SNMP.

Timing Receiver

There will be many types of timing receivers because of different form factors of legacy equipment. Every type will have a common part and a host-specific part. Altera Arria II GX FPGAs has been chosen for all receivers to limit the variety.

When the timing receiver receives a control message from the timing master it will first match it with a prefix. If it matches it will carry out the action. The details regarding event matching and actions are still being decided upon.

2.6.3 Other White Rabbit based systems

There exists no fully operating timing system developed with White Rabbit; however there are a few being developed. Documentation regarding these is not

final and sometimes non-existent.

The CERN hardware will be based on a carrier – mezzanine concept, where all the carriers are White Rabbit enabled FPGA boards for different form factors and the mezzanine is chosen by demand [29].

The developed Timing System Prototype in this report can be seen as a reference to characteristics of a timing system developed with White Rabbit.

Chapter 3

Implementation of a Timing System Prototype

In this chapter the timing system prototype is presented. First, the requirements that were the foundation for the work are lined up. Afterwards, a detailed description of the designed architecture is given.

3.1 Requirements

The task was to design a timing system using synchronization and transport layer functionality provided by the White Rabbit PTP Core. The following requirements were found feasible to implement given the time and resources available. There were no requirements on accuracy or safety. Accuracy is directly given by the White Rabbit project. Safety depends on machine specifics as it has to be integrated with a machine protection system.

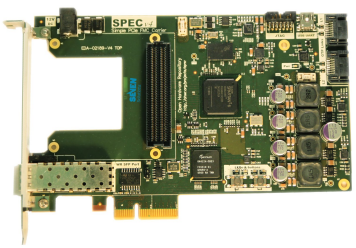
Requirements:

- fundamental timing system services
  - Trigger lines and
  - Time stamping;
- one common network for configuration and timing events;
- receivers shall be configurable over the network;
- demonstration network shall contain at least one master, two receivers and fan-out;

- master to receiver communication shall use a software to hardware protocol broadcasted over the network; and
- the data master does not have to be White Rabbit enabled.

The requirements were constrained by the hardware available:

- 1 × White Rabbit Starting Kit (Seven solutions) [30].
  - 2 × Simple PCIe FMC Carrier (SPEC) (Figure 3.1) [31].
    Xilinx Spartan 6 FPGA XC6SLX45T [32].
  - 2 × FMC 5-channel Digital I/O module (FMC DIO) [33].
  - 2 × SFP transceivers.
  - 3 × LEMO-00 Cable.
  - 3 × LEMO-BNC Adapter.
  - 1 × LC-LC Cable.
- 1 × White Rabbit 18 Port Switch (Seven solutions) [34].
- 1 × PC.



(a) *Simple PCIe FMC Carrier (SPEC).*

(b) *FPGA mezzanine card (FMC) with 5 digital input and output channels mounted on a SPEC.*

Figure 3.1: *Parts from WR starting kit. Credit: Seven Solutions.*

3.2 Architectural Overview

The timing system prototype is realized with commercial off-the-shelf hardware in combination with customized firmware that implements the timing system

services. An overview of the physical components is seen in Figure 3.2. It is easy to extend the network with more timing receivers using more White Rabbit switches as fan-out. Both timing receivers have 5 digital inputs and outputs thanks to the FMC DIO.

Etherbone is used to access the on-chip timing receiver cores over the Ethernet network, giving the timing master full control over the Timing Receivers. The timing master enumerates, configures and runs sequences using this protocol.

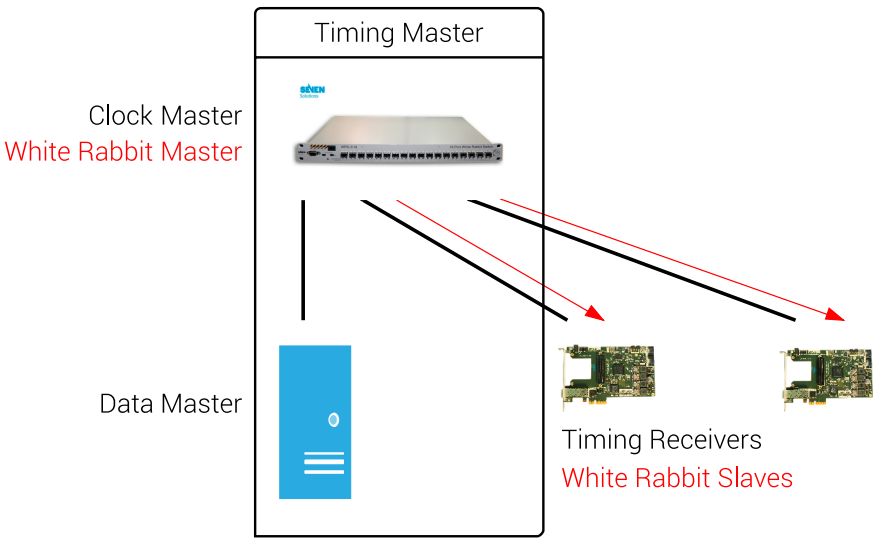


Figure 3.2: *Timing System Prototype using a White Rabbit starting kit, a White Rabbit switch and an ordinary computer. Black lines indicate Ethernet gigabit connections. Red lines indicate White Rabbit synchronized links.*

3.2.1 Etherbone

Etherbone [28] is a protocol which extends range of the on-chip wishbone interconnection over Ethernet. It is open source and hosted by the same repository as White Rabbit [5]. It consists of the Etherbone slave core and a software library. The Etherbone slave core is a slave in regard to the Etherbone protocol and a master on the wishbone interconnection. The software library provides a simple way of writing an Etherbone master.

This timing system makes use of Etherbone by having an Etherbone slave core in each timing receiver. The timing master uses the software library for sending

Etherbone packets to the timing receivers. The Etherbone slave core receives the packets and carries out the wishbone access.

3.3 Timing Master

The timing master consists of a clock master, keeping the timing network synchronized, and a data master, dictating the execution of actions. The clock master is an unmodified White Rabbit switch and the data master is an application running on regular computer with Debian Linux.

The data master is a script divided into two main parts. First, it writes the configuration to all the timing receivers and verifies that the correct data is written. Then it requests a sequence, 5 seconds in the future, by broadcasting a timing message on the timing network. It is also possible to continuously request the same sequence every second. This was deemed enough to verify the functionality of the prototype. It is, however, straight forward to convert it to higher performing machine code thanks to the Etherbone software library.

The script uses tools provided by the Etherbone project, for example, “eb-write” and “eb-read”, which do sanity checks on the data before they send the actual network packages. The tools are very convenient for quick verification of the implemented VHDL blocks.

3.4 Timing Receiver

The architecture of the implemented timing receiver, seen in Figure 3.3, consists of open hardware IP cores and custom cores. The IP cores are the White Rabbit PTP Core (WRPC), the Etherbone slave core and the Wishbone crossbar. The implemented modules are the Timing Message Receiver (TMR), the Digital Output Controller (DOC) and the Digital Input Timestamper (DIT). All cores use the same interconnection interface, Wishbone. A more elaborate architecture can be seen in the appendix, Figure A.1.

One goal of the HDL architecture was to keep the modular design imposed by the carrier – mezzanine concept of the starter kit, as little code as possible should depend on the 5 DIO FMC. With this design everything except the digital input and digital output cores can be kept if another FMC is being used.

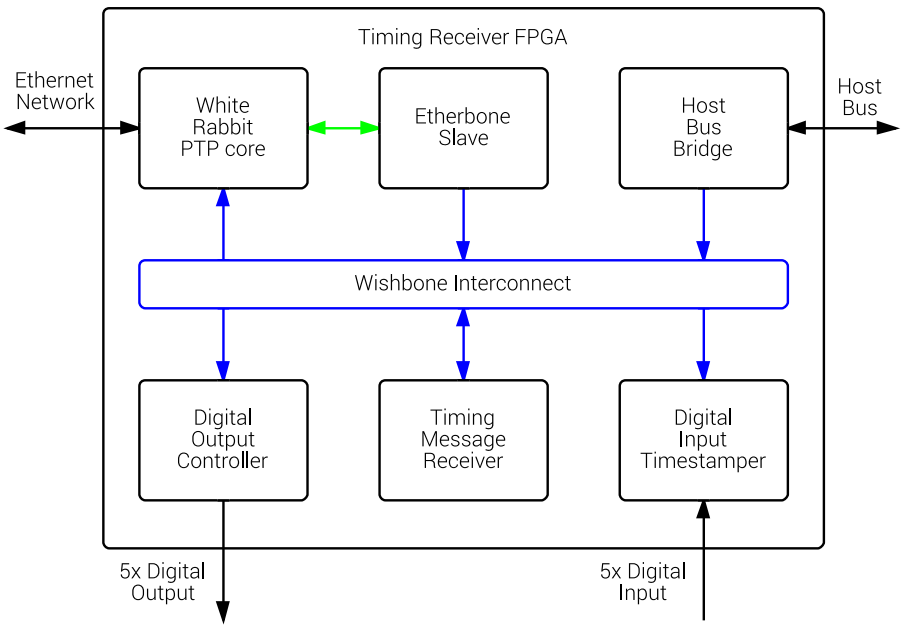


Figure 3.3: *Architecture of the implemented firmware in the timing receiver.*

A data flow for the timing receivers' real-time trigger operation can be seen Figure 3.4. The data received by the timing receiver is an Ethernet frame which is received by WRPC. This Ethernet frame is forwarded to the Etherbone core where it is unpacked. Within the Ethernet frame lies a wishbone write access containing a timing message. The Etherbone core writes the timing message to the incoming FIFO in the TMR. The timing message trigger operation of the sequencer in the TMR. It consists of a number of sequence identifiers paired with sequence start times, in TAI and clock cycles. Each sequence identifier is decoded in the TMR to a number of action messages, each with Event ID and absolute time of execution, also in TAI and clock cycles. The TMR writes the action messages to the incoming FIFO of the DOC which decodes it to events. One event correspond to a pulse and defines which digital outputs the pulse is carried out on. Each digital output has a FIFO with a succeeding pulse generator. The event is written to the FIFO of the affected digital output and is eventually carried out by the corresponding pulse generator.

Timing Message Sequence ID and absolute time of execution for sequence.

Action Message Event ID and absolute time of execution for event.

3.4.1 Wishbone

Wishbone interconnection [35] was chosen because it is fast, customizable, liked by open source projects and to stay coherent with other IP cores in the project. It is a license-free interconnection interface which is preferred at OpenCores [36] and Open Hardware Repository [5]. Both the White Rabbit PTP core and the Etherbone slave core use it.

There are two types of wishbone accesses: *standard* and *pipelined*. A standard access (one read or one write) from a Wishbone master waits for an acknowledgement from the slave before the next access can be done. Pipelined accesses can be repeated immediately after each other until a stall signal is sensed. Pipelined is therefore faster when doing consecutive writes or reads.

Currently, all wishbone interfaces are generated with a tool called Wishbone Slave Generator (Wbgen) [37]. Wbgen generates wishbone slaves from a high level description of RAMs, FIFOs, registers, etc. It is limited to generating slaves operating in, the slower, standard wishbone mode.

The Etherbone slave core is a Wishbone master that accesses in pipelined mode, hence it is incompatible with the generated wishbone slaves. To solve this, an adapter is used which implements the stall signal in such a way that master is stalled until the slave is ready to be accessed again.

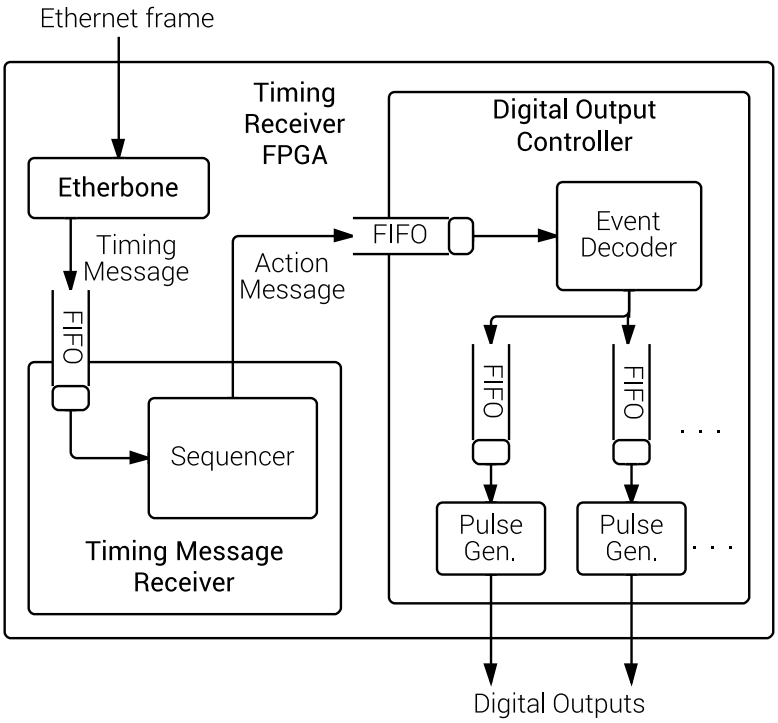


Figure 3.4: *Data flow of real-time triggering operation in the timing receiver FPGA.*

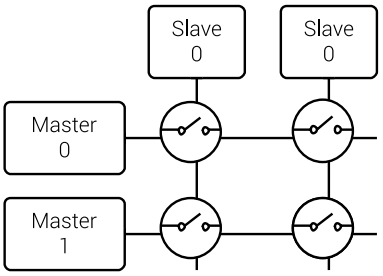
3.4.2 The Wishbone Interconnect (IP core)

The Wishbone interconnect is implemented as a crossbar switch for maximum flexibility. Therefore, multiple masters can access multiple slaves with minimal overhead. Internally it arbitrates between the masters to enable multiple connections simultaneously. A crossbar is connected to, for example, two masters, M_0 and M_1, and two slaves, S_0 and S_1. Then M_0 is able to access S_0 at the same time as M_1 accesses S_1.

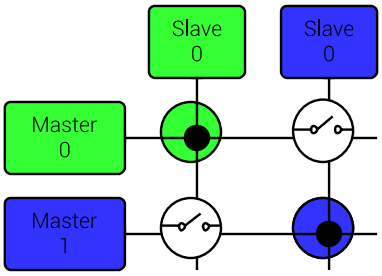
The connection is done in a matrix, see Figure 3.5. In Figure 3.5a, no Wishbone cycles are currently in operation. In Figure 3.5b, on the other hand, there are two concurrent Wishbone cycles.

The arbiter has two fundamental policies:

1. an ongoing access cycle to a slave is always preserved and
2. the master with the highest priority is granted access if multiple access cycles are initiated simultaneously to the same slave.



(a) *In the crossbar all the masters are connected with switches to all the slaves. Here no master – slave connections has been established.*



(b) *Crossbar when Master 0 is accessing Slave 0 and Master 1 is accessing Slave 1.*

Figure 3.5: *A Crossbar switch interconnection.*

3.4.3 White Rabbit PTP core (IP core)

The implemented functionality in the developed cores depends on the White Rabbit PTP core (WRPC). The WRPC handles all synchronization and PTP negotiations and provides the timing receiver with a network interface and timing interface, see Figure 3.6. There is also a Wishbone connection for management purposes, with which it is possible to read status registers and write to configure registers.

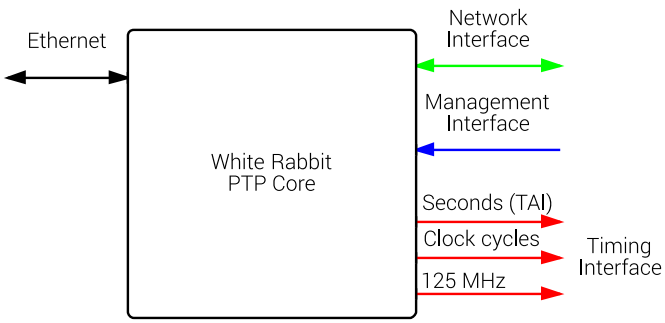


Figure 3.6: *White Rabbit PTP Core provides two important interfaces: network interface and timing interface.*

The network interface enables reception of Ethernet packets. In this implementation, the timing receiver is limited to receive timing messages and configuration data. But there is, for example, an IP core available that can be used to enable

regular network card functionality to the host via the host bus bridge. Internally it filters the Ethernet packages, because it keeps the PTP packages inside the core, see Figure 3.7.

The timing interface is a one way connection from the WRPC. It provides the time, seconds (TAI) and clock cycles, and the reference clock of 125 MHz. The granularity of the time is equal to the period time of the reference clock.

3.4.4 Etherbone Slave core (IP core)

Etherbone [28] is used for Ethernet to Wishbone access. It enables the timing master to send Wishbone commands packed in an Ethernet frame. Each Timing Receiver has an Etherbone slave core connected to the external network interface of the WRPC, see Figure 3.7. The Etherbone slave core is a slave in regard to the Etherbone protocol, on the wishbone interconnect it is a master. The intended functionality is to remotely configure and send timing messages to the timing receiver, but it has also been used for debugging and verification purposes.

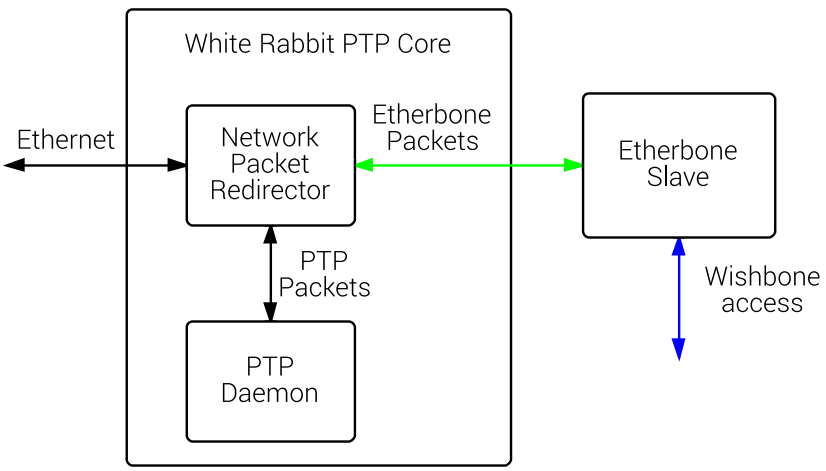


Figure 3.7: *Network packet flow from Ethernet to PTP in White Rabbit PTP Core and to Etherbone slave for wishbone access.*

Its primary function is to enable the data master to configure the Sequence RAM and Actions RAM in the Timing Message Receiver and the Event RAM in the Digital Output Controller. When they are configured the data master also uses it to send timing messages, which trigger operation. During testing, it was also used send action messages directly to the Digital Output Controller.

3.4.5 Timing Message Receiver core

The Timing Message Receiver (TMR) is a sequencer that schedules actions to the Digital Output Controller. It is implemented as a Wishbone master. It enables the timing system to be distributed, so that the timing master only has sequences of sequences (super sequences). A simplified structural architecture can be seen in Figure 3.8. Its main components are memories and a finite-state machine (FSM).

The memories consist of one FIFO for incoming messages and two RAMs containing the sequences and the actions. All memories are dual port, hence accessible both over the wishbone interface and internally in the core.

The FSM controls the read/write signals to the memories and writes to the wishbone interface. The FSM can be seen in Figure 3.9. It starts in *INIT* state and as soon as there is data in the FIFO it continues by reading the FIFO, the sequences RAM and the actions RAM. In *ST CHECK SEQ* it verifies that there is a sequence to be executed. If there is, it continues by writing to the wishbone interface in all the *WB* states. It takes 4 wishbone writes per action. As long as a *stop* action is not read, it will continue writing action messages to the wishbone interface.

The Sequences RAM simply stores addresses to the Actions RAM. This is the start of a sequence. Example data content can be seen in Table 3.1.

In the Actions RAM relative time of execution (RTOE) is stored together with an event identifier. The time is relative to the start of the sequence. The event identifiers are used in the Digital Output Controller core to look up the shape of the pulse. Every action takes 4 rows (2 rows for seconds, 1 row for clock cycles and 1 row for event identifier). Example data in the Actions RAM is shown in Table 3.2.

The data flow is as follows. The TMR receives a message via the wishbone interface, consisting of a sequence identifier (SID) and an absolute time of execution (ATOE). Using the SID as an address to the Sequences table, it looks up the first row to address in the Actions table. Then it reads the first action and continues to process actions until a special *stop* action is read. During the processing it adds the ATOE to the action's RTOE while writing the result to the DOC's incoming FIFO.

Address	Data
0	0
1	12

Table 3.1: *The Sequences RAM consists of addresses to the Actions RAM.*

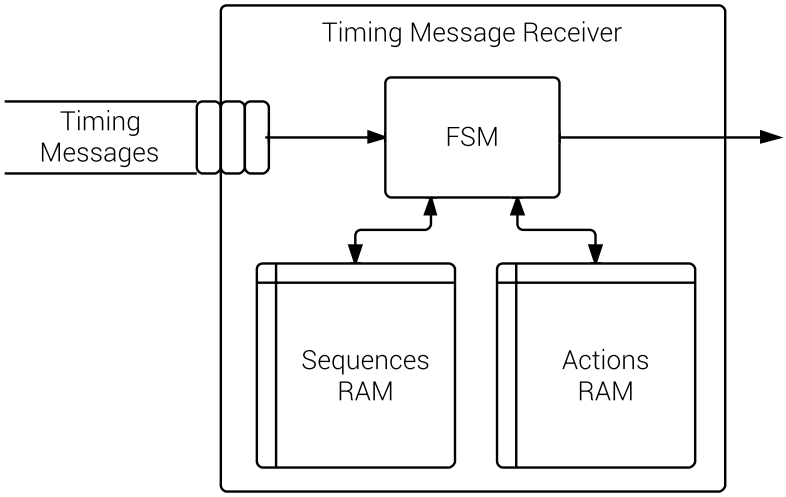


Figure 3.8: *Architecture of the Timing Message Receiver.*

Address	Data
0–3	Action 1
4–7	Action 2
8–11	STOP
12–15	Action 3
16–19	Action 4
20–23	Action 5
24–27	STOP

Table 3.2: *Example data in the Actions RAM. Each action takes 4 rows (2 rows for seconds, 1 row for clock cycles and 1 row for event identifier).*

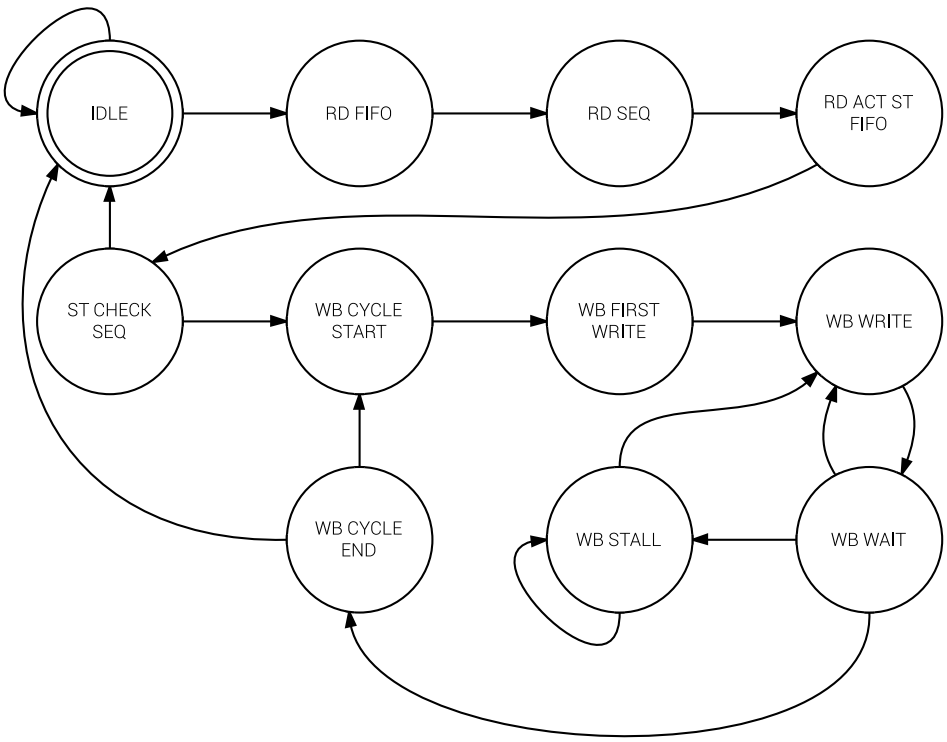


Figure 3.9: *FSM in the Timing Message Receiver.*

3.4.6 Digital Output Controller core

The Digital Output Controller's (DOC) function is to configure pulse generators for the digital outputs. It receives messages to an asynchronous FIFO, since the DOC operates in the reference clock domain (125 MHz provided from the timing interface of WRPC). This FIFO is called action message FIFO and contains action messages, see Figure 3.10. It also has an event RAM, which is used to translate incoming action messages to outputs. These memories are accessible over its wishbone slave interface.

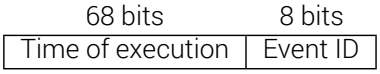


Figure 3.10: *Action message word format.*

The DOC is coarsely divided in two parts, decoding and output module, see Figure 3.12. The decoding part reads the action message FIFO, the action message consists of an event ID and the time of execution. The event ID is routed to the address line of the internal event RAM, see Table 3.3. The event RAM contains pulse width, which is routed to the channel together with time of execution forming one event, see Figure 3.11. It also contains a channel select bit field to define which channels to configure, which means that several channels can be configured with the same pulse width and time of execution.

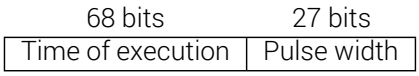


Figure 3.11: *Event format*

The output module consists of channels and pulse generators for the digital outputs. The channels are FIFOs, which are written with time of execution and pulse width, i.e., an event. The internal state machine of the pulse generator reads the channel and compares the time of execution with the current time and activates the pulse when they are equal. The pulse is then active for the amount of clock cycles defined by pulse width. This is continuously carried out as long as events are present in the channel. If the time of execution is defined in the past, compared to the reference time, the pulse generators will drop the event and continue to next. The design easily scales by adding more channels and pulse generators along with making the channel select bit field wider in the event RAM.

Address	Data	
	27 bits	5 bits
0	Pulse width	Channel Select
1	Pulse width	Channel Select
...	...	...
255	Pulse width	Channel Select

Table 3.3: *The Event RAM consists of pulse width, for configuration of the pulse generators, and channel select, to define which channel to configure.*

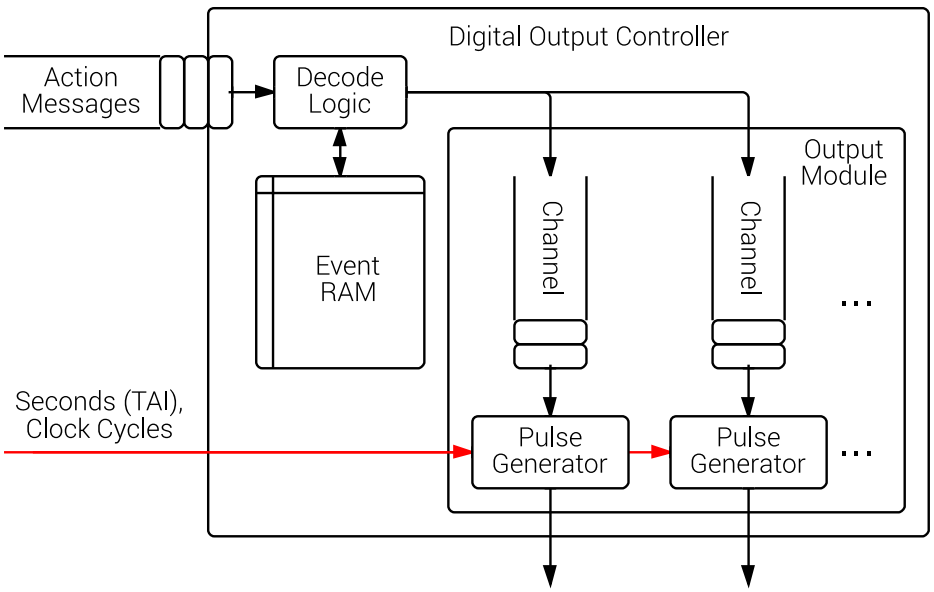


Figure 3.12: *Architecture of the Digital Output Controller.*

Event Decoder

The latency for decoding one event is 4 clock cycles. The generation of the read signal for the action message FIFO can be seen in Figure 3.13. The signal is looped back to a 3-bit shift register, which is shifted every clock cycle. This is used to limit the read rate because of delays in control signals from the memories. It checks if any of the channels becomes full after a channel write. If that occurs, it stalls until the channel is not full. It only issues a read signal if:

- action message FIFO is not empty,
- all channels are ready for data, meaning that last written data did not

generate a channel full signal.

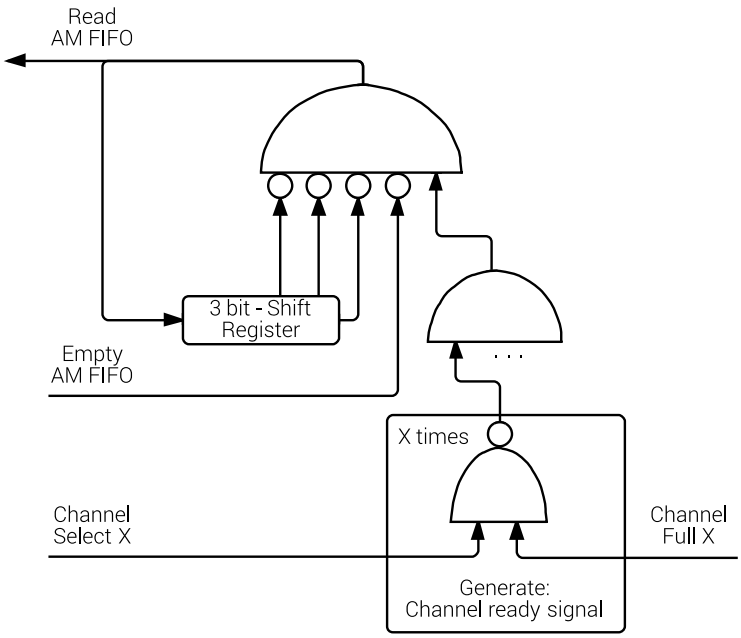


Figure 3.13: *Generation of read signal for action message FIFO.*

The generation of the write signal only looks at the channels select bit and the action message FIFO read signal, which already ensures that no data is read from the action message FIFO if any channels are full. Therefore a channel write signal is generated by an *and* operation with the select bit and action message FIFO read signal delayed 2 clock cycles. This is the total latency for action message FIFO read operation and data from the event RAM to propagate to the channels data input.

Pulse Generator

The pulse generator is implemented with a finite-state machine, see Figure 3.14, and is the final element which issues the pulses. It reads events from the channel and executes them to pulses at the specified time with the specified pulse width. The signals in the FSM are:

Inputs

Current Time, Valid Time, Trigger Time, Pulse Width, Channel Empty, Valid Signal,

Internal Register
Pulse width counter,

Output Registers
Read Channel and Pulse.

The Valid Signal is composed as:

$$\text{Valid Signal} = \text{Valid Time} \wedge \text{Channel Empty}' \tag{3.1}$$

Where Valid Time is part of the timing interface from WRPC.

Each state with functionality is as follows:

IDLE
If the valid signal is high, the channel is read and it moves to *WAIT FOR DATA*.

WAIT FOR DATA
Immediately move to *WAIT TO TRIG*.

WAIT FOR TRIG
Load the pulse width to the pulse width counter and wait until the trigger time occur, then it moves to *PULSE – CHANNEL EMPTY*. If time is not valid or trigger time already occurred, it moves to *IDLE*.

PULSE - CHANNEL EMPTY
Pulse goes high and pulse width counter decreases every clock cycle. If valid signal is high, the channel is read and it moves to *PULSE – CHANNEL NEMPTY*

PULSE - CHANNEL NEMPTY
The pulse continues and when the pulse is carried out it moves to *WAIT TO TRIG*.

In summary, the pulse is high only in state *PULSE – CHANNEL EMPTY* and *PULSE – CHANNEL NEMPTY*. The channel is only read in state *IDLE* and *PULSE – CHANNEL EMPTY*, when Valid Signal is high. After which the FSM moves to another state to ensure that the channel only is read once. The FSM is constructed so that an initiated pulse always finish even though time is no longer valid. If the valid signal is still low when the pulse is carried out it moves to *IDLE* until valid signal is high again. It is limited to issue pulses every third clock cycle.

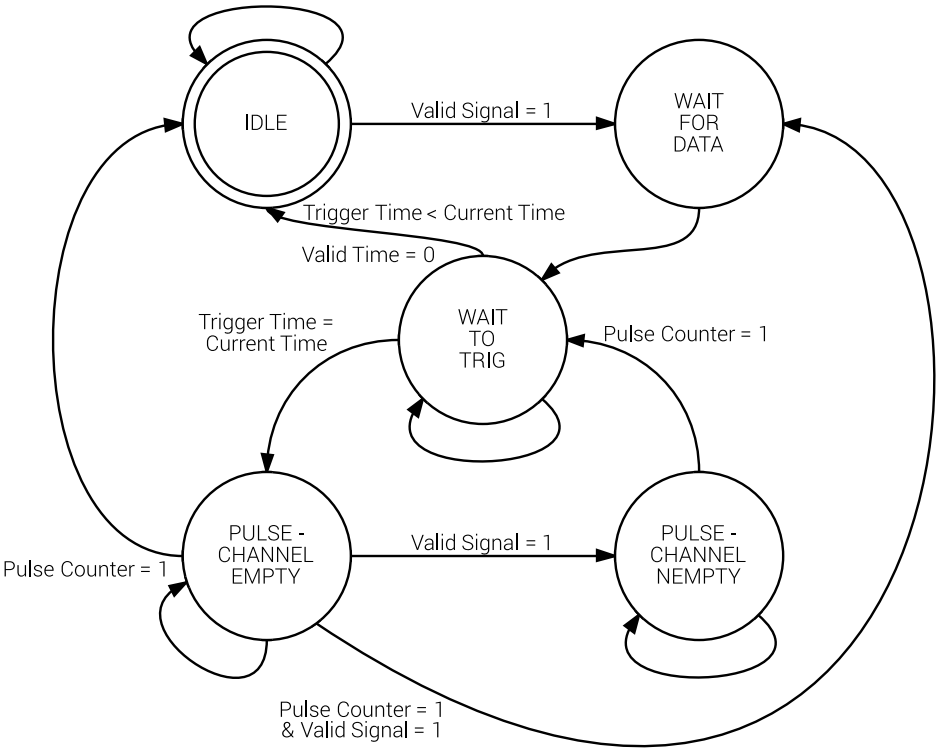


Figure 3.14: *FSM of the Pulse Generators in the Digital Output Controller.*

3.4.7 Digital Input Timestamper core

The Digital Input Timestamper (DIT) sense edges on the digital inputs and stamp them with the current time. The DIT operates in the reference clock domain (125 MHz). It has an outgoing asynchronous FIFO where the timestamps are stored and timestamp logic to determine which inputs to timestamp, see Figure 3.15. The timestamp logic consists of an input detector for each digital input which can be configured with the control register. It also generates write signal to the outgoing timestamp FIFO. The FIFO and control register are accessible via the DIT's wishbone slave interface.

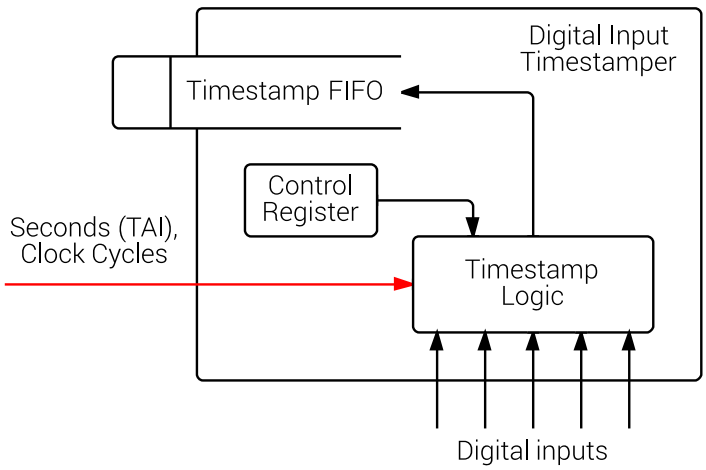


Figure 3.15: *Architecture of the Digital Input Timestamper.*

Input Detector

The input detector, see Figure 3.16, in the DIT is configurable with channel select, which enables/disables timestamping, and trigger polarity, which sets the timestamp to trigger on rising/falling edge, for each channel. By default this register is configured so that all channels are enabled with triggering on rising edge. In total there are 5 inputs meaning $X = \{0, 1, 2, 3, 4\}$.

Each digital input goes through a 2-register synchronizer to avoid metastability, since the inputs are fired asynchronously from the outside the FPGA. After which it is routed to both a falling- and rising edge detector. The control signals are evaluated and decide whether there was an edge.

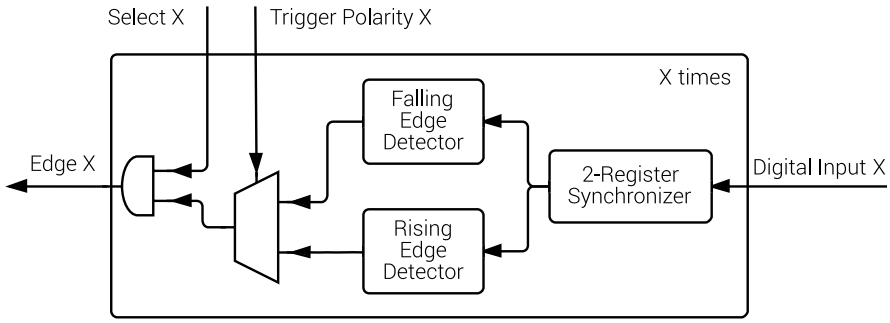


Figure 3.16: *Implementation of the input detector.*

Timestamp FIFO

The timestamps are memory demanding since each timestamp is 68 bits. This core joins timestamps from all digital inputs into one single FIFO with a channel bit mask which marks which channel was active, see Figure 3.17. The FIFO has a length of 512, but in reality only 511 timestamps can be stored to avoid collision of the read and write pointers in the FIFO. Currently these timestamps are read by polling the FIFO via Etherbone from the Timing Master.

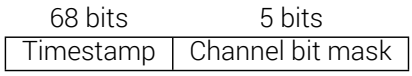


Figure 3.17: *Timestamp FIFO word format.*

The write signal for the timestamp FIFO, see Figure 3.18, is implemented with an **OR** operation of all the Edge signals and a comparison of the timestamp FIFO counter with a hysteresis. The comparison is necessary due to latency of the counter signal and the high rate of incoming edges, ensuring that the FIFO not becomes full. The result is that when any edge is sensed and FIFO is not above the hysteresis all active edges are stored in the timestamp FIFO together with the current time.

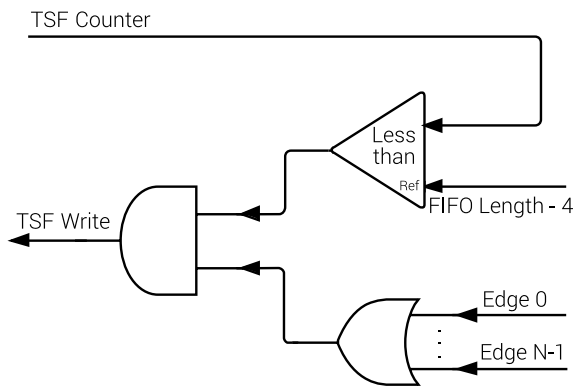


Figure 3.18: *Implementation of the write signal for the timestamp FIFO.*

Chapter 4
Result

The result of the work is primarily the timing system prototype implementation. Below are the calculated performance for different aspects of the system and the numbers verifying its functionality.

4.1 Timing System Prototype

The timing system prototype makes use of the synchronized distributed time provided by White Rabbit and Ethernet to Wishbone communication from Etherbone. It primarily provides firmware for the timing receivers and proof-of-concept software on the data master. It has the same basic functionalities as described in the timing system model, see Section 2.4

The prototype is standalone, in the point of view that no extra network is needed for configuration. The timing receivers are configurable via the White Rabbit network, which also carry the timing messages to trigger operation.

4.2 Timing Receiver

The current firmware for the timing receivers can be configured with several sequences. Each sequence consists of an arbitrary number of actions. The actions are mapped to events, configurable with pulse length and channel select. The result is a configurable independent timing receiver, where all configurations is done beforehand and timing messages can be distributed ahead of time.

All actions in a sequence must be ordered chronologically. The action messages are processed in order by the Digital Output Controller and translated to events.

A non-chronological order will result in dropping of events, since all events are blocking until their end of execution.

4.2.1 Wishbone

The throughput of Timing Messages and Action Messages is limited by the current usage wishbone slaves operating in standard mode. Both messages need 4 wishbone write clock cycles which takes minimum 8 clock cycles in standard mode. This leads to total write time of $8 \cdot \frac{1}{62.5MHz} = 120\,\text{ns}$.

4.2.2 Event Rate

Every pulse generator is preceded by a channel, a FIFO, which size decides the number of events which can be outputted with maximum event rate. Outputting events every clock cycle does not make sense, since their pulse lengths could be added. Therefore, the maximum useful event rate is $\frac{125MHz}{2} = 62.5\,\text{Mhz}$. However, the latency from the Action Message FIFO to a channel is four clock cycles, which leads to an long term event rate of $\frac{125MHz}{4} = 31.25\,\text{MHz}$ limited by the size of the action message FIFO.

4.2.3 FPGA Resources

Primitive	Occupied	Available	Usage
Slices	4247	6822	62 %
Block RAM, RAMB16BWER	110	116	95 %

Table 4.1: *Resource usage, synthesising to Xilinx Spartan 6 FPGA XC6SLX45T.*

4.3 Verification

For verification purpose the timing receiver was configured with a test sequence and the 5 channels on the FMC DIO mounted on the SPEC were measured. This is all done in a test script using tools provided by the Etherbone library. The current time on the timing receiver is read and a timing message is sent triggering a sequence 5 seconds later, afterwards the timestamp FIFO were read.

The trigger sequence can be seen in Table 4.2 and the events matching in Table 4.3. The offsets and widths are stored as integers; the values are translated to seconds

for easier comparison with graphs. The tables show when there should be pulses relative to the sequence time of execution and their lengths. The resulting timestamps from running this sequence can be seen in Table 4.4. There it can be seen that there is a latency of 40 ns between scheduled output and time stamped input.

Action ID	**Offset**		**Event ID**
	Clock cycles	translated	
0	82	0.656 µs	86
1	575	4.600 µs	30
2	1066	8.528 µs	16
3	1565	12.520 µs	232
4	2147	17.176 µs	122
5	2586	20.688 µs	255

Table 4.2: *Part of Actions RAM verification data. Seconds are calculated using 8 ns period time.*

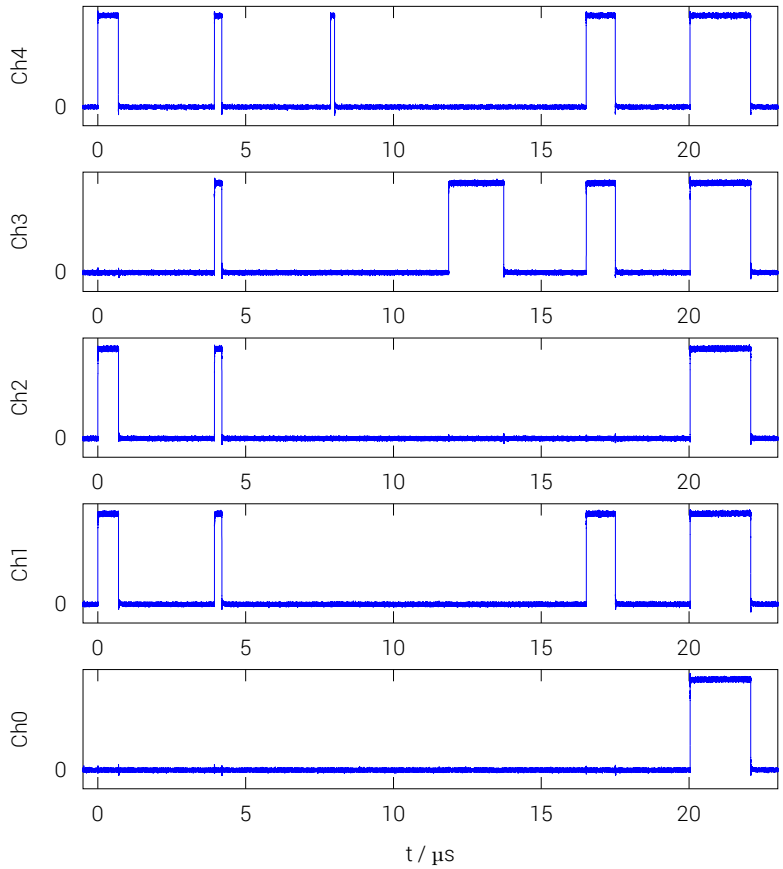
Event ID	**Pulse Width**		**Channel Select**
	Clock cycles	translated	(4,3,2,1,0)
16	17	0.136 µs	10000
30	31	0.248 µs	11110
86	87	0.696 µs	10110
122	123	0.984 µs	11010
232	233	1.864 µs	01000
255	256	2.048 µs	11111

Table 4.3: *Rows of Events RAM verification data used in test. Seconds are calculated using 8 ns period time.*

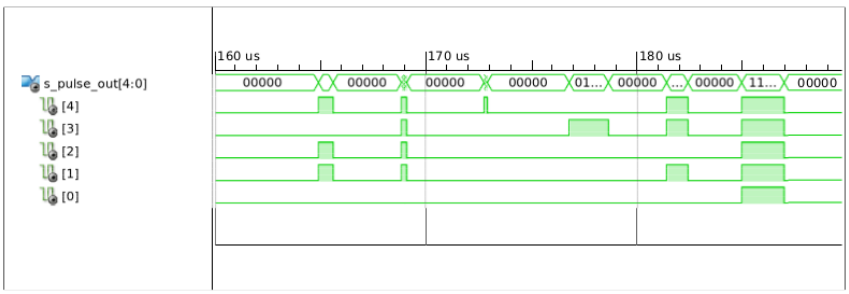
Number	**Timestamp**	**Channel Bit Mask**
	Translated	(4,3,2,1,0)
0	0.696 µs	10110
1	4.640 µs	11110
2	8.568 µs	10000
3	12.560 µs	01000
4	17.216 µs	11010
5	20.728 µs	11111

Table 4.4: *Data readout from the timestamp FIFO.*

In Figure 4.1 it is shown that the simulations graphs match the oscilloscope graphs. The oscilloscope graph can also be roughly verified with the time offset in the sequence table.



(a) *Measured data for the output channels.*



(b) *Screenshot from simulation software (Isim).*

Figure 4.1: *Comparison between simulation and measurement.*

Chapter 5

Discussion & Conclusion

In this chapter a comparison of White Rabbit with existing systems is discussed. It also presents improvements which can be done on the implemented timing system prototype. Finally there are conclusions of the work and usage of White Rabbit in timing systems.

5.1 Comparison

This section compares White Rabbit to existing systems in regard to perfomance as synchronization layer and for design of timing systems.

5.1.1 Synchronization Layer

White Rabbit differs a lot from existing event-based timing systems. While event-based timing systems depend on keeping exactly the same delay to all nodes, White Rabbit measures and compensates for delay and variations in delay on the fibres.

The use of Ethernet in White Rabbit induces latency in the network, this comes from routing delays in the switches. To achieve better determinism, packet priorization can be used. This makes it deterministic in the sense that the switch has an upper bound latency for propagation of the highest priority ethernet frames. This enables timing critical frames to reach their goal in time while the network is used for regular data transfers. Even though priorization is used, the latency is not negectable. The maximum routing delay is estimated to 13 μs. This can be compared with estimated fibre link delay of $5\frac{\mu s}{km}$ [38].

Event-based timing systems usually have a custom stripped down protocol without network routing to get low latencies and high determinism. In reality the lack of routing means that the same signal reach all the nodes. Upstream in these networks can also be solved in a custom manner. In MRF, for example, there are concentrators which connects underlying nodes on a separate fibre network.

The latency in a White Rabbit network may not be suitable for certain low latency feedback loops. On the other hand, the use of 1000BASE-BX10 Ethernet in White Rabbit standardizes transmission and provides an upstream link. The standardization simplifies expansion of the network and also enables the network to be used for regular data transfers.

White rabbit is designed to provide sub nano second accuracy of the synchronization and compensates for temperature variations on the fibre. In event-based systems no temperature compensation is done, which means that accuracy is affected by the difference in temperature between the fibre links. Hence, if temperature is kept the same throughout the facility, synchronization of the distributed nodes achieve high accuracy.

5.1.2 Timing System Design

White Rabbit addresses important aspects, such as standardization and open source, which did not exist on hardware level in timing related applications for accelerators before. The standardized FMC connector on current available hardware together with the distributed time base enables easier addition of nodes with new functionalities. A new node gets the time base as soon as it is plugged in and can act independently from the rest of the system.

It also uses a standardized on-chip interconnect. Together with the timing interface this makes prototyping simple and straight forward. A core with new functionality can easily be added to an exising design. Non standardized interconnections requires that the designer knows interals of the core which is extended and may compromise physical constraints in the design.

5.2 Future Improvements of Timing System Prototype

The implemented timing system is a prototype to show how the White Rabbit platform can be used. During the work several improvements on the architecture have been identified and are proposed in this section.

5.2.1 White Rabbit

Since White Rabbit is an ongoing project there are improvements being done continuously. To benefit from these changes it is therefore important to keep merging the HDL code with the original repository. Currently the project is moving towards better stability and management features but hopefully there will also be improvement in synchronization and package forwarding delays.

5.2.2 Data Master

In future work, the data master application could be converted to run as a single process on a real-time operating system for best determinism and highest performance, which is required in production quality implementations. This is also the path chosen at CERN and GSI for their corresponding functionality.

5.2.3 Timing Receiver

Several improvements to the timing receiver architecture have been identified since the design was finalized. This section will present improvements to the existing cores, as well as an additional core, which will extend the functionality. An architectural overview can be seen in Figure A.1.

Timing Message Receiver core

Block RAM resources are used a lot in the timing receiver because of all the FIFOs and RAMs in the different cores. There are some minor memory optimizations that can be done to the Timing Message Receiver (TMR), but in the end the memory usage depends on the number of sequences and actions required.

Currently the TMR can store 16 actions. Each action has full TAI time resolution and event ID, it is unpacked which means that it consumes 128 bits storage per action, see Table 3.2. This can be extended with 128 actions per block RAM, RAMB16BWER. The actions are ordered in their sequences, with a flag in between. This means that you get less overhead by having longer sequences.

Having full TAI time resolution in the actions RAM might be elegant and generic but the cost is high block RAM usage. The number of actions could easily be doubled by limiting the time resolution and pack it with event ID. By halving the memory usage to 64 bits per action, it is still possible to have sequences that are 6671 days long by storing time of execution with 56 bits and event identifier as 8

bits. If there is need for even more actions another solution is to use an off-chip RAM that is available.

Digital Output Controller core

The Digital Output Controller (DOC) decodes incoming event IDs to events and configure the defined pulse generator's channel with pulse length and time of execution. The decoding waits maximum time for *not empty* signal, from action message, and full signals from the FIFOs to settle and does not evaluate full signals from the channels separately. This is a simple and deterministic solution, however it compromises maximum event rate and stalls as soon as one channel is full.

To maximize the event rate while keeping an upper level latency a more sophisticated decoding functionality can be designed. An improvement would be a decoding stage which evaluates the channels full signals separately. This would remove unnecessary stalling and hence also increase the long-term maximum event rate. To further improve the event rate, the decoding stage has to be pipelined.

The flow of timing related data in the timing receiver FPGA is complex and verifying that a sequence is playable depends on a lot of factors. Therefore a run-time error check is suggested. The error check can extend functionality when the pulse generator drops an event. The time of execution together with a channel bit mask can be stored in an outgoing FIFO. If an error occur the front end controller can be interrupted and retrieve the error source in the FIFO.

The error check would be very valuable while commissioning the timing system and when doing post-mortem analysis.

Digital Input Timestamper Core

The timestamps in the Digital Input Timestamper (DIT) is polled over Ethernet with Etherbone, this generates a lot of data traffic and wishbone accesses even though no timestamps exists. Therefore it is suggested that the Timestamp FIFO generates interrupts to the host bus controller chip to trigger timestamp read from a device driver in the front end controller.

The raw timestamp data is huge and may need processing before it is transmitted on the network. To avoid data loss, it has to be sent with a reliable protocol with error check such as TCP. This is easiest done by integrating the device driver to a higher level in the control system software framework running on the front end controller, where the data can be sent over any preferred network.

At first there were inconsistencies between the digital input timestamps. Even though a pulse was scheduled at the same time on all outputs, it was sometimes timestamped with 8 ns difference. This was solved by putting constraints on registers, both in the output core and the input core.

Asynchronous Timing Message Generator core

The implemented timing system prototype only support generation of timing messages to trigger operation centrally from the data master. This is, however, not always sufficient to noticed demands on faster control loops. To solve this issue, it is suggested that a local timing message generator is implemented. The local timing message generator can generate timing messages asynchronously when a digital input is triggered; hence the core is called Asynchronous Timing Message Generator (ATMG).

It is possible to use the existing TMR and DOC for reception and processing of these timing messages. The ATMG can generate similar Timing Messages as the data master. The Timing Messages generated can be configurable and different depending on the digital input source.

Using both a data master and asynchronous generated timing messages can lead to collisions in the outputs. Therefore, a generic solution is preferable, which supports both, but not at the same time. This can be achieved with the current design, by leaving "empty" slots in the Sequence RAM in the TMR. Empty slots are simply a special code which is outside of the Actions RAM address range.

Wishbone

The wishbone slave generator is limited to generating slaves operating in *standard* wishbone mode. Hence, every wishbone write to the implemented slaves take 2 clock cycles. It would be beneficial for the DOC to be able to receive Action Messages with a higher rate than every 120 ns, since this is the bottleneck, limiting long term event rate.

If an Asynchronous Timing Message Generator is designed, this would be fundamental, since it would decide the latency between the asynchronous input and earliest start of the sequence. A way to improve this, without making any major architectural changes, would be to implement *pipelined* wishbone slaves. Another alternative is to interconnect the TMR and DOC with a wider data width. In Wishbone it is possible to have up to 64 bit wide data.

Memory prioritizations

The current White Rabbit PTP core occupies 79 block RAMs, RAM16BWER, of the Xilinx Spartan 6 LX45T. 44 of these are occupied by the WRPC's internal soft core (LatticeMico32) as program and stack memory, which might be extended further. The Xilinx Spartan 6 LX45T contains 116 block RAMs in total, which leaves 37 Block RAMs for custom applications.

The Etherbone Slave core occupies 4 Block RAMs, which gives the Timing Receiver 33 Block RAMs to use. These block RAMs have to be divided primarily between the Actions RAM in the TMR, the Action Message FIFO in the DOC and the timestamp FIFO in the DIT. The size of these memories has direct implications on number of actions, number of events with maximum event rate and number of timestamps.

Routing

There are two common problem with integrated circuits regarding delay, clock skew and data path lengths. To have precise timing on a printed circuit board (PCB) / field-programmable gate array (FPGA) with multiple outputs, the routes from the clock to the final registers need to have the same length, which minimizes clock skew. The final registers must be placed so that the data path from the registers to the output pads also is of same length. Finally, any routing on the PCBs also needs to have the same length.

Since the cards used in this report were bought, there was no way of choosing output pads or route the PCB. This limited the options to achieve better timing to *location constraints on registers* and *manual routing* when creating the bitfile for the FPGA.

5.2.4 Change Timing Format

Currently the timing format is unmodified TAI (40 bits seconds) and clock cycles. The number of bits for clock cycles are defined by the frequency, 125 MHz requires 28 bits. In total the timing format requires 68 bits.

68 bits is impractical with a 32 bit data bus interface. Without any form of bit packing it generates an overhead of $96 - 68 = 28$ bits. But, worse is, that every time the data is transferred it is 3 Wishbone writes instead of 2, generating a time overhead of 50% per write.

This timing format also makes it difficult to do timing arithmetic straight forward, since there are two time bases (8 ns and seconds). It also requires extra unnecessary block RAM in the FPGA which is already a scarce resource.

The solution is to move to 64 bit clock cycles, which still can represent 148 billion seconds (4676 years) with the same clock. 32 bits would, on the other hand, not be enough, as it can only represent 34 seconds.

5.2.5 Architectural Aspects

The bottlenecks described in aspect of event rate are mainly due to the used bus width and event decoding in the DOC. They are architecturally built in to favour a modular design with a generic TMR. This design was chosen to simplify porting of other FMCs to the same timing system while keeping the timing message interface the same.

The architecture can be redesigned so that the TMR and DOC are conjoined and the concept of actions and events are merged. The current DOC's output module can be configured directly from actions RAM with correct bus width. This would eliminate serial transmission delay on the wishbone bus and delay inflicted by decoding via the event RAM. It would also simplify verification of sequences so that they are actually playable.

5.3 Conclusion

A timing system prototype has been implemented, using a combination of open and freely available hardware/software/tools and custom developed cores. Its performance is good enough for many practical applications, and improvements have been found to make it even more generic and applicable.

White Rabbit is becoming more established and has been adopted by several recent projects. The community around it is growing strong and the maintainers are continuously improving it. When this project has matured, it will be a firm base to build upon. The existing hardware platform together with the White Rabbit FPGA firmware saves development time when inventing new timing systems, by utilizing tested and debugged hardware and HDL.

The time synchronization and Ethernet in a timing network, enables high flexibility. Timing-related data can be processed ahead of time in White Rabbit nodes, while any Gigabit Ethernet compatible device also can access the network.

An ideal timing system based on White Rabbit can be configured once with sequences, actions and events. Afterwards, messages can be distributed which

enables the nodes to function independently for a defined period. During this time there is a, high throughput, Ethernet network almost completely free, except for some low priority PTP packets. Even better, if a cut-through technique is used in the switches, the regular traffic will not interfere at all with the timing messages and the network can be used as both a timing network and data network concurrently.

Because fibre lengths are measured and compensated for, a White Rabbit network is extremely scalable. It is possible to connect a new node at any level in the hierarchy and with any (reasonable) distance to the closest switch. In the implemented prototype it is also trivial to replace a timing receiver, since it will request management data from a server and immediately start running. These are important aspects of modern accelerators that reach for highest possible availability.

In conclusion, even though White Rabbit will be the foundation for future timing systems, there are always customizations to be made. Scientific projects are so different in their nature that no single solution works for all. Design choices even had to be made in the implemented prototype, which inferred priority of some kind. And, in the end, it is up to the requirements of the machine to decide what those priorities are.

Appendix A

Architecture

The final architecture of the developed firmware facilitated in the Timing Reciever's FPGA can be seen in Figure A.1. The dashed core, Asynchronous Timing Message Generator, is not implemented.

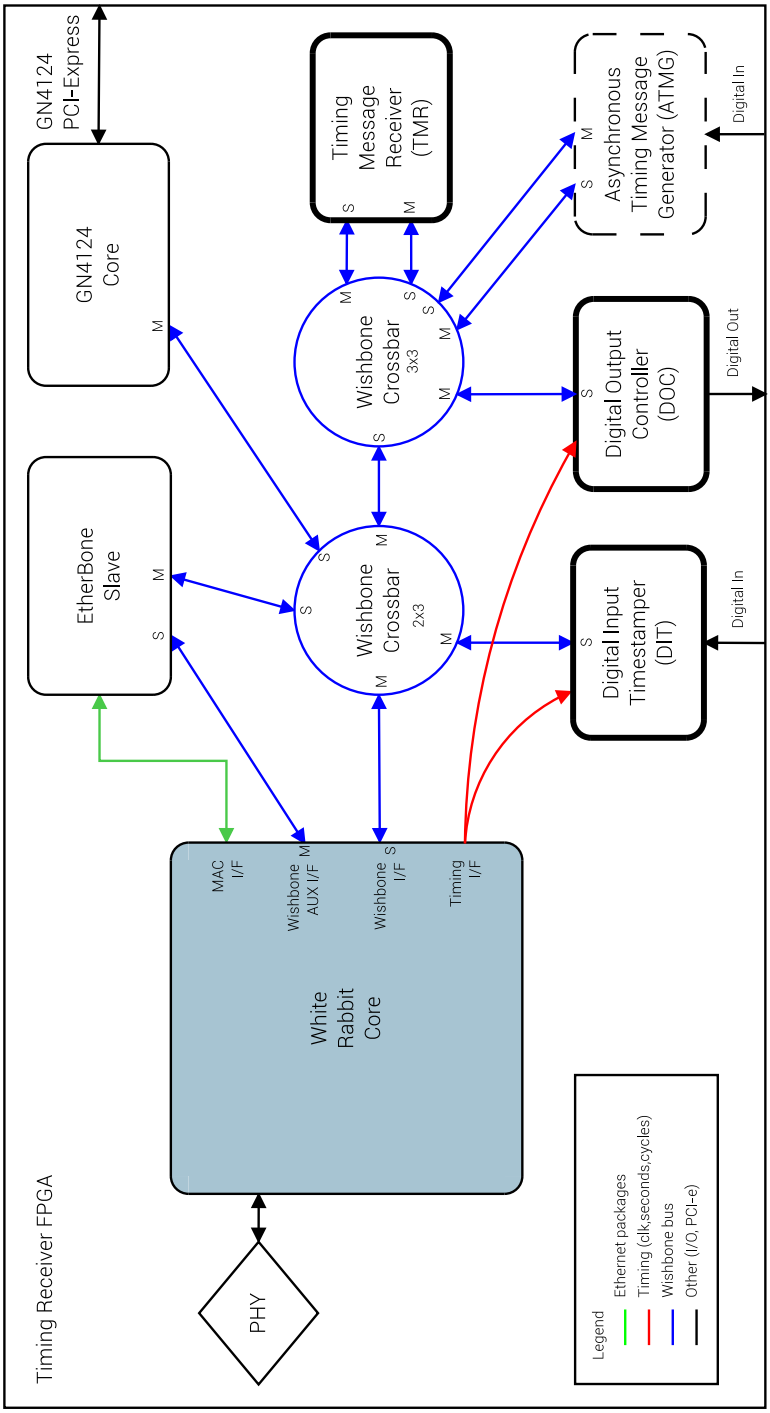


Figure A.1: *Detailed FPGA firmware architecture for the Timing receiver.*

Appendix B

Source Code

The complete project source code is available from both authors. All the code is version controlled with a distributed version control system called Git [39]. The links to the repositories given below are used together with the `git clone` command.

White Rabbit core collection
All HDL cores related to the White Rabbit project are collected in this repository.
`git://ohwr.org/hdl-core-lib/wr-cores.git`

Etherbone core
HDL core, software library and tools related to the Etherbone project.
`git://ohwr.org/hdl-core-lib/etherbone-core.git`

Platform independent core collection
Generic Wishbone HDL cores, like the Wishbone crossbar, are available through the general-cores repository. The open sourced soft core LM32 is also available through it.
`git://ohwr.org/hdl-core-lib/general-cores.git`

Gennum GN4124 core
This project provides a wishbone interface to the PCI express chip GN4124, effectively a Wishbone – PCI Express bridge
`git://ohwr.org/hdl-core-lib/gn4124-core.git`

Wishbone slave generator
The wishbone slave generator generates inferred memories and a wishbone interface to access them from a higher level description language. The memories are therefore platform independent. Currently it only generates

a *standard* wishbone interface.
`git://ohwr.org/hdl-core-lib/wishbone-gen.git`

Hdlmake
This is a platform independent project preparation tool. It generates makefiles required to synthesize to Altera and Xilinx platforms without the graphical user interfaces.
`git://ohwr.org/misc/hdl-make.git`.

Software for White Rabbit PTP core
The software that runs on the soft core (LM32) in the WRPC is located here. Amongst other things it handles the White Rabbit UART console and the PTP negotiations.
`git://ohwr.org/hdl-core-lib/wr-cores/wrpc-sw.git`.

Software support for the SPEC board
Kernel and user space Linux code. It is mainly used to flash the SPECs.
`git://ohwr.org/fmc-projects/spec/spec-sw.git`.

Appendix C

Development Environment

Both authors prefer to use Linux as their main operating system both professionally and private. It therefore came naturally that all development was done on machines running Linux. The code was compiled on laptops running different versions of Ubuntu/Mint and on a build server running Debian.

Emacs was chosen as editor because of its good VHDL support. More importantly, a coding style was chosen to ease collaborative development further. The code strictly follows [40], as it is similar to the style taught at Lund University, only stricter in some aspects.

The distributed version control system Git was chosen for the project to enable collaboration. It is superior when it comes to merging branches, which enables all the participants to work fully in parallel. It also makes it easy to have the complete repository, with all history, on all computers.

Build flow

A build flow with *Makefiles* was set-up, to enable synthesizing without any graphical tools, which speed up the process of development. The tool *hdlmake* [41] facilitated, as it generated the required collections of files to the proprietary tools. The build flow is very important if production quality results are required [42].

Testbench

Testbenches are extremely important, as they give immediate feedback if the code is synthesized to the correct hardware. All cores implemented had a proper testbench and there were also a testbench testing them put together.

This allows verification to some extent before testing on real hardware.

Continuous Integration

The work was split into different cores, so that development could be done individually. Then Jenkins [43] was used to make sure that they played nicely together the whole way. Jenkins is a continuous integration tool that builds the project, either with a predefined schedule or asynchronous.

Because of the well prepared build flow it was trivial to set-up Jenkins to automatically build all modified branches of the project upon *git push*. During the development this also gave important feedback regarding resource usage. Using the build logs it was possible to see when resource demanding code was committed to the repository.

Scripted final testing

As soon as the build machine was finished, there was a script to test the functionality on the mounted hardware. This made it easier to find regressions once the code was finalized and only bug testing remained.

Every new feature or bug fix was written on new branches and tested individually before merging into the *master* branch. This ensured that no change broke existing functionality.

All old builds were also stored, together with their reports, for reference.

Bibliography

[1] S. Peggs et al., eds. *ESS Technical Design Report*. Apr. 2013.

[2] J. Wei, M. Blaskiewicz, N. Catalan-Lasheras, D. Davino, A. Fedotov, et al. "Design optimization and the path towards a 2 MW Spallation Neutron Source". In: *Particle Accelerator Conference, 2001. PAC 2001.* Vol. 1. 2001, 319–321 vol.1. DOI: 10.1109/PAC.2001.987503.

[3] J. Lettry, L. Penescu, J. Wallner, and E. Sargsyan. "Ion sources for MedAustron". In: *Review of Scientific Instruments* 81.2, 02A328 (2010), DOI: http://dx.doi.org/10.1063/1.3277196.

[4] P. Moreira, J. Serrano, T. Wlostowski, P. Loschmidt, and G. Gaderer. "White rabbit: Sub-nanosecond timing distribution over ethernet". In: *Precision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on*. Oct. Pp. 1–5. DOI: 10.1109/ISPCS.2009.5340196.

[5] *The Open Hardware Repository*. URL: http://www.ohwr.org/.

[6] P. Vyskočil and J. Sebesta. "Relative timing characteristics of GPS timing modules for time synchronization application". In: *Satellite and Space Communications, 2009. IWSSC 2009. International Workshop on*. 2009, pp. 230–234. DOI: 10.1109/IWSSC.2009.5286378.

[7] *Micro-Research Finland*. URL: http://www.mrf.fi/.

[8] M. Lipinski, T. Wlostowski, J. Serrano, and P. Alvarez. "White rabbit: a PTP application for robust sub-nanosecond synchronization". In: *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2011 International IEEE Symposium on*. Sept. Pp. 25–30. DOI: 10.1109/ISPCS.2011.6070148.

[9] A. Widmer and P. Franaszek. "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code". In: *IBM Journal of Research and Development* 27.5 (1983), pp. 440–451. ISSN: 0018-8646. DOI: 10.1147/rd.275.0440.

[10] P. Moreira, P. Alvarez, J. Serrano, I. Darwezeh, and T. Wlostowski. "Digital dual mixer time difference for sub-nanosecond time synchronization in Ethernet". In: *Frequency Control Symposium (FCS), 2010 IEEE International*. 2010, pp. 449–453. DOI: 10.1109/FREQ.2010.5556289.

[11] D. L. Mills. *Computer network time synchronization: the Network Time Protocol*. Second. Boca Raton, FL, USA: Taylor and Francis, 2011. ISBN: 1-4398-1463-5.

[12] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2002* (2002), pp. i–144. DOI: 10.1109/IEEESTD.2002.94144.

[13] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (2008), pp. c1–269. DOI: 10.1109/IEEESTD.2008.4579760.

[14] A. Söderqvist, N. Claesson, J. N. Rodrigues, R. Tavčar, R. Štefanič, et al. "Comparison of Synchronization Layers for Desgin of Timing Systems". In: *ICALEPCS2013*. Oct. 2013.

[15] D. Beck, R. Bär, M. Kreider, C. Prados, S. Rauch, et al. "The New White Rabbit Based Timing System for the FAIR Facility". In: *PCaPAC2012*. 2012.

[16] J. Pietarinen. "12 way VME/cPCI Fan-out manual". 2009. URL: http://www.mrf.fi/dmdocuments/FOUT12-TREF-002.pdf.

[17] J. Pietarinen. "8 compactPCI fan-out concentrator". 2008. URL: http://www.mrf.fi/dmdocuments/cPCI-FCT-003.pdf.

[18] J. Pietarinen. “Event Generator, Modular Register Map Manual”. 2011. URL: http://www.mrf.fi/dmdocuments/EVG-MRM-0003.pdf.

[19] J. Pietarinen. “Event Receiver, Modular Register Map Manual”. 2011. URL: http://www.mrf.fi/dmdocuments/EVR-MRM-003.pdf.

[20] H. Kaji, K. Furukawa, M. Iwasaki, E. Kikutani, T. Kobayashi, et al. “Upgrade of Event Timing System at SuperKEKB”. In: *ICALEPCS2013*. Oct. 2013.

[21] E. G. Cota, M. Lipinski, T. Włostowski, E. (d. Bij, and J. Serrano. “White Rabbit Specification: Draft for Comments”. 2010. URL: http://www.ohwr.org/attachments/306/WhiteRabbitSpec.pdf.

[22] M. Lipinski, T. Wlostowski, J. Serrano, P. Alvarez, J. Gonzalez Cobas, et al. “Performance results of the first White Rabbit installation for CNGS time transfer”. In: *Precision Clock Synchronization for Measurement Control and Communication (ISPCS), 2012 International IEEE Symposium on*. Sept. Pp. 1–6. DOI: 10.1109/ISPCS.2012.6336610.

[23] J. Serrano, E. Gousiou, M. Cattin, E. van der Bij, T. Wlostowski, et al. *White Rabbit Status and Prospects*. Tech. rep. CERN-ACC-2013-0231. Geneva: CERN, Oct. 2013.

[24] D. Beck, M. Kreider, C. Prados, W. Terpstra, S. Rauch, et al. “The General Machine Timing System for FAIR and GSI v3.1”. In: (2013).

[25] “IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks”. In: *IEEE Std 802.1Q-1998* (1999), pp. i–. DOI: 10.1109/IEEESTD.1999.89204.

[26] M. Kreider and T. Fleck. “FAIR Timing Master”. In: *Proceedings of PCaPAC 2010*. 2010, pp. 50–52.

[27] R. Bär, T. Fleck, M. Kreider, and S. Mauro. “The Timing Master for the FAIR Accelerator Facility”. In: 2011, pp. 996–998.

[28] M. Kreider, W. Terpstra, J. Lewis, J. Serrano, and T. Wlostowski. “Etherbone — a network layer for the wishbone SoC bus”. In: *ICALEPCS2011*. Oct. 2011.

[29] P. Alvarez, M. Cattin, J. Lewis, J. Serrano, and T. Wlostowski. "FPGA Mezzanine Cards for CERN's Accelerator Control System". In: *Proceedings of ICALEPCS2009*. 2009, pp. 376–378.

[30] *White Rabbit Starter Kit*. URL: http://www.sevensols.com/en/products/wr-starting-kit.html.

[31] *Simple PCIe FMC Carrier*. URL: http://www.sevensols.com/en/products/spec.html.

[32] *Spartan-6 Family Overview*. 2011. URL: http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf.

[33] *FMC DIO 5CH TTL*. URL: http://www.sevensols.com/en/products/fmc-dio.html.

[34] *White Rabbit Switch*. URL: http://www.sevensols.com/en/products/wr-switch.html.

[35] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*. 2010. URL: http://cdn.opencores.org/downloads/wbspec_b4.pdf.

[36] *OpenCores*. URL: http://www.opencores.org.

[37] T. Wlostowski, E. G. Cota, and M. Cattin. *Wishbone Slave Generator*. 2010. URL: http://www.ohwr.org/projects/wishbone-gen.

[38] M. Kreider. "The FAIR Timing Master: a Discussion of Performance Requirements and Architictures for a High-Precision Timing System". In: *Proceedings of ICALEPCS2011*. 2011, pp. 1256–1259.

[39] *Git*. URL: http://git-scm.com.

[40] P. Loschmidt, N. Simanić, C. Prados, P. Alvarez, and J. Serrano. "Guidelines for VHDL Coding". Apr. 19, 2011.

[41] *Hdlmake*. URL: http://www.ohwr.org/projects/hdl-make.

[42] J. Dedič, K. Žagar, A. Söderqvist, N. Claesson, and J. N. Rodrigues. "FPGA Development Approach for Accelerator Systems with High Integration Complexity". In: *IPAC2013*. May 2013.

[43] *Jenkins*. URL: http://jenkins-ci.org/.

LUND
UNIVERSITY

http://www.eit.lth.se