Warsaw University of Technology
Faculty of Electronics and Information Technologies
Institute of Electronic Systems

Master of Science thesis

Grzegorz Daniluk

White Rabbit PTP Core
the sub-nanosecond time synchronization
over Ethernet

Supervisor:
dr inż. Maciej Linczuk

Warsaw, 2012
Abstract

White Rabbit PTP Core, the sub-nanosecond time synchronization over Ethernet

The aim of this thesis was to develop and implement the first standalone HDL module handling the sub-nanosecond synchronization over a regular Ethernet - the White Rabbit PTP Core. The description of the module is preceded by the theoretical background explaining some basic issues related to clock synchronization and technologies used in the project. The 3rd chapter is a short overview of synchronization protocols already available for computer and industrial systems. The thesis also contains a description of the White Rabbit synchronization scheme implemented inside the module, followed by the detailed description of the White Rabbit PTP Core internals. It is divided into two parts: HDL (gateware) and C (software) layers. Last chapters contain the measurements proving the module’s synchronization performance, discuss the possible applications of the White Rabbit PTP Core and plans for further development.

Key words: White Rabbit PTP Core, White Rabbit, time synchronization, PTP, IEEE-1588, IP-core

Streszczenie

White Rabbit PTP Core, sub-nanosekundowa synchronizacja czasu w sieci Ethernet

Tematem pracy było opracowanie i realizacja pierwszego, samodzielnego modułu HDL implementującego precyzyjną synchronizację czasu z dokładnością poniżej 1 ns. Nosi on nazwę White Rabbit PTP Core i wykorzystuje standardową sieć Ethernet. Opis urządzenia poprzedzony jest wstępem teoretycznym, który ma za zadanie przybliżyć czytelnikowi tematykę synchronizacji czasu oraz technologii użytych w projekcie. Trzeci rozdział zawiera przegląd protokołów synchronizacji czasu używanych obecnie. Zawarto też opis kolejnych etapów synchronizacji w protokole White Rabbit, które zostały zaimplementowane wewnątrz modułu. Dokładny opis konstrukcji White Rabbit PTP Core podzielony został na dwie części: opis sprzętu dla FPGA (gateware) oraz warstwa oprogramowania.Ostatnie rozdziały prezentują wyniki pomiarów, propozycje zastosowań wykonanego modułu, a także plany dalszego rozwoju.

Słowa kluczowe: White Rabbit PTP Core, White Rabbit, synchronizacja czasu, PTP, IEEE-1588, IP-core
Acknowledgements

Mr Tomasz Włostowski,
Mr Grzegorz Kasprowicz,
Mr Maciej Lipiński,
Mr Tomasz Widomski,
Mr Javier Serrano,
Mr Alessandro Rubini,
...and all White Rabbit developers

Thank you for your help and cooperation!


Special thanks for both technical and non-technical support to Ms Anna Woźna, my fiancée and soon to be wife.
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC</td>
<td>PTP Boundary Clock</td>
</tr>
<tr>
<td>BMC</td>
<td>Best Master Clock algorithm</td>
</tr>
<tr>
<td>DDMTD</td>
<td>Digital Dual Mixer Time Difference</td>
</tr>
<tr>
<td>DMTD</td>
<td>Dual Mixer Time Difference</td>
</tr>
<tr>
<td>DPRAM</td>
<td>Dual-port Random Access Memory</td>
</tr>
<tr>
<td>FMC</td>
<td>FPGA Mezzanine Card</td>
</tr>
<tr>
<td>LM32</td>
<td>Lattice Micro 32</td>
</tr>
<tr>
<td>mBMC</td>
<td>Modified Best Master Clock</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>NTPd</td>
<td>Network Time Protocol daemon</td>
</tr>
<tr>
<td>OC</td>
<td>PTP Ordinary Clock</td>
</tr>
<tr>
<td>OOB</td>
<td>Out Of Band data</td>
</tr>
<tr>
<td>PCIe</td>
<td>PCI-Express</td>
</tr>
<tr>
<td>PCS</td>
<td>Physical Coding Sublayer</td>
</tr>
<tr>
<td>PMA</td>
<td>Physical Medium Attachment</td>
</tr>
<tr>
<td>PTP</td>
<td>Precise Time Protocol</td>
</tr>
<tr>
<td>SDA</td>
<td>State Decision Algorithm</td>
</tr>
<tr>
<td>SFD</td>
<td>Start-Of-Frame Delimiter</td>
</tr>
<tr>
<td>SPEC</td>
<td>Simple PCIe FMC Carrier</td>
</tr>
<tr>
<td>STM</td>
<td>System Timing Master</td>
</tr>
<tr>
<td>Sync-E</td>
<td>Synchronous Ethernet</td>
</tr>
<tr>
<td>SysCon</td>
<td>WR PTP Core System Controller</td>
</tr>
<tr>
<td>TAI</td>
<td>International Atomic Time</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>TC</td>
<td>PTP Transparent Clock</td>
</tr>
<tr>
<td>TDM</td>
<td>Time-Division Multiplexing</td>
</tr>
<tr>
<td>TSU</td>
<td>Timestamping Unit</td>
</tr>
<tr>
<td>UTC</td>
<td>Coordinated Universal Time</td>
</tr>
<tr>
<td>WB</td>
<td>Wishbone</td>
</tr>
<tr>
<td>WR</td>
<td>White Rabbit</td>
</tr>
<tr>
<td>WR FSM</td>
<td>White Rabbit Finite-State Machine</td>
</tr>
<tr>
<td>WR PTP</td>
<td>White Rabbit PTP</td>
</tr>
</tbody>
</table>
1 Introduction

Time is one of the seven fundamental physical quantities in the International System of Units. Its definition differs in classical Newtonian mechanics and Einstein’s Special Theory of Relativity, but either way time sequences events, compares the duration of events and intervals between them. Centuries ago people used an apparent motion of the Sun across the sky as the reference time and the definition of the second. Later, the first mechanical clocks were constructed. The rapid growth of electronics and computers, creation of large computer systems or satellite navigation has raised the need for more accurate time references and time synchronization.

Development of an atomic clock provided a very stable and precise timescale. The definition of a second was modified and is currently based on a transition between two energy levels of the caesium 133 atom (9 192 631 770 periods of the radiation corresponding to this transition [1]). When introduced in 1967, it was exactly equal to the astronomical second derived from a mean solar day in 1820. This created an atomic timescale called International Atomic Time (TAI). However, the Earth’s rotation is slowing down about 1.4 ms per century, which currently results in a day approximately 2.5 ms longer than in TAI timescale [2]. Therefore, a Universal Coordinated Time (UTC) timescale was created. It is based on TAI, but takes into account the variations in Earth’s rotation period. To compensate for them the standard allows a leap second insertion twice a year. Currently (February 2012) UTC is 34 seconds behind TAI.

Nowadays, a continuous growth of complex systems requiring precise time synchronization is observed. Mobile telecommunication operators have to synchronize their base stations to offer reliable, high quality voice calls and fast data transmission. Distributed computer systems operate much more efficiently when they use a common timescale. Moreover, computer systems in financial institutions are legally obligated to keep the transactions records accurately timestamped. Industrial automation is another field where a common notion of time is necessary to maintain the production process. Even today’s national power grids are synchronized to efficiently adjust the amount of produced energy to the actual power consumption.

1Leap second could also be subtracted, but this situation has so far never happened.
2Initially UTC second lasted for approx. 1.00000015 of TAI second, after some time the length of seconds become equalled but relatively frequent step-changes of hundreds milliseconds were being inserted. Finally, those ways of dealing with astronomical time were changed to leap seconds when the difference between UTC and TAI was 10 s. Up till now 24 leap seconds were inserted to UTC, which together results in 34-second offset.
Naturally, the most demanding and requiring the most precise timing are scientific applications. The examples here are particle accelerators such as Large Hadron Collider in CERN or Joint European Torus (JET) nuclear fusion reactor.

However, the ideal clocks do not exist. Most devices use internal clocks based on crystal oscillators. They are often cheap and provide poor timing performance. The frequency they generate varies with operating conditions (temperature, associated circuit components, power supply) and manufacturing tolerance. Although more sophisticated temperature-compensated (TCXO) and oven-controlled (OCXO) oscillators exist, the clocks built upon them still need to be initially synchronized.

Those examples prove that world needs the means to synchronize single devices and large, distributed systems. There are many parties constantly working on new synchronization standards and improving those that already exist. The next-generation, Ethernet-based synchronization protocol is White Rabbit. The project is being developed by the representatives of both science organizations and commercial companies. The first White Rabbit device was an Ethernet switch with WR synchronization stack developed at CERN. I am an active White Rabbit developer and was mainly responsible for developing and implementing first standalone HDL module handling the White Rabbit sub-nanosecond synchronization.
2 Theory

2.1 Clocks synchronization

In general terms, the synchronization problem can be discussed by defining two spatially separated clocks A and B (fig.1). Each of them has its own oscillator and is set initially to a different time. The latter is expected to present exactly the same time as the former. Therefore, clock A is called master or a reference, while clock B is slave, trying to synchronize itself as close as possible to the reference. To perform the synchronization those two devices can communicate with each other throughout a data link. It can be characterized by its one-way transmission delay δ.

Figure 1: Clock synchronization scheme

Let’s then introduce some ideal, absolute timescale \( t \). Clock A operates in its own timescale \( t_A \) and clock B in \( t_B \). In perfect scenario both clocks A and B would be perfectly synchronized to the absolute notion of time \( (t_A = t \) and \( t_B = t) \). However, that is never true in real world. The timescale of each clock can be mathematically described as:

\[
\begin{align*}
  t_A &= k_A t + b_A \\
  t_B &= k_B t + b_B
\end{align*}
\]

If both clocks were synchronized to an absolute timescale then \( k_A = k_B = 0 \) and \( b_A = b_B = 0 \).

Equations above show, that both devices would be ideally synchronized if two conditions were met: \( k_B = k_A \) and \( b_B = b_A \). The former means that the slave clock’s rate is the same as the master’s. It is called syntonization and in practice occurs when the local oscillators of both devices run at exactly the same frequency. Equalization of \( b \) parameters is called the offset adjustment. The first approach to synchronize slave clock to master is sending \( k_A \) and \( b_A \) using a data link. However, those values are not known, the only information available is the current time on both clocks.
(\(t_A(t)\) and \(t_B(t)\)). Therefore, master can send \(t_A\) and slave clock would receive it after the delay caused by a transmission link (\(\delta\)). The following formula could be used to calculate its offset to the reference:

\[
\Delta = t_A - t_B + \delta
\]  

After adding \(\Delta\) to the local time counter, clock \(B\) would become synchronized to \(A\).

This straightforward approach is not suitable for the real-world applications though. The main problem is the exact value of one-way link delay, which is never known. It could be measured provided that the clocks were previously synchronized, which is not true in discussed situation. That is why the synchronization protocols were developed, as they all try to estimate one-way link delay and clock’s offset as well as possible.

### 2.2 Optical Gigabit Ethernet (IEEE 802.3z)

Ethernet-based networks are widely used for sending large amount of data between computers and machines. Ethernet is well standardized by IEEE 802.3 working group. It has evolved over the years offering higher and higher data rates and few variants supporting various physical medium: coaxial cable, twisted pair, fiber. The synchronization protocol described later in this thesis uses a full duplex Gigabit Ethernet over an optical link (1000Base-LX) described in 802.3z standard. The communication medium is a standard 9/125 \(\mu m\) single-mode optic cable, where both sites transmit and receive data using the same fibre. For this purpose the Wavelength Division Multiplexing (WDM) is used with 1490 nm and 1310 nm wavelengths.

#### 2.2.1 Low-level encoding

Depending on the medium used, Ethernet standard describes the data encoding that has to be performed before bits reach the actual link. It can be quite complex for copper cables, especially high data rates (Gigabit, 10-Gigabit). However, 1000-Base-X used in the project does not have to struggle with crosstalk or electric fields around wires. Therefore, the encoding/decoding scheme is much simpler, blocks representing the sequence of operations for both transmitter and receiver are depicted in figure 2.

The first step in the encoding process is handled by a Physical Coding Sublayer (PCS). Even with no data transmitted, it continuously sends the idle pattern, keeping the link active. It is necessary for correct operation of optoelectronics and clock recovery circuits at the receiving site. PCS also encapsulates Ethernet frames.
by adding a preamble, start, and stop special characters. Those are essential for a receiver to find the boundaries of each frame. The standard defines a set of symbols - 256 data characters and the set of special symbols. The former are used to encode each frame’s payload, while the latter carry the control information e.g. start of frame, end of frame or the error occurrence.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>8B10B symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>/C1/</td>
<td>K28.5 / D12.5</td>
<td>Configuration 1</td>
</tr>
<tr>
<td>/C2/</td>
<td>K28.5 / D2.2</td>
<td>Configuration 2</td>
</tr>
<tr>
<td>/I1/</td>
<td>K28.5 / D5.6</td>
<td>Idle (correcting disparity)</td>
</tr>
<tr>
<td>/I2/</td>
<td>K28.5 / D16.2</td>
<td>Idle (preserving disparity)</td>
</tr>
<tr>
<td>/R/</td>
<td>K23.7</td>
<td>Carrier extend</td>
</tr>
<tr>
<td>/S/</td>
<td>K27.7</td>
<td>Start-of-packet</td>
</tr>
<tr>
<td>/T/</td>
<td>K29.7</td>
<td>End-of-packet</td>
</tr>
<tr>
<td>/V/</td>
<td>K30.7</td>
<td>Error</td>
</tr>
</tbody>
</table>

Table 1: Set of 1000Base-X control characters

The set of special characters is listed in table 1. /C1/ and /C2/ are used during the auto-negotiation process. Each device announces then its communication capabilities. /I1/, /I2/ are the idle characters being sent by the PCS when no data is transmitted. /R/ symbols are used instead of /I1/, /I2/ when PCS operates in a half-duplex and transmits a burst of frames to fill gaps between them. /S/ and /T/ are added for signaling where each frame begins and ends, while /V/ indicates that
an error occurred during the transmission process. Receiver should then drop the frame because it is erroneous.

After the PCS encapsulation, data is encoded with 8B/10B code. In general, each 8-bit word is replaced with a corresponding 10-bit symbol assigned to this word in a standardized lookup table. However, not all possible 10-bit symbols are used. There is a set of requirements that each encoded word has to fulfill to be a correct 8B/10B data symbol:

- The receiver’s clock recovery circuits require that each 10-bit word cannot have more than 5 subsequent zeros or ones.

- The encoded data stream cannot contain a DC component. That is achieved with “running disparity”, which ensures that the same amount of 0’s and 1’s will be transmitted.

- The bit sequence of ”comma” character cannot occur between two transmitted 10-bit symbols. The ”comma” character is /K28.5/ symbol which is the component of /I1/, /I2/, /C1/ and /C2/ special sequences. This requirement prevents the receiver from getting confused on detecting the boundaries of received words.

The final step, before data reach the transmission medium, is the Serializer. It converts the parallel 10-bit words to a serial data stream. Those bits then drive the differential output lines connected to a fiber optic transmitter. The serial data stream is 1.25 Gbit/s, generated from the 10:1 shift register using 1.25 GHz clock. This signal is produced by the PLL which multiplies ten times a 125 MHz reference clock. The internal structure of Serializer (and also Deserializer inside the receiver) is presented in figure 3.

![Figure 3: Internal structure of 1000Base-X serializer and deserializer](image-url)

The receiving site on the other end uses a Deserializer to recover a 1.25 GHz clock from the incoming data stream. It is done by a specialized PLL with phase
detector. It has to be able to lock itself to an irregular signal transitions that came from the transmission medium. Recovered clock drives a 1:10 shift register that converts a received, serial stream of bits into 10-bit words. There is however one important issue here: the shift register itself does not know the word boundaries. Therefore, the comma alignment component looks for a unique pattern of comma special character (/K28.5/). When found, its position is used to shift the data word by appropriate number of bits and hereby adjust the recovered 125 MHz clock to match the inter-word boundaries. The alignment process is depicted in figure 4.

![Figure 4: 1000Base-X comma alignment process](image)

The data encoding process, performed in the transmitter, has to be reversed at a receiving site. Therefore, it has 8B/10B and PCS frame decoders. It converts received data into raw Ethernet frames for the successive layers of a network stack. Those blocks also perform a first-level error checking such as the detection of an incorrect 8B/10B symbols or loss of communication.

### 2.3 Synchronous Ethernet

Synchronous Ethernet (Sync-E) extends the conventional Ethernet standard by adding a physical layer clock syntonization. In a regular scenario (Local Area Network) all devices use Ethernet to exchange possibly large amount of data. Such network is completely asynchronous. Every node uses its own free running oscillator. That means, the frequencies clocking transmission and reception circuits slightly differ and wander with temperature among devices. Therefore, the differences between data transmit rates for each of the two communicating parties are compensated with asynchronous packet buffers.

On the other hand, Wide Area Networks like SDH/SONET use time-division multiplexing (TDM) techniques. Originally they were designed just to transport the voice calls. However, over the years they evolved and started transmitting also data streams just as regular Ethernet. TDM networks are circuit switched, which means each node has its own time slot when it can transmit data to its destination. The whole communication channel is therefore multiplexed in time division. That differs
from Ethernet, where each node can send packets every time when necessary. If the communication is based on time slots, devices must operate synchronously. SDH/SONET additionally propagates precise timing information to each communicating party.

![System Timing Master](image)

**Figure 5: Synchronous Ethernet hierarchical network topology**

Sync-E brings the SDH/SONET-like syntonization and TDM mechanisms to a regular Ethernet. It defines a hierarchical structure, where the topmost node is called System Timing Master (STM) (fig.5). It is being syntonized to a primary clock that later becomes the reference for the whole network. Most often it is a GPS receiver or atomic clock (Rubidium, Cesium). STM syntonizes its local oscillator to the primary clock and uses this frequency to encode the outgoing data stream. Each slave node uses internal PLLs to recover the reference clock from the incoming data stream. Recovered clock is then used to discipline the slave’s local oscillator and for sending data to both: subsequent nodes (lower in the tree topology) and back to the master. This way, the reference frequency is propagated to all nodes irrespective of their location in the hierarchy.
Receiver in a regular Ethernet-based network also recovers the transmitter’s clock frequency from the incoming data stream. However, it is used only to get the correct data bits. The local oscillator is not disciplined with the recovered clock and therefore it is also not used by a slave for sending data. That is the most important issue the Sync-E addresses and is crucial for the project described later in this thesis.

2.4 DMTD phase detector

The phase difference between two clocks can be measured with a fast digital counter. It would be triggered with corresponding edges of each signal in consideration. However, the problem with this simple approach is the frequency of the digital counter. It has to be much higher than the measured clocks frequency and becomes a difficulty when considering for example 125 MHz signals that are present in a Gigabit Ethernet physical layer [22].

The Dual Mixer Time Difference (DMTD) phase detector is a much more convenient way of measuring the phase relation of two signals. The analog DMTD system is presented in figure 6. It consists of a local oscillator, two identical sets of mixers and low-pass filters connected to a time interval counter. The input is two measured clocks \(a(t)\) and \(b(t)\) which have the same frequency \(f_{clk}\). To simplify the description of operation principle they have also identical amplitudes equal to 1. The phases of \(a(t)\) and \(b(t)\) are respectively denoted as \(\phi_a\), \(\phi_b\). The internal oscillator has to be set to the frequency \(f_{offset}\) offset by few Hz from the \(f_{clk}\). The first operation the DMTD does is multiplication of input signals with the local clock. The result for \(a(t)\) is expressed as:

\[
a(t) \cdot c(t) = \cos(2\pi tf_{clk} + \phi_a) \cdot \cos(2\pi tf_{offset} + \phi_{offset}) = \frac{1}{2} \cos(2\pi t(f_{clk} + f_{offset}) + \phi_a + \phi_{offset})
\]
\[ + \frac{1}{2} \cos(2\pi t(f_{clk} - f_{offset}) + \phi_a - \phi_{offset}) \] (3)

It consists of two components: high[2] and low[3] frequency. The low pass filter removes the high frequency product leaving only signal[3]. The key feature is the fact that this down-conversion does not change the phase relation between two clocks \(a(t), b(t)\), but only affects their frequency. Let’s discuss this example again using the actual values \(f_{clk} = 125MHz\) and \(f_{offset} = 124.99MHz\). Therefore, the product is the set of two signals having frequencies respectively \(f_{clk} + f_{offset} = 249.99MHz\) and \(f_{clk} - f_{offset} = 10kHz\). The former is filtered out. Hence the conclusion is, that having the offset frequency \(f_{offset}\) very close to the frequency of the input signals \(f_{clk}\) the phase difference between two clocks can be easily measured using a simple counter.

The cost of using the analog DMTD phase detector described here is only a few discrete components (mixers, low-pass filters). The profit on the other hand is the exceptional resolution and linearity of phase measurement. The cost issue becomes particularly important in multi-port devices like White Rabbit Switch[3] which has to perform phase tracking simultaneously on all ports.

A digital counterpart of the classical DMTD detector was developed at CERN[4]. It replaces the analog mixing operation with digital sampling. Figure[7] presents the internal structure of Digital DMTD (DDMTD). It measures the phase difference between two square wave clocks \(clk_A, clk_B\) by sampling them with two D-type flip-flops (instead of using analog mixers). The offset frequency this time is generated with a PLL from one of the measured signals (equation[4]). Therefore DDMTD can be easily implemented inside an FPGA.

\[ f_{PLL} = \frac{N}{N+1} f_{clkA} \] (4)

\[ ^3 \text{The thesis only mentions the White Rabbit Switch. The complete description of its design can be found in [3].} \]
The principle of DDMTD can be described in an intuitive way by relating it to a regular caliper. It has two scales: millimeter and vernier scale. The latter is used to increase the millimeter-measurement precision by a fraction part. A typical caliper’s vernier scale is divided into ten intervals, each the length of 4.9 mm. If one of those bars stays in line with any of the millimeter bars the result of a measurement has a fraction part. In DDMTD case, the main millimeter and vernier scale are the analogy to the input and offset clock. Each time the bars from both scales become aligned, the transition in output signal occurs. For better understanding figure 8 presents the example of two measured clocks with the resulting outputs.

Figure 8: Signals generated by digital DMTD

After measuring the phase difference between two output signals the actual phase shift between input clocks is calculated by using the following formula:

\[
\phi[n_s] = \frac{n_{cycles}}{N + 1} \cdot \frac{1}{f_{clkA}}
\]

(5)

, where \( N \) is the value previously used to generate the offset clock \( f_{PLL} \) (4).
3 Existing synchronization protocols

3.1 General overview

Time synchronization in industrial and scientific applications can be performed with many different standards and protocols. Those can be custom-made solutions based on a time code, 1-PPS signal or reference frequency. Precise timing is distributed from atomic clocks (GPS, Cesium, etc.) to every node in the network and offer a very good accuracy and precision. However, such solutions require complex and laborious calibration procedures, which increase the deploying costs.

On the other hand, there are numerous well standardized synchronization systems which do not provide as high accuracies as custom-made solutions, but also do not require complex calibration procedures. Those can be used in less demanding applications to synchronize computers or industrial equipment. The examples are:

- IRIG-B - used mainly in military and airports’ instrumentation like radars or voice recording systems but also in the industry;
- IBM Sysplex - used to synchronize IBM computer servers forming large distributed systems;
- Sonet/SDH - used in telecommunication to transmit timing information between the base stations.

The huge advantage of those systems is the fact, that they are very well standardized. Therefore, the devices from different manufacturers can coexist and cooperate with each other inside a large system. All those solutions require a dedicated cabling infrastructure (separate copper or fiber wires) just to provide timing. Depending on the actual requirements this could not be a problem or (in most cases) can unnecessarily increase the cost.

Contrary to the dedicated systems, Ethernet-based synchronization protocols allow the coexistence of both timing and regular data in the same network. The most popular representatives of this group are: Network Time Protocol (NTP) and IEEE1588 (also called Precise Time Protocol). They allow building timing systems on top of already existing network infrastructure. Unfortunately, non-determinism of Ethernet results in worse synchronization performance. The active network components: switches, network adapters and OS protocol stacks add varying, nondeterministic latencies which are difficult to estimate and compensate.
3.2 Network Time Protocol

The protocol was originally started and developed by professor David L. Mills at University of Delaware. Its creation time overlaps the early years of the Internet era. Although it never has been (and still is not) the only existing synchronization protocol, it is most widely known and used around the world. NTP as the standard and reference implementation (available at http://www.ntp.org) is continuously developed by a group of volunteers from all over the world. Those are hobbyists, universities and commercial companies as well. The NTP daemon (NTPd) source code is multiplatform. It can be downloaded free of charge from its official website and compiled for Linux, Unix, BSD, Mac OS, and Microsoft Windows.

As described in 2.1 to synchronize two clocks the precise time offset between them has to be known. To obtain this information NTP uses UDP packets exchanged between hosts and addressed to a destination port 123. The advantage of using UDP messages is the possibility of communication (and as a result synchronization) between computers in both the local network and the Internet. NTP packets used for synchronization are treated by switches and routers like a regular traffic that has to be switched/routed to its destination. In basic scenario there are two parties client and server. It is intuitive that the former is the machine that adjusts its clock as close as possible to the latter’s. That is performed by periodically exchanging packets timestamped on send and reception. The overall mechanism is presented in figure 9. This process is called the round-trip. First, client sends a UDP packet that includes $T_1$ value. It is the current state of client’s system clock at the moment when message was sent. Subsequently, the reception time at a server side is marked
as \( T_2 \) and is generated in its own time scale. Those values are stored as there is no obligation of immediate reply. That means, server can switch the context if another important task was queued. To continue client’s synchronization process the second message has to be sent. This time it goes the other way round. The packet is issued by a server, timestamped with \( T_3 \) (analogously to \( T_1 \), it marks the moment of server sending the reply packet) and carries timestamps \( T_1, T_2, T_3 \). When client receives the packet and generates \( T_4 \) it has all values needed to determine the offset between two timescales discussed here. For this purpose equation [6] is used, where \( \theta \) is the offset between client’s and server’s clock and \( \delta_1 \) is the transmission delay of the first packet (sent from client to server). However, both machines should be synchronized prior measuring the \( \delta_1 \) value. That obviously is in the opposition with equation [6] where the delay value is needed to calculate the offset. Four timestamps make it possible to compute the round-trip delay (equation [7]), which is the sum of one-way delays \( \delta_1, \delta_2 \).

\[
\theta = T_2 - T_1 - \delta_1 \quad (6)
\]

\[
\delta = (T_4 - T_1) - (T_3 - T_2) \quad (7)
\]

NTP makes an assumption and estimates each packet transmission delay as a half of the round-trip delay from equation [7]. This is of course the source of inaccuracy, since paths and delays in both directions are never the same (because of the routing in the Internet and switching in LANs).

\[
\theta = \frac{1}{2}[(T_2 - T_1) + (T_3 - T_4)] \quad (8)
\]

Network Time Protocol consists not only of this basic algorithm of computing offset and estimating delay. The software daemon implements also advanced techniques of checking the incoming packets’ correctness and authenticity. It is capable of detecting time falsifiers and selecting best time source among available\(^4\). Thus each message received by NTP daemon has to fulfill some requirements before it would be passed to synchronization-related processing. Firstly, an access list exists which allows an administrator to decide which machines can exchange packets with the server. Moreover, there is a possibility to perform authentication using symmetric or asymmetric cryptographic key\(^5\). All packets that survived up to this point could be treated as originating from a trusted source, but there is no knowledge about data they are carrying. That is why NTP checks its consistency, makes sure

---

\(^4\)Similar but much simpler mechanisms (especially for selecting best source of time) are also available in protocols described later in the thesis.

\(^5\)It is called the Autokey mechanism, more information can be found in the protocols documentation[5] or in [6].
the packet is not duplicated or bogus (which means that it was not sent as a reply for the packet previously sent by the client).

_NTP_ ranges every source of time by assigning it a special number called _stratum_. _Stratum_ defines the distance of a particular machine (clock) to an atomic clock. Cesium clock, Rubidium, Oven Controlled Crystal Oscillator, Temperature-Compensated Crystal Oscillator or GPS antenna are the examples of very precise sources of time called _stratum 0_ (atomic clocks) by _NTP_. Hereby, each server synchronizing itself to one of those sources is called _stratum 1_. By analogy, each machine or workstation whose source of time is a _stratum 1_ server is called _stratum 2_ and so on. All those create a tree topology (fig. 10) with single or multiple roots and (possibly) multiple leafs.

Each computer may have multiple sources of time (they can be higher _stratum_ servers or atomic clocks). That is why _NTP_ has to implement a mechanism for selecting the best servers among all available. It consists of four main algorithms chained in a way that each of them creates the input for the next one:

- **Clock Filter Algorithm** - responsible for selecting best available data from each server. The method is to filter out the noise spikes caused by packets collisions or overloaded network. The statistical math responsible for filtering data at this level can be simplified to a statement, that in general packets with
lowest round-trip delay are considered more reliable and thus are preferred for further algorithms.

- **Select Algorithm** - its task is to evaluate which servers provide correct and trustworthy time (*truechimers*) and which of them should be discarded (*falsetickers*). In general *truechimers* are the servers which offset is located in some interval (called intersection interval). It is computed based on previous offset measurements statistics. Servers that are beyond intersection interval are regarded to deliver incorrect time and are not passed further to Cluster Algorithm.

- **Cluster Algorithm** - from all NTP servers that survived from Select Algorithm (they are called *survivors*) maximum three are chosen at this level. The algorithm tries to select sources with the highest time accuracy, which in most cases means the least jittery servers.

- **Combine Algorithm** - the final step of processing the timing data from polled servers. The role of Combine Algorithm is to merge the data from maximum three servers, that survived from the Cluster Algorithm, into a single correction factor for the client. This value is then applied to speed up or slow down the system clock.

Network Time Protocol is certainly not suitable for all applications. However, the very important advantage of NTP is that it does not need any special hardware. Every computer with a regular network card and running one of the commonly used operating systems is able to run the NTP daemon and synchronize its system clock to the UTC time. It can also provide synchronization to other machines. On the other hand, this flexibility results in not very high timing accuracy, although sufficient for many applications. In Local Area Networks two computers can be synchronized within tenths of microseconds, in the Internet - single milliseconds.

Listing 1 presents the result of calling `ntpq` diagnostic tool on a machine running NTP daemon for few tens of days. It lists the servers polled by this particular client and each row describes one of those time sources. The units of delay, offset and jitter columns are milliseconds and a fraction of milliseconds (microseconds). The good practice is to poll many servers so that the Select Algorithm can correctly detect and reject those time sources that do not provide a reliable time.

All things considered, NTP is a great tool for synchronizing computers and other machines over the Internet or local networks. It has multiple mechanisms for selecting best sources of time and rejecting the unreliable ones. Because of the authentication
remote refid stt when poll reach delay offset jitter
+192.168.1.2 .OCXO. 1 u 1 16 377 0.123 −0.011 0.013
192.168.1.6 .PPSb. 1 u 2 64 377 0.177 0.029 0.014
192.168.1.7 .IRIG. 1 u 23 64 377 0.184 −0.063 0.022
192.168.0.12 .IRIG. 1 u 40 64 377 0.198 −0.047 0.020
192.168.0.15 .PPSa. 1 u 15 64 377 0.177 0.060 0.023
vega.cbk.poznan 213.222.193.35 2 u 63 64 377 23.906 −2.597 2.107
tempus2.gum.gov .PPS. 1 u 63 64 377 18.527 −1.565 0.168
tempus1.gum.gov .PPS. 1 u 15 64 377 18.425 −1.590 1.531
212.244.36.232 192.168.0.3 2 u 27 64 377 18.295 −1.940 0.225
tik.cesnet.cz .GPS. 1 u 47 64 377 53.696 −7.426 1.295
time.nist.gov .ACTS. 1 u 1 64 377 165.966 −0.019 1.110
time-a.nist.gov .ACTS. 1 u 19 64 377 141.060 −0.830 2.986
time-b.nist.gov .ACTS. 1 u 2 64 377 140.774 1.575 3.092
www.gerstung123 .GPS. 1 u 20 64 377 53.737 0.066 0.483
rakieta2.local 192.168.0.32 2 u 9 64 377 0.084 −0.119 0.025

Listing 1: NTP synchronization diagnostic tool (ntpq)

algorithms it is a perfect synchronization protocol for safety-critical systems (e.g. military, government, etc.). The daemon is multiplatform and does not require any sophisticated hardware. This however means that all timestamps are generated in software. The consequence is that their precision is limited by the operating system architecture, current workload, and network traffic.

3.3 Precision Time Protocol (IEEE-1588)

Despite the advantages of NTP protocol, some applications require better synchronization performance. For those the Precise Time Protocol (PTP), also called IEEE-1588, was developed. The first version was released in 2002 and is commonly known as PTPv1 or IEEE1588-2002. However, most of today implementations use the second version denoted as PTPv2 or IEEE1588-2008 (IEEE standard released in 2008). That is why wherever IEEE-1588 appears in this thesis, it really relates to IEEE1588-2008 standard.

The main source of the inaccuracy in NTP is the fact that all timestamps are generated purely in software. That means that even if they are produced in the kernel space, they still depend on current system workload. This causes random, non-deterministic latencies that can not be estimated and compensated. The major improvement of IEEE-1588 over NTP is generation of timestamps in a physical(hardware) layer of the network stack. Those hardware-based timestamps eliminate the inaccuracy caused by an operating system and result in much better link latency estimation. High quality synchronization is then performed without the need for using a real-time operating system. There is still a possibility of using PTP with
software-generated timestamps. However, in this situation the synchronization results are close to NTP capabilities. That means, PTP achieves the accuracies of few microseconds while applying hardware-based timestamps results in tens of nanoseconds. On the other hand, PTP does not provide as complex statistical mechanisms (for operating with multiple time sources and detecting the false-tickers) as NTP does. Routing the PTP messages between the networks significantly lowers the synchronization quality. Those makes PTP a suitable solution only for synchronizing devices inside a local, private network.

![PTP tree-structured synchronization network](image)

Figure 11: PTP tree-structured synchronization network

The PTP synchronization network is a tree structure (fig. 11) with a single grandmaster clock at the top and many other PTP-compliant devices synchronizing their local clocks [7]. Three types of PTP equipment are distinguished in IEEE1588-2008 standard:

- **Ordinary Clock (OC)** - single-port device (eg. computer’s network card) that can be the source of reference time for a PTP network (master), or a receiver (slave) synchronizing its local oscillator to PTP network. The example of an ordinary clock is the grandmaster clock at the top of PTP hierarchy. Its local oscillator can be synchronized to an external atomic clock (Rubidium, GPS, etc.) and its network port is PTP Master.
• **Boundary Clock (BC)** - multiple-port device (eg. network switch). In fact, it could be treated as multiple ordinary clocks placed in one device and sharing a single local oscillator. It is used to connect together different segments of a PTP network. Usually one of the ports of the boundary clock is in PTP Slave state (it is used to synchronize the local oscillator), while all others are PTP Masters, so they propagate timing to other boundary and ordinary clocks lower in the tree hierarchy.

• **Transparent Clock (TC)** - multiple-port device without a servo-controlled oscillator. It appeared in PTPv2 to avoid the cumulative effect of multiple servo loops which is a result of cascading multiple boundary clocks. It simply forwards all PTP packets, but also modifies them to include the transition time through the device. By those means, the transparent clock appears to be a "wire" for the PTP messages.

PTP standard also has a mechanism of automatically determining which clocks should be Masters and which should become Slaves in the synchronization network. It is called the Best Master Clock algorithm (BMC). The decision is based on comparing the data sets describing each PTP-compliant device (eg. the stratum number, clock variance, distance from a grandmaster clock). This mechanism is continuously active so the state of each port can be dynamically changed when a device is being attached to, or detached from the synchronization network. In general, the PTP port becomes a master when BMC detects that all clocks in the neighborhood report worse quality (based on the data set they provide). In the opposite situation, port becomes slave and synchronizes its local oscillator to the best clock among devices in the network. This forms the tree-structured network synchronized to one (the best) PTP clock.

The synchronization in IEEE1588-2008 is performed similarly to the NTP protocol. It is based on exchanging timestamped Ethernet packets and estimating the delay and offset. PTP however, distinguishes two types of messages: *event messages* and *general messages*. The former are used to measure the link delay and clock’s offset, their departure and reception time is precisely timestamped. The latter carry different types of data (parameters required to establish the clock hierarchy, timestamps) that has to be passed between nodes. The complete packet exchange process is presented in fig.12. The *Sync* and *Delay Req* are the examples of *event messages* group, while the rest are *general messages*. At the beginning each host in PTP master state broadcasts periodically the *Announce* message. It advertises the sender’s clock quality description and the fact that it currently operates in a master mode.
Those values are used by the BMC algorithm. The actual synchronization is performed based on four hardware-generated timestamps \((t_1, t_2, t_3, t_4)\) associated with \textit{Sync} and \textit{Delay\_Req} messages. However, to calculate a delay and offset values \(t_1\) and \(t_4\) have to be sent to the slave node. There are two possible variations of the PTPv2 protocol: one-step and two-step clock. The former incorporates the \(t_1\) inside the \textit{Sync} message, while the latter carries it inside a separate packet (\textit{Follow\_Up}) that is sent just after the \textit{Sync}. The \(t_4\) timestamp is always sent in \textit{Delay\_Resp} message. After obtaining those four timestamps, the slave node calculates a one-way delay and offset to correct its local clock. Equations [9] 10 and 11 are used for this purpose. IEEE1588-2008 can also transfer the frequency between synchronizing nodes, so their oscillators can be syntonized. This process is based on sending many \textit{Sync} packets so that slave can compensate the frequency drift of a local oscillator.

\[
\begin{align*}
\text{delay}_{mm} &= (t_4 - t_1) - (t_3 - t_2) \quad (9) \\
\text{delay}_{ms} &= \frac{1}{2} \text{delay}_{mm} \quad (10) \\
\text{offset} &= t_2 - t_1 - \text{delay}_{ms} \quad (11)
\end{align*}
\]

Beyond already mentioned packets needed for synchronization, PTP standard also defines \textit{management} messages. Their usage is optional, but can handle additional communication e.g. to query and update each node’s data sets or in general for management purposes. Moreover, the protocol also supports the Type-Length-Value
fields for passing some other, implementation-specific data. This is a very convenient mechanism for any protocol customizations.

Using IEEE1588-2008 with hardware generated timestamps can achieve tens of nanoseconds accuracy, which is far better than NTP. However, that is possible only when all Ethernet switches in the network are PTP-compatible (are boundary clocks or transparent clocks).
4 White Rabbit protocol

4.1 The need for better performance

Despite the existence of protocols described in section 3 there are applications that require even better accuracies that those solutions could provide. One of the examples is the timing and control system for particle accelerator facilities like Large Hadron Collider (LHC) built in European Organization for Nuclear Research (CERN). These applications are spread over a very large area of tens of kilometers and consist of thousands of nodes.

The main requirements for the timing and control system for a particle accelerator are to:

• provide the sub-nanosecond synchronization for more than a thousand nodes connected with tens of kilometers of fiber link;

• establish the deterministic, synchronous and scalable network that would not require any sophisticated configuration, maintenance or calibration procedures;

• provide mechanisms for a robust delivery and deterministic routing of high priority packets; this means ensuring they would be transmitted without misrepresentation between two nodes with the delay never exceeding a certain threshold;

• the design should be fully open and not tied to any particular hardware or software manufacturer

There are some commercially available timing systems designed especially for accelerator facilities. They are custom-made and provide excellent synchronization performance. However, ”custom-made” also means that they are not standardized and therefore incompatible with each other. There is (most often) only one manufacturer of the specialized hardware and once the system is installed it has to be supported ”in house” for the entire lifetime of the control system. Those systems are also non-, or at least hardly scalable and have some architectural limitations. For example the control system that is currently used for particle accelerators in CERN (General Machine Timing system) is unidirectional: data can travel only from master to slave. The slave node does not have any means to report its state back to the master.

On the other hand, the IT world offers a very well standardized technology for exchanging data between machines and regular computers - Ethernet. It was well
established over years and still evolves offering higher and higher transfer rates (Gigabit Ethernet, 10-Gigabit Ethernet, 100-Gigabit Ethernet). There are both copper and fiber variants of the standard. The latter can be used to deploy the network over a very large area, where distance between two devices could be in the range of several kilometers. Ethernet network is very flexible, scalable and offers high data rates, but is non-deterministic. Thus it cannot be used in its traditional form to provide a very precise timing.

The White Rabbit (WR) protocol is being created as a successor of the current dedicated timing system for CERN’s accelerators (General Machine Timing system). It was initiated by CERN, but currently is developed by the cooperation of scientific organizations (CERN, GSI) and commercial companies (Elproma Elektronika, Seven Solutions, Integrasys). It combines high accuracy of dedicated synchronization systems with the scalability and flexibility of regular Ethernet networks.

White Rabbit extends the PTP protocol to achieve the sub-nanosecond synchronization over the regular Ethernet link. Although IEEE1588-2008 offers better performance than NTP, it still has some serious limitations. First of all the resolution of hardware generated timestamps in typical implementations is limited. For example, when using a Gigabit Ethernet PHY, the timestamping counter is clocked at the speed of 125 MHz, which corresponds to a theoretical resolution of 8 ns (assuming that ingress/egress latencies of the PHY are constant). Secondly, PTPv2 assumes that the underlying networking hardware runs asynchronously, so the synontization of slave’s oscillator is performed by periodically sending Sync (or Sync/Follow_up) messages. However, in order to precisely compensate the drift of the slave clock, these messages have to be exchanged frequently, generating significant network traffic. Such extra network load may be unacceptable in particle accelerators’ applications where there is a strong need for having deterministic network that could deliver critical control messages without any disturbances. Finally, PTP estimates the one-way link delay as a half of two-way master-to-master delay (equations 9–10).

To address those limitations WR combines IEEE1588-2008 with Synchronous Ethernet, digital phase measurements, and self calibration techniques.

4.2 Network topology

White Rabbit is based on Ethernet so it allows the coexistence of both timing and regular data. Therefore, the description of its network topology (figure 13) should be discussed in two parts.

The timing network structure is similar to the Sync-E (section 2.3). It is a tree
Figure 13: White Rabbit network topology

topology with at least one Master node (System Timing Master) at the top, several layers of WR Switches and slave WR Nodes. The WR Master can be synchronized to an external, precise reference clock like GPS, Cesium or Rubidium clock. That is optional though, since for some applications the only important thing might be the synchronization to a common timescale not necessarily referenced to UTC. The White Rabbit Network can also consist of more than one WR Master to ensure redundancy. The other masters are treated as backup and one of them can be selected as System Timing Master only when the one previously used fails.

The actual tree structure is created with WR Switches. Those are multi-port devices and analogous to the PTP boundary clocks. Each switch synchronizes its local oscillator to the WR Master node or to another WR Switch. It also propagates precise timing to other devices (switches and slave nodes) that are lower in the synchronization hierarchy. WR Nodes are one-port, slave or master devices (like PTP ordinary clock). They synchronize their local oscillators to the timescale propagated through the WR Network or can feed the whole WR Network with precise timing (System Timing Master).

For the regular data, WR Network acts just as a standard Ethernet-based computer network. Any node can originate packets and send them to any destination. WR Network has a flat structure for non-timing data. WR Switches behave in the same way as regular, commercially available Ethernet switches. They store the MAC addresses and forward received frames to appropriate physical ports, based on the
destination address.

4.3 Link delay model

The important issue of the clock synchronization is to accurately estimate the offset \( \text{offset}_{MS} \) and one-way transmission delays: master-to-slave \( \text{delay}_{MS} \) and slave-to-master \( \text{delay}_{SM} \) (section 2.1). The PTP protocol assumes link delay symmetry which means the one-way delays are estimated as exactly the same and equal half of the round-trip delay \( \text{delay}_{MM} / 2 \). In fact, this simplification reduces the synchronization accuracy. WR considers few different sources of link asymmetry instead and combines them, achieving better estimation of the actual one-way delays.

The **WR Link Delay Model** used to calculate the precise master-to-slave delay is presented in figure [14]. In general it can be expressed as the sum of fixed master’s transmission circuit delay \( \Delta_{txm} \), variable transmission medium delay \( \delta_{ms} \) and fixed slave’s reception circuit delay \( \Delta_{rxs} \):

\[
\text{delay}_{MS} = \Delta_{txm} + \delta_{ms} + \Delta_{rxs} \tag{12}
\]

Slave-to-master delay is expressed analogously with \( \Delta_{txs}, \delta_{sm}, \Delta_{rxm} \). These parameters can be split into two groups: values constant for a given connection \( \Delta_{txm,rxm,txs,rxs} \) and values that can vary during the link’s operation \( \delta_{ms,sm} \). The fixed values are measured when the connection is being established (process described in section [4.7.3]).

4.4 Synchronization techniques

Obtaining the slave clock’s offset \( \text{offset}_{MS} \) and \( \text{delay}_{MS} \) is based on the clock loopback technique. It is used to measure the round-trip phase shift \( \text{phase}_{MM} \) which is later needed for calculating a precise value of two-way \( \text{delay}_{MM} \). Figure [14] presents how the clock signal is transferred from master to slave and the other way round. The subsequent steps are enumerated from 1 to 5 and described below:

1. First, the reference clock (1) is encoded in the master’s data stream and sent to the slave.

2. Slave recovers the clock signal (2), which is the exact copy of the reference clock (1) delayed by \( \text{delay}_{MS} \).

3. The clock adjustment unit in the slave device uses the recovered clock, shifts its phase by a programmable value \( \text{phase}_S \), and obtains the phase-compensated clock (3).
4. Having clock (3) slave node does the same operation as master with the reference clock. It encodes clock (3) in the data stream sent to the master.

5. Master’s receiver circuits recover clock (4) and measure the phase difference between this and the reference clock (1). It is called $\text{phase}_{MM}$ and is expressed as

$$\text{phase}_{MM} = (\Delta + \delta_{ms} + \delta_{sm} + \text{phase}_S) \mod T_{ref}$$  \hspace{1cm} (13)

where $\Delta = \Delta_{txm} + \Delta_{rxm} + \Delta_{txs} + \Delta_{rxs}$ and $T_{ref}$ is the period of Ethernet physical clock (for Gigabit Ethernet it is 8 ns which corresponds to 125 MHz).

The clock signal (3) in figure 14 is also the final result of the whole synchronization and is treated by slave as a copy of the master’s reference clock. The $\text{phase}_S$ is the phase-shift which slave has to apply to match the recovered clock’s phase with the master’s reference clock. This parameter is derived from $\text{offset}_{MS}$ which is the result of computation based on a round trip delay (computed from timestamps of packets exchanged) and a precise measurement of $\text{phase}_{MM}$. However, precise estimation of the offset between two clocks is not a trivial task. There are numerous steps that are performed one after another. In general they are divided into two parts:
• **initial synchronization** - the steps that are required to determine the precise value of $offset_{MS}$ and $phase_{S}$ set point to compensate the calculated offset;

• **phase tracking** - keeps the synchronization by continuous monitoring the changes of $phase_{MM}$ and adjusting the $phase_{S}$ to follow those changes.

The complete set of synchronization steps is presented in the block diagram in figure 15. The following sections describe in detail each of them.

![Figure 15: Block diagram of full White Rabbit synchronization scheme](image)

4.5 **Syntonization**

In the initial moment the link between two devices (master and slave) is not established. Their Ethernet PHYs are continuously sending the idle pattern but they don’t receive any meaningful data. When they become connected with a physical medium (fiber cable) each of them starts to receive an idle pattern sent from accompanying site. That is recognized as the connection is present. The communication is initiated by master, which periodically broadcasts the `ANNOUNCE` message. When slave receives the packet it recognizes that master attempts to start a WR connection and responds with `SLAVE_PRESENT`. That is the confirmation for WR Master that the connected device is also WR-compatible.

Successfully exchanging `ANNOUNCE` and `SLAVE_PRESENT` packets establishes the White Rabbit link and then the syntonization may begin. The process is based on a Synchronous Ethernet (sec. 2.3). When slave receives a `LOCK` message from master it starts to recover the clock signal from a data stream. Recovered clock
is immediately used to lock the internal PLL. When it is done, WR Slave issues the \textit{LOCKED} packet to the master. When the slave’s internal PLL is locked to the master’s clock, the syntonization is complete. From that moment both devices use a clock signals which have the same frequency but different phase. Therefore the syntonization has to be followed with a precise synchronization process, which is a complex task.

4.6 Link delay measurement

The transmission delay is estimated in two steps:

- \textbf{coarse delay measurement} - based on a PTP protocol, exchanging time-stamped packets;

- \textbf{precise delay measurement} - combining the coarse delay with a precise DDMTD phase measurement.

4.6.1 Coarse delay measurement

Obtaining the coarse link delay is performed with a \textit{PTPv2} protocol (described in 3.3). The PTP packets are exchanged in a two-step clock scheme with hardware-generated timestamps. As a result of this process, four values needed for further synchronization are obtained \((t_1, t_2, t_3, t_4)\). They are generated in Ethernet PHY PCS sublayer.

The implementation of a WR device should guarantee the timestamping accuracy of a single clock cycle. It is crucial due to merging the coarse delay with a phase measurement \((\text{phase}_{MM})\) to obtain a precise delay value. The implementation of a timestamping unit inside White Rabbit PTP Core is described in detail in 6.1.2. It generates two values for each request coming from PCS: rising and falling edge timestamp. They are further denoted with ’r’ and ’f’ suffixes (eg. \(t_{2r}, t_{2f}\)).

4.6.2 Precise delay measurement

Next step of White Rabbit synchronization is obtaining a precise value of the two-way delay\(_{MM}\). It is computed based on \(t_1...t_4\) PTP timestamps, round trip phase shift \(\text{phase}_{MM}\) and slave’s PLL setpoint \((\text{phase}_S)\). The phase measurements are done with DDMTD phase detectors described in section 2.4. The reception (Rx) timestamps are generated in a clock domain which is asynchronous to the reference or compensated clock. Therefore, the algorithm enhances the resolution of \(t_2, t_4\) timestamps using \(\text{phase}_{MM}\) and \(\text{phase}_S\). On the other hand, transmission timestamps \((t_1,\)
$t_3$) are always integer since the same clock is used to transmit and timestamp the outgoing packets. Finally $delay_{MM}$ is computed using the following formula:

$$delay_{MM} = (t_{4p} - t_1) - (t_3 - t_{2p})$$  

(14)

where $t_{2p}$ and $t_{4p}$ are $t_2$ and $t_4$ improved with a phase measurement.

The full algorithm of enhancing the timestamps resolution is depicted in figure 16. It consists of multiple steps but first it has to decide whether the rising or falling edge timestamp is reliable and should be used. The selection is based on a phase shift between the recovered (Rx) and reference clock ($phase_{MM}$). However, the $phase_{MM}$ value at which the transition in $t_{4r}$ value should occur is not known. The parameter is denoted $\phi_{trans}$ and is constant for a given device. It could be measured once in the lab during the calibration of a WR device or automatically at the startup. The second option requires looking for a transition in the Rx timestamp (e.g. $t_{4r}$) while sweeping a full clock period with a built-in phase shifter.

The example illustrating the algorithm operation for enhancing $t_4$ timestamp is presented in figure 17. The $\phi_{trans}$ equals there 6.6 ns and is the intersection of the blue, sawtooth-like trace ($phase_{MM}$) and the red trace (timestamp transitions).
The algorithm selects the falling edge timestamp \( t_4 \) if a measured \( \text{phase}_{MM} \) is within \( +/− 25\% \) \( T_{ref} \) from the transition point \( \phi_{trans} \). In other situation the rising edge timestamp \( t_4 \) is considered to be reliable and used for further calculations.

Discussed algorithm also checks whether the \( \text{phase}_{MM} \) is ahead of transition point and increases the timestamp by a full clock cycle \( T_{ref} \) if necessary. This eliminates possible transition glitches by ensuring that the timestamp value would always change when \( \text{phase}_{MM} = \phi_{trans} \).

After those steps a proper timestamp is selected and the picoseconds correction part (\( \phi \)) is applied. It is the difference between the round trip \( \text{phase}_{MM} \) measured with DDMTD and \( \phi_{trans} \):

\[
\phi = \text{phase}_{MM} - \phi_{trans}
\]  

If \( \phi \) is a negative value a full clock cycle is added to the \( t_4 \) before adding the picoseconds part. The result of the algorithm is a precise timestamp \( t_{4p} \) which values monotonically increase during the device operation (thick diagonal in figure 17).

Presented algorithm is also identical for enhancing the \( t_2 \) timestamp. The only difference is that instead of \( \text{phase}_{MM} \), the \( \text{phase}_{S} \) parameter is used. However, any changes in \( \text{phase}_{S} \) affect both \( t_{2p} \) and \( t_{4p} \). Assuming that the link delay is constant, \( t_{4p} \) would increase and \( t_{2p} \) would decrease by the same value when increasing \( \text{phase}_{S} \) (RX timestamping clock is \( \text{phase}_{S} \) ahead of the trigger). Combining those two facts
together in the $delay_{MM}$ calculation, it turns out that $phase_S$ does not affects the final result:

$$
delay_{MM} = (t'_4 - t_1) - (t_3 - t'_2) = \left| \begin{array}{l} t'_2p = t_2p - phase_S \\ t'_4p = t_4p + phase_S \end{array} \right| = (16)$$

$$
= t_4p + phase_S - t_1 - t_3 + t_2p - phase_S =
= (t_4p - t_1) - (t_3 - t_2p)
$$

4.7 Link delay asymmetry

Having a precise value of two-way $delay_{MM}$, link asymmetry must be taken into account to calculate the offset between two clocks ($offset_{MS}$). WR tries to estimate a link asymmetry based on the knowledge of used transmission circuits and a medium. The following sources of asymmetry are considered:

- PCB traces and electronic components propagation delays;
- optical transceivers (SFPs) delay asymmetry;
- fiber Tx/Rx propagation asymmetry;
- delays caused by PHY chips’ internal structure.

They are all marked in figure 18. Constant delays mentioned in 4.3 are in fact the sum of PHY, circuit and SFP latencies:

$$
\Delta_{txm} = \delta_{TX,PHY} + \delta_{TX,CIR} + \delta_{TX,SFP}
$$

$$
\Delta_{rxm} = \delta_{RX,PHY} + \delta_{RX,CIR} + \delta_{RX,SFP}
$$

4.7.1 PCB and transceivers asymmetry

There are few ingredients of a circuit asymmetry: differing lengths of PCB traces, differing propagation latencies of clock distribution electronic elements and FPGA logic delays. The situation gets even more complicated, if we take into account that those delays change with operating conditions (temperature and voltage). Therefore, depending on a required synchronization accuracy, there are few possibilities of dealing with circuit asymmetries:

- measuring the circuit delays only once, during the initial calibration in the lab and consider them as time invariant - that is the easiest approach, but results in poorest estimation;
compensating the temperature and voltage variations based on a readout from built-in sensors - could potentially result in pretty good compensation, but needs developing mathematical models describing how circuit delays depend on the operating conditions;

- eliminating delay asymmetries at the design stage, that means constructing devices that operate in the same conditions and where master and slave introduce the same asymmetry or no asymmetry at all.

Eliminating asymmetries by design is very suitable for FPGA devices. That’s the case, when the build-in PHY is used and connected with internal signals to a phase detector (DDMTD). They cannot be measured with any external equipment, therefore the only suitable method is to equalize the delays on all phase detector inputs. FPGA design software allows defining various user constrains like routing delays or even allows manually place and route modules (in this case phase detector). This can be used to both equalize delays and reduce the temperature and voltage impact on the asymmetry (by placing paths next to each other).

On the other hand, the asymmetry caused by optical transceivers can be reduced with the same approach. The requirement is to use a pair of SFPs, which similarly differ on their Tx and Rx delays, on both communication ends:

\[
\delta_{TX,SFPm} - \delta_{RX,SFPm} = \delta_{TX,SFPs} - \delta_{RX,SFPs}
\]  

(17)

4.7.2 Fiber Tx/Rx propagation asymmetry

White Rabbit uses Wavelength Division Multiplexing to transmit data through a single fiber in both directions. Therefore, different wavelengths are used for sending
and receiving data stream. Using a single fiber for two-way communication simplifies the estimation of a transmission medium asymmetry. The distance is exactly the same, so the delay asymmetry is caused only by a chromatic dispersion. The refractive indexes are slightly different for two wavelengths used in the project: 1490 nm, 1310 nm. This results in different propagation velocity, which means different delays in both directions.

White Rabbit defines a custom fiber asymmetry coefficient $\alpha$. It describes the relation between master-to-slave ($\delta_{ms}$) and slave-to-master ($\delta_{sm}$) link delay:

$$\alpha = \frac{\delta_{ms}}{\delta_{sm}} - 1 = \frac{n_{1490}}{n_{1310}} - 1$$

(18)

However, refractive indexes $n$ used for calculating $\alpha$ parameter may vary between different fiber manufacturers. Therefore, more reliable method of finding the $\alpha$ is measuring $\text{delay}_{MM}$ (with PTP) and clock offset $\text{offset}_{MS}$ (with oscilloscope) in the laboratory during the initial calibration of a WR device. Those measurements have to be done with all other asymmetries compensated. The result is the equation system (19):

$$\begin{align*}
delay_{MM} &= \delta_{ms} + \delta_{sm} + \Delta \\
\text{offset}_{MS} &= \frac{\delta_{ms} - \delta_{sm}}{2}
\end{align*}$$

(19)

where $\Delta = \Delta_{txm} + \Delta_{rxm} + \Delta_{txs} + \Delta_{rxs}$. Finally, the $\alpha$ parameter is computed based on the measurements with the following formula:

$$\alpha = \frac{\text{delay}_{MM} - \Delta + 2 \cdot \text{offset}_{MS}}{\text{delay}_{MM} - \Delta - 2 \cdot \text{offset}_{MS}}$$

(20)

### 4.7.3 PHY transceiver asymmetry

The source of PHY asymmetry is the internal structure of serializer/deserializer circuit. It can be observed as a random latency between a rising edge of TX/RX clock and the corresponding transmitted/received data stream’s inter-symbol boundary. The reason of this behavior is the fact that Ser/Des modules are optimized to use less energy and faster lock to the received data stream. Fortunately for most of the chips available in the market this latency can vary each time the PLL becomes locked, but once its done (the link is active) it becomes constant. PHYs which behave differently cannot be used in WR implementations, since they would worsen the overall performance.

Two Ser/Des circuits were examined during the development of a White Rabbit Protocol: Texas Instruments TLK1221 and Xilinx GTP transceiver. The former is a separate chip that can be placed in a PCB design, while the latter is integrated inside some of the Xilinx FPGAs (e.g. Spartan 6 and Virtex 6). This evaluation showed
that we can distinguish three components inside Ser/Des that are responsible for introducing the random delays (figure 19). For Xilinx GTP transceiver it is only the comma alignment unit. It does not make a correction to the recovered (Rx) clock phase when aligning data stream to the detected inter-symbol boundaries. However, the Rx alignment latency can be relatively easy measured and compensated. Every time, when a link is established, the automatic comma alignment has to be turned off and the device should manually bit-shift the output until it detects a valid 8B/10B sequences. This method is applied to the White Rabbit PTP Core, since by default it uses Xilinx GTP Ser/Des.

On the other hand, TLK1221 had two sources of random latencies observed: RX delay caused by a digital oversampling CDR and TX delay caused by a divider feeding clock signal to a parallel-to-serial register from the internal PLL. Both those delays are measured inside White Rabbit Switch in a calibration process when the link is being established (figure 20). The PHY is forced to send the constant

---

6Version 2 of White Rabbit Switch used TLK1221 chips
sequence of K28.5 characters. This produces a 125MHz square wave (1111100000 1111100000...) and the latency is measured with a DDMTD detector as a phase relation of signal 'before' and 'after' the PHY circuit. In exactly the same way the RX latency is measured. The difference is that a communication partner has to generate the calibration pattern. The comma alignment block has to be disabled, because a continuous stream of K28.5 symbols will cause its misoperation.

4.8 Computing one-way delay and offset

The final step, after collecting all needed parameters and timestamps is computing the one-way delay and slave-to-master offset. To clarify the explanation, some formulas used to calculate $delay_{MS}$ and $offset_{MS}$ are recalled below:

$$\Delta = \Delta_{txm} + \Delta_{rxm} + \Delta_{txs} + \Delta_{rxs} \quad (21)$$

$$delay_{MM} = \Delta + \delta_{ms} + \delta_{sm} \quad (22)$$

$$\alpha = \frac{\delta_{ms}}{\delta_{sm}} - 1 \quad (23)$$

Using equation $23$, $\delta_{sm}$ can be expressed as a function of $\delta_{ms}$, $\alpha$ and substituted to $22$ to obtain the one-way fiber delay:

$$\delta_{ms} = \frac{1 + \alpha}{2 + \alpha} (delay_{MM} - \Delta) \quad (24)$$

Adding the circuit, SFP and PHY latencies to the one-way fiber delay $\delta_{ms}$ results in the final synchronization products: master to slave delay $25$ and offset $26$.

$$delay_{MS} = \frac{1 + \alpha}{2 + \alpha} (delay_{MM} - \Delta) + \Delta_{txm} + \Delta_{rxs} \quad (25)$$

$$offset_{MS} = t_1 - t_{2p} - delay_{MS} \quad (26)$$

The $offset_{MS}$ is then used by a WR Slave to feed its adjustment algorithm and clock servo. To synchronize a device with sub-nanosecond accuracy (provided that nodes are already syntonized) the following three steps are performed:

- UTC time - the current value of UTC counter is corrected (increased or decreased) by full seconds of $offset_{MS}$;

- PPS counter - the clock cycle counter that is used to generate 1-PPS signal is corrected by the remaining (after subtracting full seconds) number of full 8 ns ($T_{ref}$) cycles;

- phase shift - the phase shifter ($phase_s$ value) is corrected with the remaining, picoseconds part of the $offset_{MS}$.
4.9 Keeping synchronized

After the initial synchronization, WR Slave node has to periodically check its offset to the master and update the adjustment values if anything changes. However, once syntonized and synchronized the phase (offset) drift is caused mainly by temperature variations. That operating condition changes very slowly, so the actual adjustment may occur rarely, even once per tens of minutes.

4.10 PTPd WR extension

White Rabbit uses a PTPv2 protocol to handle the synchronization described above. It defines some additional PTP messages, a separate White Rabbit machine of states (WR FSM) and modified Best Master Clock algorithm (mBMC) to establish a WR link (Figure 21). White Rabbit PTP (WRPTP) is backwards-compatible with the original PTP specification so all WR-compatible devices can still operate as regular PTP clocks (Boundary, Ordinary). Those modifications are based on PTP customization facilities: profiles, Type-Length-Value field.

First of all the WR Link Setup was added to collect data required for WR Link Model. It consists of the communication partner identification, syntonization, measurement and distribution of WR link parameters required later to calculate precise one-way delay and offset. It is controlled with a WR FSM, while the main PTP state machine remains in PTP_UNCALIBRATED state (slave nodes) or PTP_MASTER (master nodes). The process consist of few steps before the actual synchronization occurs:

- Initially both devices are in the Idle state, WR Master periodically broadcasts Announce message.

- Slave device receives Announce messages and uses modified Best Master Clock Algorithm to determine its place in a synchronization hierarchy.

- WR Slave responds to Master with M_SLAVE_PRESENT. When WR Master receives this message it knows, that the device he speaks to is WR-compatible.

- WR Master starts syntonization process by issuing M_LOCK. That requests Slave to syntonize its local clock to the Rx clock recovered from an incoming data stream. Once it is done WR Slave responds with M_Locked

- WR Master can also request WR Slave to send a calibration pattern (continuous sequence of K28.5 symbols) by sending WR_CALIBRATE. Master thus is able to measure its PHY’s reception fixed delay. Slave generates the calibration
pattern until it receives \textit{WR.CALIBRATED} packet. The process works also the other way round, but is optional (depending on each device implementation). WR Slave can also issue \textit{WR.CALIBRATE} and \textit{WR.CALIBRATED} messages to request a calibration pattern.

- Finally, WR Master sends \textit{M.WR.MODE.ON} to indicate that the WR Link Setup process is complete and the actual synchronization can be performed.

- Timestamped PTP packets are periodically exchanged in a two-step clock scheme to obtain $t_1..t_4$.

WRPTP modifies also slightly the original PTP BMC and State Decision Algo-
rithm (SDA). The regular BMC and SDA assume that each Boundary Clock can have only one port synchronized to a grandmaster clock. That means, if this link becomes failure resynchronization is required and time fluctuations may appear. To provide a robust, sub-nanosecond synchronization, WR network requires the possibility of having multiple master clocks and smoothly switching-over from one to another when needed. This creates a tree topology with multiple roots. Each device can then have more than one port in PTP_SLAVE state and therefore synchronize to a selected master or create a timescale as a weighted average of all slave ports.
5 Definition and project goals

The implementation of the WRPTP protocol and the underlying network hardware with WR extensions is not a trivial task. This has arisen the need to create a standalone module incorporating a complete White Rabbit sub-nanosecond synchronization. That was crucial to popularize the new protocol among companies and institutions not directly involved in the development process. This module has to simplify the integration of White Rabbit into both existing and arising embedded devices and systems. Moreover, it would be also a main timing component of the White Rabbit Node [9].

Initial assumptions of the project:

- module implemented as HDL IP-core;
- black-box, plug-and-play module with additional elements (outside FPGA) reduced to minimum and requiring minimum user interference;
- incorporating a complete White Rabbit synchronization;
- based on a regular Gigabit Ethernet standard over fiber link (802.3z);
- achieving sub-nanosecond synchronization accuracy over few kilometers of fiber, monitoring and compensating link characteristics varying with environmental conditions;
- the ability of operating as WR Master or Slave, depending on the actual configuration;
- backwards compatibility with IEEE1588-2008 (two-step ordinary clock);
- working with Xilinx Spartan-6 GTP (reference hardware equipped with Xilinx Spartan 6) but also provide a discrete Ten-Bit Interface (TBI) for other PHY chips;
- providing external packets interface - there should be a possibility of using the module as a regular Ethernet MAC with timestamping capabilities;
- providing precise timing information using 1-PPS + timecode + reference frequency interface - commonly used inside already deployed time synchronization equipment;
- providing a management interface as an optional way to control the IP-core;
• optionally providing simple configuration and control interface (e.g. UART, LEDs);

• optionally providing the interface for external EEPROM chip (storing configuration of the device);

• FPGA resources minimized to coexist with user-defined HDL modules and also placing it inside the White Rabbit Node;

• not tied to any particular FPGA manufacturer or hardware platform;

• both gateware and software entirely free and open-sourced - licensed under CERN Open Hardware License, GPL/LGPL as the rest of the White Rabbit project.
The HDL IP-core called *White Rabbit PTP Core* (WRPC) described in the following sections is a result of development and implementation process done by the author of this thesis. The result as a black-box is presented in figure 22.

It has a single Gigabit Ethernet interface, currently supporting 1000Base-X optical link. There is also an ongoing research to support high accuracy synchronization over 1000Base-T copper Ethernet, but its description is beyond the scope of this thesis. The only necessary external components placed outside the FPGA chip are two digitally tunable 125MHz oscillators. They are crucial because of the DDTMD phase detectors used in WRPC. One oscillator produces a 125 MHz reference frequency for the device, while the second is the source of the offset frequency for DDMTD. All remaining blocks outside the WRPC are optional and their usage depends on a particular application. For example the I²C interface is provided for attaching an external EEPROM. It can store the device’s configuration data like MAC address and calibration parameters. WRPC by default uses a GTP transceiver integrated inside a Xilinx FPGA, but also has a generic Ten-Bit Interface. Therefore, if the project is developed for non-Xilinx device, the external PHY (e.g. Texas Instruments TLK1221) could optionally be used.

White Rabbit PTP Core is equipped with four interfaces that can be used to communicate with the rest of the system:

- **Ethernet MAC interface**: passing non-PTP Ethernet traffic between the integrated Ethernet MAC (6.1.2) and a user application. It is based on a well established, open Wishbone interface, operating in a pipelined mode. WRPC MAC interface is also capable of providing cycle-accurate Tx/Rx timestamps to the user application for every incoming/outgoing packet.

- **Wishbone slave**: for accessing the internal control registers, debugging or loading the embedded CPU firmware.

- **Timing port**: incorporating 1-PPS, timecode (UTC and nanoseconds) and the reference frequency (125MHz). Depending on the mode of operation (master/slave), timing port either serves as an input for an external reference or outputs the recovered, synchronized signals. The WRPC can also discipline an auxiliary 125 MHz tunable oscillator.

- **Control/Status pins**: the set of GPIO lines used to report the operation state of WRPC and to provide a very basic control interface. It consists of:
- UART port: used for user interaction (White Rabbit PTP Core Sync Monitor - sec. 6.3.2);
- two output pins driving Link and Status LEDs of the SPEC board (reference hardware platform, 6.3.1);
- two input pins connected to micro-switch placed on a SPEC board;
- 1-wire Master interface.

The development of WRPC consisted of two abstraction layers discussed in the rest of this section: gateware (HDL FPGA firmware) and software. The latter consists of the WRPTP daemon ported for a soft-core processor and some additional PC tools written to simplify the development process and user interaction.

Figure 22: White Rabbit PTP Core module and its interfaces
6.1 Gateware

The full block diagram of White Rabbit PTP Core HDL design is presented in figure 23. It consists of separate modules communicating with a pipelined Wishbone interface. They are all interconnected with two Wishbone Crossbars and managed by a LatticeMicro32 32-bit RISC soft-core CPU or by an external Wishbone Master. The project consists of two clock domains to loosen the synthesis timing constraints. One group are all the modules related directly to a Gigabit Ethernet, timestamping counters and 1-PPS generation. They use 125MHz clock, while all other blocks are clocked with slower, 62.5MHz signal.

![Figure 23: White Rabbit PTP Core block diagram](image)

6.1.1 Wishbone bus

The Wishbone(WB) bus was selected to ensure the communication between all modules inside WRPC. Its advantage was the fact that it is very common among open-sourced IP-core designs and well standardized [10]. Therefore, described Core could be later easily integrated with other user-defined modules. There are however two possible modes of Wishbone bus operation: standard and pipelined. The latter is characterized with better performance when transferring multiple data words in a single Wishbone cycle (block read/write operations). Managing all peripheral
modules inside WRPC require exchanging relatively rarely little amount of data, so both modes would result with the same data rates. However, the soft-core processor used the same bus to fetch instructions and data from RAM. Those actions on the other hand, are performed constantly and with the usage of the block Wishbone operations (e.g. fetching/pushing PTP packets). Therefore finally, the pipelined bus mode has been chosen for the whole design.

In fact, two separate WB communication paths are distinguished inside the WRPC. One, mentioned above, is created from all modules interconnected with two WB Crossbars and managed with the soft-core CPU. It is used to configure and control each peripheral block as well as to fetch both instructions and data from memory to LM32. The other Wishbone bus is called WR Fabric Interface and covers Endpoint and Mini-NIC interconnected with a Fabric Redirector. It is used for transmitting back and forth both PTP and non-PTP Ethernet packets. Therefore, the selection of pipelined mode also for Fabric Interface was rather obvious.

The bus consists of two communication parties WB Master and WB Slave. The former initiates and manages each communication cycle issuing read/write operations. Depending on the device role it drives different interface’s signals:

- **WB Master**
  - \textit{cyc\_o}: by asserting this line, WB Master starts a Wishbone cycle, a valid cycle is in progress until \textit{cyc} signal is driven high.
  - \textit{stb\_o}: strobe signal, indicates a valid data transfer cycle, only when \textit{stb} is asserted the rest of the master-originated signals are considered valid. There may be multiple data transfer cycles inside a single Wishbone cycle.
  - \textit{we\_o}: indicates whether the current data transfer is \textit{read} or \textit{write} cycle.
  - \textit{adr\_o}: the address array describes which word from memory or which configuration register master wants to write to/read from.
  - \textit{dat\_o}: passes the actual data word that master wants to write to slave device.
  - \textit{dat\_i}: a data word passed by WR Slave as a result of \textit{read} operation.
  - \textit{sel\_o}: determines the data granularity by indicating which part of \textit{dat\_o}/\textit{dat\_i} holds a valid data. Therefore it is possible to transfer e.g. a single byte using 16-bit width data bus.

- **WB Slave**
  - \textit{dat\_o}: data word, a result of master’s \textit{read} request.
- **ack_o**: acknowledgement signal, indicates that the requested write/read operation was done successfully and eventually that the dat_o bus contains a valid word, the result of a read operation.

- **stall_o**: used only in pipelined mode. Slave can activate this line to indicate it cannot accept more transfer requests at the moment. WB master has to wait until stall_o is deasserted.

- **err_o**: reports that an error occurred and the cycle should be terminated. It can be used e.g. to report the situation when master tries to access the area outside the slave’s address range.

The standard and pipelined mode differ when WB Master wants to read/write multiple data words in one WB cycle. In a standard mode, master has to wait for ack signal after each read/write operation request before proceeding with subsequent data word. After getting ack, the stb signal is asserted low and again high to transfer next data. This way, a separate data transfer cycle has to be initiated for every word. On the other hand, in pipelined mode WB Master asserts stb and sends a burst of read or write requests (provided that slave did not assert the stall line). WB Slave processes them one after another and asserts ack signal for the duration of a single clock cycle for each data word written/read. After completing all acknowledgments, the Wishbone cycle is done and WB Master deasserts the cyc line. The example of a Pipelined Wishbone write cycle is presented in figure 24.

![Figure 24: Pipelined Wishbone write cycle example](image-url)
Although all modules inside the WRPC use a pipelined Wishbone bus, they have the flexibility of communicating in both modes. Most of the WRPC modules are equipped with a standard WB interface, but they use \textit{wb\_slave\_adapter} component. It can convert a standard Wishbone interface to pipelined and the other way round (depending on selected configuration). The adapter is configured with a VHDL \textit{generic} construction.

\textbf{Wishbone Crossbar switch}

\textit{WB Crossbar} is used to connect multiple WB Masters with multiple WB Slaves. The White Rabbit PTP Core uses two instances of this module. This decision mainly comes from the fact that the Lattice Micro 32 soft-core has two Wishbone master interfaces, one for the instruction memory and one for the data memory. Therefore, the former needs to access only a RAM module through the crossbar. On the other hand, all peripherals have to be accessible from LM32 data Wishbone interface. Connecting all those modules to a single crossbar would end up with lots of never used logic. Theoretically the LM32 WB interface fetching instructions would also have access to the rest of modules. In practice of course those would never be used since processor gets its program only from the memory. This way placing a separate \textit{WB Crossbar} that connects all peripheral modules together saves FPGA logic blocks.

The design of the Crossbar is fully configurable. The amount of masters and slaves can be adjusted to the actual needs, reducing the FPGA resources utilized. The arbitration is based on a selection matrix, where each row corresponds to a different WB Master and each column to a different WB Slave (fig. 25a). Therefore, more than one master can simultaneously communicate with slaves provided that each of them has requested access to a different WB Slave. However, if two WB Masters would address the same device, the access is granted to the one having the lowest index (i.e. \textit{Master0} has a priority over \textit{Master1} or \textit{Master2}).

Each WB Master connected to a crossbar can make an access request by asserting a \textit{cyc} line high and selecting a WB Slave with its address bus. In general, the address bus is divided into two parts (figure 25b). A group of least significant bits describes the address of a memory cell or a configuration register in slave’s address range that WB Master wants to access. A group of the most significant bits, on the other hand determines the WB Slave device to which the request is directed. It is used by \textit{WB Crossbar} to grant access for appropriate device, but after selection, only the first part of the address is forwarded. A particular master stays locked to its selection as long as the \textit{cyc} is held high. During that, all other masters trying to reach the same device would be stalled (\textit{stall} line remains high) until the earlier cycle ends.

46
The address map of WRPC resulting from using two crossbars is presented in table 2. It is important for the soft-core processor, as well for any Wishbone Master that would like to control the internals of the WRPC through the external Wishbone slave interface.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Crossbar No.</th>
<th>Slave index</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPRAM</td>
<td>0</td>
<td>0</td>
<td>0x00000</td>
</tr>
<tr>
<td>Mini-NIC</td>
<td>1</td>
<td>0</td>
<td>0x20000</td>
</tr>
<tr>
<td>Endpoint</td>
<td>1</td>
<td>1</td>
<td>0x20100</td>
</tr>
<tr>
<td>Soft-PLL</td>
<td>1</td>
<td>2</td>
<td>0x20200</td>
</tr>
<tr>
<td>PPS-Gen</td>
<td>1</td>
<td>3</td>
<td>0x20300</td>
</tr>
<tr>
<td>SYSCON</td>
<td>1</td>
<td>4</td>
<td>0x20400</td>
</tr>
<tr>
<td>UART</td>
<td>1</td>
<td>5</td>
<td>0x20500</td>
</tr>
<tr>
<td>1-Wire</td>
<td>1</td>
<td>6</td>
<td>0x20600</td>
</tr>
</tbody>
</table>

Table 2: WRPC Wishbone address map
6.1.2 MAC layer implementation (WR Endpoint)

One of the initial assumptions about the WRPC listed in section 5 was to be able to provide a regular Gigabit Ethernet MAC functionality. On the other hand, White Rabbit synchronization itself requires Ethernet MAC equipped with some additional features. Therefore a custom implementation called WR Endpoint was developed. It performs a low-level communication by forming Ethernet frames from packets which WRPC sends out, and decoding a data stream received from PHY into understandable, high-level packets. Its internal block structure is presented in figure 26.

A regular MAC would consist of Tx Framer and Rx Deframer. However, White Rabbit requires hardware-generated timestamps and phase detector to achieve sub-nanosecond synchronization. Therefore the 1000Base-X Physical Coding Sublayer (PCS), Timestamping Unit and DDMTD phase detector were placed there. The internal structure of those constituent blocks is described further in this section. To obtain most accurate transmission and reception timestamps their generation had to be placed as close as possible to the moment when packet leaves the device and is transmitted through the medium to the communication partner. That is why the Timestamping Unit was incorporated inside the MAC layer. However, the triggering signal is originated from PCS when the beginning of each message is detected. This effectively results in timestamps generated in PHY layer of the network OSI model.

To measure phase shift between the local 125 MHz clock and clock signal recovered from the incoming data stream the DDMTD block was used. Its advantages over traditional solutions were already discussed in section 2.4. The most important is that Digital DMTD provides great resolution and linearity while being implemented almost entirely inside FPGA. The only external component is the oscillator generating the offset frequency.

Currently, there are two possibilities of connecting the Endpoint to PHY. The Ten-Bit interface (TBI), encoded with 8B/10B, supports external PHY modules like Texas Instruments TLK1221. However, WR PTP Core uses a parallel 8-bit bus and Xilinx GTP Ser/Des integrated inside the FPGA. There are also multiple interfaces at the host side:

- **Wishbone bus**: although PCS and MAC have separate Wishbone interfaces they are merged together and visible outside as a single one; it is used for configuring and managing the WR Endpoint;

- **Fabric Source/Sink**: two interfaces for transmitting and receiving Ethernet packets; they are in fact two pipelined Wishbone buses;
• **TX timestamping**: used for passing Tx timestamps of sent packets when the timestamp generation is enabled. The timestamp is available on this interface together with the port ID and frame ID to identify to which packet does it correspond.

The simplified diagram of WR Endpoint consists of DDMTD phase detector and four main modules: *1000Base-X PCS, Tx Framer, Rx Deframer, Timestamping Unit*.

**DDMTD phase detector**

It is the implementation of a Digital DMTD phase detector described in the theoretical background - section [2.4](#). This block measures a phase shift between two 125 MHz signals: the local reference clock (*phy_ref_clk*) and the clock recovered from a data stream (*phy_rx_clk*). The offset clock (*clk_dmtd*) is provided from an external, tunable oscillator.

Although the simplified block diagram discussed in [2.4](#) has presented the overall, theoretical structure of a digital phase detector, it could not be directly implemented in FPGA design. In a real world all clocks are little jittery, which causes glitches around the transitions in the DDMTD output (fig[27](#)). They could possibly worsen the phase measurement accuracy, especially for poor quality clock, a result of PHYs CDR circuit. Therefore, a special deglitching algorithm had to be used. There are
Figure 27: DDMTD glitches caused by the input clock jitter

multiple known solutions. Some of them are very simple, some are more complicated. Document [3] analyzes five techniques that have a relatively simple FPGA implementation. This shows, that the best results were achieved with a bit median algorithm(28a). For 100 phase tag samples collected, it had a maximum phase error at the level of 0.15%. That means 12 ps for 125 MHz clocks ($T_{\text{ref}} = 8\text{ns}$).

Selected deglitching algorithm was implemented with a simple finite-state machine(FSM) and a counter. The graph of FSM is presented in figure 28b. Initially a)

![DDMTD deglitcher diagram](image)

b)

Figure 28: DDMTD deglitcher algorithm(a) and a state machine(b)

it is in \texttt{wait\_stable\_0} state waiting for a stable, low level of the clock signal. It uses a free-running counter to determine if the signal was constantly 0 for a configured
amount of clock cycles. If this condition is met the transition to \textit{wait\_edge} occurs. Deglitcher stays in this state until the transition to \textit{1} begins (first glitch). It stores the current value of a free-running counter to \textit{tag\_int} register and goes to \textit{got\_edge} state. The stored value is incremented every time the input clock is at low level (0). On the other hand, a separate counter is used to determine when the signal has stabilized at logical 1. When the pre-configured threshold is met, the transition has ended, \textit{tag\_int} value is outputted as a result of the algorithm (the phase tag at which the deglitched transition should occur) and the FSM returns to the initial state \textit{wait\_stable\_0}. For a selected deglitching technique, the outputted phase tag is the transition point in figure 28, where the number of ones to the left equals the number of zeros to the right.

\textbf{WR Fabric interface}

The WR Fabric Interface is used for transmitting Ethernet packets between the WR Endpoint, Mini-NIC and external user modules. It consists of two separate Wishbone buses, running in a pipelined mode, for received and transmitted Ethernet frames. They use exactly the same signals as Wishbone bus described in 6.1.1 In this case Tx Framer incorporates a WB Slave interface (Fabric Sink), while Rx Deframer is WB Master (Fabric Source). It uses 2-bit width address bus and 16-bit width data bus. The former describes what kind of data is currently transmitted. There are four possibilities:

- **Regular data**\textbf{(0x0)}: packet header, payload;
- **Out Of Band data**\textbf{(0x1)}: e.g. reception timestamp of the frame;
- **Status word**\textbf{(0x2)}: describes the packet transmitted in a Wishbone cycle, the structure of a \textit{status word} is presented in figure 29;
- **User data**\textbf{(0x3)}: currently not used in WRPC, can be adopted for some user-specific implementations.

Each Wishbone cycle of a Fabric Interface has to be initiated by sending a \textit{status word} and the whole packet has to be transmitted in a single Wishbone cycle. It is required by the \textit{Fabric Redirector} (6.1.4). A Wishbone cycle is performed in exactly the same way as described earlier in figure 24.

\textbf{Tx Framer}

The Tx Framer block interfaces the rest of the core with Fabric Sink interface and incorporates a state machine. The module processes each packet and passes it to the
PCS sublayer. The main tasks that Framer has to perform is to calculate the CRC, strip 802.1q headers (if necessary), detect and decode the Out-Of-Band(OOB) data. As Ethernet standard specifies, Tx Framer uses CRC-32 (polynomial $0x04C11DB7$) and the resulting checksum is embedded in the frame passed to the PCS. The Out-Of-Band data received from a Fabric interface is not passed further to PCS. As soon as a transmitted frame’s ID is received it outputs this value to a Timestamping Unit.

Described module can also set the frame’s source MAC address. However, that is optional and depends on the status word value. Each message sent from WRPC can have the source MAC already defined in its header ($has$ $Src$ $MAC$ = 1) and then it would stay unmodified. Otherwise the pre-configured Endpoint’s MAC address would be inserted into the header.

A simplified graph of Tx Framer FSM is presented in figure 30. Initially it waits in the IDLE state until it becomes enabled through a Wishbone interface and PCS reports its readiness to accept Ethernet frames. The CRC-32 generator block is then reseted and enabled. After that, FSM waits in ADDR for a new packet coming from the inside of WRPC. It is also responsible for receiving and forwarding to PCS a packet’s header. Optionally, when $has$ $Src$ $MAC$ bit in status word is set to 0, Tx Framer embeds the Endpoint’s MAC into the frame header. Payload and OOB data processing occurs in DATA state. Depending on the CRC checksum existence inside the packet ($has$ $CRC$ bit of status word) the CRC generator block is used and the value of a new checksum is suffixed to Ethernet frame. At this point a complete frame is transferred and Tx Framer waits until PCS sends it out to the PHY (GAP state).

Physical Coding Sublayer

Physical Coding Sublayer (PCS) interfaces Tx Framer and Rx Deframer to a Physical Medium Attachment(PMA) sublayer. It also generates a strobing signals to the Timestamping Unit on Start-Of-Frame Delimiter occurrence in the transmission or reception path. PCS implementation consists of three sub-modules(figure 31):
Transmission PCS, Reception PCS and Auto-negotiation module.

Transmission (Tx) PCS has multiple roles. First, it generates a stream of idle pattern, when no data is being sent to keep the link active. It also generates the packet’s preamble and all Ethernet low-level signaling: Start-of-packet, End-of-packet, Error etc. (table 1). Tx PCS works in two clock domains. It provides an interfaces for the Tx Framer which uses 62.5 MHz system clock, while on the other side has to stay synchronous to the 125 MHz PHY’s transmit clock. Therefore it incorporates an asynchronous FIFO queue. Tx Framer interface puts Ethernet packet to the FIFO (62.5 MHz clock domain) while Tx PCS logic gets it out of FIFO and processes in 125 MHz clock domain.

A complete graph of Tx PCS state machine is presented in figure 32. When there are no packets to process and no other operations were requested through a Wishbone interface, the FSM spins around between COMMA and IDLE states.
Figure 32: PCS Tx state machine

sending the continuous stream of \textit{idle} pattern (K28.5/D5.6; K28.5/D16.2). There are three possibilities of interrupting this infinite loop. The software running on soft-core processor or external user module may request PCS to send a calibration pattern. The Tx PCS enters then \textit{CAL} state and starts transmitting a continuous stream of K28.7 symbols. This remains until the calibration is done (request canceled). The Auto-negotiation block may also want Tx PCS to send its configuration register (communication capabilities). It is done in the set of \textit{CR1}, \textit{CR2}, \textit{CR3} and \textit{CR4} states (aggregated to a single \textit{CR} state in the graph).

When the Tx Framer puts a new Ethernet packet to the FIFO, Tx PCS starts transmitting it to the PMA interface. First (\textit{SPD} state) it sends a Start-of-packet delimiter (\textit{K27.7} symbol) followed by a preamble(\textit{PREAMBLE} state). The last byte of preamble, called Start-of-frame delimiter, triggers the strobing signal directed to the Timestamping Unit (\textit{SFD} state). After those preparations, required by Ethernet standard, the data received from Tx Framer is transmitted (\textit{DATA} state). Gigabit Ethernet requires the complete frame to be sent without any pauses. Therefore if Tx Framer does not provide a sufficient amount of data and FIFO becomes empty before the packet ends, Tx PCS has to generate an error code K30.7 and stop the transmission (\textit{GEN \_ERROR}). Otherwise, after sending a whole packet the transition occurs directly to \textit{EPD} state and End-of-packet special character is outputted to PHY (K29.7). The last stage of frame transmission is sending the carrier extension (\textit{EXTEND}). It is a way Ethernet standard requires to maintain the minimum and maximum frame sizes to preserve the interoperability with other existing 802.3
networks.

**Reception(Rx) PCS** creates a path between PHY and Rx Deframer. It provides synchronization between the recovered Rx clock and the WRPC internal reference clock (125 MHz). Rx PCS recognizes 8B/10B special characters coming from PHY and forms a received data stream into Ethernet packets, which are passed further to Rx Deframer module. It also generates a timestamping trigger on Start-of-frame delimiter detection and embeds the Rx timestamp got from the Timestamping Unit to the received frame.

Initially the Rx PCS FSM (figure 33) stays in **NOFRAME** state, waiting to synchronize to incoming data stream. The link synchronization is performed with a separate state machine implemented by the 802.3-2008 specification. When this initial condition is met, two situations are considered. Tx PCS may receive a valid comma character (transition to **COMMA** state) or Start-of-Packet delimiter (transition to **SPD_PREAMBLE** provided that the internal FIFO queue is not full). The former may return back to the **NOFRAME** when the second received code is erroneous or misaligned. This also occurs when receiving D5.6 or D16.2 character (*idle* pattern). However, when it gets D12.5 or D2.2, the auto-negotiation configuration register is received (**CR** state). Similarly to **COMMA**, the check is made whether the value is erroneous or PCS has lost its synchronization. If that’s the case, the

![Figure 33: PCS Rx state machine](image-url)
transition back to *NOFRAME* occurs. The same transition is made if Rx PCS has received a complete auto-negotiation data.

On the other hand, when a transition from *NOFRAME* to *SPD_PREAMBLE* occurred, the FSM expects to receive a valid preamble followed by the Start-of-frame(SFD) delimiter. When there is an encoding error, or the received preamble turns to be too long, FSM goes back to *NOFRAME* state. Otherwise, a timestamping pulse is generated to the Timestamping Unit after detecting a valid SFD. A full frame’s payload together with a header is received in *PAYLOAD* state. Similarly to previously described states, the transition to *NOFRAME* occurs when an error is detected. The FIFO utilization is also monitored. When there is no space left the reception process ends with FSM returning back to *NOFRAME*. Ethernet standard forbids the *idle* pattern inside a frame, therefore in such situation the reception is terminated and *COMMA* state is entered. In regular situation though, the frame’s payload is being received until it gets the End-of-packet delimiter. Finally, after getting a complete packet, the carrier extension is received (*EXTEND* state). Here, getting any special character different than *comma* results in signaling error and the transition to *NOFRAME* state. Otherwise the whole process is correctly ended, FSM goes to *COMMA* state and a new packet is passed further to the Rx Deframer module.

**Auto-negotiation module** handles an 802.3 auto-negotiation process for both Tx and Rx paths. The implementation is based on a single state machine controlled by software with a Wishbone controller. It provides the auto-negotiation data for Tx PCS and gets the capabilities of the link partner from Rx PCS. The latter is then available through a Wishbone interface. This incorporates the information like supporting Full/Half duplex or available communication speeds.

**Rx Deframer**

The Rx Deframer module is in fact a sequence of eight operations (figure 34), where each is performed by a separate sub-block and creates input for the subsequent processing stage.

All packets received by Rx PCS are first processed inside the *Early Address match* block. It makes a first check whether just received message is high-priority or is it an Ethernet pause frame (used in flow control mechanism). This information is passed to the *Status reg insert* block. Next, there is a *Packet filter*, which is a simple packet inspection CPU driven with microcode. The filter’s role is to inspect each incoming packet and classify (based on its content) to a specific packet class. White
Rabbit PTP Core distinguishes two types of packets: PTP and non-PTP traffic. The information about the class of each message is passed to \textit{Status reg insert} block for further processing.

Rx Deframer is the module where two clock domains are crossing: the 125 MHz Rx recovered clock, and 62.5 MHz WRPC system clock. Therefore, the data flow between those two domains has to be performed with the use of an asynchronous FIFO queue similarly to Tx PCS module. This FIFO queue is called \textit{CLK align FIFO}. That means, all blocks in figure 34 to the left from this module use 125 MHz clock, while those to the right use 62.5 MHz clock. First of the latter group is the \textit{OOB inserter} module. It checks if packet has the Rx timestamp inserted by the Rx PCS. If that is true, the \textit{OOB inserter} puts Ethernet port ID into the packet’s Out-Of-Band data section. After that, the packet is verified whether no words were biased during the transmission process. That is done by generating and comparing the packet’s CRC check sum inside the \textit{CRC size check} block.

One of the last steps before a new packet is passed to WRPC is forming the \textit{status word} (figure 29). It collects the packet class (received from \textit{Packet filter}) and the information whether the packet is high priority (received from \textit{Early Address match}). However, also the always-true facts are added, like the information that packet has a CRC checksum and a valid source MAC address embedded. The formed \textit{status word} is prefixed to the packet and everything is placed in the \textit{Buffer}. That element has a single synchronous FIFO queue which is used as a buffer before data reach the final \textit{Fabric Source} interface. As already described earlier, WR Endpoint uses a Wishbone bus in pipelined mode as a packet’s fabric interface. On the other hand, the internal fabric interface(much simpler than Wishbone) is used for transferring packets from one block to another inside the \textit{Rx Deframer}. Therefore, \textit{Fabric Source} block performs a conversion between those two interfaces.
Timestamping Unit (TSU)

The Timestamping Unit generates precise timestamps for both transmitted and received packets. It inputs two asynchronous triggering pulses originated from the 1000Base-X PCS module every time the SFD is detected in Tx or Rx path. To guarantee the timestamps accuracy of a single clock cycle the generation occurs on both rising and falling-edge of clock signal. When a simple unit with only the rising-edge timestamping would be used, the clock jitter and the crossing clock domains could cause +/-1 LSB error in a timestamp value. This situation could occur when the phases of the recovered and reference clock were very close to each other. In the idealized world, the rising edge of a triggering pulse comes after an appropriate rising edge of the reference clock and a correct counter value is latched inside a register (edge 2 in figure 35). However, when clocks are jittery (which is always the case in a real implementation) there is a chance that those transitions could occur in a reverse order (edge 1 in figure 35). The timestamp value produced at that point is erroneous.

The main timestamping logic of the TSU is presented in figure 36. The design can be divided into two parts responsible for handling the rising(blue) and falling(pink) edge timestamping. The main component of the “blue” set of blocks is CNTR_R counter. It works synchronously to the reference clock or compensated clock, depending on which device is it running, WR Master or WR Slave. They are both 125 MHz signals, so it counts from 0 to 124999999 to have a period of a full second. The important issue is that whenever PCS receives a new frame, the Rx triggering pulse is aligned to another clock domain(2 or 4 in figure 14) than TSU. Therefore, the pulse length has to be extended and synchronized to the reference clock. The former operation prevents from missing impulses, caused by the metastability of the

Figure 35: Hardware-based timestamping, jittery clock problem

58
Figure 36: White Rabbit Timestamping Unit (TSU)

chain of synchronizing D-type flip-flops. The actual timestamping is performed by latching the counter value inside the DREG_R register. The same circuit is used to generate Tx timestamps. Triggering pulse is then originated from the reference clock domain (1 or 5 in figure 14). However, the synchronizing chain of D-type flip-flops is also used to assure the same (as Rx) triggering pulse propagation latency.

The falling-edge timestamps are generated from the CNTR_F, which is a simple register latching the CNTR_R counter value on the falling edge of the reference clock. It is therefore a copy of CNTR_R delayed by a half of clock cycle. On a triggering pulse occurrence, the actual value of CNTR_F is latched in the DREG_F register. This way at least one timestamp (DREG_R or DREG_F) is correct at any moment. The selection between those two values is done in software and is based on the current phase shift between the reference and recovered (Rx) clock.

The rising edge timestamp value is 28 bits width while the falling edge timestamp takes only four least significant bits of the counter. That is sufficient to detect the +/-1 LSB error caused by a clock jitter and clock domains crossing. Those two values are then concatenated into a single 32-bit word, where bits 31 to 28 carry the falling edge timestamp, while 27 to 0 - the rising edge timestamp. The result is a sub-second part of the actual PTP $t_x$. The UTC date and time (second, minute, hour, day, etc.) is generated purely in software and combined with hardware-generated part. This way, TSU design requires less registers and counter’s length.

Each timestamp generated for received packets (as a result of Rx trigger from PCS) is outputted (in the form of 32-bit word and a valid signal) back to the 1000Base-X PCS module. It is then embedded into the received frame during the
operation of the Rx PCS state-machine. However, the transmission timestamps (generated as a result of Tx trigger from PCS) are outputted to the TxTSU interface. TxTSU interface consists of the following signals:

- *port_id* - the ID of the transmitting port (always 0x0 in WRPC design since it has only one physical port);
- *frame_id* - the ID of the frame which transmission was timestamped, TSU simply passes the value got from the *Tx Framer*;
- *tsval* - Tx timestamp in the form of 32-bit word (falling-edge timestamp concatenated with rising-edge timestamp);
- *valid* - asserted high tells that the timestamp value for the transmitted frame is available (validates *port_id*, *frame_id* and *tsval* signals);
- *ack* - acknowledge input signal, TSU outputs each timestamp value until it is acknowledged.

### 6.1.3 Mini Network Interface Card (Mini-NIC)

The DMA mechanism was implemented inside the *Mini-NIC* module. It gets the transmission requests from the soft-core processor and signalizes when a new packet has been received from Ethernet MAC. Without such mechanism the CPU would have to spend many operation cycles on pushing PTP packets directly to *WR Endpoint* and fetching received messages throughout Fabric Interface. By using the DMA engine it has only to communicate with RAM to store/read packets and originate transmission/reception requests to *Mini-NIC*. After that the CPU can continue with its regular operation while DMA is responsible for transferring network traffic to/from custom Ethernet MAC module by driving a WR Fabric Interface.

When software wants to send a packet Mini-NIC gets it directly from a given location of Dual-port RAM ([6.1.8](#)) and processes it to pass to the Tx path of the *WR Endpoint* module. The other way round, an empty buffer inside Dual-port RAM has to be allocated so that Mini-NIC could write a received data directly to it. *Mini-NIC* could be connected directly to the *WR Endpoint* but in described WRPC design the *Fabric Redirector* is used between them.

Beside WR Fabric Source and Sink, *Mini-NIC* has also a Wishbone management interface, System Memory Interface and TxTSU Interface. The memory interface is used to read and write packets from/to the Dual-port RAM. It consists of the address bus, data input/output buses and *write* signal. The TxTSU interface is used to get Tx timestamps from the *WR Endpoint* and was described in section [6.1.2](#) (Timestamping Unit). Received values are then passed to the Mini-NIC Wishbone
interface so they could be fetched by a host CPU software.

The Mini-NIC implementation consists mainly of two state machines. They can operate independently of each other, which results in competing for access to the memory interface. A simple arbitration mechanism was implemented, that grants the alternate access to each state machine (i.e. Tx FSM can write to RAM in one clock cycle, then Rx FSM can read from RAM in the next clock cycle, and so on).

Transmission State Machine

The complete graph of the transmission (Tx) FSM is presented in figure 37. Initially, it stays in IDLE state waiting for a host (soft-core CPU) to start a DMA Tx transfer. The Wishbone interface is constantly monitored to detect loading a new Tx buffer’s address. In such situation, it is stored in a local register. When host issues a transmit request, the FSM transits to READ_DESC state, where it reads the first word from a buffer. It is a descriptor carrying the size of a packet and the information about packet’s structure: whether it has a source MAC address defined and the Out-Of-Band data. Descriptor is considered valid, if the most significant bit is set to 1. In case this condition is not met, FSM goes back to the IDLE. Otherwise, the status word is created (based on the information from the packet descriptor) and sent out to the WR Fabric Source interface in the STATUS state.

START_PACKET state is responsible for initiating a Fabric cycle by asserting a cyc signal and reading from RAM the first word to transmit. After that, the Tx FSM uses HWORD and LWORD states until whole data is transmitted successfully. The memory data bus is 32-bit width, while WR Fabric uses 16-bit words. Therefore, HWORD is responsible for sending the most significant half of the 32-bit
word, while \(LWORD\) - the least significant half. Both FSM states can interrupt the transmission and transit back to the \(IDLE\) when the Fabric communication partner reports an error (asserting \(err\) line high). Otherwise, when a whole packet is taken from RAM and sent out the Tx FSM enters \(OOB\) state. The behavior here depends on the packet’s descriptor received in the \(READ\_DESC\) state. When it had a valid Out-Of-Band data, it is also sent to the Fabric interface, if not, nothing more is done and a transition to \(END\_PACKET\) occurs. This final state ends the transmission cycle (deasserting \(cyc\) line) and waits to receive all remaining acknowledge pulses. It is required by a pipelined Wishbone standard (which WR Fabric interface is based on) and is a confirmation that all data was received and accepted by a WR Fabric communication partner. FSM goes back to the \(READ\_DESC\) to check if there are more packets to be sent. If not, it waits for new requests in the \(IDLE\) state.

**Reception State Machine**

The state machine responsible for receiving packets from the WR Fabric Sink interface and storing them in RAM is presented in figure 38. When the Rx buffer is allocated in Dual-port RAM and Mini-NIC reception is enabled (through a Wishbone bus) The Rx FSM waits in the \(WAT\_SOF\) for a new WR Fabric transmission cycle to be initiated. When an associated Fabric Source rises the \(cyc\) line high, the Rx FSM stalls the communication partner and transits to \(ALLOCATE\_DESCRIPTOR\). This state is necessary to write an invalid descriptor (zeroed) as a first word to the buffer and store its address in a local register. It is filled later, at the end of the transmission with a meaningful data describing a new packet.

The reception of a complete packet is performed in a single state of FSM (\(DATA\) state). It receives 16-bit data words coming from Fabric interface and composes them to create 32-bit words stored in the RAM. The address bus of a Fabric interface

![Figure 38: Rx Mini-NIC state machine](image-url)
is used to distinguish the Status word and detect if packet contains Out-Of-Band data. The whole transmission is initiated and maintained by a Fabric communication partner (Source), so due to its operation (e.g. pausing the transmission by driving \textit{stb} low) the Rx FSM might become not synchronized with the memory arbiter. In such situation the \textit{MEM\_RESYNC} state is entered for a single clock cycle to restore the synchronization. When the transmission would be interrupted, an error would occur or the transmission would end because a complete frame was received, Rx FSM enters the \textit{MEM\_FLUSH} state for a single clock cycle to write the last, pending word to the RAM.

The final state (\textit{UPDATE\_DESC} restores previously saved address of the packet descriptor allocated in \textit{ALLOCATE\_DESCRIPTION}. It is filled with a frame size, OOB presence, packet class and an error flag (if an error occurred) and stored back to the Rx buffer. At this moment, bit indicating a new packet reception (an interrupt) is set in the Wishbone slave controller and the Rx FSM stays in \textit{WAIT\_SOF} ready for receiving next request.

\subsection*{6.1.4 Fabric Redirector}

WRPC provides an external WR Fabric interface so that it could be used by user modules as a regular Ethernet MAC. However, the actual MAC functionality is implemented inside the \textit{WR Endpoint}, which is essential for sending and receiving the PTP packets to/from the \textit{Mini-NIC}. The Fabric Redirector block was developed to allow both \textit{Mini-NIC} and external WR Fabric access the Fabric interface of the \textit{WR Endpoint}. Since WR Fabric is actually the Wishbone bus operating in a pipelined mode, the module is a very simple Wishbone interconnect.

The Fabric Redirector is divided into two independent blocks, each performing the transmission in one direction (fig.39). When \textit{WR Endpoint} receives a new frame it should be passed to the \textit{Mini-NIC} or external Fabric interface - Rx PATH. The other way round, Fabric Sink interface of the \textit{WR Endpoint} has to be shared between the \textit{Mini-NIC} and external Fabric Sink - Tx PATH.

The Rx transmission path consists of a state machine that controls a demultiplexer which connects \textit{WR Endpoint}’s Fabric Source interface with an appropriate Sink. The decision is made based on the packet class inside the \textit{status word}, which is sent as the first data word in a transmission cycle. Only when a new packet’s class indicates, that it is one of the PTP messages the Fabric Source becomes connected to \textit{Mini-NIC} Fabric Sink. All other packet classes result in forwarding Ethernet traffic to the external WR Fabric Source.

On the other hand, the Tx PATH is little more complicated. It has to arbi-
trate two WRF Source interfaces competing to access a single Endpoint’s Fabric Sink. This time decision is based on monitoring the cyc lines of two Fabric Sink interfaces. Whichever starts a transmission cycle as first, gets the access to the WR Endpoint. Until the whole packet is transmitted (cyc remains high) the second WR Fabric Source is stalled and has to wait. However, when both external Fabric and the Mini-NIC try to initiate a transmission cycle at the same time, the former has a higher priority. That is caused by the demand of a White Rabbit Network determinism. That means, user’s module may be sending a high priority packets, which delivery time cannot exceed a certain boundary. Therefore, it has to be passed to WR Endpoint without unnecessary delays. On the other hand, by using Sync-E, phase tracking and hardware-generated timestamps, WRPC can stay synchronized even if some PTP messages were lost or delayed.

6.1.5 Soft PLL

To perform syntonization of the local reference clock (125 MHz) to the Rx clock recovered from a data stream the Soft PLL module was developed. The idea for this block was to put only absolute minimum of hardware and move as much as possible to software implementation. The main reason was to minimize the FPGA footprint of the WRPC. It consists of two PLLs: helper and main, but actually only of the measurement circuits providing through Wishbone parameters necessary for software algorithm (DDMTD phase detectors). The PLLs control logic was imple-
mented purely in soft-processor firmware. That was mainly possible, since no other module inside WR PTP Core implementation was connected to the CPU interrupt. Only *Soft PLL* used a single *irq* line which allowed the implementation of fully deterministic control algorithm.

The HDL implementation of *Soft PLL* contains two DDMTD phase detectors with deglitchers, period measurement block and the Wishbone Slave interface (figure 40). The DDMTD implementation and deglitching algorithm is exactly the same as in the *WR Endpoint* described in section 6.1.2. It samples the Rx or reference clock and outputs the transition tags after deglitching (*TAGRX* *TAGREF*), which are the entry points for a control algorithm.

The *Soft PLL* module uses two external VCXO oscillators tuned by DACs. First one generates a main reference 125 MHz clock (syntonized to the Rx clock), while second outputs an offset (helper) frequency for DDMTD phase detectors. This signal is also used by a DDMTD inside the WR Endpoint. Both DACs are controlled with software throughout the Wishbone Slave interface. The *Period measurement* module is a linear frequency detector used to set the DDMTD offset frequency properly, based on the Rx recovered clock. Its implementation is based on a set of counters and it outputs the frequency error periodically to the WB interface. Produced value is positive when DDMTD offset clock is slower than the Rx recovered clock. Otherwise, it is negative when DDMTD clock is faster.

The general idea of *Soft PLL* is first to adjust the DDMTD offset frequency close to the RX clock recovered from a data stream. On the other hand, when WRPC is configured to operate as a WR Master, it uses an auxiliary clock signal (e.g. from Cesium or Rubidium) instead of the recovered one. This creates a first phase-locked-loop. When this process is done, the set of two DDMTD phase detectors is

![Figure 40: Software-PLL block diagram](image-url)
used to synchronize a reference 125 MHz local clock to the Rx (or auxiliary) clock. Additionally, a phase shift compensation, calculated by a WRPTP daemon can be added there (phase value in figure 14). That creates a second PLL and outputs the compensated clock used further as a local reference for transmission circuits and 1-PPS generator (sec.6.1.6). It is an in-phase copy of a WRPTP grandmaster reference clock.

6.1.6 1-PPS generator

1-PPS generator provides a local, real-clock UTC timer and produces a 1-PPS output signal of a configurable width. It uses two counters in the 125 MHz reference clock domain: PPS counter and UTC counter. They are controlled with the Wishbone interface, so can be adjusted and their actual value can be easily read by the CPU software. The former counts from 0 to 124999999 at each clk cycle. Taking into account that this is a 125 MHz clock, the full period of PPS counter is exactly one second. When the upper range is reached, it produces a 1-PPS pulse which can be extended to a given (configured earlier throughout Wishbone) width. The UTC counter is incremented each time the PPS counter overflows. Therefore, it counts full seconds and is used by the WRPC as a local UTC clock. Its value is fetch when the software needs a current time. For example, the readout from 1-PPS generator is merged with the value generated by the Timestamping Unit (inside the WR Endpoint) to form a precise packet’s timestamp used inside the WRPTP daemon.

There are two possibilities of changing the value of UTC and PPS counters. Each one can be immediately set to a predefined value by writing it to an appropriate Wishbone register. This occurs independently of the current counter value. Another option is adjusting PPS or UTC value. The adjustment of the PPS counter is done by waiting until it overflows and then setting its value to the one written to a Wishbone register. On the other hand, this operation differs slightly when requested on the UTC timer. The adjustment register is added to a current UTC counter value at the moment it would normally be incremented (when PPS counter overflows). Both counters are adjusted by a WRPTP daemon as a result of a synchronization process.

6.1.7 Lattice Micro 32 soft-core processor

White Rabbit PTP Core was equipped with a soft-core processor to manage all internal peripheral modules and to run the White Rabbit PTP software synchronization daemon. Numerous CPU implemented in Verilog and VHDL were considered. The one selected had to be licensed under free, open-source license (GPL, LGPL, BSD,
etc.), utilize the minimal FPGA footprint while offering a good performance. Moreover, soft-core processors with Wishbone bus were preferred since it was selected as a main communication interface for all WRPC blocks. The most promising ones were initially ZPU, LatticeMicro32 and Leon3. However, the last one was finally rejected. Although it provided quite good customization means, it was too large after turning on all options required by the Core. It utilized 3010 LUTs of Altera Stratix III FPGA, compared to 2112 LUTs for LatticeMicro32 and 1055 LUTs for ZPU.

First implementation of WRPC was based on the ZPU core. It was the smallest one of the cited three. It actually did not have any particular CPU architecture, but was implemented as a simple machine of states fetching, interpreting and executing instructions from RAM. However, during the development process it turned out that the CPU was a little buggy and its performance was not enough to fulfill the WR PTP Core needs. Therefore it was replaced with Lattice Micro 32, and this soft-core CPU turned out to be perfect for the project.

LatticeMicro32 (LM32) is a soft-core processor developed by Lattice Semiconductor Corp. and optimized for this manufacturer FPGA chips. It controls the synchronization and operation of all modules inside the White Rabbit PTP Core. LM32 is written in Verilog and open-source licensed, so can be synthesized also for Xilinx or Altera devices. The general architecture of the core is presented in figure 41. However, LM32 can be configured depending on the actual needs. Thus, the functional blocks may be added or removed to fit the particular application.

The processor is 32-bit, big-endian, Harvard architecture. It has a separate instruction and data bus, which can be accessed simultaneously. They are both 32-bit width as well as all general purpose registers inside the core. The instruction and data bus are Wishbone Master interface. Originally LM32 implements a standard mode Wishbone, but for the purpose of the WRPC design a pipelined wrapper was added. As shown earlier in WR PTP Core architecture(fig.23), both instruction and data bus of LM32 are connected throughout the WB Crossbar to one port of the Dual-port RAM. Moreover, all other WRPC peripherals are also placed in the LM32 address space which is flat and byte-addressable. They can be configured and maintained with a regular read/write cycles as operating on a chunk of memory.

Lattice Micro 32 processor uses a Reduced Instruction Set Computer (RISC) architecture. All instructions are four bytes wide, so the Program Counter (PC) has always two least significant bits zeroed. Therefore, the instruction set is simpler and the core achieves greater performance. The latter is additionally improved by using 6-stage pipeline processing. It uses bypassing and pipeline interlocking for detecting
and avoiding data hazards (the latter detects read-after-write hazards and stalls the pipeline when necessary to resolve the hazard). This results in most instructions being effectively executed in a single cycle. The LM32 pipeline consist of the following stages:

- **Address**: calculates the next instruction’s address and sends it to the instruction cache (if present);
- **Fetch**: reads the instruction from memory;
- **Decode**: decodes fetched instruction and gets needed operands from the Register File or it is bypassed from the pipeline;
- **Execute**: performs the desired instruction;
- **Memory**: is a second execution stage for more complex instructions (e.g. multiplications);
- **Writeback**: writes the instruction’s result back to the Register File.

As already mentioned, LM32 can be configured to match the needs of the actual application. There are multiple Verilog parameters to enable/disable particular functional blocks. The configuration of the processor used inside the WR PTP Core expands the minimal core by:

- **PL_MULTIPLY_ENABLED**: enables a pipelined multiplier, as a result \textit{mul}
and *muli* instructions are available and they take three cycles to complete (for comparison LUT-based multiplier needs 32 cycles);

- **PL_BARRIER_SHIFT_ENABLED**: enables a pipelined barrel shifter, adds shift instructions *sr, sri, sru, srui, sl, sli* - they take three cycles to complete;
- **SIGN_EXTEND_ENABLED**: adds sign-extension instructions *sextb, sexth*;
- **INTERRUPTS_ENABLED**: enables interrupts module with 32 external interrupt lines. However, WRPC uses only one LM32 *irq* input connected to the Soft-PLL. Therefore the Soft-PLL interrupt handling time is fully deterministic.


### 6.1.8 Dual-port RAM

The soft-core processor needs a data and instruction memory. On the other hand, the Mini-NIC requires RAM for storing received packets and reading packets which have to be sent. Those two modules also have to exchange data (packets), so the same memory area should be accessible by both of them. That is why, WR PTP Core incorporates a module called Dual-port RAM (DPRAM). It utilizes some part of the Block-RAM (BRAM) that is available inside the FPGA chip.

The reference device for the project (Xilinx Spartan-6 *XC6SLX45T*) has 116 blocks of BRAM, each of the size of 18 kb [12]. They are dual-port, which means two parties can simultaneously access memory cells. WRPC utilizes only 64kB of RAM, which still leaves plenty of resources for user-defined applications. Two ports of BRAM area were connected to the Mini-NIC and LM32, so that those modules could access it independently of each other.

The Mini-NIC exchanges data with DPRAM using a regular memory interface (*data in/out, address out, write enable out*). On the other hand, Lattice Micro 32 communicates with peripherals using a pipelined Wishbone bus. Therefore, one of the DPRAM ports became connected to a pipelined Wishbone Slave adapter.

DPRAM acts not only as data, but also instruction memory for LM32. Thus, it had to be initialized with an appropriate processor’s firmware. In the WR PTP Core design, this could be done from a host system by using an external Wishbone interface e.g. connected to PCI-Express bus (LM32 firmware loader described in section 6.2.2). However, there is also a function inside DPRAM VHDL implementation that initializes memory with a given file during synthesis process of the core. Expected initialization file should have the syntax presented in listing 2. It first determines
the address in a memory (32-bit word addressing) and then the 32-bit value that has to be written there.

Loading CPU firmware throughout the external Wishbone bus is especially convenient when writing a new software and debugging since it does not require re-synthesizing the whole WRPC project (which takes quite a lot of time). However, when system is stable, the initialization function makes it possible to give a final user a single bitstream file with FPGA and LM32 firmware integrated.

Listing 2: DPRAM initialization file example

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x&quot;98000000&quot;</td>
</tr>
<tr>
<td>1</td>
<td>x&quot;D0000000&quot;</td>
</tr>
<tr>
<td>2</td>
<td>x&quot;D0200000&quot;</td>
</tr>
<tr>
<td>3</td>
<td>x&quot;78010000&quot;</td>
</tr>
<tr>
<td>4</td>
<td>x&quot;38210000&quot;</td>
</tr>
</tbody>
</table>

6.1.9 WR PTP Core System Controller

The WR PTP Core System Controller (SysCon) is a Wishbone module integrating few tiny peripherals. They are used by the WRPC, but implementing each of them in a separate module would be a waste of resources. They are all shortly described below.

Reset Register

WRPC has two resetting signals: the LM32 reset, and Net reset. The former is connected only to the Lattice Micro 32 core, while the latter is used to reset all other peripherals. That was required by the LM32 firmware loading feature. When a user wants to initialize DPRAM through the external Wishbone Slave interface, it has to hold the Core in a reset state while writing the memory. Other modules (WB Crossbar, DPRAM) cannot naturally be hold in reset since then Block RAM could not be accessed then. Therefore, writing 0x1deadbee to the Wishbone Reset Register resets the LM32 core, while writing 0x0deadbeef deactivates the reset line and processor starts fetching instructions.

GPIOs

There are 7 GPIO lines which can be driven or read throughout GPIO Wishbone registers. First of all, there is the Net reset already mentioned above. It is used to reset all peripheral modules inside the WRPC. The reference hardware platform (described in section 6.3.1) has two LEDs (Link and Status) and two micro-switch
buttons. The software running on the Lattice Micro 32 uses GPIO WB registers to set the LEDs and readout the buttons state. Those can be used as a very simple, but legible method of reporting the WRPC internal state.

There are also two lines (SCL, SDA) reserved for software implementation of $I^2C$ bus. They are however currently not used.

**Hardware Feature Register**

Provides the information about a WRPC hardware configuration to the LM32 software. Currently only the size of available RAM is passed there, as a 4 bit value starting at $0x0$ (which means 64 kB) and going up to $0xf$ (1024 kB).

**Timer counter**

Is the implementation of a simple timer that works independently and is not adjusted during the PTP synchronization (as PPS and UTC counters inside the 1-PPS Generator). It is used by the software wrapper layer (sec. ) for dispensing time intervals (e.g. for periodically reading the DDMTD measurements).

**6.1.10 UART interface**

White Rabbit PTP Core has also a UART interface. It is especially useful for debugging purposes, when the information from the WRPTP daemon can be outputted to the user console. The module consists of a physical UART interface and a virtual UART (vUART). When the latter is used, the messages outputted by WRPC can be fetched throughout the external Wishbone interface (e.g. using PCI-Express core - sec.6.1.12).

The vUART is a FIFO queue located in a Block RAM inside FPGA. It can store up to 1024 8-bit words (1024 characters). The vUART works only in one direction from WRPC to the user console. Therefore, there is no interaction possible, messages can be only outputted from the LM32 software.

On the other hand, physical UART block implements characters transmission in both directions. However, only $RxD$ and $TxD$ lines of RS-232 are used. There is for example no handshake implementation ($CTS$, $RTS$). The physical UART is fixed to use 8 data bits, one stop bit and no parity bit. The CPU can pass bytes for sending, get received characters and configure a baudrate using a Wishbone interface.

The UART module consist of three main components(fig.42): Baudrate Generator, Transmitter and Receiver. The Baudrate Generator is a frequency divider, that (based on a configured baudrate) produces pulses for the Tx and Rx circuits. The
signal outputted to the Transmitter is directly related to a desired baudrate. Tx block sends one bit on each pulse got from the Baudrate Generator. When CPU writes a new byte and activates the UART Transmitter circuit, it first sends a start bit and then 8 bits of data word followed by a single stop bit.

On the other hand, a pulse signal generated for Receiver module is eight times faster than Transmitter’s. That is caused by the fact, that the Receiver has to probe the RxD line multiple times per one received bit. It does not precisely know the bits boundaries, but it cannot lose any data. Each time when a new byte was received, it is available through a Wishbone register and the indication bit is set to inform the CPU about new data in Rx buffer.

User can decide which UART implementation (physical or virtual) to use by setting the value of VHDL generic before synthesizing the WRPC project.

### 6.1.11 1-Wire master interface

The presence of 1-Wire Master interface is not mandatory for WRPC operation. Therefore it could be removed when synthesizing for a small FPGA chip or when user modules require more resources. The reference hardware platform (sec.6.3.1) has *Dallas DS18B20* 1-Wire digital thermometer integrated. The bus standard defines that each 1-Wire compatible device should have a unique identifier. That feature could be used to assign each White Rabbit PTP Core implemented on a SPEC board also a unique identifier derived from the *DS18B20* ID. That would be useful when multiple WR Nodes were deployed in a distributed WR Network. The communication with the digital thermometer required implementing a 1-Wire Master controller.

White Rabbit PTP Core uses 1-Wire Master implementation available from *OpenCores* [13]. It is an open Verilog code licensed under *LGPL 3*. Originally it
was designed for Altera-based systems, so the module is interfaced with the Avalon bus. In fact, it used only a subset of the Avalon signals (read, write, readdata and address), so conversion to Wishbone was not problematic.

6.1.12 PCI-Express with Gennum GN4124 core

The module described in this section is not a part of the White Rabbit PTP Core. However, it was connected to the project to combine the External Wishbone Slave interface (fig.23) with the PCI-Express bus. Therefore the future user or developer was given the opportunity to access each and every HDL module inside WRPC, configure internal registers or initialize DPRAM with LM32 firmware by using a regular Personal Computer.

GN4124 Core was developed at CERN. The module is a controller for Gennum GN4124 chip integrated on the SPEC board, which is a 4 Lane PCI-Express to local bus bridge. The IP-Core provides a pipelined Wishbone Master interface connected to the GN4124 local bus. A very brief description of GN4124 IP-Core is presented here. The complete documentation is available in [14].

Figure 43 depicts the internal architecture of the module. It consists of a regular pipelined Wishbone Master interface and a DMA Engine. The latter can be used for

![Figure 43: Gennum GN4124 core architecture](image-url)
transferring a large amount of data with high data speed rates between the PCIe bus and HDL modules. However, for managing and debugging the WR PTP Core using the regular pipelined Wishbone Master interface was much more suitable and straightforward.

The local bus of GN4124 chip is 16-bits width, so there is a Multiplexer and Demultiplexer to convert 32-bit width Wishbone. They form 32-bit words using 16-bit words received from the chip and divide 32-bit words into 16-bit words before they are sent to the GN4124. After Demultiplexer, all data travel through a Packet decoder. It strips PCIe packets coming from the local bus and extracts header, address, data, byte enables and timing controls. Thus, each packed may be addressed to the Wishbone Master or DMA engine.

The Wishbone Master block implements a Wishbone interface connected further to WRPC. It converts a PCIe write requests into WB writes and PCIe reads to WB reads. It uses a clock signal separated from the rest of the IP-Core. In described project it is fed with the 62.5 MHz WRPC system clock. Therefore, Wishbone Master controller incorporates a FIFO queue to synchronize two clock domains. L2P DMA Master and P2L DMA Master are used to perform DMA data transfers to/from the memory of a PCI-Express host. The outgoing data transfers originated by DMA engine or Wishbone Master controller are arbitrated inside the Arbiter module. It grants GN4124 local bus access to the first requester and this lasts until the end of packet.
6.2 Software

The HDL design of White Rabbit PTP Core describes a hardware layer inside the FPGA chip. It contains a soft-core processor Lattice Micro 32 and peripherals needed to perform high-quality synchronization. However, to become fully operational the CPU needs a firmware (WRPTP daemon) and a firmware loader.

6.2.1 Wrapped WRPTP daemon

Originally the implementation of WRPTP daemon was done by White Rabbit developers for the White Rabbit Switch. It was based on an open-sourced PTPv2 daemon\cite{15}, but also WR modifications described in section \ref{sec:wrptp} were added. The implementation was successfully running on a WR Switch but was unsuitable for the White Rabbit PTP Core. The main difference between those two devices is the CPU. The former used an ARM processor, running an embedded Linux operating system. WRPTP daemon, after cross-compilation could be executed the same way as on a regular PC. Floating point operations were performed, numerous library functions and operating system mechanisms were available.

On the other hand, WRPC has a simple Lattice Micro 32 microprocessor and no operating system underneath executed software. To minimize the FPGA resource footprint, there is also no floating point unit (FPU) and very limited amount of memory is available (64 kB). Those restrictions created a necessity for porting the WR Switch PTP daemon implementation. The result, WRPTP for White Rabbit PTP Core, consists of three software layers (figure \ref{fig:software_layers}):

- **device drivers** - the set of functions to manage each of the HDL module implemented inside WRPC;

- **wrapper** - custom implementations of the operating system’s mechanisms, library functions and some daemon functions that use them; additional debugging and monitoring functions;

- **PTP daemon** - modified to remove floating point arithmetic and to use functions and mechanisms from the wrapper.

Device drivers for HDL peripherals can be found in `dev/` directory of the software git repository, while header files with functions and structures declarations are collected inside `include/`. There are drivers for the following modules:

- **Endpoint** - functions for initializing the module’s registers (setting MAC address, configuring DDMTD, etc.); accessing the PCS sublayer (MDIO) registers; enabling transmission/reception circuits; checking the link status (up /
down); enabling/disabling calibration; getting $\Delta_{tx}, \Delta_{rx}$ delays - the result of a calibration process; getting the packets’ statistics.

- **I$^2$C bus** - as stated in 6.1.9, WRPC provides two regular GPIO lines for I$^2$C bus (SDA and SCL), therefore the actual bus communication was implemented in the software driver; it provides functions for initializing the bus; discovering available slave devices; sending and receiving bytes.

- **Mini-NIC** - functions for initializing and disabling the module; passing base address and size of the Rx and Tx buffers allocated in RAM; polling whether a new packet was received; fetching received frames from allocated buffer and extracting Rx timestamp embedded inside the packet; transmitting frames; getting packets’ statistics.

- **1-PPS generator** - functions for initializing the module; adjusting PPS and UTC counters; enabling 1-PPS output; getting the current time (reading the current state of UTC and PPS counters).

- **Soft-PLL** - functions for enabling and disabling PLLs; getting the PLL status; checking if the PLLs are locked; setting the phase shift that Soft-PLL should add to the local reference clock.

- **SysCon** - functions for initializing and getting the actual state of the SysCon
timer; setting the state of GPIO outputs; reading the state of GPIO inputs; function suspending program execution for a configured delay value.

- **UART** - functions for initializing the module; setting baudrate; sending a single byte and a string as well; reading an incoming byte; polling the UART’s Receiver.

The *wrapper* part of the LM32 firmware uses those drivers and adds a custom implementation of some library functions that were essential for the *WRPTP* daemon. This includes the family of `printf()` functions, which were very useful during the debugging process to print out messages to the console. The original `printf` from the standard library is a huge function with numerous formatting options, consuming quite a lot of memory. The custom implementation for WRPC writes messages to the UART or vUART module. It understands `%s, %c, %d, %u, %x` formatting options, which are completely sufficient for debugging and reporting the state of the core. Another library function that was redefined is `usleep()`. That was mainly because there is no operating system and it had to use the synchronization-independent timer built into the *SysCon* module to actively wait for a given period of time. Additionally, few inline functions were also defined to convert from host to network byte order: `htons()`, `htonl()`, `ntohs()`. The actual conversion depends on whether the *WRPTP* daemon is compiled for a WR Switch or WRPC. Since LM32 is big-endian, no byte swapping is required and the word in network byte order is exactly the same as in the host byte order.

*WRPTP* uses Linux sockets mechanism and a network interface to send and receive packets. Therefore the *wrapper* contains also a custom sockets implementation. It is based on *my_socket* structure defined to hold the information about each socket in use. It can store e.g. a local MAC address, the IP bound to the port, interface name, parameters for Rx timestamps linearization and a packets queue. That last element is a 200 bytes long circular buffer used as a FIFO storing packets after *Mini-NIC* receives them and before PTP daemon processes them. It has two pointers: `head` and `tail`. The former is the beginning of a free space, where the new, incoming packet can be stored. The latter is the first byte of the oldest packet in the queue. There is an `update_rx_queues()` function called in each round of a main infinite loop. It checks whether *Mini-NIC* has received a new packet and copies it to the circular buffer queue. It is still possible, that PTP daemon would lose incoming message when it did not fetch data from FIFO for a significant period of time. However, additional buffering layer makes it less likely.
Functions inside the WRPTP that use the wrapper mechanisms and functions provided by device drivers were externalized from the daemon source code and collected in one file (libposix/freestanding-wrapper.c) inside the ptp-noposix git repository\textsuperscript{7}. That was done mainly of the practical reasons, since it is much easier to modify only one file during development and debugging process. It consists of PTP functions handling a network interface (ptpd_netif_*()) that initialize the interface and sockets, manage calibration and syntonization process - SoftPLL locking to Rx clock. There is a method for sending PTP packets (ptp_netif_sendto()) which forms a new message based on a given payload, address etc., triggers the Mini-NIC and returns a Tx timestamp. Received packets are fetched from the socket’s FIFO and passed for further processing inside the PTP daemon by ptpd_netif_recvfrom().

It is worth to mention one of the functions called from ptpd_netif_recvfrom(). The linearize_rx_timestamp() is the implementation of the decision making process described in section \textsuperscript{4.6.2}. It chooses a reliable reception (Rx) timestamp (rising- or falling-edge timestamp).

The main daemon’s function is protocol_nonblock(). It is called in each round of the main infinite loop and executes the rest of necessary functions. Normally, the code inside this function is looped after executing a PTP daemon from Linux/Unix shell. However, the WRPC software has to perform more operations than just running the daemon itself. First of all there is the update_rx_queues() function. It has to be executed periodically to get new packets from Mini-NIC. Secondly, there is also a small piece of code implementing White Rabbit PTP Core Sync Monitor (\textsuperscript{6.3.2}). It outputs the current state of WRPC and the synchronization quality to UART in a nice and clear form presented in figure \textsuperscript{46}. All things considered, the main() function of the project contains the early hardware initialization code and an infinite loop updating GUI, Rx queues, and PTP protocol state.

\textbf{6.2.2 LM32 firmware loader}

LM32 firmware loader is a small program that can be run on a PC computer with SPEC board\textsuperscript{6.3.1} in the PCIe slot. It can be found in the tools/ directory of the software git repository\textsuperscript{8}. The loader reads the binary file of a compiled LM32 firmware and sends it throughout the PCIe bus and GN4124 core to initialize Dual-port RAM (\textsuperscript{6.1.8}).

This software uses GNURabbit Linux kernel driver for GN4124 \textsuperscript{16}. Loading the rawrabbit.ko module into the running kernel creates a /dev/ruwrabbit special

\textsuperscript{7}See appendix A.1 for the address of ptp-noposix git repository
\textsuperscript{8}See appendix A.1 for the address of LM32 firmware source code git repository
file. It also provides the \textit{rr\_io} library, which defines a set of functions to handle the operation on newly created file e.g. \textit{rr\_init()}, \textit{rr\_writel()}, \textit{rr\_readl()}. They can be further used to communicate with GN4124 HDL IP-Core.

As mentioned in GN4124 core description (sec. 6.1.12), it consists of the DMA engine and pipelined Wishbone Master interface. Therefore, each read/write request directed to the particular address of /dev/rawrabbit must define to which interface does it correspond. It is done by adding an appropriate base address - 0x80000 for Wishbone Master. Thus the modified address map of the WR PTP Core modules is listed in table 3.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPRAM</td>
<td>0x80000</td>
</tr>
<tr>
<td>Mini-NIC</td>
<td>0xA0000</td>
</tr>
<tr>
<td>Endpoint</td>
<td>0xA0100</td>
</tr>
<tr>
<td>Soft-PLL</td>
<td>0xA0200</td>
</tr>
<tr>
<td>PPS-Gen</td>
<td>0xA0300</td>
</tr>
<tr>
<td>SYSCON</td>
<td>0xA0400</td>
</tr>
<tr>
<td>UART</td>
<td>0xA0500</td>
</tr>
<tr>
<td>I-Wire</td>
<td>0xA0600</td>
</tr>
</tbody>
</table>

Table 3: GNURabbit WRPC address map

LM32 loader accepts the location and name of a firmware binary as a runtime parameter. The program starts by executing an initialization function \textit{rr\_init()} from \textit{rr\_io} library. It simply opens the /dev/rawrabbit device file for reading and writing. When the operation was successful (which means the SPEC board is present) and the firmware file is open for reading, it uses \textit{rr\_writel()} function to access the SysCon module and activate the LM32 reset line. Processor has to stay in a reset state during the firmware loading, otherwise it could corrupt data. The next step is to copy the firmware file word by word to the DPRAM. After writing whole data the loader uses \textit{rr\_readl()} function to read the DPRAM content and verify if it was properly saved. When both writing and verification process returned with success, the initialization is done, LM32 reset signal is deactivated and the processor starts operation by fetching new instructions from RAM.
6.3 Running and testing the design

6.3.1 Reference hardware platform - SPEC Board

The reference hardware platform for White Rabbit PTP Core development was SPEC (Simple PCIe FMC Carrier) board presented in figure 45. It is a PCI-Express x4 card with a Xilinx Spartan 6 \( \text{XC6SLX}45T \) FPGA. SPEC can be placed in a computer’s PCIe slot or used as a standalone development board by connecting an external power supply. In both cases the Xilinx programmer may be used with provided JTAG port. The 4-lane PCI-Express bus is handled by the Gennum \( \text{GN}4124 \) bridge that converts it to, more convenient, local bus. Spartan 6 chip can be programmed throughout the JTAG header, PCIe or an SPI 32Mbit flash PROM. That last possibility is especially useful when the firmware is stable and a reference board becomes a commercial product. Thus, after powering it up FPGA is automatically programmed without any user action.

The board also incorporates an SFP cage with two LEDs connected to FPGA’s GPIO lines. They are used to indicate the link status. Moreover, it also has Voltage Controlled Oscillators (VCXOs) that are required to produce a reference and helper frequency for DDMDTD phase detection. The mini-USB connector together with USB-UART bridge is especially convenient for outputting debug messages from
the design. The FPGA implementation of a regular UART is relatively simple and because of the bridge, it can be connected to any modern computer. For a basic user interaction two microswitch buttons were mounted.

Beside the components essential for the WR PTP Core, SPEC contains also others that could be useful for other projects or further development of the WRPC. It can carry the FPGA Mezzanine Cards (FMC) so the functionality of the board can be extended with any FMC-compatible module. There are also two SATA connectors and a DDR3 memory (2Gbit).

6.3.2 Building the project

To build a complete WRPC project one has to clone three git repositories to a local filesystem. First of all, the HDL sources have to be fetched and synthesized to build the FPGA firmware. At the time of writing this thesis, current version is stored under the `wrcore_v2` branch of `wr_cores` git repository. However, it will be soon moved to the mainline. To synthesize VHDL/Verilog sources the `hdlmake` tool is used together with set of `Manifest files` and `Xilinx ISE Design Suite`. It is a very convenient way of maintaining and building HDL projects. Therefore, without going much into details, the following commands have to be executed in the Linux shell to build a complete WRPC FPGA binary configuration file(`spec_top.bin`):

```
> git clone git://ohwr.org/hdl-core-lib/wr-cores.git  # clone the git repo
> git checkout wrcore_v2                               # change local branch to wrcore_v2
> cd wr-cores                                           # enter the project's directory
> cd syn/spec_1.1/wr_core_demo                         # create Makefiles based on
> hdlmake                                               # Manifest files
> make                                                  # synthesize the project
```

To compile the Lattice Micro 32 firmware the clone of the `github` repository with the `C` source code of the `wrapper` and `device drivers` is required. A separate git repository holds the `WRPTP` daemon for both WR Switch and WR PTP Core. To compile the firmware the LM32 toolchain is required. The compilation effort is reduced to minimum by using the `make` utility. The only demand is to clone the `WRPTP` sources to the sub-directory within the `wrapper`. Additionally, the selection can be made whether the firmware should be built for Master or Slave White Rabbit PTP Core. To get the compiled binary file (`wrc.bin`) the following commands have to be executed:

\[\text{Modern PC computers (especially laptops) unfortunately are not equipped with RS-232 ports, but they have numerous USB ports.}\]

\[\text{LM32 toolchain can be downloaded from } \text {http://www.das-labor.org/files/madex/lm32_linux_i386.tar.bz2}\]
The FPGA bitstream can be uploaded to the device with the PCIe loader (available with rawrabbit kernel module) or by using a Xilinx JTAG programmer. On the other hand, to initialize the DPRAM with the wrc.bin file, the lm32-loader (described in sec. 6.2.2) is used. Finally, after connecting the SPEC board to a complementary White Rabbit device (another SPEC or WR Switch) the White Rabbit PTP Core Sync Monitor reports the actual synchronization status via UART (fig. 46).

### 6.3.3 Synchronization performance

In order to evaluate the accuracy and precision of synchronization provided by the White Rabbit PTP Core, a test system depicted in figure 47 was built. It consisted of a White Rabbit Switch configured as WR Master and White Rabbit PTP Core as WR Slave. They were connected together with a 5 km of bare, single-mode G.652 fiber link on a plastic roll. Both communicating parties used bidirectional SFP transceivers using Wavelength Division Multiplexing operating at 1310/1550 nm. The WRPC was running on its reference hardware platform (SPEC board) implemented in Xilinx Spartan 6 (XC6SLX45T) and using a Xilinx GTP transceiver as a physical network interface.
Varying link delay was enforced by repeatedly heating up the fiber roll with a hot-air gun and cooling it down by placing outside the window. The temperature of the fiber was continuously monitored by a digital temperature sensor connected to the SPEC board. The relation between the fiber temperature and the round-trip delay is shown in figure 48. The measurement lasted for 2.5 hours and the link was tested at the temperatures from +12.5 °C to 85 °C. The resulting temperature-induced delay drift, measured by the WRPC, was of 17.5 ns.

The stability of synchronization was characterized using a high speed digital oscilloscope (LeCroy WavePro 7300A), which was measuring the offset between the 125 MHz reference clock at the master side, and the recovered 125 MHz clock on the slave side. Results are presented in figure 49. Measured drift was below 100 ps over the entire temperature range (< 1% of the change in the round-trip delay), while the short-term (< 1 minute) rms master-to-slave jitter was smaller than 11 ps. The noticeable correlation between the offset value (red trace) and the temperature (green trace) was caused by the instability of the transmit wavelengths of the SFP transceivers and can be further improved by stabilizing the temperature of the lasers. The 3.3 ns constant bias results from unmatched cable lengths used to connect the SPEC card and the WR Switch to the oscilloscope (the SPEC does not have a dedicated 125 MHz output, therefore an external LVDS to LVTTL buffer was soldered to the circuit board, introducing a constant 3.3 ns offset).

Finally, presented measurement system was also used to characterize the ac-
Figure 48: Temperature-induced changes in the round-trip link delay

Figure 49: Master-slave 125 MHz clock offset stability vs fiber temperature

accuracy of the synchronization. This was done by monitoring the offset between the 1-PPS outputs of the master and the slave node. The resulting PPS offset histogram is presented in figure 50. Note that the jitter is much higher compared to the 125 MHz reference clock offset measurement, since the PPS outputs were driven directly by the FPGA output pins. The offset between the clocks is though far below 1 ns (which was the assumption of the project).
Based on presented measurements White Rabbit PTP Core meets the project assumptions. It achieves synchronization accuracies far below 1 ns over a 5 km regular 1000Base-X Ethernet fiber link. It also correctly measures and actively compensates the link delay varying with environmental conditions.
7 Applications

White Rabbit PTP Core applications can be discussed in a twofold way. It can be considered as a standalone module providing precise time synchronization to both existing and arising devices or as a part of White Rabbit Node. The latter is WRPC surrounded with modules and mechanisms for a robust and deterministic high-priority packets delivery [9]. Its main application is timing and control systems for accelerator facilities e.g. CERN and GSI. This application is however obvious since it actually raised the necessity of developing a White Rabbit project. Therefore this section presents other, both scientific and commercial timing applications that could benefit from the WR PTP Core and WR Network.

White Rabbit PTP Core can be used in distributed data acquisition systems (sec.7.1) or in radioastronomy to synchronize the arrays of telescopes, it could also synchronize metrology institutes to create national timescales. It could be the alternative to experiments with time and frequency transfer over a fiber link done by Polish Central Office Of Measures (GUM), Polish Telecom and AGH University of Science and Technology [19].

7.1 Distributed oscilloscope

The example of a distributed data acquisition is the OASIS system at CERN. It acts as a huge oscilloscope measuring thousands of signals coming from sources distant by several kilometers (figure 51). WRPC can be used to accurately time tag the blocks of samples at the ADC cards and produce nanosecond-accurate time tags for the external trigger signals. Having the sample blocks with associated time tags, one can reconstruct the original time relations between the signals and the triggers in software and present the measurements to the operator as if it was displayed on a typical oscilloscope.

7.2 Commercial applications

The WRPC module and the WR protocol could also be used in commercial systems and applications. Radars, terrestrial precise approach instrumentation in airports, and next generation mobile telecommunication networks are the flagship examples of systems requiring very precise timing. It could also provide a very accurate synchronization for transmitters for indoor and outdoor localization based on a difference of signals’ arrival time. In general, WR PTP Core can be used everywhere in the industry where the accuracy of existing protocols (NTP, PTP) is not sufficient, while
still maintaining compatibility with numerous already deployed devices supporting IEEE-1588.

7.2.1 Mobile broadband networks

Currently we are at a time when telecom providers try to fulfill the growing need for high bandwidth and reduce the maintenance costs at the same time. The Long Term Evolution (LTE) and LTE-Advanced\textsuperscript{11} define an all-IP network architecture where all traffic (both data and voice) is transmitted using the IP packets. That is why synchronization standards used before (SONET/SDH) can no longer be used. IEEE1588-2008 was proposed as an alternative. However, the LTE-Advanced in a few years would bring up new services and solutions further increasing the amount of transmitted data and requiring even more precise timing than IEEE1588-2008 could provide. Two examples of such upcoming technologies are Multi-Cell MIMO and Multi Broadcast Multimedia Services (MBMS). The former would significantly increase the throughput especially at the cells boundary. Mobile device would have multiple antennas (eight is proposed) to be able to receive data simultaneously from multiple base stations. The other technology (MBMS) is being developed for multimedia streaming (mobile TV). This would also rely on simultaneously receiving data from multiple base stations. In both situations the devices forming the LTE-Advanced network would have to be very precisely synchronized. This could be fulfilled by using the WRPC module inside each base station and connecting them all together with WR Switches to create a White Rabbit Network. By using

\textsuperscript{11}The implementations of LTE are already being developed and tested by the telecommunication companies while LTE-Advanced is the successor of LTE.
Sync-E and phase detection techniques, White Rabbit transmits frequency, precise time and compensates the link imperfections, while coexisting with regular data and introducing negligible amount of additional traffic. The huge advantage of WR in this application is its backwards compatibility with IEEE-1588. That in turn means, PTP-compatible devices deployed for handling LTE could be also synchronized from the WR Network and become gradually upgraded in the future to new LTE-Advanced equipment.
8 Summary

White Rabbit PTP Core is a unique and complete implementation of the sub-nanosecond time synchronization stack. It is based on a regular Gigabit Ethernet, but combines IEEE-1588, Synchronous Ethernet, and phase measurements to achieve its high performance over kilometers long fiber links. It is an HDL module that can be integrated into the White Rabbit Node, but can also be used as a complete, standalone synchronization core. It can be incorporated inside both existing and arising scientific or commercial equipment. The module also has a functionality of a regular Ethernet MAC, therefore is able to send regular non-timing data.

The internal design of WRPC was optimized to achieve required performance and utilize as little FPGA resources as possible. Presented measurements prove that White Rabbit PTP Core provides a sub-nanosecond synchronization accuracy and is able to precisely measure and actively compensate varying optical link characteristics.

First successful demonstration of partially operational WRPC was presented to the White Rabbit community during the 4th White Rabbit Workshop in GSI, Darmstadt, Germany (14 April 2011).

The future work on White Rabbit PTP Core consists of placing a module inside a commercial network time server [20]. This way, it will become available for the commercial systems and customers. There is also an ongoing research, with first promising results to make the WRPC achieve high accuracy synchronization also on copper-based-Ethernet (1000Base-TX). Moreover, the White Rabbit Network consisting of WR Nodes and WR Switches will be deployed to provide a sub-nanosecond synchronization for CERN - Gran Sasso Opera experiment [21]. It will create an alternate method of verifying the breakthrough discovery of neutrinos traveling faster than light.

White Rabbit PTP Core and the entire White Rabbit project is special in two aspects: synchronization performance and openness. The design and implementation of WRPC was done using open standards (e.g. Wishbone) and open ip-cores (Lattice Micro 32). The software firmware was also based on the open-sourced PTP daemon. Although the reference hardware platform was based on Xilinx FPGA, the HDL code itself is generic and not tied to any particular manufacturer. It can be synthesized to any other FPGA device, provided that it is large enough to fit the resulting firmware. The complete source code for both HDL and software is freely and publicly available. It is licensed under LGPL and CERN Open Hardware License. This way the project can be further used by researchers and commercial companies.
References


[13] Iztok Jeras. Minimalistic 1-wire (onewire) master with Avalon MM bus interface ([http://opencores.org/project,sockit_owm](http://opencores.org/project,sockit_owm)).


[21] *Oscillation Project with Emulsion-tRacking Apparatus* (http://operaweb.lngs.infn.it/).
Appendices

A.1 Source code

The White Rabbit PTP Core has been developed as a part of White Rabbit project started and maintained by the European Organization for Nuclear Research (CERN). The whole source code of the White Rabbit PTP Core is open source and publicly available on the Open Hardware Repository (http://www.ohwr.org).

- White Rabbit PTP Core project site: http://www.ohwr.org/projects/wr-cores
- VHDL/Verilog source code: git://ohwr.org/hdl-core-lib/wr-cores.git
- LM32 firmware C source code: git://github.com/twlostow/wr-core-software.git
- WRPTP daemon source core: git://gnudd.com/ptp-noposix.git