Felix from Interlaken

Leonie Verwoert

28-06-2019

1

Education: Engineering, Elektrotechniek
Name: Leonie Verwoert
StudentID: 500709027
Supervisors: F.Schreuder (Nikhef) & N.Misconi (Hogeschool van Amsterdam)

1. *The front-page image depicts the city of Interlaken located in Switzerland. The name Interlaken originates from inter = "in between" and laken = "lakes" which originates from the fact that the city is located in between two lakes, namely lake Brienz and lake Thun. (image source: [1])*
The title also has a double meaning, "Felix from Interlaken" could refer to a Swiss person named Felix born in Interlaken, But in the context of this project it refers to the project assignment, where a new implementation was developed which implemented the Interlaken protocol on the FELIX system (explained in more detail in this document.)

Summary

Within this project, research has been done on how the Interlaken protocol can be implemented on the FELIX system. The FELIX system is the data acquisition system from the ATLAS detector, one of the experiments held at Cern [3]. A new version of the FELIX system is currently in development. In the next iteration of the project, a suitable replacement for the currently used GBT protocol [5] is needed. A possible candidate to replace this protocol is the Interlaken protocol [1]. From previous research, a proof of concept implementation was realized. This implementation still needed to be verified however, to make sure it was according to the protocol definition. This verification was done, and a new version of the Interlaken implementation was made. This new implementation was then implemented on the FELIX systems hardware. After this implementation was tested, it was concluded that the implementation was successful. This means that the Interlaken is proven to be a good candidate for the next iteration of the FELIX project. As a next step, a design was made that extended the number of transmission channels from one to four channels. In a future version of this project, it is recommended that this design is implemented. This will further optimize the implementation and raise the possible transmission speed. Another optimization method that was researched is flow control, this is also a suitable addition to be made in future versions of the design.

Samenvatting

In dit project is er onderzoek gedaan naar hoe het Interlaken protocol geïmplementeerd kan worden op het FELIX systeem, Het FELIX project gaat over het data acquisitie systeem van de detector van het ATLAS experiment in CERN [3]. Een nieuwe versie van het FELIX systeem is momenteel in ontwikkeling. In de nieuwe versie van het FELIX systeem is er een vervanging van het (momenteel gebruikte) GBT protocol [5] nodig. In eerder uitgevoerd onderzoek, is naar voren gekomen dat het Interlaken protocol [1] een goede kandidaat zou kunnen zijn om als vervanging voor GBT te gebruiken. Hieruit is ook een concept implementatie voortgekomen [6]. Deze implementatie moest echter nog wel geverifieerd worden, om zeker te weten dat het zich gedroeg volgens de opgestelde protocol definitie van Interlaken. Deze verificatie is uitgevoerd, en hieruit is een nieuwe versie van de Interlaken implementatie ontstaan. Deze implementatie is vervolgens geïmplementeerd op de FELIX hardware. Nadat deze implementatie uitvoerig was getest, was er geconcludeerd dat de Interlaken implementatie succesvol gerealiseerd was op het FELIX systeem. Dit bevestigd dat het Interlaken protocol een goede kandidaat is om te gebruiken tijdens de ontwikkeling van het nieuwe FELIX systeem. Als een volgende stap, is er een ontwerp gemaakt dat het aantal transmissiekanalen van een naar vier uitbreidde. In een volgende versie van dit project, kan dit ontwerp worden geïmplementeerd om de Interlaken implementatie verder te optimaliseren. Een andere mogelijke uitbreiding van het Interlaken protocol dat onderzocht is, is de toevoeging van *Flow control* wat de mogelijkheid toevoegt om de transmissielijnen te reguleren. Dit is ook een goed uitbreiding die gedaan kan worden in volgende versies van dit project.

Contents

List of Figures

List of Tables

Revision History

Revision	Date	Author(s)	Description
0.1	02-04-2019	Leonie Verwoert	First version
0.2	15-05-2019	Leonie Verwoert	Rough draft
0.3	27-05-2019	Leonie Verwoert	Added extra chapters
0.4	11-06-2019	Leonie Verwoert	Added introduction
0.5	25-06-2019	Leonie Verwoert	Changed chapter layout
1.0	28-06-2019	Leonie Verwoert	First finished version

1 Introduction

This document describes the research done for a graduating assignment at Nikhef in Amsterdam. Nikhef is the national institute for subatomic physics, where research is done on the subatomic particles in our universe, their underlying power and the structures of space and time. A big part of the research done at Nikhef is linked to the larger experiments held at Cern [2] and the department *Electronics Technology (ET)* at Nikhef develops electronics and firmware for these experiments. One of the experiments at Cern is the ATLAS experiment [3] where a large detector is used to research subatomic particles (see figure 1).

Figure 1: Arial view of Cern, and the location of the ATLAS detector. [2]

This experiment generates a huge amount of data, and this data needs to be collected and processed. From this data acquisition problem, the Atlas *Front-End LInk eXchange (FELIX)* project [4] originated. In the FELIX project a generic solution is being developed for data acquisition within the Atlas detector. FELIX is a server PC where the focus is on high speed data transfer. A FELIX (FLX) card (abbreviated to FLX to distinguish it from the host PC name) with a PCI-express interface (a parallel communication bus) and several uniquely developed communication protocols, communicates with the host PC. Applications on the PC handle and forward the data over the network.

The first version of FELIX is currently being expanded and in the process of being taken into use. FELIX (the server PC) has a PCI-express card and a number of optic links. These optic links communicate at high speeds (4.8Gbps) with the electronic components in the Atlas detector. The data that follows from this communication is sent to the PCI-express card and sent to the memory in the PC using DMA with a speed of 100Gbps. From the PC the data will be sent further, over a generic high performance computer network.

In the next iteration of FELIX, a more advanced, more efficient and faster protocol than the currently implemented protocol (GBT [5], 4.8Gb/s) is needed. From previous research [6] it was concluded that the Interlaken protocol might be a good candidate for this next iteration of FELIX. Before this protocol can be implemented on FELIX, the current proof of concept implementation (Core1990, made during previous research) needs to be verified to make sure it functions according to the Interlaken protocol definition. This verification will result in a new version of the implementation (Core1990_V2.0) that is verified as the Interlaken protocol and can be implemented on the FELIX hardware. In order to ensure a successful implementation, research must be done on how to make the Interlaken implementation compatible with the firmware on the FLX-card. This research can be summarized as the answer to the following research question: *How can an implementation of the Interlaken protocol be realized on FELIX?* In order to answer this research question, the current state of the FELIX hardware and firmware must be analyzed, to find out what is needed to implement the new (Core1990_V2.0) Interlaken implementation on the FLX-card.

1.1 Thesis outline

This thesis consists of theoretical and practical research. The theory behind the project will be explained first, and afterwards the theoretical knowledge is used to develop the needed implementation(s). The ATLAS and FELIX projects will be briefly explained in Chapter 2 and 3 where it will be explained that the PPP (point-to-point) protocol used in the current implementation of FELIX, can not keep up with the rising transfer speed requirements. Because of this, the possibility of implementing a different protocol on a FLX-card will be researched. The Interlaken protocol [1] has proved to be able to handle the (in future) necessary transfer rates this research is described in Chapter 5. Because this protocol requires a lot of preliminary knowledge, it's functionality is described in detail in Chapter 4. Once a clear understanding of the Interlaken protocol has been established, the current state of the FELIX firmware will be researched and the *Wupper* PCI-express wrapper will be explained in detail in Chapter 6. Wupper will be needed to transfer the data collected by Interlaken into memory of the FELIX host-PC. Once Wupper and Interlaken are both researched, a (firmware) application can be developed to establish a connection between Wupper and Interlaken. Once this implementation is completed, further research can be done on possible optimization methods for the implementation. This is described in Chapter 7.

The implementation of Interlaken developed in [6], might not meet the requirements set in the Interlaken protocol definition as shown in [1]. This results in an implementation of an "unknown" protocol, very similar to Interlaken, but not usable as such. In order to implement Interlaken on FELIX, the implementation has to be verified and tested in combination with a commercial implementation of Interlaken, to make sure it functions according to the definition. This results in an updated implementation which is described in Chapter 10.1. Once this implementation is completed and verified, the application firmware which connects Wupper to Interlaken can be developed. This is described in Chapter 10.2. Once this application is finished, the possible optimization methods researched in Chapter 7 can added to the Interlaken implementation (Core1990_V3.0.) this process is described in Chapter 10.3.

After the testing is completed the results will be discussed in Chapter 11, and Chapter 12 will present the conclusions made from this research project. Finally, some recomendations for future work will be made in Chapter 13.

2 ATLAS

ATLAS (A Toroidal LHC ApparatuS) is one of the two general purpose detectors at the Large Hadron Collider (LHC see [7]) at Cern. It's experiments include the search for the Higgs boson and the research on the particles that make up dark matter. Beams of particles from the LHC collide at the centre of the ATLAS detector creating collision debris in the form of new particles which fly out in every direction. Around the collision point, six different detection subsystems arranged in layers record the paths, momentum and energy of the particles, allowing immediate identification of the particles. A giant magnet system bends the paths of the charged particles so their properties can be analyzed. These interactions in the ATLAS detectors create an enormous flow of data. Complex data acquisition and computing systems are then used to analyze the recorded collision events. [8]

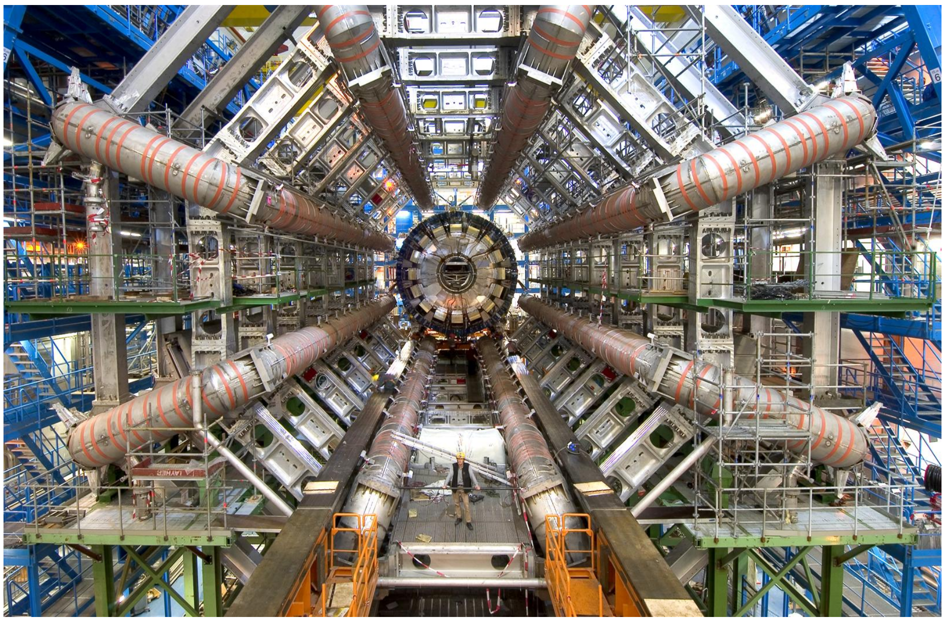


Figure 2: The ATLAS detector [9]

3 FELIX

One of the projects on data acquisition systems is the FELIX (FrontEnd LInk eXchange) project. It provides a detector-independent readout architecture which provides access between the detector data acquisition systems, the trigger electronics, and a connected network where the data is sent to the designated endpoints (see Figure 3.)

On one side of the Felix system, the Gigabit Transceiver (GBT) architecture along with the GBT protocol developed by Cern [5] provides a high speed (4.8Gbps) radiation-hard optic link for data transmissions from the detector electronics. The GBT protocol sets up several data links (so called e-links) which are time multiplexed over the same wire.

Via a Field Programmable Gate Array (FPGA) on the PCIe card (FLX-card) the data is transferred to the host PC memory. From there the data can be transferred further over the network.

The connectivity of the FELIX system is illustrated in Figure 3 The so called "Commercial off the shelf" (COTS) network switches are used to send (detector and fromd-end) control data and event readout data to and from the system. The *Timing, Trigger and Control system (TTC)* sends it's information to FELIX and this will be forwarded to the front-end electronics via the optical links.

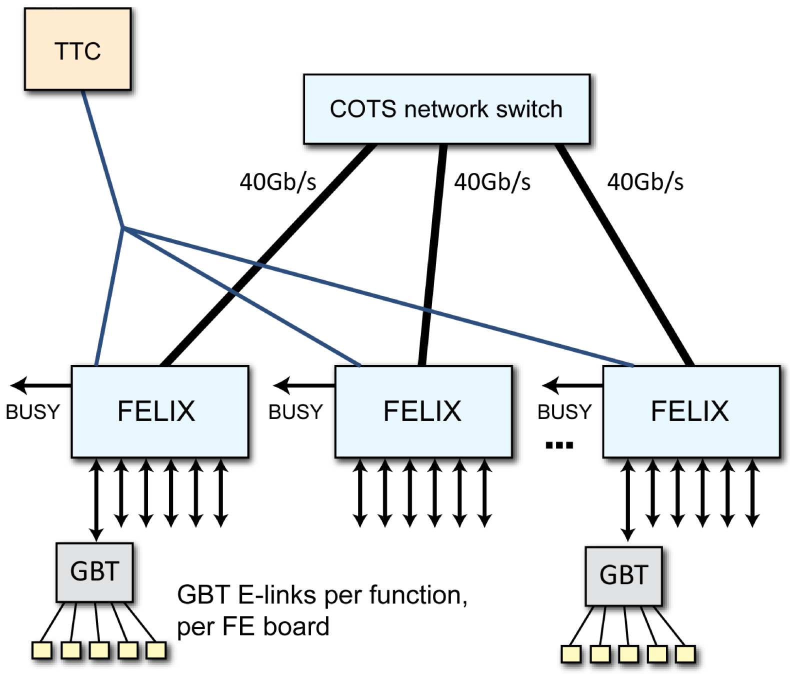


Figure 3: Overview of the connectivity of the FELIX system. [10]

The FELIX system itself (see Figure 4) is implemented with server PCs where each host PC has at least one FLX-card (a type of PCI-express card.) The FLX-card contains an FPGA, on which an interface is created, for the links connected to detector and trigger electronics. In the current technology these links use the GBT protocol [5] with a throughput of 4.8Gbps, and about 16-20 of these links are used per FPGA/card.

A PCIe engine (called Wupper) is implemented on the FPGA which provides the usage of Direct Memory Access (DMA) to access the memory of the Host PC. The FELIX (software) application running on the host PC, manages the data-transfers and the network connections. The data received from the FPGA is routed to one or more network endpoints. Data routing and the connection to the network switch is implemented with a software pipeline running on the FELIX host PC. [11]

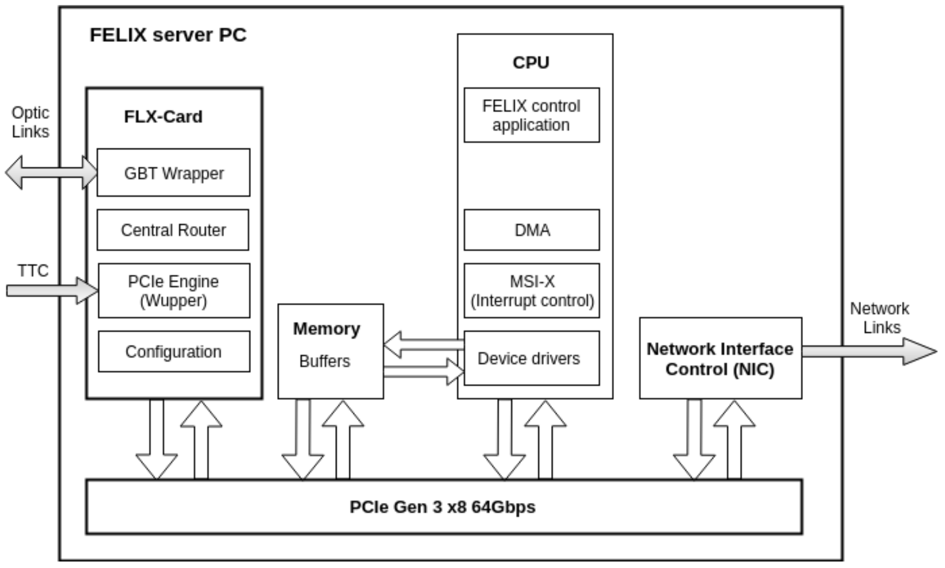


Figure 4: Block diagram of the FELIX system.

The FLX-card hardware (FLX-709) is shown in Figure 5. It consists of a Xilinx VC-709 board with a Virtex-7 X690T FPGA, a PCIe Gen3 card and 4 SFP+ (optic link) connectors [12]. This FLX-card is the same hardware that will be used during for implementation and tests within this project.

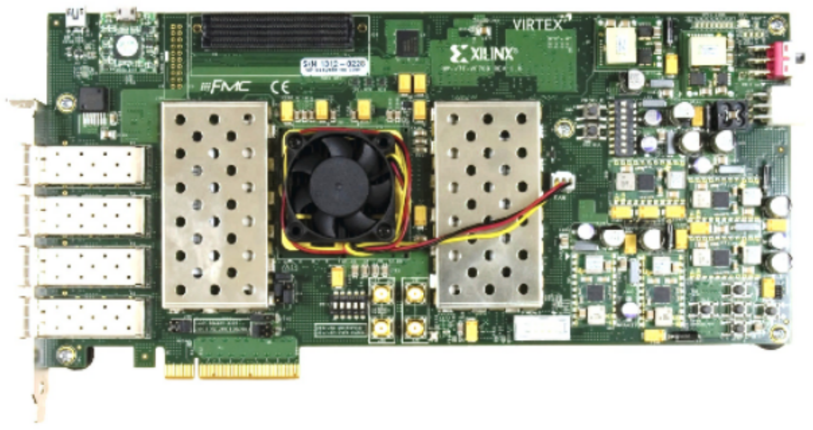


Figure 5: FELIX FLX-709 hardware. [12]

4 Interlaken

The main goal of this project, is to implement the Interlaken protocol on FELIX hardware. This is because previous research [6] has shown that Interlaken can be a suitable candidate to use on the FELIX system, replacing the GBT protocol.

Interlaken is a point-to-point protocol, that enables the design of a narrow, high-speed packet interface capable of parallel data transfers. It uses a control word structure to schedule packets and transmit diagnostics. The fundamental structures that define the Interlaken protocol are the data transmission format and the meta frame. The data is transmitted in so called *bursts* where the payload data can be divided along several bursts. Preceding and following a data burst, a control word is placed. These control words have multiple uses, such as the signaling of the start and end of the packet and error checking. Each burst is also associated with a transmission channel, which can be a physical networking port or a logically connected stream of data. When multiple data channels are in use, the data and control words are sent across all channels sequentially in groups of 8 Bytes, beginning at lane 0, ending at lane n, and repeating this process for the next data block (see Figure 6) When only one lane is used, the data is sent in a serial matter. [1]

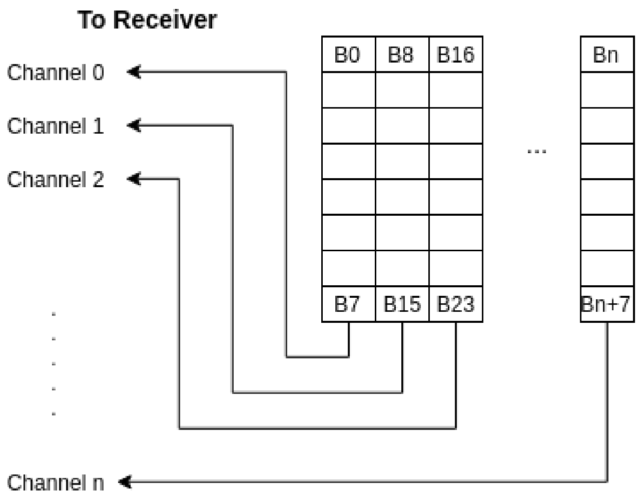


Figure 6: Interlaken data/control word transport over multiple data channels.

The data is transmitted in one or more bursts of configurable burst size. The meta frame format (which encapsulates the bursts) is defined to allow support on a parallel (SerDes) infrastructure, It includes four (unique to the meta frames) control words which can provide lane alignment, scrambler initialization, clock compensation and diagnostic functions.

The Interlaken protocol functions on the bottom two layers of the OSI model, the Physical and Data Link layer (see [13] for more information about the OSI model.) On these layers a complete Interlaken frame will be created. A functional block diagram of the Interlaken protocol is shown in Figure 7. The incoming data (top left) will be packed into bursts, and a CRC-24 check will be done. The output value of this check will be present in the control word following the burst. The framed bursts will be sent to the framing meta block, where the meta frames are created and a CRC-32 check is performed. This block adds 4 meta frame control words (which will be described in more detail later) and one of these control words will contain a CRC-32 output value. The meta frames will then be scrambled, except for two meta frame control word types that will never be scrambled

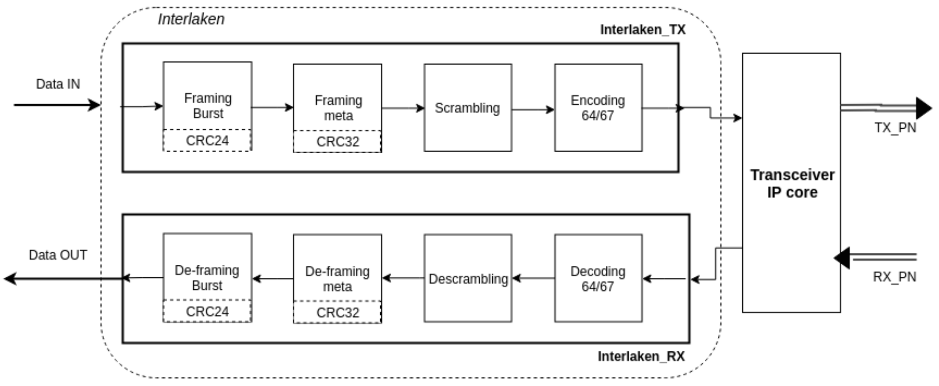


Figure 7: Interlaken block diagram.

(the synchronization meta control word and the meta control word describing the current scrambler state.) After the scrambling, the data is encoded and sent to the transceiver. The transceiver transforms the data to an optical data signal, which can be sent across the network. When this data is directly looped back to the transceiver RX-side the process is reversed, until only the original data remains at data_out.

Each part of the Interlaken protocol will be explained further in the next sections.

4.1 Word formats

The Interlaken protocol functions by alternating between data and control words. In the first framing process (Burst Framing in Figure 7) this can either be a Burst/Idle control word, or a data word. The main difference in formatting of the data word and Burst/Idle word are bits [65-64], as shown in Figure 8. The header of the data word is "01" and the header of control words is "10". Both have a preliminary inversion bit (bit 66.) Within the control words category, the formatting of these words differs. This also depends on which layer the word is found. In the burst layer, the Interlaken words can only be data words, Burst control words or Idle words. In the second layer, the meta-framing layer (see Figure 7) a new type of control words is introduced; the so called *meta frame* control words. These control words are shown side by side in Figure 9 for clarity.

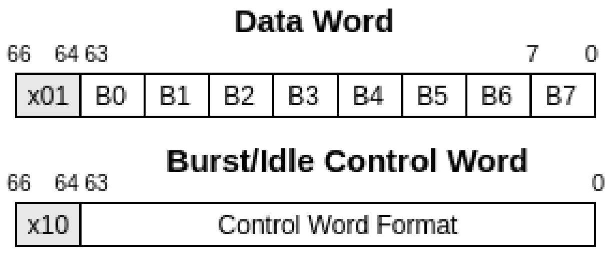


Figure 8: Burst/Idle control word format and Framing layer meta-frame control word format.

The header of these meta frame control words is the same as the Burst/Idle control words (bit [65-64] are "10" respectively) but bit [63] is set to '0' , signaling a meta framing layer control word, and to '1' for a Burst/Idle control word.

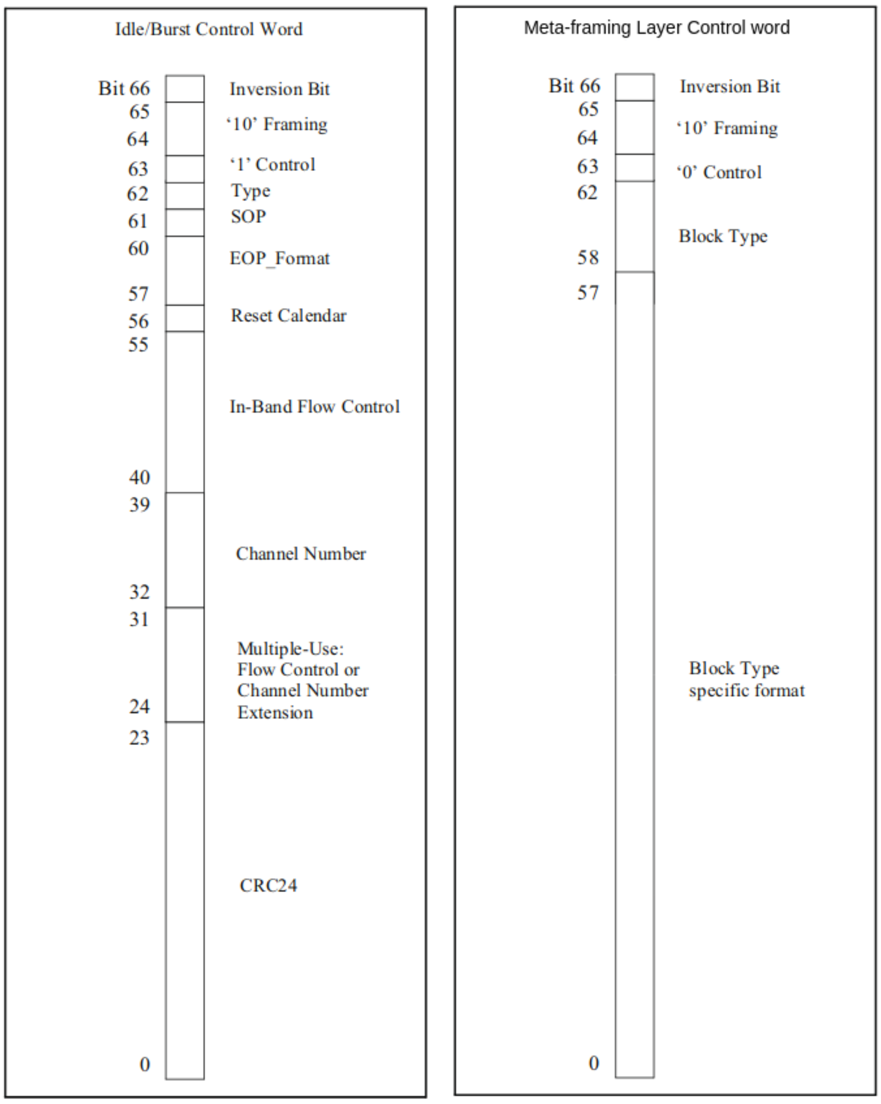


Figure 9: Burst/Idle control word format and Data Word format. [1]

4.1.1 Burst/Idle control word format

The different fields of the Burst control word (first layer) are explained in more detail in Table 1.

Table 1: Idle/Burst control word format [1]

Bits	**Description**	**Function**
66	Inversion	This bit can be inverted to limit the running disparity. '1' = Inverted, '0' = not inverted.
65:64	Framing	64/67b mechanism to distinguish between control and data words. "01' = data, "10" = control.
63	Control	Distinguish between Idle/Burst control words and Meta-frame control words. '1'= Idle/Burst and '0' = meta framing control word.
62	Type	If set to '1', the channel number and SOP fields are valid, and a data burst follows this control word. If set to '0', an Idle control word follows.
61	SOP	Start of packet. If set to '1', the data burst following this control word represents the start of a data packet. If set to '0', it is either a middle or end packet.
60:57	EOP_Format	This field to the data burst preceding this control word. as follows: '1xxx' - End-of-Packet, with bits[59:57] defining the number of valid bytes in the last 8-byte word in the burst. Bits[59:57] are encoded such that '000' means 8 bytes valid, '001' means 1 byte valid, etc., with '111' meaning 7 bytes valid; the valid bytes start with bit position [63:56] '0000' - no End-of-Packet, no ERR '0001' - Error and End-of-Packet
56	Reset Calendar [1]	If set to '1', the in band flow control status represents the beginning of the channel calendar.
55:40	In-Band Flow control [1]	The 1-bit flow control status for the 16 callendar entries. If set to '1'. the channel or channels represented by the calendar entry is XON, if set to '0', XOFF.
39:32	Channel Number [1]	The channel associated with the data burst following this control word; set to all zeroes for Idle Control Words
31:24	Multi-use [1]	This field may serve multiple purposes, depending on the application.
23:0	CRC24	A CRC error check that covers the previous data burst (if any) and this control word.

1: currently not implemented in the Core1990 implementation.

4.2 Burst structure

The transmitted data is sent over the interface in the form of one or more bursts. The bursts consist of data words enclosed in two or more control words. These control words are used to signal a start of packet, end of packet and several other functionalities. The Burst control word will then indicate a start of packet (SOP, bit 61) and data will follow. When the last data word has been sent, another control word will be sent containing the end of packet (EOP, bit 60-57) signal. When there is no new data available, Idle control words will be send (Type = '0', bit 62.) This is because the control information must still be sent to the receiver, even with no data. The size of the data bursts can be defined with the following parameters:

- BurstMax: The maximum data burst size (multiples of 64 bytes.)
- BurstShort: The minimum data burst size (min of 32 bytes, with 8-byte increments.)

In typical operation, the interface will send a burst of data (of Burstmax length), followed by a control word. Bursts are transmitted on each specified channel, until the data packet is completely transferred, at which point a new transfer can begin. Because of the channelized interface, the data *end of packet* control word, may occur several times on several channels in a short amount of time. This puts a significant timing challenge on the transmitter and receiver (to handle all of the data processing between the control words.) Because of this, a minimum data burst size *BurstShort* is defined. When the burst-size is smaller than the defined minimum value, additional Idle words will be added (see Figure 10.) [1]

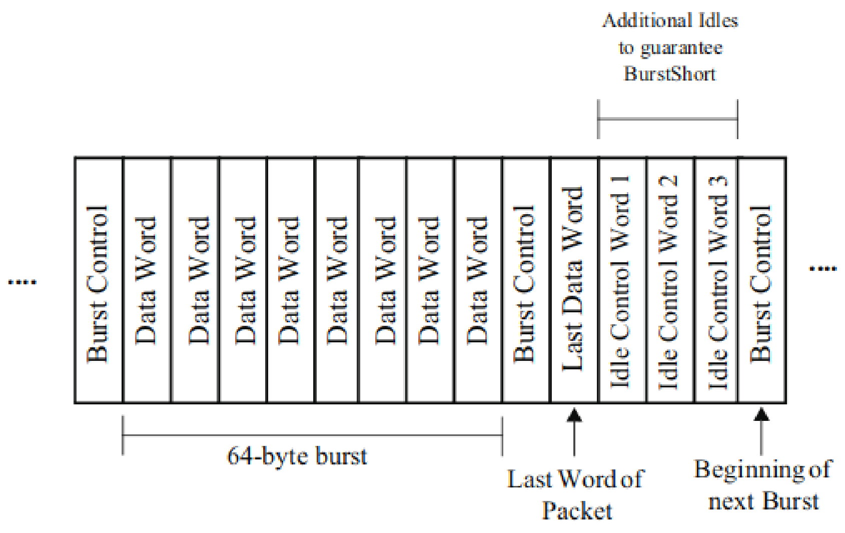


Figure 10: Filling of the bursts to guaranteed BurstShort size. [1]

4.3 Cyclic Redundancy Check (CRC24)

When data is sent across a network, there is always a possibility for errors to arise. The CRC check is used to detect these errors in a data packet. CRC is commonly used, as it is a powerful but easily implementable solution. The input of a CRC consists of a binary stream of data. This stream is divided by another binary number called the *polynomial*. The 'rest' (remaining value) of this division is called the *checksum* which is appended to the transmitted message. On the RX side, the CRC is also performed, and the same polynomial is used as on the transmitter side. The CRC checksum (from the receiver side) is already included in the received data. If the result of the division of these two checksums is zero, the transmission was successful and the data was not corrupted.

Within the *framing burst* block of Interlaken, a CRC-24 check is performed. The polynomial of the CRC-24 check is shown in Equation 1 [1].

$$X^{24} + X^{21} + X^{20} + X^{17} + X^{15} + X^{11} + X^{9} + X^{8} + X^{6} + X^{5} + X + 1 \tag{1}$$

Each coefficient represents a '1' in the 24-bits long polynomial. The value of the polynomial is calculated using Equation 1, resulting in a binary value that is transformed to a hexadecimal value for convenience (see Equation 2.) The resulting CRC-24 polynomial is *1328B63*.

$$\begin{array}{ccccccc} 2^{24} & 2^{23}2^{22}2^{21}2^{20} & 2^{19}2^{18}2^{17}2^{16} & 2^{15}2^{14}2^{13}2^{12} & 2^{11}2^{10}2^{9}2^{8} & 2^{7}2^{6}2^{5}2^{4} & 2^{3}2^{2}2^{1}2^{0} \\ 1 & 0011 & 0010 & 1000 & 1011 & 0110 & 0011 \\ 1 & 3 & 2 & 8 & B & 6 & 3 \end{array} \tag{2}$$

The EOP_Format field of the Burst Control Word (see Figure 8) identifies how many bytes of the last data word of the burst are valid. Bytes that are invalid are discarded by the receiver. Data and control word integrity is ensured by the CRC24. The CRC24 is calculated against all data in the burst, and all the fields in the control word. The data is sent into the CRC24 function most significant bit(MSB), in order of transmission. The CRC-checksum is generated by resetting the polynomial to all ones, then the data stream is sent to through the polynomial function and finally inverted and transmitted within the control word. [1]

4.4 Meta framing

The meta framing block introduces four new *meta frame control* word types. These control words each have a different format and functionality. Each lane has a set of 4 of these control words, which travel along with the payload data (burst data and control information.) The structure of the meta frame is shown in Figure 11. As mentioned before, the control words can be distinguisehd from data words by checking ths *framing* bits [65-64] (see Figure 8 and 8) A meta-frame control word can be distinguished from the burst/Idle control words with the *control* bit [63]. If this bit is set to '0' the current control word is a meta-frame control word, (if it is set to '1' it is a Burst/Idle control word.)

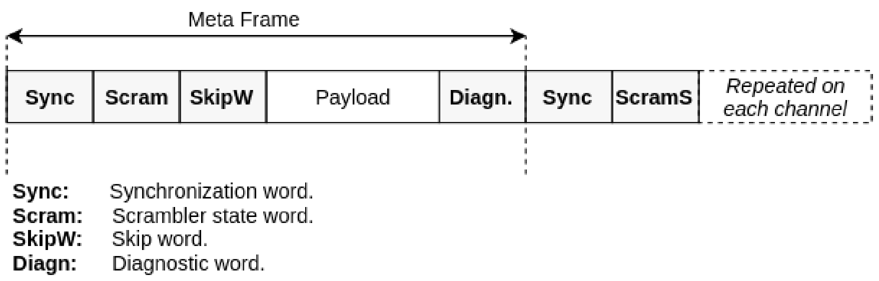


Figure 11: meta frame structure.

The four types of meta-frame control words are shown in Figure 11. The four meta-frame control words can be distinguished with the *Block type* bits [62-58] (see Figure 8) which are unique to the meta-frame control words. These bits determine which control word is sent, and this determined the format (bits [57-0]) of the control word. An overview of the different meta-frame control words and their matching block-type bits are shown in Table 2. These will now be explained in more detail.

Meta Frame Control Word	*Block Type (positive disparity)*
Synchronization	011110
Scrambler State	001010
Skip	000111
Diagnostic	011001

Table 2: Meta frame block types

4.4.1 Synchronization and Scrambler-state words

To avoid error multiplication, Interlaken uses an independent synchronous scrambler on each transmission channel of the interface. With every new meta frame, the scrambler state is sent to the receiver, to allow the receiver to de-scramble the data that follows. In order to allow for the correct decoding of the received data, the receiver must be synchronized with the state of the scrambler polynomial on the moment when the currently received data was scrambled. This synchronization takes place via a unique 64-bit Synchronization word, and the Scrambler-state word that are transmitted consecutively as part of the meta-frame. The format of the Synchronization word is shown in Figure 12 and the Scrambler State word format is shown in Figure 13.

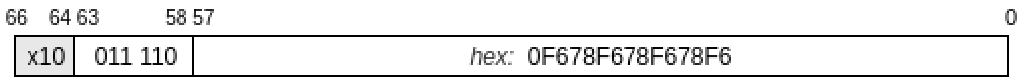


Figure 12: Synchronization word format.

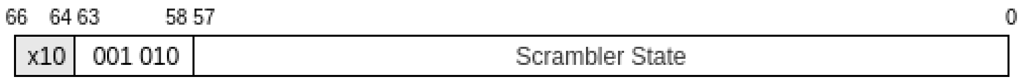
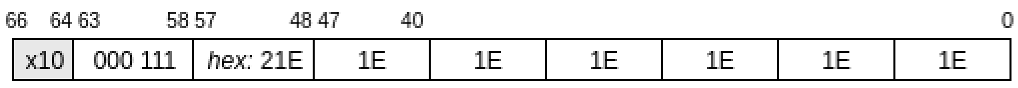


Figure 13: Scrambler State word format.

The state diagram of the scrambler Synchronization is shown in Appendix C. When in the reset state, each lane searches for the unique pattern of the Synchronization word. If the Synchronization word is found, the receiver counts until a meta-frame length amount of data has passed, and looks if another Synchronization word is found. When four consecutive Synchronization words are found, the scrambler will achieve lock, and advance it's state with each newly received data or control word (with the exception of the scrambler state and synchronization words.) Once synchronization is achieved,the interface uses the recovered (current) value of the scrambler polynomial from the scrambler-state meta-frame control word, to seed the de-scrambler and de-scramble the data. As mentioned before, the scrambler-state word and the synchronization (meta-frame control) word types are never scrambled. All of the other (data and control) words are scrambled from bits [63:0], the framing bits[66:64] are never scrambled, in order to still be able to differentiate the data words and control words.

4.4.2 Skip word(s)

The purpose of a skip word is to provide clock compensation in case a repeater is added between the TX and RX side. The format of a Skip Word is shown in Figure 14.



66	64 63	58 57	48 47	40				0
x10	000 111	*hex*: 21E	1E	1E	1E	1E	1E	1E

Figure 14: Skip word format.

If there is a slight clock difference on either side of the receiver, synchronization may not be achieved. To account for this several skip words can be added to the receiver or transmitter, depending on which sides clock is slower. A single skip word is a required part of the meta-frame format, but more Skip Words can be added at any point in the meta-frame, except between Diagnostic,Synchronization and Scrambler State words. It is required by the receiving end, to correctly identify the Skip Words and remove them from the data. An example situation is shown in figure 15.

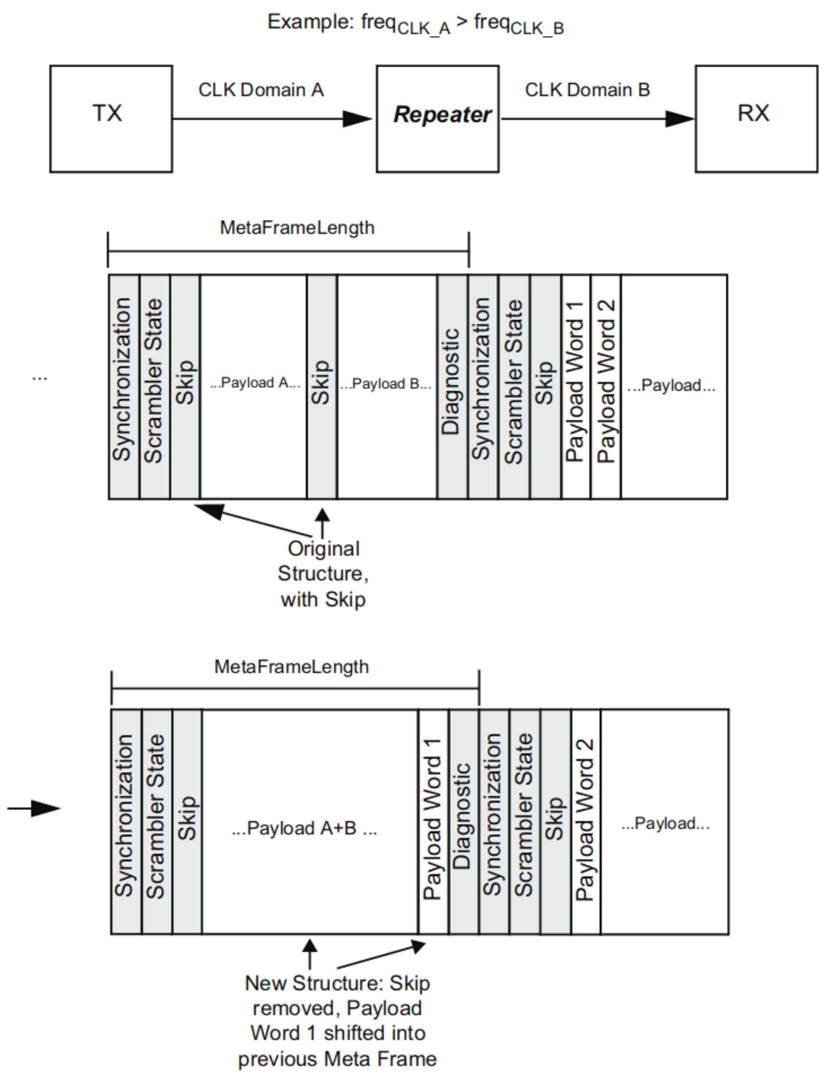


Figure 15: Example implementation of skip words for clock compensation. [1]

If there is a repeater between the original transmitter and final receiver, the repeater may compensate for a slower transit clock by silently discarding the skip word but it must keep the (predefined) *meta-frame-length*. This is achieved by adding payload data A and B together, and adding an extra payload data word from the next meta-frame to the current frame (see Figure 15.) An important thing to note is that by using this method, eventually the data payload of the next meta-frame will be zero, and the Synchronization, Scrambler State and Diagnostic words will have to be shifted into the prior meta-frame which will override their previous value. If there's no new meta-frame payload data available, the action in Figure 15 cannot be performed and the meta-frame length cannot be guaranteed. There is no mention of this use case in the protocol definition, instead its implementation details are left to the user. If the situation in Figure 15 is reversed, and CL_B is faster than CLK_A, the reverse approach is required, and an additional skip word is added, and the last payload data word is transferred into the next meta-frame. eventually, this process requires that a Diagnostic word is added, when this is the case, the status message should remain the same as the previous frame. [1]

4.4.3 Diagnostic word

The diagnostic word is another meta-frame control word type. It is identified by the *Block Type* value *011001* (see Figure 9 and the meta-frame layer control word format in Figure 8.)

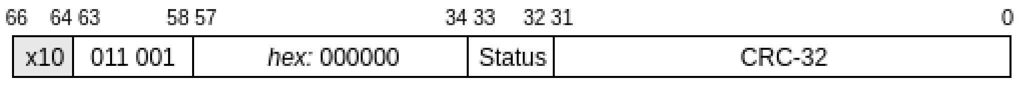


Figure 16: Diagnostic word format.

The diagnostic words have two important functions, a lane Status Message and a per-lane error detection (CRC32.) The Diagnostic word format is shown in Figure 16. The [33:32] bit status message provides a per-lane status message, where bit [33] represents the health of the current lane, and bit [32] represents the health of the entire interface. A '1' written to these bits represents a healthy condition, and a '0' indicates a problem. The CRC32 is provided as a diagnostic tool on a per-lane basis, each lane provides it's own error CRC32 check. The crc32 value is stored in bits [31:0] of the diagnostic word.

4.4.4 CRC32-calculation

The CRC32 is calculated over all of the words transmitted within the meta-frame (including the diagnostic word itself,) before scrambling and inversion. It is only calculated over bits [63:0], excluding the framing bits [66:64]. The fields over which the CRC32 is calculated are shown in Figure 17.

While the *Synchronization* and *Scrambler State* words are not included in the scrambling process, they are included in the CRC32 calculation. However, the 58-bit scrambler state of the *Scrambler-State* meta-frame control word, is treated as all zeros during the calculation of the CRC32 value. The CRC32 polynomial is defined as in Equation 3.

$$\begin{aligned} X^{32}+X^{28}+X^{27}+X^{26}+X^{25}+X^{23}+X^{22}+X^{20}+X^{19}+ \\ X^{18}+X^{14}+X^{13}+X^{11}+X^{10}+X^{9}+X^{8}+X^{6}+1 \end{aligned} \tag{3}$$

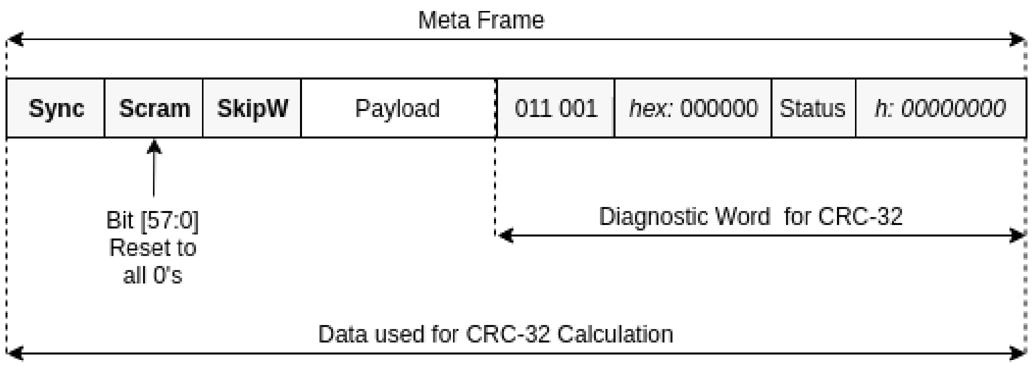


Figure 17: CRC32 calculation fields.

It's calculation is shown in Equation 4 and results in the hexadecimal value *11EDC6F41*.

$$\begin{matrix} 1 & 0001 & 1110 & 1101 & 1100 & 0110 & 1111 & 0100 & 0001 \\ 1 & 1 & E & D & C & 6 & F & 4 & 1 \end{matrix} \tag{4}$$

The CRC32 value calculated at the transmitter, is sent to the receiver in the *Diagnostic* (meta-frame) control word and this compared and checked with the CRC32 calculation done at the receiver side. When the sent data and received data does not match, an error will be raised.

4.5 Scrambler

The scrambler scrambles the incoming data bit by bit. Every bit is XOR-ed with the current scrambler state, which is calculated via the scrambler polynomial which is shown in Equation 5. The initial scrambler state (based on the polynomial) is activated after a reset, and the transmitter never resets it again. Instead, the current scrambler state advances after each incoming word, and the current scrambler state is added to the (meta-frame) control word and sent to the receiver.

$$X^{58} + X^{39} + 1 \tag{5}$$

The scrambler advances (resulting in a scrambler state change) while the interface is in operation, except for when Synchronization or Scrambler state words are received at its input. These words are never scrambled because in order to de-scramble the data on the receiver side, the receiver must be synchronized with the state of the scrambler polynomial of when the received data was scrambled. This is done with the *synchronization* and *Scrambler state* words, hence why they are not scrambled.

4.6 64/67 Encoding

An encoding and scrambling method is needed in serial interfaces, to assign word boundaries, allow for clock recovery, and to maintain DC balance. Interlaken uses 64/67 encoding, which is based off of the 64B/66B used for the IEEE 802.3ae 10 Gigabit Ethernet specification. The 64/66B encoding solves the word boundary problem by combining a scrambled 64-bit payload with two extra bits [65:66] to differentiate between data words ("01") and control words ("10".) One of the weaknesses of this implementation however, is that the DC baseline can start to "wander" after a large number of only data words (for example) are sent. Since this would mean a continuous "01" on bits [66:65] which could offset the DC baseline on the lane. To solve this problem, Interlaken introduces an extra bit [67] which acts as a disparity bit, which is either positive ('0') or negative ('1') depending on the number of ones and zeroes in the preceding words. Each lane maintains a count of the disparity (where a 0 decrements the count and a 1 increments the count.) Before transmission the disparity is calculated and if it is to high, the entire word [65:0] is inverted and a '1' is added to bit [67]. If the word remains un-inverted, bit [66] will become'1'. At the receiver side, if the *inversion bit* [66] is 0, the receiver process the word without modification. if it is 1 the receiver must un-invert the word before it is processed. [1]

4.7 Commercially available implementations

Since there is already an implementation of Interlaken (core1990) available that is vendor-independent and fully customisable, it would not make sense to switch to a commercially developed implementation. However, these commercially available ip-cores might be become useful if the implementation needs to be tested further, or to gain inspiration for possible future extensions. Several Interlaken ip-cores were found, two of which will now be briefly explained.

4.7.1 Xilinx Integrated Interlaken

The *Xilinx Intergrated Interlaken LogiCore IP* is a chip-to -chip interconnect protocol which allows for 1-12 consecutive lanes to be used to build an Interlaken Interface. It is designed to be compliant with the Interlaken protocol definition as described in [1]. The core instantiates an Interlaken integrated IP core, which is a single core in which all the individual Interlaken blocks (as mentioned in the previous paragraphs and shown in Figure 7) are embedded. The core can connect to serial transceivers at a rate of up to 25.78125 Gb/s on UltraScale+ devices. The integrated IP core handles all of the protocol related functions regarding communication with other connected devices. All of the handshaking, synchronizing and error checking is handled by the core itself. Packet data can be provided through the Local Bus (LBUS) TX interface and received via the LBUS RX interface. The LBUS is not mentioned in the Interlaken definition. It is used by the Xilinx core to allow parallel bus transfers between devices, by implementing multiple data channels. Flow control is also implemented on all of the data channels. The RX flow control information (send on bit 40-55 of the idle/burst control words) will be processed by a scheduler algorithm which will manage the data flow on the TX-side of the system. If advanced scheduling algorithms are needed, out-band flow control should be used, this is not implemented in the Xilinx core. [14]

4.7.2 Intel Interlaken IP Core

The Intel Interlaken ip-core is an Interlaken implementation developed by Intel, intended to be used with *Intel Stratix 10* devices [15] . It is compliant with the Interlaken protocol definition (see [1]) and supports 4,6 and 12 serial lanes to be used together to allow for up to 300 Gbps bandwidth. It supports per-lane data rates of 6.25, 10.3125, 12,5 and 25,3 Gbps to be used with their high speed transceivers. It also supports optional in-band or out-of-band flow-control.

5 Core1990

Previous research has been done, on point-to-point protocols, to see if a protocol could be found which met the predicted specifications needed in future projects. From this research the Interlaken protocol was concluded to be a suitable candidate to be used in future projects [6].

In order to proof that Interlaken could be implemented for the intended use case, an implementation was developed called Core1990. The proof of concept implementation was realized on a Xilinx VC707 [16] evaluation board. The implementation was successfully tested on a single board by connecting the TX and RX connections and creating a loop-back, and with two VC707 boards communication with each other. The implementation was then published as an open source project [17]. A complete overview of the architecture is shown in Figure 18 and shows all of the different components needed for Interlaken. The complete Core1990 design is described in [6].

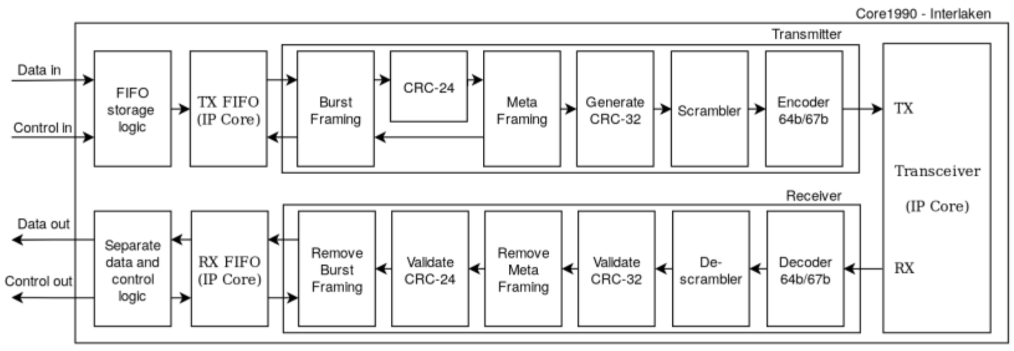


Figure 18: Complete core1990 architecture [6]

6 Wupper

Wupper is one of the framework components of the FELIX card (shown as *PCIe Engine* in the FELIX card block in Figure 19.) It provides a Direct Memory Access (DMA) interface for the Xilinx Virtex-7 PCIe Gen3 hardware present on FELIX. The data is streamed from the FPGA (present on the FELIX card) to the host PC memory, where the packet processing and routing is done. In order to sustain the high data rates, a high-bandwidth interface is required between the FELIX card and the host PC. As a solution to this problem, Wupper (a custom PCIe engine) was developed. In figure 19 a functional block diagram of Wupper is shown. [18]

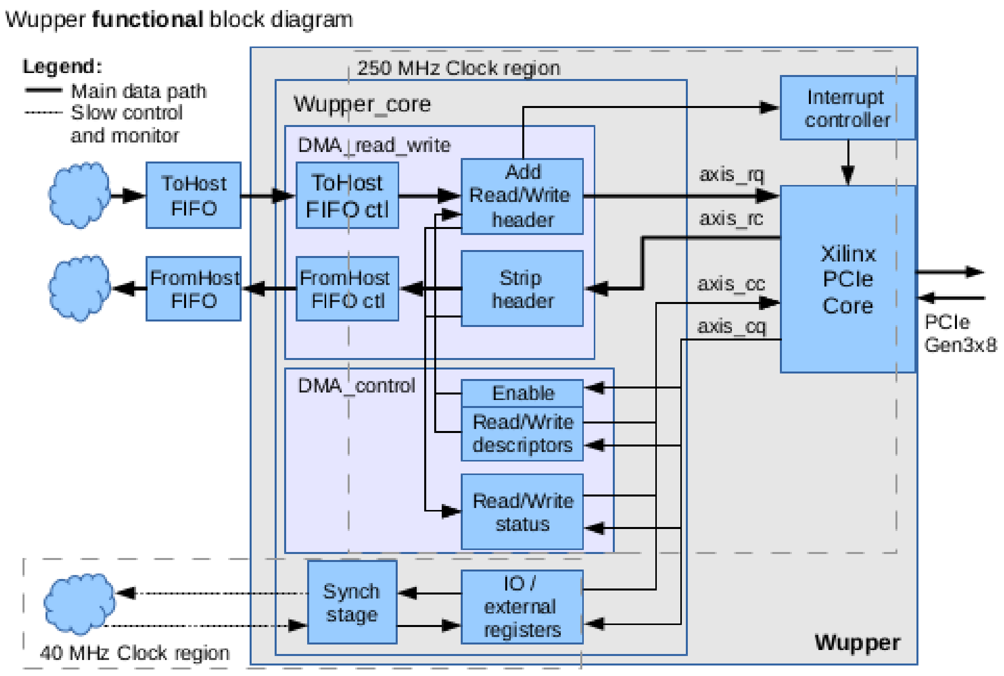


Figure 19: Functional block diagram of Wupper (PCIe engine.) [18]

The main functionality of Wupper is to handle data transfers from a user interface (such as a FIFO) to and from the host PC memory. The user application side of the FPGA design can read or write to the FIFO, and Wupper will handle the transfers to and from the memory on the host PC according to the address specified in the DMA descriptors. The Wupper core can also control and monitor the registers inside the FPGA and surrounding electronics via a register map. (The current Wupper register map is included in Appendix H.) The Wupper core communicates to the host PC via the *Wupper driver* and is controlled by a set of tools (so called *Wupper tools*.) [18]

The Wupper core is present on the FPGA in the FELIX board, and is developed in the language VHDL. It uses Direct Memory Access (DMA) and the PCI express (PCIe) lanes to move data bidirectionally to and from the host PC memory (see Figure 19). The *Xilinx PCIe core* [19] is used to transfer the data from DMA over the PCIe hardware. It uses the *AXI4-Stream Interface* [20] to communicate, which does not require address lines. The address (and other information) are inserted in the headers of the PCIe packages instead.

The use of DMA omits CPU intervention. The PC allocates a part of the available memory, and DMA is free to take control over it. The DMA control core, functions via descriptors. It will collect information such as the start and end address from these descriptors and write the status of the operation in a status register (via another descriptor.) Within the Wupper core, the DMA is divided into two parts: *DMA Control*, and *DMA Read Write* (see Figure 19).DMA Control contains a register map with addresses to the descriptors, status registers and external registers that can be read back through PCIe. DMA Read Write contains the processes which parse the DMA descriptors and transfer the data to and from the FIFO to the AXI stream bus. [18]

The main functionalities of the Wupper core will be explained further in the next paragraphs.

6.1 DMA control

The DMA control module provides a register map, which is divided into three areas: BAR0 (BAR stands for Base Address Region), BAR1, and BAR2. BAR0 contains the DMA related registers such as the DMA descriptors. These descriptors specify the addresses, direction of the transfer, data size and other settings. BAR1 is reserved for the interrupt related registers. The BAR2 region is dedicated to user applications. The DMA control module can manage all these regions, as well as respond to certain request types from the PC (only to IO and memory read/write request types.) The maximum payload data size is set 128 bits (which is the maximum payload that will fit in one 250MHZ clock cycle of the AXI4-Stream interface.) The registers can however be written in parts of 32, 64 or 128 bits at the time. [18]

6.2 DMA Read Write

The DMA Read Write module handles the data transfers to and from the FIFOs (according to the direction specified by the descriptors.) When data is moved into the *ToHost* FIFO (Read FIFO), a flag will be asserted which will start the DMA write process. This process will process the descriptors and create a header with the information obtained. The header is added to the data when it moves out of the *ToHost* FIFO. When the direction of data transfer is reversed, (*FromHost*) the data (with a header) is read from the Host PC memory. The header information in the header is parsed by the DMA control and the data is moved to the *FormHost* FIFO.

In summary, the DMA read write module contains two important processes, the adding and removing of the header to the PCIe packages. During the *add header* process, the header will be added to the PCIe packages, according to the information read from the DMA descriptor. If the descriptor is of the type *write* the payload data is added after the header. During the *remove header* process, the header of the received data is checked and the payload is moved into the FIFO. When the processing of the descriptor is completed, a interrupt (of type MSI-X) is fired through the interrupt controller. [18]

6.3 XilinX PCIe

The FPGA that is used on the Xilinx VC-709 board (used by FELIX,) has an integrated block for PCI Express Gen3 [19]. This block, handles the traffic over the PCIe bus. Within Wupper, the DMA read/write process can send and receive AXI4 commands over the AXI4-Stream bus, the PCIe block translates this into differential electric signals, which can travel over the PCIe bus.

6.4 AXI4-stream Interface

The communication between the Wupper core and Xilinx PCIe core consists of two seperate bidirectional AXI4 Stream interfaces. The two interfaces are the *requester interface*, which is used by the FPGA to issue the requests, and the PC replies, and the *completer interface* where the PC takes initiative [18]. A complete overview of the XI4 Streams and their functions is shown in Table 3.

Bus	Name	Usage	Direction
axis_rq	Reuester reQuest	Interface used for DMA. FPGA writes to this AXI4 Stream interface and the PC has to answer.	FPGA to PC
axis_rc	Requester Completer	Interface used for DMA reads, from the PC memory to the FPGA. A reply message from the PC is sent after a DMA write.	PC to FPGA
axis_cq	Completer reQuest	Interface used to write to the DMA descriptors (and other registers.)	PC to FPGA
axis_cc	Completer Completer	Interface used as reply interface for register read and as a reply header for register write.	FPGA to PC

Table 3: AXI4-Stream streams [18]

6.5 Data transfer modes

The standard data transfer mode is the so called *single shot* transfer mode. In a single shot transfer, the DMA *ToHost* process continuously sends data packets until the end address is reached, The PC can check the status of the DMA transaction by looking at the *desc_done* flag and *current_address*. Another possible operation mode is the so called *endless DMA*. In this transfer mode, the DMA continuously reads (or writes depending on the specified action in de descriptor) and when it has arrived at the end address (*end_address*), it wraps around and starts again from the start address (*start_address*). The *endless DMA* mode can be enabled by asserting the *wrap_around* bit. In this mode, another address is needed called *PC_read_pointer*. With this pointer, the PC indicates where it last read out the memory.

After DMA has "wrapped around" (started again from the start address) it continues up until the *PC_read_pointer* is reached. The *PC_read_pointer* should be updated frequenly, to avoid the DMA pointer surpassing the *PC_read_pointer* (or the other way around when the direction is set to *FromHost*), as this would cause the DMA to stop or reduce the transfer rate.

In order to keep track of whether Wupper is processing an address in front or behind the PC, the number of wrap-around occurrences are tracked. In the DMA status registers the even_cycle bits display the status of the wrap around cycle. The *even_pc* bit flags the *PC_read_pointer* wraparound and the *even_dma* flags a Wupper wrap-around. The bits start out as 0, and every wrap-around will toggle the bits. By keeping track of the wrap-arounds [18]

7 Optimization methods

When the implementation of Interlaken on the FELIX system is completed, there is still room for improvements. Within the protocol definition, there are some additional (optional) features that cam be implemented. If time allows, these optimizations will be added to the Interlaken protocol implementation.

7.1 Channel Bonding

Channel bonding is used when multiple transceiver channels are used in parallel to achieve higher transmission rates. The data words that are to be transmitted, are split into several parts (of a predefined bit length) and these are sent over separate channels. This process combines the separate communication channels together into one aggregate channel with larger bandwidth.

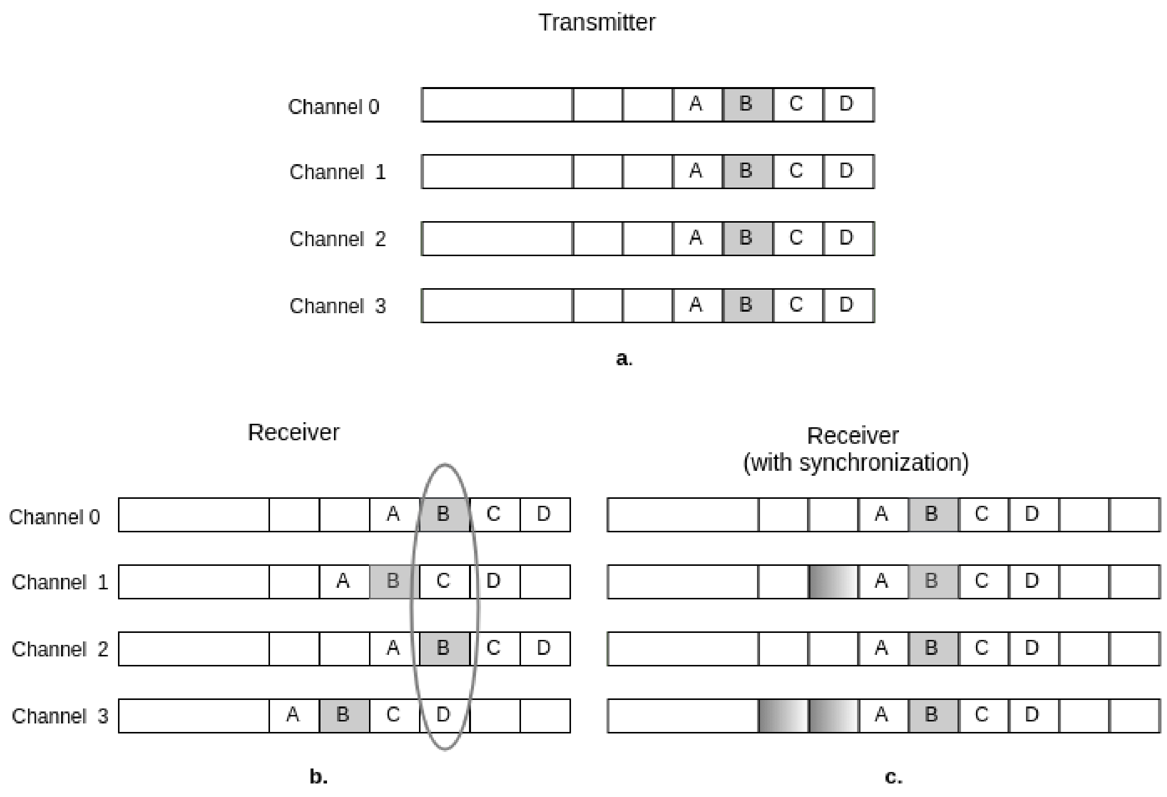


Figure 20: Channel bonding visualization.

The channel bonding process is visualized in Figure 20 where the (hexadecimal) dataword *BBBB* is split into 4-bit long parts, and transmitted simultaneously across 4 data channels (20.a). However, if the path between the TX and RX data channels creates various delays, a situation can occur where the data is misaligned and corruption of the data occurs because the receiver still expects the data to be aligned and collects the data in the same way as before. This process is visualized by the circle in Figure 20.b. The misalignment corrupts the data from *BBBB* to *BCBD*. To allow for the correction of these misaligned channels at the receiver, a lane-synchronization method needs to be implemented. This synchronization can then introduce delays (visualized by the Grey gradient blocks in Figure 20.c) between channels without delay, to re-sync them to the delayed channels.

7.1.1 Channel bonding in Interlaken

Within the Interlaken protocol, (see Chapter 4) the transmitted packets are all associated with a transmission channel (also referred to as a *lane*.) These channels all provide a serial stream of data, but for the moment, only one transmission channel is used. All of the data is transmitted serially over one channel. One of the reasons for only using one channel was because there was only one SFP (optical) link available on the previous hardware [16]. On the FELIX hardware however, there are 4 optical links available [21]. Optimization can be very rewarding, because it can improve the consistency of the link speed significantly. When data traffic increases, the single transmission channel can become congested. This will result in a slower link speed. If more than one transmission channel is used, the data can be divided between these channels, and the amount of traffic that the channels (together) can handle becomes higher.

According to the Interlaken protocol definition, an implementation of channel bonding is possible within Interlaken. The meta-frame format was especially introduced to allow for parallel data transmissions. Within the meta-framing process, the synchronization word can be used to provide lane alignment. In order to do this, all channels must send their synchronization words simultaneously, and the receiver must measure the skew between them. When the lanes are misaligned, additional *skip-word* meta-frame control words (see chapter 4.4) can be added for compensation.

The channel bonding functionality is currently not implemented in Interlaken, because only one transmission channel is currently used. It could however prove to be a useful addition.

7.2 Flow control

Once the channel bonding is implemented on Interlaken, and the four lanes are all transmitting data there is still room for further optimization. One of these optimizations could be to implement flow control to the Interlaken interface. Flow control measures the health status of each of the channels in the RX interface, and signals this back to the transmitter. An extra functionality can also be added to, for example, regulate the throughput on each of the channels.

The possible ability to communicate per-channel back pressure is a key feature of Interlaken. There are two implementation options within this feature; the *In-band* Flow control and an *out-of-band* flow control interface.

The on/off flow control status is communicated with a single status bit for each supported channel. Typically, a '1' in this status field signals an 'XON' status, which indicates to the transmitter that data can be sent over this channel. A '0' indicates an 'XOFF' status which signals to the transmitter that transmission should be ceased. Within the flow control protocol, there is no concept of tokens and/or traffic shaping other than turning the channels on or off. The transmitter can send as much data as it wants, until transmission is turned off again. The threshold for the raising of the XOFF signal is a programmable option. To provide a form of traffic control, the channels can be mapped to a so called *calendar entry* (which signals the behavior of the XON and XOFF signals.) By mapping the channels to several calendar entries, the flow control regulation can be extended. [1]

7.2.1 Out of band flow control

There is an out-of-band flow control solution defined within the Interlaken protocol to provide a solution for simplex interfaces. It is specified with three signals: *FC_CLK*: the clock to which the control data is synchronized, *FC_DATA*: The (1-bit) flow control status information, *FC_Sync*: a sync signal to signal the beginning of the flow control calendar. The out-of-band flow control signals are protected by a 4-bit CRC calculation which covers up to 64-bits of flow control data. The polynomial of this CRC is shown in Equation 6. [1]

$$X^4 + X + 1 \tag{6}$$

7.2.2 In-band flow control

When the in-band flow control function is implemented, the receiver will make use of the flow control status bits in the flow control words (see Figure 9.) This flow control bit-field in the control word is 16-bits long on bit[55:40]. These status bits, represent the ON/OFF flow control status for each of the Interlaken calendar channels with bit[55] representing the current calendar entry and bit[55+ *n*] representing the *n*'th entry. If needed, the flow control field can also be extended to bits[31:24]. To signalize the start of the calendar, the *Reset-calendar* bit of the *Idle/Burst* control words is used. When this bit is '1', the calendar entry status '0' appears in bit[55]. When the *Reset-calendar* bit is set to '0' and the flow control field continues to bit[55+*n*] until all the channels' status has been communicated. Because the flow control fields are included in the CRC24 check which is done in the *Framing burst* block (see Figure 7) a seperate CRC check for the flow control fields are not needed. [1]

8 Assignment details

During a previous internship, several point-to-point protocols have been researched and a proof of concept implementation with the Interlaken protocol was realized using an emulator board. This implementation of the Interlaken protocol needs to be verified,to make sure it functions according to the protocol definition. If it does not, it can not be implemented on FELIX because the behavior becomes unpredictable. To successfully implement the Interlaken protocol on FELIX, much research and testing is needed. If this new implementation proves to be successful, the Interlaken protocol can be implemented as part of the next iteration of the FELIX implementation. The different specifications will be divided into several different priorities using the MoSCoW method [22].

8.1 Must have's

During the project, research will be done on the Interlaken protocol and what is needed to implement this protocol on the FELIX system (more specifically on the FLX-card.) The current Interlaken implementation on the emulator board, needs to be verified to make sure it is according to the protocol definition. If the Interlaken implementation is verified, research will be done to see what is needed to implement the protocol on a FELIX board. The protocol will then be implemented on the FELIX hardware. Once the protocol is set up on FELIX, possible optimization methods can be researched and possibly implemented to improve the link efficiency.

In order to do this, a number of steps need to be taken first:

- The previous concept implementation needs to be verified: it needs to function according to Interlaken protocol definition (as described in [1].)
- Once the implementation is verified it needs to be implemented on the current FELIX hardware (FLX-712 or FLX-709) which consists of:
  - XCKu115 FPGA
  - Max 16.3Gbps per optic link
  - PCIe Gen3 x 16 (up to 128 Gbps)
- In order to realize this, research must be done on what is needed to implement Interlaken on FELIX.
- This is because, the (improved) Interlaken implementation needs to function alongside other FELIX firmware (already present.)
- To ensure compatibility between the different parts of FELIX, the output to PCIe should be in the current FELIX data format.
- To keep the current data output format, a between the Interlaken and the FELIX firmware needs to be developed.
- This link will function as the road between Interlaken and the other firmware, so the Interlaken protocol can be implemented on the FELIX hardware.
- Once this implementation is done, research can be done on possible optimization options.

8.2 Should have's

From the research on optimization methods, a recommendation will be made whether or not to implement it into the new FELIX implementation. If these optimization methods seem to be a valuable addition, they should be included in the new design.

- Add the optimization methods to the new implementation.

8.3 Could have's

The implementation tests will be done on low(/bit) level. Because this requires some preliminary knowledge on the subject, it might be useful to add a higher level user interface where test results can be visualized.

- Create a user interface where current data transmitted and/or the link speed is visualized, to make the implementation more user-friendly.

8.4 Deliverables

At the end of the project, the developed software and firmware will be published as an open source project on OpenCores [23]. This document will function as the documentation on the research that is done during the project. If needed, a user guide will be established on how to use the implementation. Finally, the solution will be presented in a presentation.

9 Functional design

In the previous chapters, a lot of theoretical research was done on the related projects developed at Nikhef and Cern, and on the Interlaken protocol and it's possible optimizations. From this theoretical research, the specifications of the design heve been set. These specifications, along with the theoretical research, will now serve as a foundation on which a design can be build. The practical design of this project is divided into three parts: First, the previous Interlaken implementation (Core1990 [6]) has be verified and (where needed) modified to make sure it functions according to the protocol definition. Secondly, a firmware solution must be designed which connects the new Interlaken implementation (Core1990_V2.0) to the firmware already present on the FLX-card (Wupper.) Third, if this implementation is successful, the design can be extended further, to include the previously researched (see Chapter 7.1) optimization method(s). This chapter will show the functional designs of al three parts of the final design.

9.1 Verificcation of Core1990

After the research of the Interlaken protocol (see Chapter 4) was completed, the Core1990 implementation could be analyzed. In order to test the implementation, the entire design will be simulated using *Vivado 2018.1* and all the separate "blocks" will be analyzed, by using several signals present in the design (or added to the design later) with which the data and control words can be followed through every step of the design. The architecture of Core1990 is shown in Figure 21. The TX and RX port of the Transceiver, will be connected, so that the design is placed in loop-back mode. This will make the design easier to debug, since only one instantiation of the design will need to be simulated. This will shorten the total simulation time.

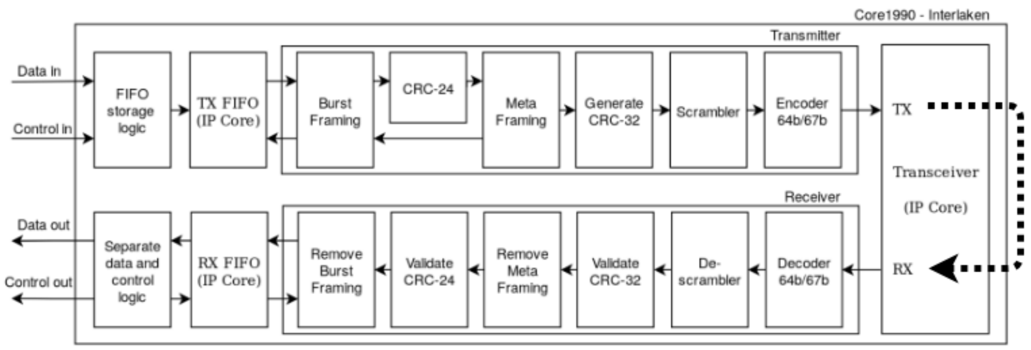


Figure 21: Complete core1990 architecture with the transceiver placed in "loop-back" mode (marked by the dotted line.)

Once the design is fully verified, every part should work according to the protocol definition. A suitable way to test the new implementation (core1990_V2.0) further, would be to find a commercially implemented version of the protocol and use it in parallel with the implementation to be verified. A test like this will also test the core1990's compatibility with other implementations. If the two Interlaken implementations can communicate, the conclusion can be made that the core1990 implementation is sufficiently according to the definition to allow for intercommunication. Previous research show that there are at least two commercial Interlaken versions available (see Chapter 4.7.) One of these solutions will be connected to the Core1990 implementation, to see if there is a possibility for communication between the two systems. Commercial implementations, are compliant with the Interlaken protocol definition, so if there's communication between the two (commercial and project) implementations the conclusion can be made that the data transfers of the implementation functions (enough) like the Interlaken definition, because both devices can communicate in the correct data format.

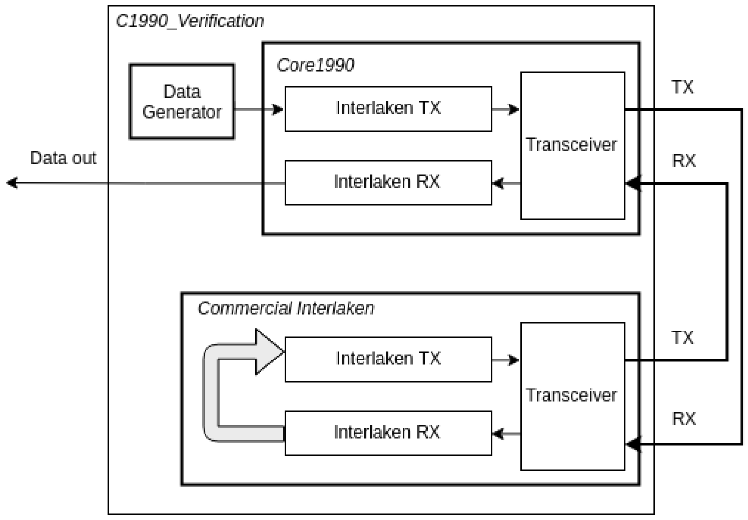


Figure 22: Functional design of the Core1990 verification setup.

The functional design of the system is shown in Figure 22. Data will be generated at the Transmitter (TX) input of Core1990. There, it will be framed into packets according to the Interlaken data format. The transceiver will convert the electrical signals to optical ones, and send the packets over to the Commercial implementation where they are converted back and received at the Receiver (RX) side of the commercial implementation. The packets are de-framed and the data is received and can be read by the user. However, to test both sides of the Core1990 implementation, the commercial implementation is *looped-back* which means that the RX output and TX input are connected (see the grey arrow in Figure 22.) This sends the data (that is received at the *RX_out signal* of the commercial implementation) back to the TX input of the Commercial Interlaken implementation. It will be framed once more, and send over the optic links to the RX side of the Core1990 implementation. The RX of Core1990 will de-frame the packets and the (original) data will be outputted at the *Data out* signal. If the Core1990 implementation functions exactly like in the protocol definition, the data generated by the *Data Generator* will be exactly the same as the data received at the *Data out* output.

9.2 Connecting Wupper and Interlaken

Once the previous design is fully verified, the next step is to implement the new version of the design (Core1990_V2.0) on FELIX. The implementation however, must be able to function alongside the firmware and software that is already present on the FELIX system. Furthermore, one of the *must-have* specifications set for the project (see Chapter 8) is that the data output to PCIe must be in the current FELIX data format. In order to meet this requirement, the design must become compatible with the current PCIe wrapper: Wupper. This requires analyses of the implementation of the Wupper design, and a connection needs to be made between Interlaken and Wupper. This connection must be set up in a way that ensures the integrity of both data formats. In order to achieve this, a piece of firmware is needed that serves as the connection between the two. This connection is visualized in Figure 23.

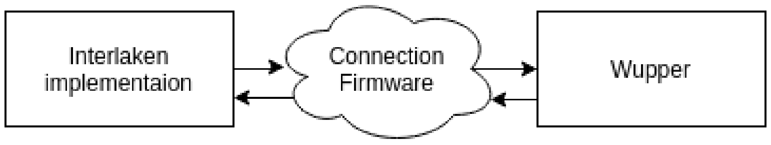


Figure 23: Visual representation of the connection between the Interlaken and Wupper Firmware.

9.3 Optimization methods

Once Interlaken is fully implemented on the Felix hardware, the next step will be to look for possible improvements within the system. The two most notable additions that can be made to the system are Flow Control and Channel Bonding (as researched in Chapter 7.) Both of these additions can be added to the Interlaken protocol, but in order to utilize the full potential of the Flow Control implementation, multiple channels should be used. When multiple channels are used, data traffic can be redirected to channels that have more bandwidth available. In the current implementation, only one data channel is used. By implementing channel bonding on Interlaken, the number of data channels can be extended (see Figure 24.)

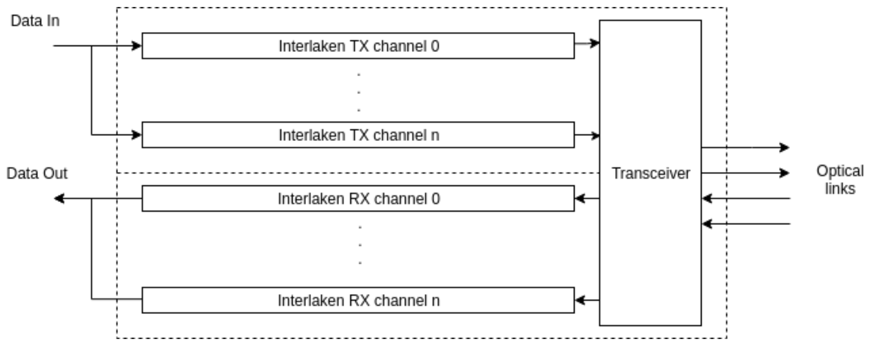


Figure 24: Visualization of multiple channels within Interlaken.

10 Implementation

From the functional design (described in the previous chapter) a general idea was formed on which solutions need to be developed and how they can be realized. In this chapter, the design will be extended further and the different aspects of the design will be implemented. Most of the design is implemented in the form of firmware developed in Vivado studio using the VHDL language. However, software applications are also needed, to control and initialize the FELIX PC.

10.1 Core1990 version 2.0

The previous research done on point-to-point protocols, concluded that the Interlaken protocol is a suitable solution to implement in the next version of FELIX. [6] In order to test the viability of using this protocol on an FPGA board with optical links, a proof of concept implementation of the Interlaken protocol was developed [17] and tested on two Xilinx VC707 [16] evaluation boards. These tests were done, to verify that the implemented protocol was working according to the specifications set, and to verify that the two boards could communicate with each other. However, it was never verified that the implemented protocol was actually the Interlaken protocol. In order to implement the protocol on FELIX, the protocol first needs to be verified, to make sure it functions according to the Interlaken protocol definition [1]. Once the protocol is verified, the next step is implementing *Core1990_Version2.0* on the Felix hardware.

The verification test will be done in several steps:

- First, the implemented protocol will be studied and a visual overview of all of the different subsystems or "blocks" of the protocol will be created.
- If all of the different parts of the protocol are checked, a new test run will be created where the full system will be simulated. The transceiver will be (internally) connected in a so called loop-back, where the transmitter side of the system will be connected to the receiver side.
- Once the loop-back design simulation is successful, the design will be connected to a commercial implementation. If these two systems are able to communicate, the implementation is verified as an implementation of the Interlaken protocol.

Once the new implementation is verified, the research to implement the protocol on the FELIX system can continue. The verification is a very important step. If the protocol does not function according to the Interlaken definition, it cannot be implemented on FELIX. If it were, it would lead to unpredictable behavior in the system. The errors that would then arise would be impossible to retrace, resulting in a dysfunctional overall system.

10.1.1 Summary of the protocol implementation

The communication between the different systems takes place by using so called control words and data words. The functional structure of the Interlaken system is drawn in Figure 25 The data coming in to the Interlaken transmitter, is framed into so called 'bursts' consisting of the raw data and control words. The control words include information about the burst size (size of a data burst and the interval between control words), start and end of packet, and a CRC-32 check.

After the *framing burst* block, the bursts are encapsulated into so called *meta-frames*. The meta frames are defined to support data transmission over Parallel transmission lines. They include four (unique meta-frame) control words, which can provide line alignment, scrambler information, clock compensation and diagnostic functions. A CRC-24 check is included on the data, the status of this check is included in the diagnostic words. The incoming words (with the exception of the synchronization and scrambler-state meta control words) are then scrambled. After the scrambling, the 66/67-encoding takes place, and afterwards the data is sent to the transceiver core, which transmits the data on a differential line (TX_PN).

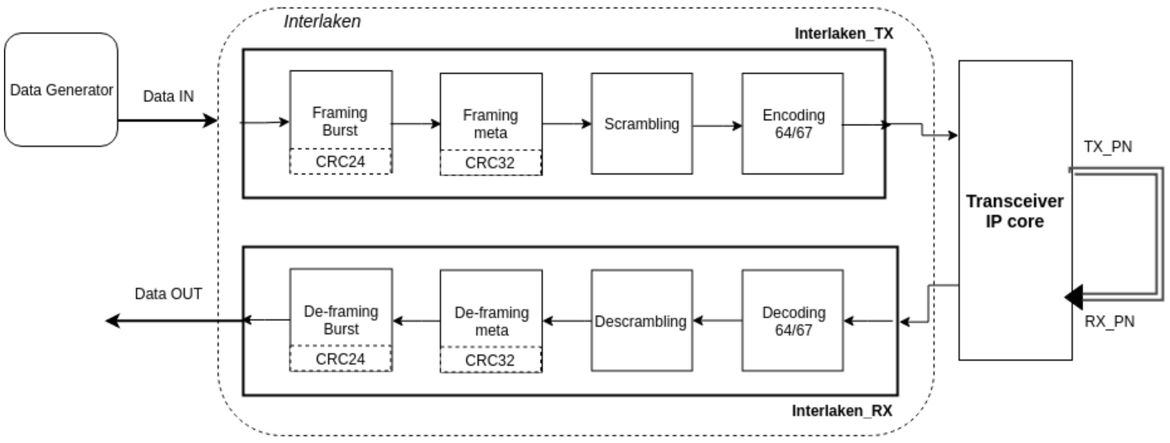


Figure 25: Structural view of the Interlaken Implementation.

The *TX_PN* and *RX_PN* of the transceiver are connected in such a way that the data going out of the transceiver is also coming back in at the receiver side. This means, that *Data Out* which is the output of the system, should be exactly the same as *Data IN* (which is generated by the data generator.) In order for this to be true, the two sides of the Interlaken protocol are mirrored to one and other.

The data coming out of the transceiver enters the *decoding* block where the 66/67 decoding is removed. Then, the data is de-scrambled and sent to the *de-framing meta* block where the data is de-framed, and the CRC-32 value inside of the diagnostic word (which was created at the transmission side) will be compared with the calculated expected value. If these values match, the data is sent to the next block; the *de-framing burst.* There, the data is de-framed and the CRC-24 check is done. When everything works accordingly, the output data (Data OUT) will be equal to the input data that was generated (Data IN.)

10.1.2 Building the project and recreating tests

A new project (*Core1990_Verification*) was developed in Vivado which recreated the Core1990 project. This project was simulated, and during simulation the input data created by the data generator was sent from the TX input (see Figure 25) through the Interlaken system, where it was received again at the RX output (Data_Out.) This means that data can be transmitted and received through the system, suggesting that the system works as intended. After simulation, a bit-file was created and programmed onto the hardware (Xilinx VC707 evaluation board [16], see Figure 26) while the optical link was connected in loop-back.

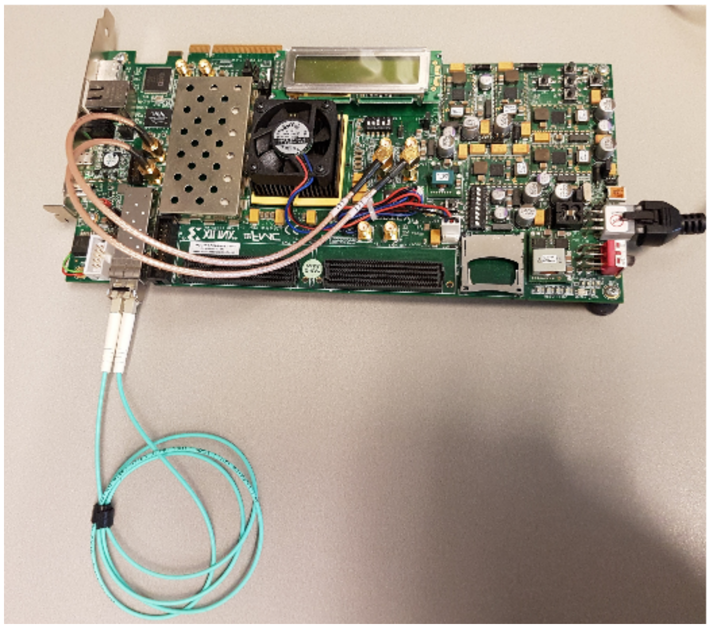


Figure 26: Xilinx VC707 emulator board used for testing Core1990

The implementation worked as expected, the data generated at the TX input was also present at the RX output, just like in the simulation. In the next test the communication was tested between two boards. In these test results the same data that was generated at one board was also received at the other board (see Appendix B.) This verifies the previous tests done in [6], but this does not yet mean that the protocol actually functions according to the Interlaken definition. In order to verify this, more tests are needed.

10.1.3 Communications tests with the Xilinx Interlaken core

As mentioned in Chapter 4.7, Xilinx and Intel have developed ip-cores that use the Interlaken protocol. The Core1990 implementation was developed for Xilinx hardware, in Vivado. The Xilinx Interlaken ip-core is also designed and tested for Vivado, so the two implementations can be added together in one project. This will make simulating and designing the verification project easier. The functional design from Figure 22 was implemented in the *Core1990_Verification* project, resulting in the design in Figure 27. The Xilinx core has many status and functional signals that are described in the documentation of the core [14]. These signals make it easy to follow the data flow within the design, and see which functionalities are not working correctly. This makes this core very suitable for the debugging of core 1990, since all the error signals that might arise will point to the parts of the implementation that are not working according to the definition. For example, a *burst_err* signal that arises in the Xilinx IL core, suggests that there is still something unexpected happening with the de-framing burst block. This type of error could be caused during the framing-burst block on the Core1990 side. If all the error signals are monitored, and the corresponding blocks of the Core1990 implementation are checked for errors, the core can be debugged.

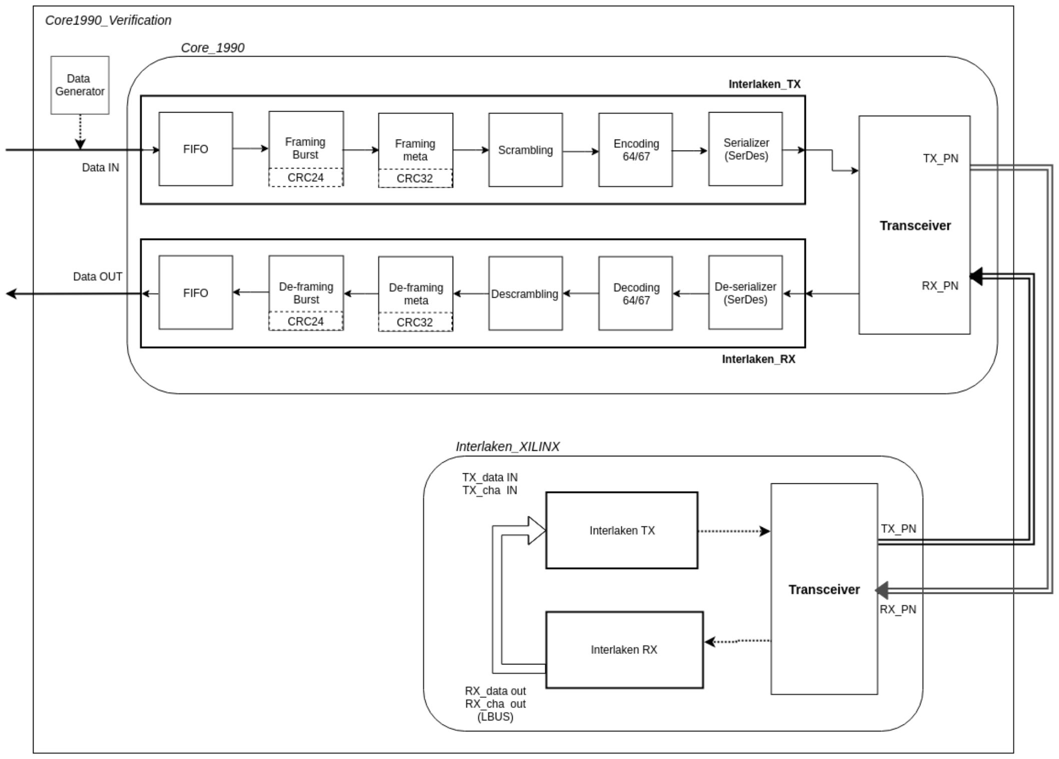


Figure 27: The core1990 implementation connected to the Xilinx Intergrated Interlaken core.

Since the Xilinx core already includes a working implementation of Interlaken, the data is looped-back on the Xilinx side. This way, the Xilinx core will raise error flags when an error occurs. These errors can then be examined at the core1990 block. The data is generated by the data generator and sent to the Interlaken TX input. The data is framed and the Interlaken frames are inputted to the transceiver where the bit-stream is sent serially to the Xilinx RX side of the transceiver. From there it arrives on the RX_in side of the Xilinx core, where it is decoded, de-scrambled, de-framed and the original data arrives on the Xilinx RX output. From there it is sent over the LBUS to the TX input of the Xilinx core, where an Interlaken frame is created once again and it is sent through the transceiver back to the RX side of the Core1990 core. There it is de-framed again and the original data is retrieved on the Data_out side of the Core1990 implementation.

At the Xilinx core, packet data is provided to the Local Bus (LBUS) TX Interface, and received from the RX LBUS interface. The LBUS is used by the Xilinx core to facilitate the implementation of multiple input/output data channels. The LBUS is not implemented in the Core1990 implementation, because currently only one data transmission channel is implemented in the Core1990 Interlaken implementation, which means that for the current communication, only one serial channel will be used. Another functionality that is already implemented in the Xilinx Interlaken core (but not in Core1990) is flow control. The RX path can extract the flow control information present in the control words, with which the TX data flow can be regulated. Because these functionalities are not yet implemented, the channel number will be put permanently on '001' and the flow control functionality will not yet be used.

10.1.4 Debugging of the core

Once all of the parts of the design were set up and the connections between them were made, the project was simulated. The firmware of the implementation was studied, and the behavior of each block was compared to the Interlaken protocol definition. The two implementations (Xilinx_Interlaken and Core1990) could not communicate once the simulation was run, and several error signals were raised among which were:

- *stat_rx_bad_type_err*: Unexpected or Illegal Meta Frame Control Word Block Type. These signals indicate an unexpected or illegal Meta Frame Control Word Block Type was detected.
- *stat_rx_burst_err*: Burst Error. This signal indicates that a burst length error was detected.
- *stat_rx_crc24_err*: Control Word CRC24 Error. This signal indicates whether or not a mismatch occurred between the received and the expected CRC24 value. A value of 1 indicates a mismatch occurred.
- *stat_rx_crc32_err*: Control Word CRC32 Error.

A structural approach of trying to solve these errors is needed. After all, solving one error might cause several others to arise if it is not the original source of the problem. As a debug approach, it was decided to check the core1990 implementation "per block" which means that the *Framing burst* and *De-framing burst* blocks will be checked first, and then the *Framing/de-framing meta* block will be checked etc. The reason for this, is that the TX and RX components of an Interlaken implementation are mirrored to one and other. If for example, the *Framing burst* block works too fundamentally different from the *de-framing burst* block, the de-framing of bursts will not be possible. The debug process will now be described in more detail.

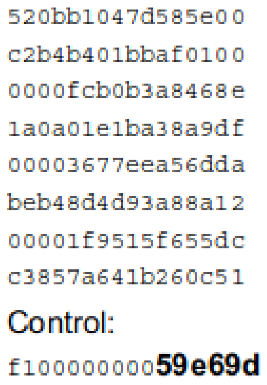
Framing Burst/De-framing burst

The creation of an Interlaken frame, begins in the *framing burst* block. The data (payload) is enclosed in control words and a burst is created. The payload data can be divided between several bursts, or sent in just one burst. One of the things that was questionable in the core1990 implementation is the fact that the header bits of every data word are not added to the data stream. Instead, they are generated separately from the data and channeled directly into the next block (framing meta.) This is questionable, because if the header bits of the data words are treated as different signals, it is very difficult to see which headers belong to which data (and/or control words.) To solve this, the headers were added to each data stream in the system, effectively changing the Data_out signal from a 64-bits signal to a 67-bits signal. Because of this change, the data and header are always together. In the protocol definition, the headers are also part of the data stream, so this change also brings the implementation closer to the definition. The functionality will remain the same (one 66-bits word, instead of a 64-bits word and a separate 3-bits word) but the readability of the program and simulation will improve greatly. Furthermore, it is easily re-traceable (from the header bits [66-63]) what kind of word (data, burst control or meta control) the data consist of. The user could decide to split up the headers again within a block, as long as the data stream between blocks is one 67-bits signal that includes the header. Besides the readability and trying to follow the protocol definition as closely as possible, the addition of the header to the data stream was also done to hopefully solve the burst error signals that arose. This was however not the case, so the problem was situated elsewhere.

CRC-24

In the Xilinx Interlaken core, a lot of CRC24 errors were raised. Not just every few seconds, but every time a CRC-check was performed. After further examination of the design, possible timing issues (the possibility of the two correct values not lining up on the exact CRC-calc clock cycle) were ruled out. This suggests that there is something fundamentally different in the way the CRC check values are calculated on the Core1990 side and the Xilinx side of the system. The CRC-24 part of the framing/deframing burst block was analyzed, and it was discovered that the polynomial used was not the same as the one specified in the protocol definition (see Equation 1 in Chapter 4.3.) The polynomial used was a polynomial that was 32-bits, and which was normally used for Ethernet. It did not correspond to either CRC polynomial defined in the definition. The use of the incorrect polynomial results in a different calculation on the RX side and the TX side , which means that the two calculated check values will never be the same. This results in a lot of CRC errors and the discarding of data on the *Xilinx-Interlaken core* side.

As a solution to this problem, the CRC-32 (from the original implementation) was changed to a CRC-24, and the correct polynomial was calculated (see Equation 2 from chapter 4.3) and implemented. This did not, however solve the errors, since the (by core199) calculated checksum value still did not match the value expected by the Xilinx system. Despite further checking of the data path, it was still unclear if the incorrect checksum values resulted purely because of a wrong CRC-24 calculation method, or if there was something within the Core1990 system that caused it. Fortunately, the Interlaken protocol definition includes some details on how the CRC-24 values are calculated (see Appendix B in [1].) These details also include some test data and the corresponding control word that should follow at the output, after the test data is entered into the CRC-24 calculation (see Figure 28.)



```
520bb1047d585e00
c2b4b401bbaf0100
0000fcb0b3a8468e
1a0a01e1ba38a9df
00003677eea56dda
beb48d4d93a88a12
00001f9515f655dc
c3857a641b260c51
Control:
f100000000**59e69d**
```

Figure 28: Example test data from the crc-24 calculation details in the Interlaken protocol definition.

To test if the CRC-24 calculation was performed correctly, it was tested in a standalone project. In this test environment, it can be tested if the faulty calculation is caused by the CRC-24 or by something else in the Core1990 project. If the calculation is working as expected, the conclusion can be made that there is something else going wrong in th

Core1990 system, and the CRC-24 is correct. In this project, the test data from Figure 28 was used as the input data and the original crc-24 check was performed. Once again, the calculated CRC-checksum did not match the control value in 28. This confirms that the calculation of the CRC-24 is indeed not according to the Interlaken definition.

According to the Interlaken definition, the data is sent through the CRC-24 calculation function in the order of byte transmission, where the most significant bit is sent first. The checksum value is generated by resetting the polynomial to all ones, sending the data stream through the polynomial function and finally, inverting the value before transmitting it within the control word.

In the original CRC-24 calculation (which is the same one used in the FELIX project) the CRC-24 values are calculated over the data and afterwards the CRC-24 is calculated again over the previous result. This is a completely different method from the Interlaken definition. because (even if the CRC-24 is only done once,) the poly value is not reset to all ones before calculation, and the final calculated value is not inverted before transmission.

An implementation of a new CRC-24 can be made by dividing each bit of the data by the polynomial one bit at the time. This would however take an excessive amount of time (and clock cycles) to calculate. Because of this, CRCs in FPGAs are usually implemented with multiple XOR functions operating on multiple data bits per clock cycle, whose sequence is computer generated. This optimizes the CRC calculation until it only takes one clock cycle to compute. Within the documentation from Intel [24] an example has been given on how to calculate the CRC-24 with the Interlaken Polynomial. Based on this, a new CRC-24 implementation was made, and tested with the test data from Figure 28 in the test project.

The wave window resulting from the simulation of the test data is shown in Appendix D. There, it is shown that the test data from Figure 28 is used as input for the CRC calculation (the *crc_calc_in* signal.) The polynomial value (signal *crc_24_test_in*) is reset to all ones, and from then on the polynomial value advances with each new data word, until it is reset again. The output of the CRC calculation is also shown (signal *CRC24_Test_out*) but this is not the final CRC value, as this value also needs to be inverted in the final step of the calculation. The final CRC-checksum value is shown by the signal *CRC24_Test_Final* and after all of the example data has passed, including the control word that will include the crc checksum value, the final checksum value is *59e69d* (see the final value of the *CRC24_Test_Final* signal in Appendix D). This corresponds to the value shown in Figure 28. An important thing to note, is that bits [23:0] of the control word which will include the CRC checksum, should be reset to all zeroes for the calculation of the checksum. The control word from Figure 28 will become *f100000000000000* in calculation.

Since this final test resulted in the correct checksum value, it can be concluded that the CRC-24 implemented is now working correctly. The implementation was added to the Core1990 project, which was simulated again. From the simulation it could be seen that the previous CRC-24 errors had disappeared.

Framing Meta/De-framing Meta

In the *Framing-meta* block, all of the checks to see what kind of control word the incoming data is, (synchronization, skip-word,scrambler-state,data or diagnostic) were done by checking a string of individual bits. This is not very readable, since in order to figure out which word is checked, the format of each control word has to be known. To solve this, each control word was added as a constant value. This made the debugging of the implementation easier, since instead of a binary value, text was used (for example *SYNCHRONIZATION, DIAGNOSTIC* etc.)

In the Diagnostic Meta control word type, there are some optional functionalities such as flow-control. But there are also compulsory functionalities that need to be included in this control word, such as the CRC-value and the so called health bits. These health bits are bit 33 and 32 of the Diagnostic meta control word, They signal the health of the lane the word is transported on, and the health of the entire interface, where '1' signals a healthy condition, and '0' signals a problem. While these health bits were implemented in the previous design, they were implemented in such a way, that '0' represented a healthy condition and '1' signaled an error. This is the exact opposite to the protocol definition, which states that '1' signals a healthy condition. This might have caused some conflicts with the Xilinx implementation, so it was fixed.

CRC-32

In the CRC-32 check, which is part of the framing meta block, the polynomial did not correspond to the one used in the protocol definition. This resulted in a false calculation of the CRC-checksum value. When the system is functioning in loop-back operation, this does not have notable consequences, after all the same polynomial used on the transmitter side to calculate the checksum is also used on the receiving side to calculate the CRC value there. It starts being a problem, when the device that the implementation functions on, has to communicate with other devices running a (legitimate) Interlaken implementation. The CRC value calculated there, will never result in a correct check with the checksum value transmitted in the control word, because a different polynomial is used on either side.

The correct Polynomial value was calculated, and implemented. The calculation of the CRC-32 was changed in a similar manner to the CRC-24 calculation in section 10.1.4. It was verified (with the correct polynomial) and implemented back in the Core1990 design.

Scrambling/De-scrambling

The data scrambling process is fully shown in appendix B of the Interlaken protocol definition (see [1].) From this basis, the (existing) core1990 scrambler implementation was made, and its functionality was presumed correct. During the simulation tests however, the TX data which was sent was not coming through to the RX side. After each previous block was checked, it was discovered that the data was not going past the *de-scrambling* function block correctly. Upon further comparison to the scrambler method in the definition, it was discovered that some functionalities of the scrambler, while they were implemented correctly, were not done in the correct order (the updating of the scrambler state is done with several XOR functions, one of these was in the wrong order.)

The scrambler works, by updating it's *scrambler state* with each new incoming word. Every data bit is XOR-ed with this current scrambler state, resulting in scrambled data. The synchronization and scrambler state meta-frame types, as well as the header bits of the data, are never scrambled. The scrambler state, does not update when these words are presented. In the previous implementation, the scrambler state advanced anyway when a scrambler state or synchronization word was passed through (without scrambling them) this resulted in an incorrect scrambler state used to de-scramble the data. This resulted in a different scrambling and de-scrambling process than the one described in the definition, the data was never de-scrambled correctly, and only gibberish was received on the RX side.

The errors in the scrambling method were found, and corrected, to make sure that the scrambling process as described in the definition was used. Afterwards the data was received unscrambled, and corresponded to the TX data which was sent.

Encoding/Decoding

An encoding method is required, to provide enough state changes to allow clock recovery and alignment of the data stream in serial interfaces. In the Interlaken protocol, a 64/67b encoding is used, which is a variant of the 64/66b encoding that is used in Ethernet (and defined in the IEEE 802.3ae 10 Gigabit Ethernet specification.) The encoding is used in combination with the scrambler, which results in a balance between the '1' an '0' values transmitted. If this encoding was not performed, a situation could occur where a string of only '1' or only '0' bits were transmitted, resulting in a DC baseline offset. The encoding and decoding methods were checked, and it is working as expected from the protocol definition stated in [1].

10.1.5 Core1990_V2.0

One of the most important changes to the implementation, is the fact that the data stream throughout the system was changed back to a 64-bit word system. This is the way it is described in the system, but it was previously not implemented that way. Only the actual payload data was transferred directly, but all of the information signals within the control words were treated as separate signals. These signals were transferred along with the data and pasted in the final data output, but every field of the control words (see Figure 9) was divided up in signals and transferred that way between the blocks of the implementation (see Figure 7.) This technically has the same result at the output, but it makes it very difficult to see which signal values belong to which data. This issue was solved, by fundamentally changing the data transfers between blocks, so that the data stream was made up of 64-bits data and/or control words. This way, it became very clear to see which signals belong to which data and the readability was improved greatly as well as the implementations efficiency. The implementation of the addition with the headers to the data also solved the *stat_rx_burst_err* because it was signalled more clearly where the burst ended, and the SOP and EOP signals were included in the control word transmission, so they could more easily debugged. This showed that some EOP signals did not arrive properly, and the necessary fixes could be made in the firmware.

Another change that improved readability was to add several *constants* to the firmware, which functioned as checks for certain bit-values corresponding to certain control words or meta-frame block types (for example for the Meta Frame *Block Type* bits [62:58] "011001" refers to the *Diagnostic* Meta Frame Type, this sequence was connected to the constant variable *Diagnostic_Meta_Type.*) This improves readability within the firmware greatly, since not all bit sequences need to be looked-up to be able to read the code. This helped in solving the *stat_rx_bad_type_err* error, because it was discovered that there was a creation of an "unknown" block type being created in certain instances.

Another important change was the attempt to improve the efficiency of the code. Some parts of the firmware were duplicate or not needed, and because the implementation needed to function along with other firmware on the FLX-card, it is important that everything is done as efficiently as possible, to prevent timing issues during hardware implementation.

Another important error in the design was the way the CRC-checks were implemented. Both of the Polynomials used did not correspond to the ones that were mentioned in the definition, and the CRC-24 was implemented as a CRC-32. Research was done on how the calculations should occur, and the correct polynomials were calculated using the presented equations in the protocol definition (see Chapter 4.3) and implemented.

Something else that was not implemented according to the definition, was the *Scrambler* implementation. The protocol definition (Appendix B in [1]) provides the exact way to implement the scrambler, but this was not exactly followed, resulting in a faulty scrambling process. The process was fixed and tested, resulting in a correctly functioning scrambler.

After the changes to the Core1990 implementation were made, the full implementation needed to be tested again to see if it was still functioning according to the definition.

10.1.6 Results

After all of the changes in the design were made, it was time to test the design again in loop-back mode. This tests solely the Core1990_V2.0 design, by shorting the TX and RX differential lines (shown by "1" in Figure 29.) This test was successful, each block of the design was (once more) checked to see if it was functioning according to the definition, which it was, and the data which was generated by the data Generator, was received at the *Data_out* output.

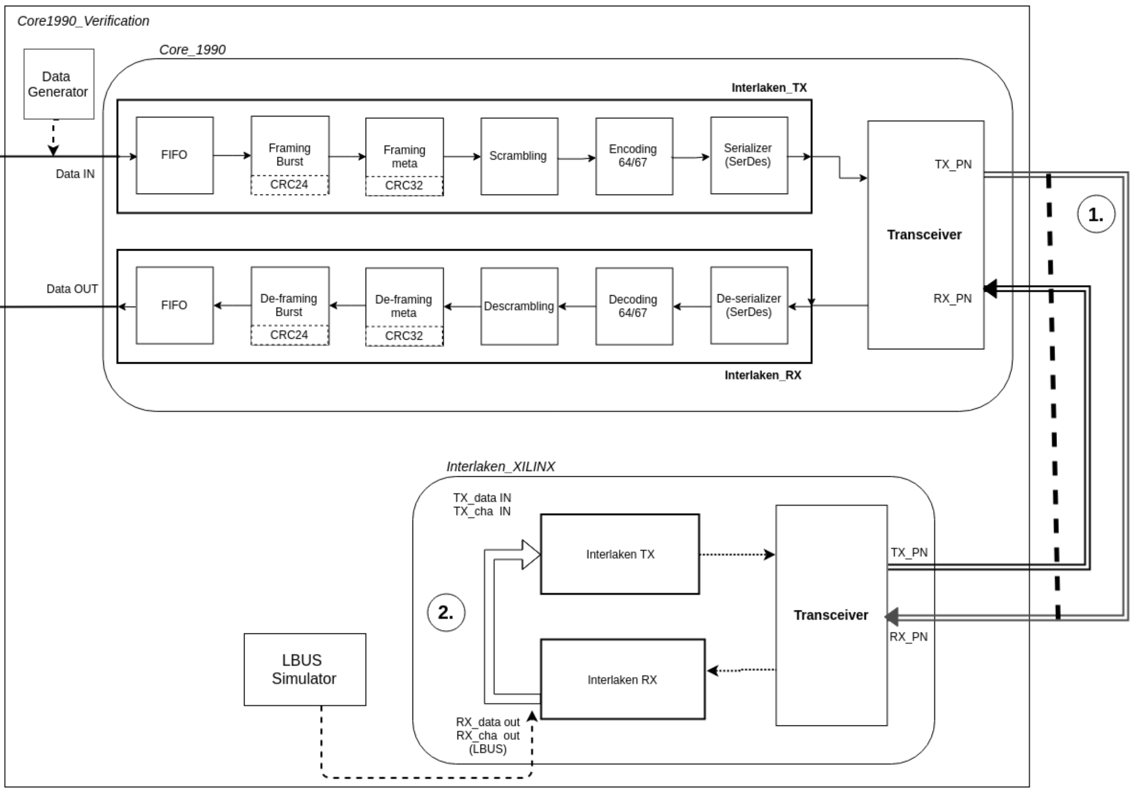


Figure 29: Block diagram of the communication test setup between the Core1990 implementation and the Xilinx Interlaken Implementation. Both systems have their own data generator that can be used, for the Xilinx core an LBUS simulator was made (see *2*) which provides parallel test data. This data can be sent back to the Core1990 side, but the systems can also be placed in loop-back mode. This is done by shorting their RX and TX communication lines, at *1.*

Since the improved Core1990 implementation was functioning as expected, the next step would be to test it again while not in loop-back mode (with the line "1" in Figure 29 not drawn). It seemed to function according to expectations, but there was no data received at the *Data_out* output of the Core1990 side. Since there were no longer errors raised, there must be something else going wrong. The data was received at the Xilinx RX input, but it was no longer present on the LBUS (See "2" in Figure 29) and also no longer present on the RX-side of the Core1990 side. This suggests that either there was something going wrong within the Xilinx core (which was not visible in the error signals) or there was a problem projecting the data onto the LBUS. This seems like a reasonable explanation, since the Xilinx core expects data from multiple data channels to arrive. If there is only one channel that is providing data, the data from the other channels should be zeroes. To check if the error was indeed spawned because of problems with the LBUS, an *LBUS simulator* was made, which generates parallel test data which is easily recognizable on multiple channels. This data will be put on the LBUS, where it will be sent over multiple channels, and after a set time the *start of packet* and *end of packet* control words will also be generated. This data is then received at the Xilinx TX input, and it will be sent over to the Core1990 RX side. There it will be received, and if no errors occur, the data will arrive at the *Data_out* output. After another simulation test, the *LBUS simulator* data did indeed arrive at the *Data_out* output of Core1990 (see Appendix E,) and no error signals were raised on either side. This confirms that the new Core1990 implementation is working according to the definition, because it can de-frame the Xilinx Interlaken frames. It also confirms that the problem with the interconnection lies somewhere with the compatibility with the LBUS definition.

10.2 Wupper-Interlaken implementation

In order to successfully implement the Interlaken protocol on the Felix hardware, the implementation also has to be compatible with the FELIX firmware. More specifically, it has to be compatible with the Wupper firmware (see Chapter 6.) The core functionality of Wupper, is to provide a PCIe wrapper to transfer data to (and from) the host PC using DMA. In order to implement the Interlaken implementation on FELIX, a connection has to be made between the existing Wupper firmware and the Interlaken firmware. This link will be realized by creating an application (consisting of both firmware and software) between the Interlaken implementation and Wupper that can transfer data between the two, without disrupting one or the other.

There are also some additional changes required for the Core1990_v2.0 implementation, because the hardware that the implementation was created for (the Xilinx VC707 evaluation board, see [16]) is different from the FELIX hardware (Xilinx VC709 [21]). One of the most notable hardware differences are the Gigabit Transceivers used in the FELIX card. There are 4 SFP Transceivers on the FELIX card, as opposed to 1 on the Interlaken development board and the schematics are different. The pin-out of the FPGA and the positions of the clock signals have also changed.

As mentioned previously, the developed application consists mostly of firmware and some software. The firmware will be developed first, and afterwards the solution will be simulated with Vivado ([25]) and if the simulation behaves as expected, a bit-file will be generated which will be implemented on the FELIX hardware (FPGA.) A debug IP core will be added to the implementation, to facilitate the debugging process (this makes it possible to generate a simulation-like wave window in Vivado.) On the FELIX PC, the Wupper drivers will be generated, and the Wupper interface can be started using the wupper-tools (see Chapter 6.) The application-specific software will be added to the wupper-tools directory for elegancy (since it uses the Wupper wrapper) and it can be started in a similar manner to the wupper-tools. The implementation can be debugged by adding registers to the Wupper register map (see Appendix H,) which represent certain debug signals. These registers can then be read from the Felix pc, using the Wupper tools software. Several signals such as loss-of-lock (for the PLL) or loss-of-signal (For the optical links) were added so they can be read from the registers (see Table 7 in Appendix H.) For other signals that are for example, time sensitive, it can be useful to add another debug method. As a a second debug method, a debug probe component was added in Vivado [25] which generates a simulation-like environment within Vivado, which is linked to the device that is programmed. This provides a simple debugging method for the signals within the firmware, without having to write them all to registers.

To implement Interlaken on the Felix hardware, it has to be able to function alongside the components that are currently implemented on the Felix-card. The incoming data from the optical links (once it has travelled through Interlaken) has to eventually be transferred to the host PCs memory, and the other way around. This is done with Wupper, (the PCIe wrapper see Chapter 6). In order to connect Wupper and Interlaken, the Wupper output data has to be collected and re-shaped to the Interlaken input format. On the other side, the header data that travels with the Interlaken output data has to be preserved. As a solution to this, the Application produces two extra "blocks" of firmware that lead the data into the correct format: *IL_to_Wupper* and *Wupper_to_IL* (see Figure 30.)

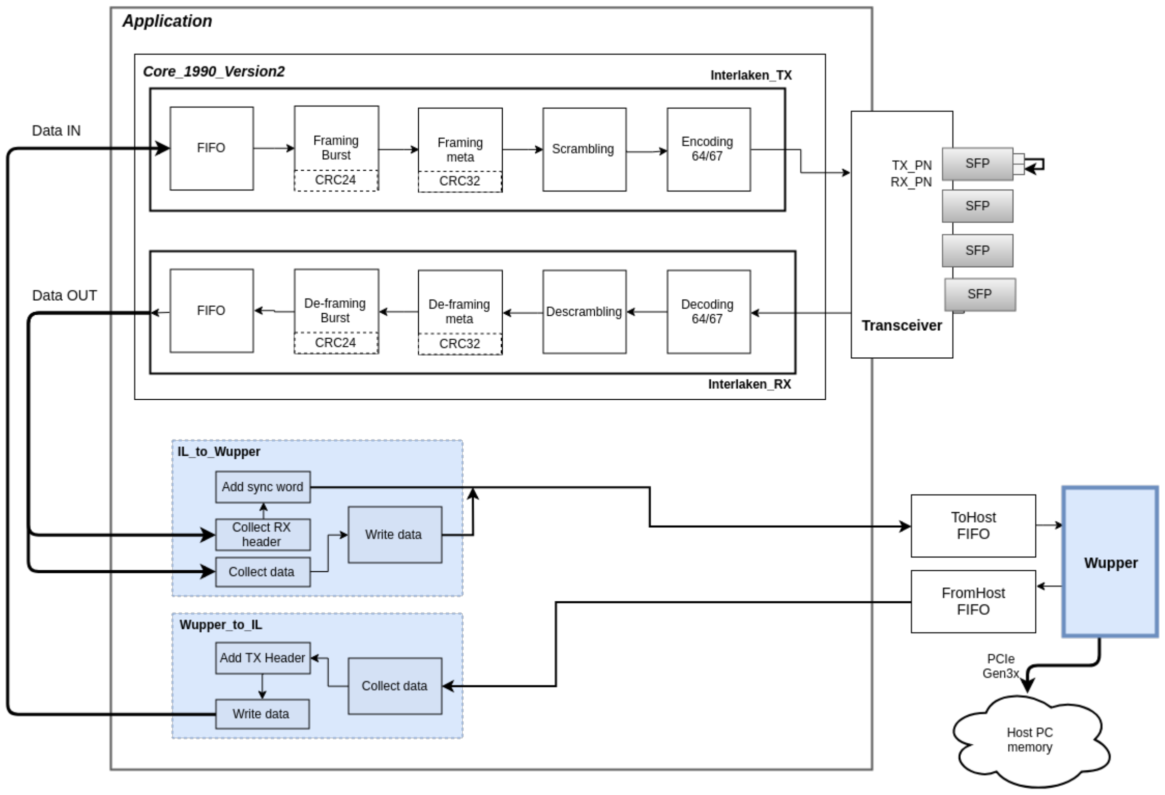


Figure 30: Block diagram of Interlaken, Wupper and the Application in between..

10.2.1 Interlaken to Wupper

The data outputted by Interlaken, is received by the application. There, several other signals are also collected from Interlaken, which are all stored in a (64-bit) so called *Sync_INFO_Word* which format is shown in Figure 31.

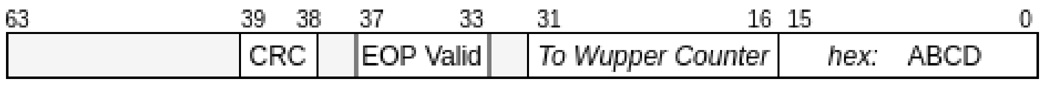


Figure 31: Sync Info word format.

These signals include a *CRC-24_occurred* signal on bit [39] and *CRC-32_occurred* signal on bit [38], which are set to '1' if a CRC error has occurred in the previous data burst. The EOP-Valid bits from the last EOP control word are also stored. The *Sync_INFO_Word* is recognizable because of the hexadecimal value *ABCD* included at bit [15:0] and also includes a *To Wupper Counter* value, which can be used to check if the right amount of data words have passed. Between these predefined fields, the word is padded with zeroes to assure the 64-bit length of every control word. The *Sync_INFO_Word* is sent to the *ToHostFIFO* after a RX EOP signal has been received. The control words can later be filtered by the program which reads the data form the PC memory, but they provide useful information about the previous data burst. Once other functionalities are added, other useful information (such as the transmission channel) can be added to the *Sync_INFO_Word* where needed.

10.2.2 Wupper to Interlaken

The data that is collected from the host-PC memory, is transferred through the Wupper system and eventually ends up in the FromHostFIFO (see Figure 30.) From there it enters the Application, and is collected. It cannot be directly transported to Interlaken however. The data from the FromHostFifo is 256-bits long, but the Interlaken TX FIFO data input should be 64-bits long. The Interlaken TX-input also expects the SOP (Start Of Packet) and EOP (End Of Packet) signals, to know when a data packet begins and ends. These signals are needed to create the SOP and EOP control words which delineate the data bursts. The signals are added to the data, in the form of a header on top of the 64-bits data stream that is created. This header contains several bits that signal the SOP (Start Of Packets), EOP (End Of Packets) and the EOP-valid bits which are required by Interlaken. The SOP is created at the beginning of the transmission, from where the EOP is generated after a amount of data equal to the pre-defined *packet_length* value has passed. The *packet_length* value is collected from one of the (application-specific) resisters.

Once the header is added and the TX-FIFO is ready to receive data, the data is written in the (Interlaken) TX-FIFO. There, the header signals are stripped again and sent to the Framing burst layer along with the data.

10.2.3 Application Tests

To test the firmware implementation, the complete design was simulated in Vivado, and data was generated with a data generator implemented in the project. For the final implementation however, the data needs to be read from the the Host PCs memory, before it is sent to the Interlaken system. For the implementations hardware tests, the data must already be present in the Host PCs memory, before it is sent to the Wupper implementation and then to through the firmware Application to Interlaken. To create this "initial/test data" and to read the data from the host PC memory, a software application was made. This software application uses the functionality of the Wupper-tools, to initialize the Wupper card, flush the FIFOs and reset the DMA controller. The data is generated from the software, of can be generated externally. The data is stored in the memory of the Host PC.

10.2.4 Clocks

Clocks are a very important part of the design. To ensure that the clocks that are generated are stable and to minimize their jitter, they are buffered and run through ICs that are present on the hardware. These ICs can be programmed to generate a clock signal on a large number of frequencies. Configuring these clocks is done by writing their registers, that are specified in their documentation.

On the Felix hardware, a clock can be created with the SI570 programmable voltage controlled crystal oscillator (VCCO) [26] present on the board. The SI570 can generate clock signals on virtually any frequency between the range of 10MHz up to 1.4 GHz. It consists of a digitally controlled oscillator (DCO) based on DSPLL technology (Digital Signal processing PLL, see [27].) which is driven by an internal fixed frequency crystal reference. The default output frequency is set, but can be reprogrammed through the I^{2}C serial port. By default, the output frequency is 156.25 MHz.

The SFP Transceivers on the FELIX hardware also need a stable clock signal, to ensure data integrity. With the crystal present in the SI570, a 156.25 MHz differential clock signal was created. In order to stabilize this clock signal and to route it through the FPGA a number of buffers are used. (See Figure 32.) These buffers stabilize the clock signal generated from the Crystal and help connect it to the hardware parts that require it. The 156.25 MHz generated from the SI570 is connected to the *User_clk_in* signal which is routed through an *IBUFDS* buffer (IO bank buffer,) then an *BUFG* buffer (global buffer) and finally via an *OBUFDS* buffer (Output buffer) to the input of the SI5324 [28] present on the board.

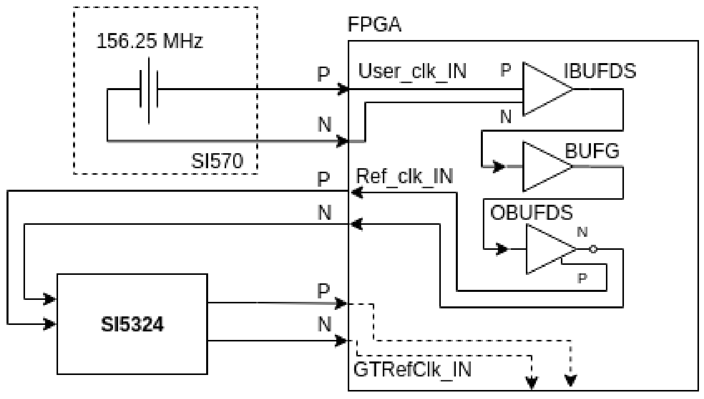


Figure 32: Clock buffers path from the crystal to the SI5324 to the Gigabit Transceiver clock input.

The buffers are needed to filter out most of the irregularities, before the signal enters the SI5324. These irregularities need to be solved, before going into the SI5324 and it's internal PLL, because otherwise the PLL will not achieve lock. Of course the PLL can be forced into lock by broadening the bandwidth, but this is undesirable (because then more jitter (noise) will be introduced.) The SI5324 is a jitter attenuating clock multiplexer. It filters the input clock signal, to ensure a reliable stable output clock. These kinds of filters are needed to minimize irregularities that arise within clock signals, for example propagation delay (the time needed for an output pulse to arrive after an input is applied,) phase errors or variations in period times.

The functional block diagram of the SI5324 is shown in Figure 33. Two input clocks can be provided, which are sent to the DSPLL that uses either a crystal or reference clock to provide the jitter attenuation. The input clock(s) can also be divided to create an output clock with a different output frequency. The control of the SI5324 is accessed through an I^2C interface, where the input clock frequency and clock multiplication ratio are programmable by writing the registers. The device is based on DSPLL technology, and the PLL bandwidth is digitally programmable. A fast lock feature is available to reduce the lock times of the PLL.

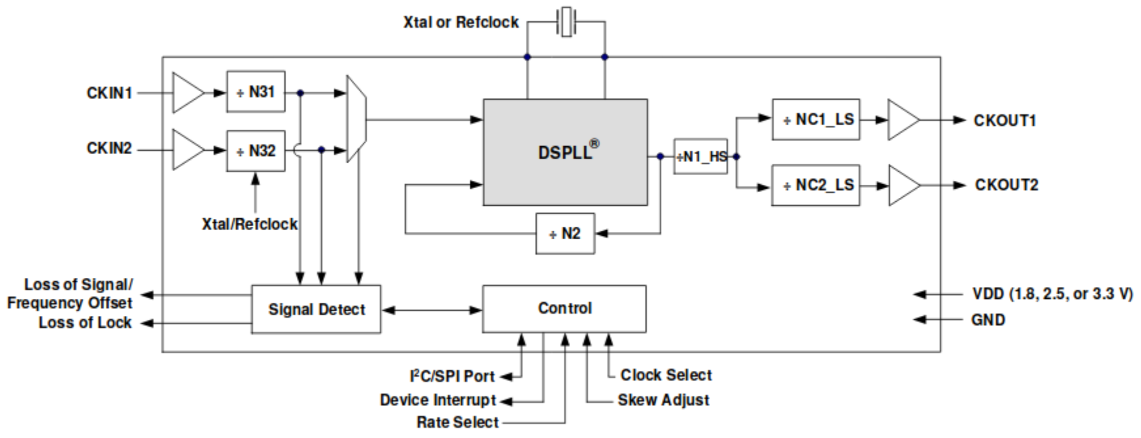


Figure 33: Functional block diagram of the SI5324. [28]

Within the current implementation, the input for the SI5324 is the *Ref_clk_in* signal (as shown in Figure 32.) There is no frequency division needed, since the needed output frequency is the same as the input frequency (the SI5324 is only required for Jitter attenuation) other control settings such as the selected reference signal, whether or not fast lock is turned on etc. can be found by studying the register map in Appendix A. These control settings determine the layout of the register map, and this register map is required to set up the SI5324 correctly. The register map is added to the *wupper-init* software, which will then set up the SI5324 (along with other devices and or functionalities specified) and initialize the wupper card.

Once the wupper card is set up, the *wupper-tools* software can also be used (see Chapter 6.) Based on these *wupper-tools* a software application was made which tests the DMA transfers, to and from the Interlaken (firmware) application.

10.2.5 Hardware test: DMA transfer

With the Wupper-tools, a DMA transfer can be started, to and from the Wupper interface. The software collects information from the registers, and starts the DMA drivers. The Wupper tools, can start a DMA sequence, and the (software part of the) application combines this functionality with the applications' firmware to send data from (the PCIe card wrapper) Wupper, through the application and through Interlaken to the Transceivers optic links. Because these optic fibers are looped-back the same data travels back through Interlaken and the application to Wupper (see Figure 30.) The software application reads the data in the Wupper system at the start of the transaction and at the end of the transaction. If the application is functioning correctly, the data from both instances is the same, minus the fact that there are synchronization words created for the *Interlaken to Wupper* transmission. These synchronization words are generated to preserve the EOP, SOP signals generated from Interlaken, as well as add some additional information (signals that are useful for the debugging of the system.) The data which is written into the Host-PCs' memory is originally generated by the software application.

A screenshot of the output of the software application is shown in Appendix G. The leftmost column shows the packet-number, where a Synchronization word is not counted as a packet, and right column shows the original data which is read out from the Host-PC's memory via Wupper and DMA. The text *synchronization word* is added on the places where the synchronization word is expected, but this is just a print to the console done within the software, the word is not inserted into the memory. The middle column is the data resulting from the entire round-trip (from PC to Wupper to Application, Interlaken and loop-backed to Interlaken, Application, Wupper and back to the PC memory). Since this output (while providing a general idea of the data residing in the PC memory) does not show how the data travels through Interlaken, the hardware test was extended. A component was added to the firmware application, which was used to provide insight into how the different signals react within the firmware, and to readout the FIFOs.

A screenshot of the firmware test wave window output is shown in Appendix F. The *packet_length* register is set to *X"0010"*, which means that after 16 data words, a data transfer will be completed and the *Sync_INFO_Word* will be added. This is signaled by the *send_sync_word* signal. The other signals (such as the start of packet, end of packet, and the FIFO signals) also behaved according to expectations. The data generated from the application software, is saved in the Host PC memory, and from there it is collected by Wupper and sent via the (firmware) application to Interlaken (shown by the signal *Wu_to_TX*.) This same data goes through the Interlaken system (see Figure 30) where it is sent back to Wupper again. It ends up in the *RX_to_WU* signal, which transports it to the *ToHostFIFO* from Wupper.

From the test outputs it can be concluded that the Wupper-Interlaken implementation works as expected.

10.3 Channel bonding on Interlaken

Since the Interlaken implementation (Core1990) is debugged and a version was made which is according to the protocol definition (Core 1990_Version2) it can be further optimized to include other functionalities. One possible optimization that can be added is channel bonding (see Chapter 7.1.) In order to implement this functionality, the Interlaken implementation needs to be extended. The dotted rectangle in Figure 34 shows the part of the total design that will be extended.

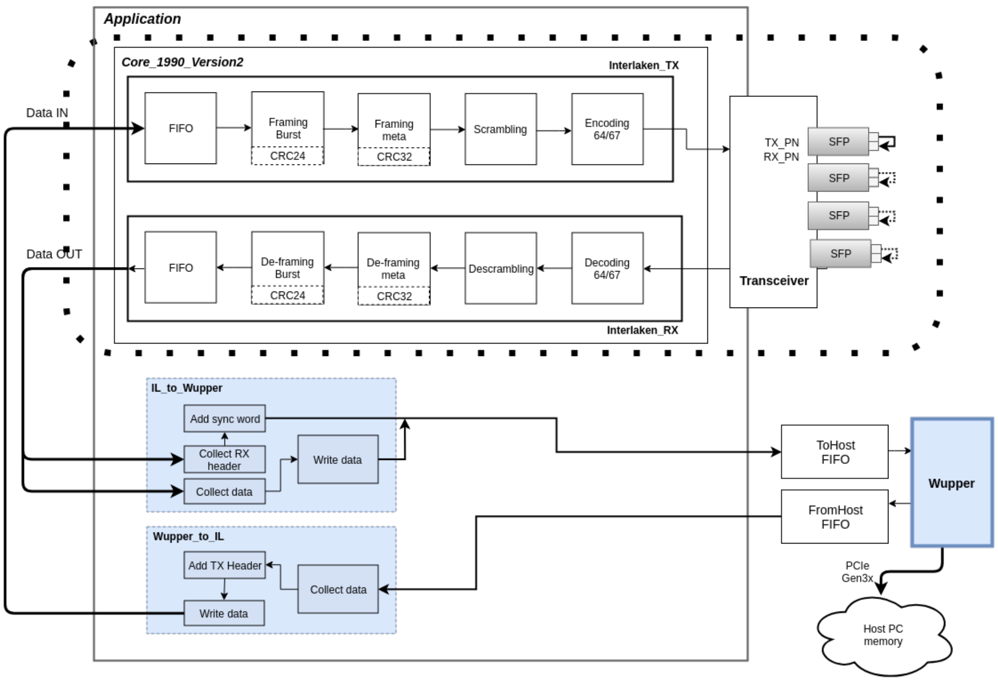


Figure 34: Block diagram with part of the design that will be extended (dotted line.)

It will be replaced by the design shown in Figure 35, which multiplies the *Interlaken_TX* and *Interlaken_Rx* by 4. This is done, because the FELIX hardware has 4 SFP receivers [21], and if the channel bonding implementation succeeds, this will mean that all 4 SFP links can be used.

10.3.1 Extended design

The TX FIFO is extended to 4 times it's width, and the data is distributed across the four lanes of each 69 bits. These bits consist of 64-bits of data, and 5-bits of control signals (SOP, EOP, EOP_valid.) The data is distributed per 64-bit words (data or control), where they are divided among the lanes, beginning with channel 0, ending at channel 3, and repeating for the next chunk of data (see Figure 6 on page 16.) Each Channel has the same layout (so *TX_Channel_0* is the same as *TX_Channel_n* and *RX_Channel_0* is the same as *RX_Channel_n*) At the Transceiver side, each line will be connected to it's own SFP receiver (This is not shown in Figure 34 to improve it's readability.) The SFP fibers are connected in loop-back mode, so the data sent from *TX_Channel_0* will be received at *RX_Channel_0*.

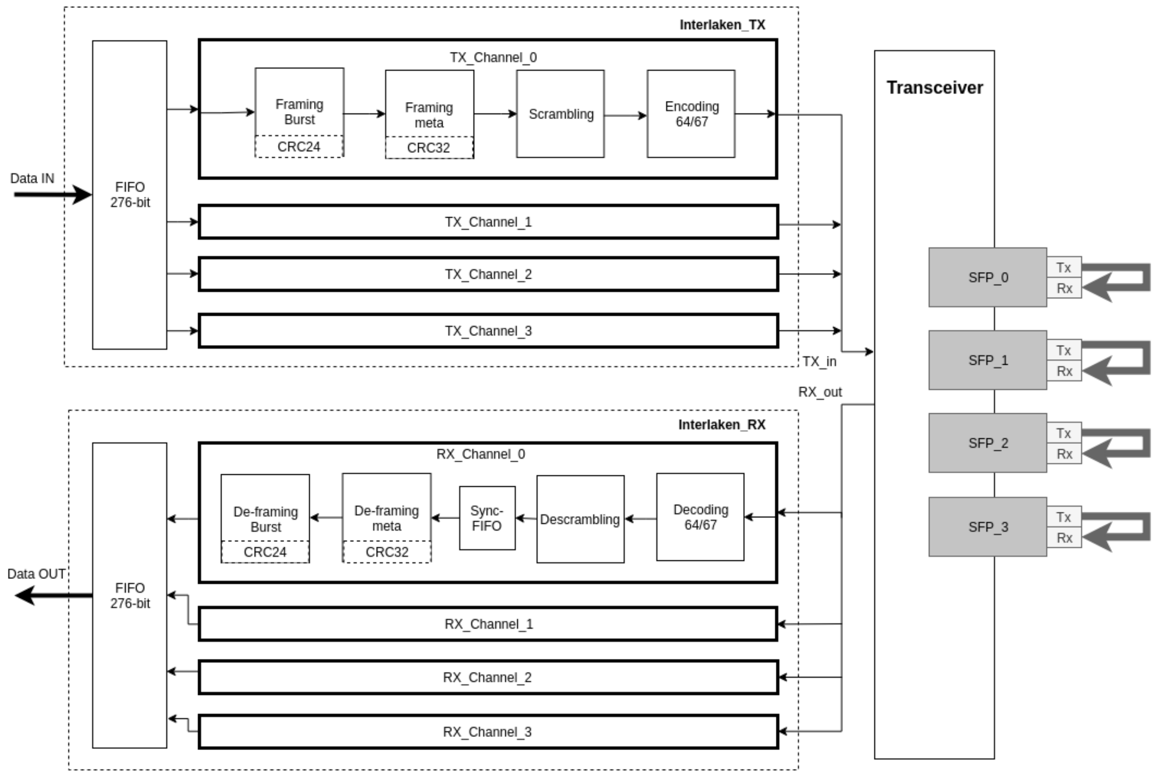


Figure 35: New design for Interlaken where Channel Bonding is implemented.

This effectively multiplies the Interlaken implementation by 4, which means that the maximum transfer speed is also multiplied by four. The Transceiver distribures the data over the four channels, and the data is collected at the FIFO. The *Data_out signal* reads from the FIFO serially, but other ways of reading to multiple channels in parallel can also be implemented so the data can be connected to any other system.

Unfortunately the implementation was not completely finished and tested, but it is recommended that in future iterations of this project the design is implemented as a new version of the Wupper-Interlaken firmware Application.

11 Discussion

Within the assignment, the goal was to implement the Interlaken protocol on the FELIX hardware. In order to do this, a previously made implementation (Core1990) had to be verified. This verification resulted in a several points of discussion, where the implementation did not meet the requirements of the Interlaken protocol definition. These problems were discovered, by connecting another implementation of Interlaken (developed by Xilinx) to the Core1990 implementation, and connecting them so they could communicate with each other. This attempted communication did not work, and several error flags were raised. These errors were debugged, and corrected where possible. Some unexpected results still remained, but these did not undermine the protocol implementation itself. They were thought to be compatibility issues between the Core1990 implementation and the Xilinx implementation, because the Xilinx core uses a parallel data bus (LBUS) format between the transmitter and receiver, which also requires it's own signals and flags. The updated version of the Interlaken implementation (Core1990_V2.0) was functioning according to the definition. This new implementation of Interlaken could then be implemented on the FELIX hardware. To achieve this, an application had to be developed that encompassed the Interlaken implementation, and added a way for the Interlaken data output to be transferred to the Wupper input and vice versa. This application was developed and tested, with success. It did however, leave room for improvement. Some functionalities that were possible within the Interlaken protocol (Channel-bonding and Flow-control) were not yet implemented. After research was done on these optimization possibilities, a design was made to allow for their implementation. The optimization design needs further tests in order to be implemented, it will prove a suitable addition in a future version of the implementation. The addition of multiple data transmission channels, also leaves room for a data bus format to be implemented. This might solve the previously mentioned compatibility issues with the Xilinx Interlaken core.

With the results of this project, an Interlaken implementation was made on the FELIX hardware, which can serve as an alternative to the previous GBT protocol. The optic links can be used to obtain the timing and trigger control information of the ATLAS system (as well as other systems where FELIX is implemented) and the link speed will be raised from 4.8 Gbps to 12.5 Gbps which is a great improvement. Furthermore, a design proposition for an implementation of Channel bonding has been made, which will raise the transfer speed even further. With this research, further developments can be done within the FELIX system, where the Interlaken protocol can possibly be implemented in the next upgrade of the FELIX system on the ATLAS detector.

12 Conclusion

In the next iteration of the Atlas FELIX project ([10],) a more advanced, efficient and faster protocol (than the currently implemented GBT protocol [5]) needs to be implemented. Within this project, research has been done into the Interlaken protocol, and what is needed to implement it on FELIX hardware. Interlaken is a suitable replacement for the GBT protocol currently implemented in FELIX. From previous projects, an Implementation of the Interlaken protocol was made but this implementation was not yet verified. In order to verify that the implementation functions correctly according to the Interlaken protocol definition, a test environment was set up where an Interlaken ip-core developed by Xilinx was connected to the (unverified) implementation. The communication between the two systems did not work, and several errors were raised. The setup was debugged, and the implementation was rewritten where needed.This resulted in a correctly working implementation of Interlaken. This new version of the Interlaken implementation was then ready to be implemented on the FELIX hardware. In order to implement it, the Interlaken implementation needed to be able to function alongside the FELIX firmware that was already present on the FELIX hardware. To solve this problem, a firmware application was developed, which connected the Interlaken system with the Wupper system (FELIX firmware PCIe wrapper, see [18]) so the original data format of the FELIX system was kept. The implementation was tested by creating some additions to the *Wupper-tools* software, which control the driver software of Wupper and FELIX. With the Wupper tools, the data transfers from Interlaken to Wupper to the memory of the FELIX Host PC (and the other way around) could be tested. These tests were done alongside other firmware tests, and were deemed successful. The Interlaken protocol was successfully implemented on the FELIX hardware, and now research could be done on possible optimization methods. From this research, it was concluded that channelbonding and flow control could prove to be a valuable addition to the Implementation. A functional design was developed, which expanded the data transmission channels of Interlaken from one channel to 4 channels. The implementation of this design will further raise the available transmission speed to up to 50Gbps (4 times the link speed of 12.5Gbps.) With the successful implementation of on FELIX and the further optimization options available, the Interlaken protocol is a very usable in the next iteration of the FELIX system.

13 Recommendations on futre work

Since the proposed channelbonding design is still incomplete, the main recommendation would be to finish and implement this design alongside the current Interlaken application on FELIX. The addition of channelbonding will extend the current design to four transmission channels, which will greatly benefit the amount of data that can be processed. The FLX-card will then be able to function with all of it's SFP connectors, which will make it more efficient. With the extention to four data channels, it might also be a worthwile to do some research on parralel transmission formats, and to implement a data bus, similar to the one in the Xilinx Interlaken core (mentioned in Chapter 10.1.3.) This addition of a data bus format, might solve the unknown issues still left with the communication between Core1990_V2.0 and the Xilinx Interlaken core mentioned in Chapter 10.1.3. It is also useful to recreate some of the tests done with the Xilinx Interlaken core, to find out what exactly caused the issues. With the addition of multiple data transmission channels, it would also be useful to add the Flow control functionality. This will allow the Receiver to manage the transmissions on the Transmitter side, by providing feedback and warning for possible congestion from within the control words. Another useful addition would be to include some sort of User Interface to manage the data transfers (and possibly the number of channels once channel bonding is implemented,) and to provide an easier way to change the register values such as packet length.

References

[1] Cortina Systems Incorporated, Cisco SystemIs Incorporated. Interlaken protocol definition. http://www.interlakenalliance.com/Interlaken_Protocol_Definition_v1.2.pdf. [Online; accessed 08-mar-2019].

[2] Cern. Cern Homepage. https://home.cern//. [Online; accessed 19-mar-2019].

[3] Cern. Atlas website. http://atlas.cern/. [Online; accessed 19-mar-2019].

[4] Cern. Felix website. https://atlas-project-felix.web.cern.ch/atlas-project-felix/. [Online; accessed 19-mar-2019].

[5] Cern. GBT architecture. http://cds.cern.ch/record/1091474. [Online; accessed 18-mar-2019].

[6] N.Boukadida. Point-to-point protocol exploration, Jul 2018. Nikhef.

[7] Cern. Lhc. https://home.cern/about/accelerators/large-hadron-collider. [Online; accessed 08-mar-2019].

[8] Cern. What is ATLAS. https://home.cern/science/experiments/atlas. [Online; accessed 18-mar-2019].

[9] Cern. ATLAS Image. https://atlas.cern/sites/atlas-public.web.cern.ch/files/0511013_01-A4-at-144-dpi_0.jpg. [Online; accessed 18-mar-2019].

[10] Nikhef. FELIX information. https://iopscience.iop.org/article/10.1088/1748-0221/11/01/C01055/pdf. [Online; accessed 18-mar-2019].

[11] Nikhef. FELIX: a PCIe based high-throughput approach for interfacing front-end and trigger electronics in the ATLAS Upgrade framework. https://doi.org/10.1088/1748-0221/11/12/C12023. [Online; accessed 18-mar-2019].

[12] G.Lehmann. Felix, the front-end link exchange system (powerpoint). https://slideplayer.com/slide/9777562/. [Online; accessed 19-mar-2019].

[13] OSI, Interfaces: Vertical (Adjacent Layer) Communication. http://www.tcpipguide.com/free/t_InterfacesVerticalAdjacentLayerCommunication.htm. [Online; accessed 11-04-2019].

[14] Integrated interlaken 150g v2.0. https://www.xilinx.com/support/documentation/ip_documentation/interlaken/v2_0/pg169-interlaken.pdf. [Online; accessed 06-05-2019].

[15] Intel. Intel Stratix 10 FPGAs overview. https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html. [Online; accessed 17-jun-2019].

[16] Xlinix. Xlinix evaluatuion board VC707 product page. https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html. [Online; accessed 20-mar-2019].

[17] N.Boukadida. OpenCores: Core1990. https://opencores.org/projects/core1990_interlaken. [Online; accessed 20-mar-2019].

[18] Wupper -a Xilinx Virtex-7 PCIe Engine. http://opencores.org/websvn,filedetails?repname=virtex7_pcie_dma&path=%2Fvirtex7_pcie_dma%2Ftrunk%2Fdocumentation%2Fwupper.pdf. [Online; accessed 17-04-2019].

[19] Virtex-7 FPGA Gen3 Integrated Block for PCI Express (PCIe). https://www.xilinx.com/support/documentation/ip_documentation/pcie3_7x/v3_0/pg023_v7_pcie_gen3.pdf. [Online; accessed 18-04-2019].

[20] Xilinx. Axi Reference Guide. https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf. [Online; accessed 17-04-2019].

[21] Xlinix. Xlinix evaluatuion board VC709 product page. https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html. [Online; accessed 22-may-2019].

[22] Agile Buisiness Consortium. Moscow Method. https://www.agilebusiness.org/content/moscow-prioritisation. [Online; accessed 19-mar-2019].

[23] OpenCores, Nikhef. OpenCores. https://opencores.org/. [Online; accessed 20-mar-2019].

[24] Intel. Altera Synthesis Cookbook, Chapter 12.3. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/stx_cookbook.pdf. [Online; accessed 12-jun-2019].

[25] Xilinx. Vivado Design Suite 2018.1. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2018-1.html. [Online; accessed 20-mar-2019].

[26] Si570 crystal oscilator/voltage controlled oscillator. https://www.silabs.com/documents/public/data-sheets/si570.pdf. [Online; accessed 14-05-2019].

[27] Dspll technoogy. https://www.silabs.com/products/timing/oscillators/dspll. [Online; accessed 14-05-2019].

[28] Si5324 clock multiplier/jitter attenuator. https://www.silabs.com/documents/public/data-sheets/Si5324.pdf. [Online; accessed 14-05-2019].

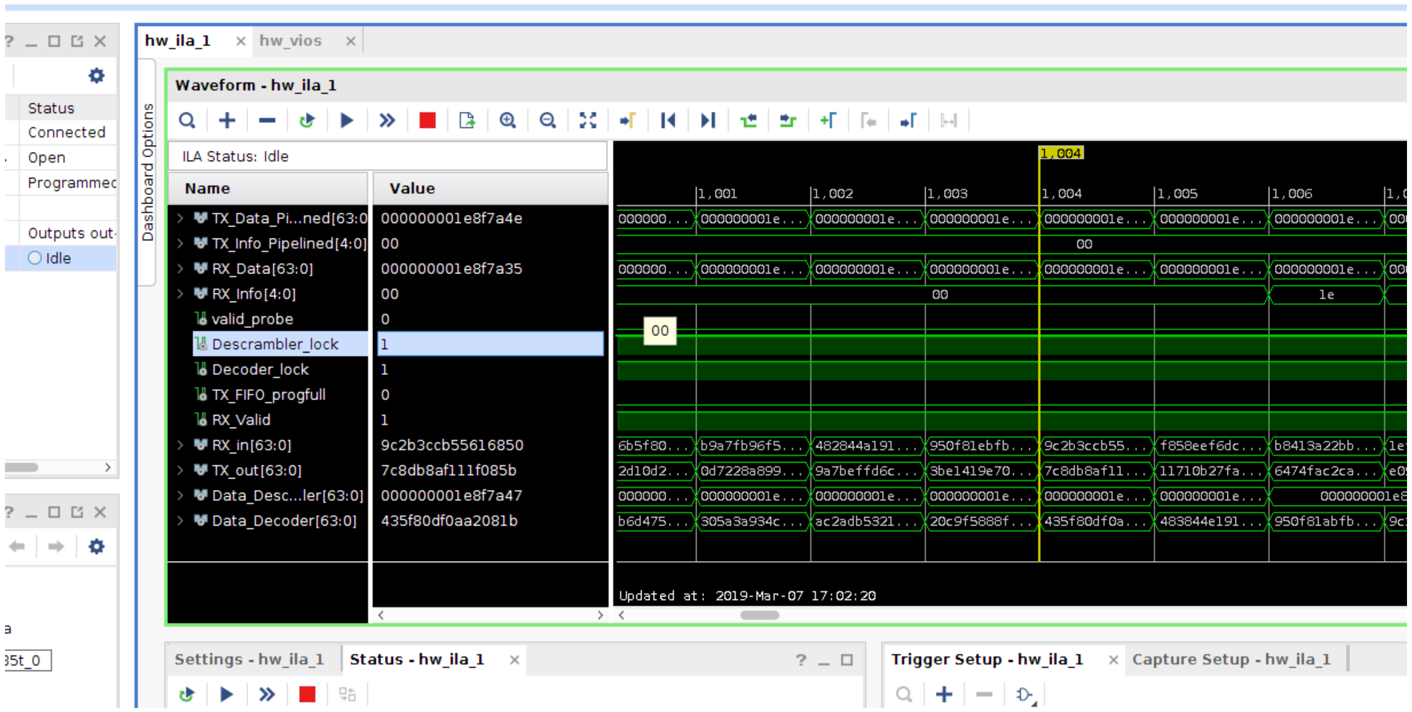
Appendices

A SI5324 register map file.

Below the register map file is shown for the SI5324 Clock jitter attenuator. The meaning of each register setting is explained in [28].

```
#HEADER
# Date: woensdag 1 mei 2019 10:58
# File Version: 3
# Software Name: Precision Clock EVB Software
# Software Version: 5.1
# Software Date: July 23, 2014
# Part number: Si5324
#END_HEADER
#PROFILE
# Name: Si5324
#INPUT
# Name: CKIN
# Channel: 1
# Frequency (MHz): 156,250000
# N3: 79
# Maximum (MHz): 157,500000
# Minimum (MHz): 134,722222
#END_INPUT
#PLL
# Name: PLL
# Frequency (MHz): 5625,000000
# f3 (MHz): 1,977848
# N1_HS: 6
# N2_HS: 9
# N2_LS: 316
# Phase Offset Resolution (ns): 1,06667
# BWSEL_REG Option: Frequency (Hz)
# 10:    7
#  9:   15
#  8:   30
#  7:   60
#  6:  120
#  5:  244
#  4:  502
#END_PLL
#OUTPUT
# Name: CKOUT
# Channel: 1
# Frequency (MHz): 156,250000
# NC1_LS: 6
# CKOUT1 to CKIN1 Ratio: 1 / 1
# Maximum (MHz): 157,500000
# Minimum (MHz): 134,722222
#END_OUTPUT
#CONTROL_FIELD
# Register-based Controls
#        FREE_RUN_EN: 0x0
#    CKOUT_ALWAYS_ON: 0x0
#         BYPASS_REG: 0x0
#          CK_PRIOR2: 0x1
#          CK_PRIOR1: 0x0
#          CKSEL_REG: 0x0
#              DHOLD: 0x0
#            SQ_ICAL: 0x1
#          BWSEL_REG: 0xA
#        AUTOSEL_REG: 0x2
#           HIST_DEL: 0x12
#              ICMOS: 0x3
#              SLEEP: 0x0
#         SFOUT2_REG: 0x5
#         SFOUT1_REG: 0x5
#          FOSREFSEL: 0x2
#             HLOG_2: 0x0
#             HLOG_1: 0x0
#           HIST_AVG: 0x18
#          DSBL2_REG: 0x1
#          DSBL1_REG: 0x0
#             PD_CK2: 0x1
#             PD_CK1: 0x0
#         FLAT_VALID: 0x1
#             FOS_EN: 0x0
#            FOS_THR: 0x1
#            VALTIME: 0x1
#              LOCKT: 0x1
#        CK2_BAD_PIN: 0x1
#        CK1_BAD_PIN: 0x1
#            LOL_PIN: 0x1
#            INT_PIN: 0x0
#         INCDEC_PIN: 0x1
#       CK1_ACTV_PIN: 0x1
#          CKSEL_PIN: 0x1
#        CK_ACTV_POL: 0x1
#         CK_BAD_POL: 0x1
#            LOLPOL: 0x1
#            INT_POL: 0x1
#           LOS2_MSK: 0x1
#           LOS1_MSK: 0x1
#           LOSX_MSK: 0x1
#           FOS2_MSK: 0x1
#           FOS1_MSK: 0x1
#            LOL_MSK: 0x1
#              N1_HS: 0x2
#             NC1_LS: 0x5
#             NC2_LS: 0x5
#              N2_LS: 0x13B
#              N2_HS: 0x5
#                N31: 0x4E
#                N32: 0x4E
#         CLKIN2RATE: 0x0
#         CLKIN1RATE: 0x0
#           FASTLOCK: 0x1
#            LOS1_EN: 0x3
#            LOS2_EN: 0x3
#            FOS1_EN: 0x1
#             FOS2_EN: 0x1
#    INDEPENDENTSKEW1: 0x0
#    INDEPENDENTSKEW2: 0x0
#END_CONTROL_FIELD
#REGISTER_MAP:
{
{  0, 0x14},
{  1, 0xE4},
{  2, 0x82},
{  3, 0x15},
{  4, 0x92},
{  5, 0xED},
{  6, 0x3F},
{  7, 0x29},
{  8, 0x00},
{  9, 0xC0},
{ 10, 0x00},
{ 11, 0x40},
{ 19, 0x29},
{ 20, 0x3E},
{ 21, 0xFE},
{ 22, 0xDF},
{ 23, 0x1F},
{ 24, 0x3F},
{ 25, 0x40},
{ 31, 0x00},
{ 32, 0x00},
{ 33, 0x05},
{ 34, 0x00},
{ 35, 0x00},
{ 36, 0x05},
{ 40, 0xA0},
{ 41, 0x01},
{ 42, 0x3B},
{ 43, 0x00},
{ 44, 0x00},
{ 45, 0x4E},
{ 46, 0x00},
{ 47, 0x00},
{ 48, 0x4E},
{ 55, 0x1B},
{131, 0x1F},
{132, 0x02},
{137, 0x01},
{138, 0x0F},
{139, 0xFF},
{142, 0x00},
{143, 0x00},
{136, 0x40},
};
#END_REGISTER_MAP
#END_PROFILE
```

B Core1990 Hardware test output

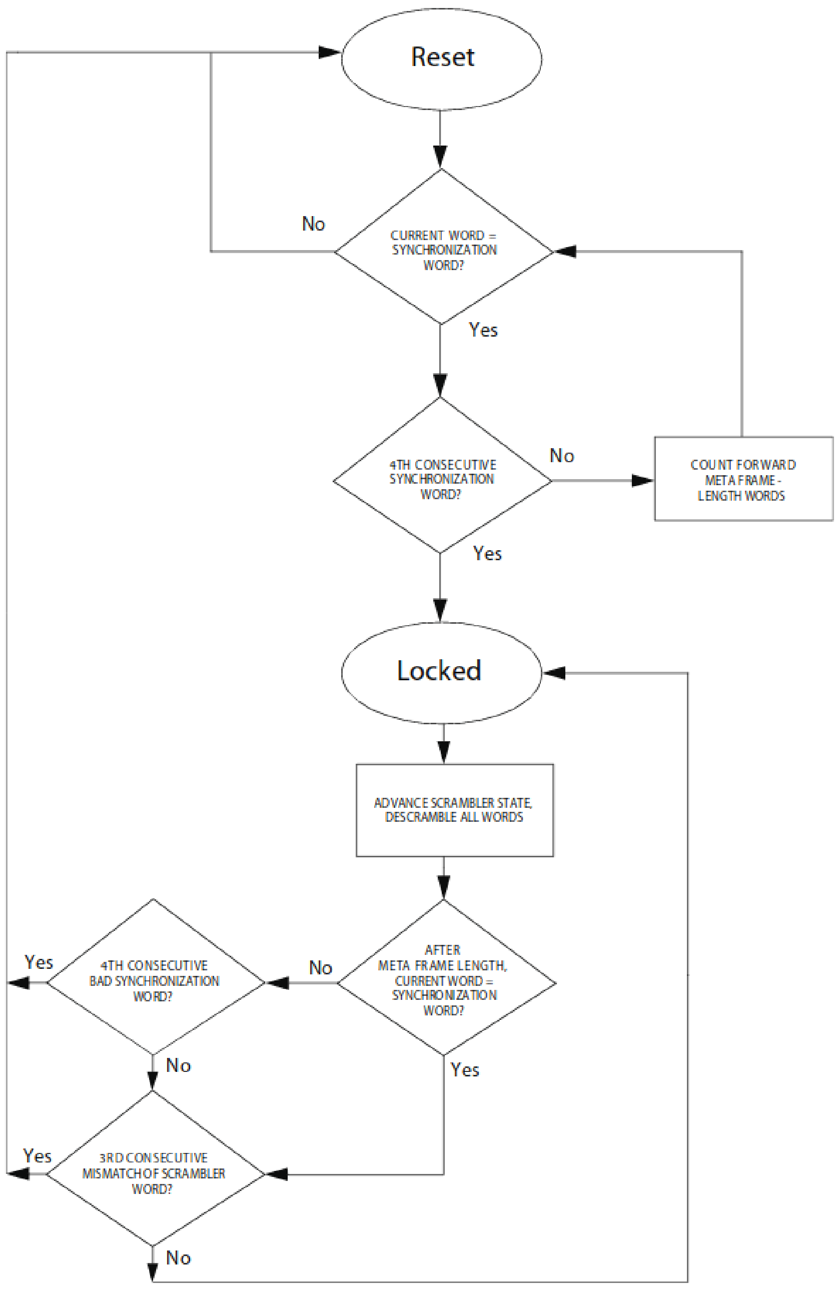

Waveform - hw_ila_1

ILA Status: Idle

```
Name                     Value
TX_Data_Pi...ned[63:0    000000001e8f7a4e
TX_Info_Pipelined[4:0]   00
RX_Data[63:0]            000000001e8f7a35
RX_Info[4:0]             00
valid_probe              0
Descrambler_lock         1
Decoder_lock             1
TX_FIFO_progfull         0
RX_Valid                 1
RX_in[63:0]              9c2b3ccb55616850
TX_out[63:0]             7c8db8af111f085b
Data_Desc...ler[63:0]    000000001e8f7a47
Data_Decoder[63:0]       435f80df0aa2081b
```

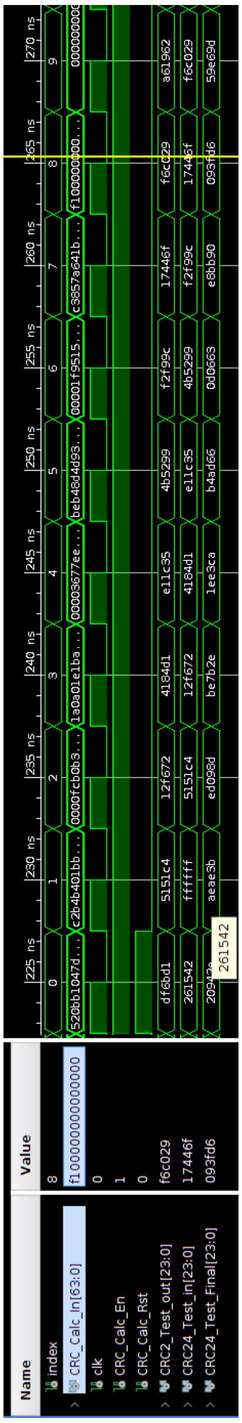
Updated at: 2019-Mar-07 17:02:20

C Scrambler Synchronization State diagram

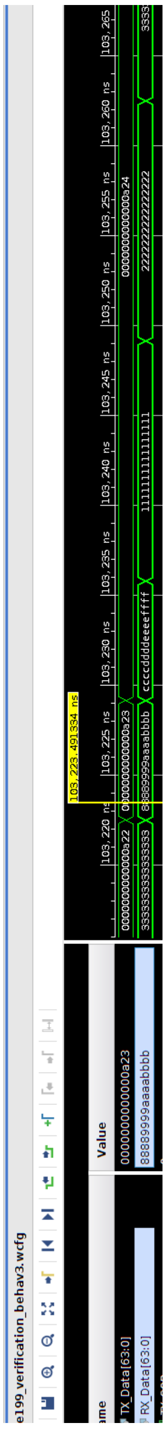

Reset
CURRENT WORD = SYNCHRONIZATION WORD?
No
Yes
4TH CONSECUTIVE SYNCHRONIZATION WORD?
No
COUNT FORWARD META FRAME-LENGTH WORDS
Yes
Locked
ADVANCE SCRAMBLER STATE, DESCRAMBLE ALL WORDS
AFTER META FRAME LENGTH, CURRENT WORD = SYNCHRONIZATION WORD?
No
Yes
4TH CONSECUTIVE BAD SYNCHRONIZATION WORD?
Yes
No
3RD CONSECUTIVE MISMATCH OF SCRAMBLER WORD?
Yes
No

D CRC-24 Check test

In the image below, the wave window is shown for the CRC-24 Test, done with the example data from the Interlaken protocol definition (see Chapter 4.3)



E Output of Core1990 with LBUS simulator



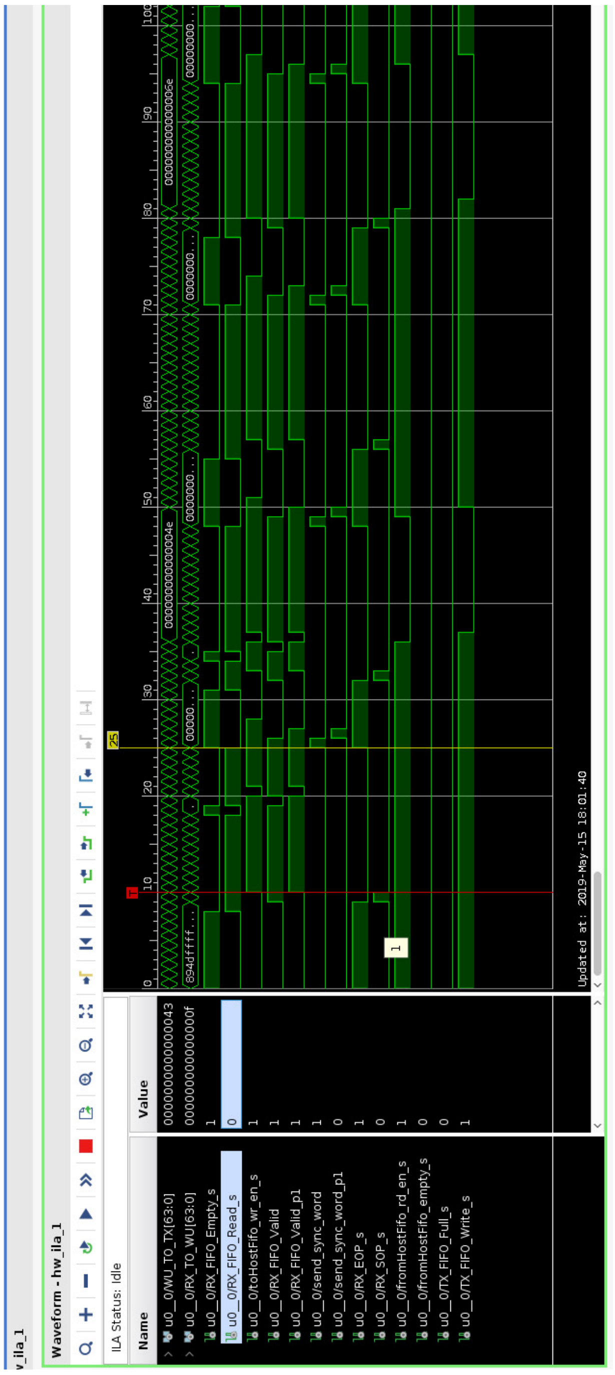
e199_verification_behav3.wcfg

ame	Value
TX_Data[63:0]	000000000000000a23
RX_Data[63:0]	88889999aaaabbbb

TX_Data: 000000000000000a22, 000000000000000a23, 000000000000000a24

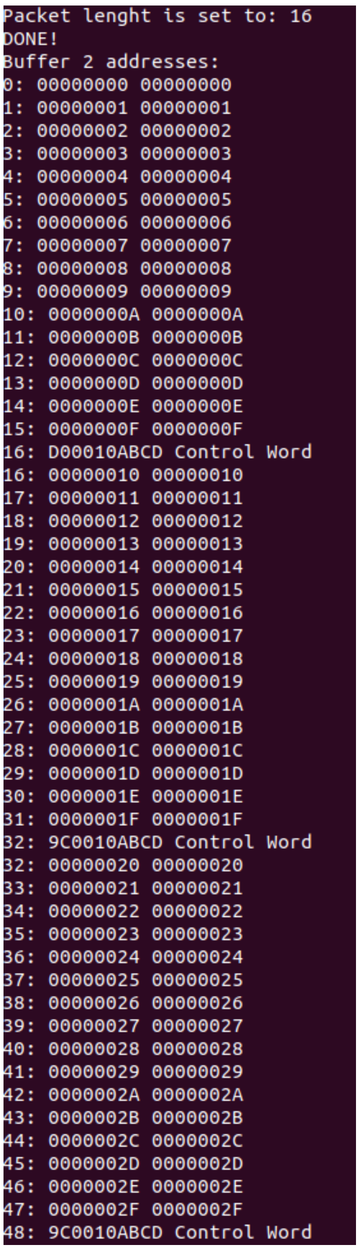
RX_Data: 3333333333333333, 88889999aaaabbbb, ccccddddeeeeffff, 1111111111111111, 2222222222222222, 3333

F Hardware test of Interlaken Application on FELIX



Name	Value
u0__0/WU_TO_TX[63:0]	0000000000000043
u0__0/RX_TO_WU[63:0]	000000000000000f
u0__0/RX_FIFO_Empty_s	1
u0__0/RX_FIFO_Read_s	0
u0__0/toHostFifo_wr_en_s	1
u0__0/RX_FIFO_Valid	1
u0__0/RX_FIFO_Valid_p1	1
u0__0/send_sync_word	1
u0__0/send_sync_word_p1	0
u0__0/RX_EOP_s	1
u0__0/RX_SOP_s	0
u0__0/fromHostFifo_rd_en_s	1
u0__0/fromHostFifo_empty_s	0
u0__0/TX_FIFO_Full_s	0
u0__0/TX_FIFO_Write_s	1

G Wupper-Interlaken Application DMA transfer output.



```
Packet lenght is set to: 16
DONE!
Buffer 2 addresses:
0: 00000000 00000000
1: 00000001 00000001
2: 00000002 00000002
3: 00000003 00000003
4: 00000004 00000004
5: 00000005 00000005
6: 00000006 00000006
7: 00000007 00000007
8: 00000008 00000008
9: 00000009 00000009
10: 0000000A 0000000A
11: 0000000B 0000000B
12: 0000000C 0000000C
13: 0000000D 0000000D
14: 0000000E 0000000E
15: 0000000F 0000000F
16: D00010ABCD Control Word
16: 00000010 00000010
17: 00000011 00000011
18: 00000012 00000012
19: 00000013 00000013
20: 00000014 00000014
21: 00000015 00000015
22: 00000016 00000016
23: 00000017 00000017
24: 00000018 00000018
25: 00000019 00000019
26: 0000001A 0000001A
27: 0000001B 0000001B
28: 0000001C 0000001C
29: 0000001D 0000001D
30: 0000001E 0000001E
31: 0000001F 0000001F
32: 9C0010ABCD Control Word
32: 00000020 00000020
33: 00000021 00000021
34: 00000022 00000022
35: 00000023 00000023
36: 00000024 00000024
37: 00000025 00000025
38: 00000026 00000026
39: 00000027 00000027
40: 00000028 00000028
41: 00000029 00000029
42: 0000002A 0000002A
43: 0000002B 0000002B
44: 0000002C 0000002C
45: 0000002D 0000002D
46: 0000002E 0000002E
47: 0000002F 0000002F
48: 9C0010ABCD Control Word
```

Figure 36: Application output.

H WUPPER register map

Starting from the offset address of BAR0, BAR1 and BAR2, the register map for BAR0 expands from 0x0000 to 0x0430 for the PCIe control registers. BAR0 only contains registers associated with DMA. The offset for BAR0 is usually 0xFBB00000.

Address	PCIe	Name/Field		Bits	Type	Description
Bar0						
DMA_DESC						
0x0000	0,1	DMA_DESC_0				
			END_ADDRESS	127:64	W	End Address
			START_ADDRESS	63:0	W	Start Address
0x0010	0,1	DMA_DESC_0a				
			RD_POINTER	127:64	W	PC Read Pointer
			WRAP_AROUND	12	W	Wrap around
			READ_WRITE	11	W	1: fromHost/ 0: toHost
			NUM_WORDS	10:0	W	Number of 32 bit words
...						
0x00E0	0,1	DMA_DESC_7				
			END_ADDRESS	127:64	W	End Address
			START_ADDRESS	63:0	W	Start Address
0x00F0	0,1	DMA_DESC_7a				
			RD_POINTER	127:64	W	PC Read Pointer
			WRAP_AROUND	12	W	Wrap around
			READ_WRITE	11	W	1: fromHost/ 0: toHost
			NUM_WORDS	10:0	W	Number of 32 bit words
DMA_DESC_STATUS						
0x0200	0,1	DMA_DESC_STATUS_0				
			EVEN_PC	66	R	Even address cycle PC
			EVEN_DMA	65	R	Even address cycle DMA
			DESC_DONE	64	R	Descriptor Done
			CURRENT_ADDRESS	63:0	R	Current Address
...						
0x0270	0,1	DMA_DESC_STATUS_7				
			EVEN_PC	66	R	Even address cycle PC
			EVEN_DMA	65	R	Even address cycle DMA
			DESC_DONE	64	R	Descriptor Done
			CURRENT_ADDRESS	63:0	R	Current Address

Address	PCIe	Name/Field		Bits	Type	Description
0x0300	0,1	BAR0_VALUE		31:0	R	Copy of BAR0 offset reg.
0x0310	0,1	BAR1_VALUE		31:0	R	Copy of BAR1 offset reg.
0x0320	0,1	BAR2_VALUE		31:0	R	Copy of BAR2 offset reg.
0x0400	0,1	DMA_DESC_ENABLE		7:0	W	Enable descriptors 7:0. One bit per descriptor. Cleared when Descriptor is handled.
0x0410	0,1	DMA_FIFO_FLUSH		any	T	Flush (reset). Any write clears the DMA Main output FIFO
0x0420	0,1	DMA_RESET		any	T	Reset Wupper Core (DMA Controller FSMs)
0x0430	0,1	SOFT_RESET		any	T	Global Software Reset. Any write resets applications, e.g. the Central Router.
0x0440	0,1	REGISTER_RESET		any	T	Resets the register map to default values. Any write triggers this reset.
0x0450	0,1	FROMHOST_FULL_THRESH				
			THRESHOLD_ASSERT	22:16	W	Assert value of the FromHost programmable full flag
			THRESHOLD_NEGATE	6:0	W	Negate value of the FromHost programmalbe full flag
0x0460	0,1	TOHOST_FULL_THRESH				
			THRESHOLD_ASSERT	27:16	W	Assert value of the ToHost programmable full flag
			THRESHOLD_NEGATE	11:0	W	Negate value of the ToHost programmalbe full flag

Table 4: FELIX register map BAR0

BAR1 stores registers associated with the Interrupt vector. The offset for BAR1 is usually 0xFBA00000.

Address	PCIe	Name/Field		Bits	Type	Description
Bar1						
INT_VEC						
0x0000	0,1	INT_VEC_0				
			INT_CTRL	127:96	W	Interrupt Control
			INT_DATA	95:64	W	Interrupt Data
			INT_ADDRESS	64:0	W	Interrupt Address
. . .						
0x0070	0,1	INT_VEC_7				
			INT_CTRL	127:96	W	Interrupt Control
			INT_DATA	95:64	W	Interrupt Data
			INT_ADDRESS	64:0	W	Interrupt Address
0x0100	0,1	INT_TAB_ENABLE		7:0	W	Interrupt Table enable Selectively enable Interrupts

Table 5: FELIX register map BAR1

BAR2 stores registers for the control and monitor of HDL modules inside the FPGA other than Wupper. A portion of this register map's section is dedicated for control and monitor of devices outside the FPGA; as for example simple SPI and I2C devices. The offset for BAR2 is usually 0xFB900000.

Address	PCIe	Name/Field		Bits	Type	Description
Bar2						
Generic Board Information						
0x0000	0	REG_MAP_VERSION		15:0	R	Register Map Version, 1.0 formatted as 0x0100
0x0010	0	BOARD_ID_TIMESTAMP		39:0	R	Board ID Date / Time in BCD format YYMMDDhhmm
0x0020	0	BOARD_ID_SVN		15:0	R	Board ID SVN Revision
0x0030	0	STATUS_LEDS		7:0	W	Board GPIO Leds
0x0040	0	GENERIC_CONSTANTS				
			INTERRUPTS	15:8	R	Number of Interrupts
			DESCRIPTORS	7:0	R	Number of Descriptors

Address	PCIe	Name/Field		Bits	Type	Description
0x0050	0	CARD_TYPE		63:0	R	Card Type: * 709 (0x2c5) VC709 * 710 (0x2c6) HTG710 * 711 (0x2c7) BNL711
			Application Specific			
0x1000	0,1	LFSR_SEED_0		63:0	W	Least significant 64 bits of the LFSR seed
0x1010	0,1	LFSR_SEED_1		63:0	W	Bits 127 downto 64 of the LFSR seed
0x1020	0,1	LFSR_SEED_2		63:0	W	Bits 191 downto 128 of the LFSR seed
0x1030	0,1	LFSR_SEED_3		63:0	W	Bits 255 downto 192 of the LFSR seed
0x1040	0,1	APP_MUX		0:0	W	Switch between multiplier or LFSR. * 0 LFSR * 1 Loopback
0x1050	0,1	LFSR_LOAD_SEED		any	T	Writing any value to this register triggers the LFSR module to reset to the LFSR_SEED value
0x1060	0,1	APP_ENABLE		0:0	W	1 Enables LFSR module or Loopback (depending on APP_MUX) 0 disable application
			House Keeping Controls And Monitors			
0x2300	0	MMCM_MAIN_PLL_LOCK		0	R	Main MMCM PLL Lock Status
0x2310	0	I2C_WR				
			I2C_WREN	any	T	Any write to this register triggers an I2C read or write sequence
			I2C_FULL	25	R	I2C FIFO full
			WRITE_2BYTES	24	W	Write two bytes
			DATA_BYTE2	23:16	W	Data byte 2
			DATA_BYTE1	15:8	W	Data byte 1
			SLAVE_ADDRESS	7:1	W	Slave address

Address	PCIe	Name/Field		Bits	Type	Description
			READ_NOT_WRITE	0	W	READ/¡o¿WRITE¡/o¿
0x2320	0	I2C_RD				
			I2C_RDEN	any	T	Any write to this register pops the last I2C data from the FIFO
			I2C_EMPTY	8	R	I2C FIFO Empty
			I2C_DOUT	7:0	R	I2C READ Data
0x2330	0	FPGA_CORE_TEMP		11:0	R	XADC temperature monitor for the FPGA CORE for Virtex7 temp (C)= ((FPGA_CORE_TEMP* 503.975)/4096)- 273.15 for Kintex Ultrascale temp (C)= ((FPGA_CORE_TEMP* 502.9098)/4096)- 273.8195
0x2340	0	FPGA_CORE_VCCINT		11:0	R	XADC voltage measurement VCCINT = (FPGA_CORE_VCCINT *3.0)/4096
0x2350	0	FPGA_CORE_VCCAUX		11:0	R	XADC voltage measurement VCCAUX = (FPGA_CORE_VCCAUX *3.0)/4096
0x2360	0	FPGA_CORE_VCCBRAM		11:0	R	XADC voltage measurement VCCBRAM = (FPGA_CORE_VCCBRAM *3.0)/4096
0x2370	0,1	FPGA_DNA		63:0	R	Unique identifier of the FPGA
0x2800	0	INT_TEST_4		any	T	Fire a test MSIx interrupt #4
0x2810	0	INT_TEST_5		any	T	Fire a test MSIx interrupt #5
			Wishbone			
0x4000	0	WISHBONE_CONTROL				

Address	PCIe	Name/Field		Bits	Type	Description
			WRITE_NOT_READ	32	W	wishbone write command wishbone read command
			ADDRESS	31:0	W	Slave address for Wishbone bus
0x4010	0	WISHBONE_WRITE				
			WRITE_ENABLE	any	T	Any write to this register triggers a write to the Wupper to Wishbone fifo
			FULL	32	R	Wishbone
			DATA	31:0	W	Wishbone
0x4020	0	WISHBONE_READ				
			READ_ENABLE	any	T	Any write to this register triggers a read from the Wishbone to Wupper fifo
			EMPTY	32	R	Indicates that the Wishbone to Wupper fifo is empty
			DATA	31:0	R	Wishbone read data
0x4030	0	WISHBONE_STATUS				
			INT	4	R	interrupt
			RETRY	3	R	Interface is not ready to accept data cycle should be retried
			STALL	2	R	When pipelined mode slave can't accept additional transactions in its queue
			ACKNOWLEDGE	1	R	Indicates the termination of a normal bus cycle
			ERROR	0	R	Address not mapped by the crossbar

Table 6: FELIX register map BAR2

The following Registers are added with the addition of the Interlaken protocol to Wupper. An Application was made, encompassing Interlaken and a connection block, which transfers the data into the *FromHostFIFO* and reads data from the *FromHostFIFO*.

Interlaken						
Address	**PCIe**	**Name/Field**		**Bits**	**Type**	**Description**
0x5000	0	INTERLAKEN_PACKET_LENGTH		15:0	W	Packet length for fromhost packet (to Interlaken)
0x5010	0	INTERLAKEN_CONTROL_STATUS				
			TRANSCEIVER_RESET	any	T	Any write to this register triggers a transceiver reset
			DECODER_LOCK	1	R	Decoder lock indication
			DESCRAMBLER_LOCK	0	R	Descrambler lock indication
0x5020	0	TRANSCEIVER				
			LOOPBACK	8	W	Interlaken
			TX_FAULT	7:4	R	SFP transceiver TX fault indication
			RX_LOS	3:0	R	Loss of signal indication

Table 7: FELIX register map Interlaken (Application.)