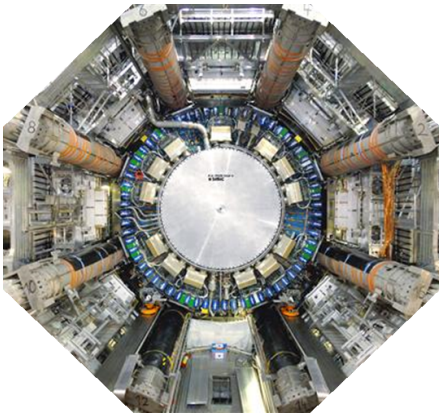
Hogeschool van Amsterdam
Amsterdam University of Applied Sciences

Nikhef



The Atlas Detector

FELIX-Card Firmware
The Interlaken Protocol

Abstract

NIKHEF the Dutch National Institute for Subatomic Physics, works together with CERN National Institute for Nuclear Research, on the ATLAS detector of the Large Hadron Collider. The ATLAS detector is used to detect and discover new particles and their physics. In the process the detector generates huge amount of data that needs processing. For this task NIKHEF is developing the Front End LInk EXchange or in short FELIX to be used as the data aquisition system. One of the goals FELIX tries to achieve is providing a very high transmission speed. In 2016 FELIX used theGBT protocol and reached a 4.8 Gb/s link speed. In 2018 new protocols were researched and the Interlaken Protocol was chosen to be the new succesor. In 2019 a Single lane version of Interlaken had been achieved with a transmission speed around the 10.0 Gb/s. In 2020 the new steps were taken in designing FELIX. With the use of a whole new FPGA and a Multi-lane version of Interlaken the transmission speed could go as far as four lanes of each 25 Gb/s, so the main goal was set to 100 Gb/s. To achieve this transmission speed the Single lane version of Interlaken got cleaned and optimized.AXI-Stream got implemented to standardize the data exchange interface, and work was done on implementing a form of Channel-Bonding, which will allow four lanes to work parallel together. In the limited time of this project Channel-Bonding wasn't finished, and the 100 Gb/s weren't met. Yet a lot of work is done working towards the 100 Gb/s which are on the AXI-Stream Interface, the optimization steps, parts of Channel-Bonding and a Transceiver used for testing. It's advised to continue with Channel-Bonding and perform a different variety of simulations to secure the working of Interlaken according to specification.

Interlaken Protocol	
Version:	1.0
Created:	February 26, 2020
Last modified:	June 23, 2020

Prepared by:	**Checked by:**	**Approved by:**
Myron Mooij Education: Intelligent Devices and Sensors	F.Schreuder (NIKHEF) W.E.Dolman (HVA)	Myron Mooij

Abstract (Dutch)

NIKHEF, het Nederlands Nationale Intstituut voor Subatomaire Fysica, werkt samen met CERN, Nationale Instituut voor Nucleair Onderzoek, aan de ATLAS detector van de Large Hadron Collider. De ATLAS detector wordt gebruikt in zoektocht naar nieuwe deeltjes en hun fysica. Tijdens het gebruik van de detector wordt er een enorme hoeveelheid aan data gegenereerd die verwerkt moet gaan worden. NIKHEF ontwikkeldt hiervoor de Front End LInk EXchange, ook wel verkort naar FELIX, om te gebruiken als data acquisitie systeem. Als een van zijn doelen probeert FELIX een hele hoge transmissie snelheid te bieden. In 2016 gebruikte FELIX het GBT protocol and haalde hiermee een snelheid van 4.8 Gb/s. In 2018 zijn er nieuwe protocollen onderzocht en het Interlaken Protocol werd gekozen. In 2019 was er een versie van Interlaken ontwikkeld die een enkel kanaal gebruikt om een transmissie snelheid te halen van ongeveer 10 Gb/s. In 2020 worden nieuwe stappen genomen in het ontwerp van FELIX. Door een nieuwe FPGA te gaan gebruiken en Interlaken te veranderen van een enkel kanaal naar meerdere kanalen zou de transmissie snelheid verhoogt kunnen worden naar 25 Gb/s per kanaal, hierom werd het doel gezet op 100 Gb/s. Om deze snelheid te bereiken is de versie van Interlaken werkend met een enkel kanaal opgeschoond en geoptimaliseerd. Vervolgens is er een AXI-Stream Interface ge-implementeerd om als een gestandaardiseerd data interface te gebruiken. Ook is er werk verricht om Channel-Bonding te implementeren, die ervoor gaat zorgen dat vier kanalen samen kunnen werken. Door de gelimiteerde duur van het project, is de Channel-Bonding niet afgekomen en is het doel om 100 Gb/s te bereiken niet gehaald. Toch is er een hoop gerealiseerd die toe werkt naar de 100 Gb/s waaronder het AXI-Stream Interface, de optimalisaties, delen van Channel-Bonding en ook nog een Transceiver om mee te testen. Het advies is om het Channel-Bonding af te maken en verschillende simulaties uit te voeren om de werking van Interlaken, volgens specificatie, the garanderen.

TABLE OF CONTENTS

PREFACE

In the 3rd year of Electrical Engineering, Intelligent devices and Sensors, at the Amsterdam University of Applied Science (HVA) an internship is needed to continue to the 4th year of education. When applying at NIKHEF an internship was agreed, for the period of 03-02-2020 till 03-07-2020 to continue the Interlaken project which is part of the ATLAS project. The intern has two supervisors, each representing its organization. From school the supervisor is W.E.Dolman and the supervisor at NIKHEF is F.Schreuder. To successfully succeed in this internship the certain requirements must be met. HVA demands 100 days of 8 hours of work at the chosen company, in addition certain competences (that are already agreed) must be developed and described in a self-reflecting paper. For NIKHEF there will be worked on the Interlaken Project and make progress with the project. In the planning chapter F, the Interlaken Project components of this document are stated.

SPECIAL THANKS

During the internship I've gained a lot of experience in the field of VHDL and FPGA's. Still there is much more to learn and I would like to thank F.Schreuder for his mentoring, help and fun as a colleague. Not only did he help me with the project but also in having fun at work/school, being part of the ET-Department and learning a bit of ping pong. I would also like to thank him for his help in connecting with people for possible future endeavor.

This year also had a strange and unique period due to the Corona virus. Working at home wasn't as difficult with my internship, because the work mainly consists of writing software and scripts. But instead I got sick and want to thank W.E.Dolman with is support and helping me with all my questions and for his effort and concerns during this period.

And I want to thank the whole ET-Department at NIKHEF, for having a great time. I have learned a lot from all of you. And shall miss and remember this great internship at NIKHEF!

1

INTRODUCTION

This chapter will give a brief introduction on ATLAS, FELIX and the Interlaken Project.

1.1 CONTEXT

NIKHEF and CERN are working on ATLAS experiment. In 1992 they first started and in 2012 they had managed to find measured proof of the existence of the Higgs particle. The ATLAS detector is used for many things. (read Appendix A) Al the measurements generate data. This data needs to be processed, so it can be made accessible for complex analysis. One of the sub-projects that help with the work of the ATLAS detector is FELIX (Front-End LInk eXchange). FELIX is designed to have a very high transmission speed and is sensor independent. At the department of Electronic Technology (ET), NIKHEF, the firmware is developed to create and shape FELIX. Down here a brief introduction of FELIX from 2019:

> *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 several optic links. These optical 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 100Gb/s. From the PC the data will be sent further, over a generic high-performance computer network.* [1]

1.2 POINT TO POINT PROTOCOL

To achieve these high speeds, a capable protocol needed to be found for FELIX. (See Appendix B) In 2018 the intern N. Boukadida researched possible protocols concerning Point to Point Protocols and how they could be implemented. In his research, N. Boukadida found the Interlaken Protocol.[2]

1.3 CORE1990 V1

After broad research of the Interlaken protocol (Appendix E) vs other Point to Point Protocols the first steps were taken to start implementing this protocol. This created Core1990.(Also in Appendix E) Named after a name on a t-shirt of N. Boukadida.[2] The Interlaken Protocol takes a couple of steps and encodes data with a Scrambler. This encoded data can be interpreted (decoded) by any other device. Interlaken allows reliable point to point data transfers.

1.4 CORE1990 V2

The Core1990 version wasn't yet fully implemented till 2019. In this year another intern L. Verwoert went on with the Interlaken Project. L.Verwoert worked on Core1990 V2 and finished a functioning 1 channel Interlaken protocol.[1] In her research about optimizing Core1990 V2, two things were noted. First, Channel-Bonding could be used to speed up Core1990 V2, as the speed would increase with each channel that is used. Second, an AXI-Stream Interface could provide a general way of data transfer and can still be used

when multichannel is available. In her last week she managed to make a general design of how the Channel-Bonding could be implemented, but there was no time to implement the design.

Next will be Chapter 2 about the main goal. It's recommended to read Appendix F because it shows the original project planning.

2

THE MAIN GOAL

Now in 2020 new steps will be made. The main goal will be to improve the performance of Core1990 V2, mainly in transmission speed. This could be achieved by implementing Channel-Bonding (Appendix C) and an AXI-Stream Interface (Appendix D). A less important thing that also could affect things, is the optimization of previous work and possibly redesign parts of the Firmware that's found to be in-efficient. Apart from the previously mentioned things some smaller things will also need work. In 2020 the FELIX card will change its current FPGA to a newer and better suitable one. The new FPGA (VU37P) would allow to bring the channel data rate from 10 Gb/s to 25 Gb/s. This will also mean that Core1990 V2 needs FPGA dependent code remapped. The main goal will be creating a version of Interlaken that uses four lanes to reach a tranmission speed of 4*25 Gb/s resulting in 100 Gb/s.

2.1 SUB GOALS

In addition to the main goal, sub goals will be achieved to work towards the main goal. The first sub-goal is to optimize all VHDL descriptions and cleanup unused code. This will mainly consist of deleting unused lines and old comments, simplify complex logic and rewriting State-Machines. Another one is to adjust the existing Tcl scripts and when necessary write new ones, to create a Dynamic Work Environment. The third one is changing the current used FIFO's in the Framing and De-Framing Burst to AXI-Stream FIFO's and create an AXI-Stream Interface, as the AXI-Stream Interface is part of the future planning of FELIX. Another goal is replacing all old test-benches by making a general test-bench to test Interlaken as a whole. And the last sub-goal is creating logic for Channel-Bonding. The idea is that all these goals participate in bringing the Interlaken project of the FELIX card steps forward towards (the main goal) the 100 Gb/s transmission speed.

2.1.1 THE MAIN GOAL IN ONE LINE

The main goal is to develop Interlaken with AXI-Stream and Channel-Bonding and reach a transmission speed of 100 Gb/s.

2.2 DOCUMENT LAY-OUT

To find answers and work towards the main goal and sub goals not only the chapter and sources are used, but also the appendices. Next will be a short introduction to what to expect in the appendices, after that the normal chapters will be discussed.

2.2.1 APPENDIX LAY-OUT

The Appendices are meant to give an in depth introduction in specific parts of the project and are recommended to be read first if you are new to this project. The Appendices have the following content.

Appendix A and Appendix B, will describe the context of the ATLAS project and FELIX, to give a general introduction on both projects. Then Appendices E, C and D are used to research and find answers for the sub-goals and main goal. The F Appendix contains the original planning, preparations and ideas that were concluded out of the research plan. In Appendix G most of the specifications are collected in one place, for a quick survey. In Appendix H each of the main modules/entities of Interlaken are described. And the last

Appendix I contains the terms, definitions and a glossary.

2.2.2 CHAPTER LAY-OUT

The content of the normal Chapters are selected to represent the following three important sub-goals: AXI-Stream Interface, Transceiver and Channel-Bonding. Some of the sub-goals are achieved in the Appendices. A good example is Appendix H which contains information about the many entities used in Interlaken and how they work and were changed. The reason for it to be an Appendix is because it has some overlap with the other chosen content for the chapters.

The upcoming Chapters 3.2.1, 4 and 5 will each contain the sections specification, implementation and testing. Then after three of these Chapters a Conclusion Chapter (6) and a Recommendation Chapter (7) will contain the Conclusion and Recommendation in one place. Now it's time to move on to The AXI-Stream Interface.

3

The AXI-Stream Interface

The first thing that is implemented (after optimizing the code see Appendix F and Appendix H for more detail) is the AXI-Stream Interface. The AXI-Stream Interface is a standard interface, that allows 2 devices to exchange data. A Handshake is used to secure a quick data transfer between a Slave and Master device. Next to data, the AXI-Stream provides an additional signal called Tuser. The Tuser signal is used (in our Interlaken implementation) for transferring internal Error Status from Slave to Master. These statuses contain information on the data integrity during transfer with Interlaken. For more information about AXI-Stream in general, read Appendix D. The AXI-Stream conclusion and advise chapter can be found in Chapter 6 Conclusion and Chapter, 7 This chapter will continue with the AXI-Stream already in context of Interlaken.

3.1 Specification

Core1990 V2 worked with an input and output FIFO. These FIFO's were not standard through the complete ATLAS project. The main disadvantage is that every device that communicates with a device containing different FIFO's, will need extra work to make it properly work. From this situation the idea of a general interface arose. In the end the AXI-Stream Interface seemed the suitable choice, as it allows for a high speed interface. AXI-Stream is now the standard in the FELIX project and needs to be implemented with the following conditions. The Data Stream mode will be used. This will remove the signal TSTRB, as no Position Bytes will be used. Because FELIX will work on high transmission speeds the signals An optional signal of the AXI-Stream and TDEST are removed. They would allow for multiple Data Stream paths in a single lane, which is slower as all data will come back together at the end of the lane and only one Data Stream path can be handled at a time. For secure communication the default Handshake will be used. In more specific specifications of the AXI-Stream for implementation can be found in Appendix D.

3.2 Implementation

The AXI-Stream implementation is spread out across three modules. Firstly an AXI-Stream FIFO was used to replace the old FIFO, how this is done will be discussed in the upcoming Chapter 3.2.1. The main reason to use and keep a FIFO is to hold data. When Interlaken is busy processing Data into Frames it cannot always assumed to be Ready in time for a next Frame. Hence the FIFO will act as a Buffer when Interlaken is slower than the input Data Rate. Interlaken in turn could try to catch up, for example, in the moments that there isn't any data available.

To make the FIFO work there is a need to communicate with External Devices, otherwise there won't be any data for the FIFO. This means that the FIFO interface needs to be connected to the data input and output of Interlaken. This means that the AXI-Stream Interface implementation will also be implemented in the the Framing Burst (Chapter 3.2.2) and the De-Framing Burst (Chapter 3.2.3).

3.2.1 AXI-Stream FIFO

In Figure 3.1 the currently used AXI-Stream FIFO is shown. On the left the signal s-axis can be found. This is the Slave side of the interface. Only the m-axis-tready (also on the left side) is passed back to the Master device, hence the prefix m, to let the Master device know the FIFO is ready. On the right side these signals are flipped from Slave to Master and the Master to Slave. All signals that are shown with no wire going in or out are signals currently unused. They are set to be always a default value, for example to zero or one.

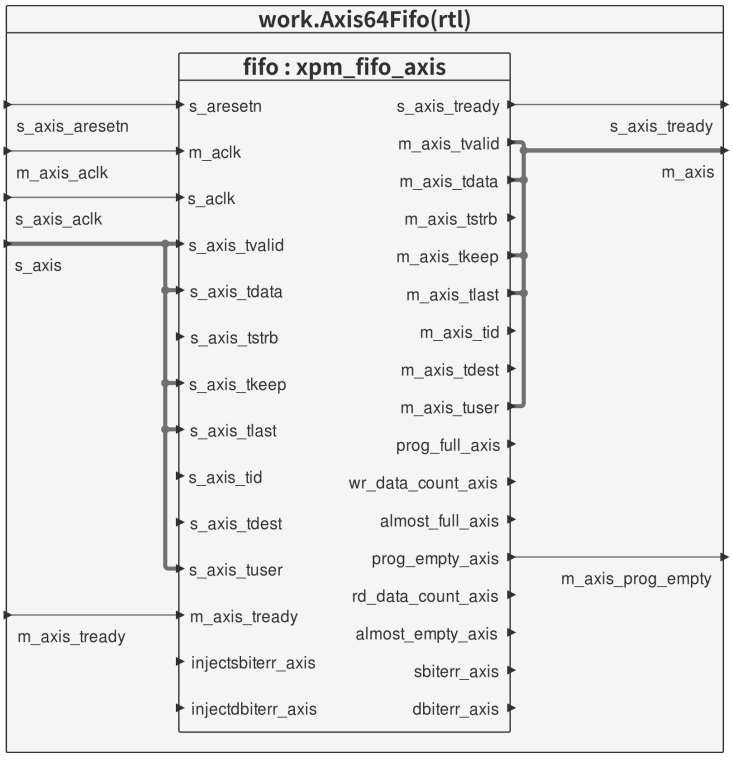


Figure 3.1: Module view of the AXI-Stream FIFO

The FIFO is built from the XPM-FIFO-AXIS template. Also it makes use of the so called AXI-Stream-Package in which the Record Types that are used to store multiple logic and Logic Vectors into the same s-axis and m-axis Array. This is a clean way to keep the signals together as a bundle. The FIFO is for now only implemented to work with 64 Bit Frames. The FIFO depth can be assigned and is currently set to 2048 Bits. There is also the m-axis-prog-empty signal. This signal is used to flag the Multiplexer that a Data Block is Ready.

3.2.2 TX AXI-Stream Interface

The Framing Burst is the first module on the TX-side. The Framing Burst uses incoming data from an External Device. To interface with the external devices it will also need an AXI-Stream Interface. In Table 3.1 the signals of the interface are listed and a short description of how the signals are used.

S-Axis Signals	*Description*
tdata	*tdata [63:0] is the data that needs to be transferred by Interlaken*
tkeep	*The tkeep signal [7:0] got a own process to convert the tkeep into at three bit signal passed on to the framing-burst format bytes [59:57]*
tlast	*The tlast is a logic signal that is used to signal the EOP, the framing-burst uses this as an EOP indicator*
tuser	*The tuser signal [3:0] contains own chosen information. For this version they are the following: Truncation[3], Flowcontrol [2],Framing-error[1], CRC-error[0]*
tready	*tready is a logic signal that flags to the master that the slave is ready to receive data.*

Table 3.1: TX AXI-Stream Interface Table

In the next Chapter 3.2.3 the same Table is made for the RX-side. (Table 3.2).

In Figure 3.2 the Transmitter-MultiChannel module is shown. On the left you can find the inputs of the Transmitter. In the middle the AXIS64-FIFO can be found. Next to it the Interlaken Transmitter is shown containing the Framing Burst, Meta-framing, Encoder and Scrambler. And on the right the outputs of the Transmitter-MultiChannel.

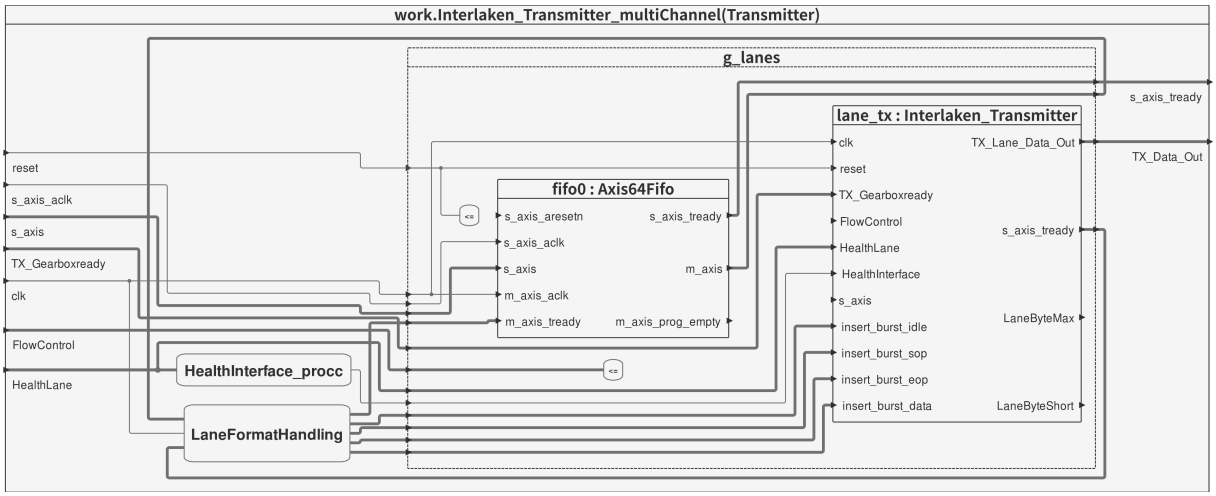


Figure 3.2: Module view of the Interlaken Transmitter MultiChannel

The specifics of the Interlaken Transmitter can be found in Appendix H. The next chapter will continue with the AXI-Stream Interface on the Receiver Side of Interlaken.

3.2.3 RX AXI-Stream Interface

At the Receiver Side of Interlaken the last module is the De-Framing Burst. The De-Framing Burst needs the AXI-Stream Interface to communicate the Data Stream onto another device. In Table 3.2 an overview is shown of how the AXI-Stream signals are used in the De-Framing Burst.

M-Axis Signals	*Description*
tdata	*tdata [63:0] is the data that transferred through interlaken.*
tkeep	*The De-Framing-Burst got an own process to convert the framing burst-format [59:57] bits to the tkeep signal [7:0] bits*
tlast	*When an EOP is found the tlast is set HIGH*
tuser	*The tuser signal [3:0] collects own chosen information. For this version they are the following: Truncation[3], Flowcontrol [2],Framing-error[1], CRC-error[0]*
tready	*tready is a logic signal that flags that the slave is ready to receive data*

Table 3.2: RX AXI-Stream Interface Table

The big different between the signals in this Table and the Table 3.2 from previous chapter is that the signals are changed from s-axis to m-axis.

In Figure 3.3 the Interlaken Receiver MultiChannel is shown. The Multichannel is mainly used to make the single lane version of Core1990 dynamic scale-able to n lanes, by re-using four times the same logic next to each-other. (The Multichannel is part of the Channel-Bonding, more in Chapter 5)

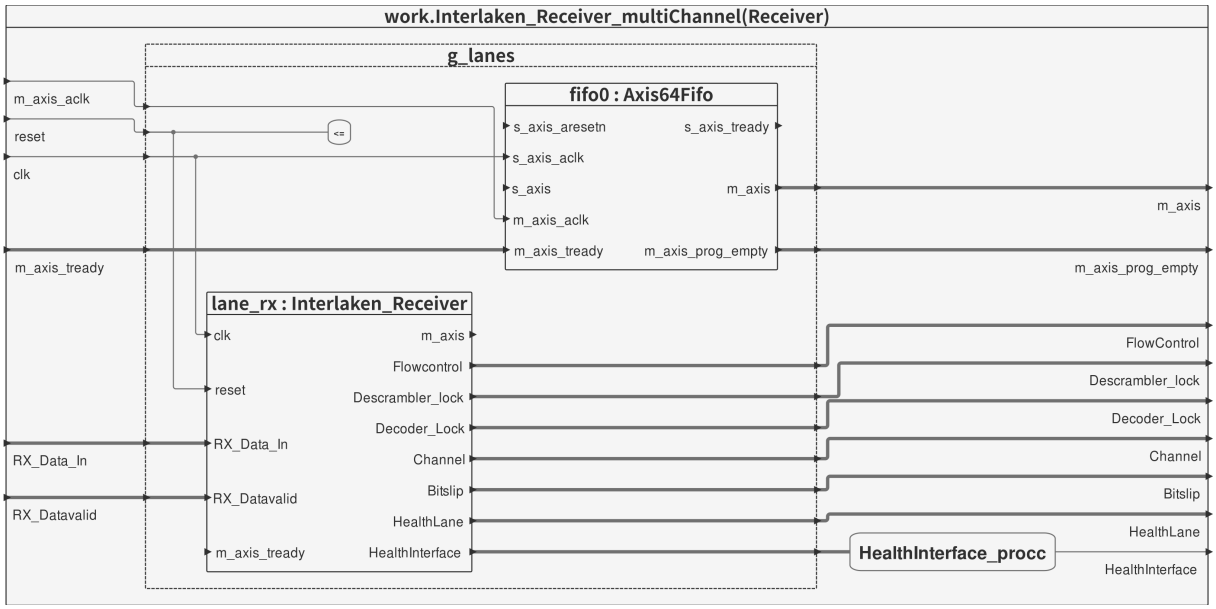


Figure 3.3: Module view of the Interlaken Receiver MultiChannel

In the Figure the Interlaken Receiver is shown and the AXIS64-FIFO. The FIFO is exactly the same FIFO used at the TX-side. The Interlaken Receiver contains the logic for a single-lane which are De-Framing Burst, De-Framing Meta, the Descrambler and the Decoder. The Lane Logic is discussed in Appendix H.

3.3 Testing AXI-Stream

After cleaning and optimizing the Core1990 version of Interlaken, the implementation of the AXI-Stream was next in order before Channel-Bonding. The implementation contained new logic and adjustments to the Framing Burst and De-Framing Burst, Small changes to the RX and Transmitter-MultiChannel and the creation of an AXI-Stream FIFO.

3.3.1 Test-bench

To test the AXI-Stream all the AXI-Stream signals were used in a test-bench. In Figure 3.4 a part of the test-bench process is shown to use as an example. The first thing to notice is the use of many AXI Signals. The process starts by setting certain AXI Interface signals and waits for a Reset. After the Reset the interface waits till all lanes are Aligned. These can either be through the HealthInterface Signal or the Stat-rx-aligned. But which one is used depends on how Interlaken is connected. In this test Interlaken was connected in the following order: Interlaken TX, Interlaken Transceiver, Interlaken RX. With this setup The HealthInterface Signal from the Interlaken RX will be used. This signal is used to flag if all lanes are locked and fully initialized.

When the lanes are ready the test-bench continues to set the right interface signals, to signal the data is Valid, and begins to make Packages. The Packages part is not fully shown in figure 3.4. Luckily the process is fairly simple to explain. The test-bench uses a For Loop to loop Packages, original these were 256 Packages just counting from 1 to 256. But in this version this already changed to making small Packages by cutting the packages in smaller ones (when count reaches 10), using the Tlast signal. For more information on how the data is split, transferred to separate lanes and how the Tlast is used with multiple lanes read chapter 5.2.2 .

```
Simulation_Framing_Burst : process
    variable n: integer := 0;
    variable count: integer := 0;
    begin
        m_axis_tready <= (others => '1');

        --Test Data patern 1 to FFFFFF
        for i in 0 to Lanes-1 loop
            s_axis(i).tvalid <= '0';
            s_axis(i).tlast <='0';
        end loop;

        wait until (Reset = '0');

        if LOOPBACK  then
            wait until (HealthInterface = '1'); -- Wait for lock before sending
        else
            wait until (stat_rx_aligned = '1'); -- @suppress "Dead code"
        end if;
        wait for DCLK_PERIOD;

        for i in 0 to Lanes-1 loop
            s_axis(i).tvalid <= '1';
            s_axis(i).tkeep <= (others => '0');
            s_axis(i).tuser <= (others => '0');
        end loop;

        for packet in 0 to 256 loop -- Send 256 packets of 1 to 256
            count := 0;
            while (count<10) loop
                for i in 0 to Lanes-1 loop
                   s_axis(i).tlast <='0';
               end loop;
```

Figure 3.4: Partial simulation code axistream

3.3.2 THE TESTED PARTS

This test-bench contained a fairly simple AXI-Stream interaction which allowed to seek out errors in the implementation. The first main goal of the test was to get Data-in trough Interlaken to Data-out.This was achieved by sending the Packages and worked on the first go and was a signal that the AXI-Stream FIFO could handle the Data Stream. Next added was the waiting for reset, at the begin of the process. This was used to test the working of the Framing Burst interface, by communicating to wait till the Master was Ready. This also worked on the first go. The last part was adding the Tlast. The Tlast was more difficult to test. Because of the Multi-lane that wasn't implemented yet the Tlast could only be manual triggered on chosen lanes which allowed for an EOP, but not a correct SOP on the right lane after. But with this data-throughput test most parts of the AXI-Stream interface were tested.

3.3.3 TEST-BENCH CONNECTIONS

To clarify the setup, in Figure 3.5 the two setups of the test-bench are shown. On the Right G-loopback determines if Interlaken gets looped-back into itself. If so the test-bench is connected in the way said earlier, if not the test-bench connects the TX-side of Interlaken to the RX-side of the Interlaken150G wrapper. This wrapper is used to connect to a Xilinx core with a full Interlaken protocol implemented. In this way there is the possibility to test if the TX-side works according to Interlaken specification. This setup will be used during Multichannel testing. (See appendix 5) This is all about AXI-Stream for now. In Chapter 6 you can find the conclusion and in Chapter 7 the recommendations. Next Chapter will be about the Transceiver necessary to reach a lane transmission speed of 25 Gb/s.

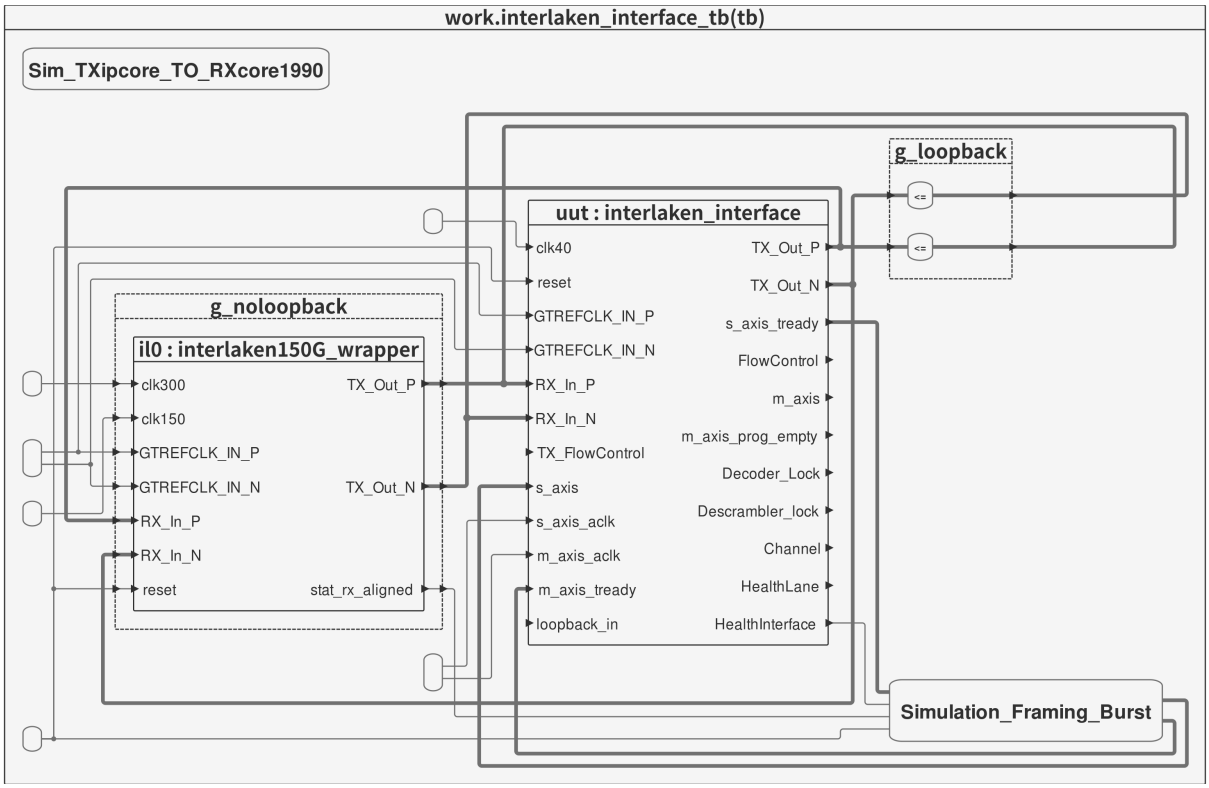


Figure 3.5: Module view of the Interlaken Interface Test Bench

4

TRANSCEIVER

The RX and TX lanes of Interlaken are both connected to a Transceiver. This Transceiver allows to connect other devices to this Interlaken device through the Transceivers LBUS. In the Core1990 V2 version of Interlaken the following Transceiver (from Xilinx) is used: Transceiver-10g-64b67b. This Transceiver can get to 10 Gb/s per channel, when using the 150MHz clock. This Transceiver will be replaced to a Transceiver containing an 400MHz clock and faster logic, that would allow to reach a channel speed up to 25 Gb/s.

4.1 SPECIFICATION

In the research of N.Boukadida [2] point to point protocols were researched to determine the best suitable protocol for FELIX. Because of the requirements shown in Figure 4.1 Interlaken seemed the best suitable protocol as it met all the requirements.

Requirement	*Description*
Line-rate	For FELIX a line-rate of 10Gb/s was specified in 2018
Range distance coverage	Range distance coverage of 10 to 200m, which corresponds to normal distances covered by cables in data centers
Forward error correction	To be prepared for noisy environments
Flow control	Allows for dynamic change of the transmission speed
Cyclic Redundancy check	CRC-check to control data integrity
Channel-Bonding	Bonded channels allow for an increase in bandwidth

Table 4.1: Requirement of the Point to Point Protocol (2018) [2]

4.1.1 CHANGES AND ADDITIONS

Now in the new version of Interlaken another Transceiver will be used. The Transceiver of the VCU128-ES1 supports Interlaken up to 25 Gb/s a channel. This is because the Virtex Ultrascale + has faster logic. As addition on the mentioned specifications the use of the IP-Core brings mandatory specifications listed in the datasheet, see source [3].

4.2 Implementation

For the implementation of the Transceiver, a Transceiver core is generated in Vivado. This is done by selecting the Interlaken150G IP-Core and configuring the core through the IP-core wizard. After these steps an instantiation template is available. From this template the port map is copied into a VHDL file. Here the Interlaken150G IP-Core gets an entity and architecture. Because of the many signals of the Transceiver it is chosen tho make this a wrapper. This means that the entity contains only signals we need and all the unused signals are defined open or zero (LOW). Because of the use of a wrapper the Test Bench containing this entity will be more clean. The file and entity are called Interlaken150G wrapper and the component is called Interlaken 0. In Figure 4.1 the Interlaken150G IP-Core is shown.

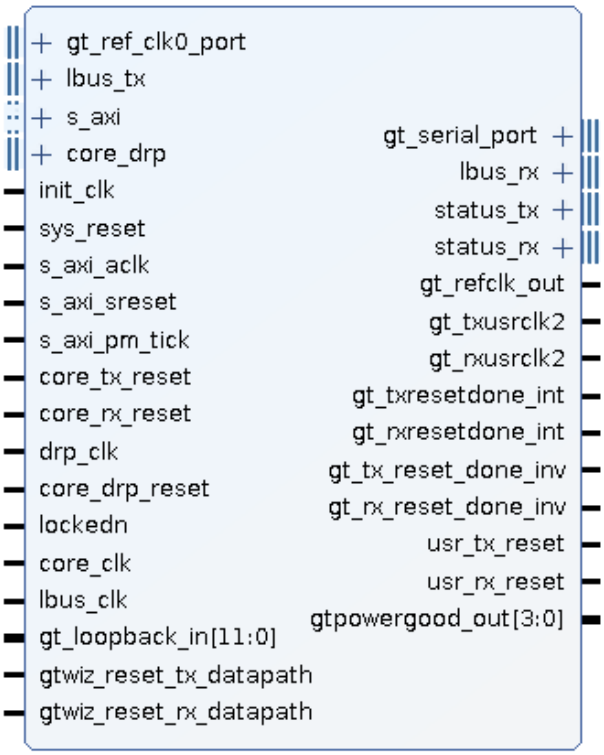


Figure 4.1: Interlaken150G IP-Core module block (Vivado IP-core wizard).

4.2.1 Features and setup

The Interlaken150G IP-Core is currently generated to expect four lanes with a 64/67 bit layout. The core allows input and output through the Lbus or through a AXI4-Light Interface. This AXI4-Light Interface is a bit more extensive than the AXI4-Light implemented in Interlaken, mainly because it uses a few extra signals to function. For this reason the AXI4-Light Interface is not used. This leaves the TX-lbus and the RX-lbus for in and output. Source [4] contains more information about different available signals.

4.3 Test Results

The Transceiver implementation was mainly used to test if Interlaken still works according to specification after all the Changes, and when Channel-Bonding is done it could be used to reach the 100 Gb/s. Testing this device is therefore linked to the tests done in the Channel-Bonding Chapter 5. In the test-bench the TX-side of Interlaken gets connected to the RX-side of the Interlaken150G IP-Core. The IP-Core itself has many error-flag signals which are in turn used to test if Interlaken150G IP-Core itself is functioning and to find out if something is wrong with the Channel-Bonding.

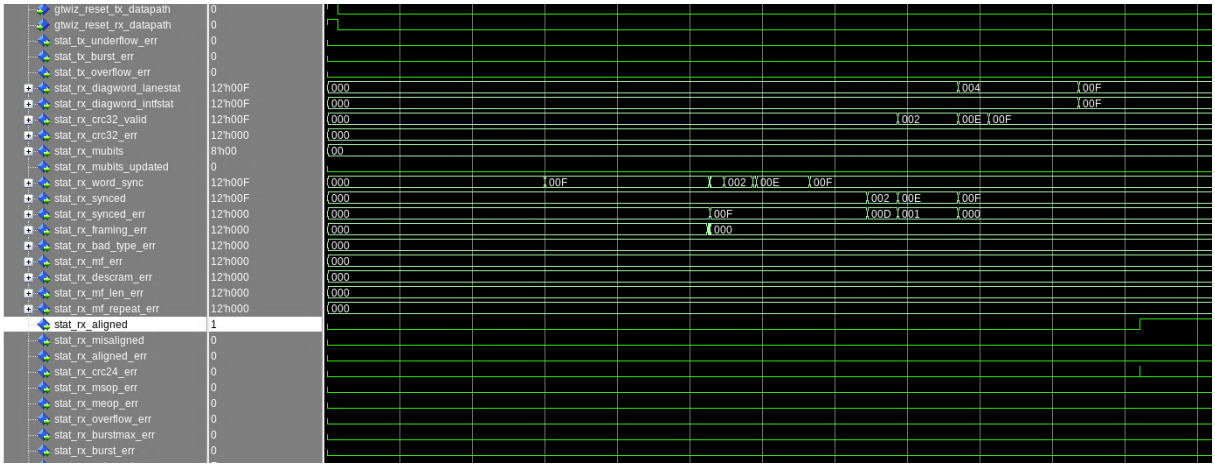


Figure 4.2: Interlaken150G IP-Core Error signals Simulation.

4.3.1 Initialization Error Signals

In Figure 4.2 an overview of the many error signals is shown, that are available in the IP-Core. The text is a bit small, but isn't necessary to understand what's happening. The signals above the selected signal (the signal on the left that is marked in white), are signals that flag errors during the initialization of the Interlaken150G IP-Core. One important signal is the rx-synced signal. This signal indicates that the IP-Core has its Lane synced, which indicates that the sync words of the TX-Meta-Frames are found and recognized, along with the other meta frame words.When the IP-Core is fully initialized and in sync the rx-aligned signal, the white marked signal in Figure 4.2, is flagged to HIGH, and the IP-Core is ready to receive a Data Stream.

4.3.2 Data Stream Error Signals

The signals below the white signal, are used to flag errors during the Data Stream. In the current version there is a CRC24 Error at the first Frame. When looked closely it's visible on the right three lanes down from the white marked signal. An example of other error signals in the same row are Framing Burst Error, Meta-Framing Error or Burstmax Error. Which each flag if a frame isn't according to the expected Interlaken format

According to all previously discussed signals, the Test Bench indicates, the TX-side performs almost according to Interlaken specification, but there is another issue when testing with the Interlaken150G IP-Core. The data goes into the IP-Core but never comes out of the IP-Core. There is not yet an explanation why this happens, as the IP-Core is lend from Xilinx and has many signals. It is probable that some settings are preventing the output from showing, or that the core waits on a slave device that wants data.

The Conclusions from this Chapter can be found in Chapter 6 and the Recommendation in Chapter 7. The upcoming Chapter will be about Channel-Bonding.

5

CHANNEL-BONDING

The main goal is reaching the channel transmission speed of 25 Gb/s. To achieve this the current used FPGA will be changed and current work must be optimized. Channel-Bonding can help FELIX to get up to a transmission speed of 100 Gb/s. This is done by using 4 lanes parallel in a Synced state. For more general information about Channel-Bonding see appendix C. This chapter will now continue with Channel-Bonding and how it's used in Interlaken.

5.1 SPECIFICATION

Channel-Bonding itself got some specifications that need to be fulfilled for it to operate correctly. In table 5.1 an overview of the most important specifications are shown.

Specification	*Description*
Dynamic Scalable Code	*In the most abstract form, Channel-Bonding can be used to bond n-amount of channels. For Interlaken a minimum of 4 are required.*
Optimized for 100 Gb/s	*The Channel-Bonding needs not only to be available, it will act as the main part of Interlaken in reaching and using an 100 Gb/s transmission speed*
Independent of Packet-length	*Interlaken and the used transceiver allow for multiple different packet-lengths. To keep this option open, the packet length must be taken into account. Optional: use same Packet-length range available by the transceiver*
Lane Interaction	*To maintain the correct Interlaken format, there must be a certain lane interaction. Such an interaction must prevent lanes from sending multiple burst, control or meta words that have already been send by another lane.*

Table 5.1: Channel-Bonding Specifications

5.1.1 SCALABLE CODE

In the table four specs are shown. These four are plenty for now as they tell much about a possible implementation. First the spec 'Dynamic Scalable Code'. It's not yet determined in Interlaken how much channels will be used. The quantity of lanes will most probable change when Interlakenis used differently. This mean that when wanted Interlaken should keep working when set for example to seven lanes.

5.1.2 OPTIMIZED AND PACKET LENGTH

The main goal is to reach 100 Gb/s hence the second spec. The third spec is about the Packet-length. The Packet-length determines how much frames will be in a Single Packet. Different devices may use different packet-lengths, but when they interact with each-other they need to be the same, or it will miss-match. So to keep Interlaken flexible, the Packet-length needs to be flexible.

5.1.3 Lane Interaction

The last one is the Lane Interaction. It may be regarded more as an implementation feature than a spec, but nonetheless it's something that is very important to choose before hand, hence it's a spec. One way or the other the Channel-Bonding must be managed and controlled. This could be either done by some sort of regulator that talks to the lanes and collects their statuses, to determine the lane actions. This could be a big impact on speed as there are a lot of steps and lanes have to wait on instructions. Another path that comes to mind is Lane Interaction. Lane Interaction could allow for lane statuses to be known directly by other lanes. This could be faster than the regulator idea.

Now a little more is known about the specifications it's time to look at how Channel-Bonding will be implemented.

5.2 Implementation

Before there will be dived into the implementation it's necessary to explain some parts of the structure of Interlaken to know where to start and what the environment looks like. When Core1990 V2 was created Interlaken worked as a Single lane protocol. To expand to four lanes a new dimension was created. For the Transmitter this was the Transmitter-MultiChannel and for the Receiver the Receiver-MultiChannel. These Multichannels are used to instantiate an N-channel amount of Transmitters or receivers. Because the Multichannel dimension is used to make the channels, it is also a good place to lay-out the Channel-Bonding implementation, as this is the place that knows which channels exist.

5.2.1 The Data Format

Knowing now where the implementation will start choices need to be made. First take the TX-side. When connected to an extern device data will first enter the Framing Burst and AXI-Stream FIFO. Then the Framing Burst must determine what to sends. This can be Data-words, Idle-words, SOP or an EOP. In a single lane situation this would mean that the lane send the following format: an SOP, then it fills the Packet with Data-words and Idle-words and ends the packet with an EOP.

5.2.2 Maintaining data format during Channel-Bonding

When multichannels are used this will slightly change. The format will stay the same, but will be divided across multiple channels. In Table 5.1 a simple version is shown as example.

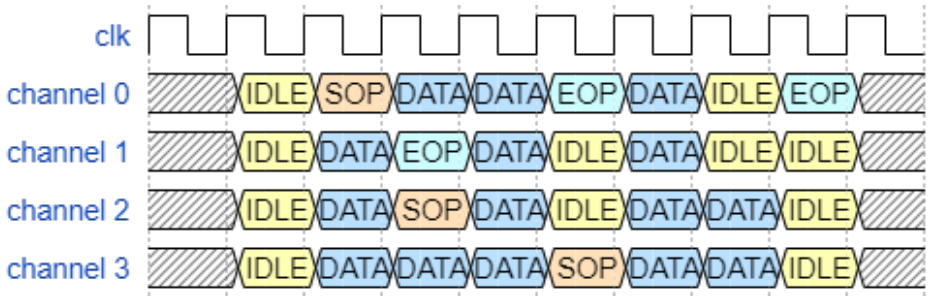


Figure 5.1: Channel-Bonding format example

Left the channel numbers are shown. First let's look at the column 'format-set 1' Channel one frames a SOP, at the same time the other three lanes frame data. In the 'format-set 2' column channel one frames the last data frame and channel two then finishes the Packet with an EOP. Then channel three needs to start

a new Packet, as the stream keeps going. When no data is available Idle-words are framed. The EOP on channel four could be triggered by an Tlast, marking an End of packet.

With this format in mind and the signal Tlast the following things need to be managed by the Channel-Bonding process(es): SOP, EOP, DATA, IDLE and Tlast, as each of these things will be distributed across the lanes. The first steps are shown in figure 5.2.

```
112  EOP_and_IDLE_Handling: process(axis, axis_tready_transmitter)
113      variable tlast_found : boolean;
114   begin
115      axis_tready_process <= axis_tready_transmitter;
116      tlast_found := false;
117      insert_burst_idle <= (others => '0');
118      for i in 0 to Lanes-2 loop
119          if axis(i).tlast = '1' and axis(i).tvalid = '1' then
120              tlast_found := true;
121          end if;
122          if tlast_found then
123              axis_tready_process(i+1) <= '0';
124              insert_burst_idle(i+1) <= '1';
125          end if;
126      end loop;
127  end process EOP_and_IDLE_Handling;
128
129  SOP_Handling: process(clk)
130   begin
131      if rising_edge(clk) then
132          insert_burst_sop <= (others => '0');
133          for i in 0 to Lanes-1 loop
134              if axis(i).tlast = '1' and axis(i).tvalid = '1' then
135                  insert_burst_sop(0) <= '1';
136                  exit;
137              end if;
138          end loop;
139      end if;
140  end process SOP_Handling;
```

Figure 5.2: First steps of Channel-Bonding Code

5.2.3 EOP, IDLE AND SOP HANDLING

The figure shows two processes. The first one is for EOP and IDLE handling and the second one for SOP handling. The EOP and IDLE handling use the Tlast from any channel to start an EOP. When Tlast stays HIGH, it inserts a burst idle. The burst idle means that a channel needs to send Idle-words. The other process handles the SOP. When a Tlast has happened, an EOP was sent, so the next Frame must be an SOP according to the format. The SOP handling communicates this through the insert burst SOP signal to the Framing Burst.

Not only the Transmitter-MultiChannel code was changed for this implementation, also the Transmitter and Framing Burst code got some slight adjustments to use and interact with the two processes.

5.3 Test Results

Before more code is written it's time to test the previously made changes. A simulation is used to test the workings of the Framing Burst. In the used testbench the Transmitter is connected to the Receiver of an Interlaken IP-Core. The Interlaken IP-Core has the Interlaken protocol running according to the Interlaken specifications. Connecting the Transmitter this way allows for a quicker way to check if the Transmitter works according to specifications as the IP-Core Receiver has a lot of monitoring signals available.

5.3.1 Test Bench

The Test Bench itself is very basic. First it waits for the complete initialization by waiting for the All lanes aligned signal of the IP-Core. When aligned data is sent on all channels. And with a for loop all lanes receive a Tlast to end a packet, for now this is after 20 data-frames.

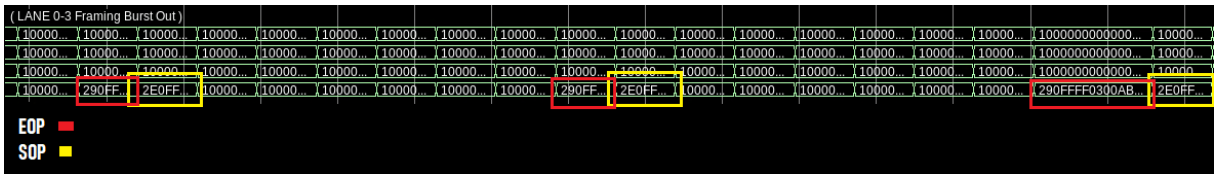


Figure 5.3: First steps of Channel-Bonding Simulation

5.3.2 Simulation

The simulation is done in Questa-sim. In Figure 5.3 a sample of the data-out signals from the Burst-Framer channel 0 to 3 (in order) are shown. When you start counting the frames from the first SOP to the next EOP (by following the order channel 1-2-3-4-1-etc..), you find that the Packet has 29 frames (SOP+27*DATA+EOP). After this Packet there are three times data and then the next SOP can be found. These three frames in between EOP and SOP are missed every time because the SOP is given on the wrong channel.

After a visual inspection of the total data sending period, another situation is found where the format also is broken. This is when a lanes Byte-counter counts a full-packet on it's own lane. This is an old mechanic that worked on a single lane cause it could count all passing data and determine how much bytes have past, because it had to pass on this single lane.

As a conclusion can be derived that a way must be found to guarantee a SOP on the right consecutive lane after a EOP. And that the byte-counters must be changed to work with N-lanes.

5.3.3 Revision

The current version of Channel-bonding needs some adjustments to work according to Interlaken format. This means a correct order of SOP, DATA and EOP frames. In the re-visioned version logic is re-designed to send certain flags to and from the Transmitter-MultiChannel. It can be said that inside the Multichannel the lanes are communicating to make the correct actions, because the Multichannel takes the different flags and flag the lanes to frame a SOP, IDLE, EOP or DATA word.

5.3.4 Lane-Format Handling part 1

On the next page, Figure 5.4 and Figure 5.5, a new version of the format handling is shown. The two processes (from Figure 5.2) are now put together into one larger process. The first for-loop of the new process

tries to handle the Tlast when it occurs on lane 0,1 or 2. Lane 3 is in this for-loop not checked for a Tlast.

```
185   LaneFormatHandling: process(clk)
186       variable SOP : boolean := true;
187       variable Idle : boolean;
188       variable EOP : boolean;
189   begin
190       if rising_edge(clk) then
191       Idle := false;
192       EOP  := false;
193       axis_tready_process  <= axis_tready_transmitter_s;
194       insert_burst_sop     <= (others => '0');
195       insert_burst_idle    <= (others => '0');
196       insert_burst_eop     <= (others => '0');
197       insert_burst_data    <= (others => '0');
198            --lane 0-1-2
199            for i in 0 to Lanes-2 loop
200                --search for tlast (EOP)
201                if axis(i).tlast = '1' and axis(i).tvalid = '1' then
202                    EOP := true;
203                    insert_burst_eop(i) <= '1';
204                    Idle:= true;
205                end if;
206
207                --remember package had EOP
208                if EOP then
209                    SOP := true;
210                end if;
211
212                --Signal the SOP after an EOP
213                if SOP and not EOP and axis(i).tvalid ='1' then
214                    insert_burst_sop(i) <='1';
215                    SOP := false;
216                    Idle:= true;
217                end if;
218
219                --Insert burst idles after SOP or EOP
220                if Idle then
221                    axis_tready_process(i+1) <= '0';
222                    insert_burst_idle(i+1) <= '1';
223                end if;
224            end loop;
225
```

Figure 5.4: Re-visioned Version of the Channel-Bonding Process part 1, 1-6-2020

The purpose of this loop is the following. When a Tlast is found on lane 0, lane 0 gets flagged to send an EOP. In addition lane 1,2 and 3 need to send an IDLE. This is to keep the lanes in sync, as it is used to let lane 1,2 and 3 wait for the SOP.

When the Tlast is found on lane 0, EOP is set to true and this allows SOP to be set to true, as a reminder to send an SOP the next clockcycle. In the next clockcycle the SOP will occur on lane 0 and because of the idles on all the other lanes, lane 1,2 and 3 will send data belonging to the new package.

The same story applies when the Tlast occurs on lane 1 and 2.

```
226             --if the last lane has EOP
227             if(axis(Lanes-1).tlast ='1' and axis(Lanes-1).tvalid = '1') then
228                 insert_burst_eop(Lanes-1) <= '1';
229                 EOP := true; SOP := true;
230             end if;
231
232             for i in 0 to Lanes-1 loop
233                 --Insert burst idles when not valid
234                 if(axis(i).tvalid ='0') then
235                     axis_tready_process(i) <= '0';
236                     insert_burst_idle(i) <= '1';
237                 end if;
238
239                 --If nothing was true then insert data
240                 if (Idle or EOP or SOP) = false then
241                     insert_burst_data(i) <= '1';
242                 end if;
243             end loop;
244     end if;
245   end process LaneFormatHandling;
```

Figure 5.5: Re-visioned Version of the Channel-Bonding Process part 2, 1-6-2020

5.3.5 Lane-Format Handling part 2

In figure 5.5 the rest of the process is shown. When the first three lanes are tested, the last lane gets checked for a Tlast. Then another loop is used to determine if a lane should send data or be idle because there is no data.

All the information for this process is passed on directly from the Framing Burst to the Transmitter-MultiChannel to be used in this process. After each clockcycle the newly determined lane flags are ready for the Framing Burst to use.

5.3.6 Simulation with Lane-Format Handling process

In figure 5.6 the re-visioned version of Channel-Bonding is shown. In red the EOP's are marked, in yellow the IDLE words and in Green the SOP. The Interlaken packet-format is reached in this section of the simulation, when using the AXI-Stream Interface Tlast as EOP flag.

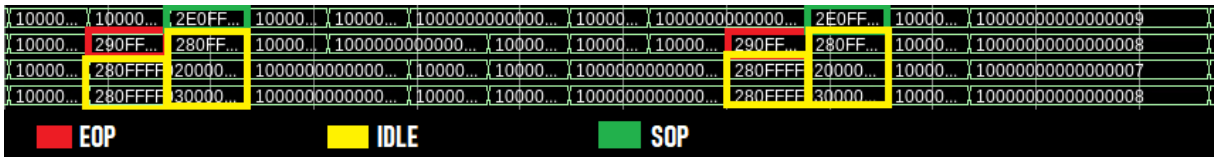


Figure 5.6: Re-visioned Version of Channel-Bonding simulation, 1-6-2020

There is still some issues why the TX-side isn't working yet. There are still some very specific signal trigger combinations that allow the lanes to skip a data frame.

The next two chapters will contain the resulting conclusion and recommendations of the project.

6

Conclusion

The conclusion Chapter tries to give all the conclusions in one general place. It contains the conclusions of Chapters 3, 4 and 5. After this Chapter a recommendation chapter will follow.

6.1 AXI-Stream Conclusion

The AXI-Stream Interface is implemented in a way it can be used during Channel-Bonding. In the current version it allows the use of Tready and Tvalid signals to have a communication with a handshake. It sends data with the Tdata signal and makes use of the Tlast and Tuser signal. The use of a AXI-Stream FIFO allows to handle overflowing data till the FIFO depth is reached. The AXI-Stream Interface is implemented for the in and outputs of Interlaken, this means it is used in the Framing Burst and the De-Framing Burst.

Testing the implementation showed that the AXI-Stream protocol now is a necessity to use when communicating to Interlaken, and that all tested features were according to specification. In the recommendations it shows that there are many more ways to make sure the complete interface is according to specification and more tests to reach this goal are shortly described.

6.2 Transceiver Conclusion

The Transceiver will change as a new FPGA will be used. The main reason is that it's internal logic should be faster. This new frequency should allow the 10 Gb/s lane to speed up to 25 Gb/s. Optimizations of all the processes were performed to help in this effort. The Interlaken150G IP-Core was generated with the Xilinx IP-Core Wizard and a wrapper file is written to use this IP-Core. Because the Core uses a fully implemented version of Interlaken it is used in almost all the test. During the tests there was some difficulty with the Transceiver to show output and this could have different reasons. In the end the Transceiver is ready to run with the 400MHz clock but has to wait on the Channel-Bonding to be done. Timing a 25 Gb/s lane transmission speed is therefore on-hold.

6.3 Channel-Bonding Conclusion

Work on Channel-Bonding had to wait till after the Core1990 optimization and the AXI-Stream Interface implementation. This is because Channel-Bonding uses the optimized modules and the AXI-Stream Interface. Also the Transceiver was done before the Channel-Bonding implementation as it was a great addition to the test-environment.

Channel-Bonding is the most important part of reaching the main goal of a 100 Gb/s transmission speed, as the lanes will distribute the throughput of the data-stream. The Channel-Bonding is mainly designed in the Transmitter-MultiChannel. Here it uses flags from the lane to determine actions and flag these actions back to the specific lanes. The code is written with as many dynamic references to pre-defined values. This allows for future up or down-scaling of number of lanes and package-size. In the end the tests show that the packet format is reached, but still contain cases where it breaks format. Because the Channel-Bonding isn't yet finished, the main goal isn't reached. But all the work (collected in this paper, including Appendices) together are still a great contribution to the effort.

7

RECOMMENDATIONS

During this project many different aspects were worked on and the project is long to be finished. This Chapter will give recommendations to help with the future process of the Interlaken Firmware.

7.1 AXI-STREAM RECOMMENDATIONS

In the test-bench Chapter, the AXI-Stream parts of the AXI-FIFO and Framing Burst were tested and on the RX-side the De-Framing Burst AXI-Stream signals that were used to produce an output. There are many more ways to improve this test.

7.1.1 IMPROVEMENTS FOR THE TEST-BENCH

The test-bench needs some improvements to make a more comprehensive test. Next will be three section to describe additional parts that need testing and ideas about how this could be added in the current version.

7.1.2 TESTING AXI-FIFO

The AXI-FIFO is a buffer. This means that the FIFO could get full as it has a limited size. The AXI-FIFO should use the AXI signals to stop the master from sending data when the FIFO is full. A test could be made in which a very long continuous stream of data gets sent, till the FIFO is full, to see if the FIFO keeps behaving correctly when full.

7.1.3 TESTING FRAMING BURST

The Framing Burst is in the current test-bench tested with a clean and simple pattern. A more difficult pattern and less clean data-stream could give more insights if the Framing Burst can keep the AXI-Stream communication up in a environment with more different situations.

7.1.4 TESTING DE-FRAMING BURST

The De-Framing Burst is only tested on sending the data out, it is currently not tested if it uses the right output signals. At least the output signals are only inspected visually. A better way to test the AXI-Stream Interface is by connecting a AXI-Stream device behind it for communication.

7.2 TRANSCEIVER RECOMMENDATIONS

The Interlaken150G IP-Core is at the moment still for the most part an unknown module. With some help from the Xilinx documentation it was possible to connect and use the IP-Core so far, but it is a serious recommendation to find a way to get output from the IP-Core or to find an alternative that works according to Interlaken specification and shows an output. With such a device the test will be more complete.

7.2.1 Other possible test layouts with the Transceiver

The Interlaken150G IP-Core can also be used to test the RX-side of Interlaken, by connecting the TX of the IP-Core to the RX-side of Interlaken. To achieve this the wrapper must be adjusted and the testbench needs to interface with a AXI4-Light Interface. Setting this up will allow future versions of this project to have a re-usable test environment to test the working of Interlaken both ways.

7.2.2 Timing the 25 Gb/s lane transmission speed

The Interlaken150G IP-Core is almost ready to be used with the 400 MHz clock. When Channel-Bonding is in it is advised to start with timing analyses to find out if the Interlaken logic is up to speed or has bottlenecks that should be optimized.

7.2.3 Chain test multiple FELIX cards

Another good way of testing the Transceiver is by chaining a few FELIX cards and let data go through multiple transceivers. Such a chain could help in finding why in the current setup the Transceiver refuses to give an output.

It will also help with testing the RX-side of Interlaken. By connecting the RX-side of device one to the TX-side of device two the outgoing communication of the RX-side could be tested and the timing of the RX-side.

7.3 Channel-Bonding Recommendations

Channel-Bonding wasn't finished yet, so the first recommendation is finishing it. This could be done by watching the Framing Burst outputs and find out in which situations they stop working according Framing Burst format.

7.3.1 Testing the 100 Gb/s

When Channel-Bonding is achieved it is advised to setup a test-environment to start with the 100 Gb/s analysis. This is done by programming Interlaken into the new FPGA and set the Transceiver to use the 400MHz clock. But before testing this it is probably best to perform the recommendations on the Transceiver first.

When testing the 100 Gb/s it is most probable that the timing isn't met. Here is some advise on which things could speed up the timing.

7.3.2 Checking your > and < operation

Some if statements will probably use a > symbol or a < symbol. These operations can be relatively logic consuming as the larger than combinations need to be made out of logic (and the same for smaller than). These operations need not to be totally avoided as there are situations were you have for example a 4bit signal and the statement says "if (x > 12)". The logic then only needs to check the bit combinations corresponding to the number 13,14 and 15. This isn't that harsh but when you change it to "if (x < 12)" there are the numbers 0-11 to test. So the > and < could quickly use up much logic but can also be quite effective if strategically used.

An addition to this knowledge could be to change 1: "if(x > 12)" to 2: "if(not(x > 12))" instead of 3: "if (x < 12)". As theoretically the second situation should only need logic to find 3 combinations + a inverting gate instead of the 12 combinations that need to be found in situation 3.

7.3.3 Reduce all different possible states

In VHDL a state occupies at least a single clock-cycle. It is therefore advised to make as many things possible in one combined state than in to separate ones. For example some of the older state-machines had five different states that did almost the same. They have now been reduced to one larger state with all the same capabilities.

Not always do you want to put states together. This is because some actions are always chained in the same order and could therefore easily put in a row by using states. For example Interlaken gives all the packages a chain of Meta-words in the Meta-framing, as they always happen in the same order, the states don't waste a full clockcycle.

7.4 Other Recommendations

There are also some other recommendations. The project currently uses many scripts in streamlining the project environments. At the start of this project many different scripts for, for example Sigasi, Vivado and Questa-sim were adapted or rewritten.

7.4.1 Questa-sim scripts

The Questa-sim script gave some ideas. In the current Import Questa script certain VHDL-files get added to the project and are compiled. After this it sets up a wave window with a signals ordered in groups. All this in just one script saved a lot of time with setting up a simulation environment. Therefore it could be a great idea to make such a script for testing the TX-side, the RX-side, the TX-side with the RX-side or maybe even specific modules as the Framing Burst, could be worthwhile.

7.4.2 Standard situation test-benches

Another idea is to make some additions to te current test-bench. At the moment the test-bench uses some defines to determine which setup to use, for example connect the TX in loopback to the RX or to connect the TX to the RX of the Transceiver IP-Core.

7.4.3 Just some advise

And as last item some advise. The advise is to try and keep all the files and all the work as dynamic as possible. The main reason is that many old parts needed to be rewritten or changed when Channel-Bonding came along. So pay attention to signals that could have the possibility to be scalable.

REFERENCES

[1] Leonie Verwoert. “Felix from Interlaken”. In: (2019), pp. 1–82 (cit. on pp. 2, 1, 3, 4).

[2] Nayib Boukadida. “Point-to-point protocol exploration”. In: (2019), pp. 1–80 (cit. on pp. 2, 13, 1).

[3] XilinX. “ug576-ultrascale-gth-transceivers”. Version 1.6. In: (2019). URL: https://www.xilinx.com/support/documentation/user_guides/ug576-ultrascale-gth-transceivers.pdf (cit. on pp. 13, 1, 2).

[4] XilinX. “Integrated Interlaken 150G”. Version V2.0. In: (2016). URL: https://www.xilinx.com/support/documentation/ip_documentation/interlaken/v2_0/pg169-interlaken.pdf (cit. on p. 14).

[5] Nikhef. “The Atlas Project”. In: (2019). URL: {https://www.nikhef.nl/program/atlas/} (cit. on p. 27).

[6] Nikhef and Cern. “Information on FELIX project”. In: (2019). URL: {https://www.nikhef.nl/~i73/FLX19} (cit. on pp. 1, 2).

[7] Nikhef and Cern. “The ATLAS FELIX Project”. In: (2019). URL: http://atlas-project-felix.web.cern.ch/atlas-project-felix/ (cit. on pp. 1, 2).

[8] NIKHEF. “The ATLAS FELIX Project”. In: (2019). URL: https://iopscience.iop.org/article/10.1088/1748-0221/11/01/C01055/pdf (cit. on pp. 1, 3).

[9] Xilinx. “Xilinx Virtex-7 FPGA VC709 Connectivity Kit”. In: (2020). URL: https://www.xilinx.com/products/boards-and-kits/dk-v7-vc709-g.html (cit. on p. 2).

[10] Xilinx. “Virtex UltraScale+ HBM VCU128-ES1 FPGA Evaluation Kit”. In: (2020). URL: https://www.xilinx.com/products/boards-and-kits/vcu128-es1.html#whatsInside (cit. on p. 2).

[11] "ARM". ""AMBA 4 AXI4-Stream Protocol”. In: () (cit. on pp. 1–4, 7).

[12] Cortina Systems Inc. “InterlakenProtocol Definition, A Joint Specification of Cortina Systems and Cisco Systems”. Version 1.2. In: (2008). URL: http://interlakenalliance.com/wp-content/uploads/2019/12/Interlaken_Protocol_Definition_v1.2.pdf (cit. on pp. 1, 5).

[13] NIKHEF and CERN. “ATLAS FELIX firmware Phase-II Upgrade: Firmware specifications”. Version 0.4. In: (2019) (cit. on p. 5).

[14] NIKHEF. “Interlaken Protocol Definition”. In: (2008). URL: http://interlakenalliance.com/wp-content/uploads/2019/12/Interlaken_Protocol_Definition_v1.2.pdf (cit. on p. 1).

Appendix A

The Atlas Experiment

To understand why FELIX is continuously developed and optimized it's best to start explaining the project it's designed for. In the next few paragraphs, you find a brief introduction to the Atlas project and what it's about.

A.1 About the Atlas Experiment

The Atlas is an experimental detector. This detector is one of the detectors of the Large Hadron Collider (LHC) from CERN Geneve. The Atlas is designed to be a so-called "general purpose" detector, used to find a complete summary of everything there is to a particle collision. The detector itself consists of multiple layers that are positioned as rings around each other. In total there are more than 150 million sensing elements onboard the atlas detector. Piecing all the information together from these elements, pictures are made that represent the collision that had been measured. [5]

A.2 Things to discover with the Atlas Experiment

In the end, all this information is used to research multiple things. The Higgs particle is one of those. This particle was theorized to exist by Peter Higgs in 1964. In 2012 the Atlas detector and the Cms detector first measured the existence of the Higgs particle. And now all the data Atlas can provide about this particle is used to confirm and reconfigure the prediction Peter Higgs made all these years ago. Another thing the Atlas detector is used for is to find measured proof of the existence of phenomena and new particles, for example, what is called "Dark matter". Also searching for supersymmetry particles (SUSY) is an example of the broad use of the Atlas detector. [5]

A.2.1 Other exploitation of the Atlas Experiment

Not only discoveries are made. The Atlas project is also used to gain better precision in measurement and modeling, as there is more knowledge about the behavior of particles. This kind of information leads to proofing and dis-proofing what is known about the forces of nature. [5]

A.3 Nikhef's contributions to the Atlas Experiment

Nikhef brings own contribution to the Atlas Experiment. Muon-detectors are one of them. They are designed, built and tested at Nikhef and later implemented in the Atlas detector. Nikhef also designed many parts of the electrical modules, the activation system, and the data-acquisition system. One of these systems uses FELIX. (Explained in the next Appendix B). FELIX is designed to have a very high transmission speed and is sensor independent. The first reason is the different use of sensors included in the Atlas project, the second the fast speed (nearly speed of light) that is the speed at which the particles collide and at which they are measured. [5]

Appendix B

THE FELIX-CARD

Knowing the 2 key features of FELIX from Appendix A is not enough to apprehend how FELIX is used, which progress FELIX has been through and the most certain future FELIX will have. These things are explained in this Appendix.

B.1 THE ROLE OF FELIX

FELIX stands for FrontEnd LInk eXchange. It's design to provide a sensor/detector independent readout architecture. FELIX is the data aquisition system. It is placed between the front-end electronics, trigger electronics and the connected network with designated endpoints. In Figure B.1 is a more detailed overview of how the FELIX is used in a system. [6] [7] [1]

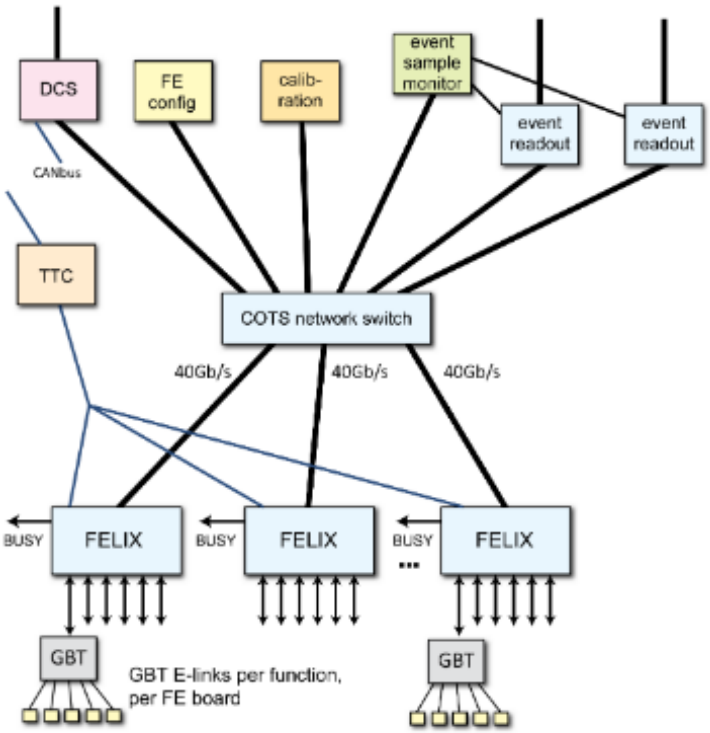


Figure B.1: Overview of the connectivity of the FELIXsystem (DCS refers to the Detector Control System,"FE Config" to Front-End configuration).[8]

B.2 PROGRESS 2016-2019

In 2016 FELIX used the GBT protocol hence the GBT E-links in Figure. B.1 This protocol was used to get to a 4.8Gb/s link speed, but they noted that another protocol must be used to make use of the 9.6GB/s link speed of the GBT E-links.

B.2.1 FELIX 2018

In 2018 the student researcher N. Boukadida started researching several possible point to point protocols[2] that could be used to reach a higher speed. He came up with the Interlaken protocol and made Core1990. Core1990 is the first version of the Interlaken protocol implementation for FELIX. But it was insufficiently developed for direct use. Nonetheless N. Boukadida managed to create much of the logic still used in the newer Core1990 V2.

B.2.2 FELIX 2019

In 2019 another student researcher named L. Verwoert continued the work on Core1990. [1] She fulfilled the not yet finished Interlaken protocol. Simultaneously she worked on adding Wupper to FELIX. Wupper is a framework component whose main functionality is to handle data transfers from a user interface to and from the host PC memory. After this was finished the design could get to a transmission speed around 10.0Gb/s. During optimization and the research for better methods, the idea of channel-bonding arose. Channel-bonding would allow 4 channels simultaneously instead of the current implementation with 1 channel and the transmission speed will raise from the 10.0Gb/s to 40Gb/s (4*10.0Gb/s). The same research also found that another interface could be realized after the channel-bonding, the AXI4-stream Interface. This interface is a more common interface for channel-bonded implementations. In the last version of Core1990, called Core1990 V2, are channel-bonding and the AXI4-stream not yet implemented. [6] [7]

B.3 Renewed FELIX 2020

The FLX-card that Core1990 V2 was designed for contains a Xilinx VC-709 board with a Virtex-7 X690T FPGA, a PCIe Gen3 card and four SFP+ optic link connectors.[9] In the 2020 design, a new evaluation board will be used. This means that the Virtex-7 X690T FPGA is replaced with the Virtex Ultrascale + (HBM) VU37P FPGA. The main reason for this change is the possibility to get to higher transmission speeds. Or simply said, the new one is faster.[10] The idea is to adjust Core1990 V2 with the new FPGA. In the 2020 plans of FELIX, Channel-bonding and an AXI4-Stream Interface will be the next step to achieve a FELIX-Interlaken design with a 25Gb/s transmission speed and a more commonly used interface.

B.4 The addresses of VU37P (FPGA)

Many modules of Interlaken are not dependant of the used FPGA. This means that not everything needs to be rewritten when another FPGA is in use. There is one main thing that needs to be rewritten, the transceiver. The Virtex Ultrascale + has 96 GTY transceivers on board, they are grouped by four. In Appendix 4 Transceiver, the changes are discussed.

Appendix C

CHANNEL-BONDING AND SYNCING LANES

When Core1990 V2 was finished there was just a little time left to research additional optimization possibilities to enhance the Interlaken implementation. It is found that the transceiver IP core, used in Core1990 V2 firmware, supports a feature called Channel-bonding. So let's discuss Channel-bonding.

C.1 GTY TRANSCEIVER

The current version of Interlaken (Core1990 V2) uses a serial transceiver connection. Such a connection is called a (single) lane. The GTY transceiver provides a way to combine multiple lanes. In a normal situation, when you try to combine lanes, you will probably get to discover the following.

> *"Unless each of the serial connections is the same length, skew between the lanes can cause data to be transmitted at the same time but arrive at different times " [3]*

C.1.1 SKEW HANDLING

Skew is, of course, a problem. When data arrives at a different pace than the way it was meant, without any additional instructions, lanes will be wrongly ordered when they are put together. For this purpose channel-bonding support on the GTY transceiver exists. The GTY channel-bonding cancels out the skew between GTY transceiver lanes by using the RX elastic buffer as a variable latency block. To manage the skew in the right way the buffer needs control characters or a sequence of characters. When the receiver receives these characters (from a transmitter), skew can be determined between the channels. With the right skew, the buffer can determine the correct latency to remove skew between the lanes. In Figure C.1 a conceptual view of channel-bonding is given.[3]

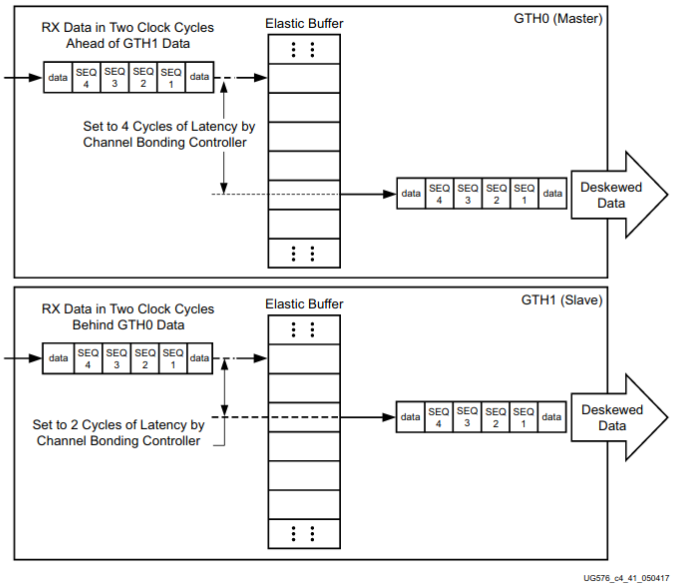


Figure C.1: Channel Bonding Conceptual View with the Elastic Buffer[3]

C.2 CORE1990 V2 WITH CHANNEL-BONDING

As said in C.1 (GTY transceiver) Core1990 V2 works with a single lane. Another thing that's said is that the receiver of the transceiver gets the control characters from the transmitter of some other device connected to the Rx side. This is what is accomplished in Figure C.2. For simplicity lets start from Figure E.3 and work up to Figure C.2. Next to channel 0, three lanes (channels) have been added that are a copy of the original lane . The lanes are connected between the entry FIFO and the transceiver. The transceiver is linked back to itself because we test without an external device. (so normally it is linked to another device). The time between control characters of different lanes will be used to determine the latency. The latency can then be used by the elastic buffer of the transceiver to remove the skew. In the next paragraph more specifics on how Channel-Bonding could be realized. [1][3]

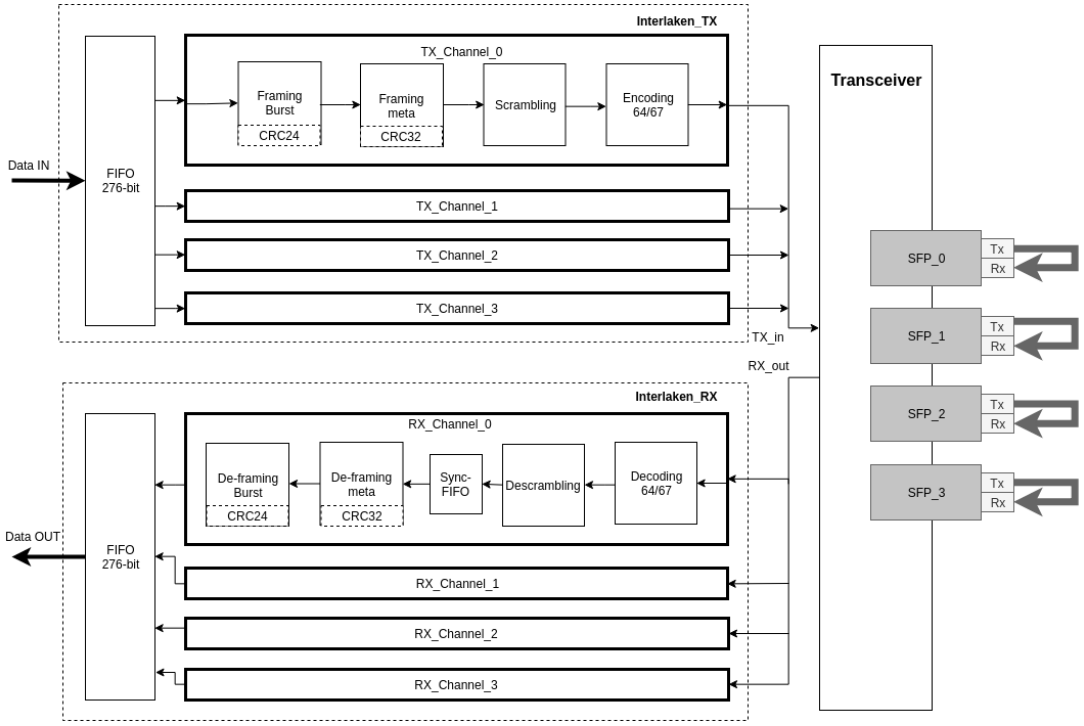


Figure C.2: 4 channel design of previous design in Figure E.3 [1]

C.3 SPECIFICS OF CHANNEL-BONDING

In the new version of Interlaken, channel-bonding will be implemented. The Encoder and decoder will have an AXI-Stream interface. From the Decoder (inside Interlaken TX) data will stream to the transceiver. This will happen with four data lanes (channels)parallel to each other. Each lane will have skew. To handle the skew, sync words will be added to the data. To find sync words each lane will be bit-shifted with one bit. The bit-shifts are used to align the sync words. For something like this to happen, it is most likely that the channels are going to communicate with each other. When the Sync words are aligned, the data flows into the Transceiver. Because of testing this single device the Transceiver will be looped into itself, connecting TX with RX. At the RX side, the data will flow into their lanes towards the decoding block. And will be handled accordingly.

Appendix D

AXI-Stream Interface

In Appendices B and E.4 AXI-Stream was already briefly mentioned. AXI-Stream or AXI4-Stream is a standard interface that allows 2 devices to exchange data. The interface can be used to connect a single slave, that generates data, to a master, that receives data. The protocol can also be used when connecting larger numbers of master and slave components. The protocol supports multiple data streams using the same set of shared wires, allowing a generic interconnect to be constructed that can perform up-sizing, downsizing and routing operations. AXI-Stream also supports a variety of different stream types[11]

D.1 General definitions and terms

The Axi-stream protocol works with a couple of byte definitions and a couple of stream terms. In Table D.1 and Table D.2 they are shown. These definitions can become a bit confusing when not addressed, so it's therefore that they are going to be explained.

D.1.1 Byte definitions

The first one is called a data byte. Data bytes are specified as a single byte that contains valid information and is transported between the source and destination. Second, there is the Position byte. This is s byte that indicates the relative positions of data bytes within the stream. This is a placeholder that does not contain any relevant data values that are transmitted between the source and destination. Last the Null byte. The null byte is a byte that does not contain any data information or any information about the relative position of data bytes within the stream. Appendix D.1.2 to Appendix D.2.4, will now tell more about different stream types, word use and basic info. If these terms are already known feel free to go ahead to Appendix D.3 about interface Signals. [11]

Byte name	*Definition*
Data	*Byte of valid data*
Position	*Byte that indicate the relative position of data bytes, used as placeholder and for data shifting*
Null	*Byte that doesn't contain data or position*

Table D.1: Byte definitions

D.1.2 Stream terms

In the next table, table D.2, are four terms shown. First its Transfer. Transfer is a single transfer of data across an AXI4-Stream interface. A single transfer is defined by a single TVALID, TREADY handshake. Second, there is a group of bytes that are called a packet. A packet may consist of a single transfer or multiple transfers. Infrastructure components can use packets to deal more efficiently with a stream in packet sized groups. The last data format is a frame that is the highest level of byte grouping in an AXI4-Stream. A-frame contains an integer number of packets. A-frame can be a very large number of bytes, for example, an entire video frame buffer.

Stream terms	*Definition*
Transfer	*Single transfer of data across the interface, it makes use of an handshake*
Packet	*Group of bytes that are transported together. Existing out of a single or multiple transfers*
Frame	*The largest byte group consisting of an integer number of packets*
DataStream	*The transport of data from one source to one destination, can be transfers or packets*

Table D.2: Stream definitions

In the fourth row you can read DataStream. Datastream is the transport of data from one source to one destination. This can either be a a series of individual byte transfers or a series of packets. Now these definitions are explained it's time to move on to explain the different data streams.[11]

D.2 Data streams

Data streams can be used in many forms. AXI-Stream explains 4 standard stream types. In Table D.3 an overview of the 4 stream types. In the end, there are no definitive data stream types that are mandatory. These four are just the most commonly used.

Stream types
Byte stream
Continuous aligned stream
Continuous unaligned stream
Sparse stream

Table D.3: Stream types

D.2.1 Byte stream

The Byte stream is a stream consisting of data bytes and null bytes. After a valid TVALID and TREADY handshake, any number of data can be transferred. It doesn't matter how much null or how much valid data bytes are used after each other, because the null bytes contain no data and can be inserted or removed when wanted.[11]

D.2.2 Continuous aligned stream

The continuous aligned stream is the most simple one. At this stream type packets have no null or position bytes. Only data is transferred. This is commonly used when there is always data available for transmission. It allows only to transfer full data byte packets.[11]

D.2.3 Continuous unaligned stream

Continuous unaligned stream allows the use of position bytes. This is used to shift the data back and forward when necessary. This shifting is used to align the packets within a stream. Commonly used when you want to align a packet between aligned packets.[11]

D.2.4 SPARSE STREAM

The sparse stream allows for the transmission of data and position bytes. The position bytes are used to align every packet. This stream type is used when the data byte position is important and data isn't yet pre-aligned.[11]

D.3 INTERFACE SIGNALS

To properly handle everything the AXI-Stream uses a lot of signals. The signals are grouped in the following groups (see Table D.4).

Signal Groups
Transfer Signals
Data Signals
Byte Qualifier Signals
Packet Boundaries Signals
Source and Destination Signals
Clock and Reset Signals
User Signals

Table D.4: Signal groups

The transfer signals are used to secure data transfers with a handshake. The data signals are used to transfer data from their source to their destination. The third one of Table D.4 (Byte Qualifier Signal) is used to distinguish position bytes from data bytes. Also, the task if a byte should transfer or should wait is handled with byte qualifier signals. Packet boundaries signals are used to control data in the form of a packet, this helps to create a more efficient infrastructure for transporting the data. Then there are Source and destination signals that determine the routing from source to destination and information about the used stream type. The clock and reset manages the speed and the asynchronously resets of all components. The last signal to discuss is called user-signal. The user signals allow for sideband signaling. Flags, parity, control signals are examples of this.

D.3.1 SPECIFYING THE INTERFACE SIGNALS

To gain proper insight in the signals that are used by the AXI-Stream interface, Table D.5 is made.

Signal	*Source*	*Description*	*Signal size*
Transfer Signals			
TVALID	*Master*	*Master Valid for transfer*	*[1]*
TREADY	*Slave*	*Slave Valid for transfer*	*[1]*
Data Signals			
TDATA	*Master*	*Data Payload*	*[(8*data bus-1):0]*
Byte Qualifier Signals			
TKEEP	*Master*	*Makes byte NULL*	*[7:0]*
**TSTRB*	*Master*	*Data [1] or Position [0]*	*[(data bus-1):0]*
Packet Boundaries Signals			
TLAST	*Master*	*Marks end of packet*	*[1]*
Source and Destination Signals			
**TID*	*Master*	*Select Data Stream*	*[(8-1):0]*
**TDEST*	*Master*	*Determines data route*	*[(4-1):0]*
Clock and Reset Signals			
ACLK	*Clock source*	*Rising edge*	*[1]*
ARESETn	*Reset Source*	*active reset when [0]*	*[1]*
User Signals			
TUSER	*Master*	*user defined information*	*[3:0]*

Table D.5: Signal groups (* means not used see Appendix D.4.1)

Almost all signals are already introduced. Instead of repeating the story per signal, take a look at the signal size. Most of them are quite simple. A simple signal like TVALID is only true or false or said to be 1 bit. Others like the byte qualifier signals are the size of the data bus, cause they need the space to qualify per data signal bit. The two signals that need a more elaborate explanation are the source and destination signals. First look at the signal TID. TID is used to select the data stream. As TID is 8-bits it will allow up to 8 channels. TDEST is 4-bit and determines the data route. The 4-bits allow up to 4 different streams sequential on the same channel. In the end, each unique combination of TDEST and TID are merged and up to a maximum of 32 different data streams can be progressed. Note that, clock depending, only a certain amount of data can be progressed sequentially. In other words, more different TDEST on the same TID can come at the cost of speed. The last thing to discuss signal size is the signal TUSER. TUSER is an extra for the user of AXI4-Stream. As a user, you can decide to use this extra signal to send any additional information to the Slave. This could be anything and could also be ignored. At Interlaken TUSER can be used to fill a large error-holding container. This could then be used to check what's wrong with a certain piece of the data or stream. (more about this table in Appendix D.4.1) [11]

D.3.2 Default Signaling Requirements

To create a system that works an initial state has te be defined to begin from. If you start with settings that are wrong, weird things could happen. in Table D.6 an overview of what the initial state of the signals should be.

The initial states are mostly locked. But a couple of signals have some special requirements. TLAST shows a full description of this requirement. (Table D.6) Yet TKEEP is written to ALL-HIGH in the Table, it isn't always the case. Hence Table D.7 shows the four possible situations. Note the fourth situation "Reserved". Reserved is assigned because situation 01 was left. Reserved is a data type that must not be used. But because TSTRB will not be used only Data byte and Null byte remain an option for Interlaken. Why some signals aren't used will be discust in Appendix D.4.1.

Signal	State
Transfer Signals	
TVALID	*LOW*
TREADY	*HIGH*
Data Signals	
TDATA	*ALL LOW (no initial data)*
Byte Qualifier Signals	
TKEEP	*All HIGH*
TSTRB	*NOT USED*
Packet Boundaries Signals	
TLAST	*HIGH if no delays, or 1 of multiple devices don't support TLAST, LOW if endless package or no devices support TLAST*
Source and Destination Signals	
TID	*NOT USED*
TDEST	*NOT USED*
Clock and Reset Signals	
ACLK	*LOW*
ARESETn	*HIGH*
User Signals	
TUSER	*ALL LOW*

Table D.6: Signal initial state

TKEEP	TSTRB	Data-type
1	*1*	*Data byte*
1	*0*	*Position byte*
0	*0*	*Null byte*
0	*1*	*Reserved*

Table D.7: TKEEP and TSTRB situations

D.4 Decisions and Pre-design

The previous parts of this Appendix gave a fast idea of the AXI-Stream interface and its capabilities. Now it's time to determine a plan and make decisions on how the AXI-Stream eventually will be used with Interlaken. This problem will be tackled in the upcoming Appendices D.4.1 and D.4.2 .

D.4.1 Decisions

let's begin by discussing the possible data stream types. (Appendix D.2) The common 4 data streams are not all necessary for our system. Felix just needs a unified protocol to send data back and forth. This is why the Byte stream is sufficient. The Byte stream consists of a handshake, data bytes, and null bytes. Another reason to choose the Byte stream is because of the other 3 stream types, they all have extra things to control with position bytes. In other words, the byte stream is the plainest choice.

When zooming in on the byte stream, a lot of signals won't be necessary for the design. The signals with a star in Table D.4 mark all the signals that will not be needed. From top to bottom first TSTRB. TSTRB makes it possible to use position bytes. As Byte Stream won't use them, the signal is no longer needed. Next TID and TDEST. They both make it possible to use multiple streams in the same device. Because FELIX has a high demand in speed these two signals will be discarded, as multiple signals would demand more decision making of which is who. It is therefore thought that dynamic signal declaration can be used if there are multiple

channels. For example, 1 channel of 64 bytes will go trough the AXI stream with 64 bytes parallel. When you upscale to 2 channels of 64 bytes which each require an AXI stream, then you simply could make the AXI stream 128 bytes (2*64). The main difference is that the multiple channels allow for parallel processing and the TID and TDEST signals would handle this sequentially. With this new knowledge a pre-design is described in Appendix D.4.2.

D.4.2 PRE-DESIGN

It is important to make a pre-design of the AXI stream block. It gives an overview of what is going to be made and how it should function. Let's start with a diagram.

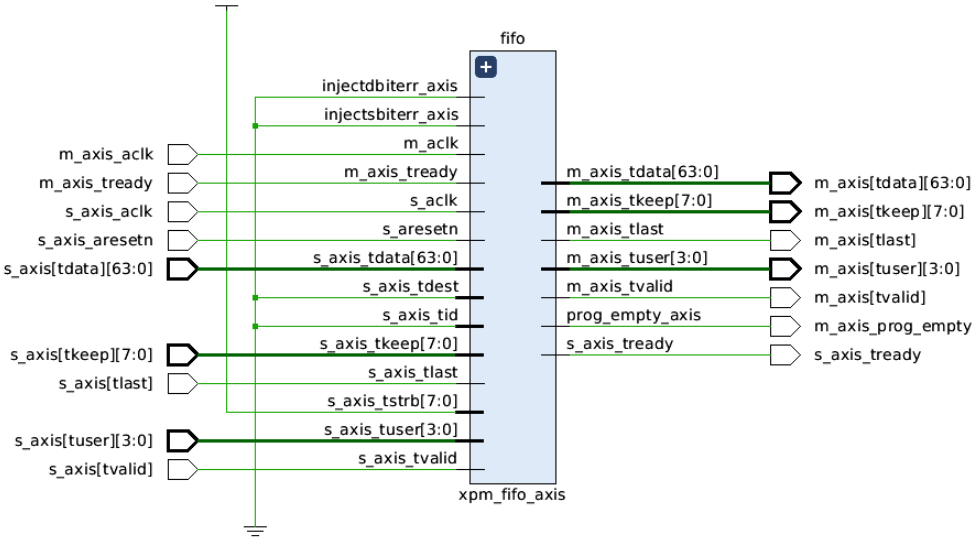


Figure D.1: The AXI-stream Diagram

Above in Figure D.1 is a AXI-Stream diagram visible. On the left are all the inputs and on the right all the outputs. Some of the inputs aren't used and are set on always HIGH (one) or always LOW (zero). The first letter m and s stand for the master device and slave device. In this design, a master device receives data from a slave. This block is made in Vivado only just as a visual and not yet a full implementation. Now, this FIFO must be connected with Interlaken.

The FIFO in Figure D.1 will be connected to the RX side of Interlaken. The end of the RX side is the deframing-burst. In figure D.2 the deframing-burst is shown. The deframing-burst is a little bit altered, now it has the right pins to connect with the AXI-Stream FIFO. Every m (master) version of a signal will be connected with an s (slave) version with the same name. For example, m-axis-data is connected with s-axis-data. The framing-burst will have the same pins but then inverted and named as a slave. Now most parts of Interlaken have been discussed, the next Appendix (E) will be about Core1990.

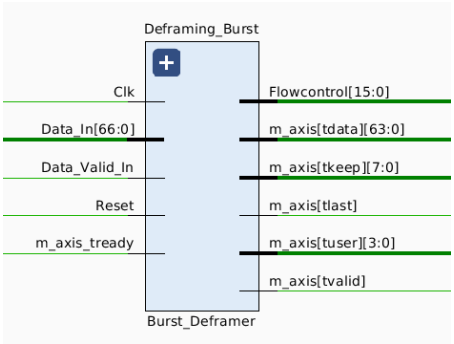


Figure D.2: The AXI-stream Diagram

[11]

Appendix E

INTERLAKEN PROTOCOL AND CORE1990

To understand Core1990, first an understanding of the Interlaken protocol is needed. My predecessors (L. Verwoert and N. Boukadida) researched this protocol. In their papers "Point-to-point protocol exploration"[2] and "Felix from Interlaken"[1] they explain the Interlaken protocol extensively. Therefore a shorter version will be explained in this Appendix.

For now a brief description:

> *The Interlaken protocol is in short a narrow, high speed channelized chip-to-chip interface. It uses a serialized channel, and can be scaled in speed by using multiple channels parallel.* [1] [2]

E.1 BURST-FORMAT

The protocol will use some steps. One of the steps is burst-control. The burst-control is used to start a data burst and stop a data burst. This is used to keep track of multiple data-burst segments that belong together as a packet. Each individual segment is eight bytes. The idle control format is used between complete data bursts to fill the empty spots and keep a continuous stream of controlled data. In other words, it ensures the state of no data when there is no data. In Figure B.1 you can see a multiple burst example.[1]

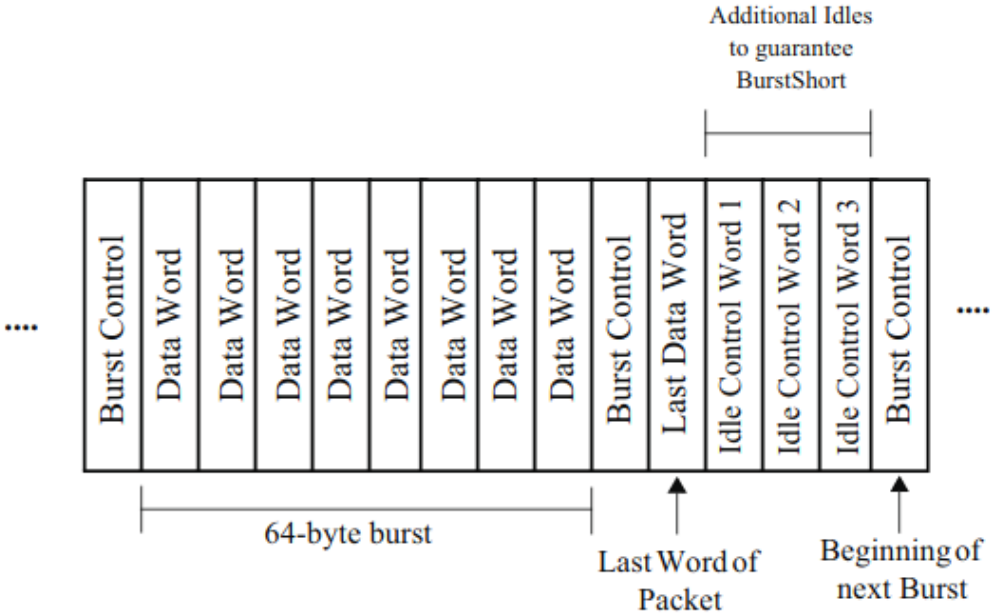


Figure E.1: BurstShort Guarantee Illustration.[12]

E.2 Checksum and meta-frame

To secure the integrity of the incoming data, the burst-control is given a checksum (CRC24). A checksum is a number that is generated by doing a calculation with the data. So if the data has changed the checksum will be different. To fulfill additional checks and controls a meta-frame is created with specific control words[1], see Table E.1.

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

Table E.1: Meta frame block types[1]

E.2.1 Synchronization and Scramble word

The Synchronization word is used in combination with the Scrambler-state word. Every new meta-frame has a Scrambler-state word, this is used to scramble the data and avoid multiplication. Then there will be searched for Synchronization words. After four words are found the scrambler will go in a locked state and let de de-scrambler de-scramble the data.[1]

E.2.2 Diagnostic word

The Diagnostic word is used to determine the integrity of the data with an additional checksum (CRC32). This Diagnostic provides a per lane error detection. The second thing the Diagnostic word provides is a Status message that tells if something is wrong.[1]

E.2.3 Skip word

The Skip word is used to provide clock compensation. This clock compensation is necessary when for example a repeater is placed between the TX and RX side. Without the Skip word data will be lost, as data will shift and make the next payload's data not usable. In Figure E.2 an example use of the Skip word.[1]

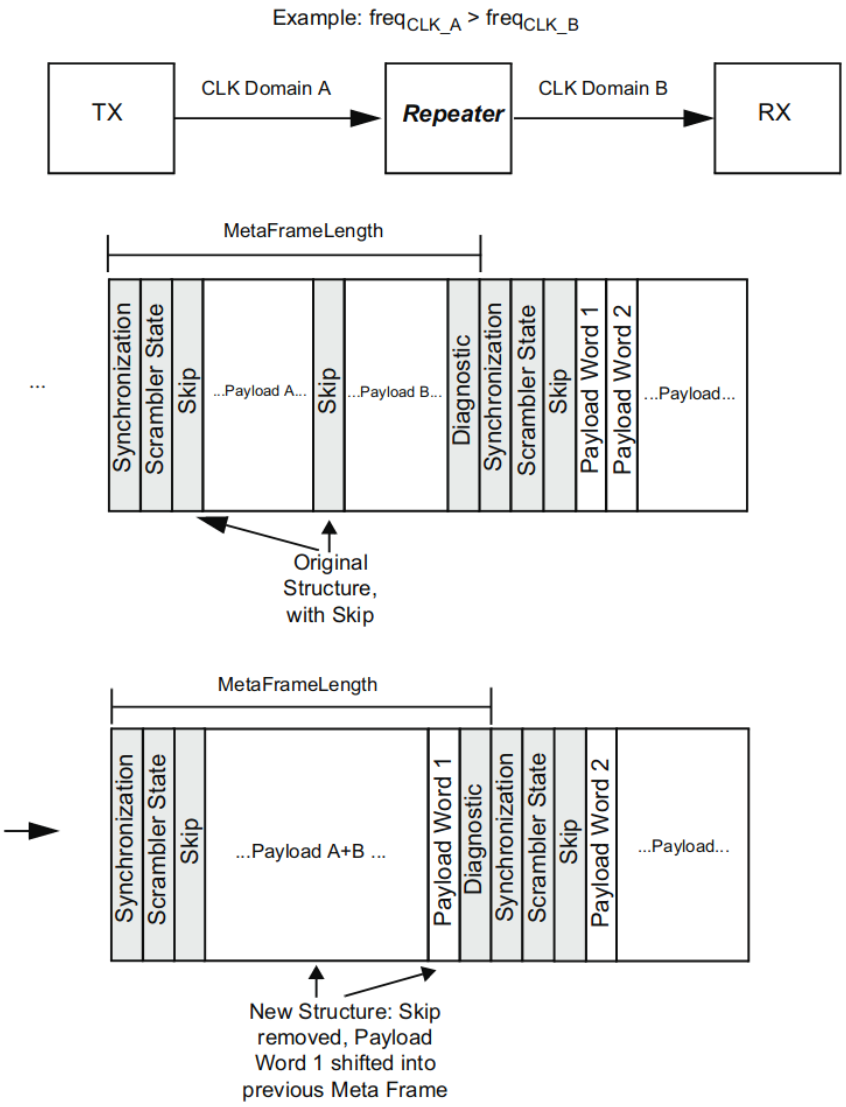


Figure E.2: Example implementation of skip words for clock compensation.[8]

E.3 CORE1990 V2

In the FELIX Appendix B is told about what Core1990 V2 should do. Now it's important to look at the architecture, to gain an idea about how Core1990 V2's is currently implemented. In Figure E.3 an overview of the complete Core 1990 architecture is given.

E.3.1 FRAMING BURST BLOCK

Left in Figure E.3 data is generated and coming in to the system. From this input to the right, first you find the framing burst block. In this block, the data is collected in 8-byte segments and a CRC24 checksum is performed. CRC24 is a 24 bits checksum, later used to verify data integrity. When completed a burst packet is formed. Then the burst continues on to the meta framing (Figure E.3).[1]

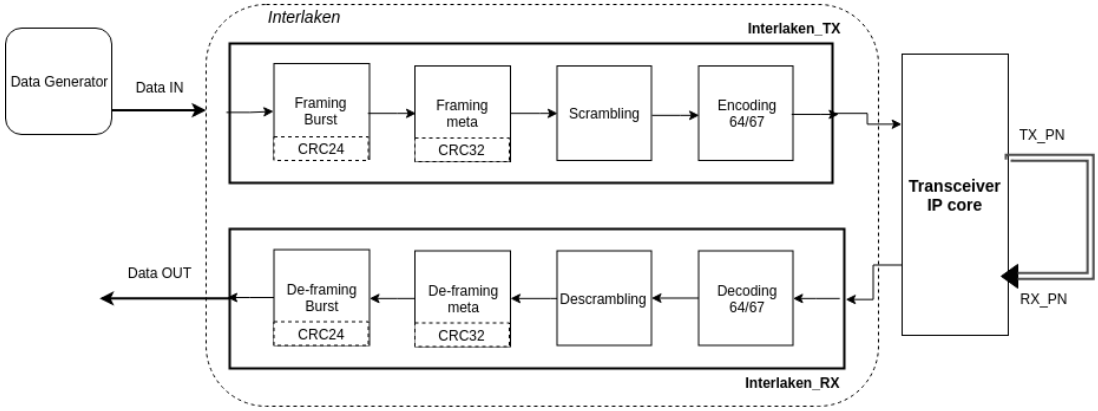


Figure E.3: Structural view of the Interlaken Implementation[1]

E.3.2 Meta Frame Block

A burst packet enters the meta frame-block. In this block, the four codewords from Table E.1 are determined and the CRC32 checksum is performed and added to the frame. The meta frame-block uses the codewords to signal the scrambler.[1]

E.3.3 Scrambler and Encoder Block

The synchronization codewords from the meta frame-block are used to find a lock with the scrambler and the decoder. When both lock, data is allowed to be scrambled and to be encoded. Data moves from the scrambler to the encoder and from the decoder to the Transceiver. [1]

E.3.4 Transceiver Block

Now the data is on the right of Figure E.3, at the TX side of the Transceiver IP core. A larger arrow (in Figure E.3) links the TX of the transceiver to the RX of the transceiver. This is because in the testing situation there is no other device "to talk to", therefore the data is looped back. In a normal situation, the transceiver IP core will be connected to another device.[1]

E.3.5 RX is reversed TX

Now the data is past the TX side (transmitter) of the Core1990 V2. Next, the data will move into the RX side (receiver). The RX side is the reversed of the TX side. First, the data will come across the decoder. After that, it will be descrambled. Next is the de-meta-framing that now checks the CRC32 checksum. If the CRC32 checksum is validated with an issue the data will be marked, and lose its integrity. The last part of the Interlaken application is the de-framing block. Now the CRC24 checksum is verified and if everything turns out te be "fine" the data pours out of the data out pin.[1]

E.4 CORE1990 V3

In the previous part of Appendix E, Core1990 V2's structure is looked into. Core1990 V2 showed to be a 1 channel implementation of Interlaken. To find out what will be needed in a newer version "Core1990 V3" or "Interlaken" its a good start to look at the "phase II FELIX firmware plans" made visible in Figure E.4.

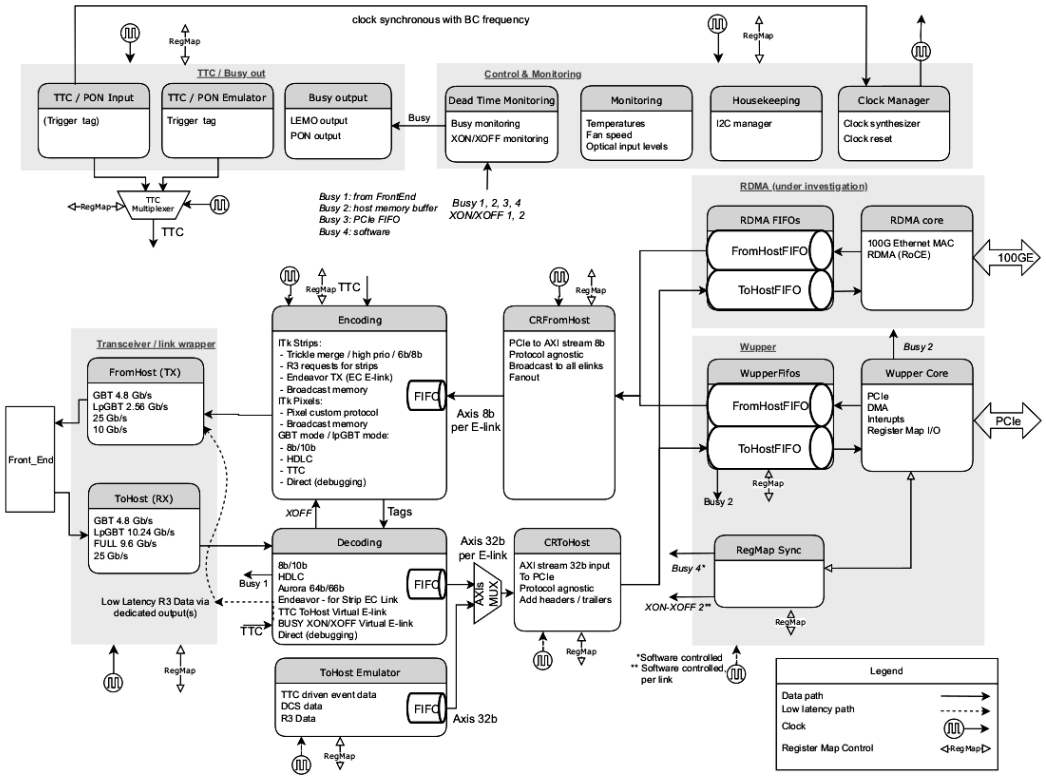


Figure E.4: Structural view of the FELIX phase II plans [13]

E.4.1 FROM HOST (TX) AND TO HOST (RX)

A lot of information is visible in the FELIX phase II plans Figure.(E.4) Let's start on the left. At the blocks "ToHost(RX)" and "FromHost(TX)" 25Gb/s is listed. This has not been the place of the now already existing Interlaken protocol. Yet this transmission speed has not been met, but will probably be after a successful implementation of Channel-bonding.[13]

E.4.2 CR FROM AND TO HOST

Round the middle of Figure E.4 the blocks "CRFromHost" and "CRToHost" can be found. These blocks show the demand for the not yet implemented AXI4-stream interface. (which will be disgust in Appendix D) This interface will allow a standard data processing protocol to handle a lot of different data. Some other things in Figure E.4 are already implemented, not yet necessary or already partly made. An example is Housekeeping which allows for I2C at the right top corner. I2C is currently used to configure the clock generator chip on the board, among other peripherals. In Appendix C more about channel-bonding.[13]

Appendix F

PLANNING

For this project, a planning is made. The planning in Figure F.1 shows all project documentation that will be made during this project. This documentation planning consists of this document, a paper, and a self-reflecting document.

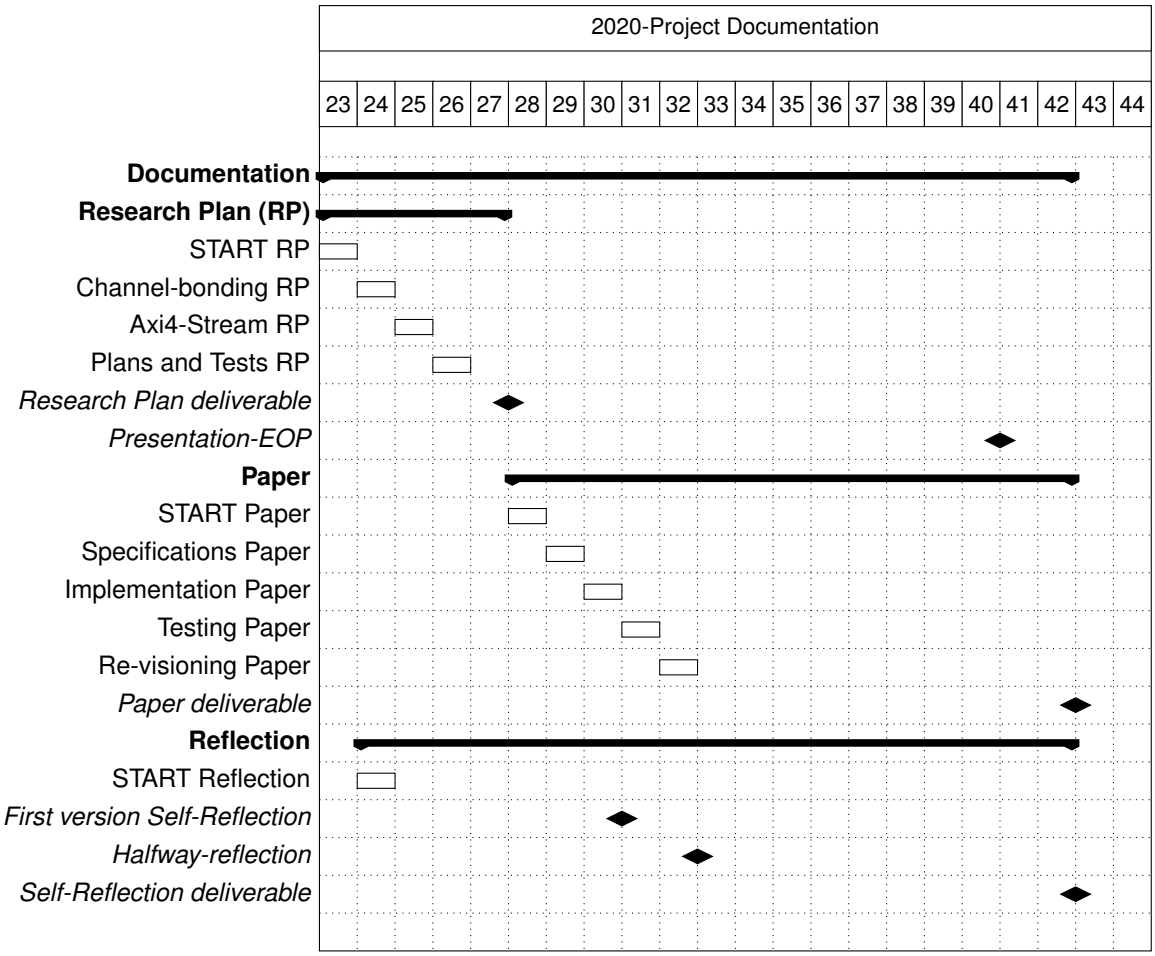


Figure F.1: All kinds of documentation in a timeline

In Figure F.2 another planning is shown, called project planning. Project planning contains everything that isn't documentation. It consists of AXI-Stream, Channel-Bonding, Transceiver-remapping and a lot of Benchmarking and simulation. Everything has two weeks except the benchmark and simulation block. The benchmark and simulation block will mainly consist of making revisions. Next will be Appendix F.1 about some additional preparation that is made for the start of this project.

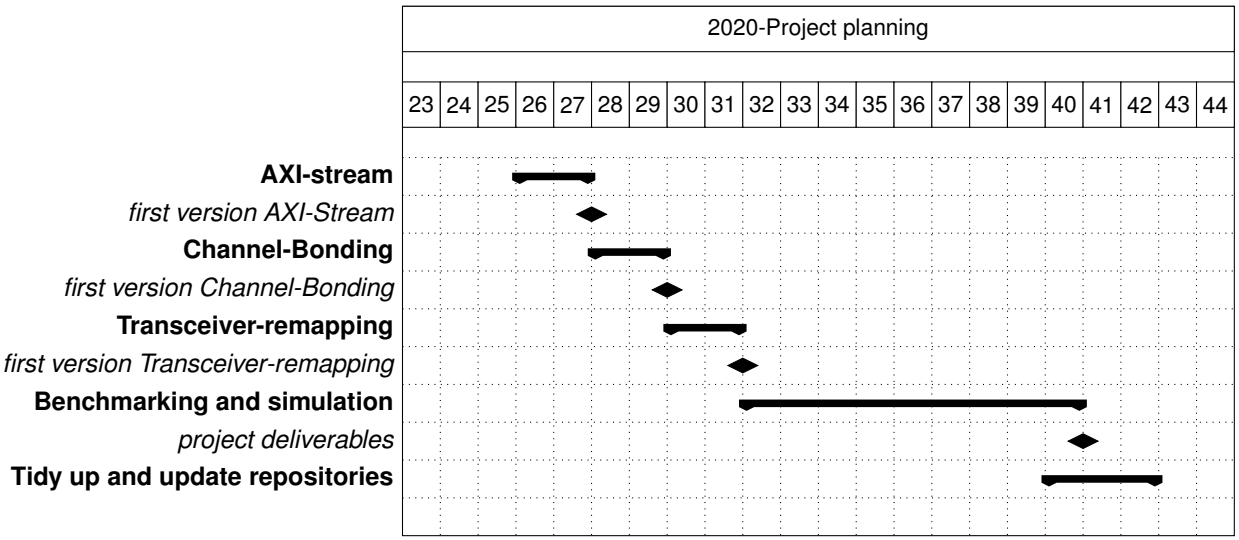


Figure F.2: All project components (that aren't documentation)

F.1 Preparation

Lot of things need to be prepared. One of the things that is done before start, is error and warning checking with Sigasi. Sigasi can help with systems using VHDL, Verilog or System Verilog. Now it is used for its capability to check syntax and other points of interest. When the project is loaded with Sigasi, the following result had come out. (See Figure F.3) There where a lot of errors and warnings. But it needs to be said that not all warnings need to be fixed and can safely be ignored. For example some warnings are intentional and have no negative effect. The idea is to use Sigasi to easily find and clean up old unused VHDL lines and fix as many errors and warnings as possible.

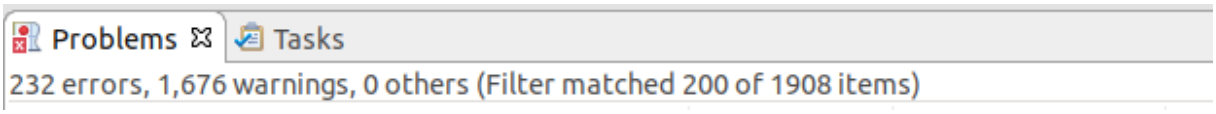


Figure F.3: Cleaning with Sigasi.

F.1.1 AXI-Stream FIFO

During cleaning a start has been made on implementing the AXI-Stream fifo. The main reason is that the old version of Interlaken used FIFO's that now could be replaced. The AXI-Stream will later be fully utilized and its internal logic is not yet finished. For now it just acts as a quick replacement.

F.1.2 Project building with Tcl files

To build the Sigasi project and the Vivado project environment .tcl files are used. These Tcl files help with setting up a Vivado project with all the dependencies that are described in the Tcl files. This allows for quick reproduction of a project, rebuilding of a project and clear management of all source files that are used. The

.tcl are distributed per part/module of the FELIX project. This means that there is, for example, an Interlaken-fileset.tcl and a housekeeping-fileset.tcl. Each containing paths to their source files. After doing all previously mentioned preperation the following result were met with Sigasi.(see Figure F.4)

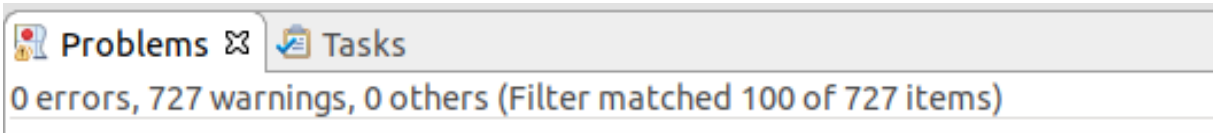


Figure F.4: After cleaning with Sigasi.

F.1.3 Integrity test after cleaning

The result of fewer errors and warnings are not a guarantee that all the VHDL firmware still works accordingly. So a last step will be necessary to conclude the preparation. A very basic testbench will be used to find out if a simple pattern will go through the system and will be replicated at the output. If this requirement isn't met then some extra work will be performed to bring the firmware to this stage. The testbench will only consist of a clock, some initial signal values and a data pattern, each controlled using wait periods. In Figure F.5 the s-axis and the m-axis signals are shown. The s-axis channel null to four are the data in, here set with the hexadecimal number '00F00F00F00C00C0'. After some time the same value shows on the m-axis channels. This means that the data goes through the device without being wrongly changed. In reality, many more signals were watched (of the framing burst, framing meta, scrambler and encoder) to test if the input gets processed correctly, but to keep it simple they will not be discussed.

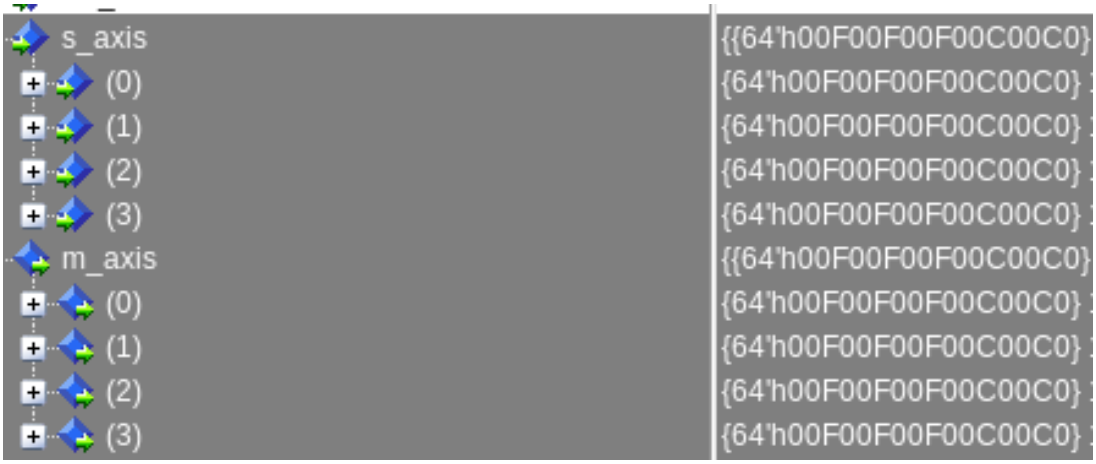


Figure F.5: Test-bench after cleaning

F.2 Conclusion

Now it's time to give a brief conclusion of this research plan by naming the current state of different parts of Interlaken: Core1990 V2 has been cleaned and edited. An AXI-Stream FIFO declaration has replaced the old FIFO and still needs some work to be fully operational. A test-bench showed that the general workings of the Interlaken application are still intact after the changes. At the moment not all signals in the Interlaken application are utilized and some still need testing. By duplicating one lane to four, four channels are made. But the Channel-bonding is not yet implemented, and lanes are yet to be synchronized. From this status the project will continue according to the planning mentioned in the Appendix (F).

In the end the main goal (Chapter2) could be reached by implementing Channel-Bonding and AXI-Stream in a new version of Interlaken. As this would increase the per-channel transfer rate from 10Gb/s to 25Gb/s will further increase the link speed. The main part of the project will be 'Benchmarking and simulation' (see planning Figure F.2). As each new made block needs to work correctly and with speed. In the end a new version of Core1990 V2 will be achieved, and will probably be named Interlaken.

Appendix G

ALL SPECIFICATIONS

In this appendix most of the specifications are listed in tables. The specifications include each module of the TX and RX side. Also the burst and meta formats are listed in this Appendix.

G.1 FRAMING BURST

The specifications of the Framing Burst are shown in table G.1.

Framing Burst	*Descriptions*
Performs a CRC-24 check	*To detect framing errors, by testing the data integrity.*
Frame data into bursts (BURST FORMAT)	*Because Interlaken is the used prototcol, the burst format must be implemented according to the Interlaken protocol specifications. [14]*
Allows AXI-Stream input	*AXI-Stream is a general data interface. The main reason for AXI-Stream is to connect two different devices faster by implementing AXI-Stream as a new standard.*
Allows FlowControl	*Flowcontrol is necessary to manage the data transmission rate between two devices. This prevents loss of data by letting the sender know that the receiver is (near) full.*
Uses Gearbox compensation	*To prevent bottlenecks on high transmission speeds (Serdes) gearboxes are needed. These can be made faster than serially processing the data stream inside the available chips [2]*
Connected to input of Framing Meta	*The Framing-Burst is the first part of Interlaken and is used to achieve the Burst-Format. After the Burst-Format the Meta-Format must be handled.*

Table G.1: Specifications Interlaken TX Framing Burst

G.2 Framing Meta

The specifications of the Framing Meta are shown in table G.2.

Framing Meta	*descriptions*
Uses Meta control Format	*The Meta control format contain the following words: SYNC, SCRAM and SKIP. These are necessary to signal certain states to the receiver side.*
Has a GearBox	*To make use of the Gearbox every part of TX needs it.*
Performs a CRC-32 check	*The CRC-32 check allows tracing the error back to the origin lane. Therefore it's much easier to find the causing lane. [2]*
Receives Healthlane and Interface	*The input Healthlane and HealthInterface are used to frame the lane and interface health status to the stream (bit 33:32)*

Table G.2: Specifications Interlaken TX Framing Meta

G.3 SCRAMBLER

The specifications of the Scrambler are shown in table G.3.

Scrambler	*descriptions*
Scrambles the data	*Data Scrambling is necessary to minimize long ones and zero's on the transmission lines. A constant voltage could be a harmful phenomenon while synchronizing a communication system*
Must be independent synchronous	*A self-synchronous scrambler doesn't need constant sync which could result in data corruption. A independent synchronous will minimize this problem*
Has a Gearbox	*To make use of the Gearbox every part of TX needs it.*

Table G.3: Specifications Interlaken TX Scrambler

G.4 ENCODER

The specifications of the Encoder are shown in table G.4.

Encoder	*descriptions*
Keeps track of running disparity	*The Encoder needs to invert to cancel possible DC unbalance, created when a one or zero is over presented. Especially necessary on systems with high transmission speed as dc unbalance can accumulate more quickly*
Needs achieve a locked state	*The encoder tries to find 64 consecutive legal sync headers at the same position, when found a locked state is achieved. When not, there is syncing problem. This is a extra precaution, because of the high transmission speed, if left unchecked sync problems can occur rather fast*
Has a Gearbox	*To make use of the Gearbox every part of TX needs it.*

Table G.4: Specifications Interlaken TX Encoder

G.5 IDLE/BURST CONTROL WORD FORMAT

The specifications of the Idle/Burst Control Word Format are shown in table G.5

Field	*Bit Position*	*Function*
Inversion	*66*	*Used to indicate whether bits [63:0] have been inverted to limit the running disparity 1 = inverted, 0 = not inverted*
Framing	*65:64*	*64B/67B mechanism to distinguish control and data words a 01 indicates data, and a 10 indicates control*
Control	*63*	*If set to 1, this is an Idle or Burst Control Word if 0, this is a Framing Layer Control Word*
Type	*62*	*If set to a 1, the channel number and SOP fields are valid and a data burst follows this control word (a Burst Control Word) if set to a 0, the channel number field and SOP fields are invalid and no data follows this control word (an Idle Control Word)*
SOP	*61*	*Start of Packet. If set to a 1, the data burst following this control word represents the start of a data packet if set to a 0, a data burst that follows this control word is either the middle or end of a packet*
EOP	*60:57*	*This field refers to the data burst preceding this control word. It is encoded 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-PacketAll other combinations are left undefined.*
Reset Calendar	*56*	*If set to a 1, indicates that the in-band flow control status represents the beginning of the channel calendar*
In-Band Flow Control	*55:40*	*The 1-bit flow control status for the current 16 calendar entries if set to a 1 the channel or channels represented by the calendar entry is XON, if set to a 0 the channel represented by the calendar entry is XOFF*
Channel Number	*39:32*	*The channel associated with the data burst following this control word set to all zeroes for Idle Control Words*
Multiple-Use	*31:24*	*This field may serve multiple purposes, depending on the application. If additional channels beyond 256 are required, these 8 bits may be used as a Channel Number Extension, representing the 8 least significant bits of the Channel Number. If additional in-band flow control bits are desired, these bits may be used to represent the flow control status for the 8 calendar entries following the 16 calendar entries represented in bits[55:40]. These bits may also be reserved for application-specific purposes beyond the scope of this specification.*
CRC24	*23:0*	*A CRC error check that covers the previous data burst (if any) and this control word*

Table G.5: Idle/Burst Control Word Format (From the Interlaken Protocol Definition PDF) [12]

G.6 META FRAMING WORD FORMAT

The specifications of the Meta Framing Word Format are shown in table G.6

Field	*Bit Position*	*Function*
Inversion	*66*	*Used to indicate whether bits [63:0] have been inverted to limit the running disparity 1 = inverted, 0 = not inverted*
Framing	*65:64*	*64B/67B mechanism to distinguish control and data words a 01 indicates data, and a 10 indicates control*
Control	*63*	*If set to 1, this is an Idle or Burst Control Word if 0, this is a Framing Layer Control Word*
Block Type	*62:58*	*Used to match specific block type, types are SYNC 011110, SCRAM 001010, SKIP 000111 and DIAG 011001*
Block Type Specific format	*57:0*	*The Specific format of the meta frame word. They are:* *SYNC h0F678F678F678F6,* *SCRAM H228000000000000,* *SKIP h21Eh1Eh1Eh1Eh1Eh1E,* *DIAG h000000 + status-bits + CRC32*

Table G.6: Meta Control Word Format

Appendix H

Transceiver (TX) and Receiver (RX)

Interlaken is part of the firmware for FELIX. It's a point-to-point protocol. The Interlaken protocol is an high speed channelized chip to chip interface which is used to achieve the transmission speed (25Gb/s per channel) mentioned in the main goal in chapter 2. Interlaken itself consist of an transmitter (TX) and receiver (RX) connected to an transceiver IP-core. The TX and RX itself consist of four modules. This Appendix will be sectioned per module. In Appendix E a more detailed description about Interlaken in general. This Appendix will now continue with all the internal parts of the TX and RX, which are each optimized, cleaned and changed since the Core1990 version of Interlaken. (Note: To prevent this appendix to be extremely long not all internal parts will have a before and after,but just an after version!)

H.1 Framing burst (TX)

The Framing Burst creates a packet out of the data stream. When an end of packet signal or the maximum packet size is reached, the Framing Burst will add a control word to the end of the packet (EOP), used to flag the EOP and will start on a new packet. The Framing burst has Axi-Stream (master) inputs connected to a slave device and outputs to the Framing Meta. When there is no data available the Framing Burst will stay idle and sends idle words.

H.1.1 State-decoder and State machine

From the previous description some states can be derived that can be used to manage the logic inside of the Framing Burst. Firstly the state IDLE. IDLE because the Framing Burst isn't always active. Secondly a DATA state, when the Framing Burst uses the input to make the packet. Thirdly a (CONTROL) WORD state, to manage the EOP words and CRC-24 check.

The main reason why the State-decoder and State machine are being discussed, is because the Core1990 V2 version of the framing burst has the following states implemented: IDLE, DATA, WORD, EOP SET, EOP FULL, FILL, EOP EMPTY, IDLE SET, IDLE FULL and IDLE EMPTY. Considering the three states (IDLE, DATA, WORD), many states used in Core1990 V2 are unnecessary and will be optimized. (optimizing is part of the main goal chapter 2)

H.1.1.1 STATE-DECODER

First look at the original state-decoder of Core1990 V2, shown in Figure H.1. On the right side the state IDLE can be found. IDLE is the initial state of this state-decoder. The role of the state IDLE, DATA and WORD were already explained in section H.1.1. Therefore let's move on to the four states EOP EMPTY, FILL, EOP SET and EOP FULL. When an end of packet is signaled. The state can reach EOP SET. When in the EOP SET state, an EOP word is added to the packet. In the EOP FULL state the EOP is communicated to the CRC-24 logic. As each packet will get an CRC-24 check added to the packet.(see as addition Figure E.1 from Appendix E, about the burst-format). The STATE EOP EMPTY does exactly the same as EOP FULL. The only difference is that EOP FULL is reach when the maximum packet size is reached and EOP EMPTY when the slave device communicates an EOP. Then the state FILL is left. FILL fills up the packet with idle words when it's below the minimum packet size. After regulating the EOP with these four states the state-decoder returns to the IDLE state.

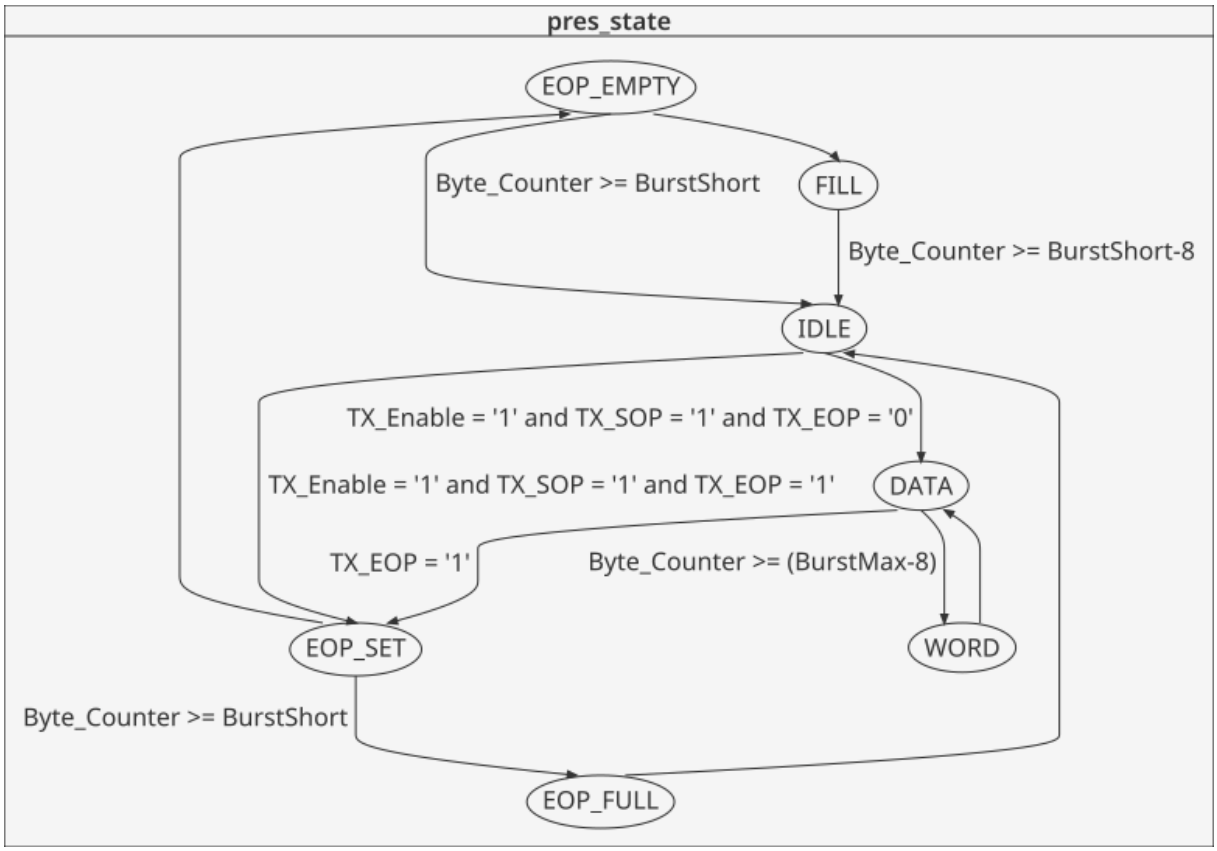


Figure H.1: Sigasi export of framing burst state-decoder, original version

Now let's determine what to do with the four states and work towards the result shown in Figure H.1. The framing burst starts in the IDLE state. When valid data is available it needs to move to the DATA state, where it will start framing the data. When an EOP is reached there is no need to switch states. Inside the DATA state the same signals can be used to evaluate if the DATA state frames data or switches to finishing a packet with a control word and signals to start a CRC-24 check. When the packet isn't yet the minimum size a the third situation will be met were idle words will be added to the packet. The WORD state is still used when a maximum packet size is reached. The end result is visible in Figure H.2

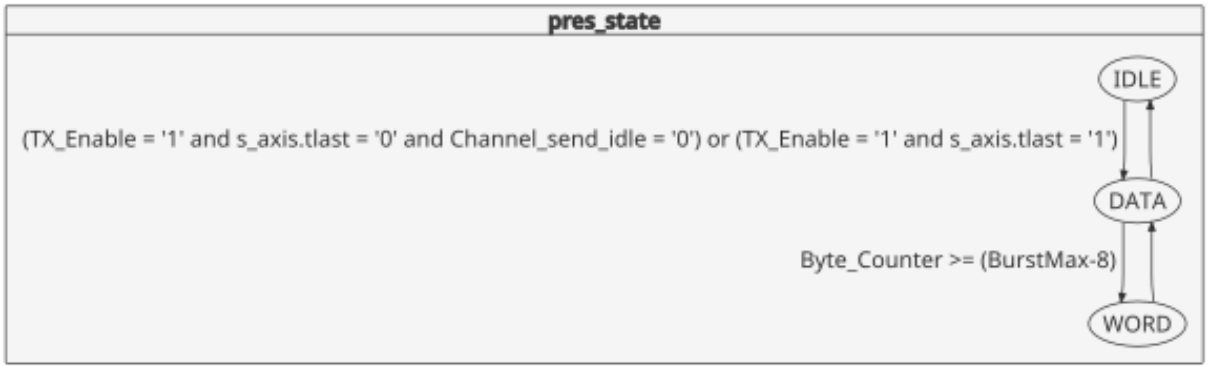


Figure H.2: Sigasi export of framing burst state-decoder, after optimization

H.1.1.2 STATE MACHINE

After compressing the state-decoder, a couple of things stand out. When looked at Figure H.2, you can see that only a single condition remains to go from the DATA state to the WORD state. So the word state can be merged with the DATA state. And with two states left and no reason to wait (except for the clock signal), the state machine could entirely replaced with a clocked process which contains no states.

In Figures H.3 and H.4 the resulting VHDL description is shown. The process is called output (line 95) and is sensitive to the rising edge of the input clk. (line 97). When BURST-tready is LOW the process sends a idle control word. (lines 101-123) Before the control word is definite there are still three more conditions that needs to be checked. When an EOP has occured the last clk-cycle a EOP word is send. (line 115). Or when no EOP has occured and the TX Enable from the transmitter is HIGH, a burst control word is send. (lines 116-121). The burst control word has two variations. The first one has bit 62-61 as "10" which means: a data burst that follows this control word is either the middle or end of a packet.(lines 116-118) And when the bits contain "01" it means: the data burst following this control word represents the start of a data packet (line 120);

framing_burst.vhd

```
94
95      output : process (clk) is
96      begin
97          if rising_edge(clk) then
98              CRC24_RST <= '0';
99              if( meta_tready = '1' and Gearboxready = '1' ) then
100                 BURST_tready <= '1';
101                 if BURST_tready = '0' then --This means that it was indicated in the data state to send a Burst co
102                     Byte_Counter <= 8;
103                     CRC24_RST <= '1';
104                     CRC24_TX(66 downto 64) <= "010"; --inversion and framing
105                     CRC24_TX(63) <= '1'; --Control
106                     CRC24_TX(62) <= '0'; --Type
107                     CRC24_TX(61) <= '0'; --SOP
108                     CRC24_TX(60 downto 57) <= x"0";--EOP_Format
109                     CRC24_TX(56) <= '0'; --Reset Calendar bit
110                     CRC24_TX(55 downto 40) <= FlowControl; --Per channel flow control, 1 means Xon, 0 means Xoff.
111                     CRC24_TX(39 downto 32) <= x"01"; --Channel number, TODO: Insert insert i from for generate cha
112                     CRC24_TX(31 downto 24) <= x"00"; --Multiple-Use field
113                     CRC24_TX(23 downto 0)  <= x"000000"; --CRC24 field
114                     if(SendEOP = '1') then
115                         CRC24_TX(60 downto 57) <= '1' & TX_ValidBytes_s;--EOP_Format, converted from tkeep.
116                     elsif(TX_Enable = '1') then
117                         CRC24_TX(62) <= '1'; --Type
118                         CRC24_TX(61) <= '0'; --SOP
119                         if (s_axis.tvalid = '1') then -- Indicates the start of data flow
120                             CRC24_TX(61) <= '1'; --SOP
121                         end if;
122                     else
123                         CRC24_TX(62) <= '0'; --Type
124                         CRC24_TX(61) <= '0'; --SOP
125                     end if;
```

Figure H.3: Framing-burst output-process (Part 1 of 2)

Now let's go back to BURST-tready. When BURST-tready isn't LOW, a data word is send. This happens in lines 127-141. When a data word is send the byte-counter is increased. Also two condition are tested. Firstly if a control word needs to be send and secondly if a EOP occurred.

```
126                 else
127                     CRC24_TX(63 downto 0) <= s_axis.tdata;
128                     CRC24_TX(66 downto 64)<= "001"; --Data word
129                     Byte_Counter <= Byte_Counter + 8;
130                     SendEOP <= '0';
131                     if ( (Byte_Counter = (BurstMax-8)) or (s_axis.tlast = '1') or (Channel_send_idle='1') ) then
132                         BURST_tready <= '0';
133                     end if;
134                     if (s_axis.tlast = '1' and Byte_Counter <= BurstShort) then
135                         SendEOP <= '1';
136                         BURST_tready <= '0';
137                     end if;
138                 end if;
139                 if(s_axis.tlast = '1' or Channel_send_idle = '1') then
140                     BURST_tready <= '0'; --Indicate that we are going to send a Burst word in the next clock cycle
141                 end if;
142             end if; --s_axis_tready
143         end if; --clk
144     end process output;
145
146 end architecture framing;--(from 491 lines to 146)
```

Figure H.4: Framing-burst output-process (Part 2 of 2)

H.1.2 AXI-STREAM FIFO (TX)

Not only the processes of the Framing Burst have been adjusted. The AXI-Stream interface implementation also changed the in and outputs of the Framing Burst. (Appendix 3) Before in Core1990 V2 a FIFO was used which themselves used many control signals. These FIFO's and signals are replaced by the AXI-Stream interface. The Framing Burst now has the input s-axis and the output s-axis-tready. The s-axis is a record type, containing the signals: Tdata, Tvalid, tlast, tkeep and tuser. In Figure H.11 an overview of the Framing Burst module is shown. On the left side the signal (record type) S-axis and on the right S-axis-tready which is an output signal.

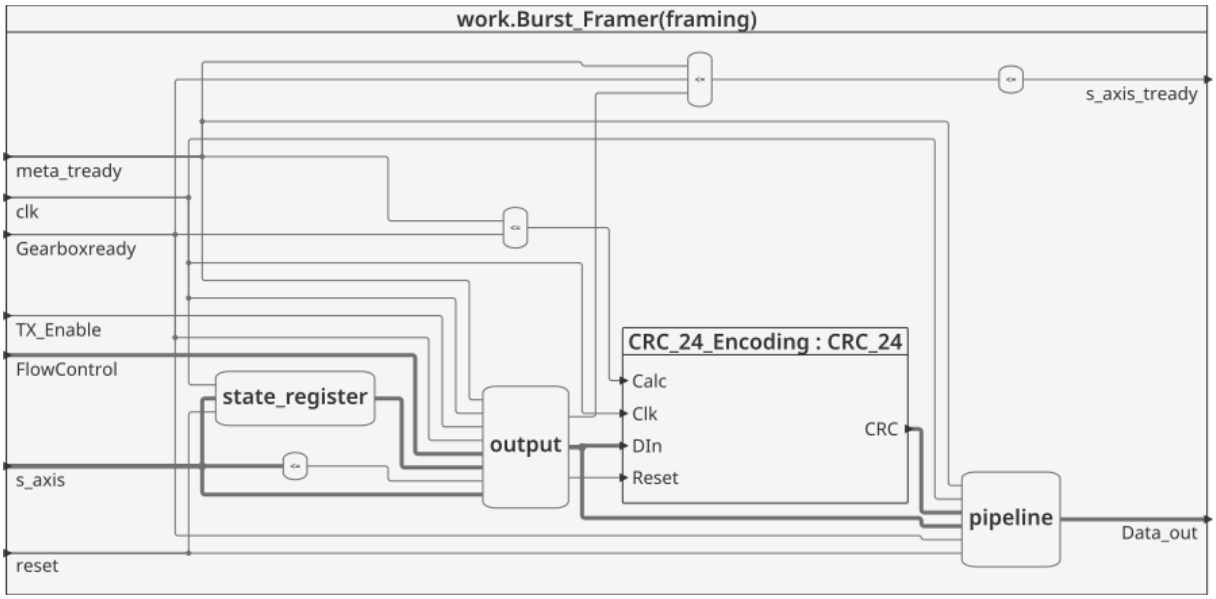


Figure H.5: Framing-burst module

The four larger blocks (in Figure H.11) are: state-register, output, CRC24-encoding and pipeline. Each of these blocks manages a sub part of the framing burst and will be discussed next.

H.1.2.1 STATE-REGISTER PROCESS OF THE FRAMING BURST

In the State-register block the input signal S-axis-tkeep is decoded to determine which bytes to keep or to be invalid and thrown away. When the Framing Burst frames an End of Packet the decoded bytes are added to the last frame, to mark the valid bytes in the packet.

H.1.2.2 OUTPUT PROCESS OF THE FRAMING BURST

The Output process could be seen as the main part of the Burst Framing. This process uses the input signals and determines which burst-format has the be framed next. This could be SOP,DATA,IDLE, SKIP and EOP. The Output process is not yet the output of the framing burst. All the output first needs to go through the pipeline.

H.1.2.3 PIPELINE PROCESS OF THE FRAMING BURST

The Pipeline process is used to "pipe" the frame from output to the Data-out signal. The pipeline is also used to "pipe" the CRC-24 data to the frame if this is required.

H.1.2.4 CRC-24 PROCESS OF THE FRAMING BURST

In the CRC-24 process a crc-24 encoder is used to determine the new checksum. It functions by passing data in the DIn signal, which will result in a new checksum on the CRC (output signal).

H.2 Framing Meta (TX)

The Framing Meta stage handles all of the meta framing in Interlaken. It is connected between the Framing Burst module and the Scrambler module. The Framing Meta adds Meta frames to the packet which are used for packet health nonrecognition and lane synchronization. The Meta frame uses the meta frame format, including SKIP, IDLE, DATA, SYNC, DIAG and SCRAM words. In addition to the meta framing a CRC-32 checksum is generated and added to the data packets.

H.2.1 Framing Meta Module

In Figure H.6 an global overview of the Framing Met Module is shown. The meta framing doesn't require much outputs. It only needs to pass the new frame on the Data-Out signal, and gives a Boolean FIFO-read signal to communicate that the data is ready. To make a frame the module makes use of a state-decoder, state-register, control-pipeline, output, hdr-or, diagnostic and crc-32 encoder process.

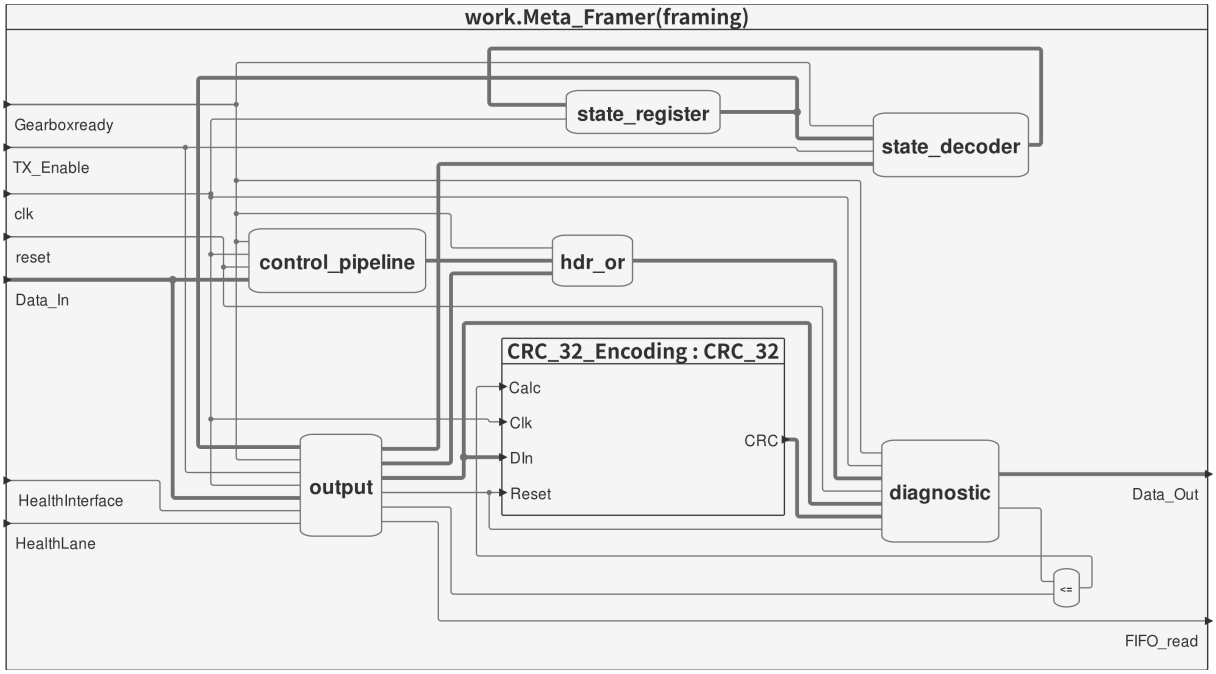


Figure H.6: Framing-meta module

H.2.1.1 State-register process of the Framing Meta

The state register is a very basic progress. It's only function is therefore to change to the next state on a rising edge of the clock signal. The next state is determined in the state-decoder.

H.2.1.2 State-decoder process of the Framing Meta

The State decoder is standard in IDLE. When the transmitter is enabled it will go to the SCRAM state and next after to the SKIP state which then continues to the DATA state. It will remain in the data state until a full packet size is reached. When this is the case the following chain will be activated: P1, P2, P3, DIAG and IDLE. And when there is more data it will chain into P1, P2, P3, DIAG, IDLE , SCRAM, Skip and return to DATA. The logic for each state is defined in the Output process.

H.2.1.3 Output process of the Framing Meta

In the Output process the states of the state-decoder are used which are acquired from the state-register. When the state is IDLE the framing meta makes a idle frame according to the meta framing format. Same happens for the Scram word in the SCRAM state and the SKIP word in the SKIP state and DATA frames a DATA. P1, P2 and P3 added to provide a static delay of three states or better said three clock cycles, before the diagnostic state is reached. After, the DIAG state will frame all kind of packet and lane-health data in the DIAG word.

H.2.1.4 HDR-OR process of the Framing Meta

The HDR-OR is a small process that uses a or to determine the right header to use for the meta frame.

H.2.1.5 Control-pipeline process of the Framing Meta

The Control pipeline is used to delay the hdr before it is evaluated by the hdr-or process. The delay is three clockcycles

H.2.1.6 CRC-32 process of the Framing Meta

The CRC-32 process is a CRC-32 encoder entity which is past data to determine and give back a CRC32-checksum.

H.2.1.7 Diagnostic process of the Framing Meta

The diagnostic process acts as another pipeline before data leaves through the data out signal. It determines if the CRC-32 must be added to the frame, which hdr to use from the hdr-or and combines it with the frame setup from the output process. All complete frames will then move to the scrambler module.

H.3 SCRAMBLING (TX)

The scrambler module uses xor logic in combination with bit shifts to scramble the incoming data frames. In Figure H.7 is a zoomed out version of the module shown. The main reason why it is still shown in such a zoomed out version is to show that the scrambler module is mainly xor logic.

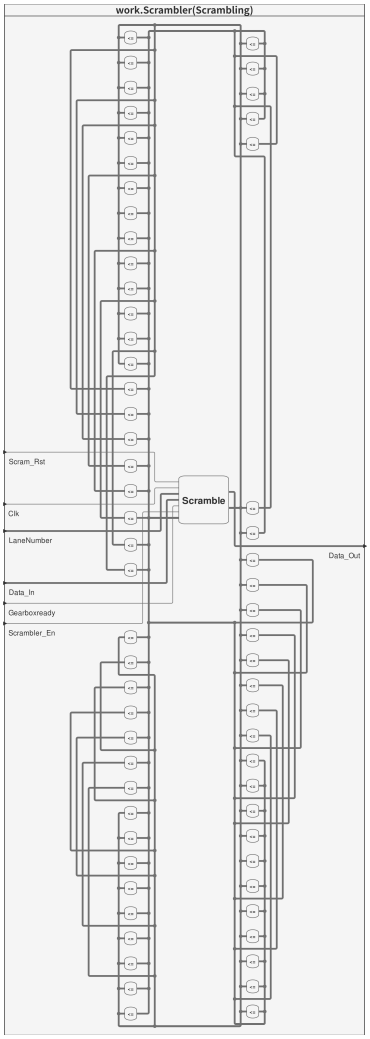


Figure H.7: scrambler module

The only used process is called scramble. This process uses the xor logic to scramble a frame and pass it on the data out signal.

H.4 Encoding (TX)

The encoder is the last step of the transmitter side. It is used to invert whole data frames to control the ones and zero's that are send. When inverted the MSB of the frame becomes one and all other bits are inverted. In Figure H.8 the module overview is shown.

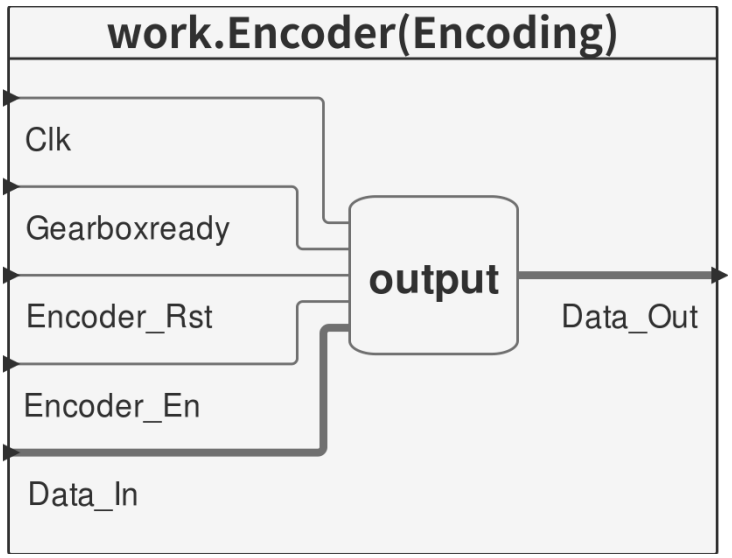


Figure H.8: Encoder module

H.5 Decoding (RX)

The decoder is the first module on the receiver side of interlaken. The decoder receives its input from the transceiver. It is assumed to be a standard that all incoming data is encoded. Hence the use of a decoder that flips the data back when necessary. In Figure H.9 an overview of the module is shown.

When the module is first used it needs to be initialized as it will try to synchronize its lane. When synchronization is reached it enters a locked state, which allows data to be processed.

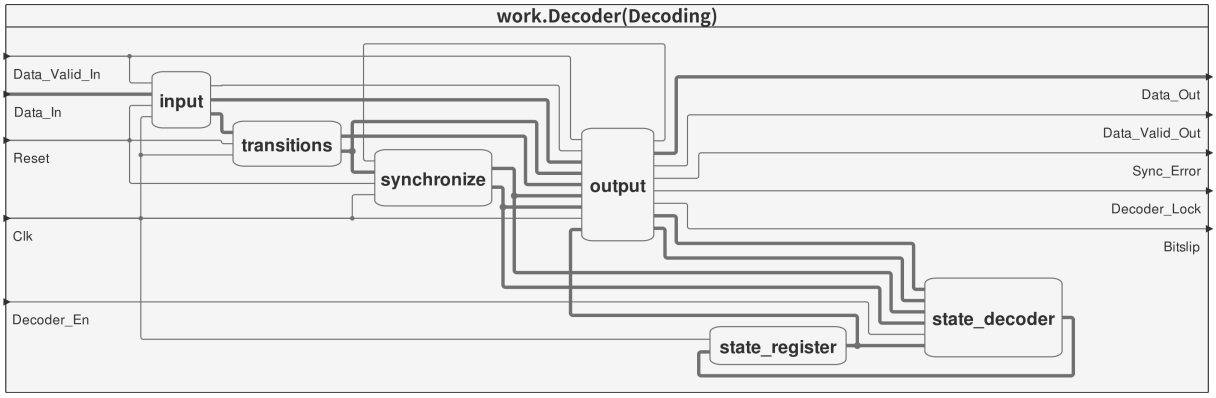


Figure H.9: Decoder module

H.5.1 Decoder module

The module from Figure H.9 is as previously mentioned connected on the input to the transceiver and on its output to the Descrambler. To process the data to a state it's ready to be interpreted by the Descrambler a few processes are in place. Next these process are each briefly summarized.

H.5.1.1 State-register process of the Decoder

Each clockcylce the State-register changes the present state to the next state, which is in turn perceived from the state-decoder.

H.5.1.2 State-decoder process of the Decoder

The state decoder tries to determine which state is next after the current states. The different used states are: IDLE, SYNC and LOCKED. The flow is relatively simple. The Decoder starts in IDLE, when enabled it goes to SYNC. When in SYNC state it stays in SYNC. In the SYNC state it can return to idle when syncing went wrong or it can continue to the LOCKED state when a lock is achieved. In the LOCKED state in then stays locked until it gets an out of sync signal, which returns the state to IDLE, so the whole process can be repeated.

H.5.1.3 Input process of the Decoder

The input process handles if data is allowed to enter the decoder. Each clockcycle it evaluates if the data in flow is valid, when it isn't it holds the old data in place.

H.5.1.4 Transitions process of the Decoder

The transitions process is the process that flips the data back. It is done by using xor logic just as how it's encoded, but then in reverse.

H.5.1.5 Synchronize process of the Decoder

The Synchronize process tries to find a sync. This is done by counting the passing frames and the passing meta frame SYNC words. When the SYNC words are two times found on the exact same location, which means the same distance between SYNC words, then it can be assumed the lane is synced. When this isn't the case the synchronize process determines how much the lane should be transitioned and tries to sync again with the new transition.

H.5.1.6 Output process of the Decoder

The output process is the process that glues all previously described processes together. This is the process that makes use of the states decoded from the state decoder and passed on through the state-register. The IDLE state can be seen as a sleep or reset mode in which certain data gets set back to a default value, waiting for the next state. In the SYNC state the output process works together with the synchronize process. This is done by resetting the sync-counter and helping with the shift and determination when something went wrong. Then there is LOCKED in which data is allowed to go to the data out signal. Here the found header is used to inverted the data before it is passed to the De-Scrambler.

H.6 DESCRAMBLING (RX)

The Descrambler shown in Figure H.10 uses the same calculation as the scrambler but in reverse. Hence the many xor logic in Figure H.10.

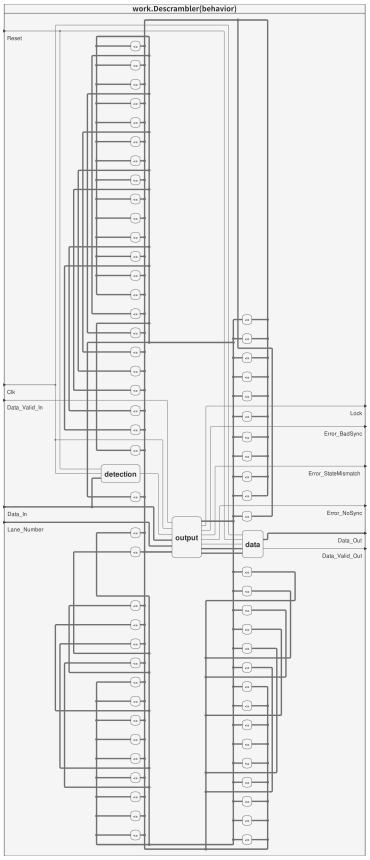


Figure H.10: De-Scrambler module

After De-Scrambling the data passes on to the De-Framing Meta.

H.7 DE-FRAMING META (RX)

The De-Framing Meta is mainly used as a checking module. During De-Framing, a CRC32-checksum is calculated and then compared to the CRC32-checksum from the passing packet. If they are exactly the same data is passed on to the next module. If the CRC32-check failed, the data gets marked with a CRC32-error bit and still continues onwards.

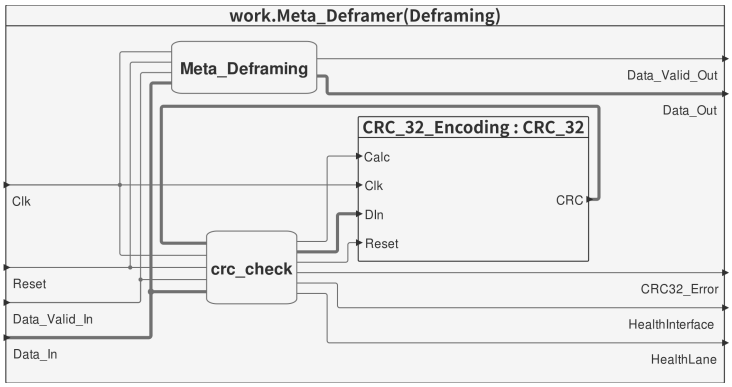


Figure H.11: De-Framing-Meta module

H.7.1 The De-Framing Meta module

To check the meta-frame for possible errors in the data, three processes are performed. Next these three processes are summarized.

H.7.1.1 CRC-Check process of the De-Framing Meta

The CRC-Check process handles the CRC data. This means that the process waits for data, to search for the CRC32 checksum to later compare to a newly calculated one. To do this it holds onto the CRC32 checksum till it is used and when falsely compared it signals an CRC32-error. The calculated CRC32 gets calculated by the CRC32 encoding process.

H.7.1.2 CRC32 Encoding process of the De-Framing Meta

The CRC32 Encoding process is a process that passes data to a CRC32 encoder to calculate a new CRC32-Checksum. This checksum is used during the CRC-check process.

H.7.1.3 Meta-De-Framing process of the De-Framing Meta

The Meta-De-Framing process is used as a pipeline to manage if data can be put on the output signal. It makes use of a data-valid-out signal to communicate that the output data is ready to be read by the next module, the De-Framing Burst.

H.8 De-Framing Burst (RX)

H.8.0.1 AXI-Stream FIFO (RX)

The De-Framing Burst changed a bit because of the AXI-Stream fifo. Now all the old FIFO signals have been removed and some AXI-Stream signals took their place. In figure H.12 are the signals shown. A m-axis-tready signal, used for the AXI-Stream handshake, is now an input signal of the De-Framing Burst. At the output the signal m-axis had replaced some signals. The m-axis signal is a record type containing: Tdata, Tvalid, tlast, tkeep and tuser.

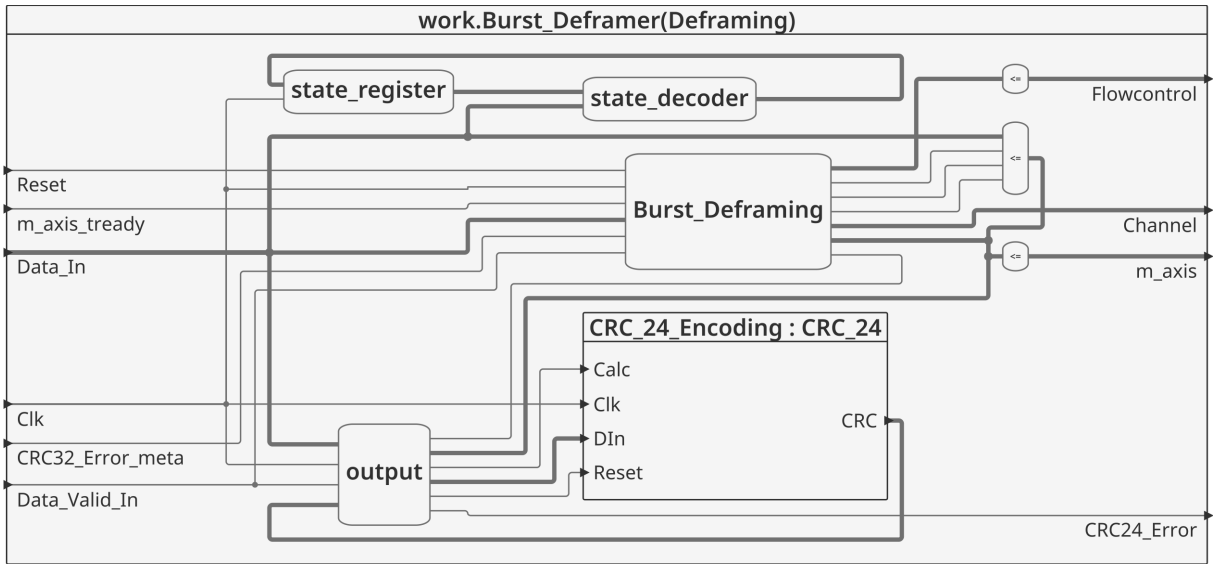


Figure H.12: De-Framing-burst module

H.8.0.2 State-register process of the De-Framing Burst

The state-register is a clocked process that changes the state to the next decoded state, decoded by the state decoder.

H.8.0.3 State-decoder process of the De-Framing Burst

The state decoder has two states. The states are called IDLE and CRC. The state-decoder starts in IDLE and waits for a SOP (Start Of Packet). When a SOP is found it's state changes to CRC. When a SOP or EOP is recognized the state will return to idle.

H.8.0.4 CRC-24 Encoding process of the De-Framing Burst

The CRC24 Encoding process is used to send data to perform CRC-24 check with an CRC-24 encoder. The CRC-24 check returns CRC-24 checksum.

H.8.0.5 Output process of the De-Framing Burst

The Output process manages if the incoming data is valid, if the CRC-24 check gives an error and passes these values to the Burst-Deframing process.

H.8.0.6 Burst-Deframing process of the De-Framing Burst

The Burst-Deframing process adds all partial signals together. (data signals ,errors signals, and others) And uses the Axi-stream interface to communicate outgoing data to a slave device.

H.9 Other mentionable signals

The modules and processes also use some signals that are quite important.

H.9.0.1 GearboxReady signal

The Gearboxready signal is one of those signals. Gearboxready allows interlaken, which is a 64/67 bit protocol to compensate for the extra three bits that make 64/64+3. The main reason for this compensation is because of the transceiver. The transceiver can handle 64 bits at a time, and the Gearboxready helps interlaken to handle the three bits in a way the transceiver can use it properly. This is done by shifting the bits with the gearbox and when ready, the interlaken modules get the gearbox signal. The signal is then used to let the module wait until the gearbox signal changes back.

H.9.0.2 Health signals

Another two signals that are important are the healthLane and healthInterface signals. The healthLane signal signals throughout the lane that the lane is still operating. This mainly suggests that the lane synchronization is still in place and that the lane is initialized properly. The healthInterface represents as it's name suggests the health of the whole interface. Both signals are part of the data stream and added to each packet.

H.9.0.3 FlowControl Signal

The last signal to be mentioned is the flow control signal. The flow control signal is an external and internal signal that is used in the Burst framing and Burst De-framing. It contains information about which bits to keep and which to throw away. Interlaken itself doesn't throw away data using this bits, instead it only marks the bits for later interpretation.

Appendix I

Terms, Definitions and Glossary

List of Tables

List of Figures

I.1 GLOSSARY

Aligned All lanes are in sync. 10

All lanes aligned signal Flags that all lanes are aligned. 19

An optional signal of the AXI-Stream An optional signal of the AXI-Stream. 6

Array A list of elements. 7

ATLAS A Toroidal LHC Apparatus. i, iii, 1, 2, 4, 6

AXI Interface A standardized data stream protocol. 10

AXI-FIFO The FIFO of the AXI-Stream Interface. v, 23

AXI-Stream A standardized data stream protocol. i, iv, v, 1, 4, 6, 9–12, 22, 23

AXI-Stream FIFO The FIFO of the AXI-Stream Interface. 1, 4, 6, 7, 10, 11, 17, 22

AXI-Stream Interface A standardized data stream protocol. i, iii, iv, 1, 2, 4–12, 21–23

AXI-Stream-Package A package written to define the record types used by the AXI-Stream. 7

AXI4-Light An extended AXI protocol with totally different features. 14

AXI4-Light Interface An extended AXI protocol with totally different features. 14, 24

AXIS64-FIFO The name of the 64-bit AXI-Stream FIFO. 8, 9

Bit A zero or one. 7

Bits More than one bit. 7

Buffer A storage to hold a certain amount of data to prevent an overflow. 6

burst idle a data-less word to keep a contious stream. 18

Burst-Framer A module of Interlaken that frames the data into bursts. 19

Burstmax Error Flags that packet is larger than packet-length. 15

Byte-counter counts the bytes that pass on the lane. 19

CERN Conseil Européen pour la Recherche Nucléaire: European Organisation for Nuclear Research. i, iii, 2

Channel-Bonding Using four lanes parallel in sync, for a single data stream. i, iii–v, 1–5, 9, 10, 15–22, 24, 25

Core1990 Core1990 is the name of the FELIX version of the Interlaken Protocol. 2, 9, 10, 22

Core1990 V2 Second version of Core1990. 2, 4, 6, 13, 17

CRC24 Error Flags that the data integrity is lossed. 15

DATA bits that represent data. 18, 19

Data the bytes that contain valid information. 6

Data Block A complete set of bits. 7

Data Rate The speed in which data get transferred. 6

Data Stream A continous flow of bits without interupption. iv, 6, 9, 11, 15

Data Stream mode A Term from the AXI-Stream documentation indicating which mode is used. 6

Data-in Bits incoming from an external device representing information. 11

Data-out Bits outgoing to an external device representing information. 11

Data-words Words representing data. 17

De-Framing Burst A module of Interlaken that de-frames the bursts into data. v, 4, 6, 9, 10, 22, 23

De-Framing Meta The part of Interlaken that de-frames metaframes. 9

Decoder A module of Interlaken that decodes the data. 9

Descrambler The module of Interlaken that descrambles a frame. 9

DMA Direct Memory Access. 2

Dynamic Work Environment An environment that can easily be ported. 4

Encoder A module of Interlaken that encodes the data. 8

End of packet Ends a packet. 18

EOP End Of Packet. v, 11, 17–21

Error Status A flag that indicates if something went wrong. 6

ET The department of Electronic Technology. 2

ET-Department The Electrical Technologies department. 1

External Device A device that is not part of the main device. 8

External Devices Multiple devices that are not part of the main device. 6

FELIX Front End LInk eXchange. i, iii, v, 2, 4, 6, 13, 16, 24

FELIX card The FPGA board containing all the Interlaken Firmware. 4

FIFO A buffer that follows the First In First Our rule. 4, 6, 7, 9, 23

FIFO depth The bit storage size of a FIFO. 7, 22

Firmware VHDL descriptions used to instruct FPGA's. 4

For Loop A structure to repeat an action for n amount. 10

FPGA Field Programmable Gate Array. i, iii, 1, 4, 16, 22, 24

Frame A single burst of data. 6, 15, 18

Frames Multiple burst of data. 6, 7

Framing Burst A module of Interlaken that frames the data into bursts. v, 6, 8, 10, 11, 17–19, 21–25

Framing Burst Error Flags that the burst format isn't reached. 15

G-loopback Determines if Interlaken is used in looped to itself. 11

Gb/s Gigabit per second. i, iii, v, 4, 11, 13, 15, 16, 22, 24

GBT protocol A point to point protocol to stream data. i, iii

Handshake A series of predefined actions that ensure valid communication. 6

HealthInterface Signal A flag signaling the health of whole Interlaken. 10

HealthInterface Signal A flag signaling the health of whole Interlaken. 10

Higgs particle An elementary particle named after Peter Higgs. 2

HVA Hogeschool van Amsterdam, Amsterdam University of Applied Sciences. 1

IDLE a data-less word to keep a contious stream. v, 18–21

Idle-words Words to fill up the data-stream when there is no data. 17, 18

Interlaken Shorter naming for Interlaken Protocol. i, iii, 1, 2, 4–6, 8–17, 19, 21–25

Interlaken 0 The name of the Interlaken150G wrapper device. 14

Interlaken Firmware VHDL descriptions used to instruct FPGA's. 23

Interlaken IP-Core An IP-Core running Interlaken Firmware. 19

Interlaken Project FELIX version of the Interlaken Protocol. 1, 2

Interlaken Protocol A point to point protocol. i, iii, 2

Interlaken Receiver The Receiver used in Interlaken. 9

Interlaken Receiver MultiChannel An addition to the interlaken Receiver to allow Multichannel use. 1, 9

Interlaken RX The Receiver used in Interlaken. 10

Interlaken Transceiver The Transceiver use by Interlaken. 10

Interlaken Transmitter The Transmitter used in Interlaken. 8

Interlaken Transmitter MultiChannel An addition to the interlaken Transmitter to allow Multichannel use. 1, 8

Interlaken TX The Transmitter used in Interlaken. 10

Interlaken150G IP-Core The new Transceiver ip-core containing the Interlaken Protocol. 1, 14, 15, 22–24

Interlaken150G wrapper The extension of the Interlaken150g Ip-Core. 11, 14

IP-Core A complete version of firmware that is build in vivado. 13, 15, 19, 22–25

IP-core wizard A vivado tool to generate an IP-Core. 14

Lane Interaction Different Lanes know what to do without conlflict, by using communicative signals. iv, 17

Lane Logic All the logic used by a lane. 9

Lane synced A signal flaggin the lane is locked in sync. 15

Large Hadron Collider A large device in GenÃÍve to perform Experimental Physics. i, iii

Lbus A data bus. 14

Logic Vectors combined logic to create a multi bit vector. 7

m-axis The master signals of the AXI-Stream. 7, 9

m-axis-prog-empty A m-axis signal that states that the FIFO is empty. 7

m-axis-tready A m-axis signal that signal the master is ready. 7

Master The master refers to the devices that sends requests. 6, 7, 11

Meta-framing The part of Interlaken that frames metaframes. 8, 25

Meta-Framing Error The meta frame format isn't correct. 15

Meta-words Words created by the meta-framer. 25

Multi-lane The use of more than one channel. 11

Multi-lane version of Interlaken A version of Interlaken that utilizes more than one lane for the data stream. i

Multichannel More than one channel. 9, 11, 17, 19

Multiplexer A device that can be used to select a multiple of output combinations. 7

N-channel n amount of channels. 17

N-lanes n amount of lanes. 19

NIKHEF Dutch National Institute for Subatomic Physics. i, iii, 1, 2

Packages Multiple sets starting with SOP followed by DATA and ending with an EOP. 10, 11

Packet One set starting with SOP followed by DATA and ending with an EOP. 17–19

Packet-length The maximum size of a packet. 16

PCI-express Peripheral Component Interconnect Express. 2

Point to Point Protocol Data protocol that transfer data directly between two devices. 1, 2, 13

Position Bytes These are an optional function of the AXI-Stream. 6

Questa A short reference to Questa sim. 25

Questa-sim A program which is used to run VHDL testbenches. v, 19, 25

Ready Referring to if a device is in a state to accept a certain action. 6, 7, 11

Receiver Receiver. 17, 19

Receiver Side The Receiver side. 8, 9

Receiver-MultiChannel An addition to the interlaken Receiver to allow Multichannel use. 17

Record Types A collection of logic and logic vectors in one bundle. 7

Reset An Asynchronous signal used to set everything back to their initial state. 10

RX Receiver. 1, 9, 10, 13, 25

RX-lbus Receiver lbus. 14

RX-side The Receiver side. 8, 11, 15, 23–25

s-axis The slave signals of the AXI-Stream. 7, 9

Scrambler Scrambles data, part of the interlaken transmitter. 2, 8

Sigasi A VHDL editor. 25

Single lane protocol A protocol utilizing only one channel. 17

Single lane version of Interlaken A version of Interlaken that utilizes one lane for the data stream. i

Single Packet One set starting with SOP followed by DATA and ending with an EOP. 16

Slave The slave refers to the devices that answers a request. 6, 7

SOP Start Of Packet. v, 11, 17–21

Stat-rx-aligned Signals if the rx-side of the transceiver is synced. 10

State-Machines A process containing states run bij a state-register and decoder. 4

Synced state The state in which a lane is synchronized and achieved a lock. 16

Tcl A script extension. 4

Tdata Signal containing data information. 22

TDEST An optional signal of the AXI-Stream. 6

Tlast A AXIS signal flagging an EOP. 10, 11, 18–22

Transceiver The Transceiver has a RX and TX side and allows connections to external devices. i, iii, v, 5, 11, 13–15, 22–25

Transceiver-10g-64b67b Name of the transceiver. 13

Transceivers LBUS A bus containing the output and input of the transceiver lanes. 13

Transmitter The Transmitter collects incoming data. 8, 17–19

Transmitter-MultiChannel An addition to the interlaken Transmitter to allow Multichannel use. 8, 10, 17–19, 21, 22

Tready Signal signalling if action is valid. 22

TSTRB An optional signal of the AXI-Stream. 6

Tuser AXI-Stream signal containing error information. 6

Tuser Signal containing error information. 22

Tvalid Signal signalling if request is valid. 22

TX Transmitter. iv, 1, 8, 13, 24, 25

TX-lbus Transceiver lbus. 14

TX-side The Tranmitter side. 8, 9, 11, 15, 17, 21, 24, 25

Valid Flags if action or bit is accepted. 10

VCU128-ES1 Name of an FPGA. 13

VHDL Very high speed integrated circuit Hardware Description Language. 1, 4, 14, 25

Virtex Ultrascale + Name of a development board. 13

Vivado A FPGA IDE. 14, 25

VU37P The name a FPGA development board. 4

wrapper An additional layer added layer. 11, 14, 22, 24

Xilinx An FPGA producing company. 13, 15, 23

Xilinx core An IP-Core written by Xilinx. 11

Xilinx IP-Core Wizard The IP-Core wizard of Xilinx. 22

XPM-FIFO-AXIS The name of the Xilinx generated FIFO. 7