Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

High Speed IO using Xilinx Aurora

Examensarbete utfört i Elektroteknik
vid Tekniska högskolan vid Linköpings universitet
av

Jeremia Nyman

LiTH-ISY-EX--13/4727--SE

Linköping 2013

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

High Speed IO using Xilinx Aurora

Examensarbete utfört i Elektroteknik
vid Tekniska högskolan vid Linköpings universitet
av

Jeremia Nyman

LiTH-ISY-EX--13/4727--SE

Handledare: **Andreas Ehliar**
ISY, Linköpings universitet
Magnus Johansson
SAAB Dynamics
Roger Johansson
SAAB Dynamics

Examinator: **Olle Seger**
ISY, Linköpings universitet

Linköping, 7 december 2013

Avdelning, Institution
Division, Department

Elektroteknik
Department of Electrical Engineering
SE-581 83 Linköping

Datum
Date

2013-12-07

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

ISBN
__

ISRN
LiTH-ISY-EX--13/4727--SE

Serietitel och serienummer
Title of series, numbering

ISSN
__

URL för elektronisk version
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-102422

Titel
Title

High Speed IO using Xilinx Aurora
High Speed IO using Xilinx Aurora

Författare
Author

Jeremia Nyman

Sammanfattning
Abstract

A VHDL evaluation platform and interface to the Xilinx Aurora 8b/10b IP has been designed, tested and evaluated. The evaluation platform takes an arbitrary amount of data sources and sends the data over 1,2,4 or 8 multi gigabit serial lanes, using the Aurora 8b/10b protocol. A lightweight communications protocol for point-to-point data transfer, error detection and recovery is used to maintain a reliable and efficient transmission scheme. Priority between sources sharing the serial link is also a part of the platform.

The Aurora 8b/10b IP is a lightweight protocol and transceiver interface for Xilinx FPGAs, based on the 8b/10b line encoding protocol.

In addition, a demonstration PCB has been developed to introduce the Kintex-7 FPGA to future products at SAAB Dynamics.

Nyckelord
Keywords

FPGA, Xilinx Aurora, HSIO, High Speed, Serial communication

Abstract

A VHDL evaluation platform and interface to the Xilinx Aurora 8b/10b IP has been designed, tested and evaluated. The evaluation platform takes an arbitrary amount of data sources and sends the data over 1,2,4 or 8 multi gigabit serial lanes, using the Aurora 8b/10b protocol. A lightweight communications protocol for point-to-point data transfer, error detection and recovery is used to maintain a reliable and efficient transmission scheme. Priority between sources sharing the serial link is also a part of the platform.

The Aurora 8b/10b IP is a lightweight protocol and transceiver interface for Xilinx FPGAs, based on the 8b/10b line encoding protocol.

In addition, a demonstration PCB has been developed to introduce the Kintex-7 FPGA to future products at SAAB Dynamics.

Acknowledgments

I would like to thank my supervisors at SAAB Dynamics, **Magnus Johansson** and **Roger Johansson** for helping me during the thesis work.

I would also like to thank **Andreas Ehliar**, my supervisor at Linköping University.

Big thanks to **Erik Karlsson**, **Martin Nielsen-Lönn** ,**Christian Svensson**, **Robert Norlander** and **Gaspar Kolumban** for all the help during my time at the university.

Last but not least, thanks to **Olle Seger** as my examiner at Linköping University and **Kent Stein** for the great opportunity to do my thesis at SAAB Dynamics.

Linköping, October 2013
Jeremia Nyman

Contents

Notation

ABBREVIATIONS

Abbreviation	Meaning
CPLL	Channel PLL
CRC	Cyclic Redundancy Check
DSP	Digital signal processing
FIFO	First In, First Out
BER	Bit error rate/ratio
EOF	End of frame
FMC	FPGA Mezzanine Card
FPGA	Field programmable gate array
GPIO	General purpose input/output
IP	Intellectual property
ISI	Inter-symbol interference
LED	Light emitting diode
LFSR	Linear Feedback Shift Register
MGT	Multi Gigabit Transceiver
PCB	Printed circuit board
PER	Packet error rate/ratio
PISO	Parallel input serial output
PLL	Phase locked loop
PRBS	Pseudo random binary sequence
QPLL	Quad PLL
SATA	Serial advance technology attachment
SIPO	Serial input parallel output
SMA	SubMiniature version A
SOF	Start of frame
UART	Universal asynchronous receiver/transmitter
USB	Universal serial bus
VHDL	VHSIC Hardware Description Language

1 Introduction

The demand of higher data rate in all kinds of applications are steadily growing. Applications such as high-resolution TV screens, high-speed camera sensors etc. needs to transmit and receive data at very high speeds. Take a 4K monitor for instance; running at a 3840×2160 resolution, 30-bit color depth per pixel and a 60 Hz update frequency. This monitor consumes almost 15 gigabit of uncompressed data every second, see equation 1.1.

$$3840 \times 2160 \times 30 \times 60 \approx 14.93 \times 10^9 [bps] \tag{1.1}$$

1.1 Background

At SAAB Dynamics, different kinds of sensors are used. These sensors in the final construction is not always placed physically close to the units that do computations on this data. A typical scenario could be sensors placed at the front of an aircraft producing data, whereas the processing units consuming this data are placed closer to the cockpit. Depending on the distances involved, PCB cost, signal integrity issues, data rate, number of data sources etcetera, a serial communication scheme might prove to be a good choice when transmitting data between the devices.

A plethora of serial communication protocols exists today; USB, SATA, Ethernet, Fibre channel, Infiniband, Serial Rapid IO to name a few. All with their pros and cons, as well as their intended use. Although they are proven to work and offer quite a lot of functionality, sometimes these protocols might not be the most optimal solution.

In a point-to-point communication scenario, where data from device A is to be transferred to device B without any switching fabric in between, many of the protocols stated

is unnecessarily complex and the inferred overhead from protocol specification reduce the throughput. The USB and SATA protocols is intended for point to point use, but the implementation of these protocols in an FPGA might prove complex and/or expensive to purchase, and might need extra hardware outside the chip.

Xilinx Inc. offers a free and open high-speed protocol for their FPGAs called Aurora. It is intended for point-to-point communication between FPGAs with speeds up to and above 10 Gbps.

1.2 Purpose and goal

The purpose of this thesis is to investigate if the Aurora IP is a viable choice for SAAB Dynamics to use in their designs where high-speed interconnects are needed. In addition, a demonstration PCB is to be designed to introduce the Xilinx 7-series FPGAs to SAAB Dynamics. The PCB will also be used to evaluate the choice of PCB material when designing with high bit rates.

This thesis will focus on testing Aurora in a somewhat realistic manner. How is it used? Is it easy to use? Is it viable to use for FPGA-FPGA communication? Is it scalable? How flexible is it? How high bit rates can the FR-4 PCB technology cope with?

1.3 Prerequisites

It is assumed that the reader is comfortable with digital electronics and synchronous digital systems. It is also assumed that the reader has some experiences working with FPGAs and RTL code.

2
Problem

This chapter will introduce the problem to be investigated in this thesis. Constraints in the system will also be covered.

2.1 Description

A number (N) of sources (src_N), which could be any data generating devices such as a camera or sensor, generates arbitrarily amounts of data and are connected to an FPGA. The sources are connected using the interface listed in table 2.1. The data needs to be sent, depending on source priority, to another FPGA using the Xilinx Aurora IP, see fig 2.1.

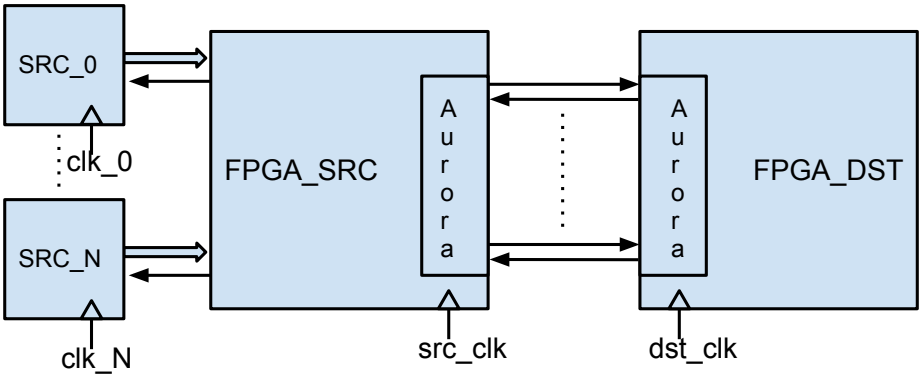

Figure 2.1: *Problem set up*

Name	Direction	Description
DATA(31..0)	out	32-bit data port.
DV	out	Asserted when DATA output is valid.
SOF	out	Start of frame. Asserted when first data of frame is on the DATA bus.
FULL/HALT	in	Asserted when the input buffer is full.
EOF	out	End of frame. Asserted when last data of frame is on the DATA bus.
SRC_CLK	out	Source clock.

Table 2.1: *Source interface*

2.1.1 Requirements and constraints

The requirements and constraints of the problem is presented below

1. The number of sources is arbitrarily large.
2. All sources are considered to have an identical interface to their surrounding, see table 2.1 and runs in their own clock domain.
3. The data generated from the sources are packaged in a frame. The word size of the source frame is 32 bits and can be arbitrarily long, but always a multiple of 32 bits. There are no partial words.
4. The sources are independent and can request to send data at any time.
5. The data generated from each src_N must be sent to a destination (dst_N) on another FPGA.
6. The serial links has to be interfaced using the Xilinx Aurora 8b10b IP.
7. The data should be sent over $N_{lanes} = 1, 2, 4, 8$ parallel serial channels (lanes).
8. The solution must allow sources with higher priority to have precedence over sources with lower priority.
9. The solution must be fair to sources with respect to their priority. Sources with the same priority cannot be allowed to starve their consumer.
10. The communication between FPGAs is point-to-point, master-slave communication. There is no need to route traffic from the master to different slave FPGAs.
11. The VHDL implementation needs to be generic with respect to the number of sources N and the number of lanes N_{lanes}.
12. The physical channel used to transmit data is considered unreliable. Bit errors can occur.

2.1.2 Considerations

Since the sources and the design are asynchronous, the input from the sources needs to be synchronized to the designs clock domain.

Since the sources can request to transmit data at any time, especially at the same time, a device is needed to select one of the sources that are ready to transmit, depending on their priority.

Since the channel is considered unreliable, error detection and error recovery must be a part of the design and protocol.

Since different sources needs to send to different destinations on the destination FPGA, some kind of addressing protocol is needed.

Since the communication is done point-to-point, the addressing part of the protocol can be significantly simplified.

3

High-speed IO

This chapter introduces high-speed serial communication, the Kintex-7 FPGA from Xilinx and some techniques used to increase performance on non-ideal transmission channels.

3.1 HSIO Background

When deciding upon a communication interface between two or more electronic devices, there are two choices, serial or parallel. Simplified, this is what it breaks down to.

When using a parallel interface, all data bits in the current word is transmitted simultaneously and each bit is sent over a separate wire, see fig 3.1. A word is a collection of bits, not strictly defined. It can have any length, although it is common to be a multiple of eight bits. In addition to the data bits, other signals are needed to indicate that the data on the bus is valid and/or if the current word is a data word or a control word, if this is not encoded in the information being sent.

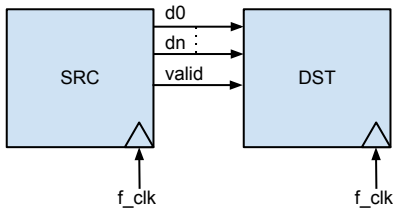

Figure 3.1: *Typical parallel communication interface.*

When using a serial interface, the bits of the current word is sent over one wire, one bit at a time, see fig 3.2. Depending on protocol, other wires may or may not be needed. Since only one wire is used to transmit data, there has to be some kind of protocol encapsulated in the transmission in order to differentiate between data/control words and start/end of current data.

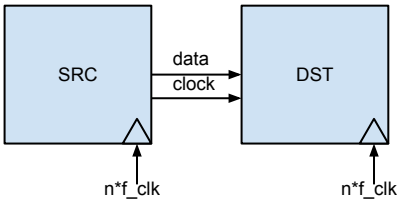

Figure 3.2: *Typical serial communication interface.*

3.1.1 Serial Vs. Parallel

A quick analysis using the information about the two approaches results in the following: If the word length is eight for example, a parallel interface needs at least eight times the amount of wires connected between the devices, but can transmit its data running at an eighth of the clock frequency needed by the serial interface. Or from the other way around: A serial interface needs an eighth of the amount of wires compared to a parallel interface, but needs to run at eight times the clock frequency to send data at the same rate. Also in the serial case, assuming a simple protocol with one start bit and one stop bit, this has to send ten bits in order to send eight "real" data bits. This means a 20% overhead and the serial interface needs to run at another 25% higher clock frequency to run at the same data rate as a parallel interface.

So why chose a serial interface over a parallel? For a given clock frequency it is, at least theoretically, possible to send ten times the amount of data when using a parallel interface.

Increasing the word size creates problem when drawing traces for the wires at the PCB. Too many wires make a larger and/or a more expensive PCB. At higher speeds the wires need to be matched in length to reduce differences in delay between the wires. All the bits need to arrive at the same time. In order to reduce crosstalk between signals, the signal traces needs to be placed at some distance from each other. The longer the distance, the lower the impact of crosstalk. This is usually done by encapsulating each signal wire between two ground wires. For eight signal wires, an additional nine wires are needed to shield them properly. A differential pair to reduce the impact of noise, doubling the amount of wires, is also needed at higher frequencies.

Since the amount of I/O pins on a given device is limited, the amount of I/O pins needed for an application might not be available, or too expensive. An eight-bit bus using differential signaling needs at least 16 wires, compared to two using a corresponding serial interface. To send eight bits using differential pairs and shielding, this would require $8 + 8 + 9 = 25$ traces on the PCB and $8 + 8 = 16$ I/O pins on the device.

Although fewer wires and lower cost in terms of PCB area, noise radiation and interconnects, a serial interface faces other problems. It still has to run at a higher clock frequency to reach the same data rate. This creates problems with for example Inter-symbol interference and impedance matching of wires and connectors . A more expensive PCB material might also be needed to reduce impedance mismatch. All this given that the technology used for source and destination is even capable of running at those speeds. In addition, it also induces protocol overhead and protocol complexity. [Athavale and Christensen, 2005]

3.1.2 Self synchronous systems and clock recovery

In order to mitigate problems when sending data and clock separately, it is common to include the clock in the data stream. At the receiver end, the clock is then extracted from the bit stream and used to synchronize the incoming data with the receiver. This is called a *self-synchronous system*. In contrast to the *source-synchronous system* when the clock is sent separately. The consequence of sending data and clock using the same wire is that the data has to be coded in such a way that enough transitions occur to extract the clock from the incoming stream of bits. A common way of coding the data is to use 8b/10b encoding. [Athavale and Christensen, 2005]

3.2 Xilinx 7-series FPGA

3.2.1 Multi Gigabit Transceivers

A multi gigabit transceiver, or MGT for short, is the heart of a high-speed serial I/O interface. Simplified, its mission is to take a word in parallel on each clock cycle at some frequency A. Then serialize the word at frequency $B = length(word) \times A$ and transmit this serial stream over a channel. On the receiver end, the serial stream is de-serialized at frequency B. It is then presented to the receiving application with a word in parallel, at frequency A, see fig 3.3. The circuitry that does the serializing and de-serializing is commonly called a SerDes. The SerDes part of the MGT in figure 3.5 is denoted as PISO (parallel input serial output) and SIPO (serial input, parallel output) for the serializer de-serializer part respectively. The 7-series FPGA from Xilinx contains transceivers

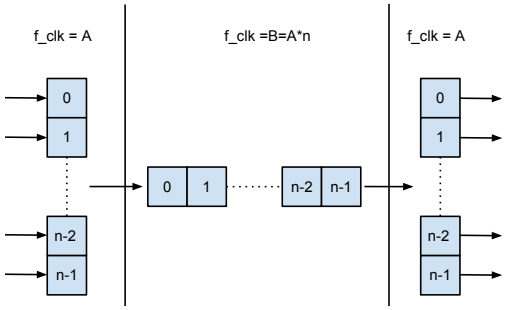

Figure 3.3: *Serializer/Deserializer, "SerDes", principle.*

which can cope with speeds up to 28 Gbps in the extreme case, using a GTZ transceiver. Although a more "modest" transceiver family, GTX, will be used in this thesis, with a capability of up to 12.5 Gbps. For the sake of completeness, the 7-series could also be equipped with a GTH or a GTP transceiver, with 13.1 and 6.6 Gbps respectively.

A transceiver in the 7-series is located in a so-called GTX Quad. A quad is a collection of four GTX transceivers placed near each other on the silica, sharing resources, see fig 3.4. The number of quads on a 7-series FPGA varies from model to model. Each physical transceiver is referenced in the data sheets using an XNYM coordinate system, where N and M are integers.

Each quad contains four so called CPLL and one QPLL. These are Xilinx names for the PLLs that synthesize the reference clocks for the transceivers. Each CPLL can synthesize different clock frequencies and this allows the four transceivers to run at different speeds independent of each other. The CPLL can operate at frequencies between 1.6 and 3.3 GHz, whereas the QPLL can operate between 5.93 and 12.5 GHz. This means that if the serial link needs to run at higher frequencies (6.6 Gbps and above), the QPLL is needed to source the transceiver. This in turn means that all transceivers in the same quad that needs to run at a 6.6Gbps+ rate has to run at the same line rate.

Each quad has two reference clock inputs. There is also a possibility to source quads directly above or below from these reference clocks [Xilinx Inc, 2013b].

The transceivers are a complex piece of circuitry, see fig 3.5 and although the details are omitted in this thesis (the data sheet is around 500 pages), some parts are worth noting.

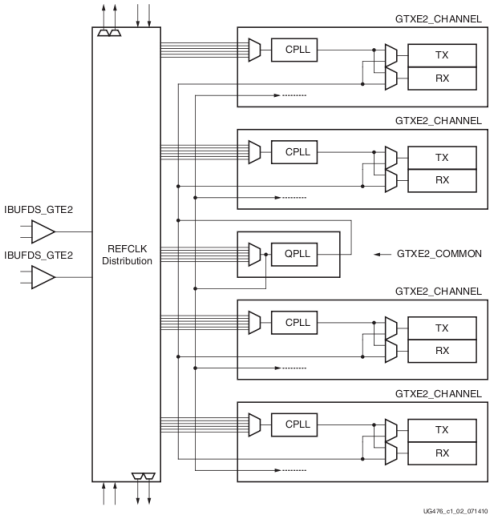

Figure 3.4: *Xilinx 7-series GTX transceiver quad. [Xilinx Inc, 2013b]*

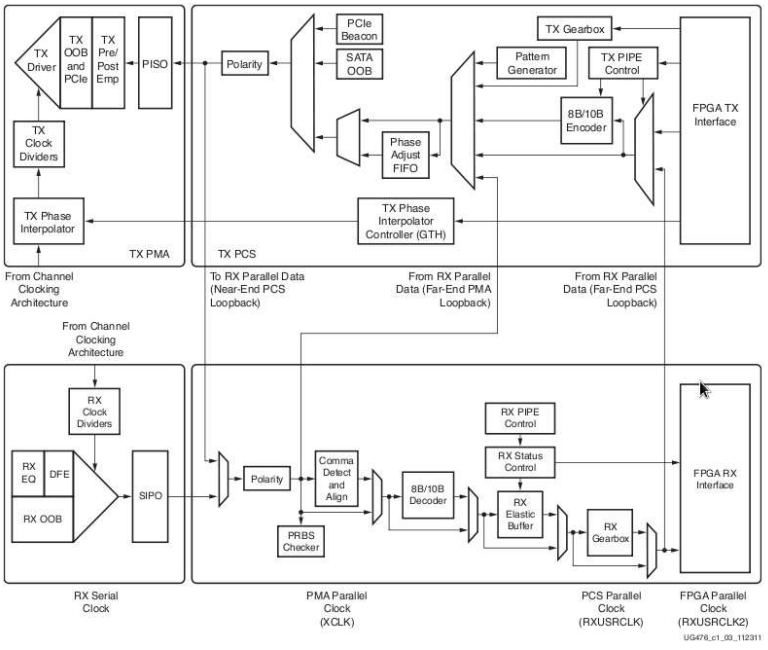

Figure 3.5: *Xilinx 7-series GTX transceiver. [Xilinx Inc, 2013b]*

Pre and post-emphasis

In the transmitter part of the transceiver, a circuitry performs Pre and post-emphasis. This is done in order to reduce the effects of ISI, short for inter-symbol interference. ISI occurs when long series of the same bit value (’0’ or ’1’) are transmitted over the channel, followed by a bit of the opposite value. The effect of this is that parasitic capacitance on the transmission line has a long time to charge or discharge, up to a level where it may not have time to charge or discharge during the short opposite value, see fig 3.6. The pre and post-emphasis circuitry reduces the effects of ISI by over or under-driving zero->one and one->zero transitions, see fig 3.7. [Athavale and Christensen, 2005]

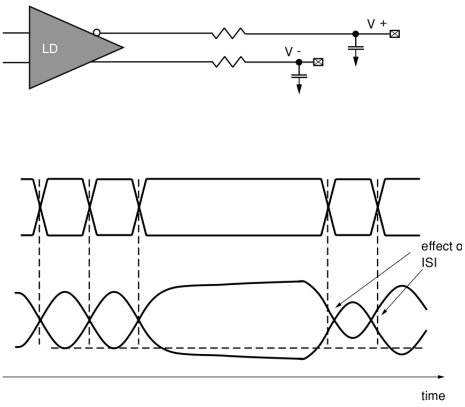

Figure 3.6: *Inter-Symbol interference [Athavale and Christensen, 2005]*

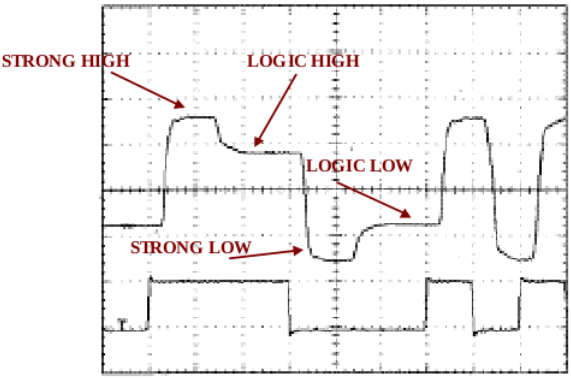

Figure 3.7: *Pre and post-emphasis principle [Athavale and Christensen, 2005]*

RX equalization

As a lossy cable length gets longer, the frequency response of the cable tends to attenuate higher frequencies, acting as a low pass filter. The low pass cut off frequency of the cable response might not impact at lower bit rates, but as the bit rate increases, the attenuation of the cable can have a large impact at the receiver end. To compensate for this, the receiver is fitted with a digital equalization filter. The goal of the equalizer is to boost high frequency components, trying to flatten the low pass tendency of the cable, moving the cut off frequency closer to the frequency of the bit stream, see fig 3.8. [Maxim Integrated, 2011]

3.3 8b/10b line encoding

8b/10b encoding is a common way of coding data in such a way that enough transitions for the clock recovery circuitry at the receiver to operate. The coding also ensures DC-balance on the wires, ensuring good electrical properties. The 8b/10b encoding maps each combination of eight bits into a ten-bit symbol that when sent has the property that the amount of ones and zeros in a row is limited, and that the ratio of one and zeros on average over 20 bits is 50%.

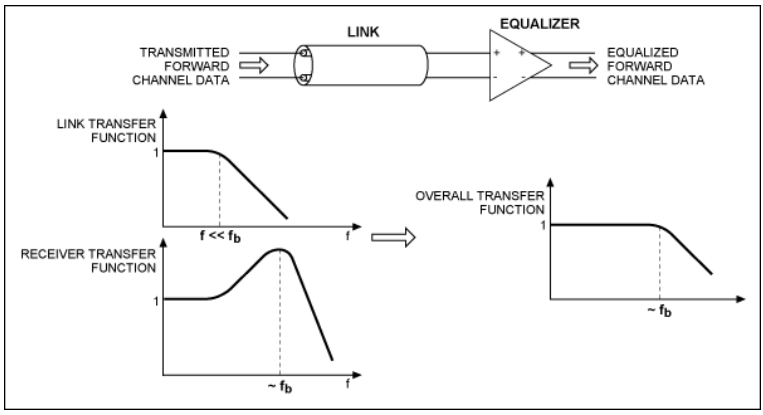

Figure 3.8: *Receiver equalization principle [Maxim Integrated, 2011]*

This is achieved by giving the transmitter for each byte, two ten-bit symbols to choose from. One that has a surplus of ones and another that has a surplus of zeros or, has the same amount of one and zeros. These different symbols are often denoted with a + or a - sign. The transmitter monitors the previous symbol sent and for the next transmission, chooses a symbol that brings the ratio to 50% [Franaszek and Widmer, 1983]. Some examples of 8b/10b symbols are given in table 3.1.

name	8bits	-symbol	+symbol
D10.7	11101010	0101011110	0101010001
D31.7	11111111	1010110001	0101001110
D4.5	10100100	1101011010	0010101010
D0.0	00000000	1001110100	0110001011
D23.0	00010111	1110100100	0001011011

Table 3.1: *Example 8b/10b symbols. [Athavale and Christensen, 2005]*

In addition to the byte mapping, the new ten-bit code also has room for special control characters. These can be used to align data in frames, to send idle characters keeping the channel open and do clock correction to keep small differences in reference clocks between transmitter and receiver in sync [Athavale and Christensen, 2005]. Since 8b/10b encoding is cheap to implement in hardware, an 8b/10b encoder and decoder is commonly found as a feature inside transceivers, see fig 3.5

3.4 LogiCORE IP Aurora 8B/10B v8.3

The Aurora 8B/10B IP is an IP from Xilinx that acts as an interface to the MGT s on Xilinx FPGAs, as well as implementing the Aurora 8b/10b protocol. Aurora adds a layer of abstraction above the MGT s, giving the user a way of sending data using the transceivers, without worrying about all the transceiver configurations.

3.4.1 Aurora 8b/10b protocol

The Aurora 8b/10b protocol defines electrical specifications and timings, but also how to initialize and maintain a channel, how to bond several lanes into one channel etc. It also specifies how combinations of 8b/10b symbols make up start and end of Aurora channel data frames. An example channel data frame can be seen in fig 3.9 where the \SCP\ and \ECP\ are certain combinations of 8b/10b symbols. For full information about the Aurora 8b/10b protocol, please refer to [Xilinx Inc, 2010].

Figure 3.9: *Typical Aurora channel data frame [Xilinx Inc, 2012]*

3.4.2 IP Core generation

In order to generate an Aurora 8b/10b IP core a program called Xilinx CORE generator system, Coregen, is used. Coregen is shipped with a number of different IP cores that is generated on the fly for the platform chosen. The IP cores can be anything from fast adders and FFTs to MGT protocols and on chip logical analyzers. A typical scenario for generating an Aurora core is shown in fig 3.10 and fig 3.11 and the parameters are explained in the sections below.

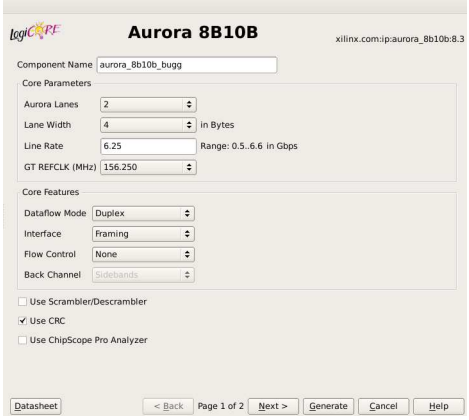

Figure 3.10: *Aurora core parameters, page 1*

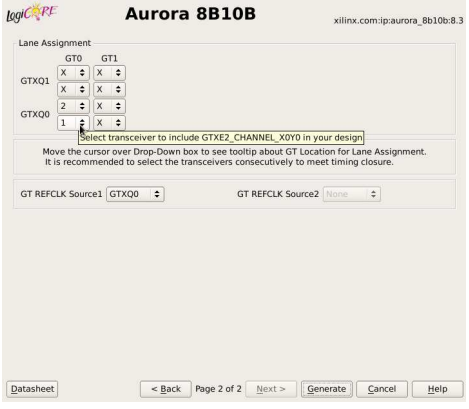

Figure 3.11: *Aurora core parameters, page 2*

Aurora Lanes

This parameter sets the amount of Aurora lanes in the generated Aurora core. One Aurora lane is connected to one transceiver. The amount of lanes supported depends on the currently used FPGA chosen when starting a Coregen project. In this case, eight is the maximum since this is the amount of transceivers actually on the chip. Multiply this value by the lane rate and get the total bit rate of the core. For instance, a 4 lane Aurora core at 3.125 Gbps gives a $4 \times 3.125 = 12.5$ Gbps system, using four transceivers and eight differential pairs.

Lane Width

This parameter, together with the amount of lanes, sets the width of the data interface to the core. Using one Aurora lane and two bytes would create a $1 \times 2 \times 8 = 16$ bit wide interface but choosing four lanes and four bytes would create a $4 \times 4 \times 8 = 128$ bit wide interface. A general formula for the bit width of the interface is given in equation 3.1

$$bitWidth = N_{lanes} \times N_{laneWidth} \times 8 \tag{3.1}$$

Each lane is mapped to one transceiver on the FPGA.

Line Rate

Using this parameter the user sets the desired line rate of each Aurora lane in Gbps. This is limited by what circuit that was selected, but also by which encoding that is used. Using the Aurora 8b/10b protocol, this is limited to 6.6 Gbps.

This parameter, together with lane width sets the minimum clock frequency that the user design has to run at to reach maximum efficiency. Setting the line rate to 6.25 Gbps and a lane width of 4 bytes means that the user design needs to present the interface with 32 bits of data at a frequency of $6250000000/40 = 156250000 = 156.25$MHz. 40 bits in the denominator instead of 32 due to the 8b/10b coding that needs to send two extra bits for each byte.

GT REFCLK

In this field, the user has to set at what frequency the reference oscillator runs at. Only certain integer fractions of the line rate are supported, due to the internal clocking circuitry.

Dataflow Mode

Possible choices here are either simplex or duplex. Simplex comes in two flavors, RX-only simplex or TX-only simplex, where the device can only receive or transmit data. Choosing any of the simplex modes opens up the Back Channel choices. The duplex mode gives access to both the TX and RX interfaces of the transceivers.

Interface

Here the user can choose from either a framing or a streaming interface to the core. A framing interface means that data is sent using frames, using start and end of frame indicators. Using a streaming interface, the user application sends data on a stream instead, leaving any start/end of frame indicators to some overlaying protocol. Using a streaming interface grays out the “use CRC” option below.

Flow Control

The flow control parameters opens up possibilities for the receiver application to control the rate of incoming data, and or send high priority messages that pauses any current transmissions. The flow control comes in four flavors but the full details are omitted. The reader is urged to read the Xilinx Aurora 8b/10b user guide for more information. [Xilinx Inc, 2012]

Back Channel

The back channel field is enabled when one of the simplex modes are chosen. This allows the user to choose from two different ways of initialize the channel and report errors.

Scrambler/Descrambler

Enabling this option introduces a scrambler on the transmitter side, and a corresponding de-scrambler on the receiver side. This is done in order to make data seem more random, breaking repetitive patterns and gaining some desirable electrical characteristics.

CRC

This option includes an error checking CRC interface to the core. This option only enables detection of errors. The core does not react to errors more than presenting to the user that it has happened. CRC checking is only valid when the framing interface is used.

Chipscope Pro Analyzer

This option inserts a Chipscope core into the design. This allows the designer to probe internal signals when the design is running on the FPGA.

Lane Assignment

In this tab of the core generator, the user has to assign each lane to an available transceiver. In this case, the FPGA has two quads and eight transceivers. Hovering the mouse over the choices give the user feedback on which coordinates the current transceiver has. In this case the two lanes, one and two, were placed at transceiver X0Y0 and X0Y3 respectively.

GT REFCLK Source

Here the user selects which reference clock to use for the core. Since the quads can be sourced from the incoming reference clock, or from the quads directly above or below there are multiple choices here. If the lanes are separated by at least one quad, a second reference clock is needed. In this case, there are only two quads so a separation of at least one quad is not possible, thus graying out the source2 field.

3.4.3 Aurora 8b/10b IP Interface

The Aurora interface will differ depending on choices made in the Coregen wizard so only a subset of the interface will be presented. In table 3.3 at page 18 the data interface for an Aurora core generated with the parameters used in table 3.2 is described.

Parameter	value
Aurora Lanes	2
Lane Width	4
Line Rate	6.25
GT REFCLK	125
Dataflow Mode	Duplex
Interface	Framing
Flow Control	None
Back Channel	N/A
Use Scrambler/Descrambler	off
Use CRC	on
Use Chipscope Pro Analyzer	off

Table 3.2: *Parameters used for a 2 lane, 4 byte, 6.25Gbps Aurora core*

name	Direction	Description
S_AXI_TX_TDATA(0..63)	in	64 bit wide TX data port. Grows with Parameters Aurora lanes and lane width. Data on the TX_TDATA port is only passed when TX_TVALID is asserted.
S_AXI_TX_TKEEP(0..7)	in	Specifies the number of valid bytes in the last word of the transmission. Grows with parameters Aurora lanes and lane width. One hot encoded and only valid when TX_TLAST is asserted.
S_AXI_TX_TLAST	in	Assert when the last word of the frame is presented on the TX_TDATA field. Acts as end of frame.
S_AXI_TX_TVALID	in	Data valid signal. Assert this when valid data is one the TX_TDATA bus. Acts as start of frame.
S_AXI_TX_TREADY	out	Asserted from Aurora core when interface is ready to receive data.
M_AXI_RX_TDATA(0..63)	out	RX data out port. Only valid when RX_TVALID is asserted.
M_AXI_RX_TKEEP(0..7)	out	Specifies valid bytes in last word. Only valid when RX_TLAST is asserted.
M_AXI_RX_TLAST	out	Asserted when last word of current frame has been received - end of frame.
M_AXI_RX_TVALID	out	Asserted when data is valid on the RX_TDATA bus is valid. Also acts as start of frame.
LANE_UP(0..1)	out	Each bit asserted when the respective Aurora lane is initialized.
CHANNEL_UP	out	Asserted when both Aurora Lanes are initialized and the channel has been opened.
SOFT_ERR	out	Asserted when an error has been detected in the 8b/10b encoding, or the disparity (number of ones or zeroes in a row) rules are not fulfilled.
HARD_ERR	out	Asserted when too many SOFT_ERRs has occurred in a small amount of time. Also asserted when TX/RX buffers over or underflows. Issues a channel reset that brings down the Aurora channel and starts it up again.
FRAME_ERR	out	Asserted when errors to the Aurora channel frame have been detected.
CRC_VALID	out	Asserted for one clock cycle when a valid CRC has been calculated and checked by the RX interface.
CRC_PASS_FAIL_N	out	Asserted when the calculated CRC matches the received CRC.

Table 3.3: *One possible subset of the interface to Aurora core*

The user TX/RX data interface is a subset of the AXI4 interface, called AXI4-stream interface. See fig 3.12, 3.13 and 3.14 for an example of how to use the interface.

Writing data to the core follows this sequence: If TX_TREADY is asserted the user application starts a transmission by putting data on the TX_TDATA bus and asserting TX_TVALID. Keep TX_TVALID asserted until the last data word is on the bus, then assert the TX_TLAST signal and put the correct byte mask pattern on the TX_TKEEP bus. In fig 3.12, assuming a 4-byte word, if all 32 bits contain valid data TX_TKEEP is set to "1111" but if only the first two bytes contains valid data TX_TKEEP is set to "1100". TX_TVALID can be de-asserted by the user application at any time, pausing the transaction for as long as it is needed as seen in fig 3.13. If the TX_TREADY signal is de-asserted, no data will be sampled by the Aurora core even if TX_TVALID is asserted.

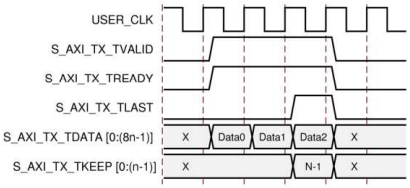

Figure 3.12: *Aurora TX interface write with partial word [Xilinx Inc, 2012]*

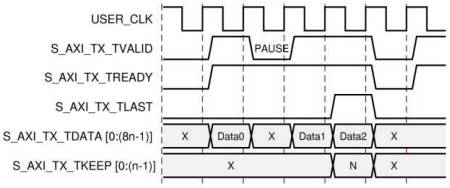

Figure 3.13: *Aurora TX interface write with pause [Xilinx Inc, 2012]*

When receiving data the same thing happens, but the interface is driven by the Aurora Core. RX_TVALID is asserted when the first word is on the data bus, and either held and or sometimes paused until RX_TLAST is asserted, completing one frame. Here it is up to the receiving application to handle the data and interpret the RX_TKEEP signal, see fig 3.14.

The CRC_VALID signal is asserted one clock cycle at the same time as RX_TLAST, and it is up to the user to sample the CRC_PASS_FAIL_N signal and take appropriate action.

The Aurora core will also source the user application with a reference clock in order to meet the timing requirements. The user application has to run at $\frac{LineRate}{8 \times LaneWidth + 2 \times laneWidth}$. For a 6.25 Gbps line rate and a lane width of four, the clock generated will run at $\frac{6.25 \times 10^9}{40} = 156.25$ MHz.

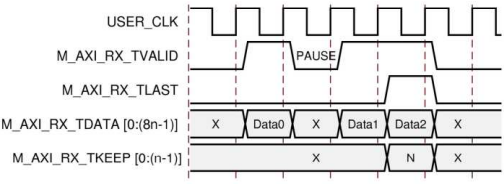

Figure 3.14: *Aurora RX interface read. [Xilinx Inc, 2012]*

4

Multi source multi channel Aurora Interface

In this chapter the architecture of a proposed multi source multi channel Aurora interface, MSMCAI for short, is presented. The architecture meets all the requirements stated in section 2.1.1 as well as some additional features.

4.1 Overview

The purpose of the architecture is to create a layer of abstraction above the Aurora interface that is transparent to the end user circuitry, where data generating devices can plug themselves in with little or no knowledge about the underlying protocol. The design idea could be compared with a software application opening a socket connection riding above TCP/IP and Ethernet.

In the following sections, necessary parts will be discussed and then the final architecture will be presented.

4.1.1 Source frame and Aurora Data frames

A source frame in the following sections is defined as a number of data words from the data source contained between a start of frame, SOF, and an end of frame, EOF. This frame can be arbitrarily long. An Aurora data frame follows the same idea, but due to implementation issues this has to be limited in length. The source frame has to be chopped up, transmitted and then re-assembled again at the receiver.

4.2 MSMCAI Protocol

In addition to raw data, some extra information needs to be sent to the receiver. A source with ID N needs to send to a receiver application with ID N so this address needs to be communicated. Information about SOF and EOF needs to be sent as well. Since the connection between source and destination is point to point, the protocol can be made simple. A 32-bit header containing an 8-bit destination address and two bits to indicate SOF/EOF starts each Aurora data frame. If the SOF field is '1', then the first word of the frame is the start of source frame. If the EOF bit is '1' then the last word in the frame is the last word of the source frame. When the receiver has received the current frame, the CRC port of the Aurora interface is checked for errors. If the frame is error free, an ACK is sent from the destination FPGA to the source FPGA, using the address received from the header. If an error has been found, a NACK is sent instead.

If a message was received at the receiver error free, there is still a possibility that the ACK is corrupted when received at the source FPGA. If no ACK/NACK is received or the message was corrupted, this will trigger a retransmission after a timeout. To make sure that the receiver does not receive the same data twice, the previous CRC is saved in the receiver and compared with the next. To make sure that identical data generates a different CRC each time, a revolving ACK-counter is also included in the header. If an ACK is received at the source, the ACK counter is incremented and used in the next header. If an ACK was sent but not received at the source, the counter will not increment, thus generating an identical CRC to the previous, causing the receiver to ignore this frame.

In order to reduce the header size, the source address is assumed the same as the destination address. This means that if the source is connected to address 0x0004 on the source FPGA then the destination application needs to be connected to address 0x0004 at the receiver FPGA. If this is the case, then there is no need to send extra information about the address of the source.

In order to reduce the possibility of addressing the wrong destination application, before checking the CRC, the source address is sent two times in the header and is compared in the receiver. See fig 4.1 for full header and ACK/NACK message bit numbering.

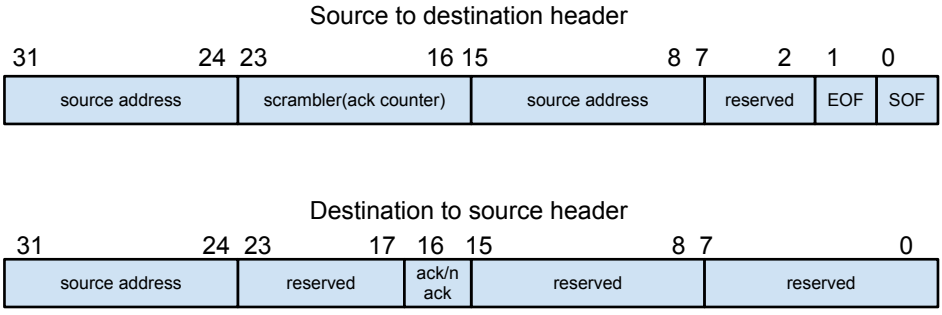

Figure 4.1: *MSMCAI header specification.*

When using multiple lanes bits 33 and upwards of the header are all zeros.

4.3 Synchronization

Since the sources are considered asynchronous to the system, running at a different clock frequency, all relevant signals from the source must be synchronized to the system clock running the design. This is done by inserting a FIFO between the outputs of the source and the inputs of the design. The FIFO is generated using Coregen and has separate read and write clock inputs. The write clock is connected to the source clock and the read clock is connected to the system clock. See fig 4.2. The size of the FIFO s is easily customized in the Coregen wizard. During this thesis, the size of the FIFO has been set to 1024 words.

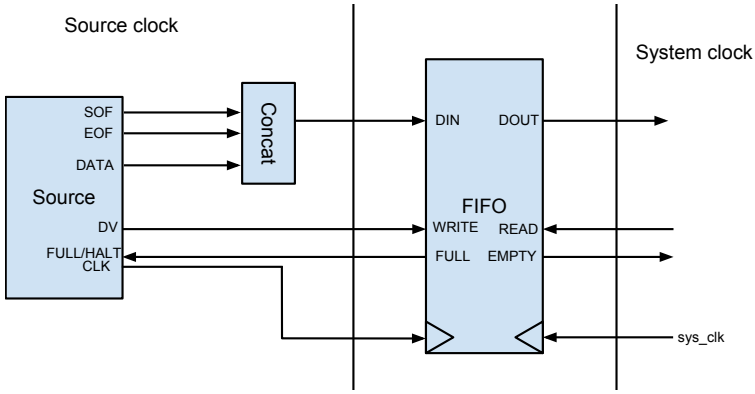

Figure 4.2: *Source-System synchronization using Asynchronous* FIFO *s*

4.4 Framer

Each source is connected to a framer and it is inside the framer that the AXI4-stream protocol to the Aurora core is implemented. The framer is granted the AXI4-streaming interface by an arbiter that is introduced in the next sections. The framer also chops up the source data frame into the smaller Aurora data frames.

Since each packet needs a header, it is desirable to keep the frame of data sent as large as possible, in order to reduce the percentage of inferred overhead from the header.

4.4.1 Frame sizing

Since the data frames from the source can be arbitrarily short or long a circuitry is needed to chop this frame up into smaller pieces. This is done for several reasons. One reason is to create a round robin behavior since the Aurora interface is a shared resource, letting other sources also use the bus not only after one source has sent its entire frame. Another reason is that if the source is running at a lower clock frequency than the system, there might not be a steady stream of data in to the Aurora interface. Leaving the interface waiting for data will lower the utilization ratio and it is better to buffer up a number of data words before requesting the bus to let other sources use the bus during this time. When the buffer is full, the source can request the Aurora interface and send a steady

stream of data without pauses until the buffer is empty.

Yet another reason is the cost of retransmissions if bit errors occur. Assume an unreliable channel where bit errors are known to occur and an error detection scheme where the only information available is that an error occurred in the frame or not. This needs to trigger a re transmission of the entire frame. Reducing the frame size does not only lower the probability of an error occurring in individual frames (assuming that error rate is independent of frame size), but also lower the amount of bits that needs to be resent if an error occurs [Qi et al., 2007].

In this design the frame size is either the depth of the transmit buffers (request the bus when the buffer is full) or the size of the last piece of the source frame (request the bus when end of frame has been detected). A FIFO with a depth of 66 words has been chosen during the implementation of the system. This is easily changed later by regenerating the FIFO primitive using Coregen.

4.4.2 Aurora interface and frame buffering

Although no partial 32-bit words are generated from the source, a multi-lane Aurora implementation needs careful consideration. In a two-lane case, the Aurora interface data bus is 64 bits wide and the source needs to be able to send only 32 of those 64 bits. In order to fill the EOF field of the header, the transmit buffer needs to either be filled, no EOF found, or filled until the EOF is found. The partial long word information could be placed in the header, but the TKEEP port of the Aurora interface wanted to be tested so this is utilized to indicate partial long words. A long word is defined when the number of Aurora lanes is above one. A four lane long word is constructed of four 32-bit words. A partial long word is a long word where not all 32 bit words are used.

The Aurora interface is $32 \times Nlanes$ bits wide, whereas the source is only 32 bits, so in order to fully utilize the wider Aurora data port the source data has to be parallelized. This is done by using $N = N_{Lanes}$ 36 bit wide FIFOs for each source and writing to each FIFO in a round robin manner. When sending data to the Aurora interface, all FIFOs are read in parallel.

No data is sent to the Aurora interface before the FIFO is either full or an EOF was found.

4.4.3 Retransmission

If the frame was corrupted on the way to the destination, the destination FPGA answers with a NACK after checking the CRC. If the NACK or ACK was not received, a time-out counter starts and when this reaches its threshold a re-transmission is issued. The retransmission is also started when a NACK is received.

This means that data at the transmitter cannot be discarded until an ACK has been received. This is solved by writing each word sent to the Aurora interface back at the top of the transmit buffer. In order to know when to stop reading from the transmit buffer when sending, one extra bit is written in addition to the data to indicate when to stop.

4.4.4 Double buffering

Since the buffer needs to be filled before transmitting, a double buffering scheme is introduced. This means that the second buffer can start to fill while the first one is transmitting data, reducing the delay between the data-generating source and start of transmission when long data bursts from the source are occurring [Bai and Liu, 2005].

4.4.5 Framer block diagram

In figure 4.3 the block diagram of the framer is shown. Data from the synchronization

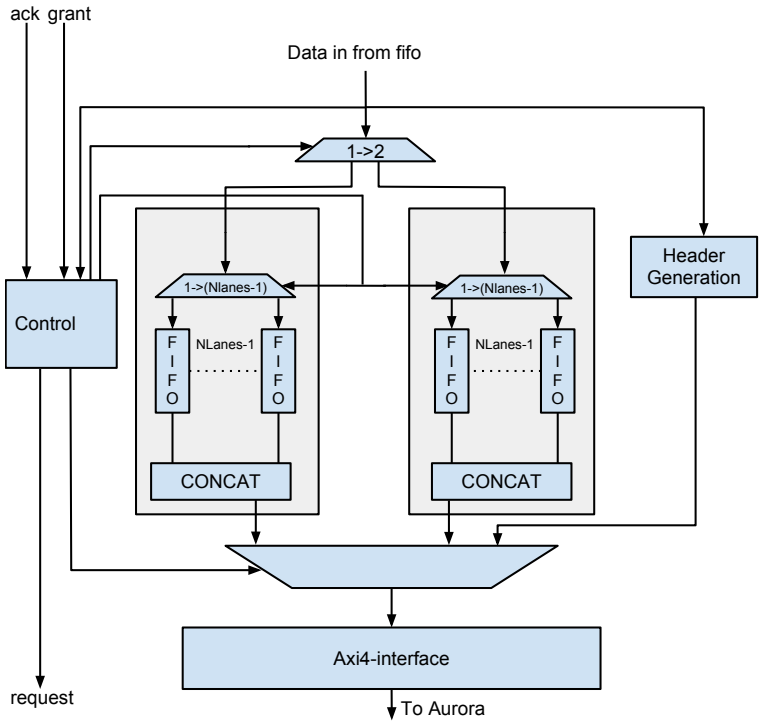

Figure 4.3: *Framer block diagram*

FIFO is written in one of the buffers inside of the framer. If the buffer gets full or an EOF is found, a request signal is sent to the system arbiter. When the grant comes, the framer is given access to the Aurora AXI4-streaming interface. The framer generates a header and starts to transmit data. When the contents of the current buffer has been sent, the framer waits for an ACK/NACK to be received, or in the case that the ACK/NACK was lost, waits for a timeout counter to reach zero. In case of an ACK/NACK reception, the framer either starts to transmit data from the second buffer if there is data there. If the next buffer is empty or not yet filled, the framer goes to a wait state. If a NACK was received, or the timeout counter reaches zero, the framer starts resending the contents in the first buffer. During the transmission and ACK/NACK wait, if one of the buffer is empty, the framer can still receive data from the source until both buffers are full.

4.5 Arbiter

In order to handle sources that want to access the Aurora interface at the same time, a structure for handling simultaneous requests is needed. Requests to use the bus can come at any time, and though it is unlikely that they arrive at the exact time, they might queue up when one framer is already granted the bus. Keywords in this section are *request* and *grant*. A framer issues a request when it wants to use the Aurora interface, and is granted the bus via a grant signal when the arbiter has selected from the set of requesting sources.

There exists *many* proposals for different arbiter architectures. One considered was the one used in the OpenCores Wishbone bus, see fig 4.4. It uses a simple state machine to rotate around all connected devices, checking if the current one is requesting the bus. If the current device does not need the bus, it changes state and checks the next device on the next clock cycle. [OpenCores, 2010] One pro of this idea is that it is simple to implement, but on the negative side, it has a high latency if the requesting source is "far" away from the current state. Worst case for an arbiter with this architecture with 32 sources connected is 31 clock cycles. In its original architecture, it also lacks priority between devices.

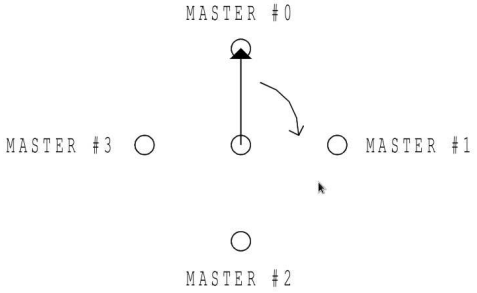

Figure 4.4: *Arbiter used in OpenCores Wishbone bus*

This being a high-speed system, a single clock cycle arbiter is needed. As previously stated there are many of arbiters out there designed to grant the connected devices in a single clock cycle. Two designs considered involving a binary search algorithm can be found in [Zheng and Yang, 2007] and [Zheng et al., 2002]. Although capable and allegedly simple, they probably need some implementation effort and there is still no priority involved.

An interesting approach requiring almost no implementation effort is proposed by [Krill, 2009]. Basically, four lines of VHDL code and the result is a fully functional round robin arbiter. A VHDL implementation based on [Krill, 2009] is shown in listing 4.1. Note that some of the VHDL syntax has been removed from the example to save space.

```
1  request_vector  : in  std_logic_vector(size-1 downto 0);
2  output_vector   : out std_logic_vector(size-1 downto 0));
3
4  rr_arb: process (sys_clk)
5  begin  -- process fsm
6    if(rising_edge(sys_clk))
7     if (rst_n = '0') then
8         gntm <= (others => '0');
9     elsif(enable = '1') then
10        if (bitor = '1') then
11         gntm <= gnts;
12        else
13         gntm <= gnt;
14        end if;
15     end if;
16   end if;
17 end process;
18
19   gnt <= request_vector and ((not request_vector) + 1) ;
20   reqs <= request_vector and (not ((gntm- 1) or gntm));
21   gnts <= reqs and ((not reqs)  +1);
22   bitor <= '0' when unsigned(reqs) = 0 else '1';
23
24   output_vector <= gnts when bitor = '1' else gnt;
```

Listing 4.1: *Round robin arbiter VHDL example*

At line 19 in listing 4.1, the request vector containing request signals from all devices will be and:ed with its twos complement negation. The resulting vector gnt will then contain the first non-zero element in request_vector from the right. See example in listing 4.2.

```
1   request_vector               = ''0010111010'';
2   B <= not request_vector      = ''1101000101'';
3   C <= B + 1                   = ''1101000110'';
4   gnt <= request_vector and C  = ''0000000010'';
```

Listing 4.2: *First part of the arbiter. Find first non-zero from right.*

At line 20 in listing 4.1 the current request vector is masked with the previous grant. This is done in order to create the round robin behavior. Gntm contains the most recent grant and is updated each time a grant has been issued. See 4.3 for two examples. One just after reset, and then one additional. The statement creates a vector where all elements to the left of the '1' are set to '1', creating a mask. The rest will be zero.

```
1  --Example one. Gntm all zeros. No previous grants have been
       issued or
2  --cycle is reset.
3    request_vector               = ''0010111010'';
4    gntm                         = ''0000000000'';
5    gntm - 1                     = ''1111111111'';
6    A <= (gntm - 1) or gntm      = ''1111111111'';
7    B <= not A                   = ''0000000000'';
8    reqs <= request_vector and B = ''0000000000'';
9
10 --Example two. Gntm has non-zero elements.
11   request_vector               = ''0010111001'';
12   gntm                         = ''0000001000'';
13   gntm - 1                     = ''0000000111'';
14   A <= (gntm - 1) or gntm      = ''0000001111'';
15   B <= not A                   = ''1111110000'';
16   reqs <= request_vector and B = ''0010110000'';
17   --If (A and -A) is done on reqs, then the grant vector
18   --will be ''0000010000''
```

Listing 4.3: *Second part of the arbiter. Mask creation*

At line 21 in listing 4.1 it finds the first non-zero element from the right. This time in masked variant of the request_vector, reqs.

Now it is just a matter of choose which grant vector to use. The reqs vector being non-zero implies that a complete round robin cycle is not done, so use gnts. If reqs is all zeroes, either the circuit has just been reset, or a complete cycle is done and it should start over. In this case, use gnt instead. All this in one clock cycle and four lines of VHDL.

Looking at the resulting hardware in fig 4.5 it is easy to see that there are two adders in series. This will probably result in a quite slow arbiter for large request vectors, although easily pipelined if needed. Still, the arbiter proposed by [Krill, 2009] has no prioritizing scheme.

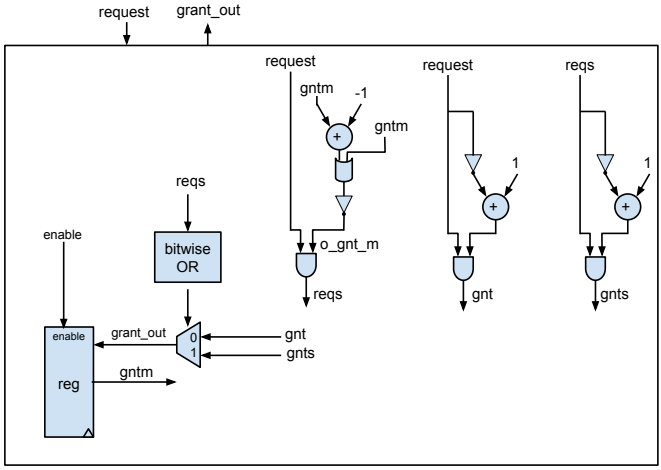

Figure 4.5: *Resulting arbiter logic*

4.5.1 Priority

Using an arbiter with no priority features means that this has to be solved in another manner. A proposal for a simple prioritizer is presented in this section. It features three levels of priority using three [Krill, 2009] arbiters in parallel.

Each source is given a three-bit wide priority vector that is one hot encoded. A priority level of one is coded to “001”, two to “010” and three “100”. Stacking priority outputs from each source creates a matrix with each source’s priority as rows and three columns. Cutting out each column and transposing it, the three resulting vectors contain information about which sources that are connected to the current priority level. And:ing these three vectors with the request vector from the sources creates three vectors where the current requests for each priority level are indicated. Running each of these generated vectors through separate arbiters in parallel results in three grant vectors, one for each priority level. It is then a simple thing to choose which of the resulting grant vectors to use. See fig 4.6.

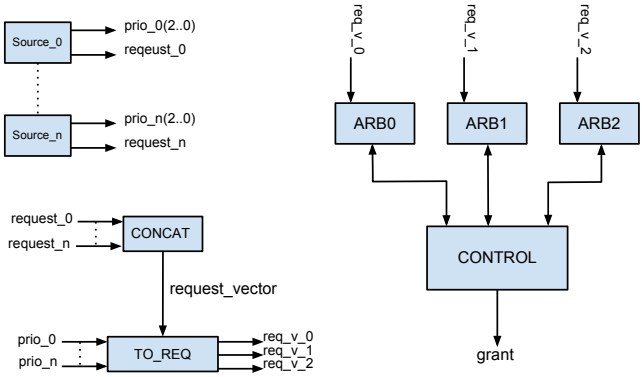

Figure 4.6: *Prioritizer block diagram*

This proposal increases the already high gate delay of the arbiter. Once again though, it is easily pipelined if needed.

4.6 The full system

In this section a block diagram of the entire system is shown and walked through. See fig 4.7 for the top level of the source side FPGA architecture.

Each source, S_x, are connected to a synchronization FIFO. Each FIFO in turn is connected to one framer, F_x. Inside the framer the incoming data is buffered until ready and then requests the AXI4-streaming interface to the Aurora core. Once given access it starts to transmit data to the destination FPGA. The arbiter control unit monitors the request outputs from the sources and takes action depending on priority of the sources, as well as previously granted sources. An ACK module receives data from the destination FPGA containing ACK/NACK messages and forwards these to the correct framer.

At the destination FPGA, the data is received on the RX ports of the Aurora interface. Based on information in the header, the RX control module gives the appropriate receiver access to the bus. Then it is up to the receiving application to handle the incoming data. See fig 4.8.

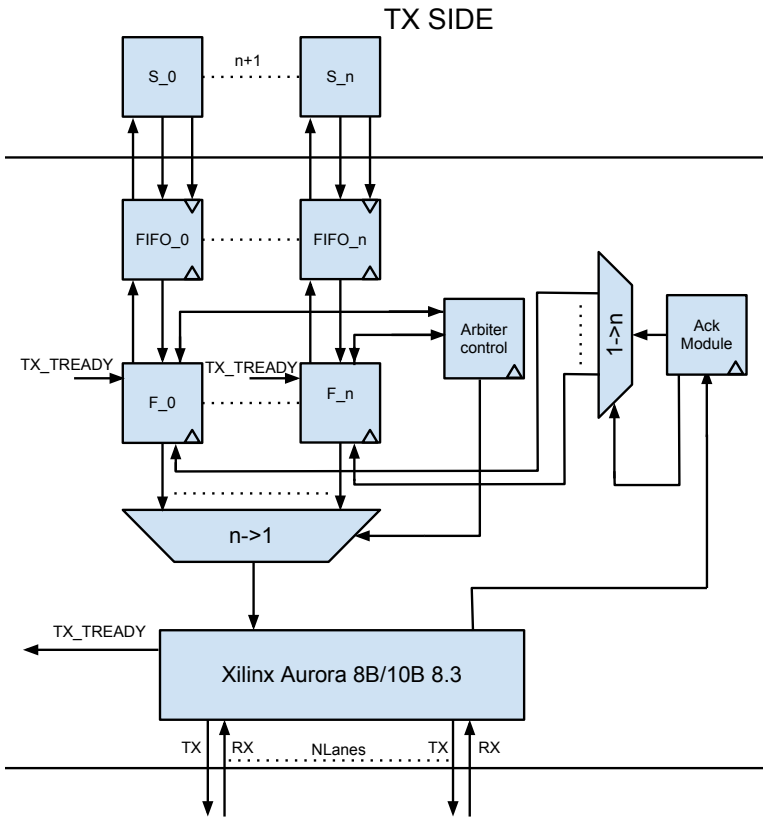

Figure 4.7: *Source side top level*

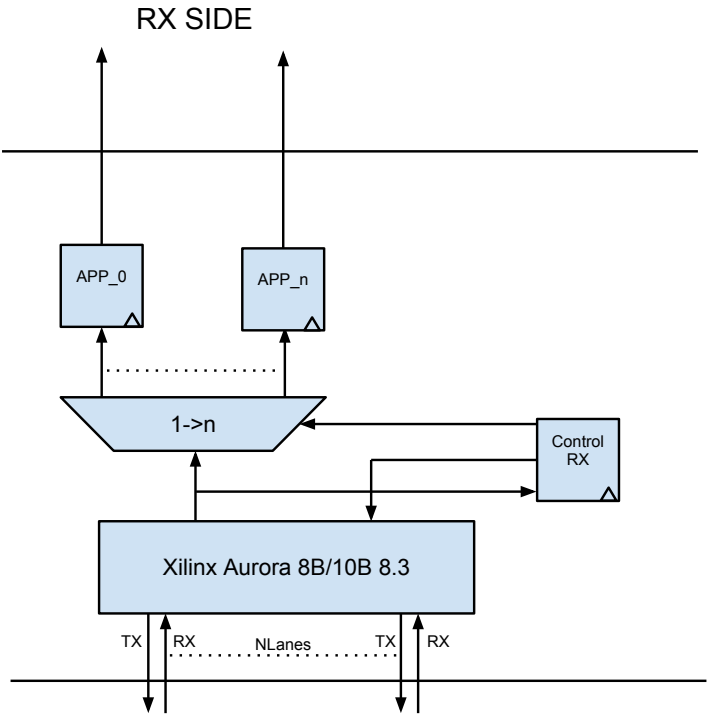

Figure 4.8: *Top level of receiving* FPGA *architecture*

5
Test Platform

In this chapter, the testing strategies for the implemented architecture will covered. In order to test the architecture during the design phase a VHDL testbench has been created. This testbench has also been modified in order to run at a target FPGA.

5.1 VHDL testbench

The VHDL testbench simulates one source FPGA sending generated data over the Aurora channel and one destination FPGA checking this generated data.

The data generator consists of one PRBS module which generates a 32-bit long PRBS pattern using an LFSR. The PRBS pattern is commonly used in testing serial links due to its completely deterministic, but random looking behavior. The data generator can be configured using generic parameters in its instantiation to control how long a user data frame should be, and how often these should be generated. An identical PRBS generator is connected at the receiver side and is used to compare the incoming data. In the complete system, with error detection and re-transmission enabled, *any* faults in this comparison are considered a critical error and must not happen. The tester can connect an arbitrarily large amount of data generators as sources to the complete design. This tests not only the correctness of the Aurora channel, but also the requirement that any number of sources should be able to connect to the implementation. In order to test functions such as error recovery when simulating, the TX/RX wires are fed through an error-injecting module $e(t)$. See fig 5.1.

By setting generic parameters in the testbench, the tester can control how large frames to be generated in the source generator, how many sources to connect and how many Aurora lanes to be used in the simulation.

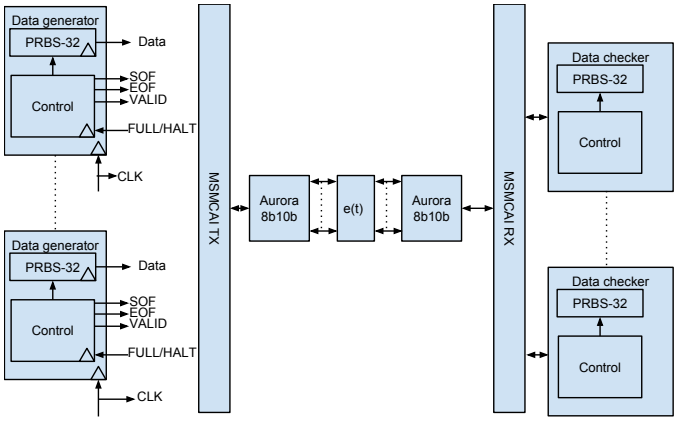

Figure 5.1: *MSMCAI testbench*

The design is simulated in Modelsim and any errors in the comparisons at the receiver are asserted in the console output. When running in hardware, counters are available that count the number of erroneously received words at the receiver. These should never count if everything is working. These counters can also be probed by using Chipscope. To make sure that everything is alive, counters are available that counts the number of correct words received. Worth to note is that because of how the pattern checking device works, if only one error occurs, the right word counter will never again count upwards.

A simple script has also been written in order to test combinations of number of sources and number of lanes, preferably running over night due to the quite slow simulations. Remember that the transceivers run at 1GHz+, requiring picosecond resolution in Modelsim.

5.2 Chipscope PRO

In order to test and debug the design when running on target, Xilinx offers an IP core called Chipscope. Chipscope can be configured as a logic analyzer and downloaded to the target together with the design. This way, the user is able to trigger on signals and debug in a way that is similar to using an ordinary logic analyzer.

5.2.1 Chipscope PRO iBERT

Another IP available in the Chipscope portfolio is the iBERT core. iBERT is short for integrated bit error rate tester and can be used to check the health of the transceiver channels. This IP cannot be downloaded to the target together with the user design, but come in handy when the user wants to check the status of the transceiver connections. If the iBERT core reports a failure, then the user design will most certainly not work.

It also has the possibility to plot eye diagrams of the channels after equalization filtering inside the transceivers has been done. This is most interesting since the eye might be completely closed if probing with an oscilloscope at the FPGA pin, but might have a clear

opening after the equalization filter.

Another feature is that the user can change some of the transceiver settings such as pre/-post emphasis settings and line driving strength without re-configuring the FPGA. This allows the user to fine-tune the transceiver settings so that maximum performance is achieved for the current physical link. These settings can then manually be edited in the transceiver setting files.

5.3 Xilinx KC705 evaluation board

The Xilinx KC705 evaluation board features a Kintex-7, XC7K325T, FPGA, together with some peripheral devices. Noteworthy is the FMC connector used in order to route some of the transceiver pins out of the chip, see number 30 in fig 5.2. Connected to this FMC connector is an FMC module from HiTech Global, see fig 5.3. The FMC module has 32 SMA connectors that can be used to connect the FPGAs transceiver pins. On the FMC module is also a low jitter oscillator which output frequency is controlled by a dip-switch. Unfortunately, only four out of the total 16 available transceivers on the XC7K325T FPGA are routed to the FMC connector, making it impossible to test using more than two Aurora lanes.

The SMA cables used in the setup are 12 inches long. The total path for the signals is approximately 30 inches.

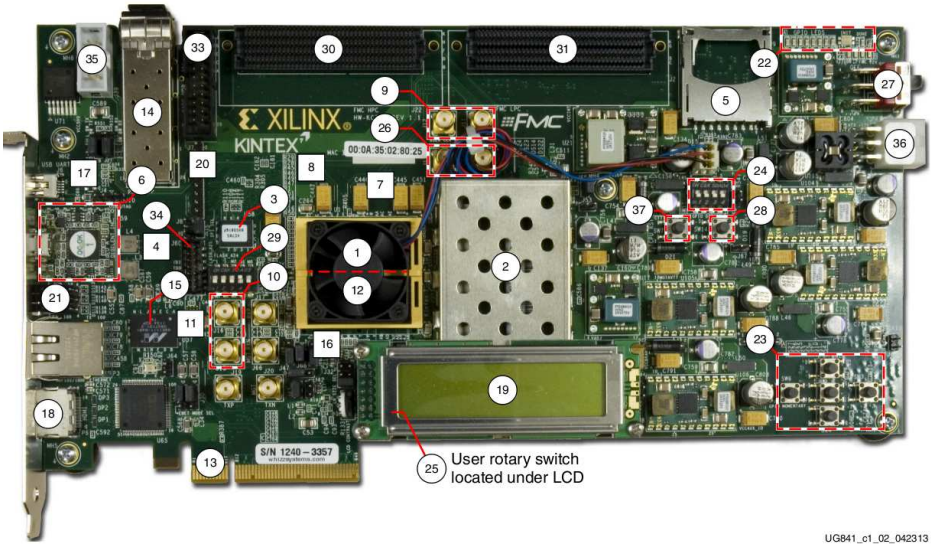

Figure 5.2: *KC705 Kintex 7 evaluation board*

The same testbench used for simulation is easily ported to work on actual hardware, assuming stimuli for clocks, reset and such exists on board. Instead of feeding the transceiver TX and RX lines to the receiver in software, the signals are fed out of the chip via the FMC module and SMA cables, back to the receiver at the same FPGA. There is

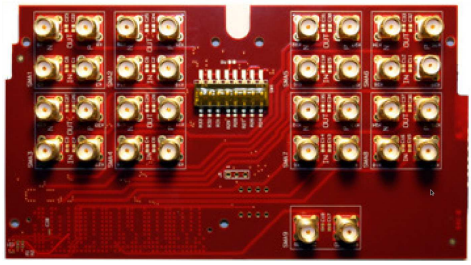

Figure 5.3: *FMC expansion card from HiTech Global*

no easy way of injecting errors here, so the error module $e(t)$ in fig 5.1 is the actual errors generated from impedance mismatches etc.

5.4 Test strategies

The following features have to be tested.

5.4.1 Error detection and recovery

Test this in simulation by generating errors on the TX/RX lines using the error inject module. The design should be robust to errors in data and lost ACK/NACKs.

5.4.2 Transmit buffers

The transmission buffers have to be tested with different fill ratios. In a multi-lane implementation, the buffers have to be tested with partial words, almost empty buffers, almost full buffers and full buffers. Test this by varying the amount of data generated by the source generators and how often they should generate new frames.

5.4.3 Number of sources, arbiter and priority

The number of sources that shares the MSMCAI is easily varied by changing parameters at the test bench top level. Test by varying this parameter and make sure that no receiver is starving. This is easily probed in both Modelsim and Chipscope by watching the grant vector output. The priority for each source is changed by setting the priority port for each connected framer.

5.4.4 Number of Aurora Lanes

Change this parameter in the testbench top file. Make sure that no erroneously data is received at the receiver by watching the log output in Modelsim. When running in hardware, an error counter is connected to each receiver, which can be monitored using Chipscope.

6
Demo PCB

This chapter will introduce the demonstration PCB that was also created during the thesis work.

6.1 Purpose

The purpose of the demo PCB is for SAAB Dynamics in Linköping to create a hardware platform for the Xilinx Kintex-7 FPGA family, perhaps using it in future projects. It is also intended to see how far the FR-4 PCB technology can be pushed in terms of frequency and bit rate.

6.2 Schematic

The platform contains only the most necessary components to get the FPGA up and running, together with some peripheral units. The platform consists of one UART driver circuit, a flash memory for power on configuration, status and debug LEDs, oscillators for initialization and transceiver reference clocks, high speed connectors for transceiver routing and six DC/DC voltage regulators. Connectors for logic analyzer probes and some general GPIO are also available.

Two high-speed connectors with 40 pins each are used, one male and one female. The plan is to connect two PCBs together, but also to connect the PCBs to a platform where several cards are connected. The board will feature eight MGTs routed to the connectors.

The platform is based on previous SAAB Dynamics designs, the KC705 evaluation board and data sheet requirements.

6.2.1 Voltages

The FPGA needs 1.0V for internal logic and block RAMs, 1.8V for auxiliary and high-speed GPIO pins, and 3.3 volts for ordinary GPIO. These are generated using switched DC/DC regulators, which unfortunately generates a bit of noise on the power rails but offer very good efficiency. The Transceivers needs much cleaner power rails, so these are fed separately using linear regulators with lower efficiency. Here, 1.0, 1.2 and 1.8 Volts are needed.

6.3 Layout

The layout work will be outsourced to experts working for SAAB Dynamics due to the knowledge needed for routing high-speed differential traces and lack of time.

7
Results

In this section, the results using the MSMCAI architecture will be presented. Both in simulation and on target. In addition, some analog characteristics of the system will also be covered.

7.1 VHDL implementation

The VHDL implementation was done in a manner that it is completely generic with respect to the amount of sources connected. Unfortunately, creating a fully generic solution with respect to the amount of Aurora Lanes was not possible. A semi-generic implementation with respect to the amount of lanes was created. Since the amount of lanes is set when generating the Aurora core in Coregen, a separate entity for each value of $N_{lanes} = [1, 2, 4, 8]$ had to be generated, although all interfaces in the proposed architecture is automatically widened or shortened in order to work with the current setting of N_{Lanes}.

7.2 Test and validation in ModelSim

7.2.1 Number of sources, arbiter and priority

In the first test, eight sources are simulated. Each source has an ID ranging from zero to seven. Each source are given the priority prio = ID modulo 3, where a priority of zero has the highest priority. Each source generates $(1 + ID) \times 32$ bits of data every 10th clock cycle, partitioned in 32-bit words. The sources run at 100 MHz and the system clock is 6.25GHz/40 = 156.25 MHz. The number of lanes is set to one.

In figure 7.1 at page 41, the start of a Modelsim simulation is shown. Prior to cursor1, the system has been started by issuing an external reset. This triggers the simulation models

of the transceivers and the Aurora core to run its reset and channel initialization circuitry. This takes approximately five μs. During this time the eight sources connected has had plenty of time to start generating data. The sources have started to fill the synchronization FIFO and has also begun to fill the transmit buffer inside the framers. Looking at the request_o signal at cursor1 all framers request to use the bus but framer zero. This framer has already been granted the bus, but is waiting for the Aurora interface to assert its TX_TREADY signal.

Just before cursor 1, the CHANNEL_Up signal has been asserted, indicating that the Aurora channel is now up and ready to receive input. At cursor 1 the framer starts the transmission by asserting the TX_TVALID signal and by placing data on the TX_TDATA bus. The first data on the bus is the header followed by one 32-bit word before TX_TLAST is asserted. The next clock cycle both TX_TVALID and TD_TLAST are de-asserted, completing one transmission. At cursor 2, after 281.6ns or 44 clock cycles, the data arrives at the receiver application. During this time the seven other framers has been able to use the channel. When the RX_TLAST signal is asserted, the CRC signals are checked for errors and an ACK/NACK header is sent back to the source FPGA at cursor 3.

After another 289 μs at cursor 4, the ACK/NACK header has been received at the source. The ACK/NACK is forwarded to the correct framer and since there is data in the transmit buffer the request_o signal from source zero is asserted again and a new transmission is started at cursor 5.

The signal mux_control_integer is the unsigned equivalent of the one hot encoded grant vector. A grant vector with the value “0001000” has the unsigned equivalent of a 3, a vector with value “0000001” is mapped to a 0. Since all source framers request the bus at the same time, the framers are chosen with respect to their priority. Looking at the mux_control_integer signal, the order in which the framers are chosen are 0,3,6,1,4,7,2,5, which is to be expected since source 0,3 and 6 belong to the priority 0 group, 1,4,7 belong to the priority 1 group and 2,5 belong to the priority 2 group which has the lowest priority.

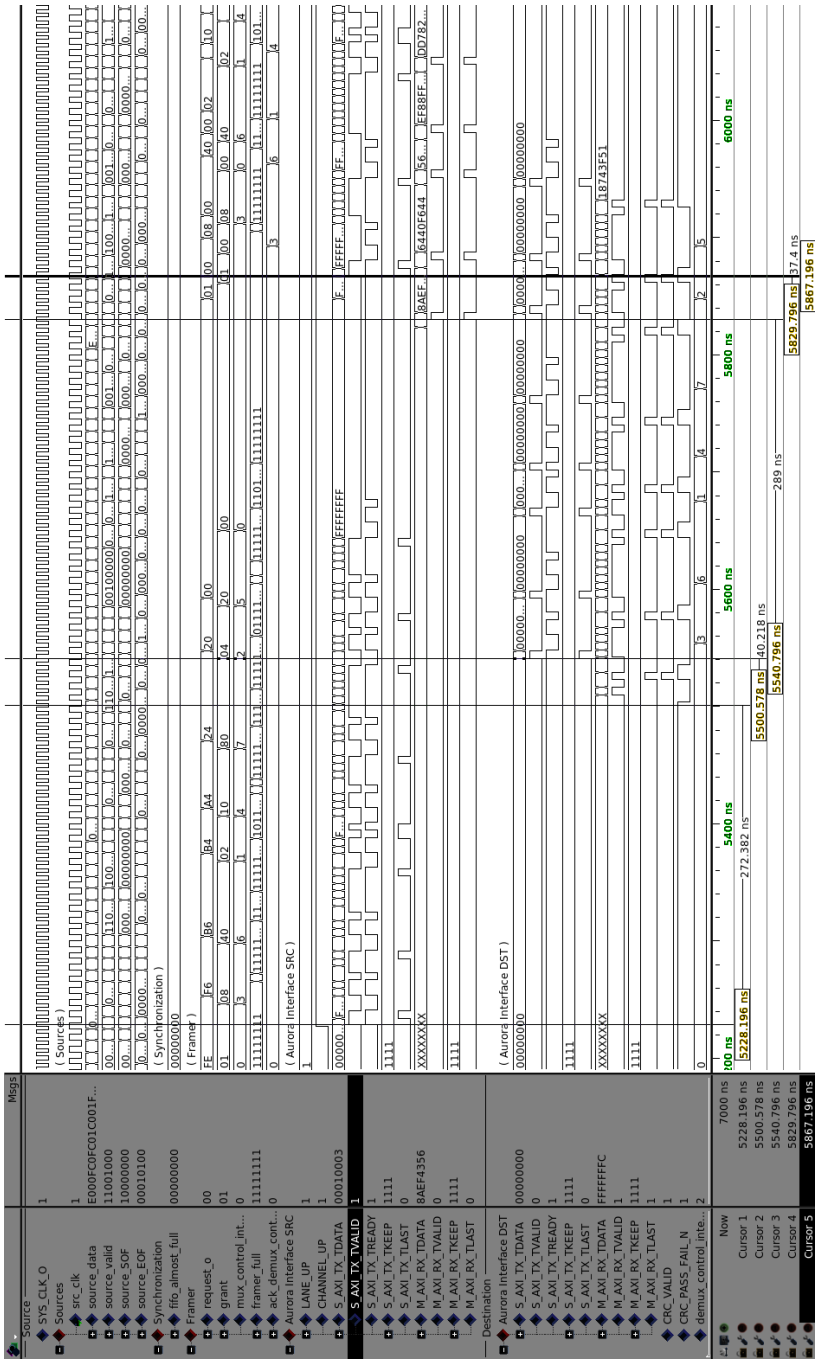

Figure 7.1: *Simulation run with eight sources with different priority levels*

7.2.2 Number of Lanes

In the second test, most settings are the same as in the previous test. This time the amount of data generated from each source is set to generate 200 32-bit words each 1200th source clock cycle. This translates to $200 \times 8 \times 32 \times \frac{100 \times 10^6}{1200} = 4.2667$Gbps, which is getting close to the limit of the link. The capacity of the 6.25Gbps link after 10b/8b encoding is 5Gbps, not taking into account overhead from the MSMCAI protocol. The throughput of the system after the impact of protocol overhead and bit error rate will be discussed further in section 7.6. The only thing changed in the source code is a value of how many lanes to use; the rest is solved by generics in VHDL

In figure 7.2 at page 43, a one-lane implementation is used. Looking at cursor 1, 2, 3 and the TX_TVALID signal, selected in black, it is obvious that the link is close to its limit. It only has a few percentage left of its bandwidth before some lower priority source will begin to starve their receiver. The transmitter uses $100 \times (1 - \frac{726}{11270.4+726}) = 93.95\%$ of the available bandwidth. If this were a channel where bit errors are common, there is a possibility that this transmitter would need more than 100% of the available bandwidth due to retransmissions.

In figure 7.3 at page 43, the number of lanes are set to two and the transmitter uses $100 \times (1 - \frac{6131.2}{5875.2+6131.2}) = 48.93\%$ of the available bandwidth which is less than half of that needed when only using one lane. Also worth noting is the lane_up signal that is now a vector since two lanes are used.

The reason that the percentage is more than halved has to do with the transmit FIFOs. When using a one-lane implementation there is only one 66 word deep FIFO, meaning that the transmitter has to send $ceil(\frac{200}{66}) = 4$ times to get the whole message over to the receiver. This increases the amount of headers sent, and increases the amount of time a framer might need to wait for an ACK/NACK, which will lower the utilization. When using the two lane implementation, there are two transmit FIFOs with 66 words each, meaning that only $ceil(200/132) = 2$ transmissions are needed.

The right_count_slow signals are included in figures 7.2 and 7.3 to show that the implementation really is working. These are counters, placed in the receiver application, which counts all correct words received.

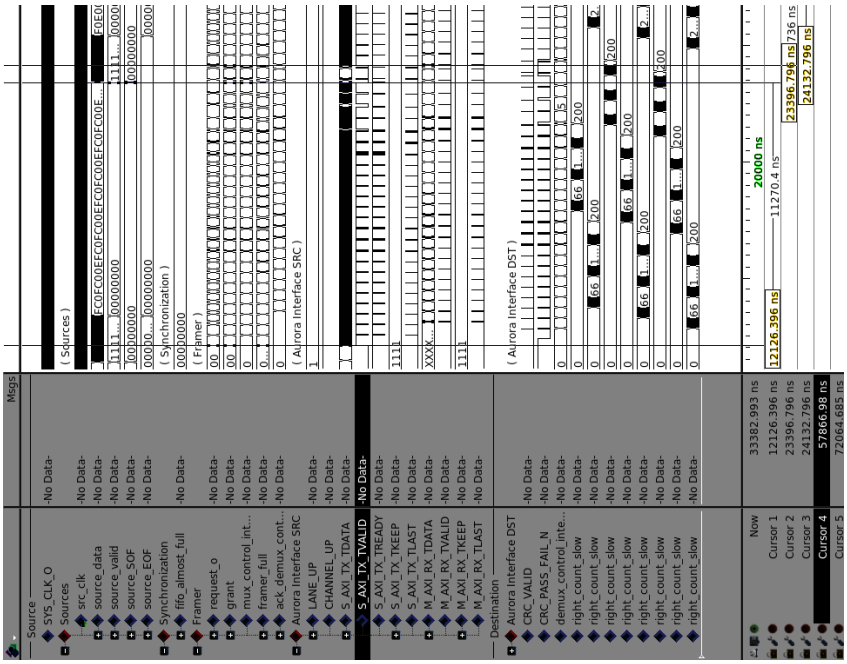

Figure 7.2: *Simulation run with one Aurora lane*

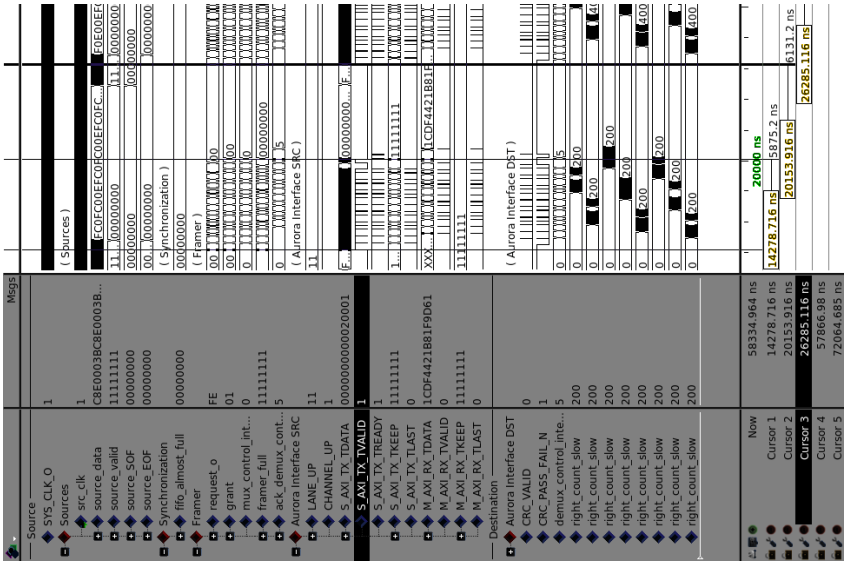

Figure 7.3: *Simulation run with two Aurora lanes*

7.2.3 Error detection and recovery

To test the robustness with respect to bit errors on the link, an error injector in the testbench is enabled. It flips the bits on the RX/TX lines in short intervals, simulating errors on the channel. A two-lane implementation with eight sources running at 6.25Gbps and 100 MHz respectively are used. Bits are flipped on the TX/RX one million times per second. With a line rate of 6.25Gbps, this translates to a bit error rate, BER, of $\frac{1\times10^6}{6.25\times10^9} = 1.6 \times 10^{-4}$. A very bad channel indeed, but good for testing the robustness.

In figure 7.4 at page 45, follow the signals **demux_control_integer**, **topRX/CRC_VALID** and
topRX/CRC_PASS_FAIL_N. At cursor 5, receiver application 0 is just about to finish receiving data, and the next in line is receiver 3. At cursor 1, receiver 3 is just about to finish, thus checking the CRC signals. Since the PASS_FAIL_N signal is not asserted the receiver returns a NACK to the transmitter, issuing a retransmission. In this particular scenario, receiver 3 receives two more CRC errors at cursor 2 and 3. Not until cursor 4 does the non-faulty frame arrive at the receiver, and the right counter for receiver 3 starts to count upward.

Figure 7.5 at page 46 shows a sequence where the ACK/NACK header from the receiver is corrupted on the way back to the transmitter. At cursor 1, the frame arrives from framer 5 and is received without CRC errors. Shortly thereafter at cursor 2, observing the CRC signals in "toppen", the CRC_PASS_FAIL_N signal is not asserted, meaning that the ACK/NACK message contained errors and must be discarded. The framer connected to source 5 then has to wait until the ACK/NACK timeout counter reaches zero which happens at cursor 3, see signal /framer/timeout_counter (this resets to 200 when it reaches zero). When the counter reaches zero, framer 5 is granted the bus again and begins to retransmit the same data that has already been accepted at the receiver. Unfortunately at cursor 4, the data from framer 5 is found to have errors in it and another ACK/NACK is sent back to the transmitter. This time the NACK is received without errors and yet another retransmission is initiated. Finally, at cursor 5 the data is received error free. However, since the data has already been accepted once, this frame is discarded at the receiver, thus not incrementing the right_count counter.

This is why the header has to contain some kind of seed for the CRC calculator, because the circuitry that checks for duplicate frames do this by checking the CRC of the frame. If two frames with exactly the same data were sent and actually should be received, then these two are separated by the seed in the header, generating two different check sums.

Figure 7.4: *Simulation run with errors injected. Error detection and recovery.*

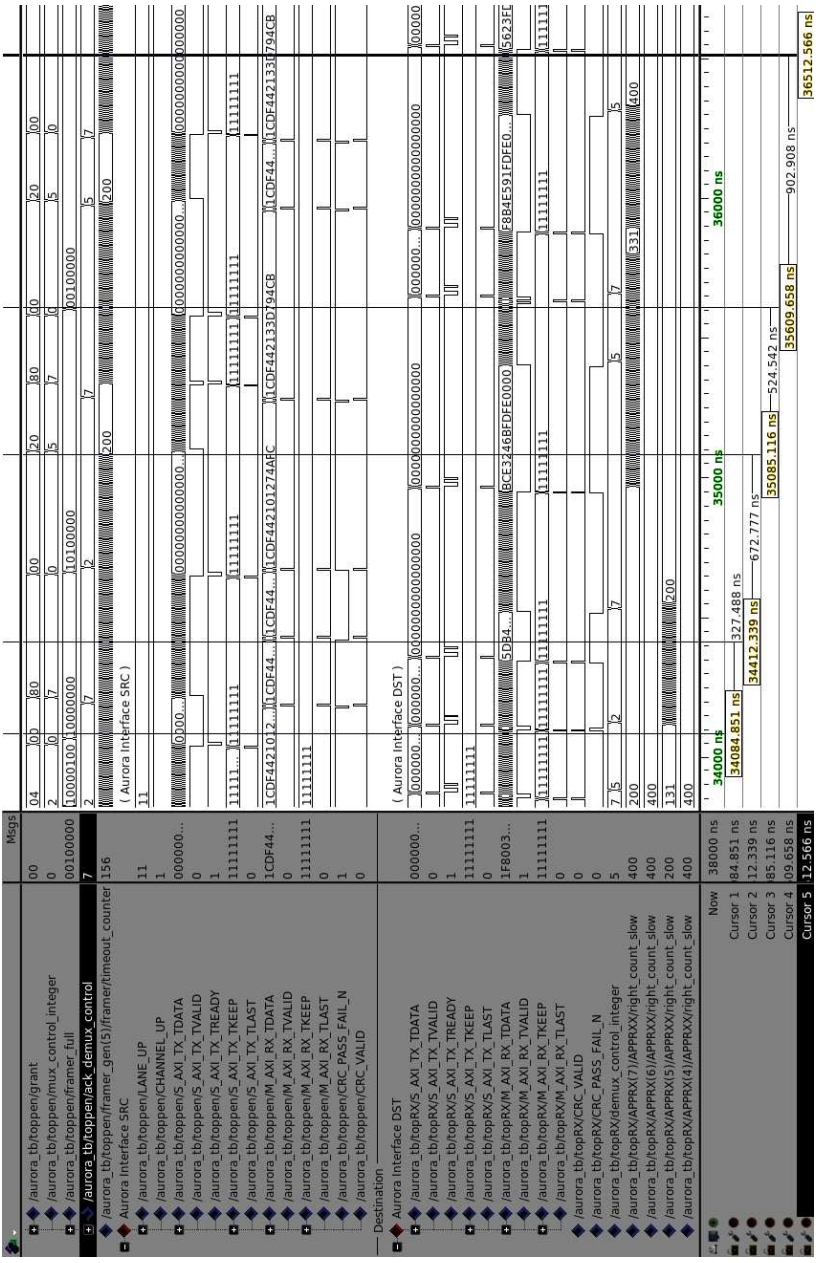

Figure 7.5: *Simulation run with errors injected. Lost ACK/NACK*

7.2.4 Transmit buffers

In order to test the transmit buffers two simulations are ran. In the first, two lanes are used with eight sources. Each source generate $1 + ID$ amount of data, thus testing one word in only one of the two transmission buffer FIFOs, testing one word in each, and so on.

In the second test, each source sends $128 + ID$ amount of 32-bit words in each frame. Thus just about to fill the both FIFOs, filling them exactly and then forcing the implementation to use chop up the longer user frame into two transmissions, since the FIFOs became full. Remember when using two lanes, the amount of 32-bit words that can fit inside the buffer is $2 \times 66 = 132$ (actually the chopping up threshold is $2 \times 66 - 1 = 131$ due to implementation issues). The error injector is disabled for this test in order to simplify reading the results.

In figure 7.6 at page 48, the first test is shown. Looking at the **right_count** signals, they increment as intended and no erroneous behavior is observed.

In figure 7.7 at page 49, small gaps between series of increments in the **right_count** signals can be seen at cursor 1, 2, 3 and 4, indicating that the 131+ frames are being chopped up.

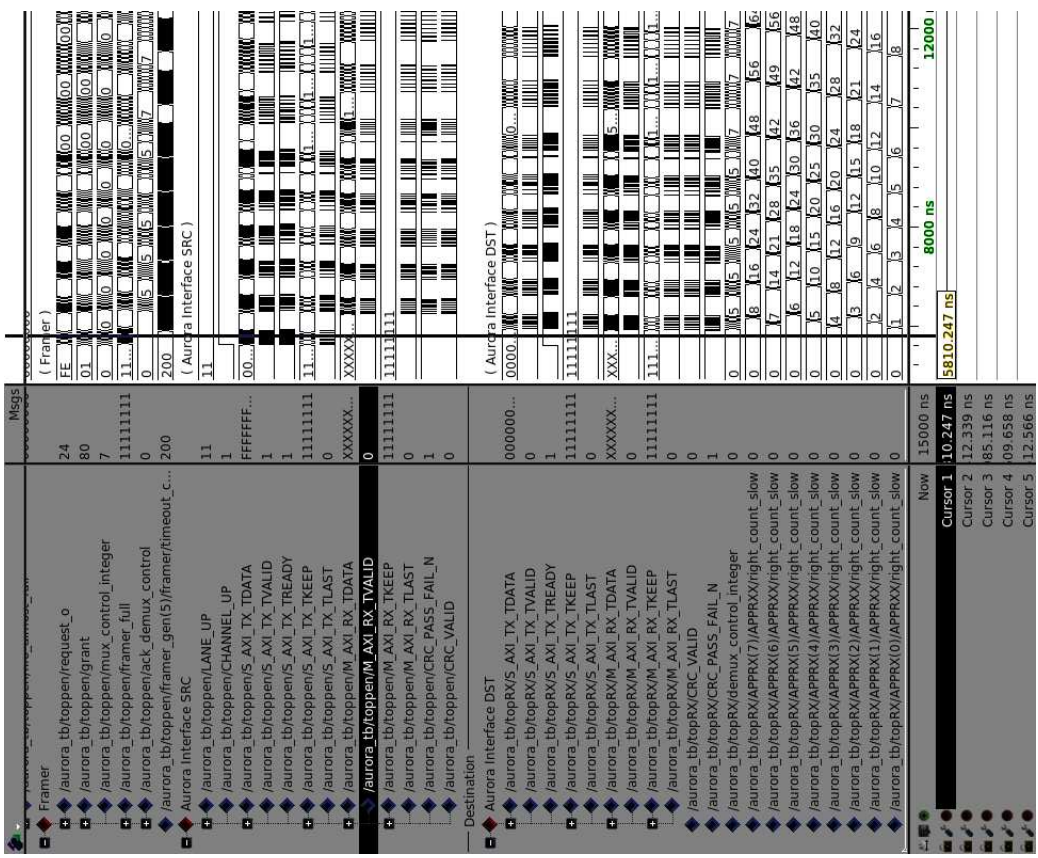

Figure 7.6: *Simulation run at the boundaries of small user frames filling the transmit buffers*

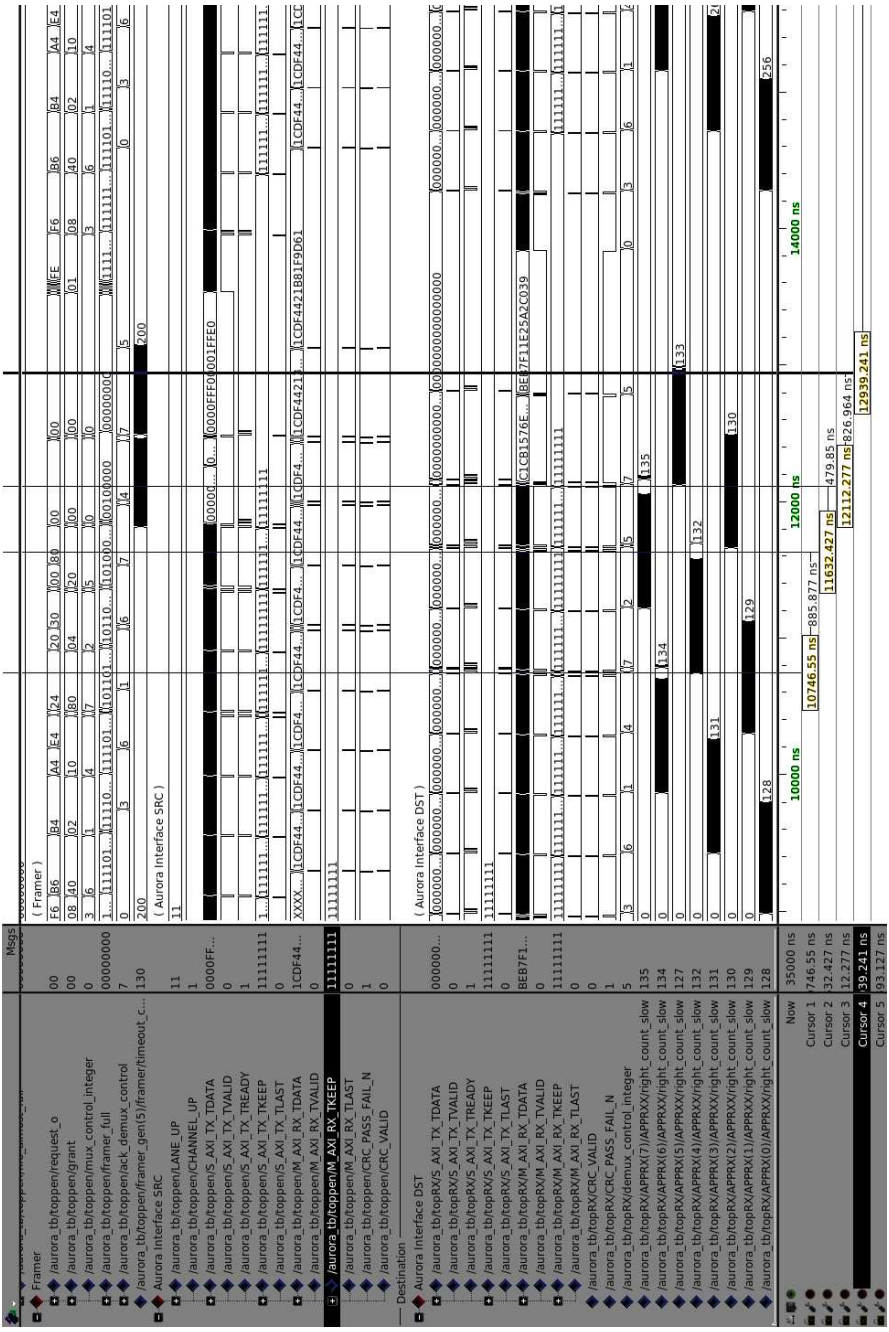

Figure 7.7: *Simulation run with sources generating frames on the boundary of being chopped up into several transmit frames*

7.3 Test and validation on KC705 evaluation board

Since the design is validated using the simulation test bench and recorded in section 7.2, a shorter demonstration of the working hardware is shown here.

7.3.1 Long run simulation

In the following on target validation, a two-lane implementation is used running at 6.25Gbps each. The sources generate user frames with a size of 1048 words as fast as the previously user frame was done. This will not only test the maximum possible throughput of the system, but also test that the functionality of halting the sources when the synchronization FIFOs become full. This generates approximately $8 \times 32 \times 100 \times 10^6 = 25.6$Gb of data each second, greatly outproducing the capacity of the $6.25 \times 2 = 12.5$ Gbps link. Each source is given the same priority in order to show that all source-destination pairs are alive. The transmitter and receiver are located in the same FPGA due to only having one evaluation card available. This also simulates that one FPGA can contain both receivers and transmitters that work independently of each other.

Figure 7.8 at page 51, is a screen capture from Chipscope after the design has been up and running for 24+ hours. The image shows the lowest bit of the **right_count** counters from each receiver toggling. This shows that no faulty word has been received during this time. As previously stated, if one wrong word is received, the counters stop counting, indefinitely.

Figure 7.9 at page 52, is a screen capture where a CRC error has occurred at the receiver, showcasing that the error detection and recovery circuitry works in hardware.

In the bottom of fig 7.8 there are some statistics about the current run. In the receiver controller at the destination side, counters are counting the total number of words received, the number of valid words received, the number of CRC checks done and the number of faulty CRCs received each second. To get the total bit rate it is necessary to multiply the value by $N_{Lanes} \times 32$.

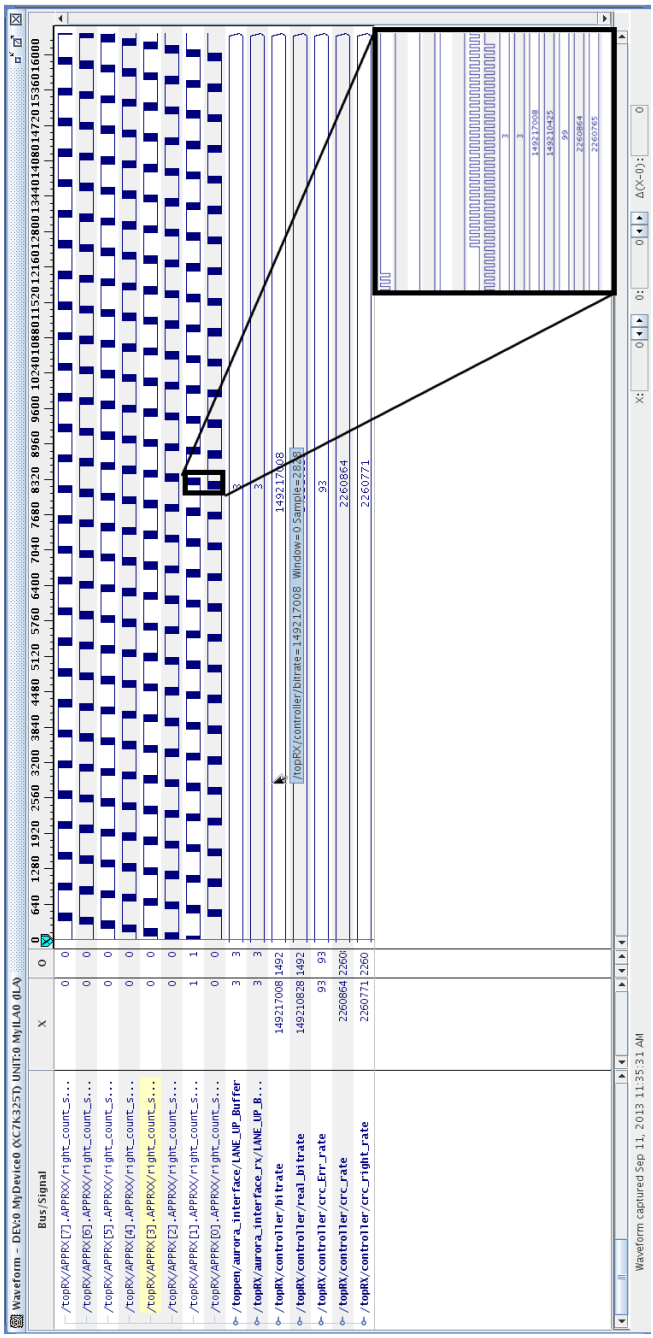

Figure 7.8: *Screen capture of implementation running at the KC705 platform.*

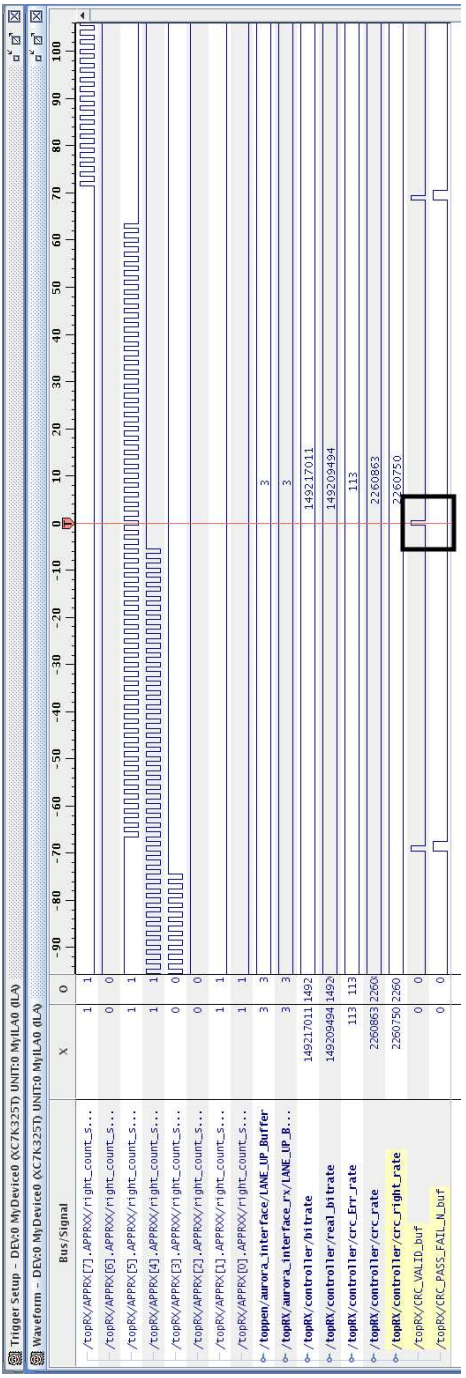

Figure 7.9: *A* CRC *error has occurred*

7.4 Area and resources

The resources occupied by the complete implementation, receiver and transmitter in one FPGA, in terms of look up tables, flip-flops and block rams are presented in fig 7.10, 7.11 and 7.12

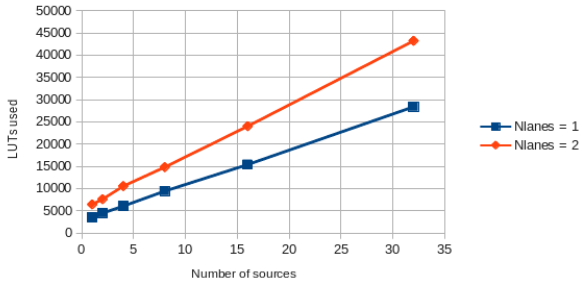

Figure 7.10: *LUTs for a one and two-lane implementation with different amount of sources connected*

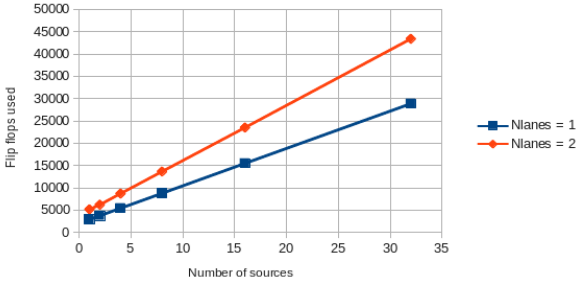

Figure 7.11: *FFs needed for a one and two-lane implementation with different amount of sources connected*

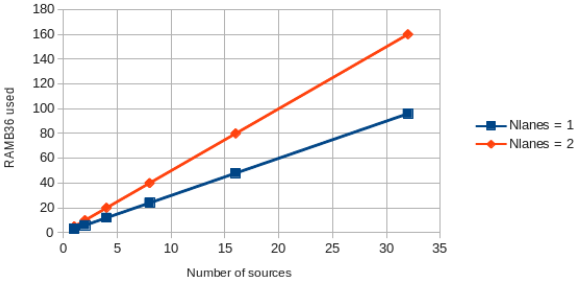

Figure 7.12: *36 bit block RAMs needed for a one and two-lane implementation with different amount of sources connected*

The number of LUTs, FFs and block RAMs grows linearly and for a two-lane implementation the difference between the amount of resources needed is less than double compared to a one-lane implementation. This is expected since most of the interfaces between the units in the architecture have to be doubled in order to use the wider data path to the Aurora core. These figures also include the data generator and checking logic.

7.5 Clock frequency

In figure 7.13, the maximum clock frequency possible after synthesis is presented. The maximum frequency decreases with the amount of sources connected which is expected, since the critical paths in the arbiter will be larger for a higher number of sources. .

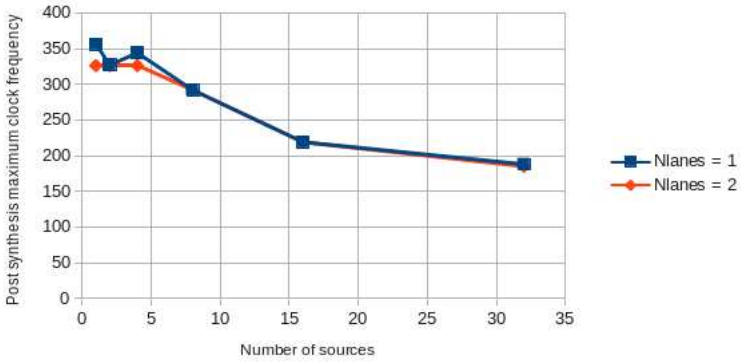

Figure 7.13: *Maximum clock frequency after logic synthesis*

The maximum clock frequency after the place and route stage is above the 156.25 MHz needed to meet the timing requirements from the Aurora core running at 6.25 Gbps. Up to 32 sources was synthesized, although using 32 sources with ≈ 35 I/O pins each would need $32 * 35 = 1120$ I/O pins on the FPGA to connect them.

The difference in speed between the one and two-lane implementation is very small, because many of the computations done inside the Aurora core is done in parallel when using more than one lane.

7.6 Throughput

MSMCAI suffers from three sources of reduction in throughput. Overhead from the Aurora 8b/10b protocol, headers and clock correction sequences used in the Aurora protocol.

There a 20% overhead when using the 8b/10b protocol, for every ten bits transmitted, only eight contain “real” data.

When using the AXI-4 interface to the Aurora core, there are a two clock cycle delay after one data frame has been sent, before the TX_TREADY signal is asserted again. See

fig 7.14. Lastly, each frame is accompanied with a one-word header, translating into one additional clock cycle without "real" data. A clock correction in this implementation is 4 clock cycles long and is done each 2500th clock cycle

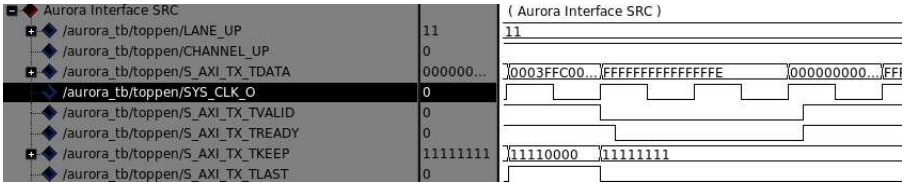

Figure 7.14: *Overhead from TREADY signal after last word in frame*

7.6.1 Theoretical limits

A formula for the theoretical maximum data rate on a multi-lane system is given in equation 7.1.

$$MaxDataRate = N_{Lanes} \times LaneRate \times 0.8 \times \frac{FrameSize}{FrameSize + 3} \times \frac{2496}{2500} \quad (7.1)$$

Inserting values for a two lane system with a $FrameSize$ of 66 and a lane rate of 6.25 Gbps equals to

$$MaxDataRate = 2 \times 6.25 \times 0.8 \times \frac{66}{69} \times \frac{2496}{2500} = 9.54991304348[Gbps] \quad (7.2)$$

Gbps. For a one-lane system it becomes

$$MaxDataRate = 1 \times 6.25 \times 0.8 \times \frac{66}{69} \times \frac{2496}{2500} = 4.77495652174[Gbps] \quad (7.3)$$

Given BER, the packet error rate, PER, can be approximated by equation 7.4, assuming the errors are evenly distributed and that the BER is small [Baseri and Motamedi, 2013].

$$PER = [1 - (1 - BER)^L] \approx (1 - e^{-BER \times L}) \quad (7.4)$$

L is the number of bits in one frame, $L = FrameSize \times N_{Lanes} \times 32$

The amount of packages per second is given by $PacketsPerSecond = \frac{MaxDataRate}{FrameSize \times N_{Lanes} \times 32}$
The cost of re sending one packet is $Cost = FrameSize \times N_{Lanes} \times 32$. The header is already taken into account from 7.1.

Using the PER, the throughput of the system can be calculated by using equation 7.5.

$$Throughput = MaxDataRate - PER \times PacketsPerSecond \times Cost \quad (7.5)$$

In figure 7.15, a number of lines are plotted to show the impact of an increased BER and increasing the $FrameSize$ variable. It shows that for each BER there is an optimal size for the frames, using the $FrameSize$ that generates the local maximum. Increasing the frame size further will lower the throughput, beacause the probability and cost of a CRC error in the frame increases.

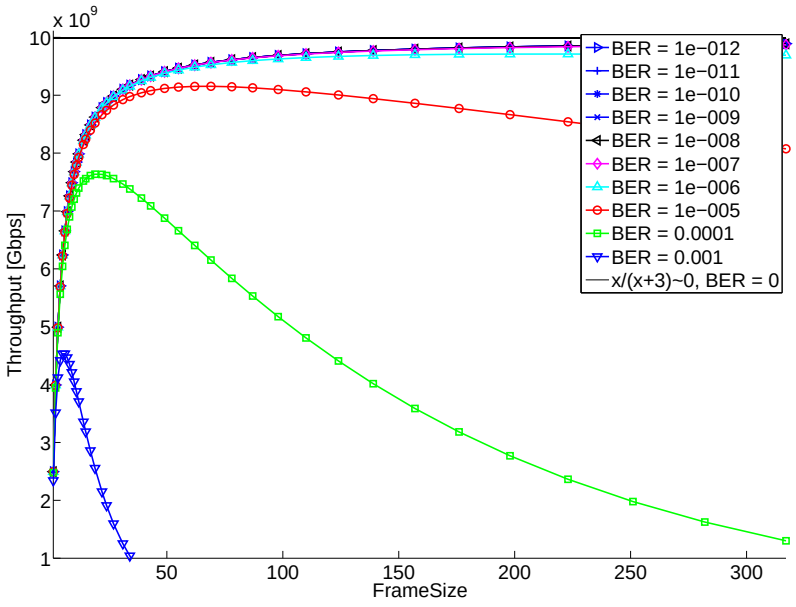

Figure 7.15: *Throughput vs. FrameSize and* BER

Studying the graphs further, it shows that increasing the frame sizes for a relativity low BER will only marginally lower the overhead induced by the header and AXI-4 protocol beyond a certain point. Since memory in the FPGA is limited, a buffer size of 100-200 words may be a good compromise, assuming the BER is low.

7.6.2 Measurements Vs. Theoretical limits

Using the counters inside the receiver control unit, real figures of throughput can be measured. The implementation uses two lanes at 6.25 Gbps and eight sources. Between resets of the device, there is a difference in the amount of CRC errors the receiver experiences. This is probably due to the receiver equalization filter, which trains itself during the initialization. The difference in CRC errors per second range from approximately 20-2000 CRC errors per second.

Due to the difference in the CRC errors, it is possible to evaluate the approximation of throughput in equation 7.5. Each value of CRC errors/s (Basically the current PER) translate into a BER by using the approximation in equation 7.4. In figure 7.16 the values of equation 7.6 is plotted, with respect to different levels of BER.

$$DiffToMax = MaxDataRate - Throughput \tag{7.6}$$

Where $MaxDataRate$ is the maximum possible throughput using equation 7.5 with a BER $= 0$ and a $FrameSize = 66$. $Throughput$ is the values given when using equation

7.5 with a BER $\neq 0$ and a $FrameSize = 66$. Throughput is also taken from actual measurements using the counters in the FPGA implementation.

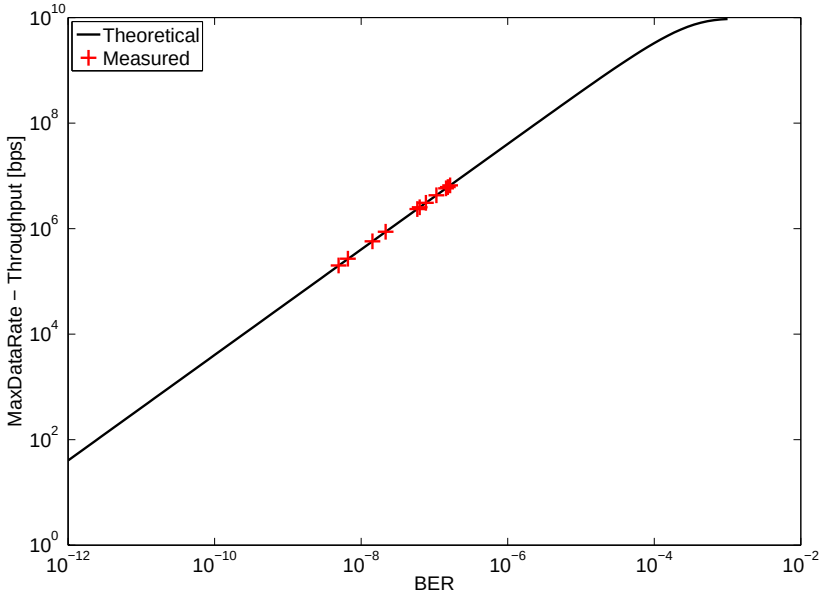

Figure 7.16: *The difference between maximum throughput with* BER *= 0 and actual throughput with* BER $\neq$ *0. Theoretical and measured values*

As can be seen, the difference between the two is small to say the least, meaning that the approximation in equation 7.5 is reasonable. Zooming in on the graphs shows that the difference between measured and theoretical values is in the order of 100-200Kbps.

Following the line in figure 7.16, it is possible to see that the difference between the theoretical maximum throughput and the measured throughput should reach zero if the BER reaches zero. The difference in throughput because of the BER experienced when running the design on target ranges from ≤ 1 Mbps to 6 Mbps, depending on the amount of errors.

Looking at the raw bit rate in the receiver, it averages around 9.549888512 Gbps. Raw bit rate means that the cost of retransmissions are discarded - all received bits are counted. Comparing the rate of bits in the system with the maximum calculated in equation 7.2 the difference is only $9.54991304348 - 9.549888512 = 0.00002453147 \approx 24.5[Kbps]$ which indicates that the assumptions made about overhead in equation 7.2 are correct.

In terms of total available bandwidth, which is 10Gbps after the 8b/10b encoding. The total overhead using a 66 word frame is $100 \times (1 - 9.549888512/10) \approx 4.5\%$, assuming that BER $= 0$. Increasing the frame size to 140 will cut the overhead percentage by half $(100 \times (1 - (\frac{140}{140+3} \times \frac{2496}{2500})) \approx 2.26\%)$.

When running a two-lane implementation at 3.125 Gbps, no CRC errors have been encountered.

7.7 Chipscope iBERT and Channel characteristics

Using Chipscope, it is possible to use an IP core from Xilinx called Chipscope iBERT. Using iBERT, it is possible to check the health of the high-speed serial links, as well as changing some of the parameters on the fly in order to maximize the performance of the links. In figure 7.17, a screen capture of a non-edited iBERT core is shown, running with a 6.25 Gbps line rate. By using a 31-bit PRBS test pattern, it is possible to force errors on the transceivers, as indicated by the error and BER counters. As a comparison an iBERT core running 3.125 Gbps is shown in figure 7.18 where no errors has occurred by the time the screen capture was taken. This shows that the signal integrity problems faced by PCB engineers are a real problem when higher speeds are used. Note that the parameters called pre and post cursor corresponds to the pre and post emphasis settings discussed previously. A seven bit PRBS pattern can also be used to somewhat simulate 8b/10b data encoding.

By changing parameters for drive strength, pre and post-emphasis it is possible to lower the BER considerably on some links, where as it seem impossible to get decent results on other links. In figure 7.19, the BER counter is shown for a 6.25 Gbps link with more optimized transceiver parameters.

The results from figures 7.17 and 7.18 shows that by optimizing the parameters for voltage swing and pre and post emphasis BER is improved for transceiver X0Y14 and X0Y15.

The tool is also great to check simple things such as correctness in cable setup and routing, since the iBERT core is not driven by a user design.

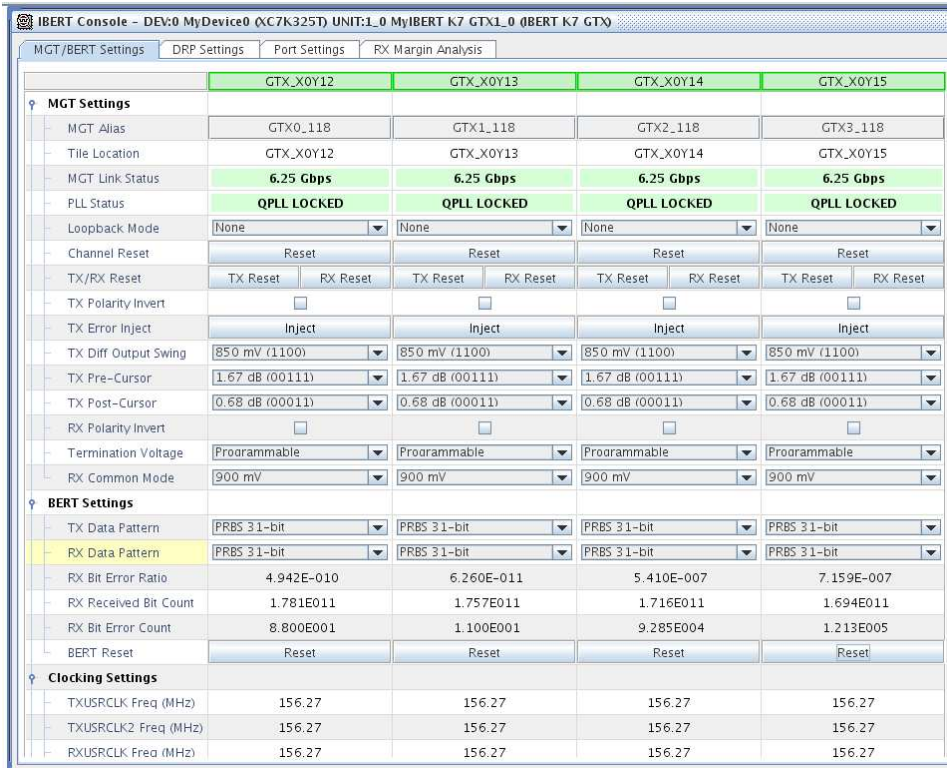

Figure 7.17: *6.25 Gbps iBERT core with default transceiver settings*

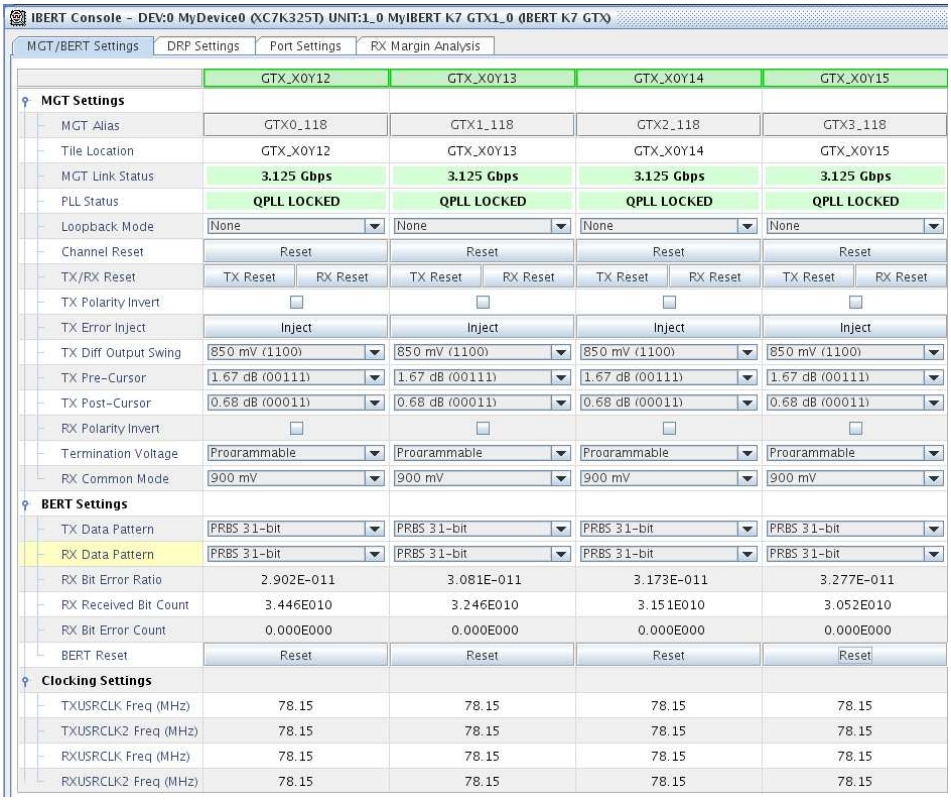

Figure 7.18: *3.125 Gbps iBERT core with default transceiver settings*

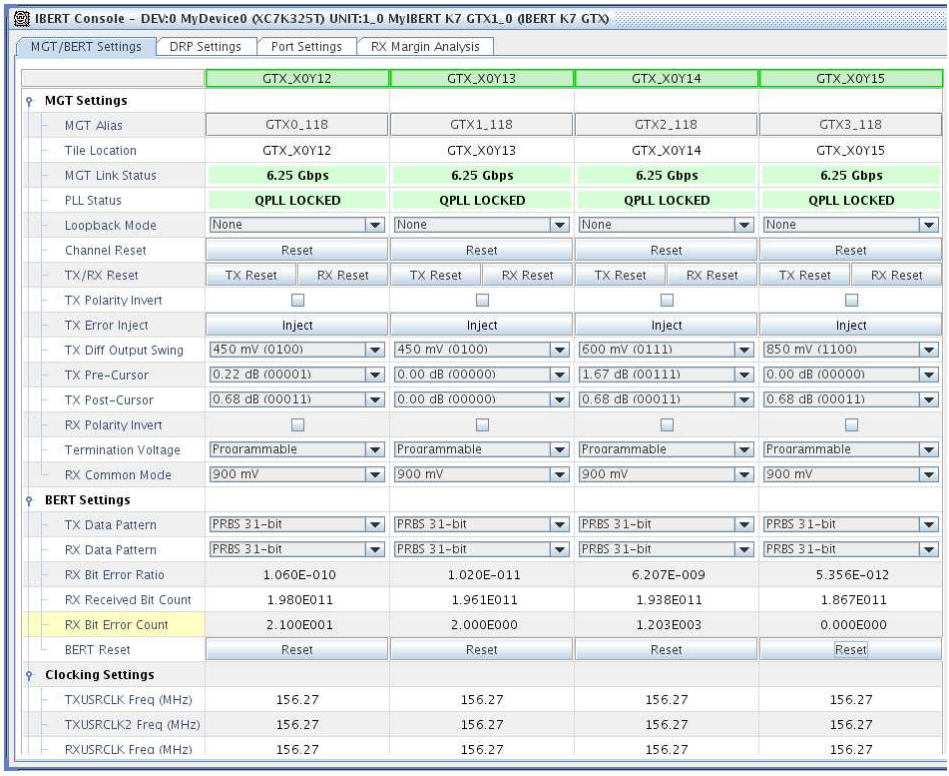

Figure 7.19: *6.25 Gbps iBERT core with optimized transceiver settings*

7.8 PCB

In figure 7.20 one of the total three identical PCBs are shown. In figure 7.21 three "branches" are put together on a bottom plate providing power and interconnects between the three branches. By connecting two branches together using the bottom connectors, it is possible to use eight full duplex MGTs between the two FPGAs Some general purpose I/O and RS232 lines are also routed to the connectors. When using the bottom plate, the branches are connected in a ring setup and for every branch, four MGT are routed to the branch in front of it, and four to the one at the back.

All features and circuitry used on the board have been tested and works, validating the design.

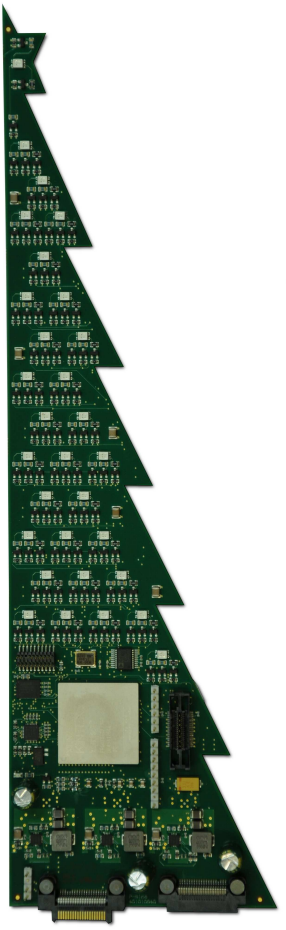

Figure 7.20: *One out of three demonstration* PCBs *produced*

Figure 7.21: *Three branches put together on a power and interconnect platform*

7.9 Analog measurements

Eye diagrams using the iBERT core and the proposed architecture as sources were done using an Agilent DSA90804A digital signal analyzer. The iBERT core cannot generate 8b/10b data, so different PRBS polynomials are used to stress the channel.

7.9.1 KC705 evaluation Board

On the KC705 evaluation board a probe was soldered to the AC-coupling capacitor on the RX side of the FMC extension module. Optimal placement of the probe would have been as close as possible to the RX pins at the FPGA. Now the probe was placed approximately three quarters of the way to the receiver pin.

Generating an eye diagram for a 3.125 Gbps iBERT core, transmitting a 7 bit PRBS pattern using default parameters shows that a clear opening in the eye exists. See fig 7.22. This

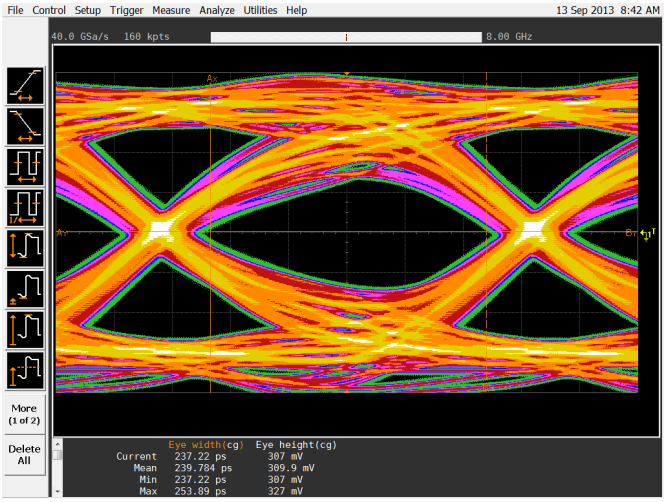

Figure 7.22: *Eye diagram for a 7-bit* PRBS *pattern at 3.125 Gbps*

mirrors previous result where the BER is low when running at this speed. In fig 7.23, an eye diagram is generated for the same pattern, but at double the speed, 6.25 Gbps. The eye is completely closed, suggesting that bit errors most likely will occur. Although, when looking at the BER counter in iBERT, none are recorded. This is most probably due to the equalization filters at the FPGA transceivers used before clock recovery and data acquisition is done. By changing settings for pre emphasis, it is possible to open the eye. See fig 7.24.

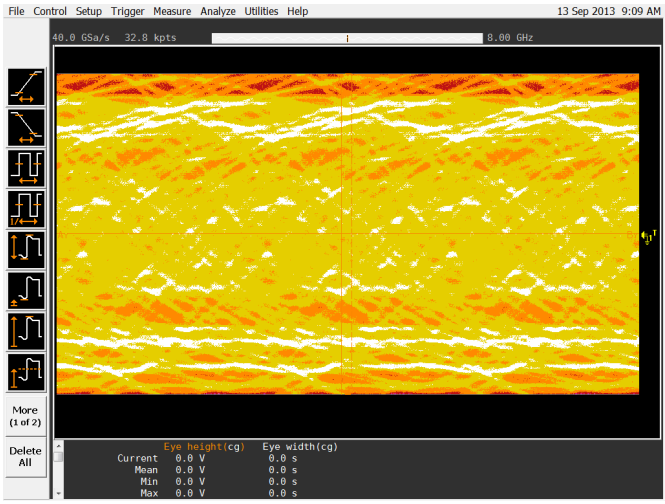

Figure 7.23: *Eye diagram for a 7-bit* PRBS *pattern at 6.25 Gbps. Default transceiver settings*

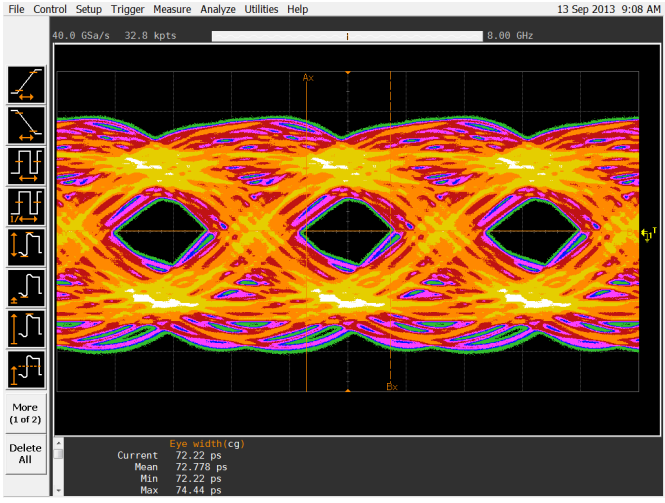

Figure 7.24: *Eye diagram for a 7-bit* PRBS *pattern at 6.25 Gbps. Optimized transceiver settings*

7.9.2 Demo PCB

When running four out of eight MGTs on each FPGA using an identical two lane Aurora implementation at 6.25 Gbps, the complete tree in figure 7.21 uses approximately 1.2 Amperes of current on the 12V power line, which translates to $1.2 * 12 = 14.4W$.

Each branch is loaded with the same logic that is used when evaluating the design on the

KC705 board, but is configured differently in order to use the available transceivers. Each branch sends data to the next in the ring, and receives data from the previous one in the ring. An eye diagram sending a 7-bit PRBS pattern is shown in figure 7.25 with default transceiver parameters. Optimizing the parameters in iBERT the oscilloscope produces the eye diagram in figure 7.26.

The self made PCB shows a much wider eye opening, although one must remember that these signals pass through fewer connects, uses shorter traces and no SMA cables compared to the KC705 board. The probe in this measurement is placed as close to the receiver as possible, on the AC-coupling capacitor very close to the FPGA RX pin.

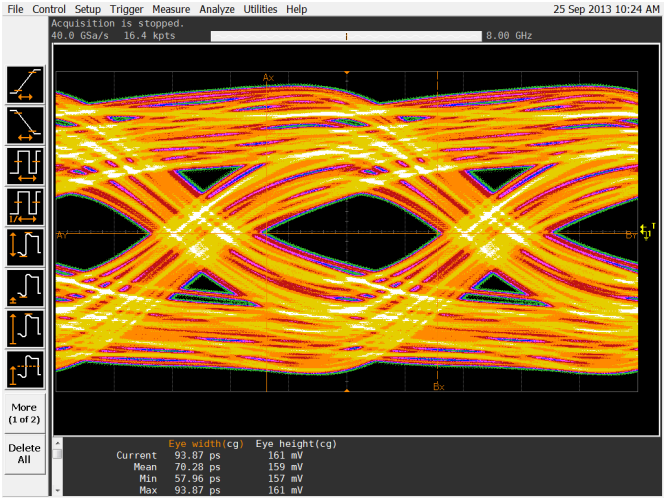

Figure 7.25: *Eye diagram for a 7-bit* PRBS *pattern at 6.25 Gbps using the demo* PCB.*Non optimized transceiver settings*

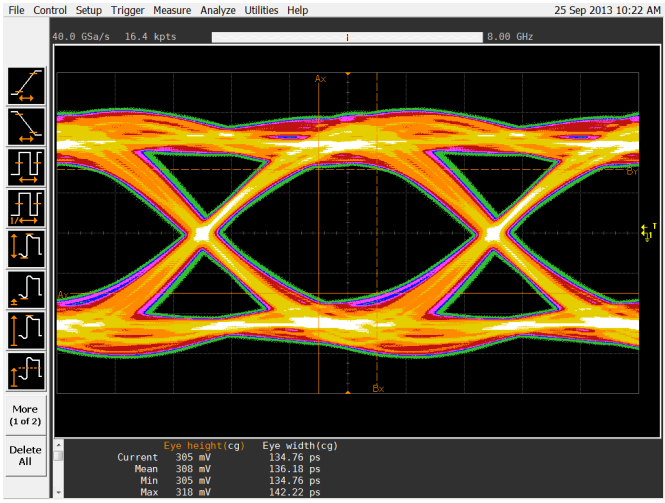

Figure 7.26: *Eye diagram for a 7-bit* PRBS *pattern at 6.25 Gbps using the demo* PCB*. Optimized transceiver settings*

8 Discussion

8.1 Viability of the Aurora 8b/10b IP core

It is the author's firm belief that for point-to-point chip communication, the Aurora core is a most adequate choice. The user protocol is entirely up to the user, thus a minimal overhead protocol can be implemented to maximize throughput. The AXI-4 streaming interface to the Aurora core is lightweight and easy to work with and works as expected. Data put in at the transmitter end is presented to the receiver end after a delay induced by the internal circuitry, and delay in the channel.

The included example modules generated by Coregen is only intended as a demonstration example when using the Aurora core. So in a *real* application, modifications are needed. However, the workload of porting the example design to a custom design was not that high, since many of the included modules could be reused.

The answer to how well the Aurora IP scales and its flexibility are two folded. If the user wants to include several Aurora cores on the same FPGA *and* in the same quad, user modifications are needed. Take for example if the designer needs four single lane Aurora implementations and has to run them all in the same quad. If a high lane rate is needed then the core has to use the QPLL. However, there is only one QPLL in each quad and all four Aurora cores will try to instantiate one QPLL each. This will never pass the MAP stage in the bit file generation process, where instances of primitives are mapped to real hardware on the FPGA. In the project, this was solved by having one master and one slave transmitter/receiver Aurora interface. One who instantiated the QPLL and slaves where these entities were removed.

If the designer is not constrained by this, the scaling and flexibility is good and the MSM-CAI resource and clock frequency scales linearly. The effort of generating an Aurora core with two or four lanes instead of one is minimal. The only thing the designer has to worry

about is the increase in bit length of the interface to the core. Although, if the designer needs to change transceiver parameters, such as emphasis, equalization filter taps etc, the designer needs to change these "manually".

When generating an Aurora 8b/10b IP the tool generates files that are used to interface the transceivers. In these files, all of the transceiver settings are set, but there is no way of controlling these settings when generating the Aurora IP. To change these parameters, the designer has to manually edit the generated files, or generate new files using another IP core in Coregen and copying over the old ones. It would have been a nice feature to customize these settings in the Aurora IP generation as well in order to avoid unnecessary problems when copying the new files, which have different entity names.

Bit errors are caught by the CRC checker and in the final on-target simulations, no erroneously received frames have been passed through the CRC checker without asserting the appropriate CRC signals. The reader is urged to read the section about bugs in the included CRC checker though. See section 8.5.

Looking at the area footprint of an Aurora core, it scales well with increased amount of lanes, showcasing a linear increase in resources needed. In fig 8.1 a table from the Aurora data sheet is shown, showing the resources needed for the Aurora core alone.

Virtex-7/Virtex-7 Lower Power/ Kintex-7/Kintex-7 Lower Power Family			Streaming		
Lanes	Lane Width	Resource Type	Duplex	Simplex	
			Full-Duplex	TX Only	RX Only
1	4	FFs	308	158	180
		LUTs	270	126	119
2	4	FFs	543	211	365
		LUTs	492	191	272
4	4	FFs	940	294	665
		LUTs	880	307	493
8	4	FFs	1734	452	1265
		LUTs	1593	542	902
16	4	FFs	3325	780	2465
		LUTs	3144	979	1723

Figure 8.1: *Aurora IP, resources occupied [Xilinx Inc, 2013c]*

8.2 PCB technology limits

In this project, a 6.25Gbps lane rate was used as the max rate. Running 6.25 Gbps through almost 20 inches of cable and four connectors on the KC705 board, the channel works. A PER of 20-2000 packet errors per second is not insignificant, but the channel still operates. The reduction in throughput due to these errors is in the order of 1 to 6 Mbps, approximately 0.063% of the total bandwidth. The PCB technology used in the KC705 and the FMC extension module is unknown, so the viability of FR-4 at 6.25 Gbps cannot be confirmed on this board.

8.3 Future Work

Since the design met timing constraints early on, probably due to the new Kintex 7 FPGA, no real optimization effort with respect to speed has been done. Running the design on a slower FPGA might require more pipeline stages and might also require a more advanced arbiter to reach a high enough clock frequency.

A test with a real image sensor should be done. In order to test the synchronization FIFOs halt signal in a more realistic manner.

The design could also be implemented to support partial words. Instead of having a 32-bit word as the smallest data size, a design which could also handle individual bytes could be created, since there is support in the Aurora core.

Concerning the protocol, the one presented is more of a proof of concept. If the design requires another field in the header, to transmit for example control data between devices, it is easily expanded to support this kind of feature. The 32-bit header has extra room in it as it is, but it could also be extended to several 32 words if needed. The simplification of assuming source and destination address might be an over simplification and could require a remake.

The current error detection and recovery architecture has a flaw. In order to handle lost ACKs, the receiver compares the currently received CRC with the one previously received. But if two valid data sets are received that has generated the same CRC checksum, the later data set would be discarded. This could be solved by introducing a check on the ack_counter inside the header as well.

The framer module, responsible for the AXI4 streaming protocol and the transmit buffers, turned out quite messy. This module could use a complete overhaul, since the current implementation is unnecessarily complicated. When designing RTL code, it is very important to keep things as simple as possible.

In the problem description an EOF signal from the source is assumed. This signal is not guaranteed to arrive due to how it is generated. This may create a situation where two SOFs are sent without an EOF in between. This requirement is not covered in design and a future user of the interface needs to introduce additional logic in order to handle this situation.

8.4 An alternative architecture

An alternative architecture that was considered in the early phases was to multiplex data from several sources into a large word when using more than one Aurora Lane. This would keep the area requirements much lower since every framer could work with a 32-bit data interface. A controller unit could then create a larger data word using the outputs from the 32-bit frames. This idea was not followed through because of the fear of complexity of the controller unit and the low utilization ratio for a use case when few sources are connected to design.

8.5 Aurora 8b10b v8.3 CRC checking circuitry

Early during the design phase when the error detection and recovery feature was to be implemented, the CRC interface from the Aurora core did not work as intended. When using more than one lane, the CRC_PASS_FAIL_N signal was never asserted. After a couple of days debugging the issue I noticed that the included CRC circuitry was faulty. The bug is located in the files generated from Coregen in the directory name_of_core/src/name_of_core_txcrc.vhd and name_of_core/src/name_of_core_rxcrc.vhd.

In short, the TKEEP signal equivalent used when calculating the CRC for more than two lanes and the concatenation of calculated partial CRCs are erroneously connected when using more than one lane. This was worked around by switching inputs in the previously stated files and a Webcase has been filed at Xilinx web page. As of today no corrections has been done to the Aurora Core by Xilinx but the error has been reproduced and accepted by my Xilinx contact. This error might impact the robustness of the system, since the testing I have done on the CRC circuitry is limited at best.

The Webcase was filed 28th of June and there is also a forum post concerning the issue found at http://forums.xilinx.com/t5/Connectivity/Aurora-8b10b-v8-3-CRC-issue/td-p/317869.

9

Conclusions

An evaluation environment, a generic interface to the Aurora 8b/10b IP and a custom, low overhead protocol has been designed and evaluated in terms of protocol efficiency, clock frequency on target and resource usage. A demonstration/reference PCB has been designed in order to introduce the Kintex-7 FPGA to SAAB Dynamics. The viability of routing high-speed signals on FR4 PCB technology has been investigated.

With a few customizations, the Aurora IP seems like a good candidate to use instead of more complex and expensive serial protocols. The only drawback is the highly simplified user interface when generating the IP core. Many of the transceiver settings available are not present here. They have to be configured using another Coregen wizard or manually changed in the generated VHDL files. Finding bugs in the CRC circuitry makes one wonder though about the reliability of the IP.

Writing VHDL code using generics is challenging and limited in use, but powerful when it can be used.

Today, the speed of serial communications is not limited by user design, but is almost purely a problem in the analog domain. Running analog simulations on the transmission lines and connectors used have not been done during this thesis but it should be considered when designing with high speed devices. If a simulation of the system does not work, it is highly unlikely that the real hardware will work. It is then a matter of what is cheapest: Create a PCB, test it and iterate until it works, or use an expensive analog simulator and simulate the entire chain of cables, connectors and traces before doing the PCB layout.

In whole, the proposed architecture for a multi-source, multi-lane Aurora interface feels complete. A real world test with real camera sensor would have been interesting though.

Bibliography

Abhijit Athavale and Carl Christensen. *High-Speed Serial I/O Made Simple*. Xilinx, Inc., 1.0 edition, April 2005. http://www.xilinx.com/publications/archives/books/serialio.pdf. Cited on pages 9, 12, and 13.

Ying-Wen Bai and Chang-Chih Liu. The performance improvement of a photo card reader by the use of a high-integration chip solution with double fifo buffers. *IEEE Transactions onc onsumer Electronics (Volume:51 , Issue: 2)*, pages 329–334, May 2005. Cited on page 25.

Mohamad Baseri and Seyed Ahmad Motamedi. Simulation study of packet length for improving throughput of ieee 802.15.4 for image transmission in wsns. *2013 International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE)*, pages 6–9, 2013. Cited on page 55.

Peter A. Franaszek and Albert X. Widmer. A dc-balanced, partitioned-block, 8b/10b transmission code. *IBM Journal of Research and Development*, pages 440–451, 1983. Cited on page 13.

Benjamin Krill. Round robin arbiter (vhdl). http://un.codiert.org/2009/04/round-robin-arbiter-vhdl/, 2009. Cited on pages 26, 29, and 30.

Maxim Integrated. An introduction to preemphasis and equalization in maxim gmsl serdes devices, November 2011. www.maximintegrated.com/app-notes/index.mvp/id/5045. Cited on pages 12 and 13.

OpenCores. Wishbone system-on-chip (soc)interconnection architecture for portable ip cores, 2010. http://cdn.opencores.org/downloads/wbspec_b4.pdf. Cited on page 26.

Jian Qi, S. Aissa, and Xinsheng Zhao. Optimal frame length for keeping normalized goodput with lowest requirement on ber. *Innovations in Information Technology*, pages 715–719, 2007. Cited on page 24.

Xilinx Inc. *Aurora 8B/10B Protocol Specification*, April 2010. http://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b_protocol_spec_sp002.pdf. Cited on page 14.

Xilinx Inc. *LogiCORE IP Aurora 8B/10B v8.1 User Guide*, April 2012. `http://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b/v8_1/aurora_8b10b_ug766.pdf`. Cited on pages 14, 16, 19, and 20.

Xilinx Inc. *7 Series FPGAs GTX/GTH Transceivers User Guide*, April 2013b. `http://www.xilinx.com/support/documentation/user_guides/ug476_7Series_Transceivers.pdf`. Cited on pages 10 and 11.

Xilinx Inc. *LogiCORE IP Aurora 8B/10B v9.0 Product Guide for Vivado Design Suite*, March 2013c. `http://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b/v8_3/pg046-aurora-8b10b.pdf`. Cited on page 70.

Si Qing Zheng and Mei Yang. Algorithm-hardware codesign of fast parallel round-robin arbiters. *IEEE Transactions on Parallel and Distributed Systems (Volume: 18 , Issue: 1)*, pages 84–95, 2007. Cited on page 26.

S.Q. Zheng, Mei Yang, J. Blanton, P. Golla, and D. Verchere. A simple and fast parallel round-robin arbiter for high-speed switch control and scheduling. *The 2002 45th Midwest Symposium on Circuits and Systems, 2002. MWSCAS-2002.*, pages 671–674, 2002. Cited on page 26.