Department of Physics and Technology
University of Bergen

Master Thesis

Interface Design for the Gigabit Transceiver Common Readout Unit

Anders Østevik

June 2016

Abstract

The future upgrade of the Large Hadron Collider accelerator, the High-Luminosity LHC, has its goal of increasing the beam luminosity by ten times. This will lead to a corresponding growth of the amount of data to be treated by the data acquisition systems, and an increase in radiation. The GigaBit Transceiver ASICs and transmission protocol was developed to provide a high radiation tolerant, high speed, optical transmission line capable of simultaneous transfer of readout data, timing and trigger signals in addition to slow control and monitoring data. The GBT system can be separated into two parts: the on-detector part (GBT custom made ASICs) and the off-detector part (Common Readout Unit).

The primary objective of this thesis has been to design a control interface software for the CRU, along with the design of a PCB that provides physical connection between the CRU and the GBT ASICs. A hardware module was written for the control interface to allow for communication between the software and the CRU. The communication involves a UART and the RS-232 protocol, and the CRU is connected to the PC using an RS232 adapter cable with voltage conversion. The control interface was written in the C-language and is cross-platform compatible. The PCB connects to the CRU using a HSMC contact, and has 10 HDMIs and a SFP-module which allows for connection with the on-detector GBTx ASIC using e-link or fiber-optical connection. The software is not completed, but is well on its way to become usable. The PCB is completed, but some testing remains. In addition, a full system test remains involving the software and PCB together with the GBT ASICs.

Acknowledgement

The work of this thesis has been carried out within the Microelectronics Group at the University of Bergen between August 2015 and June 2016.

First and foremost, I would like to thank my supervisor, Kjetil Ullaland, for guidance and support throughout the work of this thesis.

A thanks goes to Ph. D. student Arild Velure for invaluable documents and the discussions we had regarding the design of the PCB, thoughts and ideas regarding VHDL design and for helping me out understanding SignalTap II.

A special thanks goes to Attiq Ur Rehman, for helping me out with a strategy on how to approach the thesis, and were to begin my work. Also thanks for helping me out with my experiments and for being overall positive to me and my work, and for keeping me motivated through the last semester.

I would also like to thank my friends and fellow students at room 312 for two good years of interesting discussions and shared experiences, with and without academic content.

Bergen, June 2016

Anders Østevik

Acronyms

ASIC	Application Specific Integrated Circuit.
BNC	Bayonet Neill–Concelman.
CDR	Clock & Data Recovery.
CERN	European Organization for Nuclear Research.
CLB	Configurable Logic Block.
CML	Current-Mode Logic.
COTS	Commercial Off-The-Shelf.
CRU	Common Readout Unit.
DCE	Data Communication Equipment.
DFF	D Flip-Flop.
DTE	Data Terminal Equipment.
EMI	Electromagnetic Interference.
FPGA	Field-Programmable Gate Array.
GBT	GigaBit Transceiver.
GBT-SCA	GBT - Slow Control Adapter.
GPIO	General Purpose In/Out.
GUI	Guided User Interface.
HDL	Hardware Description Language.
HDMI	High-Definition Multimedia Interface.
HL-LHC	High-Luminosity LHC.

HPS	Hard Processor System.
HSMC	High-Speed Mezzanine Card.
I2C	Inter-Integrated Circuit.
IC	Integrated Circuit.
IDE	Integrated Development Environment.
ISSP	In-System Source And Probe Editor.
LED	Light-Emitting Diode.
LHC	Large Hadron Collider.
LSB	Least Significant Bit.
LUT	LookUp Table.
LVDS	Low-Voltage Differential Signaling.
MGT	Multi-Gigabit Transceiver.
MSB	Most Significant Bit.
PC	Personal Computer.
PCB	Printed Circuit Board.
PCI	Peripheral Component Interconnect.
PLL	Phase-Locked Loop.
SATA	Serial Advanced Technology Attachment.
SDI	Serial Digital Interface.
SFP	Small Form-Factor Pluggable HSMC board.
SLVS	Scalable Low-Voltage Signaling.
UART	Universal Asynchronous Receiver/Transmitter.
USB	Universal Serial Bus.
VHDL	Very High Speed Integrated Circuit Hardware Description Language.
VLDB	Versatile Link Demo Board.
XCVR	Transceiver.

Contents

Chapter 1

Introduction

1.1 Large Hadron Collider

The Large Hadron Collider (LHC) is the largest and most powerful particle collider in the world. It was built by the European Organization for Nuclear Research (CERN) [1] between 1998 and 2008, and had its first start-up on 10 September 2008. This massive machine lies 175 metres under ground, beneath the France–Switzerland border near Geneva, Switzerland; and consists of a 27 km long circular tunnel of superconductor magnets with additional accelerating structures. Its basic operation is to accelerate particles (protons and heavy ions, e.g. Pb) in bunches[1] to near the speed of light in opposite directions and collide them. The particle bunches are accelerated along two parallel beam lines running through the superconductor magnets, and collided at four locations were experiments like; ATLAS[2], ALICE[3], CMS[4] and LHCb[5] take place. As of 20 May 2015, the LHC can achieve beam energies up to 13 TeV, or 6.5 TeV per beam. Figure 1.1 shows an overall view of the LHC experiments.

1.2 High-Luminosity LHC

The future upgrade of the LHC accelerator, the High-Luminosity LHC (HL-LHC), has its goal of increasing the beam luminosity by ten times. This will lead to a corresponding growth of the amount of data to be treated by the data acquisition systems, and an increase in radiation. This will thus require high rate data links and Application Specific Integrated Circuits (ASICs) capable of tolerating high doses of radiation.

To address these needs, the GigaBit Transceiver (GBT) ASICs and transmission protocol was developed to provide a high radiation tolerant, high speed, optical transmission line capable of simultaneous transfer of readout data, timing and trigger signals in addition to slow control and monitoring data.

[1]The particles are accelerated in bunches to increase the probability that a collision will occur.

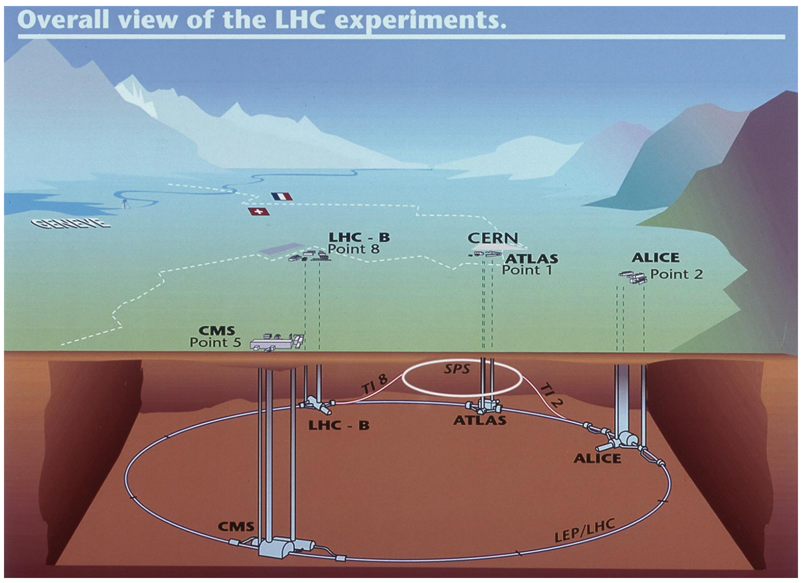


Figure 1.1: LHC overview [6].

1.3 The Gigabit Transceiver system

As illustrated in figure 1.2, the GBT system can be separated into two parts: The on-detector part, and the off-detector part of the system. The below sections gives a brief description of these:

1.3.1 On-detector

The on-detector part consists of radiation hard GBT ASICs that will provide interface and data transmission to the detectors and will thus be located in the radiation zone. These ASICs are used to implement bi-directional multipurpose 4.8 Gbit/s optical links for the high-energy physics experiments.

GBTx

The GBTx ASIC is a serializer-de-serializer chip responsible for the high speed bi-directional optical link. It has a bandwidth of $3.2 - 4.8$ Gbit/s, and combines three data paths for Trigger and Timing Control (TTC), Data Acquisition (DAQ) and Slow Control (SC) information in one physical link; using two optical fibers. The GBTx encodes and decodes this information into what is known as the GBT-Frame, and provides interface to the front end electronics embedded in the detectors. [8]

GBT frame formats

The GBTx transmits a 120-bit frame every 25 ns (40 MHz), which is triggered by LHC particle bunch crossings. The GBTx supports three different encoding modes:

"GBT-Frame", "8B/10B" and "Wide-Bus" mode. Figure 1.3 illustrates the "GBT-Frame" mode. The frame is divided into four parts: Header (H), Slow Control (SC), User Data (D) and Forward Error Correction (FEC). The Header field is a 4-bit field transmitted at the beginning of each frame, used to synchronize the data stream at the frame level. The header field can be set to "idle" (0110) or "data" (0101). The Slow Control field is a 4-bit field dedicated for routine and control operations that do not require precise timing. The User Data field is a 80-bit field reserved for generic transmission of data with a corresponding bandwidth of 3.2 Gbit/s. The remaining field reserves 32 bits for Forward Error Correction. This involves using Reed-Solomon encoding capable of correcting up to 16 concecutive corrupted bits. To achieve DC-balancing, a self-synchronizing scrambler distributes the 0's and 1's in the data stream.

The "8B/10B" and "Wide-Bus" mode share simularities with the "GBT-Frame" mode, but favors data width over reduced error correction (the "Wide-Bus" has none). Both modes are only available in the transmitter part of the GBTx, but requires less resources from the FPGA (1.3.2) than the "GBT-Frame". The "8B/10B" does not require scrambling because it is in itself DC-balanced [8]. These two modes are not yet available in the GBT FPGA example design. Both GBTx encoding and decoding operations can be done within a single clock cycle at 40 MHz.

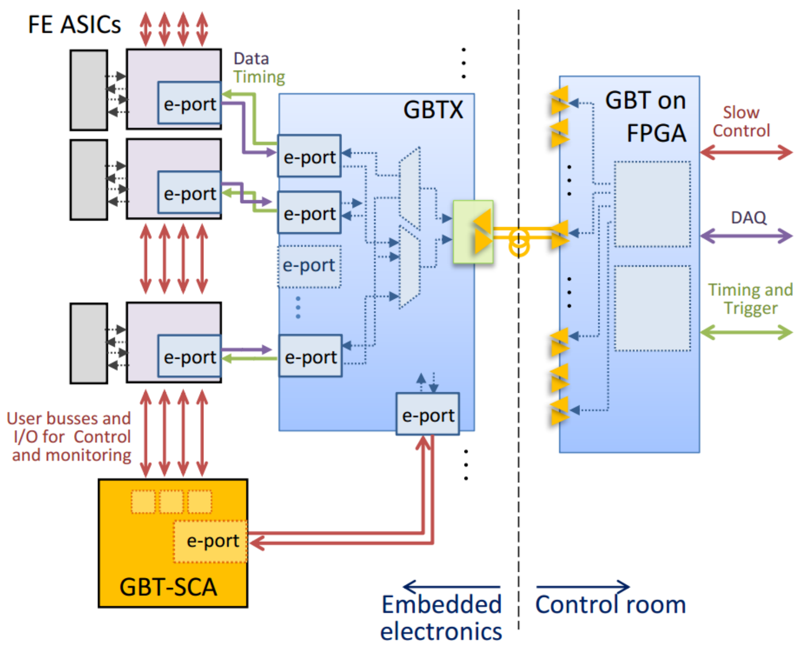


Figure 1.2: The GBT-link in its entirety. The on-detector (Embedded electronics) consists of custom made ASICs with an optical link connecting the off-detector (control room) Field-Programmable Gate Array (FPGA) with the GBT-FPGA implemented. [7, Figure 1].

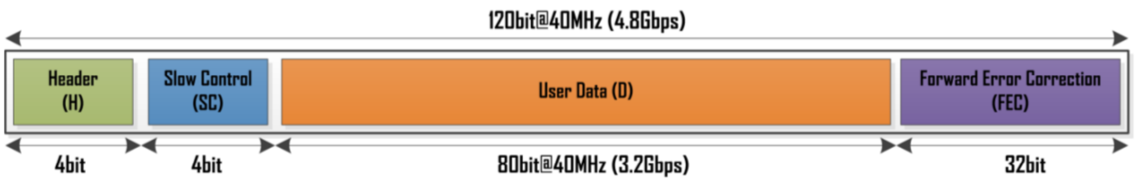


Figure 1.3: The GBT-Frame format. [9, Figure 4].

GBT-SCA

The GBT - Slow Control Adapter (GBT-SCA) ASIC is the part of the GBT chipset which distributes control and monitoring signals to the front-end electronics embedded in the detectors. It connects to the GBTx through a dedicated 80 Mbit/s e-link (1.5) and provides a number of user interface options for the front-end detectors, which includes: SPI, I2C, JTAG and a number of GPIOs [10].

All chips have been implemented using a commercial 130 nm process because of benefits regarding inherent resistance to ionising radiation [11].

1.3.2 Off-detector

The off-detector part is located in the counting room and consists of a Common Readout Unit (CRU), that will provide an interface between the detector ASICs and an online computer farm, with the GBTx as the middle joint. The CRU consists of Commercial Off-The-Shelf (COTS) components, mainly an FPGA, and will through optical links receive the data from the radiation detector.

FPGA - Cyclone V GT

Altera's Cyclone V GT FPGA board was chosen for use in this thesis. It was chosen mainly because of the on-board transceivers that are capable of reaching speeds that surpass the requirements of the GBT-FPGA Multi-Gigabit Transceiver (MGT), i.e 4.8 Gbit/s; "GT" indicates that the FPGA has transceivers that support speeds up to 6 Gbit/s [12].

Originally, a Terasic Cyclone V SX development board was aquired for use with this thesis. The Terasic board has advantages over the Cyclone V GT board in terms of communication with the outside world, such as an on-board Usb-to-Uart interface (more on this in chapter 4). However, it was discovered that the transceivers on the Terasic board were not fast enough for the GBT MGT; maximum supported transceiver speed is only 3.125 Gbit/s [12]. Because of this, the more powerful Cyclone V GT FPGA development board was ordered, replacing the Terasic.

GBT-FPGA project

The GBT-FPGA project provides a firmware library for Altera and XilinX FPGAs for communication with the GBTx chipset. It allows for one or several GBT links of type "Standard" or "Latency-Optimized" (The latter providing low, fixed and deterministic latency). Each GBT link is composed of three components: a GBT Rx, a GBT Tx and an MGT. The GBT Rx is responsible for receiving, decoding and de-scrambling the data from the MGT. The GBT Tx is responsible for scrambling and encoding data before transmitting it through the MGT. The MGT is responsible for the actual transmitting, receiving, serialization and de-serialization of the GBT data. It is divided into a transmitter and a receiver: The transmitter shifts in 40 bit words from the GBT Tx, serializes the data and sends it out with the help of a dedicated PLL that generates a serial clock of 2400MHz. The receiver de-serializes the incoming data before shifting it to the GBT Rx. The receiver contains a Clock & Data Recovery (CDR) block to recover the clock signal directly from the incoming data stream [2]. The MGT requires an external clock of 120 MHz [9].
The GBT link supports all three encodings described in section 1.3.1.

GBT-example design

The firmware library comes with an example design that incorporates a single GBT link of the "Standard" type. Included in the example is a pattern generator, connected to the GBT Tx; a pattern checker, connected to the GBT Rx; and a user control interface implemented using the Quartus-bound In-System Source And Probe Editor (ISSP). The example enables for external and internal loopback testing.

1.4 Versatile Link Demo Board

The Versatile Link Demo Board (VLDB), shown in figure 1.4, is the evaluation kit for the radiation hard optical link. It includes the main elements of the GBT Link, the radiation hard ASICs; GBTx, GBT-SCA and VTRx/VTTx (optical-link modules), in addition to radiation hard DC/DC converters. The VLDB has 20 e-links reachable through HDMI-connectors that connects to the front-end electronics, and a fiber-optical link that connects to the off-detector FPGA.

1.5 E-Links

E-links are electrical local duplex serial links suitable for transmission over PCBs or cables, within a distance of a few meters, and can operate at any data rate up

[2]To be able to recover the clock using a CDR, it is important that the serial data has even transitions of 1's and 0's. This is one of the reasons why scrambling the data is important.

to 320 Mbit/s [3]. It was designed with the GBTx in mind, having radiation hard and single event upset (SEU) resistant transmitter and receiver blocks. The e-links supports both SLVS and LVDS electrical standards [14]. Each E-link consists of three differential signal lines: a clock line (dClk+/dClk-), a downlink output (dOut+/dOut-) and a uplink input (dIn+/dIn-).

The GBTx arranges e-links in 5 groups, with up to 8 e-links per group corresponding to 16 bits in the uplink and the downlink frames; making a total of 80 bits [8]. Each group can be programmed to data rates of 80 Mbit/s, with 8 e-links per group; 160 Mbit/s, with 4 e-links per group; or 320 Mbit/s, with 2 e-links per group. Faster data rates comes with the expense of less physical e-link connections available for the front-end detectors.

1.6 Primary objective

The primary objective of this thesis has been to design a CRU control interface software, along with the design of a Printed Circuit Board (PCB) that provides physical connection between the CRU (FPGA) and the VLDB card. The control interface was developed with the goal of one day replacing the Quartus-bound ISSP, which is used today to manipulate the GBT control-signals; and instead introduce a cross-platform, open-source solution. The software is not completed, but well on its way to become usable. The PCB is completed, but some testing remains along with a full system test with the GBTx involved.

[3]The GBTx can only handle 320/160/80 Mbit/s

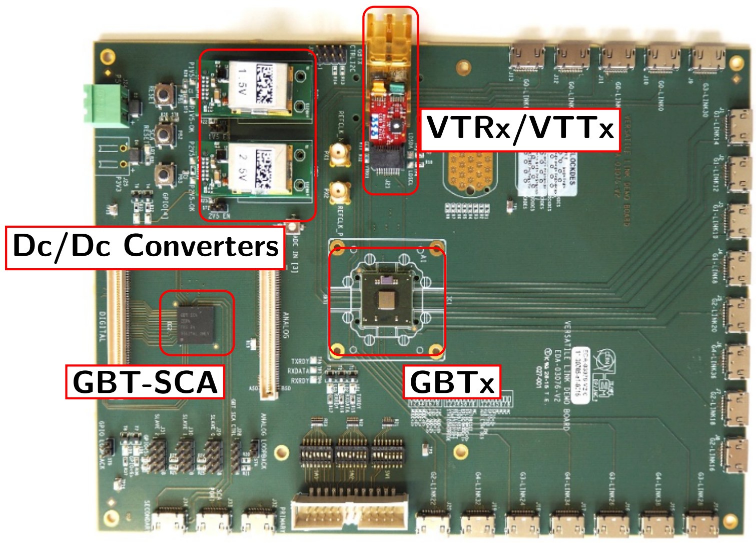


Figure 1.4: Versatile Link Demo Board overview [13].

1.7 Outline

This thesis is devided into six chapters, including this one. Chapter 2 gives a brief description of the transceiver technologies that enables communication between the GBTx and the CRU. Chapter 3 gives a brief overview over the process of designing a PCB that connects the FPGA and the VLDB togheter using e-links and optical link. Chapter 4 gives a brief overview of the development of the PC to CRU software design, both on the software and hardware side. Chapter 5 presents the different tests performed on the developed PCB, software and hardware. Finally, chapter 6 summarizes and concludes the thesis.

Chapter 2

Cyclone V Transceiver Technology

For the FPGA to be able to transmit and receive serial data in the gigahertz domain, a high-speed transceiver is required. The Cyclone V GT-series supports a number of transceiver technologies through the High-Speed Mezzanine Card (HSMC) physical interface that can reach speeds up to 5.0 Gbit/s. This section gives a general description of some of these protocols.

2.1 Differential Signals

Common for all protocols described here is the fact that the signals are treated differentially. While a single ended signal involves one conductor between the transmitter and receiver, with the signal swinging from a given voltage to ground; differential signals involve a conductor pair of two signals that are identical, but with opposite polarity. The pair would ideally have equal path lenghts in order to have zero return currents, avoiding problems like Electromagnetic Interference (EMI). In addition, placing the signals as close as possible to one another will give benefits in terms of common-mode noise rejection [15].

When implemented correctly, differential signals have advantages over single ended signals such as effective isolation from power systems, minimized crosstalk and noise immunity through common-mode noise rejection. It also improves S/N ratio and effectively doubles the signal level at the output ($+v-(-v)=2v$), which makes it especially useful in low level signal applications. The disadvantage comes in an increase in pin count and space required, since differential signals consists of two wires instead of one [15].

2.2 Low-Voltage Differential Signaling

Low-Voltage Differential Signaling (LVDS) is said to be the most commonly used differential interface. The interface offers a low power consumption with a voltage swing of 350 mV and good noise immunity. With the right conditions, the standard can be able to deliver data rates up to 3.125 Gbit/s [16].

The Cyclone V GT board has 17 LVDS channels available on the HSMC port A connector. The channels have the ability to transmit and receive data at a rate up to 840 Mbit/s, with support for serialization and de-serialization through internal logic [12].

2.3 Current-Mode Logic

For data rates that exceeds 3.125 Gbit/s, Current-Mode Logic (CML) signaling is preferred. This is due to the fact that certain communication standards such as PCI-express, SATA and HDMI, shares consistency with CML in signal amplitude and reference to Vcc. CML can reach a data rate in excess of 10 Gbit/s, but has a higher power consumption than LVDS, with a voltage swing of approximately 800 mV [16].

The Cyclone V GT board has 4 Pseudo-CML (PCML) channels available on both port A and B HSMC connectors. The channels have the ability to transmit and receive data at a rate up to 5.0 Gbit/s, just over the 4.8 Gbit/s range required by the GBT MGT [17].

Chapter 3

HSMC-to-VLDB PCB Design

In order to test the GBTx chip, a PCB that acts as a connecting bridge between the FPGA and the VLDB is needed. As mentioned in section 1.4, the VLDB has a optical link which will provide connection to the FPGA in the counting room, in addition to twenty e-links reachable through physical HDMI contacts, which will be available connections to the front-end detectors.

A basic test example is to transmit test data from the FPGA to the GBTx via e-links. The data will represent data sent from the front-end detectors. The GBTx will then transmit the same data back to the FPGA via the fiber-optic cable, making it possible to analyze the received data and also compare it to the transmitted data.

Because of limited available physical LVDS connections on the FPGA board, the 320 Mbit/s data rate was chosen for the e-link connections. The required LVDS connections would thus be 20 receiver pairs (for input and clock pairs) and 10 transmitter pairs (for output pairs). As mentioned in section 2.2, the HSMC port A connection on the board has only 17 available LVDS channels, or 17 transmitter and 17 receiver pairs. Since the clock on each e-link will be the same during testing, the 10 clock pairs can be reduced to just one, reducing the total receiver pairs needed on the FPGA from 20 to just 11 pairs.

The resulting PCB has 10 individual High-Definition Multimedia Interface (HDMI) connector with each having a receiver and a transmitter pair. The J4 HDMI contact contains an additional receiver reserved for an input clock from the VLDB. Figure 3.1 illustrates the connection between the FPGA and the VLDB using the PCB as a connection board between the two.

This chapter explains the design process of this PCB; beginning with the discussion of different design approaches, to the theory behind designing a high speed PCB. It then explains the final PCB design parameters and discusses the resulting product.

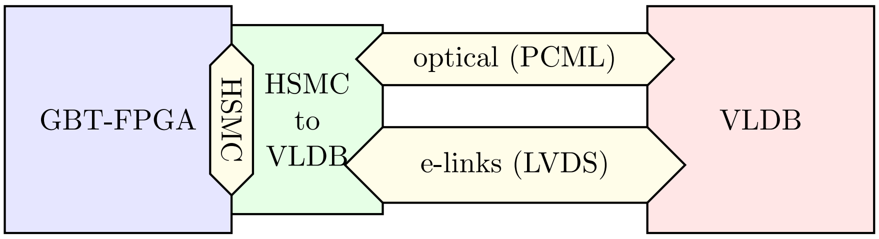


Figure 3.1: Block diagram showing the basic connection between the FPGA and VLDB using the HSMC-to-VLDB PCB as a connection board.

3.1 Design Discussion

The design discussion and planning of the PCB was done together with Ph. D. student Arild Velure.
The initial plan was to design a PCB with a male Small Form-Factor Pluggable HSMC board (SFP)-contact in one end connected with two HDMI connectors, one on each side of the PCB, making an adapter that can be plugged directly into the SFP-connectors on a SFP-board with HDMI cables connected to the VLDB. This design was quickly scrapped as there was no loose male SFP connectors available on the marked, only female. The design plan was then changed to use only one PCB board with female SFPs-connectors on one side, wired to HDMI-connectors on the other side, acting as a middle joint between the SFP-board and the VLDB with SFP- and HDMI-cables connecting the three boards together.

The end result was a design that could be directly mounted on the FPGA through the HSMC connector. By copying one of the SFP-to-HSMC connections from the original SFP-board design files (the XCVR based SFP connectors, see [18]), we found that we could remove the SFP-board completely from the connection chain and eliminate the resulting need for electrical SFP-cables. This would also remove the middle joint in the chain, reducing complexity and lowering the chance of signal reflections.

3.2 High Speed PCB Design

This section gives a brief explanation of high frequency signal behavior and the compensation methods that was practiced during the design of the HSMC-to-VLDB PCB.

3.2.1 Transmission Lines

When signal rise/fall times becomes comparable with the propagation delay of the conductor, the signal can no longer be assumed to change immediately. The conductor becomes what is known as a transmission line, with the signals voltage and

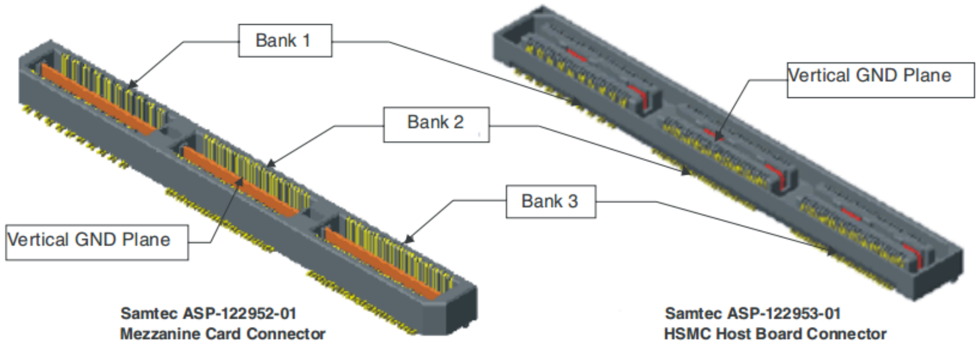


Figure 3.2: Male (ASP-122952) and female (ASP-122953) HSMC-connectors. The male type is to be connected at the bottom of the HSMC-to-VLDB PCB [19, Figure 2-1].

current behaving like waves propagation through the conductor. The rule of thumb for determining if a signal is propagating along a transmission line is when the rise/-fall time of the signal is less than 1/4 of the signal period, so that the high and low states are recognizable [20].
The propagation velocity of a signal is given by:

$$v = \frac{c}{\sqrt{\epsilon_r}} \tag{3.1}$$

, where ϵ_r is the dielectric constant of the dielectric material, FR4, often used as a material to separate the copper layers on the PCBs. $\epsilon_r = 4.05$ @ 5 GHz [21]. A signal thus propagates through the conductor at a velocity of approximately 15 cm/ns [20, example 13.7].

If assuming a constant transmitting frequency of 4.8 Gbit/s (208 ps period), the conductor becomes a transmission line if the length of the conductor stretches longer than 3.1 cm between the transmitter and receiver. This short length is very difficult to avoid when designing a PCB with microstrip traces running from one connector out to eleven other connectors. The traces on the HSMC-to-VLDB PCB are therefore considered as transmission lines.

3.2.2 Reflections and Characteristic Impedance

Since the signals no longer changes immediately, the transmitter is not directly loaded by the receiving end at the time it sends a pulse down the conducting channel, but is instead loaded by the impedance coming from the channel itself. This impedance is called the characteristic impedance, Z_0, of the channel [20].

Since the signals no longer changes immediately, the transmitter can not see what is connected at the receiving end at the time it sends a pulse down the conducting channel. All it sees is the channel impedance, called the characteristic impedance, Z_0, of the channel [20].

The type of trace that is used in the HSMC-to-VLDB PCB design is called a microstrip, which is when the signal traces run along traces on the outer layers of the PCB with the ground plane on the layer underneath.

A microstrip has a characteristic impedance that is given by:

$$Z_0 = \frac{60}{\sqrt{0.475\epsilon_r + 0.67}} \times \ln \frac{4h}{0.67(0.8w + t)} \tag{3.2}$$

, where ϵ_r is the dielectric constant, h is the height of the microstrip seen from the ground plane, i.e the thickness of the FR4 layer, w is the width of the microstrip, and t is the thickness of the copper [20].

It is important to match the impedance of the trace to that of the load impedance at the receiving end, or vice versa. With these being different, the energy of the signal cannot be fully absorbed at the receiving end, resulting in a partly reflection of the signal wave back to the transmitter. The ratio of the wave reflected back is given by:

$$\Gamma = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{3.3}$$

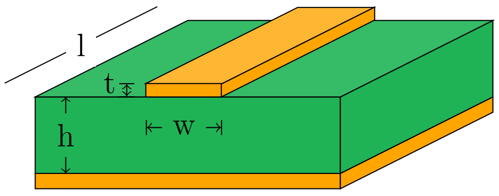
, where Z_L is the load impedance and Z_0 is the characteristic impedance.

Ideally the Z_L and Z_0 should be equal to avoid reflections. Impedance mismatch can cause waste of signal energy and interference with other signal pulses being transmitted through the channel [20].

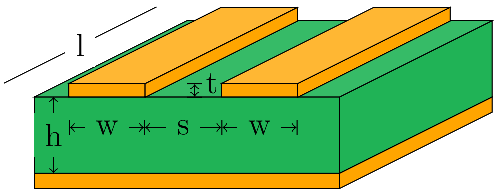
When looking at the datasheet for cables with differential signals, such as HDMI-cables, it is often supplied a differential impedance between the cables instead of a characteristic impedance for each cable. Knowing the characteristic impedance, it is possible to calculate the corresponding differential impedance between two microstrip traces (or vice versa):

$$Z_{diff} = 2 \times Z_0[1 - 0.48e^{(-0.96\times\frac{s}{h})}] \tag{3.4}$$

, where Z_0 is the characteristic impedance of the individual microstrips (assuming they are equal and have the same length), s is the spacing between the microstrips and h is the same as in the equation above [22].



(a) Single-ended.



(b) Differential.

Figure 3.3: Microstrip, cross-section. The microstrip is the copper trace(s) on top, followed by a dielectric layer and a ground plane. h is the thickness of the dielectric, t is the copper thickness, l is the microstrip length, and w is the copper width. s is the spacing between the differential strips.

3.2.3 Routing

As mentioned in section 2.1, differential signals have noise advantages over single ended signals. This is the case only if the pairs have *equal* path lengths and the individual wires in each pair are routed as close to one another as possible. In addition, to keep cross-talk to its minimum, individual pairs has to be routed a distance away from other wires (In reality, this only becomes a problem when dealing with wiring in the micro-scale domain). Twisting the individual wires in each pair to some extent will also contribute to common-mode noise rejection [20]. This was taken into consideration when routing the PCB.

The individual wires in each pair was kept as close as possible, with a space constraint (See table 3.1) according to the required differential impedance Z_{diff} of approximately 100 Ω (See equation 3.6). When routing, twisting the wires (where possible) was achieved by purposely making the wires overlap at the viases. After routing the wires, attempts were made to separate the pairs a distance from one another, to further reduce the cross-talk.

3.3 PCB Design Parameters

The PCB schematic was designed using *Orcad Capture CIS* and the layout using *Orcad PCB Editor*, with design parameters meeting *Elprint's* capabilities [23]. The PCB layout was then exported into *Gerber-files* and imported into *Macaos* for further manufacture specification and validation. The PCB was then ordered from Elprint.

To avoid signal reflections, the PCB needed a characteristic impedance matching that of the cables and termination, in this case a single wire impedance of 50 Ω, and a differential impedance of 100 Ω. This was derived from the fact that the HDMI-cables were specified to have a differential impedance of approximately 100 Ω, and that the individual transceiver lines from the FPGA has a characteristic impedance of 50 Ω, with a selectable termination resistance at the receiving end of 100 Ω, connected between the lines.

The PCB was chosen to be four layers, with signals running on the top and bottom copper layers, and the two middle planes for voltages and ground respectively.

For thicknesses of the different PCB copper layers and the FR4 in between, Elprint has a set of predefined *stack-ups* available in Macaos. It is possible to define custom stack-ups as well, but this will result in a more costly PCB. The 4036 predefined stack-up has a copper thickness of 18 µm on the two outer layers following an FR4 thickness of 65 µm between the outer layers and the power planes, with the power planes having a thickness of 35 µm. Between the power planes is another FR4 layer with a thickness of 1.4mm making the total PCB thickness of a standard 1.6mm. With a mircostrip width of 100 µm, the formula for characteristic impedance yields:

$$Z_0 = \frac{60}{\sqrt{(0.475 \times 4.05) + 0.67}} \ln 3.96 \tag{3.5}$$

, which gives a characteristic impedance Z_0 of 51.3 Ω @ 5 GHz. With a 300 µm spacing between the differential traces, the formula for differential impedance yields:

$$Z_{diff} = 2 \times 51.3\ \Omega[1 - 0.48e^{(-0.96 \times \frac{300}{65})}] \tag{3.6}$$

, which gives a differential impedance Z_{diff} of 102 Ω @ 5 GHz.

Table 3.1 shows data extracted from Orcad PCB Editor, which has a built-in impedance calculator that yields similar results as shown above:

	Layer	Type	t [µm]	ϵ_r	w [µm]	$Z_0[\Omega]$	Spacing [µm]	$Z_{diff}[\Omega]$
1	Surface	Air		1				
2	Top	Conductor	18		100	51.3	300	102
3		Dielectric	65	4.05				
4	Voltage	Plane	35					
5		Dielectric	1400	4.05				
6	Gnd	Plane	35					
7		Dielectric	65	4.05				
8	Bottom	Conductor	18		100	51.3	300	102
9	Surface	Air		1				

Table 3.1: The layers of the PCB and their traits, where t is the layer thickness and w is the width of the trace. The PCB has an overall thickness of 1.6 mm

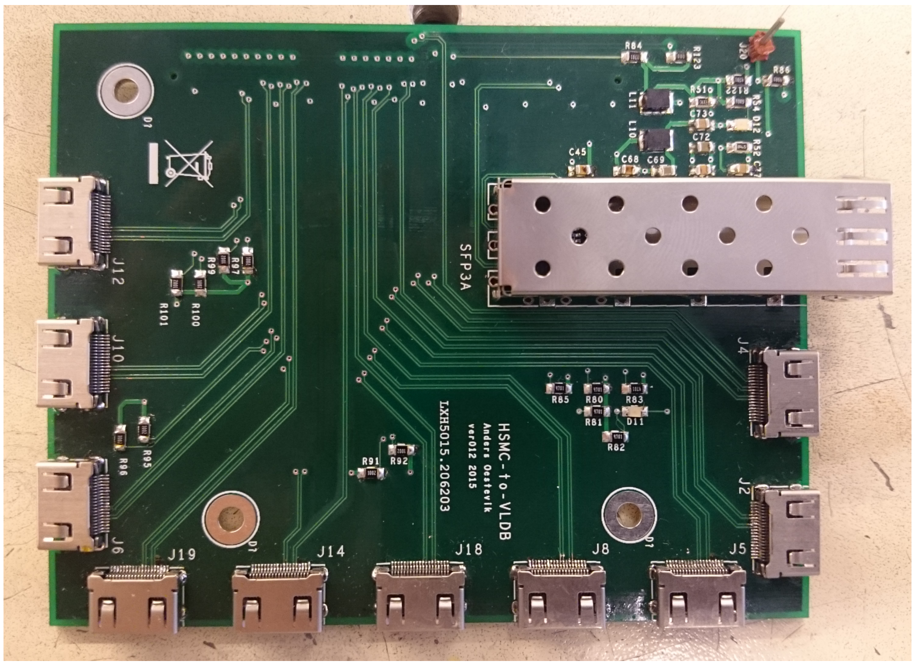


Figure 3.4: HDMI-to-VLDB PCB with components soldered on.

3.4 Pads and Footprints

All pads and footprints were custom made using *Cadence Pad Designer*, with dimensions collected from datasheets of the different components. The only exception was for the HSMC footprint and pads, which were acquired from the manufacturer. The pads were dimensioned a bit longer than the actual pins, making it easier to make contact with the soldering iron when soldering by hand.

3.5 Soldering Process

All the components on the PCB were soldered, with the exception of the ground pads underneath the HSMC, which had to be soldered in hot air using solder paste. See appendix E for more details about the soldering process.

The finished PCB is shown in figure 3.4. The HSMC contact is located at the bottom of the PCB.

3.6 PCB Faults and Compensations

During the process of hand soldering, it was discovered that the HSMC pads were dimensioned for re-flow soldering using a solder oven. This meant that the pads were dimensioned to match exactly that of the pins, making it very difficult to solder by hand. With 160 individual pins to be soldered to the PCB board, this resulted in a time-consuming work with a lot of effort trying to keep the pins from shorting. To ease the soldering process of the HSMC, the pads should be dimensioned a bit longer.

After receiving the PCBs, further inspection revealed that all viases had missing solder mask due to a design flaw. This exposes the metal of the vias to the surface of the PCB and can possibly cause connections between the viases and the pads and/or metallic casings when soldering. A solution to fix this in a future PCB print would obviously be to include a solder mask in the pad file. With the current PCB print, however, a temporary solution would be to apply thin *kapton tape* where most critical.

The exposed viases later showed not to be as critical as first thought, due to the following points:

- The viases that are exposed underneath the HDMI casings are in fact connected to ground, making the possible shorts between the viases and the already ground connected casing not a critical problem.
- The exposed viases underneath the HSMC-connector could potentially connect to the ground pads when applying solder paste to the pads. This was avoided by carefully applying a thin line of solder paste in the middle of the ground pads. This prevented the solder paste to float over to the neighboring viases when melting it in the oven. A possible drawback to this would be bad connection between the ground pads and the HSMC-connector, but a quick check using a multimeter proved that the HSMC was indeed connected to all the ground pads.

In fact, when testing the PCB, the exposure of viases on the transmitter and receiver lines revealed to be quite useful probe points when using an oscilloscope to measure the signal integrity. Due to the lack of extra added probe points, measuring signals at the PCB viases became the most convenient way of directly measuring the signals on the transmitter and receiver lines during signal measurement. A lesson learned is to include easily accessible probe points on lines that are relevant for oscilloscope measurement.

Chapter 4

PC to CRU control interface

In order to connect and control the CRU from a user PC, some sort of serial communication between the user PC and the FPGA is needed. The purpose of the serial link in this thesis is mainly to monitor and manipulate the GBT control signals (see appendix C for an overview). Because of this, the speed requirements of the link is not crucial.
Figure 4.1 illustrates the different blocks that make up the serial interface: The interface on the PC side consists of a terminal-like interface that lets the user type in "requests" that enables the user to read and/or write to the GBT control signals via the link. These requests are essentially reserved byte-codes that are sent out via the transmitter of the PC COM port to the Universal Asynchronous Receiver/-Transmitter (UART) receiver on the FPGA side. Here, the requests are stored in a FIFO-buffer before they get interpreted by a decoder. The decoder can change a value in the GBT control register (write operation), or send out a byte-code containing data (Most Significant Bit (MSB)) and the address of the data-value (remaining 7 bits) to the UART transmitter for further transmission to the PC COM port. The PC interface stores the data-values in arrays that imitate the GBT control registers.

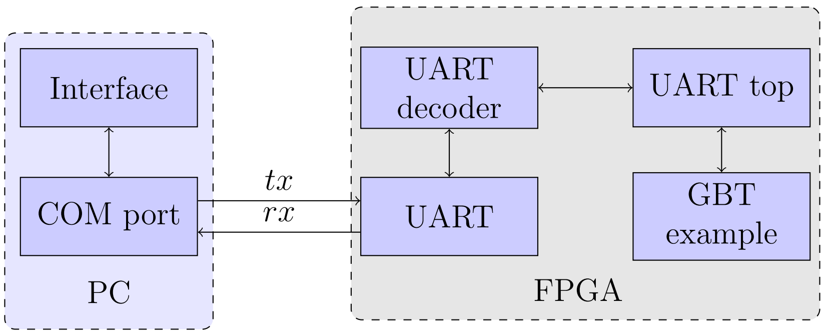


Figure 4.1: Simplified diagram over the PC to CRU serial interface. The VLDB and PCB bridge is not included (see figure 3.1).

4.1 Readily Available Standards

The FPGA board used in this thesis (section 1.3.2) has various forms of communication standards readily available. The following sections starts off with an evaluation of the different communication standards available on the FPGA, and continues with a description of the final implementation of the serial communication software and hardware.

When choosing the serial communication between the FPGA and the PC, factors like physical compatibility, implementation, availability and complexity were taken into consideration. The FPGA board has the following communication capabilities readily available: Peripheral Component Interconnect (PCI)-express, Ethernet, JTAG UART (through the USB Blaster II), and an SDI-transceiver. The following short sections describe advantages and disadvantages of using one of the standards mentioned above in context with the thesis.

PCI-Express

The PCI-express connection requires the FPGA to be directly mounted on the motherboard of the PC, which is in this situation impractical and not necessary for a simple communication link. This option also limits the compatibility with some FPGA boards and PCs that do not have a PCI-express connection available. It does, however, enable very fast data transmission and removes potential noise generated by using external cabling.

Ethernet

The ethernet connection is integrated in most FPGA boards, but requires layers of protocols in order to communicate (no direct serial communication). While it is possible to send serial data over ethernet, it would require a serial-to-ethernet adapter [24]; It would just be easier to send serial data directly from the FPGA-pins using an USB-to-RS232 adapter[1] (see section 4.2). The upside is that an ethernet connection offers long distances between the PC and FPGA, either through networking or long cabling. Using ethernet for communication only requires a known IP-address between the devices for connection and transmission.

SDI-Transceiver

The SDI-transceiver is meant for audio/video transmission and uses BNC-connectors for this task. It therefore requires special audio/video equipment on the PC side for connection and transmission, which is not necessary for a simple communication link.

[1]Adapter to convert between USB and Rs-232 signaling and voltages.

Altera JTAG

Serial communication through the Altera JTAG UART IP is possible through the only USB-connection on the board; the USB Blaster II, which is primary used for programming the FPGA. This requires the implementation of the Nios II soft processor. A soft processor is a microprocessor implemented into the FPGA with the help of logic synthesis only. While it is also possible to use a Hard Processor System (HPS), only a few FPGA-boards (SoC-boards) comes with one implemented. There is no support for HPS on the FPGA board used in this thesis. A Nios II-extension for the Eclipse IDE in combination with the C programming language is commonly used to program the soft processor. Using this approach not only limits the user to send and receive data through a dedicated Altera System Console [25]: the Nios II also occupies a large portion of the FPGA.

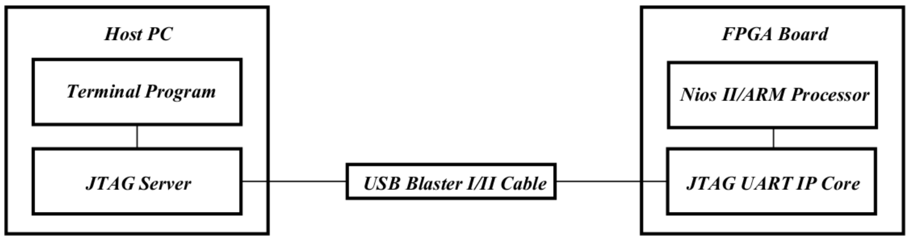


Figure 4.2: JTAG UART communication link between the host PC and the FPGA board [25, Figure 1].

Attempts were made to implement the Nios II and write a simple send- and receive routine in C through the Altera JTAG, but was quickly abandoned due to issues with debugging using the Eclipse Nios II IDE. The most serious debugging issue was the fact that the compiler continued displaying errors after the errors were corrected. Even after the actual bugged function or line of code were deleted completely, the compiler would still complain on the same errors as before it was deleted. This made the dedicated Nios II compiler unreliable, and the code impossible to debug. Not only did this approach make it a whole lot more complicated, but it was also more error prone and permitted the use of user defined software for communication with the FPGA.

4.2 User Defined Communication

In addition to the communication standards described above, which in turn requires specific cable- and socket types for connection, it is possible to connect the transmit and receive signals directly to the FPGA pinout. The only requirement is that the given FPGA has unoccupied transmitter and receiver signal pins available for the user; either through a HSMC-port (with the help of an additional GPIO-extension

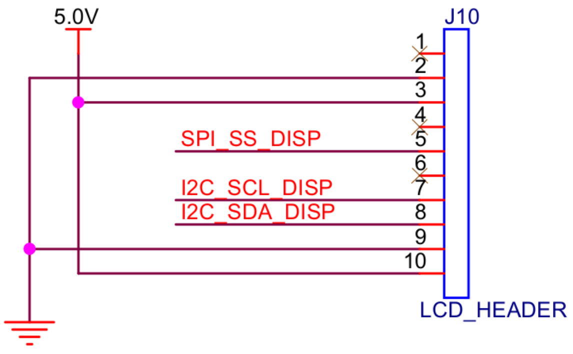


Figure 4.3: J10 header from the Cyclone V GT board schematic. The SCL and SDA signals can be used as single-ended transmitter and receiver signals.

board), or through a prototype area or header.[2]

The FPGA board used in this thesis has no dedicated prototyping area for external signal connection. It instead comes with an GPIO-extension board that connects to one of the two on-board HSMC-ports.[3] However, if removing the on-mounted LCD-display, it is possible to use the exposed header for signaling. The header (J10 in the board schematic, as shown in figure 4.3) is connected to a transmitter- and receiver pair (both running on a voltage of 5 V), several 5 V output pins and a ground pin. By using the transmitter- and receiver pair from the available header on the FPGA, it is possible to implement a type of asynchronous serial protocol. Most FPGA developement boards, including the Terasic Cyclone SX board (see section 1.3.2), comes with a built-in UART-to-USB interface. This allows you to connect the board directly to the USB-port of the PC as a serial connection. The FPGA board used in this thesis has no integrated UART interface, so this has to be implemented manually.

4.3 Duplex Systems

Common to all communication systems considered is that they are all duplex systems, which simply means two connected devices that can both transmit and receive signals. While *full-duplex* enables both devices to transmit and receive simultaneously (like a telephone), *half-duplex* only enables one device to transmit at a time (like a walkie-talkie). Common to both the duplex systems is that they have two communication channels.

[2]The user has to assign the available pins to the associated transmitter and receive signals in the *Pin Planner* program, as part of the Quartus II suite.

[3]By using both the HSMC-ports on the FPGA, the GPIO-PCB could not fit properly side-by-side with the HSMC-to-VLDB PCB (see chapter 3) because the latter PCB is a bit to wide.

4.4 Choosing Communication Protocol

Perhaps the most well-known and supported serial communications protocol out there is the RS-232 standard. It supports both synchronous and asynchronous transmission, and only requires a single transmit- and receiver pair (if excluding the data control signals, which are not crucial). It is compatible over a wide range of voltage levels, and can be connected directly to the serial port of a PC (if one is available) or through a USB-to-RS232 adapter. The RS-232-standard was chosen mainly because it only requires two wires (one for transmitting and the other for receiving), and can be easily implemented using available C-libraries. See appendix A.2 for more information about the RS-232 standard.

The standard will be used for asynchronous transmission of data between the PC and FPGA. A simple UART with a byte-decoder will be implemented on the FPGA-side, while a dedicated C-program with access to the COM port will be implemented on the PC-side. A USB-to-RS232 adapter with a voltage converter will be used as the connection between the FPGA and the PC. The following chapters describe the implementation of the serial interface, first on the FPGA side (section 4.5) and then on the PC side (section 4.6).

4.5 Hardware Design on the FPGA Side

The FPGA hardware design is responsible for treating incoming requests made by the user PC. It will essentially become a module to connect with the GBT example design, were it replaces the ISSP-module that is used today. The module must have access to the GBT control signal register, and this is done by re-connecting the already defined probe and source signals, which is connected to the ISSP, to this module. The module is mostly completed; what remains is to implement it in the GBT example design and connect the probe and source registers to the module, effectively replacing the ISSP module. For now, the module uses dummy registers instead of the real probe and source registers.

4.5.1 Specification

When finished, the implemented hardware logic is to contain the following specifications:

- Communication with the user PC using UART and RS-232.
  - Receive requests sent from the user PC. The requests are reserved byte-codes in which the decoder translates to read or write commands to use on the GBT control registers.
  - Send out data from the GBT control-register to the user PC when told. The address of the data must be specified and sent from the user PC.

– Manage to send and receive data at a reasonable speed, so that the control register information is updated at a practical rate on the user PC.

4.5.2 Hardware Components

The below sections gives a brief description of each part of the module design.
The UART itself was based on the UART-design by Pong P. Chu, found in chapter 7 from his book *FPGA Prototyping By VHDL Examples* [26]. The Baud Rate Generator was borrowed from the *Uart2Bus* VHDL-design by Arild Velure [27]. The UART-decoder was written in conjunction with the C-program on the PC side, and is the one component that ties the communication together.

The end result forms a design that uses a UART to receive and transmit $19200 - 8 - N - 1$ RS-232 data-bytes. The received bytes are stored in a FIFO until treated by the UART-decoder. The "decoder" translates the received bytes into requests, i.e a read-request if 0xDD or a write-request if 0xEE (write value 0) or 0xFF (write value 1), and manipulates or sends out data-bits from the GBT-register according to the received requests. The UART itself is optimized for 19200 bit/s, but has been tested to work at speeds up to 57600 bit/s.

UART

Simply put, a UART is a circuit that transmits and receives parallel data through a serial line [26]. Since it has no dedicated clock line, it uses the concept of oversampling to synchronize with the incoming data. This involves using a sample-clock which is 16 times faster than the transmitted/received data; the transmitting and receiving end must thus have matched data rates.

The UART-design is divided into five parts:

- A Receiver that receives the serial data and reassembles it into parallel data.
- A Transmitter that sends parallel data bit by bit out the serial line.
- A Baud Rate Generator that generates the right amount of ticks relative to the baud rate and global clock.
- Two FIFO-buffers connected to the transmitter and receiver to temporary store the bytes in the order of arrival.

Oversampling and the Baud Rate Generator

To obtain an accurate sampling of the received signal, an asynchronous system like the UART uses what is referred to as oversampling. Figure 4.4 illustrates the oversampling technique. With a typical rate of 16 times the baud rate, the receiver listens for the line to go from idle to the first start bit. When pulled low or high (depending on the system), a counter starts counting from 0 to 7. When the counter reaches 7, the received signal is roughly in the middle of the start bit. By sampling the bit in the middle of its time frame, the receiver avoids the noise and ringing that are generated whenever a serial bit changes [28]. When in the middle of the start bit, the counter needs to tick 16 times before it reaches the middle of the first data bit, the Least Significant Bit (LSB). The LSB can now be sampled, and the procedure is repeated $N-1$ times until reaching the last data bit, the MSB. If there is a parity bit, the same procedure is repeated one more time to retrieve it. After retrieving all the data bits, the same procedure is used one last time to sample the M stop bits at the end of the signal. After this, the line is held high until a new start bit arrives.

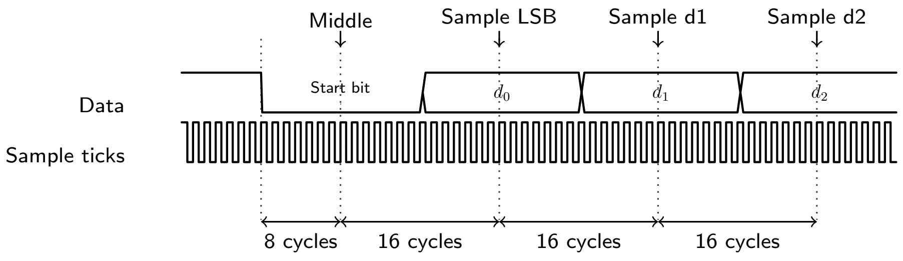


Figure 4.4: UART receive synchronisation and data sampling points with 16 times the sampling rate.

To achieve a sampling rate of 16 times the baud rate, a Baud Rate Generator module is implemented into the design. The module generates a one-clock-cycle tick once every $\frac{clock}{16\times baud\ rate}$ clock cycles. This is achieved by counting in certain steps given by the formula below:

$$Baud\ frequency = \frac{16 \times baud\ rate}{GCD(clock, 16 \times baud\ rate)} \tag{4.1}$$

, where baud frequency is the count steps, and GCD is the greatest common divisor between the global clock and the baud rate times 16 [27].
For a baud rate of 19200 bit/s and a clock of 50 MHz, the counter must count to 96 per clock cycle.

Once the counter reaches a given baud limit, the counter resets and the tick-signal is pulled high. After one clock cycle, the tick-signal is pulled low and the counter starts to count upwards again. The baud limit is given by:

$$Baud\ limit = \frac{clock}{GCD(clock, 16 \times baud\ rate)} - baud\ frequency \tag{4.2}$$

, where clock is the global clock of the system and GCD is the greatest common divisor between the global clock and the baud rate times 16 [27].
For a baud rate of 19200 bit/s and a clock of 50 MHz, the counter must count in steps of 96 up to the baud limit of 15565, before pulling the tick-signal high and start over again.

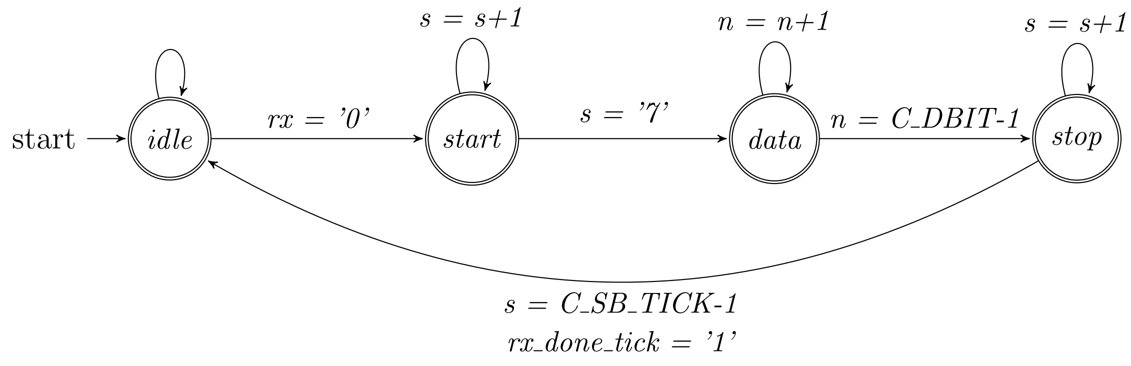


Figure 4.5: UART receiver state machine.

Receiver

The receiver is essentially a finite state machine, divided into four states: the idle-, start-, data- and stop state. It uses the Start and Stop bits to reset the state machine in an attempt to synchronize the clock phase to the incoming signal. For this, the receiver contains three registers: the s- and n registers for counting, and a b register for data storing. The s-register keeps track of the sample ticks and n-register the number of data bits sampled. There are two constants defined for the receiver: the *C_DBIT* constant, which indicates the number of data bits; and the *C_SB_TICK* constant, which indicates how many ticks that is required for the stop bit(s) (see *uart_gbt_pkg.vhdl*).

Transmitter

The UART transmitter has a similar design to that of the receiver; it uses the same state machine structure, but for the purpose of shifting out data. In addition to the s-, n-, and b-registers used for counting, the transmitter contains a din-register for parallel input data and a 1 bit tx-register for shifting out the data. To prevent multiple clocks, the baud rate generator is also used to clock the transmitter. For the transmitter to be properly synchronized with the receiver, it instead uses the counter registers to slow down the operation 16 times. This is because there is no oversampling involved in transmitting a signal.

FIFO Buffers

Both the UART transmitter and receiver has FIFO-buffers that stores the incoming data sequentially in a "First In, First Out" manner. On the receiver side, the incoming data is stored until it gets the read-out signal (*rd_uart* goes high). It then places the first stored byte on the (*r_data*)-line for one clock cycle. As long as the read-out signal remains high and there is bytes stored, the FIFO will spew out data

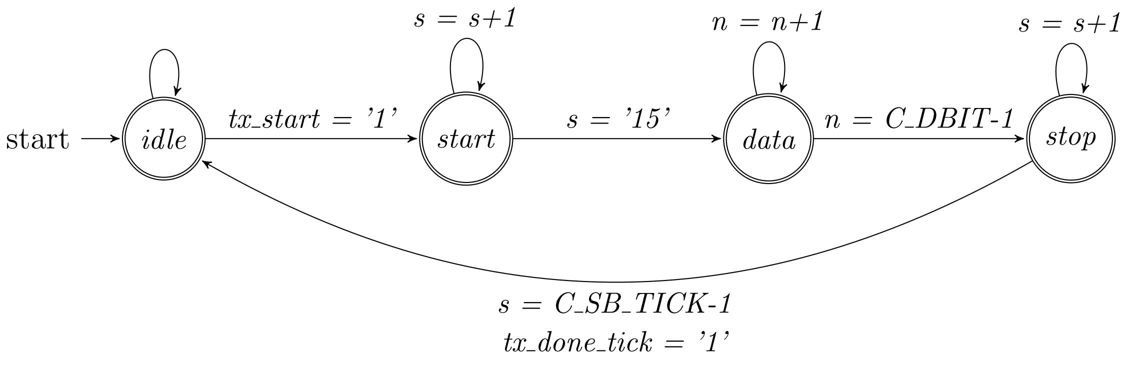


Figure 4.6: UART transmitter state machine, similar to that of the receiver.

on the (*r_data*)-line with the rising edge of the clock. The *rx_empty* signal indicates whether there is one or more bytes stored in the FIFO. On the transmitter side, when data from the FPGA is written into the FIFO, it sends a signal to the transmitter to start shifting out the data stored in the buffer, oldest byte first. Having FIFOs to store data between the UART and the rest of the FPGA logic is necessary to prevent data loss, as the FPGA logic operates at a much higher clock speeds than the UART can transmit and receive data. If a FIFO gets filled up, no new data will be written to it until data is read out of it, freeing one slot.

Decoder

The UART decoder is a state machine connected to the other end of the UART receiver- and transmitter FIFOs. It reads out the stored received bytes and does tasks according to the order and value of the bytes. Starting at the idle state, the decoder waits for a request byte from the FIFO followed by an address byte. Legal request bytes are 0xDD, for read; 0xEE, for write value '1'; and 0xFF, for write value '0'. The request-bytes must then be followed by a byte containing an legal address between 0x00 - 0x40. If all requirements are met, the state machine will perform the requested action on the probe and source registers of the GBT example design. A read of a given address will result in the decoder instructing the UART to transmit a byte containing the data value (stored in the MSB of the byte) and the register address of the data value (stored in the last seven bits of the byte). A write of a given address will result in the decoder changing the value of the given address in the source register of the GBT example design. The address must thus be smaller than 0x24. See appendix C for more information about the GBT source and probe registers.

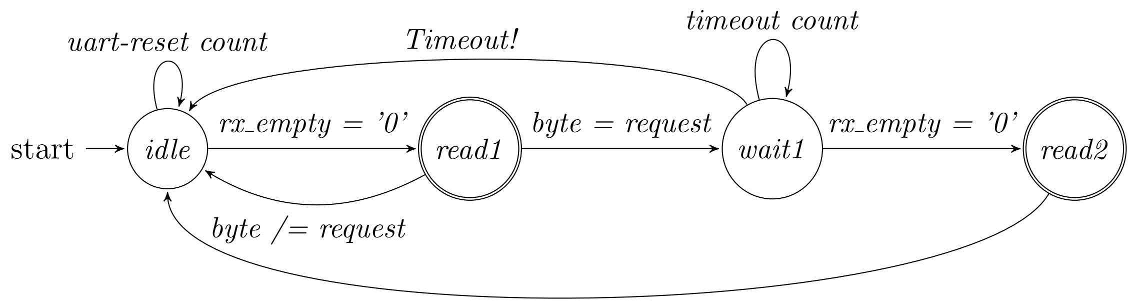


Figure 4.7: UART decoder state machine.

4.5.3 Conclusion

The goal of this hardware design was to develop a module that would eventually be integrated into the GBT example design, and together with a custom made software, manipulate the GBT control signals, effectively replacing the already existing ISSP module. The custom made module communicates well with the software; receiving requests and sends out information accordingly. During testing, it was discovered that the UART would hang after a random period of time. This was fixed by having a reset timer that would reset the UART if it did not respond for a given time period (see section 5.3.2).

4.6 Software on the PC Side

The program was written with the goal of one day replacing the Quartus-bound In-System-Source-And-Probe Editor, which is used today to access the GBT control-signals; and instead introduce a cross-platform, open-source solution. It was written in conjuction with the custom hardware, with the intention of communicating with the hardware to send and retrieve information from the GBT probe and source control registers. To access the serial port on the PC, the software uses free and cross-platform C-libraries (see appendix B). The software communicates with the hardware using the RS-232 protocol, using 8 data bits, no parity and 1 stop bit, and a baud rate at 19200 bit/s. The hardware module must have the same specification in order for this to work. The C programming language was chosen for this task mainly because of previous experience with the language.

4.6.1 Specification

The program specifications are as follows:

- The ability to send and receive bytes from the UART on the FPGA.

- An user interface that makes it easy to control and monitor the FPGA, i.e a command console.
- Cross-platform. This was not a critical requirement, but was added later because of the language the program was written in and also the already cross-platform libraries used.

4.6.2 Software Structure and Flowchart

The software is divided into two modules; one module for sending and receiving data, to and from the FPGA; and one module that acts as the actual software interface. Both these modules were intended to be merged together into one program, both because of time constraints were left separated. The below sections describes the inner workings of each of the two modules, and provides flowcharts to further illustrate the behaviour.

Interface module

Figure 4.8 illustrates the behaviour of the interface module. The main loop start by checking for any changes in screen size of the main terminal window. This allows the user to freely change the window size, although full screen is preferred. If a resize has occurred, the program stores the size-parameter and uses it to further resize and properly align all the internal windows (”Command History”, ”Commandline” and ”Signals”) using *wresize*, and clears the content using *wclear*. The content will later be redrawn with coordinates aligned according to the new size.

With ncurses, by using *getch* in conjunction with *nodelay*, one can check for key-presses without having to pause the entire program (see section B.4). If a key is pressed, the program analyses the input: If a character is typed, it is appended to a command-string and displayed in the ”Commandline” window.

If it is a return-character (’\n’), the program processes the command-string and compares it up against a table of legal commands. The legal commands has associated functions that executes if there is a match between the input and an entry in the table. For it to be a match, the first word of the input-string must be a legal command followed by legal arguments. To see the usage of a command, one can just type in the command without any arguments. It is also possible to type in just a letter or a partial word to print out possible commands that contain the letter or partial word typed in. Legal commands include: *write*, which writes a bit value to one or more of the switch-signals; *read*, which sends read requests for one or more signals to the FPGA; *open*, which opens the COM port if it is not already open; *close*, which closes the comport if it is open; *status*, which prints out information about the COM port and its current status; and *exit*, which closes the COM port and exits the program.

If either return or a character has been typed in, the program checks the input

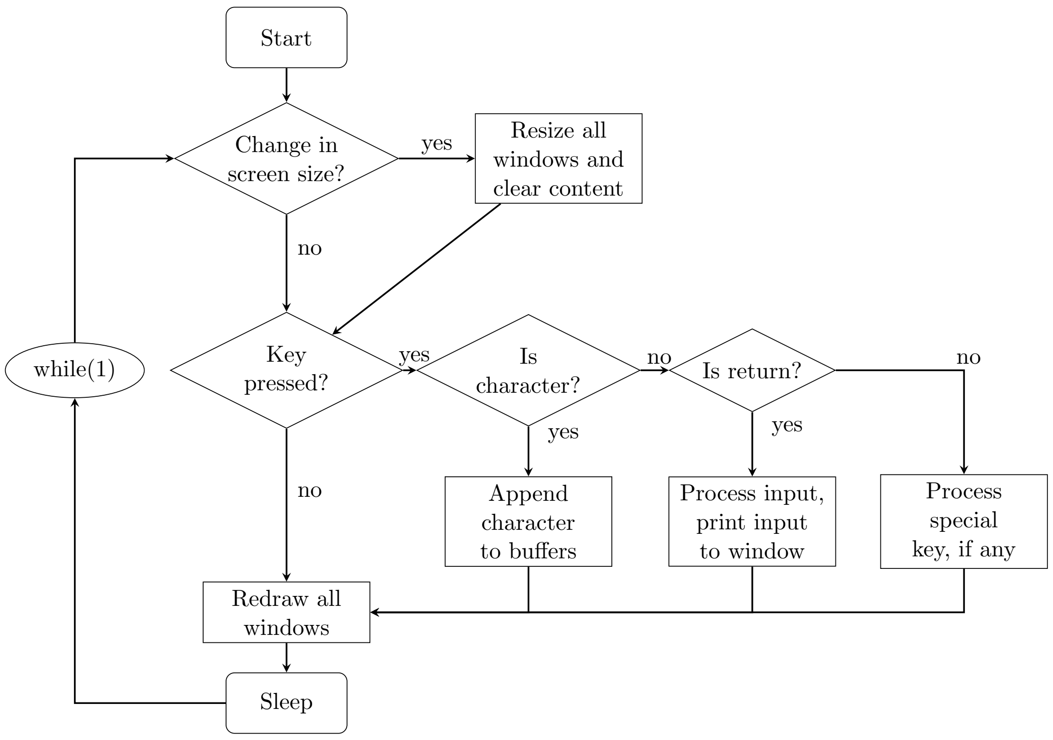

Start
Change in screen size?
yes
Resize all windows and clear content
no
while(1)
Key pressed?
yes
Is character?
no
Is return?
no
yes
yes
no
Append character to buffers
Process input, print input to window
Process special key, if any
Redraw all windows
Sleep

Figure 4.8: Flowchart over the main loop of the interface module.

for a special key. Special keys include: "Page Up" and "Page Down", that selects which signals to display in the "Signals" window (probes, switches or both); and "Arrow up" and "Arrow down", that browse previous typed commands and displays them in the "Commandline" window.

After checking for key-presses, the program redraws the windows using *wrefresh* and sleeps with a small delay (to prevent screen flickering and unnecessary cpu loads). It then returns to the top of the main loop and the process starts over again.

Send/Receive module

Figures 4.9 and 4.10 illustrates the behaviour of the main transmitter and receiver functions of the send/receive module. The basic operation goes as follows: The program sends a data-pattern from a 2d-array via the COM port to the GBT-registers on the FPGA side. It then sends a read request to read all values of the GBT-register back to confirm that the data pattern was transmittet properly. There are five different patterns that are sent in sequence to the registers, and each time a pattern is sent it is read back to the program on the PC side and displayed. This is to confirm that the communication link is working properly.

There are four variables that controls the transmitter: *txStatus*, *txReq*, *txData[2]* and *txAdr*. The *txStatus* variable controls the behaviour of both transmitter and receiver function. For the program to be able to send a new byte out the COM port, the *txStatus* variable must either be flagged as idle (0x00) or as repeat (0xFD). When flagged idle, the program is not waiting for any received address-byte and is ready to transmit a new request-byte to the COM port. When set to repeat, the program has already sent a request, but has not received the desired data and address byte, and must send the same read-request again. The *txReq* variable determines what request is to be sent to the FPGA. When equal to a read-request (0xDD), both *txStatus* and *txData[0]* is set to 0xDD. If *txReq* is equal to a write-request (0xEE), however, data from the *GBT data table* is copied into *txData[0]* using *txAdr* as index. *txData[1]* is set always equal to *txAdr*, which is now the desired address to the GBT-register in which to read from or write to. Sending data to the COM port involves using the *RS232_Sendbuf*-function (see section B.1). *txAdr* is then incremented by one or reset to 0 if it exceeds the width of the GBT data-register. If the latter is the case, it means that the read/write operation is finished, and *txReq* is set to the opposite operation. If a write operation is done, the pattern index is also incremented, so that the next write operation sends out a different pattern. The transmitter function is always called before the receiver function, with a specified delay in between the two function calls.

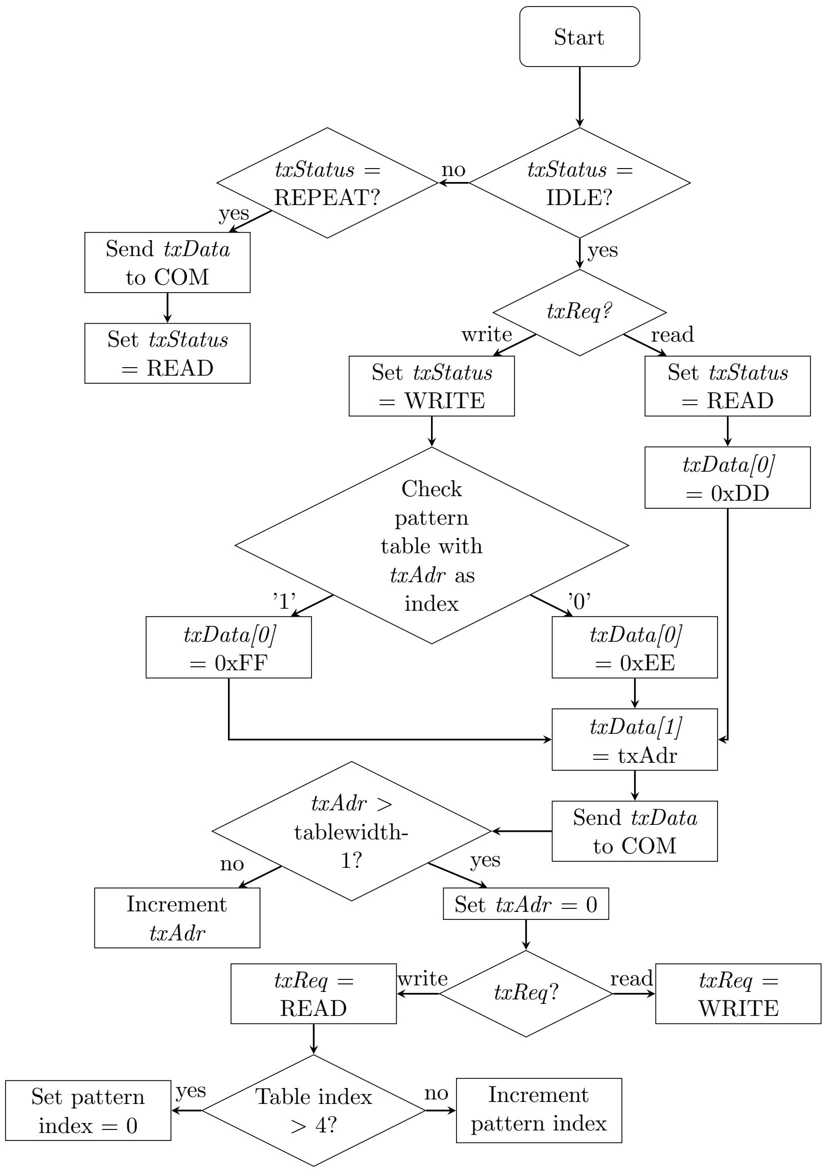


Figure 4.9: Flowchart over the transmitter function.

For the receiver function to read out what is stored in the COM port buffer, *txStatus* must be set to read (0xDD). The process of reading out a byte from the COM port involves using the *RS232_PollComport*-function (see appendix B.1). The n variable is equal to the number of bytes read from the COM port. If no bytes are present, this means that the previous read-request has not been received or processed properly by the FPGA UART state machine. *txStatus* is therefore set equal to repeat (0xFD), signaling the transmitter function to send the same previous read-request to the COM port. If n is larger than 0, the data and address values are filtered out of the byte(s); the data bit being the MSB and the address the remaining bits.

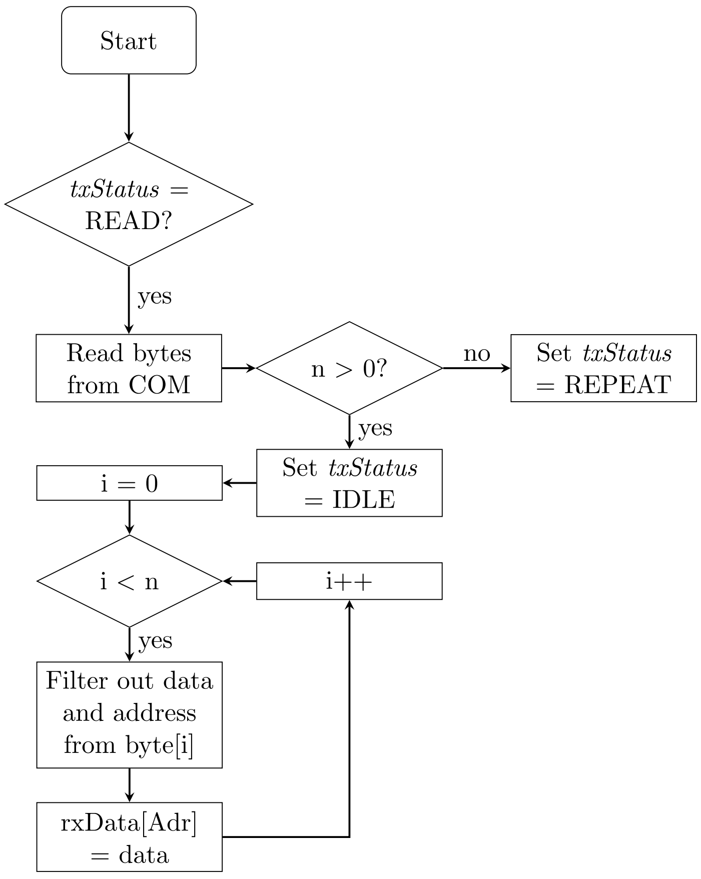


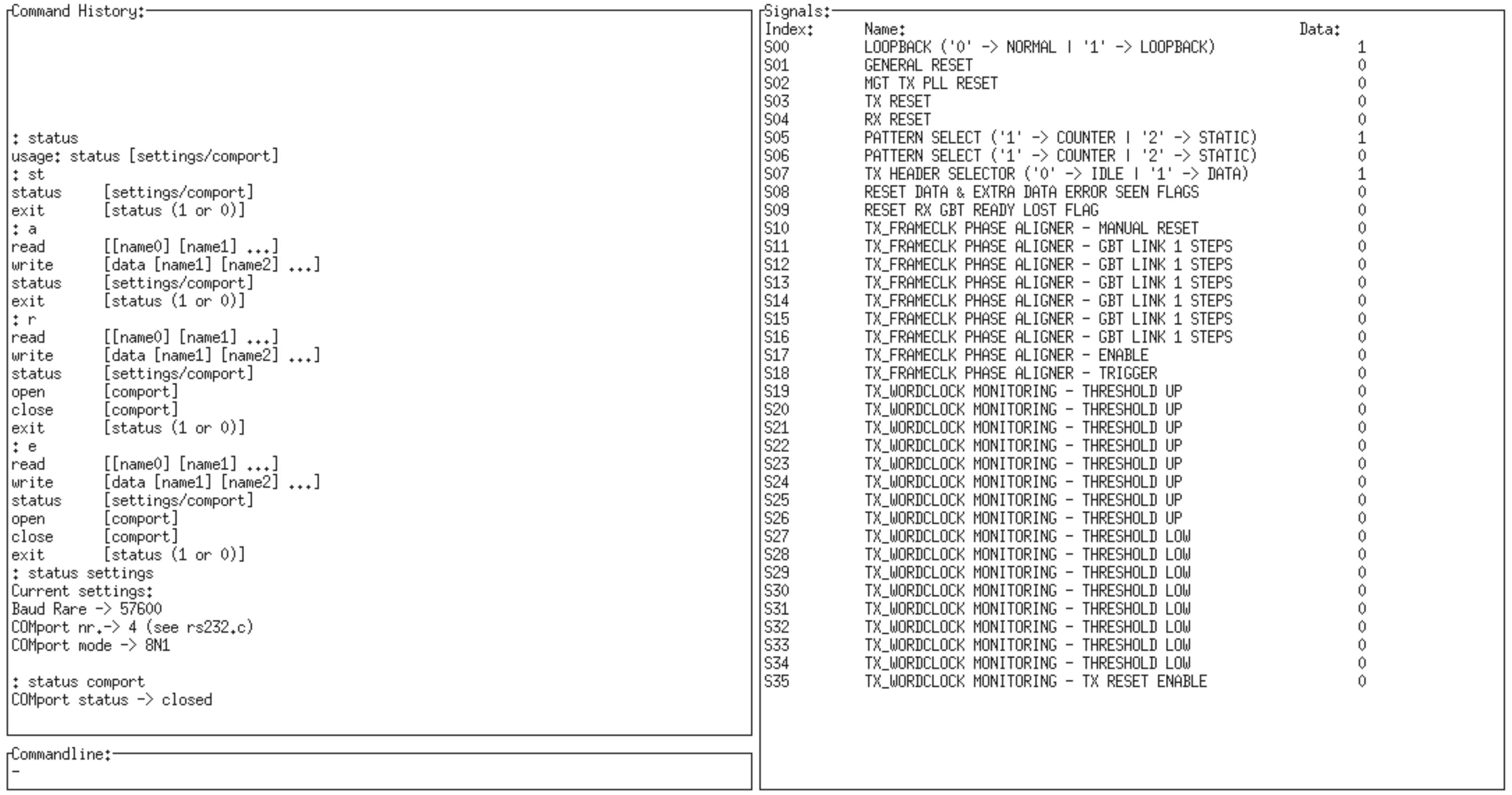
Figure 4.10: Flowchart over the receiver function.

4.6.3 Conclusion

The goal of the software was, together with the hardware design, to one day replace the Quartus-bound In- System-Source-And-Probe Editor, which is used today to access the GBT control-signals, and instead introduce a cross-platform, open-source and thus customizable software solution.

As for now, the serial interface software consists of two modules: the sending and receiving module, and the ncurses command interface. Both modules work more or less as they should: The send/receive module writes a pattern to the GBT-register in the FPGA and reads it back to the terminal, and the interface module allows you to perform write- and read-commands and monitor the GBT control signals. What remains is to fully integrate the send/receive module into the interface module. They are partly integrated in terms of merging the two into one program. What is left undone is to rewrite the main transmit and receive routines from the send/receive module, which are not completed. In addition, some adaptations needs to be made to the write- and read command-entries in order for them to invoke and direct the send/receive module. Also, a final test of the software as a whole remains.

A useful software feature to be added in future updates would be the ability to log the GBT register information into date-sorted files. This feature was intended to be integrated, but because of time constraints it had to be dropped. Figure 4.11 shows a snapshot of the GBT control interface.



```
Command History:



: status
usage: status [settings/comport]
: st
status     [settings/comport]
exit       [status (1 or 0)]
: a
read       [[name0] [name1] ...]
write      [data [name1] [name2] ...]
status     [settings/comport]
exit       [status (1 or 0)]
: r
read       [[name0] [name1] ...]
write      [data [name1] [name2] ...]
status     [settings/comport]
open       [comport]
close      [comport]
exit       [status (1 or 0)]
: e
read       [[name0] [name1] ...]
write      [data [name1] [name2] ...]
status     [settings/comport]
open       [comport]
close      [comport]
exit       [status (1 or 0)]
: status settings
Current settings:
Baud Rare -> 57600
COMport nr.-> 4 (see rs232.c)
COMport mode -> 8N1

: status comport
COMport status -> closed

Commandline:
-
```

```
Signals:
Index:     Name:                                                   Data:
S00        LOOPBACK ('0' -> NORMAL | '1' -> LOOPBACK)                  1
S01        GENERAL RESET                                               0
S02        MGT TX PLL RESET                                            0
S03        TX RESET                                                    0
S04        RX RESET                                                    0
S05        PATTERN SELECT ('1' -> COUNTER | '2' -> STATIC)             1
S06        PATTERN SELECT ('1' -> COUNTER | '2' -> STATIC)             0
S07        TX HEADER SELECTOR ('0' -> IDLE | '1' -> DATA)              1
S08        RESET DATA & EXTRA DATA ERROR SEEN FLAGS                    0
S09        RESET RX GBT READY LOST FLAG                                0
S10        TX_FRAMECLK PHASE ALIGNER - MANUAL RESET                    0
S11        TX_FRAMECLK PHASE ALIGNER - GBT LINK 1 STEPS                0
S12        TX_FRAMECLK PHASE ALIGNER - GBT LINK 1 STEPS                0
S13        TX_FRAMECLK PHASE ALIGNER - GBT LINK 1 STEPS                0
S14        TX_FRAMECLK PHASE ALIGNER - GBT LINK 1 STEPS                0
S15        TX_FRAMECLK PHASE ALIGNER - GBT LINK 1 STEPS                0
S16        TX_FRAMECLK PHASE ALIGNER - GBT LINK 1 STEPS                0
S17        TX_FRAMECLK PHASE ALIGNER - ENABLE                          0
S18        TX_FRAMECLK PHASE ALIGNER - TRIGGER                         0
S19        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S20        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S21        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S22        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S23        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S24        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S25        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S26        TX_WORDCLOCK MONITORING - THRESHOLD UP                      0
S27        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S28        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S29        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S30        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S31        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S32        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S33        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S34        TX_WORDCLOCK MONITORING - THRESHOLD LOW                     0
S35        TX_WORDCLOCK MONITORING - TX RESET ENABLE                   0
```

Figure 4.11: Snapshot of the GBT control interface in action. The colors in this picture has been inverted for a better visual representation.

Chapter 5

Testing and Verification

The follow sections gives a brief overview over the tests that were conducted on the PCB, software and hardware designed during this thesis. Loopback testing with the PCB was conducted using the GBT example design.

5.1 120 MHz Reference Clock

To be able to conduct a proper loopback test using the GBT example design, the GBT Link MGT and Phase-Locked Loops (PLLs) must have an input clock frequency of 120 MHz. There are a number of ways to achieve this on the Cyclone V GT board:

- Using an external clock, like a square wave signal generator.
- Using one of the onboard programmable oscillators.
- Implementing a PLL into the design that multiplies the onboard 50 MHz global clock up to the desired frequency of 120 MHz.

The original approach is to use an external signal generator with differential output to generate the reference clock, as shown in the GBT tutorial videos [29]. However, at the time of conducting the first tests, there was no available signal generators that could generate a 120 MHz square wave clock for the experiment. Because of this, some time was spent to investigate how to use the internal programmable oscillator as a reference clock.

The approach of implementing an extra PLL into the GBT-example design was also investigated, but attempts of doing so resulted in conflicts between the already implemented transceiver PLLs in the design.

The below sections gives the following descriptions:

- How to configure the onboard oscillator on the Cyclone V GT board for use as reference clock.

- How to configure the *Si*5338 external oscillator for use as reference clock.

5.1.1 Configuring the onboard Oscillator on the Cyclone V Board

To achieve a reference clock of 120 MHz without an external clock, the FPGA has an onboard programmable oscillator; the *Si*570 from Silabs. It uses Inter-Integrated Circuit (I2C) for serial communication and can be programmed to output frequencies up to 810 MHz $\pm 50 ppm$. To program the oscillator, Altera provides a dedicated software called "Clock Control". The Clock Control software is part of the Java based "Board Test System" software, included in the Cyclone V kit which can be found at Altera's websites [17]. The Cyclone V kit is board specific, so it is therefore important to use the right kit with the right board.

To make use of the Clock Control software has proven to be difficult, mainly because of the software being outdated in relevance to the current version of Quartus (at the time of writing, Quartus 15.0 is the newest edition). The solution was to install an older Quartus (version 13.1) and specify the right paths for the related environment variables. See appendix D for a description on how to setup the clock control software using Windows.

5.1.2 Configuring the Si5338 External Oscillator

For an external clock, the *Si*5338 evaluation board was used (see figure 5.1). It has an onboard 25 MHz crystal oscillator in addition to six SMA connectors reserved for external input clocks. For simplicity, the onboard oscillator was used in the thesis experiments. The remaining eight SMA connectors is reserved for differential output clocks. The evaluation board communicates with I2C and connects to the Personal Computer (PC) via Universal Serial Bus (USB) cabling. A dedicated clock builder software [30] lets you select clock outputs and type of signaling. For the experiments, a 2.5 V LVDS clock with 120 MHz was selected, and connected to the Cyclone V GT board using SMA cabling. To enable the SMA connectors as input to the reference clock on the Cyclone V GT board, *CLKSEL* on DIP switch SW4 must be selected ON (default is OFF).

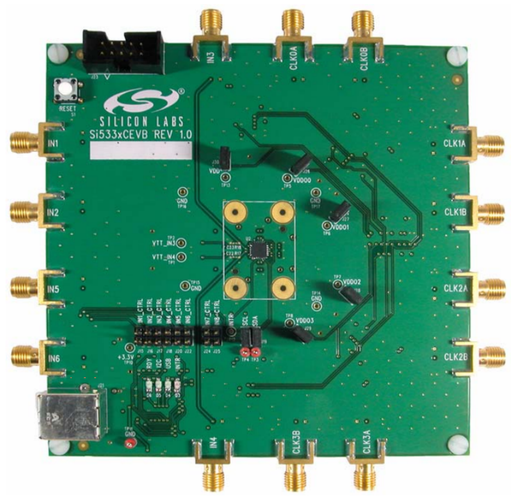


Figure 5.1: The *Si*5338 evaluation board [31].

5.2 Testing and Verification of the HDMI Daughter Card

To verify a working PCB, the HDMI daughter card underwent a series of tests, as summarized in the following sections.

5.2.1 Connectivity Test

Since all the components on the PCB were hand soldered (with the exception of the ground pads underneath the HSMC contact), it was particularly important to check for accidental shorts between pins and/or pads.

Purpose of Test

- Verify that there are no shorts between the pins and/or pads of the PCB.
- Check for current draw to confirm that there are no short on the power-lines.

Experimental Setup

The setup involves the use of a multimeter to probe all pins and check for connection faults according to the schematic. After all shorts have been eliminated, the PCB is to be connected to an external power supply to verify current-draw.

Results

Using a multimeter, all connections were checked and verified that there were no shorts (Some pins on the HSMC were indeed shorted and had to be re-soldered). The PCB was then connected to an external power supply, with a 3.3 V output and a current output limited to 100 mA. It was verified that the current draw, with all HDMI-connections left open, were no more than the current drawn from the power-LED and the 1.25 V voltage divider, i.e around 40 mA.

To confirm connectivity and that there were no further connecting shorts between the pads and/or pins, a simple test-circuit was written in Quartus. Beginning with the transmit-signals: all the relevant LVDS transmit-signals that are physically connected to the HSMC (port A) connector of the FPGA were connected to a given clock signal. The PCB was then connected to the Cyclone V board, and the HDMI connectors were probed with the help of an oscilloscope and verified that there was an output signal on every HDMI-transmitter.

The PCB is now ready for connection with the host FPGA for further testing of the transmitter and receiver signals.

5.2.2 External Loop-back Test for the Fiber-Optic Connector

To verify that the HDMI daughter card SFP-connector for fiber-optic communication is working correctly, an external loop-back test was conducted.

Purpose of Test

- Test the dedicated SFP-connector on the HDMI-daughter card for fiber-optic communication and see that it is capable of sending and receiving information at speeds corresponding the GBT-standard of 4.8 Gbit/s.

Experimental Setup

A fiber-optic cable connected from the transmitter to the receiver using a fiber-module, forming an external loop through the cable, was connected to the SFP-connector of the HDMI-daughter card PCB. Using the GBT Quartus-example together with the ISSP and SignalTap II, it was possible to perform a pattern check test by comparing the transmitted signals with the received signals. To achieve this test, ISSP was used to setup the pattern generator of the GBT example to count with increments of one (PATTERN SELECT = "1h") and the receiver to receive signals through external cabling (LOOPBACK = '0'). SignalTap II was used to monitor and verify that the receiver line received the same incremented counter that the transmitter sent out.

Results

Running the test resulted in a continuous stream of bits sent from the transmitter to the receiver. Using SignalTap II as a monitor limits to only observe parts of the transmission, but it was sufficient enough to see that the received sum of bits were indeed incrementing by one each time. The results are comparable with the same test using internal loopback (LOOPBACK = '1'), i.e observing that the receiving end is counting by increments of one. This concludes that the SFP-connector on the HDMI-daughter card works as intended at the desired speed of 4.8 Gbit/s.

5.2.3 External Loop-back Test for the HDMI Connectors

Purpose of Tests

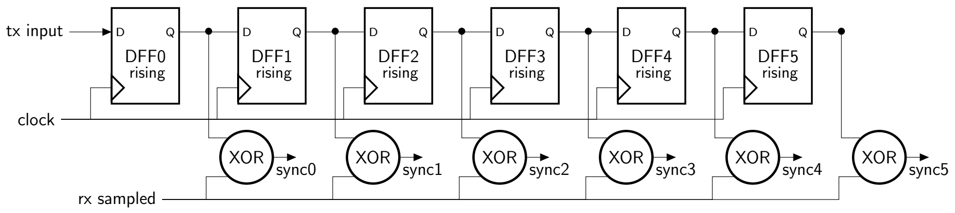
- Measure the quality of the signal (eye-diagram) at different frequencies up to 300 MHz and see how reflections affects the signal.
- Measure crosstalk between neighboring signal paths.
- Create a test environment using Quartus II in conjunction with SignalTap II to:
  - See if it is possible to sample the received signals at different frequencies up to 300 MHz.
  - Calculate the bit-error rate at different frequencies up to 300 MHz.

Experimental Setup

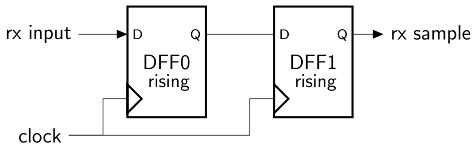
Each HDMI-connector has at least one transmitter- and receiver line. To simulate signal transmission over a distance, a HDMI cable was cut in half and the transmit- and receive lines were soldered together, creating an external loop-back. VHDL code was written to simulate a data-stream using a pseudo-random generator to generate random bit-patterns. The data-stream would travel out via one of the transmitters of the FPGA, out through the HDMI-connector, following the cable back into the same HDMI-connector into the receiver input. This would simulate the transmission path to the VLDB card. The transmitted and received bits would then be compared using xor-logic. A bit-counter register would count each transmitted bit, and a bit-error register would count each time two bits are different in comparison. The bit error rate is thus given by $\frac{bit\ errors}{number\ of\ bits}$.

Because of the trace- and cable-lengths and the fact that a signal has a finite propagation time, a delay is introduced between the transmitted and received bit-signals [1]. To cope with the delay differences, a delay chain is added in parallel with the transceiver line: While a bit-signal is travelling through the cable, the same

[1] As mentioned in chapter 3, a signal propagates through the conductor at a velocity of approximately 15 mm/ns.



(a) Delay chain on the transmitter line.



(b) Delay chain on the receiver line.

Figure 5.2: Transmitter and receiver delay chains.

bit-signal is delayed through a series of D Flip-Flops (DFFs). For each clock cycle, the value at the receiver input is compared with the output of each DFF using XOR-logic. The individual outputs of the XORs are connected to Light-Emitting Diodes (LEDs). The LED that has the least amount of toggling is the one XOR, and thus selected DFFs, that is most synchronized with the receiver (ideally, it should be toggle-free), and can be used to count bit errors. A smaller delay-chain was also added at the receiver line to synchronize the incoming bits with the clock. Figure 5.2 a and b illustrates these delay chains.

Cables with different lengths were used to see how this would affect the bit-error rate. This would thus introduce varying delay differences between the transmitter and receiver, and would therefore produce different results comparing the signals in the delay chain. A series of attempts were therefore made to find the most synchronized outputs of the delay-chains at different cable lengths.

Results

During the first samplings using SignalTap II, it was suspected that when running at a 300 MHz clock, the received signal were being sampled when in a metastable state, i.e in the middle of the rising edge of the bit. To compensate for this, a new delay

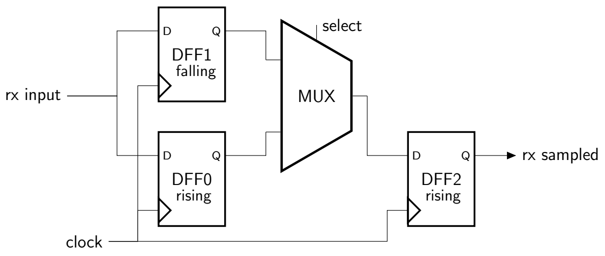


Figure 5.3: Muxed delay chain on the receiver line.

Clock [Mhz]	sync#	Signal path [cm]	Rising/Falling DFF	Comments
100	1	110	Rising	Some spikes occur.
200	2	110	Rising	No spikes.
300	4	110	Both	No spikes.
100	1	220	Rising	Some spikes occur.
200	3	220	Rising	Some periodical spikes.
300	5	220	Rising	No spikes.

Table 5.1: SignalTap II measurements on the J10 HDMI connector of the HDMI daughter card. "sync#" is the most synchronized XOR-output in the delay chain. Length is the approximate traveling path (cable and trace) for the signal. The "Rising/Falling DFF" tab shows which edge triggered DFF that worked best as the first stage at the receiver delay chain.

chain was designed on the receiver line (figure 5.3): two DFFs that would trigger on different clock edges (rising and falling), were placed in parallel as the first stage of the receiver delay chain. By using a MUX, one of the DFF outputs were selected to go into a third DFF, and the output of this third DFF was then compared with all outputs in the transmitter delay chain. The "Rising/Falling DFF" tab in table 5.1 states which of the clock edges that caused less toggling on the first delay stage on the receiver.

With the help of SignalTap II it was possible to sample the delay chain signals and produce the results displayed in table 5.1.

5.2.4 Conclusion and Discussions

When doing the external loopback test for the HDMI connectors (section 5.2.3), while sampling the LEDs using SignalTap II, some "spikes" occurred at the one LED that seemed toggle-free when observing it, and thus what seemed the most "in sync". The "spikes" were narrower than the 300 MHz clock, and seemed to occur randomly. Whether this is a interference or will affect the error-counter hasn't been confirmed, as the test was aborted due to an accident with the FPGA. The accident caused two of the pins on one of the current regulators on the FPGA board to short, resulting in a burned regulator. This rendered the FPGA useless and halted all remaining tests. Due to time constraints, the FPGA was not repaired in time for the delivery of the thesis.

What remains of this test is to use the most synchronized XOR-output for an error-count measurement over a certain time interval to calculate the bit error rate. Further investigation is also needed to see whether this way of synchronizing the transmitter and receiver produces credible error-counts at all.

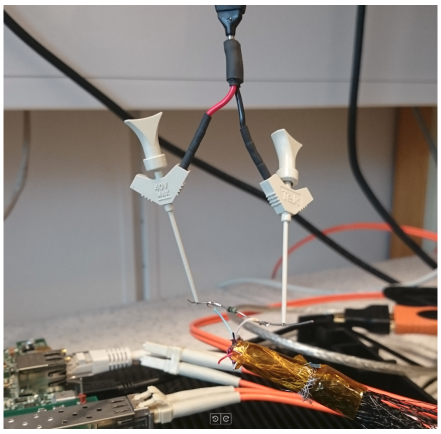
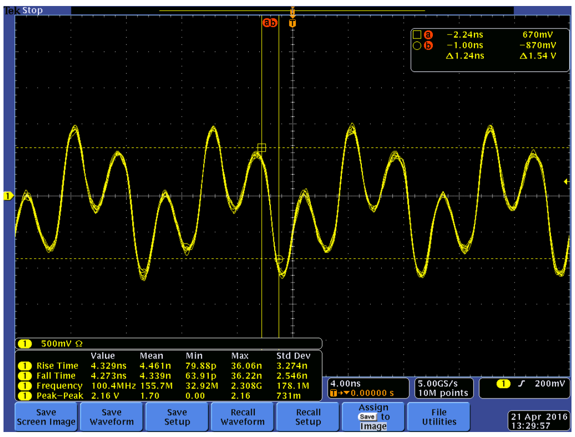


Figure 5.4: Small clamps caused huge reflections when measuring the high-speed signal.



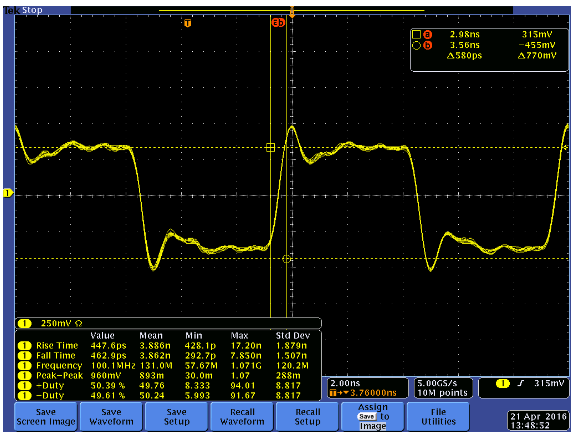
```
a  -2.24ns    670mV
b  -1.00ns   -870mV
   Δ1.24ns   Δ1.54 V

1  500mV Ω
                Value     Mean    Min     Max     Std Dev
1  Rise Time    4.329ns   4.461n  79.88p  36.06n  3.274n
1  Fall Time    4.273ns   4.339n  63.91p  36.22n  2.546n
1  Frequency    100.4MHz  155.7M  32.92M  2.308G  178.1M
1  Peak-Peak    2.16 V    1.70    0.00    2.16    731m

4.00ns        5.00GS/s      1  ƒ  200mV
T→▼0.00000 s  10M points

21 Apr 2016
13:29:57
```

(a) Using clamps.



```
a  2.98ns     315mV
b  3.56ns    -455mV
   Δ580ps    Δ770mV

1  250mV Ω
                Value     Mean    Min     Max     Std Dev
1  Rise Time    447.6ps   3.886n  428.1p  17.20n  1.879n
1  Fall Time    462.9ps   3.862n  292.7p  7.850n  1.507n
1  Frequency    100.1MHz  131.0M  57.67M  1.071G  120.2M
1  Peak-Peak    960mV     893m    30.0m   1.07    288m
1  +Duty        50.39 %   49.76   8.333   94.01   8.817
1  -Duty        49.61 %   50.24   5.993   91.67   8.817

2.00ns          5.00GS/s      1  ƒ  315mV
T→▼3.76000ns    10M points

21 Apr 2016
13:48:52
```

(b) Using stubs.

Figure 5.5: 100 MHz transmitter signal measured with differential probes.

When measuring a high-speed signal, the way you measure the signal can have a major impact on the result. This was experienced when attempting to measure the LVDS signals using the differential probes. On the first attempt, small clamps connected to each of the differential probes (figure 5.4) were coupled to a differential pair that were running from one end of a hdmi-cable, with a terminating resistor in between; in to a transmitter on the HDMI-daughter card. A clock signal was sent from the FPGA through the HDMI-daughter card to the end of the hdmi-cable.

The measurement resulted in signals that were heavily distorted by reflections. It was first thought that these reflections might originate from the traces on the FPGA itself, since the same results were produced when sending the clock signals through a different PCB (a GPIO-card) in place of the HDMI-daughter card. However, the theory was quickly rejected when using a completely different FPGA board produced the same reflections. The cause was in fact due to the small clamps that was used to connect the differential probes. By replacing these with small stubs and redo the measure, the result was quite different, as shown in figure 5.5.

By doing the same measurements again on different probe-points (viases) and with both the GPIO-card and the HDMI-daughter card on both available FPGAs, the reflections remained somewhat the same. It was concluded that the most notable signal reflections is caused by the measurement setup, and will not affect the digital data when transmitting or receiving signals. However, measuring signal integrity and creating an eye-diagram could not be done with this measurement setup. This lead to the loop-back test using SignalTap II in an attempt to sample the signals (see chapter 5.3).

5.3 Testing and Verification of the Serial Interface

5.3.1 Hardware Simulation using Testbench in Modelsim

To verify that the hardware design was working properly in terms of correct signal timing between the baudrate, UART and decoder, a testbench was developed. The testbench was made using the Bitvis Utility Library and the design simulation was done using Altera's Modelsim.

Purpose of Tests

- Verify correct timing between the baudrate generator, clock and uart.
- Verify correct timing for the uart decoder.
- Verify that the design is working correctly in simulation.

Bitvis Utility Library

The *Bitvis Utility Library* by Bitvis is an open source VHDL testbench infrastructure library for verification of FPGAs and ASICs [32]. It was developed with the aim of simplifying the testbench development process when testing VHDL designs. It was chosen for this test because of the ease of use and fast implementation (see uart_tb.vhd for testbench implementation).

Experimental Setup

The main testbench process goes as follows: Send a request byte to the rx-signal, followed by a register-address. The request can be a read-request (0xDD) or a write-request (0xEE for '1' and 0xFF for '0'), and the address must be between 0x00 -

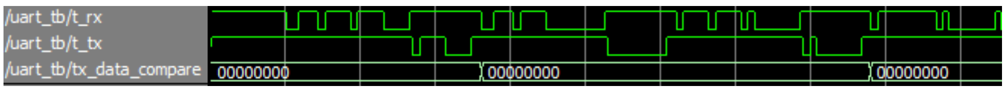


Figure 5.6: Three read operations followed by a write.

0xC1. Then, the tx-signal are sampled and compared with the requested address byte and checked for equality. If it is not equal, the test goes to a halt with a corresponding fault-message. The *UART_WRITE_BYTE* and *UART_READ_BYTE* testbench procedures were written with this in mind. To simulate an incoming byte at the receiver, *UART_WRITE_BYTE* inputs a vector of 8 bits as the data-byte. With the period of a transmitted bit, which is equal to $1/baudrate$, it outputs first a start bit, then the data bits and finally the stop bit. The rx-line takes the output of *UART_WRITE_BYTE* as input. *UART_READ_BYTE* takes the tx-line as input, and with the period of a transmitted bit shifts the tx-value into a data vector. Using the Bitvis *check_value*-function, the data vector is then checked and compared with the address-byte previously sent to the receiver. The *UART_READ_BYTE* procedure must be executed after the *UART_WRITE_BYTE* procedure.

Other test procedures involves checking the constants found in *uart_gbt_pkg.vhd* up against legal values, and also make sure that the tx- and rx-lines are in an idle state at the start of the test, i.e a high state.

Results

Figure 5.6 shows three read operations followed by a write operation. The *tx_data_compare* vector compares the address received with the address transmitted to confirm that the UART decoder does what it is requested to, i.e send out the correct addresses when receiving a read-request and write a bit-value to the GBT register when receiving a write-operation. The data-bit (MSB of the transmitted signal) is not included in the *tx_data_compare* comparison, nor is the write operation.

5.3.2 Running the PC software design together with the FPGA hardware design

Check if the FPGA design is responding correctly to the bytes sent from the C-program on the PC side.

Purpose of Tests

- Verify that the C-program is in control of the COM port.
- Verify that there is communication between the FPGA UART and the PC COM port.

- Confirm that the FPGA design and C-program is working together the way they should.

Experimental Setup

The Cyclone V Gt FPGA was programmed with the hardware design (section 4.5) and connected to the user PC using a USB-to-RS232 cable with a 5 V voltage converter between the FPGA and the cable. The software was granted PC admin access so that it was allowed access to the COM port. The PC COM port itself was configured in device manager (Windows) with a baud rate corresponding to the baud rate set in the hardware design and software.

Results

The software on the PC side was able to communicate with the FPGA via the COM port, and was able to read the registers as well as write to them. In some occurrences the software did not receive all addresses it requested a read on, resulting in some values not being read as often as others. To fix this, the repeat-functionality was implemented into the software (see section 4.6.2), and this fixed the problem.

While testing the hardware module together with the send/receive module, when instructing the UART to continuously send data from the GBT registers (by sending read-requests using the software), the UART would stop responding after a random period of time. To make it respond again, the UART had to be hard-reset. The reason why this is happening is not known, but it might be caused by a possible design flaw. A quick fix to this, however, was to simply implement a timer that would reset the UART unit on the condition that no bytes arrives in a given amount of time when it is in the idle state; this time period are larger than the time it takes for two bytes to arrive. This is not a permanent fix, since there is a chance that the UART might reset while a byte is under transmission. This might give an explanation as to why the C-program in some cases did not receive all the data it requested during transmission.

To further investigate this fault, one could use SignalTap II to probe the UART signals and identify and correct irregular behavior in relevant signals. Due to time constraints, the quick fix was preferred.

5.3.3 Conclusion and Discussion

By using a testbench, it was possible to verify that the hardware design behaved correctly. The decoder performed as it should in terms of reading and writing to the registers with correct timing. The C program (send/receive module) was able to send and receive information from the hardware registers located in the FPGA, confirming a working communication between the PC software and FPGA hardware.

While doing continuous read and write operations, the UART would eventually stop responding. This was fixed by implementing a timed reset on the UART unit. The timed reset approach is not a permanent fix, and should be investigated further.

Chapter 6

Summary and Conclusion

The future upgrade of the Large Hadron Collider accelerator, the High-Luminosity LHC, has its goal of increasing the beam luminosity by ten times. The GigaBit Transceiver ASICs and transmission protocol was developed with this in mind, and provides a high radiation tolerant, high speed, optical transmission line capable of simultaneous transfer of readout data, timing and trigger signals in addition to slow control and monitoring data. This thesis has had its focus on designing a control interface for the off-detector part of the GBT, the Common Readout Unit, along with a PCB that connects the CRU to the GBTx chip to allow for testing of the chip detector. The PCB was developed with high-speed transmission in mind, allowing for 4.8 Gbit/s optical transmission with minimum reflections using an SFP-contact. In addition, it has 10 HDMI-contacts allowing for e-link interface to the GBTx with a 320 Mbit/s transmission rate. The control interface was written in the C-language, using freely available libraries. A library was also written for the purpose of this thesis, with the GBT control signals in mind (Appendix B). A hardware module was implemented in the FPGA to allow the software to read and write information to the GBT probe and source registers. The software consists of two modules: the sending and receiving module, and the ncurses command interface. Both modules work more or less as they should: The send/receive module writes a pattern to the GBT-register in the FPGA and reads it back to the terminal, and the interface module allows you to perform write- and read-commands and monitor the GBT control signals. What remains is to fully integrate the send/receive module into the interface module.

When attempting to measure the signal integrity directly from the traces on the PCB, the measurement equipment caused major reflections, making the measurements unreliable. The reflections remained constant when measuring at different probe points, but varied when changing the probe-tips for the differential probes. External loopback tests were conducted on the HDMIs and SFP-contact using the GBT example design. The SFP-contact had a fiber-optic cable arranged in a loopback manner. Using the GBT example design, a counter was transmitted and received with the same increment at 4.8 Gbit/s, confirming a working SFP-contact.

The HDMI connection were tested by having a HDMI cable cut in half an soldering the transmitter and receiver pairs in a loopback manner. A pseudo-random signal was sent through the cable and a delay chain attempted to synchronize the two signals so that an bit errors could be counted and thus determine the bit error rate. The delay chain was completed, but because of an accident that lead to a damaged FPGA board, the bit error-count test was not conducted. Other remaining tests includes measure of crosstalk between paths and determining the signal integrity using eye-diagram.

The communication hardware on the FPGA side was tested and approved by writing a testbench using the Bitvis Utility Library. Correct timing between the UART and the decoder were confirmed, and the design was confirmed to be working in simulation. The send/receive software module were tested by connecting the COM port of the PC to the FPGA using a USB-to-RS232 cable, and by running the custom hardware on the FPGA, the software was able to communicate with the FPGA and was able to read the GBT probe and source registers as well as write to them.

The hardware module is not yet implemented into the GBT example design itself. During testing, dummy registers were therefore used instead of the actual GBT registers. The module should be integrated by removing the ISSP module in the GBT example design and connecting the leftover probe and source registers signals to the hardware modules dummy register signals.

Appendix A

Basics

This chapter contains information about basic concepts and devices used in this thesis.

A.1 Field Programmable Gate Array

A FPGA is a high density Integrated Circuit (IC) that is designed to be completely programmable by the customer after manufacturing (i.e. when the chip is shipped and "in the field"). The chips are shipped completely "blank", meaning that there are no pre-programmed logic[1]. An FPGA can either be re-programmed using SRAM technology[2], or one-time programmed by burning antifuses. The latter method makes it less prone to soft errors when exposed to radiation.

FPGAs are composed of arrays of Configurable Logic Blocks (CLBs) surrounded by programmable routing resources and I/O pads. A CLB consists of a LookUp Table (LUT) together with a clock and simple write logic, and uses the address bus of the LUT as the function input pins and the value at the selected address as the function output. LUTs are considered fast logic, since computing a complex function only requires a single memory lookup. The LUT can be made out of an SRAM memory-block [20].

The FPGA (section 1.3.2) used in this thesis is a re-programmable type. It stores the user hardware setup in an SRAM memory to configure routing and logic functions. For a permanent program storage, the FPGA can be configures to store the hardware setup in the on-board flash memory[3], which then programs the SRAM

[1]With the exception of FPGA evaluation boards, which comes shipped with a pre-programmed hardware setup for demo purposes.

[2]Short for Static Random Access Memory, SRAM is a type of volatile memory that uses internal feedback to keep on its data.[20] Volatile meaning that the memory loses all the data once the power is switched off.

[3]A nonvolatile memory type known by its name because of its ability to erase memory blocks

when powered on.

A.1.1 Hardware Description Language

The most common way to program an FPGA is by Hardware Description Language (HDL), with the major ones being Verilog and VHDL. HDLs consist of text-based expressions used to describe digital hardware and use this to further simulate and synthesize the hardware described. A hardware module is simulated by applying information at the inputs and then do a check on the corresponding outputs and verify that they behave as intended. Synthesis of hardware means transforming the HDL code into a netlist of logic and wire connections describing the hardware. This can then be compiled into an FPGA which re-wires the available internal CLBs according to the given instructions. HDLs have become more and more useful as system complexity have increased [20]. The hardware described in this thesis is done using VHDL, a strongly typed HDL that is known for its capability of describing parallel processes.

Below is a small example describing the logic of a D Flip-Flop using VHDL:

```
-- Example of a D Flip-Flop triggered on the rising edge of the clock.
library IEEE;
use IEEE.std_logic_1164.all;

entity DFF_high is
  port(
    D : in STD_LOGIC;
    Q : out STD_LOGIC;
    Qbar : out STD_LOGIC;
    RESET : in STD_LOGIC;
    CLK : in STD_LOGIC
  );
end DFF_high;

architecture rtl of DFF_high is
begin

process(CLK,RESET)
begin
  if (RESET='1') then
    Q <= '0';
    Qbar <= '1';
  elsif (CLK'event and CLK='1') then -- Can also use rising_edge(clk)
    Q <= D;
    Qbar <= not D;
  end if;
end process;

end rtl;
```

all at once "in a flash".[20] Nonvolatile meaning that the memory will keep all its data even if the power is switched off.

A.2 RS-232

RS-232 standard is a transmission protocol for full duplex, serial transmitting and receiving of data. The standard defines electrical characteristics and timing of signals, the "meaning" of the signal, and also physical size and pin-out for connectors. For communications to work properly, the Data Terminal Equipment (DTE), in this case the PC; and the Data Communication Equipment (DCE), in this case the FPGA, needs to agree on the same data-packet settings, i.e the baud rate, the number of data bits, any additional parity bit and the number of stop bits.

A typical RS-232 data-packet consists of a start bit, followed by 5, 7 or 8 data bits; 1 parity bit, for error checking; and 1 or 2 stop bits, indicating that the transmission is done. The start bit is typically a logical low and the stop bit(s) high (this is usually the case, but can be system dependent). The transmission line remains high until the start bit pulls it down low, and the data transfer begins until the stop bit is reached. The line is then kept high until a new start bit pulls it low again for a new byte to be transfered. Data-packets are often described in the form: $19200 - 8 - N - 1$, which simply means 19200 bit/s, 8 *data bits*, *No parity* and 1 *stop bit*. Figure A.1 demonstrates a typical RS-232 signal.

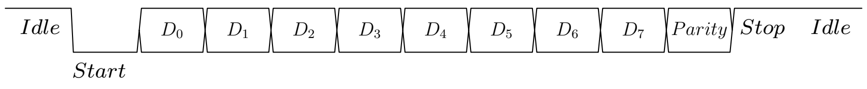


Figure A.1: Example of an RS-232 signal with 8 data bits, 1 parity bit (Odd, Even or No parity) and 1 stop bit.

Appendix B

Non-standard C-Libraries

The control interface software was developed with the help of a few non-standard libraries; non-standard meaning that the libraries are not a part of the C standard libraries. In addition, the ”Signals” library was written, which is dedicated to hold the GBT control-signal information. The following sections gives brief descriptions of each of these libraries and the associated functions used in the control interface software:

B.1 RS232

The RS232 library (*rs232.h* and *rs232.c*) is a cross-platform C-library for sending and receiving bytes from the COM of a PC. It was written by Teunis van Beelen, and is licensed under the ”GPL version 3” licence [33]. The library compiles with GCC on Linux and Mingw-w64 on Windows. By specifying baud rate, the name of the relevant COM, and the transmission mode (8N1 is standard), the library provides access to the COM and allows for both reading and writing to it. For more information about the library and functions, visit http://www.teuniz.net/RS-232/#.

B.1.1 Associated functions

```
int RS232_OpenComport(int comport_number, int baudrate, const char * mode)
```

Opens the COM. The user must specify a COM number, a baudrate number and the transmission mode (see list of valid inputs in *rs232.c*). It is important that the user grants super-user/administration priveliges to the program, or it might not be able to open the COM or change the baudrate correctly. The function returns 1 if it does not succeed in opening the COM.

```
int RS232_PollComport(int comport_number, unsigned char *buf, int size)
```

Returns the amount of bytes (in integers) received from the COM. The received bytes are stored in a buffer and pointed to by *buf*. One must specify the *size* of the buf pointer. It is recommended to call this function routinely from a timer.

```
int RS232_SendBuf(int comport_number, unsigned char *buf,int size)
```

Sends a buffer of bytes via the COM. The *buf* pointer must point to an array of bytes, and *size* must specify the correct size of the array pointed to by the *buf*.

```
void RS232_CloseComport(int comport_number)
```

Closes the COM. To prevent errors, it is important to disable any timers that calls COM related functions before closing the COM.

B.2 Timer

The Timer library (*timer.h* and *timer.c*) is a library for handling timers in a C environment. It was written by Teunis van Beelen, and is licensed under the "GPL version 3" licence [33]. The accuracy of the timer is system dependent. It supports a resolution down to 1 ms on the Windows platform and a resolution down to 1 µs on the Linux platform (though, in real-life situations, the timer might not be as accurate). The timer function is used with the *RS232_PollComport*-function in mind (see function definition above). For more information about the library and functions, visit http://www.teuniz.net/RS-232/#.

B.2.1 Associated functions

```
int start_timer(int milliSeconds, void (*)(void))
```

Starts the timer. At a given time interval of *milliSeconds* (milliseconds in Windows, microseconds in Linux), it calls a given function. The call-function must be a void function with no inputs (just like a standard main function without the argument calls). The *start_timer* function are called only ones in main, and repeats calling the given function every *milliSeconds* interval until the *stop_timer* is called.

```
int stop_timer( void )
```

Stops the timer if *start_timer* has previously been called. If the timer routinely calls RS232-functions, *stop_timer* must be called before closing the COM at the end of the program to prevent errors.

B.3 Signals

The Signals library (*signals.h* and *signals.c*) was written with the GBT control-signals in mind. It features methods for storing and manipulating the control-signal information, and in some extent encapsulates the information by partly hiding the information from the user. This is done by defining the main information-structure inside the *signals.c*-file, instead of defining it in the header file (and thus exposing it to the main program). A dedicated pointer is used to point to the structures namespace, and the functions defined in the header file calls the pointer as input instead of the structure itself. The main program only have access to what is defined in the header, thus limiting the user to only have access to the structure through dedicated library functions.

B.3.1 Associated structures

```
typedef struct {
  char *name;
  char *index;
  Byte type : 1;
} Type;

 struct _Signal {
  Type type;
  int i;
  Byte data;
};
```

The *Type*-structure contains the signal name and index in string-form, and a 1-bit type variable that defines the signal as either a probe (0) or a switch (1). *_Signal*-structure is the main structure and contains the *Type*-structure along with a "real" index, i, which is the actual data-address used to access the correct register-address on the FPGA; and the data byte, data, which stores the data bit of the signal. If the structure is defined as a probe, the data can only be read from the FPGA. If it is defined as a switch, on the other hand, it can be both read from the FPGA or the main program can write data to the FPGA.

B.3.2 Associated functions

```
Signal Signal_New (void)
```

Assigns a new pointer to the *_Signal*-structure and allocates enough memory.

```
Signal Signal_Init(char *index, char *name, Byte data)
```

Uses *Signal_New* to define a new signal, and in addition assigns values to the signal variables. The *index* string must contain either an *'S'* or a *'P'* character followed

by two number-characters. This is to indicate that the signal is either a probe or a switch with a given register address, for example: "P00", "S35".
The first letter is used to assign the *type*-variable and the two number-characters are converted to an integer and assigned to the "real" index, i, of the structure. The *name* string should contain a descriptive name of the signal, for example: "TX_FRAMECLK PHASE ALIGNER - PLL LOCKED".

```
void Signal_InitFromFile(Signal s[], int width, char *filename)
```

Uses *Signal_Init* to define a signal-array using information given from a file. The array has a width given by the *width*-variable. Refer to *signal_probe.txt* or *signal_switch.txt* for examples on how to set up a text-file. In the program, the *Signal_InitFromFile*-function is used twice to define two signal-arrays; one for the *Switches* and one for the *Probes* of the GBT control-signals. Defining two signal-arrays makes it more convenient when separating what is writeable and what is read-only.

```
void Signal_Free(Signal s)
```

Frees the given signal-pointer and all the associated variables.

```
void Signal_FreeArray(Signal *s, int n)
```

Uses *Signal_Free* in a loop to free up an array of signal-pointers.

```
int Signal_PrintSet(WINDOW *win, int *gy, Signal s[], int width)
```

Prints out all printable signal information of a signal-array to a *curses*-window (see section B.4) with a row-position given by *gy*. *gy* is defined as a pointer because it reports back to the program on which line in the window the information is printed on.

```
void Signal_setData(Signal s, Byte value)
Byte Signal_getData(Signal s)
```

Functions related to reading and writing to the data-variable, data, of a given signal.

```
void Signal_setIndex(Signal s, int i)
int Signal_getIndex(Signal s)
```

Functions related to reading and writing to the "real" index-variable, *i*, of a given signal.

```
void Signal_setType(Signal s, Byte type, char *name, char *index)
void Signal_getType(Signal s, Byte *type, char *name, char *index)
```

Functions related to reading and writing to the *Type*-structure of a given signal.

```
void Signal_setTypeType(Signal s, Byte type)
Byte Signal_getTypeType(Signal s)
```

Functions related to reading and writing to the type-variable, *type*, of the *Type*-structure of a given signal.

```
void Signal_setTypeName(Signal s, char *name)
char *Signal_getTypeName(Signal s)
```

Functions related to reading and writing to the name string, *name*, of the *Type*-structure of a given signal.

```
void Signal_setTypeIndex(Signal s, char *name)
char *Signal_getTypeIndex(Signal s)
```

Functions related to reading and writing to the index string, *index*, of the *Type*-structure of a given signal.

B.4 ncurses

The ncurses library is a terminal control library commonly used with Unix-systems. It is "freely redistributable in source form" [34], and enables you to construct interfaces using the terminal as the base. The main purpose of using ncurses for the GBT software interface was to enable multiple windows; one window to continuously update and display the GBT control signals, and another window for the user to write commands. To compile ncurses-based programs in Windows, one can use the *pdcurses* library. The *pdcurses* library has the same function names and overall functionality, so no alterations of code is needed. The following library and function information has been taken from http://linux.die.net/man/, which is a website collection of Linux-related documentation (ncurses is integrated into the Linux system).

B.4.1 Associated functions - Initialization

```
WINDOW *initscr(void);
```

Initialization function that sets up a new ncurses window. This needs to be called once, and before any other ncurses related function.

```
int noecho(void)
```

Disables the user input from being printed back on screen, i.e being echoed.

```
int cbreak(void)
```

Disables line buffering, making characters typed by the user available to the program one-by-one.

```
int keypad(WINDOW *win, bool bf)
```

Function that, when **TRUE**, allows the program to capture special keys using the function *getch()*, like *backspace* and the arrow keys. For this software, *win* is defined as *stdscr*, which is the main terminal screen.

```
int nodelay(WINDOW *win, bool bf)
```

Function that keeps the input capture function, *getch()*, from blocking the program when no input has been received. When *bf* is **TRUE**, getch() will return **ERR** instead of waiting for a key to be pressed. This function was necessary to allow one window to update freely while another window handled input. *win* is defined as *stdscr*.

```
void getmaxyx(WINDOW *win, int y, int x)
```

Function that store the current beginning coordinates and size of the specified window. In the case of the GBT software interface, *win* is specified as the main terminal screen, *stdscr*. These coordinates are needed when adding individual windows inside *stdscr*.

```
WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x)
```

Function used to initiate a new window within the main terminal screen, *stdscr*, where *nlines* a *ncols* is the number of lines and columns the window is to contain, and *begin_y* and *begin_x* is the upper left hand corner of the window. This function is used to initiate the three windows, "Command History", "Commandline" and "Signals" in the control interface software.

```
int scrollok(WINDOW *win, bool bf)
```

Function that, if **TRUE**, enables scrolling of typed in characters, either by using the additional *scroll*-function, or when the cursor reached the last column of the last line in the specified window *win*. This function is only used with the "Command History" window to scroll commands that are read in.

```
int leaveok(WINDOW *win, bool bf)
```

Function to disable the cursor at the specified window. It is used to disable the cursor in the "Command History" and "Signals" windows in the control interface software.

B.4.2 Associated functions - Various

```
int wresize(WINDOW *win, int lines, int columns)
```

Enables for resizing of the initiated windows.

```
int wclear(WINDOW *win)
```

Clears the specified window of its content.

```
int mvwprintw(WINDOW *win, int y, int x, const char *fmt, ...)
```

Analogous to the printf routine, found in the C standard library. However, this allows to print output in a specified window with coordinates. *mvwprintw* is used when printing information out on the different windows.

```
int wrefresh(WINDOW *win)
```

Function to update the individual windows. This function must be called to get actual output to the terminal, preferably at the end of the main loop.

```
int wmove(WINDOW *win, int y, int x)
```

Function that allows for repositioning of the cursor at the specified window. This function is called to reposition the cursor in the "Commandline" at the end of an input string.

Appendix C

GBT Control Signals

The Quartus-bound ISSP offers a way for the user to control and monitor the control signals of the GBT example design. Table C.2 and C.1 gives a brief description of these control signals. Since the latency optimized version of the GBT is not covered in this thesis, the signals related to this version is not described. The information was obtained from the GBT FPGA video tutorials [29]. The source (S) signals are described as switches that can be manipulated by the user, like resets and pattern selects. The probe (P) signals are emulated measuring probes that monitors the different components of the GBT-FPGA.

Index	Name	Description
S00	LOOPBACK	Select internal loopback inside the transceiver ('1'), or an external loopback via cabling ('0').
S01	GENERAL RESET	Main reset signal of the example design.
S02	MGT TX PLL RESET	Individual reset signal for the MGT pll.
S03	TX RESET	Individual reset signal for the transceiver.
S04	RX RESET	Individual reset signal for the receiver.
S[05..06]	PATTERN SELECT	Selects the pattern that is sent through the transmitter line. It can send a counter value ("1h") that increments by 1, or a static value ("2h").
S07	TX HEADER SELECTOR	Chooses the header of the frame: '0' for idle and '1' for data.
S08	RESET DATA & EXTRA DATA ERROR SEEN FLAGS	Resets **P26**.
S09	RESET RX GBT READY LOST FLAG	Resets **P25**.
S10	TX_FRAMECLK PHASE ALIGNER - MANUAL RESET	Related to the latency optimized version.
S[11..16]	TX_FRAMECLK PHASE ALIGNER - GBT LINK 1 STEPS	Related to the latency optimized version.
S17	TX_FRAMECLK PHASE ALIGNER - ENABLE	Related to the latency optimized version.
S18	TX_FRAMECLK PHASE ALIGNER - TRIGGER	Related to the latency optimized version.
S[19..26]	TX_WORDCLOCK MONITORING - THRESHOLD UP	Related to the latency optimized version.
S[27..34]	TX_WORDCLOCK MONITORING - THRESHOLD LOW	Related to the latency optimized version.
S35	TX_WORDCLOCK MONITORING - TX RESET ENABLE	Related to the latency optimized version.

Table C.1: GBT control signals overview, switches.

Index	Name	Description
P00	LATENCY-OPTIMIZED GBT LINK - TX	Indicates whether the GBT example design is using the latency optimized ('1') version or not ('0').
P01	LATENCY-OPTIMIZED GBT LINK - RX	Indicates whether the GBT example design is using the latency optimized ('1') version or not ('0').
P02	MGT TX PLL LOCKED	Shows the status of the MGT pll, and remains high ('1') if the pll is locked.
P03	TX_FRAMECLK PHASE ALIGNER - PLL LOCKED	Related to the latency optimized version.
P04	TX_FRAMECLK PHASE ALIGNER - PHASE SHIFT DONE	Related to the latency optimized version
P[05..12]	TX_WORDCLOCK MONITORING - STATS	Related to the latency optimized version
P13	TX_WORDCLOCK MONITORING - TX_WORDCLK PHASE OK	Related to the latency optimized version
P14	MGT READY	Asserted high ('1') to show that the MGT transceiver is ready.
P15	RX_WORDCLK READY	Asserted high ('1') to show that the RX_WORDCLK is ready.
P16	RX_FRAMECLK READY	Asserted high ('1') to show that the RX_FRAMECLK is ready.
P17	RX GBT READY	Asserted high ('1') to show that the receiver of the GBT-link is ready.
P[18..23]	RX BITSLIP NUMBER	Related to the latency optimized version, and must remain at "00h" to indicate that the RX and TX are properly aligned.
P24	RX HEADER IS DATA FLAG	Indicates whether the received data has a header of the frame that is idle ('0') or data ('1').
P25	RX GBT READY LOST FLAG	Indicates that the connection has been lost, and remains high until **S09** is asserted high.
P26	RX DATA ERROR SEEN FLAGS	Is asserted high ('1') if the pattern checker detects an indifference in the transmitted and received pattern.
P27	RX EXTRA DATA WIDE-BUS ERROR SEEN FLAG	Same as for **P26**, only related to the *wide-bus* mode.
P28	RX EXTRA DATA GBT8B10B ERROR SEEN FLAG	Same as for **P26**, only related to the *8B10B* mode.
P29	ISSP PLL Locked	Shows the status of the issp pll. This signal does not need to be monitored by the serial interface.

Table C.2: GBT control signals overview, probes.

Appendix D

Clock Control Software Setup

The Cyclone V GT kit is dependent on a number of Quartus related files, including the USB Blaster II device driver, the jtagconfig software and various device libraries included in the Quartus environment. It is therefore important to have the right version of Quartus installed for the Clock Control program to work properly. The version number of the kit (13.0.0.1 at the time of writing) corresponds to the supported version of Quartus, in this case Quartus 13.x. Newer versions of Quartus have not proven to be backwards compatible with the Cyclone V GT kit.

By using Windows, the following steps have been proven to be the best approach to make the Clock Control software work properly. The installed path to Quartus is in this case: `D:\Quartus_13.1`.

D.1 Steps for Configuring Windows to run the Clock Control Software

1. Install Quartus 13.x (includes jtagconfig) together with the Cyclone V device support [35].
2. Set appropriate environment variables. In Windows, this should be set automatically.
   - PATH – `"D:\Quartus_13.1"`
   - QUARTUS_ROOTDIR – `"D:\Quartus_13.1\quartus"`
   - SOPC_KIT_NIOS2 – `"D:\Quartus_13.1\nios2eds"`
3. Connect the Cyclone V board to the PC using USB Blaster II (Refer to the manual for instructions on how to install the USB Blaster II [36]).
4. In Command Prompt (cmd.exe):
   - Navigate to the "board_test_system" folder located inside the Cyclone V

kit.

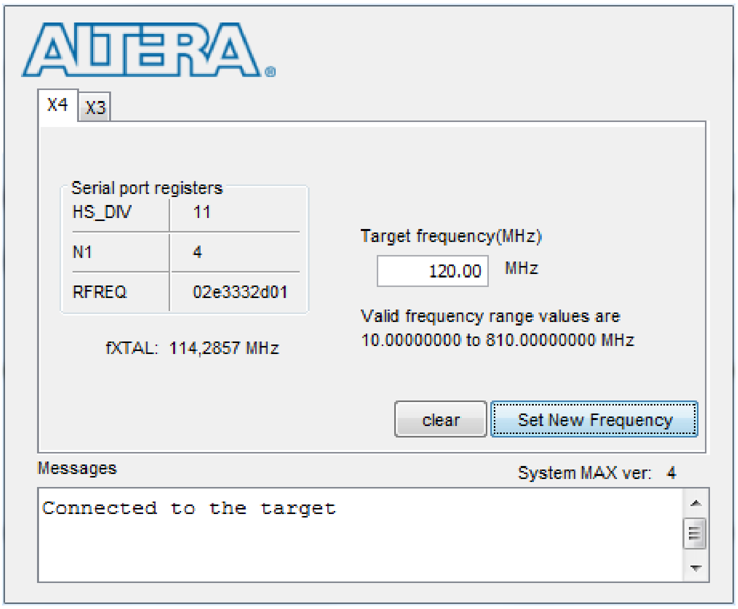
- run ”jtagconfig” and confirm connection with the board. If the Command Prompt cannot find jtagconfig, navigate to `D:\Quartus_13.1\quartus\bin` using a file explorer and manually start jtagconfig.exe from there

- run ”java -Djava.library.path=`"D:\Quartus\_13.1\bin"` -jar clk_cont.jar”. The library path is to ensure that the Java environment have access to the appropriate Quartus libraries it needs to connect with the board.

If done correctly, the Clock Control software will start up and display ”Connected to the target” in the message window. The default output frequency of the $Si570$ oscillator is 100 MHz. The output frequency is calculated using the following equation:

$$f_{out} = \frac{f_{XTAL} \times RFREQ}{HSDIV \times N1} \tag{D.1}$$

, where f_{XTAL} is a fixed frequency of $114,2857$ MHz; RFREQ is a floating point 38-bit word; and HSDIV and N1 is the output dividers [37]. The parameters are determined by the Clock Control software based on the user typed frequency. The parameters are then sent serially via the USB Blaster II to the $Si570$ chip, where the above formula is used to set the internal registers for the new frequency. Figure D.1 shows the Clock Control software after a new frequency of 120 MHz is set.



Serial port registers

HS_DIV	11
N1	4
RFREQ	02e3332d01

fXTAL: 114,2857 MHz

Target frequency(MHz)
120.00 MHz

Valid frequency range values are
10.00000000 to 810.00000000 MHz

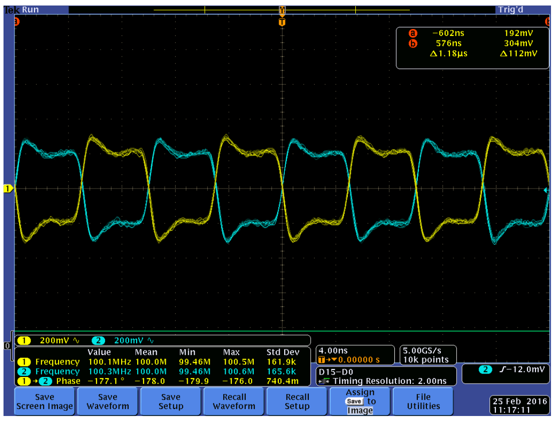
Messages System MAX ver: 4

```
Connected to the target
```

Figure D.1: Clock Control software by Altera used to program the $Si570$.

The $Si570$ is volatile, meaning that the output frequency is reset back to 100 MHz if power is lost. The procedure must therefore be repeated every time the FPGA is

powered on.
To confirm correct operation, a quick measurement of the output clock was done using an oscilloscope. Figures D.2 and D.3 shows the output frequency, before and after configuration.

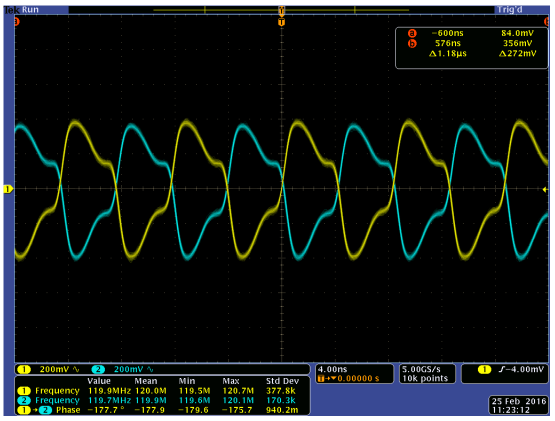


```
a  -602ns     192mV
b   576ns     304mV
   Δ1.18µs   Δ112mV

1  200mV   2  200mV
                 Value     Mean    Min     Max     Std Dev
1 Frequency      100.1MHz  100.0M  99.46M  100.5M  161.9k
2 Frequency      100.3MHz  100.0M  99.46M  100.6M  165.6k
1→2 Phase        -177.1 °  -178.0  -179.9  -176.0  740.4m

4.00ns  T→0.00000 s    5.00GS/s  10k points    2  -12.0mV
D15-D0  Timing Resolution: 2.00ns
25 Feb 2016 11:17:11
```

Figure D.2: *Si*570 Before configuration: 100 MHz.



```
a  -600ns     84.0mV
b   576ns     356mV
   Δ1.18µs   Δ272mV

1  200mV   2  200mV
                 Value     Mean    Min     Max     Std Dev
1 Frequency      119.9MHz  120.0M  119.5M  120.7M  377.8k
2 Frequency      119.7MHz  119.9M  119.6M  120.1M  170.3k
1→2 Phase        -177.7 °  -177.9  -179.6  -175.7  940.2m

4.00ns  T→0.00000 s    5.00GS/s  10k points    1  -4.00mV
25 Feb 2016 11:23:12
```

Figure D.3: *Si*570 After configuration: 120 MHz.

Appendix E

Soldering the Ground Pads Underneath the HSMC Contact

The HSMC-connector has several ground pads underneath the connector itself, making it impossible to reach when soldering by hand. The solution was to use a solder oven with solder paste on the pads. The solder paste was applied on the pads with the help of the dispenser module on the *Martin Rework Station*, which was available in the lab. The dispenser module forces the solder paste out of the syringe using pressurized air. By pressing a foot pedal connected to the dispenser, you apply a controlled air pulse that pushes on the piston of the solder paste syringe. The force of the air pulse can be adjusted using the dispenser GUI. For the ground pads, an air flow of 2.50 CMM (Cubic Meter per Minute) was suitable.

The solder oven that was used had the option to set up a *Ramp-Soak-Spike* type thermal profile. Samtec, the manufacturer for the HSMC contact, recommends a maximum peak temperature of no more than 260 $°C$ with no more than 30 seconds above 255$°C$. The thermal profile setup is based on the profile recommended by EFD, the manufacturer for the particular solder paste used (see figure E.1). After some trial and error with a dummy-PCB, the oven was set to a activation temperature of 150$°C$ for 60 *seconds*, and then a reflow temperature of 220$°C$ with a climb-time of 120 *seconds*.

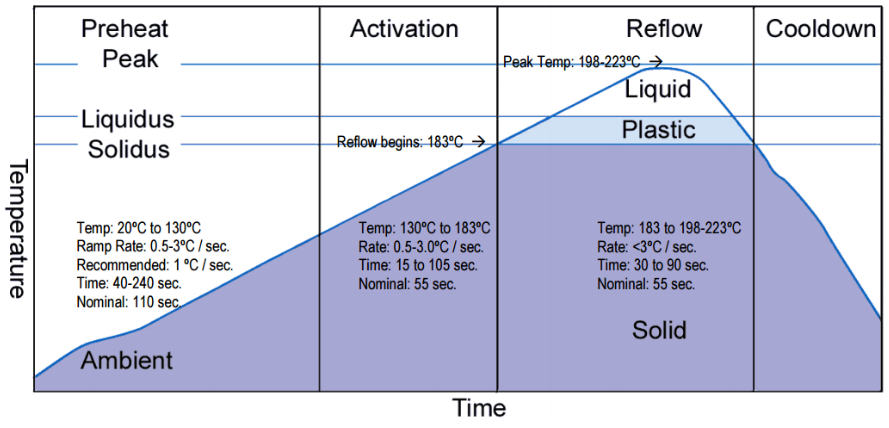


Figure E.1: EFD reflow thermal profile recommendations [38].

Appendix F

Code Files

F.1 C Code

Table F.1 lists the different C files that make up the GBT control interface module and the send/receive module. For more information about the actual functions described in the files, see appendix B. The files are available at https://github.com/aoe033/GBT_control_interface.

File	Description
cmds.h/cmds.c	Contain functions and definitions related to analysis, treatment and excecution of user input.
main.h	Contains global variables and structures, such as the signal structures and rs232 related variables.
main.c	Main program. There are one main.c for each module.
rs232.h/rs232.c	Contains functions related to rs232 communication (See B.1)
signals.h/signals.c	Contains the signal structure and related functions (See B.3).
timer.h/timer.c	Contains functions related to the program timer. (See B.2)

Table F.1: List of C files used in the GBT Control Interface module and the Send/Receive module

F.2 VHDL Code

Table F.2 lists the different VHDL files used in this thesis. For more information, see section 4.5.2. The files are available at https://github.com/aoe033/GBT_control_interface.

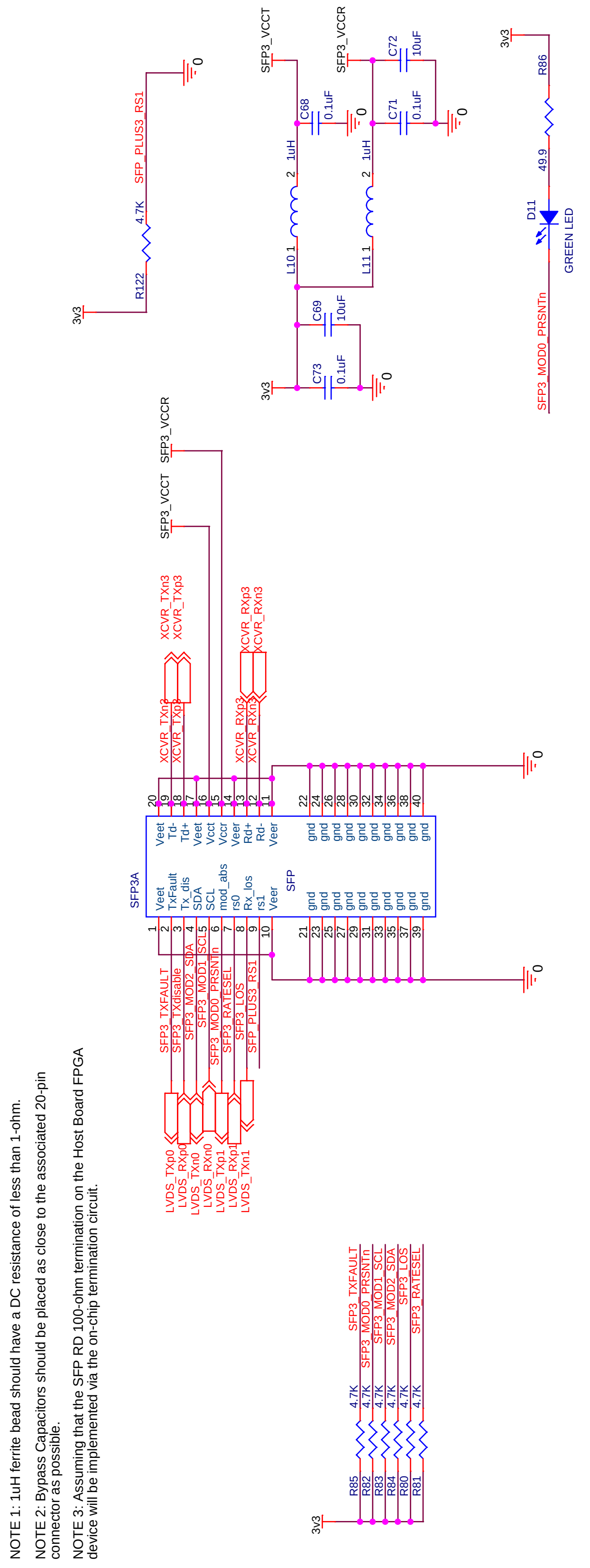
File	Description
baudGen.vhd	Baud generator that generates a tick signal every 16. clock cycles.
fifo_buffer_chu.vhd	Describes the fifo-buffers used in the uart unit.
uart.vhd	Combines fifo_buffer_chu.vhd, baudGen.vhd, uart_rx.vhd and uart_tx.vhd into an uart unit.
uart_decoder.vhd	Reads bytes from the uart and manipulates the gbt registers (not done).
uart_gbt_pkg.vhd	Defines constants (request-bytes and uart data bit width) and records.
uart_rx.vhd	Uart receiver.
uart_tx.vhd	Uart transmitter.
uart_top.vhd	Combines all components together into one entity.

Table F.2: List of VHDL files used in this thesis.

Appendix G

Schematic and PCB Layout

The schematic of the HSMC-to-VLDB PCB is shown in figures G.1, G.2 and G.3. The PCB layout is shown in figures G.4 and G.5. Notes: Do not solder $R51 - R52, R91 - R92, R95 - R97, R99 - R101$.


NOTE 1: 1uH ferrite bead should have a DC resistance of less than 1-ohm.

NOTE 2: Bypass Capacitors should be placed as close to the associated 20-pin connector as possible.

NOTE 3: Assuming that the SFP RD 100-ohm termination on the Host Board FPGA device will be implemented via the on-chip termination circuit.

Figure G.1: PCB Schematic, SFP connector.

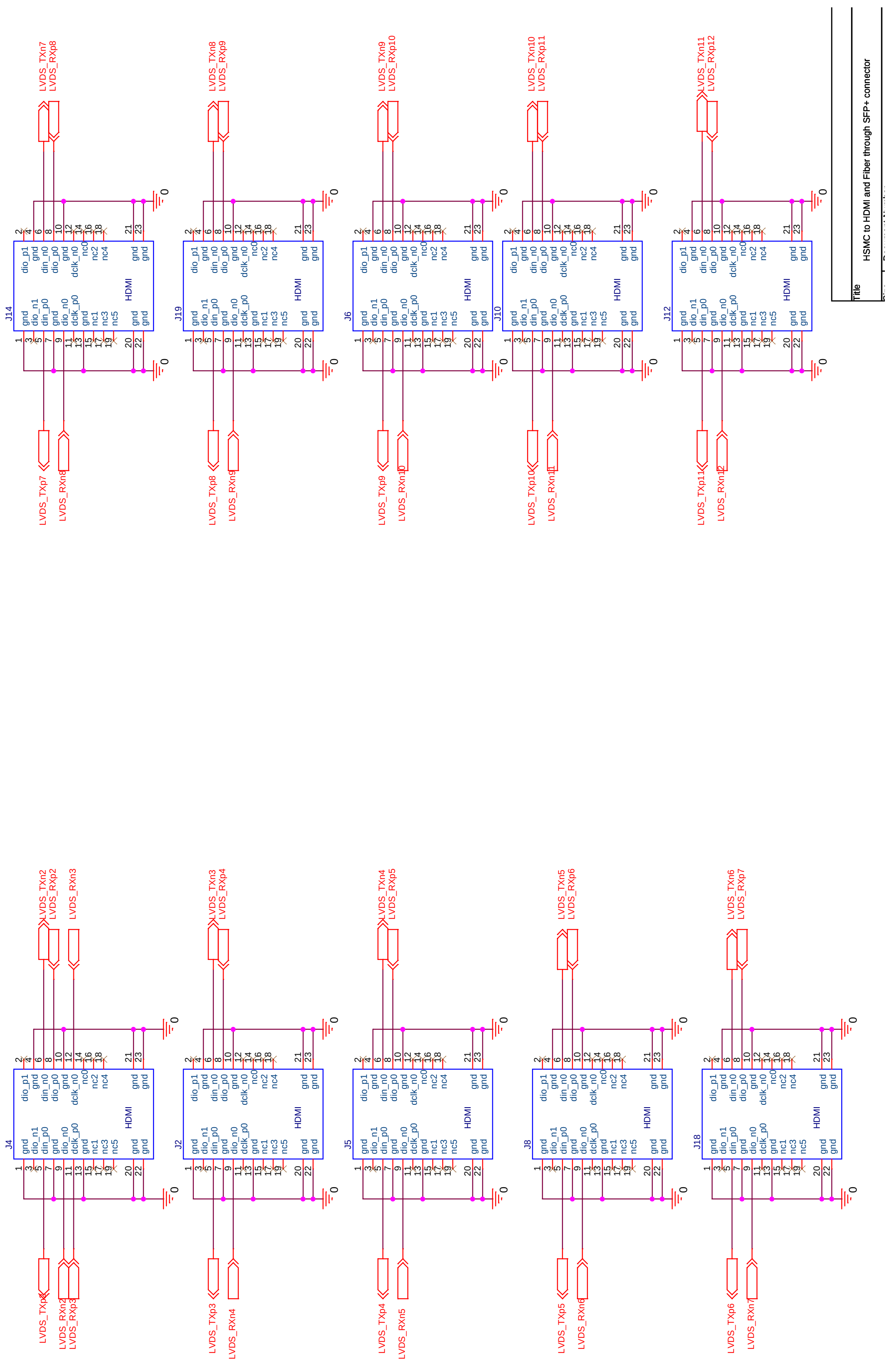


Figure G.2: PCB Schematic, HDMI connectors.

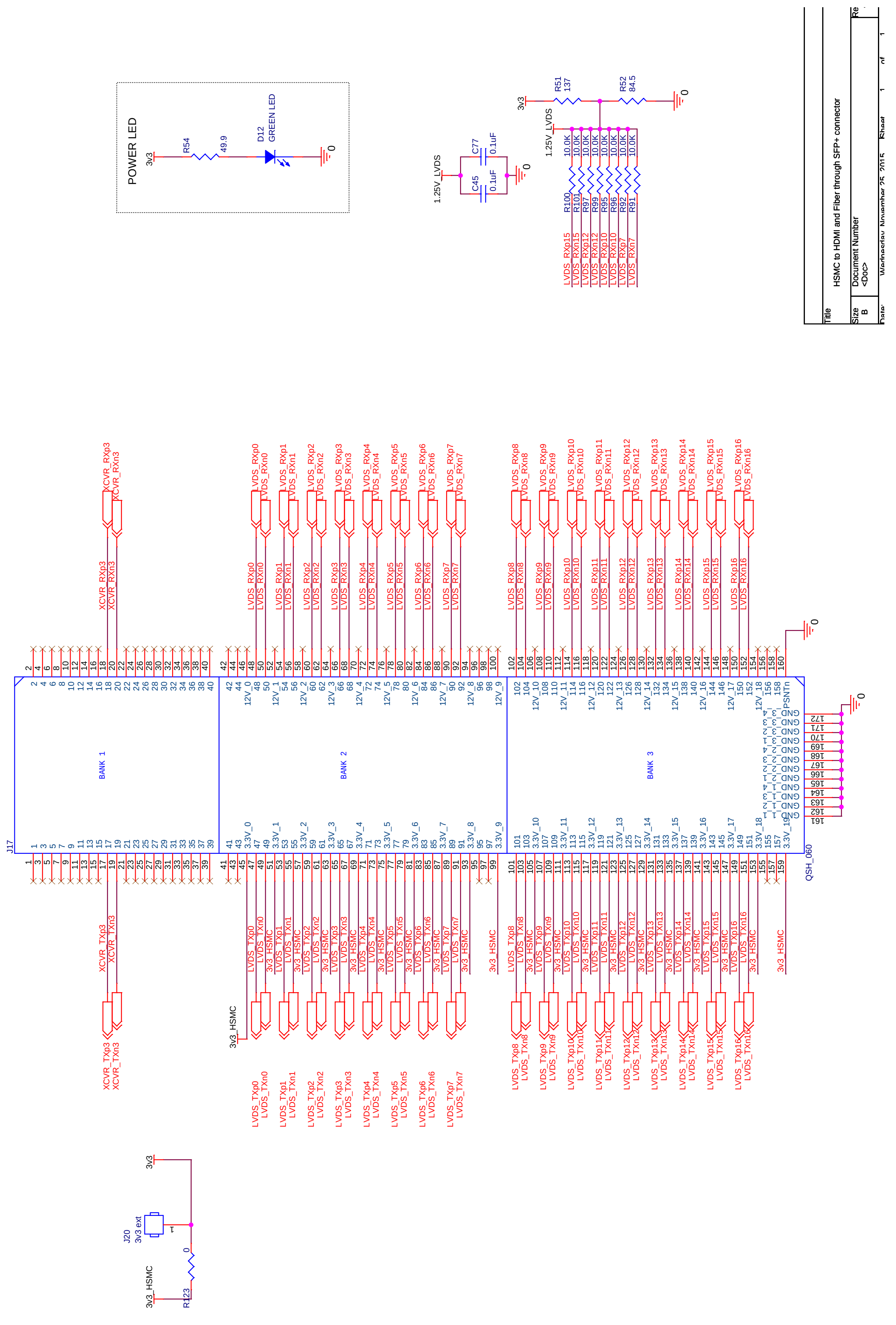
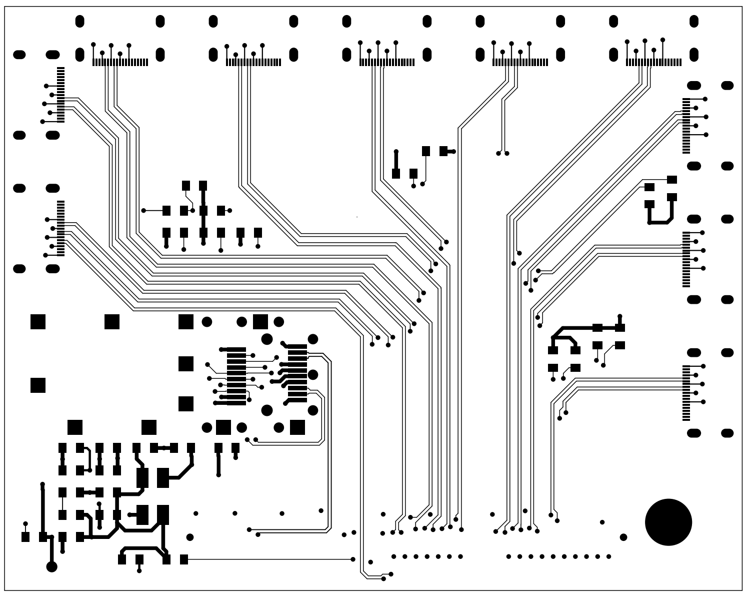
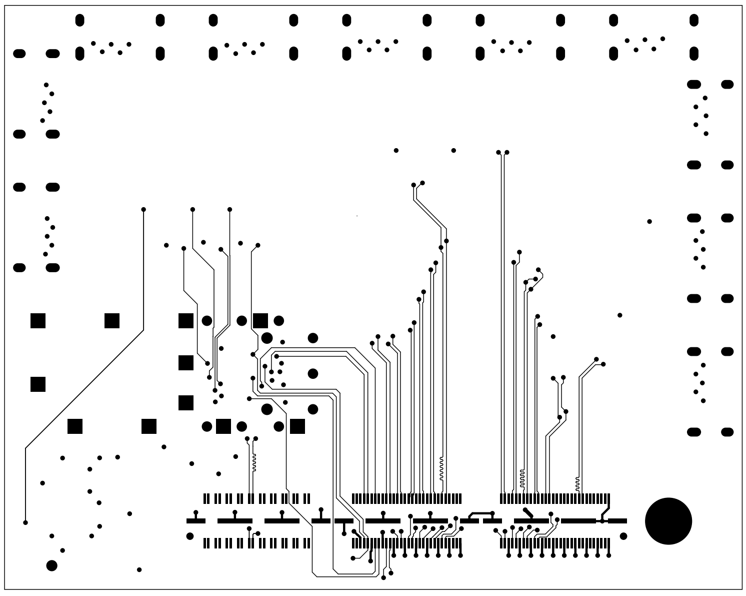


Figure G.3: PCB Schematic, HSMC connector.

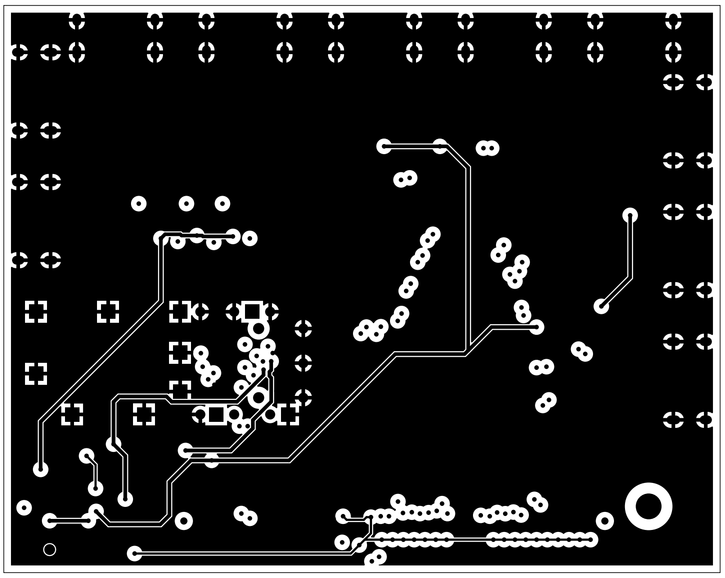


(a) Copper top.

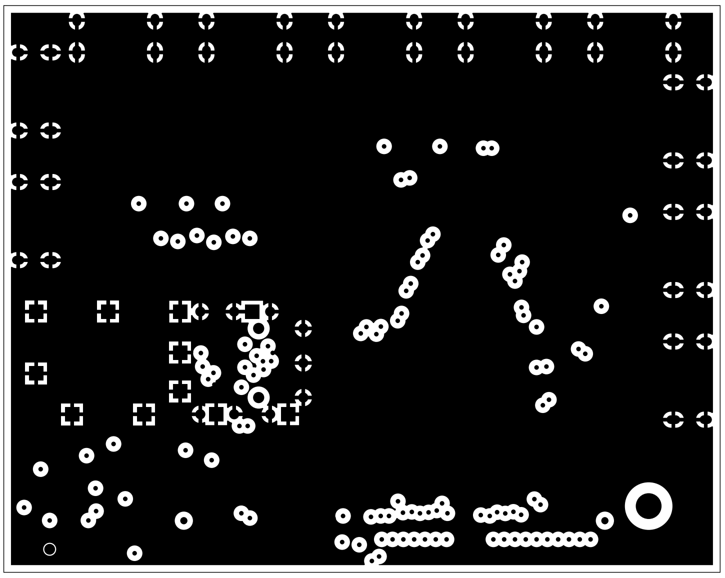


(b) Copper bottom.

Figure G.4: PCB Layout, top and bottom layers.



(a) Inner layer: voltage.



(b) Inner layer: ground.

Figure G.5: PCB Layout, inner layers.

List of Figures

List of Tables

Bibliography

[1] CERN Webpages. http://home.cern/topics/large-hadron-collider, 2016.

[2] ATLAS Webpages. http://home.cern/about/experiments/atlas, 2016.

[3] ALICE Webpages. http://aliceinfo.cern.ch/Public/Welcome.html, 2016.

[4] CMS Webpages. http://cms.web.cern.ch/, 2016.

[5] LHCb Webpages. http://lhcb-public.web.cern.ch/lhcb-public/, 2016.

[6] LHC Overview. http://orbiterchspacenews.blogspot.no/2013/08/cern-magnet-movers-replacing-last-lhc.html, 2013.

[7] Alessandro Caratelli. GBT-SCA Powerpoint Presentation. https://indico.cern.ch/event/291722/contributions/668241/attachments/545329/751766/GBT-SCA_LHCb_Upgrade_meeting_12_06_2014.pdf, 2014.

[8] K. Wyllie P. Moreira, J. Christiansen. GBTx Manual v1.7. https://cms-docdb.cern.ch/cgi-bin/PublicDocDB/RetrieveFile?docid=3857&version=3&filename=gbtxSpecsV1.7.pdf, 2011.

[9] Sophie Baron and Manoel Barros Marin. GBT-FPGA User Guide. https://espace.cern.ch/GBT-Project/GBT-FPGA, 2015. Accessed: 29/02/2016.

[10] A. Caratelli, S. Bonacini, K. Kloukinas, A. Marchioro, P. Moreira, R. De Oliveira and C. Paillard. The GBT-SCA, a radiation tolerant ASIC for detector control applications in SLHC experiments. http://iopscience.iop.org/article/10.1088/1748-0221/10/03/C03034/pdf, 2015.

[11] S. Baron, J. P. Cachemiche, F. Marin, P. Moreira, C. Soos. The GBT Project. https://espace.cern.ch/GBT-Project, 2010.

[12] Cyclone V Device Overview. https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_51001.pdf, 2015. Accessed: 13/05/2016.

[13] VLDB Poster. https://indico.cern.ch/event/468486/contributions/1144349/attachments/1247041/1837036/Poster_ACES2016_Raul_SB.PDF, 2016.

[14] S. Bonacini, K. Kloukinas, P. Moreira. e-link: A Radiation-Hard Low-Power Electrical Link for Chip-to-Chip Communication. http://cds.cern.ch/record/1235849/files/p422.pdf, 2009.

[15] Douglas Brooks. Differential Signals, Rules To Live By. http://www.ieee.li/pdf/essay/differential_signals.pdf, 2001. Accessed: 07/09/2015.

[16] LVDS Owners Manual, Including High-Speed CML And Signal Conditioning. http://www.ti.com/lit/ml/snla187/snla187.pdf?keyMatch=lvds&tisearch=Search-EN-TechDocs, 2008. Accessed: 08/09/2015.

[17] Cyclone V Kit Download Page. https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-gt.html, 2016. Accessed: 24/02/2016.

[18] SFP HSMC Schematic (hsmc_sfp_revb.pdf). http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=342&PartNo=3, 2013. Accessed: 03/12/2015.

[19] High Speed Mezzanine Card (HSMC) Specification. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ds/hsmc_spec.pdf, 2009. Accessed: 10/12/2015.

[20] Neil H. E. Weste and David Money Harris. *CMOS VLSI Design, A Circuit And Systems Perspective.* Addison-Wesley, Pearson, fourth edition, 2011.

[21] Transmission lines and testing at operating frequency. http://www.polarinstruments.com/support/cits/AP155.html, 2015. Accessed: 03/12/2015.

[22] Douglas Brooks. Differential Impedance, Whats The Difference. http://www.ultracad.com/articles/diff_z.pdf, 1998. Accessed: 03/12/2015.

[23] Elprint Kapabilitet. http://www.elprint.dk/kapabilitet/, 2015. Accessed: 01/12/2015.

[24] Serial To Ethernet Adapter. http://www.digi.com/products/serial-servers/serial-device-servers/portserverts#overview, 2016. Accessed: 05/05/2016.

[25] Using Terminals With DE-Series Boards. ftp://ftp.altera.com/up/pub/Altera_Material/14.0/Tutorials/Using_Terminals.pdf, 2014. Accessed: 16/02/2016.

[26] Pong P. Chu. *FPGA Prototyping By VHDL Examples.* Wiley-Interscience, 2008. Accessed: 12/02/2016.

[27] Arild Velure. Uart2Bus vhdl module, 2010. Design used as a reference to the serial communication part on the FPGA side. Private communication.

[28] J.O. Hamblen, T.S. Hall and M.D. Furman. *Rapid Prototyping Of Digital Systems.* Springer, 2008. Accessed: 12/02/2016.

[29] GBT FPGA Video Tutorials. https://espace.cern.ch/GBT-Project/GBT-FPGA, 2016. Accessed: 29/02/2016.

[30] Silabs Clock Software Development Tools. http://www.silabs.com/products/clocksoscillators/Pages/Timing-Software-Development-Tools.aspx, 2016. Accessed: 23/05/2016.

[31] Si5338-EVB Documentation. https://www.silabs.com/Support%20Documents/TechnicalDocs/Si5338-EVB.pdf, 2011. Accessed: 23/05/2016.

[32] Bitvis Utility Library homepage. http://bitvis.no/products/bitvis-utility-library/, 2016. Library used to create the uart testbench. Accessed: 07/04/2016.

[33] GPL version 3. http://www.gnu.org/licenses/gpl-3.0.txt, 2007.

[34] Zeyd M. Ben-Halim, Eric S. Raymond, Thomas E. Dickey. ncurses documentation site. http://invisible-island.net/ncurses/man/ncurses.3x.html, 2016. Accessed: 17/05/2016.

[35] Quartus 13.1 Download Page. https://dl.altera.com/13.1/?edition=web, 2013. Accessed: 24/02/2016.

[36] USB Blaster II Install Manual. https://www.altera.com/support/support-resources/download/drivers/usb-blaster/dri-usb-blaster-vista.html, 2016. Accessed: 24/02/2016.

[37] Si570/Si571 Datasheet. https://www.silabs.com/Support%20Documents/TechnicalDocs/si570.pdf, 2014. Accessed: 29/02/2016.

[38] 6-SN63-221 Product Specification. http://www.krepro.no/dispenseringspasta-sn62pb36ag2.html, 2009. Accessed: 20/01/2016.