Department of Physics and Technology
University of Bergen

Master Thesis

Firmware Design for the GBT Common Readout Unit

Anders Østevik

May 19, 2016

Abstract

Acknowledgements

Contents

Chapter 1

Introduction

The future upgrade of the Large Hadron Collider (LHC) accelerator, the Super Large Hadron Collider (SLHC), will increase the beam luminosity leading to a corresponding growth of the amount of data to be treated by the data acquisition systems. This will thus require high rate data links and high radiation tolerant Application Specific Integrated Circuits (ASICs).

To address these needs, the Gigabit Transceiver (GBT) architecture and transmission protocol was developed to provide the simultaneous transfer of readout data, timing and trigger signals in addition to slow control and monitoring data.

The GBT system can be described in two parts: First part of the system consists of radiation hard GBT ASICs that will act as detectors and will thus be located in the radioactive zone. These ASICs are used to implement bidirectional multipurpose $4.8\ \mathrm{Gbit/s}$ optical links for the high-energy physics experiments.
The last part of the system is located in the counting room and consists of the Common Readout Unit (CRU) that will provide an interface between the detector ASICs and an online computer-farm. 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.

This thesis will mainly focus on the CRU software interface on the PC side, and the physical connection between the CRU and the Versatile Link (VLDB) card, where the radiation hard ASIC, the GBTx, is located.

1.1 Field Programmable Gate Array

A Field-Programmable Gate Array (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, 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 [19].

The FPGA (section 1.1.1) 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, which then programs the SRAM when powered on).

1.1.1 Altera's Cyclone V GT

The FPGA used in this thesis is the Cyclone V GT by Altera. GT indicates that the FPGA has transceivers that supports speeds up to $6\ \mathrm{Gbit/s}$ [8]. It was chosen for this thesis 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 $4.8\ \mathrm{Gbit/s}$. Originally, a Terasic Cyclone V GX development board was intended to be used for this thesis. The Terasic board has advantages over the Cyclone V GT board in terms of communication with the outside world such as on-board Usb-to-Uart (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 $3.125\ \mathrm{Gbit/s}$ [8]. Because of this, the more powerful Cyclone V GT FPGA development board was ordered from the Altera web-pages, replacing the Terasic.

1.2 Hardware Description Language

The most common way to program an FPGA is by Hardware Description Language (HDL), with the major ones being SystemVerilog and VHDL. HDL consists 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 [19]. The hardware described in this thesis is done using VHDL, a

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

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;
```

1.3 Cyclone V transceiver Technology

To be able to send serial data in the gigahertz domain, a high-speed transceiver is required. The Cyclone V GT-series FPGAs supports a number of transceiver technologies through the High-Speed Mezzanine Card (HSMC) physical interface that can reach speeds up to 6.144 Gbit/s. This section gives a general description of some of these protocols.

1.3.1 Differential Signals

Common for all protocols described 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 involves a conductor pair with 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 *EMI*. In addition, placing the signals as close as possible to one another will give benefits in terms of common noise rejection [17].

When done 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 signal applications. The disadvantage comes in an increase in pin count and space required, since differential signals consists of two wires instead of one [17].

1.3.2 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 $350\ \mathrm{mV}$ and good noise immunity. LVDS can deliver data rates up to $3.125\ \mathrm{Gbit/s}$ [2].

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 $840\ \mathrm{Mbit/s}$, with support for serialization and deserialization through internal logic. [8]

1.3.3 Current-Mode Logic

For data rates that exceeds $3.125\ \mathrm{Gbit/s}$, Current-Mode Logic (CML) signaling is preferred. This is due to the fact that certain communication standards such as PCIe, SATA and HDMI, shares consistency with CML in signal amplitude and reference to Vcc. CML can reach a data rate in excess of $10\ \mathrm{Gbit/s}$, but has a higher power consumption than LVDS, with a voltage swing of approximately $800\ \mathrm{mV}$ [2].

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 $5.0\ \mathrm{Gbit/s}$, just over the $4.8\ \mathrm{Gbit/s}$ range required by the GBT MGT. [12]

1.3.4 Phase-Locked Loops

Chapter 2

The Gigabit Transceiver

History, mgt, Short about the electronic components, gbt-sca, gbtx

2.1 Encoding modes

The ”GBT-Frame” mode, which is Reed-Solomon Based; the ”8b10b” mode; and the ”Wide-Bus” mode which is without encoding.

2.2 GBT-FPGA Core

The following sections describe the different components that makes up the GBT-FPGA Core. The information was obtained by reading the GBT-FPGA User Guide [21].

2.2.1 GBT Bank

The GBT Bank is defined as the top module of the GBT-FPGA Core. It integrates up to four GBT Links and contains the ports required to operate the GBT Links.

2.2.2 GBT Link

The GBT Link is the actual channel of the link. It is composed of three components: GBT Tx, GBT Rx, and the MGT. The following subsections gives a brief description of these components.

GBT Tx

The GBT Tx component is responsible for scrambling and encoding data before transmitting it through the MGT.

GBT Rx

The GBT Rx component is responsible for receiving, decoding and de-scrambling the data through the MGT.

Multi-Gigabit Transceiver

The MGT is responsible for the transmitting, receiving, serialization and de-serialization of the GBT data. It is divided into a transmitter and a receiver part.
The transmitter contains a PISO with two input clocks; one for parallel data and one for serial data. It shifts in $40\ \mathrm{bit}$ words from the GBT Tx with a reference clock of $120\ \mathrm{MHz}$, serializes the data and sends it out with the help of a dedicated Tx Phase-Locked Loop (PLL) that generates a serial clock of $2400\mathrm{MHz}$.
The receiver contains a Clock & Data Recovery (CDR) block, a SIPO, a RXRECCLK Phase Aligner block and a Barrel-shifter.

Chapter 3

HSMC-to-VLDB PCB design

In order to interface the radiation hard GBTx chip (see chapter 2), a form of electrical connection between the FPGA and the VLDB is required. The VLDB has twenty additional HDMI female connectors which connects to LVDS transceivers for high-speed communication with the outside world. The FPGA board that is used for this thesis has two HSMC connectors with port A containing pins that connects to LVDS transceivers. In addition, the VLDB connection also requires a radiation-hard optical link, i.e support for high-speed SFP fiber-optic cabling. For this, the HSMC port A connector has also available pins that connects to Transceiver (XCVR) transceivers for transmission speeds up to $6.144\ \mathrm{Gbit/s}$. The resulting PCB has 10 individual High-Definition Multimedia Interface (HDMI) connector with each having one receiver and transmitter. Because of limited I/O, only one HDMI contains a receiver reserved for input clock from the VLDB. This chapter explains the process of making such a Printed Circuit Board (PCB), from the discussion of different design approaches to the theory behind designing a high speed PCB.

3.1 Specification

3.2 Design discussion

The design discussion and planning of the PCB was done together with Ph. D. 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 the SFP-board with HDMI cables to the VLDB. This design was quickly scratched 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 one concerning the XCVR-protocol, see [5]), 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.

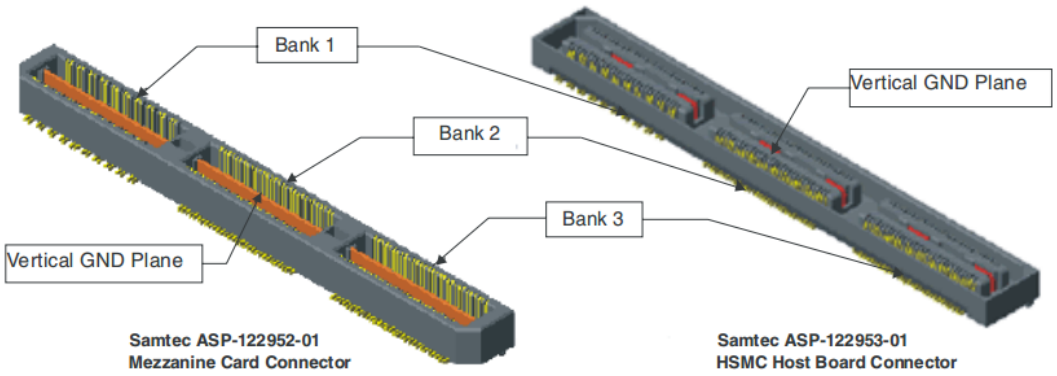

Figure 3.1: Male (ASP-122952) and female (ASP-122953) HSMC-connectors. The male type is to be connected at the bottom of the HSMC-to-VLDB PCB [3, Figure 2-1].

3.3 High Speed PCB Design

When transmitting signals through a conductor at high frequencies, the conductor no longer acts as an ideal wire. The signal voltage stops acting instantaneous across the conducting path, and factors like impedance mismatch and reflections becomes important for the quality of the signal to remain the same through the whole transmission. 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.

3.3.1 Transmission lines

When signal rise/fall times becomes comparable with the propagation delay of the conductor, the signal no longer acts instantaneous. The conductor becomes what is known as a transmission line, with the signals voltage and current acting like waves propagation through the conductor. The general rule 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 [19].

For an LVDS-protocol, with signal speeds reaching up to $3.125\ \mathrm{Gbit/s}$, the trace becomes a transmission line if the signal propagation time along the trace is less than 1/4 of the signal period, i.e $80\mathrm{ps}$.

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 epoxy material FR4, often used as a material to separate the copper layers on the PCBs. ϵ_r has a value of around 4.05 @ $3\ \mathrm{GHz}$ [10]. A signal thus propagates through the conductor at a velocity of approximately $15\ \mathrm{mm/ns}$ [19, example 13.7].

Thus, by assuming a constant transmitting frequency of $3.125\ \mathrm{Gbit/s}$, a conductor becomes a transmission line if the length of the conductor stretches longer than $12\ \mathrm{mm}$ between 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.

3.3.2 Reflections and characteristic impedance

Since the signals no longer acts instantaneous, 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 [19].

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.

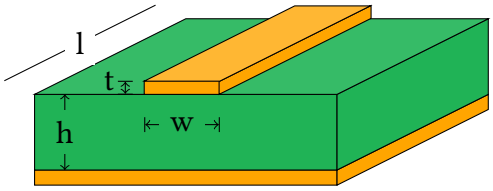

Figure 3.2: Cross-section of a single-ended microstrip with the copper trace on top, followed by a dielectric layer and a ground plane. h is the thickness of the dielectric, t is the copper thickness, l is the microstrip length, and w is the copper width.

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 [19].

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 [19].

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 [16].

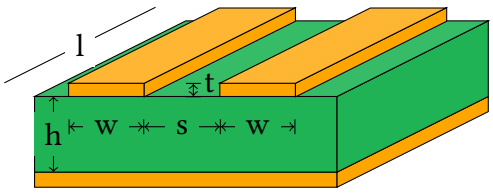

Figure 3.3: Cross-section of a corresponding differential microstrip. s is the spacing between the strips.

3.3.3 Routing

As mentioned in section 1.3.1, 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 crosstalk 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 noise rejection [19]. 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 $100\ \Omega$ (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, but this was not a critical stage.

3.4 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 [9]. 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 $50\ \Omega$, and a differential impedance of $100\ \Omega$. This was derived from the fact that the HDMI-cables were specified to have a differential impedance of approximately $100\ \Omega$, and that the individual transceiver lines from the FPGA has a characteristic impedance of $50\ \Omega$, with a selectable termination resistance at the receiving end of $100\ \Omega$, 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 pre-defined stack-up has a copper thickness of $18\ \mu\text{m}$ on the two outer layers following an FR4 thickness of $65\ \mu\text{m}$ between the outer layers and the power planes, with the power planes having a thickness of $35\ \mu\text{m}$. 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 $100\ \mu\text{m}$, 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 $51.3\ \Omega$ @ $3\ \text{GHz}$.

With a $300\ \mu\text{m}$ 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 $102\ \Omega$ @ $3\ \text{GHz}$.

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

3.5 Soldering process

All the components on the PCB was soldered by hand, with the exception of the ground pads underneath the HSMC, which had to be soldered using solder paste and a solder oven.

3.5.1 Martin Rework Station

The HSMC-connector has several ground pads underneath the connector itself, making it impossible to reach when soldering by hand. The solution to this was to apply solder paste on the pads with the help the Martin Rework Station dispenser module that 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 Guided User Interface (GUI). For the ground pads, a force of $2.50ccm$ was suitable.

3.5.2 Solder Oven

The oven was set to a warm-up temperature of 150 for 60 seconds, and then a climb temperature of max 220 with a climb-time of 120 seconds.

3.6 PCB faults and compensations

After receiving the PCBs, further inspection revealed that all viases had missing solder mask. This exposes the metal of the vias to the surface of the PCB and can possibly cause connection shorts between the viases and the pads and/or metallic casings when soldering. The reason for the viases not to have a solder mask is because it was missing in the pad file. 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 was later shown 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 (see subsection below). A lesson until next PCB-prototype is to include easily accessible probe points on lines that are relevant for oscilloscope measurement.

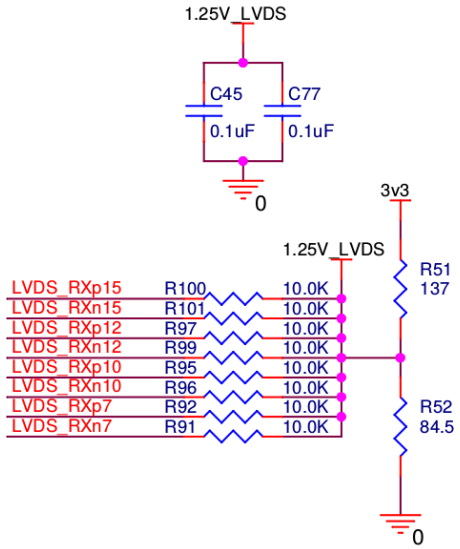

Figure 3.4: Pull-up resistors on some of the receiver lines.

On the current version of the PCB, some of the receiver lines includes a $10\ \mathrm{k}\Omega$ pull-up resistor (R91, R92, R95, R96, R97, R99, R100 and R101 in the schematic). The reason why these are included could possibly be due to a misinterpretation of the SFP-schematic in which the design of the PCB was based on. This is concluded because two of the pull-up resistors (R100 and R101 in the schematic) are connected to receiver signals that are only used in the SFP-design, but not in this design. The pull-up resistor pads were therefore not soldered on the board, and the lines were left open.

Chapter 4

PC to CRU serial 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. Because of this, the speed requirements of the link is not a crucial matter.
Figure 4.1 illustrates the different blocks that makes up the serial interface: The interface on the Personal Computer (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 then gets translated into byte-codes and sent out via the transmitter of the COM port to the Universal Asynchronous Receiver/Transmitter (UART) on the fpga side. Here, the bytes are stored in a buffer before they get interpreted by a decoder. Depending on the user-request, the decoder can write to a GBT-switch or read from a GBT-probe.

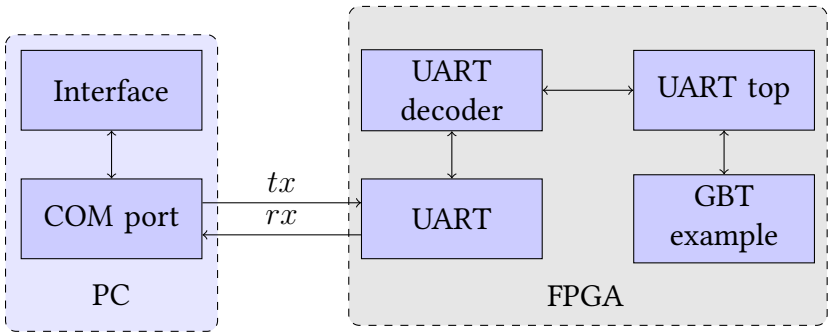

Figure 4.1: Simplified diagram over the PC to CRU serial interface.

4.1 ISSP and the GBT control signals

The Quartus-bound In-System-Source-And-Probe (ISSP) Editor offers a way for the user to control and monitor the control signals of the GBT example design. Table 4.2 and 4.1 gives a brief description of these signals. The information was obtained from the GBT FPGA video tutorials [13].

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 4.1: 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 4.2: GBT control signals overview, probes.

4.2 Readily available standards

The FPGA board used in this thesis (section 1.1.1) has (although sparse) 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 for 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 [14]; It would just be easier to send serial data directly from the FPGA-pins using an USB-to-RS232 adapter-cable (see section 4.3). 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.

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 [7]: the Nios II also occupies a large portion of the FPGA.

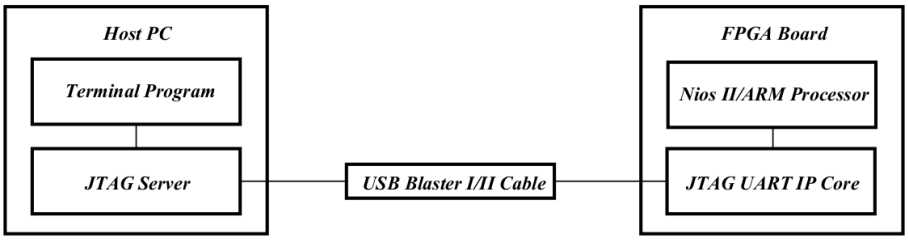

Figure 4.2: JTAG UART communication link between the host PC and the FPGA board [7, Figure 1].

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.

4.3 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 board), or through a prototype area or header.[1]

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.[2] 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

[1]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.

[2]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.

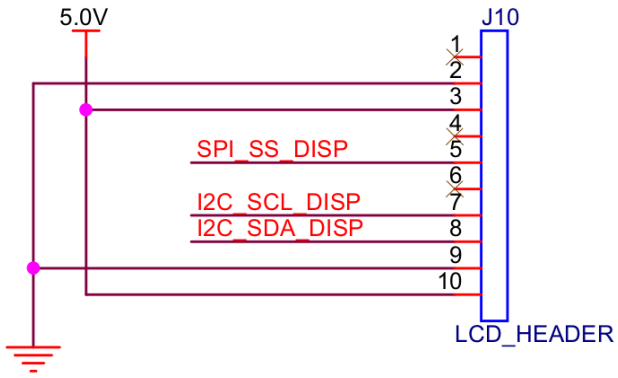

Figure 4.3: J10 header from the Cyclone V GT board schematic. The SCL and SDA signals can be used as single ended transmitter and receiver signals.

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 GX board (see section 1.1.1), 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.

4.4 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.

4.5 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). It is supported by most Operating Systems (OSs) used today, and can be easily implemented using available C-libraries.

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 (chapter 5) and then on the PC side (chapter 6).

Chapter 5

Hardware design on the FPGA side

5.1 Specification

The implemented hardware logic is to contain the following specifications:

- Communication with the user PC using RS-232.
  - Receive requests sent from the user PC. The requests are reserved byte-codes in which the byte-decoder translates to read or write commands.
  - 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.
- Read or write to the GBT control-register.

5.2 Transmission protocol: 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 theData 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 5.1 demonstrates a typical RS-232 signal.

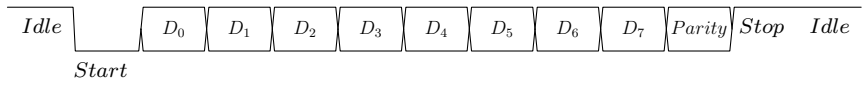

Figure 5.1: Example of an RS-232 signal with 8 data bits, 1 parity bit (Odd, Even or No parity) and 1 stop bit.

5.3 Hardware Components

The below sections gives a brief description of each part of the design.
The UART itself is based on the UART-design by Pong P. Chu, found in chapter 7 from his book *FPGA Prototyping By VHDL Examples* [20]. The Baud Rate Generator is borrowed from the *Uart2Bus* VHDL-design by Arild Velure. The UART-decoder was written in conjunction with the C-program on the PC side, and is the one component which ties the communication together.

The end result forms a design that uses an 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 read or write requests, 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.

5.3.1 UART

For communication with the user PC, a simple UART was implemented into the FPGA.
Simply put, an UART is a circuit that transmits and receives parallel data through a serial line [20]. 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 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-registers connected to the transmitter and receiver to shift the data in or out.

5.3.2 UART 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.

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 [18]. 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.

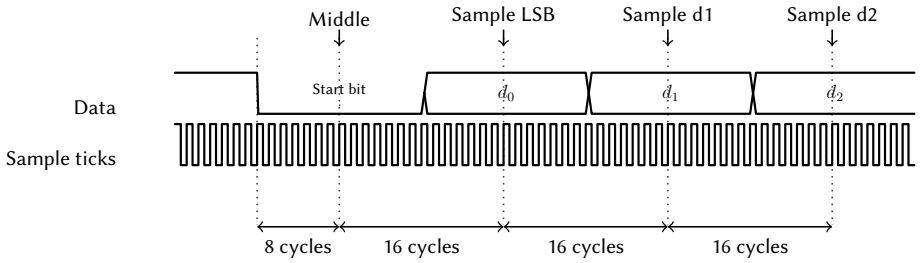

Figure 5.2: UART receive synchronisation and data sampling points with 16 times the sampling rate.

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. For a baud rate of $19200\ \mathrm{bit/s}$ and a clock of $50\ \mathrm{MHz}$, there must be a one-clock-cycle tick once every 163 clock cycle. This is achieved by counting in certain steps given by the formula below:

$$Baud\ frequency = \frac{16 \times baud\ rate}{GCD(clock, 16 \times baud\ rate)} \tag{5.1}$$

, where baud frequency is the count steps, GCD is the greatest common divisor between the global clock and the baud rate times 16 [22].
For a baud rate of $19200\ \mathrm{bit/s}$ and a clock of $50\ \mathrm{MHz}$, the counter must count in steps of 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:

$$Baud\ limit = \frac{clock}{GCD(clock, 16 \times baud\ rate)} - baud\ frequency \tag{5.2}$$

, 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 [22].
For a baud rate of $19200\ \mathrm{bit/s}$ and a clock of $50\ \mathrm{MHz}$, 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.

5.3.3 UART 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) (16 ticks for 1 stop bit).

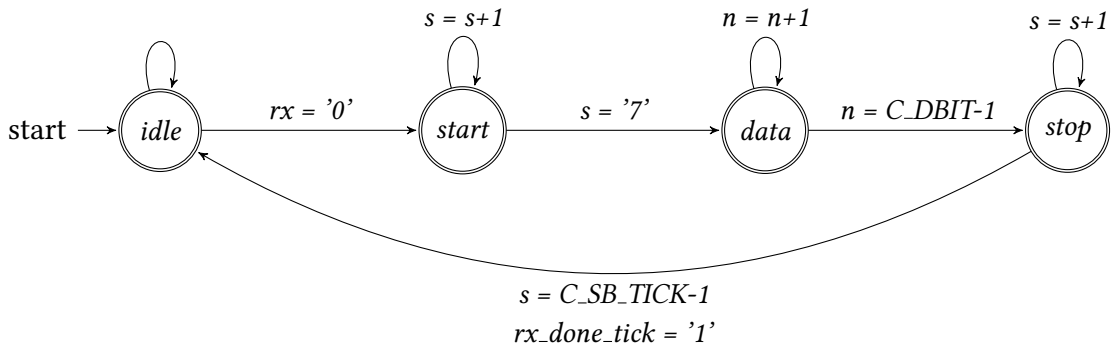

Figure 5.3: UART receiver state machine.

Idle state

Starting with the idle state: given that there is not already a signal being received and sampled (i.e $rx\_done\_tick =' 1'$), the receiver waits for the rx-signal to go low (i.e detecting a start signal). The s-register then resets and the state machine goes to the next state: the start state.

Start state

When in the start state, the ticks generated by the baud rate generator clocks the s-registers and waits for it to count up to 7 (i.e in the middle of the start-bit). It then resets the s- and n-registers and sets the next state to the data state.

Data state

Since the received signal so far is the middle of the start bit, the s-register must count up to 15 before reaching the middle of the data bit so that the b-register can sample the data. Each time the s-register reaches 15, the b-register shifts in the rx by 1 bit while the n-register increments by 1, keeping track of the bit width. When the n-register reaches a count equal to the C_DBIT constant minus 1, the transmission is at its end, and the state machine shift to the last state: the stop state.

Stop state

The stop state uses the sample ticks in conjunction with the s-register to count the stop bit(s) duration by using the $C\_SB\_TICK - 1$ as the upper count limit. When done, the rx_done_tick signal is set to 1 and the state machine shifts back to the idle state. We have now successfully received a serially transmitted byte.

5.3.4 UART 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 input parallel data and a 1 bit tx-register for shifting out the data. Connected to the tx-register is a tx output signal. 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.

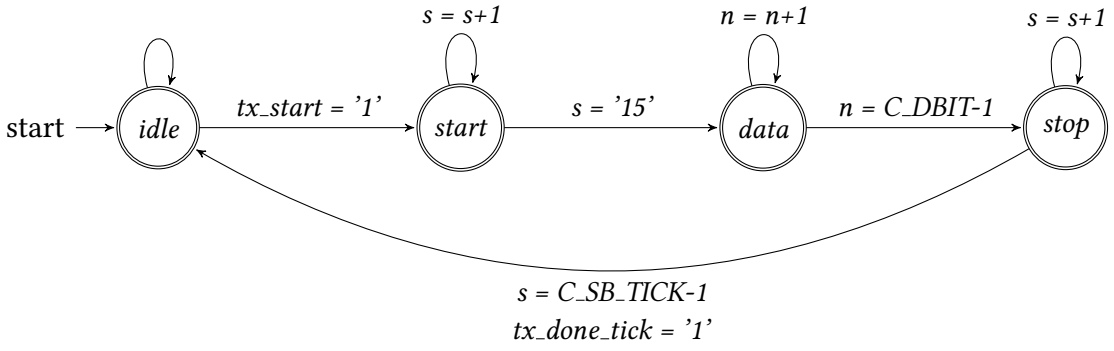

Figure 5.4: UART transmitter state machine, similar to that of the receiver.

Idle state

When in the idle state, the tx-register is held high for idle line and the b-register is connected to the din register for data input. The state machine remains in the idle state until the tx_start

signal changes from low to high. It then shift to the start state to begin the transmission of the data stored in the b-register.

Start state

When in the start state, the tx-register is held low for the start bit to be transmitted out. When the s-register reached the count of 15, the start bit is transmitted and the state machine shifts to the data state to further transmit the actual data bits.

Data state

When in the data state, the tx-register is set to the first bit of the b-register, while the b-register shift its data using a simple right shift operation every 16 clock ticks (s-register reaches 15). To keep track of how many data bits that have been transmitted, the n-register increments by 1 at the same rate as the b-register shifts the data (same as for the receiver). When reaching $C\_DBIT - 1$, the data bits is finished transmitting out and the state machine shifts to the stop state.

Stop state

When in the stop state, the tx-register is held high indicating a stop bit is being transmitted. When the s-register reaches a count equal $C\_SBIT - 1$, the stop bits are finished transmitting. The tx_done_tick signal is set to high and the state machine shifts back to the idle state, ready to transmit the next data byte.

5.3.5 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 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.

5.3.6 UART decoder

The UART decoder is a state machine connected to the other end of the UART receiver- and transmitter FIFOs. It reads out the 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. The request byte can for instance be a read or a write. The following small sections describes the different decoder states.

Idle state

Being initially in the idle state, the decoder is triggered by the rx_empty signal coming from the rx-fifo, which is held low if there is at least one byte stored in the rx-fifo. If rx_empty is low, the decoder sends a read signal (rd_uart), telling the rx-fifo to place the next in-line byte on the read-bus (r_data), and in the same delta-time assigns the read-bus to a new register ($b\_reg(0)$) for temporary storage. The decoder then goes to a read state.

Read1 state

In the read1 state, the first temporary stored byte is analyzed. The byte must have a value that is equal to one of three predefined constants, called requests (See $uart\_gbt\_pkg.vhdl$). This first byte is interpreted as a read- or write-request, sent from the client PC through the serial connection. If the request-byte is equal to a "read" request, a "write 0" request or a "write 1" request, the decoder goes on to a wait state. If the byte is not a request-value, the state machine returns back to idle and reset the registers.

Wait1 state

In the wait1 state, the decoder waits for the next arriving byte after the first one. The wait state does the same as the idle state, i.e waits for rx_empty to become low before reading out the next byte from the rx-fifo. To prevent the state machine from getting stuck in the wait state (if an address byte does not arrive in time), an additional count-register, triggered by the baud generator ticks, counts up to a predefined value, $C\_TIMEOUT$. $C\_TIMEOUT$ is defined to be a reasonable larger value than the pre-estimated time it takes for the next byte to arrive, i.e $16\ ticks/\text{bit} \times 10\ bits$. If rx_empty is not triggered low before the count-register has finished counting, the state machine resets to idle. If the next byte arrives before timeout, the decoder stores it and goes to the final state.

Read2 state

In the read2 state, the value of the first byte, the request-byte, decides if the decoder should perform a read or a write operation on the gbt-register. The value of the last received byte is treated as an address to the data of the gbt-register the client wants to read or write to.

Chapter 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.

To access the serial port on the PC, the software uses free and cross-platform C-libraries (see sections below).

The C programming language was chosen for this task mainly because of previous experience with the language, but also because of desire to learn to use the language more professionally.

6.1 Specification

The program is to contain the following specifications:

- 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.

6.2 Non-standard libraries

The serial communications 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, a library dedicated to hold the GBT control-signal information was written. The following sections gives brief descriptions of each of these libraries and the associated functions used in the serial communications software:

6.2.1 RS232

The RS232 library (*rs232.h* and *rs232.c*) is a cross-platform C-library for sending and receiving bytes from the COM port of a PC. It was written by Teunis van Beelen, and is licensed under the "GPL version 3" licence [1]. The library compiles with GCC on Linux and Mingw-w64 on Windows. By specifying baud rate, the name of the relevant COM port, and the transmission mode (8N1 is standard), the library provides access to the COM port and allows for both reading and writing to it. For more information about the library and functions, visit `http://www.teuniz.net/RS-232/#`.

Associated functions

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

Opens the COM port. The user must specify a COM port 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 port or change the baudrate correctly. The function returns 1 if it does not succeed in opening the COM port.

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

Returns the amount of bytes (in integers) received from the COM port. 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 port. 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 port. To prevent errors, it is important to disable any timers that calls COM port related functions before closing the COM port.

6.2.2 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 [1]. 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/#`.

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 port at the end of the program to prevent errors.

6.2.3 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.

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.

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 6.2.4) 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.

6.2.4 ncurses

The ncurses library is a terminal control library commonly used with Unix-systems. It is ”freely redistributable in source form” [23], 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).

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. In the GBT serial interface, *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 GBT serial interface.

```
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 GBT serial interface.

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.

6.3 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 are intended to be merged together into one program. The below sections describes the inner workings of each of the two modules, and provides flowcharts to further illustrate the behaviour.

6.3.1 Interface module

Figure 6.1 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 6.2.4). 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 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.

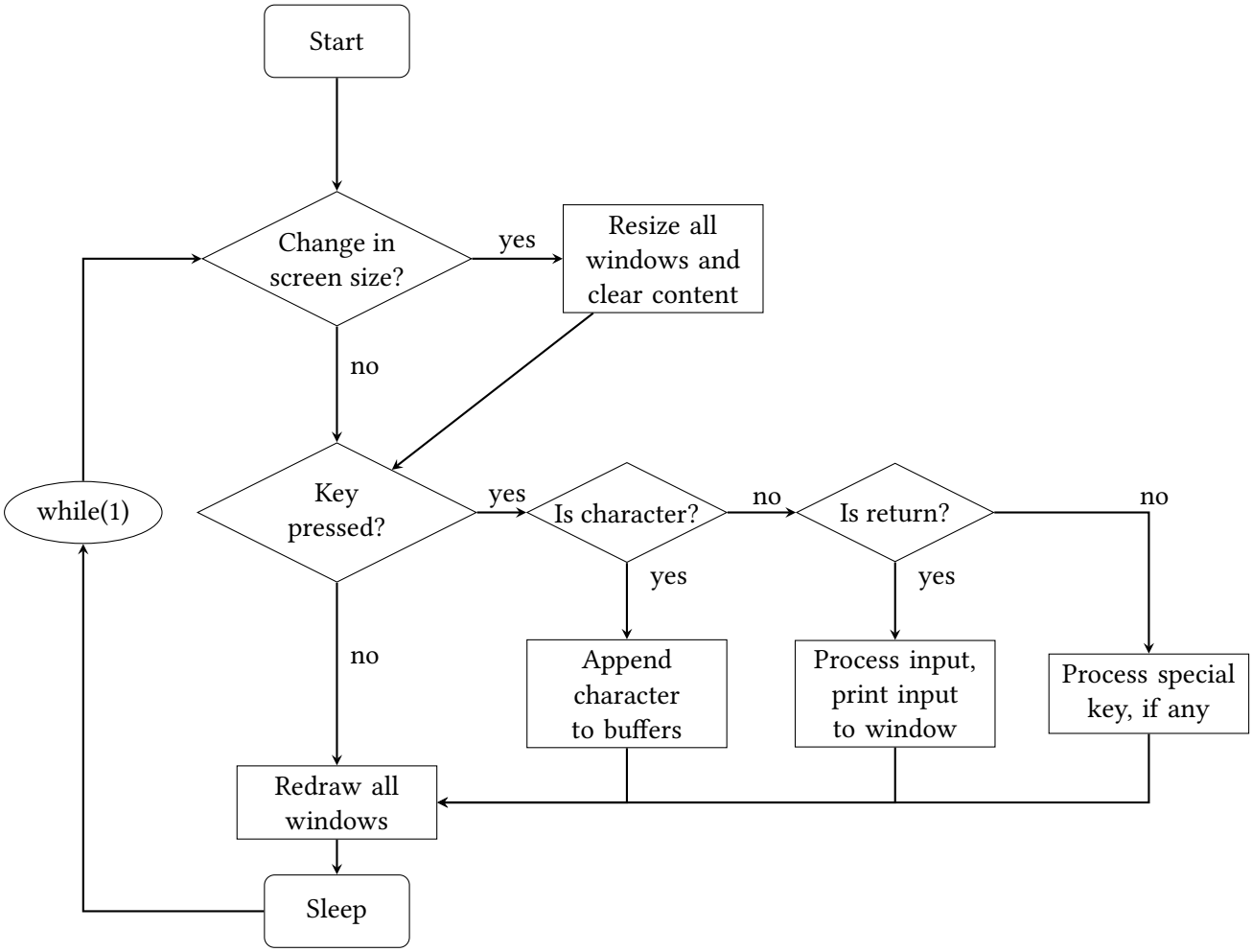

Figure 6.1: Flowchart over the main loop of the interface module.

6.3.2 Send/receive module

6.4 Conclusion and Discussion

As for now, the serial interface software consists of two modules: the sending and receiving module, and the ncurses command interface. Both modules works 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 integrate the send/receive module into the interface module. In the latest version of the interface module, this is half-way done. What remains is the main transmit and receive routines, which are not completely done. 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.

6.4.1 Notable problems

To keep the GBT-register persistent with the data-pattern, the software has to continuously write to it. Tests were conducted were the software would send a write-pattern only once, and then read it back continuously: When running the software over a period of time, certain bits would at random times switch back to their initiated values. For instance, if the data-bit of address ”x0C” is initiated ’0’ when the FPGA is turned on; when written a ’1’ to it by the software, it would change back to ’0’ after a random period of time if it is not written to again. This seemed to occur more often for certain bits than others, but when running the software in ”read-only” for longer periods of time, more and more bits would eventually switch back to their initiated values. The problem seemed to become worse at higher baud rates. However, when the program kept writing to the register after every read, the bits would remain at the desired values.

When writing input-strings in the commandline of the interface module, one can only use *backspace* and *enter* to get rid of the string that is typed in. *Backspace* removes the last character typed in, and *enter* will result in the program trying to execute the typed-in string and then remove it completely from the commandline (and respond with an error if the string does not match any built in commands). Attempts were made to implement similar behaviour found in linux-terminals: using the keyboard buttons, *up* and *down*, to copy previous written commands into the commandline, and use the *left* and *right* buttons to move the cursor in between the characters of the string. While the *up* and *down* commands are usable, scrolling between characters with *left* and *right* did not work very well, and the functionality were left out.

Chapter 7

External and Internal Loopback test of the GBT bank Quartus Example

7.1 120 MHz reference clock

To be able to conduct a proper loopback test, the MGT and the PLLs of the GBT must have a input clock frequency of $120\ \mathrm{MHz}$. There are a number of ways to achieve this on the Cyclone V board:

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

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 [13]. However, at the time of conducting the first tests, there was no available signal generators that could generate a $120\ \mathrm{MHz}$ 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 setup the internal oscillator on the Cyclone V board for use as reference clock.
- How to setup the $Si338$ external oscillator for use as reference clock.

7.2 Configuring the on-board oscillator on the Cyclone V board

To achieve a reference clock of 120 MHz without an external clock, the FPGA has an on-board programmable oscillator; the $Si570$ from Silabs. It uses Inter-Integrated Circuit (I2C) for serial communication and can be programmed to output frequencies up to 810 MHz $\pm$ $50ppm$. 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 [12]. 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 been proven 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) using the Windows Operating System (Linux was also attempted, but without any success) and specify the right paths for the related environment variables. The below sections gives a brief description on how this was achieved.

7.2.1 Clock Control software setup

The Cyclone V 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 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`.

7.2.2 Steps for configuring Windows to run the Clock Control software

1. Install Quartus 13.x (includes jtagconfig) together with the Cyclone V device support [4].
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 [15]).

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:

$$f_{out} = \frac{f_{XTAL} \times RFREQ}{HSDIV \times N1} \tag{7.1}$$

, 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 [6]. 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 7.1 shows the Clock Control software after a new frequency of 120 MHz is set.

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 7.2 and 7.3 shows the output frequency, before and after configuration.

7.3 Configuring the $Si338$ external oscillator

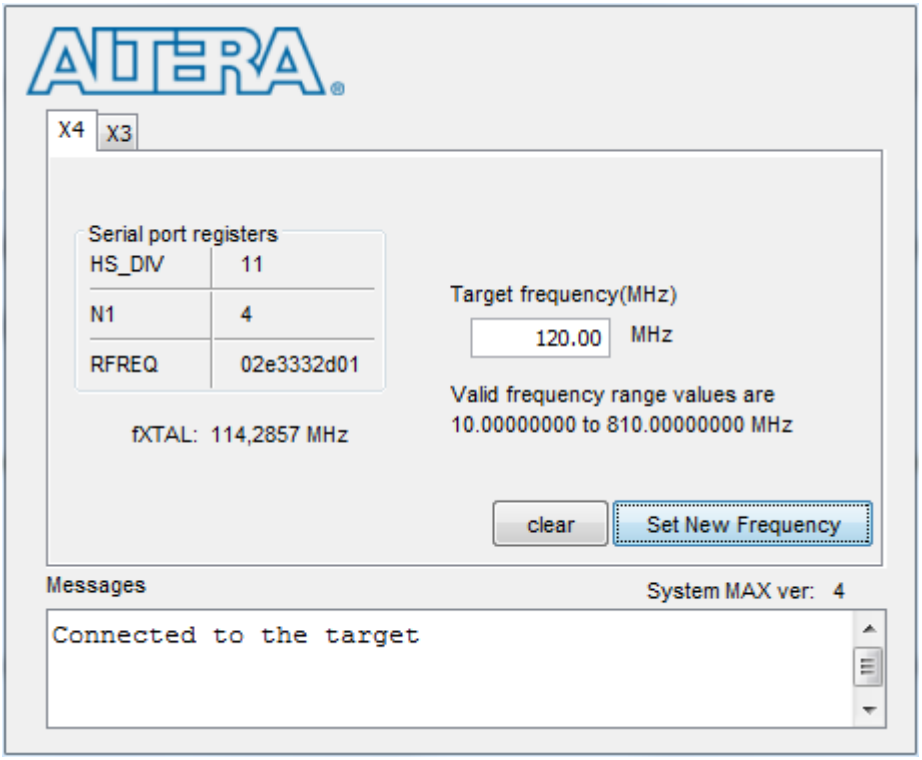

Figure 7.1: Clock Control software by Altera used to program the $Si570$.

Figure 7.2: $Si570$ Before configuration: 100 MHz.

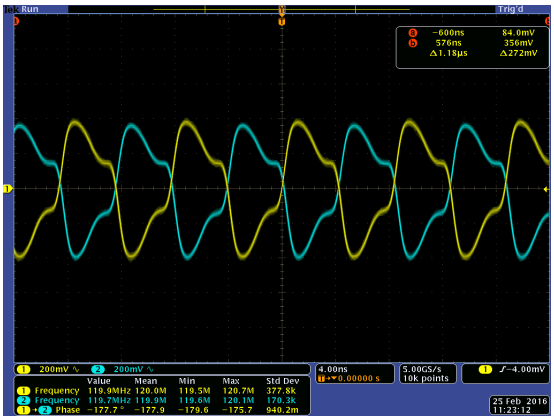

Figure 7.3: $Si570$ After configuration: 120 MHz.

Chapter 8

Testing and verification of the HDMI daughter card

To verify a working PCB, the HDMI daughter card underwent a series of tests. The sections below summarizes these tests.

8.1 Connectivity test

Since all the components on the PCB was 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.

8.1.1 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.

8.1.2 Experimental setup

The setup involves the use of a multimeter to go through 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.

8.1.3 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 $3.3\ \mathrm{V}$ output and a current output limited to $100\ \mathrm{mA}$. 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 $1.25\ \mathrm{V}$ voltage divider, i.e around $40\ \mathrm{mA}$.

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.

8.2 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.

8.2.1 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 $4.8\mathrm{Gbit/s}$.

8.2.2 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 In-system-source-and-probe editor and SignalTap II, it was possible to connect the transmitter to a internal counter that counted with increments of one. SignalTap II was then used to monitor and verify that the receiver line received the same incremented counter as the transmitter sent out.

8.2.3 Results

8.3 External loop-back test for the HDMI connectors

8.3.1 Purpose of tests

- Measure the quality of the signal (eye-diagram) at different frequencies up to $300\ \mathrm{MHz}$ 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.

8.3.2 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. A pseudo-random generated bit-signal would travel out via one of the transmitters of the FPGA, out through a HDMI-connector, following the cable back into the same HDMI-connector into the receiver input. This is to simulate the transmission path to the VLDB card. Because of the trace- and cable-length, a delay is introduced between the transmitted and received signal. As mentioned in chapter 3, a signal propagates through the conductor at a velocity of approximately $15\ \mathrm{mm/ns}$. Cables with two different lengths were tested, to see if it would affect the bit-error rate.

To measure the quality of the LVDS signals with an oscilloscope, differential probes were used.

8.3.3 Results

8.4 Conclusion and discussions

When measuring the quality of 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 8.1) 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 throught the HDMI-daughter card to the end of the hdmi-cable.

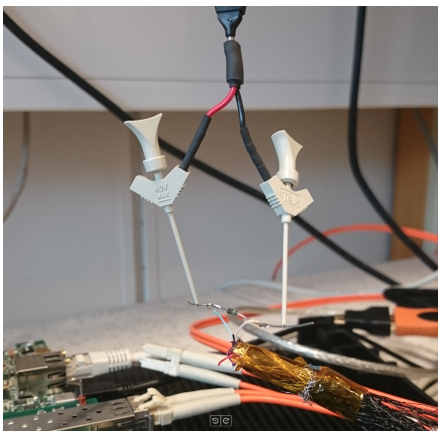

Figure 8.1: Small clamps was first used to measure the high-speed signal.

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 figures 8.2 and 8.3.

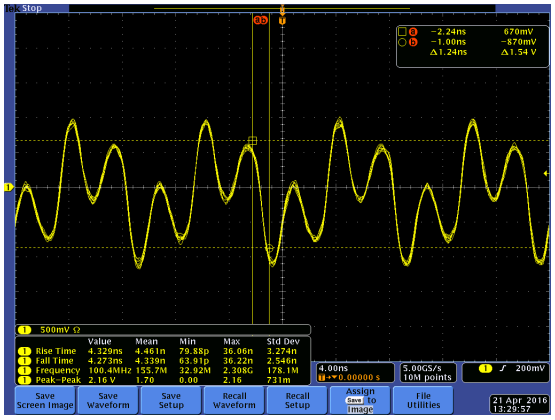

Figure 8.2: 100 MHz measured using clamps.

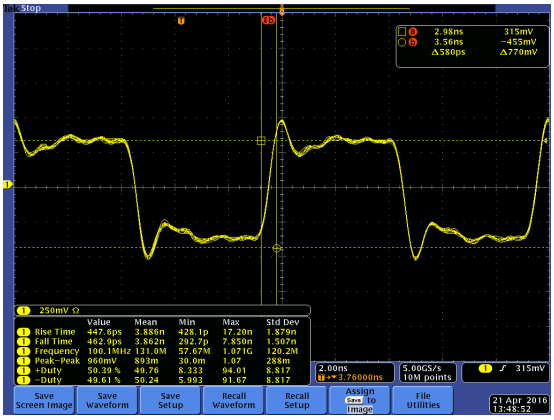

Figure 8.3: 100 MHz measured using stubs.

As shown in figure 8.3, reflections still occurs. By doing the same measurements again with both the GPIO-card and the HDMI-daughter card on both available FPGAs, the reflections remained somewhat the same. It is 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. This lead to the loop-back test using SignalTap II in an attempt to sample the signals (see chapter 9).

Chapter 9

Testing and verification of the Serial interface

9.1 Hardware simulation using testbench in Modelsim

9.1.1 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.

9.1.2 Bitvis Utility Library

The *Bitvis Utility Library* is an open source Very High Speed Integrated Circuit Hardware Description Language (VHDL) testbench infrastructure library for verification of FPGA and ASIC [11]. It was developed by Bitvis 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).

9.1.3 Experimental setup

9.1.4 Results

9.2 Connection between COM port and UART using SignalTap II

By using SignalTap II to tap the transmitter and receiver lines, it is possible to verify that the FPGA design is responding correctly to the bytes sent from the C-program on the PC side.

9.2.1 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.

9.2.2 Experimental setup

9.2.3 Results

While testing the FPGA design, it occured that the UART would hang after a random period of time. The reason as to why this was happening has yet to be found. A quick fix to this, however, was to simply implement a timer that would reset the UART on the condition that no bytes have arrived in a given amount of time when the UART is in the idle state. 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.

9.3 Conclusion and discussions

Chapter 10

Conclusion and discussion

Glossary

COM port Serial Communications port on a PC.. 15, 21, 32, 33, 50

FIFO Short for First In First Out, a shift-register connected as a circular buffer. The data is written and read in the same order i.e. first data in, first data out. FIFOs can be used to buffer data between two asynchronous streams.[19]. 24, 28

flash A nonvolatile memory type known by its name because of its ability to erase memory blocks all at once ”in a flash”.[19] Nonvolatile meaning that the memory will keep all its data even if the power is switched off.. 2

PISO Short for Parallel In Serial Out, a PISO is a shift-register circuit for serializing input parallel data.. 6

SIPO Short for Serial In Parallel Out, a SIPO is a shift-register circuit that converts input serial data into parallel data.. 6

SRAM Short for Static Random Access Memory, a type of volatile memory that uses internal feedback to keep on its data.[19] Volatile meaning that the memory loses all the data once the power is switched off.. 2

USB-to-RS232 adapter Adapter to convert between USB and Rs-232 signaling and voltages.. 18, 20, 21

Acronyms

ASIC Application Specific Integrated Circuit. 1

BNC Bayonet Neill–Concelman. 18

CDR Clock & Data Recovery. 6
CLB Configurable Logic Block. 2
CML Current-Mode Logic. 4
COTS Commercial Off-The-Shelf. 1
CRU Common Readout Unit. 1, 15

DCE Data Communication Equipment. 23
DTE Data Terminal Equipment. 23

FPGA Field-Programmable Gate Array. 1–3, 5, 7, 8, 11, 15, 18–21, 23, 24, 28, 31, 34, 38–40, 42, 43, 46–50

GBT Gigabit Transceiver. 1, 2, 4–6, 15, 17, 23, 24, 31, 33–35, 39–41, 46
GUI Guided User Interface. 12

HDL Hardware Description Language. 2, 3
HDMI High-Definition Multimedia Interface. 4, 7, 11, 13, 45–48
HPS Hard Processor System. 19
HSMC High-Speed Mezzanine Card. 3, 4, 7–9, 12, 13, 19, 45, 46

I2C Inter-Integrated Circuit. 42
IC Integrated Circuit. 1
IDE Integrated Development Environment. 19

LED Light-Emitting Diode. 46
LHC Large Hadron Collider. 1
LSB Least Significant Bit. 25
LUT LookUp Table. 2
LVDS Low-Voltage Differential Signaling. 4, 7, 9, 46, 47

MGT Multi-Gigabit Transceiver. 2, 4–6, 41
MSB Most Significant Bit. 25

OS Operating System. 20

PC Personal Computer. 1, 15, 18, 20, 21, 23, 24, 29, 31, 32, 43, 49, 50
PCB Printed Circuit Board. 7–14, 19, 45, 46, 48, 59
PCI Peripheral Component Interconnect. 18
PCIe Peripheral Component Interconnect Express. 4
PLL Phase-Locked Loop. 6, 41

RS-232 RS-232. 20, 23, 24, 59

SATA Serial AT Attachment. 4
SDI Serial Digital Interface. 18
SFP Small Form-Factor Pluggable HSMC board. 7, 8, 14
SLHC Super Large Hadron Collider. 1

UART Universal Asynchronous Receiver/Transmitter. 15, 20, 21, 24–28, 31, 50, 59
USB Universal Serial Bus. 18

VHDL Very High Speed Integrated Circuit Hardware Description Language. 2, 3, 24, 49
VLDB Versatile Link. 1, 7–9, 19, 47

XCVR Transceiver. 7, 8

Bibliography

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

[2] 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.

[3] 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.

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

[5] *SFP HSMC Schematic (hsmc_sfp_revb.pdf)*, 2013. Accessed: 03/12/2015.

[6] *Si570/Si571 Datasheet*, 2014. Accessed: 29/02/2016.

[7] 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.

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

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

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

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

[12] 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.

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

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

[15] 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.

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

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

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

[19] D. M. H. Neil H. E. Weste. *CMOS VLSI Design, A Circuit And Systems Perspective*. Addison-Wesley, Pearson, fourth edition, 2011.

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

[21] Sophie Baron and Manoel Barros Marin. *GBT-FPGA User Guide*, 2015. Accessed: 29/02/2016.

[22] A. Velure. Uart2Bus vhdl module, 2010. Design used as a reference to the serial communication part on the FPGA side. Accessed: 12/02/2016.

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

List of Figures

List of Tables