0.1 Hardware Design on the FPGA Side

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

0.1.1 Specification

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

- Communication with the user PC using UART and RS-232.
  - Receive requests sent from the user PC. The requests are reserved byte-codes in which the byte-decoder translates to read or write commands.
  - Send out data from the Gigabit Transceiver (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 and write to the GBT control-register.

0.1.2 Hardware Components

The below sections gives a brief description of each part of the module design.
The Universal Asynchronous Receiver/Transmitter (UART) itself was based on the UART-design by Pong P. Chu, found in chapter 7 from his book *FPGA Prototyping By VHDL Examples* [?]. The Baud Rate Generator was 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 a read-request if 0xDD or a write-request if 0xEE (write value 0) or 0xFF (write value 1), and manipulates or sends out data-bits from the GBT-register according to the received requests. The UART itself is optimized for 19200 bit/s, but has been tested to work at speeds up to 57600 bit/s.

UART

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

The UART-design is divided into five parts:

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

Oversampling and the Baud Rate Generator

To obtain an accurate sampling of the received signal, an asynchronous system like the UART uses what is referred to as oversampling.

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 [?]. 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.

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

$$Baud\ frequency = \frac{16 \times baud\ rate}{GCD(clock, 16 \times baud\ rate)} \tag{1}$$

, where baud frequency is the count steps, and *GCD* is the greatest common divisor between the global clock and the baud rate times 16 [?].
For a baud rate of 19200 $\mathrm{bit/s}$ and a clock of 50 MHz, the counter must count in steps of 96 per clock cycle.

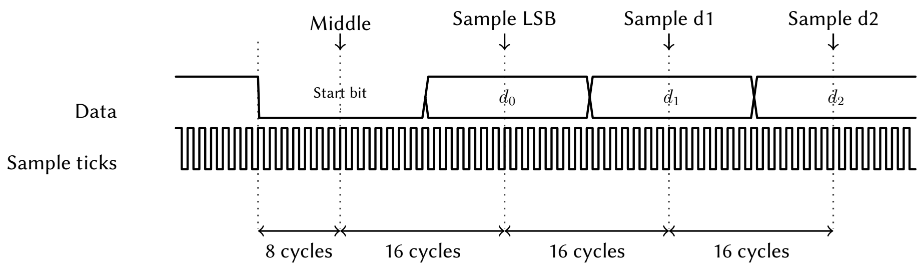

Figure 1: UART receive synchronisation and data sampling points with 16 times the sampling rate.

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:

$$Baud\,limit = \frac{clock}{GCD(clock, 16 \times baud\,rate)} - baud\,frequency \tag{2}$$

, 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 [**?**].
For a baud rate of 19200 $\mathrm{bit/s}$ and a clock of 50 MHz, the counter must count in steps of 96 up to the baud limit of 15565, before pulling the tick-signal high and start over again.

Receiver

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

Transmitter

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

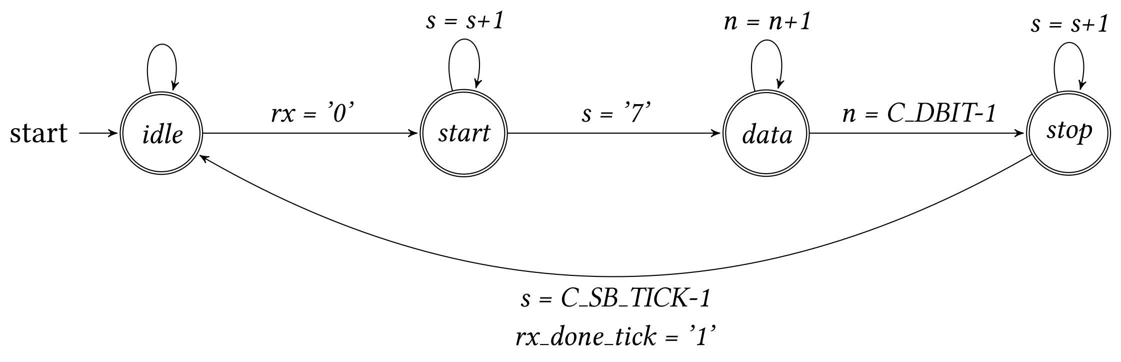

Figure 2: UART receiver state machine.

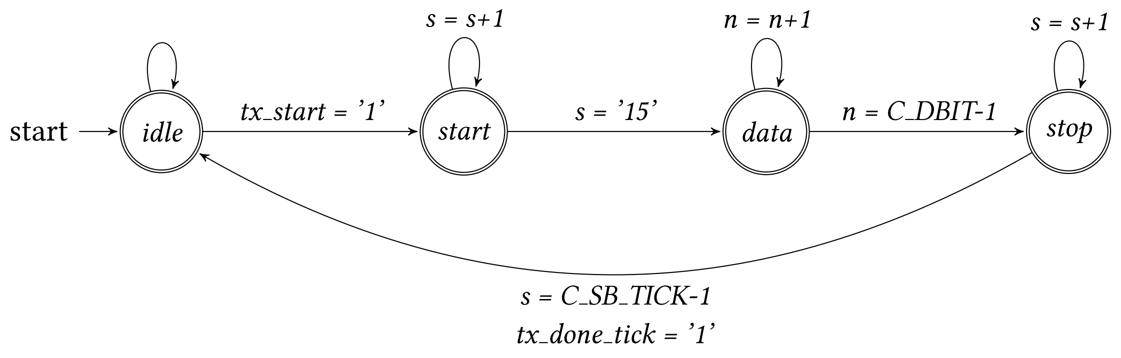

Figure 3: UART transmitter state machine, similar to that of the receiver.

signal.

FIFO Buffers

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

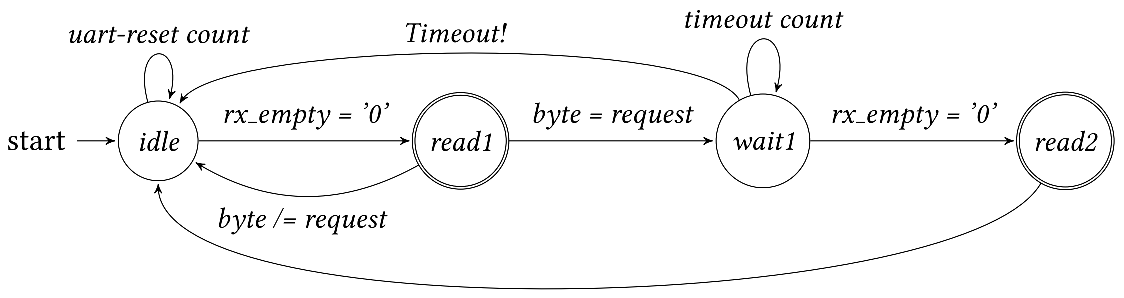

Figure 4: UART decoder state machine.

Decoder

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

0.1.3 Conclusion

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