Using UART on DE2-115 FPGA board

1. Overview

The objective of this project is to design several simple applications to showcase the use of the Universal Asynchronous Receiver/Transmitter (UART) to connect the FPGA chip on the DE2-115 board [1] to the host PC computer.

2. Overview

Before anything, we should first discuss a little the Universal Asynchronous Receiver/Transmitter (UART). If you are already familiar with it, you can skip this section. This discussion here has been adapted from lab#4 of the Embedded Systems course I teach here [2] and the chapter on UART from this textbook [3].

The most basic method for communication with the FPGA chip or an embedded processor is asynchronous serial. It is implemented over a symmetric pair of wires connecting two devices (referred as host and target here, though these terms are arbitrary). Whenever the host has data to send to the target, it does so by sending an encoded bit stream over its transmit (TX) wire. This data is received by the target over its receive (RX) wire. The communication is similar in the opposite direction. This simple arrangement is illustrated in Fig.2.1 below. This mode of communications is called "asynchronous" because the host and target share no time reference (no clock signal). Instead, temporal properties are encoded in the bit stream by the transmitter and must be decoded by the receiver.

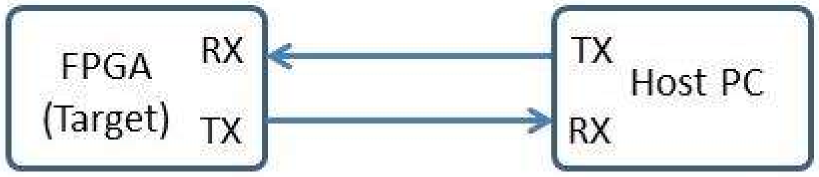


Figure 2.1: Basic serial communication.

A commonly used device for encoding and decoding such asynchronous bit streams is a Universal Asynchronous Receiver/Transmitter (UART). **UART is a circuit that sends parallel data through a serial line.** UARTs are frequently used in conjunction with the RS-232 standard (or specification), which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment.

A UART includes a transmitter and a receiver. The transmitter is essentially a special shift register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts in data bit by bit and reassembles the data. The serial line is '1' when it is idle. The transmission starts with a start-bit, which is '0', followed by data-bits and an optional parity-bit, and ends with stop-bits, which are ‘1’. The number of data-bits can be 6, 7, or 8. The optional parity bit is used for error detection. For odd parity, it is set to ‘0’ when the data bits have an odd number of ‘1’s. For even parity, it is set to ‘0’ when the data-bits have an even number of ‘1’s. The number of stop-bits can be 1, 1.5, or 2. The transmission with 8 data-bits, no parity, and 1 stop-bit is shown in Fig.2.2 (note that the LSB of the data word is transmitted first).

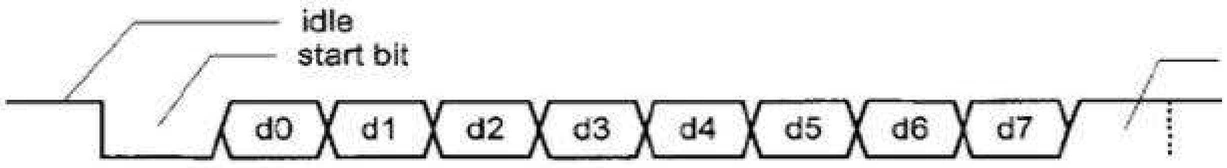


Figure 2.2: Transmission of a byte.

No clock information is conveyed through the serial line. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud rate, the number of data bits and stop bits, and use of parity bit.

To understand how the UART's receiver extracts encoded data, assume it has a clock running at a multiple of the baud rate (e.g., 16x). Starting in the idle state (as shown in Fig.2.3), the receiver “samples” its RX signal until it detects a high-low transition. Then, it waits 1.5 bit periods (24 clock periods) to sample its RX signal at what it estimates to be the center of data bit 0. The receiver then samples RX at bit-period intervals (16 clock periods) until it has read the remaining 7 data bits and the stop bit. From that point this process is repeated. Successful extraction of the data from a frame requires that, over 10.5 bit periods, the drift of the receiver clock relative to the transmitter clock be less than 0.5 periods in order to correctly detect the stop bit.

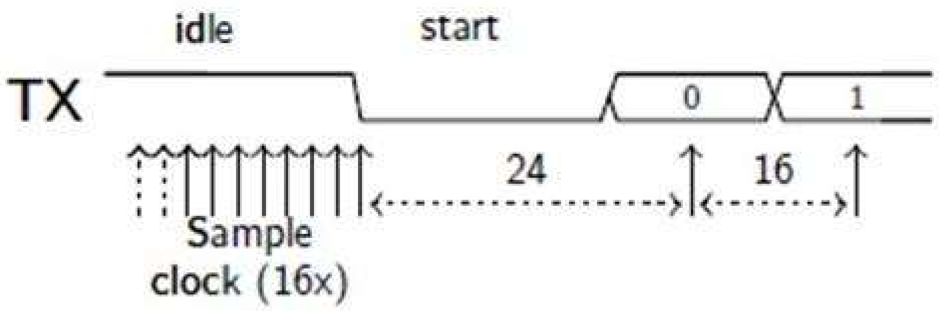


Figure 2.3: Illustration of signal decoding.

UARTs can be used to interface to a wide variety of other peripherals. For example, widely available GSM/GPRS cell phone modems and Bluetooth modems can be interfaced to a microcontroller UART. Similarly GPS receivers frequently support UART interfaces. For more details on UART and RS-232, please see [4] and references therein.

In this project, we'll use an UART controller - completely described in VHDL - to connect the FPGA chip of the DE2-115 board to the host PC. The FPGA on the DE2-115 board can be connected to the host PC through the RS-232 connector on the board as shown in Fig.2.4 (which is an excerpt from the user manual of the board). The DE2-115 board uses the ZT3232 transceiver chip and a 9-pin DB9 connector for RS-232 communications. For detailed information on the ZT3232 transceiver, here is its datasheet [5].

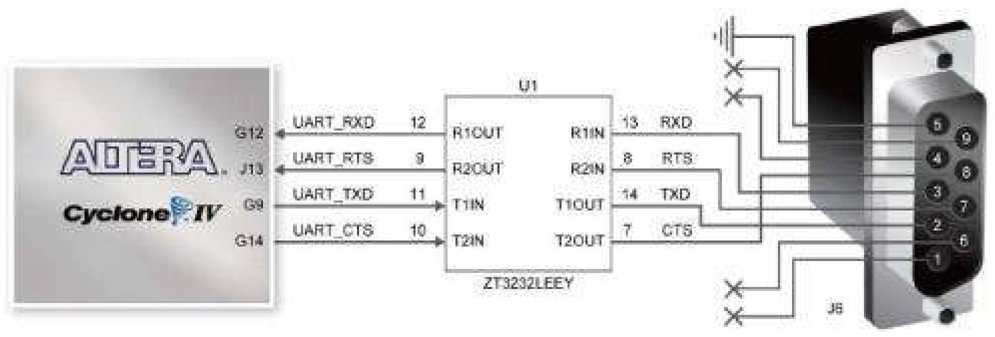


Figure 2.4: Connections between FPGA and ZT3232 (RS-232 chip) and the DB9 connector on the DE2-115 board.

3. Implementation #1: Send characters from host PC to design entity on FPGA, which "increments" and loops back the incremented chars

Here, we simply replicate an example from the textbook [2], with some minor changes to adapt it to work on the DE2-115 board. This is a simple design entity running on the FPGA and constructed with basically a UART controller - completely specified in VHDL - which receives characters we send from the host PC, increments them, and sends them back (i.e., loop back) to the host PC. On the host PC, we use a simple serial hyperterminal-like program such as putty to send characters and to receive and display the looped-back data. The block diagram of the entire design is shown in Fig.3.1.

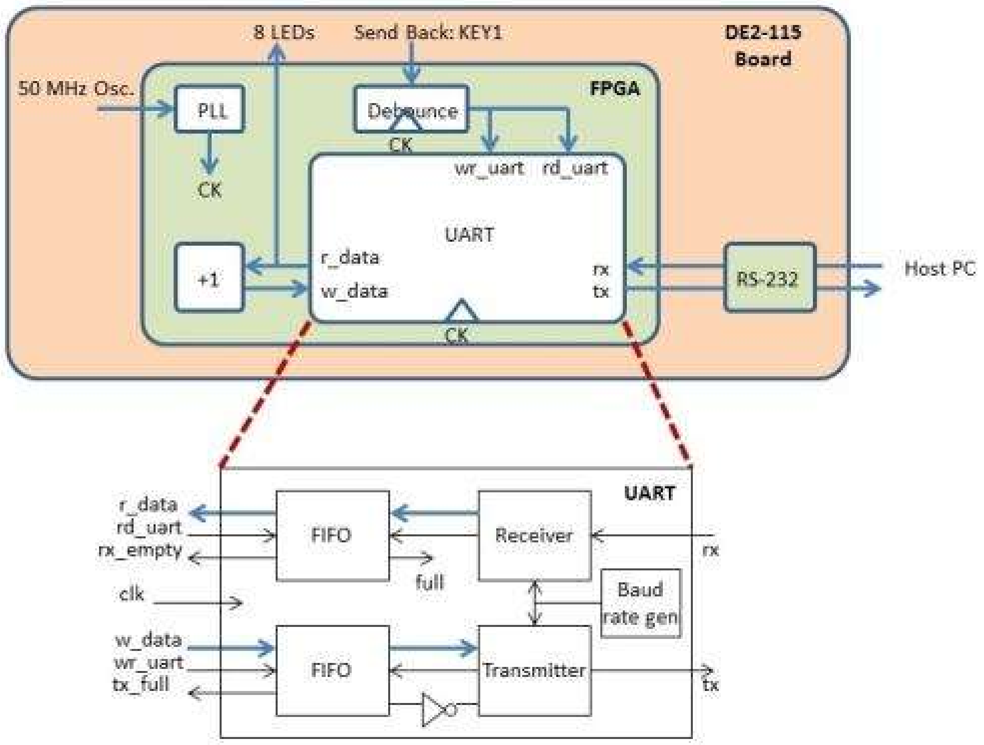
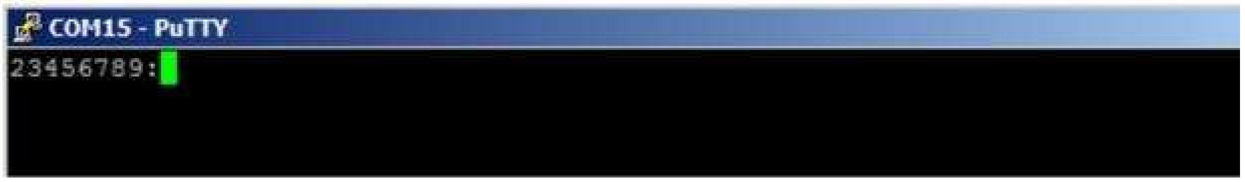


Figure 3.1: Block diagram of design that uses VHDL UART controller to connect FPGA of DE2-115 board to host PC. User on host PC sends characters via a Putty terminal; characters are "incremented" on FPGA, and sent back to host PC.

You can download the entire Quartus II project directory at the bottom of this page. To set it up, you need to program your DE2-115 board, connect the board with a serial cable to the host PC (possibly using a USB to Serial adapter, like I do on my laptop, which does not have a serial connector), and use a Serial terminal set up with a baud rate of 19200. The required baud rate is 19200 because that is how the VHDL UART controller is set up using some hard coded values. If you want to change this, you must change the hard coded values inside the VHDL code (look inside 2_uart.vhd) and then "re-compile" the whole project to generate a new programming file. In this project I use a Putty terminal. Once, everything is set up, if you type in the Putty terminal for example "123456789" and then push KEY1 push button on the DE2-115 enough times, you should get that data looped back and displayed inside the putty terminal as "23456789:" - as shown in Fig.3.2 below. Pushing one time KEY1 button on DE2-115 board has as effect: reading one data from the RX buffer, incrementing the data, and placing the incremented data into the TX, which will then transmit towards the host PC.



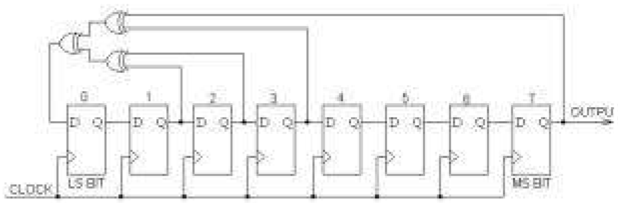
```
23456789:
```

Figure 3.2: Putty terminal shows data received from the FPGA, which incremented and looped-back what we sent in the first place.

4. Implementation #2: FPGA sends stream of data to host PC where a realtime moving plot displays the data

Here, we use the UART controller on the FPGA to send a stream of values that we use to display a realtime moving plot on the host PC. On the host PC, a Python based application receives the stream of data and displays the plot. The stream of values sent from within the FPGA could represent for example sensor data (such as temperature readings, DC drives related measurements, etc.) that you would like to monitor during operation/debug of your design.

We'll use a pseudo-random number generator to generate 8-bit random numbers, which we transmit via the UART controller to the host PC. The pseudo-random number generator is implemented using a linear feedback shift register (LFSR). An n-bit LFSR is an n-bit length shift register with feedback to its input. The feedback is formed by usually XORing the outputs of selected stages of the shift register - referred to as "taps" - and then inputting this to the least significant bit (stage 0). The "linear" part of the term LFSR derives from the fact that XOR is a linear function. This is the so called *many-to-1* topology. In contrast, in the *1-to-many* topology, the feedback is taken from the MS bit and combined into taps at various stages. This LFSR topology will still be maximal length, but will provide a different sequence of numbers. Examples of a 8-bit LFSRs are shown in Fig.4.1.



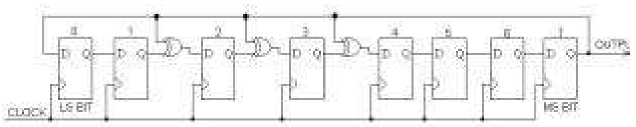


Figure 4.1: a) 8-bit LFSR using the so called many-to-1 topology. b) 8-bit LFSR using the so called 1-to-many topology.

The block diagram of the entire design is shown in Fig.4.2.

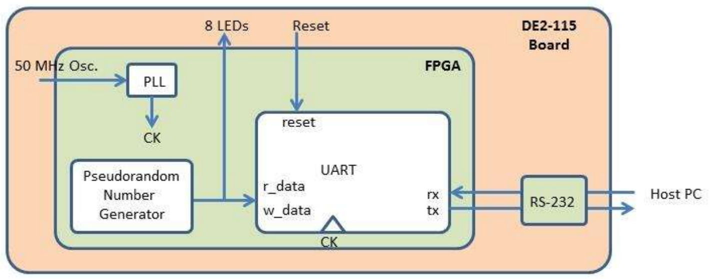


Figure 4.2: Random 8-bit numbers are generated using LFSR on the FPGA and sent to the host PC, which displays them as moving plot.

You can download the entire Quartus II project directory at the bottom of this page. To set it up, you generally need to go through the same steps as in the previous section. However, this time, on the host PC you must run the Python script (I assume you have Python installed on your PC and know the basics of running simple Pythin programs) included with the downloadable archive of this project instead of the Putty terminal. Once you have everything set-up and running, on the host PC you should get something like Fig.4.3.

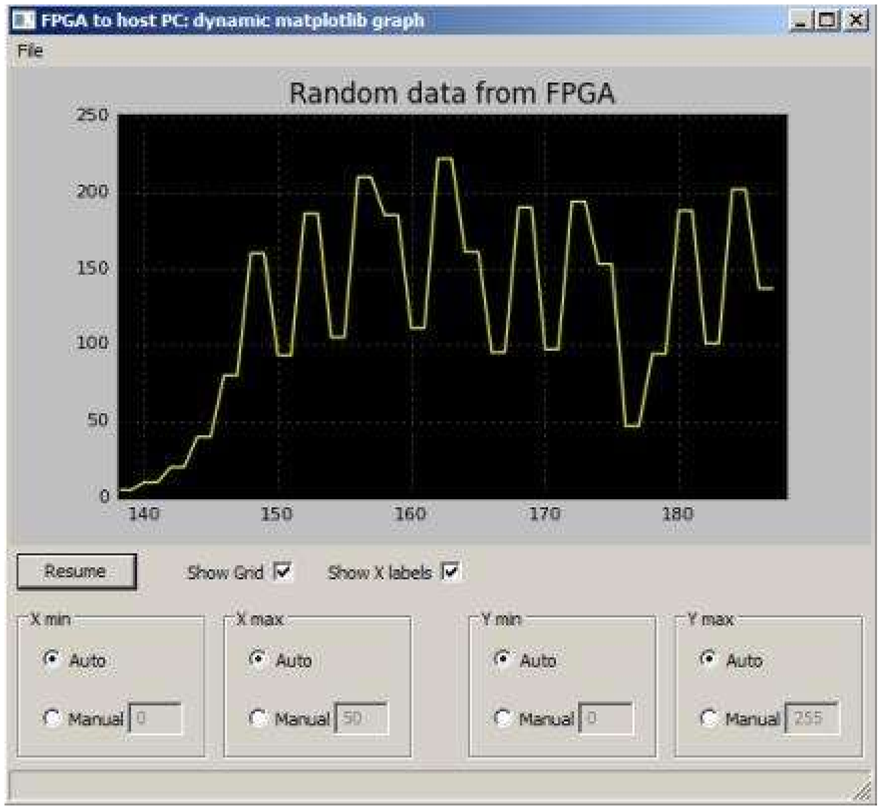


Figure 4.3: Python script creates a simple GUI to display as a moving plot the data (8-bit numbers) received from the FPGA.

5. Implementation #3: Host PC sends image, FPGA receives it, filters it, and sends it back to host PC, which displays it inside simple GUI

In the third and last implementation, we complicate things a little bit. From within an application running on the host PC, we send a BMP image to the FPGA, where we apply a grey filter, and send the data back to the host PC. Actually, we send to the FPGA only the pixel data; we omit the header portion of the BMP image to keep things simple. To learn more about the BMP format, you can read the excellent Wikipedia entry [6].