EE109 Lab 1: USB Serial UART

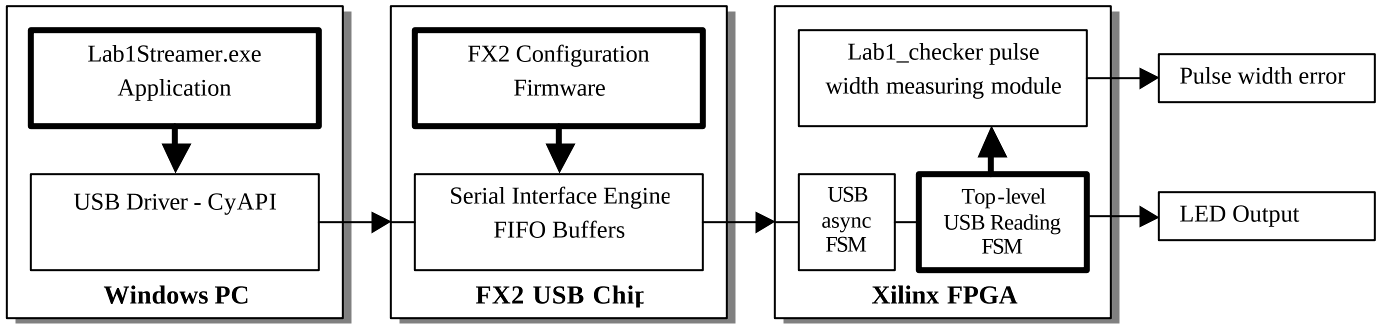
Overview: This lab is designed to compare the performance of a hardware buffered serial output UART to that of a non-buffered, software driven one. For Part A of the lab you will implement a simple system to output a stream of 1s and 0s under software control. For Part B you will modify this to include a hardware buffer to (hopefully) improve performance. The performance will be measured by the maximum rate at which your design can send 2048 bits without having the time between any two transitions differ by more than 6% from the average of the first 8 transmissions.

Contents

Revisions

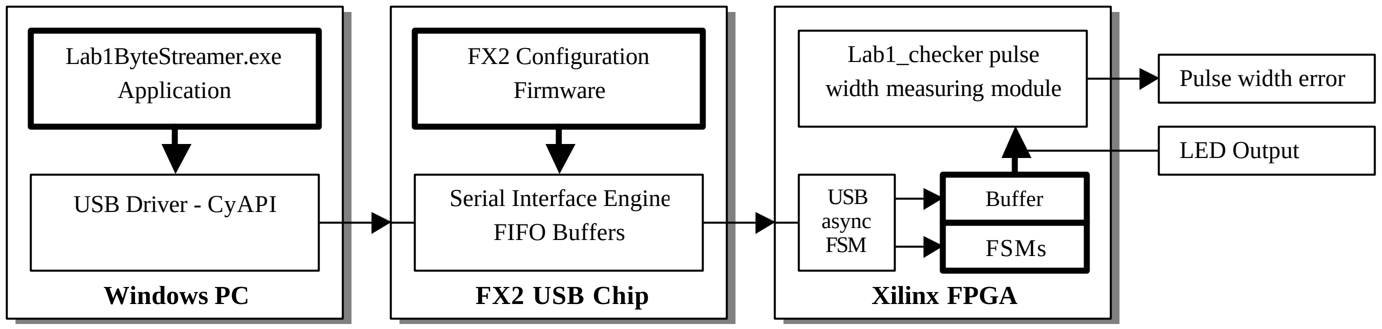
19-March-2004: Version 1.0, by David Black-Schaffer
24-March-2004: Version 1.1, added more hints and more clearly specified required measurements.
3-April-2004: Version 1.2, updated the SR counter discussion to include more detail

Part A System Overview



The architecture for Part A is specified above. You are to implement a Windows application that will send out a stream of single-byte USB packets with one 1 or 0 in each. You should configure the FX2 to accept these packets over USB Endpoint 2 OUT and pass them on to FIFO 0 where the FPGA will read them out and send the one bit on to an LED for display. It should also send this data to the provided lab1_checker module, which will be responsible for averaging the first 8 transitions and comparing subsequent transitions to them. If any subsequent transitions are more than 6% longer or shorter than the average of the first 8 it will report an error by turning on an LED.

Part B System Overview



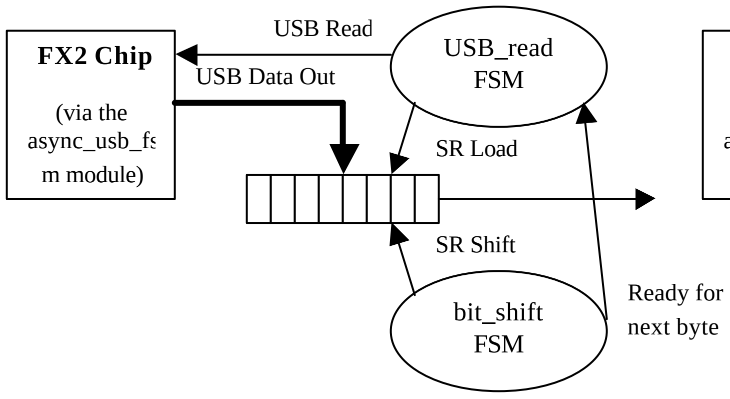
The architecture for Part B is less clearly specified. You are expected to implement a Windows application that will send out a stream of *n*-byte USB packets, where each bit will be used for generating the serial data output stream. You should read these bytes into some sort of a buffer on the FPGA and then use a shift register to shift them out one at a time. The rate at which you shift them out will be controlled by a hardware counter, whose speed you will vary to see how fast your system can go. Your job is to try to get that rate as high as possible by designing a good buffer. (Note that you should systematically adjust the rate of the counter that controls the hardware shift register to find the speed at which your throughput is maximized. You can do this most efficiently by using a binary tree to test the various speeds. For example: you might start at 50MHz and determine that is too fast. Then try half of that, or 25MHz, which may work fine. If it does then your top speed will be 25MHz because there is no way to divide down a 50MHz clock without a DLL to get anywhere between 50 and 25. If 25MHz does not work, then again decrease by half again and try it at 12.5MHz. Keep dividing incrementally adjusting your speed by half the last step until you get reasonably close to the maximum speed you can sustain.)

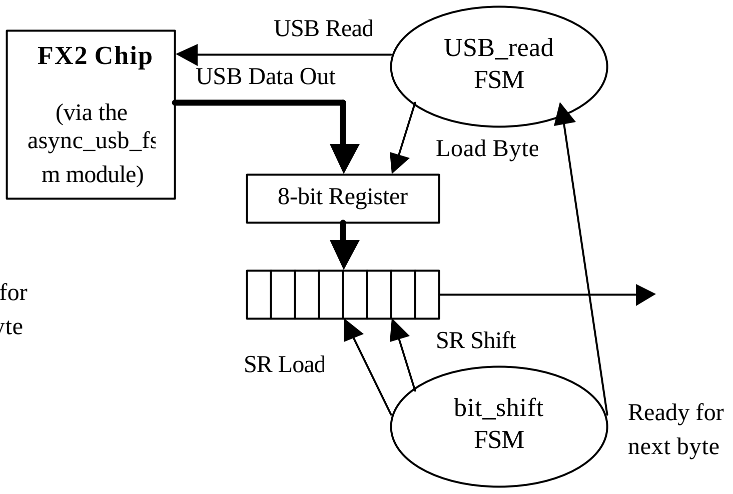
Since all our logic will be synchronous to the 50MHz system clock you will need to create a hardware counter that will generate SR_enable signal that tells the SR when to shift a bit. This is the piece you will need to adjust to see how fast you can go. Building such a counter is very simple, but you must remember that its output is only to enable your SR. The SR, as with all your other logic, must be clocked off your global 50MHz clock. Here is an example of building such a simple counter:

```
wire sr_enable;
wire [15:0]sr_count;

dffre #(16) my_sr_enable_counter (
      .clk(clk),
      .en(enable),
      .r(reset || sr_enable),
      .d(sr_count + 16'd1),
      .q(sr_count)
      );

`define DIVIDER 16'd3

assign sr_enable = (sr_count == `DIVIDER) ? 1'b1 : 1'b0;

// Then your SR would be connected as:
sr my_sr (
      .clk(clk),              // REMEMBER: we always clock off the global clk
      .shift_en(sr_enable),   // Use an enable to try different speeds
      // Other connections
      );
```

You can either build your own SR (easy) or use a CoreGen module if you'd prefer. Remember that drawing out a block diagram of how you want it to work first will make it easier to figure out if you're not already completely comfortable with how a SR functions.

Some comments on the buffer design: Obviously once you have the shift register (SR) filled with bits you can then shift out 8 bits as fast as your FPGA runs. However, the hard part is going to come when you finish shifting them out of your SR and you have to re-load it. In that case your bit_shift FSM will have to tell the USB_read FSM to read in the next byte, and it may have to wait for a while. This time you spend waiting will limit your performance. However, if you were to have a second register that fed the shift register you could re-load that while you were in the processes of shifting out bits, and thereby significantly improve your performance. One could imagine adding more and more of these registers, but at some point you will simply be limited by the rate at which the computer can send out bytes in the first place.





Sample Files Overview

A rather thorough description of the sample files for Part A can be found in the tutorial.

Performance

To evaluate performance you will experimentally determine the maximum rate at which you can send out 2048 transistions in a row without an error for both Parts A and Part B while only sending 1 byte per USB packet. (Remember that Part A only uses one bit of the byte so you will be sending 8 transitions in 1 byte for Part B in this comparison vs. only 1 for Part A.) You will then compare this to the performance if you send out 2, 4, 32, and 128 bytes at a time (or 16, 32, 256, and 1024 transitions) for Part B. Please graph the results in terms of Mb/s, where each transition represents 1 bit. Note that you should try varying the SR counter for the different numbers of bytes you send at once, but you do not have to adjust it perfectly. Please do specify the shift rate (in MHz) that gave you the highest throughput for each one on your graph.

Additional Write Up Material

In addition to the standard write up (for Part B only) as specified on the course web page you should answer the following questions from your experience with Lab 1.

1. Please explain all warnings the Xilinx tools gave you in synthesizing and implementing Part B. You can explain them briefly, but you should make it clear why each one is acceptable.

2. What was the limiting factor in how quickly you could send bits with Part A? What was the minimum speed you could request and what was the minimum average speed you got? What caused this discrepancy? (i.e., what was taking up the time?)

3. What are you effectively building if you add many extra registers to Part B? If you let the number of such registers increase to a large number, what would limit the maximum output rate you could obtain over a short period of time? What would limit the maximum output rate you could obtain over a long period of time?

4. Include a screenshot of your system when it re-loads the shift register from ChipScope. Please carefully annotate this screenshot so we can easily tell exactly what is going on.