EECS 151/251A FPGA Lab
Lab 5: Serial I/O - UART - I^{2}S Audio Clocks

Prof. Jan Rabaey
TAs: Ali Moin, George Alexandrov
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Contents

1 Before You Start This Lab

Before you proceed with the contents of this lab, we suggest that you get acquainted with what a ready/valid handshake is. Read `resources/Verilog/ready_valid_interface.pdf` in the `fpga_labs_fa18` folder. It may be helpful to draw out the timing diagrams for the ready/valid handshake to gain understanding. If you are having trouble, ask a TA. Please note that the serial line itself is not a ready/valid interface. Rather, it is the modules you will work with in this lab (`UATransmit` and `UAReceive`) that have the ready/valid handshake for interfacing with other modules on the FPGA.

In the first part of this lab you will implement a UART (Universal Asynchronous Receiver / Transmitter) device, otherwise known as a serial interface. Your working UART from this lab will be used in your project to talk to your workstation (desktop) over a serial line.

2 Lab Setup

Run `git pull` in the `fpga_labs_fa18` folder to fetch the latest skeleton files. You will need to copy over your `debouncer.v`, `edge_detector.v`, and `synchronizer.v` from Lab 4 to your sources directory (overwriting the skeleton files). (You could also just copy the contents.)

3 Serial Device

You are responsible only for implementing the **transmit** side of the UART for this lab. As you should have inferred from reading the ready/valid tutorial, the UART transmit and receive modules use a ready/valid interface to communicate with other modules on the FPGA.

Both the UART's receive and transmit modules will have their own separate set of ready/valid interfaces connected appropriately to external modules.

3.1 Framing

On the `Pynq-Z1` board, the physical signaling aspects (such as voltage level) of the serial connection will be taken care of by off-FPGA devices. From the FPGA's perspective, there are two signals, `FPGA_SERIAL_RX` and `FPGA_SERIAL_TX`, which correspond to the receive-side and transmit-side pins of the serial port. The FPGA's job is to correctly frame characters going back and forth across the serial connection. Figure 1 below shows a single character frame being transmitted and will be extremely useful in understanding the protocol.

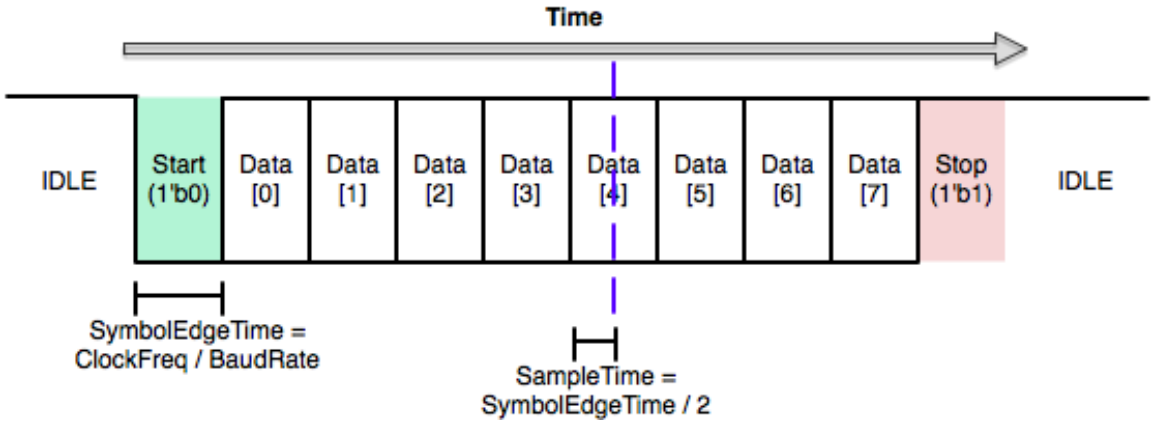

Figure 1: UART Frame Structure

In the idle state the serial line is held high. When the TX side is ready to send a character, it pulls the line low. This is called the start bit. Because UART is an asynchronous protocol, all timing within the frame is relative to when the start bit is first sent (or detected, on the receive side).

The frame is divided up in to 10 uniformly sized bits: the start bit, 8 data bits, and then the stop bit. The width of a bit in cycles of the system clock is then naturally given by the system clock frequency divided by the baudrate. The baudrate is the number of bits sent per second; in this lab the baudrate will be 115200. Notice that both sides must agree on a baudrate for this scheme to be feasible.

3.2 Transmitting

Let us first think about sending a character using this scheme. Once we have a character that we want to send out, transmitting it is simply a matter of shifting each bit of the character, plus the start and stop bits, out of a shift register on to the serial line.

Remember, the serial baudrate is much slower than the system clock, so we must wait $SymbolEdgeTime = \frac{ClockFreq}{BaudRate}$ cycles between changing the character we're putting on the serial line. After we have shifted all 10 bits out of the shift register, we are done unless we see another transmission immediately after.

3.3 Receiving

The receive side is a bit more complicated. Fortunately, we will provide the receiver module. Open `lab5/lab5.srcs/sources_1/new/uart_receiver.v` so you can see the explanation below implemented.

Like the transmit side, the receive side of the serial device is essentially just a shift register, but this time we are shifting bits from the serial line into the shift register. However, care must be taken into determining when to shift bits in. If we attempt to sample the serial signal directly on the edge between two symbols, we are exceedingly likely to sample on the wrong side of the edge (or worse, when the signal is transitioning) and get the wrong value for that bit. The correct solution is to wait halfway into a cycle (until `SampleTime` on the diagram) before reading a bit in to the shift register.

One other subtlety of the receive side is correctly implementing the ready/valid interface. Once we have received a full character over the serial port, we want to hold the valid signal high until the ready signal goes high, after which the valid signal will be driven low until we receive another character.

This requires using an extra flip-flop (the `has_byte` reg in `uart_receiver.v`) that is set when the last character is shifted in to the shift register and cleared when the ready signal is asserted. This allows us to correctly implement the ready/valid handshake.

3.4 Putting It All Together

Although the receive side and transmit side of the UART you will be building are essentially orthogonal, we are packaging them into one UART module to keep things tidy. The diagram below shows the entire setup:

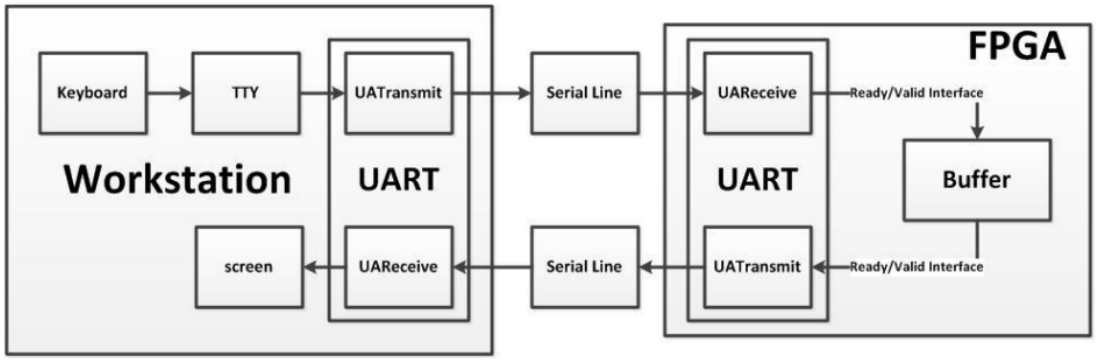

Figure 2: High Level Diagram

3.5 Simulation

We have provided a simple testbench, called `uart_testbench` that will run some basic tests on two instantiations of the UART module with their RX and TX signals crossed so that they can talk to each other. There is also a `.do` file that will run the test. You should note that this test bench reporting success is **not** by itself a good indication that your UART is working properly. The testbench does not attempt to test back to back UART transmissions so you will have to add that test in yourself. Due to the way x's are treated by Modelsim if many signals in your design are undefined the testbench may erroneously pass. Make sure to look at the waveform to see that everything appears to be working properly and that you adequately purge your simulation of high Z and undefined X signals.

If the testbench prints out `# <EOF>` it means that it timed out; this indicates that the testbench was stuck waiting for a condition that never became true. Inspect the waveform and match it up to the testbench code to see where it hangs and why. You shouldn't need to increase any of the timeouts in the `.do` files.

4 Echo

Your UART will eventually be used to interact with your CPU from your workstation. However, since you don't have a CPU yet, you need some other way to test that your UART works on the board.

We have provided this for you. The provided `z1top` contains a very simple finite state machine that does the following continuously:

- Pulls a received character from the `uart_receiver` using ready/valid
- If the received character is an ASCII letter (A-Z or a-z), its case is inverted (lower to upper case or upper or lower case)
- If the received character isn't an ASCII letter, it is unmodified
- The possibly modified character is sent to the `uart_transmitter` using ready/valid to be sent over the serial line one bit at a time

Check using the provided `echo_testbench.v` testbench that everything works as it should in simulation. This testbench is heavily commented to help you understand the communication between the 2 UARTs and the communication over the ready/valid interface. The file often refers to the UART on the workstation as the off-chip UART and the UART on the FPGA as the on-chip UART.

4.1 PMOD serial interface

The Pynq-Z1 does not have an RS-232 serial interface! Well, ok, it does, kind of. The USB interface you use for programming the board (and for RS-232 communication with the board's ARM-based system) is actually only connected to the Processor Subsystem, not the Programmable Logic. We can't use it from our design directly. So, we need to add a serial interface: this is usually a chip wired to connect a device as a serial host/client to another device, with appropriate voltage level shifting to meet the electrical specification (e.g. RS-232). Before you can continue, therefore, you must upgrade your Pynq-Z1 with one: the Pmod USBUART is available for you in this lab. (The Pmod RS232 is another module you could use, but it doesn't take care of framing your data in the USB protocol for you.)

Have a read over the Pmod USBUART Reference Manual (`resources/pmodusbuart_rm.pdf`). You might agree that mapping the pin on the Zynq chip to a PMOD port, then to the right pin on the PMOD header, then to the right pin on the transceiver chip, can be quite confusing. See if you can work it out. Here's a little help:

- Connect `FPGA_SERIAL_TX` in your design to the Pmod USBUART's `RXD` pin (PMOD header pin 2).
- Connect `FPGA_SERIAL_RX` in your design to the Pmod USBUART's `TXD` pin (PMOD header pin 3).

What do RTS and CTS do? Respectively Request to Send and Clear to Send, these signals are used for hardware flow control. We will ignore them, so make sure to disable hardware flow control in the software you use to connect to your board (or more likely, make sure not to enable it by accident).

Note: Make sure that the power selection jumper on the Pmod USBUART is set to LCL3V3 - as you'll read in the reference manual, this is because we're powered the system from an external source and *not* through the tiny USB interface chip on the PMOD module.

4.2 Implement your design

Synthesize your design and generate the bitstream, then program the board just like you have done in previous labs.

Pay attention to the warnings generated by the tool chain. Again, it's possible to write your Verilog in such a way that it passes behavioural simulation but doesn't work in implementation. Warnings about "multi driven nets", for example, can mean that certain logic pathways are never implemented on chip.

If you get stuck, it will help to structure your Verilog as a state machine in a very similar way to the provided `uart_receiver.v`.

4.3 Hello, world!

Now, make sure the USB serial cable is plugged in between the Pynq-Z1 board and your workstation and then run:

```
screen $SERIALTTY 115200
```

This tells `screen`, a highly versatile terminal emulator, to open up the serial device with a baud rate of 115200 (you might have to run as `root`). When you type a character into the terminal, it is sent to the FPGA over the `FPGA_SERIAL_RX` line, encoded in ASCII. The state machine in `z1top` may modify the character you sent it and will then push a new character over the `FPGA_SERIAL_TX` line to your workstation. When `screen` receives a character, it will display it in the terminal.

You can find which `$SERIALTTY` to connect to by perusing the output of the `dmesg` command (in Linux) or checking the Device Manager (in Windows).

Now, if you have a properly working design, you should be able to tap a few characters into the terminal and have them echoed to you (with inverted case if you type letters). Make sure that if you type really fast that all characters still display properly. If you see some weird garbage symbols then the data is getting corrupted and something is likely wrong. If you see this happening very infrequently, don't just hope that it won't happen while the TA is doing the checkoff; take the time now to figure out what is wrong. UART bugs are a common source of headaches for groups during the first project checkpoint.

To close `screen`, type `Ctrl-a` then `Shift-k` and answer `y` to the confirmation prompt. If you don't close screen properly, other students won't be able to access the serial port. If you try opening `screen` and it terminates after a few seconds with an error saying "Sorry, can't find a PTY" or "Device is busy", execute the command `killscreen` which will kill all open screen sessions that other students may have left open. Then run `screen` again.

Use `screen -r` to re-attach to a non-terminated screen session. You can also reboot the computer to clear all active `screen` sessions.

5 I²S Audio

In this and next week's labs we will first develop an interface for and then use an external audio DAC (Digital/Analog Converter). Since our Pynq-Z1 boards do not have one, we will attach one through another PMOD module: the Pmod I2S. (The reference manual is also available as a PDF in `resources/pmodi2s_rm.pdf`.)

The DAC enables our board to output high-fidelity stereo audio. In previous labs you have approximated audio signals using a square wave with single bit resolution. An output filter on the Pynq-Z1 made that wave seem nice, but it still only had 1-bit resolution. The Pmod I2S uses a Cirrus Logic CS4344 D/A converter (datasheet also in `resources/CS4344-45-48_F2.pdf`). "I2S", also written I²S, is the name of the interface format used to communicate with the chip.

5.1 Interface Setup

1. Read the Pmod I2S reference manual carefully.
2. Look over the CS4344 datasheet to reinforce what the reference manual said.
3. If it helps, skim wider resources like the Wikipedia I²S article to get a feel for what you're implementing.

The I²S interface is a lot simpler than other common digital audio interfaces, like AC'97. Like AC'97, however, it requires us to generate very specific clocks for communication. Your **first task** in this part will be to generate the three requisite clock signals for the I²S interface: the master clock `MCLK`, a bit clock `SCLK`, and a left/right channel-select clock `LRCK`. (That means that we will use an "external" `SCLK` source for the CS4344.) These clocks are all derived from our 125 MHz system clock.

Note the special requirements on audio bit alignment to the clock edges, and on which bits are transmitted when. Your **second task** is to generate a bit counter that will track which bit of each sample to output for each bit clock. Even if you don't later use this counter to output the right bit, getting it right is a good exercise in designing to the interface specification.

Figure 3 summarises timing for the interfaces. Use the simple testbench provided (`i2s_controller_testbench.v`) to make sure your waveforms match it. (It's very simple: it just creates a system clock and sends an initial reset signal.)

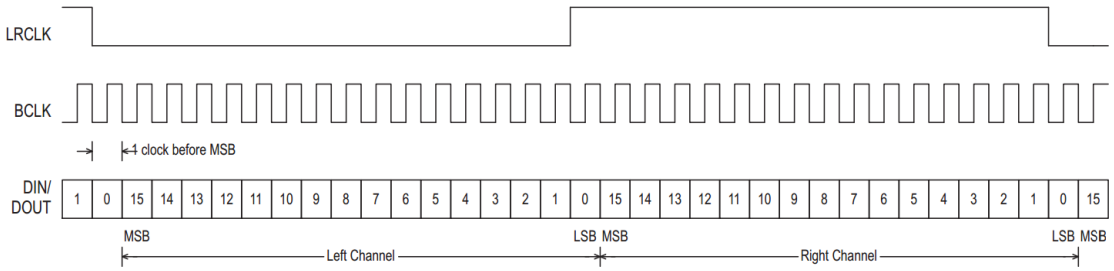

Figure 3: I^{2}S timing summary (credit: Texas Instruments)

The DAC chip allows us to select bit depth and sampling rate. Use:

- Bit depth: 24
- Sample rate (`LRCK`): 88.2 kHz
- `MCLK` to `LRCK` ratio: 128

You might have to change these later, so don't hard code any values you derive from these.

5.2 Checkoff

1. Walk your TA through the simulation results and show that your UART behaves as expected.
2. Show your TA that you can successfully type characters on the keyboard and have them echoed back to display on your `screen` session.
3. Demonstrate to your TA that your I^{2}S clocks waveforms match the requirements in the reference manual.