EECS 151/251A FPGA Lab
Lab 6: External Communication and I^{2}S Audio Clocks

Prof. John Wawrzynek, Nicholas Weaver
TAs: Arya Reais-Parsi, Taehwan Kim
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Contents

1 Finish last week's UART

Recall that in Lab 5 we designed a UART module for our FPGAs but without any way to plug them into our workstations. A problem no more! The first part of this week's lab will be getting you to communicate with your UART using a new PMOD expansion module.

1.1 PMOD serial interface

Recall that the Pynq-Z1 does not have an RS-232 serial interface. 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.

1.2 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. Yours truly's first attempt failed miserably upon 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`.

1.3 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.

2 I^{2}S 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 I^{2}S, is the name of the interface format used to communicate with the chip.

2.1 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 I^{2}S article to get a feel for what you're implementing.

The I^{2}S 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 lab will be to generate the three requisite clock signals for the I^{2}S 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 1 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.)

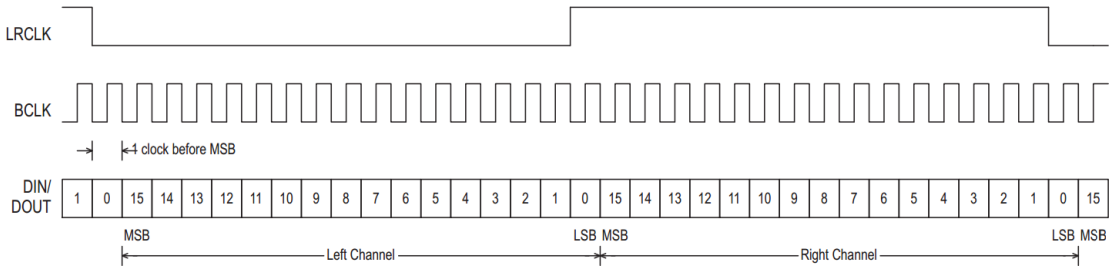

Figure 1: I^{2}S timing summary (credit: Texas Instruments)

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.

3 Conclusion + Checkoff

3.1 Checkoff Tasks

1. Show your TA that you can successfully type characters on the keyboard and have them echoed back to display on your `screen` session.
2. Demonstrate to your TA that your I^{2}S clocks waveforms match the requirements in the reference manual.