Lab 5A: Serial I/O

University of California, Berkeley

Department of Electrical Engineering and Computer Sciences

EECS150 Components and Design Techniques for Digital Systems

John Wawrzynek, James Parker, Daiwei Li

Due March 6th, 2013 @ 2:00PM

Table of Contents

0 Introduction

In this lab you will implement a UART (Universal Asynchronous Receiver / Transmitter) device, otherwise known as a serial interface. This task will tie together the tools you have learned in the past 5 labs and will provide a design experience similar to the course project. In addition, your working UART from this lab will be used in your project to talk to your local machine over a serial line.

Partners

You are permitted to work with a partner for this lab. You will be checking in your UART and associated files to the Git repository, you are required to work with your project partner.

1 PreLab

For the prelab, do the following:

1. Get the lab distribution from the class website.

```
wget http://inst.eecs.berkeley.edu/~cs150/sp13/lab5/lab5a.tar.gz
tar –zxvf lab5.tar.gz
```

2. Understand what a ready/valid handshake is. Read the tutorial posted on the website or Piazza. It may be helpful to draw out the timing diagrams for the ready/valid handshake to see exactly what is going on.

3. Read the Serial Device section. Make sure you understand how you are going to implement your part of the UART.

4. Lab lecture slides from last semester have also been posted on the website that may be useful. If you do use them, please disregard any of the logistics at the beginning. The section on Git subversion control will also be relevant later in the semester.

Serial Device

You are responsible only for implementing the **transmit** side of the UART for this lab. We have given you a complete receiving side. As you should have inferred from reading the ready/valid tutorial, the UART also uses a ready/valid interface to communicate over the serial line.

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

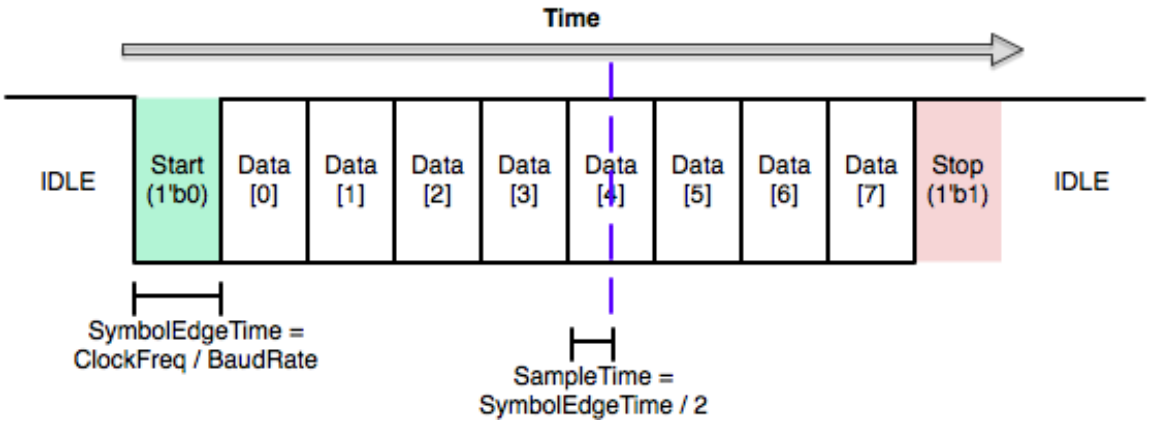
You would be wise to model your receiver solution after the design we have given you that implements the receive side because they will end up being quite similar.

2 Lab Procedure

Framing

On the ML505, the physical signaling aspects (such as voltage level) of the serial connection are taken care of by off-FPGA devices. From the FPGA's perspective, there are two signals, `FPGA_SERIAL_RX` & `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. The figure below shows a single character frame and will be extremely useful in understanding the protocol.



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. Notice that both sides must agree on a baudrate for this scheme to be feasible.

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 = (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.

Receiving

The receive side is a bit more complicated. Fortunately for you, it is already implemented and is given to you as part of the lab distribution. You may want to open up `lab5/src/UAReceive.v` and follow along as you are reading this section.

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 receive 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 should be low until we receive another character.

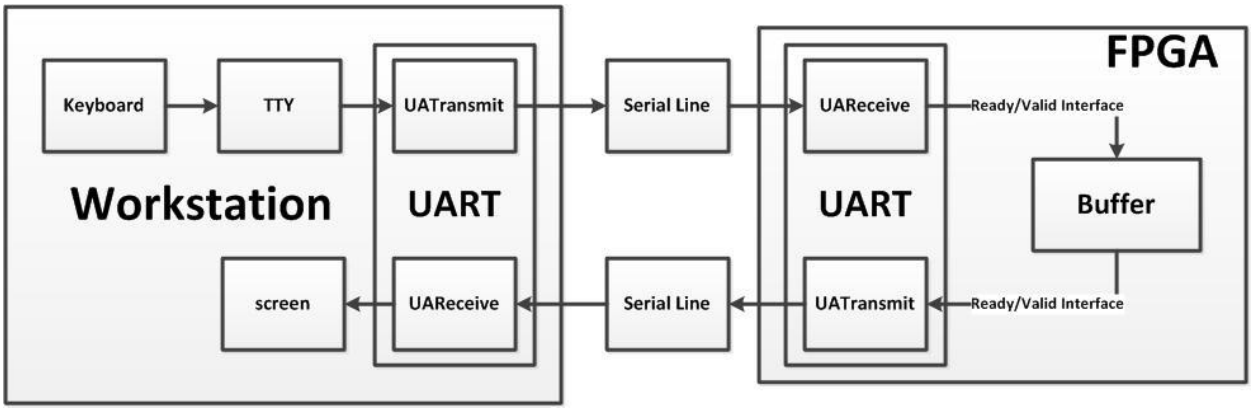
This requires using an extra flip-flop that is set when the last character is shifted in to the shift register and cleared when the ready signal is asserted. This allows us to correctly implement the ready/valid handshake.

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. If you look at `UART.v`, you will see that this module is mostly straightforward instantiations of `UAReceive` and `UATransmit`, but there are also two `IORegisters` that the serial lines are fed through. Think about why these might be there.

An `IORegister` is simply a register that attempts to pack itself into a special block called a `IOB`, which are used to drive and sense from the IO pins. Using an `IORegister` helps ensure that you will have a nice, clean, well-behaved off-chip signal to use as an input or output to your serial modules.

The following diagram shows how all of this is fits into the set up with the FPGA



Note that there is NO ready valid interface over the serial line.

Testing

You will use tools that you have learned in labs 2-4 to implement and test this lab. Make sure to refer back to these old labs if you have any questions about how to run any of these tools.

Simulation

We have provided a simple testbench, called `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 yourself.

Do to the way `x`'s are treated by Modelsim if a large number of 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 design of high Z and undefined X signals.

Echo

Your UART will eventually be used to interact with your CPU from your workstation. Without a CPU, you need some other way to test that your UART works on the board.

We have provided this for you. The provided `FPGA_TOP_ML505` contains a very simple finite state machine that simply sends ever character it receives back over the serial device. This will be extremely useful for testing. Before programming your board, check with the provided `EchoTestbench` module that everything works as it should in simulation.

Once you have echo working in simulation, it is time to try it on the board. Synthesize your design and program the board with it just like you have done in previous labs. Now, make sure the serial cable is plugged in between the ML505 and your workstation and then run:[1]

```
% screen $SERIALTTY 115200
```

This tells `screen`, a highly versatile terminal emulator, to open up the serial device with a baud rate of 115200. Now, if you have a properly working design, you should be able to tap a few characters into the terminal and have them displayed on the screen as they are echoed back to you.

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. Use `screen -x` to re-attach an improperly closed screen session.

Turn-in

Since you will be using the contents of this lab in your project, you will be turning this lab in via github. However, you are also required to have a TA check you off in addition to successfully committing and pushing the files to your github repository.

Setting Up SSH Keys

Github authenticates you for access to your repository using ssh keys. You can generate a ssh key using the `ssh-keygen` command. Accepting the defaults should be fine.

Once you have a key generated, go to your github account settings. Then go to the `SSH Public Keys` menu.

Here, click `Add another public key`. In this dialog insert the contents of your **public** key file in to the `Key` box.

By default the public key file will be in `~/.ssh/id_rsa.pub`.

Make sure you don't accidentally copy the contents of the `~/.ssh/id_rsa` file. This is your private key, which is equivalent to your password.

[1] Note that if someone else has locked the serial cable on your workstation, you will not be able to execute this command. Since we don't have sudo access, the only way to resolve this problem is to either find the offending user and have him kill his session, or reboot the computer.

After you paste your key in, give it a name and you are done.

Acquiring Skeleton Files, Setting Up Repository

The skeleton files for the project will be available through a git repository provided by the staff. The suggested way for initializing your repository with the skeleton files is as follows. First, set up your ssh keys as described above. Then execute the following commands:

```
% git clone git@github.com:EECS150/skeleton.git
% cd skeleton
% git remote add my-repo git@github.com:EECS150/teamX.git
% git push my-repo master
```

This will make a single commit to your repository with the base files, we suggest you then do the following:

```
% cd ..
% rm -rf skeleton
% git clone git@github.com:EECS150/teamX.git
% cd teamX
% git remote add staff git@github.com:EECS150/skeleton.git
```

Where the variable `X` in `teamX` denotes your two digit team number. Note that if you haven't emailed cs150-ta@imail.eecs.berkeley.edu with your account information you will not have a Git repository to turn in you assignment yet so do that ASAP.

These commands will delete the skeleton repository you cloned, clone your repository that now has a single commit, and add a remote repository named `staff` that points to the skeleton files to allow easy future merges of staff updates.

For the next part, if you are working with a partner, only one of you will need to do this. Here are commands to add the modules from this lab implementing UART to your git repository. Make sure to first change into the hardware `src` directory of the skeleton files. Similarly, you will want to add the `ALU` and `ALUdec` you wrote in Lab 4 to your repository.

```
% cp ~/lab5/src/UATransmit.v .
% git add UATransmit.v
% git commit -m "Adding UART transmit module"
% git push
```

The `git add` command tells your local git repository to add the file "`UATransmit.v`" to the list of tracked files or stage it for the next commit. The `git commit` command tells your local repository to actually commit any of the files you added and create a new change set with the changes in the files you added. Finally the `git push` command sends your new local commit to the remote repository so that you or your partner can pull it later.