EECS 151/251A FPGA Lab 5: FSMs and UART

Prof. Sophia Shao
TAs: Harrison Liew, Charles Hong, Jingyi Xu, Kareem Ahmad, Zhenghan Lin
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Before You Start This Lab

Run `git pull` in . Copy the modules you created in the previous lab to this lab:

```
cd fpga_labs_fa20
cp lab4/src/synchronizer.v lab5/src/.
cp lab4/src/debouncer.v lab5/src/.
cp lab4/src/edge_detector.v lab5/src/.
cp lab4/src/tone_generator.v lab5/src/.
cp lab4/src/music_streamer.v lab5/src/.
```

Review these documents:

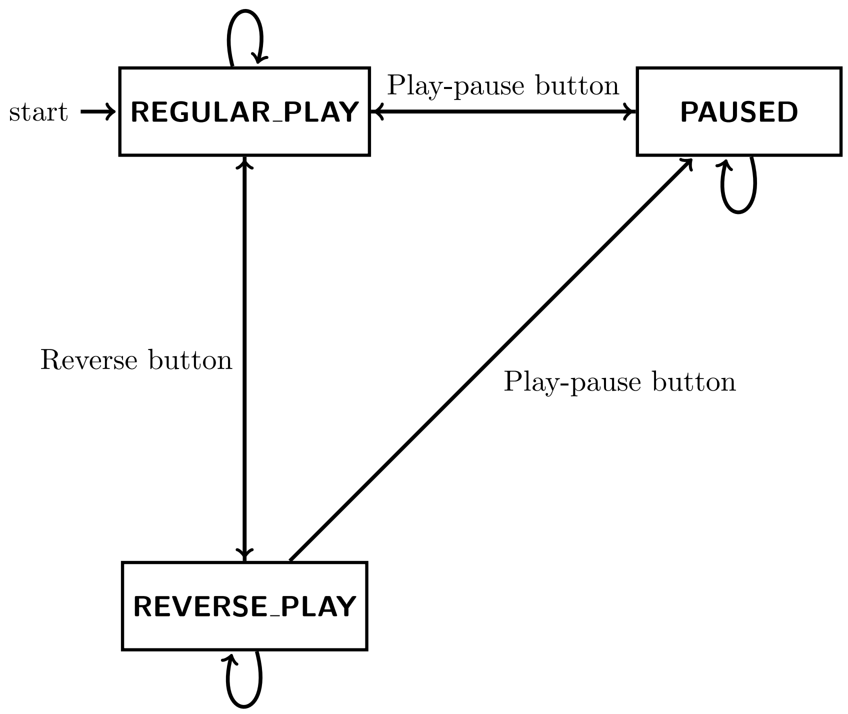
1. verilog_fsm.pdf - constructing FSMs in Verilog.
2. ready_valid.pdf - ready/valid interfaces and handshakes. Try drawing out the timing diagrams for the ready/valid handshake before trying to write any Verilog for the UART.

In the first part of this lab, you will extend the `music_streamer` from lab 4 to implement a simple FSM. Then 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 Music Streamer FSM

Implement a simple FSM in the `music_streamer`.

The FSM has 3 states: `PAUSED`, `REGULAR_PLAY`, `REVERSE_PLAY`. Here is the state transition diagram:



1. The initial state should be `REGULAR_PLAY`.
2. Pressing the `play_pause` button should transition you into the `PAUSED` state from either the `REGULAR_PLAY` or `REVERSE_PLAY` states. Pressing the same button while in the `PAUSED` state should transition the FSM to the `REGULAR_PLAY` state.
3. In the `PAUSED` state, the ROM address should be held steady at its value before the transition into `PAUSED` and no sound should come out of the speaker. After leaving the `PAUSED` state the ROM address should begin incrementing again from where it left off and the speaker should play the tones.
4. You can toggle between the `REGULAR_PLAY` and `REVERSE_PLAY` states by using the `reverse` button. In the `REVERSE_PLAY` state you should decrement the ROM address by 1 rather than incrementing it by 1 every X clock cycles as defined by the tempo.
5. If you don't press any buttons, the FSM shouldn't transition to another state.

The `music_streamer` takes in user button inputs (`play_pause, reverse`) that it can use to transition states.

Also, drive the `leds` output so that they show the current state as such:

LED	**Value**
leds[0]	`current_state` == `REGULAR_PLAY`
leds[1]	`current_state` == `PAUSED`
leds[2]	`current_state` == `REVERSE_PLAY`

Look at the commented code in `music_streamer_testbench.v` that tests the state machine. Uncomment and run the test and inspect the waveform to check that the FSM is performing correctly.

Next, modify `z1top.v` to connect the `play_pause` and `reverse` ports to `buttons_pressed` [2] and [3] respectively, and the bottom 3 bits of `LEDS` to the `leds` output of the `music_streamer`. You can copy the instantiations of the `tone_generator` and `music_streamer` from `z1top.v` from lab 4, but leave the top-level IOs in the skeleton untouched.

Finally, program the FPGA with `make impl` and `make program HW_SERVER_PORT=<PORT NUMBER>`, and verify the functionality of the FSM.

3 UART Serial Device

You are responsible only for implementing the **transmit** side of the UART for this lab (`lab5/src/uart_transmitter.v`). 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.

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 (`uart_transmitter` and `uart_receiver`) that have the ready/valid handshake for interfacing with other 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.

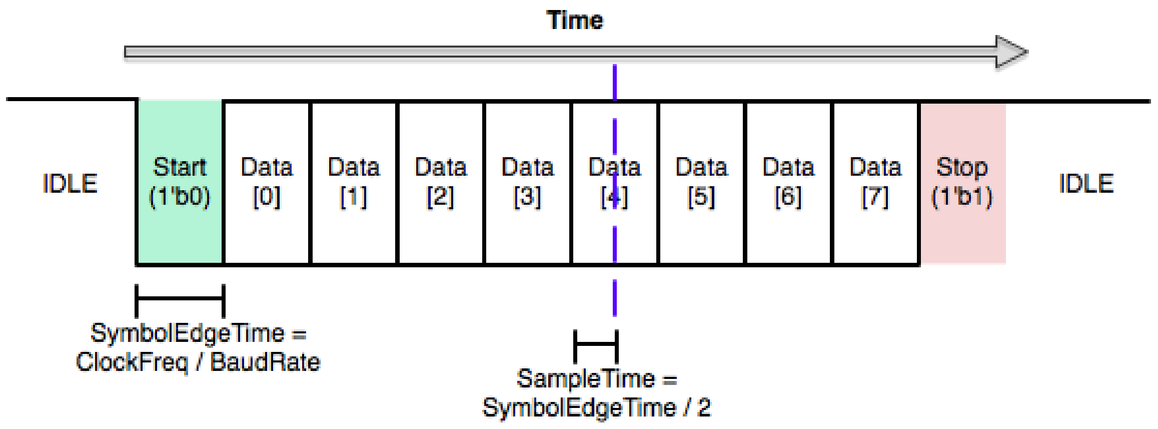


Figure 1: UART Frame Structure

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 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's 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 have to send another frame immediately after.

3.3 Receiving

The receive side is a bit more complicated. Fortunately, we will provide the receiver module. Open `lab5/src/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 likely to sample on the wrong side of the edge and get the wrong value for that bit. One 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 the `uart_receiver` 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:

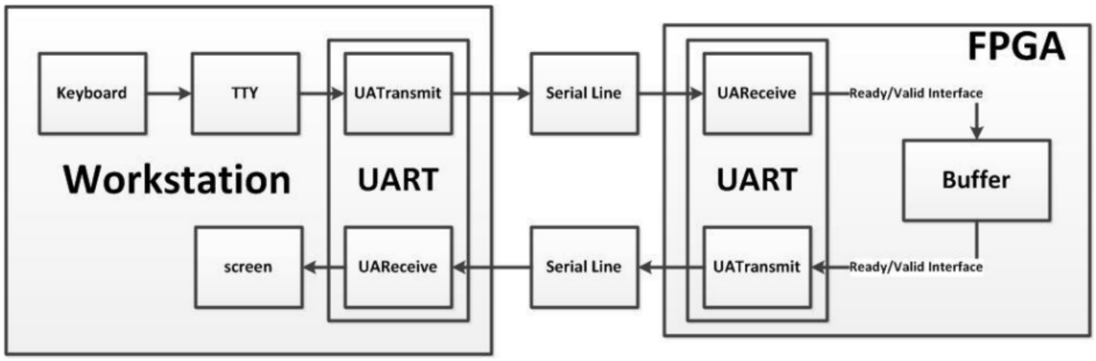


Figure 2: High Level Diagram

3.5 Simulation

We have provided a simple testbench, in `sim/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 talk to each other. 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 should add that case yourself. 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.

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 `src/z1top.v` 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 commented to help you understand the communication between the 2 UARTs and the communication over the ready/valid interface. The test 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 USB-UART

The Pynq-Z1 does not have an RS-232 serial interface connected to the FPGA fabric. So we'll be using the Pmod USB-UART extension module to add a UART interface to the Pynq. Connect the

PMOD module to the **top** row of the PMOD A port on the Pynq, and connect a USB cable from the USB-UART PMOD to your computer.

Note: Make sure that the power selection jumper on the Pmod USBUART is set to LCL3V3

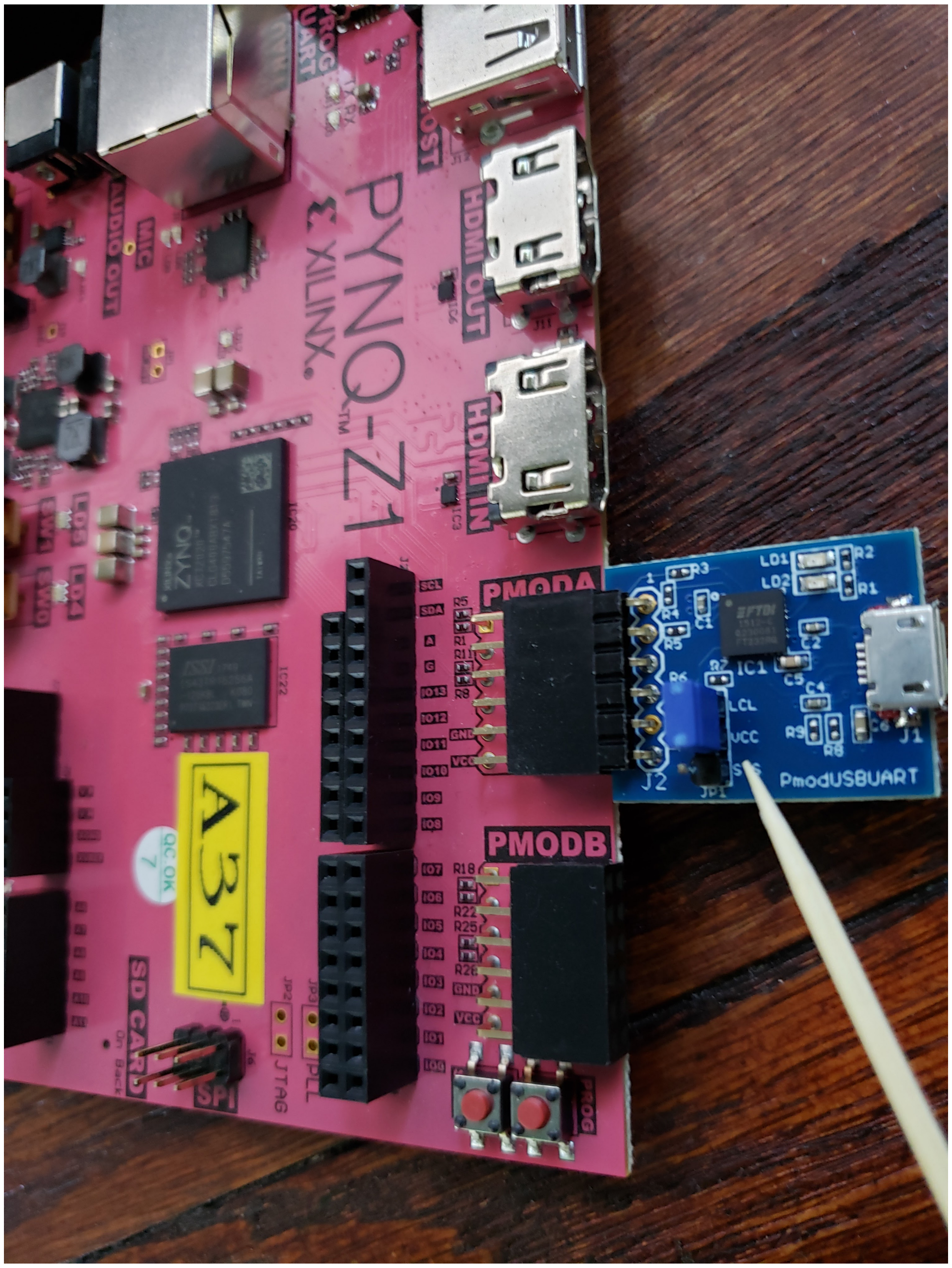


Figure 3: PMOD USBUART plugged in w/ correct power jumper setting (blue).

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 in `build/synth/synth.log`. 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 a USB cable is plugged in between the PMOD module and your computer (you can disconnect the programming cable and use that if you don't have an extra, so long as you don't turn off the fpga).

4.3.1 VM Setup

Since the fpga is connected to your local machine, we will use the VM to communicate with the fpga over UART. You will need to install `screen` on your VM if it is not already installed.

```
sudo yum install screen
```

4.3.2 Connecting

In your vm, in a new terminal, run `dmesg`. You should get a result that looks like this:

```
[7444636.941491] ftdi_sio 1-2:1.0: FTDI USB Serial Device converter detected
[7444636.941621] usb 1-2: Detected FT232RL
[7444636.942062] usb 1-2: FTDI USB Serial Device converter now attached to ttyUSB0
```

Now you can connect with

```
sudo screen /dev/ttyUSB0 115200
```

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.

If you have a working design, you can type 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 on your workstation.

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. Alternatively you can use `pkill screen` if the first option doesn't work. Then run `screen` again.

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

4.4 Personal Laptop Instructions

4.4.1 Linux/OSX

Same instructions as the VM (4.3.1).

4.4.2 Windows

After plugging in the USB cable, you may be prompted to install the FTDI drivers, so do that. Follow the steps from here to use PuTTY to connect to the UART.

5 Checkoff

1. Demonstrate the music streamer on the FPGA (FSM with play, pause, and reverse play / tempo control / reset)
2. Go through the UART simulation results and show that your UART behaves as expected. What do the testbenches do?
3. Demonstrate that you can type characters rapidly on the keyboard and have them echoed back in your `screen` session

5.1 Lab Report

No lab report!

Ackowlegement

This lab is the result of the work of many EECS151/251 GSIs over the years including:

- Sp12: James Parker, Daiwei Li, Shaoyi Cheng
- Sp13: Shaoyi Cheng, Vincent Lee
- Fa14: Simon Scott, Ian Juch
- Fa15: James Martin
- Fa16: Vighnesh Iyer
- Fa17: George Alexandrov, Vighnesh Iyer, Nathan Narevsky
- Sp18: Arya Reais-Parsi, Taehwan Kim
- Fa18: Ali Moin, George Alexandrov, Andy Zhou
- Fa19: Vighnesh Iyer, Rebekah Zhao, Ryan Kaveh
- Fa20: Charles Hong, Kareem Ahmad, Zhenghan Lin