EECS 151/251A FPGA Lab 6: FIFOs, UART Piano

Prof. Borivoje Nikolic and Prof. Sophia Shao
TAs: Cem Yalcin, Rebekah Zhao, Ryan Kaveh, Vighnesh Iyer
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

1 Introduction

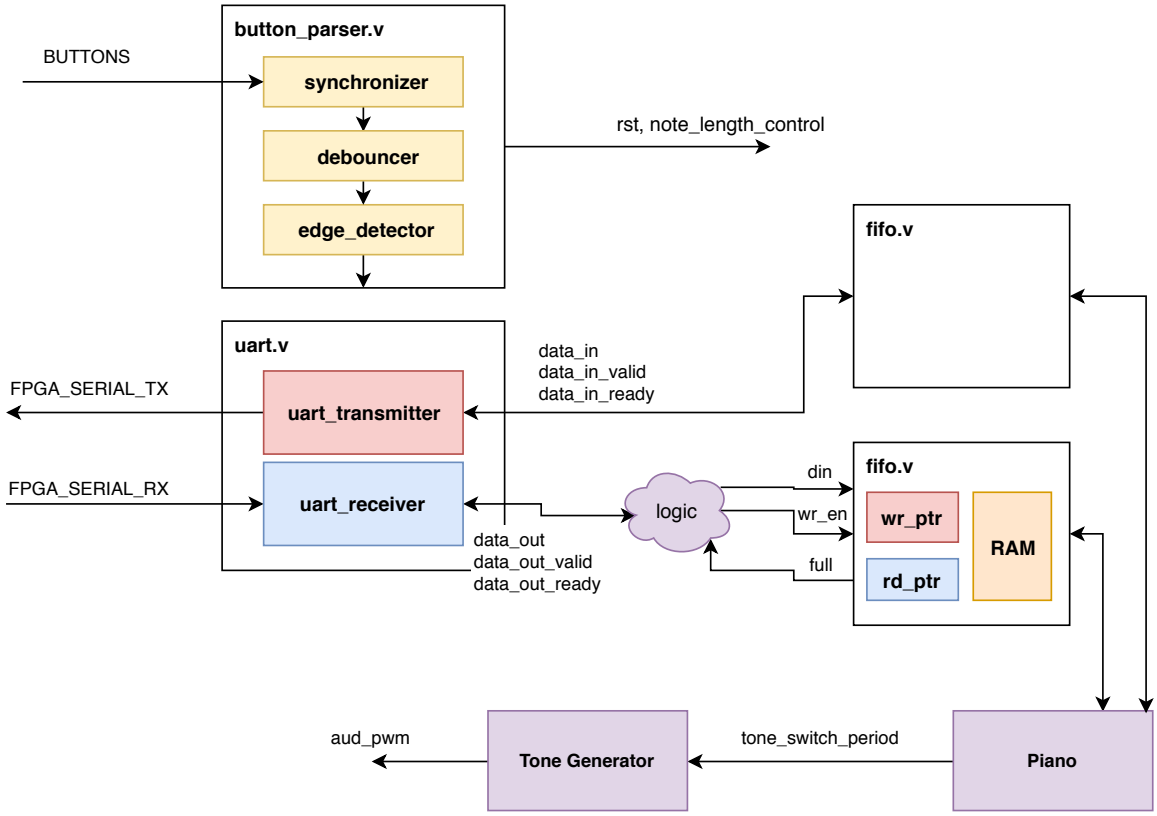
Run `git pull` in `fpga_labs_fa19`. Copy the modules you created in the previous lab to this lab:

```
cd fpga_labs_fa19
cp lab5/src/synchronizer.v lab6/src/.
cp lab5/src/debouncer.v lab6/src/.
cp lab5/src/edge_detector.v lab6/src/.
cp lab5/src/tone_generator.v lab6/src/.
cp lab5/src/music_streamer.v lab6/src/.
cp lab5/src/uart_transmitter.v lab6/src/.
```

1.1 Overview

In this lab, you will integrate the UART created in Lab 5 and the tone generator from the previous labs to create a "piano". In the first part, you will build a synchronous FIFO and verify its functionality using a block-level testbench. Then you will use a FIFO to bridge the UART with an FSM that plays notes with a fixed duration send from your computer. You will extend the FSM to support playing variable length notes. In the final part, you will extend the tone generator's volume control capability and implement a note amplitude envelope to resemble a real instrument.

Here is an overview of the entire system in `z1top` we are going to build. You may find it useful to refer to this block diagram while doing this lab.



2 Buliding a Synchronous FIFO

A FIFO (first in, first out) data buffer is a circuit that allows data elements to be queued through a write interface, and read out sequentially by a read interface. The FIFO we will build in this section will have both the read and write interfaces clocked by the same clock; this circuit is known as a synchronous FIFO.

2.1 FIFO Functionality

A FIFO is implemented with a circular buffer (2D reg) and two pointers: a read pointer and a write pointer. These pointers address the buffer inside the FIFO, and they indicate where the next read or write operation should be performed. When the FIFO is reset, these pointers are set to the same value.

When a write to the FIFO is performed, the write pointer increments and the data provided to the FIFO is written to the buffer. When a read from the FIFO is performed, the read pointer increments, and the data present at the read pointer's location is sent out of the FIFO.

A comparison between the values of the read and write pointers indicate whether the FIFO is full or empty. You can choose to implement this logic as you please. The `Electronics` section of the

FIFO Wikipedia article will likely aid you in creating your FIFO.

Here is a block diagram of the FIFO you should create from page 103 of the Xilinx FIFO IP Manual.

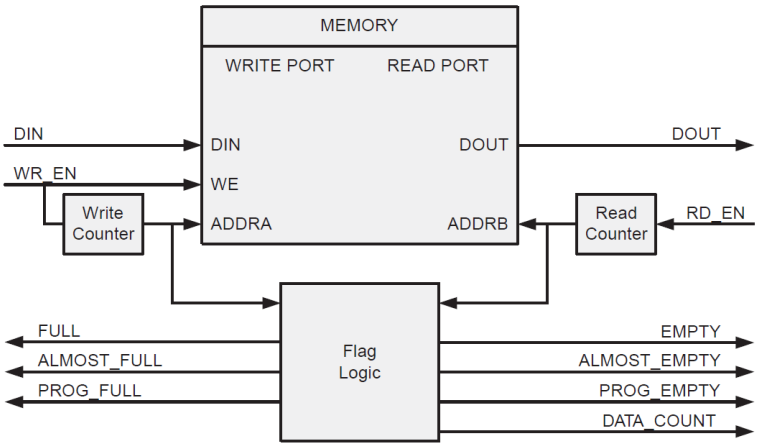


Figure 5-4: **Functional Implementation of a Common Clock FIFO using Block RAM or Distributed RAM**

The interface of our FIFO will contain a subset of the signals enumerated in the diagram above.

2.2 FIFO Interface

Look at the FIFO skeleton in `src/fifo.v`.

The FIFO is parameterized by:

- `data_width` - The number of bits per entry in the FIFO
- `fifo_depth` - The number of entries in the FIFO.
- `addr_width` - Calculated as the width of the read and write pointers.

The common FIFO signals are:

- `clk` - Clock used for both read and write interfaces of the FIFO.
- `rst` - Reset (synchronous with `clk`); should force the FIFO to become empty.

The FIFO write interface consists of:

- `input wr_en` - When this signal is high, on the rising edge of the clock, the data on `din` should be written to the FIFO.
- `input [data_width-1:0] din` - The data to be written to the FIFO.
- `output full` - When this signal is high, the FIFO is full.

The FIFO read interface consists of:

- `input rd_en` - When this signal is high, on the rising edge of the clock, the FIFO should

output the data indexed by the read pointer on `dout`

- `output [data_width-1:0] dout` - The data that was read from the FIFO after the rising edge on which `rd_en` was asserted
- `output empty` - When this signal is high, the FIFO is empty.

2.3 FIFO Timing

The FIFO that you design should conform to the specs above. To further, clarify here are the read and write timing diagrams from the Xilinx FIFO IP Manual. These diagrams can be found on pages 105 and 107. Your FIFO should behave similarly.

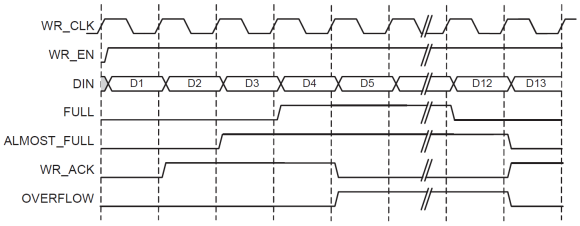


Figure 5-6: **Write Operation for a FIFO with Independent Clocks**

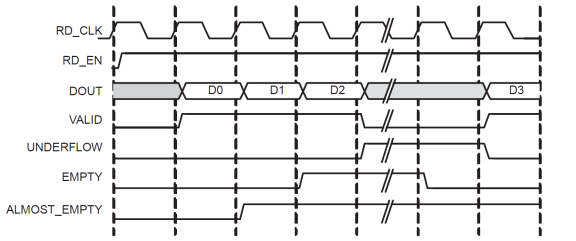


Figure 5-7: **Standard Read Operation for a FIFO with Independent Clocks**

Your FIFO doesn't need to support the `ALMOST_FULL`, `WR_ACK`, or `OVERFLOW` signals on the write interface and it doesn't need to support the `VALID`, `UNDERFLOW`, or `ALMOST_EMPTY` signals on the read interface.

2.4 FIFO Testing

We have provided a testbench in `sim/fifo_testbench.v`.

The testbench performs the following test sequence:

1. Checks initial conditions after reset (FIFO not full and is empty)
2. Generates random data which will be used for testing
3. Pushes the data into the FIFO, and checks at every step that the FIFO is no longer empty
4. When the last piece of data has been pushed into the FIFO, it checks that the FIFO is not empty and is full
5. Verifies that cycling the clock and trying to overflow the FIFO doesn't cause any corruption of data or corruption of the full and empty flags
6. Reads the data from the FIFO, and checks at every step that the FIFO is no longer full
7. When the last piece of data has been read from the FIFO, it checks that the FIFO is not full and is empty
8. Verifies that cycling the clock and trying to underflow the FIFO doesn't cause any corruption

of data or corruption of the full and empty flags

9. Checks that the data read from the FIFO matches the data that was originally written to the FIFO

10. Prints out test debug info

This testbench tests one particular way of interfacing with the FIFO. Of course, it is not comprehensive, and there are conditions and access patterns it does not test. We recommend adding some more tests to this testbench (or writing a new testbench) to verify your FIFO performs as expected. Here are a few tests to try:

- Several times in a row, write to, then read from the FIFO with no clock cycle delays. This will test the FIFO in a way that it's likely to be used when buffering user I/O.
- Try writing and reading from the FIFO on the same cycle. This will require you to use fork/join to run two threads in parallel. Make sure that no data gets corrupted.

3 Building the Piano

The piano interfaces the UART with the tone generator. Its job is to fetch notes sent from the UART, convert them to a `tone_switch_period`, and send them to the `tone_generator` for a fixed note length. This is the 'piano' module in the diagram in the lab intro.

Fill in `src/piano.v`. It has access to the UART transmitter FIFO, the UART receiver FIFO, and the `tone_generator` inputs. It also has access to a reset signal and the other buttons. The piano should implement the following functionality:

- When the UART receiver FIFO contains a character, the FSM should pull the character from the FIFO and echo it back without modification through the UART transmitter FIFO.
- Once a character is pulled, its corresponding `tone_switch_period` should be read from the supplied `piano_scale_rom.v`.
- For a fixed amount of time (`note_length`), the tone should be played by sending it to the `tone_generator`.
- The `note_length` should default to 1/5th of a second, and can be changed by a fixed amount with the buttons. Let `buttons[0]` increase/decrease the `note_length` depending on the value of `switches[0]`. This is similar to the runtime tempo modification you implemented in the `music_streamer`.
- Through doing all of this, your FSM should take care to ensure that if a FIFO is full, that it waits until it isn't full before pushing through data.
- You can use the `leds` output for debugging

You don't need to design the `piano` as an explicit FSM with states; the design is entirely up to you.

A ROM containing mappings from ASCII character codes to the `tone_switch_period` of the note to be played can be found in `src/piano_scale_rom.v`. If you wish to re-generate this file, use these commands:

```
cd lab6
python scripts/piano_scale_generator.py scale.txt
python scripts/rom_generator.py scale.txt src/piano_scale_rom.v 256 24
# Modify piano_scale_rom.v to change the module name to 'piano_scale_rom'
```

It is possible that the UART receiver FIFO can fill up with samples so fast that the piano can't keep up; similar overflow conditions are possible with other parts of this system. You don't need to concern yourself with detecting 'backpressure' on the entire system and can just assume that your FIFOs are large enough to buffer all the user input.

3.1 Modify z1top

Open up `z1top.v` and modify it to include the new modules you wrote. Wire up the FIFOs and your piano + tone generator according to the block diagram in the lab intro. You will have to add a few lines of logic (purple cloud) representing the bridge between the ready/valid interface and the FIFO's `rd_en, wr_en, full, empty` interface.

Make sure that you parameterize your FIFOs properly so that they have the proper `data_width`. You can make your FIFOs as deep as you want, but 8 should be enough.

4 Writing a System-Level Testbench

This design involves quite a few moving parts that communicate with each other. We want to make sure that the complete integration of our system works as expected. To that end, you will have to write a system-level testbench that stimulates the top-level of your design and observes the top-level outputs to confirm correct behavior.

We have provided a template for a system testbench in `sim/system_testbench.v`. Fill in the `initial` block to test all the parts of the piano.

To make the waveform shorter and easier to debug, it is suggested that you change the `note_length` default value in the `piano` to something much smaller than 1/5th of a second, just for testing.

5 FPGA Testing

Generate a bitstream and program the FPGA as usual. Read the synthesis and implementation reports (`build/synth/synth.log`) to see if there are any unexpected warnings.

You should watch out specifically for warnings like "found x-bit latch" or "x signal unconnected" or "x signal assigned but never used". If you see that the synthesis tool inferred a latch, you should definitely fix that warning by completing any if-elseif-else or case statements that don't have a default signal assignment. The other 2 warning types are dependent on your design and you should ignore them only if you know they are expected.

Once you put your design on the FPGA you can send data to the on-chip UART by using `screen $SERIALTTY 115200`. The piano keys are mapped such that 'z' through '<' go from C3 to C4 (middle C) and 'q' through 'i' go from C4 to C5. Holding down shift moves the lower row's octave down and the upper row's octave up.

You can use the 1st button to change the `note_length` of your piano. You should test the case where you make the `note_length` long, and fill up your UART FIFO by typing really fast. Then watch your FIFO drain slowly as each note is played for `note_length` time.

For Checkoff: Save the bitstream of the fixed note length piano by copying the `build` directory, so it can be used during checkoff:

```
cp -r build build_fixed
```

6 Variable Note Lengths

We would like to set the length of each note based on how long each key is pressed, instead of having a fixed note length. We have provided a Python script that captures keyboard keyup/keydown events and translates them into 2 UART frames (a packet) indicating whether a note should begin playing or finish playing:

- A key press causes 2 UART frames to be sent in order: `0x80, character` where `character` is the key that was pressed
- A key release causes 2 UART frames to be sent in order: `0x81, character` where `character` is the key that was released

Once you get a key press packet, you should begin playing the note associated with `character`, similar to before. If you get a key release packet that has a matching `character` with the note being played, you should stop playing it.

If you're playing a note and you get key press packet, you should throw it away. If you're playing a note and get a key release packet for a different `character`, you should throw it away.

You don't have to drive any characters into the UART TX in this part. Drive an LED indicating when a note is being played.

6.1 Testing

Modify the `system_testbench` to test variable length note playing, and test the edge cases where multiple keys are pressed before they are released.

Program it on the FPGA as usual. Then run `python3 scripts/piano.py` which captures keyboard events and sends them to the FPGA via UART. You can play the same notes as beforehand.

6.1.1 Testing Locally

First, install some python packages (`pip3 install pynput pyserial`).

If you're using Linux or OSX, you may have to launch the `piano.py` script with `sudo`. On Ubuntu, you may have to run this line first if you get an X11 error (`xhost si:localuser:root`).

If you're on Windows, in addition to installing the Python packages, you should modify the `scripts/piano.py` file (line 10) with the COM port assigned to the USB-UART device (according to the Device Manager).

7 Checkoff Tasks

1. Show the system-level testbench you wrote and its methodology for testing your piano's functionality.
2. Show the output waveform of your testbench and explain how data moves through your system.
3. For the fixed note length piano, prove the existence of your UART RX and TX FIFOs by making the note length long, then pressing multiple keys in quick succession, filling the RX FIFO and, then seeing the notes drain out slowly into the piano.
4. Demonstrate the variable note length piano working on the FPGA.

7.1 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