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

Prof. John Wawrzynek
TAs: Christopher Yarp, Arya Reais-Parsi
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

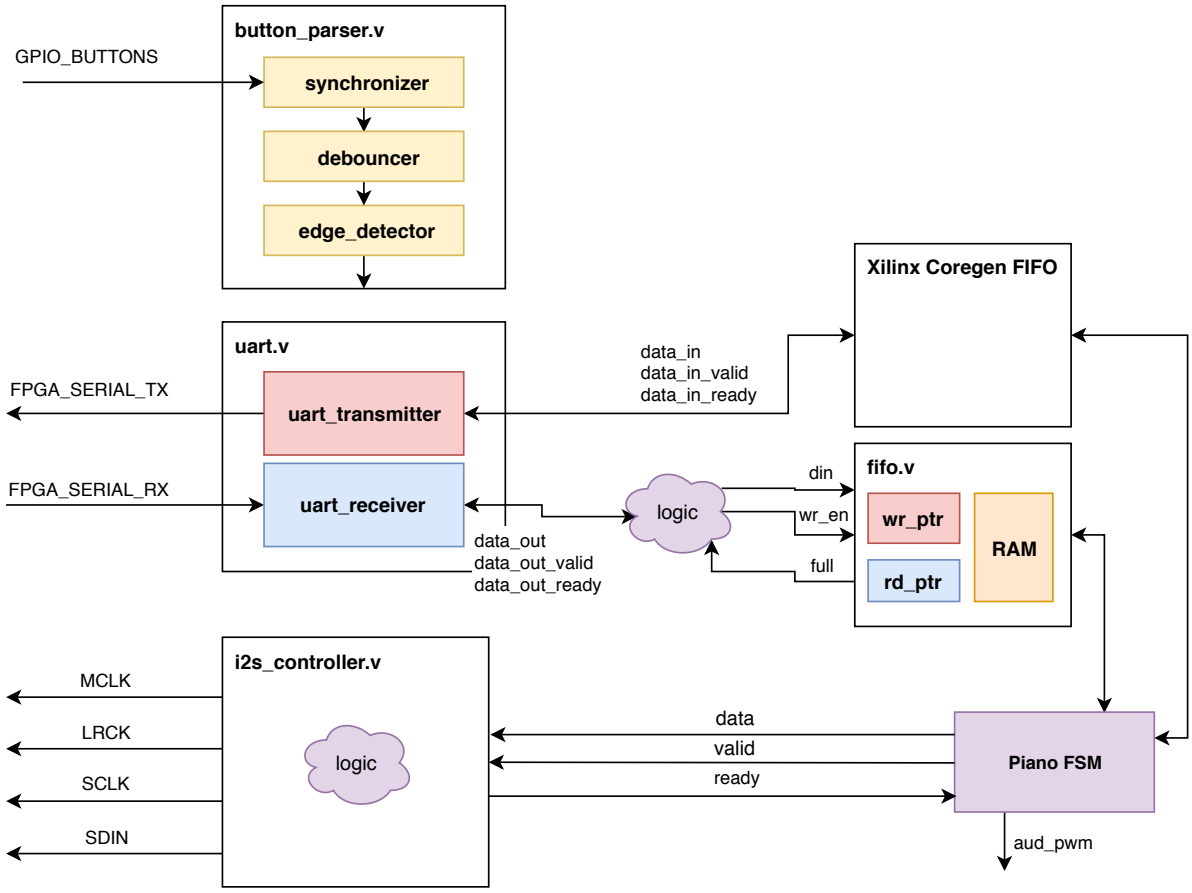
Contents

1 Introduction

In this lab, you will integrate the components you created in Lab 5 (UART and incomplete I^{2}S controller). You will begin by building a synchronous FIFO and verifying its functionality using a

block-level testbench. You will then finish your I^{2}S controller to emit a signal using the FIFO as its PCM data source. Finally, you will create some logic that integrates all these components to form a "piano".

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



In this lab, you will be building the FIFOs, modifying your I^{2}S controller to use one, and designing the piano FSM. You will then construct a system-level testbench to verify functionality in simulation.

1.1 Copying Files From Previous Labs

You will need the following files from previous labs:

```
uart.v
uart_receiver.v
uart_transmitter.v
```

```
tone_generator.v
synchronizer.v
debouncer.v
edge_detector.v
i2s_controller.v
```

2 Building a Synchronous FIFO

A FIFO (first in, first out) data buffer is a circuit that has two interfaces: a read side and a write side. The FIFO we will build in this section will have both the read and write side 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.

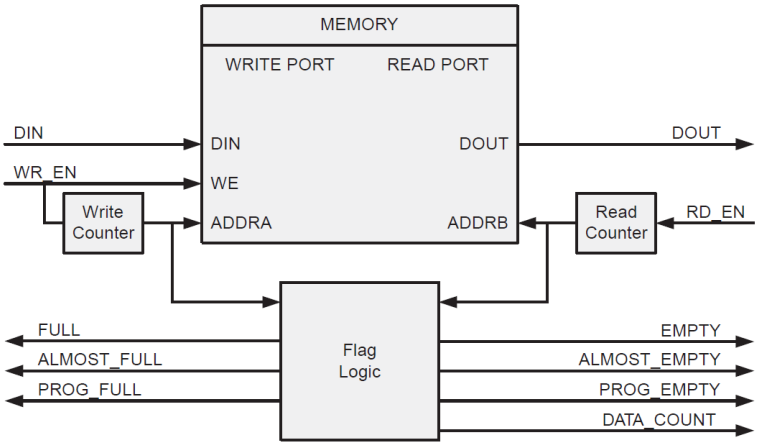


Figure 5-4: **Functional Implementation of a Common Clock FIFO using Block RAM or Distributed RAM**

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

2.2 FIFO Interface

Take a look at the FIFO skeleton in `fifo.v`.

Our FIFO is parameterized by these parameters:

- `data_width` - This parameter represents the number of bits per entry in the FIFO.
- `fifo_depth` - This parameter represents the number of entries in the FIFO.
- `addr_width` - This parameter is automatically filled by the log2 macro to be the number of bits for your read and write pointers.

The common FIFO signals are:

- `clk` - Clock used for both read and write interfaces of the FIFO.
- `rst` - Reset synchronous to the clock; should cause the read and write pointers to be reset.

The FIFO write interface consists of three signals:

- `wr_en` - When this signal is high, on the rising edge of the clock, the data on `din` will be written to the FIFO.
- `[data_width-1:0] din` - The data to be written to the FIFO should be present on this net.
- `full` - When this signal is high, it indicates that the FIFO is full.

The FIFO read interface consists of three signals:

- `rd_en` - When this signal is high, on the rising edge of the clock, the FIFO should present the data indexed by the read pointer on `dout`
- `[data_width-1:0] dout` - The data that was read from the FIFO after the rising edge on which `rd_en` was asserted
- `empty` - When this signal is high, it indicates that 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.

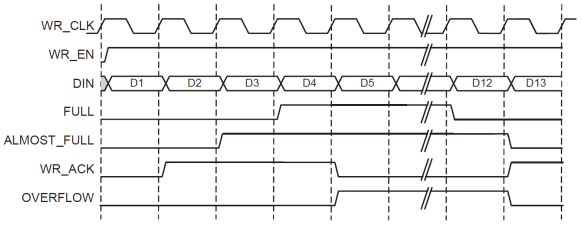


Figure 5-6: **Write Operation for a FIFO with Independent Clocks**

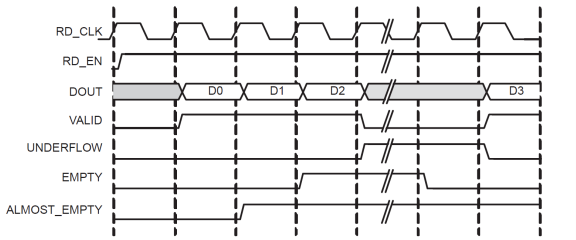


Figure 5-7: **Standard Read Operation for a FIFO with Independent Clocks**

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 for your synchronous FIFO which can be found in `fifo_testbench.v`. This testbench can test either the synchronous or the asynchronous FIFO you will create later in the project. To change which DUT is tested, comment out or reenable the defines at the top of the testbench (`SYNC_FIFO_TEST`, `ASYNC_FIFO_TEST`).

You can test this in the Vivado Design Suite or with ModelSim through the `sim` directory, as in previous labs.

The testbench we have provided performs the following test sequence, which you should understand well.

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 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.

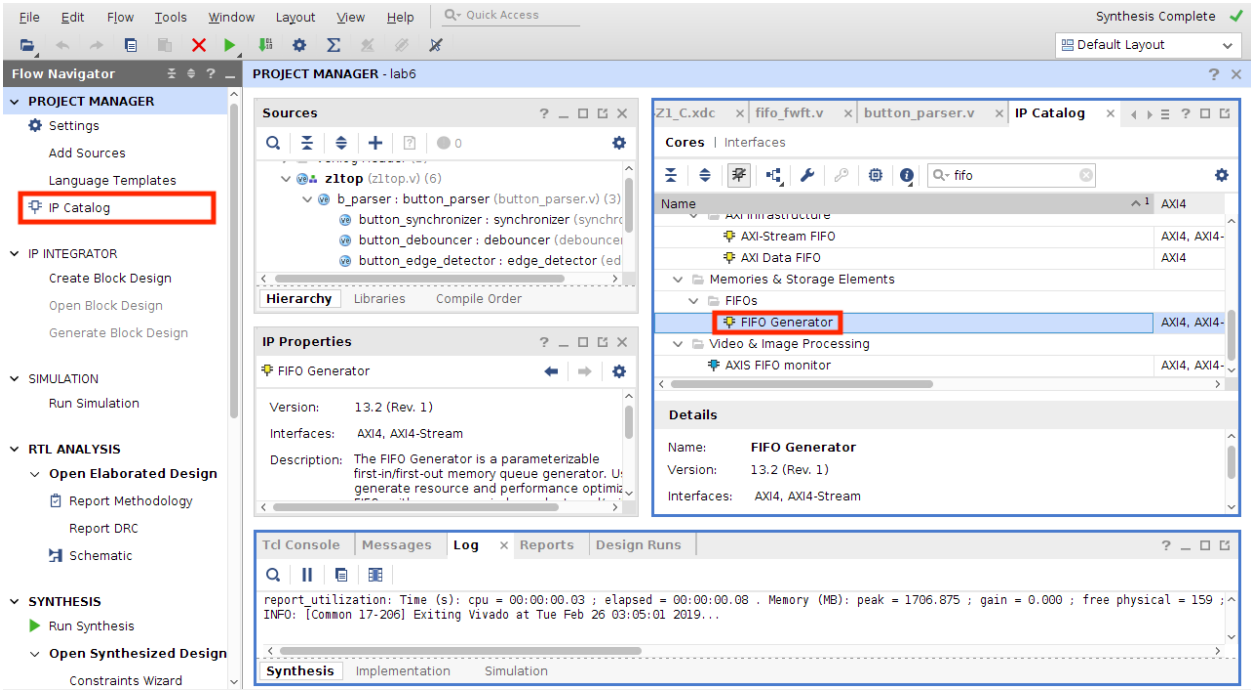
3 Using the Xilinx FIFO Generator

Up to this point, you have been using modules that were either written by yourself from scratch or were provided to you by the teaching staff. In practice, there is an alternative for modules that are either extremely common or heavily specialized: IP Cores. IP (Intellectual Property) cores are hardware modules that have been written by third parties to be used in a variety of customer designs. IP Cores can range from very common components, such as FIFOs, to advanced and specialized components, such as AES Encryptors/Decryptors. The goal behind IP cores is similar to vendor optimized libraries in software: they accelerate development by allowing the design reuse of a component written by an expert third party.

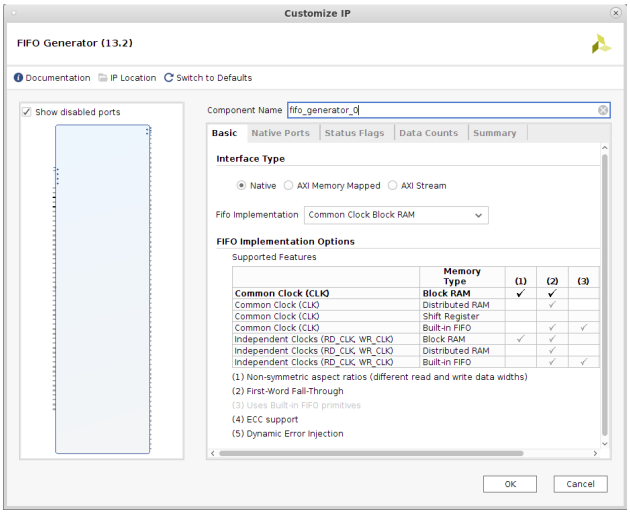
In this lab section, you will be creating an instance of the FIFO IP core provided by Xilinx.

First, you will need to open the FIFO Generator IP to configure your instance:

1. In the *Flow Navigator* side pane, click on *Project Manager* → *IP Catalog*
2. In the *IP Catalog* window, locate the *FIFO Generator* under *Memories & Storage Elements* → *FIFOs*
3. Double click on *FIFO Generator*



This will open an interface for customizing your FIFO.



The first thing you should do is give your FIFO a name. Enter this into the *Component Name* field at the top of the *Customize IP* window.

The Xilinx provided FIFO generator is written to work with many design. As such, it offers many different configurations options. For our purposes, we are interested in instantiating an 8 bit wide "First-Word Fall-Through" FIFO. "First-Word Fall-Through" means that the data at the head of the FIFO is immediately presented on the data output lines. By treating the `empty` signal as `!valid` and `read_en` as `ready`, the read interface of a First-Word Fall-Through FIFO conforms to the ready/valid semantic we have been using in other modules.

We also need to select the depth of our FIFO. For this example, let's say that our FIFO is 256 entries deep.

Now, let's configure the FIFO. In the *Basic* tab:

1. Select *Native* as the *Interface Type*[1]
2. Select *Common Clock Block RAM* as the *FIFO Implementation*[2]

Under the *Native Ports* tab:

1. Select *First-Word Fall-Through* as the *Read Mode*
2. Set the *Write Width* to 8
3. Set the *Write Depth* to 256
4. Set the *Read Width* to 8
5. Make sure *Reset Pin* is checked and *Reset Type* is *Synchronous Reset*

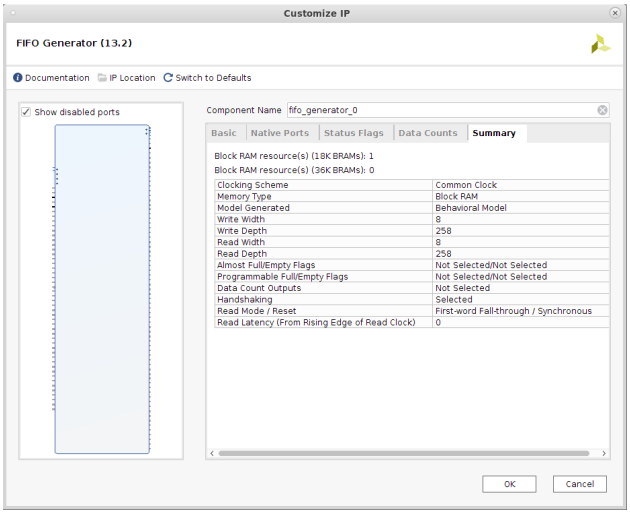
[1]We could have also selected *AXI Stream* but the native interface provides us with some additional flexibility. We are also not using all of the features of AXI Stream.

[2]We could also create a clock crossing FIFO by selecting *Independent Clocks Block RAM*. This would create separate clock inputs for the write and read sides of the FIFO. This allows us to easily pass data between clock domains.

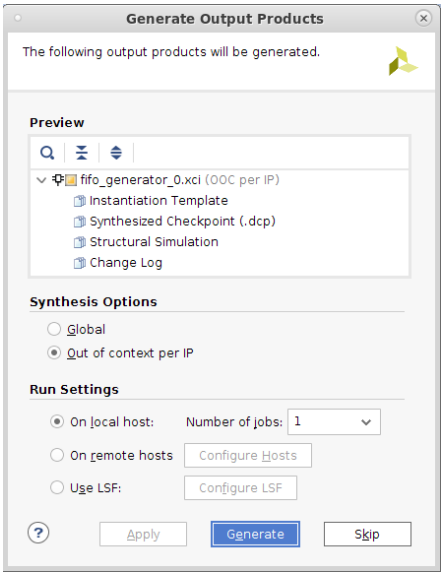
Under the *Status Flags* tab:

1. Check *Valid Flag* and select *Active High*

If you open the *Summary* tab, your *Customize IP* window should look close to this:



Once you are satisfied with your configuration, click *OK*. Vivado will then prompt you with a window asking how you would like to generate the IP core.

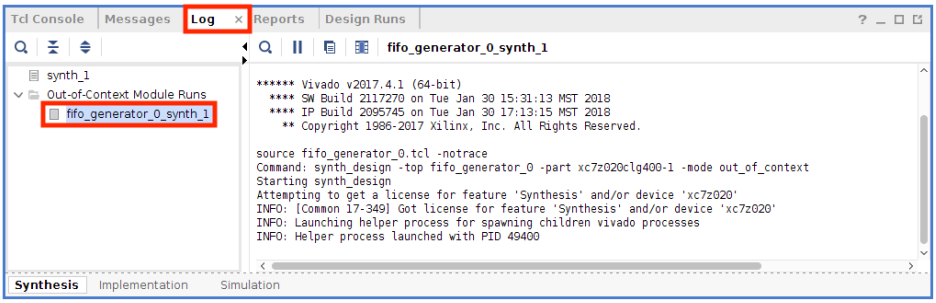


You have 2 main options: *Global* or *Out of Context*. Global will create the FIFO configuration files and generate a template file. However, the FIFO will be synthesized as part of your design's overall synthesis task. Out of context runs allow synthesis to be performed for the FIFO separately. The result of this synthsis is then used when synthesizing your overall design. The advantage of

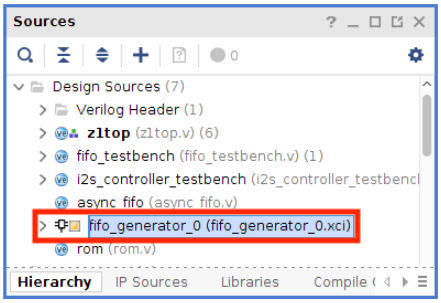
out of context runs is that the FIFO does not need to be re-synthesized each time your design is re-synthesized.

Select *Out of context per IP* and click *Generate*.

Vivado will now begin the out of context run for the FIFO. You can observe the progress by clicking on the *Log* tab in the bottom panel. Note that there is now a sub-pane displaying 2 logs for synthesis: synth_1 and one for fifo_generator_0_synth_1.



The FIFO should also appear in your design hierarchy.



You should now be able to use the FIFO in your design. Durring the generation process, Vivado creates a template file which shows how to integrate the FIFO into your design. If you named your FIFO fifo_generator_0_synth_1, the template will be in `lab6/lab6.srcs/sources_1/ip/fifo_generator_0/fifo_generator_0.veo`.

Also, if you are using git, you should add the new files in `lab6/lab6.srcs/sources_1/ip`.

4 Finishing the I^{2}S Controller

Now that you can generate the various clocking signals for the I^{2}S controller, it's time to extend it to accept and output actual PCM data. We need to do this before we can plug it into the FSM!

To make things simple, we will initially use the I^{2}S controller as a glorified PWM output with the same PWM signal you used to drive audio out in labs 3 and 4. The "high" PWM signal will correspond to the maximum PCM value for a given sample width (bit depth). The "low" PWM signal will correspond to the minimum PCM value. Recalling your reading in Lab 6, the I^{2}S chip

accepts twos-complement-signed PCM with our choice of bit width, so if our sample width was 20 bits:

- the maximum PCM value would be 0x7FFFF;
- the minimum PCM value would be 0x80000.

We need to add a data input bus to connect to the Piano FSM.

Modify your I^{2}S controller to accept one piece of data from the Piano FSM for each frame. Remember that in the I^{2}S protocol, frames are sent starting from the *second* bit clock period after each left-right clock transition. (Refer to the Lab 6 resources if you need.). A two bit ready signal and a two bit valid signal are provided in the template for the I^{2}S module. One is for the left channel while the other is for the right channel.

You are not required to implement the ability for the I^{2}S controller to buffer several PCM values to be sent out. Since the I^{2}S must constantly send serial audio data, its opportunities for accepting new PCM values are limited. Remember that, as the I^{2}S controller author, you are able to bring the ready signal high whenever you can accept new data and are also able to bring it low when you can no longer accept data. In the case when no valid PCM data is available when you need it during frame transmission, you should continue sending the last value that you had received. You should then check for new valid data on the next frame.

4.1 Modify your I^{2}S controller testbench

Copy over the I^{2}S controller testbench you used to verify the clock waveforms in Lab 6.

Modify the I^{2}S controller testbench by sending data to it using the `system_clock` in the `initial` block. Then execute the testbench again, and verify that your I^{2}S controller is able to properly interface with the piano FSM. You may have to modify the default parameter values.

5 Building the Piano FSM

Now we will design the logic that interfaces the FIFOs coming from the UART and the async FIFO that provides data for the I^{2}S controller. This module is the "Piano FSM" in the block diagram in the lab intro.

The skeleton for the `piano_fsm` is provided in `piano_fsm.v`. You can see that it has access to the UART transmitter FIFO, the UART receiver FIFO, and the I^{2}S sample async FIFO. It also has access to a reset signal coming from the board's RESET and the other momentary buttons. This FSM 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 given amount of time (`note_length`), the tone should be played by sending samples of the tone (at 44.1, 48, or 88.2 kHz) into the I^2S controller
- The `note_length` should default to 1/5th of a second, and can be changed by a fixed amount with the the board's push buttons. This should be similar to the tempo changing 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.
- If the UART receiver FIFO is empty, the I^2S controller can be passed a constant value (like 0) for every sample or not not passed anything at all. The `audio_pwm` output shouldn't oscillate if there's nothing to play.
- The `audio_pwm` output of this FSM connects directly to the mono audio out port on the Pynq-Z1 board. It should be driven with the square wave that's playing through the I^2S controller.

You don't need to design the `piano_fsm` 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
```

You will then have to modify `piano_scale_rom.v` to use the module name `piano_scale_rom`.

A possible implementation of this module would be to include an instance of your `tone_generator`, and sample its output at 48 kHz to send samples into the I^2S sample FIFO. The details of the implementation are all up to you.

It is possible that the UART receiver FIFO can fill up with samples so fast that the piano FSM 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 and audio output.

5.1 Modify z1top

Now open up `z1top.v` and modify it at the bottom to include the new modules you wrote. Wire up the FIFOs and your piano FSM 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. For the interface between the Piano FSM and the UART Transmitter, you should use the Xilinx First-Word Fall-Through FIFO you instantiated earlier. Consult the Updated Xilinx FIFO IP Manual and refer to the the "AXI Interface FIFOs" diagram and the "First-Word Fall-Through (FWFT)" section for information on how to interface your FIFO with a ready/valid interface. Note that AXI Stream is a type of ready/valid 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 is a good default depth.

6 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 `system_testbench.v` (it's under `lab6.srcs/sim_1`). It will be up to you to 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 your `note_length` default value in the `piano_fsm` to something much smaller than 1/5th of a second, just for testing.

7 FPGA Testing

Generate a bitstream and program the FPGA as usual. Read the synthesis and implementation reports 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`. You can reset your design by pressing your `RESET` button. Use the slide `SWITCHES` to turn on your design. The last switch will turn on the `AUDIO_PWM` output, and the I^2S output is on by default.

You also should be able to use the remaining buttons 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.

8 (Optional)

- Implement different output waveforms rather than just a square wave. You can choose to implement a triangle or sawtooth or sine wave output and have the different waveform output modes toggled with the remaining button. Only 1 extra waveform is needed for extra credit.
- Use the hex keypad as an alternative controller for the piano FSM.

- Implement a partial ADSR (attack, delay, sustain, release) envelope of your output waveforms. Here is a reference link `https://en.wikipedia.org/wiki/Synthesizer#Attack_Decay_Sustain_Release_.28ADSR.29_envelope`. You only need to implement an attack and release volume rolloff, and can use a single volume level for both the delay and sustain portions of your waveform. You should include some way of changing the attack and release periods (either through the rotary encoder or buttons).
- Any other ideas you have to make this FPGA design more like a "real" keyboard.

9 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. Demonstrate the piano working on the FPGA both through the mono audio out and the I^2S controller's headphone output.
4. Prove the existence of your UART RX and TX FIFOs by increasing the `note_length` and filling the RX FIFO and seeing the data drain out slowly into your FSM.

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