

EECS 151/251A FPGA Lab
Lab 7: FIFOs, UART Piano, Multi-bit Clock-Crossing

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

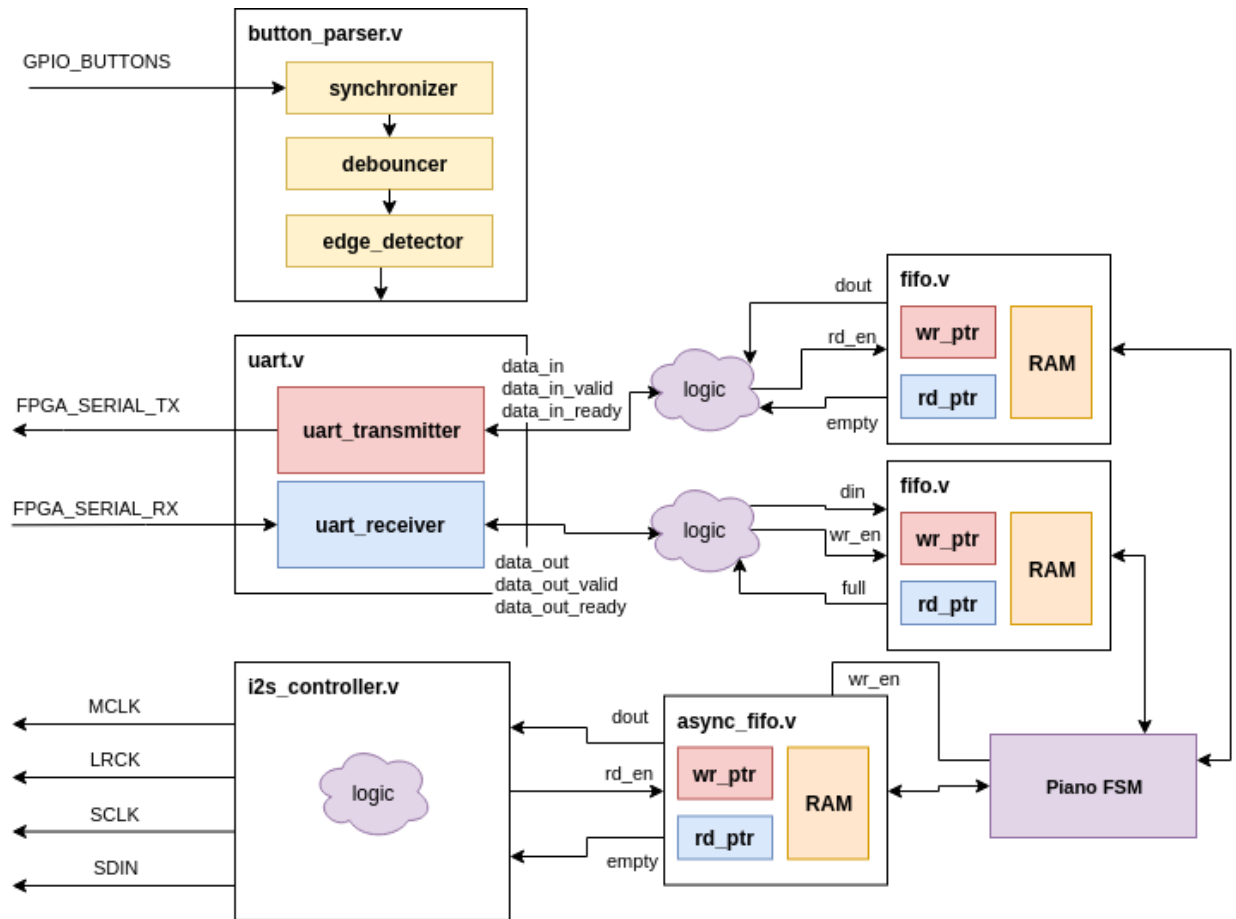
Contents

1	Introduction	2
1.1	Copying Files From Previous Labs	2
2	Building a Synchronous FIFO	3
2.1	FIFO Functionality	3
2.2	FIFO Interface	4
2.3	FIFO Timing	5
2.4	FIFO Testing	5
3	Finishing the I²S Controller	6
3.1	Modify your I ² S controller testbench	7
4	Building the Piano FSM	7
4.1	Modify z1top	8
5	Writing a System-Level Testbench	8
6	FPGA Testing	8
7	Asynchronous FIFOs, Multi-bit Clock Crossing	9
7.1	Async FIFO Construction	9
7.2	Async FIFO Reset Behavior	11
7.3	Async FIFO Interface	11
7.4	Async FIFO Timing	11
7.5	Async FIFO Testing	11
8	(Optional) Extra Credit	12
9	Checkoff Tasks	12

1 Introduction

In this lab, you will integrate the components you created in Labs 5 and 6 (UART and I²S controller). You will begin by building a synchronous FIFO and verifying its functionality using a block-level testbench. You will then modify your I²S controller to use the FIFO as its PCM data source. Then you will create some logic that integrates all these components to form a “piano”. Finally, you will learn about and build an asynchronous FIFO as an exercise.

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 I²S 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.

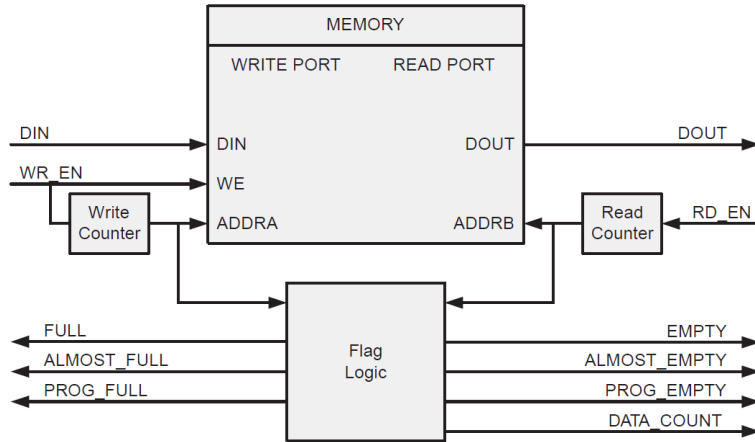


Figure 5-4: Functional Implementation of a Common Clock FIFO using Block RAM or Distributed RAM

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 write 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.

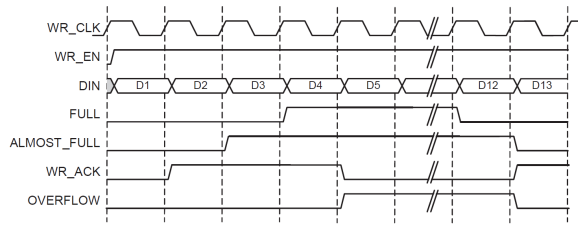


Figure 5-6: Write Operation for a FIFO with Independent Clocks

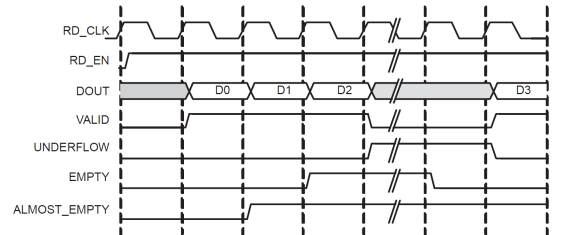


Figure 5-7: Standard Read Operation for a FIFO with Independent Clocks

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 reenale 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 Finishing the I²S Controller

Now that you can generate the various clocking signals for the I²S 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 I²S 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 I²S 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 FIFO's `dout[19:0]` output, an input to take the FIFO's `empty`, and an output ready signal to drive the FIFO's `rd_en`. We will use the data stored in this FIFO as the PCM data we want to send for each audio frame.

Modify your I²S controller to pull one piece of data from the FIFO for each frame. Remember that in the I²S 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.)

If the FIFO is empty when you attempt to pull data from it, you should continue sending the last value that you had pulled from it on every frame, until the FIFO is no longer empty on the subsequent pull attempt.

3.1 Modify your I²S controller testbench

Copy over the I²S controller testbench you used to verify the clock waveforms in Lab 6.

Modify the I²S controller testbench by instantiating a FIFO and sending data to it using the `system_clock` in the `initial` block. Then execute the testbench again, and verify that your I²S controller is able to properly interface with the FIFO. You may have to modify the default parameter values when you instantiate the `fifo`.

4 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 I²S 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 I²S 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 I²S sample FIFO
- 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 I²S FIFO can be filled with a constant value (like 0) for every sample or not filled 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 I²S 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 lab7
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 I²S 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.

4.1 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.

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.

5 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 `lab7.sracs/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.

6 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 `I2S` 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.

7 Asynchronous FIFOs, Multi-bit Clock Crossing

In a previous lab, we built a single-bit synchronizer, which brought an asynchronous signal from off the FPGA (i.e. buttons) into the system clock domain. The synchronizer consisted of two D flip-flops connected in series, clocked by the system clock. This synchronizer however only works for a single bit. To synchronize an entire bus across clock domains requires a more complex synchronization scheme.

One solution, among others, is an asynchronous FIFO which works like a synchronous FIFO, except for the fact that the read and write interfaces are clocked by different clocks (with no known phase or frequency relation).

Some peripheral devices, like older AC'97 audio controllers, operate in their own clock domains. To communicate with them requires bridging data between their domains and that of our FPGA. For example, an AC'97 controller might run at 12.288 MHz while our system clock runs at 125 MHz. The `I2S` controller we use does not require this complex a synchronization scheme, but it's useful to learn to build one anyway.

7.1 Async FIFO Construction

An asynchronous FIFO is constructed similarly to a synchronous FIFO with a two ported RAM, a read and write pointer, and some logic to generate the full and empty signals. One difference is that the two ported RAM has two independently clocked ports. Another difference is that the read and write pointers need to be properly transferred to the other clock domain before going through the full and empty signal generation logic. Here is an overview of the internals of an async FIFO from the Xilinx FIFO IP Manual, page 100.

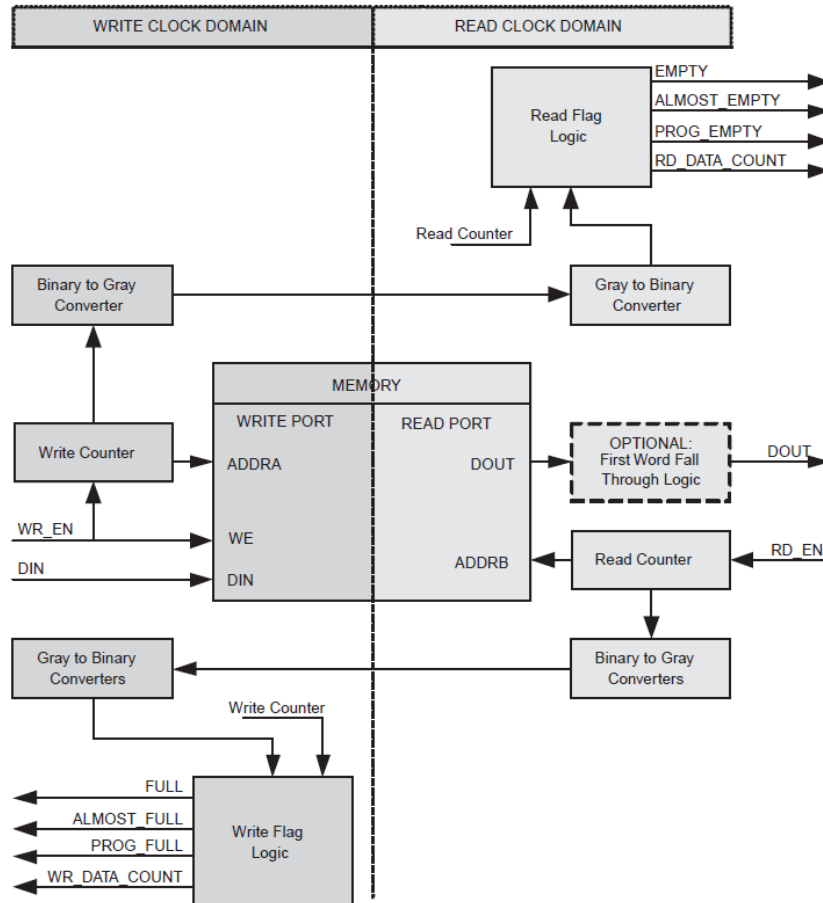


Figure 5-2: Functional Implementation of a FIFO with Independent Clock Domains

Notice that there is a clear divide between the two different clock domains. The two arrows that cross the clock domain are the gray-coded write and read pointers. This clock domain crossing should be performed by using 2 registers in series clocked by the other clock domain, just like the 1-bit synchronizer. This will provide a robust clock crossing to avoid data corruption.

This means that the binary write pointer should be converted to a gray-coded write pointer, before passing through 2 registers in series clocked by the read clock. Then, the gray-coded write pointer has been successfully moved to the read clock domain, where it can be converted back to binary and compared against the read pointer for generating the `empty` signal. The same logic chain applies to the read pointer transfer into the write clock domain.

As you build the async FIFO, you might find it useful to create binary to gray and gray to binary modules that you can instantiate in your design. The gray code Wikipedia article has some pseudocode that can help you write these modules in the 'Converting to and from Gray code' section. You can make the assumption that the async FIFO for this lab and coming project will not have read and write pointers greater than 16 bits, and can design your gray code converters appropriately.

7.2 Async FIFO Reset Behavior

Handling resets in an async FIFO is tricky since a synchronous reset from one clock domain won't necessarily be an appropriate reset for the other clock domain. Here are a couple reset strategies:

- Use one synchronous reset signal with respect to one clock domain, and propagate the reset to the other clock domain internally with a 1-bit synchronizer.
- Use two reset signals, one for each clock domain, and have each reset be synchronous to its respective clock.
- Use one global asynchronous reset signal to reset both clock domains at once.
- Don't use an explicit external reset signal but define initial reg values which can be synthesized for an FPGA.

For this lab, we are going to go with the last option as it is the most robust when you have, for example, an FPGA and a temperamental AC'97 bit clock. This means that you should initialize any `reg` nets that represent actual registers in your async FIFO design.

7.3 Async FIFO Interface

The interface of the async FIFO template in `async_fifo.v` is identical to the interface of the synchronous FIFO with the exception of having an independent read and write clock.

7.4 Async FIFO Timing

The timing of the async FIFO is similar to that of the synchronous FIFO with the exception that the full and empty signals can have 'delayed' transitions (of a few clock cycles) due to the need to synchronize the read and write pointers. Refer to page 110 of the Xilinx FIFO IP Manual for the timing diagrams.

7.5 Async FIFO Testing

The same testbench used for testing the synchronous FIFO can be used to test the async FIFO at `fifo_testbench.v`. The only change is that at the top of the testbench file, uncomment the `ASYNC_FIFO_TEST` define and comment out the `SYNC_FIFO_TEST` define. Then run the testbench as usual within Vivado or with `make sim`.

The same additional tests recommended with the synchronous FIFO are recommended for this FIFO as well. These additional tests are even more important to get right with the asynchronous FIFO, so we highly recommend writing them.

By now, you should have a working async FIFO, at least in simulation. As you may discover, basic functional simulation isn't usually a good way of telling whether a clock crossing works as expected, although it will confirm that your logic is sound.

8 (Optional) Extra Credit

You can complete any of the tasks in this section to get a boost of your lab grade. Everything here is optional. If you would like to try something else for extra credit, ask a TA.

- 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 I²S 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.
5. Show that your asynchronous FIFO works in simulation and explain the differences in implementation and behaviour (for example through simulation) to the synchronous FIFO.