RS-232 and Music Players using a DE2-115 FPGA

Reuben Beeler
Department of Physics, University of California Santa Barbara, Santa Barbara CA 93106
(Dated: February 14, 2025)

This experiment explores RS-232 serial communication using a DE2-115 FPGA, demonstrating both a loopback interface and a practical music player application. By implementing a universal asynchronous receiver-transmitter (UART), we analyzed data transmission integrity, signal timing, and decoding accuracy under varying parameters. The results highlight the robustness of RS-232, its susceptibility to misconfiguration (e.g., parity errors), and its application in streaming audio data. Insights from oscilloscope traces and FPGA display outputs confirm the expected behavior of the protocol, underscoring its continued relevance in embedded systems.

INTRODUCTION and METHODS

Serial communication is a fundamental method for data exchange between digital systems, with the RS-232 protocol serving as a widely used standard. This experiment focuses on implementing and analyzing RS-232 communication using an FPGA-based loopback interface designed in Quartus. By sending and receiving data through a universal asynchronous receiver-transmitter (UART), we examine how the protocol encodes and transmits information.

This experiment first focuses on analyzing RS-232 communication between an FPGA and a desktop computer, and then demonstrates a real-world example by sending an audio file via RS-232 to the FPGA which plays it on a speaker.

The experiment consists of two main components: (1) testing a loopback connection using a terminal emulator, FGPA displays, and an oscilloscope, and (2) sending an audio file via RS-232 to the FPGA to be played on a speaker. The first task involves learning about RS-232 and interacting with a pre-designed RS-232 decoder, while the second explores processing arbitrarily large data sent via RS-232. By altering transmission parameters and observing the effects, we gain insight into the robustness and limitations of RS-232 communication.

Understanding these concepts is essential for developing reliable embedded systems and interfaces, particularly in applications requiring real-time data exchange. This report details the experimental setup, observations, and key insights derived from analyzing RS-232 communication on an FPGA platform.

Questions

There are a few questions to answer here.

1. Look at the two small LEDs next to the serial port, RX and TX. Hold down a key. Which of the two lights up? One or both? Flip the loopback switch and try again.

The FPGA has two lights serial port communication, specifically RX and TX for receive and transmit, respectively. After connecting a terminal emulator with the loopback interface switched to transmit, both RX and TX light up dimly when holding down a key because the character being pressed is being repeatedly received and transmitted. When the loopback interface is switched HI so that TX the transmit line is grounded, the RX light behaves the same as before but TX is constantly on since the LED is active LO. This is likely because LO indicates data transfer in RS-232 (and probably other serial communication as well) while HI is the default, non-transmitting state.

2. Try a few key presses and note how they are decoded in binary. Try <Ctrl+A> = 00000001_2, <Ctrl+B> = 00000010_2, <Ctrl+D> = 00000100_2, and U = 01010101_2. Why would the characters not be decoded correctly if you set Parity to "ODD"? Try it.

With ODD parity (sent after the 8 data bits), PuTTY sends the following sequence for each byte of data: 1 start bit (LO), 8 data bits, 1 parity bit, 1 stop bit (HI). However, our FPGA program is not configured to check for parity, so it only reads 1 start bit, 8 data bits, and 1 stop bit, which are followed by a logical level 1 representing the default state (no data transmission). When parity of the data is EVEN, then PuTTY sets the parity bit to 1 to make the total parity ODD, but the FPGA sees this parity bit as the stop bit and terminates, so it works out anyway as long as there are no errors in data transfer. However, when the data has ODD parity, the parity bit is 0 but the FPGA (which isn't expecting a parity bit) considers this as the stop bit (which is usually set HI), but because it is set LO the program does not actually reset the 13-bit counter. Instead the counter continues counting up to 8191 (which is about 9.37 bits past the middle of the stop bit) at which point the counter carries out and resets to 0. It keeps counting from 0 and thus begins recording data from the serial port (which is now in the default HI state). This

means it reads 8 data bits (111111112, or 0xFF) ignoring the start bit since our circuit technically doesn't check for a start bit. It then comes across another HI bit which it interprets as a STOP bit, and then it resets the counter which waits until the next LO input for the next character. This means the FPGA correctly reads the output of data with odd parity too, but then changes it to 0xFF after about 82μs (which is much too short to notice by eye!). In short, our RS-232 decoder circuit works by accident for data with even parity and fails with 0xFF for data with odd parity. For more complicated serial inputs, the value displayed on the FPGA for data with odd parity may be something other than 0xFF (for example, <Ctrl+E> is 0xFd), but that is beyond the scope of the paper. So, with PuTTY set to odd parity, U and <Ctrl+C> are interpreted correctly (assuming no data errors) since they have even parity while <Ctrl+A>, <Ctrl+B>, and <Ctrl+D> incorrectly display as 0xFF since they have odd parity.

3. Set the oscilloscope to trigger on your key presses so it keeps your data packets on the display for easy analysis. What are the waveform characteristics? What defines the start and stop of the signal? How can you tell it is the letter you typed?

PuTTY sends the following sequence for each byte of data: 1 start bit (LO), 8 data bits in reverse order, 1 parity bit, 1 stop bit (HI). Default state (no transmission) before and after the sequence is HI. We can tell that the letter typed is correctly sent by confirming via oscilloscope (demonstrated later) that 8 binary voltage levels after the start bit correspond to the binary representation of the typed letter in the order LSB to MSB.

4. Try changing the baud rate at the terminal and send packets again. Why are they no longer decoded correctly?

The oscilloscope shows the correct signal when sending data at a baud rate of 9600bps, for instance, instead of 115200bps. This is because it is displaying the correct PuTTY signal from the serial port without trying to interpret the data digitally. On the other hand, the FPGA program tries to complete digital conversion which are now occurring at the wrong times. It shows unusual values on the display (most of which are 0xFc) because the counters/clocks wait 434 clock cycles (8.68μs) for each bit when they should wait 104.2μs according to the 9600bps baud rate. If the baud rate was close to the RS-232's intended rate of 115200bps by about 10% or less, it would still yield the correct result due to RS-232's flexibility on timing, however 9600bps is sufficiently different from 115200bps so it yields consistently incorrect results.

ANALYSIS

RS-232 Decoder

The first part of the RS-232 decoder circuit we are working with appears in Fig. 1. The main inputs we are concerned with are the RS-232 receiver line (incoming data from the desktop) and the FPGA's 50 MHz clock. The outputs of concern are (1) the RS-232 input redirected to a GPIO pin for probing via oscilloscope, (2) the transmit line for echoing the serial data received back to the sender (optionally, depending on the loopback switch), and (3) the data lines from the RS-232 decoding that displays the most recently decoded byte on the 8 green LEDs and the two rightmost 7-segment displays. The bus "q[7..0]" is the output of the RS-232 decoder, which is shown in Fig. 2. In simple terms, the RS-232 decoder has a timer and essentially state machine in the top, which determines whether the decoder is reading or not.

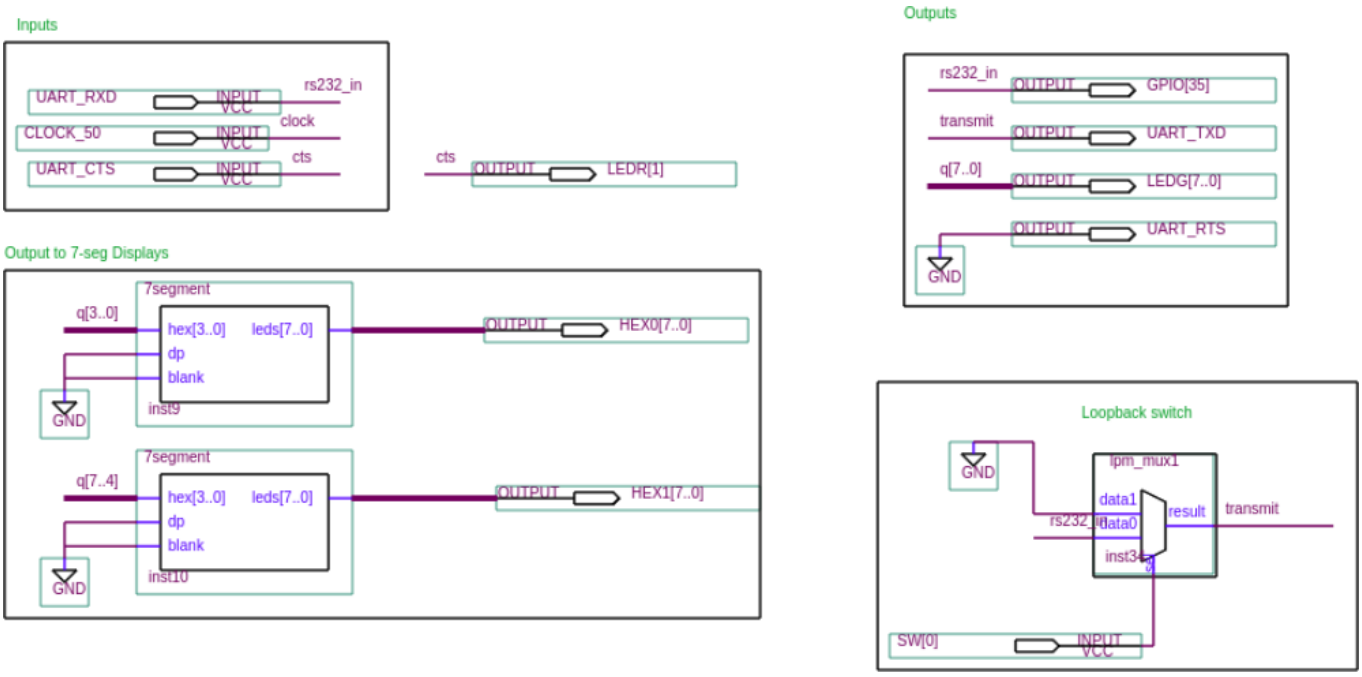

FIG. 1: The inputs and outputs of the RS-232 circuit.

If it is reading, the 13-bit counter counts time that is sent to each of the "compare" submodules below. With a baud rate of 115200bps and a clock of 50 MHz, each bit of serial input lasts for 434 cycles. The start bit begins the counter and can be ignored. The first data bit (LSB) starts at the 435th cycle. To account for the most flexibility, our comparators aim for the middle of each bit; for the LSB data bit, this is 434/2 cycles after 434, so it reads the value when the counter hits 651. This one-shot pulse sets the DFF conrolling q's LSB ("q[0]") to be the value of the serial line (first bit) at cycle 651. One bit later (434 clock cycles later), the second comparator sets the next DFF controlling q[1], and it continues like so until the 9th comparator, which expects a stop bit. Then if the stop bit is set to HI, which it should be, "reset" is a one-shot pulse that changes the decoder to its waiting state by clearing the counter so that the decoder may begin the next batch.

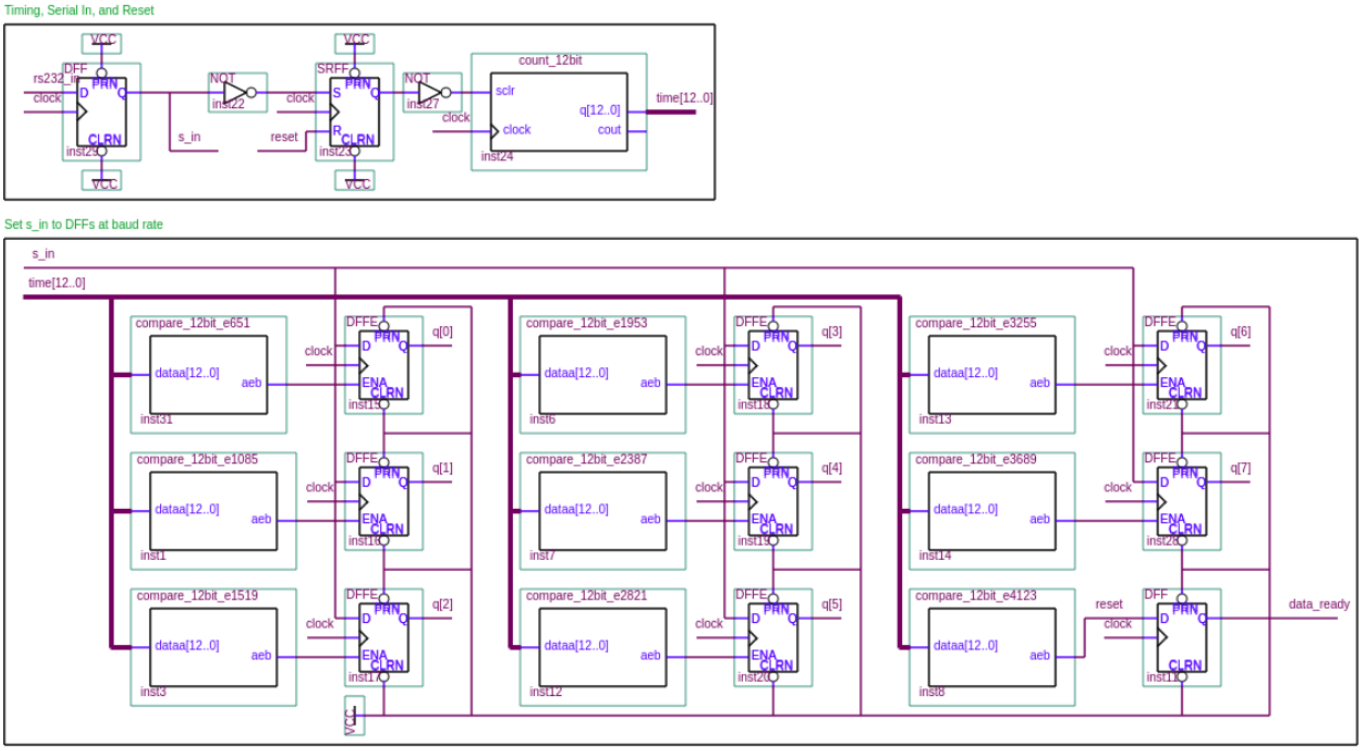

FIG. 2: The RS-232 decoder implementation

Once this circuit is programmed onto the FPGA, it can be tested from the desktop with a serial connection between the FGPA's UART port and the desktop's USB port. We can control the input to the FPGA using PuTTY, a terminal emulator, which allows us to send keystrokes as single characters to the FGPA. When the loopback switch is LO, the RS-232 input is redirected to output back to the PuTTY terminal exactly as received, so each keystroke appears doubled in the PuTTY terminal window. When the loopback switch is HI, nothing is transmitted back to the desktop. An example with the loopback switch set to HI is shown in Fig. 3 when the input from the computer is the character 'a'; the FPGA displays 0x61, which is the ASCII encoding for character 'a'. Similarly, an input of 'b' puts 0x62 on the FPGA's hexidecimal display.

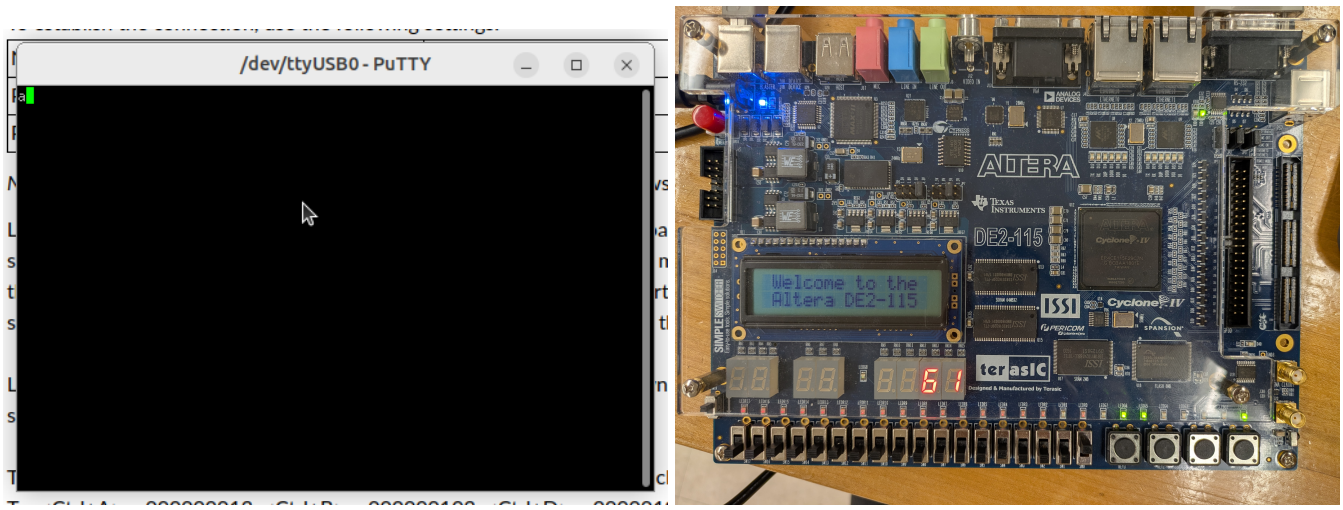

FIG. 3: The PuTTY terminal and FPGA with output hexadecimal output 0x61 after input character 'a'

As explained earlier, including an ODD parity bit in the PuTTY terminal settings creates problems for data with already ODD parity. For instance, if we now send the character 'a' with an odd parity bit, the FPGA reads 0xFF

instead of 0x61 as shown in Fig. 4.

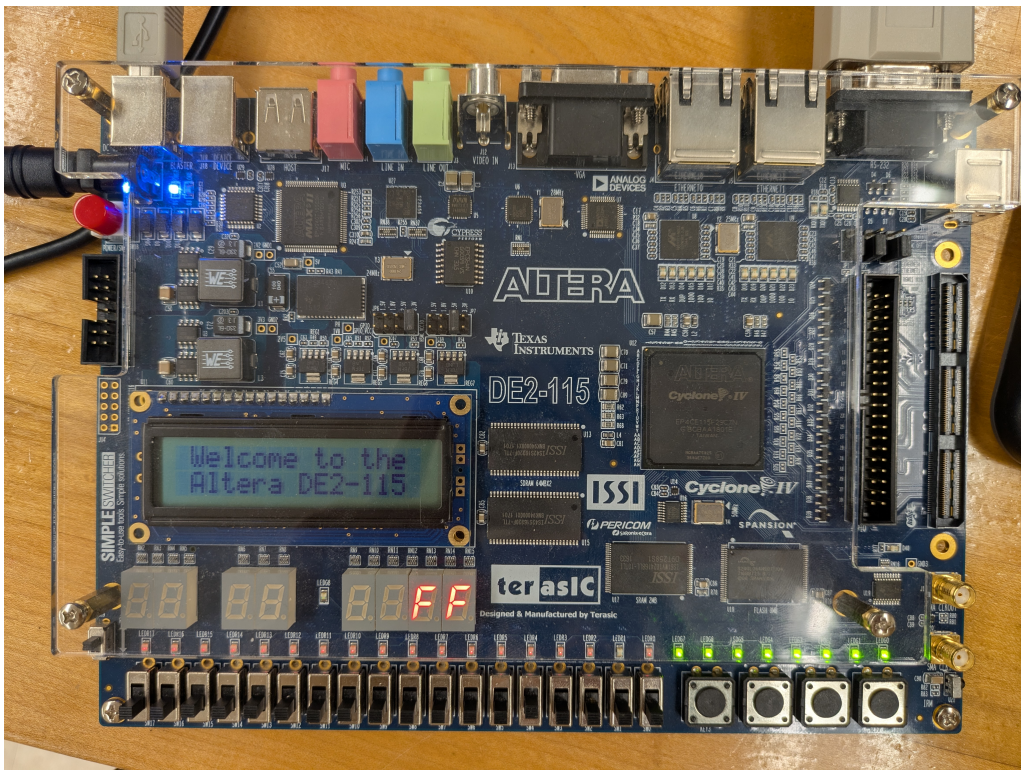

FIG. 4: The FPGA with output hexadecimal output 0xFF after incorrectly decoding input character 'a'

The RS-232 input to the FPGA is redirected to GPIO[35], which is oddly labeled 39, and can be probed by the oscilloscope. Consider the scope trace in Fig. 5 for the input character 'W', which has ASCII value 01010111_2, transmitted with ODD parity. The bits are visible 0111010100 surrounded by 1s. The first 0 is the START bit. the next 8 bits are 11101010, which is the binary ASCII value for 'W' in reverse order. Then, there is the parity bit, which is set to 0 so that the parity of the 9 bits in-between START and STOP is odd. Then, the signal goes HI again, where the bit after the parity bit is technically the STOP bit (HI), but it is difficult to distinguish because the line stays HI afterwards.

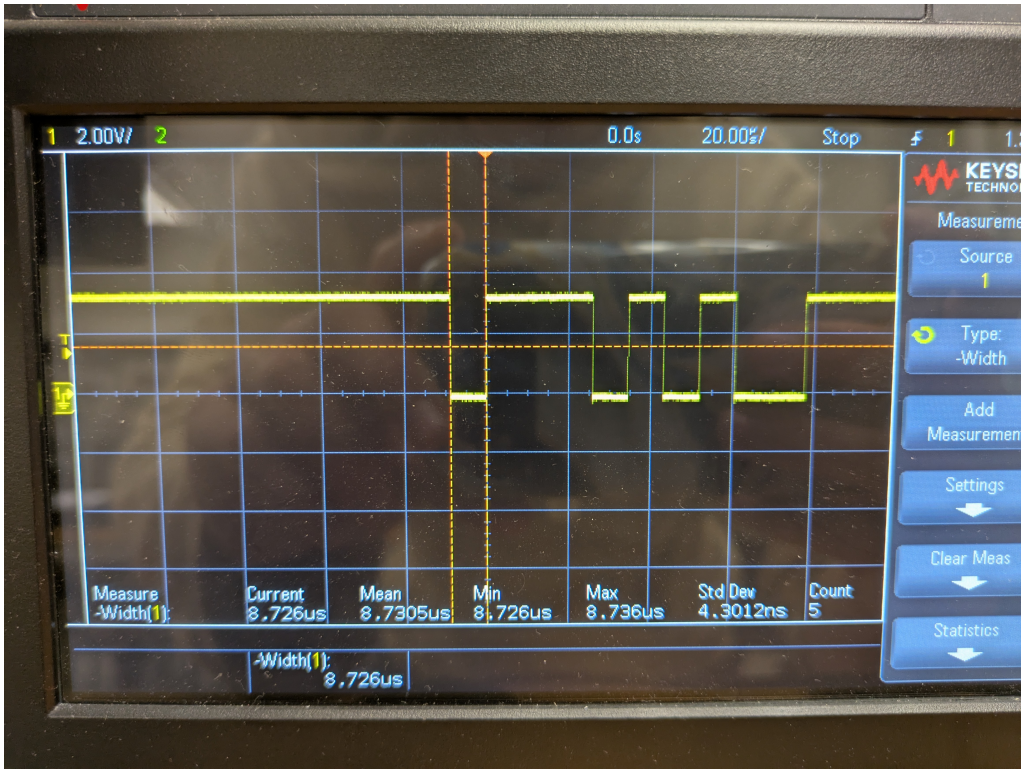

FIG. 5: The oscilloscope trace of 'W' sent with ODD parity at a baud rate of 115200bps.

Referring back to Fig. 5, it can be seen that the width of a single bit is about $(8.7305 \pm 0.0043)\mu$s, according to 5 measurements on the oscilloscope. This corresponds to a baud rate of nearly (114541 ± 56)bps, which is only 0.6% different from the intended baud rate 115200bps. Although the scope only took 5 samples, the uncertainty of the measurement suggests that the actual baud rate is not precisely 115200bps (which is over 11 stdev away), but instead it is near 114541bps which is still similar to 115200bps.

RS-232 Music Player

Using the same RS-232 decoder as above, we implement a music player which accepts files over RS-232 and plays them on a speaker by temporarily storing them in a dual-port RAM. The music player circuit additionally contains the schematic in Fig. 6.

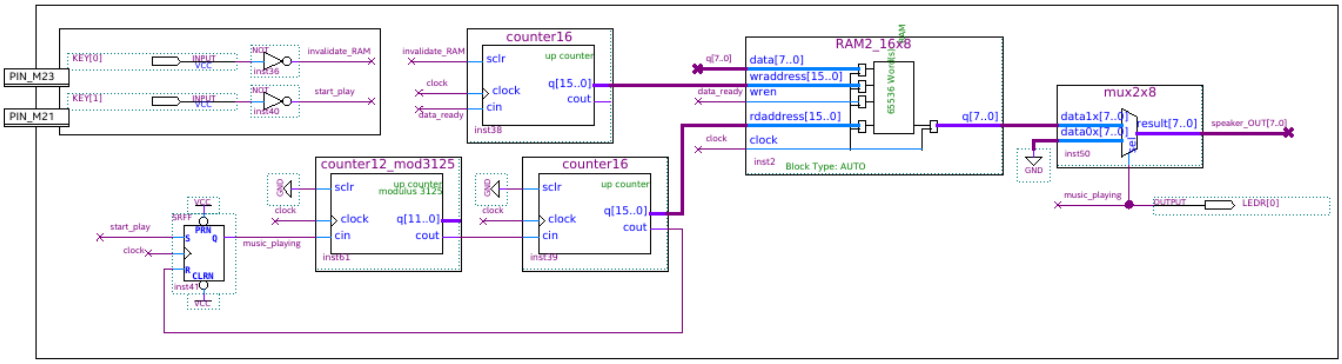

FIG. 6: Music player circuit which stores RS-232 data to RAM.

The first thing to notice is the RAM with 2^{16} one-byte addresses. The RAM is populated by the RS-232 output

"q[7..0]" every time the "data_ready" pulse from Fig. 2 goes HI. This circuit also uses two additional inputs, as shown on the left, which are push-buttons: one to reset the RAM's write-address (for writing another file) and the other to begin playing the audio file. Music is played by reading the RAM sequentially once through, starting from address 0. Note that without resetting the RAM's address, sending multiple files would write them sequentially in RAM, so reading the RAM to play music would result in the same file being played multiple times, which was not our goal, and hence why there is a push-button to reset the RAM's write-address in-between sending audio files from the computer.

When reading, it is important to play the file at the right speed. In this case, we wish to read a file that is designed for 16 kHz, so we put a counter to 3125 to divide the 50 Mhz by 3125 to make a 16 kHz counter that addresses the RAM for reading. The RAM output is sent to the speaker_OUT output, which is grounded when the music is not playing to prevent unwanted noise. Additionally, LEDR[0] indicates when the music is playing, although I unfortunately do not have a video of the music player. To complete the music player, of course we need a speaker, which is connected across VGA_R and GND, because we use VGA_R as our VGA output pin corresponding to the audio level by connecting speaker_OUT to VGA_R as shown in Fig. 7.

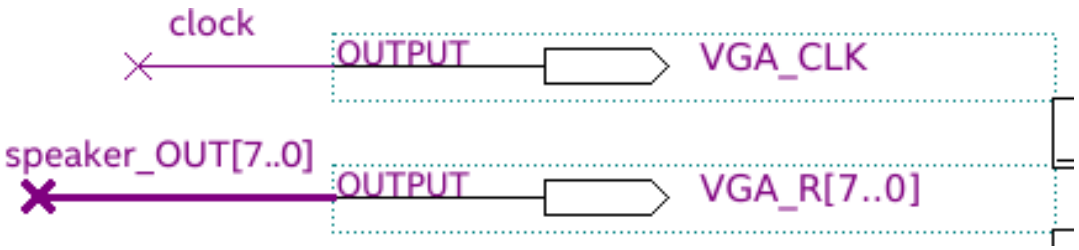

FIG. 7: Additional VGA output pins.

A picture of the full setup is in Fig. 8, which shows the FPGA connected via VGA to the speaker. Additionally, the FPGA displays 0x20 because that is the last byte of the music file transmitted over RS-232. The speaker was very quiet and exhibited low-frequency static, which was unexpected because the audio file was intended to be fairly high-pitch chimes. Out of curiosity and trust in my program, I decided to amplify the speaker with an op-amp as shown to more easily hear the music. The low-frequency static also became much louder, so I included a high-pass filter in hopes of filtering out anything that wasn't the intended chimes. Unfortunately, that did not work well as it resulted in a piercing high-pitch audio that still did not sound quite right, so I removed the filter (and hence it is not shown in the image).

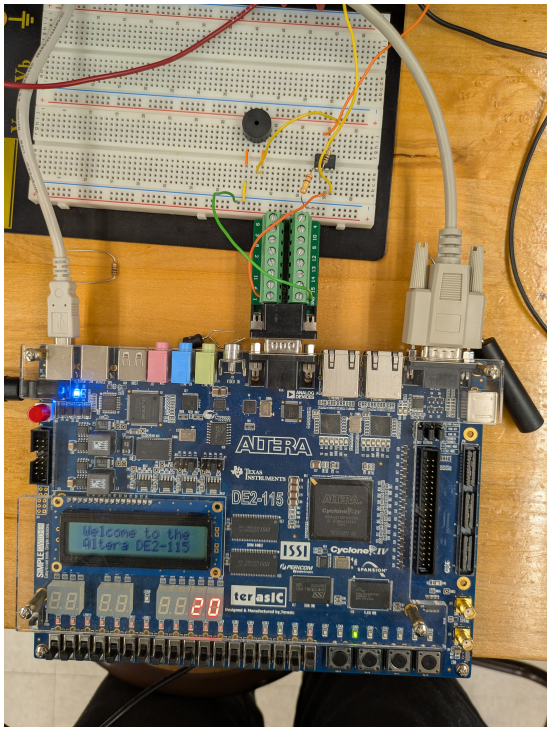

FIG. 8: The FGPA setup with the speaker.

The reason for my troubles turned out to be an error in the decoding process, which I will address soon. First, here is a brief description of the audio file transmission procedure. To transmit the audio file in the first place, I followed two steps. First, I set the serial port parameters (including baud rate, no parity bit, etc.) using the following shell command: `stty -F /dev/ttyUSB0 115200 cs8 -parenb -cstopb crtsct`. Then, I would write the audio file to the program using `cat audiofile.txt > /dev/ttyUSB0`. It turns out that when I had the loopback interface echoing the RS-232 input back to the computer, there were delays in the input, but the RS-232 decoder would keep sending values to the RAM, often times being some constant value. I determined this using the oscilloscope and my own test file that was just a ramp function, and I could see that when the loopback interface was set LO to echo the input, the oscilloscope recorded irregularities in the transmitted data, with frequent gaps between what should have been a perfect ramp. When switching the loopback interface to HI, the files began streaming flawlessly. I believe the error is in our RS-232 decoder circuit, which mistakenly does not check explicitly that the last (stop) bit is actually HI. However, it could also be the timing of the FPGA and delays from simultaneously transmitting and receiving. Alternatively, it could have been a problem with the desktop process writing data to the FPGA in a delayed fashion (failing to adhere to RS-232) due to unexpected data coming from the FPGA. Whatever was the problem, it is beyond the scope of the paper, and a reliable solution is to switch off the loopback interface.

The final test involved playing a "real" song. This time, we played an excerpt of "Barracuda" by Heart using an 8 kHz sampling frequency. We must then modify the modulus of the counter which controls the frequency to be 6250 instead of the previous 3125 (for 16 kHz). Also, this audio file is too small for the RAM from before, which can only store about 8 seconds of music at 8 kHz. So, the final implementation uses the FPGA's one-way SRAM, which has 16 times as many addresses. The circuit replacing Fig. 6 is updated in Fig. 9. The address into the SRAM is either the read or write address (selected using a mux) because there is only one address line. Similarly, there is a single I/O pin for the SRAM data ("SRAM_DQ[7..0]"), which is visibly a dual-port due to it's shape. Then, a tri-state device in conjunction with the SRAM I/O allow writing to the SRAM when "data_ready" and also outputting to the speaker. (Note that when "data_ready" is LO, the tri-state has HI output impedance, and the line is dominated by the SRAM's read data.) Although the provided circuit suggested I play the music when writing the SRAM in addition to reading from the SRAM, I chose to ground the speaker when writing the SRAM so that it would only play music on button press (by reading from the SRAM).

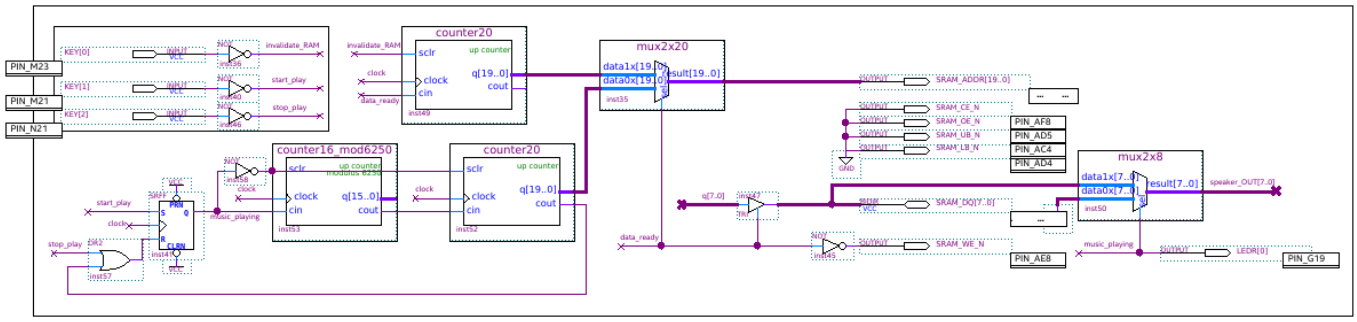

FIG. 9: Music player circuit which stores RS-232 data to the FPGA's SRAM.

Unfortunately, I did not capture a video of the music player, but the speaker was sufficiently quiet and low-quality that it was difficult to discern anyway. That being said, it was still possible to recognize some singing and the different instruments, so the mission was accomplished.

DISCUSSION

The RS-232 loopback experiment confirmed the correct reception and retransmission of data using an FPGA-based UART decoder. The observed LED behavior and oscilloscope traces validated the expected bit sequences, demonstrating the successful interpretation of ASCII characters. However, misconfiguring parity settings led to systematic errors in data interpretation, revealing the importance of matching protocol parameters between sender and receiver.

Further, the oscilloscope data verified the transmitted baud rate's accuracy, showing minor deviations within an acceptable margin. This demonstrated RS-232's tolerance for slight variations in timing. Changing the baud rate at the terminal without adjusting the FPGA decoder resulted in garbled outputs, confirming the necessity of synchronized transmission rates.

In the music player implementation, RS-232 successfully transferred audio data to RAM for sequential playback. The FPGA's ability to read and output audio signals at a controlled frequency confirmed proper timing and buffering. The successful playback underscores RS-232's applicability in continuous data streaming, albeit limited by its relatively low bandwidth compared to modern alternatives.

Overall, this experiment reinforced foundational serial communication concepts, emphasizing the importance of protocol configuration and timing precision in digital systems.