UART LABORATORY ASSIGNMENT

Ammon Dodson

Jake McKenzie

May 25, 2018

Introduction

The most ubiquitous serial port adapters found within microcontrollers today are universal asynchronous receiver transmitters (UART), tracing it's roots back to Gordon Bell who developed it for the PDP series of computers (Kohlenberg & Shkatov, 2013). In this laboratory assignment we will demonstrate this serial communication, both by sending and receiving, between two Altera DE2-115 boards the message "Hello World!\n" via a UART. The data will be sent via 8-N-1, a common shorthand for a serial port setting in which there are eight data bits, no parity bits and one stop bit. The primary objective of this laboratory report is to gain experience with UART and these other common techniques associated with UART. The primary problem associated with UART is to ensure synchronization of an asynchronous signal.

Experimental Design

To send this serial communication we needed to develop a driver to push bits around. This is accomplished by "clocking" the device in specific ways (Scherz & Monk, 2016, p. 844). A counter was developed that sends an enable, once per clock cycle (50 MHz), to a finite state machine that would then begin reading our data containing "Hello World!\n". Many revisions happened to the clock cycle of our finite state machine. We used slides from MIT in the design of this driver (Hom & Steinmyer, 2017), although simplifications were made along the design process to ensure the enable was being set properly and the bits were being read when we wanted them to be.

The first incarnation of the driver, which can be seen in figure 1, had a state where we set all the variables in the initial state then went to a start state. The state only transitioned between *start* to *process data* when the UART is ready for another word, then the states *process data* and *send data* would bounce off each other until the entire file was sent, where *process data* would go to idle until one second had passed. The second incarnation of the driver, which can be seen in figure 2, fixed an issue of the reset.

Prior to this update, the reset would not stop the data transmission until after one full second. The third iteration of our driver involved state reduction. For such a state reduction to occur, we looked for states which have an identical jump to the next state and identical output or equivalent output for the next state. The *start state* ended up being equivalent to the *send data state*. This could be seen by sorting by identical output then seeing whether, for the same set of given inputs, you ask then question: do they jump to a different output? By separating these into distinct groups, then combining redundant states, you can find an optimized state machine, eventually. In the end we needed a fifth state to obtain the correct timing signal in the end. The final iteration of the FSM had seen in figure 4 had us add a wait said to only send the signal once per second.

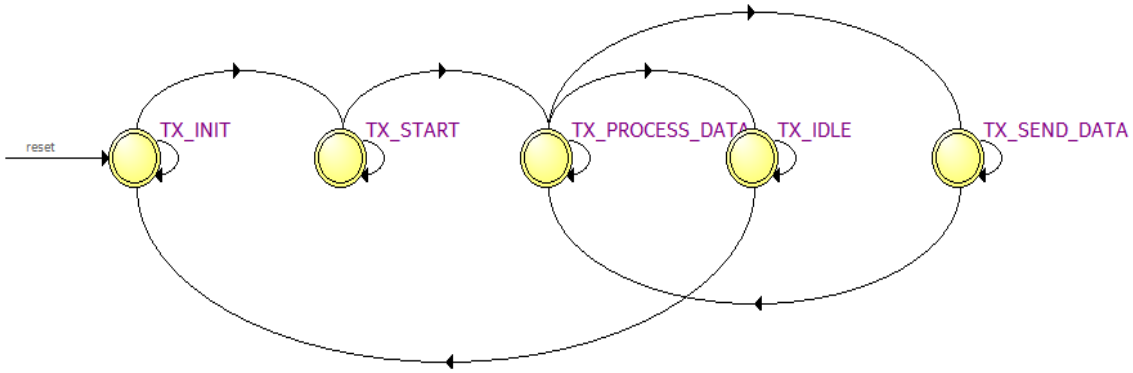

Figure 1: The first FSM design without a loopback

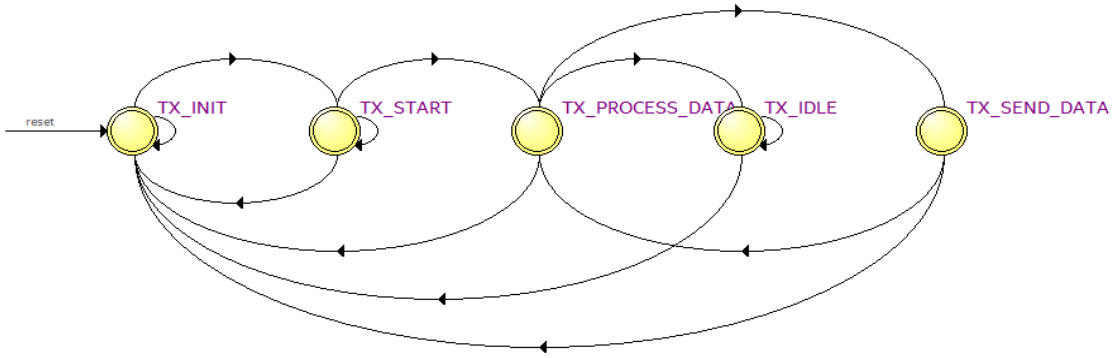

Figure 2: The second FSM design with a loopback added

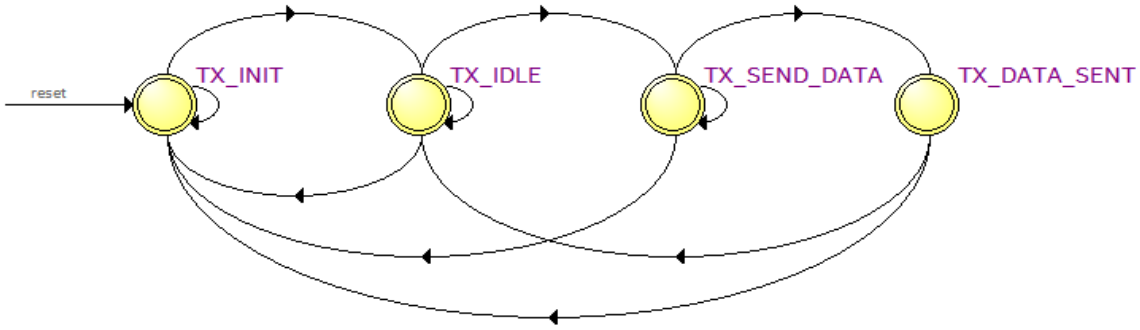

Figure 3: Third FSM design after a state reduction was applied, eventually we realized we needed that wait state.

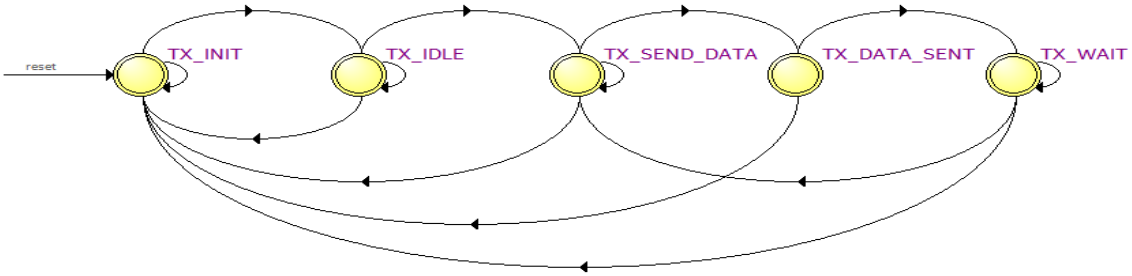

Figure 4: The final FSM after some optimizations were applied which a which reduced the logic elements by 3. (in the final submission).

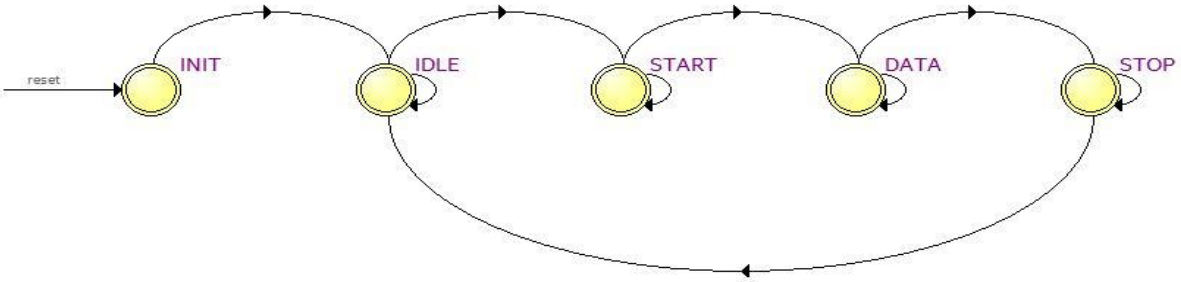

Figure 5: The FSM for the receiver with generalized sampling rate. As we increase the sampling rate the number of elements increases along with the power consumption.

UARTs' are specifically used for guaranteeing correct communications. There are two sides to the communication: the transmitter (TX) and the receiver (RX) (Harris, 2018). The transmitter has parallel data coming in, typically in terms of bytes, as it was in this laboratory assignment. Those eight bits come in, then it sends those bits on serial out. The bits then get received by the receiver on serial in and then it creates parallel data of its own. To send bits of data over a line they must be sent in bits at a time, this process is known as serialization. The bits are then deserialized on the receiver's end and repackaged back into byte chunks for further processing by the system.

Serialization/deserialization is accomplished by creating a buffer both for receiving data and transmitting that data. This is accomplished in serialization in the idle, send data state and data sent state loop. For the deserialization portion, we did not use a finite state machine. Instead the serial data is just fed into a buffer which converts the serial message in to parallel data. When buffers are full data cannot be sent or received. The most crucial step in stable communication is ensuring that the receiver is receiving data at the same rate that the transmitter is transmitting it. Each bit has a duration, which must be agreed upon by the transmitter and receiver to ensure correct communication, which is determined by the baud rate(the maximum number of transitions per second). Another way of thinking about the baud rate is that it is the maximum data transmission rate that can never be achieved (Harris, 2018). T

Synchronization happens when the receiver knows when to accept data from the transmitter. The receiver must know when to expect when each bit will arrive, this is accomplished by setting the start bit low and stop bit high. The byte is refreshed when the stop bit is read to be low, which resets the transmission and forces the receiver to resynchronize. UART was designed to be robust in the face of noise. If the start bit that is read is low for too short of a time, the receiver does not treat that as a real start. The receiver and transmitter must agree on both their samples for a *correct* signal criterion to be established and receivers typically sample at least sixteen times faster than the baud rate (Harris, 2018) (Scherz & Monk, 2016, p. 789).

The UART module was written first by developing a parallel to serial with a shift register. Most parallel data is sent in serial movements and implemented with shift registers (Scherz & Monk, 2016, p. 789). Our original proof of concept worked by not worrying about the start and stop bit. Once we proved that the shift register worked without the start and stop bit we integrated the proof of concept by appending a start and stop bit our byte length message. Implementing the UART module was quite straight forward as it just requires a shift register and we've made shift registers on prior laboratory assignments. The data throughput will never be higher than the baud rate. Since we have 8 bits of data and 2 bits for the start and stop, our data throughput is represented by the equation $\frac{Data\ Bits}{Total\ Bits} Baud\ Rate$ we obtain a throughput of $\frac{8}{10}(38400)\frac{1}{8(1024)} = 3.75\ kilobytes$ per second (Harris, 2018).

Both designs, that of the UART via a shift register and driver produced circuits which primarily used multiplexers instead of D-flipflops as we learned they would be implemented theoretically from the textbook. This is just another example of where the optimizations that Quartus produces in the synthesis process diverge from what we've learned in class. Despite this the circuits are readable, and it is clear from both our code and the synthesized circuit what is going on.

Designing the receiver came late in the project with great aid from Ian Harris's IOT course at Coursera™. Designing a RX is considerably more challenging than designing a TX, both must be designed to be robust in the face of noise and support different baud rates. As Ian mentions in his course, we measure the start bit to see we take a vote, sampling the signal at 16 times the rate that it is sent over the wire. If we tally that the start bit is truly a start bit, majority rules, then we treat that initial signal as a true start bit. Otherwise we cast out the start signal. If the election goes through, then the receiver synchronizes and begins a new election process on the next bit of serial data over the wire.

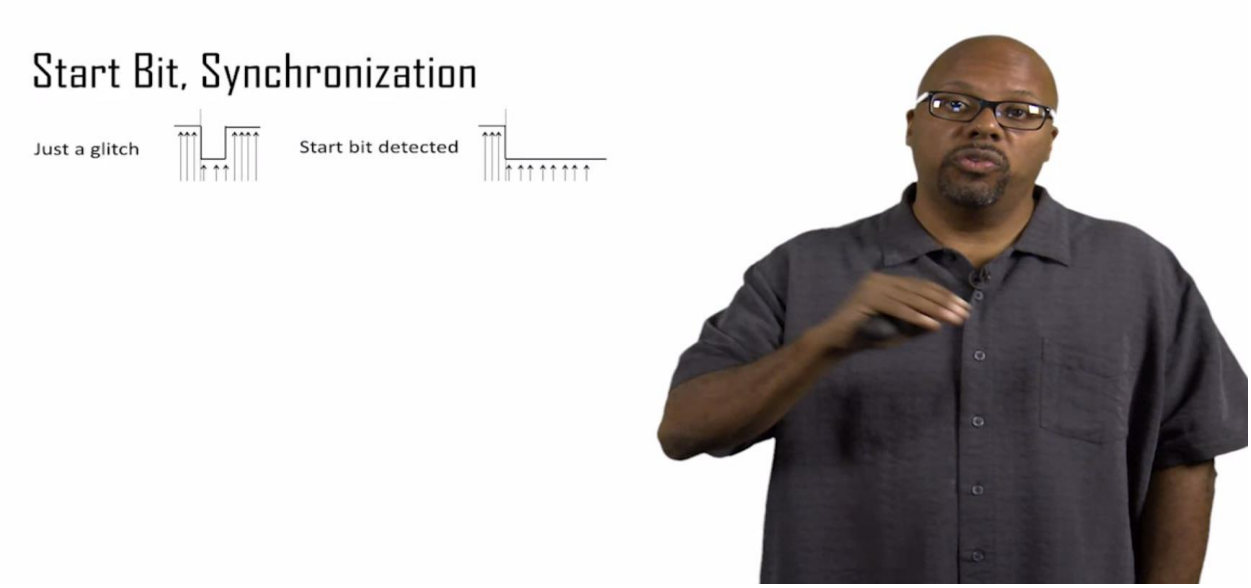

Figure 6: The receiver should ignore a short glitch. If it is measured as low for less than half the sampling rate then it's a glitch that should be ignored (Harris, 2018).

Simulations

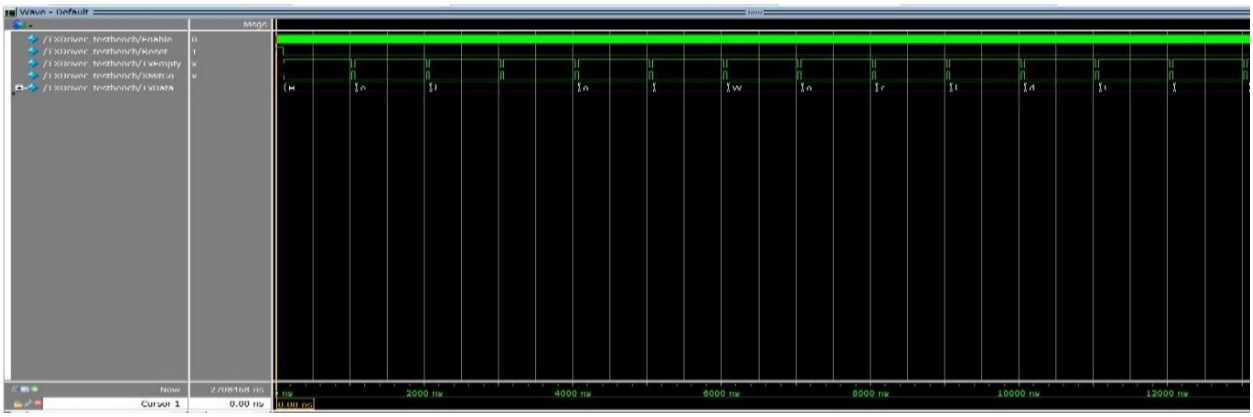

Figure 7: The TXDriver running the message as instructed

In our simulations both the TXDriver and UART_TX performed amply. The primary realization of writing a good design under test is to display values on the terminal, verify results on waveforms and to have asserts for correctness. For the driver an event-based simulation was performed, establishing a clock, then setting reset low then high. Then the signal is let run, producing the "Hello World!\n" transmission repeated. Three console statements were printed at each stage of the character to ensure the state machine was operating correctly. First as XMitGo and TxEmpty are low, then as XMitGo goes high and TxEmpty stays low, then again as XMitGo goes low

then TxEmpty goes high. Each time the data we expect is stored inside TxData. With this, our experimental design matches our simulated results.

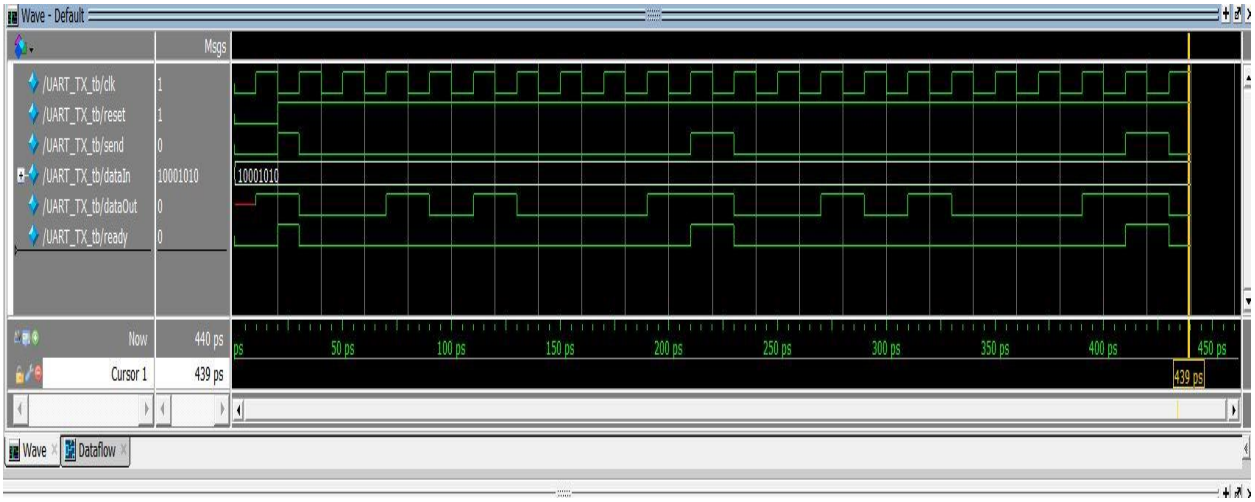

Figure 8: The UART module fed the message 0x8A

As stated previously the receiver must sample at a faster rate, typically 16 times the baud rate. If the transmitter is constantly transmitting, the receiver could not possibly ever receive the message, so a delay was added to ensure that the message could be received. The receiver needs some wiggle room, to allow itself to stop and realize that for instance the start bit was a glitch.

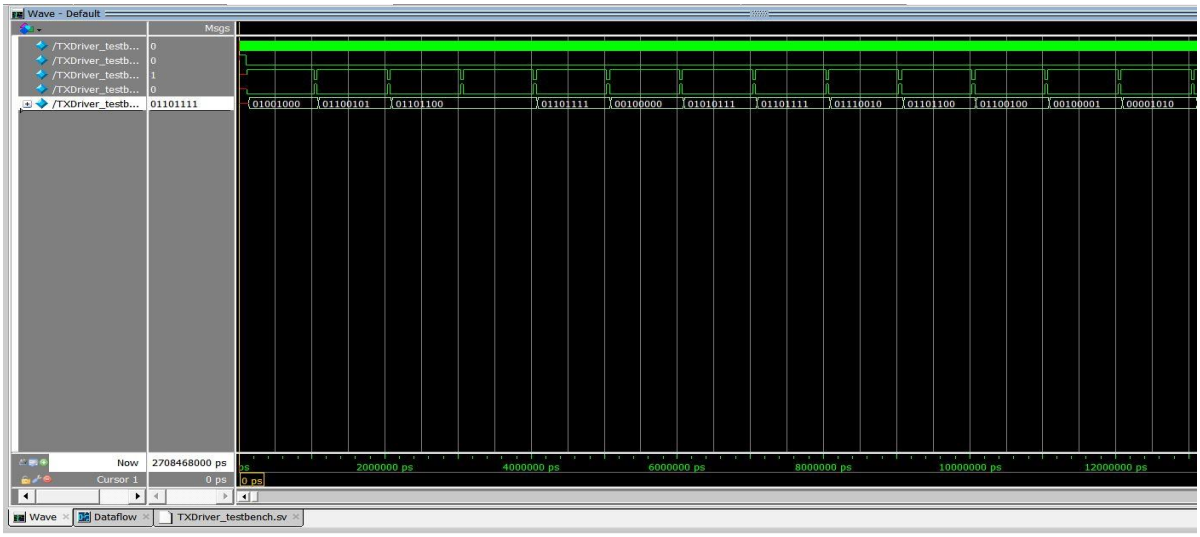

Figure 9: The TXDriver sending the message as instructed (added to make Figure 9 & 10 more readable)

Experimental Results

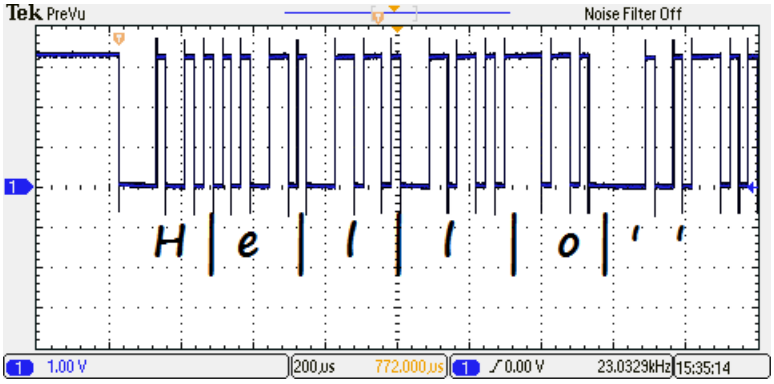

Figure 10:The first half of our message; refer to figure 8 for clarity of the signal bits

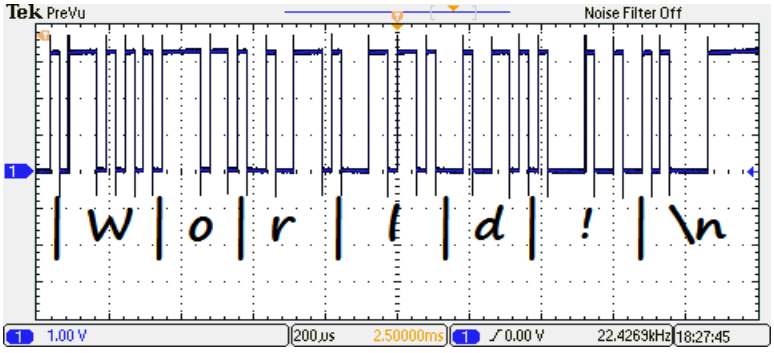

Figure 11:The second half of our message; refer to figure 8 for clarity of the signal bits

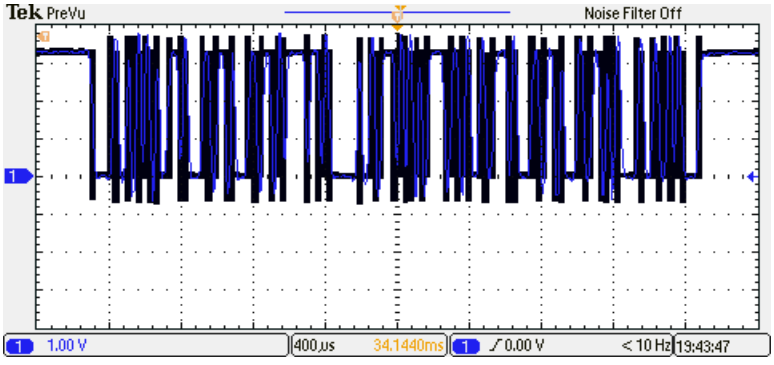

Figure 12: Consideration of pixel density of oscilloscopes is important

When we first ran our experiment, our signal was wrong and unreadable(see fig 12), but after gaining more experience with oscilloscopes we realized a solution. Resolution of signals matter.

If you are zoomed out and you pause a signal the oscilloscope does not have the pixel density to show a clear signal. By finding where our signal outputs by zooming out then zooming back in, then allowing the signal to refresh itself, we were then able to gain a clear signal.

Aberrations are what you want, being just as valuable as correct results. Engineering is made up of mistakes, but they are mistakes which it is useful to make, because they lead piece by piece to good products. Interesting to note that the Gibb's phenomenon is in full effect with our output signal, as can be seen in figure 10 and figure 11. Something that we haven't seen much in practice up to this point in our electronics career.

Discussion

Our experimental results match the simulation, near exactly, aside from the insoluble intrinsicalities common to any design. The time spent on this design process panned out. It is worth noting that the number of design elements that is synthesized fluctuates quite a lot from 81 to 327 with and without the LPM library 1-port ROM module. It appears that the LPM library module for 1-port ROM modules can be decidedly inefficient in the number of elements it adds to the design. With nearly four times as many elements for the same circuit.

Conclusion

This design has great flexibility, high integration, with good reference values. It might appear, at the offset, that parallel communication is more efficient but there is a reason for the wide use of serial communication. For one, serial communication uses less pins for the same data transfer (Harris, 2018). Devices today communicate with many other devices and pins are a finite resource, reducing the number of pins needed for so many devices to talk to each other. Asynchronous communication is not possible in parallel. Due to the UART protocol, data cannot be skewed or exhibit crosstalk, unlike parallel communication. The only downside we see to serial communication is real concern of grappling with complexity but

References

Harris, I. (2018). *The Arduino Platform and C Programming.* Retrieved from Coursera: https://www.coursera.org/learn/arduino-platform/lecture/OPTv7/lecture-2-1-uart-protocol

Hom, G. P., & Steinmyer, J. (2017). *Finite State Machines.* Retrieved from MIT: http://web.mit.edu/6.111/www/f2017/handouts/L06.pdf

Kohlenberg, T., & Shkatov, M. (2013). UART Thou Mad? *Blackhat*, p. 2. Retrieved from An Introduction to the UART Hardware Interface

Scherz, P., & Monk, S. (2016). *Practical Electronics for Inventors.* New York: McGraw-Hill Education TAB.