UART - Communication

Description

The goal of this project is to create a communication between two Nexys 100T boards using the UART protocol. A transmitter sends an eight bit number based on a switch input to a receiver. The receiver then displays the number on the seven segment display.

Getting it to work

Uploading the code

The code is uploaded to the board using Vivado. The project file is attached in this github repository. Once the project is opened in Vivado, the code can be uploaded to the board by clicking the "Generate Bitstream" button. Once the bitstream is generated, the code can be uploaded to the board by clicking the "Program Device" button.

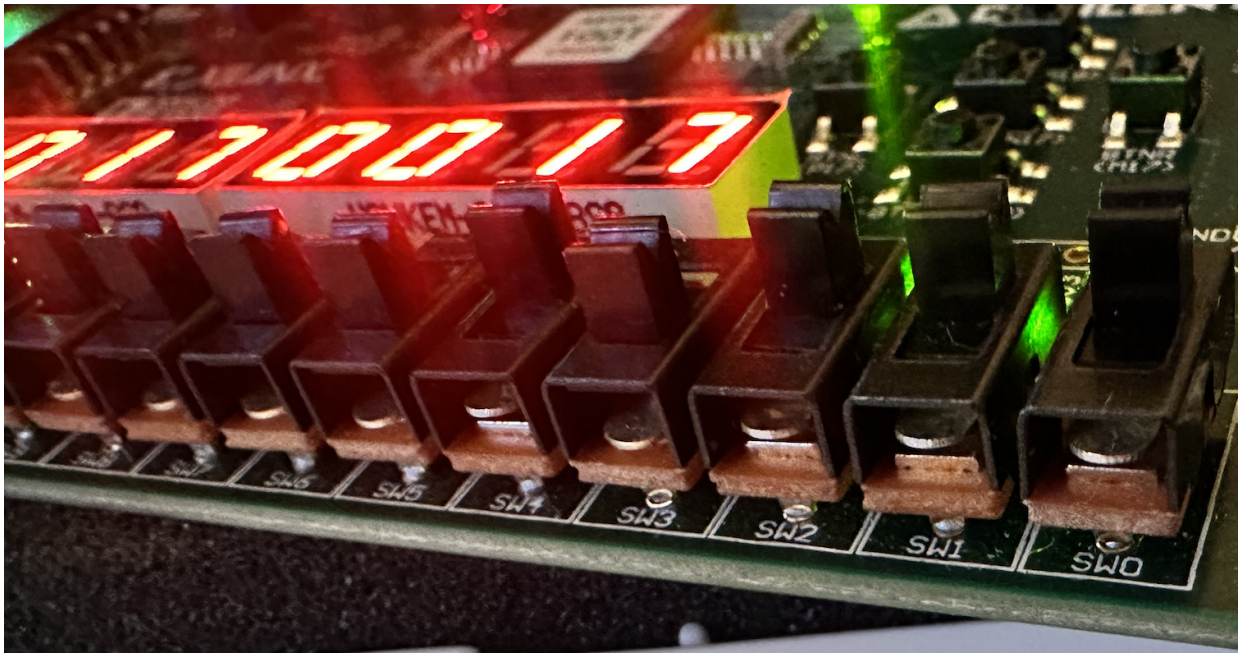
Plugging in the respective pins

The `C17` port on the JA pin block (transmit pin) must be attached to the `D14` port on the JB pin block (receive pin). If you're using two boards then connect the transmit pin on one board to the receive pin on the other board and vice versa.

Selecting the number to transmit

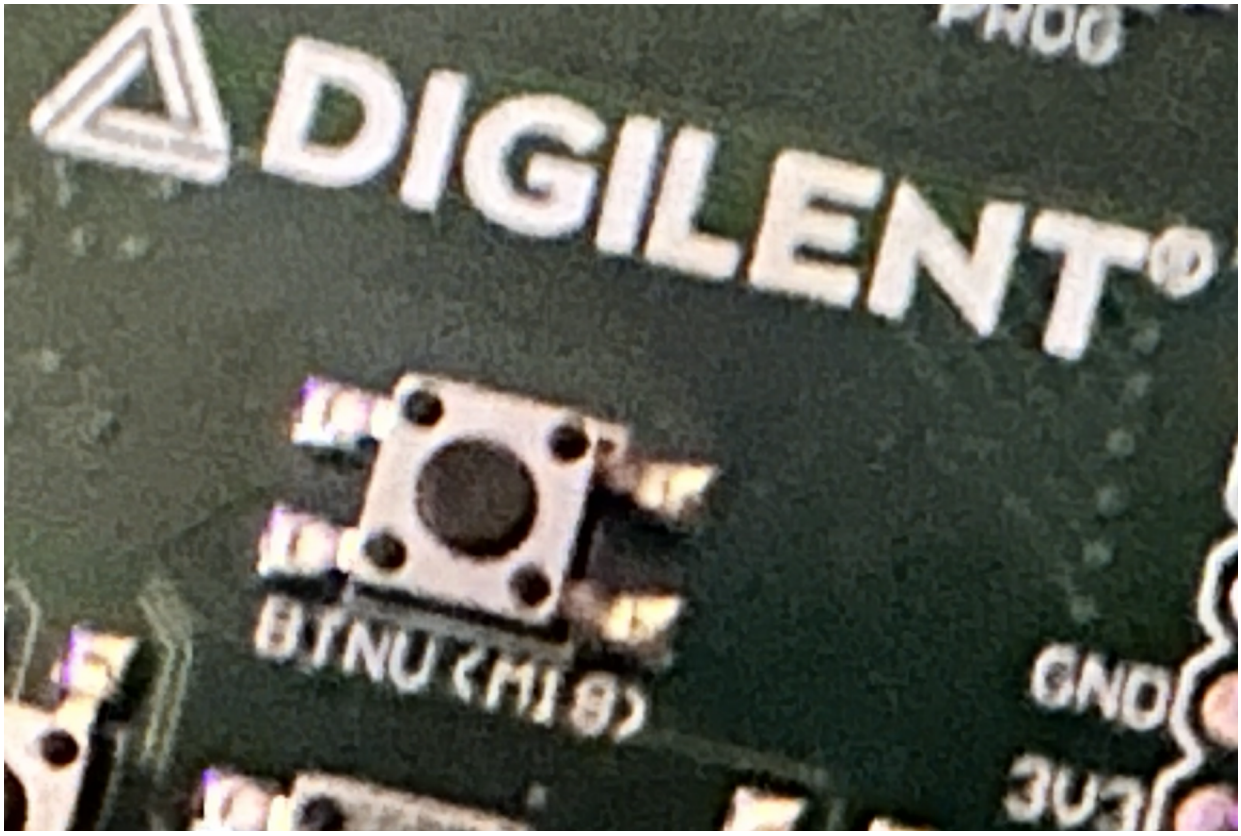
The number that is being transmitted is selected by the switches on the board. On the board they are labeled as switches `J15`, `L16`, `M13`, `R15`, `R17`, `T18`, `U18`, and `R13`. The number is selected by flipping the switches on and off. If the switch is on, then the bit is set to 1. If the

switch is off, then the bit is set to 0. The switches are read from left to right.

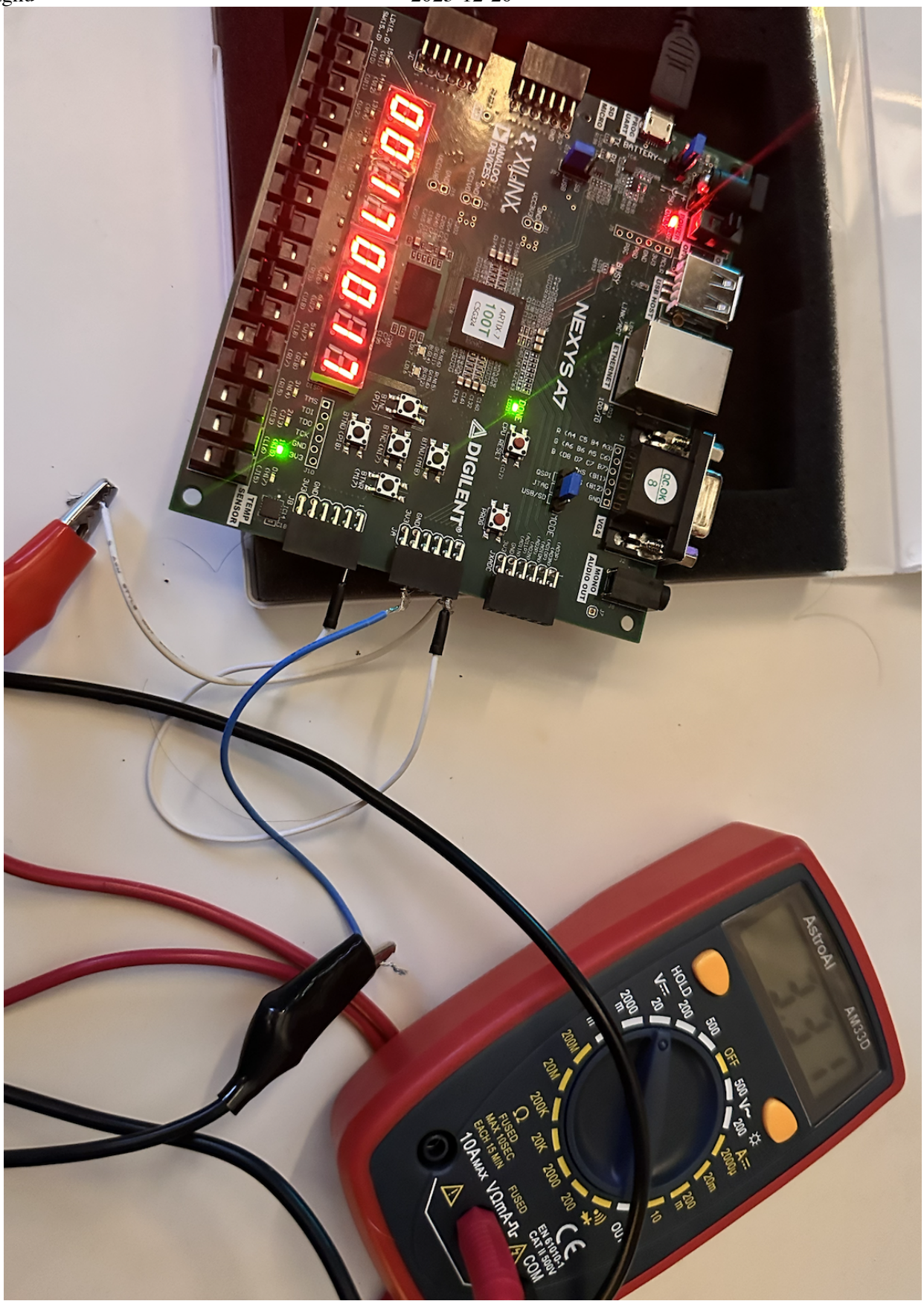


Transmitting the number

The transmission is triggered by the `M18` button on the board. Once the button is pressed, the number will be transmitted.



Results

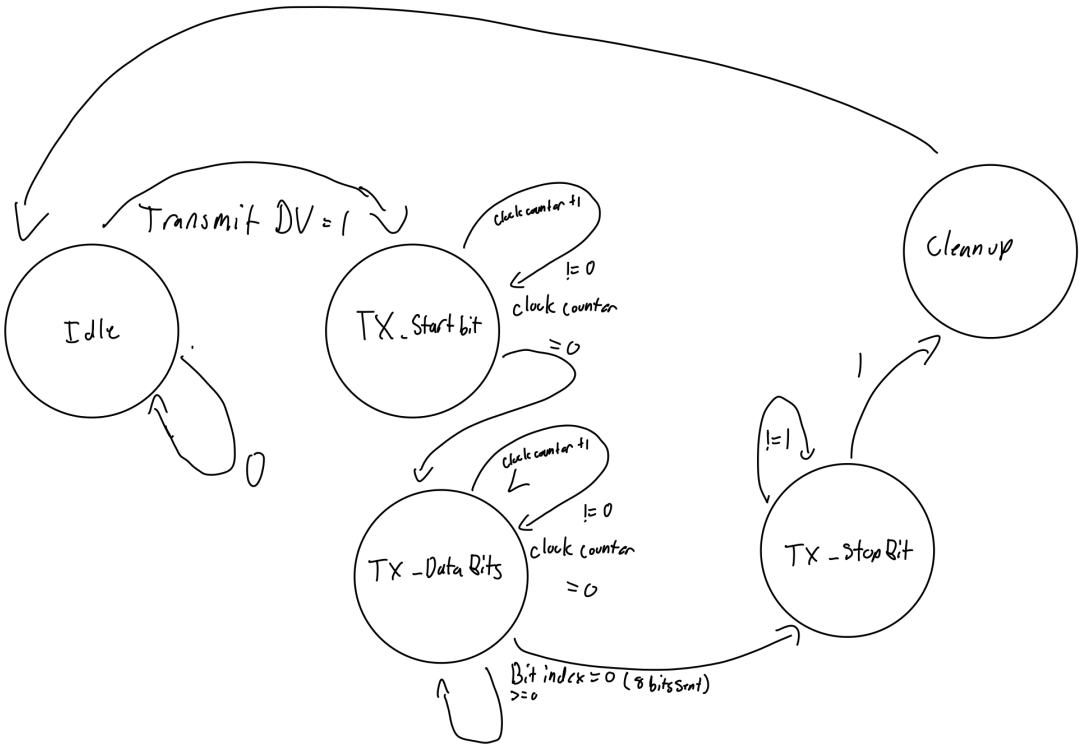


The program was successfully able to transmit the number to the receiver. The left four displays show the number that is being transmitted. The right four displays show the number that is being recieved. In this case we transmitted and recieved the hex number 17. They display the same number because we have them connected in the same board.

How it works?

The whole program relies on the board's built in 100MHz clock. Each bit is sent in intervals. Let's first look at the `UART_Transmitter.vhd` script and analyze it.

Transmitter




```
ClkCyclesPerBit : integer := 100000000
```

In this case, because we're using a 100MHz clock, we can calculate how long each bit will be sent.

$$\frac{100000000 \text{ cycles}}{1 \text{ bit}} \cdot \frac{1\text{s}}{100000000 \text{ cycles}} = 1\text{s per bit}$$

So with the 100000000 cycles per bit, each bit will be sent for 1 second (This would be a Baud Rate of 1)

This is done for demonstration purposes, as normally the Baud Rates seen in actual implementations of UART are much faster.

To change the baud rate to the common rate of 9600, the code in UART_Transmitter.vhd needs to be changed as follows:

```
ClkCyclesPerBit : integer := 10417
```

Important Signals/Ports

TransmitDV

```
TransmitDV    : in  std_logic;                    -- Data Valid signal
for transmission
```

This is attached to a button on the board that will trigger the transmission of the data.

TransmitByte

```
TransmitByte : in  std_logic_vector(7 downto 0); -- Data to be
transmitted
```

This is attached to the switches on the board. This is the number that will be transmitted.

TX_Active

```
TX_Active     : out std_logic;                    -- Transmitter Active
signal
```

This is attached to an LED on the board. This will light up when the transmitter is active.

TX_Serial

```
TX_Serial     : out std_logic;                    -- Serial output
```

This is the current bit that is being sent. It is attached to the output pin on the board.

Stages

The transmission cycle is split up into 5 parts:

```
type StateMachine is (Idle, TX_StartBit, TX_DataBits,
                      TX_StopBit, Cleanup);
signal CurrentState : StateMachine := Idle;
```

Idle

```
TX_Active <= '0';
TX_Serial <= '1';          -- Line High for Idle
TX_Complete <= '0';
ClockCounter <= 0;
BitIndex <= 0;

if TransmitDV = '1' then
TX_Data <= TransmitByte;
CurrentState <= TX_StartBit;
else
CurrentState <= Idle;
end if;
```

In this state, the transmitter is idle. It will wait for the `TransmitDV` signal to be high. Once it is high, it will move to the `TX_StartBit` state.

TX_StartBit

```
TX_Active <= '1';
TX_Serial <= '0';

-- Timing for Start Bit
if ClockCounter < ClkCyclesPerBit-1 then
ClockCounter <= ClockCounter + 1;
CurrentState <= TX_StartBit;
else
ClockCounter <= 0;
CurrentState <= TX_DataBits;
end if;
```

In this state, the transmitter will switch from sending a high signal to a low signal. This is the start bit. It will then move to the `TX_DataBits` state.

TX_DataBits

```
TX_Serial <= TX_Data(BitIndex);

-- Timing for each Data Bit
if ClockCounter < ClkCyclesPerBit-1 then
    ClockCounter <= ClockCounter + 1;
    CurrentState <= TX_DataBits;
else
    ClockCounter <= 0;
    if BitIndex < 7 then
        BitIndex <= BitIndex + 1;
        CurrentState <= TX_DataBits;
    else
        BitIndex <= 0;
        CurrentState <= TX_StopBit;
    end if;
end if;
```

In this state, the transmitter will send the data bits. It will send the bits in order from the least significant bit to the most significant bit. Once it has sent all the bits, it will move to the `TX_StopBit` state.

TX_StopBit

```
TX_Serial <= '1';

-- Timing for Stop Bit
if ClockCounter < ClkCyclesPerBit-1 then
    ClockCounter <= ClockCounter + 1;
    CurrentState <= TX_StopBit;
else
    TX_Complete <= '1';
    ClockCounter <= 0;
    CurrentState <= Cleanup;
end if;
```

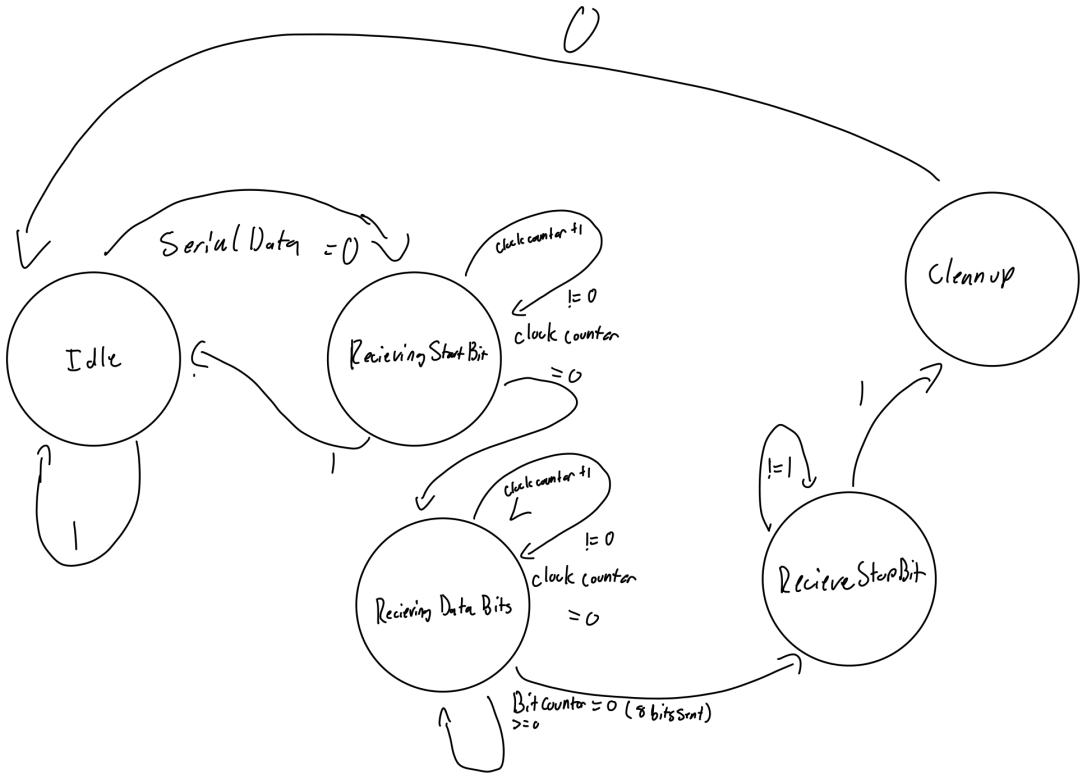
In this state, the transmitter will send a high signal. This is the stop bit. Once it has sent the stop bit, it will move to the `Cleanup` state.

Cleanup

```
TX_Active <= '0';
TX_Complete <= '1';
CurrentState <= Idle;
```

In this state, the transmitter will set the `TX_Active` and `TX_Complete` signals to high. It will then move back to the `Idle` state.

Receiver



The reciever is a little more complicated than the transmitter. It has to be able to read the data that is being sent and display it on the seven segment display.

Like the transmitter, the code was also set to a slow baud rate (this time a rate of .33) for demonstration purposes:

```
ClkCyclesPerBit : integer := 300000000
```

Like previously done for in UART_Transmitter.vhd, to change the baud rate to 9600, the code in Reciever.vhd needs to be changed as follows:

```
ClkCyclesPerBit : integer := 10417
```

Important Signals/Ports

SerialData

```
signal SerialData    : std_logic := '0';
```

This is the current bit that is being recieved. It is attached to the input pin on the board.

ByteAssembly

```
signal ByteAssembly  : std_logic_vector(7 downto 0) := (others =>
'0');
```

This is the current byte that is being assembled. It is initialized to all zeros. As the bits are recieved, the respective bit in the byte will be changed.

DataReady

```
signal DataReady     : std_logic := '0';
```

This indicated whether or not the data is finished being recieved. It is attached to an LED on the board.

Stages

Idle

```
DataReady   <= '0';
ClockCounter <= 0;
BitCounter  <= 0;
if SerialData = '0' then
    CurrentState <= ReceivingStartBit;
else
    CurrentState <= Idle;
end if;
```

In this state, the receiver is idle. It will wait for the `SerialData` signal to be low. Once it is low, it will move to the `ReceivingStartBit` state.

ReceivingStartBit

```
if ClockCounter >= (ClkCyclesPerBit-1)/2 then
    if SerialData = '0' then
        ClockCounter <= 0;
        CurrentState <= ReceivingDataBits;
    else
        CurrentState <= Idle;
    end if;
else
    ClockCounter <= ClockCounter + 1;
    CurrentState <= ReceivingStartBit;
end if;
```

In this state, the receiver will wait for the `SerialData` signal to be high. Once it is high, it will move to the `ReceivingDataBits` state.

ReceivingDataBits

```
if ClockCounter < ClkCyclesPerBit-1 then
    ClockCounter <= ClockCounter + 1;
    CurrentState <= ReceivingDataBits;
else
    ClockCounter         <= 0;
    ByteAssembly(BitCounter) <= SerialData;
    if BitCounter < 7 then
        BitCounter  <= BitCounter + 1;
        CurrentState <= ReceivingDataBits;
    else
        BitCounter  <= 0;
        CurrentState <= ReceivingStopBit;
    end if;
end if;
```

In this state, the receiver will read the data bits. It will read the bits in order from the least significant bit to the most significant bit. Once it has read all the bits, it will move to the `ReceivingStopBit` state.

ReceivingStopBit

```
if ClockCounter < ClkCyclesPerBit-1 then
    ClockCounter <= ClockCounter + 1;
    CurrentState <= ReceivingStopBit;
else
    DataReady    <= '1';
    ClockCounter <= 0;
    CurrentState <= Cleanup;
end if;
```

In this state, the receiver will wait for the `SerialData` signal to be high. Once it is high, it will move to the `Cleanup` state.

Cleanup

```
CurrentState <= Idle;
DataReady    <= '0';
```

In this state, the receiver will set the `DataReady` signal to high. It will then move back to the `Idle` state.

Displaying the Data

The reciever script is also where we implement the seven segment display. From past labs we know how the leddec module works. It just needs to be adjusted to use 8 bits.

```
data : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
```

It's connected to the following ports:

```
L1 : leddec
PORT MAP (
    dig => dig,
    data => ByteAssembly,
    anode => anode,
    seg => seg
);
```

For this to work we had to create a new cnt variable that would increment every clock cycle. Dig (the screen that is being displayed) is set to 19th-17th bits of the cnt variable. This allows the display to cycle.

The data is set to the `ByteAssembly` signal, which if you remember is the byte that is being assembled from the bits that are being recieved.

Conclusion

We were able to successfully implement a UART communication. While we were not able to get together and communicate between two boards, we were able to communicate between the transmitter and receiver on the same board. We were also able to display the number that was being transmitted on the seven segment display which was a feat in itself. Matthew was responsible for implementing the transmission and reception logic. Ryan was responsible for implementing the seven segment display, and Robert was responsible for adding the input buttons/switches and getting the physical voltage to output (over using a direct connection between the transmitter and receiver).

The hardest part was getting the timing right. We had to make sure that the transmitter and receiver were in sync. Otherwise, the receiver would read the data incorrectly.

Overall, we learned a lot about the UART protocol and how to implement it in VHDL.