TECHNICAL MEMORANDUM

Lafayette College

Department of Electrical and Computer Engineering

Title: Asynchronous Serial Receiver

Authors: Zainab Hussein and Kemal Dilsiz

Date: October 4, 2016

Abstract

This lab is an asynchronous serial receiver integrated together with a transmitter from lab 2 and also able to get inputs from other platforms such as RealTerm computer simulation, developed on an FPGA circuit. We made use of stimulus-checking testbenches to verify the functionality of the design.

1. Introduction

This design of receival is very simple and looks for the START and STOP bit in the asynchronous serial transmission. There is an error functionality which helps us to determine if a transmission is erroneous.

The design is based on a much faster BaudRate clock enable and collects the data bits at appropriate positions.

2. The Design

We have thought about using 8 different states for the RECEIVE state showing each individual data bit received. Instead we decided to use a counter for the RECEIVE state to make our code cleaner and brief.

We decided to use two counters: time counter and state counter with two different enables for each to keep track of change in state from receive to stop state, and also the baudrate division of 16 per state. This made it easier to decide when to sample the data and when to check for

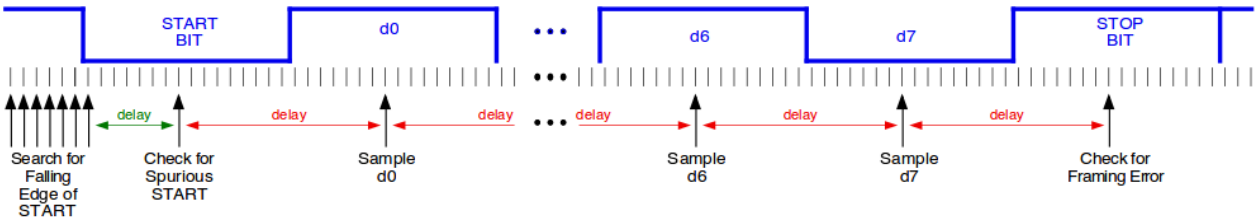
spurious START. We decided on two counters than one counter for clarity when keeping track of

either the baudrate or for RECEIVE-STOP state transition.

We have two counters and we change the state every time the time counter reaches 16, whereas we change the data receival bit with each state counter value. Counters are simple flip flop structures with enables named as **trigger** for bit_count and **increase** for time_count. Code is very simple as follows:

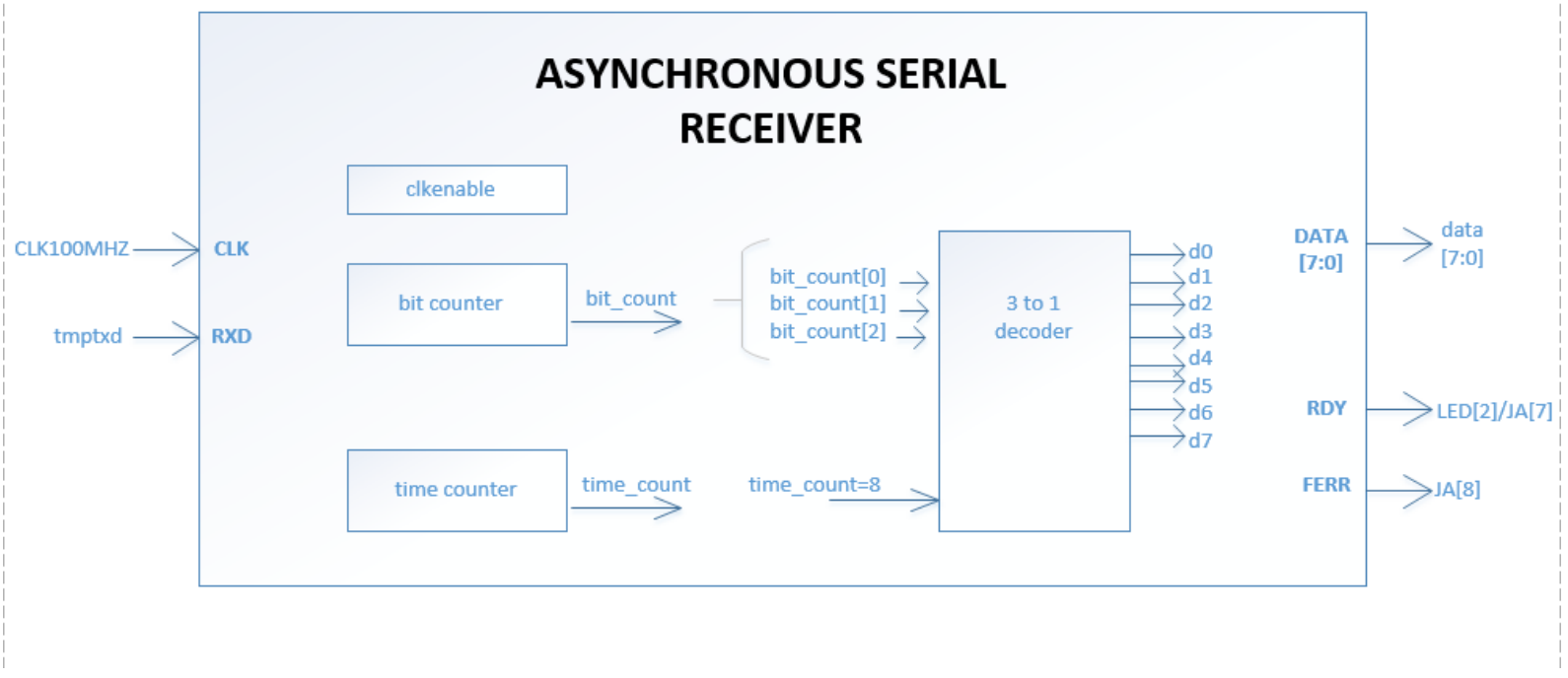
```
logic reset_time_count;
logic reset_bit_count;
logic trigger, increase;
logic [2:0] bit_count;
logic [3:0] time_count;
always_ff@(posedge clk)
	begin
		if((rst || reset_bit_count))
			bit_count <= 3'b000;
		else if(trigger && BaudRate)
			bit_count <= bit_count +1;
	end

always_ff@(posedge clk)
	begin
		if((rst || reset_time_count))
			time_count <= 4'b0000;
		else if(increase && BaudRate)
			time_count <= time_count +1;
	end
```



As you can see on the diagram, each data bit is collected at specific time_count locations and this only happens if the START bit is not erroneous. Furthermore, BaudRate that we have chosen is 16 times faster than the Baudrate for the transmitter. This is a must for the system to work. We simply create a parameter called SIXTEENBAUDRATE for this automatically and the top module has the same BaudRate input for both the transmitter and the receiver.

Here is a diagram of the top module and the receiver module:



We have 4 states in the state machine: IDLE, START, RECEIVE and STOP. All the states rely on the time counter to ensure that each state is a complete baudrate, i.e. 16 baudrates per state.

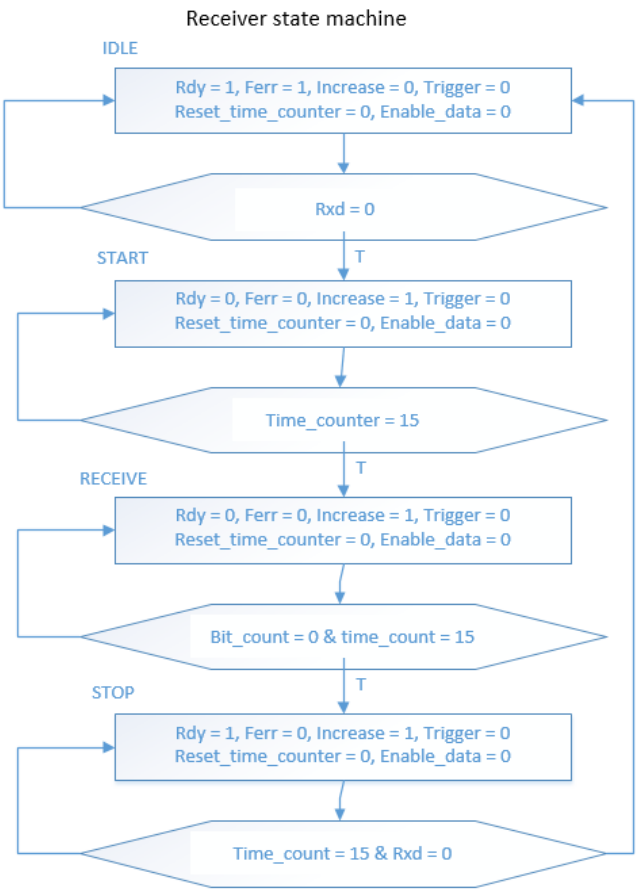
IDLE: This is our default state for when the state machine is reset or if a spurious START error is found in the START state.

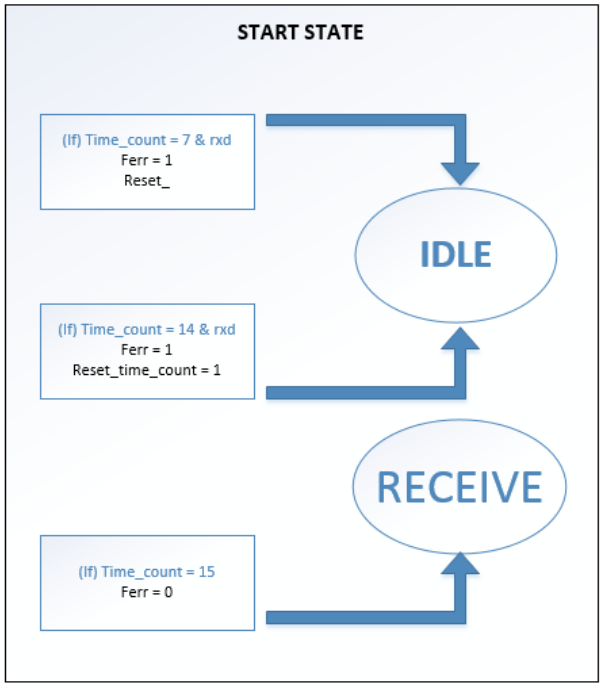
START: This state is for the receival of the data and for rdy to be asserted low to indicate that the receiver already is receiving for that moment and no more data should be transmitted to it at that time.

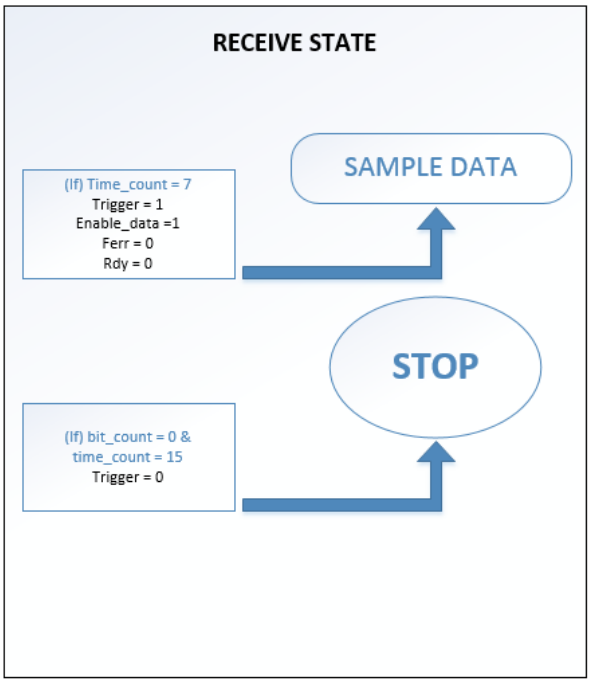
RECEIVE: This state is to facilitate the sampling of the data bits from the transmitter by changing the data receival bit with each bit counter value from the bit counter as shown in the code below:

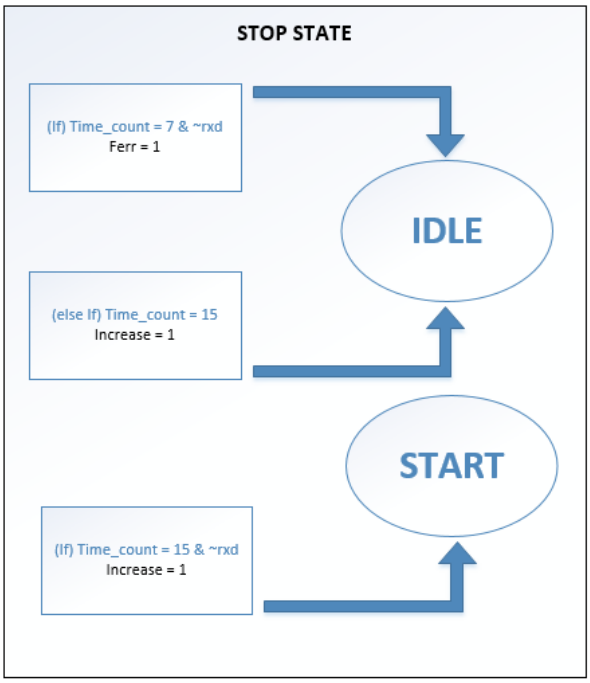
```
logic enable_data;
logic d0,d1,d2,d3,d4,d5,d6,d7;
assign data = {d7,d6,d5,d4,d3,d2,d1,d0};
        always_ff@(posedge clk)
        begin
            if(enable_data)
            begin
              case (bit_count)
                    3'd0 : d0 = rxd;
                    3'd1 : d1 = rxd;
                    3'd2 : d2 = rxd;
                    3'd3 : d3 = rxd;
                    3'd4 : d4 = rxd;
                    3'd5 : d5 = rxd;
                    3'd6 : d6 = rxd;
                    3'd7 : d7 = rxd;
              endcase
            end
end // always_ff
```

STOP: This state is necessary for the receival of the data to happen to facilitate rdy to be asserted high to indicate that the receiver is ready to receive new data because the current receival is finished.









The details can be seen in these diagrams above

Lastly, we have added some extra features for easy use and quick display. We have seven segment display for the data output from the receiver and this keeps the last value of the data meaning that the last sent data will be shown until a new one takes its place.

From first lab, we have implemented the following module:

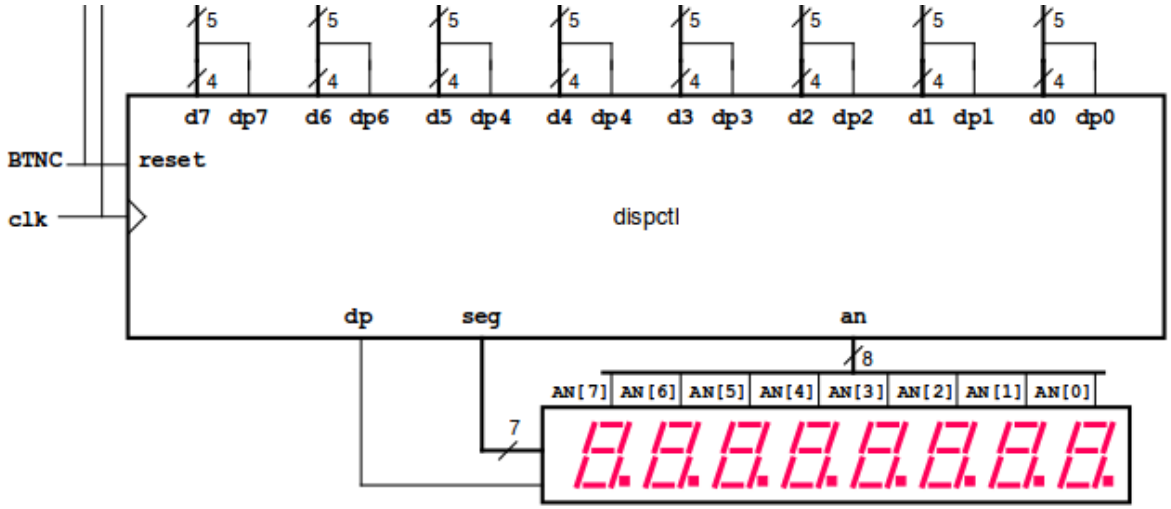
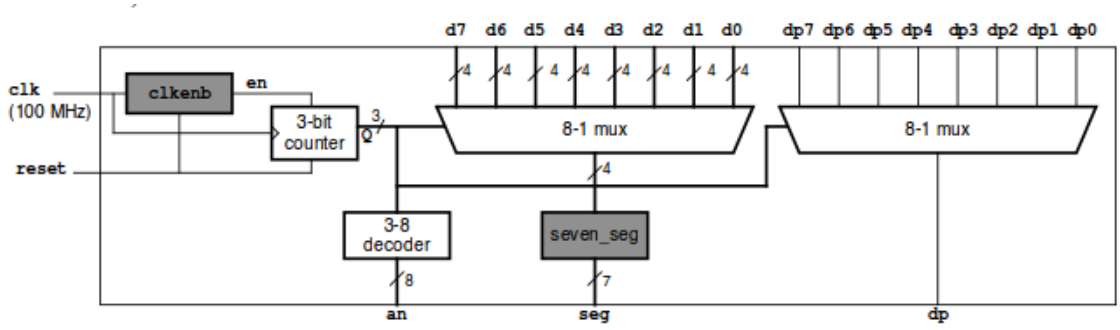


Figure 4 – `dispctl` Test Circuit

The only difference is that the input is not from the switches but from data output of the receiver. Therefore, we do not pass 5 bits but only 2 bits of data for d and dp pairs. Additionally, d0 to d7 receives the specific data[0] to data[7] bits but all of the dp values are tied to 1'b0 for simplicity.

Also, inside the dispctl module:



we have changed the mux input bits to 1 which made the output to be only ONE or ZERO, ignoring the values from TWO to FIFTEEN.

Lastly, we have added the transmitter from lab 2 for easy user input data transmission without the need of RealTerm.

```
        logic temptxd;
        always_ff@(posedge CLK100MHZ)
        begin
            if(SW[8])
            begin
                temptxd <= UART_RXD_OUT;
            end
            else
            begin
                temptxd <= UART_TXD_IN;
            end
end
```

And here UART_RXD_OUT is the output of the transmitter and UART_TXD_IN is connected to the usb which gives access to RealTerm.

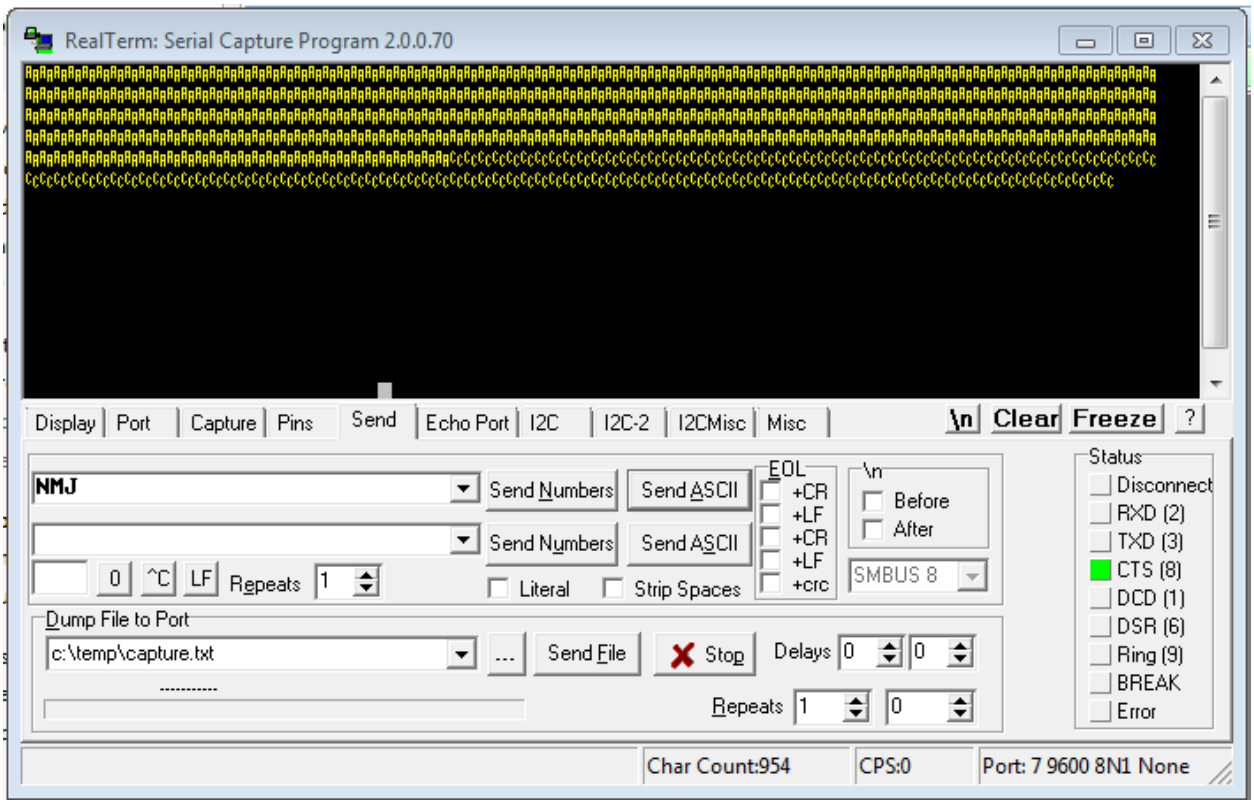
3. Design Verification

Description	Test Method	Detailed Results
1. Module Interface	Code Inspection	We will add few extra inputs and outputs compared to the Lab design, i.e. 1. Check for error in the start bit by adding a 3rd check in the START state. 2. Added Transmitter from Lab2 for switching in between RealTerm data input and user given switches data input
2. Module function: accepts rxd input and receives data bits one by one, ferr output for when there is a frame error and indicates readiness for the next receiving with rdy output	Demonstration in oscilloscope and test bench simulation and Nexys4DDR board: **TEST BENCH** 1. Proper display of rdy output with a LED of the board 2. Proper display of 10101010, 11001100 data transmissions on test bench 3. Proper display of time_count and bit_count on the test bench 4. Display of data received in test bench 5. Display of STOP-START transition in the test bench 6. Proper display of reset on the test bench 7. Proper display of the BaudRate on the test bench	**OSCILLOSCOPE & FPGA HARDWARE** 1. Proper display of rdy output with a LED of the board 2. Proper display of 01010101, 00110011, 00001111, 00000000, 11111111 data inputs with seven segment display Nexys board 3. Proper display of BaudRate with oscilloscope using the rdy state 4. Proper display of rdy output on oscilloscope 5. Proper display of reset on seven segment display on Nexys board 6. Proper display of ferr on the oscilloscope
3. Uses Nexys4 board 100Mhz clock; all flip-flop clock inputs tied directly to this signal	Code inspection *(all the instances of the clk use in the modules are provided)*	Provide the clock report from vivado
4. Contains no latches	Inspection of Synthesis Report	Provide the section in vivado synthesis report
5. Test circuit – show test that test circuit functions properly to exercises circuit.	Demonstration in hardware 1. Demonstrated 10101010, 11001100, 11110000, 00000000, 11111111 on the seven segments and the oscilloscope 2. Demonstrated two consecutive data transmission of NM letters through RealTerm 3. Demostrated BaudRate from rdy signal 4. Demostrated not sticky ferr signal 5. Demostrated every bit of the data after receival 6. Demostrated the transition from user input transmitter to RealTerm input	Accepted Demonstration by Professor Nadovich (except the ferr output)

In submitting this checklist as part of our report, I/We certify that the tests described above were conducted and that the results of these tests are accurately described and represented. I/We understand that any misrepresentation of the tests or the results constitutes a violation of the College policy on academic dishonesty.
Name(s):Kemal Dilsiz & Zainab Hussein *Date: 10/04/2016*

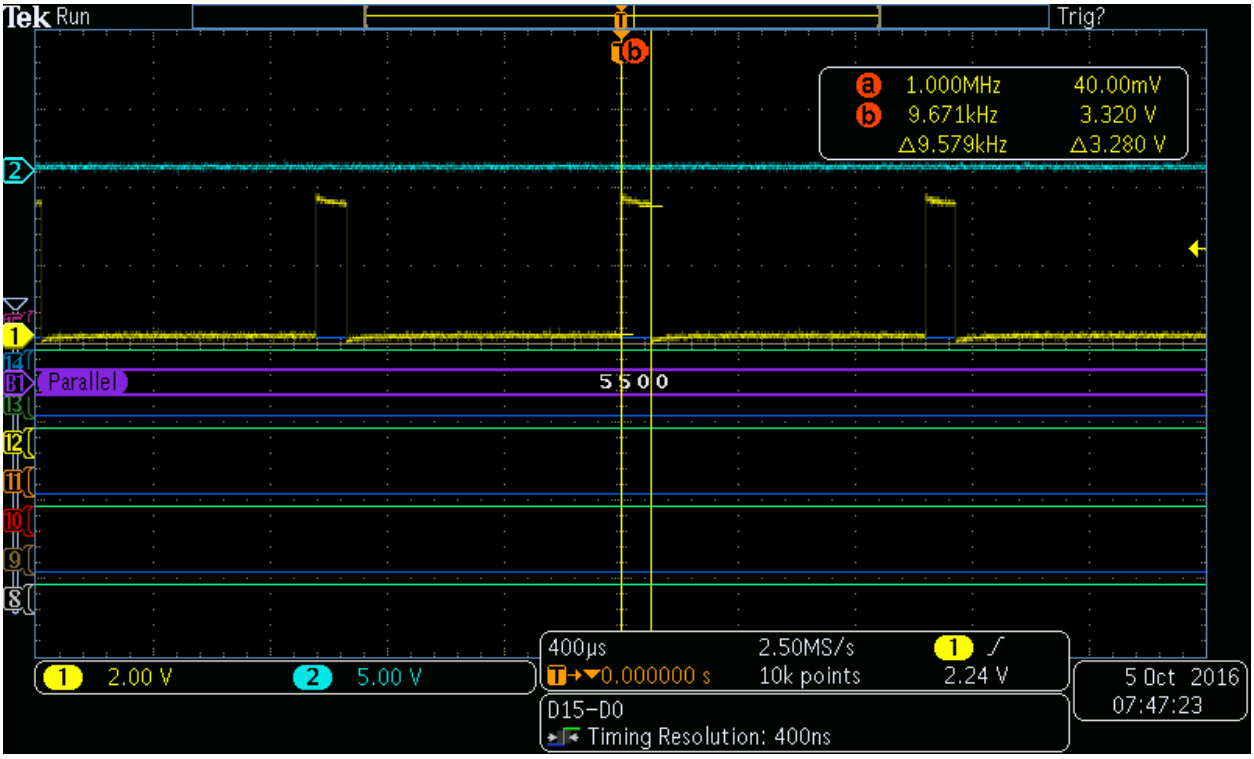
Here are the pictures:

Realterm shows the data that has been sent to the oscilloscope. Data N,M and J sent to oscilloscope through command: send ASCII. Ferr set by setting a page break through: Pins then SET PAGEBREAK.

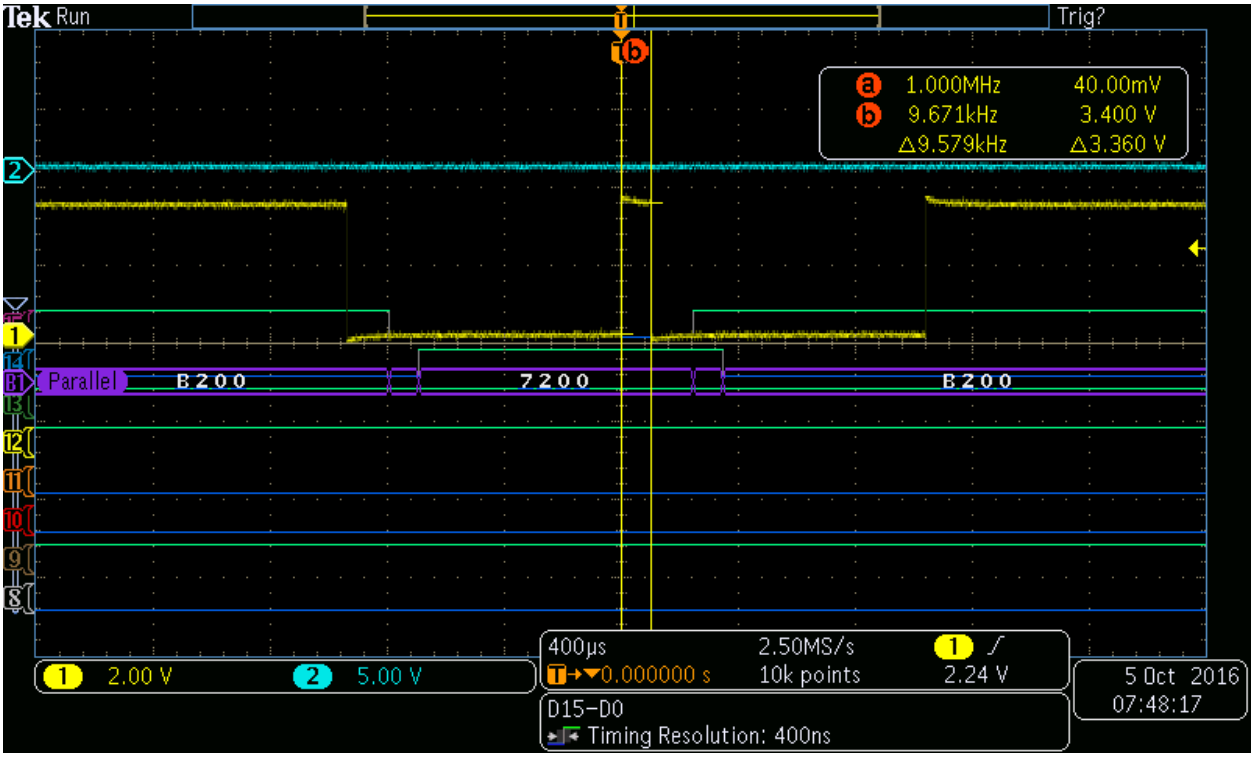


The oscilloscope shows consecutive data bytes receival:

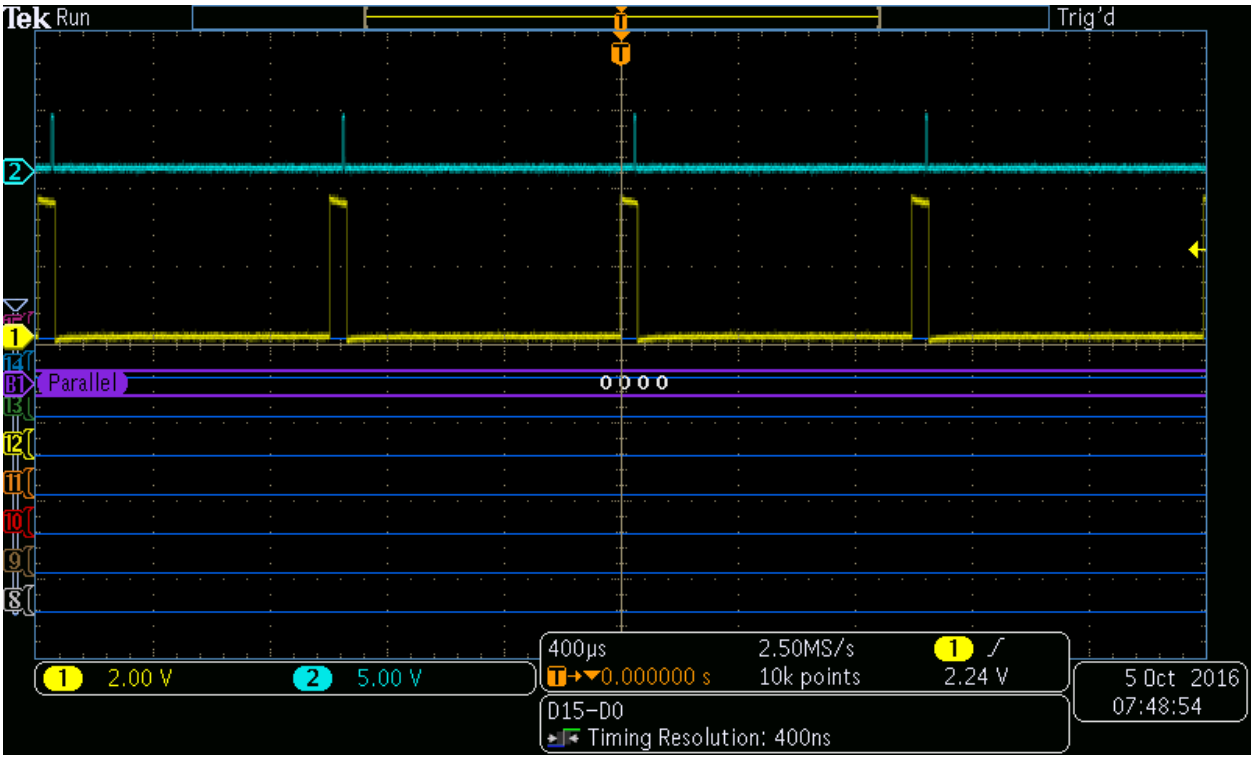
1. Shows baudrate, rdy yellow and single 0101010101



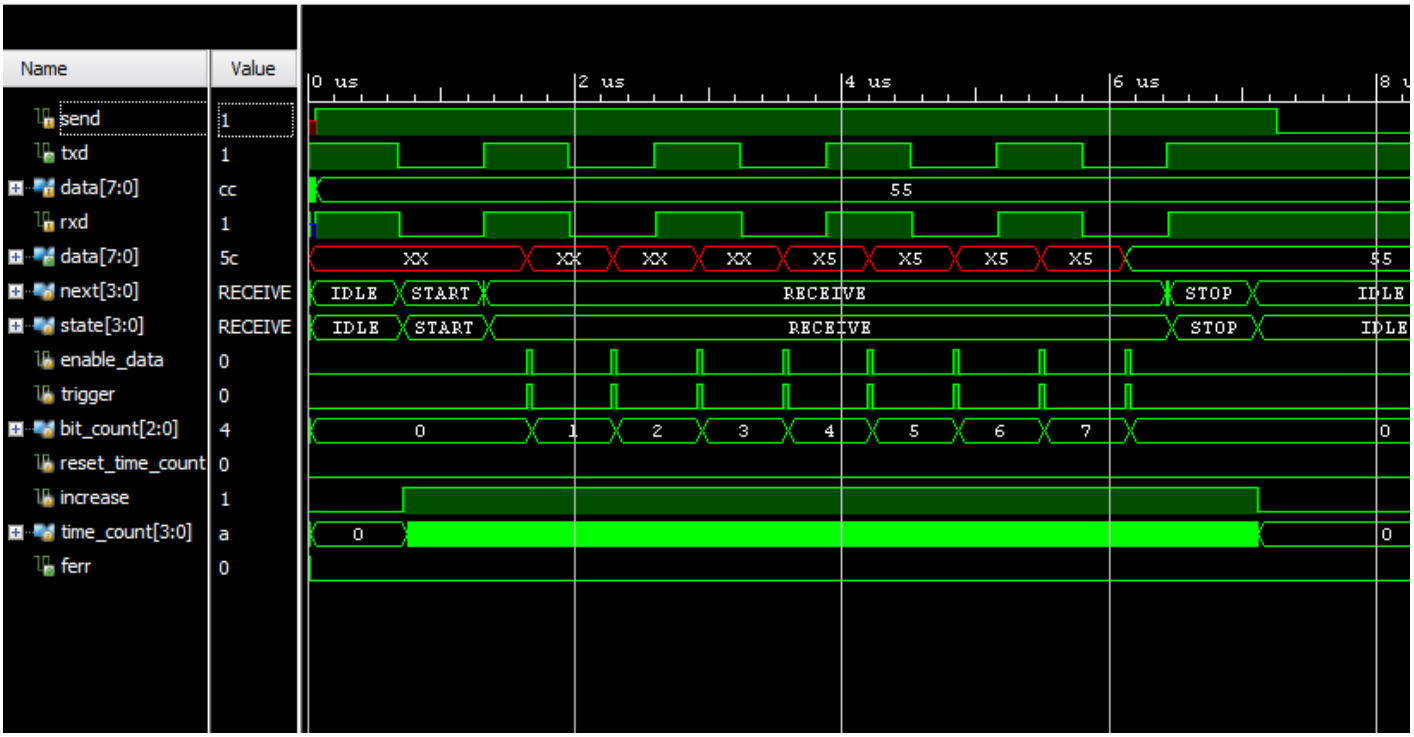
2. Shows baudrate, rdy in yellow and consecutive data bytes N, M and J from realterm

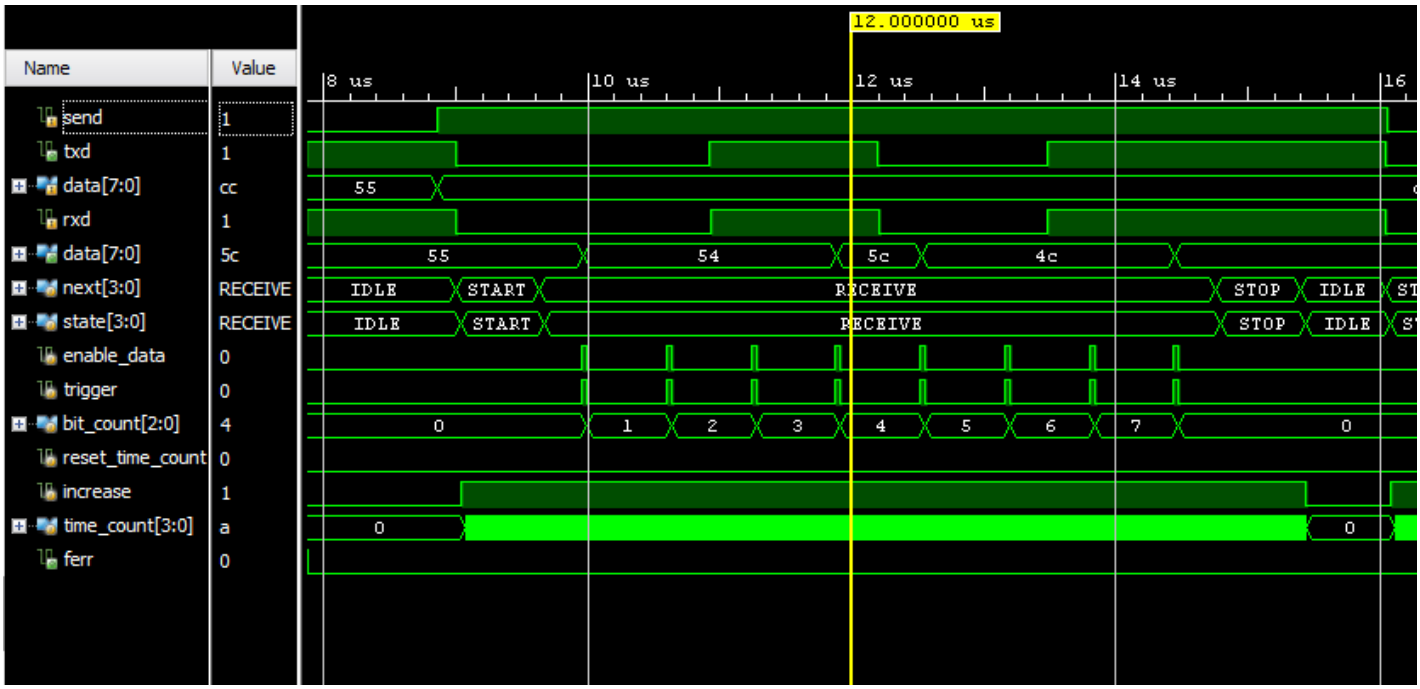


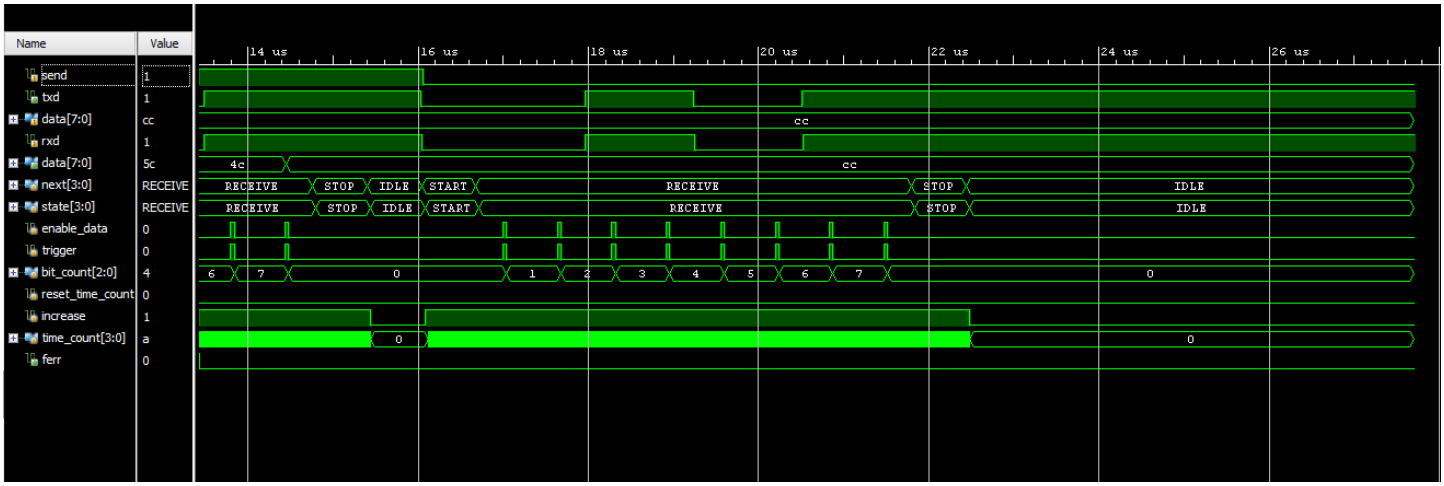
3. Shows ferr when page break is set from realterm. (ferr is turquoise, rdy is yellow and data set to zero shown on bus)



Testbench simulation of the top module shown below of the consecutive data 10101010 and 11001100:







4. Conclusion

We accomplished building an asynchronous data receiver and used our previous transmitter and seven segment display modules to show the functionality of our hardware.

Lessons Learned:

> More experience with test benches and debugging

> Experience with multi input oscilloscope cable and display

> Paying attention to even simple errors in the Synthesis report

References

No references for this lab. Except for the lab manual.