UNIVERSITY OF LIVERPOOL

ELEC473, ASSIGNMENT 1

Hague, Benjamin

NOVEMBER 29, 2019

THE UNIVERSITY OF LIVERPOOL

CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

1 INTRODUCTION

1.1 OBJECTIVES

1.1.1 PART A

The first part of this assignment contains designing and implementing a UART (Universal Asynchronous Receiver-Transmitter) system on the altera DE2 board. The system will take input from switches 12 to 6 and present the output in hexadecimal on the 7 segment displays (Hex0 and Hex1). The system will use an even parity and send 7 bits of data. The system must be able to communicate with a PC through a UART cable.

1.1.2 PART B

The system must then be further adapted to comply with the IrDA standard for data transmission.

1.2 INTRODUCTION TO THE UART STANDARD

UART is a recognised standard for bidirectional asynchronous data transfer between devices.

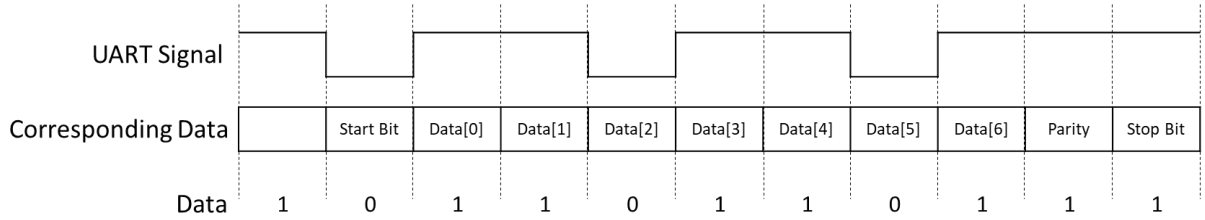


FIGURE 1, UART DATA TRANSMISSION

Figure 1 visually represents the standard. The signal is transmitted with a set Baud Rate, this is equal to the amount of time each data bit is sent for the signal is of known length, and the start and end of the signal are denoted with start and stop bits, (0 and 1 respectively). It is common practice to account for a parity bit at the end of the data signal.

Figure 1 shows the data that will be transmitted within the system implemented here.

2 TRANSMITTER

The Transmitter is where the remanded of the blocks are connected the top-level diagram for this section is shown in **FIGURE 2**. All the blocks featured here were written for this module. The code reuse in this module is minimal. This module takes advantage of the Parity, the baud counter, the bit counter, the controller and the shift register.

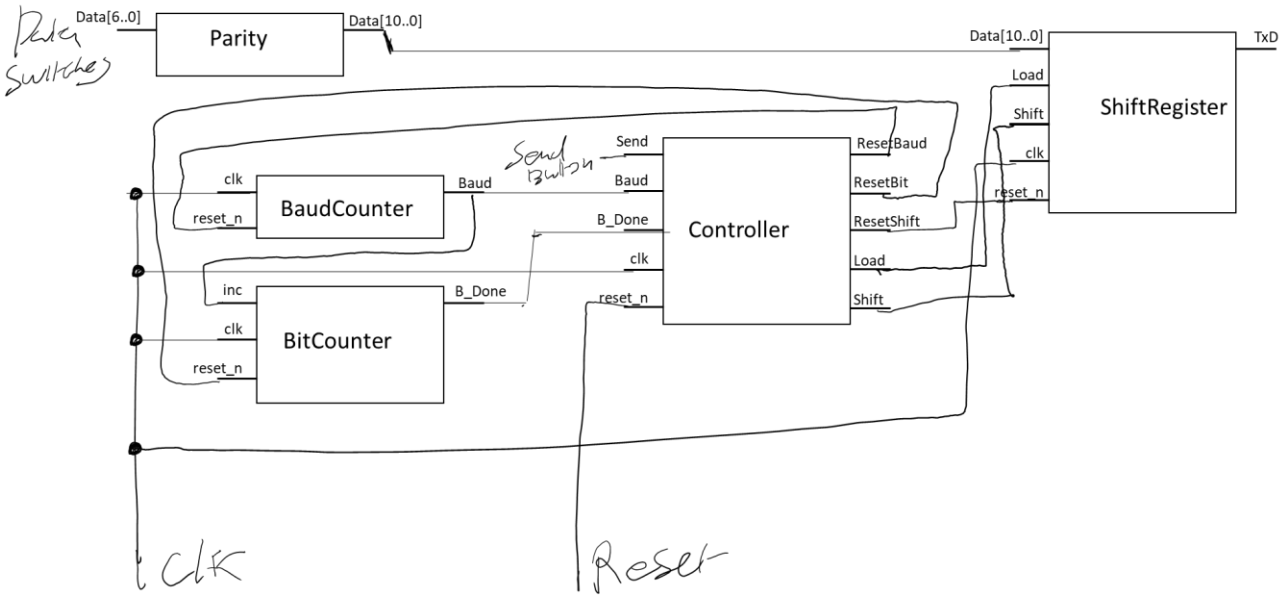


FIGURE 2, CONNECTED TRANSMITTER BLOCK DIAGRAM

The debounce module has been omitted for simplicity. The diagram below, Figure 3, shows the BDF as drawn and simulated within the Quartus software.

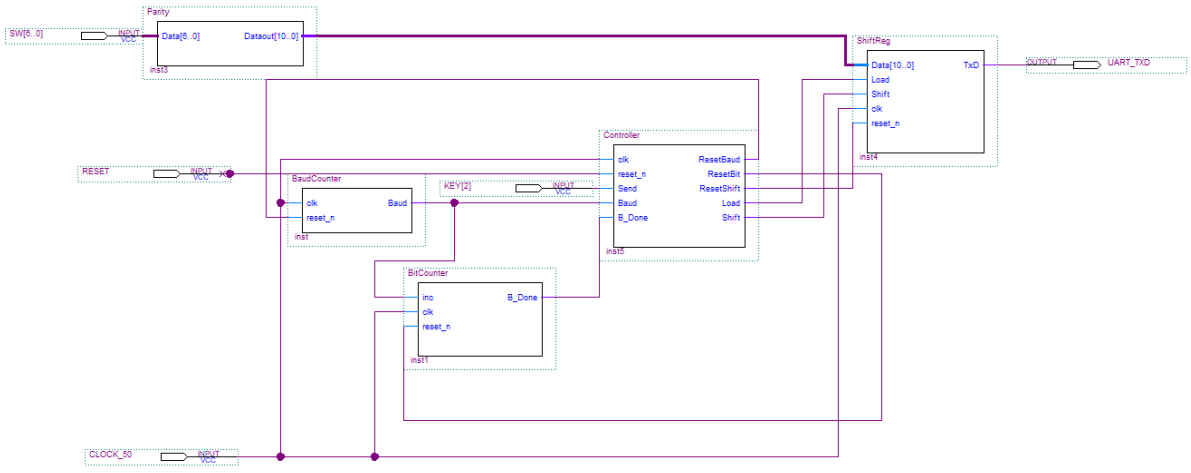


FIGURE 3, TRANSMITTER BDF IN QUARTUS SOFTWARE

3 RECEIVER

The receiver is included within the top-level diagram along with the transmitter, the block diagram for the receiver is shown Figure 4. the Receiver takes advantage of the modular format of the block design and builds upon the blocks designed and built for the transmitter.

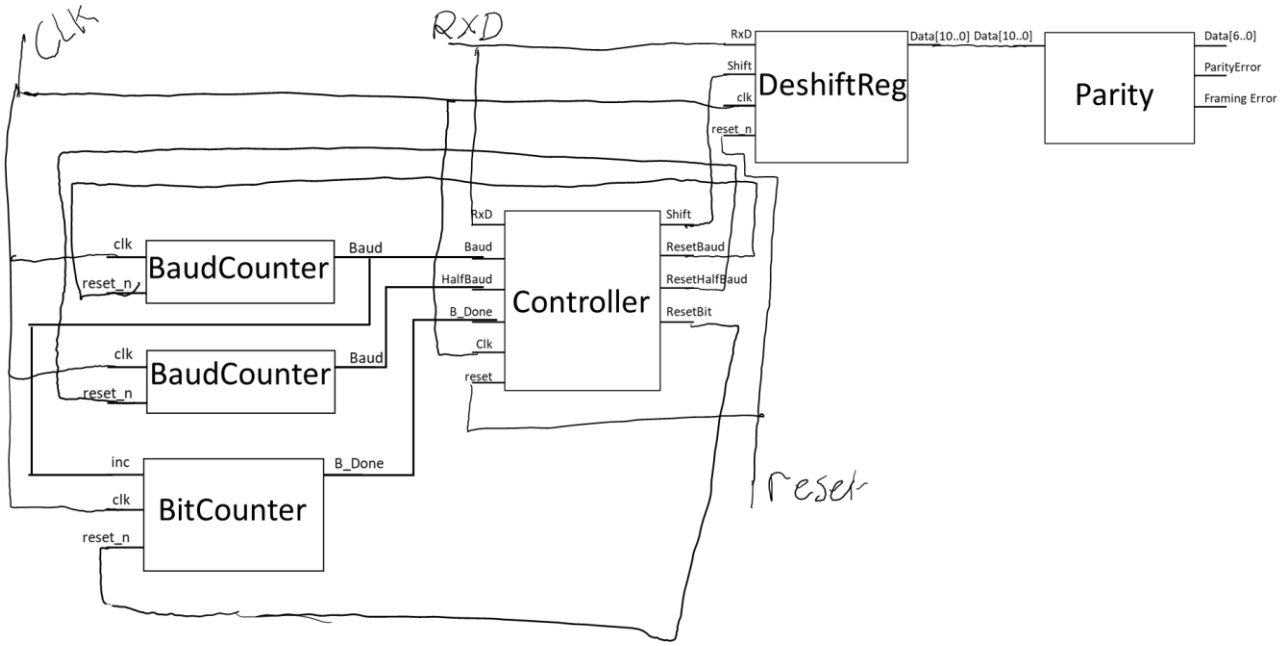


FIGURE 4, CONNECTED RECEIVER BLOCK DIAGRAM

Figure 5 shows the receiver drawn within the Quartus software.

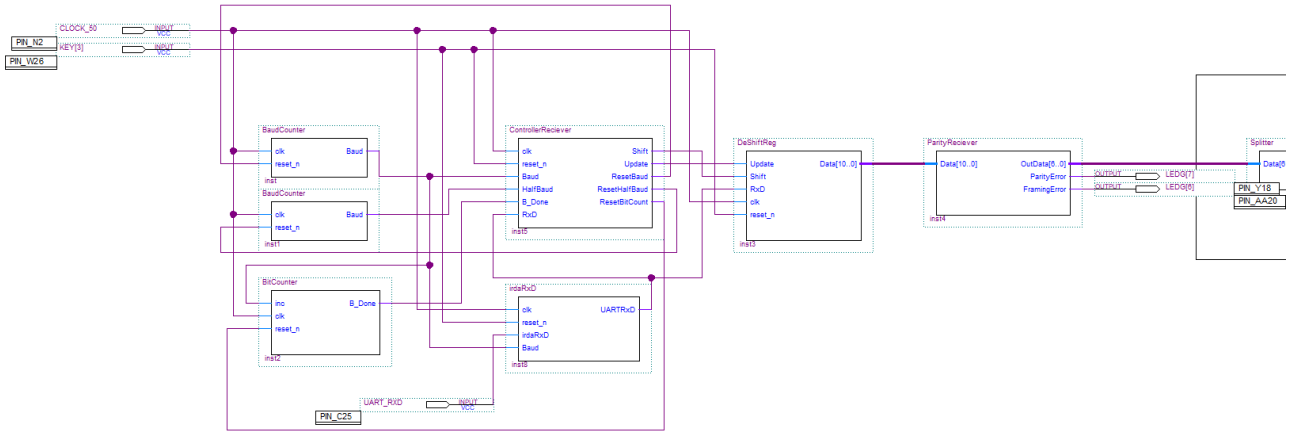


FIGURE 5, RECEIVER WITHIN QUARTUS SOFTWARE

For Completeness the 7 segment decoder is shown in Figure 6.

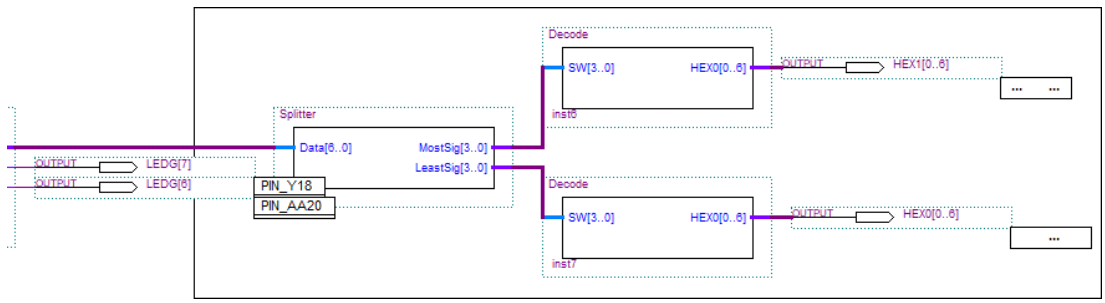


FIGURE 6, SEVEN SEGMENT DECODER

4 SHARED MODULES

4.1 BAUD GENERATOR (BAUDCOUNTER.V)

The baud generator is a vital part of both the transmission and receiving circuits. The baud generator counts clock cycles, once a critical number is reached the Baud goes high, and the counter resets to go again. Figure 7 shows the block diagram.

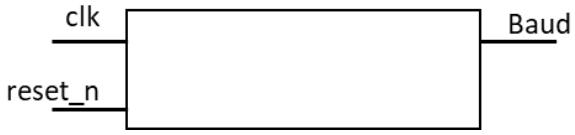


FIGURE 7, BAUD GENERATOR BLOCK DIAGRAM

4.1.1 ASM

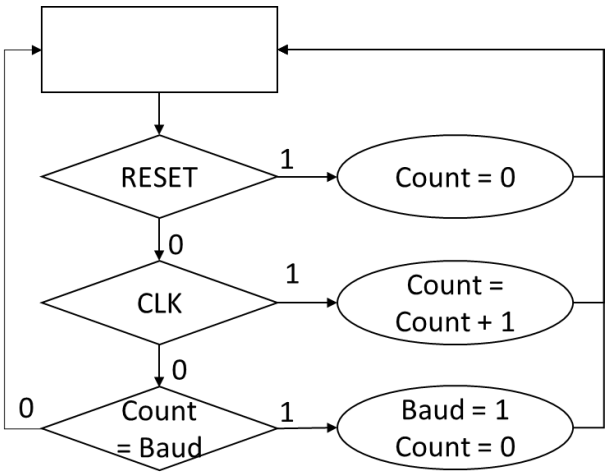


FIGURE 8, BAUD GENERATOR ASM

Figure 8, shows the ASM for the Baud Generator, The value for Baud is equivalent to the number of clock cycles per bit sent, this is calculated using Equation 1.

EQUATION 1, CALCULATION FOR BIT TIME IN CLOCK CYCLES

$$ClockPeriod = \frac{1}{(50 \times 10^6)}$$

$$BitLength = \frac{1}{(38400)}$$

$$Baud = \frac{BitLength}{ClockPeriod}$$

4.1.2 VERILOG

```
module BaudCounter (clk, reset_n, Baud);
	// Set Inputs, clk and reset_n
	input clk, reset_n;
	// Set Baud as a reg output
	// Must be reg to allow writing to the output
	output reg Baud;
```

```
	// declare 2 counts (Maximum counted to as 1302, or
BaudRate/ClockPeriod)
	reg [10:0] pcount, ncount;
	// Declare the stop point for the count as a parameter, Allows it to
change in BDF
	parameter maxCount = 11'd1302;

	// Synchronous block with asynchronous reset
	always @(posedge clk, negedge reset_n)
	if (reset_n == 0)
		pcount<=11'b0;
	else
		pcount<=ncount;

	// Execute the following block at update of pcount
	always @(pcount)
	begin
		// Set Defaults to prevent latches
		Baud = 0;
		ncount = pcount + 1'b1;
		// When finished counting, set baud high and reset the Counter
to 0
		if (pcount == maxCount)
			begin
				Baud = 1;
				ncount = 11'd0;
			end
	end
endmodule
```

4.1.3 TESTING

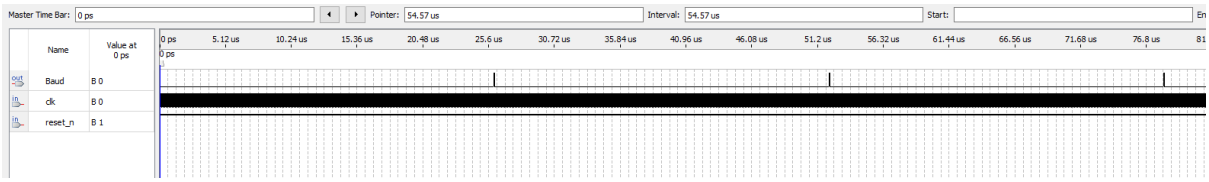


FIGURE 9, FUNCTIONAL SIMULATION FOR BAUD COUNTER

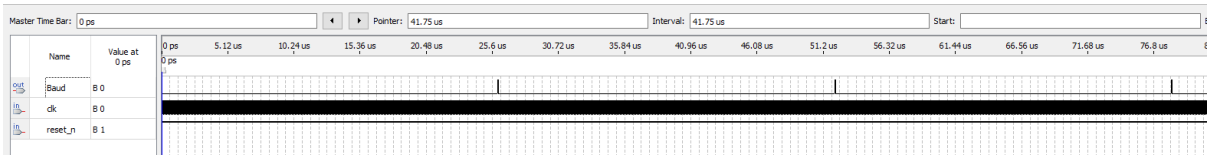


FIGURE 10, TIMING SIMULATION FOR BAUD COUNTER

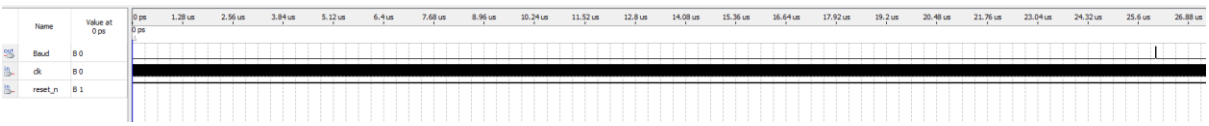


FIGURE 11, BAUD SIMULATION VERIFICATION

Figure 9 and Figure 10 show the functional and timing simulations for the baud counter, we can see with these simulations that the baud becomes high for one clock cycle at a regular interval. Equation

2 verifies that the baud signal occurs every 26µs which corresponds to the timing shown in Figure 11.

EQUATION 2, BAUD VERIFICATION

$$\frac{1}{38400} \approx 26 \times 10^{-5}$$

This simple module works exactly as expected, declaring the maximum count as a parameter allows us to change the module according to the relevant demand. This is useful to in later parts of the assignment.

4.2 INPUT BUTTON DEBOUNCER (DEBOUNCER.V)

The purpose of this module is to ensure that the send key only holds the signal high for a short period, this prevents the continuous sending of data and allows a single bit to be sent. The block diagram is shown in Figure 12.

FIGURE 12, DEBOUNCE BLOCK DIAGRAM

4.2.1 ASM

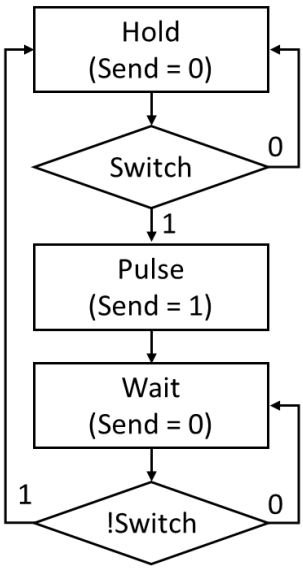


FIGURE 13, DEBOUNCE ASM

4.2.2 VERILOG

```
module debouncer(switch, Send, clk, reset_n);
	// define clock and reset input
	input clk, reset_n;
	//Define Switch input
	input switch;
```

```
	//Define Output
	output reg Send;

	//define state parameters

	parameter [1:0] Hold = 2'b00, Pulse = 2'b01, Wait = 2'b10;

	//define current and next states
	reg [1:0] pstate = Hold, nstate;

	//Define Synchronous state movement
	always @(posedge clk, negedge reset_n)
		if (reset_n == 0)
			pstate = Hold;
		else
			pstate = nstate;

	//State Machine
	always @(clk,switch)
	begin
		// Set Defaults to prevent latches
		nstate = pstate;
		Send = 0;
		case(pstate)
		Hold:
			if (switch)
			begin
				nstate = Pulse;
			end
		Pulse:
			begin
				nstate = Wait;
				Send = 1;
			end
		Wait:
			if(!switch)
				nstate = Hold;
		endcase
	end
endmodule
```

4.2.3 Testing

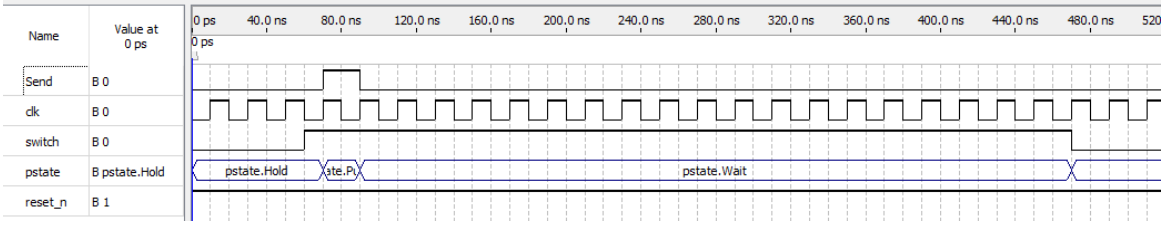


Figure 14, Testing Debounce

Figure 14 shows the testing of the debounce module, we can see that the first full clock cycle is turned into a single clock length pulse. The send key is then held low until the button is next pressed having been cycled down for a full clock cycle. This functions exactly as expected.

4.3 BIT COUNTER (BITCOUNTER.V)

The bit counter counts the number of bits being transmitted (10) and once the critical number is reached sets the signal B_Done to high, this tells the system that the full message has been sent. Figure 15 Shows the block diagram for the bit counter.

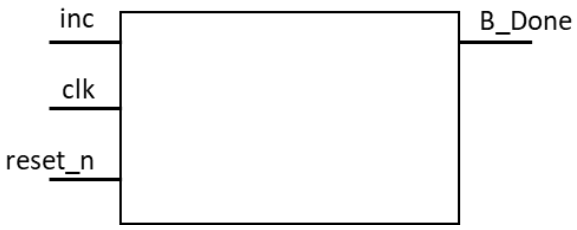


FIGURE 15, BLOCK DIAGRAM FOR BIT COUNTER

4.3.1 ASM

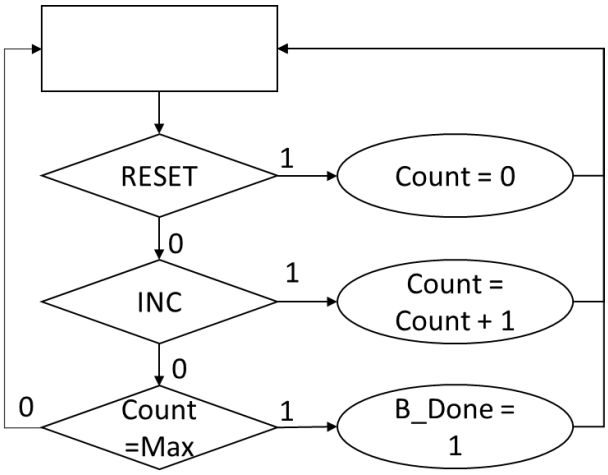


FIGURE 16, BIT COUNTER ASM

Figure 16, shows the ASM for the Bit counter, the Max value is defined as a parameter and allows the count to be more flexible for a wider range of uses.

4.3.2 VERILOG

```
module BitCounter (inc, clk, reset_n, B_Done);
	// Set Inputs, clk and reset_n
	input  clk, reset_n;
	// set inc as seperate input
	input inc;
	// Set B_Done as a reg output
	// Must be reg to allow writing to the output
	output reg B_Done;

	// declare 2 counts (Maximum counted to as 10, or start + stop + data
+ parity)
	reg [3:0] pcount, ncount;
	// Declare the stop point for the count as a parameter, Allows it to
change in BDF
	parameter maxCount = 4'b1010;

	// Synchronous block with asynchronous reset
	always @(posedge clk, negedge reset_n)
	if (reset_n == 0)
```

```
            pcount<=4'b0000;
      else
            pcount<=ncount;

      // Execute the following block at update of inc and pcount.
      always @(pcount,inc)
      begin
            // Set Defaults to prevent latches
            B_Done = 0;
            ncount = pcount;
            // When inc is high, increment counter
            if (inc)
                  begin
                        ncount = pcount + 1'b1;
                  end
            // When finished counting, set B_Done high
            if (pcount == maxCount)
                  begin
                        B_Done = 1;
                  end
      end
endmodule
```

4.3.3 TESTING

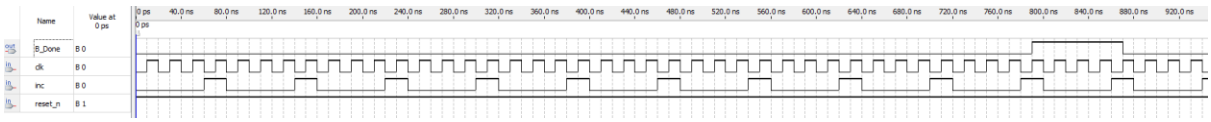


FIGURE 17, FUNCTIONAL SIMULATION OF BIT COUNTER

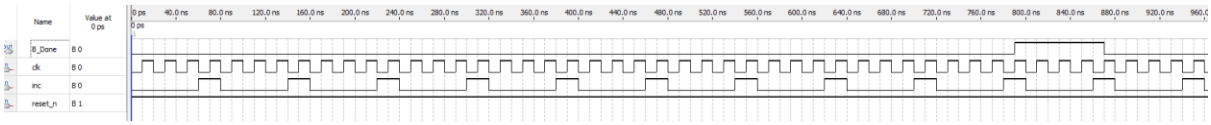


FIGURE 18, TIMING SIMULATION OF BIT COUNTER

Figure 17 and Figure 18, show the simulations for the Bit Counter module, within this module we can see that for the time where the increment has been triggered 10 times, the B_Done signal becomes high. We can also see that there is no timing issues present.

This module is incredibly similar to the former module, baud generator, the increment on this module is however external rather than on the clock. This module functions correctly as shown.

5 TRANSMITTER MODULES

5.1 PARITY GENERATOR (PARITY.V)

5.1.1 DESCRIPTION

This module takes the input from the switches, (insert here), Calculates the parity, and adds the start and stop bits. This module is made up solely of combinational logic. The breakdown of the output is shown in Table 1. This details what data will correspond to what location within the output bus. Whilst a simple ASM can be done for this section, due to the lack of states and the asynchronous aspect of this module the ASM has been neglected. Figure 19 shows the block diagram for this module.

TABLE 1, OUTPUT DATA MAP FOR PARITY GENERATOR

Location	Data Stored there
Data[0]	Start Bit (0)
Data[1]	Input data Length of 7
Data[2]	
Data[3]	
Data[4]	
Data[5]	
Data[6]	
Data[7]	
Data[8]	Parity Bit
Data[9]	Stop Bit (1)
Data[10]	Stop Bit (1)

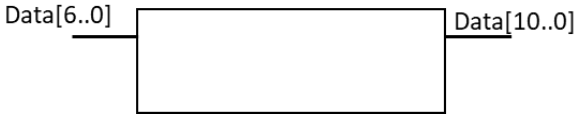


FIGURE 19, BLOCK DIAGRAM FOR PARITY GENERATOR

5.1.2 VERILOG

```
module Parity(Data, Dataout);
	// No Clock and Reset in this block as it is purely combinational
logic.
	// Input for Data (width 7)
	input [6:0] Data;
	// Output for Data (width 11) with start stop and parity added
	output reg [10:0] Dataout;
	// ParityBit must be reg to write data to it
	// Here to help readability
	reg ParityBit;

	// Start and Stop Bits as registers so they can be updated in the BDF
	parameter startBit = 1'b0, stopBit = 2'b11;

	// Execute block on change of data
	always @ (Data)
		begin
			// Even Parity declaration
			ParityBit = !(^Data);
```

```
            // Concoct the data output
            Dataout = {stopBit, ParityBit, Data, startBit};
        end
endmodule
```

5.1.3 TESTING

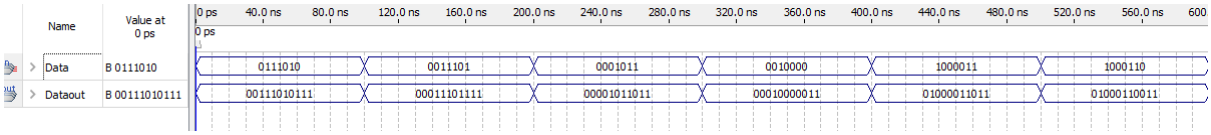


FIGURE 20, SIMULATION OF PARITY MODULE

Figure 20 shows a variety of inputs and the outputs that correspond to them for this module, due to the lack of clock in this module, it is not necessary to do both timing and functional simulations. We can see that throughout the execution, start, stop and parity bits are added to the signal.

5.2 SHIFT REGISTER (SHIFTREG.V)

This module takes input from the load button, the bit counter, baud generator and the parity bit. It takes the data from the parity bit, shifts it one place and outputs the current lowest significant bit on the baud count. Once this is done the shift generator waits for the next shift signal. Figure 21 shows the Shift Register Block Diagram. This is the last module before the data is transmitted.

FIGURE 21, BLOCK DIAGRAM FOR SHIFT REGISTER

5.2.1 ASM

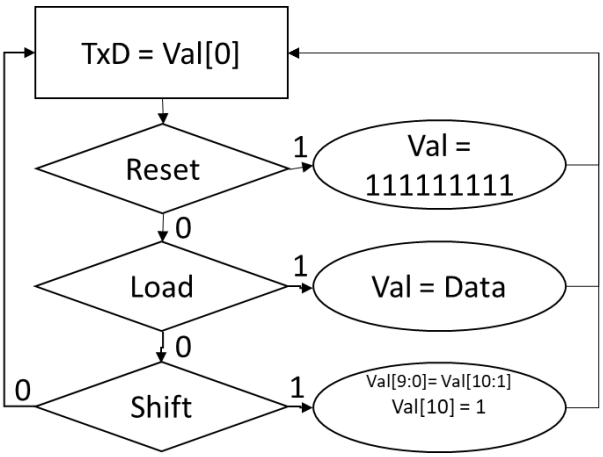


FIGURE 22, SHIFT REGISTER ASM

Figure 22 shows the ASM for the shift register, it is worth noting that the data within the register only updates when the load signal becomes high.

5.2.2 VERILOG

```
module ShiftReg(Data, Load, Shift, TxD, clk, reset_n);
	// Input for Clock and Reset, Always Needed
	input clk, reset_n;
	// Input for Data (width 11), Load and Shift
	input [10:0] Data;
	input Load, Shift;
	//Define the output TxD as reg to enable writing to it
	output reg TxD;

	// reg to hold Present and Next Value for cycled Data
	reg [10:0] p_val = defVal, n_val;

	// Define Default value as parameter
	parameter defVal = 11'b11111111111;

	// Synchronous block with asynchronous reset
	always @ (posedge clk, negedge reset_n)
		if (reset_n == 0)
			p_val <= defVal;
		else p_val <= n_val;

	// Always block executes on Load and Shift
	always @ (Load, Shift)
		begin
			//Set Defaults to prevent Latches
			n_val = p_val;
			TxD = p_val[0];
			//if loading Update the local data store
			if (Load)
				n_val = Data;
			else if (Shift == 1)
				// if shift is high, shift the data
				n_val = {1'b1, p_val[10:1] };
		end
endmodule
```

5.2.3 TESTING

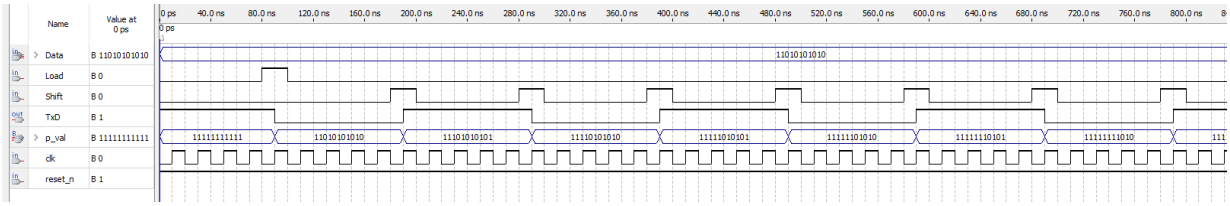


FIGURE 23, FUNCTIONAL SIMULATION OF SHIFT REGISTER

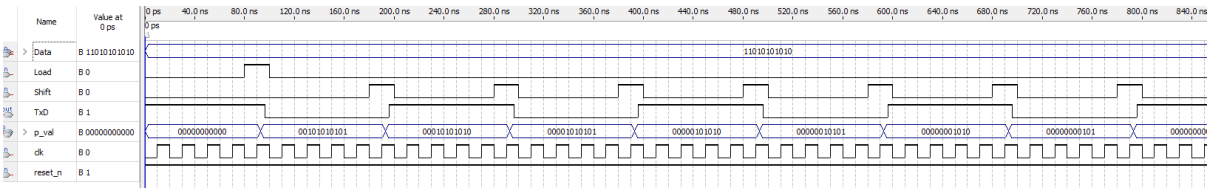


FIGURE 24, TIMING SIMULATION OF SHIFT REGISTER

Figure 23 and Figure 24 show the operation of the shift register, To aid the verification of this testing criteria we have chosen to also view the p_val within the waveform simulation. By looking at p_val in the simulations we can observe the bits shifting to the least significant bit. And the value of TxD changing to show the currently transmitted bit. We can observe within the timing simulation that propagation delay does not cause an issue within this module.

5.3 CONTROLLER (CONTROLLER.V)

The controller is a key part of the sending process. All the other modules are controlled here, this dictates when to reset, when to load data into the shift register and when to stop sending. The Block diagram is shown in Figure 25. This demonstrates the inputs and outputs for the system.

FIGURE 25, CONTROLLER BLOCK DIAGRAM

5.3.1 ASM

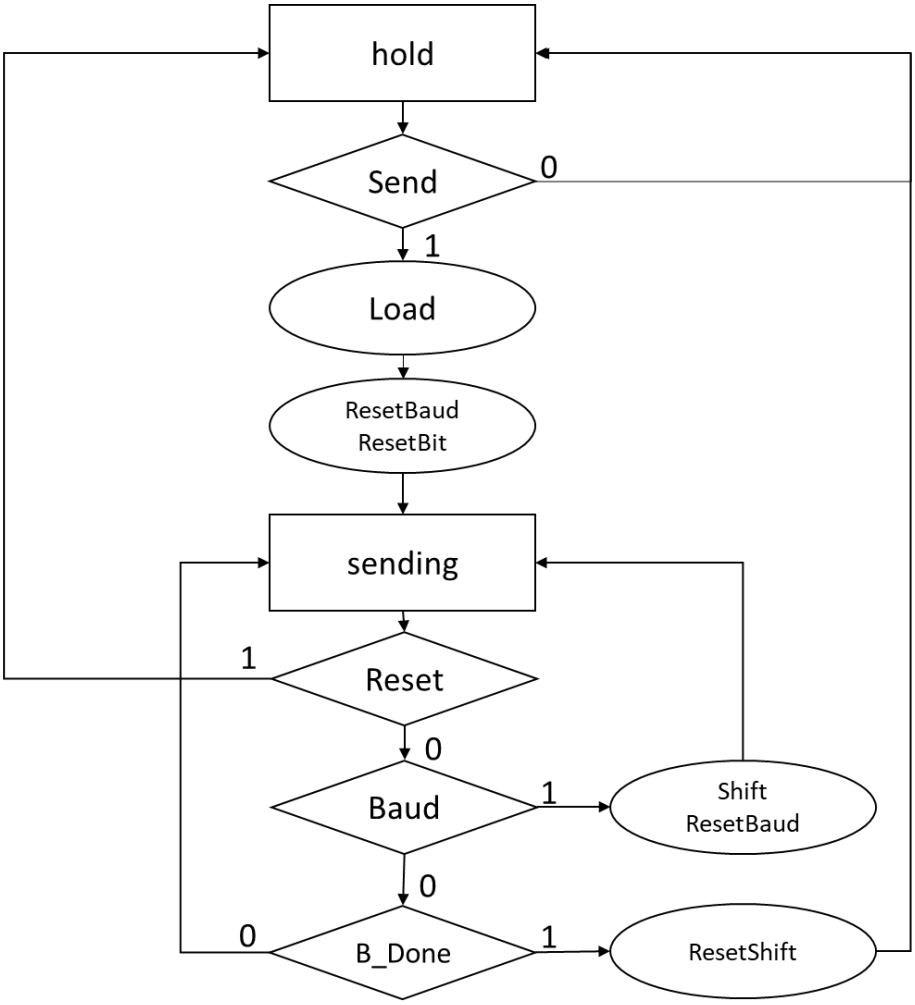


FIGURE 26, CONTROLLER ASM

5.3.2 VERILOG

```
module Controller(clk, reset_n,Send, Baud, B_Done,ResetBaud,
ResetBit,ResetShift, Load,Shift);
      //Define inputs for clk and reset
      input clk,reset_n;
      // inputs for send baud and bdone
      input Send,Baud,B_Done;
      //reset signal outputs
      output reg ResetBaud,ResetBit,ResetShift;
      //control signal outputs;
      output reg Load,Shift;

      //parameters for the states
      parameter hold = 1'b0, sending = 1'b1;
      // define the current and next state
      reg pstate,nstate;

      // synchronising block for reset and state transistion
   always @ (posedge clk, negedge reset_n)
      if (reset_n == 0)
            pstate <= hold;
      else pstate <= nstate;
```

```
	always @ (Send, Baud, B_Done, pstate)
		begin// Set Defaults (Reset Active High)
		ResetBaud = 1;
		ResetBit = 1;
		ResetShift = 1;
		// Set Defaults for control signals (Shift Reg)
		Load = 0;
		Shift = 0;
		// Set antiLatch State assignment
		nstate = pstate;
		// State Machine
		case (pstate)
		hold:
		begin
			if (Send) // if send is high reset the baud and bit then
move to sending state
				begin
					Load = 1;
					nstate = sending;
					ResetBaud = 0;
					ResetBit = 0;
				end
		end
		sending: //Sending State
		begin
			if (Baud)    //on Baud Shift and reset the Baud Count
			begin
				Shift = 1;
			end
			if (B_Done) //on send complete (B_Done) Reset the shifter
and move to hold state
			begin
				nstate = hold;
				ResetShift = 0;
			end

		end
		endcase
	end
endmodule
```

5.3.3 TESTING

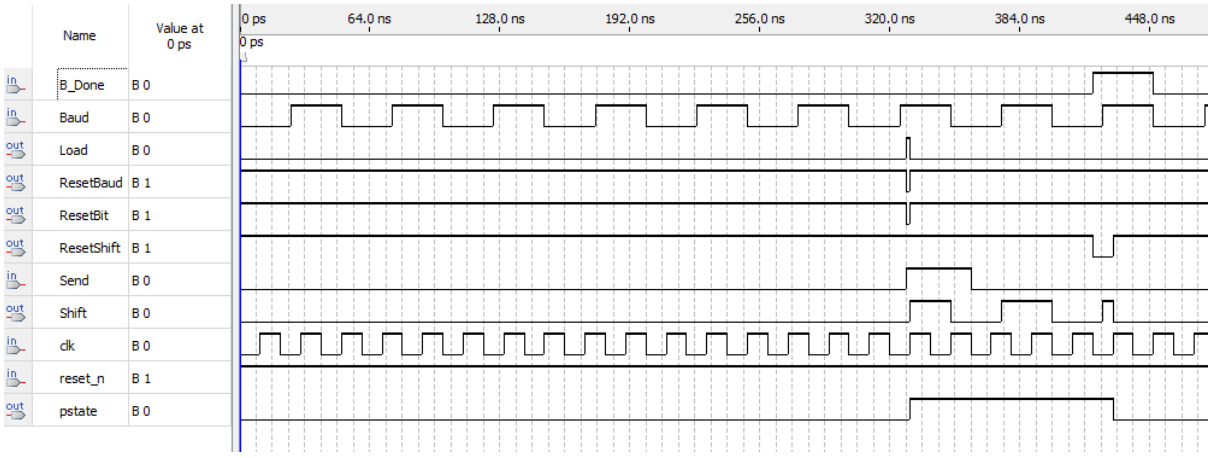


FIGURE 27, TESTING OF CONTROLLER

Figure 27 shows the testing of the controller, whilst looking complicated it shows that the system works as intended, on receipt of a send signal the bit and baud counters are reset, the shift reg is loaded and the data is instructed to shift on the occurrence of the baud. When B_Done is high the system then resets the shift register and awaits the next send command.

6 RECEIVER MODULES

6.1 PARITY GENERATOR (PARITYRECEIVER.V)

This module takes the input from receiving shift register, strips the data down, ensures it is framed correctly and checks if the data is corrupted by checking the parity. This module is made up solely of combinational logic. The breakdown of the input is shown in Table 2. Whilst a simple ASM can be done for this section, due to the lack of states and the asynchronous aspect of this module the ASM has been neglected. Figure 28 shows the block diagram for this module.

TABLE 2, PARITY GENERATOR INPUTS

Input Data Location	Used
Data[0]	Start Bit (0)
Data[1]	Output Data
Data[2]	
Data[3]	
Data[4]	
Data[5]	
Data[6]	
Data[7]	
Data[8]	Parity Bit
Data[9]	Stop Bit (1)
Data[10]	Stop Bit (1)

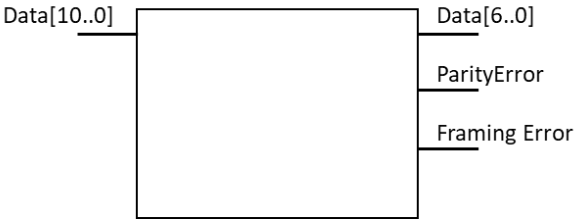


FIGURE 28, BLOCK DIAGRAM FOR PARITY GENERATOR

6.1.1 VERILOG

```
module ParityReciever(Data, OutData, ParityError, FramingError);
	// No Clock and Reset in this block as it is purely combinational
logic.

	// Input for Data (width 11)
	input [10:0] Data;

	//Output for Data width 6
	output [6:0] OutData;
	//output for ParityErrors and Framing errors
	output ParityError, FramingError;
```

```
        // Wires for start, stop and parity
        wire Parity;
        wire [2:0]StartStop;

        // Assign The output to the relevant data
        assign OutData[6:0] = Data[7:1];

        //Assign the parity
        assign Parity = Data[8];
        // Assign the concocted start and stop bits
        assign StartStop[2:0] = {Data[0], Data[10:9]};

        // Check if the parity bit corrosponds to data parity
        assign ParityError = ^OutData != Parity;
        //Check if the start stop bits corrospond with the expected start and
stop bits
        assign FramingError = (StartStop != 3'b011);

endmodule
```

6.1.2 TESTING

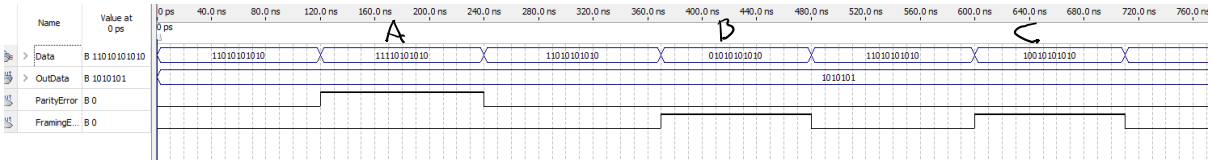


FIGURE 29, TESTING THE RECEIVER PARITY

Figure 29 shows a variety of inputs and the outputs that correspond to them for this module, due to the lack of clock in this module, it is not necessary to do both timing and functional simulations.

Within Figure 29 there are 3 labels, A, B and C.

Label A corresponds to a parity error, this is where the parity does not correspond to the parity of the signal. Label B and C each correspond to different framing errors. By looking at the OutData part of the output we can observe that the transmitted data does not change throughout causing these errors. We can therefore say that this module works as expected.

6.2 SHIFT REGISTER (DESHIFTREG.V)

The shift register for the UART decoder performs a similar operation to the UART encoder shift register, the data is shifted on the baud signal and the current value of RxD appended to the end of the register. The current value is then outputted to be seen by the rest of the system.

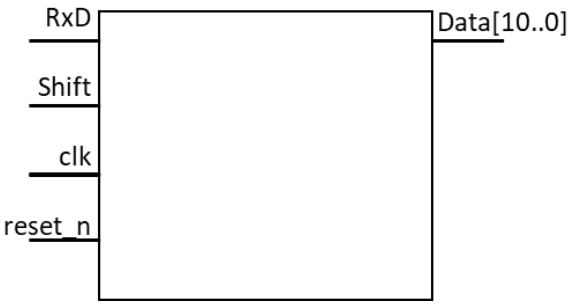


FIGURE 30, BLOCK DIAGRAM FOR SHIFT REGISTER

6.2.1 ASM

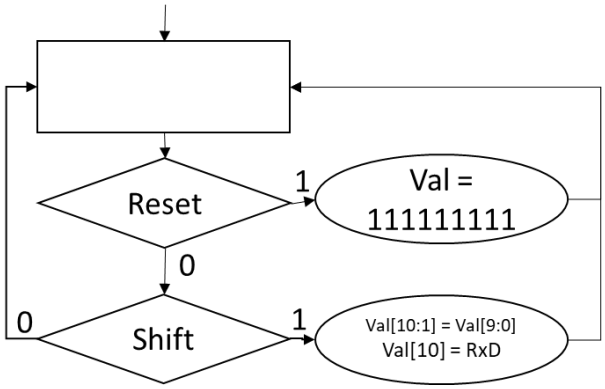


FIGURE 31, ASM FOR SHIFT REGISTER

6.2.2 VERILOG

```
module DeShiftReg(Data, Shift, RxD, clk, reset_n);
	// Input for Clock and Reset, Always Needed
	input clk, reset_n;
	// Input for RxD, and Shift
	input RxD;
	input Shift;
	// reg to hold Present and Next Value for cycled Data
	reg [10:0] p_val, n_val;
	//Reg to hold the currently outputted data
	output reg [10:0] Data;
	// Define the default value
	parameter defVal = 11'b01111111111;
	// Synchronous Block, on clock and reset update val or reset it
	always @ (posedge clk, negedge reset_n)
		if (reset_n == 0)
			p_val <= defVal;
		else p_val <= n_val;
	// functional block, on shift update the contens of n_val
	always @ (Shift)
		begin
		//Defaults to prevent latch synthsis
			n_val = p_val;
			Data = p_val;
			// functional if statement.
			if (Shift == 1)
				n_val = {RxD, p_val[10:1]};
		end
endmodule
```

6.2.3 TESTING

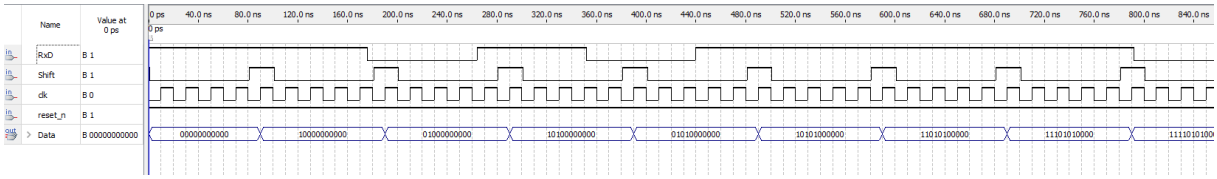


FIGURE 32, SIMULATION OF SHIFT REGISTER

Figure 32 shows us the basic functionality of the shift register, we can observe that on each of the shift commands the data is shifted and the value from RxD is added to the end of the register. We

can observe that the shifting occurs regularly and therefore this module operates exactly as expected.

6.3 CONTROLLER (CONTROLLERRECIEVER.V)

The controller for the receiver is responsible for triggering the shifting for sampling, resetting the baud and bit counters, and detecting the start of a signal. Figure 33 shows the block diagram for this module

FIGURE 33, BLOCK DIAGRAM FOR RECEIVER CONTROLLER

6.3.1 ASM

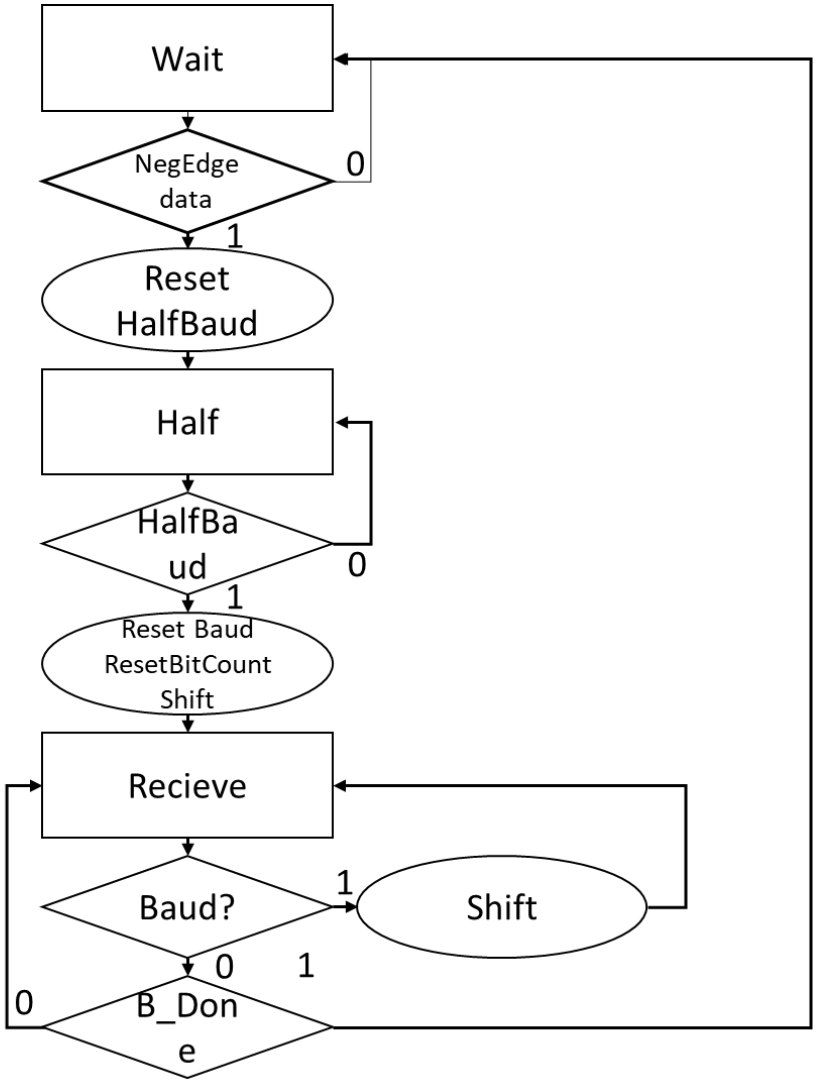


FIGURE 34, ASM FOR CONTROLLER

6.3.2 VERILOG

```
module ControllerReciever(clk,reset_n,Baud,HalfBaud,B_Done,RxD,
Shift,ResetBaud,ResetHalfBaud,ResetBitCount );
// clk and reset input(Active high)
input clk, reset_n;
// Event inputs
input Baud, HalfBaud, B_Done, RxD;
// Control signals outputs
output reg Shift,ResetBaud,ResetHalfBaud,ResetBitCount;
// Define States as parameters
parameter[1:0] Wait = 2'b00, Half =2'b01, Receive = 2'b11;
//reg for current and next state
reg[1:0] pstate, nstate;

// Recieving edge detection
reg Recieving;
always @(negedge RxD, posedge clk)
      if (RxD == 0)
            Recieving = 1;
      else Recieving = 0;
```

```
// synchronous block for state movement and asynchronous reset
always @ (posedge clk, negedge reset_n)
      if (reset_n == 0)
            pstate <= Wait;
      else pstate <= nstate;


always @(Shift, Baud, HalfBaud,pstate, B_Done, Recieving)
begin
      // Set defaults to prevent latch Synthasis
      // Reset High not low
      ResetBaud = 1;
      ResetHalfBaud = 1;
      ResetBitCount = 1;
      Shift = 0;
      nstate = pstate;
      // State Machine
      case (pstate)
      Wait: // Detect the start of the signal
                  // move to state Half
                  // reset half baud counter
      begin
            if (Recieving)
                  begin
                        ResetHalfBaud = 0;
                        nstate = Half;
                  end
      end
      Half: // wait until a half baud has appeared
      begin
            if (HalfBaud)
            begin
                  ResetBaud = 0;
                  ResetBitCount = 0;
                  nstate = Receive;
                  Shift = 1;
            end
      end
      Receive: // pass the shift command through
                        // but only until the end of the signal
      begin
            if (Baud)
            begin
                  Shift = 1;
            end
            if (B_Done)
            begin
                  nstate = Wait;
            end
      end
      endcase
end
endmodule
```

6.3.3 TESTING

v

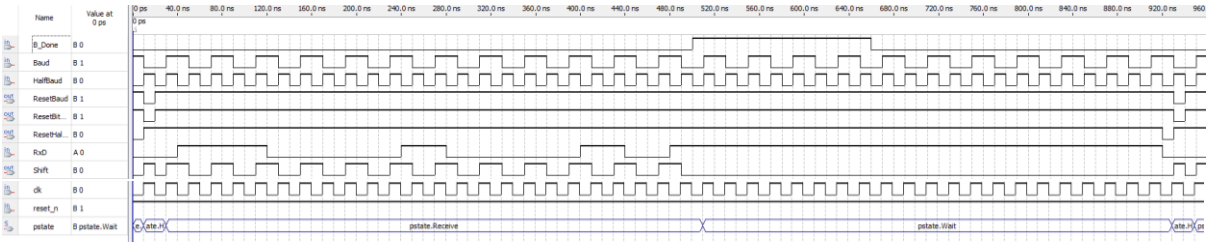


FIGURE 35, TESTING OF CONTROLLER FOR RECEIVING DATA

Although all sped up, we can see in Figure 35 the test for the controller above where we can observe the functionality of the controller, the system detects the negative edge of the signal, shown at 920ns and begins the operation, we can see the effects as the controller operates from 0-20ns for initialisation. The current state is shown at the bottom in the pstate row. This shows that the state is exactly as expected, being in the receive state for checking all the data then moving to the wait state until the next signal begins. The ½ state is used to delay the sampling.

6.4 SEPARATOR (FOR 7 SEGMENT DISPLAYS)

The separator module is a simple module for splitting the data into the inputs for the 2 seven segment display drivers. The most significant bit is added to ensure that the output from this module is 2 busses of width 4. This will be the simplest module in the whole system.

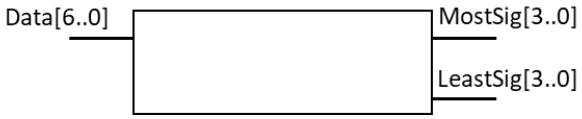


FIGURE 36, BLOCK DIAGRAM FOR SPLITTER MODULE

6.4.1 VERILOG

```
module Splitter(Data, MostSig, LeastSig);
	input [6:0] Data;
	output [3:0] MostSig, LeastSig;
	assign MostSig = {1'b0,Data[6:4]};
	assign LeastSig = Data[3:0];
endmodule
```

6.5 7 SEGMENT DECODERS

The 7-segment decoder relies fully on combinational logic. Table 3 shows the Truth Table for this module, as this is hard to understand in the context of numbers, the testing of this module took place directly on the altera board to ensure correct output.

TABLE 3, TRUTH TABLE FOR 7 SEGMENT DECODER

Input	Hex Input	Output
0000	0	000_0001
0001	1	100_1111
0010	2	001_0010
0011	3	000_0110
0100	4	100_1100
0101	5	010_0100
0110	6	010_0000

0111	7	000_1111
1000	8	000_0000
1001	9	000_0100
1010	A	000_1000
1011	B	110_0000
1100	C	011_0001
1101	D	100_0100
1110	E	011_0000
1111	F	011_n1000

6.5.1 VERILOG

```
module Decode(SW, HEX0);
	// Define inputs
	input [3:0] SW;
	// Define Outputs
	output reg [0:6] HEX0;

	always @(SW)	begin
	// Truth table for values 0 - 15
		case(SW)
			4'd0:HEX0 <=  7'b000_0001;
			4'd1:HEX0 <=  7'b100_1111;
			4'd2:HEX0 <=  7'b001_0010;
			4'd3:HEX0 <=  7'b000_0110;
			4'd4:HEX0 <=  7'b100_1100;
			4'd5:HEX0 <=  7'b010_0100;
			4'd6:HEX0 <=  7'b010_0000;
			4'd7:HEX0 <=  7'b000_1111;
			4'd8:HEX0 <=  7'b000_0000;
			4'd9:HEX0 <=  7'b000_0100;
			4'd10:HEX0 <= 7'b000_1000;
			4'd11:HEX0 <= 7'b110_0000;
			4'd12:HEX0 <= 7'b011_0001;
			4'd13:HEX0 <= 7'b100_0100;
			4'd14:HEX0 <= 7'b011_0000;
			4'd15:HEX0 <= 7'b011_1000;
			default:HEX0 <= 7'b1111111;
		endcase
	end
endmodule
```

6.5.2 TESTING

The 4 rightmost Switches labelled 0-3 in Figure 37 and Figure 38 show the input, the Hex0 (highlighted shows the output which is as expected, only the bounds have been shown here but in reality a full functional test for all inputs and outputs took place ensuring that all 16 characters appear as appropriate.

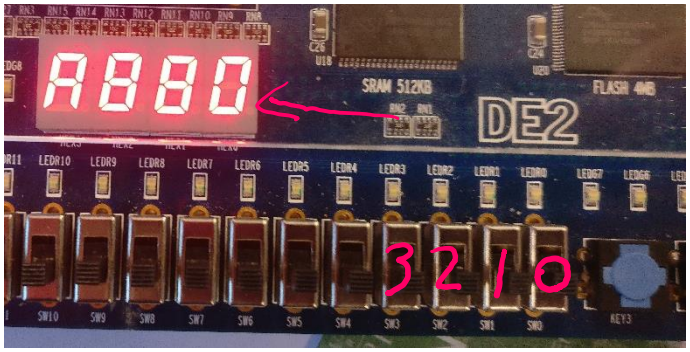


FIGURE 37, TESTING 7 SEGMENT DECODER WITH BINARY 0000, HEX 0

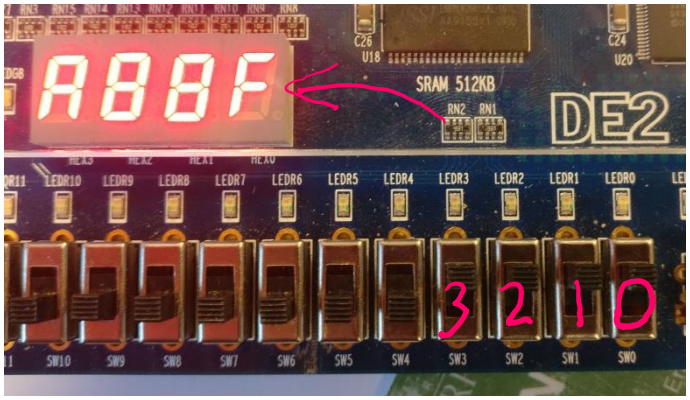


FIGURE 38, TESTING THE 7 SEGMENT DECODER WITH BINARY 1111, HEX F

This module fully functions as expected, the module outputs exactly the correct output for a full range of chosen inputs.

7 FULL SYSTEM TESTING

Following design and implementation for the full system, both transmitter and receiver have been tested together in synthesis. Due to the system taking roughly 260us to send a message, we are unable to do full simulations of the software as the end time limit is 100us. Figure 39 shows sending a full range of data from the system. Figure 40 to Figure 43 shows the bounds testing of the system.

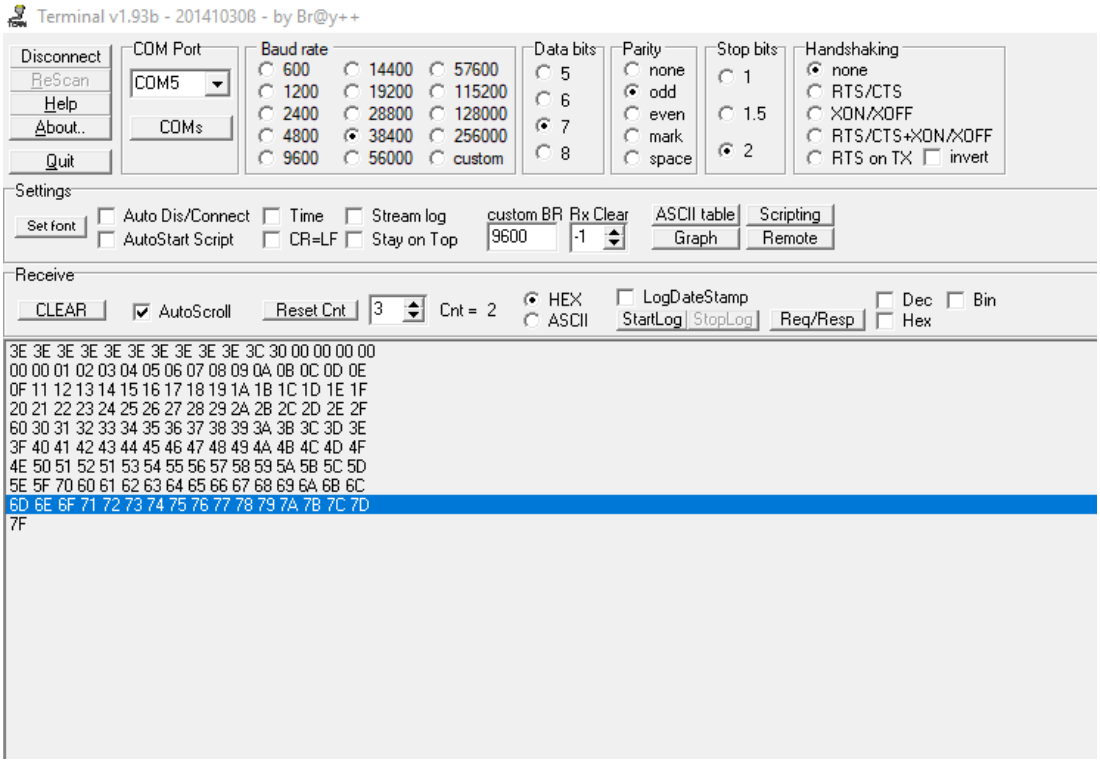


FIGURE 39, UART TRANSMITTER TESTING

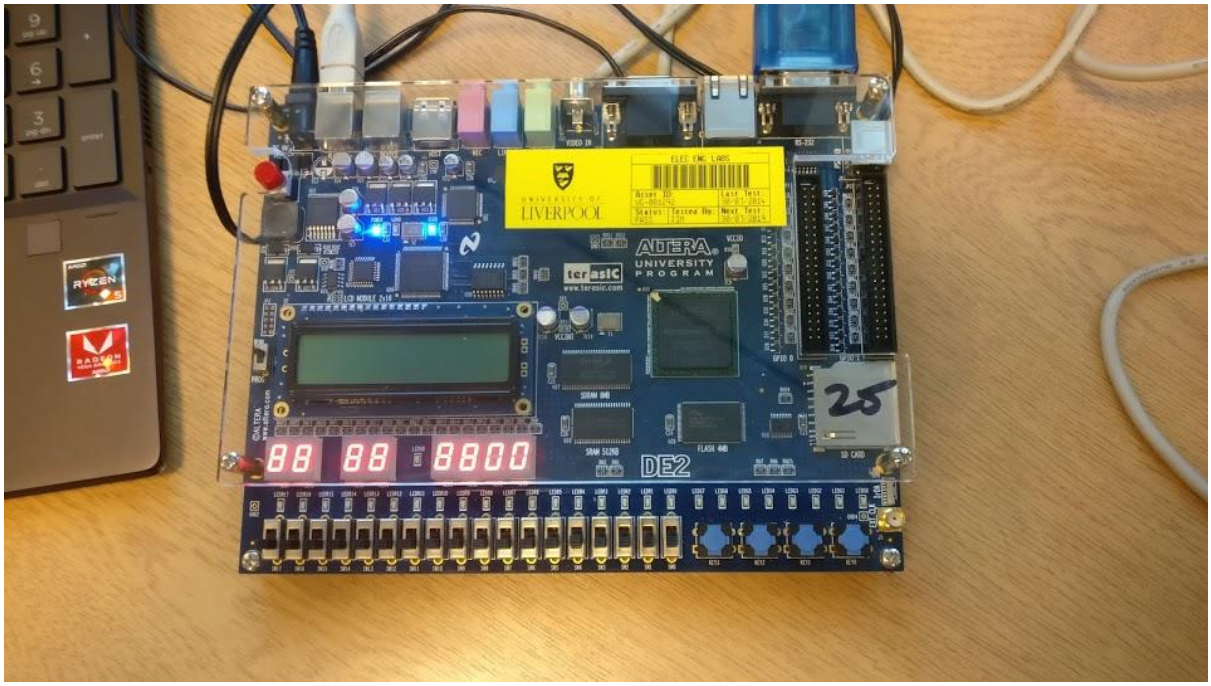


FIGURE 40, UART SYSTEM 00 BOUNDS TEST (0000000)

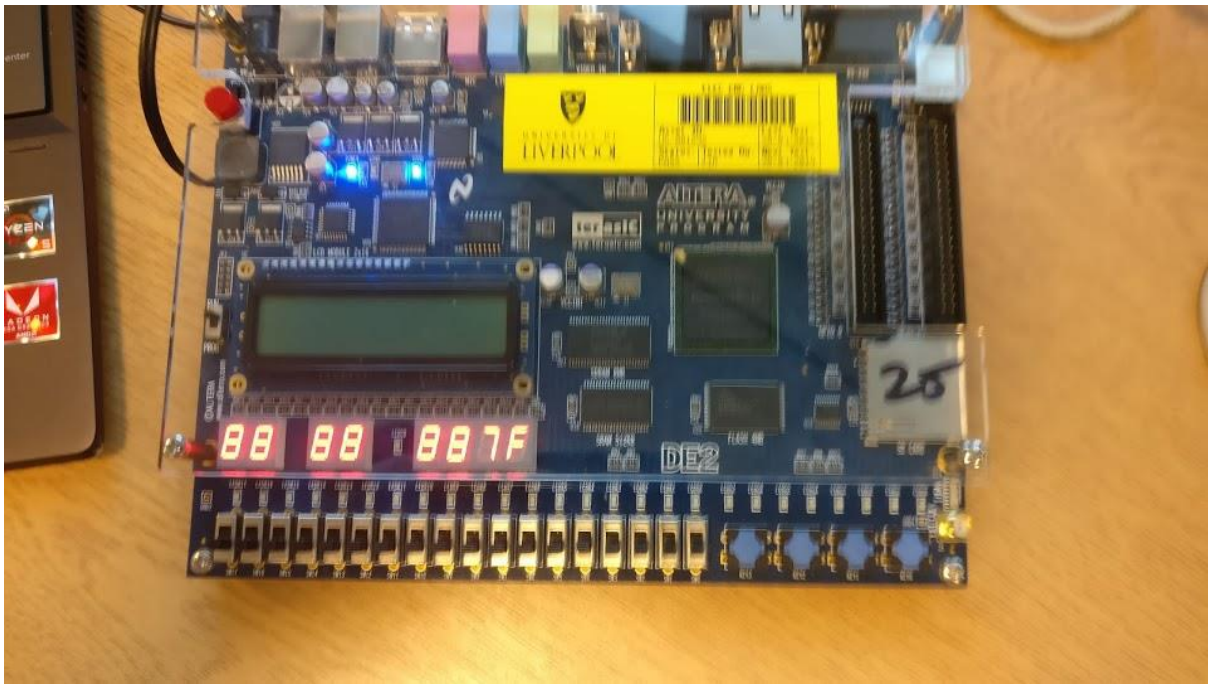


FIGURE 41, UART SYSTEM 7F BOUNDS TEST (1111111)

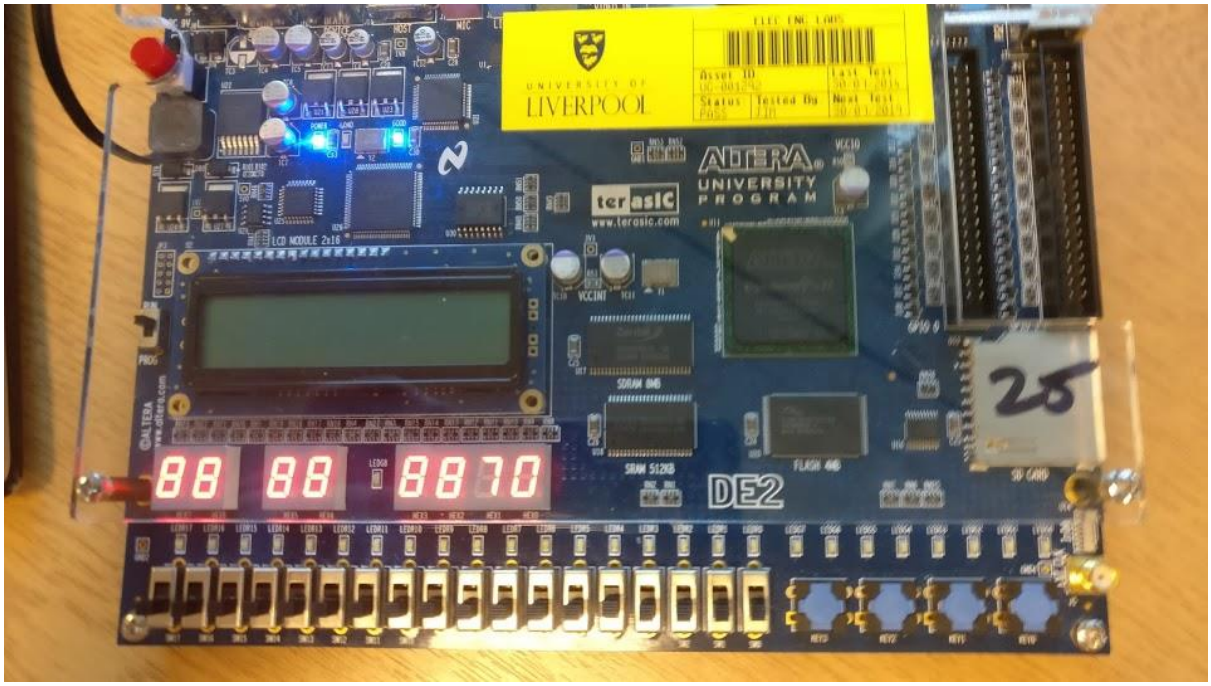


FIGURE 42, UART SYSTEM 70 BOUNDS TEST (1110000)

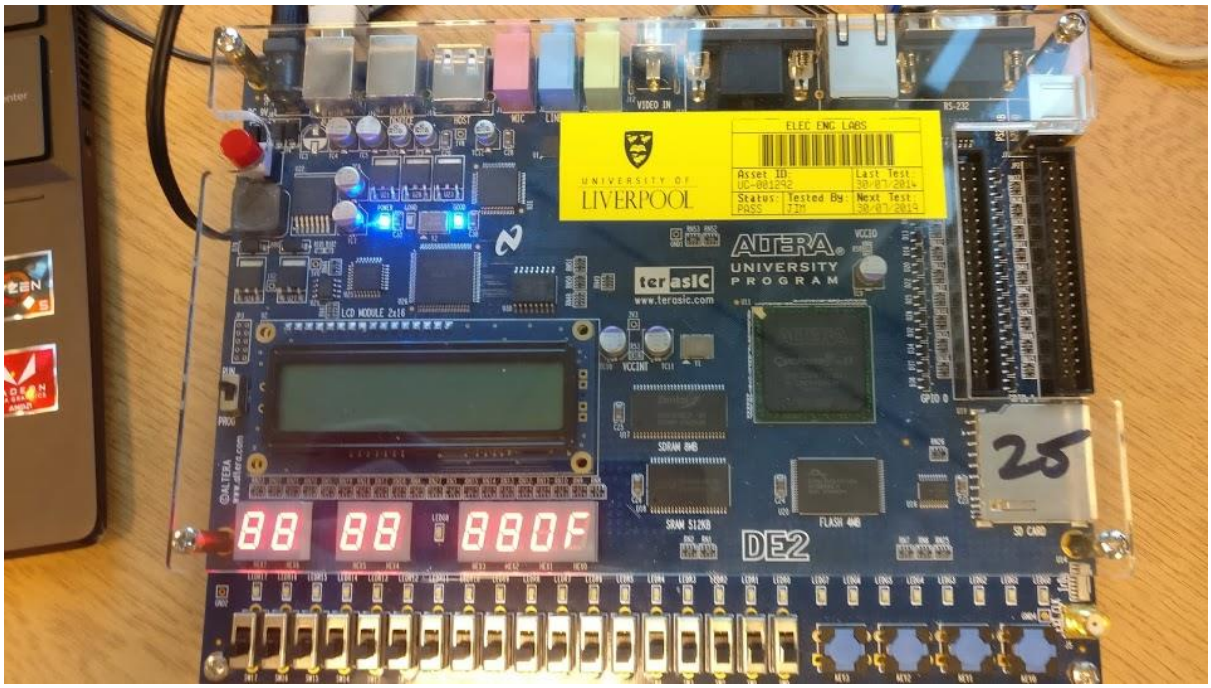


FIGURE 43, UART SYSTEM 0F BOUNDS TEST (0001111)

8 MODIFICATIONS FOR IRDA

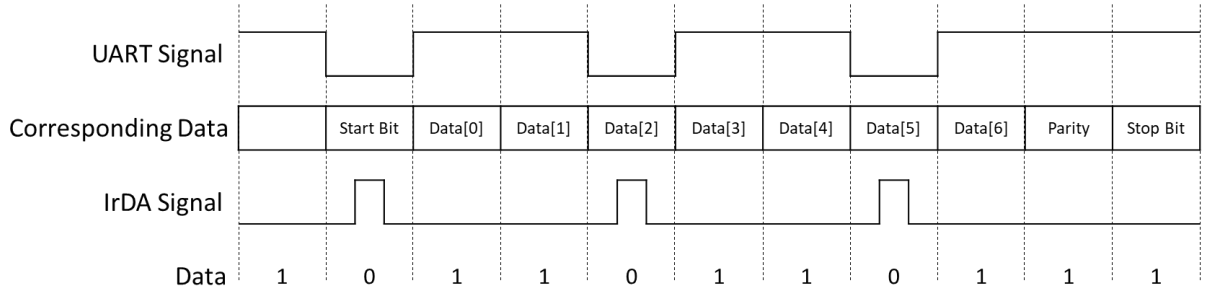


FIGURE 44, UART TO IRDA INTERFACE

Figure 44, shows how the IrDA standard extends the basic principles for a UART connection to allow for sending data over the infrared spectrum. The main difference between the UART signal and the IrDA signal is the inversion, i.e IrDA is active for 0 and not active for 1, and the length of time a signal remains high for. The IrDA standard dictates that the signal remains high for 3/16ths of the overall baud time.

There are 2 main changes that must be accounted for to enable us to transmit over IrDA, we must be able to detect the shorter length high corresponding to the data, this is also used to denote the start of the signal, we must also adjust the period of time that the signal is held high for. This can be done whilst making minimal changes to the UART functionality. The First adaption can be achieved by changing the delay before we begin sampling the data. The data sheet for our IrDA module ????? introduces a constant 2.4µs low to denote receiving a pulse. This corresponds to 120 Clock cycles of the 50Mhz clock, we must therefore sample after 60 clock cycles not the ½ baud time being sampled within the UART. The other change requires an additional counter to count for 3/16^{ths} of a baud. This should be triggered by the shift signal entering the shift register as this triggers a change in transmitted bit.

There is one new block added to the design to allow transmission of IrDA signals the ASM for this block is detailed in Figure 45. This is used in conjunction with a baud counter set to 3/16 of the baud rate this is used to generate the count pulse.

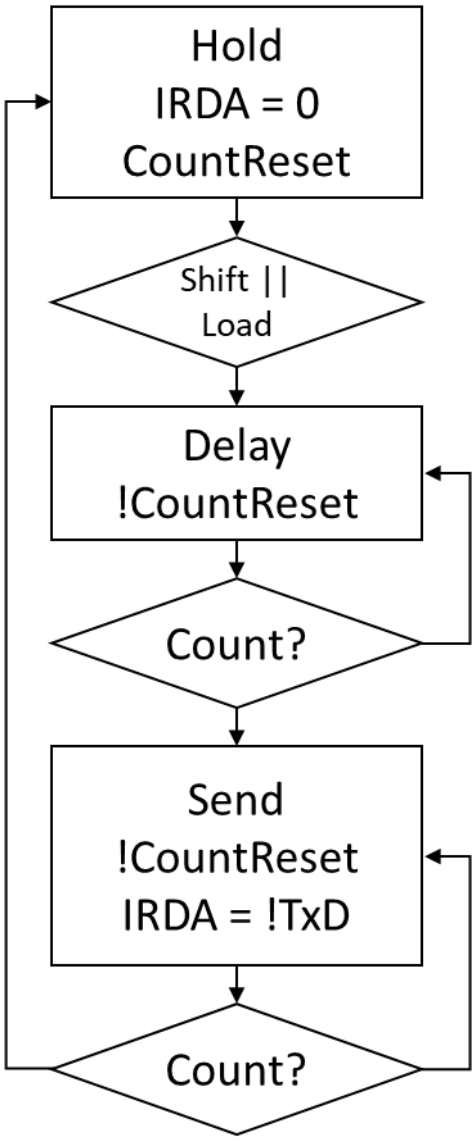


FIGURE 45, IRDA ENCODER ASM

8.1 IRDA ENCODER VERILOG

```
module irdaTxD(Load,Shift,Count,TxD,clk,reset_n,IrDA,CountReset);
      // define inputs for clock and reset
            input clk,reset_n;
            // define control inputs
            input Load, Shift,Count,TxD;
            // define control outputs
            output reg IrDA, CountReset;

            // set states as parameters
            parameter[1:0] Delay = 2'b11, Send = 2'b10, Hold = 2'b00;

            // define default states for current and next
            reg [1:0]pstate = Hold, nstate;

            // Synchronous state movement, asynchronous reset
            always@(posedge clk, negedge reset_n)
                  if (reset_n==0)
                        pstate = Hold;
                  else
```

```
                        pstate = nstate;
                // always block dictated state machine
                always@(Load, Shift,Count, pstate, TxD)
                begin
                // set defaults to prevent latches
                IrDA = 0;
                CountReset = 0;
                nstate = pstate;
                case (pstate)
                Delay: // delay ensures that the command is sent in the middle
of the message
                begin
                        CountReset = 1;
                        if (Count)
                        begin
                                nstate = Send;
                        end
                end
                Send: // send sends the command for a set length period
                begin
                        CountReset = 1;
                        IrDA = !TxD;
                        if (Count)
                                nstate = Hold;
                end
                Hold: // Wait for the remander of the bit length then delay
again
                        if (Shift || Load)
                        begin
                                nstate = Delay;
                        end
                endcase
                end
        endmodule
```

8.2 TESTING THE IRDA ENCODER

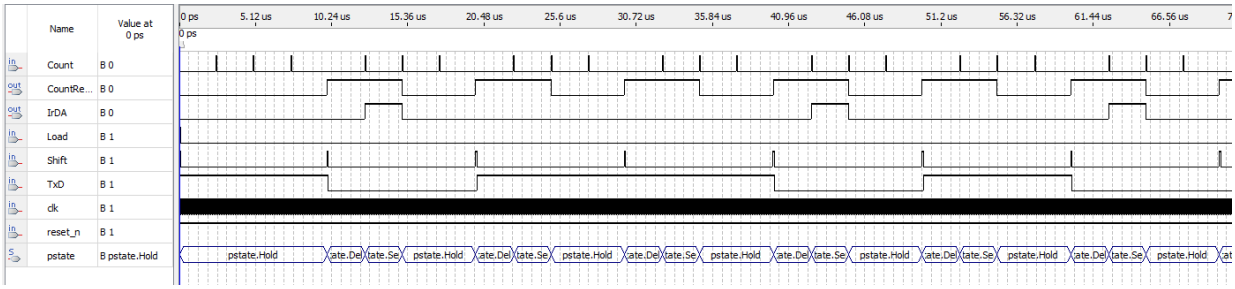


FIGURE 46, IRDA ENCODER TESTING

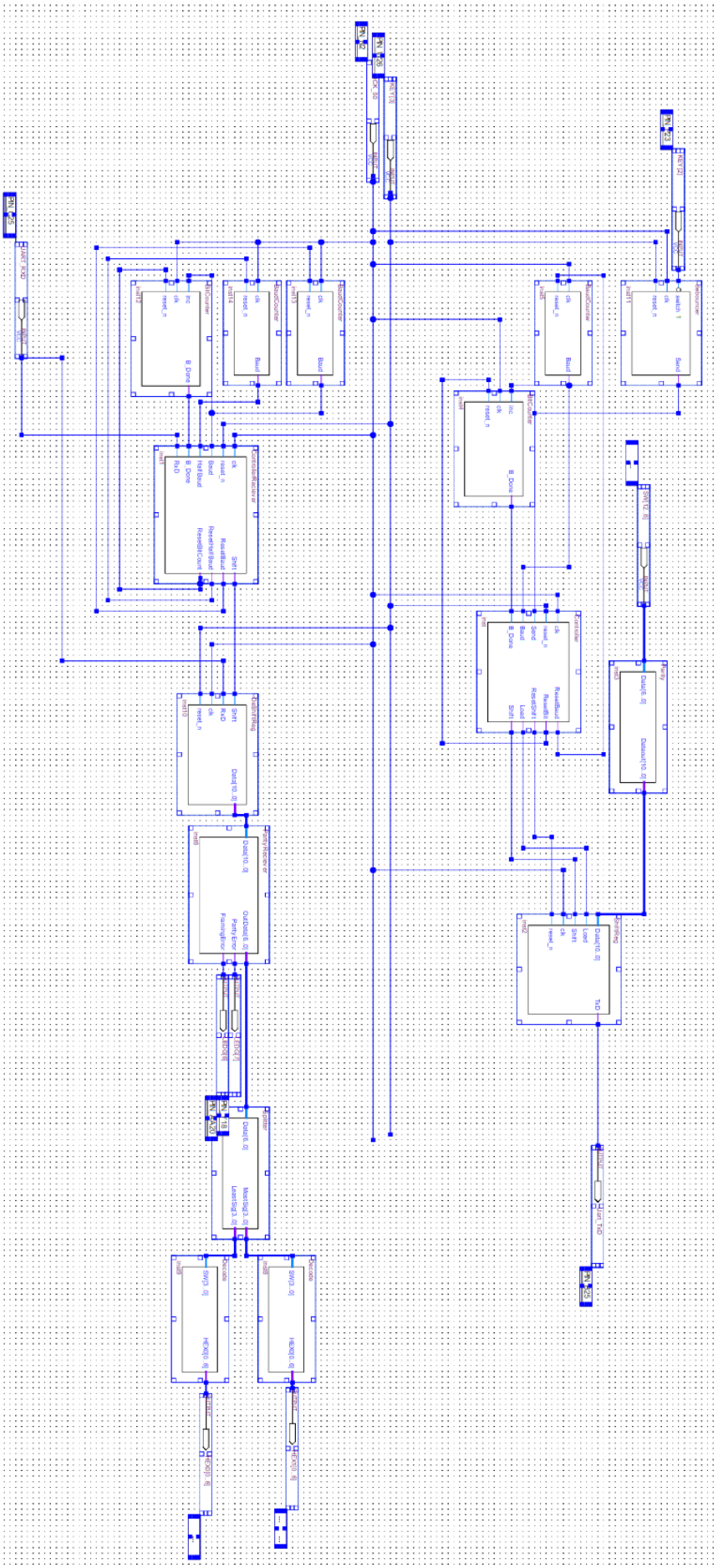
We can see above that the IRDA encoder functions and sends a IRDA signal within the 2 pulses of the counter. When the count reset is high. This means that ideally the encoder will fully work. This is not the case. The full IRDA BDF is included in 0.

9 CONCLUSION

The system in question fully functions to comply with the RS232 standard for UART communication and therefore fulfils part A of the assignment. Unfortunately, part B is not functioning for unknown reasons and then has expired on the ability to further test this functionality. The full methodology is explained above.

10 APPENDIX

10.1 FULL BDF FOR UART



10.2 FULL BDF FOR IRDA