**ECE 901 Digital Systems Prototyping**

**Mini-Project 1**

**A Special Purpose Asynchronous Receiver/Transmitter (SPART)**



**Team:**

Kushagra Garg 907 007 5420

Rohit Shukla 906 719 3145

Vignesh Chandrasekaran 906 982 9563

# Table of Contents

[Table of Contents ii](#_Toc379220837)

[Table of Figures iii](#_Toc379220838)

[1. Block Diagram 1](#_Toc379220839)

[2. Blocks 2](#_Toc379220840)

[2.1 Top level 2](#_Toc379220841)

[2.2 SPART 4](#_Toc379220842)

[2.3 Baud Rate Generator 7](#_Toc379220843)

[2.4 Receiver 10](#_Toc379220844)

[2.5 Transmitter 16](#_Toc379220845)

[2.6 Driver 20](#_Toc379220846)

[3. Problems encountered 26](#_Toc379220847)

[4. Experiment conducted 27](#_Toc379220848)

# Table of Figures

[Figure 1: Schematic of Top\_level module 1](#_Toc379220849)

[Figure 2: Schematic showing the interconnection between the Driver and SPART module 1](#_Toc379220850)

[Figure 3: Module Hierarchy 1](#_Toc379220851)

[Figure 4: Input synchronization in receiver module 10](#_Toc379220852)

[Figure 5: Parallel In Serial Out Shift register 16](#_Toc379220853)

# 1. Block Diagram

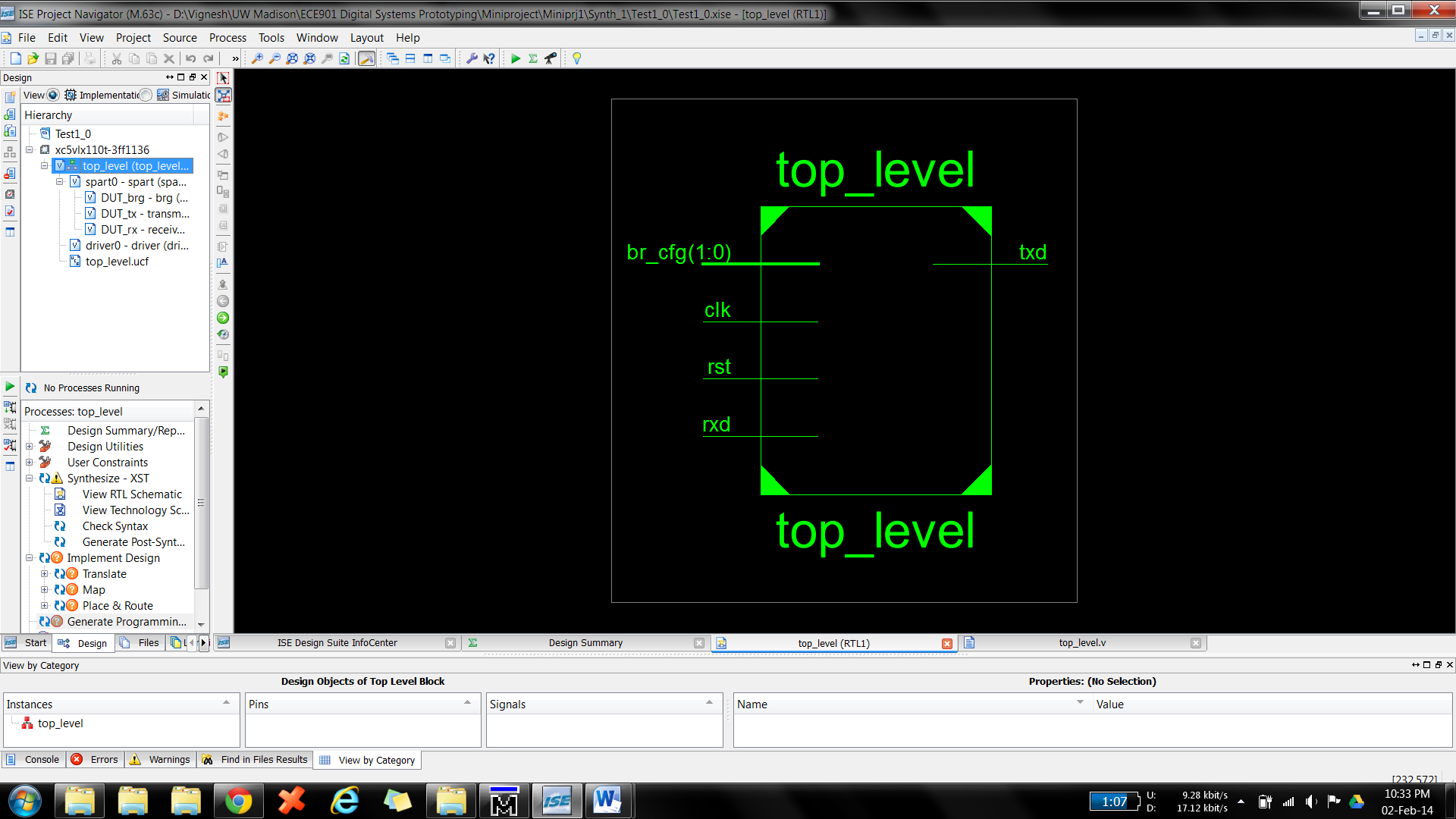


Figure 1: Schematic of Top\_level module

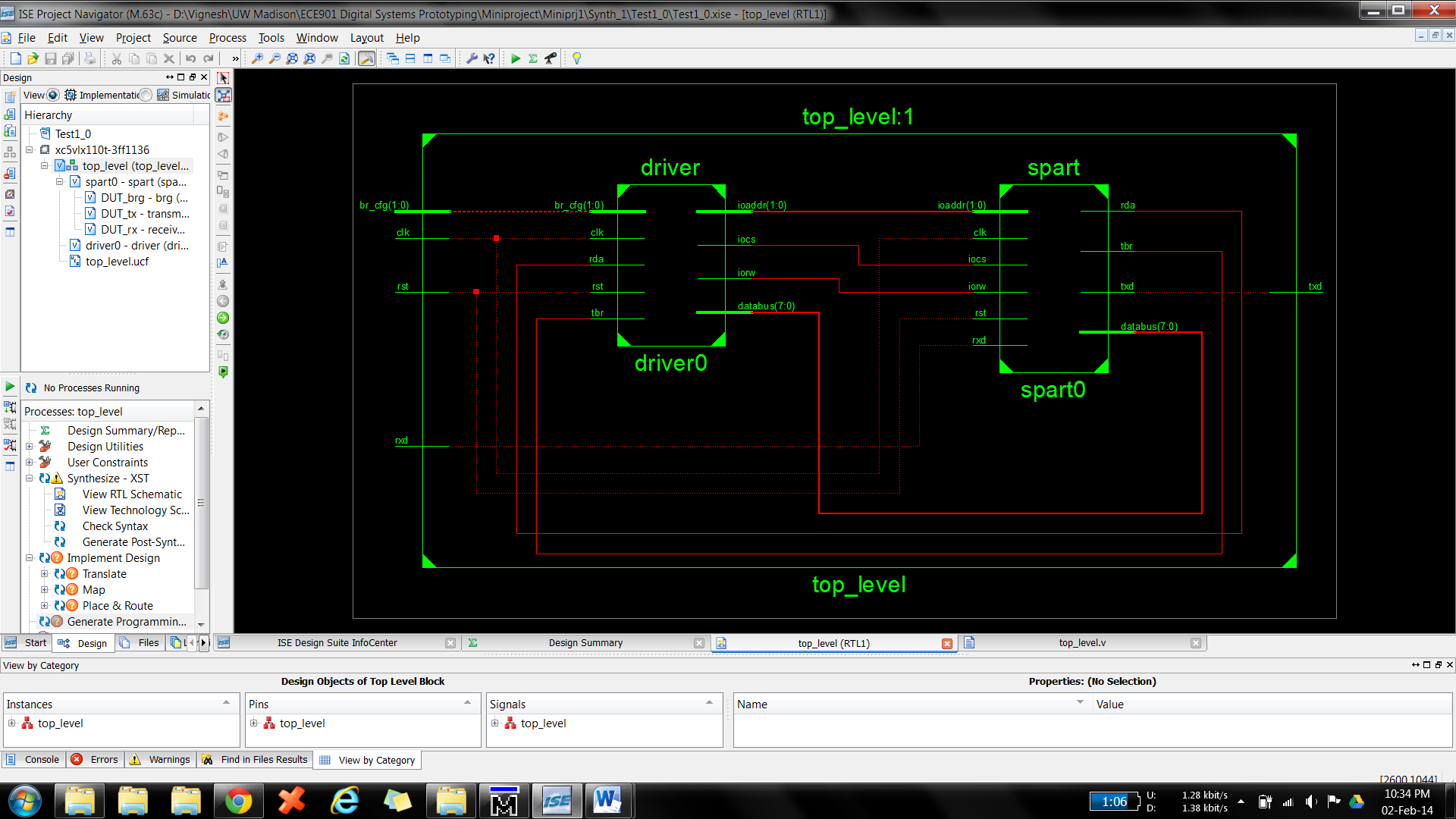


Figure 2: Schematic showing the interconnection between the Driver and SPART module

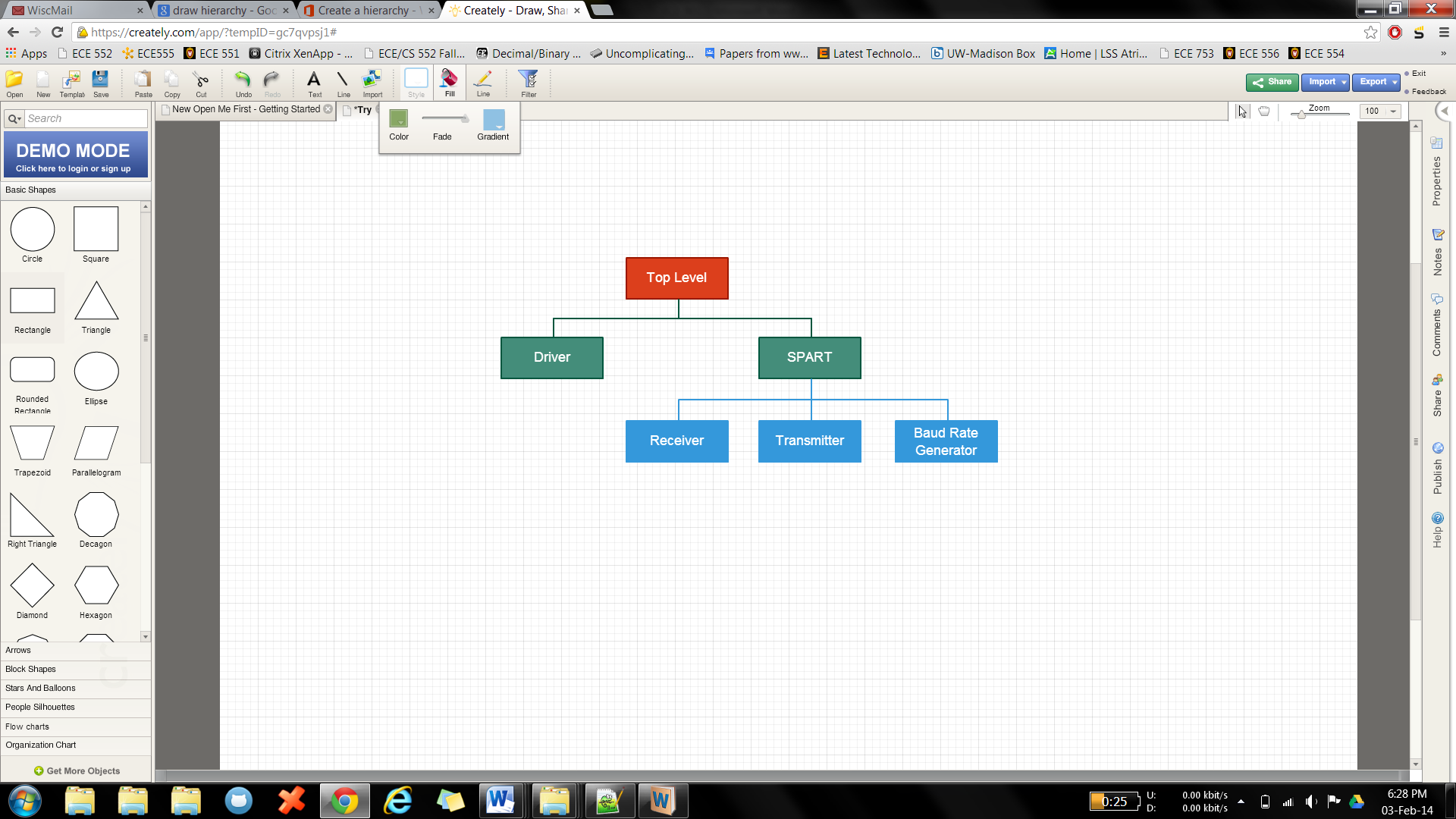


Figure : Module Hierarchy

# 2. Blocks

## 2.1 Top level

Top level module instantiates the Driver module and SPART module. It takes care of the interconnection between them

**Verilog code:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module top\_level(

input clk, // 100mhz clock

input rst, // Asynchronous reset, tied to dip switch 0

output txd, // RS232 Transmit Data

input rxd, // RS232 Receive Data

input [1:0] br\_cfg // Baud Rate Configuration, Tied to dip switches 2 and 3

);

wire iocs;

wire iorw;

wire rda;

wire tbr;

wire [1:0] ioaddr;

wire [7:0] databus;

// Instantiate your SPART here

spart spart0( .clk(clk),

.rst(rst),

.iocs(iocs),

.iorw(iorw),

.rda(rda),

.tbr(tbr),

.ioaddr(ioaddr),

.databus(databus),

.txd(txd),

.rxd(rxd)

);

// Instantiate your driver here

driver driver0( .clk(clk),

.rst(rst),

.br\_cfg(br\_cfg),

.iocs(iocs),

.iorw(iorw),

.rda(rda),

.tbr(tbr),

.ioaddr(ioaddr),

.databus(databus)

);

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## 2.2 SPART

This module instantiates the Baud Rate Generator (BRG) module, Receiver module and Transmitter module. It also deals with the bus interface that is responsible for managing the interface between the Databus, Rx, Tx, BRG and driver. This becomes very essential as the databus is bidirectional so the contention for this bus must be avoided. Depending on the IOADDR, IOR/W and IOCS signals, the module multiplexes the following signals.

|  |  |
| --- | --- |
| **IOADDR** | **SPART Register** |
| 00 | Transmit Buffer (IOR/W = 0); Receive Buffer (IOR/W = 1) |
| 01 | Status Register (IOR/W = 1) |
| 10 | DB(Low) Division Buffer (IOR/W = 0) |
| 11 | DB(High) Division Buffer (IOR/W = 0) |

**Verilog code:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module spart(

input clk,

input rst,

input iocs,

input iorw,

output rda,

output tbr,

input [1:0] ioaddr,

inout [7:0] databus,

output txd,

input rxd

);

wire [7:0] rx\_databus;

wire brg\_en, brg\_full;

reg clr\_rda;

reg rx\_tri\_en, status\_tri\_en, brg\_tri\_en;

// Instantiate sub-modules

brg DUT\_brg(.databus(databus),

.clk(clk),

.rst(rst),

.brg\_en(brg\_en),

.brg\_full(brg\_full),

.ioaddr(ioaddr)

);

transmit DUT\_tx(.databus(databus),

.clk( clk),

.rst( rst),

.tbr(tbr),

.brg\_full(brg\_full),

.txd(txd),

.ioaddr(ioaddr),

.iorw(iorw),

.iocs(iocs)

);

receiver DUT\_rx(.DATABUS(rx\_databus),

.clk(clk),

.rst(rst),

.RDA(rda),

.RX(rxd),

.brg\_en(brg\_en),

.clr\_rda(clr\_rda)

);

// Bus Interface

// Enable tri-states for Receiver and Status to drive the bus when required.

always @(\*) begin

rx\_tri\_en = 1'b0;

status\_tri\_en = 1'b0;

clr\_rda = 1'b0;

if (iocs) begin

if (ioaddr == 2'b00 && iorw == 1'b1) begin // Read command

rx\_tri\_en = 1'b1;

clr\_rda = 1'b1;

end

if (ioaddr == 2'b01 && iorw == 1'b1) // Reading status register

status\_tri\_en = 1'b1;

end

else begin

rx\_tri\_en = 1'b0;

status\_tri\_en = 1'b0;

end

end

// If command is to read RX buffer or status register, databus is driven by either rx\_databus or status register else 'ZZ

assign databus = rx\_tri\_en ? rx\_databus : (status\_tri\_en ? {6'h00,tbr,rda}: 8'hzz);

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## 2.3 Baud Rate Generator

* BRG sets the division buffer to a default value of 650 upon reset which corresponds to the counter value for 9600 Baud rate at 100MHz clock rate.
* Depending on the value of IO Address which is set using the DIP switches, the division buffer gets loaded through the Databus with the appropriate data. Since the Div\_buffer is 2 bytes long, it’s loaded one byte at a time.
* BRG issues the enable signals for the transmitter and the receiver modules. The enable signal for the transmitter module is generated with the frequency that is 2n \* Baud rate, where n=4 in this case. The enable signal for the receiver module is generated when the counter that contains the corresponding value for the given baud rate runs down to zero. Therefore the frequency of the enable signal for the receiver signal is 16x the frequency of the enable signal for the transmitter.

**Verilog code:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module brg(

input [7:0] databus,

input clk, rst,

input [1:0] ioaddr,

output brg\_en, //Signifies 1/16 of a baud .. Send to Rx

output brg\_full //Goes high every baud .. Send to Tx

);

reg [15:0] div\_buffer, div\_buffer\_next;

reg [15:0] cnt, cnt\_next;

reg [3:0] full\_cnt;

wire [3:0] full\_cnt\_next;

wire zero; // If Zero true or not

always @(posedge clk) begin

if(rst == 1'b1) begin

// Default DB to 100 MHz and 9600 baud

cnt <= 16'd650; // Gets DB

full\_cnt <= 4'hf; // Counts down from 15 to 0

div\_buffer<= 16'd650; // Rounded to 100M/9600 - 1

end

else begin

cnt<= cnt\_next;

full\_cnt <= full\_cnt\_next;

div\_buffer <= div\_buffer\_next;

end

end

always @(\*) begin

// Default condition

div\_buffer\_next = div\_buffer;

cnt\_next = cnt - 1;

// Load DivisionBuffer(high)

if(ioaddr == 2'b11)

div\_buffer\_next = {databus, div\_buffer[7:0]};

// Load DivisionBuffer(low)

if(ioaddr == 2'b10)

div\_buffer\_next = {div\_buffer[15:8], databus};

// Reset/Roll cnt to the contents of the DivisionBuffer

if(zero == 1'b1)

cnt\_next = div\_buffer;

end

assign zero = (cnt == 16'h0000) ? 1'b1 : 1'b0;

assign full\_cnt\_next = full\_cnt - zero; // Only decrements when cnt is 0

assign brg\_en = zero;

assign brg\_full = (full\_cnt == 4'h0 && brg\_en == 1'b1) ? 1'b1 : 1'b0;

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## 2.4 Receiver

Receiver module in the SPART architecture continuously receives asynchronous data from the computer at the baud rate set in computer’s HyperTerminal. Baud rate generator (BRG) outputs its signal brg\_en to the receiver module and whenever this signal is set, receiver module checks the one bit received input data. To circumvent the problem of meta-stability when receiving asynchronous data, we store the input bits in two one bit registers as shown in Fig. 1.

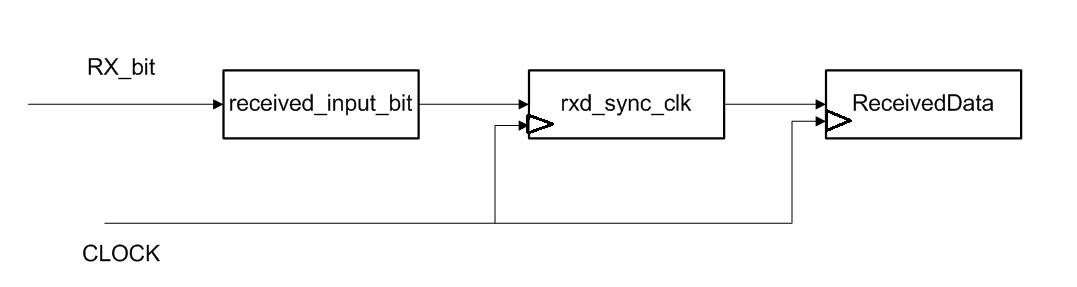


Figure : Input synchronization in receiver module

There is a four bit sample counter that counts the number of times brg\_en signal was asserted and a four bit sample accumulator counter that sums the received input bit. When the value of sample counter is 15, the receiver module checks the value MSB bit in sample accumulator counter. If the MSB value is 1, this indicates that receiver has received more number of bit ones than bit zeros. Thus receiver should interpret this as input bit 1. If the MSB value of sample accumulator is 0, the receiver will interpret that it has received bit 0. This implementation makes sure we sample the input data 16 times and do not read a wrong data that might occur due to spikes.

If the receiver sees bit zero for the first time, this indicates the start bit of input stream. A separate counter starts incrementing from 0 and goes on till 9, to ensure that all the 8 input bits have been received. Counter value is set to 1 when it reads the start bit. The received input bits that appear after the start bit are stored in a shift register named ReceivedData. After all of the 8 data bits have been stored, the value of counter is reset to zero and RDA signal is asserted to tell the processor that data is ready to be transferred from SPART. The RDA is reset when the top level SPART module sends a clr\_RDA signal to the receiver module. The receiver will repeat the process of checking the start bit again.

**Verilog code:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module receiver(

input RX,

output [7:0] DATABUS,

output reg RDA,

input brg\_en,

input clk,

input rst,

input clr\_rda

);

// Counter increments from 0 to 9. After its value becomes 9, this means that we have received all the serial data //from computer. Now reset it to 0.

reg [3:0] counter, counter\_next, sample\_count, sample\_count\_next, sample\_accum, sample\_accum\_next;

// A one bit state to keep in track whether we have started receiving the serial data.

reg [1:0] state;

reg [1:0] next\_state;

// Store the one bit input data in the register first.

reg received\_input\_bit;

reg rxd\_sync\_clk;

reg RDA\_next;

// Sore the received bits in the ReceivedData register. When all of the input serial data has been received

// send it to the processor through DATABUS.

reg [7:0] ReceivedData, ReceivedData\_next;

assign DATABUS = ReceivedData;

// STATES

localparam NOT\_RECEIVING\_DATA = 0,

RECEIVE\_DATA = 1,

SET\_RDA = 2;

// INPUT BITS STATE

localparam START\_BIT = 0,

COMPLETED\_RECEIVING\_DATA = 9;

always @(posedge clk) begin

if(rst == 1'b1) begin

ReceivedData <= 8'h0;

state <= NOT\_RECEIVING\_DATA;

counter <= 4'h0;

sample\_count <= 4'h0;

sample\_accum <= 4'h0;

RDA <= 1'b0;

received\_input\_bit <= 1'b1;

rxd\_sync\_clk <= 1'b1;

end

else begin

ReceivedData <= ReceivedData\_next;

state <= next\_state;

counter <= counter\_next;

sample\_count <= sample\_count\_next;

sample\_accum <= sample\_accum\_next;

RDA <= RDA\_next;

//synchronize the RX line with CLK

received\_input\_bit <= RX;

rxd\_sync\_clk <= received\_input\_bit;

end

end

always @ (\*) begin

sample\_accum\_next = sample\_accum;

ReceivedData\_next = ReceivedData;

counter\_next = counter;

RDA\_next = RDA;

sample\_count\_next = sample\_count;

case (state)

NOT\_RECEIVING\_DATA: begin

next\_state = NOT\_RECEIVING\_DATA;

if (clr\_rda)

RDA\_next = 1'b0;

if (brg\_en) begin

//Sample the bit. Search for START\_BIT

if (sample\_count == 4'h0) begin

if (rxd\_sync\_clk == 1'b0) begin

// Finding the START\_BIT. Start incrementing the accumulator.

sample\_count\_next = 4'h1;

next\_state = NOT\_RECEIVING\_DATA;

sample\_accum\_next = 4'h0;

end

else begin

sample\_count\_next = 4'h0;

next\_state = NOT\_RECEIVING\_DATA;

sample\_accum\_next = 4'h0;

end

end

// Sampled 16 times. Pick the correct bit.

else if (sample\_count == 4'hF) begin

sample\_accum\_next = 4'h0;

sample\_count\_next = 4'h0;

if (sample\_accum[3] == 1'b0) begin

//start bit found, begin receiving other bits

next\_state= RECEIVE\_DATA;

ReceivedData\_next = 8'h00;

counter\_next = 4'h0;

end

else

next\_state = NOT\_RECEIVING\_DATA;

end

// Continue sampling 16 times.

else begin

// Keep count of the number of samples per brg\_full signal

sample\_count\_next = sample\_count + 1;

// Add the RX and keep accumulating.

sample\_accum\_next = sample\_accum + rxd\_sync\_clk;

next\_state = NOT\_RECEIVING\_DATA;

end

end

end

RECEIVE\_DATA: begin

RDA\_next = 1'b0; //byte not ready

if (brg\_en) begin

// Add all the received bits and pick the MSB as the denoted RX bit.

sample\_accum\_next = sample\_accum + rxd\_sync\_clk;

sample\_count\_next = sample\_count + 1;

end

if(brg\_en && sample\_count == 4'hF) begin

// received bit is sampled 16 times

ReceivedData\_next = {sample\_accum[3],ReceivedData[7:1]};

sample\_accum\_next = 4'h0;

counter\_next = counter + 1;

end

if (counter == 4'd8)begin

//we have all of our bits for this transmission

next\_state = NOT\_RECEIVING\_DATA;

RDA\_next = 1'b1;

sample\_accum\_next = 4'h0;

sample\_count\_next = 4'h0;

end

else begin

//keep sampling and shifting bits in

next\_state = RECEIVE\_DATA;

end

end//RECIEVING

endcase

end

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## 2.5 Transmitter

Tx block transmits the 8 bit data stored in the write buffer in serial mode. Enable signal from baud rate generator block is used in making sure serial transfer happens at the correct baud rate. From the design point of view, transmitter has a shift register block and write buffer block, instead of implementing them in separate blocks, it was implemented as Parallel In Serial Out (PISO) shift register as shown below.

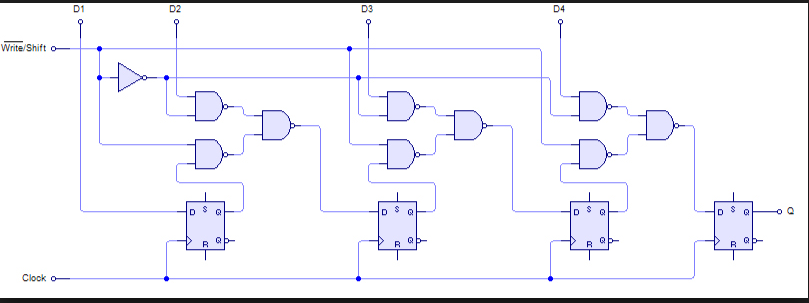


Figure : Parallel In Serial Out Shift register

Transmitter deals with the following operations:

* Initially, write buffer is empty and shift registers are all parallel loaded with 1's so that shifter transmit stop bit (1’b1).
* When Write occurs, TBR is set, indicating to the processor that write buffer is full. Also, shift register[0] bit is set to 0 on BRG enable so that it transmits the start bit.
* Counter keeps track of serial transmission of each bit. Once all bits are transferred, shift registers are set to 1's to transfer stop bits. TBR is reset to indicate to the processor that Write buffer is empty and ready to receive data.

**Verilog code:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module transmit(

clk,

rst,

brg\_full,

//baud rate generator en

iorw, //constitutes wr\_en

iocs, //constitutes wr\_en

databus,

ioaddr,

tbr,

txd

);

//Input ports

input clk;

input rst;

input brg\_full;

input iorw;

input iocs;

input [1:0] ioaddr;

input [7:0] databus;

// Output ports

output tbr;

output txd;

reg [8:0] piso; // parallel in serial out shifter

reg [3:0] count;

reg buffer\_full;

assign tbr = ~buffer\_full;

assign txd = piso[0]; // Last bit of shifter sent out

assign cnt\_flag = (count == 10);

always @ (posedge clk)

begin

if(rst) begin

piso <= 9'h1FF; // Should send out STOP bit

end

else if (iocs & ~iorw & (ioaddr == 2'd0)) begin

piso <= {databus[7:0],1'b1};

end

else if (buffer\_full && brg\_full && ~cnt\_flag ) begin

if (count == 0)

piso[0] <= 1'b0; // Start bit

else

piso <= {1'b1, piso[8:1]}; // Shift

end

else if (cnt\_flag & brg\_full) begin

piso <= 9'h1FF;

end

end

//Different block for buffer\_full

always @ (posedge clk)

begin

if (rst)

buffer\_full <= 1'b0;

else if (cnt\_flag & brg\_full)

buffer\_full <= 1'b0;

else if (iocs & ~iorw & (ioaddr == 2'b0))

buffer\_full <= 1'b1;

end

// Different block for count

always @ (posedge clk)

begin

if (rst)

count <= 0;

else if (cnt\_flag & brg\_full) // When all bits are sent, reset count to 0

count <= 0;

else if (brg\_full & buffer\_full) // On each brg en, increment count by one

count <= count + 1;

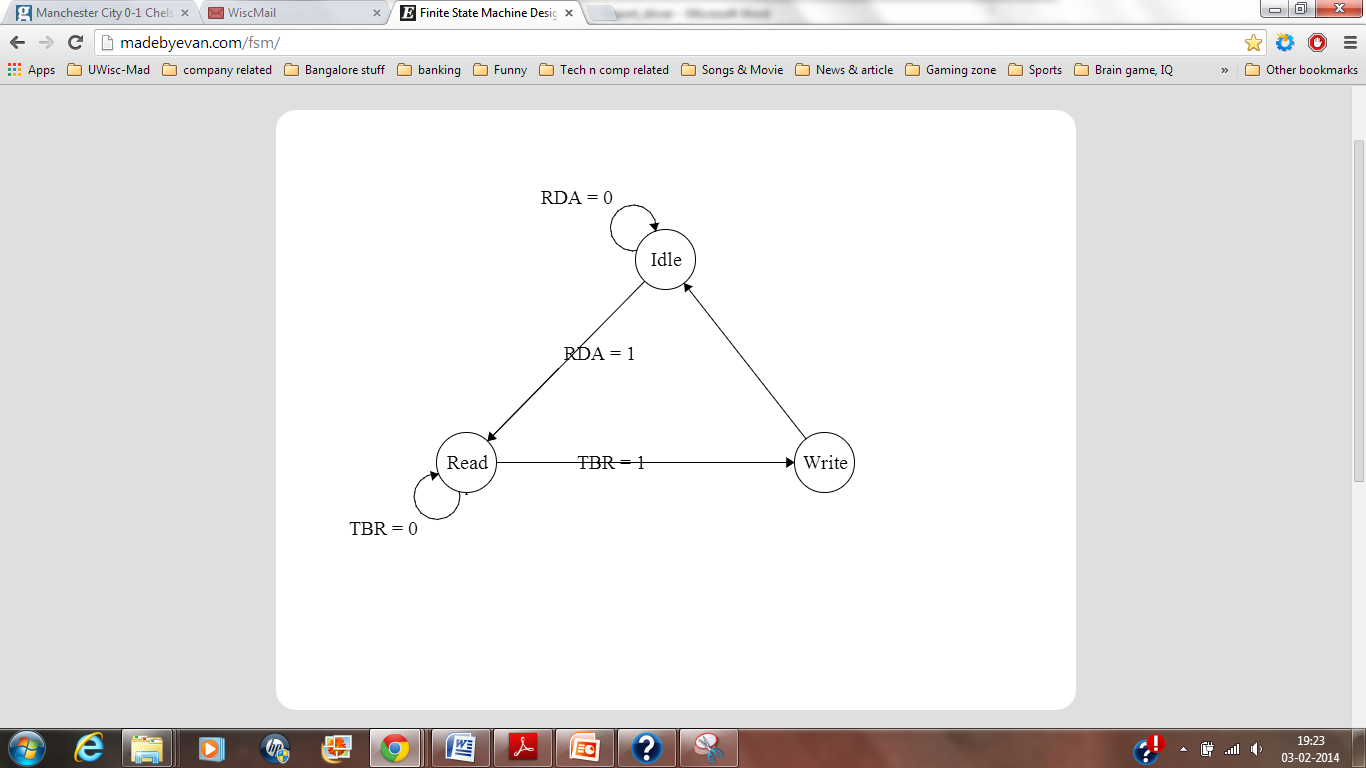
end

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## 2.6 Driver

Driver block implements a simple finite state machine. State machine makes sure to read data on RXD from keyboard and transmit (echos) back on the TXD. Diagram below shows the state transition diagram for driver block.



* Initially, driver is in IDLE state. Upon reset, it issues two write commands to write in division buffers low & high according to two pins which determines baud rate.
* After that, it waits for RDA to go high i.e. once read data in available in receive buffer. Once RDA = 1, it changes state to READ and issues a read command.
* Now the state machine waits for TBR = 1 i.e. once transmit buffer is ready to receive data. Once TBR = 1, it changes state to WRITE and issues a write command by sending the received data to write buffer of transmit block.
* After that state goes back to IDLE again, waiting for another read on RXD.

**Verilog code:**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

module driver(

input clk,

input rst,

input [1:0] br\_cfg,

output reg iocs,

output reg iorw,

input rda,

input tbr,

output reg [1:0] ioaddr,

inout [7:0] databus

);

parameter IDLE = 2'b00;

parameter WRITE = 2'b01;

parameter READ = 2'b10;

// Baud rate configurations

parameter BRG\_CGF\_325 = 2'b00;

parameter BRG\_CGF\_162 = 2'b01;

parameter BRG\_CGF\_81 = 2'b10;

parameter BRG\_CGF\_40 = 2'b11;

reg [1:0] state; // 00 implies idle, 01 implies write state & 10 implies read state

reg [1:0] next\_state;

reg [1:0] ready\_rw;

reg [7:0] databus\_drive; // Data which will drive the bus

reg [7:0] databus\_input; // Data which will store the input while reading

reg [7:0] div\_low;

reg [7:0] div\_high;

// tri state logic, databus driven only when write command is issued otherwise 'Z' is driven

assign databus = (iorw == 0 & iocs == 1 ) ? databus\_drive : 8'hzz;

always @ (posedge clk) begin

if(rst)

databus\_input <= 8'h00;

else if (iorw == 1 & iocs == 1) // Read command

databus\_input <= databus;

end

// Assign div\_low and div\_high on based on br\_cfg inputs

always @ (\*) begin

case (br\_cfg)

BRG\_CGF\_325: begin

div\_low <= 8'h16;

div\_high <= 8'h05;

end

BRG\_CGF\_162: begin

div\_low <= 8'h8B;

div\_high <= 8'h02;

end

BRG\_CGF\_81: begin

div\_low <= 8'h46;

div\_high <= 8'h01;

end

BRG\_CGF\_40: begin

div\_low <= 8'hA3;

div\_high <= 8'h00;

end

endcase

end

always @ (posedge clk)

begin

if(rst) begin

state <= IDLE;

end

else

state <= next\_state;

end

// State machine transition logic

always @(state or tbr or rda or ready\_rw)

begin

case(state)

// If the read data is available, change state from IDLE to read

IDLE : if ( (rda == 1) & (ready\_rw == 2) )

next\_state = READ;

else

next\_state = IDLE;

// After write, change state back to idle

WRITE : next\_state = IDLE;

// After Read, change state to write so that data which read is send for transmit

READ : if ((tbr == 1) & (ready\_rw == 2 ))

next\_state = WRITE;

else

next\_state = READ;

endcase

end

always @ (posedge clk)

begin

if (rst) begin

iocs <=0;

iorw <= 1;

ioaddr <= 2'b00;

databus\_drive <= 8'h00;

ready\_rw <= 2'b00;

end

// Upon reset program div buf

else if ( ready\_rw == 0) begin

ioaddr <= 2'b10; // Div buffer low

iocs <= 1;

iorw <= 0;

databus\_drive <= div\_low;

ready\_rw <= ready\_rw + 1;

end

else if ( ready\_rw == 1) begin

ioaddr <= 2'b11; // Div buffer low

iocs <= 1;

iorw <= 0;

databus\_drive <= div\_high;

ready\_rw <= ready\_rw + 1;

end

// condition executed when both the div\_buffer writes are done, outputs change based on current state

else if ( ready\_rw == 2 ) begin

ioaddr <= 2'b00; // To prevent writing to div buffer again

case(state)

IDLE : begin

iocs <=0;

iorw <= 1;

end

WRITE : begin // Write command

iocs <= 1;

iorw <= 0;

databus\_drive <= databus; // Generate random value may be later

ioaddr <= 2'b00;

end

READ : begin // Read Command

iocs <= 1;

iorw <= 1;

ioaddr <= 2'b00;

end

endcase

end

end

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 3. Problems encountered

* Though the overall problem specification of the SPART implementation was straightforward, attention had to be paid to the specifics and corner cases. For example, Transmitter initially worked for a single write but it failed for consecutive writes. The problem was resolved by changing the logic for resetting the TBR signal.
* The receiver module failed to sample the received data 16 times and take the average/majority bit as the received bit but instead it sampled just once. This was later rectified.

# 4. Experiment conducted

Driver module was implemented to echo back the inputs that were fed in through the keyboard. Data was fed through the RxD port using Putty as the interface and the same data was echoed back as the output on Putty. The result was verified on all four baud rates by changing the design parameters using the DIP switches on the development kit as well as changing the baud rate on the putty interface.

Inputs were verified in two scenarios:

* Sending the same character multiple times.
* Sending several characters.