ECE 901 Digital Systems Prototyping

Mini-Project 1

A Special Purpose Asynchronous Receiver/Transmitter (SPART)

THE UNIVERSITY of WISCONSIN MADISON

Team Name: Untitled Design

Kushagra Garg	907 007 5420
Rohit Shukla	906 719 3145
Vignesh Chandrasekaran	906 982 9563

Table of Contents

Table of Figures

1. Block Diagram

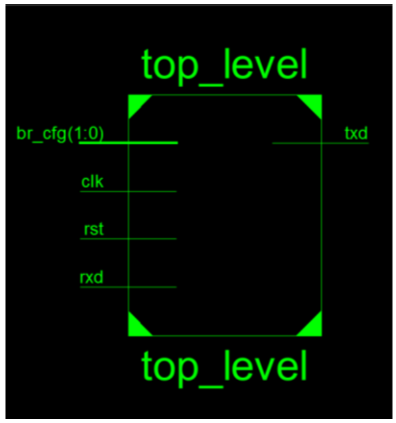


Figure 1: Schematic of Top_level module

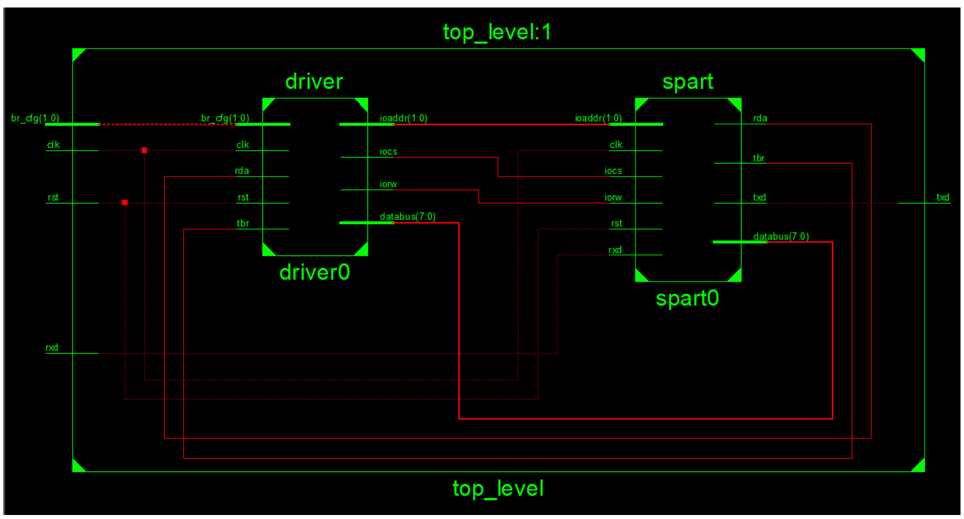


Figure 2: Schematic showing the interconnection between the Driver and SPART module

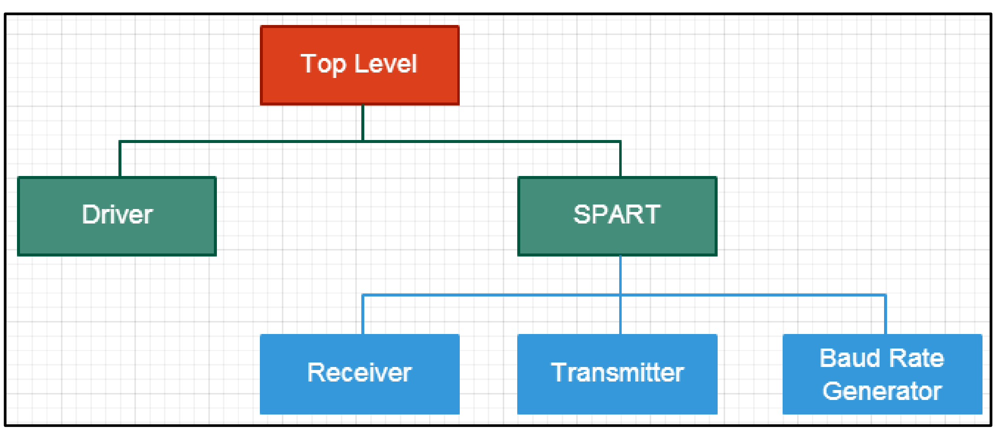


Figure 3: Module Hierarchy

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 2^n * 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.

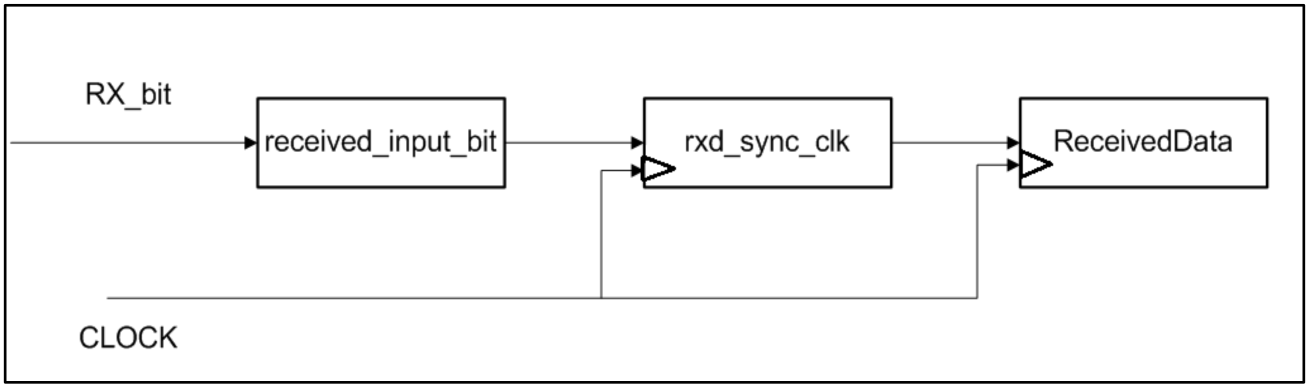


Figure 4: Input synchronization in receiver module

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.

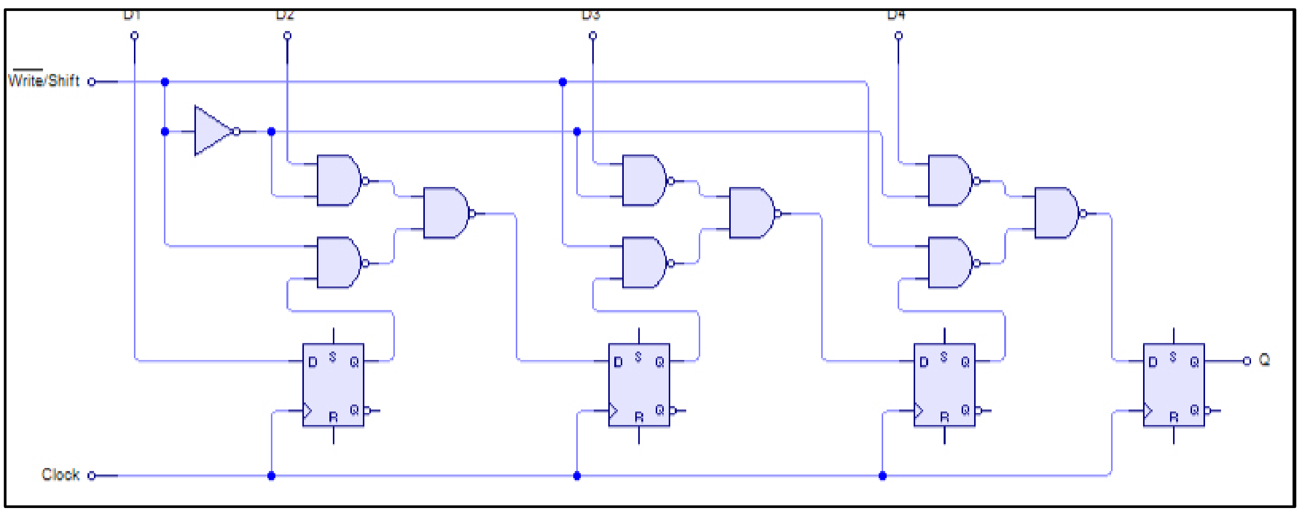


Figure 5: Parallel In Serial Out Shift register

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:

****

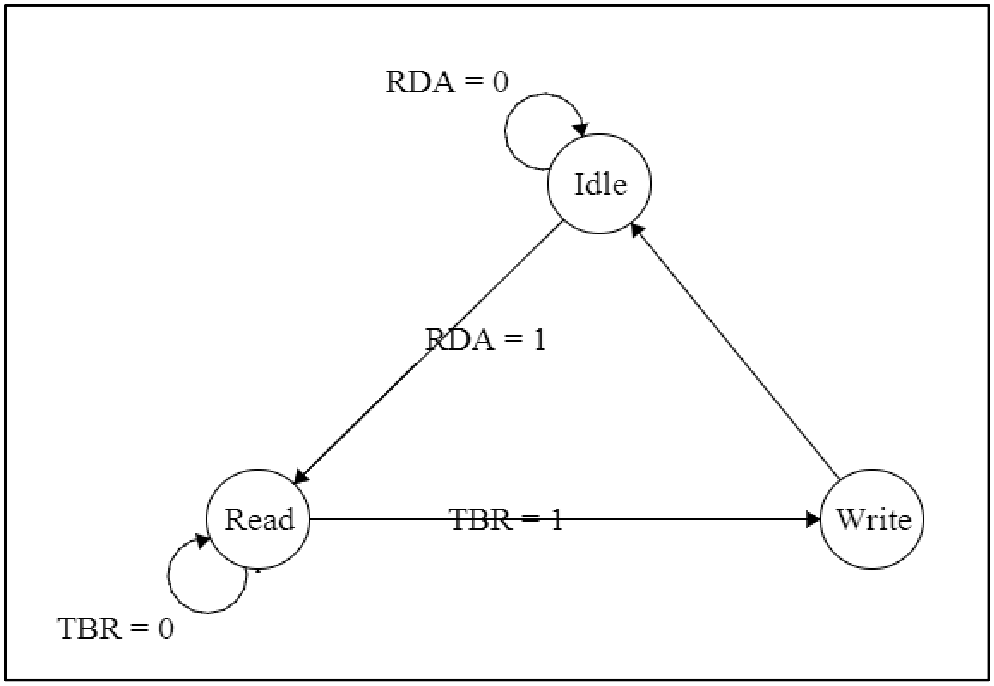
```
module transmit(
clk,
rst,
brg_full,
//baud rate generator en
iorw, //constitutes wr_en
iocs, //constitutes wr_en
databus,
ioaddr,
tbr,
txd
);

//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.