SPART

A Special Purpose Asynchronous Receiver/Transmitter

Introduction

In this miniproject you are to implement a Special Purpose Asynchronous Receiver/Transmitter (SPART). The SPART can be integrated into the processor of your final project to serve as the serial I/O interface between the processor and serial I/O port on the lab workstations. Using the Hypertermial Accessory program, this will permit you to input characters from the keyboard and to output characters to the screen on the lab workstations.

The objectives of this *miniproject* are to:

- Familiarize you with design in the ECE 554 Virtex-5 Board environment
- Practice the use of an HDL in design
- Generate a useful design for your final project
- Acquire an initial experience in efficiently and effectively performing a design as a team

SPART Design

SPART Functional Description

This section specifies the subsystem to be designed. In order to classify the description, some terminology is necessary. The term output or write are used when the processor is sending information to the SPART. The term *transmit* is used when the SPART is transmitting data to the serial I/O port on the workstation. Conversely, the terms *input* or *read* are used when the processor is retrieving information from the SPART. Finally, the term *receive* is used when the SPART is receiving data from the serial I/O port on the workstation.

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
11	DB(High) Division Buffer

Table 1: Address Mappings

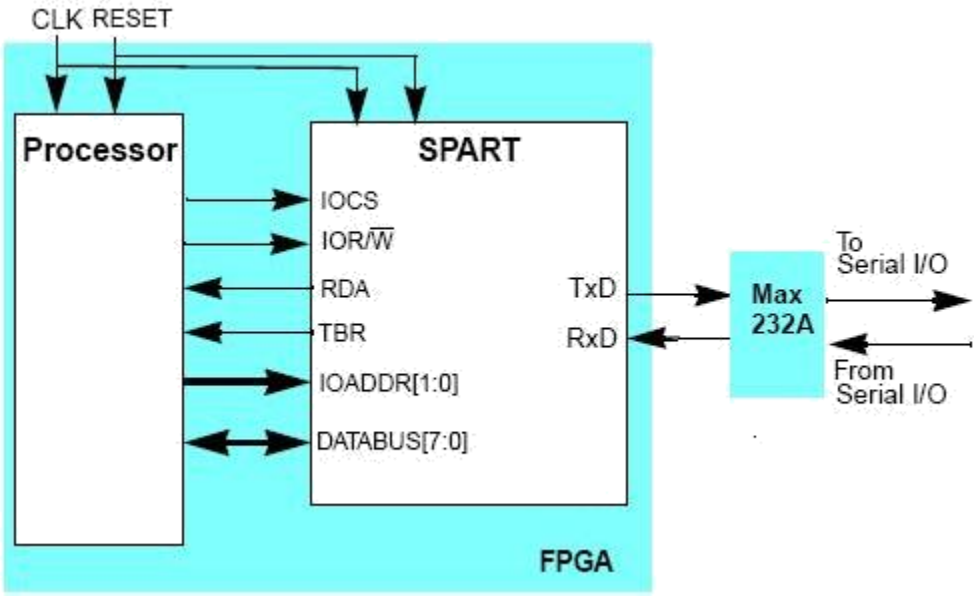


Figure 1: SPART Environment

A top level diagram of the SPART and its environment is shown in Figure 1. The FPGA interfaces with a chip on the board which generates appropriate voltage levels for the RS232 interface. The TxD pin transmits serial data from the FPGA and RxD receives serial data.

The SPART and Processor driver share many interconnections in order to control the reception and transmission of data. On the left, the SPART interfaces to an 8-bit, 3-state bidirectional bus, DATABUS[7:0]. This bus is used to transfer data and control information between the Processor and the SPART. In addition, there is a 2-bit address bus, IOADDR[1:0] which is used to select the particular register that interacts with the DATABUS during an I/O operation. The IOR/W signal determines the direction of data transfer between the Processor and SPART. For a Read (IOR/W=1), data is transferred from the SPART to the Processor and for a Write (IOR/W=0), data is transferred from the processor to the SPART. IOCS and IOR/W are crucial signals in properly controlling the three-state buffer on DATABUS within the SPART. Receive Data Available (RDA), is a status signal which indicates that a byte of data has been received and is ready to be read from the SPART to the Processor. When the read operation is performed, RDA is reset. Transmit Buffer Ready (TBR) is a status signal which indicates that the transmit buffer in the SPART is ready to accept a byte for transmission. When a write operation is performed and the SPART is not ready for more transmission data, TBR is reset. The SPART is fully synchronous with the clock signal CLK; this implies that transfers between the Processor and SPART can be controlled by applying IOCS, IOR/W, IOADDR, and DATABUS (in the case of a write operation) for a single clock cycle and capturing the transferred data on the next positive clock edge. The received data on RxD, however,

is asynchronous with respect to CLK. Also, the serial I/O port on the workstation which receives the transmitted data from TxD has no access to CLK. This interface thus constitutes the "A" for "Asynchronous" in SPART and requires an understanding of RS-232 signal timing and (re)synchronization.

SPART Structure

A block diagram of the SPART is given in Figure 2. Each subsystem is briefly described in this section.

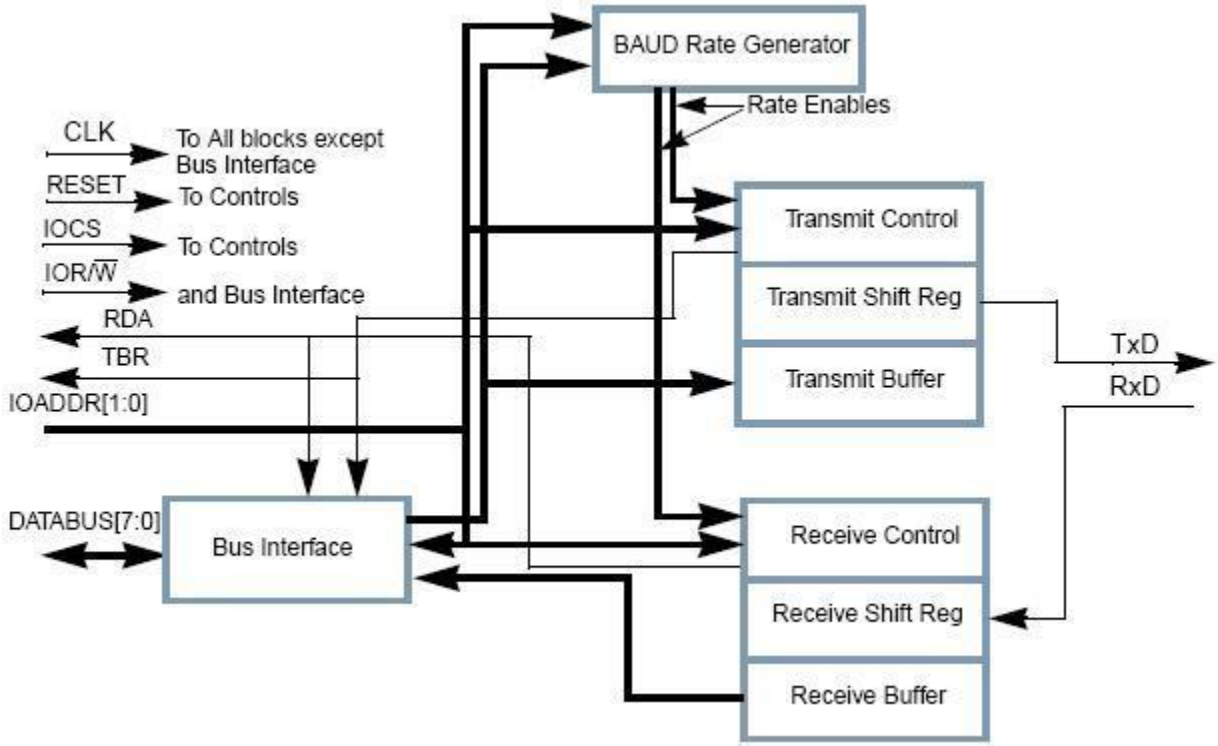


Figure 2: SPART Block Diagram

Bus Interface

The Bus Interface contains the 3-state drives which attach the SPART to the DATABUS. In addition, it contains the multiplexer which selects the Receive Buffer or the Status Register. The Status Register consists of RDA and TBR in positions 0 and 1, respectively. The Status Register is not actually a register, but just connections from RDA and TBR which are stored at their respective sources. The remaining six bits connected to the multiplexer for the Status Register are zeros. Note that RDA and TBR are provided both as direct signals to the Processor and as part of the Status Register content accessible by the Processor via the DATABUS. If interrupt-based I/O is used for the SPART, then the

direct signals can be used as inputs to the interrupt system. If program-based I/O is used, then the Status Register content (RDA, TBR) can be accessed by the program using an I/O read operation on the Status Register to determine if an I/O data operation is needed. In either case, RDA and TBR can be used as part of a "handshake" between the processor and the SPART during I/O transactions.

In addition to the above datapath constructs, the Bus Interface also contains combinational control logic for the above. In particular, it uses IOCS and IOR/W to make sure that the 3-state drivers are never turned on in conflict with other drivers on DATABUS.

Baud Rate Generator

The BAUD Rate Generator (BRG) produces an enabling signal or signals for controlling the transmitter and the receiver. In traditional UART designs, transmitter and receiver clocks, which typically are the same frequency, are used to perform the necessary timing for controlling the BAUD rate of the transmitted serial information and for controlling the sampling of received information. Since we have no separate clock source, we cannot use this approach, but must instead depend upon the BRG to produce enable signals for these purposes instead of separate clocks. The reason for producing an enable signal instead of a clock is to avoid the problem of having multiple clock domains. The enable signal is produced by a down counter and decoder circuit to perform divisions of the frequency of CLK. Note that in Verilog, an enable is not used as a clock, but as a condition for performing or not performing actions:

```
always@(posedge clk)
    if (enable)
      ...
    else
      ...
```

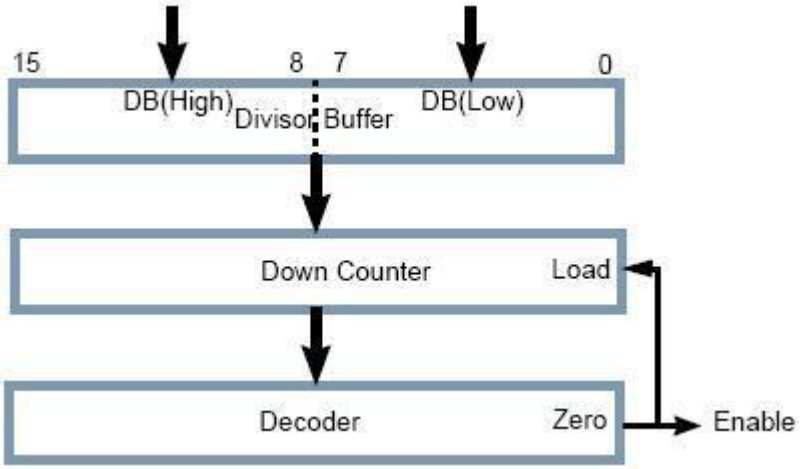


Figure 3: Baud Rate Generator

The frequency of the occurrence of the most frequent enable, which has duration of one CLK period, is typically 2^n × baud rate, where n ranges from 2 to 4. We will assume that 4 is used. In the design of the BRG, we have a special problem in that we will vary the frequency of CLK driving the BRG. Thus, the BRG must be programmable to maintain a fixed baud rate in the face of a changing clock frequency. Programming is achieve by the processor loading two bytes, DB(High) and DB(Low) into the Divisor Buffer in Fig. 3. This buffer drives the data inputs to the down counter as shown in Figure 3. The divisor is the nearest integer to (clock frequency/(2^n × baud rate) − 1). When the counter contains 0, it is loaded with the divisor. So the count goes from the divisor to zero at the CLK rate. If the decoder consists of a zero detect on the entire counter, then an enable is produced for a single clock period at a rate of one every (CLK/divisor + 1) = CLK/(2^n × baud rate) With the divisor ranging from 1 to 65,535, division by 2 through 65,536 can be accomplished. By using appropriate divisor and designing appropriate decoder, pulses can be produced at 2^n times the baud rate. If an additional enable is required, for example, at the baud rate itself, it can be generated by a 4-bit down counter with a zero decoder and with the (2^n × baud rate) enable as an input. The following example illustrates the Basic Baud Rate concept.

Example: *Suppose that CLK has frequency 25 MHz and that the desired enable frequency is (2^n × baud rate) where n = 4 and the baud rate is 9600 (bits/second). The divisor required is 25,000,000/(16 × 9,600) − 1 = 161.76 which is rounded to 162. This becomes A2 in hexadecimal. At count time 0, the counter will be loaded with 162 and will be decremented every 40 ns. The counter will be 0 at count time 163, so the interval of time for the counter to be loaded and count down is 163 × 40 ns = 6.52 μs. Inverting, the frequency is 153,374.23 Hz. Dividing by 16, this corresponds to a baud rate of 9586 bits/second. Based on calculations, we have estimated that an error of + or - 3 percent is tolerable, so this design is well within tolerance.*

In your final project, to insure communication with the "console" (Hyperterminal) immediately after a reset, a divisor should be loaded into the Divisor Buffer upon processor reset. This divisor should be in dedicated locations in memory. The memory should also contain a "boot program" that executes automatically on reset to load the divisor in memory into the Divisor Buffer. The clock frequency has been set before the reset is applied. Further, in order to provide means of setting a division before your system can execute code, the reset should initialize the Divisor Buffer to the divisor corresponding to a clock of 100 MHz and 9600 bits/second. You can modify this value to something else if necessary for your design.

More Information

For information on the remaining parts of the SPART, please consult the reference and the files in the folder Miscellaneous HDL Code for UART-Like Hardware in the FAQ folder on the course Website. Note that this design does not need to contain many of the features in these examples. For example, there is no synchronous operation as in the 8251A and there is no initialization except for Reset and the loading of the BAUD Rate Generator. Further, there is no error checking, no parity bit and a fixed number of stop bits.

Hardware Testbench

In order to test your SPART in the lab, you will need circuitry to mimic the behavior of the Processor. We will refer to this as a hardware testbench. This hardware testbench should be able to:

- Demonstrate the ability to transmit and receive characters by, for example, entering characters on a dumb terminal keyboard and echoing them back to the dumb terminal display
- Loading the Baud Rate Generator with an arbitrary value.

The testbench needs to provide four hardwired divisor values. The testbench must load one of these values into the Divisor Buffer after reset has been applied and removed. The value loaded will be determined by the values provided by two DIP switches that are set before reset is applied.

DIP Setting	Baud Rate
00	4800
01	9600
10	19200
11	38400

Hardware Harness

Due to the complexity of the XUP board and the possibility of causing damage by improper design or pin assignment, you are required to use a harness that surrounds your design. Later designs done by you will use your own pin assignments and configurations, but for now a harness will be provided. There are 4 provided files for the miniproject.

- Top_level.v - A the top level of the project which connects to external I/O pins as well as instantiating the SPART and your "Processor"

- Top_level.ucf – the Universal Constraints File which specifies I/O pins and clocking parameters
- Spart.v – An empty module for you to create your spart within.
- Driver.v – An empty module for you to create your Processor driver within.

Implementation Information

When doing implementation, note that the family is Virtex-5 and the device model is XC5VLX110T.

Lab Work

In lab you are to demonstrate the operation of your SPART as follows:

- Show that characters can be transferred between the dumb terminal and your hardware testbench via the SPART.
- By transmitting at multiple baud rates

Report

Your report should consist of the following:

- Verilog code for your entire design with clear, useful commenting
- An accompanying narrative description of the function for the overall SPART and each of the blocks including the testbench
- A record of the experiment conducted including the characters transmitted for a basic test
- A discussion of problems encountered in the design and solutions employed.

Updated 9/1/2014

ECE 554 Miniproject 1
Design Document
Fall 2015
Kai Zhao
John Roy

Function of SPART

The SPART is divided into several modules that each perform specific parts of the functionality of the device. These modules are the bus interface, the baud rate generator, the transmit unit, and the receive unit.

The bus interface acts as the connection between the processor and the SPART. Its external connections are the databus and the I/O address, ioaddr. The databus acts as a bi-directional 8-bit bus between the processor and the SPART. It carries data that is sent and received as well as baud rate information. The ioaddr line specifies which data will be sent and where. It can specify that data through the databus is to be received, sent, used as the baud rate, or used by the processor as a status register.

The baud rate generator gives enable signals to the transmit and receive control units. It is loaded with a count-down corresponding to a baud rate specified by the processor (either 4800, 9600, 19200, or 38400) and then it counts down from that number on each clock cycle. When the counter reaches zero, an enable signal is sent to the transmit or the receive control unit to send or capture data in the serial line. The serial line operates at a frequency much slower than the internal clock signal of the SPART, so each serial bit is sent at a specific multiple of the internal clock frequency. Similarly, the line is sampled at a multiple of the internal clock frequency specific to the baud rate.

The transmit control unit takes data from the bus interface and sends it down the serial line. It shifts the data out to convert it from parallel to serial. The shifting happens the baud rate generator gives an enable signal. This ensures that the data is sent out at the rate specified for the serial line.

The receive control unit takes data from the serial port and shifts it into a register to send to the bus interface. It only shifts data in when the baud rate generator sends an enable signal when the divisor is counted down, so the unit only samples data once per cycle of the baud rate.

Together, these units send out some control signals to the processor to ensure proper operation. The receive control unit sends a receive data available signal (rda) to ensure the processor takes in data when it is full and available. The transmit control unit sends a transmit buffer ready (tbr) signal to the processor so the processor won't send data before the buffer is sending out previous data. The processor also sends signals to the SPART including chip selection, clock, reset, and read/write. All of these ensure proper functionality of the SPART.

Record of Experiments

Unit testing was used to ensure proper functionality of each module. For the bus interface, the tests included writing to each baud rate generator register, reading the status register, writing to the transmit controller, and reading from the receive controller. For the baud rate generator, the tests included writing to each baud rate register, timing the enables after each possible baud rate configuration, and ensuring that no enable signals were sent after chip select became low. For the transmit unit, the tests included writing out a specific value to the serial line and checking that it had the proper start and stop bits. For the receive unit, the tests included receiving a specific value from the serial line with the proper start and stop bits and ensuring that the received value was correct.

All of these tests included checks to ensure the values being passed were correct and informational error messages were included to inform the user of the incorrect value being passed.

Discussion of Problems

The main problem we had was figuring exactly what signals each finite state machine should wait for and exactly when to change states. For example, when receiving, we were not sure whether it is done when we receive all 10 bits (1 start bit, 8 bits of data, and 1 stop bit), or whether it was done when we receive only 9 bits and 1 stop bit. We were also not sure what signal we had to wait for before going back to IDLE state and being able to receive again.

One major problem we had was with the hardware and software issues. Xilinx did not work on some computers, ModelSim did not work in some computer, and the internet browser did not work on other computers, some Vertix 5 boards did not work, and some serial cables did not work. Xilinx ISE software, which can be used for development and testing, had frequently crashed or throw file not found errors. To work around this, we did most of our development and testing in ModelSim.

Another problem we faced was with our finite state machines, especially in the hardware testbench. Once we ensured that our state transitions were logically separate from our inputs and outputs the device worked as intended.

A final problem we faced was with viewing internal signals in ModelSim. Internal signals are essential to proper test-benching, and without them we were left with a black box to test. In order to view internal signals, we figured out that we had to run the command “log -r /*” to put all the signals, external and internal, in the objects window, then add them to the waveform from there.

Verilog Code: top_level

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     top_level
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     top_level for demonstration
//
// Dependencies:    spart, driver
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
// IOCS = I/O chip select. Set to one to activate the SPART
// IOR/Wbar - When 1, the reading from SPART to the PROCESSOR, when 0 reading from the PROCESSOR to the SPART
// RDA - Receive data available => data can be read by the processor from the SPART.
// TBR - Transmit buffer ready => data can be sent from the processor to the SPART
// IOADDR - I/O address of register to read or write
// DATABUS - Data to be sent or received
// SPART is fully synchronous with the clock - all transfers occur on a positive clock edge.
// The received data on RxD is asynchronous. The transmit via TxD is also asynchronous.
//////////////////////////////////////////////////////////////////////////////////

module top_level(       // inputs and outputs
    input clk,          // 100mhz clock
    input rst,          // Asynchronous reset, tied to dip switch 0
    output txd,         // RS232 Transmit Data
    input rxd,          // RS232 Recieve Data
    input [1:0] br_cfg  // Baud Rate Configuration, Tied to dip switches 2 and 3
);

    wire iocs;          // wires the connect the spart to driver
    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
```

Spart

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:          University of Wisconsin-Madison
// Engineer:         Kai Zhao, John Roy
//
// Create Date:      2015 Sept 15
// Design Name:      Miniproject1
// Module Name:      spart
// Project Name:
// Target Devices:   Vertix 5
// Tool versions:    ModelSim SE 10.3c; Xilinx 14.7
// Description:      spart for handling transmission to/from controller
//
// Dependencies:     bus_interface, baud_rate_generator, transmit_unit, receive_unit
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module spart(                       // inputs and outputs
    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] rate_tx_data;        // wires to connect submodules
    wire [7:0] data_received;

    bus_interface bus_interface0(   // instantiate the DUT
        .iocs(iocs),
        .iosrw(iosrw),
        .rda(rda),
        .tbr(tbr),
        .ioaddr(ioaddr),
        .databus(databus),
        .rate_tx_data(rate_tx_data),
        .data_received(data_received)
    );

    baud_rate_generator baud_rate_generator0( // instantiate th
        .clk(clk),
        .rst(rst), // confirm? not shown in diagram, but we th:
        .iocs(iocs),
        .ioaddr(ioaddr),
        .rate_tx_data(rate_tx_data),
        .enable(enable)
    );

    transmit_unit transmit_unit0(   // instantiate the DUT
        .clk(clk),
        .rst(rst),
        .iocs(iocs),
        .iorw(iorw),
        .tbr(tbr),
        .ioaddr(ioaddr),
        .txd(txd),
        .rate_tx_data(rate_tx_data),
        .enable(enable)
    );

    receive_unit receive_unit0(     // instantiate the DUT
        .clk(clk),
        .rst(rst),
        .iocs(iocs),
        .iorw(iorw),
        .rda(rda),
        .ioaddr(ioaddr),
        .rxd(rxd),
        .data_received(data_received),
        .enable(enable)
    );

endmodule
```

Bus Interface

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:          University of Wisconsin-Madison
// Engineer:         Kai Zhao, John Roy
//
// Create Date:      2015 Sept 15
// Design Name:      Miniproject1
// Module Name:      bus_interface
// Project Name:
// Target Devices:   Vertix 5
// Tool versions:    ModelSim SE 10.3c; Xilinx 14.7
// Description:      bus_interface to control which data to send and where to send it
//
// Dependencies:     none
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
```

```
module bus_interface(
    input iocs,
    input iosrw,
    input rda,
    input tbr,
    input [1:0] ioaddr,
    inout [7:0] databus,
    output reg [7:0] rate_tx_data,
    input [7:0] data_received
);

    always @(*) begin
        //if (ioaddr[1]) begin // if (ioaddr[1] == 1)                        // processor writing to baud rate generator
        //  rate_tx_data <= databus;
        //end
        rate_tx_data <= databus;
    end

    wire [7:0] databus_out;
    assign databus_out = (ioaddr[0]) ? {6'b000000, tbr, rda} : data_received;   // processor reading/SPART writing
    assign databus = (!iosrw || ioaddr[1]) ? 8'hzz : databus_out;               // high impedence is required for processor writing/SPART reading

    //initial begin                                 // pseudocode for the assign statement
    //  if (iosrw || ioaddr[1]) begin
    //      databus = 8'bzzzzzzzz;
    //  end else if (ioaddr[0]) begin
    //      databus = {6'b000000, tbr, rda};
    //  end else begin
    //      databus = data_received;
    //  end
    //end


endmodule
```

Bus Interface Test bench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     bus_interface_tb
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     test bench for testing bus_interface
//
// Dependencies:    bus_interface
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//      test case 1: test test processor writing to baud rate generator low");
//      test case 2: test processor writing a different value to baud rate generator high");
//      test case 3: test processor reading the status register demo");
//      test case 4: test processor writing a value to transmit controller");
//////////////////////////////////////////////////////////////////////////////////

module bus_interface_tb();

    reg iocs;               // signals that are connected to the DUT
    reg iosrw;
    reg rda;
    reg tbr;
    reg [1:0] ioaddr;
    wire [7:0] databus;
    wire [7:0] rate_tx_data;
    reg [7:0] data_received;

    reg [7:0] databus_reg;          // register for handling inouts
    assign databus = databus_reg;

    bus_interface DUT (         // instantiate the DUT
        .iocs(iocs),
        .iosrw(iosrw),
        .rda(rda),
        .tbr(tbr),
        .ioaddr(ioaddr),
        .databus(databus),
        .rate_tx_data(rate_tx_data),
        .data_received(data_received)
    );

    initial begin
        $display("testing bus_interface");
        iocs = 1;
        iosrw = 1;
        ioaddr = 2'b00;
        data_received = 8'h55;

        #5
        $display("test processor writing to baud rate generator low");
        ioaddr = 2'b10;
        databus_reg = 8'haa;
        #5
        if (rate_tx_data == databus_reg) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, expected: rate_tx_data == databus_reg, rate_tx_data = 8'h%h, databus_reg = 8'h%h", rate_tx_data, databus_reg);
        end
```

```
        #5
        $display("test processor writing a different value to baud rate generator high");
        ioaddr = 2'b11;
        databus_reg = 8'hbc;
        #5
        if (rate_tx_data == databus_reg) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, expected: rate_tx_data == databus_reg, rate_tx_data = 8'h%h, databus_reg = 8'h%h", rate_tx_data, databus_reg);
        end

        #5
        $display("test processor reading the status register demo");
        databus_reg = 8'hzz;
        tbr = 1;
        rda = 0;
        ioaddr = 2'b01;
        iosrw = 1;
        #5
        if (databus == {6'b000000, tbr, rda}) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, expected: databus == {6'b000000, tbr, rda}, databus = 8'h%h, {6'b000000, tbr, rda} = 8'h%h", databus, {
            6'b000000, tbr, rda});
        end

        #5
        $display("test processor writing a value to transmit controller");
        iosrw = 0;
        databus_reg = 8'hde;
        ioaddr = 2'b00;
        #5
        if (rate_tx_data == databus_reg) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, expected: rate_tx_data == databus_reg, rate_tx_data = 8'h%h, databus_reg = 8'h%h", rate_tx_data, databus_reg);
        end
```

```
        #5
        $display("test processor reading a value from receive controller");
        ioaddr = 2'b00;
        iosrw = 1;
        data_received = 8'hf1;
        databus_reg = 8'hzz;
        #5
        if (databus == data_received) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, expected: databus == data_received, rate_tx_data = 8'h%h, databus_reg = 8'h%h", databus, data_received);
        end

        $stop;
    end

endmodule
```

Baud rate generator

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     baud_rate_generator
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     baud_rate_generator to take baud_rate input and control enable signal
//
// Dependencies:    none
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//  need to support baud rate 4800, 9600, 19200, 38400
//      divsion for 4800 = 100000000/(16*4800) - 1 = 1300
//      divsion for 9600 = 100000000/(16*9600) - 1 = 650
//      divsion for 19200 = 100000000/(16*19200) - 1 = 325
//      divsion for 38400 = 100000000/(16*38400) - 1 = 162
//////////////////////////////////////////////////////////////////////////////////

module baud_rate_generator(                                         // inputs and outputs
    input clk,
    input rst,
    input iocs,
    input [1:0] ioaddr,
    input [7:0] rate_tx_data,
    output reg enable
);

    reg [15:0] division_buffer;
    reg [15:0] baud_rate_counter;
```

```
    always @(posedge clk) begin
        if (iocs) begin
            if (rst) begin                                          // if reset, then enable
                division_buffer <= 1;
                baud_rate_counter <= 1;
            end if (ioaddr == 2'b11 && division_buffer[15:8] != rate_tx_data) begin
                division_buffer[15:8] <= rate_tx_data;              // if high and high is not already set
                baud_rate_counter <= {rate_tx_data, division_buffer[7:0]} - 1;
            end else if (ioaddr == 2'b10 && division_buffer[7:0] != rate_tx_data) begin
                division_buffer[7:0] <= rate_tx_data;               // if low and low is not already set
                baud_rate_counter <= {division_buffer[15:8], rate_tx_data} - 1;
            end else if (baud_rate_counter == 0) begin              // if 0, then reset
                baud_rate_counter <= division_buffer;
            end else begin                                          // normally decrement
                baud_rate_counter <= baud_rate_counter - 1;
            end
        end
    end

    always @(posedge clk) begin
        if (iocs) begin                                             // never enable if iocs is low
            if (baud_rate_counter == 0) begin                       // enable when counter reaches 0
                enable <= 1;
            end else begin
                enable <= 0;
            end
        end
    end


endmodule
```

Baud rate generator test bench

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     baud_rate_generator_tb
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     test bench for testing baud_rate_generator
//
// Dependencies:    baud_rate_generator
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//      test case 1: test first cycle of baud rate by setting only the low byte
//      test case 2: test second cycle of baud rate by setting only the low byte
//      test case 3: test first cycle of baud rate by setting only the high byte
//      test case 4: test second cycle of baud rate by setting only the high byte
//      test case 5: test first cycle of baud rate by setting baud rate = 16'ha531
//      test case 6: test second cycle of baud rate by setting baud rate = 16'ha531
//      test case 7: test baud rate still works after changing the address
//      test case 8: test that enable is never high when iocs is low
////////////////////////////////////////////////////////////////////////////////

module baud_rate_generator_tb();

    reg clk;                    // signals that are connected to the DUT
    reg rst;
    reg iocs;
    reg [1:0] ioaddr;
    reg [7:0] rate_tx_data;
    wire enable;

    baud_rate_generator DUT (       // instantiate the DUT
        .clk(clk),
        .rst(rst),
        .iocs(iocs),
        .ioaddr(ioaddr),
        .rate_tx_data(rate_tx_data),
        .enable(enable)
    );

    initial begin                   // initialize all variables
        $display("testing baud_rate_generator");
        clk = 0;
        rst = 1;
        iocs = 1;
        ioaddr = 2'b00;
        rate_tx_data = 0;

        @(posedge clk);
        rst = 0;
        @(posedge clk);

        //      test case 1: test first cycle of baud rate by setting only the low byte
        $display("test first cycle of processor only writing to baud rate generator low");
        ioaddr = 2'b11;
        rate_tx_data = 0;
        @(posedge clk);
        ioaddr = 2'b10;
        rate_tx_data = 3;
        @(posedge clk);             // wait 1 cycle for baud rate to load
        repeat (3) begin            // wait 3 cycles since 3 is loaded
            @(posedge clk);
            if (enable) begin
                $display("\ttest failed, enable should have been kept low for 3 cycles");
            end
        end
        @(posedge clk);             // wait another for baud_rate_generator to react to baud_rate_counter reaching 0
        if (enable) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, enable should have been pulsed high on the 4th cycle");
        end

        //      test case 2: test second cycle of baud rate by setting only the low byte
        $display("test second cycle of processor only writing to baud rate generator low");
        repeat (3) begin
            @(posedge clk);
            if (enable) begin
                $display("\ttest failed, enable should have been kept low for 3 cycles");
            end
        end
        @(posedge clk);
        if (enable) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, enable should have been pulsed high on the 4th cycle");
        end
```

```
96          //      test case 3: test first cycle of baud rate by setting only the high byte
97          @(posedge clk);
98          $display("test first cycle of processor only writing to baud rate generator high");
99          ioaddr = 2'b11;
100         rate_tx_data = 8'h20;
101         @(posedge clk);
102         ioaddr = 2'b10;
103         rate_tx_data = 0;
104         @(posedge clk);
105         repeat (8192) begin // equivalent to 16'h2000
106             @(posedge clk);
107             if (enable) begin
108                 $display("\ttest failed, enable should have been kept low for 8192 cycles");
109             end
110         end
111         @(posedge clk);
112         if (enable) begin
113             $display("\ttest passed");
114         end else begin
115             $display("\ttest failed, enable should have been pulsed high on the 8193rd cycle");
116         end
117
118
119         //      test case 4: test second cycle of baud rate by setting only the high byte
120         $display("test second cycle of processor only writing to baud rate generator high");
121         repeat (8192) begin
122             @(posedge clk);
123             if (enable) begin
124                 $display("\ttest failed, enable should have been kept low for 8192 cycles");
125             end
126         end
127         @(posedge clk);
128         if (enable) begin
129             $display("\ttest passed");
130         end else begin
131             $display("\ttest failed, enable should have been pulsed high");
132         end
```

```
135         //      test case 5: test first cycle of baud rate by setting baud rate = 16'ha531
136         @(posedge clk);
137         $display("test first cycle of processor writing 8'ha531 to baud rate generator");
138         ioaddr = 2'b11;
139         rate_tx_data = 8'ha5;
140         @(posedge clk);
141         ioaddr = 2'b10;
142         rate_tx_data = 8'h31;
143         @(posedge clk);
144         repeat (42289) begin // equivalent to 16'ha531
145             @(posedge clk);
146             if (enable) begin
147                 $display("\ttest failed, enable should have been kept low for 42289 cycles");
148             end
149         end
150         @(posedge clk);
151         if (enable) begin
152             $display("\ttest passed");
153         end else begin
154             $display("\ttest failed, enable should have been pulsed high");
155         end
156
157
158         //      test case 6: test second cycle of baud rate by setting baud rate = 16'ha531
159         $display("test second cycle of processor writing 8'ha531 to baud rate generator");
160         repeat (42289) begin
161             @(posedge clk);
162             if (enable) begin
163                 $display("\ttest failed, enable should have been kept low for 42289 cycles");
164             end
165         end
166         @(posedge clk);
167         if (enable) begin
168             $display("\ttest passed");
169         end else begin
170             $display("\ttest failed, enable should have been pulsed high");
171         end
```

```
174         //      test case 7: test baud rate still works after changing the address
175         $display("test baud rate generator still workes even after changing address");
176         ioaddr = 2'b00;
177         rate_tx_data = 8'h76;
178         repeat (42289) begin
179             @(posedge clk);
180             if (enable) begin
181                 $display("\ttest failed, enable should have been kept low for 42289 cycles");
182             end
183         end
184         @(posedge clk);
185         if (enable) begin
186             $display("\ttest passed");
187         end else begin
188             $display("\ttest failed, enable should have been pulsed high");
189         end
190 
191 
192         //      test case 8: test that enable is never high when iocs is low
193         $display("test that enable is never high when iocs is low");
194         iocs = 0;
195         repeat (65536) begin    // equivalent to 16'hffff + 1
196             @(posedge clk);
197             if (enable) begin
198                 $display("\ttest failed, enable should should not go high if iocs is low");
199                 $stop;
200             end
201         end
202         $display("\ttest passed");
203 
204         $stop;
205     end
206 
207     always
208         #2 clk = !clk;
209 
210 endmodule
```

Transmit unit

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:          University of Wisconsin-Madison
// Engineer:         Kai Zhao, John Roy
//
// Create Date:      2015 Sept 15
// Design Name:      Miniproject1
// Module Name:      transmit_unit
// Project Name:
// Target Devices:   Vertix 5
// Tool versions:    ModelSim SE 10.3c; Xilinx 14.7
// Description:      transmit_unit to transmit data from SPART to RS232
//
// Dependencies:     none
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module transmit_unit(                   // inputs and outputs
    input clk,
    input rst,
    input iocs,
    input iorw,
    output reg tbr,
    input [1:0] ioaddr,
    output txd,
    input [7:0] rate_tx_data,
    input enable
);

    reg [9:0] shift_reg;
    reg [3:0] bit_cnt;
    reg load, shift, set_tbr;
    localparam numberOfBitsPerPacket = 9; // 1 start bit, 1 stop bit, and 8 bits of
```

```
    reg [9:0] shift_reg;
    reg [3:0] bit_cnt;
    reg load, shift, set_tbr;
    localparam numberOfBitsPerPacket = 9; // 1 start bit, 1 stop bit, and 8 bits of data, -1 because checked after shifting

    // send 1 bit of txd at a time
    assign txd = shift_reg[0];

    // handle tx_buffer, shift_reg, and bit_cnt based on the FSM output
    // If data is ready to tx, send 1 low start bit, then 8 data bits, then 1 high stop bit.
    // Otherwise, hold the output high
    always @(posedge clk) begin
        if (load) begin
            // start at 1, because this takes a clock cycle as well
            shift_reg <= {1'b1, rate_tx_data, 1'b0};
            bit_cnt <= numberOfBitsPerPacket;
        end else if (set_tbr) begin
            // set tx_data_out[0] to not start the receiver
            shift_reg[0] <= 1'b1;
        end else if (shift) begin
            bit_cnt <= bit_cnt - 1;
            shift_reg <= {shift_reg[0], shift_reg[9:1]};
        end
    end
```

```
    localparam IDLE = 1'b0, TRANS = 1'b1;
    reg state, nxt_state;

    // handle start transitions of the FSM
    always @(posedge clk) begin
        if(rst) begin
            state <= IDLE;
            tbr <= 0;
        end else begin
            state <= nxt_state;
            tbr <= set_tbr;
        end
    end

    always @(*) begin
        // set defaults
        nxt_state = IDLE;
        load = 0;
        shift = 0;
        set_tbr = 0;
        case(state)
            IDLE: begin
                // if begin, then wait for trmt signal
                if (ioaddr == 2'b00 && !iorw) begin // if write signal, then load variables
                    load = 1;
                    nxt_state = TRANS;
                end else begin
                    set_tbr = 1;
                end
            end
            TRANS: begin
                // wait for baud count to increase until it is time to shift
                nxt_state = TRANS;
                // if enable, then shift
                if (enable) begin
                    shift = 1;
                    // if bit_cnt == 0, then done and return to IDLE
                    if (bit_cnt == 0) begin
                        nxt_state = IDLE;
                    end
                end
            end
            default: begin
                // default state for safety
                // return to IDLE
            end
        endcase
    end

endmodule
```

Transmit unit test bench

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     transmit_unit_tb
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     test bench for testing transmit_unit
//
// Dependencies:    transmit_unit
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//      test case 1: test transmitting 8'haa
//      test case 2: test transmitting 8'b0011_1001
////////////////////////////////////////////////////////////////////////////////

module transmit_unit_tb();

    reg clk;                // signals that are connected to the DUT
    reg rst;
    reg iocs;
    reg iorw;
    wire tbr;
    reg [1:0] ioaddr;
    wire txd;
    reg [7:0] rate_tx_data;
    reg enable;

    transmit_unit DUT(          // instantiate the DUT
        .clk(clk),
        .rst(rst),
        .iocs(iocs),
        .iorw(iorw),
        .tbr(tbr),
        .ioaddr(ioaddr),
        .txd(txd),
        .rate_tx_data(rate_tx_data),
        .enable(enable)
    );

    initial begin               // initialize all variables
        $display("testing transmit_unit");
        clk = 0;
        rst = 1;
        iocs = 1;
        iorw = 1;
        ioaddr = 0;
        rate_tx_data = 8'haa; // 8'b1010_1010 = 10'b11_0101_0100 = 10'h354
        enable = 0;
        @(posedge clk);
        rst = 0;
        ioaddr = 2'b00;

        $display("testing transmitting first data byte of oscillating 0s and 1s (8'haa)");
        iorw = 0;
        @(posedge clk);
        iorw = 1;
        while(tbr != 1) begin   // wait until done
            pulse_enable();
        end
```

```
68          $display("testing transmitting second data byte of 8'b0011_1001");
69          rate_tx_data = 8'b0011_1001; // 10_0111_0010 = 272
70          iorw = 0;
71          @(posedge clk);
72          iorw = 1;
73          @(posedge clk);
74          while(tbr != 1) begin   // wait until done
75              pulse_enable();
76          end
77          $stop;
78      end
79
80
81      task pulse_enable;          // task to pulse enable
82          begin                   // so that receiver and transmitter can continue
83              repeat (50)         // wait some time
84                  @(posedge clk);
85              enable = 1;         // set enable high
86              @(posedge clk);     // for 1 clock cycle
87              enable = 0;         // then reset enable
88          end
89      endtask
90
91
92      always
93          #2 clk = !clk;
94  endmodule
```

Receive unit

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     receive_unit
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     receive_unit to receive data from RS232 into SPART
//
// Dependencies:    none
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module receive_unit(                // inputs and outputs
    input clk,
    input rst,
    input iocs,
    input iorw,
    output reg rda,
    input [1:0] ioaddr,
    input rxd,
    output reg [7:0] data_received,
    input enable
);
```

```
    reg [9:0] shift_reg;            // used to hold data
    reg [3:0] bit_cnt;              // used to count number of bits to determine whether it is finished
    reg load, shift, set_rda;       // FSM signals
    localparam numberOfBitsPerPacket = 9;   // 1 start bit, 1 stop bit, and 8 bits of data, -1 because checked after shifting

    always @(posedge clk) begin
        if (load) begin             // if load, then set the number of bits
            bit_cnt <= numberOfBitsPerPacket;
        end else if (shift) begin   // if shift, then rotate shift register and decrement bit counter
            bit_cnt <= bit_cnt - 1;
            shift_reg <= {rxd, shift_reg[9:1]};
        end else if (set_rda) begin // if done, then move byte to output
            data_received <= shift_reg[8:1];
        end
    end

    localparam IDLE = 2'b00, RECV = 2'b01, DONE = 2'b10; // states
    reg [1:0] state, nxt_state;

    // handle start transitions of the FSM
    always @(posedge clk) begin     // always go to next state and set rda <= set_rda
        if (rst) begin
            state <= IDLE;
            rda <= 0;
        end else begin
            state <= nxt_state;
            rda <= set_rda;
        end
    end
```

```
64      always @(*) begin
65          // set defaults
66          nxt_state = IDLE;
67          load = 0;
68          shift = 0;
69          set_rda = 0;
70          case (state)
71              IDLE: begin     // initial IDLE state, go to RECV receiving and if saw a start bit
72                  if (ioaddr == 2'b00 && iorw && !rxd) begin
73                      load = 1;
74                      nxt_state = RECV;
75                  end
76              end
77              RECV: begin     // RECV state to receive data, shift everytime a bit comes in
78                  nxt_state = RECV;
79                  // if enable, then shift
80                  if (enable) begin
81                      shift = 1;
82                      // if bit_cnt == 0, then done, hold data until it is read
83                      if (bit_cnt == 0) begin
84                          nxt_state = DONE;
85                      end
86                  end
87              end
88              default: begin  // same as DONE state to hold data
89                  // wait for transmit signal to be able to receive next byte of data
90                  nxt_state = DONE;
91                  set_rda = 1;
92                  if (ioaddr == 2'b00 && !iorw) begin
93                      nxt_state = IDLE;
94                  end
95              end
96          endcase
97      end
98
99  endmodule
```

Receive unit test bench

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:          University of Wisconsin-Madison
// Engineer:         Kai Zhao, John Roy
//
// Create Date:      2015 Sept 15
// Design Name:      Miniproject1
// Module Name:      receive_unit_tb
// Project Name:
// Target Devices:   Vertix 5
// Tool versions:    ModelSim SE 10.3c; Xilinx 14.7
// Description:      test bench for testing receive_unit
//
// Dependencies:     receive_unit
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//       test case 1: test receiving first data byte of oscillating 0s and 1s
//       test case 2: test receiving second data byte of 8'b0011_1001
////////////////////////////////////////////////////////////////////////////////

module receive_unit_tb();

    reg clk;            // signals that are connected to the DUT
    reg rst;
    reg iocs;
    reg iorw;
    wire rda;
    reg [1:0] ioaddr;
    reg rxd;
    wire [7:0] data_receieved;
    reg enable;

    receive_unit DUT(       // instantiate the DUT
        .clk(clk),
        .rst(rst),
        .iocs(iocs),
        .iorw(iorw),
        .rda(rda),
        .ioaddr(ioaddr),
        .rxd(rxd),
        .data_receieved(data_receieved),
        .enable(enable)
    );

    initial begin           // initialize all variables
        $display("testing receive_unit");
        clk = 0;
        rst = 1;
        iocs = 1;
        iorw = 0;
        ioaddr = 0;
        rxd = 0;
        enable = 0;
        @(posedge clk);
        rst = 0;
        ioaddr = 2'b00;

        $display("testing receiving first data byte of oscillating 0s and 1s");
        iorw = 0;       // set iorw = 0 to make receiver avialable to receive
        @(posedge clk);
        iorw = 1;       // set iorw = 0 to receive
        while(rda != 1) begin   // oscillate RX to see 01010101 in cmd
            set_data(1);
            set_data(0);
        end
        if (data_receieved == 8'h55) begin // check if equals 8'h55 since i sent it oscillating 0s and 1st
            $display("test passed");
        end else begin
            $display("test failed, expected data_receieved = 8'h55, actual data_receieved = 8'h%h", data_receieved);
        end

        $display("testing receiving second data byte of 8'b0011_1001");
        rxd = 1;        // need to send 1 for 1 clock cycle for stop bit
        iorw = 0;       // make receiver avialable again
        @(posedge clk);
        iorw = 1;
        set_data(0);        // 1st bit, need to be 0 to start
        set_data(1);        // 2nd bit (least significant data bit)
        set_data(0);        // 3rd bit
        set_data(0);        // 4th bit
        set_data(1);        // 5th bit
        set_data(1);        // 6th bit
        set_data(1);        // 7th bit
        set_data(0);        // 8th bit
        set_data(0);        // 9th bit
        set_data(1);        // 10th bit, need to be 1 to stop
```

```
89          while(rda != 1) begin   // wait until done
90              #2;
91          end
92          if (data_receieved == 8'b0011_1001) begin // check if it makes the data i sent
93              $display("test passed");
94          end else begin
95              $display("test failed, expected data_receieved = 8'b0011_1001, actual data_receieved = 8'b%b", data_receieved);
96          end
97          $stop;
98      end
99
100
101     task set_data;          // task for sending data
102         input data;
103         begin
104             rxd = data;
105             #200;
106             @(posedge clk);
107             enable = 1;
108             @(posedge clk);
109             enable = 0;
110         end
111     endtask
112
113
114     always              // clock
115         #2 clk = !clk;
116
117 endmodule
```

Transmit receive unit test bench

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     transmit_receive_unit_tb
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     test bench for testing transmit_unit and receive_unit
//
// Dependencies:    transmit_unit, receive_unit
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//      test case 1: test transmitting and receiving oscillating 0s and 1s (8'haa)
//      test case 2: test transmitting and receiving 8'b0011_1001
//////////////////////////////////////////////////////////////////////////////////

module transmit_receive_unit_tb();

    reg clk;                // signals that are connected to the DUT
    reg rst;
    reg iocs;
    reg iorw;
    wire tbr;
    wire rda;
    reg [1:0] ioaddr;
    wire trxd;
    reg [7:0] rate_tx_data;
    wire [7:0] data_receieved;
    reg enable;
```

```
    transmit_unit tDUT(         // instantiate the first DUT
        .clk(clk),
        .rst(rst),
        .iocs(iocs),
        .iorw(iorw),
        .tbr(tbr),
        .ioaddr(ioaddr),
        .txd(trxd),
        .rate_tx_data(rate_tx_data),
        .enable(enable)
    );

    receive_unit rDUT(          // instantiate the second DUT
        .clk(clk),
        .rst(rst),
        .iocs(iocs),
        .iorw(iorw),
        .rda(rda),
        .ioaddr(ioaddr),
        .rxd(trxd),
        .data_receieved(data_receieved),
        .enable(enable)
    );
```

```
    initial begin                // initialize all variables
        $display("testing transmit_unit and receive_unit");
        clk = 0;
        rst = 1;
        iocs = 1;
        iorw = 1;
        ioaddr = 0;
        rate_tx_data = 8'haa; // 8'b1010_1010 = 10'b11_0101_0100 = 10'h354
        enable = 0;
        @(posedge clk);
        rst = 0;
        ioaddr = 2'b00;

        $display("testing transmitting first data byte of oscillating 0s and 1s (8'haa)");
        iorw = 0;
        @(posedge clk);
        iorw = 1;
        while(tbr != 1) begin   // wait until done
            pulse_enable();
        end
        if (data_receieved == 8'haa) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, expected data_receieved = 8'haa, actual data_receieved = 8'h%h", data_receieved);
        end

        $display("testing transmitting second data byte of 8'b0011_1001");
        rate_tx_data = 8'b0011_1001; // 10_0111_0010 = 272
        iorw = 0;
        @(posedge clk);
        iorw = 1;
        @(posedge clk);
        while(tbr != 1) begin   // wait until done
            pulse_enable();
        end
        if (data_receieved == 8'b0011_1001) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, expected data_receieved = 8'8'b0011_1001, actual data_receieved = 8'b%b", data_receieved);
        end
        $stop;
    end


    task pulse_enable;          // task to pulse enable signal
        begin
            repeat (50)         // wait multiple cycl cycles
                @(posedge clk);
            enable = 1;         // set enable high
            @(posedge clk);     // for 1 clock cycle
            enable = 0;         // reset enable
        end
    endtask


    always
        #2 clk = !clk;
endmodule
```

Driver

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     driver
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     driver for sending data to SPART
//
// Dependencies:    none
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module driver(                          // inputs and outputs
    input clk,
    input rst,
    input [1:0] br_cfg,
    output iocs,
    output iorw,
    input rda,
    input tbr,
    output reg [1:0] ioaddr,
    inout [7:0] databus
);
```

```
    assign iocs = 1;                    // can't stop, won't stop
    reg iorw_reg;
    assign iorw = iorw_reg; // set to reg so that it can be set and keep its prototype
    reg [7:0] inout_data;
    reg [7:0] held_data;
    reg update_baud, start_sending_baud_hi, start_sending_baud_lo, start_receiving, start_transmitting, receiving, transmitting;

    reg [2:0] old_br_cfg;
    localparam BAUD_4800 = 16'h0514;    // divsion for 4800 = 100000000/(16*4800) - 1 = d1300 = h0514
    localparam BAUD_9600 = 16'h028a;    // divsion for 9600 = 100000000/(16*9600) - 1 = d650 = h028a
    localparam BAUD_19200 = 16'h0145;   // divsion for 19200 = 100000000/(16*19200) - 1 = d325 = h0145
    localparam BAUD_38400 = 16'h00a2;   // divsion for 38400 = 100000000/(16*38400) - 1 = d162 = h00a2
    reg [15:0] new_baud;

    localparam IDLE = 3'b000, BAUDHI = 3'b001, BAUDLO = 3'b010, RECV = 3'b011, TRANS = 3'b100;
    reg [2:0] state, nxt_state;
```

```
    // handle start transitions of the FSM
    always @(posedge clk) begin
        if (rst) begin
            state <= IDLE;
            old_br_cfg <= 3'b100;
        end else begin
            if (update_baud) begin                      // load baud into new_baud register
                old_br_cfg <= {1'b0, br_cfg};
                new_baud <=                             // case statement to load bad rate
                    br_cfg == 0 ? BAUD_4800 :           // based on br_cfg
                    br_cfg == 2 ? BAUD_19200 :
                    br_cfg == 3 ? BAUD_38400 :
                    BAUD_9600;                          // default case
            end else if (start_sending_baud_hi) begin   // load baud rate high
                inout_data <= new_baud[15:8];
                ioaddr <= 2'b11;
            end else if (start_sending_baud_lo) begin   // load baud rate low
                inout_data <= new_baud[7:0];
                ioaddr <= 2'b10;
            end else if (start_receiving) begin         // load available data
                inout_data <= databus;
            end else if (start_transmitting) begin      // put available data back on inout bus
                // do nothing, since value is already loaded in inout_data, which is pushed by databus if in receive state
                inout_data <= held_data;
            end else if (receiving) begin               // set it to receive
                ioaddr <= 2'b00;
                iorw_reg <= 1;
            end else if (transmitting) begin            // set it to transmite
                ioaddr <= 2'b00;
                iorw_reg <= 0;
            end else begin
                ioaddr <= 2'b01;                        // set to be able to read rda and tbr
            end
            state <= nxt_state;
        end
    end
```

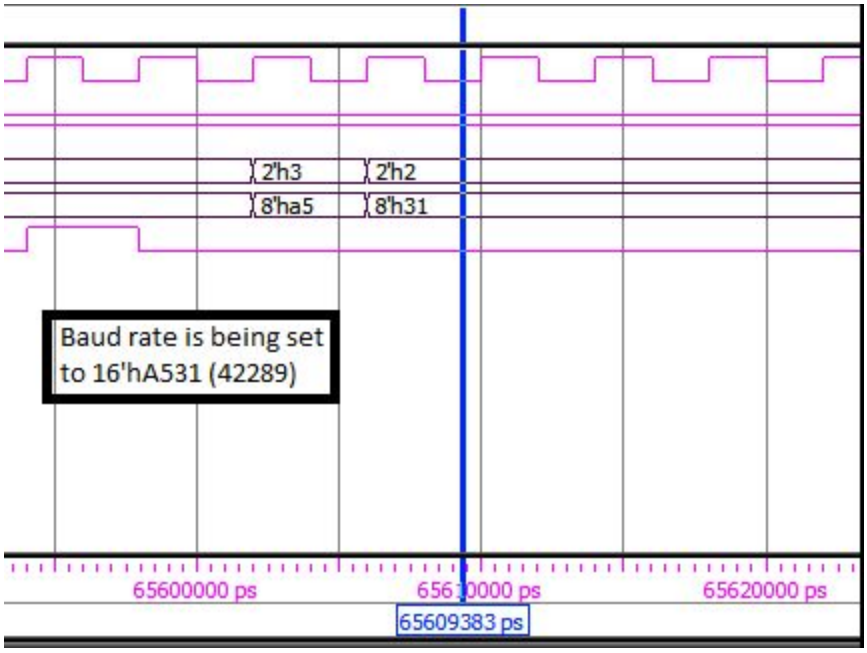
```
88      assign databus = (state == IDLE || state == RECV) ? 8'bzz : inout_data;
89
90      always @(*) begin
91          // set defaults
92          nxt_state = IDLE;
93          update_baud = 0;
94          start_sending_baud_hi = 0;
95          start_sending_baud_lo = 0;
96          start_receiving = 0;
97          receiving = 0;
98          start_transmitting = 0;
99          transmitting = 0;
100         case (state)
101             IDLE: begin                 // send baud rate if unitialized
102                 if (old_br_cfg[2] || br_cfg != old_br_cfg[1:0]) begin
103                     update_baud = 1;
104                     nxt_state = BAUDHI;
105                 end else if (rda) begin // receive data if data is ready to be received
106                     start_receiving = 1;
107                     nxt_state = RECV;
108                 end
109             end
110             BAUDHI: begin               // send baud rate high
111                 start_sending_baud_hi = 1;
112                 nxt_state = BAUDLO;
113             end
114             BAUDLO: begin               // send baud rate low
115                 start_sending_baud_lo = 1;
116                 nxt_state = IDLE;
117             end
118             RECV: begin                 // stay until you can transmit the data back
119                 nxt_state = RECV;
120                 receiving = 1;
121                 if (tbr) begin
122                     held_date = databus;
123                     start_transmitting = 1;
124                     nxt_state = TRANS;
125                 end
126             end
127             TRANS: begin                // stay until done transmitting
128                 nxt_state = TRANS;
129                 transmitting = 1;
130                 if (tbr) begin
131                     nxt_state = IDLE;
132                 end
133             end
134             default: begin
135                 // nothing, so return to IDLE
136             end
137         endcase
138     end
139
140 endmodule
```

Driver test bench

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:         University of Wisconsin-Madison
// Engineer:        Kai Zhao, John Roy
//
// Create Date:     2015 Sept 15
// Design Name:     Miniproject1
// Module Name:     driver_tb
// Project Name:
// Target Devices:  Vertix 5
// Tool versions:   ModelSim SE 10.3c; Xilinx 14.7
// Description:     test bench for testing driver
//
// Dependencies:    driver
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//      test case 1: test baud rate 4800
//      test case 2: test baud rate 9600
//      test case 3: test baud rate 19200
//      test case 4: test baud rate 38400
////////////////////////////////////////////////////////////////////////////////

module driver_tb();

    reg clk;            // signals that are connected to the DUT
    reg rst;
    reg [1:0] br_cfg;
    wire iocs;
    wire iorw;
    reg rda;
    reg tbr;
    wire [1:0] ioaddr;
    wire [7:0] databus;

    reg[7:0] databus_reg;       // register for handling inouts

    assign databus = databus_reg;

    driver DUT(         // instantiate the DUT
        .clk(clk),
        .rst(rst),
        .br_cfg(br_cfg),
        .iocs(iocs),
        .iorw(iorw),
        .rda(rda),
        .tbr(tbr),
        .ioaddr(ioaddr),
        .databus(databus)
    );

    initial begin           // initialize variables
        $display("testing driver");
        clk = 0;
        rst = 1;
        br_cfg = 2'b01;
        @(posedge clk);
        rst = 0;
        rda = 0;
        tbr = 0;
        databus_reg = 8'hzz;
        @(posedge clk);

        $display("Testing baud rate 4800");
        br_cfg = 2'b00;
        repeat (5)      // wait for baud_rate to load
            @(posedge clk);
        if (DUT.new_baud == DUT.BAUD_4800) begin
            $display("\ttest passed");
        end else begin
            $display("\ttest failed, DUT.new_baud = 16'h%h, DUT.BAUD_4800 = 16'h%h", DUT.new_baud, DUT.BAUD_4800);
        end

        $display("Testing baud rate 9600");
```

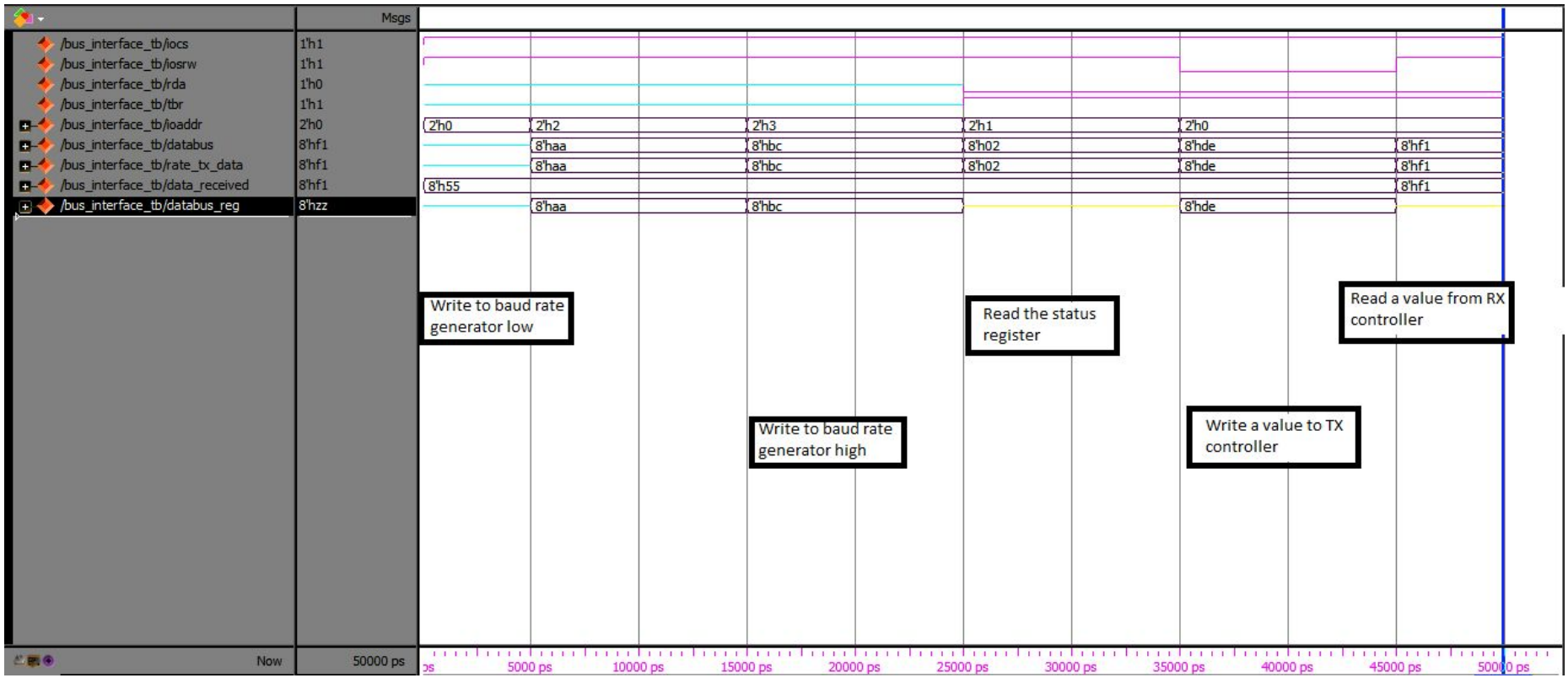
```
75          br_cfg = 2'b01;
76          repeat (5)
77              @(posedge clk);
78          if (DUT.new_baud == DUT.BAUD_9600) begin
79              $display("\ttest passed");
80          end else begin
81              $display("\ttest failed, DUT.new_baud = 16'h%h, DUT.BAUD_4800 = 16'h%h", DUT.new_baud, DUT.BAUD_9600);
82          end
83
84          $display("Testing baud rate 19200");
85          br_cfg = 2'b10;
86          repeat (5)
87              @(posedge clk);
88          if (DUT.new_baud == DUT.BAUD_19200) begin
89              $display("\ttest passed");
90          end else begin
91              $display("\ttest failed, DUT.new_baud = 16'h%h, DUT.BAUD_4800 = 16'h%h", DUT.new_baud, DUT.BAUD_19200);
92          end
93
94          $display("Testing baud rate 38400");
95          br_cfg = 2'b11;
96          repeat (5)
97              @(posedge clk);
98          if (DUT.new_baud == DUT.BAUD_38400) begin
99              $display("\ttest passed");
100         end else begin
101             $display("\ttest failed, DUT.new_baud = 16'h%h, DUT.BAUD_4800 = 16'h%h", DUT.new_baud, DUT.BAUD_38400);
102         end
103
104         #100;
105         $stop;
106     end
107
108     always
109         #2 clk = !clk;
110
111 endmodule
```

Baud Rate Generator Testbench



```
VSIM 5> run -all
# testing baud_rate_generator
# test first cycle of processor only writing to baud rate generator low
#       test passed
# test second cycle of processor only writing to baud rate generator low
#       test passed
# test first cycle of processor only writing to baud rate generator high
#       test passed
# test second cycle of processor only writing to baud rate generator high
#       test passed
# test first cycle of processor writing 8'ha531 to baud rate generator
#       test passed
# test second cycle of processor writing 8'ha531 to baud rate generator
#       test passed
# test baud rate generator still workes even after changing address
#       test passed
# test that enable is never high when iocs is low
#       test passed
# ** Note: $stop    : I:/0school/ece554/miniproject1/baud_rate_generator_tb.v(188)
#    Time: 835238 ns  Iteration: 1  Instance: /baud_rate_generator_tb
# Break in Module baud_rate_generator_tb at I:/0school/ece554/miniproject1/baud_rate_generator_tb.v line 188
```

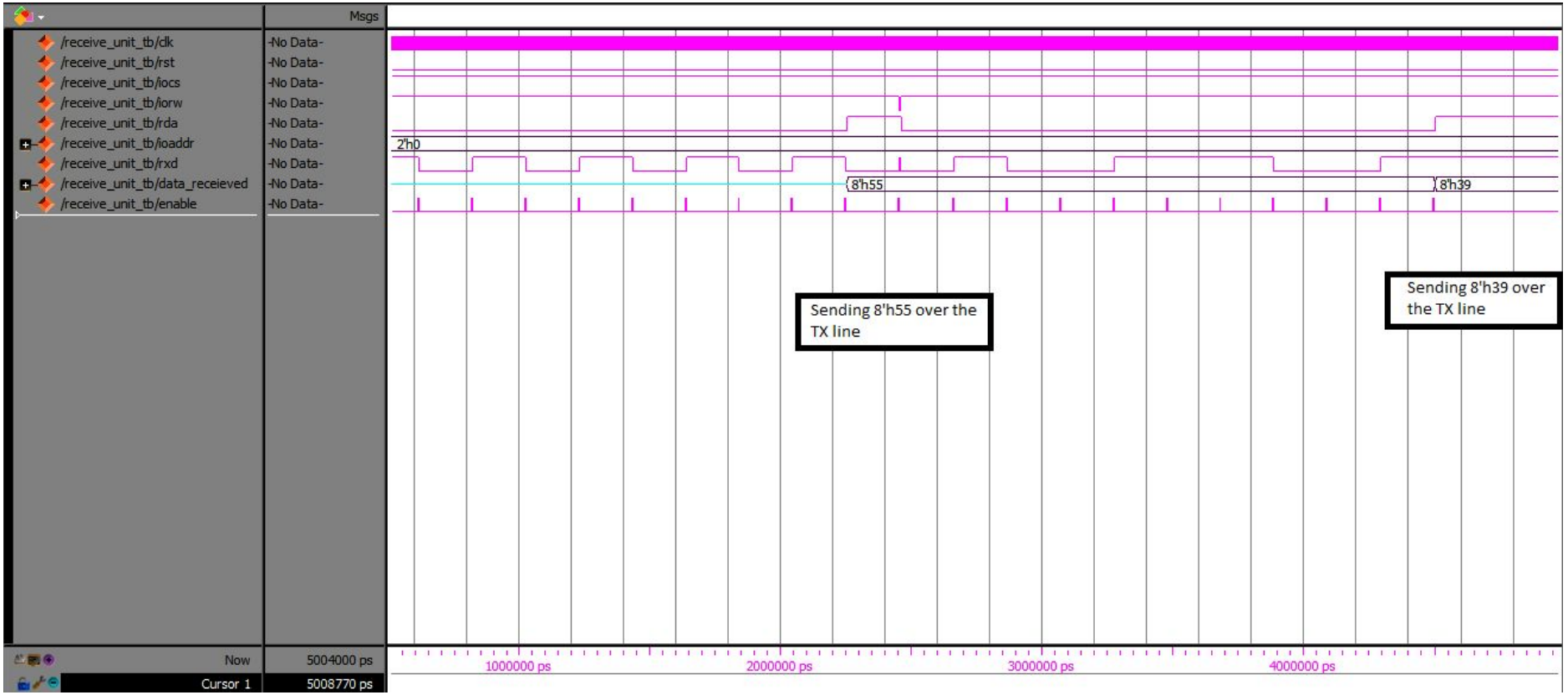
Bus Interface Testbench


Msgs
/bus_interface_tb/iocs
/bus_interface_tb/iosrw
/bus_interface_tb/rda
/bus_interface_tb/tbr
/bus_interface_tb/ioaddr
/bus_interface_tb/databus
/bus_interface_tb/rate_tx_data
/bus_interface_tb/data_received
/bus_interface_tb/databus_reg
1'h1
1'h1
1'h0
1'h1
2'h0
8'hf1
8'hf1
8'hf1
8'hzz
2'h0
2'h2
2'h3
2'h1
2'h0
8'haa
8'hbc
8'h02
8'hde
8'hf1
8'haa
8'hbc
8'h02
8'hde
8'hf1
8'h55
8'hf1
8'haa
8'hbc
8'hde
Write to baud rate generator low
Write to baud rate generator high
Read the status register
Write a value to TX controller
Read a value from RX controller
Now
50000 ps
5000 ps
10000 ps
15000 ps
20000 ps
25000 ps
30000 ps
35000 ps
40000 ps
45000 ps

```
VSIM 13> run -all
# testing bus interface
# test processor writing to baud rate generator low
#       test passed
# test processor writing a different value to baud rate generator high
#       test passed
# test processor reading the status register demo
#       test passed
# test processor writing a value to transmit controller
#       test passed
# test processor reading a value from receive controller
#       test passed
# ** Note: $stop    : I:/0school/ece554/miniproject1/bus_interface_tb.v(114)
#    Time: 50 ns  Iteration: 0  Instance: /bus_interface_tb
# Break in Module bus_interface_tb at I:/0school/ece554/miniproject1/bus_interface_tb.v line 114
```

Transmit Unit Testbench

```
VSIM 2> run -all
# testing transmitting first data byte of oscillating 0s and 1s (8'haa)
# testing transmitting second data byte of 8'b0011_1001
# ** Note: $stop    : I:/0school/ece554/miniproject1/transmit_unit_tb.v(74)
#    Time: 2258 ns  Iteration: 1  Instance: /transmit_unit_tb
# Break at I:/0school/ece554/miniproject1/transmit_unit_tb.v line 74
```


Msgs
/receive_unit_tb/clk
/receive_unit_tb/rst
/receive_unit_tb/iocs
/receive_unit_tb/iorw
/receive_unit_tb/rda
/receive_unit_tb/ioaddr
/receive_unit_tb/rxd
/receive_unit_tb/data_receieved
/receive_unit_tb/enable
-No Data-
2'h0
8'h55
8'h39
Sending 8'h55 over the TX line
Sending 8'h39 over the TX line
Now
5004000 ps
Cursor 1
5008770 ps
1000000 ps
2000000 ps
3000000 ps
4000000 ps

Receive Unit Testbench

```
ModelSim> run -all
# testing receiving first data byte of oscillating 0s and 1s
# test passed
# testing receiving second data byte of 8'b0011_1001
# test passed
# ** Note: $stop    : I:/0school/ece554/miniproject1/receive_unit_tb.v(94)
#    Time: 4504 ns  Iteration: 0  Instance: /receive_unit_tb
# Break in Module receive_unit_tb at I:/0school/ece554/miniproject1/receive_unit_tb.v line 94
```


Msgs
/receive_unit_tb/clk
/receive_unit_tb/rst
/receive_unit_tb/iocs
/receive_unit_tb/iorw
/receive_unit_tb/rda
/receive_unit_tb/io...
/receive_unit_tb/rxd
/receive_unit_tb/da...
/receive_unit_tb/en...
1'h1
1'h0
1'h1
1'h1
1'h1
2'h0
1'h1
8'h39
1'h0
2'h0
8'h55
8'h39
Receiving 8'h55 over the RX line
Receiving 8'h39 over the RX line
Now
5004000 ps
1000000 ps
2000000 ps
3000000 ps
4000000 ps
5000000 ps

Driver Testbench

```
VSIM 17> run -all
# Testing baud rate 4800
#       test passed
# Testing baud rate 9600
#       test passed
# Testing baud rate 19200
#       test passed
# Testing baud rate 38400
#       test passed
# ** Note: $stop    : I:/0school/ece554/miniproject1/driver_tb.v(108)
#    Time: 186 ns  Iteration: 0  Instance: /driver_tb
# Break in Module driver_tb at I:/0school/ece554/miniproject1/driver_tb.v line 108
```