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 Hyperterminal 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.

<table>
<thead>
<tr>
<th>IOADDR</th>
<th>SPART Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Transmit Buffer (IOR/W = 0); Receive Buffer (IOR/W = 1)</td>
</tr>
<tr>
<td>01</td>
<td>Status Register (IOR/W = 1)</td>
</tr>
<tr>
<td>10</td>
<td>DB(Low) Division Buffer</td>
</tr>
<tr>
<td>11</td>
<td>DB(High) Division Buffer</td>
</tr>
</tbody>
</table>

Table 1: Address Mappings
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](image)

**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:

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

![Diagram of Baud Rate Generator](image-url)
The frequency of the occurrence of the most frequent enable, which has duration of one CLK period, is typically $2^n \times \text{baud rate}$, 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 achieved 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 $(\text{clock frequency}/(2^n \times \text{baud rate}) - 1)$. 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 $(\text{CLK}/(2^n \times \text{baud rate}) + 1) = \text{CLK}/(2^n \times \text{baud rate})$. 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 $(2^n \times \text{baud rate})$ 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 $(2^n \times \text{baud rate})$ where $n = 4$ and the baud rate is 9600 (bits/second). The divisor required is $25,000,000/(16 \times 9,600) - 1 = 161.76$ 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 $163 \times 40 \text{ ns} = 6.52 \mu \text{s}$. Inverting, the frequency is $153,374.23 \text{ Hz}$. 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.

<table>
<thead>
<tr>
<th>DIP Setting</th>
<th>Baud Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>4800</td>
</tr>
<tr>
<td>01</td>
<td>9600</td>
</tr>
<tr>
<td>10</td>
<td>19200</td>
</tr>
<tr>
<td>11</td>
<td>38400</td>
</tr>
</tbody>
</table>

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
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

```verilog
module top_level(
    input clk, // 100Mhz clock
    input reset, // Asynchronous reset, tied to dip switch 0
    output txd, // RS232 Transmit Data
    input rx, // RS232 Receive Data
    input [1:0] br_cfg // Baud Rate Configuration, Tied to dip switches 2 and 3
);

wire iocen; // wires the connect the spart to driver
wire iorw;
wire rds;
wire tbr;
wire [1:0] ioload;
wire [7:0] databus;

// Instantiate your SPART here
spart spart0(
    .clk(clk),
    .reset(reset),
    .iocen(iocen),
    .iorw(iorw),
    .rd(a),
    .tbr(tbr),
    .ioload(ioload),
    .databus(databus),
    .txd(txd),
    .rx(rxd)
);

// Instantiate your driver here
driver driver0(
    .clk(clk),
    .reset(reset),
    .br_cfg(br_cfg),
    .iocen(iocen),
    .iorw(iorw),
    . rd(a),
    .tbr(tbr),
    .ioload(ioload),
    .databus(databus)
);
endmodule
```
module apart {  // inputs and outputs
    input clk,
    input rst,
    input loca,
    input locw,
    output rch,
    output thr,
    output ino[8] ioaddr,
    output ino[8] idatabus,
    input txd,
    input rxd
};

wire [1:0] rate_tx_data;  // wires to connect submodules
wire [1:0] data_received;

module bus_interface(  // instantiate the DUT
    ioa(locas),
    ioaw(ioasw),
    rda(rda),
    tbr(tbr),
    ioaddr(ioadd),
    databus(databus),
    rate_tx_data(rate_tx_data),
    data_received(data_received)
);

module baud_rate_generator(  // instantiate ti:
    .clk(clk),
    .rat(rat),  // confirm? not shown is diagram, but we th:
    .ioa(locas),
    .ioaw(ioasw),
    .rda(rda),
    .tbr(tbr),
    .ioaddr(ioadd),
    .databus(databus),
    .rate_tx_data(rate_tx_data),
    .enable(enable)
);

module transmit_unit(  // instantiate the DUT
    .clk(clk),
    .rat(rat),
    .ioa(locas),
    .ioaw(ioasw),
    .rda(rda),
    .tbr(tbr),
    .ioaddr(ioadd),
    .txd(txd),
    .rate_tx_data(rate_tx_data),
    .enable(enable)
);

module receive_unit(  // instantiate the DUT
    .clk(clk),
    .rat(rat),
    .ioa(locas),
    .ioaw(ioasw),
    .rda(rda),
    .tbr(tbr),
    .ioaddr(ioadd),
    .rxd(rxd),
    .data_received(data_received),
    .enable(enable)
);
endmodule
module bus_interface:
  input iosw,
  input iosrv,
  input rds,
  input tbc,
  input [1:0] loadr,
  inout [1:0] daddr,
  output reg [1:0] rate_tx_data,
  input [1:0] data_received
);

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

wire [7:0] databus_out;
assign databus_out = {ioaddr[1]} ? '{8'b000000, tbc, rds} : data_received; // processor reading/SPIFF writing
assign databus = {iosrv || ioaddr[1]} ? '{8'hex : databus_out; // high impedance is required for processor writing/SPIFF reading

初选 begin // pseudocode for the assign statement
  // if (iosw || ioaddr[1]) begin
  //   databus = '{8'hex; // databus = '{8'hex; // databus = '{8'hex; // databus = '{8'hex; // databus = '{8'hex;
  // end else if (ioaddr[0]) begin // end else if (ioaddr[0]) begin // end else begin
  //   databus = '{8'hex; // databus = '{8'hex; // databus = '{8'hex; // databus = '{8'hex; // databus = '{8'hex;
  // end
  // end else begin // end else begin // end else begin // end else begin // end else begin
  //   databus = data_received; // databus = data_received; // databus = data_received; // databus = data_received; // databus = data_received;
  // end
  // end
endmodule
module bus_interface_tb();

    reg loc1; // signals that are connected to the DUT
    reg loc0;
    reg rsh;
    reg tsh;
    reg [15:0] lomdr;
    wire [15:0] databus;
    wire [15:0] rata_tx_data;
    wire [15:0] data_received;
    reg [7:0] databus_reg; // register for handling inputs
    assign databus = databus_reg;

    bus_interface DUT { // instantiate the DUT
        .lomdr(loc1),
        .lomdr(loc0),
        .rsh(rsh),
        .tsh(tsh),
        .lomdr(lomdr),
        .databus(databus),
        .rata_tx_data(rata_tx_data),
        .data_received(data_received)
    };

    initial begin
        $display("Testing bus_interface");
        loc1 = 1;
        loc0 = 1;
        lomdr = 2'h00;
        data_received = 6'h00;
        #5
        $display("Test processor writing to baud rate generator low");
        lomdr = 2'h01;
        databus_reg = 8'h02;
        #5
        if (rata_tx_data == databus_reg) begin
            $display("Test passed");
        end else begin
            $display("Test failed, expected: rate_t_data == databus_reg, rate_t_data = 8'h02, databus_reg = 8'h02", rata_tx_data, databus_reg);
        end
    end
display("Test processor writing a different value to baud rate generator high");

incadr = '2';
data_bus_reg = '8'h0d;

if (rate_tx_data == databus_reg) begin
    #
    display("Test processend: rate_tx_data == databus_reg");
end else begin
    #
    display("Test processend: rate_tx_data != databus_reg");
end

display("Test processor reading the status register dump");
data_bus_reg = '8'h0e;
thr = 1;
rdm = 1;
fetch = 1;

if (databus == '4'b000000, thr, rdm) begin
    #
    display("Test processend: databus == 8'h00, thr, rdm = 1");
end else begin
    #
    display("Test processend: databus != 8'h00, thr, rdm = 1");
end

display("Test processor writing a value to transmit controller");
incadr = 0;
data_bus_reg = '8'h0f;

if (rate_tx_data == databus_reg) begin
    #
    display("Test processend: rate tx data == databus_reg");
end else begin
    #
    display("Test processend: rate tx data != databus_reg");
end

display("Test processor reading a value from receive controller");
incadr = '2';
data_bus_reg = '8'h00;
data_received = 0;
data_bus_reg = '8'h1;

if (databus == data_received) begin
    #
    display("Test processend: databus == data_received");
end else begin
    #
    display("Test processend: databus != data_received");
end

end

endmodule
module baud_rate_generator;  // inputs and outputs
  input clk,  // for internal logic
  input rst,  // for internal logic
  input [15:0] ioc0addr,  // for internal logic
  input [12:0] rate_tx_data,  // for internal logic
  output reg [15:0] division_buffer;  // for internal logic
  reg [15:0] baud_rate_counter;  // for internal logic
endmodule  // for internal logic

always @(posedge clk) begin  // if reset, then enable
  if (rst) begin
    division_buffer <= 1;  // for internal logic
    baud_rate_counter <= 1;  // for internal logic
  end else if (ioc0addr == 2'b01) && (division_buffer[15:8] != rate_tx_data) begin
    division_buffer[15:8] <= rate_tx_data;  // for internal logic
    baud_rate_counter <= (division_buffer[15:8] == rate_tx_data);  // for internal logic
    if (division_buffer[7:0] <= rate_tx_data) begin
      division_buffer[7:0] <= rate_tx_data;  // for internal logic
      baud_rate_counter <= (division_buffer[15:0] == rate_tx_data);  // for internal logic
    end else if (baud_rate_counter == 0) begin  // for internal logic
      baud_rate_counter <= division_buffer;  // for internal logic
    end else begin
      baud_rate_counter <= baud_rate_counter - 1;  // for internal logic
    end
  end
end

always @(posedge clk) begin  // never enable if iocs is low
  if (ioc0) begin
    if (baud_rate_counter == 0) begin  // enable when counter reaches 0
      enable <= 1;  // for internal logic
    end else begin  // for internal logic
      enable <= 0;  // for internal logic
    end
  end
end module
Baud rate generator test bench

```verilog
module baud_rate_generator_tb();

reg clk;
reg rst;
reg [7:0] rate_txn_data;
wire enable;

baud_rate_generator #

initial begin
  // initialize all variables
  $display("Testing baud_rate_generator");
  clk = 0;
  rst = 1;
  rate_txn_data = 0;
  enable = 0;

  // test case 1: test first cycle of baud rate by setting only the low byte
  $display("Test first cycle of baud rate by setting only the low byte");
  icmdr = 2'H35;
  rate_txn_data = 0;
  $display("initial value loaded");
  rate_txn_data = 1;
  $display(" wait 1 cycle for baud rate to load");
  repeat (5) begin
    $display("wait 3 cycles when 3 is loaded");
    if (enable) begin
      $display("test failed, enable should have been kept low for 3 cycles");
      exit:
    end
  end
  $display("test another for baud_rate_generator to react to baud_rate_counter reaching 0");
  if (enable) begin
    $display("test passed");
    and
  end
  $display("test failed, enable should have been pulsed high on the 6th cycle");
end

// test case 2: test second cycle of baud rate by setting only the low byte
$display("Test second cycle of baud rate by setting only the low byte");
repeat (5) begin
  if (enable) begin
    $display("test failed, enable should have been kept low for 3 cycles");
  end
  exit:
end

// test case 3: test second cycle of baud rate by setting only the high byte
$display("Test second cycle of baud rate by setting only the high byte");
repeat (5) begin
  if (enable) begin
    $display("test failed, enable should have been kept low for 3 cycles");
  end
end

end
```
// test case 3: test first cycle of baud rate by setting only the high byte
$queadsy c1h$ $display("test first cycle of processor only writing to baud rate generator high");
$cmdir = 'Z'c1h$;
$rate_tx_data = 'Z'b80$;
$queadsy c1b$;
$cmdir = 'Z'c1b$;
$rate_tx_data = 'Z'$;
$queadsy c1b$
$repeat (919)$ begin // equivalent to 16'h0000
$queadsy c1h$
if (enable) begin
$display("test failed, enable should have been kept low for 8192 cycles");
end
$display("test passed");
if (enable) begin
$display("test failed, enable should have been pulsed high on the 8193rd cycle");
end
$display("test passed");
end

// test case 4: test second cycle of baud rate by setting only the high byte
$display("test second cycle of processor only writing to baud rate generator high");
$repeat (919)$ begin
$queadsy c1h$
if (enable) begin
$display("test failed, enable should have been kept low for 8192 cycles");
end
$display("test passed");
if (enable) begin
$display("test failed, enable should have been pulsed high");
end
end

// test case 5: test first cycle of baud rate by setting baud rate = 16'hba51
$queadsy c1h$ $display("test first cycle of processor writing 8'ba51 to baud rate generator");
$cmdir = 'Z'c1h$;
$rate_tx_data = 'Z'ba51$;
$queadsy c1b$;
$cmdir = 'Z'c1b$;
$rate_tx_data = 'Z'$;
$queadsy c1b$
$repeat (4199)$ begin // equivalent to 16'ha501
$queadsy c1h$
if (enable) begin
$display("test failed, enable should have been kept low for 41925 cycles");
end
$display("test passed");
if (enable) begin
$display("test failed, enable should have been pulsed high");
end
end

// test case 6: test second cycle of baud rate by setting baud rate = 16'ha551
$display("test second cycle of processor writing 8'ha551 to baud rate generator");
$repeat (4199)$ begin
$queadsy c1h$
if (enable) begin
$display("test failed, enable should have been kept low for 41925 cycles");
end
$display("test passed");
if (enable) begin
$display("test failed, enable should have been pulsed high");
end
end
// test case 1: test baud rate still works after changing the address  
  iocdr = 2'b00;  
  rate_meta_data = {2'h00};  
  always @ (posedge clkb) begin  
    if (enable) begin  
      $display("test failed, enable should have been kept low for 6289 cycles");  
    end  
    end  
  $display("test passed");  
  end  
end  
  $display("test failed, enable should have been pulsed high");  
end  
  
// test case 2: test that enable is never high when iocd is low  
  iocd = 0;  
  always @ (posedge clkb) begin  
    // equivalent to 16'hffff + 1  
    if (enable) begin  
      $display("test failed, enable should not go high if iocd is low");  
      $display("test passed");  
    end  
  end  
end  
always  
@ (posedge iocl)  
$ iocl = 1'b1;  
endmodule
Transmit unit

```verilog
module transmit_unit;  // inputs and outputs
  input clk,
  input rst,
  input iorw,
  input iorw,
  output reg tbr,
  input [1:0] load, shift, set_thr;
  reg [9:0] shift_reg;
  reg [2:0] bit_cnt;
  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_thr;
  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[7];
  // 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 <= '{7'b1, rate_tx_data, 1'b0};
      bit_cnt <= numberOfBitsPerPacket;
      end else if (set_thr) begin
        // set tx_data_out[0] to not start the receiver
        shift_reg[0] <= 4'b1;
      end else if (shift) begin
        bit_cnt <= bit_cnt - 1;
        shift_reg <= (shift_reg[1], shift_reg[9:1]);
      end
    end
  end
endmodule
```
localparam IDLE = 1'b0, TRANS = 1'b1;
reg state, nxt_state;

// handle start transitions of the PWM
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 (loadadr == 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) 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
    end
  endcase
end
endmodule
module transmit_unit_tb();
    reg clk; // signals that are connected to the DUT
    reg rst;
    reg ioccs;
    reg iorw;
    wire thr;
    reg [1:0] iocaddr;
    wire txd;
    reg [1:0] rate_tx_data;
    reg enable;

transmit_unit DUT( // instantiate the DUT
    .clk(clk),
    .reset(rst),
    .ioccs(ioccs),
    .iorw(iorw),
    .thr(thr),
    .ioca(iocaddr),
    .txd(txd),
    .rate_tx_data(rate_tx_data),
    .enable(enable)
);

initial begin // initialize all variables
$display("testing transmit_unit");
    clk = 0;
    rst = 1;
    ioccs = 1;
    iorw = 1;
    iocaddr = 0;
    rate_tx_data = 8'h00; // 8'b1010_1010 = 10'b11_0101_0100 = 10'h354
    enable = 0;
    @(posedge clk);
    rst = 0;
    iocaddr = 2'b00;
    $display("testing transmitting first data byte of oscillating 0s and 1s (8'h00)");
    iorw = 0;
    @(posedge clk);
    iorw = 1;
    while(thr != 1) begin // wait until done
        pulse_enable();
    end
end
```verilog
$display("testing transmitting second data byte of 8'b0011_1001");
rate_tx_data = 8'b0011_1001; // 10_0111_0010 = 272
iorw = 0;
#(posedge clk);
iorw = 1;
#(posedge clk);
while(cbr != 1) begin // wait until done
    pulse_enable();
end
#$stop;
end

task pulse_enable; // task to pulse enable
begin // so that receiver and transmitter can continue
    repeat (50) // wait some time
        #(posedge clk);
    enable = 1; // set enable high
    #1; // for 1 clock cycle
    enable = 0; // then reset enable
end
taskend
always
    #2 clk = !clk;
endmodule
```
module receive_unit(
    input clk,
    input rst,
    input loca,
    input lcrw,
    output reg rda,
    input [1:0] inaddr,
    output reg [7:0] data_received,
    input enable
);

reg [1:0] shift_reg;  // used to hold data
reg [8:1] bit_cnt;    // used to count number of bits to determine whether it is finished
reg load, shift, set_rda;  // FSM signals

localparam number_of_bits_per_packet = 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 <= number_of_bits_per_packet;
    end else if (shift) begin  // if shift, then rotate shift register and decrement bit counter
        bit_cnt <= bit_cnt - 1;
        shift_reg <= [data_received, shift_reg[7:1]];
    end else if (set_rda) begin  // if done, then move byte to output
        data_received <= shift_reg[7:1];
    end
end

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

// handle start transitions of the FSM
always @(posedge clk) begin  // always go to next state and set rda <= set_rda
    if (start) begin
        state <= IDLE;
        rda <= 0;
    end else begin
        state <= next_state;
        rda <= set_rda;
    end
end
always @(*) begin
  // set defaults
  next_state = IDLE;
  load = 0;
  shift = 0;
  set_rds = 0;
  case (state)
    IDLE: begin  // initial IDLE state, go to RECVR receiving and if saw a start bit
      if (loaddr == 2'bx00 & iscr & !rdw) begin
        load = 1;
        next_state = RECVR;
      end
    end
    RECVR: begin  // RECVR state to receive data, shift every time a bit comes in
      next_state = RECVR;
      // if enable, then shift
      if (enable) begin
        shift = 1;
        // if bit_cnt == 0, then done, hold data until it is read
        if (bit_cnt == 0) begin
          next_state = DONE;
        end
      end
    end
    default: begin  // same as DONE state to hold data
      // wait for transmit signal to be able to receive next byte of data
      next_state = DONE;
      set_rds = 1;
      if (loaddr == 2'bx00 & !low) begin
        next_state = IDLE;
      end
    end
  endcase
endmodule
Receive unit test bench

module receive_unit_tb();

  reg clk; // signals that are connected to the DUT
  reg rst;
  reg locc;
  reg icrv;
  wire rda;
  reg [1:0] laddr;
  reg rad;
  wire [63:0] data_received;
  reg enable;

receive_unit DUT( // instantiate the DUT
  .clk(clk),
  .rst(rst),
  .loc(locc),
  .icrv(icrv),
  .rda(rda),
  .laddr(laddr),
  .rad(rad),
  .data_received(data_received),
  .enable(enable));

initial begin // initialise all variables
  $display("testing receive unit");
  clk = 0;
  rda = 1;
  incr = 1;
  icrv = 0;
  laddr = 0;
  rad = 0;
  enable = 0;
  $posedge clk);
  rst = 0;
  $posedge clk);
  incr = 1;

while(clk'==1'b1) begin // oscillate XX to see 0010101 in cmd
  set_data();
  set_data();
  if(data_received == 8'h03) begin // check if equals 8'h03 since i sent it oscillating 0s and 1s
    $display("cor porre");
    end
    if(data_received == 8'h30) begin // check if equals 8'h30 since i sent it oscillating 0s and 1s
      $display("t fal se, expected data_received == 8'h30, actual data_received == 8'h30, data_received");
      end
  $display("testing receiving second data byte of 8'h0011_1001");
  rda = 1; // need to send 1 for 1 clock cycle for stop bit
  incr = 1; // make receiver available again
  }$
  $posedge clk);
  incr = 1;
  set_data(); // 1st bit, need to be 0 to start
  set_data(); // 2nd bit (least significant data bit)
  set_data(); // 3rd bit
  set_data(); // 4th bit
  set_data(); // 5th bit
  set_data(); // 6th bit
  set_data(); // 7th bit
  set_data(); // 8th bit
  set_data(); // need need to be 1 to stop
while (rda != 0) begin  // wait until done
  #;
end

if (data_received == 'b"0011_1001") begin // check if it matches the data I sent
  $display("test passed");
end else begin
  $display("test failed, expected data_received = 'b"0011_1001", actual data_received = 'b"", data_received");
end
end

task set_data;  // task for sending data
input data;
begin
  rd = data;
  #0;
  @(posedge clk);
  enable = 1;
  @(posedge clk);
  enable = 0;
end
endtask

always  // clock
  #2 clk = !clk;
endmodule
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 [7:0] rate_tx_data;
wire [7:0] data_received;
reg enable;

transmit_unit tDUT();   // instantiate the first 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 rDUT();     // instantiate the second DUT
    .clk(clk),
    .rst(rst),
    .iocs(iocs),
    .iorw(iorw),
    .rda(rda),
    .ioaddr(ioaddr),
    .rxd(rxd),
    .data_received(data_received),
    .enable(enable)
);
initial begin // initialize all variables

$display("testing transmit_unit and receive_unit");
clk = 0;
rat = 1;
lrcs = 1;
lrcw = 1;
loadbr = 1;
rate_tx_data = 8'b1010_1010 = 10'b1101_0100 = 10'b354
enable = 0;
$posedge clk);
rat = 0;
loadbr = 2'b00;

$display("testing transmitting first data byte of oscillating 0s and 1s (8'bhas)");
lrcw = 0;
$posedge clk);
lrcw = 1;
while(br != 1) begin // wait until done
  pulse_enables();
end
if (data_received == 8'bhas) begin
  $display("test passed");
end else begin
  $display("test failed, expected data_received = 8'bhas, actual data_received = 8'bhh", data_received);
end

$display("testing transmitting second data byte of 8'b0011_1001");
rat = 8'b0011_1001; // 0011_1001 = 272
lrcw = 0;
$posedge clk);
lrcw = 1;
$posedge clk);
while(lrcw != 1) begin // wait until done
  pulse_enables();
end
if (data_received == 8'b0011_1001) begin
  $display("test passed");
end else begin
  $display("test failed, expected data_received = 8'b0011_1001, actual data_received = 8'bhh", data_received);
end

reset;

// task to pulse enable signal

task pulse_enable;
begin
  repeat (10) // wait multiple cycle cycles
    $posedge clk);
    enable = 1; // set enable high
    $posedge clk);
    enable = 0; // reset enable
  end
endtask
always
  #2 clk = !clk;
endmodule
Driver

```verilog
module driver();

input clk,
input rst,
input [15:0] br_cfg,
output incr,
output en,
input dds,
input txr,
output req [11:0] loadSr,
output [7:0] databus;

assign locs = 1; // can't stop, won't stop
reg lo_rdy;
assign irw_rdy = irw_reg; // set to reg so that it can be set and keep its prototype
reg [11:0] inout_data;
reg [11:0] held_data;
reg update_baud, startSending_baud_hi, startSending_baud_lo, startReceiving, start_transmitting, receiving, transmitting;
reg [15:0] old_br_cfg;
localparam baud_4800 = 16'b0014; // divisor 4800 = 100000000/(16*600) - 1 = d130 = h034
localparam baud_9600 = 16'b0022; // divisor 9600 = 100000000/(16*600) - 1 = d65 = h02a
localparam baud_19200 = 16'b0045; // divisor 19200 = 100000000/(16*1200) - 1 = d65 = h02a
localparam baud_38400 = 16'b0092; // divisor 38400 = 100000000/(16*3000) - 1 = d162 = h062a
reg [19:0] new_baud;
localparam IDE  = 3'b000, BAUD1 = 3'b001, BAUD2 = 3'b010, BAUD3 = 3'b011, TRANS = 3'b100;
reg [23:1] state, stat_state;

// handle start transitions of the FSM
always @(posedge clk) begin
    if (rst) begin
        state <= IDE;
        old_br_cfg <= 3'b100;
    end else begin
        if (update_baud) begin
            old_br_cfg <= [1'b0, br_cfg];
            new_baud <= baud_4800; // case statement to load baud rate
            br_cfg <= 2'b0 $test_baud:
            br_cfg <= 2'b0 $test_baud;
            baud_4800: // default case
        end else if (startSending_baud_hi) begin // load baud rate high
            inout_data <= new_baud[15:1];
            loaddr <= [7:0] 3'b100;
        end else if (startSending_baud_lo) begin // load baud rate low
            inout_data <= new_baud[15:1];
            loaddr <= [7:0] 3'b100;
        end else if (startReceiving) begin // load available data
            inout_data <= databus;
        end else if (transmitting) begin // put available data back on inout bus
            if (receive) begin
                loaddr <= 3'b10;
                irw_reg <= 1;
            end else if (transmitting) begin
                loaddr <= 3'b10;
                irw_reg <= 0;
        end else begin
            irw_reg <= 0;
        end
        end

end
```

assign databus = (state == IDLE || state == RECV) ? 8'bzz : inout_data;

always @(*) begin
  // set defaults
  nxt_state = IDLE;
  update_baud = 0;
  start_sending_baud_hi = 0;
  start_sending_baud_lo = 0;
  start_receiving = 0;
  receiving = 0;
  start_transmitting = 0;
  transmitting = 0;
  case (state)
    IDLE: begin // send baud rate if uninitialized
      if (old_baud_cfg[1] || baud_cfg != old_baud_cfg[1:0]) begin
        update_baud = 1;
        nxt_state = BAUDRX;
      end else if (rda) begin // receive data if data is ready to be received
        start_receiving = 1;
        nxt_state = RECV;
      end
    end
    BAUDRX: begin // send baud rate high
      start_sending_baud_hi = 1;
      nxt_state = BAUDLO;
    end
    BAUDLO: begin // send baud rate low
      start_sending_baud_lo = 1;
      nxt_state = IDLE;
    end
    RECV: begin // stay until you can transmit the data back
      nxt_state = RECV;
      receiving = 1;
      if (trb) begin
        held_data = databus;
        start_transmitting = 1;
        nxt_state = TRANS;
      end
    end
    TRANS: begin // stay until done transmitting
      nxt_state = TRANS;
      transmitting = 1;
      if (trb) begin
        nxt_state = IDLE;
      end
    end
    default: begin
      // nothing, so return to IDLE
    end
  endcase
endmodule
Driver test bench

module driver_tb();

reg clk;  // signals that are connected to the DUT
reg rst;
reg [2:0] br_cfg;
wire locs;
wire irorw;
reg rda;
reg tbr;
wire [10:0] inaddr;
wire [7:0] databus;

reg[10:0] databus_reg;  // register for handling inouts

assign databus = databus_reg;

driver DUT(  // instantiate the DUT
  .clk(clk),
  .rst(rst),
  .br_cfg(br_cfg),
  .locs(locs),
  .iorw(ilorw),
  .rda(rda),
  .tbr(tbr),
  .inaddr(inaddr),
  .databus(databus)
);

initial begin  // initialize variables
  $display("Testing driver");
  clk = 0;
  rst = 1;
  br_cfg = 'h01;
  #(posedge clk);
  rst = 0;
  rda = 0;
  tbr = 0;
  databus_reg = 'h00;
  #(posedge clk);
  $display("Testing baud rate 4800");
  br_cfg = 'h00;  // wait for baud_rate to load
  #(posedge clk);
  if (DUT.new_baud == DUT.BAUD_4800) begin
    $display("Test passed");
  end else begin
    $display("Test failed. DUT.new_baud = 16'h00, DUT.BAUD_4800 = 16'h40");
  end
end

$display("Testing baud rate 9600");
br_cfg = 2'b01;
repeat ()
    @(posedge clk);
    if (DUT.new_baud == DUT.BAUD_9600) begin
        $display("ttest passed");
    end else begin
        $display("ttest failed, DUT.new_baud = 16'hhh, DUT.BAUD_4800 = 16'hhhh", DUT.new_baud, DUT.BAUD_9600);
    end
    $display("Testing baud rate 19200");
    br_cfg = 2'b10;
    repeat ()
        @(posedge clk);
        if (DUT.new_baud == DUT.BAUD_19200) begin
            $display("ttest passed");
        end else begin
            $display("ttest failed, DUT.new_baud = 16'hhh, DUT.BAUD_4800 = 16'hhhh", DUT.new_baud, DUT.BAUD_19200);
        end
        $display("Testing baud rate 38400");
        br_cfg = 2'b11;
        repeat ()
            @(posedge clk);
            if (DUT.new_baud == DUT.BAUD_38400) begin
                $display("ttest passed");
            end else begin
                $display("ttest failed, DUT.new_baud = 16'hhh, DUT.BAUD_4800 = 16'hhhh", DUT.new_baud, DUT.BAUD_38400);
            end
        end
    end
always
    #100;
    $stop;
endmodule

endmodule
Baud Rate Generator Testbench

Baud rate is being set to 16'hA531 (42289)

Enable is set to high after 16'hA531 cycles
Bus Interface Testbench
VSM 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:/Oschool/ece554/miniproject1/bus_interface_tb.v(114)
#    Time: 50 ns  Iteration: 0  Instance: /bus_interface_tb
# Break in Module bus_interface_tb at I:/Oschool/ece554/miniproject1/bus_interface_tb.v line 114
Transmit Unit Testbench

<table>
<thead>
<tr>
<th>SIM&gt;</th>
<th>run -all</th>
</tr>
</thead>
<tbody>
<tr>
<td># testing transmitting first data byte of oscillating 0s and 1s (8'haa)</td>
<td></td>
</tr>
<tr>
<td># testing transmitting second data byte of 8'b0011_1001</td>
<td></td>
</tr>
<tr>
<td># ** Note: @stop : I:/0school/ece554/miniproject1/transmit_unit_tb.v(74)</td>
<td></td>
</tr>
<tr>
<td>#  Time: 2258 ns Iteration: 1 Instance: /transmit_unit_tb</td>
<td></td>
</tr>
<tr>
<td># Break at I:/0school/ece554/miniproject1/transmit_unit_tb.v line 74</td>
<td></td>
</tr>
</tbody>
</table>
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
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