Codes : Module for UART Transmitter.

`timescale 1ns / 1ps

module Tx\_UART

#(parameter CLKS\_PER\_BIT= 87)

(input i\_Clock,

input i\_Tx\_DV,

input [7:0] i\_Tx\_Byte,

output o\_Tx\_Active,

output reg o\_Tx\_Serial,

output o\_Tx\_Done

);

parameter s\_IDLE = 3'b000;

parameter s\_TX\_START\_BIT = 3'b001;

parameter s\_TX\_DATA\_BITS = 3'b010;

parameter s\_TX\_STOP\_BIT = 3'b011;

parameter s\_CLEANUP = 3'b100;

reg [2:0] r\_SM\_Main = 0;

reg [7:0] r\_Clock\_Count = 0;

reg [2:0] r\_Bit\_Index = 0;

reg [7:0] r\_Tx\_Data = 0;

reg r\_Tx\_Done = 0;

reg r\_Tx\_Active = 0;

always @(posedge i\_Clock)

begin

case (r\_SM\_Main)

s\_IDLE :

begin

o\_Tx\_Serial <= 1'b1; // Drive Line High for Idle

r\_Tx\_Done <= 1'b0;

r\_Clock\_Count <= 0;

r\_Bit\_Index <= 0;

if (i\_Tx\_DV == 1'b1)

begin

r\_Tx\_Active <= 1'b1;

r\_Tx\_Data <= i\_Tx\_Byte;

r\_SM\_Main <= s\_TX\_START\_BIT;

end

else

r\_SM\_Main <= s\_IDLE;

end // case: s\_IDLE

// Send out Start Bit. Start bit = 0

s\_TX\_START\_BIT :

begin

o\_Tx\_Serial <= 1'b0;

// Wait CLKS\_PER\_BIT-1 clock cycles for start bit to finish

if (r\_Clock\_Count < CLKS\_PER\_BIT-1)

begin

r\_Clock\_Count <= r\_Clock\_Count + 1;

r\_SM\_Main <= s\_TX\_START\_BIT;

end

else

begin

r\_Clock\_Count <= 0;

r\_SM\_Main <= s\_TX\_DATA\_BITS;

end

end // case: s\_TX\_START\_BIT

// Wait CLKS\_PER\_BIT-1 clock cycles for data bits to finish

s\_TX\_DATA\_BITS :

begin

o\_Tx\_Serial <= r\_Tx\_Data[r\_Bit\_Index];

if (r\_Clock\_Count < CLKS\_PER\_BIT-1)

begin

r\_Clock\_Count <= r\_Clock\_Count + 1;

r\_SM\_Main <= s\_TX\_DATA\_BITS;

end

else

begin

r\_Clock\_Count <= 0;

// Check if we have sent out all bits

if (r\_Bit\_Index < 7)

begin

r\_Bit\_Index <= r\_Bit\_Index + 1;

r\_SM\_Main <= s\_TX\_DATA\_BITS;

end

else

begin

r\_Bit\_Index <= 0;

r\_SM\_Main <= s\_TX\_STOP\_BIT;

end

end

end // case: s\_TX\_DATA\_BITS

// Send out Stop bit. Stop bit = 1

s\_TX\_STOP\_BIT :

begin

o\_Tx\_Serial <= 1'b1;

// Wait CLKS\_PER\_BIT-1 clock cycles for Stop bit to finish

if (r\_Clock\_Count < CLKS\_PER\_BIT-1)

begin

r\_Clock\_Count <= r\_Clock\_Count + 1;

r\_SM\_Main <= s\_TX\_STOP\_BIT;

end

else

begin

r\_Tx\_Done <= 1'b1;

r\_Clock\_Count <= 0;

r\_SM\_Main <= s\_CLEANUP;

r\_Tx\_Active <= 1'b0;

end

end // case: s\_Tx\_STOP\_BIT

// Stay here 1 clock

s\_CLEANUP :

begin

r\_Tx\_Done <= 1'b1;

r\_SM\_Main <= s\_IDLE;

end

default :

r\_SM\_Main <= s\_IDLE;

endcase

end

assign o\_Tx\_Active = r\_Tx\_Active;

assign o\_Tx\_Done = r\_Tx\_Done;

endmodule

Codes: Module for UART Reciever

module Rx\_UART

#(parameter CLKS\_PER\_BIT = 87)

(

input i\_Clock,

input i\_Rx\_Serial,

output o\_Rx\_DV,

output [7:0] o\_Rx\_Byte

);

parameter s\_IDLE = 3'b000;

parameter s\_RX\_START\_BIT = 3'b001;

parameter s\_RX\_DATA\_BITS = 3'b010;

parameter s\_RX\_STOP\_BIT = 3'b011;

parameter s\_CLEANUP = 3'b100;

reg r\_Rx\_Data\_R = 1'b1;

reg r\_Rx\_Data = 1'b1;

reg [7:0] r\_Clock\_Count = 0;

reg [2:0] r\_Bit\_Index = 0; //8 bits total

reg [7:0] r\_Rx\_Byte = 0;

reg r\_Rx\_DV = 0;

reg [2:0] r\_SM\_Main = 0;

// Purpose: Double-register the incoming data.

// This allows it to be used in the UART RX Clock Domain.

// (It removes problems caused by metastability)

always @(posedge i\_Clock)

begin

r\_Rx\_Data\_R <= i\_Rx\_Serial;

r\_Rx\_Data <= r\_Rx\_Data\_R;

end

// Purpose: Control RX state machine

always @(posedge i\_Clock)

begin

case (r\_SM\_Main)

s\_IDLE :

begin

r\_Rx\_DV <= 1'b0;

r\_Clock\_Count <= 0;

r\_Bit\_Index <= 0;

if (r\_Rx\_Data == 1'b0) // Start bit detected

r\_SM\_Main <= s\_RX\_START\_BIT;

else

r\_SM\_Main <= s\_IDLE;

end

// Check middle of start bit to make sure it's still low

s\_RX\_START\_BIT :

begin

if (r\_Clock\_Count == (CLKS\_PER\_BIT-1)/2)

begin

if (r\_Rx\_Data == 1'b0)

begin

r\_Clock\_Count <= 0; // reset counter, found the middle

r\_SM\_Main <= s\_RX\_DATA\_BITS;

end

else

r\_SM\_Main <= s\_IDLE;

end

else

begin

r\_Clock\_Count <= r\_Clock\_Count + 1;

r\_SM\_Main <= s\_RX\_START\_BIT;

end

end // case: s\_RX\_START\_BIT

// Wait CLKS\_PER\_BIT-1 clock cycles to sample serial data

s\_RX\_DATA\_BITS :

begin

if (r\_Clock\_Count < CLKS\_PER\_BIT-1)

begin

r\_Clock\_Count <= r\_Clock\_Count + 1;

r\_SM\_Main <= s\_RX\_DATA\_BITS;

end

else

begin

r\_Clock\_Count <= 0;

r\_Rx\_Byte[r\_Bit\_Index] <= r\_Rx\_Data;

// Check if we have received all bits

if (r\_Bit\_Index < 7)

begin

r\_Bit\_Index <= r\_Bit\_Index + 1;

r\_SM\_Main <= s\_RX\_DATA\_BITS;

end

else

begin

r\_Bit\_Index <= 0;

r\_SM\_Main <= s\_RX\_STOP\_BIT;

end

end

end // case: s\_RX\_DATA\_BITS

// Receive Stop bit. Stop bit = 1

s\_RX\_STOP\_BIT :

begin

// Wait CLKS\_PER\_BIT-1 clock cycles for Stop bit to finish

if (r\_Clock\_Count < CLKS\_PER\_BIT-1)

begin

r\_Clock\_Count <= r\_Clock\_Count + 1;

r\_SM\_Main <= s\_RX\_STOP\_BIT;

end

else

begin

r\_Rx\_DV <= 1'b1;

r\_Clock\_Count <= 0;

r\_SM\_Main <= s\_CLEANUP;

end

end // case: s\_RX\_STOP\_BIT

// Stay here 1 clock

s\_CLEANUP :

begin

r\_SM\_Main <= s\_IDLE;

r\_Rx\_DV <= 1'b0;

end

default :

r\_SM\_Main <= s\_IDLE;

endcase

end

assign o\_Rx\_DV = r\_Rx\_DV;

assign o\_Rx\_Byte = r\_Rx\_Byte;

endmodule // uart\_rx

**CODES: Testbench (**

This testbench will exercise both the UART Tx and RX.

module Tx\_TB();

// Testbench uses a 10 MHz clock

// Want to interface to 115200 baud UART

// 10000000 / 115200 = 87 Clocks Per Bit.

parameter c\_CLOCK\_PERIOD\_NS = 100;

parameter c\_CLKS\_PER\_BIT = 87;

parameter c\_BIT\_PERIOD = 8600;

reg r\_Clock = 0;

reg r\_Tx\_DV = 0;

wire w\_Tx\_Done;

reg [7:0] r\_Tx\_Byte = 0;

reg r\_Rx\_Serial = 1;

wire [7:0] w\_Rx\_Byte;

// Takes in input byte and serializes it

task UART\_WRITE\_BYTE;

input [7:0] i\_Data;

integer ii;

begin

// Send Start Bit

r\_Rx\_Serial <= 1'b0;

#(c\_BIT\_PERIOD);

#1000;

// Send Data Byte

for (ii=0; ii<8; ii=ii+1)

begin

r\_Rx\_Serial <= i\_Data[ii];

#(c\_BIT\_PERIOD);

end

// Send Stop Bit

r\_Rx\_Serial <= 1'b1;

#(c\_BIT\_PERIOD);

end

endtask // UART\_WRITE\_BYTE

uart\_rx #(.CLKS\_PER\_BIT(c\_CLKS\_PER\_BIT)) UART\_RX\_INST

(.i\_Clock(r\_Clock),

.i\_Rx\_Serial(r\_Rx\_Serial),

.o\_Rx\_DV(),

.o\_Rx\_Byte(w\_Rx\_Byte)

);

uart\_tx #(.CLKS\_PER\_BIT(c\_CLKS\_PER\_BIT)) UART\_TX\_INST

(.i\_Clock(r\_Clock),

.i\_Tx\_DV(r\_Tx\_DV),

.i\_Tx\_Byte(r\_Tx\_Byte),

.o\_Tx\_Active(),

.o\_Tx\_Serial(),

.o\_Tx\_Done(w\_Tx\_Done)

);

always

#(c\_CLOCK\_PERIOD\_NS/2) r\_Clock <= !r\_Clock;

// Main Testing:

initial

begin

// Tell UART to send a command (exercise Tx)

@(posedge r\_Clock);

@(posedge r\_Clock);

r\_Tx\_DV <= 1'b1;

r\_Tx\_Byte <= 8'hAB;

@(posedge r\_Clock);

r\_Tx\_DV <= 1'b0;

@(posedge w\_Tx\_Done);

// Send a command to the UART (exercise Rx)

@(posedge r\_Clock);

UART\_WRITE\_BYTE(8'h3F);

@(posedge r\_Clock);

// Check that the correct command was received

if (w\_Rx\_Byte == 8'h3F)

$display("Test Passed - Correct Byte Received");

else

$display("Test Failed - Incorrect Byte Received");

end

endmodule