**MINI PROJECT**

**Controller Area Network Transmission Module Using Verilog HDL**

**DESIGN MODULE FOR THE CAN TRANSMISSION:**

// Code your design here

module can\_tx(

output reg tx,

output reg can\_bitstuff,

output reg txing,

input rx,

input[10:0] address,

input clk,

input baud\_clk,

input rst,

input [7:0] data,

input send\_data,

input bitstuffed\_output,

input clear\_to\_tx

);

assign rx\_buf = rx;

parameter all\_ones = 15'b111111111111111;

parameter idle = 8'h0, start\_of\_frame = 8'h1, addressing =8'h2 ,rtr = 8'h3 ,ide = 8'h4, reserve\_bit = 8'h5, num\_of\_bytes = 8'h6,

data\_out = 8'h7, crc\_out = 8'h8, crc\_delimiter = 8'h9 , ack = 8'hA, ack\_delimiter = 8'hB, end\_of\_frame = 8'hC, waiting = 8'hD;

parameter bytes = 5'd8;

reg[10:0] address\_count = 0, crc\_count = 0, eof\_count = 0 , data\_bit\_count = 0, data\_byte\_count = 0;

reg[7:0] c\_state=0, n\_state=0;

initial txing = 0;

reg[14:0] crc\_output, crc\_holder;

wire one\_shotted\_send;

wire[14:0] crc\_buff;

CRC cyclic\_red\_check(data, one\_shotted\_send, crc\_buff,rst,clk);//caliing crc module

OneShot os(send\_data, clk, rst, one\_shotted\_send);//calling one shot pulse module

always @(crc\_buff or crc\_holder) begin

if(crc\_buff != all\_ones)

crc\_output <= crc\_buff;

else

crc\_output <= crc\_holder;

end

always @ (posedge clk or posedge rst) begin

if(rst == 1) begin

crc\_holder <= 15'd0;

end

else begin

crc\_holder <= crc\_output;

end

end

//Update Logic

always @ (posedge baud\_clk or posedge rst) begin

if(rst == 1) begin

c\_state <= 32'd0;

end

else begin

c\_state <= n\_state;

end

end

//Counting Logic

always @ (posedge baud\_clk) begin

case(c\_state)

idle: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

waiting: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

start\_of\_frame:begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

addressing: begin

address\_count <= address\_count + 1'b1;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

rtr: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

ide: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

reserve\_bit: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

num\_of\_bytes: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= data\_byte\_count +1'b1;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

data\_out: begin

address\_count <= 11'd0;

data\_bit\_count<= data\_bit\_count +1'b1;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

crc\_out: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= crc\_count + 1'b1;

eof\_count <= 11'd0;

end

crc\_delimiter: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

ack: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

ack\_delimiter:begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

end\_of\_frame: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= eof\_count +1'b1;

end

default: begin

address\_count <= 11'd0;

data\_bit\_count<= 11'd0;

data\_byte\_count<= 11'd0;

crc\_count <= 11'd0;

eof\_count <= 11'd0;

end

endcase

end

//Next State Logic

always @ (c\_state or rx\_buf or data or send\_data or address\_count or bitstuffed\_output or data\_byte\_count

or data\_bit\_count or crc\_count or eof\_count or clear\_to\_tx or crc\_output) begin

case(c\_state)

idle: begin

if(send\_data && clear\_to\_tx) begin

n\_state <= start\_of\_frame;

end

else begin

n\_state <= idle;

end

end

start\_of\_frame: begin

if(!rx\_buf) begin

n\_state <= addressing;

end

else begin

n\_state <= waiting;

end

end

waiting: begin

if(send\_data && clear\_to\_tx) begin

n\_state <= start\_of\_frame;

end

else begin

n\_state <= waiting;

end

end

addressing: begin

if(rx\_buf != bitstuffed\_output) begin

n\_state <= waiting; //Lost Arbitration

end

else if(address\_count == 11'd10) begin

n\_state <= rtr;

end

else begin

n\_state <= addressing;

end

end

rtr: begin

n\_state <= ide;

end

ide: begin

n\_state <= reserve\_bit;

end

reserve\_bit: begin

n\_state <= num\_of\_bytes;

end

num\_of\_bytes: begin

if(data\_byte\_count == 11'd3) begin

n\_state <= data\_out;

end

else begin

n\_state <= num\_of\_bytes;

end

end

data\_out: begin

if(data\_bit\_count == 11'd63) begin

n\_state <= crc\_out;

end

else begin

n\_state <= data\_out;

end

end

crc\_out: begin

if(crc\_count == 11'd14) begin

n\_state <= crc\_delimiter;

end

else begin

n\_state <= crc\_out;

end

end

crc\_delimiter: begin

n\_state <= ack;

end

ack: begin

n\_state <= ack\_delimiter;

end

ack\_delimiter: begin

n\_state <= end\_of\_frame;

end

end\_of\_frame: begin

if(eof\_count == 11'd6) begin

n\_state <= idle;

end

else begin

n\_state <= end\_of\_frame;

end

end

default:

begin

n\_state <= idle;

end

endcase

end

//Output Logic

always @(c\_state or address or data or crc\_output or crc\_count or data\_byte\_count or data\_bit\_count or address\_count) begin

case(c\_state)

idle: begin

tx <= 1;

can\_bitstuff <= 0;

txing <= 1'b0;

end

addressing: begin

tx <= address[11'd10-address\_count];

can\_bitstuff <= 1;

txing <= 1'b1;

end

start\_of\_frame: begin

tx<= 0;

can\_bitstuff <= 1'b0;

txing <= 1'b1;

end

rtr: begin

tx <= 0;

can\_bitstuff <= 1;

txing <= 1'b1;

end

ide: begin

tx <= 0;

can\_bitstuff <= 1;

txing <= 1'b1;

end

reserve\_bit: begin

tx <= 0;

can\_bitstuff <= 1;

txing <= 1'b1;

end

num\_of\_bytes: begin

tx <= bytes[11'd3-data\_byte\_count];

can\_bitstuff <= 1;

txing <= 1'b1;

end

data\_out: begin

tx <= data[11'd63-data\_bit\_count];

can\_bitstuff <= 1;

txing <= 1'b1;

end

crc\_out: begin

tx <= crc\_output[11'd14-crc\_count];

can\_bitstuff <= 1;

txing <= 1'b1;

end

crc\_delimiter: begin

tx <= 1;

can\_bitstuff <= 0;

txing <= 1'b1;

end

ack: begin

tx <= 1;

can\_bitstuff <= 0;

txing <= 1'b1;

end

ack\_delimiter:begin

tx <= 1;

can\_bitstuff <= 0;

txing <= 1'b1;

end

end\_of\_frame: begin

tx <= 1;

can\_bitstuff <= 0;

txing <= 1'b1;

end

waiting: begin

tx <= 1;

can\_bitstuff <= 0;

txing <= 1'b0;

end

default: begin

tx <= 1;

can\_bitstuff <= 0;

txing <= 1'b1;

end

endcase

end

endmodule

module CRC (

input [7:0] data\_in,

input crc\_en,

output reg [4:0] crc\_out,

input clk,

input reset

);

reg [4:0] lfsr\_q, lfsr\_c;

always @\* begin

lfsr\_c[0] = data\_in[7] ^ data\_in[5] ^ data\_in[4] ^ data\_in[3] ^ data\_in[0] ^ lfsr\_q[4] ^ lfsr\_q[3] ^ lfsr\_q[2] ^ lfsr\_q[0];

lfsr\_c[1] = data\_in[7] ^ data\_in[6] ^ data\_in[5] ^ data\_in[2] ^ data\_in[0] ^ lfsr\_q[4] ^ lfsr\_q[3] ^ lfsr\_q[1];

lfsr\_c[2] = data\_in[6] ^ data\_in[5] ^ data\_in[4] ^ data\_in[1] ^ data\_in[0] ^ lfsr\_q[4] ^ lfsr\_q[2];

lfsr\_c[3] = data\_in[7] ^ data\_in[5] ^ data\_in[4] ^ data\_in[3] ^ data\_in[1] ^ lfsr\_q[3] ^ lfsr\_q[1] ^ lfsr\_q[0];

lfsr\_c[4] = data\_in[6] ^ data\_in[5] ^ data\_in[2] ^ data\_in[1] ^ lfsr\_q[4] ^ lfsr\_q[2] ^ lfsr\_q[1] ^ lfsr\_q[0];

end

always @(posedge clk, posedge reset) begin

if (reset)

lfsr\_q <= 0;

else

lfsr\_q <= lfsr\_c;

end

always @(posedge clk) begin

crc\_out <= lfsr\_q;

end

endmodule

module OneShot(

input pulse,

input clk,

input rst,

output reg out

);

initial out = 0;

parameter waiting\_l = 2'b00, on = 2'b01, waiting\_h = 2'b10;

reg[1:0] next\_state, current\_state;

always @ (posedge clk or posedge rst) begin

if(rst) begin

current\_state <= waiting\_l;

end

else begin

current\_state <= next\_state;

end

end

always @ (current\_state or pulse) begin

if(current\_state == on) begin

next\_state <= waiting\_h;

end

else if(current\_state == waiting\_h) begin

if(pulse) begin

next\_state <= waiting\_h;

end

else begin

next\_state <= waiting\_l;

end

end

else if(pulse) begin

next\_state<= on;

end

else begin

next\_state<= waiting\_l;

end

end

always @(current\_state or rst) begin

if(rst)

out <= 1'b0;

else if(current\_state == on)

out <= 1'b1;

else

out <= 1'b0;

end

endmodule