ELEC373
Digital Systems Design
Assignment 2
Serial IrDA Communications
Design Report

Name: Junhao Zhang
Student ID: 201377244

Content

1 Introduction

A Universal Asynchronous Receiver and Transmitter (UART) is a physical circuit in a microcontroller which is mainly used to transmit and receiver serial data. Two UARTs can communicate directly with each other in UART communication. The transmitter converts parallel data into serial, and send it through data bus to the receiver, where data are converted back to parallel form. The advantage for a UART is that the transmission of data between two UARTs devices requires only two wires, or rather, the data bus where data flows out of Tx pin of the transmitter, and data flows into Rx pin of the receiver.

In this assignment, it is required to design a IrDA communication system where data can be transmitted or received the between two DE2 boards via the on-board IrDA. Therefore, it is necessary to encode the UART data to proper IrDA data for transmitting, and also decode the IrDA data back to UART data for receiving.

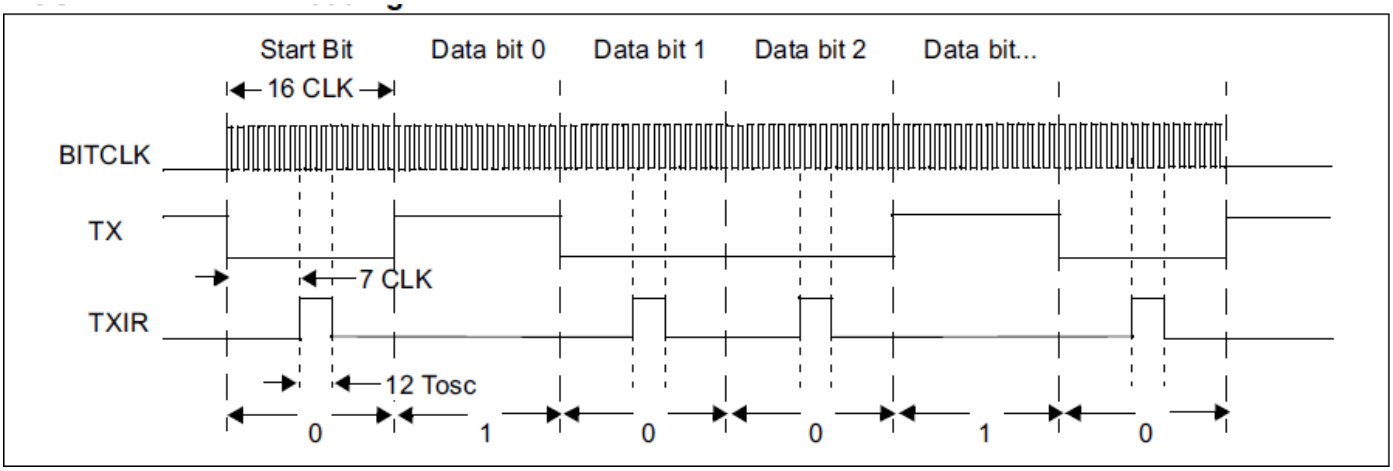

Figure 1. Example of Encoding TX to TXIR

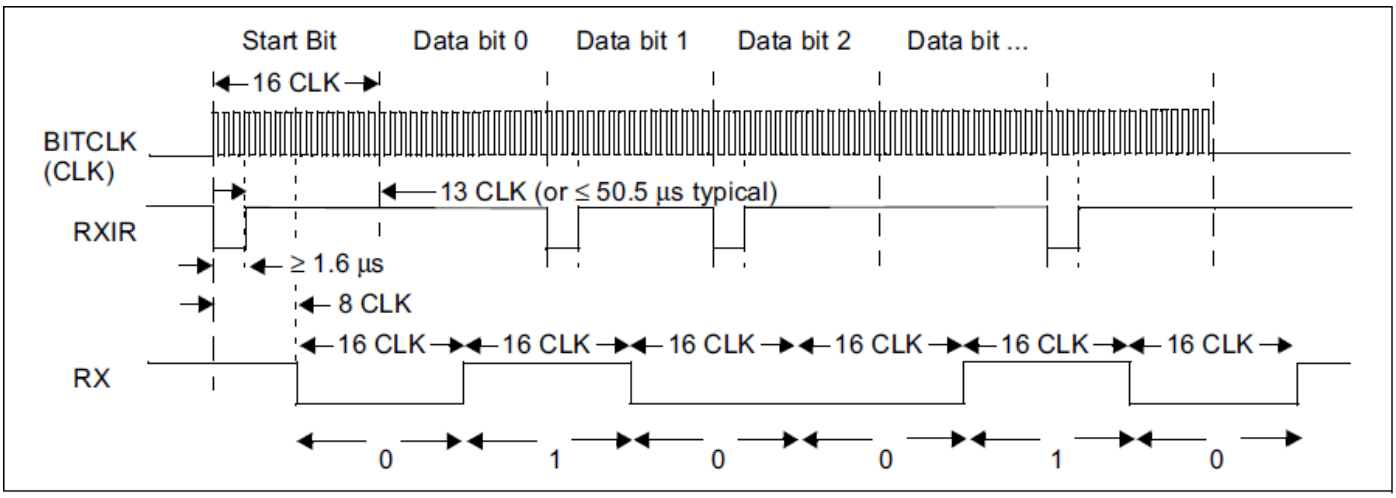

Figure 2. Example of Decoding RXIR to RX

According to Table 1, the total number of bits in a transmission of data bits is supposed to be: 10 bit = 1 start bit + 7 data bits + 1 parity bit + 1 stop bit. Table 2 below showcases an example transmission of character 'A'.

Table 1. Data Frame Transmitting Character 'A'

Stop bit	Parity bit	Data bits	Start bit
1	1	1000001	0

2 Top Level Design

2.1 IrDA Transceiver

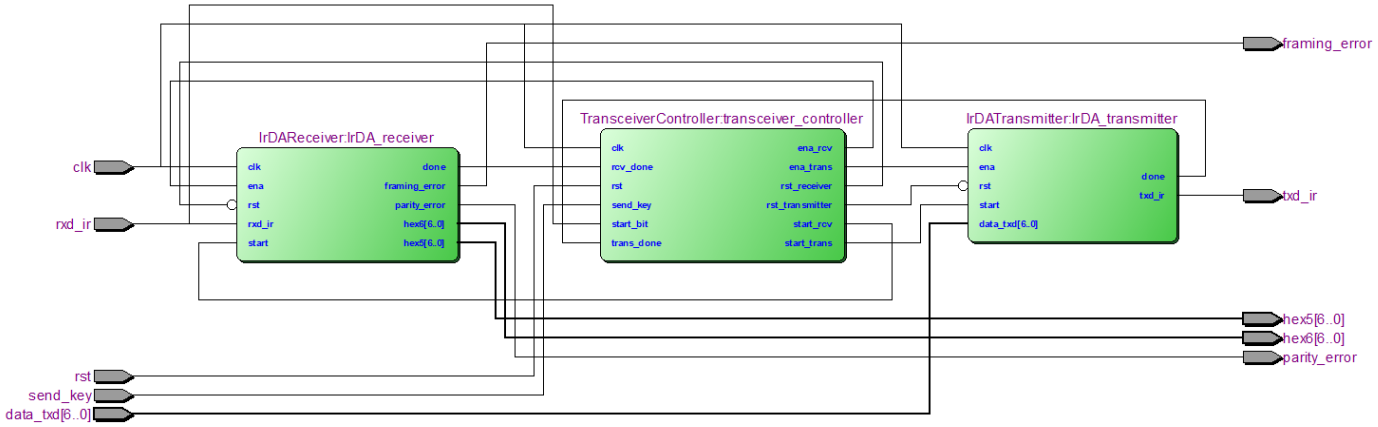

Figure 3. Transceiver Architecture

Figure 1 showcases the interconnection amongst the transmitter, the receiver and the controller. The transceiver controller is used to control the process of data transmission and reception. When a set of data is on progress of transmission, the controller will enable the transmitter only and disable the receiver, and vice versa. The done signal from the transmitter and the receiver will indicate the complete of the transmission and the reception.

2.1.1 Transceiver Controller

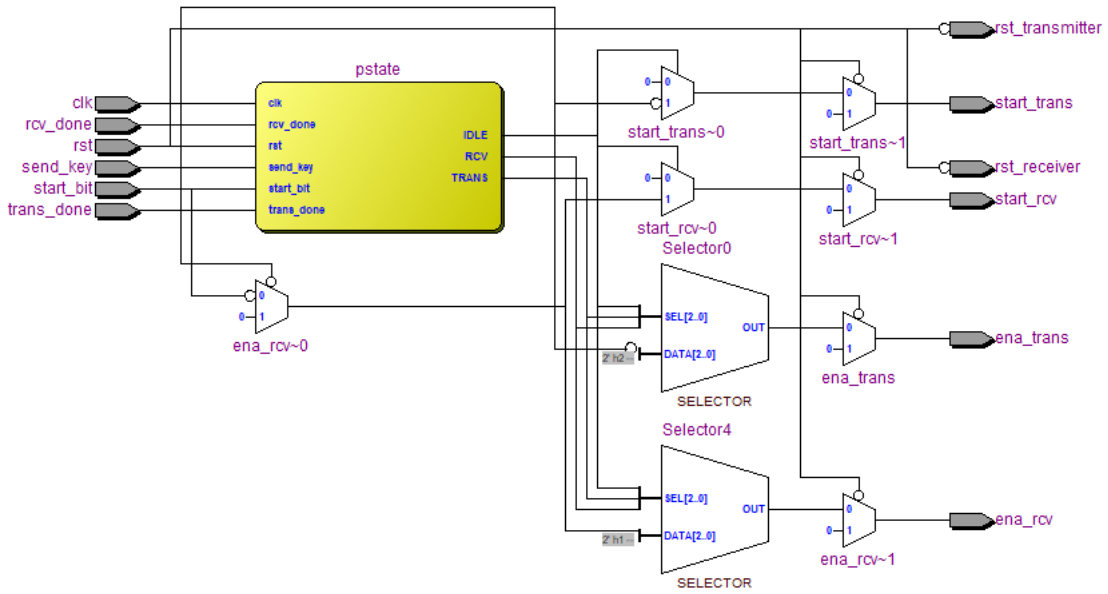

Figure 4. Transceiver Controller RTL View

The controller of the transceiver control the transmitter and the receiver so that both of them can work without conflict, the following table demonstrates the functions of the inputs and the outputs of the transceiver controller.

Table 2. Signal Functions and Active States

Input/Output	Signal	Function	Active State
Input	rst	Connected to the reset key to reset the whole transceiver	Active high
Input	send_key	Connected to the send key	Active low
Input	start_bit	Connected to the IrDA rxd input	Active low
Input	trans_done	Connected to the transmitter module to detect whether the transmission is done	Active high
Input	rcv_done	Connected to the receiver module to detect whether the receive is done	Active high

Output	rst_transmitter	Reset the transmitter	Active low
Output	rst_receiver	Reset the receiver	Active low
Output	start_rcv	Start the receiver	Active high
Output	start_trans	Start the transmitter	Active high
Output	ena_rcv	Enable the receiver	Active high
Output	ena_trans	Enable the transmitter	Active high

ASM Chart

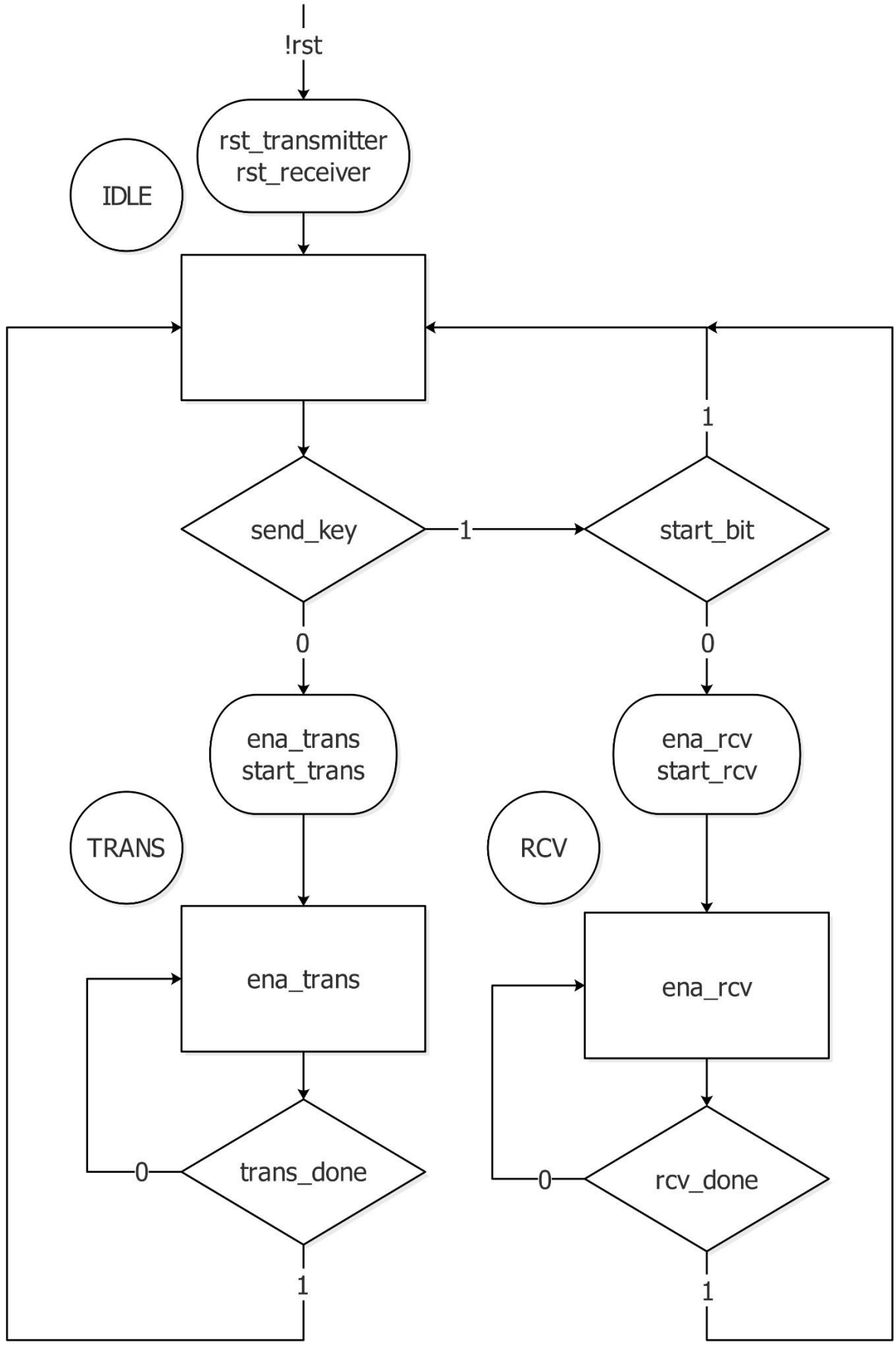

Figure 5. Transceiver Controller ASM

State Transition Diagram

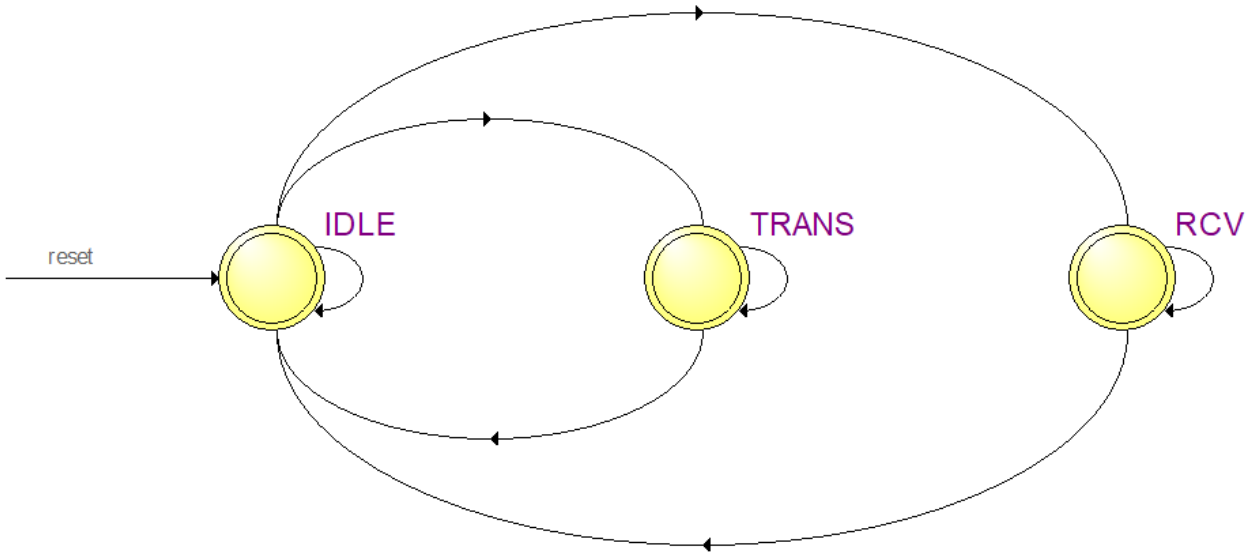

Figure 6. Transceiver Controller State Transition Diagram

	Source State	Destination State	Condition
1	IDLE	TRANS	(!send_key).(rst)
2	IDLE	IDLE	(!start_bit).(!rst) + (start_bit).(!send_key).(!rst) + (start_bit).(send_key)
3	IDLE	RCV	(!start_bit).(send_key).(rst)
4	RCV	IDLE	(!rcv_done).(!rst) + (rcv_done)
5	RCV	RCV	(!rcv_done).(rst)
6	TRANS	TRANS	(!trans_done).(rst)
7	TRANS	IDLE	(!trans_done).(!rst) + (trans_done)

Figure 7. State Table

Verilog Code

```
module TransceiverController
(
      input        clk,
      input        rst,
      input        send_key,
      input        start_bit,
      input        trans_done,
      input        rcv_done,
      output reg   ena_trans,
      output reg   ena_rcv,
      output reg   start_trans,
      output reg   start_rcv,
```

```
      output reg   rst_transmitter,
      output reg   rst_receiver
);

parameter IDLE     = 2'b00;
parameter TRANS    = 2'b01;
parameter RCV      = 2'b11;

reg [1:0] pstate, nstate;

always @(posedge clk)
begin
      pstate <= nstate;
end

always @(pstate, rst, send_key, start_bit, trans_done, rcv_done)
begin
      ena_trans          = 1'b0;
      ena_rcv            = 1'b0;
      start_trans        = 1'b0;
      start_rcv          = 1'b0;
      rst_transmitter    = 1'b0;
      rst_receiver       = 1'b0;
      nstate             = pstate;

      if(!rst) begin
            rst_transmitter    = 1'b1;
            rst_receiver       = 1'b1;
            nstate             = IDLE;
      end
      else begin
            case(pstate)
                  IDLE: begin
                        if(send_key == 0) begin
                              ena_trans    = 1'b1;
                              start_trans  = 1'b1;
                              nstate       = TRANS;
                        end
                        else if(start_bit == 0) begin
                              ena_rcv      = 1'b1;
                              start_rcv    = 1'b1;
                              nstate       = RCV;
                        end
                  end

                  TRANS: begin
                        ena_trans = 1'b1;

                        if(trans_done)
                              nstate = IDLE;
                  end

                  RCV: begin
                        ena_rcv = 1'b1;

                        if(rcv_done)
                              nstate = IDLE;
                  end

                  default: begin
                        nstate = IDLE;
                  end
            endcase
      end
end
endmodule
```

Simulation Results

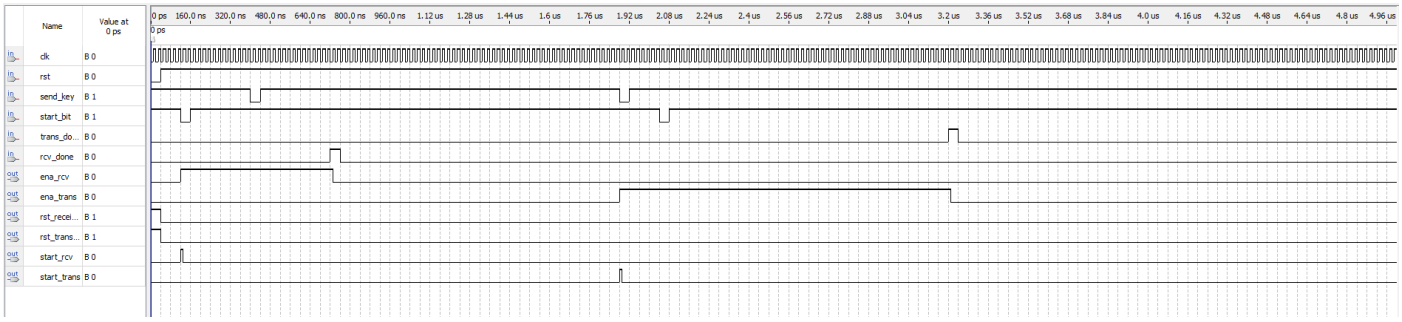

Figure 8. Transceiver Controller Functional Simulation

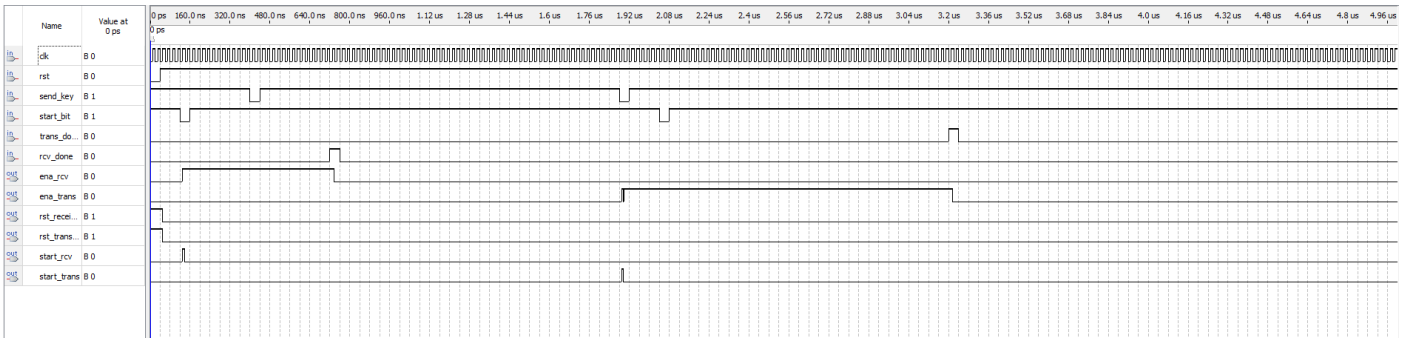

Figure 9. Transceiver Controller Timing Simulation

The simulation indicates that the controller can enable and disable the corresponding modules correctly. When a send key input is detected low, the controller will then output a start_trans signal routed to the transmitter to start the transmission, while in transmission, the start bit will not be detected so that the receiver module will not be activated; Similarly, if the start bit is firstly detected, the signal of send_key will not detected, so that the transceiver will not activate the transmitter module while in progress of receiving.

2.1.2 IrDA Transmitter

Interconnections of Modules

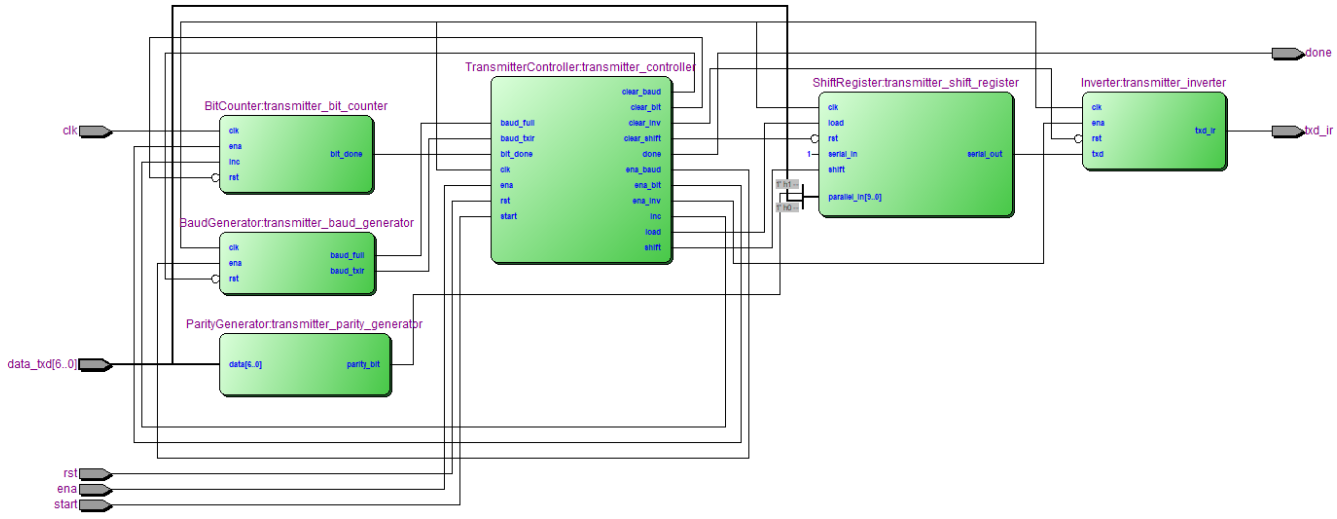

Figure 10. Module Interconnections

Verilog Code

```
module IrDATransmitter
(
       output       bit_done,
       output       shift,
       output       inc,
       output       baud_txir,
       output       baud_full,
       output       ena_bit,
       output       ena_baud,
       output       ena_inv,

       input        clk,
       input        rst,
       input        ena,
       input        start,
       input [6:0] data_txd,
       output       txd_ir,
       output       done
);

wire clear_baud;
wire clear_bit;
wire clear_shift;
wire clear_inv;
wire bit_done;
wire shift;
wire inc;
wire baud_full;
wire baud_txir;
wire ena_bit;
wire ena_baud;
wire ena_inv;
wire load;
```

```
wire parity_bit;
wire txd;


ShiftRegister transmitter_shift_register
(
	.clk(clk),
	.rst(!clear_shift),
	.shift(shift),
	.load(load),
	.serial_in(1'b1),
	.parallel_in({1'b1, parity_bit, data_txd, 1'b0}),
	.serial_out(txd)
);

BaudGenerator transmitter_baud_generator
(
	.clk(clk),
	.rst(!clear_baud),
	.ena(ena_baud),
	.baud_full(baud_full),
	.baud_txir(baud_txir)
);

BitCounter transmitter_bit_counter
(
	.clk(clk),
	.rst(!clear_bit),
	.ena(ena_bit),
	.inc(inc),
	.bit_done(bit_done)
);

ParityGenerator transmitter_parity_generator
(
	.data(data_txd),
	.parity_bit(parity_bit)
);

Inverter transmitter_inverter
(
	.clk(clk),
	.rst(!clear_inv),
	.ena(ena_inv),
	.txd(txd),
	.txd_ir(txd_ir)
);

TransmitterController transmitter_controller
(
	.clk(clk),
	.rst(rst),
	.ena(ena),
	.start(start),
	.bit_done(bit_done),
	.baud_full(baud_full),
	.baud_txir(baud_txir),
	.shift(shift),
	.load(load),
	.inc(inc),
	.ena_baud(ena_baud),
	.ena_bit(ena_bit),
	.ena_inv(ena_inv),
	.clear_baud(clear_baud),
	.clear_bit(clear_bit),
	.clear_shift(clear_shift),
	.clear_inv(clear_inv),
	.done(done)
```

```
);
endmodule
```

2.1.3 IrDA Receiver

Interconnections of Modules

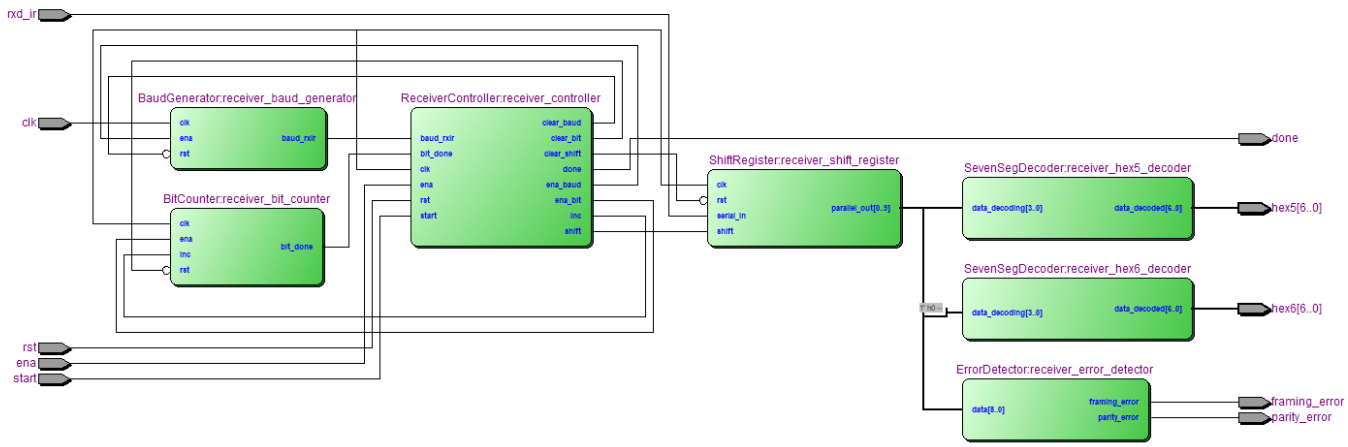

Figure 11. Module Interconnections

Datapath of the receiver: The data is input bit by bit from the rxd port in a specific baud rate. There is also a copy of the input data flows into the controller of the receiver for it to determine whether there is a start bit. It is worth noting that the start bit(0) is neglected when transferring from shift register to the data register as it will be useless in next processes. Finally, from the shift register, the data are drawn out in different form: a whole set of 8-bits data is taken by the error detector; the first four bits of the data, indicating the lower hexadecimal bit, are given to the 7-segment decoder to turn on the hex5 7-segment LED; the last three bits with the stop bit excluded and a binary 0 added to the MSB, indicating the upper hexadecimal bit, are given to the 7-segment decoder to turn on the hex6 7-segment LED.

Verilog Code

```
module IrDAReceiver
(
	input		clk,
	input		rst,
	input		ena,
	input		start,
	input		rxd_ir,
	output [6:0]	hex6,
	output [6:0]	hex5,
	output		parity_error,
	output		framing_error,
	output		done
);

wire [9:0] data_rxd;
wire clear_baud;
wire clear_bit;
wire clear_shift;
wire bit_done;
```

```
wire shift;
wire inc;
wire baud_rxir;
wire ena_bit;
wire ena_baud;
wire ena_inv;
wire parity_bit;
wire txd;

ShiftRegister receiver_shift_register
(
	.clk(clk),
	.rst(!clear_shift),
	.shift(shift),
	.serial_in(rxd_ir),
	.parallel_out(data_rxd)
);

BaudGenerator receiver_baud_generator
(
	.clk(clk),
	.rst(!clear_baud),
	.ena(ena_baud),
	.baud_rxir(baud_rxir)
);

BitCounter receiver_bit_counter
(
	.clk(clk),
	.rst(!clear_bit),
	.ena(ena_bit),
	.inc(inc),
	.bit_done(bit_done)
);

ErrorDetector receiver_error_detector
(
	.data(data_rxd[9:1]),
	.parity_error(parity_error),
	.framing_error(framing_error)
);

SevenSegDecoder receiver_hex6_decoder
(
	.data_decoding({1'b0, data_rxd[7:5]}),		// a 0 is put in front to meet the
ASCII table
	.data_decoded(hex6)
);

SevenSegDecoder receiver_hex5_decoder
(
	.data_decoding(data_rxd[4:1]),
	.data_decoded(hex5)
);

ReceiverController receiver_controller
(
	.clk(clk),
	.rst(rst),
	.ena(ena),
	.start(start),
	.bit_done(bit_done),
	.baud_rxir(baud_rxir),
	.shift(shift),
	.inc(inc),
	.ena_baud(ena_baud),
	.ena_bit(ena_bit),
	.clear_baud(clear_baud),
```

```
        .clear_bit(clear_bit),
        .clear_shift(clear_shift),
        .done(done)
);
endmodule
```

3 Module Documentation

3.1 Generic Modules

In this assignment, the baud generator, bit counter and the shift register can be implemented by generic modules and can be derived to different instances for either the transmitter or the receiver. The unnecessary pins will not be used if the transmitter or the receiver does not require the relative function.

3.1.1 Baud Generator

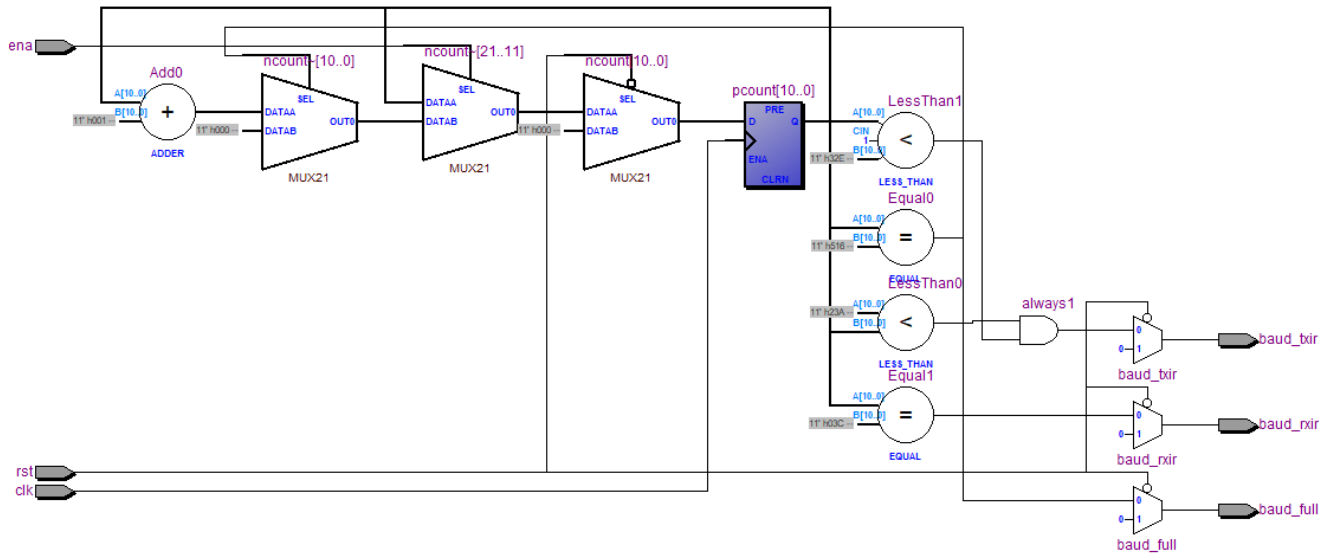

Figure 12. Generic Baud Generator RTL View

The generic baud generator module generates three different baud ticks by counting the number of system clocks. The baud generator has a enable input ena which enables the modules and a reset input rst which reset the internal counter to 0, their active states are listed below:

Table 3. Baud Generator Active State

Signal	Active State
ena	active high
rst	active low

baud_txir	active high
baud_rxir	active high
baud_fxir	active high

ASM Chart

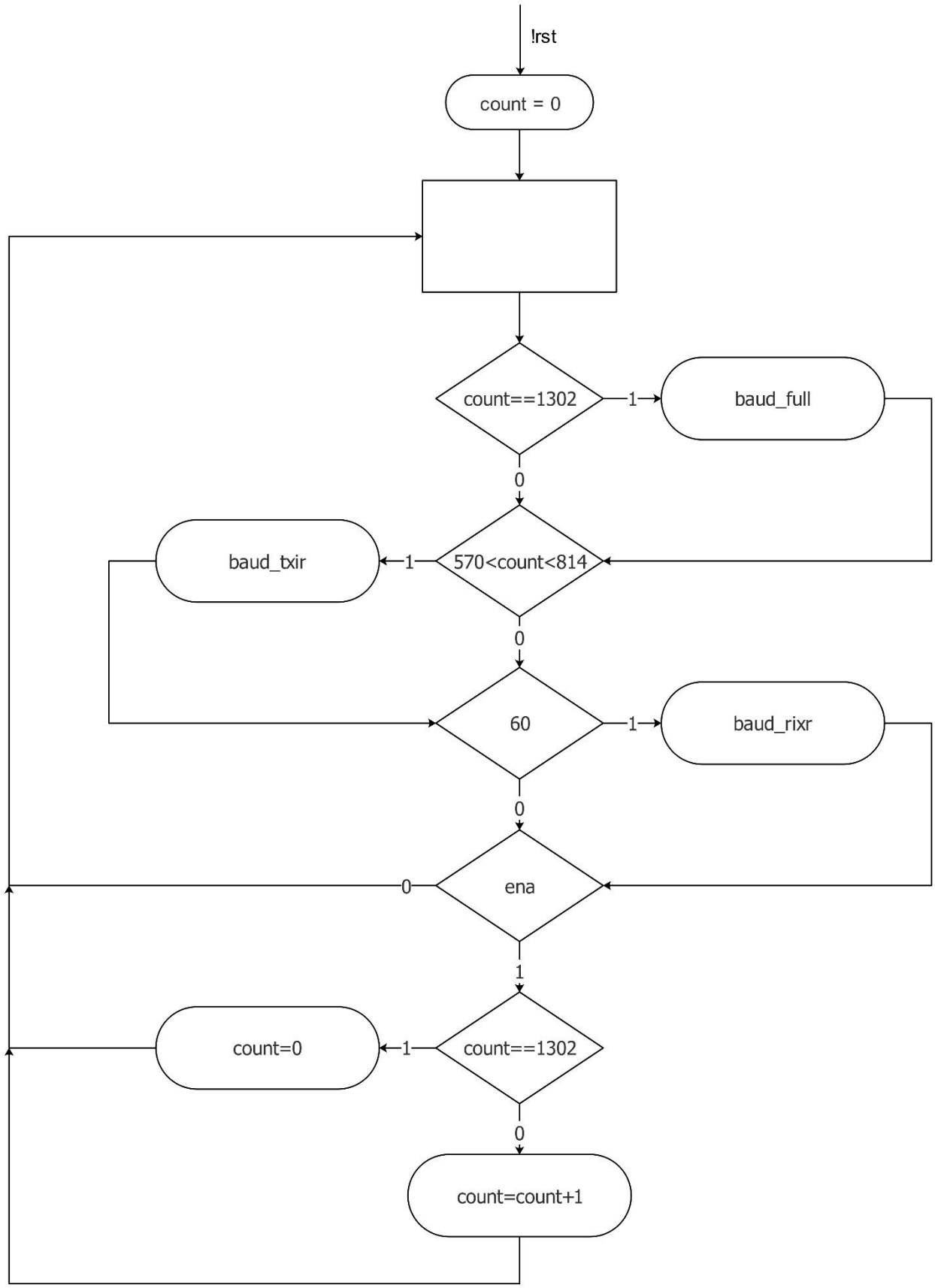

Figure 13. Generic Baud Generator ASM

The baud generator has a counter to count the number of system clocks. Upon reset, the counter is set to 0, and it generates several outputs at different conditions.

The baud generator generates a pulse at the maximum counting number 1302, which is calculated based on the given baud rate and the frequency system clock:

$$count = \frac{50000000}{38400} = 1302$$

Every time when the counter reaches this number, the baud generator will output a signal to indicates that, and the signal will then be used by the controller and other modules to transmit or receive the data.

Similarly, the baud generator also generates signals when the counter is 60, or in the range from 570 to 814. They are calculated based on the IrDA protocol standard, which is, 3/16 of the data period of the IrDA transmitter, and 2.4 micro-seconds for the IrDA receiver. Therefore, based on this, the baud generator generates a signal to sample the transmitting data at:

$$\frac{7}{16} \times 1302 < count < \frac{10}{16} \times 1302$$

For the IrDA receiver a pulse should be generated at the half of the 2.4 micro-seconds, which is 1.2 micro-seconds, to ensure the data accuracy when sampling. Since the frequency of the system clock is 50MHz, the necessary count number can be calculated as:

$$count = 1.2 \times 50 = 60$$

Evert time the internal count of the baud generator reaches these number, the baud generator generate a corresponding signal to the controller so that the controller can further control the other units based on that.

Table 4. Output Conditions

Number of System Clocks	Output
1302	baud_full = 1
570 - 814	baud_txir = 1
60	baud_rxir = 1

Verilog Code

```
/* Generic Baud Generator Module */

/* This module generator a tick every different number of system clocks */

module BaudGenerator
(
      input  clk,
```

```
	input rst,
	input ena,
	output reg   baud_full,
	output reg   baud_txir,
	output reg  baud_rxir
);


parameter BAUD38400             = 11'd1302;
parameter SEVEN_SIXTEENTH       = 11'd570;
parameter TEN_SIXTEENTH         = 11'd814;
parameter HALF_PERIOD           = 11'd60;

// counter that counts up to 2^11 = 2048 because we need 1302 as the max number
reg [10:0] pcount, ncount;

always @(posedge clk)
begin
	pcount <= ncount;
end

always @(rst, pcount, ena)
begin
	baud_full    = 1'b0;
	baud_txir    = 1'b0;
	baud_rxir    = 1'b0;
	ncount              = pcount;

	if(!rst)
		ncount = 11'd0;
	else begin
		if(pcount == BAUD38400)
			baud_full = 1'b1;

		if(pcount > SEVEN_SIXTEENTH && pcount <= TEN_SIXTEENTH)
			baud_txir = 1'b1;

		if(pcount == HALF_PERIOD)
			baud_rxir = 1'b1;

		if(ena) begin
			if(pcount == BAUD38400)
				ncount = 11'd0;
			else
				ncount = pcount + 11'd1;
		end
	end
end
endmodule
```

Simulation Results

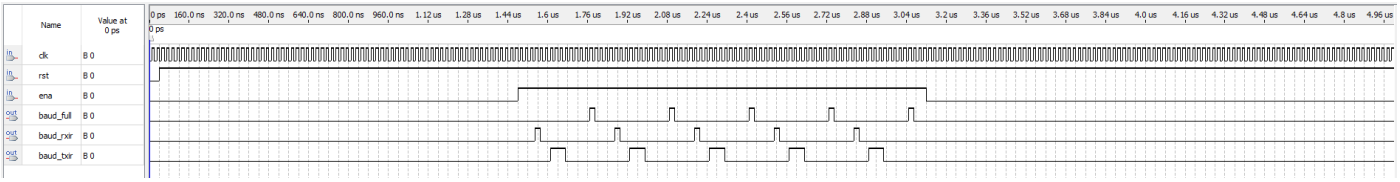

Figure 14. Generic Baud Generator Functional Simulation

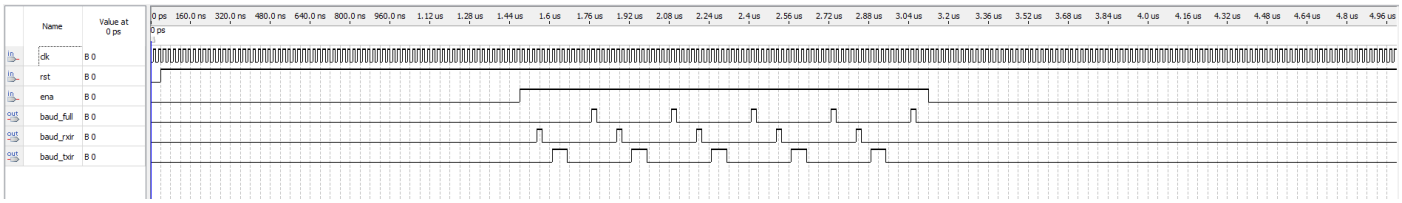

Figure 15. Generic Baud Generator Timing Simulation

For convenience of the simulation, the baud generator is set to generate a full pulse every 16 clock cycles, the range for IrDA transmitter is set to be 6 – 9, and the sampling point for IrDA receiver is set to be 4 (assuming the pulse width of the receiver is 8). As the simulation shows, the enable signal is set to logic 0 at the start to reset the internal counter. When the enable signal is logic 0, the baud generator is disabled thus no baud pulse is generated. As the enable signal is logic 1, the baud generator is then enabled and a full pulse is generated after each 16 clock cycles, a rxir pulse is generated at each 4 clock cycle, and a txir pulse is generated in that corresponding range.

3.1.2 Bit Counter

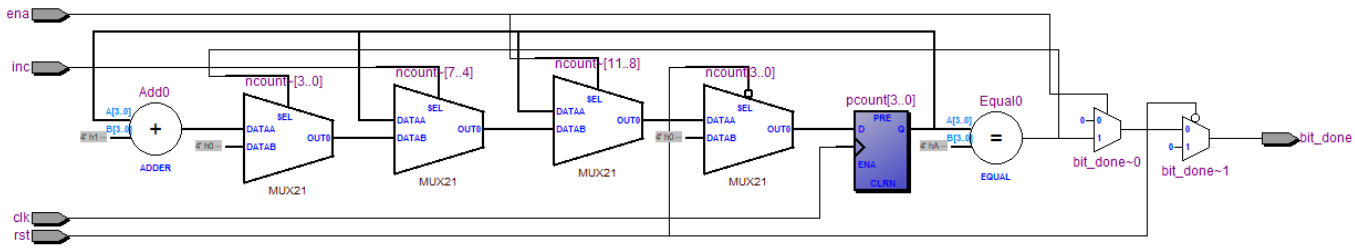

Figure 16. Generic Bit Counter RTL View

The bit counter module has the reset input rst to reset the current counted number to 0. The enable signal ena enables the bit counter but it will not count until the inc signal is input indicate to increment the internal counter.

Table 5. Bit Counter Active State

Signal	Active State
rst	active low
ena	active high
inc	active high
bit_done	active high

ASM Chart

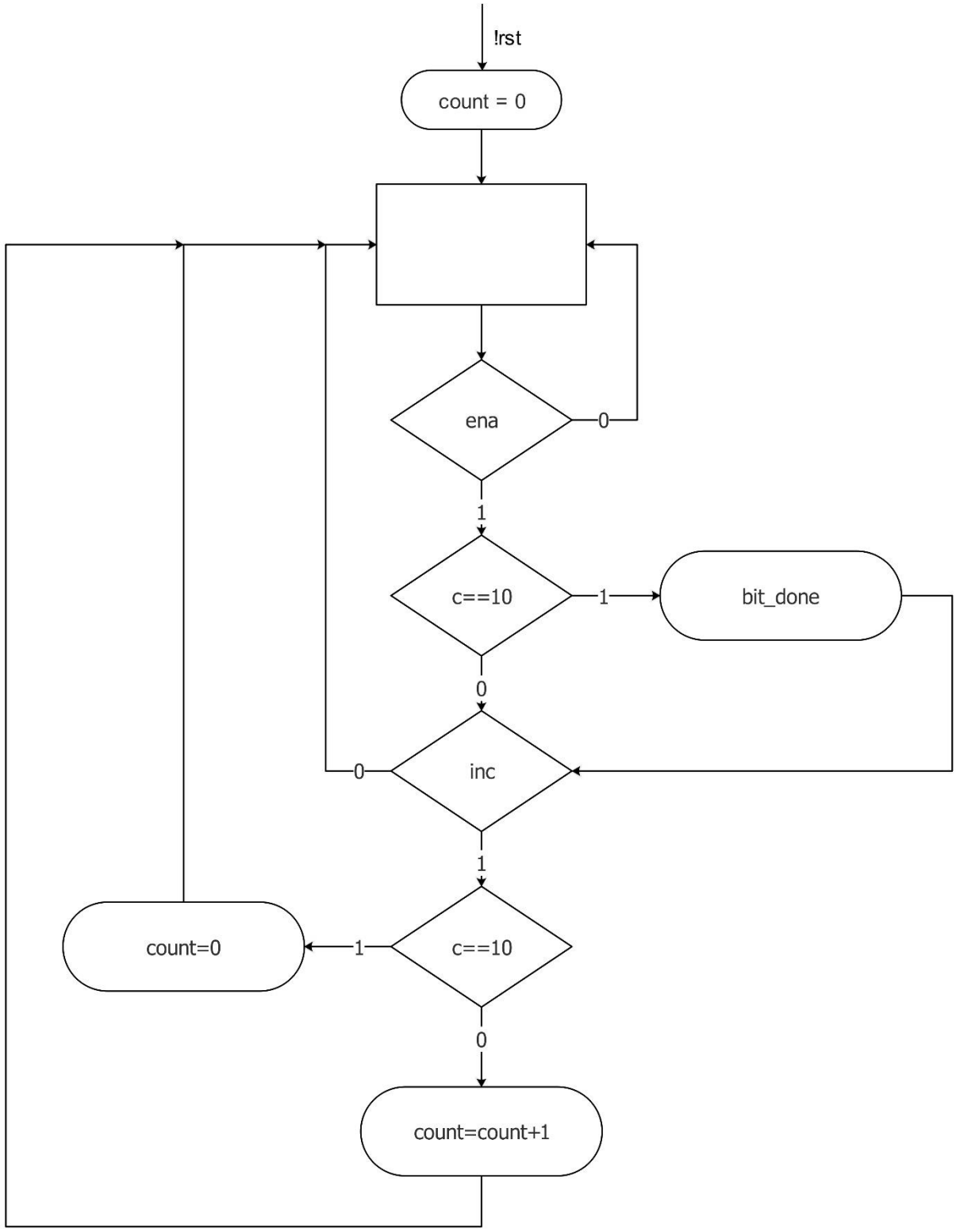

Figure 17. Generic Bit Counter ASM

Verilog Code

```
/* Generic Bit Counter Module */

/* The bit counter has a increment of 1 whenever it receives a
```

```
INC signal from the controller. If the counted bits reaches the
desired value, it outputs a bit done signal of 1 */

module BitCounter
(
	input		clk,
	input		rst,
	input		ena,
	input		inc,
	output reg	bit_done
);

parameter BITNUM = 4'd10;

reg [3:0] pcount, ncount;

always @(posedge clk)
begin
	pcount <= ncount;
end

always @(rst, ena, pcount, inc)
begin
	bit_done = 1'b0;
	ncount = pcount;

	if(!rst)
		ncount = 4'd0;
	else begin
		if(ena) begin
			if (pcount == BITNUM)
				bit_done = 1'b1;

			if (inc)
				if (pcount == BITNUM)
					ncount = 4'd0;
				else
					ncount = pcount + 4'd1;
		end
	end
end
endmodule
```

Simulation Results

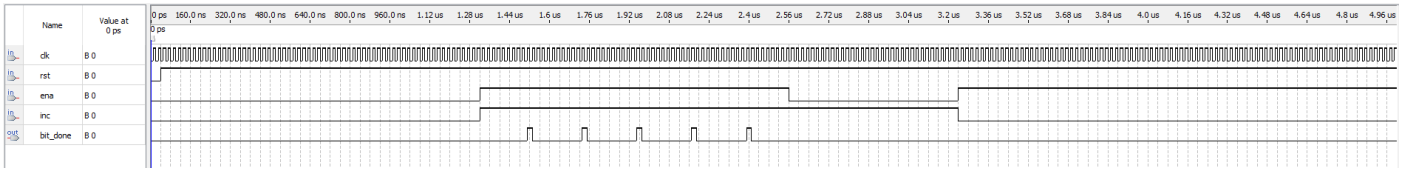

Figure 18. Generic Bit Counter Functional Simulation

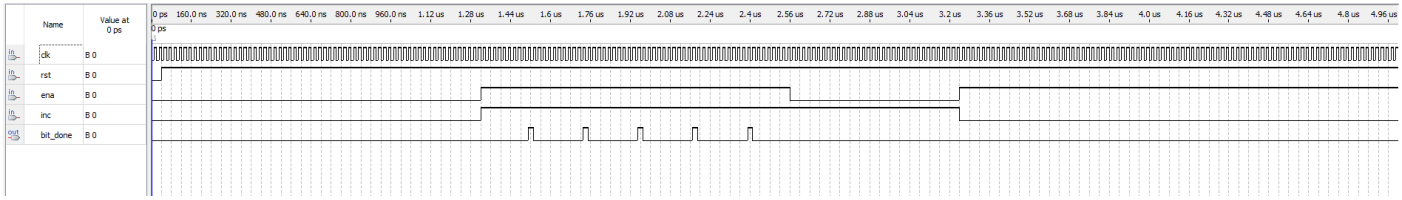

Figure 19. Generic Bit Counter Timing Simulation

As shown in the simulations, the counter only starts to work when both ena and inc is logic 1 and it generates a pulse bit_done every 10 clock cycles. Either a logic 0 of input for ena and inc will disable

the bit counter. The difference of ena and inc is, if inc is logic 0 but ena is logic 1, the bit counter is still enabled, which means its internal counter is counting, but no pulse will be generated for the bit_done.

3.1.3 Shift Register

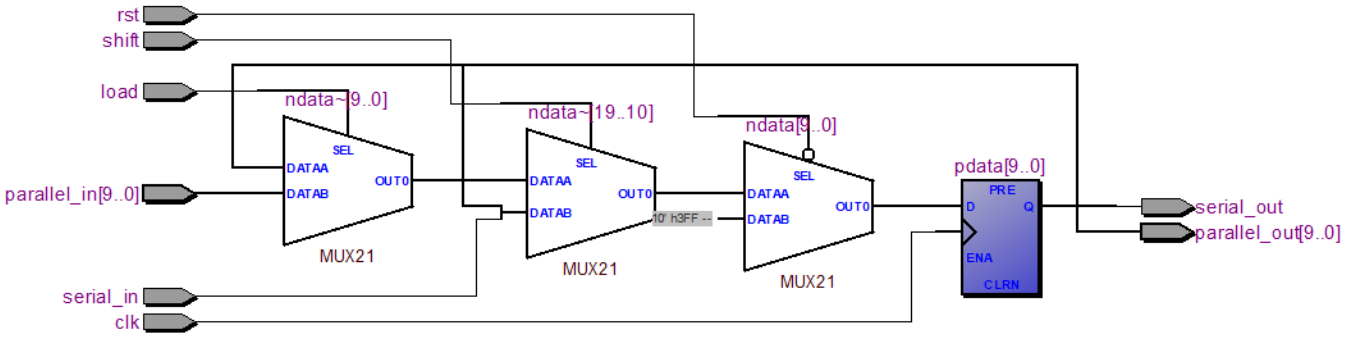

Figure 20. Generic Shift Register RTL View

The generic shift register is both a serial-in-parallel-out and parallel-in-serial-out shift register so that it can be used for both the transmitter and the receiver. For the transmitter, it is only necessary to use the shift, load, serial_in ,parallel_in, and serial_out port; for the receiver, the data input and output that are used are serial_in and parallel_out. It is worth noting that for the transmitter the serial_in input is connected to constantly transmit logic 1 bits, which is the idle bits. The input load loads the data from parallel_in to the shift register and the input shift shifts the shift register 1 bit to the right.

Table 6. Shift Register Active State

Signal	Active State
rst	active low
load	active high
shift	active high

ASM Chart

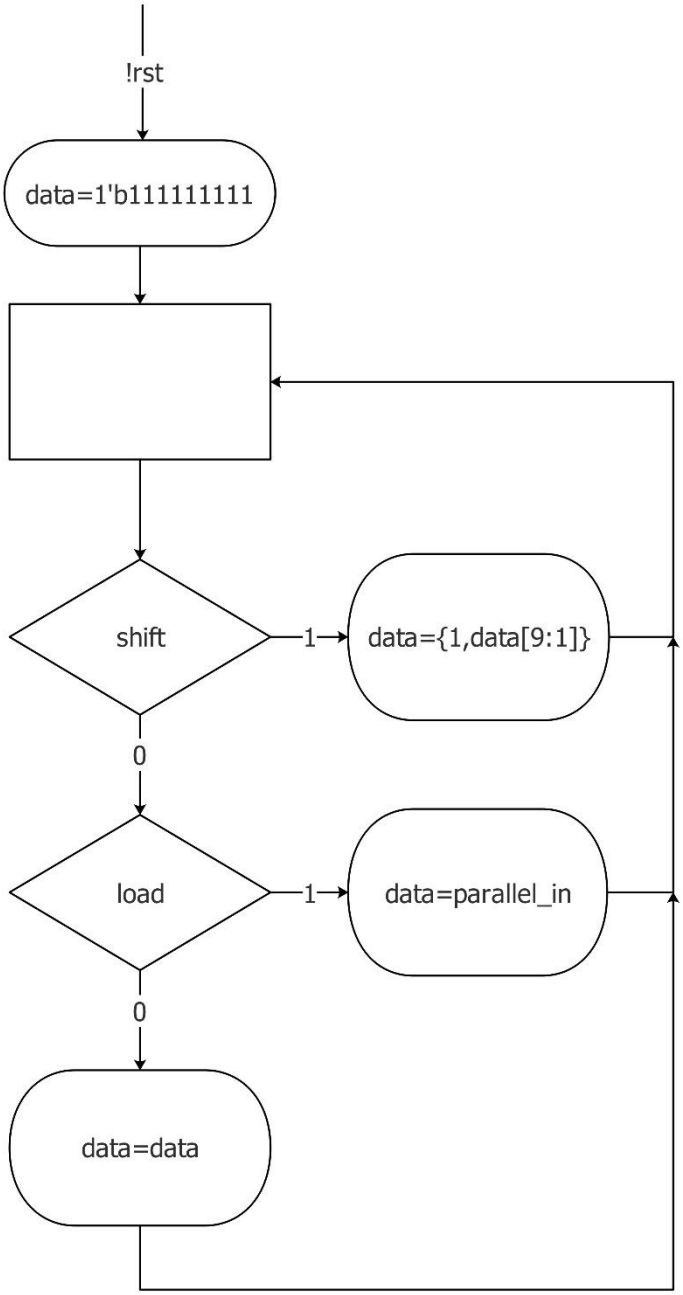

Figure 21. Generic Shift Register ASM

As shown in the ASM chart, if the shift register takes a logic 1 load signal, it loads in the parallel data. Each time it receives a positive baud clock, the data will be shifted to the right, and the most significant bit will be replaced by a 0. The serial output is always the least significant bit. Additionally, a logic 0 reset signal should reset the data to 10 bits of 1, which indicate the idle bits.

As shown in the ASM chart, the shift register has the registered data as its output. A logic 0 reset signal resets its registered value to 10-bit 1s. At each baud clock, the shift register shift in a new data bit as MSB and shift out the LSB without storing.

Verilog Code

```
/* Generic Shift Register */

/* This shift register is used for both the transmitter
and the receiver since it conbines both the PISO and
SIPO functionality */

module ShiftRegister
(
      input             clk,
      input             rst,
      input             shift,// signal to shift the data in the shift register
      input             load, // signal to load the parallel data to the shift register
      input             serial_in,
      input  [9:0]      parallel_in,
      output            serial_out,
      output [9:0]      parallel_out
);

parameter IDLEBITS = 10'b111_111_111_1;

reg [9:0] pdata, ndata;  // present data and next data

assign serial_out  = pdata[0];
assign parallel_out      = pdata;

always @(posedge clk)
begin
      pdata <= ndata;
end

always @(rst, pdata, shift, load, serial_in, parallel_in)
begin
      ndata = pdata;                  // hold the data in the register

      if(!rst)
            ndata = IDLEBITS;         // reset the data as idle bits (i.e. logic 1)
      else begin
            if(shift)
                  ndata = {serial_in, pdata[9:1]};
// shift the data by 1 bit, set the serial data input as the MSB
            else if(load)
                  ndata = parallel_in;            // load the parallel data input
      end
end
endmodule
```

Simulation Results

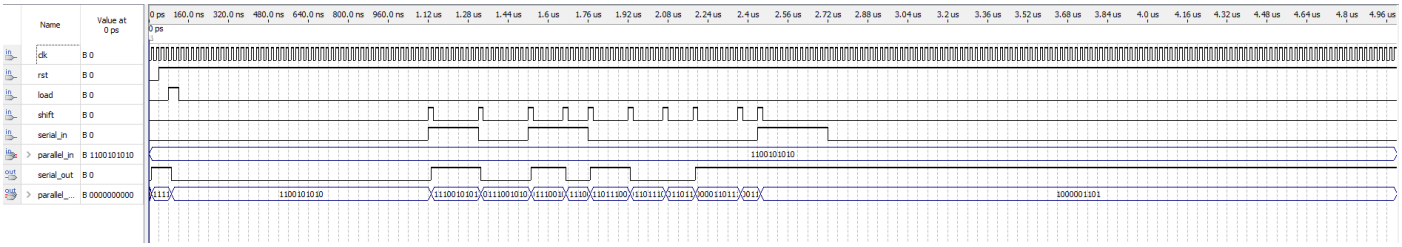

Figure 22. Generic Shift Register Functional Simulation

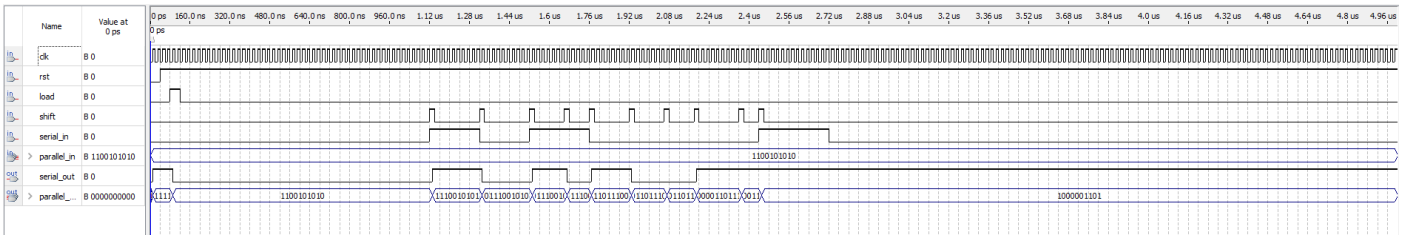

Figure 23. Generic Shift Register Timing Simulation

The simulations had tested the function of the load and shift of the generic shift register. Firstly, the load signal make the shift register load the data from the parallel_in input and hold it. The parallel_out output is then become that data, which is 1100101010 as the input indicates. Then to test the shift function, 10 pulses of shift input is introduced and they shifts the shift register 10 times, the LSB in the shift register is shifted out while a new MSB is shifted in from the serial_in input. As illustrated in the simulation, the serial_in input data combined with the shift signal gives the data from MSB to LSB: 1000001101, which is the final result of the parallel_out output. Hence the simulations show that the generic shift register is suitable for the use of both transmitter and the receiver.

3.2 Transmitter Modules

The parity generator module and the controller module are designed for the IrDA transmitter only and they are not able for the receiver to use.

3.2.1 Parity Generator

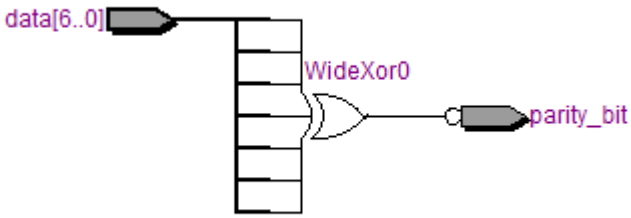

Figure 24. Parity Generator RTL View

The parity generator module is a pure combinational module. It generates a parity bit based on the input data bits. Since the odd parity is followed in this design, to generate the parity bit, it is only necessary to XOR all the data bits. If the result is 1, it indicates that the data contains the odd number of bits, therefore the parity bit is generated as 0, and vice versa.

Table 7. Parity Generator Active State

Signal	Active State
parity_bit	active low

ASM Chart

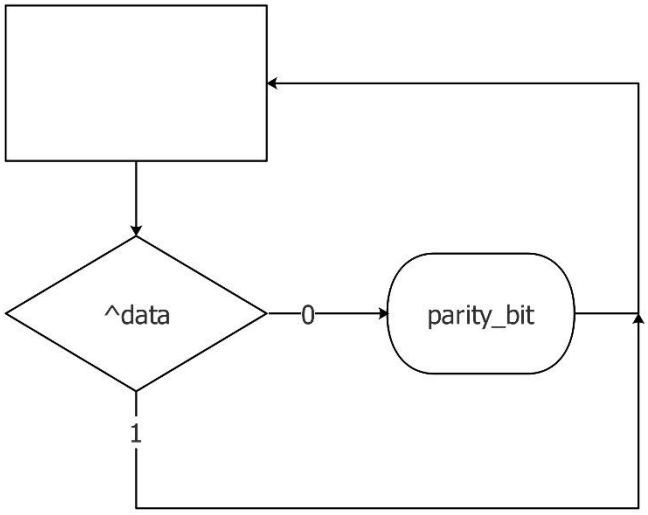

Figure 25. Parity Generator ASM

To determine what the parity bit should be, a bitwise XOR operation will give the result either 1 or 0. If the result is 1, that means there is odd number of 1s in the data bits, which leads the parity bit to be 0 in odd parity rule. Otherwise, a result of 0 indicates that there is even number of 1s in the data bits so that a parity bit of 1 is necessary to make the data frame become odd parity.

Verilog Code

```
/* Transmitter */

/* Parity Generator Module */

/* This module generate a parity bit
according to the given data bits. Odd
parity is followed */

module ParityGenerator
(
      input [6:0]  data,
      output       parity_bit
);

assign parity_bit = !(^data);

/* if XOR the data bits result a 1, it means the total
number of 1 in data bits is odd, thus the parity bit is
0; if XOR the data bits result a 0, it means the total
number of 1 in data bits is even, thus the parity bit is 1 */
endmodule
```

Simulation Results

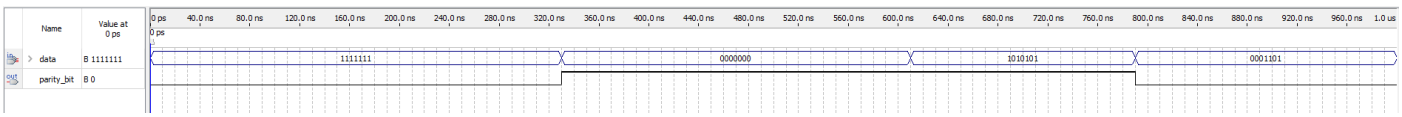

Figure 26. Parity Generator Functional Simulation

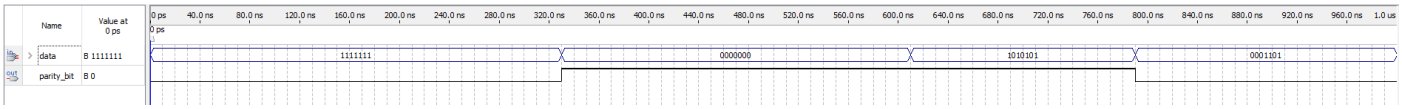

Figure 27. Parity Generator Timing Simulation

As a combinational logic module, there is no difference between functional and timing simulations. The simulation result shows that the logic works well to generate a required parity bit for the data bits.

From 0 – 330ns, the data bits are 1111111, which comprises odd number of 1s, so that the parity bit is 0.

From 330 – 610ns, the data bits are 0000000, which comprises none of 1s. Since 0 is also an even number, the parity bit is accordingly set to 1.

From 610 – 790ns, the data bits are 1010101, which comprises 4 of 1s, and the parity bit is 1.

From 790ns, the data bits are 0001101, which comprises 3 of 1s, as a result the parity bit is 0.

3.2.2 Inverter

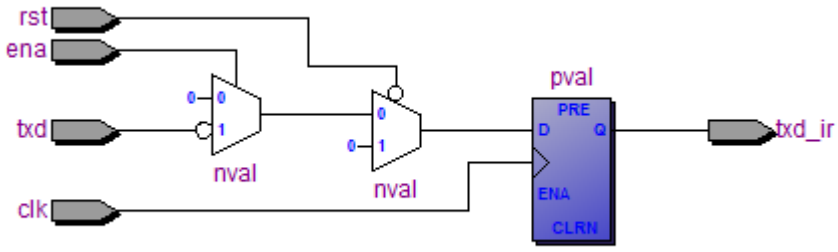

Figure 28. Inverter RTL View

The inverter is used to invert the transmitter data, for idle bits, from logic 1 to logic 0, and for the data bits, from logic 0 to logic 1 in 3/16 of the data period. The transmitter data txd is only inverted when the enable input ena is activated.

Table 8. Inverter Active State

Signal	Active State
rst	active low
txd	active low
ena	active high
txd_ir	active high

ASM Chart

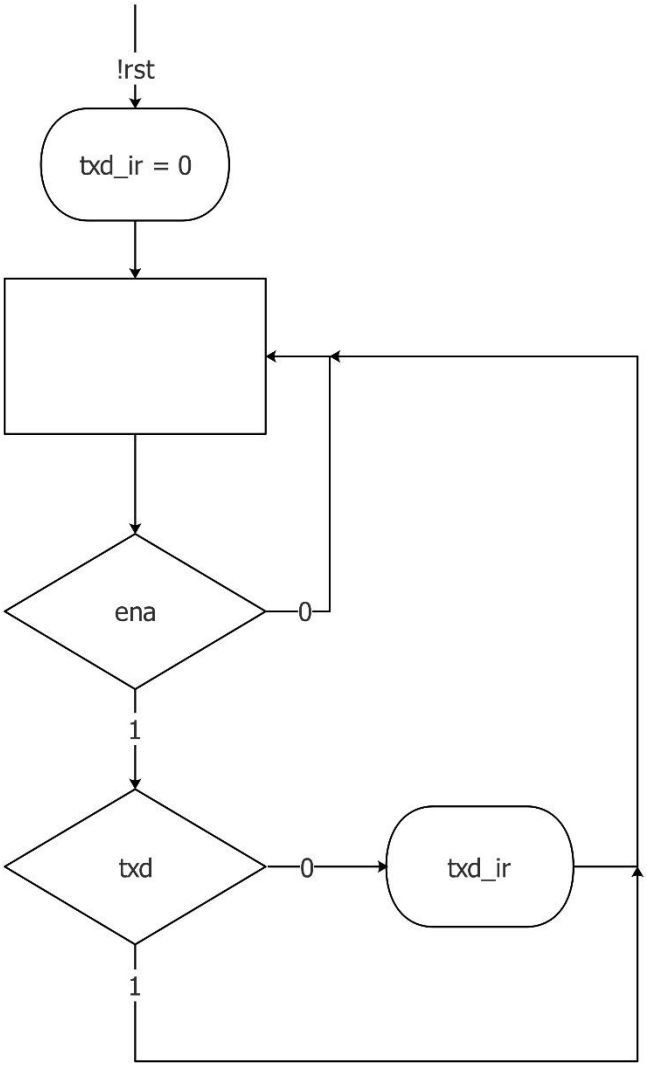

Figure 29. Inverter ASM

Verilog Code

```
module Inverter
(
	input		clk,
	input		rst,
	input		ena,
	input	txd,
	output	txd_ir
);

reg pval, nval;

assign txd_ir = pval;

always @(posedge clk)
begin
	pval <= nval;
```

```
end

always @(rst, pval, ena, txd)
begin
	nval = 1'b0;

	if(!rst)
		nval = 1'b0;
	else if(ena)
		if(txd == 1'b0)
			nval = 1'b1;
end
endmodule
```

Simulation Results

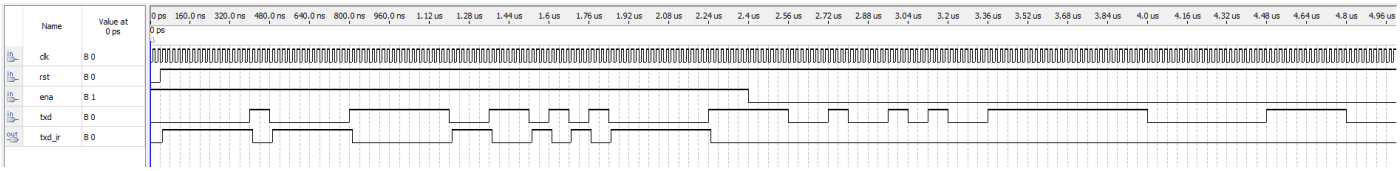

Figure 30. Inverter Functional Simulation

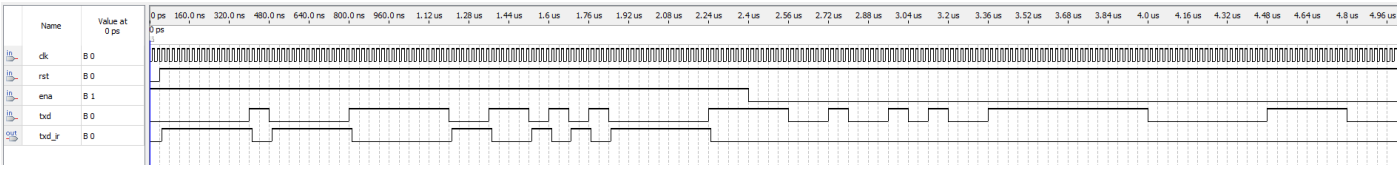

Figure 31. Inverter Timing Simulation

When the inverter is enabled, the simulation shows that the txd data is inverted from logic 1 to logic 0 or from logic 0 to logic 1. It is worth noting that the txd_ir output is clocked so that it is only inverted when a positive edge of the clock is detected. Also, when the enable signal ena is logic 0, the inverted stops to work and the output data becomes logic 0 by default.

3.2.3 Controller

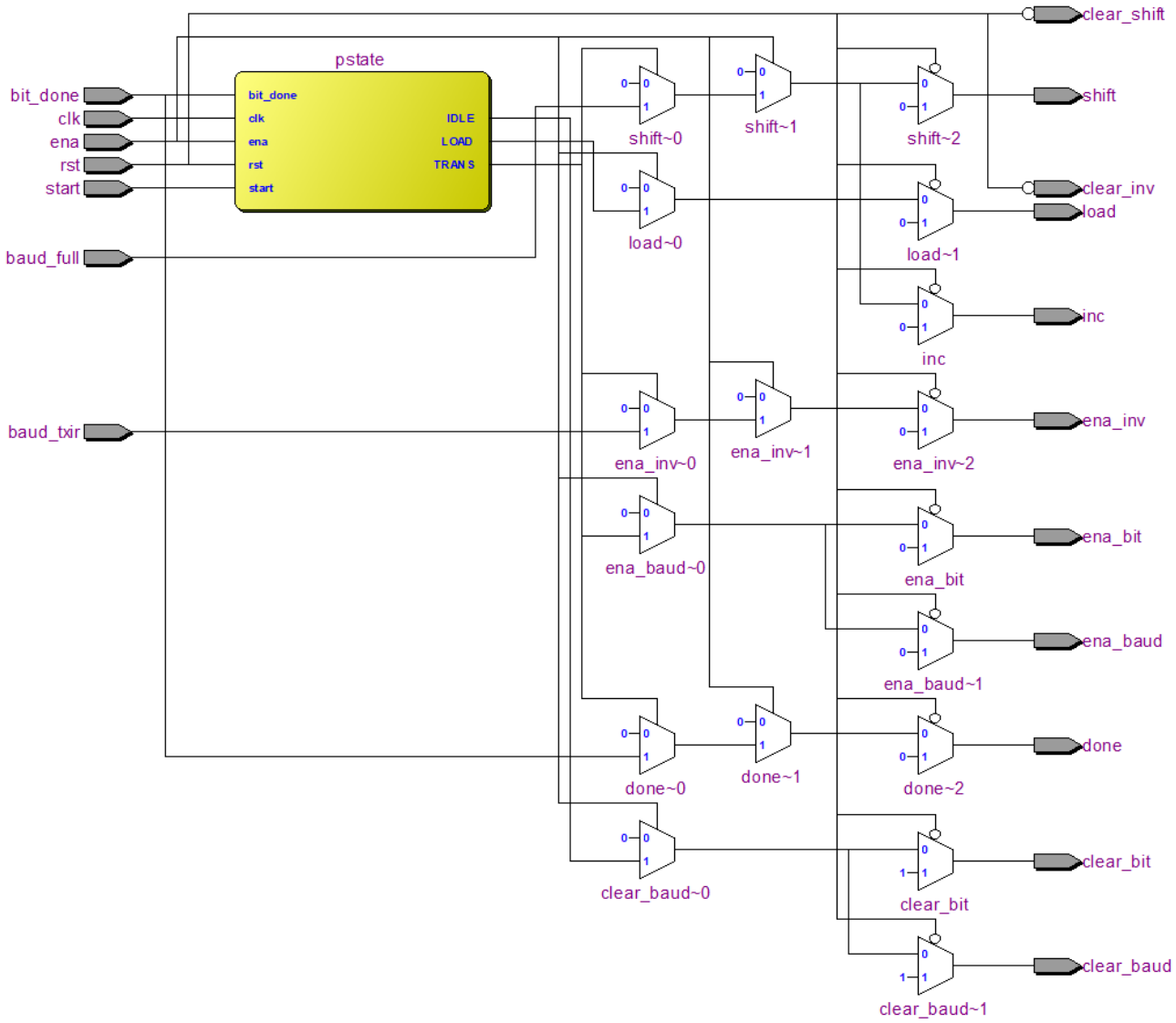

Figure 32. Transmitter Controller RTL View

The controller module of the transmitter controls the other modules by sending control signals and receiving status signals. Altogether they constitute the complete transmitter module. The controller module receives the signal from the baud generator, bit counter and the peripheral input signals. Depending on the input signals, the controller output some control signal that either turn on or off the other modules of the transmitter. An additional done output signal is introduced to indicate the complete of the transmission.

Different from the UART controller, the IrDA transmitter has an additional ena_inv signal which enables the inverter when the baud_txir signal is detected which indicates that 3/16 of the data pulse width, so that it can invert the data pulse at that period to fit the IrDA transmission standard.

ASM Chart

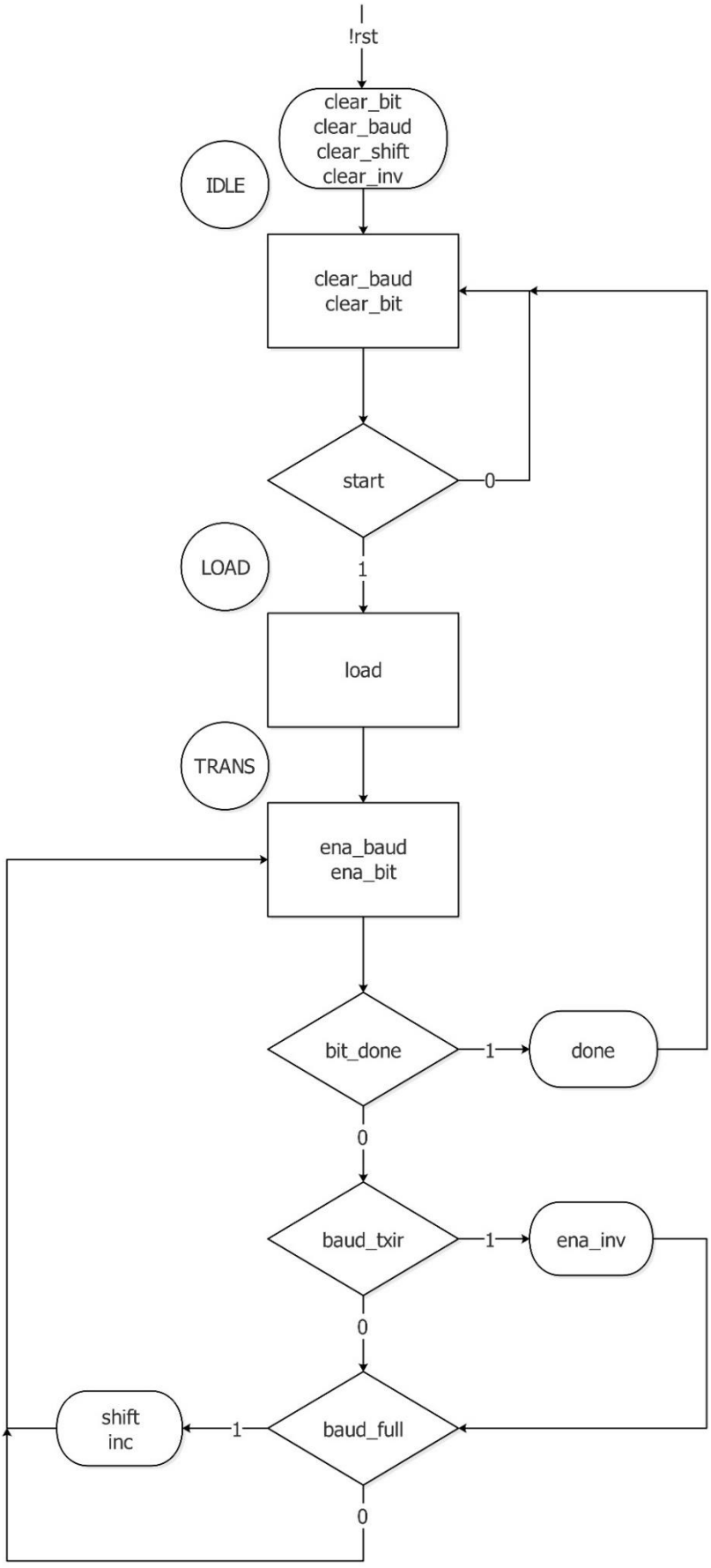

Figure 33. Transmitter Controller ASM

State Transition Diagram

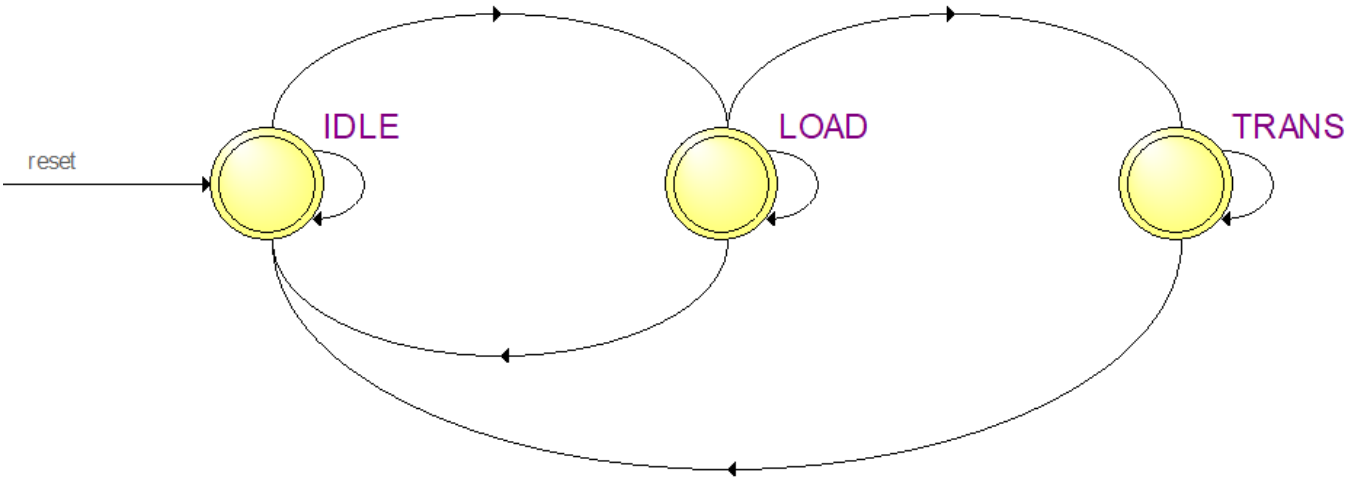

Figure 34. Transmitter Controller State Transition Diagram

	Source State	Destination State	Condition
1	IDLE	LOAD	(start).(ena).(rst)
2	IDLE	IDLE	(!start) + (start).(!ena) + (start).(ena).(!rst)
3	LOAD	LOAD	(!ena).(rst)
4	LOAD	TRANS	(ena).(rst)
5	LOAD	IDLE	(!rst)
6	TRANS	TRANS	(!bit_done).(rst) + (bit_done).(!ena).(rst)
7	TRANS	IDLE	(!bit_done).(!rst) + (bit_done).(!ena).(!rst) + (bit_done).(ena)

Figure 35. State Table

The transmitter controller has 3 states: IDLE, LOAD, and TRANS. A reset signal will always set the next state to the IDLE state no matter what the present state is. The controller begins with the IDLE state by a reset input.

In IDLE state, the transmitter is ready to transmit a set of data and if there is no send bit detected, the state will remain in IDLE. As soon as a start bit of logic 1 is detected, the state moves to the next state, the LOAD state.

In the LOAD state, the controller enable send a load signal to the shift register, therefore, the shift register loads the input data. After one system clock cycle, the state will move to the TRANS state in order to transmit the loaded data.

In the TRANS state, the transmitter is in progress transmitting the data,. The load signal is turned off so new input data will not be loaded. Instead, the controller will turn on the baud generator and the bit counter and monitor the transmission state. The current loaded data will be transmitted according to the baud rate generated by the baud generator. The transmitting process will complete

when a bit done signal is detected from the bit counter, and the controller will move to the IDLE state and wait for the next sending signal.

Verilog Code

```
/* Transmitter */
/* Controller Module */

/* Controller module for the transmitter
which has 4 state: IDLE, LOAD, TRANS, and HOLD,
which indicates, respectively, the transmitter is idle,
the transmitter loads the data, the transmitter is
transmitting  the data, the transmitter is preparing
for the next idle state. The controller output
relative signals to control the other module of the transmitter */

module TransmitterController
(
      input        clk,
      input        rst,
      input        ena,
      input        start,
      input        bit_done,
      input        baud_full,
      input        baud_txir,
      output reg   shift,       // shift the shift register
      output reg   load,        // load data to the shift register
      output reg   inc,         // increment the bit counter
      output reg   ena_baud,    // enable the baud generator
      output reg   ena_bit,     // enable the bit counter
      output reg   ena_inv,     // enable the inverter
      output reg   clear_baud,
      output reg   clear_bit,
      output reg   clear_shift,
      output reg   clear_inv,
      output reg   done
);

parameter IDLE     = 2'b00;     // idle state, where data is ready to be transmitted
parameter LOAD     = 2'b01;     // load state, where data is loaded
parameter TRANS    = 2'b11;     // transmitting state, where data is transmitted

reg [1:0] pstate, nstate;

always @(posedge clk)
begin
      pstate <= nstate;
end

always @(pstate, rst, ena, start, bit_done, baud_full, baud_txir)
begin
      shift        = 1'b0;
      load         = 1'b0;
      inc          = 1'b0;
      ena_baud     = 1'b0;
      ena_bit      = 1'b0;
      ena_inv      = 1'b0;
      clear_baud   = 1'b0;
      clear_bit    = 1'b0;
      clear_shift  = 1'b0;
      clear_inv    = 1'b0;
      done         = 1'b0;
      nstate       = pstate;          // default keep the present state in loop

      if(!rst) begin
            clear_baud   = 1'b1;
```

```
                clear_bit    = 1'b1;
                clear_shift  = 1'b1;
                clear_inv    = 1'b1;
                nstate       = IDLE;
        end
        else if(ena) begin
                case(pstate)
                        IDLE: begin
                        clear_baud   = 1'b1;
                        clear_bit    = 1'b1;

                        if(start)
                                nstate = LOAD;     // start to transmit when pushbutton is pressed
                        end

                        LOAD: begin
                                load   = 1'b1;     // load the data to the shift register
                                nstate= TRANS;     // move to the TRANS state
                        end

                        TRANS: begin
                                ena_baud = 1'b1;          // enable the baud generator
                                ena_bit  = 1'b1;          // enable the bit counter

                                if(bit_done) begin
// when 10 bits of data is counted, a frame of data is completed transmitting
                                        done   = 1'b1;
                                        nstate = IDLE;     // move to the HOLD state
                                end

                                if(baud_txir)
                                        ena_inv = 1'b1;

                                if(baud_full) begin       // shift and increment every baud tick
                                        shift  = 1'b1;
                                        inc    = 1'b1;
                                end
                        end

                        default begin
                                nstate = IDLE;            // IDLE state by default
                        end
                endcase
        end
end
endmodule
```

Simulation Results

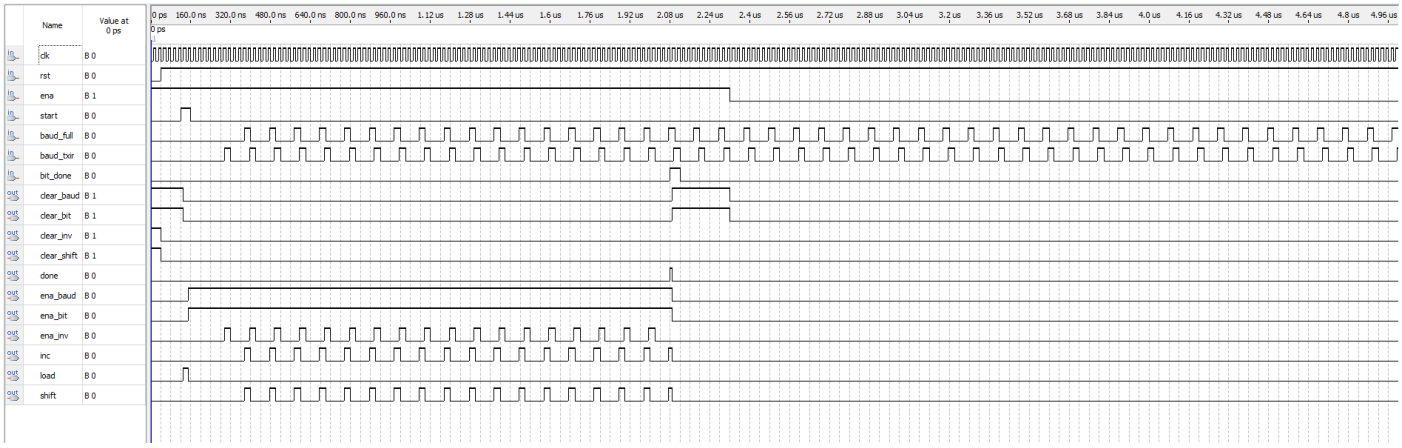

Figure 36. Transmitter Controller Functional Simulation

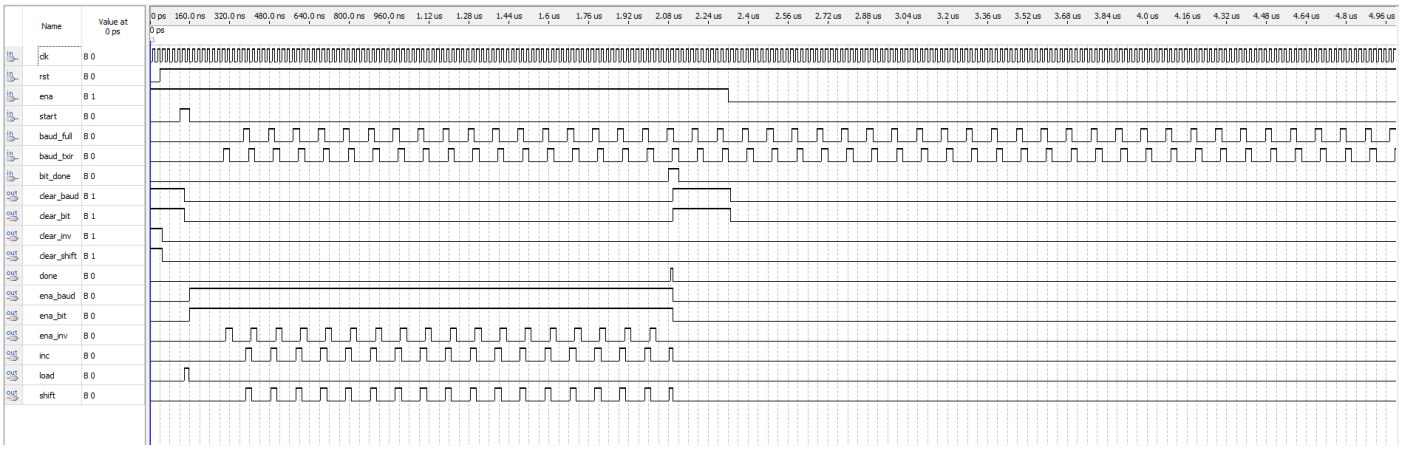

Figure 37. Transmitter Controller Timing Simulation

The simulation result shows that the controller can handle with different conditions appropriately when using the transmitter. It is worth pointing that the bit counter is set to count up to 11 which will be detected by the controller as the completion of the transmission.

Firstly, the controller is reset so that the state is set to the IDLE state, so that it clears the baud generator and the bit counter. By enabling the controller, it start to work. When a start signal is detected, the controller set the clear signals to 0, and the baud and the bit counter start to be generated, and the controller is entered into the LOAD state so that a load signal is generated.

Next, the controller enters the TRANS state where it controls the transmission based on the baud generator and the bit counter. At every baud_full pulse, the control send an inc and a shift signal to the bit counter and the shift register to shift the data. At every baud_txir pulse, the controller turns on the inverter so that the data pulses are inverted at that period.

When a bit_done signal is detected from the bit counter, the controller stop the TRANS state and send a done signal to indicate that the transmission is completed, and the controller is back to the IDLE state and waits for the next start signal.

3.3 Receiver Modules

The exclusive modules for the IrDA receiver contains the error detector module, and the 7-segment LED decoder module and the receiver controller module.

3.3.1 Error Detector

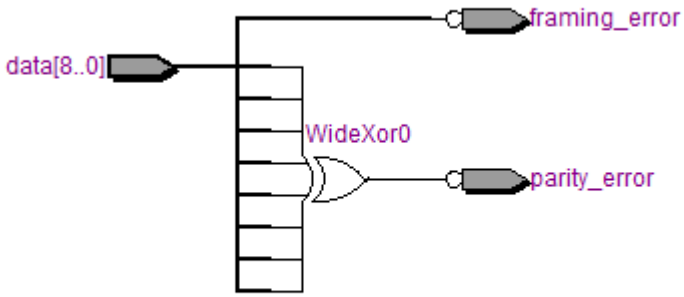

Figure 38. Error Detector RTL View

The error detector module of the receiver determines whether there is any parity error or framing error during the receiving process of the data. The module takes 1 input: data, which is 9-bit data composed of 1 stop bit + 1 parity bit + 7 data bits, and 2 output: framing_error and parity_error which indicate if there are any errors happening.

Table 9. Error Detector Active State

Signal	Active State
framing_error	active low
parity_error	active low

ASM Chart

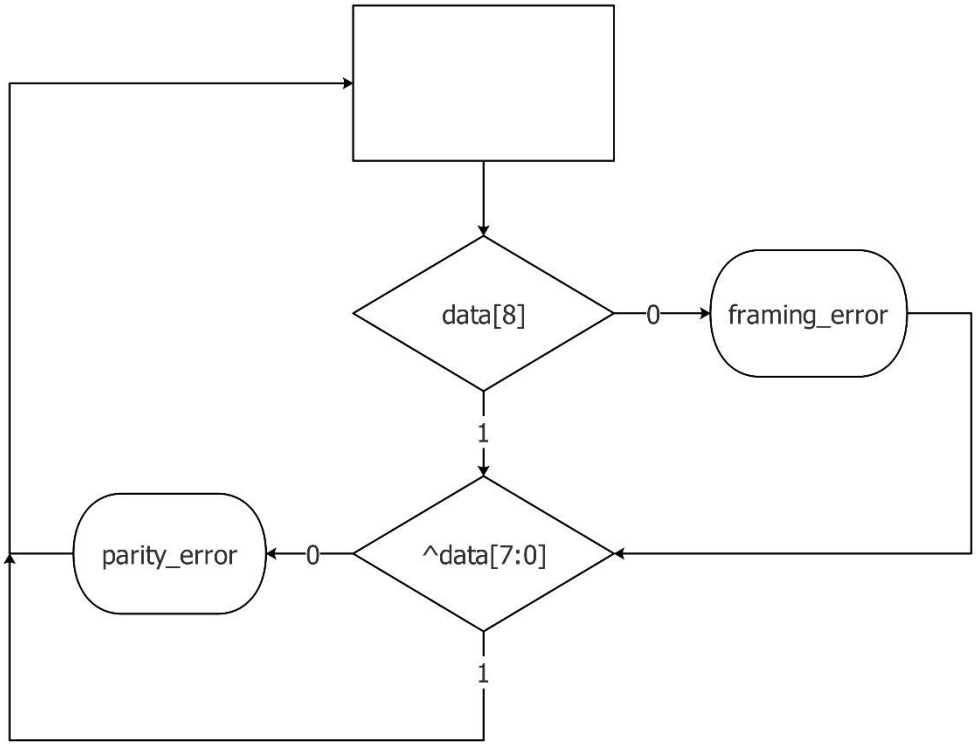

Figure 39. Error Detector ASM

Figure 42 showcases the algorithm of deciding parity errors and framing errors. For the framing error, the error detector simply looks at the stop bit of the input data. If the stop bit of a frame of data is not 1, it means the data is incorrectly framed which can cause incomplete data to be transferred, so that the framing_error is set to 1.

Next for the parity error, the stop bit should be neglected at first. By XORing both the parity bit and the data bits, the result of 1 should indicate the total number of 1 in the data set is an odd number, therefore for odd parity rule, there is no parity error happening during the data transferring. Otherwise, the parity_error is set to 1 indicating that there is a parity error.

Verilog Code

```
/* Receiver */
/* Error Detector Module */

/* This module gets the data from the receiver data register to decide whether there is a
parity error or framing error. Odd parity is used for checking the parity error */

module ErrorDetectorRx
(
      data,
      parity_error,
      framing_error
);

// 9 bits: 1 stop bit + 1 parity bit + 7 data bits
input       [8:0]       data;

output                  parity_error;
output            framing_error;
```

```
/* if the stop bit is not 1, it indicates that there is a framing error upon transmission */
assign framing_error = (data[8] != 1'b1) ? 1'b1 : 1'b0;

/* if XOR the 8 bits (1 parity bit + 7 data bits) do not result a 1, that means that the
number of 1 in the data bits number is not odd number, which indicates that a parity error
happens */
assign parity_error = (^data[7:0] != 1'b1) ? 1'b1 : 1'b0;
endmodule
```

Simulation Results

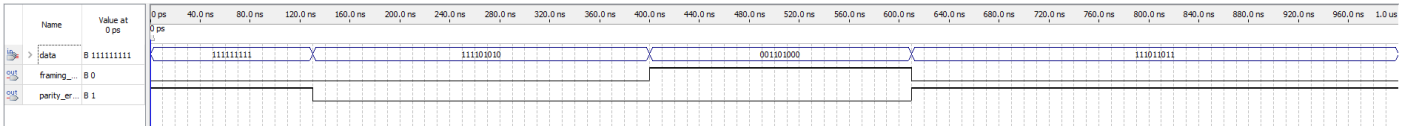

Figure 40. Receiver Error Detector Functional Simulation

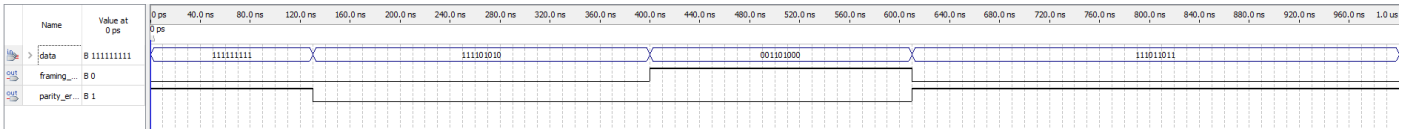

Figure 41. Receiver Error Detector Timing Simulation

The simulation indicates that the functionality of the error detector is correctly implemented. When the MSB of the data input is not logic 1, the framing_error output is then logic 1 suggesting that the stop bit is not correctly sampled. When the data contains a even number of bits, the parity_error output is then logic 1 suggesting that the parity bit is not successfully set or the data is transmitted incorrectly.

3.3.2 7-Segment Decoder

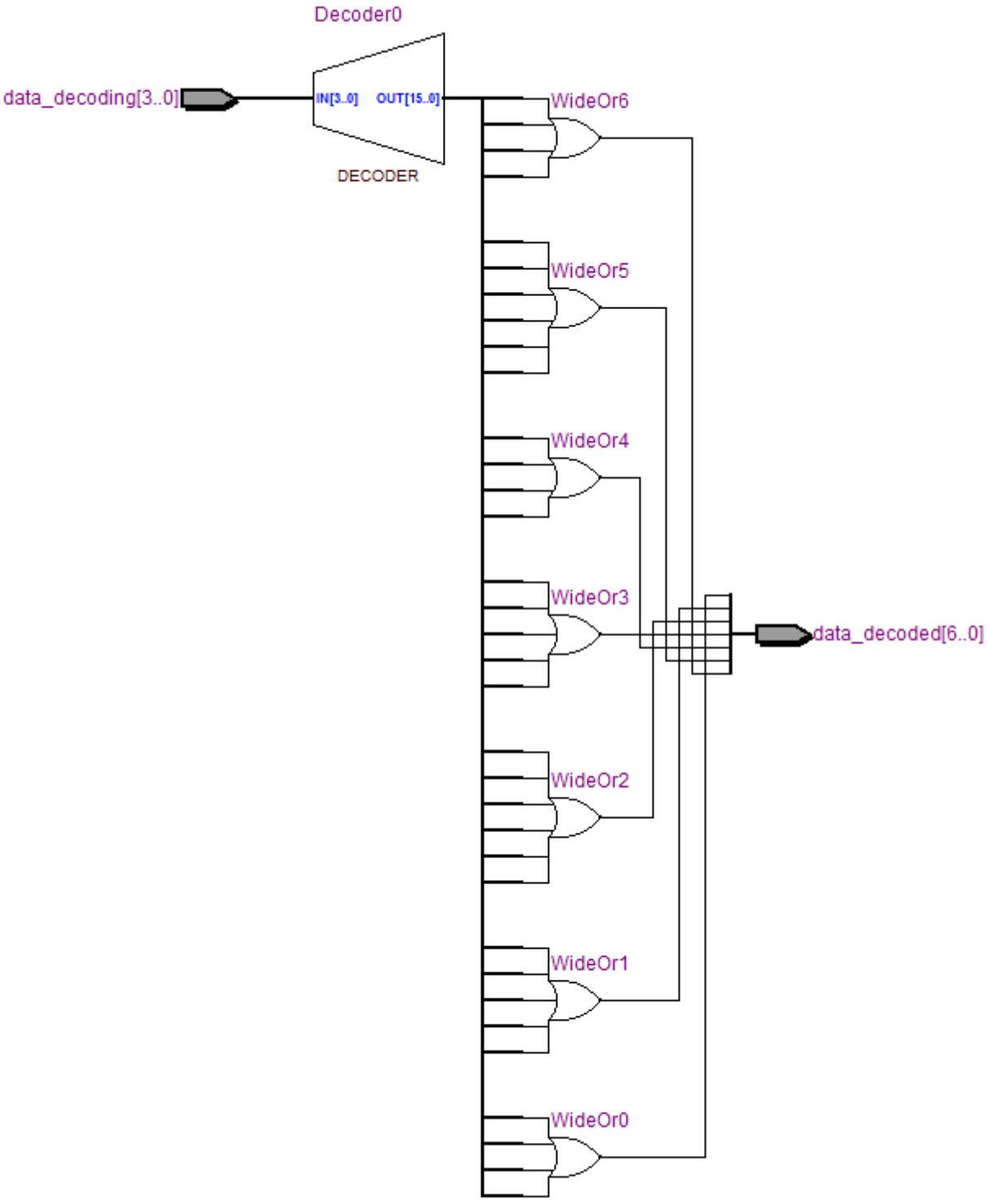

Figure 42. 7-Segment Decoder RTL View

The 7-segment decoder module is combinational logic module takes 1 input: data_decoding, which is the data needed decoding, and 1 output: data_decoded, which is the data that has been decoded.

ASM Chart

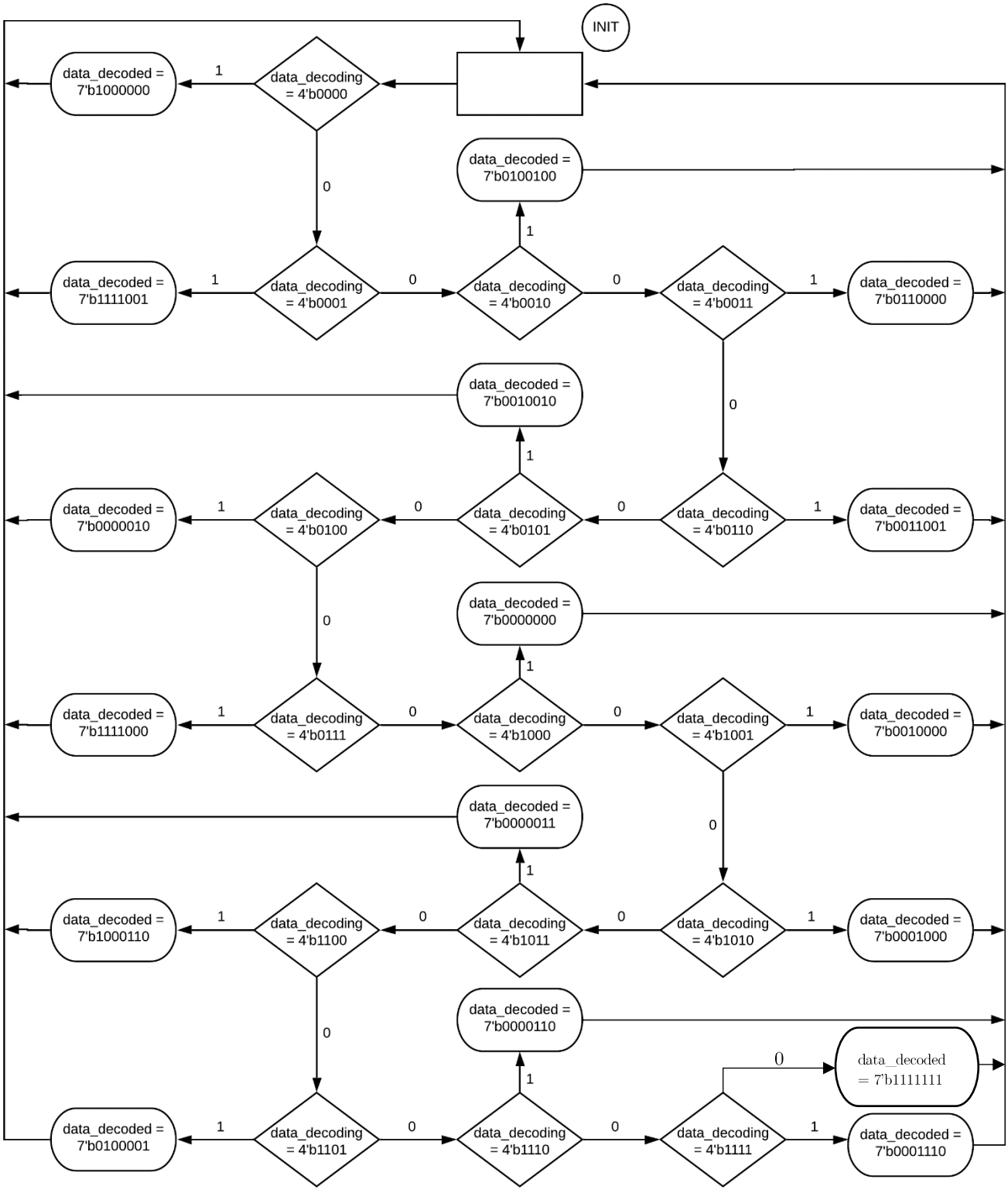

Figure 43. 7-Segment Decoder ASM

This ASM chart illustrates the what the decoded data should be based on the input data. Although in this ASM, if-else statements constitute the conditional logic, the case-statement is used in code implement since it is of higher readability.

Verilog Code

```
/* Receiver */
/* 7-Segment Decoder Module */

/* 7-segnment decoder which decodes the receiver data from the ASCII code to the
corresponding hexadecimal number and shows them on the 7-segment LED. */

module Decoder7SegmentRx
(
      data_decoding,
      data_decoded
);

// data need decodeing: half part of ASCII code of 4 bits
input       [3:0]       data_decoding;

// data decoded: 7 segment LED data: 7 bits
output      [6:0]       data_decoded;

reg         [6:0]       data_decoded;


always @(data_decoding)
begin
      case(data_decoding)
            // 0 for turn on the LED of the 7-segment
            4'b0000:    data_decoded = 7'b1000000;   // 0
            4'b0001:    data_decoded = 7'b1111001;   // 1
            4'b0010:    data_decoded = 7'b0100100;   // 2
            4'b0011:    data_decoded = 7'b0110000;   // 3
            4'b0100:    data_decoded = 7'b0011001;   // 4
            4'b0101:    data_decoded = 7'b0010010;   // 5
            4'b0110:    data_decoded = 7'b0000010;   // 6
            4'b0111:    data_decoded = 7'b1111000;   // 7
            4'b1000:    data_decoded = 7'b0000000;   // 8
            4'b1001:    data_decoded = 7'b0010000;   // 9
            4'b1010:    data_decoded = 7'b0001000;   // A
            4'b1011:    data_decoded = 7'b0000011;   // B
            4'b1100:    data_decoded = 7'b1000110;   // C
            4'b1101:    data_decoded = 7'b0100001;   // D
            4'b1110:    data_decoded = 7'b0000110;   // E
            4'b1111:    data_decoded = 7'b0001110;   // F
            default:    data_decoded = 7'b1111111;   // all lights turn off
      endcase
end
endmodule
```

Simulation Results

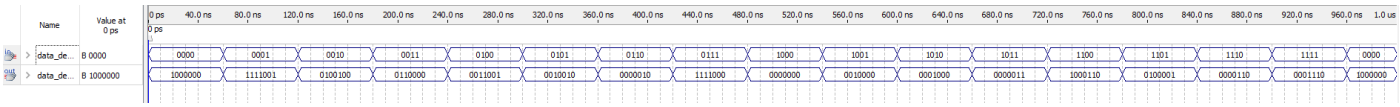

Figure 44. 7-Segment Decoder Functional Simulation

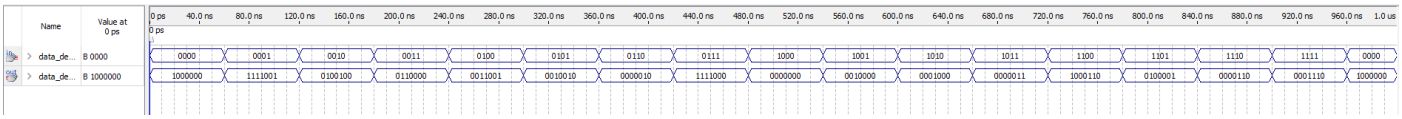

Figure 45. 7-Segment Decoder Timing Simulation

The simulation results substantiate that the decoder output the correct signal for the 7-segment LED to turn on appropriate lights based on the input number.

For instance, when the input number is binary number 1000 which is decimal number 8, the output signal will be binary 0000000, which turns on all the LED on the 7-segment, which is the decimal number 8.

3.3.3 Controller

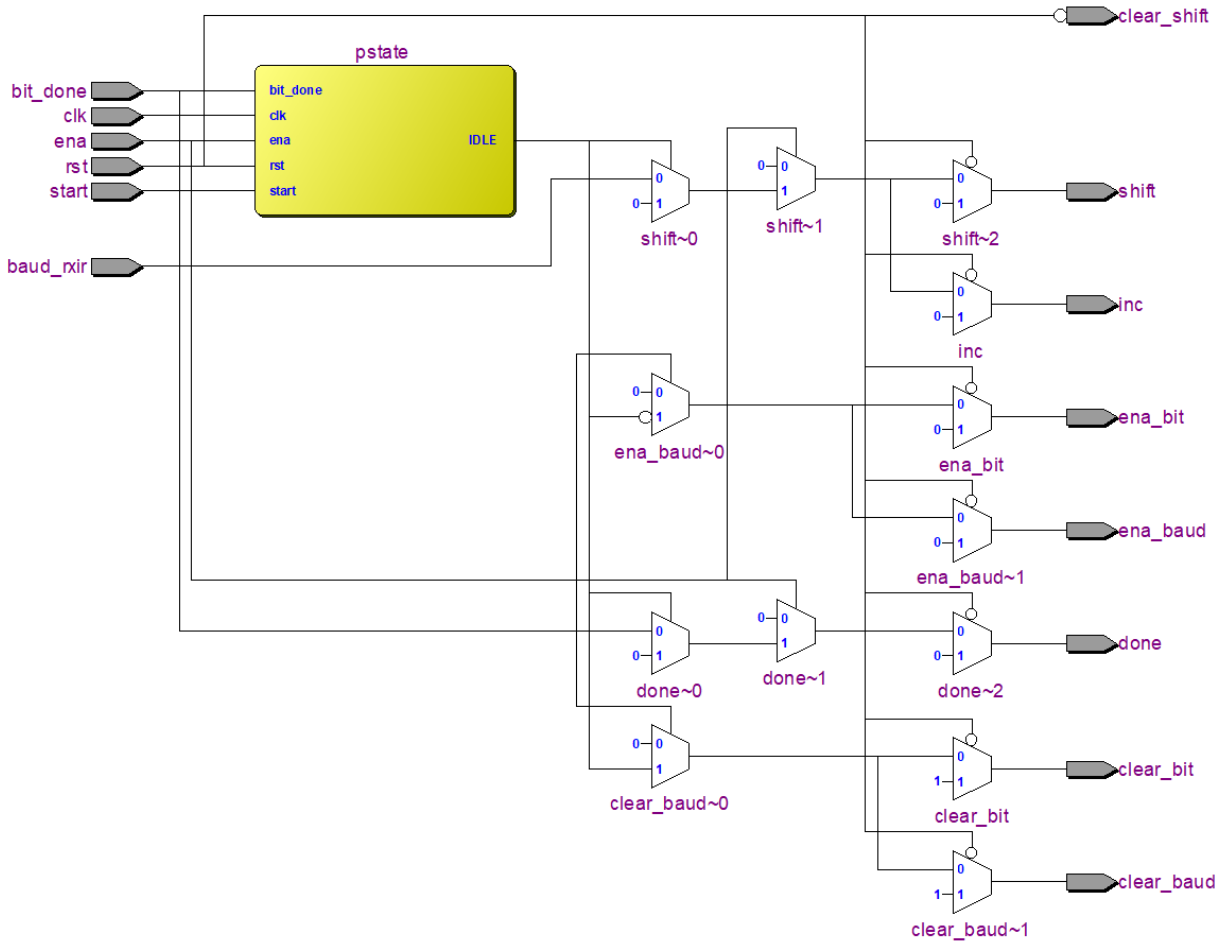

Figure 46. Receiver Controller RTL View

The ports of the controller of the receiver is identical to that of the transmitter despite it does not have the inverter control signal.

ASM Chart

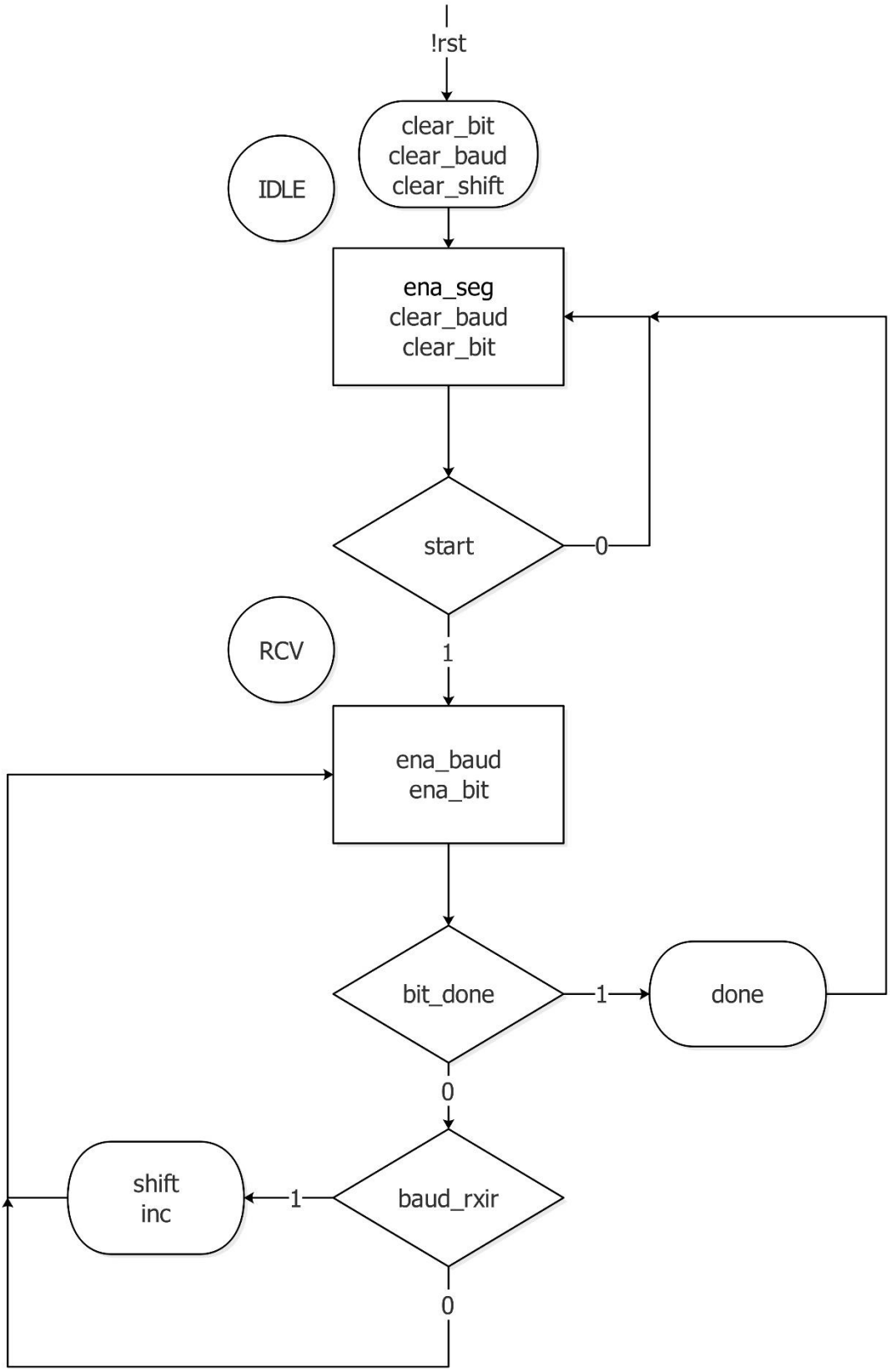

Figure 47. Receiver Controller ASM

State Transition Diagram

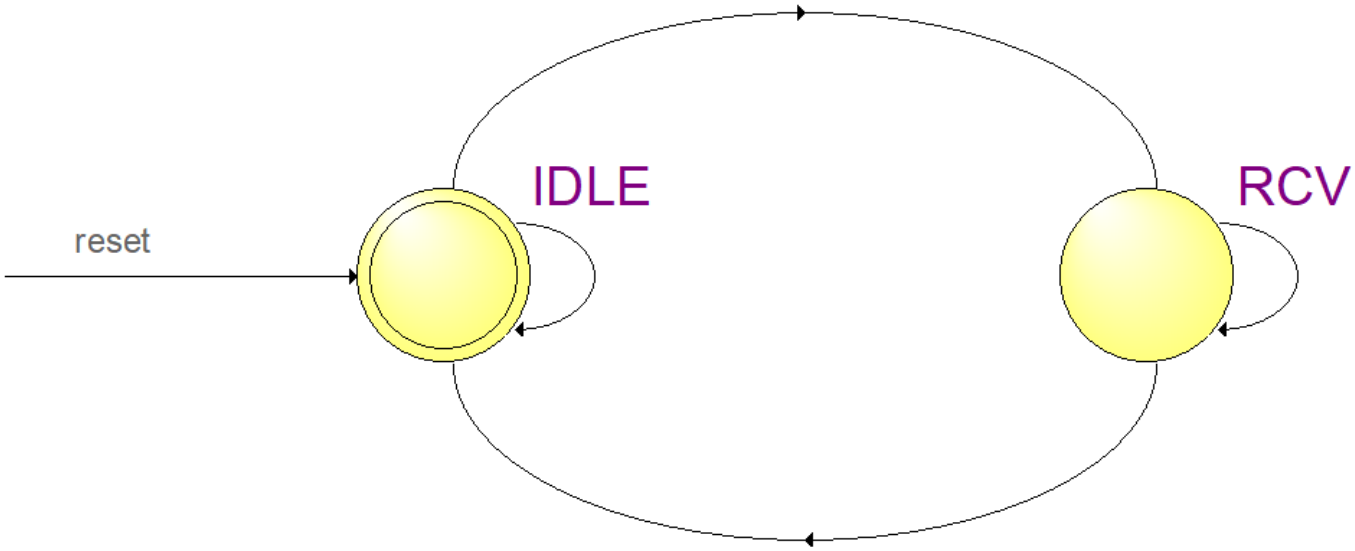

Figure 48. Receiver Controller State Transition Diagram

	Source State	Destination State	Condition
1	IDLE	RCV	(start).(ena).(rst)
2	IDLE	IDLE	(!start) + (start).(!ena) + (start).(ena).(!rst)
3	RCV	RCV	(!bit_done).(rst) + (bit_done).(!ena).(rst)
4	RCV	IDLE	(!bit_done).(!rst) + (bit_done).(!ena).(!rst) + (bit_done).(ena)

Figure 49. State Table

The receiver controller has 2 states: IDLE and RCV. A reset signal will always set the next state to the IDLE state no matter what the present state is. The controller begins with the IDLE state.

In the IDLE state, the receiver is ready to receive a set of data, and if there is no start bit detected, the state will remain in IDLE. As soon as a start bit of logic 1 is detected, the state moves to the next state, the RCV state.

In the RCV state, the receiver is receiving a set of data. The controller then enable the baud generator and the bit counter. At every pulse of the baud_rxir, the controller shifts the shift register and increment the bit counter. After ten bits of shifting, a bit_done signal should be detected from the bit counter and the receiver controller will send a done signal to indicate the complete of the receive. Then the control moves back to IDLE state.

Verilog Code

```
/* Receiver */
/* Controller Module */

/* Controller module for the receiver
which has 3 state: IDLE, RCV, and LOAD,
which indicates, respectively, the receiver
```

```
is idle, the receiver is receiving the data,
the receiver loads the data to the data register
for display. The controller output relative
signals to control the other module of the receiver */
module ReceiverController
(
	input		clk,
	input		rst,
	input		ena,
	input		start,
	input		bit_done,
	input		baud_rxir,
	output reg	shift,		// shift the shift register
	output reg	inc,			// increment the bit counter
	output reg	ena_baud,	// enable the baud generator
	output reg	ena_bit,		// enable the bit counter
	output reg	clear_baud,
	output reg	clear_bit,
	output reg	clear_shift,
	output reg	done
);

parameter IDLE = 2'b00;		// idle state, data are ready to be received
parameter RCV	= 2'b01;		// receive state, data are on the progress of
receiving
//parameter LOAD	= 2'b11;		// load state, data are displayed on the 7 segment
led

reg [1:0] pstate, nstate;

always @(posedge clk)
begin
	pstate <= nstate;
end

always @(pstate, rst, ena, start, bit_done, baud_rxir)
begin
	shift		= 1'b0;
	inc		= 1'b0;
	ena_baud	= 1'b0;
	ena_bit	= 1'b0;
	clear_baud	= 1'b0;
	clear_bit	= 1'b0;
	clear_shift	= 1'b0;
	done		= 1'b0;
	nstate	= pstate;	// default keep the present state in loop

	if(!rst) begin
		clear_baud	= 1'b1;
		clear_bit	= 1'b1;
		clear_shift	= 1'b1;
		nstate	= IDLE;
	end
	else if(ena) begin
		case(pstate)
			IDLE: begin
				clear_baud	= 1'b1;
				clear_bit	= 1'b1;

				if(start)
					nstate = RCV;// start to transmit when pushbutton is pressed
			end

			RCV: begin
				ena_baud = 1'b1;	// enable the baud generator
				ena_bit  = 1'b1;	// enable the bit counter

				if(bit_done) begin
```

```
// when 10 bits of data is counted, a frame of data is completed receiving
                                    done         = 1'b1;
                                    nstate       = IDLE;      // move to the IDLE state
                                end

                                if(baud_rxir) begin// shift and increment every baud tick
                                    shift = 1'b1;
                                    inc   = 1'b1;
                                end
                            end

                            default begin
                                nstate = IDLE;      // IDLE state by default
                            end
                        endcase
                end
        end
endmodule
```

Simulation Results

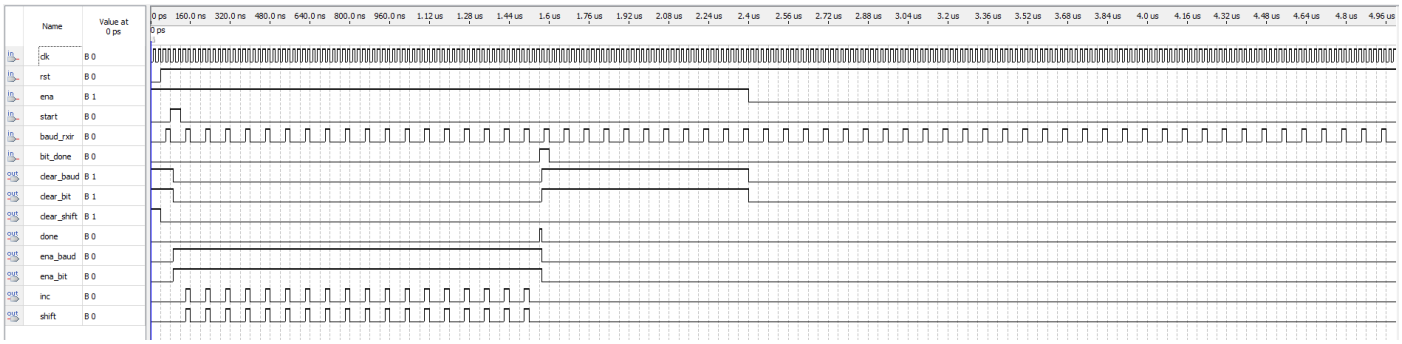

Figure 50. Receiver Controller Functional Simulation

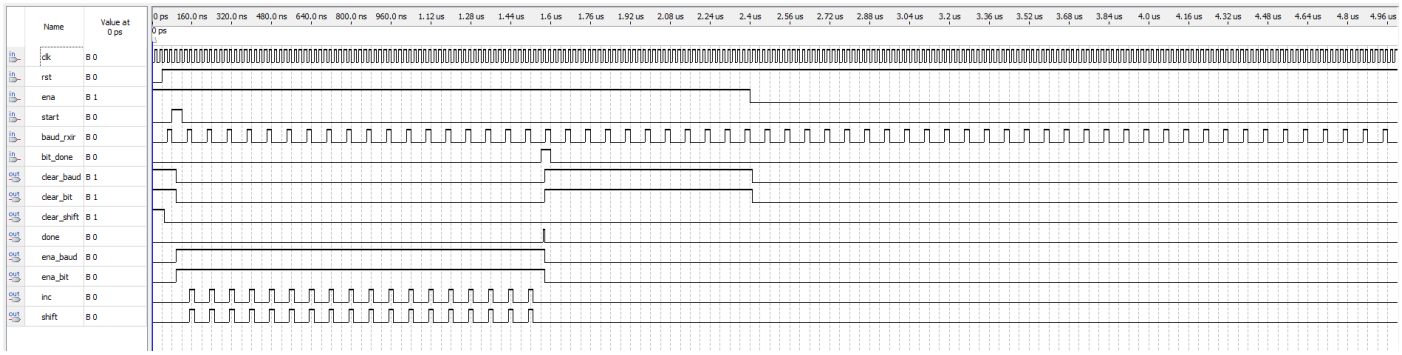

Figure 51. Receiver Controller Timing Simulation

Figurer 39 and 40 show the functional and timing simulation results of the receiver controller. It is worth noting that in the timing simulation, there is approximately a half of the clock cycle as the time delay, in spite of that, the logic of the signal input and output is identical to that of the functional simulation.

As long as the controller is enabled by the ena signal, a start signal is given as logic 1 and the controller will move from IDLE state to the RCV state, so that the receiver starts to receive the incoming data as the ena_baud and ena_bit are then set to logic 1 so that the baud generator and the bit counter are enabled, which means the shift register starts to shift the data and the bit counter starts to count.

When the controller receives a bit_done signal, it then disables both the baud generator and the bit counter so that the receive is completed.

4 Full System Simulation

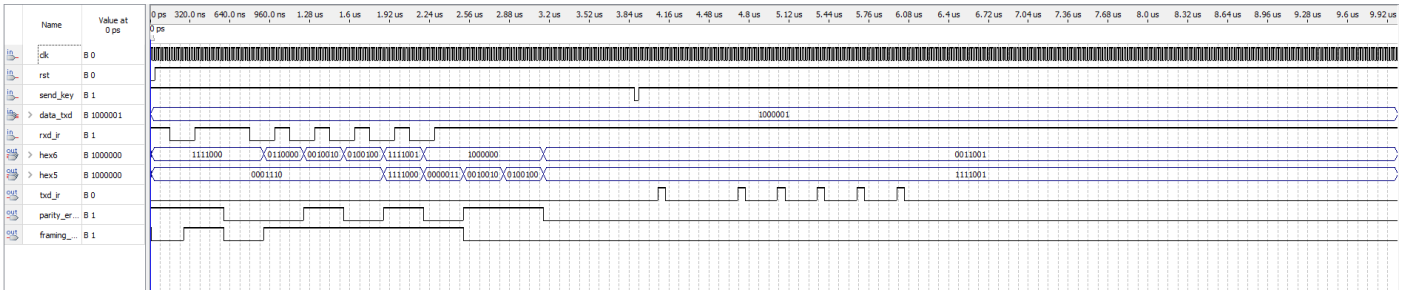

Figure 52. Full System Functional Simulation

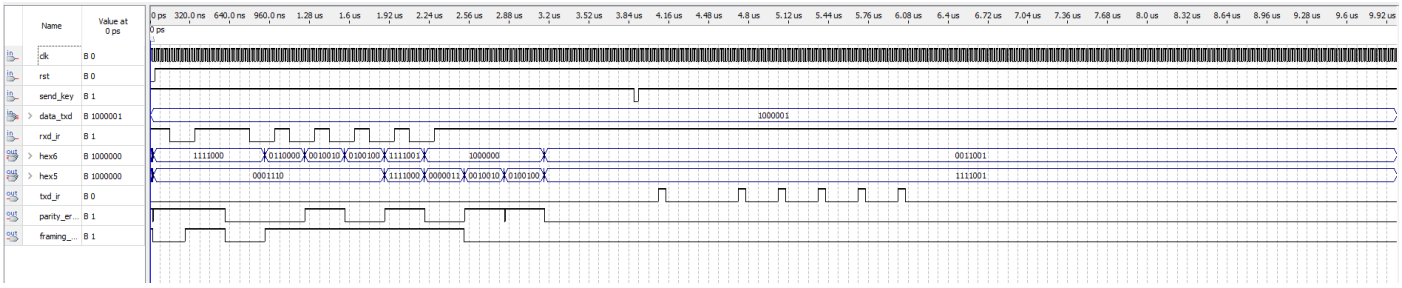

Figure 53. Full System Timing Simulation

The simulation shows that the transceiver works properly. It is worth noting that there is some small glitches appearing on the timing simulation. However, the presence of these glitches will not affect the accuracy of the transceiver system since its pulse width is far more less than the system clock.

The reset signal is firstly set to logic 0 so that all the modules are reset. The data inside the shift register is set to logic 1, which is the idle bits in RS232 protocol. As shown in the simulation, the hex6 and hex5 7-segment LED becomes '41' indicating the data bits are 1000001. It is worth noting that the MSB of the data is always 0 in order to properly decode according to the ASCII table.

The send_key signal is set to logic 0 so that the transmitter starts to load the input data and it to the tx port. The data_in_tx is 1011001 at the time it is loaded, so the output tx signal should be, from the LSB to MSB of the input and plus 1 start bit, 1 parity bit, and 1 stop bit: 0100110111. From 50 – 350ns, it can be seen that the output tx signal is just what is expected.

The rxd_ir signal can be read as 0 (start bit) + 1000001 + 1(parity bit) + 1(stop bit), and the data from MSB to LSB is 1110000010, thus the received data bits excluding start, parity, and stop bit, should be 1000001, which is 41 in hexadecimal. According to the output data of hex6 and data of hex5, the LED on the 7-segments show the character '4' and '1', which fit the received data.

Because the rxd_ir is the standard RS232 data which both the parity bit and the stop bit are correctly set, it can be noticed that the framing_error and the parity_error output is both logic 0 once the data is complete to be received. Therefore the error detector works fine for the receiver module.

5 Test Documentation

Both the transmitter and the receiver works properly when they are tested experimentally. The board which acts as the receiver can correctly receive the data from the boards acting as the transmitter and show the corresponding data on the 7-Segment LED. The boards that receive the data can still act as the transmitter and transmit data to the other data, and its received data will still stored in its shift register so that the LED turns on while its transmission.

It is worth noting that the LED of the parity error is turned on after reset the boards. This is because that after each reset, the shift register is loaded with 10-bit of 1s so that the parity bit is not correctly set according to the odd parity rule. However, after receiving its first set of data, the parity LED will works properly.

6 Conclusion

This report demonstrated the implement of a IrDA serial communication system implemented by Verilog. Each module of the transmitter and the receiver was demonstrated along with its ASM chart, code implementation with comments and the simulation results and discussions. The full system composed of the transmitter and the receiver was tested and works properly both in the simulation and on the DE2 board.

Further improvement can be on the controller for both transmitter and the receiver. Because both controllers only have few states, chances are that unexpected conditions might exist that the controller cannot handle which can cause the whole system to disrupt. In addition, the incipient appearance of logic 1 of the parity error is possible to be corrected by initiate the data register of the register with 7 bits of 1 and 1 bit of 0 as the parity bit.

ELEC373 2019/20 Assignment 2 Demonstrators Check Sheet

Student Name: Zhang, Junhao

Student ID No: 201377244 Module Level: UG

Demonstrator Name: J.S. Smith

Baud Rate: **38400** Parity: **Odd** Data Bits: 7 Inputs: **Sw[16:10]** Send: **Key2**

Output: **Hex6-5** ✓ Framing Error: **LEDG[2]** Parity Error: **LEDG[0]** Reset: KEY3

Top level Design: BDF or Verilog

Transmit Working: ✓

ASMs for Transmit:

Simulations?

Receive Working:

ASMs for Receive:

Simulations:

Fully Synchronous?

Comments on RTL View:

Students understanding of code:

Good

2 Needs to add 1 byte to rtx.

Code corresponding to ASMs:

Matches on 7 seg display.

Signature: J.S. Smith

Date/Time: 5/2/19