ELEC373 Assignment 1
Serial Communications

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.

Since a UART is asynchronous transmitting and receiving method, there is no clock signal to synchronise the transmitter and the receiver. Instead, a start bit and 1 – 2 stop bits are appended to the data frame for transferring. As long as the receiver detects a start bit distinguished from idle bits, it begins to sample the following bits at a specific baud rate. The baud rates of a transmitter and a receiver must be identical, otherwise all sampling bit will be displaced, and data will be received incorrectly.

In this assignment, it is required to design a UART that transmit the data from DE2 board to a computer terminal the receive the data from a computer to the DE2 board then display the data on the 7-segment LEDs in hexadecimal number. RS232 embarked on the DE2 board will be used to connect to a 9-pin connector, which is then connected to the computer. The data format in this assignment is shown in the table below:

Table 1. RS232 Data Format

Start bit	Binary 0
Data bits	7 bits
Parity bit	Odd parity
Stop bit	Binary 1
Idle bit	Binary 1

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 2. Data Frame Transmitting Character 'A'

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

2 Module Documentation

2.1 Transmitter

2.1.1 Baud Generator

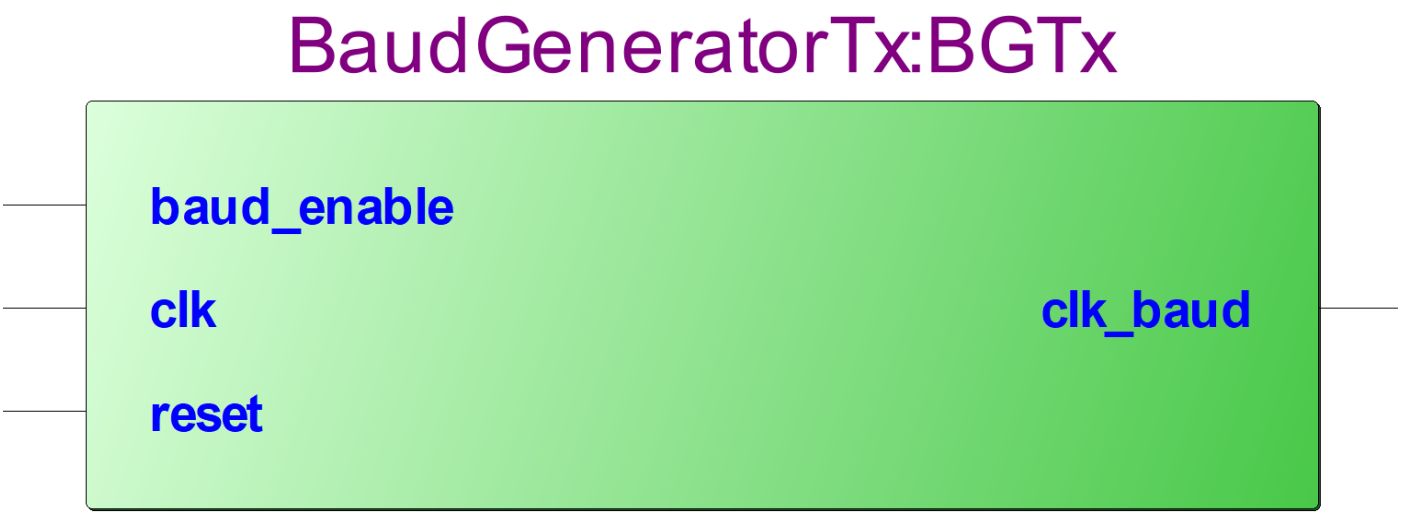

Figure 1. Transmitter Baud Generator

The baud generator module of the transmitter module consists of 3 inputs: baud_enable, which enables and disables the module; clk, which is the system clock; reset, which resets the module; and 1 output: clk_baud, which is the generated baud clock.

ASM Chart

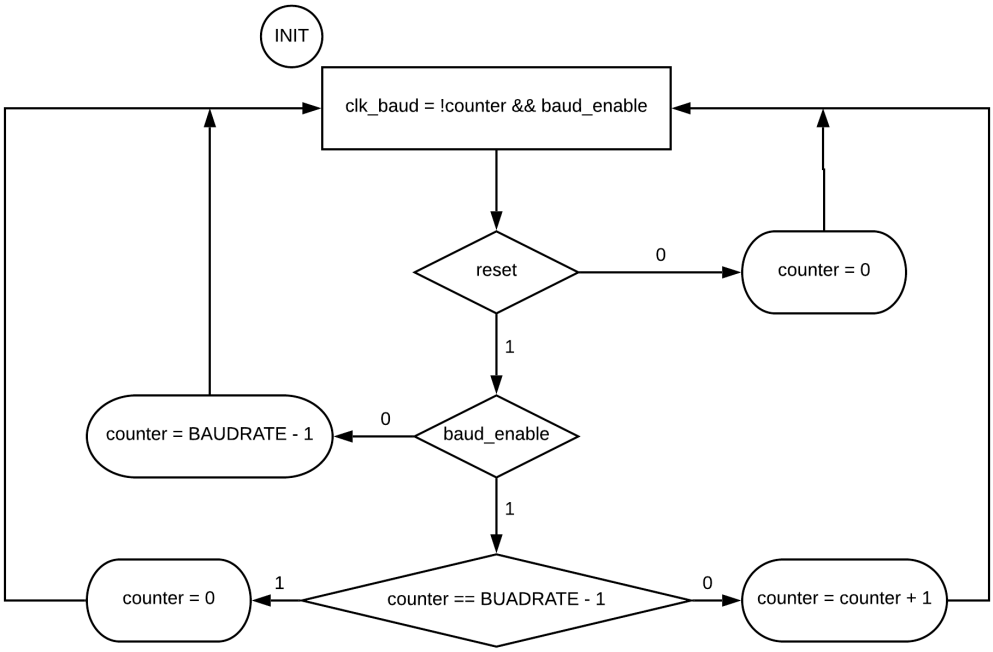

Figure 2. Transmitter Baud Generator ASM

When the module is not enabled, the counter is always set to the maximum value of the baud rate (1 less because starting from 0). Every time when the baud generator is enabled, the counter goes back to 0 and generate a single tick of the baud clock. Then the counter will count up to the maximum value again to generate the next clock tick.

Verilog Code

"BaudGeneratorTx.v"

```
/* Transmitter */
/* Baud Generator Module */

/* This module generator a baud clock which is based on the given baud rate and the
system clock which is 50MHz. For the transmitter, the baud clock doesn't need to be
delay for a half period. */

module BaudGeneratorTx
#(
      BAUDRATE
)
(
      clk,
      reset,
      baud_enable,
      clk_baud
);

input                   clk;
input                   reset;
input                   baud_enable;

output                  clk_baud;

// counter that counts up to 2^11 = 2048 because we need 1302 as the max number
```

```
reg         [10:0]      counter;


always @(posedge clk)
begin
      if(!reset)
            counter <= 11'b0;

      else if(baud_enable)
            // count up to max when enabled, if at maximum number, return to 0
            counter <= (counter == BAUDRATE - 1'b1) ? 1'b0 : counter + 1'b1;

      else
            /* when not enabled, the counter is set to maximum, preparing for the
next enable */
            counter <= BAUDRATE - 1'b1;
end

assign clk_baud = !counter && baud_enable;
endmodule
```

Simulation Results

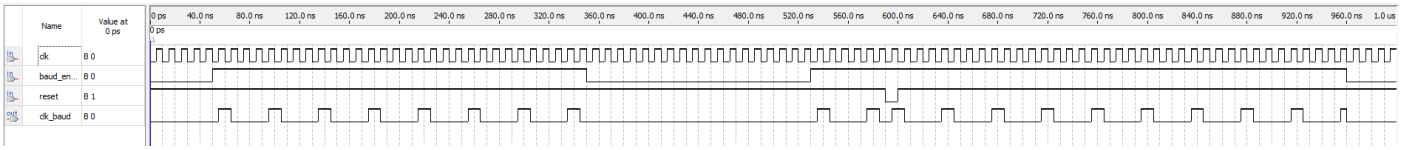

Figure 3. Transmitter Baud Generator Functional Simulation

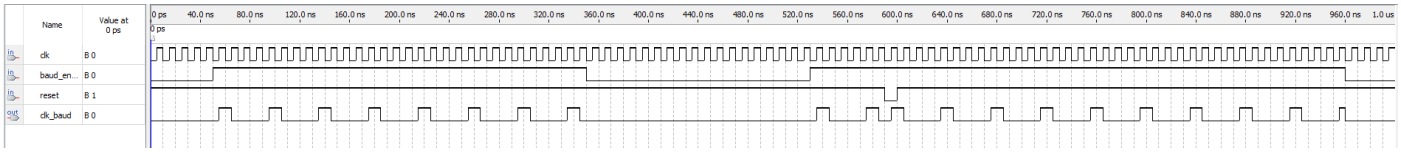

Figure 4. Transmitter Baud Generator Timing Simulation

For the convenience of the simulation, the baud rate is set to 4 times of one clock cycle, which means there is a baud clock generated every four system clock.

At 50ns, the baud_enable is set to logic 1 and the system clock reaches the first positive edge, the baud generator is enabled and starts to generate the baud clock in a consistent frequency, which is known as the baud rate.

At 350ns, the baud_enable signal is set to logic 0, therefore the baud generator is disabled and there is no output clock being generated.

At 590ns, the reset signal is set to logic 0 and the module is reset by set the internal counter back to 0. Thus, the output clk_baud generates immediately a new tick in spite of the unfinished previous clock.

2.1.2 Bit Counter

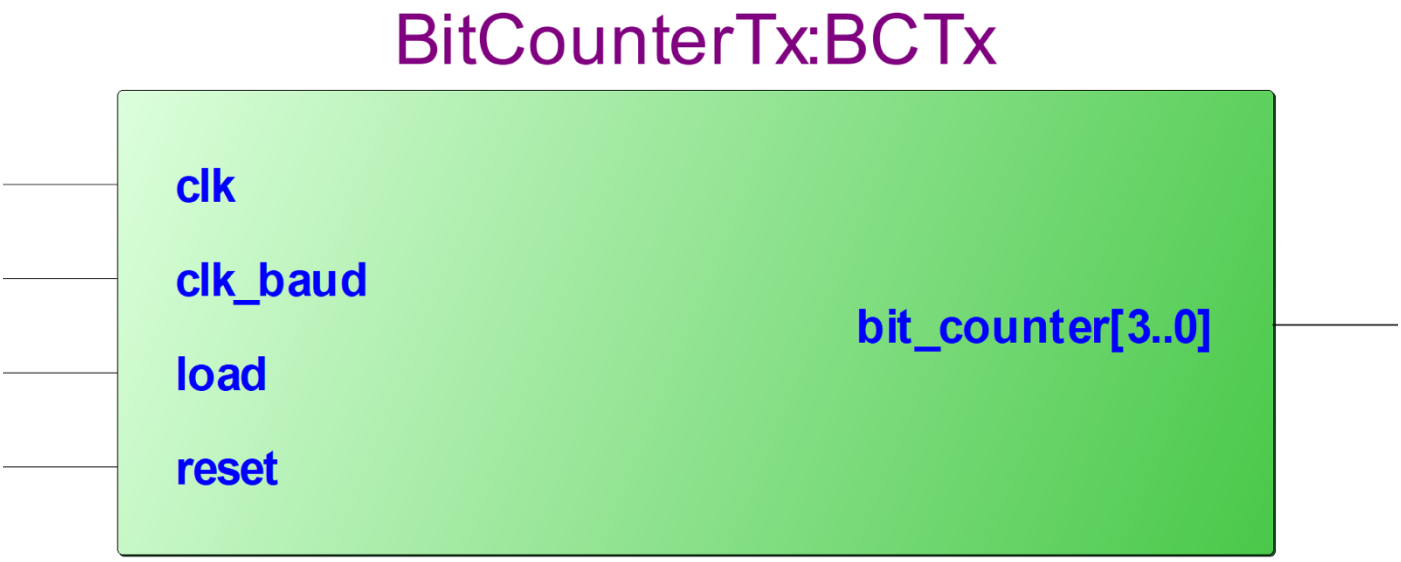

Figure 5. Transmitter Bit Counter

The bit counter module of the transmitter consists of 4 inputs: clk, which is the system clock; clk_baud, which is the baud clock; load, and reset, which both reset the bit counter; and 1 output: bit_counter, which is the counting number.

ASM Chart

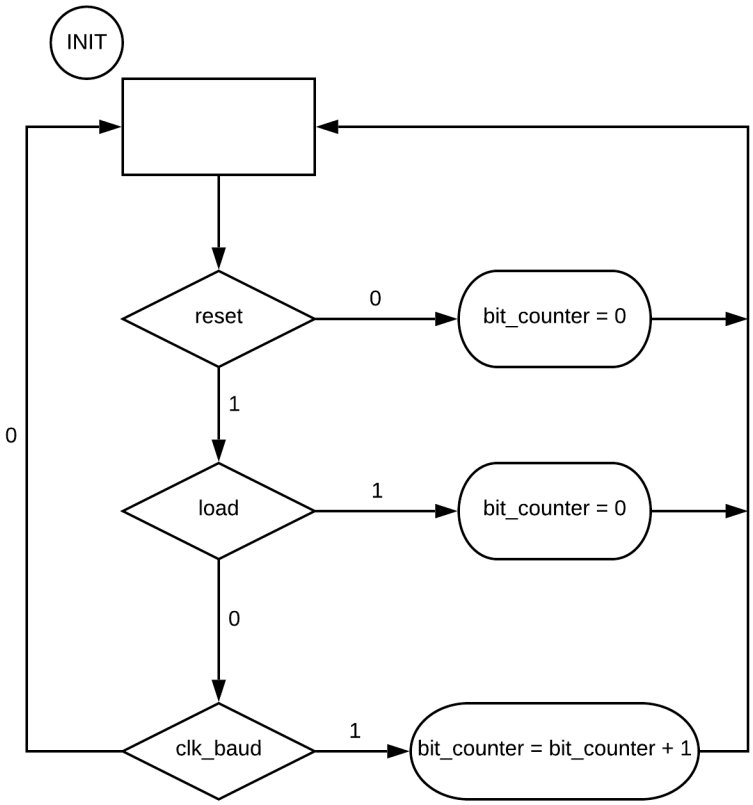

Figure 6. Transmitter Bit Counter ASM

The bit counter module has a counter with a increment of 1 every baud clock. Either logic 0 reset or logic 1 load signal should clear the counter. The difference is that the reset signal is given by an external input, but the load signal is passed by the interconnected controller.

Verilog Code

"BitCounterTx.v"

```
/* Transmitter */
/* Bit Counter Module */

/* This module counts the data bits that have been transmitted and output the counting
number to the transmitter controller to detect whether the
receiving process is completed. */

module BitCounterTx
(
	clk,
	clk_baud,
	reset,
	load,
	bit_counter
);

input			clk;
input			clk_baud;
input			reset;
input			load;

output	[3:0]	bit_counter;

reg		[3:0]	bit_counter;


always @(posedge clk)
begin
	if(!reset)
		bit_counter <= 4'b0000;

	else if(load)
		// start from 0 when data is loaded
		bit_counter <= 4'b0000;

	else if(!load && clk_baud)
		// increment of 1 at every baud clock
		bit_counter <= bit_counter + 1'b1;
end
endmodule
```

Simulation Results

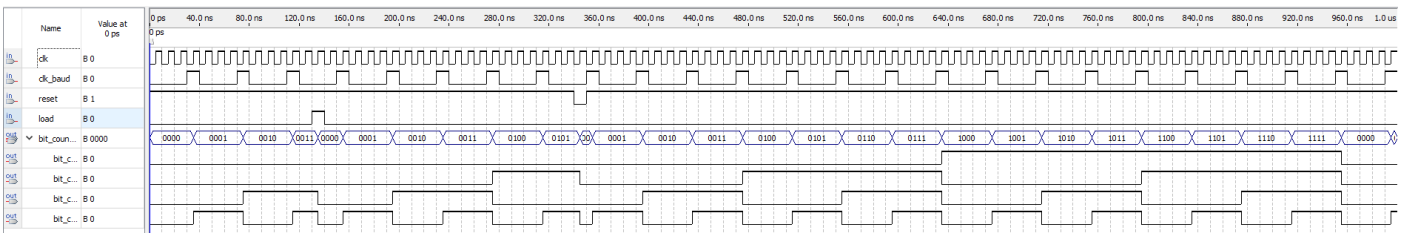

Figure 7. Transmitter Bit Counter Functional Simulation

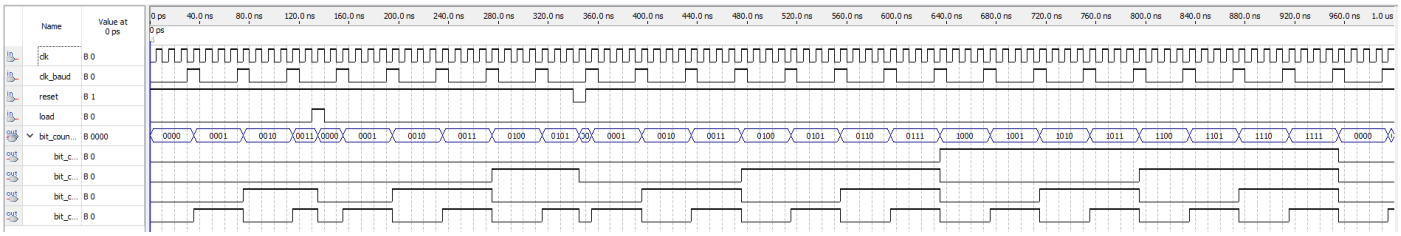

Figure 8. Transmitter Bit Counter Timing Simulation

As shown in the simulations, the counter increases by 1 each baud clock at the positive edge of the system clock. The identical results of timing simulation to the functional simulation indicates that no time delay exists.

At 130ns, the load signal is set to logic 1, therefore the counter is reset immediately and counts from 0 again.

At 340ns, the reset signal is set to logic 0, and it sets the counter to 0 as well as the load signal.

2.1.3 Shift Register

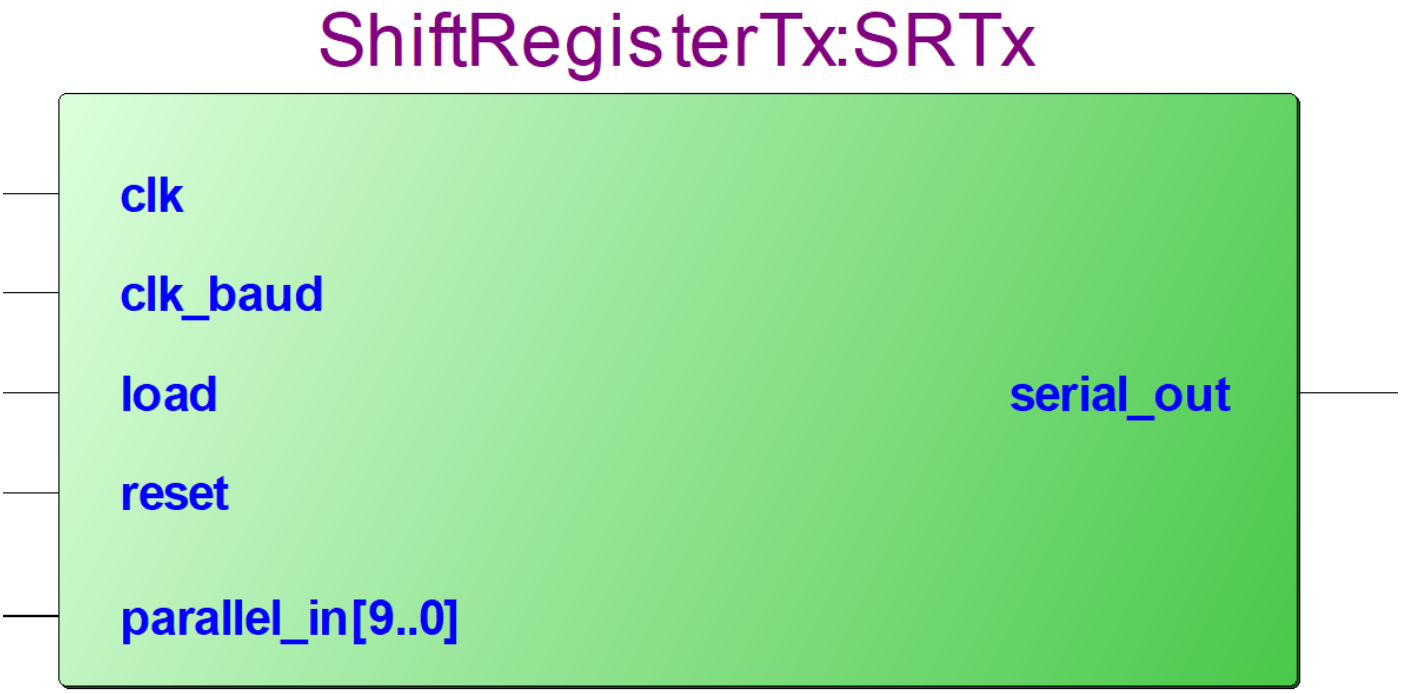

Figure 9. Transmitter Shift Register

The shift register of the transmitter is the type ofparallel-in-serial-out, where it loads a parallel of data altogether and shift them out one by one at a serial way. The shift register of the transmitter takes 5 inputs: clk, which is the system clock; clk_baud, which is the baud clock; load, which control the shift register to load the parallel data; reset, which reset the registered data in the shift register to default value; and 1 output: serial_out, which is the bit shifted out.

ASM Chart

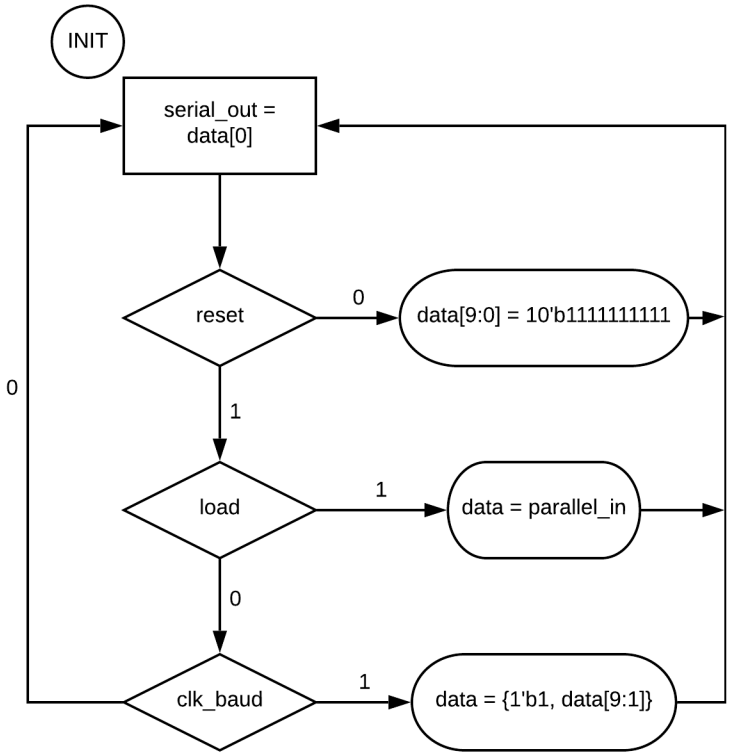

Figure 10. Transmitter Shift Register ASM

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.

Verilog Code

"ShiftRegisterTx.v"

```
/* Transmitter */
/* PISO Shift Register Module */

/* A Parallel In Serial Out shift register which takes a parallel of data bits from
tx and shift them out to the next module. The data will be shifted one by one. */

module ShiftRegisterTx
(
      clk,
      clk_baud,
      reset,
      load,
      parallel_in,
      serial_out
);

input                   clk;
input                   clk_baud;
input                   reset;
// the load signal which make the shift register load the data inside
input                   load;
```

```
input       [9:0]       parallel_in;

// output data 1 by 1
output                  serial_out;

reg         [9:0]       data;


always @(posedge clk)
begin
      if(!reset)
            data <= 10'b11_1111111_1;

      else if(load)
            // load the data from the input
            data <= parallel_in;

      else if(clk_baud)
            // shift the data to right at each baud clock
            data <= {1'b1, data[9:1]};
end

// assign the output as the LSB of the data
assign serial_out = data[0];
endmodule
```

Simulation Results

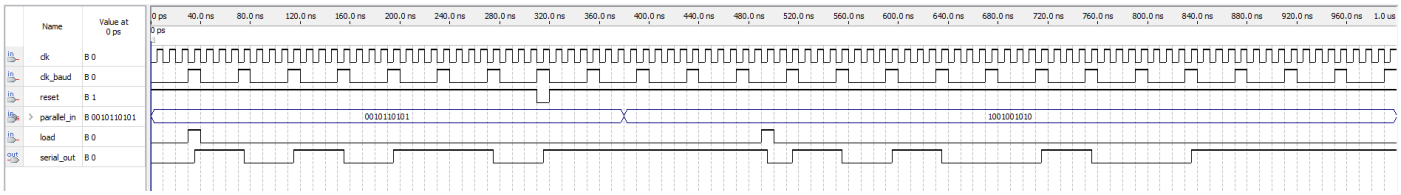

Figure 11. Transmitter Shift Register Functional Simulation

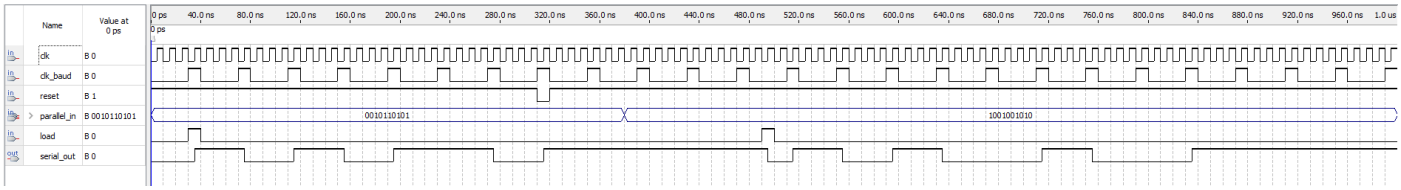

Figure 12. Transmitter Shift Register Timing Simulation

Both functional and timing simulation substantiate that the shift register is working as expected, with all input correctly control the module and there is barely time delay as shown in the timing simulation.

At 30ns, the load is set to logic 1, thus the shift register loads the data from the parallel data input b0010110101. With the baud clock always turned on, the data is shifted to the right at each positive level of baud clock, and the first bit shifted out is the least significant bit of the input data, which is 1, then 0, 1, 0, 1, 1, etc. as shown by the output serial_out.

At 310ns, the reset signal resets the registered data to all of 1s so that the output becomes consistent 1.

At 490ns, a logic 1 load signal loads the shift register with another set of data, and then the data is shifted to right. It is worth noting that despite that the first bit shifted out has not lasted for a complete baud clock period, the second bit is shifted out with the next baud clock. This is due to the

load signal is set to logic 1 at the negative baud clock. This effect might cause the least significant to be neglected or lost during the transmission. However, since the duty cycle of the baud clock is quite low at actual transmission, which is 1/1302 =0.07%, that transmission is of very low possibility to happen.

2.1.4 Data Register

Figure 13. Transmitter Data Register

The transmitter data register module takes 3 inputs: clk, which is the system clock; reset, which is the reset signal; data_in, which is a parallel of 8-bit data including 1 parity bit and 7 data bits; and 1 output: data_out, which is the same data as input data_in.

ASM Chart

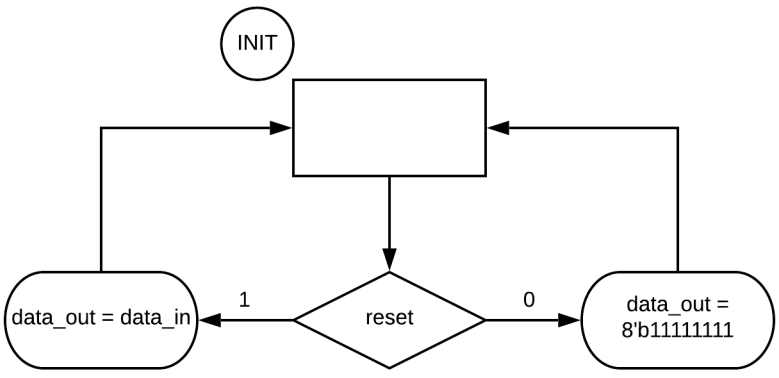

Figure 14. Transmitter Data Register ASM

As shown in Figure 14, the ASM chart indicates that a logic 0 of reset signal temporarily fill the data register with 8-bit 1, but as long as the reset goes to 1, the registered data will again the same as the input data.

Verilog Code

"DataRegisterTx.v"

```
/* Transmitter */
/* Data Register Module */

/* This module stores the transmitting data which comprises 7 bits that are necessary
to be handled: 7 data bits. The shift register can load the data from this module to
shift to the tx output. */

module DataRegisterTx
(
	clk,
	reset,
	data_in,
	data_out
);

input			clk;
input			reset;
input	[7:0]		data_in;

output	[7:0]		data_out;

reg	[7:0]		data_out;


always @(posedge clk)
begin
	if(!reset)
		data_out <= 8'b1_1111111;

	else
		data_out <= data_in;
end
endmodule
```

Simulation Results

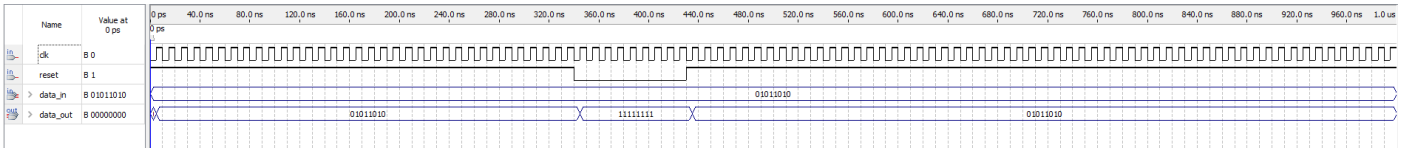

Figure 15. Transmitter Data Register Functional Simulation

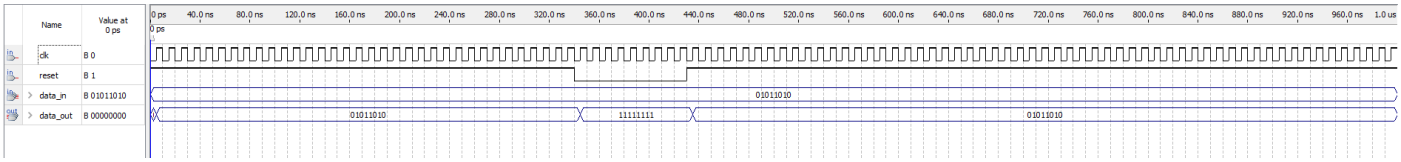

Figure 16. Transmitter Data Register Timing Simulation

At around 0ns, a positive edge of the system clock drives the data register to load and register the input data, and the output becomes the registered data constantly.

At 340ns, the data register receives a logic 0 reset signal, and the register is then filled with 1s as a result. However, as illustrated, as long as the reset becomes logic 1 again at 430ns, the data register registers the input data again at the first positive clock edge.

2.1.5 Parity Generator

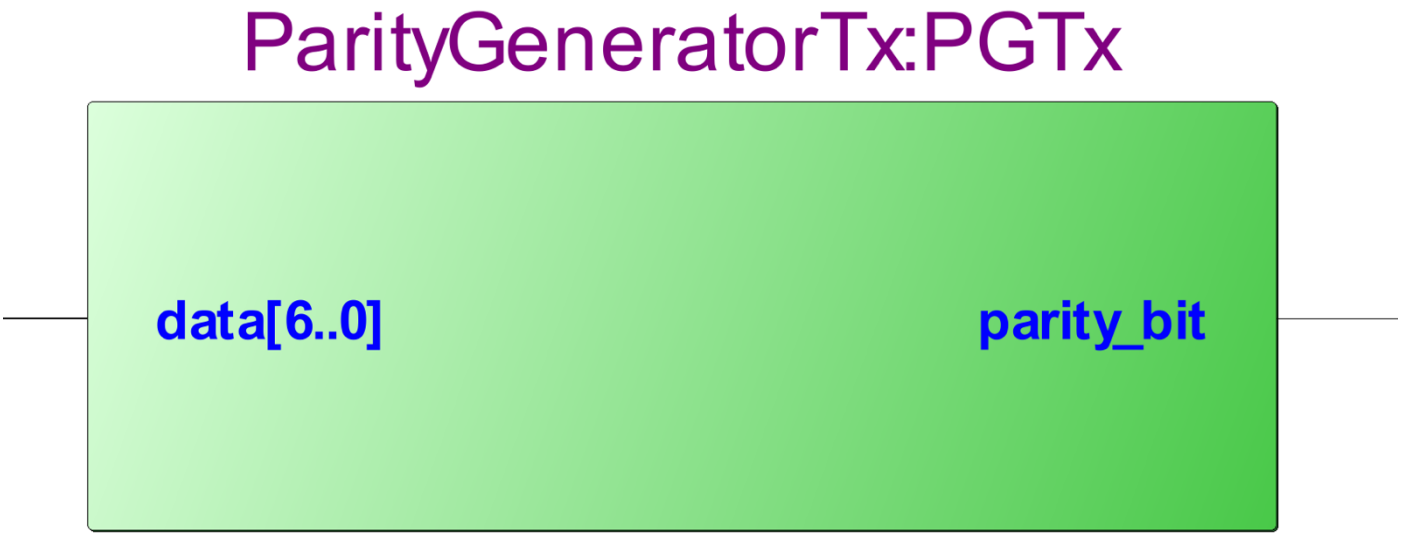

Figure 17. Transmitter Parity Generator

The parity generator module generates a parity bit based on the input data bits. Odd parity is followed in this design. The module takes 1 input: data, which is the 7-bit data bits, and 1 output: parity_bit, which is the generated parity bit. This module is combinational logic, thus there is no clock system needed to synchronise itself.

ASM Chart

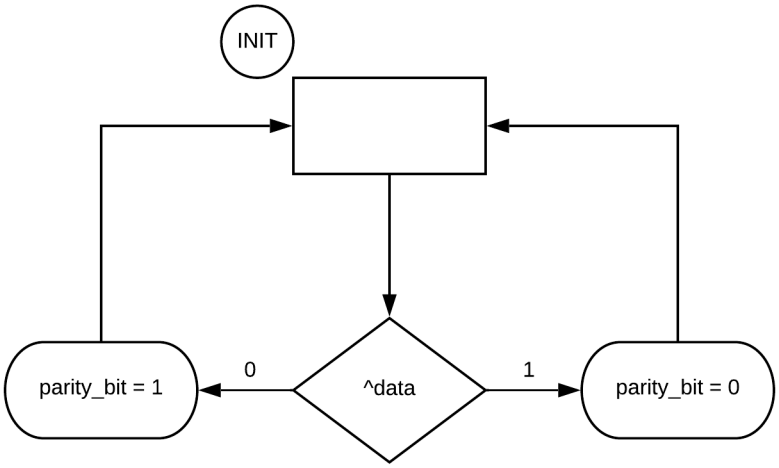

Figure 18. Transmitter Parity Generator ASM

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

"ParityGeneratorTx.v"

```
/* Transmitter */
/* Parity Generator Module */

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

module ParityGeneratorTx
(
      data,
      parity_bit
);

input       [6:0]       data;

output                  parity_bit;

/* 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 */
assign parity_bit = !(^data);
endmodule
```

Simulation Results

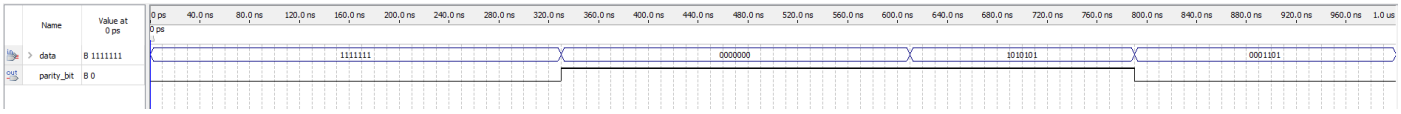

Figure 19. Transmitter Parity Generator Functional Simulation

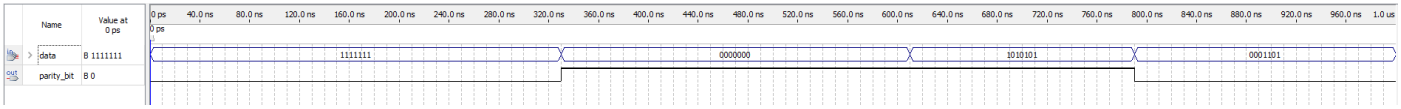

Figure 20. Transmitter Parity Generator Timing Simulation

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.

2.1.6 Controller

Figure 21. Transmitter Controller

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 takes 4 inputs: clk, which is the system clock; reset, which resets the state of the controller; start, which is the signal to transmit the data; bit_counter, which is the signal to stop transmitting; and 2 outputs: baud_enable, which controls the baud generator module; load, which control the bit counter module and shift register modules.

ASM Chart

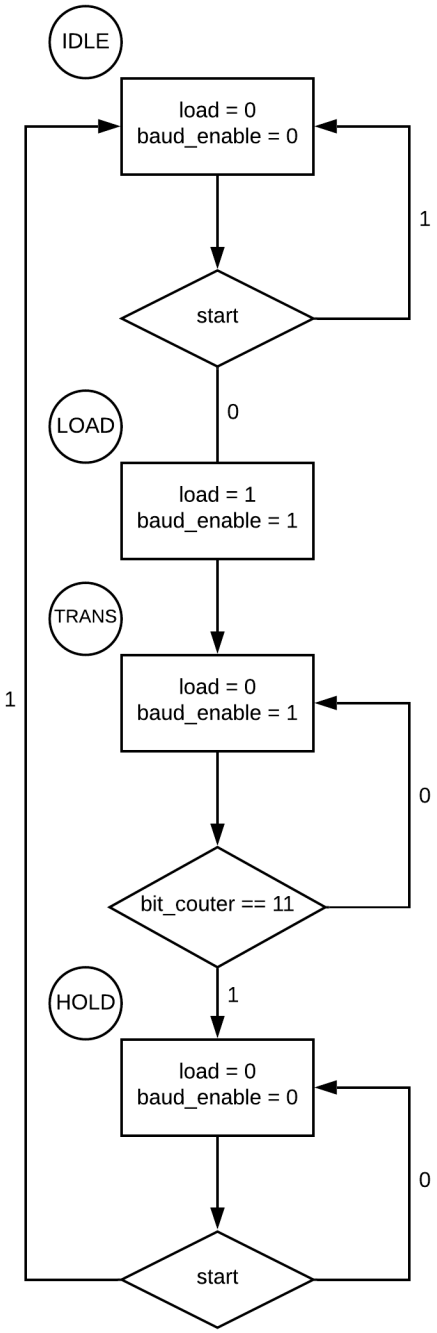

Figure 22. Transmitter Controller ASM

State Transition Diagram

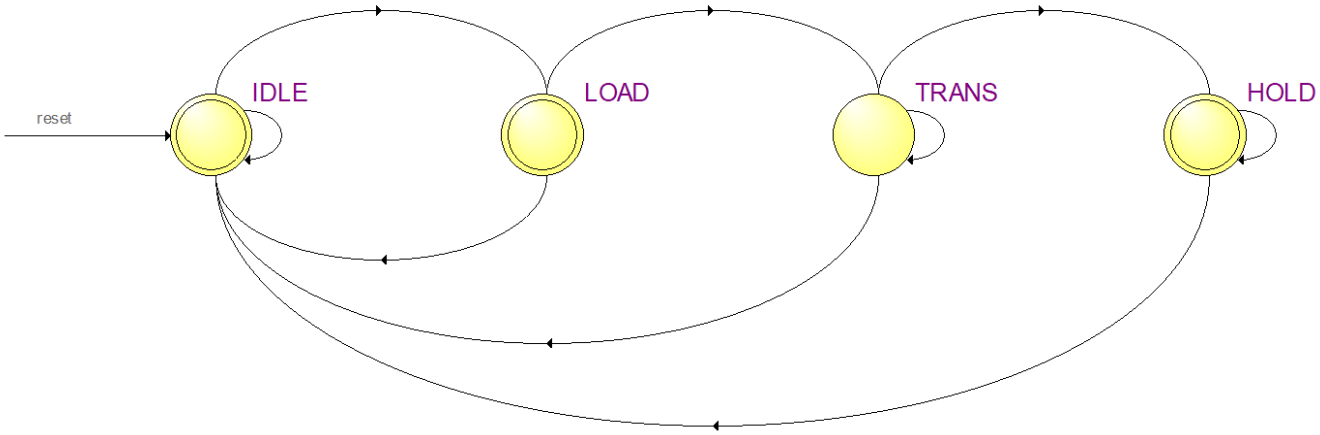

Figure 23. Transmitter Controller State Transition Diagram

As illustrated in Figure 22 and 23, the transmitter controller has 4 states: IDLE, LOAD, TRANS, and HOLD. 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 where the load and baud_enable equals 0.

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 send bit of logic 0 is detected, the state moves to the next state, the LOAD state.

In the LOAD state, the controller set both the load and the baud_enable signal to 1, therefore, the shift register, and the baud generator will start to operate. 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. The current loaded data will be transmitted according to the baud rate generated by the baud generator. The transmitting process will complete, and the controller will move to the next state once the bit counter counts up to the necessary value, otherwise the transmission will keep going on.

In the HOLD state, both the load and the baud_enable signal turns off, but the difference from IDLE state is that in the HOLD state the controller does not try to detect a send signal, in contrast, it detects a inversed send signal, which actually will be generate by releasing the pushbutton. This will ensure that the transmitter only send 1 set of data after pushing the key and the next set of data will only be sent after the pushbutton is released and pushed again.

Verilog Code

“ControllerTx.v”

```
/* 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 ControllerTx
(
	clk,
	reset,
	start,
	bit_counter,
	load,
	baud_enable
);

// idle state, where data is ready to be transmitted
parameter		IDLE  = 2'b00;
// load state, where data is loaded
parameter		LOAD  = 2'b01;
// transmitting state, where data is transmitted
parameter		TRANS = 2'b10;
// hold state, hold for next transmit when the pushbutton is released
parameter		HOLD = 2'b11;

input			clk;
input			reset;
input			start;		// connect to pushbutton to send data
input	[3:0]	bit_counter;

output			load;
output			baud_enable;

reg				load;		// load data to the shift register
reg				baud_enable;	// enable the baud generator
reg		[1:0]	pstate;
reg		[1:0]	nstate;


// state machine
always @(posedge clk)
begin
	if(!reset)
		pstate <= IDLE;

	else
		pstate <= nstate;
end

always @(pstate, start, bit_counter)
begin
	// default keep the present state in loop
	nstate		= pstate;

	// default for load disabled and baud generator disabled
	load		= 1'b0;
	baud_enable = 1'b0;

	case(pstate)
		IDLE:
		begin
			// start to transmit when pushbutton is pressed
			if(!start)
				nstate	= LOAD;
		end

		LOAD:
		begin
			// load the data to the shift register
```

```
                load              = 1'b1;
                // enable the baud generator

                baud_enable       = 1'b1;
                // move to the TRANS state
                nstate            = TRANS;
            end

            TRANS:
            begin
                // enable the baud generator
                baud_enable = 1'b1;

                /* when 11 bits of data is counted, that means 1 set of data is
completed transmitting */
                if(bit_counter == 4'd11)
                    // move to the HOLD state
                    nstate      = HOLD;
            end

            HOLD:
            begin
                if(start)
                    // move to the IDLE state when pushbutton is released
                    nstate      = IDLE;
            end

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

Simulation Results

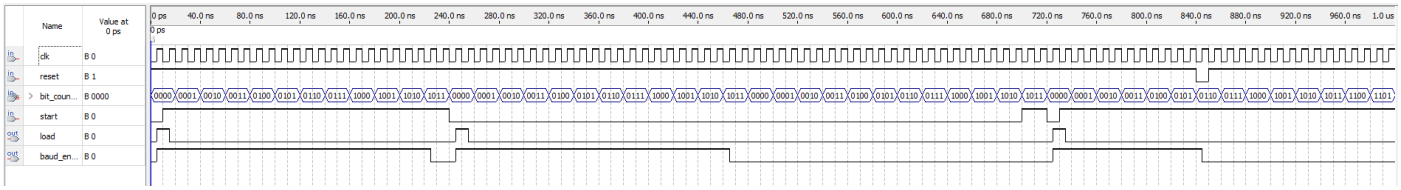

Figure 24. Transmitter Controller Simulation

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.

At around 0ns, a start signal is given as logic 0 and the controller will move from IDLE state to the LOAD state, where the load baud_enable signal is logic 1 as shown in Figure 24. At next positive edge of the clock, the controller move to the TRANS state and the load signal is set to logic which means the shift register no longer load in any new data.

At 220ns, the bit counter counts to binary 1011, which makes the controller goes to HOLD state where it tries to detect a logic 1 of start, then it moves to IDLE state again.

At 240ns, the simulation indicates the fact that if the send button is not released after sending 1 set of data, the next set of data will not be send and the load and baud_enable signal will be 0 until the pushbutton is released and a logic 1 of start signal is detected.

At 840ns, a reset signal resets the controller, so the controller immediately goes to IDLE state and the baud_enable signal becomes logic 0 again.

2.2 Receiver

For modules of the receiver, there is only a few differences compared to those of the transmitter. Those differences will be illustrated at necessary places.

2.2.1 Baud Generator

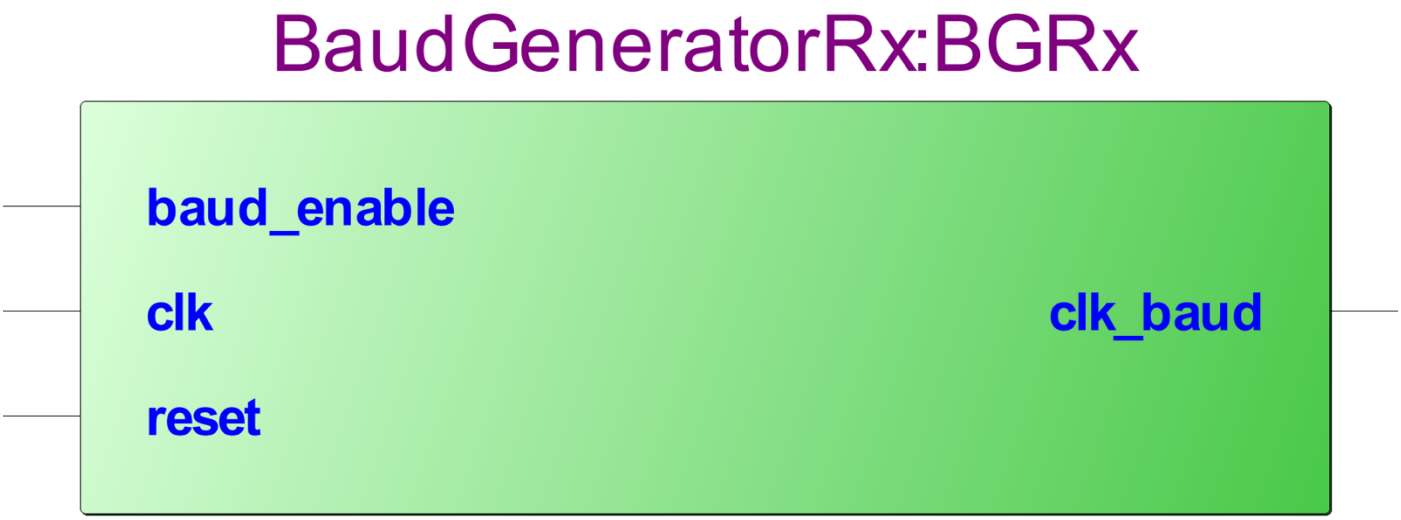

Figure 25. Receiver Baud Generator

ASM Chart

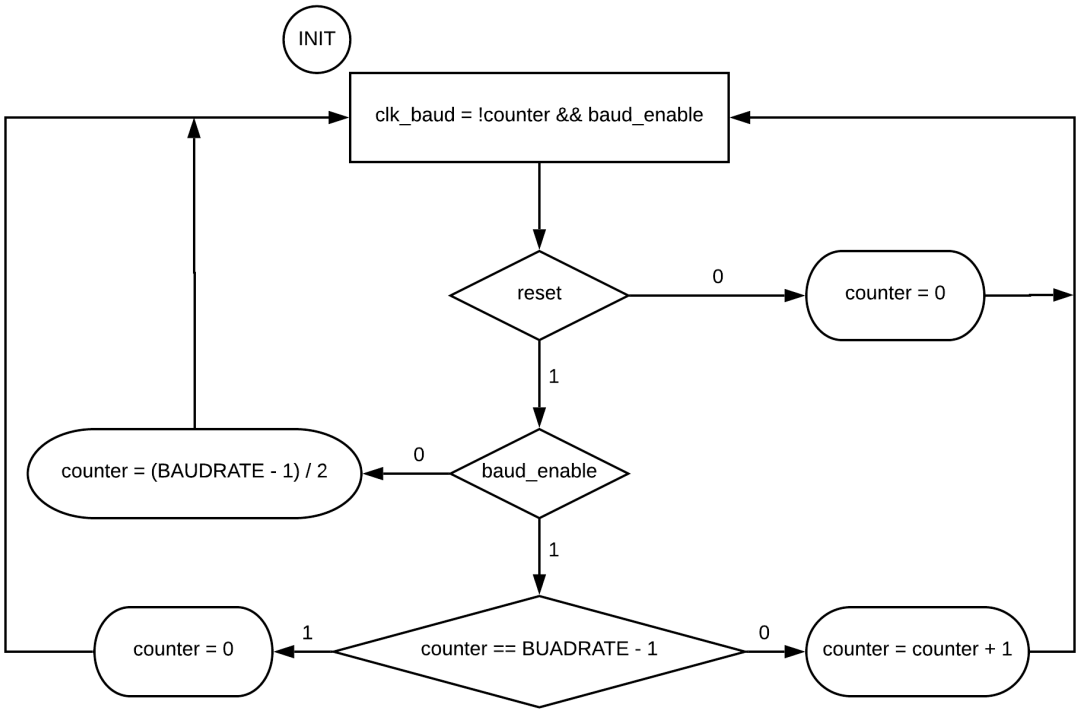

Figure 26. Receiver Baud Generator ASM

Different from the transmitter, the baud generator holds the value of half of the maximum baud rate value when it is not enabled. Therefore, when it receives a enable signal, the baud generator will not immediately output a baud tick, instead, its counter will continue to increase until it reaches the maximum value.

This change causes the baud generator generates a baud clock that is delayed by half period of the baud rate. The reason is that for the receiver, the SIPO shift register should not register the incoming data immediately at every baud clock cycle, which can cause sampling error as the signal is at metastable state. Instead, it should sample the data at its middle way.

Verilog Code

"BaudGeneratorRx.v"

```
/* Receiver */
/* Baud Generator Module */

/* This module generator a baud clock which is based on the given baud rate and the
system clock which is 50MHz. For the receiver, the baud clock is necessary to be
delay for a half period*/

module BaudGeneratorRx
#(
      BAUDRATE
)
(
      clk,
      reset,
      baud_enable,
      clk_baud
```

```
);

input                   clk;
input                   reset;
input                   baud_enable;

output                  clk_baud;

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


always @(posedge clk)
begin
      if(!reset)
            counter <= 11'b0;

      else if(baud_enable)
            // count up to max when enabled
            counter <= (counter == BAUDRATE - 1'b1) ? 1'b0 : counter + 1'b1;

      else
            // counter start at half to delay the baud clock for a half period
            counter <= (BAUDRATE - 1'b1) / 2'd2;
end

assign clk_baud = !counter && baud_enable;
endmodule
```

Simulation Results

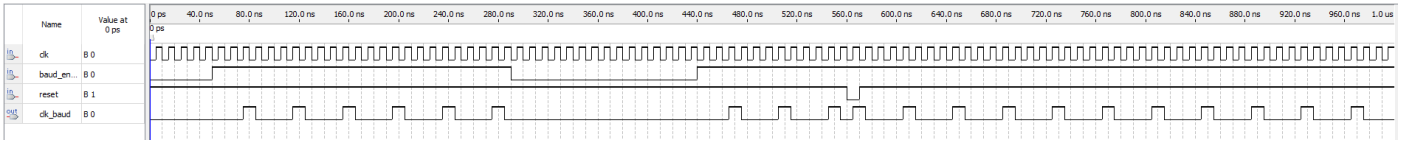

Figure 27. Receiver Baud Generator Functional Simulation

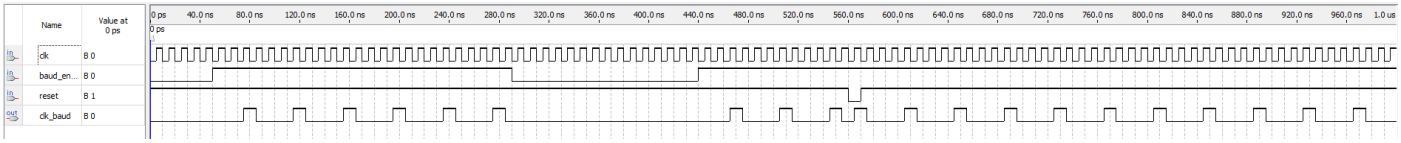

Figure 28. Receiver Baud Generator Timing Simulation

For convenience of the simulation, the baud rate is set to 4 times of 1 clock cycle. As the simulation shows, the baud_enable signal is set to logic 1 at 50ns, and the clk_baud output the first clock at 70ns. This indicates that the output clk_baud does not generate the first clock tick when the baud_enable signal is set to logic 1, instead, half of the baud period is delay until the first baud tick is generated.

2.2.2 Bit Counter

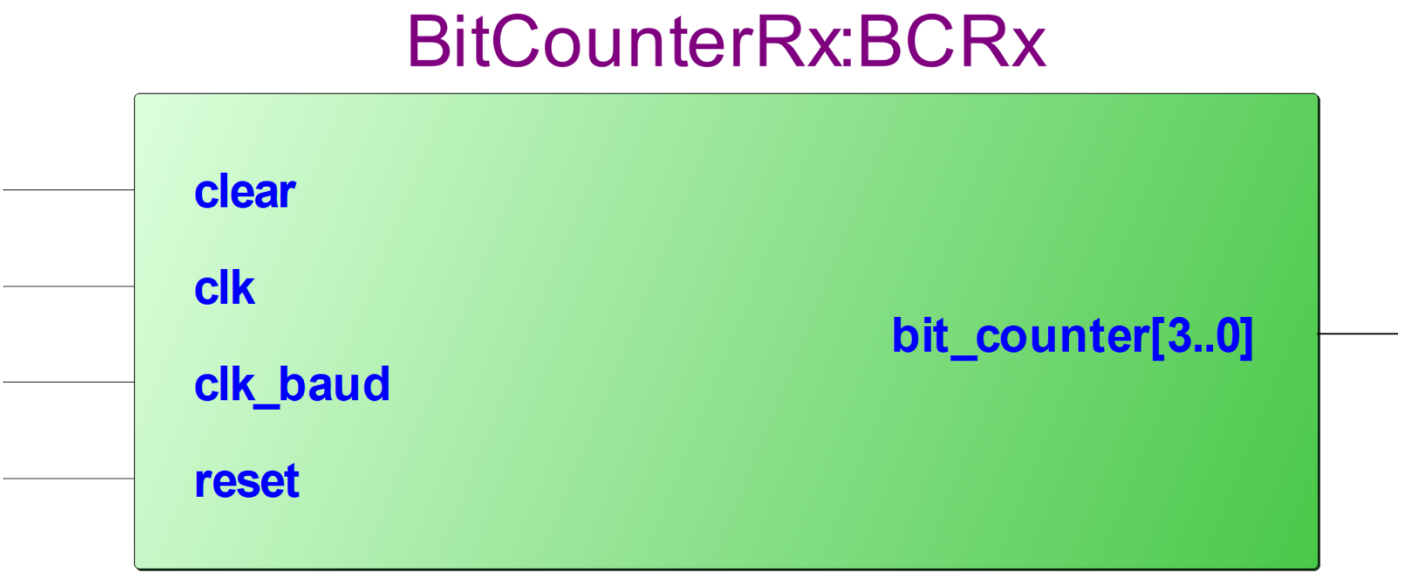

Figure 29. Receiver Bit Counter

Instead of a load signal, the bit counter receives a clear signal which clear the counter to 0 as well as the load signal in the transmitter.

ASM Chart

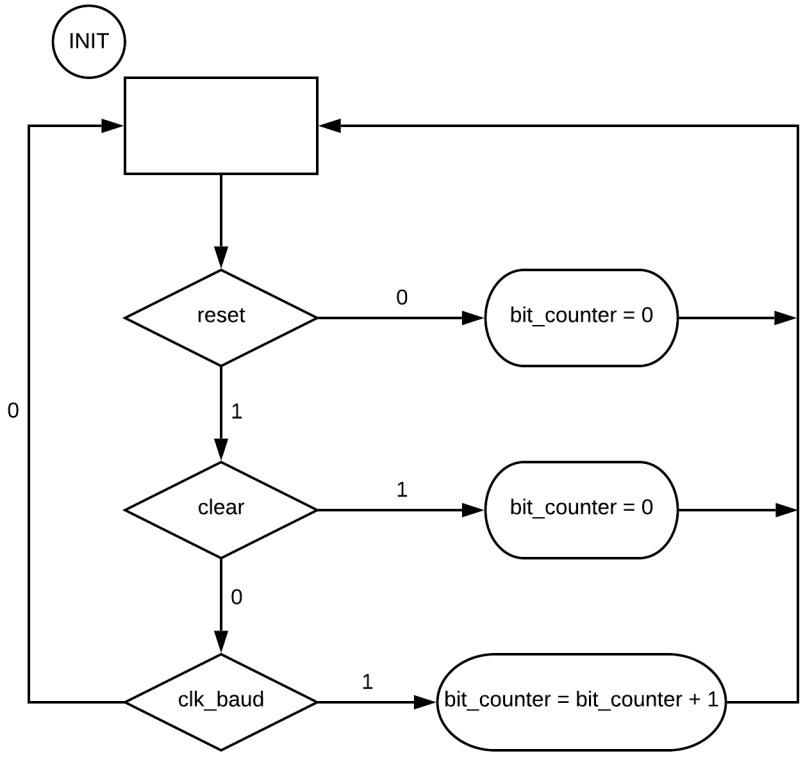

Figure 30. Receiver Bit Counter ASM

As long as the clear input is 1, the bit_counter will not counting up instead have a constant 0 registered in the counter. This is useful for the receiver because the bit counter should not count up when the receiver is not receiving any data.

Verilog Code

"BitCounterRx.v"

```
/* Receiver */
/* Bit Counter Module */

/* This module counts the data bits that have been received and output the counting
number to the receiver controller to detect whether the receiving process is completed.
*/

module BitCounterRx
(
	clk,
	clk_baud,
	reset,
	clear,
	bit_counter
);

input			clk;
input			clk_baud;
input			reset;
input			clear;

output	[3:0]	bit_counter;

reg		[3:0]	bit_counter;


always @(posedge clk)
begin
	if(!reset)
		bit_counter <= 4'b0000;

	else if(clear)
		// reset the bit counter when it receives a clear signal
		bit_counter <= 4'b0000;

	else if(!clear && clk_baud)
		// increment by 1 at every baud clock
		bit_counter <= bit_counter + 1'b1;
end
endmodule
```

Simulation Results

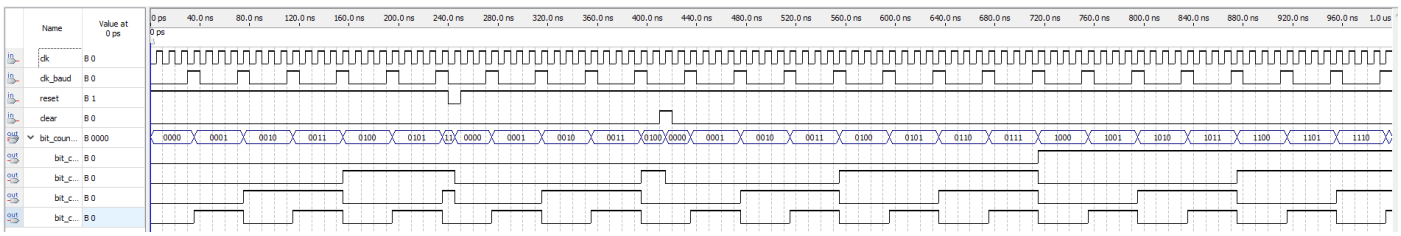

Figure 31. Receiver Bit Counter Functional Simulation

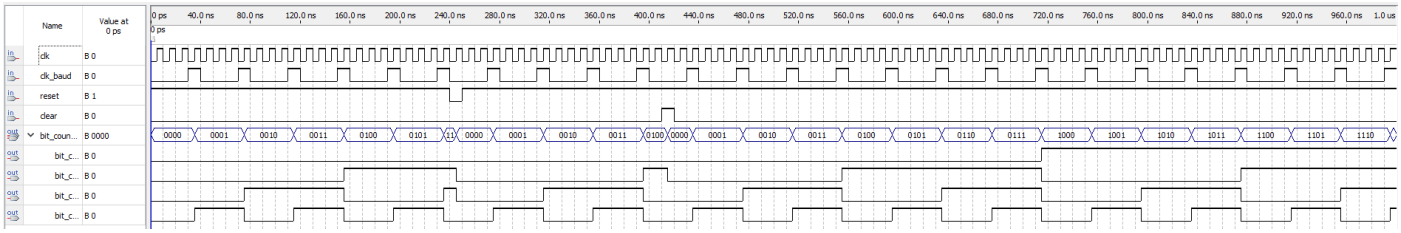

Figure 32. Receiver Bit Counter Timing Simulation

Same as the bit counter in the transmitter, the clear and reset signal set the bit counter to 0, in spite of what number of the counter it currently holds.

2.2.3 Shift Register

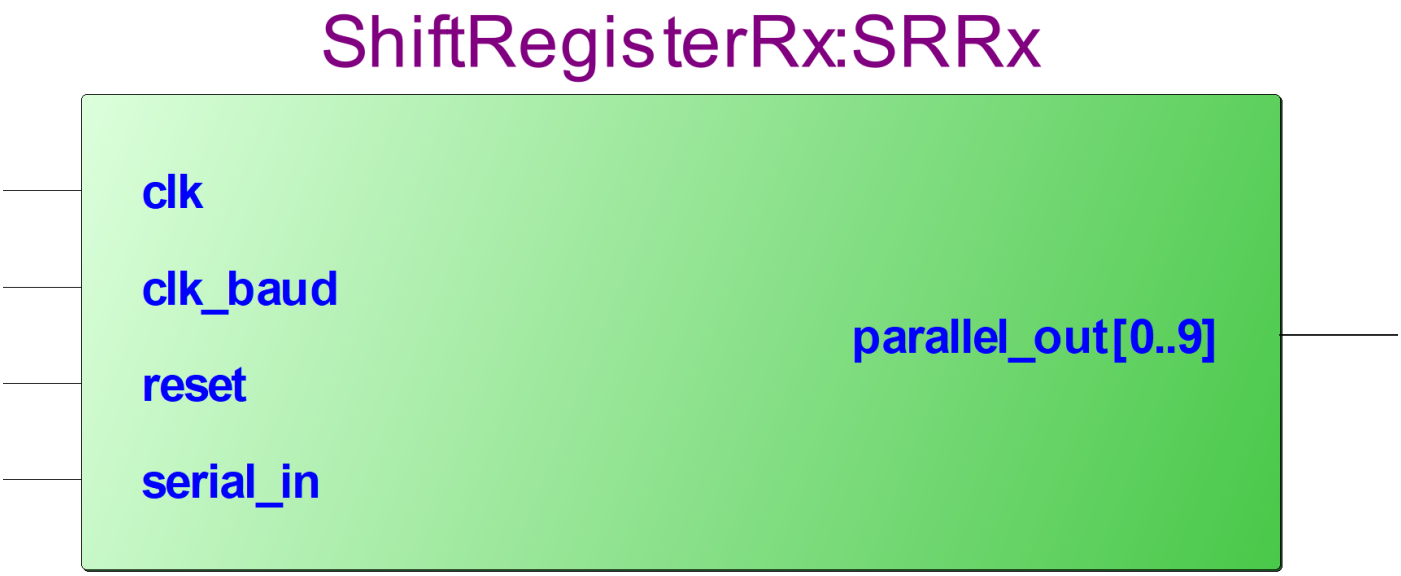

Figure 33. Receiver Shift Register

The shift register of the receiver is the type of serial-in-parallel-out, which means it receives data bits in serial way, and output a set of data parallelly.

ASM Chart

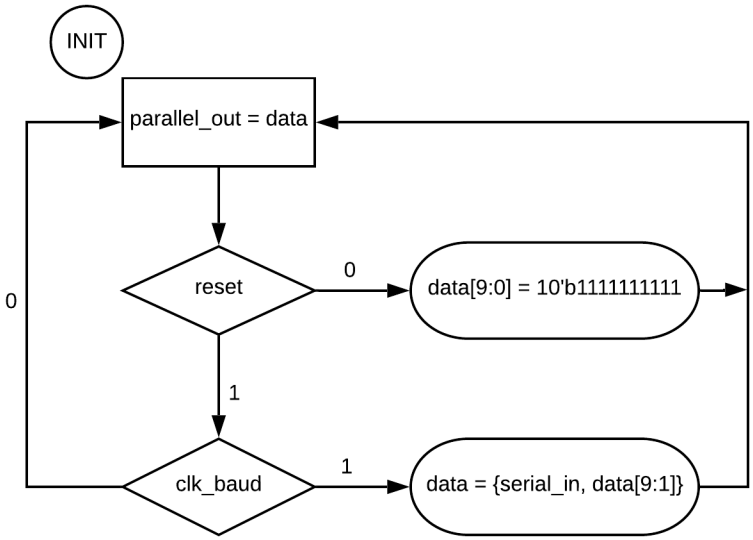

Figure 34. Receiver Shift Register ASM

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

"ShiftRegisterRx.v"

```
/* Receiver */
/* SIPO Shift Register Module */

/* A Serial In Parallel Out shift register which take a serial of data bits from rx
and shift them into the register. Once all the data are received, they will be output
at once. */

module ShiftRegisterRx
(
      clk,
      clk_baud,
      reset,
      serial_in,
      parallel_out
);

input                   clk;
input                   clk_baud;
input                   reset;
input                   serial_in;

output      [9:0]       parallel_out;

reg         [9:0]       data;


always @(posedge clk)
begin
      if(!reset)
```

```
            data <= 10'b11_1111111_1;

      else if(clk_baud)
            // shift the data to right
            data <= {serial_in, data[9:1]};
end

// assign the output as the total set of data bits
assign parallel_out = data;
endmodule
```

Simulation Results

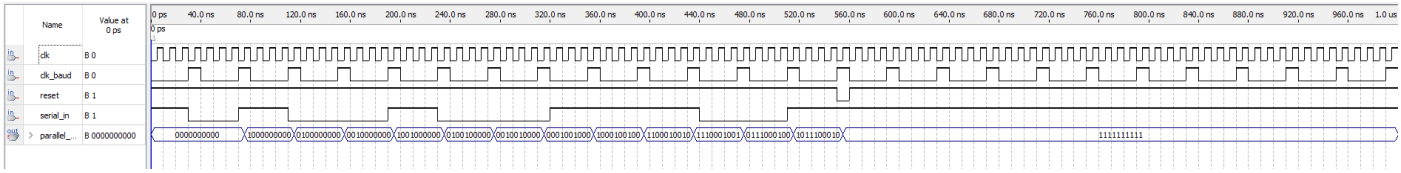

Figure 35. Receiver Shift Register Functional Simulation

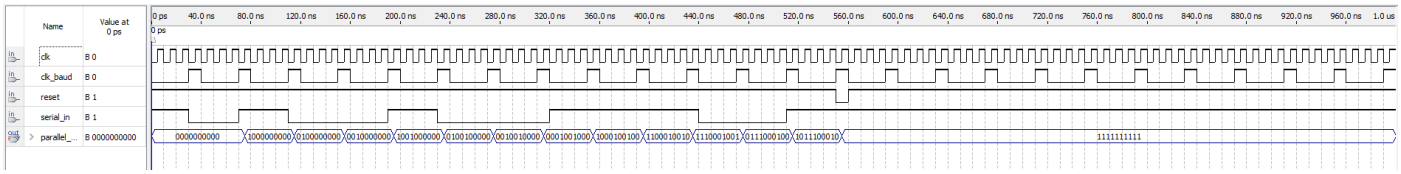

Figure 36. Receiver Shift Register Timing Simulation

As shown in both simulations, the parallel_out is always holding the value of the currently registered data.

At 0ns, though the serial_in bit is 1, it is not registered in the shift register because there is no positive baud clock.

At 70ns, a 1 is registered as the MSB so that the parallel_out becomes binary 1000000000.

From 110 – 190ns, there are two 0s input with two positive baud clocks so that the parallel_out becomes binary 0010000000.

At 550ns, a reset signal resets the shift register, as a result, the registered data become all 1s.

2.2.4 Data Register

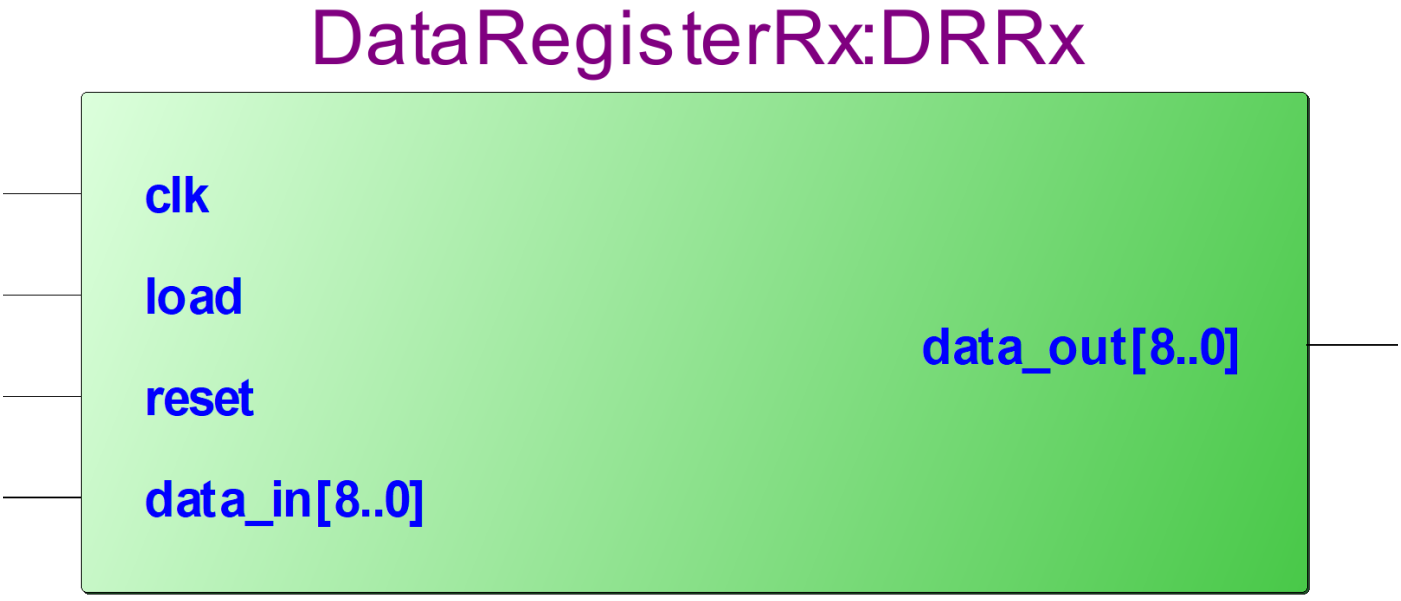

Figure 37. Receiver Data Register

Different from the transmitter data register, the receiver's take another input load which control the data register whether to load the input data or not. This is necessary for a receiver in the design because unlike the transmitter take a parallel of data at one clock cycle, the receiver required more than one clock cycle to receive a complete set of data. Hence, the data register only output a packet of data when it is completed received, which is controlled by this load signal.

ASM Chart

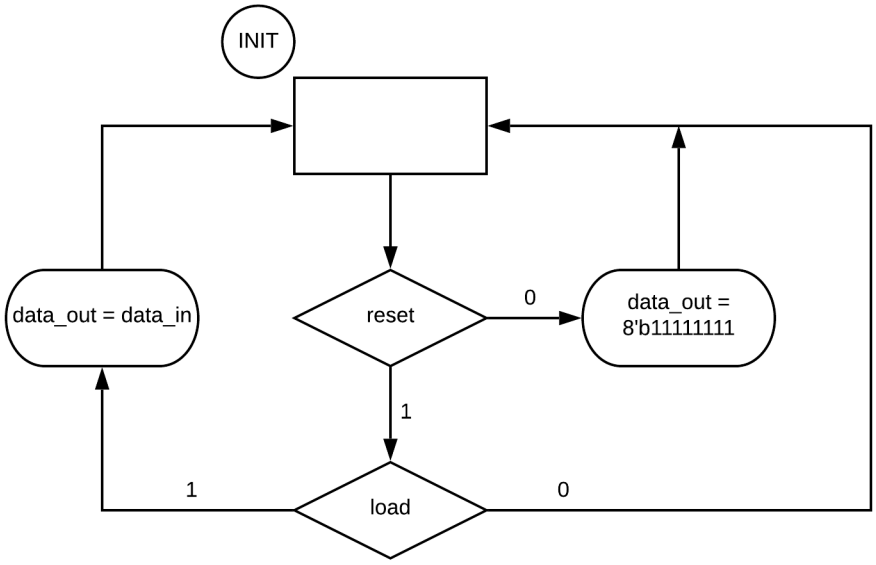

Figure 38. Receiver Data Register ASM

Verilog Code

"DataRegisterRx.v"

```
/* Receiver */
/* Data Register Module */

/* This module stores the received data which comprises 9 bits that are necessary to
be handled: 1 stop bits + 1 parity bits + 7 data bits the 7-segment decoder and the
error detector take the data stored in this module to do some further operation. */

module DataRegisterRx
(
	clk,
	reset,
	load,
	data_in,
	data_out
);

input			clk;
input			reset;
input			load;
input	[8:0]		data_in;

output	[8:0]		data_out;

reg	[8:0]		data_out;


always @(posedge clk)
begin
	if(!reset)
		data_out <= 9'b11_1111111;

	else if(load)
		/* when load signal is received from the controller, the module loads
the data from the shift register */
		data_out <= data_in;
end
endmodule
```

Simulation Results

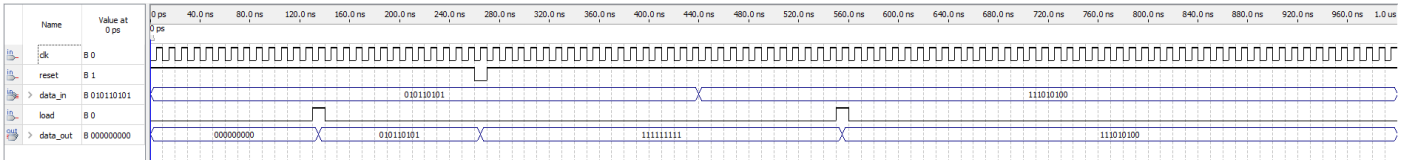

Figure 39. Receiver Data Register Functional Simulation

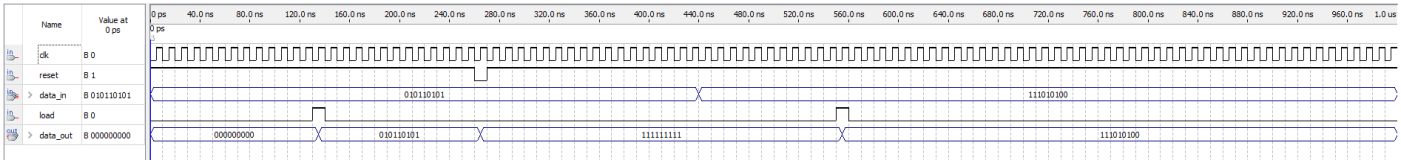

Figure 40. Receiver Data Register Timing Simulation

Two packet of data 010110101 and 111010100 are set as the input. It can be seen that the data register does not register the first set of data despite it is of the input, but only when a load signal is detected.

At 130ns, a logic 1 load signal renders the data register loads the data from current data input data_in, so that the data_output becomes the same as the data_in, meaning the input data is registered.

At 260ns, a logic 0 reset signal resets the data register and fill it with 10-bit 1s, since 1 indicates the idle bits in RS232 protocol.

At 550ns, a logic 1 load signal loads the other set of data as the data_out becomes the same as the data_in.

2.2.5 Error Detector

Figure 41. Receiver Error Detector

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.

ASM Chart

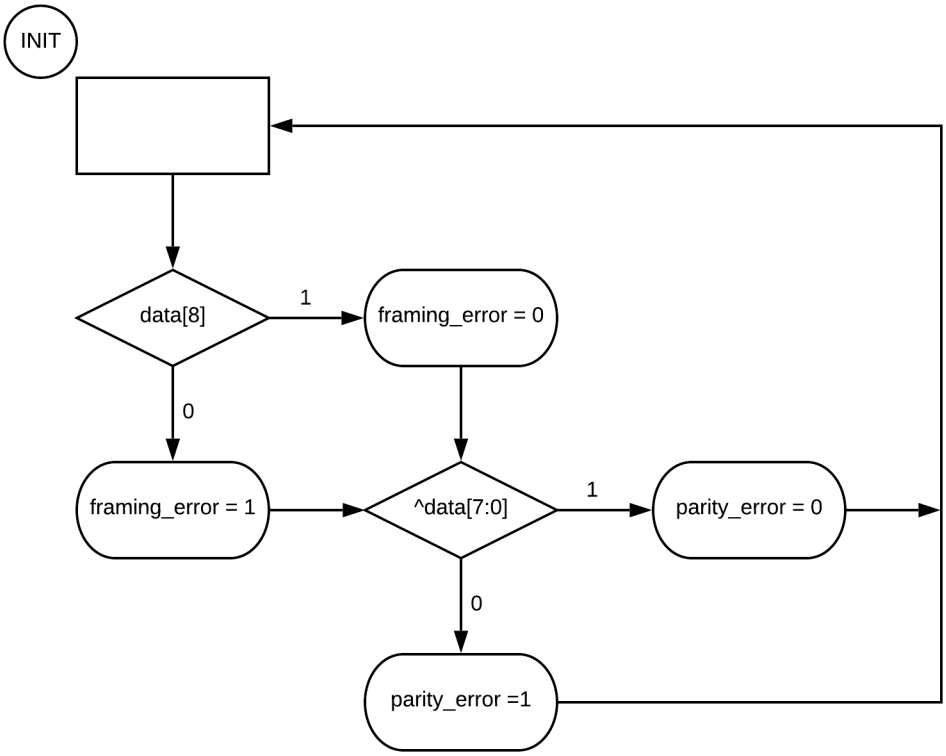

Figure 42. Receiver Error Detector ASM

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

"ErrorDetectorRx.v"

```
/* 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 43. Receiver Error Detector Functional Simulation

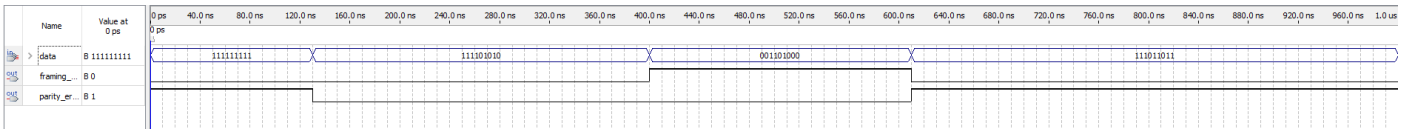

Figure 44. Receiver Error Detector Timing Simulation

2.2.6 7-Segment Decoder

Figure 45. Receiver 7-Segment Decoder Hex6

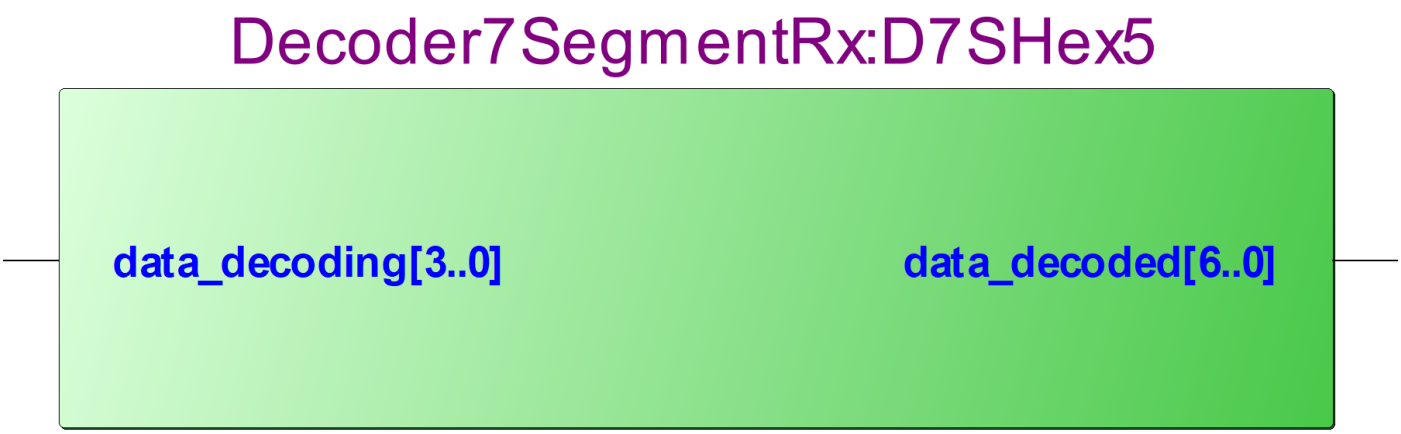

Figure 46. Receiver 7-Segment Decoder Hex5

Figure 45 and 46 are two different instances for the one same module: the 7-segment decoder module. This 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

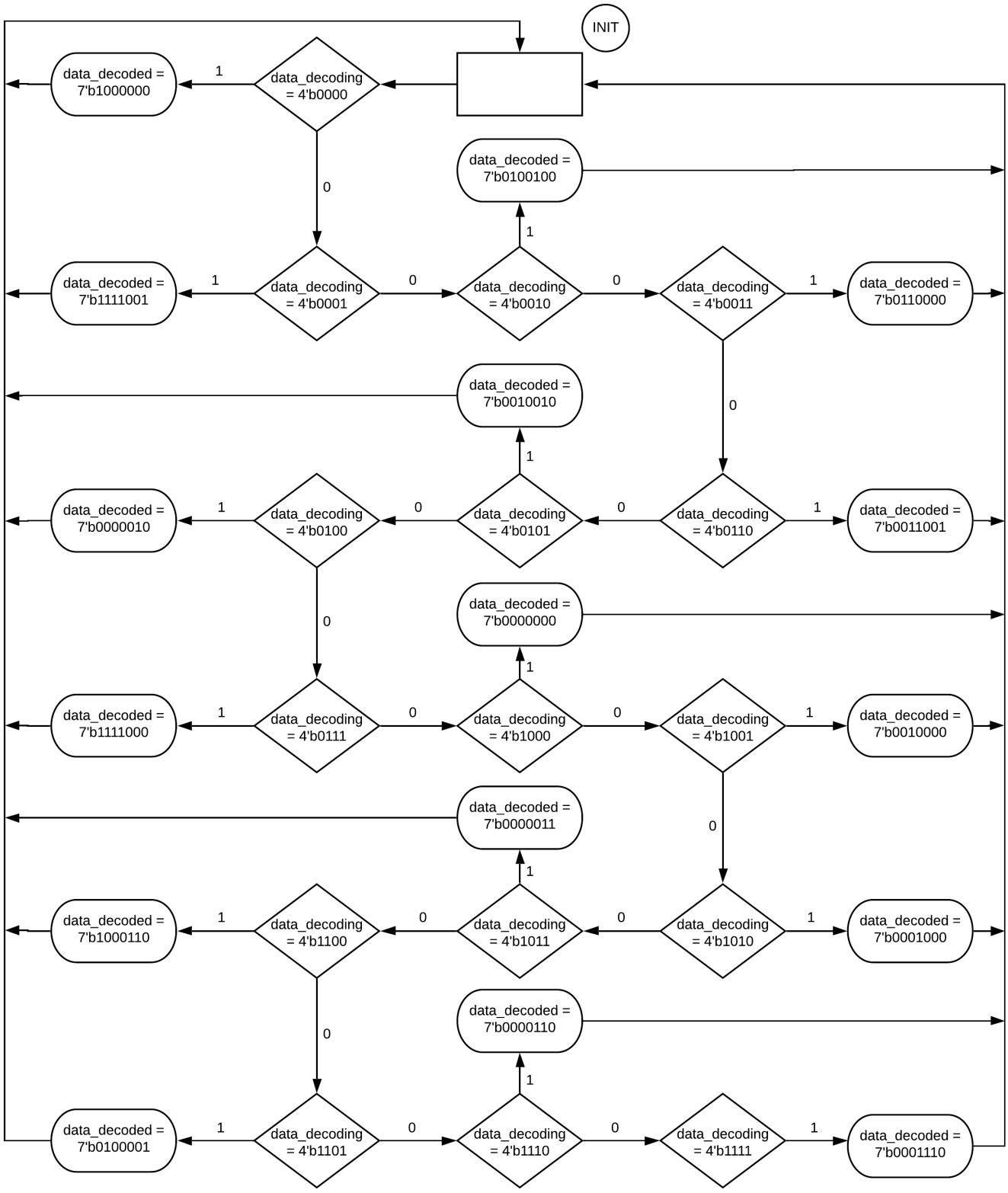

Figure 47. Receiver 7-Segment Decoder ASM

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

"Decoder7SegmentRx.v"

```
/* 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

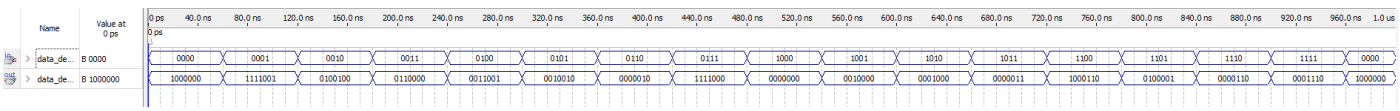

Figure 48. Receiver 7-Segment Decoder Functional Simulation

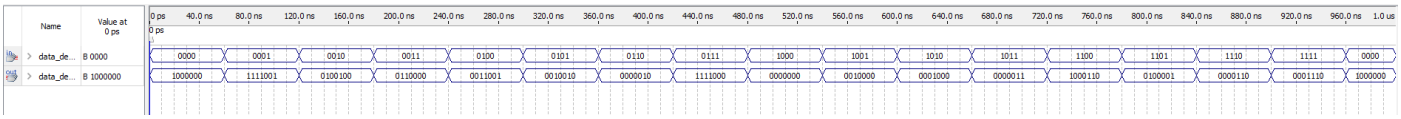

Figure 49. Receiver 7-Segment Decoder Timing Simulation

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.

2.2.7 Controller

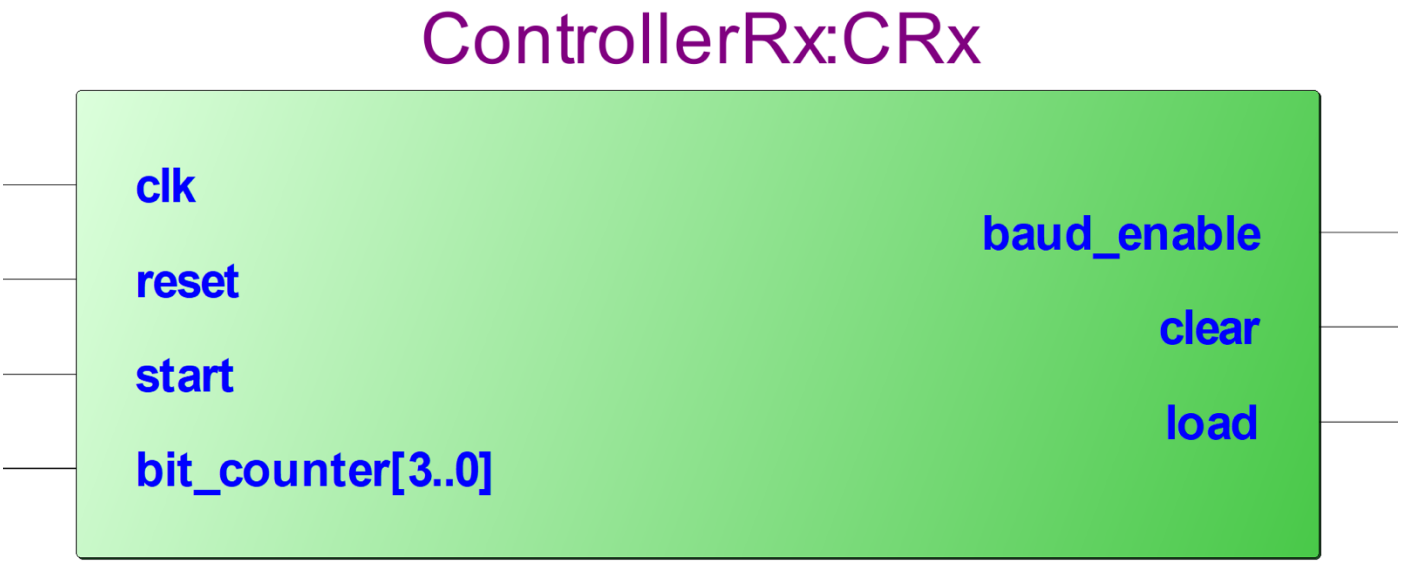

Figure 50. Receiver Controller

The ports of the controller of the receiver is identical to that of the transmitter in spite of an additional output of control signal clear which control the bit counter of the receiver.

ASM Chart

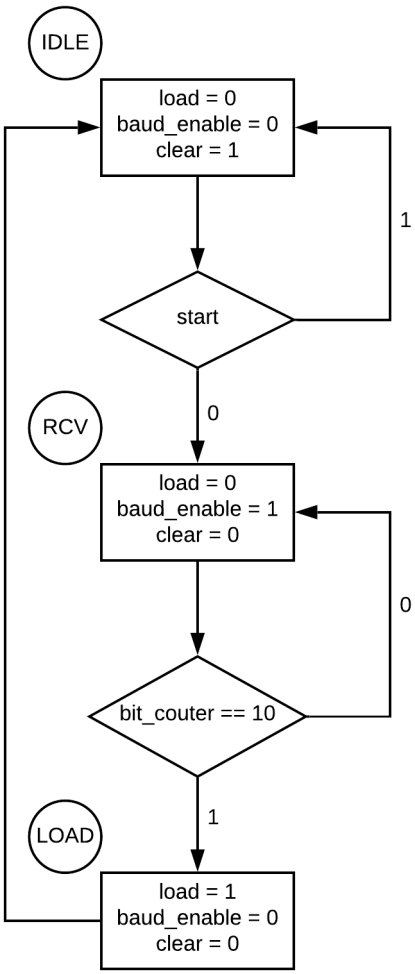

Figure 51. Receiver Controller ASM

State Transition Diagram

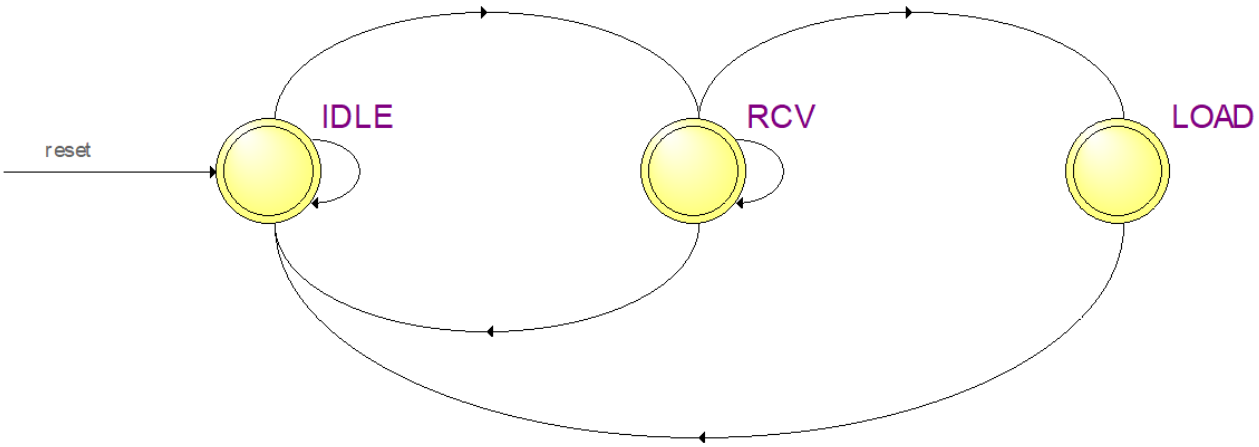

Figure 52. Receiver Controller State Transition Diagram

As illustrated in Figure 50 and 51, the receiver controller has 3 states: IDLE, RCV, and LOAD. 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 where the load and baud_enable equals 0, and the clear equals 1.

In 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 0 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 set the baud_enable to 1 and clear to 0. In this case, the baud generator will start to work, and the bit counter will count up for the receiving bits. Once the bit counter reaches its maximum number of 10 bits, the controller moves to the next state, the LOAD state.

In the LOAD state, the receiver loads the data into the data register for the 7-segment to display. The controller set the load signal to 1 and others to 0. Therefore, the data register will register one set of the received data. After one system clock cycle, the state will move to the IDLE and the receiver is ready to receive another set of data.

Verilog Code

"ControllerRx.v"

```
/* 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 ControllerRx
(
      clk,
      reset,
      start,
      bit_counter,
      load,
      clear,
      baud_enable
);

// idle state, data are ready to be received
parameter              IDLE  = 2'b00;
// receive state, data are on the progress of receiving
parameter              RCV   = 2'b01;
/* load state, data are loaded to the data register and displayed on the 7-segment
led */
parameter              LOAD  = 2'b10;

input                  clk;
input                  reset;
// a start bit of 0 trigger the process of receiving
input                  start;
input       [3:0]      bit_counter;

// load the data to the data register
output                 load;
```

```
output                  baud_enable;     // enable baud generator
output                  clear;           // clear the bit counter

reg                     clear;
reg                     load;
reg                     baud_enable;
reg         [1:0]       pstate;
reg         [1:0]       nstate;


// state machine
always @(posedge clk)
begin
      if(!reset)
            pstate <= IDLE;

      else
            pstate <= nstate;
end

always @(pstate, start, bit_counter)
begin

      // default keep the present state in loop
      nstate            = pstate;
      load              = 1'b0;
      baud_enable = 1'b0;
      clear             = 1'b0;

      case(pstate)
            IDLE:
            begin
                  // clear the bit counter for receiving
                  clear             = 1'b1;
                  if(!start)
                        // move to RCV state for receiving data
                        nstate = RCV;
            end

            RCV:
            begin
                  // enable the baud generator
                  baud_enable = 1'b1;

                  // when 10 bits of data is received load them in
                  if(bit_counter == 4'd10)
                        nstate = LOAD;
            end

            LOAD:
            begin
                  load              = 1'b1;     // load them into data register

                  nstate            = IDLE;     // back to IDLE state
            end

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

Simulation Results

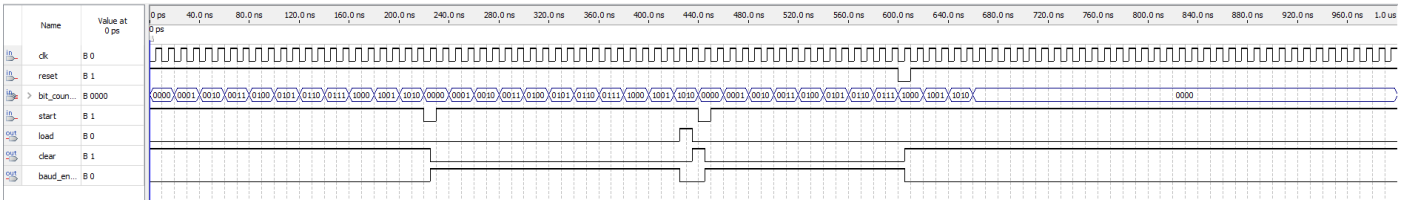

Figure 53. Receiver Controller Functional Simulation

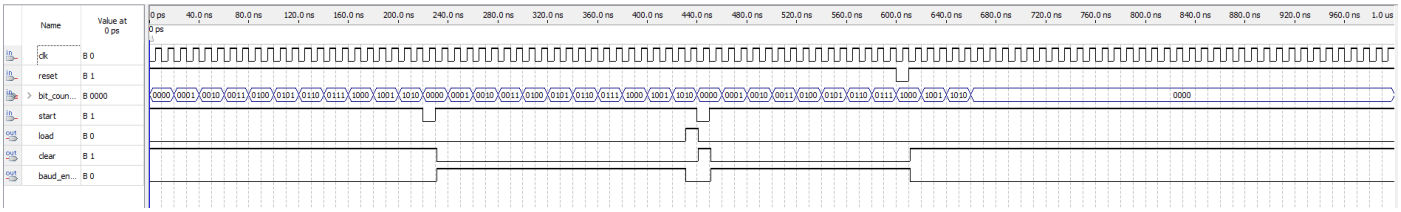

Figure 54. Receiver Controller Timing Simulation

Figurer 53 and 54 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.

At 220ns, a start signal is given as logic 0 and the controller will move from IDLE state to the RCV state, so that the receiver starts to receive the incoming data as the baud_enable is set to logic 1 and bit counter to logic 0. which means the shift register starts to shift the data and the bit counter starts to count.

At 430ns, there is a single pause of the load signal which lasts for 1 clock cycle. That means the state is moved to LOAD and a load single will load the data into the data register.

At 600ns, a reset signal resets the controller, therefore the controller immediately goes to IDLE state where the baud_enable and clear become logic 0 and logic 1 respectively again.

3 Full System Documentation

3.1 Interconnections of Modules

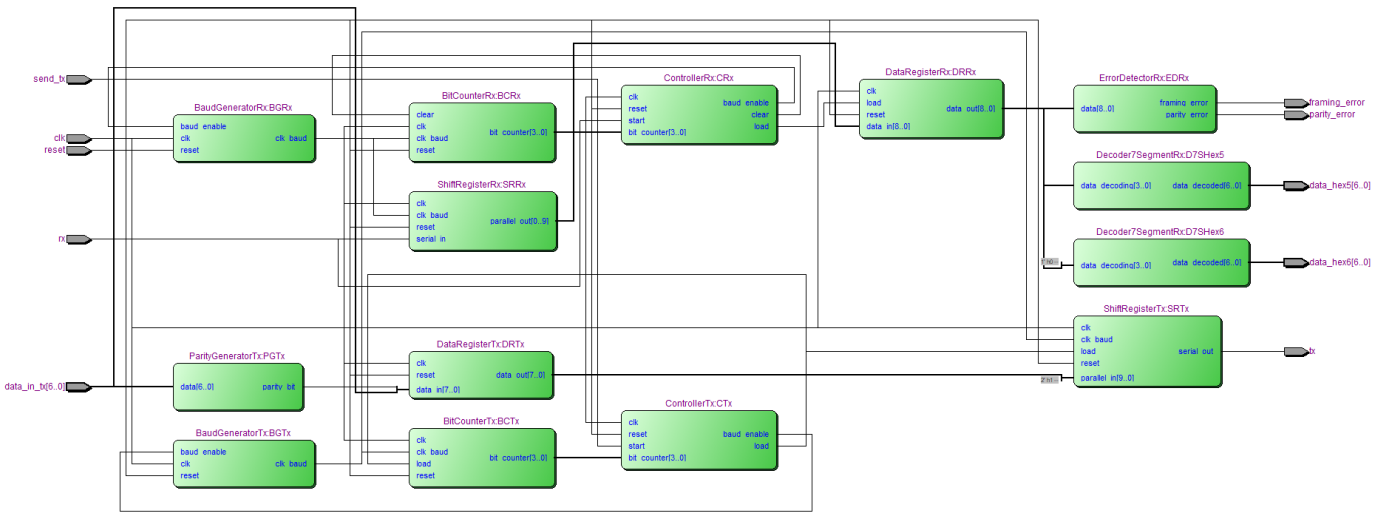

Figure 55. Module Interconnections

Figure 55 showcases the interconnection amongst all modules of the transmitter and the receiver. This full system can transmit and receive data simultaneously under the same system clock and reset command but without affecting the other one.

Datapath of the transmitter: 7-bit data flow into the system from the data_in_tx input port, and are duplicated into 2 sets: 1 set flows into the parity generator to generate a parity bit, then enters the data register module; the other set flows into the data register directly. Together, the two sets of data constitute an 8-bit data (1 parity bit + 7 data bits) and is registered into the data register. Then, the shift register will load the registered data in parallel when a load signal is detected and shifts them out in serial to the tx port.

Datapath of the receiver: The data is input bit by bit from the rx 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. Then, from the shift register, once all 10-bit data is registered, they will be loaded into the data register when a load signal is given to the register. 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 data 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.

Controllers: The controllers of the transmitter and the receiver are nearly identical in spite of a slight difference. The transmitter controller output a load signal to control its shift register to load the data from external input, whereas the receiver controller output one to control its data register to load the data from its internal shift register. Additionally, the receiver controller has an extra clear output to clear its bit counter while the transmitter controller uses the load signal to control its bit counter.

Both controllers of the transmitter and the receiver control their baud generators with baud_enable output signals and receive the status of their bit counter with a bit_counter input signals.

3.2 Verilog Code

“Top.v”

```
/* Top Module */

/* All interconnection of modules is shown in this Verilog file. */

module Top
(
      clk,                                  // input
      reset,                                // input
      send_tx,                              // input
      data_in_tx,                           // input
      data_hex6,                            // output
      data_hex5,                            // output
      parity_error,                         // output
      framing_error,                        // output
      tx,                                   // output
      rx                                    // input
);

// set the baudrate to 50MHz / 38400 = 1302
parameter               BAUDRATE = 11'd1302;
// 50MHz system clock
input                   clk;
// reset all states
input                   reset;
// a send command signal generated from a pushbutton
input                   send_tx;
// rx signal as the input of data
input                   rx;
// the input data bits generated from 7 switches
input       [6:0]       data_in_tx;
// serial output to the terminal computer
output                  tx;
// 7-segment to display the first hex number
output      [6:0]       data_hex6;
// 7-segment to display the second hex number
output      [6:0]       data_hex5;
// output signal to control the parity error LED
output                  parity_error;
// output signal to control the framing error LED
output                  framing_error;
// baud clock generated for the transmitter
wire                    clk_baud_tx;
// baud clock generated for the receiver
wire                    clk_baud_rx;
// load signal for shift register of transmitter
wire                    load_tx;
// load signal for data register of receiver
wire                    load_rx;
// enable signal for baud generator of transmitter
wire                    baud_enable_tx;
// enable signal for baud generator of receiver
wire                    baud_enable_rx;
// transmitting data: 7 data bits + 1 parity bit
wire        [7:0]       data_tx;
// receiving data: 1 start bit + 7 data bits + 1 parity bit + 1 stop bit
```

```
wire        [9:0]       data_rx;
// bit counting signal for transmitter
wire        [3:0]       bit_counter_tx;
// bit counting signal for receiver
wire        [3:0]       bit_counter_rx;
// generated parity bit for transmitter
wire                    parity_bit_tx;
// clear signal for the bit counter of receiver
wire                    clear_rx;
// parallel output data for receiver: 7 data bits + 1 parity bit + 1 stop bit
wire        [8:0]       data_out_rx;


// transmitter baud generator
BaudGeneratorTx #(.BAUDRATE(BAUDRATE)) BGTx
(
      .clk(clk),
      .reset(reset),
      .baud_enable(baud_enable_tx),
      .clk_baud(clk_baud_tx)
);

// receiver baud generator
BaudGeneratorRx #(.BAUDRATE(BAUDRATE)) BGRx
(
      .clk(clk),
      .reset(reset),
      .baud_enable(baud_enable_rx),
      .clk_baud(clk_baud_rx)
);

// transmitter data register
DataRegisterTx DRTx
(
      .clk(clk),
      .reset(reset),
      .data_in({parity_bit_tx, data_in_tx}),
      .data_out(data_tx)
);

// receiver data register
DataRegisterRx DRRx
(
      .clk(clk),
      .reset(reset),
      .load(load_rx),
      // storing the received data, but throwing the start bit
      .data_in(data_rx[9:1]),
      .data_out(data_out_rx)
);

// transmitter shift register
ShiftRegisterTx SRTx
(
      .clk(clk),
      .clk_baud(clk_baud_tx),
      .reset(reset),
      .load(load_tx),
      /* the two less significant bits are 0 - start bit, 1 - bit to represent the
IDLE state */
      .parallel_in({data_tx, 2'b01}),
      .serial_out(tx)
);
```

```
// receiver shift register
ShiftRegisterRx SRRx
(
	.clk(clk),
	.clk_baud(clk_baud_rx),
	.reset(reset),
	.serial_in(rx),
	.parallel_out(data_rx)
);

// transmitter bit counter
BitCounterTx BCTx
(
	.clk(clk),
	.clk_baud(clk_baud_tx),
	.reset(reset),
	.load(load_tx),
	.bit_counter(bit_counter_tx)
);

// receiver bit counter
BitCounterRx BCRx
(
	.clk(clk),
	.clk_baud(clk_baud_rx),
	.reset(reset),
	.clear(clear_rx),
	.bit_counter(bit_counter_rx)
);

// 7 segment decoder for hex6
Decoder7SegmentRx D7SHex6
(
	// a 0 is put in front to meet the ASCII table
	.data_decoding({1'b0, data_out_rx[6:4]}),
	.data_decoded(data_hex6)
);

// 7 segment decoder for hex5
Decoder7SegmentRx D7SHex5
(
	.data_decoding(data_out_rx[3:0]),
	.data_decoded(data_hex5)
);

// transmitter controller
ControllerTx CTx
(
	.clk(clk),
	.reset(reset),
	.start(send_tx),
	.bit_counter(bit_counter_tx),
	.load(load_tx),
	.baud_enable(baud_enable_tx)
);

// receiver controller
ControllerRx CRx
(
	.clk(clk),
	.reset(reset),
	.start(rx),
	.bit_counter(bit_counter_rx),
	.load(load_rx),
```

```
        .clear(clear_rx),
        .baud_enable(baud_enable_rx)
);

// parity bit generator for the transmitting data
ParityGeneratorTx PGTx
(
        .data(data_in_tx),
        .parity_bit(parity_bit_tx)
);

// framing and parity bit detector for the receiving data
ErrorDetectorRx EDRx
(
        .data(data_out_rx),
        .parity_error(parity_error),
        .framing_error(framing_error)
);
endmodule
```

3.3 Simulation Results

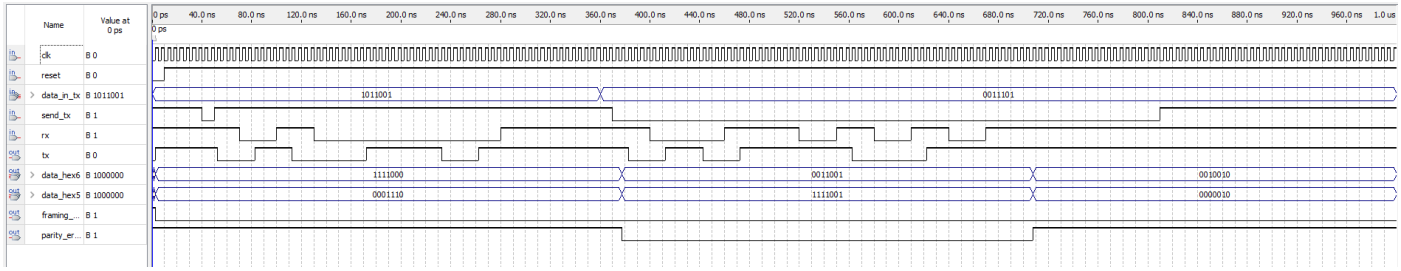

Figure 56. Full System Functional Simulation

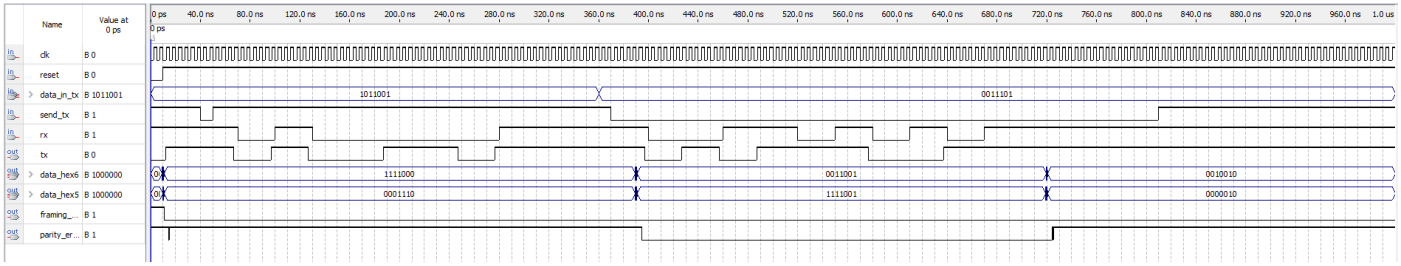

Figure 57. Full System Timing Simulation

Noticed from the difference between the functional simulation and the timing simulation of the full system, the time delay is approximately 1 – 2 system clocks, which is, in this simulation, 5 – 10ns.

At 0ns, though not necessarily, the reset signal is set to logic 0 to set the whole system into a stable state, so that all the data inside the system is set to logic 1, which is the idle bits in RS232 protocol. As shown after 10ns, the hex6 and hex5 7-segment LED becomes '7F' indicating the data bits are 7 bits of 1s. It is worth noting that the MSB of the data is always 0 in order to properly decode according to the ASCII table.

At 40ns, the send_tx signal is set to low so that the transmitter starts to load the input data and ns 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 rx signal, from 70 to 370ns, 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_hex6 and data_hex5, the LED on the 7-segments show the character '4' and '1', which fit the received data.

From 720ns, another set of data on rx input signal, can be read as, 0 + 0110101 + 0 + 1, and the data from MSB to LSB should be 1010101100. As the data bits consist of even number of 1 but the parity bit is 0, the parity error should arise, which is indicated by the logic 1 of parity_error output signal in the simulation. It is worth noting that the parity error is generated after the start as a result that the 8-bit data register of the receiver is fill with 00000000 before the reset and 11111111 after the reset, of which both the odd parity is not correctly set.

4 Discussion and Conclusion

This report demonstrated the implement of a UART 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. The computer terminal showed the correct data input from the DE2 board and displayed the corresponding character based on the ASCII code; the 7-segment LED on the DE2 board also displayed the correct hexadecimal number of the ASCII code for the character received from the computer terminal.

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.

Appendix

Zhang, Junhao

ELEC373 2019/20 Assignment 1 Demonstrators Check Sheet

Student Name: Zhang, Junhao

Student ID No: 201377244 Module Level: UG

Demonstrator Name: J.S. Smith, A. Hernandez Martinez, Z Cao, R. Gillies

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: Key 3.

Top level Design: BDF or Verilog Verilog.

Transmit Working: ✓

ASMs for Transmit: See rep

Simulations?

Receive Working: ✓

ASMs for Receive: See rep

Simulations:

Fully Synchronous? ✓

Comments on RTL View: OK

Students understanding of code:

Good

Code corresponding to ASMs:

Signature:

Date/Time: 27/11/19