ELEC473
Digital Systems Design

Assignment 2
Serial Communications

Khanthapak Thaipakdee

Department of Electrical Engineering and Electronics,
University of Liverpool,
Brownlow Hill, Liverpool L69 3GJ, UK

Table of Contents

Contents **Page**

List of Figures

Figure **Page**

1. Description of Architecture

The Universal Asynchronous Receiver-Transmitter (UART) plays a pivotal role in serial communication within digital systems, often implemented using Verilog hardware description language. Verilog facilitates the development of modules for asynchronous communication, enabling precise transmission and reception of messages via UART. Typical Verilog code encompasses components such as the shift register, baud rate generator, and control logic, ensuring accurate timing and frame design for data transmission. This seamless integration simplifies communication between various components within digital systems, making UART with Verilog a versatile solution for diverse applications, from embedded systems to communication interfaces in electronic products.

In this project, the DE2 Board and RS232 interface serve as a testing platform for a comprehensive system that incorporates both UART and Infrared (IR) communication. Data transmission from the DE2 board to a PC occurs through a serial link, with terminal software like “Putty” displaying ASCII values corresponding to the transmitted data. The architecture for data transmission, depicted in Figure 1, comprises essential components such as the pulsed detector, baud rate generator, parity generator, bit counter, shift register, controller, and seven-segment decoder. These components ensure efficient and accurate transmission of data from the DE2 board to external devices.

Conversely, the architecture for data reception, illustrated in Figure 2, handles the reception of incoming data from external sources, decoding it for display on the DE2 board. Through pin assignments and hardware connections, the architecture interfaces seamlessly with the physical environment, facilitating robust communication between digital systems and external devices.

Additionally, the project extends its functionality by integrating IR communication capabilities using the HSDL-3201 module. This allows for wireless data transmission alongside traditional wired UART communication, enhancing the system's versatility and applicability in various scenarios requiring both wired and wireless communication capabilities. Through rigorous testing and validation, the project ensures reliable and efficient communication between digital systems and external devices, catering to a wide range of application requirements.

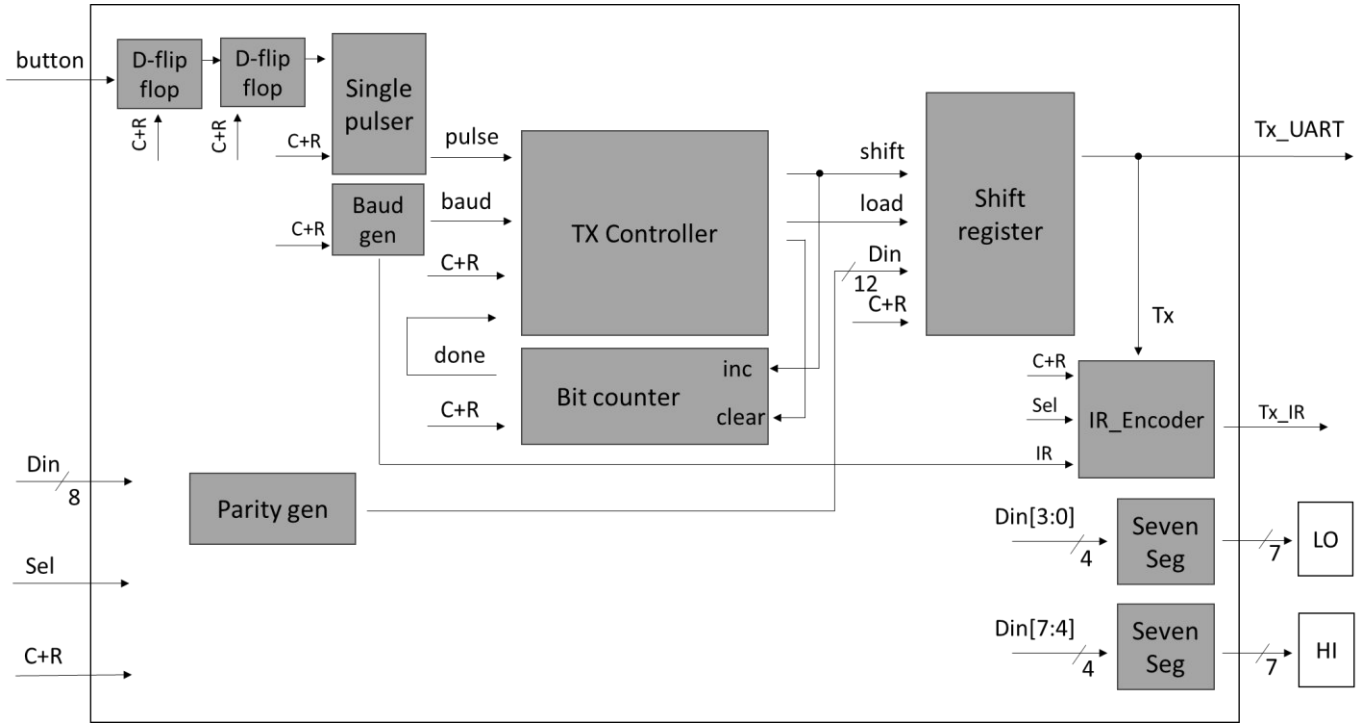


Figure 1 Block diagram of UART Transmitter

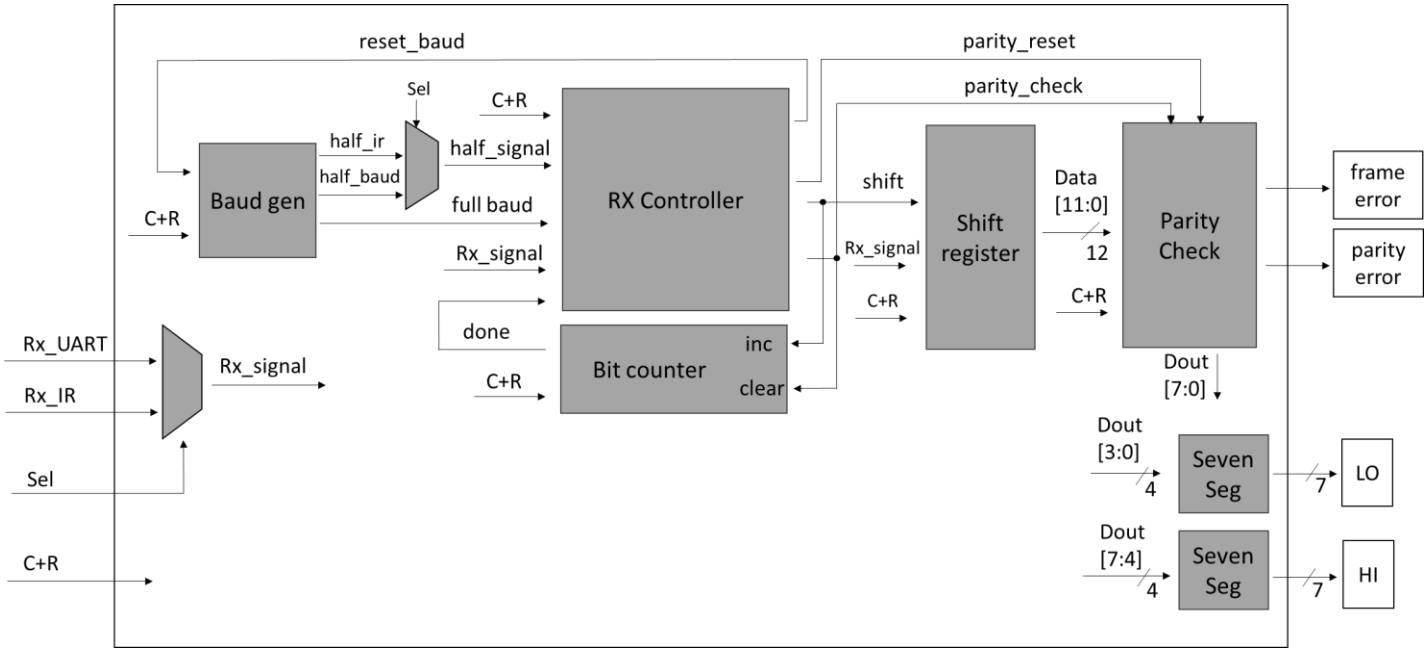


Figure 2 Block diagram of UART Receiver

2. Description of each module

2.1 Single Pulser

The single pulser module is designed to detect button presses in an active-low configuration. It operates by monitoring the input signal Key_n, which transitions to a low state when the button is pressed. Upon detecting this transition, indicating a button press, the module generates a pulse on the output Key_p and enters the “WAIT_L” state. It remains in this state until the button is released, and Key_n returns to its idle, high state. At this point, the module transitions back to the “WAIT_H” state, ready to detect the next button press.

In cases where the button is pressed simultaneously with the reset button, the module remains in the “WAIT_H” state, ensuring proper initialization. Additionally, the module is capable of detecting every falling edge of Key_n without the need for a reset, as demonstrated in the red box in Figure 4.

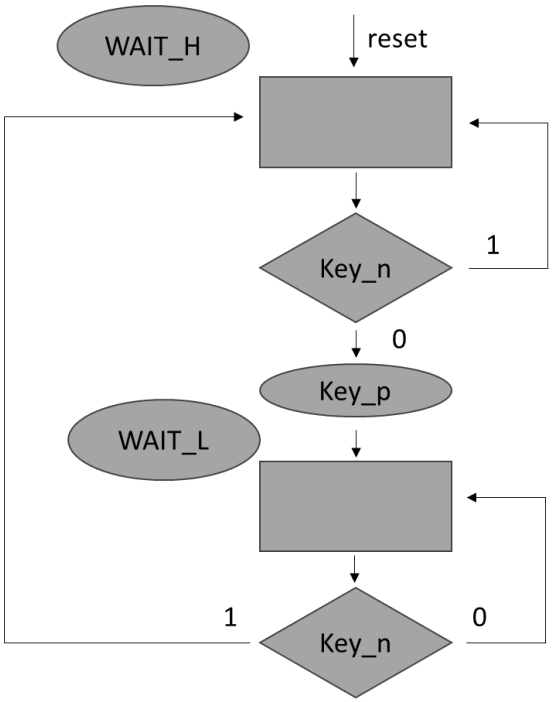


Figure 3 Single Pulser ASM

```
module switch (clock, reset, key_n, key_p);
    input clock;
    input reset;
    input key_n;
    tri0 reset;
    tri0 key_n;
    output key_p;

    reg key_p;
    reg [1:0] n_state, p_state;
    parameter WAIT_H=0, WAIT_L=1;

    always @(posedge clock)
        begin
            p_state <= n_state;
        end

   always @(p_state or reset or key_n)
        begin
            if (~reset)
                begin
                    n_state = WAIT_H;
                    key_p = 1'b0;
                end
            else
                begin
                    //Inserting to prevent latch inference
                    key_p = 1'b0;
                    n_state = p_state;
                    case (p_state)
                        WAIT_H:
                            begin
                                if (~(key_n))
                                    begin
                                        n_state = WAIT_L;
                                        key_p = 1'b1;
                                    end
                                else if (key_n)
                                    n_state = WAIT_H;
                            end
                        WAIT_L:
                            begin
                                if (key_n)
                                    n_state = WAIT_H;
                                else if (~(key_n))
                                    n_state = WAIT_L;
                            end
                         default:
                            key_p = 1'bx;
                    endcase
                end
        end
endmodule
```

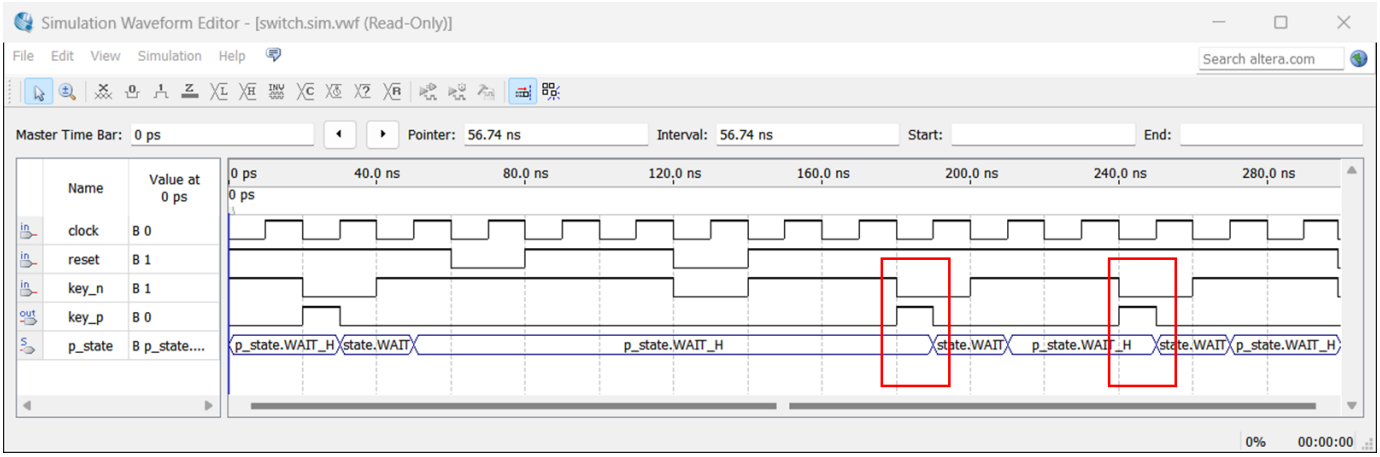


Figure 4 Single Pulser Simulation

2.2 Parity Generator

The Verilog module oddparity designs to generate a data stream with odd parity. It takes an 8-bit input data_in and produces a 12-bit output data_out with a start bit, stop bits, and an additional bit for odd parity as shown in Figure 5.

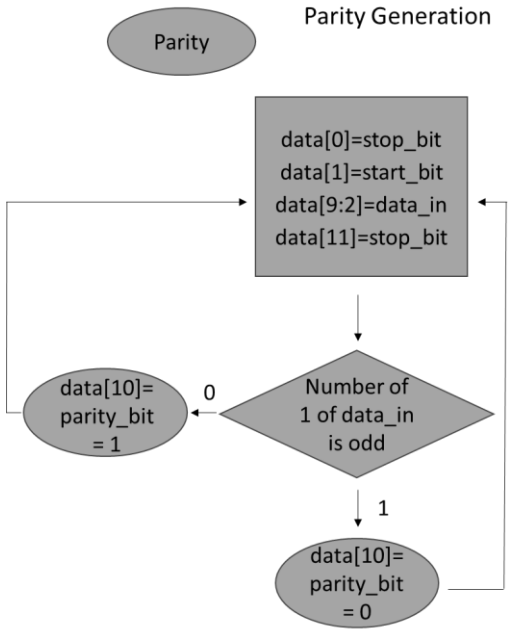


Figure 5 Parity generator ASM

In Figure 6, two scenarios are depicted with data bits containing 1 and 2 ones, respectively. The simulated parity bit in the output aligns with the logic of odd parity checking. Specifically, when the data bits have an odd number of ones, the parity bit is set to 0; conversely, when the data bits have an even number of ones, the parity bit is set to 1.

```
module oddparity(
  input wire [7:0] data_in,
  output reg [11:0] data_out
);

assign data_out = {1'b1,~(^data_in),
datain,1'b0,1'b1};

endmodule
```

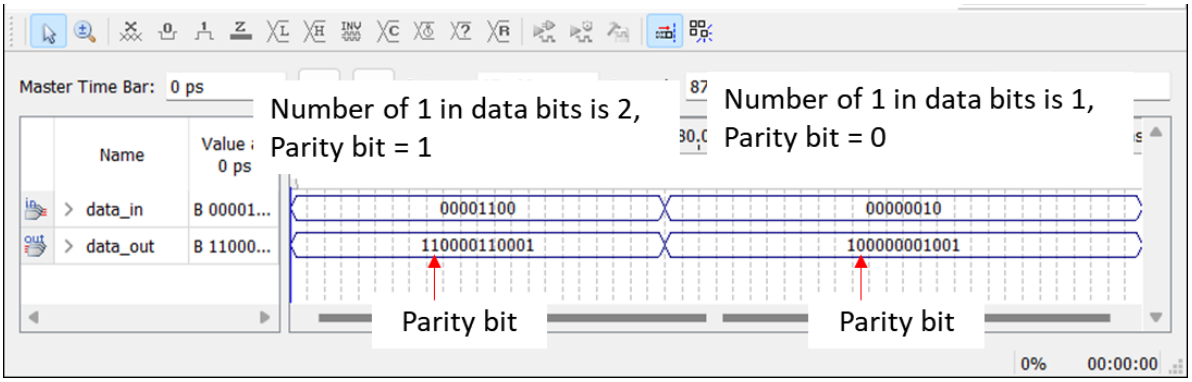


Figure 6 Parity Generator Simulation

2.3 Seven-Segment Display

The display module is 4-to-7 decoder which transfer from 4 bits data to 7 bits seven-segment. The comparison of transferring can see from Table 1. The seven-segment display simulation illustrates the corresponding output for a 4-bit data input in HEX form.

Table 1 Seven-segment decoder truth table.

Hex	Binary Output [6:0]						
0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	1
2	0	1	0	0	1	0	0
3	0	1	1	0	0	0	0
4	0	0	1	1	0	0	1
5	0	0	1	0	0	1	0
6	0	0	0	0	0	1	0
7	1	1	1	1	0	0	0
8	0	0	0	0	0	0	0
9	0	0	1	1	0	0	0
A	0	0	0	1	0	0	0
B	0	0	0	0	0	1	1
C	1	0	0	0	1	1	0
D	0	1	0	0	0	0	1
E	0	0	0	0	1	1	0
F	0	0	0	1	1	1	0
Default	x	x	x	x	x	x	x

The outcomes of the simulation ,depicted in Figure 5 correlate with the mentioned value in the truth table, guaranteeing reliability in all possible situations.

```
module sevseg(
    input wire [3:0] dataIn,
    output reg [6:0] seven_seg0
);

always @(*)
    begin
        case (dataIn[3:0])  // Display 4 bits
        4'b0000: seven_seg0 = 7'b1000000; // Hex '0'
        4'b0001: seven_seg0 = 7'b1111001; // Hex '1'
        4'b0010: seven_seg0 = 7'b0100100; // Hex '2'
        4'b0011: seven_seg0 = 7'b0110000; // Hex '3'
        4'b0100: seven_seg0 = 7'b0011001; // Hex '4'
        4'b0101: seven_seg0 = 7'b0010010; // Hex '5'
        4'b0110: seven_seg0 = 7'b0000010; // Hex '6'
        4'b0111: seven_seg0 = 7'b1111000; // Hex '7'
        4'b1000: seven_seg0 = 7'b0000000; // Hex '8'
        4'b1001: seven_seg0 = 7'b0011000; // Hex '9'
        4'b1010: seven_seg0 = 7'b0001000; // Hex 'A'
        4'b1011: seven_seg0 = 7'b0000011; // Hex 'B'
        4'b1100: seven_seg0 = 7'b1000110; // Hex 'C'
        4'b1101: seven_seg0 = 7'b0100001; // Hex 'D'
        4'b1110: seven_seg0 = 7'b0000110; // Hex 'E'
        4'b1111: seven_seg0 = 7'b0001110; // Hex 'F'
        default: seven_seg0 = 7'bxxxxxxx; // Don't
care
        endcase
    end
endmodule
```

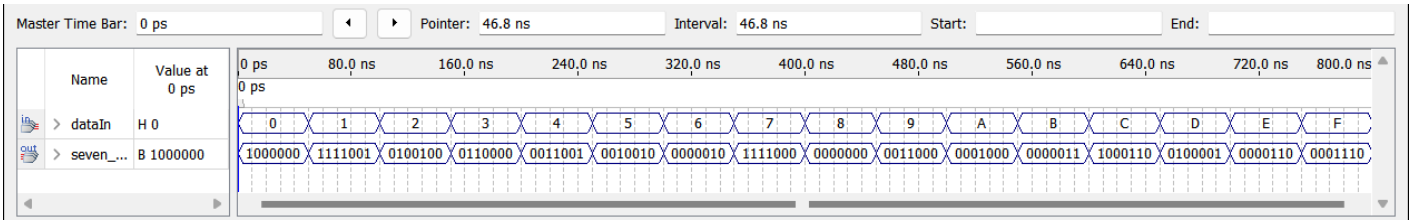


Figure 5 : Seven-segment Display Simulation

2.4 Bit Counter

The module is a 4-bit up-counter featuring asynchronous reset capability. It increments upon receiving an "inc" signal, progressing up to a count of 12. Reset functionality is provided via the "reset" input, while clearing can be achieved using "clear" (indicated by the red arrows in Figure 7). Upon reaching a count of 12, the "done" signal is raised high, indicating completion of an operation, as depicted in Figure 7. Figure 8 further confirms the correct operation of the counter, illustrating the incrementing count in response to "inc" signals, with "done" being asserted after reaching the 12th bit.

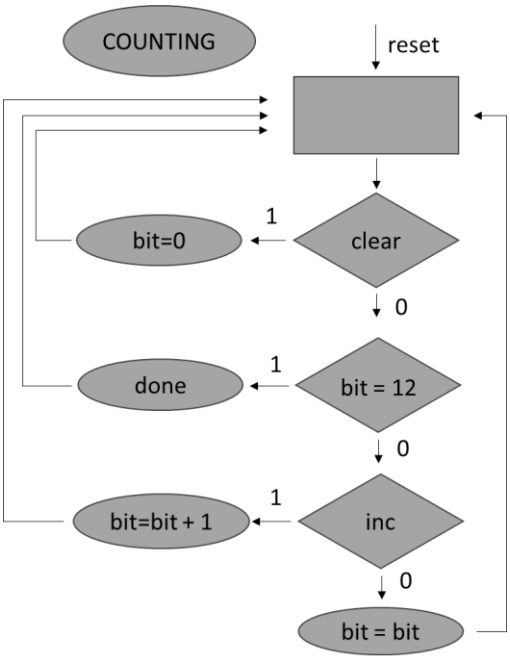


Figure 7 Bit counter ASM

```
module counter (
    input wire clock,
    input wire reset,
    input wire inc,
    input wire clear,
    output reg done
);

    reg [3:0] p_count, n_count;

    always @(posedge clock)
        begin
            if (~reset)
                p_count <= 4'b0;
            else
                p_count <= n_count;
        end

    always @(p_count, inc, clear)
        begin
            n_count = p_count;
            done = 1'b0;
            if (clear)
                n_count = 4'b0;
            if (p_count == 4'd12)
                done = 1'b1;
            else if (inc)
                n_count = p_count + 4'd1;
        end

endmodule
```

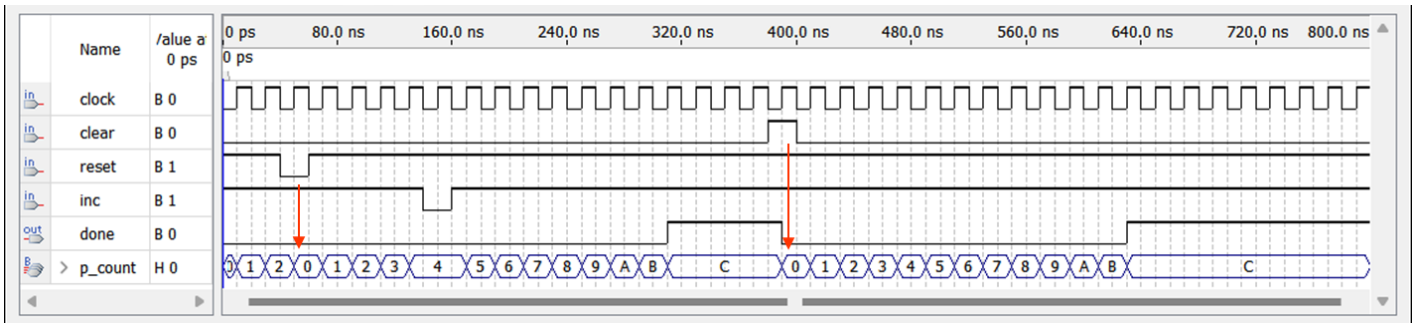


Figure 8 Bit Counter Simulation

2.5 Shift Register

The module is designed as a 12-bit shift register featuring parallel-in, serial-out functionality for transmission and serial-in, parallel-out functionality for reception. When the preset signal is activated (low, active-low), the register resets to its initial value (12'hFFF). Upon assertion of the load signal, the register receives and stores the 12-bit data input. Activating the shift signal causes the register to right-shift its contents, with the output 'Tx' reflecting the value of the least significant bit. The parallel output 'Dout' represents the received data in parallel form as shown in Figure 9.

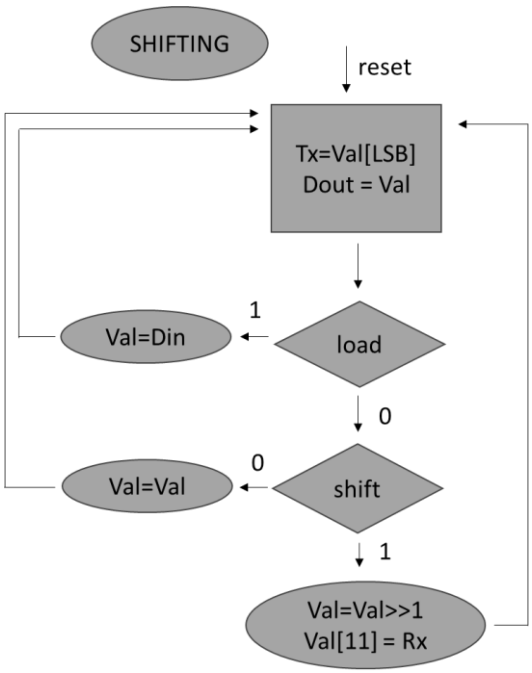


Figure 9 Shift Register ASM

```
module shift_register (
    input wire clock,
    input wire preset,
    input wire shift,
    input wire load,
    input wire [11:0] d_in,
    input wire rx,
    output tx,
    output [11:0] d_out
);

    reg [11:0] p_reg_data, n_reg_data;

    assign tx = p_reg_data[0];
    assign d_out = p_reg_data;

    initial
        p_reg_data <= 12'hFFF;

    always @(posedge clock)
        begin
            if (~preset)
                p_reg_data <= 12'hFFF;
            else
                p_reg_data <= n_reg_data;
        end

    always @(p_reg_data, load, shift, d_in, rx)
        begin
            n_reg_data = p_reg_data;
            if (load)
                n_reg_data = d_in;
            else if (shift)
                n_reg_data = { rx, p_reg_data[11:1]};
        end

endmodule
```

Figure 10 illustrates the simulation of the shift register module. When the preset signal is activated (indicated by the red arrow in Figure 10), the register's value changes to 12'hFFF in accordance with the active-low configuration. Subsequently, when the "load" signal is asserted (as indicated by the yellow arrows in Figure 10), the register stores the value of "d_in" (12'h999). Upon activation of the "shift" signal, the register performs a 1-bit right shift operation. The output signal "tx" represents the least significant bit of the register, while the most significant bit stores the value from "rx".

As observed in Figure 10, the "tx" output produces the correct serial output (12'b100110011001) corresponding to the input data "d_in". Moreover, the "d_out" signal correctly stores the value of the serial input "rx". During the time interval from 80 ns to 300 ns, the "rx" input is 0; at 320 ns, the "d_out" holds the value 12'h000.

Conversely, during the interval from 380 ns to 600 ns, the "rx" input is 1; at 620 ns, the "d_out" holds the value 12'hFFF.

Finally, the "d_out" register exhibits the correct values as compared to those outlined in Table 2.

Table 2 Shift register value table.

BINARY												HEX		
1	0	0	1	1	0	0	1	1	0	0	1	9	9	9
0	1	0	0	1	1	0	0	1	1	0	0	4	C	C
0	0	1	0	0	1	1	0	0	1	1	0	2	6	6
0	0	0	1	0	0	1	1	0	0	1	1	1	3	3
0	0	0	0	1	0	0	1	1	0	0	1	0	9	9
0	0	0	0	0	1	0	0	1	1	0	0	0	4	C
0	0	0	0	0	0	1	0	0	1	1	0	0	2	6
0	0	0	0	0	0	0	1	0	0	1	1	0	1	3
0	0	0	0	0	0	0	0	1	0	0	1	0	0	9
0	0	0	0	0	0	0	0	0	1	0	0	0	0	4
0	0	0	0	0	0	0	0	0	0	1	0	0	0	2
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	8	0	0
1	1	0	0	0	0	0	0	0	0	0	0	C	0	0
1	1	1	0	0	0	0	0	0	0	0	0	E	0	0
1	1	1	1	0	0	0	0	0	0	0	0	F	0	0
1	1	1	1	1	0	0	0	0	0	0	0	F	8	0
1	1	1	1	1	1	0	0	0	0	0	0	F	C	0
1	1	1	1	1	1	1	0	0	0	0	0	F	E	0
1	1	1	1	1	1	1	1	0	0	0	0	F	F	0
1	1	1	1	1	1	1	1	1	0	0	0	F	F	8
1	1	1	1	1	1	1	1	1	1	0	0	F	F	C
1	1	1	1	1	1	1	1	1	1	1	0	F	F	E
1	1	1	1	1	1	1	1	1	1	1	1	F	F	F

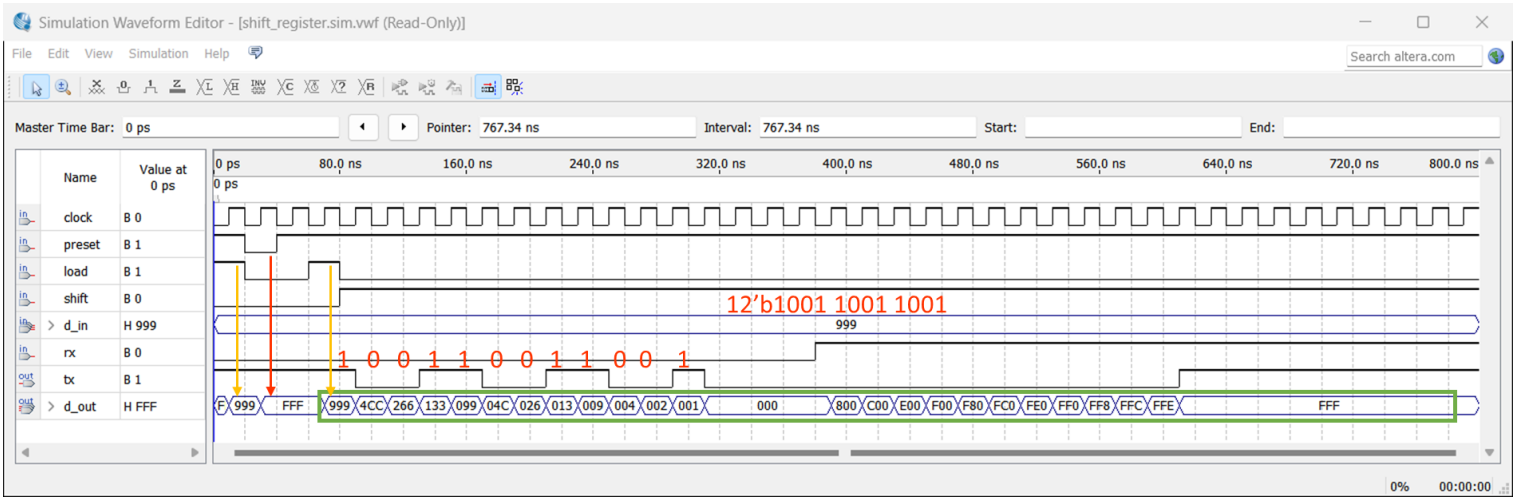


Figure 10 Shift Register Simulation

2.6 Baud rate Generator

The “baud_ir” module serves to generate timing signals essential for baud rate and infrared (IR) communication. Employing clock signals for synchronization and featuring an asynchronous reset for the baud rate counter, the module ensures reliable operation. By calculating the clock divisor based on predefined baud rate and clock frequency parameters, it achieves precise timing generation. Output signals such as "tick" and “halftick” denote full and half baud rate clock cycles, respectively, while “ir” and “halfir” signify the IR signal and half of its period, as illustrated in Figure 11. With efficient updates to the baud rate counter on each clock cycle, the module guarantees accurate timing crucial for communication protocols.

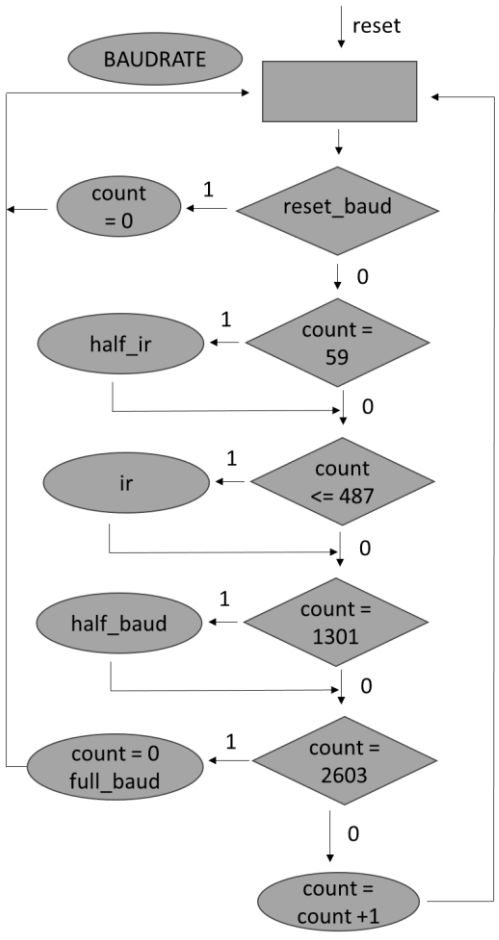


Figure 11 Baud rate Generator ASM

```
module baud_ir (
    input wire clock,
    input wire reset,
    input wire reset_baud,
    output halftick,
    output halfir,
    output ir,
    output tick
);

    parameter BAUD_RATE = 19200;
    parameter CLOCK_FREQUENCY =
    50000000; // 1 clock cycle = 20ns

    // Calculate the clock divisor
    // 50000000/19200 = 2604
    localparam CLOCK_DIVISOR =
    CLOCK_FREQUENCY / BAUD_RATE;

    // Internal counter for baud rate generation
    reg [11:0] p_baud_counter,n_baud_counter;
    //log2(2604) = 12

    assign tick = (p_baud_counter ==
    (CLOCK_DIVISOR - 1));
    assign halftick = (p_baud_counter ==
    (CLOCK_DIVISOR/2 - 1));
    assign ir = (p_baud_counter <=
    (3*(CLOCK_DIVISOR/16) - 1));
    // Rx_IR pulse = 2.45 µs
    // half of Rx_IR pulse = 1.2 µs = 60 clock cycle
    assign halfir = (p_baud_counter == 12'd59);

    always @(posedge clock)
        begin
            if (~reset)
                p_baud_counter <= 0;
            else
                p_baud_counter <= n_baud_counter;
        end

    always @(reset_baud, p_baud_counter)
        begin
            n_baud_counter = p_baud_counter;
            if(reset_baud)
                n_baud_counter = 0;
            else if (p_baud_counter ==
                    (CLOCK_DIVISOR - 1))
                n_baud_counter = 12'd0;
            else
                n_baud_counter = p_baud_counter +
                12'd1;
        end
endmodule
```

Figure 12A and 12B depict the simulation results of the baud rate generator, with the baud rate set to 125,000, contrasting the assigned baud rate of 19,200. The simulation employs a higher baud rate to overcome visibility limitations imposed by the end time constraint of 100 microseconds. The generator ensures precise timing, with Figure 12A showcasing tick intervals of 8 microseconds and "halftick" signals rising midway through each interval. The 'ir' signal remains high for three-sixteenths of the baud interval, while "halfir" toggles high after 1.2 microseconds of each tick, aligning with the midpoint of the Rx_IR pulse.

Zooming into the red box of Figure 12A in Figure 12B reveals the baud_count resetting to zero following the activation of reset_baud or when reset is asserted (low, active-low).

Figure 12C illustrates the simulation of the baud rate generator at a different baud rate (200,000), where "tick" and "halftick" outputs remain accurate. The "ir" signal maintains its three-sixteenth duration within the baud interval. Moreover, "halfir" generates a high signal after 1.2 microseconds of each tick, consistent with the datasheet specification indicating a 2.45-microsecond width for the Rx_IR pulse, independent of data rate.

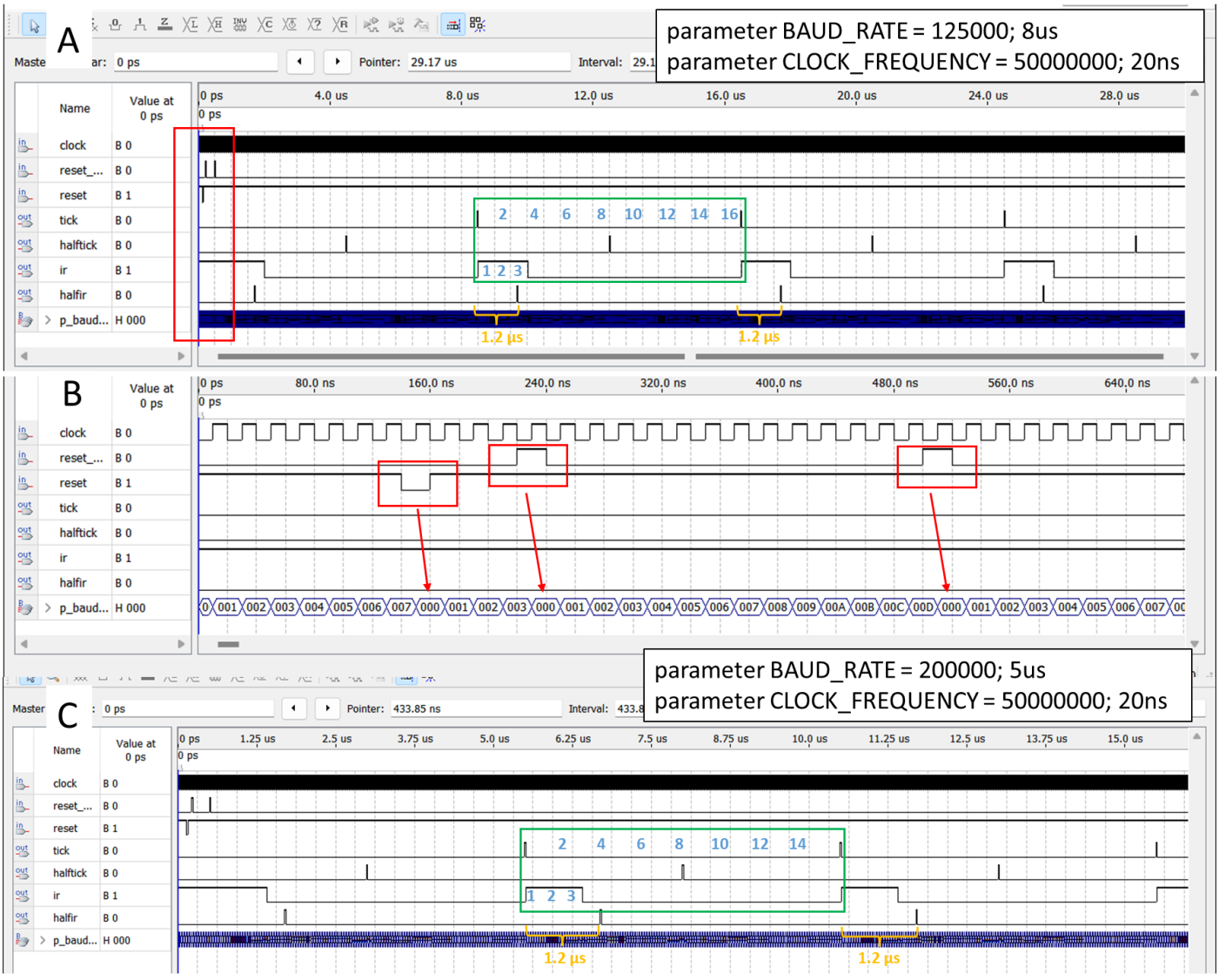


Figure 12 Baud rate Generator Simulation

2.7 Tx Controller

The module "txcontroller" serves as a crucial component for data transmission control, orchestrating the shifting, loading, and clearing of data signals. Operating within two distinct states, namely TX_IDLE and TX_DATA, the controller effectively manages state transitions based on input conditions and signals. Upon detecting a “pulse” signal, indicative of data availability, the controller initiates data loading and transitions to the TX_DATA state. Subsequently, in the TX_DATA state, the controller monitors the “baud” signal to determine the timing for data shifting and ensures continuous transmission. Additionally, it responds to the completion of data transmission signaled by “done”, triggering the clearing of data and transitioning back to the TX_IDLE state, thus maintaining efficient control over the data transmission process as shown in Figure 13.

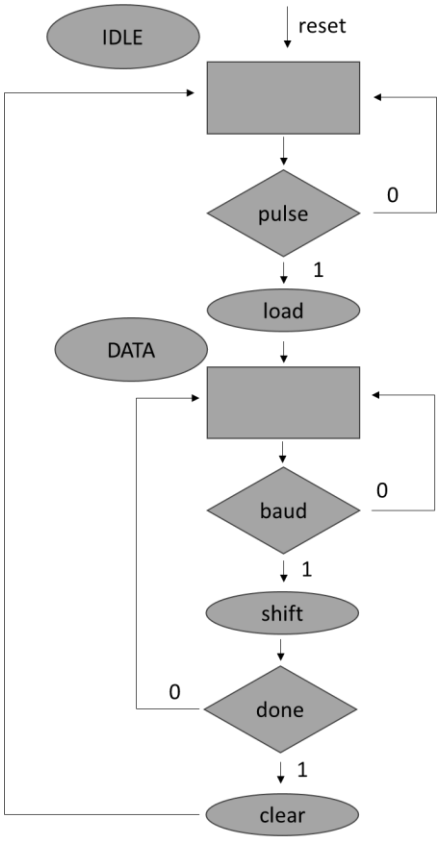


Figure 13 Tx Controller ASM

```
module txcontroller(
    input wire clock,
    input wire reset,
    input wire baud,
    input wire done,
    input wire pulse,
    output reg shift,
    output reg load,
    output reg clear
);

    // Define states
    parameter TX_IDLE = 1'b0;
    parameter TX_DATA = 1'b1;

    reg   p_state, n_state;

    always @(posedge clock)
        begin
            if (~reset)
                p_state <= TX_IDLE;
            else
                p_state <= n_state;
        end

    always @(p_state, baud, done, pulse)
        begin
            n_state = p_state;
            shift = 1'b0;
            load = 1'b0;
            clear = 1'b0;
            case (p_state)
                TX_IDLE:
                    begin
                        if (pulse)
                            begin
                              load = 1'b1;
                              n_state = TX_DATA;
                            end
                    end
                TX_DATA:
                    begin
                        if (done == 1'b1)
                            begin
                                clear = 1'b1;
                                n_state = TX_IDLE;
                            end
                        else if (baud == 1'b1)
                            shift = 1'b1;
                    end
            endcase
        end
endmodule
```

Figure 14 illustrates the simulation of the Tx controller module. Upon the “pulse” signal becoming active, the “load” signal is activated, facilitating data loading and prompting a transition to the TX_DATA state (as depicted by the green arrows in Figure 14). In the event of activation of the “reset” signal (low, active-low) during this process, the state reverts back to TX_IDLE (as indicated by the red box in Figure 14), ensuring effective management of reset conditions. Additionally, while in the TX_DATA state, activation of the “baud” signal triggers the generation of the “shift” signal (as depicted by the blue arrows in Figure 14). When the controller monitors the “done” signal, it produces the clear output and changes state back to TX_IDLE (as indicated by the yellow arrows in Figure 14).

In the TX_IDLE state, the module does not produce any output upon detecting the “baud” and “done” signals, as it is not in the correct state for data transmission (as shown in the green box in Figure 14). This ensures that output signals are generated only when the module is in the appropriate state, maintaining proper functionality and synchronization within the system.

Figure 14 Tx Controller Simulation

2.8 Rx Controller

The module “rxcontroller” functions as a controller for receiving data, overseeing the shifting, baud rate and parity resetting, and clearing based on input conditions and state transitions. Operating within three distinct states - RX_IDLE, RX_WAIT, and RX_DATA - the controller effectively manages transitions between these states to facilitate the reception process. Upon detection of a 'start bit (rx == 0), the controller initiates the resetting of the baud rate and parity, transitioning to the “RX_WAIT” state. Within this state, upon recognizing the presence of the “halfsignal” (which comprises half_baud or half_ir), strategically positioned to capture the midpoint of the data reception pulse, the controller seamlessly progresses to the “RX_DATA” state. Here, it monitors the “fullbaud” signal to control data shifting. Finally, when it detects completion of data reception (the “done” signal), it produces a “clear” output signal and returns to the “RX_IDLE” state as shown in Figure 15.

```
module rxcontroller(
    input wire clock,
    input wire reset,
    input wire rx,
    input wire done,
    input wire fullbaud,
    input wire halfsignal,
    output reg shift,
    output reg reset_baud,
    output reg parity_reset,
    output reg clear
);

    // Define states
    parameter RX_IDLE = 2'b00, RX_WAIT =
    2'b10, RX_DATA = 2'b11;

    reg[1:0] p_state, n_state;

    always @(posedge clock)
        begin
            if (~reset)
                p_state <= RX_IDLE;
            else
                p_state <= n_state;
        end

    always @(p_state, fullbaud, halfsignal, rx,
                done)
        begin
            n_state = p_state;
            shift = 1'b0;
            reset_baud = 1'b0;
            clear = 1'b0;
            parity_reset = 1'b0;
            case (p_state)
                RX_IDLE:
                    begin
                        if (rx == 1'b0) //detect start bit
                            begin
                                reset_baud = 1'b1;
                                parity_reset = 1'b1;
                                n_state = RX_WAIT;
                            end
                    end
                RX_WAIT:
                    begin // half baud or half ir
                        if (halfsignal)
                            begin
                                reset_baud = 1'b1;
                                n_state = RX_DATA;
                            end
                    end
                RX_DATA:
                    begin
                        if (fullbaud == 1'b1)
                            begin
                                shift = 1'b1;
                                n_state = RX_DATA;
                            end
                        if (done == 1'b1)
                            begin
                                clear = 1'b1;
                                n_state = RX_IDLE;
                            end
                    end
            endcase
        end
endmodule
```

Figure 16 illustrates the simulation of the Rx controller module. Upon receiving a start bit (rx == 0), it triggers the production of “reset_baud” and “parity_reset” signals, prompting a transition from the “RX_IDLE” state to the “RX_WAIT” state (as denoted by the red arrows in Figure 16). While in the “RX_WAIT” state, the module awaits the arrival of the “halfsignal”. Upon its arrival, the ‘reset_baud” signal is generated, facilitating the transition from the “RX_WAIT” state to the “RX_DATA” state (as represented by the green arrows in Figure 16).

In the RX_DATA state, detecting the “fullbaud” signal triggers the “shift” signal (as denoted by the purple arrows in Figure 16). When the “done” signal is detected, it generates the “clear” signal and changes the state back to the “RX_IDLE” state (as shown by the blue arrows in Figure 16). Notably, the “done” signal only activates the “clear” signal in the “RX_DATA” state, evident from the absence of output upon its activation in the other states (as illustrated in the blue box in Figure 16). Additionally, the rxcontroller module recommences operation upon detecting a new start bit (as observed at time = 560 ns in Figure 16) without any reset signals. Furthermore, when the “reset” signal is activated (low, active low), it transitions the state back to the “RX_IDLE” state (as indicated by the yellow arrow in Figure 16).

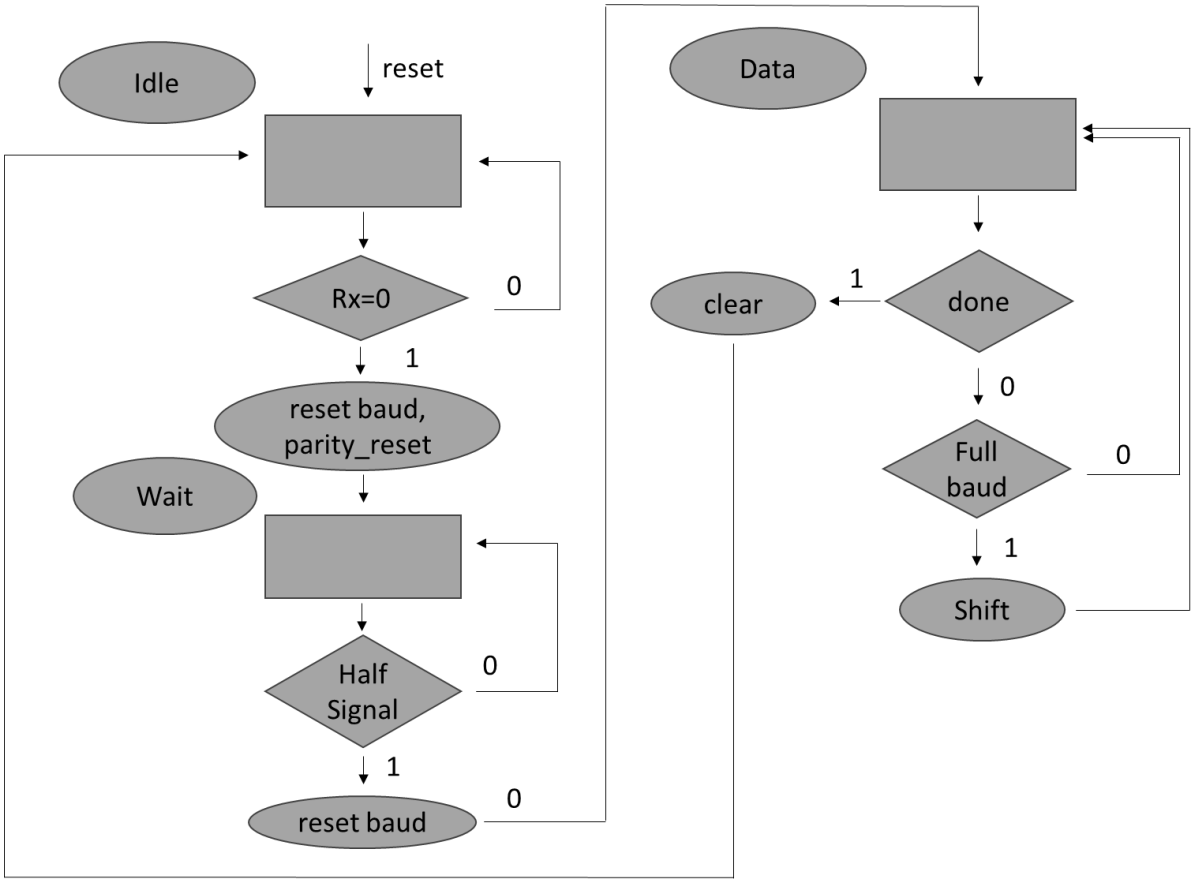


Figure 15 Rx Controller ASM

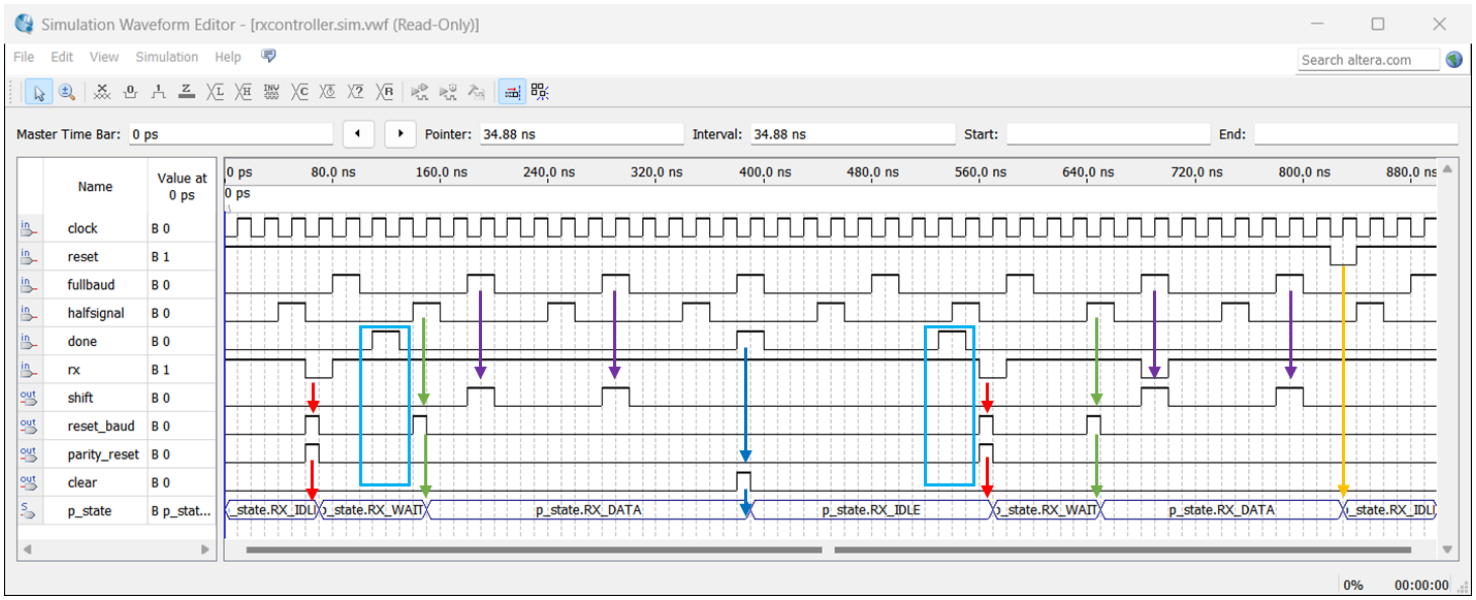


Figure 16 Rx Controller Simulation

2.9 IR Encoder

The module "tx_ir" serves as a transformer, converting UART data frames into IR data frames. As depicted in Figure 18, when a bit is high in the UART system, it corresponds to a low bit in the IR system, and vice versa. Notably, the pulse duration for high bits in the IR system spans three-sixteenths of the bit frame duration. The "IR" signal from baud rate generator comprises high pulses occurring during three-sixteenths of each baud interval. To construct the IR data frame, the UART data bit values are inverted and combined with the "IR" signal using AND logic, as illustrated in Figure 17. This process ensures the accurate transformation of data between UART and IR formats for reliable transmission. Figure 19 illustrates the simulation of the IR encoder. When the "sel" switch, which enables the IR mode, is set high and the "tx" signal (representing the data bit in UART) is low, the "tx_ir" signal will be high for three-sixteenths of the bit duration. Conversely, when "tx" is high or the "sel" switch is inactive, the "tx_ir" signal remains low.

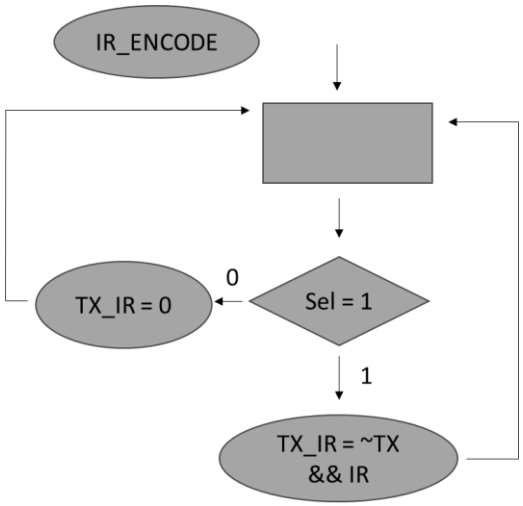


Figure 17 IR Encoder ASM

```
module tx_ir(
    input wire tx, ir, sel,
    output reg tx_ir
);


    always @(tx, ir, sel)
        begin
            if (sel)
                tx_ir = ~tx & ir;
            else
                tx_ir = 0;
        end
```

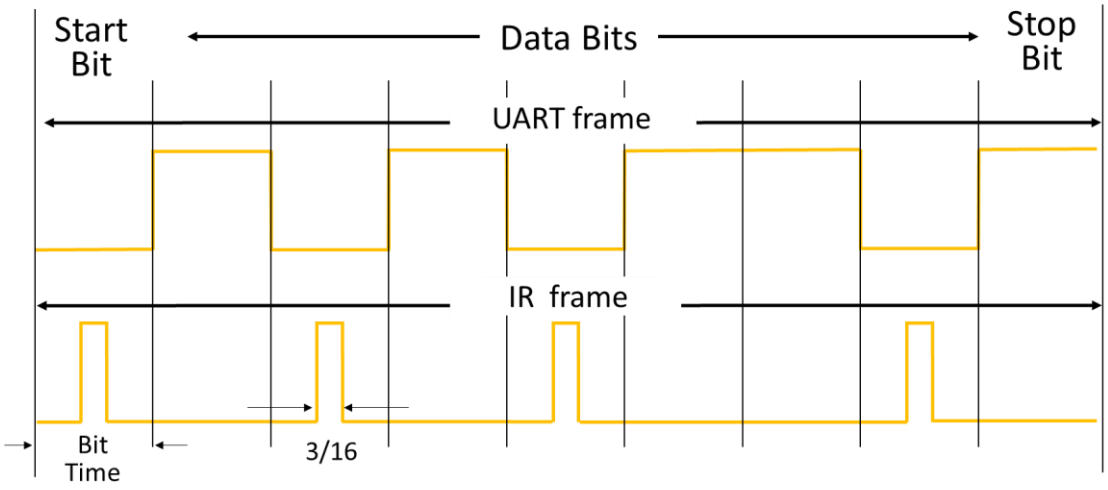


Figure 18 Comparison of UART & IR Transmission Data frame

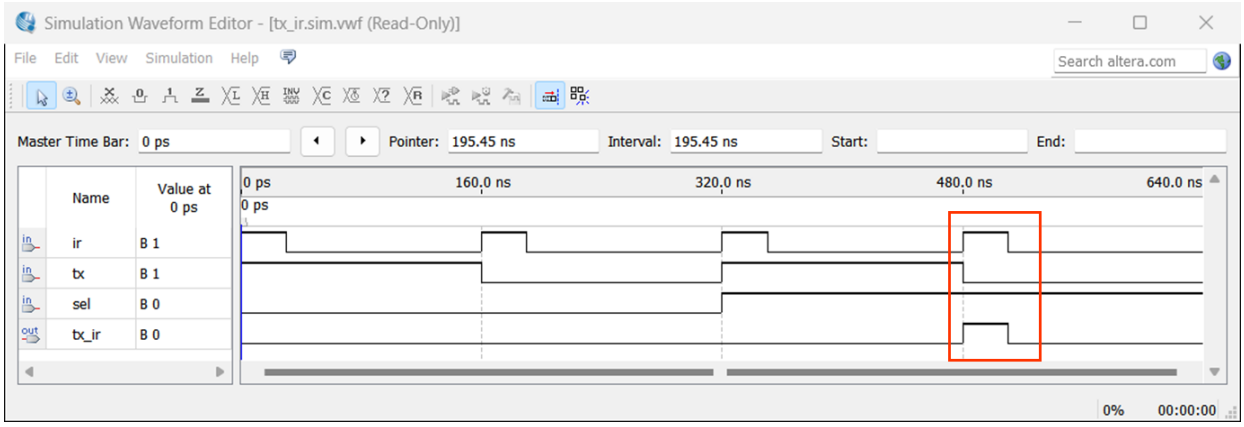


Figure 19 IR Encoder Simulation

2.10 Rx Selection & Half signal Selection

The modules “rx_ir” and “half_signal” act as signal selectors based on the status of the IR switch. When the IR switch is off, the “rx” output will be “rx_uart”, and the “half_signal” will be “half_baud”. Conversely, when the IR switch is on, the “rx” output will be “rx_ir”, and the “half_signal” will be “half_ir”, as depicted in Figures 20 and 21. This configuration allows for the flexible routing of signals depending on the operational mode selected by the IR switch, facilitating seamless integration into different communication systems.

Figures 22 and 23 depict simulations of the “rx_ir” and “half_signal” modules. When the “sel” signal is high, both the “rx” output and the “half_signal” signal are set to “rx_ir” and “half_ir”, respectively, regardless of whether the data input is low or high (as highlighted by the red boxes in Figures 22 and 23). Conversely, when the “sel” signal is low, the “rx” output becomes “rx_uart”, and the “half_signal” becomes “half_baud”.

```
module rx_ir(
    input rx_uart, rx_ir, sel,
    output wire rx
);


    assign rx =   (rx_uart & ~sel)|
                  (rx_ir & sel);

endmodule
```

```
module half_signal (
    input wire half_baud,
    input wire half_ir,
    input sel,
    output half_signal
);

    assign half_signal =  (sel == 1'b0) ? half_baud:
                          (sel == 1'b1) ? half_ir:
                          1'bx;
endmodule
```

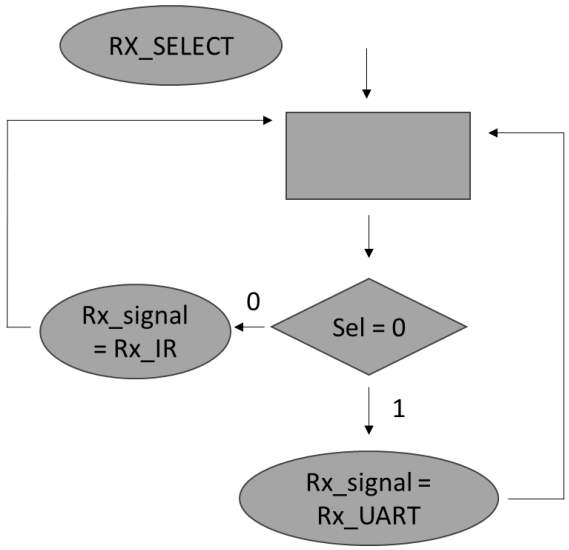


Figure 20 Rx Selection ASM

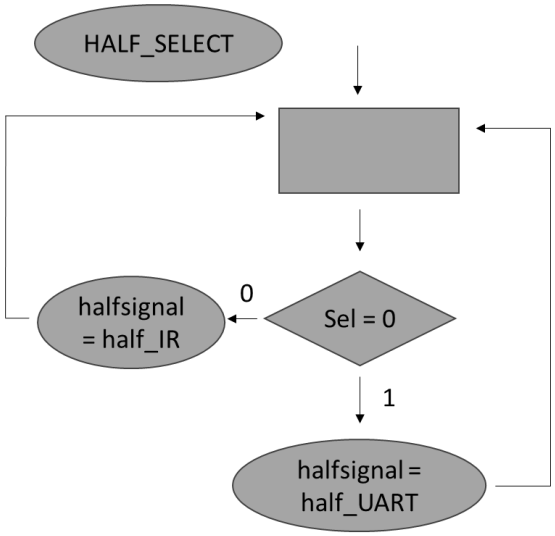


Figure 21 Half signal Selection ASM

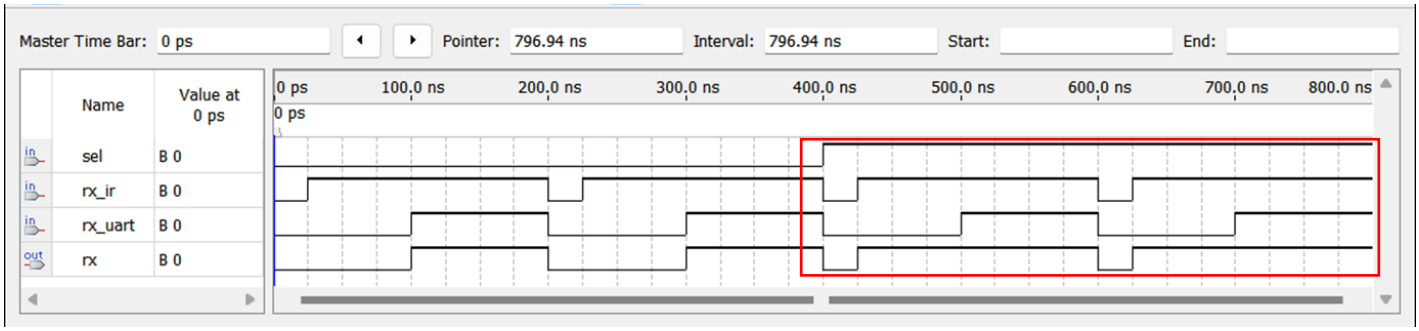


Figure 22 Rx Selection Simulation

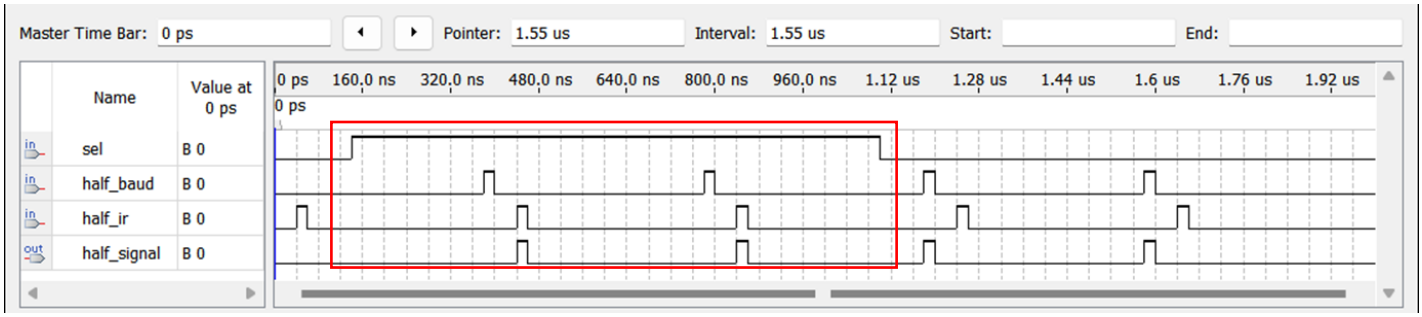


Figure 23 Half signal Selection Simulation

2.11 Parity Check

The "parity_check" module performs parity and framing error checking on input data packets. It operates in two states: P_IDLE and P_CHECK. In the P_IDLE state, the module waits for a signal to enable parity checking. Once activated, it transitions to the P_CHECK state. Here, the module splits the input data into two parts, extracts the most significant bits (MSBs) and least significant bits (LSBs), and performs parity and framing error checks. If a parity error is detected, the "parity_error" signal is set high, and if a framing error is detected, the "framing_error" signal is set high.

The "parity_error" signal signifies that the eighth bit of the "data_in" packet lacks the expected odd parity from the switch-derived input data. Similarly, the "framing_error" signal activates when the ninth bit of the "data_in" packet doesn't match the anticipated value of 1, as shown in Figure 24.

The module outputs the MSBs and LSBs of the input data packet to "data_out_msb" and "data_out_lsb" registers, respectively.

Figure 25 illustrates the simulation of the parity check module. In the P_IDLE state, indicated by a low (active-low) reset signal, the module remains inactive. Upon activation of the parity_set signal, signifying the initiation of parity checking, the module transitions to the P_CHECK state. Conversely, when the parity_reset signal is activated, the module reverts from the P_CHECK state to the P_IDLE state. Within the P_CHECK state, the module examines the eighth bit of the data_in input: if it's 0, no parity error is detected (as indicated by the yellow arrows), while a value of 1 triggers a parity_error output due to the odd parity requirement (as depicted by the red arrows). Similarly, the module examines the ninth bit of the data_in input in this state: a value of 0 results in a framing error (as depicted by the green arrows), whereas a value of 1 indicates no framing error (as shown by the blue arrows).

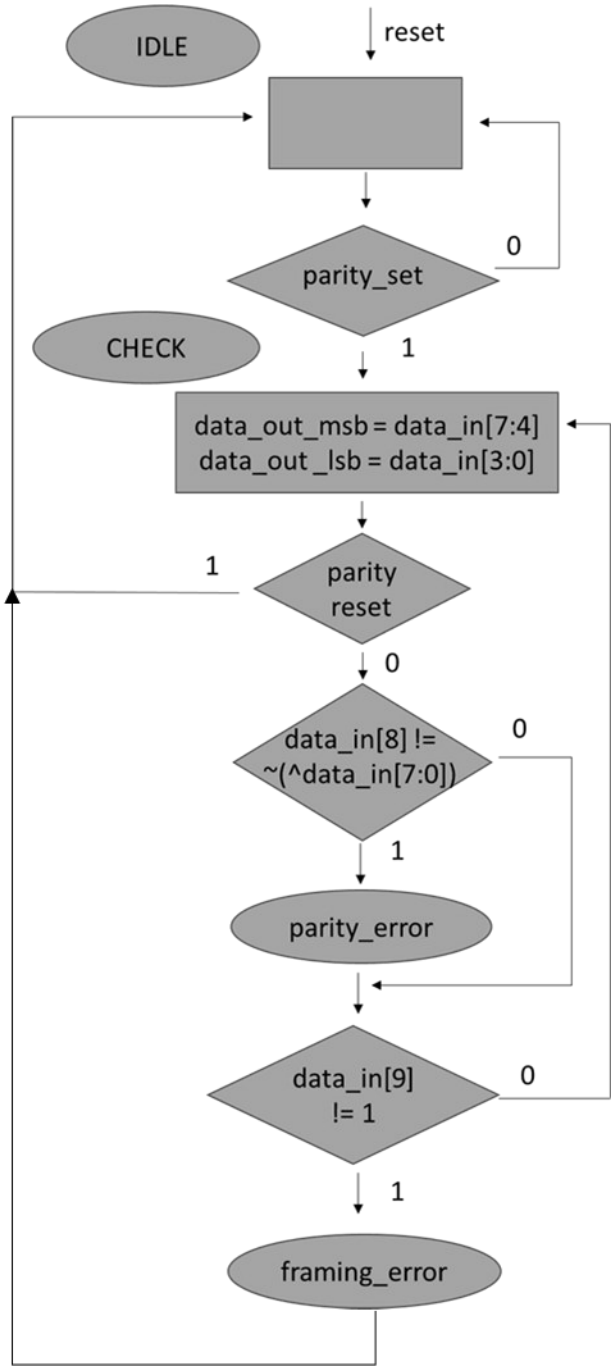


Figure 24 Parity Check ASM

```
module parity_check(
    input wire clock,
    input wire reset,
    input wire parity_set,
    input wire parity_reset,
    input [11:0] data_in,
    output reg [3:0] data_out_msb,
    output reg [3:0] data_out_lsb,
    output reg parity_error,
    output reg framing_error
);

    // Define states
    parameter P_IDLE = 1'b0, P_CHECK= 1'b1;

    regp_state, n_state;

    always @(posedge clock)
        begin
            if (~reset)
                p_state <= P_IDLE;
            else
                p_state <= n_state;
        end
    always @(p_state, parity_set, parity_reset,
                data_in)
        begin
            n_state = p_state;
            parity_error = 1'b0;
            framing_error = 1'b0;
            data_out_msb = 4'b0;
            data_out_lsb = 4'b0;
            case (p_state)
                P_IDLE:
                    begin
                        if (parity_set)
                            n_state = P_CHECK;
                    end
                P_CHECK:
                    begin
                        data_out_msb = data_in[7:4];
                        data_out_lsb = data_in[3:0];
                        if (parity_reset)
                            n_state = P_IDLE;
                        if (data_in[8] != ~(^data_in[7:0]))
                            parity_error = 1'b1;
                        if (data_in[9] != 1'b1)
                            framing_error = 1'b1;
                    end
            endcase
        end
endmodule
```

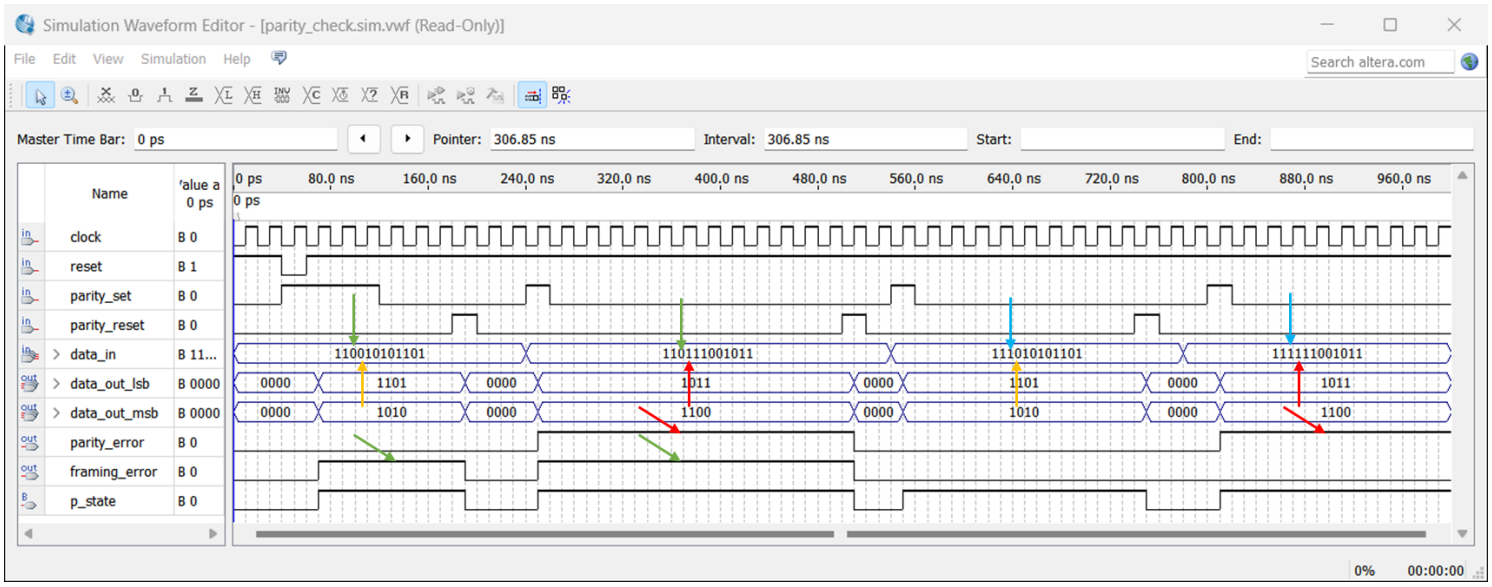


Figure 25 Parity Check Simulation

3. Schematic of the full system

The Verilog code comprises inputs including the DE2 clock, a reset button signal, a send button signal, an IR switch, and data values from switches intended for transmission, along with received data (either UART or IR). Outputs encompass four seven-segment displays, with two (sevseg sender1 and sevseg sender2) for transmitting data and the other two (sevseg receiver1 and sevseg receiver2) for receiving data. Additionally, there are outputs for transmitting data to the PC (tx_out) and for transmitting signals for IR communication (tx_ir). The module also features LEDs indicating parity error and framing error conditions.

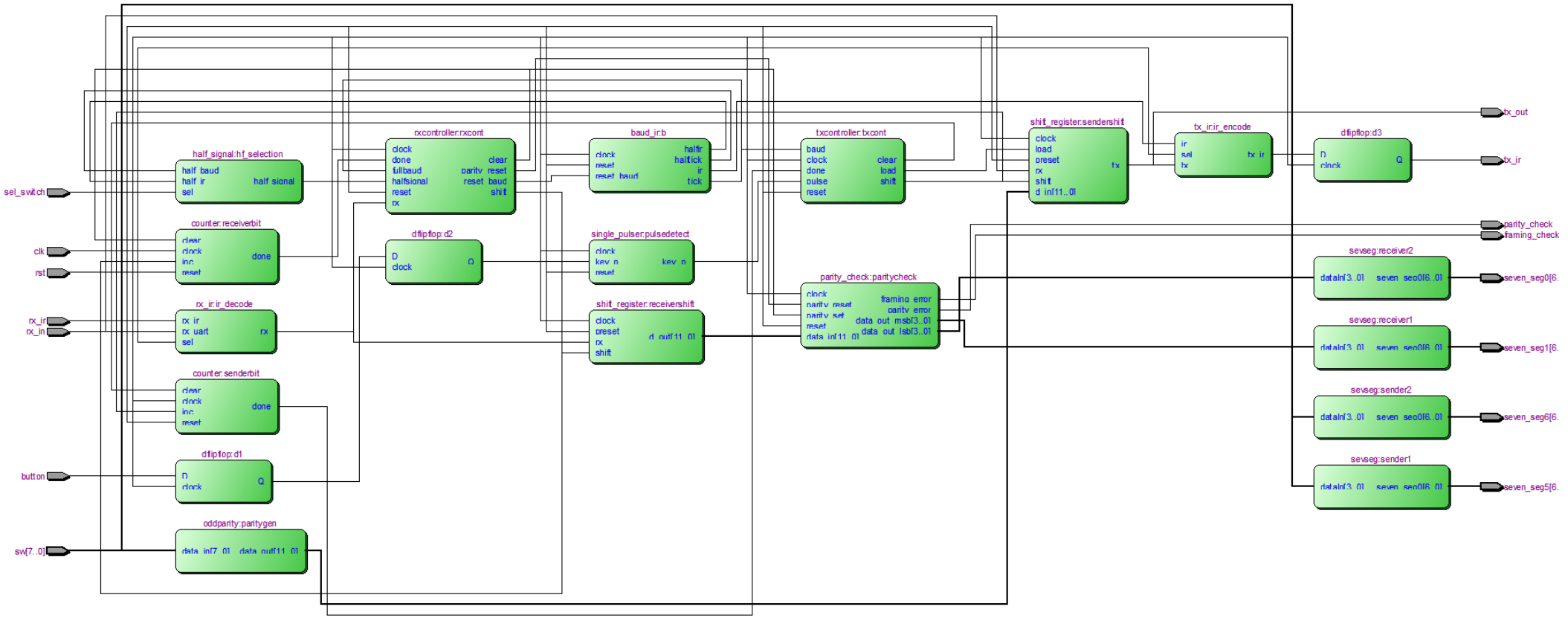
```
module uart_ir (
    input wire clk,
    input wire rst,
    input wire button,
    input wire [7:0] sw,
    input wire rx_in,
    input wire rx_ir,
    input wire sel_switch,
    output [6:0] seven_seg5, //HEX5 Sender MSB
    output [6:0] seven_seg6, //HEX6 Sender LSB
    output [6:0] seven_seg0, //HEX0 Receiver MSB
    output [6:0] seven_seg1, //HEX1 Receiver LSB
    output tx_out,
    output tx_ir,
    output parity_check,
    output framing_check
);

    wire button_meta;
    wire button_syn;
    wire baud_clk;
    wire halfbaud_clk;
    wire half_ir;
    wire half_signal;
    wire ir;
    wire rx_pulse;
    wire tx_ir_out;
```

```
baud_ir b (
    .clock(clk),
    .reset(rst),
    .reset_baud(reset_baud_r),
    .tick(baud_clk),
    .halftick(halfbaud_clk),
    .halfir(half_ir),
    .ir(ir)
);

sevseg sender1(
    .dataIn(sw[3:0]),
    .seven_seg0(seven_seg5)
);

sevseg sender2(
    .dataIn(sw[7:4]),
    .seven_seg0(seven_seg6)
);
sevseg receiver1(
    .dataIn(data_msb[3:0]),
    .seven_seg0(seven_seg1)
);

sevseg receiver2(
    .dataIn(data_lsb[3:0]),
    .seven_seg0(seven_seg0)
);

single_pulser pulsedetect(
    .clock(clk),
    .reset(rst),
    .key_n(button_syn),
    .key_p(pulse_s)
);

counter senderbit(
    .clock(clk),
    .reset(rst),
    .inc(shift_s),
    .clear(clear_s),
    .done(done_s)
);

counter receiverbit(
    .clock(clk),
    .reset(rst),
    .inc(shift_r),
    .clear(clear_r),
    .done(done_r)
);
```

```
txcontroller txcont(
    .clock(clk),
    .reset(rst),
    .baud(baud_clk),
    .done(done_s),
    .pulse(pulse_s),
    .shift(shift_s),
    .load(load_s),
    .clear(clear_s)
);

rxcontroller rxcont(
    .clock(clk),
    .reset(rst),
    .rx(rx_pulse),
    .done(done_r),
    .fullbaud(baud_clk),
    .halfsignal(half_signal),
    .shift(shift_r),
    .reset_baud(reset_baud_r),
    .parity_reset(parity_reset),
    .clear(clear_r)
);

shift_register sendershift(
    .clock(clk),
    .preset(rst),
    .shift(shift_s),
    .load(load_s),
    .d_in(data_out_parity[11:0]),
    .tx(tx_out),
    .rx(rx_in)
);

shift_register receivershift(
    .clock(clk),
    .preset(rst),
    .shift(shift_r),
    .rx(rx_pulse),
    .d_out(data_out_receiver[11:0])
);

oddparity paritygen(
    .data_in(sw[7:0]),
    .data_out(data_out_parity[11:0])
);
```

```
parity_check paritycheck(
    .clock(clk),
    .reset(rst),
    .parity_set(clear_r),
    .parity_reset(parity_reset),
    .data_in(data_out_receiver[11:0]),
    .data_out_msb(data_msb[3:0]),
    .data_out_lsb(data_lsb[3:0]),
    .parity_error(parity_check),
    .framing_error(framing_check)
);

dflipflop d1(
    .clock(clk),
    .D(button),
    .Q(button_meta)
);

dflipflop d2(
    .clock(clk),
    .D(button_meta),
    .Q(button_syn)
);

dflipflop d3(
    .clock(clk),
    .D(tx_ir_out),
    .Q(tx_ir)
);

tx_ir ir_encode(
    .tx(tx_out),
    .ir(ir),
    .sel(sel_switch),
    .tx_ir(tx_ir_out)
);

rx_ir ir_decode(
    .rx_uart(rx_in),
    .rx_ir(rx_ir),
    .sel(sel_switch),
    .rx(rx_pulse)
);

half_signal hf_selection(
    .half_baud(halfbaud_clk),
    .half_ir(half_ir),
    .sel(sel_switch),
    .half_signal(half_signal)
);

endmodule
```

4. RTF view of the full system.



5. Simulation of the full system

Figure 26 illustrates the transmission simulation of UART and IR communication. The "tx_out" signal accurately reflects the UART data frame, sourced from the switch inputs. Similarly, the "tx_ir" signal appropriately represents the IR data frame. Notably, when a data bit in UART is low, it corresponds to a high bit in the IR system for three-sixteenths of a bit time, as indicated by the yellow boxes. Moreover, when the "txcontroller" module is in the TX_DATA state and detects the "baud" signal, the "shift" signal transitions high at the correct timing, as highlighted by the red arrows. Additionally, upon receiving the "done" signal, the state transitions to the TX_IDLE state, as demonstrated by the green box.

Figures 27 and 28 illustrate the reception simulation of UART and IR communication. In the UART reception segment, upon detecting the start bit (rx == 0), the system resets the baud rate (green arrow) and transitions the "rxcontroller" from RX_IDLE to RX_WAIT. It then waits for the half_signal (halfbaud), resetting the baud rate again (green arrow), and moves to RX_DATA. In RX_DATA, received data (rx_in) is assigned to the MSBs of the shift register (red arrows in Figure 27). Lastly, correctness is confirmed for "data_msb" and "data_lsb", with framing error triggered by the ninth bit being 0 (yellow arrow).

Similarly, when the "sel" enables the IR communication. Figure 28 illustrates this transition, where the system detects the start bit in IR mode ("rx_ir" == 0 for 2.45 µs) and reset baud rate, waiting for the half_signal (half_ir), and resetting the baud rate again (green arrows). In RX_DATA, received data (rx_ir) is assigned to the MSBs of the shift register (red arrows in Figure 28). Lastly, correctness is validated for "data_msb" and "data_lsb", with a parity error triggered by the eighth bit being 1 (blue arrow in Figure 28).

Figure 26 Overall Simulation (Transmitting) UART+IR

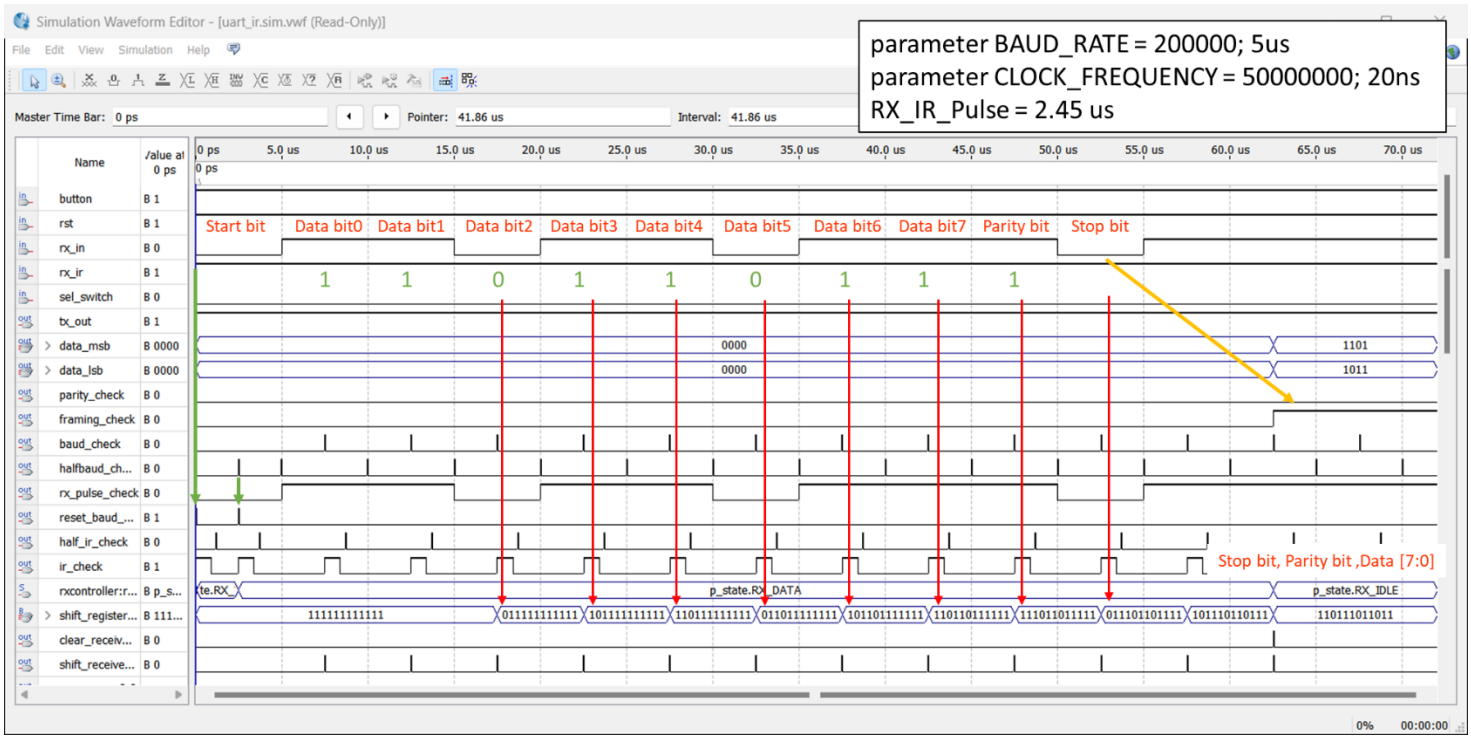


Figure 27 Overall Simulation (Receiving) UART

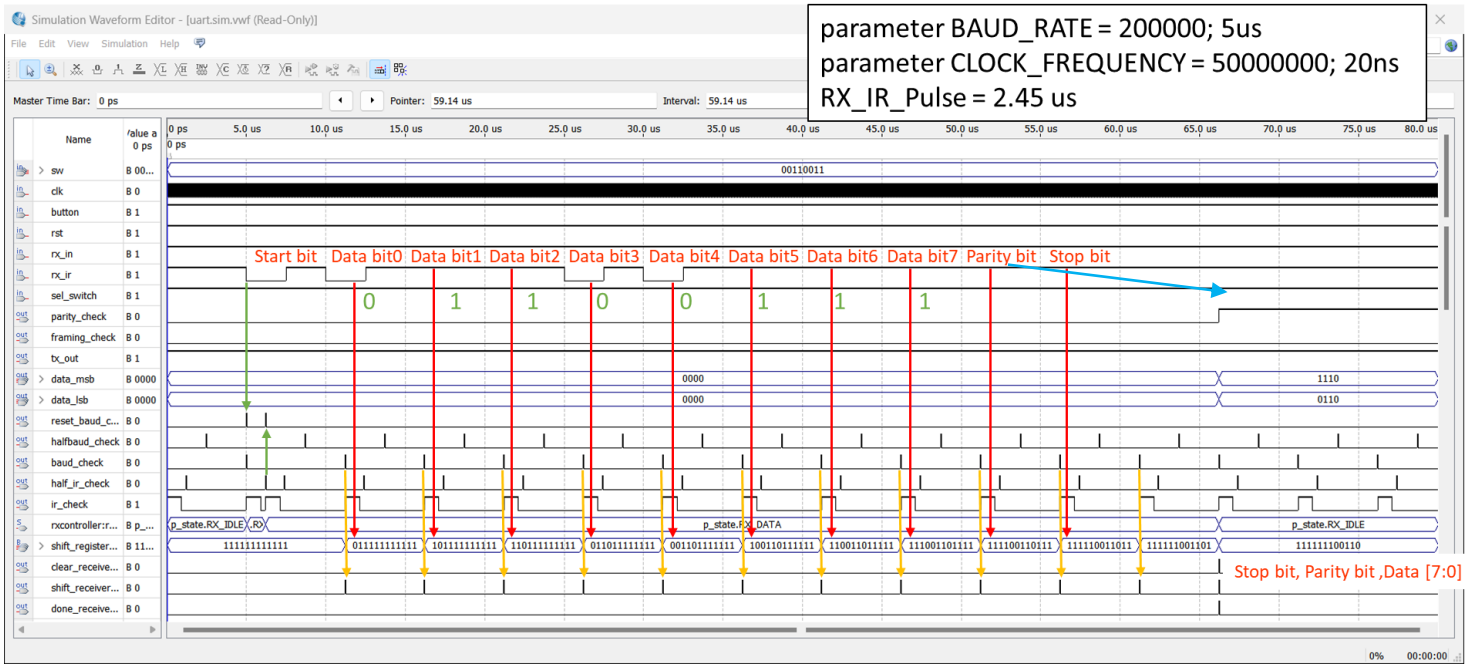


Figure 28 Overall Simulation (Receiving) IR

6. Test Results & Conclusion

The output results from the seven-segment display aligned precisely with the simulation, faithfully reflecting the values from switch_13 to switch_6, as depicted in Figure 29. Furthermore, the “tx_out” output, designed for transmission to the PC, was both visible and accurate on Putty. Figure 29 illustrates that when the HEX value of the switches is ‘41’, it sends ‘A’ to the screen. Similarly, when the HEX value is ‘42’ and ‘43’, it transmits ‘B’ and ‘C’ to the screen, respectively.

In the receiving aspect of UART, when the user inputs ‘c’ on Putty, the data is sent back through the board and accurately displayed as ‘63’ (the HEX value corresponding to the ASCII code of ‘c’). Regarding IR communication, upon enabling the IR switch and pressing the button, data transmission occurs using the HSDL-3201, reflecting the transmitted data back to display on the DE2 board, as depicted in Figure 30. Additionally, by connecting the IRDA_TX pin (pin AE24) to the LED, LED blinking confirms successful IR transmission, serving as a practical validation of the transmission process.

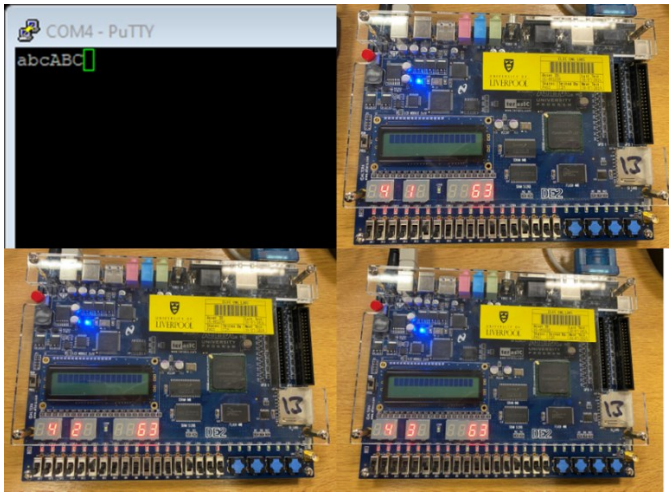


Figure 29 Seven-segment display on DE2 board and Putty on the PC screen

This project represents a comprehensive integration of UART and IR communication systems on the DE2 board, showcasing the versatility of FPGA-based designs. By utilizing Verilog modules and simulations, the project demonstrates robust data transmission and reception capabilities. The system effectively handles ASCII character exchange via UART, providing real-time feedback through Putty terminal connections and displaying received data on seven-segment displays. Furthermore, the integration of IR communication offers wireless data transfer capabilities, enhancing the system's versatility and practicality. However, it's worth noting that a limitation was encountered during experimentation: the green LED on some DE2 boards was found to be non-functional, impacting the ability to observe parity or frame errors directly on the board. Despite this constraint, the project successfully illustrates the implementation of bidirectional communication protocols, highlighting the FPGA's efficiency in facilitating seamless data exchange across different interfaces.

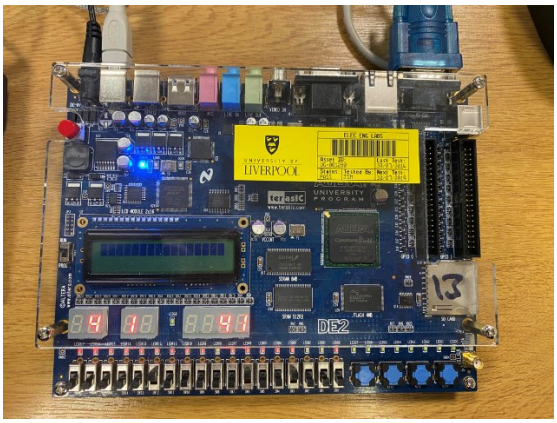


Figure 30 Seven-segment display on DE2 board with IR mode enable