

PROYECTO PROTOCOLO UART FULL-DUPLEX

IE-0523: CIRCUITOS
DIGITALES II

ISAÍ GONZÁLEZ S.
DANIEL DE LA O R.
JAVIER SACA

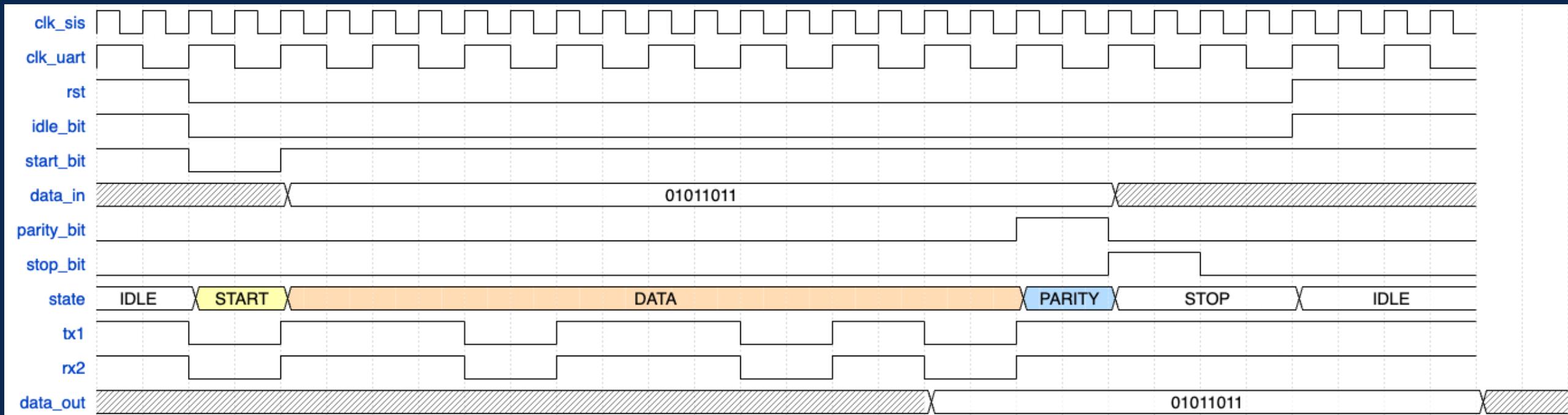
INDICE DE CONTENIDOS

```
4  # Prevent database truncation if the transaction fails
5  abort("The Rails environment is running in production mode!
6  require 'spec_helper'
7  require 'rspec/rails'
8
9  require 'capybara/rspec'
10 require 'capybara/rails'
11
12 Capybara.javascript_driver = :webkit
13 Category.delete_all; Category.create!(name: "Electronics")
14 Shoulda::Matchers.configure do |config|
15   config.integrate do |sp|
16     sp.with.test_framework :rspec
17     sp.with.library :rails
18   end
19 end
20
21 # Add additional requires below this line if you need them
22
23 # Requires supporting ruby files with custom matchers
24 # in spec/support/ and its subdirectories. This directory
25 # is relative to the root of your project.
26 # in _spec.rb will both be required by default.
27 # You can add additional files to spec/support/ if you
28 # want to split them up.
29 # end with _spec.rb. You can configure the
30 # :keyword_finding in the configuration file
31 # to find matching files in subdirectories.
32
33 # No results found for 'mongoid'
```

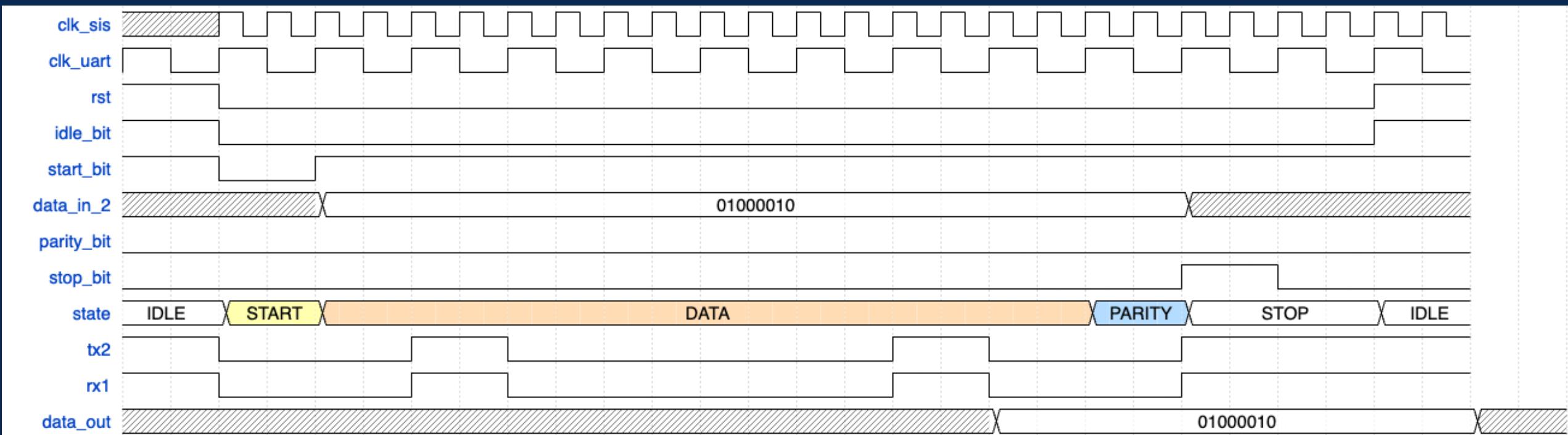
1. UART1 (TRANSMISSOR)	3
2. UART2 (RECEIVER)	5
3. WAVEFORM GTKWAVE	7
	9
	11

/ WAVEDROAM:

DATA_IN 1



DATA_IN 2



1 /
UART1(TRANSMISSOR)

1/ UART1

```
v UART1_dut.v
module UART1_dut(
    input wire clk_uart,
    input wire clk_sis,
    input wire rst,
    input wire start_bit,
    input wire [7:0] data_in,
    input wire stop_bit,
    input wire rx1,
    output reg tx1,
    output reg [7:0] data_out
);

// Declaración de los estados del transmisor UART
localparam IDLE = 3'b000,
    START = 3'b001,
    DATA = 3'b010,
    PARITY = 3'b011,
    STOP = 3'b100;

reg [2:0] state, next_state;
reg idle_bit;
reg [3:0] cnt_bits, next_cnt_bits;
reg parity_bit;
reg [7:0] shift_register;
reg [2:0] cont_bit;
reg update_out;

// **Lógica Secuencial
always @(posedge clk_uart or posedge rst) begin
    if (rst) begin
        // Reinicia todos los registros al estado inicial
        state      <= IDLE;           // Comienza en el estado IDLE
        tx1       <= 1'b1;            // Línea en alto (reposo)
        parity_bit <= 1'b0;           // Inicializa el bit de paridad
        cnt_bits   <= 4'b0;
        idle_bit   = 1;
    end else begin
        // Actualización de los registros en función de la lógica combinacional
        state      <= next_state;
        cnt_bits   <= next_cnt_bits;
    end
end
end
```

```
// **Lógica Combinacional
always @(*) begin
    // Valores predeterminados para evitar latches
    next_state = state;           // Por defecto, el estado no cambia
    next_cnt_bits = cnt_bits;     // Por defecto, el contador no cambia

    // Máquina de estados finita (FSM) //Se va armando la señal de tx1.
    case (state)
        IDLE: begin
            idle_bit = 1;
            tx1 = 1'b1;
            if (idle_bit)
                next_state = START;
        end

        START: begin
            idle_bit = 0;
            tx1 = 1'b0;
            next_state = DATA;
        end

        DATA: begin
            // Transmite los bits de datos en serie uno por uno
            idle_bit = 0;
            tx1 = data_in[cnt_bits];    // Selecciona el bit correspondiente LSB
            next_cnt_bits = cnt_bits + 1;
            data_out[0 + cnt_bits] = rx1;
            if (cnt_bits == 7)          // Si todos los bits han sido transmitidos
                next_state = PARITY;
        end

        PARITY: begin
            idle_bit = 0;
            parity_bit = ^data_in;     // Calcula la paridad XOR de los datos
            tx1 = parity_bit;         // Transmite el bit de paridad
            next_state = STOP;        // Cambia al estado de parada
        end

        STOP: begin
            idle_bit = 0;
            tx1 = 1'b1;                // Transmite el bit de parada (alto)
            if (stop_bit)
                next_state = IDLE;
        end
    endcase
endmodule
```

2 / U A R T 2
(RECEIVER)

2 / UART 2

```

module UART2_dut(
    // Entradas
    input wire clk_sis,           // Reloj del sistema
    input wire clk_uart,          // Reloj para las operaciones UART
    input wire rst,
    input wire start_bit,
    input wire [7:0] data_in,      // Dato para paridad y transmisión en `tx2`)
    input wire stop_bit,
    input wire rx2,               // Línea de recepción en serie desde el transmisor
    // Salidas
    output reg tx2,              // Línea de transmisión en serie hacia el rx1
    output reg [7:0] data_out     // Salida paralela que almacena el dato recibido
);
// Definición de estados para la máquina de estados finita (FSM)
localparam IDLE = 3'b000,
    START = 3'b001,
    DATA = 3'b010,
    PARITY = 3'b011,
    STOP = 3'b100;

// Registros internos para la FSM y la lógica del módulo
reg [2:0] state, next_state;
reg idle_bit;
reg [3:0] cnt_bits, next_cnt_bits; // Contador de bits recibidos
reg parity_bit;                  // Bit de paridad calculado
reg [7:0] shift_register;        // Registro de desplazamiento para almacenar el dato recibido
reg [2:0] cont_bit;              // Contador auxiliar para desplazamiento en `data_out`
reg update_out;                 // Bandera para actualizar la salida paralela

// **Lógica Secuencial**
// Actualización del estado y registros en el flanco positivo del reloj o reset
always @(posedge clk_uart or posedge rst) begin
    if (rst) begin
        // Inicialización de los registros en caso de reset
        state      <= IDLE;
        tx2       <= 1'b0;
        parity_bit <= 1'b0;
        cnt_bits   <= 0;
    end else begin
        // Actualización de los estados y contadores
        state      <= next_state;
        cnt_bits   <= next_cnt_bits;
    end
end

```

```

// **Lógica Combinacional**
// Generación de los estados siguientes y valores de salida
always @(*) begin
    // Valores predeterminados para evitar comportamientos indefinidos
    next_state = state;
    next_cnt_bits = cnt_bits;
    idle_bit = 1;                                // Por defecto, el sistema está en reposo

    // Máquina de estados finita (FSM)
    case(state)
        IDLE: begin
            idle_bit  = 0;                         // Detecta que estamos en reposo
            tx2       = 1'b1;                        // Línea de transmisión en alto (estado inactivo)
            if (!idle_bit)                          // Si detecta actividad, pasa al estado START
                | next_state = START;
        end

        START: begin
            tx2       = 1'b0;                         // Envía el bit de inicio
            next_state = DATA;                      // Cambia al estado de recepción de datos
        end

        DATA: begin
            // Transmite el bit actual de `data_in` por `tx2`
            tx2 = data_in[0 + cnt_bits];
            // Incrementa el contador de bits recibidos
            next_cnt_bits = cnt_bits + 1;

            // Si se han recibido 8 bits, pasa al estado de paridad
            if (cnt_bits == 7) begin
                next_state = PARITY;
            end

            // Construye el dato paralelo a partir de los bits recibidos en `rx2`
            data_out[7 - cnt_bits] = rx2;
            cont_bit = cont_bit + 1;                // Actualiza el contador auxiliar
        end

        PARITY: begin
            parity_bit = ^data_in;                  // Calcula el bit de paridad (XOR de `data_in`)
            tx2       = parity_bit;                // Envía el bit de paridad por `tx2`
            next_state = STOP;                   // Cambia al estado de parada
        end

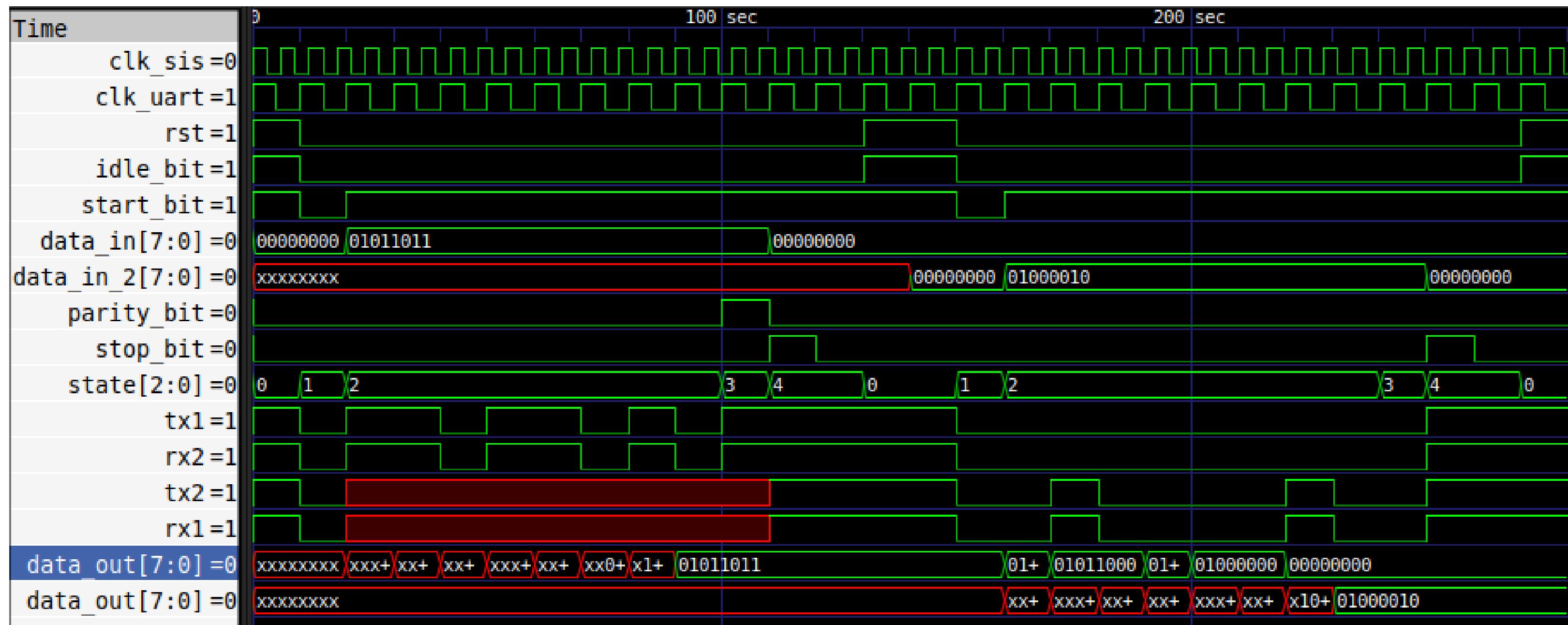
        STOP: begin
            // Envía el bit de parada (alto) y vuelve al estado IDLE
        end
    endcase
end

```

3 / WAVEFORM

GTKWAVE

3 / WAVEFORM



\TESTBENCH

```
module UART_tb;
    wire clk_uart;
    wire clk_sis;
    wire rst;
    wire start_bit;
    wire [7:0] data_in;
    wire [7:0] data_in_2;
    wire stop_bit;
    wire tx2;
    wire tx1;
    wire rx2;
    wire rx1;
    wire serial1;           // Cable de conexion entre tx1 y rx2
    wire serial2;           // Cable de conexion entre tx2 y rx1

    UART1_dut dut1 [
        .clk_sis(clk_sis),
        .clk_uart(clk_uart),
        .rst(rst),
        .start_bit(start_bit),
        .data_in(data_in),
        .stop_bit(stop_bit),
        .tx1(serial1),      //Conexion entre UART's
        .rx1(serial2)       // Conexion entre UART's
    ];

    UART2_dut dut2 (
        .clk_sis(clk_sis),
        .clk_uart(clk_uart),
        .rst(rst),
        .start_bit(start_bit),
        .data_in_2(data_in_2),
        .stop_bit(stop_bit),
        .tx2(serial2),
        .rx2(serial1)
    );

    UART_tester tester (
        .clk_sis(clk_sis),
        .clk_uart(clk_uart),
        .rst(rst),
        .start_bit(start_bit),
        .data_in(data_in),
        .data_in_2(data_in_2),
        .stop_bit(stop_bit)
    );

```

MUCHAS GRACIAS

```
    render() {
      return (
        <React.Fragment>
          <div className="py-5">
            <div className="container">
              <Title name="our" title="product">
              <div className="row">
                <ProductConsumer>
                  {(value) => {
                    |   |   |   console.log(value)
                    |   |   |
                  }}
                </ProductConsumer>
              </div>
            </div>
          </div>
        <React.Fragment>
```