Cátedra de Arquitectura de Computadoras

Trabajo Práctico N° II
UART

Carrizo, Aixa Mariel
Piñero, Tomás Santiago
19 de Noviembre de 2020

Índice

Índice **1**

1. Enunciado **2**

2. Desarrollo **2**
2.1. ALU . . . 3
2.1.1. Diseño . . . 3
2.2. UART . . . 4
2.2.1. Diseño . . . 4
2.2.1.1. *Baud rate generator* . . . 4
2.2.1.2. Receptor (RX) . . . 7
2.2.1.3. Transmisor (TX) . . . 11
2.2.1.4. *top_uart* . . . 16
2.2.2. *Testbench* . . . 17
2.3. Interfaz . . . 19
2.3.1. Diseño . . . 19
2.3.2. *Testbench* . . . 21
2.4. *top_all* . . . 23

3. Cálculo de frecuencia máxima **28**

4. Conclusiones **28**

Referencias **28**

1. Enunciado

El objetivo de este trabajo es implementar en FPGA, en Verilog, un módulo UART que se comunique con el módulo ALU realizado en el trabajo práctico anterior. Los requerimientos del trabajo son los siguientes:

- El *baudrate* al que trabaja el módulo UART debe ser parametrizable;
- Debe existir una interfaz que comunique el módulo UART con el módulo ALU;
- El módulo UART y la interfaz deben ser diseñados como una máquina de estados (FSM);
- Validar el desarrollo por medio de *Test Bench*.

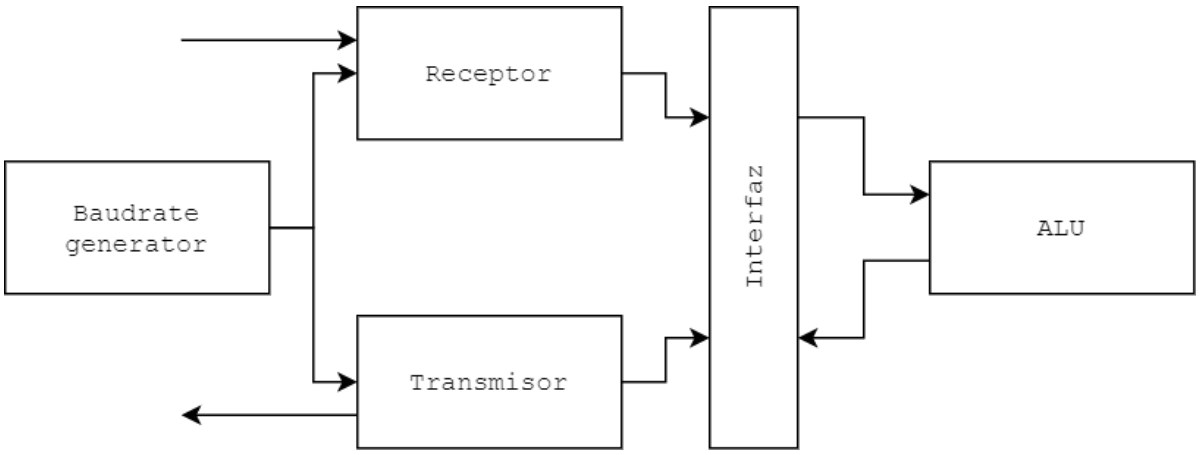

Figura 1: Esquema del proyecto.

2. Desarrollo

El diseño se dividió en cuatro módulos principales:

- **alu**: tiene tres entradas y una única salida. Las dos entradas que corresponden a los datos y la salida son de tamaño parametrizable, `N_BITS`, mientras que la entrada correspondiente a la operación a realizar es fija (6 bits).
- **top_uart**: se utiliza para corroborar el funcionamiento correcto de los módulos que conforman el UART, conectando la salida del receptor a la entrada del transmisor.
- **interface**: es el encargado de realizar la comunicación entre el UART y la ALU.
- **top_all**: instancia los módulos que constituyen el UART, la interfaz y la ALU.

2.1. ALU

2.1.1. Diseño

Es el diseño que se utilizó en el trabajo práctico I, con una modificación: las operaciones a realizar son parámetros locales del módulo. Su código es el siguiente:

Código 1: Código fuente de la ALU.

```
1  `timescale 1ns / 1ps
2
3  module alu#(
4    //Parameters
5    parameter    N_BITS           = 8
6  )
7  (
8    localparam ADD =6'b100000;
9    localparam SUB =6'b100010;
10   localparam AND =6'b100100;
11   localparam OR  =6'b100101;
12   localparam XOR =6'b100110;
13   localparam SRA =6'b000011;
14   localparam SRL =6'b000010;
15   localparam NOR =6'b100111;
16
17   //inputs
18   input wire [N_BITS-1:0]     i_dato_A,
19   input wire [N_BITS-1:0]     i_dato_B,
20   input wire [5:0]            i_operacion,
21
22   //output
23   output reg [N_BITS-1:0]     o_alu
24 );
25
26     always@(*) begin:alu
27         case(i_operacion)
28         ADD:  o_alu = i_dato_A + i_dato_B;     //suma
29         SUB:  o_alu = i_dato_A - i_dato_B;     //resta
30         AND:  o_alu = i_dato_A & i_dato_B;     //and
31         OR:   o_alu = i_dato_A | i_dato_B;     //or
32         XOR:  o_alu = i_dato_A ^ i_dato_B;     //xor
33         SRA:  o_alu = i_dato_A >>> i_dato_B;  //SRA (arithmetic): extiende
               el signo
34         SRL:  o_alu = i_dato_A >> i_dato_B;   //SRL (logic): insterta 0
35         NOR:  o_alu = ~(i_dato_A | i_dato_B); //nor
36         default: o_alu = {N_BITS{1'b0}};      //default = 0
37         endcase
38     end
39 endmodule
```

2.2. UART

2.2.1. Diseño

El diseño del módulo UART se basa en el libro '*FPGA Prototyping by Verilog examples*' [1]. Dicho diseño consta de cuatro módulos:

- Receptor;
- Transmisor;
- Generador de *baudrate*;
- Interfaz FIFO.

En este caso, la interfaz FIFO no fue implementada debido a que se reemplaza por la interfaz que comunica el UART con la ALU.

Los parámetros que utiliza el módulo UART son los siguientes:

- `N_BITS`: el tamaño de los datos a recibir/enviar. En este caso, los datos son de 8 bits.
- `F_CLOCK`: la frecuencia a la que está trabajando el circuito. Para este trabajo la frecuencia es de 50 [MHz].
- `BAUDRATE`: número de bits por segundo (bps). El valor utilizado es 9600 bps.
- `SAMPLING`: cantidad de *ticks* a contar para detectar los bits de *stop*. Dichos *ticks* son 16, 24 y 32 para 1, 1.5 y 2 bits de *stop*, respectivamente.

En la Figura **2** se observa cómo el receptor y el transmisor interpretan el dato: un bit de inicio, 8 bits de datos y un bit de stop. El bit de paridad no se utiliza.

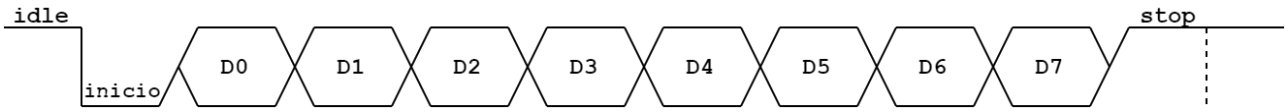

Figura 2: Recepción/Transmisión de bits.

A continuación se explicará el diseño de cada uno de los módulos del UART junto a su validación.

2.2.1.1 *Baud rate generator*

Genera una señal de muestreo cuya frecuencia es 16, 24 ó 32 veces el *baud rate* del UART. Esta señal funciona como *ticks* válidos para evitar añadir un nuevo dominio de *clock*.

La fórmula para la tasa de muestreo es la siguiente:

$$ticks = \frac{f_{clock}}{baudrate * sampling} \tag{1}$$

Como en este caso se trabaja con un una frecuencia de *clock* de 50 [MHz], 9600 bps y un bit de *stop*, la ecuación **1** queda:

$$ticks = \frac{50 * 10^6}{9600 * 16}$$

$$ticks = \frac{50 * 10^6}{153600}$$

$$ticks = 326$$

Por lo tanto, se generará un 1 lógico cada 326 ciclos del *clock*.

En el Código **2** se muestra el diseño de este módulo, que consiste en un contador descendente de `N_BITS` según el resultado de `CUENTA`, que utiliza la ecuación **1**. Al tener un resultado con punto flotante, se realiza una reducción del contador para corroborar que haya llegado a 0.

Código 2: Código fuente de '*baudrate_generator*'.

```
1  `timescale 1ns / 1ps
2
3  module baudrate_generator#
4  (
5    parameter F_CLOCK  = 50E6,
6    parameter BAUDRATE = 9600,
7    parameter SAMPLING = 16
8  )
9  (
10   input  wire i_clk, i_reset,
11   output wire o_tick
12 );
13
14   localparam CUENTA = F_CLOCK / (BAUDRATE * SAMPLING);
15   localparam N_BITS = $clog2(CUENTA);
16
17   reg  [N_BITS-1:0] contador;
18
19   always @(posedge i_clk) begin:cuenta
20     if(i_reset || o_tick)
21       contador <= CUENTA - 1;
22     else
23         contador <= contador - 1;
24   end
25
26   //como se trabaja con valores flotantes (CUENTA)
27   //se realiza una reduccion al contador para poner
28   //en 1 ó 0 la salida, indicada por |
29   assign o_tick = ~(|contador); //cuando contador == 0, se produce un tick
30 endmodule
```

A continuación se muestra el código fuente del *testbench* y su resultado.

Código 3: *Testbench* de '*baudrate_generator*'.

```
`timescale 1ns / 1ps

module tb_baudrate_generator();

    reg clk, reset;
    wire ticks;

    initial begin
        clk   = 1'b0;
        reset = 1'b1;

        #10
        reset = 1'b0;

        #654
        $finish();
    end

    always #1 clk = ~clk;

    baudrate_generator
    #(
        .F_CLOCK(50E6), .BAUDRATE(9600), .SAMPLING(16)
     )
    u_bd_gen
     (
        .i_clk(clk), .i_reset(reset),
        .o_tick(ticks)
     );
endmodule
```

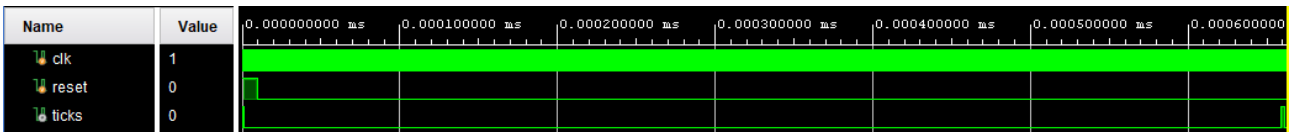

Figura 3: Resultado de '*tb_baudrate_generator*'.

2.2.1.2 Receptor (RX)

Es una máquina de estados que evoluciona según las entradas `i_rx` e `i_ticks`. Su tarea consiste en recibir el dato bit a bit, moviéndolos hacia la derecha `DATA_BITS` veces hasta obtener el dato completo. `DATA_BITS` es un parámetro que vale 8 para este caso.

En la siguiente Figura se muestra el diagrama de estados:

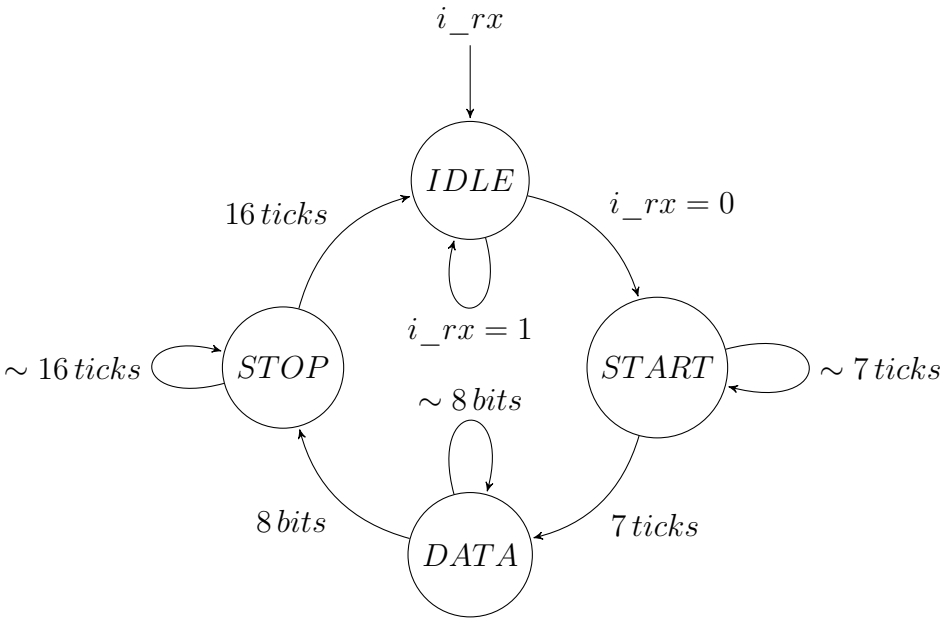

Figura 4: Diagrama de estados del RX.

El receptor tiene como estado inicial el estado *IDLE*. El autómata queda en este estado hasta recibir un bit de inicio a través de la señal de entrada *i_rx*, pasando así al estado *START*. En este estado, espera 7 *ticks* para situarse en el medio del bit de inicio y así detectarlo. Una vez detectado este bit, se pasa al estado *DATA*, donde se esperan 16 *ticks* para asegurarse que se encuentra en el centro del bit recibido y así guardarlo. Ya recibidos los 8 bits, se pasa al estado *STOP*, en el que lee un único bit de stop, indica que la recepción terminó mediante la salida `o_rx_done` y vuelve al estado inicial.

El código es el siguiente:

Código 4: Código fuente de '*rx_uart*'.

```
1  `timescale 1ns / 1ps
2
3  module rx_uart#
4  (
5    parameter DATA_BITS = 8,
6    parameter N_TICKS = 16 //Ticks para los bits de stop
7  )
8  (
9    input  wire          i_clk, i_reset,
10   input  wire          i_rx, i_ticks,
11   output reg           o_rx_done,
12   output wire [DATA_BITS-1:0] o_data_out
13 );
14
```

```
15  //declaracion de los estados
16  localparam [1:0] IDLE  = 2'b00;
17  localparam [1:0] START = 2'b01;
18  localparam [1:0] DATA  = 2'b10;
19  localparam [1:0] STOP  = 2'b11;
20
21  //declaracion de variables
22  //2 en funcion de los data bits: para contar la cantidad de bits
23  //3 en funcion del n_ticks: 16n
24
25  reg [1:0]       state_reg, next_state;
26  reg [2:0]       bits_counter, next_bit_counter; //contador de los bits
27  reg [3:0]       sampling_counter, next_sampling_counter; //contador de
        ticks
28  reg [DATA_BITS-1:0] buffer, next_buffer; //bits recibidos
29    reg [DATA_BITS-1:0] data_out;
30
31  //cambios de estado
32  always @(posedge i_clk) begin:check_state
33    if(i_reset)
34      begin
35        state_reg      <= IDLE;
36        sampling_counter <= 0;
37        bits_counter   <= 0;
38        buffer           <= 0;
39      end
40    else
41      begin
42        state_reg      <= next_state;
43        sampling_counter <= next_sampling_counter;
44        bits_counter   <= next_bit_counter;
45        buffer           <= next_buffer;
46      end
47  end//check_state
48
49  //estados siguientes
50  always @(*) begin:next
51    next_state = state_reg;
52    o_rx_done  = 1'b0;
53
54    next_sampling_counter = sampling_counter;
55    next_bit_counter    = bits_counter;
56    next_buffer         = buffer;
57
58    case(state_reg)
59      IDLE:
60      begin
61        if(~i_rx)
62        begin
63          next_state = START;
64
65          next_sampling_counter = 0;
66        end
67      end
68
69
70
71
```

```
72        START:
73        begin
74          if(i_ticks)
75          begin
76            if(sampling_counter == 7)
77            begin
78              next_state = DATA;
79
80              next_sampling_counter = 0;
81              next_bit_counter    = 0;
82            end
83            else
84              next_sampling_counter = sampling_counter + 1;
85          end
86        end
87
88        DATA:
89        begin
90          if(i_ticks)
91          begin
92            if(sampling_counter == 15)
93            begin
94              next_sampling_counter = 0;
95              next_buffer = {i_rx, buffer[7:1]};//orderna los bits recibidos
96
97              if(next_bit_counter == (DATA_BITS-1))
98                next_state = STOP;
99              else
100               next_bit_counter = bits_counter + 1;
101           end
102           else
103             next_sampling_counter = sampling_counter + 1;
104         end
105       end
106
107       STOP:
108       begin
109         if(i_ticks)
110         begin
111           if(sampling_counter == (N_TICKS-1))
112           begin
113             next_state = IDLE;
114             o_rx_done  = 1'b1;
115             data_out   = buffer;
116           end
117           else
118             next_sampling_counter = sampling_counter + 1;
119         end
120       end
121     endcase
122   end
123
124   //output
125   assign o_data_out = data_out;
126
127 endmodule
```

El *testbench* y su resultado se muestran a continuación:

Código 5: *Testbench* de '*rx_uart*'.

```
`timescale 1ns / 1ps

module tb_rx_uart();

    localparam  F_CLOCK  = 50E6;
    localparam  BAUDRATE = 9600;
    localparam  DBITS = 8;
    localparam  TICKS = 16;
    localparam CNT_BITS = $clog2(DBITS);

    reg              clk, reset;
    reg              rx;
    wire             tick;
    wire             rx_done;
    wire [DBITS-1:0] data_out;

    reg [CNT_BITS-1:0] ii;

    initial begin
        reset = 1'b1;
        clk = 1'b0;
        rx  = 1'b1;

        #40 reset = 1'b0;

        #104320
        rx = 1'b0; //bit de inicio

        //comienzo a enviar el dato random de N_BITS bits
        //de LSB a MSB
        for(ii = 0; ii < DBITS; ii = ii + 1)
            #104320 rx = 1'b1;

        #104320
        rx = 1'b1; //bit de stop
    end

    always @(*)
    begin
        if(rx_done == 1)
        begin
            $display("------ TEST OK! ------");
            $finish;
        end
    end

    always #10 clk = ~clk;

    rx_uart
    #(
        .DATA_BITS(DBITS), .N_TICKS(TICKS)
     )
    u_rx_uart
     (
        .i_clk(clk), .i_reset(reset),
```

```
56          .i_rx(rx), .i_ticks(tick),
57          .o_rx_done(rx_done),
58          .o_data_out(data_out)
59      );
60
61      baudrate_generator
62      #(
63          .F_CLOCK(F_CLOCK), .BAUDRATE(BAUDRATE), .SAMPLING(TICKS)
64      )
65      u_bd_generator
66      (
67          .i_clk(clk), .i_reset(reset),
68          .o_tick(tick)
69      );
70
71  endmodule
```

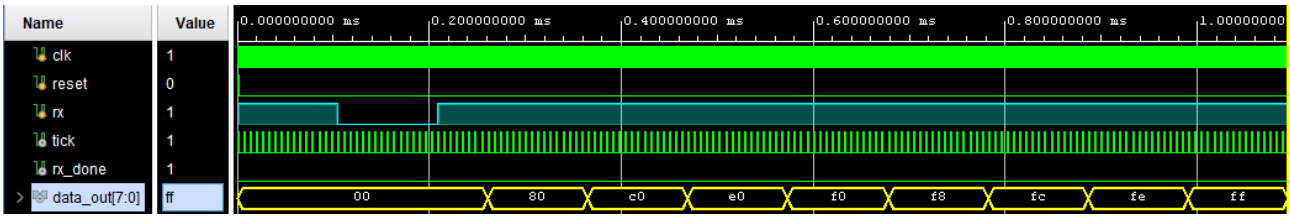

Figura 5: Resultado de '*tb_rx_uart*'.

2.2.1.3 Transmisor (TX)

Es una máquina de estados que evoluciona según las entradas `i_tx_start` e `i_data_in`. Su tarea consiste en enviar bit a bit los 8 bits recibidos mediante la señal `i_data_in` como se mencionó anteriormente (ver Figura **2**). Esto se logra moviendo los bits del dato hacia la derecha a medida que se van transmitiendo.

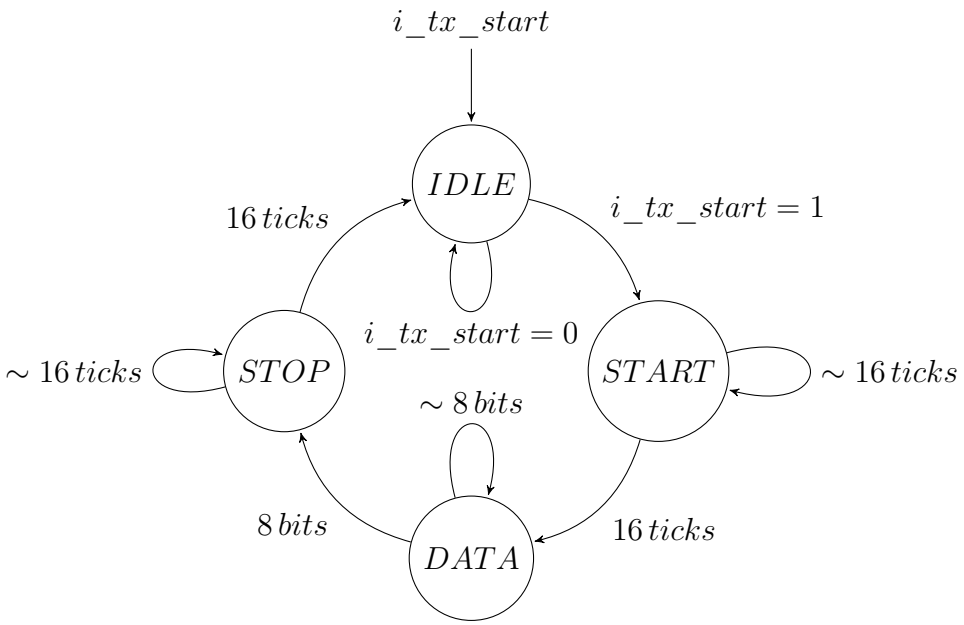

Figura 6: Diagrama de estados del TX.

En la Figura **6** se muestra la máquina de estados del transmisor, que es similar a la del receptor. La diferencia reside en que en el estado *START* se esperan 16 *ticks* en lugar de 7, ya que la posibilidad de sobre muestreo no existe.

El código del transmisor es el siguiente:

Código 6: Código fuente de '*tx_uart*'.

```
`timescale 1ns / 1ps

module tx_uart#
(
  parameter DATA_BITS = 8,
  parameter N_TICKS = 16 //Ticks para los bits de stop
)
(
  input  wire          i_clk, i_reset,
  input  wire          i_tx_start, i_ticks,
  input  wire [DATA_BITS-1:0] i_data_in,
  output reg           o_tx_done,
  output wire          o_data_out
);

  //declaracion de los estados
  localparam [1:0] IDLE  = 2'b00;
  localparam [1:0] START = 2'b01;
  localparam [1:0] DATA  = 2'b10;
  localparam [1:0] STOP  = 2'b11;

  //declaracion de variables
  reg [1:0]     state_reg, next_state;
  reg [2:0]     bits_counter, next_bit_counter; //contador de los bits
  reg [3:0]       sampling_counter, next_sampling_counter; //contador de
     ticks
  reg [DATA_BITS-1:0] buffer, next_buffer;
  reg          tx_data, next_tx;

  //cambios de estado
  always @(posedge i_clk) begin:check_state
    if(i_reset)
    begin
      state_reg <= IDLE;

      sampling_counter <= 0;
      bits_counter   <= 0;
      buffer         <= 0;
      tx_data      <= 1'b1;
    end
    else
    begin
      state_reg <= next_state;

      sampling_counter <= next_sampling_counter;
      bits_counter   <= next_bit_counter;
      buffer         <= next_buffer;
      tx_data      <= next_tx;
    end
  end

```

```
51  //estado siguiente
52  always @(*) begin:next
53    next_state = state_reg;
54    o_tx_done  = 1'b0;
55
56    next_sampling_counter = sampling_counter;
57    next_bit_counter    = bits_counter;
58    next_buffer       = buffer;
59
60    case(state_reg)
61      IDLE:
62      begin
63        next_tx = 1'b1;
64        if(i_tx_start)
65        begin
66          next_state = START;
67
68          next_sampling_counter = 0;
69          next_buffer           = i_data_in;
70        end
71      end
72
73      START:
74      begin
75        next_tx = 1'b0;
76        if(i_ticks)
77        begin
78          if(sampling_counter == 15)
79          begin
80            next_state = DATA;
81
82            next_sampling_counter = 0;
83            next_bit_counter    = 0;
84          end
85          else
86            next_sampling_counter = sampling_counter + 1;
87        end
88      end
89
90      DATA:
91      begin
92        next_tx = buffer[0];
93        if(i_ticks)
94        begin
95          if(sampling_counter == 15)
96          begin
97            next_sampling_counter = 0;
98            next_buffer = buffer >> 1;
99            if(bits_counter == (DATA_BITS-1))
100             next_state = STOP;
101           else
102             next_bit_counter = bits_counter + 1;
103         end
104         else
105           next_sampling_counter = sampling_counter + 1;
106       end
107     end
108
```

```
109       STOP:
110       begin
111         next_tx = 1'b1;
112         if(i_ticks)
113         begin
114           if(sampling_counter == (N_TICKS-1))
115           begin
116             next_state = IDLE;
117             o_tx_done  = 1'b1;
118           end
119           else
120             next_sampling_counter = sampling_counter + 1;
121         end
122       end
123     endcase
124   end
125
126   //output
127   assign o_data_out = tx_data;
128 endmodule
```

A continuación se muestra el *testbench* realizado junto con el resultado obtenido.

Código 7: *Testbench* de '*tx_uart*'.

```
1  `timescale 1ns / 1ps
2
3  module tb_tx_uart();
4
5      localparam  F_CLOCK  = 50E6;
6      localparam  BAUDRATE = 9600;
7      localparam  DBITS = 8;
8      localparam  TICKS = 16;
9
10     reg               clk, reset;
11     reg               start;
12     wire              tick;
13     wire              tx_done;
14     reg  [DBITS-1:0]  data_in;
15     wire              data_out;
16
17     initial begin
18         reset  = 1'b1;
19         clk    = 1'b0;
20         start  = 1'b0;
21         //shiftea un 1 cada 16 ticks
22         #40 reset = 1'b0;
23
24         #104320
25         data_in = {$urandom()};
26
27         #104320
28         start = 1'b1;
29
30         #104320
31         wait(tx_done == 1);
32
33         reset = 1'b1;
34         #40
```

```
35          $finish;
36      end
37
38      always #10 clk = ~clk;
39
40      tx_uart
41      #(
42          .DATA_BITS(DBITS), .N_TICKS(TICKS)
43       )
44      u_tx_uart
45       (
46          .i_clk(clk), .i_reset(reset),
47          .i_tx_start(start), .i_ticks(tick),
48          .i_data_in(data_in),
49          .o_tx_done(tx_done),
50          .o_data_out(data_out)
51       );
52
53      baudrate_generator
54      #(
55          .F_CLOCK(F_CLOCK), .BAUDRATE(BAUDRATE), .SAMPLING(TICKS)
56       )
57      u_bd_generator
58       (
59          .i_clk(clk), .i_reset(reset),
60          .o_tick(tick)
61       );
62
63  endmodule
```

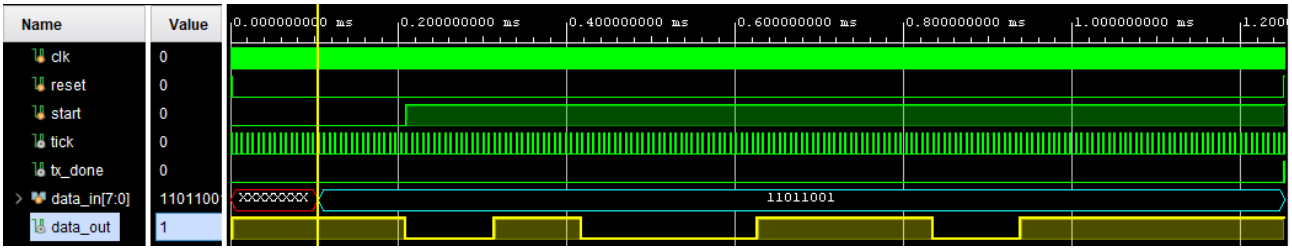

Figura 7: Resultado de '*tb_tx_uart*'.

2.2.1.4 *top_ uart*

Para realizar una prueba del módulo UART completo, se realizó un módulo que instancie los tres módulos explicados anteriormente. En dicha instancia, se conectó la salida del Receptor a la entrada del Transmisor, formando una conexión tipo *loopback*.

El código de este módulo es el siguiente:

Código 8: Código fuente de '*top_ uart*'.

```
1  `timescale 1ns / 1ps
2
3  module top_uart
4  #(
5      parameter N_BITS    = 8,
6      parameter F_CLOCK   = 50E6,
7      parameter BAUDRATE  = 9600,
8      parameter SAMPLING  = 16
9   )
10  (
11     input  wire i_clk, i_reset,
12     input  wire i_rx_top,
13     output wire o_tx_top,
14     output wire o_rx_done,
15     output wire o_tx_done
16  );
17
18     //signals
19     wire                tick;
20     wire                rx_done;
21     wire                tx_done;
22     wire [N_BITS-1:0]   rx_data_out;
23     wire [N_BITS-1:0]   tx_data_in;
24     wire                o_tx;
25
26     assign o_tx_top  = o_tx;
27     assign o_tx_done = tx_done;
28     assign o_rx_done = rx_done;
29
30     //instancia modulos con RX->TX (loopback)
31     //receptor
32     rx_uart u_rx_uart(
33         .i_clk(i_clk), .i_reset(i_reset),
34         .i_rx(i_rx_top), .i_ticks(tick),
35         .o_rx_done(rx_done), .o_data_out(rx_data_out)
36         );
37
38     //transmisor
39     tx_uart u_tx_uart(
40         .i_clk(i_clk), .i_reset(i_reset),
41         .i_tx_start(rx_done), .i_ticks(tick),
42         .i_data_in(rx_data_out),
43         .o_tx_done(tx_done), .o_data_out(o_tx)
44         );
45
46     //baudrate generator
47     baudrate_generator
48     #(
49         .F_CLOCK(F_CLOCK), .BAUDRATE(BAUDRATE), .SAMPLING(SAMPLING)
```

```
50      )
51      u_bd_generator
52      (
53          .i_clk(i_clk), .i_reset(i_reset),
54          .o_tick(tick)
55      );
56  endmodule
```

2.2.2. *Testbench*

Instancia el módulo '*top_uart*' y realiza el envío según los siguientes cálculos.

Al producirse 1 *tick* cada 326 ciclos de *clock*, se contarán los 16 *ticks* necesarios para el muestreo en 5216 ciclos. Como en este caso se utiliza una frecuencia de *clock* de 50 [MHz] (20 [ns]), se necesitarán 104320 [ns] para lograr la adquisición correcta de los bits a enviar/recibir.

El *testbench* genera un número sin signo de manera aleatoria, lo envía y espera a que el transmisor termine de enviar dicho número para terminar la ejecución. Esto se muestra en el Código **9** y en la Figura **8**.

Código 9: *Testbench* de '*top_uart*'.

```
1  `timescale 1ns / 1ps
2
3  module tb_uart();
4
5      localparam N_BITS   = 8;
6      localparam F_CLOCK  = 50E6;
7      localparam BAUDRATE = 9600;
8      localparam SAMPLING = 16;
9      localparam CNT_BITS = $clog2(N_BITS);
10
11     reg               clk, reset;
12     reg  [N_BITS-1:0] data;
13     reg               rx;
14     wire              tx_done;
15     wire              tx;
16
17     reg [CNT_BITS-1:0] ii;
18
19     /*
20      * cada 326 ciclos de reloj se produce 1 tick
21      * hay que contar 16 ticks, entonces
22      * 326*16 = 5216
23      *
24      * 1 ciclo            -> 20ns
25      * 326 ciclos (1tick) -> 104320ns
26      */
27     initial begin
28         clk   = 1'b0;
29         rx    = 1'b1;
30         reset = 1'b1;
31
32         #40
33         reset = 1'b0;
34
35         //Mando dato random
```

```
36          data = {$urandom()};
37
38          #104320
39          rx = 1'b0; //bit de inicio
40
41          //comienzo a enviar el dato random de N_BITS bits
42          //de LSB a MSB
43          for(ii = 0; ii < N_BITS; ii = ii + 1)
44              #104320 rx = data[ii];
45
46          #104320
47          rx = 1'b1; //bit de stop
48      end
49
50      always #10 clk = ~clk;
51
52      always @(posedge clk)
53      begin
54          if(tx_done == 1)
55          begin
56              $display("------ TEST OK! ------");
57              $finish;
58          end
59      end
60
61      top_uart
62      #(
63          .N_BITS(N_BITS), .F_CLOCK(F_CLOCK),
64          .BAUDRATE(BAUDRATE), .SAMPLING(SAMPLING)
65       )
66      u_top_uart
67      (
68          .i_clk(clk), .i_reset(reset),
69          .i_rx_top(rx),
70          .o_tx_top(tx), .o_tx_done(tx_done)
71      );
72  endmodule
```

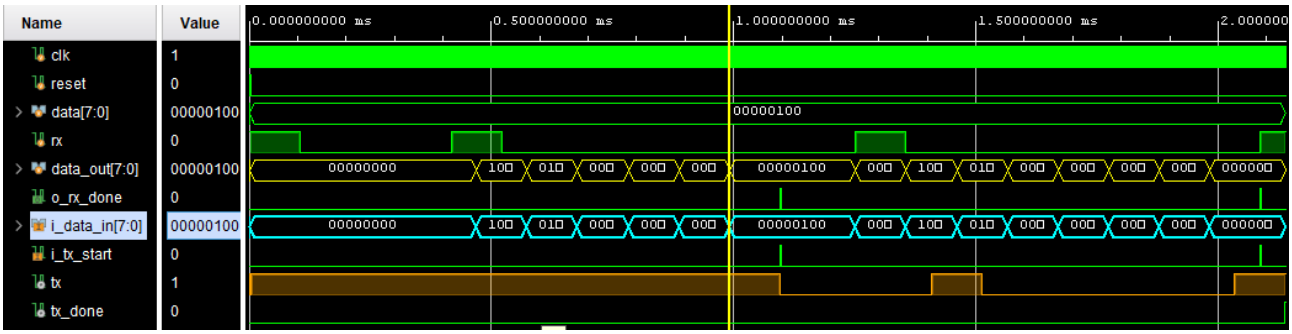

Figura 8: Resultado de '*tb_top_uart*'.

2.3. Interfaz

2.3.1. Diseño

Como se mencionó en la Sección **2**, este módulo es el encargado de comunicar el UART con la ALU.

Es una máquina de estados que evoluciona según la señal `i_rx_done`, que indica la recepción completa del dato. Si esta señal se encuentra en 1, el dato se asignará a la entrada de la ALU correspondiente al estado en que se encuentre y se pasará al estado siguiente, como se muestra en la Figura **9**.

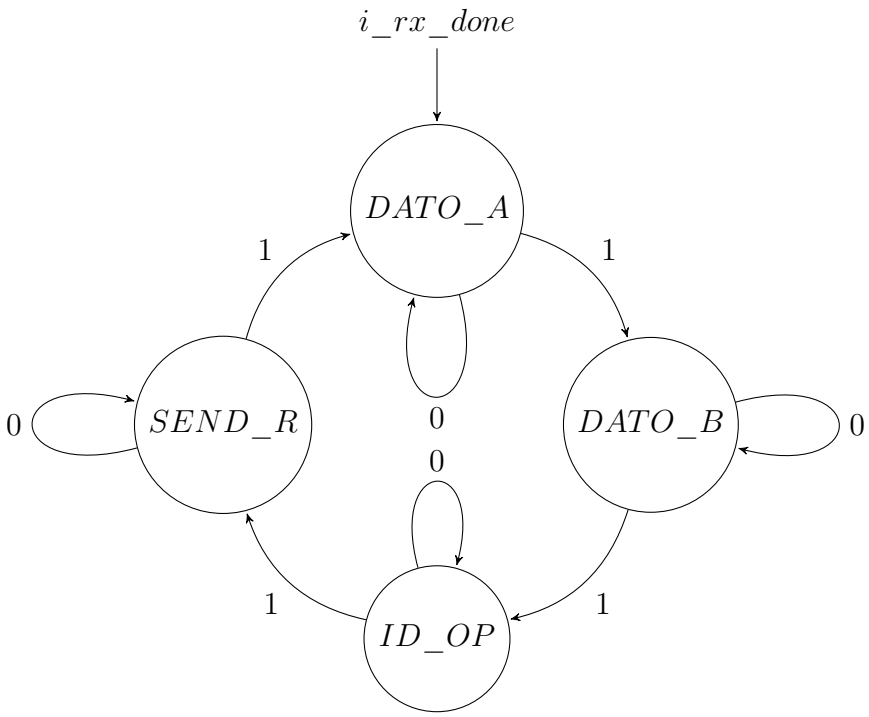

Figura 9: Diagrama de estados de la interfaz.

El código es el siguiente:

Código 10: Código fuente de '*interface*'.

```
1  `timescale 1ns / 1ps
2
3  module interface
4  #(
5      parameter DATA_BITS = 8,
6      parameter OP_BITS   = 6
7   )
8   (
9      input wire                  i_clk, i_reset,
10     input wire                  i_rx_done,
11     input       [DATA_BITS-1:0] i_rx_data_in,  //el dato ingresado
12     input       [DATA_BITS-1:0] i_alu_data_in, //el resultado de la ALU
13
14
15
```

```
16      output reg [DATA_BITS-1:0] o_dato_A,
17      output reg [DATA_BITS-1:0] o_dato_B,
18      output reg [OP_BITS-1:0]   o_op,
19      output reg                 o_tx_start,
20      output     [DATA_BITS-1:0] o_data_out     //dato transmitido
21  );
22
23      //estados
24      localparam [1:0] DATO_A   = 2'b00;
25      localparam [1:0] DATO_B   = 2'b01;
26      localparam [1:0] ID_OP    = 2'b10;
27      localparam [1:0] SEND_RES = 3'b11;
28
29      //registros para las transiciones
30      reg [1:0]           state_reg, next_state;
31
32      reg [DATA_BITS-1:0] dato_a    = {DATA_BITS {1'b0}};
33      reg [DATA_BITS-1:0] dato_b    = {DATA_BITS {1'b0}};
34      reg [OP_BITS-1:0]   operacion = {OP_BITS   {1'b0}};
35
36      assign o_data_out = i_alu_data_in;
37
38      always@(posedge i_clk) begin:states
39          if(i_reset)
40          begin
41              state_reg  <= DATO_A;
42              o_dato_A   <= {DATA_BITS {1'b0}};
43              o_dato_B   <= {DATA_BITS {1'b0}};
44              o_op       <= {OP_BITS   {1'b0}};
45          end
46          else
47          begin
48              state_reg  <= next_state;
49              o_dato_A   <= dato_a;
50              o_dato_B   <= dato_b;
51              o_op       <= operacion;
52          end
53      end//states
54
55      always @(posedge i_clk) begin:data
56          next_state = state_reg;
57
58          o_tx_start = 1'b0;
59
60          if(state_reg == SEND_RES)
61          begin
62              next_state  = DATO_A;
63              o_tx_start  = 1'b1;
64          end
65
66          if(i_rx_done)
67          begin
68              case(state_reg)
69              DATO_A:
70              begin
71                  dato_a    = i_rx_data_in;
72                  next_state = DATO_B;
73              end
```

```
74
75              DATO_B:
76              begin
77                  dato_b    = i_rx_data_in;
78                  next_state = ID_OP;
79              end
80
81              ID_OP:
82              begin
83                  operacion  = i_rx_data_in;
84                  next_state = SEND_RES;
85              end
86
87              SEND_RES:;
88              endcase
89          end
90      end
91  endmodule
```

2.3.2. *Testbench*

Consiste en transmitir el resultado de la suma de los datos recibidos. En este caso, dichos datos son los números 1 (dato A) y 2 (dato B). En el Código **11** se muestra el *testbench* y la Figura **10** su resultado.

Código 11: *Testbench* de '*interface*'.

```
1  `timescale 1ns / 1ps
2
3  module tb_interface();
4
5      localparam  DATA_BITS  = 8;  // # buffer bits
6      localparam  OP_BITS    = 6;  // Operation bits
7
8      //INPUT
9      reg                        clk, reset;
10     reg                        rx_done;
11     reg       [DATA_BITS-1:0]  rx_data_in;
12     wire      [DATA_BITS-1:0]  alu_data_in;
13
14     //OUTPUT
15     wire      [DATA_BITS-1 : 0] dato_a;
16     wire      [DATA_BITS-1 : 0] dato_b;
17     wire      [OP_BITS-1 : 0]   operation;
18     wire                        tx_start;
19     wire      [DATA_BITS-1:0]   data_out;
20
21     initial begin
22         clk        = 1'b0;
23         reset      = 1'b1;
24         rx_done    = 1'b0;
25         rx_data_in = {DATA_BITS{1'b0}};
26
27         #40 reset = 1'b0; // Desactivo la accion del reset.
28
29         rx_data_in = 8'b00000001; //data a
30         #20
```

```
31
32          rx_done = 1'b1;
33          #20
34          rx_done = 1'b0;
35
36          rx_data_in = 8'b00000010; //data b
37          #20
38
39          rx_done = 1'b1;
40          #20
41          rx_done = 1'b0;
42
43          rx_data_in = 8'b00100000; //suma
44          #20
45
46          rx_done = 1'b1;
47          #20
48          rx_done = 1'b0;
49
50          #60 reset = 1'b1;
51          $finish;
52      end
53
54      always #10 clk = ~clk; // Simulacion de clock.
55
56      interface
57      #(
58          .DATA_BITS    (DATA_BITS),
59          .OP_BITS      (OP_BITS)
60      )
61      u_interface
62      (
63          .i_clk              (clk),
64          .i_reset            (reset),
65          .i_rx_done          (rx_done),
66          .i_rx_data_in       (rx_data_in),
67          .i_alu_data_in      (alu_data_in),
68          .o_dato_A           (dato_a),
69          .o_dato_B           (dato_b),
70          .o_op               (operation),
71          .o_tx_start         (tx_start),
72          .o_data_out         (data_out)
73      );
74
75      alu #(
76          .NB_DATA (DATA_BITS)
77      )
78
79      u_alu (
80          .i_dato_A (dato_a),
81          .i_dato_B (dato_b),
82          .i_operacion (operation),
83          .o_alu (alu_data_in)
84      );
85
86  endmodule
```

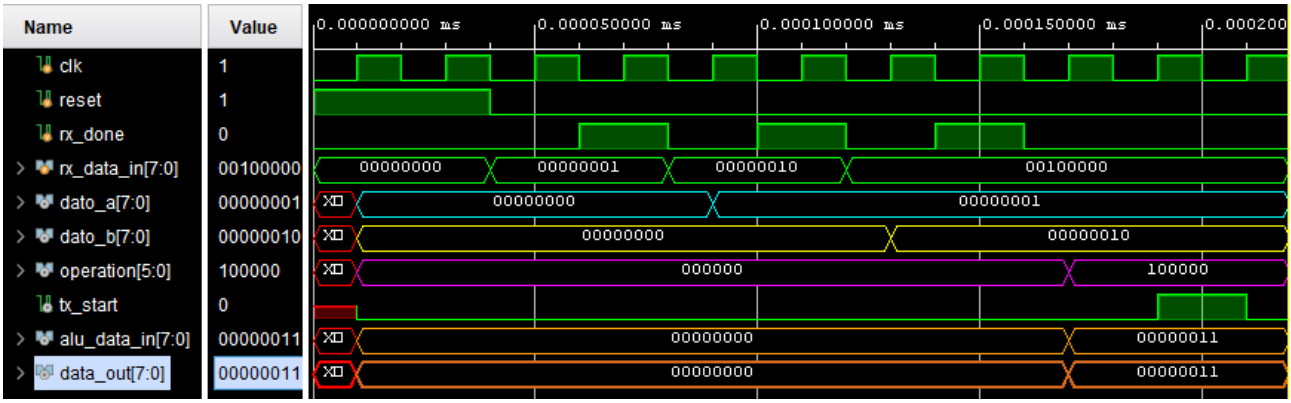

Figura 10: Resultado de '*tb_interface*'.

2.4. *top_all*

Es el módulo que se encarga de instanciar todos los componentes del diseño. Tiene como entrada las señales de *clock*, *reset* y la entrada de datos del receptor y, como salida el dato que se transmite (`o_tx_top`) y la finalización de la transmisión (`o_tx_done`).

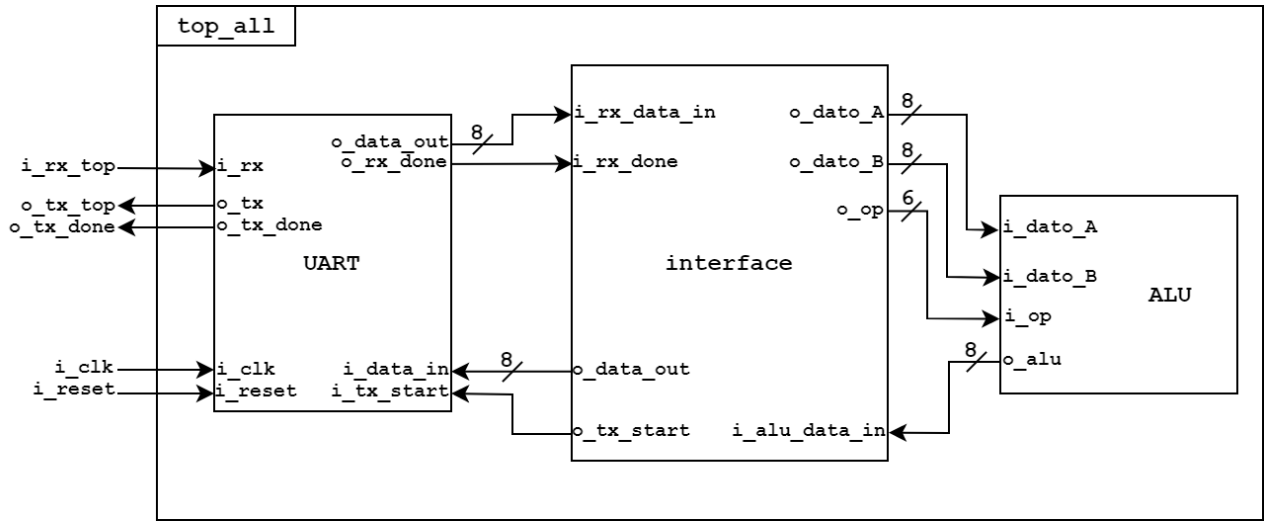

Figura 11: Esquema del módulo *'top_all'*.

En la Figura **11** se muestra l diseño que se sintetizó. El esquema resultante es el siguiente:

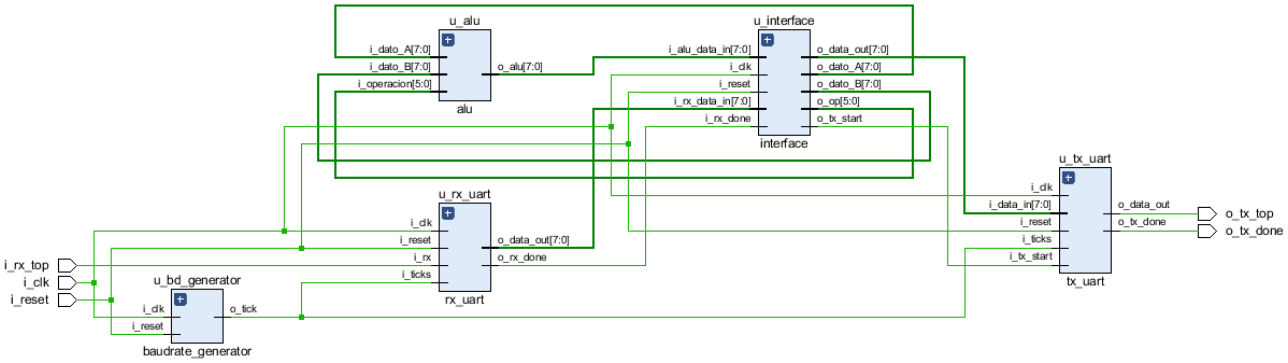

Figura 12: Esquema RTL del módulo *'top_all'*.

A continuación se muestra el código del diseño.

Código 12: Código fuente de '*top_all*'.

```
1  `timescale 1ns / 1ps
2
3  module top_all
4  #(
5      parameter N_BITS   = 8,
6      parameter OP_BITS  = 6,
7      parameter F_CLOCK  = 50E6,
8      parameter BAUDRATE = 9600,
9      parameter SAMPLING = 16
10 )
11 (
12     input  wire i_clk, i_reset,
13     input  wire i_rx_top,
14     output wire o_tx_top,
15     output wire o_tx_done
16 );
17
18     //signals
19     wire                tick;
20     wire                rx_done;
21     wire                tx_done;
22     wire                tx_start;
23     wire [N_BITS-1:0]   tx_data_in;
24     wire [N_BITS-1:0]   rx_data_out;
25     wire [N_BITS-1:0]   dato_a;
26     wire [N_BITS-1:0]   dato_b;
27     wire [N_BITS-1:0]   o_alu;
28     wire [OP_BITS-1:0]  operacion;
29     wire                o_tx;
30
31     assign o_tx_top  = o_tx;
32     assign o_tx_done = tx_done;
33
34     //==================== UART ====================
35
36     //instancia modulos
37     //receptor
38     rx_uart
```

```
39      #(
40          .DATA_BITS(N_BITS), .N_TICKS(SAMPLING)
41       )
42       u_rx_uart
43       (
44          .i_clk(i_clk), .i_reset(i_reset),
45          .i_rx(i_rx_top), .i_ticks(tick),
46          .o_rx_done(rx_done), .o_data_out(rx_data_out)
47       );
48
49      //transmisor
50      tx_uart
51      #(
52          .DATA_BITS(N_BITS), .N_TICKS(SAMPLING)
53       )
54       u_tx_uart
55       (
56          .i_clk(i_clk), .i_reset(i_reset),
57          .i_tx_start(tx_start), .i_ticks(tick),
58          .i_data_in(tx_data_in),
59          .o_tx_done(tx_done), .o_data_out(o_tx)
60        );
61
62      //baudrate generator
63      baudrate_generator
64      #(
65          .F_CLOCK(F_CLOCK), .BAUDRATE(BAUDRATE), .SAMPLING(SAMPLING)
66       )
67       u_bd_generator
68       (
69          .i_clk(i_clk), .i_reset(i_reset),
70          .o_tick(tick)
71       );
72
73      //=============================================
74
75       //interfaz
76       interface
77       #(
78          .DATA_BITS(N_BITS), .OP_BITS(OP_BITS)
79        )
80        u_interface
81        (
82          .i_clk(i_clk), .i_reset(i_reset),
83          .i_rx_done(rx_done), .i_rx_data_in(rx_data_out),
84          .i_alu_data_in(o_alu),
85          .o_dato_A(dato_a), .o_dato_B(dato_b), .o_op(operacion),
86          .o_tx_start(tx_start), .o_data_out(tx_data_in)
87        );
88
89       //ALU
90       alu
91       #(
92          .N_BITS(N_BITS)
93        )
94        u_alu
95        (
96          .i_dato_A(dato_a), .i_dato_B(dato_b),
```

```
97          .i_operacion(operacion),
98          .o_alu(o_alu)
99      );
100
101 endmodule
```

El código del *testbench* se muestra a continuación y el resultado de la simulación con tiempo en la Figura **13**.

Código 13: *Testbench* de '*top_all*'.

```
1  `timescale 1ns / 1ps
2
3  module tb_top_all();
4
5      localparam N_BITS   = 8;
6      localparam OP_BITS  = 6;
7      localparam F_CLOCK  = 50E6;
8      localparam BAUDRATE = 9600;
9      localparam SAMPLING = 16;
10     localparam CNT_BITS = $clog2(N_BITS);
11
12     reg                clk, reset;
13     reg  [N_BITS-1:0]  data;
14     reg                rx;
15     wire               tx_done;
16     wire               tx;
17
18     reg [CNT_BITS:0] ii;
19
20     /*
21      * cada 326 ciclos de reloj se produce 1 tick
22      * hay que contar 16 ticks, entonces
23      * 326*16 = 5216
24      *
25      * 1 ciclo            -> 20ns
26      * 326 ciclos (1tick) -> 104320ns
27      */
28     initial begin
29         clk   = 1'b0;
30         rx    = 1'b1;
31         reset = 1'b1;
32
33         #40
34         reset = 1'b0;
35
36         //Mando dato A
37         data = 8'b00000001;//{$urandom()};
38
39         #104320
40         rx = 1'b0; //bit de inicio
41
42         //comienzo a enviar el dato random de N_BITS bits
43         //de LSB a MSB
44         for(ii = 0; ii < N_BITS; ii = ii + 1)
45             #104320 rx = data[ii];
46
47         #104320
48         rx = 1'b1; //bit de stop
```

```
49
50          //Mando dato B
51          #20
52          data = 8'b00000010;//{$urandom()};
53
54          #104320
55          rx = 1'b0; //bit de inicio
56
57          //comienzo a enviar el dato random de N_BITS bits
58          //de LSB a MSB
59          for(ii = 0; ii < N_BITS; ii = ii + 1)
60              #104320 rx = data[ii];
61
62          #104320
63          rx = 1'b1; //bit de stop
64
65          //Mando operacion (SUMA)
66          data = 8'b00100000;//{$urandom()};
67
68          #104320
69          rx = 1'b0; //bit de inicio
70
71          //comienzo a enviar el dato random de N_BITS bits
72          //de LSB a MSB
73          for(ii = 0; ii < N_BITS; ii = ii + 1)
74              #104320 rx = data[ii];
75
76          #104320
77          rx = 1'b1; //bit de stop
78      end
79
80      always #10 clk = ~clk;
81
82      always @(*)
83      begin
84          if(tx_done == 1)
85          begin
86              $display("------ TEST OK! ------");
87              $finish;
88          end
89      end
90
91      top_all
92      #(
93          .N_BITS(N_BITS), .OP_BITS(OP_BITS), .F_CLOCK(F_CLOCK),
94          .BAUDRATE(BAUDRATE), .SAMPLING(SAMPLING)
95       )
96      u_top_all
97      (
98          .i_clk(clk), .i_reset(reset),
99          .i_rx_top(rx),
100         .o_tx_top(tx),
101         .o_tx_done(tx_done)
102     );
103
104 endmodule
```

Figura 13: Simulación con tiempo del sistema completo.

3. Cálculo de frecuencia máxima

Para obtener el valor de la máxima frecuencia a la que puede trabajar el circuito se utilizó el reporte de tiempo que provee Vivado. Para obtener este reporte se deben hacer los siguientes pasos:

1. Dirigirse a la opción '*Edit timing constraints*';
2. Seleccionar '*Create clock*' y crear uno nuevo;
3. Agregar como '*source object*' la entrada del *clock* del diseño realizado;
4. Definir la frecuencia del *clock*.

Una vez hecho esto, el reporte de tiempo realizado por Vivado muestra que el tiempo de *setup* es de `13.038 [ns]` y el tiempo de *hold* es de `0.165 [ns]`.

Sumando estos tiempos, se obtiene el tiempo del *clock*:

$$\begin{aligned} t_{clock} &= t_{setup} + t_{hold} \\ t_{clock} &= 13{,}038 + 0{,}165 \\ t_{clock} &= 13{,}203\,[ns] \end{aligned}$$

Finalmente, el periodo del *clock* es:

$$T_{clock} = 75{,}7\,[MHz] \tag{2}$$

4. Conclusiones

Se logró implementar un diseño modularizado del modelo propuesto, que permitió encontrar los errores con más facilidad cuando las simulaciones no mostraban los resultados esperados.

Se incorporó la utilización de herramientas de análisis de tiempo provistas por Vivado, lo que facilitó el cálculo de la frecuencia máxima de operación del circuito.

Referencias

[1] Pong P. Chu, *FPGA Prototyping by Verilog Examples*. Cleveland State University, 2008