Arquitectura de Computadoras

Trabajo Práctico 2

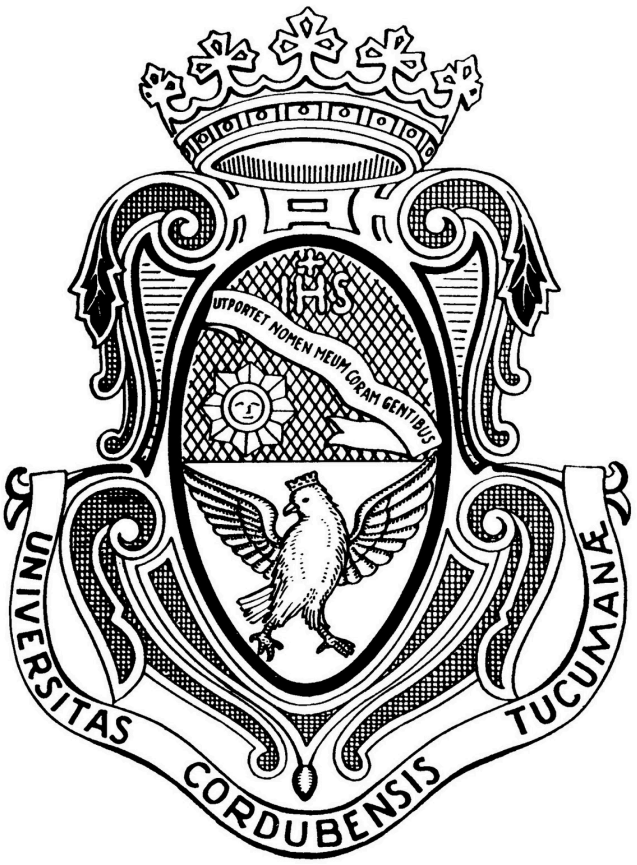
Máquinas de Estado Finitas
TP: UART

Autores

BORGATELLO, Ignacio

DALLARI LARROSA, Gian Franco

2025



1. Introducción

El trabajo práctico trata sobre la implementación de un UART (Universal Asynchronous Receiver and Transmitter) utilizando Máquinas de Estado Finitas (FSM) en Verilog. Se debe diseñar un sistema que incluya:

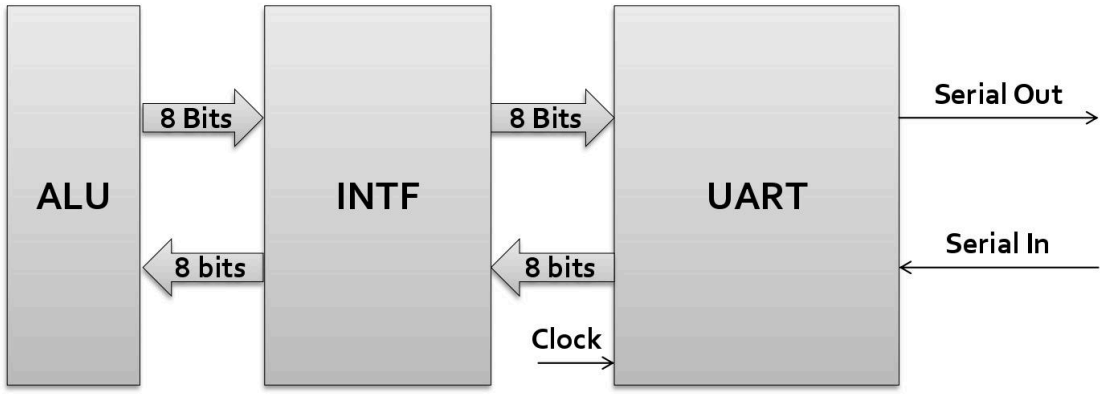
1. Generador de Baud Rate, que controla la velocidad de transmisión de datos.
2. Receptor UART (Rx), que sigue una secuencia de estados para recibir datos en serie, sincronizándose con los bits de inicio, datos y parada.
3. Transmisor UART (Tx), que envía datos en serie con el formato adecuado.
4. Interfaz con una ALU, para procesar los datos recibidos.

Secuencia de estados Rx

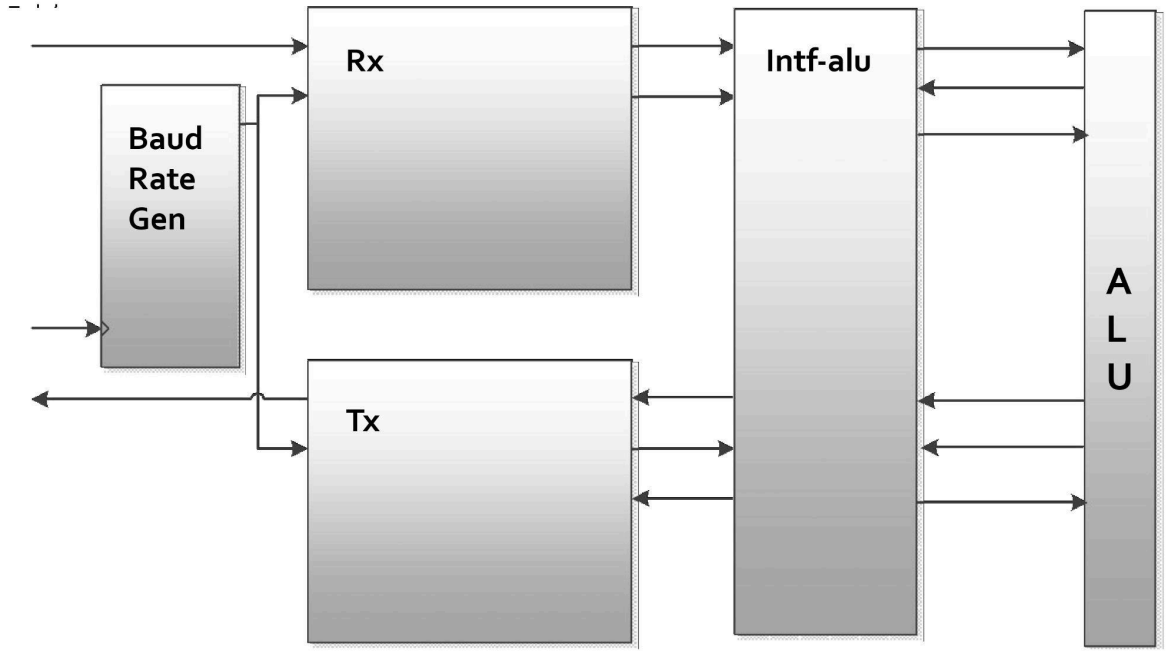
- Asumiendo N bits de datos, M bits de Stop.
  - 1) Esperar a que la señal de entrada sea 0, momento en el que inicia el bit de Start. Iniciar el Tick Counter.
  - 2) Cuando el contador llega a 7, la señal de entrada está en el punto medio del bit de Start. Reinicar el contador.
  - 3) Cuando el contador llega a 15, la señal de entrada avanza 1 bit, y alcanza la mitad del primer bit de datos. Tomar este valor e ingresarlo en un shift register. Reinicar el contador.
  - 4) Repetir el paso 3 N-1 veces para tomar los bits restantes.
  - 5) Si se usa bit de paridad, repetir el paso 3 una vez mas.
  - 6) Reperir el paso 3 M veces, para obtener los bits de Stop.

TP: UART

- Universal Asynchronous Receiver and Transmitter



TP: UART



2. Desarrollo

a. Introducción

En este informe, se describe el desarrollo e implementación de módulos en Verilog para la comunicación UART, siguiendo un enfoque incremental. Se comenzó con la generación del Baud Rate, seguido de la implementación del receptor UART (UART RX) y el transmisor UART (UART TX), ambos basados en máquinas de estados finitos (FSM). Posteriormente, se desarrolló una FIFO como interfaz de almacenamiento y, finalmente, un módulo top encargado de interpretar y definir los operandos de una ALU.

b. Módulo Baud Rate

El primer módulo desarrollado fue el generador de Baud Rate, esencial para sincronizar la comunicación UART. Se implementó un divisor de frecuencia basado en un contador para ajustar la tasa de transmisión a un valor definido, por ejemplo, 19200 baudios.

```
`timescale 1ns / 1ps

module mod_m_counter
   #(
    parameter N = 4,
    parameter M = 10
   )
   (
    input wire clk,
    input wire reset,
    output wire max_tick,
    output wire [N-1:0] q
   );

   reg [N-1:0] r_reg;
   wire [N-1:0] r_next;

   always @(posedge clk)
      if (reset)
         r_reg <= 0;
      else
         r_reg <= r_next;

   assign r_next = (r_reg == (M-1)) ? 0 : r_reg + 1;
   assign max_tick = (r_reg == (M-1)) ? 1'b1 : 1'b0;
   assign q = r_reg;
endmodule
```

Teniendo en cuenta que

$$\frac{Clock}{BaudRate * 16} = Ticks$$

Al trabajar con la frecuencia de reloj predeterminada de 100 MHz, especificada en la restricción (constraint) de la placa, a continuación:

```
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports clk]
```

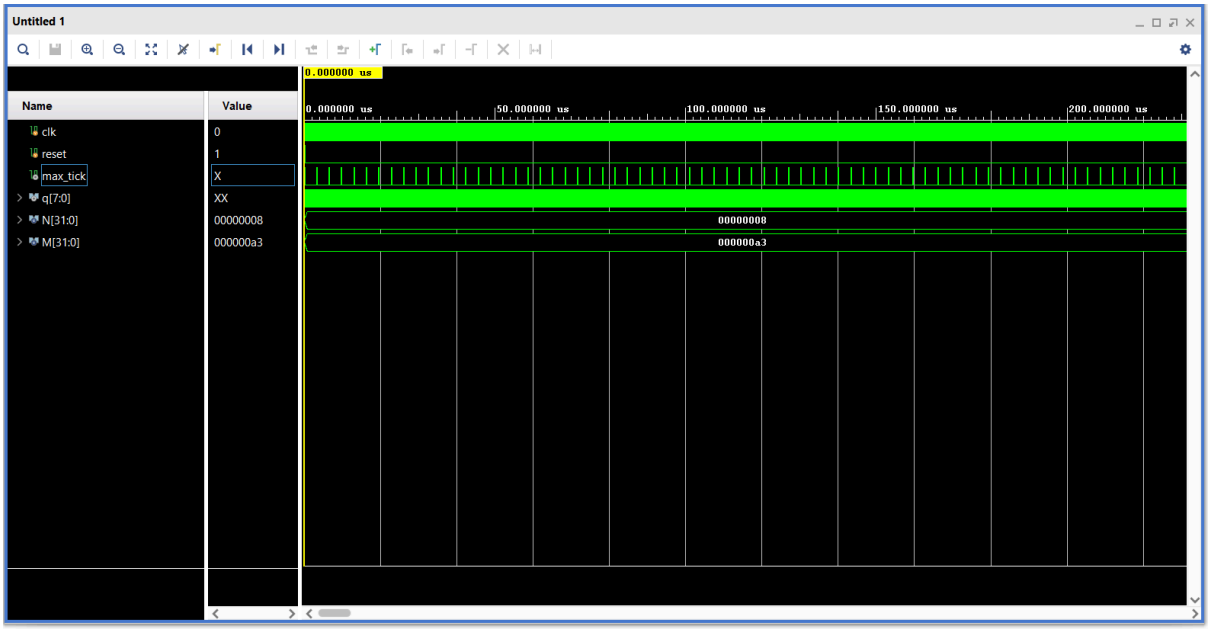
entonces, la cantidad de ticks que debemos definir en el mod_m_counter es la siguiente:
}

$$\frac{100\ MHz}{19200 * 16} = 325$$

Se diseñó un testbench para verificar el funcionamiento del módulo baud_rate. Se genera un reloj con un período de 10 ns (100 MHz) y se aplica un reinicio inicial. Luego, el testbench observa la evolución de la salida del contador y la señal max_tick.

```
module baud_rate_tb;

parameter N = 8;
parameter M = 163;

reg clk;
reg reset;
wire max_tick;
wire [N-1:0] q;

mod_m_counter #(.N(N), .M(M)) dut (
    .clk(clk),
    .reset(reset),
    .max_tick(max_tick),
    .q(q)
);

always #10 clk = ~clk;

initial begin
    clk = 0;
    reset = 1;
    #20 reset = 0;
    #200;
    $stop;
end

initial begin
    $monitor("Time=%0t | q=%d | max_tick=%b", $time, q, max_tick);
```

```
end
endmodule
```



c. Módulo UART Rx

El siguiente paso fue la implementación del receptor UART. Este módulo detecta el bit de inicio, realiza la muestra de los bits de datos y verifica la integridad de la información mediante un bit de paridad. Se utilizó una máquina de estados finitos (FSM) para gestionar la captura de bits y ensamblar los datos recibidos en un registro.

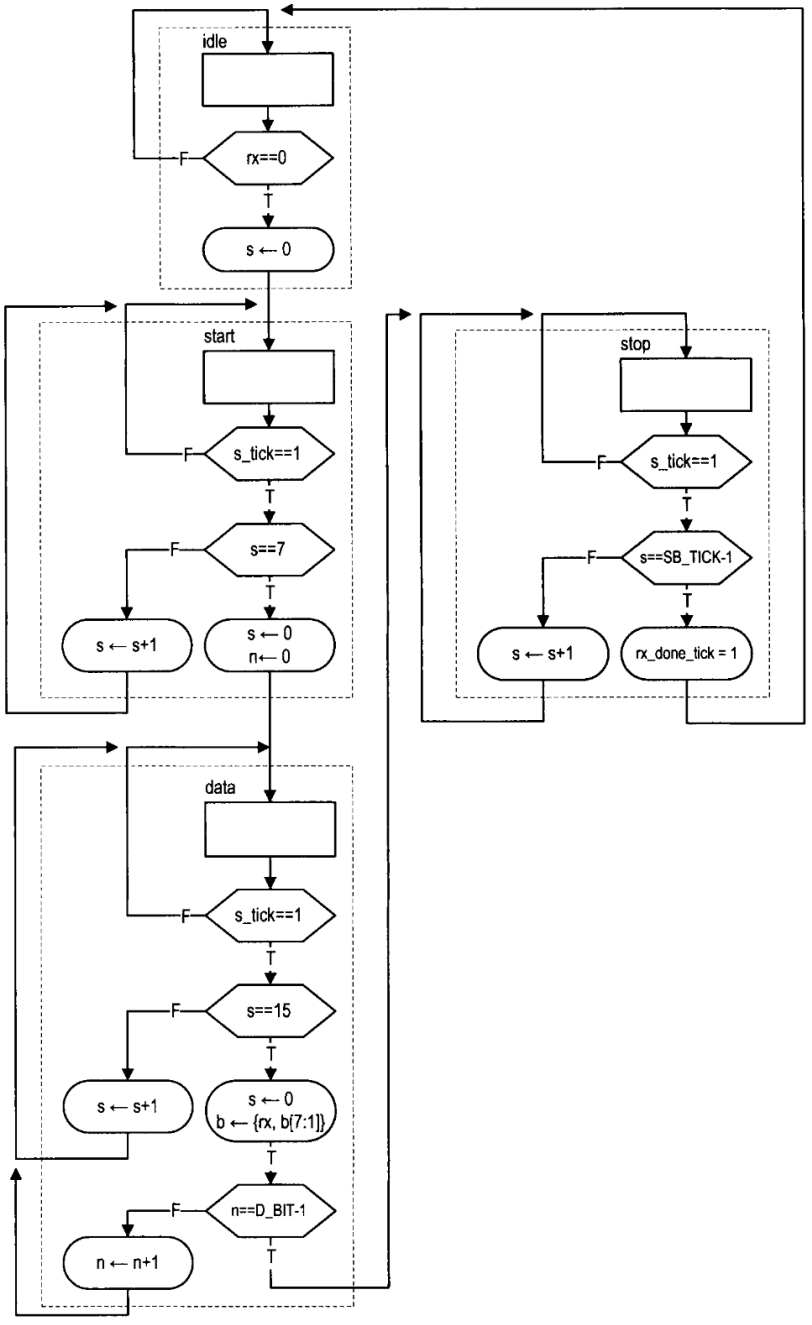


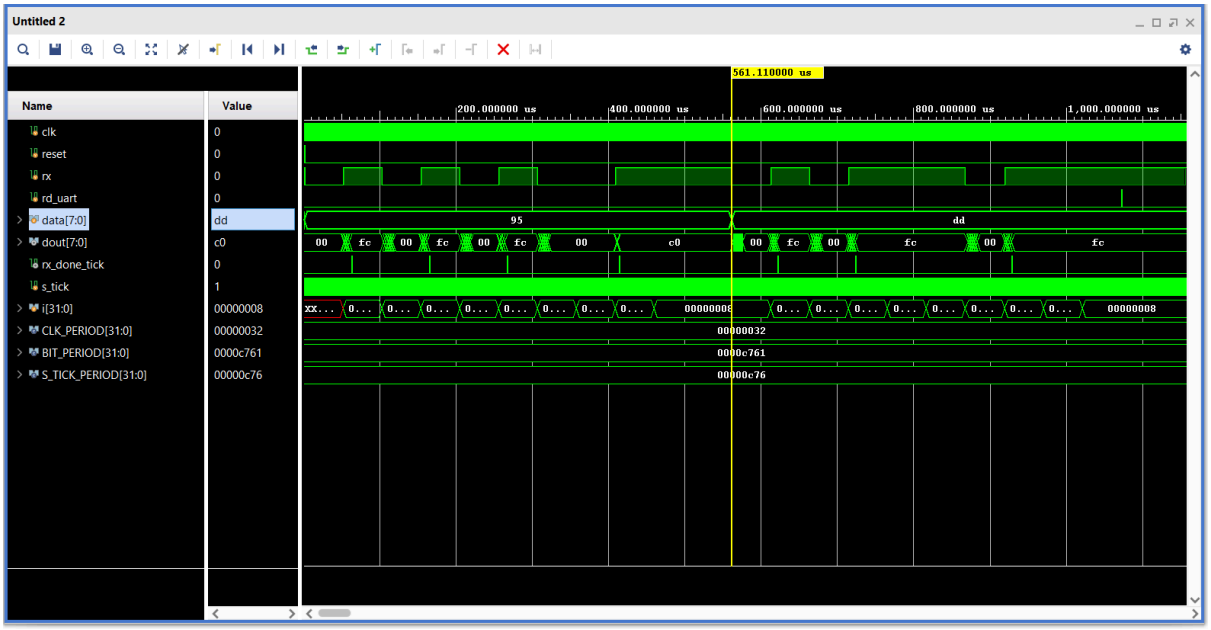
Figure 8.3 ASMD chart of a UART receiver.

```
// FSMD next-state logic
always @*
```

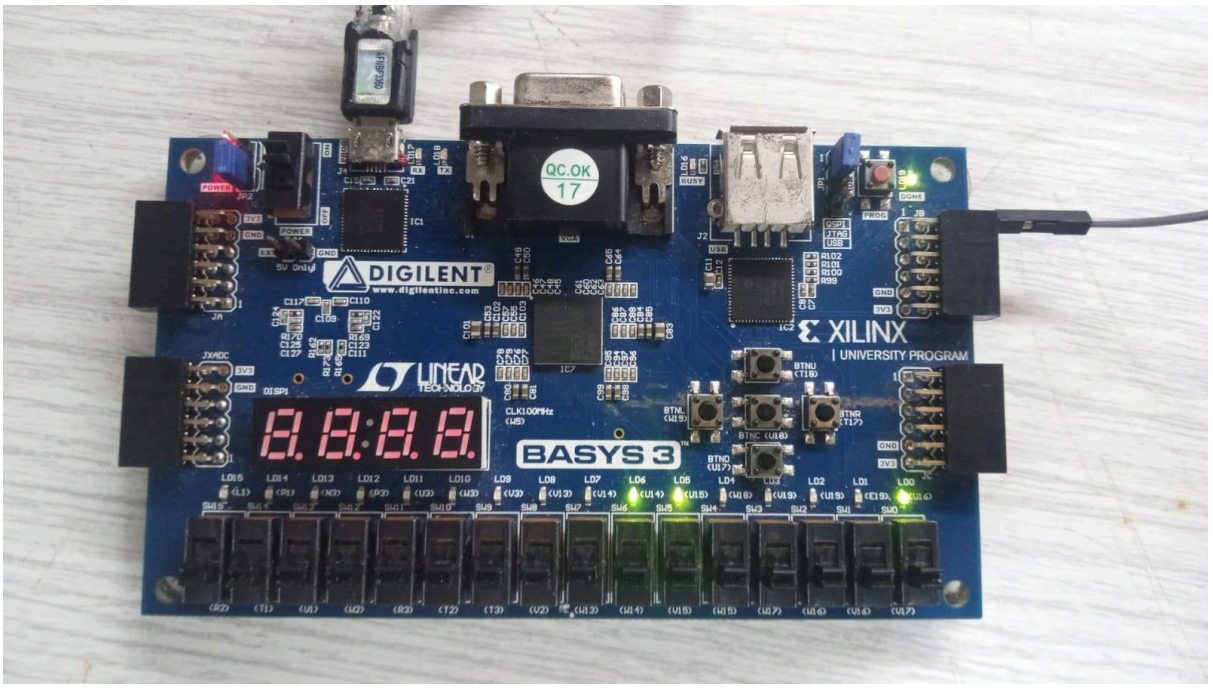
```
begin
   state_next = state_reg;
   rx_done_tick = 1'b0;
   s_next = s_reg;
   n_next = n_reg;
   b_next = b_reg;
   case (state_reg)
      idle:
         if (~rx)
            begin
               state_next = start;
               s_next = 0;
            end
      start:
         if (s_tick)
            if (s_reg==7)
            begin
                state_next = ~rx ? data : idle;
                s_next = 0;
                n_next = 0;
            end
            else
               s_next = s_reg + 1;
      data:
         if (s_tick)
            if (s_reg==(SB_TICK-1))
               begin
                  s_next = 0;
                  b_next = {rx, b_reg[7:1]};
                  if (n_reg==(DBIT-1))
                     state_next = stop ;
                   else
                     n_next = n_reg + 1;
                end
            else
               s_next = s_reg + 1;
      stop:
         if (s_tick)
            if (s_reg==(SB_TICK-1))
               begin
                  state_next = idle;
                  if(rx)
                     rx_done_tick =1'b1;
               end
            else
               s_next = s_reg + 1;
   endcase
end
```

El testbench verifica el funcionamiento de un receptor UART (UART RX) simulando la recepción de bytes de datos en la línea rx. Se genera una secuencia de bits con los tiempos adecuados para imitar una transmisión UART real, incluyendo bits de inicio, datos y parada. Además, se controla la señal rd uart para leer los datos recibidos.

```
initial begin
    clk = 0;
    reset = 1;
    rd_uart = 0;
    rx = 1;
    #20 reset = 0;

    // Primer byte
    #500;
    data = 8'b10010101;
    rx = 0; // start bit
    #51041;
    for (i = 0; i < 8; i = i + 1) begin
        rx = data[i];
        #51041;
    end
    rx = 1; // stop bit
    #51041;

    // Segundo byte
    #50000;
    data = 8'b10111101;
    rx = 0; // start bit
    #51041;
    for (i = 0; i < 8; i = i + 1) begin
        rx = data[i];
        #51041;
    end
    rx = 1; // stop bit
    #51041;

    // Lectura UART simulada
    #500;
    rd_uart = 1;
    #500;
    rd_uart = 0;

    #500;
    $finish;
end
```



Se realizó la implementación en la placa del receptor UART (UART RX), lo que nos permitió recibir datos desde la computadora. Al enviar el carácter "a" (código ASCII 0x61 o 8'b01100001) desde el terminal el módulo debe detectar correctamente el bit de inicio, leer los bits de datos y reconocer el bit de parada. A continuación, se muestra la placa con los leds encendidos.



d. FIFO

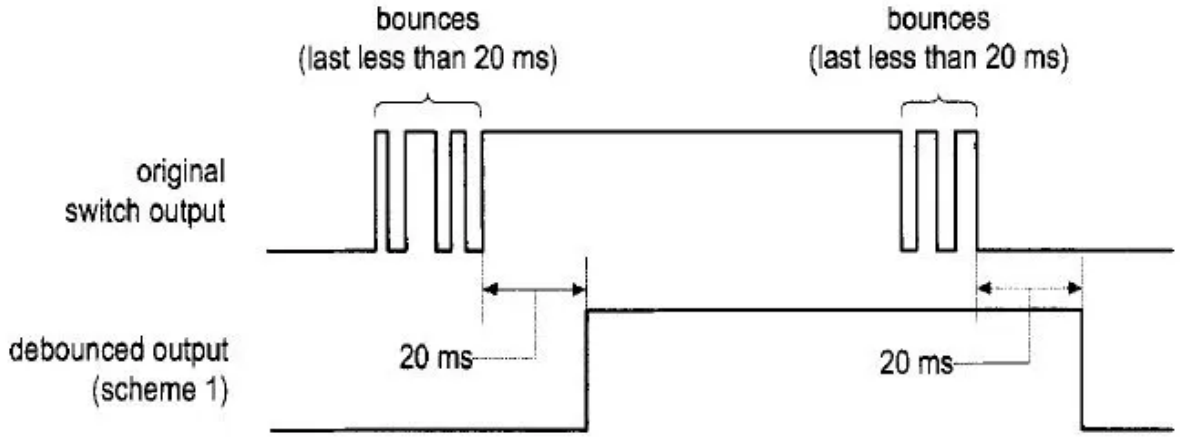
Para gestionar la comunicación y evitar pérdida de datos, se implementó una FIFO (First In, First Out). Esta memoria intermedia permite almacenar temporalmente los datos recibidos por el UART RX antes de ser procesados por otros módulos del sistema. Se utilizó una estructura de memoria con punteros de lectura y escritura, asegurando un acceso ordenado a los datos.

```
// next-state logic for read and write pointers
always @*
begin
   // successive pointer values
   w_ptr_succ = w_ptr_reg + 1;
   r_ptr_succ = r_ptr_reg + 1;
   // default: keep old values
   w_ptr_next = w_ptr_reg;
   r_ptr_next = r_ptr_reg;
   full_next = full_reg;
   empty_next = empty_reg;
   case ({wr, rd})
      // 2'b00:  no op
      2'b01: // read
         if (~empty_reg) // not empty
            begin
               r_ptr_next = r_ptr_succ;
               full_next = 1'b0;
               if (r_ptr_succ==w_ptr_reg)
                  empty_next = 1'b1;
            end
      2'b10: // write
         if (~full_reg) // not full
            begin
               w_ptr_next = w_ptr_succ;
               empty_next = 1'b0;
               if (w_ptr_succ==r_ptr_reg)
                  full_next = 1'b1;
            end
      2'b11: // write and read
         begin
            w_ptr_next = w_ptr_succ;
            r_ptr_next = r_ptr_succ;
         end
   endcase
```

e. Módulo Antirrebotes

El módulo antirrebotes (debounce) es una técnica utilizada para estabilizar la señal de un interruptor mecánico al eliminar los rebotes eléctricos que ocurren al presionarlo o soltarlo. Cuando un interruptor cambia de estado, sus contactos metálicos no realizan la transición de manera instantánea, sino que vibran durante unos milisegundos, generando señales erráticas. Sin un sistema de control, un microcontrolador podría interpretar múltiples pulsos en lugar de un único evento.

Para evitar esto, el módulo espera un número determinado de ciclos de reloj antes de validar el cambio de estado del pulsador. Una vez confirmado, mantiene el valor estable durante un ciclo de reloj y luego lo restablece a 0, asegurando una detección precisa y evitando falsas activaciones.

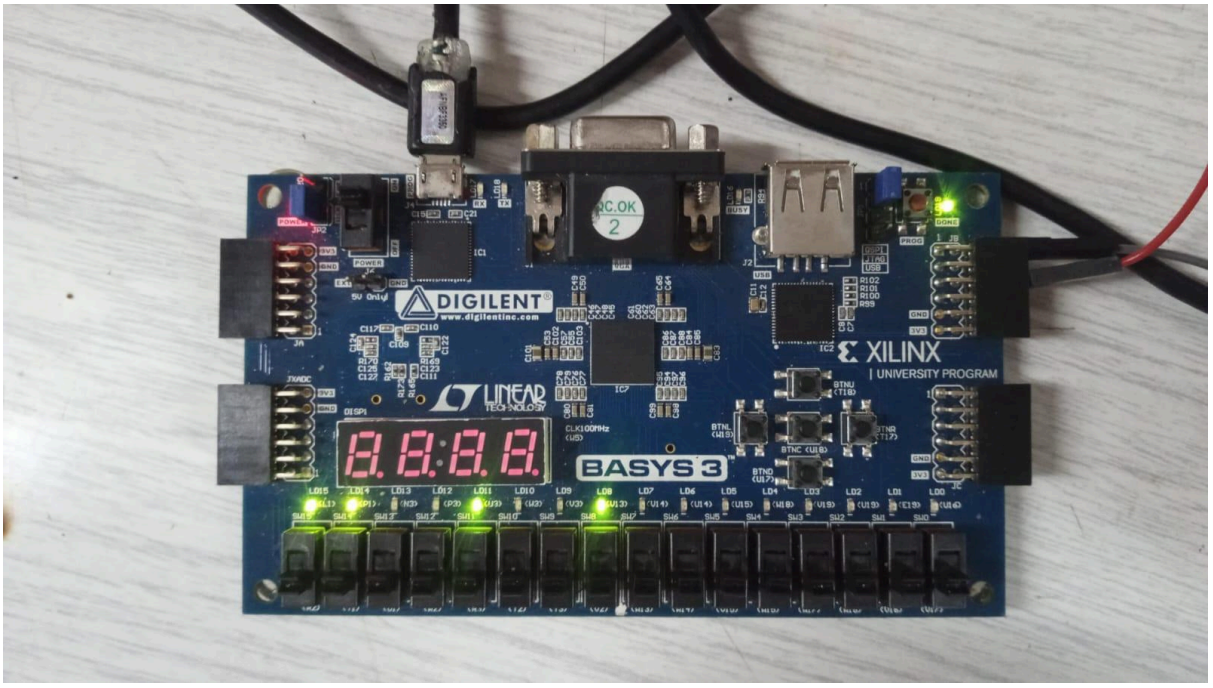


f. ALU

La ALU utilizada en este proyecto es la misma que se implementó en el trabajo anterior. Sin embargo, se realizaron ajustes en los parámetros de operación (códigos de función) para simplificar la implementación y la decodificación en la etapa actual.

```
// Operaciones
localparam ADD  = 8'b00100000;
localparam SUB  = 8'b00100010;
localparam AND  = 8'b00100100;
localparam OR   = 8'b00100101;
localparam XOR  = 8'b00100110;
localparam SRA  = 8'b00000011;
localparam SRL  = 8'b00000010;
localparam NOR  = 8'b00100111;
```

Los datos transmitidos son: 'e' (01100100), 'd' (01100101) y 'Espacio' (00100000). La operación realizada es una suma (ADD) entre los operandos, y el resultado esperado es 11001001. Esto representa la combinación de los valores en una operación aritmética, asegurando que el procesamiento de los datos se realice correctamente.



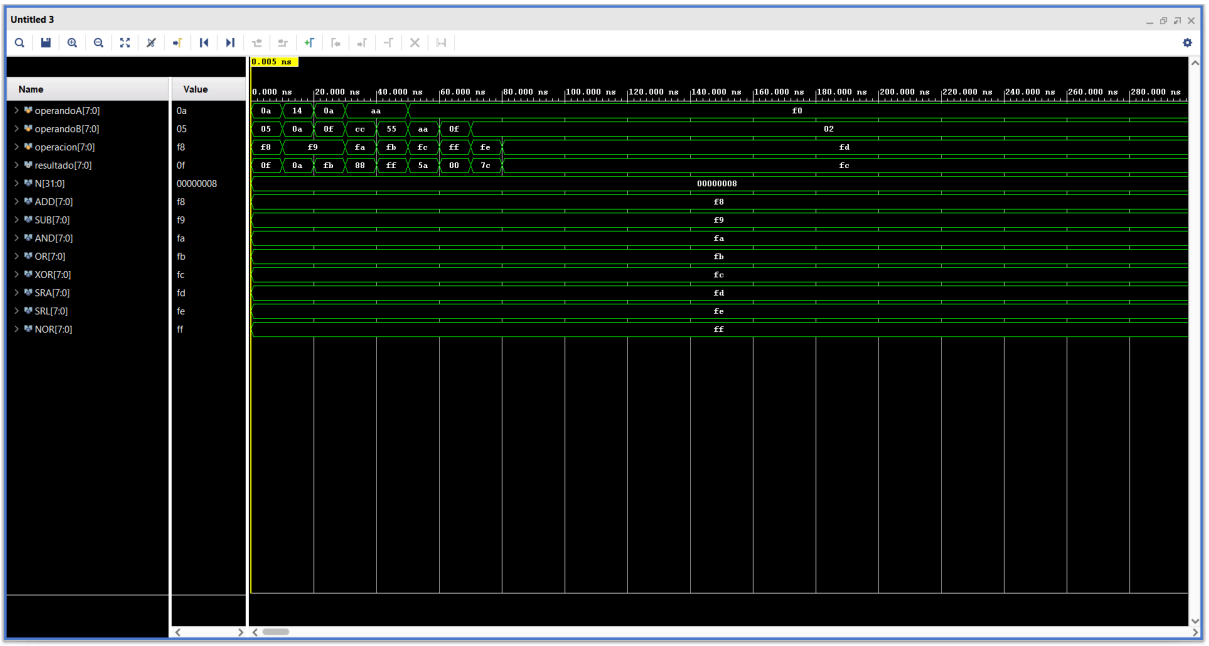
Los LEDs más significativos [8:15] indican el dato recibido a través de UART, mientras que los LEDs [0:7] muestran el resultado de la operación realizada sobre dicho dato. Esto permite una visualización clara tanto de la entrada como del procesamiento, facilitando la verificación del funcionamiento correcto del sistema.

Corrección del Práctico 1

En la versión anterior, los operandos operandoA y operandoB eran tratados como valores sin signo (unsigned). Esto provocaba un comportamiento incorrecto en operaciones aritméticas que generaban resultados negativos.

Se aplicó el operador **`$signed()`** a los operandos antes de realizar las operaciones aritméticas, lo que indica al simulador/sintetizador que los valores deben interpretarse según el formato de complemento a dos.

```
input  wire signed [tamanioEntrada-1:0] operandoA,
input  wire signed [tamanioEntrada-1:0] operandoB,
```



g. Módulo UART Tx

Además del receptor, se desarrolló el módulo de transmisión UART (UART TX), encargado de enviar los datos de manera secuencial siguiendo el protocolo UART. Al igual que el receptor, se utilizó una máquina de estados finitos (FSM) para gestionar el envío de bits, asegurando una correcta temporización y transmisión de los datos.

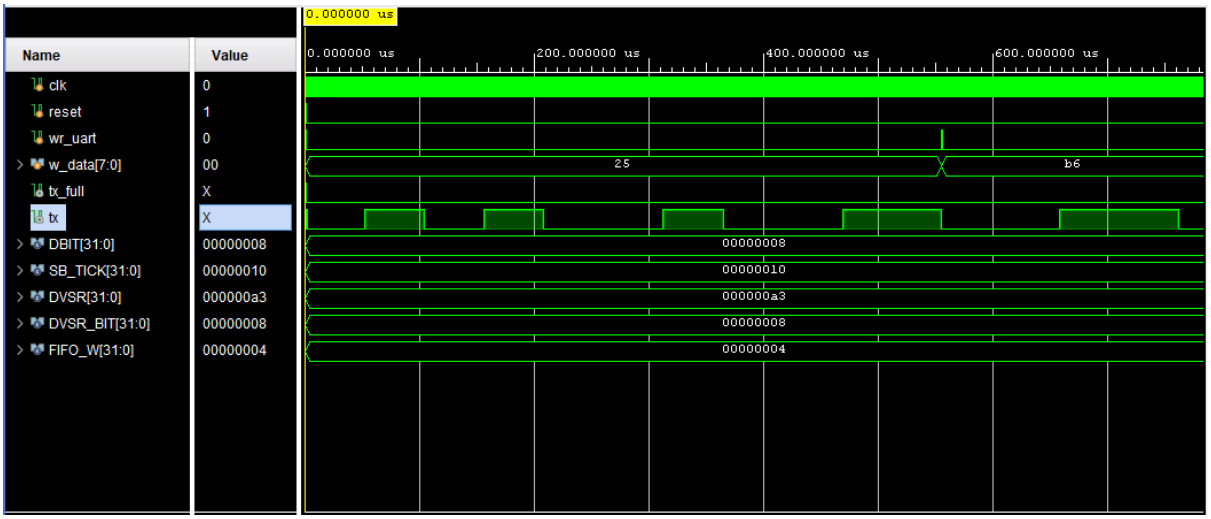
```
// FSMD next-state logic & functional units

   always @*
   begin
      state_next = state_reg;
      tx_done_tick = 1'b0;
      s_next = s_reg;
      n_next = n_reg;
      b_next = b_reg;
      tx_next = tx_reg ;
      case (state_reg)
         idle:
            begin
               tx_next = 1'b1;
               if (tx_start)
                  begin
                     state_next = start;
                     s_next = 0;
                     b_next = din;
                  end
            end
         start:
            begin
```

```
            tx_next = 1'b0;
            if (s_tick)
               if (s_reg==(SB_TICK-1))
                  begin
                     state_next = data;
                     s_next = 0;
                     n_next = 0;
                  end
               else
                  s_next = s_reg + 1;
         end
      data:
         begin
            tx_next = b_reg[0];
            if (s_tick)
               if (s_reg==(SB_TICK-1))
                  begin
                     s_next = 0;
                     b_next = b_reg >> 1;
                     if (n_reg==(DBIT-1))
                        state_next = stop ;
                     else
                        n_next = n_reg + 1;
                  end
               else
                  s_next = s_reg + 1;
         end
      stop:
         begin
            tx_next = 1'b1;
            if (s_tick)
               if (s_reg==(SB_TICK-1))
                  begin
                     state_next = idle;
                     tx_done_tick = 1'b1;
                  end
               else
                  s_next = s_reg + 1;
         end
   endcase
end
```

Se generan señales de reloj y reset, y se proporcionan datos de entrada al transmisor para comprobar que los bits se transmitan correctamente. Se incluyen pruebas para enviar distintos datos y verificar que el transmisor responda según lo esperado, emitiendo los bits en el orden adecuado y con los tiempos de espera correctos entre ellos.

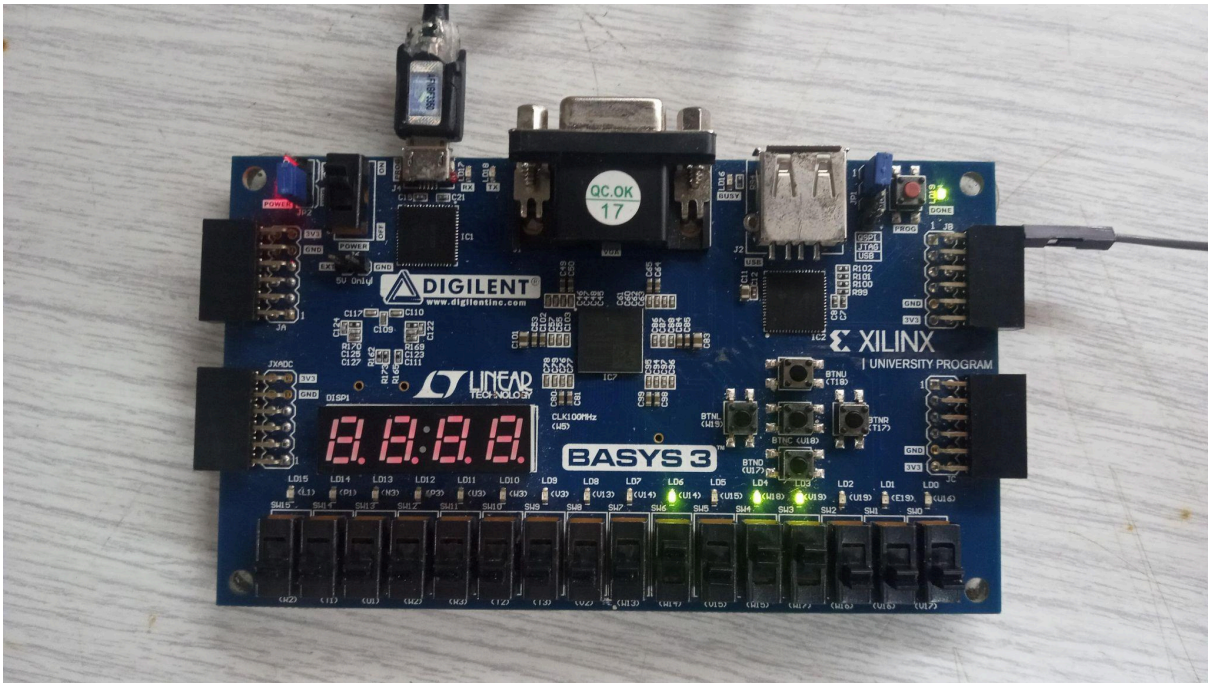
```
initial begin
```

```
    clk = 0;
    reset = 1;
    wr_uart = 0;
    w_data = 8'h00;

    #20 reset = 0;

    // Primer byte
    #20 w_data = 8'b01001011;
    wr_uart = 1;
    #20;
    wr_uart = 0;
    #555400;

    // Segundo byte
    #20 w_data = 8'b10110110;
    wr_uart = 1;
    #20;
    wr_uart = 0;
    #555400;

    $stop;
end
```



Se intenta transmitir el símbolo X = 01011000 utilizando un pulsador. Sin el mecanismo de antirrebote, los rebotes del pulsador generan fluctuaciones en la señal TX, resultando en una transmisión errónea y datos corruptos en la PC.

Con el mecanismo de antirrebote implementado, la señal se estabiliza, eliminando los rebotes y permitiendo una transmisión precisa y confiable. Así, el símbolo X = 01011000 se recibe correctamente en la PC, asegurando una comunicación exitosa y sin errores.



3. Implementación

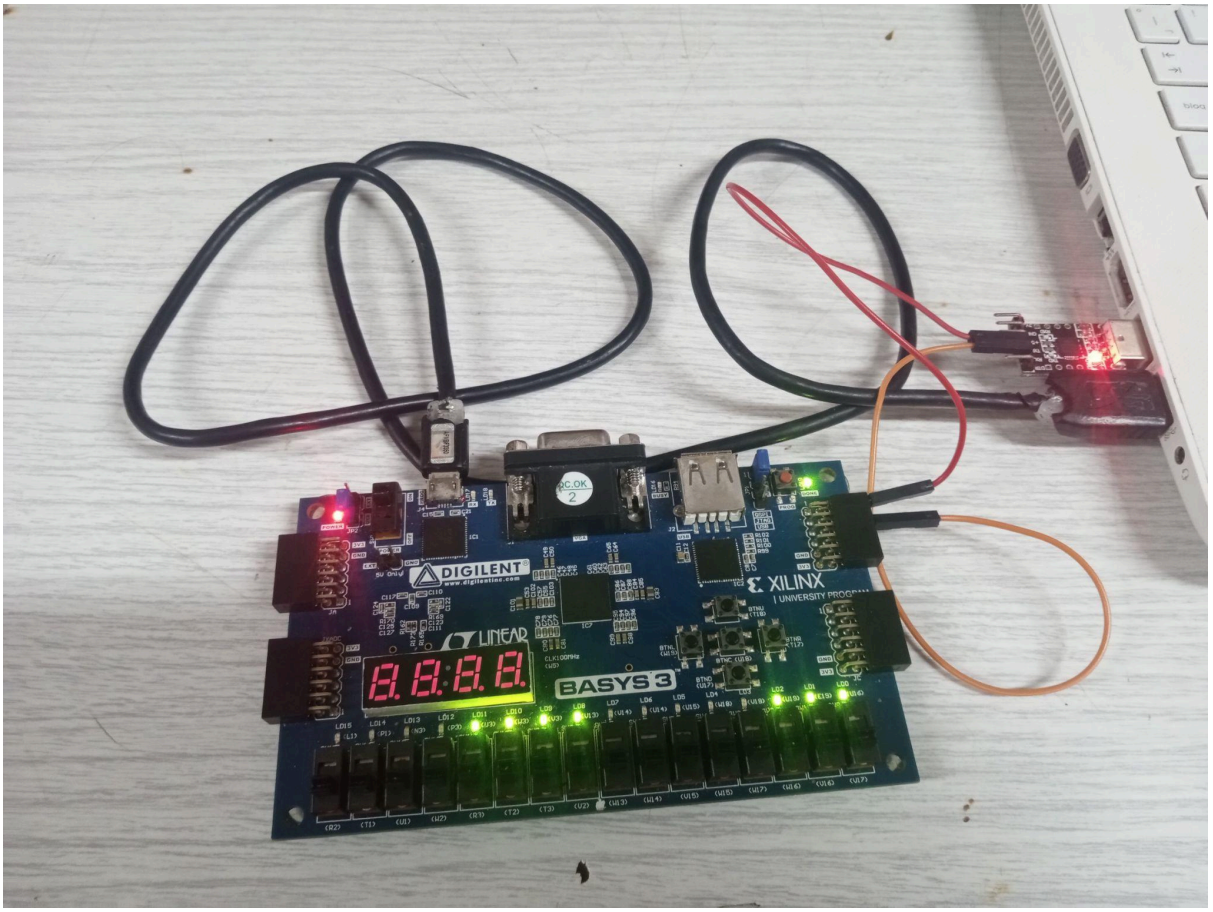
Cada módulo fue desarrollado de forma independiente, lo que permitió una construcción flexible y modular del sistema. Posteriormente, se integraron todos los componentes en un diseño completo y estructurado. En este sistema, los datos enviados desde la PC se registran en los operandos de la ALU, cuya operación genera un resultado que se visualiza en los LED de la placa. Además, el resultado puede ser opcionalmente transmitido nuevamente a la PC a través del transmisor, permitiendo su visualización en el equipo.

```
module top
   #( // Default setting:
      // 19200 baud, 8 data bits, 1 stop bit, 2^2 FIFO
      parameter DBIT = 8,
                SB_TICK = 16,
                DVSR = 325,    // baud rate divisor
                               // DVSR = 100MHz/(16*baudios)
                DVSR_BIT = 10,// # bits of DVSR
                FIFO_W = 4     // # addr bits of FIFO
                               // # words in FIFO=2^FIFO_W
   )
   (
    input wire clk, reset,
    input wire rd_uart, rx,
    input wire wr_uart,
    output wire [7:0] r_data,
    output wire [7:0] alu_output,
    output wire tx_full, tx,
    output wire rx_empty
```

```
);

// Señales internas
wire tick, rx_done_tick;
wire [7:0] rx_data_out;
wire rd_uart_tick, wr_tick;
wire [7:0] w_data;

wire tx_done_tick;
wire tx_empty, tx_fifo_not_empty;
wire [7:0] tx_fifo_out;

reg [7:0] operandoA = 0;
reg [7:0] operandoB = 0;
reg [7:0] codigoOperacion = 0;
reg [7:0] prev_opcode = 0; // Estado previo del opcode

// Registro para salida de la ALU (asegura limpieza en reset)
reg [7:0] alu_output_reg;

// Registro opcional para r_data (asegura limpieza en reset)
reg [7:0] r_data_reg;

// Debounce de botones
debounce btn1_db_unit
 (.clk(clk), .reset(reset), .sw(rd_uart),
  .db_level(), .db_tick(rd_uart_tick));

debounce btn2_db_unit
 (.clk(clk), .reset(reset), .sw(wr_uart),
  .db_level(), .db_tick(wr_tick));

// Generador de baud
mod_m_counter #(.M(DVSR), .N(DVSR_BIT)) baud_gen_unit
   (.clk(clk), .reset(reset), .q(), .max_tick(tick));

// UART RX
uart_rx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_rx_unit
   (.clk(clk), .reset(reset), .rx(rx), .s_tick(tick),
    .rx_done_tick(rx_done_tick), .dout(rx_data_out));

// FIFO RX
fifo #(.B(DBIT), .W(FIFO_W)) fifo_rx_unit
   (.clk(clk), .reset(reset), .rd(rd_uart_tick),
    .wr(rx_done_tick), .w_data(rx_data_out),
    .empty(rx_empty), .full(), .r_data(r_data_reg));

// OpCodes (últimos 3 valores de 8 bits)
```

```
   localparam ALU_DATA_A_OP   = 8'd253; // 11111101
   localparam ALU_DATA_B_OP   = 8'd254; // 11111110
   localparam ALU_OPERATOR_OP = 8'd255; // 11111111

   // Lógica de asignación de operandos
   always @(posedge clk or posedge reset) begin
       if (reset) begin
           operandoA       <= 0;
           operandoB       <= 0;
           codigoOperacion <= 0;
           prev_opcode     <= 0;
           alu_output_reg  <= 0;  // limpia la salida de la ALU
           r_data_reg      <= 0;  // limpia r_data
       end
       else if (!rx_empty && rd_uart_tick) begin
           if (r_data_reg == ALU_DATA_A_OP ||
               r_data_reg == ALU_DATA_B_OP ||
               r_data_reg == ALU_OPERATOR_OP) begin
               prev_opcode <= r_data_reg;
           end
           else begin
               case (prev_opcode)
                   ALU_DATA_A_OP:    operandoA       <=
r_data_reg;
                   ALU_DATA_B_OP:    operandoB       <=
r_data_reg;
                   ALU_OPERATOR_OP:  codigoOperacion <=
r_data_reg;
               endcase
           end
       end

       // Registrar salida de la ALU
       alu_output_reg <= alu_output_reg; // por default mantiene
valor
   end

   // Instancia ALU
   alu alu_inst (
       .operandoA(operandoA),
       .operandoB(operandoB),
       .operacion(codigoOperacion),
       .resultado(alu_output_reg)
   );

   // FIFO TX
   fifo #(.B(DBIT), .W(FIFO_W)) fifo_tx_unit
      (.clk(clk), .reset(reset), .rd(tx_done_tick),
```

```
        .wr(wr_tick), .w_data(w_data), .empty(tx_empty),
        .full(tx_full), .r_data(tx_fifo_out));

    // UART TX
    uart_tx #(.DBIT(DBIT), .SB_TICK(SB_TICK)) uart_tx_unit
       (.clk(clk), .reset(reset), .tx_start(tx_fifo_not_empty),
        .s_tick(tick), .din(tx_fifo_out),
        .tx_done_tick(tx_done_tick), .tx(tx));

    // Conexiones
    assign w_data = alu_output_reg;
    assign tx_fifo_not_empty = ~tx_empty;
    assign r_data = r_data_reg;
    assign alu_output = alu_output_reg;

endmodule
```



4. Repositorio de GitHub

https://github.com/nachoborgatello/tp2_uart

5. Referencias

1. Chu PP (2008) FPGA prototyping by VHDL Examples: Xilinx Spartan-3 version. Wiley, New York