Arquitectura de Computadoras

Trabajo Práctico 2

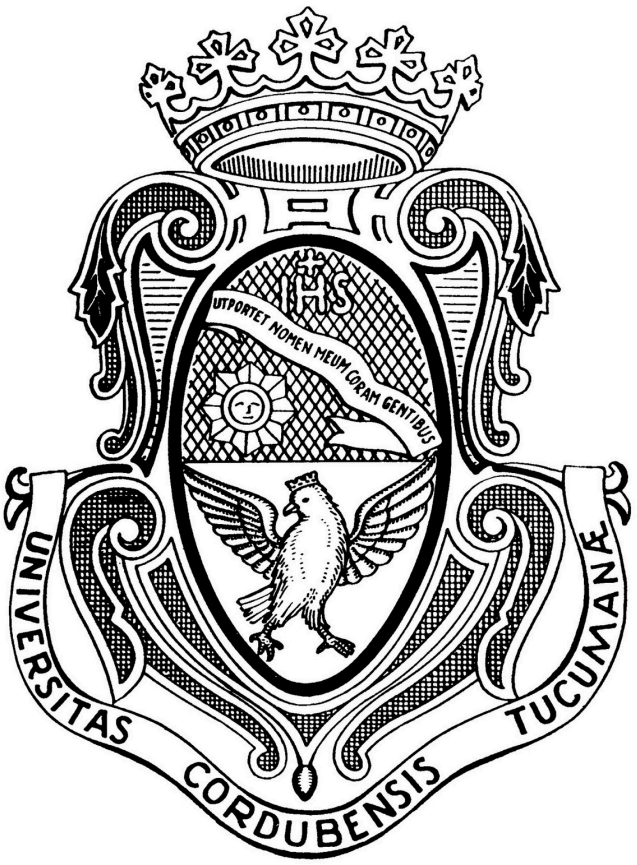
Máquinas de Estado Finitas
TP: UART

Autores

BORGATELLO, Ignacio

DALLARI LARROSA, Gian Franco

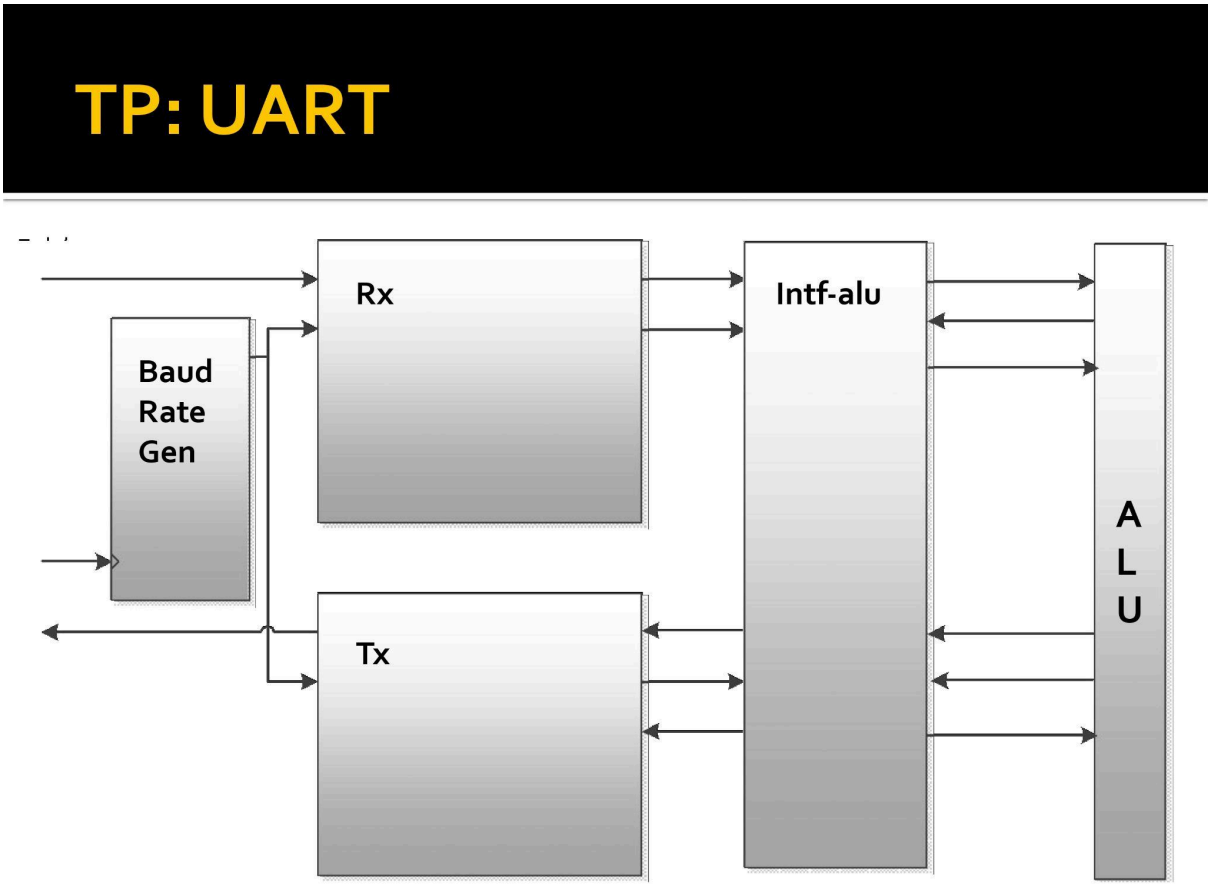
2025



1. CONSIGNA

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.



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. Baud Rate

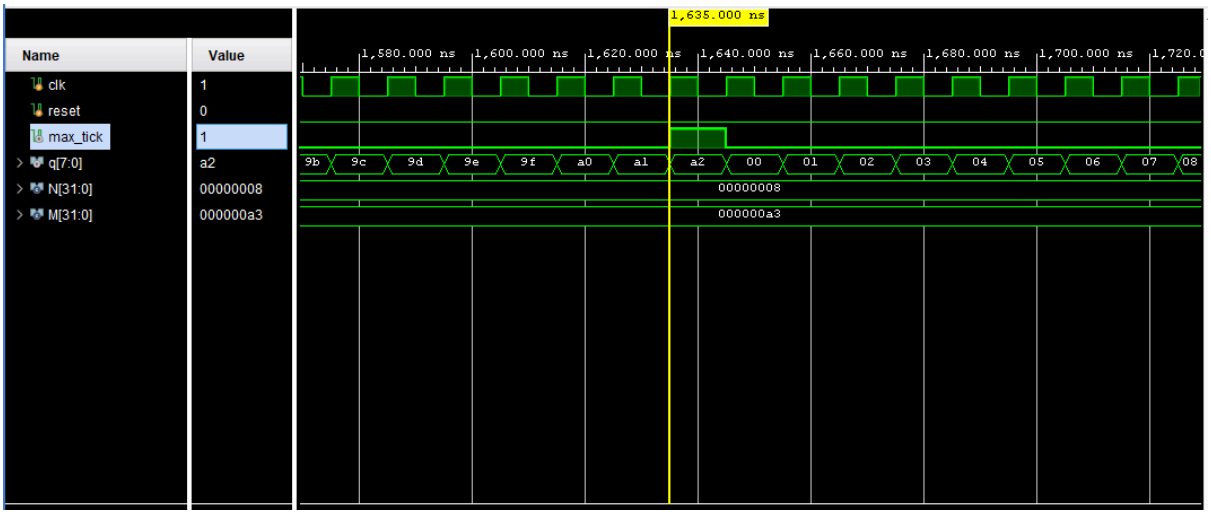
El primer módulo desarrollado fue el generador de Baud Rate, esencial para sincronizar la comunicación UART. Se implementó un divisor de frecuencia basado en un contador para ajustar la tasa de transmisión a un valor predefinido, por ejemplo, 19200 baudios.

```
module mod_m_counter
   #(
    parameter N=4,
              M=10
   )
   (
    input wire clk, 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
```

Se diseñó un testbench para verificar el correcto funcionamiento del módulo **baud_rate**, que implementa un contador de N bits con un valor máximo M. 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**, asegurando que el módulo se

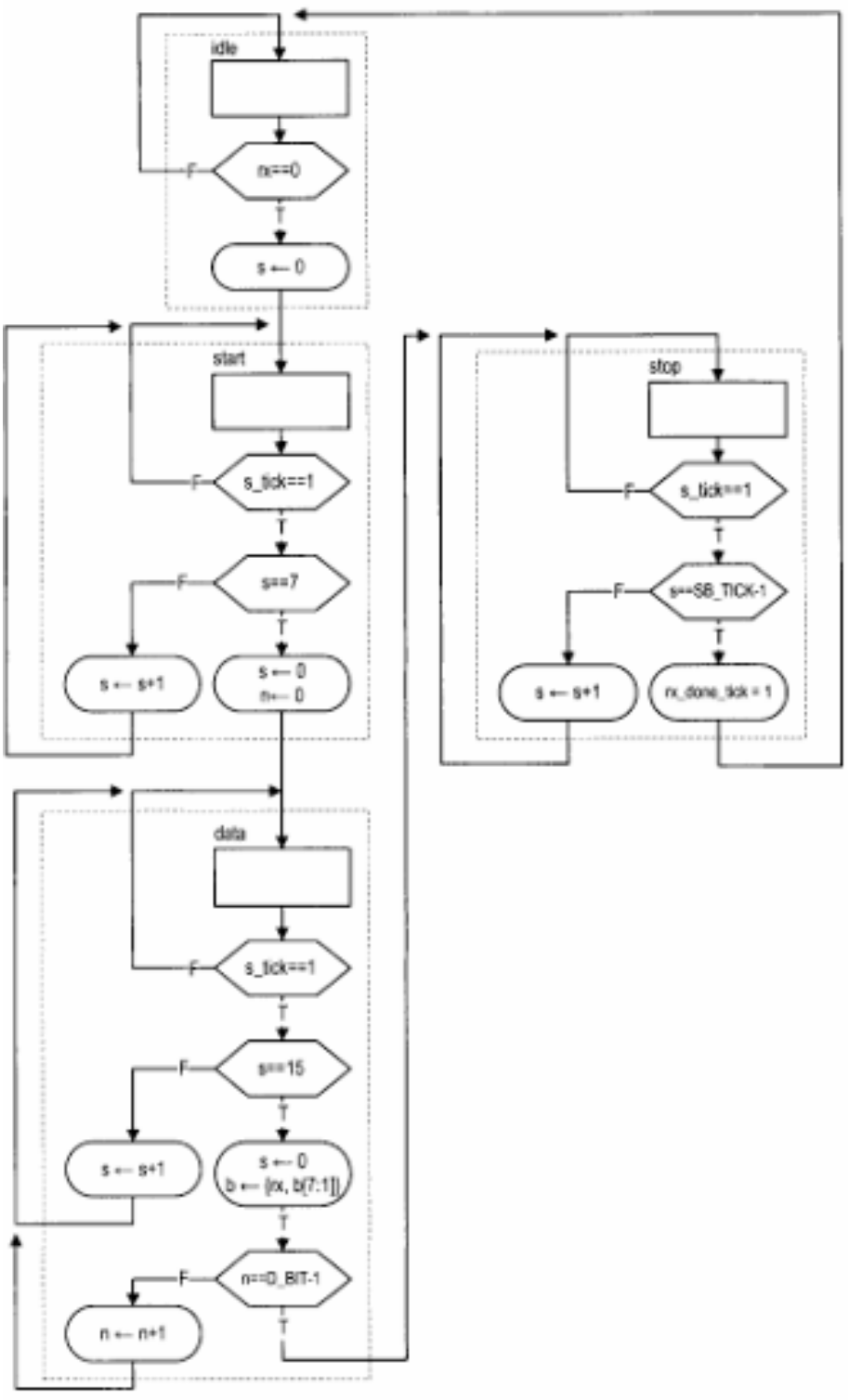
comporte según lo esperado. Durante la simulación, se monitorean las señales clave para evaluar su respuesta en diferentes instantes de tiempo.

```
module baud_rate_tb;

  parameter N = 8;
  parameter M = 163;

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

  baud_rate #(.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. UART RX

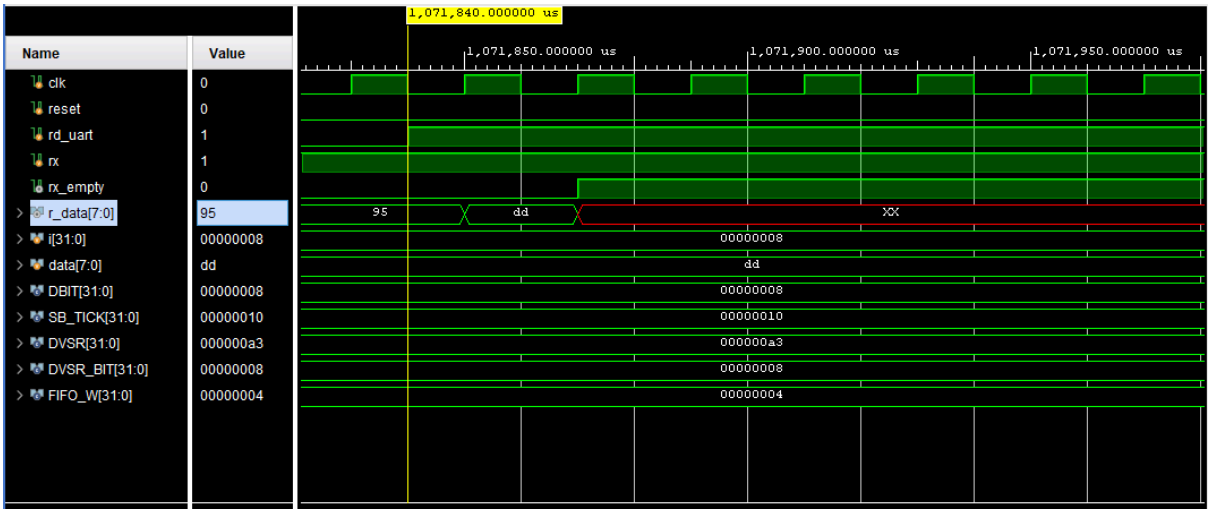
El siguiente paso fue la implementación del receptor UART. Este módulo detecta el bit de inicio, realiza la muestra de los bits de datos y verifica la integridad de la información mediante un bit de paridad. Se utilizó una máquina de estados finitos (FSM) para gestionar la captura de bits y ensamblar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



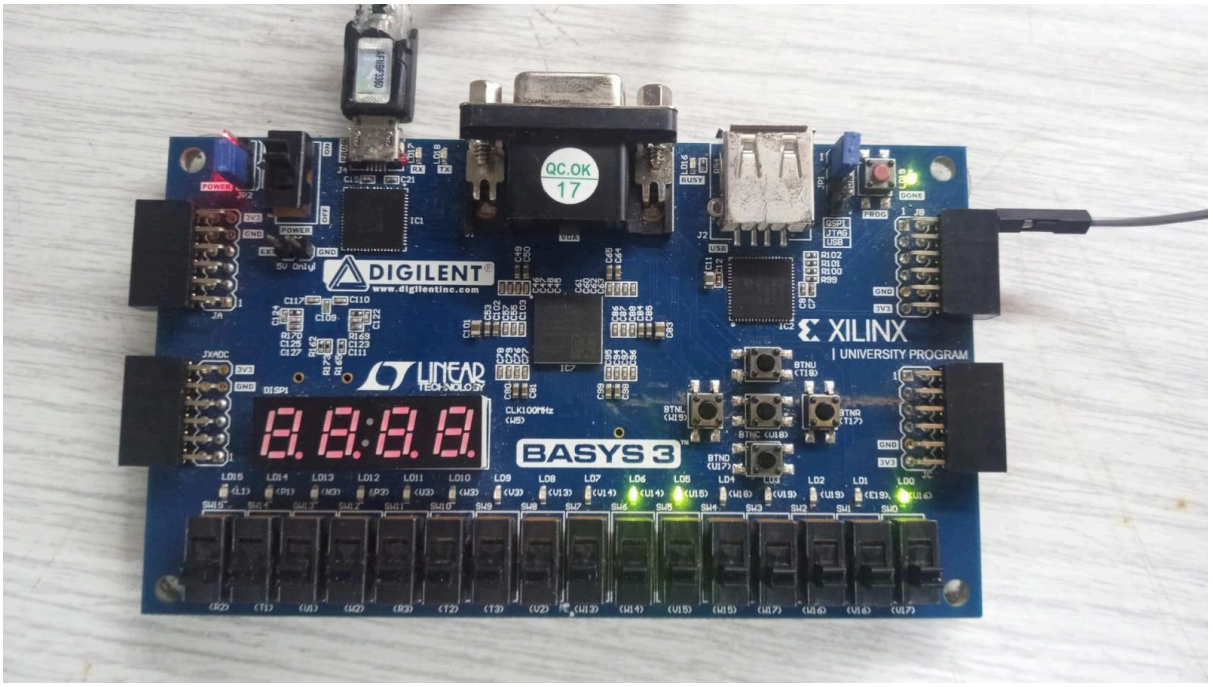
```
always @*
begin
   state_next = state_reg;
   rx_done_tick = 1'b0;
   s_next = s_reg;
   n_next = n_reg;
   b_next = b_reg;
   case (state_reg)
      idle:
         if (~rx)
            begin
               state_next = start;
               s_next = 0;
            end
      start:
         if (s_tick)
            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. La simulación permite evaluar si el módulo interpreta correctamente los bits entrantes y genera la salida esperada.

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

   #500
   data = 8'b10010101;
   rx = 0;
   #51041;
   for (i = 0; i < 8; i = i + 1) begin
      rx = data[i];
      #51041;
   end
   rx = 1;
   #51041;

   #50000
   data = 8'b11011101;
   rx = 0;
   #51041;
   for (i = 0; i < 8; i = i + 1) begin
      rx = data[i];
      #51041;
   end
   rx = 1;
   #51041;

   #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 en serie desde un dispositivo externo, como una computadora o un microcontrolador. Al enviar el carácter **"a"** (código ASCII 0x61 o 8'b01100001) desde un terminal o generador de datos UART, el módulo debe detectar correctamente el bit de inicio, leer los bits de datos y reconocer el bit de parada.



d. FIFO

Para gestionar eficientemente la comunicación y evitar la 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.

```
always @*
begin
   w_ptr_succ = w_ptr_reg + 1;
   r_ptr_succ = r_ptr_reg + 1;
   w_ptr_next = w_ptr_reg;
   r_ptr_next = r_ptr_reg;
   full_next = full_reg;
   empty_next = empty_reg;
   case ({wr, rd})
      2'b01:
         if (~empty_reg)
         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:
         if (~full_reg)
         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:
         begin
            w_ptr_next = w_ptr_succ;
            r_ptr_next = r_ptr_succ;
         end
   endcase
end
```

e. Módulo Antirrebotes (debounce)

El **módulo antirrebotes** (debounce) es un circuito o 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.

```
always @*
begin
state_next = state_reg;
q_next = q_reg;
db_tick = 1'b0;
case (state_reg)
    zero:
        begin
            db_level = 1'b0;
            if (sw)
                begin
                    state_next = wait1;
                    q_next = {N{1'b1}};
                end
        end
    wait1:
        begin
            db_level = 1'b0;
            if(sw)
                begin
                    q_next = q_reg - 1;
                    if (q_next==0)
                        begin
                            state_next = one;
                            db_tick = 1'b1;
                        end
                end
            else
                state_next = zero;
        end
    one:
        begin
            db_level = 1'b1;
            if (~sw)
```

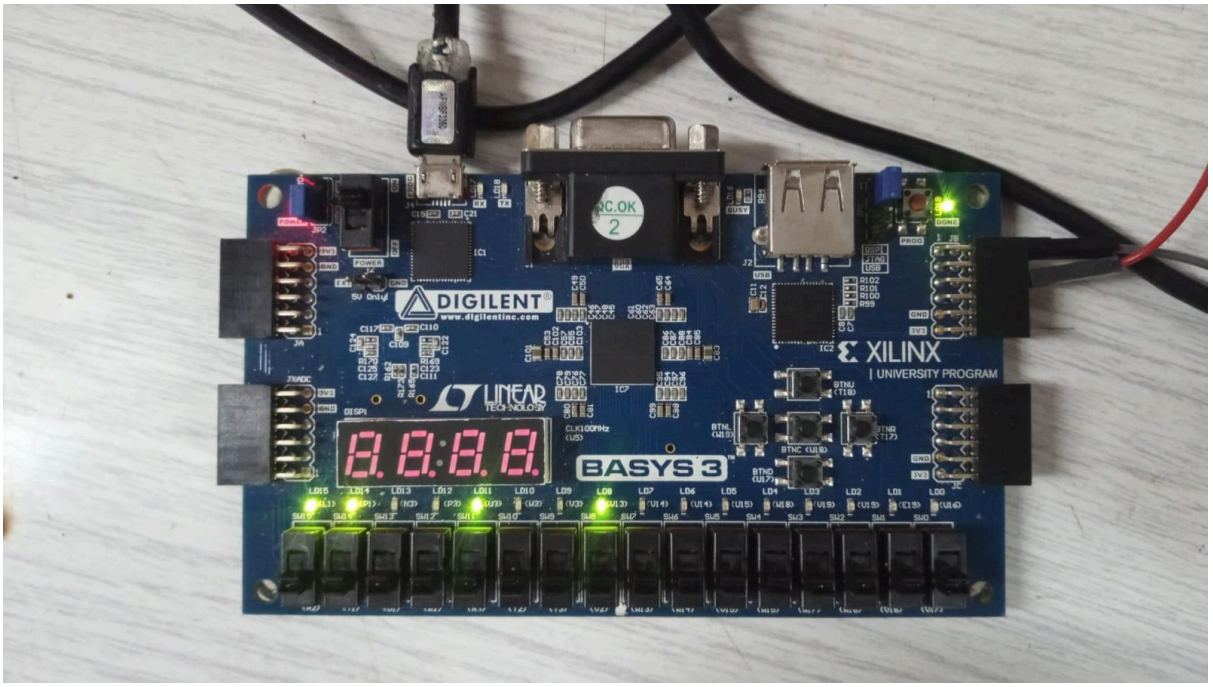
```
                begin
                    state_next = wait0;
                    q_next = {N{1'b1}};
                end
        end
    wait0:
        begin
            db_level = 1'b1;
            if (~sw)
                begin
                    q_next = q_reg - 1;
                    if (q_next == 0)
                        state_next = zero;
                end
            else
                state_next = one;
        end
    default: state_next = zero;
endcase
end
```

f. ALU

```
// 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;

reg signed [tamanioSalida-1:0] temp;

always @(*)
    begin
        case(operacion)
            ADD : temp = operandoA + operandoB;
            SUB : temp = operandoA - operandoB;
            AND : temp = operandoA & operandoB;
            OR  : temp = operandoA | operandoB;
            XOR : temp = operandoA ^ operandoB;
            SRA : temp = $signed(operandoA) >>> operandoB;
            SRL : temp = operandoA >> operandoB;
            NOR : temp = ~(operandoA | operandoB);
            default : temp = {tamanioSalida{1'b0}};
```

```
            endcase
        end
    assign resultado = temp;
```

Los datos transmitidos son: **'e' (01100100), 'd' (01100101) y 'Espacio' (00100000)**. La operación realizada es una **suma binaria (ADD)** entre los operandos, cuyo 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.

g. 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.

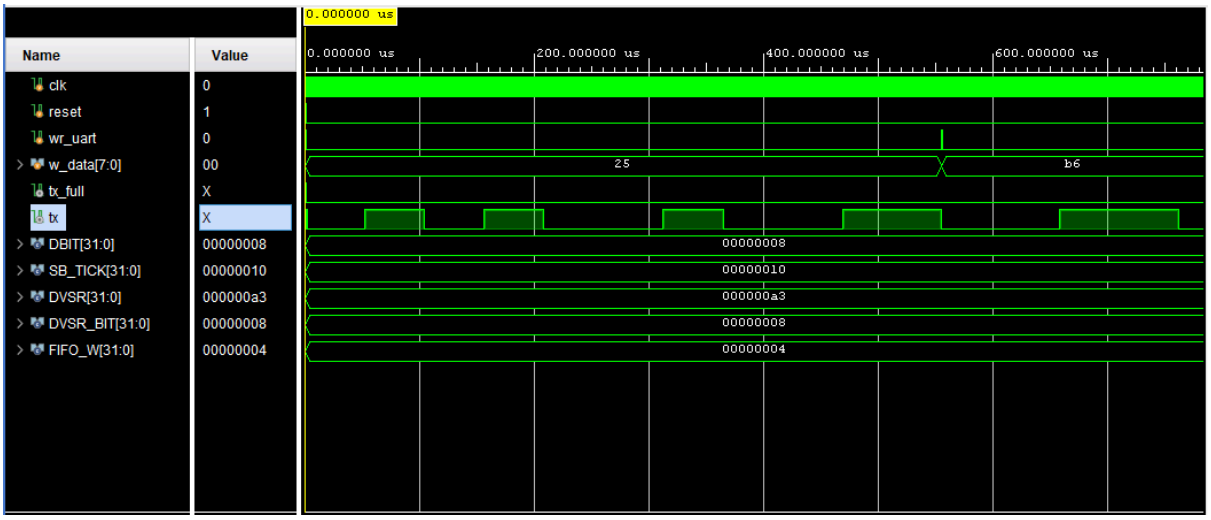
```
always @*
begin
   state_next = state_reg;
   tx_done_tick = 1'b0;
   s_next = s_reg;
   n_next = n_reg;
   b_next = b_reg;
```

```
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
assign tx = tx_reg;
```

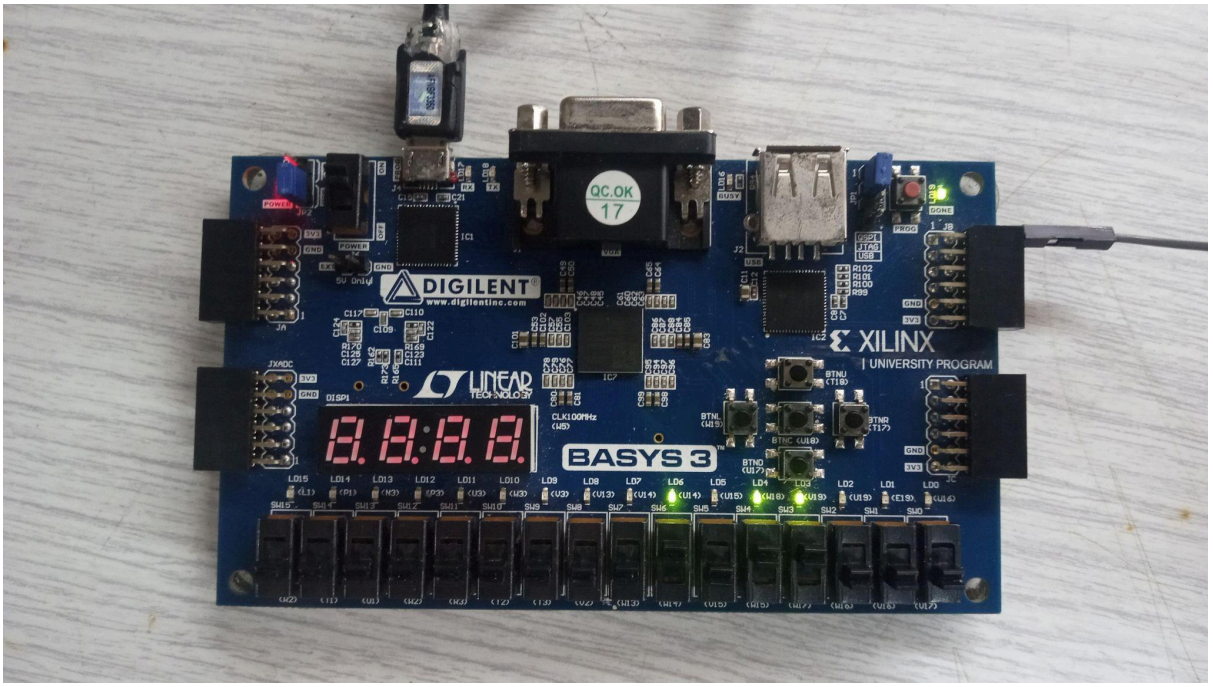
Se generan señales de reloj y reset, y se proporcionan datos de entrada al transmisor para comprobar que los bits se transmitan correctamente. Se incluyen pruebas para enviar distintos datos y verificar que el transmisor responda según lo esperado, emitiendo los bits en el orden adecuado y con los tiempos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

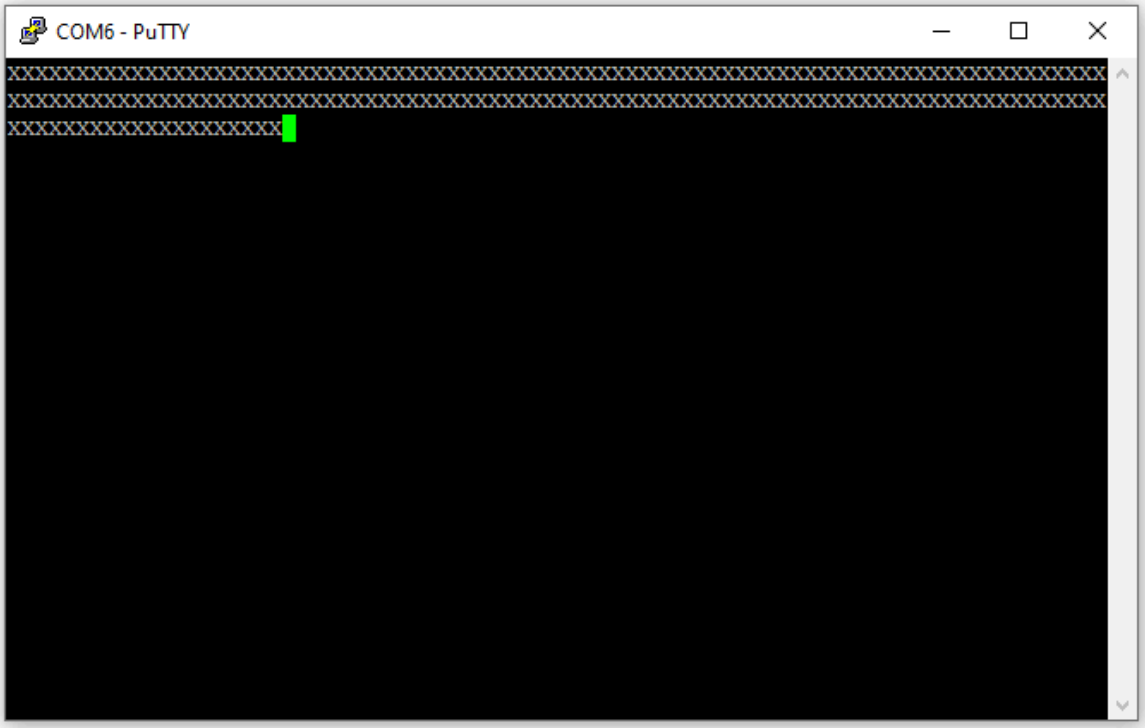
```
   initial begin
    clk = 0;
    reset = 1;
    wr_uart = 0;
    w_data = 8'h00;

    #20 reset = 0;

    #20 w_data = 8'b00100101;
    wr_uart = 1;
    #20;
    wr_uart = 0;
    #555400;

    #20 w_data = 8'b10110110;
    wr_uart = 1;
    #20;
    wr_uart = 0;
    #555400;
    $stop;
end
```

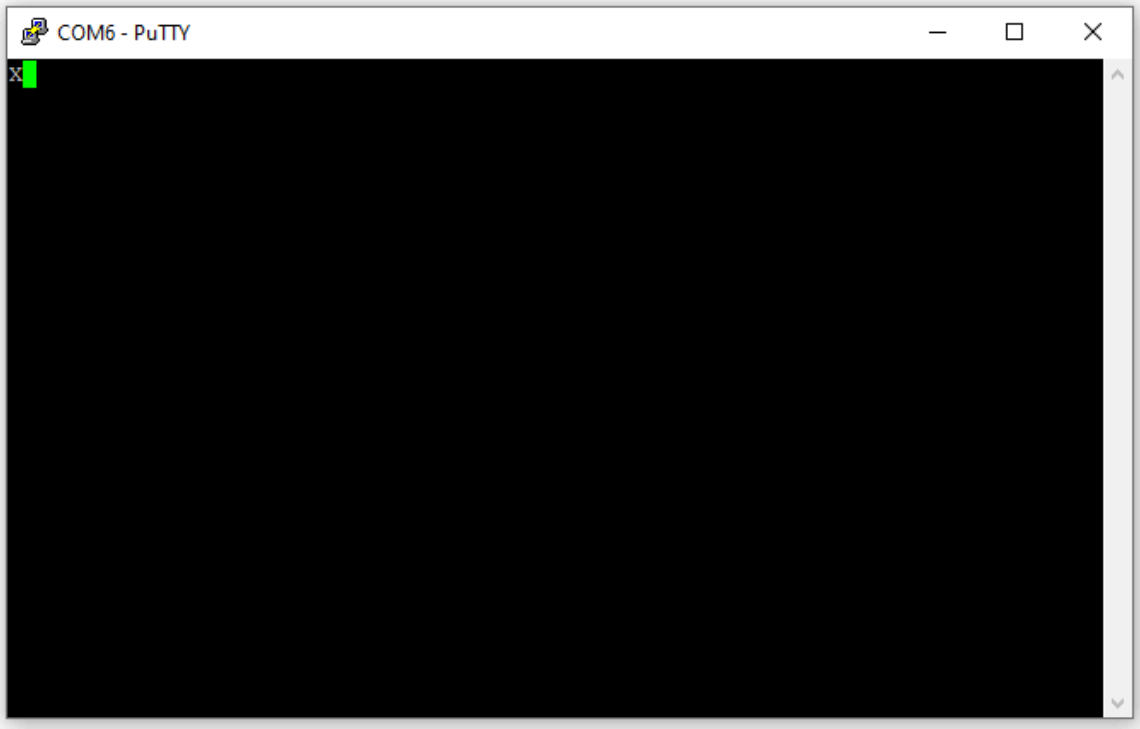


Se intenta transmitir el símbolo **X = 01011000** utilizando un pulsador. Sin el mecanismo de antirrebote, los rebotes del pulsador generan fluctuaciones en la señal **TX**, resultando en una transmisión errónea y datos corruptos en la PC.

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.



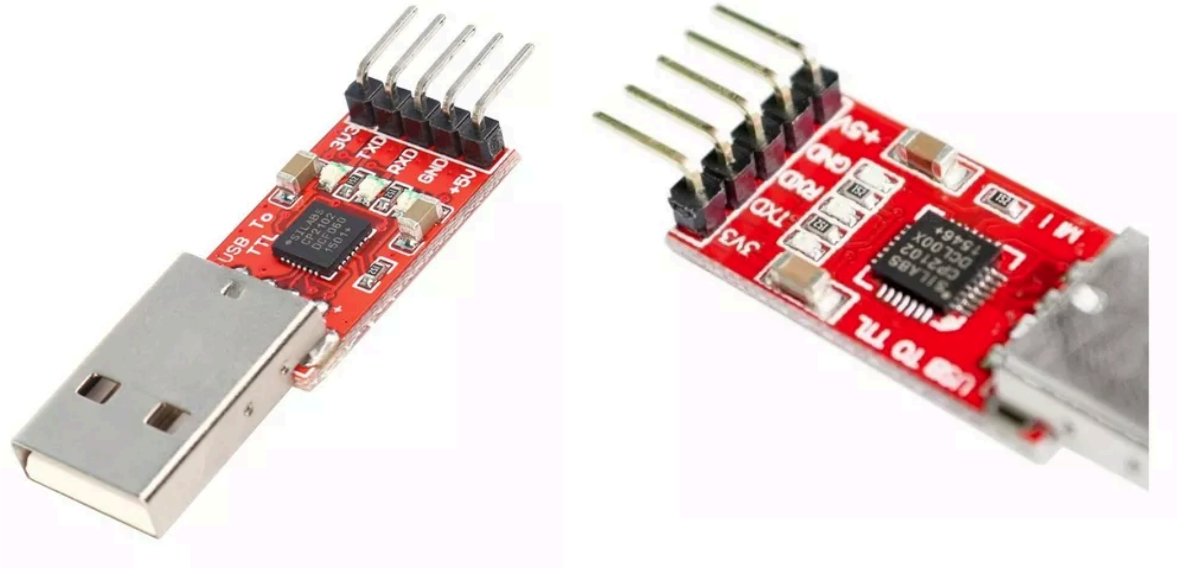


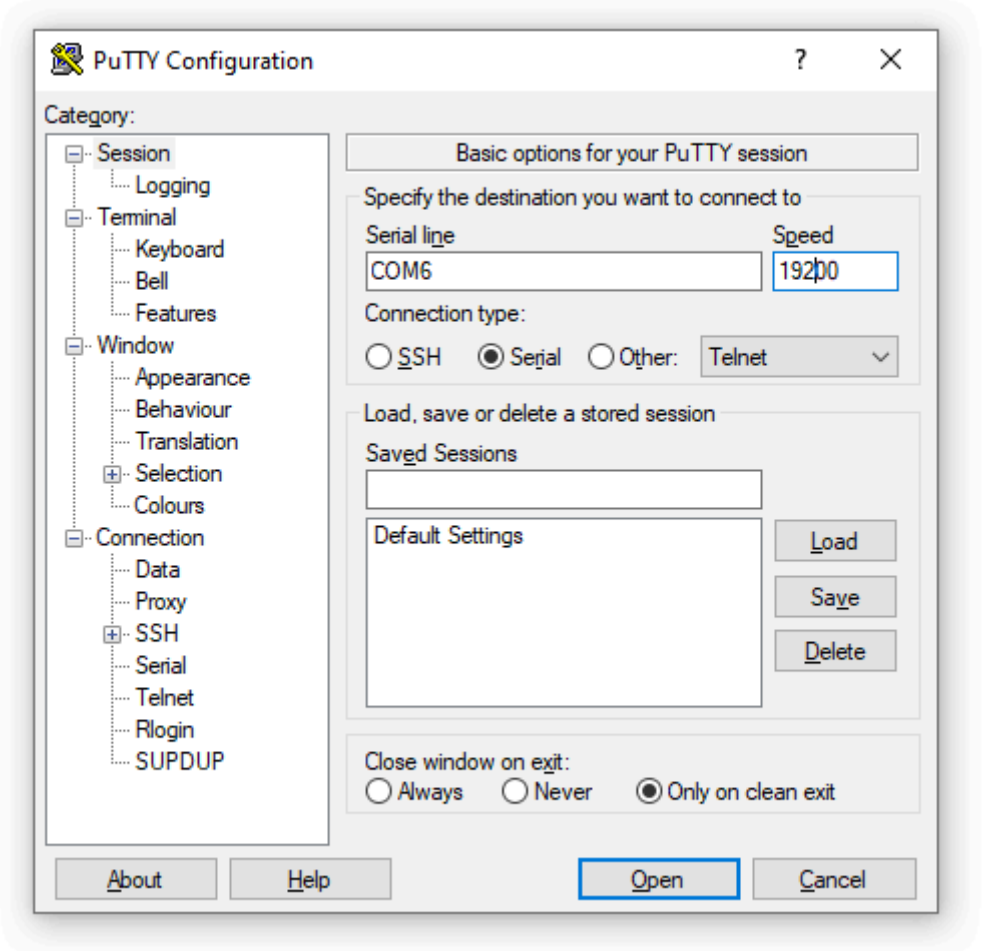


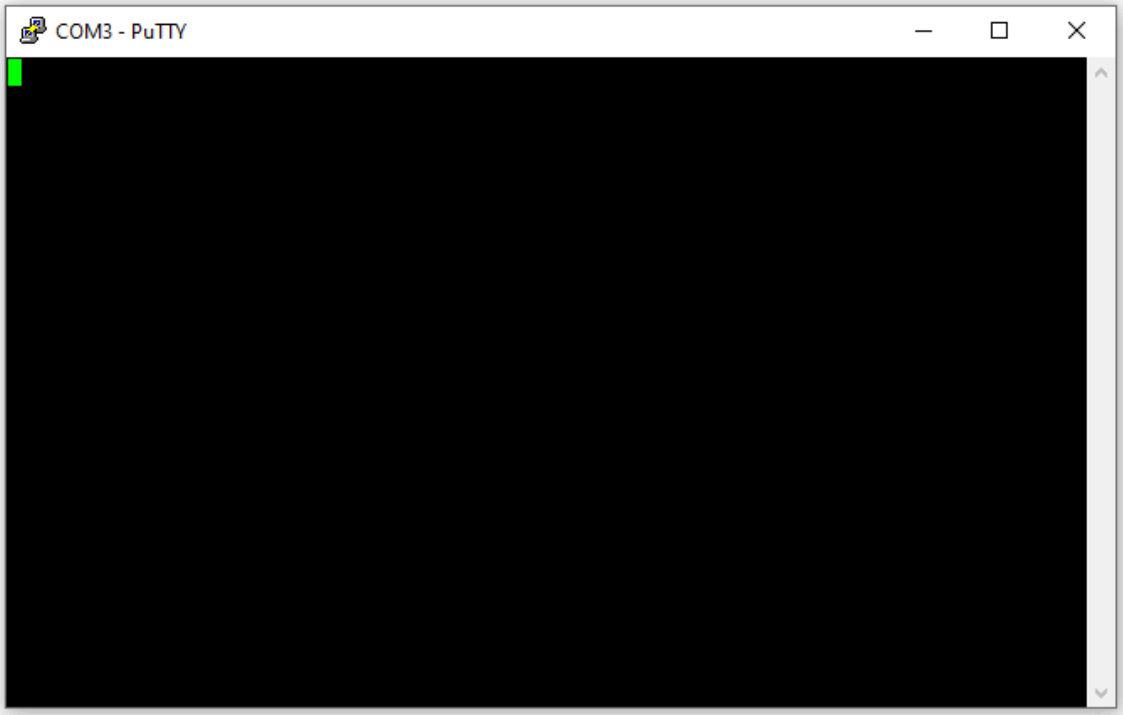
3. COMUNICACIÓN CON LA PC

Se utilizó un **conversor USB-UART** para establecer la conexión entre la PC y la placa, permitiendo la comunicación entre ambos dispositivos a través del protocolo UART. Este

conversor actúa como intermediario, transformando las señales de datos de la placa en señales USB comprensibles para la PC y viceversa, facilitando el intercambio de información entre ambos.






COM3 - PuTTY

a. Transmisor en Python

```
import serial

puerto = 'COM6'
baud_rate = 19200

try:
    ser = serial.Serial(puerto, baud_rate, timeout=1)
    print(f"Conexión establecida en {puerto} a {baud_rate} bauds.")

    while True:
        dato_uart = input("Ingresa un número (0-255) o 'salir' para
terminar: ")

        if dato_uart.lower() == 'salir':
            print("Saliendo del programa...")
            break

        if dato_uart.isdigit():
            valor_numerico = int(dato_uart)

            if 0 <= valor_numerico <= 255:
                dato_byte = valor_numerico.to_bytes(1, byteorder='big')

                print(f"Valor numérico: {valor_numerico}")
                print(f"Valor binario (8 bits): {bin(valor_numerico)}")

                ser.write(dato_byte)
                print(f"Enviando a la ALU: {dato_byte}")
            else:
                print("Error: El número debe estar en el rango de 0 a
255.")
        else:
            print("Error: Debes ingresar un número válido.")

except serial.SerialException as e:
    print(f"Error al abrir el puerto serie: {e}")

finally:
    if 'ser' in locals() and ser.is_open:
        ser.close()
        print("Puerto serie cerrado.")
```

b. Receptor en Python

```
import serial

puerto = 'COM6'
baud_rate = 19200

try:
    ser = serial.Serial(puerto, baud_rate, timeout=1)
    print(f"Receptor UART iniciado en {puerto} a {baud_rate} bauds.")
    print("Esperando datos...")

    while True:
        dato_recibido = ser.read(1)

        if dato_recibido:
            valor_numerico = int.from_bytes(dato_recibido,
byteorder='big')

            print(f"Byte recibido: {dato_recibido}")
            print(f"Valor numérico: {valor_numerico}")
            print(f"Valor binario: {bin(valor_numerico)}")
            print(f"Valor hexadecimal: {hex(valor_numerico)}")
            print("------")
        else:
            print("No se recibieron datos. Cerrando receptor...")
            break

except serial.SerialException as e:
    print(f"Error al abrir el puerto serie: {e}")

finally:
    if 'ser' in locals() and ser.is_open:
        ser.close()
        print("Puerto serie cerrado.")
```

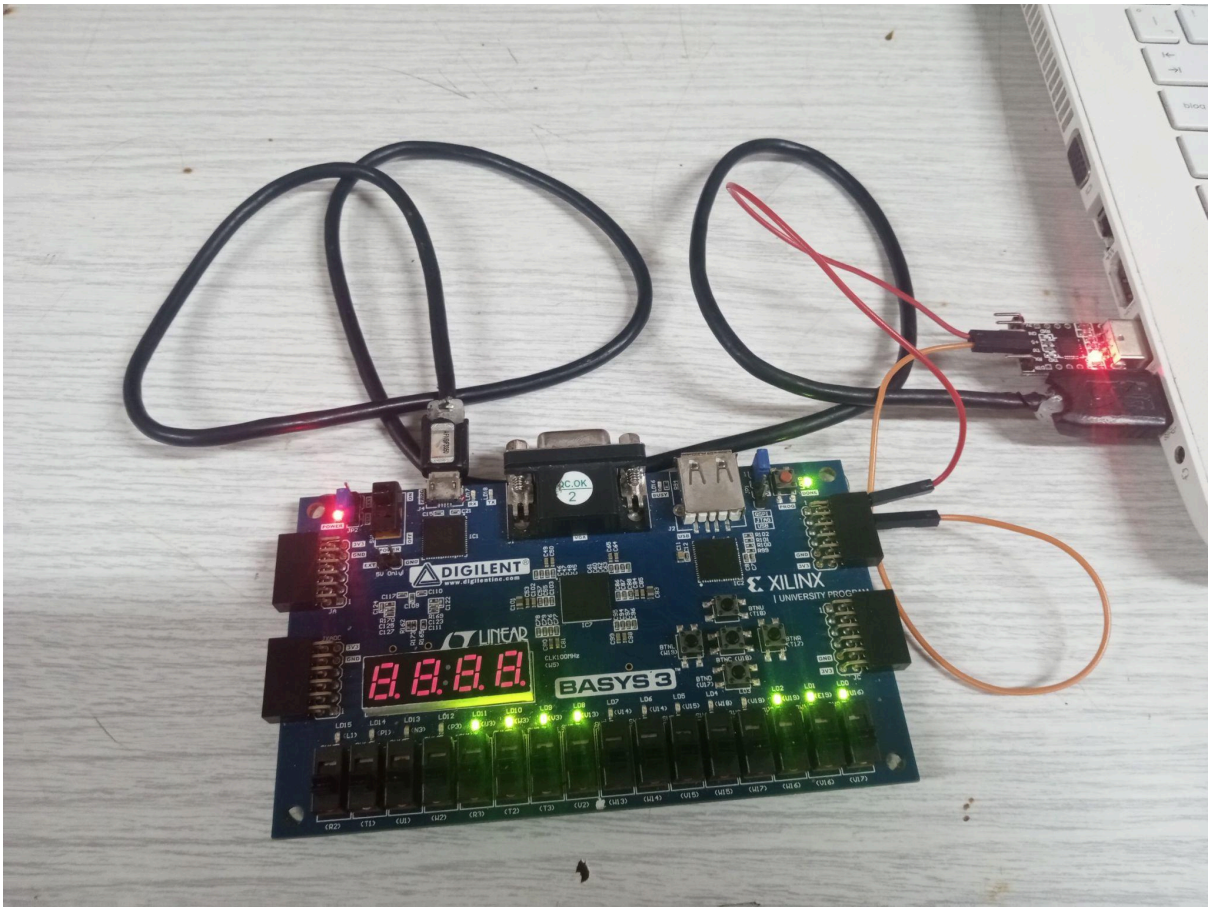
4. IMPLEMENTACIÓN

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.

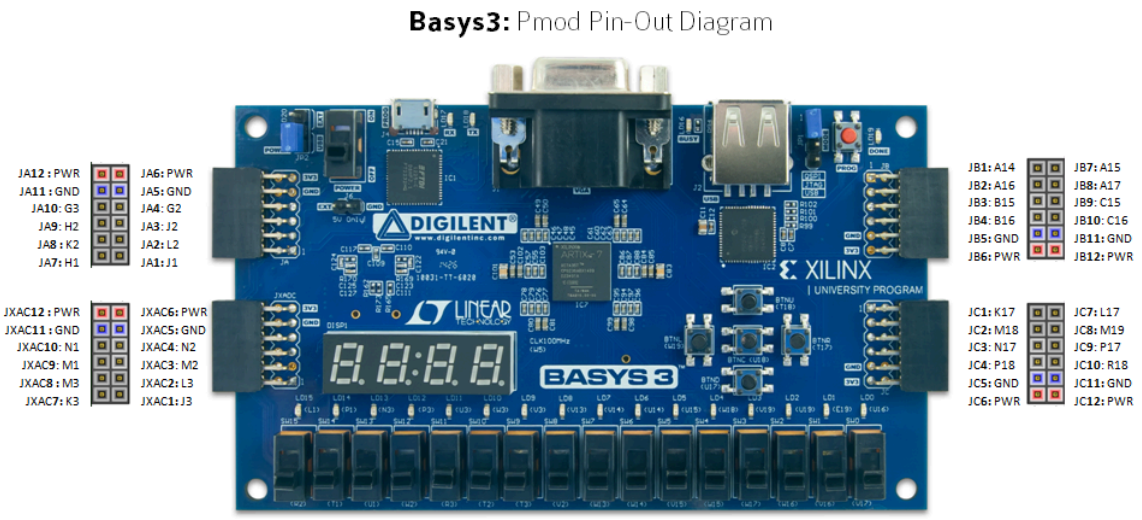
```
module top
   #( // Default setting:
      // 19200 baud, 8 data bits, 1 stop bit, 2^2 FIFO
      parameter DBIT = 8,
                SB_TICK = 16,
                DVSR = 325,   // baud rate divisor
                              // DVSR = 100MHz/(16*baudios)
                DVSR_BIT = 10,// # bits 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
```


DIGILENT
XILINX
LINEAR
BASYS 3

5. PINES




```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 }
[get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5}
[get_ports clk]

set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 }
[get_ports {rx}];#Sch name = JB1
set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 }
[get_ports {rx_empty}];#Sch name = JB2
set_property -dict { PACKAGE_PIN B15   IOSTANDARD LVCMOS33 }
[get_ports {tx}];#Sch name = JB3
set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVCMOS33 }
[get_ports {tx_full}];#Sch name = JB4
set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 }
[get_ports {rx_empty}];#Sch name = JB7

##Buttons
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 }
[get_ports reset]
set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 }
[get_ports rd_uart]
#set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 }
[get_ports btnL]
#set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 }
[get_ports btnR]
set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 }
[get_ports wr_uart]
```

6. REPOSITORIO DE GITHUB

https://github.com/nachoborgatello/uart_tp2

7. REFERENCIAS

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