Arquitectura de Computadoras

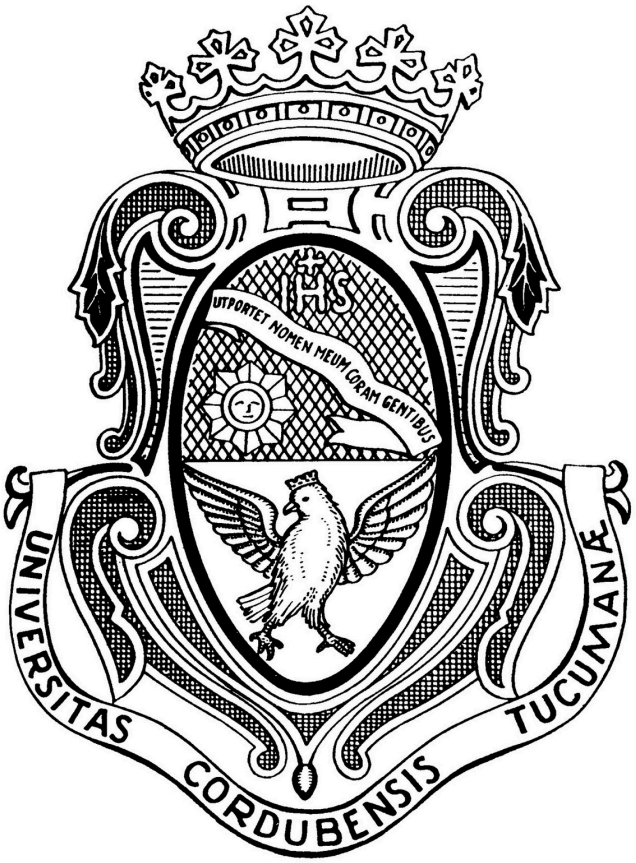
Trabajo Práctico 2

UART

Autores

BORGATELLO, Ignacio

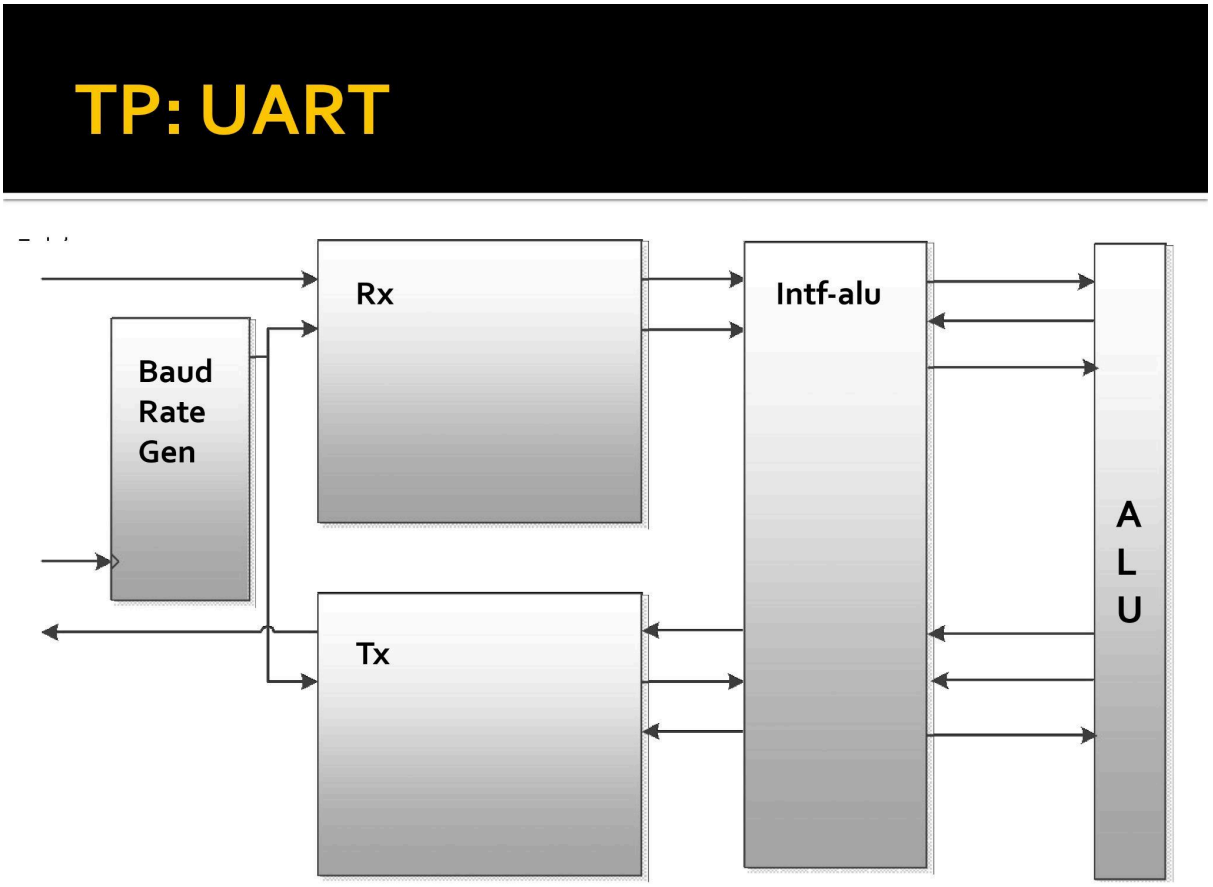
DALLARI LARROSA, Gian Franco



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:

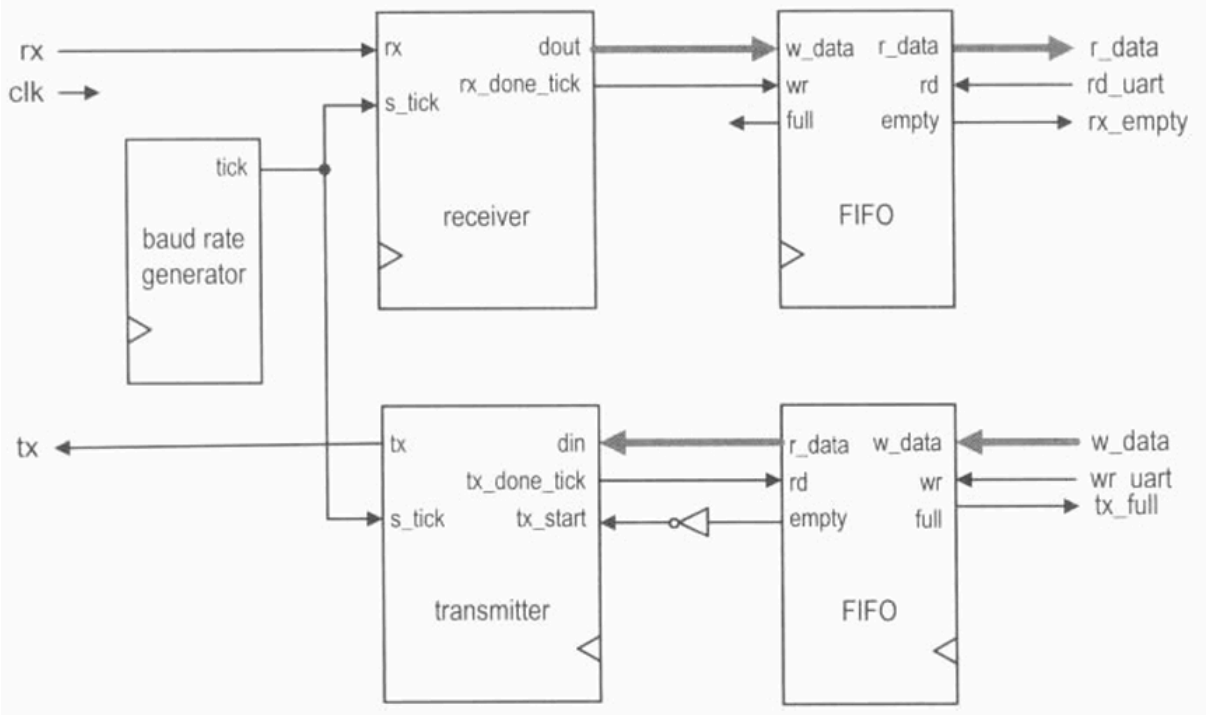
1. Generador de Baud Rate, que controla la velocidad de transmisión de datos.
2. Receptor UART (Rx), que sigue una secuencia de estados para recibir datos en serie, sincronizándose con los bits de inicio, datos y parada.
3. Transmisor UART (Tx), que envía datos en serie con el formato adecuado.
4. Interfaz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



2. DESARROLLO

a. Introducción

La comunicación UART (Universal Asynchronous Receiver-Transmitter) es un protocolo de comunicación serie utilizado en sistemas embebidos para la transferencia de datos entre dispositivos. 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, 19600 baudios.

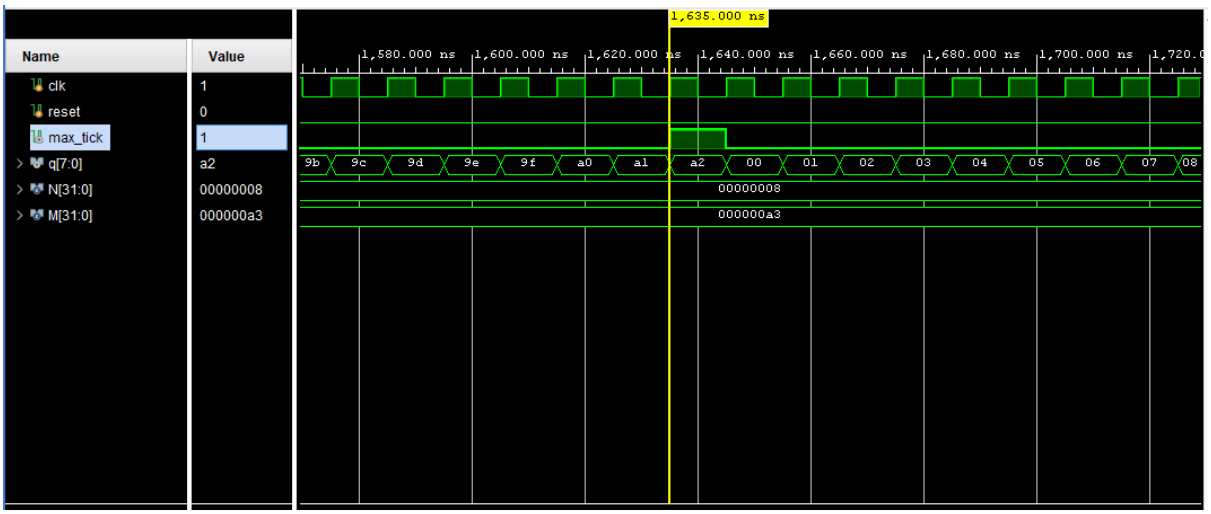
```
module baud_rate
   #(
    parameter N=4, // Número de bits en el contador
              M=10 // Valor máximo del contador (mod-M)
   )
   (
    input wire clk, reset,      // Entradas: señal de reloj y
reset
    output wire max_tick,       // Salida: indica cuando el
contador alcanza M-1
    output wire [N-1:0] q       // Salida: valor actual del
contador
   );

   // Declaración de señales internas
   reg [N-1:0] r_reg;   // Registro que almacena el estado actual
del contador
   wire [N-1:0] r_next; // Señal para el próximo estado del
contador

   // Lógica secuencial (Flip-Flop con reset síncrono)
   always @(posedge clk)
      if (reset)           // Si reset está activo, reinicia el
contador a 0
         r_reg <= 0;
      else
         r_reg <= r_next;  // De lo contrario, actualiza el
contador con el siguiente estado

   // Lógica de transición de estados
   assign r_next = (r_reg == (M-1)) ? 0 : r_reg + 1; // Reinicia
el contador cuando alcanza M-1, sino incrementa

   // Generación de la señal max_tick
   assign max_tick = (r_reg == (M-1)) ? 1'b1 : 1'b0; // Activa
max_tick cuando el contador llega a M-1

   // Asignación de la salida del contador
   assign q = r_reg; // La salida q refleja el estado actual del
contador
endmodule
```

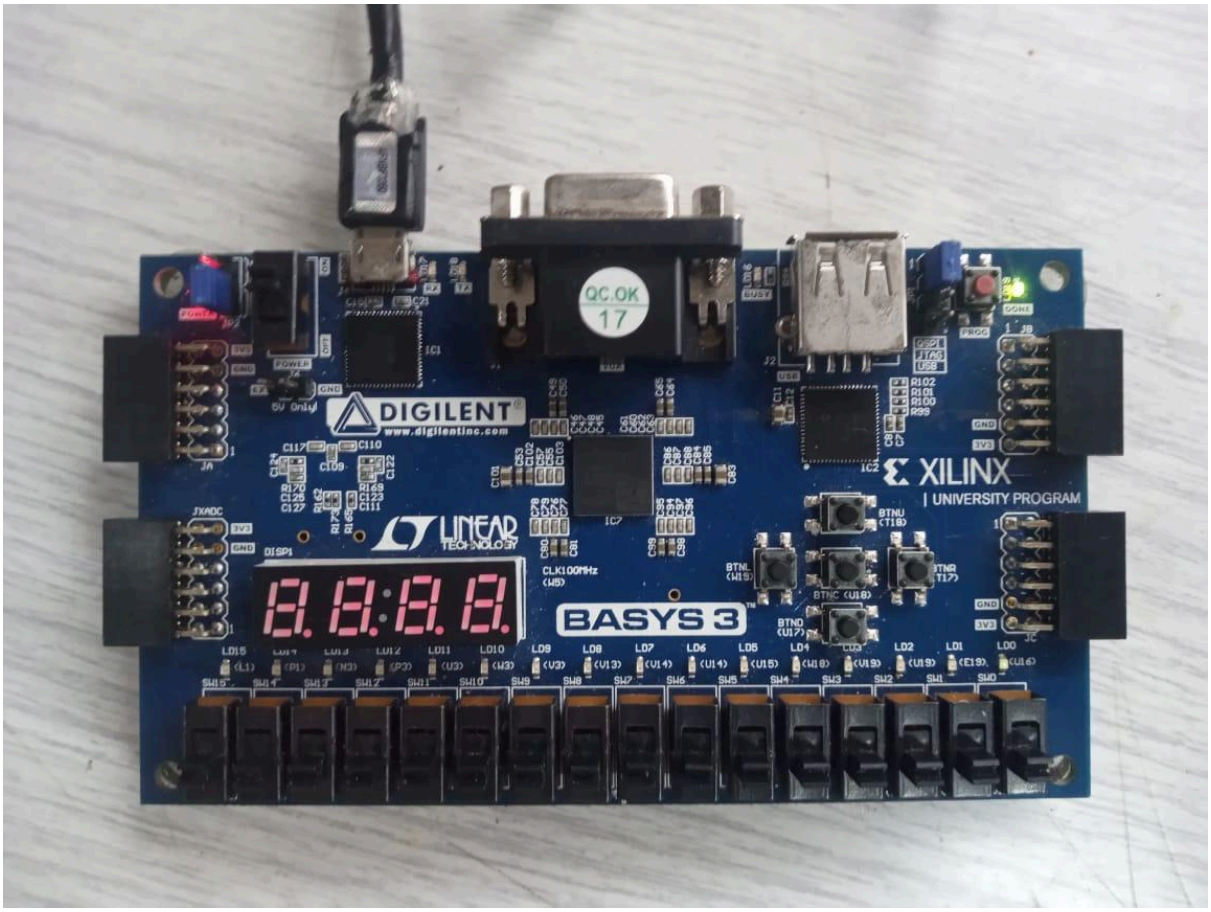
Se desarrolló el módulo Baud Rate, el cual fue probado tanto en simulación como en la placa. El módulo de prueba es el siguiente:

```
`timescale 1ns / 1ps

module baud_rate_tb;

  // Parámetros del módulo a testear
  parameter N = 8;
  parameter M = 163;

  // Señales de prueba
  reg clk;
  reg reset;
  wire max_tick;
  wire [N-1:0] q;

  // Instancia del módulo bajo prueba (DUT)
  baud_rate #(.N(N), .M(M)) dut (
    .clk(clk),
    .reset(reset),
    .max_tick(max_tick),
    .q(q)
  );

  // Generación de reloj (período de 10 ns => 100 MHz)
  always #10 clk = ~clk;

  // Procedimiento de prueba
  initial begin
    // Inicialización
    clk = 0;
    reset = 1;

    // Mantener reset activo por un tiempo
    #20 reset = 0;

    // Simular por suficiente tiempo para observar varios ciclos
    #200;

    // Finalizar simulación
    $stop;
  end

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

```
endmodule
```

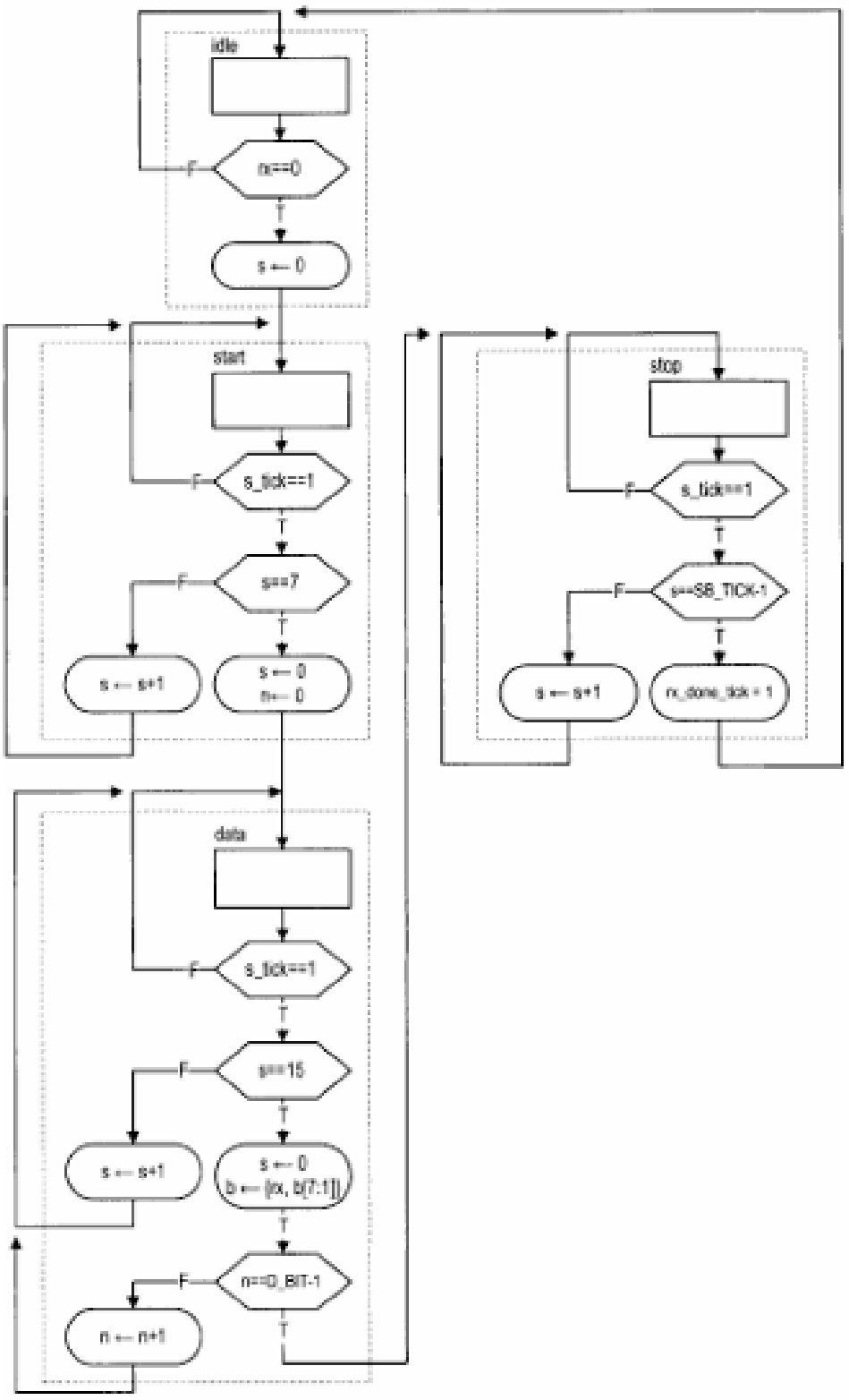
El resultado obtenido tras la simulación es el siguiente:





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.

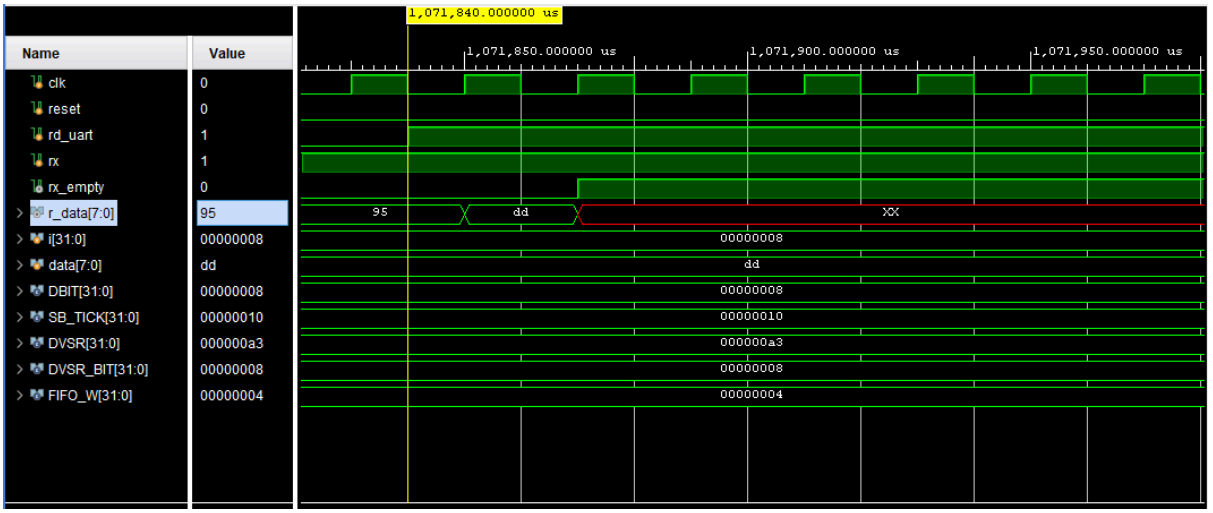


```
// FSMD next-state logic
   always @*
   begin
      state_next = state_reg;
      rx_done_tick = 1'b0;
      s_next = s_reg;
      n_next = n_reg;
      b_next = b_reg;
      case (state_reg)
         idle:
            if (~rx)
               begin
                  state_next = start;
                  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;
```

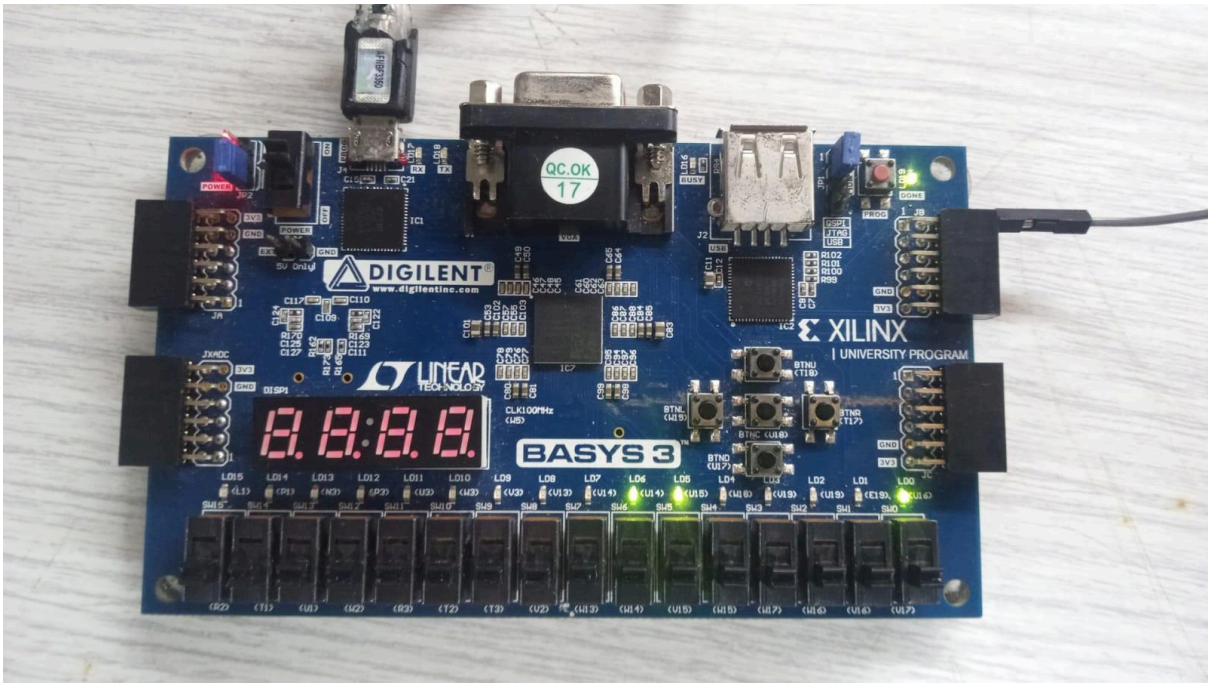
```
      endcase
   end
```

Se desarrolló el módulo UART Rx, el cual fue probado tanto en simulación como en la placa. El módulo de prueba es el siguiente:

```
initial begin
      // Inicialización
      clk = 0;
      reset = 1;
      rd_uart = 0;
      rx = 1; // Línea RX en estado inactivo (UART idle)
      #20 reset = 0;  // Desactivar reset

      #500
      // Simulación de trama UART (envío de un byte)
      data = 8'b10010101;
      rx = 0;  // Bit de inicio
      #51041;  // Esperar un ciclo de baud
      for (i = 0; i < 8; i = i + 1) begin
         rx = data[i];  // Enviar cada bit de datos
         #51041;  // Esperar un ciclo de baud entre bits
      end
      rx = 1;  // Bit de parada
      #51041;  // Esperar tiempo de stop bit

      #50000
      // Simulación de trama UART (envío de un byte)
      data = 8'b11011101;
      rx = 0;  // Bit de inicio
      #51041;  // Esperar un ciclo de baud
      for (i = 0; i < 8; i = i + 1) begin
         rx = data[i];  // Enviar cada bit de datos
         #51041;  // Esperar un ciclo de baud entre bits
      end
      rx = 1;  // Bit de parada
      #51041;  // Esperar tiempo de stop bit

      #500
      rd_uart = 1;
      #500
      rd_uart = 0;

      // Finalizar simulación
      #500;
      $finish;
   end
```



Se envía el dato a = 01100001 y se observa en la placa lo siguiente



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.

```
// 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
   end
```

e. Módulo Antirrebotes (debounce)

El **módulo antirrebotes** (o **debounce**) es un circuito o técnica utilizada para eliminar los rebotes eléctricos generados cuando se presiona o suelta un interruptor mecánico. Esto es fundamental en sistemas digitales y microcontroladores, donde los pulsos no deseados pueden causar múltiples detecciones en lugar de una sola.

Los interruptores y botones mecánicos no cambian instantáneamente entre abierto y cerrado; en su lugar, el contacto metálico vibra durante unos milisegundos, produciendo señales erráticas. Si no se controla, un microcontrolador podría interpretar varios pulsos en lugar de uno solo.

```
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;
```

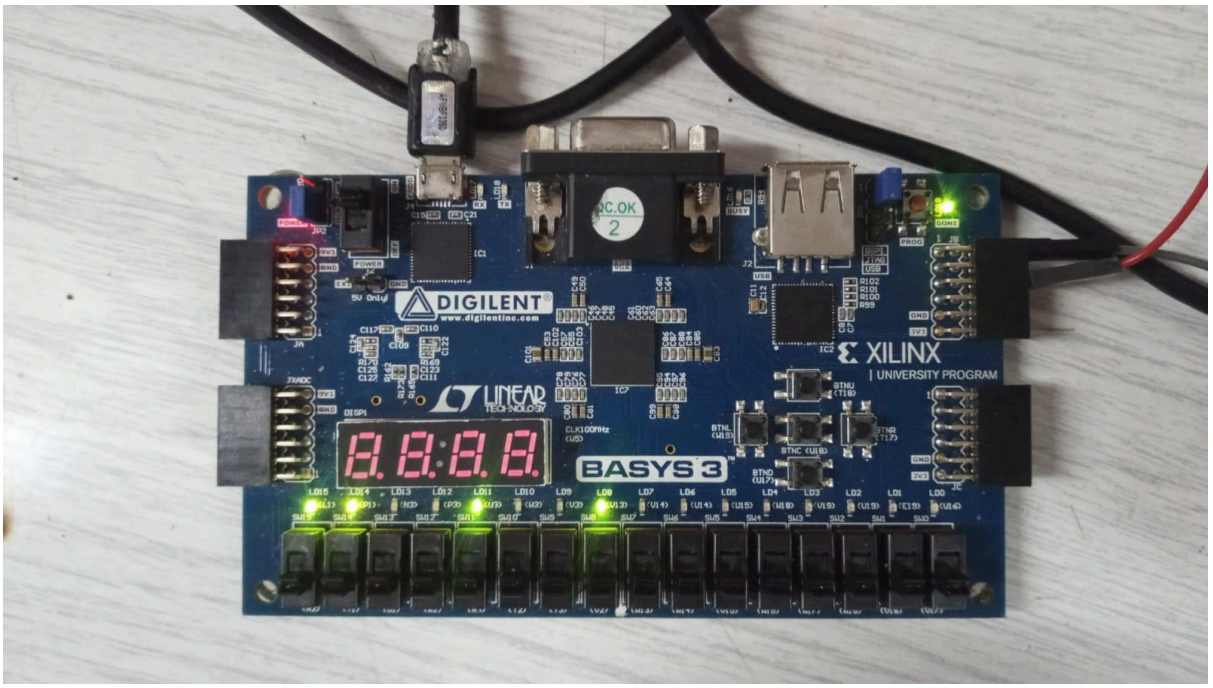
```
                    q_next = {N{1'b1}};
                end
        end
    wait0:
        begin
            db_level = 1'b1;
            if (~sw)
                begin
                    q_next = q_reg - 1;
                    if (q_next == 0)
                        state_next = zero;
       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

f. ALU

La ALU (Arithmetic Logic Unit) es el módulo encargado de realizar operaciones aritméticas y lógicas sobre los datos recibidos. Esta unidad recibe los operandos desde el módulo latch y ejecuta operaciones como suma, resta, AND, OR y desplazamientos. Dependiendo del código de operación recibido, la ALU genera un resultado que puede ser utilizado por otros módulos del sistema.

```
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 enviados son: e = 01100100, d = 01100101 y Espacio = 00100000. El resultado esperado corresponde a la operación ADD (suma) de los operandos, obteniendo 11001001.



Los leds más significativos [8:15] muestran el dato recibido por UART, mientras que los leds [0:7] muestran el resultado obtenido luego de la operación.

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.

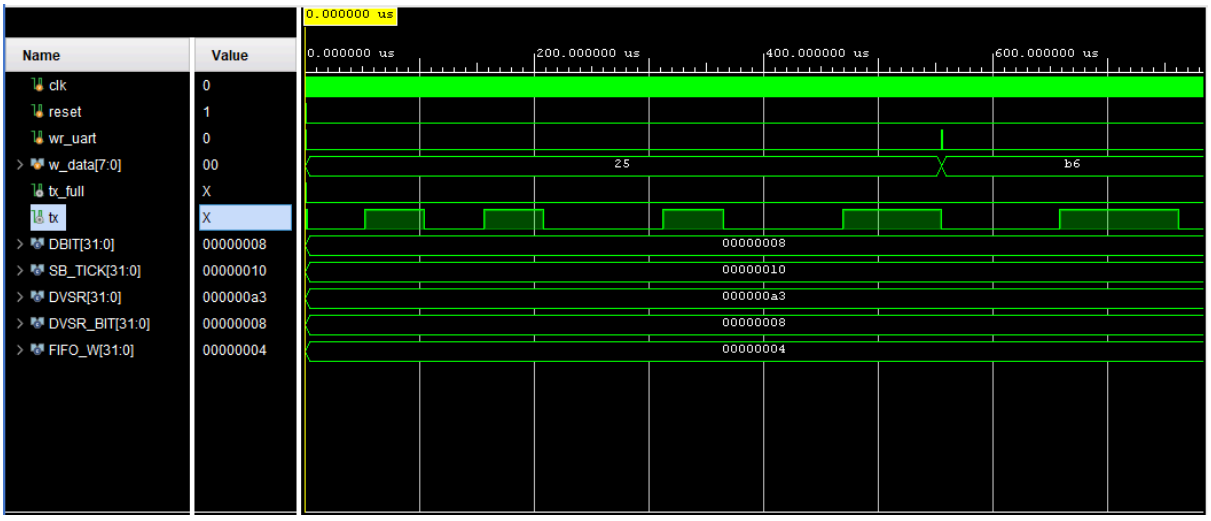
```
// FSMD next-state logic & functional units
always @*
begin
   state_next = state_reg;
   tx_done_tick = 1'b0;
   s_next = s_reg;
   n_next = n_reg;
   b_next = b_reg;
   tx_next = tx_reg ;
   case (state_reg)
      idle:
         begin
            tx_next = 1'b1;
            if (tx_start)
               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
// output
```

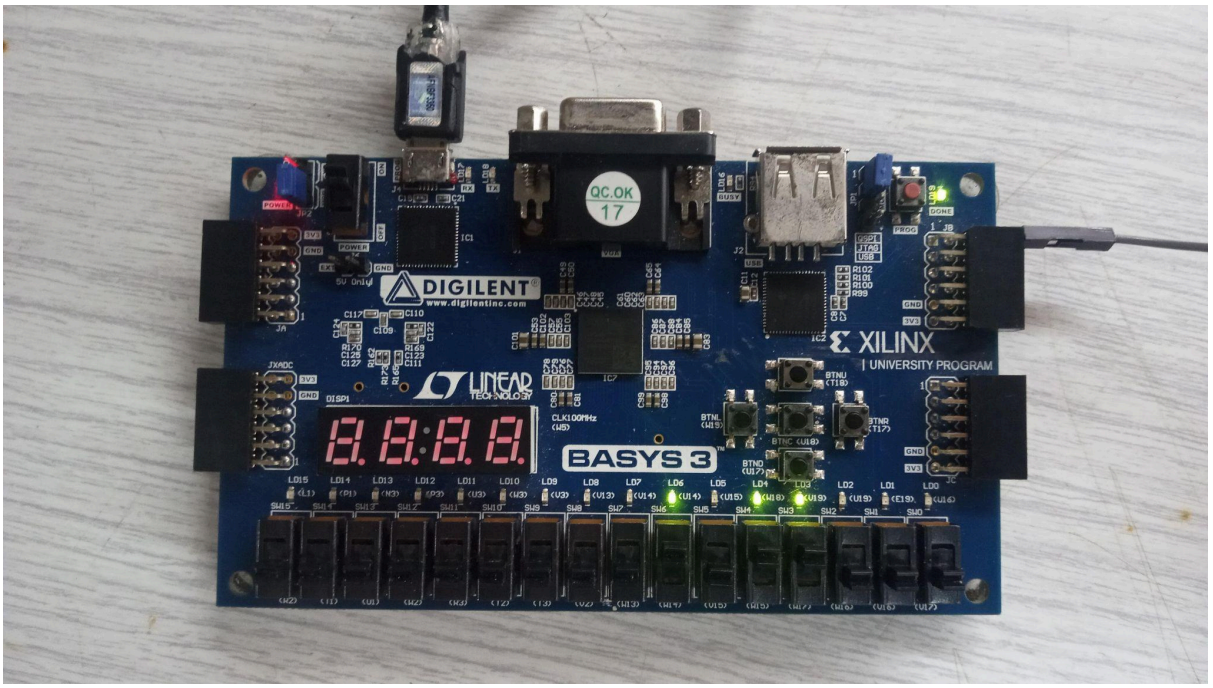
```
assign tx = tx_reg;
```

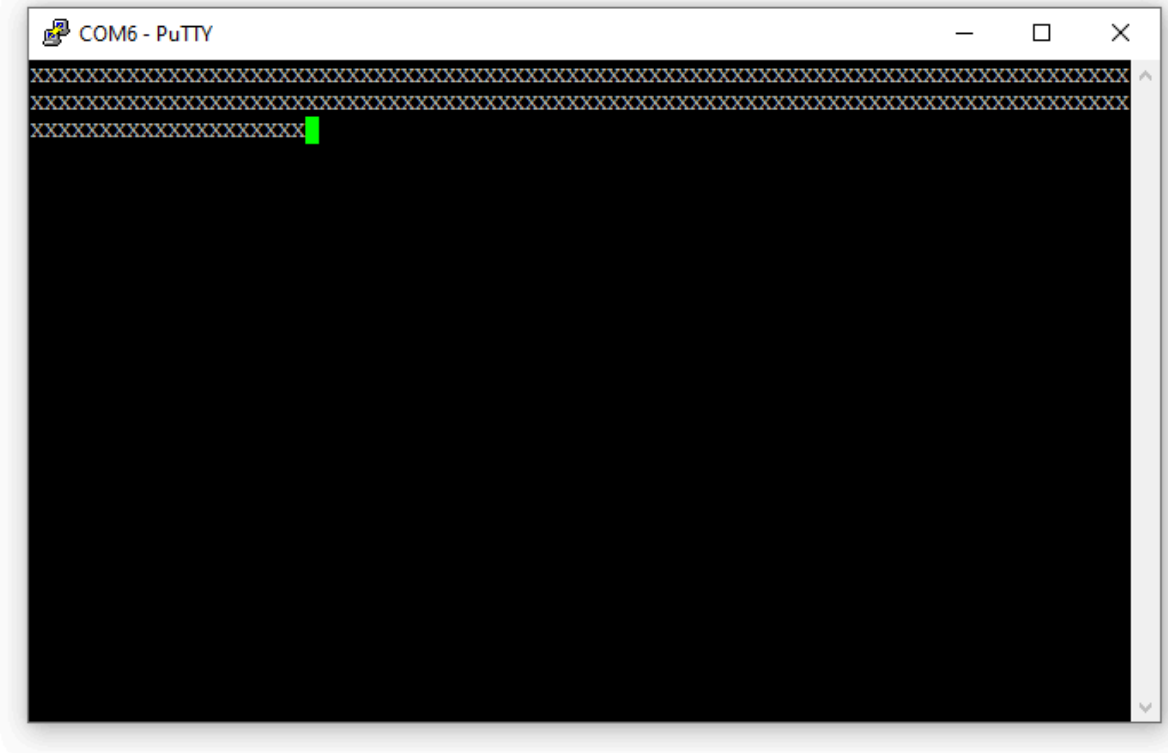
Se desarrolló el módulo UART Tx, el cual fue probado tanto en simulación como en la placa. El módulo de prueba es el siguiente:

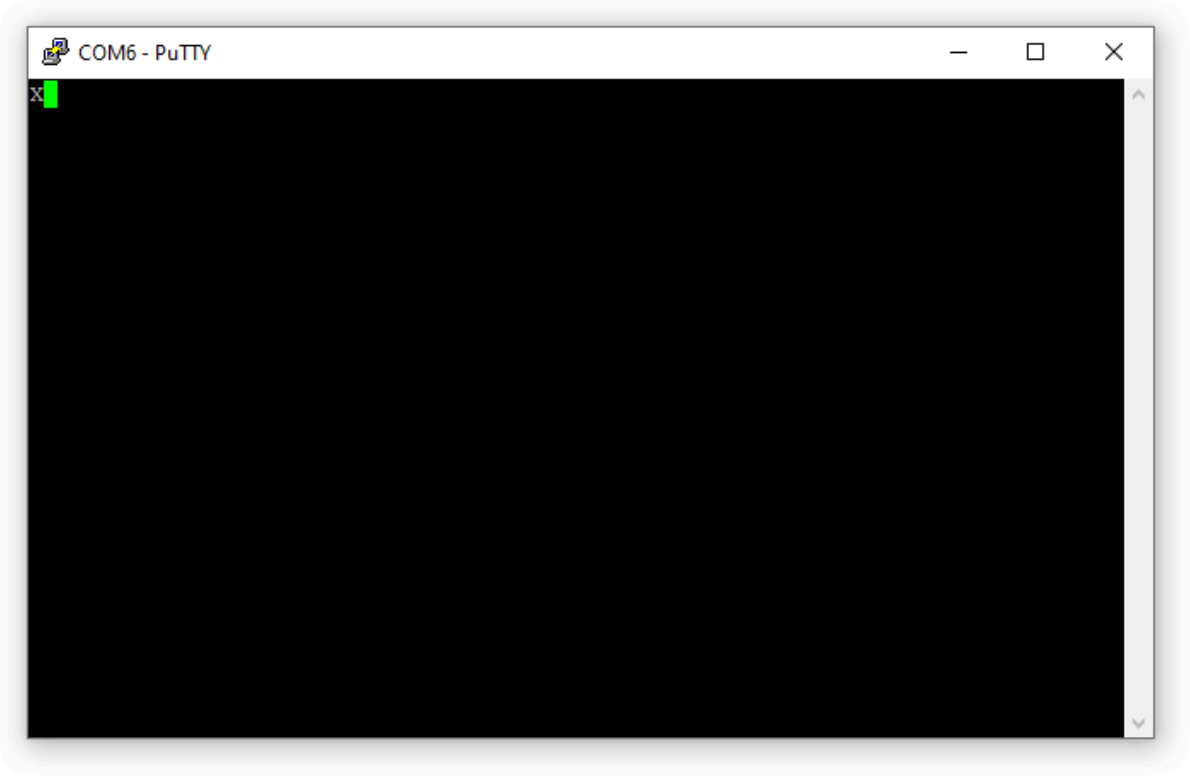
```
// Procedimiento de prueba
initial begin
    // Inicialización
    clk = 0;
    reset = 1;
    wr_uart = 0;
    w_data = 8'h00;

    // Liberar reset
    #20 reset = 0;

    // Enviar datos al UART
    #20 w_data = 8'b00100101; // Dato de prueba
    wr_uart = 1;
    #20;
    wr_uart = 0;
    #555400; // Esperar suficiente tiempo

    // Enviar datos al UART
    #20 w_data = 8'b10110110; // Dato de prueba
    wr_uart = 1;
    #20;
    wr_uart = 0;
    #555400; // Esperar suficiente tiempo

    // Finalizar simulación
    $stop;
end
```



Se intenta transmitir el símbolo **X = 01011000**. A continuación, se presentan las observaciones realizadas en la placa y en la PC, primero sin aplicar el mecanismo de antirrebote en el pulsador y luego con dicho mecanismo implementado.

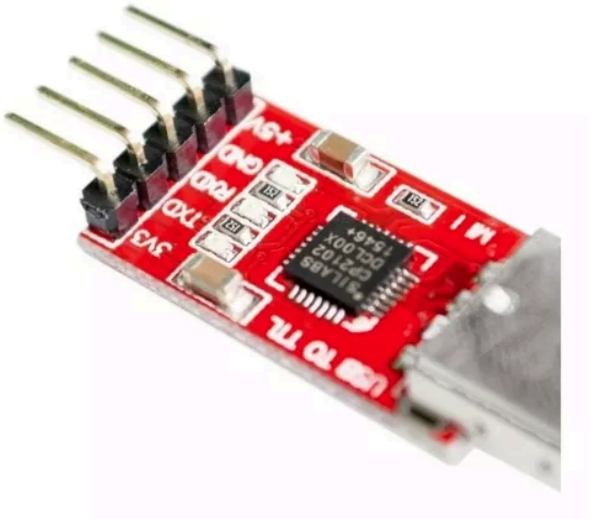


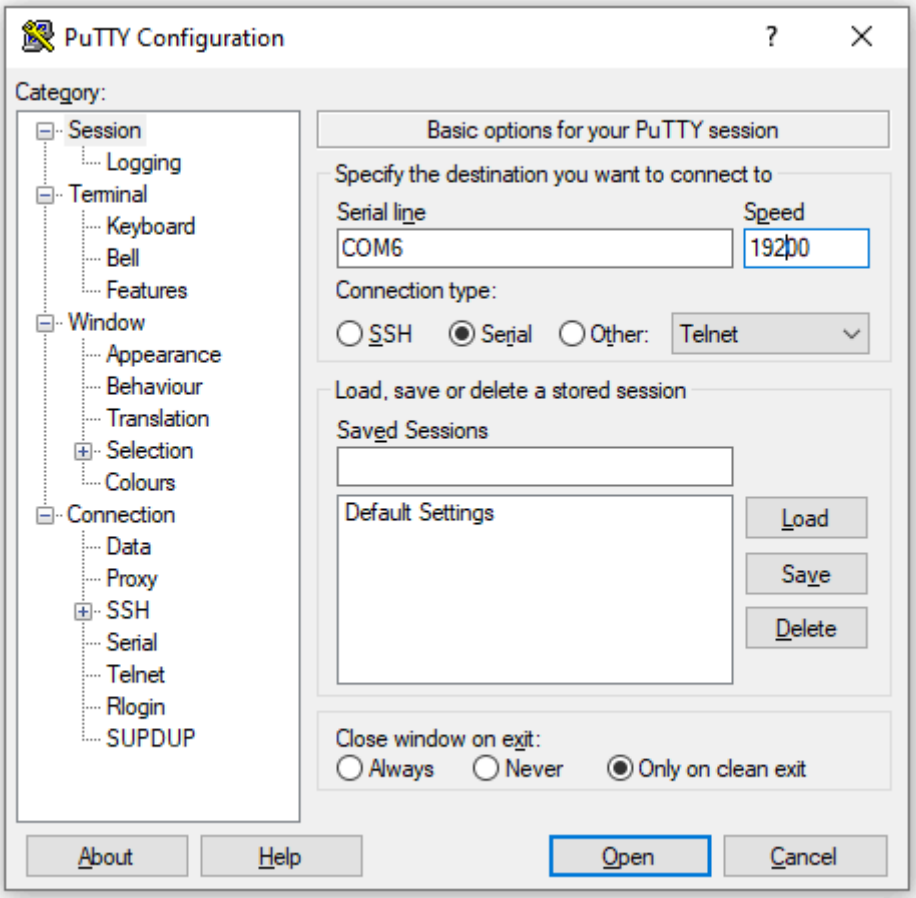


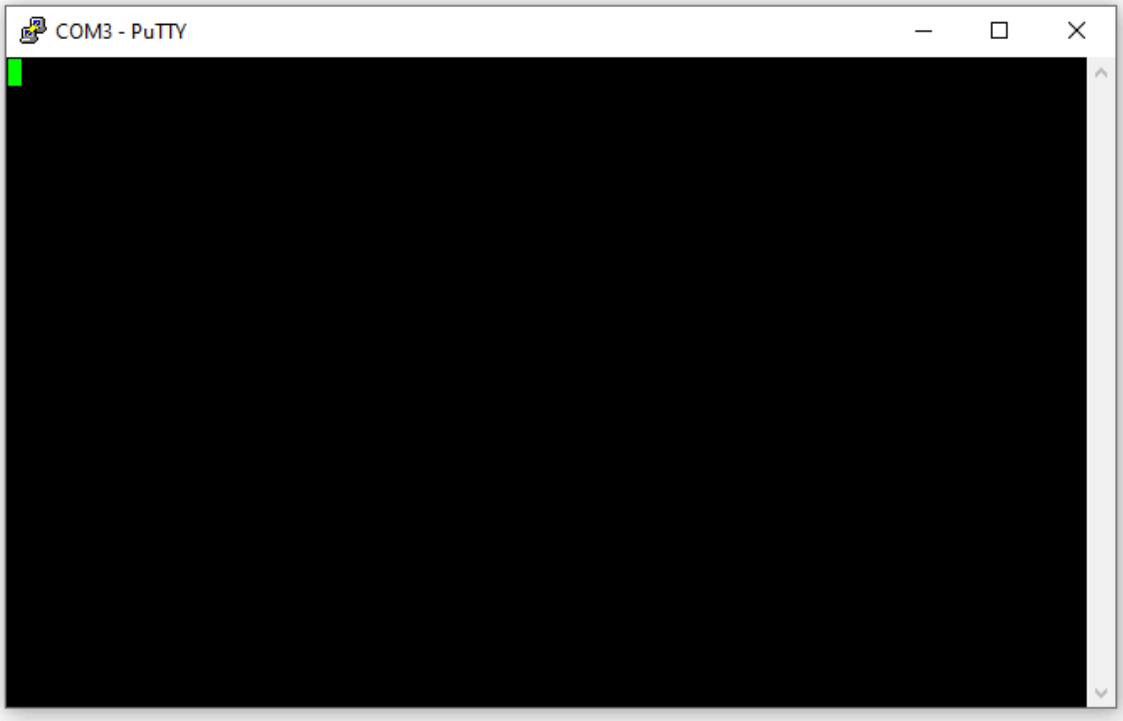


3. COMUNICACIÓN CON LA PC

Se utilizó un conversor USB-UART para establecer la conexión entre la PC y la placa.




PuTTY Configuration
Category:
Session
Logging
Terminal
Keyboard
Bell
Features
Window
Appearance
Behaviour
Translation
Selection
Colours
Connection
Data
Proxy
SSH
Serial
Telnet
Rlogin
SUPDUP
Basic options for your PuTTY session
Specify the destination you want to connect to
Serial line
Speed
COM6
19200
Connection type:
SSH
Serial
Other:
Telnet
Load, save or delete a stored session
Saved Sessions
Default Settings
Load
Save
Delete
Close window on exit:
Always
Never
Only on clean exit
About
Help
Open
Cancel


COM3 - PuTTY

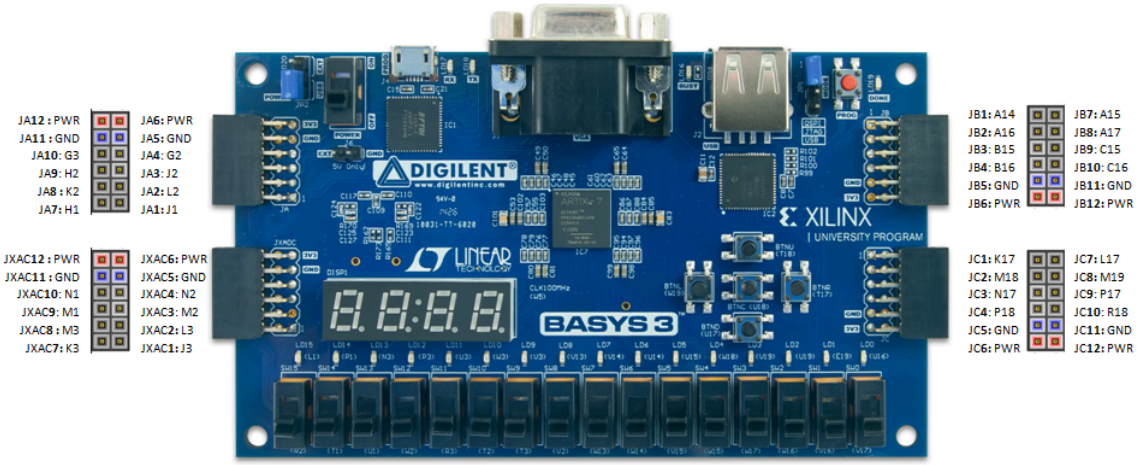
4. IMPLEMENTACIÓN

Todos los módulos se desarrollaron de manera independiente. Posteriormente, se implementó un diseño completo y estructurado en el que los datos enviados desde la PC se registran en los operandos de la ALU. Luego, el resultado se muestra en los LED de la placa y, opcionalmente, se transmite a la PC a través del transmisor para su visualización.

```
`timescale 1ns / 1ps

module top
   #(
      parameter DBIT = 8,
                SB_TICK = 16,
                DVSR = 325,
                DVSR_BIT = 10,
                FIFO_W = 4
   )
   (
    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
   );

   wire tick, rx_done_tick, tx_done_tick;
   wire [7:0] rx_data_out, w_data, tx_fifo_out;
   wire rd_uart_tick, wr_tick;
   wire tx_empty, tx_fifo_not_empty;

    reg [7:0] operandoA = 0, operandoB = 0, codigoOperacion = 0,
prev_opcode = 0;

    debounce btn1_db_unit (.clk(clk), .reset(reset), .sw(rd_uart),
.db_tick(rd_uart_tick));
    debounce btn2_db_unit (.clk(clk), .reset(reset), .sw(wr_uart),
.db_tick(wr_tick));

        mod_m_counter  #(.M(DVSR),  .N(DVSR_BIT))  baud_gen_unit
(.clk(clk), .reset(reset), .max_tick(tick));

       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   #(.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), .r_data(r_data));

   localparam ALU_DATA_A_OP    = 8'b01100001;
   localparam ALU_DATA_B_OP    = 8'b01100010;
   localparam ALU_OPERATOR_OP  = 8'b01100011;

   always @(posedge clk or posedge reset) begin
       if (reset) begin
           operandoA <= 0;
           operandoB <= 0;
           codigoOperacion <= 0;
           prev_opcode <= 0;
       end else if (!rx_empty && rd_uart_tick) begin
            if (r_data == ALU_DATA_A_OP || r_data == ALU_DATA_B_OP
|| r_data == ALU_OPERATOR_OP)
               prev_opcode <= r_data;
           else begin
               case (prev_opcode)
                   ALU_DATA_A_OP:    operandoA       <= r_data;
                   ALU_DATA_B_OP:    operandoB       <= r_data;
                   ALU_OPERATOR_OP:  codigoOperacion <= r_data;
               endcase
           end
       end
   end

     alu  alu_unit  (.operandoA(operandoA),  .operandoB(operandoB),
.operacion(codigoOperacion), .resultado(alu_output));

        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   #(.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));

   assign w_data = alu_output;
   assign tx_fifo_not_empty = ~tx_empty;

endmodule
```

Basys3: Pmod Pin-Out Diagram



https://github.com/Digilent/digilent-xdc/blob/master/Basys-3-Master.xdc

```
## 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]

## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {r_data[0]}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {r_data[1]}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {r_data[2]}]
set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {r_data[3]}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {r_data[4]}]
set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {r_data[5]}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {r_data[6]}]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {r_data[7]}]
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {alu_output[0]}]
set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {alu_output[1]}]
set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {alu_output[2]}]
set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {alu_output[3]}]
set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {alu_output[4]}]
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {alu_output[5]}]
set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {alu_output[6]}]
set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {alu_output[7]}]

##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]

##Pmod Header JB
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
#set_property -dict { PACKAGE_PIN A17   IOSTANDARD LVCMOS33 } [get_ports {JB[5]}];#Sch name = JB8
#set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports {JB[6]}];#Sch name = JB9
#set_property -dict { PACKAGE_PIN C16   IOSTANDARD LVCMOS33 } [get_ports {JB[7]}];#Sch name = JB10
```

5. REPOSITORIO DE GITHUB

https://github.com/nachoborgatello/uart_tp2

6. REFERENCIAS

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