Práctica de laboratorio: UART

Jorge A. Arbach Lizio; Bianca A. Fraga

Facultad de Ciencias Exactas, Físicas y Naturales

Arquitectura de Computadoras

Octubre, 2025

jorge.arbach@mi.unc.edu.ar

bianca.fraga@mi.unc.edu.ar

INTRODUCCIÓN

El presente trabajo práctico tiene como objetivo implementar y analizar un sistema digital que permite establecer comunicación serie mediante el protocolo UART, integrando la Unidad Aritmético-Lógica (ALU) desarrollada en el Trabajo Práctico N°1.

A partir del diseño de la ALU previa, en esta nueva etapa se incorpora una interfaz de comunicación, la cual posibilita enviar y recibir datos desde la FPGA hacia un dispositivo externo. Los datos, que incluyen los operandos y el código de operación (OpCode), son recibidos en formato serial a través de la línea de RX, procesados internamente por la ALU, y finalmente el resultado de la operación se transmite nuevamente por la línea de TX.

El diseño fue desarrollado en Verilog HDL, aplicando máquinas de estado finitas (FSM) en los módulos transmisor y receptor, y un generador de baud rate para sincronizar la comunicación. Todos los módulos utilizan reset síncrono para garantizar una inicialización controlada.

El desarrollo de este trabajo permitió afianzar los conceptos de máquinas de estado finitas (FSM), sincronismo de señales y comunicación serie, comprendiendo la interacción entre los distintos bloques de hardware dentro de un sistema digital programable.

DESARROLLO

El diseño se abordó de forma modular, dividiendo el sistema en bloques funcionales independientes que interactúan entre sí mediante señales de control y datos. Esta estructura facilita la claridad del diseño, la reutilización de componentes y la verificación individual de cada módulo, asegurando un funcionamiento ordenado y escalable del sistema.

1. ENFOQUE MODULAR

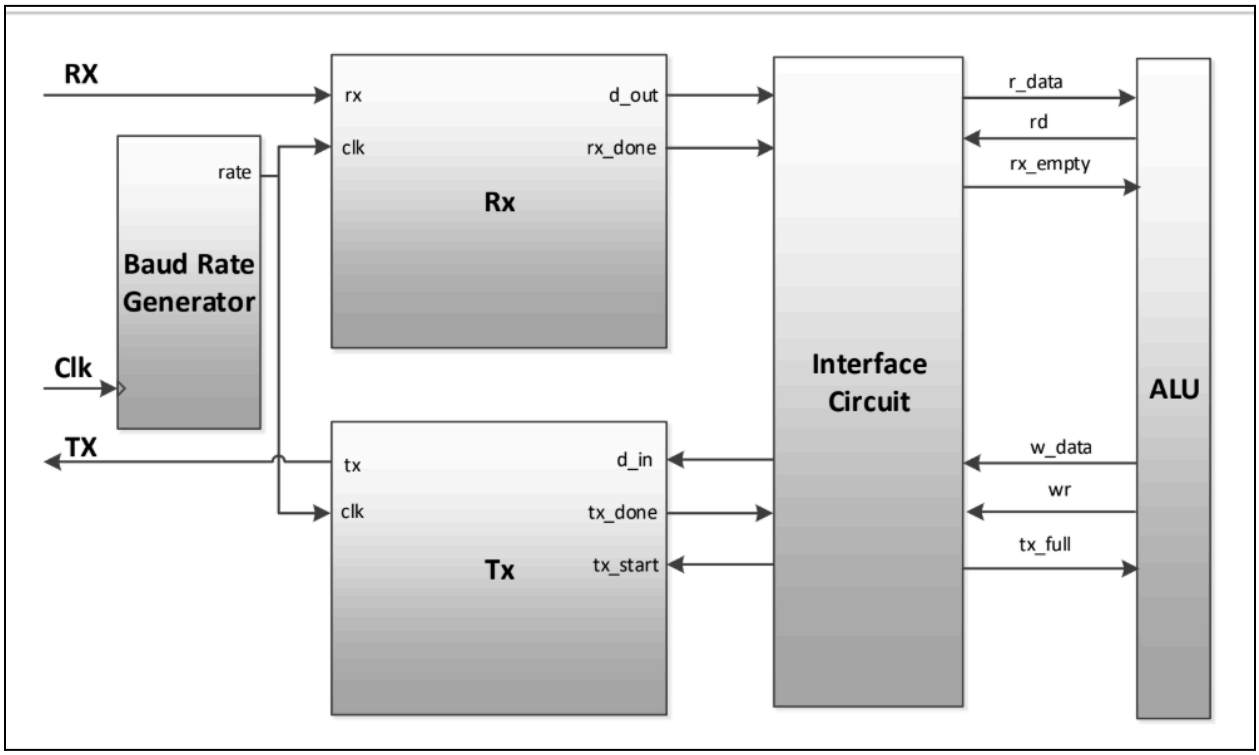

Figura 1

Como se observa en la figura, el diseño está compuesto por los siguientes bloques principales:

- **Baud Rate Generator:** genera los pulsos de temporización (*rate*) que sincronizan la transmisión y recepción de datos UART.
- **UART Receiver (Rx):** recibe los bits seriales desde la línea **RX**, los agrupa en un byte y genera una señal de finalización (*rx_done*).

- **UART Transmitter (Tx):** envía los datos serialmente por la línea **TX** cuando recibe la orden de transmisión (*tx_start*).
- **Interface Circuit:** coordina el flujo de datos entre la **UART** y la **ALU**, gestionando la lectura de operandos, la ejecución de la operación y el envío del resultado.
- **ALU**: realiza las operaciones aritmético-lógicas (ADD, SUB, AND, OR, XOR, SRA, SRL, NOR) a partir de los datos recibidos.

Este enfoque modular facilita la ***verificación individual*** de cada bloque mediante simulación, favorece la ***escalabilidad del sistema*** y permite realizar futuras modificaciones sin alterar la estructura general.

De esta manera, cada componente puede ser analizado y validado por separado antes de la integración final.

2. MÓDULOS

2.1 *baud_gen* - Generador de ticks de Baud Rate

Este módulo genera los ***pulsos de sincronismo*** necesarios para el funcionamiento del transmisor y receptor UART.

A partir del reloj principal de ***50 MHz***, produce un pulso (*tick*) cada ***fclk / (baud × 16)*** ciclos, permitiendo un muestreo de 16 veces por bit.

```
//Se ejecuta en cada flanco positivo del reloj (clk)
always @(posedge clk) begin
    if (rst) begin
        // Si se resetea, el contador y tick se reinician
        counter <= 0;
        tick <= 0;
    end else begin
        // Cuando el contador llega al máximo, se reinicia y emite un tick
        if (counter == COUNT_MAX - 1) begin
            counter <= 0;
            tick <= 1;
        end else begin
            // Mientras tanto, sigue contando
            counter <= counter + 1;
            tick <= 0; // tick apagado (solo se enciende un ciclo)
        end
```

Figura 2

Cuando el contador interno alcanza el valor máximo (***COUNT_MAX - 1***), se reinicia y genera un tick de un solo ciclo, que luego utilizan los módulos ***uart_rx*** y ***uart_tx*** para sincronizar la transmisión y recepción de bits.

2.2 *uart_rx* - Receptor UART

El receptor ***uart_rx*** implementa una ***máquina de estados finita (FSM)*** encargada de detectar el bit de inicio, muestrear los bits de datos y verificar el bit de stop antes de entregar el byte recibido.
Sus estados principales son: **IDLE, START, DATA, STOP y DONE.**

```
// Lógica del próximo estado
always @(*)
begin
    state_next = state_reg;
    s_next = s_reg;
    n_next = n_reg;
    b_next = b_reg;
    rx_done = 1'b0;
    case (state_reg)
        S_IDLE:
            if (~rx)
            begin
                s_next = 0;
                state_next = S_START;
            end
        S_START:
            if (tick)
                if (OVERSAMPLING/2 - 1)     // Cuando llega a 7 reseteamos los ticks a 0 para poder leer a la mitad de los bits a los 16 ticks
                begin
                    s_next = 0;
                    n_next = 0;
                    state_next = S_DATA;
                end
                else
                    s_next = s_reg + 1;
```

Figura 3

```
        S_DATA:
            if (tick)
                if (s_reg == (OVERSAMPLING -1))     // Nos encontramos en la mitad del bit
                begin
                    s_next = 0; // Reinciamos los ticks
                    b_next = {rx, b_reg[DATA_BITS-1:1]};    // Shift a la derecha
                    if (n_reg == (DATA_BITS-1))
                        state_next = S_STOP;    // Si ya leimos el ultimo bit, paramos
                    else
                        n_next = n_reg + 1;     // Continuamos para leer el próximo bit
                end
                else
                    s_next = s_reg + 1;
        S_STOP:
            if (tick)
                if (s_reg == (OVERSAMPLING -1))
                begin
                    rx_done = 1'b1;
                    state_next = S_IDLE;
                end
                else
                    s_next = s_reg + 1;
        default:
            state_next = S_IDLE;
    endcase
end
```

Figura 4

- En **IDLE**, el sistema espera que la línea ***rx*** pase a nivel bajo (inicio de trama).
- En **START**, se confirma la validez del inicio de transmisión.
- En **DATA**, se registran los bits entrantes en orden LSB primero.
- En **STOP**, se verifica el bit de parada.
- Finalmente, en **DONE**, se activa la señal ***rx_done***, indicando que el byte recibido es válido.

2.3 *uart_tx* - Transmisor UART

El transmisor ***uart_tx*** también se implementa como una ***FSM*** que gestiona la secuencia de transmisión: **IDLE, START, DATA, STOP y DONE.**
Envía de forma serial los bits de un byte almacenado en ***din***, comenzando con el bit de inicio (0), seguido por los bits de datos (LSB primero) y finalizando con el bit de stop (1).

```
//Lógica del siguiente estado
always@(*)begin
    state_next=state_reg;
    s_next=s_reg;
    n_next=n_reg;
    //tx_done = 1'b0; // tx_done ahora es síncrono
    tx_next=tx;//Mantener valor actual por defecto

    case(state_reg)
        S_IDLE:begin
            tx_next= 1;//Línea en alto
            if(tx_start)begin
                s_next= 0;
                n_next= 0;
                state_next=S_START;
                tx_next= 0;    //Bit de inicio (abajo)
            end
        end

        S_START:begin
            tx_next= 0;    //Mantenemos bit de inicio
            if(tick)begin
                if(s_reg==(OVERSAMPLING- 1))begin
                    s_next= 0;
                    state_next=S_DATA;
                    tx_next=data_reg[0];  //Primer bit de datos
                end else
                    s_next = s_reg +1;
            end
        end
```

```
S_DATA:begin
    tx_next=data_reg[n_reg];  //Mantenemos el bit actual
    if(tick)begin
        if(s_reg==(OVERSAMPLING- 1))begin
            s_next= 0;
            if(n_reg==(DATA_BITS- 1))begin
                state_next=S_STOP;
                tx_next= 1;//Bit de parada
            end else begin
                n_next=n_reg+ 1;
                tx_next=data_reg[n_reg+ 1];  //Siguiente bit
            end
        end else
            s_next = s_reg +1;
    end
end

S_STOP:begin
    tx_next= 1;    //Bit de parada (alto)
    if(tick)begin
        if(s_reg==(OVERSAMPLING- 1))begin
            //tx_done se genera en el flanco de reloj para que la FSM lo detecte
            state_next=S_IDLE;
        end else
            s_next = s_reg +1;
    end
end

default:begin
    state_next=S_IDLE;
    tx_next= 1;
```

Figura 5-6

Además, el módulo utiliza contadores internos (***s_reg***, ***n_reg***) para controlar el tiempo de cada bit de acuerdo al baud rate, activando ***tx_done*** al finalizar la transmisión completa.

2.4 alu - Unidad Aritmético-Lógica

La ***ALU*** realiza las operaciones definidas en el TP1 según el código de operación recibido. Entre las operaciones implementadas se encuentran: ***ADD (100000), SUB (100010), AND (100100), OR (100101), XOR (100110), NOR (100111), SRA (000011), SRL (000010).***
El resultado de la operación es entregado a la interfaz para ser enviado nuevamente por la UART.

```
// códigos de operación según el cuadro
localparam ADD = 6'b100000;
localparam SUB = 6'b100010;
localparam AND_ = 6'b100100;
localparam OR_  = 6'b100101;
localparam XOR_ = 6'b100110;
localparam SRA  = 6'b000011;
localparam SRL  = 6'b000010;
localparam NOR  = 6'b100111;
localparam SHIFT_BITS = $clog2(N_data);     // cantidad de bits necesarios para hacer l
always @(*) begin        // valores por defecto
    S     = 0;
    carry = 0;
    zero  = 0;

    case (OpCode)
        ADD:  {carry, S} = A + B;             // suma
        SUB:  {carry, S} = A - B;             // resta (no es un carry sino un borrow)
        AND_: S = A & B;                      // and
        OR_:  S = A | B;                      // or
        XOR_: S = A ^ B;                      // xor
        SRA:  S = $signed(A) >>> B[SHIFT_BITS-1:0];      // shift der aritmético
        SRL:  S = A >> B[SHIFT_BITS-1:0];                // shift der logico
        NOR:  S = ~(A | B);                   // nor
        default: S = 8'b00000000;             // por las dudas
    endcase

    // flag de zero
    zero = (S == 0);
```

Figura 7

2.5 interface - Coordinador de comunicación

Este módulo controla el ***flujo de datos entre la UART y la ALU***.
Recibe los tres bytes correspondientes a ***A***, ***B*** y el código de operación (***OpCode***), habilita el procesamiento en la ALU y luego transmite el resultado.

En este bloque se implementa una ***máquina de estados finita (FSM)*** que actúa como controlador de la comunicación entre la UART y la ALU.

- Los estados ***S_GET_A, S_GET_B y S_GET_OP*** controlan la recepción secuencial de los operandos y el código de operación desde el PC.
- Luego, la ***FSM*** pasa a ***S_SEND_RESULT***, donde se envía el resultado calculado por la ALU, y finalmente a ***S_SEND_FLAGS***, donde se transmiten las banderas carry y zero.
- Entre cada envío o recepción, los estados ***S_WAIT_RESULT*** y ***S_WAIT_FLAGS*** garantizan la correcta sincronización con las señales ***rx_done*** y ***tx_done***.

Este diseño asegura una transferencia ordenada y coordinada de los datos sin pérdida ni superposición, cumpliendo el principio de control síncrono mediante una FSM.

```
case (state)
    S_GET_A: begin
        if (rx_done) begin
            alu_a <= rx_data;
            state <= S_GET_B;
        end
    end

    S_GET_B: begin
        if (rx_done) begin
            alu_b <= rx_data;
            state <= S_GET_OP;
        end
    end

    S_GET_OP: begin
        if (rx_done) begin
            alu_opcode <= rx_data[N_OP-1:0];
            // Como la ALU es combinacional, pasamos directamente al estado para mandar el resultado
            state <= S_SEND_RESULT;
        end
    end

    S_SEND_RESULT: begin
        tx_data <= alu_result;
        tx_start <= 1'b1;
        state <= S_WAIT_RESULT;
    end
```

Figura 8

2.6 uart_alu_top - Integración del sistema

El módulo ***uart_alu_top*** integra todos los bloques anteriores y controla la secuencia global del sistema.

Mediante una FSM interna, gestiona los estados:

- ***WAIT_A***: espera el primer operando.
- ***WAIT_B***: espera el segundo operando.
- ***WAIT_OP***: espera el código de operación.
- ***SEND_R***: envía el resultado.

```
// Instanciamos el baud rate generator
baud_gen #(
    .CLK_FREQ(CLK_FREQ),
    .BAUD_RATE(BAUD_RATE),
    .OVERSAMPLING(16)
) baud_gen_inst (
    .clk(clk),
    .rst(rst),
    .tick(tick)
);

// Instanciamos el receptor UART
uart_rx #(
    .DATA_BITS(DATA_WIDTH),
    .OVERSAMPLING(16)
) uart_rx_inst (
    .clk(clk),
    .rst(rst),
    .rx(rx),
    .tick(tick),
    .rx_done(rx_done),
    .data_out(rx_data)
);

// Instanciamos el transmisor UART
uart_tx #(
    .DATA_BITS(DATA_WIDTH),
    .OVERSAMPLING(16)
) uart_tx_inst (
    .clk(clk),
    .rst(rst),
    .tx_start(tx_start),
    .data_in(tx_data),
    .tick(tick),
    .tx(tx),
    .tx_done(tx_done)
);
```

Figura 9

Este bloque es el responsable de coordinar la comunicación entre los módulos ***UART***, la ***ALU*** y la ***interfaz***, garantizando un flujo de datos continuo y sin conflictos.

3. TESTBENCH

El proceso de verificación funcional se realizó mediante simulaciones en Vivado, aplicando ***testbenches independientes para cada módulo*** del sistema.

El objetivo fue validar tanto el comportamiento individual de los bloques (***uart_tx***, ***uart_rx***, ***interface***, ***etc.***) como la integración completa dentro del módulo ***uart_alu_top***, que conecta la ***ALU del TP1 con la interfaz UART***.

Cada testbench genera estímulos controlados, observa las respuestas y verifica la correcta transición de estados y sincronización entre transmisión, recepción y procesamiento de datos.

3.1 Testbench del transmisor UART (tb_uart_tx)

Este testbench se diseñó para comprobar el ***envío de datos seriales*** mediante la máquina de estados del transmisor (***uart_tx***).
Durante la simulación, se observa cómo la línea ***tx*** cambia de nivel según el bit de inicio, los bits de datos y el bit de parada.

Además, se valida que la señal ***tx_done*** se active al finalizar correctamente el envío de un byte.

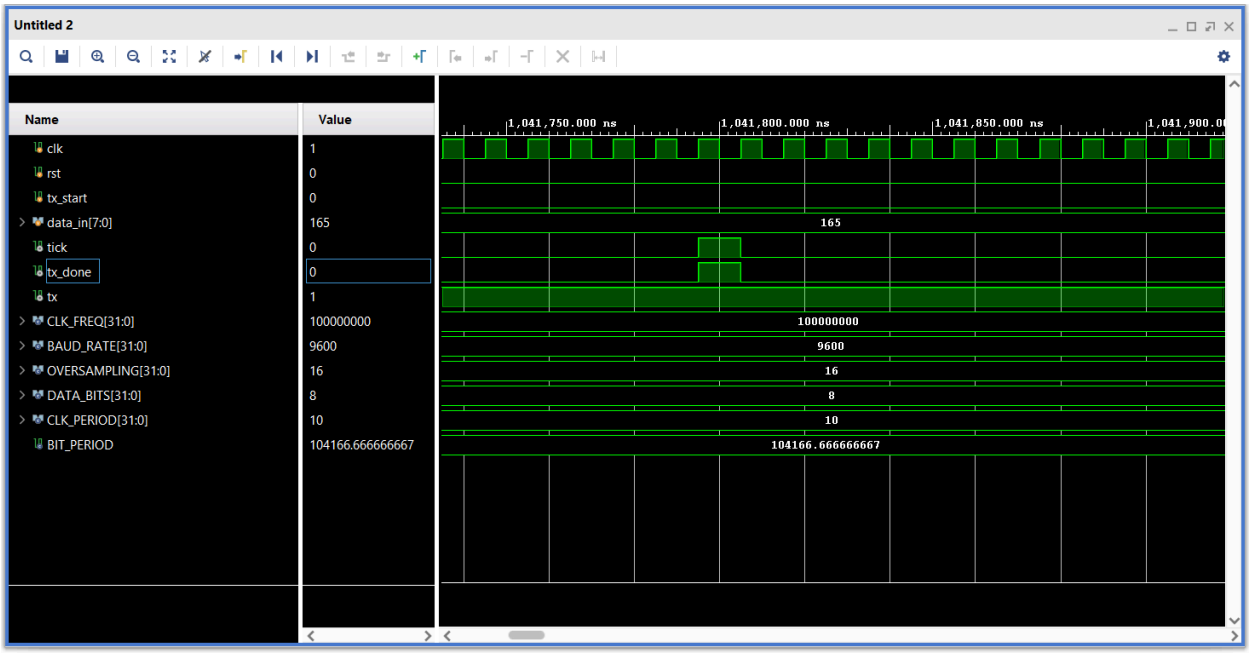

Figura 10

En la figura se observa la secuencia de transmisión generada por el módulo ***uart_tx***.

La señal ***tx_start*** inicia el envío del dato paralelo ***data_in = 165*** (equivalente a **10100101** en binario).

El módulo comienza transmitiendo un ***bit de inicio*** ('0'), seguido por los ***8 bits de datos***, y finalmente el ***bit de parada*** ('1').

La línea ***tx*** refleja claramente esta secuencia, donde cada bit permanece estable durante el período determinado por el ***baud rate*** configurado (9600 baudios con reloj de 100 MHz y sobremuestreo ×16).

La señal ***tx_done*** se activa brevemente al finalizar la transmisión del último bit, confirmando que el proceso fue completado correctamente.

3.2 Testbench del receptor UART (tb_uart_rx)

Este test verifica la ***recepción asíncrona de datos***, asegurando que los bits recibidos se reconstruyan correctamente en el bus ***data_out*** y que la señal ***rx_done*** se active al completar un byte.

Se simula una transmisión bit a bit en la línea ***rx***, sincronizada con el tick generado por ***baud_gen***.

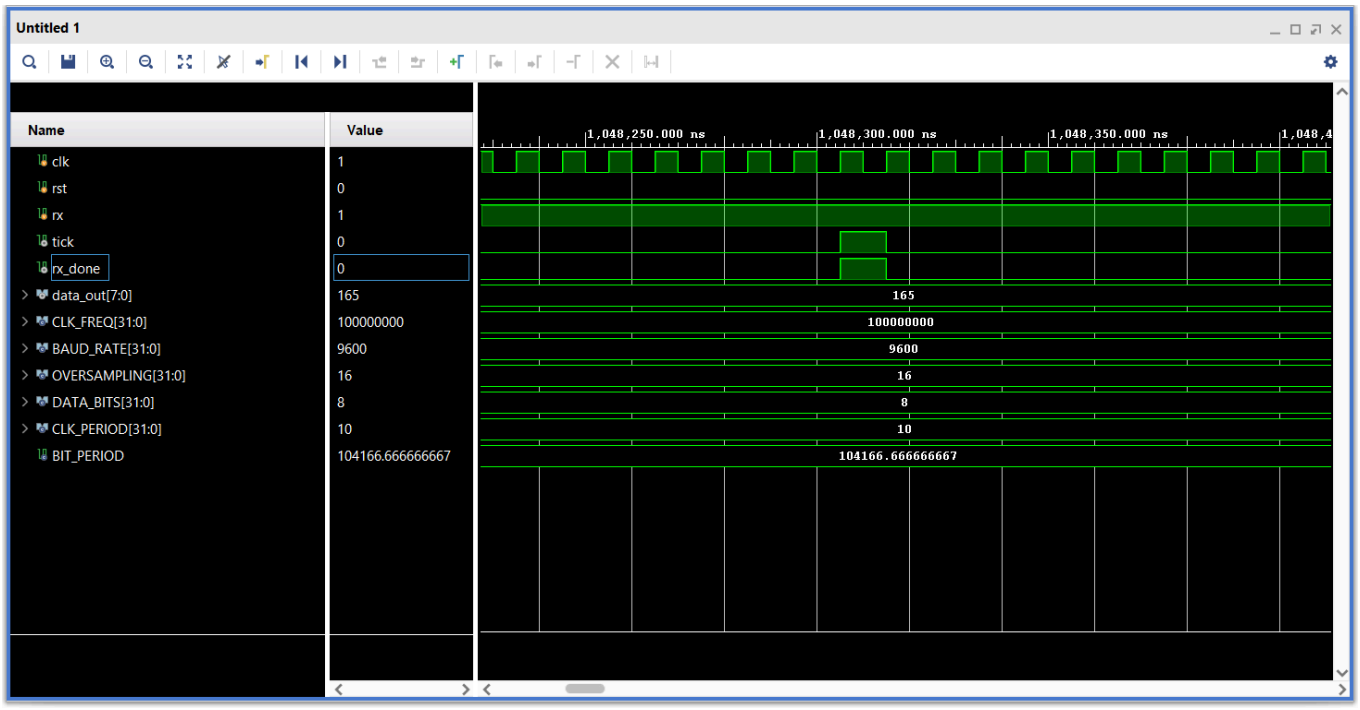

Figura 11

En la figura se observa la señal de reloj del sistema (***clk***), el reset inicial (***rst***), la línea serie de entrada (***rx***) y las señales internas del módulo, incluyendo el tick generado por el ***baud_gen***. Durante la simulación, el dato serial recibido se reconstruye correctamente en ***data_out*** (valor **165** en decimal, equivalente a **10100101** en binario), activándose la señal ***rx_done*** durante un ciclo de reloj al completarse la recepción del byte.

El valor de ***BIT_PERIOD*** y ***CLK_FREQ*** confirma que el tiempo de muestreo corresponde al baud rate configurado de **9600 baudios** con un reloj de **100 MHz**, utilizando un factor de sobremuestreo de 16, lo que garantiza la correcta sincronización entre transmisor y receptor.

3.3 Testbench de la interfaz de control (tb_interface)

En este testbench se validó la ***máquina de estados de la interfaz***, responsable de coordinar el flujo de datos entre UART y ALU.

La simulación muestra cómo se reciben los operandos ***A*** y ***B***, luego el código de operación (***opcode***), y finalmente se transmite el resultado seguido de los flags (***carry*** y ***zero***).

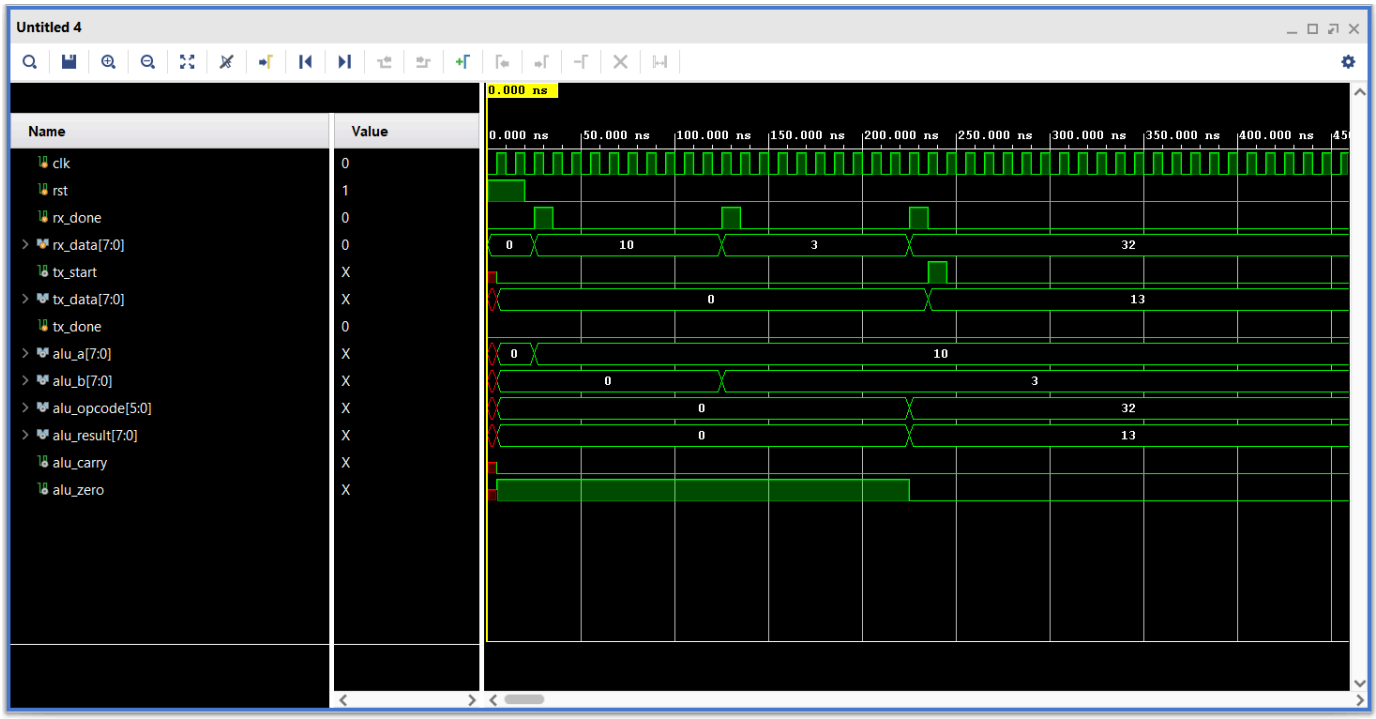

Figura 12

3.4 Testbench del modo loopback (tb_uart_loopback)

Este test conecta internamente el transmisor y el receptor UART, permitiendo ***verificar la comunicación completa TX → RX*** dentro del mismo sistema.
El objetivo fue comprobar la correcta reconstrucción del byte transmitido, validando los tiempos del baud rate y la consistencia entre ambos módulos.

3.5 Testbench del sistema completo (tb_uart_alu_top)

Finalmente, se implementó la verificación integral del ***sistema UART + ALU***, donde el flujo completo incluye:

1. Recepción de los operandos y operación vía UART.
2. Procesamiento en la ALU.
3. Envío del resultado y los flags nuevamente por UART.

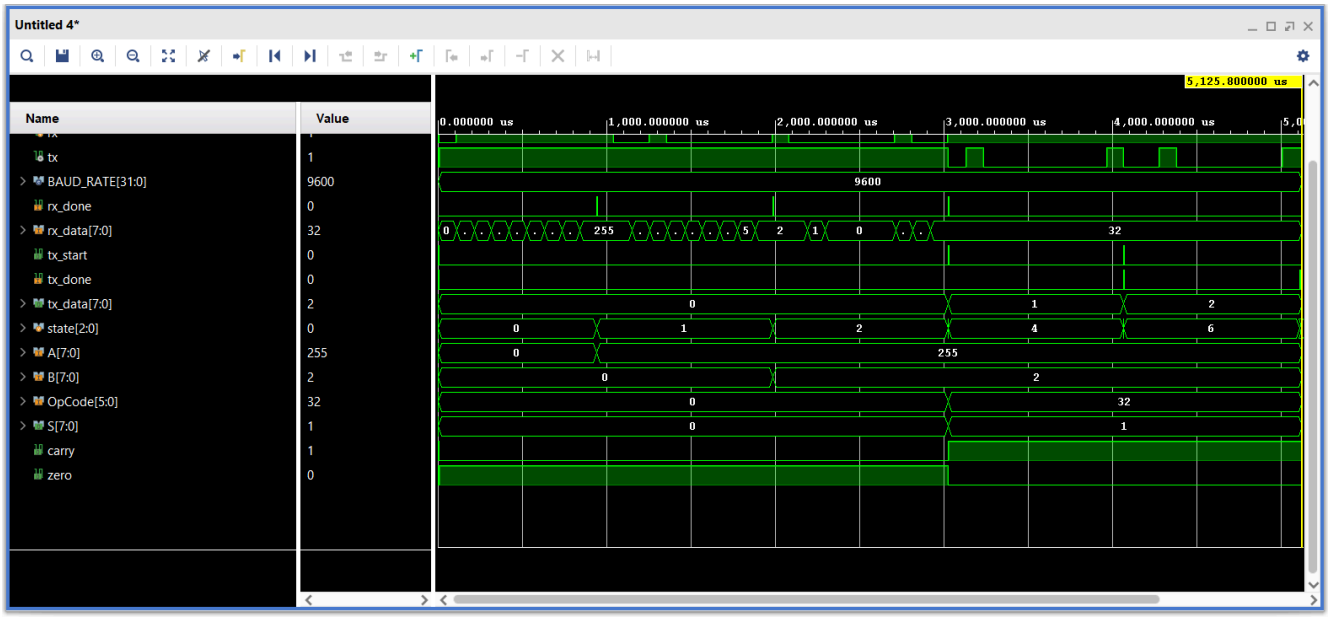

Figura 13

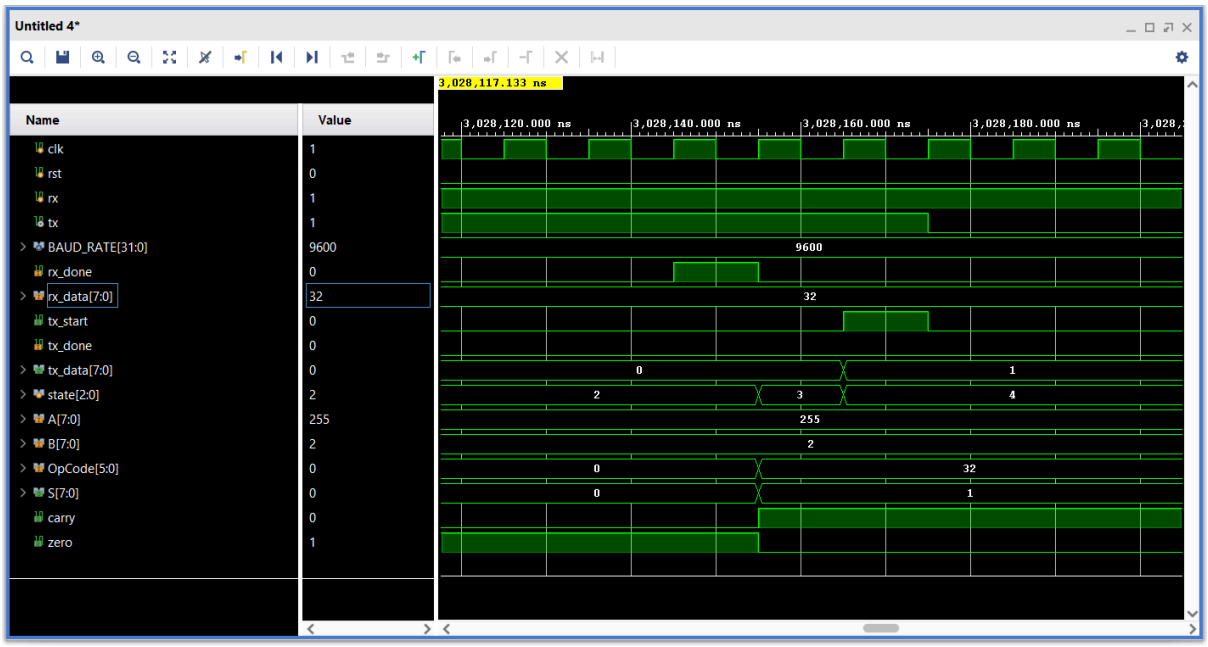

Figura 14

Se comparan dos instantes de simulación digital donde se observa la interacción entre señales de control, datos y estados. En ambos casos se verifica la correcta ***recepción*** y ***transmisión*** UART, cambios en el registro de datos, activación de flags (***carry***, ***zero***) y transición de estados según el ***OpCode***. Las imágenes permiten validar el comportamiento esperado del sistema y justificar cada etapa del flujo de ejecución.'

Se puede apreciar en la Figura 14, que cuando el pulso de rx_done termina, la interfaz carga adecuadamente el operando en la ALU, en este caso al estar en el estado 2 (WAIT_OP), carga el valor en OpCode, ya obteniendo el resultado', y pasa al estado 4 (), iniciando la transmisión con el pulso de tx_start.

4. INTERFAZ DE COMUNICACIÓN Y VERIFICACIÓN

Con el fin de comprobar el correcto funcionamiento del sistema completo (UART + ALU) implementado en la FPGA, se desarrolló un ***script en lenguaje Python*** que permite establecer una comunicación serie con la placa a través del ***puerto UART***.

Esta herramienta se utilizó tanto para el envío de datos de prueba como para la recepción y visualización de los resultados calculados por la ALU, brindando un entorno de verificación más práctico y confiable.

4.1 Objetivo del script

El objetivo principal del programa fue ***interactuar de manera directa con el hardware***, enviando los operandos y el código de operación (OpCode) hacia la FPGA y recibiendo la respuesta correspondiente.

De esta forma, fue posible validar las operaciones aritméticas y lógicas de la ALU en tiempo real, confirmando el correcto flujo de comunicación y procesamiento entre los módulos UART y ALU.

El script permite seleccionar una de las ocho operaciones definidas en la ALU (***ADD***, ***SUB***, ***AND***, ***OR***, ***XOR***, ***SRA***, ***SRL***, ***NOR***), ingresando los operandos A y B desde la consola.
Una vez enviados los datos, el sistema retorna el resultado y las banderas de estado (***carry*** y ***zero***), que se muestran en pantalla de manera legible tanto en formato decimal como hexadecimal y binario.

4.2 Estructura y funcionamiento general

El programa fue estructurado de forma modular, de manera similar al diseño hardware del sistema.
Las funciones principales son:

- ***OPERATIONS***: tabla de operaciones disponibles en la ALU. Contiene para cada operación su nombre, símbolo y código (OpCode).
  Ejemplo: ADD → 0x20, SUB → 0x22, etc.

```python
# Operaciones disponibles en la ALU
OPERATIONS = {
    '1': {'name': 'ADD', 'opcode': 0x20, 'symbol': '+'},
    '2': {'name': 'SUB', 'opcode': 0x22, 'symbol': '-'},
    '3': {'name': 'AND', 'opcode': 0x24, 'symbol': '&'},
    '4': {'name': 'OR', 'opcode': 0x25, 'symbol': '|'},
    '5': {'name': 'XOR', 'opcode': 0x26, 'symbol': '^'},
    '6': {'name': 'SRA', 'opcode': 0x03, 'symbol': '>>'},
    '7': {'name': 'SRL', 'opcode': 0x02, 'symbol': '>>>'},
    '8': {'name': 'NOR', 'opcode': 0x27, 'symbol': 'NOR'},
}
```

Figura 15

- ***print_menu()***: presenta el menú de operaciones disponible en consola, permitiendo al usuario seleccionar cuál desea ejecutar.

- ***get_operand()***: solicita al usuario el ingreso de los operandos A y B. Admite valores en decimal o hexadecimal (por ejemplo, ***25*** o ***0x19***), validando que se encuentren dentro del rango permitido (0–255).

- ***send_and_receive()***: se encarga de la comunicación efectiva por el puerto serie. Envía tres bytes consecutivos:

1. Operando A
2. Operando B
3. Código de operación (***OpCode***)

   Luego, espera la respuesta de la FPGA, la cual consiste en dos bytes:
4. Resultado de la operación
5. Flags (***carry*** y ***zero***)

Para asegurar una comunicación estable, la función implementa una política de *timeout*, gestión de búferes (***reset_input_buffer***, ***reset_output_buffer***) y espera activa hasta recibir todos los datos.

```python
def send_and_receive(ser, a, b, opcode):
    """Envía los operandos y opcode, y recibe la respuesta"""
    # Limpiar buffers y enviar
    ser.reset_input_buffer()
    ser.reset_output_buffer()
    ser.write(bytes([a, b, opcode]))
    ser.flush()

    time.sleep(0.01)

    # Lectura
    resp = bytearray()
    start = time.time()
    deadline = start + 2.0
    last_rx = start
```

Figura 16

- ***display_result()***: interpreta y muestra los resultados recibidos. Presenta los valores en distintos formatos (decimal, hexadecimal y binario), además de los flags de estado. En el caso particular de la resta (***SUB***), el script identifica si hubo ***borrow*** (préstamo) y realiza la conversión al complemento a dos para mostrar el resultado negativo correctamente.

```python
def display_result(a, b, op_info, resp):
    """Muestra los resultados de la operación"""
    print("\n" + "-"*50)
    print(f"Operación: {a} (0x{a:02X}) {op_info['symbol']} {b} (0x{b:02X})")
    print("-"*50)

    res = resp[0] if len(resp) >= 1 else None
    flags = resp[1] if len(resp) >= 2 else None

    if res is not None:
        print(f"Resultado: {res} (0x{res:02X}) [binario: {res:08b}]")
    else:
        print("Resultado: No recibido")

    if flags is not None:
        carry = (flags >> 1) & 1
        zero = flags & 1
        print(f"Flags: 0x{flags:02X}")
        print(f"  - Carry/Borrow: {carry}")
        print(f"  - Zero:  {zero}")
        if carry and op_info['name'] == 'SUB':
            # Si la operacion era una resta y el carry está activo, indica borrow por lo que debemos mostrar el resultado como negativo
            print()
            print("Nota: Resultado negativo (underflow en resta)")
            res = (~res & 0xFF) + 0x01  # complemento a dos
            print(f"  - Resultado correcto: -{res} (0x{res:02X}) [binario: {res:08b}]")
    else:
        print("Flags: No recibidas")
```

Figura 17

- ***main()***: contiene el flujo principal del programa. Configura el puerto serie (por ejemplo, ***COM8*** a ***9600 baudios***), establece la conexión con la FPGA y administra la interacción con el usuario. Permite realizar múltiples operaciones de forma consecutiva hasta que el usuario decida finalizar la ejecución.

4.3 Funcionamiento en conjunto con el sistema UART-ALU

El script interactúa directamente con el módulo ***uart_alu_top*** implementado en la FPGA. Cada dato transmitido desde Python se recibe en la interfaz UART de la FPGA, donde los módulos ***uart_rx*** y ***interface*** se encargan de dirigir los operandos y el código de operación hacia la ALU.
Una vez calculado el resultado, la interfaz envía de vuelta los datos a través del ***uart_tx***, que son capturados por el script para su visualización.

Este esquema de prueba permitió verificar en condiciones reales de operación:
- La correcta transmisión y recepción de datos por UART a 9600 baudios.
- El procesamiento inmediato en la ALU de los operandos recibidos.
- La integridad de los resultados y flags retornados al entorno de Python.

5. RESULTADOS

En las simulaciones realizadas con ***Vivado***, se comprobó el correcto funcionamiento de la comunicación UART a ***9600 baudios*** con un reloj de ***100 MHz***.
Las señales observadas en las formas de onda mostraron la secuencia completa: recepción de los bytes A, B y OpCode, ejecución de la operación correspondiente y transmisión del resultado por la línea TX.

Aunque la simulación mostraba un funcionamiento correcto, en la implementación sobre placa se detectó una falla: solo se recibía el byte de resultado, pero no el segundo byte que contenía las flags (***carry***, ***zero***). Además, tras cada operación era necesario resetear el sistema, lo que indicaba que la ***FSM*** quedaba bloqueada en el estado ***S_WAIT_RESULT***.

El análisis reveló que el pulso ***tx_done***, generado en lógica combinacional dentro de ***uart_tx***, no era reconocido por la FSM de ***interface***, que lo muestrea en el flanco de reloj. Al no estar sincronizado, el pulso podía perderse entre flancos, impidiendo el avance de estado.

Para resolverlo, se modificó ***tx_done*** para que se genere de forma secuencial en el flanco positivo de ***clk***, garantizando su detección por la FSM. Además, se registró ***data_in*** al inicio de la transmisión, evitando cambios durante el envío.

Tras aplicar la corrección en la generación de ***tx_done***, el sistema comenzó a funcionar correctamente en la placa. Se reciben ambos bytes (***resultado*** y ***flags***) sin necesidad de reiniciar, y la ***FSM*** avanza de forma estable desde ***S_WAIT_RESULT*** al estado inicial. Esto confirma que el pulso sincronizado permite una detección fiable por parte de la FSM.

```
¿Realizar otra operación? (s/n): s

================================================
ALU SERIAL - Operaciones Disponibles
================================================
1. ADD    (+)
2. SUB    (-)
3. AND    (&)
4. OR     (|)
5. XOR    (^)
6. SRA    (>>)
7. SRL    (>>>)
8. NOR    (NOR)
0. Salir
================================================

Seleccione una operación: 7

--- Operación: SRL ---
Ingrese operando A (0-255 o hex 0x00-0xFF): 0x04
Ingrese operando B (0-255 o hex 0x00-0xFF): 3

Enviando datos...

------------------------------------------------
Operación: 4 (0x04) >>> 3 (0x03)
------------------------------------------------
Resultado: 0 (0x00) [binario: 00000000]
Flags: 0x01
  - Carry/Borrow: 0
  - Zero:  1
------------------------------------------------

¿Realizar otra operación? (s/n):
```

Figura 18

```
PS C:\Users\jorge\OneDrive\Escritorio\OneDrive - mi.unc.edu.ar\UNC\5to año\2do semestre\Arqui\P
ráctico\TPs-Arquitectura> py -u .\TP2\scripts\serial_script.py

Conectando a COM8 a 9600 baud...
Conexión establecida ✓

================================================
ALU SERIAL - Operaciones Disponibles
================================================
1. ADD    (+)
2. SUB    (-)
3. AND    (&)
4. OR     (|)
5. XOR    (^)
6. SRA    (>>)
7. SRL    (>>>)
8. NOR    (NOR)
0. Salir
================================================

Seleccione una operación: 2

--- Operación: SUB ---
Ingrese operando A (0-255 o hex 0x00-0xFF): 124
Ingrese operando B (0-255 o hex 0x00-0xFF): 156

Enviando datos...

------------------------------------------------
Operación: 124 (0x7C) - 156 (0x9C)
------------------------------------------------
Resultado: 224 (0xE0) [binario: 11100000]
Flags: 0x02
  - Carry/Borrow: 1
  - Zero:  0

Nota: Resultado negativo (underflow en resta)
  - Resultado correcto: -32 (0x20) [binario: 00100000]
------------------------------------------------

¿Realizar otra operación? (s/n): s
```

Figura 19

CONCLUSIÓN

El desarrollo de este trabajo permitió integrar los conocimientos adquiridos en el diseño digital, combinando la lógica aritmético-lógica (ALU) con una interfaz de comunicación UART implementada en Verilog HDL. A través de la simulación y la prueba en la FPGA, se comprobó el correcto funcionamiento del sistema, evidenciando la transmisión y recepción de datos de manera asíncrona y la ejecución precisa de las operaciones definidas.

El trabajo destacó la importancia del diseño modular, la sincronización mediante FSM y el uso del reset síncrono para garantizar un comportamiento estable y predecible del hardware. En conjunto, la experiencia fortaleció la comprensión sobre cómo los distintos bloques digitales pueden interactuar coordinadamente para formar sistemas más complejos y funcionales.