Contador ascendente/descendente controlado por UART.

Autor: John Aníbal Rivera Burgos.

Profesor: Nicolás Alvarez.

Institución: Universidad de Buenos Aires.

Octubre 2025.

Índice

Este informe presenta el diseño, la simulación y la implementación de un sistema UART con un contador ascendente y descendente en la FPGA Arty Z7-10. El desarrollo utiliza VHDL en Vivado, con módulos `uart_rx` y `uart_tx`, un contador de cuatro bits y lógica de control de comandos. Se evalúan utilización de recursos, temporización y consumo de potencia. El sistema demuestra operación estable a 125 MHz y comunicación confiable a 115 200 bps en formato 8N1.

1. Introducción.

1.1. Contexto y motivación.

Interactuar con hardware por consola serie ofrece una forma directa y confiable de validar diseños RTL. UART es un estándar disponible en casi todos los sistemas embebidos y computadoras, por lo que simplifica las pruebas. Un contador controlado por UART resulta un ejemplo pedagógico útil. Integra comunicación, control secuencial y temporización en un caso concreto. Permite ver resultados inmediatos en una terminal serie y en los LEDs de la placa Arty Z7.

1.2. Problema y requisitos.

Problema: diseñar un sistema que reciba comandos por UART y controle un contador ascendente y descendente, para mostrar el estado en LEDs y monitor serial.

1.2.1. Requisitos funcionales.

- Recibir y procesar comandos desde la interfaz UART.
  - `'+'`: incremento del contador.
  - `'-'`: decremento del contador.
  - `'s'`: activar modo automático.
  - `'p'`: pausar modo automático.
  - `'z'`: reiniciar.
- Mostrar el valor del conteo en los `leds[3:0]` y enviar el valor actual como eco en formato hexadecimal por la línea UART TX.
- Ejecutar conteo automático a una frecuencia de 10 Hz cuando el modo automático (`s`) esté activo.

1.3. Alcances y limitaciones.

El diseño opera con un único reloj de 125 MHz. El protocolo UART es 8N1 sin paridad ni control de flujo. El contador es de cuatro bits con rango 0–15.

1.4. Objetivos.

- Implementar un receptor y un transmisor UART 8N1 a 115 200 bps.
- Integrar un contador ascendente y descendente de cuatro bits con control por comandos.
- Verificar funcionalidad por simulación y pruebas en hardware con eco en consola.

2. Plataforma y Herramientas.

2.1. Hardware.

- Placa **Arty Z7-10** basada en Zynq-7000 `xc7z010clg400-1`.
- Conector PMOD JA para driver UART externo.
- Cuatro LEDs de usuario que ya estan en la placa.
- Pulsador de reinicio.
- Adaptador USB–UART de 3,3 V hacia el PC.

2.2. Software.

- **Vivado 2024.1** para síntesis e implementación.
- **XSIM** para simulación.
- **Terminal serie** configurado a 115200 8N1.
- **Visor de ondas** de XSIM para depuración.

2.3. Esquemático general.

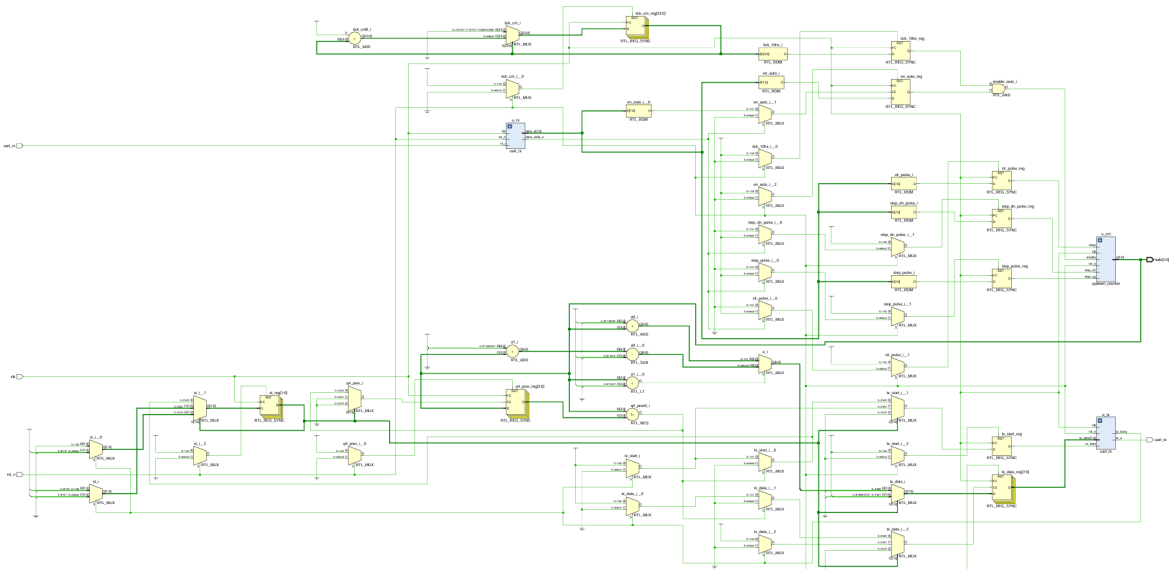

Figura 1: Vista RTL integrada del diseño con `uart_rx`, lógica de control, contador y `uart_tx`.

La figura 1 muestra el flujo completo desde la recepción UART hasta la actualización del contador y el eco por transmisión. El diseño opera con un único reloj y utiliza registros de sincronismo, multiplexores y comparadores simples para decidir cada transición de estado. La arquitectura permanece compacta y de baja ocupación de LUTs y FFs.

- **Entrada y sincronización.** La señal `uart_rx` ingresa por el banco de E/S y atraviesa doble registro de sincronización para mitigar metastabilidad antes del receptor UART. El `rst_n` se distribuye a los registros de estado y contadores de forma síncrona.
- **Decodificación UART RX.** El bloque `uart_rx` realiza sobremuestreo, detecta inicio de trama y entrega `data_o[7:0]` junto con un pulso `data_valid_o`. Este pulso habilita la lógica de comandos.
- **Lógica de control.** Una FSM ligera interpreta los comandos `'+'`, `'-'`, `'s'`, `'p'` y `'z'`. Genera pulsos de un ciclo `step_up` y `step_dn`, activa `enable` para el modo automático y produce `clear` cuando corresponde.
- **Generación de *tick*.** Un divisor de frecuencia produce `tick_10Hz`. Este pulso avanza el contador cuando el modo automático está activo. El camino de reloj se mantiene único y atraviesa un BUFG.
- **Contador up/down.** El bloque `updown_counter` suma o resta sobre cuatro bits con multiplexores que seleccionan la operación según `step_up`, `step_dn` y `enable`. `clear` fuerza el valor cero. La salida `q[3:0]` alimenta los LEDs.
- **Interfaz de salida.** La lógica de eco convierte el valor del contador a ASCII hexadecimal y solicita transmisión a `uart_tx` cuando `tx_busy` está inactivo. El camino hacia `uart_tx` está arbitrado para evitar colisiones.
- **E/S hacia placa.** `q[3:0]` se enruta a `leds[3:0]`. Las líneas `uart_rx` y `uart_tx` se fijan en los pines asignados del PMOD JA con `LVCMOS33`. No se usan BRAM ni DSP.

1. Configuración del proyecto en Vivado..

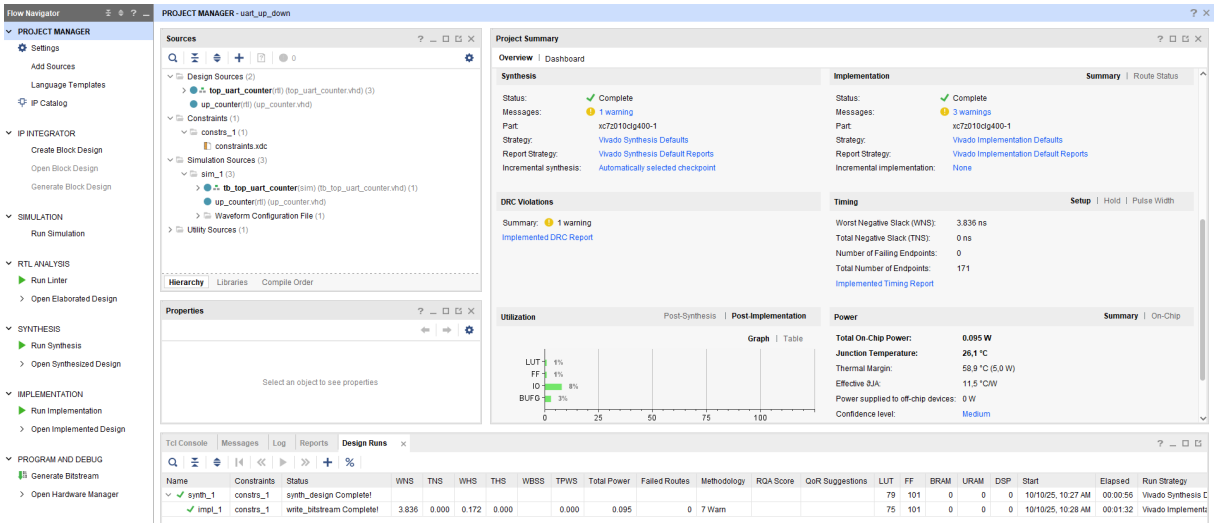

Figura 2: Panel de configuración del proyecto `uart_up_down` en Vivado.

Se preparó un proyecto en Vivado para la placa Arty Z7-10 con un diseño propio en VHDL. Se definieron pines, reloj, señales de UART y LEDs según la tarjeta. El proyecto usa parámetros simples para la velocidad serie y un pulso de 10 Hz. Se sintetizó e implementó con las opciones por defecto y sin violaciones de temporización. Los reportes muestran baja ocupación y una potencia estimada cercana a 0.095 W. La simulación envía comandos por UART, actualiza el contador y confirma el eco por transmisión. La organización de archivos es sencilla y el bitstream final se generó correctamente. El diseño quedó listo para cargar en la FPGA y probar en hardware.

```
1  ## Clock PL 125 MHz
2  set_property PACKAGE_PIN H16 [get_ports clk]
3  set_property IOSTANDARD LVCMOS33 [get_ports clk]
4  create_clock -period 8.000 -name sys_clk [get_ports clk]
5
6  ## Reset activo en 0 (SW0)
7  set_property PACKAGE_PIN M20 [get_ports rst_n]
8  set_property IOSTANDARD LVCMOS33 [get_ports rst_n]
9  set_property PULLUP true [get_ports rst_n]
10
11 ## UART RX desde PMOD JA1 (TX del USB-UART → JA1)
12 set_property PACKAGE_PIN Y18 [get_ports uart_rx]
13 set_property IOSTANDARD LVCMOS33 [get_ports uart_rx]
14 set_property PULLUP true [get_ports uart_rx]
15
16 ## LEDs
17 set_property PACKAGE_PIN R14 [get_ports {leds[0]}]
18 set_property PACKAGE_PIN P14 [get_ports {leds[1]}]
19 set_property PACKAGE_PIN N16 [get_ports {leds[2]}]
20 set_property PACKAGE_PIN M14 [get_ports {leds[3]}]
21 set_property IOSTANDARD LVCMOS33 [get_ports {leds[*]}]
22
23 ## UART TX hacia PMOD JA2 (conectar a RX del USB-UART)
24 set_property PACKAGE_PIN Y16 [get_ports uart_tx]
25 set_property IOSTANDARD LVCMOS33 [get_ports uart_tx]
```

Figura 3: Bloque `updown_counter` con salidas hacia `leds[3:0]`.

El archivo XDC fija el reloj del PL a 125 MHz en el pin H16 y crea el reloj lógico `sys_clk` con período de 8 ns. Define el reset activo en bajo en el pin M20 con *pull-up* interno. Asigna

la entrada `uart_rx` al pin Y18 con estándar LVCMOS33 y *pull-up*, y la salida `uart_tx` al pin Y16 con LVCMOS33. Mapea `leds[3:0]` a R14, P14, N16 y M14. Estas restricciones aseguran la coincidencia entre los puertos RTL y la placa Arty Z7-10 y permiten a la herramienta realizar el análisis temporal con el reloj correcto y los estándares de E/S adecuados.

2. Contador up/down de 4 bits..

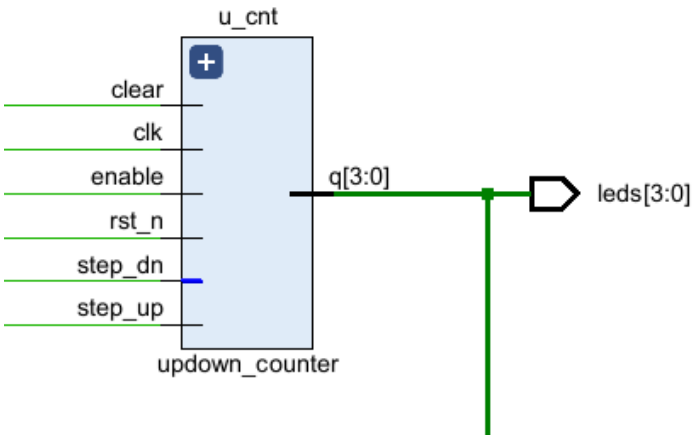

Figura 4: Bloque `updown_counter` con salidas hacia `leds[3:0]`.

El contador implementa una cuenta módulo 16 con operación ascendente y descendente sobre un único dominio de reloj. Expone entradas `clk`, `rst_n`, `clear`, `enable`, `step_up` y `step_dn`, y entrega `q[3:0]` conectado directamente a `leds[3:0]`. `rst_n` es un reinicio activo en bajo que pone el valor en cero de forma síncrona. `clear` fuerza el conteo a cero cuando se lo activa desde la lógica de comandos. `step_up` y `step_dn` son pulsos de un ciclo que incrementan o decrementan el valor en una unidad con aritmética de "wrap" en 4 bits. `enable` permite el avance automático cuando llega el *tick* del divisor, manteniendo el valor cuando está inactivo.

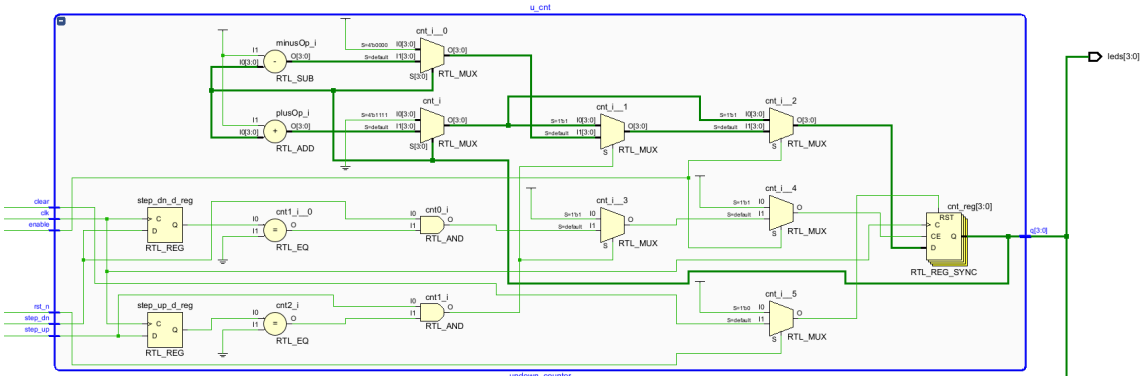

Figura 5: Vista del bloque contador de 4 bits conectado a los leds.

Es un contador de 4 bits que puede subir o bajar según la orden recibida. Recibe como entradas el reloj, un reinicio activo en bajo, un borrado a cero, un habilitador de conteo y dos pulsos para incrementar o decrementar. Cuando llega un pulso de subida, el valor aumenta en una unidad. Cuando llega un pulso de bajada, el valor disminuye en una unidad. Si enable está activo, el bloque puede avanzar de forma automática con el tick del sistema. La entrada clear fuerza el valor a cero de manera inmediata y controlada. El reinicio establece el contador en un estado conocido y evita estados indeterminados. La salida q[3:0] se conecta directamente a los cuatro leds de usuario para visualizar el valor. El diseño opera en un único dominio de reloj y asegura transiciones limpias entre estados. El comportamiento es estable, simple de integrar y ocupa pocos recursos de la fpga.

3. Módulo UART RX.

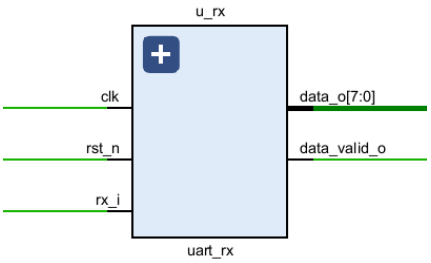

Figura 6: Bloque funcional del receptor `uart_rx`.

El bloque recibe la señal serie por la entrada rx_i y la sincroniza al reloj del sistema. Decodifica tramas 8N1 y entrega el byte recibido en data_o[7:0]. Emite un pulso en data_valid_o para señalar que el dato está listo y estable. La entrada rst_n pone al módulo en un estado conocido y limpia cualquier recepción en curso. El diseño opera con sobremuestreo y muestreo al centro del bit para mayor robustez. La interfaz es sencilla y permite conectar el receptor a lógica de control sin señales adicionales.

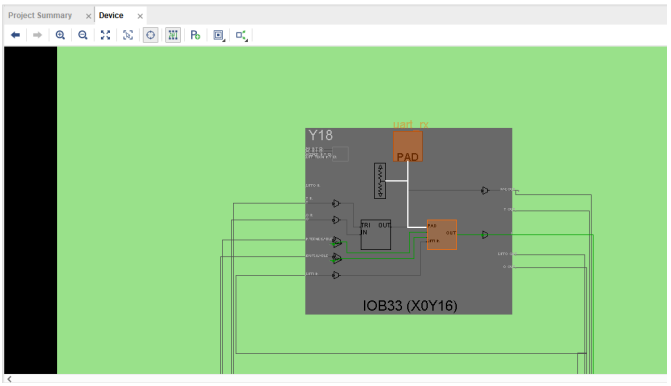

Figura 7: Esquema RTL del módulo `uart_rx`.

La Figura 6 muestra el bloque funcional del receptor, donde se observan sus puertos principales: *clk*, *rst_n* y *rx_i* como entradas, y *data_o[7:0]* junto con *data_valid_o* como salidas. El objetivo del bloque es tomar la señal serie 8N1, alinearla al reloj del sistema y entregar el byte recibido acompañado de un pulso de validez. El diseño mantiene una interfaz mínima para facilitar la integración con la lógica de control.

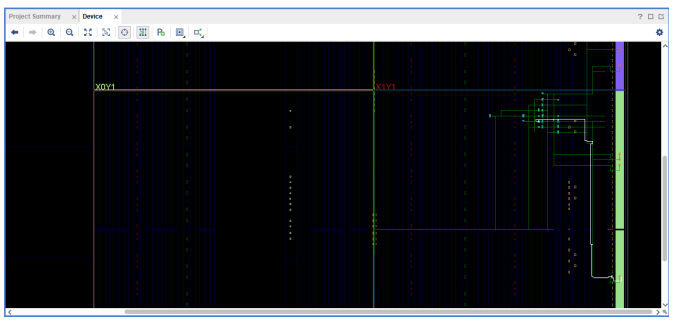

Figura 8: Ubicación física del pin `uart_rx` en Y18.

La Figura 7 corresponde a la vista física en el dispositivo y resalta el sitio de E/S Y18 asignado a *uart_rx*. Se confirma que el pin está configurado como entrada LVCMOS33 y conectado a la

cadena de sincronización en el banco de E/S. El enrutamiento hacia la lógica interna es directo y no introduce recorridos críticos apreciables, lo que favorece el margen temporal del receptor.

4. Módulo UART TX.

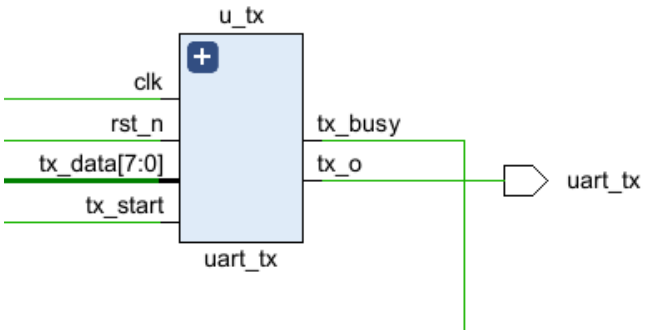

Figura 9: Esquema del módulo uart_tx.

El bloque representa el transmisor UART con una interfaz simple. Recibe clk y rst_n como señales de control, el byte a enviar por tx_data[7:0] y un pulso de arranque en tx_start. Genera la trama 8N1 por la salida tx_o con nivel de reposo alto y envío LSB primero. Expone tx_busy para indicar ocupación mientras dura la transmisión y evitar que se inicien tramas superpuestas. El uso de tx_start como pulso de un ciclo facilita el acoplamiento con la lógica que arbitra los envíos desde el contador.

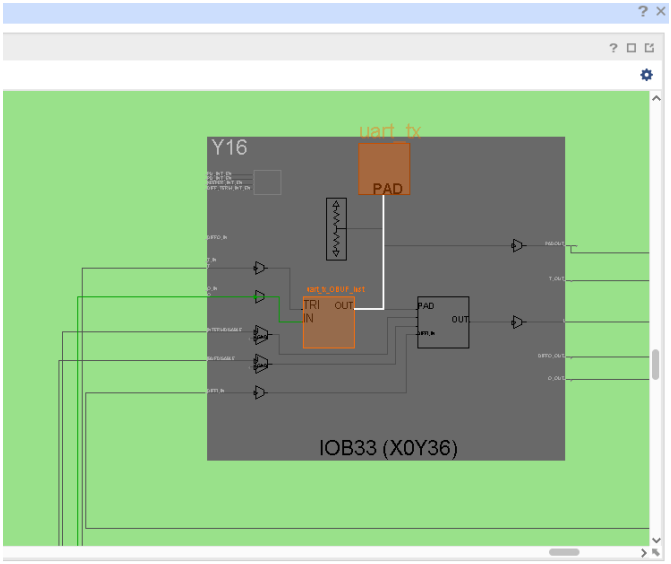

Figura 10: Esquema RTL del módulo `uart_tx`.

La Figura 9 muestra el bloque funcional del transmisor, con las señales de entrada *clk*, *rst_n*, *tx_data[7:0]* y *tx_start*, y las salidas *tx_o* y *tx_busy*. El módulo serializa el byte recibido en formato 8N1 y genera *tx_busy* mientras dura la transmisión para evitar colisiones de arranque. La interfaz es directa y permite conectar el emisor a la lógica de control sin señales adicionales.

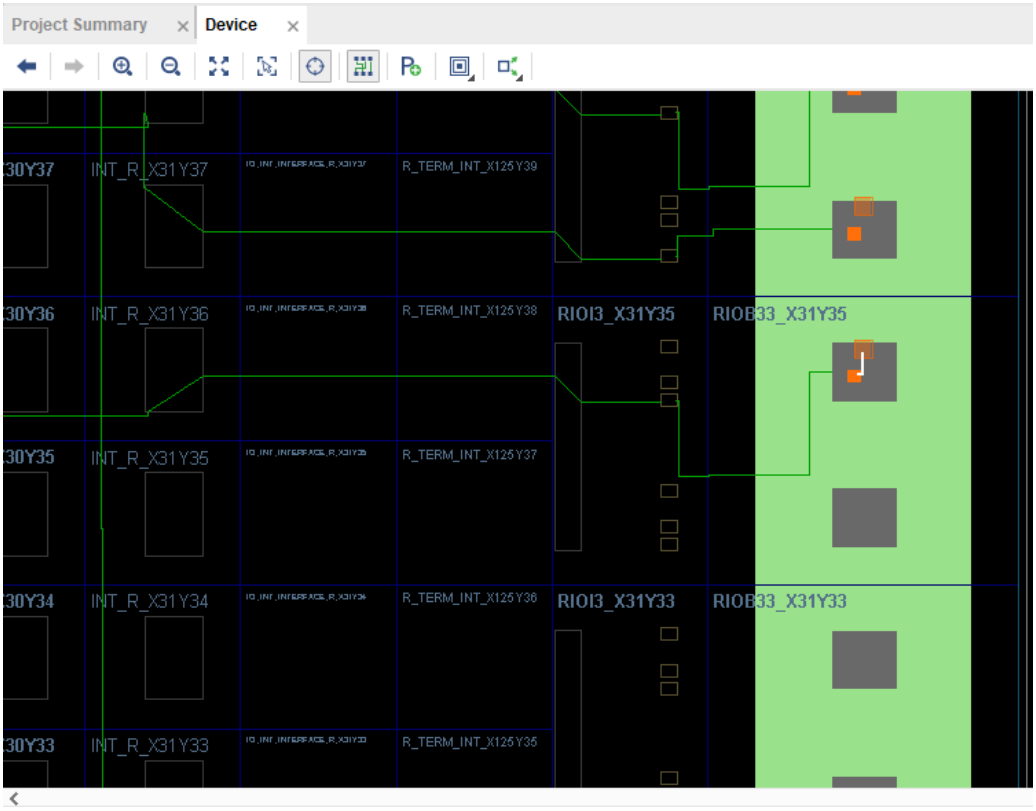

Figura 11: Ubicación física del pin `uart_tx` en Y16.

La Figura 10 corresponde a la vista física del pin *uart_tx* ubicado en Y16. El pad está configurado como salida LVCMOS33 y se observa el buffer de salida asociado al banco de E/S. El enrutamiento desde el bloque UART hacia el pad es corto y regular, lo que ayuda a conservar margen temporal y una conmutación limpia en la línea serie.

5. Simulación funcional.

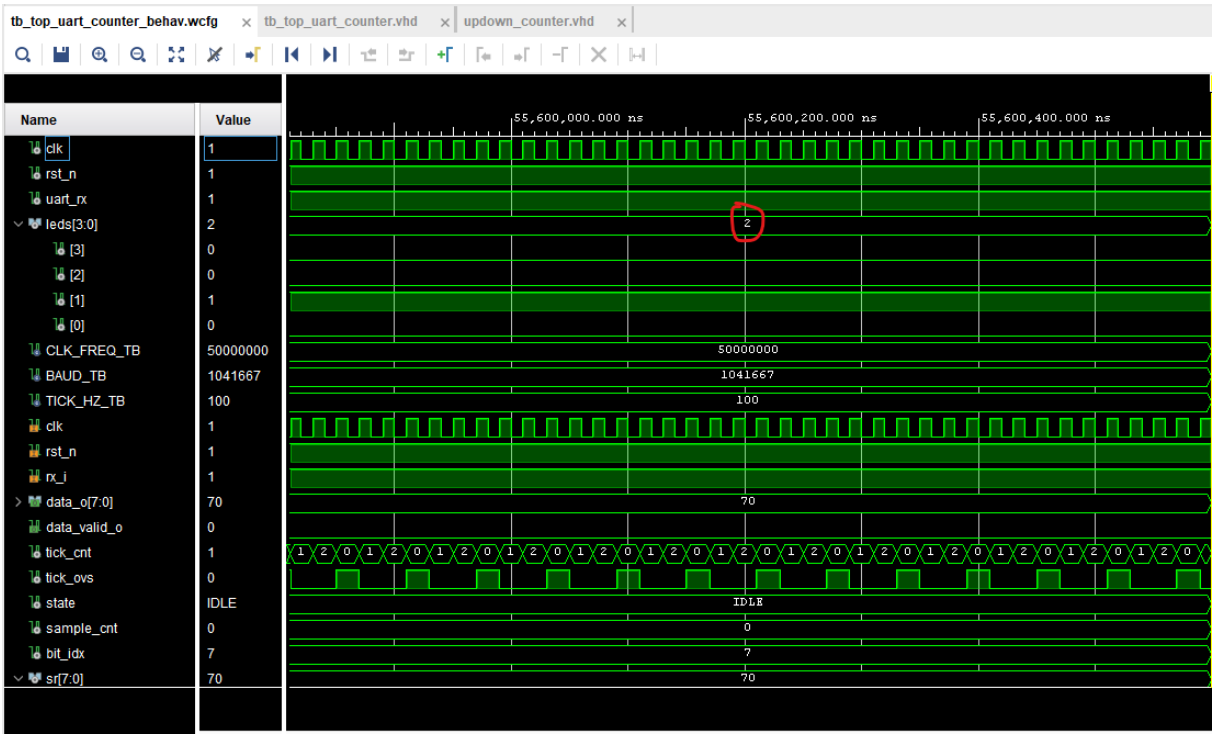

Figura 12: Ondas de XSIM con recepción de bytes y actualización del conteo.

El banco de pruebas instanció el diseño `top_uart_counter` y lo ejercitó como si los bytes llegaran por la línea `uart_rx`.

La secuencia de estímulos fue la siguiente.

- Se envió `'z'` para limpiar el contador y dejar los leds en cero.
- Se enviaron dos veces `'+'` para incrementar manualmente. Tras estos dos pulsos el valor esperado en `leds[3:0]` es `0010` (decimal 2).
- Se envió `'s'` para habilitar el modo automático y se dejó contar durante un intervalo.
- Se envió `'p'` para pausar el conteo automático.

Las ondas de simulación muestran que, luego del reinicio y del comando `'z'`, el contador queda en cero. Con cada `'+'` la cuenta aumenta en una unidad hasta alcanzar dos, tal como se observa en la salida de los leds. Al activar `'s'` el conteo progresa en forma periódica y se detiene con `'p'`. Esta secuencia confirma el funcionamiento esperado del sistema.

6. Resultados de implementación..

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (4400)	LUT as Logic (17600)	Bonded IOB (100)	BUFGCTRL (32)
top_uart_counter	75	101	32	75	8	1
u_cnt (updown_counter)	13	6	8	13	0	0
u_rx (uart_rx)	37	28	12	37	0	0
u_tx (uart_tx)	20	25	9	20	0	0

Figura 13: Utilización de recursos por jerarquía.

La Figura 12 resume la utilización por jerarquía y confirma un uso global muy bajo de recursos. El diseño principal ocupa decenas de LUTs y poco más de un centenar de registros, con ocho IOBs asignados para pines de usuario y un solo BUFG para el reloj. Los módulos `uart_rx` y `uart_tx` concentran la mayor parte de la lógica, mientras que el contador aporta una fracción menor.

7. Análisis de potencia..

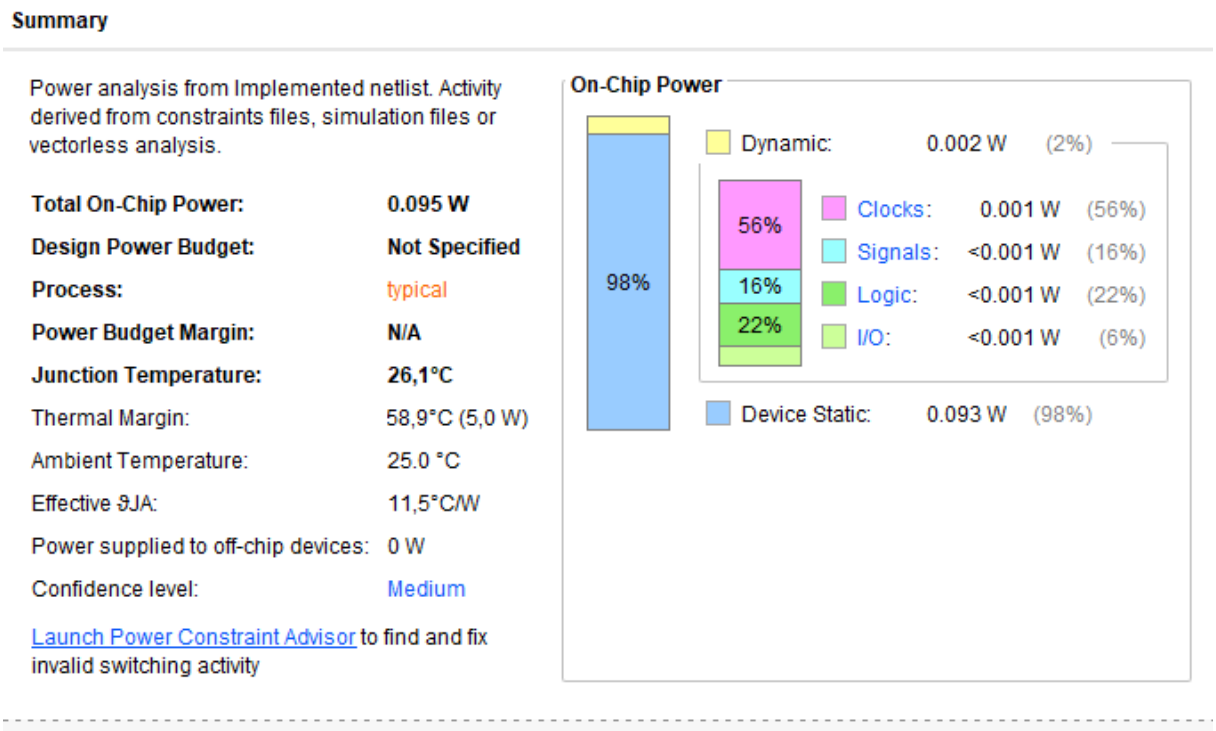

Figura 14: Resumen de potencia estática y dinámica.

La Figura 14 muestra el resumen de potencia estimada tras la implementación. La potencia total en chip es de aproximadamente 0.095 W, dominada por el componente estático. La potencia dinámica es cercana a 0.002 W y se reparte principalmente entre la red de reloj, señales y lógica. La temperatura de unión reportada se mantiene baja y el margen térmico es amplio. Estos resultados indican que el diseño opera con consumo reducido y deja margen suficiente para futuras extensiones sin comprometer el presupuesto de potencia.

8. Análisis temporal.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	3,664 ns	Worst Hold Slack (WHS):	0,136 ns	Worst Pulse Width Slack (WPWS):	3,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	171	Total Number of Endpoints:	171	Total Number of Endpoints:	102

Figura 15: Resumen de temporización postimplementación con WNS y TNS válidos.

Tomando en cuenta que:

- Setup slack (WNS/TNS): margen antes del flanco de captura. Debe ser ≥ 0 ns en todas las rutas.
- Hold slack (WHS/THS): margen inmediatamente después del flanco de captura. Debe ser ≥ 0 ns.
- Pulse width (WPWS/TPWS): ancho mínimo de pulsos de reloj o señal. Debe ser ≥ 0 ns.
- Número de *endpoints* fallidos: debe ser 0.

En este diseño los *slacks* son positivos y no hay *endpoints* fallidos, por lo que se cumple la temporización a la frecuencia objetivo.

9. Pruebas en hardware.

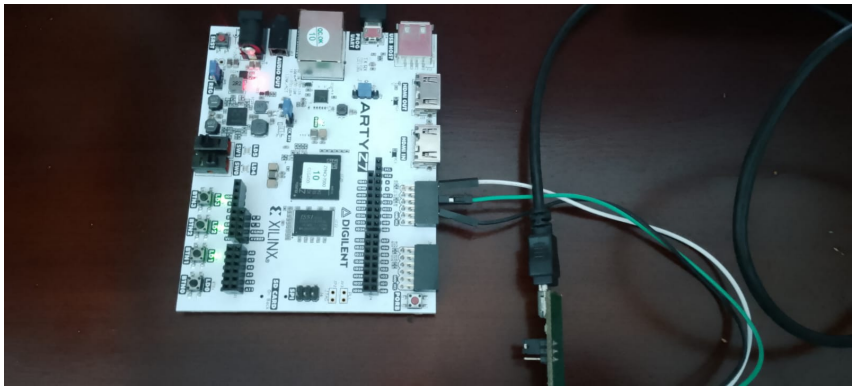

Figura 16: Hardware conectado y en funcionamiento.

Se implementó el diseño en hardware y se programó la FPGA con el bitstream generado. La interfaz serie del sistema se conectó a un convertidor USB–serial basado en FTDI a nivel de 3,3 V, compartiendo tierra con la placa. La señal de transmisión del diseño se llevó al pin del adaptador y la recepción se conectó desde el adaptador al pin de entrada del módulo, respetando la cruz RX–TX. En la PC se utilizó la terminal AccessPort configurada a 115200 bps, 8N1 y sin control de flujo.

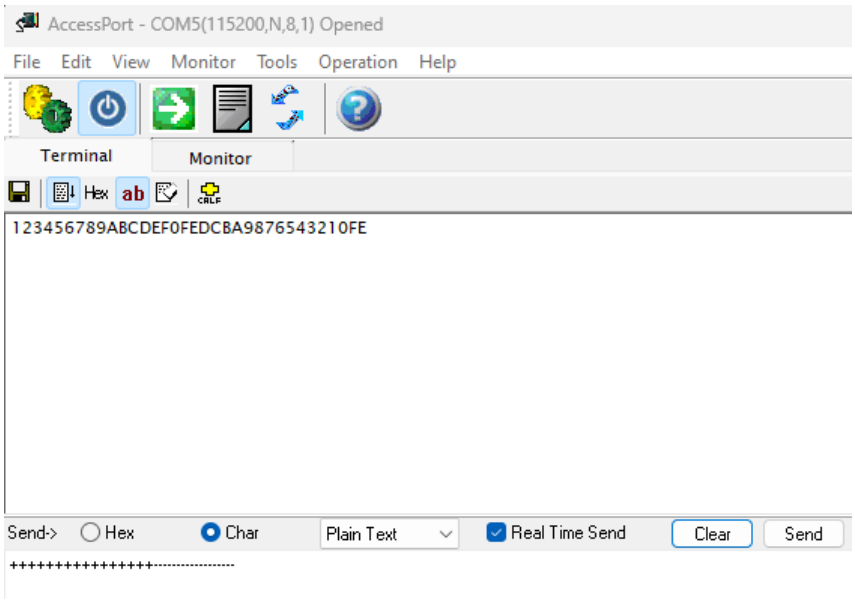

Figura 17: Terminal a 115 200 bps con eco del valor del contador en hexadecimal.

Durante las pruebas se enviaron los comandos `z`, `+`, `-`, `s` y `p`. El equipo respondió con el eco esperado y el valor del contador en hexadecimal en la ventana de la terminal. Los LEDs de usuario reflejaron el mismo conteo en todo momento. No se observaron errores de trama ni pérdidas de caracteres, lo que confirma el funcionamiento correcto de la comunicación y del control del contador en la placa.

10. Conclusiones.

Se implementó correctamente un receptor y un transmisor UART con formato 8N1 a 115 200 bps. La comunicación mostró estabilidad y no se observaron errores de trama durante las pruebas.

Se integró un contador ascendente y descendente de cuatro bits controlado por comandos recibidos por UART. Las órdenes `z`, `+`, `-`, `s` y `p` actuaron según lo especificado y el valor se reflejó en `leds[3:0]` y en el eco por consola.

La verificación por simulación y en hardware confirmó la funcionalidad del sistema. Las ondas de simulación mostraron la decodificación correcta y el pulso de validez. En la placa, el terminal serie mostró el eco esperado y la temporización cumplió a la frecuencia objetivo.

El diseño presentó baja utilización de recursos y potencia estimada reducida. La arquitectura modular facilitó la validación por partes y deja margen para escalar el ancho del contador o ajustar parámetros de UART y *tick*. En conjunto, los objetivos planteados se alcanzaron satisfactoriamente.

11. Recomendaciones.

- Crear y ejecutar archivos de prueba antes de pasar a implementación para detectar errores temprano y evitar tiempos perdidos en producción.
- Revisar cuidadosamente el archivo `.xdc` para prevenir fallos de conexión y posibles daños en pines por asignaciones incorrectas.
- Verificar la compatibilidad de niveles lógicos. La FPGA trabaja a 3,3 V LVCMOS y el módulo USB–serial debe operar al mismo nivel.
- Conectar las señales cruzadas correctamente. El pin `TX` del módulo externo debe ir al `RX` de la FPGA y el `TX` de la FPGA debe ir al `RX` del módulo.
- Confirmar que los controladores del adaptador USB–serial estén instalados y funcionando en el sistema operativo.
- Probar los módulos por separado antes de integrarlos. Validar `uart_rx`, `uart_tx` y el contador de forma individual con bancos de prueba simples.

12. Bibliografías..

Referencias

[1] Digilent Inc., *Arty Z7 Reference Manual*. Disponible en: https://digilent.com/reference/programmable-logic/arty-z7/reference-manual. Accedido: octubre de 2025.

[2] AMD Xilinx, *Vivado Design Suite User Guide: Synthesis (UG901)*. Disponible en: https://docs.xilinx.com/. Accedido: octubre de 2025.

[3] AMD Xilinx, *Vivado Design Suite User Guide: Using Constraints (UG903)*. Disponible en: https://docs.xilinx.com/. Accedido: octubre de 2025.

[4] AMD Xilinx, *Vivado Design Suite User Guide: Implementation (UG904)*. Disponible en: https://docs.xilinx.com/. Accedido: octubre de 2025.

[5] AMD Xilinx, *UltraFast Design Methodology Guide (UG949)*. Disponible en: https://docs.xilinx.com/. Accedido: octubre de 2025.

[6] AMD Xilinx, *7 Series FPGAs SelectIO Resources (UG471)*. Disponible en: https://docs.xilinx.com/. Accedido: octubre de 2025.

[7] Analog Devices, *UART: A Hardware Communication Protocol — Understanding Universal Asynchronous Receiver/Transmitter*. Disponible en: https://www.analog.com/. Accedido: octubre de 2025.

[8] Wikipedia, *Universal asynchronous receiver-transmitter*. Disponible en: https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter. Accedido: octubre de 2025.

13. Anexos.

14. Anexos..

A. Archivo superior: `top_uart_counter.vhd`.

Listing 14.1: Módulo superior `top_uart_counter`.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_uart_counter is
  generic(
    CLK_FREQ : integer := 100_000_000; -- ajusta al reloj de tu FPGA
    BAUD : integer := 115_200
  );
  port(
    clk : in std_logic;
    rst_n : in std_logic; -- activo en 0
    uart_rx : in std_logic; -- pin RX desde el adaptador USB-UART
    leds : out std_logic_vector(3 downto 0)
  );
end entity;

architecture rtl of top_uart_counter is
  -- UART
  signal rx_data : std_logic_vector(7 downto 0);
  signal rx_valid : std_logic;

  -- Control
  signal en_auto : std_logic := '0';
  signal step_pulse : std_logic := '0';
  signal clr_pulse : std_logic := '0';

  -- Tick ~10 Hz para modo automtico
  constant TICK_HZ : integer := 10;
  constant TICK_DIV : integer := integer(real(CLK_FREQ)/real(TICK_HZ) + 0.5);
  signal tick_cnt : integer range 0 to TICK_DIV-1 := 0;
  signal tick_10hz : std_logic := '0';

  signal q4 : std_logic_vector(3 downto 0);
begin
  leds <= q4;

  -- Instancia UART RX
  u_rx: entity work.uart_rx
    generic map(
      CLK_FREQ => CLK_FREQ,
      BAUD => BAUD
    )
    port map(
      clk => clk,
      rst_n => rst_n,
      rx_i => uart_rx,
      data_o => rx_data,
      data_valid_o => rx_valid
    );

```

```
52  -- Generador de tick 10 Hz (para conteo automtico)
53  process(clk)
54  begin
55    if rising_edge(clk) then
56      if rst_n = '0' then
57        tick_cnt <= 0;
58        tick_10hz <= '0';
59      else
60        if tick_cnt = TICK_DIV-1 then
61          tick_cnt <= 0;
62          tick_10hz <= '1';
63        else
64          tick_cnt <= tick_cnt + 1;
65          tick_10hz <= '0';
66        end if;
67      end if;
68    end if;
69  end process;
70
71  -- Decodificacin de comandos UART
72  -- '+' o 'i' -> step
73  -- 'z' o 'r' -> clear
74  -- 's' -> start (enable auto)
75  -- 'p' -> pause (disable auto)
76  process(clk)
77  begin
78    if rising_edge(clk) then
79      if rst_n = '0' then
80        en_auto <= '0';
81        step_pulse <= '0';
82        clr_pulse <= '0';
83      else
84        step_pulse <= '0';
85        clr_pulse <= '0';
86
87        if rx_valid = '1' then
88          case character'val(to_integer(unsigned(rx_data))) is
89            when '+' | 'i' =>
90              step_pulse <= '1';
91            when 'z' | 'r' =>
92              clr_pulse <= '1';
93            when 's' =>
94              en_auto <= '1';
95            when 'p' =>
96              en_auto <= '0';
97            when others =>
98              null;
99          end case;
100       end if;
101     end if;
102   end if;
103 end process;
104
105 -- Instancia del contador (enable con tick si auto)
106 u_cnt: entity work.up_counter
107   port map(
108     clk => clk,
109     rst_n => rst_n,
```

```
110       enable => (tick_10hz and en_auto),
111       step => step_pulse,
112       clear => clr_pulse,
113       q => q4
114     );
115 end architecture;
```

B. Receptor UART: `uart_rx.vhd`.

Listing 14.2: Bloque `uart_rx` con sobremuestreo y pulso de validez.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity uart_rx is
6    generic(
7      CLK_FREQ : integer := 100_000_000; -- Hz
8      BAUD : integer := 115_200 -- bps
9    );
10   port(
11     clk : in std_logic;
12     rst_n : in std_logic; -- activo en 0
13     rx_i : in std_logic; -- lnea RX desde el pin
14     data_o : out std_logic_vector(7 downto 0);
15     data_valid_o : out std_logic -- pulso 1 clk cuando data_o es vlida
16   );
17 end entity;
18
19 architecture rtl of uart_rx is
20   constant OVS : integer := 16; -- sobremuestreo x16
21   constant BAUD_OVS : integer := BAUD * OVS;
22   constant DIVISOR : integer := integer(real(CLK_FREQ) / real(BAUD_OVS) + 0.5); --
         redondeo
23
24   signal tick_cnt : integer range 0 to DIVISOR-1 := 0;
25   signal tick_ovs : std_logic := '0';
26
27   type state_t is (IDLE, START, DATA, STOP);
28   signal state : state_t := IDLE;
29   signal sample_cnt : integer range 0 to OVS-1 := 0;
30   signal bit_idx : integer range 0 to 7 := 0;
31   signal sr : std_logic_vector(7 downto 0) := (others=>'0');
32   signal rx_sync0, rx_sync1 : std_logic := '1';
33   signal data_valid_r : std_logic := '0';
34 begin
35   data_o <= sr;
36   data_valid_o <= data_valid_r;
37
38   -- Sincronizadores para rx_i (evitar metastabilidad)
39   process(clk)
40   begin
41     if rising_edge(clk) then
42       rx_sync0 <= rx_i;
43       rx_sync1 <= rx_sync0;
44     end if;
```

```
45  end process;
46
47  -- Generador de tick a BAUD*16
48  process(clk)
49  begin
50    if rising_edge(clk) then
51      if rst_n = '0' then
52        tick_cnt <= 0;
53        tick_ovs <= '0';
54      else
55        if tick_cnt = DIVISOR-1 then
56          tick_cnt <= 0;
57          tick_ovs <= '1';
58        else
59          tick_cnt <= tick_cnt + 1;
60          tick_ovs <= '0';
61        end if;
62      end if;
63    end if;
64  end process;
65
66  -- FSM receptor
67  process(clk)
68  begin
69    if rising_edge(clk) then
70      data_valid_r <= '0';
71      if rst_n = '0' then
72        state <= IDLE;
73        sample_cnt <= 0;
74        bit_idx <= 0;
75        sr <= (others=>'0');
76      else
77        if tick_ovs = '1' then
78          case state is
79            when IDLE =>
80              if rx_sync1 = '0' then -- borde de inicio
81                state <= START;
82                sample_cnt <= OVS/2; -- muestreo al medio del bit
83              end if;
84
85            when START =>
86              if sample_cnt = 0 then
87                if rx_sync1 = '0' then -- start vlido
88                  state <= DATA;
89                  sample_cnt <= OVS-1;
90                  bit_idx <= 0;
91                else
92                  state <= IDLE; -- falso inicio
93                end if;
94              else
95                sample_cnt <= sample_cnt - 1;
96              end if;
97
98            when DATA =>
99              if sample_cnt = 0 then
100               -- muestrear bit de datos LSB primero
101               sr(bit_idx) <= rx_sync1;
102               if bit_idx = 7 then
```

```
103                 state <= STOP;
104               else
105                 bit_idx <= bit_idx + 1;
106               end if;
107               sample_cnt <= OVS-1;
108             else
109               sample_cnt <= sample_cnt - 1;
110             end if;
111
112           when STOP =>
113             if sample_cnt = 0 then
114               -- comprobar bit de stop (opcional); se asume correcto
115               data_valid_r <= '1';
116               state <= IDLE;
117             else
118               sample_cnt <= sample_cnt - 1;
119             end if;
120         end case;
121       end if;
122     end if;
123   end if;
124  end process;
125 end architecture;
```

C. Transmisor UART: `uart_tx.vhd`.

Listing 14.3: Bloque `uart_tx` para tramas 8N1.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity uart_tx is
6    generic(
7      CLK_FREQ : integer := 125_000_000; -- Hz
8      BAUD : integer := 115_200 -- bps
9    );
10   port(
11     clk : in std_logic;
12     rst_n : in std_logic; -- activo en 0
13     tx_start : in std_logic; -- pulso 1 clk para iniciar envo
14     tx_data : in std_logic_vector(7 downto 0); -- byte a enviar
15     tx_busy : out std_logic; -- '1' mientras transmite
16     tx_o : out std_logic -- lnea TX (idle='1')
17   );
18 end entity;
19
20 architecture rtl of uart_tx is
21   constant DIVISOR : integer := integer(real(CLK_FREQ)/real(BAUD) + 0.5);
22
23   type state_t is (IDLE, START, DATA, STOP);
24   signal state : state_t := IDLE;
25   signal bit_cnt : integer range 0 to 7 := 0;
26   signal shreg : std_logic_vector(7 downto 0) := (others=>'0');
27
28   signal baud_cnt : integer range 0 to DIVISOR-1 := 0;
```

```
29    signal baud_tick: std_logic := '0';
30
31    signal tx_r : std_logic := '1';
32    signal busy_r : std_logic := '0';
33  begin
34    tx_o <= tx_r;
35    tx_busy<= busy_r;
36
37    -- generador de tick a BAUD
38    process(clk)
39    begin
40      if rising_edge(clk) then
41        if rst_n = '0' then
42          baud_cnt <= 0;
43          baud_tick <= '0';
44        else
45          if baud_cnt = DIVISOR-1 then
46            baud_cnt <= 0;
47            baud_tick <= '1';
48          else
49            baud_cnt <= baud_cnt + 1;
50            baud_tick <= '0';
51          end if;
52        end if;
53      end if;
54    end process;
55
56    -- FSM de transmisin
57    process(clk)
58    begin
59      if rising_edge(clk) then
60        if rst_n = '0' then
61          state <= IDLE;
62          tx_r <= '1';
63          busy_r <= '0';
64          bit_cnt <= 0;
65          shreg <= (others=>'0');
66        else
67          case state is
68            when IDLE =>
69              busy_r <= '0';
70              tx_r <= '1';
71              if tx_start = '1' then
72                shreg <= tx_data;
73                bit_cnt <= 0;
74                busy_r <= '1';
75                state <= START;
76              end if;
77
78            when START =>
79              if baud_tick = '1' then
80                tx_r <= '0'; -- start bit
81                state <= DATA;
82              end if;
83
84            when DATA =>
85              if baud_tick = '1' then
86                tx_r <= shreg(bit_cnt);
```

```
87              if bit_cnt = 7 then
88                state <= STOP;
89              else
90                bit_cnt <= bit_cnt + 1;
91              end if;
92            end if;
93
94          when STOP =>
95            if baud_tick = '1' then
96              tx_r <= '1'; -- stop bit
97              state <= IDLE;
98            end if;
99        end case;
100     end if;
101   end if;
102 end process;
103 end architecture;
```

D. Contador up/down: `updown_counter.vhd`.

Listing 14.4: Contador de 4 bits con control de subida y bajada.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity updown_counter is
6    port(
7      clk : in std_logic;
8      rst_n : in std_logic; -- activo en 0
9      enable : in std_logic; -- auto (+1 por tick, como antes)
10     step_up : in std_logic; -- pulso: +1
11     step_dn : in std_logic; -- pulso: -1
12     clear : in std_logic; -- pulso: pone a 0
13     q : out std_logic_vector(3 downto 0)
14   );
15 end entity;
16
17 architecture rtl of updown_counter is
18   signal cnt : unsigned(3 downto 0) := (others=>'0');
19   signal step_up_d : std_logic := '0';
20   signal step_dn_d : std_logic := '0';
21 begin
22   q <= std_logic_vector(cnt);
23
24   process(clk)
25   begin
26     if rising_edge(clk) then
27       -- detectores de flanco
28       step_up_d <= step_up;
29       step_dn_d <= step_dn;
30
31       if rst_n = '0' then
32         cnt <= (others=>'0');
33
34       elsif clear = '1' then
```

```
35        cnt <= (others=>'0');
36
37      elsif enable = '1' then
38        -- auto: igual que antes (solo +1 por tick)
39        if cnt = "1111" then cnt <= (others=>'0'); else cnt <= cnt + 1; end if;
40
41      elsif (step_up = '1' and step_up_d = '0') then
42        if cnt = "1111" then cnt <= (others=>'0'); else cnt <= cnt + 1; end if;
43
44      elsif (step_dn = '1' and step_dn_d = '0') then
45        if cnt = "0000" then cnt <= "1111"; else cnt <= cnt - 1; end if;
46
47      end if;
48    end if;
49  end process;
50 end architecture;
```

E. Banco de pruebas: `tb_top_uart_counter.vhd`.

Listing 14.5: Testbench con envío de comandos `z`, `+`, `s` y `p`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity tb_top_uart_counter is
6  end entity;
7
8  architecture sim of tb_top_uart_counter is
9    -- Parmetros de la sim
10   constant CLK_FREQ_TB : integer := 50_000_000; -- 50 MHz para sim
11   constant BAUD_TB : integer := 1_000_000; -- 1 Mbps para hacerla corta
12   constant TICK_HZ_TB : integer := 100; -- conteo auto ms gil en sim
13
14   signal clk : std_logic := '0';
15   signal rst_n : std_logic := '0';
16   signal uart_rx : std_logic := '1'; -- lnea en reposo = '1'
17   signal leds : std_logic_vector(3 downto 0);
18
19   -- Seales internas opcionales a monitorear (desde top con 'signal' o via waves)
20 begin
21   -- Reloj 50 MHz
22   clk <= not clk after 10 ns; -- periodo 20 ns
23
24   -- Reset
25   process
26   begin
27     rst_n <= '0';
28     wait for 200 ns;
29     rst_n <= '1';
30     wait;
31   end process;
32
33   -- DUT: instanciamos top con genricos ajustados
34   dut: entity work.top_uart_counter
35     generic map(
```

```
36      CLK_FREQ => CLK_FREQ_TB,
37      BAUD => BAUD_TB
38    )
39    port map(
40      clk => clk,
41      rst_n => rst_n,
42      uart_rx => uart_rx,
43      leds => leds
44    );
45
46  ----------------------------------------------------------------------
47  -- Tarea/Procedimiento: enviar un byte por UART en uart_rx (8N1)
48  ----------------------------------------------------------------------
49  procedure uart_send_byte(signal rx : out std_logic; constant byte : std_logic_vector
        (7 downto 0)) is
50    constant bit_time : time := 1 sec / BAUD_TB; -- periodo de bit
51  begin
52    -- start bit (0)
53    rx <= '0';
54    wait for bit_time;
55    -- 8 bits de datos LSB primero
56    for i in 0 to 7 loop
57      rx <= byte(i);
58      wait for bit_time;
59    end loop;
60    -- stop bit (1)
61    rx <= '1';
62    wait for bit_time;
63    -- un pequeo idle
64    wait for bit_time/2;
65  end procedure;
66
67  -- Secuencia de estmulos
68  stimulus: process
69    function c2slv(ch: character) return std_logic_vector is
70      variable tmp: std_logic_vector(7 downto 0);
71    begin
72      tmp := std_logic_vector(to_unsigned(character'pos(ch), 8));
73      return tmp;
74    end function;
75  begin
76    -- Espera salir de reset
77    wait until rst_n = '1';
78    wait for 50 us;
79
80    -- z -> clear
81    uart_send_byte(uart_rx, c2slv('z'));
82    wait for 200 us;
83
84    -- + -> step (2 veces)
85    uart_send_byte(uart_rx, c2slv('+'));
86    wait for 100 us;
87    uart_send_byte(uart_rx, c2slv('+'));
88    wait for 200 us;
89
90    -- s -> start auto
91    uart_send_byte(uart_rx, c2slv('s'));
92
```

```
93      -- deja correr un rato para que auto cuente
94      wait for 50 ms;
95
96      -- p -> pause
97      uart_send_byte(uart_rx, c2slv('p'));
98
99      -- espera y termina
100     wait for 5 ms;
101     assert false report "FIN DE LA SIMULACION" severity failure;
102   end process;
103
104 end architecture;
```