FUNDACIÓN FULGOR

CURSO DE DISEÑO DIGITAL

TP5 - Plataforma de Verificación MicroBlaze

PROCOM

DOCENTES: Dr. Pola, Ariel
Ing. Garaventta Pascual, Santiago
Ing. Pesce, Agustín

ALUMNO: Gatti, Ignacio Ezequiel

CÓRDOBA, ARGENTINA
2025

Índice

1. Introducción

Este TP5 plantea el cierre del circuito completo entre una computadora y una FPGA, utilizando una comunicación serie sencilla para **enviar órdenes** y **recibir respuestas**. El objetivo específico es permitir, desde la PC, seleccionar un LED de la placa y asignarle un color, así como consultar el estado de los *switches*. Para ello se implementaron dos componentes de software que se comunican entre sí: un programa en C que corre sobre MicroBlaze (en la FPGA) y un script en Python que corre en la PC. Ambos emplean un lenguaje de mensajes simple y consistente.

Más allá del encendido de LEDs, el aporte central del trabajo es la **integración** de contenidos vistos en la asignatura: el diseño de un protocolo básico (qué se envía, en qué orden y cómo se responde), su vinculación con los GPIO de la placa, y la verificación de que las órdenes enviadas desde la PC se reflejen efectivamente en el hardware. En este proceso se abordan decisiones típicas de sistemas embebidos: definición del formato de la trama, distinción entre comandos de **escritura** (control de LEDs) y de **lectura** (consulta de switches), y organización del código para facilitar su prueba y mantenimiento.

Desde el punto de vista de hardware, se partió de un diseño base de Vivado provisto por la Fundación, suficiente para enfocarse en la lógica de comunicación. Dicho archivo Verilog no fue modificado, priorizando un enfoque centrado en el comportamiento del sistema. El desarrollo se concentra, por un lado, en el firmware que interpreta los mensajes y acciona los periféricos, y por otro, en la herramienta de usuario (Python) que permite ejercitar y validar el conjunto de manera práctica.

En síntesis, el TP consolida el ida y vuelta PC $\leftrightarrow$ FPGA mediante un protocolo propio y un banco de pruebas simple, evidenciando el flujo completo: concepción de la interfaz, implementación y validación sobre la placa. Las secciones siguientes describen el protocolo propuesto, las decisiones de diseño en C y Python, los casos de prueba realizados y los resultados obtenidos.

2. Marco Teórico

2.1. Comunicación serie UART

La interfaz UART es un medio de comunicación serie asíncrono y punto a punto. A diferencia de buses síncronos, no transmite una señal de reloj dedicada: emisor y receptor acuerdan un ritmo de muestreo (baudios) y siguen una estructura de trama simple compuesta por un bit de inicio, un conjunto de bits de datos (LSB primero), un bit de paridad opcional y uno o más bits de parada. Su sencillez y bajo costo de integración la vuelven adecuada para control y monitoreo en sistemas embebidos. Como contrapartida, la protección frente a errores es limitada (típicamente sólo paridad) y no existe direccionamiento intrínseco, por lo que cualquier esquema de *comando–respuesta* debe definirse a nivel de aplicación.

2.2. Modelo por capas PC–FPGA

El intercambio PC–FPGA puede analizarse en dos niveles:

- **Capa física/enlace (UART):** establece cómo viajan los bits (formato de trama y temporización asíncrona).
- **Capa de aplicación (protocolo):** define el significado de los bytes. Un protocolo mínimo suele incluir: tipo de comando, identificadores de dispositivo o canal, datos asociados y, opcionalmente, algún mecanismo simple de verificación (eco, *checksum* o códigos de error).

Separar ambas capas permite razonar y depurar con claridad: si el enlace es confiable pero la respuesta no tiene el formato esperado, el problema reside en la capa de aplicación y no en UART.

2.3. MicroBlaze y GPIO mapeados a memoria

MicroBlaze es un *soft–core* que se integra con periféricos mediante un bus (p. ej., AXI). Los controladores de GPIO exponen *registros mapeados a memoria* que el procesador puede leer o escribir como variables normales. En este contexto:

- **Salida (LEDs):** escribir en los registros de salida actualiza el estado de los pines conectados a los LEDs. Es habitual definir una convención de bits (por ejemplo, 3 bits por color RGB).
- **Entrada (switches):** leer los registros de entrada devuelve el estado instantáneo de los interruptores. Según la placa y la IP de GPIO, pueden considerarse aspectos como la polaridad activa y la sincronización de entradas asíncronas.

2.4. Estrategias de E/S: sondeo vs. interrupciones

Para sistemas simples de control, el **sondeo** (*polling*) resulta suficiente: el firmware revisa periódicamente si llegó un comando por UART y actúa en consecuencia. En aplicaciones más exigentes en tiempo o consumo, las **interrupciones** permiten reaccionar de manera inmediata a eventos (recepción de un byte, cambio en GPIO), a costa de mayor complejidad de software. La elección depende del volumen de tráfico, los plazos de respuesta y la simplicidad buscada.

2.5. Criterios en protocolos simples

Aun con tramas cortas, es recomendable mantener reglas básicas: delimitar el inicio de un mensaje o usar longitud fija conocida; validar campos (rango de ID, tipo de comando); y prever respuestas explícitas (ACK/NACK o eco de comando) para facilitar la verificación desde la PC. Estas prácticas reducen ambigüedades y aceleran la depuración.

2.6. Validación

La verificación del sistema se apoya en casos de prueba reproducibles: (i) escritura determinista de un LED y confirmación visual; (ii) lectura del estado de los switches y decodificación consistente en PC; (iii) pruebas de error (comando no válido o datos fuera de rango) con la respuesta esperada. Este enfoque evidencia el correcto acople entre la capa UART y el protocolo de aplicación.

3. Consigna

- **MicroBlaze** El objetivo es utilizar el MicroBlaze para controlar los leds y leer el estado de los switch..
- **Ejercicio 1** Utilizando el tutorial desarrollado en la presentacion MicroBlaze.pdf y la aplicacion de Python que controla el puerto UART enviando una trama, vamos a controlar el encendido de los leds RGB y leer el estado de los switch.

  1. El usuario debe:
     a) Encender los leds
        - El led que quiere prender o apagar (0,1,2,3).
        - El color que desea prender en cada led. Puede ser Rojo, Verde o Azul o cualquier combinación de ellos.

     b) Leer el estado de los switch.

     c) a forma de utilizar la trama sera de nido por el desarrollador. Es decir, el campo Device y Data serán utilizados como el diseñador desee

     d) Un ejemplo del diseño completo se muestra en la 1

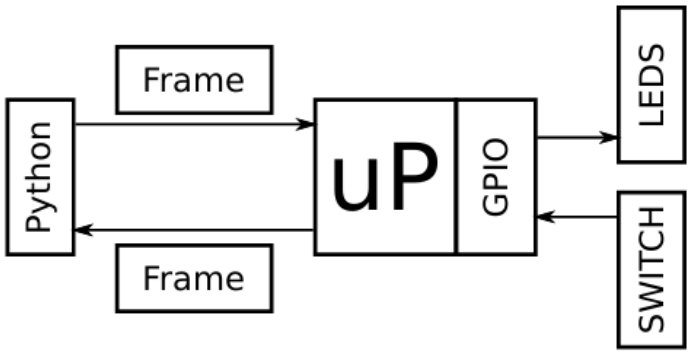

Figura 1: Diagrama en bloques del diseño completo.

Figura 1: Diagrama en bloques del diseño completo.

4. Implementación

4.1. Visión general del software

El diseño se divide en dos componentes:

1. **Firmware en C (MicroBlaze):** recibe tramas desde la PC, decodifica el comando y actúa sobre los GPIO para controlar los LEDs o leer los *switches*. *Archivo:* `protocolo_main.c`.

2. **Herramienta en Python (host):** ofrece un menú simple para el usuario, arma y envía las tramas al MicroBlaze y, cuando corresponde, decodifica la respuesta y muestra el estado de los *switches*. *Archivo:* `TP5_UART_FPGA_pv3.py`.

El archivo `fpga_procom.v` provisto por cátedra se mantuvo sin cambios, por lo que el foco de esta sección está en el protocolo y en la lógica de *firmware*/*host*.

4.2. Protocolo de aplicación

Se adoptó una trama mínima de **tres bytes** desde PC hacia MicroBlaze:

```
[ led_id , color , cmd ]
```

- `cmd = 1`: fijar color del LED seleccionado (`led_id` en $[0,3]$, `color` en $[0,7]$, 3 bits por LED).
- `cmd = 2`: solicitar lectura de *switches*. La respuesta del MicroBlaze es **2 bytes** con el identificador de periférico en los bits altos y los datos en los 12 bits bajos. Para el caso de *switches*, `device` = 1.

4.3. Firmware en C (MicroBlaze): `protocolo_main.c`

Inicialización. Se configuran los GPIO de entrada/salida y el módulo UART; las direcciones base provienen de `xparameters.h`. Las salidas inician en cero y las entradas se leen en su estado actual.

Recepción de trama. En el bucle principal se leen **3 bytes** de manera bloqueante y se separan en `led_id`, `color` y `cmd`.

Comando `cmd=1` (control de LEDs). Si el identificador es válido $[0,3]$, se construye una máscara de 3 bits por LED, se limpian los bits previos y se cargan los 3 bits del color solicitado; luego se actualiza el registro GPIO.

Comando `cmd=2` (lectura de switches). Se lee el registro de entrada GPIO, se empaqueta la respuesta en 2 bytes (`device` en los bits altos y `data` en los 12 bits bajos) y se transmite cuando el UART está libre.

4.4. Herramienta en Python (host): `TP5_UART_FPGA_pv3.py`

Apertura y menú. El script toma el puerto por parámetro (`argv[1]`), abre la conexión serie y muestra un menú con las acciones `L` (LED), `S` (switches) y `exit`.

Control de LED (`L`). Se valida `led_id` en $[0,3]$ y `color` en $[0,7]$; luego se envía la trama $[led\_id, color, 1]$. Se informa por consola el LED y el color aplicado.

Lectura de switches (`S`). Se envía $[0,0,2]$, se leen **2 bytes** y se decodifican: `device` en los bits altos y `data` en los 12 bits bajos. Se imprime el valor en binario y un desglose legible de `SW0..SW3`. Si la longitud recibida es incorrecta, se notifica el error.

4.5. Resumen de campos y convenciones

Campo	Descripción
`led_id`	Identificador de LED físico, rango $[0, 3]$.
`color`	Código de color en 3 bits ($0 \ldots 7$); se mapea a GPIO con máscara por LED.
`cmd`	1: setear LED; 2: leer *switches*.
`device`	En respuesta, bits altos (valor 1 para *switches*).
`data`	En respuesta, 12 bits con estado de *switches*.

4.6. Decisiones de diseño y consideraciones

- **Trama fija y corta.** Tres bytes hacen más sencilla la **lectura del comando** y evitan delimitadores; la respuesta se fija en 2 bytes para consultas rápidas.
- **Mapeo de color por LED.** Se reservan 3 bits por LED; el color se aplica con máscara y actualización atómica del registro GPIO.
- **Validaciones del lado PC.** Se verifica rango de `led_id` y `color`; se informa al usuario si la lectura no trae 2 bytes.
- **Separación de capas.** UART queda como medio de transporte; el significado de los bytes se define en la capa de aplicación, facilitando futuras extensiones (nuevos `cmd` o `device`).

5. Resultados

5.1. Encendido de Leds y Combinaciones

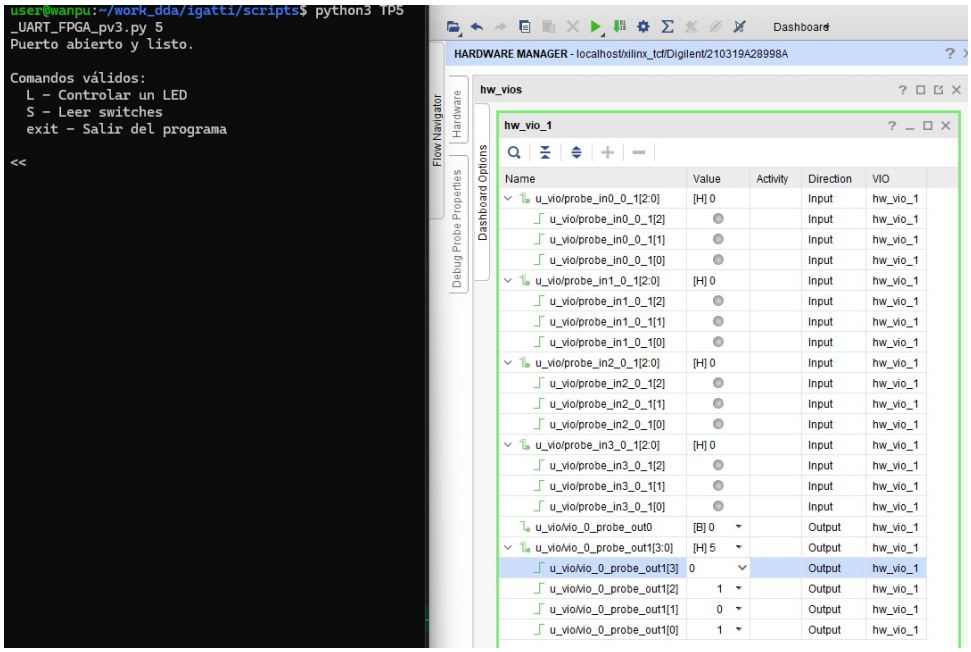

Figura 2: Estado Inicial del Sistema

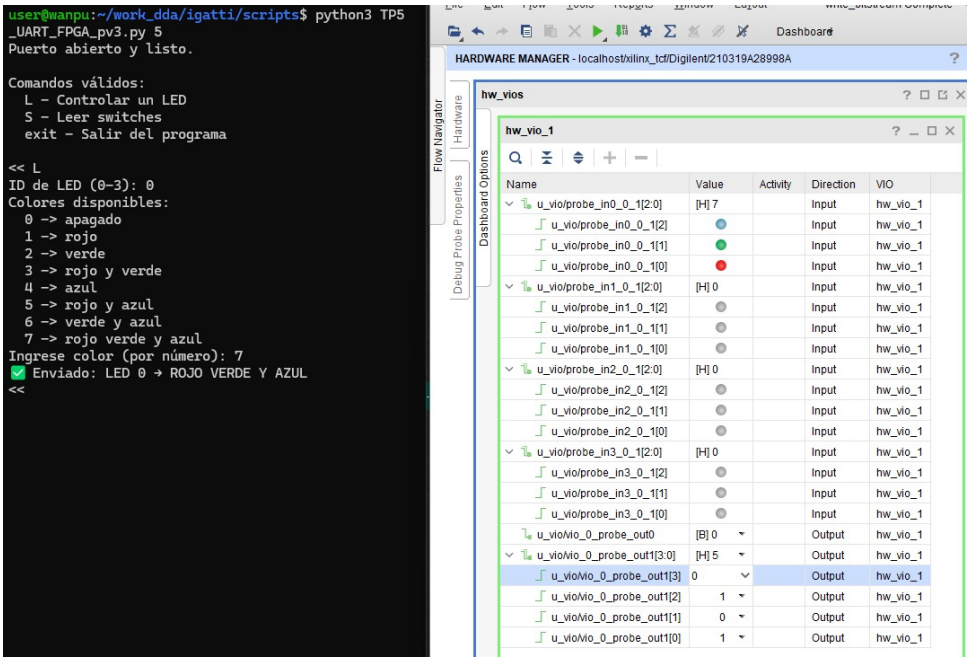

Figura 3: Selección del Led [0] y opción 7 (Rojo,Verde y Azul)

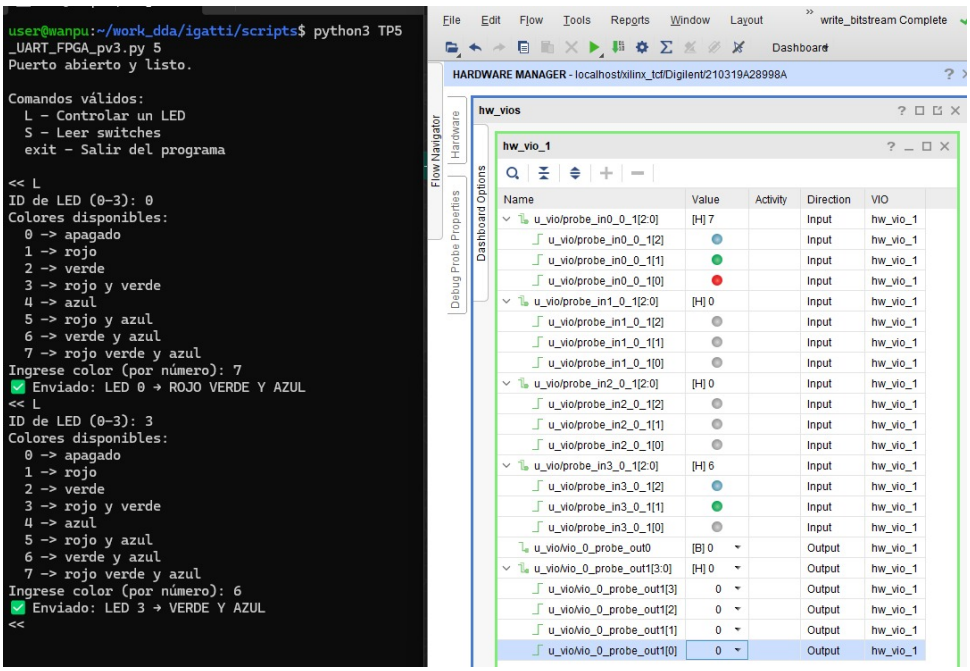

Figura 4: Selección del Led [3] y opción 6 (Verde y Azul)

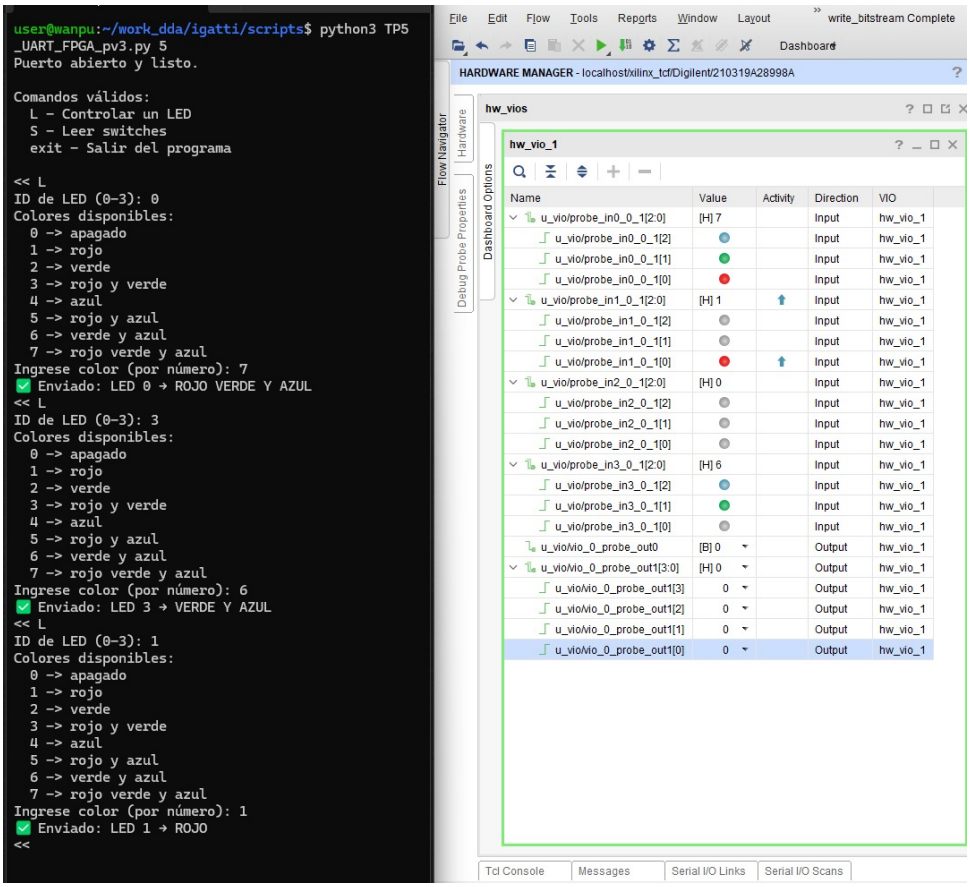

Figura 5: Selección del Led [1] y opción 1 (Rojo

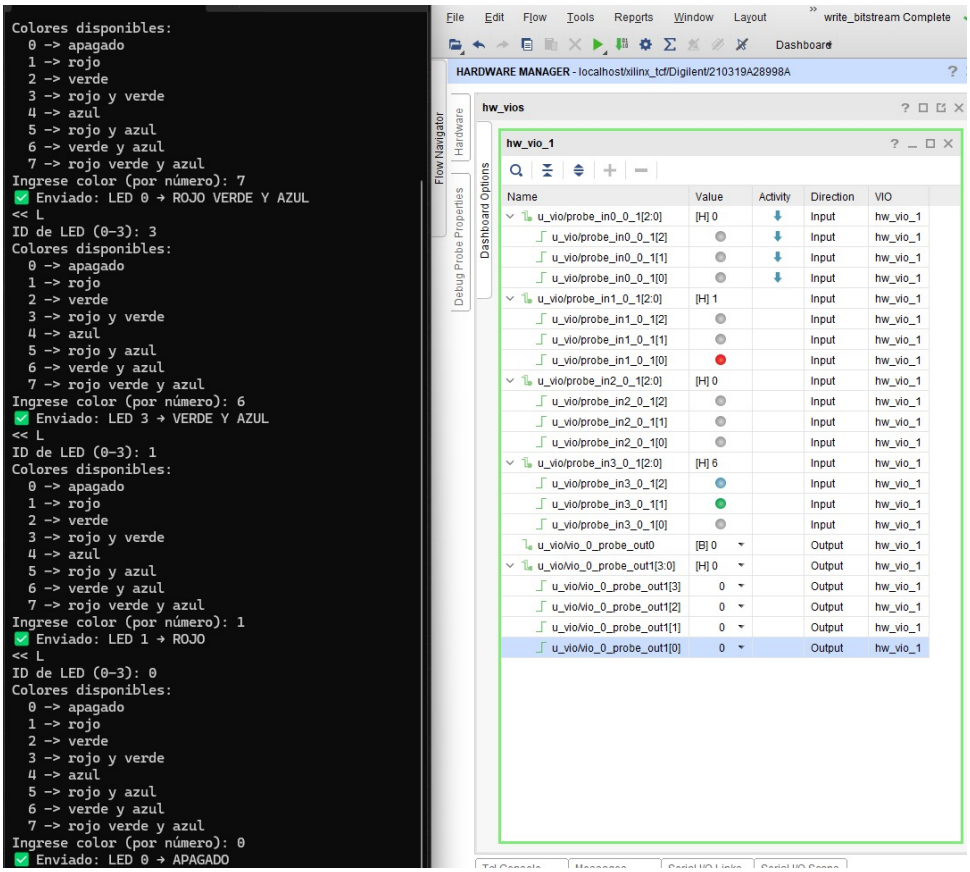

Figura 6: Selección del Led [0] y opción 0 (Apagado)

5.2. Lectura de Switches

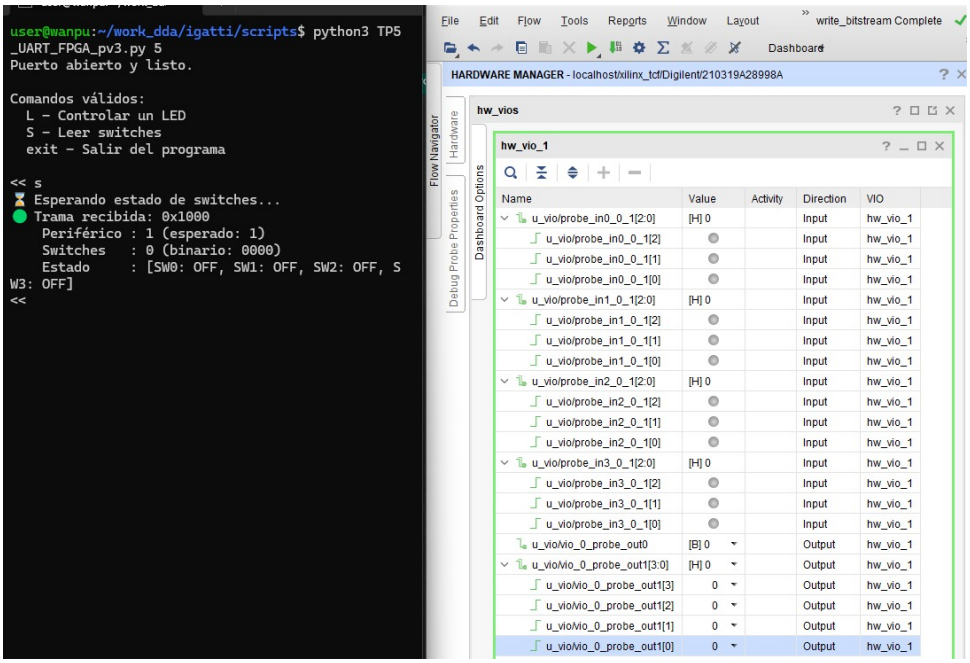

Figura 7: Switches OFF

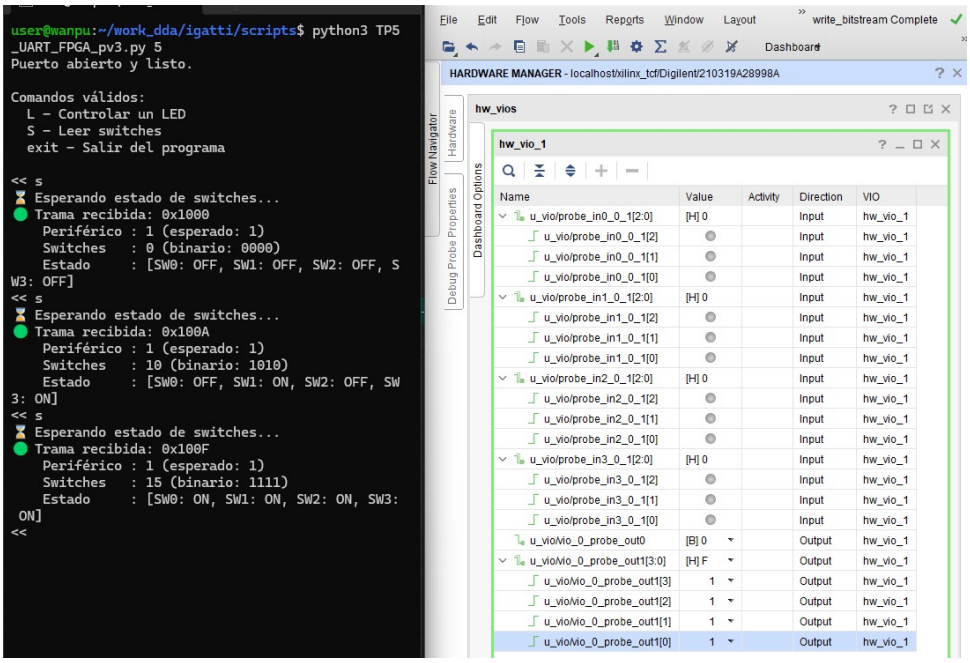

Figura 8: Swithches ON

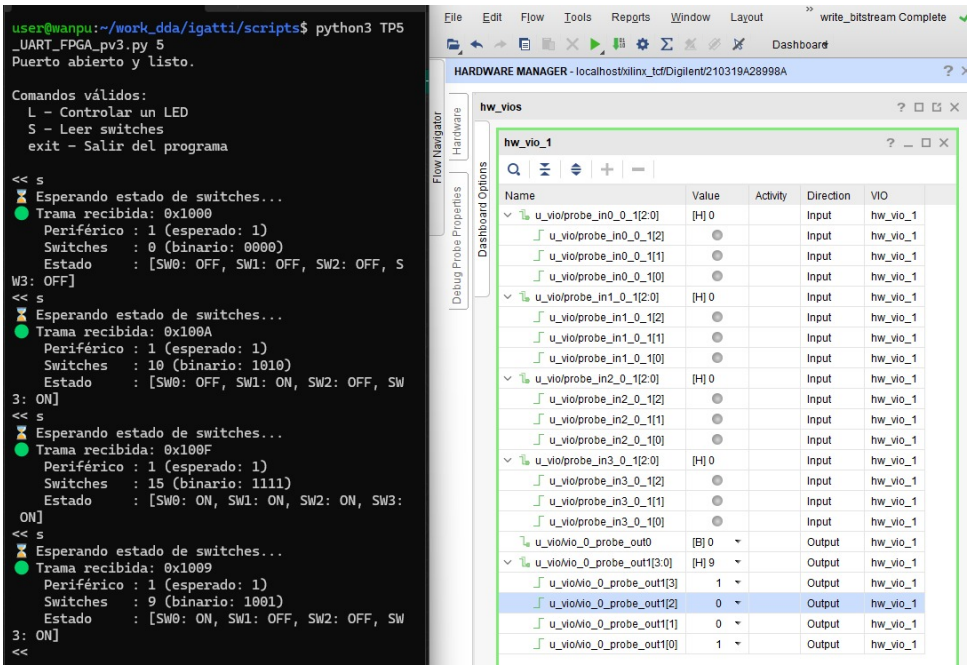

Figura 9: Switches [0] y [3] ON

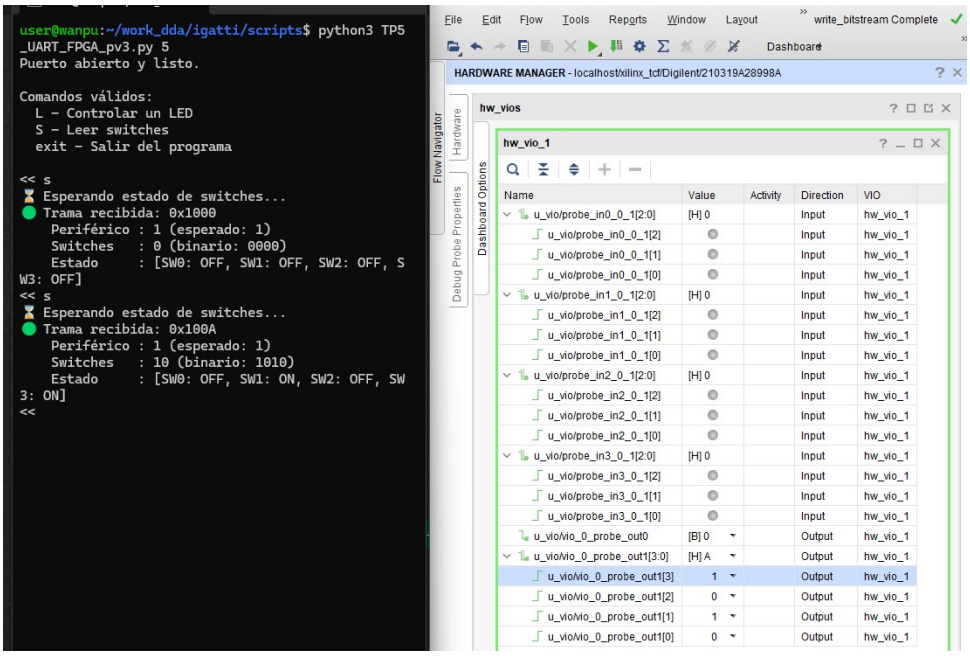

Figura 10: Switches [1] y [3] ON

6. Conclusión

El trabajo permitió cerrar el intercambio entre PC y FPGA mediante una comunicación serie sencilla, combinando un firmware en C sobre MicroBlaze con una herramienta en Python del lado de la PC. Con este esquema se consiguió controlar LEDs y consultar el estado de los *switches*, manteniendo sin cambios el diseño base de Vivado provisto por la Fundación. Más que la funcionalidad puntual, el valor del TP estuvo en integrar contenidos vistos por separado y comprobarlos directamente en la placa.

Desde lo conceptual, se reforzó la separación de responsabilidades: la UART como medio de transporte y, por encima, un protocolo de aplicación claro que define qué significa cada byte. Optar por mensajes breves y de longitud fija simplificó la verificación y deja margen para extender el sistema sin romper lo existente. También se afianzó el uso de GPIO mapeados a memoria, diferenciando con claridad la escritura de salidas (LEDs) y la lectura de entradas (switches).

En la implementación, se tomaron decisiones típicas de sistemas embebidos: validación básica de parámetros en la PC, construcción e *interpretación* de mensajes en el firmware y un manejo por sondeo (*polling*) para mantener el flujo simple. La comprobación se apoyó en casos concretos reproducibles (encendido determinista de un LED, lectura consistente de switches y tratamiento de datos fuera de rango), lo que facilitó aislar problemas y confirmar el comportamiento esperado.

Como aprendizaje general, se destacó que un protocolo mínimo pero ordenado reduce ambigüedades y agiliza la integración entre software y hardware. Documentar las convenciones (IDs, colores, asignación de bits) resultó clave para que ambos extremos del enlace “hablen el mismo idioma”.