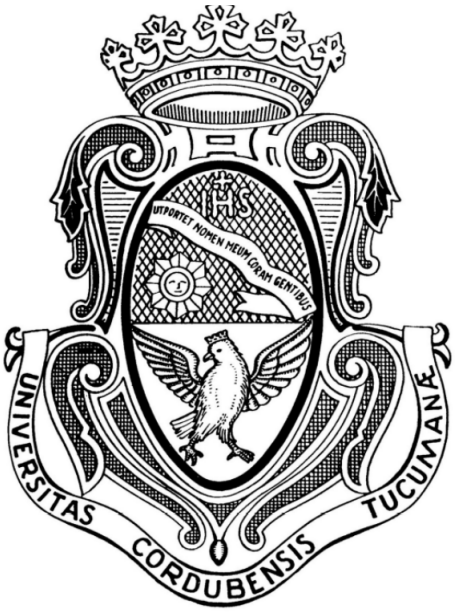
Universidad Nacional de Córdoba



Facultad de Ciencias Exactas, Físicas y Naturales

Cátedra de Arquitectura de Computadoras

Trabajo Práctico 2: UART Full Duplex

Profesores: - Martin Pereyra, Santiago Rodriguez
Integrantes:
Pallardó Agustín - apallardo@mi.unc.edu.ar
Trachta Agustín - agutrachta@mi.unc.edu.ar

19 de octubre de 2025

Índice

1. Introducción

La comunicación serie es uno de los mecanismos más utilizados en sistemas digitales para transmitir información de manera simple y eficiente. Dentro de este esquema, el estándar **UART** (Universal Asynchronous Receiver/Transmitter) constituye una de las soluciones más difundidas debido a su simplicidad de implementación, bajo costo y compatibilidad con una amplia variedad de dispositivos.

En este trabajo se presenta el diseño e implementación de un sistema digital basado en **UART**, desarrollado en lenguaje *Verilog HDL*. El sistema no solo incluye los bloques clásicos de transmisión y recepción, sino que además incorpora:

- **Un generador de baudios** para sincronizar la comunicación.
- **Memorias FIFO** que permiten desacoplar los datos recibidos y transmitidos.
- **Una Unidad Aritmético-Lógica (ALU)** encargada de procesar los operandos recibidos.
- **Un bloque de interfaz** que organiza los bytes provenientes del canal serie, ensambla una instrucción de 32 bits y orienta la secuencia completa: lectura de operandos, ejecución en la ALU, write-back y/o envío por UART.
- **Un decodificador de instrucciones** que clasifica la instrucción (tipo R/I), extrae campos y selecciona el operador correspondiente.
- **Un banco de registros** que provee los operandos leídos por dirección y recibe el resultado bajo señal de escritura.

El objetivo principal es demostrar cómo la arquitectura UART puede extenderse más allá de la simple comunicación, integrándose con módulos de procesamiento para construir sistemas digitales completos. En particular, se implementa un mecanismo mediante el cual se reciben instrucciones por el puerto serie, se decodifican y procesan mediante la ALU con apoyo de un regfile, y los resultados se reenvían al transmisor UART.

El trabajo se organiza de la siguiente manera: primero se describen los módulos fundamentales en orden de ejecución (generador de baudios, receptor, FIFO, transmisor, ALU, interfaz, decoder y regfile). Posteriormente, se analiza la integración en el módulo top, se presentan las simulaciones y pruebas realizadas, y finalmente se discuten los resultados obtenidos. Cabe destacar que la migración desde una interfaz de tres bytes en orden fijo hacia una interfaz basada en comandos con decodificador y regfile constituye mejoras propuestas e implementadas en esta versión.

2. Descripción general del sistema

El diseño implementado corresponde a un sistema UART extendido con capacidades de procesamiento mediante una **Unidad Aritmético-Lógica (ALU)**. El bloque superior `top` integra todos los módulos que lo componen y define el flujo de datos completo desde la entrada serie (`rx`) hasta la salida (`tx`).

2.1. Bloque `top`

El módulo `top` actúa como entidad principal, interconectando:

- El generador de baudios, encargado de derivar la señal de muestreo `sample_tick`.
- El receptor UART (`uart_rx`), que reconstruye bytes a partir de la señal serie entrante.
- Dos memorias FIFO: una para los datos recibidos (FIFO RX) y otra para los datos a transmitir (FIFO TX).
- El decodificador de instrucciones (`rv_decoder`), que clasifica la instrucción (tipo R/I), extrae campos (`rd`, `rs1`, `rs2`, `imm`) y selecciona `alu_op`.
- El banco de registros (`regfile`), que provee operandos por dirección y recibe el resultado en `rd` bajo señal de escritura, preservando el registro nulo.
- El módulo de interfaz (`rv_interface`), que ensambla una instrucción de 32 bits con 4 bytes UART, coordina las lecturas/escrituras del `regfile`, alimenta la ALU, realiza el *write-back* y, si corresponde, genera la respuesta hacia la FIFO TX.
- La ALU, que realiza las operaciones aritméticas y lógicas definidas.
- El transmisor UART (`uart_tx`), que serializa los datos de salida.

2.2. Esquemático general

La Figura 1 muestra un diagrama de bloques típico de un sistema UART completo, donde se observa la integración del generador de baudios, los bloques de transmisión y recepción, y las memorias FIFO.

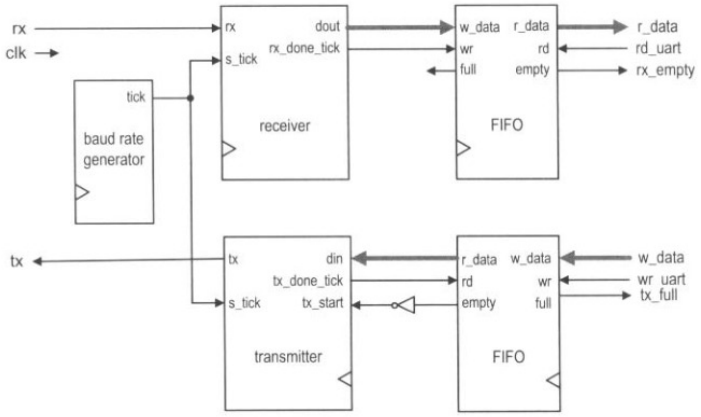


Figure 8.5 Block diagram of a complete UART.

Figura 1: Diagrama de bloques de un UART completo [**?**].

2.3. Flujo de datos

El flujo de información en el sistema sigue la siguiente secuencia:

1. Los datos ingresan por la línea `rx` y son muestreados por el `uart_rx`.
2. Los bytes reconstruidos se almacenan en la **FIFO RX**.
3. La **interfaz** ensambla una **instrucción de 32 bits** a partir de cuatro bytes UART y la entrega al **decodificador**.
4. El **decodificador** determina el tipo (R/I), extrae `rd`, `rs1`, `rs2` e `imm`, y selecciona `alu_op`.
5. La **interfaz** coordina el acceso al **regfile** para leer operandos (`rs1` y, según el caso, `rs2` o `imm`) y **alimenta la ALU**.
6. La **ALU** procesa los operandos y genera un **resultado**.
7. Si corresponde (tipo R/I y `rd`$\neq$ 0), la **interfaz** realiza el **write-back** en el `regfile` (escritura en `rd`).
8. La **interfaz** puede formar una **respuesta** y escribirla en la **FIFO TX**.
9. Finalmente, el `uart_tx` serializa el dato y lo envía por la línea `tx`.

3. Generador de baudios (`baud_gen`)

El primer bloque del sistema es el generador de baudios, cuya función es producir la señal de muestreo necesaria para sincronizar tanto la recepción como la transmisión de datos.

3.1. Principio de funcionamiento

El módulo recibe como entradas el reloj principal de la FPGA (`clk`) y la señal de `reset`. A partir de un divisor interno, genera una señal periódica llamada `sample_tick`, que es utilizada por los módulos `uart_rx` y `uart_tx` para sincronizar sus máquinas de estados.

El cálculo del divisor se realiza en función de la frecuencia de reloj de la FPGA y de la tasa de baudios deseada:

$$DIVISOR = \frac{f_{clk}}{BAUD\_RATE \times 16}$$

donde el factor 16 corresponde al *oversampling*, una técnica habitual en UART que permite mejorar la detección de bits y reducir errores por jitter o pequeñas diferencias de frecuencia.

3.2. Oversampling en UART

En UART asíncrono el emisor y el receptor no comparten reloj, por lo que el receptor debe *reconstruir* el instante de muestreo de cada bit a partir del flanco del *start bit*. Para mejorar la robustez frente a pequeñas desintonías de baudios, jitter y ruido, se utiliza **oversampling**: muestrear cada bit varias veces a una frecuencia M veces superior a la tasa de baudios.

En este diseño se emplea $M = 16$, por lo que el generador de baudios produce una señal de *tick* con:

$$f_{\text{tick}} = M \cdot BAUD\_RATE \qquad \Rightarrow \qquad T_{\text{tick}} = \frac{T_{\text{bit}}}{M}$$

El receptor detecta el flanco descendente del start, espera $\frac{M}{2}$ ticks (centro del start) y a partir de allí toma una muestra cada M ticks, que caen en el **centro** de cada bit de datos. Muestrear en el centro maximiza el margen a distorsiones temporales.

3.3. Flujo interno

- Un contador se incrementa en cada flanco ascendente del reloj principal.
- Cuando el contador alcanza el valor del divisor, se reinicia y se genera un pulso en `sample_tick`.

- Este pulso marca los instantes exactos en que deben muestrearse o transmitirse los bits.

3.4. Importancia en el sistema

El `baud_gen` es esencial porque garantiza que tanto el transmisor como el receptor trabajen bajo la misma temporización. En caso contrario, existiría desalineación entre los bits enviados y los recibidos, ocasionando errores en la reconstrucción de los datos. Gracias a su parametrización, este módulo es flexible y puede adaptarse fácilmente a distintas frecuencias de reloj y diferentes tasas de transmisión.

4. Receptor UART (`uart_rx`)

El receptor es responsable de transformar la señal serie `rx` en un byte paralelo y anunciar cuándo el dato está listo. Emplea **oversampling** a 16× y una máquina de estados finitos (FSM) con temporización por ticks.

4.1. Trama y convenciones

La línea `rx` permanece en nivel alto en reposo. Cada trama se compone de:

- **Start**: 1 bit en nivel bajo.
- **Datos**: `DBIT` = 8 bits, orden *LSB first*.
- **Stop**: 1, 1.5 o 2 bits en alto, parametrizados por `SB_TICK` $\in \{16, 24, 32\}$.

4.2. Señales y contadores internos

- `sample_tick`: pulso de temporización a 16× la tasa de baudios.
- `tick_count`: cuenta los ticks dentro del bit (0..15 para datos).
- `bit_count`: cuenta cuántos bits de datos se han muestreado (0..DBIT-1).
- `rx_shift_reg`: registro de desplazamiento donde se van incorporando los bits recibidos.
- `rx_done_tick`: pulso de un ciclo de `clk` que indica "byte listo".

4.3. Máquina de estados y flujo temporal

La FSM consta de cuatro estados: **IDLE**, **START**, **DATA** y **STOP**. El funcionamiento se describe paso a paso a continuación y se ilustra en la Figura 2.

1) IDLE (espera de start) La línea `rx` está alta. Al detectarse un nivel bajo (*start bit*), se pasa a **START** y se reinicia `tick_count`.

2) START (centrado de muestreo) Con cada `sample_tick` se incrementa `tick_count`. Al alcanzar `tick_count` = 7 (8 ticks desde el flanco), se considera que estamos en el **centro** del start. Entonces:

1. Se pone `tick_count` en 0 para comenzar a cronometrar el primer bit de datos.
2. Se pone `bit_count` en 0.
3. Se transita a **DATA**.

3) DATA (muestreo de 8 bits) Cada vez que `tick_count` = 15 y llega `sample_tick`, se toma la muestra del bit (centro del bit) y se la introduce al MSB del `rx_shift_reg` desplazando a la derecha. Esto, repetido 8 veces con *LSB first*, deja el registro en orden natural $[b7 \dots b0]$. Tras cada muestreo:

- `tick_count` se reinicia a 0.
- `bit_count` se incrementa. Si `bit_count` = `DBIT` − 1, se pasa a **STOP**.

4) STOP (validación y fin de trama) Se esperan `SB_TICK` ticks (por defecto 16 para 1 bit de stop). Al cumplirse:

- Se genera un pulso `rx_done_tick` indicando que el byte en `dout` es válido.
- Se vuelve a **IDLE**.

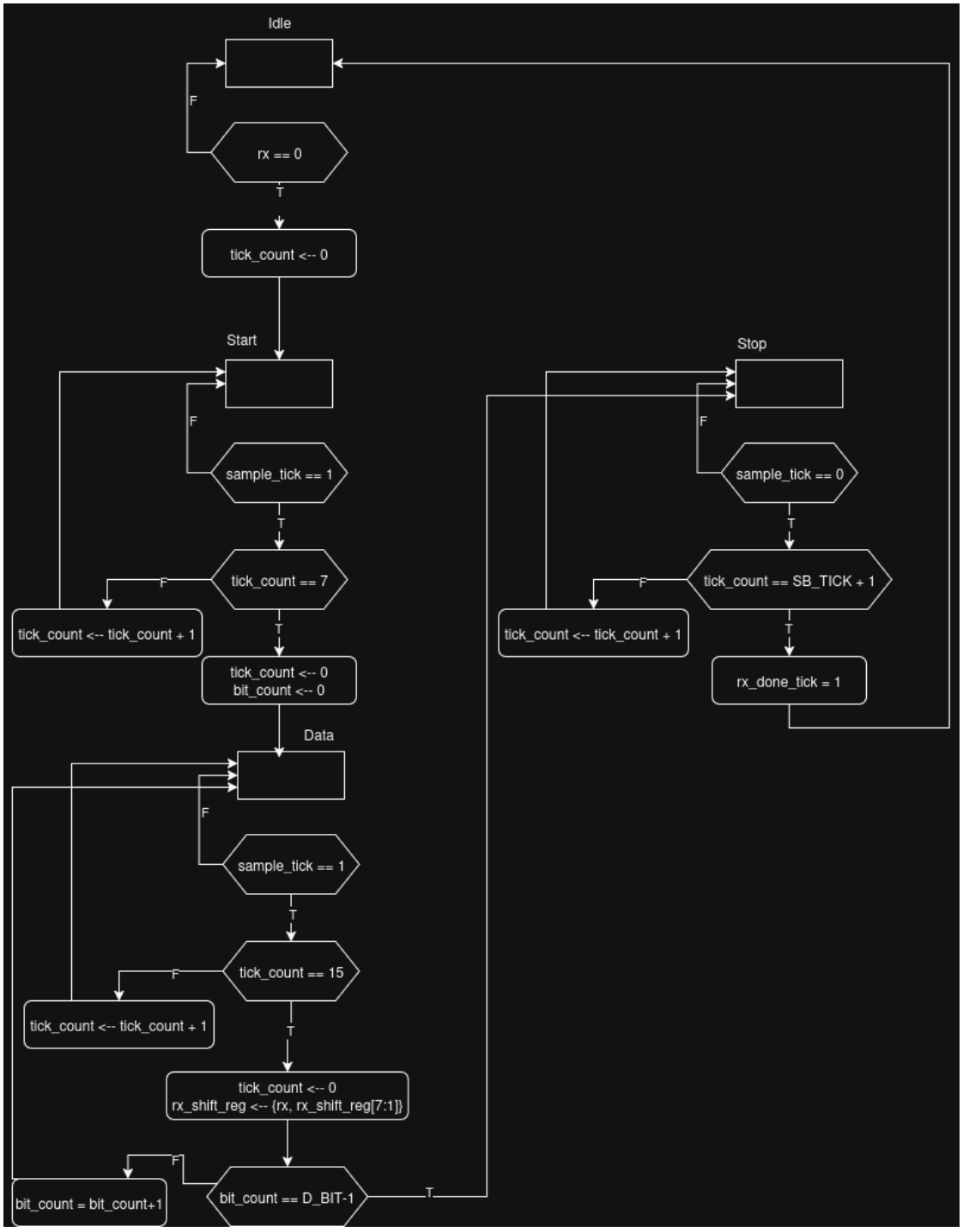


Figura 2: Diagrama de flujo de la FSM del receptor UART con oversampling a 16×.

4.4. Cronometría: instantes de muestreo

Tras detectar el start, el primer muestreo útil ocurre 8 ticks después (centro del start) y luego cada 16 ticks para cada bit de datos. El tiempo total entre el flanco de start y

el anuncio del byte listo es:

$$N_{\text{ticks}} = 8 + 16 \cdot \texttt{DBIT} + \texttt{SB\_TICK}$$

Para `DBIT` $= 8$ y `SB_TICK` $= 16$: $N_{\text{ticks}} = 152$. Como $T_{\text{tick}} = \frac{1}{16 \cdot BAUD\_RATE}$, el retardo es aproximadamente $152 \cdot T_{\text{tick}} \approx 0{,}99\,\text{ms}$ a 9600 bps, coherente con una trama de 10 bits ($\sim 1{,}04\,\text{ms}$), considerando que el muestreo se inicia en el centro del start.

4.5. Parámetros y salidas

- `DBIT`: número de bits de datos (8 en este diseño).
- `SB_TICK`: cantidad de ticks de stop (16, 24 o 32 $\Rightarrow$ 1, 1.5 o 2 bits).
- `dout`: byte paralelo reconstruido a partir de `rx_shift_reg`.
- `rx_done_tick`: pulso de 1 ciclo de `clk` indicando dato válido.

4.6. Notas de diseño y posibles mejoras

- **Sincronización del pin `rx`**: al ser asíncrono respecto de `clk`, es recomendable anteponer un *doble flip-flop* de sincronización para mitigar metastabilidad.
- **Detección de error de trama**: durante **STOP** verificar que `rx` esté alto; si no, reportar *framing error*.

5. Transmisor UART (`uart_tx`)

El bloque `uart_tx` es el encargado de serializar un dato paralelo de 8 bits y enviarlo a través de la línea `tx`, siguiendo la convención de trama UART: un bit de inicio, 8 bits de datos y un bit de parada (o más, según configuración).

5.1. Principio de funcionamiento

El transmisor se activa cuando recibe la señal `tx_start`, lo que indica que existe un byte válido en su registro de entrada (`din`). A partir de ese momento, controla la línea `tx` mediante una máquina de estados finitos (FSM) que sigue la misma temporización definida por los pulsos de `sample_tick` generados por el módulo `baud_gen`.

5.2. Máquina de estados

El transmisor implementa los mismos cuatro estados que el receptor:

- **IDLE**: la línea `tx` permanece en nivel alto. Espera que la señal `tx_start` se active.
- **START**: coloca la línea en nivel bajo durante 16 ticks para indicar el comienzo de la trama.
- **DATA**: envía uno a uno los bits de datos, en orden *LSB first*. Cada bit permanece estable durante 16 ticks.
- **STOP**: fuerza la línea a nivel alto durante `SB_TICK` ticks. Una vez cumplido, emite el pulso `tx_done_tick` y vuelve a **IDLE**.

5.3. Flujo interno

El dato a transmitir se carga en un registro de desplazamiento (`tx_shift_reg`) cuando `tx_start` se activa. En cada ciclo de envío de bit:

- El bit menos significativo del registro se coloca en la línea `tx`.
- Al completarse los 16 ticks, el registro se desplaza a la derecha para preparar el siguiente bit.

5.4. Comparación con el receptor

El `uart_tx` utiliza la misma estructura general que el receptor:

- Ambos cuentan con una FSM con los estados **IDLE**, **START**, **DATA** y **STOP**.

- Ambos utilizan los pulsos de `sample_tick` para sincronizar los instantes de transmisión o muestreo.

Las diferencias clave son:

- El receptor `uart_rx` reconstruye bits desde la línea serie hacia un registro, mientras que el transmisor hace el proceso inverso: toma un byte paralelo y lo serializa.
- El transmisor siempre conoce los tiempos exactos en los que debe cambiar el valor de la línea, mientras que el receptor debe detectar el inicio de la trama y re-alinearse a partir de allí.

5.5. Señales principales

- `din`: dato paralelo de entrada (8 bits).
- `tx`: línea de transmisión serie.
- `tx_start`: pulso que indica al transmisor que debe enviar el byte cargado.
- `tx_done_tick`: pulso de un ciclo de `clk` que indica que la transmisión finalizó.

6. Memoria FIFO (`fifo`)

La FIFO (First-In, First-Out) es el bloque que **desacopla** temporalmente los productores y consumidores de datos. En este proyecto se utilizan dos, como vimos en la figura 1: una a la salida del receptor (**FIFO RX**) y otra a la entrada del transmisor (**FIFO TX**). Su función es absorber ráfagas y tolerar pequeñas desincronizaciones entre módulos.

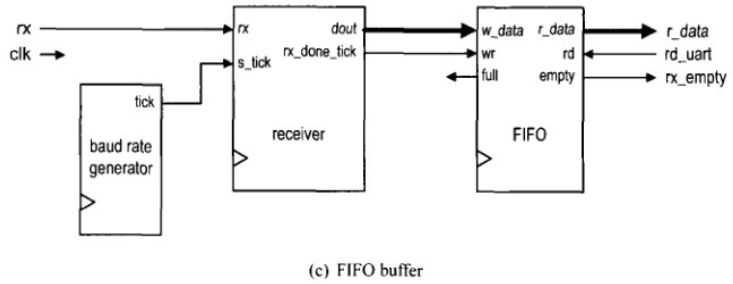


Figura 3: Inserción de una FIFO a la salida del receptor UART.

6.1. Estructura y señales

La FIFO es **sincronizada a un único reloj** (`clk`) y parametrizable:

- **Ancho de palabra** W (bits por elemento).
- **Profundidad** N (cantidad de elementos).

Interfaz:

- `w_data` (entrada de datos), `wr` (pedido de escritura) y `full` (FIFO llena).
- `r_data` (salida de datos), `rd` (pedido de lectura) y `empty` (FIFO vacía).

6.2. Punteros y contador

Internamente mantiene:

- **Puntero de escritura** w_ptr y **puntero de lectura** r_ptr, ambos de ancho $\lceil \log_2 N \rceil$. Al incrementarse, **envolvieron** (wrap-around) de manera circular.
- **Contador de ocupación** *count* en el rango $0 \ldots N$. De él se derivan las banderas:

$$\texttt{empty} \Leftrightarrow count = 0, \qquad \texttt{full} \Leftrightarrow count = N.$$

Nota de diseño: si N *no* es potencia de 2, conviene implementar un *wrap* explícito ($w\_ptr \leftarrow 0$ cuando $w\_ptr = N-1$, idem r_ptr) para garantizar que nunca se indexe fuera de $0 \ldots N-1$.

6.3. Escritura y lectura

Escritura (lado productor). En el flanco ascendente de `clk`, si `wr` está activo y la FIFO **no** está llena, se copia `w_data` en la posición indicada por w_ptr y luego:

$$w\_ptr \leftarrow w\_ptr + 1, \quad count \leftarrow count + 1.$$

Lectura (lado consumidor). La salida `r_data` refleja el contenido de la posición apuntada por r_ptr (*estilo FWFT, first-word fall-through*). Cuando el consumidor afirma `rd` y la FIFO **no** está vacía, en el próximo flanco:

$$r\_ptr \leftarrow r\_ptr + 1, \quad count \leftarrow count - 1.$$

Esto "avanza" la ventana y el siguiente dato queda disponible de inmediato en `r_data` tras ese flanco.

6.4. Lectura y escritura simultáneas

Si en un mismo ciclo se cumple `wr` $\wedge$ $\neg$`full` y `rd` $\wedge$ $\neg$`empty`:

$$w\_ptr \leftarrow w\_ptr + 1, \qquad r\_ptr \leftarrow r\_ptr + 1, \qquad count \text{ no cambia.}$$

La ocupación permanece constante, lo que permite **canalizar** (pipeline) productor y consumidor sin bloqueo.

6.5. Reset y mapeo en hardware

Cuando se activa la señal de `reset`, la FIFO vuelve a su estado inicial:

- Los punteros de lectura y escritura se ponen en cero.
- El contador de ocupación también se pone en cero.

Esto asegura que la memoria comience **vacía**.

6.6. Manejo correcto del *handshake*

- El **productor** debe escribir solo cuando $\neg$`full`.
- El **consumidor** debe leer solo cuando $\neg$`empty`.
- Las banderas `full`/`empty` evitan *overflow/underflow*.

Uso en el receptor (FIFO RX)

A la salida del `uart_rx`, cada vez que se completa una trama se genera `rx_done_tick`; ese pulso se utiliza como `wr` de la FIFO RX y el byte recibido ingresa como `w_data`. El bloque `interface` actúa como **consumidor**: monitorea `empty`; cuando hay datos, aserta `rd` (señal `rd_uart` en el diseño) para avanzar y tomar el próximo byte en `r_data`. Así, el receptor puede seguir capturando tramas aunque el procesamiento sea más lento.

Uso en el transmisor (FIFO TX)

La `interface` escribe en la FIFO TX el **resultado** de la ALU (`wr_uart` como `wr`), siempre que ¬`full`. El `uart_tx` es el **consumidor**: cuando termina de enviar un byte, aserta `tx_done_tick` que hace de `rd` para tomar automáticamente el siguiente. En el `top`, la señal `tx_start` se activa mientras la FIFO TX **no** esté vacía (`tx_start` ← ¬`tx_empty`), logrando una transmisión continua si hay datos en cola.

7. Unidad Aritmético-Lógica (ALU)

La ALU es el bloque encargado de realizar operaciones aritméticas y lógicas sobre dos operandos de 8 bits. Se controla mediante un código de operación de 6 bits que determina qué función ejecutar.

7.1. Operaciones soportadas

Código (binario)	Operación	Descripción
100000	ADD	Suma de `A + B`
100010	SUB	Resta `A - B`
100100	AND	AND bit a bit
100101	OR	OR bit a bit
100110	XOR	XOR bit a bit
000011	SRA	Desplazamiento aritmético a derecha
000010	SRL	Desplazamiento lógico a derecha
100111	NOR	NOR bit a bit

Cuadro 1: Operaciones soportadas por la ALU.

7.2. Rol en el sistema

La ALU recibe como entradas dos datos (`data_a`, `data_b`) y un código de operación (`op`), generando un resultado de 8 bits que se almacena en la FIFO de transmisión. Esto permite extender la comunicación UART más allá de la simple transferencia de bytes, agregando capacidad de procesamiento en hardware.

8. Interfaz, decoder y regfile

8.1. Máquina de estados de la interfaz

La interfaz se rediseñó para recibir cuatro bytes por UART, ensamblar una instrucción de 32 bits y despachar su ejecución a través del *decoder*, el *regfile* y la ALU. La FSM emplea los siguientes estados:

```
localparam S_IDLE = 3'd0,
           S_I0   = 3'd1,
           S_I1   = 3'd2,
           S_I2   = 3'd3,
           S_I3   = 3'd4,
           S_EX   = 3'd5,
           S_WB   = 3'd6,
           S_TX   = 3'd7;
```

- **S_IDLE (reposo).** Estado de espera tras *reset*. Si la bandera de FIFO RX indica disponibilidad de dato (`~rx_empty`), se inicia la secuencia de lectura y se transita a **S_I0**.
- **S_I0, S_I1, S_I2 (fetch de bytes 0–2).** En cada uno de estos estados, cuando hay dato válido en FIFO RX y se aserta `rd_uart`, se lee un byte y se almacena en el *slice* correspondiente de la instrucción (LSB primero). Cada estado avanza al siguiente: **S_I0**→**S_I1**→**S_I2**.
- **S_I3 (fetch del byte 3).** Se captura el cuarto byte (MSB) y, con la instrucción completa (`instr`[31:0]), se avanza a **S_EX** para ejecutar.
- **S_EX (ejecución).** Se habilita el flujo hacia el *decoder* para clasificar la instrucción y obtener campos (`rd`, `rs1`, `rs2`, `imm`) y el selector `alu_op`. Con esta información, la interfaz coordina la lectura de operandos del *regfile* y presenta `alu_a`, `alu_b` y `alu_op` a la ALU. El estado siguiente es **S_WB**.
- **S_WB (write-back).** Una vez estable el resultado de la ALU, si la instrucción es de tipo R/I y el destino es distinto de $x0$, se habilita la escritura del *regfile* (`we`, `waddr`=`rd`, `wdata`=`alu_result`). Luego se transita a **S_TX**.
- **S_TX (respuesta).** Si la FIFO de transmisión (**FIFO TX**) no está llena, se inserta un byte de respuesta (`w_data`, típicamente `alu_result`[7:0]) mediante un pulso `wr_uart`. Concluida la inserción, la FSM retorna a **S_IDLE**.

8.2. Bloque secuencial de ensamblado de instrucción

El ensamblado de la palabra de 32 bits se realiza LSB→MSB, tal como se indica:

```
case (state)
  S_I0: if (!rx_empty && rd_uart) instr[7:0]   <= r_data;   // byte 0 (LSB)
  S_I1: if (!rx_empty && rd_uart) instr[15:8]  <= r_data;   // byte 1
  S_I2: if (!rx_empty && rd_uart) instr[23:16] <= r_data;   // byte 2
  S_I3: if (!rx_empty && rd_uart) instr[31:24] <= r_data;   // byte 3 (MSB)
  default: ;
endcase
```

Un segundo bloque secuencial independiente actualiza el registro de estado (`state`) con `state_n` en el flanco de reloj, desacoplando la lógica combinacional de siguiente estado del almacenamiento sincrónico.

8.3. Decodificación de la instrucción

El *decoder* clasifica la instrucción observando `opcode`/`funct3`/`funct7` y expone:

- Formato: `is_rtype` (registro–registro) o `is_itype` (registro–inmediato).
- Campos: `rd`, `rs1`, `rs2`; e `imm_i` (signado) para tipo I.
- Selector de operación de ALU: `alu_op`.

En particular, si `opcode` = `7'b0110011` la instrucción es **tipo R** (ALU reg–reg), y si `opcode` = `7'b0010011` es **tipo I** (ALU reg–imm). El *decoder* no ejecuta; únicamente clasifica y provee los campos para la etapa de ejecución.

8.4. Camino de datos: *regfile*, ALU y respuesta

Con los campos del *decoder*, la interfaz direcciona el *regfile* para leer operandos:

- Tipo R: `alu_a` ← `rdata_a`(`rs1`), `alu_b` ← `rdata_b`(`rs2`).
- Tipo I: `alu_a` ← `rdata_a`(`rs1`), `alu_b` ← `imm_i` (ajustado al ancho de dato).

La ALU computa el resultado en función de `alu_op`. En **S_WB**, si `rd` $\neq 0$ y la instrucción es R/I, la interfaz habilita la escritura del *regfile* con `alu_result`. En **S_TX**, la interfaz inserta en FIFO TX un byte de respuesta (si el protocolo lo requiere) y retorna a **S_IDLE**.

8.5. Observaciones de diseño

- La política LSB→MSB en el ensamblado asegura compatibilidad con el flujo de bytes UART.
- Es recomendable registrar `alu_result` al final de **S_EX** para garantizar estabilidad temporal en **S_WB**/**S_TX**.
- Como mejora futura, puede añadirse un temporizador de *timeout* durante **S_I0**–**S_I3** para descartar instrucciones incompletas, junto con un mecanismo básico de verificación de integridad.

9. Simulación del *testbench*

Previo a la verificación física en la FPGA, se realizó una simulación funcional del sistema completo utilizando **Vivado**. El objetivo fue comprobar el correcto flujo de datos desde la recepción serie hasta la ejecución de la instrucción por la ALU y el almacenamiento del resultado en el banco de registros.

9.1. Configuración de la simulación

El *testbench* inicializa las señales principales:

- `clk`: señal de reloj principal.
- `reset`: reinicio sincrónico del sistema.
- `rx`: entrada serie simulada, donde se inyectan las tramas UART correspondientes a las instrucciones RISC-V.
- `tx`: línea de salida, por donde se observa el resultado emitido por la FPGA.

También se parametrizan los valores de:

- Frecuencia del reloj: `CLK_FREQ = 100 MHz`.
- Tasa de baudios: `BAUD_RATE = 9600`.
- Tiempo por bit: `BIT_TIME_NS = 104166.6 ns`.

9.2. Ejecución de instrucciones

Durante la simulación, el *testbench* envía tres instrucciones consecutivas:

1. **ADDI x1, x0, 5** – carga el valor 5 en el registro `x1`.
2. **ADDI x2, x0, 10** – carga el valor 10 en el registro `x2`.
3. **ADD x3, x1, x2** – suma ambos registros y almacena el resultado (15) en `x3`.

A medida que cada instrucción se reconstruye desde la entrada serie, el **decoder** identifica los campos `opcode`, `funct3`, `rs1`, `rs2`, `rd` e `imm`, generando las señales de control adecuadas para el flujo interno.

9.3. Análisis de señales

En la Figura 4 se muestran las principales señales observadas en la simulación:

- `rx_dout[7:0]` representa los bytes recibidos por la línea serie. Se distinguen los valores `0x05`, `0x0A` y `0x0F` correspondientes a los resultados de las tres operaciones.

- `alu_result[7:0]` muestra el valor calculado por la ALU: primero 5, luego 10 y finalmente 15.
- `rf_wdata[7:0]` confirma la escritura en el banco de registros durante el estado de *write-back*, coincidiendo con los valores de la ALU.
- Las direcciones de registro `rd`, `rs1` y `rs2` se actualizan en sincronía con el flujo de instrucciones, evidenciando la coordinación del módulo `rv_interface`.

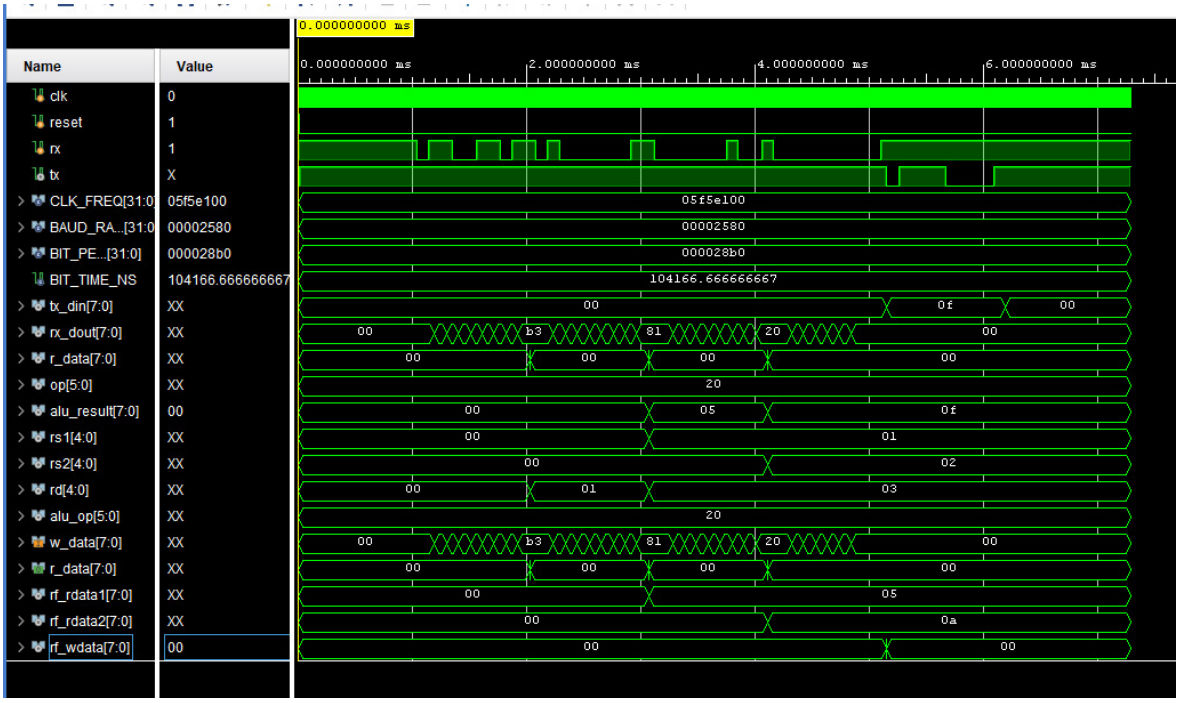


Figura 4: Simulación del *testbench* en Vivado mostrando la ejecución secuencial de las instrucciones ADDI y ADD.

9.4. Validación funcional

La simulación confirma que:

- El receptor UART decodifica correctamente las tramas y ensambla los bytes en orden **little-endian**.
- La interfaz coordina exitosamente la lectura y escritura de registros.
- La ALU realiza operaciones aritméticas sin errores de temporización ni de sincronismo.
- La salida `tx` emite los resultados esperados tras cada instrucción, respetando el protocolo UART 8N1.

En conjunto, la simulación demuestra la correcta integración de todos los bloques del sistema, validando el diseño antes de su implementación física.

10. Prueba funcional del sistema completo

Una vez implementados todos los módulos del sistema UART con ALU, se realizó una prueba funcional conectando la FPGA a una computadora mediante un conversor USB–UART (CP2102). El objetivo fue validar el flujo completo de ejecución de instrucciones RISC-V codificadas en 32 bits, desde la recepción por UART hasta la ejecución en hardware y la devolución del resultado.

10.1. Procedimiento de prueba

Para facilitar la interacción con la FPGA se desarrolló un script en `Python`, denominado `uart_rv_client.py`. Este script utiliza la biblioteca `pyserial` para detectar automáticamente los puertos seriales disponibles, permitir la selección del puerto activo y enviar una secuencia de tres instrucciones codificadas:

1. **ADDI x1, x0, 5**
   Carga el valor inmediato 5 en el registro `x1`.
2. **ADDI x2, x0, 10**
   Carga el valor inmediato 10 en el registro `x2`.
3. **ADD x3, x1, x2**
   Suma los registros `x1` y `x2`, almacenando el resultado (15) en `x3`.

Cada instrucción se empaqueta en formato **little-endian** (byte menos significativo primero) y se envía por UART a 9600 baudios, 8 bits de datos, sin paridad y 1 bit de stop (8N1). El script espera una respuesta por cada instrucción ejecutada, la cual proviene directamente de la ALU a través de la FIFO de transmisión.

10.2. Resultados obtenidos

En la Figura 5 se muestra la salida obtenida en consola. El sistema recibe correctamente las tres instrucciones y devuelve los resultados esperados:

- `Resultado recibido: 5` para la primera instrucción (`ADDI x1, x0, 5`);
- `Resultado recibido: 10` para la segunda (`ADDI x2, x0, 10`);
- `Resultado recibido: 15` para la tercera (`ADD x3, x1, x2`).

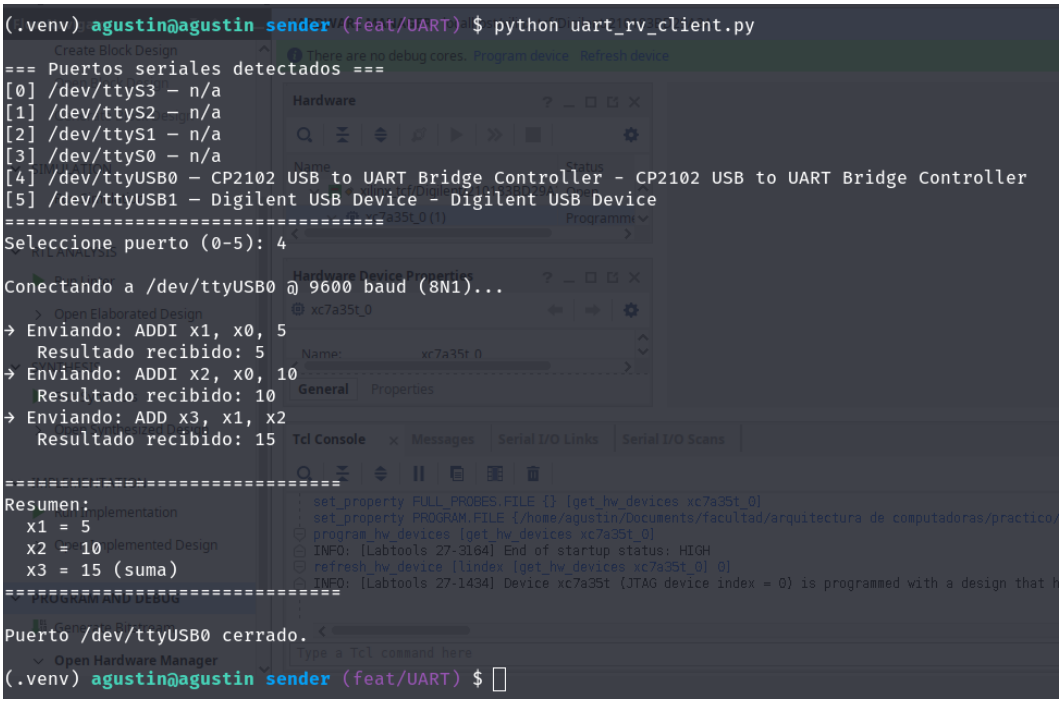


Figura 5: Ejecución de prueba: tres instrucciones enviadas por UART y resultados recibidos desde la FPGA.

10.3. Análisis

El resultado final demuestra que:

- El flujo UART–FIFO–`interface`–`decoder`–`regfile`–`ALU`–UART funciona de forma coherente y sincronizada.
- Los registros de propósito general (`x1`, `x2`, `x3`) son correctamente escritos y leídos durante las etapas de ejecución y *write-back*.
- La ALU ejecuta correctamente operaciones tipo `ADDI` y `ADD`.
- El sistema puede ampliarse fácilmente para incorporar más instrucciones o tasas de baudios distintas, manteniendo la estructura modular.

10.4. Captura del analizador lógico

Para corroborar el intercambio de datos a nivel físico se utilizó el software **Logic 2** con un analizador Saleae, monitoreando las líneas `tx` y `rx` del sistema. En la Figura 6 se observan tres pares de tramas UART correspondientes a las instrucciones enviadas y las respuestas de la FPGA:

- En la primera transmisión se envía la instrucción `ADDI x1, x0, 5`, y se observa la respuesta con el valor `5`.
- En la segunda, la instrucción `ADDI x2, x0, 10` y su resultado `10`.

- En la tercera, `ADD x3, x1, x2`, cuyo resultado `15` confirma la suma correcta de los operandos.

Las tramas fueron decodificadas correctamente a 9600 bps (8N1). El tiempo entre el envío de una instrucción y la respuesta de la FPGA es del orden de pocos milisegundos, demostrando la eficiencia del flujo interno de la interfaz.

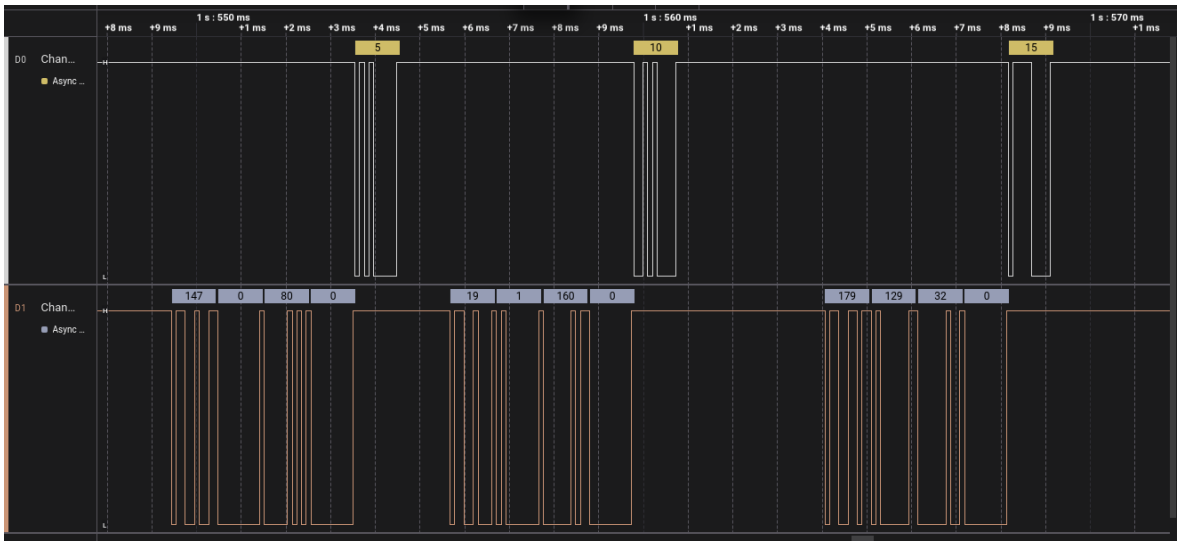


Figura 6: Captura del analizador lógico en Logic 2 mostrando la secuencia de tramas UART de transmisión (D1) y recepción (D0).