Arithmetic Logic Unit (ALU) de 4 bits

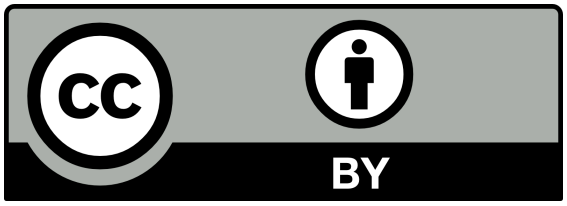
Trabajo Final de Microarquitecturas y Softcores

Agustín Jesús Vazquez

(vazqueza193@gmail.com)

24/06/2025

versión A



Historial de cambios

Versión	Fecha	Descripción	Autor	Revisores
A	24/06/2025	Versión Original	Agustin Vazquez	

Índice de contenido

1. Unidad aritmético lógica (ALU)

Una Unidad aritmética lógica (ALU) es un circuito digital que se utiliza para realizar operaciones aritméticas y lógicas. Es uno de los componentes esenciales en los microcontroladores. Es capaz de realizar las siguientes operaciones:

1. Suma
2. Resta
3. Multiplicación
4. División
5. Desplazamiento
6. Operaciones lógicas (AND, OR, NOT y XOR)

El uso de FPGAs permite diseñar ALUs personalizadas que se ajustan a las necesidades específicas de una aplicación. Esto se hace utilizando lenguajes de descripción de hardware como VHDL. En la *figura 1* se muestra la representación de la ALU de 4 bits que tiene como entrada dos parámetros (A y B) y como salida se obtiene el resultado (Result) de la operación que se le indicó realizar.

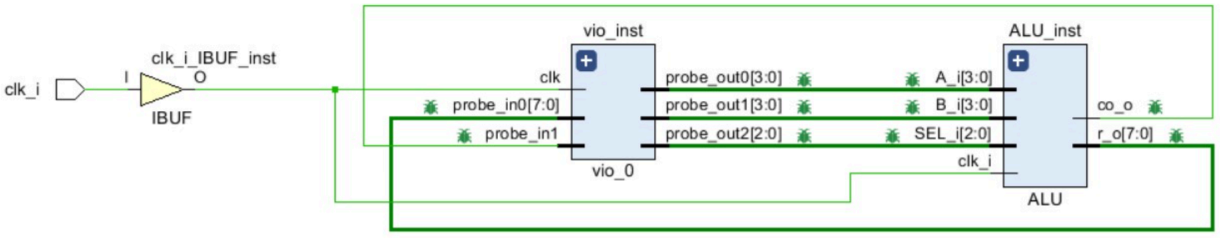


Figura 1. Diagrama de bloques de la ALU de 4 bits.

1.1. Selector de operaciones

Para seleccionar qué operación aritmética se quiere realizar se utiliza un número de operación. Las operaciones que podemos realizar según el valor indicado se indican en la *Tabla 1*.

Tabla 1. Operaciones de la ALU.

ALU_Sel	Operación	Resultado
000	Suma	A + B
001	Resta	A - B

010	Multiplicación	A * B
011	División	A / B
100	Módulo	A % B
101	Desplazamiento	A >> B

2. Componentes principales

2.1. Suma de 4 bits

Realiza la suma de dos números de 4 bits con acarreo. La *figura 2* detalla los archivos utilizados para la simulación en el software *Modelsim*.

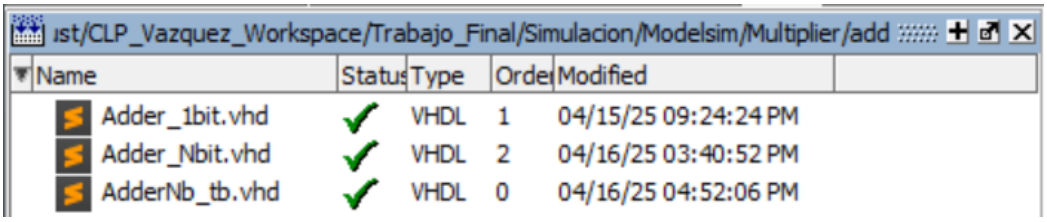


Figura 2. Banco de pruebas de la operación de suma realizada en Modelsim.

addNb es el sumador de N bits; para N = 4 instancia cuatro sumadores de 1 bit (**add1b**) en cascada, propagando el acarreo de un bit al siguiente y generando un acarreo final. En la *figura 3* puede observarse el resultado de la simulación, cuyos casos de prueba fueron redactados en el testbench.

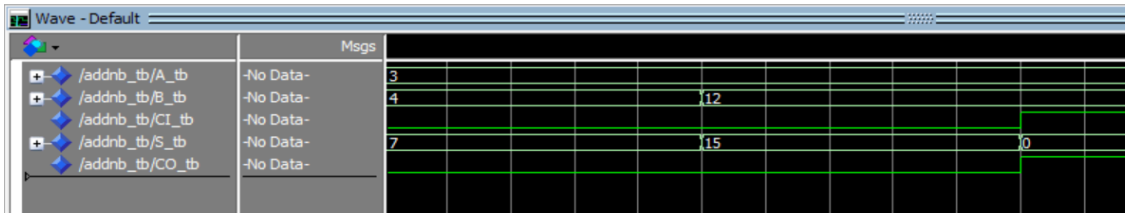


Figura 3. Simulación del componente Adder_Nb.

2.2. Resta de 4 bits

Calcula la diferencia de dos números de 4 bits y detecta el préstamo. La *figura 4* detalla los archivos utilizados para la simulación en el software *Modelsim*.

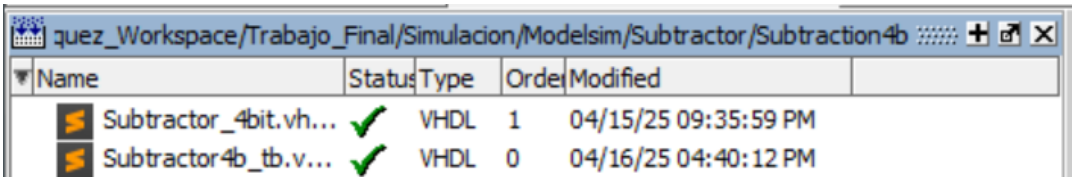


Figura 4. Banco de pruebas de la operación de resta realizada en Modelsim.

sub4b convierte ambos operandos a unsigned de 5 bits (añadiendo un 0 MSB), efectúa la resta y usa el quinto bit resultante para indicar si hubo préstamo (underflow). En la *figura 5* puede observarse el resultado de la simulación, cuyos casos de prueba fueron redactados en el testbench.

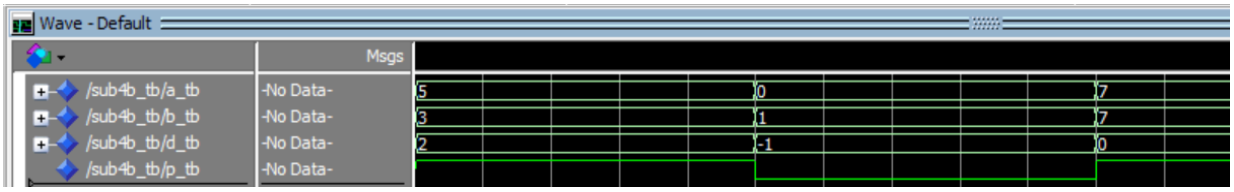


Figura 5. Simulación del componente Subtractor_4bit.

2.3. Multiplicación de 4 bits

Obtiene un producto de hasta 8 bits mediante sumas parciales. La *figura 6* detalla los archivos utilizados para la simulación en el software *Modelsim*.

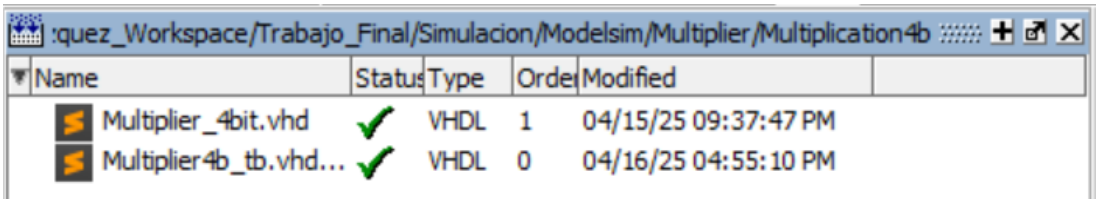


Figura 6. Banco de pruebas de la operación de multiplicación realizada en Modelsim.

`mult4b` aplica el algoritmo “shift-and-add”: por cada bit ‘1’ del multiplicador desplaza el multiplicando la posición correspondiente y suma el resultado al acumulador de 8 bits. En la *figura 7* puede observarse el resultado de la simulación, cuyos casos de prueba fueron redactados en el testbench.

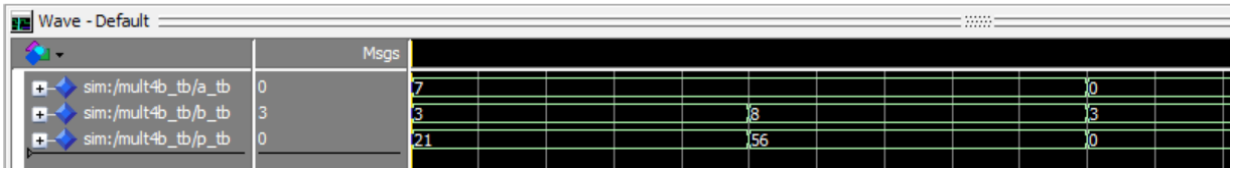


Figura 7. Simulación del componente Multiplier_4bit.

2.4. División de 4 bits

Calcula el cociente y el resto de la división entera de dos números de 4 bits. La *figura 8* detalla los archivos utilizados para la simulación en el software *Modelsim*.

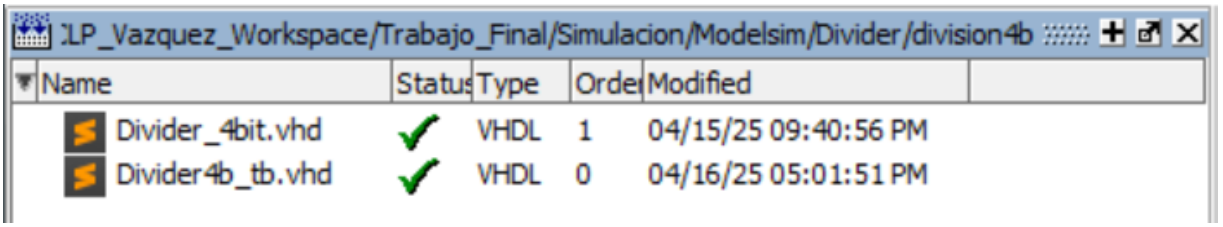


Figura 8. Banco de pruebas de la operación de división realizada en Modelsim.

`div4b` implementa un divisor secuencial sincronizado: en cada ciclo de reloj resta el divisor al resto parcial y, según el signo, ajusta el bit de cociente correspondiente; incluye detección de división por cero. En la *figura 9* puede observarse el resultado de la simulación, cuyos casos de prueba fueron redactados en el testbench.

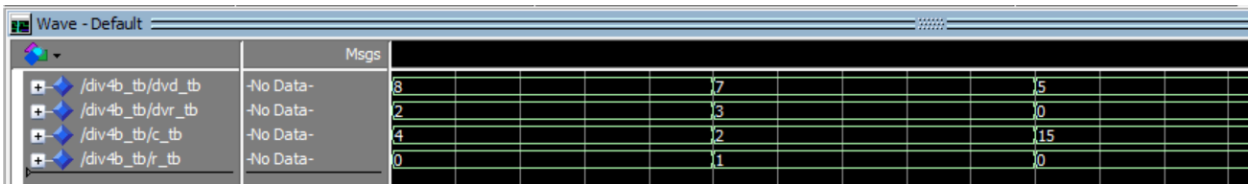


Figura 9. Simulación del componente Divider_4bit.

2.5. Desplazamiento de 4 bits

Desplaza lógicamente un vector de 4 bits a la derecha o a la izquierda. La *figura 10* detalla los archivos utilizados para la simulación en el software *Modelsim*.

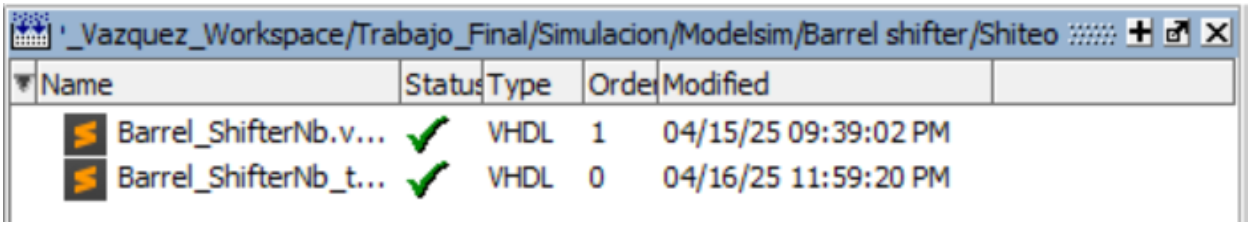


Figura 10. Banco de pruebas de la operación de desplazamiento realizada en Modelsim.

`bShifterNb` convierte la entrada a unsigned y usa la función `shift_right` con el valor de desplazamiento para mover los bits, rellenando con ceros los espacios vacíos.

En la *figura 11* puede observarse el resultado de la simulación, cuyos casos de prueba fueron redactados en el testbench.

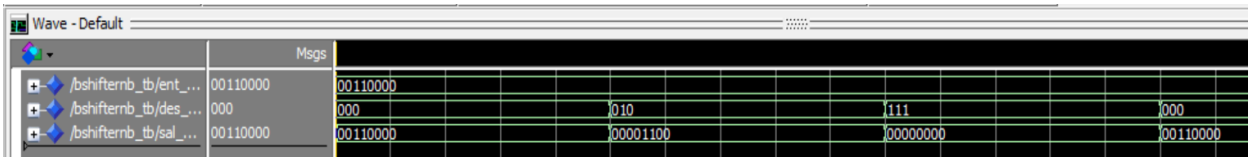


Figura 11. Simulación del componente Barrel_ShifterNb.

3. Proyecto ALU con IP Core

El presente trabajo consiste en encapsular una Unidad Aritmética Lógica (ALU) dentro de un IP Core personalizado para su implementación en una FPGA utilizando la plataforma Xilinx Zynq. Este IP Core permite la reutilización eficiente del diseño y facilita la interacción entre el procesador embebido ARM y la lógica personalizada.

3.1. Descripción del diseño

La ALU diseñada ofrece operaciones básicas como suma, resta, multiplicación, división, módulo y operaciones de desplazamiento (shifting). Estas operaciones fueron encapsuladas en un IP Core utilizando Vivado, configurándose con una interfaz AXI para facilitar la comunicación con el procesador ARM embebido.

La figura 12 muestra la configuración del IP Core personalizado. Se especificaron parámetros críticos como:

- **Ancho de datos**: 32 bits.
- **Ancho de dirección**: 4 bits.
- **Direcciones base y alta** definidas por defecto.

Esta configuración permite integrar el IP Core adecuadamente en el diseño general del sistema basado en FPGA.

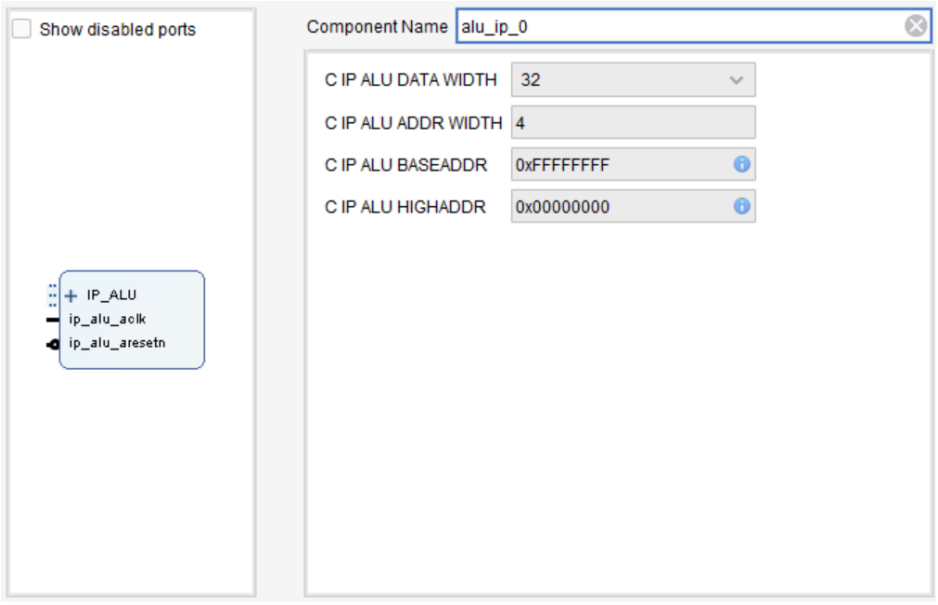


Figura 12. IP Core personalizado.

3.2. Implementación en Vivado

La *figura 13* muestra el diagrama de bloques generado en Vivado para el sistema completo. Este sistema integra:

- **Processing System ZYNQ7**: que aloja el procesador ARM.
- **AXI Interconnect**: encargado de gestionar las conexiones AXI entre el procesador y el IP Core.
- **IP Core ALU personalizado**: conectado mediante interfaces AXI (puertos **`ip_alu_aclk`** y ***`ip_alu_aresetn`***).

La comunicación se establece mediante una interfaz AXI4-Lite, facilitando la escritura y lectura de registros internos del IP Core.

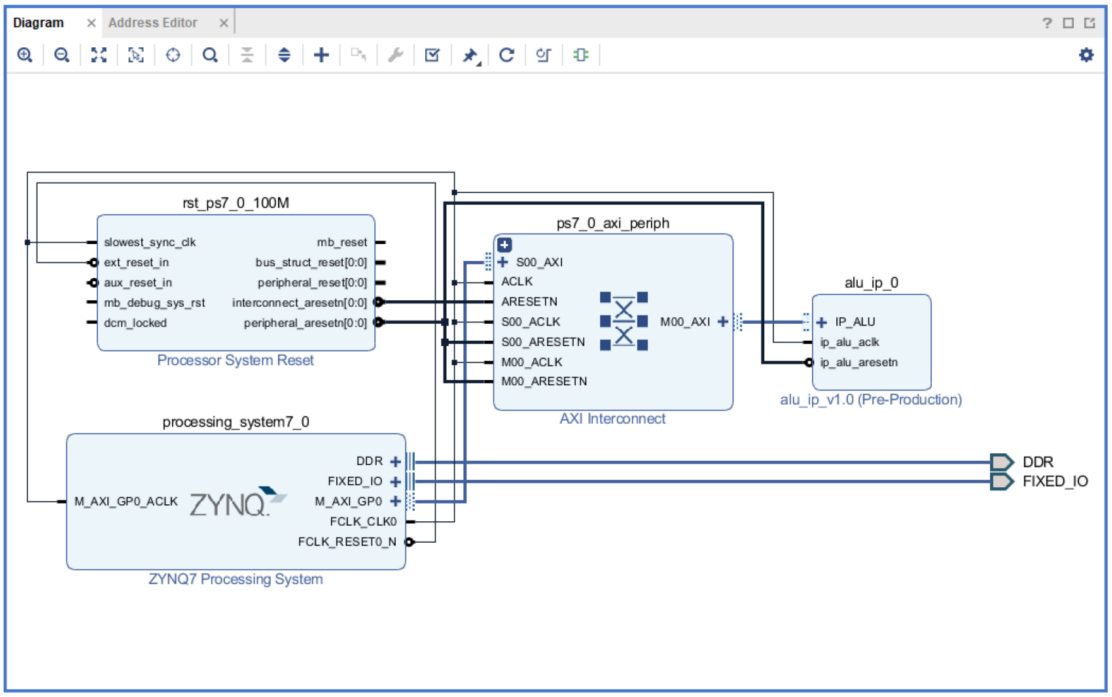


Figura 13. Diagrama de bloques de la ALU.

3.3. Implementación en Vivado

La *figura 14* muestra la implementación del código en C en la plataforma SDK de Xilinx. El software desarrollado realiza una interacción con el IP Core mediante la escritura en sus registros. Se emplearon funciones específicas de escritura (`ALU_IP_mWriteReg`) generadas automáticamente por el entorno, facilitando la interacción con el IP Core.

Se definieron operaciones enumeradas (`operation_t`) para mejorar la legibilidad del código, permitiendo escribir claramente cada operación en el IP Core.

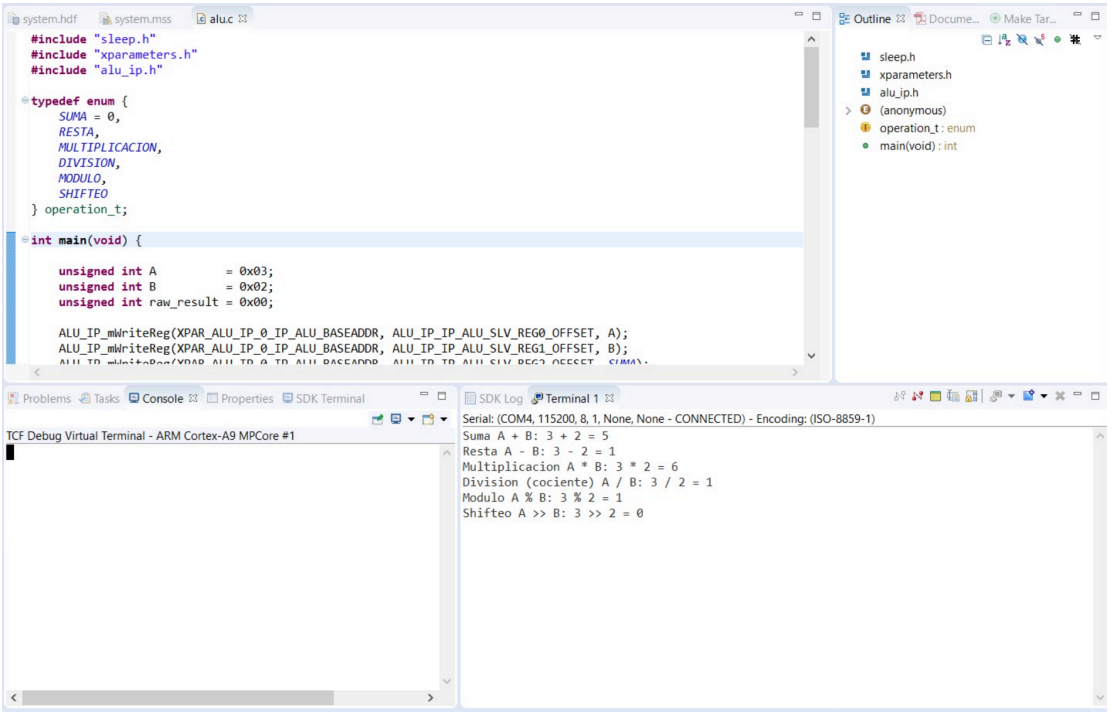


Figura 14. Implementación de la ALU.

Finalmente, la *figura 15* presenta los resultados obtenidos mediante una terminal serial conectada a la FPGA. Se ejecutaron pruebas funcionales verificando cada operación definida:

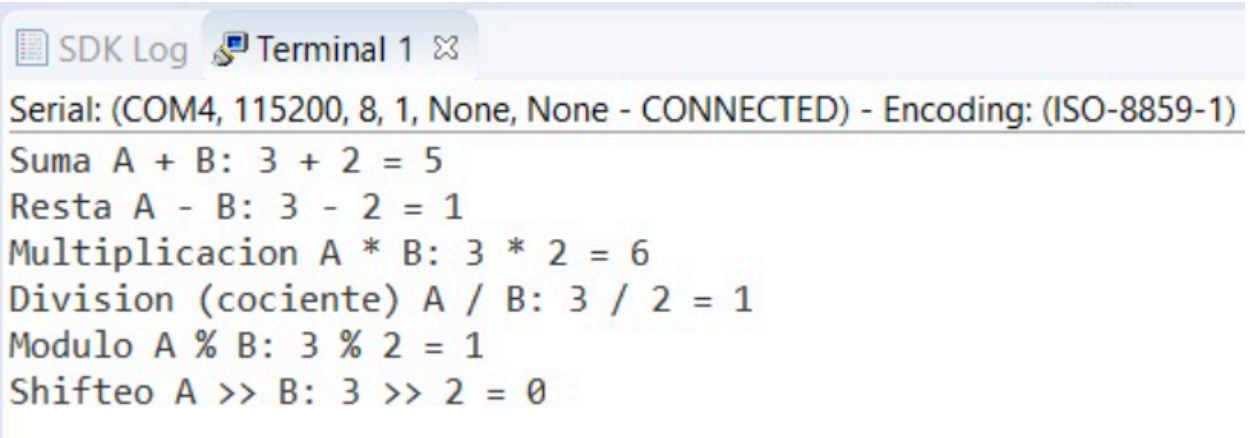


Figura 15. Resultados de implementación.

Estos resultados confirman el correcto funcionamiento del IP Core implementado en la FPGA, mostrando una interacción efectiva y una ejecución precisa de las operaciones solicitadas.