

ALU

(Unidad Aritmética Lógica)

Trabajo Práctico Final
Circuitos Lógicos Programables

Gustavo Campero
(gus.campero@gmail.com)
13/10/2025

Especialización en Sistemas Embebidos

Esta obra está bajo una
[Licencia Creative Commons Atribución](#)
[4.0 Internacional](#).

Índice de contenido

1. Introducción	2
1.1. Propósito	2
1.2. Alcance	3
2. Desarrollo	3
2.1. Arquitectura por bloques vs ALU monolítica	3
2.2. Descripción general	4
2.3. Diagrama en bloques inicial	4
2.4. Funciones del Producto	4
2.5. Características de los Usuarios	4
2.6. Restricciones	6
2.7. Suposiciones y Dependencias	6
2.8. Requisitos Futuros	7
3. Requisitos Específicos	7
3.1. Interfaces Externas	7
3.2. Funciones	7
3.2.1. Establecer conectividad BLE con dispositivos móviles	7
3.2.2. Envío y recepción de comandos desde la app móvil	7
3.2.3. Exponer datos de diagnóstico y estado interno	8
3.2.4. Permitir la modificación de parámetros operativos	8
3.2.5. Ejecutar actualizaciones OTA (Over The Air)	8
3.2.6. Operar sin interferir con otras funcionalidades críticas	8
3.3. Requisitos de Rendimiento	8
3.4. Restricciones de Diseño	8
3.5. Atributos del Sistema	9
3.5.1. Mantenibilidad	9
3.5.2. Portabilidad	9
3.5.3. Fiabilidad	9
3.6. Otros Requisitos	9
4. Apéndices	9

1. Introducción

1.1. Propósito

Este trabajo práctico aborda la implementación de un bloque de hardware digital descrito en VHDL y destinado a ejecutarse en una FPGA. El bloque seleccionado es una Unidad Aritmético Lógica (ALU), componente central y reutilizable en múltiples arquitecturas digitales. El flujo seguido comprende:

1. Descripción RTL en VHDL
2. Simulación funcional mediante testbenches
3. Implementación en FPGA.

Se utilizó Vivado como herramienta para simulación, síntesis, implementación del diseño y creación del archivo de configuración.

Al no contar con una placa física, se utilizó la siguiente modalidad provista por la cátedra: programación en un kit conectado al servidor y validación en tiempo real mediante el IP Virtual I/O (VIO) de Vivado.

1.2. Alcance

La ALU fue desarrollada con ancho configurable (W) y con la capacidad de cubrir las siguientes operaciones:

- ADD: suma binaria.
- SUB: resta binaria.
- MUL: parte baja del producto (W bits menos significativos).
- AND, OR, XOR, NOT: operaciones lógicas bit a bit.
- SLL: shift lógico a la izquierda.
- SRL: shift lógico a la derecha.
- SRA: shift aritmético a la derecha.
- ROL: rotación circular a la izquierda.
- ROR: rotación circular a la derecha.

2. Desarrollo

2.1. Arquitectura por bloques vs ALU monolítica

Para implementar la ALU surgió la disyuntiva entre describir una ALU monolítica (todas las operaciones dentro de un único case) o construir una ALU modular por bloques. A continuación se presentan las ventajas y desventajas de utilizar una ALU por bloques:

Ventajas:

- Modularidad: cada operación es una entidad independiente, con interfaz clara. Facilita lectura, depuración y evolución del diseño.
- Reutilización: los bloques pueden reutilizarse en otros proyectos o laboratorios, y reemplazarse por versiones alternativas.
- Verificación por partes: permite testbenches unitarios por operación (menor tiempo de diagnóstico y mayor cobertura dirigida).
- Escalabilidad: es directo habilitar/deshabilitar funcionalidades o ajustar anchos específicos, sin reescribir una ALU grande.

Desventajas:

- Área potencialmente mayor: si se instancian bloques redundantes (ej. sumador y restador por separado) puede crecer el uso de LUTs/DSPs.
- Mayor potencia dinámica: todos los bloques conmutan siempre, aunque no se usen, generando más consumo y temperatura.

Teniendo en cuenta esto y con el objetivo de priorizar claridad, verificabilidad y flexibilidad se decidió por una implementación modular pero tomando los siguientes cuidados:

1. Habilitación por bloque: se agrega a cada componente una señal EN codificada a partir de la operación elegida. Cuando EN sea 0 se enmascaran las entradas del bloque con ceros evitando que la lógica interna conmute.
2. Agrupación de operaciones afines: las siguientes operaciones se implementan en bloques compartidos para mejorar el aprovechamiento de hardware
 - a. Suma/resta
 - b. Operaciones lógicas
 - c. Desplazamientos/rotaciones

2.2. Descripción general

Entradas:

- A, B [W-1 downto 0]: operandos
- OP [3 downto 0]: código de operación.

Salidas:

- Y [W-1 downto 0]: resultado de la operación.
- Z: flag que indica resultado igual a 0
- N: flag que indica resultado negativo
- C: flag que indica acarreo en la salida
- oF: flag que indica overflow

2.3. Diagrama en bloques inicial

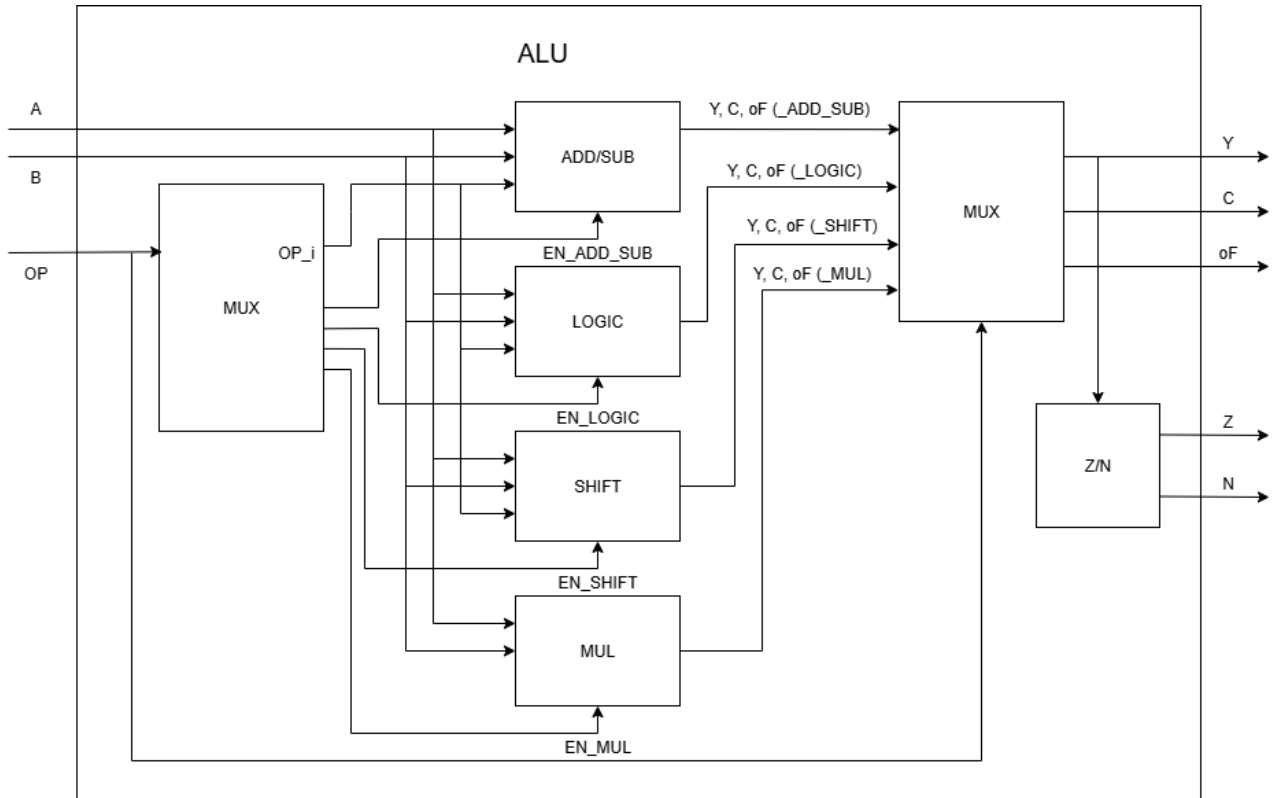


Figura 1. Diagrama en bloques inicial de la implementación.

2.4. Implementación

Siguiendo el diagrama en bloques se pensó una implementación con un bloque principal que instancie los componentes de los 4 bloques de operaciones. El programa funciona de forma combinacional. Utiliza la entrada OP para generar las señales OP_i de cada bloque que indican cuál de las operaciones se debe realizar y la señal EN correspondiente para habilitar el funcionamiento del bloque específico. El bloque de operaciones generará las señales de salida correspondientes al resultado y a los flags Z, N, C y oF (si corresponde) y luego otro MUX selecciona la salida de qué bloque deberá mapear a la salida de la ALU dependiendo de la entrada OP.

La implementación se realizó entonces en 5 archivos .vhd:

- *alu.vhd* (bloque principal)
- *addsub.vhd*
- *logic.vhd*

- *mul.vhd*
- *shift.vhd*

Estos archivos se entregan a la vez que este informe.

Una vez desarrollado, es posible crear el proyecto en Vivado y realizar un análisis RTL del código fuente. A continuación se deja una imagen del esquemático generado por la herramienta. Para poder ver la imagen más en detalle se adjunta como anexo un archivo PDF con el esquemático completo.

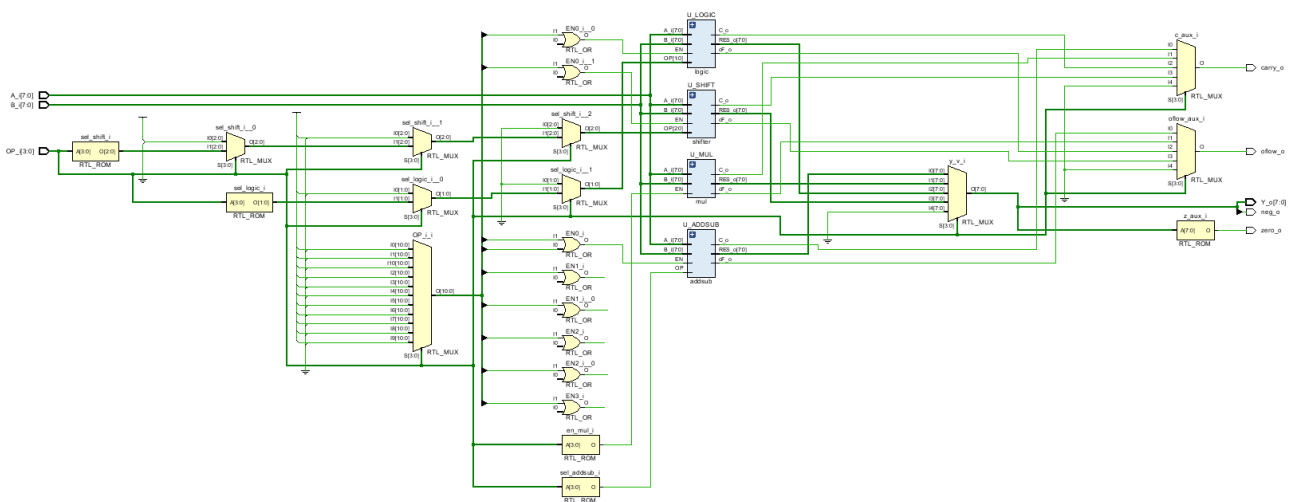


Figura 2. Esquemático generado en Vivado.

3. Simulación

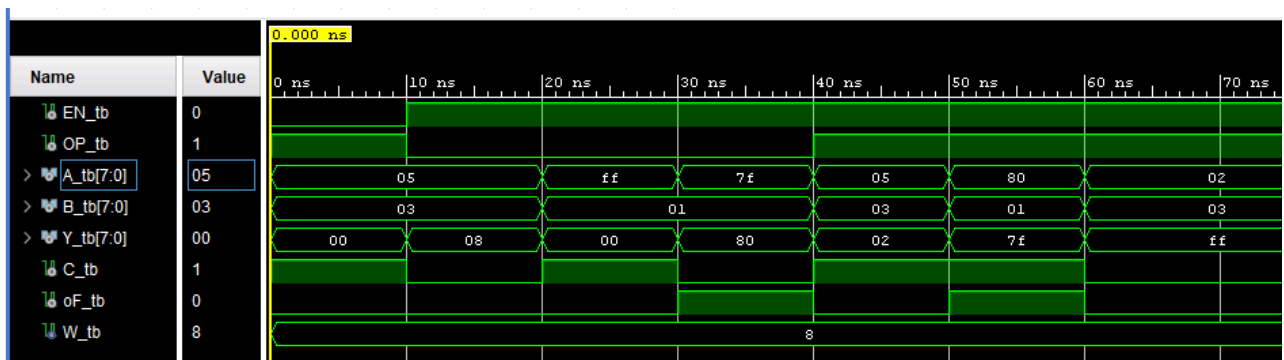
Como se explicó anteriormente, esta implementación permite realizar simulaciones por bloque para verificar que todas las operaciones funcionen correctamente. Por lo tanto se implementó un test bench para probar que los resultados de cada bloque sean correctos y un test bench general para la ALU. A continuación se presentan las simulaciones mas destacables:

3.1. ADD/SUB

Esperado:

#	EN	OP	A	B	RES	C	oF
1	0	1	05	03	00	1	0
2	1	0	05	03	08	0	0
3	1	0	FF	01	00	1	0
4	1	0	7F	01	80	0	1
5	1	1	05	03	02	1	0
6	1	1	80	01	7F	1	1
7	1	1	02	03	FF	0	0

Simulación:



3.2. MUL

A tener en cuenta:

- C = 1 si parte alta del RES es igual a 0
- oF no se implementó, siempre devuelve 0

Esperado:

#	EN	A	B	Detalle	RES	C	oF
1	0	05	03	EN = 0	00	0	0
2	1	0F	10	$0x0F * 0x10 = 0x00F0$	F0	0	0
3	1	10	10	$0x10 * 0x10 = 0x0100$	00	1	0
4	1	FF	FF	$0xFF * 0xFF = 0xFE01$	01	1	0
5	1	00	AB	$0x00 * 0xAB = 0x0000$	00	0	0

Simulación:



3.3. LOGIC

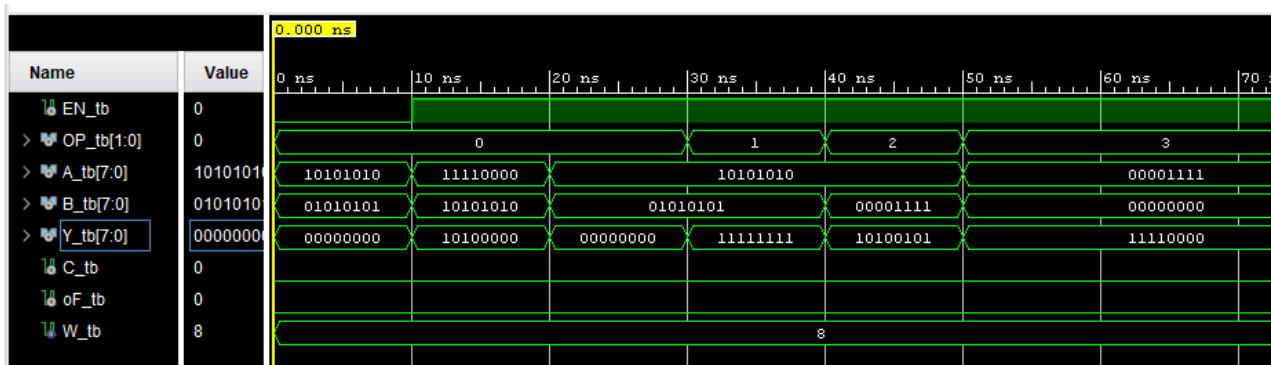
A tener en cuenta:

- C siempre 0. No se implementa
- oF siempre 0. No se implementa

Esperado:

#	EN	A	B	OP	RES	C	oF
1	0	10101010	01010101	00	00000000	0	0
2	1	11110000	10101010	00	10100000	0	0
3	1	10101010	01010101	00	00000000	0	0
4	1	10101010	01010101	01	11111111	0	0
5	1	10101010	00001111	10	10100101	0	0
6	1	00001111	00000000	11	11110000	0	0

Simulación:



3.4. SHIFT

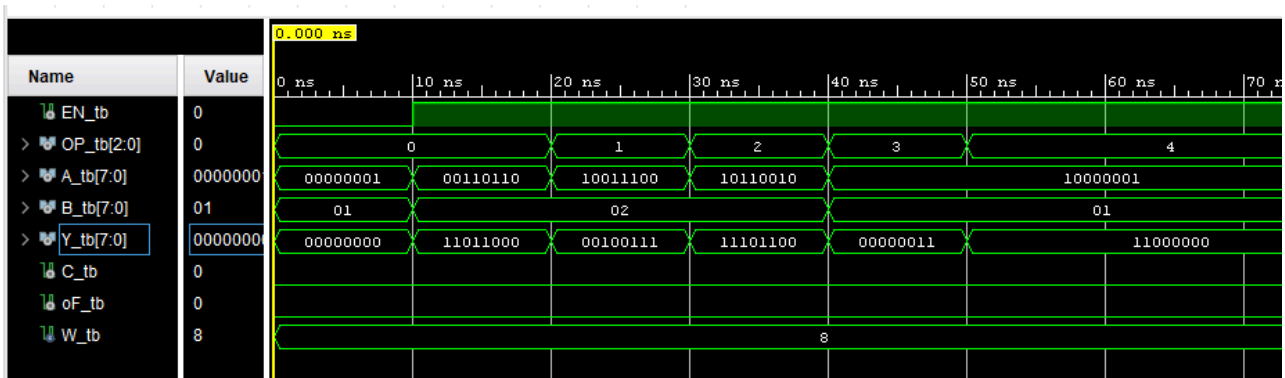
A tener en cuenta:

- C siempre 0. No se implementa
- oF siempre 0. No se implementa

Esperado:

#	EN	A	B	OP	RES	C	oF
1	0	00000001	01	00	00000000	0	0
2	1	00110110	02	00	11011000	0	0
3	1	10011100	02	00	00100111	0	0
4	1	10110010	02	01	11101100	0	0
5	1	10000001	01	10	00000011	0	0
6	1	10000001	01	11	11000000	0	0

Simulación:



3.5. ALU

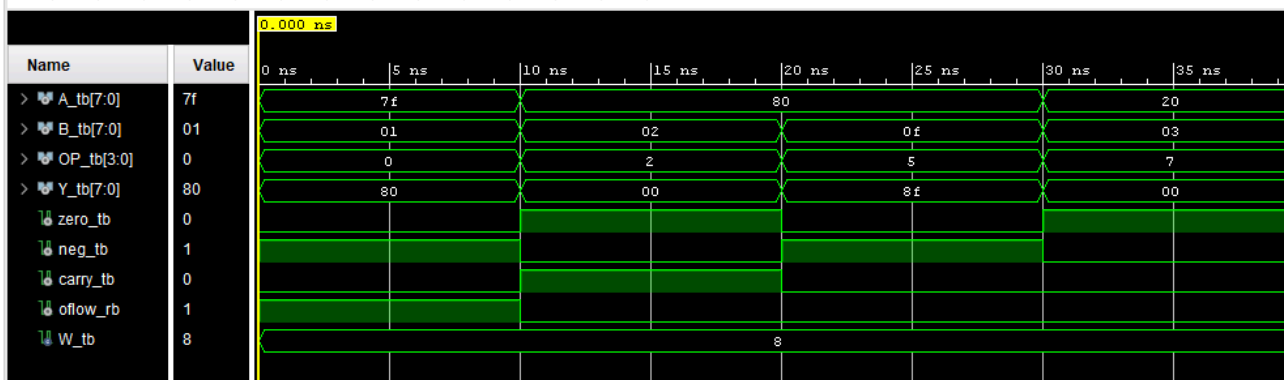
A tener en cuenta:

Como ya se simularon todas las operaciones en este caso solo es necesario probar que el OP seleccione el bloque correcto y que las flags que se setean fuera de los bloques de operaciones funcionan por lo que solo se requiere cuatro pruebas.

Esperable:

#	A	B	OP	Y	Z	N	C	oF
1	7F	01	0000	80	0	1	0	1
2	80	02	0010	00	1	0	1	0
3	80	0F	0101	8F	0	1	0	0
4	20	03	0111	00	1	0	0	0

Simulación:



4. Programación

Al no contar con una placa física, se utilizó la siguiente modalidad provista por la cátedra: programación en un kit conectado al servidor y validación en tiempo real mediante el IP Virtual I/O (VIO) de Vivado.

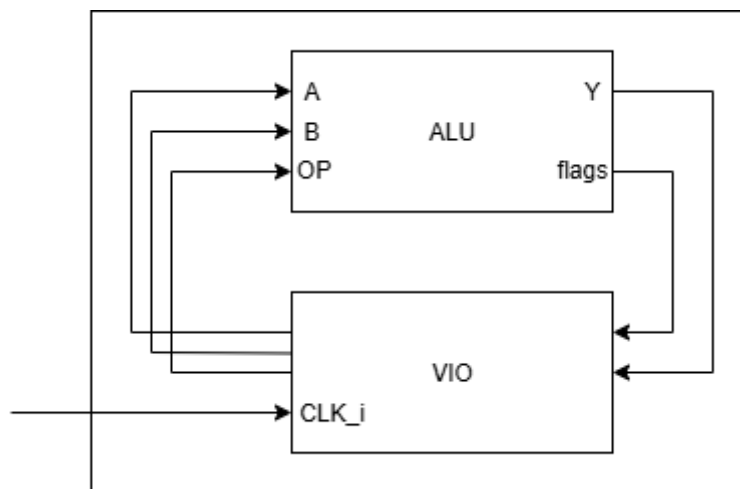


Figura 3. Diagrama ALU con VIO

Para facilitar la implementación del bloque VIO se juntaron las 4 flags Z, N, C, oF en una señal de prueba *probe_flags* (3 downto 0).

Una vez agregado el VIO el análisis RTL nos genera el siguiente esquemático:

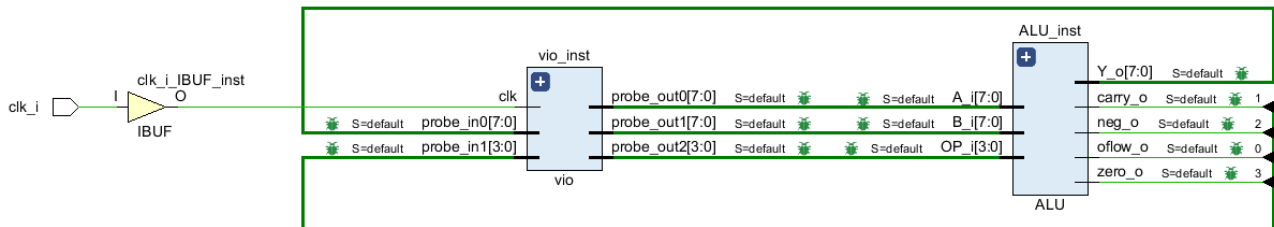
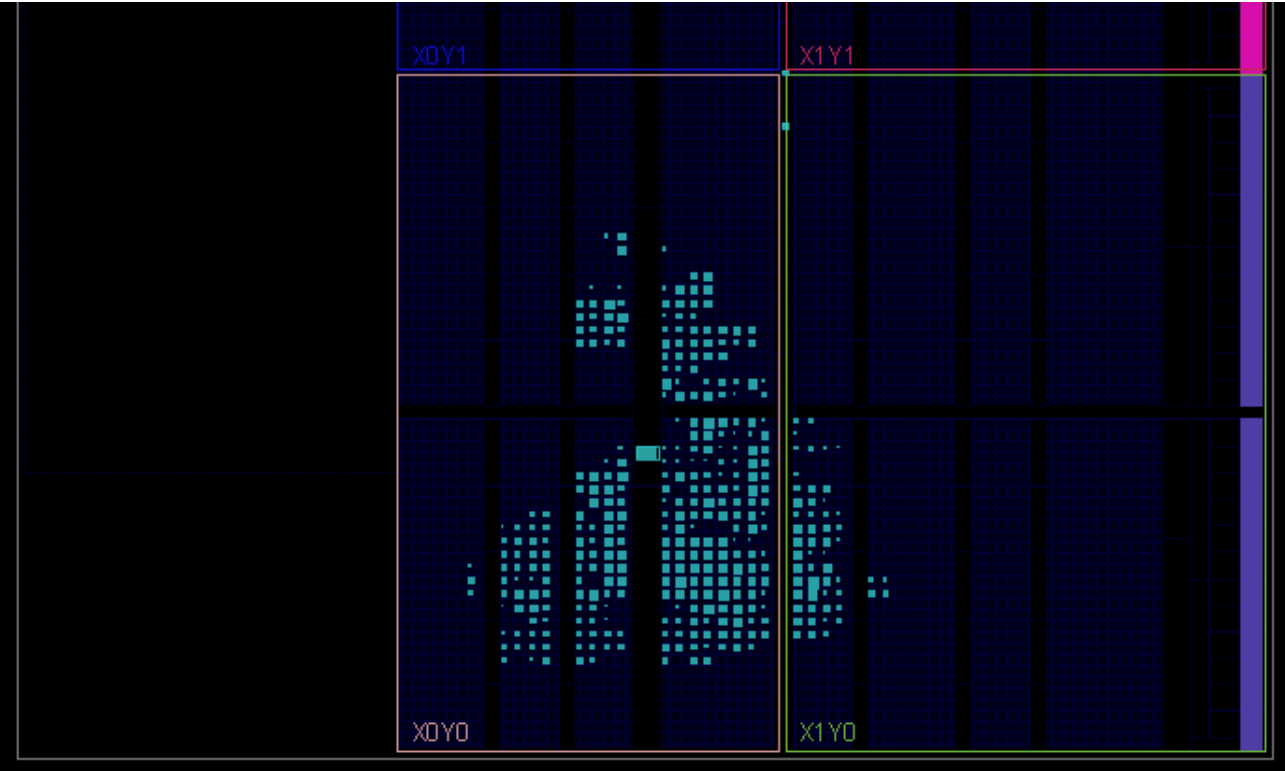


Figura 4. Esquemático (ALU-VIO)

4.1. Tabla de uso de recursos de la FPGA

Name	Slice LUTs (17600)	Slice Registers (35200)	Slice (440 0)	LUT as Logic (17600)	LUT as Memory (6000)	LUT Flip Flop Pairs (17600)	Bonded IOB (100)	BUFGCTRL (32)	BSCANE2 (4)
ALU_VIO	874	1086	366	850	24	451	1	2	1
ALU_inst (ALU)	244	0	73	244	0	0	0	0	0
dbg_hub (dbg_hub)	463	723	222	439	24	295	0	1	1
vio_inst (vio)	167	363	92	167	0	135	0	0	0

Resource	Utilization	Available	Utilization %
LUT	874	17600	4.97
LUTRAM	24	6000	0.40
FF	1086	35200	3.09
IO	1	100	1.00



4.2. Programación en FPGA

A continuación se muestran ejemplos de uso de la implementación en la placa FPGA remota:

hw_vio_1

🔍

⏏

⏏










+

-

Name	Value	Activity	Direction	VIO
> probe_A[7:0]	[H] 7F ▾		Output	hw_vio_1
> probe_B[7:0]	[H] 01 ▾		Output	hw_vio_1
> probe_OP[3:0]	[H] 0 ▾		Output	hw_vio_1
> probe_Y[7:0]	[H] 80		Input	hw_vio_1
∨ probe_flags[3:0]	[H] 5		Input	hw_vio_1
probe_flags[3]	0		Input	hw_vio_1
probe_flags[2]	1		Input	hw_vio_1
probe_flags[1]	0		Input	hw_vio_1
probe_flags[0]	1		Input	hw_vio_1

hw_vio_1					
<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>					
Name	Value	Activity	Direction	VIO	
> probe_A[7:0]	[H] 80		Output	hw_vio_1	
> probe_B[7:0]	[H] 02		Output	hw_vio_1	
> probe_OP[3:0]	[B] 0010		Output	hw_vio_1	
> probe_Y[7:0]	[H] 00		Input	hw_vio_1	
∨ probe_flags[3:0]	[H] A		Input	hw_vio_1	
probe_flags[3]		1	Input	hw_vio_1	
probe_flags[2]		0	Input	hw_vio_1	
probe_flags[1]		1	Input	hw_vio_1	
probe_flags[0]		0	Input	hw_vio_1	

hw_vio_1					
<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>					
Name	Value	Activity	Direction	VIO	
> probe_A[7:0]	[H] 80		Output	hw_vio_1	
> probe_B[7:0]	[H] 0F		Output	hw_vio_1	
> probe_OP[3:0]	[B] 0101		Output	hw_vio_1	
> probe_Y[7:0]	[H] 8F		Input	hw_vio_1	
∨ probe_flags[3:0]	[H] 4		Input	hw_vio_1	
probe_flags[3]		0	Input	hw_vio_1	
probe_flags[2]		1	Input	hw_vio_1	
probe_flags[1]		0	Input	hw_vio_1	
probe_flags[0]		0	Input	hw_vio_1	

hw_vio_1					
<div> <div></div> <div></div> <div></div> <div></div> <div></div> </div>					
Name	Value	Activity	Direction	VIO	
>  probe_A[7:0]	[B] 0001_0000 ▾		Output	hw_vio_1	
>  probe_B[7:0]	[H] 03 ▾		Output	hw_vio_1	
>  probe_OP[3:0]	[B] 0111 ▾		Output	hw_vio_1	
>  probe_Y[7:0]	[B] 1000_0000	↑	Input	hw_vio_1	
▼  probe_flags[3:0]	[H] 4	↕	Input	hw_vio_1	
└─  probe_flags[3]	0	↓	Input	hw_vio_1	
└─  probe_flags[2]	1	↑	Input	hw_vio_1	
└─  probe_flags[1]	0		Input	hw_vio_1	
└─  probe_flags[0]	0		Input	hw_vio_1	