

Trabajo práctico final Circuitos Lógicos Programables

Unidad Lógica Aritmética (ALU)

Alumno:

- **Ing. Alejo S. García Mata**

Docente:

- **Ing. Nicolás Álvarez**

Índice

Índice.....	2
Registro de cambios.....	2
1. Introducción.....	4
2. Descripción funcional.....	4
2.1 Entradas y salidas.....	5
2.2 Operaciones implementadas.....	5
3. Implementación en VHDL.....	6
3.1 Implementación principal de la ALU.....	6
Arquitectura general:.....	7
4. Verificación por simulación.....	10
5. Síntesis e implementación.....	13
5.1 Generación del bitstream y prueba en hardware real.....	15

Registro de cambios

Revisión	Cambios realizados	Fecha
1.0	Creación del documento	14/10/2025

1. Introducción

El presente trabajo práctico final tiene como objetivo el diseño, descripción, simulación e implementación en hardware de una **Unidad Lógica Aritmética (ALU)** parametrizable, utilizando el lenguaje VHDL sobre una FPGA Xilinx Zynq-7000 (Arty Z7-10).

La finalidad del desarrollo es aplicar los conceptos de diseño digital combinacional y el flujo de trabajo completo de síntesis e implementación en Vivado Design Suite, incorporando técnicas de verificación tanto por simulación como en hardware mediante el uso del núcleo Virtual Input/Output (VIO).

El proyecto busca demostrar la capacidad de integrar módulos HDL, paquetes de constantes, bancos de prueba y periféricos virtuales, obteniendo un bloque funcional reutilizable que cumpla los requisitos típicos de un procesador elemental.

La ALU diseñada es modular, completamente combinacional y configurable en el tamaño de palabra a través del parámetro genérico N.

2. Descripción funcional

La ALU (Arithmetic Logic Unit) implementada realiza operaciones aritméticas y lógicas entre dos operandos de **N** bits (A y B), seleccionadas mediante un código de operación de 4 bits (OP).

El resultado se entrega en la salida Y, acompañado de un conjunto de banderas de estado:

- **Z (Zero)**: se activa cuando el resultado es nulo.
- **C (Carry)**: indica acarreo según la operación.
- **V (Overflow)**: detecta desbordamiento
- **Nf (Negativo)**: refleja el bit más significativo del resultado.

La ALU está diseñada para ser **pura combinacional**, sin elementos secuenciales ni reloj, lo que la convierte en un bloque de cálculo instantáneo dependiente únicamente de sus entradas.

El parámetro genérico N permite escalar el ancho de palabra según las necesidades del sistema.

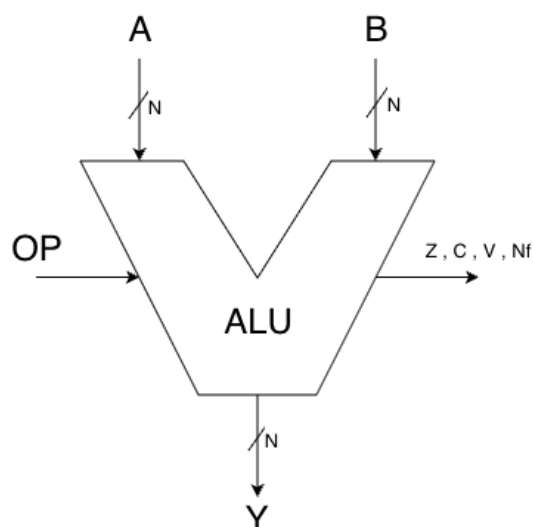


Figura 1: Diagrama de la ALU

2.1 Entradas y salidas

Señal	Descripción	Tipo
A, B	Operandos de entrada	std_logic_vector(N-1 downto 0)
OP	Código de operación	std_logic_vector(3 downto 0)
Y	Resultado	std_logic_vector(N-1 downto 0)
Z, C, V, Nf	Banderas de estado	std_logic

2.2 Operaciones implementadas

Código (OP)	Operación	Descripción
0000	ADD	Suma
0001	SUB	Resta

0010	AND	Lógica AND
0011	OR	Lógica OR
0100	XOR	Lógica XOR
0101	SLL	Desplazamiento lógico a izquierda
0110	SRL	Desplazamiento lógico a derecha
0111	SRA	Desplazamiento aritmético a derecha
1000	SLT	Comparación signed (Set Less Than)
1001	SLTU	Comparación unsigned (Set Less Than Unsigned)

3. Implementación en VHDL

El proyecto se organizó en cuatro módulos principales:

a) alu_pkg.vhd: Contiene la codificación simbólica de las operaciones mediante constantes de 4 bits.

b) alu.vhd: Implementa la lógica principal de la ALU, el módulo es genérico y admite cualquier valor de N, siendo por defecto 32 bits.

c) tb_alu.vhd: Banco de pruebas automatizado.

d) alu_VIO.vhd: Wrapper para implementación en hardware real. Integra la ALU con un **IP VIO** (Virtual Input/Output) de Vivado.

3.1 Implementación principal de la ALU

El núcleo del diseño, contenido en `alu.vhd`, implementa la lógica combinacional de la ALU mediante un enfoque modular y parametrizable.

El componente se estructura en torno a un parámetro genérico N que define el ancho de palabra, permitiendo reutilizar la misma descripción para diferentes tamaños de dato (por defecto 32 bits, en este proyecto se instanció con 8 bits).

Arquitectura general:

El bloque está compuesto por tres secciones internas:

1. Declaraciones previas y señales auxiliares

```
architecture alu_arq of alu is
  function clog2(n: integer) return integer is
    variable r: integer := 0; variable v: integer := n-1;
  begin
    while v > 0 loop r := r + 1; v := v / 2; end loop;
    if r = 0 then return 1; else return r; end if;
  end function;
  constant SHW : integer := clog2(N);

  signal add_ext : unsigned(N downto 0);
  signal sub_ext : unsigned(N downto 0);
  signal y_next : std_logic_vector(N-1 downto 0);
  signal c_next : std_logic := '0';
  signal v_next : boolean := false;
  signal shamt : integer range 0 to N-1;
```

Figura 2: declaraciones previas

Cómo se puede ver en la figura 2, dentro de la arquitectura de la ALU se definen las señales y funciones que permiten mantener la coherencia del diseño en cualquier configuración de tamaño de palabra.

El bloque comienza con la función **cLog2**, encargada de calcular en forma automática el número mínimo de bits necesarios para representar el desplazamiento máximo de un vector de N bits. La función recibe un entero n y devuelve el entero r tal que $2^r \geq n$. Esto evita tener que ajustar manualmente el tamaño del bus de desplazamiento cada vez que cambia el parámetro N. Por ejemplo, para N=8 devuelve 3 bits, y para N=32 devuelve 5. El valor resultante se almacena en la constante **SHW**, utilizada luego para determinar cuántos bits menos significativos del operando B se usarán como cantidad de desplazamiento.

Las señales **add_ext** y **sub_ext** son vectores de N+1 bits que almacenan los resultados intermedios de la suma y la resta respectivamente, con un bit adicional reservado para capturar el acarreo.

La señal **y_next** representa el resultado intermedio de la operación seleccionada, mientras que **c_next** y **v_next** almacenan de forma temporal los valores calculados de las banderas de carry y overflow antes de ser asignadas a las salidas principales.

También se define la señal entera **shamt**, encargada de contener el valor numérico de desplazamiento utilizado en las operaciones SLL, SRL y SRA.

2. Proceso combinacional principal

```
begin
  add_ext <= unsigned('0' & A) + unsigned('0' & B);
  sub_ext <= unsigned('0' & A) + unsigned('0' & (not B)) + 1;

  shamt <= to_integer(unsigned(B(SHW-1 downto 0)));

  process(A,B,OP,add_ext,sub_ext,shamt)
    variable As : signed(N-1 downto 0);
    variable Bs : signed(N-1 downto 0);
    variable Yv : std_logic_vector(N-1 downto 0);
    variable Cv : std_logic := '0';
    variable Vv : boolean := false;
  begin
    As := signed(A);
    Bs := signed(B);
    Yv := (others => '0'); Cv := '0'; Vv := false;

    case OP is
      when OP_ADD =>
        Yv := std_logic_vector(add_ext(N-1 downto 0));
        Cv := add_ext(N);
        Vv := (A(N-1)=B(N-1)) and (Yv(N-1)/=A(N-1));
      when OP_SUB =>
        Yv := std_logic_vector(sub_ext(N-1 downto 0));
        Cv := sub_ext(N);
        Vv := (A(N-1)/=B(N-1)) and (Yv(N-1)/=A(N-1));
      when OP_AND => Yv := A and B;
      when OP_OR  => Yv := A or  B;
      when OP_XOR => Yv := A xor B;
      when OP_SLL =>
        Yv := std_logic_vector(shift_left(unsigned(A), shamt));
      when OP_SRL =>
        Yv := std_logic_vector(shift_right(unsigned(A), shamt));
      when OP_SRA =>
        Yv := std_logic_vector(shift_right(signed(A), shamt));
      when OP_SLT =>
        if As < Bs then Yv := (others => '0'); Yv(0) := '1'; else Yv := (others => '0'); end if;
      when OP_SLTU =>
        if unsigned(A) < unsigned(B) then Yv := (others => '0'); Yv(0) := '1'; else Yv := (others => '0'); end if;
      when others =>
        Yv := (others => '0');
    end case;
  end process;
```

Figura 3: núcleo del proceso combinacional de la ALU

La **Figura 3** muestra la estructura completa del bloque combinacional que define el funcionamiento interno de la ALU. Se observa cómo se preparan las señales auxiliares y las operaciones extendidas antes del proceso principal, garantizando que cada cálculo esté correctamente dimensionado y sincronizado con el parámetro genérico N.

El bloque comienza con las asignaciones de **add_ext** y **sub_ext**, que generan las versiones extendidas de la suma y la resta.

Ambas señales son de tipo unsigned y poseen un bit adicional de longitud para permitir la detección del acarreo.

En **add_ext**, los operandos A y B se amplían con un bit '0' en la posición más significativa y luego se suman directamente. En **sub_ext**, el cálculo se realiza como $A + (\sim B) + 1$, implementando la resta mediante complemento a dos.

A continuación se define la señal **shamt**, que representa la cantidad de desplazamiento usada por las operaciones de *shift*. Este valor se obtiene convirtiendo los bits menos

significativos del operando B en un número entero mediante la función `to_integer(unsigned(...))`.

El ancho de los bits considerados se determina automáticamente con la función `clog2`, en función del tamaño de palabra N, garantizando que el número de desplazamientos posibles coincida con la longitud del operando.

El núcleo del proceso está formado por una sentencia `case` que evalúa el código de operación OP.

Cada rama implementa una de las funciones definidas en el paquete *alu_pkg.vhd*:

- Las operaciones **aritméticas** (ADD, SUB) toman el resultado de `add_ext` o `sub_ext` y actualizan las banderas C y V de acuerdo con las condiciones de overflow y acarreo.
- Las **operaciones lógicas** (AND, OR, XOR) se realizan directamente bit a bit entre A y B.
- Las **operaciones de desplazamiento** (SLL, SRL, SRA) usan las funciones `shift_left` y `shift_right` sobre `unsigned` o `signed`, según corresponda, y aplican la cantidad de desplazamiento indicada por `shamt`.
- Las **comparaciones** (SLT, SLTU) establecen el bit menos significativo de Yv en '1' si la condición de menor se cumple, dejando el resto del vector en '0'.
- Finalmente, el caso `others` garantiza que cualquier código no definido produzca una salida nula.

La figura muestra cómo el flujo de datos queda totalmente determinado por combinaciones de señales, sin procesos secuenciales ni elementos de memoria.

De esta forma, la ALU se comporta como un bloque combinacional puro, donde cada cambio en las entradas A, B u OP produce inmediatamente un nuevo valor en la salida Y y en las banderas de estado, respetando la parametrización genérica y manteniendo la coherencia del diseño en cualquier ancho de palabra.

3. Asignación final de salidas

- El resultado Y y las banderas C y V se actualizan con los valores calculados.
- La bandera **Negativo (Nf)** toma directamente el bit más significativo del resultado (`Y(N-1)`).
- La bandera **Zero (Z)** se evalúa comparando todo el vector Y con cero.

4. Verificación por simulación

La verificación de la ALU se realizó mediante un banco de pruebas (tb_alu.vhd), ejecutado en el simulador de Vivado. El testbench genera los vectores de entrada A, B y OP, aplicando una secuencia de pruebas que cubre todas las operaciones implementadas. Cada resultado se compara con el valor esperado mediante sentencias `assert`, lo que permite detectar automáticamente cualquier discrepancia funcional.

Las pruebas abarcan casos de suma y resta con y sin overflow, operaciones lógicas bit a bit, desplazamientos y comparaciones signed/unsigned. Cada operación verifica también el comportamiento de las banderas Z, C, V y Nf. El banco de pruebas está diseñado para recorrer distintos escenarios de valores límite, como suma de máximos positivos y negativos, desplazamientos y comparaciones entre números consecutivos.

En la simulación mostrada en la figura 4, se observaron en el visor de ondas las señales internas A, B, OP, Y, Z, C, V y Nf. En todos los casos, las salidas coincidieron con los resultados teóricos, confirmando que la ALU responde correctamente.

Durante la simulación se ejecutaron los siguientes casos de prueba definidos en el banco de pruebas:

- **Suma (ADD):**
 - $0x7F + 0x01 \rightarrow 0x80$, con $V=1$ (overflow positivo).
 - $0xC0 + 0xC0 \rightarrow 0x80$, con $V=0$ y $C=1$ (sin overflow, con carry).
 - $0xFF + 0xFF \rightarrow 0xFE$, con $C=1$ (suma máxima unsigned).

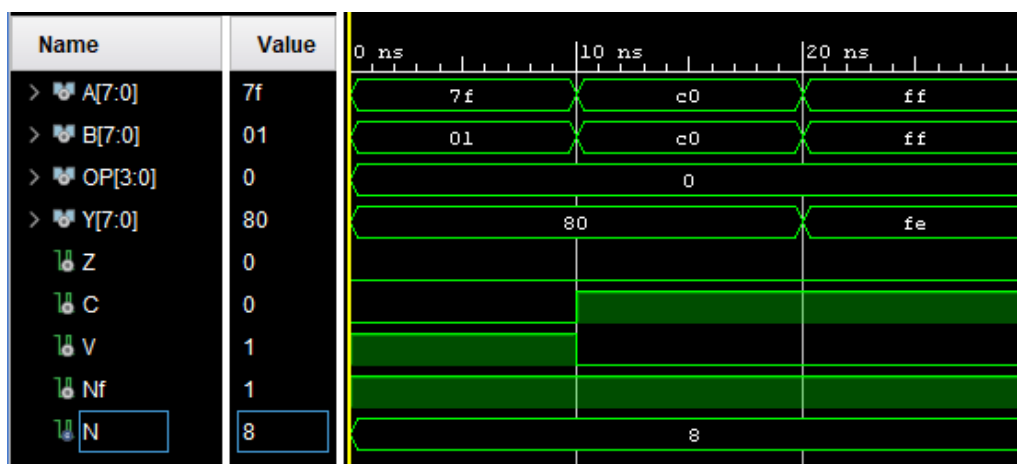


Figura 4

• **Resta (SUB):**

- $0x05 - 0x01 \rightarrow 0x04$, con $C=1$, $V=0$ (sin borrow).
- $0x00 - 0x01 \rightarrow 0xFF$, con $C=0$ (borrow).
- $0x80 - 0x80 \rightarrow 0x00$, con $Z=1$ (resultado nulo).

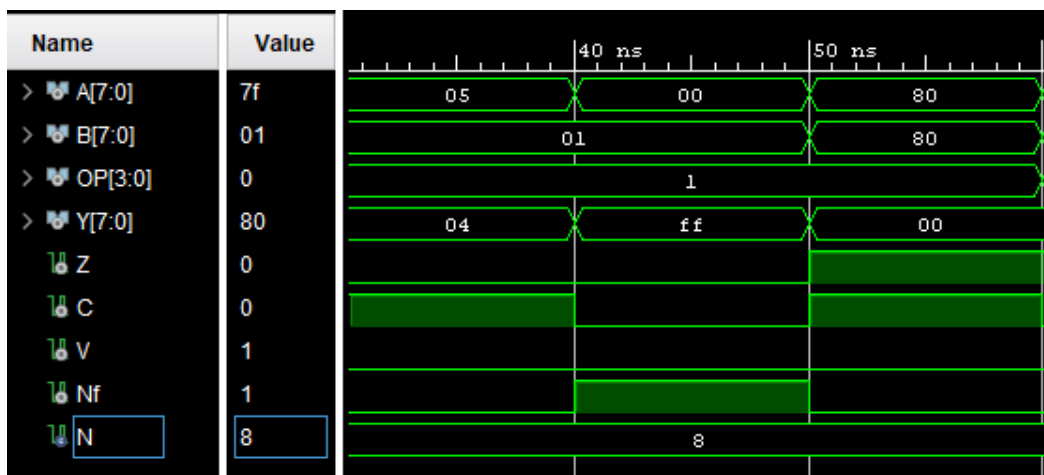


Figura 5

• **Operaciones lógicas:**

- $0xAA \text{ AND } 0x00 \rightarrow 0x00$, activa $Z=1$.
- $0xA0 \text{ OR } 0x0F \rightarrow 0xAF$.
- $0xAA \text{ XOR } 0x55 \rightarrow 0xFF$ (complementarios).
- $0x5A \text{ XOR } 0x5A \rightarrow 0x00$, activa $Z=1$.

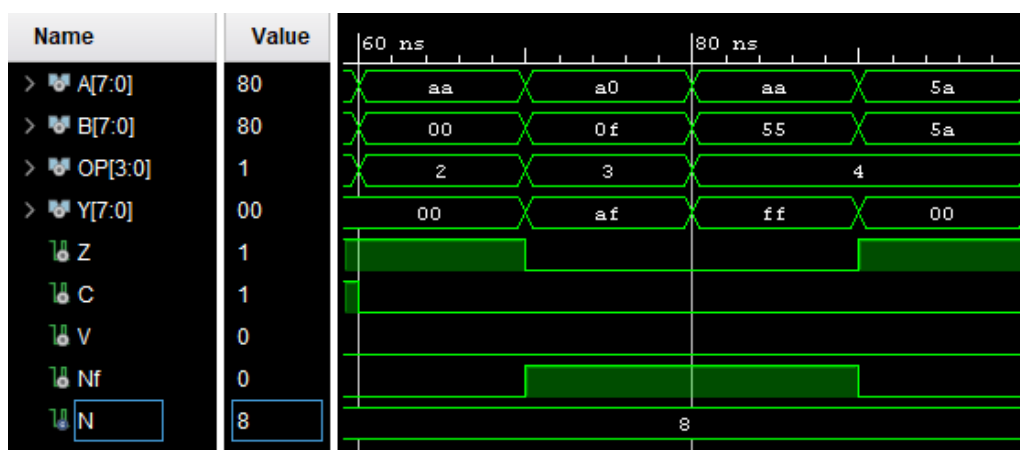


Figura 6

● **Desplazamientos:**

- $SLL(0x01, 7) \rightarrow 0x80$ (desplazamiento total a izquierda).
- $SRL(0x80, 7) \rightarrow 0x01$ (desplazamiento total a derecha).
- $SRA(0xF0, 4) \rightarrow 0xFF$ (extensión correcta del bit de signo).

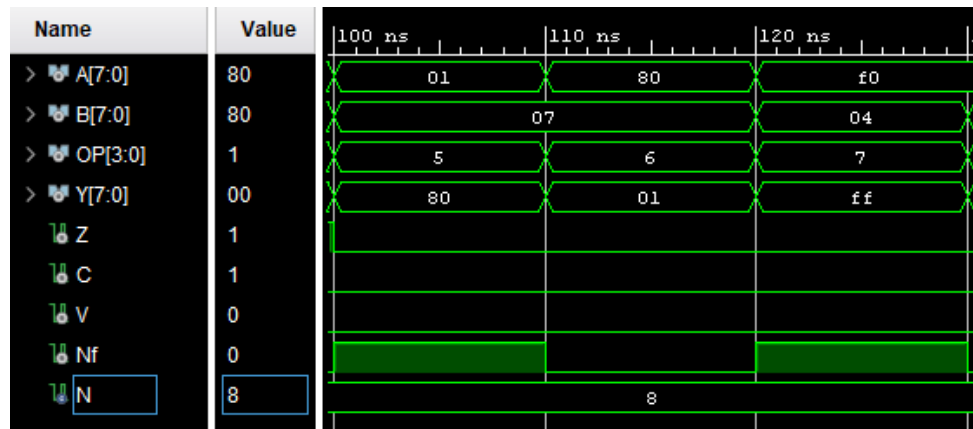


Figura 7

● **Comparaciones:**

- Signed: $0xF0 < 0x10 \rightarrow Y=0x01$.
- Signed: $0x10 \geq 0xF0 \rightarrow Y=0x00$.
- Unsigned: $0xFF > 0x01 \rightarrow Y=0x00$.

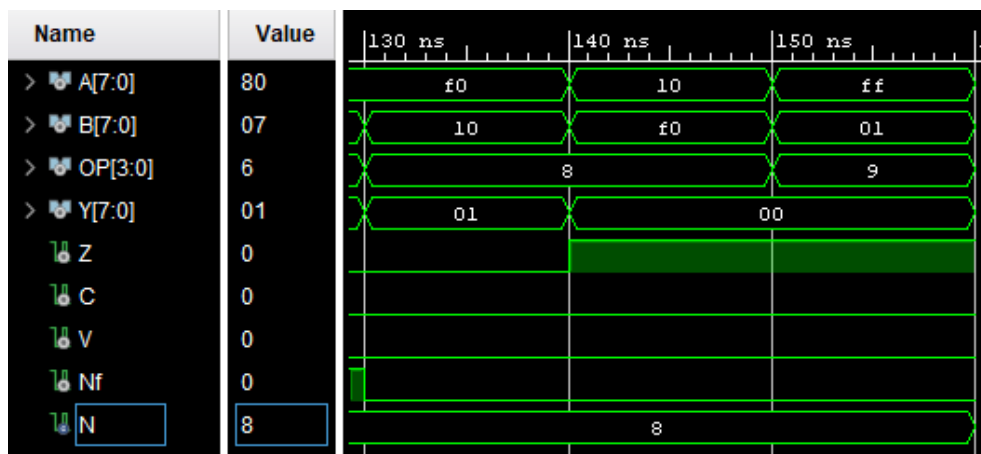


Figura 8

5. Síntesis e implementación

Luego de la verificación funcional, se realizó la **síntesis e implementación del diseño** sobre la FPGA **Arty Z7-10**, con el fin de evaluar el uso de recursos lógicos y confirmar la correcta adaptación del código a hardware real.

La síntesis se ejecutó en **Vivado 2018.1**, el proceso se completó sin advertencias críticas ni inferencias no deseadas, generando una red lógica puramente combinacional, consistente con la arquitectura de la ALU. A continuación se muestra la tabla de utilización de recursos (Utilization Report) generada por Vivado, donde se observa un uso reducido de LUTs, slices y pines de E/S, confirmando la baja complejidad del diseño.

Resource	Utilization	Available	Utilization %
LUT	707	17600	4.02
LUTRAM	24	6000	0.40
FF	1090	35200	3.10
IO	1	100	1.00
BUFG	2	32	6.25

Figura 9

La figura siguiente muestra la vista del esquema sintetizado, donde se identifican las operaciones aritmético-lógicas y la conexión con el núcleo **VIO**, utilizado para la estimulación y observación de señales desde el hardware manager.

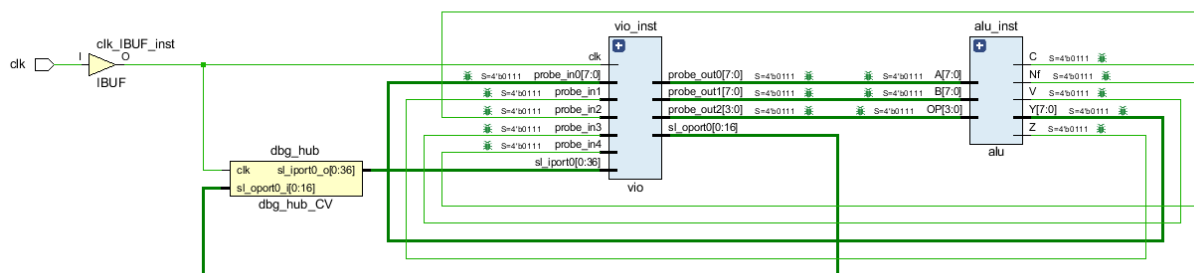


Figura 8: esquemático de síntesis

Durante la etapa de síntesis se generó el esquema RTL del diseño. En esta vista se observa la estructura interna de la ALU, donde cada operación aritmética y lógica se representa mediante los bloques lógicos inferidos por el sintetizador. El análisis RTL permitió verificar la correcta correspondencia entre el código VHDL y el hardware sintetizado, confirmando que todas las operaciones (ADD, SUB, AND, OR, XOR, desplazamientos y comparaciones) se implementan como circuitos combinacionales sin registros intermedios.

En la figura siguiente se muestra el RTL Schematic obtenido, donde se distinguen los módulos principales:

- El bloque ALU como entidad superior.
- Las señales de entrada A, B, y OP.
- Las salidas Y, Z, C, V, Nf.
- La interconexión lógica interna que representa las operaciones seleccionadas en función del código de operación (OP).

El análisis RTL confirmó que el sintetizador no infirió elementos secuenciales, latches ni memorias no deseadas, manteniendo la naturaleza combinacional del diseño.

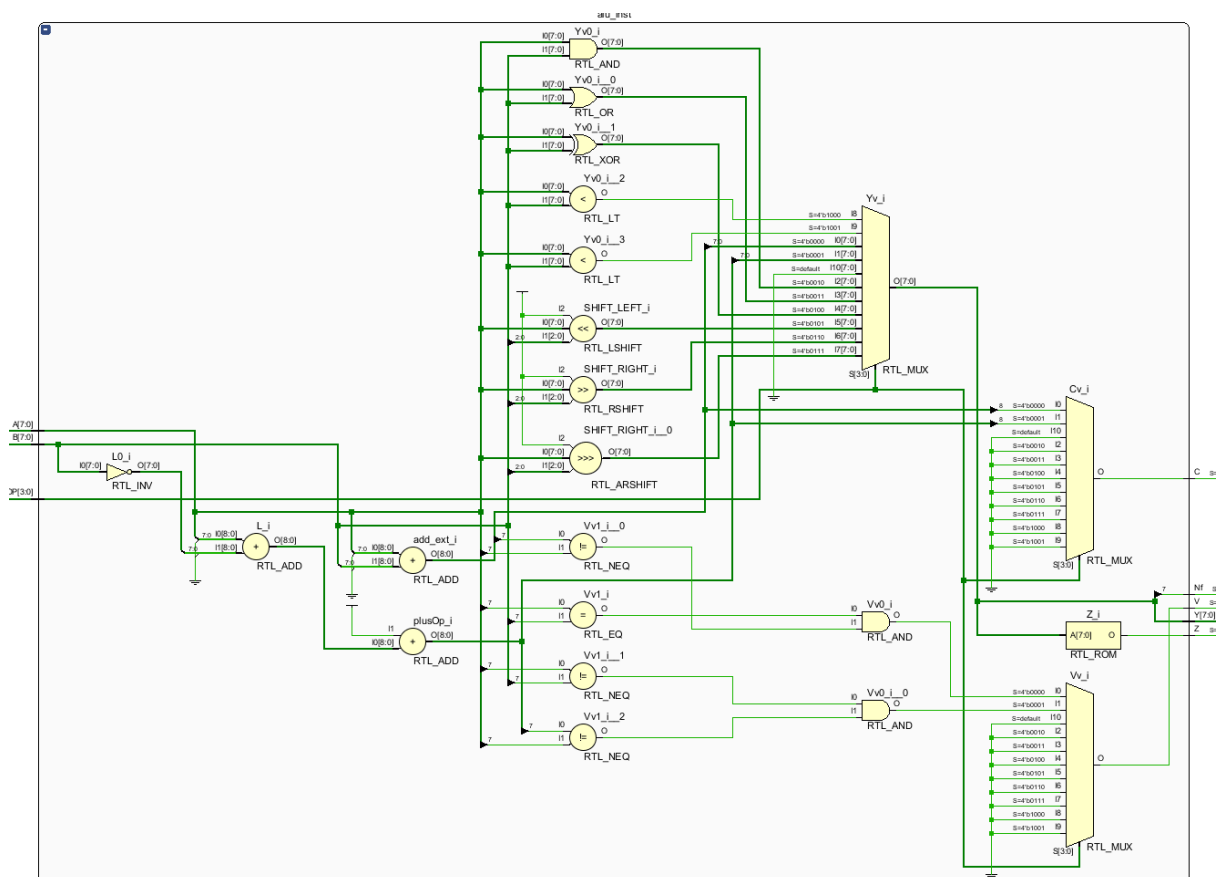


Figura 9: esquemático de análisis RTL

5.1 Generación del bitstream y prueba en hardware real

Tras verificar la correcta síntesis e implementación, se procedió a la generación del bitstream para programar la FPGA Arty Z7-10. El flujo se completó en Vivado sin errores ni advertencias de temporización, confirmando la viabilidad física del diseño dentro del dispositivo seleccionado. Una vez generado el archivo **.bit**, se cargó en la FPGA remota a través del Hardware Manager. Para realizar las pruebas funcionales en tiempo real, se integró un núcleo Virtual Input/Output (VIO), que permitió controlar las señales de entrada y observar las salidas directamente desde el entorno de Vivado, sin necesidad de periféricos externos.

En esta etapa se verificaron las operaciones de la ALU de manera interactiva:

- Los operandos A y B se modificaron desde el panel VIO.
- El código de operación OP se seleccionó manualmente para recorrer todas las funciones implementadas (ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU).
- Las salidas Y, Z, C, V y Nf se visualizaron en tiempo real en el monitor del VIO.

Durante las pruebas, la respuesta fue inmediata y coherente con los resultados obtenidos en simulación. No se observaron comportamientos indeterminados ni retardos visibles, confirmando la **naturaleza combinacional** de la ALU y la correcta propagación de las señales a través del hardware. En las siguientes figuras se muestran algunas capturas de las pruebas realizadas.









Name	Value	Activity	Direction	VIO
>  A[7:0]	[S] 127		Output	hw_vio_1
>  B[7:0]	[S] 1		Output	hw_vio_1
>  Y[7:0]	[S] -128		Input	hw_vio_1
>  OP[3:0]	[B] 0000		Output	hw_vio_1
 C_vec	[B] 0		Input	hw_vio_1
 Nf_vec	[B] 1		Input	hw_vio_1
 V_vec	[B] 1		Input	hw_vio_1
 Z_vec	[B] 0		Input	hw_vio_1

Figura 10: Suma - $0x7F + 0x01 \rightarrow 0x80$, con $V=1$, $Nf = 1$ (overflow positivo).









Name	Value
>  A[7:0]	[H] 05 ▾
>  B[7:0]	[H] 01 ▾
>  Y[7:0]	[H] 04
>  OP[3:0]	[B] 0001 ▾
 C_vec	[B] 1
 Nf_vec	[B] 0
 V_vec	[B] 0
 Z_vec	[B] 0

Figura 11: Resta - $0x05 - 0x01 \rightarrow 0x04$, con $C=1$, $V=0$ (sin borrow).









Name	Value	A
>  A[7:0]	[H] 00 ▾	
>  B[7:0]	[H] 01 ▾	
>  Y[7:0]	[S] -1	
>  OP[3:0]	[B] 0001 ▾	
 C_vec	[B] 0	
 Nf_vec	[B] 1	
 V_vec	[B] 0	
 Z_vec	[B] 0	

Figura 12: Resta - $0x00 - 0x01 \rightarrow 0xFF$, con $C=0$ (borrow).









Name	Value
>  A[7:0]	[H] AA ▼
>  B[7:0]	[H] 55 ▼
>  Y[7:0]	[H] FF
>  OP[3:0]	[B] 0100 ▼
 C_vec	[B] 0
 Nf_vec	[B] 1
 V_vec	[B] 0
 Z_vec	[B] 0

Figura 13: XOR - 0xAA XOR 0x55 → 0xFF (complementarios).

Name	Value
>  A[7:0]	[B] 1111_0000 ▼
>  B[7:0]	[U] 4 ▼
>  Y[7:0]	[B] 1111_1111
>  OP[3:0]	[B] 0111 ▼
 C_vec	[B] 0
 Nf_vec	[B] 1
 V_vec	[B] 0
 Z_vec	[B] 0

Figura 14: SRA(0xF0, 4) → 0xFF (extensión correcta del bit de signo).

La validación sobre la FPGA permitió comprobar no solo la correcta implementación lógica, sino también la integración del bloque con las herramientas de depuración de Vivado. El flujo completo desde la descripción en VHDL hasta la verificación en hardware demuestra la síntesis exitosa y funcionalidad real del diseño.

6. Conclusiones

El desarrollo de la ALU permitió aplicar conceptos de diseño combinacional en hardware reconfigurable, validando el flujo completo desde la descripción VHDL hasta la ejecución sobre la FPGA. La arquitectura implementada logró cumplir con todos los requerimientos funcionales, demostrando un comportamiento correcto en simulación y en pruebas reales.

El banco de pruebas (tb_alu.vhd) verificó exhaustivamente las operaciones aritméticas, lógicas, de desplazamiento y comparación, asegurando la correcta actualización de las banderas de estado (Z, C, V, Nf). Los resultados obtenidos en simulación coincidieron plenamente con los valores teóricos, y la verificación en hardware mediante el núcleo VIO confirmó el mismo comportamiento en tiempo real.

La síntesis en la FPGA Arty Z7-10 mostró un uso mínimo de recursos, sin violaciones de temporización, lo que evidencia la eficiencia del diseño y su potencial integración dentro de un procesador simple o como coprocesador aritmético. La estructura parametrizable de la ALU también facilita su extensión a diferentes anchos de palabra o la incorporación de nuevas operaciones.

En conclusión, el trabajo cumplió con los objetivos propuestos:

- Implementar una ALU combinacional parametrizable en VHDL.
- Validar su funcionamiento mediante simulación exhaustiva.
- Sintetizar e implementar el diseño sobre FPGA, verificando su desempeño en hardware real.

El resultado es un bloque funcional, compacto y verificable, que puede servir como base para proyectos más avanzados de arquitectura de procesadores o sistemas digitales programables.