DISEÑO LÓGICO

Práctica 2

Diseño de la Unidad Aritmético Lógica

Práctica 2

Diseño de la Unidad Aritmético Lógica

2.1 Objetivos

En esta práctica se pretende que el alumno:

- Repase los conceptos relacionados con la unidad aritmético lógica de un sistema basado en microprocesador.
- Diseñe una unidad aritmético lógica de 32 bits para números enteros.
- Amplíe sus conocimientos sobre el lenguaje VHDL.

2.2 Material

El único material necesario para la realización de esta práctica es el paquete de software Xilinx, instalado en el ordenador del laboratorio.

2.3 Problema propuesto

En esta práctica se tiene que realizar el diseño de la unidad aritmético lógica para el procesador que se ha estudiado en la asignatura Estructura y Tecnología de Computadores (para los alumnos de la escuela) o Fundamentos de Computadores (para los alumnos de la facultad).

En ETC/FCO se diseñó un sencillo procesador con su ruta de datos y unidad de control que permitía ejecutar las siguientes instrucciones relacionadas con la unidad aritmético lógica:

```
add, sub, and, or y slt
```

Es decir, puede realizar sumas y restas de números enteros de 32 bits, así como las operaciones lógicas AND y OR, también sobre 32 bits, y, por último, comparar dos operandos para saber si uno es mayor que otro.

Además, como recordaréis del curso anterior, la Unidad de Control necesita recibir, además, el indicador de si el resultado es cero (Z), para las instrucciones de salto. Este indicador lo proporciona también la unidad aritmético lógica.

2.4 Descripción de la práctica

En esta sección vamos a describir cómo realizar la unidad aritmético lógica descrita en el apartado anterior de una manera incremental. Para ello, y tras presentar la definición de la entidad VHDL a realizar, descompondremos el diseño en estructuras menores: empezaremos con el diseño de un sencillo sumador, a continuación realizaremos una ALU de 1 bit (las dos versiones que veremos que se necesitan), para alcanzar, al final, el diseño completo que incluirá todos los diseños realizados previamente.

Obviamente, como se ha remarcado en las clases de teoría, hay diversos estilos de realizar un diseño de un circuito en VHDL. Entre ellos podemos destacar el comportamental, el estructural, o el mixto basado en los dos anteriores. En esta práctica pretendemos realizar el diseño con un estilo de tipo estructural del que destacamos a continuación las ideas principales.

En primer lugar, aunque debe estar ya claro, vamos a presentar la definición de la entidad (la ALU completa a realizar). La unidad aritmético lógica tendrá la siguiente definición de entradas y salidas **(NOTA.- Para que los diseños realizados por l@s alumn@s sean compatibles con el modelo de la ruta de datos que se proporcionará en la práctica 5, se recomienda que se utilicen las declaraciones de las entidades como se indican en los enunciados)**:

```
entity ALU is
  Port (OP_Ual : in  STD_LOGIC_VECTOR (2 downto 0);
        Op1 : in  STD_LOGIC_VECTOR (31 downto 0);
        Op2 : in  STD_LOGIC_VECTOR (31 downto 0);
        Resultado: out  STD_LOGIC_VECTOR (31 downto 0);
        Z : out  STD_LOGIC);
end ALU;
```

donde `Op1` y `Op2` son los dos operandos, `Resultado` es, como su nombre indica, el resultado de la operación a realizar, `Z` es la salida que valdrá 1 si el resultado es cero y `OP_Ual` son las líneas de control que le indicarán a la unidad aritmético lógica el tipo de instrucción a realizar. En la Tabla 1 se muestran los códigos que indican el tipo de operación a realizar por la ALU.

Como hemos dicho, vamos a realizar un ***diseño estructural basado en componentes*** de la unidad aritmético lógica. Para ello, comenzaremos realizando, como en la práctica inicial del curso, un **sumador de 1 bit.** Tras el sumador de un bit realizaremos una **unidad aritmético lógica de 1 bit**. Para ello, esta unidad de 1 bit integrará al sumador de un bit junto con el resto de operaciones lógicas y la de *menor que*. Para finalizar, realizaremos la **unidad aritmético lógica de 32 bits** mediante la integración de 32 unidades de 1 bit. Vamos a detallar, a modo de ayuda, cada uno de los diseños en las secciones siguientes.

OP_UAL (3 BITS)			OPERACIÓN
OP_UAL2	OP_UAL1	OP_UAL0	
0	0	0	SUMA (+)
0	0	1	RESTA (-)
0	1	0	SUMA LÓGICA (OR bit a bit)
0	1	1	PRODUCTO LÓGICO (AND bit a bit)
1	0	0	MENOR QUE (<)

Tabla 1. Códigos de operación de la UAL.

2.4.1 Sumador de 1 bit

En la práctica 1 se planteó ya el diseño de un sumador de 1 bit. Este sumador tiene como entradas, los dos operandos de 1 bit (A y B) y el acarreo de entrada Cin. Como salidas, el resultado S y el acarreo de salida Cout.


```
entity sumador is
  Port (A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        Cin : in  STD_LOGIC;
        S : out  STD_LOGIC;
        Cout : out  STD_LOGIC);
end sumador;

architecture Behavioral of sumador is
begin

 S <= ((not A) AND (not B) AND Cin) OR ( (not A) and
      B and (not Cin)) OR (A AND (not B) AND (not Cin))
      OR (A   and B and Cin);
```

```
 Cout <= (A and B) or (A and Cin) or (B and Cin);
end Behavioral;
```

2.4.2 Unidad aritmético lógica de 1 bit

A partir del sumador de 1 bit, vamos a realizar la unidad aritmético lógica de 1 bit que integre las operaciones que necesita nuestro procesador. Aplicaremos la metodología de diseño *"Bottom-Up"* y, por ello, vamos a partir de una sencilla ALU de 1 bit a la que iremos añadiendo la circuitería y funciones necesarias para cumplir su misión.

En la Figura 1 se muestra el esquema de una ALU que es capaz de sumar y realizar las operaciones AND y OR. Como se puede apreciar, el multiplexor controlado por las entradas llamadas "operación" determina qué salida es la que llega al exterior, es decir, realmente se realizan en paralelo todas las funciones, seleccionándose mediante este multiplexor el resultado de la operación indicada. Nótese que el bloque indicado con el símbolo "+" corresponde al sumador de 1 bit realizado en la sección anterior.

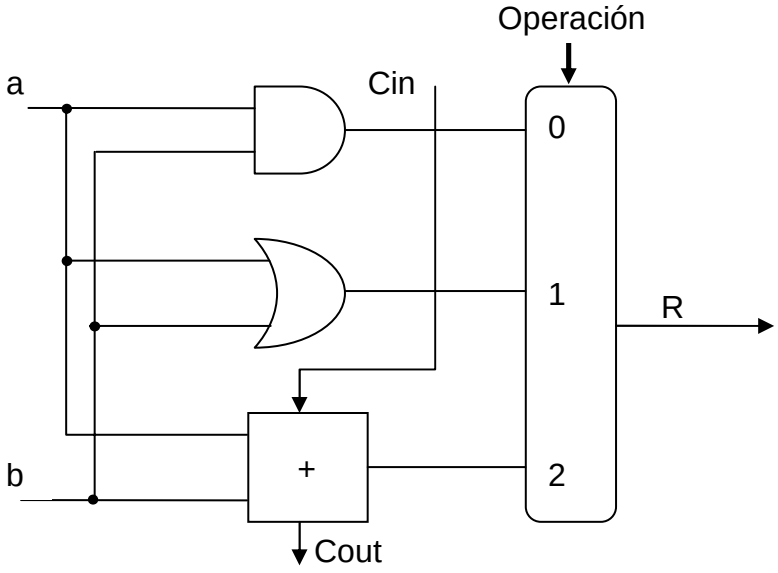

Figura 1. ALU de 1 bit que realiza las operaciones AND, OR y suma.

Además de las funciones implementadas, la ALU debe de ser capaz de restar. Para ello, como sabéis, restar es igual que sumar la versión negativa del operando. De esta forma, negar un número en complemento a dos es invertir cada

bit (lo que se denomina *complemento a uno*) y después sumar 1. Con el fin de invertir cada bit, sencillamente añadiremos un multiplexor 2 a 1 que escoja entre el operando b o /b en función de una señal de control llamada *Binvert*. Por último, para sumar 1 al complemento a 1 haremos que el bit Cin valga 1 cuando se seleccione la operación de resta. Por tanto, el esquema conceptual queda como se detalla en la Figura 2.

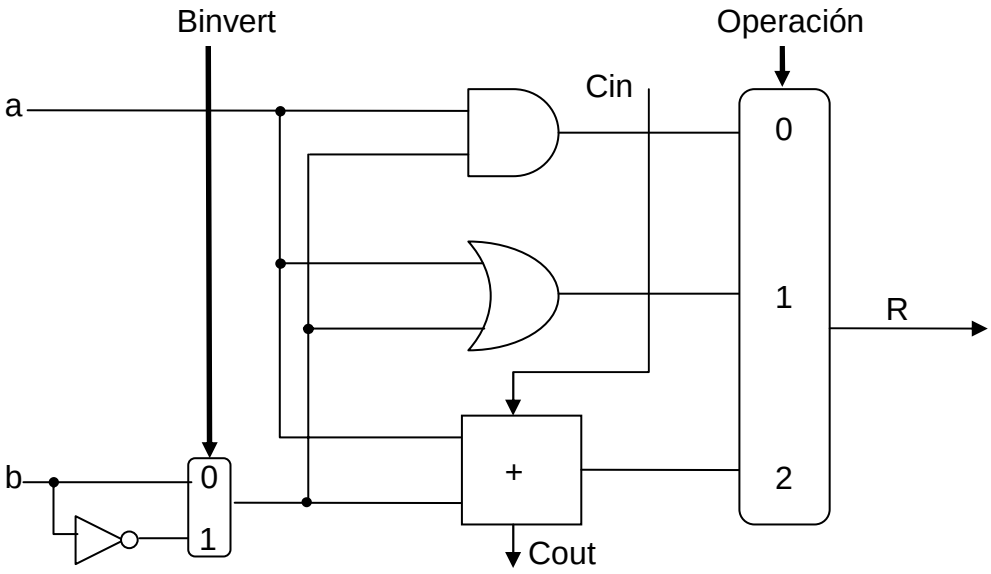

Figura 2. ALU de 1 bit que realiza las operaciones AND, OR, suma de a y b o a y /b.

Por último, una instrucción que necesita soporte de la ALU es la de *inicializar si menor que* (*set-on-less-than*). Recordad que esta operación genera 1 si Rs < Rt, y 0 en otro caso. Ej:

```
slt $1,$2,$3 significa que
      if ($2 < $3)
            then $1 = 1;
            else $1 = 0;
```

Por lo tanto, la instrucción *inicializar si menor que* (slt) pondrá el bit menos significativo del resultado a 0 ó 1 dependiendo del resultado de la comparación (los restantes bits, del 1 al 31, se dejarán a cero). Así, necesitaremos expandir el multiplexor para lograr el valor de la comparación "menor que" para cada bit de la ALU. La idea es tener una entrada nueva en la ALU, llamada `Less`, que sea el resultado para la operación `slt`. Esta entrada se tendrá que establecer a

0 o a 1 para generar el resultado correcto cuando se ordene una instrucción `slt`. Concretamente, se ha de hacer *algo* para que la entrada `Less` del bit de menor peso de la ALU se establezca a 0 a 1 y el resto de bits a 0.

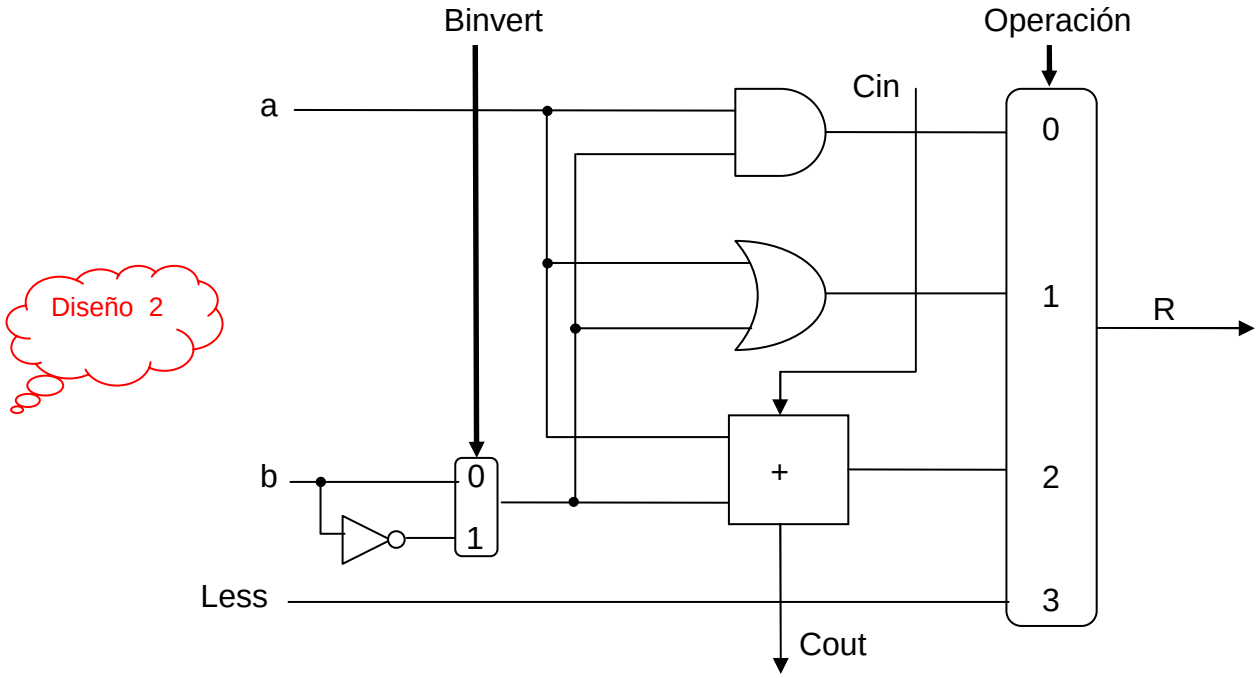

Figura 3. ALU de 1 bit que realiza las operaciones AND, OR, suma de a y b o a y /b e incluye una entrada directa que se conecta para realizar la operación de inicializar si menor que.

Lo que nos queda por pensar es cómo comparar y poner el bit menos significativo para la instrucción inicializar si menor que: en esta instrucción, si restamos el contenido del registro Rt del contenido del registro Rs y la diferencia es negativa, entonces es que Rs < Rt, ya que:

⇨ (Rs-Rt) < 0
⇨ ((Rs-Rt) + Rt) < (0+Rt)
⇨ (0 + Rs) < (0 + Rt)
⇨ Rs < Rt

Por lo tanto, queremos que el bit menos significativo del resultado sea 1 si la diferencia es negativa y 0 si es positiva. Esto tiene una correspondencia exacta con los valores del bit de signo de la resta de ambos datos: 1 significa negativo y 0 positivo. Siguiendo este razonamiento, **sólo necesitaremos conectar el bit de**

signo de la salida del sumador tras realizar la resta de los dos operandos, al bit menos significativo del resultado (entrada `Less` de la ALU del bit de menor peso) para obtener lo que queremos. Es decir, cuando se tenga que ejecutar esta instrucción, la ALU, realmente, ejecutará la resta y analizando el bit de signo estableceremos el resultado.

Pero, desgraciadamente, el bit más significativo de la ALU para la operación "inicializar si menor que" no es la salida del sumador; la salida de la ALU para la operación menor es el valor de entrada `Less` (Menor). Por ello, necesitamos una nueva ALU para el bit más significativo (**sólo**), tal que la salida de la suma esté disponible como salida. La Figura 4 muestra el diseño, con esta nueva línea llamada **`Set`** (que se conectará a la entrada `Less` del bit menos significativo cuando se genere la ALU de 32 bits, estando el resto de entradas `Less` de los bits 1 a 31 a '0'). La figura 5 muestra el esquema completo de la ALU en el que se pueden clarificar estas ideas. Nótese que todas las entradas `Less` de los bits 1 a 31 están a cero y la salida del sumador (`Set`) de la ALU del bit de mayor peso se conecta a la entrada `Less` de la ALU del bit de menor peso.

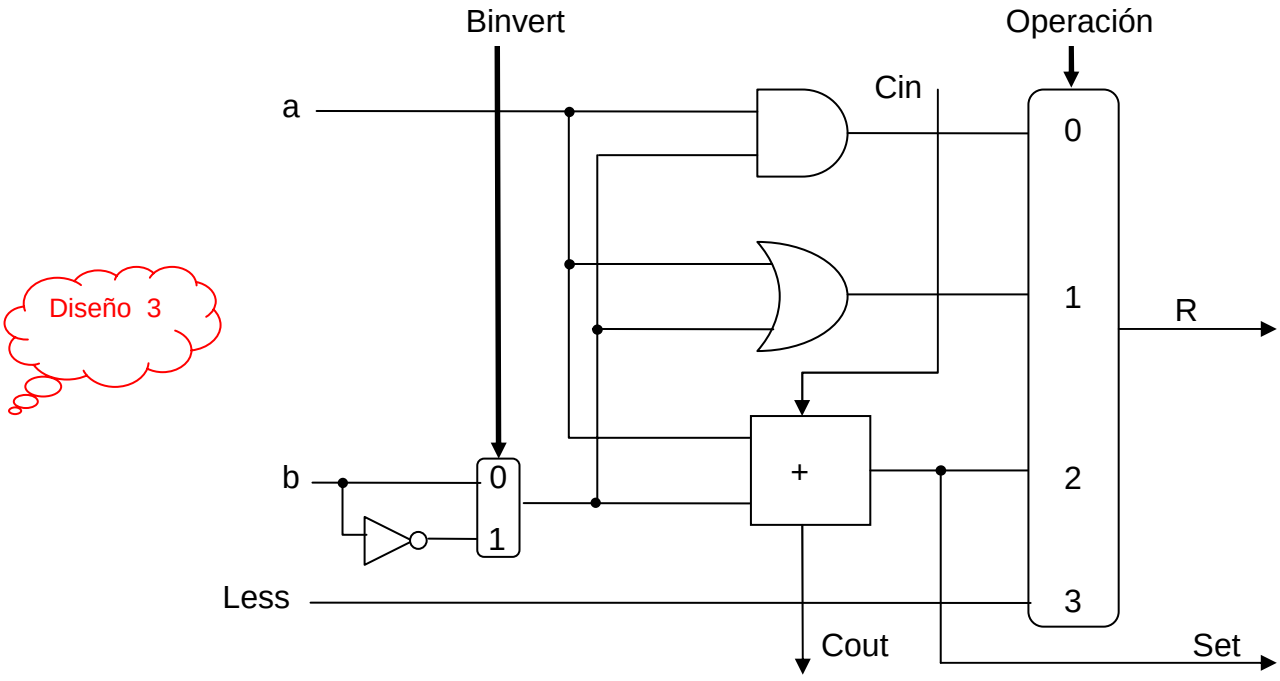

Figura 4. ALU de 1 bit para el bit más significativo

(*Nota: el test sobre menor que es un poco más complicado de lo descrito si tenemos en cuenta un posible desbordamiento. Dado el objetivo de la práctica no consideramos necesario tenerlo en cuenta*).

2.4.3 Unidad aritmético lógica de 32 bits

Realizar la unidad arimético lógica de 32 bits no es más que reunir las unidades de 1 bit que hemos descrito previamente: concretamente tendremos que usar 31 unidades de 1 bit como las esquematizadas en la Figura 3 más la del bit más significativo, representada en la Figura 4. Estas unidades, como hemos visto incluirán al sumador básico presentado en la sección 2.4.1.

Una vez reunidas, hemos de añadir el indicador sobre si el resultado es Cero (Z). Este indicador es necesario para poder realizar instrucciones de salto condicional. Estas instrucciones (como recordaréis de primer curso), producen un salto a otra posición del código o cuando dos registros son iguales o cuando no lo son. La forma más fácil, por tanto, es restar el contenido de los dos registros y ver si el resultado es o no cero para decidir sobre la condición de salto. Por esto, necesitamos añadir a nuestra ALU de 32 bits el indicador Z que no será más que la negación de la OR de los 32 bits del resultado. El esquema final de la ALU se puede ver en la Figura 5.

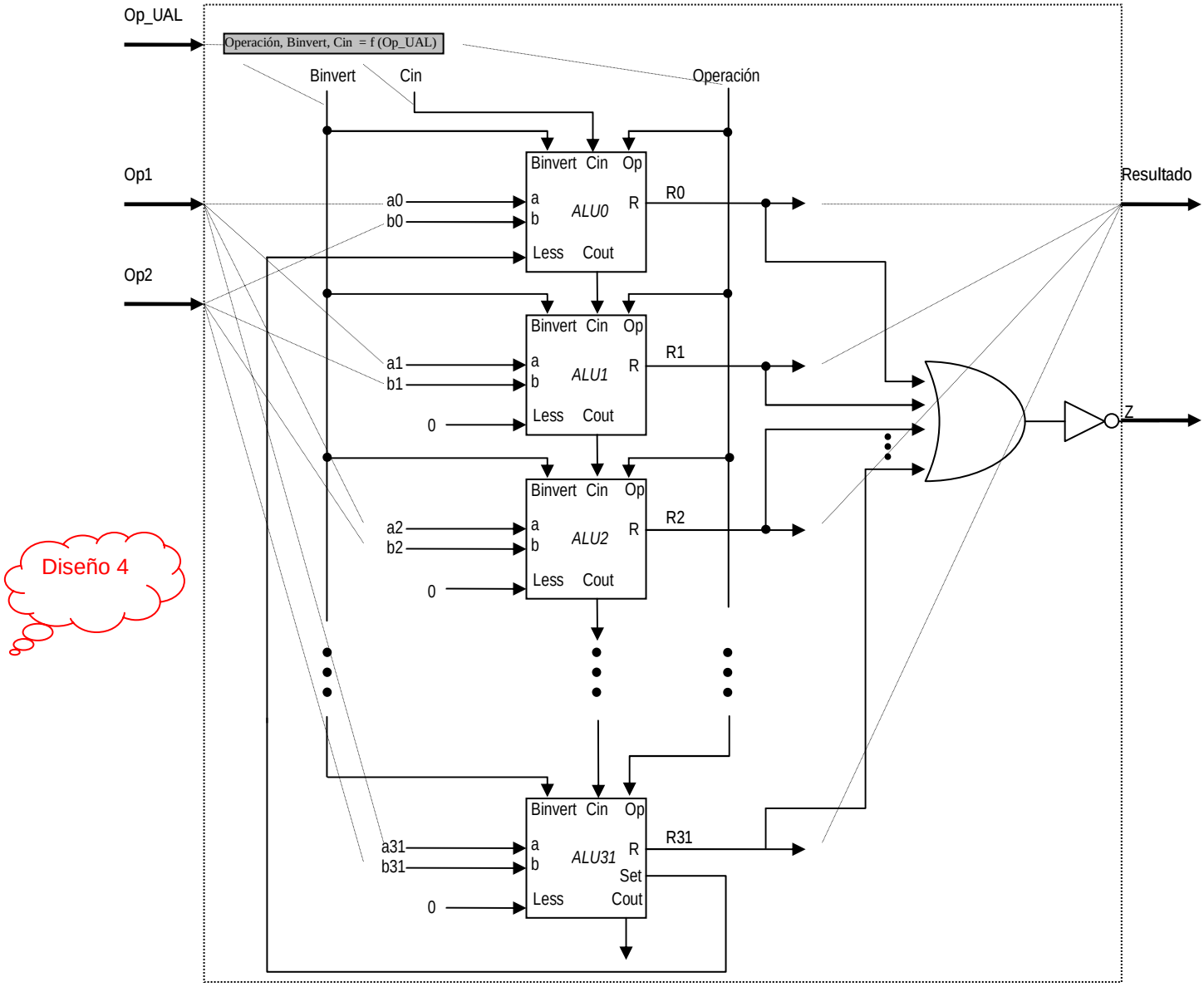

Figura 5. ALU final de 32 bit con detector de cero

2.5 Ejercicios

2.5.1 Ejercicio 1

Realizar el diseño correspondiente al sumador de la sección 2.4.1 y verifique su correcto funcionamiento con el simulador de la herramienta.

2.5.2 Ejercicio 2

Realizar el diseño correspondiente a la unidad aritmético lógica de un bit (Figura 3) y verifique su correcto funcionamiento con el simulador de la herramienta. A partir de esta unidad aritmético lógica, realizar las modificaciones pertinentes para construir la del bit más significativo (Figura 4).

2.5.3 Ejercicio 3

Construya la unidad aritmética de 32 bits (Figura 5) a partir de los diseños anteriores y verifique su funcionamiento con la ayuda del simulador.

En todos los casos, los alumnos presentarán al profesor de prácticas el código VHDL realizado para cada uno de los ejercicios, así como las simulaciones que demuestren el funcionamiento de los circuitos realizados.

2.6 Código de apoyo

```
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;



entity ALU1bit is
  Port (A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        CarryIn : in  STD_LOGIC;
        Operacion :in  STD_LOGIC_VECTOR (1 downto 0);
        Less: in STD_LOGIC;
        BInvert : in  STD_LOGIC;
        Result : out  STD_LOGIC;
        CarryOut: out STD_LOGIC);
end ALU1bit;

architecture Behavioral of ALU1bit is
component sumador is
  Port (A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        Cin : in  STD_LOGIC;
        S : out  STD_LOGIC;
        Cout : out  STD_LOGIC);
end component;

signal

begin

……
sumar: sumador port map(…,…,…,…,…);

…….

end Behavioral;
```

Diseño 2


```
entity ALU1bitMSB is
  Port (A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        CarryIn : in  STD_LOGIC;
        Operacion: in  STD_LOGIC_VECTOR (1 downto 0);
        Less: in STD_LOGIC;
        BInvert : in  STD_LOGIC;
        Result : out  STD_LOGIC;
        CarryOut: out STD_LOGIC;
        Set: out STD_LOGIC);
end ALU1bitMSB;

architecture Behavioral of ALU1bitMSB is
component sumador is
  Port (A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        Cin : in  STD_LOGIC;
        S : out  STD_LOGIC;
        Cout : out  STD_LOGIC);
end component;
signal ……..;

begin

……..
sumar: sumador port map(…,…,…,…,…);

-- el bit de signo msb (si no hay overflow) genera el
-- bit de menor peso para slt

end Behavioral;
```

```
entity ALU is
  Port (OP_Ual : in  STD_LOGIC_VECTOR (2 downto 0));
        Op1: in STD_LOGIC_VECTOR (31 downto 0);
        Op2: in STD_LOGIC_VECTOR (31 downto 0);
        Resultado:out STD_LOGIC_VECTOR (31 downto 0);
        Z : out  STD_LOGIC;
end ALU;

architecture ALU_Archl of ALU is
component ALU1bit is
  Port (A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        CarryIn : in  STD_LOGIC;
        Operacion: in  STD_LOGIC_VECTOR (1 downto 0);
        Less: in STD_LOGIC;
        BInvert : in  STD_LOGIC;
        Result : out  STD_LOGIC;
        CarryOut: out STD_LOGIC);
end component;
component ALU1bitMSB is
  Port (A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        CarryIn : in  STD_LOGIC;
        Operacion: in  STD_LOGIC_VECTOR (1 downto 0);
        Less: in STD_LOGIC;
        BInvert : in  STD_LOGIC;
        Result : out  STD_LOGIC;
        CarryOut: out STD_LOGIC;
        Set: out STD_LOGIC);
end component ;

signal     ……..;


begin
……
……

-- suma 2, and 0, or 1, slt 3
Operacion(0) <= '1' when ((OP_Ual = "010") OR (OP_Ual =
"100")) else '0';
```

Diseño 4

```
Operacion(1) <= '1' when ((OP_UAL = "000") or (OP_UAL =
"100") or (OP_Ual = "001")) else '0';

BInvert <= '1' when ((OP_UAL = "001") or (OP_UAL =
"100")) else '0';
Cin <= BInvert;


sum0: ALU1bit port map (…,…,…, …, …, …, …, …);
sum1: ALU1bit port map (…,…,…, …, …, …, …, …);
----- completar hasta sum30:

sum31: ALU1bitMSB port map (…,…,…,…,…,…,…,…,…,…);

Resultado <= ….;
Z <= …….

end ALU_Archl;
```