Ingeniería de Computadores 3

Ásel Martínez Leirós

Contenido

Ejercicio 1

Se nos pide diseñar un circuito que implemente las funciones F y G representadas en la siguiente tabla

x	y	z	F	G
'0'	'0'	'0'	'0'	'1'
'0'	'0'	'1'	'1'	'1'
'0'	'1'	'0'	'0'	'0'
'0'	'1'	'1'	'1'	'1'
'1'	'0'	'0'	'0'	'0'
'1'	'0'	'1'	'0'	'0'
'1'	'1'	'0'	'1'	'0'
'1'	'1'	'1'	'1'	'0'

En primer lugar, obtenemos las funciones:

F = X'Y'Z + X'YZ + XYZ' + XYZ

G = X'Y'Z' + X'Y'Z + X'YZ

En segundo lugar, procedemos a realizar el mapa de Karnaugh de ambas funciones para simplificarlas y utilizar el menor número de puertas lógicas posibles:

F

Z \ XY	00	01	11	10
0	0	0	1	0
1	1	1	1	0

Tras realizar la simplificación, F = X'Z + XY

G

Z \ XY	00	01	11	10
0	1	0	0	0
1	1	1	0	0

Tras realizar la simplificación, G = X'Z + X'Y'

Entity-Architecture Circuito

Una vez obtenidas ambas funciones, escribimos la entity del circuito el cual las implemente:

```
library IEEE;
use IEEE.std_logic_1164.all;

--Entity
entity Circuito is
    port(

        --3 Señales entrada
        X,Y,Z : in std_logic;

        --2 Señales salida
        F,G   : out std_logic

    );
end entity;
```

Entity circuito en CIRCUITO.vhd

Como podemos observar, para implementar la entidad circuito, hemos utilizado la función port a través de la cual le daremos las señales de entrada y de salida a nuestro circuito, en este caso:

X, Y, Z como entrada utilizando in std_logic

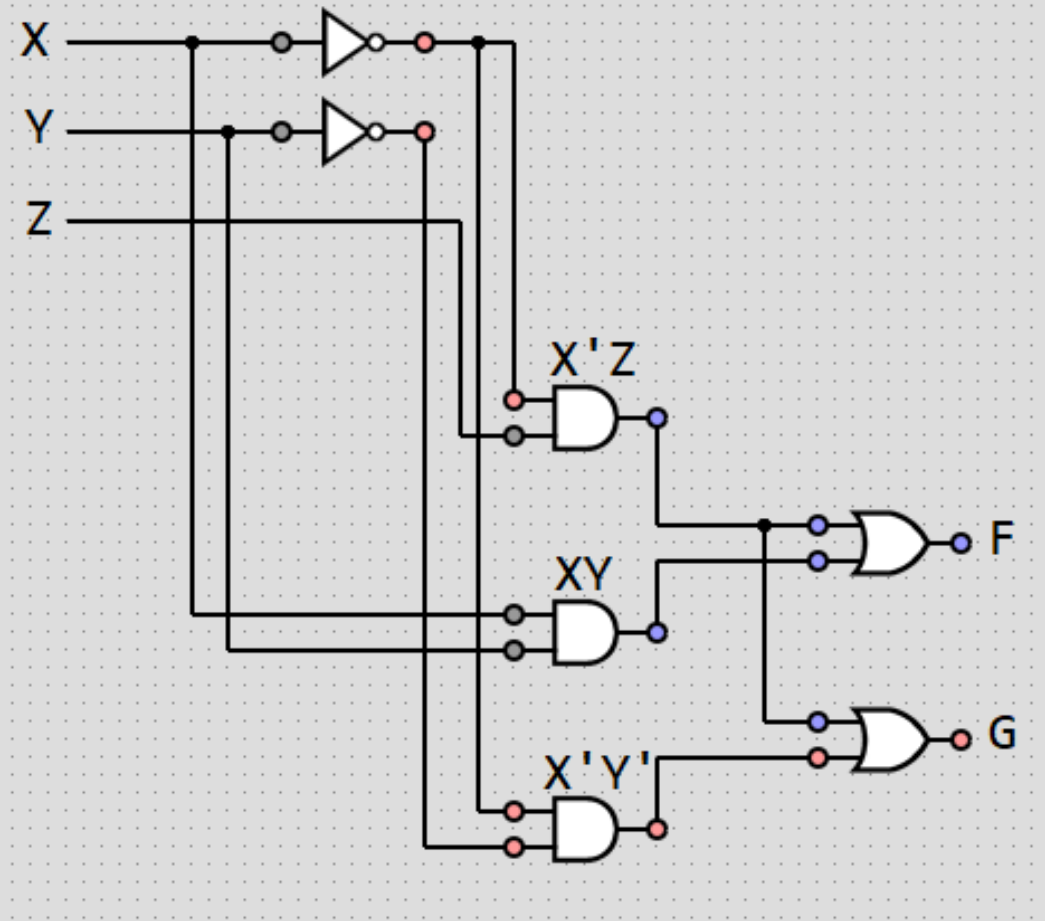
F, G como salida utilizando out std_logic

Tras ello, procedemos a implementar la arquitectura de este:

```
--Architecture
architecture Comp of Circuito is
begin

    --F = x'z+xy
    F <= ((not x) and z) or (x and y);

    --G = x'z+x'y'
    G <= ((not x) and z) or ((not x) and (not y));

end architecture;
```

Le damos un comportamiento al circuito según las funciones sacadas realizando los mapas de Karnaugh en apartado anterior.

Diagrama del circuito



Como observamos en el diagrama, vamos a utilizar 2 puertas NOT, 3 puertas AND y 2 puertas OR, por ello tendremos que implementar la entity y architecture de cada uno de los tipos de puerta lógica a utilizar

Entity-Architecture NOT

```
library IEEE;
use IEEE.std_logic_1164.all;

--Entity puerta NOT
entity not1 is

    port (

        --1 señal de salida
        y0 : out std_logic;
        --1 señales de entrada
        x0 : in std_logic );

end entity;

--Arquitectura puerta NOT (funcionamiento)
architecture not1 of not1 is
begin

    y0 <= not x0;

end architecture;
```

NOT.vhd

Como vemos en el diagrama, cada puerta NOT que utilizamos tiene una entrada (x0) y una salida (y0) lo cual implementamos en la entity, y su funcionamiento lo implementamos en su architecture diciéndole que en la salida debemos tener lo negado de lo que entra.

Entity-Architecture AND

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

--Entity AND
entity and2 is

    port (

        --1 señal salida
        y0 : out std_logic;

        --2 señales entrada
        x0, x1 : in std_logic

    );

end entity;

--Arquitectura puerta AND (funcionamiento)
architecture and2 of and2 is
begin

    y0 <= x0 and x1;

end architecture;
```

AND.vhd

Al igual que en el apartado anterior, en el diagrama observamos que cada AND utilizada tiene 2 entradas (x0, x1) y 1 salida (y0) lo cual implementamos en la entity. El comportamiento es implementado en el architecture diciéndole que a la salida debemos tener la función lógica AND de ambas entradas.

Entity-Architecture OR

```
library IEEE;
use IEEE.std_logic_1164.all;

--Entity puerta OR
entity or2 is

    port (

        --1 señal de salida
        y0 : out std_logic;
        --2 señales de entrada
        x0, x1 : in std_logic );

end entity;

--Arquitectura puerta OR (funcionamiento)
architecture or2 of or2 is
begin

    y0 <= x0 or x1;

end architecture;
```

OR.vhd

Por último, implementamos la última de las puertas lógicas, en la cual podemos ver que tendremos dos entradas (x0, x1) y una salida (y0)

En su architecture implementaremos su funcionamiento, a través del cual queremos que en la salida tengamos la funcion OR de ambas entradas.

Architecture de la estructura del circuito

```
architecture Estruc_Circuito of Circuito is

signal not_x_and_z, xy, not_x_and_not_y, not_x, not_y: std_logic;

-- Declaración de las clases de los componentes
component and2 is
    port (

        y0 : out std_logic;

        x0, x1 : in std_logic

    );
end component;

component not1 is
    port (

        y0 : out std_logic;

        x0 : in std_logic
    );
end component;

component or2 is
    port (

        y0 : out std_logic;

        x0, x1 : in std_logic
    );
end component;

begin
-- Instanciación y conexión de los componentes
not1_1 : component not1 port map (not_x, x);
not1_2 : component not1 port map (not_y, y);
and2_1 : component and2 port map (not_x_and_z, not_x, z);
and2_2 : component and2 port map (xy, x, y);
and2_3 : component and2 port map (not_x_and_not_y, not_x, not_y);
or2_1 : component or2 port map (F, xy, not_x_and_z);
or2_2: component or2 port map (G, not_x_and_not_y, not_x_and_z);
end architecture Estruc_Circuito;
```

CIRCUITO.vhd

Para implementar la estructura del circuito, implementamos todas las señales en el circuito, las podemos ver en el diagrama, tenemos:

X’Z

XY

X’Y’

X’

Y’

Después, declaramos todos los componentes que vamos a utilizar, en primer lugar el componente NOT1 con 1 salida y 1 entrada, en segundo lugar el componente AND2 con 2 entradas y 1 salida y por último el componente OR2 con 2 entradas y una salida

Para finalizar, tendremos que instanciar y conectar los componentes entre sí, lo cual podemos observar en la última parte de la imagen

Banco de pruebas

```
-- Banco de pruebas
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


entity BP_Circuito is
end entity BP_Circuito;

architecture BP_Circuito of BP_Circuito is

    signal F, G : std_logic; -- Conectar salidas UUT
    signal x0, x1, x2 : std_logic; -- Conectar entradas UUT

component Circuito is
    port(
        F, G : out std_logic;
        x, y, z: in std_logic
    );
end component Circuito;

begin
-- Instanciar y conectar UUT
    uut : component Circuito port map
    (
        F => F, G => G,
        x => x0, y => x1, z => x2);

gen_vec_test : process
variable test_in : unsigned (2 downto 0); -- Vector de test
begin
test_in := B"000";
for count in 0 to 7 loop
x2 <= test_in(2);
x1 <= test_in(1);
x0 <= test_in(0);
wait for 10 ns;
test_in := test_in + 1;
end loop;
wait;
end process gen_vec_test;
end architecture BP_Circuito;
```

BANCOPRUEBAS.vhd

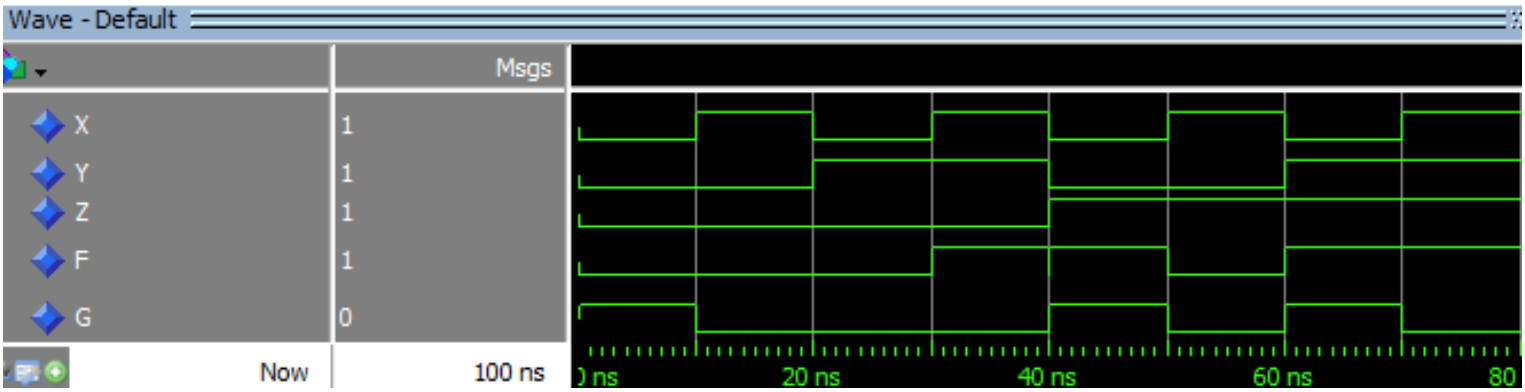
En este banco de pruebas, crearemos la entidad y le daremos el comportamiento en la arquitectura,

donde tendremos las señales X, Y, Z (x0, x1, x2) y las señales F y G.

Después crearemos un componente circuito el cual tendrá como salida F, G y como entradas X, Y, Z asignados a cada una de las señales F, G, x0, x1, x2.

En la última parte tendremos el test a realizar, el cual utilizando un bucle for dará todos los valores posibles al circuito, cambiando las entradas cada 10ns

El resultado de la prueba lo observamos en el siguiente cronograma, sacado a través de la pantalla Wave del ModelSim y la función simulación de este:



Si nos fijamos en el cronograma, cada 10 ns cambia los valores de las entradas (X, Y, Z) y según las entradas, variarán también las salidas, vamos a realizar la comprobación de que el circuito está realizando las funciones tal cual se nos había pedido

En los primeros 10ns, tenemos:

X = 0 Y = 0 Z = 0 F = 0 G = 1

De 10ns a 20ns tenemos:

X = 1 Y = 0 Z = 0 F = 0 G = 0

De 20ns a 30ns tenemos:

X = 0 Y = 1 Z = 0 F = 0 G = 0

De 30ns a 40ns tenemos:

X = 1 Y = 1 Z = 0 F = 1 G = 0

De 40ns a 50ns tenemos:

X = 0 Y = 0 Z = 1 F = 1 G = 1

De 50ns a 60ns tenemos:

X = 1 Y = 0 Z = 1 F = 0 G = 0

De 60ns a 70ns tenemos:

X = 0 Y = 1 Z = 1 F = 1 G = 1

De 70ns a 80ns tenemos:

X = 1 Y = 1 Z = 1 F = 1 G = 0

Lo cual, si comprobamos con la tabla dada en el enunciado, podremos comprobar que el circuito que hemos implementado está realizando las funciones correctamente, ya que coinciden todos los resultados de las salidas para los mismos valores de entrada.

Ejercicio 2

Planteamiento

Según el enunciado, tenemos que sumar dos números de n bits en complemento a dos, los cuales podrían ser negativos o positivos, teniendo en cuenta un bit de acarreo de entrada.

Lo he planteado de la siguiente manera:

Crear en primer lugar un valor constante N, indicándolo como GENERIC de la entidad CircuitoCombinacional, que marcará el número de bits que tendrán los datos de entrada a y b.

Tras ello, en el apartado PORT, crear las entradas y salidas, donde para mi parecer, lo más lógico es crear las entradas y la suma de ellas con tipo de dato SIGNED, el cual tiene en cuenta el bit de mas valor para su signo.

Como tendremos que tener en cuenta el acarreo de entrada, la solución que he visto más viable ha sido la de crear esta entrada cin como tipo de dato std_logic, la cual luego se verá apoyada en una señal

auxiliar de tipo signed para poder ser sumada en el CircuitoCombinacional con a y b.

Entity

El resultado de la entity sería el siguiente:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;


--ENTITY CIRCUITO COMBINACIONAL
ENTITY CircuitoCombinacional IS

	--CONSTANT GENERIC QUE INDICARÁ EL NÚMERO DE BITS DE LOS CUALES ESTARÁN
	GENERIC(
		CONSTANT n	: INTEGER := 4
	);
	PORT(
		--SALIDAS
		res		: OUT signed(n-1 DOWNTO 0) := (others => '0');
		desbordamiento	: OUT std_logic := '0';
		cero		: OUT std_logic := '0';
		signo		: OUT std_logic := '0';
		--ENTRADAS
		a		: IN signed(n-1 DOWNTO 0);
		b		: IN signed(n-1 DOWNTO 0);
		cin		: IN std_logic
	);

END ENTITY;
```

También vemos que hemos declarado las señales de salida desbordamiento, cero y signo, tal y como pide el enunciado.

Architecture

Dentro de la parte architecture, donde implementaremos el comportamiento de nuestro circuito, en primer lugar, declaramos las señales:

CinAux -> Esta señal nos ayudará a la hora de hallar el resultado final (res), ya que al ser de tipo SIGNED, el lenguaje es capaz de sumarlo sin realizar ningún tipo de función o escritura extra. Como vemos un poco más debajo de donde está declarada, lo único que hacemos es cambiar su bit de menor peso según si cin es '1' o '0'.

S -> Esta señal nos ayudará a poder acceder como lectura al resultado (res), que nos hará falta para hallar más adelante signo y cero.

checkD -> Esta señal nos ayudará a poder acceder como lectura al desbordamiento, que nos hará falta para hallar más adelante el signo.

Código Architecture

```
ARCHITECTURE Comportamiento OF CircuitoCombinacional IS

		--SEÑALES AUXILIARES
		SIGNAL cinAux,S : signed (n-1 DOWNTO 0) := (others => '0');
		SIGNAL checkD   : std_logic;

BEGIN

	--USAREMOS LA SEÑAL cinAux PARA SUMARLE 1 A NUESTRO CIRCUITO COMBINACIONAL
	--PARA PODER SUMARSELO, TENEMOS QUE TENER UNA VARIABLE DE TIPO SIGNED
	--ES POR ELLO QUE CREAMOS ESTE BUCLE WHEN-ELSE
	cinAux(0) <= '1' when cin = '1' else
	'0';

	--SUMARIAMOS LAS SEÑALES TIPO SIGNED A, B Y CINAUX, LO CUAL NOS DEVOLVERIA EL
	--VALOR DE NUESTRO RESULTADO FINAL
	S <= a + b + cinAux;
	res <= S;

	--BUCLE PARA VERIFICAR SI HAY DESBORDAMIENTO, CUANDO EL BIT DE MAS VALOR DE A SEA DISTINTO
	--DEL BIT DE MÁS VALOR DE B O CUANDO EL BIT DE MAS VALOR DE A, DE B Y DEL RESULTADO SEAN IGUALES,
	--EL DESBORDAMIENTO SE FIJARÁ A '0'
	checkD <= '0' when a(n-1) /= b(n-1) else
	'0' when a(n-1) = b(n-1) and a(n-1) = S(n-1) else
	'1';

	desbordamiento <= checkD;

	--LA SEÑAL CERO SERÁ '1' SOLO CUANDO EL VALOR DEL RESULTADO CONVERTIDO A INTEGER SEA 0 Y QUE EL VALOR
	--DE LA SEÑAL DESBORDAMIENTO SEA '0', SI NO, SU VALOR SERÁ '0'.
	cero <= '1' when to_integer(S) = 0  and checkD = '0' else
	'0';

	--LA SEÑAL SIGNO, SERÁ EL BIT DE MAYOR VALOR DEL RESULTADO SIEMPRE Y CUANDO NO HAYA DESBORDAMIENTO, SI NO,
	--SERA EL OPUESTO AL BIT DE MAYOR VALOR DEL RESULTADO.
	signo <= S(n-1) when checkD = '0' else
	not S(n-1);



END ARCHITECTURE;
```

Una breve explicación de lo que hace nuestro circuito, si vemos la imagen del código, tras el BEGIN:

Le damos el valor de cin a la señal de apoyo cinAux, que es de tipo SIGNED y tras ello, sumamos a + b + cinAux para darle valor a la señal res y a S.

Para hallar el desbordamiento, hemos utilizado una estructura WHEN-ELSE, bastante sencilla ya que no necesita de un proceso para poder ejecutarse, lo que hará esta estructura será darle un valor a la variable checkD, el cual será '0' si el bit de signo de a y b son diferentes o si el bit de signo de a, b y res son iguales, si no, le dará el valor '1'. Tras asignarle un valor, también le asignará el mismo a la señal desbordamiento.

Para hallar la señal cero, lo que hice es, mediante el uso de la funcion to_integer(), la cual convertirá nuestro SIGNED a integer, verificar si nuestro resultado (res) es 0, en caso de serlo y no haber desbordamiento, pondrá la señal cero a '1', en cualquier otro caso, la pondrá a '0'.

Para hallar la señal signo, utilizando de nuevo un WHEN-ELSE verificaremos que sí no hay desbordamiento, la señal

signo será el bit de signo del resultado (res) y en caso de haberlo, será el contrario, lo cual, indicamos con un NOT.

Banco de Pruebas

Para el diseño del banco de pruebas, he pensado en un diseño que sea escalable a cualquier tamaño de entrada de datos, tal y como pide el enunciado, es por ello, que hemos declarado en esta parte también una constante de tipo integer, la cual modificará el tamaño de las palabras a, b y res.

En cuanto al funcionamiento del banco de pruebas, implementado en su architecture, podemos decir, que he pensado en un bucle for a través del cual se comprueben todas las entradas posibles en a,b y cin.

Teniendo en cuenta que tenemos n bits para cada palabra, si por ejemplo n es 4, tenemos 4 bits de entrada por parte de a, 4 bits de entrada por parte de b y 1 bit de entrada de cin, es decir, tendríamos 9 bits, que equivaldría a 2^9 = 512 casos posibles. Transformándolo a una expresión algebraica sería (2^(N*2)+1), con un bucle que vaya de 1 a 2^(N*2)+1 podriamos recorrer todos los casos posibles con estas entradas, ya que, a mayores,

dentro de este bucle for, tendremos otros dos bucles for que le darán valor a cada uno de los bits que componen las palabras a,b y cin.

Si nos fijamos, a y b siempre tendrán el mismo tamaño y cin siempre será 1 bit, es por ello, que, para darles valor a cada uno de estos bits, se creará una palabra auxiliar con n bits de a + n bits de b + 1 bit de cin, es decir (n*2)+1 bit, a esta señal auxiliar le llamamos vectorPrueba, de tipo SIGNED y declarado como n*2 downto 0 bits.

Con ello, a cada loop del bucle for le iremos sumando 1 a este vectorPrueba, y le daremos el valor de cada uno de los bits de este vector a nuestras palabras de entrada de la siguiente forma:

Contando que cin siempre será un bit, le daremos a cin el mismo valor que tenga el bit de menos valor del vectorPrueba, es decir, vectorPrueba(0).

Y ahora tendríamos los bits sobrantes del vectorPrueba para darles valor a las palabras a y b,

Hacemos un for que vaya desde 1 a n para darle valor a los bits de a, y con ayuda de una variable auxiliar (auxA), introduciremos el valor del bit en el índice de a correspondiente, lo mismo sucederá

para b, el cual usará un bucle de n+1 a n*2 para recibir los valores de sus bits.

La primera parte del Architecture:

```
Architecture Comportamiento of BP_CircuitoCombinacional is

	COMPONENT CircuitoCombinacional
		PORT(
			res			: OUT signed(n - 1 DOWNTO 0);
			desbordamiento	: OUT std_logic;
			cero			: OUT std_logic;
			signo			: OUT std_logic;
			a			: IN signed(n - 1 DOWNTO 0);
			b			: IN signed(n - 1 DOWNTO 0);
			cin			: IN std_logic
		 );
	END COMPONENT;

	--señales salida
	SIGNAL res			: signed(n - 1 DOWNTO 0);
	SIGNAL desbordamiento	: std_logic;
	SIGNAL cero			: std_logic;
	SIGNAL signo			: std_logic;
	SIGNAL errores		: integer := 0;

	--señales entrada
	SIGNAL a			: signed(n - 1 DOWNTO 0);
	SIGNAL b			: signed(n - 1 DOWNTO 0);
	SIGNAL cin			: std_logic;

BEGIN
	--RELACIONAMOS LAS ENTRADAS Y SALIDAS
	instancia:  COMPONENT CircuitoCombinacional
		PORT MAP(
			res			=> res,
			cero			=> cero,
			desbordamiento	=> desbordamiento,
			signo			=> signo,
			a			=> a,
			b			=> b,
			cin			=> cin
		);
```

Segunda parte de Architecture:

```
test: PROCESS
        --VARIABLES A UTILIZAR EN EL PROCESO TEST
        variable vectorPrueba           : signed(n*2 downto 0) := (others=>'0');
        variable varA, varB, S          : signed(n-1 downto 0);
        variable varC                   : signed(n-1 downto 0) := (others=>'0');
        variable AuxA, AuxB             : integer := 0;
        variable int1                   : integer;

        BEGIN
                --SE RECORRERÁ ESTÉ BUCLE TANTAS VECES COMO 2^(N*2)+1, PARA PODEI
                FOR i IN 1 TO 2 ** ((n*2)+1) LOOP

                        --ESTE BUCLE LE DA VALOR A B, UTILIZAMOS AUXB COMO INDICI
                        FOR j IN 1 TO n LOOP
                                varA(AuxB) := vectorPrueba(j);
                                AuxB := AuxB+1;
                        END LOOP;

                        --DEVOLVEMOS VALOR 0 A AUXB
                                AuxB := 0;

                        --ESTE BUCLE LE DA VALOR A A, UTILIZAMOS AUXA COMO INDICI
                        FOR k IN n+1 TO n*2 LOOP
                                varB(AuxA) := vectorPrueba(k);
                                AuxA := AuxA+1;
                        END LOOP;

                        --DEVOLVEMOS 0 A AUXA
                        AuxA := 0;
                        --A CIN LE DAREMOS EL VALOR DEL BIT MAS BAJO DE VECTORPRU
                        a <= varA;
                        b <= varB;
                        cin <= vectorPrueba(0);

                        if(vectorPrueba(0) = '1')then
                                varC(0) := '1';
                        else
                                varC(0) := '0';
                        end if;

                        S := varA + varB + varC;
                        int1 := to_integer(varA)+to_integer(varB);

                        --CONDICIONAL QUE VERIFICARÁ SI EL RESULTADO ES CORRECTO
                        if(vectorPrueba(0) = '1')then
                                int1 := int1+1;
                        end if;
```

Tercera y última parte del Architecture:

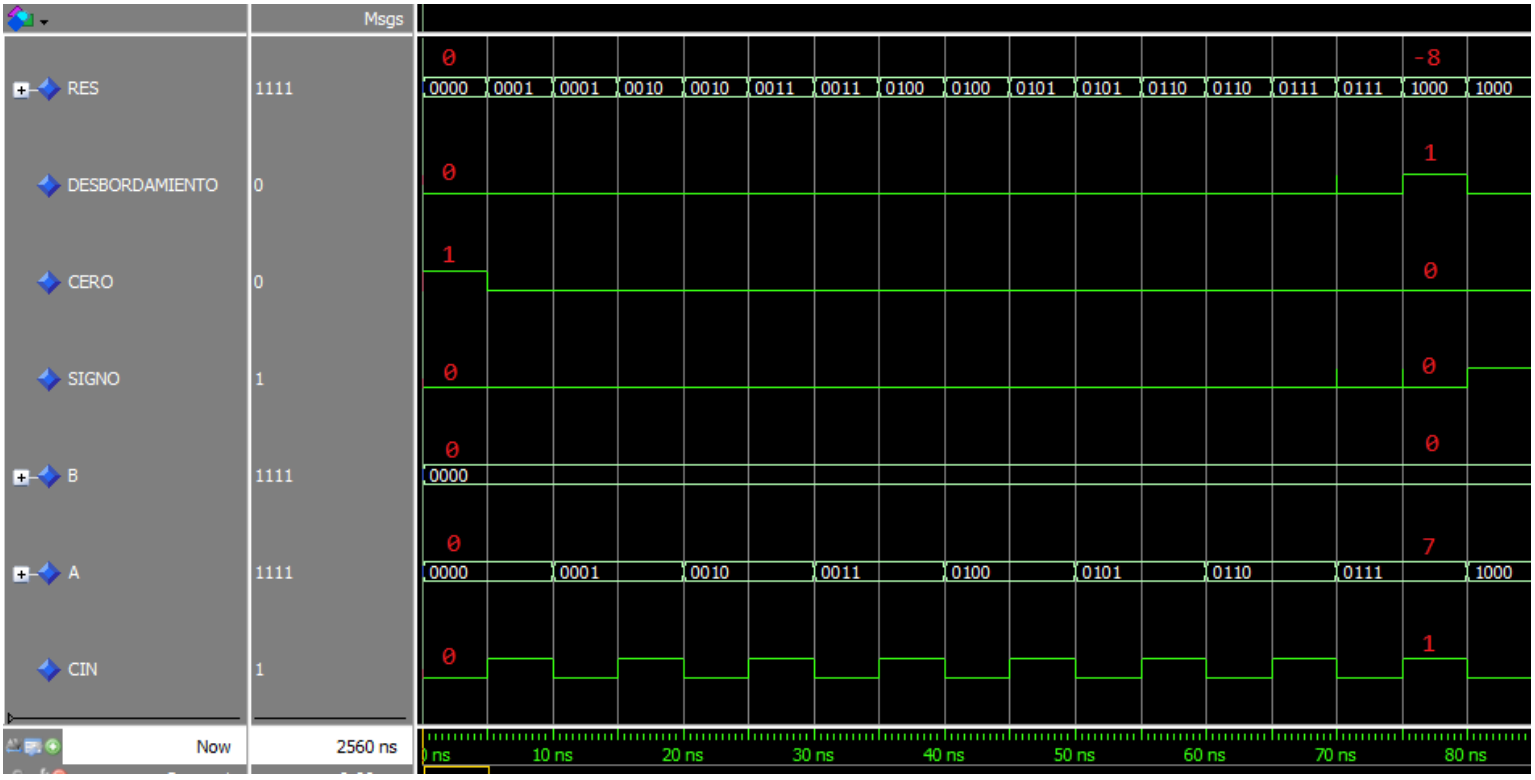
```
                        if(intl = to_integer(S))then
                        else
                                report "Resultado esperado INCORRECTO, se esperaba " & integer'
                                errores <= errores+1;
                        end if;

                        --ESPERAMOS 5 NS PARA INICIAR EL BUCLE PRINCIPAL DE NUEVO
                        wait for 5 ns;

                        --LE SUMAMOS 1 AL VECTOR DE PRUEBA, A TRAVÉS DEL CUAL TOMAMOS LOS BITS
                        --ES DECIR, SI CADA PALABRA SON 4 BITS, Y TENEMOS 2 PALABRAS, NECESITAR
                        --PARA CONTAR CON TODOS LOS VALORES POSIBLES DE TEST, POR ELLO ESTE VEC
                        vectorPrueba := vectorPrueba + 1;

                END LOOP;
                          --REPORT FINAL CON EL NÚMERO DE ERRORES TOTALES
                          report "Test completado con " & integer'image(errores) & " errores";
                WAIT;
        END PROCESS test;
END ARCHITECTURE;
```

Como observamos, también comprobaremos que los resultados dados por el programa son los buenos mediante un condicional que comparará el resultado final S (transformado a integer con la funcion to_integer) con la suma de las señales a y b transformadas a integer, en caso de no ser igual, nos lo reportará por consola y le sumará 1 a la variable “errores”, a través de la cual al final utilizaremos para reportar el número total de errores.

Prueba para n = 4

Cada 5 ns obtendremos un valor nuevo para nuestras señales de entrada, por ello, al ser n = 4, necesitaremos 2^9 = 512 casos posibles * 5 ns = 2560 ns para la ejecución completa de nuestro bucle que da valores a las señales de entrada

Ejecutando una prueba de 2560 ns, obtenemos un cronograma tal que:



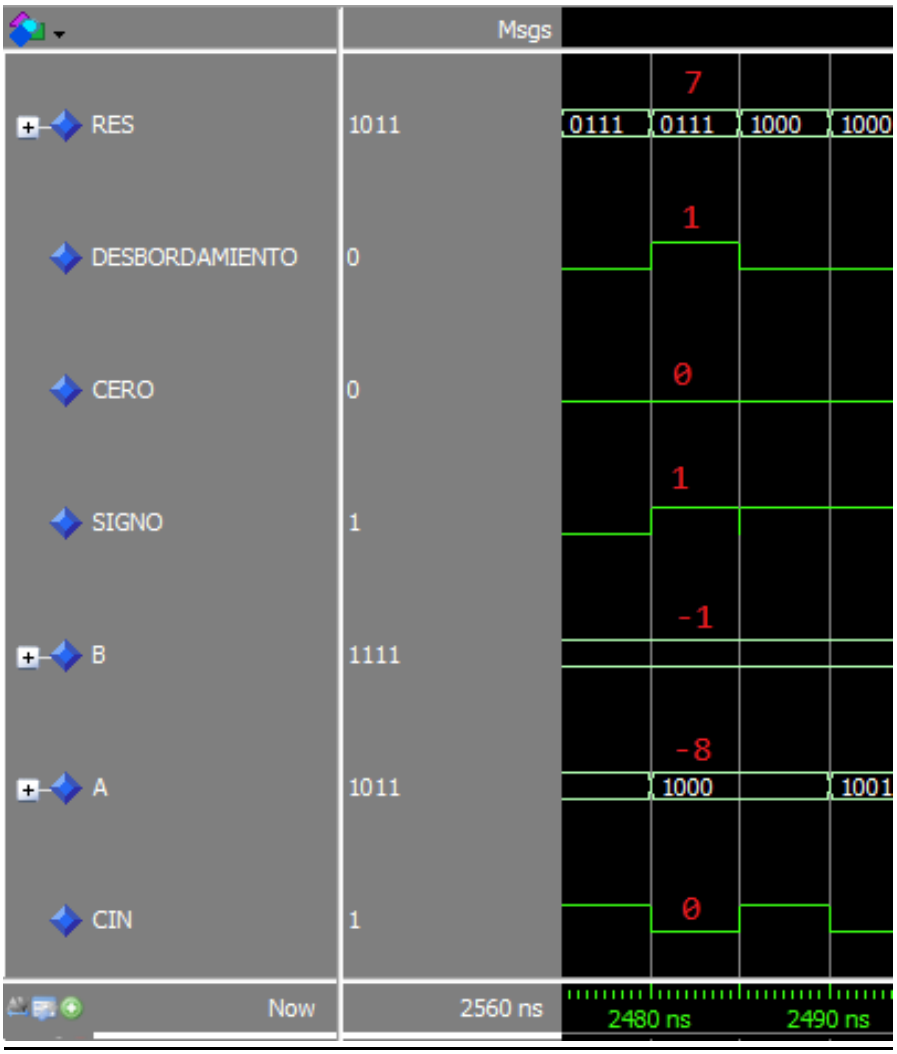
Y por pantalla se nos mostrará un mensaje

```
#    Time: 0 ps  Iteration: 1  Instance: /bp_circuitocombinacional/instancia
# ** Note: Resultado esperado INCORRECTO, se esperaba 8 y el resultado ha sido -8
#    Time: 75 ns  Iteration: 0  Instance: /bp_circuitocombinacional
```

En la imagen del cronograma, podemos ver que el programa está funcionando correctamente, solo que el resultado que nos saca el programa en el caso del primer error (a los 75ns), es 1000 (-8), cuando está sumando 7 de a y 1 de cin, lo cual debería de ser 8, pero esque en realidad está sacándonos 8, ya que está también marcando con 1 que hay desbordamiento y con 0 el signo del resultado, con lo cual podríamos confirmar que el programa está funcionando correctamente. También podríamos verlo en el siguiente caso, el cual sería el último error mostrado por pantalla antes de darnos el número total de errores:

```
# ** Note: Resultado esperado INCORRECTO, se esperaba -9 y el resultado ha sido 7
#    Time: 2480 ns  Iteration: 0  Instance: /bp_circuitocombinacional
# ** Note: Test completado con 128 errores
```

Si nos vamos a los 2480 ns, donde indica resultado incorrecto:



Vemos que -8 + -1 debería de mostrarnos un -9 (10111), pero el resultado al no poder dar 5 bits, nos muestra solo 4 de ellos (0111), pero si que podemos ver que el signo está a 1, con lo cual el resultado estaría correcto.

El resto del cronograma se puede ver en el archivo CRONOGRAMA N = 4 dentro de la carpeta ejercicio 2.