TÍTULO DE LA ACTIVIDAD: **Subsistemas aritméticos**	**CÓDIGO:** **TEXTO**

FECHA:			
NOMBRE:		APELLIDOS:	

MODALIDAD:	Texto	TIPO:		DURACIÓN:	

CALENDARIO:		REQUISITOS:	

CRITERIO DE ÉXITO:	

COMENTARIOS E INCIDENCIAS:

TIEMPO DEDICADO:	minutos	AUTOEVALUACIÓN: [entre 0 y 10 puntos]	No procede

PARTE I. Introducción

Este texto está dedicado al estudio de los subsistemas aritméticos. En principio, podemos considerar subsistema aritmético a cualquier módulo digital capaz de realizar una operación matemática, pero esta definición engloba a un grupo de subsistemas digitales de muy variada complejidad e interés práctico; de entre ellos este texto sólo analizará los más simples que, por otra parte, son también los más frecuentemente utilizados.

Modelo Lógico de los subsistemas aritméticos

Para definir el modelo lógico de un subsistema aritmético hay que conocer:

- El tipo de operación que realiza

  El tipo de operación (suma, comparación, multiplicación, etcétera) que realiza el subsistema determina su función y, en gran medida, su interfaz. Las operaciones básicas de mayor interés práctico son:

  - Las comparaciones de magnitud: "mayor", "menor", "igual" y "distinto".
  - La suma y la resta
  - La multiplicación y la división

  Hay subsistemas aritméticos que materializan funciones matemáticas más complejas (raíz cuadrada, logaritmo, etcétera). También hay subsistemas versátiles, capaces de realizar diversos tipos de operaciones: un sumador-restador, por ejemplo, es un sistema que, según convenga, puede sumar o restar dos números.

  En el ámbito de este texto se estudiarán los sumadores, restadores y comparadores, así como algunos casos particulares de multiplicadores y divisores.

- El código y longitud de los números con que operan

  Ambas características influyen en la definición de la interfaz del sistema. Los códigos más empleados son Binario Natural y Complemento a Dos, pero hay subsistemas que operan con números codificados en BCD y en punto flotante.

  En este texto se estudian únicamente subsistemas que realizan cálculos con números codificados en binario natural o en complemento a 2.

- La naturaleza de los operandos

  Hay subsistemas que realizan operaciones en las que uno de los operandos es una constante (los incrementadores, por ejemplo). Los subsistemas aritméticos que operan con constantes tienen un único operando de entrada y su estructura interna es más simple que la de los subsistemas que, realizando la misma operación, tienen como entrada dos operandos.

En este texto se estudiará cómo afecta a los subsistemas aritméticos el hecho de que uno de sus operandos tenga un valor constante.

- El carácter combinacional o secuencial del subsistema.

  Aunque esta característica repercute, principalmente, en la velocidad del subsistema –en el número de operaciones por segundo que puede realizar– y en el número de recursos que hay que emplear en su realización, también afecta a la interfaz: si es un subsistema secuencial dispondrá de entrada de reloj e inicialización asíncrona y, casi con toda seguridad, de una entrada de inicialización síncrona; también ocurre que los subsistemas aritméticos secuenciales pueden disponer de interfaces en las que se ingresan los operandos en serie.

  Este texto está dedicado exclusivamente al estudio de subsistemas aritméticos combinacionales que, por otra parte, son los comúnmente utilizados para la realización de operaciones aritméticas simples.

Modelado de la interfaz

El modelo de la interfaz de los subsistemas aritméticos se obtiene fácilmente a partir de la determinación de las características anteriores para cada caso particular. Por ejemplo, si un subsistema calcula cuál es el mayor de dos números de 8 bits codificados en binario natural, su interfaz queda completamente definida y puede representarse mediante un símbolo como el de la figura 1a. El símbolo de la figura 1b representa la interfaz de un sumador de números de 8 bits con acarreo de entrada.

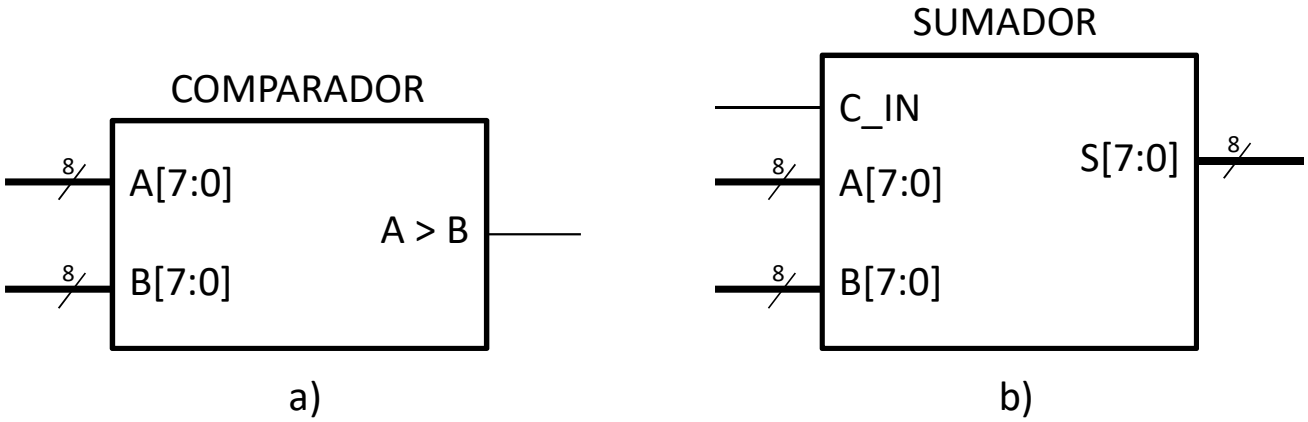


Figura 1.- Símbolos de subsistemas aritméticos

El modelo VHDL de la interfaz de ambos circuitos sería:

INTERFAZ DEL COMPARADOR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity comp_8b is
port(
     A: in std_logic_vector(7 downto 0);
     B: in std_logic_vector(7 downto 0);
     A_may_B: buffer std_logic
    );
```

INTERFAZ DEL SUMADOR

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity sum_8b_c_in is
port(
     C_in: in std_logic;
     A: in std_logic_vector(7 downto 0);
     B: in std_logic_vector(7 downto 0);
     S: buffer std_logic_vector(7 downto 0)
    );
```

```
end entity;
```

```
end entity;
```

Modelado del funcionamiento

Desde la perspectiva de las metodologías clásicas de diseño, los subsistemas combinacionales aritméticos que realizan operaciones simples (sumas, restas y comparaciones) son, a nivel lógico y salvo contadas excepciones, sistemas combinacionales complejos, es decir, sistemas en los que no resulta viable, o razonable, realizar una descripción detallada del funcionamiento (con una tabla de verdad) –la descripción de la función del sumador de la figura 1b, por ejemplo, requeriría la realización de una tabla de verdad de 2^{17} filas (131.072 filas) y 25 columnas.

Esta característica obliga a abordar el diseño lógico de los subsistemas aritméticos como una estructura de módulos simples interconectados. Debido a la importancia práctica de este tipo de subsistemas, el diseño de arquitecturas óptimas para su realización es un problema resuelto hace décadas, por lo que basta con conocer las soluciones más interesantes para la realización de cada tipo de operación y elegir aquella cuyas características se adapten mejor a las necesidades de la aplicación que se esté desarrollando.

La complejidad lógica de los subsistemas aritméticos lleva aparejadas dos características que son comunes a la mayor parte de las soluciones de diseño:

- Un considerable consumo de recursos *hardware*, que es proporcional, además, al número de bits de los operandos.

- Unos tiempos de propagación grandes proporcionales, también, a la longitud de los operandos.

El empleo de metodologías de diseño basadas en el uso de HDLs permite describir el funcionamiento de los subsistemas aritméticos mediante las operaciones matemáticas que realizan. Por ello, en estas metodologías los subsistemas aritméticos se pueden considerar sistemas simples. Por ejemplo, tanto el funcionamiento del comparador como el del sumador de la figura 1 pueden modelarse con un proceso VHDL:

FUNCIONAMIENTO DEL COMPARADOR

```
process(A, B)
begin
  if A > B then
    A_may_B <= '1';

  else
    A_may_B <= '0';
  end if;
end process;
```

FUNCIONAMIENTO DEL SUMADOR

```
process(C_in, A, B)
begin
  S <= A + B + C_in;

end process;
```

Por tanto, estos subsistemas, que desde la perspectiva de una metodología de diseño clásico son complejos, resultan ser simples (muy simples) observados dentro del contexto del diseño basado en HDLs –que los sistemas aritméticos puedan tener una estructura lógica muy compleja, compuesta por un gran número de células lógicas, y que, además, puedan tener tiempos de propagación

notablemente grandes no puede ser ignorado al aplicar metodologías de diseño en que se emplean HDLs.
En definitiva, aunque se empleen metodologías basadas en el uso de HDLs, no puede perderse de vista el hecho de que los subsistemas que realizan operaciones aritméticas tienen características que pueden penalizar las prestaciones y el coste de los sistemas de los que forman parte.

Por eso en este texto se van a estudiar las arquitecturas para la realización de circuitos sumadores, restadores y comparadores, porque es muy importante que todo ingeniero dedicado al diseño de sistemas digitales cableados las conozca y tenga en cuenta sus características a la hora de abordar el diseño de sistemas complejos que contengan subsistemas aritméticos.

PARTE II. Sumadores

Un sumador[1] es un subsistema que realiza la suma de dos números. La interfaz de un sumador binario cuenta siempre con entradas para los sumandos y una salida para entregar el resultado de la suma:

- Entradas de sumandos

  Un sumador dispondrá, normalmente, de dos entradas para los sumandos, salvo en el caso de que uno de ellos se reduzca a un valor constante –tal y como ocurre, por ejemplo, en los incrementadores, que son sumadores en los que uno de los sumandos vale 1. Cuando se precisa disponer de un subsistema que realice la suma de más de dos números se pueden combinar varios sumadores (figura 2) o, alternativamente, puede recurrirse al empleo de un acumulador[2].

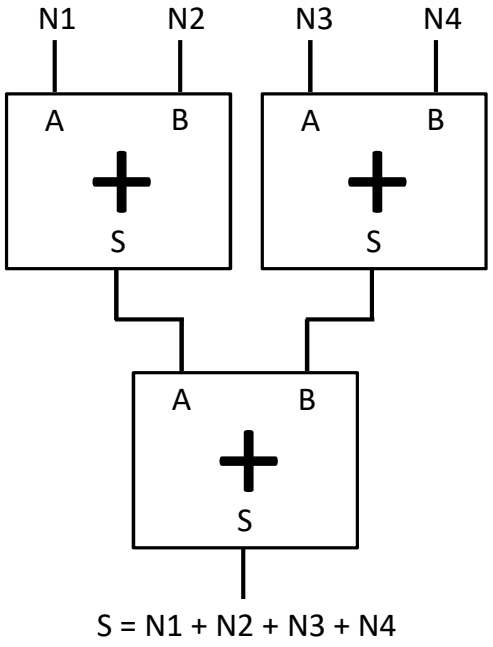


Figura 2.- Sumador de cuatro sumandos

- Salida de resultado

  La salida en la que se entrega el resultado de la suma suele tener el mismo número de bits que los sumandos. Por ello, y puesto que el resultado de la suma de dos números de N bits puede necesitar hasta N+1 bit para ser representado, en la salida de un sumador puede aparecer un valor incorrecto, produciéndose lo que se conoce como desbordamiento (*overflow*).

[1] Este apartado está dedicado exclusivamente al estudio de sumadores combinacionales, calificativo que, por tal motivo, no se utilizará en el texto pero que debe sobreentenderse en todo momento.

[2] Un acumulador es un subsistema aritmético secuencial que es capaz de acumular –de ahí su nombre- la suma de una serie indefinida de números. Es la solución preferible para sumar varios números, ya que estructuras como la de la figura 2 dan lugar a circuitos caros (su realización supone el empleo de muchas células lógicas) y lentos. El análisis y diseño de acumuladores se tratará más adelante en este mismo curso.

Los sumadores más frecuentemente utilizados, los sumadores binarios, son capaces de operar, indistintamente, con números en binario natural o complemento a dos, que son los códigos más utilizados para la representación de enteros con o sin signo.

Modelado VHDL de sumadores

Los modelos HDL sintetizables automáticamente pueden hacer uso de operaciones aritméticas básicas (sumas, restas y comparaciones). Por ello la realización de modelos de los subsistemas aritméticos básicos resulta muy sencilla. Por ejemplo, para realizar el modelo VHDL de un sumador binario hay que:

- Obtener visibilidad sobre el paquete *std_logic_unsigned*.
- Incluir en un proceso combinacional una sentencia en la que se asigna a la salida el valor de la suma de dos o más objetos.

Realice las siguientes operaciones:

1. Descomprima, en el directorio correspondiente, el fichero *BT1_A5.zip*.
2. Arranque ModelSim y cree el proyecto *Proy_BT1_A5*.
3. Añada a dicho proyecto los ficheros *sum4bits.vhd* y *sum4bits_tb.vhd*.
4. Abra, con el editor de ModelSim, el fichero *sum4bits.vhd*.

```
-- Modelo de sumador de 4 bits
-- realizado por MAFR para la
-- actividad BT1_AINP12 (30-5-2011).
-- Versión 1.0

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; -- para poder usar +

entity sum4bits is
port(
     A: in     std_logic_vector(3 downto 0);
     B: in     std_logic_vector(3 downto 0);
     S: buffer std_logic_vector(3 downto 0)
    );
end entity;

architecture rtl of sum4bits is
begin
  process(A, B) -- Modelado de la suma
  begin
    S <= A + B;

  end process;
end rtl;
```

El funcionamiento del sumador se modela en un proceso, sensible a los dos sumandos, A y B, por medio de una sentencia de asignación en la que se emplea la operación suma aplicada a dos

objetos de tipo *std_logic_vector*. Para que este uso no ocasione un error sintáctico, se ha utilizado una cláusula de visibilidad sobre el paquete *std_logic_unsigned*, que es donde está definida esta operación.
El lenguaje VHDL dispone de fórmulas sintácticas que permiten simplificar el código de procesos que contienen una única sentencia, son las denominadas *sentencias concurrentes*. Una *sentencia concurrente de asignación* permite reducir el código del modelo de funcionamiento del sumador a:

```
architecture rtl of sum4bits is
begin
    S <= A + B;
end rtl;
```

Una sentencia concurrente no es otra cosa que un proceso con una sintaxis modificada. La sentencia es sensible a las señales que están a la derecha de la asignación –se ejecuta cuando estas señales sufren eventos durante una simulación; en el caso del modelo del sumador, cuando A o B cambian de valor.

Es recomendable utilizar sentencias concurrentes siempre que sea posible, ya que mejoran la inteligibilidad del código y reducen el tiempo que se emplea en codificar y depurar los modelos.

5. En el modelo del sumador de 4 bits, sustituya el proceso por una sentencia concurrente –para ello basta con que borre todo el código del proceso menos la sentencia de asignación.

6. Abra, con el editor de ModelSim, el fichero *sum4bits_tb.vhd*.

Observe el proceso del *test-bench* que maneja los estímulos: genera todas las posibles combinaciones que pueden darse en los sumandos de entrada empleando bucles FOR. En este tipo de bucles se declara una variable implícita (en el ejemplo *i*, en el bucle externo, y *j*, en el interno) que controla las iteraciones.

```
-- Definición de estímulos
process
begin
  A <= "0000";
  B <= "0000";

  for i in 0 to 15 loop
    for j in 0 to 15 loop
      wait for 100 ns;
      B <= B + 1;

    end loop;
    A <= A + 1;

  end loop;
  wait;
end process;
```

Los bucles FOR son de gran utilidad en los *test-benches*, pues permiten generar un gran número de combinaciones de valores para los estímulos con un puñado de líneas de código. Por ejemplo, en el código del test del sumador:

- El bucle externo secuencia la asignación de dieciséis combinaciones diferentes de ceros y unos al sumando A.

- Durante la ejecución del bucle interno el valor de A permanece constante, por lo que el bucle interno prueba, para cada combinación de A, las dieciséis posibles combinaciones de B. Cada combinación se mantiene estable durante 100 nanosegundos. En consecuencia, con un código tan sencillo como el de este proceso, se generan las 256 combinaciones distintas de ceros y unos que puede haber en la entrada del sumador de 4 bits.

7. Compile el modelo del sumador y su *test-bench*.

8. Ejecute una simulación (*Run-All)* tras haber configurado el visor de formas de onda para que muestre las dos entradas, A y B, y la salida S del sumador.

9. Utilice los comandos de *zoom* para obtener una vista de los resultados de la simulación como los de la figura 3 –se trata de que pueda revisar las combinaciones en las que A vale "0100".

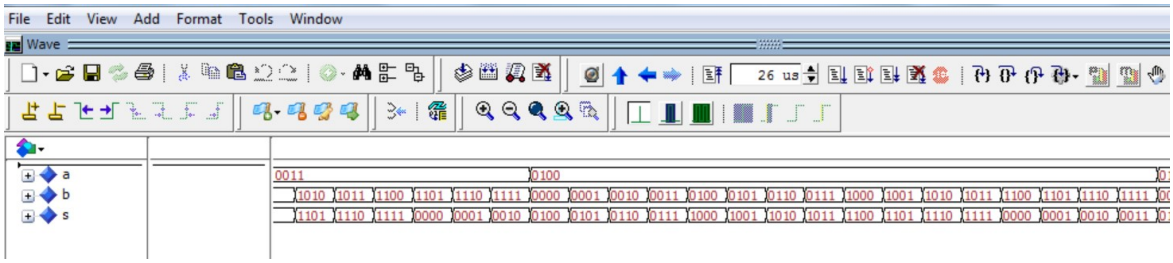


Figura 3.- Resultados de la simulación del sumador

Para facilitar la verificación del funcionamiento del sumador conviene cambiar el código con el que el visor de formas de onda muestra los valores de los buses. Si los sumandos y el resultado codifican los números en binario natural, conviene que el visor de formas de onda traduzca sus valores a números naturales en base decimal.

10. Seleccione con el ratón la señal A, pulse a continuación el botón derecho del ratón y, en el menú de *pop-up* que aparece, elija en la sección *Radix* la opción *Unsigned*.

11. Repita la operación para las señales B y S. Obtendrá una representación similar a la que se observa en la figura 4.

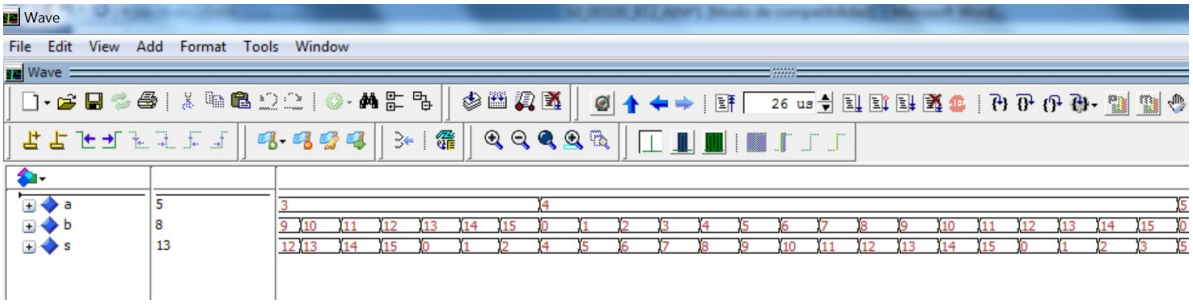


Figura 4.- Resultados de la suma de números naturales

Ahora resulta fácil verificar los resultados; puede comprobarse que mientras la suma de A y B es menor de 16, el valor en la salida es A + B, pero, como no podía ser de otra forma, cuando el resultado es mayor o igual que 16 no puede expresarse con cuatro bits –como

cuando, por ejemplo, se suman 4 y 14- y el valor de salida es incorrecto porque hay desbordamiento.

El modelo que se ha simulado corresponde a un sumador binario y, como ya se ha comentado, estos subsistemas son capaces de sumar, indistintamente, números en binario natural o en complemento a 2. Por tanto, si se utiliza este último código para interpretar el valor de los datos, el circuito debe realizar también sumas correctas –siempre y cuando no se produzca desbordamiento.

12. Para que el visor de formas de onda traduzca las cadenas de ceros y unos de acuerdo con el código complemento a 2 hay que repetir los pasos 10 y 11 seleccionando ahora, en lugar de la opción *unsigned*, la opción *decimal*.

    La ventana del visor deberá adoptar un aspecto similar al de la figura 5.

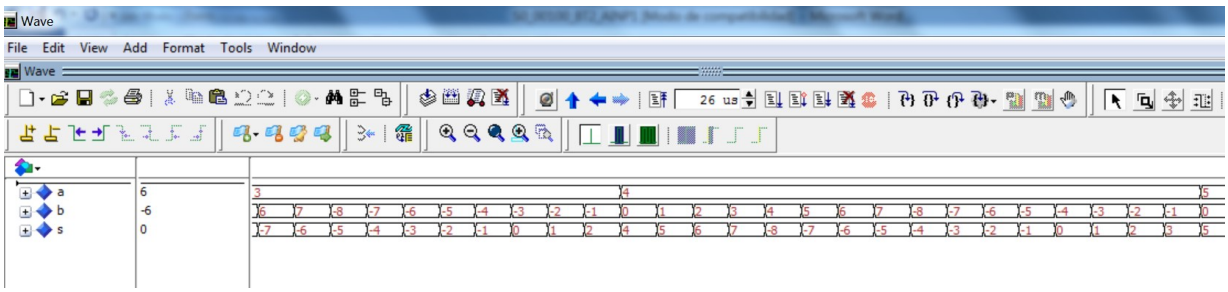


Figura 5.- Resultados de la suma de números enteros

Puede comprobarse que el resultado, cuando puede expresarse en complemento a 2 con cuatro bits (si está entre -8 y +7), es efectivamente la suma de A y B.

Entradas y salidas de acarreo

Además de las entradas correspondientes a los sumandos, y la salida para la suma, la interfaz de un sumador puede disponer de una entrada y una salida de acarreo. La salida de acarreo se pone a 1 cuando los sumandos generan acarreo. Esta salida puede tener tres usos distintos:

- Como bit de mayor peso del resultado, para evitar el desbordamiento.

  Es un uso poco frecuente de la salida de acarreo, ya que implica que los sumandos se codifican con N bits y el resultado con N+1 (los N bits de la salida de suma más el bit de acarreo), algo que no es posible aceptar en sistemas digitales que emplean una longitud de palabra fija para codificar datos –en los microprocesadores, por ejemplo, y por eso cuando suman y detectan que el resultado *no cabe* en los bits de la palabra que lo codifica, activan un *flag* en el registro de estado.

- Como *flag* para señalar la ocurrencia de desbordamiento en la suma cuando el sumador opera con números codificados en binario natural

  Cuando los números están en binario natural y el resultado de la suma requiere N+1 bits para ser codificado, es porque los dos bits de mayor peso generan acarreo y, en consecuencia, la salida de acarreo se pone a 1, luego detecta la ocurrencia de desbordamiento. Esta propiedad

no se verifica cuando los números están expresados en complemento a dos: hay que tener claro que, en este caso, la salida de acarreo no indica la ocurrencia de desbordamiento.

- Para entregar el valor del acarreo generado por los dos bits de mayor peso

  Es el uso originalmente propio de esta salida –de ahí su nombre, salida de acarreo. La salida de acarreo permite encadenar M sumadores de números de N bits, con entrada y salida de acarreo, para construir un sumador de datos de M x N bits (figura 6). De este modo, cada sumador calcula la suma de una porción de los sumandos. Este es el procedimiento clásico para el diseño y construcción de sumadores serie –un procedimiento que en las metodologías de diseño basadas en el uso de HDLs carece de interés.

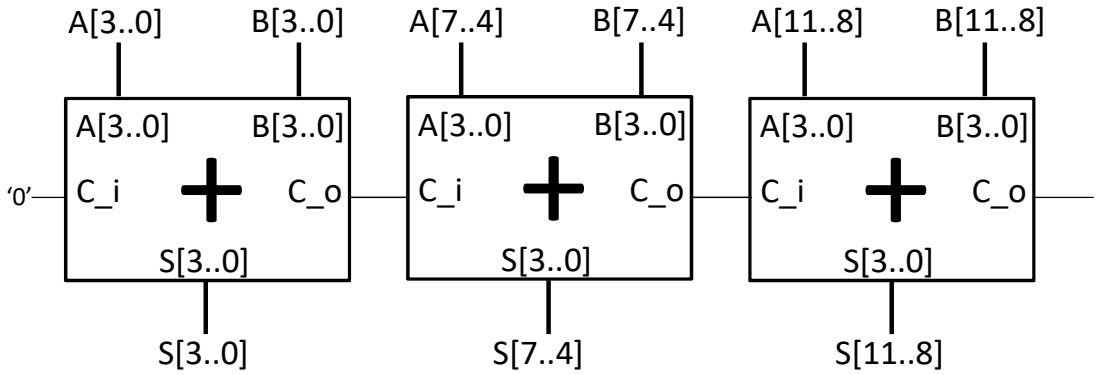


Figura 6.- Sumador de doce bits

La principal utilidad de la entrada de acarreo es la serialización de sumas empleando sumadores que disponen también de salida de acarreo (figura 6).

Salida de detección de desbordamiento

Por último, los sumadores que operan en Complemento a 2 pueden disponer de una salida para detectar el desbordamiento, una salida de *overflow* –como ya sabe, la detección de desbordamiento en binario natural se hace con la salida de acarreo. La lógica de detección de desbordamiento en la suma en complemento a dos es simple, ya que el desbordamiento ocurre cuando el bit de signo de los dos sumandos (A_{MSB} y B_{MSB}) es el mismo y el del resultado (S_{MSB}) no coincide con el de ellos:

$$Ov = \left(\overline{A_{MSB} \oplus B_{MSB}}\right) \cdot \left(A_{MSB} \oplus S_{MSB}\right)$$

Realice las siguientes operaciones:

1. Añada los ficheros *sum8bits.vhd* y *sum8bits_tb.vhd* al proyecto *Proy_BT1_A5*.

2. Abra, con el editor de ModelSim, el fichero *sum8bits.vhd*.

   El fichero contiene el modelo de un sumador de 8 bits con acarreo de entrada. Observe que la función del sumador se modela nuevamente con una sentencia concurrente en la que, ahora, se calcula la suma del acarreo y los dos sumandos de entrada:

```
architecture rtl of sum8bits is
begin
    S <= A + B + C_in;

end rtl;
```

En la sentencia de asignación, los objetos A, B y S son de 8 bits, mientras que C_in es de 1 bit. El hecho de que los objetos que se suman tengan distinto número de bits no ocasiona ningún problema, pero hay una condición que sí debe cumplirse para poder asignar el valor de la suma a un objeto: *el número de bits del objeto al que se asigna el resultado de la suma debe ser igual al que tenga el sumando con mayor número de bits*.

En el modelo del sumador se cumple esta condición porque S es de 8 bits y A y B (los sumandos más *largos*) también son de 8 bits. Pero es, precisamente, esta restricción en el uso de la operación de suma la causante de un pequeño inconveniente a la hora de modelar el funcionamiento del acarreo de salida de los sumadores.

En un sumador binario de N bits hay acarreo de salida cuando la suma no puede representarse con N bits. La idea más simple para modelar el funcionamiento del acarreo de salida consiste en asignar la suma (de los sumandos de N bits) a una señal de N+1 bit, de modo que el bit de mayor peso de la salida sea el acarreo de salida –ya que sólo valdrá 1 cuando el resultado no quepa en N bits–, mientras que el resultado de salida (S) estaría formado por los N bits restantes.

Pero esta solución choca con la restricción para el uso de la suma antes mencionada. Para solventar el problema hay que ajustar, *instrumentalmente*, el tamaño de los sumandos. Si la expresión:

```
S <= A + B;
```

da lugar a un error sintáctico porque S es de N+1 bit, mientras que A y B son de N bits, basta con añadir un 0 a la izquierda a A o B para que se ajuste el número de bits a los requeridos por la operación de suma –normalmente se añade un 0 a cada uno:

```
S <= ('0' & A) + ('0' & B);
```

En las siguientes operaciones se añade una salida de acarreo al modelo del sumador de 8 bits.

3. Añada un puerto de salida, de 1 bit y nombre *C_out*, a la declaración de entidad del sumador de 8 bits:

```
entity sum8bits is
port(
     C_in:     in     std_logic;
     A:        in     std_logic_vector(7 downto 0);
     B:        in     std_logic_vector(7 downto 0);
     S:        buffer std_logic_vector(7 downto 0);
     C_out:    buffer std_logic
    );
end entity;
```

4. Declare una señal auxiliar, *S_aux*, en el cuerpo de arquitectura, con un bit más (9) que los sumandos:

```
architecture rtl of sum8bits is
  signal S_aux: std_logic_vector(8 downto 0);
```

5. Asigne a la señal auxiliar la suma de los sumandos y el acarreo de entrada, añadiendo un 0 a la izquierda a A y B:

```
S_aux <= ('0' & A) + ('0' & B) + C_in;
```

6. Para terminar, empleando dos sentencias concurrentes se asigna: al acarreo de salida el bit de mayor peso de *S_aux* y a la salida de suma, S, los n bits restantes;

```
architecture rtl of sum8bits is
  signal S_aux: std_logic_vector(8 downto 0);

begin
    S_aux <= ('0' & A) + ('0' & B) + C_in;
    C_out <= S_aux(8);
    S <= S_aux(7 downto 0);

end rtl;
```

Es necesario declarar una señal auxiliar en el cuerpo de arquitectura porque a la izquierda de una sentencia asignación no puede haber una asociación de objetos –no se puede asignar la suma directamente a *C_out & S*. Las dos sentencias de asignación que segregan *S_aux* para dar valor a *C_out y S* modelan, simplemente, la identidad existente entre los bits de *S_aux* y las salidas del sumador.

El modelado de la salida de desbordamiento en sumadores que operan en Complemento a 2 se realiza en base a la propiedad que define su ocurrencia: hay desbordamiento cuando el signo de los sumandos es el mismo y diferente del signo del resultado.

Como el bit de signo en los números codificados en complemento a dos es el bit de mayor peso, el siguiente código modelaría el funcionamiento de una salida (*OV*) de detección de desbordamiento en el sumador de 8 bits:

```
process(A(7), B(7), S(7))
begin
  if (A(7) = B(7)) AND (S(7) /= A(7)) then
      OV <= '1';

  else
      OV <= '0';

  end if;
end process;
```

Este proceso contiene una única sentencia IF. Existe un tipo de sentencia concurrente que equivale a este tipo de procesos: la *sentencia concurrente de asignación condicional*. Empleándola, el modelado de la salida de *overflow* quedaría:

```
OV <= '1' when (A(7) = B(7)) and (A(7) /= S(7))
      else '0';
```

7. Añada una salida de *overflow*, *OV*, a la declaración de entidad del sumador de 8 bits y modele su funcionamiento con una sentencia concurrente de asignación condicional.

   El modelo VHDL completo del sumador debe ser el siguiente:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity sum8bits is
port(
     C_in:  in     std_logic;
     A:     in     std_logic_vector(7 downto 0);
     B:     in     std_logic_vector(7 downto 0);
     S:     buffer std_logic_vector(7 downto 0);
     C_out: buffer std_logic;
     OV:    buffer std_logic
    );
end entity;

architecture rtl of sum8bits is
  signal S_aux: std_logic_vector(8 downto 0);

begin
    S_aux <= ('0' & A) + ('0' & B) + C_in;
    C_out <= S_aux(8);
    S <= S_aux(7 downto 0);

    OV <= '1' when (A(7) = B(7)) and (A(7) /= S(7))
          else '0';
end rtl;
```

8. Salve el modelo del sumador de 8 bits y abra, con el editor de ModelSim, el fichero *sum8bits_tb.vhd*.

   El proceso que maneja los estímulos hace uso de un bucle indefinido, *loop*, para probar todas las combinaciones posibles de los sumandos (2^{16}) con y sin acarreo de entrada. La sentencia *exit* permite salir de este tipo de bucles.

```
process
begin
  C_in <= '0';
  A <= (others => '0');
  B <= (others => '0');
  loop
    for i in 0 to 255 loop
      for j in 0 to 255 loop
        wait for 100 ns;
        B <= B + 1;

      end loop;
      A <= A + 1;
    end loop;

    if C_in = '1' then
      exit;
    end if;
    C_in <= not C_in;
```

```
    end loop;
    wait;
end process;
```

9. Repase el código del proceso que maneja los estímulos del *test-bench*, para entender cómo genera los estímulos.

10. Compile los el modelo del sumador de 8 bits y su *test-bench*.

11. Ejecute una simulación (*Run-All*) tras haber añadido al visor de formas de onda todas las señales del *test-bench* y la señal *S_aux* del modelo del sumador.

12. Configure el visor de formas de onda para que traduzca los valores de los buses empleando código binario natural (*unsigned*).

13. Compruebe que:

    a. El modelo suma correctamente con y sin acarreo
    b. Que el acarreo de salida detecta la ocurrencia de desbordamiento
    c. Que la señal *S_aux* representa correctamente el resultado de salida

14. Configure el visor de formas de onda para que traduzca los valores de los buses empleando código Complemento a Dos (*decimal*).

15. Compruebe que:

    a. El modelo suma correctamente con y sin acarreo
    b. Que la salida *OV* detecta la ocurrencia de desbordamiento

    Observe que ahora la salida *S_aux* no representa siempre el valor correcto de la suma.

Diseño lógico de sumadores

Los sumadores son, desde el punto de vista de su realización lógica, sistemas complejos cuya realización no puede abordarse, en general[3], directamente a partir de su tabla de verdad –la tabla de verdad de un sumador de números de 16 bits, por ejemplo, tiene miles de millones de filas (2^{32} filas). Para construirlos no queda más remedio que recurrir al diseño de una estructura interconectada de sistemas combinacionales simples. En la práctica, se puede optar entre tres alternativas:

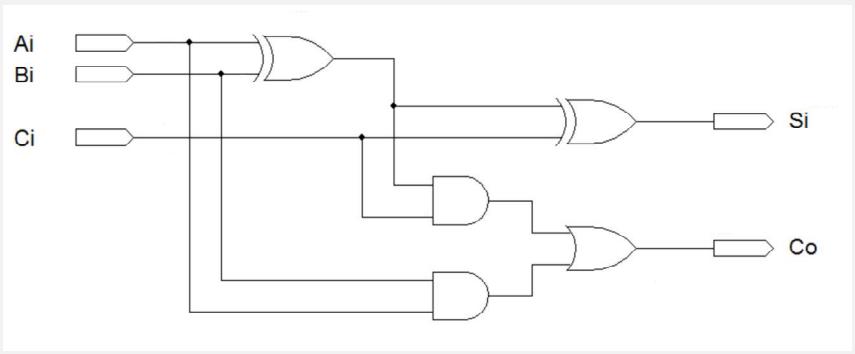
- la realización de un sumador serie
- la realización de un sumador con acarreo anticipado
- la realización de un sumador mixto.

[3] Sólo resulta razonable realizar tablas de verdad de sumadores que operan con datos de 2 ó 3 bits.

Sumadores serie

La forma más simple de diseñar un sumador binario consiste en encadenar en serie sumadores completos de 1 bit. Un sumador completo de un bit es capaz de calcular la suma (S_i) de los bits homólogos de dos sumandos (A_i y B_i), considerando el acarreo de entrada (C_i) generado por los dos bits de peso i-1, y generar el acarreo de salida (C_o) para los bits de peso i+1 de los sumandos. En la figura 7 se muestra su tabla de verdad y su realización con células estándar.

Ai	Bi	Ci	*Si*	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$$Si = Ai \oplus Bi \oplus Ci$$
$$Co = (Ai \oplus Bi) \cdot Ci + Ai \cdot Bi$$

Figura 7.- Sumador completo de 1 bit

Conectando en serie N sumadores completos de 1 bit se construye un sumador completo de N bits.

En la figura 8 se muestra la estructura de un sumador completo de 4 bits.

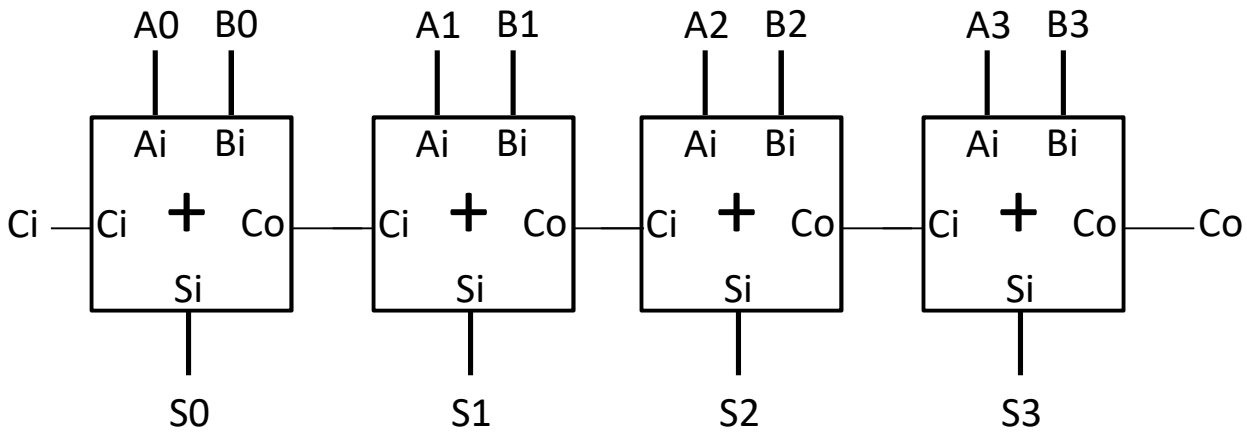


Figura 8.- Sumador completo de 4 bits

Como los sumadores completos de 1 bit pueden realizarse con un circuito lógico tan sencillo como el que se muestra en la figura 7 –con cinco puertas lógicas, o empleando 2 LUTs de al menos 3

entradas, o dos PAL– y la construcción de un sumador serie no requiere ningún otro tipo de circuito, se trata de una solución muy económica en términos de área *hardware*.

En una arquitectura como la de la figura 8, cada sumador completo de 1 bit calcula correctamente un bit del resultado y el acarreo de salida, a partir del momento en que dispone de valores válidos en sus tres entradas. El problema que presenta este sumador es que el tiempo que tardan en establecerse valores válidos y estables del acarreo de entrada (C_i) en cada sumador completo depende del peso de los bits que sume.

Supongamos, por simplicidad, que el tiempo de retardo para las dos salidas de cada sumador completo (Si y Co) de 1 bit es Tp. Tomando como origen de tiempos (T = 0) el instante en que el acarreo de entrada y los sumandos conmutan a un determinado valor:

1. El acarreo de entrada para el sumador que opera con los bits de menor peso está disponible en T= 0 y el bit de suma, S_0, es válido en T = Tp.

2. El acarreo para el sumador que opera con los bits de peso 1 está disponible en T= Tp; por tanto, el bit de suma, S_1, es válido en T = 2 x Tp.

3. El acarreo para el sumador que opera con los bits de peso 2 está disponible en T= 2 x Tp y el bit de suma, S_2, es válido en T = 3 x Tp.

4. En general, para el sumador que opera con los bits de peso M, el acarreo de entrada toma un valor válido y estable en T = M x Tp y el bit de suma, S_M, en T = (M+1) x Tp.

Este análisis muestra que el acarreo se va propagando por los sumadores, de modo que cuanto mayor es el peso de los bits que se operan, más tiempo tarda en calcularse el bit de suma correspondiente. Como el retardo máximo del circuito es el que determina el tiempo que tarda el circuito en calcular el valor de salida, los sumadores con una estructura de conexión en serie son lentos, y tanto más lentos cuanto mayor sea la longitud de los datos que suman.

En conclusión:

- Los sumadores serie son circuitos que tienen la ventaja de que pueden realizarse con un bajo coste en área *hardware* (cinco puertas lógicas o dos LUTs por cada bit del sumador).

- Tienen el inconveniente de que son circuitos lentos, especialmente cuando los sumandos tienen un gran número de bits.

Sumadores con acarreo anticipado

Los sumadores con acarreo anticipado (*Carry-Look-ahead Adders*) son sumadores que consiguen mejorar la velocidad de cálculo –reducir el retardo– respecto a los sumadores serie a costa de añadir lógica para acelerar la generación de los acarreos.

La salida S_i de un sumador completo se obtiene como la or-exclusiva de los bits de peso i de los sumandos (A_i y B_i) y el acarreo generado por los bits de peso i-1 (C_i), tal y como puede observarse en el circuito de la figura 7. La lentitud de cálculo de los sumadores serie viene provocada por el hecho de que los acarreos de entrada se generan por medio de una señal que se propaga a lo largo de

la cadena de sumadores conectados en serie. La idea básica de diseño de los sumadores con acarreo anticipado consiste en romper esa cadena y añadir una lógica que genera simultáneamente, y con un retardo pequeño, el acarreo para todos los sumadores completos (figura 9).

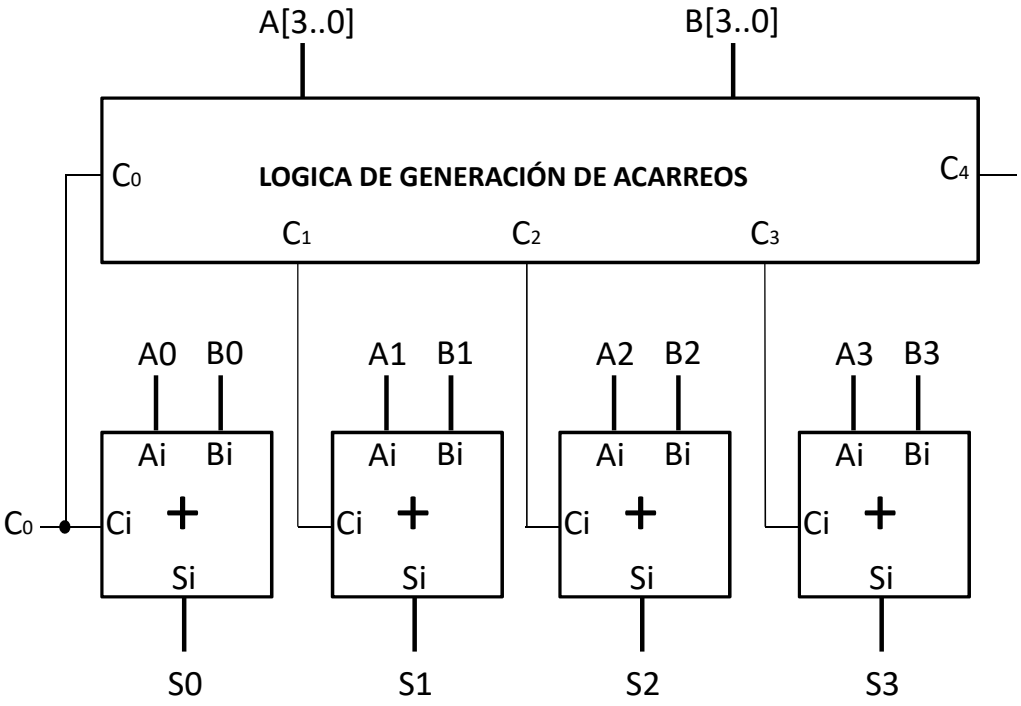


Figura 9.- Sumador con acarreo anticipado

Los sumadores de un bit del esquema de la figura 9 tienen una estructura más simple que los que se emplean para realizar sumadores serie, ya que no tienen que generar el acarreo de salida C_o (figura 7), pero el bloque de lógica que genera los acarreos anticipados es, en cambio, notablemente complejo, y tanto más cuanto mayor es el peso del acarreo que se genera –entiéndase: las ecuaciones del circuito que calcula C_4, por ejemplo, son mucho más complejas que las del que genera C_1.

Ecuaciones del acarreo anticipado

El módulo que calcula los acarreos lo hace a partir de los bits de los sumandos y del acarreo de entrada del sumador (C_0). Puede obtenerse una expresión genérica para la ecuación lógica del acarreo C_i en base al siguiente razonamiento: el acarreo de peso i, C_i, vale 1 si los dos bits homólogos de peso i-1 de los sumandos, A_{i-1} y B_{i-1}, **generan** acarreo –lo que sólo ocurre cuando ambos valen 1– o cuando el acarreo C_{i-1} vale 1 y A_{i-1} y B_{i-1} **propagan** el acarreo –lo que ocurre cuando uno de los dos vale 1 y el otro 0 (cuando son distintos).

Expresando en términos lógicos las condiciones de generación (G_{i-1}) y propagación (P_{i-1}) de acarreo de los bits de peso i-1, A_{i-1} y B_{i-1}:

$$G_{i-1} = A_{i-1} \cdot B_{i-1}$$
$$P_{i-1} = A_{i-1} \oplus B_{i-1}$$

Se obtiene la expresión:

$$C_i = G_{i-1} + P_{i-1} \cdot C_{i-1}$$

Con esta ecuación se puede obtener la función de acarreo para cualquier peso. El acarreo de peso 1, C_1:

$$C_1 = G_0 + P_0 \cdot C_0 = A_o \cdot B_o + (A_o \oplus B_o) \cdot C_o$$

El de peso 2:

$$C_2 = G_1 + P_1 \cdot C_1 = G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0) = A_1 \cdot B_1 + (A_1 \oplus B_1) \cdot (A_o \cdot B_o + (A_o \oplus B_o) \cdot C_o)$$

El de peso 3:

$$C_3 = G_2 + P_2 \cdot C_2 = G_2 + P_2 \cdot (G_1 + P_1 \cdot C_1) = G_2 + P_2 \cdot (G_1 + P_1 \cdot (G_0 + P_0 \cdot C_0))$$
$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$

Interpretando esta última ecuación puede resumirse la lógica de acarreo, en términos proposicionales, a:

> "*Hay acarreo de peso 3 cuando lo generan los bits de peso 2, o lo generan los de peso 1 y lo propagan los de peso 2, o hay acarreo C_0 y lo propagan los de peso 0, 1 y 2*"

Enunciado generalizable para el peso i:

> "*Hay acarreo de peso i cuando lo generan los bits de peso i-1, o lo generan los de peso i-2 y lo propagan los de peso i-1, o lo generan los de peso i-3 y lo propagan los de peso i-2 e i-1, etcétera*"

Este último enunciado proposicional puede expresarse mediante la ecuación lógica:

$$C_i = G_{i-1} + P_{i-1} \cdot G_{i-2} + P_{i-1} \cdot P_{i-2} \cdot G_{i-3} + ... + P_{i-1} \cdot P_{i-2} \cdot .... \cdot P_1 \cdot P_0 C_0$$

Es fácil constatar en esta ecuación que la complejidad de la lógica de generación del acarreo anticipado aumenta con el peso del acarreo. A modo de ejemplo, la ecuación para el acarreo C_6 supondría la suma lógica de 7 términos; el más sencillo de ellos es un producto de dos variables (G_5) y, el más complejo, un producto de 6 or-exclusivas y el acarreo de entrada C_0. Una consecuencia de esta complejidad creciente es que no resulta razonable generar acarreos anticipados de pesos altos.

La ventaja que se obtiene con esta arquitectura es un aumento en la velocidad de cálculo: las ecuaciones de acarreo pueden obtenerse como sumas de productos con 2 ó 3 niveles de lógica y unos retardos muy pequeños. Si el retardo máximo del generador de acarreo es T_G y el de cada sumador de un bit T_S, el retardo total del circuito es $T_G + T_S$, un retardo independiente de la longitud de los datos que se suman.

En resumen:

- Los sumadores con acarreo anticipado tienen la ventaja de que son más rápidos (tienen retardos más pequeños) que los sumadores serie.

- Tienen el inconveniente de que su realización requiere muchos más recursos *hardware* que los necesarios para construir un sumador serie.

Sumadores Mixtos

La complejidad creciente de la función de generación de acarreo anticipado, a medida que aumenta el peso del acarreo, hace que esta solución no pueda aplicarse a sumadores con un gran número de bits; en la práctica suele limitarse la generación de acarreo anticipado hasta el peso 4, tal y como representa, exactamente, el diagrama de bloques de la figura 9. Cuando se desea realizar sumadores rápidos para datos de longitud mayor no queda más remedio que recurrir a una solución híbrida entre la arquitectura serie y la arquitectura con acarreo anticipado, para realizar lo que se conoce como un sumador mixto.

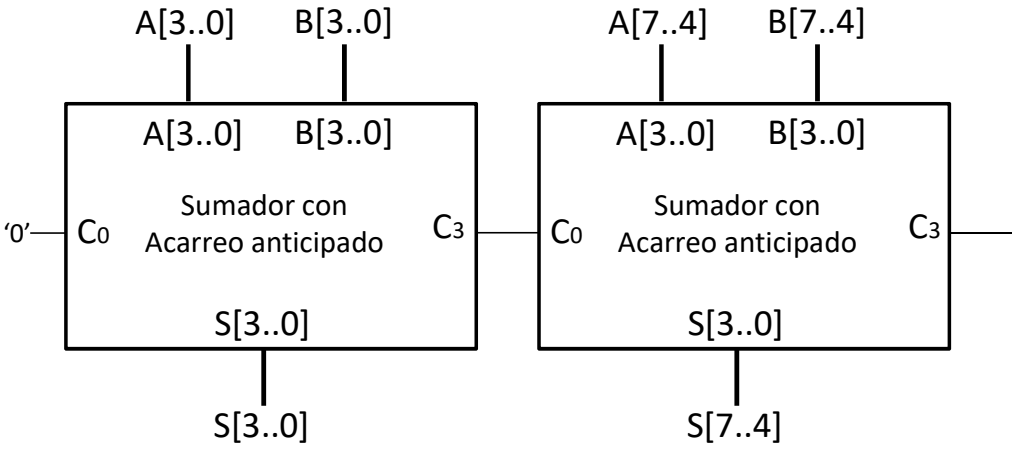


Figura 10.- Sumador Mixto

En un sumador mixto, como el de la figura 10, se conectan en serie sumadores completos de varios bits –típicamente 4– que internamente disponen de un generador de acarreo anticipado. La clave de esta solución es moderar el coste *hardware* de realización, limitando el peso máximo del acarreo generado a la longitud de los módulos que se empleen –si los módulos son de 4 bits la función más compleja que se materializa es C_4– a costa de cierta penalización en el retardo derivada de la conexión en serie de los módulos.

Observando la figura 10, puede comprobarse que si el acarreo de salida, C_3, del primer módulo es generado anticipadamente con un retardo pequeño, T_G, el retardo total del sumador de 8 bits, T_M, sería la suma de T_G y el retardo propio del segundo módulo, T: $T_M = T_G + T$

Si se añadiera en serie un tercer módulo para construir un sumador de 12 bits, el retardo total sería:

$$T_M = 2 \times T_G + T$$

Con valores de T_G pequeños pueden realizarse sumadores relativamente rápidos y de un considerable número de bits, con un sobrecoste *hardware* moderado respecto a la versión serie que es la opción de realización con menor coste en área *hardware*.

En definitiva:

- Los sumadores mixtos son considerablemente más rápidos que los sumadores serie, pero su realización requiere un mayor número de recursos *hardware*.

- Son la única opción viable para la realización de sumadores combinacionales rápidos para datos de tamaño medio y grande.

Área *versus* Velocidad

Las características de las arquitecturas para la realización de sumadores combinacionales ponen claramente de manifiesto una propiedad que debe tener siempre presente un diseñador de sistemas digitales cableados complejos: *"La mejora de la capacidad de cálculo de un sistema digital suele traer consigo un aumento en el número de recursos hardware necesarios para realizarlo –un aumento del área del circuito-, y viceversa, una optimización de los recursos empleados para realizar un sistema suele acarrear una disminución de la velocidad de procesamiento"*.

Es muy infrecuente encontrar casos donde esta regla se incumpla. Es necesario tenerla siempre en cuenta a la hora de valorar los objetivos de un diseño; cuando se conjuntan exigencias de velocidad y economía de recursos, la dificultad del diseño es máxima. Para comparar soluciones alternativas a una misma funcionalidad, con distintas velocidades y área *hardware*, suele utilizarse como factor de mérito el producto de ambos factores para cada una de las soluciones. Los resultados de este tipo de *evaluaciones* dependen de la tecnología de realización y su representación gráfica permite comparar las diferentes soluciones. La figura 11 muestra una valoración de las tres arquitecturas de sumadores combinacionales en función de los recursos consumidos (área) y la velocidad (tiempo de retardo).

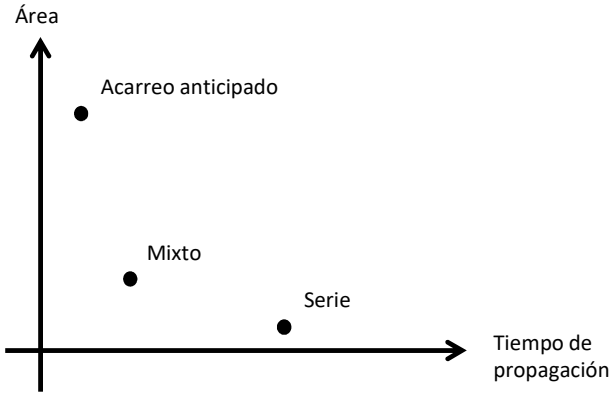


Figura 11.- Área vs. Velocidad

Suma con constantes

Cuando uno de los sumandos es constante, la complejidad lógica de los sumadores puede reducirse considerablemente. En la figura 12 se muestra el grado de simplificación lógica que se alcanza en un sumador completo de 1 bit cuando uno de los sumandos vale 0 ó 1.

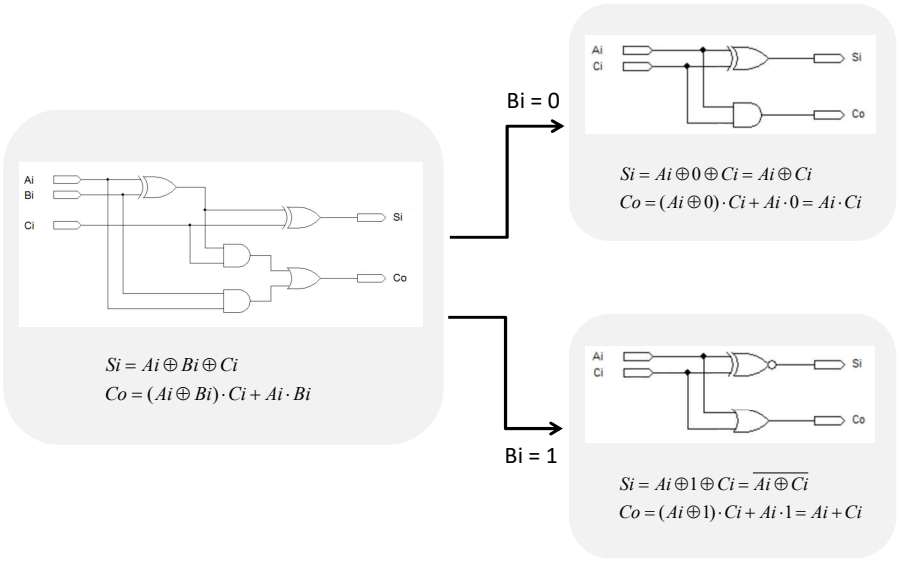


Figura 12.- Sumador completo simplificado

Pero la reducción del área del circuito y del retardo de propagación depende de la tecnología de realización:

- Con células estándar, el hecho de que uno de los sumandos sea constante supone una reducción cercana al 60% del área del circuito –se ahorran tres de cinco puertas por bit del sumador–, en cambio, en una realización con chips configurables no se consigue prácticamente ningún ahorro, pues se sigue necesitando emplear dos LUTs (o dos PAL) –con una entrada menos que en un sumador completo, pero cualquier LUT de un chip configurable tiene, al menos, cuatro entradas (y cualquier PAL decenas).

- El retardo en una realización con células estándar disminuye también notablemente, ya que un sumador completo tiene hasta tres niveles de puertas entre las entradas y la salida de acarreo, mientras que las versiones simplificadas sólo tienen una puerta entre cualquier entrada y salida. En una realización con células configurables no se consigue ningún aumento apreciable de velocidad, ya que la estructura del circuito es la misma que en el caso del sumador completo: una LUT (o PAL) para generar el bit de la suma y otra para generar el de acarreo.

Puede realizarse un análisis análogo al anterior para las células que se utilizan para construir sumadores con constantes con acarreo anticipado. En este caso, las células sumadoras son como las de la figura 12 pero sin la lógica de generación de acarreo, es decir, quedan reducidas a una puerta XOR o XNOR. Las funciones de generación de acarreo anticipado también se simplifican, ya que, si en el sumando de valor constante el bit de peso i vale 0:

$$G_i = A_i \cdot B_i = A_i \cdot 0 = 0$$
$$P_i = A_i \oplus B_i = A_i \oplus 0 = A_i$$

Y si vale 1:

$$G_i = A_i \cdot B_i = A_i \cdot 1 = A_i$$
$$P_i = A_i \oplus B_i = A_i \oplus 1 = \overline{A_i}$$

Así, por ejemplo, para el acarreo de peso 3, C_3, la expresión general es:

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0$$
$$C_3 = A_2 \cdot B_2 + (A_2 \oplus B_2) \cdot (A_1 \cdot B_1 + (A_1 \oplus B_1) \cdot (A_0 \cdot B_0 + (A_1 \oplus B_1) \cdot C_0))$$

Pero esta compleja expresión se reduce sustancialmente cuando B es constante; si vale, por ejemplo, 1010:

$$C_3 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_0 = 0 + A_2 \cdot A_1 + A_2 \cdot \overline{A_1} \cdot 0 + A_2 \cdot \overline{A_1} \cdot A_0 \cdot C_0$$
$$C_3 = A_2 \cdot A_1 + A_2 \cdot \overline{A_1} \cdot A_0 \cdot C_0 = A_2 \cdot (A_1 + A_0 \cdot C_0)$$

Esta reducción de complejidad se traslada a un menor uso de recursos y del retardo de generación de los acarreos en cualquier tecnología, si bien el grado de simplificación depende del valor concreto que tome la constante y el acarreo de entrada C_0 –normalmente los sumadores en que uno de los sumandos es constante tienen el acarreo de entrada puesto a 0.

Incrementadores

Un incrementador es un sumador en el que uno de los sumandos vale 1. Un incrementador de N bits se realiza, con una arquitectura serie, empleando para el cálculo del bit de menor peso un inversor[4] y, para los restantes N-1 bits, la simplificación del sumador completo cuando un bit de entrada vale 0. En la figura 13 se muestra la realización serie de un incrementador de 4 bits.

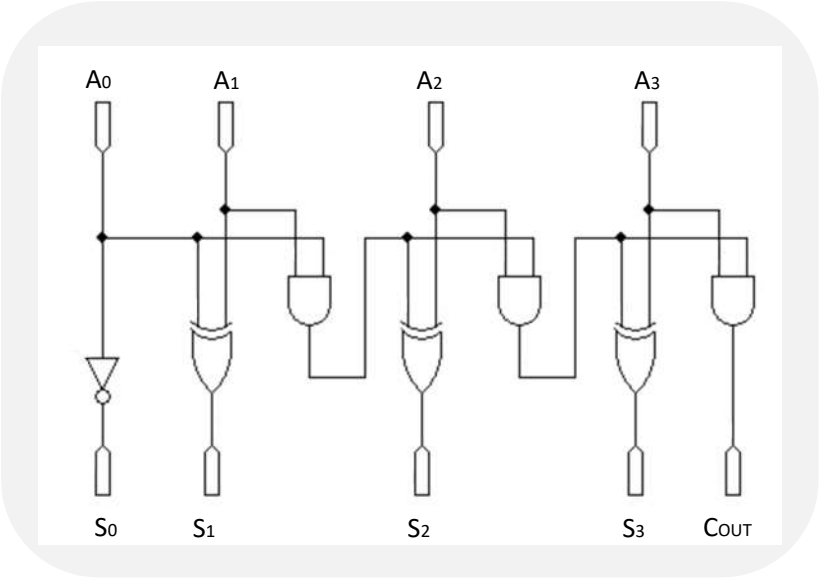


Figura 13.- Incrementador serie de 4 bits

En la figura 14 se muestra la versión con acarreo anticipado. La expresión genérica del acarreo de peso i, C_i, es:

$$C_i = A_{i-1} \cdot A_{i-2} \cdot ....A_2 \cdot A_1 \cdot A_0$$

[4] El sumador de un bit en el que un sumando vale 1 queda reducido a un simple inversor cuando el acarreo de entrada es 0.

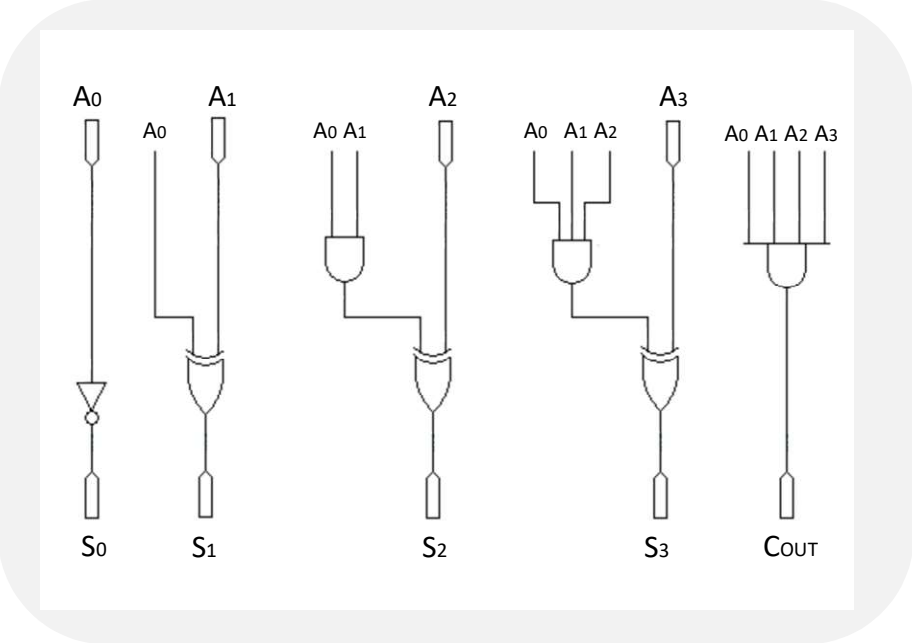


Figura 14.- Incrementador de 4 bits con acarreo anticipado

La comparación entre el modo en que se calculan los bits de acarreo en los circuitos de la figura 13 y 14 permite apreciar fácilmente, debido a la simplicidad de ambos circuitos, la relación entre la estructura y las propiedades características de los sumadores serie y con acarreo anticipado:

- En el circuito de la figura 13 el acarreo de salida, C_3, se genera mediante la conexión serie de tres puertas AND de dos entradas que, por otra parte participan en la generación de otros dos acarreos (C_1 y C_2). Como consecuencia se obtiene una solución con poco consumo de recursos, pero lenta.

- En el circuito de la figura 14, el acarreo se genera con un único nivel de puertas AND, lo que da lugar a un circuito veloz, pero, a cambio, cada puerta sólo participa en la generación de un acarreo y el circuito requiere más área hardware, ya que emplea puertas AND de dos, tres y cuatro entradas.

Modelado de sumadores con constantes

El siguiente código modela el funcionamiento de un subsistema que suma 7 a un dato de entrada de 10 bits:

```
architecture rtl of inc_7 is
begin
    S <= A + 7;

end rtl;
```

Observe que el valor del sumando constante se expresa como un número entero; también puede codificarse como una ristra de bits:

```
S <= A + "111";
```

O también expresando el valor constante con el número de bits que tenga A; si es 10:

```
S <= A + "0000000111";
```

Realice las siguientes operaciones:

1. Añada los ficheros *inc_3.vhd* e *inc_3_tb.vhd* al proyecto *Proy_BT1_A5*.

2. Abra, con el editor de ModelSim, el fichero *inc_3.vhd*, que contiene el modelo de un sistema que suma 3 a un dato de 5 bits.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity inc3 is
port(A: in     std_logic_vector(4 downto 0);
     S: buffer std_logic_vector(4 downto 0));
end entity;

architecture rtl of inc_3 is
begin
    S <= A + 3;

end rtl;
```

3. Revise el modelo y abra el fichero que contiene el *test-bench*

   El *test-bench* está incompleto, pues no se definen valores para A en el proceso de manejo de estímulos.

```
architecture test of inc_3_tb is
  signal A: std_logic_vector(4 downto 0);
  signal S: std_logic_vector(4 downto 0);

begin
  dut: entity work.inc_3(rtl)
  port map(
           A => A,
           S => S);
  process
  begin
    A <= (others => '0');
    -- Completar con bucle FOR

  end process;
end test;
```

4. Complete el código para generar todas las posibles combinaciones de A empleando un bucle FOR, compile el modelo del sumador y el *test-bench*, y realice una simulación para comprobar el funcionamiento del modelo.

PARTE III. Restadores

La resta de dos números N_1 y N_2, $R = N_1 - N_2$, puede reducirse al cálculo de una suma, $R = N_1 + (-N_2)$, si $-N_2$ se expresa como el complemento a la base de N_2.

El complemento a la base, B, de un número, N, es:

$$[N]_B = B^m - N$$

Donde m es el número de dígitos con que se expresa N en dicha base.

Si expresamos el valor –N como el complemento a la base de N, $[N]_B$, la resta de N_1 y N_2 puede calcularse:

$$N_1 - N_2 = N_1 + (-N_2) = N_1 + [N_2]_B = N_1 + (B^m - N_2) = B^m + (N_1 - N_2) \quad (1)$$

Si $N_1 > N_2$, el resultado de la resta es positivo y vale $N_1 - N_2$, que son los **m** dígitos de menor peso de $B^m + (N_1 - N_2)$.

Por ejemplo:

- El complemento a 10 de 7, $[7]_{10}$, es 3, puesto que m = 1: 10 – 7 = 3

  9 – 7 se puede calcular como 9 + 3 = **1**2
  Cogiendo los m (1) dígitos menos significativos, se obtiene el resultado correcto: 2.

- El complemento a 10 de 48, $[48]_{10}$, es 100 – 48 = 52

  75 – 48 se puede calcular como 75 + 52 = **1**27
  Cogiendo los m (2) dígitos menos significativos, se obtiene el resultado correcto: 27.

Como puede observarse en ambos ejemplos, cuando el resultado de la resta es positivo se genera un **acarreo** que debe ignorarse para extraer el resultado correcto.

Si $N_1 < N_2$, el resultado de la resta es negativo y debe valer $-(N_2 - N_1)$, pero como los números negativos se expresan como el complemento a la base de su módulo, el resultado que debe obtenerse es el complemento a la base de $N_2 - N_1$:

$$[N_2 - N_1]_B = B^m - (N_2 - N_1)$$

Y efectivamente, manipulando la ecuación (1):

$$B^m + (N_1 - N_2) = \quad B^m - (N_2 - N_1)$$

Por ejemplo:

- El complemento a 10 de 84, $[84]_{10}$, es 16, puesto que m = 2: 100 – 84 = 16

  65 – 84 = -19, se puede calcular: 65 + 16 = 81

  Como $[19]_{10}$ = 81, el resultado es correcto

- El complemento a 10 de 4.100, $[4.100]_{10}$, es 10.000 – 4.100 = 5.900

  1.200 – 4.100 = -2.900, se puede calcular como: 1.200 + 5.900 = **7100**

  El resultado de la resta es correcto puesto que $[2.900]_{10}$ = 7.100

Observe que en este caso no se genera acarreo

En resumen:

1. La resta de dos números X e Y puede realizarse sumando a X el complemento a la base de Y.

   $$X - Y = X + [Y]_B$$

2. El resultado de la resta es positivo si al realizar la resta se genera acarreo; se extrae cogiendo todas las cifras del resultado excepto el acarreo.

   Si al restar dos números decimales de 3 cifras se obtiene como resultado **1**412, la resta es positiva y de valor +412

3. El resultado de la resta es negativo si no hay acarreo; en este caso el módulo del resultado se expresa en forma de complemento a la base

   Si al restar dos números decimales de 3 cifras se obtiene como resultado 84, la resta es negativa y el módulo del resultado es el número de tres cifras cuyo complemento a la base es 84. Para averiguar cuál es este número basta con calcular el complemento a la base de 84 con tres cifras:

   1.000 – 84 = 916

   El resultado de la resta es -916.

<u>Complemento a 2 de un número</u>

El complemento a 2 de un número binario N, $[N]_2$, de m bits vale 2^m – N. Y puede generarse fácilmente sin necesidad de realizar la resta; basta con complementar todos los bits de N y sumar uno al número resultante.

- Por ejemplo, el complemento a 2 de 10110 (22_D) es 2^5 – 22, en binario 01010

  Puede obtenerse invirtiendo los bits, NOT(10110) = 01001, y sumando 1, 01001 + 1 = 01010

Diseño de restadores

El diseño de los restadores binarios se basa en este procedimiento para el cálculo del complemento a 2, Consiste en emplear un sumador en el que uno de los sumandos es el minuendo y el otro el complemento a 2 del sustraendo (figura 15a).

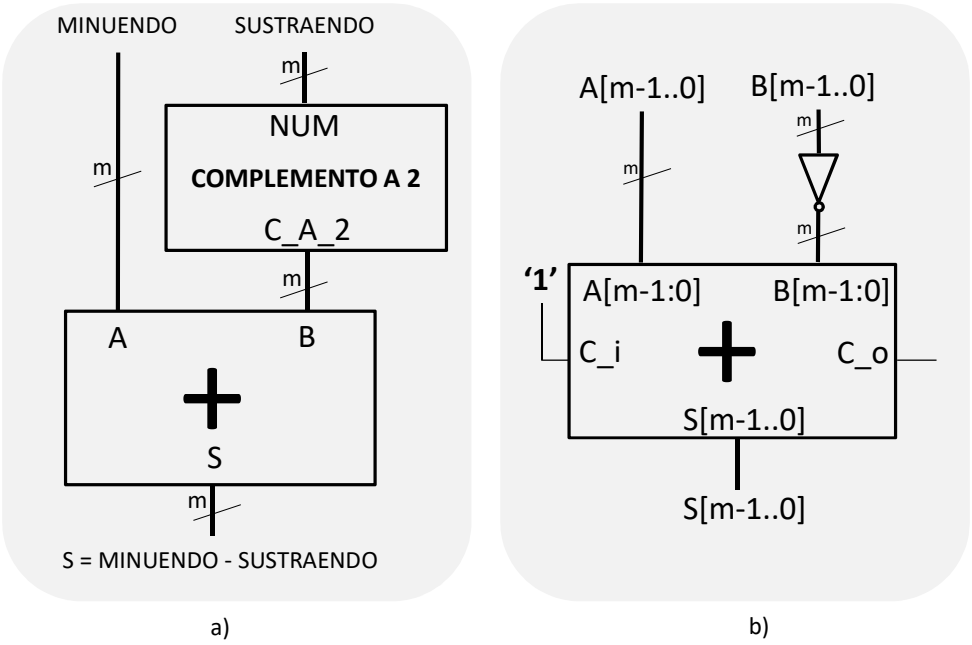


Figura 12.- Restador

La figura 15b representa la realización detallada del restador. El complemento a dos del minuendo se genera utilizando *m* inversores –tantos como bits tengan los datos que se restan– y poniendo a nivel alto el acarreo de entrada del sumador. La operación que realiza el circuito es, por tanto:

$$S = A + (NOT\ (B) + 1) = A + [B]_2 = A - B$$

Para la realización del sumador puede utilizarse cualquiera de las tres arquitecturas que se estudiaron en el apartado 1 de este texto; las características del restador (velocidad y área *hardware*) corresponderán con las del tipo de sumador elegido –y por ello no se va a repetir aquí la exposición hecha en la PARTE I.

Es interesante señalar algunos detalles del funcionamiento del circuito de la figura 12b.

1. Si los datos con que opera el circuito (entradas y salidas) están codificados en binario natural:

    a. El acarreo de salida se activa cuando la resta es positiva ó vale 0. Es, por tanto, una salida que indica cuando el minuendo es mayor o igual que el sustraendo.

    b. Cuando el sustraendo es mayor que el minuendo, el resultado correcto de la resta es negativo. En este caso se produce desbordamiento, porque en binario natural no pueden representarse números negativos, y el acarreo de salida vale 0. El resultado que se obtiene puede interpretarse como el complemento a 2 del módulo de la resta.

c. A la salida de acarreo de los restadores se la suele denominar, en textos técnicos y hojas de datos, salida de *borrow*. Equivale al acarreo de salida de la suma, pero es activa a nivel bajo si se utiliza como *flag de overflow* -cuando vale 0 indica que se ha producido desbordamiento en la resta.

2. Si los datos con que opera el circuito están codificados en complemento a 2:

   a. El *borrow* del restador no detecta la ocurrencia de desbordamiento.

   b. Para detectar el desbordamiento hay que añadir lógica que detecte los casos en que, teniendo el minuendo y sustraendo diferente signo, el signo del resultado es distinto del signo del minuendo.

Resta con constantes

Cuando el minuendo o el sustraendo tienen un valor constante, las consecuencias que se derivan son las mismas que en el caso de los sumadores: se simplifica la estructura lógica de las células y, dependiendo de la tecnología de realización que se emplee, se pueden conseguir, o no, mejoras sustanciales en las prestaciones y recursos empleados para realizar el circuito.

Realizaciones alternativas

El diseño de un restador puede afrontarse también en base a la realización de una célula básica, un restador completo de 1 bit, capaz de calcular la resta de bits homólogos del minuendo y sustraendo, considerando un *borrow* de entrada y generando un *borrow* de salida. Pero, de este modo, la célula básica es la de un sumador completo con una de sus entradas complementada, y equivale a las que formarían el circuito de la figura 12b si se integran los inversores del minuendo en las células del sumador completo y, precisamente por esto, es un procedimiento de diseño que no se utiliza.

Modelado VHDL de restadores

El siguiente código corresponde al modelo de un restador de 4 bits.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rest4bits is
port(
       A: in     std_logic_vector(3 downto 0);
       B: in     std_logic_vector(3 downto 0);
       S: buffer std_logic_vector(3 downto 0)
    );
end entity;

architecture rtl of rest4bits is
begin
    S <= A + (not B) + 1;

end rtl;
```

Observe que la resta se realiza sumando a A el complemento a 2 de B –y esto es independiente del hecho de que el código empleado para codificar los números sea Binario Natural o Complemento a 2. La operación de inversión aplicada al vector B (*not B*) está definida en el paquete *std_logic_1164*.

Realice las siguientes operaciones:

1. Cree un nuevo fichero, de nombre *rest4b.vhd*, en el proyecto *Proy_BT1_A5*.

2. Escriba en el fichero el código del restador de 4 bits listado en la página anterior; para que pueda realizar esta tarea rápidamente, copie al fichero el modelo del sumador de 4 bits, *sum4bits*, cambie el nombre de la entidad –no olvide cambiar también la referencia al nombre en la arquitectura– y modifique la sentencia de asignación concurrente.

3. Salve y compile el modelo.

   Para simular el modelo puede utilizar el *test-bench* que empleó para probar el modelo del sumador de cuatro bits.

4. Abra con el editor de ModelSim el fichero *sum4bits_tb.vhd* y sálvelo con el nombre *rest4bits_tb.vhd*.

5. Edite el fichero para cambiar el nombre del *test-bench* –llámelo *rest4bits_tb* y no olvide cambiar también la referencia a este nombre en el cuerpo de arquitectura- y el nombre del modelo emplazado en el cuerpo de arquitectura –debe ser *rest4bits* en lugar de *sum4bits*.

6. Salve el *test-bench* y compílelo

7. Realice una simulación para comprobar que:

   a. Las restas son correctas en binario natural si el resultado es positivo (si no hay desbordamiento).

   b. Las restas son correctas en complemento a dos si el resultado puede expresarse con 4 bits (si no hay desbordamiento).

El funcionamiento del restador también puede modelarse empleando, directamente, la operación de resta que está definida en el paquete *std_logic_unsigned*. En el modelo del restador sólo habría que modificar la sentencia de asignación concurrente, que quedaría reducida a:

```
S <= A - B;
```

Realice las siguientes operaciones:

1. Modifique el modelo del restador para definir la funcionalidad con la operación de resta, recompile el modelo, y ejecute una simulación para comprobar su correcto funcionamiento.

Modelado de restadores con entradas y salidas de acarreo y detección de desbordamiento

Los restadores, al igual que los sumadores, pueden tener entrada y salida de acarreo (*borrow*) y salida de *overflow*. Estas entradas y salidas tienen exactamente la misma función y aplicación que en los sumadores y se modelan de modo análogo en VHDL.

Realice las siguientes operaciones:

1. Añada al proyecto *Proy_BT1_A5* los ficheros *rest6bits.vhd* y *rest6bits_tb.vhd*.

2. Abra, con el editor de ModelSim, el fichero *rest6bits.vhd*.

   El fichero contiene el modelo de un restador de 6 bits con entrada y salida de acarreo (*B_in* y *B_out*) y salida de detección de desbordamiento (*OV*).

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rest6bits is
port(
     B_in:  in     std_logic;
     A:     in     std_logic_vector(5 downto 0);
     B:     in     std_logic_vector(5 downto 0);
     R:     buffer std_logic_vector(5 downto 0);
     B_out: buffer std_logic;
     OV:    buffer std_logic
    );
end entity;

architecture rtl of rest6bits is
  signal R_aux: std_logic_vector(6 downto 0);

begin
    R_aux <= ('0' & A) + ('0' & (not B)) + B_in;
    B_out <= R_aux(6);
    R <= R_aux(5 downto 0);

    OV <= '1' when (A(5) /= B(5)) and (A(5) /= R(5))
          else '0';
end rtl;
```

   Los restadores con entrada de acarreo realizan la suma del minuendo, el valor complementario del sustraendo y el acarreo (*borrow*) de entrada, de modo que:

   - Si el acarreo de entrada vale 1 –se dice, entonces, que "*no hay borrow*"– se resta el sustraendo del minuendo (A – B).

   - Si el acarreo de entrada vale 0 –si hay *borrow*– se resta al minuendo el sustraendo más uno (A - (B + 1)).

   Por tanto, la sentencia concurrente:

```
R_aux <= ('0' & A) + ('0' & (not B)) + B_in;
```

podría sustituirse por:

```
R_aux <= ('0' & A) - ('0' & B) -( not B_in);
```

Observe, también, que para modelar el acarreo (*borrow*) de salida y la detección de desbordamiento se utiliza un código similar al utilizado en los sumadores –únicamente cambia la condición asociada a la detección de desbordamiento, ya que en los restadores sólo puede ocurrir cuando el minuendo y el sustraendo tienen diferente signo.

3. Revise el código del *test-bench* del restador, que tiene una estructura similar a la del test del modelo del sumador de 8 bits, y, a continuación, compile los ficheros *rest6bits.vhd* y *rest6bits_tb.vhd*.

4. Ejecute una simulación (*Run-All*) tras haber añadido al visor de formas de onda todas las señales del *test-bench*.

5. Configure el visor de formas de onda para que traduzca los valores de los buses empleando código binario natural (*unsigned*).

6. Compruebe que:

   a. El modelo realiza la operación A – B cuando el acarreo de entrada vale 1 (cuando no hay *borrow*) y la operación A – (B + 1) en caso contrario.
   b. Que la salida de acarreo detecta con un 0 la ocurrencia de desbordamiento –cuando hay *borrow* de salida se produce desbordamiento.

7. Configure el visor de formas de onda para que traduzca los valores de los buses empleando código complemento a dos (*decimal*).

8. Compruebe que:

   a. El modelo resta A y B cuando no hay *borrow*, o A y B + 1 en caso de que lo haya.
   b. Que la salida *OV* detecta la ocurrencia de desbordamiento.

Modelado de restadores con una entrada constante

El modelado de restadores en que el minuendo o el sustraendo son constantes resulta muy sencillo. La siguiente sentencia modela el funcionamiento de un subsistema que resta 4 a un dato de entrada de 5 bits:

```
S <= A - 4;
```

Observe que el valor del minuendo se expresa como un número entero; si se prefiere, puede codificarse como una ristra de bits:

```
S <= A - "100";
```

O también puede expresarse como la suma de A y el complemento a 2 de 4:

```
S <= A + "11100";
```

PARTE IV. Multiplicadores y divisores

Para multiplicar o dividir números codificados en Binario Natural o Complemento a 2 es necesario diseñar un sistema digital complejo empleando sumadores y otros subsistemas digitales, combinacionales y secuenciales, dependiendo del tipo de arquitectura de realización que se elija. El estudio de este tipo de subsistemas está fuera del alcance de este texto introductorio al diseño de subsistemas aritméticos.

Pero existen casos particulares –interesantes además por su mucha utilidad– en los que el diseño de la lógica de multiplicación o división se simplifica considerablemente. En concreto:

- Cuando se multiplica un número por una potencia de dos

  La multiplicación de un número (codificado en binario natural o en complemento a 2) por una potencia de dos, 2^m, se realiza añadiendo *m* ceros a la derecha del número, por ejemplo:

  - $100\ 1010_{BN}$ por $2^3 = 10\ 0101\ 0000_{BN}$ (74 x 8 = 592)
  - $11\ 1101_{C2}$ por $2^4 = 11\ 1101\ 0000_{C2}$ (-3 x 16 = -48)

  En general, al multiplicar un número de *n* bits por 2^m se necesitan $n+m$ bits para representar el resultado y, por tanto, esta operación puede dar lugar a la ocurrencia de desbordamiento cuando no se puede emplear este número de bits para representar el resultado.

  - Por ejemplo, si se desea multiplicar el número 6, codificado con 4 bits y en binario natural, por 4 (2^2) y el resultado (6 x 4 = 24) tiene que expresarse también con 4 bits, se produce desbordamiento (0110 x 100 = **01** 1000) porque se necesitan al menos 5 bits para representar el número 24 en binario natural.

- Cuando se divide un número por una potencia de dos

  La división de un número por una potencia de dos, 2^m, se realiza eliminando los *m* bits menos significativos del número. Con este procedimiento se obtiene el cociente de la división; el resto de la división es el valor de los bits eliminados –interpretados en binario natural, tanto si el dividendo está codificado así, como si lo está en complemento a 2. Por ejemplo:

  - $100\ 1010_{BN}$ dividido por $2^3 = 1001_{BN}$ (74/8 = 9); el resto es 010_{BN} (2)
  - $11\ 1101_{C2}$ por $2^4 = 11_{C2}$ (-3/16 = -1); el resto es 1101_{BN} (13)

  Si se desea expresar el resultado de la división con el mismo número de bits que el dividendo, hay que añadir *m* ceros a la izquierda si se utiliza código binario natural, o *m* bits de signo a la izquierda si se emplea codificación en complemento a 2. Por ejemplo

  - $100\ 1010_{BN}$ dividido por $2^3 =$ **000** 1001_{BN}
  - $11\ 1101_{C2}$ por $2^4 =$ **11 11**11_{C2}

- Cuando se multiplica un número por una constante que no es potencia de 2

  Si la codificación de la constante en binario natural tiene *pocos unos*, el multiplicador puede realizarse con una combinación simple de sumadores. El procedimiento que hay que seguir para diseñarlo consiste en descomponer la constante en una suma de potencias de dos y

realizar un sumador con tantos sumandos de entrada como potencias tenga la descomposición realizada. Por ejemplo, si se desea realizar un multiplicador por 10:

- 10_D (1010_{BN}) es igual a 8 (1000_{BN}) + 2 (0010_{BN})
- N x 10_D es igual a N x (8 + 2) = N x 8 + N x 2 = N & "000" + N & '0'

La complejidad de este tipo de multiplicadores depende del número de unos que tenga el código de la constante: un multiplicador por 65 (100 0001), por ejemplo, es mucho más sencillo que otro que multiplica por 43 (010 1011).

El siguiente código VHDL, por ejemplo, modela un sistema que convierte un número BCD de tres dígitos (centenas, decenas y unidades) a binario natural y lo divide por 8.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity BCDtoBN is
port(
     unidades:     in     std_logic_vector(3 downto 0);
     decenas:      in     std_logic_vector(3 downto 0);
     centenas:     in     std_logic_vector(3 downto 0);
     num_bn_div8: buffer std_logic_vector(6 downto 0)
    );
end entity;

architecture rtl of BCDtoBN is
  signal S1_aux: std_logic_vector(6 downto 0);
  signal S2_aux: std_logic_vector(9 downto 0);
begin
    -- S1_aux <= 10*centenas + decenas
    S1_aux <= (centenas & "000") + (centenas &  '0') + decenas;

    -- S2_aux <= 10*(10 centenas + decenas) + unidades
    S2_aux <= (S1_aux & "000") + (S1_aux &  '0') + unidades;

    -- num_bn_div8 <= (100*centenas + 10*decenas + unidades)/8
    num_bn_div8 <= S2_aux(9 downto 3);

end rtl;
```

Para pasar un dato codificado en BCD a binario natural hay que multiplicar cada dígito BCD por la potencia correspondiente a su peso: las unidades por uno, las decenas por diez, las centenas por cien, etcétera. En el modelo VHDL la conversión se hace en dos pasos: la primera sentencia concurrente –la que asigna valor a S1_aux– multiplica por diez las centenas y le suma las decenas; el resultado de esta operación se multiplica otra vez por diez y se le suman las unidades –esto se hace en la sentencia concurrente que asigna valor a S2_aux-, la combinación de ambas operaciones permite obtener un número de diez bits en binario natural de valor:

100 x centenas + 10 x decenas + unidades

Para dividirlo por 8 se desplaza tres posiciones a la derecha, asignando a la salida los 7 bits de mayor peso de S2_aux.

El código del conversor de BCD a binario natural se encuentra en el fichero *BCDtoBN.vhd*.

Realice las siguientes operaciones:

1. Descomprima el fichero *BT1_P5.zip*. Arranque la herramienta *Quartus Prime* y cree, en la carpeta *quartus* de la actividad, un nuevo proyecto, denominado BCDtoBN y añádale el fichero de diseño *BCDtoBN.vhd*.

2. Seleccione culquier dispositivo de la familia MAX-10.

3. Realice *Analysis and Elaboration* y cuando termine abra el visor de la *netlist* RTL. Como puede ver (figura 13) el circuito se ha sintetizado con 4 sumadores.

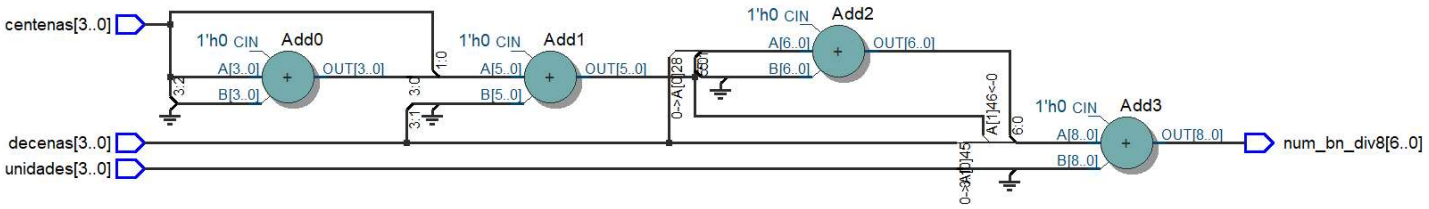


Figura 13.- Síntesis del convertidor de BCD a binario natural

PARTE V. Comparadores

Los comparadores son subsistemas capaces de calcular si un número es mayor, menor o igual que otro. Un comparador puede tener dos entradas, cuando compara el valor de dos datos, o una única entrada si compara el valor de un dato con una constante. Normalmente tendrá una salida de comparación, que podrá ser:

- "A *Igual a B*": Esta salida se activa cuando los números que se comparan tienen el mismo valor.

  Cuando los números que se comparan se codifican con el mismo código –que es la situación que se da en la práctica–, la función de comparación de magnitud coincide con la función de *equivalencia*, es decir, con la detección de que dos palabras de código son iguales.

- *"A distinto de B"*: Se puede obtener negando la función *"A igual a B*".

- *"A mayor que B"*: Esta salida se activa cuando el valor de A es mayor que el de B.

- *"A menor o igual que B"*: Es la función complementaria de *"A mayor que B"*.

- *"A menor que B"*: Esta salida se activa cuando el valor de A es menor que el de B.

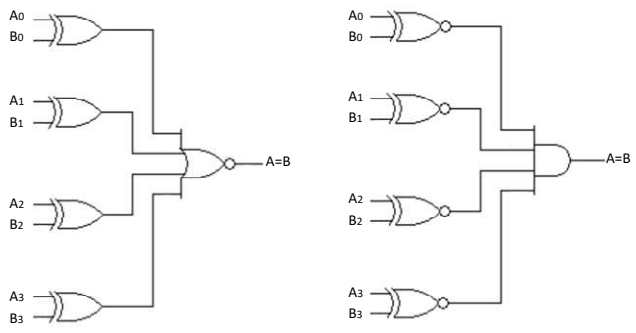
- *"A mayor o igual que B"*: Es la función complementaria de *"A menor que B"*.

Diseño lógico y modelado HDL de comparadores de igualdad

Los comparadores con una estructura más simple son los que detectan si dos números son iguales o distintos. Cuando dos bits, A_i y B_i, son iguales, su *or-exclusiva* vale 0. Por tanto, si dos datos de *n* bits son iguales, la *xor* de todos los bits homólogos –con el mismo peso– de ambos datos debe ser 0:

$$Si\ A = B \Rightarrow \forall i,\ A_i \oplus B_i = 0 \Rightarrow (A_0 \oplus B_0) + (A_1 \oplus B_1) + (A_2 \oplus B_2) + \ldots + (A_{n-1} \oplus B_{n-1}) = 0$$

Esta condición define el diseño de este tipo de comparadores: basta con hacer la *xor* de los bits homólogos de los dos datos y hacer la suma lógica negada de todas las *xor* para detectar, con una salida activa a nivel alto, que los datos son iguales (figura 14a). La figura 14b muestra una realización alternativa con puertas *xnor*.



a) b)

Figura 14.- Comparadores de Igualdad

El funcionamiento de un comparador de igualdad puede modelarse en VHDL empleando una sentencia concurrente de asignación condicional; por ejemplo, el comparador de la figura 14 podría modelarse con el siguiente código:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity compeq4bits is
port(
     A:      in     std_logic_vector(3 downto 0);
     B:      in     std_logic_vector(3 downto 0);
     A_eq_B: buffer std_logic
    );
end entity;

architecture rtl of compeq4bits is
begin
    A_eq_B <= '1' when A = B else
              '0';

end rtl;
```

Cuando se compara el valor del dato de entrada con una constante, la comparación queda reducida a una decodificación –a activar una salida cuando en la entrada hay una determinada combinación de ceros y unos–. Por ejemplo, el circuito de la figura 15 compara el valor de un dato de entrada con 10_D o, lo que es lo mismo, decodifica la combinación 1010. Como puede observarse en este ejemplo, la estructura lógica de un comparador de igualdad con un valor constante es mucho más simple que la de un comparador con dos datos de entrada como el de la figura 14

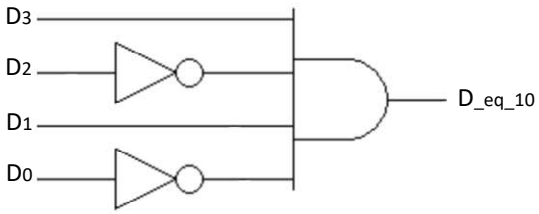


Figura 15.- Comparador con 10

El modelado del funcionamiento de estos subsistemas puede abordarse interpretando su función:

- Como una comparación de magnitud: Con una sentencia concurrente que hace uso de la comparación con un número.

```
D_eq_10 <= '1' when D = 10 else
           '0';
```

- Como una decodificación: Con una sentencia que detecta una combinación en la entrada.

```
D_eq_10 <= D(3) and (not D(2)) and D(1) and (not D(0));
```

o:

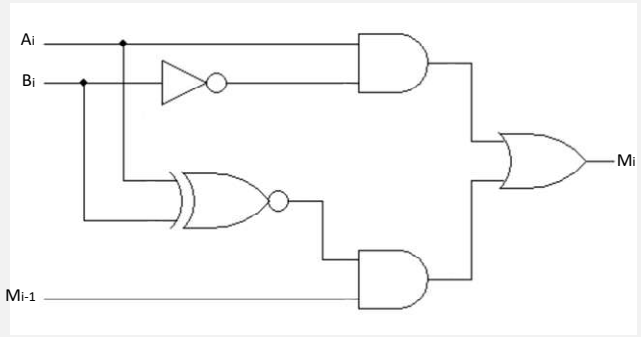
```
D_eq_10 <= '1' when D = "1010" else '0';
```

Diseño lógico y modelado HDL de comparadores de magnitud

Los comparadores que calculan cuál es el mayor, o el menor, de dos números son subsistemas complejos que, al igual que los sumadores y restadores, deben realizarse conectando en serie células elementales capaces de procesar un bit de cada uno de los datos que se comparan. En consecuencia son, también, sistemas cuya complejidad estructural y retardo son proporcionales al número de bits de dichos datos.

Por ejemplo, un comparador de números, codificados en binario natural, cuya salida se activa, a nivel alto, cuando el dato de entrada A es mayor que el dato de entrada B, puede realizarse conectando en serie células como las de la figura 16.

Ai	Bi	M_{i-1}	*Mi*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1



$$Mi = A_i \cdot \overline{B_i} + \overline{A_i \oplus B_i} \cdot M_{i-1}$$

Figura 16.- Célula A > B

La salida de esta célula, M_i, vale 1 cuando el bit de peso i de A es mayor que el bit homólogo de B (cuando $A_i = 1$ y $B_i = 0$) o cuando ambos bits son iguales (su *xnor* es 1) y la entrada M_{i-1} vale 1. M_{i-1} es una entrada de acarreo que indica a la célula el resultado de la comparación de los bits de peso menor que i –por eso decide el valor de la salida M_i cuando A_i y B_i son iguales. Conectando en serie n células como la de la figura 16 se construye un comparador para datos de N bits. En la figura 17 se muestra la realización de un comparador de 4 bits.

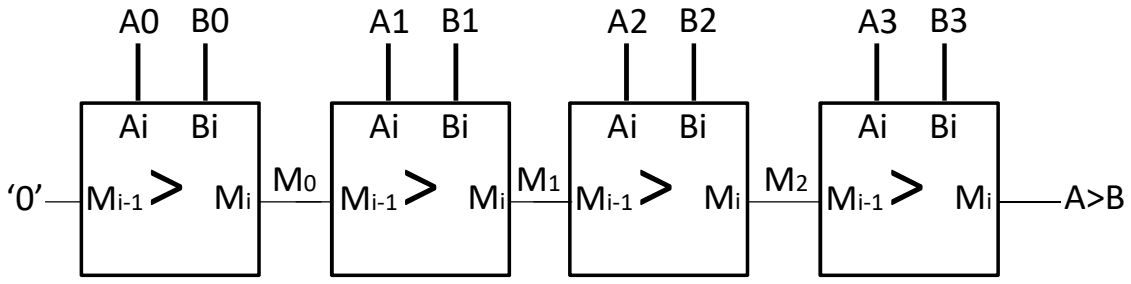


Figura 17.- Comparador de 4 bits A > B

La célula que procesa los bits de mayor peso determina el valor de la salida a partir de A_3 y B_3, si estos bits son distintos (si A_3 vale 1 y B_3 vale 0, A es mayor que B, en caso contrario A es menor que B), pero si A_3 y B_3 son iguales, el valor de la salida depende del nivel de la *entrada de acarreo* M_{i-1}. Pero M_{i-1} está conectado a la salida (M_2) de la célula que procesa A_2 y B_2 por lo que, en definitiva, si A_3 y B_3 son iguales, es la comparación entre A_2 y B_2 la que determina si A es o no mayor que B; si

estos bits también son iguales, la decisión dependería de la comparación entre A_1 y B_1 y, si estos también son iguales, de A_0 y B_0.

En un circuito como el de la figura 17 el tiempo de retardo máximo depende de la longitud de la cadena de propagación de acarreo, que va desde la salida de la célula que procesa los bits de menor peso hasta la entrada de la célula que procesa los de mayor peso: cuanto mayor sea la longitud de los datos que se comparan, más lento será el circuito. El consumo de recursos es también proporcional al número de bits de los datos, ya que hay que utilizar una célula como la de la figura 16 por cada bit.

Invirtiendo la salida del circuito de la figura 17 se construye un comparador con salida $A \leq B$. Y, por otra parte, si se intercambia la conexión de A_i y B_i en el circuito de la figura 16, se obtiene una célula que, con una conexión como la de la figura 17, permite realizar un comparador con salida $A < B$.

El modelado VHDL del funcionamiento de estos subsistemas es muy simple, puede realizarse con una sentencia concurrente de asignación condicional:

- $A > B$

```
A_my_B <= '1' when A > B else
          '0';
```

- $A \geq B$

```
A_my_eq_B <= '1' when A >= B else
             '0';
```

- $A < B$

```
A_mn_B <= '1' when A < B else
          '0';
```

- $A \leq B$

```
A_mn_eq_B <= '1' when A <= B else
             '0';
```

Cuando se compara la magnitud de un número con un valor constante, la complejidad de estos comparadores se reduce sustancialmente. El grado de simplificación depende del tipo de comparación y del valor concreto con el que se compare. Por ejemplo, el circuito de la figura 18a representa la estructura lógica de un circuito que indica si un dato de 4 bits es mayor que 9; en la figura 18b se representa un circuito que activa su salida cuando un dato, también de 4 bits, es menor o igual que 7.

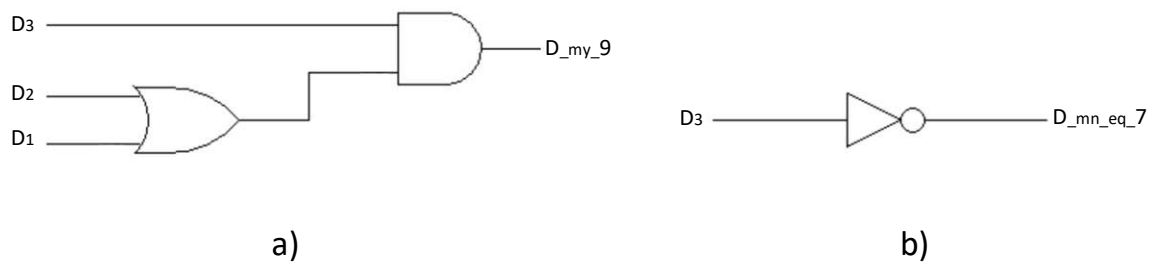


Figura 18.- Comparadores con constantes

El modelado de estos comparadores también puede realizarse con sentencias concurrentes de asignación condicional. Por ejemplo:

- $D > 9$

```
D_my_9 <= '1' when D > 9 else
          '0';
```

- $D \leq 7$

```
D_mn_eq_7 <= '1' when D <= 7 else
             '0';
```

Comparadores de magnitud de números codificados en Complemento a 2

El comparador de magnitud de la figura 17 es capaz de determinar el mayor de dos números codificados en binario natural, pero si en su entrada se ingresan números codificados en complemento a 2, la validez del resultado de la comparación depende del signo de los números:

1. Si se comparan dos números con el mismo signo, el resultado de la comparación es siempre correcto

POSITIVOS		NEGATIVOS	
0111	+7	**1**111	-1
0110	+6	**1**110	-2
0101	+5	**1**101	-3
0100	+4	**1**100	-4
0011	+3	**1**011	-5
0010	+2	**1**010	-6
0001	+1	**1**001	-7
0000	0	**1**000	-8

2. Si los números que se comparan tienen diferente signo, el resultado de la comparación es siempre erróneo, ya que el bit de signo de los positivos es cero y el de los negativos 1.

Teniendo en cuenta este comportamiento, resulta muy sencillo realizar un comparador de magnitud en complemento a dos a partir de uno binario: basta con invertir la salida del comparador cuando los bits de signo son distintos y *dejarla estar* cuando son iguales (figura 19).

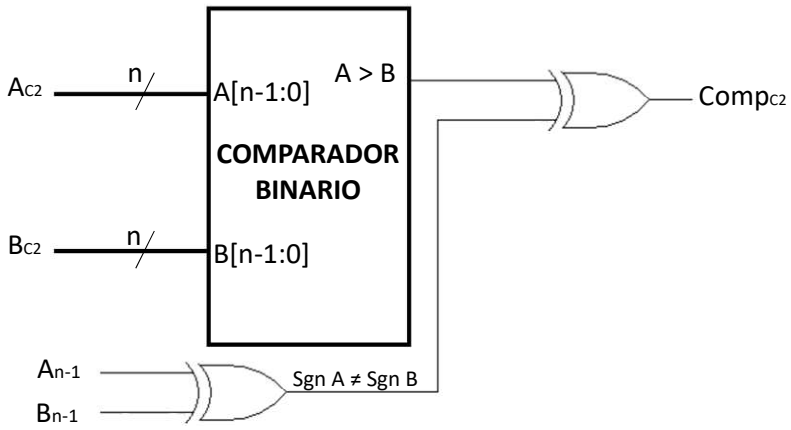


Figura 19.- Comparador en complemento a 2

Realice las siguientes operaciones:

1. Dentro del directorio correspondiente a la actividad, añada los ficheros *comp_bn_4bits.vhd* y *comp_bn_4bits_tb.vhd* a un nuevo proyecto –que debe crear- de ModelSim: *Proy_BT1_P5*.

2. Abra, con el editor, el fichero *comp_bn_4bits.vhd*, que contiene el modelo de un comparador de números en binario natural de 4 bits.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity comp_bn_4bits is
port(
     A:          in     std_logic_vector(3 downto 0);
     B:          in     std_logic_vector(3 downto 0);
     A_eq_B:     buffer std_logic; -- A igual a B
     A_neq_B:    buffer std_logic; -- A distinto de B
     A_my_B:     buffer std_logic; -- A mayor que B
     A_mn_eq_B: buffer std_logic; -- A menor o igual que B
     A_mn_B:     buffer std_logic; -- A menor que B
     A_my_eq_B: buffer std_logic  -- A mayor o igual que B
    );
end entity;

architecture rtl of comp_bn_4bits is
begin
    -- A igual a B y A distinto de B
    A_eq_B <= '1' when A = B else
              '0';
    A_neq_B <= not A_eq_B;

    -- A mayor que B y A menor o igual que B
    A_my_B <= '1' when A > B else
              '0';
    A_mn_eq_B <= not A_my_B;

    -- A menor que B y A mayor o igual que B
    A_my_eq_B <= A_my_B or A_eq_B;
    A_mn_B <= not A_my_eq_B;

end rtl;
```

   El subsistema modelado dispone de todas las posibles salidas de comparación de magnitud que puede haber y muestra diversas maneras de generar unas en función de otras.

3. Revise el código para comprender el modelo del comparador

   El fichero *comp_bn_4bits_tb.vhd* contiene el test-bench del comparador. El proceso que genera los estímulos utiliza dos bucles FOR para generar todas las posibles combinaciones de entrada.

4. Compile el modelo del comparador y el test-bench; a continuación ejecute una simulación y compruebe con el visor de formas de onda que las salidas se activan correctamente si los datos de entrada, A y B, se interpretan como números en binario natural.

5. Una vez hecha la comprobación anterior, cambie la representación de A y B a decimal para analizar el comportamiento de las salidas y comprobar que se comportan correctamente cuando A y B tienen el mismo signo y erróneamente cuando su signo es distinto –a excepción de la salida de igualdad y su complementaria que, independientemente del código con que se interpreten los datos, siempre funcionan bien.

   Si se desea modelar el funcionamiento de un comparador para números codificados en complemento a 2, hay dos opciones:

   - Modificar el código del cuerpo de arquitectura para complementar el bit de salida cuando el signo de A y B es distinto

   - Cambiar la cláusula de visibilidad sobre el paquete *std_logic_unsigned*, que antecede a la declaración de entidad, por otra que permite utilizar el contenido del paquete *std_logic_signed*.

   Con el primer procedimiento, bastaría cambiar la sentencia concurrente:

```
A_my_B <= '1' when A > B else
          '0';
```

   por:

```
A_my_B <= A(3) xnor B(3) when A > B else
          A(3) xor  B(3);
```

   Porque el resto de salidas a las que afecta el cambio de código se generan a partir de ésta.

   El segundo procedimiento es más simple. Basta con cambiar:

```
use ieee.std_logic_unsigned.all;
```

   por

```
use ieee.std_logic_signed.all;
```

   Al hacer esta modificación las operaciones de comparación adquieren un funcionamiento distinto, porque las comparaciones definidas en el paquete *signed* operan, precisamente, como si los datos de tipo *std_logic_vector* codificaran números en complemento a 2.

6. Modifique la cláusula de visibilidad del modelo del comparador para poder utilizar el paquete *std_logic_signed*.

7. Recompile el modelo modificado y su test-bench y ejecute una simulación para comprobar, con el visor de formas de onda, que el funcionamiento del modelo se corresponde ahora con el de un comparador de números en complemento a 2.

   Lógicamente el nuevo modelo tendría ahora un comportamiento anómalo si se interpretaran las entradas A y B como números codificados en binario natural.

   En conclusión: A diferencia de los sumadores y restadores, los comparadores de magnitud no pueden operar indistintamente con números en Complemento a 2 y Binario Natural.