4 Práctica 3. Sumadores

4.1 Introducción

Esta práctica tiene los siguientes objetivos:

1. Aprender a parametrizar la descripcion de una entidad usando los genéricos de VHDL.
2. Aprender a diseñar sistemas combinacionales iterativos mediante el uso de la clausula **generate** de VHDL.
3. Estudiar cómo distintas implementaciones de un mismo diseño poseen distintas características físicas.
4. Estudiar la escalabilidad de las distintas implementaciones en función del tamaño de los operandos.
5. Introducir nociones básicas de segmentación como técnica para aumentar el rendimiento en los circuitos digitales.

Vamos a lograr estos objetivos de aprendizaje diseñando tres tipos de sumadores para números en binario natural (esto es, números sin signo) de n bits: (1) sumador descrito usando el paquete `numeric_std`; (2) sumador con propagación de arrastres; y (3) sumador con anticipación de arrastres.

4.2 Especificaciones

Las tres implementaciones deben satisfacer las siguientes especificaciones:

Spec 1. El sumador se implementará como un circuito combinacional. No posee ni señal de reloj ni reset.

Spec 2. El sistema tiene tres puertos de entrada, `cin` de ancho 1 bit y `op1` y `op2` de g_width bits de ancho.

Spec 3. El sumador tiene un puerto de salida, `add`, de $g\_width + 1$ bits de ancho.

Spec 4. La entidad sumador viene definida por el siguiente código VHDL:

```
entity adder is
  generic (g_width : natural := 32)
  port    (cin     : in  std_logic;
           op1     : in  std_logic_vector(g_width-1 downto 0);
           op2     : in  std_logic_vector(g_width-1 downto 0);
           add     : out std_logic_vector(g_width   downto 0));
end adder ;
```

Dado que la definición de entidad será idéntica para las tres arquitecturas debemos de estructurar el código de forma que sólo sea necesario definir una entidad y tres arquitecturas distintas: una para cada tipo de sumador. Para ello, por un lado, definimos un archivo que contendrá sólo la definición de la entidad. Es decir sólo contendrá la definición expuesta en el código anterior. Por otro lado, cada una de las arquitecturas estará contenida en un archivo distinto que sólo contendrá la arquitectura. Así pues tendremos cuatro archivos:

- adder.vhd: que contendrá la definición entidad.
- rtl.vhd: que contendrá la descripción funcional del sumador.
- cpa.vhd: que contendrá la descripción del sumador con propagación de arrastres.
- cla.vhd: que contendrá la descripcion del sumador con anticipación de correos

Existen tres estilos de descripción en VHDL para especificar, con un mayor o menor nivel de detalle, la estructura del circuito que se desea que la herramienta de síntesis genere:

1. Estilo que permite la inferencia de una estructura. En él se especifica únicamente el comportamiento del circuito del modo más abstracto posible dando libertad a la herramienta para que elija la estructura y componentes más adecuadas bajo ciertas condiciones de funcionamiento.
2. Estilo que implica cierta estructura. En él se especifica tanto el comportamiento del circuito como una estructura global del mismo en base al conocimiento que tiene el diseñador de las condiciones de funcionamiento del circuito. La herramienta seleccionará las componentes del circuito
3. Estilo en el que se instancian ciertas componentes. En él se describe todo: función, estructura y componentes.

Los dos primeros estilos son independientes de tecnología, el tercero no. Por esta razón y salvo en contadas ocasiones deberá utilizarse preferiblemente el primer estilo; el segundo se reserva para pocas ocasiones y el tercero no es recomendable.

4.3 Sumador basado en el uso del paquete `numeric_std`

4.3.1 Diseño

En esta primera sección vamos realizar una descripción completamente funcional, sin incluir ningún tipo de descripción de la estructura que se debe implementar. Dejaremos que sea la herramienta de síntesis la que a partir de nuestra descripción infiera la implementación adecuada.

Vamos a emplear el paquete `numeric_std` para describir el sumador. Recordad que con anterioridad al año 2008 existían hasta tres paquetes distintos en los que se describían como realizar operaciones aritméticas en VHDL. Estos paquetes eran: std_logic_arith, std_logic_unsigned, std_logic signed. En la actualidad, el IEEE recomienda la sustitución de estos tres paquetes, ninguno de los cuales había sido definido por el IEEE, por un nuevo paquete, definido por un grupo de trabajo del IEEE, y que establece y unifica los procedimientos para realizar operaciones aritméticas en VHDL. Este paquete es `numeric_std`.

En este paquete se definen dos nuevos tipos de datos, **unsigned** y **signed**, y su aritmetica.

El tipo **unsigned** especifica un número natural representado en binario puro (números sin signo) estando el bit más significativo colocado a la izquierda. Todos los operadores que trabajan con este tipo de operandos realizan las operaciones en aritmética sin signo. La descripción de este tipo es en el paquete `numeric_std` es:

```
type unsigned is array (natural range <>) of std_logic;
```

El tipo **signed** especifica un número entero representado en complemento a 2, estando el bit de signo colocado a la izquierda. La descripción de este tipo es en el paquete numeric_std es:

```
type signed is array (natural range <>) of std_logic;
```

También se defiene los operadores aritméticos (p.e. +, -, *, /, ...) y las funciones de conversión entre estos tipos y los tipos nativos de VHDL (**std_logic_vector**, **integer**, ...). A continuación se presenta algunos ejemplos de operadores definidos en este paquete.

```
function "+" (L, R: UNSIGNED) return UNSIGNED;
  -- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Adds two UNSIGNED vectors that may be of different lengths.

  -- Id: A.4
  function "+" (L, R: SIGNED) return SIGNED;
  -- Result subtype: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Adds two SIGNED vectors that may be of different lengths.

  -- Id: A.5
  function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0).
  -- Result: Adds an UNSIGNED vector, L, with a non-negative INTEGER, R.
```

Las funciones de conversion convierten el entero a una reprentación con una anchura dada: si el número de bits requeridos por el valor a convertir fuera menor que el especificado, la representación se extiende adecuadamente; si por el contrario fuera mayor, se trunca por la izquierda.

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
  -- Result subtype: NATURAL. Value cannot be negative since parameter is an
  --              UNSIGNED vector.
  -- Result: Converts the UNSIGNED vector to an INTEGER.

  -- Id: D.2
  function TO_INTEGER (ARG: SIGNED) return INTEGER;
  -- Result subtype: INTEGER
  -- Result: Converts a SIGNED vector to an INTEGER.

  -- Id: D.3
  function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
  -- Result subtype: UNSIGNED(SIZE-1 downto 0)
  -- Result: Converts a non-negative INTEGER to an UNSIGNED vector with
  --         the specified SIZE.

  -- Id: D.4
  function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
  -- Result subtype: SIGNED(SIZE-1 downto 0)
  -- Result: Converts an INTEGER to a SIGNED vector of the specified SIZE.
```

En el paquete numeric_std no están escritos operadores aritméticos sobre el tipo **std_logic_vector**. Por lo tanto cualquier operación aritmética que se realice directamente sobre este tipo de datos dará lugar a un error de compilación.

Veamos a continuación un ejemplo de cómo realizar una suma de dos números de tipo **std_logic_vector** usando las funciones y tipos de datos definidos en esta librería:

```
architecture rtl of ejemplo is
   op1     : in  std_logic_vector(4 downto 0);
   op2     : in  std_logic_vector(4 downto 0);
   add     : in  std_logic_vector(4 downto 0);

   ...
begin
   ...
```

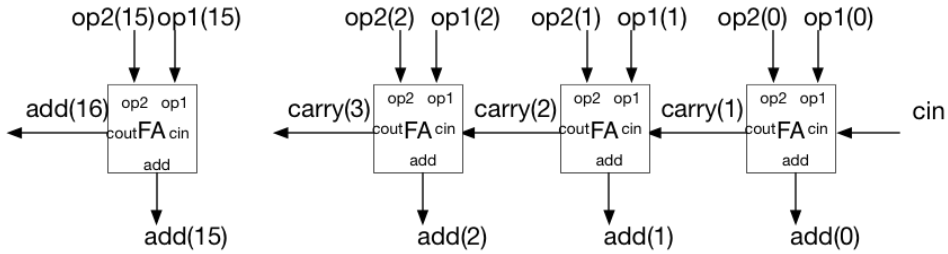

Figura 4.1: Diagrama de bloques de un sumador con propagación de arrastres de 16 bits. Cada uno de los bloques representa un sumador completo con sus tres entradas y dos salidas.

```
add   <= std_logic_vector(unsigned(op1) + unsigned(op2));
...
```

4.4 Sumador con propagacion de arrastres

4.4.1 Diseño

El sumador con propagacion de arrastres tiene la estructura que se muestra en la figura 4.1. Está formado por una colección de sumadores completos (FA) conectados en cascada de manera que el arrastre de salida, c_{out} de cada de los sumadores está conectado al arrastre de entrada del sumador adyacente, c_{in}.

Cada uno de los sumadores completos implementa las siguientes ecuaciones:

$$c_{out} = op_1 \cdot op_2 + c_{in} \cdot (op_1 + op_2) \quad (4.1)$$

$$add = op_1 \oplus op_2 \oplus c_{in} \quad (4.2)$$

4.4.2 Descripción VHDL

Vamos a estructurar el código de este diseño en dos archivos: fa.vhd, que contiene la descripción del sumador completo; y cpa.vhd que contiene la descripción de la arquitectura del sumador con propagacion de arrastres. La definicion de entidad del módulo fa será:

```
entity fa is
  port(op1  : in  std_logic;
       op2  : in  std_logic;
       cin  : in  std_logic;
       cout : out std_logic;
       sum  : out std_logic);
end entity fa;
```

Puesto que deseamos que la descripción del sumador con propagación de correos sea parametrizable para cualquier ancho de los operandos op1 y op2, vamos hacer uso de la sentencia generate de VHDL para poder generar estructuras parametrizables de tamaño variable. El código siguiente ilustra cómo podemos hacer uso de esta sentencia para construir la estructura del sumador con propagacion de arrastres.

```
p_cpa : for i in 0 to g_width-1 generate
  i_fa : fa port map (
    op1  => op1(i),
    op2  => op2(i),
    cin  => carry(i),
```

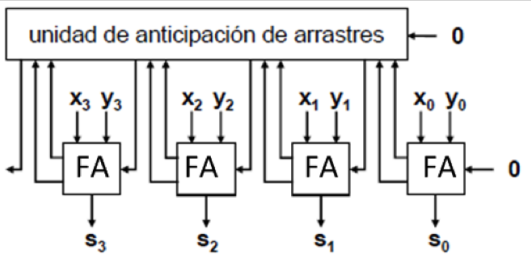

Figura 4.2: Diagrama de bloques de un sumador con propagación de arrastres de 4 bits. Cada uno de los bloques representa un sumador completo con sus tres entradas y dos salidas.

```
        sum  => add(i),
        cout => carry(i+1));
    end generate p_cpa;
```

En el código anterior se hace uso de la señal arrastre. Por lo tanto, esta señal debe ser definida en la parte declarativa de la arquitectura:

```
signal carry : std_logic_vector(g_width downto 0);
```

4.4.3 Análisis del sumador

El sumador con propagación de arrastres tiene la siguientes características:

1. Camino critico largo, comienza en el puerto de arrastre de entrada, c_{in} y finaliza en el bit más significativo de la suma.
2. Área reducida.
3. La longitud del camino crítico crece de forma lineal con la longitud de los operandos.

4.5 Sumador con anticipación de arrastres

4.5.1 Diseño

El sumador con anticipación de arrastres, también conocido como *carry lookahead adder*, es un sumador en el que los arrastres se calculan en paralelo, mediante una lógica específica con un camino crítico más corto. Todo ello con el objetivo de acelerar el cálculo de la suma.

El sumador con anticipacion de arrastres tiene la estructura que se muestra en la figura 4.2.

El objetivo por este tipo de sumadores es reducir el tiempo de cálculo mediante la reduccion del camino crítico. Para ello, ahora cada uno de los sumadores completos de 1-bit, FA, ya no calcula el arrastre de salida sino que calcula dos nuevas variables g y p que son utilizadas por la unidad de anticipacion de arrastres (uaa) para calcular el arrastre de entrada de cada uno de los FAs. Así cada uno de los sumadores completos implementa las siguientes ecuaciones:

$$add = op_1 \oplus op_2 \oplus c_{in} \tag{4.3}$$
$$g = op_1 \cdot op_2 \tag{4.4}$$
$$p = op_1 \oplus op_2 \tag{4.5}$$

La señal g, llamada generación, valdrá 1 si el sumador completo genera un arrastre. Es decir, cuando ambos operandos de entrada, op_1 y op_2 sean igual a 1. La señal p, llamada propagación,

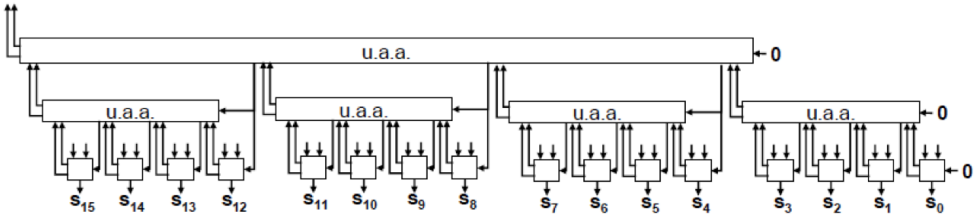

Figura 4.3: Diagrama de bloques de un sumador de 16 bits con propagación de arrastres multinivel. Cada uno de los bloques representa un sumador completo con sus tres entradas y dos salidas.

valdrá 1 si el sumador completo propaga un arrastre. Es decir, cuando bien op_1 u op_2 sean igual a 1. Sustituyendo estas expresiones en la Ecuación (4.2) obtenemos:

$$c_{out} = op_1 \cdot op_2 + c_{in} \cdot (op_1 + op_2) \quad (4.6)$$
$$= g + p \cdot c_{in} \quad (4.7)$$

La unidad de anticipación de arrastres calcula el arrastre de entrada de cada uno de los sumadores usando la Ecuación (4.7):

$$c_1 = g_0 + p_0 \cdot c_0 \quad (4.8)$$
$$c_2 = g_1 + p_1 \cdot c_1 \quad (4.9)$$
$$= g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) \quad (4.10)$$
$$= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \quad (4.11)$$
$$c_3 = g_2 + p_2 \cdot c_2 \quad (4.12)$$
$$= g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) \quad (4.13)$$
$$= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \quad (4.14)$$
$$c_4 = g_3 + p_3 \cdot c_3 \quad (4.15)$$
$$= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \quad (4.16)$$

El problema de esta estructura es que al aumentar el número de bits de los operandos op_1 y op_2 aumenta la complejidad de la unidad de anticipacion de arrastres y su retardo. Por ejemplo, en el caso de tener que calcular el arrastre de entrada del sumador completo en la posición 32 se requerirían calcular 32 términos producto cada uno de ellos con 33 variables.

$$c_{31} = g_{30} + p_{30} \cdot c_{30} \quad (4.17)$$
$$= g_{30} + p_{30} \cdot g_{29} + p_{30} \cdot p_{29} \cdot g_{28} + \ldots + p_{30} \cdot p_{29} \cdot \ldots \cdot p_0 \cdot c_0 \quad (4.18)$$

Para evitar este crecimiento en el tamaño de la lógica se utilizará una aproximación basada en anticipación de arrastre multinivel. La figura 4.3 muestra la estructura un sumador de 16 bits con anticipación de arratre multinivel.

Ahora las unidades de anticipación de arrastre generan dos nuevas señales g^* y p^*. La señal g^* valdrá 1 si el conjunto de sumadores completos conectados a dicha unidad genera un arrastre y este se propaga hasta el arrastre de salida del sumador situado más a la izquierda. Es decir, se generara un arrastre si se genera en alguna de los sumadores completos y éste se propaga hasta la salida. La señal p^* valdrá 1 si hay un arrastre de entrada y todas los sumadores completos lo propagan hasta el arrastre de salida. Las ecuaciones siguientes expresan esta relación:

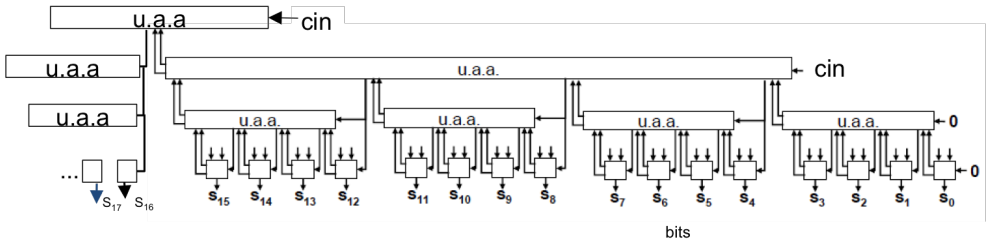

Figura 4.4: Diagrama de bloques de un sumador de 32 bits con anticipación de arrastre multinivel de 4 bits.

$$g^* = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \quad (4.19)$$

$$p^* = p_3 \cdot p_2 \cdot p_1 \cdot p_0 \quad (4.20)$$

$$c_{out} = g^* + p^* \cdot cin_{in} \quad (4.21)$$

Esta aproximación puede extenderse a un número arbitrario de bits. La figura 4.4 ilustra la estructura de un sumador de 32 bits con anticipación de arrastre y unidades de anticipación de arrastres de 4 bits.

4.5.2 Descripción VHDL

Vamos a estructurar el código de este diseño en tres archivos: gp.vhd, que contiene la descripción del sumador completo; uaa. vhd que contiene la descripción de la unidad de anticipación de arrastres y cpa.vhd que contiene la descripción de la arquitectura del sumador con anticipación de arrastres. La definicion de entidad del módulo gp será:

```
entity gp is
  port (
    op1 : in  std_logic;
    op2 : in  std_logic;
    cin : in  std_logic;
    g   : out std_logic;
    p   : out  std_logic;
    add : out  std_logic);
end entity gp;
```

La definicion de entidad del módulo uaa será:

```
entity uaa is
  port (
    cin   : in  std_logic;
    pin   : in  std_logic_vector(3 downto 0);
    gin   : in  std_logic_vector(3 downto 0);
    pout  : out std_logic;
    gout  : out std_logic;
    carry : out std_logic_vector(3 downto 0);
    cout  : out std_logic);
end entity uaa;
```

En este caso no deseamos que la estructura sea parametrizable a cualquier ancho de los operandos op1 y op2 pero si deseamos que la descripción sea lo más compacta posible haciendo uso de la sentencia `generate`. Para ello vamos a utilizar tres bucles for ...generate que servirán para instanciar todos los componentes del diseño. El siguiente código ilustra cómo podemos hacer uso de esta sentencia para construir la estructura de las unidades de anticipación de arrastres de primer nivel.

```
uaa_1 : for i in 0 to c_uaa1-1 generate
  i_uaa : uaa port map (
    cin   => carry1(i),
    pin   => p0(4*i+3 downto 4*i),
    gin   => g0(4*i+3 downto 4*i),
    pout  => p1(i),
    gout  => g1(i),
    carry => carry0(4*i+3 downto 4*i),
    cout  => cdummy(i));
```

donde la constante se define en la parte declarativa de la arquitectura

```
constant c_uaa1 : natural := g_width/4;
```

4.6 Cuestiones y resultados experimentales

La documentación a presentar en la memoria es:

1. Describir la implementación realizada por XST de los FA para el sumador con propagación de arrastres.
2. Describir la implementación realizada por XST de los FA para el sumador con anticipación de arrastres.
3. Crear una gráfica en la que se muestre el retardo del camino crítico para los tres tipos de sumadores y para anchos de los operandos de 16 y 32 bits. ¿Cuál es el sumador más rápido? ¿Cuál es el más lento? ¿Te han sorprendido los resultados? Si es así, indica por qué.
4. Crear una gráfica en la que se muestre el número de slices para los tres tipos de sumadores y para anchos de los operandos de 16 y 32 bits.
5. Describir el camino crítico encontrado por la herramienta para los sumadores de 16 bits con propagación de arrastres y el sumador de 16 bits con anticipación de arrastres: señales fuente y destino e indicar sobre el diagrama de bloques de las Figuras 4.1 y 4.3 por dónde transcurre dicho camino.
6. Realizar una simulación post-place&route. ¿Existen glitches a las salidas del circuito? Si existen, indicad un valor de las entradas para las que aparecen dichos glitches. ¿Existe algún valor en las entradas para las cuales el retardo del sumador con propagación de acarreos coincida con el camino crítico calculado por XST?