4 Práctica 3. Multiplicador en array

El objetivo de esta práctica es doble.

1. Por un lado, diseñar sistemas combinacionales iterativos mediante el uso de la clausula **generate** de VHDL.
2. Introducir nociones básicas de segmentación como técnica para aumentar el rendimiento en los circuitos digitales.

El vehículo de pruebas que vamos a utilizar para cumplir estos objetivo es el multiplicador en array de números sin signo.

4.1 Multiplicador de 5×5 bits

4.1.1 Especificaciones

En la primera parte de la práctica vamos a diseñar, simular, síntetizar e implementar un multiplicador en array para número sin signo de 5x5 bits.

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

Spec 2. El sistema tiene dos puertos de entrada, `op1` y `op2`, cada uno de 5 bits de ancho.

Spec 3. El multiplicador tiene un puerto de salida, `result`, de 10 bits de ancho.

Spec 4. La entidad de la muñeca viene definida por el siguiente código VHDL:

```
entity mult is
  port (op1    : in  std_logic_vector(4 downto 0);
        op2    : in  std_logic_vector(4 downto 0);
        result : out std_logic_vector(9 downto 0));
end mult;
```

4.1.2 Diseño

La siguiente figura ilustra la multiplicación de dos números, a y x, de 5 bits sin signo.

					a_4	a_3	a_2	a_1	a_0
					x_4	x_3	x_2	x_1	x_0
					a_4x_0	a_3x_0	a_2x_0	a_1x_0	a_0x_0
				a_4x_1	a_3x_1	a_2x_1	a_1x_1	a_0x_1	
			a_4x_2	a_3x_2	a_2x_2	a_1x_2	a_0x_2		
		a_4x_3	a_3x_3	a_2x_3	a_1x_3	a_0x_3			
	a_4x_4	a_3x_4	a_2x_4	a_1x_4	a_0x_4				
PP_9	PP_8	PP_7	PP_6	PP_5	PP_4	PP_3	PP_2	PP_1	PP_0

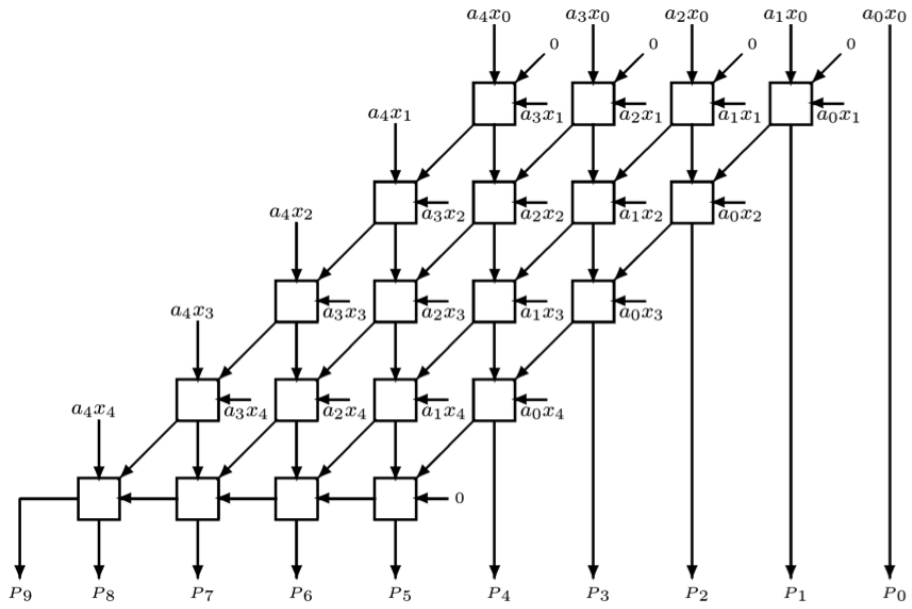

Figura 4.1: Diagrama de bloques de un multiplicador en array de 5×5 bits. Cada uno de los bloques representa un sumador completo con sus tres entradas y dos salidas.

Cada término $a_i x_j$ recibe el nombre de sumando, S_{ij}, y cada fila de sumados es llamada producto parcial. Los productos parciales pueden calcularse simultáneamente y organizarse en una matriz para realizar su suma.

Esta multiplicación puede realizarse con el multiplicador en array de 5×5 bits que se presenta en la Figura 4.1, cuya organización reproduce el procedimiento manual de multiplicación.

Para implementarlo es necesario un único tipo de sumadores completos (Full Adder, FA) organizados tal y como se ilustra en la Figura 4.1. Cada fila de FAs calcula un nuevo PP y lo añade al PP previamente calculado.

Esta clase de multiplicadores son interesantes por tres razones:

1. Una estructura bastante regular que simplifica su implantación.
2. La estructura con conexiones locales muy cortas que reducen los tiempos de los caminos críticos.
3. Pueden ser segmentados con bastante facilidad.

Su mayor inconveniente es el tiempo de propagación. Su valor para un multiplicador de $n \times m$ bits:

$$t_{\mathtt{pro}} = t_{\mathtt{PP}} + (n-1) \cdot t_{\mathtt{sum}} + ((m-1) + (n-2)) \cdot t_{\mathtt{carry}} \tag{4.1}$$

donde $t_{\mathtt{sum}}$ es el tiempo de cálculo de la suma en cada FA y $t_{\mathtt{carry}}$ es el tiempo de cálculo del carry en cada FA. Hay múltiples caminos candidatos a tener el tiempo de propagación más largo al tener una longitud muy similar. Por lo tanto la mejora del tiempo de cálculo requiere acelerar tanto la generación de las sumas como de los acarreos para todos ellos. No obstante, en esta práctica no se va a utilizar ninguna técnica de aceleración.

En el caso que nos ocupa, la multiplicación requiere 25 puertas AND de 2 bits, aunque la mayor parte del área del multiplicador está dedicada a la suma de los 5 PPs, que requiere 2 FAs. Como se puede ver en la Figura 4.2 los FAs constan de tres entradas, x, y y c_{in}, y dos salidas, s y c_{out}.

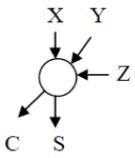

Figura 4.2: Entradas/salidas de un sumador completo de 1 bit.

Las ecuaciones del FA son:

$$s = a \oplus b \oplus cin \tag{4.2}$$

$$cout = a \cdot b + cin \cdot (a \oplus b) \tag{4.3}$$

4.1.3 Descripción VHDL

La estructura regular del diseño permite la utilización de la clausula **generate**. Dado que el circuito es un array 2D de sumadores es necesario anidar dos construcciones **generate**: la primera sirve para generar la estructura de sumadores de cada fila –conectando todos los sumadores de una fila organizados en columnas–; la segunda sirve para recorrer todas las filas desde la primera hasta la quinta.

```
rows : for i in 0 to 3 generate
  columns : for j in 0 to 9 generate
    ...
  end generate columns;
end generate rows;
```

En el cuerpo del bucle **generate** se instancia un sumador completo de 1 bit conectando sus puertos de entrada y salida mediante 3 arrays de señales 2D –c, mul y sum– y un array 1D –suma_ini–.

```
i_u : adder port map(a    => sum_ini(j),
                     b    => c(i, j),
                     cin  => mul(i, j),
                     sum  => sum(i+1, j),
                     cout => c(i+1, j+1));
```

La forma más cómoda de describir los arrays de señales 2D es definiendo un nuevo tipo –t_matrix– y declarando las señales usando este tipo.

```
type t_matrix is array (0 to 5, 10 downto 0) of std_logic;

signal c, sum, mul : t_matrix                    ;
signal sum_ini     : std_logic_vector(9 downto 0) ;
```

El código anterior genera la estructura regular de 5×5 sumadores que se ilustra en la Figura 4.3. Como se puede apreciar, hay un exceso de sumadores. Para que estos sumadores no intervengan en la suma, la solución más sencilla es que al menos uno de sus operandos sea 0. Esa es la razón por la que la señales c, sum, mul y sum_ini se inicializan a cero. Esas señales tendrán el valor 0 salvo en aquellos casos que tengan que contener un sumado S_{ij}.

Para asignar el valor S_{ij} a cada una de las señales se usará un proceso combinacional que calcula el valor S_{ij} para todas las componentes de los operandos op1 y op2.

```
p_multiplications : process (a, b)
--variable v_producto : t_pp := ((others => (others => '0')));
begin  -- process multiplications
  for i in 0 to 4 loop
    for j in 0 to 4 loop
```

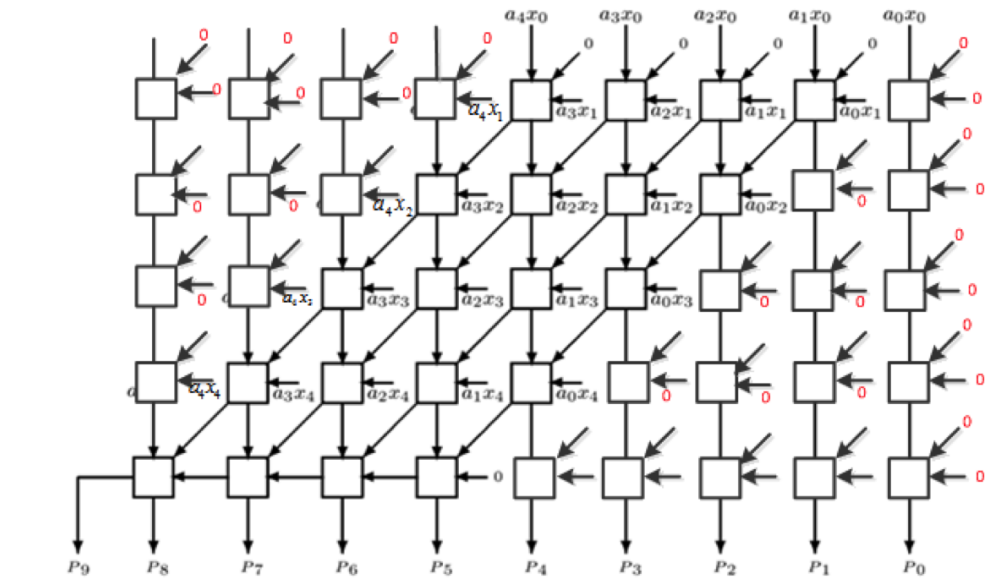

Figura 4.3: Multiplicador 5×5 descrito en VHDL.

```
      if i = 0 then
        sum_ini(j) <= a(0) and b(j);
      else
        mul(i-1, j+i) <= a(i) and b(j);
      end if;
    end loop;  -- j
  end loop;  -- i
end process p_multiplications;
```

4.1.4 Síntesis

Durante el proceso de síntesis, y siempre que no se indique explícitamente lo contrario, la herramienta XSP elimina toda aquella lógica innecesaria. En nuestro diseño son innecesarios todos aquellos FAs que tienen dos entradas igual a 0, puesto que la suma de cualquier valor con 0 es siempre igual al mismo valor. Por tanto todos esos sumadores serán sustituidos por una conexión que unirá la entrada distinta de cero con la salida. Así, la herramienta eliminará todos los sumadores marcados con una cruz en la Figura 4.4.

Tras el proceso de eliminación, la estructura resultante es idéntica a la presentada en la Figura 4.1.

4.2 Multiplicador en array de $n \times n$ bits

4.2.1 Especificaciones

En esta segunda parte de la práctica vamos a diseñar, simular, síntetizar e implementar un multiplicador en array para número sin signo de $n \times n$ bits. El diseño del multiplicador debe ser parametrizable en función de un genérico, n. En el laboratorio se comprobará la funcionalidad para distintos valores de n.

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

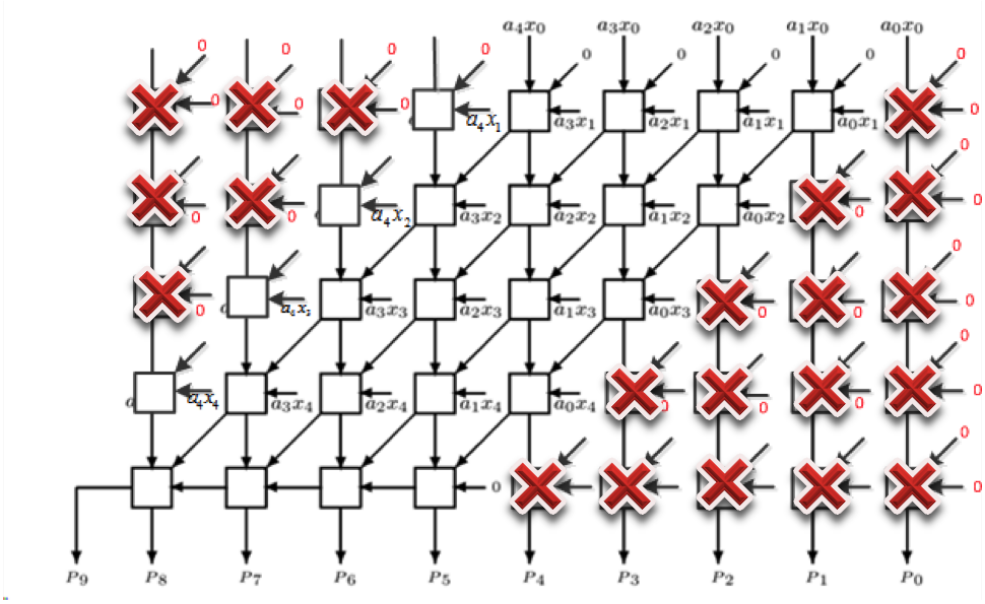

Figura 4.4: Resultado proceso de síntesis.

Spec 2. El sistema tiene dos puertos de entrada, op1 y op2, cada uno de n bits de ancho.

Spec 3. El multiplicador tiene un puerto de salida, `result`, de $2n$ bits de ancho.

Spec 4. La entidad de la muñeca viene definida por el siguiente código VHDL:

```
entity mult is
  generic (n    : natural);
  port (op1     : in  std_logic_vector(n-1   downto 0);
        op2     : in  std_logic_vector(n-1   downto 0);
        result  : out std_logic_vector(2*n-1 downto 0));
end mult;
```

4.2.2 Descripción VHDL

La estructura regular del diseño permite la utilización de la clausula **generate**. Dado que el circuito es un array 2D de sumadores es necesario anidar dos construcciones **generate**: la primera sirve para generar la estructura de sumadores de cada fila –conectando todos los sumadores de una fila organizados en columnas–; la segunda sirve para recorrer todas las filas desde la primera hasta la quinta.

```
rows : for i in 0 to n-2 generate
  columns : for j in 0 to 2*n-1 generate
    ...
  end generate columns;
end generate rows;
```

En el cuerpo del bucle **generate** se instancia un sumador completo de 1 bit conectando sus puertos de entrada y salida mediante 3 arrays de señales 2D –c, mul y sum– y un array 1D –suma_ini–.

```
i_u : adder port map(a    => sum_ini(j),
                     b    => c(i, j),
                     cin  => mul(i, j),
                     sum  => sum(i+1, j),
                     cout => c(i+1, j+1));
```

La forma más cómoda de describir los arrays de señales 2D es definiendo un nuevo tipo –t_matrix– y declarando las señales usando este tipo.

```
type t_matrix is array (0 to n, 2*n downto 0) of std_logic;

signal c, sum, mul : t_matrix                          := ((others => (others => '0')));
signal sum_ini     : std_logic_vector(2*n-1 downto 0) := (others  => '0');
```

El código anterior genera una estructura regular de $n \times n$ sumadores. Nuevamente hay un exceso de sumadores y para que estos sumadores no intervengan en la suma, la solución más sencilla es que al menos uno de sus operandos sea 0. Esa es la razón por la que la señales c, sum, mul y sum_ini se inicializan a cero. Esas señales tendrán el valor 0 salvo en aquellos casos que tengan que contener un sumado S_{ij}.

Para asignar el valor S_{ij} a cada una de las señales se usará un proceso combinacional que realiza calcula el S_{ij} para todas las componentes de los operandos op1 y op2.

```
p_multiplications : process (a, b)
--variable v_producto : t_pp := ((others => (others => '0')));
begin  -- process multiplications
  for i in 0 to n-1 loop
    for j in 0 to n-1 loop
      if i = 0 then
        sum_ini(j) <= a(0) and b(j);
      else
        mul(i-1, j+i) <= a(i) and b(j);
      end if;
    end loop;  -- j
  end loop;  -- i
end process p_multiplications;
```

4.3 Multiplicador en array segmentado

Como se mencionó en la Sección 4.1, una de las ventajas de este tipo de multiplicadores es su facilidad para ser segmentados. En este sección vamos a segmentar el multiplicador de 5x5 bits en cinco etapas con el propósito de aumentar la tasa a la que pueden realizar multiplicaciones. Para ello deben insertarse flip-flops a la salida de cada productos parciales del diagrama de bloques del multiplicador de la Figura 4.1.

4.4 Cuestiones y resultados experimentales

La documentación a presentar en la memoria del apartado 4.1 es:

1. Describir la implementación realizada por XST de los FA.
2. Describir la implementación realizada por XST de la red iterativa 2D.
3. Indicar el número de FF, LUT y puertas básicas que ha inferido XST.
4. Describir el camino crítico encontrado por la herramienta: señales fuente y destino e indicar sobre el diagrama de bloques de la Figura 4.1 por donde transcurre dicho camino. ¿Cuál es la frecuencia máxima de trabajo?
5. Realizar una simulación post-place&route y determinar:
   a) ¿Existen glitches a las salidas del circuito? Si existen indicar valores de las entradas con las que aparecen dichos glitches.
   b) ¿Existe algún valor en las entradas para las cuales el retardo en calcular las salidas coincida con el camino crítico calculado por XST?

Para que el código de la práctica sea considerado correcto se deben cumplir los siguientes criterios:

1. La simulación del sistema debe ser correcta.
2. El código debe reflejar los detalles del diseño presentado en la memoria.
3. El código debe seguir todas las reglas incluidas en el documento “Reglas de estilo”.