4 Bloques aritméticos combinacionales para números naturales

Juan J. Navarro
Toni Juan

Primera versión: 09-2007, versión actual: 07-2014

4.1 Introducción

En el capítulo 3 hemos tratado un método sistemático de sintetizar circuitos combinacionales a partir de su tabla de verdad: obtención de la expresión en suma de minterms para cada salida y dibujo del esquema lógico que es una implementación directa de estas expresiones con puertas Not, And y Or, o con un decodificador y puertas Or, o con una ROM. No obstante, este método no sirve cuando se trata de circuitos con muchas entradas. Por ejemplo, un sumador de dos números naturales representados en binario con 16 bits cada uno. El sumador combinacional tiene 32 bits de entrada, 16 para codificar cada sumando, y 16 bits (o 17) de salida, para codificar el resultado. Una tabla de verdad que especifique este circuito tiene 2^{32} filas y 16 (o 17) columnas de salida. Las salidas no son un problema, pero las entradas sí: el número de filas crece exponencialmente con el número de entradas del circuito. Este sumador requiere una tabla de verdad con más de 4.000 millones de filas, lo que es claramente irrealizable. Pero no sólo es imposible dibujar la tabla de verdad, lo peor es tener que implementar los cientos o miles de millones de posibles minterms de 32 entradas; y esto solo para un sumador...

En un computador se realizan diferentes operaciones sobre números codificados con bastantes bits (16 en nuestro computador, pero lo usual son 32 o 64 bits por número) y por lo tanto no podemos usar el método de síntesis del capítulo anterior. ¿Cómo lo haremos?

Sigamos con el ejemplo del sumador. Pensemos cómo sumamos en decimal con varios dígitos. ¿Sabemos de memoria el resultado de sumar cualquier par de números (por ejemplo: 16965 + 94672)? La respuesta es ¡no! Sólo sabemos de memoria la suma de los números de un dígito, por ejemplo: 5 + 2 = 7, 6 + 7 = 13, ... Para números de varios dígitos aplicamos un **algoritmo**, esto es, los pasos a seguir para encontrar cada uno de los dígitos que representan el resultado. Aplicamos tantos pasos como dígitos tienen los números y en cada paso sumamos los dos dígitos de ese paso más el posible acarreo (lo que nos llevamos) del paso anterior. Sólo necesitamos saber de memoria el algoritmo y la suma de dos dígitos cualesquiera.

En la implementación del sumador binario que hacemos en este capítulo usamos un algoritmo similar al que ya conocemos para sumar en decimal, pero adaptado a binario. Primero definiremos el dispositivo Full-adder, capaz de sumar dos dígitos (dos bits, en el caso binario) más un posible acarreo; que es lo que se hace en cada paso del algoritmo. El Full-adder sólo tiene 3 bits de entrada (y dos de salida), por lo que sí podemos implementarlo a partir de su tabla de verdad, usando el método sistemático estudiado en el capítulo anterior. Una vez tengamos este dispositivo lo replicaremos tantas veces como bits tienen los números a sumar y conectaremos adecuadamente los Full-adders para que realicen el algoritmo. Esto no es un diseño sistemático como los del capítulo anterior, es un **diseño ad-hoc**, y por ello es más difícil, requiere experiencia en diseños parecidos, tener ideas nuevas, etc.

En este capítulo vamos a ver unos cuantos ejemplos de diseño de circuitos combinacionales que procesan números de n bits (n=16, si no se indica lo contrario). Estos circuitos consistirán en la interconexión de **bloques** o dispositivos combinacionales más pequeños. En algunos casos, cada uno de estos bloques estará formado, a su vez, por la interconexión de bloques todavía más pequeños. Esto es, haremos un diseño ad-hoc a bloques **con varios niveles de bloques**. Sólo cuando lleguemos a un bloque que tenga pocas entradas, por ejemplo 4 o menos, lo implementaremos con el método sistemático que conocemos: en suma de minterms.

Concretamente, tratamos las operaciones aritméticas básicas sobre números naturales representados en binario: suma (sección 4.2). resta (4.3), multiplicación y división por potencias de dos (4.4 y 4.5) y comparaciones (4.6). Para cada operación, primero obtenemos el **algoritmo aritmético**: procedimiento para encontrar los bits que representan el resultado de la operación a partir de los bits que representan a los operandos. Aunque desde pequeños conocemos estos algoritmos para la representación decimal, aquí los obtenemos demostrando su validez para el sistema convencional en base b (o directamente para el caso binario, b = 2). Después, para cada algoritmo dibujamos el esquema lógico de un circuito combinacional que implementa el algoritmo en binario. Además, en este capítulo, vemos otros bloques que no son aritméticos: operadores lógicos bit a bit sobre vectores de n bits (sección 4.7) o multiplexores grandes construidos con multiplexores más pequeños (4.8). Por último, en la sección 4.10 diseñamos nuevos bloques aritméticos usando los bloques básicos vistos en las secciones anteriores.

4.2 El sumador binario

En la sección 4.2.1 obtenemos el algoritmo aritmético de la suma de dos números naturales y en la 4.2.2 lo implementamos conectando Full-adders en propagación del acarreo. Por último, en la sección 4.2.2 se implementa el bloque Full-adder mediante dos Half-adders y una puerta Or-2 y el Half-adder mediante una puerta And-2 y una Xor-2. Con ello vemos el bloque sumador como un ejemplo de diseño ad-hoc con tres niveles de bloques internos.

4.2.1 Algoritmo aritmético de suma en base b

Planteamiento del problema de la suma

Dados dos vectores de n dígitos, $X = x_{n-1}x_{n-2}\cdots x_2x_1x_0$ e $Y = y_{n-1}y_{n-2}\cdots y_2y_1y_0$ con $x_i, y_i \in \{0, ..., b\text{-}1\}$ para i = 0, ..., n-1, que representan a los números naturales X_u e Y_u en el sistema convencional en base b, queremos encontrar los dígitos del vector W tales que: $W_u = X_u + Y_u$.

Lo primero que nos planteamos es cuántos dígitos tiene que tener W para poder representar a $X_u + Y_u$. Como el rango de posibles valores de X_u e Y_u es $0 \le X_u, Y_u \le b^n - 1$, la suma $W_u = X_u + Y_u$ estará en el rango $0 \le W_u \le 2b^n - 2$. Esto nos indica que el resultado W_u necesita, en el peor de los casos, un vector de n+1 dígitos ($W = w_n w_{n-1} w_{n-2}\cdots w_2 w_1 w_0$), ya que $b^{n+1} - 1 > 2b^n - 2 > b^n - 1$: con $0 \le w_i \le b - 1$.

Así que realizar la suma consiste en encontrar w_i para i = 0, ..., n, tales que cumplan

$$\sum_{i=0}^{n} w_i b^i = \sum_{i=0}^{n-1} x_i b^i + \sum_{i=0}^{n-1} y_i b^i \qquad \textbf{(EQ 1)}$$

y que los valores w_i sean dígitos válidos en base b,

$$0 \le w_i \le b - 1 \qquad \textbf{(EQ 2)}$$

Una solución en dos fases al problema de la suma

La EQ (1) define el problema de encontrar n +1 incógnitas que cumplen una sola ecuación. Este problema tiene infinitas soluciones (es un sistema de ecuaciones indeterminado). Pero si aplicamos la restricción que define la EQ (2) sobre los posibles valores que pueden tener los dígitos w_i, la solución es única, como veremos a continuación. Vamos a encontrar esta solución en dos fases. En la primera fase encontramos una solución a la EQ (1) y en la segunda transformamos en n pasos la solución anterior para que cumpla además la restricción de la EQ (2).

Fase 1. Manipulando la EQ (1) encontramos la siguiente expresión equivalente:

$$w_n b^n + \sum_{i=0}^{n-1} w_i b^i = \sum_{i=0}^{n-1} (x_i + y_i) b^i$$

Observando esta ecuación podemos encontrar una solución trivial, de las infinitas que hay que no tienen porqué cumplir la EQ (2), y que denotamos con un sombrerito:

$$\hat{w}_i = x_i + y_i \qquad \text{para} \qquad 0 \le i \le n-1 \quad \text{y} \quad \hat{w}_n = 0$$

La siguiente tabla muestra el resultado de aplicar la primera fase del algoritmo de suma en base b =10 y n = 4, a los vectores de dígitos X = 6493 e Y = 8199.

		Dígito 4	**Dígito 3**	**Dígito 2**	**Dígito 1**	**Dígito 0**
	X		6	4	9	3
	Y		8	1	9	9
$\hat{W}$		0	14	5	18	12

Posible transformación de la solución de la primera fase. El valor de los dígitos encontrados puede no ser menor que la base b (de hecho: $0 \le \hat{w}_i \le 2b-2$), por lo que la solución no siempre cumple la EQ (2). Si nos fijamos en un dígito k que no cumpla la EQ (2), $w_k \ge b$, y deseamos una solución en la que al menos el dígito k cumpla con la EQ (2) además de cumplir con la EQ (1), solamente tenemos que modificar el valor del dígito w_k y del w_{k+1} de la solución anterior. Para que la nueva solución, que denominamos $\tilde{W} = \tilde{w}_n \tilde{w}_{n-1} \tilde{w}_{n-2} \cdots \tilde{w}_2 \tilde{w}_1 \tilde{w}_0$,cumpla la EQ (1) dejamos todos los dígitos iguales a la anterior solución excepto el k y el k+1. Sumamos y restamos la base b al dígito k y manipulamos la expresión original como sigue (sólo mostramos los sumandos de los dos dígitos que manipulamos):

$$\hat{w}_{k+1} b^{k+1} + \hat{w}_k b^k = \hat{w}_{k+1} b^{k+1} + (\hat{w}_k - b + b) b^k = \hat{w}_{k+1} b^{k+1} + (\hat{w}_k - b) b^k + b^{k+1} =$$
$$= (\hat{w}_{k+1} + 1) b^{k+1} + (\hat{w}_k - b) b^k$$

La nueva solución es:

$$\tilde{w}_i = \hat{w}_i = x_i + y_i \quad \text{para} \quad i = 0, ..., k-1, k+2, ..., n-1; \quad \tilde{w}_n = 0;$$
$$\tilde{w}_k = \hat{w}_k - b = x_k + y_k - b \text{ y}$$
$$\tilde{w}_{k+1} = \hat{w}_{k+1} + 1 = x_{k+1} + y_{k+1} + 1.$$

Esto hace que la nueva solución cumpla la EQ (1) y que el nuevo dígito k cumpla también la EQ (2). El valor que se suma al dígito k+1 se denomina **acarreo**, o en ingles *carry*, y se denota con la letra c_{k+1}. A continuación encontramos otra solución en la que, por ejemplo, el dígito k = 1 cumple con la EQ (2).

		Dígito 4	**Dígito 3**	**Dígito 2**	**Dígito 1**	**Dígito 0**
$\hat{W}$		0	14	5	18	12
$\tilde{W}$		0	14	6	8	12

Fase 2. Consiste en aplicar la transformación que acabamos de ver, empezando por el dígito de menor peso (k = 0) hasta llegar al de más peso (k = n-1), siempre que el valor del dígito resulte mayor o igual a la base, b. De esta forma obtenemos la solución que cumple la EQ (1) y EQ (2) a la vez. La siguiente tabla muestra el algoritmo de suma en dos fases para el ejemplo anterior, detallando cada paso de la segunda fase en una fila distinta (desde k = 0 a k = n-1).

		Dígito 4	**Dígito 3**	**Dígito 2**	**Dígito 1**	**Dígito 0**
	X		6	4	9	3
	Y		8	1	9	9
$\hat{W}$		0	14	5	18	12
W	k =0	0	14	5	19	2
	k =1	0	14	6	9	2
	k = 2	0	14	6	9	2
	k = 3	1	4	6	9	2

Es interesante ver que si la suma de los dos dígitos de la posición k más el posible acarreo que viene de la posición anterior es mayor o igual que la base, $x_k + y_k + c_k \geq b$, siempre se puede convertir $x_k + y_k + c_k$ en un dígito restandole el valor b. Esto es así, porque el máximo valor posible de un dígito de entrada es b-1 y el máximo valor del acarreo es 1, por lo que el máximo valor de $x_k + y_k + c_k$ es 2b-1 (como en el dígito 1 del ejemplo). Así, al restarle b a este valor da b-1, que es un dígito válido, pues es menor que b.

Hemos usado este algoritmo en dos fases para entender la suma. Pero nosotros ya sabíamos sumar en decimal con un algoritmo con una sola fase de n pasos, en la que se mezclan las dos fases anteriores. Expresamos el algoritmo de suma en n pasos (k = 0, ..., n-1) en un pseudolenguaje de alto nivel.

4.1.1

Algoritmo de suma con propagación del acarreo en el sistema convencional en base b. Para sumar en binario particularizar para b = 2. Dados los n dígitos de X e Y (x_i e y_i para i = 0, ..., n-1) el siguiente algoritmo obtiene los n+1 dígitos de W (w_i para i = 0, ..., n-1) tales que $W_u = X_u + Y_u$. (los operadores + y - son la suma y la resta aritméticas sobre números naturales).

```
c0 = 0;
for (k = 0; k < n; k = k+ 1){
   wk = xk + yk + ck;
   ck+1 = 0;
   if (wk >= b) {
      wk = wk – b;
      ck+1 = 1;
   }
}
wn = cn;
```

En la tabla vemos los 4 pasos de la suma con propagación del acarreo para el ejemplo anterior.

		Dígito 4	**Dígito 3**	**Dígito 2**	**Dígito 1**	**Dígito 0**
	X		6	4	9	3
	Y		8	1	9	9
k =0	w_0					2
	c_1				1	
k =1	w_1				9	
	c_2			1		
k = 2	w_2			6		
	c_3		0			
k = 3	w_3		4			
	c_4	1				
	W	1	4	6	9	2

En la figura 4.1a se muestra una tabla sencilla para aplicar el algoritmo de suma, que es equivalente a la anterior pero ahora se han juntado las n filas de los acarreos (de k=0 a n-1) en una única fila. Esta representación es más cercana a la que solemos usar al sumar con lápiz y papel (figura 4.1b).

a)

	6	**4**	**9**	**3**
+	**8**	**1**	**9**	**9**
1	0	1	1	0
1	4	6	9	2

b)

	6	**4**	**9**	**3**
+	**8**	**1**	**9**	**9**
1	4	6	9	2

Fig. 4.1 Tablas sencillas para la aplicación del algoritmo de suma con propagación del acarreo en decimal. a) mostrando los acarreos y b) sin mostrarlos

Ejemplo 1

Aplicamos el algoritmo aritmético de suma con propagación del acarreo en dos casos: a) para realizar la suma de dos números representados en hexadecimal por los vectores de dígitos X=0xB92C e Y=0x69E1 y b) para binario con X=1011 e Y=1001. Las tablas de la figura 4.2 muestran la solución (sin mostrar los acarreos).

a)

	B	**9**	**2**	**C**
+	**6**	**5**	**E**	**1**
1	1	F	0	D

b)

	1	**0**	**1**	**1**
+	**1**	**0**	**0**	**1**
1	0	1	0	0

Fig. 4.2 Resultado del ejemplo 4.

Ejemplo 2

En la figura 4.3 se ve el resultado de aplicar el algoritmo de suma en binario para cuatro casos.

✍ 1

a)	b)	c)	d)
1 1 1 0 1 0 0 0	**1 1 1 0 1 0 0 1**	**0 1 1 0 1 1 0 0**	**0 0 1 0 1 0 0 0**
+ 0 1 1 0 1 0 0 1	**+ 0 0 1 1 1 0 0 0**	**+ 0 0 0 0 1 1 1 1**	**+ 0 1 0 1 0 0 1 0**
1 0 1 0 1 0 0 0 1	1 0 0 1 0 0 0 0 1	0 1 1 1 1 0 1 1	0 1 1 1 1 0 1 0

Fig. 4.3 Suma en binario para cuatro casos concretos.

4.2.2 Implementación del sumador binario

Full-Adder

Particularizando el algoritmo de la suma en propagación del acarreo para el caso binario, $b = 2$, podemos observar que el cálculo de los bits c_{k+1} y w_k que se hace dentro del cuerpo del bucle, en cada una de las n iteraciones, es función de solamente tres bits: x_k, y_k y c_k. Así pues, podemos implementar un dispositivo combinacional, denominado Full-adder, con tres bits de entrada y dos de salida, que realiza la función del cuerpo del bucle del algoritmo de suma. En la izquierda de la figura 4.4 hemos escrito el cuerpo del bucle para binario y en la derecha hemos rellenado la tabla de verdad del Full-adder, siguiendo el algoritmo para las 8 posibles combinaciones de los tres bits de entrada.

Algoritmo:

```
w_k = x_k + y_k + c_k ;
c_{k+1} = 0 ;
if (w_k >= 2) {
    w_k = w_k - 2 ;
    c_{k+1} = 1 ;
}
```

Tabla de verdad del Full-adder

x_k	y_k	c_k	c_{k+1}	w_k
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fig. 4.4 Algoritmo y tabla de verdad del Full-adder.

A continuación se muestran los 8 posibles valores de tres bits y sus sumas en binario. Esto es lo que hace el Full-adder: suma los tres bits de entrada (cuenta el número de unos que hay en sus tres entradas)

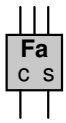
y codifica en binario el resultado en el vector de dos bits de su salida (con dos bits es suficiente para codificar el valor máximo de la suma, 3).

x_k	0	0	0	0	1	1	1	1
y_k	0	0	1	1	0	0	1	1
+ c_k	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1	+ 0	+ 1
c_{k+1} w_k	0 0	0 1	0 1	1 0	0 1	1 0	1 0	1 1

A continuación definimos el Full-adder de dos formas diferentes y, como es habitual, eliminamos el subíndice k de los nombres de las señales y denominamos las salidas c y s (del ingles *Carry* y *Sum*).

4.2.2

Full-adder. El Full-adder, **Fa**, es un dispositivo combinacional con tres entradas (x, y, z) y dos salidas (c, s). El vector de salida W = (c, s) codifica en binario el número de entradas con valor igual a 1, esto es:



$$W_u = X_u + Y_u + Z_u = x + y + z$$

con $W_u = c \times 2^1 + s \times 2^0$ y siendo los operadores + y x la suma y el producto de la aritmética sobre números naturales. Ya que las tres entradas del Fa son conmutativas (a nivel aritmético y lógico) no las etiquetamos en el símbolo del Fa, como se muestra en la figura, pero sí que etiquetamos las salidas c y s, que no son intercambiables. Otra forma de definir el bloque Fa es mediante su tabla de verdad, que se ve en la figura.

x y z	c s
0 0 0	0 0
0 0 1	0 1
0 1 0	0 1
0 1 1	1 0
1 0 0	0 1
1 0 1	1 0
1 1 0	1 0
1 1 1	1 1

✍ 2

Sumador binario con Full-adders en propagación del acarreo

Si denotamos a la función que implementa el Full-adder como $(c_{k+1}, w_k) = Fa(x_k, y_k, c_k)$ podemos escribir el algoritmo de suma en binario como sigue.

```
c0 = 0;
for (k = 0; k < n; k = k + 1) {
   (ck+1, wk) = Fa(xk, yk, ck);
}
wn = cn;
```

La implementación combinacional de un algoritmo iterativo como este se puede obtener *desenrollando totalmente el bucle* (escribiendo un algoritmo equivalente pero sin la sentencia `for`: replicando n veces el cuerpo del bucle). Esto se ve en la izquierda de la figura 4.5 para n = 4.

```
c0 = 0;
(c1, w0) = Fa(x0, y0, c0);
(c2, w1) = Fa(x1, y1, c1);
(c3, w2) = Fa(x2, y2, c2);
(c4, w3) = Fa(x3, y3, c3);
w4 = c4;
```

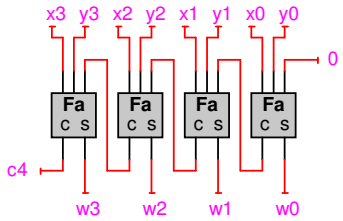


Fig. 4.5 Algoritmo de suma con Full-adders e implementación combinacional (n=4).

Ahora solamente tenemos que disponer de n Full-adders (tantos como iteraciones tenía el algoritmo, en nuestro ejemplo n = 4) y conectarlos entre sí como indica el algoritmo desenrollado que acabamos de obtener. Ya que c_0 vale 0, el primer Full-adder podría ser otro dispositivo más sencillo que sumara solamente 2 bits, pero dejamos un Full-adder con una entrada a 0 para tener un diseño más homogéneo.

¿Cuándo el resultado de la suma no es representable con n bits?

La salida w_n del resultado la hemos denominado c_n (esto es correcto ya que $w_n = c_n$). No la hemos nombrado w_n porque en los computadores generalmente los operandos y el resultado tienen el mismo número de bits que coinciden con el número de bits de los dispositivos de almacenamiento (registros) de donde se leen los operandos y donde se escribe el resultado. En este caso lo más razonable es almacenar los n bits de menor peso, que nos dan el resultado correcto de la suma de dos números de n bits en muchos casos, pero no en todos. ¿Cómo saber en qué casos los n bits de menor peso no representan el resultado correcto?

Esto es lo mismo que preguntarse cuándo el vector $W = w_{n-1}w_{n-2}\cdots w_2w_1w_0$ no representa en binario el mismo valor que el vector $W = w_nw_{n-1}w_{n-2}\cdots w_2w_1w_0$. La respuesta es: cuando w_n vale 1, o lo que es lo mismo, cuando c_n vale 1. En estos casos se dice que se ha producido desbordamiento al sumar en binario, que quiere decir que el resultado de la suma de dos números binarios de n bits no puede representarse con n bits (hacen falta n+1). La salida c_n del sumador binario nos indica cuando vale 1 que el resultado correcto de la suma no puede representarse con n bits y por lo tanto que $W = w_{n-1}w_{n-2}\cdots w_2w_1w_0$ no representa el resultado correcto.

Bloque ADD

Cuando diseñemos otros circuitos para hacer otras operaciones aritméticas que incluyan sumas, puede ser interesante disponer de un símbolo para el bloque sumador binario de dos números de n bits (si no se dice lo contrario n = 16). Este símbolo, que se muestra en la parte superior derecha de la figura 4.6, tiene dos buses de entrada X y Y y una de salida W. Dibujamos los buses como líneas más gruesas que las de los bits (y en azul, cuando se usan colores). Además, en este caso tiene un bit de salida para el acarreo, c_n, que indica cuando vale 1 que el resultado de la suma $X_u + Y_u$ no es representable en n bits (por lo que W_u != X_u+Y_u). Cuando no haga falta usaremos el bloque ADD sin salida de acarreo.

En la figura 4.6 se ve el circuito interno del sumador con salida de acarreo. Obsérvese que los conectores de entrada/salida de buses de 16 bits son más grandes que los conectores de bits y que los bits de todos los buses se denotan siempre como b_{15}, b_{14}, ..., b_1, b_0 (bit 15, bit 14, ..., bit 1, bit 0).

4.3.2

Símbolo y circuito interno del sumador con Full-adders en propagación del acarreo, ADD, con salida de resultado no representable, c (para n=16 bits). .

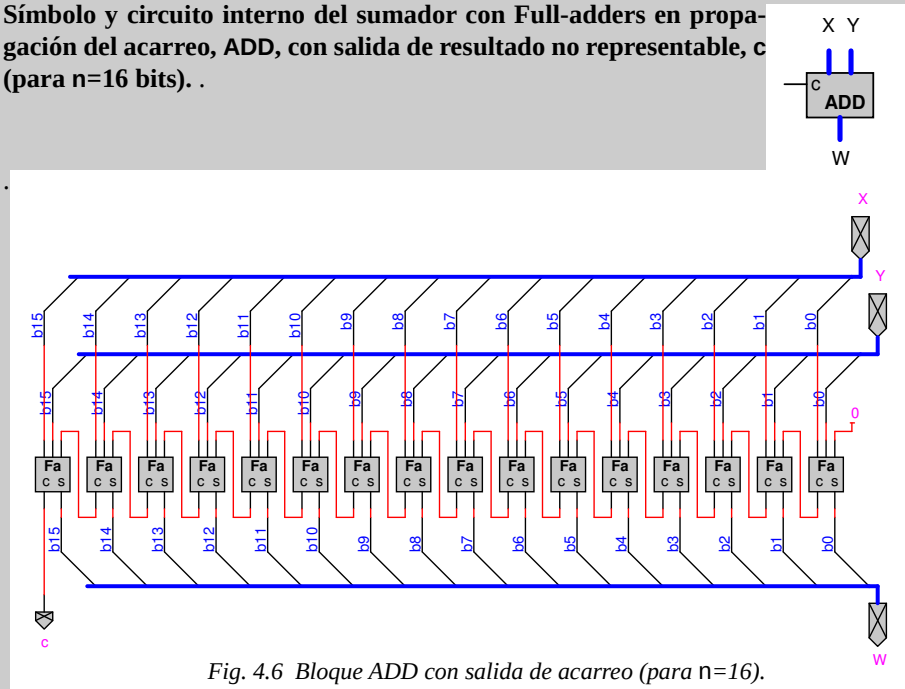


Fig. 4.6 Bloque ADD con salida de acarreo (para n=16).

3

4.2.3 Implementación del Full-adder con Half-adders y estos con una And y una Xor

En decimal sabemos de memoria la suma de dos dígitos y usamos esto dos veces en cada paso del algoritmo, en el que hay que sumar los dos dígitos de los operandos más el posible acarreo del paso anterior. Ya que estamos trabajando el diseño ad-hoc de bloques combinacionales, y aunque el Full-adder tiene pocas entradas y se puede implementar bien en suma de minterms, vamos a diseñar un Full-adder usando varias copias de un dispositivo más simple todavía, que sume solamente dos bits. Este dispositivo se denomina Half-adder y lo definimos e implementamos en suma de minterms antes de usarlo para implementar el Full-adder.

Half-Adder

Un Half-adder es un dispositivo combinacional con dos entradas x e y y dos salidas c y s. El vector que forman los dos bits de salida, W = (c, s) codifica en binario un número natural que es igual al número de entradas que valen 1. Dicho de otra forma, W codifica en binario la suma aritmética de los dos bits de entrada, considerando que los 2 bits tienen el mismo peso (peso 1 =2^0):

$$W_u = x + y$$

Donde $W_u = c \times 2^1 + s \times 2^0$ y el operador + es la suma aritmética.

Para implementar el circuito Half-adder primero especificamos el valor de las salidas del circuito para cada uno de los posibles valores de las entradas y ordenamos esta información en la tabla de verdad. Podemos ver el Half-adder como un circuito que suma dos números de un bit cada uno):

x	**0**	**0**	**1**	**1**
+ y	**+ 0**	**+ 1**	**+ 0**	**+ 1**
c s	0 0	0 1	0 1	1 0

La tabla de verdad donde se ha colocando adecuadamente esta información se ve en la izquierda de la figura 4.7. A partir de la tabla de verdad sabemos construir el Half-adder con puertas Not, And y Or mediante la expresión en suma de minterms de cada una de sus dos salidas. No obstante, para simplificar el dibujo del esquema del circuito interno usamos una puerta Xor-2 para implementar la salida s (hemos sintetizado la Xor-2 en suma de minterms en el capítulo 2). El símbolo del dispositivo Half-adder y su esquema interno se muestran a la derecha de la figura 4.7.

4.2.1

Half-adder: tabla de verdad, símbolo y circuito interno

4

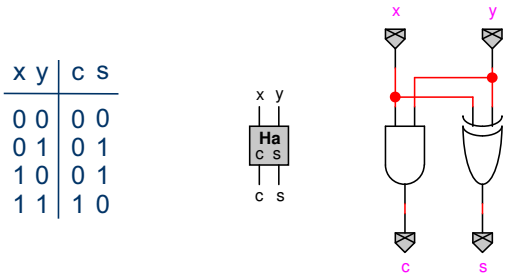


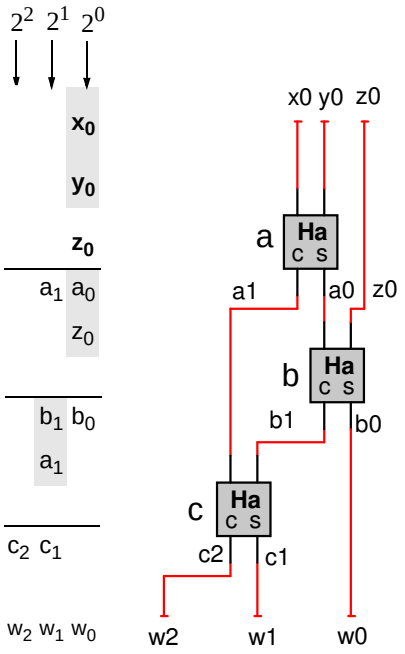
Fig. 4.7 Tabla de verdad, símbolo y circuito interno del Half-adder.

Diseño del Full-adder con Half-adders

Sabemos que un Half-adder suma dos bits del mismo peso y que un Full-adder suma tres. ¿Sabríamos sumar tres bits haciendo varias sumas de dos bits cada una? Si sabéis hacerlo con lápiz y papel, ¡también podéis implementar un circuito que lo haga! La derecha de la figura 4.8 muestra el esquema de interconexión en tres niveles de half-adders (denominados a, b y c) para sumar tres bits x_0, y_0, z_0 (el subíndice 0 indica que los tres tienen el mismo peso: 2^0). A la izquierda se han dispuesto los tres bits de

entrada en la misma columna, como cuando sumamos a mano. Los bits x_0 e y_0 entran en el Half-adder a. Por ello, estos bits se muestran a la izquierda con un fondo gris (el z_0 no se ha sombreado pues no entra en el half-adder de este nivel, solamente pasa de la entrada al siguiente nivel de suma: Half-adder b). La suma de x_0 e y_0 produce el acarreo, a_1, y el bit de suma, a_0, (que se muestran a la izquierda debajo de la primera línea horizontal que separa el primer nivel de sumas del siguiente). El a_1 se ha dibujado en la siguiente columna hacia la izquierda, que representa a los bits de peso 2^1. Al Half-adder b le entran los bits a_0 y el z_0 (sombreados en el segundo nivel de suma) y salen los bits b_1 y b_0. El esquema lógico y la gráfica con los bits encolumnados según el peso, continúa hasta que se suman todos los bits: sumando siempre juntos los bits del mismo peso. La tabla de verdad de la derecha de la figura 4.8 muestra el comportamiento de las variables internas y de salida del circuito.

Optimización. La tabla de verdad nos dice lo que ya podíamos saber de antemano, que la salida w_2 vale siempre 0, ya que al sumar tres bits el valor máximo es 3 y el 3 se puede codificar con sólo dos bits: 11. Así que del Half-adder c solamente necesitamos la salida c_1 que se implementa internamente con una Xor-2. Pero si nos fijamos, aun podemos simplificar más el diseño. Se puede ver en la tabla de verdad, y también pensando en lo que estamos haciendo, que los bits a_1 y b_1 nunca valen los dos 1 a la vez: o valen los dos 0 o vale uno 0 y el otro 1. Por ello, la Xor-2 (que requiere internamente dos Not, dos And-2 y una Or-2) que produce c_1, se puede sustituir por una sola puerta Or-2. La figura 4.9 muestra el símbolo y el esquema interno del Full-adder con dos Half-adders y una puerta Or-2, y con cada Half-adder implementado con una And-2 y una Xor-2.



x_0	y_0	z_0	a_1	a_0	b_1	b_0	c_2	c_1	w_2	w_1	w_0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	1
0	1	0	0	1	0	1	0	0	0	0	1
0	1	1	0	1	1	0	0	1	0	1	0
1	0	0	0	1	0	1	0	0	0	0	1
1	0	1	0	1	1	0	0	1	0	1	0
1	1	0	1	0	0	0	0	1	0	1	0
1	1	1	1	0	0	1	0	1	0	1	1

Fig. 4.8 Full-adder construido con tres Half-adders. A la izquierda se muestran los bits que entran y salen de los Half-adders encolumnados según su peso.

4.2.2

Full-adder construido con dos Half-adder y una Or-2

5

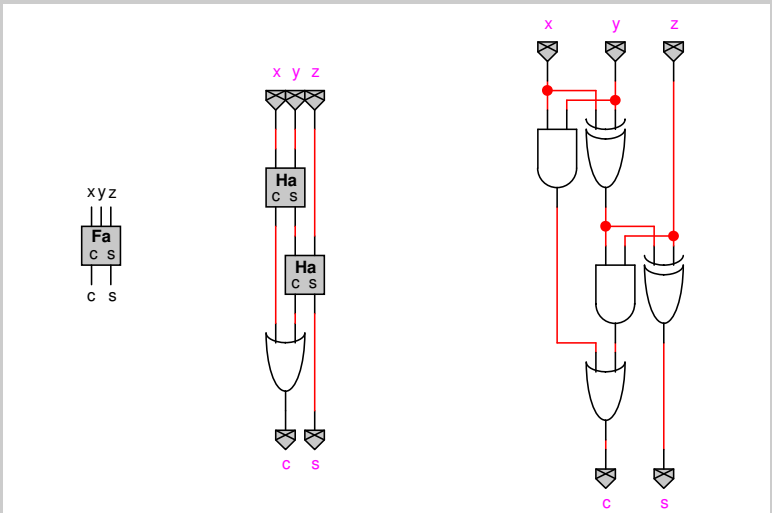


Fig. 4.9 Full-adder con dos Half-adders y una Or-2.

¿Qué implementación es mejor en suma de minterms o con dos Half-adders y una Or? Todas las implementaciones tienen el mismo comportamiento lógico (la misma tabla de verdad) pero pueden tener distinto comportamiento en cuanto a tiempos de propagación, número de puertas y tipos de puertas utilizadas. Dejamos al lector esta comparación y su repercusiones en el diseño del sumador con Full-adders en propagación del acarreo.

4.3 El restador binario

La resta es muy parecida a la suma, así que vamos a seguir los mismos pasos pero particularizando para el caso binario y con menos detalle.

4.3.1 Algoritmo aritmético de la resta en binario

Queremos obtener la representación de $W_u = X_u - Y_u$ a partir de las representaciones de X_u e Y_u. Si los operandos están representados con n bits, $0 \le X_u, Y_u \le 2^n - 1$, el rango del resultado de la resta es $-(2^n - 1) \le W_u \le 2^n - 1$. Sin embargo, en el sistema convencional en base 2 sólo podemos representar números naturales, así que, si el resultado es negativo no lo podremos representar. Esto es ligeramente

diferente al caso de la suma con n bits en la que el resultado correcto siempre se puede representar con n+1 bits. Aun así, como en un computador solo almacenamos los n bits de menor peso del resultado, sabemos detectar cuándo el resultado no es representable en n bits. Ahora, en la resta, también encontraremos una regla para saber cuándo el resultado no es representable porque es negativo (no es un número natural) y por eso no se puede representar en el sistema convencional en base 2 (binario), ni siquiera usando un número ilimitado de bits.

Planteamiento del problema de la resta binaria

Dados dos vectores de n bits, $X = x_{n-1}x_{n-2}\cdots x_2x_1x_0$ e $Y = y_{n-1}y_{n-2}\cdots y_2y_1y_0$ con $x_i, y_i \in \{0, 1\}$ para i = 0, ..., n, que representan dos números naturales X_u e Y_u en el sistema convencional en base 2, encontrar los n bits del vector $W = w_{n-1}w_{n-2}\cdots w_2w_1w_0$ tales que $W_u = X_u - Y_u$,

$$\sum_{i=0}^{n-1} w_i 2^i = \sum_{i=0}^{n-1} x_i 2^i - \sum_{i=0}^{n-1} y_i 2^i \qquad \textbf{(EQ 3)}$$

y que los valores w_i sean dígitos válidos en base 2, esto es:

$$0 \le w_i \le 1 \qquad \textbf{(EQ 4)}$$

Además, hay que detectar cuándo no existe solución a las ecuaciones anteriores (porque el resultado de la resta es negativo): resultado no representable.

Una solución en dos fases al problema de la resta

En la primera fase encontramos una solución a la EQ (3) y en la segunda transformamos en n pasos la solución anterior para que cumpla además la restricción de la EQ (4).

Fase 1. Manipulando la EQ (3) encontramos una solución trivial, de las infinitas que hay que no tienen porqué cumplir la EQ (4), y que la denominamos con un sombrerito:

$$\hat{w}_i = x_i - y_i \qquad \text{para} \qquad 0 \le i \le n-1 \quad \text{y}$$

El resultado de aplicar la primera fase del algoritmo de resta a los vectores de bits X = 1001 e Y = 0011 es el siguiente. El bit 1 de esta solución no es un dígito válido, no es un bit, ya que vale -1.

		bit 3	**bit 2**	**bit 1**	**bit 0**
	X	1	0	0	1
	Y	0	0	1	1
$\hat{W}$		1	0	-1	0

Posible transformación de la solución de la primera fase. La solución que acabamos de encontrar no siempre cumple la EQ (4), ya que en general el valor de los dígitos que hemos encontrado puede ser negativo: $-1 \le \hat{w}_i \le 1$. Si el dígito k que no cumple la EQ (4), $w_k < 0$, una solución en la que al menos el

dígito k cumpla las dos ecuaciones, se obtiene manipulando el sumando del dígito k, y como consecuencia el del k+1, de la expresión $X_u = \sum_{i=0}^{n-1} x_i 2^i$. La manipulación de estos dos sumandos es:

$$\hat{w}_{k+1}2^{k+1} + \hat{w}_k 2^k = \hat{w}_{k+1}2^{k+1} + (\hat{w}_k - 2 + 2)2^k = \hat{w}_{k+1}2^{k+1} + (\hat{w}_k + 2)2^k - 2^{k+1} =$$

$$= (\hat{w}_{k+1} - 1)2^{k+1} + (\hat{w}_k + 2)2^k$$

La nueva solución es igual a la anterior para los dígitos de los sumandos no manipulados:

$$\tilde{w}_i = \hat{w}_i = x_i - y_i \qquad \text{para} \qquad i = 0, ..., k-1, k+2, ..., n-1; \qquad \tilde{w}_n = 0$$

mientras que para el dígito k y k+1 es:

$$\tilde{w}_k = \hat{w}_k + 2 = x_k - y_k + 2$$

$$\tilde{w}_{k+1} = \hat{w}_{k+1} - 1 = x_{k+1} - y_{k+1} - 1$$

Esto hace que la nueva solución siga cumpliendo la EQ (3) y que además el nuevo bit k cumpla también la EQ (4). El valor que se resta al bit k+1 se denomina **débito**, o en ingles ***borrow***, y se denota con la letra b_{k+1}. (aunque a veces no diferenciamos entre suma y resta y lo denominamos también acarreo). A continuación mostramos cómo encontrar la solución del ejemplo anterior para que el bit 1 cumpla con la EQ (4). En este ejemplo hemos arreglado el bit 1 pero ahora es el 2 el que no cumple con la EQ (4). La solución cuando pasa esto es aplicar otra vez al bit k+1 lo que acabamos de hacer al k.

		Bit 3	**Bit 2**	**Bit 1**	**Bit 0**
$\hat{W}$		1	0	-1	0
$\tilde{W}$		1	-1	1	0

Fase 2. La segunda fase del algoritmo de resta consiste en ir aplicando la regla anterior para todos los dígitos que no cumplan con la EQ (4) desde k = 0 hasta n-1 de forma que obtenemos la solución que cumple la EQ (3) y EQ (4) a la vez. No obstante, si resulta que el bit de más peso (bit n-1) es negativo, no podemos hacer nada para que se cumpla la EQ (4). Si aplicáramos la regla de la fase 1 al dígito n-1 nos resultaría que es ahora el dígito n el que es -1, y así hasta el infinito. Esto quiere decir que el resultado de la resta es negativo, no es un número natural y por ello no se puede representar en el sistema convencional en base 2, en binario.

La siguiente tabla muestra el algoritmo de resta en dos fases para el ejemplo anterior.

		Bit 3	**Bit 2**	**Bit 1**	**Bit 0**
	X	1	0	0	1
	Y	0	0	1	1
$\hat{W}$		1	0	-1	0
W	k =0	1	0	-1	0
	k =1	1	-1	1	0
	k = 2	0	1	1	0
	k = 3	0	1	1	0

Es interesante ver que si la resta de los dos dígitos de la posición k menos el *borrow* que viene de la posición anterior es negativa, $x_k - y_k - b_k < 0$, siempre se puede convertir el resultado en un dígito binario si se le suma el número 2, ya que el máximo valor negativo de $x_k - y_k - b_k$ es -2.

Expresamos a continuación el algoritmo de resta en n pasos (k = 0, ..., n-1) juntando las dos fases en cada paso. Es importante resaltar que si como resultado de aplicar el algoritmo, el *borrow* que sale del paso n-1, b_n, es 1 el resultado de la resta no es representable, porque es negativo.

Algoritmo de resta con propagación del débito:

```
b_0 = 0;
for (k = 0; k < n; k = k + 1){
   w_k = x_k - y_k - b_k;
   b_{k+1} = 0;
   if (w_k < 0) {
      w_k = w_k + 2;
      b_{k+1} = 1;
   }
}
```

Otra forma de expresar este algoritmo, que se parece más a cómo restamos en decimal con lápiz y papel, se ve en el recuadro siguiente.

4.1.2

Algoritmo de resta con propagación del débito en binario. Dados los n bits de X e Y (x_i e y_i para i = 0, ..., n-1) el siguiente algoritmo obtiene los n bits de W (w_i para i = 0, ..., n-1) tales que $W_u = X_u - Y_u$ cuando este resultado es positivo (los operadores + y - son la suma y la resta aritméticas sobre números naturales). Si el débito final, b_n, vale 1 indica que el resultado no es representable, ya que es negativo y al no ser un número natural no se puede representar en el sistema convencional en base 2.

$b_0 = 0$;
for (k = 0; k < n; k = k + 1){
  if ($y_k + b_k \leq x_k$) **{**
    $w_k = x_k - (y_k + b_k)$;
    $b_{k+1} = 0$;
  } else {
    $w_k = 2 + x_k - (y_k + b_k)$;
    $b_{k+1} = 1$;
  }
}

Ejemplo 3

Aplicamos el algoritmo aritmético de resta en binario que acabamos de encontrar a los cuatro ejemplos que se muestran en la figura 4.3 para n = 8 bits (los valores del borrow no se muestran). Sólo en el caso d) los n bits del resultado no expresan el resultado correcto de la resta, ya que el acarreo (mejor débito) de salida, b_8, es 1.

✍ 6

a)	b)	c)	d)
1 1 1 0 1 0 0 0	**1 1 1 0 1 0 0 1**	**0 1 1 0 1 1 0 0**	**0 0 1 0 1 0 0 0**
- 0 1 1 0 1 0 0 1	**- 0 0 1 1 1 0 0 0**	**- 0 0 0 0 1 1 1 1**	**- 0 1 0 1 0 0 1 0**
0 1 1 1 1 1 1 1	1 0 1 1 0 0 0 1	0 1 0 1 1 1 0 1	1 1 0 1 0 1 1 0
			b_8 =1
			No representable

Fig. 4.10 Tablas sencillas resultado de aplicar el algoritmo aritmético de suma en binario para cuatro casos concretos.

4.3.2 Implementación del restador binario

Como hicimos con el sumador, vamos a implementar el dispositivo combinacional que realiza las sentencias del cuerpo del bucle `for` (`if...else...`) del algoritmo de resta. El algoritmo de este dispositivo, que se denomina Full-subtractor, Fs, se reproduce a continuación:

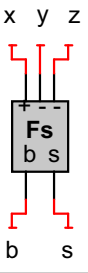
```
wk = xk − yk − bk ;
bk+1 = 0 ;
if (wk < 0) {
    wk = wk + 2 ;
    bk+1 = 1 ;
}
```

El símbolo del Full-subtractor se muestra en el recuadro siguiente, junto con los nombres usados para sus entradas y salidas (como es usual quitamos el subíndice k del algoritmo y llamamos z a la entrada b_k y s (*subtract*) a la salida w_k). Como las entradas no son intercambiables se han etiquetado con los símbolos + y -. También se muestra la tabla de verdad del Fs, que se obtiene siguiendo el código anterior para las 8 posibles combinaciones de los tres bits de entrada (x_k, y_k y b_k: en la tabla x, y y z).

4.2.3

Full-subtractor. El Full-subtractor, **Fs**, es un dispositivo combinacional con tres entradas (x, y, z) y dos salidas (b, s) (cuya función la denotamos (b, s) = Fs(x, y, z)) que tiene la siguiente tabla de verdad.

x y z	b s
0 0 0	0 0
0 0 1	1 1
0 1 0	1 1
0 1 1	1 0
1 0 0	0 1
1 0 1	0 0
1 1 0	0 0
1 1 1	1 1



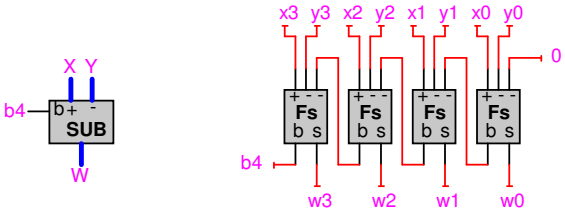
✍ 7

Restador binario con Full-subtractors en propagación del débito

El esquema lógico del restador binario con Full-subtractors tiene la misma estructura que el sumador binario (ver recuadro). Cuando Xu-Yu es negativo no se puede representar en el sistema convencional en base 2. Esto se indica poniendo a 1 el débito que sale del Fs de más peso.

4.3.3

Símbolo y circuito del restador con Full-subtractors en propagación del débito.



✍ 8

4.4 Desplazador de k bits a la izquierda (multiplicador por 2^k)

Vemos primero el algoritmo aritmético del caso particular de multiplicación por 2, luego el general de multiplicación por potencias de 2 y por último su implementación en hardware.

4.4.1 Algoritmo aritmético de multiplicación por 2 en binario.

A partir de la representación en binario con n bits, $X = x_{n-1}x_{n-2}\cdots x_2x_1x_0$, de un número natural, X_u, encontrar la representación del número $W_u = 2 \cdot X_u$. Dado que el rango de X_u es $0 \le X_u \le 2^n - 1$, el de W_u es $0 \le X_u \le 2^{n+1} - 2$ y por lo tanto se requieren, en el peor de los casos, n+1 bits para representarlo. Encontrar el algoritmo aritmético de la multiplicación por 2 es equivalente a encontrar los bits de $W = w_n w_{n-1} \cdots w_2 w_1 w_0$ tales que:

$$\sum_{i=0}^{n} w_i 2^i = 2\left(\sum_{i=0}^{n-1} x_i 2^i\right) \qquad \textbf{(EQ 5)}$$

y que los valores w_i sean dígitos válidos en base 2, esto es:

$$0 \le w_i \le 1 \qquad \textbf{(EQ 6)}$$

Como $2x2^i = 2^{i+1}$, la EQ (5) expandida, para que se vea más claro, y después de multiplicar todos los sumandos del sumatorio por 2, queda:

$$w_n 2^n + w_{n-1} 2^{n-1} + \cdots + w_2 2^2 + w_1 2^1 + w_0 2^0 = x_{n-1} 2^n + x_{n-2} 2^{n-1} + \cdots + x_1 2^2 + x_0 2^1$$

De aquí se ve que la siguiente solución

$$w_0 = 0 \quad \text{y}$$

$$w_i = x_{i-1} \qquad \text{para} \qquad 1 \le i \le n,$$

cumple con EQ (5) y EQ (6), ya que los dígitos de X son bits y w_0=0 también.

> *Por lo tanto, multiplicar por 2 un número natural representado en binario consiste en desplazar los bits de la representación una posición a la izquierda y poner un 0 como bit de menor peso.*

Obsérvese que esto es equivalente a lo que hacemos en decimal cuando multiplicamos un número por 10. Así que este es el algoritmo para multiplicar por la base b un número representado en el sistema convencional en base b.

Ejemplo 4

La figura 4.11 muestra un ejemplo de aplicación del algoritmo aritmético de la multiplicación por 2 en binario.

Valor	**Representación en binario**				
	Bit 4	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
X_u=9		1	0	0	1
$2X_u$=18	1	0	0	1	0

Fig. 4.11 Ejemplo de aplicación de multiplicación por 2 en binario.

Detección de resultado no representable en n bits. Como es habitual en un computador de n bits, sólo podemos almacenar los n bits de menor peso del resultado de la multiplicación por 2, que requiere, en el peor de los casos n+1 bits. ¿Cómo sabemos si los n bits de menor peso representan correctamente al resultado? La respuesta es trivial: si el bit n del resultado es igual a 1 (W_n=1), o lo que es lo mismo, si el bit n-1 del operando es igual a 1 (X_{n-1}=1) el resultado de la multiplicación por 2 no es representable en n bits.

4.4.2 Algoritmo aritmético de multiplicación por potencias de 2

Dado que multiplicar un número por 2^k consiste en multiplicarlo por 2 repetidamente k veces, el algoritmo aritmético para multiplicar por 2^k consiste en aplicar el algoritmo de multiplicación por dos k veces consecutivas.

4.1.3

Algoritmo para obtener la representación de X_u2^k a partir de X. Dado $X = x_{n-1}x_{n-2}\cdots x_2x_1x_0$ se obtiene $W = w_{n+k-1}w_{n+k}\cdots w_2w_1w_0$ tal que $W_u = X_u2^k$, asignando los n bits de X a los n bits de más peso de W (desplazando k posiciones a la izquierda los bits de X) y asignando k ceros a los k bits de menor peso de W.

Además, el resultado de la operación, X_u2^k, no es representable con solo los n bits de menor peso de W cuando alguno de los k bits de más peso de X es distinto de 0.

Ejemplo 5

En la siguiente tabla se muestra la representación en binario con 8 bits del número 23 y sus multiplicaciones por 2^k para k = 1, 3 y 5. Es de resaltar que $23x2^5$ requiere 10 bits para representarse correctamente en binario: 1011100000, por lo que los 8 bits de menor peso, 11100000,

no dan el resultado correcto. El resultado no es representable en 8 bits porque en los k=5 bits de más peso del operando (00010111) hay al menos un 1.

✍ 9

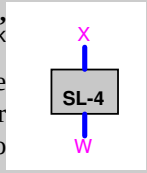
Valor:	$X_u = 23$	$2X_u = 46$	$2^3X_u = 184$	$2^5X_u = 736$
Representación con n = 8 bits:	00010111	00101110	10111000	11100000 No representable

4.4.3 Implementación del multiplicador por 2^k (SL-k)

La figura 4.12 muestra el esquema lógico del circuito combinacional para multiplicar un número natural representado en binario con 16 bits por 2^k, para k=4, que implementa en hardware el algoritmo de la sección anterior. Para formar los 16 bits el resultado no se necesitan puertas, sólo hay que conectar los cables de la entrada a la salida desplazados k posiciones a la izquierda y poner a 0 los k bits de menor peso. Si queremos saber cuándo los 16 bits de salida no representan el resultado correcto, es necesaria una puerta Or-k que detecte si hay algún 1 entre los k bits de mayor peso de la entradas. Este dispositivo que multiplica por 2^k se denomina, SL-k (Shift Left - k): desplazador de k bits a la izquierda.

☞ 4.4.1

Símbolo y circuito interno del desplazador de k bits a la izquierda, SL-k (Shift Left- k), para k = 4. Implementa la operación $W_u = X_u 2^k$ siempre que los k bits de mayor peso de X sean 0. El símbolo no tiene salida de resultado no representable con n bits ya que en el computador que diseñaremos no la utilizaremos. No obstante, en el esquema interno si se ha implementado con una puerta Or.



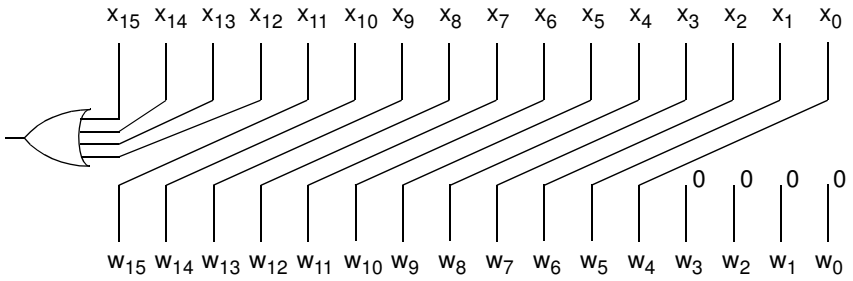


Fig. 4.12 Símbolo del desplazador de 4 bits a la izquierda (Shift Left - 4) y esquema interno para n=16. El símbolo no tiene la salida de resultado no representable.

✍ 10

4.5 Desplazador de k bits a la derecha (divisor por 2^k)

Vemos primero el algoritmo aritmético del caso particular de división por 2, luego el de división por potencias de 2 y por último su implementación en hardware. Como trabajamos con números naturales, siempre que hablemos de división en este capítulo, nos referimos a la división entera, sin decimales (parte entera por abajo). Esta operación es muy parecida a la multiplicación por potencias de 2 ya que si bien $2^i x 2^k = 2^{i+k}$, resulta que $2^i/2^k = 2^{i-k}$. La diferencia entre sumar k al exponente del 2 o restarlo hace que lo que en la multiplicación supone desplazar k bits a la **izquierda** poniendo los k bits de **menor peso** a cero, en la división suponga desplazar a la **derecha** y poner ceros en los bits de **mayor peso**.

4.5.1 Algoritmo aritmético de división por 2 en binario.

A partir de la representación en binario con n bits, $X = x_{n-1}x_{n-2}\cdots x_2x_1x_0$, de un número natural, X_u, encontrar la representación del número $W_u = X_u/2$. Dado que el rango de X_u es $0 \le X_u \le 2^n - 1$, el de W_u es $0 \le X_u \le 2^{n-2} - 1$ y por lo tanto se requieren, en el peor de los casos, n-1 bits para representarlo. Como en el computador se representan con n bits (n=16 en nuestro caso) tanto el operando como el resultado de la operación, en el caso de la división por 2 el resultado siempre es representable, no tiene sentido hablar de detección de resultado no representable. Así que, encontrar el algoritmo aritmético de la división por 2 es equivalente a encontrar los bits de $W = w_{n-1}w_{n-2}\cdots w_2w_1w_0$ tales que:

$$\sum_{i=0}^{n-1} w_i 2^i = \left(\sum_{i=0}^{n-1} x_i 2^i \right) / 2 \qquad \textbf{(EQ 7)}$$

y que los valores w_i sean dígitos válidos en base 2, esto es: $0 \le w_i \le 1$ **(EQ 8)**

Como $2^i/2 = 2^{i-1}$, la EQ (7) expandida y después de dividir todos los sumandos del sumatorio por 2, y recordando que $x_0/2 = 0$, queda:

$$w_{n-1}2^{n-1} + w_{n-2}2^{n-2} + \cdots + w_2 2^2 + w_1 2^1 + w_0 2^0 = x_{n-1}2^{n-2} + x_{n-2}2^{n-3} + \cdots + x_2 2^1 + x_1 2^0$$

De aquí se ve que la siguiente solución

$$w_i = x_{i+1} \qquad \text{para} \qquad 0 \le i \le n-2 \text{ y}$$

$$w_{n-1} = 0$$

cumple con EQ (7) y EQ (8), ya que los dígitos de X son bits y $w_{n-1}=0$ también.

Por lo tanto, dividir por 2 un número natural representado en binario consiste en desplazar los bits de la representación una posición a la derecha y poner un 0 como bit de mayor peso (el bit de menor peso del operando desaparece).

Obsérvese que esto es equivalente a lo que hacemos en decimal cuando realizamos la división por 10 (parte entera por abajo) de un número natural. Así que este es el algoritmo para dividir por la base b un número natural representado en el sistema convencional en base b.

Ejemplo 6

La figura 4.11 muestra un ejemplo de aplicación del algoritmo aritmético de la división por 2 en binario.

Valor		**Representación en binario**			
		Bit 3	**Bit 2**	**Bit 1**	**Bit 0**
$X_u = 9$		1	0	0	1
$X_u / 2 = 4$		0	1	0	0

Fig. 4.13 Ejemplo de aplicación de división por 2 en binario.

4.5.2 Algoritmo aritmético de división por potencias de 2

Dado que dividir un número por 2^k consiste en dividirlo por 2 repetidamente k veces, el algoritmo aritmético para dividir por 2^k consiste en aplicar el algoritmo de división por dos k veces consecutivas.

4.1.4

Algoritmo para obtener la representación de $X_u/2^k$ a partir de X. Dado $X = x_{n-1}x_{n-2}\cdots x_2x_1x_0$ se obtiene $W = w_{n-1}w_{n-2}\cdots w_2w_1w_0$ tal que $W_u = X_u/2^k$, asignando k ceros a los k bits de mayor peso de W y asignado los n-k bits de más peso de X a los n-k bits de menos peso de W (desplazando k posiciones a la derecha los bits de X).

Además, el resultado correcto de la operación, $X_u/2^k$, siempre es representable en los n bits de W.

Ejemplo 7

En la siguiente tabla se muestra la representación en binario con 8 bits del número 23 y sus divisiones por 2^k para k = 1, 3 y 5.

✍ 11

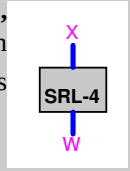
Valor:	$X_u = 23$	$X_u /2 = 11$	$X_u /2^3 = 2$	$X_u /2^5 = 0$
Representación con n = 8 bits:	00010111	00001011	00000010	00000000

4.5.3 Implementación del divisor por 2^k (SRL-k)

La figura 4.14 muestra el esquema lógico del circuito combinacional para dividir un número natural representado en binario con 16 bits por 2^k, para k=4. Para formar los 16 bits del resultado no se necesitan puertas, sólo hay que conectar los cables de la entrada a la salida desplazados k posiciones a la derecha y poner a 0 los k bits de mayor peso. El resultado siempre es representable. Este dispositivo que divide por 2^k se denomina, SRL-k (Shift Right Logically - k): desplazador lógico de k bits a la derecha. Como veremos en el próximo capítulo, cuando estudiemos la representación y los bloques aritméticos de números enteros, la multiplicación por potencias de 2 para enteros es igual que para naturales (excepto la detección de resultado no representable) pero la división por potencias de 2 es distinta para enteros que para naturales. El dispositivo para enteros se denomina SRA-k (*Shift Right Arithmetically* - k). Así que la L final se pone para indicar que se trabaja con naturales y la A para indicar enteros.

4.4.2

Símbolo y circuito interno del desplazador de k bits a la derecha, SRL-k (Shift Rigth Logically - k), para k = 4. Implementa la operación $W_u = X_u/2^k$. El resultado siempre es representable con los mismos bits que el operando.



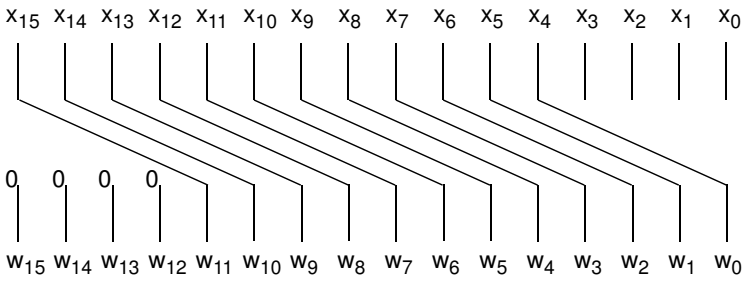


Fig. 4.14 Símbolo del desplazador lógico de 4 bits a la derecha (Shift Right Logicaly - 4) y esquema interno para n=16.

✍ 12

4.6 Comparadores de números naturales

Diseñamos los circuitos combinacionales que denominamos EQ, LTU, y LEU. Tienen dos buses de entrada de n bits, X e Y, y una señal de salida de un bit, w. La funcionalidad de estos circuitos es:

- w = EQ(X,Y), igual (*EQual*):
  `if (X==Y) w=1; else w=0;`
- w = LTU(X,Y), menor que para naturales (*Less Than Unsigned*):
  `if (`X_u`<`Y_u`) w=1; else w=0;`
- w = LEU(Xu,Yu), menor o igual para naturales (*Less or Equal Unsigned*):
  `if (`X_u`<=`Y_u`) w=1; else w=0;`

El comparador de igualdad, compara que dos vectores de bits sean iguales bit a bit, no importa qué representen los vectores, por eso no lleva al final la letra U. Sin embargo, los comparadores de menor que y menor o igual efectúan la comparación considerando que los dos vectores de bits representan a dos números naturales codificados en binario, por eso terminan en U, de *Unsigned*.

Para efectuar estas comparaciones realizamos la resta binaria de X e Y y analizamos el resultado. Si los dos vectores son iguales el resultado de la resta será un vector de n ceros. Si hacemos la Or-n de estos n bits y negamos la salida, esta valdrá 1 solamente cuando los n bits sean 0: es la función EQ que buscamos. La figura 4.15 muestra el símbolo del comparador EQ y su esquema interno usando el restador de la sección 4.3 y el nuevo bloque Zero, cuyo símbolo y esquema también se muestra.

4.6.1

4.5

Símbolo y circuito de los bloques a) comparador de igualdad, EQ, y b) Zero, Z.

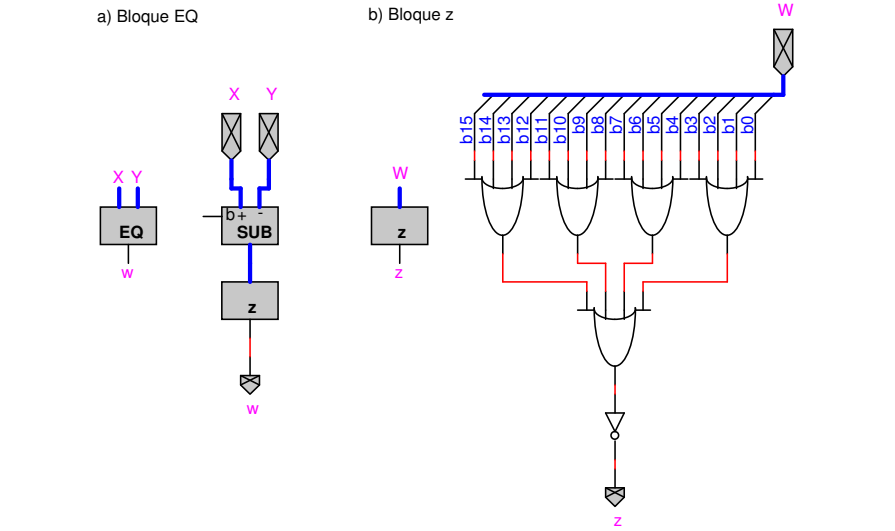


Fig. 4.15 Símbolos y esquemas de a) EQ y b) Z (para n=16).

✍ 13-14

Como vimos en la sección 4.3 al estudiar la resta, la salida b (borrow) del restador binario vale 1 cuando X_u-Y_u es negativo. Esto es lo mismo que decir que cuando b=1 es porque X_u es menor que Y_u. Así que la salida b del restador es directamente la salida del bloque LTU, como se ve en la figura 4.16. A la derecha de esta figura se encuentra el diseño del bloque LEU, que es la Or-2 de las funciones EQ y LTU.

4.6.2

4.6.3

Símbolo y circuito de los bloques a) menor que, LTU, y b) menor o igual, LEU,

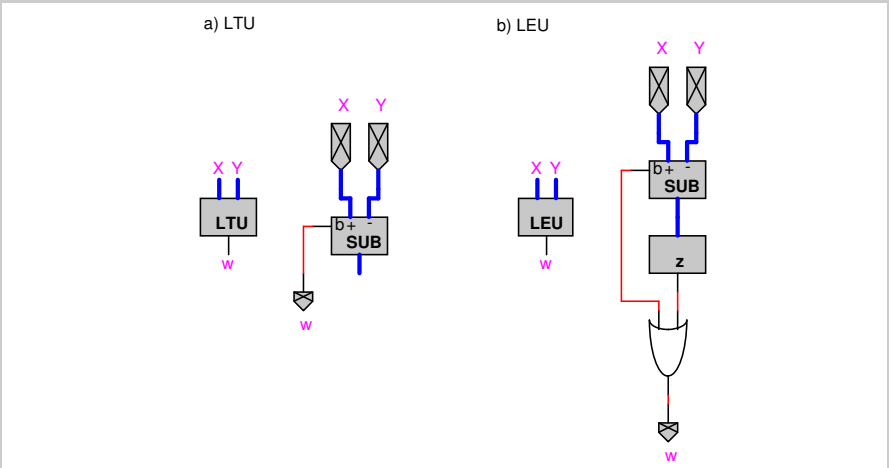


Fig. 4.16 Símbolo de los comparadores y sus esquemas internos: a) LTU y b) LEU.

✍ 15-16

4.7 Operadores lógicos bit a bit

Los operadores lógicos, aunque los estudiamos en este capítulo, no efectúan operaciones aritméticas sobre números naturales, sino que realizan operaciones lógicas bit a bit sobre vectores de bits, que pueden representar cualquier tipo de información. Por ello, en estas operaciones no tiene sentido hablar de resultado no representable. Veamos primero la operación AND, y después resumimos las demás.

4.7.1 AND bit a bit

La operación lógica AND bit a bit la denotamos con todos sus caracteres en mayúsculas para indicar que opera con vectores de n bits. La definición formal es como sigue. Dados dos vectores de n bits, $X = x_{n-1}x_{n-2}\cdots x_2x_1x_0$ e $Y = y_{n-1}y_{n-2}\cdots y_2y_1y_0$ con $x_i, y_i \in \{0, 1\}$ para i = 0,..., n-1, la operación lógica AND bit a bit de X e Y da como resultado el vector de n bits $W = w_{n-1}w_{n-2}\cdots w_2w_1w_0$ tal que,

$$w_i = x_i \cdot y_i \qquad \text{para } i = 0 \text{ hasta } n-1 \qquad \textbf{(EQ 9)}$$

Donde el $\cdot$ representa el producto lógico, la operación lógica And de los dos bits.

Esta operación se denota como W=AND(X,Y) y el símbolo del bloque combinacional que la implementa junto con su esquema lógico interno se muestra en la figura 4.17.

4.7

Símbolo y circuito interno del operador lógico, AND bit a bit.

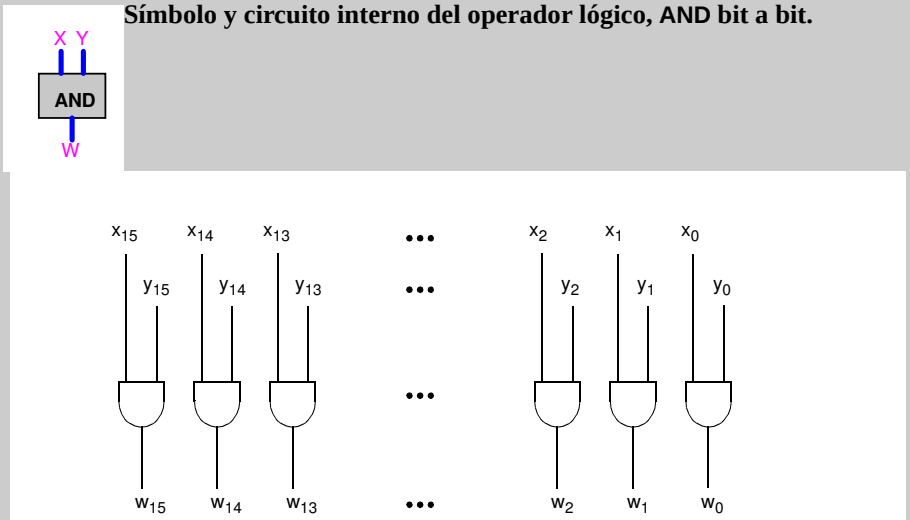


Fig. 4.17 Símbolo del bloque AND y su esquema lógico interno para n=16.

Ejemplo 8

Aplicamos la operación lógica AND bit a bit a cuatro ejemplos que se muestran en la figura 4.3.

a)	b)	c)	d)
1 1 1 0 1 0 0 0	**1 1 1 0 1 0 0 1**	**0 1 1 0 1 1 0 0**	**0 0 1 0 1 0 0 0**
0 1 1 0 1 0 0 1	**0 0 1 1 1 0 0 0**	**0 0 0 0 1 1 1 1**	**0 1 0 1 0 0 1 0**
0 1 1 0 1 0 0 0	0 0 1 0 1 0 0 0	0 0 0 0 1 1 0 0	0 0 0 0 0 0 0 0

Fig. 4.18 Tablas sencillas resultado de aplicar la operación AND a cuatro casos concretos.

4.7.2 OR y XOR y NOT bit a bit

Las operaciones OR y XOR bit a bit son como la AND que acabamos de ver pero sustituyendo la operación/puerta And-2 por la Or-2 o la Xor-2, respectivamente. La operación NOT sólo tiene un operando, el vector de bits X, y da como resultado el W tal que $w_i = !x_i$ para i = 0, ..., n-1.

Ejemplo 9

La figura 4.19 muestra la aplicación a distintos vectores de bits de las operaciones bit a bit AND, OR, XOR y NOT.

a) AND	b) OR	c) XOR	d) NOT
1 1 1 0 1 0 0 0	**1 1 1 0 1 0 0 1**	**0 1 1 0 1 1 0 0**	**0 1 0 1 0 0 1 0**
0 1 1 0 1 0 0 1	**0 0 1 1 1 0 0 0**	**0 0 0 0 1 1 1 1**	1 0 1 0 1 1 0 1
0 1 1 0 1 0 0 0	1 1 1 1 1 0 0 1	0 1 1 0 0 0 1 1	

Fig. 4.19 Tablas sencillas resultado de aplicar la operación a) AND*, b)* OR*, c)* XOR *y d)* NOT *a distintos operandos.*

Los símbolos de los dispositivos combinacionales para realizar las operaciones W=OR(X,Y), W=XOR(X,Y) y W=NOT(X) se muestran en la figura 4.20. No mostramos el esquema interno de estos tres bloques ya que tienen la misma estructura que el AND.

4.7

Símbolos de los operadores lógicos bit a bit: OR, XOR y NOT.

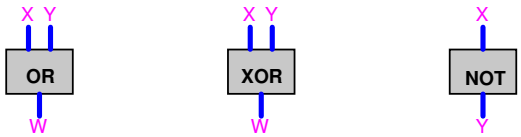


Fig. 4.20 Símbolos de los bloques OR, XOR *y* NOT.

17

4.8 Diseño de multiplexores

Aunque los multiplexores de bits no son bloques combinacionales que procesan números naturales, los vemos en este capítulo porque pueden llegar a tener bastantes bits de entrada y no podemos usar para su diseño el método de síntesis en suma de minterms que vimos en el capítulo 2. Aquí vamos a hacer un diseño ad-hoc de un multiplexor de 8 entradas de datos de un bit cada una, x_7, x_6, ..., x_1 y x_0, tres entradas de selección s_2, s_1, y s_0, y una salida w, que denominamos Mx-8-1. Este ejemplo nos servirá para comprender la estructura del diseño y ser capaces de generalizar la idea para diseñar cualquier otro multiplexor.

4.8.1 Multiplexor Mx-2-1

En primer lugar recordamos la definición del multiplexor Mx-2-1 que vimos en el capítulo 2. Tiene dos entradas de datos x_1 y x_0 y una de selección s y una salida w. El símbolo de este multiplexor y su funcionalidad representada mediante un dibujo, mediante su tabla de verdad y mediante una versión compacta de su tabla de verdad se muestra en la figura 4.21. Este multiplexor tiene tres entradas de un bit y se puede implementar en suma de minterms, como se hizo en el capítulo 2, mediante 3 puertas Not, 4 puertas And-3 para los minterms y una Or-4 para sumar los 4 minterms. Una implementación con menos puertas se muestra en la figura 4.22. La expresión lógica que nos lleva a este diseño se puede conseguir manipulando la expresión en suma de minterms mediante las propiedades algebraicas de las funciones Not, And y Or. No obstante, es fácil hacer un diseño ad-hoc si pensamos en una puerta And-2 como un dispositivo que tiene una entrada de control y la otra de datos. Cuando en la entrada de control hay un 1 deja pasar lo que hay en la entrada de datos a la salida y cuando hay un 0 saca un 0 en la salida. Con esta visión y viendo la funcionalidad del multiplexor con su tabla de verdad compacta, el diseño de la figura 4.22 se entiende fácilmente.

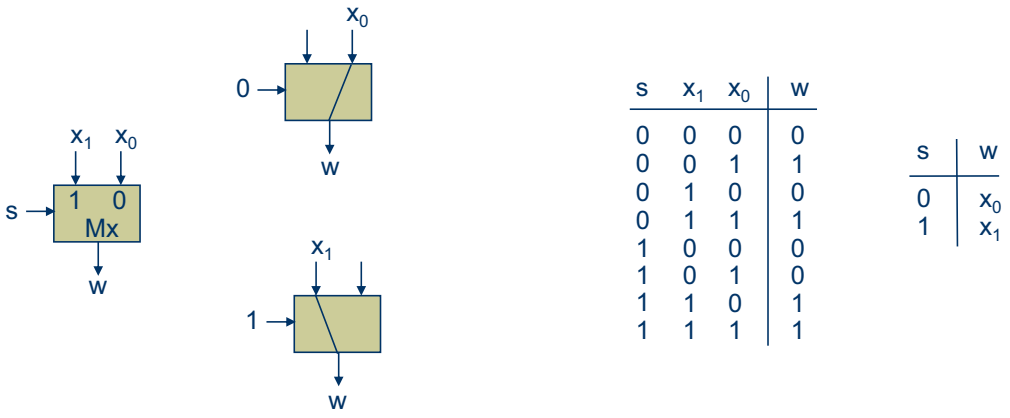


Fig. 4.21 Símbolo del multiplexor Mx-2-1 y funcionalidad descrita mediante un dibujo, su tabla de verdad y su tabla de verdad compacta.

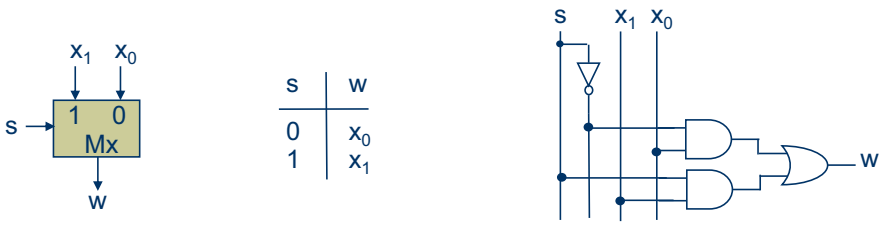
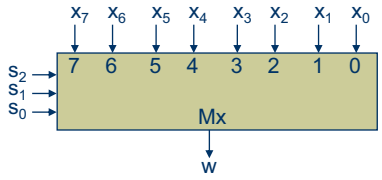


Fig. 4.22 Símbolo del multiplexor Mx-2-1, tabla de verdad compacta e implementación ad-hoc.

4.8.2 Multiplexor Mx-8-1

Vamos a diseñar el Mx-8-1 cuyo símbolo y tabla de verdad compacta se muestra en la figura 4.23.



s_2	s_1	s_0	w
0	0	0	x_0
0	0	1	x_1
0	1	0	x_2
0	1	1	x_3
1	0	0	x_4
1	0	1	x_5
1	1	0	x_6
1	1	1	x_7

Fig. 4.23 Símbolo y tabla de verdad compacta del multiplexor Mx-8-1.

La figura 4.24 muestra el diseño del Mx-8-1 usando multiplexores Mx-2-1. Para comprender el diseño nos tenemos que fijar en la tabla de verdad compacta. La entrada s_2 selecciona entre dos conjuntos de entradas. Cuando vale 0 deja pasar a la salida una de las entradas de la x_0 a la x_3 (cuál de ellas depende de los otros bis de selección) y cuando vale 1 deja pasar una de las x_4 a x_7. Como un Mx-2-1 deja pasar una de entre dos entradas, dependiendo del valor de su entrada de selección, lo podemos usar conectando s_2 a su entrada de selección para decidir cuál de los dos conjuntos de entradas se deja pasar a la salida. Este Mx-2-1 es el multiplexor de más abajo en el esquema de la figura 4.24.

4.8

Símbolo del Mx-8-1 y circuito interno usando Mx-2-1.

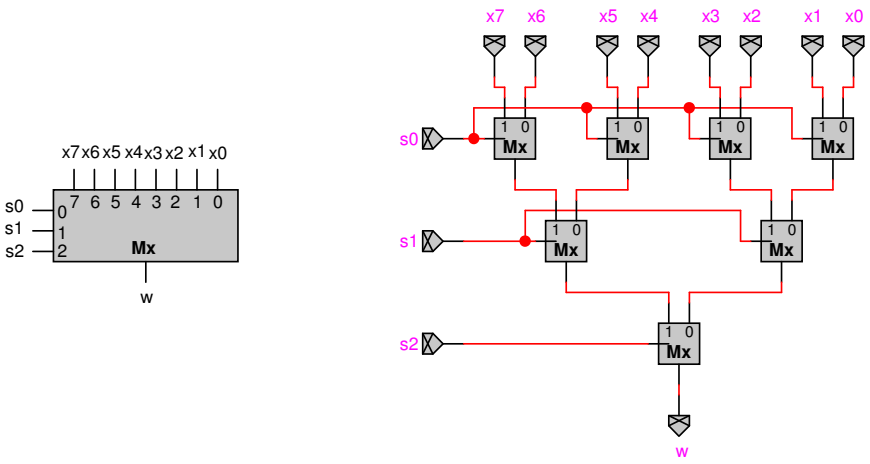


Fig. 4.24 Símbolo y esquema lógico interno del multiplexor Mx-8-1.

18

Ahora tenemos que ser capaces de tener en la entrada de datos 0 de este multiplexor una de las entradas de la x_0 a la x_3, según nos indiquen las entradas de selección s_1 y s_0 y en la otra entrada de datos, la 1, tenemos que tener una de las entradas x_4 a x_7. Ahora es cuestión de repetir el razonamiento a la mitad de la tabla. Para ver cómo seleccionamos una de las entradas x_0 a x_3 en función de s_1 y s_0, nos fijamos que s1 selecciona entre dos mitades. Cuando s_1 vale 0 la salida es una de las entradas x_0 o x_1 (dependiendo de s_0) y cuando vale 1 la salida es una de x_2 o x_3... Así repetimos el proceso y vamos subiendo en el árbol del diseño de la figura 4.24. Comprendido este diseño, somos capaces de construir cualquier multiplexor usando multiplexores 2-1.

4.9 Análisis de circuitos con bloques

Las entradas a un circuito construido con bloques aritméticos (también lógicos y de comparación) son buses de n bits (con n=16 para el computador que vamos a construir y n=8 para muchos de los ejercicios y ejemplos que hacemos). Por lo tanto, no podemos hacer un análisis del circuito como el que hacíamos en el capítulo anterior: a partir del esquema lógico del circuito obtener su tabla de verdad. Por ejemplo, en el circuito de la figura 4.25 y considerando que los buses son de 8 bits, la tabla de verdad completa del circuito tiene 2^{24} filas, tantas como combinaciones diferentes hay de los 8 bits de cada uno de los 3 vectores de entrada. Claramente, esta tabla de verdad es irrealizable.

En estos casos no vamos a obtener el valor que toman las salidas del circuito para todas las posibles combinaciones de valores de las entradas (análisis exhaustivo que resulta en la tabla de verdad), sino que haremos un análisis particular, considerando solamente algunos casos concretos de valores de las entradas: sólo calcularemos una o muy pocas filas de la tabla de verdad. Esto es lo que se hace en el ejemplo siguiente.

Ejemplo 10

Escribiremos el valor lógico de los bits de salida del circuito de la figura 4.25 (con buses y bloques de 8 bits) para dos casos concretos de valores de las entradas:

a) X = 01010011, Y = 10110111 y Z = 11000101.

b) X = 11011001, Y = 10110111 y Z = 11000101.

La solución (particularizando para los valores del apartado a) consiste en los siguientes pasos:

- Aplicar el algoritmo aritmético de la resta (ver sección 4.3.1, objetivo 4.1.2) a los vectores de bits X e Y que entran al bloque SUB obteniendo el vector resultado de 8 bits 10011100 y un 1 en la salida de borrow, b (ver figura 4.26a), con lo que se obtiene el resultado final: p=1.
- Aplicar el algoritmo aritmético de la división por potencias de 2, en concreto la división por 4 (ver sección 4.5.2, objetivo 4.1.4) al vector de bits Z que entra al bloque SRL-2 obteniendo el vector resultado de 8 bits 00110001 (ver figura 4.26b).
- Aplicar el algoritmo aritmético de la suma (ver sección 4.2.1, objetivo 4.1.1) a los vectores de 8 bits resultantes de los dos pasos anteriores, 10011100 y 00110001, que entran al bloque ADD obteniendo el vector resultado de 8 bits 11001101 y un 0 en la salida del acarreo, c (ver

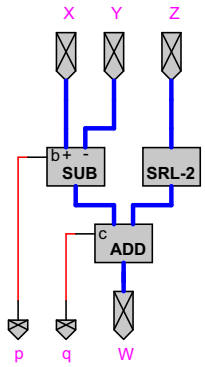


Fig. 4.25 Esquema lógico interno dede cun circuito combinacional construido interconectando los bloques aritméticos SUB, SRL-2 y ADD.

figura 4.26d), con lo que se obtienen los dos resultados finales que faltaban: q=0 y W=11001101.

Por lo que la solución de a) es: p=1, q=0, W=11001101.

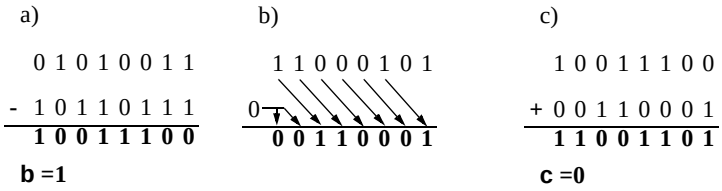


Fig. 4.26 Aplicación de algoritmos aritméticos a vectores de bits concretos a) resta, b) desplazamiento lógico a la derecha de dos bits (o división por 4 de un número natural codificado en binario) y c) suma.

La solución del apartado b) es la siguiente. La salida del bloque SUB es 00100010 y b=0, la del bloque SRL-2 es la misma que para el caso anterior 00110001 y la del bloque ADD es 01010011 y c=0. Por lo tanto la solución es p=0, q=0 y W=01010011.

Otra forma de analizar un circuito aritmético, mucho más general que la anterior, consiste en expresar el valor que representa el vector de bits de la salida (o los vectores, si hay varios buses de salida) interpretado como un número natural codificado en binario (que es el caso de aritmética y representación que tratamos en este capítulo) en función de las operaciones aritméticas que realiza el circuito sobre los valores de los números naturales representados en binario en los buses de entrada. Esto, en muchas ocasiones sólo es fácil hacerlo para el caso en que el resultado de cada una de las operaciones sea representable en n bits, siendo n el número de bits de los buses del circuito. Esto es lo que hacemos en el ejemplo siguiente. Expresar el resultado para todos los casos, incluidos los casos en que no todos los resultados parciales y finales sean representables, es en general bastante difícil y no lo pediremos, dado el carácter introductorio de este libro.

Ejemplo 11

Vamos a escribir la expresión aritmética que da el valor del número natural representado en el bus de salida W del circuito de la figura 4.25 en función de los valores representados en los buses de entrada X, Y y Z. La expresión será válida cuando el resultado de cada una de las operaciones aritméticas realizadas en el circuito sea representable en n bits, siendo n el tamaño de los buses y bloques del circuito (en este ejemplo n= 8 bits). No indicaremos el valor de la salida para los casos en que en uno o varios de los bloques aritméticos del circuito el resultado no sea representable.

Dado que el bloque SUB realiza la resta aritmética de los dos números naturales Xu e Yu representados en binario en los buses de entrada X e Y (siempre que el resultado sea representable, siempre que b=0), que el bloque SRL-k realiza la división por 2^k del valor Zu representado en el bus Z de su entrada (en este bloque el resultado siempre es representable) y que el bloque ADD realiza la suma de sus entradas, que son las salidas de los bloques anteriores (observad la interconexión de los bloques en la figura 4.25), la expresión es:

$$W_u = \left(X_u - Y_u + \frac{Z_u}{4}\right)$$

Veamos que esta expresión es cierta para los datos de entrada del caso b) del ejemplo 10 (en el que los resultados de todas las operaciones son representables). Para estos vectores de bits de entrada los valores representados son Xu = 217, Yu = 183 y Zu = 197. Aplicando la expresión anterior para estos valores obtenemos que Wu = 83, que es el valor representado por W (que según el resultado b) del ejemplo 10 es 01010011).

4.10 Diseño de nuevos bloques aritméticos

Para finalizar este capítulo de diseño ad-hoc de dispositivos combinacionales vamos a crear nuevos circuitos que realizan operaciones aritméticas sobre números naturales representados en binario, usando para ello los dispositivos que hemos creado en este capítulo y/o dispositivos con pocas entradas que podemos definir y sintetizar en suma de minterms y/o las puertas lógicas necesarias. Vamos a ver dos ejemplos.

4.10.1 Incrementador

Un incrementador binario es un dispositivo combinacional, cuyo símbolo se muestra a la derecha del párrafo, que tiene un bus de entrada X de n bits y otro de salida W también de n bits, que son los n bits de menor peso del número natural que representa la entrada incrementada en una unidad.

.

4.3.1 **Símbolo y circuito interno del incrementador, INC, con Half-adders.**

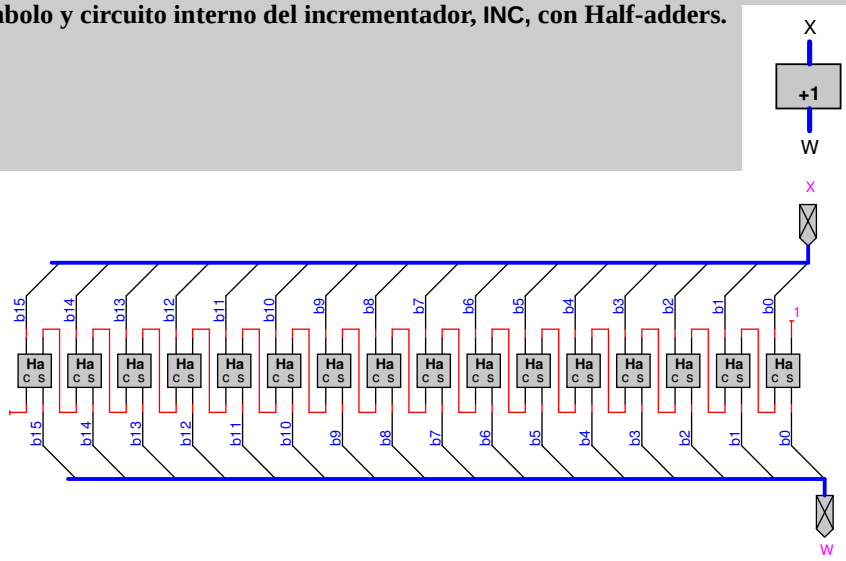


Fig. 4.27 Símbolo y esquema lógico interno del incrementador binario con Half-adders.

Como sumar 1 a un número es un caso particular de sumar dos números, un posible diseño del incrementador es hacerlo con n Full-adders en propagación del acarreo y fijar un operando con los bits $000\cdots0001$, que codifican al 1. Pero como sumar unbit con valor cero a los otros dos bits en cada uno de los Full-adders es igual a sumar simplemente los dos bits, es más sencillo usar n Half-adders en propagación del acarreo como muestra la figura 4.27, sumando un 1 en el bit de menor peso.

4.10.2 Multiplicador por 5

Deseamos diseñar un circuito combinacional con una entrada X de n bits y salida W de m bits tal que:

$$W_u = 5X_u \qquad \textbf{(EQ 10)}$$

¿Cuánto debe valer m para que el resultado codificado en los m bits de W sea siempre correcto?

La idea para obtener el diseño es representar en binario el operando constante 5 y manipular la EQ (10) hasta que aparezcan operaciones mas sencillas, que sepamos realizar en hardware. Como el 5 se codifica como 101, ya que $5 = 1x2^2+0x2^1+1x2^0 = 2^2+1$, la EQ (10) es equivalente a,

$$W_u = X_u 2^2 + X_u \qquad \textbf{(EQ 11)}$$

Resulta que ya sabemos multiplicar por una potencia de dos un número natural codificado en binario y también sabemos sumar dos números naturales. Como nos dice el enunciado que tenemos que codificar W_u con m bits, tantos como sean necesarios para que el resultado sea siempre representable, haremos las dos operaciones con los bits necesarios para que el resultado sea siempre representable.

La multiplicación por 2^2 requiere n+2 bits de resultado, que se obtiene desplazando 2 posiciones los bits de X a la izquierda y poniendo los dos bits de menor peso a 0. Ahora se deben sumar estos n+2 bits que codifican $4X_u$ más los n bits que codifican X_u. Para ello tenemos que usar un sumador binario con n+2 Full-adders en propagación del acarreo, del cual nos tenemos que quedar como salida con los n+2 bits de W a los que se tiene que concatenar por la izquierda, como bit de peso 2n+3 el acarreo de salida, para formar los m = n+3 bits que codifican siempre correctamente al resultado $5X_u$. El operando X_u que entra al sumador de n+2 bits hay que representarlo con n+2 bits. Para ello se aplica el algoritmo de extensión de rango de números naturales codificados en binario que vimos en el capítulo 1: a los n bits que representan X_u se le concatenan por la izquierda dos bits más con valor 0. La figura 4.28 muestra el diseño para n=4 bits.

Optimización. Puede hacerse un diseño más eficiente eliminando los Fa para calcular w_0 y w_1 y poniendo Half-adders en vez de Full-adders para calcular w_2, w_4 y w_5 (y w_6).

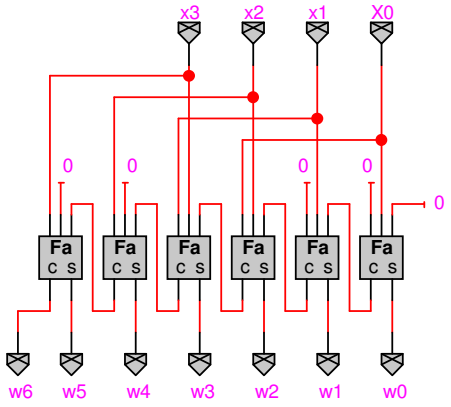


Fig. 4.28 Esquema lógico interno del multiplicador por 5, Wu = 5Xu, para n=4 y m=7