

## 5 Números enteros: representación, operaciones básicas y bloques aritméticos combinacionales

|      |  |    |
|------|--|----|
| 5.1  | Un repaso a la representación y suma de números naturales .....        | 2  |
| 5.2  | En busca de una representación eficiente para los números enteros..... | 4  |
| 5.3  | Representación de enteros en complemento a dos .....                   | 8  |
| 5.4  | Suma con detección de resultado no representable .....                 | 15 |
| 5.5  | Cambio de signo.....   | 21 |
| 5.6  | Resta .....  | 26 |
| 5.7  | Implementación de un sumador/restador .....                            | 26 |
| 5.8  | Multiplicación por potencias de dos.....                               | 27 |
| 5.9  | División por potencias de dos .....                                    | 30 |
| 5.10 | Comparación de números enteros .....                                   | 33 |

## 5.1 Un repaso a la representación y suma de números naturales

**Sistema de numeración binario.** Como hemos visto, los números naturales  $\{0, 1, 2, \dots\}$  (en inglés *unsigned integers*, o *non-negative integers*) se representan en el computador en el sistema de numeración binario. En binario, el vector  $X$  de  $n$  bits

$$X = x_{n-1}x_{n-2} \dots x_2x_1x_0 \quad x_i \in \{0,1\} \quad (\text{EQ 1})$$

representa el valor numérico

$$X_u = \sum_{i=0}^{n-1} x_i 2^i \quad (\text{EQ 2})$$

(el subíndice  $u$  indica que estamos interpretando el vector de bits como un número natural (*unsigned*) representado en binario). Por ejemplo, el vector  $X = 11001$  representa en binario el valor  $X_u = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ , que en decimal (que es como nosotros representamos los números naturales) se representa como  $X_u = 25$  (se obtiene representando las potencias de dos en decimal y realizando las operaciones en decimal).

**Rango.** El rango de números naturales representables en binario con  $n$  bits es:

$$0 \leq X_u \leq 2^n - 1 \quad (\text{EQ 3})$$

### Sumador binario

En la tabla de la figura 5.1 se muestran todos los posibles vectores de 3 bits y su valor asociado al interpretar cada vector como un número natural codificado en binario. Las entradas de esta tabla se han ordenado en función del valor de  $X_u$ , de menor a mayor.

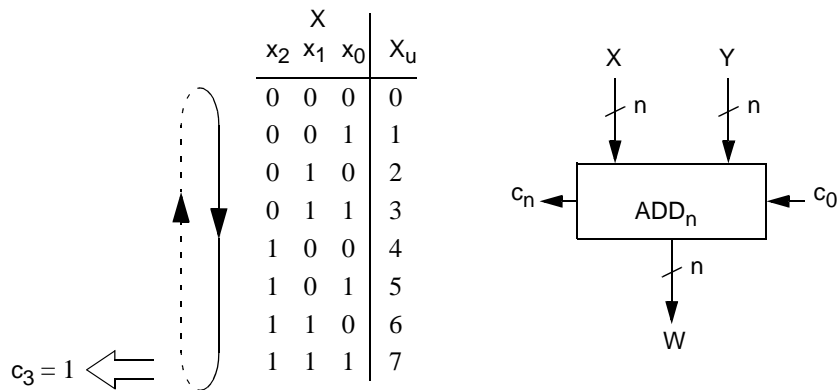


Fig. 5.1 Tabla de representación de números naturales codificados en binario con 3 bits y sumador binario de  $n$  bits: la suma como recorrido circular de la tabla de representación.

En la figura 5.1 también se representa el símbolo del sumador binario de  $n$  bits. Las entradas de este bloque combinacional son dos vectores de  $n$  bits  $X$  e  $Y$ , que representan dos números naturales  $X_u$  e  $Y_u$ , y un bit,  $c_0$ , denominado acarreo de entrada (en inglés *carry in*). El sumador calcula el valor  $W_u$ , representado mediante el vector de  $n$  bits  $W = w_{n-1}...w_1w_0$  y el bit  $c_n$ , denominado acarreo de salida, tales que:

$$c_n = \begin{cases} 0 & \text{si } X_u + Y_u + c_0 \leq 2^n - 1 \\ 1 & \text{si } X_u + Y_u + c_0 > 2^n - 1 \end{cases} \quad \text{y} \quad (\text{EQ 4})$$

$$W_u = X_u + Y_u + c_0 \quad \text{si } X_u + Y_u + c_0 \leq 2^n - 1 \quad (\text{si } c_n = 0) \quad (\text{EQ 5})$$

Esto es, el vector de bits  $W$  representa el valor correcto de la suma  $X_u + Y_u + c_0$ , cuando este resultado puede representarse en binario con  $n$  bits. Cuando el acarreo de salida,  $c_n$ , se activa indica que el resultado correcto de la suma no es representable con  $n$  bits en binario (*resultado no representable*) y por lo tanto  $w_{n-1}...w_1w_0$  no representa el resultado correcto,  $W_u \neq X_u + Y_u + c_0$ . Usando la tabla de la figura 5.1 para realizar algunas sumas que produzcan resultados no representables, observamos que siempre la diferencia entre el resultado correcto y el que se obtiene con 3 bits es de 8 ( $= 2^3$ ) unidades. Llevando esta conclusión al caso general de  $n$  bits, tenemos:

$$W_u = X_u + Y_u + c_0 - 2^n \quad \text{si } X_u + Y_u + c_0 > 2^n - 1 \quad (\text{si } c_n = 1) \quad (\text{EQ 6})$$

Otra forma de expresar la EQ (5) y la EQ (6) en una sola es:

$$W_u = (X_u + Y_u + c_0) \bmod 2^n \quad (\text{EQ 7})$$

Denotamos las funciones lógicas que obtienen los bits de salida del sumador binario de  $n$  bits, en función de los bits de entrada, como:

$$W = \text{ADD}_n(X, Y, c_0) \quad (\text{EQ 8})$$

$$c_n = \text{CARRY}_n(X, Y, c_0) \quad (\text{EQ 9})$$

En el nombre de los operadores  $\text{ADD}_n$  y  $\text{CARRY}_n$  no pondremos el subíndice  $n$  (tamaño del sumador binario) para el caso común en este curso de  $n = 16$ .

### La suma como recorrido circular de la tabla de representación

La tabla de la representación de números naturales para  $n = 3$  de la figura 5.1 puede ayudarnos a entender como funciona el sumador binario. Esta visión que damos a continuación nos será útil en la sección siguiente. Consideremos dos situaciones:

- Si tomamos una fila de la tabla, un vector de bits  $X$  (que representa el número  $X_U$ ) y le sumamos, con el sumador binario, el vector 001 (que representa el 1), obtenemos un número que está una posición adelante (más hacia abajo) en la tabla, siguiendo la dirección que indica la flecha de trazo continuo de la izquierda de la tabla. Si repetimos esta operación, vamos avanzando en la tabla hasta que llegamos a la fila 111 (que representa al número 7). En cada uno de estos casos, la representación de  $X_U+1$  se obtiene correctamente y el acarreo de salida del sumador binario,  $c_3$ , vale 0.
- Si ahora al vector 111 le sumamos el vector 001, intentando obtener la representación del valor  $7+1$ , obtenemos el vector 000 que representa el valor 0. Esto es así porque el 8, que es el resultado correcto, no puede representarse con 3 bits. El paso del 111 al 000 se indica mediante una flecha discontinua<sup>1</sup> que une el final de la tabla con el principio. Este paso se detecta en el sumador binario porque  $c_3$  vale 1, indicando *resultado no representable*.

Esto se puede generalizar, de forma que la representación binaria de  $X_U + Y_U$  con 3 bits puede encontrarse contando  $Y_U$  posiciones en la dirección de la flecha a partir de la fila que codifica el valor de  $X_U$  en la tabla de la figura 5.1. Si durante este proceso de contar pasamos del final de la tabla al principio, el resultado no es representable con 3 bits y  $c_3$  valdrá 1 (al sumar dos números de 3 bits, a lo sumo se pasa una sola vez del final de la tabla al principio).

## 5.2 En busca de una representación eficiente para los números enteros

En esta sección obtenemos la representación que usan los computadores para almacenar y hacer cálculos con números enteros. Un número perteneciente al conjunto de los enteros  $\{\dots, -2, -1, 0, 1, 2, \dots\}$  (en inglés *signed integer* o simplemente *integer*) se representa internamente en el computador, como cualquier otra información, mediante un vector de  $n$  bits.

$$X = x_{n-1}x_{n-2} \dots x_2x_1x_0 \text{ con } x_i \in \{0,1\}.$$

Definir una representación consiste en encontrar una tabla o una expresión aritmética (como se ha hecho en la sección anterior para los números naturales) que para cada posible vector de bits nos indique el número que representa. La fórmula/tabla no puede ser la misma para los enteros que para los naturales (figura 5.1 o EQ (2)), ya que con esta expresión sólo se obtienen números positivos.

*Hay muchos posibles sistemas de representar un número entero mediante un vector de bits, la diferencia entre ellos estriba en lo sencillo o complicado que resulta realizar operaciones aritméticas con cada uno de estos sistemas. Como caso más usual de operación aritmética consideramos la suma, que además es una operación básica para cálculos más complejos como la resta y la multiplicación.*

Veamos algunos sistemas de representación de números enteros.

---

1. Pasar la frontera del 111 al 000 en la tabla, es equivalente a hacer la operación *mod* 8. Así que el sumador binario ha calculado  $(7 + 1) \bmod 8 = 0$ .

### 5.2.1 Signo y magnitud en base 2

La representación de un número entero mediante un vector de bits más familiar para nosotros, puesto que es similar a la que usamos normalmente en base 10, es la representación en signo y magnitud en base 2. Dado un vector  $X$  de  $n$  bits que representa al número entero  $X_{sm}$  en signo y magnitud (en inglés *sign and magnitude*), el bit de más a la izquierda del vector de bits codifica el signo, si  $x_{n-1}$  vale 0, indica que el número es positivo y si vale 1 que es negativo, y los  $n-1$  bits restantes representan en binario el valor absoluto de  $X_{sm}$  (la magnitud de  $X_{sm}$ ), que es un número natural.

La figura 5.2 muestra en una tabla los 8 posibles vectores de 3 bits y el valor del número entero que representan en signo y magnitud. La expresión algebraica general para obtener el valor de un número a partir del vector de bits que lo representa también se muestra en la figura. De esta fórmula es fácil deducir que el rango de números representables positivos y negativos es el mismo,  $-(2^{n-1} - 1) \leq X_{sm} \leq 2^{n-1} - 1$ ; pero hay dos posibles representaciones del valor 0.

| X     |       |       | $X_{sm}$ |
|-------|-------|-------|----------|
| $x_2$ | $x_1$ | $x_0$ |          |
| 0     | 0     | 0     | 0        |
| 0     | 0     | 1     | 1        |
| 0     | 1     | 0     | 2        |
| 0     | 1     | 1     | 3        |
| 1     | 0     | 0     | -0       |
| 1     | 0     | 1     | -1       |
| 1     | 1     | 0     | -2       |
| 1     | 1     | 1     | -3       |

$$X_{sm} = \begin{cases} \sum_{i=0}^{n-2} x_i 2^i & \text{si } x_{n-1} = 0 \\ -\left(\sum_{i=0}^{n-2} x_i 2^i\right) & \text{si } x_{n-1} = 1 \end{cases}$$

Fig. 5.2 Tabla de la representación signo y magnitud en binario con 3 bits y fórmula para  $n$  bits.

Repasemos ahora el algoritmo que usamos habitualmente para sumar en signo y magnitud en decimal, ya que haremos lo mismo en signo y magnitud en binario:

- Cuando los operandos tienen el mismo signo, la magnitud del resultado es la suma de las magnitudes de los operandos y el signo del resultado es el signo de los operandos. Por ejemplo,  $(+34) + (+12) = +46$  y también  $(-34) + (-12) = -46$ .
- Cuando los operandos son de distinto signo, para obtener la magnitud del resultado se resta a la magnitud mayor la menor y el signo del resultado es el signo del operando de mayor magnitud. Por ejemplo,  $(+34) + (-12) = +24$  y también  $(-34) + (+12) = -24$ .

Este algoritmo requiere un sumador de números naturales codificados en binario con  $n-1$  bits, para cuando los dos operandos son del mismo signo y un comparador y un restador de números naturales codificados en binario con  $n-1$  bits, para cuando los números tienen distinto signo. Aunque la comparación y la resta pueden hacerse con pequeñas modificaciones del sumador binario y aunque existen algoritmos más óptimos, podemos concluir lo siguiente.

*La suma de números enteros en signo y magnitud es más costosa en hardware y/o en tiempo de propagación que la suma de números naturales en binario. Por esta razón, los computadores actuales no usan la representación en signo y magnitud para los números enteros, aunque para los humanos sea la más familiar.*

### 5.2.2 Otras representaciones más efectivas

Los computadores usan una representación en la que la suma de números enteros se puede realizar con el mismo sumador que se usa para los números naturales codificados en binario, que hemos denominado *sumador binario*. ¿Sabrías encontrar una representación con estas características?

Desde luego, la representación en signo y magnitud no cumple este objetivo para todos los números. La figura 5.3 muestra dos ejemplos de suma de números representados en signo y magnitud usando el sumador binario. En el primer ejemplo se suman dos números positivos y se obtiene el resultado correcto. Esto es así porque la representación en signo y magnitud de los números enteros positivos es la misma que la de los números naturales representados en binario. En el segundo ejemplo, en el que se suman dos números de distinto signo, el resultado de la suma usando un sumador binario no es correcto (a pesar de que el resultado correcto sí que es representable en signo y magnitud con 3 bits).

| a) | Vectores de bits   | Suma correcta en Sign. y Mag.                        | b) | Vectores de bits   | Suma correcta en Sign. y Mag.                         |
|----|--|--|----|--|---|
|    | $\begin{array}{r} 0\ 0\ 1 \\ +\ 0\ 0\ 1 \\ \hline 0\ 1\ 0 \end{array}$ | $\begin{array}{r} 1 \\ +\ 1 \\ \hline 2 \end{array}$ |    | $\begin{array}{r} 1\ 0\ 1 \\ +\ 0\ 0\ 1 \\ \hline 1\ 1\ 0 \end{array}$ | $\begin{array}{r} -1 \\ +\ 1 \\ \hline 0 \end{array}$ |
|    |  | =  |    |  | ≠   |

Fig. 5.3 Ejemplos de suma de 2 números enteros en signo y magnitud usando el sumador binario. En el primer ejemplo el resultado es correcto, mientras que en el segundo no.

Un posible razonamiento para encontrar una representación en la que la suma se realice correctamente con el sumador binario es el siguiente:

- Codificamos los números positivos como en binario (como ocurre en la codificación en signo y magnitud en base 2), con lo que nos aseguramos que la suma de positivos será correcta con el sumador binario (siempre que el resultado sea representable, claro).
- ¿Cómo codificamos los negativos? Si recordamos como opera el sumador binario, contando en la dirección de la flecha en la tabla “circular” de la figura 5.1, podemos deducir que el valor -1 se tiene que codificar como 111 (para  $n=3$ ). Este es el único vector de bits que al sumarle, con el sumador binario, el vector 001, que como hemos dicho en el punto anterior representa al 1, obtenemos el vector 000, que representa el 0 y es el resultado correcto de  $-1+1$ .

**Notación.** Llamamos  $X_s$  al valor representado por el vector de bits  $X$  en la representación que estamos buscando para números enteros; la  $s$  del subíndice viene del inglés *signed integers*).

Una vez decidido que los números positivos se codifican como en binario, tenemos que codificar los negativos de forma que la codificación del número  $X_S+1$  sea la siguiente en la tabla a la de  $X_S$ , avanzando en la dirección de la flecha. Por ello, el -2 debe codificarse como 110, para que al sumarle, con el sumador de binario, el vector 001 se obtenga el vector 111, que acabamos de decidir que representa al -1.

En la figura 5.4 se muestran tres posibles representaciones para los números enteros con 3 bits que cumplen la condición de que la suma en estos sistemas se puede realizar con el sumador binario, obteniéndose el resultado correcto, excepto cuando el resultado no puede representarse con tres bits, en cada uno de los sistemas. Podéis comprobar esto con facilidad.

**a)**

| X     |       |       |       |
|-------|-------|-------|-------|
| $x_2$ | $x_1$ | $x_0$ | $X_a$ |
| 0     | 0     | 0     | 0     |
| 0     | 0     | 1     | 1     |
| 0     | 1     | 0     | 2     |
| 0     | 1     | 1     | 3     |
| 1     | 0     | 0     | 4     |
| 1     | 0     | 1     | -3    |
| 1     | 1     | 0     | -2    |
| 1     | 1     | 1     | -1    |

**b)**

| X     |       |       |       |
|-------|-------|-------|-------|
| $x_2$ | $x_1$ | $x_0$ | $X_b$ |
| 0     | 0     | 0     | 0     |
| 0     | 0     | 1     | 1     |
| 0     | 1     | 0     | 2     |
| 0     | 1     | 1     | -5    |
| 1     | 0     | 0     | -4    |
| 1     | 0     | 1     | -3    |
| 1     | 1     | 0     | -2    |
| 1     | 1     | 1     | -1    |

**c)**

| X     |       |       |       |
|-------|-------|-------|-------|
| $x_2$ | $x_1$ | $x_0$ | $X_c$ |
| 0     | 0     | 0     | 0     |
| 0     | 0     | 1     | 1     |
| 0     | 1     | 0     | 2     |
| 0     | 1     | 1     | 3     |
| 1     | 0     | 0     | -4    |
| 1     | 0     | 1     | -3    |
| 1     | 1     | 0     | -2    |
| 1     | 1     | 1     | -1    |

Fig. 5.4 Tres posibles representaciones de números enteros con 3 bits. Se diferencian en el rango de valores representables pero en los tres casos la suma puede realizarse con el sumador binario (para resultados dentro del rango).

¿En qué difieren las tres representaciones, a), b) y c), de la figura 5.4? Se diferencian en el rango: en qué números pueden representarse con 3 bits. Para cada una de las representaciones el rango es:

a)  $-3 \leq X_a \leq 4$ ,

b)  $-5 \leq X_b \leq 2$ ,

c)  $-4 \leq X_c \leq 3$ .

Ello hace que la detección de *resultado no representable* con 3 bits difiera en cada uno de estos tres sistemas de representación y sea diferente también con el caso de binario. Además, en ninguno de estos tres sistemas, pasar de 111 al 000 (del -1 al 0) indica resultado no representable, a diferencia del caso de binario. Esto es, el acarreo de salida del sumador binario, para estas tres representaciones, no indica resultado de la suma no representable con n bits.

### 5.2.3 Complemento a dos

¿Cuál de las representaciones de la figura 5.4 se usa para codificar un número entero en un computador? Aunque alguno de los primeros computadores no la usaba, actualmente todos usan la representación c), que se denomina **Complemento a dos** (de forma abreviada **Ca2**). El Ca2, a diferencia de las representaciones a) y b) o de otras en las que se pueda sumar con el sumador binario, aunque tengan distinto rango, tiene las siguientes características:

- Tiene un rango más simétrico que las otras representaciones, aunque no es simétrico del todo ya que puede representar un negativo mayor en valor absoluto que el mayor de los positivos (esto es inevitable porque en estos sistemas el 0 tiene una sola representación (ver figura 5.4).
- Saber el signo del número representado por un vector de bits es sencillo. Para saber si un número es positivo o negativo sólo hay que observar el valor del dígito de más a la izquierda, si vale 0 el número es positivo y si vale 1 es negativo (ver figura 5.4). Con esto estamos considerando, como es usual, que el 0 es un número positivo.
- La detección de resultado no representable es más sencilla que en los otros casos (esto es así por la facilidad en la identificación del signo y lo estudiamos más adelante).

#### 5.1

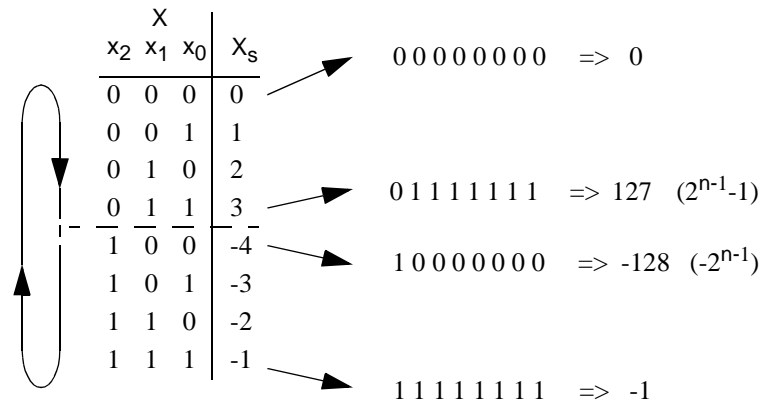
Las ventajas de usar dentro del computador la representación en Ca2 para los números enteros frente a la de signo y magnitud en base dos (más parecida al sistema que usamos los humanos) son las siguientes:

- El sumador en Ca2 es el mismo que el sumador para naturales representados en binario (excepto la detección de resultado no representable con  $n$  bits) por lo que no harán falta dos sumadores distintos en el computador: con uno solo podremos sumar tanto números naturales como enteros.
- El sumador binario es más sencillo (menos hardware y menor tiempo de propagación) que el sumador en signo y magnitud en base dos (aunque esto no lo hemos visto en detalle).

### 5.3 Representación de enteros en complemento a dos

Se trata ahora de generalizar para  $n$  bits la representación que hemos visto para 3 bits en la figura 5.4 c) ¿Sabrías escribir una tabla con la representación para  $n = 8$ ? Seguro que sí, pero sería muy pesado ya que la tabla tiene 256 entradas. Vamos a escribir solamente algunos números característicos, por ejemplo, la representación de los números más pequeños y más grandes, tanto positivos como negativos. La figura 5.5 repite la tabla para  $n = 3$  y muestra la generalización de estos números para  $n = 8$ .





| X     |       |       | $X_s$ |  |
|-------|-------|-------|-------|--|
| $x_2$ | $x_1$ | $x_0$ |       |  |
| 0     | 0     | 0     | 0     | $00000000 \Rightarrow 0$                     |
| 0     | 0     | 1     | 1     |  |
| 0     | 1     | 0     | 2     |  |
| 0     | 1     | 1     | 3     | $01111111 \Rightarrow 127 \quad (2^{n-1}-1)$ |
| 1     | 0     | 0     | -4    | $10000000 \Rightarrow -128 \quad (-2^{n-1})$ |
| 1     | 0     | 1     | -3    |  |
| 1     | 1     | 0     | -2    |  |
| 1     | 1     | 1     | -1    | $11111111 \Rightarrow -1$                    |

Fig. 5.5 Tabla de la representación en Ca2 para 3 bits y representación con 8 bits de algunos números característicos.

### 5.3.1 Fórmula del valor de un número en función del valor de los dígitos que lo representan

¿Sabríais encontrar una fórmula para obtener el valor de un número entero a partir de los  $n$  dígitos binarios de su representación en Ca2? ¿Una fórmula en función de  $n$  es la mejor generalización!

En primer lugar, consideremos por separado el caso en que el valor representado,  $X_s$ , es positivo y el caso en que es negativo.

- Número positivo ( $X_s \geq 0$ , o lo que es lo mismo  $x_{n-1} = 0$ )

Como la representación de un número positivo en Ca2 es igual que la representación en binario, la fórmula tiene que ser la misma que para el caso de naturales representados en binario, EQ (2). No obstante, en este caso no hace falta que intervenga el bit de más a la izquierda,  $x_{n-1}$ , puesto que para los positivos siempre vale cero.

$$X_s = \sum_{i=0}^{n-2} x_i 2^i \quad \text{para } x_{n-1} = 0$$

- Número negativo ( $X_s < 0$ , o lo que es lo mismo  $x_{n-1} = 1$ )

Observemos la figura 5.5 para los números negativos. Si al dígito de mas a la izquierda le damos el mismo peso que le corresponde en binario ( $2^{n-1}$ ) pero con signo negativo y al resto de dígitos el peso y el signo positivo correspondiente a binario, obtenemos los valores correctos de los números

-128 ( $-128 = -128 + 0$ ) y -1 ( $-1 = -128 + 127$ , ya que de la representación de los números naturales sabemos que  $\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$ ). Generalizando obtenemos que para los negativos:

$$X_s = -2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \quad \text{para } x_{n-1} = 1$$

5.2

La fórmula que da el valor de un número entero,  $X_s$ , en función del valor de los  $n$  bits,  $x_{n-1}x_{n-2} \dots x_2x_1x_0$ , del vector  $X$  que lo representa en complemento a dos (tanto para los positivos ( $x_{n-1} = 0$ ) como para los negativos ( $x_{n-1} = 1$ ), es:

$$X_s = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \quad \text{(EQ 10)}$$

### 5.3.2 Rango

¿Cuál es el rango de la representación en Ca2 para  $n$  bits? El número negativo con mayor valor absoluto que se puede obtener de la EQ (10) se encuentra dando valor 1 a  $x_{n-1}$  y valor 0 a todos los demás dígitos. Este número es el  $-2^{n-1}$ . Por otro lado, el positivo mayor se obtiene dando valor 0 a  $x_{n-1}$  para que no reste nada y valor 1 a todos los demás dígitos que suman. Este número es el  $2^{n-1}-1$ . Como todos los números enteros entre estos dos están representados:

5.3

El **rango** de la representación en complemento a dos con  $n$  bits es:

$$-2^{n-1} \leq X_s \leq 2^{n-1} - 1 \quad \text{(EQ 11)}$$

### 5.3.3 Extensión de rango

Dado el vector  $X$  de  $n$  bits que representa en Ca2 al número entero  $X_s$ , ¿cómo se obtiene en vector  $W$  de  $n+1$  bits que representa a ese mismo número,  $W_s = X_s$ ? La figura 5.6 muestra todos los números representables en Ca2 con 3 y 4 bits. Las flechas indican como se representa con 4 bits cada uno de los 8 posibles números representados con 3 bits. Generalizando esta idea, se dice que la extensión de rango

se efectúa extendiendo el bit de signo. Esto es, los  $n$  bits de menos peso de la nueva representación son los mismos que los de la representación original y como bit de más peso de la nueva representación se replica el bit de más peso de la original. Por ejemplo, el número 3 con 3 bits se representa como 011 y con 4 bits como 0011. El número -3 se representa con 3 bits como 101 y con 4 como 1101. Para los números positivos la demostración es trivial usando la EQ (10) y para demostrar el caso de los negativos sólo hay que saber que  $-2^{n-1} = -2^n + 2^{n-1}$ .

¿Sabrías implementar un bloque combinacional para extender el rango de  $n$  a  $n+1$  bits? En la figura 5.6 puede verse la implementación para pasar de 3 a 4 bits (le denominamos SE del inglés *Sign Extension*).

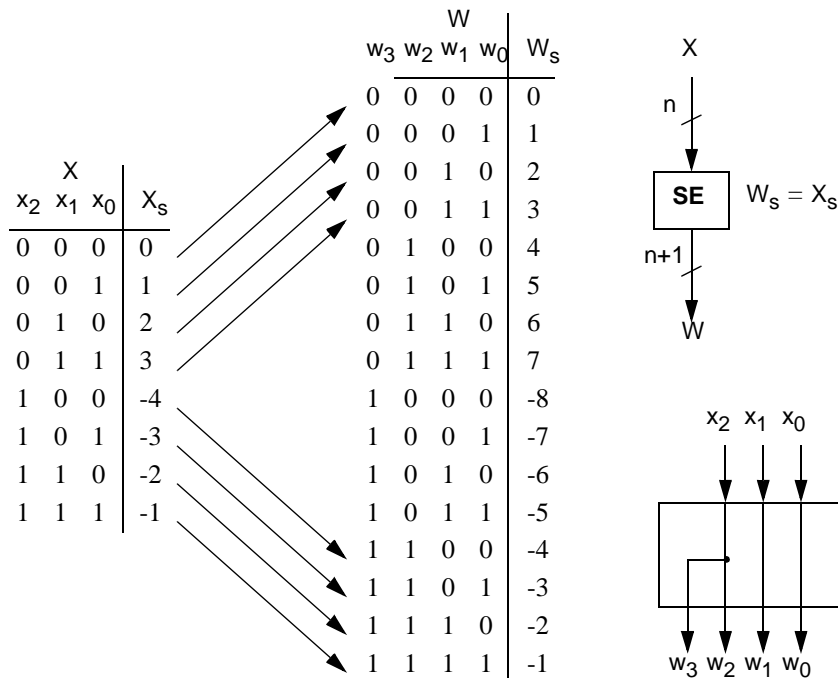


Fig. 5.6 Tablas de las representaciones en Ca2 para 3 y 4 bits. Implementación de un bloque de extensión de rango de 3 a 4 bits.

Obsérvese que este bloque se implementa sin ninguna puerta lógica, sólo requiere cables. ¡Qué sencillo!

¿Sabrías hacer ahora la extensión de rango de  $n$  a  $n + k$  bits? La figura 5.7 muestra el bloque que extiende el rango en Ca2 de 3 a 6 bits.

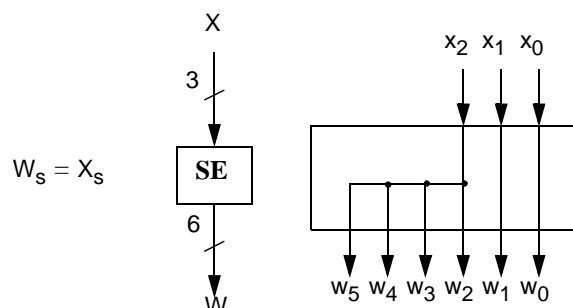


Fig. 5.7 Implementación de la extensión de rango de 3 a 6 bits.

### 5.3.4 Cambio de representación para números enteros

#### De complemento a dos a decimal en signo y magnitud

Para encontrar el valor de un número entero (y representarlo en signo y magnitud decimal, que es como estamos acostumbrados a representar los enteros) a partir de su representación en complemento a dos mediante el vector de  $n$  bits  $X = x_{n-1}x_{n-2} \dots x_2x_1x_0$  con  $x_i \in \{0,1\}$  simplemente hay que operar los bits de  $X$  de acuerdo con la fórmula de la definición del Ca2, EQ (10):

$$X_s = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

#### Ejemplo 1

¿Qué número entero representa el vector de 8 bits 11001010 en complemento a 2? Aplicando la fórmula anterior tenemos que:

$$X_s = -2^7 + 2^6 + 2^3 + 2 = -54$$

#### De decimal en signo y magnitud a complemento a dos

Vamos a ver ahora dos procedimientos para obtener los bits de la representación de un número entero. Después de ver el algoritmo de cambio de signo veremos otra forma más de hacerlo.

**Procedimiento de las divisiones por dos.** De la misma forma que se demostró como pasar un número natural de decimal a binario, se puede demostrar el procedimiento para pasar un número entero de decimal en signo y magnitud a Ca2. La fórmula de la representación en binario y en Ca2 es casi la misma, un sumatorio de todos los dígitos cada uno de ellos multiplicado por su peso, excepto que el Ca2 el dígito de mas peso se resta en vez de sumarse. Esto se puede interpretar como que el dígito de

más peso vale 0 o -1 en vez de 0 o 1. Por ello, el procedimiento para obtener los dígitos binarios de la representación en Ca2 a partir de un número decimal es similar al caso binario. En ambos casos se obtienen los dígitos comenzando por el de más peso hasta obtener finalmente el de menos peso. Para binario, se divide el número entre 2 y el resto es el dígito de menos peso, el cociente se divide a su vez por 2 y el resto es el siguiente dígito y se repite el proceso hasta que el cociente vale 0 y el resto es el dígito de más peso. Las divisiones se efectúan de forma que los restos sean positivos (0 o 1, pues son los dígitos). Esto no hace falta decirlo ya que el número es siempre positivo y hacerlo así es lo normal). Pero para obtener la representación en Ca2 hay que efectuar las divisiones de forma que los restos sean positivos (y esto no es lo que hacemos normalmente al dividir un número negativo entre un positivo) hasta que el cociente pueda ser cero y el resto de esta última división sea 0 o -1. La figura 5.8 ejemplifica este algoritmo para dos casos, -5 y 5.

$$\begin{array}{r}
 -5 \quad \underline{2} \\
 -6 \quad -3 \quad \underline{2} \\
 1 \quad -4 \quad -2 \quad \underline{2} \\
 \quad \quad 1 \quad -2 \quad -1 \quad \underline{2} \\
 \quad \quad \quad 0 \quad 0 \quad 0 \\
 \quad \quad \quad \quad -1
 \end{array}
 \quad -5 \quad \Rightarrow \quad 1 \ 0 \ 1 \ 1$$
  

$$\begin{array}{r}
 5 \quad \underline{2} \\
 4 \quad 2 \quad \underline{2} \\
 1 \quad 2 \quad 1 \quad \underline{2} \\
 \quad 0 \quad 0 \quad 0 \quad \underline{2} \\
 \quad \quad 1 \quad 0 \quad 0 \\
 \quad \quad \quad 0
 \end{array}
 \quad 5 \quad \Rightarrow \quad 0 \ 1 \ 0 \ 1$$

Fig. 5.8 divisiones sucesivas por 2 para obtener la representación en Ca2 del -5 y del 5.

**Procedimiento de obtener  $X$  pasando por el cálculo de  $X_u$  a partir de  $X_s$ .** Dado un vector de  $n$  bits,  $X$ , podemos encontrar el valor que representa interpretándolo como un número natural codificado en binario aplicando la EQ (2) y también podemos encontrar el valor que representa  $X$  interpretándolo como un número entero codificado en Ca2 aplicando la EQ (10).

Veamos la relación que existe entre estas dos interpretaciones. La EQ (2) se puede expresar, quitando del sumatorio el término de más peso, como:

$$X_u = x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

Si en esta ecuación sustituimos el valor del sumatorio que se obtiene de despejar la EQ (10),

$\sum_{i=0}^{n-2} x_i 2^i = X_s + x_{n-1} 2^{n-1}$ , obtenemos que:

$$X_u = X_s + x_{n-1} \cdot 2^n \quad (\text{EQ 12})$$

Dado un número entero  $X_s$  y un número de bits  $n$  tal que  $X_s$  pueda representarse en Ca2 con  $n$  bits ( $n$  tal que  $-2^{n-1} \leq X_s \leq 2^{n-1} - 1$ ) sabemos que si  $X_s$  es positivo entonces  $x_{n-1}$  es 0 y si  $X_s$  es negativo entonces  $x_{n-1}$  es 1. Esto hace que la relación de la EQ (12) pueda escribirse como:

$$X_u = \begin{cases} X_s & \text{si } X_s \text{ es positivo} \\ X_s + 2^n & \text{si } X_s \text{ es negativo} \end{cases} \quad (\text{EQ 13})$$

Aplicando esta fórmula, dado el número entero  $X_s$  encontramos fácilmente el valor natural  $X_u$ . Ahora mediante el procedimiento de las divisiones sucesivas entre 2 que vimos en el capítulo 1 para pasar de decimal a binario encontramos el vector de  $n$  bits  $X$  que representa a  $X_u$  en binario, pero que también representa a  $X_s$  en Ca2.

### Ejemplo 2

Vamos a encontrar el vector de bits que representa en Ca2 al número entero  $X_s = -54$ .

Para representar al -54 hacen falta al menos 7 bits, ya que  $-2^{7-1} \leq -54 \leq 2^{7-1} - 1$ . De la EQ (13) para  $n = 7$  obtenemos que, como -54 es negativo,  $X_u = -54 + 128 = 74$ . Ahora mediante sucesivas divisiones entre 2 obtenemos que el 74 se representa en binario mediante el vector de bits  $X = 1001010$ .

Si en el enunciado del problema nos dicen que quieren, por ejemplo, la representación en Ca2 del -54 en 8 bits (u otro valor mayor que 7) aplicamos la EQ (13) para  $n=8$  y obtenemos que  $X_u = -54 + 256 = 202$  que en binario se representa mediante  $X = 11001010$ . Como vemos es la extensión de rango en Ca2 del vector que obtuvimos para 7 bits, 1001010. También se puede comprobar la corrección del resultado repasando el ejemplo 1.

### Ejemplo 3

Cada fila de la tabla siguiente tiene 3 columnas con: el vector de 8 bits  $X$ , el valor que representa  $X$  interpretado como un número natural codificado en binario,  $X_u$ , y el valor que representa  $X$  interpretado como un número entero codificado en complemento a dos,  $X_s$ . Usando la EQ (12) o la EQ (13), para cada fila, a partir del valor de una columna de la tabla se pueden obtener los valores

de las otras dos columnas.

| X        | $X_u$ | $X_s$ |
|----------|-------|-------|
| 11001011 | 203   | -53   |
| 00101110 | 46    | 46    |
| 11100110 | 230   | -26   |
| 00011010 | 26    | 26    |

## 5.4 Suma con detección de resultado no representable

Como ya hemos visto, la suma de dos números enteros representados en Ca2 se efectúa con el mismo sumador que usamos para naturales representados en binario. Este era el objetivo de la representación en Ca2. La diferencia entre la suma de números naturales representados en binario y la de números enteros representados en Ca2 radica en la detección de resultado no representable.

### 5.4.1 Detección de resultado no representable

Si trabajamos con naturales en binario, el resultado de la suma no es representable con  $n$  bits cuando el acarreo de salida del sumador,  $c_n$ , vale 1. Esto ocurre por ejemplo para  $n = 3$  al sumarle 1 (representado por 001) al mayor número positivo que se puede representar con 3 bits, que es el 7 (111). En este caso, el resultado del sumador es 000, que representa el 0 y  $c_3$  vale 1 que indica el resultado de la suma no es representable con 3 bits. Sin embargo, en Ca2, el acarreo de salida del sumador,  $c_n$ , no indica que el resultado no sea representable. Para los mismos vectores de bits que en el ejemplo anterior, estamos sumando 1 (representado por 001) al -1 (representado por 111) y el resultado del sumador es el correcto, 000, que representa el 0, aunque el acarreo de salida  $c_3$  vale 1.

¿Cómo se sabe si los  $n$  bits del resultado del sumador binario cuando trabajamos en Ca2 representan correctamente el resultado de la suma? La detección de resultado no representable al sumar dos números en Ca2 puede efectuarse observando los bits que indican el signo de los operandos,  $x_{n-1}$  e  $y_{n-1}$ , y del resultado,  $w_{n-1}$ . Para encontrar la expresión lógica de resultado no representable, descomponemos el problema en dos casos (y lo particularizamos para  $n=3$ , cuya representación se muestra en la tabla de la figura 5.5). En el razonamiento que sigue hay que tener claro que en Ca2 con  $n$  bits podemos representar todos los enteros entre el más negativo de los representables ( $-2^{n-1} = -4$ ) y el más positivo ( $2^{n-1} - 1 = 3$ ), como hemos visto al hablar del rango.

### Operandos con distinto signo.

La suma de un número positivo más un número negativo más el valor más desfavorable del acarreo de entrada nunca da un número más alejado del 0 que el operando de valor absoluto mayor. Por ejemplo, para  $n = 3$ ,  $X_s + Y_s + c_{in} = 3 + (-1) + 1 = 3$ , y también  $X_s + Y_s + c_{in} = -4 + 0 + 0 = -4$ . De la afirmación anterior se deduce que, si los operandos  $X_s$  e  $Y_s$  son representables con  $n$  bits, el resultado  $X_s + Y_s + c_{in}$  también lo es. Por lo tanto, *el resultado siempre es representable al sumar un positivo con un negativo.*

### Operandos con el mismo signo

**Suma de positivos.** La suma de cualquier par de números positivos representables con  $n$  bits en Ca2, del 0 al  $2^{n-1}-1$  (del 0 al 3), más el acarreo de entrada (el valor más desfavorable para este caso es  $c_{in} = 1$ ) da números entre el 0 y el  $2^n-1$  (entre el 0 y el 7). Si sumamos los números representados en Ca2 con el sumador binario, obtendremos números en Ca2:

- entre el 0 y el  $2^{n-1}-1$  (entre el 0 y el 3), que son los casos de resultados correctos, o
- entre el  $-2^{n-1}$  y el -1 (entre el -4 y el -1), que son los casos de resultados incorrectos, ya que en estos casos no se pueden representar los resultados correctos, que van del  $2^{n-1}$  al  $2^n-1$  (del 4 al 7).

Nunca, por muy grandes que sean los operandos positivos y con  $c_{in} = 1$  (por ejemplo,  $3+3+1$ ), pasaremos la frontera del 111 al 000. Es decir, *que si al sumar dos números positivos el resultado es positivo este resultado es correcto, pero si el resultado es negativo, este resultado no es correcto, porque el resultado correcto no es representable con  $n$  bits en Ca2.*

**Suma de negativos.** Un razonamiento parecido puede hacerse al sumar dos negativos. Para  $n=3$ , al sumar dos negativos (del -4 al -1) más el acarreo más desfavorable (para este caso  $c_{in} = 0$ ), el resultado en Ca2 con 3 bits que nos entrega el sumador binario se encuentra:

- entre el -4 y el -2 (casos de resultado correcto) o
- entre el 0 y el 3 (resultados incorrectos, ya que los resultados correctos, que van del -8 al -5, no se pueden representar con 3 bits).

Es decir, *si al sumar dos números negativos el resultado es negativo, este resultado es correcto; pero si el resultado es positivo, este resultado no es correcto, porque el resultado correcto no es representable con  $n$  bits en Ca2.*

### Conclusión

La regla completa de detección de resultado no representable para la suma en Ca2 con  $n$  bits es: si al sumar dos números enteros del mismo signo (sea cual sea el acarreo de entrada) el resultado que nos dan los  $n$  bits del sumador binario tiene signo contrario, este resultado no es correcto ya que el resultado correcto no es representable con  $n$  bits en Ca2. En cualquier otro caso el resultado es correcto.

Del texto anterior obtenemos la siguiente expresión lógica para la detección de resultado no representable para la suma en Ca2 con  $n$  bits:

$$v_n = \bar{x}_{n-1} \cdot \bar{y}_{n-1} \cdot w_{n-1} + x_{n-1} \cdot y_{n-1} \cdot \bar{w}_{n-1} \quad (\text{EQ 14})$$

Por ello, si queremos saber si la salida  $W$  del sumador binario es correcta cuando interpretamos los vectores de bits  $X$ ,  $Y$  y  $W$  como números enteros representados en Ca2, debemos añadirle al circuito sumador el bit de salida  $v_n$ . Denotamos a la función lógica que lo calcula, operador  $OVF_n$ :

$$v_n = OVF_n(X, Y, c_0) \quad (\text{EQ 15})$$



Ni en el nombre de la variable lógica  $v_n$  ni en el del operador  $OVF_n$  pondremos el subíndice  $n$  (tamaño del sumador binario) para el caso común en este curso de  $n = 16$ .

OVF son las primeras letras de la palabra en inglés *Overflow*, que significa desbordamiento. En la literatura técnica anglosajona se usa el término *Overflow* (o en muchas ocasiones la letra  $v$ ) para referirse al caso de resultado no representable en Ca2, pero no se suele usar para el caso de representación de naturales en binario. Aunque la palabra desbordamiento da una idea clara de que el resultado no es correcto porque excede de la capacidad de representación del vector de  $n$  bits y esto es válido tanto para binario como para Ca2, no usaremos la palabra desbordamiento (*overflow*) para referirnos a esto en general, sino que usaremos el termino resultado no representable. Usaremos el termino desbordamiento, *overflow*, OVF o la letra  $v$ , solamente para el caso de Ca2.

#### Ejemplo 4

¿Sabrías demostrar que es válida la siguiente expresión lógica alternativa para la detección de resultado no representable en Ca2 (el símbolo  $\wedge$  denota a la función lógica Xor de dos variables)?

$$v_n = c_n \wedge c_{n-1} \quad (\text{EQ 16})$$

Para demostrarlo, construimos la tabla de verdad del Full-adder de más peso del sumador binario construido con Full-adders conectados con propagación del acarreo, ya que es el que genera las señales  $c_n$  y  $w_{n-1}$  a partir de  $x_{n-1}$ ,  $y_{n-1}$  y  $c_{n-1}$ . El resultado puede verse en las cinco primeras columnas de la tabla de la figura 5.9. Por último, puede observarse que la columna  $v_n$  de la tabla de la figura 5.9 se obtiene tanto a partir de la expresión EQ (14) como de la expresión EQ (16), con lo que queda demostrado que ambas expresiones del overflow son válidas.

| $x_{n-1}$ | $y_{n-1}$ | $c_{n-1}$ | $c_n$ | $w_{n-1}$ | $v_n$ |
|-----------|-----------|-----------|-------|-----------|-------|
| 0         | 0         | 0         | 0     | 0         | 0     |
| 0         | 0         | 1         | 0     | 1         | 1     |
| 0         | 1         | 0         | 0     | 1         | 0     |
| 0         | 1         | 1         | 1     | 0         | 0     |
| 1         | 0         | 0         | 0     | 1         | 0     |
| 1         | 0         | 1         | 1     | 0         | 0     |
| 1         | 1         | 0         | 1     | 0         | 1     |
| 1         | 1         | 1         | 1     | 1         | 0     |

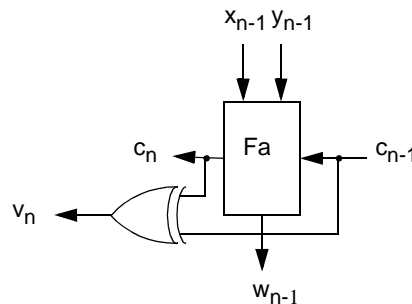


Fig. 5.9 Tabla de verdad del Full-adder de más peso de un sumador binario. Detección de resultado no representable ( $v_n$ ) cuando se opera en Ca2.

#### Ejemplo 5

Encontrad la expresión algebraica del valor  $W_s$  que obtiene el sumador binario cuando el resultado de la suma de  $X_s + Y_s + c_0$  no es representable con  $n$  bits en Ca2.

Vamos a proceder como hicimos en el caso de los números naturales, observando casos concretos de sumas con desbordamiento para el caso de  $n = 3$  (ver tabla de la figura 5.5). En la Tabla 5.1 se muestran algunos casos. En los casos en que se debería obtener un valor positivo y se obtiene uno negativo, se cumple que  $W_s = X_s + Y_s - 8$ , mientras que en los casos en que se debería obtener un valor negativo y se obtiene uno positivo, se cumple que  $W_s = X_s + Y_s + 8$ .

Tabla 5.1 Casos de desbordamiento en Ca2 con 3 bits

| $X_s$ | $Y_s$ | $X_s + Y_s$ | $W_s$ |
|-------|-------|-------------|-------|
| 3     | 3     | 6           | -2    |
| 1     | 3     | 4           | -4    |
| -4    | -4    | -8          | 0     |
| -3    | -2    | -5          | 3     |

Generalizando para  $n$  bits, tenemos que el comportamiento del sumador binario para el caso de resultado no representable en Ca2 es:

$$W_s = \begin{cases} X_s + Y_s + c_0 - 2^n & \text{si } X_s + Y_s + c_0 > 2^{n-1} - 1 \\ X_s + Y_s + c_0 + 2^n & \text{si } -2^{n-1} > X_s + Y_s + c_0 \end{cases} \quad (\text{EQ 17})$$

#### 5.4.2 Sumador binario para naturales y enteros con detección de resultado no representable

De lo visto hasta ahora en este capítulo podemos decir que el bloque combinacional de la figura 5.10 es un sumador tanto de números naturales representados en binario como de números enteros representados en Ca2. En la figura se muestra su implementación interna para  $n = 3$  bits. Para detectar cuándo el resultado no es representable con  $n$  bits, se proporciona el bit de salida  $c_n$  para el caso de sumar números en binario y el bit  $v_n$  para el caso de Ca2. A continuación, se formaliza la funcionalidad del sumador:

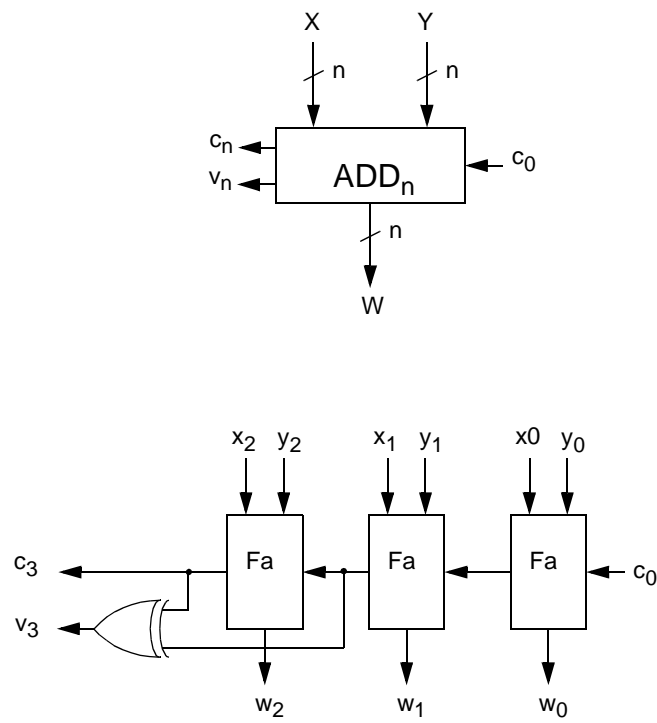


Fig. 5.10 Sumador de dos números naturales o enteros representados en binario o en Ca2 con 3 bits en Ca2 usando Full-adders con propagación del acarreo. Detección de resultado no representable con 3 bits para ambas representaciones.

$$\begin{aligned}
W_u &= \begin{cases} X_u + Y_u + c_0 & \text{si } X_u + Y_u + c_0 \leq 2^n - 1 \\ X_u + Y_u + c_0 - 2^n & \text{si } X_u + Y_u + c_0 > 2^n - 1 \end{cases} \\
W_s &= \begin{cases} X_s + Y_s + c_0 & \text{si } -2^{n-1} \leq X_s + Y_s + c_0 \leq 2^{n-1} - 1 \\ X_s + Y_s + c_0 - 2^n & \text{si } X_s + Y_s + c_0 > 2^{n-1} - 1 \\ X_s + Y_s + c_0 + 2^n & \text{si } -2^{n-1} > X_s + Y_s + c_0 \end{cases} \\
c_n &= \begin{cases} 0 & \text{si } X_u + Y_u + c_0 \leq 2^n - 1 \\ 1 & \text{si } X_u + Y_u + c_0 > 2^n - 1 \end{cases} \\
v_n &= \begin{cases} 0 & \text{si } -2^{n-1} \leq X_s + Y_s + c_0 \leq 2^{n-1} - 1 \\ 1 & \text{si } -2^{n-1} > X_s + Y_s + c_0 > 2^{n-1} - 1 \end{cases}
\end{aligned} \tag{EQ 18}$$

A los operadores que calculan el vector de bits  $W$  y los bits de condición  $c_n$  y  $v_n$ , los denotamos como:

$$W = \text{ADD}_n(X, Y, c_0) \tag{EQ 19}$$

$$c_n = \text{CARRY}_n(X, Y, c_0) \tag{EQ 20}$$

$$v_n = \text{OVF}_n(X, Y, c_0) \tag{EQ 21}$$

### Ejemplo 6

Diseñad un sumador de dos números enteros representados en Ca2 con  $n$  bits que obtenga el resultado de la suma en Ca2 con  $n+1$  bits. ¿Por qué en este sumador no se produce nunca overflow?

Porque si los operandos están representados con  $n$  bits, incluso si el acarreo de entrada vale 1, el resultado de la suma siempre se puede representar con  $n+1$  bits: si  $-2^{n-1} \leq X_s \leq 2^{n-1} - 1$  y  $-2^{n-1} \leq Y_s \leq 2^{n-1} - 1$  entonces  $-2^n \leq X_s + Y_s + 1 \leq 2^n - 1$ .

Una forma directa de diseñar el circuito consiste en usar un sumador binario de dos números de  $n+1$  bits (por ejemplo conectando  $n+1$  Full-adders con propagación del acarreo) y alimentarlo con los dos números después de extender el rango de cada uno de ellos a  $n+1$  bits. Los  $n+1$  bits de salida de este sumador representan el resultado sin overflow.

¿Hacen falta realmente  $n+1$  Full-adders, o con  $n$  Full-adders y una pequeña lógica sería suficiente? El diseño inicial puede optimizarse, ya que la salida del acarreo del Full-adder de más peso,  $c_{n+1}$ , no se usa. Una implementación con menos hardware consiste en usar un sumador de  $n$  bits e implementar directamente  $w_n = x_n \wedge y_n \wedge c_n$  mediante la expresión (ya que se ha extendido el rango de los operandos):

$$w_n = x_{n-1} \wedge y_{n-1} \wedge c_n. \quad (\text{EQ 22})$$

Si el sumador de  $n$  bits que estamos usando tiene salida de overflow, podemos usar la siguiente expresión alternativa que se obtiene a partir de EQ (22) usando las igualdades lógicas:  $c_n = 0 \wedge c_n$  y  $0 = c_{n-1} \wedge c_{n-1}$  y conociendo la expresión del overflow de la EQ (16) y la expresión del bit de suma del Full-adder de más peso:  $w_{n-1} = x_{n-1} \wedge y_{n-1} \wedge c_{n-1}$ .

$$w_n = w_{n-1} \vee v_n \quad (\text{EQ 23})$$

Dicho con palabras:  $w_n$  es igual a  $w_{n-1}$  si no hay overflow y es igual a  $v_n$  si hay overflow.

## 5.5 Cambio de signo

Con el objetivo de diseñar un circuito negador, primero vamos a encontrar el algoritmo de la operación aritmética de cambio de signo (negación) de un número entero representado en Ca2 con  $n$  bits. Esto es, vamos a encontrar las operaciones lógicas a seguir para obtener el vector  $W$  de  $n$  bits a partir de los  $n$  bits del vector  $X$ , tales que:

$$W_s = -X_s$$

### 5.5.1 Algoritmo aritmético de cambio de signo

Observando la tabla de la figura 5.11 vemos que, tanto en la columna del valor numérico representado (columna  $X_s$ ) como en su representación (columna  $X = x_2x_1x_0$ ) existe algún tipo de simetría.

- La columna  $X_s$  tiene una simetría respecto del -4. Los números simétricos son el  $X_s$  y el  $-X_s$ . Por ejemplo, el 2 se encuentra a 2 posiciones hacia arriba del -4 y el -2 a 2 posiciones hacia abajo. Esto es, cambiar de signo un número representado en Ca2 con 3 bits, tanto positivo como negativo, consiste en encontrar su simétrico en la tabla respecto de la fila 100, que representa el -4. Esto se muestra en la figura con trazo muy grueso sombreado.
- La columna  $X_s = x_2x_1x_0$  tiene una simetría respecto de la línea que separa la representación de los números positivos de los negativos. Las representaciones simétricas son la  $x_2x_1x_0$  y la  $!x_2!x_1!x_0$ . Esto se muestra en la figura con trazo grueso discontinuo. Esto es, complementar<sup>1</sup> (cambiar ceros por unos y unos por ceros) la representación de un número, tanto positivo como negativo, es equivalente a encontrar la representación simétrica respecto de la línea que separa los positivos de los negativos. Por ejemplo, complementando el 010 (que representa al 2) se obtiene el 101 (que representa el -3).

De lo anterior se deduce que para obtener la representación en Ca2 de  $-X_s$  a partir de la representación en Ca2 de  $X_s$ , tanto para valores de  $X_s$  positivos como negativos, solamente hay que complementar la representación de  $X_s$  y sumar 1 (001 para  $n=3$ ) utilizando el sumador binario. Por supuesto que al

1. Más correctamente se denomina complementar a 1.

| X              |                |                |  | X <sub>s</sub> |
|----------------|----------------|----------------|--|----------------|
| x <sub>2</sub> | x <sub>1</sub> | x <sub>0</sub> |  |                |
| 0              | 0              | 0              |  | 0              |
| 0              | 0              | 1              |  | 1              |
| 0              | 1              | 0              |  | 2              |
| 0              | 1              | 1              |  | 3              |
| 1              | 0              | 0              |  | -4             |
| 1              | 0              | 1              |  | -3             |
| 1              | 1              | 0              |  | -2             |
| 1              | 1              | 1              |  | -1             |

$!x_2$   $!x_1$   $!x_0$

$+1$

$-X_s$

Fig. 5.11 Simetría en la tabla de representación para obtener  $-X_s$  complementando el vector de bits que lo representa (cambiando ceros por unos y unos por ceros) y sumando 1 (con el sumador binario).

sumar 1 con el sumador binario nos quedamos con los 3 bits del resultado, sin tener en cuenta el acarreo de salida. La suma de +1 se muestra en la figura mediante una flecha de trazo continuo. Por ejemplo, si complementamos el 010 (que representa al 2), obtenemos el 101 y si al 101 le sumamos 001 nos da el 110, que representa el -2.

#### Ejemplo 7

Si queremos obtener la representación del  $-(-1)$  a partir de la representación del -1 (que es 111) hacemos:

- el complemento de 111 que es el 000 y
- le sumamos 1 con el sumador binario:  $001 = \text{ADD}_3(000, 001, 1)$

Efectivamente, el vector 001 representa el 1.

#### Ejemplo 8

Si efectuamos el complemento de la representación del 0 y sumamos 1 obtenemos el 0, cosa que es correcta, ya que  $-0 = +0$ :

$$\text{ADD}_3(!0 !0 !0, 0 0 1, 0) = \text{ADD}_3(1 1 1, 0 0 1, 0) = 0 0 0.$$

### 5.5.2 Detección de resultado no representable

¿Nos indica el acarreo de salida del sumador binario que ha habido overflow en la operación de cambio de signo? No, el acarreo de salida cuando trabajamos en Ca2, tanto en el cambio de signo como en la suma, no indica que el resultado no pueda representarse con n bits. El único caso en el que se produce acarreo de salida es en el cambio de signo del 0 y el resultado obtenido es correcto.

¿En qué casos se produce overflow? Al cambiar de signo al número más negativo que se puede representar con n bits ( $-2^{n-1}$ ) deberíamos obtener el  $-(-2^{n-1}) = 2^{n-1}$ , que no se puede representar

con  $n$  bits en Ca2. Éste es el único caso de overflow en la operación de cambio de signo en Ca2. Para el caso de 3 bits, si complementamos el 100 obtenemos el 011 y si le sumamos 001 obtenemos el 100 que vuelve a representar el -4.

¿Sabes encontrar ahora una expresión lógica para la detección de overflow? Se produce overflow en el único caso en que el operando tiene signo negativo y el resultado que se obtiene tiene también signo negativo. Expresiones lógicas para la detección de overflow en la negación son:

$$v_n = x_{n-1} \cdot w_{n-1} \quad (\text{EQ 24})$$

### 5.5.3 Implementación de un negador

Después de tener claro el algoritmo de cambio de signo, su implementación es sencilla. La figura 5.12 muestra un negador en Ca2 para  $n=3$  bits usando el sumador binario de la figura 5.1. Para la detección de overflow se ha usado la EQ (24).

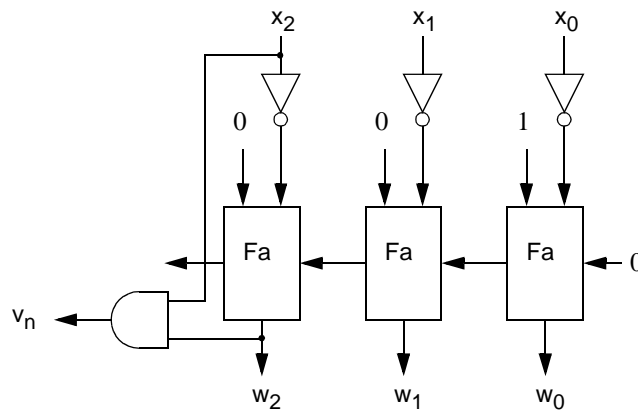


Fig. 5.12 Negador de 3 bits en Ca2 usando Full-Adders.

#### Ejemplo 9

Una expresión alternativa a la EQ (24), para la detección de overflow en la negación en Ca2 es:

$$v_n = c_n \wedge c_{n-1} \quad (\text{EQ 25})$$

Para demostrarlo, en las cuatro primeras columnas de la tabla de la figura 5.13 construimos la tabla de verdad de la rebanada de más peso del negador de la figura 5.12 ya que es este hardware (que se muestra a la derecha en la figura 5.13) el que genera las señales  $c_n$  y  $w_{n-1}$  a partir de  $x_{n-1}$ , y  $c_{n-1}$ . Puede observarse que la columna de  $v_n$  se obtiene tanto a partir de la expresión EQ (24) como de la expresión EQ (25), con lo que queda demostrado que ambas expresiones son válidas.

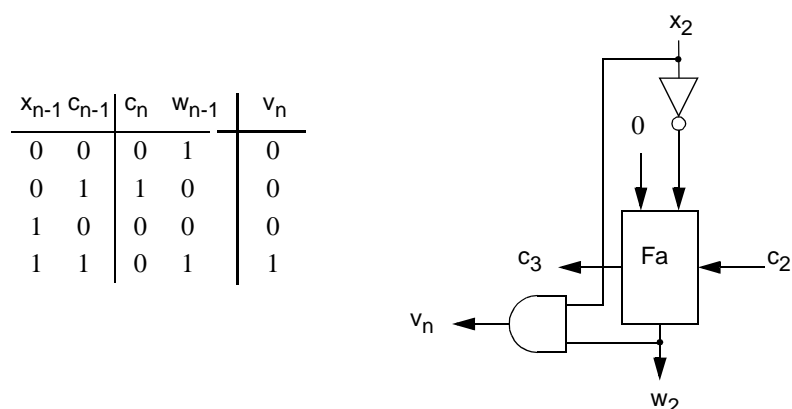


Fig. 5.13 Tabla de verdad del Full-adder de más peso del negador de la figura 5.12 para la detección de overflow y su implementación.

### Ejemplo 10

¿Hacen falta  $n$  FAs para implementar el negador? Para sumar el vector  $001$  al vector  $x_2x_1x_0$  no hace falta un sumador construido con 3 Full-Adders. En el negador de la figura 5.14 la suma del valor 1 se ha efectuado con Half-adders en vez de usar Full-adders y además se ha usado la EQ (25) para  $v_n$ .

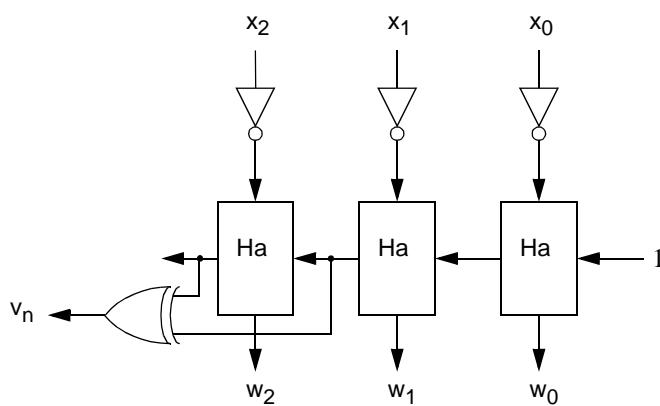


Fig. 5.14 Negador de 3 bits en Ca2 usando Half-adders.



**Ejemplo 11**

Veamos ahora cómo el cambio de signo nos puede ayudar a obtener la representación en Ca2 de un número negativo, por ejemplo el número -5, usando el siguiente algoritmo.

Primero obtendremos la representación en Ca2 del valor absoluto del número en cuestión. Como el valor absoluto es un número positivo, la representación en Ca2 es como la representación en binario, pero asegurándonos que hay un 0 como bit de más peso. Esto se hace mediante divisiones sucesivas entre 2, empezando por el número y siguiendo por los cocientes hasta que el cociente es 0 y el resto es también 0. Los sucesivos restos son los dígitos de la representación, empezando por el dígito de menos peso y terminando por el de más peso que es 0. Después aplicaremos el algoritmo de cambio de signo a la representación en Ca2. Estas divisiones se indican en la figura 5.15 (parte de la izquierda). A la izquierda de la figura se obtiene la representación del -5 a partir de la representación del 5 mediante el algoritmo de cambio de signo (cambiar ceros por unos y unos por ceros y sumar 1). El resultado es:  $1011 = ADD_3(!0 !1 !0 !1, 0001, 0)$ .

|  |   |
|--|---|
| $  \begin{array}{r}  5 \quad   \quad 2 \\  \hline  4 \quad 2 \quad   \quad 2 \\  \hline  1 \quad 2 \quad 1 \quad   \quad 2 \\  \hline  \quad \quad 0 \quad 0 \quad 0 \quad   \quad 2 \\  \hline  \quad \quad \quad 1 \quad 0 \quad 0 \\  \hline  \quad \quad \quad \quad 0  \end{array}  $ | $  \begin{array}{rcl}  5 & \Rightarrow & 0 \ 1 \ 0 \ 1 \\  & & \Downarrow \text{complementar} \\  & & 1 \ 0 \ 1 \ 0 \\  & & \Downarrow + 0001 \\  -5 & \Leftarrow & 1 \ 0 \ 1 \ 1  \end{array}  $ |
|--|---|

Fig. 5.15 Obtención de la representación en Ca2 del -5.

**Ejemplo 12**

Encontrar el número entero que se representa en Ca2 por el vector de bits  $X = 10110111$ .

Para ello hay que sustituir el valor de los bits en la fórmula de la representación, EQ (10):

$$X_s = -128 + 32 + 16 + 4 + 2 + 1 = -73 \quad (\text{EQ 26})$$

El mismo resultado también se puede obtener, al observar que el número es negativo, pues X tiene un 1 en el bit de más peso,

- cambiando de signo la representación, sin que se produzca overflow (para ello se complementa y suma 1 previa extensión de rango, si es necesario –que no es el caso del ejemplo),

$$01001001 = ADD_8(!1 !0 !1 !1 !0 !1 !1 !1, 00000001, 0) = ADD_8(01001000, 00000001, 0), \quad (\text{EQ 27})$$

- calculando el valor representado por el vector de bits resultante,

$$73 = 64 + 8 + 1, y \quad (\text{EQ 28})$$

- cambiando de signo al valor resultante,

$$-73. \quad (\text{EQ 29})$$

## 5.6 Resta

¿Cómo podemos construir un restador aprovechando al máximo los bloques aritméticos que ya hemos diseñado? Calcularemos  $W_s = X_s - Y_s$  usando la expresión equivalente  $W_s = X_s + (-Y_s)$ , ya que sabemos sumar y sabemos cambiar de signo. Para ello, usamos el negador de la figura 5.12 de la siguiente forma:

- En el Full-adder de menor peso, el bit a 1 que entra por una de las entradas superiores lo hacemos entrar ahora por la entrada de  $c_0$  (esto no es ningún problema, pues las tres entradas de un Full-adder son intercambiables).
- Por las entradas de la izquierda de los Full-adders, en vez de entrar ceros, ahora entraran los  $n$  bits de  $X$ .
- Por la entrada del negador entraremos los  $n$  bits del vector  $Y$ .

Esto es,  $W = \text{ADD}_n(X, !Y, 1)$ . La figura 5.16 muestra este diseño.

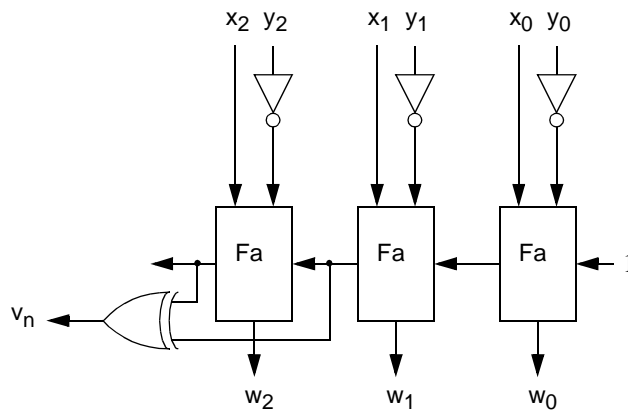


Fig. 5.16 Restador de enteros de 3 bits en Ca2 usando Full-adders con propagación del acarreo.

## 5.7 Implementación de un sumador/restador

Dado que el sumador y el restador en Ca2 son parecidos, nos planteamos construir un sumador/restador que sume o reste dos números dependiendo de si su entrada de control, que denominamos  $!s/r$ , vale 0 o 1. Partimos de los diseños del sumador y del restador con propagación del acarreo de la figura 5.16, que vamos a fundir en uno solo. Para ello tenemos que ver en primer lugar que en la función Xor de 2 bits,  $w = a \wedge b$ , la entrada  $b$  se puede interpretar como una entrada que controla la funcionalidad de la salida  $w$  en función de la entrada  $a$  de la siguiente manera:

$$w = \begin{cases} a & \text{si } b = 0 \\ !a & \text{si } b = 1 \end{cases}$$

Sustituimos, ahora, las puertas Not de la figura 5.16 por puertas Xor y conectamos la entrada de control !s/r a la entrada de control b de las puertas Xor y al acarreo de entrada del Full-adder de menos peso. Además, para que este sumador/restador sirva también para la suma y resta de números naturales codificados en binario, se ha negado el acarreo de salida en caso de resta, para obtener el borrow, y efectuar la detección de resultado no representable para la resta de naturales (como se estudió en el capítulo anterior). El sumador/restador resultante se muestra en la figura 5.17.

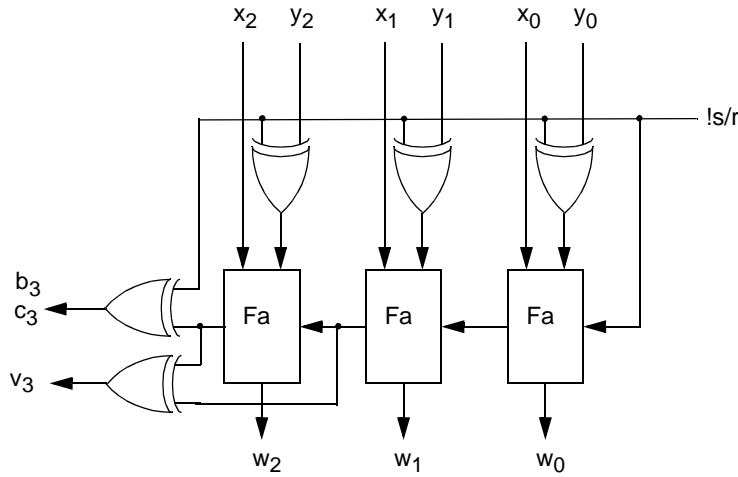


Fig. 5.17 Sumador/restador de 3 bits tanto para números naturales en binario como para enteros en Ca2 con detección de resultado no representable para las dos representaciones.

## 5.8 Multiplicación por potencias de dos

Veamos primero el algoritmo de multiplicación por 2 y luego el de multiplicación por potencias de 2.

### 5.8.1 Algoritmo aritmético de multiplicación por 2

A partir de la representación en Ca2 con  $n$  bits,  $X = x_{n-1}x_{n-2} \dots x_2x_1x_0$ , de un número entero,  $X_s$ , encontrar la representación del número  $W_s = 2 \cdot X_s$ . Dado que el rango de  $X_s$  es  $-2^{n-1} \leq X_s \leq 2^{n-1} - 1$ , el de  $W_s$  es  $-2^n \leq W_s \leq 2^n - 2$  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  $W = w_n w_{n-1} \dots w_2 w_1 w_0$  tales que  $W_s = 2 \cdot X_s$ , o lo que es lo mismo:

$$-w_n 2^n + \sum_{i=0}^{n-1} w_i 2^i = 2 \left( -w_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} w_i 2^i \right) \quad (\text{EQ 30})$$

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

$$0 \leq w_i \leq 1 \quad (\text{EQ 31})$$

Como  $2 \times 2^i = 2^{i+1}$ , la EQ (30) expandida, para que se vea más claro, y después de multiplicar todos los sumandos del sumatorio por 2, queda:

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

De aquí se ve que la siguiente solución

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

$$w_i = x_{i-1} \quad \text{para} \quad 1 \leq i \leq n,$$

cumple con la ecuación anterior, que es la EQ (30), y con la EQ (31), ya que los dígitos de X la cumplen (son bits) y  $w_0=0$  también.

Por lo tanto, **multiplicar por 2 un número entero representado en Ca2 consiste en desplazar los bits de la representación una posición a la izquierda y poner un 0 como bit de menor peso.**

### Ejemplo 13

La figura 5.18 muestra un ejemplo de aplicación del algoritmo aritmético de la multiplicación por 2 en Ca2, el operando representado con 4 bits y el resultado con 5.

| Valor        | Representación en Ca2 |       |       |       |       |
|--------------|-----------------------|-------|-------|-------|-------|
|              | Bit 4                 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| $X_s = -7$   |                       | 1     | 0     | 0     | 1     |
| $2X_u = -14$ | 1                     | 0     | 0     | 1     | 0     |

Fig. 5.18 Ejemplo de aplicación de multiplicación por 2 en Ca2.

**Detección de resultado no representable en n bits en Ca2.** 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 se obtiene del algoritmo de extensión de rango, pero aplicado para reducir el rango: si el bit n y el n-1 del resultado son distintos, o lo que es lo mismo, si el bit n-1 y el n-2 del operando son distintos, el resultado de la multiplicación por 2 no es representable en Ca2 en n bits.

### 5.8.2 Algoritmo aritmético de multiplicación por $2^k$ y su implementación

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. Esto es, desplazar  $k$  posiciones a la izquierda los bits que representan al número y poner  $k$  ceros en los  $k$  bits de menor peso. Además, el resultado no es representable en Ca2 con  $n$  bits excepto si los  $k+1$  bits de más peso del operando son todos ceros o todos unos.

#### Ejemplo 14

En la siguiente tabla se muestra la representación en Ca2 con 8 bits del número 23 y sus multiplicaciones por  $2^k$  para  $k = 1, 3$  y 5. Es de resaltar que  $23 \times 2^3$  y  $23 \times 2^5$  requieren 9 y 11 bits para representarse correctamente en Ca2: 010111000 y 01011100000, por lo que los 8 bits de menor peso, 10111000 y 11100000, no dan el resultado correcto. El resultado no es representable en 8 bits en estos dos casos porque los  $k+1$  bits de más peso del operando (00010111), para  $k$  igual a 3 y a 5 respectivamente, no son  $k+1$  ceros ni  $k+1$  unos.

| Valor:                           | $X_s = 23$ | $2X_s = 46$ | $2^3X_s = 184$               | $2^5X_s = 736$               |
|----------------------------------|------------|-------------|------------------------------|------------------------------|
| Representación con $n = 8$ bits: | 00010111   | 00101110    | 10111000<br>No representable | 11100000<br>No representable |

#### Ejemplo 15

En la siguiente tabla se muestra la representación en Ca2 con 8 bits del número -12 y sus multiplicaciones por  $2^k$  para  $k = 1, 3$  y 5 y la indicación de cuándo el resultado con 8 bits no es representable.

| Valor:                           | $X_s = -12$ | $2X_s = -24$ | $2^3X_s = -96$ | $2^5X_s = -384$              |
|----------------------------------|-------------|--------------|----------------|------------------------------|
| Representación con $n = 8$ bits: | 11110100    | 11101000     | 10100000       | 10000000<br>No representable |

### Implementación del multiplicador por $2^k$ (SL-k)

La figura 5.19 muestra el esquema lógico del circuito combinacional para multiplicar un número entero representado en Ca2 con 16 bits por  $2^k$ , para  $k=4$ . El algoritmo para los  $n$  bits del resultado es el mismo que el de multiplicación por  $2^k$  para naturales en binario. La diferencia está en la detección del resultado no representable. El resultado es representable en binario cuando el operando tiene los  $k$  bits de más peso iguales a cero mientras que en Ca2 lo es cuando los  $k+1$  bits de más peso son todos 0 o todos 1. No obstante, como en nuestro computador no vamos a detectar por hardware cuándo el resultado de las operaciones aritméticas es o no representable, ni en binario ni en Ca2, usamos el mismo dispositivo que para naturales, el SL- $k$  (*Shift Left - k*), desplazador de  $k$  bits a la izquierda, cuyo símbolo se muestra en la figura para  $k=4$ .

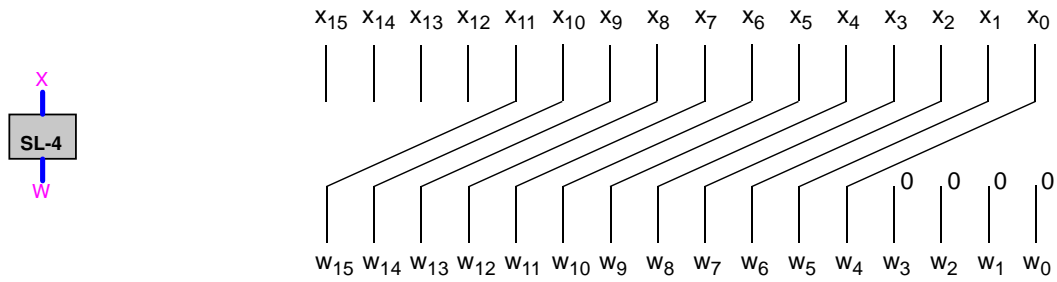


Fig. 5.19 Símbolo del desplazador de 4 bits a la izquierda (Shift Left - 4) y esquema interno para  $n=16$  (sin detección de resultado no representable ni para naturales en binario ni para enteros en complemento a dos).

## 5.9 División por potencias de dos

Vemos primero el algoritmo de división por 2 y luego el de división por potencias de 2.

### 5.9.1 Algoritmo aritmético de división por 2

A partir de la representación en Ca2 con  $n$  bits,  $X = x_{n-1}x_{n-2} \dots x_2x_1x_0$ , de un número entero,  $X_s$ , debemos encontrar la representación en Ca2 con  $n$  bits del número  $W_s = X_s/2$ . De hecho  $W_s$  se puede representar correctamente en Ca2 con  $n-1$  bits, por lo que el resultado con  $n$  bits que vamos a obtener siempre será correcto, no tiene sentido hablar de detección de resultado no representable. Así que, encontrar el algoritmo aritmético de la división por 2 en Ca2 es equivalente a encontrar los bits de  $W = w_{n-1}w_{n-2} \dots w_2w_1w_0$  tales que:

$$-w_{n-1}2^{n-1} + \sum_{i=0}^{n-2} w_i 2^i = \left( -w_{n-1}2^{n-1} + \sum_{i=0}^{n-2} w_i 2^i \right) / 2 \quad (\text{EQ 32})$$

$$\text{y que los valores } w_i \text{ sean dígitos válidos en base 2, esto es: } 0 \leq w_i \leq 1 \quad (\text{EQ 33})$$

Como  $2^i/2 = 2^{i-1}$ , la EQ (32) expandida y después de dividir todos los sumandos del sumatorio por 2, y recordando que  $x_0/2 = 0$  (división entera), queda:

$$-w_{n-1}2^{n-1} + w_{n-2}2^{n-2} + \dots + w_22^2 + w_12^1 + w_02^0 = -x_{n-1}2^{n-2} + x_{n-2}2^{n-3} + \dots + x_22^1 + x_12^0$$

Como  $2^k = 2^{k+1} \cdot 2^{-1}$ , podemos sustituir el sumando de más peso de la parte de la derecha de la igualdad anterior por

$$-x_{n-1}2^{n-2} = -x_{n-1}2^{n-1} + x_{n-1}2^{n-2}$$

Después de esta sustitución se ve que la siguiente solución,

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

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

cumple con la EQ (32) y también con la EQ (33) ya que los dígitos de X la cumplen (son bits).

Por lo tanto, **dividir por 2 un número entero representado en Ca2 consiste en desplazar los bits de la representación una posición a la derecha replicando el bit de signo (el bit de menor peso del operando desaparece).**

### Ejemplo 16

La figura 5.18 muestra un ejemplo de aplicación del algoritmo aritmético de la división por 2 en Ca2.

| Valor         | Representación en Ca2 |       |       |       |       |
|---------------|-----------------------|-------|-------|-------|-------|
|               |                       | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| $X_s = 7$     |                       | 0     | 1     | 1     | 1     |
| $X_s / 2 = 3$ |                       | 0     | 0     | 1     | 1     |

Fig. 5.20 Ejemplo de aplicación de división por 2 en Ca2.

### Ejemplo 17

La figura 5.18 muestra un ejemplo de aplicación del algoritmo aritmético de la división por 2 en Ca2.

| Valor          | Representación en Ca2 |       |       |       |       |
|----------------|-----------------------|-------|-------|-------|-------|
|                |                       | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
| $X_s = -7$     |                       | 1     | 0     | 0     | 1     |
| $X_s / 2 = -4$ |                       | 1     | 1     | 0     | 0     |

Fig. 5.21 Ejemplo de aplicación de división por 2 en Ca2.

Es muy importante darse cuenta de qué tipo de división entera estamos realizando. En las deducciones anteriores hemos dicho que  $x_0/2 = 0$ .  $x_0$  es el resto positivo de dividir  $X_s$  entre 2. Cuando  $X_s$  es un número positivo (como en el ejemplo 16) el resultado que obtenemos es el de la parte entera por abajo

del resultado, o lo que se denomina el primer entero más próximo hacia el 0. Pero cuando  $X_s$  es negativo (como en el ejemplo 17) es el entero más próximo hacia  $-\infty$ . En cualquier caso es el entero más próximo hacia  $-\infty$ . Dicho de otra forma, el algoritmo de división por 2 que hemos encontrado calcula el cociente entero tal que el resto sea siempre positivo, sea cual sea el signo del operando. Si al dividir -7 entre 2 obtenemos el cociente -3 es porque el resto es -1, ya que de la prueba de la división:  $-7 = -3 \times 2 - 1$ . Sin embargo, si tal como hacemos aquí, al dividir -7 entre 2 nos da el cociente -4 es porque el resto es 1, ya que:  $-7 = -4 \times 2 + 1$ .

### 5.9.2 Algoritmo aritmético de división por $2^k$ y su implementación

Dado que dividir un número por  $2^k$  consiste en dividirlo por 2 repetidamente  $k$  veces, el algoritmo aritmético consiste en aplicar el algoritmo de división por dos  $k$  veces consecutivas. Esto es, desplazar  $k$  posiciones a la derecha los bits que representan al número y poner el bit de signo del operando en las  $k$  posiciones de más peso del resultado. Además, el resultado siempre es representable en los mismos bits que el operando.

#### Ejemplo 18

En la siguiente tabla se muestra la representación en Ca2 con 8 bits del número 23 y sus divisiones por  $2^k$  para  $k = 1, 3$  y 5. Al ser un número entero positivo en Ca2 el algoritmo y la representación del operando y del resultado son los mismos que en binario.

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

#### Ejemplo 19

En la siguiente tabla se muestra la representación en Ca2 con 8 bits del número -13 y sus divisiones por  $2^k$  para  $k = 1, 3$  y 5 (la división es al entero más próximo hacia  $-\infty$ ).

| Valor:                           | $X_s = -13$ | $X_s / 2 = -7$ | $X_s / 2^3 = -2$ | $X_s / 2^5 = -1$ |
|----------------------------------|-------------|----------------|------------------|------------------|
| Representación con $n = 8$ bits: | 11110011    | 11111001       | 11111110         | 11111111         |

### Implementación del divisor por $2^k$ (SRA-k)

La figura 5.19 muestra el circuito combinacional para dividir un número entero representado en Ca2 con 16 bits por  $2^k$ , para  $k=4$ . La salida se forma desplazando a la derecha  $k$  bits la entrada y extendiendo el bit de signo a los  $k$  bits de más peso. El resultado siempre es representable. El símbolo de este bloque se muestra a la izquierda de la figura y se denomina, SRA- $k$  (*Shift Right Arithmetically - k*): desplazador aritmético de  $k$  bits a la derecha (para diferenciarlo del desplazador lógico a la derecha, SRL, que usamos para dividir por potencias de 2 números naturales representados en binario).



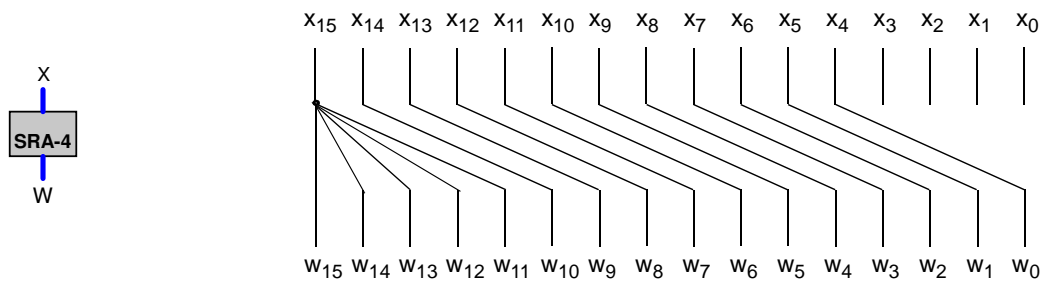


Fig. 5.22 Símbolo del desplazador aritmético de 4 bits a la derecha (Shift Right Arithmetically - 4) y esquema interno para  $n=16$ .

## 5.10 Comparación de números enteros

Al igual que hicimos con los números naturales representados en binario, para comparar los dos números enteros  $X_S$  e  $Y_S$  representados en Ca2 con  $n$  bits por  $X$  e  $Y$ , realizamos la resta de los dos números y analizamos el resultado. Como la ecuación  $X_S < Y_S$  es equivalente a la ecuación  $X_S - Y_S < 0$ , decimos que  $X_S$  será menor que  $Y_S$  cuando  $X_S - Y_S$  sea negativo.

Hay que recordar que el restador obtiene el vector de  $n$  bits  $W$ , que lo forman los  $n$  bits de menor peso del resultado correcto de la resta. El restador también obtiene el bit de overflow,  $v$ , que se activa cuando  $W$  no representa el resultado correcto  $W_S \neq X_S - Y_S$ .

Cuando  $v$  vale 0 el resultado en  $W$  es correcto, luego  $W_S = X_S - Y_S$ . En este caso, si  $W_S$  es negativo, o lo que es lo mismo si el bit de signo de  $W$ ,  $w_{n-1}$ , es 1 entonces  $X_S$  es menor que  $Y_S$ .

Cuando  $v$  vale 1 es porque  $X_S - Y_S$  es un número negativo y  $W$  representa a un positivo o porque  $X_S - Y_S$  es un número positivo y  $W$  representa a un negativo. Así que, cuando el resultado de la resta no es representable en  $n$  bits, o lo que es lo mismo cuando  $v$  vale 1, para saber el signo del resultado correcto hay que emplear este algoritmo: si el bit de signo de  $W$ ,  $w_{n-1}$ , es 1 el resultado correcto es positivo y si el bit de signo es 0 el resultado correcto es negativo.

De los dos últimos párrafos se deduce que, denominando  $s$  al bit de signo de  $W$ ,  $w_{n-1}$ ,  $X_S$  es menor que  $Y_S$  cuando vale 1 la expresión lógica siguiente:

$$!v \cdot s + v \cdot !s$$

Esta es la expresión de la operación Xor de  $v$  y  $s$ :  $v \wedge s$ .

La figura 5.23a muestra el bloque LT que tiene dos buses de  $n$  bits,  $X$  e  $Y$ , y cuya salida de un bit  $w$  vale 1 cuando  $X_s$  es menor que  $Y_s$  junto con el circuito interno del bloque. Hay que resaltar que el resultado de la comparación es siempre correcto, aunque no lo sea el resultado de la resta.

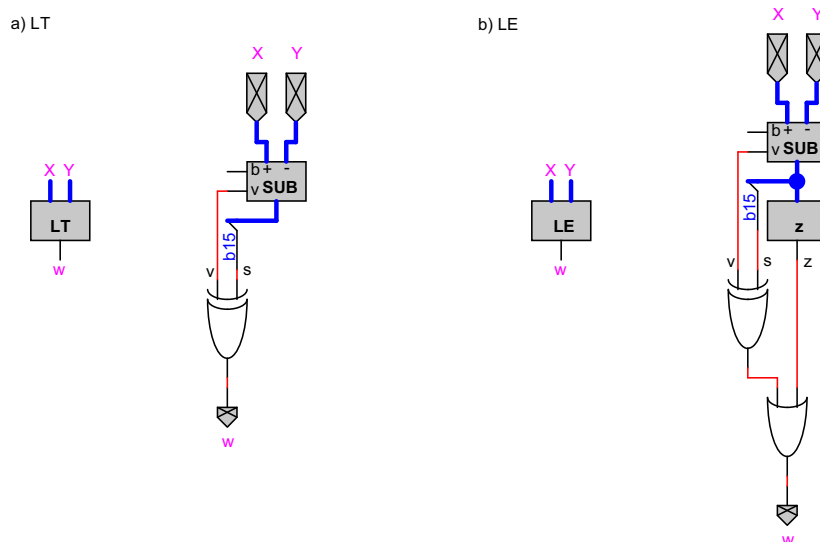


Fig. 5.23 Símbolo y esquema interno de los comparadores de números enteros en Ca2: a) LT y b) LE.

En la figura 5.23b se muestra el bloque LE con sus dos buses de entrada de  $n$  bits y su salida de un bit  $w$ . En este caso,  $w$  vale 1 cuando  $X_s$  es menor o igual que  $Y_s$ . Si denominamos  $z$  a la variable lógica que vale 1 cuando  $W_s$  es 0, la expresión lógica que vale 1 cuando  $X_s$  es menor o igual que  $Y_s$  es:

$$(v \wedge s) + z$$

El circuito interno del bloque se forma añadiendo al circuito LT un bloque Zero, para obtener  $z$  y una puerta Or hacer la suma con la salida del bloque LT ( $v \wedge s$ ), como se ve en la figura 5.23b.

Tanto para el bloque LT como para el LT hemos usado un restador con salida de borrow,  $b$ , y de overflow,  $v$ . En estos diseños la salida de borrow no se usa ya que no estamos considerando números naturales sino enteros, por lo que usamos la salida de overflow,  $v$ .