3 BLOQUES ARITMÉTICOS Y CODIFICACIÓN NUMÉRICA

Las operaciones aritméticas con números binarios pueden ser expresadas en forma de funciones booleanas. El resultado de la suma de dos números de n dígitos puede llegar a tener n+1 dígitos, de forma que el correspondiente sumador incluirá n+1 funciones booleanas; lo mismo sucederá con la resta, a cuyo resultado (de n bits) habrá que añadir un bit suplementario (n+1) para informar si la resta es viable o si el sustraendo es mayor que el minuendo. En cambio, la comparación de números de n bits requiere solamente tres funciones booleanas: igual, mayor, menor.

Suma, resta y comparación pueden ser construidas modularmente, mediante celdas básicas que realicen la operación para un solo dígito. Las celdas básicas de la suma y la resta presentarán dos salidas relativas al resultado y al posible acarreo (me llevo) y dispondrán de las dos entradas correspondientes a los operandos y una entrada adicional para el bit de acarreo o «arrastre», la cual permite la conexión de celdas en cadena para formar bloques de mayor número de bits.

Asimismo, las entradas de «arrastre» permiten conectar sucesivos bloques sumadores, restadores o comparadores para operar con mayor número de dígitos. En el caso de los comparadores son necesarias dos entradas de arrastre (igual, mayor) para recibir la información de la celda o del bloque anterior.

La multiplicación, al no ser operación lineal, no puede construirse mediante celdas básicas en cadena; pero puede configurarse mediante una combinación matricial de operaciones "y" (producto) y celdas sumadoras.

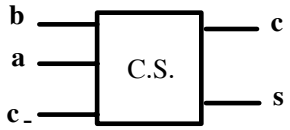
También podemos considerar como bloques operacionales los conjuntos de puertas lógicas que realizan la correspondiente operación booleana entre dos palabras, bit a bit. Las siglas ALU (unidad lógica y aritmética) designan a un bloque «multioperación», capaz de realizar diferentes operaciones aritméticas y lógicas sobre dos números de n dígitos; unas entradas de selección determinan la operación a realizar en cada momento.

A la par de las propias operaciones aritméticas, resulta oportuno considerar cómo pueden representarse en palabras binarias los números negativos o los que tienen parte decimal (números no enteros), de forma que puedan efectuar sus operaciones con los mismos bloques aritméticos. Además de la forma binaria directa, correspondiente a la numeración en base 2, en muchas ocasiones resulta conveniente conservar la estructura decimal habitual (base 10); para ello, en lugar de convertir a base 2 el número completo, se traslada a binario cada una de sus cifras individuales: esta forma de representar los números en palabras binarias recibe el nombre de BCD (decimal codificado en binario).

Tomás Pollán Santamaría. Tecnología Electrónica. E.U.I.T.I.Z. Universidad de Zaragoza. tpollan@posta.unizar.es

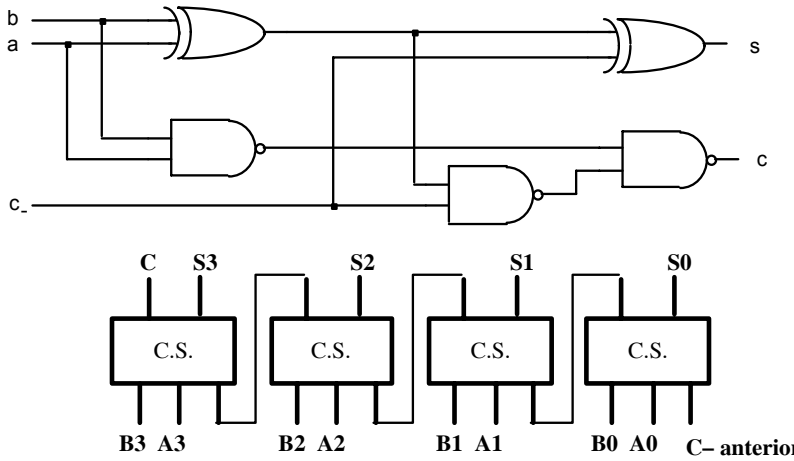
3.1. Operaciones aritméticas: suma, resta, comparación y producto

A partir de la celda sumadora básica de dos dígitos **a** y **b** más un tercero de acarreo (me llevo: *carry*) **c**, puede configurarse un *sumador* de dos números binarios de **n** dígitos mediante la conexión en cadena de **n** celdas. [El diseño de una celda sumadora se encuentra detallado en el capítulo anterior (epígrafe II.3.1) como ejercicio de síntesis de funciones booleanas.]



$$s = \overline{c_-}.(\overline{b}.a + b.\overline{a}) + c_-.(\overline{b.a} + b.a) = c_- \oplus (b \oplus a)$$

$$c = b.a + c_-.(b \oplus a) = (b * a) * \left[c_- * (b \oplus a)\right]$$



Sumador de dos números de 4 dígitos.

A su vez, estos bloques sumadores de **n** cifras pueden conectarse en cadena, con la salida **C** de cada uno de ellos unida a la entrada **C-** del siguiente más significativo, para formar sumadores de mayor número de cifras binarias.

En la celda sumadora, la función arrastre **C** obtenida mediante un mapa de Karnaugh queda un poco más simplificada que la anterior:

$$c = b.a + c_-.(b + a)$$

esta expresión simplificada nos interesa para compararla con la que resulta en el caso de una celda restadora.

En los sumadores de muchos bits, los tiempos de propagación de las celdas sucesivas se suman y resulta un tiempo de propagación global alto, lo cual puede limitar mucho la velocidad de estos sumadores. El apéndice **A2** muestra otra forma de diseño de sumadores aprovechando la recursividad en el cálculo del arrastre: arrastre anticipado (*look ahead carry*) en lugar del arrastre propagado celda a celda (*ripple carry*).

Los *restadores* se configuran con la misma técnica modular: a partir de la tabla funcional correspondiente , se obtienen las funciones booleanas que configuran la célula básica restadora (para **A-B**); las variables de entrada de la tabla funcional son **a** minuendo, **b** sustraendo y **c-** arrastre y para la obtención de la misma ha de tenerse en cuenta que el arrastre de la resta se suma al sustraendo:

a	**b**	**c-**	suma: **s**	acarreo: **c+**
0	**0**	**0**	**0**	**0**
0	**0**	**1**	**1**	**1**
0	**1**	**0**	**1**	**1**
0	**1**	**1**	**0**	**1**
1	**0**	**0**	**1**	**0**
1	**0**	**1**	**0**	**0**
1	**1**	**0**	**0**	**0**
1	**1**	**1**	**1**	**1**

$$r = \overline{c_-}.(\bar{b}.a + b.\bar{a}) + c_-.(\overline{\bar{b}.\bar{a}} + b.a) = c_- \oplus (b \oplus a)$$

$$c = b.\bar{a} + c_-.(b + \bar{a})$$

La primera función coincide con la que calcula el resultado de la suma **s** y la segunda difiere del arrastre de la suma solamente en la negación de la variable **b**.

Para restar dos números positivos **A-B** es preciso que **A** $\geq$ **B**; en otro caso, el arrastre más significativo valdrá **1**, indicando que **B** es mayor que **A** y, por tanto, el resultado no corresponde a un número positivo. [En el apartado 4 de este mismo capítulo se estudiará la representación y forma de operar con números negativos.]

Comparando las funciones básicas de sumadores y restadores se observa que difieren solamente en las relativas al arrastre y, en éstas, solamente en la negación de una variable:

arrastre de la suma $c = b.a + c_-.(b + a)$

arrastre de la resta $c = b.\bar{a} + c_-.(b + \bar{a})$

Resulta sencillo construir bloques sumadores/restadores, capaces de efectuar las dos operaciones, con una entrada de selección **d** que diferencie entre ambas; bastará añadir una operación "**o-exclusiva**" $a \oplus d$, para determinar cuándo la variable **a** debe actuar afirmada (**d = 0**: suma) o negada (d **= 1**: resta) en las funciones que calculan los arrastres:

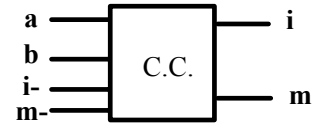
resultado $r = c_- \oplus (b \oplus a)$

arrastre $c = b.a' + c_-.(b + a')$ siendo $a' = a \oplus d$

Un *comparador* entre dos números binarios de **n** dígitos puede construirse conectando en cadena **n** celdas: la celda básica comparadora tiene como entradas, además de los dígitos correspondientes de los números a comparar $\mathbf{a_i}$ $\mathbf{b_i}$, las que le informan sobre si los conjuntos de dígitos anteriores (de menor valor significativo) son iguales (**i-**) y sobre si es mayor el que corresponde al número **A** (**m-**), y tiene dos salidas que indican igualdad **i** y "**A** mayor que **B**" **m**, respectivamente. La tercera función de comparación "**A** menor que **B**" resulta redundante con las otras dos: $\text{A menor que B} = \overline{i + m} = \bar{i}.\bar{m}$.

Las funciones booleanas de la célula comparadora responden al siguiente razonamiento [véase el ejemplo de comparación de números de dos cifras detallado en II.3.2]:

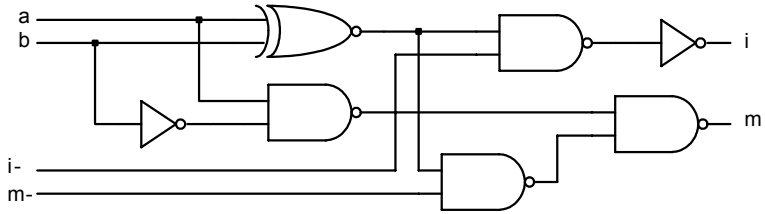
- para comparar dos números hay que comenzar por la cifra más significativa y, caso de que ambos dígitos sean iguales, irse desplazando hacia la derecha efectuando la comparación cifra a cifra;
- la función igualdad de una celda comparadora se activará cuando sus dos dígitos **a b** son iguales (es decir, cuando ambos valen **0** o ambos valen **1**: $\bar{a}.\bar{b} + a.b$) y, además, la información que reciba por su entrada **i-** respecto a los siguientes dígitos sea de que son iguales (**i- = 1**);
- la función "**A** mayor que **B**" se activará cuando **a = 1** y **b = 0** o cuando ambos dígitos son iguales ($\bar{a}.\bar{b} + a.b$) y la información que reciba por su entrada **m-** respecto a los siguientes dígitos sea de que "**A** mayor que **B**" (**m- = 1**)**.**



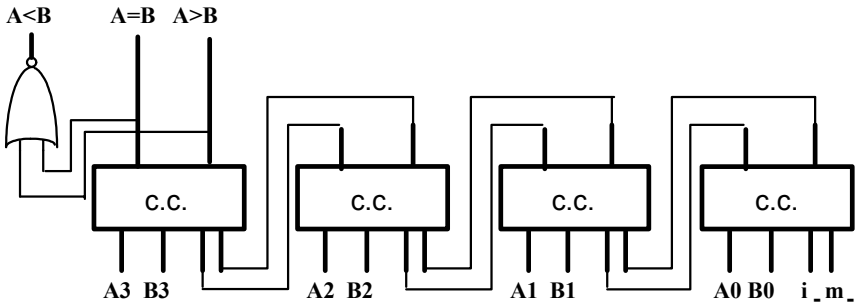
$$i = (\bar{b}.\bar{a} + b.a).i_- = (b \otimes a).i_-$$

$$m = \bar{b}.a + (\bar{b}.\bar{a} + b.a).m_-$$

$$= (\bar{b} * a) * [(b \otimes a) * m_-]$$



Agrupando **n** células comparadoras «en cadena» (las salidas **i m** unidas a las entradas **i- m-** de la celda siguiente) se obtendrá un comparador de dos números de **n** dígitos [véase figura en la página siguiente].



Comparador de dos números de 4 dígitos.

[La salida global A<B se activa cuando las otras dos A=B y A>B se encuentran a **0**.]

Asimismo, los bloques comparadores de números de **n** cifras pueden conectarse en cascada para configurar comparadores de mayor número de cifras binarias: las salidas **i** (=) y **m** (>) de cada uno de ellos unidas a las correspondientes entradas **i-** y **m-** del siguiente bloque más significativo; las entradas **i-** y **m-** del primero de los bloques (el menos significativo) deberán ir conectadas a **1** y **0**, respectivamente (situación inicial: ser iguales).

Un bloque aritmético de mayor complejidad es el *multiplicador* que realiza el producto de dos números binarios de **n** dígitos: ni su diseño ni su utilización son modulares (por células en cadena) sino que es preciso configurar el desarrollo completo de las correspondientes funciones booleanas.

Un bloque multiplicador de 2 números de **n** dígitos dispondrá de **2n** entradas (**n** para cada número) y **2n** salidas, pues tal puede ser el número de dígitos del resultado; el producto de dos números binarios puede organizarse como suma de productos de sus dígitos en la misma forma que el producto decimal:

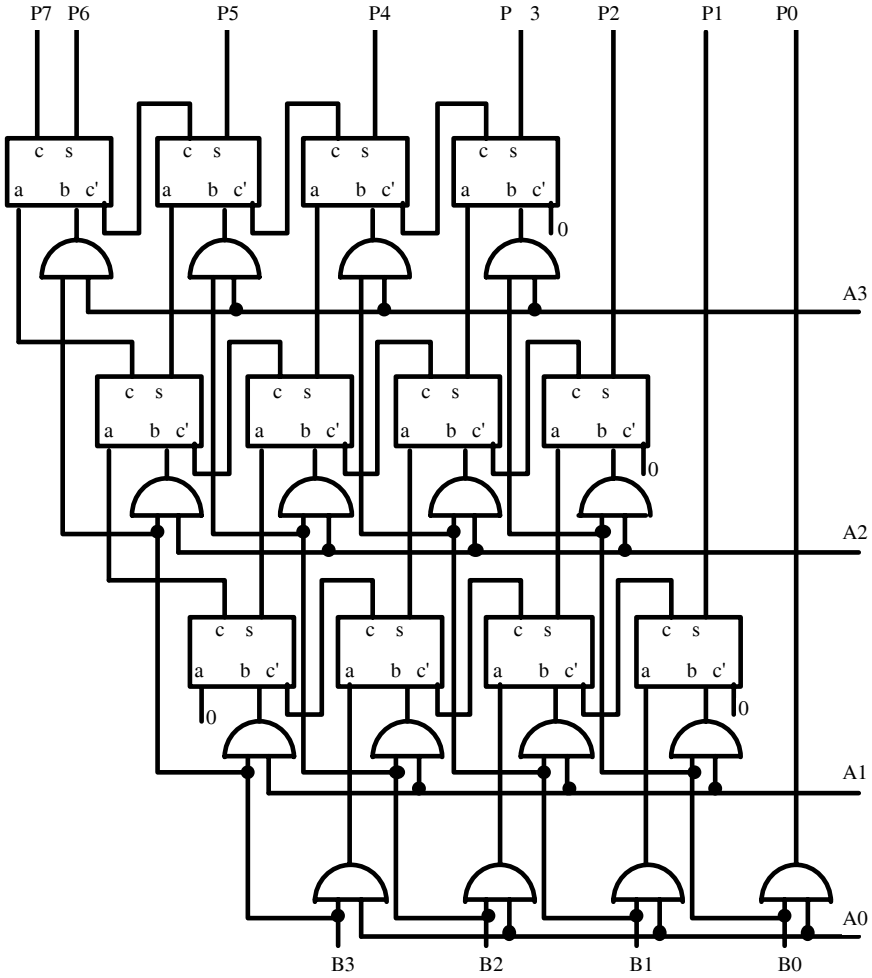
			b_{n-1}	a_{n-2}	……	b_4	b_3	b_2	b_1	b_0
x			a_{n-1}	a_{n-2}	……	a_4	a_3	a_2	a_1	a_0
			$b_{n-1}.a_0$	$b_{n-2}.a_0$	……	$b_4.a_0$	$b_3.a_0$	$b_2.a_0$	$b_1.a_0$	$b_0.a_0$
		$b_{n-1}.a_1$	$b_{n-2}.a_1$	$b_{n-3}.a_1$	……	$b_3.a_1$	$b_2.a_1$	$b_1.a_1$	$b_0.a_1$	
	$b_{n-1}.a_2$	$b_{n-2}.a_2$	$b_{n-3}.a_2$	$b_{n-4}.a_2$	……	$b_2.a_2$	$b_1.a_2$	$b_0.a_2$		
$b_{n-1}.a_3$	$b_{n-2}.a_3$	$b_{n-3}.a_3$	$b_{n-4}.a_3$	$b_{n-5}.a_3$	……	$b_1.a_3$	$b_0.a_3$			
…….....										
…….....										
…….....										

Cada una de las columnas contribuye a configurar el correspondiente dígito como suma de productos de los dígitos de los factores:

$$a_i.b_0 + a_{i-1}.b_1 + a_{i-2}.b_2 + \ldots + a_2.b_{i-2} + a_1.b_{i-1} + a_0.b_i \ ;$$

pero, además, es preciso tener en cuenta los arrastres (*me llevo*) procedentes de las sumas que configuran los dígitos anteriores.

En conjunto, para multiplicar números de 4 dígitos resulta el esquema de la figura siguiente.



Multiplicador de dos números de 4 dígitos.

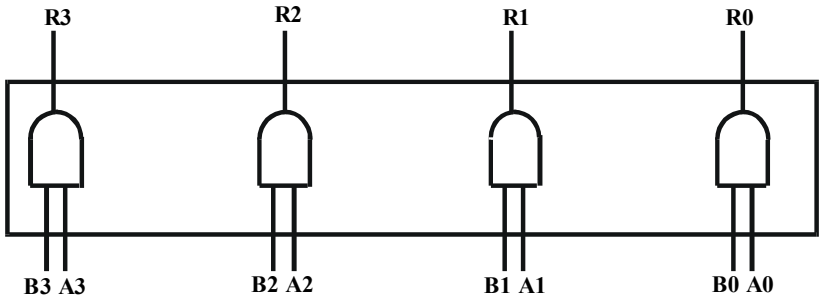
La configuración anterior, con una disposición de tipo matricial de puertas "**y**" y celdas sumadoras, puede ampliarse directamente a mayor número de dígitos ya que su estructura básica es repetitiva: basta seguir añadiendo filas y columnas de puertas "**y**" y celdas sumadoras, siguiendo la misma configuración matricial.

Pero no puede separarse en módulos iguales interconectables, de forma que el multiplicador para números de **2n** dígitos no puede realizarse mediante la simple conexión de varios multiplicadores de **n** dígitos. [Un multiplicador de números de 8 bits requiere 4 multiplicadores de 4 bits más 5 sumadores de números de 4 bits.]

3.2. Unidad lógica y aritmética ALU

También podemos considerar como bloques operacionales a los que realizan operaciones booleanas, constituidos por conjuntos de **n** puertas iguales de dos entradas que efectúan la operación lógica correspondiente (sea esta operación "**o**", "**y**", "**o-exclusiva**", "**y-negada**", "**o-negada**",...) sobre dos palabras de **n** dígitos o bits, operación lógica que se realiza individualmente, bit a bit.

En la siguiente figura, se representa uno de estos bloques, configurado por un conjunto de 4 puertas que realizan la operación "**y**" entre dos palabras de 4 bits.



Bloque operacional lógico ("**y**") para palabras de 4 bits

Disponemos, pues, de bloques con capacidad de efectuar las operaciones básicas de cálculo binario (suma, resta, comparación,...) y de otros capaces de hacer operaciones booleanas ("+", ".", "⊕", "∗", "Δ",...) sobre dos números o palabras binarias de **n** dígitos o bits.

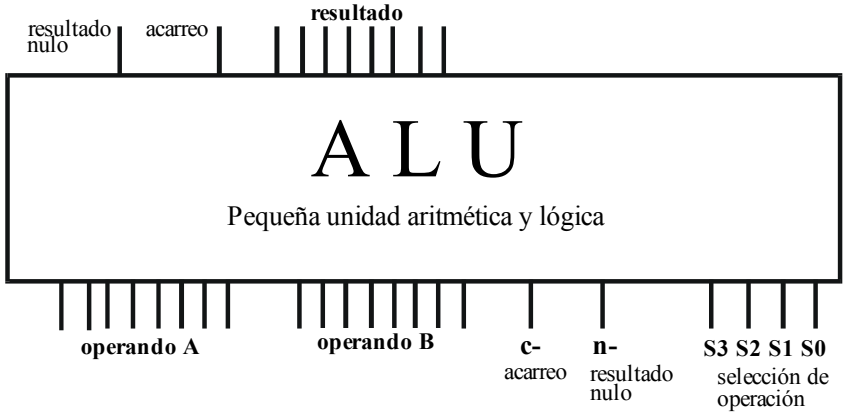
La longitud de palabra de los bloques operacionales integrados **n** suele ser de 4, 8 o 16 bits; pero este número no es limitativo en modo alguno pues, salvo el caso particular de producto, la longitud de los operandos se puede ampliar indefinidamente sin más que conectar en cadena bloques iguales:

- en el caso de suma o resta, la interconexión entre ellos se refiere al bit de acarreo
- en la comparación, cada bloque ha de comunicar al siguiente sus salidas ("=" y ">")
- en las operaciones lógicas, no es precisa interconexión alguna (son bit a bit).

Un paso más en la complejidad y posibilidades funcionales de los bloques operacionales es la configuración de un «operador genérico», capaz de realizar no una sino toda una amplia serie de operaciones aritméticas y lógicas sobre dos palabras de **n** bits: un bloque multifunción diseñado para efectuar **k** operaciones distintas, de forma que en cada momento se selecciona la operación que interesa mediante unas entradas de control.

Un bloque digital de este tipo recibe el nombre genérico de *unidad aritmética y lógica* **ALU**. Recibirá como entradas dos operandos de **n** dígitos y los terminales de salida de las diversas operaciones serán únicos, apareciendo sobre ellos el resultado de la operación seleccionada.

La siguiente figura representa una **ALU** de tipo sencillo que opera sobre palabras de 8 bits y tiene capacidad para 16 operaciones (4 entradas de control).



Unidad lógica y aritmética **ALU** para palabras de 8 bits

Dos salidas adicionales y sus correspondientes entradas permiten la ampliación de la **ALU** para palabras de más de 8 dígitos: la entrada y salida de acarreo **c** para la suma y la resta y la entrada y salida de resultado nulo **n**, que se activa cuando todos los dígitos del resultado son **0**.

La operación de comparación (entre ambos números **A** y **B**) se realiza mediante una operación de resta **A-B**, quedando el resultado reflejado sobre las salidas **n** (resultado nulo) y **c** (acarreo), en la forma siguiente:

- **n = 1** $\Rightarrow$ **A = B**
- **c = 1** $\Rightarrow$ **A < B**
- **n = 0** y **c = 0** $\Rightarrow$ **A > B**

La siguiente tabla detalla posibles operaciones a realizar por una **ALU**: la entrada de selección S_3 distingue entre operaciones aritméticas $S_3 = 0$ y operaciones lógicas $S_3 = 1$, mientras que los ocho valores posibles de las otras tres entradas de control $S_2\ S_1\ S_0$ permiten seleccionar ocho operaciones de cada uno de ambos tipos:

			$S_3 = 0$	$S_3 = 1$
S_2	S_1	S_0	**Operaciones Aritméticas**	**Operaciones Lógicas**
0	0	0	**Sumar A y B (B + A)**	**Invertir A** ($\overline{A}$)
0	0	1	**Incrementar A (A + 1)**	**Invertir B** ($\overline{B}$)
0	1	0	**Incrementar B (B + 1)**	**"o"** ($A+B$)
0	1	1	**Restar A y B (A - B)**	**"y"** ($A.B$)
1	0	0	**Decrementar A (A - 1)**	**"o-negada"** ($A\Delta B$)
1	0	1	**Decrementar B (B - 1)**	**"y-negada"** ($A*B$)
1	1	0	**Negativo de A (0 - A)**	**"o-exclusiva"** ($A\oplus B$)
1	1	1	**Negativo de B (0 - B)**	**"y-inclusiva"** ($A\otimes B$)

3.3. Codificación de números en binario

El sistema binario de numeración, con base 2, constituye la forma natural de codificar números en palabras binarias, aptas para ser procesadas por los sistemas digitales; dicho sistema de numeración constituye la base de la aritmética digital.

Este apartado se dedica a la representación de números negativos y de números con parte decimal en forma de palabras binarias (siendo el **0** y el **1** los únicos símbolos permitidos). Se trata de utilizar un convenio de representación que sea compatible con las operaciones de suma y resta de los números enteros, de forma que puedan utilizarse los bloques operacionales tal como han sido construidos en este capítulo.

3.3.1. Codificación binaria de números negativos

Es posible distinguir los números positivos y negativos mediante un bit inicial de signo: **0** para los números positivos y **1** para los negativos. Una primera forma de codificar los números enteros consistirá en añadir un **0** delante del número binario que expresa el valor absoluto para indicar número positivo y añadir un **1** delante del mismo valor absoluto para representarlo en negativo.

Esta codificación no es compatible con las operaciones de suma y resta tal como han sido construidas para los números naturales. Consideremos números binarios inferiores al número decimal 100, cuyo valor absoluto se expresa en 7 bits y utilicemos un octavo bit inicial para el signo:

```
          +22      0 0 0 1 0 1 1 0
          -22      1 0 0 1 0 1 1 0
          ---------------------------------
suma              1 0 1 0 1 1 0 0   =  -44   erróneo
```

Si representamos en una lista los diversos números enteros de 8 bits resultantes de esta codificación (signo - valor absoluto), ordenados según su valor numérico:

```
0 1 1 1 1 1 1 1     +127
0 1 1 1 1 1 1 0     +126
---------------
0 0 0 0 0 0 1 0       +2
0 0 0 0 0 0 0 1       +1
0 0 0 0 0 0 0 0        0
1 0 0 0 0 0 0 1       -1
1 0 0 0 0 0 1 0       -2
---------------
1 1 1 1 1 1 1 0     -127
1 1 1 1 1 1 1 1     -128
```

podemos comprobar que en los números negativos no se conserva el «contaje» binario (a, a+1): el -1 no es el siguiente binario del -2 (**10000010** + 1 ≠ **1000001**); al no conservarse el contaje, tampoco se conservan las operaciones suma y resta, en su forma normal (pues suma y resta no son sino el resultado de contajes/descontajes sucesivos). De hecho los números negativos resultan ordenados en sentido inverso, respecto al contaje.

Interesa una codificación que resulte compatible con el contaje y, por ello, con las operaciones aritméticas de suma y resta tal como son realizadas por los bloques operacionales. Para ello, representemos la lista de los números enteros de 8 bits, construyendo los negativos mediante la operación de restar **1** al anterior (descontaje):

```
0 1 1 1 1 1 1 1     +127
0 1 1 1 1 1 1 0     +126
---------------
0 0 0 0 0 0 1 0       +2
0 0 0 0 0 0 0 1       +1
0 0 0 0 0 0 0 0        0
1 1 1 1 1 1 1 1       -1
1 1 1 1 1 1 1 0       -2
---------------
1 0 0 0 0 0 0 1     -127
1 0 0 0 0 0 0 0     -128
```

Esta representación equivale a obtener el negativo de un número mediante la operación **y = 0 – x** (olvidando el arrastre que se produce en el bit más significativo):

	0	**0 0 0 0 0 0 0 0**
	+22	**0 0 0 1 0 1 1 0**
		---
resta		**1 1 1 0 1 0 1 0**
	-22 **=**	**1 1 1 0 1 0 1 0**

Comprobemos que esta codificación es compatible con las operaciones de suma y resta:

+22	**0 0 0 1 0 1 1 0**		+22	**0 0 0 1 0 1 1 0**	
-22	**1 1 1 0 1 0 1 0**		-22	**1 1 1 0 1 0 1 0**	
---			---		
suma	**0 0 0 0 0 0 0 0** = 0		resta	**0 0 1 0 1 1 0 0** = +44	*correcto*

(en ambos casos se ha ignorado el arrastre que se produce en el bit más significativo, es decir, los dígitos que van más allá del bit de signo).

Esta codificación se denomina en *complemento a 2ⁿ*, o más abreviadamente, *complemento a 2*; en ella el **0** equivale a $\mathbf{2^n}$, siendo **n** el número de bits de la palabra, y los números se cambian de signo restándolos de $\mathbf{2^n}$:

$$\mathbf{-A = 2^n - (A) = 100000...}\text{(n «ceros»)} \mathbf{- (A)}$$

La anterior sustracción puede ser realizada con mayor facilidad en dos pasos, en la forma que sigue:

$$\mathbf{2^n - (A) = 11111...}\text{(n «unos»)} \mathbf{+ 1 - (A) = A' + 1}$$

donde **A' = 11111...** (n «unos») **- (A)** es precisamente el número que resulta de invertir cada uno de los bits de **A**: $A' = \overline{A}$.

Ejemplo de complemento a 2:

83	=	***0 1 0 1 0 0 1 1***	*se invierten los bits*
83'	=	***1 0 1 0 1 1 0 0***	*+1*
- 83	=	***1 0 1 0 1 1 0 1***	

$2^8 - 83 = 100000000 - 01010011 = 10101101.$

La conversión inversa, de negativo a positivo, se realiza mediante el mismo proceso:

- 83	=	***1 0 1 0 1 1 0 1***	*se invierten los bits*
- 83'	=	***0 1 0 1 0 0 1 0***	*+1*
+ 83	=	***0 1 0 1 0 0 1 1***	

$2^8 - (-83) = 100000000 - 10101101 = 01010011.$

Habida cuenta de que la codificación en *complemento a 2* conserva el contaje binario, también resulta compatible con las operaciones de suma y resta aritméticas definidas para los números naturales.

Un número de **n** dígitos *complemento a 2* estará comprendido entre **1000...** = -2^{n-1} (el menor número negativo de **n** bits) y **0111... = +** $2^{n-1} - 1$ (el mayor número de **n** bits).

Existen casos en que el valor absoluto del resultado no cabe en los **n-1** bits disponibles para ello: se produce entonces un *desbordamiento* (*over-flow*) y el resultado no es correcto ya que requiere mayor número de dígitos. En estos casos el valor numérico interfiere con el bit de signo y genera un resultado erróneo.

Por ejemplo, $100_{(10} + 100_{(10} =$ **01100100 + 01100100 = 11001000** $= -56_{(10}$ resultado obviamente erróneo.

El *desbordamiento* (*over-flow*) se produce:

a) cuando se suman dos números positivos y aparece un resultado negativo (lo cual es debido a que el resultado es mayor que $\mathbf{2^{n-1}-1}$ y, al no caber en los **n-1** bits del campo numérico, interfiere con el bit de signo y lo modifica erróneamente)

b) cuando se suman dos números negativos y se genera un número positivo (debido a que el resultado es menor que $\mathbf{-2^{n-1}}$, que no cabe en el campo numérico)

c) cuando se restan dos números de distinto signo y en tal resta se produce una de las dos situaciones anteriores a) o b).

En cambio, no puede producirse desbordamiento en caso de sumar números de distinto signo o de restar números del mismo signo, ya que en ambas situaciones el resultado es inferior al mayor de los operandos y tiene perfecta cabida en el campo numérico sin interferir con el signo.

Situaciones de desbordamiento:

0---------	**+**	**0---------**	**=**	**1---------**
1---------	**+**	**1---------**	**=**	**0---------**
0---------	**-**	**1---------**	**=**	**1---------**
1---------	**-**	**0---------**	**=**	**0---------**

Así, pues, las operaciones de suma y resta de números enteros en complemento a 2 pueden realizarse con los bloques sumadores y restadores descritos en el primer apartado de este capítulo (ya que dichas operaciones son el resultado de contajes/descontajes sucesivos y esta codificación conserva el contaje binario); en todo caso habrá que comprobar que no se produce desbordamiento (*over-flow*).

La comparación entre números en complemento a 2 no presenta dificultades cuando ambos números son del mismo signo; tanto los positivos como los negativos se encuentran en orden correcto conforme a la comparación aritmética que efectúan los bloques comparadores.

Pero la comparación directa entre números de diferente signo llevaría a declarar el número negativo como mayor (pues comienza por **1** y, en cambio, el positivo lo hace por **0**); este error puede evitarse invirtiendo el bit de signo, es decir, añadiendo al comparador sendos inversores previos que actúen sobre el bit más significativo de cada número.

3.3.2 Codificación binaria de números racionales

Los números racionales presentan una parte entera seguida de una parte decimal; dado que la denominación «parte decimal» da lugar a un doble sentido según que el término «decimal» se refiera a dicha parte o al sistema de numeración decimal, se utilizará en lo sucesivo la denominación «parte no entera».

La forma más directa de codificar los números racionales consiste en reservar un determinado número de dígitos para la parte no entera: se desprecian aquellas cifras decimales que sobrepasen (inferiores en valor relativo) los dígitos disponibles y, en cambio, se completan con ceros cuando el número de cifras de la parte no entera sea menor que el de dígitos fijados.

Esta forma de representar números con parte no entera se denomina codificación en coma fija: el número de bits reservados para la parte no entera del número es fijo, de forma que la longitud de la parte no entera ha de ajustarse a dicho número (despreciando cifras decimales de menor valor significativo o añadiendo ceros, según proceda).

Supongamos una codificación en coma fija de 6 bits: se prescindirá de los dígitos de la parte no entera que ocupan lugares posteriores al sexto y se complementará con ceros caso de que el número de dígitos decimales sea inferior a 6.

Ejemplos:	***0,1***	***= 0,100000***	***0,101***	***= 0,101000***
	0,01011001	***= 0,010110***	***0,00000001***	***= 0,000000***
	10	***= 10,000000***	***101,011***	***= 101,011000***

Las operaciones de suma y resta construidas para los números naturales pueden aplicarse directamente a los números racionales en coma fija y el resultado será un número racional expresado en coma fija con el mismo número de dígitos. El establecimiento de una longitud fija para la parte no entera permite tratar a estos números como enteros: al interpretar el resultado se añadirá la coma en la posición que corresponde al número de bits de la parte no entera.

La conversión de la parte no entera de un número desde el sistema decimal al binario se realiza multiplicando sucesivamente por 2 dicha parte no entera:

	719
1	438
0	876
1	752
1	504
1	008
0	016
...	...

0,719 **= 0,101110**

La conversión recíproca del sistema binario al decimal se realiza multiplicando cada bit por su valor significativo $\mathbf{2^{-i} = 1/2^i}$, siendo **i** el número de orden que ocupa en la parte no entera:

0,101110... = 1 x 0,5 + **0** x 0,25 + **1** x 0,125 + **1** x 0,0625 + **1** x 0,03125 + **...**
= 0,5 + 0,125 + 0,0625 + 0,03125 **=** 0,71875 ≈ 0,719

La codificación en complemento a 2 es válida para manejar los números negativos en coma fija: en este caso, el cambio de signo de un número se realiza invirtiendo todos sus bits y sumando un bit **1** al número resultante; dicha suma ha de efectuarse sobre el bit menos significativo de la coma fija.

Ejemplo:	*2,719*	=	***00000010,101110***	*invertir los bits*
	2,719'	=	***11111101,010001***	*+0,000001*
	- 2,719	=	***11111101,010010***	

Con palabras binarias de 32 bits, reservando 16 para la parte entera (de ellas, un bit para el signo) y otras 16 para la parte no entera, pueden representarse números con valor absoluto de hasta $2^{15} = 32.768$ y con una precisión de $2^{-16} = 0{,}000015$.

Con 64 bits, 40 para la parte entera y 24 para la no entera, se alcanzan números cuya magnitud llega a $2^{39} = 2^9 \cdot 2^{30} \approx 500 \cdot 10^9$ (ya que $2^{10} \approx 10^3$), medio billón, y cuya precisión decimal sea de $2^{-24} = 2^{-4} \cdot 2^{-20} \approx 0{,}0625 \cdot 10^{-6}$, una diez millonésima.

3.3.3. Codificación en coma flotante

La forma exponencial permite codificar los números racionales, ampliando el rango o la precisión de los mismos.

Todo número **a** puede expresarse mediante su cifra más significativa como parte entera, seguida del resto del número como parte no entera **a'**, multiplicado por la base del sistema de numeración elevada al correspondiente exponente **e**.

En sistema binario $\mathbf{a = 1{,}a' \cdot 2^e}$ (mejor, $1{,}a' \cdot 10^e$, ya que $2 = 10_{(2)}$ donde **a'** es el resultado de prescindir en **a** del bit más significativo (y de la coma de separación decimal) y $\mathbf{2^e}$ es el valor numérico relativo de dicho bit.

a' recibe el nombre de mantisa, **e** es el exponente y la codificación en coma flotante se configura reservando un bit para el signo, un número fijo de bits para el exponente y otro número, fijo también, de bits para la mantisa:

• el número de bits del exponente determina la magnitud (tanto en grande como en pequeño) de los números que pueden representarse;

• el número de bits de la mantisa obliga a ajustarla a dicho tamaño, despreciando, en su caso, los dígitos menos significativos, o añadiendo, si es preciso, ceros para completar el número establecido de bits; ello equivale a fijar el número de cifras significativas que se utilizan.

a) Coma flotante de simple precisión

La codificación binaria en coma flotante de simple precisión utiliza 32 bits, de los cuales el más significativo expresa el signo del número, los 8 siguientes contienen el exponente y los 23 restantes están reservados para la mantisa.

En principio, el exponente se expresa en código exceso 128, de forma que el exponente **10000000** corresponde al 0 y exponentes que comienzan por **0** son negativos:

00000000 = -128; **01111111 =** -1; **10000000 =** 0; **11111111** = +127

el rango de los números irá de $2^{127} \sim 10^{38}$, sixtillones, en cuanto a números grandes, hasta $2^{-128} \sim 10^{-38}$, una sixtillonésima, para números pequeños. Los 23 bits de la mantisa equivalen a 7 cifras decimales significativas ($2^{23} \sim 10^{7}$).

Ahora bien, esta codificación no permite representar el valor **0** (el más cercano posible será el $1{,}0 \cdot 2^{-128}$); para incluir el **0** (y para mejorar la precisión de los números de valor absoluto muy pequeño) la forma de codificar en coma flotante de simple precisión se ha establecido según las siguientes normas (estándar IEEE 754):

- El primer bit indica el signo (**0 = +**; **1 = –**), los 8 bits siguientes expresan el exponente y los 23 bits restantes contienen la mantisa, normalizada en la forma **1,mantisa**: $\mathbf{(-1)^{signo} \cdot 1,mantisa \cdot 2^{exponente-127}}$
- El exponente se codifica en 8 bits en exceso 127, desde **00000001** = -126 hasta **11111110** = +127, pasando por **01111110** = -1; **01111111** = 0; **10000000** = +1; **10000001** = +2 ;**…**
- Cuando el exponente es **00000000** se utiliza la normalización **0.mantisa**; si la mantisa es nula representa el cero y si no es nula, representa el número: $\mathbf{0,mantisa \cdot 2^{-126}}$
- El exponente **11111111** se reserva para valores infinitos o indeterminados:
  - cuando la mantisa es nula: ∞ infinito, +∞ o –∞ según el signo (primer bit)
  - si la mantisa no es nula: **NaN** no es un número (por ejemplo: 0/0, etc.;...).

b) Coma flotante de doble precisión

La codificación en coma flotante de doble precisión utiliza 64 bits, el más significativo para el signo, 11 bits para el exponente y los últimos 52 bits expresan la mantisa; las normas de codificación son las siguientes (estándar IEEE 754):

- La mantisa normalizada en la forma **1.mantisa**: $\mathbf{(-1)^{signo} \cdot 1,mantisa \cdot 2^{exponente-1023}}$
- El exponente se codifica en 11 bits en exceso 1023, desde **00000000001** = -1022 hasta **11111111110** = +1023
- Para exponente **00000000000** se utiliza la normalización **0.mantisa**: mantisa nula para el cero y si no lo es representa el número: $\mathbf{0,mantisa \cdot 2^{-1022}}$.
- El exponente **11111111111** se reserva para valores infinitos o indeterminados: ∞ para mantisa nula y **NaN** cuando la mantisa es distinta de 0.

Los 52 bits de la mantisa equivalen a 15 cifras decimales significativas ($2^{52} \sim 10^{15}$), mientras que el exponente cubre un rango numérico de $10^{\pm 300}$.

3.4. Codificación de números en BCD

En general, los humanos estamos acostumbrados al sistema de numeración decimal (debido al hecho de que disponemos de 10 dedos) y nos resulta más cómodo utilizar los datos numéricos en base 10. Para ello, la entrada de datos en los sistemas digitales requiere la correspondiente conversión *decimal → binario*, mientras que la salida de resultados requiere la conversión inversa *binario → decimal*. Tales conversiones no son sencillas y no pueden realizarse modularmente «a trozos» sino que es preciso operar sobre el número completo a convertir.

Por ello, en muchos sistemas digitales, no se codifican los números en binario sino que se respeta su representación en sistema decimal y se codifica por separado cada una de sus cifras: *codificación* **BCD** (decimal codificado en binario). De esta forma, el número continúa siendo decimal (base 10), con sus cifras codificadas en binario.

Cada cifra decimal necesita 4 bits (habida cuenta de que 9 = **1001**) y se representan siempre los cuatro bits, incluidos los ceros no significativos:

0	**0000**	1	**0001**	2	**0010**	3	**0011**	4	**0100**
5	**0101**	6	**0110**	7	**0111**	8	**1000**	9	**1001**

Las palabras binarias **1010**, **1011**, **1100**, **1101**, **1110**, **1111** no tienen significado en codificación **BCD**.

Ejemplos de codificación binaria y **BCD***:*

173 = $10101101_{(2}$ = $\mathbf{0001\ 0111\ 0011}_{BCD}$

592 = $1001010000_{(2}$ = $\mathbf{0101\ 1001\ 0010}_{BCD}$

846 = $1101001110_{(2}$ = $\mathbf{1000\ 0100\ 0110}_{BCD}$

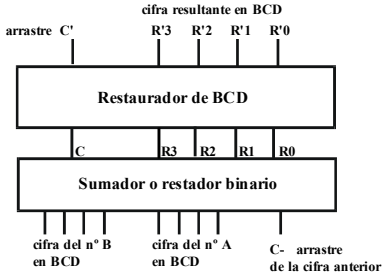
La codificación **BCD** permite introducir directamente los números en un sistema digital a través de 10 teclas, una para cada cifra decimal, y representar directamente los resultados sobre visualizadores de 7 segmentos, a través del correspondiente conversor BCD → 7 segmentos. En ambos casos no es preciso efectuar cambios de código que afecten al número global; será preciso que cada tecla genere el código **BCD** de su cifra y que cada cifra del resultado sea convertida a 7 segmentos para activar el visualizador.

Las operaciones de suma y resta en **BCD** se efectúan de igual forma que en binario, cifra a cifra, añadiendo las siguientes correcciones:

- en el caso de que el resultado **R** de la suma parcial en una cifra sea superior a 9, es necesario sumar 6 unidades adicionales sobre la misma
- en el caso de que la cifra del minuendo sea inferior a la del sustraendo, se restan 6 unidades.

Esta corrección de 6 unidades se debe al hecho de que en BCD se pasa directamente del 9 al 0 (y *me llevo 1*), mientras que en binario hay 6 unidades intermedias entre el **1001** y el **0000** siguiente: las palabras de 4 bits que corresponden a los números del 10 al 15 (**1010**, **1011**, **1100**, **1101**, **1110**, **1111**).

Tales correcciones pueden realizarse directamente mediante un conversor como el de la figura (**c** = arrastre, **R** = cifra resultante):



cuya configuración booleana corresponde a las siguientes funciones:

SUMA	**c**	**R**	**c'**	**R'**	**RESTA**	**c**	**R**	**c'**	**R'**
	0	**0000**	**0**	**0000**		**0**	**0000**	**0**	**0000**
	0	**0001**	**0**	**0001**		**0**	**0001**	**0**	**0001**
	0	**0010**	**0**	**0010**		**0**	**0010**	**0**	**0010**
	0	**…**	**0**	**…**		**0**	**…**	**0**	**…**
	0	**1001**	**0**	**1001**		**0**	**1001**	**0**	**1001**
	0	**1010**	**1**	**0000**		**1**	**1111**	**1**	**1001**
	0	**1011**	**1**	**0001**		**1**	**1110**	**1**	**1000**
	0	**1100**	**1**	**0010**		**1**	**1101**	**1**	**0111**
	0	**1101**	**1**	**0011**		**1**	**1100**	**1**	**0110**
	0	**1110**	**1**	**0100**		**1**	**1011**	**1**	**0101**
	0	**1111**	**1**	**0101**		**1**	**1010**	**1**	**0100**
	1	**0000**	**1**	**0110**		**1**	**1001**	**1**	**0011**
	1	**0001**	**1**	**0111**		**1**	**1000**	**1**	**0010**
	1	**0010**	**1**	**1000**		**1**	**0111**	**1**	**0001**
	1	**0011**	**1**	**1001**		**1**	**0110**	**1**	**0000**

- En la tabla de la suma el «diez» **1010** binario equivale al ***1 0000*** *BCD*, el «once» **1011** al ***1 0001*** *BCD*, el «doce» **1100** al ***1 0010*** *BCD*, etc.;... y el mayor resultado será el que se obtiene al sumar 9 + 9 + arrastre 1 = 19: **10011** binario que equivale a ***1 1001*** *BCD*.
- Al restar 0 – 1 **0000 - 0001** se obtiene **1111** y debería obtenerse 9 ***1001***, si se resta 0 – 2 **0000 - 0010** se obtiene **1110** y debería obtenerse 8 ***1000***, etc.; y el resultado extremo se obtiene al restar 0 – 9 – arrastre 1 = 0 – 10 que da **0110** y debería ser ***0000***.
- Los vectores de entrada no incluidos en estas tablas no se presentan nunca y pueden ser asignados con valor **X** a efectos de simplificar las funciones.

Las unidades aritméticas y lógicas **ALU** incorporan esta conversión en muchos casos, en particular en los microprocesadores de 8 bits, de forma que admiten la doble posibilidad de suma y resta en binario y en **BCD**.

Para representar los números negativos en sistema de numeración binario se utiliza el *complemento a* 2^n (identificando el 0 con 2^n, siendo n el número de dígitos empleados). De la misma forma, en el sistema decimal los números negativos pueden codificarse en complemento a $\mathbf{10^n}$ (complemento a 10), identificando el 0 con 10^n (para n = número de dígitos empleados).

Por ejemplo, para números de valor absoluto inferior a 100.000, añadiendo la cifra de signo en la primera posición resulta un número global de 6 cifras:

+1027 = 001027 - 8395 = 10^6 - 8395 = 1000000 - 8395 = 99605

La cifra inicial 0 es propia de los números positivos, mientras que un 9 inicial indica que el número es negativo.

El cambio de signo puede realizarse sin efectuar la sustracción $\mathbf{10^n-A}$; basta calcular el complemento a 9 de cada cifra y sumar al número resultante una unidad:

de positivo a negativo	*+1027*	=	*001027*	*se complementa a 9 cada cifra*
	1027'	=	*998972*	*+1*
	-1027	=	*998973*	
de negativo a positivo	*-1027*	=	*98973*	*se complementa a 9 cada cifra*
	-1027'	=	*01026*	*+1*
	+1027	=	*01027*	

Esta codificación es compatible con la suma y la resta aritmética de números naturales en **BCD** y el desbordamiento (*over-flow*) se produce cuando:

suma	0---------	+	0---------	=	1---------
	9---------	+	9---------	=	8---------
resta	0---------	-	9---------	=	1---------
	9---------	-	0---------	=	8---------

es decir, cuando el bit de signo deja de ser 0 ó 9.

Para operar en **BCD** con números racionales (con parte no entera) se utiliza la representación en coma fija, reservando un número fijo de dígitos BCD para la parte no entera (en forma análoga a la indicada anteriormente para los números racionales en base 2). En **BCD** no se utiliza la representación en coma flotante, ya que, cuando tal codificación resulta necesaria, por razones de precisión, de rango numérico o de velocidad de cálculo, por las mismas razones resulta también conveniente expresar los números y operar con ellos directamente en sistema binario.