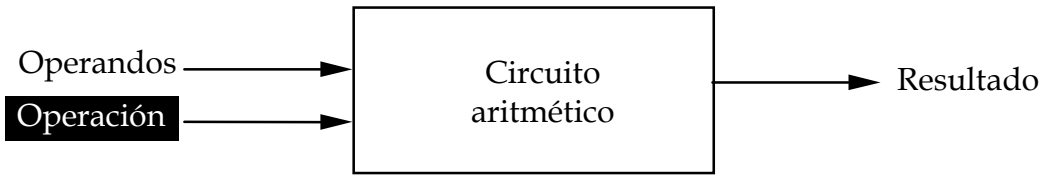
ALGORITMOS Y PROCESADORES ARITMÉTICOS

ÍNDICE

0. INTRODUCCIÓN

El objetivo de este tema es examinar los diferentes algoritmos de cálculo aritmético, centrándonos en los algoritmos de suma, resta, multiplicación y división, así como el *hardware* asociado a dichos algoritmos, comenzando con un repaso de los sistemas de representación, necesario para el tratamiento de los datos numéricos. En general trabajaremos con enteros, aunque en algunos casos utilizaremos los reales.

Visto desde fuera, el esquema lógico de un circuito aritmético será el que aparece en la figura: un sistema que efectúa operaciones sobre datos numéricos de entrada obteniendo un resultado de salida.



Los *operandos* y el *resultado* se definen por su **sistema de representación** y su **rango**. Cuando sea posible realizar más de una operación con los operandos de entrada, un código de operación indicará cuál es la operación que hay que ejecutar. Los resultados pueden ser del mismo tipo que los operandos (en una suma, por ejemplo), pueden ser valores lógicos (una comparación) y pueden indicar también condiciones singulares (dividir entre 0, desbordamiento...). Es posible que además de las señales mencionadas, el procesador aritmético utilice otras señales de control o sincronización para indicar el comienzo y final de la operación.

Los circuitos que vamos a analizar forman el núcleo de cualquier unidad aritmético-lógica. Por ello, resulta crucial analizar tanto el tiempo de cálculo como el coste de los algoritmos y circuitos que vamos a proponer. Es obvio que nos interesa minimizar ambos, aunque, como veremos, será necesario buscar un compromiso entre la velocidad de ejecución y el coste. Para presentar los circuitos utilizaremos, como siempre, bloques estándar, bien secuenciales o bien combinacionales.

En las páginas que siguen vamos a presentar todas estas cuestiones. En el primer capítulo veremos los sistemas de representación de números naturales, enteros y reales de coma fija. En el segundo capítulo analizaremos los algoritmos y el *hardware* para la suma y la resta. Los problemas creados por la llevada de la suma y su solución serán analizados en el tercer capítulo. En el cuarto capítulo veremos el producto de números enteros y en el quinto la división de números naturales. Finalmente, dedicaremos el capítulo sexto a realizar un breve resumen de la aritmética de coma flotante.

0.1 Bibliografía

Estas notas han sido tomadas de diferentes libros; entre ellos, los más utilizados han sido los siguientes:

1. M. D. Ercegovac, T. Lang
   Digital Arithmetic.
   Ed. Morgan Kaufmann, 2004

2. D. A. Patterson & J. L. Hennessy
   Estructura y diseño de computadores (capítulo 4, volumen 1)
   Ed. Reverté, 2000

3. J. L. Hennessy & D. A. Patterson
   Computer Architecture, A Quantitative Approach (Apéndice H)
   3.edicion, Ed. Morgan Kaufmann, 2002

4. K. Hwang
   Computer Arithmetic.
   Ed. J. Wiley and Sons, 1979

5. N. R. Scott
   Computer Number Systems & Arithmetic.
   Ed. Prentice Hall, 1985

6. J. F. Cavanagh
   Digital Computer Arithmetic.
   Ed. McGraw-Hill, 1985

7. IEEE Standars Board
   IEEE Standard for Binary Floating-Point Arithmetic, 1985

En caso de estar interesado, el lector no tendría problemas para encontrar otros textos equivalentes en la extensa bibliografía que hay sobre este tema. También podrá encontrar multitud de artículos sobre algoritmos y circuitos aritméticos en las revistas técnicas.

1. SISTEMAS DE REPRESENTACIÓN

Un número se representa mediante una n_tupla ordenada. Cada elemento de la n_tupla es un **dígito** y la n_tupla se denomina **vector de dígitos**.

número:x vector de dígitos: $X = (X_{n-1}X_{n-2}.....X_1X_0)$

La cantidad de números que se pueden representar con n dígitos es limitada. Estos números constituyen el **rango** de una determinada representación. Para describir el sistema de represe-tación, se requiere:

- Una regla de interpretación que relacione el número con su vector de dígitos.
- Un conjunto de valores para los dígitos, D_i. Por ejemplo, {0,1,2,....,9} es el conjunto de dígitos D_i para el sistema de representación en base 10.

Un sistema de representación es **no redundante** si cada vector de dígitos representa un único número y cada número es representado por un único vector de dígitos: es decir, se trata de una correspondencia biyectiva. Es **redundante** en caso contrario.

Los sistemas de representación más utilizados son los **sistemas posicionales** o **sistemas de pesos,** donde cada dígito del vector de dígitos tiene un valor dependiendo de la posición en la que aparece. En estos sistemas, la función de representación es:

$$x = \sum_{i=0}^{n-1} X_i W_i$$

siendo $W = (W_{n-1}W_{n-2}.....W_1W_0)$ el **vector de pesos.**

En sistemas de bases (*radix number-system*) el vector de pesos se relaciona con el **vector de bases** $R = (R_{n-1}R_{n-2}.....R_1R_0)$ de la siguiente forma:

$$W_0 = 1, \qquad W_i = W_{i-1} \cdot R_{i-1} = \prod_{j=0}^{i-1} R_j$$

Los sistemas de bases se clasifican, de acuerdo al vector de bases, en sistemas de base fija y de base variable. En sistemas de base fija todos los elementos del vector de bases tienen el mismo valor, r. El vector de pesos resulta ser $W = (r^{n-1}r^{n-2}.....r^2r1)$ y el valor de x se calcula como

$$x = \sum_{i=0}^{n-1} X_i r^i$$

En sistemas de base variable los elementos del vector de bases son diferentes. Por ejemplo, en el sistema horario R = (24, 60, 60) y W = (3600, 60, 1).

De acuerdo con el conjunto de valores que se utilizan para definir los dígitos, los sistemas se clasifican en **canónicos** y **no canónicos**. En un sistema canónico $D_i = \{0, 1, 2, ..., r-1\}$. Un sistema con una base fija positiva r y un conjunto canónico para los dígitos se denomina sistema de numeración convencional en base r.

Existen también sistemas de numeración que no utilizan un vector de pesos, son los sistemas no ponderados. Un ejemplo es el **sistema de residuos**. Dado un vector de números primos entre sí $P = (P_{n-1}P_{n-2}.....P_1P_0)$, un entero x se representa por un vector de dígitos X tal que $X_i = x \bmod P_i$.

En adelante sólo tomaremos en cuenta los sistemas convencionales.

1.1. Representación de Naturales

Utilizando n dígitos, sólo se puede representar un subconjunto de $\mathbb{N}$. Un número x queda representado mediante el vector de dígitos $X = (X_{n-1}X_{n-2}.....X_1X_0)$, siendo

$$x = \sum_{i=0}^{n-1} X_i r^i$$

El algoritmo para obtener el vector de dígitos correspondiente a un número x es el siguiente:

$X_0 = x \bmod r$; $\quad Q_1 = x \operatorname{div} r$

$X_1 = Q_1 \bmod r$; $\quad Q_2 = Q_1 \operatorname{div} r$ $\quad$ etc...

El rango de esta representación es $\left[0, 1, 2, ..., \left(r^{n-1}\right)\right]$.

1.2. Representación de Enteros

Al igual que en el caso anterior, mediante n dígitos, $X = (X_{n-1}X_{n-2}.....X_1X_0)$, sólo es posible representar un subconjunto de $\mathbb{Z}$. Existen diferentes alternativas para la representación de los números negativos, de las que sólo vamos a analizar las más utilizadas.

1.2.1. Representación en Signo-Magnitud

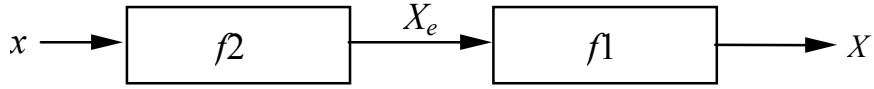
Un número entero x se representa mediante un dígito de signo y la magnitud. El signo viene dado por el dígito de más peso, X_{n-1}. Si éste es 0, el número es positivo y si es $(r-1)$, es negativo. La magnitud se calcula mediante el valor explícito del resto de los dígitos de la representación:

$$|x| = \sum_{i=0}^{n-2} X_i r^i$$

El rango de este sistema de representación es $\left[-\left(r^{n-1}-1\right), ..., -1, -0, +0, +1, ..., +\left(r^{n-1}-1\right)\right]$. Como puede observarse, la representación del 0 es doble (+0, -0), dado que se trata el signo como algo aparte.

1.2.2. Representación en formas complementadas

En este sistema no se hace distinción entre el signo y la magnitud. Un entero x (*valor implícito*) es representado por un natural X_e (*valor explícito*) que, finalmente, es representado por su vector de dígitos X.



(valor implícito) (valor explícito) (vector de dígitos)

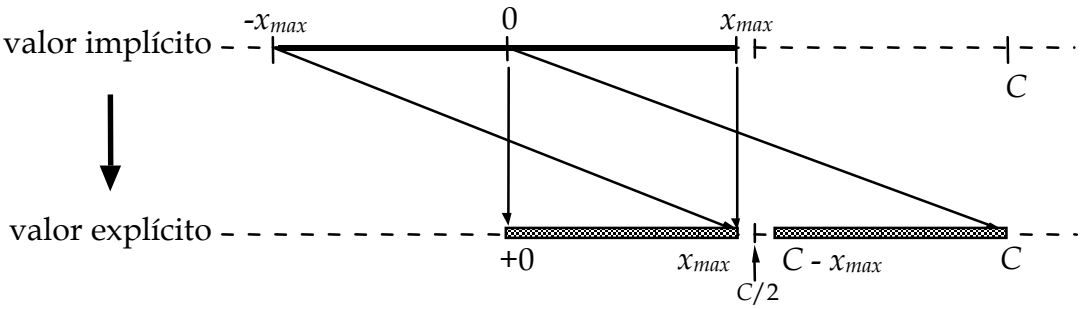
La determinación del valor explícito de un número, X_e, se realiza mediante la función *f2*, de la siguiente manera, siendo C la constante de complementación:

$X_e = x$ $\quad$ si $x \geq 0$

$X_e = C - |x|$ $\quad$ si $x < 0$

En esta representación, los números positivos se representan directamente y los negativos se representan de forma complementada $(C - |x|)$. Siendo esto así, los números x = C/2 y x = -C/2 estarían representados por el mismo valor explícito: C/2. Para evitar esta ambigüedad debe cumplirse que $|x_{\max}| < C/2$. En ese caso, *f2* es la función módulo: $f2 = (x \bmod C)$.

En la siguiente gráfica se presenta la relación entre los valores implícitos y explícitos. Como puede observarse, el valor explícito $C/2$ no se utiliza para representar ningún número.



Dado un valor explícito X_e, el valor implícito $x = f(X_e)$ se define como:

$$x = X_e \qquad \text{si } X_e < C/2$$
$$x = X_e - C \qquad \text{si } X_e \geq C/2$$

A la hora de escoger la constante de complementación C, dos son las alternativa principales:

$C = r^n$ *complemento a la base*

$C = r^n - 1$ *complemento restringido a la base*

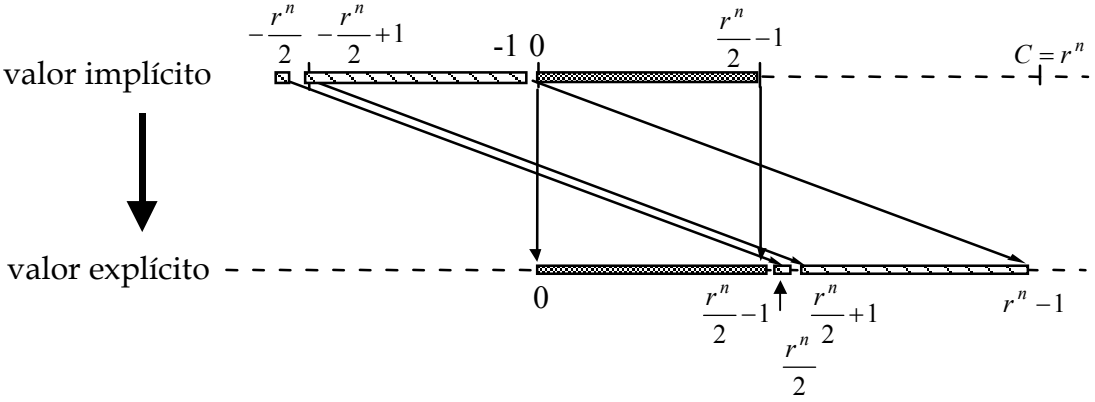
Analicemos estos dos casos.

1.2.2.1 $C = r^n$, complemento a la base

El conjunto de valores explícitos representable con n dígitos, $\lfloor 0, r^n - 1 \rfloor$, se divide en dos intervalos iguales:

$\left[0, \frac{r^n}{2} - 1\right]$ para representar a los positivos y al 0: $\left[0, \frac{r^n}{2} - 1\right]$

$\left[\frac{r^n}{2}, r^n - 1\right]$ para representar a los negativos: $\left[-\frac{r^n}{2}, -1\right]$



Como puede verse, el 0 tiene una única representación (000 ... 0). Además, en contra de lo indicado anteriormente para el caso general ($x<C/2$), se utiliza el valor explícito $C/2$ para representar un número negativo más, $-C/2$. Debido a ello, el rango de la representación es asimétrico: se representan más números negativos que positivos. Dado el vector de dígitos X, el valor implícito del número x se obtiene:

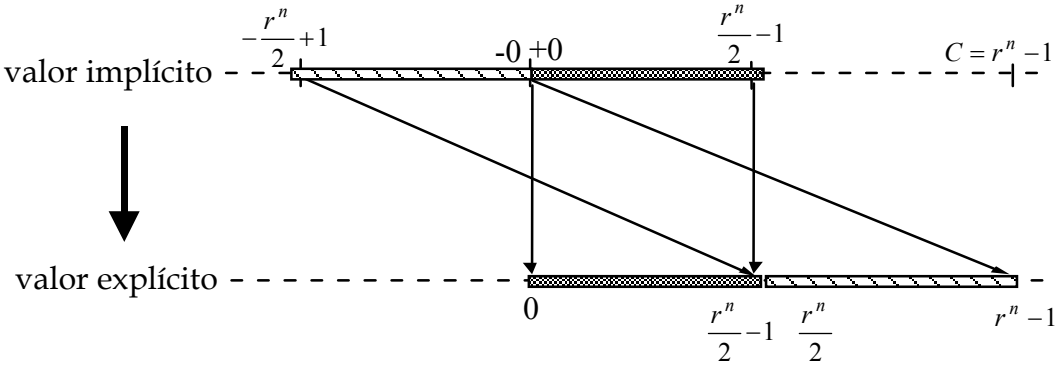
$$x = \sum_{i=0}^{n-1} X_i r^i \quad \text{si } X_{n-1} < \frac{r}{2} \qquad \text{o bien} \qquad x = \sum_{i=0}^{n-1} X_i r^i - r^n \quad \text{si } X_{n-1} \geq \frac{r}{2}$$

1.2.2.2 $C = r^n - 1$, complemento restringido a la base

El conjunto de valores explícitos representable con *n* dígitos se utiliza en este caso de la siguiente manera:

$\left[0, \frac{r^n}{2} - 1\right]$ para representar a los positivos y al 0: $\left[0, \frac{r^n}{2} - 1\right]$

$\left[\frac{r^n}{2}, r^n - 1\right]$ para representar a los negativos y al 0: $\left[-\left(\frac{r^n}{2} - 1\right), 0\right]$



En esta representación el cero tiene dos representaciones posibles:

$+0 \quad \rightarrow \quad X_e = 0$

$-0 \quad \rightarrow \quad X_e = r^n - 1$

En cambio, el rango de esta representación es simétrico: el número de positivos y negativos representados, el 0 aparte, es el mismo.

Dado un vector de dígitos *X*, el valor implícito del número, *x*, se obtiene de manera similar a lo visto anteriormente:

$$x = \sum_{i=0}^{n-1} X_i r^i \quad \text{si } X_{n-1} < \frac{r}{2} \qquad \text{o bien} \qquad x = \sum_{i=0}^{n-1} X_i r^i - \left(r^n - 1\right) \quad \text{si } X_{n-1} \geq \frac{r}{2}$$

1.2.3 Representación por exceso (*biased*)

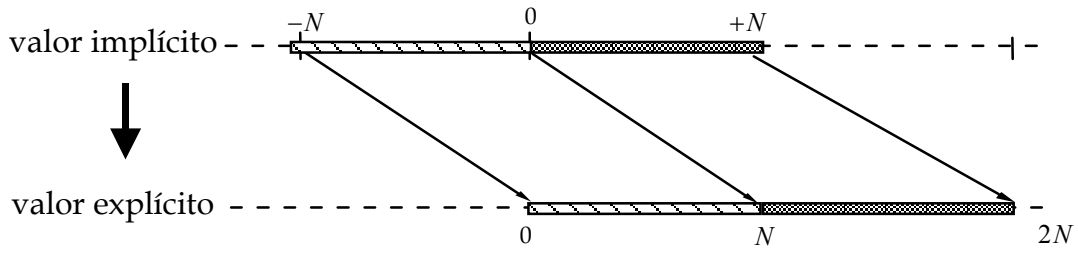
Por último, existe otra forma de representar números positivos y negativos. Al igual que en las representaciones anteriores, es necesario aplicar una transformación para obtener el valor explícito, y en este caso la función es la siguiente:

$$X_e = x + N$$

Todos los números se trasladan por igual y el valor de la constante *N* ha de ser tal que convierta en 0 el valor más negativo. Habitualmente se utilizan los valores $r^n/2$ o $(r^n/2)$-1. En comparación con las representaciones anteriores, ahora se "desplazan" todos los números, ya que a todos se les suma *N*, y no sólo a los negativos:

$$3 \rightarrow 3 + N \qquad\qquad -3 \rightarrow -3 + N$$

A esta representación se le denomina *exceso N*. La siguiente figura muestra la relación entre los valores implícitos y los explícitos.



Dado el valor explícito, la función a aplicar para determinar el valor implícito será la siguiente:

$$x = X_e - N$$

Resumiendo todas las representaciones indicadas para $r=2$, $n=4$:

valor explícito	valor implícito				
Xe	Comp. a la Base **C2**	Comp. Rest. Base **C1**	Signo-Magnitud **S/M**	Exceso $\left(2^{n-1}-1\right)$ **Exceso 7**	Exceso 2^{n-1} **Exceso 8**
0	0	+0	+0	−7	−8
1	1	1	1	−6	−7
2	2	2	2	−5	−6
3	3	3	3	−4	−5
4	4	4	4	−3	−4
5	5	5	5	−2	−3
6	6	6	6	−1	−2
7	7	7	7	0	−1
8	−8	−7	−0	1	0
9	−7	−6	−1	2	1
10	−6	−5	−2	3	2
11	−5	−4	−3	4	3
12	−4	−3	−4	5	4
13	−3	−2	−5	6	5
14	−2	−1	−6	7	6
15	−1	−0	−7	8	7
	[−8,+7]	[−7,+7]	[−7,+7]	[−7,+8]	[−8,+7]

1.2.4. Detección del signo

Sea *signo*(*x*) la función que representa al signo del número *x*: *signo*(*x*) = 0, si el número es positivo o cero; y *signo*(*x*) = 1, si el número es negativo. En Signo Magnitud detectar el signo del número a partir de su vector de dígitos es trivial, basta con inspeccionar el dígito de signo X_{n-1}. En formas complementadas, considerando cómo se representan los números, el signo del número resulta ser:

$$\text{Si } X_e < C/2 \quad \rightarrow \quad signo(x) = 0$$
$$\text{Si } X_e \geq C/2 \quad \rightarrow \quad signo(x) = 1$$

Así pues, y para las representaciones que estamos analizando, el signo se determina inspeccionando el dígito más significativo del vector de dígitos *X*, de la siguiente forma:

$$X_{n-1} < r/2 \quad \rightarrow \quad X_e < C/2 \quad \rightarrow \quad x \geq 0 \text{ (positivo)}$$
$$X_{n-1} \geq r/2 \quad \rightarrow \quad X_e \geq C/2 \quad \rightarrow \quad x < 0 \text{ (negativo)}$$

En particular, para *r*=2, el signo corresponde al bit más significativo; es decir, $signo(x) = X_{n-1}$. En caso de bases impares debe ser inspeccionada la totalidad del vector *X* para determinar el signo, siendo ésta una de las razones por las que una base impar es poco deseable.

Análogamente ocurre en la representación por exceso:

$$\text{Si } X_e \geq N \quad \rightarrow \quad signo(x) = 0$$
$$\text{Si } X_e < \text{N} \quad \rightarrow \quad signo(x) = 1$$

y para conocer el signo del número es suficiente con examinar el dígito de más peso, X_{n-1}

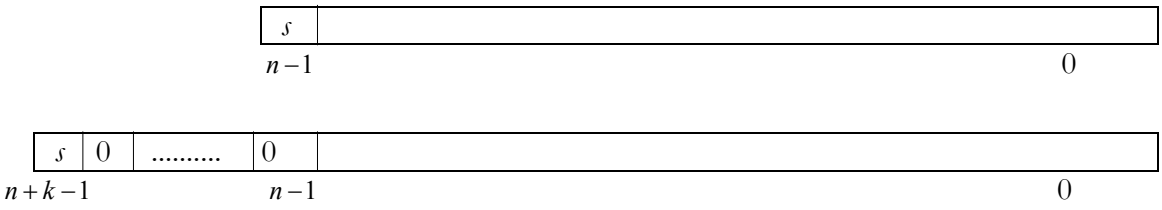
$$X_{n-1} \geq r/2 \quad \rightarrow \quad X_e \geq N \quad \rightarrow \quad signo(x) = 0 \text{ (positivo)}$$
$$X_{n-1} < r/2 \quad \rightarrow \quad X_e < N \quad \rightarrow \quad signo(x) = 1 \text{ (negativo)}$$

Ahora bien, en el caso particular de que la base sea $r = 2$, en esta representación el signo es lo contrario del bit de más peso: si este bit es 0 el número es negativo ($signo(x)=1$), y si dicho bit es 1 es positivo ($signo(x)=0$); es decir, $signo(x) = \overline{X_{n-1}}$.

1.2.5. Ampliación de los márgenes de representación

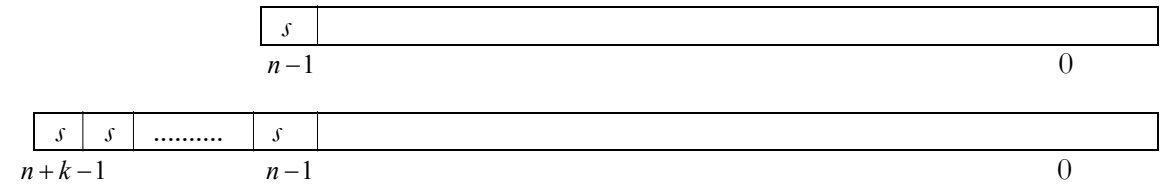
Dado un número representado con n dígitos, ¿cómo se representa dicho número si utilizamos $n+k$ dígitos? Veamos los tres casos que estamos analizando (tomaremos $r = 2$).

a. En Signo Magnitud



es decir, se copia el bit de signo, bit $n-1$, de la representación inicial en el bit $n+k-1$ y se ponen a 0 los bits comprendidos entre $[n+k-2, n-1]$.

b. En Complemento a la base y Complemento restringido



es decir, se repite el bit de signo inicial en todos los bits añadidos a la representación. Esto se conoce como extensión del signo.

(NOTA: si la base no es 2, la extensión se realiza utilizando el dígito $D_i = (r-1) \cdot signo(x)$. Por ejemplo, si $r = 10$, el dígito que se extiende es 0 si el número es positivo y 9 si es negativo)

1.3. Representación de reales en coma fija

Para representar números reales puede utilizarse una representación de coma fija, separando la parte entera y la parte fraccionaria del número.

$$X = (X_{n-1}X_{n-2}.....X_1X_0, X_{-1}X_{-2}.....X_{-k})$$

parte entera (*n* dígitos)parte fraccionaria (*k* dígitos)

El peso de los dígitos de la parte fraccionaria es: $(r^{-1}, r^{-2}, ..., r^{-k})$.

Si bien la cantidad de números enteros o naturales dentro de un intervalo es finita, con los números reales ocurre que en un intervalo dicha cantidad es infinita. Por tanto, no va a ser posible la representación de cualquier número real de dicho intervalo, sino sólo de un subconjunto de ellos. Por eso, la representación de números reales, habitualmente, no es exacta sino que se comete un error de representación (en el último capítulo de estas notas puedes ver el error cometido al elegir una representación). A esto hay que añadir que un número real que puede ser representado de forma exacta en una base, puede requerir un número infinito de dígitos en otra base (por ejemplo 1/3 no se puede representar en base 10 con un número finito de dígitos, pero es muy fácil de representar en base 3).

1.3.1. Cambio de base

El procedimiento para codificar un número real en una base diferente es el siguiente:

- la parte entera, como si se tratara de un número entero
- la parte fraccionaria: se hacen sucesivas multiplicaciones por la base *r* y se guarda lo que va quedando a la izquierda de la coma

Ejemplo: $12{,}28_{10} \rightarrow 1\,1\,0\,0,\,0\,1\,0\,0_2$

$12_{10} = 1\,1\,0\,0_2$

0,28 * 2 = 0,56 → 0
0,56 * 2 = 1,12 → 1
0,12 * 2 = 0,24 → 0
0,24 * 2 = 0,48 → 0 etc..

1.3.2. Sistemas de representación de reales

Al igual que en la representación de enteros, vamos a ver varios sistemas de representación.

a. **Signo Magnitud**: Un dígito para el signo y el resto para la magnitud.

$+121{,}53_{10} \rightarrow 0121{,}53_{10}$
$-121{,}53_{10} \rightarrow 9121{,}53_{10}$

b. **Complemento restringido a la base**. $C = r^n - r^{-k}$. Por tanto,

si $x \geq 0$ → $X_e = x$
si $x < 0$ → $X_e = (r^n - r^{-k}) - |x|$

Para obtener el valor del número a partir de su vector de dígitos

$$\text{Si } X_{n-1} < \frac{r}{2} \quad \rightarrow \quad x = \sum_{i=-k}^{n-1} X_i r^i$$

$$\text{Si } X_{n-1} \geq \frac{r}{2} \quad \rightarrow \quad x = \sum_{i=-k}^{n-1} X_i r^i - \left(r^n - r^{-k}\right)$$

Por ejemplo, $n = 3$, $k = 2$, $r = 10$

$+275{,}21_{10} \quad \rightarrow \quad 275{,}21_{10}$
$-121{,}37_{10} \quad \rightarrow \quad 878{,}62_{10}$

c. **<u>Complemento a la base</u>**. $C = r^n$. Por tanto,

$$\text{si } x \geq 0 \quad \rightarrow \quad X_e = x$$
$$\text{si } x < 0 \quad \rightarrow \quad X_e = r^n - |x|$$

Para obtener el valor del número a partir de su vector de dígitos

$$\text{Si } X_{n-1} < \frac{r}{2} \quad \rightarrow \quad x = \sum_{i=-k}^{n-1} X_i r^i$$

$$\text{Si } X_{n-1} \geq \frac{r}{2} \quad \rightarrow \quad x = \sum_{i=-k}^{n-1} X_i r^i - r^n$$

Por ejemplo, $n = 3$, $k = 2$, $r = 10$

$+275{,}21_{10} \quad \rightarrow \quad 275{,}21_{10}$
$-121{,}37_{10} \quad \rightarrow \quad 878{,}63_{10}$

2. ALGORITMOS DE SUMA Y RESTA PARA NÚMEROS ENTEROS

Sean x e y dos números enteros representados por los vectores X e Y. El algoritmo de suma ADD proporciona un vector de dígitos S que representa al entero $s = x + y$.

$$S = \text{ADD}\ (X,Y)$$

Para la resta, $d = x - y = x + (-y)$, es suficiente considerar el algoritmo de suma ADD y el de cambio de signo CS.

$$D = \text{ADD}\ (X,\text{CS}(Y))$$

Si el rango de S y D es el mismo que el de X e Y, la operación puede dar lugar a resultados no representables (desbordamiento). La complejidad de implementar en hardware ADD y CS depende del sistema de representación elegido. Vamos a analizar dichas operaciones para los sistemas de representación en signo magnitud y las formas complementadas, dejando el análisis de la representación en exceso, puesto que es menos utilizada.

2.1. Suma en Signo Magnitud

Sean x, y, y s enteros representados por (x_s, x_m), (y_s, y_m), (s_s, s_m) en Signo Magnitud. Las magnitudes se representarán mediante vectores de dígitos X, Y, S. Asumiendo que el número de dígitos para la magnitud es n, tenemos que

$$0 \leq x_m, y_m, s_m \leq r^n - 1$$

Para efectuar la suma, es necesario analizar los signos de los operandos, ya que en algunos casos habrá que hacer la suma de las magnitudes y en otros la resta. Además, la suma de magnitudes se efectuará módulo r^n (n dígitos), y por tanto, se producirá desbordamiento cuando $|S| \geq r^n$. El algoritmo para sumar en Signo Magnitud es el siguiente:

```
if (x_s = y_s) then
    begin
        s_m = (x_m + y_m) mod r^n;
        s_s = y_s;
        if (x_m + y_m) ≥ r^n then OVF = 1;
    end
else if (x_m ≥ y_m) then                (* x_s ≠ y_s *)
    begin
        s_m = x_m - y_m;
        s_s = x_s;
    end
else if (x_m < y_m) then                (* x_s ≠ y_s; x_m < y_m *)
    begin
        s_m = y_m - x_m;
        s_s = y_s;
    end
```

Este algoritmo es relativamente complejo pues requiere de una comparación de signos, una comparación de magnitudes y una suma o resta. Se deja como ejercicio el diseño del circuito que realiza este algoritmo.

2.2. Suma en sistemas complementados

Si no hay desbordamiento, esto es, si $|x + y| < C/2$, para obtener $s = x + y$ basta calcular $S_e = (X_e + Y_e) \bmod C$. El algoritmo se limita a efectuar una suma y una operación módulo. El funcionamiento es independiente de las magnitudes relativas y del signo.

Ejemplo: $C = 64$ $\quad -32 \leq x \leq 31$.

x	y	Xe	Ye	Se	s
13	9	13	9	22	22
13	-9	13	55	68 mod 64=4	4
-13	9	51	9	60	-4
-13	-9	51	55	106 mod 64=42	-22

La suma se realizará con un sumador. Ahora vamos a considerar la operación módulo. Sea Ze = Xe + Ye. Como Xe, Ye < C, entonces Ze < 2C y la operación módulo Se = Ze mod C resulta ser:

Ze	si Ze < C
Ze - C	si C ≤ Ze < 2C

Por tanto, esta operación consiste en determinar cuándo Ze ≥ C para restarle C. La complejidad de esta operación depende de la C utilizada.

2.2.1. Complemento a la base

La representación de Ze < 2C en base r ocupa n+1 dígitos, Z =(Z_n Z_{n-1} ... Z_0), donde el dígito más significativo puede ser 1 ó 0. Por tanto,

$Ze < r^n$ si Zn=0
$Ze >= r^n$ si Zn=1

En el primer caso Se = Ze mod C = Ze y su representación es Z (n dígitos). En el segundo caso es preciso restar r^n de Ze.

$$Ze \bmod r^n = (1,Z_{n-1}, \dots , Z_0) - (1,0, \dots , 0) = (0, Z_{n-1}, \dots, Z_0)$$

Por tanto, la operación *mod* r^n equivale a despreciar el dígito más significativo. Si se utiliza r=2, el bit que se desprecia corresponde a la llevada de salida (Cout) del sumador que produzca Ze a partir de Xe e Ye.

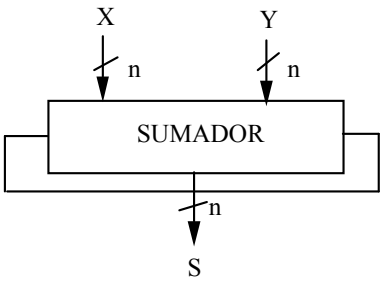
2.2.2. Complemento restringido a la base

En este sistema Se = Ze mod (r^n -1) y hay que considerar los siguientes casos:

1. Si $Ze < r^n - 1$ => Zn = 0 y Ze mod (r^n -1) = Ze
2. Si $Ze = r^n - 1$ => Zn = 0 y Ze mod (r^n -1) = 0
3. Si $Ze > r^n - 1$ => Zn = 1 y Ze mod (r^n -1) = Ze -(r^n -1) = Ze -r^n + 1

Por tanto, si Zn = 0 el resultado es igual a Z y si Zn = 1 el resultado se obtiene despreciando Zn (restando r^n) y sumando un 1.

Cuando r=2, la operación anterior se puede realizar de forma muy sencilla. Teniendo en cuenta que el bit Zn se produce como Cout del sumador, la adición del 1 se consigue mediante la realimentación de la llevada.



EJEMPLOS

Ejemplo de suma en complemento a la base

- n = 4 C= 2^4

x = -5	X_e = 11	X = 1011
y = 5	Y_e = 5	Y = 0101
	Z_e = 16	Z = 10000
s = 0	$\mathbf{S}_e$ = 0	**S** = 0000

- n = 8 C = 2^8

x = -38	X_e = 218	X = 11011010
y = -15	Y_e = 241	Y = 11110001
	Z_e = 459	Z = 111001011
s = -53	**Se** = 203	**S** = 11001011

Ejemplo de suma en complemento restringido a la base

1.- n = 4 C= 2^4 -1

x = 7	X_e = 7	X = 0111
y = -4	Y_e = 11	Y = 1011
	Z_e = 18	Z = 10010
		1
s = 3	$\mathbf{S}_e$ = 3	**S** = 0011

2.3. Operación de cambio de signo

Como ya se ha comentado, para realizar la resta es necesario realizar un cambio de signo y una suma. Por ello, se debe analizar cómo se realiza el cambio de signo en los tres sistemas de representación con los que estamos trabajando.

2.3.1. Cambio de signo en Signo Magnitud

En este sistema de representación el cambio de signo es trivial, se limita a la complementación del dígito de signo.

2.3.2. Cambio de signo en formas complementadas

La operación cambio de signo produce un nuevo número z, tal que z = -x. Por tanto,

$$Ze = (-x) \bmod C = C - x \bmod C = C - X_e$$

El cambio de signo consiste en restar el valor explícito Xe a la constante de complementación C. La complejidad de la operación depende de la C escogida.

a. Complemento restringido a la base

   La constante de complementación se representa por el vector (r -1 r-1 r-1 .... r-1). Por ello, la resta C-X_e se realiza complementando dígito a dígito, respecto a r-1, el vector de dígitos X, para obtener el vector X´.

b. Complemento a la base

   En este caso la resta C - X_e requiere un proceso de resta completo, lo que es complejo. Si consideramos $r^n = (r^n - 1) + 1$. el cambio de signo se realiza en dos pasos: complementando cada dígito respecto a (r-1) y sumando 1. La suma del 1 se realiza poniendo la llevada de entrada a 1, C_{in} = 1, en el sumador.

2.4. Resta en formas complementadas

Tal y como se ha indicado, para realizar la resta hay que combinar una suma y un cambio de signo, ya que d = x - y = x + (-y). El algoritmo correspondiente es:

SUB_{cb}: D <- ADD_e(X,Y´,1)
SUB_{crb}: D <- ADD_e (X,Y´,C_{out})

Resumiendo los algoritmos vistos en formas complementadas, tenemos:

Operación	CB	CRB
s = x + y	S <- ADDe (X, Y, 0)	S <- ADDe (X, Y, C_{out})
z = -x	Z <- ADDe (X´, 0, 1)	Z <- X´
d = x - y	D <- ADDe (X, Y´, 1)	D <- ADDe(X, Y´,C_{out})

La suma en formas complementadas es mucho más simple que en Signo Magnitud, porque no se requieren comparaciones de signo ni de magnitud. Por ello, la mayoría de los sumadores emplean formas complementadas.

El cambio de signo es más complejo en complemento a la base que en complemento restringido, lo contrario que la suma. Debido a la doble representación del cero y a que la aritmética en precisión múltiple es más lenta se utiliza principalmente el sistema de complemento y no el de complemento restringido.

2.5. Detección de desbordamiento (overflow)

Se presenta *desbordamiento* cuando el resultado de una suma/resta supera al mayor/menor entero representable. Por tanto, en este caso el resultado es incorrecto y, por ello, es necesario detectar esta situación.

Cuando se suman números naturales de n bits el desbordamiento es el propio bit de llevada del sumador, esto es, cuando Cn=1 hay desbordamiento. En cambio, en el caso de la resta, la situación es la contraria: si Cn=0, se ha producido desbordamiento; el resultado es negativo y por tanto, no representable.

Analicemos ahora el caso de los números enteros. En Signo Magnitud se trabaja con las magnitudes, es decir, con números naturales. Por tanto habrá desbordamiento si los dos operandos son del mismo signo y existe llevada en la suma.

En formas complementadas existe desbordamiento cuando los operandos son del mismo signo y el resultado de la suma produce un número de signo contrario. En particular, en base 2 se detecta desbordamiento cuando:

$$OVF = \overline{X_{n-1}}\ \overline{Y_{n-1}}\ S_{n-1} + X_{n-1}\ Y_{n-1}\ \overline{S_{n-1}}$$

En complemento a la base también puede detectarse OVF mediante la siguiente función:

$$OVF = C_n \oplus C_{n-1}$$

(Nota: esta fórmula no vale para números representados en complemento restringido a la base. Se deja como ejercicio su demostración).

2.6. Diseño de un sumador Riple Carry Adder –RCA– (sumador serie)

2.6.1. Sumador de un bit

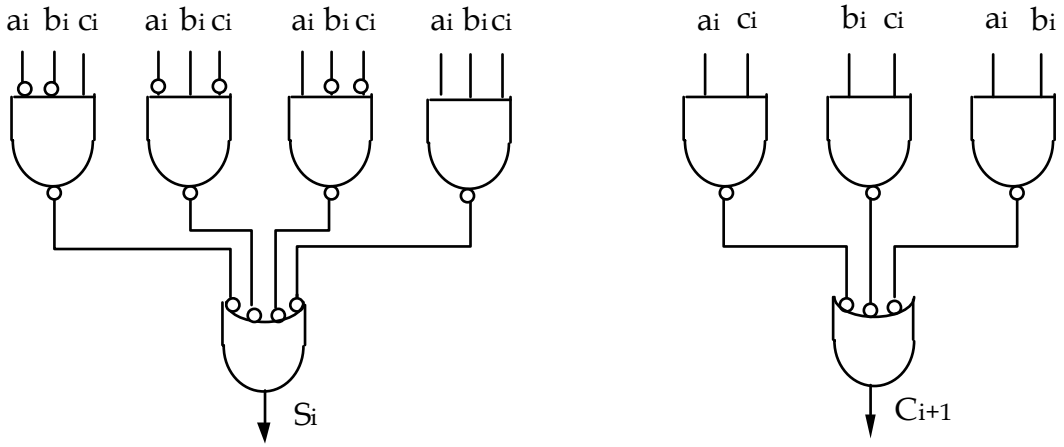
Veamos ahora la construcción de un sumador de 2 números de un bit. Si no tenemos en cuenta la llevada de entrada, el circuito resultante recibe el nombre de semisumador (half-adder HA). En cambio, si sí se considera el circuito resultante recibe el nombre de sumador total (full-adder FA). La siguiente tabla de verdad representa a un sumador total:

a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$s_i = a_i \oplus b_i \oplus c_i$$

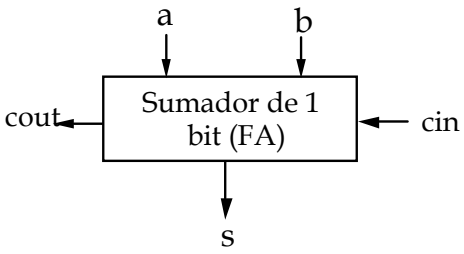
$$c_{i+1} = a_i b_i + (a_i + b_i)\, c_i = a_i b_i + (a_i \oplus b_i)\, c_i$$

Una posible implementación de S_i y C_i+1 es la siguiente:



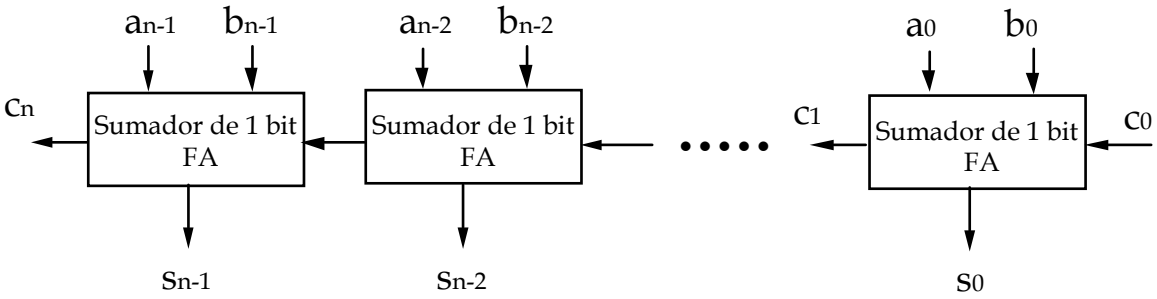
Si consideramos que el retardo de una puerta lógica es Δ, entonces el tiempo necesario para obtener tanto el bit S_i como C_{i+1} resulta ser 2Δ (and / or).

El esquema lógico de un sumador de números de 1 bit es el siguiente:

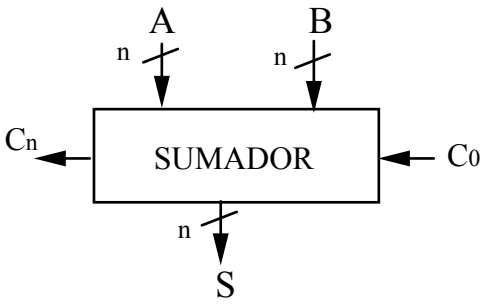


2.6.2. Sumador de n bits

A partir del sumador de 1 bit podemos construir sumadores de n bits fácilmente. Para ello se deben conectar en cascada n sumadores de 1 bit, pasando la llevada de uno a otro. Al circuito resultante se le llama sumador en serie (ripple carry adder RCA) o en ocasiones también sumador de propagación de la llevada (CPA).



Esta cadena de n sumadores es un sumador de n bits que representaremos así:



El problema principal de estos sumadores es su retardo: el resultado final no es estable hasta que la llevada (carry) haya terminado de propagarse. El caso peor es aquel en el que se produce la llevada en el bit de menos peso y ésta se propaga hasta el bit de más peso. El retardo de este sumador es proporcional a n, y resulta ser:

$$\text{retardo} = 2\Delta\,(n-1) + 2\Delta = 2n\Delta$$

Ejemplo:

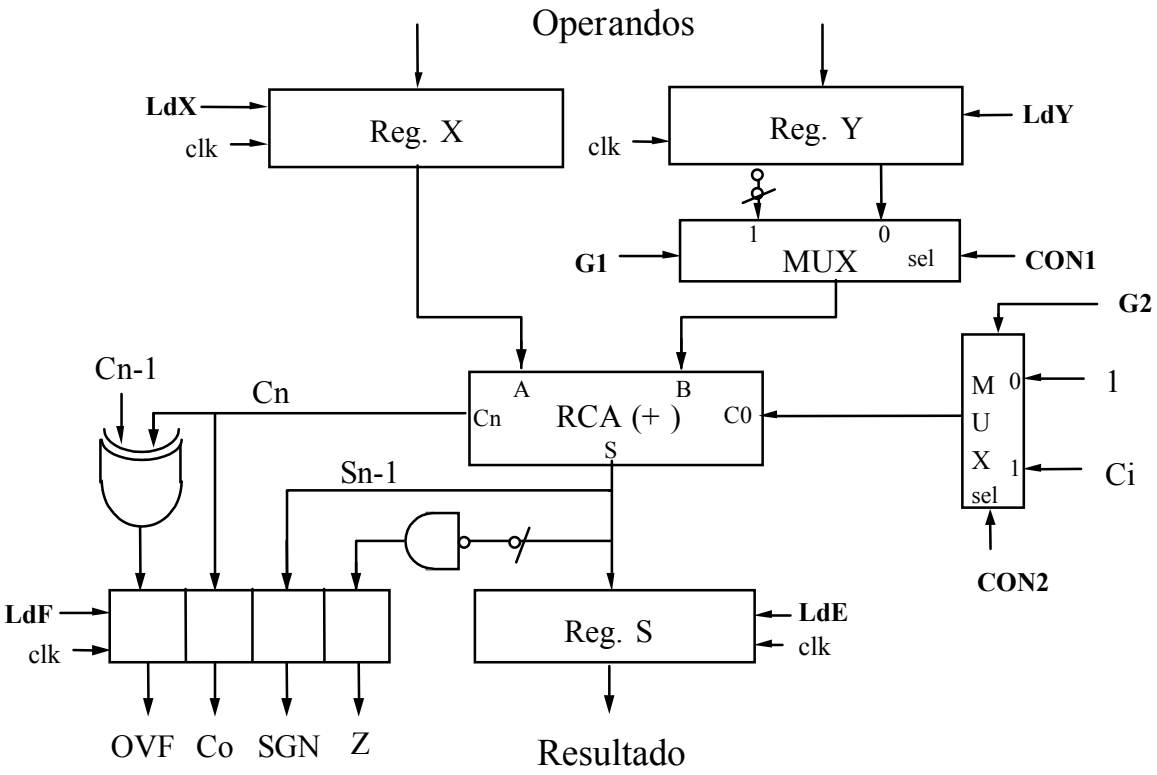
```
0 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1
0 0 1 0 1 0 1 0 1 0 1 0 0 1 1 1
-------------------------------
0 1 1 0 0 0 0 0 0 1 0 1 0 1 0 0
```

En este caso, la cadena más larga de propagación de la llevada es de 6 bits. Por lo tanto, si bien el tiempo de respuesta general es 32Δ , para estos operandos sería 14Δ. De todos modos, como no sabemos qué números se van a sumar, el tiempo de respuesta general será 2nΔ (el peor caso).

2.7. Implementación de un sumador/restador en complemento a la base

La siguiente figura representa implementación típica de una unidad aritmética sencilla para operaciones de enteros en complemento a 2. Los operandos se cargan en los registros X e Y y el resultado en el registro S. Junto al resultado, se generan 4 flags

OVF	desbordamiento
SGN	signo
Z	cero
Co	carry out



Se deja como ejercicio analizar las operaciones que se pueden realizar con este circuito, así como especificar las señales de control necesarias para las mismas.

3. SUMADORES RÁPIDOS

Tal y como hemos comentado en apartados anteriores, una de las principales características de un sumador es su tiempo de respuesta. En el caso del sumador en serie que hemos diseñado en el apartado anterior su tiempo de respuesta es proporcional al número de bits. Sin embargo, cuando se utiliza un sumador en serie -RCA, CPA- sabemos que el tiempo necesario para obtener el resultado no es constante; se deben generar y propagar los bits de llevada y esa propagación depende de los números que se suman. Por ejemplo, cuando se realiza la suma 0110 + 1001, no se generan bits de llevada, por lo que el resultado de la suma se obtiene en todos los bits a la vez. En este caso, el tiempo de respuesta es mínimo, 2Δ. Por otra parte, cuando se suman 1111 y 0001, la llevada que se genera en el primer bit se propaga hasta la última posición, y los bits de suma van cambiando mientras la llevada "avanza" hasta llegar a la última posición; en estas circunstancias el tiempo de respuesta es máximo, $2\Delta*n = 8\Delta$. El tiempo de respuesta que se asigna a un sumador RCA es el que se necesita en el peor de los casos, dado que la mayoría de las veces no se conocen los números que se suman. Debido a esto, el tiempo de respuesta es proporcional al tamaño en bits de los números a sumar. Como por otro lado, la suma es la operación básica de cualquier unidad aritmética, nos vemos obligados a buscar algoritmos y circuitos más rápidos para la suma.

Una forma de acelerar la suma, consiste en detectar la finalización de la propagación de la llevada, dado que en ese momento se puede decir que la suma está estabilizada. Aunque esta técnica es fácilmente aplicable, no se utiliza en las unidades aritméticas de los ordenadores porque da lugar a un tiempo de respuesta variable, dependiente de los números que se suman, y eso no interesa dentro de un computador.

En los próximos apartados analizaremos cuatro circuitos sumadores que se utilizan en las unidades aritméticas; por un lado tres sumadores de tipo CPA, que propagan la llevada, pero de manera eficiente, y por otro, los sumadores CSA que procesan los bits de llevada de una manera especial.

3.1 Generación de la llevada (carry look-ahead)

Para poder superar el problema de la propagación de la llevada es importante conocer cómo se genera y se propaga la misma. Cuando sumamos dos bits lo que ocurre es lo siguiente:

A_i	B_i	C_i	C_{i+1}	
0	0	0/1	0	no se genera
0	1	0/1	0/1	Si C_i es 1, se **propaga** la llevada
1	0	0/1	0/1	Si C_i es 1, se **propaga** la llevada
1	1	0/1	1	Se **genera** la llevada

Basandonos en la tabla anterior, definiremos dos nuevas funciones:

Generador de llevada	$G_i = A_i\ B_i$	(generator)
Propagador de llevada	$P_i = A_i \oplus B_i$	(propagator)

El generador (G) y el propagador (P) se pueden utilizar para expresar las funciones de suma y llevada S_i y C_{i+1}:

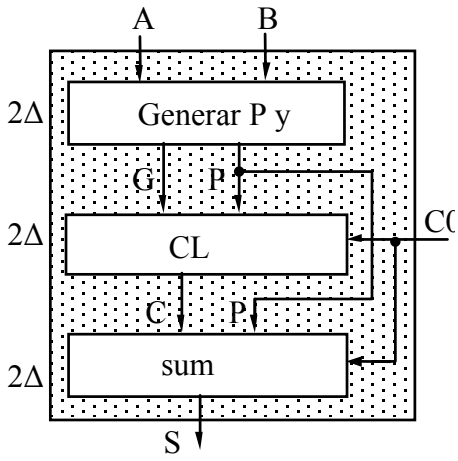
$$\mathbf{S_i} = A_i \oplus B_i \oplus C_i = \mathbf{P_i} \oplus \mathbf{C_i}$$
$$\mathbf{C_{i+1}} = A_i B_i + C_i (A_i \oplus B_i) = \mathbf{G_i + P_i C_i}$$

Nos interesa la última ecuación dado que para generar los bits C_i definimos una recurrencia que si desarrollamos:

$$C_1 = G_0 + C_0 P_0$$
$$C_2 = G_1 + C_1 P_1 = G_1 + G_0 P_1 + C_0 P_0 P_1$$
$$C_3 = G_2 + C_2 P_2 = G_2 + G_1 P_2 + G_0 P_1 P_2 + C_0 P_0 P_1 P_2$$
$$C_{i+1} = G_i + G_{i-1}\prod_{j=i}^{i} P_j + G_{i-2}\prod_{j=i-1}^{i} P_j + \ldots + G_0\prod_{j=1}^{i} P_j + C_0\prod_{j=0}^{i} P_j$$

La consecuencia es clara: para generar el bit C_{i+1} no hace falta conocer el C_i; es suficiente conocer todos los P_i y G_i anteriores (y el C_0). Dado que los bits P_i y G_i se pueden generar de manera independiente para todos los bits, todos los bits C_i se obtendrán **a la vez**. De esta forma desaparece la propagación de los bits de llevada en serie. La suma se obtiene de modo muy eficiente:

- generar todos los P_i y G_i en paralelo:
  tiempo = 2Δ (función xor)
- generar, también en paralelo, todos los bits C_i mediante la ecuación anterior:
  tiempo = 2Δ (and + or)
- generar la suma, todos los bits a la vez:
  tiempo = 2Δ (xor)

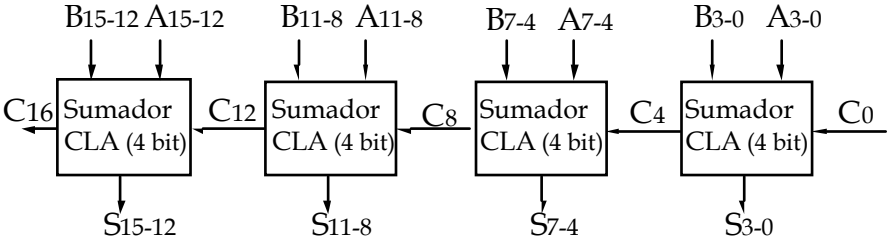


El tiempo de respuesta de este nuevo sumador llamado **CLA** (*carry look-ahead adder*) es, por tanto, 6Δ. Además hemos conseguido que sea **constante** para "cualquier" número de bits.

En cualquier caso, el análisis que hemos realizado presenta un problema en el tiempo de respuesta del circuito que genera los bits de llevada. Aunque en teoría este tiempo es de 2Δ –una puerta *and* y una *or*–, en la práctica no es posible mantener ese tiempo cuando crece el número de bits, es decir, no tiene el mismo tiempo de respuesta una *and* de 2 entradas que una de 8. Debido a esto, es necesario limitar el

tamaño del circuito que genera los bits de llevada (2-4 bits). ¿Qué podemos hacer entonces cuando queramos sumar dos números de 64 bits?

Una posibilidad consiste en combinar sumadores CLA con otros en serie, calculando algunos de los bits de llevada en paralelo mientras que otros se calculan en serie. Esta es precisamente la técnica utilizada en el sumador de 16 bits que se presenta en la figura siguiente, donde en cada módulo de 4 bits los bits de llevada internos se calculan en paralelo mediante el circuito CLA antes de hacer la suma. Sin embargo las llevadas entre módulos (C_4, C_8 ...) se pasan en serie. En cualquier caso, ésta no es la estructura que se utiliza habitualmente.



3.1.1 Estructuras CLA de múltiples niveles (árboles CLA)

Los circuitos CLA que hemos presentado, se pueden organizar en forma de árbol, para poder calcular todos los bits C_i en paralelo. Fíjate de nuevo en las ecuaciones que generan los bits C_i:

$$C_2 = G_1 + C_1 P_1 = G_1 + G_0 P_1 + C_0 P_0 P_1$$

Se podrían reescribir de esta otra forma:

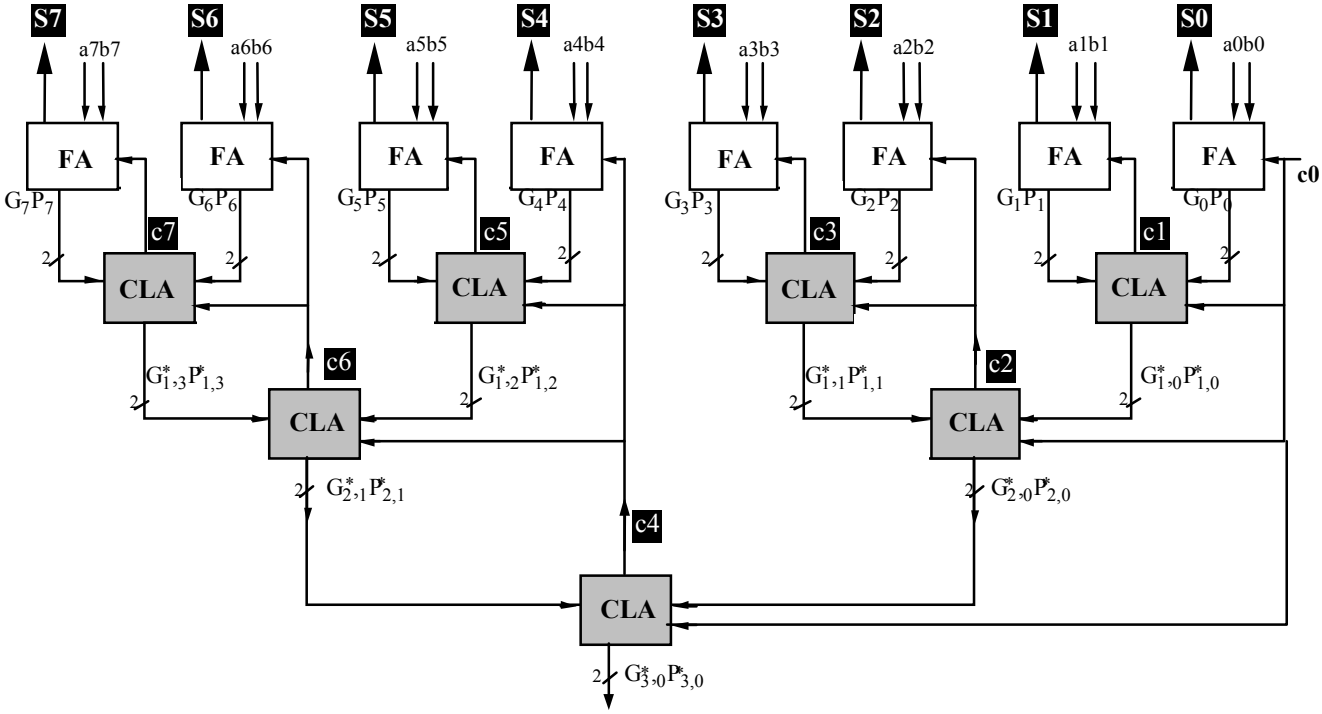
$$C_2 = G^* + C_0 P^* \qquad \text{donde} \qquad G^* = G_1 + G_0 P_1 \quad \text{y} \quad P^* = P_0 P_1.$$

Es decir, C_2 se genera si lo hace en el bloque de dos bits (G*) o si propaga el generado en el bloque anterior. Por tanto, para calcular C_2 se puede organizar otro nivel de CLA que tomará G* y P* del nivel anterior para con ellos calcular C_2. Los bloques CLA de un nivel, no generan su último bit C_i, sino los correspondientes G* y P* (a veces, se les llama también Gi,j para expresar el bit j del nivel i). Para calcular estas dos funciones no hace falta conocer la llevada de entrada y esto permite crear un árbol tal y como representa la figura. En la misma, aparece un sumador CLA de 8 bits, con circuitos CLA de 2 bits (evidentemente, esta estructura se puede generalizar al número de bits que se desee).

El tiempo de respuesta del sumador de la figura es el siguiente:

(1) generar los bits G_i y P_i a partir de los bits de datos --> 2Δ
(2) generar todos los bits G^* y P^* de primer nivel de CLA --> 2Δ
(3) generar todos los bits G^* y P^* de segundo nivel de CLA --> 2Δ
(4) generar los bits G^* y P^* y C_4 de tercer nivel de CLA --> 2Δ
(3) generar C_6 y C_5 (C_1 y C_2 ya están calculados) --> 2Δ
(2) generar C_7 (C_3 ya está calculado) --> 2Δ
(1) conseguir la suma --> 2Δ

Por lo tanto, el tiempo de respuesta es 14Δ.



(La llevada de salida de este sumador es $C_8 = G_{3,0} + C_0 P_{3,0}$)

Aunque en este ejemplo se hayan utilizado módulos de 2 bits, también son muy comunes los de 4 bits. Generalizando la idea anterior:

$$C_4 = G_3 + C_3 P_3 = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + C_0 P_0 P_1 P_2 P_3$$
$$C_4 = G^* + C_0 P^*$$
donde $$G^* = G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3$$
$$P^* = P_0 P_1 P_2 P_3$$

Se ve claramente que, en este caso, obtener las funciones P* y G* exige más hardware y, quizá, más tiempo. Sin embargo, si se utilizaran bloques CLA de 4 bits, el número de niveles del árbol sería menor y por tanto, necesitaríamos menos tiempo para obtener la suma (suponiendo que el tiempo de respuesta de los circuitos CLA se mantuviera). En general, el sumador se organiza en $\log_m n$ niveles (n = número de bits de los operandos, m = número de bits del CLA). Si admitimos que el tiempo de respuesta de cada CLA es 4Δ –2Δpara generar los bits G/P y 2Δ para los bits C– el tiempo de respuesta de estos sumadores es :

$$t = 4\Delta\lceil \log_m n \rceil + 2\Delta$$

3.2 Sumadores rápidos

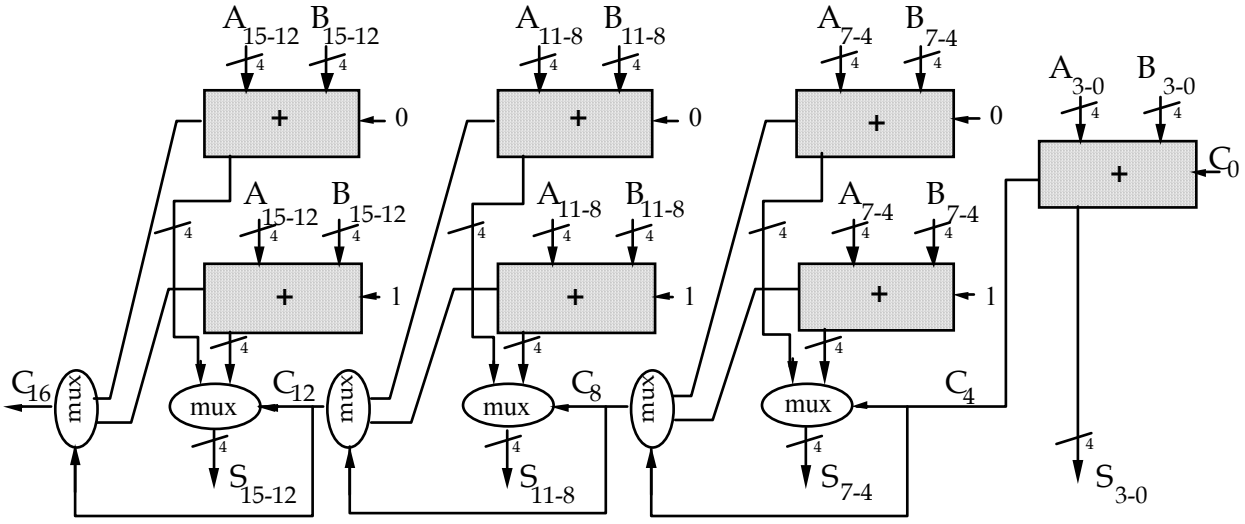
Como hemos visto, los sumadores básicos, tipo RCA, son lentos pero baratos en cuanto a hardware, mientras que el sumador CLA es mucho más rápido pero también mucho más caro, debido a la cantidad de hardware que utiliza para generar los bits de llevada. Al ser el sumador el elemento básico de una unidad aritmética, han surgido muchos y diversos diseños cuyo objetivo principal era mejorar la velocidad y disminuir el coste. Vamos a presentar dos que se utilizan en los diseños VLSI *(carry-select adder* y *carry-skip adder)* y que se sitúan entre los dos ya analizados, tanto en tiempo de respuesta, como en el hardware necesario.

3.2.1 Sumador Carry-Select

La idea es bien sencilla: se divide la suma, por ejemplo, en módulos de 4 bits. Cada uno de estos módulos necesita el bit de llevada generado en el módulo anterior (C_4, C_8, C_{12} ...), pero en vez de esperar, se duplica el módulo de suma de manera que a un módulo se le da 0 como entrada de llevada y al otro 1. Cuando se calcula la llevada real en el módulo anterior, se elige entre las respuestas de los dos módulos tanto la suma como la llevada de dicho módulo.

En la figura a continuación, se presenta un sumador de 16 bits de este tipo. La suma se realiza en paralelo en todos los módulos y a continuación se realiza la selección en serie: con el bit C_4 se eligen los bits $S_{4\text{-}7}$ y C_8 reales, después, con C_8 se eligen $S_{8\text{-}11}$ y C_{12}, y por último con C_{12} los bits $S_{12\text{-}15}$ y C_{16}.

El bloque básico de esta estructura es el sumador RCA de 4 bits cuyo retardo es de 8Δ según hemos visto en el capítulo anterior. Si suponemos un retardo de 2Δ para los multiplexores, el tiempo de respuesta del sumador completo será: 8Δ+2Δ+2Δ+2Δ=14Δ.



En general, si n es el número de bits de los operandos y k el número de bits de los sumadores RCA, el tiempo de respuesta será el siguiente:

$$t = \left(k + \frac{n}{k} - 1\right) * 2\Delta$$

3.2.2 Sumador Carry-Skip

Tal y como hemos visto, el bit de llevada de un bloque de n bits en un sumador CLA se calcula como sigue:

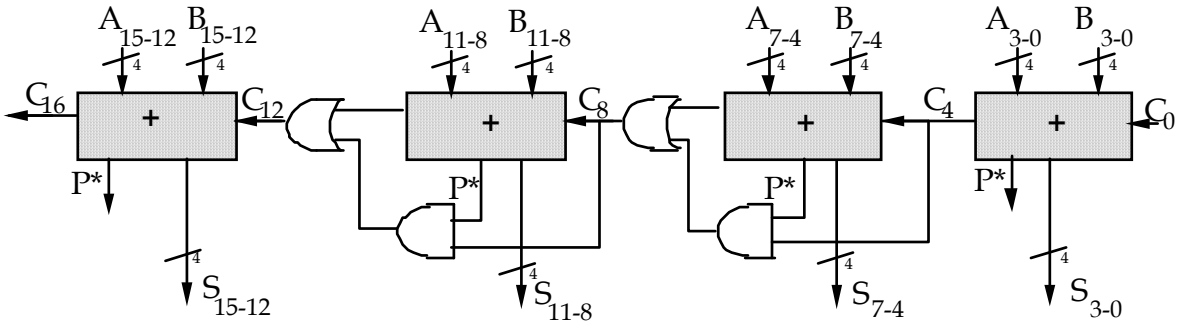
$$Cin_{m+1} = Cout_m = G^* + Cin_m P^*$$

Respecto al hardware, la función G* es la más "cara", y además, su complejidad se incrementa rápidamente con el número de bits (debido a lo cual, hemos limitado los circuitos CLA a 2-4 bits). La función P* sin embargo, no es tan cara.

Existe otra forma de limitar el coste, que consiste en utilizar solamente las funciones P*. Si no se utiliza G*, el bit de llevada que se puede generar en un bloque habrá que generarlo en serie. La ecuación anterior la escribiremos como:

$$Cin_{m+1} = C'out_m + Cin_m P^*$$

Es decir, en un módulo tendremos llevada de entrada, si ésta se genera en el módulo inmediatamente anterior, o bien, si dicho módulo propaga la de algún módulo anterior a él. En la figura que se presenta a continuación podemos ver un sumador carry-skip de 16 bits, formado por sumadores serie de 4 bits. El primer sumador generara su llevada, C_4, en un tiempo 8Δ. En el peor de los casos, la propagación de la llevada hasta el último sumador necesitará 4Δ, con lo cual, en 12Δ tendremos C_{12}; después de otros 8Δ el último sumador ofrecerá su resultado, con lo cual el retardo total será de 20Δ.

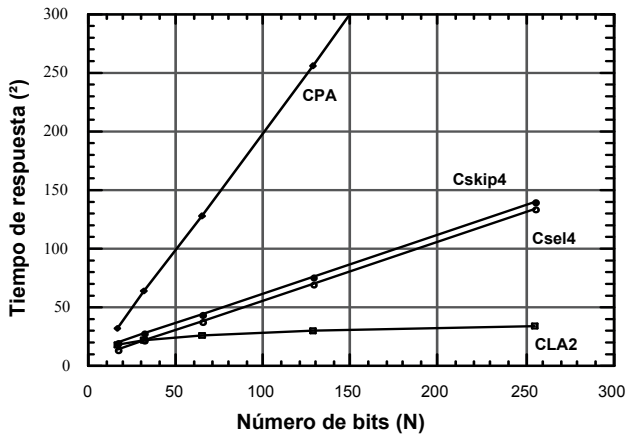


En general, si el número de bits es n y el de los módulos es k, el tiempo de respuesta será:

$$\left(2k+\frac{n}{k}-2\right)*2\Delta$$

En la tabla siguiente se presentan los diferentes tiempos de respuesta para diferentes números de bits de los operandos, y en la gráfica se puede ver la relación entre el tamaño y el tiempo de respuesta de cada sumador.

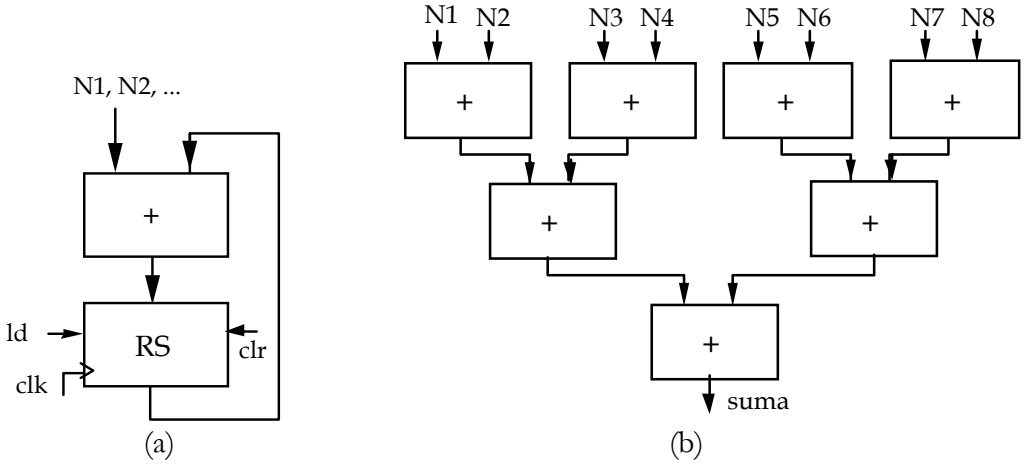
Sumador	**Tiempo respuesta**	**n = 64 bits**	**n = 128 bits**
RCA	$n2\Delta$	128Δ	256Δ
CLA (de 2 bits)	$4\Delta \log_2 n + 2\Delta$	26Δ	30Δ
Carry Select (k=4)	$\left(k+\frac{n}{k}-1\right)*2\Delta$	38Δ	70Δ
Carry Skip (k=4)	$\left(2k+\frac{n}{k}-2\right)*2\Delta$	44Δ	76Δ



Considerando órdenes de magnitud, es posible demostrar que el tiempo de respuesta de los sumadores serie es de O(n); el del sumador CLA de O(log n) y el de los sumadores Carry-Select y Carry-Skip de $O(n^{1/2})$.

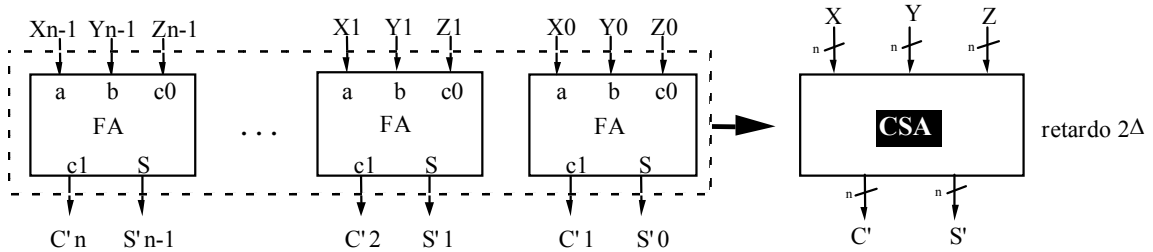
3.3 Sumadores multioperando (carry save adders)

Hasta ahora hemos visto sumadores de sólo dos operandos. Supongamos que queremos sumar 8 números. En los esquemas de la figura se pueden ver dos estructuras que podrían realizar esta operación bien en serie, esquema (a), bien en paralelo, esquema (b). En el primer caso, los números se suman de uno en uno y las sumas parciales se guardan en el registro RS. Aunque en este esquema se utilice poco hardware (ver figura (a)), el tiempo de respuesta es alto: para sumar k números de n bits el tiempo será $T = k(2n\Delta + t_r)$, donde t_r es el tiempo de carga del registro: el tiempo para sumar 8 números de 16 bits, considerando $t_r = \Delta$, es 264Δ. En el segundo caso por el contrario, las sumas se hacen en paralelo, con la intención de agilizar la operación (ver figura (b)); los sumadores se organizan en diferentes niveles y en consecuencia el tiempo de respuesta es menor. Se ve que el hardware necesario es mucho más. Siguiendo con el ejemplo, el tiempo necesario para sumar los 8 números de 16 bits utilizando el esquema de la figura sería 36Δ.



En el esquema (b) se podrían poner registros entre los diferentes niveles: el sumador quedaría "segmentado" y sería posible comenzar con una nueva suma en cada ciclo. En dicho caso, el tiempo de respuesta no sería el que se ha comentado anteriormente.

Para acelerar este proceso, desarrollaremos una nueva estructura, los sumadores de llevada retardada (carry save adders, CSA). La unidad básica de estos sumadores es un sumador común de 1 bit (FA), pero la llevada no se pasa de una unidad a otra, sino que se trata la entrada de llevada como una entrada más, y se aprovecha para introducir un nuevo operando. Como resultado obtenemos dos vectores de n bits, la "semi"suma parcial S', y la "semi"llevada C'. El esquema del circuito es el siguiente:



Hay que tener en cuenta que de este modo no se obtiene la suma de 3 operandos, dado que todavía no se han propagado los bits de llevada. Se han sumado 3 números y se han obtenido 2 que a su vez hay que sumar, esta vez sí, propagando la llevada de un bit a otro. Por lo tanto, hasta llegar a la suma final, los números se suman en un tiempo mínimo dado que no se propaga la llevada entre los bits. Así pues, es posible sumar muchos operandos eficientemente utilizando circuitos CSA, ya sea utilizando un único sumador CSA, o muchos organizados en estructuras tipo árbol.

3.3.1 Algoritmo iterativo para sumar k números de n bits

Tal como se presenta en la figura, la idea es utilizar un único sumador CSA para sumar k números en serie (en cada paso se introduce un nuevo número). Los vectores de salida del CSA, C' y S', se guardan en dos registros para poder utilizarlos la siguiente iteración. Con el símbolo de la flecha se pretende representar el desplazamiento de un bit (los bits de llevada hay que sumarlos desplazados una posición). Cuando se han introducido todos los números, todavía no se tiene la suma finalizada, dado que al final hay que sumar los bits de C'. Para realizar este último paso hay dos posibilidades: o se coloca un sumador básico que propague la llevada para realizar esta última suma, o se continúa con las iteraciones introduciendo 0s en la entrada Z hasta que todos los bits de C' sean 0, lo cual indica que ha finalizado la propagación de los bits de llevada y la suma de los k números está en la salida del registro (o en la salida S del CSA). Con esta segunda opción, no se añade hardware, pero en el peor de los casos hay que realizar n-1 pasos de suma (siendo n el número de bits). Veamos un ejemplo:

N1 = 0 0 1 0 1 0 (10) N2 = 0 0 1 1 0 0 (12) N3 = 0 0 0 1 1 0 (6)
N4 = 0 0 0 1 0 1 (5) N5 = 0 0 0 1 1 1 (7)

❶ S' = 0 0 0 0 0 0 inicialización
C' = 0 0 0 0 0 0

---

❷ X = 0 0 0 0 0 0 S'0
Y = 00 0 0 0 0 2∗C'0 (←)
Z = 0 0 1 0 1 0 N1

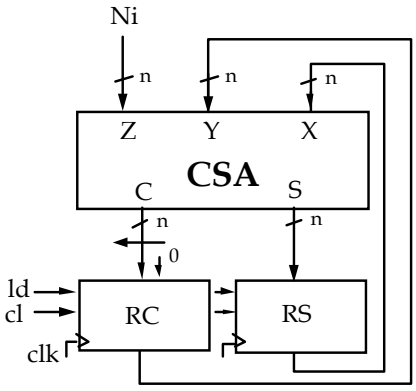
S'= 0 0 1 0 1 0
C'= 0 0 0 0 0 0

X = 0 0 1 0 1 0 S'1
Y = 0 0 0 0 0 0 2∗C'1 (←)
Z = 0 0 1 1 0 0 N2

S'= 0 0 0 1 1 0
C'= 0 0 1 0 0 0

X = 0 0 0 1 1 0 S'2
Y = 0 1 0 0 0 0 2∗C'2 (←)
Z = 0 0 0 1 1 0 N3

S'= 0 1 0 0 0 0
C'= 0 0 0 1 1 0



X = 0 1 0 0 0 0 S'3
Y = 0 0 1 1 0 0 2∗C'3 (←)
Z = 0 0 0 1 0 1 N4

S'= 0 1 1 0 0 1
C'= 0 0 0 1 0 0

X = 0 1 1 0 0 1 S'4
Y = 0 0 1 0 0 0 2∗C'4 (←)
Z = 0 0 0 1 1 1 N5

S'= 0 1 0 1 1 0
C'= 0 0 1 0 0 1

---

❸.a Se suman al final C' y S'

```
0 1 0 1 1 0     S'5
0 1 0 0 1 0     2*C'5
-----------
1 0 1 0 0 0     FIN
```

❸.b Se continúa con las iteraciones, siendo Z=0, hasta que todos los bits C' sean 0

```
X = 0 1 0 1 1 0   S'5
Y = 0 1 0 0 1 0   2*C'5   (←)
Z = 0 0 0 0 0 0   0,
-----------
S'= 0 0 0 1 0 0
C'= 0 1 0 0 1 0

X = 0 0 0 1 0 0   S'6
Y = 1 0 0 1 0 0   2*C'6   (←)
Z = 0 0 0 0 0 0
-----------
S'= 1 0 0 0 0 0
C'= 0 0 0 1 0 0
X = 1 0 0 0 0 0   S'7
Y = 0 0 1 0 0 0   2*C'7
Z = 0 0 0 0 0 0
-----------
S'= 1 0 1 0 0 0   = 40   FIN
C'= 0 0 0 0 0 0
```

Si para sumar los últimos C' y S' se utiliza un sumador RCA, el tiempo de respuesta será:

$t_S = k \cdot (2\Delta + t_r) + 2n\Delta$

(Si continuáramos con las iteraciones (otras n-1 iteraciones), el tiempo de respuesta sería:

$t_S = (k+n-1) \cdot (2\Delta + t_r)$)

Se ha comentado al comienzo, que utilizando un único sumador RCA para sumar 8 números de 16 bits necesitamos 264Δ de tiempo. Vemos ahora que con un sumador CSA y un sumador RCA para la suma final, el tiempo de respuesta sería de 56Δ (considerando la opción 3.a: 8(2Δ + Δ) + 2*16Δ).

3.3.2 Esquema de CSA multinivel. Árboles de Wallace

Al igual que con los sumadores comunes, se puede realizar la suma en paralelo de múltiples números sin entrar en iteraciones. Para ello, se construyen estructuras en árbol con circuitos CSA generando lo que se denominan **árboles de Wallace**. El objetivo de estas estructuras es conseguir un paralelismo alto en los cálculos. El diseño óptimo será el que tenga el mínimo número de niveles CSA.

En la figura se presentan sumadores para 4, 5, 6 y 7 números. Al final, en cualquier caso, habrá que sumar los vectores C y S para conseguir la suma.

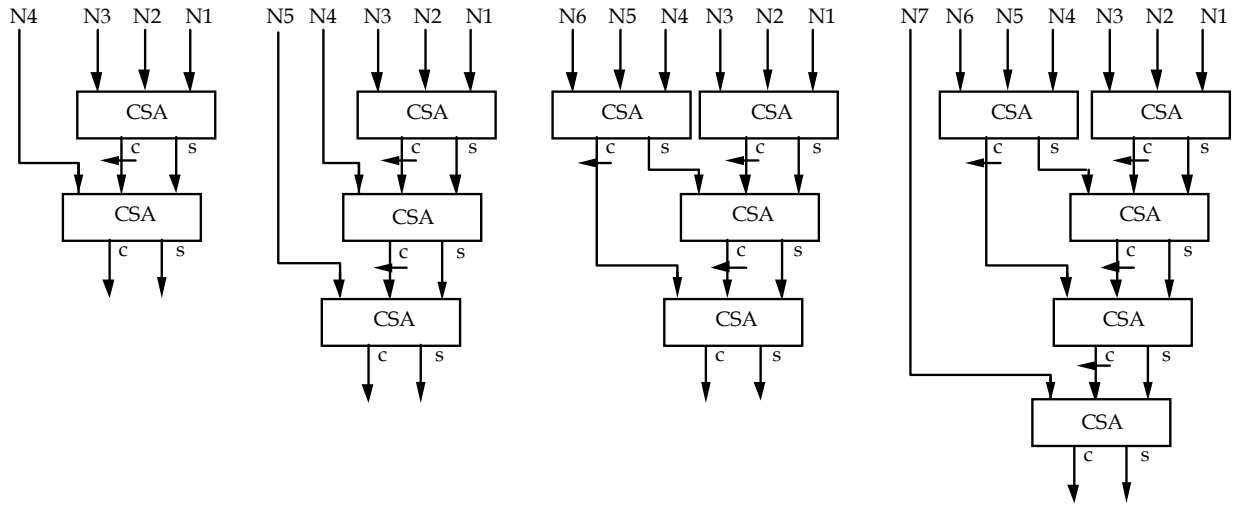
Como se puede ver, para sumar N números, necesitamos N-2 sumadores CSA. El tiempo de respuesta total depende del número de niveles del árbol:

$$2\Delta \cdot nm + 2n\Delta \qquad (nm = \text{número de niveles})$$

Sea *nm* el número de niveles del árbol y *O(nm)* el máximo número de operandos que se pueden procesar en paralelo en ese árbol. La relación entre ellos es la siguiente:

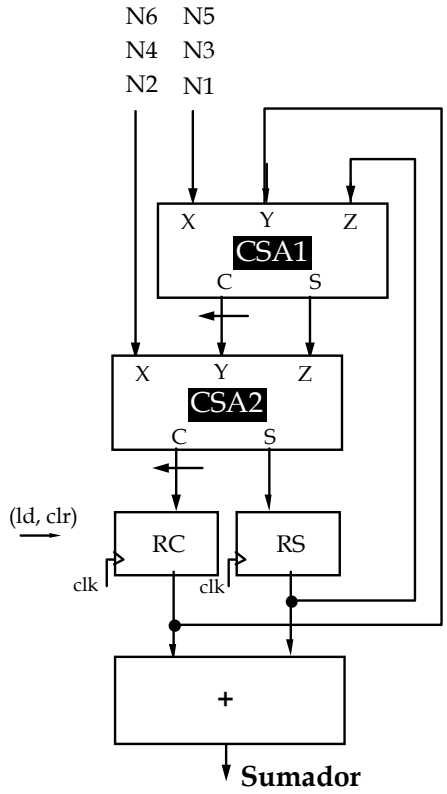
$$O(nm) = 3\left\lfloor \frac{O(n\,m\text{-}1)}{2} \right\rfloor + O(nm\text{-}1) \bmod 2$$

donde O(1) = 3.



3.3.3 Esquema de CSA mixto

Los dos esquemas presentados, el iterativo y el de árboles de Wallace, se pueden combinar para crear sumadores CSA multinivel. La siguiente figura presenta un ejemplo de este tipo, donde los bucles de suma se realizan con dos niveles de CSA. Se intenta conseguir un compromiso entre el coste y tiempo.



No es difícil diseñar un circuito CSA/RCA, que con una señal de control, realice la suma CSA o la suma común. Esto se propone como ejercicio a realizar por el lector.

4. MULTIPLICACIÓN DE NÚMEROS NATURALES Y ENTEROS

La mayoría de los algoritmos de multiplicación se basan en el método de suma y desplazamiento. Por ello, analizaremos los desplazamientos previamente.

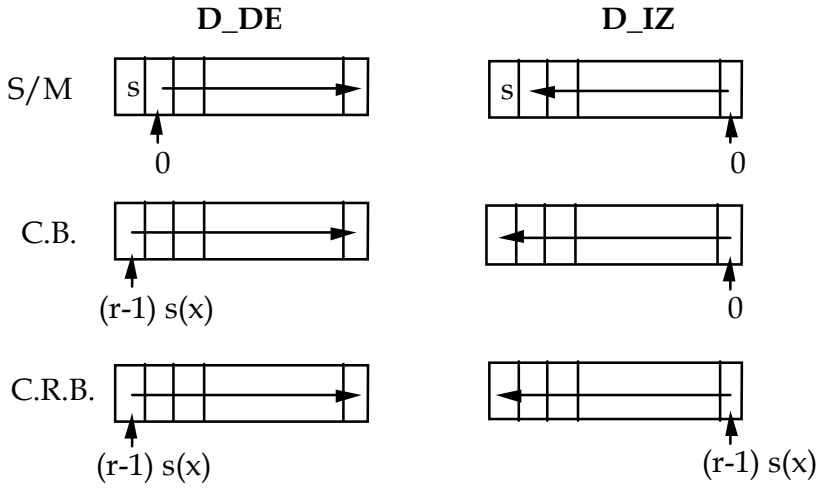
4.1. Desplazamientos

Los desplazamientos pueden ser hacia la derecha y hacia la izquierda. La definición del desplazamiento de un dígito es la siguiente:

$$D\_IZ\ (x) = r * x$$
$$D\_DE\ (x) = x/r + e \quad \text{donde } |e| < r$$

El algoritmo de desplazamiento para los tres casos de representación estudiados se puede ver de forma gráfica en la siguiente figura. En el caso de Signo y Magnitud, sólo es necesario desplazar la magnitud del número, manteniendo el signo, y rellenar de ceros los "huecos" que van quedando al desplazar. En los otros dos casos, se desplaza todo el número rellenando, en general, con una extensión del bit de signo.

En los tres casos supondremos que no se produce desbordamiento.



s(x)= 0 si x>0
s(x)= 1 si x<0

4.2. Multiplicación de números naturales

Sean x e y naturales representados por vectores de n dígitos X e Y en un sistema convencional de base r. La operación de multiplicar produce p = x * y, representándose p mediante un vector de dígitos P. El método usual de multiplicación mediante lápiz y papel es el siguiente:

$$
\begin{array}{cccccc}
 & & X_3 & X_2 & X_1 & X_0 \\
 & & Y_3 & Y_2 & Y_1 & Y_0 \\
\hline
 & & X_3Y_0 & X_2Y_0 & X_1Y_0 & X_0Y_0 \\
 & X_3Y_1 & X_2Y_1 & X_1Y_1 & X_0Y_1 & \\
X_3Y_2 & X_2Y_2 & X_1Y_2 & X_0Y_2 & & \\
X_3Y_3 & X_2Y_3 & X_1Y_3 & X_0Y_3 & &
\end{array}
\qquad
\begin{array}{l}
\\ \\ \\
\times\, Y_0 \\
\times\, Y_1\, r \\
\times\, Y_2\, r^2 \\
\times\, Y_3\, r^3
\end{array}
$$

que se puede describir por la siguiente expresión:

$$x * y = x \sum_{i=0}^{n-1} Y_i\ r^i = \sum_{i=0}^{n-1} x\ r^i\ Y_i$$

Esta operación indica que primero se calculan los n términos xr^iY_i y finalmente se efectúa la suma. El cálculo del i-ésimo término requiere un desplazamiento a la izquierda de X y una multiplicación por un dígito Y_i en base r. Una mecanización directa de este método no es muy adecuada, ya que requiere excesiva memoria y además se necesita capacidad para realizar sumas con operandos múltiples.

En lugar de este método, puede utilizarse el algoritmo de **"suma + desplazamiento"**, con el que se genera una secuencia de sumas parciales. Para ello se generan los términos xY_j y se van sumando uno a uno, teniendo en cuenta que antes de sumar los productos parciales, hay que desplazarlos a la izquierda (multiplicar por r).

Sin embargo, hay otra posibilidad que consiste en desplazar la suma parcial hacia la derecha (dividir por r) en lugar de desplazar los sumandos a la izquierda.

Este algoritmo puede definirse mediante la siguiente recurrencia:

$$p^{(0)} = 0$$
$$p^{(j+1)} = (p^{(j)} + r^n \times Y_j)1/r \qquad \text{para } j=0,1, ..., n-1$$

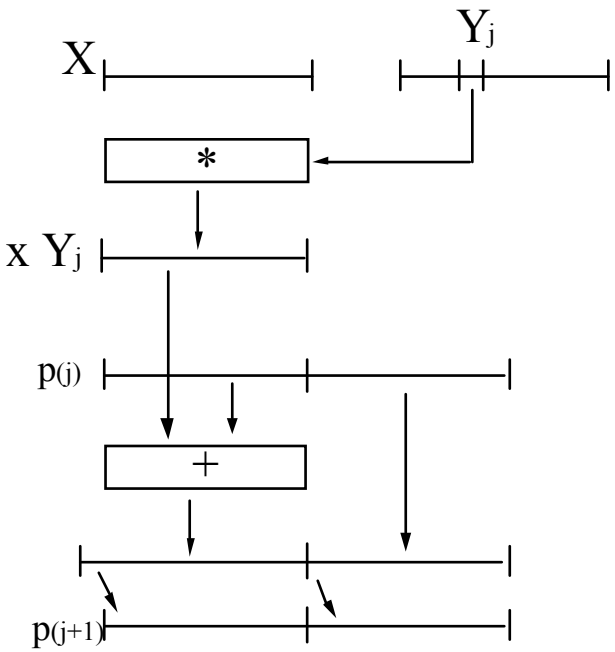
La expansión de esta recurrencia da lugar al producto

$$p^{(n)} = x * y$$

Fijémonos, por ejemplo, en el caso de trabajar con 4 dígitos. Entonces tenemos que:

$$p^{(0)} = 0$$
$$p^{(1)} = (r^4 \times Y_0)\ 1/r = r^3 \times Y_0$$
$$P^{(2)} = (r^3 \times Y_0 + r^4 \times Y_1)\ 1/r = r^2 \times Y_0 + r^3 \times Y_1$$
$$p(3) = (r^2 \times Y_0 + r^3 \times Y_1 + r^4 \times Y_2)\ 1/r = r^1 \times Y_0 + r^2 \times Y_1 + r^3 \times Y_2$$
$$p(4) = (r^1 \times Y_0 + r^2 \times Y_1 + r^3 \times Y_2 + r^4 \times Y_3)\ 1/r = \times Y_0 + r^1 \times Y_1 + r^2 \times Y_2 + r^3 \times Y_3$$

Por tanto, el producto de dos números de n bits se obtiene en n pasos mediante este método indirecto. Los productos parciales se suman en los bits de más peso, y el resultado se desplaza un bit a la derecha (1/r). Para ello basta utilizar un sumador de n bits. La siguiente figura representa un paso del algoritmo:



La implementación de este algoritmo resulta especialmente atractiva con r=2 ya que xY_j resulta o bien 0 o bien x dependiendo de que Y_j sea 0 o 1.

A continuación se muestra un ejemplo de la ejecución de este algoritmo.

n =5	r =2	x=23 X = 1 0 1 1 1	y =26 Y = 1 1 0 1 0
xY_0	0 0 0 0 0		
$p^{(0)}$	0 0 0 0 0		
	0 0 0 0 0		
xY_1	1 0 1 1 1		
$p^{(1)}$	0 0 0 0 0	0	
	1 0 1 1 1		
xY_2	0 0 0 0 0		
$p^{(2)}$	0 1 0 1 1	1 0	
	0 1 0 1 1		
xY_3	1 0 1 1 1		
$p^{(3)}$	0 0 1 0 1	1 1 0	
	1 1 1 0 0		
xY4	1 0 1 1 1		
p(4)	0 1 1 1 0	0 1 1 0	
	1 0 0 1 0 1		
p(5)	**1 0 0 1 0**	**1 0 1 1 0**	**= 598**

Obsérvese que puede existir un desbordamiento temporal que se corrige al realizarse el desplazamiento hacia la derecha.

4.3. Multiplicación de enteros (r=2)

4.3.1. Signo Magnitud

La multiplicación de las magnitudes se realiza como multiplicación de números naturales. Al resultado final se le añade el signo, de modo que

$$p_s = x_s \oplus y_s$$

$$p_m = x_m \cdot y_m$$

4.3.2. Complemento a 2

El valor implícito del multiplicando en complemento a 2 es:

$$y = y_e - 2^n \text{signo}(y) = \sum_{i=0}^{n-1} Y_i 2^i - 2^n \text{signo}(y) = \sum_{i=0}^{n-2} Y_i 2^i - 2^{n-1} Y_{n-1}$$

Entonces, para obtener el producto, la operación que hay que realizar es:

$$x^* y = x \sum_{i=0}^{n-2} Y_i 2^i - x Y_{n-1} 2^{n-1}$$

Por tanto, el producto se realiza hasta el bit n-2 de acuerdo al algoritmo que acabamos de ver; a continuación es necesario realizar un paso de corrección:

si $Y_{n-1} = 0$ (número positivo) se suma 0

si $Y_{n-1} = 1$ (número negativo) se suma $-x\ 2^{n-1}$

Las sumas se realizan en complemento a 2, y por tanto hay que considerar la extensión del signo.

4.3.3. Complemento a 1

La multiplicación en C1 también requiere un paso de corrección que puede ser especificado de forma similar al caso de C2. La multiplicación se realiza igual que en el caso anterior hasta el bit n-2. El peso del bit n-1 es $-2^{n-1}+1$

si $Y_{n-1} = 0$ (número positivo) → se suma 0

si $Y_{n-1} = 1$ (número negativo) → se suma $-x\ 2^{n-1} + x$

En este caso, el producto se obtiene

$$x * y = x \sum_{i=0}^{n-2} Y_i\ 2^i\ -\ Y_{n-1}\ (2^{n-1}\ x - x)$$

Para evitar este paso de corrección complicado, otra posibilidad para multiplicar dos números siendo el multiplicador negativo es cambiar el signo del multiplicador (lo que es sencillo en C1), se multiplican los dos números y el resultado final se vuelve a cambiar de signo, obteniendo así el producto correcto.

4.4. Desbordamiento en la multiplicación

Vamos a analizar el desbordamiento en la multiplicación dependiendo del sistema de representación utilizado. Multiplicamos 2 números de n bits y el resultado se almacena en una doble palabra, 2n bits.

- *Signo Magnitud*

En Signo Magnitud el margen de representación de un número de n bits viene dado por el intervalo $-(2^{n-1} -1) \leq x \leq 2^{n-1} -1$. El caso peor es la multiplicación de $(2^{n-1} -1) * (2^{n-1} -1)$, que da como resultado un número cuya magnitud es representable con 2n-2 bits. Necesitaremos otro bit para indicar el signo de forma explícita, es decir, se necesitan 2n-1 bits. Por lo tanto, no hay problemas de desbordamiento. Es usual que el bit restante se mantenga a 0 y no se considere.

- *Complemento a 2*

En C2 el margen de representación de un número de n bits es $-2^{n-1} \leq x \leq 2^{n-1} -1$. El caso de multiplicación más desfavorable es multiplicar $(-2^{n-1}) * (-2^{n-1}) = 2^{2n-2}$. Con 2n bits el intervalo representable es $-2^{2n-1} \leq x \leq 2^{2n-1} –1$, con lo cual el valor anterior es representable con 2n bits y, por tanto, no se produce desbordamiento.

- *Complemento a 1*

En C1 el margen de representación de un número de n bits es $-(2^{n-1} -1) \leq x \leq 2^{n-1} -1$. El caso más desfavorable es la multiplicación $(2^{n-1} -1) * (2^{n-1} -1) < 2^{2n-2}$. Como en el caso anterior, no hay desbordamiento si se utilizan 2n bits para representar el producto, extendiendo el signo del resultado.

4.5. Hardware para el producto en C2

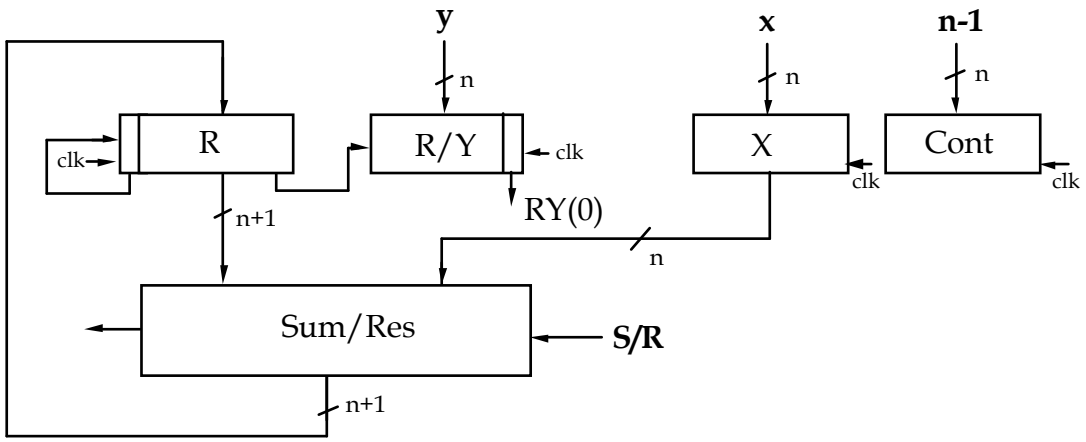
Una modificación habitual del algoritmo de multiplicación consiste en reducir el número de sumas a realizar, evitando las sumas con 0. A este algoritmo se le denomina algoritmo de **"suma o salta".** En cada iteración haremos:

$Y_i = 1$ fi sumar y desplazar
$Y_i = 0$ fi sólo desplazar

Para implementar este algoritmo necesitaremos el siguiente hardware:

- 3 registros: multiplicando (X, n bits), multiplicador (Y, n bits) y multiplicación (R, 2n bits). Con el fin de simplificar el hardware, el multiplicador y multiplicación pueden compartir el registro, ya que inicialmente el resultado parcial es de n bits, luego de n+1, etc. En cada iteración, se pierde el bit del multiplicador que acaba de ser procesado.
  Además, a pesar de que no va a haber desbordamiento en la multiplicación, es posible que las sumas parciales sí produzcan overflow. Para resolver este problema es suficiente realizar la suma en n+1 bits, propagando el signo: los números a sumar son de n bits, pero queremos el resultado en n+1 bits; de este modo, no habrá desbordamiento. Si hiciéramos la suma en n bits, habría que detectar el overflow, para saber si al desplazar a la derecha el resultado habría que introducir el bit Cn o replicar el bit de signo. Como ejemplo, puedes realizar el siguiente producto, -10 * -9, pero utilizando únicamente 5 bits para las sumas parciales y ver qué ocurre.

  Por tanto los registros serán los siguientes:
  - X, registro de n bits, multiplicando
  - R, registro de desplazamiento de n+1 bits, para guardar los bits de más peso del resultado (extendiendo el signo)
  - R/Y, registro desplazamiento de n bits, para guardar los bits de menos peso del resultado y/o el multiplicador.
- Un contador para controlar n productos/sumas parciales.
- Un sumador/restador de n+1 bits en complemento a 2, para realizar tanto las sumas parciales como las restas (en el paso de corrección)



Utilizando este circuito, el algoritmo para multiplicar números de n bits es:

1: X:=x; Cont:=n-1; RY:=y; R:=0;
2: Cont:=Cont-1; if RY(0) = 0 then goto 4;
3: R:=ADD(R, X);
4: D_DE (R-RY); if Cont ≠ 0 then goto 2;
5: if RY(0) = 1then R:=SUB (R,X);
6: D_DE (R-RY); FIN:=1;

Ejemplo

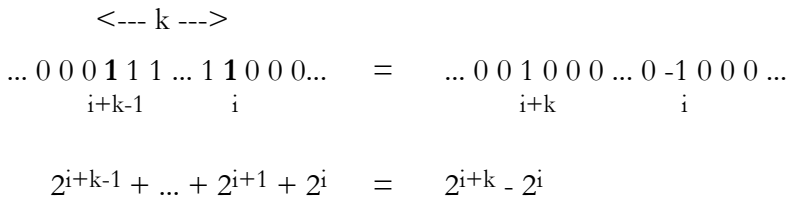
X = 0 0 0 1 0 2
Y = 1 1 0 1 1 -5

	R	**R/Y**	**X**	**CONT**
	0 0 0 0 0 0	1 1 0 1 **1**	0 0 0 1 0	4
+	0 0 0 0 1 0			
	0 0 0 0 1 0			
despl →	0 0 0 0 0 1	0 _ 1 1 0 **1**		3
+	0 0 0 0 1 0			
	0 0 0 0 1 1			
despl →	0 0 0 0 0 1	1 0 _ 1 1 **0**		2
despl →	0 0 0 0 0 0	1 1 0 _ 1 **1**		1
+	0 0 0 0 1 0			
	0 0 0 0 1 0			
despl →	0 0 0 0 0 1	0 1 1 0 _ **1**		0
	0 0 0 0 0 1	0 1 1 0 _ **1**	como es 1 se debe corregir	
	1 1 1 1 1 0		sumando el C2 de 00010	
	1 1 1 1 1 1			
despl →	1 **1 1 1 1 1**	**1 0 1 1 0**	**-10**	(representado con 2n bits)

Como media se realizan n desplazamientos y n/2 sumas (el resto son 0). Sin embargo, en el peor caso habrá que hacer *n* sumas, de modo que si tuviéramos que dar el retardo del multiplicador construido, tendríamos que tener en cuenta ese caso, el peor.

4.6. Algoritmo de Booth

En el algoritmo anterior tenemos una fase de corrección al llegar al último bit del multiplicador: si este bit es 1, la última operación no es una suma sino una resta. De este modo, no se tratan igual los multiplicadores positivos y los negativos. La mayoría de las unidades aritméticas, en lugar de utilizar este algoritmo, utilizan el denominado algoritmo de Booth. Mediante este algoritmo se tratan del mismo modo todos los multiplicadores, ya sean positivos o negativos. Consiste en "recodificar" el multiplicador, pero utilizando un conjunto de dígitos distinto: $D_i = \{-1, 0, 1\}$ (Este conjunto no es canónico). La base de esta recodificación es la que se presenta en la figura:



$$2^{i+k-1} + \ldots + 2^{i+1} + 2^i = 2^{i+k} - 2^i$$

Es decir, si en el código original hay una cadena de k bits a 1, desaparece, y en su lugar se aparece una cadena "más simple", que tiene un 1 al comienzo (en el dígito de más peso), y un -1 al final (en el dígito de menos peso), siendo los dígitos intermedios 0. Por ejemplo:

109	→	0110 1101	→	10-11 0-11-1 (128-32+16-4+2-1 = 109)
-109	→	1001 0011	→	-101-1 010-1 (-128+32-16+4-1= -109)

Tal y como puede apreciarse, en la nueva representación el peso del dígito D_i es 2^i para cualquier dígito, pero ahora puede haber dígitos negativos.

Al hacer la multiplicación podemos utilizar la nueva codificación de modo que la operación a realizar en cada paso será: si se trata de un 0 nada; si es 1 una suma; si es -1 una resta. Todos las iteraciones son iguales, y no hay una especial como ocurría antes. En lo que al número de iteraciones se refiere, no ha cambiado, ya que la nueva codificación tiene el mismo número de dígitos.

Para conseguir esa nueva codificación es necesario analizar dos bits del código original y en función de su valor elegir como nuevo dígito el 0, 1 o -1. La siguiente tabla indica cómo se realiza la elección:

X_i	X_{i-1}	→ **Nuevo dígito**	
0	0	0	
0	1	1	finaliza cadena
1	0	-1	comienza cadena
1	1	0	

Se supone $X_{-1}=0$

Un ejemplo:

A = 0 1 1 1 0 (14)
B = 1 0 1 1 1 (-9) B recodificado, -1 1 0 0 -1

$$-1\ 1\ 0\ 0\ -1 = -1\ 2^0 + 0\ 2^1 + 0\ 2^2 + 1\ 2^3 - 1\ 2^4 = -9$$

0 0 0 0 0 0

-1	**(-A)**	1 1 0 0 1 0		
		1 1 0 0 1 0		
	→	1 1 1 0 0 1	0	
0	→	1 1 1 1 0 0	1 0	
0	→	1 1 1 1 1 0	0 1 0	
1	**(+A)**	0 0 1 1 1 0		
		0 0 1 1 0 0		
	→	0 0 0 1 1 0	0 0 1 0	
-1	**(-A)**	1 1 0 0 1 0		
		1 1 1 0 0 0		
	→	1 **1 1 1 0 0**	**0 0 0 1 0**	(-126)

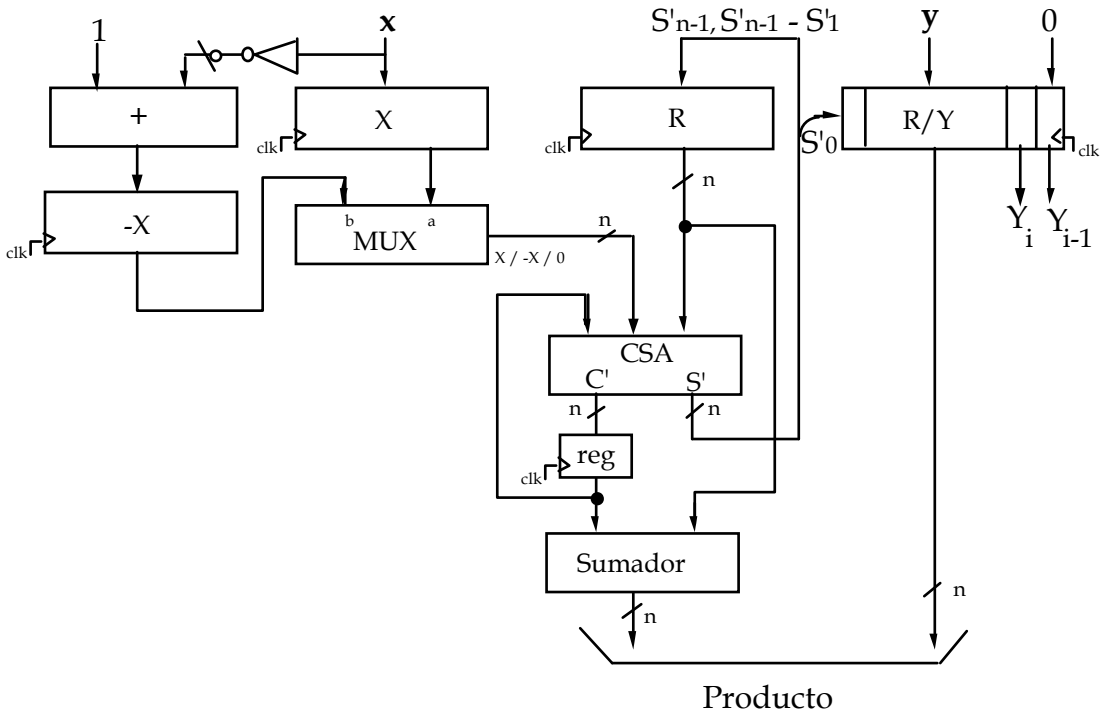
Este algoritmo tiene un efecto lateral: en ocasiones (no siempre), la nueva codificación conseguida tiene más dígitos 0, con lo cual el número de sumas/restas que se realiza es menor. En estos casos, la multiplicación se obtiene antes, y en ocasiones eso puede resultar interesante.

4.7 Algoritmos para multiplicación rápida

Los dos algoritmos vistos necesitan un tiempo $T = nT_s$ para ejecutarse, donde T_s es el tiempo de cada iteración y n el número de iteraciones a realizar. Se puede acelerar el proceso de dos modos: reduciendo T_s, por ejemplo haciendo sumas rápidas, o reduciendo el número de pasos.

4.7.1. Reducir el tiempo de suma: multiplicación con CSA

Las sumas que hay que realizar para conseguir el producto pueden acelerarse utilizando un sumador CSA. Como ya sabes, en un sumador CSA se suman 3 números para conseguir otros dos como resultado: los vectores S' y C'. Por tanto, en la iteración j se suman: $C'^{(j)}$, $S'^{(j)}$ desplazado, y X o -X o 0 (de acuerdo a Booth), consiguiendo así $C'^{(j+1)}$ y $S'^{(j+1)}$. Después, hay que desplazar S' un bit a la derecha (recuerda que C' debe sumarse desplazado un bit respecto a S'). La figura representa un multiplicador mediante CSA. La carga de S' y el desplazamiento se hacen al mismo tiempo: los n-1 bits de más peso se cargan en el registro R, y el de menos peso S´0, en el registro R/Y, realizando así el desplazamiento a la derecha. Al desplazar los bits de S' hacia la derecha, no es necesario desplazar C' hacia la izquierda. La suma final de los vectores S' y C' la haremos mediante un sumador común que propague la llevada.



Veamos un ejemplo: n = 5 r = 2

x = -3 X = 1 1 1 0 1 -X = 0 0 0 1 1
y = -13 Y = 1 0 0 1 1 Y_{booth} = -1 0 1 0 -1

	R/Y
$S'^{(0)}$	0 0 0 0 0 -1 0 1 0 **-1**
$C'^{(0)}$	0 0 0 0 0
$\mathbf{xY_0}$	0 0 0 1 1
$S'^{(1)}$	0 0 0 0 1 1 /-1 0 1 **0**
$C'^{(1)}$	0 0 0 0 0
$\mathbf{xY_1}$	0 0 0 0 0
$S'^{(2)}$	0 0 0 0 0 1 1 / -1 0 **1**
$C'^{(2)}$	0 0 0 0 0
$\mathbf{xY_2}$	1 1 1 0 1
$S'^{(3)}$	1 1 1 1 0 1 1 1 / -1 **0**
$C'^{(3)}$	0 0 0 0 0
$\mathbf{xY_3}$	0 0 0 0 0
$S'^{(4)}$	1 1 1 1 1 0 1 1 1 / **-1**
$C'^{(4)}$	0 0 0 0 0
$\mathbf{xY_4}$	0 0 0 1 1
$S'^{(5)}$	1 1 1 1 0 0 0 1 1 1
$C'^{(5)}$	0 0 0 1 1
E	0 0 0 0 1 0 0 1 1 1 = 39

(mediante un sumador común)

¿Cuántas veces más rápido es este multiplicador si lo comparamos con el anterior? Como primera aproximación, y simplificando (por ejemplo, sin considerar las cargas de los registros y los desplazamientos), en este multiplicador se realizan n sumas CSA y una suma en serie al final; por tanto, $t = n\ 2\Delta + 2n\ \Delta \approx 4n\ \Delta$. Si se realizaran todas las sumas mediante un sumador serie $t = n\ 2n\ \Delta \approx 2n^2\Delta$. Es decir, que este multiplicador en n/2 veces más rápido que el anterior (de ese orden).

4.7.2. Reducción del número de ciclos

La velocidad de cálculo de la multiplicación también puede aumentar si en cada ciclo se analiza más de un bit, es decir, si en lugar de trabajar en base 2 se trabaja en una base más alta. Por ejemplo, si tratamos los bits del multiplicador de dos en dos, reduciremos a la mitad el número de "sumas/desplazamientos". Sin embargo, si se tratan a la vez varios bits, después de la operación será necesario hacer más de un desplazamiento.

4.7.2.1 Recodificación en una base más alta

La recodificación del multiplicador en una base más alta será más o menos sencilla dependiendo de la base elegida. Por ejemplo, si r=4 es la nueva base, es suficiente analizar 2 bits del multiplicador a la vez: $011110_2 = 132_4$. De este modo, el número de dígitos a multiplicar es menor (si r=4, la mitad), pero tenemos otro problema: en la multiplicaciones parciales hay que sumar más múltiplos distintos del multiplicando, ya que en la nueva base hay más dígitos. Si la base es 4, analizando dos bits tenemos los siguientes casos:

multiplicador		
Y(1)	**Y(0)**	**Sumas**
0	0	0
0	1	x
1	0	2x
1	1	2x+x = 3x

Para generar los múltiplos se necesita tiempo y hardware y en algunos casos no es sencillo, como por ejemplo en el caso de 3x, que para conseguirlo hay que hacer un desplazamiento y una suma (2x + x). Por otro lado, trabajar con números negativos en esta base es más complicado (es fácil comprobarlo con un ejemplo). Si utilizáramos otra base, los múltiplos serían otros; por ejemplo, en base 8 los problemas surgen con los múltiplos 3, 5, 6 y 7. Así pues, aunque el número de iteraciones es menor, éstas pueden ser más complicadas.

4.7.2.2 Algoritmo de Booth en una base más alta

El algoritmo de Booth también puede utilizarse en una base más alta. En base 4, el conjunto de dígitos utilizado para la recodificación es $D_i = \{-2, -1, 0, 1, 2\}$. Por un lado, es fácil conseguir los múltiplos correspondientes a esos dígitos, y, por otro, tal y como hemos visto al principio, se tratarán igual todos los multiplicadores (no es más complicado un multiplicador negativo).

Ahora, para obtener un nuevo dígito se tienen en cuenta 3 bits de la codificación original, tal y como refleja la siguiente tabla:

X_{2i+1}	X_{2i}	X_{2i-1}	**Nuevo dígito D_i**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	-2
1	0	1	-1
1	1	0	-1
1	1	1	0

Veamos en un ejemplo cómo utilizar esta nueva recodificación para hacer multiplicaciones:

A = 0 1 1 1 (7)				A	= 0111
B = 1 0 1 0 (-6)	→	1 0 1 0 (0)	en base 2	- A	= 1001
	→	**-1 -2**	**Booth base 4**	-2A	= 10010

		0 0 0 0 0	
(-2)		1 0 0 1 0	
		1 0 0 1 0	
→→ (desplazar 2 bits)		1 1 1 0 0	1 0
(-1)		1 1 0 0 1	
		1 0 1 0 1	
→→ (desplazar 2 bits)		1 1 1 0 1	0 1 1 0 = -42

4.8. Multiplicadores combinacionales

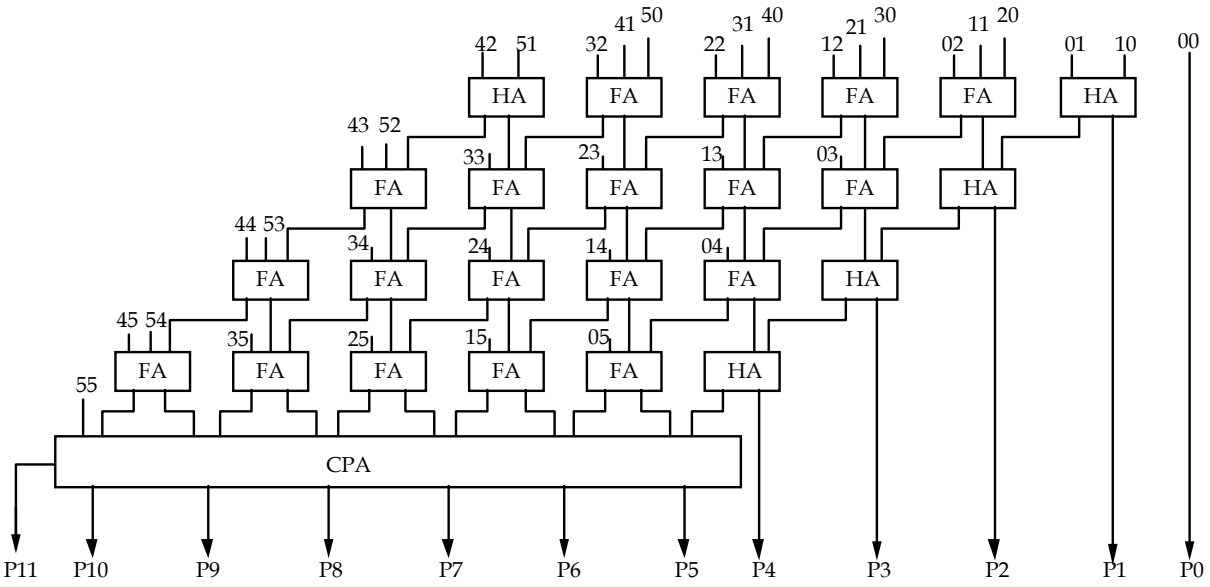
Los algoritmos anteriores de multiplicación eran secuenciales (iterativos) ya que las multiplicaciones parciales se hacen una a una. Otra posibilidad es conseguir el producto mediante un circuito puramente combinacional, construyendo una matriz de sumadores. Para simplificar, vamos a ver únicamente el caso de r=2 y números naturales.

Los principios de esta implementación son los siguientes:

1) Generar en paralelo los n^2 bits del producto $X_i Y_j 2^{i+j}$
2) Formar un array de n filas y 2n-1 columnas
3) Sumar todos los elementos de ese array

$$
\begin{array}{cccc}
 & & X_3 X_2 X_1 X_0 & \\
 & & Y_3 Y_2 Y_1 Y_0 & \\
\hline
 & & X_3Y_0 \; X_2Y_0 \; X_1Y_0 \; X_0Y_0 & \\
 & X_3Y_1 \; X_2Y_1 \; X_1Y_1 \; X_0Y_1 & & \\
X_3Y_2 \; X_2Y_2 \; X_1Y_2 \; X_0Y_2 & & & \\
X_3Y_3 \; X_2Y_3 \; X_1Y_3 \; X_0Y_3 & & & \\
\hline
\end{array}
$$

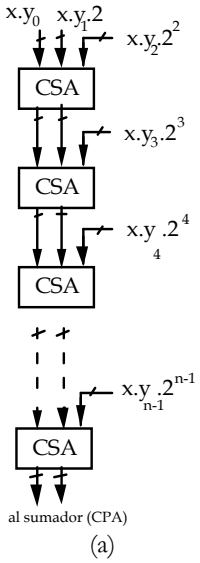
En la siguiente figura se ilustra la multiplicación de números naturales de 6 bits utilizando Full-Adders. En la figura el elemento del array X_i Y_j se denota por ij; las entradas a los sumadores de la columna k son: los elementos del array con i+j=k y las llevadas producidas por los sumadores de la columna k-1.



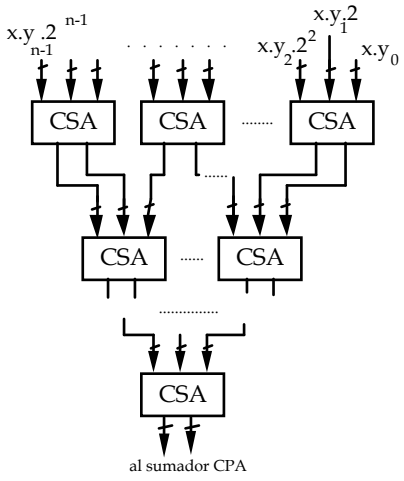
El tiempo para obtener el producto es del orden de

$$T = T_{and} + (n-2)\, T_{fa} + T_{rca} = \Delta + (n-2)2\Delta + 2n\Delta = (4n - 3)\Delta$$

Para sumar todos los componentes de la matriz se pueden utilizar circuitos CSA. En las figuras siguientes se presentan dos posibilidades para hacer las sumas: **(a)** en serie (*multiplicador carry-save iterativo)* eta **(b)** en paralelo, mediante un árbol de Wallace (*multiplicador carry-save en árbol*).



(a)



(b)

El problema de estos multiplicadores es el costo hardware que requieren. Se gana en tiempo de cálculo a costa de una gran cantidad de hardware.

5. DIVISIÓN DE NATURALES

Sean y dividendo y x divisor dos números naturales. La división, y/x, consiste en encontrar un número z, tal que

$$y = z * x + h$$

donde h es el resto de la división (Y, X, Z y H sus correspondientes vectores de dígitos). Vamos a suponer las siguientes condiciones: $0 < x \leq y < r^n x$. Así, nunca ocurrirá que $z = 0$, ni dividir por 0 ni se producirá desbordamiento.

Como hemos visto, la multiplicación puede llevarse a cabo mediante sucesivas operaciones de suma y desplazamiento. De la misma forma, la división se efectúa mediante múltiples restas y desplazamientos. Sin embargo, existe una diferencia entre ambas operaciones y es que los dígitos del cociente no se conocen inicialmente. En cada paso de división hay que ir probando para determinar un dígito del cociente, mientras que en la multiplicación los dígitos del multiplicador se conocen a priori, de ahí que la división sea más difícil que la multiplicación.

Para obtener el cociente puede utilizarse el siguiente algoritmo:

$$h^{(0)} = y$$
$$h^{(j+1)} = r\, h^{(j)} - r^n \times Z_{n-1-j} \qquad j = 0, \ldots n-1$$

Desarrollando esta recurrencia obtenemos:

$$h^{(1)} = r\, y - r^n \times Z_{n-1}$$
$$h^{(2)} = r(r\, y - r^n \times Z_{n-1}) - r^n \times Z_{n-2} = r^2 (y - x (r^{n-1} Z_{n-1} + r^{n-2} Z_{n-2}))$$
$$\ldots\ldots$$
$$h^{(n)} = r^n (y - x \sum(r^k Z_k)) = r^n (y - x\, z)$$

es decir, $\Rightarrow \quad y = x\, z + h^{(n)}\, r^{-n}$

En cada paso del algoritmo se debe elegir un dígito del cociente, Z_i, manteniendo el resto en el margen $0 \leq h < x$. Aunque hay muchas elecciones posibles, en un sistema de representación canónica la elección es única: *elegir para Z_{n-1-j} el mayor número que da resto $h^{(j+1)}$ positivo.*

Por tanto, para utilizar el algoritmo se necesita un desplazamiento ($r\, h^{(j)}$), una comparación y, finalmente, una resta, para obtener así el menor resto positivo. La comparación se realiza habitualmente mediante una resta.

5.1. División con restauración y sin restauración

El algoritmo de división anterior es muy adecuado si r = 2, ya que los dígitos de Z sólo pueden ser 1 o 0. Los pasos a seguir son los siguientes:

0 $h^{(0)} := y$

en cada iteración

1 Cálculo del resto parcial (´ = hipótesis)

$$h'^{(i+1)} = 2\,h^{(i)} - 2^n x$$

2 Elección del dígito del cociente en función del resto parcial

si $h'^{(i+1)} \geq 0$ *entonces* $Z_{n-1-i} = 1;$ $h^{(i+1)} = h'^{(i+1)}$

si no (<0) $Z_{n-1-i} = 0;$ $h^{(i+1)} = h'^{(i+1)} + 2^n x = 2\,h^{(i)}$

hasta finalizar con todos los bits

Es decir, cuando el resto parcial es negativo, el nuevo resto parcial se obtiene sumando $2^n x$ (o lo que es equivalente, se mantiene el anterior desplazado). A este método, que mantiene el resto parcial siempre positivo se le denomina **división con restauración**.

Pero no es la única opción, ya que también se puede trabajar con restos negativos. Veamos cómo. En cada paso del algoritmo se obtiene un resto:

si es positivo --> $h^{(i+1)} = 2\,h^{(i)} - 2^n x$

si no --> $h^{(i+1)} = 2\,h^{(i)}$

Esta distinción no es necesaria, ya que continuando con la segunda opción el próximo resto será:

$h^{(i+1)} = 2\,h^{(i)}$ (si el anterior es negativo)

$h^{(i+2)} = 2\,h^{(i+1)} - 2^n x = 4\,h^{(i)} - 2^n x$

Pero se puede obtener el mismo resultado continuando con la primera opción, si para calcular el resto en el siguiente paso, en lugar de una resta ($- 2^n x$),se hace una suma ($+ 2^n x$):

$h^{(i+1)} = 2\,h^{(i)} - 2^n x$ (aunque sea negativo)

$h^{(i+2)} = 2\,h^{(i+1)} \mathbf{+ 2^n x} = 2\,(2\,h^{(i)} - 2^n x) + 2^n x = 4\,h^{(i)} - 2^n x$

Por tanto, el "nuevo" algoritmo es casi igual: si el nuevo resto es positivo, para calcular el del siguiente paso se realiza una resta; si por el contrario es negativo, en el siguiente paso se hace una suma. Sólo se presenta un problema al final: si el último resto obtenido es negativo, entonces sí es necesario restaurarlo para obtener un valor positivo. A este segundo algoritmo se le denomina **división sin restauración**.

Dicho algoritmo se resume a continuación:

$h^{(0)} := y;$
$h^{(1)} := 2h^{(0)} - 2^n x;$
para j de 1 hasta n-1 hacer
comienzo
si $h \geq 0$ *entonces* $Z_{n-j} = 1;$ $h := 2h - 2^n x$
sino $Z_{n-j} = 0;$ $h := 2h + 2^n x$
fin
si $h^{(n)} < 0$ *entonces* $Z_0 := 0;$ $h^{(n)} := h^{(n)} + 2^n x$ (* para lograr un resto positivo *)
si no $Z_0 := 1$

Ejemplo

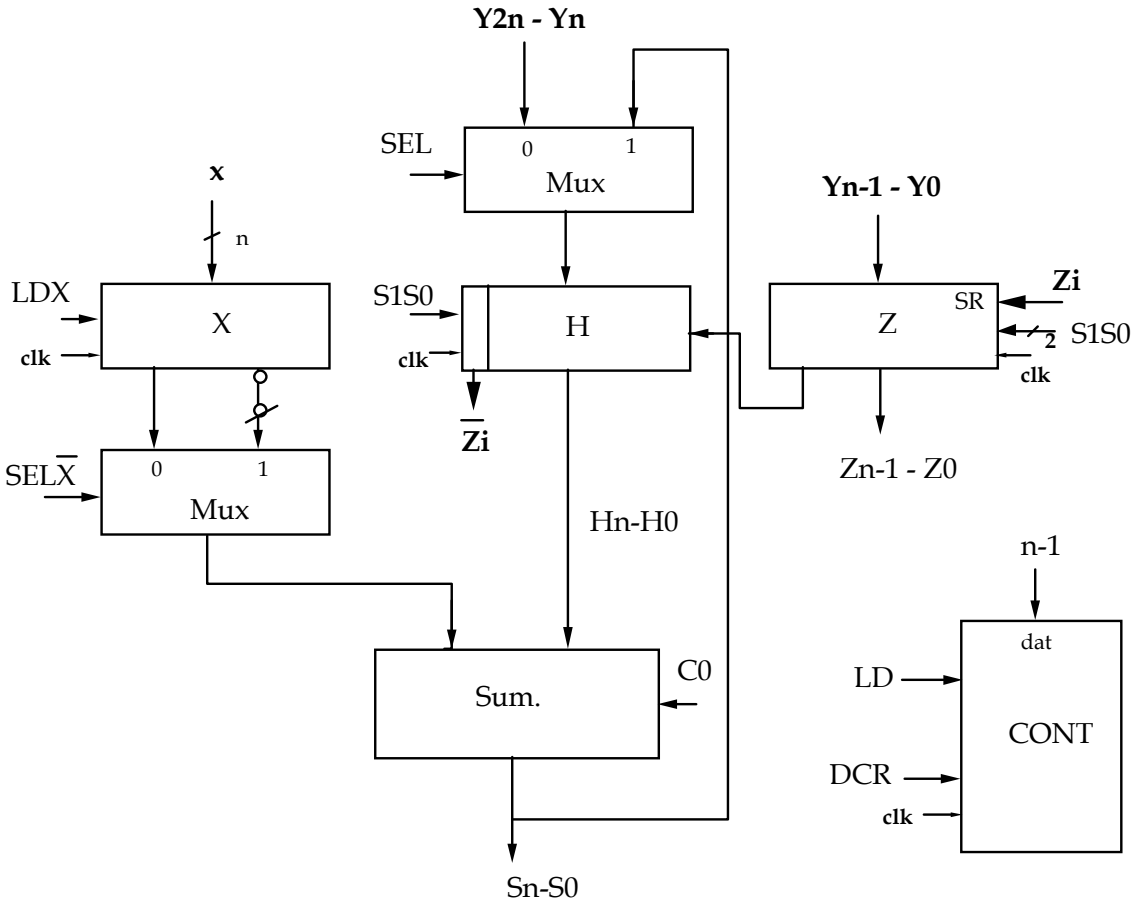
n = 4 r = 2 y = 11 Y = 1011
x = 2 X = 0010

$h^{(0)}$	0 0 0 0 0	1 0 1 1	
$2h^{(0)}$	0 0 0 0 1	0 1 1 0	
$-2^n x$	1 1 1 1 0		
$h^{(1)}$	1 1 1 1 1	0 1 1 0	$Z_3 = 0$
$2h^{(1)}$	1 1 1 1 0	1 1 0 0	
$+2^n x$	0 0 0 1 0		
$h^{(2)}$	0 0 0 0 0	1 1 0 0	$Z_2 = 1$
$2h^{(2)}$	0 0 0 0 1	1 0 0 0	
$-2^n x$	1 1 1 1 0		
$h^{(3)}$	1 1 1 1 1	1 0 0 0	$Z_1 = 0$
$2h^{(3)}$	1 1 1 1 1	0 0 0 0	
$+2^n x$	0 0 0 1 0		
$h^{(4)}$	0 0 0 0 1	0 0 0 0	$Z_0 = 1$

Luego, **Z = 0 1 0 1** **H = 0 0 0 1**
z = 5 h = 1

5.2. Hardware para la división sin restauración

La siguiente figura presenta el hardware que implementa el algoritmo visto (una posible implementación).



ALGORITMO

1: X := x; H-Z := y; CONT := n-1:
2: D_IZ (H-Z);
3: H := ADD (H, not X, 1);
4: CONT:= CONT - 1; D_IZ (H-Z); Z_0 := not H_n ;
5: if $H_n = 0$ then H := ADD (H, not X, 1)
else H := ADD (H, X, 0);
if CONT≠ 0 then goto 4;
6: D_IZ (Z); Z_0 := not H_n ; if $H_n = 1$ then H := ADD (H, X, 0);

6. ARITMÉTICA DE COMA FLOTANTE

La notación en coma fija es útil para manejar números no muy grandes con un orden de magnitud acotado, pero para ampliar este rango hay que recurrir a la representación exponencial denominada representación en coma flotante.

6.1. Representación en coma flotante

Un número en coma flotante se representa mediante la mantisa y el exponente. La base del exponente siempre ha de ser igual a la base de representación de la mantisa. El número se identifica del siguiente modo:

$$x = m * r^e = (m, e)$$

donde m= mantisa, e = exponente y r = raíz o base

Un mismo número puede tener distintas representaciones debido a que el punto decimal que separa la parte entera de la parte fraccionaria de la mantisa se puede colocar en distinto lugar si el exponente se ajusta convenientemente. Cuando la mantisa se desplaza k lugares a la izquierda (derecha) el exponente debe decrementarse (incrementarse) en k. Por ejemplo:

$$1110{,}110100\ 2^{0111} = 111{,}0110100\ 2^{1000} = 11{,}10110100\ 2^{1001} = 11101{,}10100\ 2^{0110}$$

Como la representación no es única hay que definir lo que se denomina una mantisa "normalizada". Aunque no hay una única definición de mantisa normalizada, prácticamente todas las máquinas actuales utilizan el estándar definido por IEEE. Ése es el que veremos.

6.2. Formato estándar (IEEE)

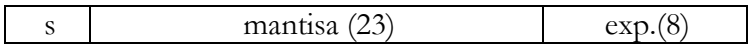
Vamos a hacer un resumen del formato IEEE que es el estándar actual.

La mantisa se representa en signo-magnitud: un bit para el signo y *p* bits para la magnitud. La mantisa **normalizada** de un número en este formato tiene un valor entre 1 y 2, es decir, $1 \leq |m| < 2$. Por tanto, su representación tendrá la forma 1,xxxx. Al ser el bit de más peso de la mantisa normalizada siempre 1, el de la parte entera, no es necesario guardarlo, de modo que se guardan sólo los bits de la parte fraccionaria y así se consigue aumentar la precisión de la representación. La idea es simple: si sólo pueden guardarse p bits, y uno de ellos es conocido porque es fijo, ése no se guarda, y así se consigue tener p+1 bits para trabajar. Al bit de la parte entera se le denomina *bit oculto* y a pesar de que no se guarde, siempre hay que utilizarlo en la operaciones.

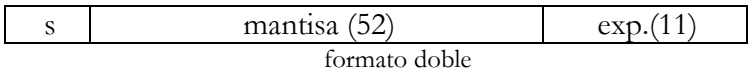
El exponente se representa mediante la representación por exceso: para representar cualquier exponente, positivo o negativo, se suma el exceso N. En el formato IEEE, se utiliza como valor de exceso $N = 2^{q-1} - 1$, donde *q* es el número de bits que se utilizan para el exponente.

6.2.1. Márgenes de la representación

En cualquier formato de coma flotante, sólo se puede representar un subconjunto de los números reales. En este estándar se definen dos formatos: el formato sencillo y el formato doble. El formato sencillo es de 32 bits: 1 bit para el signo, 23 para la mantisa y 8 para el exponente, representado en exceso 127. En el formato doble se tienen 64 bits: 52 para la mantisa, 11 bits para el exponente y 1 bit para el signo.



formato sencillo



formato doble

En el estándar se definen una serie de códigos especiales que son los que vamos a enumerar a continuación considerando el formato simple.

- un **número normalizado** se representa mediante un código con un exponente entre 0 y 255, es decir, $0 < e < 255$ (valor explícito). Por tanto, el menor exponente normalizado será 1-127=-126, y el mayor 254-127=127.

- El **número 0** se representa mediante un código especial que tiene mantisa 0 y exponente también 0. El signo puede variar: +0 o -0.

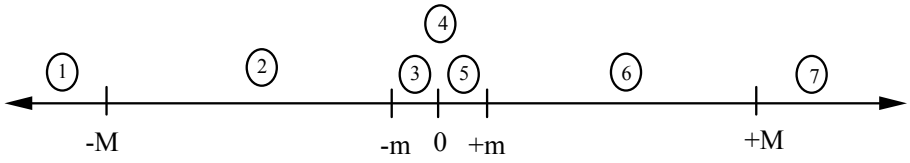
- Cualquier otro código que tenga como exponente el 0 (valor explícito) representa un **número desnormalizado**, es decir, un número demasiado pequeño para ser representado de modo normalizado, pero que es posible representar si se considera la parte entera 0. Por lo tanto la representación con $e = 0$ y $m \neq 0$, se interpreta como:

  $$(-1)^s \; 2^{-126} \; (\mathbf{0},m)$$

- El exponente 255 (valor explícito) y la mantisa 0 se utilizan para **infinito**, $+\infty$ y $-\infty$, según el signo. Por tanto, $e = 255$ y $m = 0$: $(-1)^s \infty$.
- El exponente 255 (valor explícito) y la mantisa distinta de 0, $e = 255$ y $m \neq 0$, se utilizan para representar los **caracteres NAN** (Not A Number). Estos son códigos especiales y no representan un número. Se utilizan para proporcionar información sobre lo ocurrido en una operación concreta. Entre otras cosas, se utilizan para indicar las excepciones:

operaciones no legales (por ejemplo $0 * \infty$, raíz cuadrada de un número negativo, ...), división por cero, *overflow* y *underflow*. Si se produce alguna de estas excepciones es posible ejecutar una rutina *trap* si el usuario quiere.

Por tanto, al dibujar la recta de los reales se pueden distinguir las siguientes zonas:



donde los valores de los límites son:

$|M|$: $(1+2^{-1}+2^{-2}+...+2^{-23}) * 2^{254-127} = (2 - 2^{-23}) * 2^{127}$

	1111 1111 1111 1111 1111 111	1111 1110

$|m|$: $1 * 2^{1-127} = 1 * 2^{-126}$

	0000 0000 0000 0000 0000 000	0000 0001

Tal y como se ha comentado antes:

- Los intervalos 2 y 6 corresponden a números normalizados
- El intervalo 4 corresponde al 0
- Los intervalos 3 y 5 corresponden a números demasiado pequeños para representarlos de forma normalizada. Algunos de ellos se pueden representar de forma desnormalizada, pero el resto implican underflow.
- Los intervalos 1 y 7 corresponden a números no representables en el formato por ser demasiado grandes, y por tanto, implican overflow.

6.3. Operaciones en coma flotante

La aritmética de coma flotante se descompone en dos aritméticas de coma fija, aritmética para fracciones cuando se trabaja con las mantisas y aritmética para enteros cuando se trabaja con los exponentes. A continuación utilizaremos la mantisa normalizada ($1 \leq m < 2$). Al finalizar una operación puede ocurrir que el resultado no esté normalizado y habrá que hacer un paso denominado de *postnormalización*. Por ejemplo, en la siguiente resta

$$1{,}00\ 10^0 - 7{,}00\ 10^{-1} = 1{,}00\ 10^0 - 0{,}70\ 10^0 = 0{,}30\ 10^0$$

no obtenemos un resultado normalizado y hay que corregir. Para ello se desplaza la mantisa a la izquierda y se decrementa el exponente, obteniendo como resultado normalizado $3{,}00\ 10^{-1}$.

6.3.1. Suma/Resta

Para realizar sumas o restas es necesario alinear las mantisas para igualar los exponentes de ambos números. Para conseguir en ambos números el exponente más alto es necesario desplazar a la derecha la mantisa que tiene el menor de los exponentes tantas veces como sea necesario. Tras esto, se realiza la operación, suma o resta, entre las mantisas y, finalmente, hay que postnormalizar el resultado.

Sean los números x1 (m1, e1) y x2 (m2, e2). Para hacer la suma o la resta es necesario analizar los exponentes.

$e1 > e2 \qquad x1 \pm x2 = (m1 \pm m2 \cdot 2^{-(e1-e2)}, e1)$

$e1 < e2 \qquad x1 \pm x2 = (m1 \cdot 2^{-(e2-e1)} \pm m2, e2)$

Para normalizar el resultado, último paso, puede ocurrir:

que $2 \leq \|m\| < 4$	se desplaza la mantisa una vez a la derecha y se incrementa el exponente en 1
que $\|m\| < 1$	se desplaza la mantisa a la izquierda y se decrementa el exponente tantas veces como sea necesario

Para realizar la operación entre las mantisas hay que trabajar en signo-magnitud, es decir, analizar las magnitudes y según el caso realizar la operación adecuada (en el capítulo 2 se presenta el algoritmo de suma para signo-magnitud). Es posible que el resultado necesite un dígito más. Sin embargo, al postnormalizar hay que mantener el formato, es decir, hay que utilizar el mismo número de dígitos para la mantisa, lo que implica que quizá se pierda algún dígito. Esto no se considera desbordamiento, sino fuente de error.

6.3.2. Multiplicación

Para realizar la multiplicación, hay que multiplicar las mantisas y sumar los exponentes. Por último, hay que postnormalizar el resultado.

$$y = x1 * x2 = (m1*m2, e1+e2)$$

La mantisa resultado del producto está dentro del rango $1 \leq m < 4$. Se pueden distinguir dos casos:

$1 \leq m < 2$	ya normalizada
$2 \leq m < 4$	hay que normalizar, desplazando la mantisa a la derecha una vez e incrementando el exponente en 1.

El producto entre las mantisas se puede hacer siguiente el algoritmo de multiplicación visto para signo-magnitud en el capítulo 4. De todos modos, el número de dígitos para el producto es el doble, y en el formato que estamos manejando el número de dígitos de la mantisa es fijo, lo cual hace necesario manejar el resultado para mantener dicho número de dígitos. De nuevo, el paso de postnormalización implicará la pérdida de un conjunto de dígitos significativos para mantener el número de dígitos de la mantisa, pero eso no implica desbordamiento.

6.3.3. División

Para hacer una división hay que dividir las mantisas y restar los exponentes. Después hay que normalizar el resultado.

$$y = x1 / x2 = (m1/m2, e1-e2)$$

El cociente entre las mantisas está dentro del rango $1/2 < m < 2$. Se pueden distinguir dos casos:

$1 \leq m < 2$ ya está normalizada

$1/2 < m < 1$ hay que normalizarla, desplazando una vez la mantisa a la izquierda y decrementando el exponente en 1

Para realizar la división podemos utilizar el algoritmo visto en el tema 5, olvidando la coma al hacer la división. Lo que ocurre es que en este caso no nos interesa obtener sólo la parte entera, que será 0 o 1, sino continuar con el algoritmo para obtener los dígitos de la parte fraccionaria.

6.4. Fuentes de error

En la aritmética de coma flotante hay dos fuentes de error posibles. Por un lado, el error debido a la representación de los números reales, que no es exacta, y por otro, la pérdida de dígitos significativos.

Dado que el número de dígitos es limitado en cualquier sistema de representación, a la hora de representar números reales se puede dar un error de precisión debido a que se necesitaría un número de dígitos infinito. Ocurre pues, que el error de representación se genera al sustituir los números reales por números máquina.

Sea M el conjunto de números máquina que se pueden representar con total precisión. Para representar un número real n mediante uno de los números máquina de M existen distintos criterios:

- *Inmediato inferior* (INF): número inferior más cercano
  (*round toward* $-\infty$)
  $$\mathbf{INF}\ (n) = \max \{ x \quad M / x \leq n\}$$
- *Inmediato superior* (SUP): número superior más cercano
  (*round toward* $+\infty$)
  $$\mathbf{SUP}\ (n) = \min \{ x \quad M / x \geq n\}$$
- *Truncamiento:* número más cercano que no es superior en valor absoluto
  (*round toward 0*)
  $$\mathbf{INF}\ (n) \quad \text{si } n \geq 0$$
  $$\mathbf{SUP}\ (n) \quad \text{si } n < 0$$
  por tanto, acerca los números a 0
- *Redondeo:* el número más cercano
  (*round to nearest*)
  $$\mathbf{INF}\ (n) \quad \text{si } INF(n) \leq n < (INF(n) + SUP(n))\ /2$$
  $$\mathbf{SUP}\ (n) \quad \text{si } (INF(n) + SUP(n))\ /2 \ \leq n \ < SUP(n)$$

Vamos a ver un ejemplo. Sea el siguiente formato binario normalizado: 2 bits de mantisa y 3 bits de exponente, ¿qué número máquina asignaremos al número 8,48 = 1000,01... con la mantisa normalizada según el estándar IEEE?

Dado que la representación del número debe ser normalizada hay que desplazar la coma 3 posiciones hacia la izquierda, e incrementar el exponente en 3. Dado que la representación del exponente es en *exceso 3*, el valor explícito a representar es el 6: 110.

$$8{,}48 \ \Rightarrow 1,\ 0\,0\,0\,0\,1\,1 \ldots * 2^3$$

La representación normalizada del 8,48 no será exacta y hay que tomar una decisión en cuanto al número que se va a utilizar para representarlo. A la vista del código anterior la opción está entre: (1), 0 0 y (1), 0 1

Dado que tenemos distintas aproximaciones:

		error
INF	= (1),00	8 - 8,48 = -0,48
SUP	= (1),01	10 - 8,48 = 1,52
Truncam.	= (1),00	8 - 8,48 = -0,48
Redondeo	= (1),00	8 - 8,48 = -0,48

El error de representación está relacionado con la precisión de la representación y la magnitud del número. Al representar un número concreto, el error cometido está acotado, en valor absoluto, por la siguiente expresión:

$$E = \Delta(m)\ r^{e}$$

donde $\Delta(m)$ representa la menor variación posible de la mantisa, es decir, la que corresponde al bit de menos peso, y e es el exponente del número. En el caso anterior, el límite superior del error es $2\ (1/4 * 2^3)$.

Como ya se ha dicho antes, otra fuente de error es la propia operación, que puede llevar consigo la pérdida de dígitos significativos, en ocasiones porque no se calculan, y en otras porque a pesar de haberse calculado no se pueden incluir debido al número limitado de dígitos del formato.