ARITMÉTICA ENTERA

AMPLIACIÓN DE ESTRUCTURA DE COMPUTADORES

Daniel Mozos, José Luis Risco, Daniel Chaver
Facultad de Informática

Aritmética

1.- Representación de reales
2.- Suma y resta de enteros
3.- Multiplicación de enteros
    - Multiplicación de enteros sin signo
    - Multiplicación de enteros con signo
        - Algoritmo de Booth
        - Multiplicadores recodificados
        - Salva-arrastre
    - Multiplicación combinacional
        - Multiplicador de Pezaris
        - Multiplicador de Baugh-Wooley
        - Multiplicadores recodificados
4.- División de enteros
    - División de enteros sin signo
    - División por convergencia
    - División combinacional
5.- Cálculos trigonométricos: método Cordic

Bibliografía

1.-"Computer arithmetic algorithms". I. Koren, Prentice Hall, 2002

2.- “Computer Arithmetic: Algoritms and Hardware Design”, B.Parhami. Oxford UP. 2000

3.-"Computer architecture. A quantitative approach". Hennessy & Patterson, Morgan Kaufmann, 1995. Apéndice A.

4.-"Computer arithmetic". K. Hwang, John Wiley & Sons, 1979.

5.- “Digital computer arithmetic”, J. Cavanagh, McGraw Hill, 1985

Representación de reales

- **Representación en coma fija**
  - Número fijo de bits para la parte entera y para la parte decimal. Dejando implícito que la coma decimal se coloca entre ellos.
  - La aritmética se realizará procesando la parte entera y la decimal separadamente usando instrucciones aritméticas enteras.
  - Rango de representación bastante limitado.
- **Representación en coma flotante**
  - Las aplicaciones científicas frecuentemente usan números muy pequeños o muy grandes y con la representación en coma fija estos números sólo tienen cabida si se utilizan muchos bits para la representación.
  - Esta notación consta de los siguientes elementos:
    - s = signo
    - F = fracción
    - E = exponente
    - R = raíz
  - El valor V se calculará como: $V = (-1)^s * F * R^E$.

Suma y resta de enteros

- Sean *x* e *y* dos números enteros, representados por los vectores de bits X e Y. El algoritmo de la suma, SUMA, produce como resultado el vector de dígitos S que representa al número entero *s*, tal que s=x+y.
  - S=SUMA(X,Y)
- Para la resta, introducimos la operación cambio de signo (CS), siendo D un vector que representa al resultado *d* de la resta, d=x-y,
  - D=SUMA(X,CS(Y)).
- Si el rango de enteros representable por S o D es el mismo que el de X o Y, el resultado de la suma o resta puede no ser representable. En ese caso el resultado del algoritmo no sería correcto y se debería dar una señal de error por rebose.

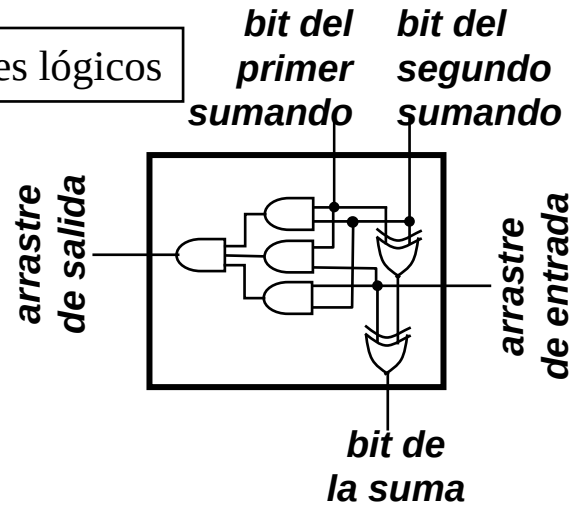
Suma de enteros

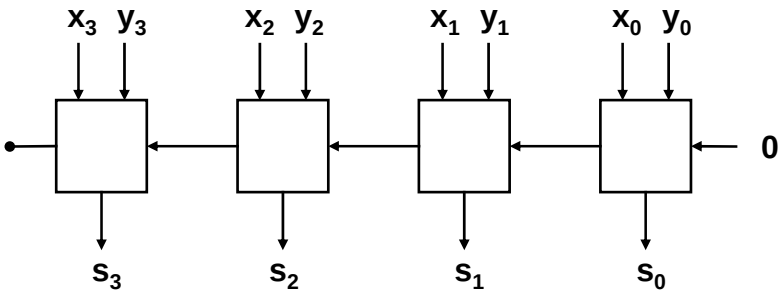
Sumador con propagación de arrastres

- Diseñar una celda combinacional que, tomando dos dígitos y un posible arrastre anterior, genere la suma y un arrastre posterior
- Replicar la celda tantas veces como dígitos tenga el número

$$s_i = (x_i \oplus y_i) \oplus c_i$$
$$c_{i+1} = x_i \cdot y_i + x_i \cdot c_i + c_i \cdot y_i$$
$$= x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i$$

2 niveles lógicos





Características:
- Lentos
- Baratos

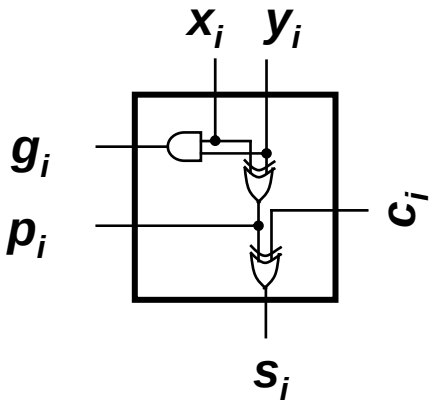
Suma de enteros

Sumador con anticipación de arrastres

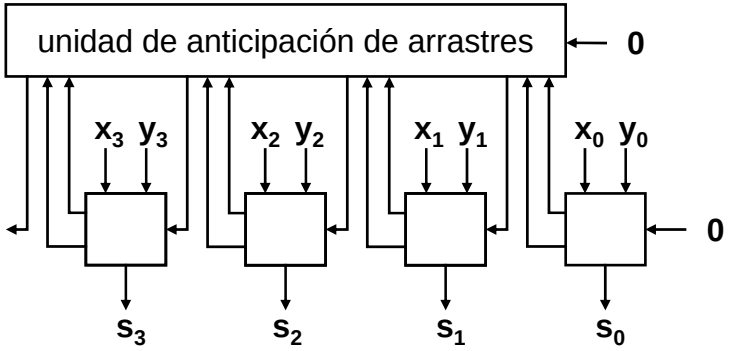
- Se busca reducir el tiempo de cálculo:

$$c_{i+1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i = g_i + p_i c_i$$

- g_i es 1 si una celda genera un arrastre, es decir $x_i=y_i=1$
- p_i es 1 si una celda propaga un arrastre, es decir $x_i \oplus y_i=1$



$$s_i = (x_i \oplus y_i) \oplus c_i = P_i \oplus c_i$$
$$c_{i+1} = x_i \cdot y_i + x_i \cdot c_i + c_i \cdot y_i = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_i = G_i + P_i \cdot c_i$$



$$c_1 = G_0 + P_0 \cdot c_0$$
$$c_2 = G_1 + P_1 \cdot c_1$$
$$= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0$$
$$c_3 = G_2 + P_2 \cdot c_2$$
$$= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0$$
$$c_4 = G_3 + P_3 \cdot c_3$$
$$= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0$$

- 4 niveles de puertas para cualquier bit de la suma
- Conforme aumenta el número de bits el número de términos producto y el número de factores en ellos crece demasiado: para 32 bits el cálculo de c_{32} tiene 33 t.p y 33 factores

Suma de enteros

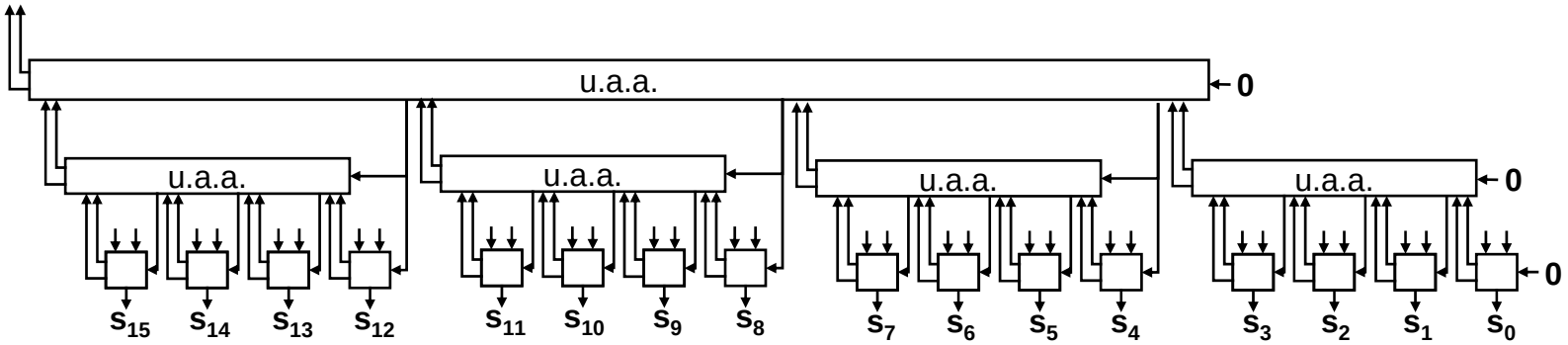
Sumador con varios niveles de anticipación de arrastres

- Utilizar la anticipación de arrastres, no solamente sobre celdas de un bit de suma, sino sobre módulos de suma con mayor número de bits.
  - Un módulo genera un arrastre si se genera en alguna de sus celdas internas y se propaga hasta la salida
  - Un módulo propaga un arrastre si el arrastre de entrada es uno y todas las celdas intermedias lo propagan

Para 4 bits, dichas señales resultan:

$$G^* = G_3 + G_2 \cdot P_3 + G_1 \cdot P_2 \cdot P_3 + G_0 \cdot P_1 \cdot P_2 \cdot P_3$$
$$P^* = P_0 \cdot P_1 \cdot P_3 \cdot P_4$$

- El arrastre de salida se calcula como: $c_{out} = G^* + P^* \cdot c_{in}$

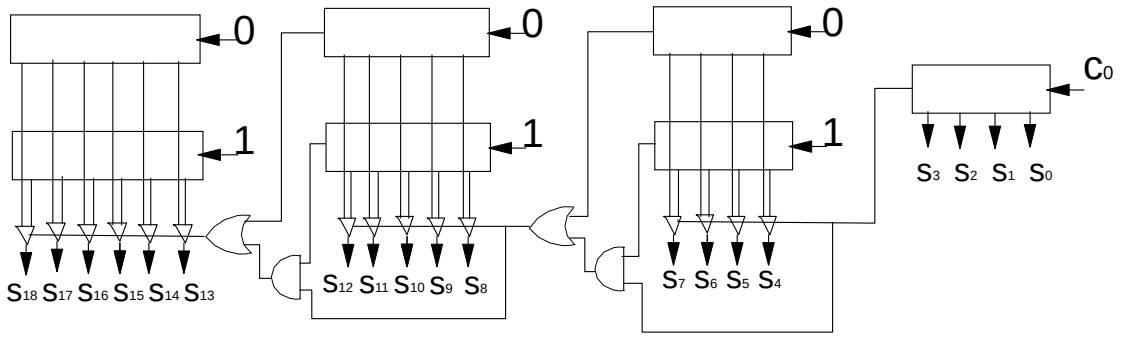


Suma de enteros

Sumador con selección de arrastres

Idea:

- Dividir el sumador en módulos que se implementen con alguno de los métodos anteriores.
- Duplicar el nº de módulos de forma que se calculen en paralelo los resultados para $c_i = 0$ y para $c_i = 1$.
- Cuando se conoce el c_i real se selecciona el valor adecuado.



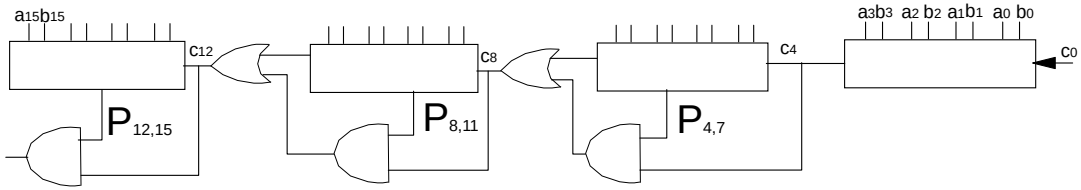
Los niveles de puertas requeridos para realizar una suma vienen dados por el camino crítico entre el arrastre de entrada y la selección del bit más significativo de la suma

Suma de enteros

Sumador con puenteo de arrastres (carry-skìp adder)

Idea:

- Aprovechar los datos de generación y propagación de arrastres sin usar un módulo de anticipación de arrastres.
  - $P_{03} = p_0p_1p_2p_3$
  - $G_{03} = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0$
    - ➢ Modificamos ligeramente los sumadores para poder calcular P.
    - ➢ La señal G es el arrastre de salida de cada módulo si el arrastre de entrada era 0
- Es una mezcla de propagación y generación.



- Los niveles de puertas requeridos para realizar una suma vienen dados por el camino crítico entre el arrastre de entrada y el cálculo del bit más significativo de la suma

Suma de enteros

Sumador carry save

Objetivo:

•Acelerar la suma cuando se tienen que sumar mas de dos operandos, tratando de evitar la propagación de arrastres.

•Al sumar dos operandos los arrastres no se propagan, sino que se usan como tercer operando en la suma siguiente. Sólo en la última suma habrá que propagar los arrastres.

•Ejemplo en decimal: S=357+409+143+112

Con propagación de arrastres

```
  357
+ 409
  766
+ 143
  909
+ 112
 1021
```

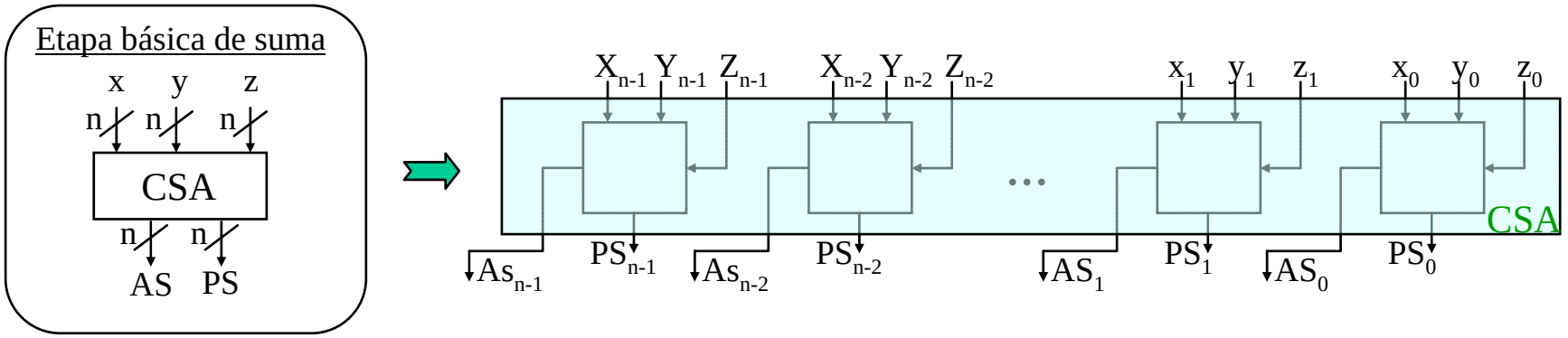
Con salva-arrastre

```
          357
          409
CSA     + 143
          899  ←  PseudoSuma (PS)
          001  ←  Arrastre Salvado (AS)
CSA     + 112
          911  (PS)
Paso Final
        + 11   (AS)
         1021
```

Suma de enteros

Sumador carry save

Construcción de un sumador salva-arrastre

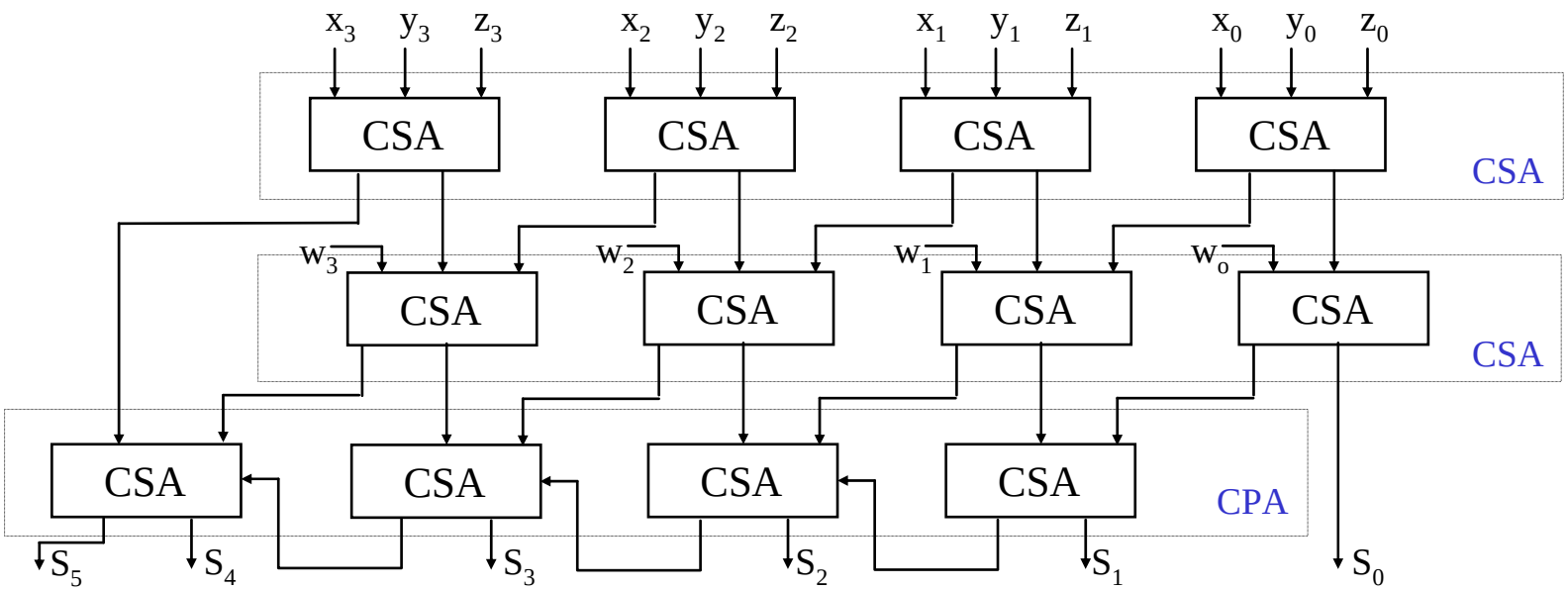


Cada una de las celdas individuales del CSA son sumadores completos, en los que el arrastre se utiliza como tercer sumando.

Suma de enteros

Sumador carry save

Ej: diseño de un sumador carry save de 4 operandos de 4 bits.



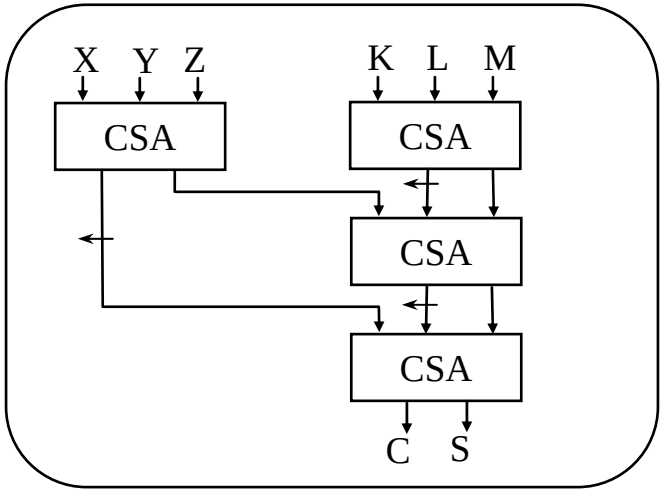
CSA: Carry Save Adder
CPA: Carry Propagate Adder

Suma de enteros

Sumador carry save: Árboles de Wallace

Es otra forma de organizar los CSA para tratar de mejorar el rendimiento.

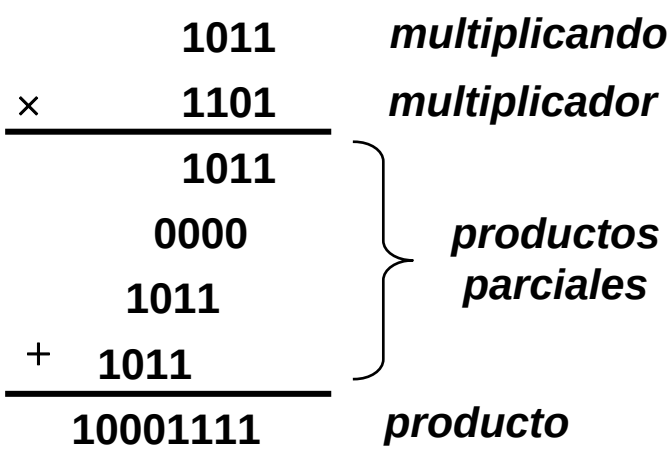
Arbol de Wallace con 6 operandos



Número de niveles en un árbol de Wallace para k operandos

Nº de operandos	Nº de niveles
3	1
4	2
5≤k≤6	3
7≤k≤9	4
10≤k≤13	5
14≤k≤19	6
20≤k≤28	7
29≤k≤42	8
43≤k≤63	9

Multiplicación de enteros sin signo



Método tradicional de multiplicación :

- Obtener los productos parciales
- Cada producto parcial debe estar desplazado una posición a la izquierda respecto al producto parcial anterior
- Una vez calculados todos los productos parciales se suman para obtener el producto

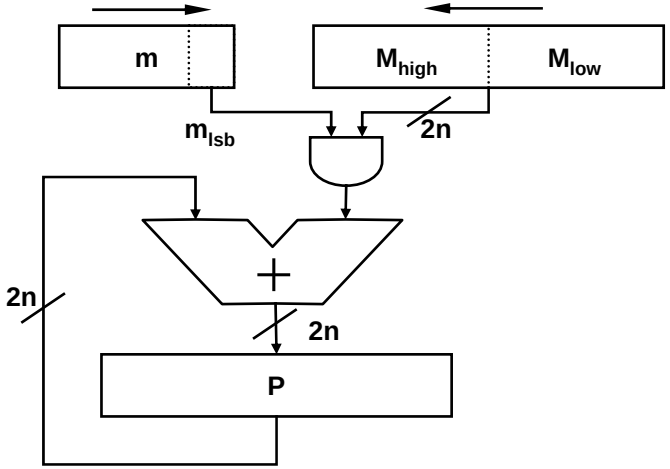
- Para un multiplicando de ***n*** bits y un multiplicador de ***m*** bits el producto ocupa como máximo ***n+m*** bits

Ruta de datos:

- 3 registros: multiplicando, multiplicador y producto
- 1 sumador de 2 entradas, en cada iteración sumar el producto parcial obtenido a la suma de los anteriores
- Para alinear correctamente los productos parciales, en cada iteración desplazar el multiplicando a la izquierda
- Para leer del mismo lugar cada uno de los bits del multiplicador, en cada iteración desplazarlo a la derecha

Multiplicación de enteros sin signo

Ruta de datos:



Algoritmo:

S0 : cargar multiplicador en *m*
cargar multiplicando en M_{low}
borrar M_{high}
borrar *P*

S1 : si m_{lsb} = 1 entonces $P \leftarrow P+M$
si m_{lsb} = 0 entonces $P \leftarrow P+0$

S2 : desplazar *M* a la izquierda
desplazar *m* a la derecha
si *S1*-*S2* no se han repetido *n* veces, ir a *S1*

tras	*m*	*M*	*P*
S0	**1101**	0000**1011**	00000000
S1	110**1**	00001011	00001011
S2	011**0**	00010110	00001011
S1	0110	00010110	00001011
S2	001**1**	00101100	00001011
S1	0011	00101100	00110111
S2	000**1**	01011000	00110111
S1	0001	01011000	10001111
S2	0000	10110000	**10001111**

Multiplicación de enteros con signo

La multiplicación en C2 **no es coherente** con la multiplicación sin signo.
Sin embargo, puede modificarse el algoritmo de suma y desplazamiento para que opere directamente en C2. Para ello distinguiremos 3 casos posibles:

1er. caso:
multiplicando positivo
multiplicador positivo

No requiere modificación

(+3)×(+5) = (+15)

```
      0011
×     0101
------------
      0011
     0000
    0011
+  0000
------------
  00001111
```

15 = 3×5 ✓

2º caso: ⇒
multiplicando negativo
multiplicador positivo

Extender correctamente el signo del dividendo

(-5)×(+5) = (-25)

```
      1011                1011
×     0101          ×     0101
------------        ------------
  00001011            11111011
  0000000             0000000
  001011              111011
+ 00000             + 00000
------------        ------------
  00110111            11100111
```

55 = 11×5 ✗

25 = (-5)×5 ✓

✓ Al sumar los productos parciales se asume implícitamente que los bits más significativos de los sumandos son 0, esto sólo es correcto si el multiplicando es positivo, si es negativo se está extendiendo incorrectamente su signo.

Multiplicación de enteros con signo

3er. caso:
multiplicador negativo

(-5)×(-3) = (+15)

	1011		1011
×	1101	×	1101
	11111011		11111011
	0000000		0000000
	111011		111011
+	11011	+	00101
	10111111		00001111
	-65 ✘		+15 ✓

En la última iteración sumar o restar el multiplicando en función del signo del multiplicador

Estudiemos la operación que se realiza cuando el **multiplicador es negativo**:

- ✓ El multiplicador está representado en C2 luego su valor es:

$$V(m) = -m_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} m_i \cdot 2^i$$

- ✓ Si se ignora el signo del multiplicador (y se realiza la multiplicación teniendo en cuenta sólo los n-1 bits menos significativos del multiplicador) el resultado del método tradicional es:

$$\begin{aligned} V(P) &= V(M) \cdot \sum_{i=0}^{n-2} m_i \cdot 2^i \\ &= V(M) \cdot \left(V(m) + m_{n-1} \cdot 2^{n-1}\right) \\ &= V(M) \cdot V(m) + V(M) \cdot m_{n-1} \cdot 2^{n-1} \end{aligned}$$

- ✓ El resultado buscado es V(M)·V(m), luego cuando m_{n-1}= 1 es necesario corregir el resultado parcial obtenido, restando el multiplicando a la mitad mas significativa de P:

$$V(M) \cdot V(m) = V(P) - V(M) \cdot m_{n-1} \cdot 2^{n-1}$$

Multiplicadores binarios recodificados: algoritmo de Booth

• Permite multiplicar directamente enteros representados en C2.

• Evita ejecutar sumas consecutivas cuando el multiplicador presenta cadenas de *0s* o de *1s*.

Idea: Convertir el **multiplicador** en un número recodificado sobre un sistema binario no canónico bajo la forma de dígitos con signo:

Sistema binario canónico **D={0,1}** => Sistema binario no canónico **D={-1,0,1}.**

Dada la cadena de bits:

Peso: ... i+k, i+k-1, i+k-2, ..., i+1, i, i-1 ...

Bits: 0 1 1 ... 1 1 0

k 1's seguidos

teniendo en cuenta la igualdad:

$$\mathbf{2^{i+k-1} + 2^{i+k-2} +...+ 2^{i}} = (2^{k-1} + 2^{k-2} +...+ 2^{0})2^{i} = (2^{k}-1)\,2^{i} = \mathbf{2^{\,i+k}-2^{i}}$$

la cadena de 1's podemos sustituirla por:

Peso: ... i+k, i+k-1, i+k-2, ..., i+1, i, i-1 ...

Bits: 1 0 0 ... 0 -1 0

(k-1) 0's seguidos

Multiplicadores binarios recodificados: algoritmo de Booth

Para el caso de números negativos, en C2, cuyo bit más significativo, i+k es un 1, podemos demostrar también que se cumple esta equivalencia:

Peso: ... i+k, i+k-1, i+k-2, ..., i+1, i, i-1 ...

Bits: 1 1 1 ... 1 1 0

k+1 1's iniciales

Esto es equivalente a:

$$-2^{i+k} + 2^{i+k-1} + 2^{i+k-2} + \ldots + 2^{i} = -2^{i+k} + (2^{k-1} + 2^{k-2} + \ldots + 2^{0})2^{i} =$$
$$= -2^{i+k} + (2^{k}-1)\,2^{i} = -2^{i+k} + 2^{i+k} - 2^{i} = -2^{i}$$

Es decir, un **–1** en la posición i.

Multiplicadores binarios recodificados: algoritmo de Booth

Recodificación del multiplicador: (Y_{n-1}, Y_{n-2}, ... , Y_0):

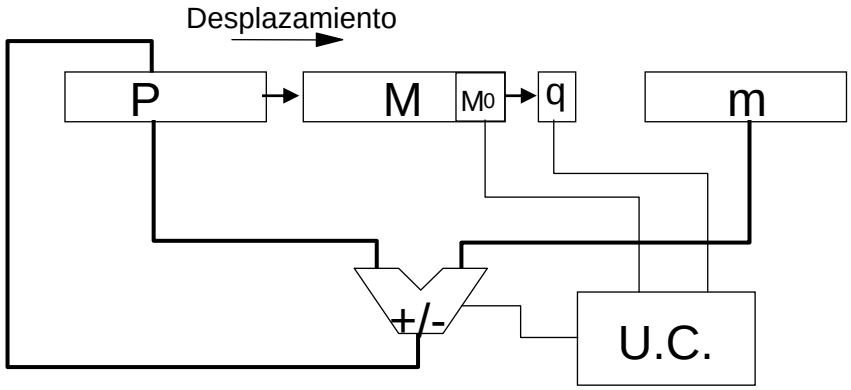
- Se analizan todos los bits del multiplicador y se sustituyen las cadenas de 1s por un (–1) en la posición del 1 menos significativo y un 1 en la posición del 0 que las precede.
- De este modo resultará un multiplicador representado con los dígitos {-1,0,1} y al realizar la multiplicación habrá que hacer **sumas y restas**.

En resumen:

Bits del multiplicador	**Dígito recodificado**	**Operación a realizar**
Y_i Y_{i-1}	Z_i	
0 0	0	0 * multiplicando
0 1	1	1 * multiplicando
1 0	-1	-1 * multiplicando
1 1	0	0 * multiplicando

Multiplicadores binarios recodificados: algoritmo de Booth

Ruta de datos para un multiplicador de Booth



Algoritmo de multiplicación

1. M ← multiplicando | M ← multiplicador | P = q ← 0.

2. Si M_oq =00 ó M_oq =11 => $DD_{arit}(P,M,q)$
Si M_oq =10 => P ← P-m | $DD_{arit}(P,M,q)$
Si M_oq =01 => P ← P+m | $DD_{arit}(P,M,q)$

El paso 2 se realiza *n* veces, siendo n el número de bits del multiplicador.
Una vez finalizado, el resultado se hallará en los registros **P** (parte más significativa) y **M** (parte menos significativa).

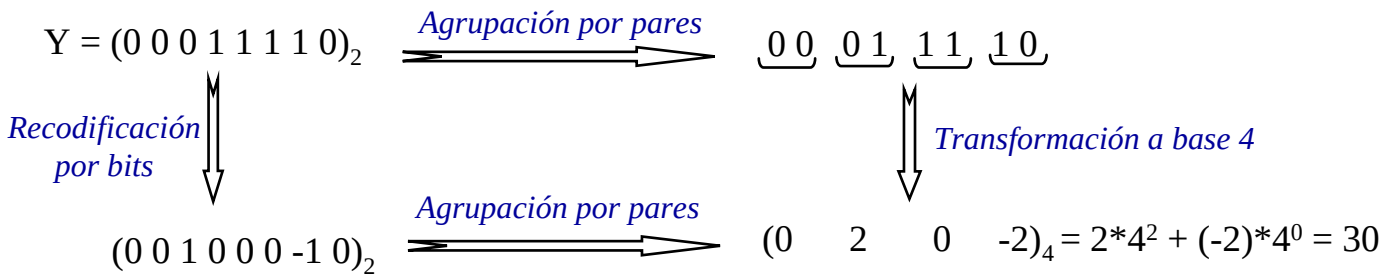
Multiplicadores recodificados: Recodificación por pares de bits

- Como método de aceleración de la multiplicación se puede proceder tratando en cada paso un grupo de 2 bits del multiplicador en vez de uno solo.

- Nos servirá para multiplicar directamente números en C2, garantizando que para un multiplicador de n bits habrá como máximo n/2 productos parciales.

- Se realiza la misma acción que en la recodificación de Booth pero considerando pares de bits, junto con el bit que está a su derecha.

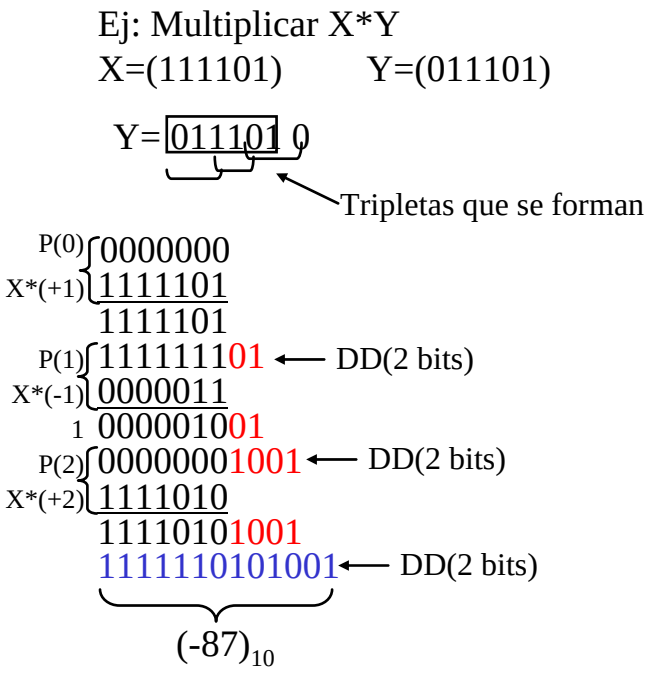
Ejemplo:



Multiplicadores recodificados: Recodificación por pares de bits

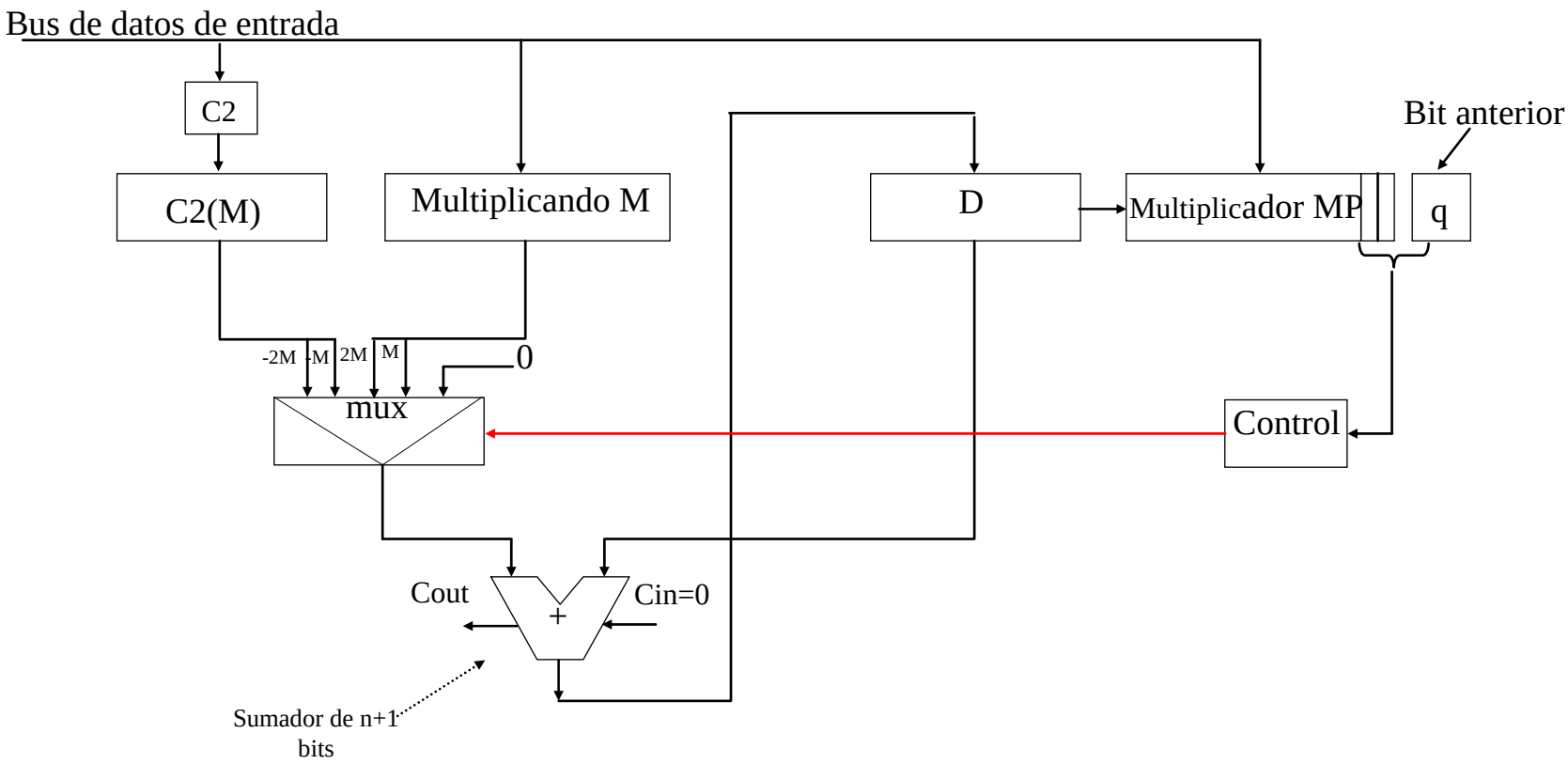
• El proceso de recodificación por pares de bits puede hacerse de forma directa, observando cada par de bits del multiplicador y el bit a su derecha.

Par de bits		**Bit anterior**	**Digito base 4 recodificado**		**Operación a realizar**
i+1	**i**	**i-1**			
0	0	0	(0 0)	0	0* multiplicando
0	0	1	(0 1)	1	+1* multiplicando
0	1	0	(1 –1)	1	+1* multiplicando
0	1	1	(1 0)	2	+2* multiplicando
1	0	0	(-1 0)	-2	-2* multiplicando
1	0	1	(-1 1)	-1	-1* multiplicando
1	1	0	(0 –1)	-1	-1* multiplicando
1	1	1	(0 0)	0	0* multiplicando



Multiplicadores recodificados: Recodificación por pares de bits

Recodificación por pares de bits: ruta de datos:



Multiplicación salva-arrastre

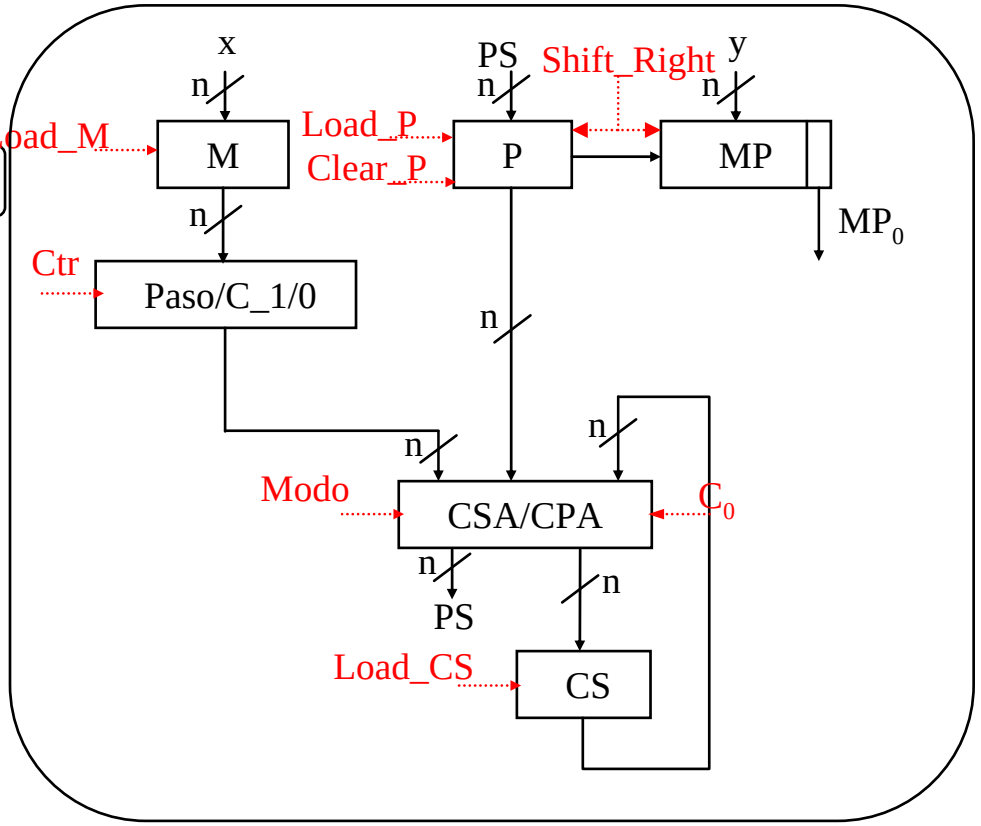
Idea: Usar sumadores salva-arrastre, dado que hay que realizar sumas con varios sumandos

Cada paso implica:

$(PS^{(j+1)}, CS^{(j+1)}) <= SR(CSA(PS^{(j)}, CS^{(j)}, X*Y_j))$

CSA/CPA= Carry save adder/Carry propagate adder

Ruta de datos de un multiplicador CS



Multiplicación salva-arrastre

P= 11101*01011

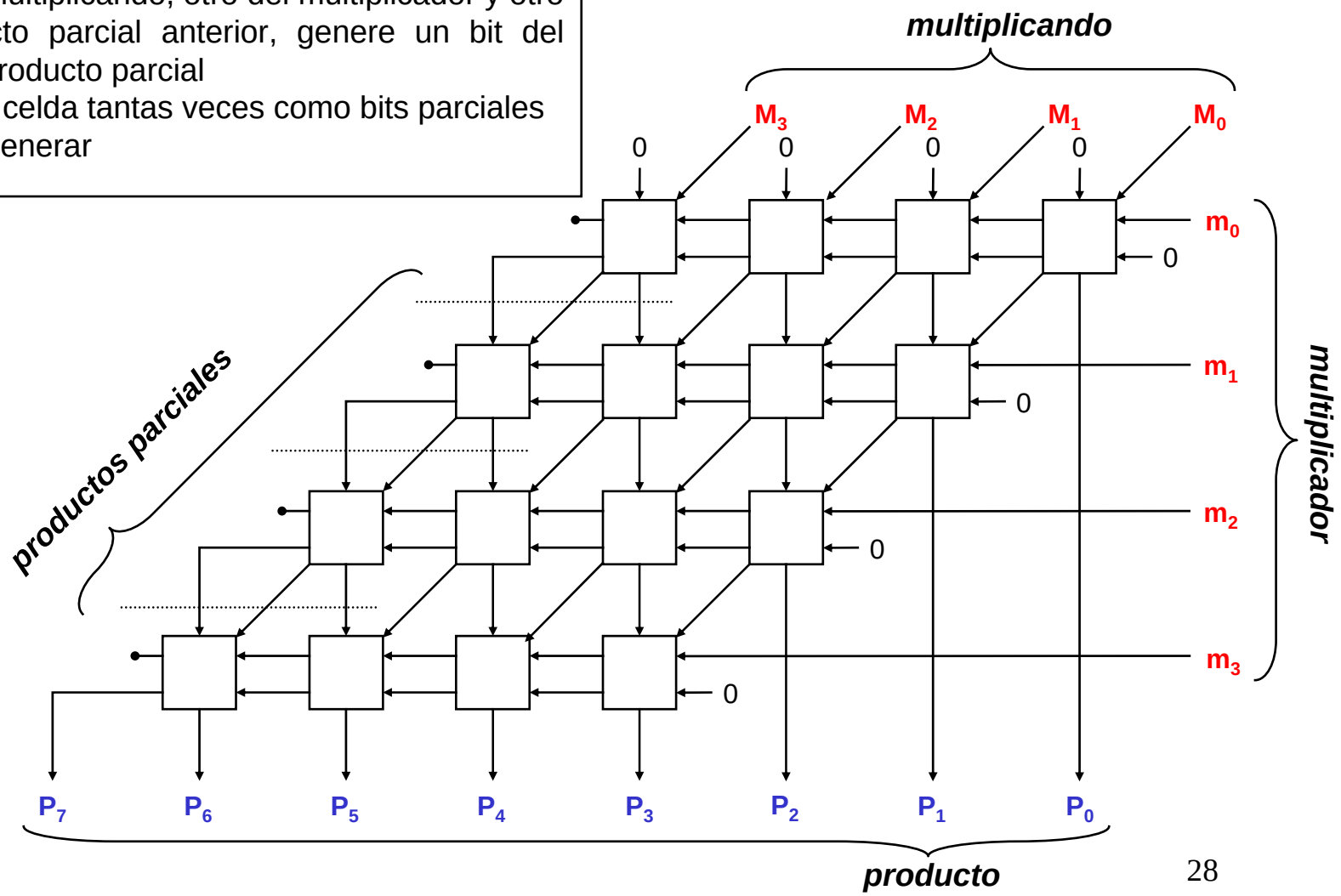
$PS^{(0)}$ 0 0 0 0 0
$CS^{(0)}$ 0 0 0 0 0
Xy_0 1 1 1 0 1
1 1 1 0 1
0 0 0 0 0 DD
$PS^{(1)}$ 1 1 1 1 0 1
$CS^{(1)}$ 0 0 0 0 0
Xy_1 1 1 1 0 1
0 0 0 1 1
1 1 1 0 0 DD
$PS^{(2)}$ 0 0 0 0 1 1 1
$CS^{(2)}$ 1 1 1 0 0
Xy_2 0 0 0 0 0
1 1 1 0 1
0 0 0 0 0 DD
$PS^{(3)}$ 1 1 1 1 0 1 1 1
$CS^{(3)}$ 0 0 0 0 0
Xy_3 1 1 1 0 1
0 0 0 1 1
1 1 1 0 0 DD
$PS^{(4)}$ 0 0 0 0 1 1 1 1 1
$CS^{(4)}$ 1 1 1 0 0
Xy_4 0 0 0 0 0
1 1 1 0 1
0 0 0 0 0 DD
$PS^{(5)}$ 1 1 1 1 0 1 1 1 1 1
CPA → $CS^{(5)}$ 0 0 0 0 0
Producto: 1 1 1 1 0 1 1 1 1 1

P= 01011*11101

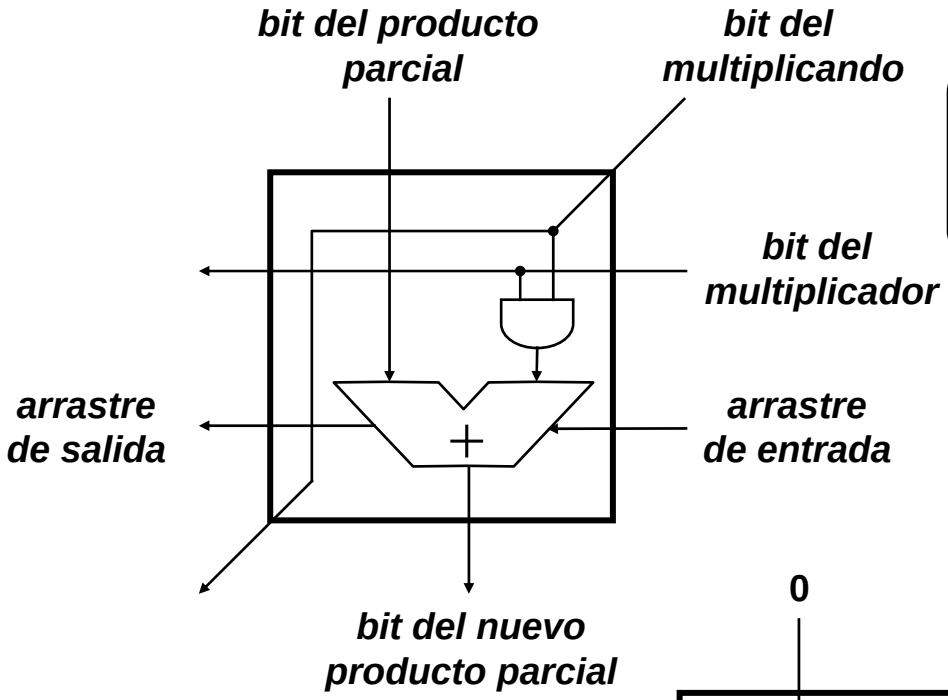
$PS^{(0)}$ 0 0 0 0 0
$CS^{(0)}$ 0 0 0 0 0
Xy_0 0 1 0 1 1
0 1 0 1 1
0 0 0 0 0 DD
$PS^{(1)}$ 0 0 1 0 1 1
$CS^{(1)}$ 0 0 0 0 0
Xy_1 0 0 0 0 0
0 0 1 0 1
0 0 0 0 0 DD
$PS^{(2)}$ 0 0 0 1 0 1 1
$CS^{(2)}$ 0 0 0 0 0
Xy_2 0 1 0 1 1
0 1 0 0 1
0 0 0 1 0 DD
$PS^{(3)}$ 0 0 1 0 0 1 1 1
$CS^{(3)}$ 0 0 0 1 0
Xy_3 0 1 0 1 1
0 1 1 0 1
0 0 0 1 0 DD
$PS^{(4)}$ 0 0 1 1 0 1 1 1 1
$CS^{(4)}$ 0 0 0 1 0
$C1(Xy_4)$ 1 0 1 0 0
1 0 0 0 0
0 0 1 1 0 DD
$PS^{(5)}$ 1 1 0 0 0 0 1 1 1 1
$CS^{(5)}$ 0 0 1 1 0 1 ← +1 para calcular el C2
Producto: 1 1 1 1 0 1 1 1 1 1

Multiplicación combinacional

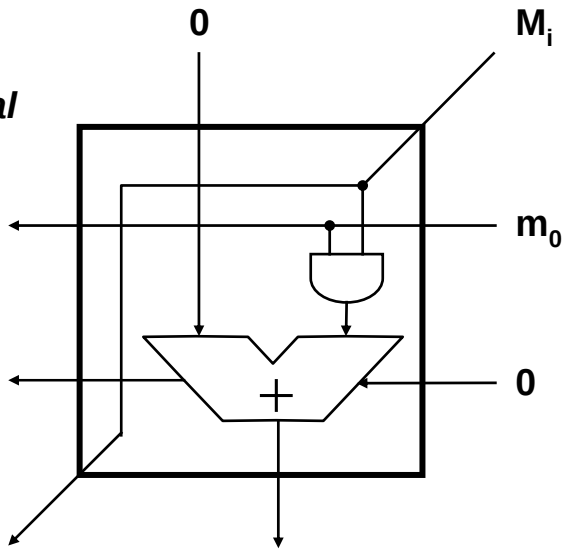
- Diseñar una celda combinacional que, tomando un dígito del multiplicando, otro del multiplicador y otro del producto parcial anterior, genere un bit del siguiente producto parcial
- Replicar la celda tantas veces como bits parciales haya que generar

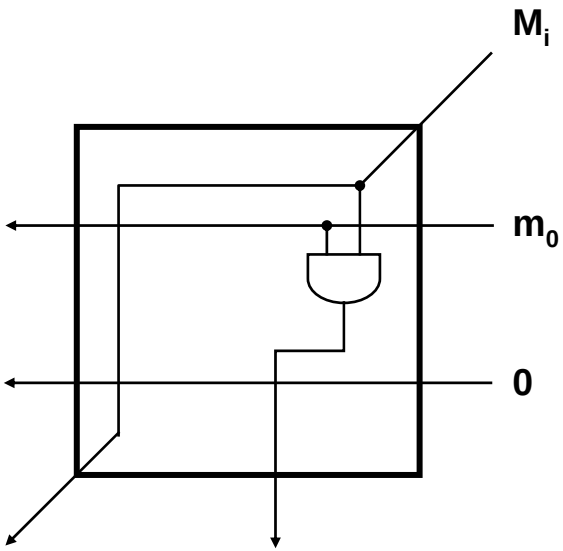


Multiplicación combinacional



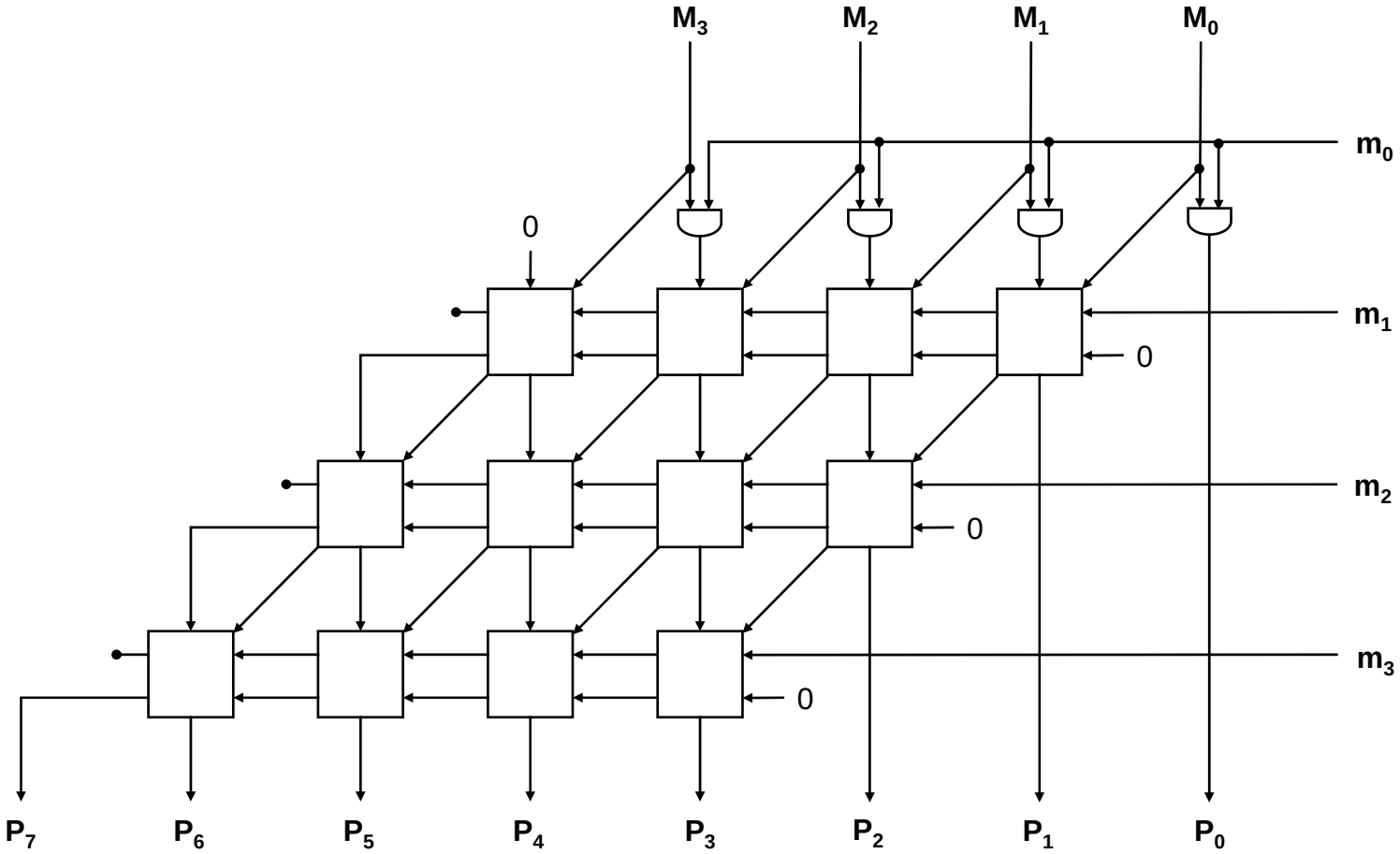
- ✓ Adaptar cada una de las celdas según el lugar que ocupan dentro del multiplicador



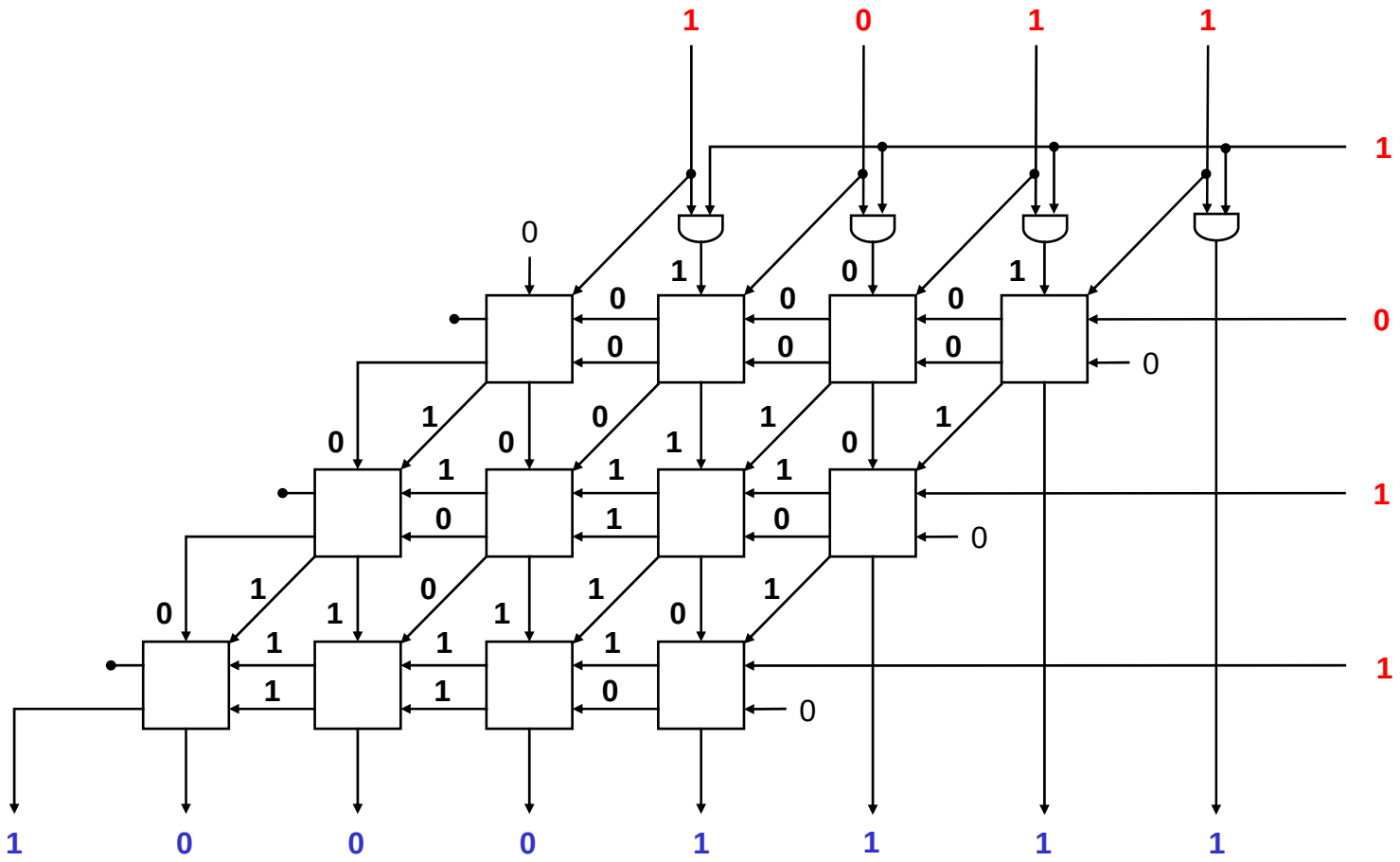


Multiplicación combinacional

Multiplicador combinacional (4 bits)



Multiplicación combinacional



Multiplicación combinacional

Multiplicación directa en C2: Multiplicador de Pezaris

Un nº m(m_{n-1}, m_{n-2},... m_1, m_0)representado en C2 tiene un valor igual a: $V(m) = -m_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} m_i \cdot 2^i$

La multiplicación de dos números A=(a_{n-1}, a_{n-2},... a_1, a_0) y B =(b_{n-1}, b_{n-2},... b_1, b_0) representados en C2 será por tanto:

$$A*B = A*\sum_{i=0}^{n-2} b_i * 2^i - A*b_{n-1} * 2^{n-1} =$$

$$A*B = (\sum_{j=0}^{n-2} a_j * 2^j - a_{n-1} * 2^{n-1})(\sum_{i=0}^{n-2} b_i * 2^i) - (\sum_{j=0}^{n-2} a_j * 2^j - a_{n-1} * 2^{n-1}) * b_{n-1} * 2^{n-1} =$$

$$A*B = \sum_{i=0}^{n-2} (\sum_{j=0}^{n-2} a_j * b_i * 2^{i+j}) - \sum_{i=0}^{n-2} a_{n-1} * b_i * 2^{n-1+i} - \sum_{j=0}^{n-2} a_j * b_{n-1} * 2^{j+n-1} + a_{n-1} * b_{n-1} * 2^{2n-2}$$

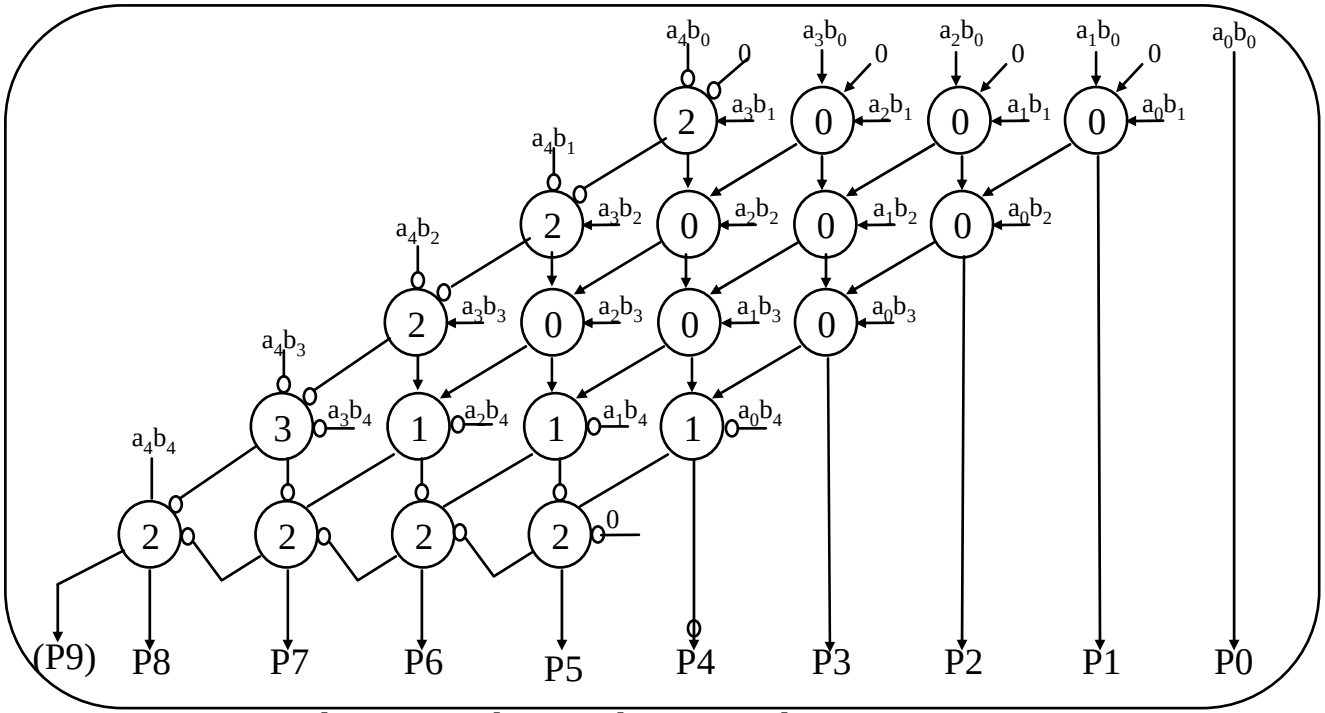
Ejemplo:

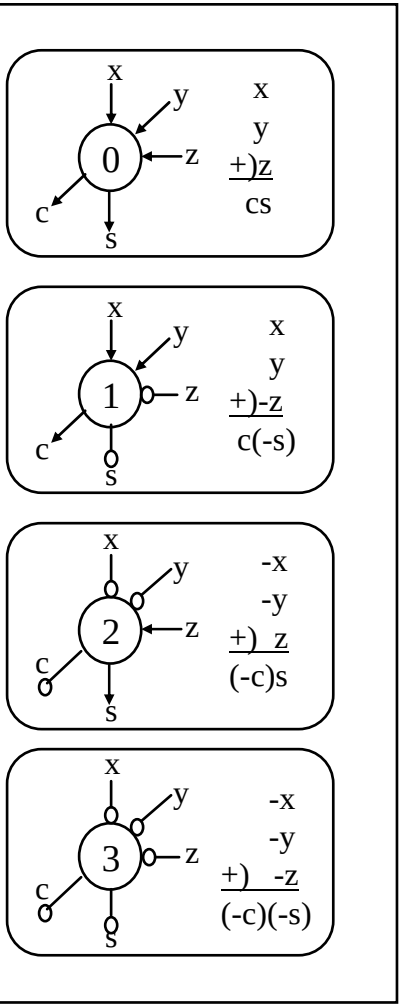
					(a_4)	a_3	a_2	a_1	a_0	= A
					* (b_4)	b_3	b_2	b_1	b_0	= B
				(a_4b_0)	a_3b_0	a_2b_0	a_1b_0	a_0b_0		
			(a_4b_1)	a_3b_1	a_2b_1	a_1b_1	a_0b_1			
		(a_4b_2)	a_3b_2	a_2b_2	a_1b_2	a_0b_2				
	(a_4b_3)	a_3b_3	a_2b_3	a_1b_3	a_0b_3					
a_4b_4	(a_3b_4)	(a_2b_4)	(a_1b_4)	(a_0b_4)						
P9	P8	P7	P6	P5	P4	P3	P2	P1	P0	= P

Entre paréntesis aparecen los productos con peso negativo

Multiplicación combinacional

Multiplicador de Pezaris





Ecuaciones para los 4 tipos de sumadores completos:

Tipos 0,1,2,3 $\Longrightarrow$ $S = X \otimes Y \otimes Z$

Tipos 0, 3 $\Longrightarrow$ $C = XY + YZ + ZX$

Tipo 1 $\Longrightarrow$ $C = XY + X\overline{Z} + Y\overline{Z}$

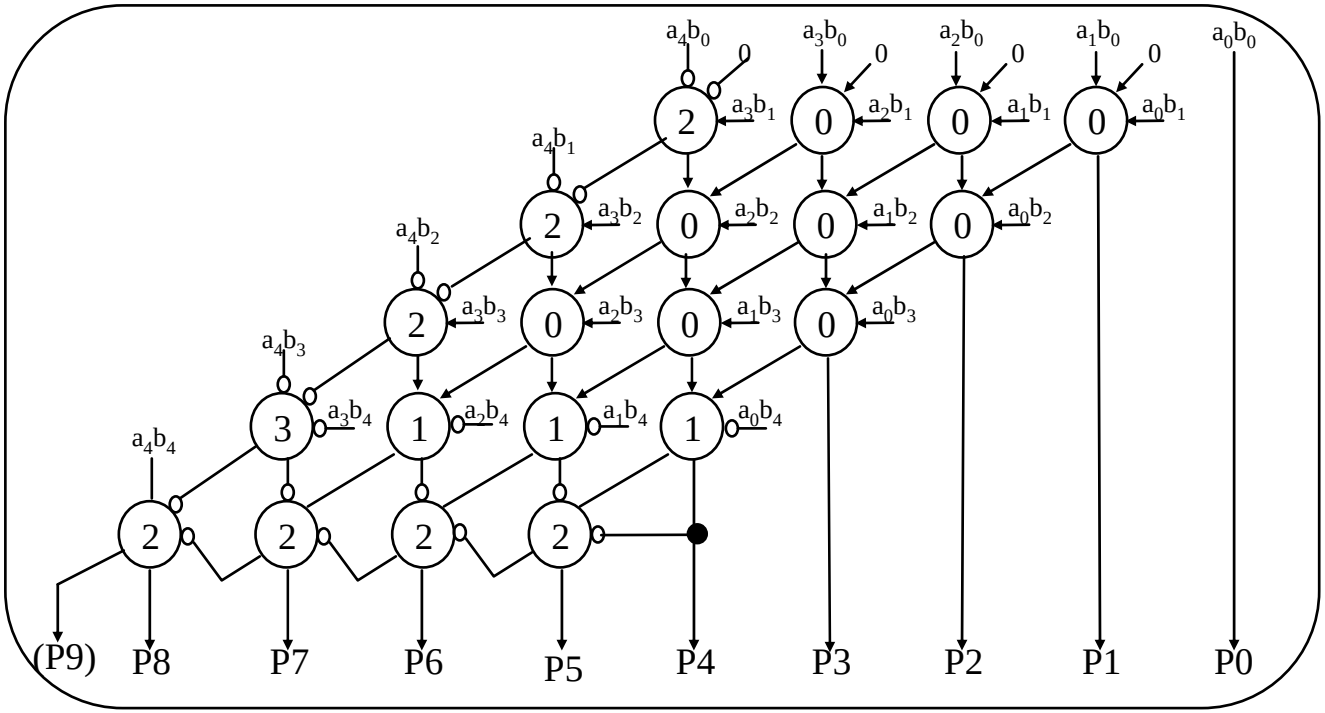
Tipo 2 $\Longrightarrow$ $C = ZY + Z\overline{X} + Y\overline{X}$

Ej: 11110*00111

Nota: El peso de P4 es negativo. Para evitar este tipo de salidas podemos conectar p4 con la entrada derecha del sumador siguiente, como se muestra en la siguiente página.

Multiplicación combinacional

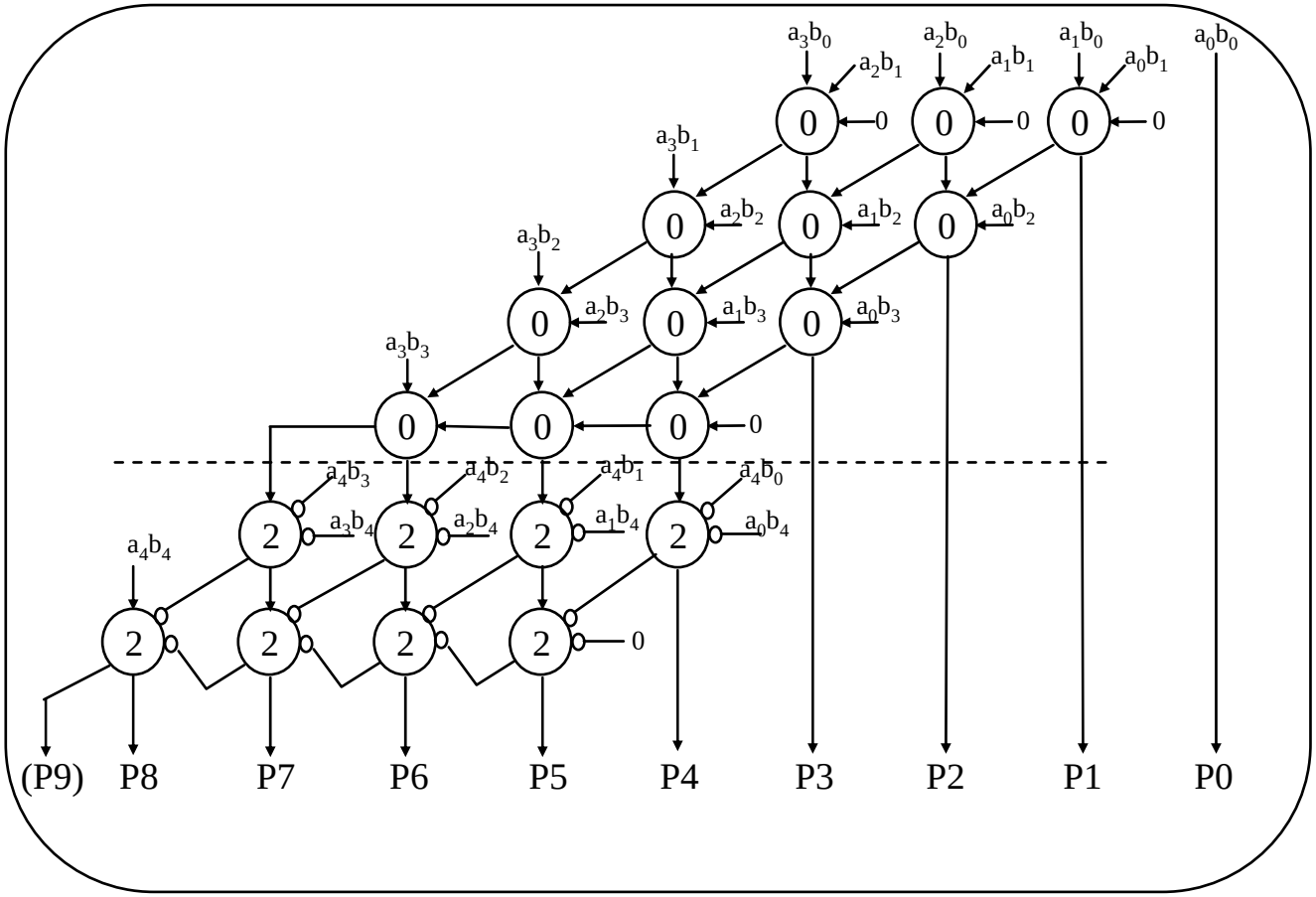
Multiplicador de Pezaris



Multiplicación combinacional

Variación del Multiplicador de Pezaris

Se trata de separar los sumandos positivos de los negativos, para reducir los tipos de sumadores, y hacer la estructura más uniforme.



Multiplicación combinacional

Multiplicación directa en C2: Multiplicador de Baugh-Wooley

▪Se busca incrementar la regularidad de la estructura usando sólo un tipo de sumador para todas las operaciones.

Fundamento:

- Se puede comprobar que **restar** el siguiente vector de m+1 bits:

  $X=(0,0, a_{m-2}\ k, a_{m-3}\ k,...,a_0\ k),\ k\in\{0,1\}$

- Es igual a **sumar** los vectores Y y Z siendo:

  $Y = (0,\overline{k},\overline{a_{m-2}}k,\overline{a_{m-3}}k,...,\overline{a_0}k)$

  $Z = (1,1,0,0,...,0,k)$

Demostración:

•Si k=0 => V(X)=0
V(Y+Z)=?

$\Longrightarrow$

Y = (0,1,0,...,0)
Z = (1,1,0,...,0)+
1(0,0,0,...,0)

Si k=1 => $X=(0,0,a_{m-2},...,a_0)$

$Y = (0,0,\overline{a_{m-2}},\overline{a_{m-3}},...,\overline{a_0})$

$Z = (1,1,0,0,...,1)$

$Y+Z = S1+S2$ con

$S1 = (1,1,\overline{a_{m-2}},\overline{a_{m-3}},...,\overline{a_0})$

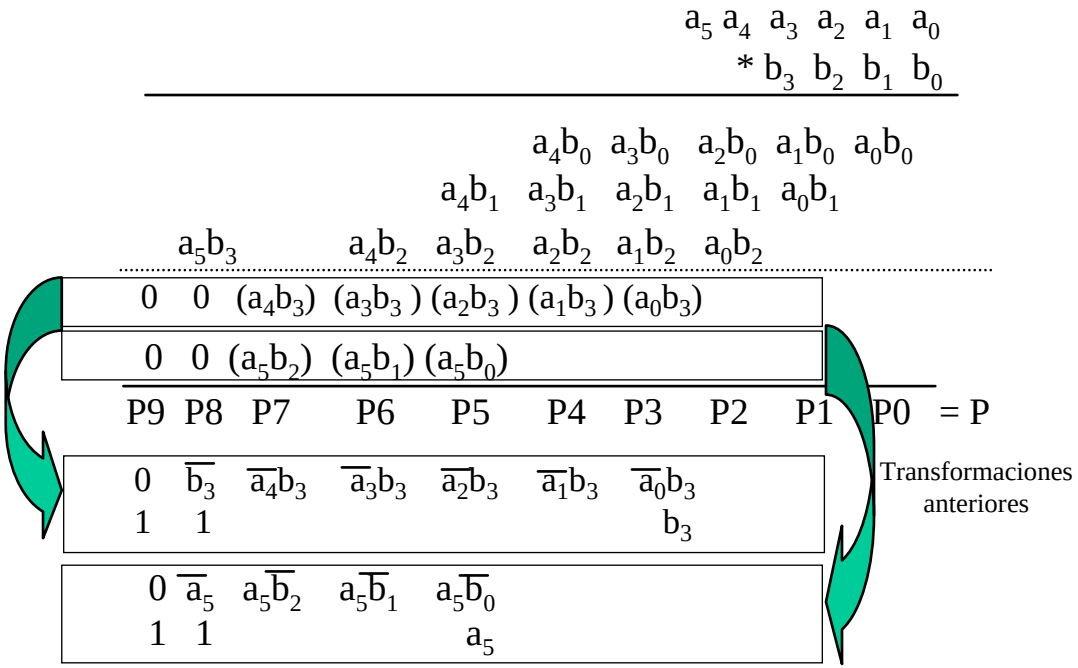
$S2 = (0,0,0,0,...,0,1)$

Por definición V(S1+S2)=V(C1(X)+1)=-X

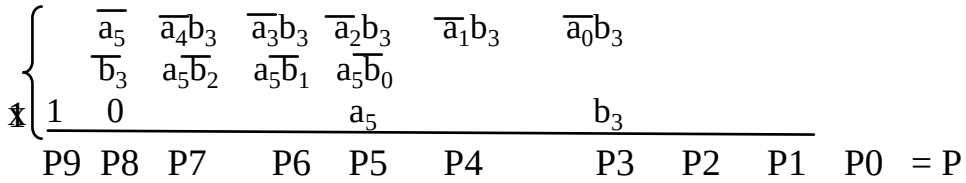
Multiplicación combinacional

Multiplicación directa en C2: Multiplicador de Baugh-Wooley

A*B siendo A = (a_5 a_4 a_3 a_2 a_1 a_0) m = 6
B = (b_3 b_2 b_1 b_0) n = 4

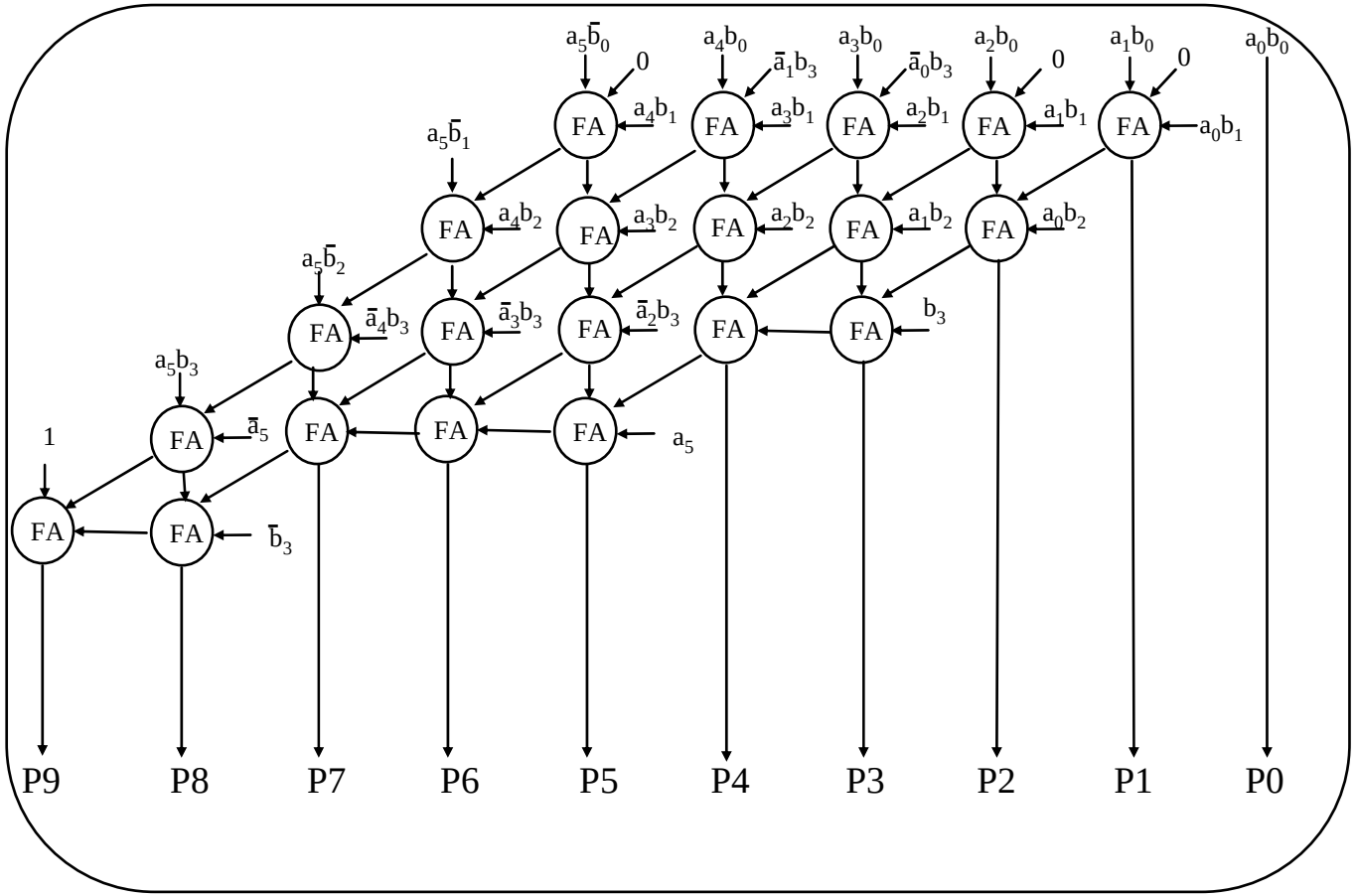


Entre paréntesis aparecen los productos negativos



Multiplicación combinacional

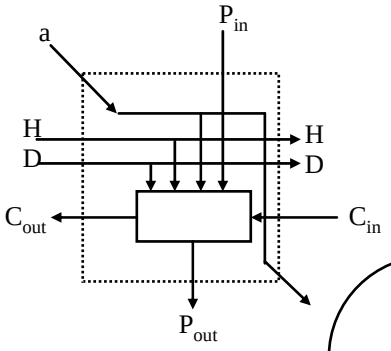
Multiplicador de Baugh-Wooley



Multiplicación combinacional

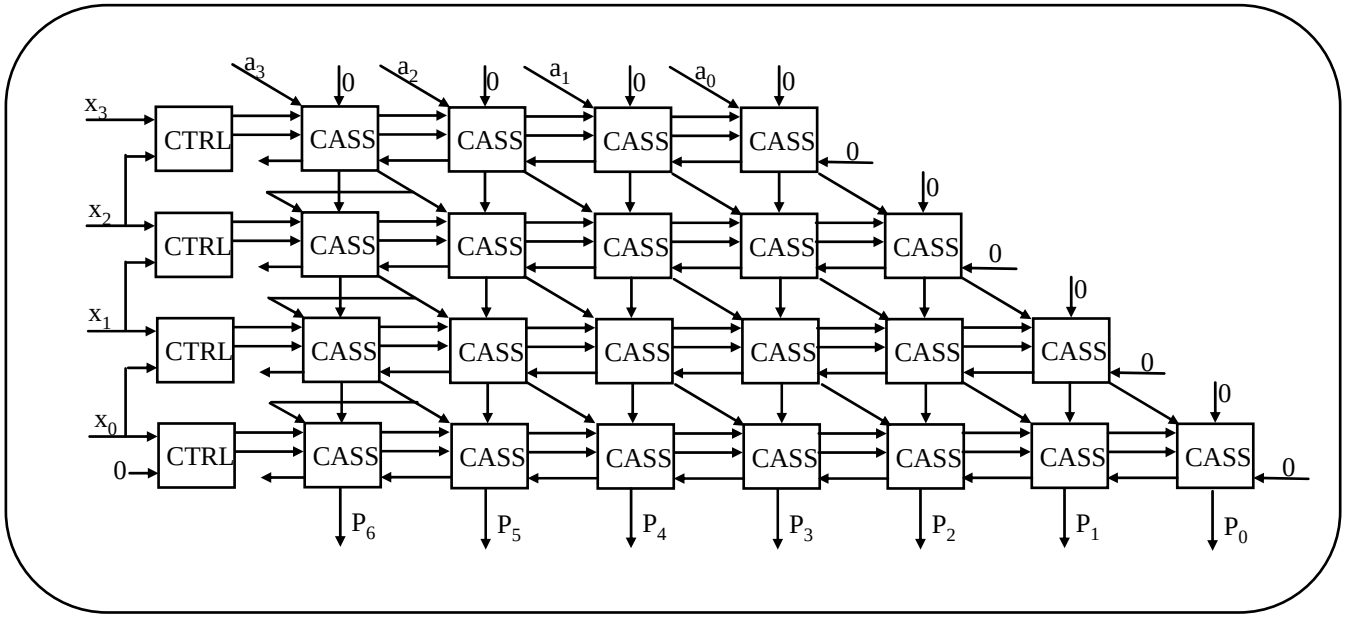
Multiplicadores recodificados

- Basado en el algoritmo de Booth, pero combinacional.
- Se crea una celda básica que sume, reste o desplace en función de dos bits del multiplicador:
  CASS (Controlled add/substract/shift).

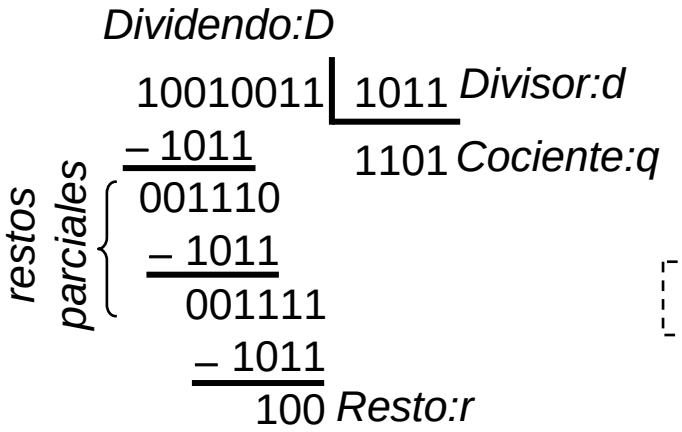


Si H=0 =>Desplazamiento => $P_{out}=P_{in}$
Si H=1 y D= 0 Suma
Si H=1 y D=1 Resta

$$P_{out} = P_{in} \oplus (a * H) \oplus (c_{in} * H)$$
$$c_{out} = (P_{in} \oplus D) * (a + c_{in}) + a * c_{in}$$



División de enteros sin signo



$D = d*q+r$

147 / 11 = 13, resto=4

Suposiciones:
Dividendo=D => 2n bits
Divisor=d => n bits
Cociente: q => n bits
Resto: r => n bits

Restricciones:
$0 \le r < d$
$0 < d \le D < 2^n*d \Rightarrow 0<q<2^n$

Impide:
- División por cero
- Cociente cero
- Rebose del cociente

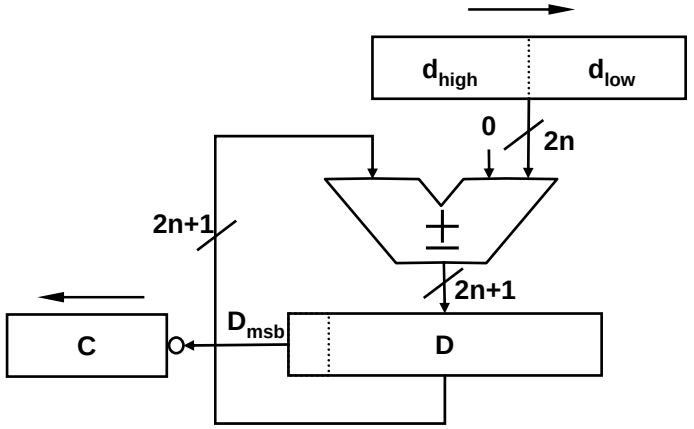
Método tradicional de división :

- Obtener los restos parciales y los bits del cociente recorriendo de izquierda a derecha los bits del dividendo:
  - si el resto parcial es mayor que el divisor, añadir un 1 al cociente; el nuevo resto parcial será la resta del resto parcial y del divisor
  - si el resto parcial es menor que el divisor, añadir un 0 al cociente y ampliar el resto parcial con un bit más del dividendo.

División de enteros sin signo

- usar 3 registros: resto/dividendo, divisor y cociente
- para alinear correctamente los restos parciales y el divisor, en cada iteración desplazar el divisor a la derecha
- para escribir en el mismo lugar cada uno de los bits del cociente, en cada iteración desplazarlo a la izquierda
- para evitar tener un comparador y un restador, usar éste último para comparar: el signo de la resta determinará si el resto parcial “cabe” entre el divisor

S0 : cargar (0,dividendo) en *D*
cargar divisor en d_{high}
$d_{low} = 0$
$C = 0$
S1 : $D \leftarrow D-(0,d)$
S2 : si $D_{msb} = 0$ entonces
desplazar *C* a la izquierda insertando un 1
si $D_{msb} = 1$ entonces
desplazar *C* a la izquierda insertando un 0
$D \leftarrow D+(0,d)$
desplazar *d* a la derecha
si *S1-S2* no se han repetido *n+1* veces ir a *S1*



División de enteros sin signo

Ejemplo:

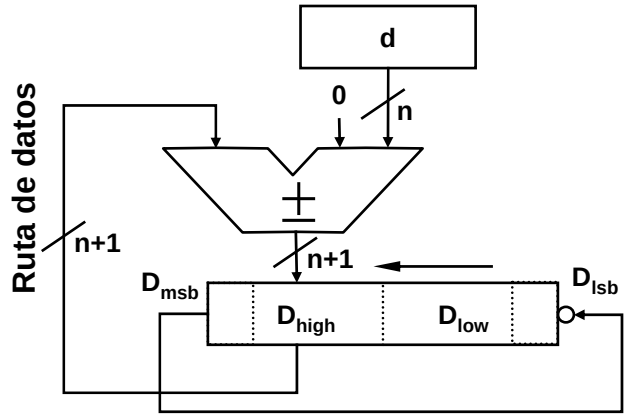
tras	*D*	*d*	*C*
S0	0**10010011**	**1011**0000	0000
S1	**1**11100011	10110000	0000
S2	010010011	01011000	0000
S1	**0**00111011	01011000	0000
S2	000111011	00101100	0001
S1	**0**00001111	00101100	0001
S2	000001111	00010110	0011
S1	**1**11111001	00010110	0011
S2	000001111	00001011	0110
S1	**0**00000100	00001011	0110
S2	00000**0100**	00000101	**1101**

147	11*16=178	
-29		0 (0)
147	11*8 = 88	
59		1 (1)
59	11*4=44	
15		1 (3)
15	11*2=22	
-7		0 (6)
15	11	
4		1 (13)
4	11	13

División de enteros sin signo

División sin restauración:

- ✓ considerar la secuencia de operaciones que se realiza tras la resta en S2:
  - si D_{msb} = 0 (“cabe”) se desplaza D a la izquierda y se resta d. Queda: 2·D-d
  - si D_{msb} = 1 (“no cabe”) se suma d, se desplaza el resultado y se resta d. Queda: 2(D+d)-d = 2·D+d
- ✓ entonces, en lugar de restaurar:
  - sumar o restar d en función de D_{msb}
  - en la última iteración restaurar el resto (sumándole d) si es necesario



S0: cargar (0,dividendo) en D
cargar divisor en d
S1: desplazar D a la izquierda
S2: $D_{high} \leftarrow D_{high}-(0,d)$
S3: si D_{msb} = 0 entonces $D_{lsb} \leftarrow 1$
si D_{msb} = 1 entonces $D_{lsb} \leftarrow 0$
S4: si D_{msb} = 0 entonces: a) desplazar D a la izquierda
b) $D_{high} \leftarrow D_{high}-(0,d)$
si D_{msb} = 1 entonces a) desplazar D a la izquierda
b) $D_{high} \leftarrow D_{high}+(0,d)$
si S3-S4 no se han repetido n-1 veces ir a S3
S5: si D_{msb} = 0 entonces $D_{lsb} \leftarrow 1$
si D_{msb} = 1 entonces $D_{high} \leftarrow D_{high}+(0,d)$, $D_{lsb} \leftarrow 0$

tras	*D*	*d*	
S0	0**1001 0011**	1011	(9, 3)
S1	**10010** 011 *0*	1011	(18, 6) <
S2	**0**0111 011 *0*	1011	(7, 6) R
S3	00111 011 *1*	1011	(7, 6) 1
S4	**01110** 11 *10*	1011	(14, 12) <
S4	**0**0011 11 *10*	1011	(3, 12) R
S3	00011 11 *11*	1011	(3, 12) 1
S4	**00111** 1 *110*	1011	(7, 8) <
S4	**1**1100 1 *110*	1011	(-4, 8) R
S3	11100 1 *110*	1011	(-4, 8) 0
S4	**11001** *1100*	1011	(-7, 0) <
S4	**0**0100 *1100*	1011	(4, 0) S
S5	0**0100** ***1101***	1011	(4, 0) 1

División por convergencia

<u>División por convergencia: Divisor multiplicativo</u>

Se puede usar en sistemas que tengan un multiplicador rápido.

<u>Sólo</u> calcula el cociente: Q=A/B

Idea: *Hallar una fracción equivalente a A/B pero con denominador 1.*

Proceso iterativo:

$$B*F_0*F_1*...*F_n \rightarrow 1$$
$$A*F_0*F_1*...*F_n \rightarrow Q$$

<u>Suposiciones:</u>
- A y B son fracciones positivas
- B está normalizado B=0,1xxxxx => B≥1/2
- A está alineado convenientemente.

<u>Proceso:</u>

B= 1-δ siendo $0 < \delta \leq ½$

La secuencia de denominadores que se construyen son de la forma:

- $B_i = B_{i-1} * F_i$

Eligiendo los F_i de forma que: $B_{i-1} < B_i < 1$

Puede tomarse: $\underline{F_0 = 1 + \delta}$

$B_0 = B*F_0 = (1 - \delta)* (1 + \delta) = 1 - \delta^2$

Con $F_0 > 1$ => $B_0 > B$, y $B_0 < 1$

En la siguiente iteración se elige: $F_1 = 1 + \delta^2$

$B_1 = B_0 * F_1 = (1 - \delta^2) (1 + \delta^2) = 1 - \delta^4$

Y de nuevo $B_1 > B_0$.

Para la i-ésima iteración: $F_i = 1 + \delta^{2^i}$

$$B_i = B_{i-1} * F_i = (1-\delta^{2^i})(1+\delta^{2^i}) = 1-(\delta^{2^i})^2 = 1-\delta^{2^{i+1}}$$

División por convergencia

Convergencia:

De la fórmula anterior se puede ver que B_i converge exponencialmente a 1.

También se puede ver teniendo en cuenta que:

$\delta \leq ½ \Rightarrow \delta^2 \leq ¼ \Rightarrow B_0 = 1 - \delta^2 \geq 0{,}11_{bin}$

$B_1 = 1 - \delta^4 \geq 1 - 1/16 = 15/16 = 0{,}1111$

En cada paso el número de 1s iniciales se duplica. Por ej. Para una máquina de 64 bits, obtener B= 0,111....111 sólo requiere 6 pasos.

Cálculo práctico de las iteraciones:

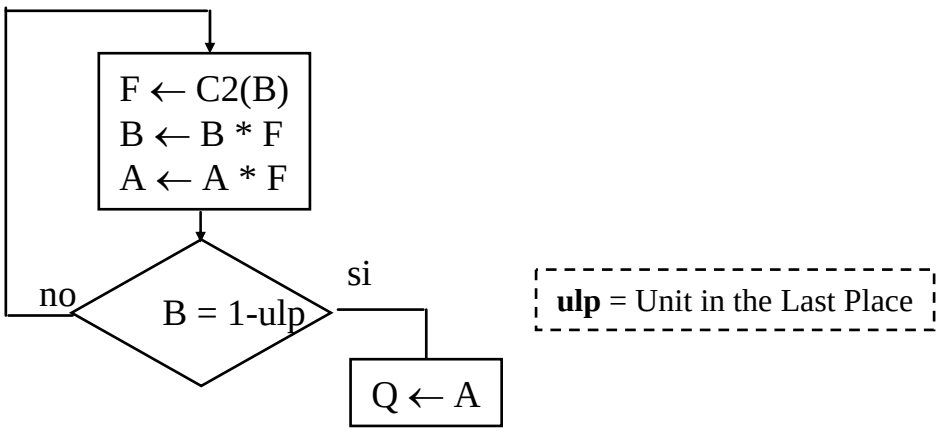
El cálculo de Fi es difícil y por ello habrá que calcularlo de otra manera. Se puede ver que:

$$F_i = 1 + \delta^{2^i} = 2 - (1 - \delta^{2^i}) = 2 - B_{i-1}$$

Y como $B_{i-1} = 0{,}1... \Rightarrow 2 - B_{i-1} = C2(B_{i-1})$

Por tanto para calcular cada F_i solamente se necesita calcular el C2 de B_{i-1}.

Algoritmo:



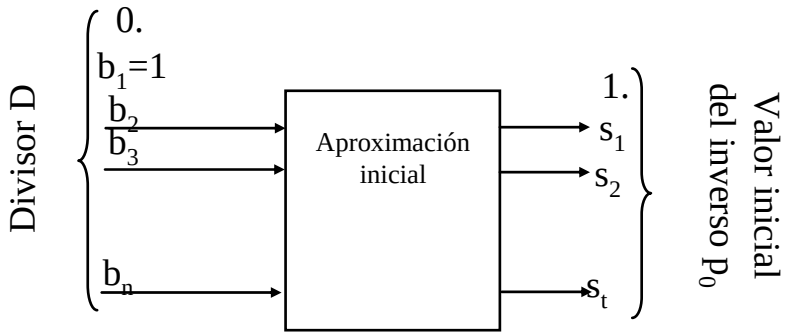
División por convergencia

División por convergencia: Cálculo del recíproco

Se puede usar en sistemas que tengan un multiplicador rápido.
Sólo calcula el cociente: Q=A/B
Idea: *Hallar Q multiplicando A* (1/B)*

Suposiciones:
- A y B son fracciones positivas
- B está normalizado B=0,1xxxxx => 1>B≥1/2 => 1<1/B≤2
- A<B para que Q<1.
- A está alineado convenientemente.

Método:
Se parte de una aproximación inicial al valor del inverso de B, de la forma:
$P_0 = 1,s_1s_2...s_t$
Esta aproximación inicial se puede almacenar en una tabla en ROM, de forma que dados los k bits más significativos de B (excluyendo el 0,1 inicial que es constante) la salida muestra una aproximación lo más exacta posible, sobre t bits al inverso de B.



Ej: Tabla que genera 4 bits del inverso a partir de 2 bits de B.

Entradas		Salidas(p_0)	
b_2 b_3	(valor decimal)	s_1 s_2 s_3 s_4	(Valor decimal)
0 0	0,5	1 1 1 1	1,9375
0 1	0,625	1 0 0 1	1,5625
1 0	0,75	0 1 0 1	1,3125
1 1	0,875	0 0 1 0	1,125

División por convergencia

<u>División por convergencia: Cálculo del recíproco</u>

El proceso iterativo para calcular el inverso de B consiste en:

- p_0 se toma de la tabla.
- $I_0 = p_0 * B$

- $p_i = p_{i-1} * (2 - I_{i-1})$
- $I_i = I_{i-1} * (2 - I_{i-1})$

Puede comprobarse que: $\lim_{i \to \infty} I_i = 1$ (1)

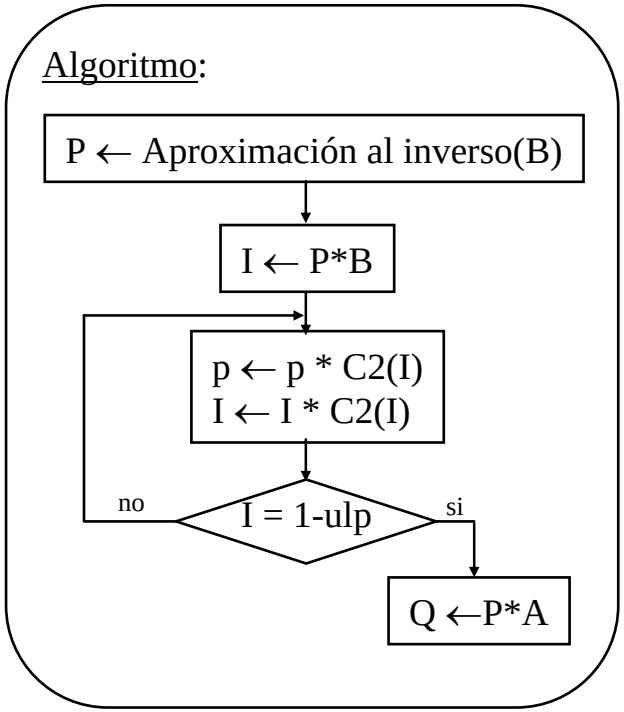
Por tanto:

p_0	$I_0 = p_0 * B$
$p_1 = p_0 * (2 - I_0)$	$I_1 = I_0(2 - I_0) = p_0 * B * (2 - I_0) = p_1 * B$
$p_2 = p_1 * (2 - I_1)$	$I_2 = I_1(2 - I_1) = p_1 * B * (2 - I_1) = p_2 * B$

En general:

$I_i = p_i * B$ y según (1): $\lim_{i \to \infty} p_i = \frac{1}{B}$

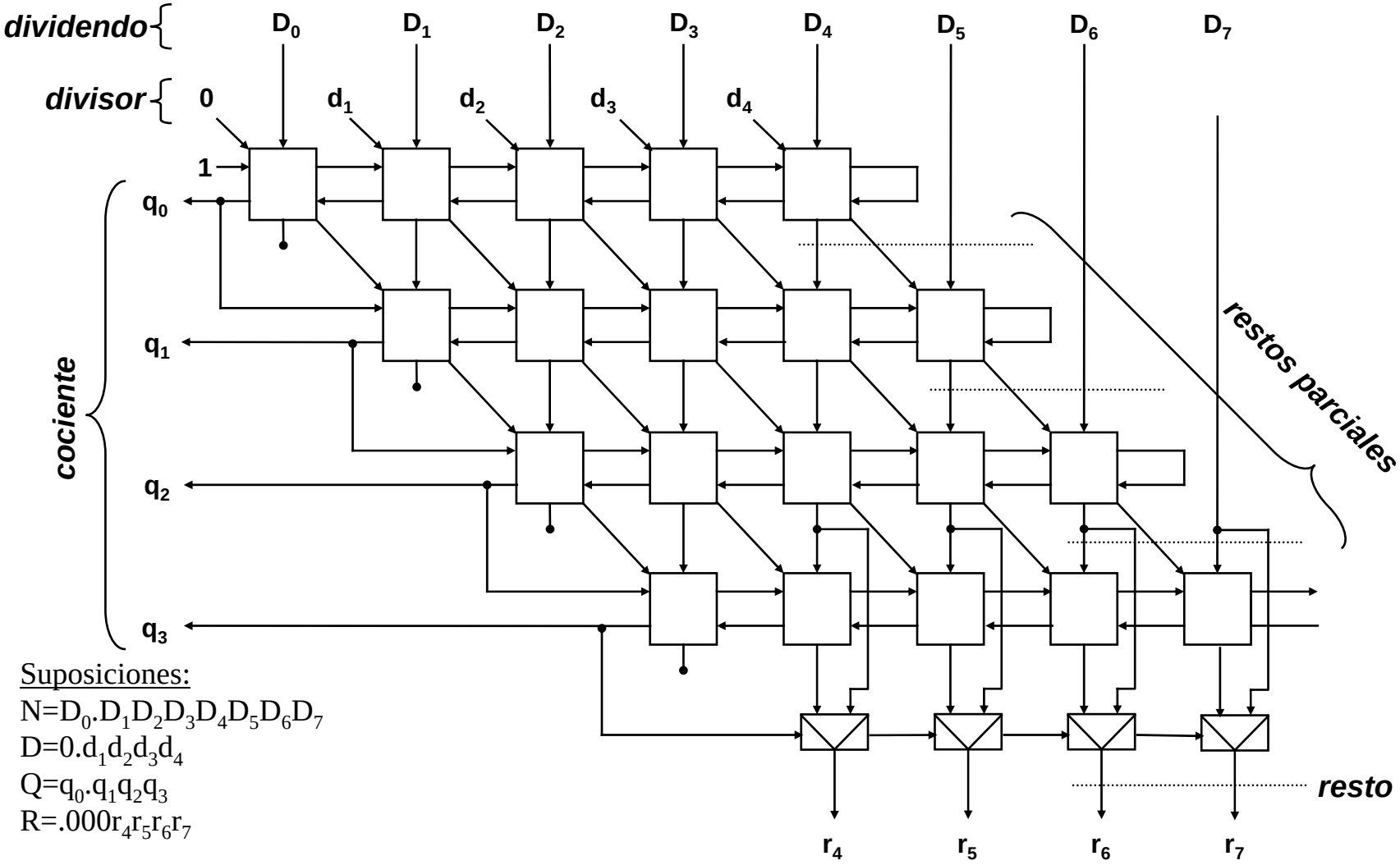
<u>Algoritmo</u>:



ulp = Unit in the Last Place

División combinacional

División sin restauración



Suposiciones:

$N=D_0.D_1D_2D_3D_4D_5D_6D_7$

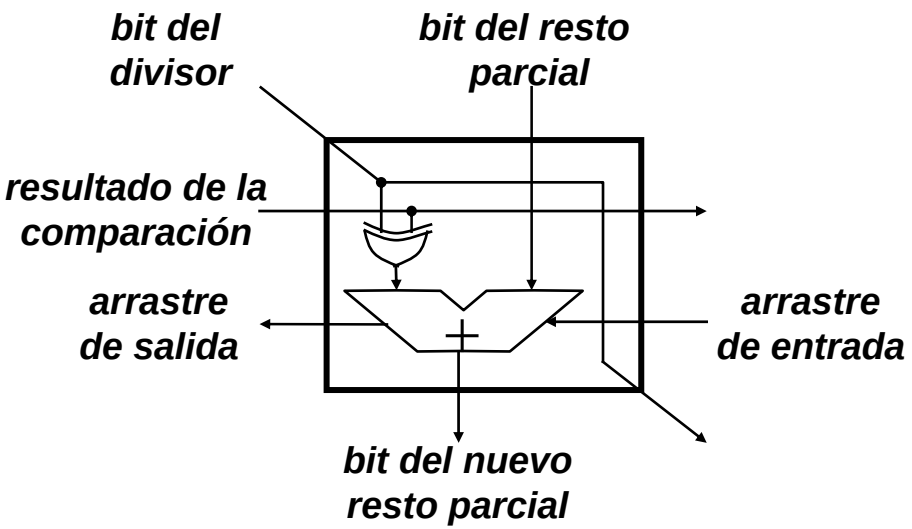
$D=0.d_1d_2d_3d_4$

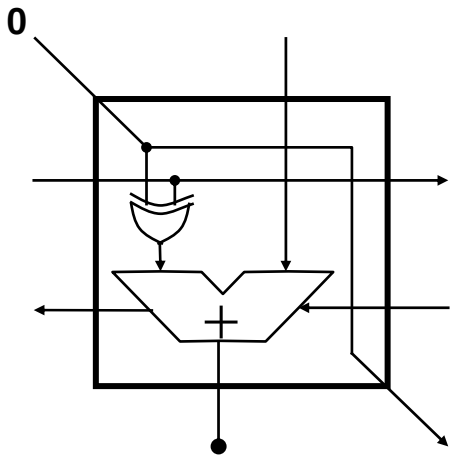
$Q=q_0.q_1q_2q_3$

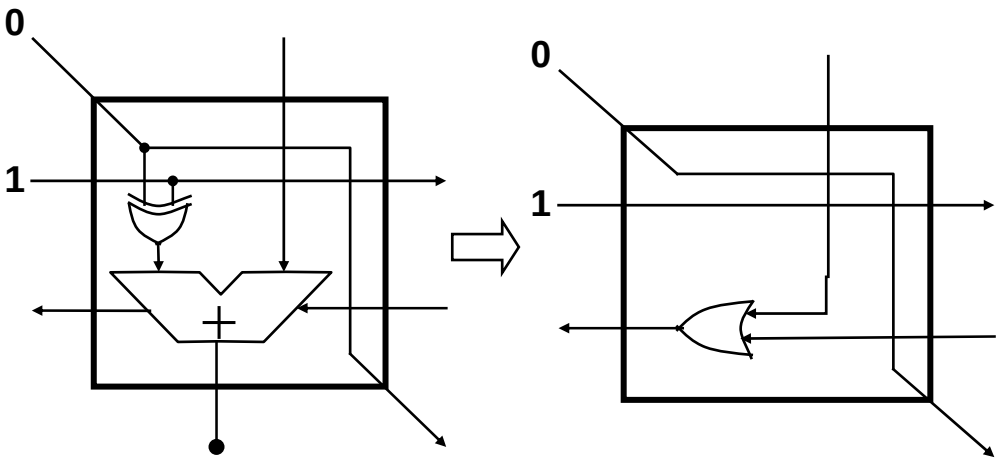
$R=.000r_4r_5r_6r_7$

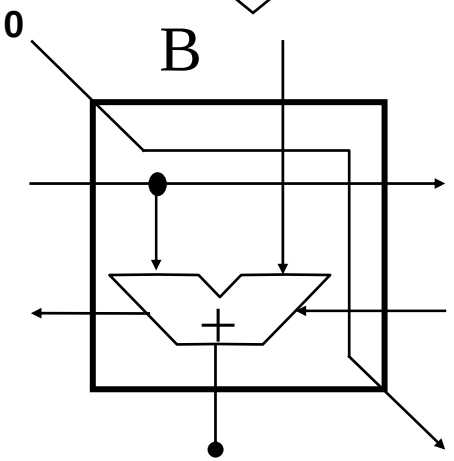
División combinacional

Celdas básicas

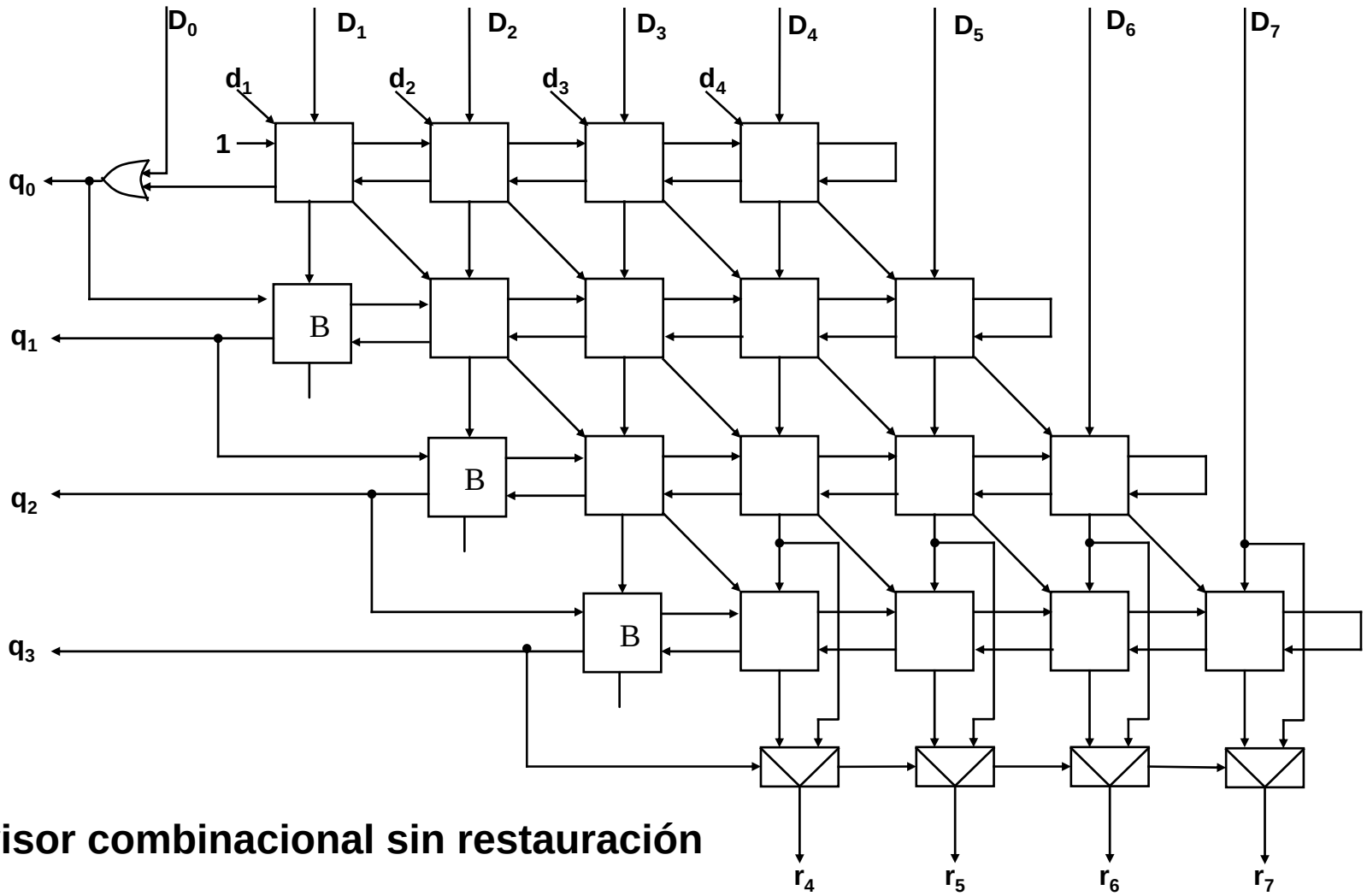






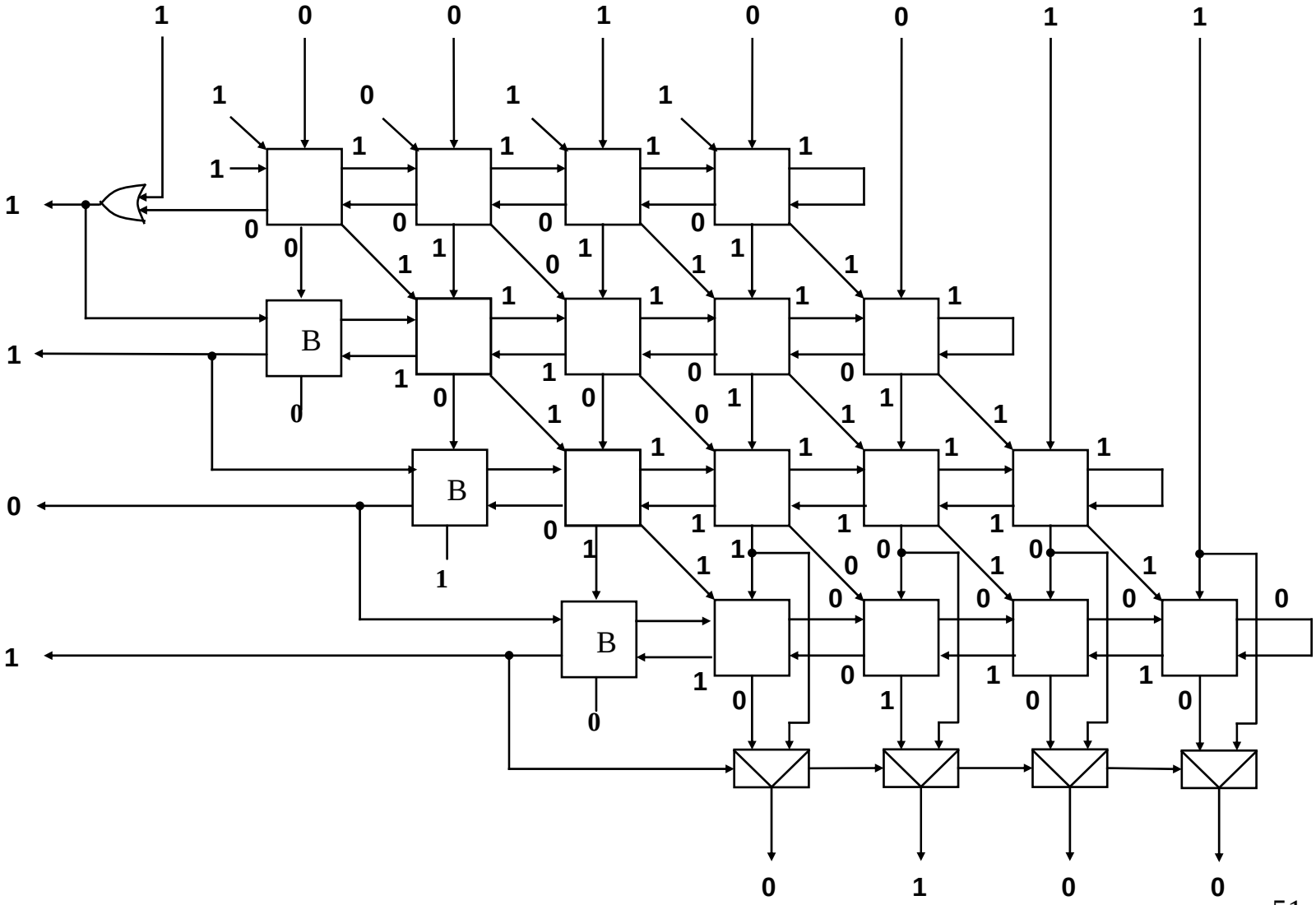


División combinacional



Divisor combinacional sin restauración

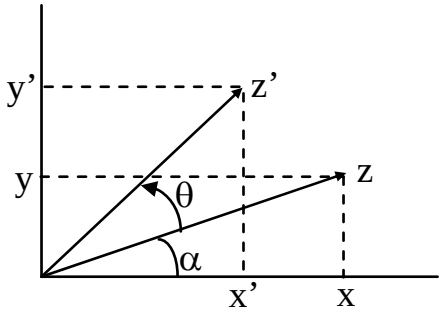
División combinacional



Cálculos trigonométricos: CORDIC

COordinate Rotation DIgital Computer

Objetivo: Calcular el seno de un ángulo de modo iterativo y sencillo



z=(x,y)
$x = |z|\cos\alpha$
$y = |z|\text{sen}\,\alpha$

Z'=(x',y')

$X' = |z'|\cos(\alpha+\theta) = |z|(\cos\alpha\cos\theta - \text{sen}\,\alpha\,\text{sen}\theta) = x\cos\theta - y\,\text{sen}\,\theta$

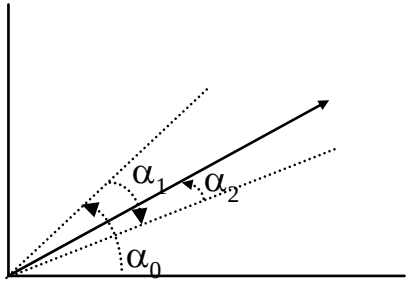
Si la rotación hubiese sido anti-horaria cambiaría el signo de y sen θ

$X' = x\cos\theta - y\,\text{sen}\,\theta = \cos\theta\,(x - y\tan\theta)$

$y' = y\cos\theta + x\,\text{sen}\,\theta = \cos\theta\,(y + x\tan\theta)$

Si tan θ es una potencia de 2 => las coordenadas de z' pueden calcularse mediante sumas/restas y desplazamientos.

Partiendo de un vector z_0 de argumento 0^0, hacer rotaciones sucesivas hasta alcanzar el ángulo θ, del que queremos hallar el seno o el coseno.



$\theta = \alpha_0 \pm \alpha_1 \pm \alpha_2 \pm \ldots \pm \alpha_n$

Siendo $\alpha_i = \text{atan}(2^{-i})$

$|Z''| = |Z'| / \cos\theta$

Arco	tan
45	2^0
26,56	2^{-1}
14,03	2^{-2}
7,12	2^{-3}
...	

Cálculos trigonométricos: CORDIC

Proceso iterativo:

Se parte de : $z_0 = (x_0, y_0) = (x_0, 0)$

Al hacer sucesivas rotaciones, llegaremos a vectores z_{i+1} con coordenadas:

$x_{i+1} = x_i \pm y_i * 2^{-i}$
$y_{i+1} = y_i \pm x_i * 2^{-i}$ ⟹

El cálculo sólo implica sumas/restas y desplazamientos

Después de n iteraciones se dispondrá de un vector z_n tal que:

$$|z_n| = \frac{1}{\cos\alpha_{n-1}} |z_{n-1}| = \frac{1}{\cos\alpha_{n-1}} \frac{1}{\cos\alpha_{n-2}} |z_{n-2}| = \ldots = \frac{1}{\prod_{1=0}^{n-1} \cos\alpha_i} |z_0|$$

Pero: $\lim_{n \to \infty} (\prod_{1=0}^{n-1} \cos\alpha_i) = 0{,}6073$

Por ello, eligiendo z_0=(0,6073,0), para n suficientemente grande se obtendrá un vector z_n con módulo 1, y con argumento θ, y por tanto:

x_n=cos θ

y_n=sen θ