Unidad aritmética Lógica

Multiplicación y división

Arquitectura de Computadoras I

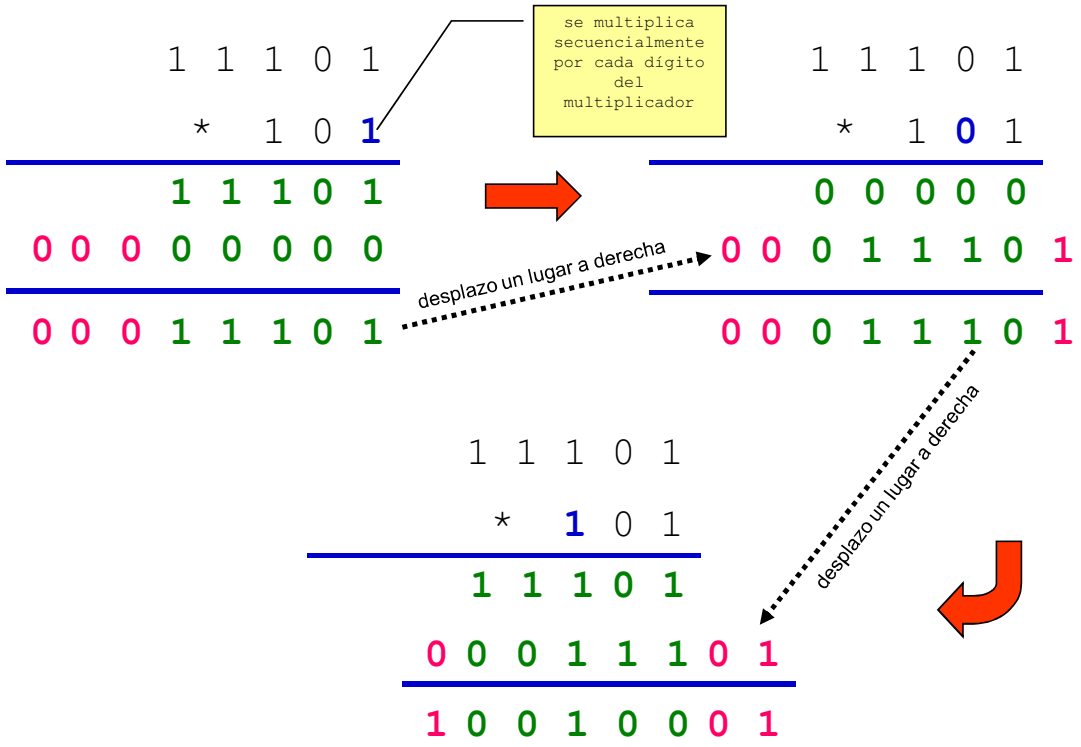
Prof. Marcelo Tosini
2011

Multiplicación binaria sin signo

1. Se multiplica cada digito del multiplicador por el multiplicando.
2. Luego se suman los resultados.

```
      1 1 1 0 1   ◂ Multiplicando
    *   1 0 1     ◂ Multiplicador
  -----------
      1 1 1 0 1
    0 0 0 0 0
+ 1 1 1 0 1
  -----------
1 0 0 1 0 0 0 1
```

Multiplicación binaria sin signo



Algoritmo de Horner

Sean: $\begin{cases} X = x_{n-1}, x_{n-2}, \ldots, x_1, x_0 \\ Y = y_{n-1}, y_{n-2}, \ldots, y_1, y_0 \end{cases}$ $x_i, y_i \in \{0, B-1\}$

dos operandos de n dígitos c/u.

Sea $Z = X \cdot Y$

con $Z = z_{2n-1}, z_{2n-2}, \ldots, z_1, z_0$

entonces

$$Z = \sum_{i=0}^{n-1} x_i B^i Y$$

que se expande como

$$Z = ((\ldots(0 \cdot B + x_{n-1}Y)B + x_{n-2}Y)B + \ldots)B + x_2 Y)B + x_1 Y)B + x_0 Y$$ (expansión de Hörner)

Algoritmo:

$P(n-1) = x_{n-1}Y$

for i ***in*** n-2, n-3, …, 1, 0 ***loop***

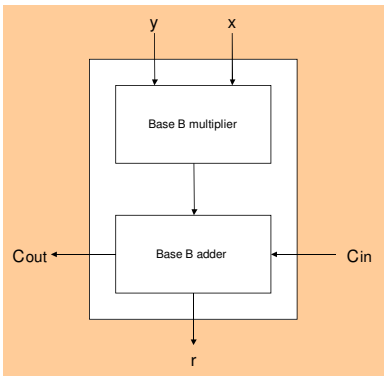
$P(i) = P(i+1) \cdot B + x_i Y$

end loop

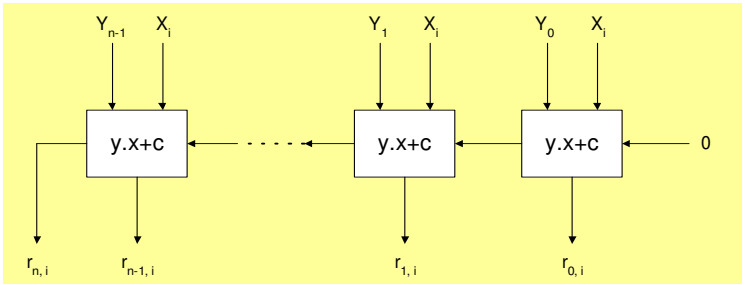
$Z = P(0)$

Algoritmo de Horner - Circuito I

Celda básica

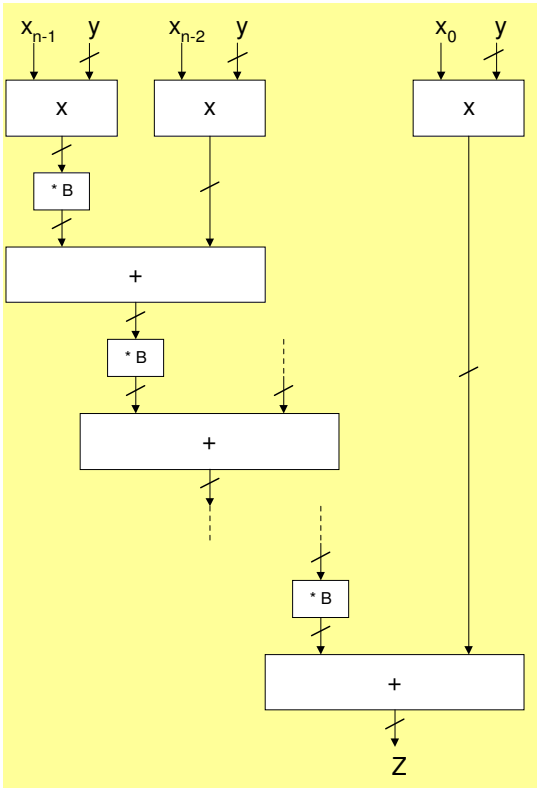


Iteración en el tiempo



Algoritmo de Horner - Circuito II

Iteración en el espacio



Algoritmo de Horner - Factorización derecha a izquierda

PERMITE REDUCIR A LA MITAD LA LONGITUD DEL SUMADOR

Sea n = 4 → $X = x_3, x_2, x_1, x_0$

Factorizando de izquierda a derecha

$$Z = \sum_{i=0}^{3} x_i B^i Y = x_0 Y + x_1 B^1 Y + x_2 B^2 Y + x_3 B^3 Y$$

$$Z = x_0 Y + B(x_1 Y + B x_2 Y + B^2 x_3 Y)$$

$$Z = x_0 Y + B(x_1 Y + B(x_2 Y + B x_3 Y))$$

$$Z = x_0 Y + B(x_1 Y + B(x_2 Y + B(x_3 Y)))$$

Y necesita 3 bits

Z necesita 7 bits

Factorizando de derecha a izquierda

$$\frac{Z}{B^3} = \frac{x_0 Y}{B^3} + \frac{x_1 B^1 Y}{B^3} + \frac{x_2 B^2 Y}{B^3} + \frac{x_3 B^3 Y}{B^3}$$

$$\frac{Z}{B^3} = \frac{x_0 Y}{B^3} + \frac{x_1 Y}{B^2} + \frac{x_2 Y}{B^1} + \frac{x_3 Y}{B^0}$$

$$\frac{Z}{B^3} = x_0 B^{-3} Y + x_1 B^{-2} Y + x_2 B^{-1} Y + x_3 Y$$

$$\frac{Z}{B^3} = (((x_0 Y) B^{-1} + x_1 Y) B^{-1} + x_2 Y) B^{-1} + x_3 Y$$

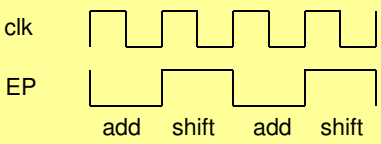
Y necesita 3 bits

Z necesita 3 bits

Algoritmo Shift & Add

Algoritmo:

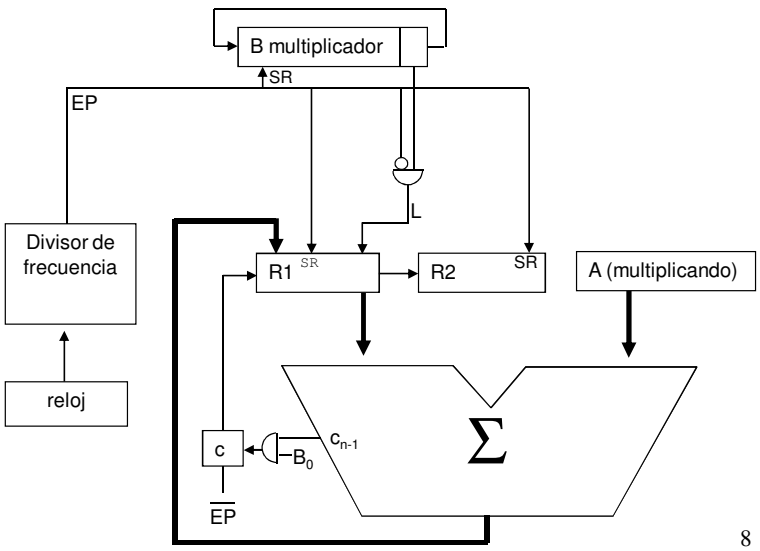
a) El resultado se inicia en cero
b) Si el i-esimo bit del multiplicador es
   "0", no se hace nada
   "1", se suma el multiplicando al resultado
c) Se desplaza el resultado a derecha un lugar
d) Se vuelve a b) hasta terminar los bits del multiplicador



EP = 0: (Add)

Si $B_0 = 1 \rightarrow L = 1$ y R1 = R1 + A
C = C_{n-1} and B_0

EP = 1: (Shift)
B = B >>1
R2 = $R1_0$ concat (R2 >>1)
R1 = C concat (R1 >>1)



L: *Load (cargar)*
SR: *Shift Right*
EP: *Enable Pulse*

Algoritmo de multiplicación por sumas y restas

- Multiplicación binaria sin signo
- Método para multiplicar operandos sin signo**:**
  1. Añadir al multiplicador un 0 por la izquierda
  2. Añadir al multiplicando (A) un 0 por la izquierda y calcular -A (en C´2)
  3. Recorrer el multiplicador desde el LSB al MSB y
     - restar $A \cdot 2^i$ cuando se alcance el primer 1 (de una secuencia)
     - sumar $A \cdot 2^i$ cuando se alcance el siguiente 0 (fin de secuencia)

Ej. 1: Sea A: 1100 (12), y B:1010 (10):

1. B': **0**1010
2. A : **0**1100 y −A : 10100
3. A x B = $-A \cdot 2^1 + A \cdot 2^2 - A \cdot 2^3 + A \cdot 2^4$

```
  1 1 1 1 0 1 0 0 0 : -A·2^1
  0 0 0 1 1 0 0 0 0 :  A·2^2
  1 1 0 1 0 0 0 0 0 : -A·2^3
  0 1 1 0 0 0 0 0 0 :  A·2^4
  -----------------------------
  0 0 1 1 1 1 0 0 0 : 120
```

Ej. 2: Sea A: 1100 (12), y B:0011 (3):

1. B': **0**0011
2. A : **0**1100 y −A : 10100
3. A x B = $-A \cdot 2^0 + A \cdot 2^2$

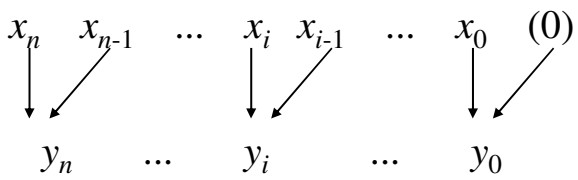
```
  1 1 1 1 1 0 1 0 0 : -A·2^0
  0 0 0 1 1 0 0 0 0 :  A·2^2
  -----------------------------
  0 0 0 1 0 0 1 0 0 : 36.
```


Algoritmo de recodificación de Booth

- Desarrollado para acelerar las multiplicaciones en computadoras antiguas
- Al ocurrir productos parciales de 0´s se pueden evitar las sumas y realizar solamente los desplazamientos.
- Demora variable – Depende del número de 1´s en la cadena
- Booth observó que una cadena de 1's puede ser reemplazada por:

$$2^j + 2^{j-1} + \cdots + 2^{i+1} + 2^i = 2^{j+1} - 2^i$$

Ejemplo de recodificación de Booth



$y_i = x_{i-1} - x_i$

x_i	x_{i-1}	**Operación**	**Comentario**	y_i
0	0	sólo shift	cadena de 0´s	0
1	1	sólo shift	cadena de 1´s	0
1	0	resta y shift	inicio cadena 1´s	-1
0	1	suma y shift	fin cadena de 1´s	1

Ejemplo

0011110011(0)

$0100\overline{1}0\overline{1}0\overline{1}$

Multiplicación de Booth

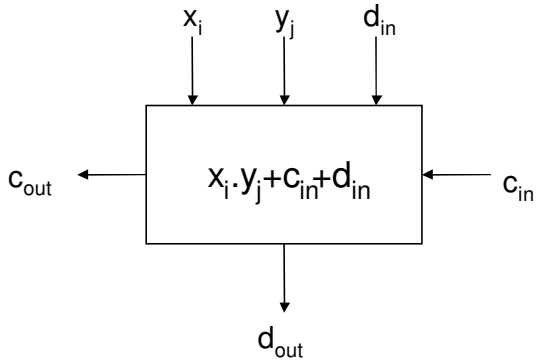
	A	**1 0 1 1**	**(-5_{10})**
	X	**1 1 0 1**	**(-3_{10})**

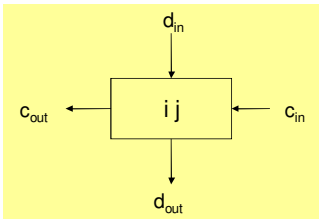
	Y	**0 -1 1 -1**	**(recodificación)**

(-1) suma –A		**0 1 0 1**	
Shift		**0 0 1 0 1**	
(+1) suma +A		**1 0 1 1**	
----------			
		1 1 0 1 1	
Shift		**1 1 1 0 1 1**	
(-1) suma –A		**0 1 0 1**	
----------			
		0 0 1 1 1 1	
Shift		**0 0 0 1 1 1 1**	**($+15_{10}$)**

Algoritmo de multiplicación Ripple-Carry

Celda básica





Con

$$z_{ij} = (z_1, z_0) = x_i y_j + c_{in} + d_{in} = (c_{out}, d_{out})$$

Ejemplo: Si x_i=1 , y_j=1 , c_{in} = 1 y d_{in} = 1 entonces

(z_1,z_0) = 1.1 + 1 + 1 = (1, 1)

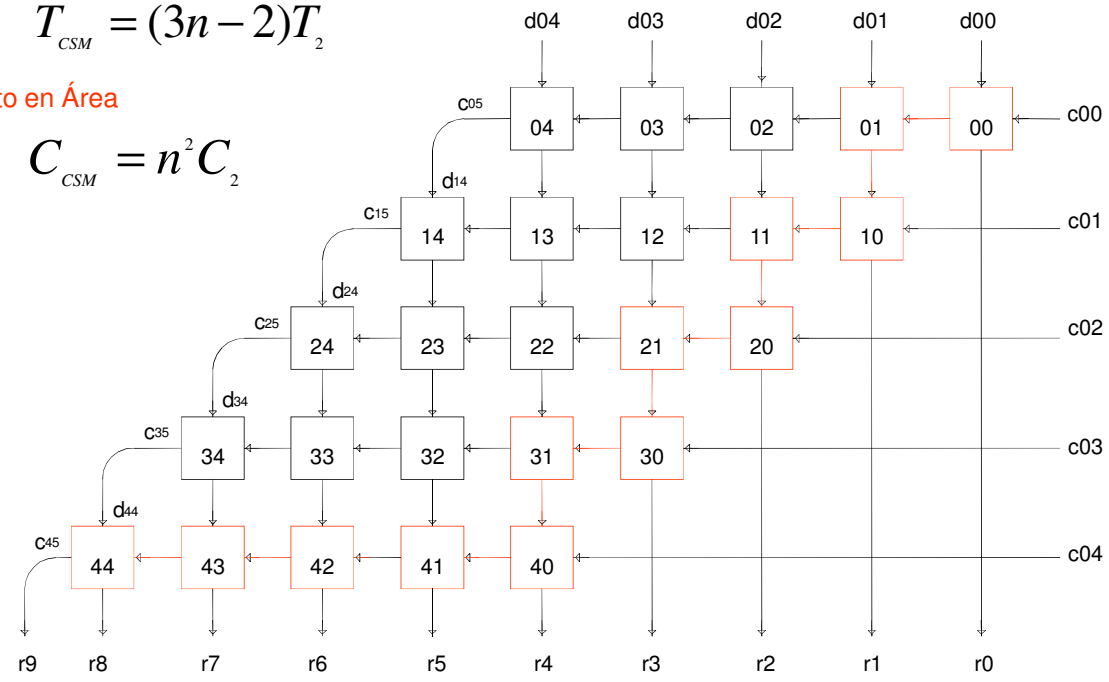
Algoritmo de multiplicación Ripple-Carry

Tiempo de cálculo

$$T_{CSM} = (3n-2)T_2$$

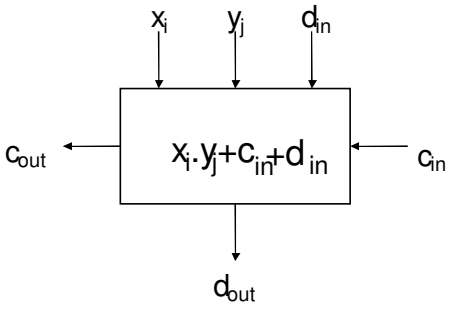
Costo en Área

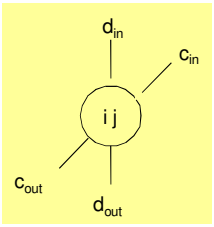
$$C_{CSM} = n^2 C_2$$



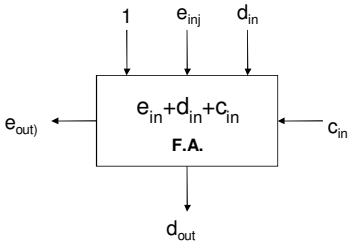
Algoritmo de multiplicación Carry-Save

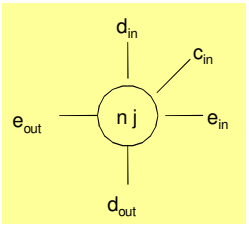
Celda básica: La misma que para ripple-carry





Agrega una fila adicional con la celda siguiente:





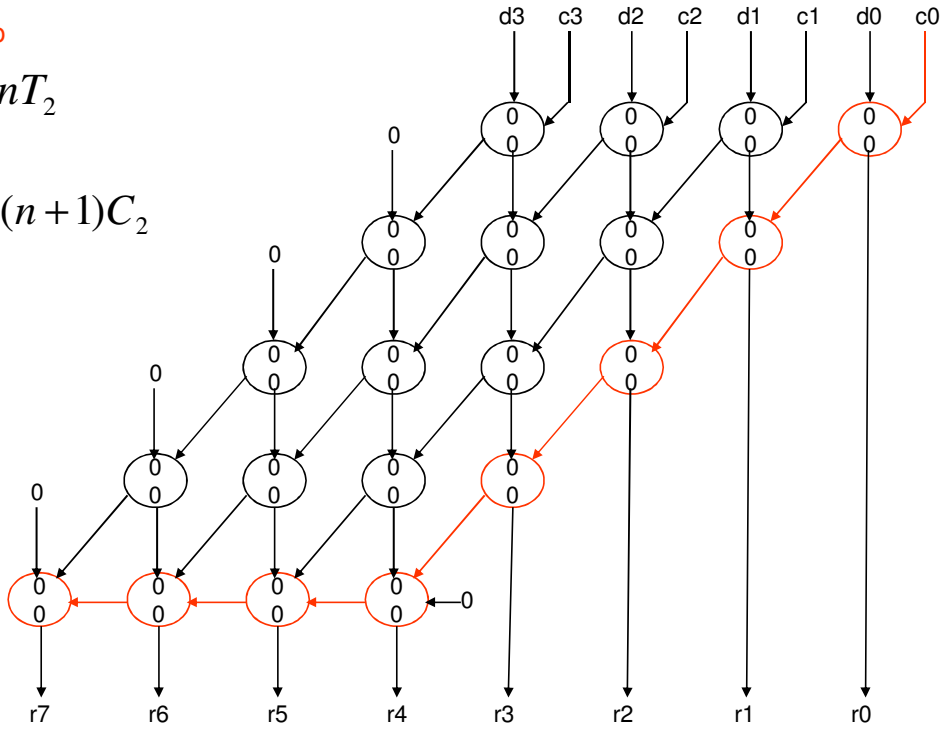
Algoritmo de multiplicación Carry-Save

Tiempo de cálculo

$$T_{CSM} = 2nT_2$$

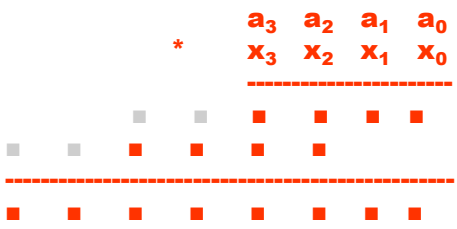
Costo en Área

$$C_{CSM} = n(n+1)C_2$$



Multiplicación en otras bases mayores

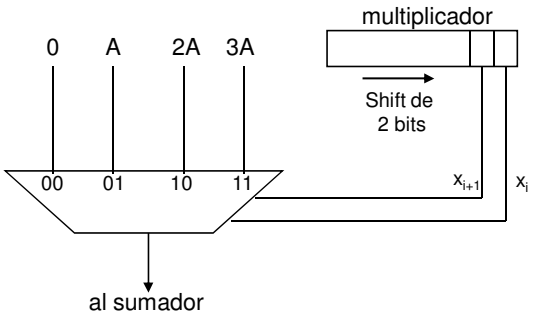
- Reduce el número de productos parciales a ser sumados
- Agrupa varios bits del multiplicador juntos
- Trabaja en bases mayores a 2
- Suma los productos parciales mas rápido



Consideraciones:

- El desplazamiento es multibit
- La celda básica de producto ya no es un AND
- Es necesario un MUX 4:1 con entradas: 0, A, 2A y 3A

Multiplicación en otras bases mayores

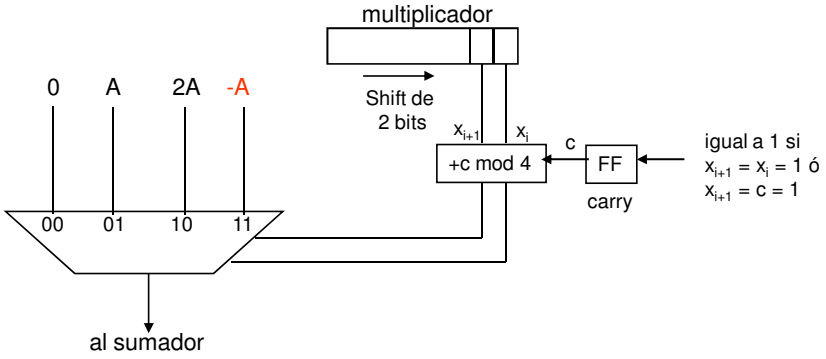


- **0, A y 2A son fáciles de calcular**
- **3A = A + 2A → requiere una suma previa**
- **Eventualmente se puede precalcular 3A y almacenarlo en un registro**

Multiplicación en otras bases mayores

- **Otra solución es sumar –A y generar un acarreo a la siguiente etapa de multiplicación de manera que:**
  - **Si debía sumar 0, ahora suma A**
  - **Si debía sumar A, ahora suma 2A**
  - **Si debía sumar 2A, ahora suma 3A**
  - **Si debía sumar 3A, ahora genera acarreo a la sig. etapa**

NOTA: Recordar que el producto parcial se desplaza 2 lugares a derecha con lo que el efecto real de sumar A es sumar 4A, el de sumar 2A es sumar 8A, y así sucesivamente.



Multiplicación en otras bases mayores

Ejemplo:

Sea A = 000101 (5)
-A = 111011 (-5)
3A =001111 (15) y
P parcial = 00000111 (7)

Multiplicando = 0011

Por definición: primero sumo 3A y luego sumo 0

Con mejora: primero resto A y luego sumo A

P = ((P + 3A) >> 2) + 0

P	0 0 0 0 0 1 1 1	(7)
3A	0 0 0 0 1 1 1 1	(15)

P´	0 0 0 1 0 1 1 0	(22)
>>	0 0 0 0 0 1 0 1 1 0	
0	0 0 0 0 0 0 0 0	
P´´	0 0 0 0 0 1 0 1 1 0	

P = ((P + -A) >> 2) + A

P	0 0 0 0 0 1 1 1	(7)
-A	1 1 1 1 1 0 1 1	(-5)

P´	0 0 0 0 0 0 1 0	(2)
>>	0 0 0 0 0 0 0 0 1 0	
A	0 0 0 0 0 1 0 1	
P´´	0 0 0 0 0 1 0 1 1 0	

IGUALES

División en base B

A diferencia de la suma, resta y multiplicación, la división, en general, no tiene un resultado con un número de dígitos finito.

- El número de ciclos en el algoritmo de la división depende de la precisión deseada, no del ancho de los operandos.

Clases de algoritmos de división:

1. Por iteración de dígitos.
   - División con restauración (lápiz y papel)
   - División sin restauración
2. Iteración funcional.
   - *Newton-Raphson*
3. *Very high radix*.
4. Latencia variable.

División en base B de números naturales

Sean

$$X = x_{n-1},x_{n-2},\ldots,x_1,x_0 \qquad x_i, y_i \in \{0,1,\ldots,B-1\}$$
$$Y = y_{n-1},y_{n-2},\ldots,y_1,y_0$$

Se nota el cociente como

$$Q = 0{\cdot}q_1,q_2,\ldots q_m = X/Y \qquad q_i \in \{0,1,\ldots,B-1\}$$

o sea por su representación en base B con precisión de m dígitos.

Si los enteros r_{i-1} y Y satisfacen $r_{i-1}<Y$, entonces existe un par único de enteros q_i y r_i que satisfacen

$$B{\cdot}r_{i-1} = Y{\cdot}q_i + r_i \qquad qi \in \{0,1,\ldots,B-1\} \qquad con\ 0 \le r_i \le Y-1.$$

Como consecuencia; y asumiendo que $r_0 = X$, la aplicación recursiva de la ecuación anterior procura

$$X = Y\,(q_1{\cdot}B^{-1} + q_2{\cdot}B^{-2} + \ldots + q_mB^{-m}) + r_m\,{\cdot}B^{-m}$$

ó

$$Q = X/Y = 0{\cdot}q_1,q_2,\ldots q_m + (r_m\,{\cdot}B^{-m})/Y$$

¿Qué pasa si X > Y?

El algoritmo general se aplica cuando X < Y. Si eso no se cumple, se requiere de un paso de alineación previo:

1. Hallar s tal que $X < B^s$
2. Sustituír Y por $Y'=B^sY$
3. Dividir X por Y' con una presición de m+s dígitos

   $B^{m+s}X = Y' \cdot Q + r' \qquad con\ r' < Y'$

   Entonces $B^mX = Y \cdot Q + r'/B^s \ con\ r'/B^s < Y$

   $B^mX = Y \cdot Q + r$
4. Para hallar r hubo que hacer $r = r'/B^s$

¿Cómo se aplica el algoritmo cuando B=2?

Si B=2, el teorema fundamental de la división $B \cdot r_{i-1} = Y \cdot q_i + r_i$

queda: $2 \cdot r_{i-1} = Y \cdot q_i + r_i \quad con\ r_i < Y$

q_i naturalmente puede ser 1 ó 0, probamos $q_i = 1$:

$2 \cdot r_{i-1} - Y = r_i$

Si $r_i \geq 0$, entonces acertamos, $q_i = 1$ y tenemos r_i

Si $r_i < 0$, entonces $q_i = 0$ y tenemos $r_i = 2.r_{i-1}$

Como se restaura el resto (y se desplaza), se dice que este es el algoritmo de la división CON restauración.

Ejemplos de división con restauración…

Sea X = 74 y Y = 8 entonces X/Y se calcula:

```
 1001010 | 1000
-1000    |  1001
 0001010 ←
- 1000
  -01010 ←
 0001010
-  1000
   -1010 ←
 0001010
-   1000
 0000010 ←
```

desplazando el divisor a derecha en cada vuelta

Otra solución alternativa

```
   1001010 | 1000
 - 1000    |  1001
   0001010 ←
  0001010 ←
 - 1000
  -01010 ←
 0001010
 - 1000
   -1010 ←
0001010 ←
 - 1000
0000010 ←
```

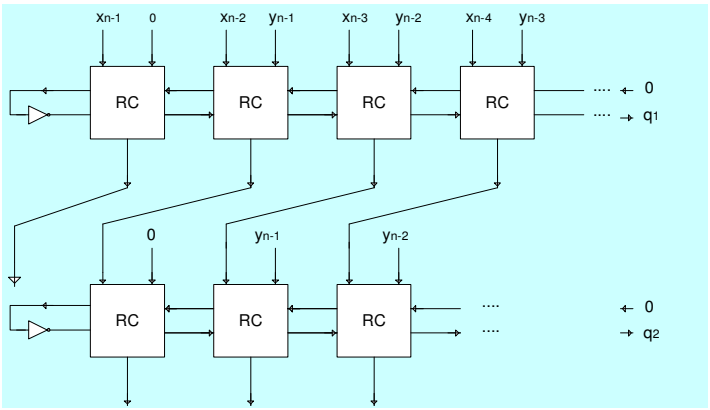
desplazando el dividendo a izquierda en cada vuelta

Algoritmo de división con restauración (Restoring)

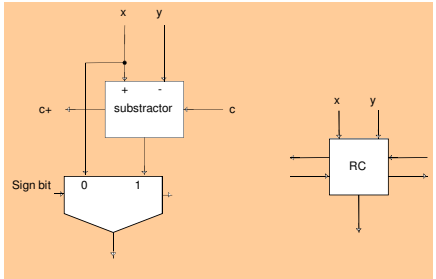
Algoritmo

```
r(0) := X;
For i in 1,...,m loop
    r(i) := r(i-1)-Y;
    if r(i) ≥ 0 then
        q(i) = 1;
    else
        q(i) = 0 ;
        r(i) := r(i-1);
    end if;
     r(i) := r(i) * 2;
endloop;
```

Arreglo de división *Restoring (paralelo)*



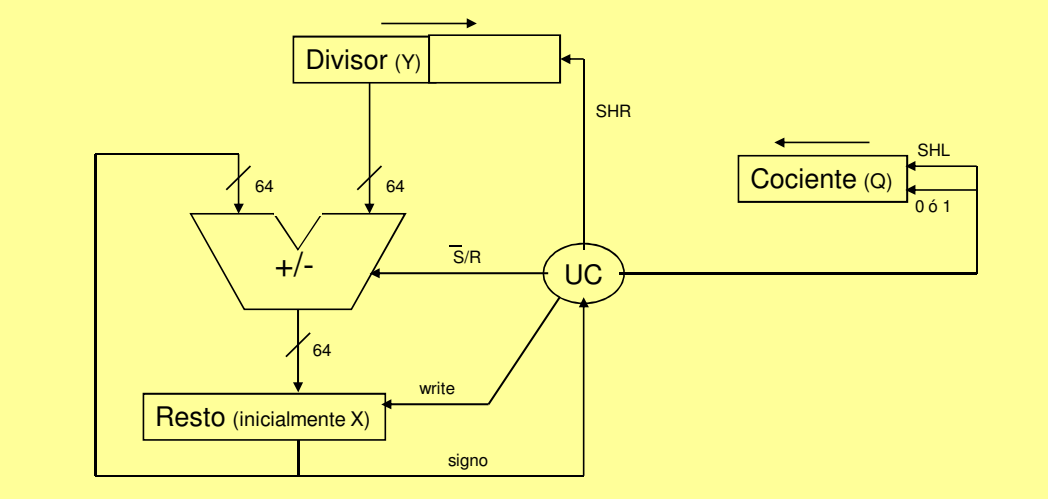
Celda elemental



Algoritmo Restoring - Circuito secuencial (básico)

La Unidad de control (UC):

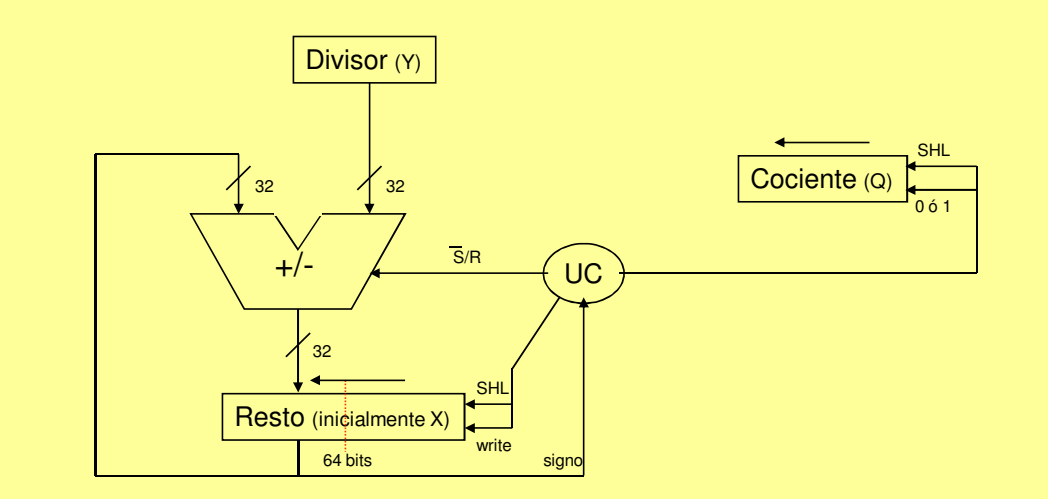
1. resta R – Y
2. si signo es 0 → agrega “1” al cociente y lo desplaza a izquierda
3. si signo es 1 → agrega “0” al cociente y lo desplaza a izquierda
   recupera el resto (R = R+Y)
4. desplaza el divisor (Y) a derecha



Algoritmo Restoring - Circuito secuencial (mejorado)

Funcionalmente igual al diseño anterior pero:

1. El registro Divisor ahora es de 32 bits
2. El sumador es de 32 bits

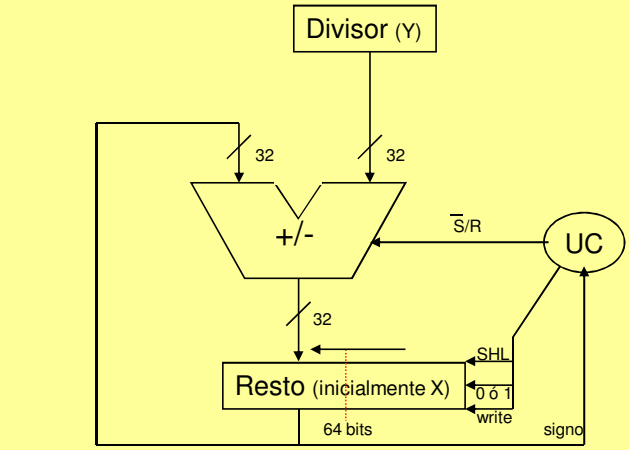


Algoritmo Restoring - Circuito secuencial (optimizado)

Funcionalmente igual al diseño anterior pero:

1. El registro Divisor ahora es de 32 bits
2. El sumador es de 32 bits
3. Los bits parciales q_i se van metiendo en la parte baja del registro "Resto"

Al terminar queda: | Resto | Cociente |



División sin restauración (Non Restoring)

Según el algoritmo de Restoring:

inicialmente: R = X
Y = divisor

Se resta el divisor
R = X – Y

Si R<0 → R = (X - Y) + Y
se desplaza
R = [(X – Y + Y) . 2]
En la siguiente vuelta
R = [(X – Y + Y) . 2] – Y

R = (X - Y) . 2 + Y

Al mismo estado se llega de otra forma:

inicialmente: R = X
Y = divisor

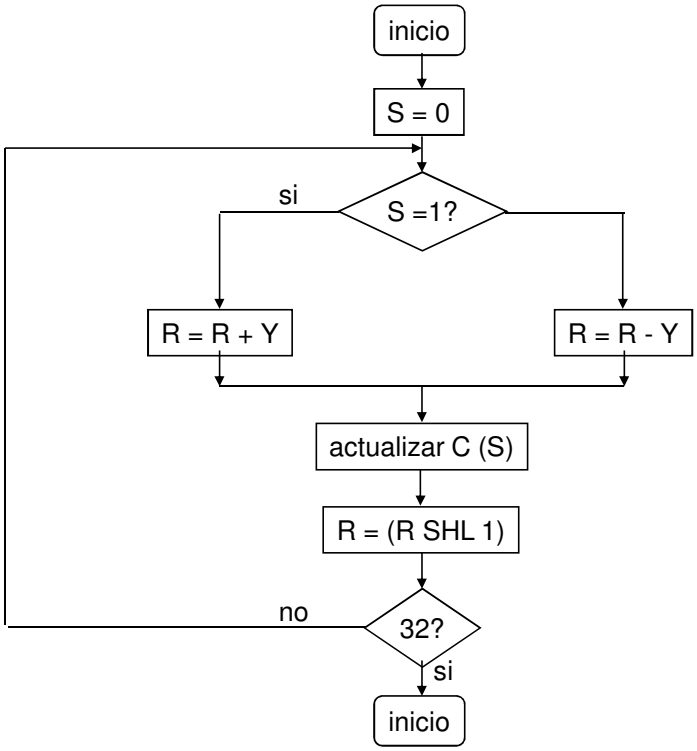
Se resta el divisor
R = X – Y

se desplaza
R = (X - Y) . 2

al ser R<0 en la vuelta anterior
R = [(X – Y) . 2] + Y

El nuevo algoritmo sugiere que si R < 0 en cualquier vuelta no es necesario recuperar el resto anterior sino que basta con sumar Y en la vuelta siguiente (en lugar de restarlo)

División sin restauración (Non Restoring)



Ejemplo Restoring

$X = 74_{(10} = 1001010_{(2}$
$Y = 8_{(10} = 1000_{(2}$

Se expresan ambos valores en la forma: 0,xxxx para asegurar el cociente de forma 0,qqqq (punto fijo)

X = 00.1001010
Y = 00,1000
-Y = 11,1000

(En lugar de restar Y, sumo -Y)

Paso	Operaciones	cociente	Comentarios
1)	00.1001010		R0 = X
	11.1000000		sumo –Y
	00,0001010 >0	q = **1,**	mayor que 0 (no restauro)
2)	00,0010100		R1 = (R0 + -Y) << 1
	11.1000000		sumo –Y
	11,1010100 <0	q = 1,**0**	menor que 0 (restauro)
3)	00,0101000		R2 = R1 << 1
	11.1000000		sumo –Y
	11,1101000 <0	q = 1,0**0**	menor que 0 (restauro)
4)	00,1010000		R3 = R2 << 1
	11.1000000		sumo –Y
	00,0010000 >0	q = 1,00**1**	fin del cálculo

Ejemplo Non Restoring

$X = 74_{(10} = 1001010_{(2}$
$Y = 8_{(10} = 1000_{(2}$

Se expresan ambos valores en la forma: 0,xxxx para asegurar el cociente de forma 0,qqqq (punto fijo)

X = 00.1001010
Y = 00,1000
-Y = 11,1000

(En lugar de restar Y, sumo -Y)

Notar que, puesto que NO se cumple que X<Y, entonces el cociente es de la forma 1,XXX

Paso	Operaciones	cociente	Comentarios
1)	00.1001010		R0 = X
	11.1000000		sumo –Y
	00,0001010 >0	q = **1**,	mayor que 0 (en 2 resto Y)
2)	00,0010100		R1 = (R0 + -Y) << 1
	11.1000000		sumo –Y
	11,1010100 <0	q = 1,**0**	menor que 0 (en 3 sumo Y)
3)	11,0101000		R2 = (R1 + -Y) << 1
	00.1000000		sumo +Y
	11,1101000 <0	q = 1,0**0**	menor que 0 (en 4 sumo Y)
4)	11,1010000		R3 = (R2 + -Y) << 1
	00.1000000		sumo +Y
	00,0010000 >0	q = 1,00**1**	fin del cálculo

División por el método de Newton-Raphson

D = d.q + r

Si r (resto) = 0 → q = D / d = D * 1/d

Newton-Raphson calcula primero el recíproco de ´d´

x = 1/d → 1/x = d → 1/x – d = 0

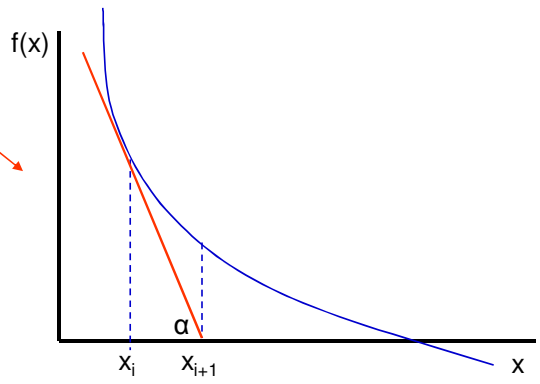
→ **f(x) = 1/x - d = 0**

Para solucionar f(x) se puede hacer

$tg(\alpha) = op/adj = f(x_i) / (x_i - x_{i+1}) = f'(x_i)$

Entonces

$x_{i+1} = xi - f(x_i) / f'(x_i)$



División por el método de Newton-Raphson

Sea *x0 (≠0) la primera aproximación al resultado, entonces*

$f(x0) = 1/x0 - d$ y $[df/dx]_{x=0} = -1/x_0^2$

entonces

$(1/x_0 - d)' = (1/x_0 - d) / (x_0 - x1)$

$-1/x_0^2 = (1/x_0 - d) / (x_0 - x1)$

$-((x_0 - x_1) / x_0^2) = 1/x_0 - d$

$x_1 - x_0 = x_0^2 (1/x_0 - d)$

$x_1 = x_0 + x_0 - d.x_0^2$

$x_1 = 2 x_0 - d.x_0^2$

$x_1 = x_0 (2 - d.x_0)$

y

$x_2 = x_1(2 - d.x_1)$

…

$$x_{i+1} = x_i(2 - d.x_i) \qquad \text{con } 1/B <= d < 1.$$

División por el método de Newton-Raphson

Algoritmo de cálculo de recíproco

```
x(0) := LUT (d);
For i in 0,...,k-1 loop
    x(i+1) := x(i)*(2–d*x(i))
endloop;
```

Ejemplo

Sea

d = 0.161828 (base 10)

siendo 32 la precisión deseada para 1/d.

Se supone disponible una tabla LUT de 4 dígitos de precisión.

x0 = LUT (d) = **6.179**

d · x0 = 0.161828 · 6.179 = 0.99993521

x1 = 6.179 · (2 - 0.99993521) = **6.1794003**

d · x1 = 0.161828 · 6.1794003 = 0.9999999917484

x2 = 6.1794003 · (2 - 0.9999999917484) = **6.179400350989939**

d · x2 = 0.161828 · 6.179400350989939 = 0.99999999999999848492

x3 = 6.179400350989939 · (2 - 0.99999999999999848492)

= **6.17940035098993993622858837777837**

Cálculo de valores fuera de rango (en base b)

Puesto que *d debe estar* en el rango (normalizado) $1/b \leq d < 1$

si d > 1 entonces se divide por b^n de manera que quede un d' = 0,d

$$d' = d/b^n$$

entonces

$$x' = 1/d' = 1/(d/b^n) = b^n * 1/d = b^n * x$$

Ejemplo: calcular el recíproco de d = 5 en base 2:

$$d = 101_{(2}$$

como d está fuera del rango, entonces

$$d' = d/2^3 = 101/1000 = 0,101$$

entonces

$$x' = 1/d' = 1/0,101_{(2} = 1000_{(2}/101_{(2} = 8_{(10}/5_{(10} = \mathbf{1,6}$$

Ahora,

$$1,6_{(10} \approx 1,10011_{(2}$$

Para d' = 0,101 , x' = 1/d' = 1,10011

pero

$x' = 2^3/d$ entonces

$x'/2^3 = 1/d = x$

$x = 1,10011_{(2}/1000_{(2} = 0,00110011_{(2} \approx 0,2_{(10} = 1/d = 1/5_{(10}$