Unidades Aritméticas– Lógicas (ALU)

Capítulo 1. Sistemas de Numeración

1. Binario puro sin signo

Sea el vector binario

$$\vec{X} = x_{n-1}, x_{n-2}, \ldots, x_1, x_0 \qquad x_i \in \{0,1\} \tag{1.1}$$

el valor numérico de *x* se calcula como

$$|\vec{X}| = \sum_{i=0}^{n-1} x_i.2^i \tag{1.2}$$

Generalizando en base B ($x_i \in \{0,1,...,B-1\}$), se calcula el valor de *x* como

$$|\vec{X}| = \sum_{i=0}^{n-1} x_i.B^i \tag{1.3}$$

donde B^i es el peso asociado con el bit x_i.

2. Binario con signo valor absoluto

En esta representación el bit x_{n-1} en (1.1) se interpreta como bit de signo sin peso asociado:

$$|\vec{X}| = (-1)^{x_{n-1}}. \sum_{i=0}^{n-2} x_i \tag{1.4}$$

3. Binario complemento a uno

En esta representación el bit x_{n-1}, también llamado bit de signo, tiene peso negativo ($1-2^{n-1}$).

$$|\vec{X}| = x_{n-1}(1-2^{n-1}) + \sum_{i=0}^{n-2} x_i 2^i \tag{1.5}$$

El rango de definición de *X* es: $[x_{n-1}(1-2^{n-1}),2^{n-1}-1]$

Propiedades

(i) El cero tiene dos representaciones

$$0 = 1\ 1 \ldots 1\ 1 = 0\ 0 \ldots 0\ 0 \tag{1.6}$$

Demostración.
Aplicando la fórmula (1.5) al vector $\vec{0}$ es trivial; aplicando (1.5) al vector $\vec{1}$:

$$|\overrightarrow{X}| = -(2^n-1)+(2^n-1) = 0$$

(ii) Cambio de signo

$$-\overrightarrow{X} = \overrightarrow{\overrightarrow{X}} \tag{1.7}$$

Demostración:

$$-\overrightarrow{X} = \vec{0} - \overrightarrow{X}, \tag{1.8}$$

aplicando (1.6), $0 = \vec{1}$ permite escribir en forma vectorial (operación bit por bit)

$$-\overrightarrow{X} = (1, 1, ..., 1, 1) - (x_{n-1}, x_{n-2}, ..., x_1, x_0) \tag{1.9}$$

aplicando la relación:

$$1 - x_i = \overline{x_i} \tag{1.10}$$

se completa la demostración.

Un circuito de cambio de signo está presentado en fig. 1, en donde *CS=1* controla el cambio de signo.

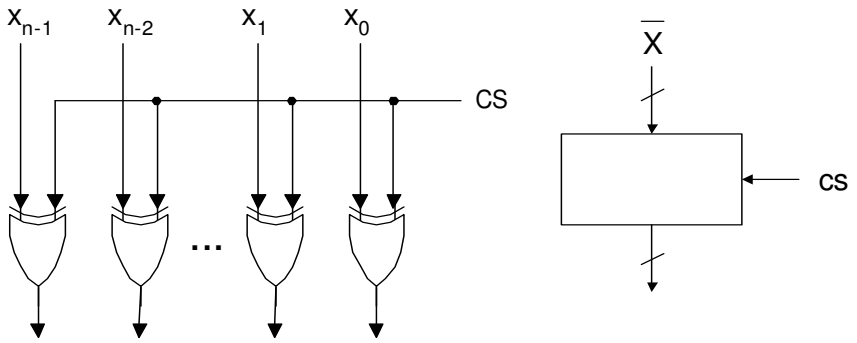


Fig. 1. Cambio de signo en representación complemento a uno

4. Binario Complemento a dos

Esta representación es similar a la precedente con una diferencia en el peso del bit de signo: (-2^{n-1}).
Entonces:

$$|\vec{X}| = x_{n-1}(-2^{n-1}) + \sum_{i=0}^{n-2} x_i 2^i \qquad (1.11)$$

El rango de definición es ahora $[-2^{n-1}, 2^{n-1}-1]$ con una sola representación para cero:

$$0 = \vec{0} = 0\ 0\ \ldots\ 0\ 0 \qquad (1.12)$$

Propiedades

(i) $|\vec{1}| = -1$ (1.13)

Demostración.
Aplicando (1.11):

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

Corolario:

$$|\vec{0}| = |\vec{1}| + 1 \qquad (1.14)$$

(ii) Cambio de signo

$$-\vec{X} = \vec{\vec{X}} + 1 \qquad (1.15)$$

Demostración:

$$-\vec{X} = \vec{0} - \vec{X},$$

Aplicando (1.14)

$$-\vec{X} = |\vec{1}| + 1 - \vec{X} = |\vec{1}| - \vec{X} + 1$$

Aplicando (1.10)

$$-\vec{X} = \vec{\vec{X}} + 1$$

Operador de cambio de signo.
La operación caracterizada por la fórmula 1.15 se puede sustituir por la complementación de todos los bits a la izquierda del último bit 1 a la derecha.

Ejemplo:

$X = 1011100 = (-36)_{10}$
$-X = 0100100 = (36)_{10}$

La fig. 2 representa un circuito que detecta el primer uno y genera las complementaciones de los bits ulteriores.

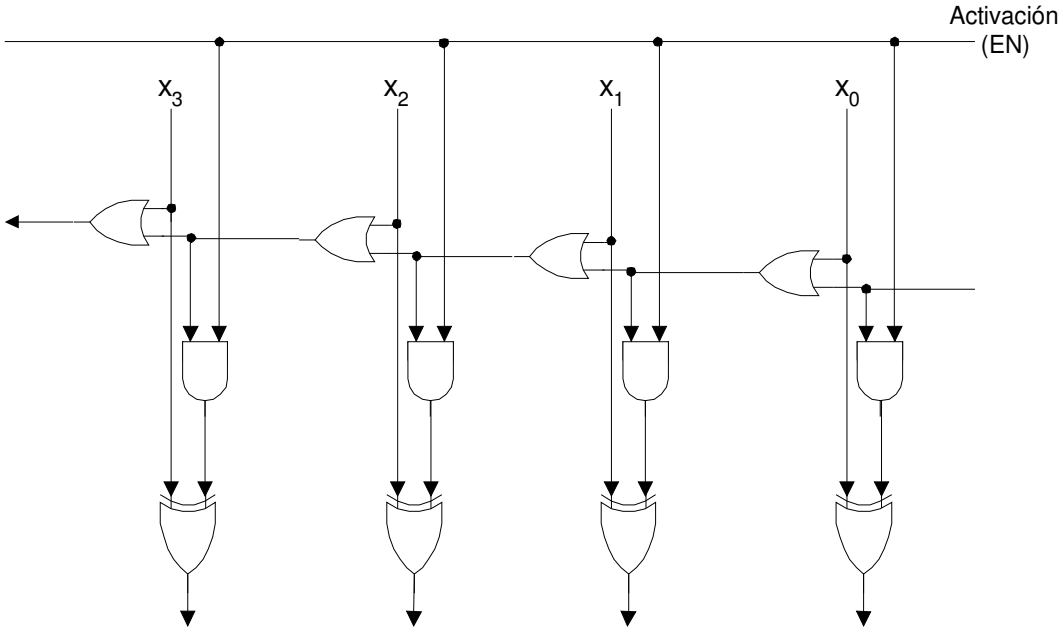


Fig. 2. Cambio de signo en representación complemento a dos

5. Binario cero desplazado

En esta representación se aplica el valor cero a un valor $\vec{Z}$ tal que:

$$0 < |\vec{Z}| < 2^{n-1} \tag{1.16}$$

Calculando $|\vec{X}|_z$ como:

$$|\vec{X}|_z = |\vec{X}| - |\vec{Z}| \tag{1.17}$$

Ejemplo: $n = 4;\ \vec{Z} = 1000$

$$1100_z = 1100 - 1000 = +\ 0100$$
$$0011_z = 0011 - 1000 = -\ 0101$$
$$1000_z = 1000 - 1000 = \ \ 0000$$

6. Representación Binario normalizado, punto flotante estándar IEEE P754

La representación de punto flotante normalizado representa los números en dos partes.
(i) Mantisa: Contiene los dígitos significativos del número, asumiendo que el bit de mayor peso es siempre uno y tiene peso 2^{-1}, o sea:

$$M_x = 0.\ x_{-1}\, x_{-2} \ldots x_{-k} = 0.\ 1x_{-2}\, x_{-3} \ldots x_{-k} \tag{1.18}$$

(ii) Exponente o factor de escala: potencia de la base *2*.
Entonces:

$$X = m_x \cdot 2^{e_x} \qquad (1.19)$$

Ejemplo *1*: $X = 10110 = 0.\ 10110 \cdot 2^{101}$

El uso de exponentes y mantisas con signo permite representar números fraccionarios con signo.

Ejemplo *2*: $X = -\ 0.\ 00101 = -\ 0.\ 101.\ 2^{-010}$

En el estándar IEEE P754 (fig. 3), la mantisa representa con *23* bits (*52* de doble precisión) y signo puro (*1* bit), mientras que el exponente usa *8* bits (*11* en doble precisión) y signo por cero desplazado (2^7-1).
Siendo los dos primeros bits de la mantisa siempre iguales a 0.1, no se almacenan: son bits implícitos.

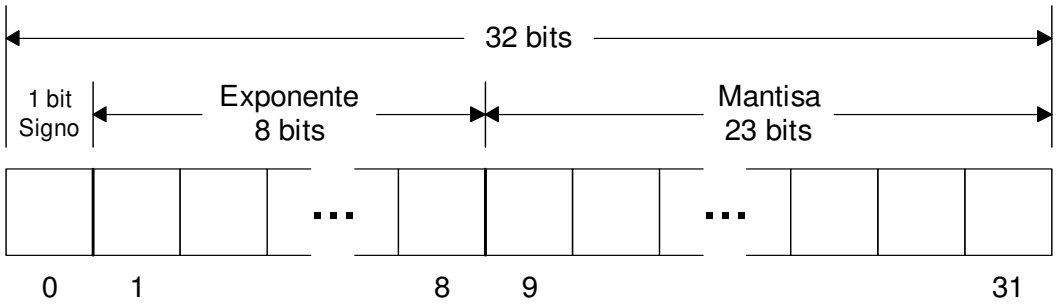


Fig. 3. Simple precisión IEEE P754

Ejemplo 4: X = 1 10101010 10100000000000000111111

$$= -\ \overbrace{0.1}^{\text{Bits implícitos}}10100000000000000111111 \cdot 2^{00101011}$$

Capítulo 2. Sumadores

1. Sumadores en binario sin signo

1.1. Sumador completo

El sumador binario más clásico (Ripple-Carry Adder) utiliza en forma recursiva (en el tiempo o en el espacio) un sumador de dos bits x_i, y_i más acarreo previo c_{i-1} que proporciona dos bits de resultados: la suma y el acarreo saliente c_i .Es el sumador completo (Full-Adder; FA: fig. 4). Cumple las siguientes funciones:

$$\begin{cases} r_i = x_i \oplus y_i \oplus c_{i-1} \\ c_i = M(x_i, y_i, c_{i-1}) \end{cases} \tag{2.1}$$

donde M es la función mayoría que también se puede expresar como:

$$M(x_i, y_i, c_{i-1}) = x_i y_i \ \vee \ x_i c_{i-1} \ \vee \ y_i c_{i-1} \tag{2.2}$$

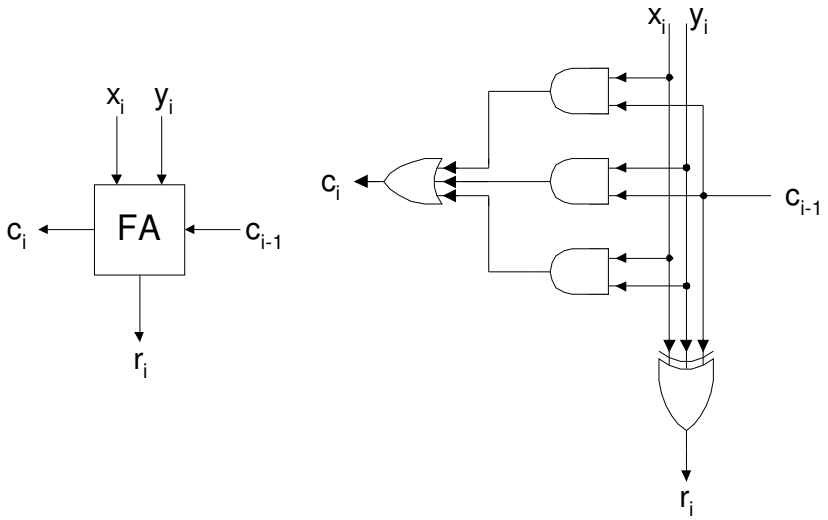


Fig. 4. Sumador Completo

Definiciones

(i) Función de generación de acarreo G (generador)

$$G_i = x_i \, . \, y_i \tag{2.3}$$

$G_i = 1$ genera un carry c_i independientemente del valor del carry entrante c_{i-1}.

(ii) Función de propagación de acarreo (propagador)

$$P_i = x_i \oplus y_i \tag{2.4}$$

$P_i = 1$ $(x_i \neq y_i)$ cumple en transmitir c_{i-1} a la salida c_i, o sea:

$$P_i = 1 \Rightarrow c_i = c_{i-1} \tag{2.5}$$

(iii) Función de aniquilación de acarreo (carry-kill)

$$K_i = NOR\ (x_i, y_i) = NOT\ (x_i \ v \ y_i) \tag{2.6}$$

$K_i = 1$ $(x_i = y_i = 0)$ cumple en generar un acarreo $c_i = 0$ independientemente del acarreo entrante c_{i-1}.

Con estas definiciones el acarreo saliente se puede expresar como:

$$c_i = G_i \ v \ P_i \cdot c_{i-1} \tag{2.7}$$

lo que sugiere la representación del sumador completo de la fig. 5.

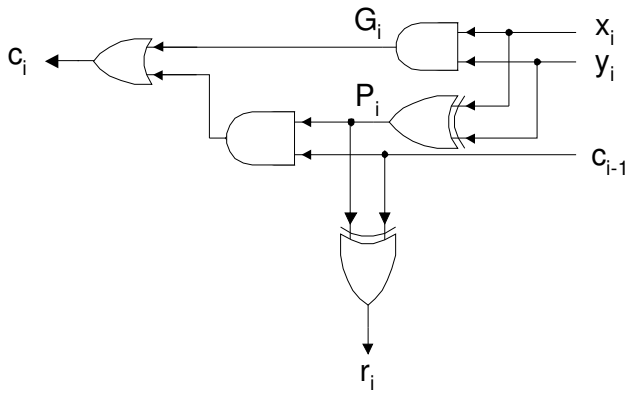


Fig. 5. Sumador completo con eslabón de cadena de acarreo

El uso recursivo de (2.7) genera un circuito llamado cadena de acarreo (carry chain).

Comentario
Si bien la demora del circuito de acarreo de la fig. 5 es superior que lo que sugiere la fig. 4, la utilización recursiva del circuito de la fig. 5 no implica más que una demora de dos compuertas por nivel adicional ya que los P_i's (como los G_i's) se pueden calcular en paralelo.

1.2. Sumador paralelo con propagación de acarreo (Ripple-carry adder)

El sumador paralelo se puede realizar por iteración en el espacio del sumador completo, fig. 6.

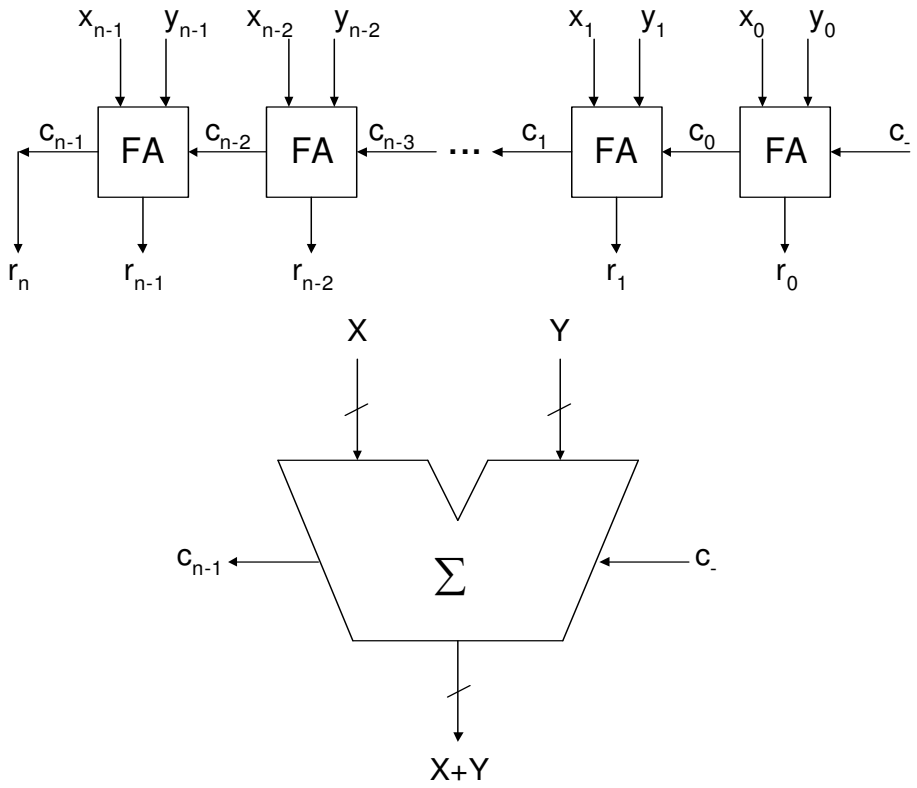


Fig. 6. Sumador paralelo

2. Sumadores en binario con signo

2.1. Sumador paralelo en complemento a dos

Sean *X,Y* dos números binarios expresados en complemento a dos, la suma *R=X+Y* se puede ilustrar por el siguiente esquema:

$$\begin{array}{cccccccc} c_{n-1} & c_{n-2} & c_{n-3} & \ldots & c_1 & c_0 & c_{-1} \\ & x_{n-1} & x_{n-2} & \ldots & & x_1 & x_0 \\ & y_{n-1} & y_{n-2} & \ldots & & y_1 & y_0 \\ \hline & r_{n-1} & r_{n-2} & \ldots\ldots & & r_1 & r_0 \end{array} \qquad (2.8)$$

Comentarios

- Los bits indexados $i \in \{0,1,\ldots,n-2\}$ tienen pesos 2^i, con lo cual sumadores completos (FA) clásicos se pueden usar para esta parte.
- Los bits x_{n-1}, y_{n-1} tienen peso negativo -2^{n-1} mientras que el acarreo eventual c_{n-2} tendrá peso $+2^{n-1}$ con lo cual un tratamiento especial tiene que dedicarse al nivel *n-1*.
- Se usa un sumador completo común para procesar x_{n-1}, y_{n-1} y c_{n-2}, generando un acarreo c_{n-1} que no se interpretará como un dígito más, sino como una información acerca de la cantidad de "*1*" en el nivel *n-1*.
- Del análisis de x_{n-1}, y_{n-1}, c_{n-2} y c_{n-1} resultarán las decisiones pertinentes acerca del overflow eventual y del resultado de la suma: tabla 2.1.

En la tabla 2.1, las celdas tachadas corresponden a situaciones inaccesibles. Por ejemplo, es imposible generar $c_{n-1} = 1$, sumando ceros (fila 3, columna 1). Las conclusiones resultan de las siguientes interpretaciones:

$c_{n-2} = 1$ Corresponde a un acarreo positivo de peso $+2^{n-1}$ a compensar por el valor -2^{n-1} de un bit de signo. Si no se compensa hay overflow positivo.

$c_{n-1} = 1$ Informa de la presentación de dos o más bits $\neq 0$ en el nivel $n-1$. Uno de estos dos o tres bits "*1*" tiene que asociarse con $c_{n-2} = 1$ y se compensará con un bit de signo (x_{n-1} ó y_{n-1}). Si no, hay overflow negativo.

Filas ↓ / Columnas →	c_{n-1}	c_{n-2}	1 $x_{n-1} = y_{n-1} = 0$	2 $x_{n-1} = y_{n-1} = 1$	3 $x_{n-1} \neq y_{n-1}$ $(x_{n-1} \oplus y_{n-1} = 1)$
1	**0**	**0**	$X, Y > 0$ $R > 0$		$r_{n-1} = 1$ $R < 0$
2	**0**	**1**	Overflow positivo		
3	**1**	**0**		Overflow negativo	
4	**1**	**1**		$r_{n-1} = 1$ $R < 0$	$r_{n-1} = 0$ $R > 0$

Tabla 2.1.

Regla práctica
Del análisis de la tabla 2.1, resulta que la detección del overflow se expresará como

$$OV = c_{n-2} \oplus c_{n-1} \tag{2.9}$$

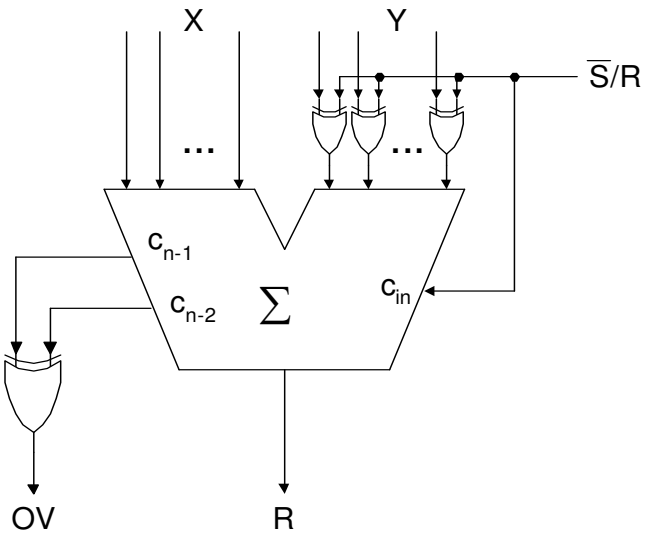


Fig. 7. Sumador restador en complemento a dos

La fig. 7 ilustra un sumador restador usando un sumador paralelo y un cambiador de signo correspondiente a la ecuación (1.15).

2.2. Sumador paralelo en complemento a uno

El razonamiento de la sección 2.1. se aplica de manera similar para sumas en complemento a uno, con una diferencia generada por el peso del bit de signo:

$-(2^{n-1}-1)$.

En caso de compensar con $c_{n-2} = 1$, de peso $+2^{n-1}$, una diferencia de $+1$ se tomará en cuenta. Para saldar esta diferencia se sumará 1, cada vez que $c_{n-1} = 1$.

En realidad este caso ocurrirá cada vez que $c_{n-1} = c_{n-2} = 1$, con lo cual parece indiferente de considerar c_{n-1} ó c_{n-2}.

Una situación aparente de overflow tal como $R = 0111...1$ con $c_{n-1} = 1$, o sea

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

se levantará observando que $-(2^{n-1}-1)$ se compensa exactamente por R. El ejemplo a continuación ilustra este caso, en donde $c_{n-2} = 1$ aparece después de sumar el 1 generado por $c_{n-1} = 1$, fig. 8.

$1 = c_{n-2}$ después de la corrección

$c_{n-1} = 1$ 0

v 1 0 1 0 0

v

v 1 1 0 1 1

v _____

v 0 1 1 1 1

v → → → → → + 1

_____

1 0 0 0 0

Fig. 8.

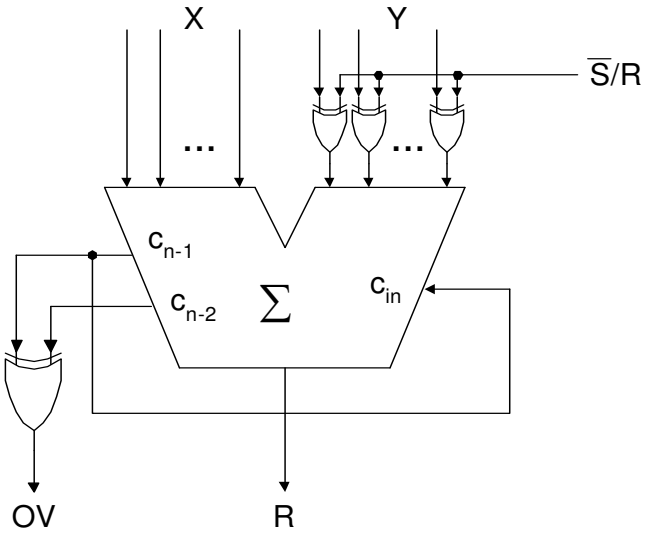


Fig. 9 – Sumador restador en complemento a uno

La fig. 9 ilustra el sumador restador paralelo en complemento a uno

3. Sumador con cálculo anticipado de acarreo (Carry look-ahead adder-CLA)

Se aplica de manera recursiva la ecuación (2.7)

$$\begin{cases} C_0 = G_0 \ v \ P_0 \, c_- \\ \dots \\ C_i = G_i \ v \ P_i \, c_{i-1} \end{cases} \qquad (2.10)$$

reemplazando c_{i-1} en las expresiones sucesivas de c_i por su correspondiente de la fórmula anterior.
Las siguientes expresiones se establecen en forma recursiva:

$$\begin{cases} c_0 = G_0 \ v \ P_0 \, c_- \\ c_1 = G_1 \ v \ P_1 \, G_0 \ v \ P_1 \, P_0 \, c_- \\ c_2 = G_2 \ v \ P_2 \, G_1 \ v \ P_2 \, P_1 \, G_0 \ v \ P_2 \, P_1 \, P_0 \, c_- \\ \dots \\ c_k = G_k \ v \ P_k \, G_{k-1} \ v \ P_k \, P_{k-1} \, G_{k-2} \ v \dots v \ P_k \, P_{k-1} \dots P_1 \, P_0 \, c_- \end{cases} \qquad (2.11)$$

Dentro de los límites de fan-in de compuertas, los c_k's se pueden calcular en 3 niveles lógicos, o sea:

- Cálculo de G_i's, P_i's
- Cálculo de productos (AND)
- Suma (OR) de los productos

El resultado final se calculará como

$$r_i = P_i \oplus c_{i-1} \qquad (2.12)$$

Un sumador CLA de 4 bits se representa en la fig. 10.

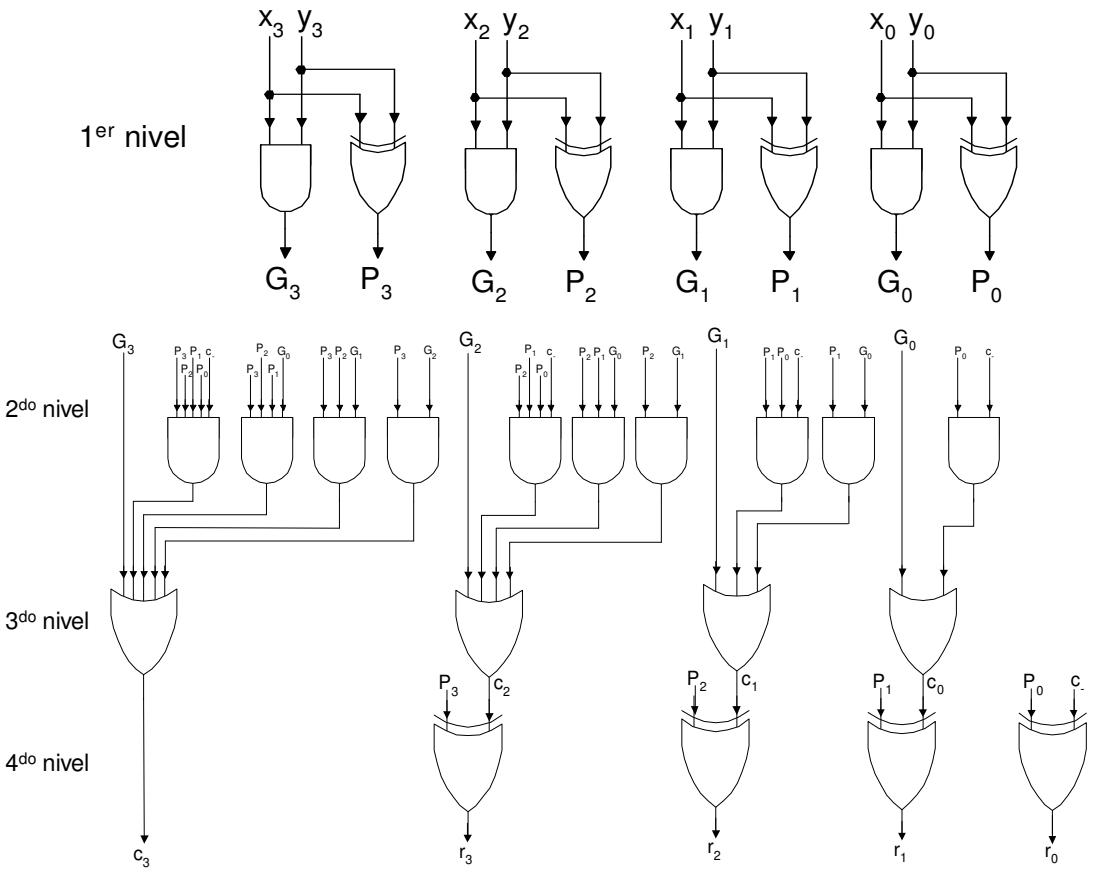


Fig. 10 – Sumador CLA de 4 bits

4. Sumador con salto de acarreo (Carry–skip adder)

Sea *n* la cantidad de bits a sumar.
Se descompone la cadena de acarreos en *n/s* bloques de *s* bits cada uno.
Considerando que la celda elemental de la cadena de acarreos calcula:

$$c_i = G_i \ v \ P_i \, c_{i-1}$$

Siendo $c_{(k+1)s-1}$ el acarreo de salida del grupo *k* (*k = 0, 1, ..., (n/s)-1*) la ecuación del acarreo saliente del bloque *k* se escribe

$$\boldsymbol{C}_k = c_{(k-1)s-1} . \overline{P}_k \ v \ \boldsymbol{P}_k . \boldsymbol{C}_{k-1} \qquad (2.13)$$

con

$$\boldsymbol{P}_k = P_{ks} \, . \, P_{ks+1} \, . \, \ldots \, P_{(k+1)s-1} \qquad (2.14)$$

Si $\boldsymbol{P}_k = 1$, entonces el tiempo de cálculo de $\boldsymbol{C}_k$ se determinará por el tiempo de $\boldsymbol{C}_{k-1}$ más el tiempo del multiplexor materializando la fórmula (2.13). En caso contrario, o

sea si $P_k = 0$, un acarreo se generará dentro del bloque *k* y el plazo no superará el tiempo de propagación del acarreo dentro de bloque. Con la iteración de este razonamiento, el acarreo $C_{(n/s)-2}$, tendrá un tiempo de cálculo no superior al tiempo de propagación de la cadena de acarreo de un solo bloque más la demora de *n/s –1* multiplexores. Las fig. 11 y 12 presentan un bloque y el esquema global del sumador respectivamente.

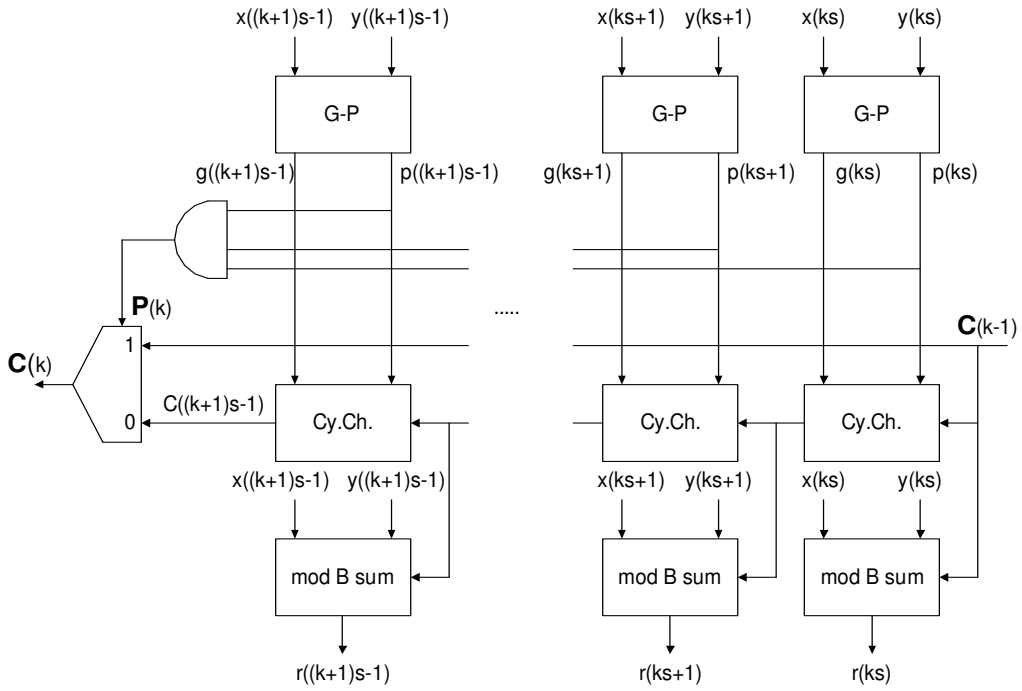


Fig. 11. Bloque k del sumador de salto de acarreo

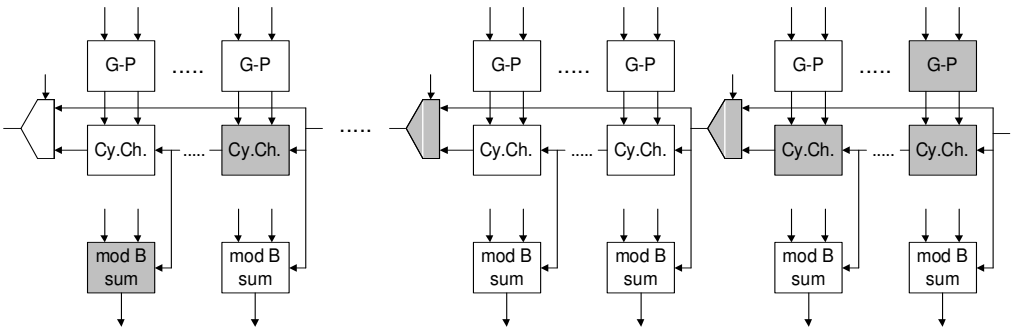


Fig. 12. Sumador de salto de acarreo (camino crítico resaltado)

Capitulo3. Multiplicación

1. Introducción

Los algoritmos mas populares para multiplicar dos números de n dígitos en base B son los algoritmos de desplazamiento y suma (shift and add algorithms). Proceden de dos fases:

(i) productos parciales dígitos por dígitos (n^2).
(ii) suma de n términos.

Las características de costo y velocidad son factores claves para determinar compromisos entre esquemas combinatorios paralelos y implementaciones secuenciales. Los principales algoritmos, con esquemas de implementación se describen a continuación..

2. Algoritmos de desplazamiento y suma (Shift and Add)

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

dos operandos de n dígitos c/u.

Sea $Z = X \cdot Y$ (3.1)
con $Z = z_{2n-1}, z_{2n-2}, ..., z_1, z_0$

2.1. Algoritmo de Hörner

La ecuación (3.1) se puede escribir como

$$Z = \sum_{i=0}^{n-1} x_i B^i Y \tag{3.2}$$

que se expande como

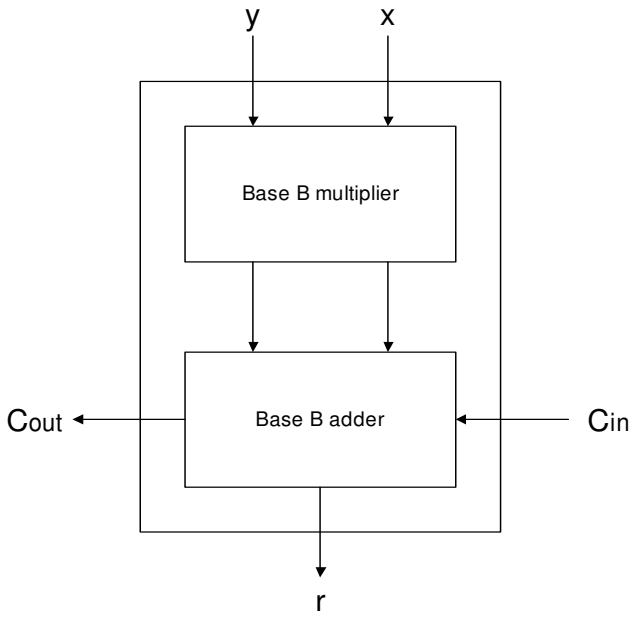
$$Z = ((...(0 \cdot B + x_{n-1}Y)B + x_{n-2}Y)B + ...)B + x_2 Y)B + x_1 Y)B + x_0 Y \tag{3.3}$$

llamada expansión de Hörner, sugiriendo el siguiente 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
$Z = P(0)$

Una implementación posible de $x_i Y$ se muestra en la fig. 13

(a) celda $(y \cdot x + c)$



(b) Cadena de Propagación de acarreo (Ripple – carry Chaín)

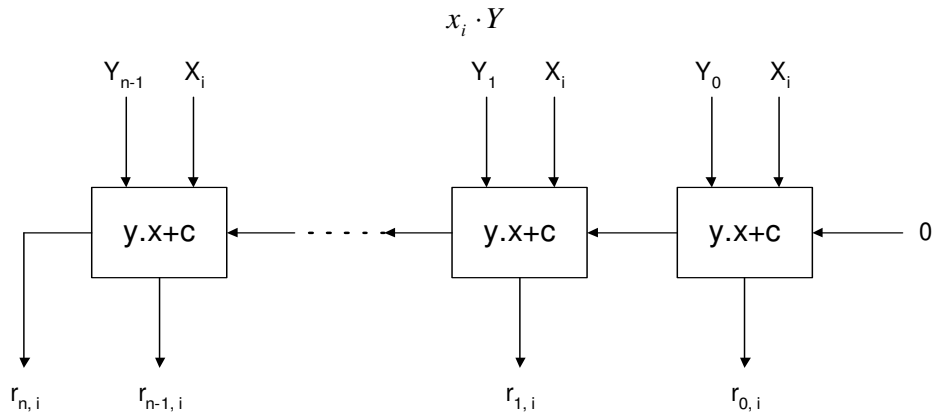


Fig. 13.

El esquema de Hörner se puede implementar en forma secuencial (iteración en el tiempo) por un arreglo combinatorio (iteración en el espacio): fig. 14.

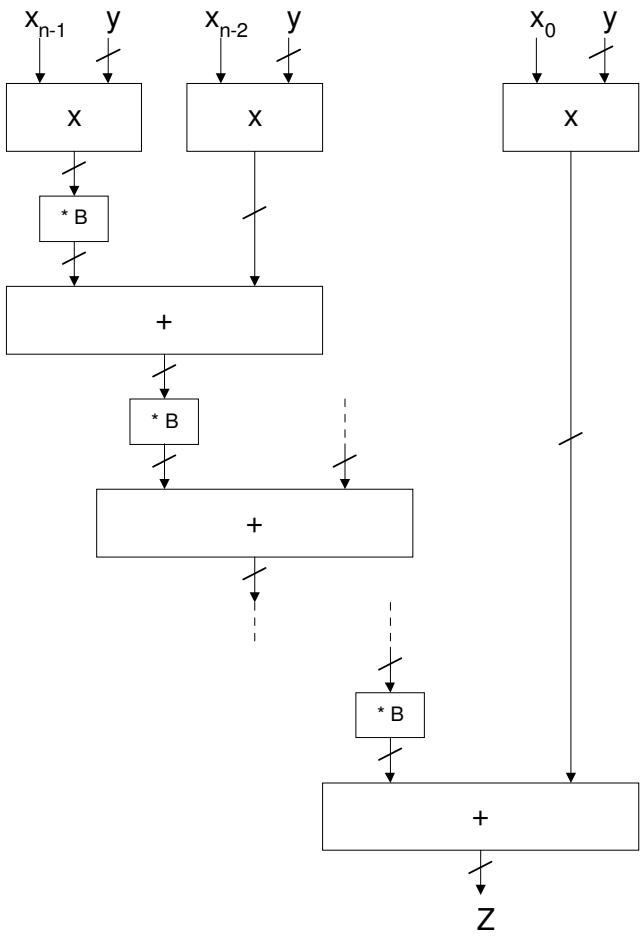


Fig. 14.

Comentario: El esquema de Hörner conviene más para una implementación secuencial, usando el mismo sumador en forma recursiva. Otros circuitos combinatorios (arreglos celulares) con mejores características de costo se estudian más adelante.

2.2. Se puede mostrar que la factorización de derecha a izquierda (3.4) reduce a la mitad la longitud del sumador.

$$\frac{Z}{B^{n-1}} = x_{n-1}Y + B^{-1}(x_{n-2}Y + B^{-1}...+ B^{-1}(x_1Y + B^{-1}(x_0Y + 0))...) \qquad (3.4)$$

3. Multiplicador celular

3.1. Celda básica

Para utilizar la capacidad de la celda de multiplicación y suma, conviene definir la función

$$z = x \cdot y + c + d \qquad x, y, c, d \in \{0, B-1\} \tag{3.5}$$

Se verifica

$$0 \le z \le B^2 - 1$$

con

$$z = (z_1, z_0) \qquad 0 \le z_1, z_0 \le B-1$$

3.2. Multiplicador con propagación de acarreo (Ripple-carry multiplier)

3.2.1. Celda básica del "Ripple-carry"

La celda que implementa la ecuación (5) se muestra en la fig. 15

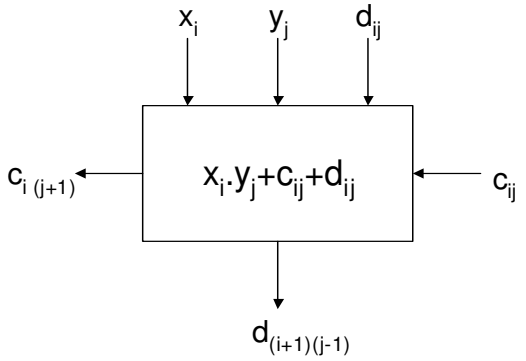


Fig. 15.

La celda de la fig. 15 se representa sintéticamente en la fig. 16 donde las entradas x_i and y_j son implícitas.

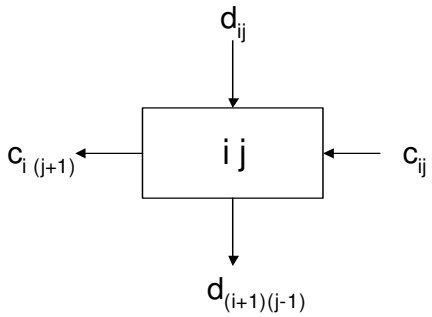


Fig. 16.

3.2.2. Algoritmo "Ripple-carry"

Sean

$$\begin{aligned} X &= (x_{n-1}, x_{n-2}, ..., x_1, x_0) \\ Y &= (y_{n-1}, y_{n-2}, ..., y_1, y_0) \\ C &= (c_{(n-1)(0)}, c_{(n-2)(0)}, ..., c_{10}, c_{00}) \\ D &= (d_{(0)(n-1)}, d_{(0)(n-2)}, ..., d_{01}, d_{00}) \end{aligned} \tag{3.6}$$

en donde

$$x_i, y_i, c_{ij}, d_{ji} \in \{0, B-1\}.$$

Además,

$$c_{in} = d_{(i+1)(n-1)} \qquad \text{for } i = 0,1,...,n-1 \tag{3.7}$$

$$z_{ij} = x_i y_j + c_{ij} + d_{ij} = (c_{(i)(j+1)}, d_{(i+1)(j-1)}) \tag{3.8}$$

for i ***in*** 0,1, …, n-1 ***loop***
  for j ***in*** 0,1, …, n-1 ***loop***
    compute z_{ij}
  end loop
end loop

El resultado es

$$R = \left(d_{n(n-1)}, d_{n(n-2)}, ..., d_{n0}, d_{n(-1)}, ..., d_{2(-1)}, d_{1(-1)}\right) \tag{3.9}$$

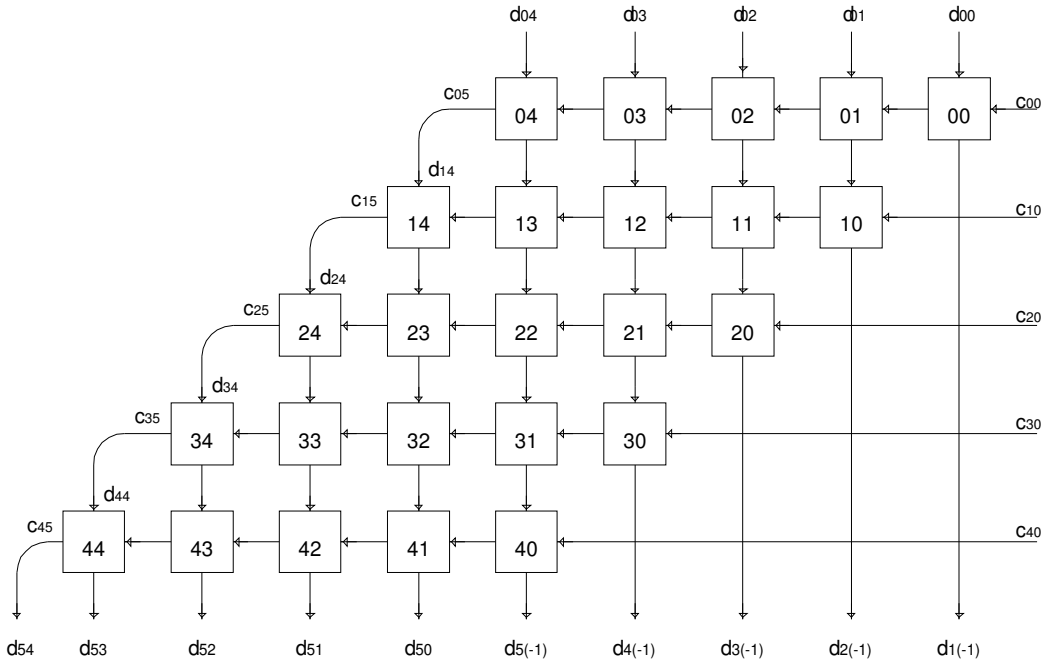


Fig. 17. Multiplicador "Ripple – Carry"

El circuito combinatorio de la fig. 17 materializa la iteración en el espacio del algoritmo, n=5.

3.3. Multiplicador "Carry-save"

3.3.1. Celda básica "Carry-save"

La celda de las fig. 15 y 16 se puede modificar como se muestra en las fig. 18 y 19. Para las celdas de salida se define una versión especial como se muestra en las fig. 20 y 21.

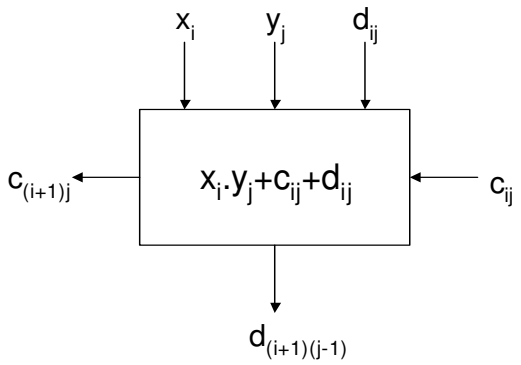


Fig. 18.

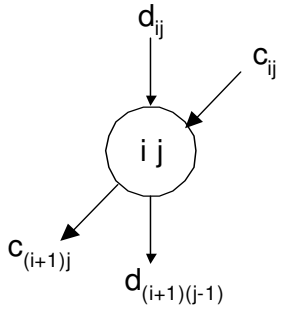


Fig. 19.

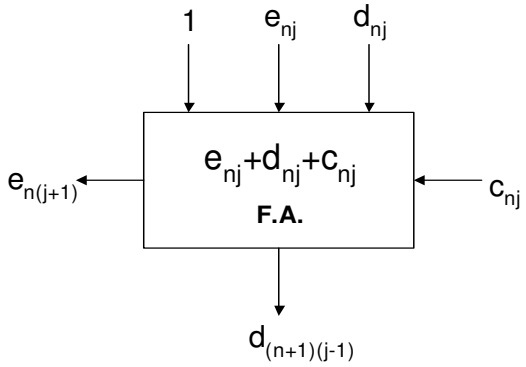


Fig. 20.

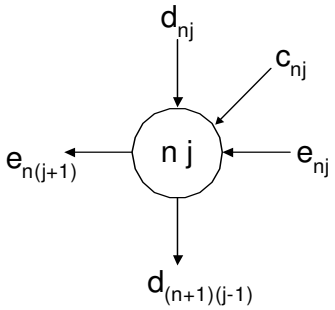


Fig. 21.

3.3.2. Algoritmo "Carry-Save"

Sean

$$\begin{aligned} X &= (x_{n-1}, x_{n-2}, ..., x_1, x_0) \\ Y &= (y_{n-1}, y_{n-2}, ..., y_1, y_0) \\ C &= (c_{(0)(n-1)}, c_{(0)(n-2)}, ..., c_{01}, c_{00}) \\ D &= (d_{(0)(n-1)}, d_{(0)(n-2)}, ..., d_{01}, d_{00}) \end{aligned} \tag{3.10}$$

con

$$x_i, y_i, c_{ij}, d_{ji} \in \{0, B-1\}.$$

Además:

$$z_{ij} = x_i y_j + c_{ij} + d_{ij} = (c_{(i+1)j}, d_{(i+1)(j-1)}) \qquad i<n \tag{3.11}$$

y

$$\begin{aligned} z_{nj} &= e_{nj} + c_{nj} + d_{nj} = (e_{n(j+1)}, d_{(n+1)(j-1)}) \\ e_{n0} &:= 0 \end{aligned} \tag{3.12}$$

for i ***in*** 0,1, …, n ***loop***
  for j ***in*** 0,1, …, n-1 ***loop***
    compute z_{ij}
  end loop
end loop

El resultado es

$$R = \left(d_{(n+1)(n-2)}, d_{(n+1)(n-3)}, ..., d_{(n+1)(-1)}, d_{n(-1)}, d_{(n-1)(-1)}, ..., d_{1(-1)}\right) \tag{3.13}$$

El arreglo combinatorio de la fig. 22 materializa la iteración espacial de este algoritmo.

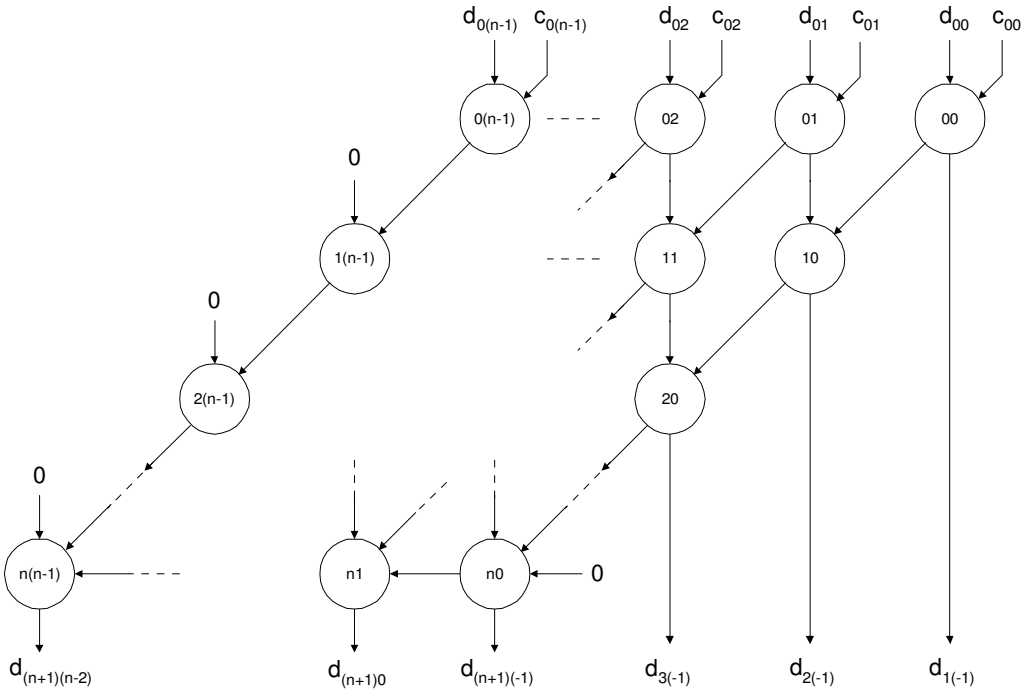


Fig. 22. Carry – Save multiplier

Comentario
(i) De hecho, la computación de z_{nj}, última fila de la fig. 22, corresponde a una suma "Ripple-carry". Para lograr la regularidad y expansibilidad en el circuito, conviene diseñar sumadores a partir de celdas estándares como las de la fig. 18.
(ii) Los circuitos de las fig. 15 y fig. 18 son equivalentes, la diferencia conceptual viene de la re-indexación de las salidas. Este proceso de re-indexación genera la diferencia algorítmica y las variaciones en la estructura de interconexiones.

3.4. Figuras de mérito

Sean T_2 and C_2 las figuras de complejidad de tiempo y material (cantidad de compuertas) respectivamente de las celdas estándares de las fig. 15 or 18. El multiplicador "Ripple-Carry" se caracteriza por figuras de costo siguientes:

$$T_{RCM} = (3n-2)T_2$$
$$C_{RCM} = n^2 C_2 \qquad (3.14)$$

mientras que la implementación "Carry-Save" corresponde a las figuras:

$$T_{CSM} = 2nT_2$$
$$C_{CSM} = n(n+1)C_2 \qquad (3.15)$$

Finalmente, si la etapa de suma de salida (z_{ni}) se implementa usando técnicas de sumas rápidas, las fórmulas (3.15) se pueden mejorar.

4. <u>Multiplicador binario secuencial</u>

El multiplicador secuencial (iteración en el tiempo) utiliza en forma recursiva un único recurso de suma (Σ). Consta de dos registros de n dígitos c/u para el multiplicador y el multiplicando, y de un registro acumulador de doble longitud para el resultado. A cada paso, el multiplicando se suma (o no) al resultado parcial desplazado. Los bits sucesivos del multiplicador controlan la carga del acumulador con la nueva suma (acumulador desplazado + multiplicando). La fig. 23 ilustra el proceso.
Cuando el pulso de reloj baja a cero, se carga el acumulador (y el flag del acarreo) si y solo si el último bit del multiplicando es ***1***. Cuando sube el pulso, se desplaza el acumulador y el multiplicador de una posición a la derecha. Se justifica un desplazamiento circular (SRC) para el multiplicador para recuperar la información de datos al fin del ciclo completo de cálculo.

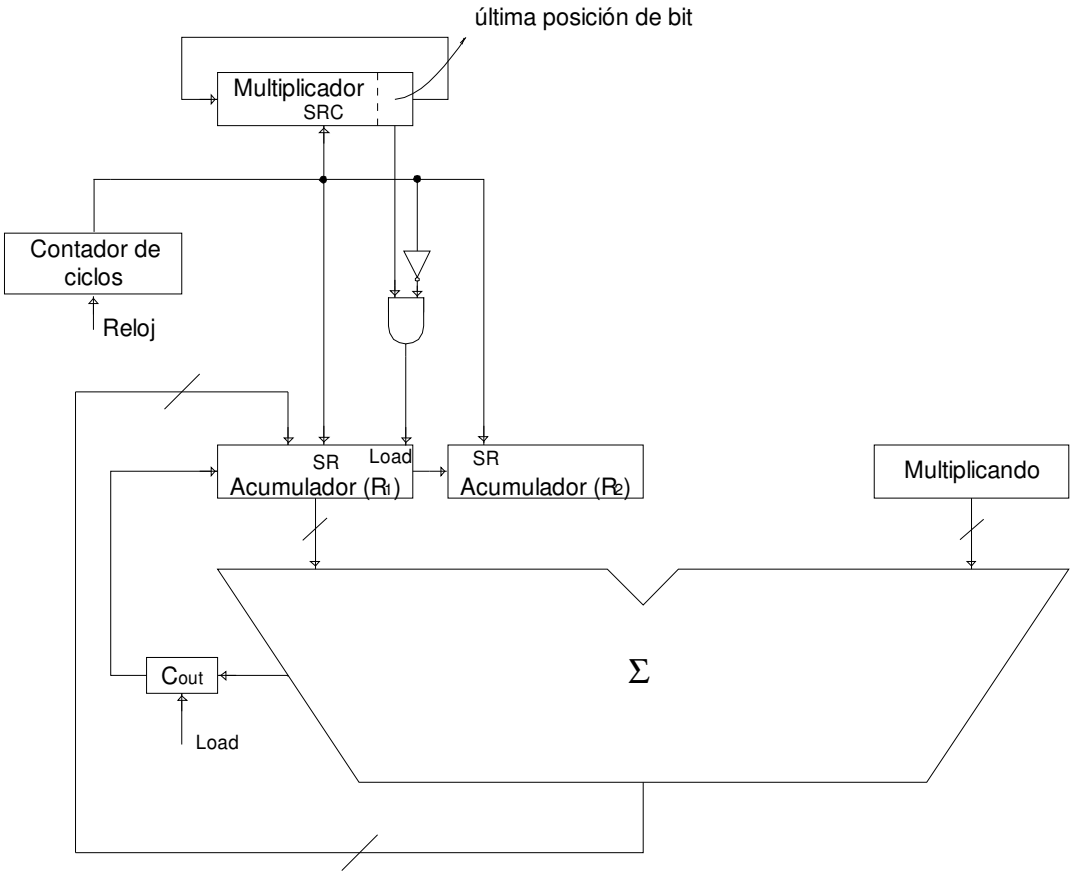


Fig. 23. Multiplicador secuencial por desplazamiento y suma.

Capítulo 4. División

1. Introducción

Los enteros o los números fraccionales de longitud finita pueden multiplicarse entre ellos con total precisión, siempre y cuando el resultado se pueda almacenar en registro suficientemente largo. La división no comparte esta característica. En general el resultado de una división no tiene longitud finita. La precisión deseada tiene que definirse de ante mano, definiendo la longitud del resultado. La cantidad de ciclos de algoritmo va a depender entonces de la precisión deseada, no de la longitud de los operandos.

2. Algoritmos de división por recurrencia de dígitos

2.1. Ecuación de la división en base *B*

Sean

$$X = x_{n-1},x_{n-2},\ldots,x_1,x_0$$
$$Y = y_{n-1},y_{n-2},\ldots,y_1,y_0 \qquad x_i,\ y_i \in \{0,1,\ldots,B-1\} \qquad (4.1)$$

las expresiones en base *B* del dividendo y del divisor respectivamente. Sin perder en generalidad se supone que $X<Y$ lo que se puede lograr por desplazamiento de operandos (en representación normalizada, los desplazamientos de las mantisas se compensan por modificaciones en los exponentes).

Se nota el cociente como

$$Q = 0\cdot q_1,q_2,\ldots q_m = X/Y \qquad q_i \in \{0,1,\ldots,B-1\} \qquad (4.2)$$

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

2.1.1. Lema (lema de convergencia de la división en base *B*)

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 satisfaciendo

$$B\cdot r_{i-1} = Y\cdot q_i + r_i \qquad q_i \in \{0,1,\ldots,B-1\} \qquad (4.3)$$

con condición $0 \le r_i \le Y-1$ sobre el **resto** r_i .

Como consecuencia del lema 1, y asumiendo $r_0 = X$, la aplicación recursiva de (3) procura

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

(4.4)

o

$$Q = X/Y = 0\cdot q_1,q_2,\ldots q_m + (r_m\cdot B^{-m})/Y. \qquad (4.5)$$

2.2. Algoritmo de división en base *B*

Nota: "/ " representa la división entera

r(0) := X ;
For** i **in** 1,...,m **loop

```
   q(i) := B*r(i-1)/Y;
   r(i) := B*r(i-1)-Y*q(i);
endloop;
   r := r(m);
```

2.3. Algoritmo de división en base 2

2.3.1. Algoritmo de división con restauración (Restoring)

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

Los circuitos a continuación (fig. 24,25) materializan este algoritmo.

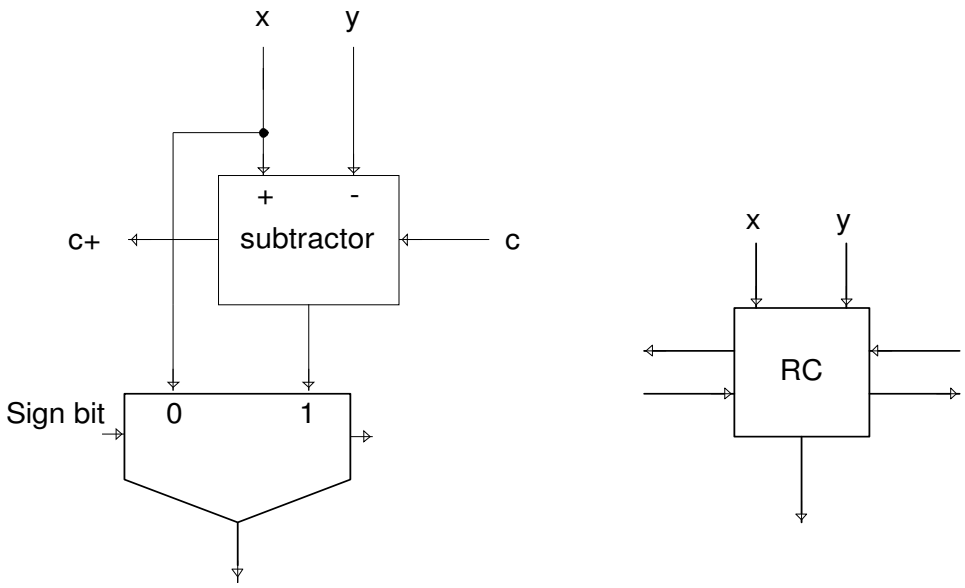


Fig.24. Celda Elemental RC *"Restoring cell"*

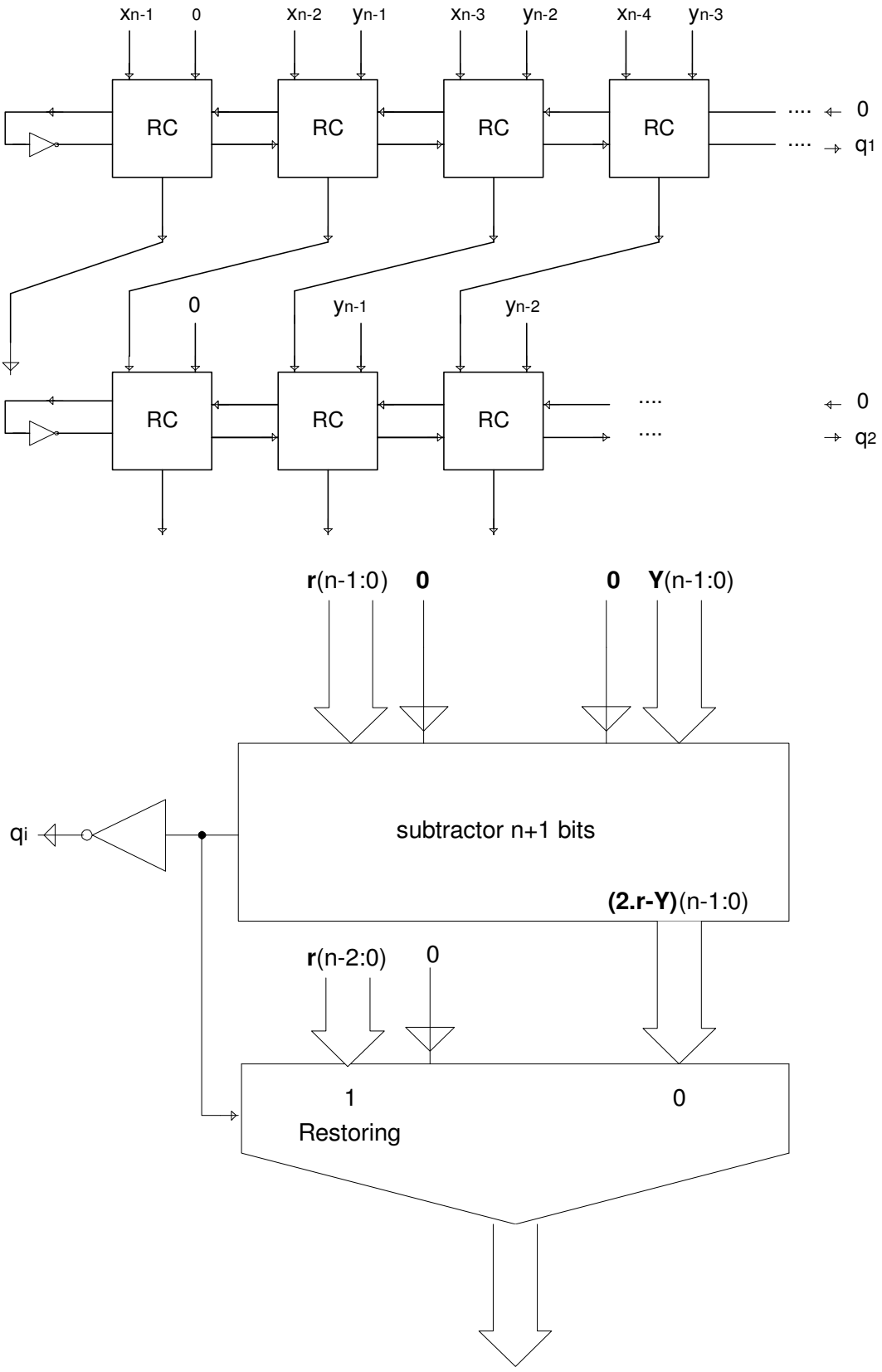


Fig. 25. Arreglo de división *"Restoring"*

La celda elemental RC (fig. 24) esta constituida por un restador (full subtractor) y un multiplexor que selecciona entre el resultado de la resta y en caso de resultado negativo, el operando inicial desplazado (restauración).
En la fig. 25, el primer paso, trivial *($q_0 = 0$)* no esta representado, *$2 \cdot X - Y$* es la operación realizada en el primer paso.

2.3.2. Algoritmo de división sin restauración (Non-restoring)

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

El circuito de la fig. 26 materializa el algoritmo.

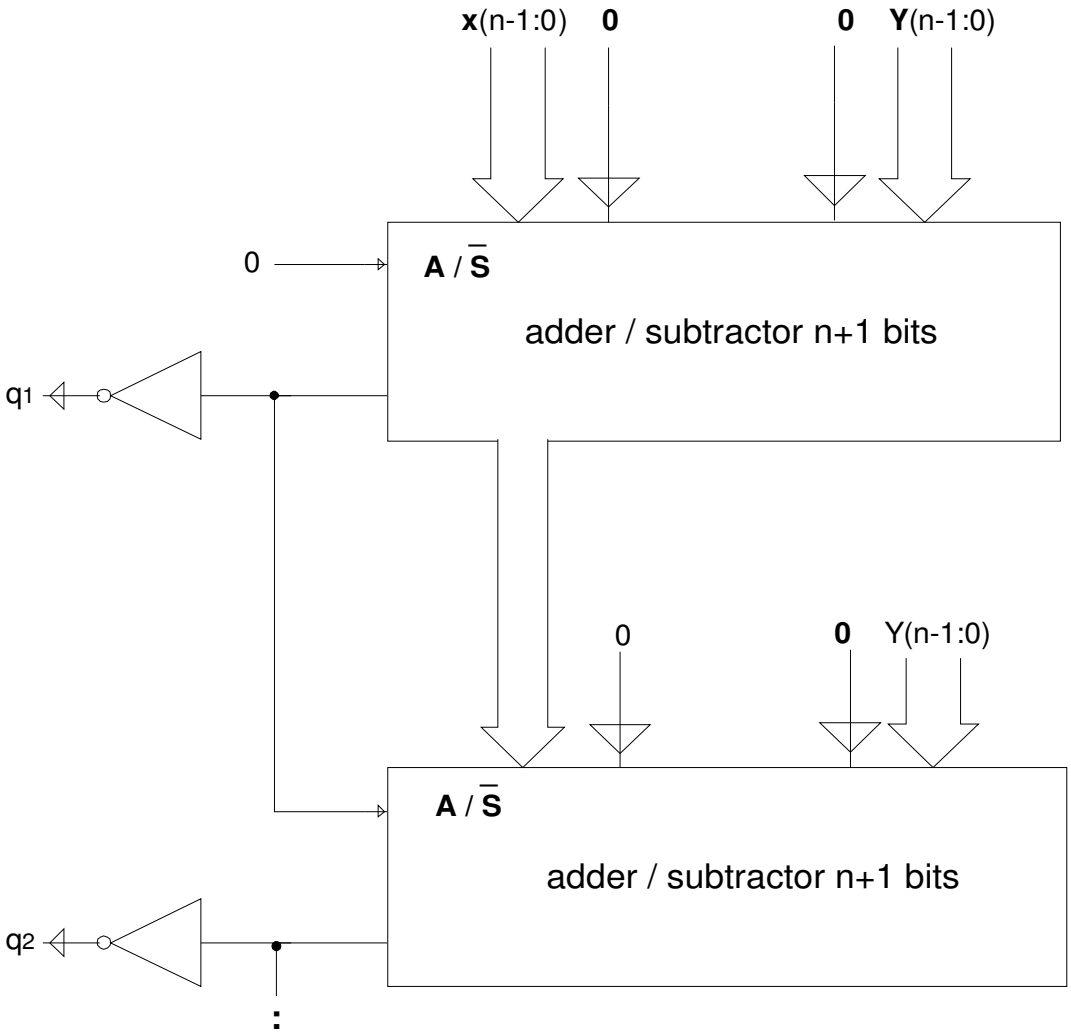


Fig.26. Arreglo de división "Non-restoring"

2.3.3. Ejemplos

Sean

X=0.1110111011101101

Y=0.1101

mantisas normalizadas del dividendo y del divisor respectivamente

Se propone calcular la mantisa normalizada de Q=X/Y con 9 bits de precisión.

Para garantizar la normalización de Q, se necesita:

X<Y.

Como dicha condición no se cumple, se necesita un paso preliminar de división por 2 de X (desplazamiento de X de un bit a la derecha). Dicho desplazamiento se compensará por un aumento del exponente (+1).

Después de agregar un bit de signo (representación en signo puro), los datos se escriben como

X =00.01110111011101101

Y =00.11010000000000000

-Y=10.11010000000000000

a) <u>División con restauración</u> (restoring)

Paso #	Operaciones	identificadores	dígitos del cociente	Comentarios
1	00.01110111011101101	R(0)=X		
	10.11010000000000000	-Y		
	- - - - - - - - - - - - - - - - - -	R(1)=R(0)-Y<0	0.	no se calcula ; q=0
2	00.11101110111011010	2xR(0)		restauración
	10.11010000000000000	-Y		
	00.00011110111011010	R(2)>0	0.1	q=1
3	00.00111101110110100	2xR(2)		resto desplazado
	10.11010000000000000	-Y		
	- - - - - - - - - - - - - - - - - -	R(3)<0	0.10	no se calcula ; q=0
4	00.01110111011101000	2x2xR(2)		restauración
	10.11010000000000000	-Y		
	- - - - - - - - - - - - - - - - - -	R(4)<0	0.100	no se calcula ; q=0
5	00.11101110111010000	2x2x2xR(2)		restauración
	10.11010000000000000	-Y		
	00.00100111011010000	R(5)>0	0.1001	q=1
6	00.01001110110100000	2xR(5)		resto desplazado
	10.11010000000000000	-Y		
	- - - - - - - - - - - - - - - - - -	R(6)<0	0.10010	no se calcula ; q=0
7	00.10011101101000000	2x2xR(5)		restauración
	10.11010000000000000	-Y		
	- - - - - - - - - - - - - - - - - -	R(7)<0	0.100100	no se calcula ; q=0
8	01.00111011010000000	2x2x2xR(5)		restauración
	10.11010000000000000	-Y		
	00.01101011010000000	R(8)>0	0.1001001	q=1
9	00.11010110100000000	2xR(8)		resto desplazado
	10.11010000000000000	-Y		
	00.00001110100000000	R(9)>0	0.10010011	q=1

b) División sin restauración (no-restoring)

Usando la representación en complemento a dos, los datos se escriben como
X =00.01110111011101101
Y =00.11010000000000000
-Y=11.00110000000000000

Paso #	Operaciones	identificadores	dígitos del cociente	Comentarios
1	00.01110111011101101	R(0)=X		
	11.00110000000000000	-Y		X>0: resta Y
	11.10100111011101101	R(1)=R(0)-Y<0	0.	q=0
2	11.01001110111011010	2xR(1)		resto desplazado
	00.11010000000000000	+Y		resto<0: suma Y
	00.00011110111011010	R(2)>0	0.1	q=1
3	00.00111101110110100	2xR(2)		resto desplazado
	11.00110000000000000	-Y		resto>0: resta Y
	11.01101101110110100	R(3)<0	0.10	q=0
4	10.11011011101101000	2xR(3)		resto desplazado
	00.11010000000000000	+Y		resto<0: suma Y
	11.10101011101101000	R(4)<0	0.100	q=0
5	11.01010111011010000	2xR(4)		resto desplazado
	00.11010000000000000	+Y		resto<0: suma Y
	00.00100111011010000	R(5)>0	0.1001	q=1
6	00.01001110110100000	2xR(5)		resto desplazado
	11.00110000000000000	-Y		resto>0: resta Y
	11.01111110110100000	R(6)<0	0.10010	q=0
7	10.11111101101000000	2xR(6)		resto desplazado
	00.11010000000000000	+Y		resto<0: suma Y
	11.11001101101000000	R(7)<0	0.100100	q=0
8	11.10011011010000000	2xR(7)		resto desplazado
	00.11010000000000000	+Y		resto<0: suma Y
	00.01101011010000000	R(8)>0	0.1001001	q=1
9	00.11010110100000000	2xR(8)		resto desplazado
	11.00110000000000000	-Y		resto>0: resta Y
	00.00000110100000000	R(9)>0	0.10010011	q=1

2.3.4. Comentarios

- El algoritmo de división "non-restoring" puede llegar a procesar números negativos, aun siendo positivos ambos operandos, dividendo y divisor. El sumador-restador (adder-subtractor) de la fig. 26 esta diseñado para procesar números con signo. Si se usa la notación complemento a dos, un sumador estándar se puede usar: cada vez que se necesite restar, la señal de control *A/S'*, generada por el bit de signo del resultado de la etapa anterior, procurará complementar *Y* y sumar *1* (cambio de signo en notación complemento a *dos*).
- La división con signo se conceptúa más fácilmente como una división en valor absoluto y cambio de signo cada vez que los operandos sean de signos

opuestos. En base *B*, los algoritmos de división con o sin restauración no son extensiones inmediatas de los algoritmos en base *2,* básicamente porque la evaluación de los dígitos del cociente es más complicada a cada paso.

3. Algoritmos de convergencia (iteración funcional)

3.1. Introducción

Los algoritmos de iteración funcional representan la división como una función. Se usan técnicas de cálculo numérico para resolver ecuaciones como la de Newton-Raphson o un desarrollo de Taylor-MacLaurin. Estos métodos proporcionan convergencia mejor que linear, pero la complejidad de cada paso es mayor que la de los algoritmos de recurrencia de dígitos. Como condición inicial su supone que sea normalizado el divisor, o sea incluido en el rango [1/B, 1[.

3.2. Técnica de iteración de Newton-Raphson

Consideramos la ecuación fundamental de la división:

$D = d \cdot Q + r$ (4.6)

El cociente exacto teórico *(r=0)* puede escribirse como

$Q = D/d = D \cdot (1/d)$

De hecho, el método de Newton-Raphson calcula primero el recíproco *x* del divisor *d* *(d≠0)*, con la precisión requerida; a continuación se multiplica el resultado por el dividendo *D*.

Siendo $x=1/d$ la raíz buscada, la función primaría

$f(x) = 1/x - d$ (4.7)

puede ser considerada para la extracción de la raíz $\{x \mid f(x)=0\}$.

Para solucionar $f(x)=0$, se usa la ecuación

$x_{i+1} = x_i - f(x_i) / f'(x_i)$ (4.8)

en forma recursiva para evaluar *x.*

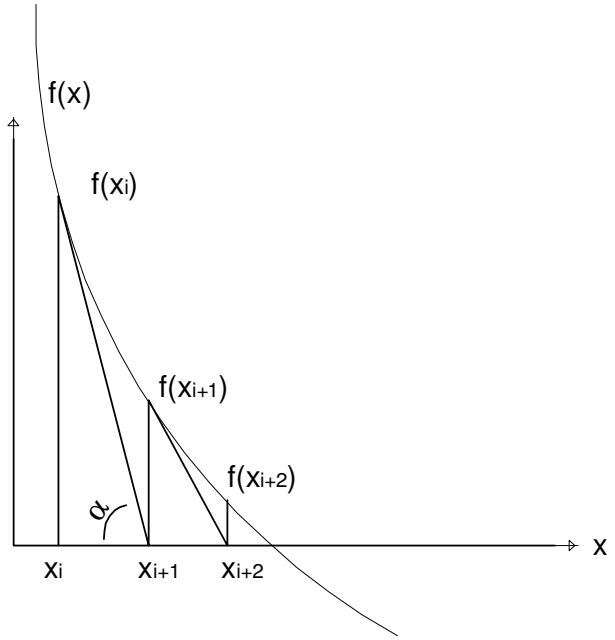


fig.27. Grafo de convergencia

La ecuación (4.8), en donde $f'(x_i)$ representa $[df/dx]_{x=x_i}$, derivada primera de $f(x)$ en el punto x_i, se ilustra a la fig. 27.
La ecuación (4.8) se deduce de:

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

proporcionando x_{i+1} como la intersección del eje x con la tangente a $f(x)$ al punto $x=x_i$.
La función (4.7) es continua y derivable en la zona (suficientemente cerca de la raíz) en donde las sucesivas aproximaciones se generan. En general, se usa una técnica de consulta de tabla (table look-up: LUT) para la primera aproximación.
Sea x_0 $(\neq 0)$ esta primera aproximación, entonces

$f(x_0) = 1/x_0 - d$

$[df/dx]_{x=0} = -1/x_0^2$

entonces

$$(1/x_0 - d)' = (1/x_0 - d) / (x_0 - x_1)$$

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

$$-((x_0 - x_1) / x_0^2) = 1/x^0 - d$$

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

$$x_1 = x_0 + x_0 - dx_0^2$$

$$x_1 = 2 x_0 - dx_0^2$$

$$x_1 = x_0 (2 - dx_0)$$

y

$x_2 = x_1(2 - dx_1)$

…

$$x_{i+1} = x_i(2 - dx_i). \qquad (4.9)$$

En base 2, siendo d en el rango (normalizado)

$$½ \leq d < 1 \qquad (4.10)$$

entonces

$1 < 1/d \leq 2$

Seleccionando x_0 dentro del rango $]1,2]$, asegurará una rápida convergencia cuadrática.
De hecho, asumiendo $x_i = 1/d + \varepsilon_i$, el error ε_i , acorde con (4.9), cumplirá con

$$1/d + \varepsilon_{i+1} = (1/d + \varepsilon_i)(2 - d(1/d + \varepsilon_i)) \qquad (4.11)$$

o sea

$$|\varepsilon_{i+1}| = d\,\varepsilon_i^2 \qquad (4.12)$$

Lo que implica que la cantidad de dígitos significativos se multiplica por dos a cada paso.
Si se extrae x_0 de una tabla LUT con p dígitos de precisión, la precisión mínima m se logrará después de $k = \lceil log_2 m/p \rceil$ pasos.

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.
x_0 = 6.179 (LUT)
$d \cdot x_0$ = 0.161828·6.179 = 0.99993521
x_1 = 6.179·(2- 0.99993521) = 6.1794003
$d \cdot x_1$ = 0.161828·6.1794003 = 0.9999999917484
x_2 = 6.1794003·(2-0.9999999917484) = 6.179400350989939
$d \cdot x_2$ = 0.161828·6.179400350989939 = 0.99999999999999848492
x_3 = 6.179400350989939·(2-0.99999999999999848492)
= $6.17940035098993993622858837778 37 \pm 10^{-31}$

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 lo divido por b^n de manera que quede un d' = 0,d

$d' = d/b^n$

entonces

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

Por ejemplo, calcular el recíproco de d = 5 en base 2 sera:

$d = 101_{(2}$

como d está furea del rango, entonces

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

entonces

$x' = 1/d' = 1/0,101_{(2} = 1000_{(2}/101_{(2} = 8^{(10}/5_{(10} =$ **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

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

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

Ejemplo en base 2 del método de Newton-Rapson

x0 = 1,1 (primer valor aproximado del resultado extraído de LUT)

x1 = 1,1 * (10 – 0,101 * 1,1)

x1 = 1,1 * (10 – 0,1111)

x1 = 1,1 * (1,0001)

x1 = 1,10011

x2 = 1,10011 * (10 – 0,101 * 1,10011)

x2 = 1,10011 * (10 – 1,1111111)

x2 = 1,10011 * 1,0000001

x2 = 1,100110110011

NOTA:

En base dos, la operación:

2 – d*xi

es

10,00..00 – 0,xx..xx (10 seguido de n ceros menos 0, y n dígitos significativos)

igual a

01,11..11 + 00,00..01 – 0,xx..xx = (01,11..11 – 0,xx..xx) + 00,00..01

igual a

$\overline{0,xx..xx}$ + 00,00..01

y equivalente a

$\overline{0,xx..xx}$

Esto significa que se puede abreviar la operación de substracción por el complemento a uno de d*xi a costa de perder un '1' en el dígito menos significativo.
Si bien esta simplificación favorece la implementación del circuito de hardware puede traer problemas de convergencia como se ve a continuación con el ejemplo tratado arriba (recíproco de d = 101)

x0 = 1,1

x1 = 1,1 * (10,0000 – 0,1111)

x1 = 1,1 * (1,0000)

x1 = 1,100

Es directo inferir que x2 será igual a x1 y que este resultado ($1,1_{(2}$) es igual a $1,5_{(10}$ (aproximado a $1,6_{(10}$ que es el resultado exacto)

Comentario

Cada paso de iteración implica dos multiplicaciones y una resta. En base *2*, la resta es equivalente a una operación de complemento a dos; Además, por al costo de un bit de precisión a cada paso, la operación de complemento a dos puede substituirse por una simple complementación bit por bit. La fig.28 a continuación muestra a implementación secuencial del algoritmo.

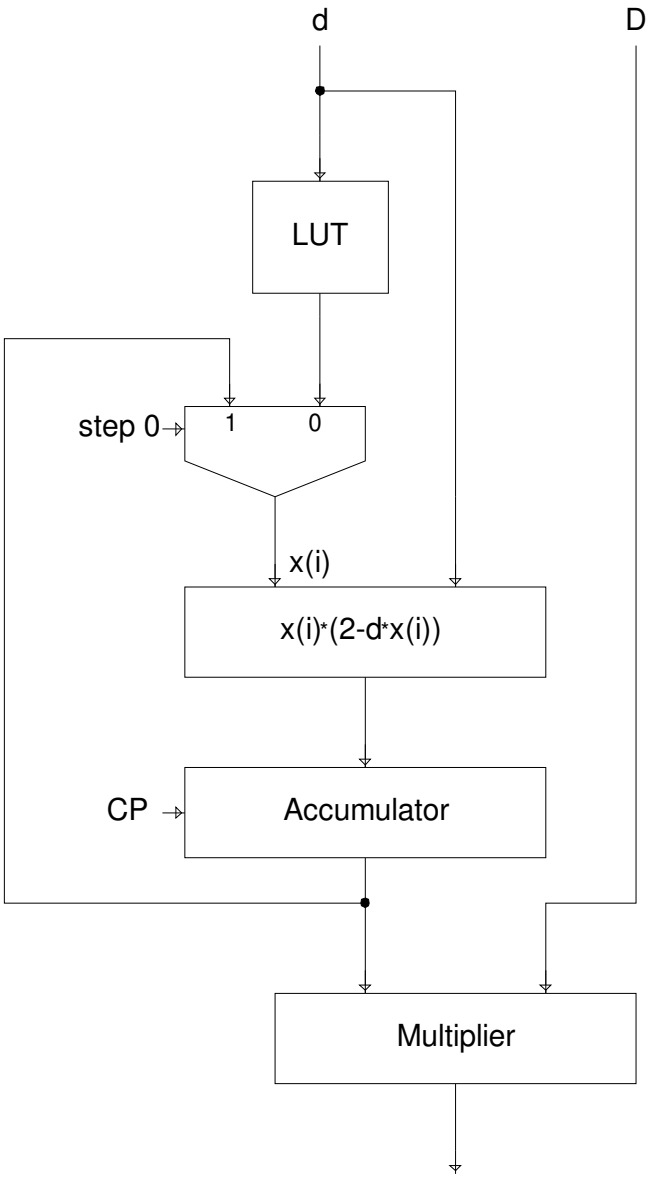
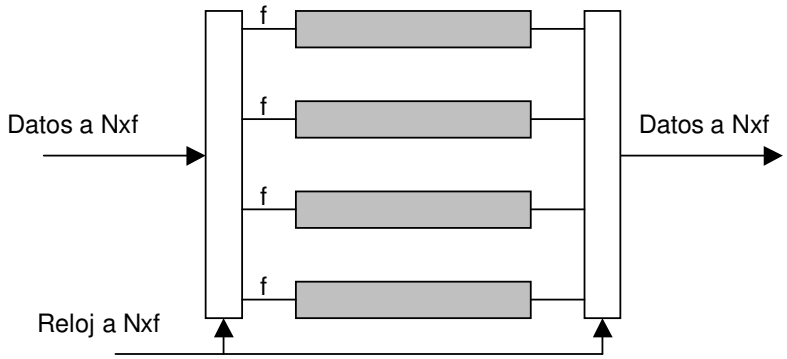


Fig. 28. Implementación secuencial del algoritmo de Newton-Raphson.

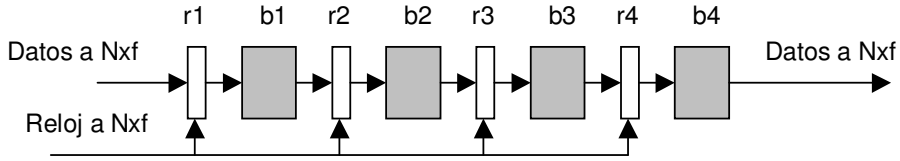
Capitulo 5. Segmentación

1. Introducción

Si se pretende acelerar la velocidad de cálculo de un circuito determinado existen 2 alternativas básicas. Por un lado, se puede optar por replicar el hardware del circuito a fin de poder realizar más operaciones de determinado tipo simultáneamente (Fig. 29-a). La segunda alternativa, que se estudiará en profundidad en este capítulo, es la segmentación de circuitos o pipeline (Fig. 29-b).



a) Paralelismo espacial o replicación de hardware



b) Paralelismo temporal o segmentación

Fig. 29. Alternativas para la aceleración de cálculo de un circuito

En principio, la técnica de segmentación permite aumentar la frecuencia de muestreo de los datos de entrada de un sistema, así como la frecuencia de generación de salidas del mismo, sin que se modifique el tiempo de cálculo de cada resultado particular. Si se establece una analogía con una tuberia de agua (pipeline) se observa que en un pipeline digital se deben cumplir 2 requisitos fundamentales:

- Ningún dato puede entrar al circuito si a la vez no sale otro por el extremo final (no hay compresión de datos).
- Al igual que en el agua (si se desprecia la fricción), la velocidad del flujo de datos no depende de la longitud del pipeline.

Si se observa nuevamente la figura 29-b se verá la estructura básica de un circuito segmentado, el cual consiste en una secuencia de bloques combinacionales (bi) separados (y comunicados) por bancos de registros (ri), de manera que cada resultado generado por el bloque bi se almacena en el registro ri hasta el final del presente ciclo de reloj. En el ciclo siguiente el valor de dicho registro se utilizará como entrada del bloque bi+1.
De este modo, la técnica de segmentación permite, a cada una de las etapas de ejecución, de trabajar en paralelo sin influir sobre las demás etapas. La planificación de las ejecuciones es la siguiente:
Sean $D_1, D_2, \ldots D_k$, las sucesivas muestras de datos y $E_1, E_2, \ldots, E_k$ las ejecuciones vinculadas con las fases *1,2,...,k* respectivamente.
Sea $E_{lj} = E_l(D_j)$ la salida de la fases *l* relativa a la entrada D_j ; en forma recursiva, $E_{ijlm} = E_i(E_j(E_l(D_m)))$ la tabla 5.1 a continuación muestra la secuencia de las ejecuciones paralelas.

# Etapa	nivel 1	nivel 2	nivel 3	nivel 4	...
1	$E_1(D_1)$				
2	$E_1(D_2)$	$E_2(E_1(D_1)) = E_{211}$			
3	$E_1(D_3)$	E_{212}	E_{3211}		
4	$E_1(D_4)$	E_{213}	E_{3212}	E_{43211}	
...	...	...	...	...	...
k-1	$E_1(D_{(k-1)})$	$E_{21(k-2)}$	$E_{321(k-3)}$	$E_{4321(k-4)}$	...
k	$E_1(D_k)$	$E_{21(k-1)}$	$E_{321(k-2)}$	$E_{4321(k-3)}$	...

Tabla 5.1.

Después de *k-1* ciclos de inicialización, el circuito provee una salida en cada etapa, o sea ejecuta *k* operaciones durante cada periodo de la señal de sincronización.

Una particularidad del esquema presentado (Fig. 29-a) es que el bloque más lento es quien fija la velocidad de procesamiento del sistema, aún cuando los demás bloques tengan una velocidad infinita. Por lo tanto, para aumentar la velocidad, la partición del circuito debe ser balanceada. Es decir, que todos los blques tengan el mismo retardo (tiempo de cálculo).
Una forma trivial (aunque no siempre posible) de conseguir el balance computacional es hacer todos los bloques iguales.

2. Parámetros importantes de un circuito segmentado

Los tres parámetros más importantes son la velocidad de procesamiento, la latencia y la aceleración del pipeline.

- La velocidad de procesamiento (ancho de banda) es el número de datos por unidad de tiempo que pueden ser procesados por el circuito. La magnitud inversa es el período del pipeline.

- La latencia es la cantidad de tiempo que transcurre entre la entrada del primer bit de un dato al circuito y la salida del último bit del resultado, para una única computación. Tambien se lo llama retardo o tiempo de respuesta del pipeline.

- La aceleración representa el aumento de rendimiento del pipeline respecto de la versión puramente combinacional del circuito. Numéricamente es el cociente entre el número de resultados por unidad de tiempo del circuito segmentado y el número de resultados en la misma unidad de tiempo de la versión combinacional.

A fin de ver la relación entre los tres parámetros supongamos un pipeline ideal (e irreal) con registros e interconexión de retardo nulo. El circuito original es combinacional con retardo total C. Dicho circuito se divide en N etapas balanceadas con retardo C/N cada una.
De este modo, el circuito producirá un resultado válido cada C/N unidades de tiempo, por lo que su velocidad de funcionamiento será de N/C.
Cada dato que ingrese al pipeline debe atravesar N etapas de período C/N por lo que la latencia es de C unidades de tiempo.
Puesto que el circuito adquiere máximo rendimiento al procesar varios datos continuamente, el tiempo necesario para procesar M datos contiguos vale:

$$t(M) = (N + M - 1) \cdot C/N \qquad (5.1)$$

O sea que se tarda C unidades de tiempo en sacar el primer resultado y los M-1 datos restantes se procesan a uno por ciclo, es decir a C/N unidades de tiempo cada uno.

Otra forma de definir la velocidad de proceso es considerar el cociente entre los datos procesados y el tiempo que demanda ese proceso, por lo tanto:

$$V(M,N) = (M.N) / ((N + M - 1).C) \qquad (5.2)$$

Esta última expresión permite calcular el número umbral de operaciones sucesivas para la cual la versión segmentada de un circuito es superior a la combinacional.
De las definiciones anteriores se obtiene que para un número elevado de operaciones ($M \rightarrow \infty$):

Latencia = C
Velocidad = N/C
Aceleración = N

Otros dos parámetros a tener en cuenta son los tiempo de llenado (fill time) y vaciado del pipeline. El fill time es el tiempo requerido para obtener el primer resultado válido luego de insertar el primer dato válido en el circuito. Es importante destacar que antes de ese tiempo las salidas del pipeline no arrojan ningun dato

válido. Funcionalmente el tiempo de llenado y el tiempo de vaciado son iguales a la latencia del circuito.

Por último, la granularidad (β) define el número de bloques elementales (procesadores elementales -PE-) que hay entre registros sucesivos de segmentación. En la figura 30 se observa el mismo circuito con una granulariodad de 1 (fig. 30-a), 2 (fig. 30-b) y 4 (fig. 30-c).

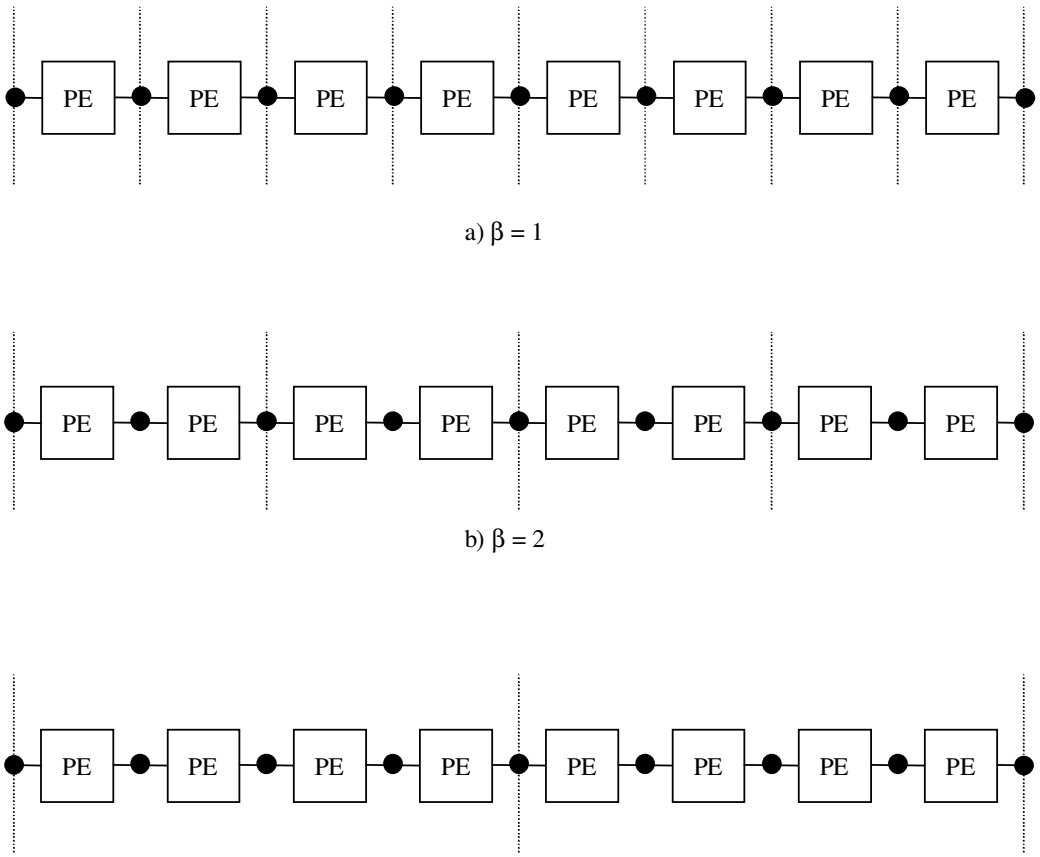


Fig. 30. Distintas granularidades de un mismo circuito.

3. Construcción y cálculo de un pipeline

La construcción de un circuito segmentado a partir de un combinacional es un compromiso entre costos (de los registros de segmentación) y velocidad. Para la mayoria de los circuitos combinacionales el proceso de segmentación demanda gran número de registros, lo que hace costosa su materialización final.
El proceso de segmentación de un sumador de 8 bits (Fig. 31-a) permitirá analizar las distintas alternativas y comlplicaciones comentadas.

Suponiendo una demora de cálculo dPE por cada procesador escalar básico (cada Full Adder - FA), el circuito combinacional del sumador completo de 8 bits (Fig 31-a) tardará un tiempo igual a:

$$Tcomb = 8 * dPE$$

Para el circuito segmentado con granularidad 1 (1 PE entre registros de segmentación consecutivos) (Fig 31-b) los parámetros serán

Latencia = 8 * (dPE + ts + tp)
Velocidad = 9 bits/ciclo (8 bits de resultado más el acarreo de salida)
Periodo mínimo = dPE + ts + tp
Aceleración = Tcomb / período = (8 * dPE) / dPE + ts + tp
Aceleración ($M \rightarrow \infty$) ≈ 8
NR = 117 registros

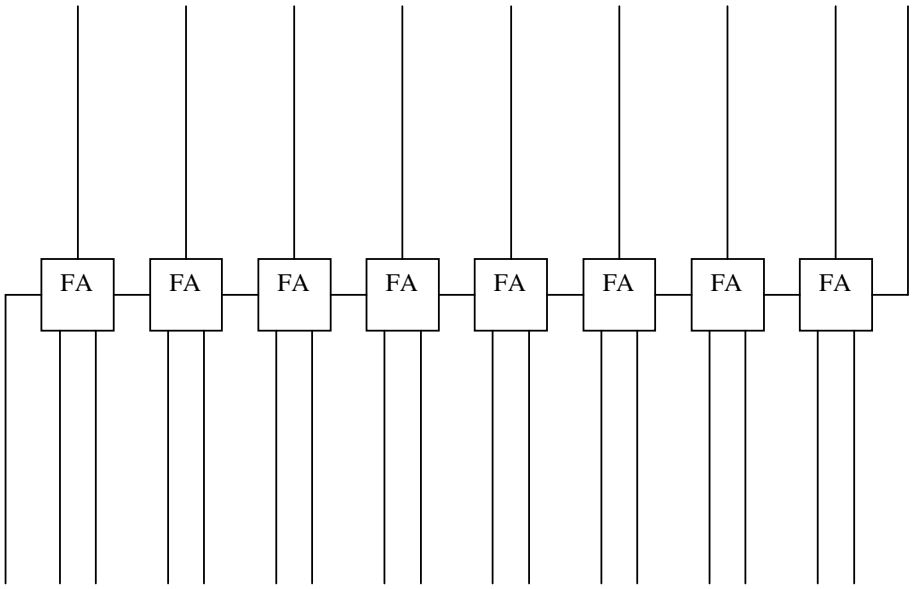
Siendo ts = tiempo de setup de cada registro de segmentación, tp = tiempo de propagación de los registros de segmentación y NR = cantidad de registros de segmentación utilizados.

Para el circuito segmentado con granularidad 2 (Fig 31-c) los parámetros serán

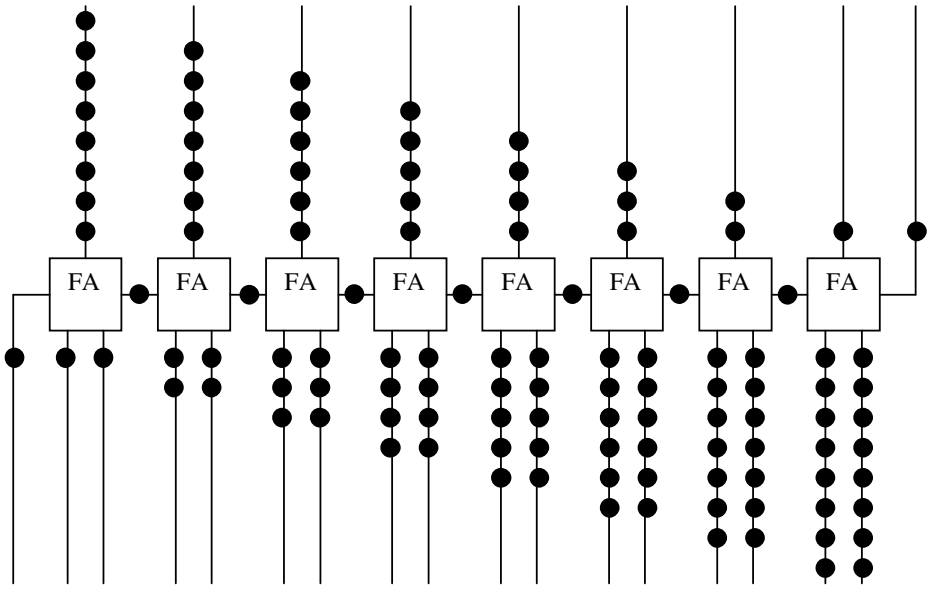
Latencia = 4 * (2 * dPE + ts + tp)
Velocidad = 9/2 bits/ciclo (8 bits de resultado más el acarreo en 2 ciclos)
Periodo mínimo = 2 * dPE + ts + tp
Aceleración = Tcomb / período = (8 * dPE) / 2 * dPE + ts + tp
Aceleración ($M \rightarrow \infty$) ≈ 4
NR = 56 registros

Para el circuito segmentado con granularidad 4 (Fig 31-d) los parámetros serán

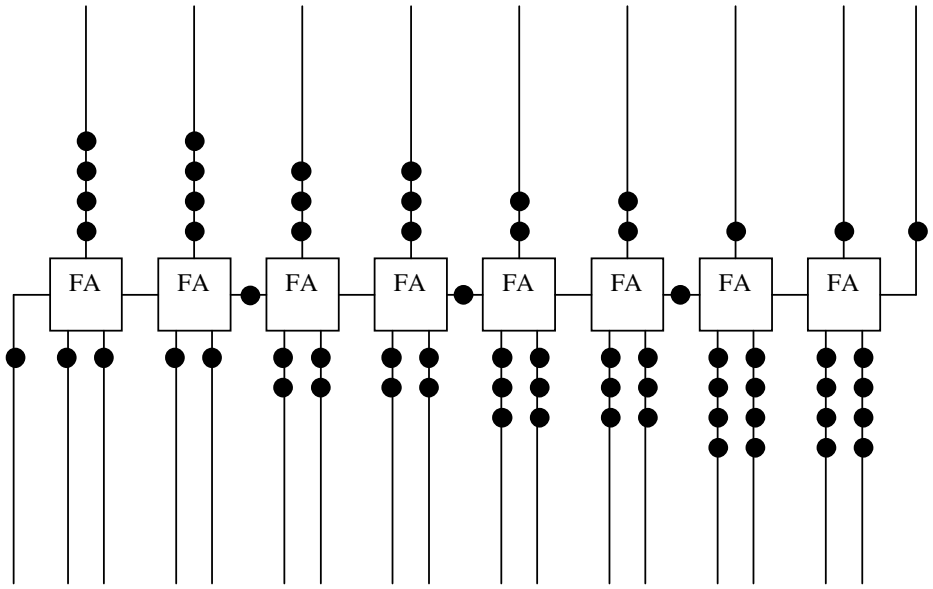
Latencia = 2 * (4 * dPE + ts + tp)
Velocidad = 9/4 bits/ciclo (8 bits de resultado más el acarreo en 4 ciclos)
Periodo mínimo = 4 * dPE + ts + tp
Aceleración = Tcomb / período = (8 * dPE) / 4 * dPE + ts + tp
Aceleración ($M \rightarrow \infty$) ≈ 2
NR = 39 registros



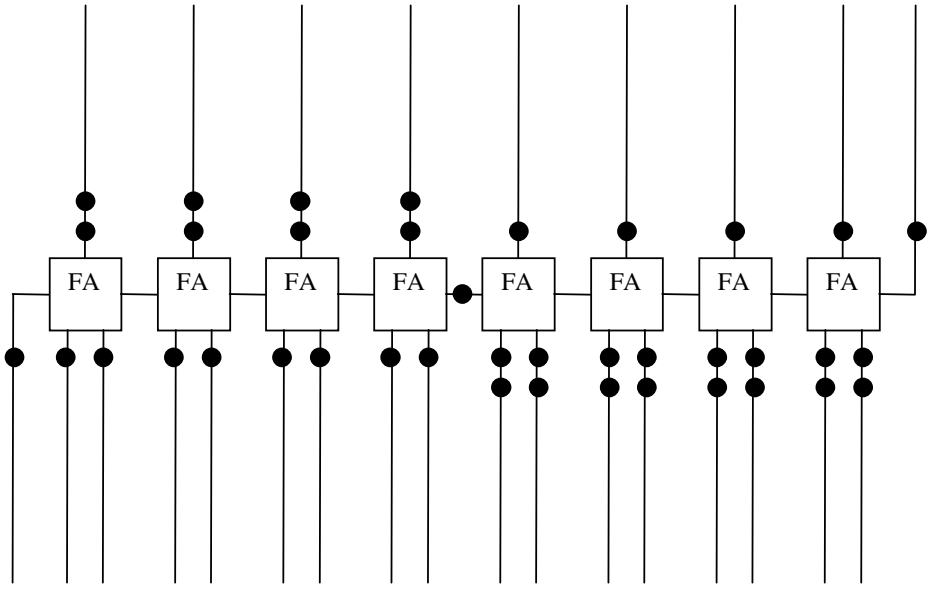
a) Sumador combinacional de 8 bits



b) Sumador segmentado de 8 bits con $\beta = 1$



c) Sumador segmentado de 8 bits con $\beta = 2$



d) Sumador segmentado de 8 bits con $\beta = 4$

Fig. 31. Sumador segmentado de 8 bits con granularidad de 1, 2 y 4

4. Ejemplos

Ejemplo 1.

En el sumador segmentado de la fig.32, se representa el flujo de datos para dos sumas consecutivos A+B y $\alpha+\beta$ respectivamente, el estado representado corresponde a la segunda etapa de cálculo para A+B y la primera para $\alpha+\beta$.

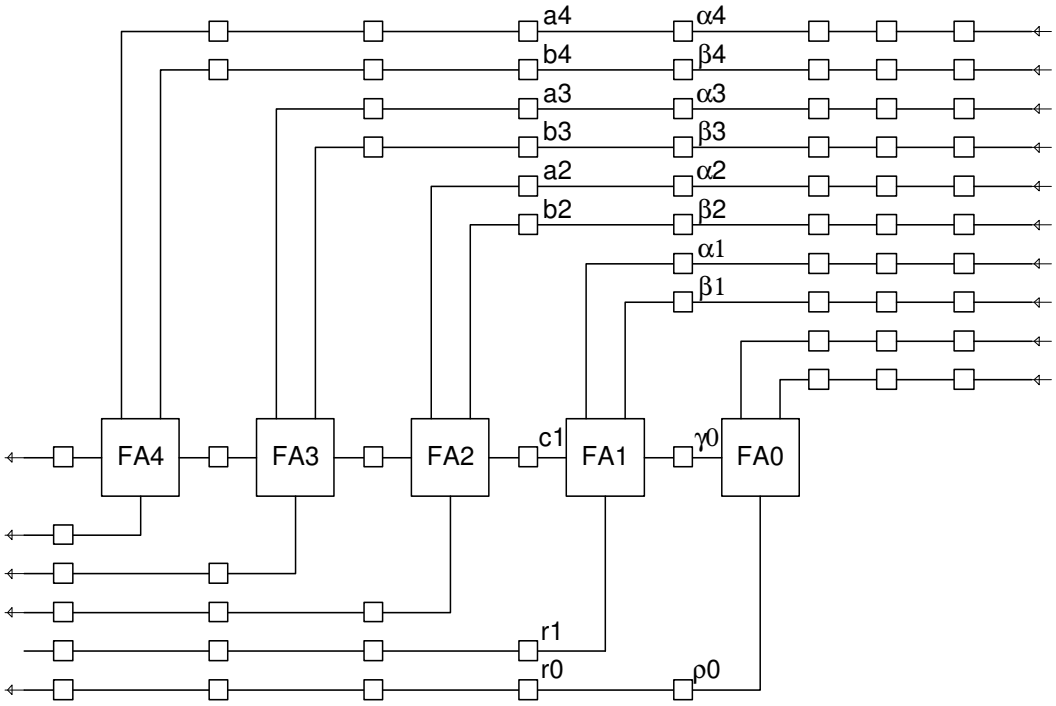


Fig. 32. Sumador paralelo segmentado

Ejemplo 2

La fig. 33 representa un multiplicador del tipo “carry-save” con segmentos de dos niveles de celdas de cálculo.

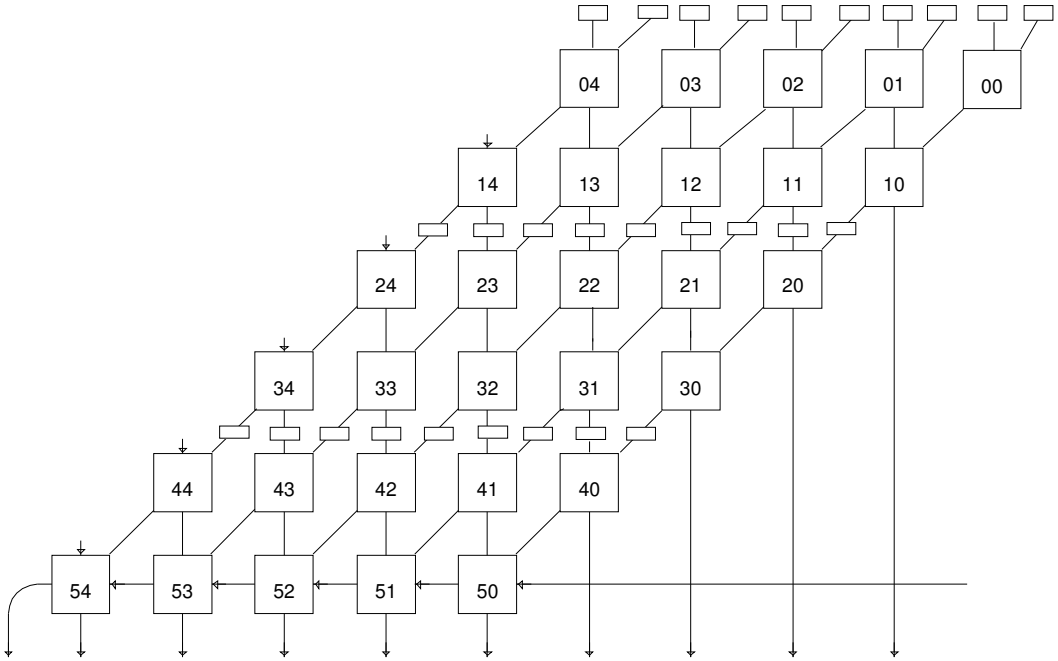


Fig. 33. Multiplicador segmentado "Carry – Save"

5. Multiplicador de matrices

El arreglo de la fig.34b representa un arreglo segmentado (sistólico) para calcular el producto matrices *3x3*; la extensión a nxn es inmediata. La fig.34a, representa la celda elemental del arreglo. El ciclo básico de tiempo tiene que ser igual o superior a la latencia de la celda elemental más el tiempo de sincronización de los cerrojos. Como se ve en la fig.34b, un nuevo producto puede aparecer cada tres ciclos (cada *n* ciclos para un producto *nxn*).

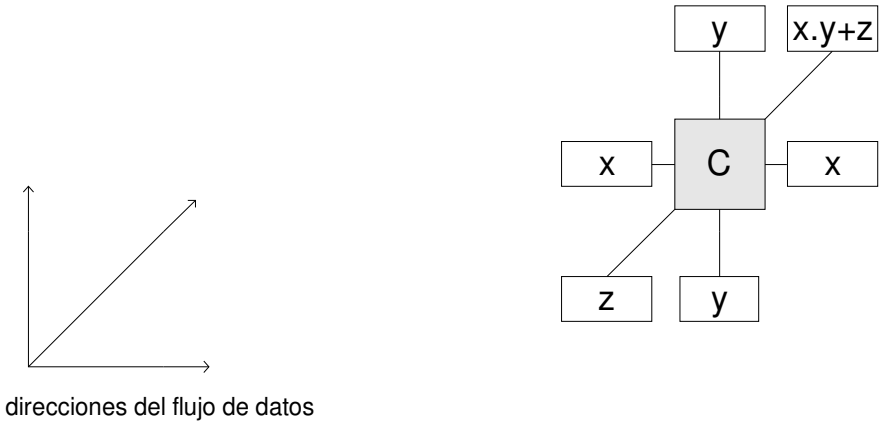


Fig. 34a. Celda elemental

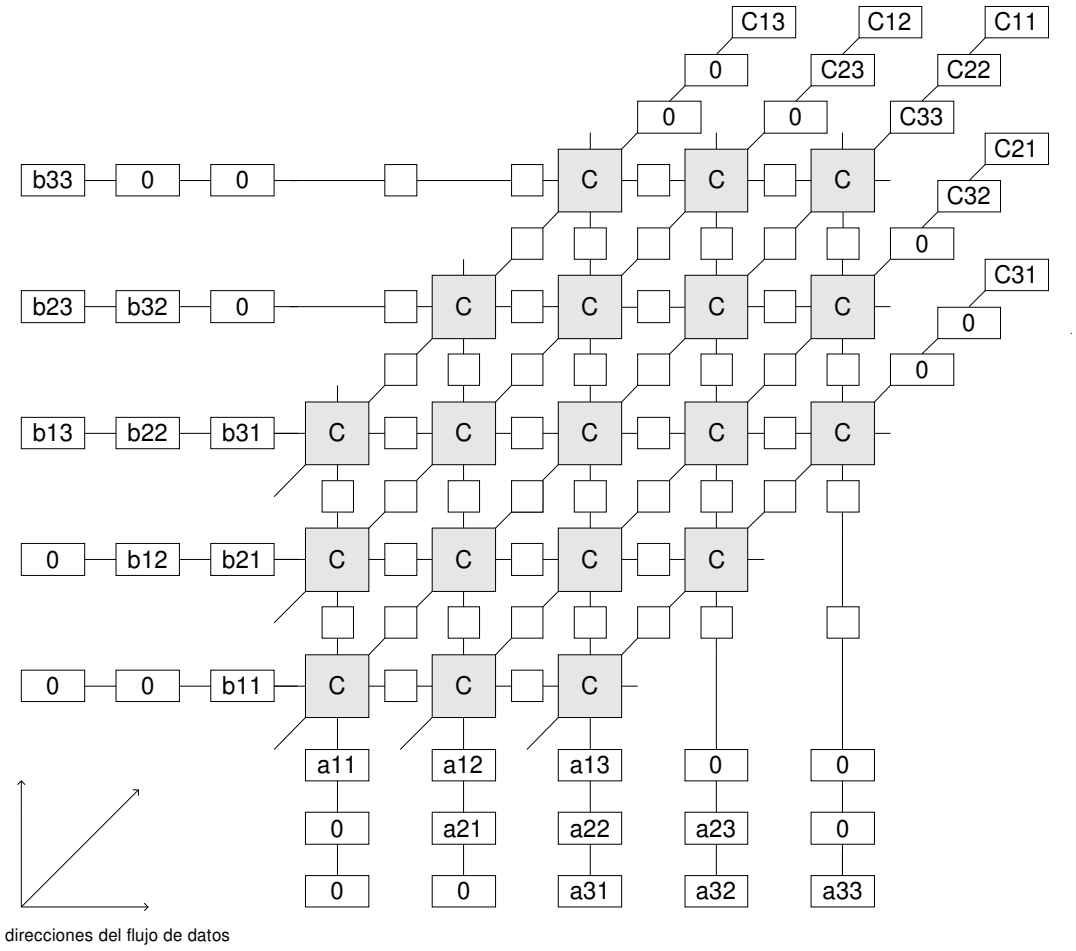


Fig. 34b. Arreglo sistólico para producto de matrices 3x3

Ejemplo 3

Arreglo sistólico para producto de matrices *2x2*.
La fig.35 ilustra las fases sucesivas de un producto de matrices *2x2*, modulo *3*:

$$\begin{vmatrix} 0 & 1 \\ 2 & 0 \end{vmatrix} \times \begin{vmatrix} 2 & 1 \\ 0 & 2 \end{vmatrix} = \begin{vmatrix} 0 & 2 \\ 1 & 2 \end{vmatrix}$$

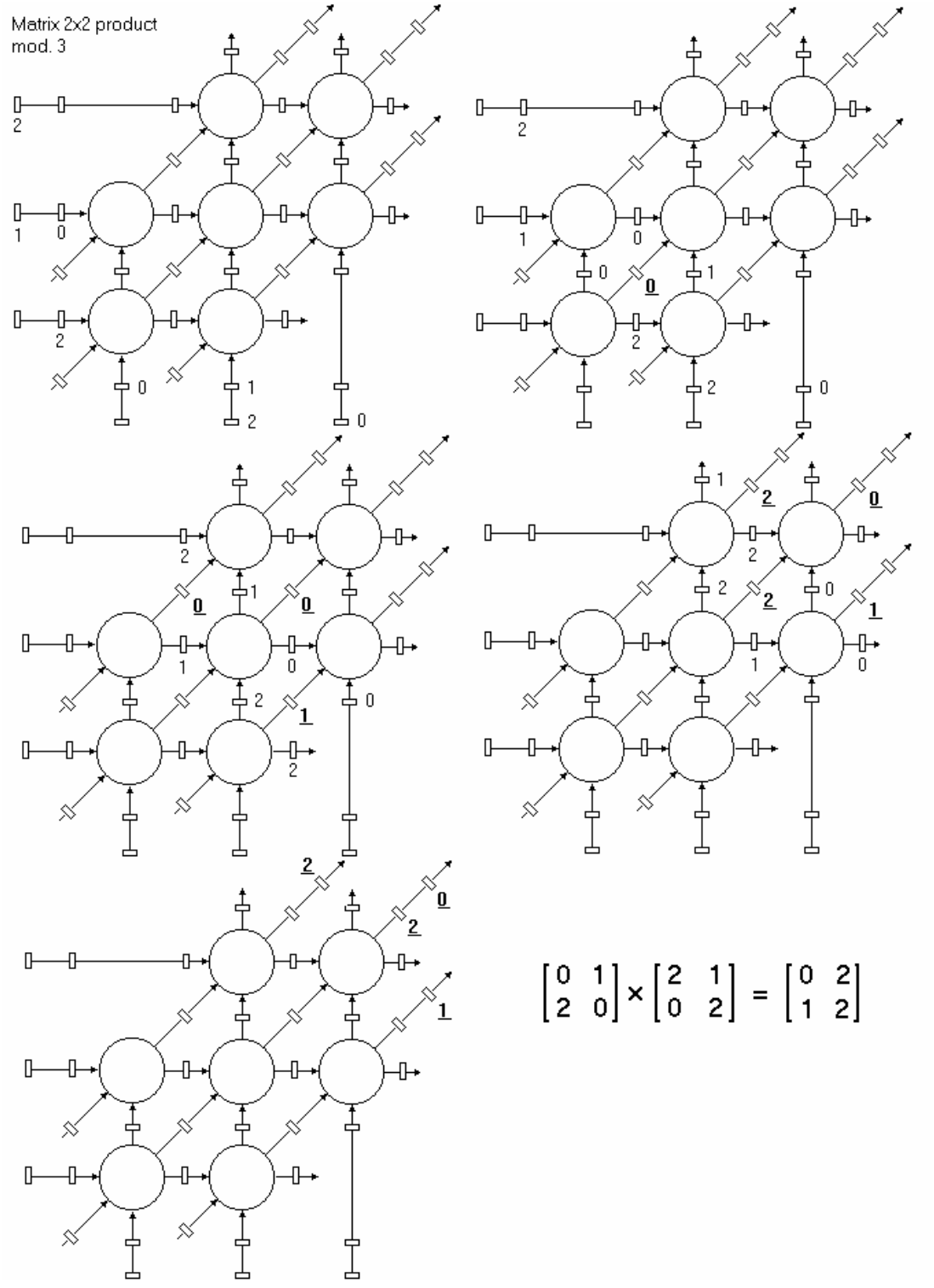


Fig. 35. producto de matrices 2x2