Arquitectura de Computadoras

(Cód. 5561)
1° Cuatrimestre 2018

Dra. Dana K. Urribarri
DCIC - UNS

CLAA en varios niveles

- Tecnología MSI
  - CLAA en ripple
  - Carry-lookahead generator (CLAG)
- Tecnología VLSI
  - Lookahead tree adder

CLAA en ripple

- Dividir las *n* etapas en grupos, cada uno con su propio CLAA y conectados en ripple.
- Grupos de igual tamaño beneficia la modularidad y el diseño de un único CI.
- Consideramos grupos de 4 bits:
  - 4 es divisor de la mayoría de los tamaños de palabras.
  - Existe el CI (74F382)

$$
\begin{aligned}
c_1 &= G_0 + c_0 P_0, \\
c_2 &= G_1 + G_0 P_1 + c_0 P_0 P_1, \\
c_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2, \\
c_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3
\end{aligned}
$$

CLAA en ripple

- Para n bits y grupos de 4, se necesitan n/4 grupos.
- Se necesita:
  - 1ΔG para todos los P_i y G_i.

    2ΔG para propagar el carry en un grupo una vez conocidos P_i, G_i y C_0.
  - (n/4)2ΔG = (n/2)ΔG para propagar el carry a través de todos los grupos.
  - 2ΔG para propagar la suma final
- Total: 1ΔG + (n/2)ΔG + 2ΔG = (n/2+3)ΔG
- Sigue siendo O(n), pero es una reducción de casi un 75% frente al ripple adder de 2nΔG

Carry-lookahead sobre grupos

- Además del lookahead interno a cada grupo, se puede proveer el carry generado y el carry propagado grupales.
- G* es el carry generado
- P* es el carry propagado
- Si G* = 1 el grupo genera (internamente) carry de salida.
- Si P* = 1 el carry de entrada al grupo se puede propagar para producir un carry de salida del grupo.

Carry-lookahead sobre grupos

- Para un grupo de 4 bits, teníamos que C_4:

$$C_4 = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3 + C_0P_0P_1P_2P_3$$

Generación grupal

Propagación grupal

- Luego los carries generados y propagados grupales son:

$$G^* = G_3 + G_2P_3 + G_1P_2P_3 + G_0P_1P_2P_3$$

$$P^* = P_0P_1P_2P_3$$

Carry-lookahead generator (CLAG)

Los P* y G* de varios grupos pueden usarse para generar los carries de entrada a cada grupo (similar a los carries de entrada a un único bit).

Carry-lookahead generator: CI que implementa estas ecuaciones.

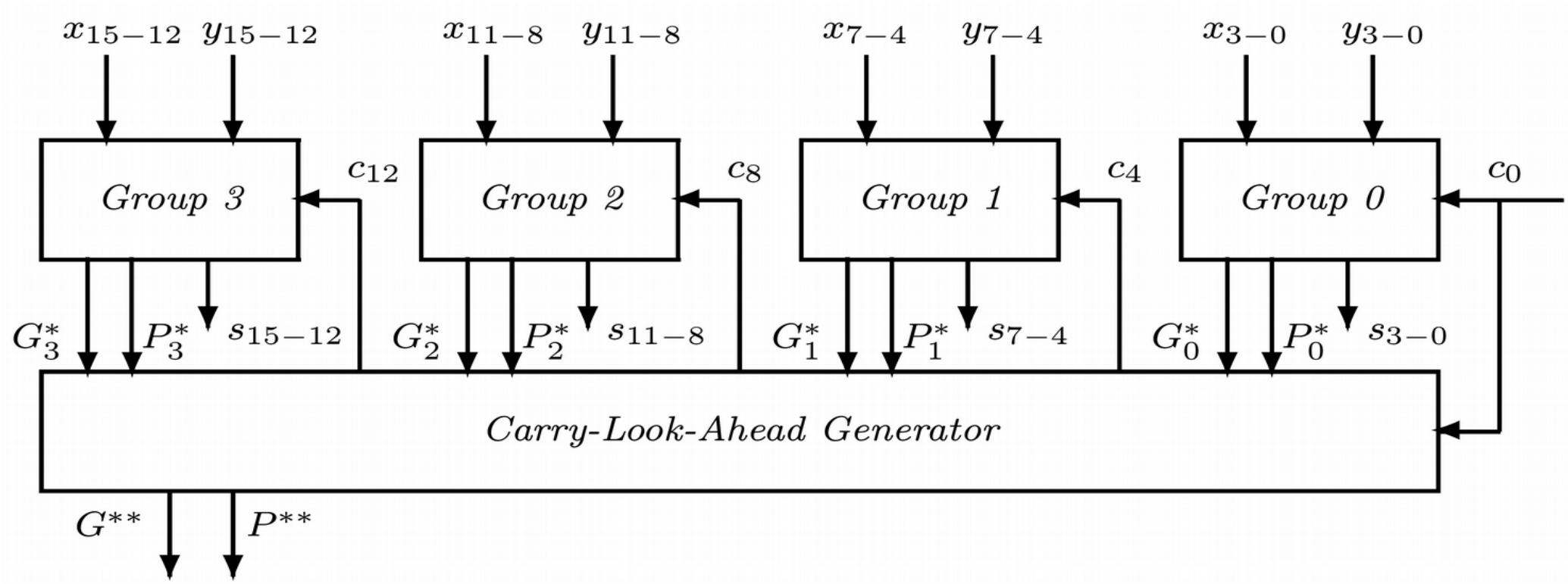
- CI 74F381 CLAA de 4 bits con salidas P* y G*.
- CI 74F182 CLAG con 4 P* y G* de entrada.

CLAA de 16 bits en 2 niveles

- Para n = 16 hay 4 grupos con salidas P_0^*, G_0^*, P_1^*, G_1^*, P_2^*, G_2^* y P_3^*, G_3^*.
- Estos P* y G* y C_0 son las entradas al CLAG.
- Las salidas del CLAG serán C_4, C_8 y C_{12}

$$
\begin{aligned}
c_4 &= G_0^* + c_0 P_0^*, \\
c_8 &= G_1^* + G_0^* P_1^* + c_0 P_0^* P_1^*, \\
c_{12} &= G_2^* + G_1^* P_2^* + G_0^* P_1^* P_2^* + c_0 P_0^* P_1^* P_2^*
\end{aligned}
$$

CLAA de 16 bits en 2 niveles



CLAA de 16 bits en 2 niveles

Pasos y retardos de la suma:

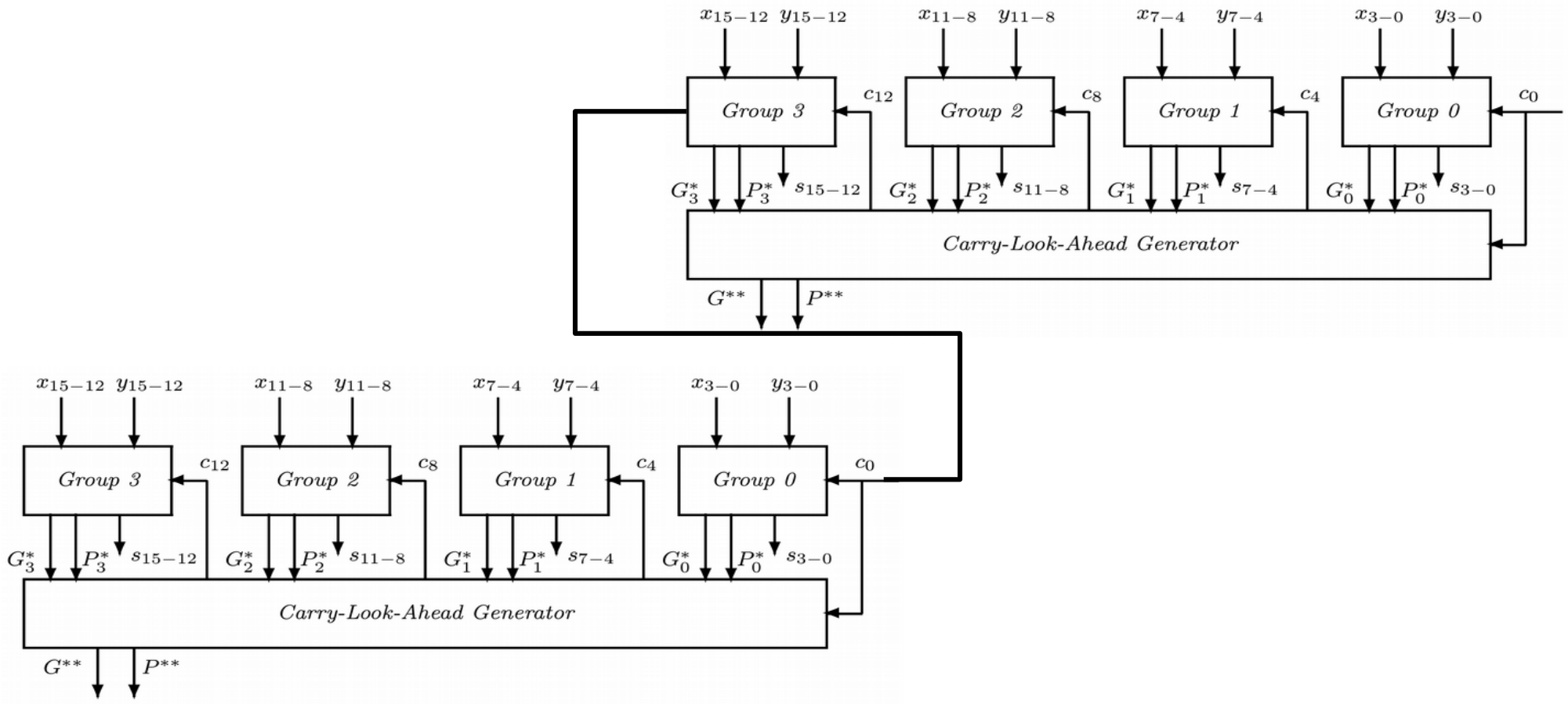
- Retardo Δ_G: Todos los grupos generan en paralelo los 4 P_i y G_i.
- Retardo $2\Delta_G$: Todos los grupos generan en paralelo P* y G*
- Retardo $2\Delta_G$: El CLAG produce los C_4, C_8 y C_{12} de salida, que son entrada a cada uno de los grupos.
- Retardo $4\Delta_G$: cada grupo calcula internamente y en paralelo la suma de los 4 bits ($2\Delta_G$ para generar los bits de carry y $2\Delta_G$ para generar los bits de suma).

Tiempo total: $9\Delta_G$

Suma de 16 bits con 4 CLAA en ripple: $(16/2+3)\Delta_G = 11\Delta_G$

CLAG en ripple

- Para lograr sumas de más bits (32bits) los CLAG se pueden conectar en ripple.



Salidas P** y G** en el CLAG

- El CLAG también produce salidas G y P que representan el carry generado de la sección (G**) y el carry propagado de la sección (P**).
- Las expresiones son equivalentes a generar el P* y G* en un CLAA

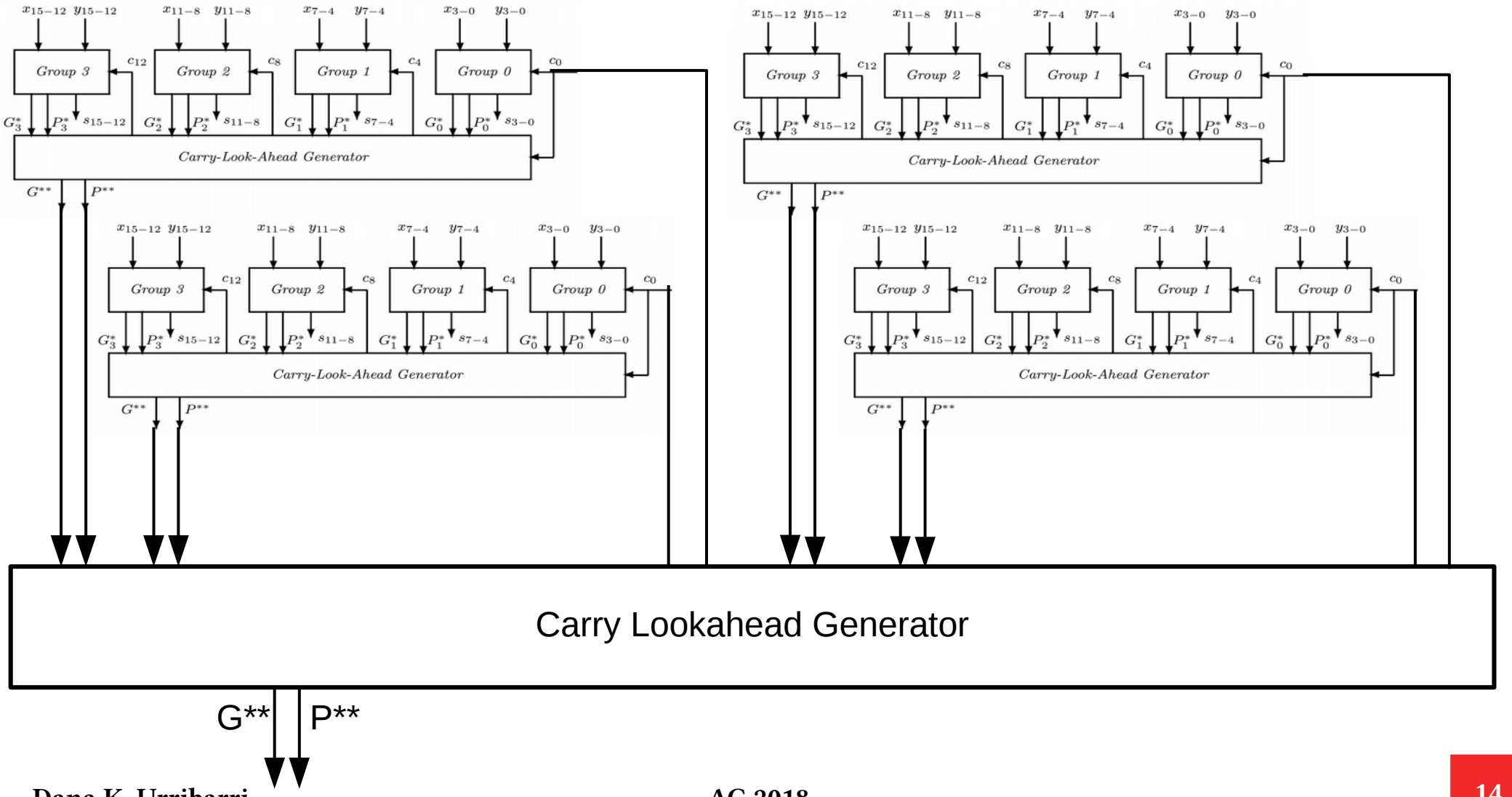
$$G^{**} = G^*_3 + G^*_2P^*_3 + G^*_1P^*_2P^*_3 + G^*_0P^*_1P^*_2P^*_3$$

$$P^{**} = P^*_0P^*_1P^*_2P^*_3$$

CLAG en varios niveles

- Estas salidas P** y G** permiten agregar un segundo nivel de CLAG.
- El CLAG en la raíz recibe los 4 pares de G** y P** y produce los carries C_{16}, C_{32} y C_{48}. (suma hasta 64 bits)
- A medida que *n* crece, se pueden agregar más niveles de CLAG.
- Para *n* bits y bloques de *b* bits, se necesitan Log_b *n* niveles.

CLAG en varios niveles



Lookahead tree adder

- Generalización de las ecuaciones
- Dado el grupo de bits en las posiciones j, j+1, … , i (j ≤ i) notamos:
  - $P_{j:i}$ carry propagado
  - $G_{j:i}$ carry generado
- Si $P_{j:i} = 1$, el carry de entrada en la posición j se propaga hasta la posición i+1
- Si $G_{j:i} = 1$, el carry se genera en alguna posición entre i y j y se propaga hasta i+1

Lookahead tree adder

Las funciones P y G pueden calcularse usando las ecuaciones:

$$P_{j:i} = \begin{cases} P_i & \text{Si } i = j \\ P_i \cdot P_{j:i-1} & \text{Si } j < i \end{cases}$$

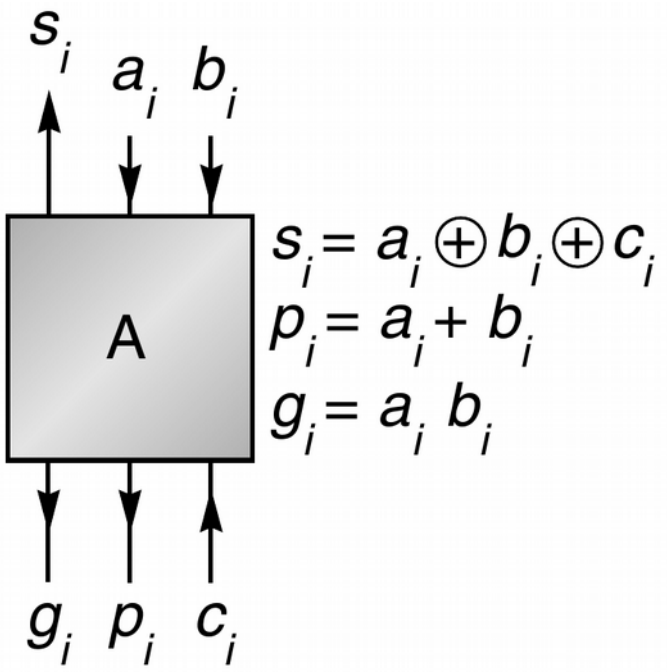
$$G_{j:i} = \begin{cases} G_i & \text{Si } i = j \\ G_i + P_i \cdot G_{j:i-1} & \text{Si } j < i \end{cases}$$

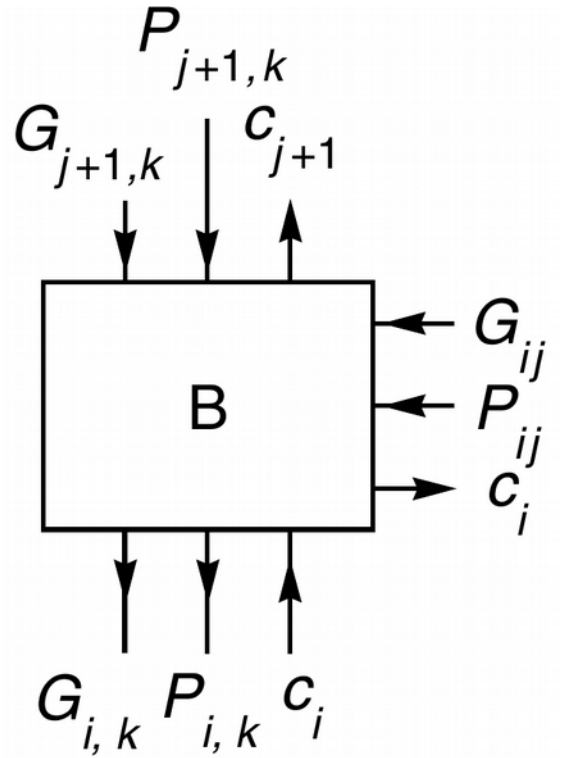
De manera recursiva pueden generalizarse a ($j+1 \leq m \leq i$):

- $P_{j:i} = P_{m:i} \cdot P_{j:m-1}$
- $G_{j:i} = G_{m:i} + P_{m:i} \cdot G_{j:m-1}$

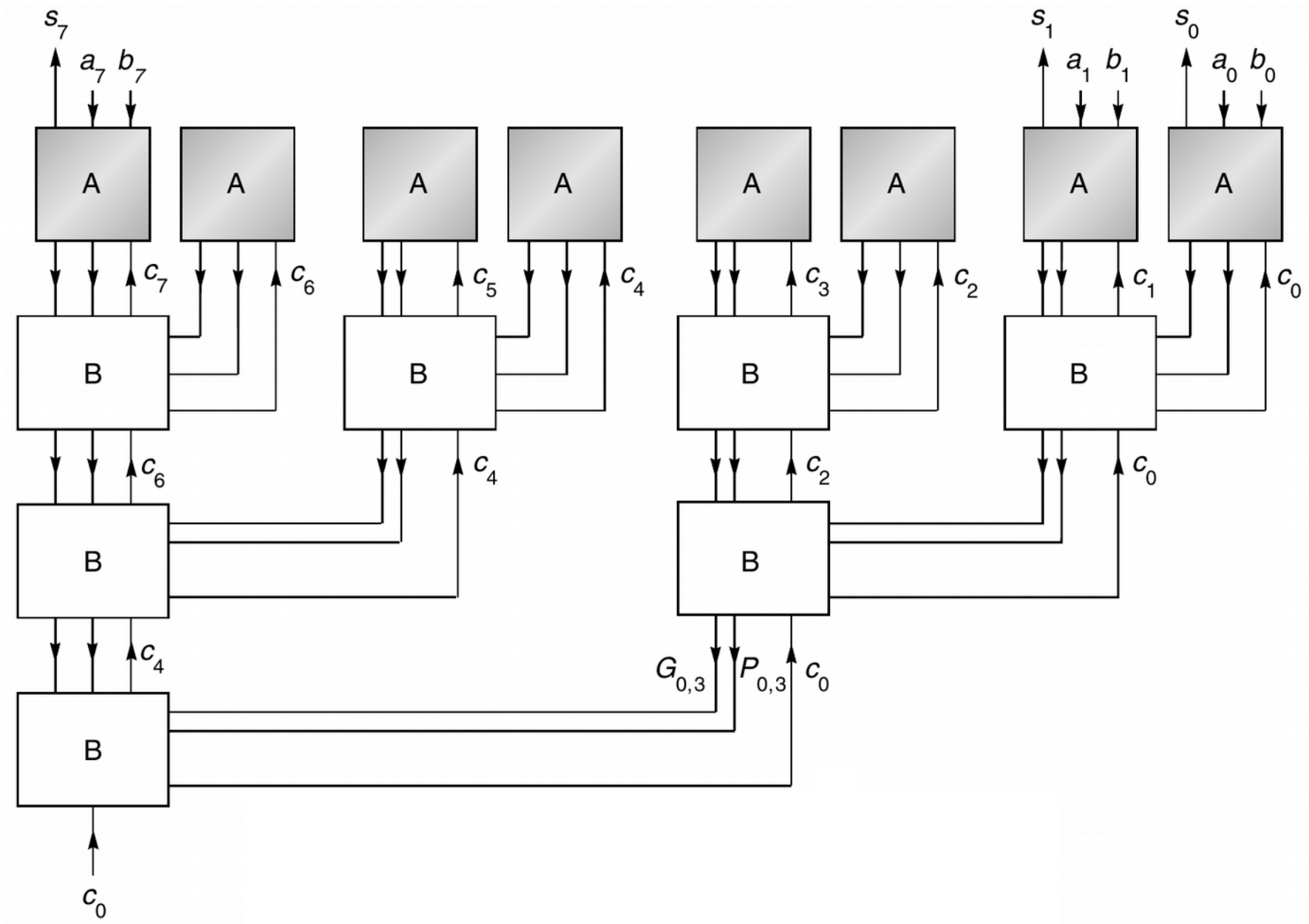
Lookahead tree adder

- Tenemos dos tipos de bloques:
  - A: a partir de los bits a_i, b_i y c_i calcula: p_i, g_i y s_i
  - B: a partir de los P y G de dos grupos distintos consecutivos, calcula los P y G del grupo completo y el carry final.

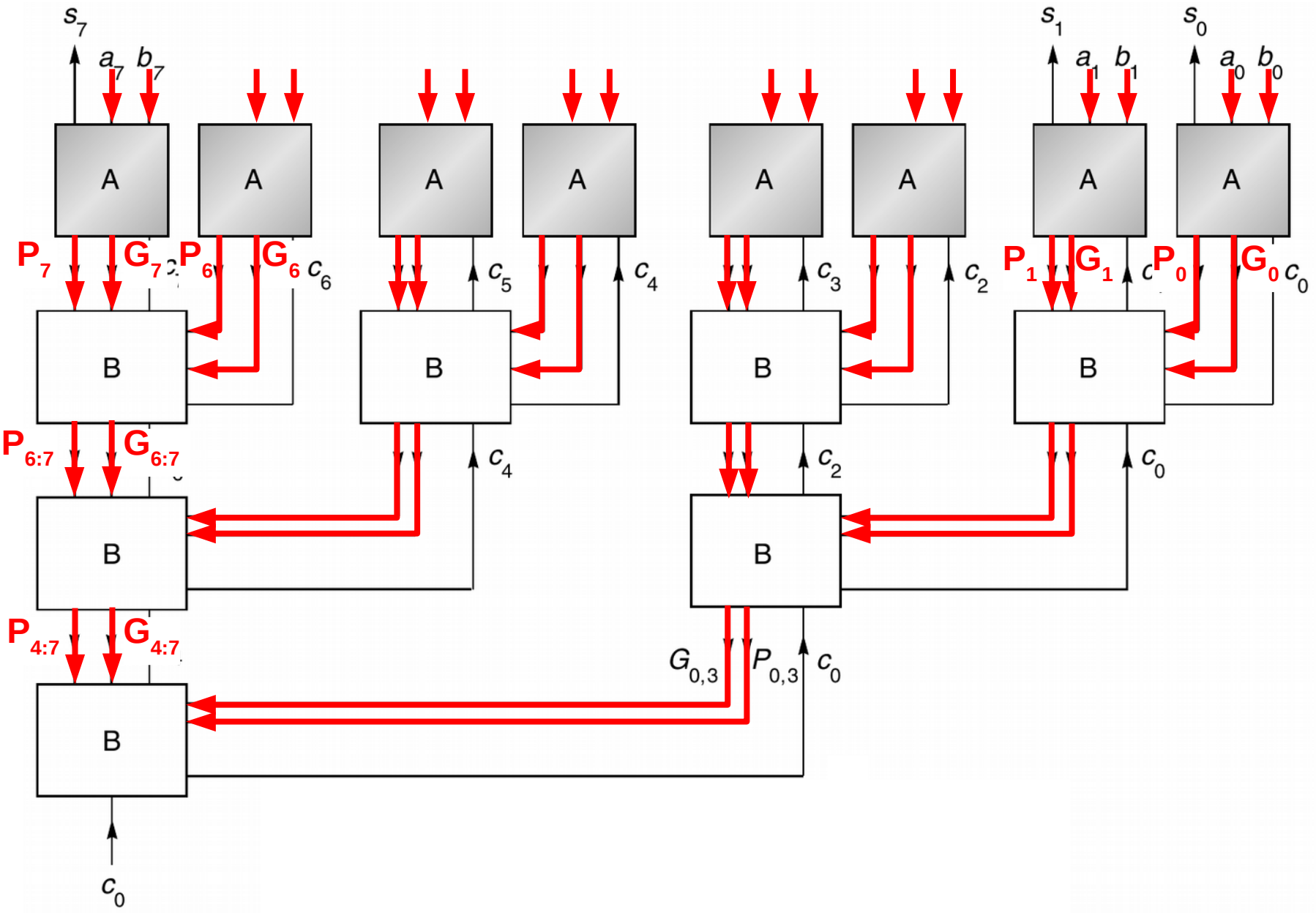




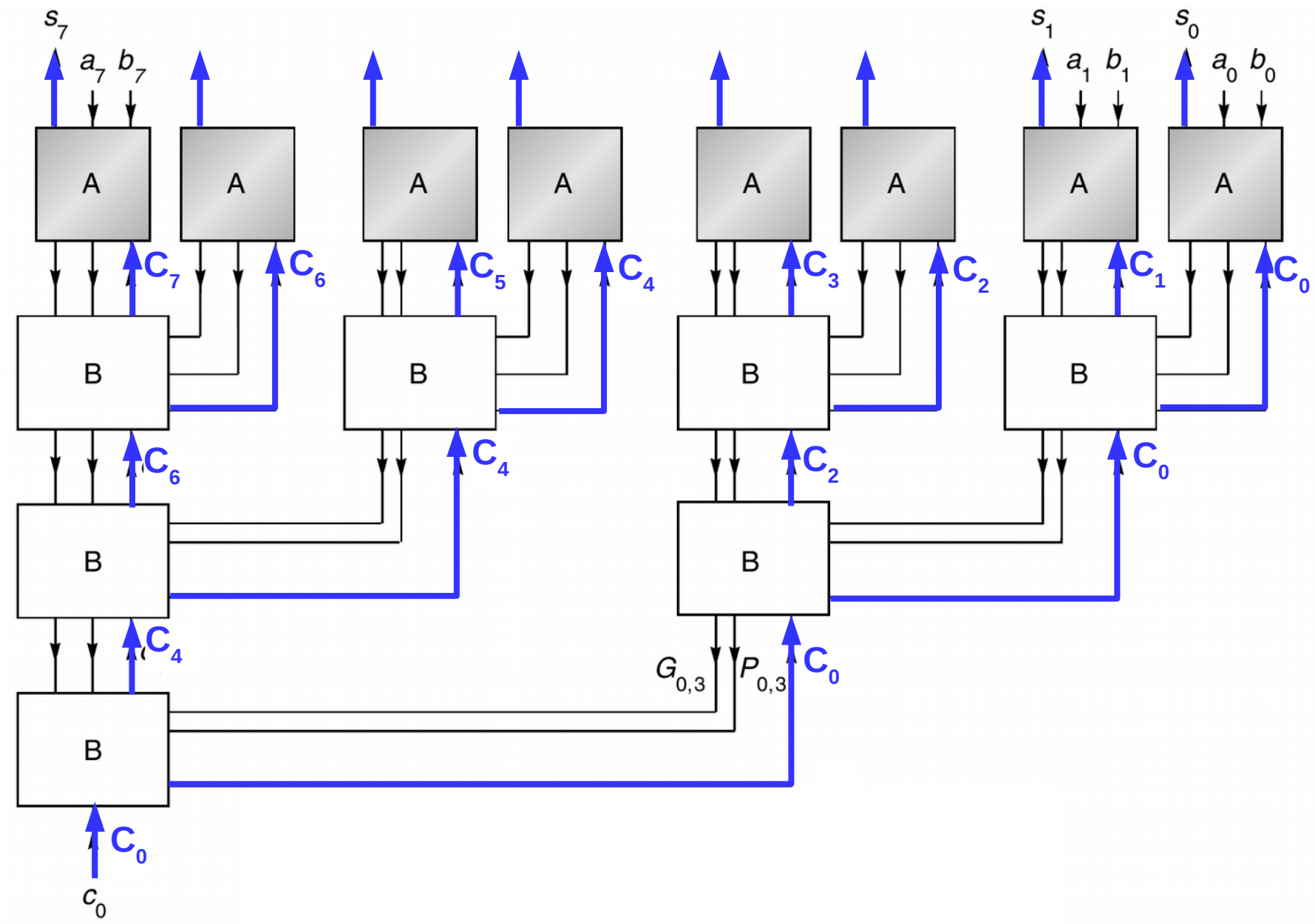
Lookahead tree adder



Lookahead tree adder



Lookahead tree adder



Lookahead tree adder

- Gran mejora en performance:

  Los bits deben atravesar en el orden de $\log_2 n$ niveles lógicos, comparados con los $2n$ del ripple.

- Incremento en el tamaño:

  $2n$ celdas en un espacio n $\text{Log}_2 n$ (nodos de dos entradas) frente a n celdas en el ripple.

Sumadores de n bits

- Sumador serie
- Sumadores paralelos
  - Ripple adder
  - Carry-lookahead adder
  - Carry-skip adder
  - Carry-select adder
  - Semisumador: Carry-save adder

Carry-skip adder

- La tecnología ripple es simple y rápida.
- Pero no tiene buen rendimiento para grandes valores de *n*.
- Soluciones intermedias son
  - Carry-skip adder
  - Carry select adder

Carry-skip adder

- Divide los operandos en grupos no necesariamente iguales.
- Reduce el tiempo de propagación del carry saltando sobre grupos consecutivos.

- Nota:
  - Se hizo popular con la tecnología VLSI.
  - Para los tamaños de palabras usuales, la velocidad es comparable a las técnicas de CLA. Además, requiere menos área y consume menos.

Carry-skip adder

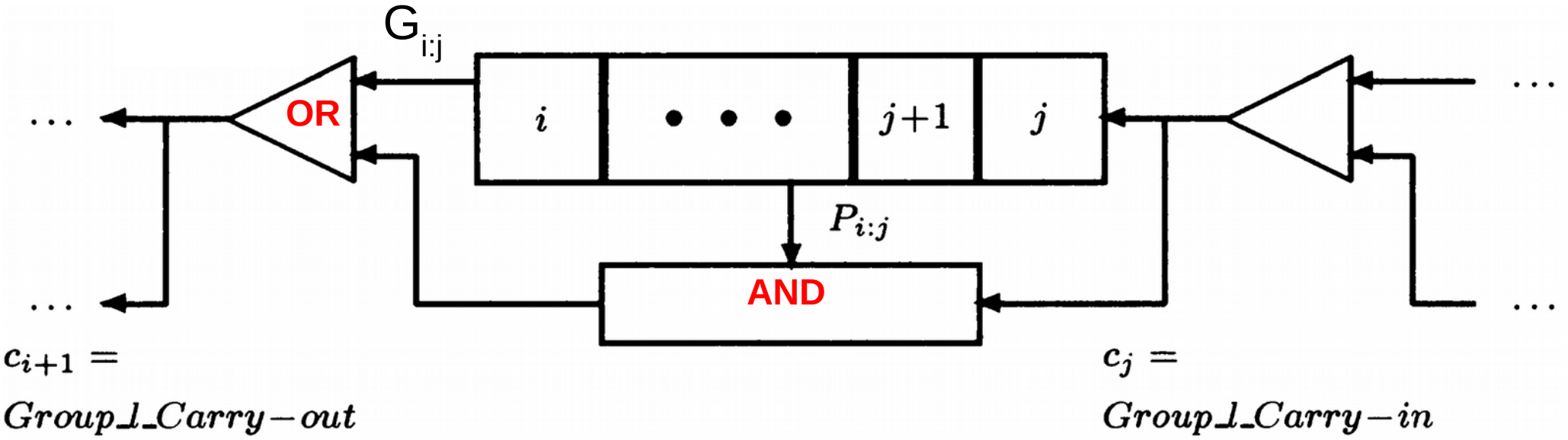
- El funcionamiento se basa en que:
  - La propagación del carry al sumar los bits *i* puede saltearse si $x_i \neq y_i$ ($P_i = x_i \oplus y_i = 1$)
  - La propagación del carry puede saltearse la suma de varios bits consecutivos si todos satisfacen $x_i \neq y_i$.
- Un sumador de *n* bits puede dividirse en grupos de bits consecutivos.
- Cada grupo suma los *m* bits con un esquema ripple.

Carry-skip adder

- El cálculo de P es más simple que el cálculo de G.
- Cada grupo calcula sólo P_m
  - P_m = 1 si el grupo propaga el carry de entrada. De esta forma, permite que al carry de entrada saltear el grupo.
- Si un bloque genera carry, habrá carry de salida aunque el carry de entrada no sea el correcto.
- Si todos los carry de entrada a los grupos se inicializan en cero, el carry de salida puede interpretarse como G.
  El carry-skip adder es práctico sólo si esto puede hacerse de manera fácil al comienzo de cada operación.

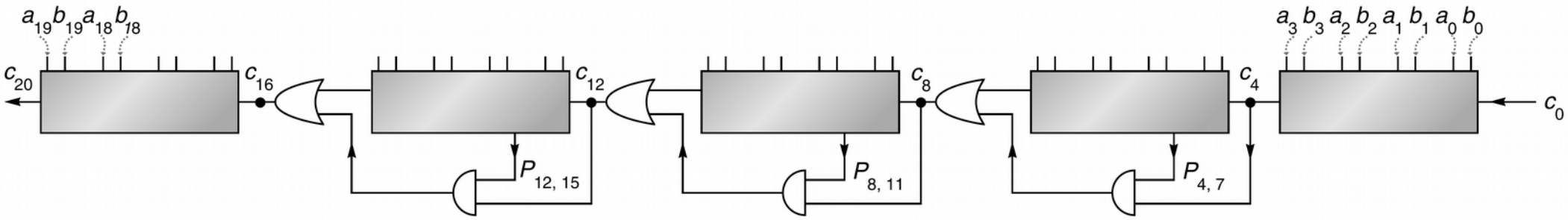
Carry-skip adder

$$C_{i+1} = G_{i:j} + P_{i:j} \cdot C_j$$



Carry-skip adder

- El carry-out del bloque *i* es el carry-in del bloque *i*+1.
- Luego, si el bloque *i*+1 propaga carry ($P_{i+1} = 1$) y hay carry-in ($C_{i+1} = 1$), ya se sabe que habrá carry-in en el bloque *i*+2 ($C_{i+2} = 1$).



Tamaño de grupos óptimo

Todos los grupos de igual tamaño *k* y *n* múltiplo de *k*

- *k* debe minimizar la cadena de propagación de carry más larga.
- La propagación más larga se da cuando el carry se genera en el bloque 1 (ripple), luego se propaga por los bloques 2,3,…,*n*/*k*-1 y finalmente hace ripple en el bloque *n*/*k*.

$$T = t_{\text{Ripple en k bits}} + (n/k - 2)\,(t_{AND} + t_{OR}) + t_{\text{Ripple en k bits}}$$

Tamaño de grupos óptimo

$$T = \underbrace{t_{\text{Ripple en k bits}}}_{\text{Primer sumador}} + \underbrace{(n/k - 2)\,(t_{AND} + t_{OR})}_{\text{Tiempo de saltear los sumadores del medio}} + \underbrace{t_{\text{Ripple en k bits}}}_{\text{Último sumador}}$$

Si cada compuerta tiene retardo Δ_G

El tiempo del ripple es $2\Delta_G \times$cantidad de bits en ripple (k)

$$T = 2k\Delta_G + (n/k - 2)\,(\Delta_G + \Delta_G) + 2k\Delta_G$$
$$= \Delta_G\,(4k + 2n/k - 4)$$

Tamaño de grupos óptimo

$$T = \Delta_G (4k + 2n/k - 4)$$

Para hallar el óptimo k, se deriva T con respecto a k y se iguala a 0.

- $dT/dk = 4\Delta_G - 2n\Delta_G/k^2 = 0$
- $k = \sqrt{(n/2)}$

El tamaño del grupo óptimo y el tiempo de propagación del carry son proporcionales a $\sqrt{n}$.

Para 32 bits, el óptimo es $k = 4$

$$T = 28\Delta_G \text{ contra } T_{ripple} = 64\Delta_G$$

Tamaño de grupos óptimo

Asumiendo bloques de diferentes tamaños

- El tiempo de saltear un sumador no depende de la cantidad de bits a sumar.
- Se puede optimizar:
  - Incrementando el tamaño de los grupos centrales
  - Reduciendo el tamaño del primer y del último grupo.

Sumadores de *n* bits

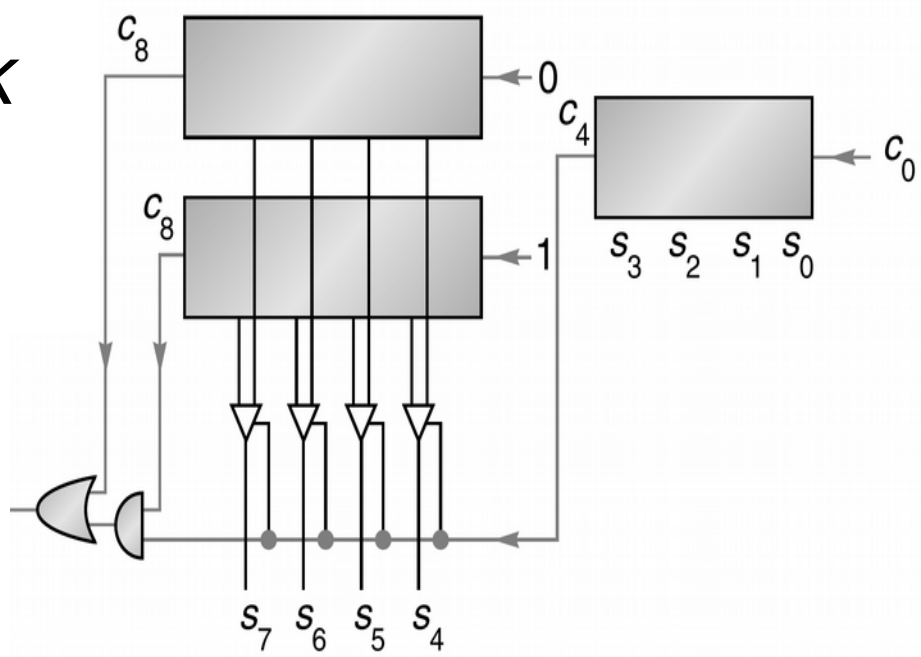
- Sumador serie
- Sumadores paralelos
  - Ripple adder
  - Carry-lookahead adder
  - Carry-skip adder
  - Carry-select adder
  - Semisumador: Carry-save adder

Carry-select adder

- Los *n* bits a sumar se dividen en grupos (pueden ser de diferentes tamaños)
- Cada grupo genera dos sumas y dos carries.
  - Una suma y un carry se corresponden a la suma con carry inicial 0.
  - La otra suma y el otro carry se corresponden a la suma con carry inicial 1.
- Cuando se tiene el carry de entrada real, se selecciona la suma con su carry correspondientes.

Carry-select adder

- Cada bloque calcula las sumas en ripple.
- Una vez que termina el primer bloque, y se tiene el carry real, se seleccionan los carries de los bloques siguientes.
- Para bloques de tamaño *k*
  - $2k\Delta_G$ para calcular todas las sumas
  - $2\Delta_G(n/k - 1)$ para seleccionar el carry correcto y multiplexar (2 niveles de compuertas) la suma.



- Total: $2k\Delta_G + 2\Delta_G(n/k - 1) = (2k + 2n/k - 2)\Delta_G$

Tamaño de grupos óptimo

- Sin embargo, para el Carry-select adder el mejor diseño es con bloques de tamaño variable:
  - Si a cada bloque le lleva $2k\Delta_G$ realizar la suma de números de k bits
  - Y seleccionar el carry correcto lleva $2\Delta_G$
  - Entonces, lo ideal es que cada bloque sea un bit más ancho que el anterior.
- Los grupos deberían seguir la serie 1,1,2,3,…

Tamaño de grupos óptimo

- Los grupos deberían seguir la serie 1,1,2,3,…
- Si L_{max} es el tamaño del mayor bloque, para *n* bits se debe satisfacer que:

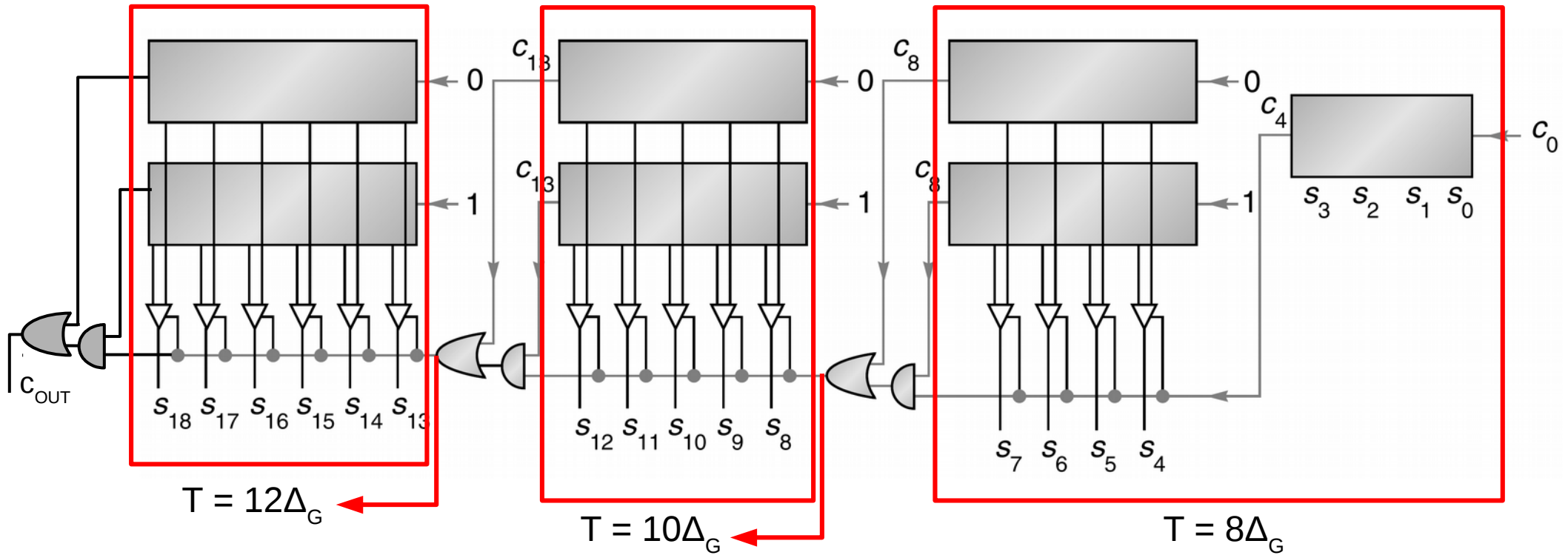
$$1 + L_{max} (L_{max} - 1) / 2 > n$$

- Por lo tanto

$$L_{max} (L_{max} - 1) > 2(n - 1)$$

- El tamaño del mayor grupo y el tiempo de ejecución son del orden de $\sqrt{n}$

Carry-select adder



T = 0 → Comienza

T = 8Δ$_G$ → Terminan el primero y el segundo bloque

T = 10Δ$_G$ → Terminan el tercer bloque y ya se calculó el carry out del segundo

T = 12Δ$_G$ → Termina el cuarto bloque y ya se calculó el carry out del tercero.
Luego del multiplexado (despreciable), ya está la suma final.

T = 14Δ$_G$ → Calculó el carry out final

Sumadores de n bits

- Sumador serie
- Sumadores paralelos
  - Ripple adder
  - Carry-lookahead adder
  - Carry-skip adder
  - Carry-select adder
  - Semisumador: Carry-save adder

Carry-save adder (CSA)

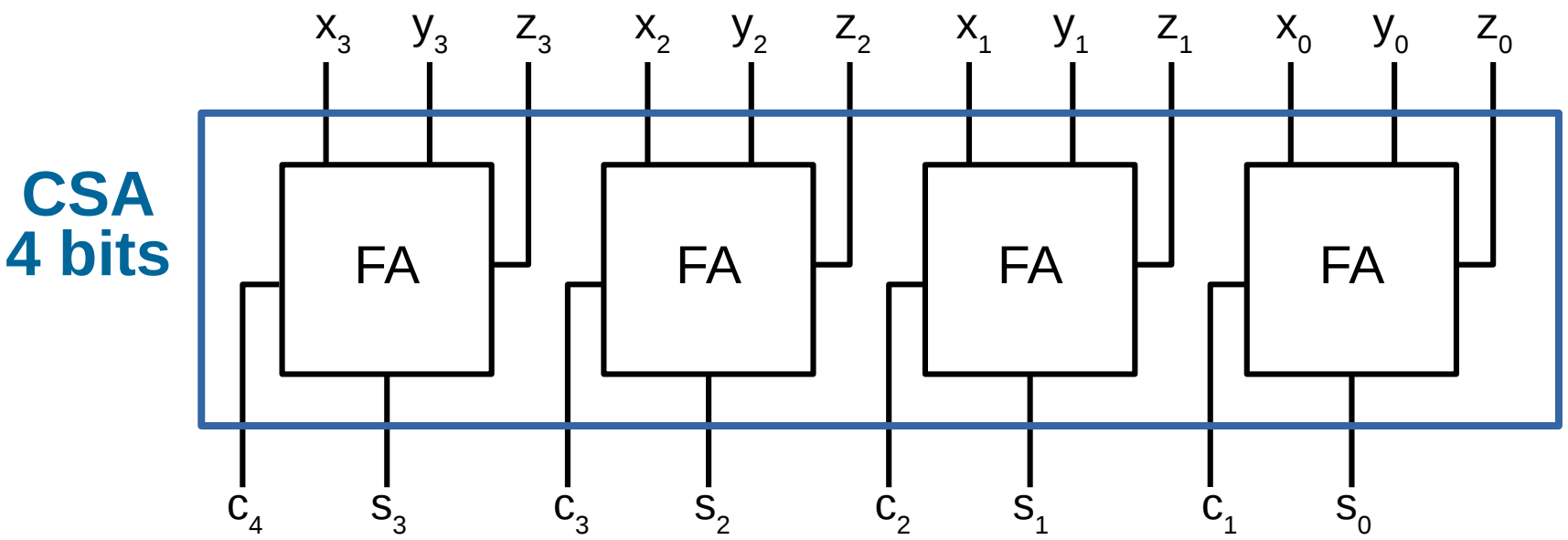
- Es un semisumador.
- Sumar más de 2 operandos simultáneamente (como en la multiplicación) usando sumadores de dos operandos es costoso:
  - Para *k* operandos la propagación del carry debe repetirse *k* – 1 veces
- La estrategia usando *Carry-save adder* reduce ese costo.
  - Generar sumas parciales y secuencias de carry
  - Propagar el carry sólo en el último paso.

Carry-save adder (CSA)

- El CSA recibe 3 operandos de n bits y genera dos resultados n bits:
  - La suma parcial de n bits
  - El carry de n bits.
- Reduce la suma de 3 operandos a la suma de 2 operandos.
- La implementación más simple es con *n* FA en paralelo.

Carry-save adder (CSA)

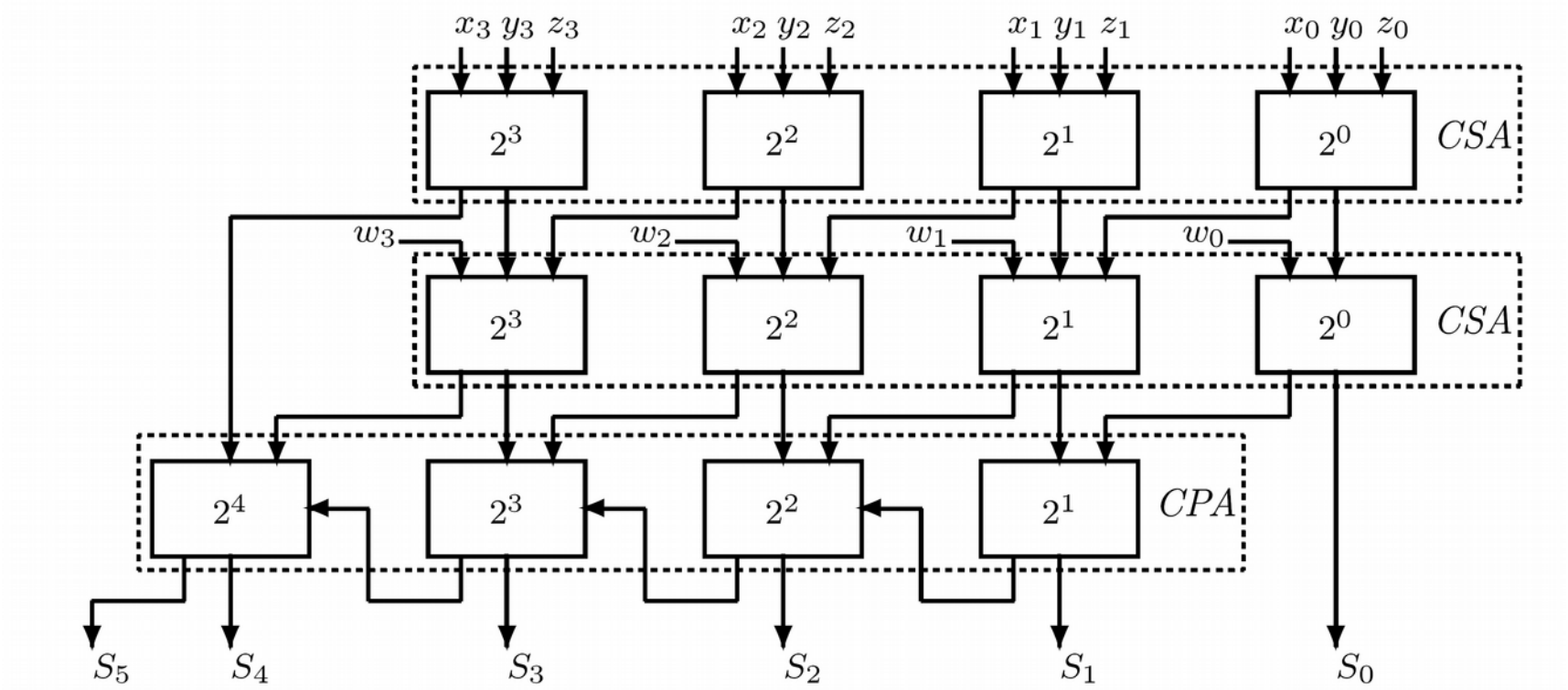
- La implementación más simple es con *n* FA en paralelo.



$$+\ s_3 s_2 s_1 s_0 \quad c_4 c_3 c_2 c_1 c_0 \quad \Rightarrow \quad +\ s_3 s_2 s_1 s_0 \quad c_4 c_3 c_2 c_1 0$$

Carry-save adder (CSA)

- Sumador de 4 operandos (*x*,*y*,*z*,*w*) de 4 bits cada uno usando dos niveles de CSA.
- El último nivel debe ser un sumador paralelo que propague carry (CPA: ripple, lookahead, skip, etc)



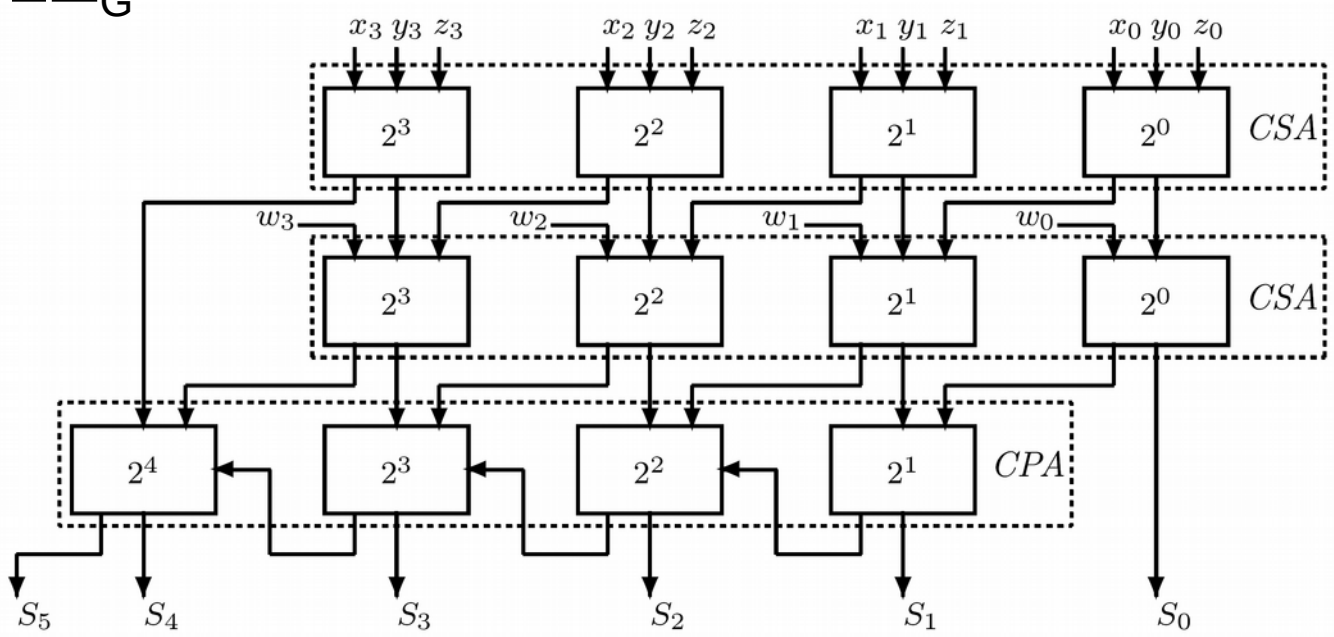
Carry-save adder (CSA)

Sumar k operandos de n bits

- Requiere k – 2 CSA y un CPA
- El tiempo de suma será:

$$(k-2)\,T_{CSA} + T_{CPA}$$

  - T_{CPA} = retardo del sumador paralelo.
  - $T_{CSA} = \Delta_{FA} = 2\Delta_G$



Carry-save adder (CSA)

La suma de k operandos de n bits, puede valer hasta $(2^n - 1)\,k$.

El resultado final puede tener hasta $n + \lceil \log_2 k \rceil$ bits

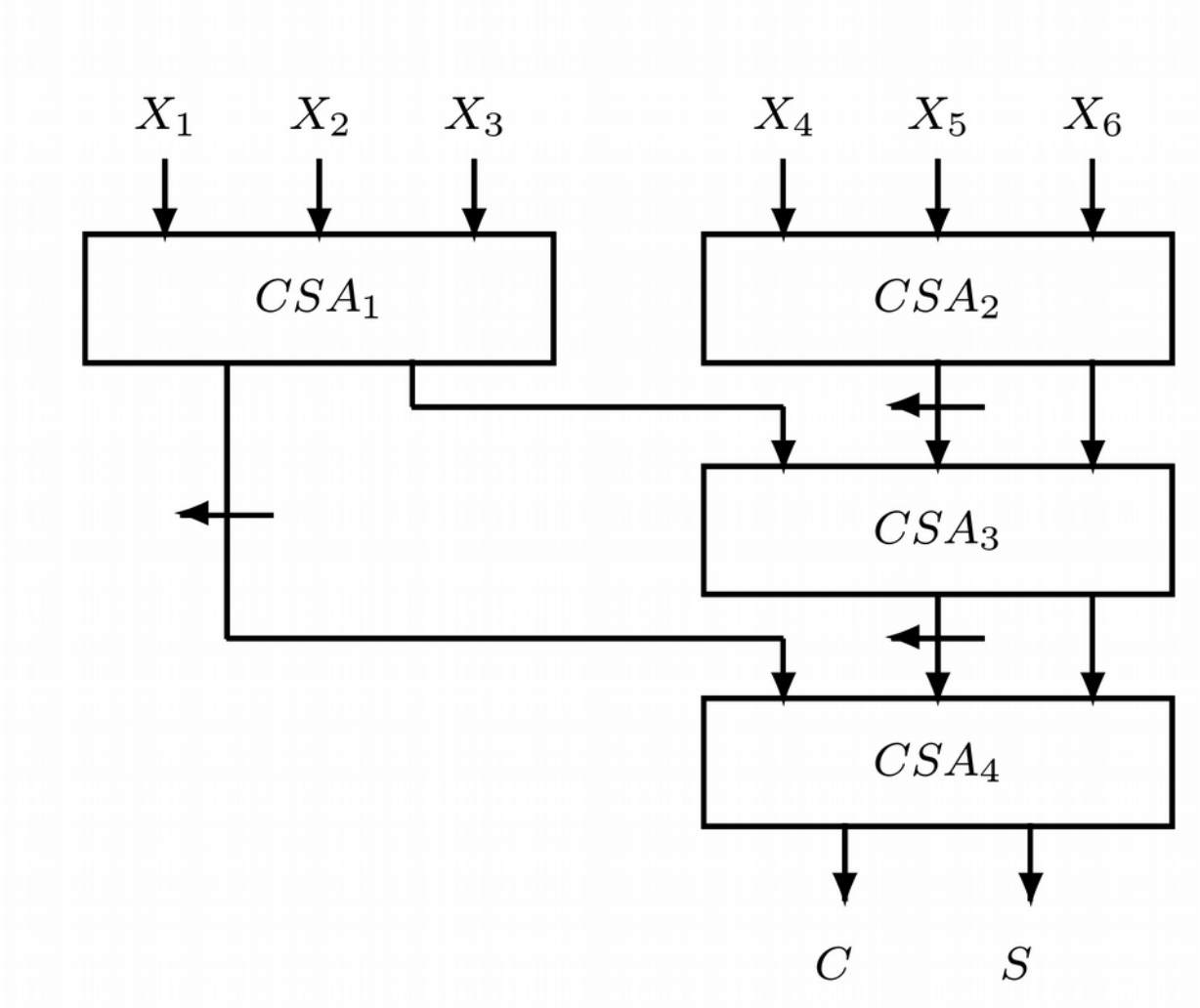
El tiempo total de suma:

$$(k - 2)\,2\Delta_G + T_{CPA}(n + \lceil \log_2 k \rceil)$$

Dependerá del sumador paralelo de la última etapa.

Árbol de CSA

- La mejor organización de CSA es el Wallace tree



Árbol de CSA

- En cada nivel, el número de operandos se reduce en ⅔.
- Por lo tanto, si L es la cantidad de niveles requeridos

$$k\,(⅔)^{L} \leq 2$$

- Es decir que se necesitarán una cantidad estimada de niveles:

$$\frac{\log \frac{2}{n}}{\log \frac{2}{3}} = \frac{\log \frac{n}{2}}{\log \frac{3}{2}}$$

El resultado final requiere atravesar O(log(n)) CSA.

Árbol de CSA

Number of operands	Number of levels
3	1
4	2
$5 \leq k \leq 6$	3
$7 \leq k \leq 9$	4
$10 \leq k \leq 13$	5
$14 \leq k \leq 19$	6
$20 \leq k \leq 28$	7
$29 \leq k \leq 42$	8
$43 \leq k \leq 63$	9

Restadores

- Los sumadores pueden sumar números signados o no signados.
- Con un sumador puede realizarse la resta usando complemento a la base.
  - Puede implementarse circuito restador que realice la operación directamente.
- Para calcular *A – B*
  1) Se complementa B : X ← B'
  2) Se calcula A + X forzando el carry inicial c_0 a 1.

Bibliografía

- *Capítulo 5*. *Computer Arithmetic Algorithms.* Israel Koren, 2da Edición, A K Peters, Natick, MA, 2002.

  Adapted from Koren, UMass. Copyright 2008 Koren, UMass and A.K. Peters.

- *Capítulo 6*. *M. Morris Mano & Michael D. Celetti. Digital Design: With an Introduction to the Verilog HDL*. Pearson. (2015, 5ta Ed.)

Suplementaria

- *Apéndice J*. J. Hennessy & D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers INC. 2011, 5ta Ed.
- *Apéndice B.* David A. Patterson & John L. Hennessy. *Computer Organization and Design. The Hardware/Software Interface.* Elsevier. (5ta Ed. 2014)