Bloque: CIRCUITOS DE PROCESADO DE DATOS

Tema 5: Aritmética en punto fijo

Tema 6: Multiplicadores, divisores y generación de funciones

Tema 7: Aritmética en punto flotante

http://www.ecs.umass.edu/ece/koren/arith/simulator

Tema 5. ARITMÉTICA EN PUNTO FIJO

- Sistemas de numeración
- Circuitos básicos para aritmética binaria
- Arquitecturas de sumadores
- Ejercicios resueltos

Tema 5. ARITMÉTICA EN PUNTO FIJO

- Sistemas de numeración
- Circuitos básicos para aritmética binaria
- Arquitecturas de sumadores
- Ejercicios resueltos

Introducción

- La razón de que estudiemos los circuitos aritméticos es que las operaciones que realizan son el núcleo del procesado de datos.
- Nuestro objetivo principal es la implementación en *hardware* de operadores aritméticas
  - → atendiendo a criterios de diseño como: velocidad, coste, simplicidad, etc.
- Campos de aplicación muy diversos y amplios:
  - CPU´s
  - Procesamiento digital de señal
  - Audio/video digital
  - Criptografía
  - Telefonía móvil

Sistema de numeración binario

- En los ordenadores digitales convencionales, los enteros se representan como números binarios de longitud fija.
- Un número binario de longitud n es una secuencia ordenada $(x_{n-1}x_{n-2}\cdots x_1x_0)$ de dígitos binarios, donde cada dígito x_i (bit) toma valores 0 o 1.
- La longitud n de la secuencia es importante ya que los números binarios se almacenan en registros de longitud fija.
- El valor numérico de la secuencia anterior viene dado por:

$$(x_{n-1}x_{n-2}\cdots x_1x_0) \leftrightarrow x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0 = \sum_{j=0}^{n-1} x_j \cdot 2^j$$

- La base (*radix*) del sistema de numeración binario es 2 (*radix*-2 *numbers*).
- Los números decimales tienen por base el 10 (*radix*-10 *numbers*).
  - ⇨ $101_{10} \rightarrow 101$ $\qquad$ $101_2 \rightarrow 5$

Sistema de numeración binario: rango de representación

- En una unidad aritmética los operandos y el resultado se almacenan en registros de longitud n, que sólo pueden almacenar un número finito de valores diferentes.
  - ⇨ marca los principales problemas a la hora de operar ya que da lugar a errores
- Llamaremos X_{max} al mayor valor representable, y X_{min} al menor.
  - ⇨ El rango de números representables será $[X_{min}, X_{max}]$.
  - ⇨ Un resultado mayor que X_{max} o menor que X_{min} se representará de forma incorrecta.
  - ⇨ La unidad aritmética debería indicar que el resultado generado está representado de forma incorrecta (indicación de *overflow*).

❖ Ejemplo: enteros sin signo representados usando 5 dígitos binarios.

- ⇨ $X_{max} = (31)_{10}$ representado por $(11111)_2$ $X_{min} = (0)_{10}$ representado por $(00000)_2$

Si incrementamos X_{max} en 1, obtenemos $(32)_{10} = (100000)_2$, pero en una representación de 5 bits sólo se retienen los últimos 5 bits, por lo que se almacena $(00000)_2 = (0)_{10}$

- ⇨ En general, si X no está en el rango $[X_{min}, X_{max}] = [0, 31]$, se almacena X mod 32.
- ⇨ Si $X + Y$ excede X_{max}, el resultado que se almacena es $(X + Y)$ mod 32

```
 X    10001    17
+Y    10010    18
    1 00011     3 = 35 mod 32
```

Representación de números en máquinas

- El sistema binario es un ejemplo específico de sistema de numeración que puede usarse para representar valores numéricos en una unidad aritmética.
- Un sistema de numeración se define por:
  - el conjunto de valores que puede tomar cada bit
  - la regla que define el mapeo entre las secuencias de dígitos y sus valores numéricos.
- Hay dos tipos de sistemas de numeración:
  - los convencionales (binario, decimal, …)
  - los no convencionales (bases negativas, dígitos con signo, …)

Sistemas de numeración: convencionales → Propiedades

- No redundante:
  - cada número tiene una única representación, lo que significa que
  - no hay dos secuencias con el mismo valor numérico.
- Pesado:
  - una secuencia de pesos $(w_{n-1}w_{n-2}\cdots w_1w_0)$ determina el valor de la secuencia $(x_{n-1}x_{n-2}\cdots x_1x_0)$ como:

$$X = \sum_{i=0}^{n-1} x_i \cdot w_i$$

  - w_i es el peso asociado a x_i, es decir, al dígito en la posición i-ésima.
- Posicional:
  - el peso w_i depende sólo de la posición i del dígito x_i.
  - en sistemas de numeración convencionales, $w_i = r^i$, por lo que se suelen denominar **sistemas de base fija** (*fixed-radix systems*)
  - el dígito x_i satisface $0 \le x_i \le r-1$ (de lo contrario se introduce redundancia)

Sistemas de numeración: convencionales de base fija

- r es la base (*radix*) del sistema de numeración
- Si no hay redundancia, $0 \leq x_i \leq r-1$
- Si $x_i \geq r$, aparece redundancia en el sistema de numeración de base fija

$$x_i r^i = (x_i - r) r^i + 1 \cdot r^{i+1}$$

  - ⇨ hay dos representaciones para el mismo valor:

$$(\cdots, x_{i+1}, x_i, \cdots) \text{ y } (\cdots, x_{i+1}+1, x_i - r, \cdots)$$

- Una secuencia de n dígitos puede representar un número con parte entera (k dígitos) y fraccionaria (m dígitos), donde $n = k + m$
- El valor de una secuencia con un "*punto*" entre los k bits más significativos y los m menos significativos:

$$\left(\underbrace{x_{k-1}x_{k-2}\cdots x_1 x_0}_{\text{parte entera}} . \underbrace{x_{-1}x_{-2}\cdots x_{-m}}_{\text{parte fracc.}}\right)_r$$

$$X = x_{k-1}r^{k-1} + x_{k-2}r^{k-2} + \cdots + x_1 r^1 + x_0 r^0 + x_{-1}r^{-1} + \cdots + x_{-m}r^{-m} = \sum_{i=-m}^{k-1} x_i \cdot r^i$$

Sistemas de numeración: conversión entre bases

- Pretendemos convertir la representación de un número en base r en otra representación, ahora en base $R \neq r$.

$$\left(\underbrace{x_{k-1}x_{k-2}\cdots x_1 x_0}_{\text{parte entera}} . \underbrace{x_{-1}x_{-2}\cdots x_{-m}}_{\text{parte fracc.}}\right)_r \rightarrow \left(\underbrace{X_{K-1}X_{K-2}\cdots X_1 X_0}_{\text{parte entera}} . \underbrace{X_{-1}X_{-2}\cdots C_{-M}}_{\text{parte fracc.}}\right)_R$$

- Para ello, lo más conveniente suele ser pasar por la base 10.

Parte entera: $(105)_{10} = (?)_5$

Dividimos por 5	Cociente	Resto
	105	0
	21	1
	4	4
	0	

$(105)_{10} = (410)_5$

Parte fraccionaria: $(105.486)_{10} = (410.?)_5$

Multiplicamos por 5	Parte entera	Fracción
		.486
0.486 × 5 = 2.430	2	.430
0.430 × 5 = 2.150	2	.150
…	0	.750
	3	.750
	3	.750

$(105.486)_{10} \cong (410.22033)_5$

Sistemas de numeración: representaciones de punto fijo

- El peso r^{-m} del dígito menos significativo se denomina ***ulp*** (*unit in the last position*)
- El "*punto*" no se almacena en el registro, sino que se sabe dónde está colocado: entre los k bits más significativos y los m menos significativos:
  → estas son las **representaciones en punto fijo**
- Esto no supone ninguna limitación ya que los operandos pueden escalarse (el mismo para todos)
  - las operaciones de suma y resta son correctas:
    $a \cdot X \pm a \cdot Y = a(X \pm Y)$, donde a es el factor de escalado
  - se requieren correcciones para la multiplicación y la división
    $(a \cdot X) \cdot (a \cdot Y) = a^2 X \cdot Y$ $(a \cdot X) / (a \cdot Y) = X / Y$
- posiciones más comunes para el "punto":
  - lo más a la derecha posible, lo que nos da enteros puros ($m = 0$)
  - lo más a la izquierda posible, lo que nos da fracciones puras ($k = 0$)

Sistemas de numeración: números negativos

- Consideraremos **representaciones en punto fijo en un sistema de base r.**
- Dos formas básicas para representarlos:
  - Signo-magnitud (o *signed-magnitude representation*)
  - Representación por complementos, con dos alternativas
    - complemento a la base (complemento a 2 en el sistema binario)
    - complemento a la base disminuida en 1 (complemento a 1 en el sistema binario)

Representación de números negativos: signo-magnitud

- **Signo-magnitud**:
  - El signo y la magnitud se representan en forma separada

$$\underbrace{x_s}_{\text{signo}}\ \underbrace{x_{n-2}\ x_{n-3}\ \cdots\ x_2\ x_1\ x_0}_{\text{magnitud}}$$

donde x_s es el dígito de signo y $x_{n-2} \cdots x_1 x_0$ la magnitud ($n-1$ bits)

  - en general,
    0: positivo, $r-1$: negativo.
  - En el caso binario, 1 indica negativo

→ Sólo se usan $2r^{n-1}$ de las r^n posibles secuencias

→ Hay dos representaciones posibles para el 0: en binario, por ejemplo, `0000...00` y `1000...00`.

→ En binario, el rango es $[-(2^{n-1}-1), +(2^{n-1}-1)]$.

Representación de números negativos: signo-magnitud

- **Desventajas de la representación signo-magnitud**:
  - dado que hay dos representaciones para el cero, en un testado del mismo se deben chequear las dos.
  - La operación puede depender de los signos de los operandos:
    - → Imaginemos la suma de un número positivo X y otro negativo, $-Y$: $X + (-Y)$
      - Si $Y > X$, el resultado final es $-(Y - X)$
        Cálculo:
        - a) se conmuta el orden de los operandos
        - b) se realiza una substracción, en vez de una suma
        - c) se añade el signo menos
  - → Por tanto, hay que tomar una secuencia de decisiones que tienen su coste en lógica de control y tiempo de ejecución
  - Todo ello se evita utilizando la representación mediante complementos.

Representación de números negativos: complementos

- **Complementos**
  - Dos alternativas
    - complemento a la base (complemento a 2 en el sistema binario)
    - complemento a la base disminuida en 1 (complemento a 1 en el sistema binario)
  - En ambas, los números positivos se representan como en signo-magnitud.
  - Un número negativo $(-Y)$ se representa por $(R - Y)$, donde R es una constante.
  - Esta representación satisface $-(-Y) = Y$ ya que $R - (R - Y) = Y$.

Tema 5: Aritmética en punto fijo

Representación de números negativos: complementos

- Entre las **ventajas** de esta representación:
  - $X - Y = X + (-Y)$, y $-Y$ se representa como $(R - Y)$.

  a) No hay que decidir antes de realizar una adición o una substracción.
  - la suma se realiza como $X + (R - Y) = R - (Y - X)$
  - si $Y > X$, $-(Y - X)$ ya está representado como $R - (Y - X)$

  b) No hay necesidad de intercambiar el orden de los operandos.
  - Si $Y < X$, la suma se realiza como $X + (R - Y) = R + (X - Y)$, en vez de $(X - Y)$, por lo que debemos eliminar el R adicional.
  - Otro requerimiento sería que el cálculo del complemento debería ser simple y poderse realizar a gran velocidad.
- ¿Qué se requiere para elegir R? Definimos:
  - complemento de un dígito $\bar{x}_i$: $\bar{x}_i = (r - 1) - x_i$
  - complemento de la secuencia de n dígitos (n-tupla) X:

$$X = (x_{k-1}, x_{k-2}, \cdots, x_{-m}) \qquad \bar{X} = (\bar{x}_{k-1}, \bar{x}_{k-2}, \cdots, \bar{x}_{-m})$$

Representación de números negativos: complementos

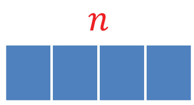
- Consideremos nuestra representación en complementos y sumemos X y $\bar{X}$:

X		x_{k-1}	x_{k-2}	$\cdots$	x_{-m}	
$+\bar{X}$		$\bar{x}_{k-1}$	$\bar{x}_{k-2}$	$\cdots$	$\bar{x}_{-m}$	
		$(r-1)$	$(r-1)$	$\cdots$	$(r-1)$	
$+ulp$					1	
	1	0	0	$\cdots$	0	$= r^k$

⇨ $\boxed{X + \bar{X} + ulp = r^k}$

- ❖ Con el resultado almacenado en un registro de longitud n $(= k + m)$, se descartaría el dígito más significativo, por lo que el resultado sería cero.
- ❖ En general, almacenar el resultado de cualquier operación aritmética en un registro de longitud fija equivale a quedarnos con el resto después de dividir por r^k.
- ❖ Reordenando el resultado anterior: $r^k - X = \bar{X} + ulp$, con lo que, si seleccionamos R como r^k, obtenemos: $R - X = \bar{X} + ulp$
- ⇨ El cálculo del complemento $(R - X)$ para X es muy simple e independiente de k. Esta representación se denomina complemento a la base (*radix-complement*).
  - ⇨ No se necesita corrección cuando el resultado $X + (R - Y)$ es positivo $(X > Y)$ ya que $R = r^k$ se descarta al calcular $R + (X - Y)$.

Representación de números negativos: ejemplos



➔ **Ejemplo 1**: Considere $r = 2$, $k = \underbrace{n}_{=k+m} = 4 \rightarrow m = 0$, $ulp = 2^0 = 1$.

- ⇨ el complemento a la base (que se llama, en este caso, complemento a 2, Ca2) de un número X es $2^4 - X$.
- ⇨ en vez de calcularlo así, usaremos $\bar{X} + ulp$.
- ⇨ las secuencias `0000` a `0111` representan los números 0_{10} a 7_{10}, respectivamente.
- ⇨ el complemento a 2 del mayor número positivo es `1000+1`, y representa $(-7)_{10}$
- ⇨ el complemento a 2 de cero es `1111+1 = 10000 = 0 mod` 2^4.
- ⇨ cada número positivo tiene un negativo correspondiente que comienza por `1`.
- ⇨ hay una secuencia adicional que comienza por `1000`, que representa a $(-8)_{10}$ y que no tiene contrapartida positiva.
- ⇨ el rango de números binarios que puede obtenerse con $k = n = 4$ es $-8 \le X \le 7$.

Representación de números negativos: complementos

- En resumen, y para $r = 2$

$$b_s b_{n-2} \cdots b_2 b_1 b_0$$

donde b_s: bit de signo
(0: positivo, 1: negativo)

$b_{n-2} \cdots b_2 b_1 b_0$: magnitud ($n-1$ bits)

→ el complemento a 1 con signo de $-N$ ($N > 0$) se obtiene complementando todos los bits de $+N$ incluyendo el bit de signo.

→ el complemento a 2 con signo de $-N$ ($N > 0$) se obtiene complementando todos los bits de $+N$ incluyendo el bit de signo y sumando 1.

- Misma representación si $N \geq 0$ con 0 en msb.
- Una sola representación para 0 en comp. a 2.
- En todos los negativos hay un 1 en msb.

Dec.	Signed 2's Comp.	Signed 1's Comp.	Sign Mag.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
–0	—	1111	1000
–1	1111	1110	1001
–2	1110	1101	1010
–3	1101	1100	1011
–4	1100	1011	1100
–5	1011	1010	1101
–6	1010	1001	1110
–7	1001	1000	1111
–8	1000	—	—

Representación de números negativos: ejemplos

➔ **Ejemplo 1**: Represente 26_{10} en complemento a 1 (Ca1) y complemento a 2 (Ca2).

Para ello,

26_{10} = 11010
= 011010

⇨ 26_{10} = $011010_{(Ca1}$
⇨ 26_{10} = $011010_{(Ca2}$

➔ **Ejemplo 2**: Represente $(-26)_{10}$ en complemento a 1 y complemento a 2.

26_{10} = 11010
= 011010
Comp. = 100101

⇨ $(-26)_{10}$ = $100101_{(Ca1}$
⇨ $(-26)_{10}$ = $100110_{(Ca2}$

➔ **Ejemplo 3**: Represente $(-26)_{10}$ en complemento a 1 y a 2 usando 8 bits.

26_{10} = 0001 1010
Comp. = 1110 0101

⇨ $(-26)_{10}$ = $(1110\ 0101)_{(Ca1}$
⇨ $(-26)_{10}$ = $(1110\ 0110)_{(Ca2}$

Representación de números negativos: operación

- **Resumen:** El método directo de substracción, tal como lo usamos con lápiz y papel, es poco eficiente cuando se implementa en *hardware*.
  - En este caso, se emplean los complementos.

→ substracción de dos números sin signo, M y N, $(M-N)$, con complemento a r:

1) Añada M al complemento a r de N: $M+(r^n-N) \longrightarrow M-N+r^n$.
2) Si $M \geq N$, la suma produce un acarreo final (r^n) que debe eliminarse.
3) Si $M < N$, la suma no produce un acarreo final y es igual a $r^n-(N-M)$, que es el complemento a r de $(N-M)$.
   Para obtener la respuesta en una forma más familiar, obtenemos el complemento a r de la suma y colocamos un signo menos delante.

Ejemplo 1:

```
M = 37  =  100101(2;     N = 28 = 011100(2; M > N
N(Ca2 =  100011+000001 =  100100
Suma = 100101+ 100100 = 1001001  (end carry = 1)
Descarte 2^6 =  001001;           M - N  =  001001 = 9(10

M = 28; N = 37; M < N
N(Ca2     →  0100101 → 1011010+0000001 = 1011011
Suma:        0011100+1011011 = 1110111 → 0001000+0000001 → -9(10
```


Representación de números negativos: operación

- **Resumen:** El método directo de substracción, tal como lo usamos con lápiz y papel, es poco eficiente cuando se implementa en *hardware*.
  - En este caso, se emplean los complementos.

→ substracción de dos números sin signo, M y N, con complemento a $(r-1)$:

1) Añada M al complemento a $(r-1)$ de N: $M+(r^n-1-N) \longrightarrow M-N-1+r^n$.
2) Si $M \geq N$, la suma $(M-N-1)$ produce un acarreo final que debe eliminarse.
3) Si $M < N$, la suma no produce un acarreo final y es igual a $r^n-(N-M+1)$, que es el complemento a $(r-1)$ de $(N-M)$.
   Para obtener la respuesta en una forma más familiar, obtenemos el complemento a $(r-1)$ de la suma y colocamos un signo menos delante.

Representación de números con signo

- **Comparación**
  - → La representación signo-magnitud es la que se emplea de forma natural, pero es ineficiente en ordenadores debido al uso separado del signo y la magnitud.
  - → La representación mediante complementos es la que se suele utilizar:
  - → El complemento a 1
    - es útil como operación lógica ya que el intercambio entre ceros y unos es equivalente a una operación lógica, aunque raramente se usa para operaciones aritméticas.
    - presenta dos ceros, al igual que la representación signo-magnitud.
    - rango simétrico: $[-(2^{n-1}-1), +(2^{n-1}-1)]$
  - → El complemento a 2
    - las operaciones aritméticas se realizan de forma eficiente
    - presenta un cero.
    - rango asimétrico: $[-2^{n-1}, +(2^{n-1}-1)]$

Representación de números en máquinas

- Otras representaciones binarias:
  - ⇨ Representaciones con posiciones de peso redundante
  - ⇨ Representaciones con posiciones de peso con signo (GSD: *Generalized signed-bit*)
  - ⇨ Representaciones Logarítmicas
  - ⇨ Representaciones RNS (*Residue Number Systems*)
- En adelante nos centraremos en las representaciones clásicas de números en punto fijo para presentar sus circuitos aritméticos.

Sistemas de numeración: no convencionales (1)

➔ **Ejemplo 1**: ternario balanceado:

$b = 3$, y el conjunto de dígitos es $\{-1, 0, 1\} \rightarrow \{\mathtt{T}, 0, 1\}$

puede representar todos los enteros sin que se necesite una posición para el signo menos, "–".

$$10\mathtt{T}_{bal3} = 1 \times 3^2 + 0 \times 3^1 + (-1) \times 3^0 = 8_{(10}$$

y las fracciones

$$1.\mathtt{T}1_{bal3} = 1 \times 3^0 + (-1) \times 3^{-1} + 1 \times 3^{-2}$$

$$1.\mathtt{T}1_{bal3} = 1 + (-1) \times \frac{1}{3} + 1 \times \frac{1}{9} = \frac{5}{9}$$

$$0.\bar{1}_{bal3} = 0 + \frac{1}{3} + \frac{1}{3^2} + \frac{1}{3^3} + \cdots = \frac{\frac{1}{3}}{1 - \frac{1}{3}} = \frac{1}{2}$$

$$1.\bar{\mathtt{T}}_{bal3} = 1 \times 3^0 - \frac{1}{3} - \frac{1}{3^2} - \frac{1}{3^3} + \cdots = 1 - \frac{1}{2} = \frac{1}{2}$$

Dec	Bal3	Expansi	Dec	Bal3	Expansion
0	0				
1	1		−1	T	−1
2	1T	+3	−2	T1	−3+1
3	10		−3	T0	−3
4	11	+3	−4	TT	−3−1
5	1TT	+9−3	−5	T11	−9+3+1
6	1T0	+9	−6	T10	−9+3
7	1T1	+9−3	−7	T1T	−9+3−1
8	10T	+9	−8	T01	−9+1
9	100		−9	T00	−9
10	101	+9	−10	T0T	−9−1
11	11T	+9+3	−11	TT1	−9−3+1
12	110	+9	−12	TT0	−9−3
13	111	+9+3	−13	TTT	−9−3−1

Sistemas de numeración: no convencionales (2)

➔ **Ejemplo 2**: sistemas de numeración de base negativa

$r = -k$, y el conjunto de dígitos es $\{0, 1, \cdots, k-1\}$

también puede representar todos los enteros sin que se necesite una posición para el signo menos, pero aumenta la complejidad de las operaciones aritméticas.

Si $k = 2$, → Negabinario

Cada entero M puede escribirse de forma unívoca como

$$M = \sum_{j=0}^{n-1} x_j \cdot (-r)^j$$

Dec.	Negabin.
−10	1010
−9	1011
−8	1000
−7	1001
−6	1110
−5	1111
−4	1100
−3	1101
−2	0010
−1	0011
0	0000
1	0001
2	0110
3	0111
4	0100
5	0101

Sistemas de numeración: no convencionales (3)

➔ **Ejemplo 3**: sistema de numeración de base $2i$
$b = 2i$, y el conjunto de dígitos es $\{0,1,2,3\}$
(*Quater-imaginary number system*).

que nos permite representar números complejos también.

→ Por ejemplo, 1101_{2i} corresponderá a

$$\sum_{j=0}^{3} x_j \cdot (2i)^j = 1 \cdot (2i)^3 + 1 \cdot (2i)^2 + 0 \cdot (2i)^1 + 1 \cdot (2i)^0 =$$

$$= -8i - 4 + 0 + 1 = -3 - 8i$$

→ Otro caso, 1030003_{2i}

$$1030003_{2i} = 1 \cdot (2i)^6 + 3 \cdot (2i)^4 + 3 \cdot (2i)^0 = -64 + 3 \cdot 16 + 3$$

$$1030003_{2i} = -13$$

Dec.	Base $2i$
−12	300
−11	301
−10	302
−9	303
−8	200
−7	201
−6	202
−5	203
−4	100
−3	101
−2	102
−1	103
0	000
1	001
2	002
3	003

Tema 5. ARITMÉTICA EN PUNTO FIJO

- Sistemas de numeración
- Circuitos básicos para aritmética binaria
- Arquitecturas de sumadores
- Ejercicios resueltos

Representación en punto fijo: aritmética

- Los números en punto fijo pueden tratarse como números enteros que conservan la posición del punto fraccionario en un lugar prefijado (fijo en la suma/resta, como suma de los bits fraccionarios en la multiplicación, etc.)

```
  0000.1000
+ 0000.1000
= 0001.0000
```

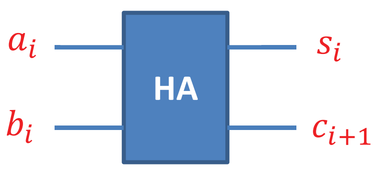
```
      1 0.1 1
    × 0 1 1
      1 0 1 1
    1 0 1 1
  1 0 0 0.0 1
```

- En adelante trataremos con palabras de "n" bits, fijando la posición como si todos ellos fueran parte entera

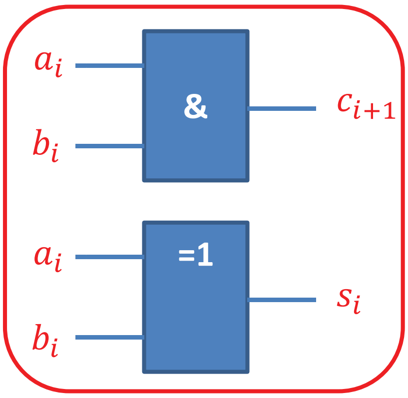
Aritmética binaria: semisumador

- En aritmética binaria la suma de dos bits, $a_i + b_i$, es:
  - $\longrightarrow$ $a_i + b_i = (c_{i+1} s_i)_2$, siendo c_{i+1} el acarreo y s_i la suma
  - $\longrightarrow$ El circuito que suma dos bits es el *Half Adder* (**HA**) o semisumador.

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



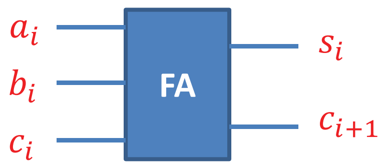
$$\begin{cases} c_{i+1} = a_i \cdot b_i \\ s_i = a_i \oplus b_i \end{cases}$$



Aritmética binaria: sumador completo

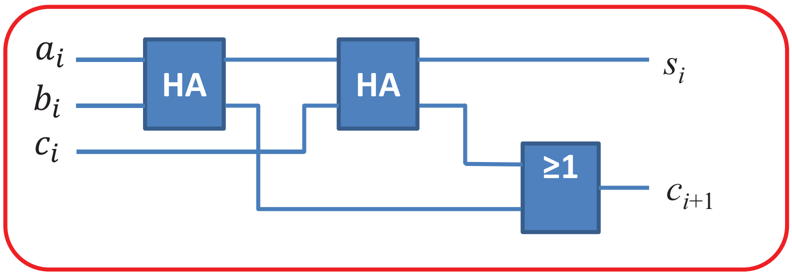
- Salvo en la columna LSB (*Least Significative Bit*), al sumar 2 datos binarios hay que sumar 3 bits: uno de cada sumando y el acarreado desde la columna anterior. Esta suma es:
  - → $a_i + b_i + c_i = (c_{i+1}s_i)_2$, siendo c_{i+1} el acarreo y s_i la suma
  - → El circuito correspondiente es el *Full Adder* (**FA**) o sumador completo.

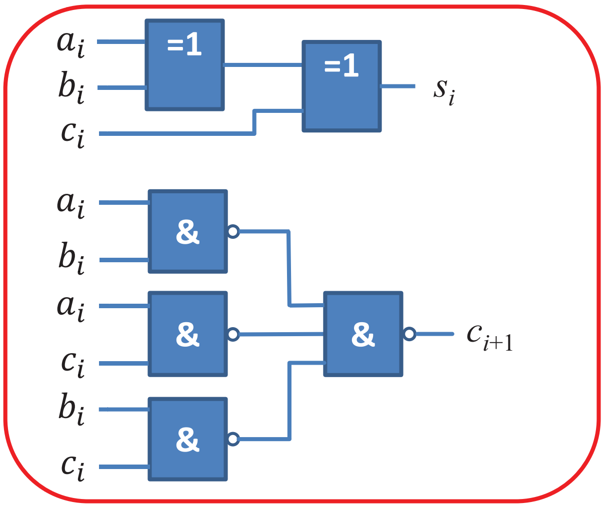
a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

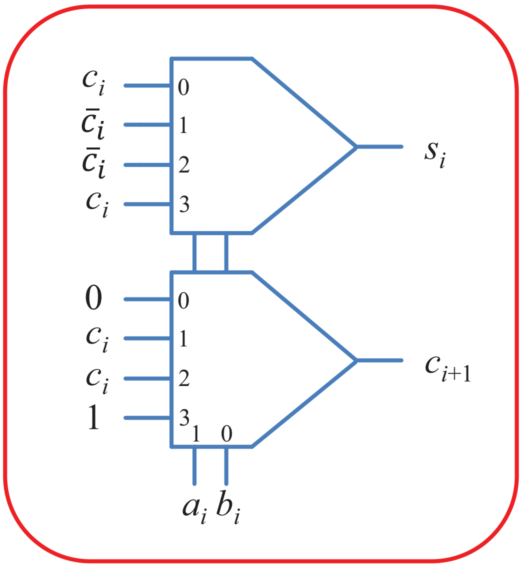


$$\begin{cases} c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i \\ s_i = a_i \oplus b_i \oplus c_i \end{cases}$$

Aritmética binaria: realizaciones del FA





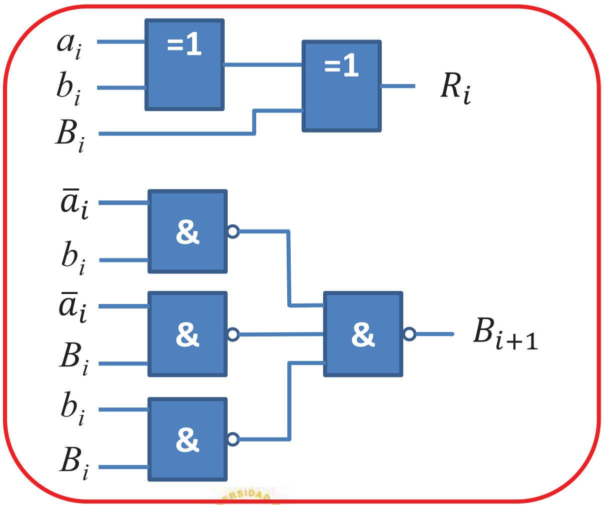


Aritmética binaria: restador

- Se trata de un circuito que recibe tres entradas y genera 2 salidas (resta y *borrow*).

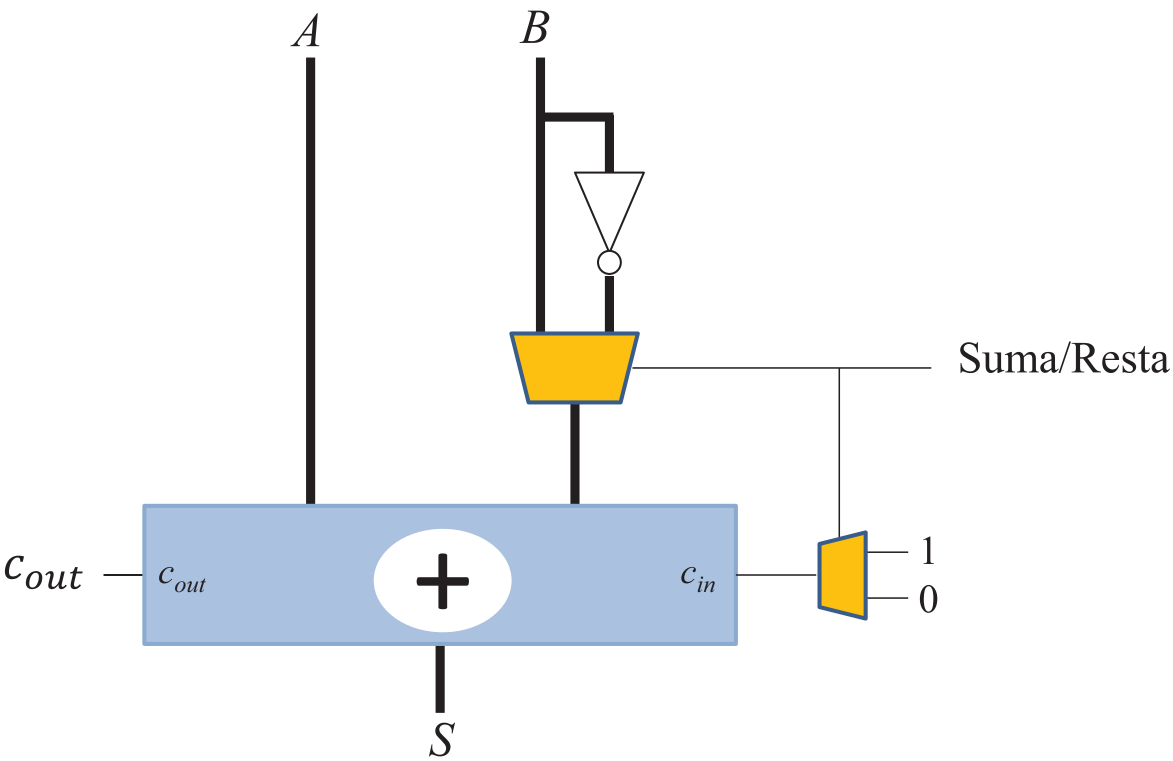
a_i	b_i	B_i	R_i	B_{i+1}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

a_i: minuendo (pos. i)
b_i: substraendo (pos. i)
B_i: *borrow-in* de la etapa siguiente (pos. i)
R_i: resta (pos. i)
B_{i+1}: *borrow-out* para la etapa siguiente (pos. i)



$$\begin{cases} B_{i+1} = \bar{a}_i \cdot b_i + \bar{a}_i \cdot B_i + b_i \cdot B_i \\ R_i = a_i \oplus b_i \oplus B_i \end{cases}$$

Sumador/restador

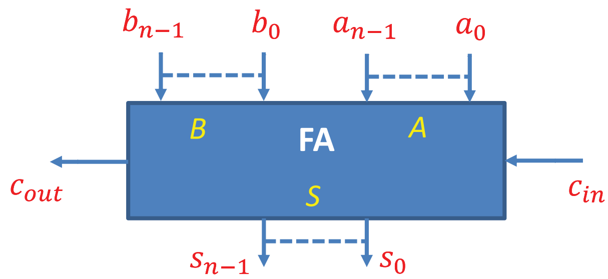


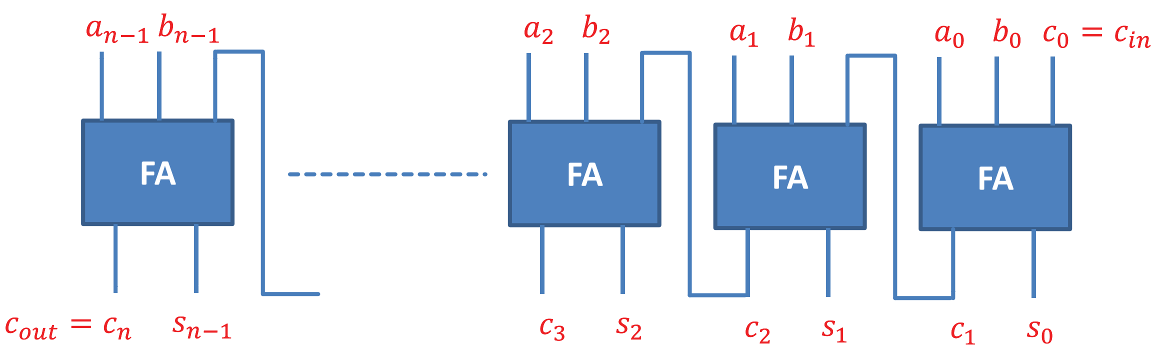
Tema 5. ARITMÉTICA EN PUNTO FIJO

- Sistemas de numeración
- Aritmética binaria
- Arquitecturas de sumadores
- Ejercicios resueltos

Sumador serie (RCA, *Ripple Carry Adder*)

- Los $2n$ bits de entrada están disponibles simultáneamente.

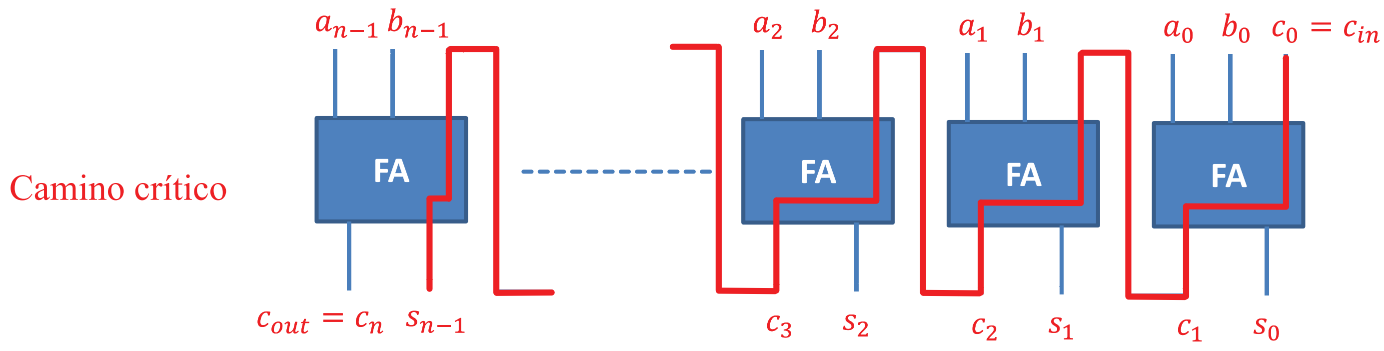




- c_{out} marca el desbordamiento (n bits) y el propio valor s_n

  Es lento debido a la propagación serie del acarreo

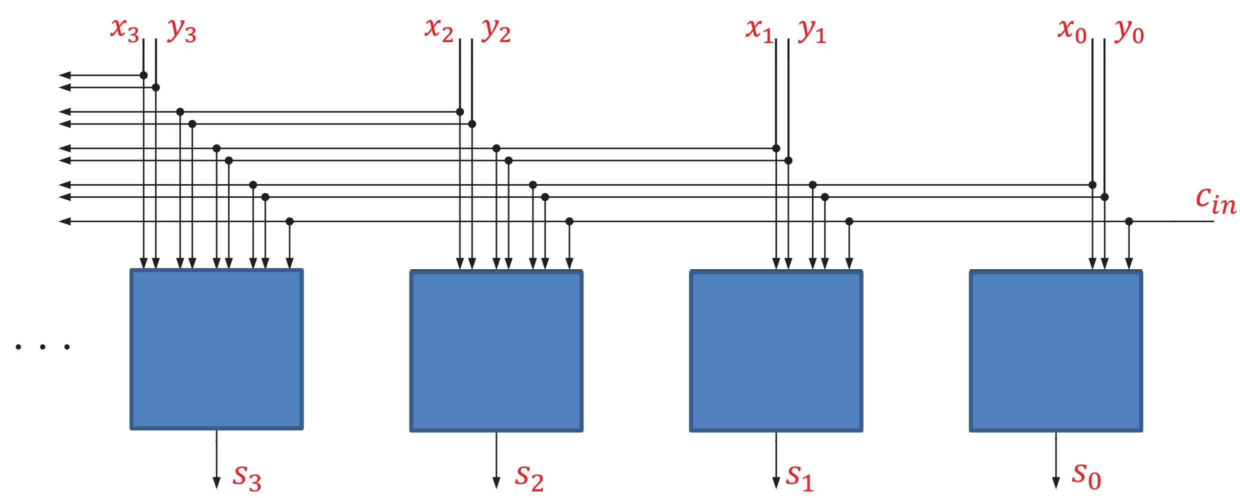
Sumador serie (RCA, *Ripple Carry Adder*)



- Necesitamos esperar hasta que los acarreos atraviesen los n FAs antes de dar validez a la suma
  - → $T_{ripple\ add} = T_{FA}(a_0, b_0 \to c_1) + (n-2) \cdot T_{FA}(c_{in} \to c_{out}) + T_{FA}(c_{in} \to s_{n-1}) \cong n \cdot T_{FA}$.
  - → el FA_i considera que su $c_i = 0$ al comienzo de la operación (circuito combinacional), y producirá una suma s_i acorde a sus entradas.
  - → el c_i correcto tardará en llegar, y producirá, en su caso, un cambio en s_i.
  - → En operaciones de suma, $c_0 = 0$, y el FA_0 pasa a ser un HA.
  - → Para implementar la resta en complemento a 2, se deja el FA_0 y se fuerza $c_0 = 1$.

Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)

- En teoría es posible derivar los bits de suma a partir de las de las que directamente dependan.



Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)

- Es el esquema más común para mejorar la velocidad (*carry-look-ahead*):
  - → se generan en paralelo todos los acarreos y ello es posible dado que dependen sólo de las entradas, que están disponibles simultáneamente.
  - → utiliza las funciones llamadas
    - de generación (g_i) y
    - de propagación de acarreo (p_i).
      - Usualmente se implementa como $p_i = a_i + b_i$
        - La OR es más "barata" que la EXOR
        - $c_{i+1} = g_i + p_i \cdot c_i$, pero $s_i \neq p_i \oplus c_i$

$$\boxed{\begin{cases} g_i = a_i \cdot b_i \\ p_i = a_i \oplus b_i \end{cases}}$$

$$\begin{cases} s_i = p_i \oplus c_i \\ c_{i+1} = g_i + p_i \cdot c_i \end{cases}$$

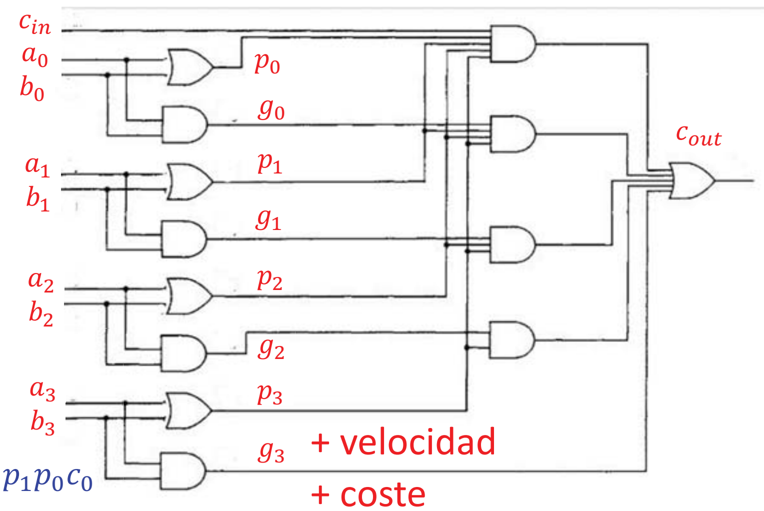
$$\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0 \\
c_3 &= g_2 + p_2 \cdot c_2 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
c_4 &= g_3 + p_3 \cdot c_3 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \\
&\dots
\end{aligned}$$

Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)

CLA de 4 bits

$$\begin{aligned}
g_0 &= a_0 b_0 & p_0 &= a_0 + b_0 \\
g_1 &= a_1 b_1 & p_1 &= a_1 + b_1 \\
g_2 &= a_2 b_2 & p_2 &= a_2 + b_2 \\
g_3 &= a_3 b_3 & p_3 &= a_3 + b_3
\end{aligned}$$

$$\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 g_0 + p_1 p_0 c_0 \\
c_3 &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \\
c_4 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0
\end{aligned}$$



+ velocidad

+ coste

- Retraso: Sea Δ_G el retraso de una puerta,
  - → Δ_G es el retraso al generar todos los p_i and g_i.
  - → $2\Delta_G$ es el retraso al generar todos c_i (implementación en dos niveles)
  - → $2\Delta_G$ es el retraso al generar todos s_i (implementación en dos niveles)
    - ⇨ $5\Delta_G$ es el retraso total, independiente de n.

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

Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)

- $5\Delta_G$ es el retraso total, independiente de n.
- Si n es grande ($n = 32$), hay un buen número de puertas con alto *fan-in*.

$$c_1 = g_0 + p_0 \cdot c_0$$
$$c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$$
$$c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$$
$$\cdots\cdots\cdots$$

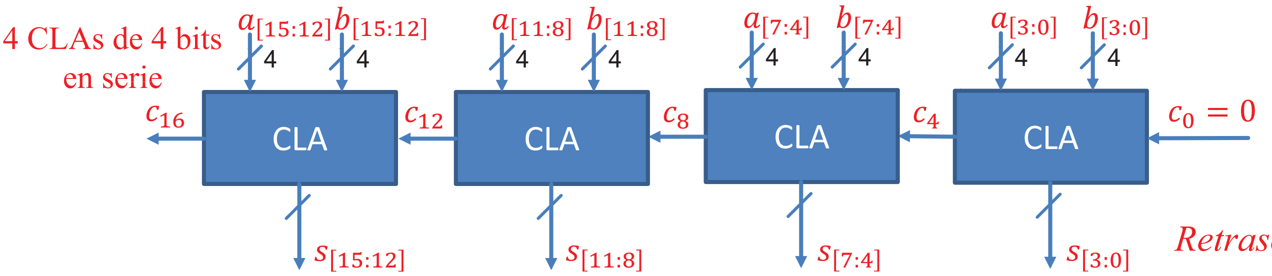
$$c_{31} = g_{30} + p_{30} g_{29} + p_{30} p_{29} g_{28} + p_{30} p_{29} p_{28} g_{27} + \cdots + \overbrace{p_{30} p_{29} p_{28} \cdots p_2 p_1 p_0 c_0}^{\text{32-input AND}}$$

. . .

32-input OR

Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)

- Sumador de 16 bits con acarreo serie entre cada sumador de cuatro bits



Retraso $= 11\Delta_G$

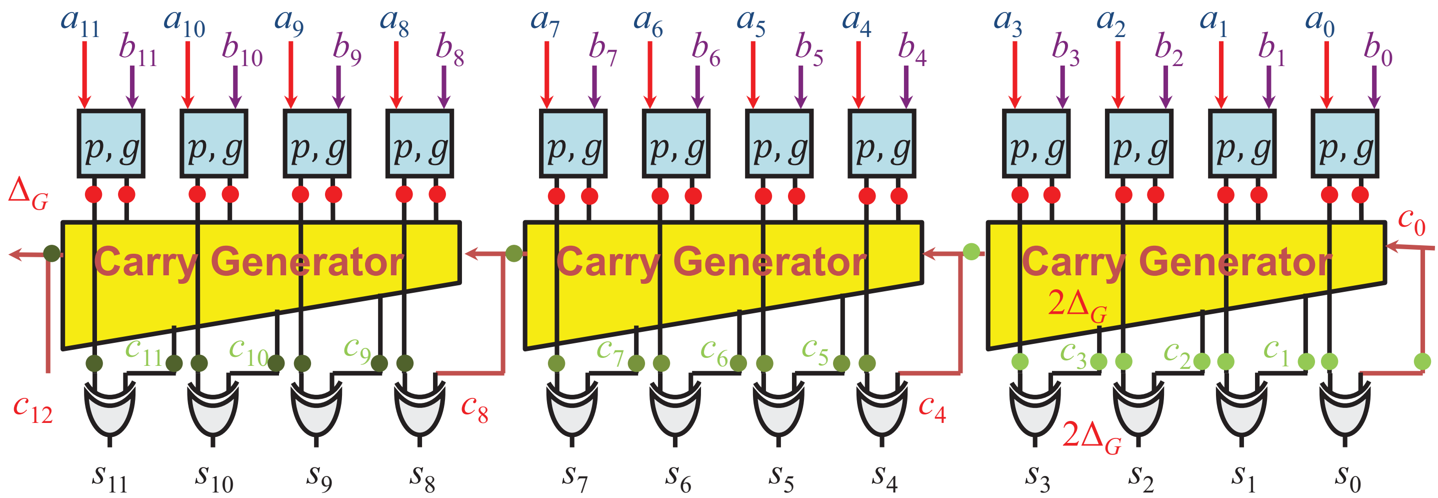
⇨ Mejoría frente al retraso en un RCA:
$11\Delta_G$ mientras que un RCA sería de $16 \times 2\Delta_G = 32\Delta_G$.

- En general, para un sumador de n bits divididos en grupos de 4:
  - Δ_G unidades de tiempo para generar los p_i y g_i.
  - $2\Delta_G$ para generar acarreos intermedios, si tenemos los p_i, g_i y el acarreo inicial
    - hay $\left\lceil \frac{n}{4} \right\rceil$ bloques CLA, cada uno introduciendo un retraso $2\Delta_G$.
  - $2\Delta_G$ para cada salida

  ⇨ Retraso del sumador: $\Delta_G + \left\lceil \frac{n}{4} \right\rceil 2\Delta_G + 2\Delta_G = \left(3 + 2\left\lceil \frac{n}{4} \right\rceil\right)\Delta_G$ unidades de tiempo.

Ejemplo de CLA con 12 bits

- Ejemplo de sumador de 12 bits con acarreo serie entre cada sumador de cuatro bits



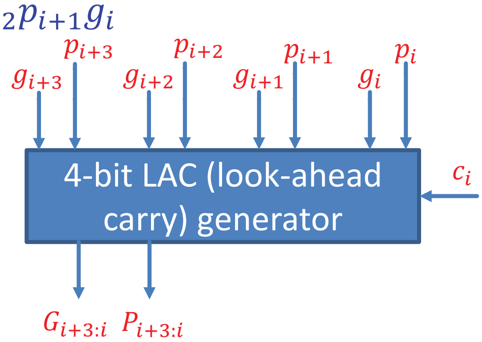
$$\left(\Delta_G + \left\lceil \frac{n}{4} \right\rceil 2\Delta_G + 2\Delta_G\right) = \left(3 + 2\left\lceil \frac{n}{4} \right\rceil\right)\Delta_G = 9\Delta_G$$

$$c_{4k+1} = g_{4k} + p_{4k}c_{4k}$$
$$c_{4k+2} = g_{4k+1} + p_{4k+1}g_{4k} + p_{4k+1}p_{4k}c_{4k}$$
$$c_{4k+3} = g_{4k+2} + p_{4k+2}g_{4k+1} + p_{4k+2}p_{4k+1}g_i + p_{4k+2}p_{4k+1}p_{4k}c_{4k}$$
$$c_{4k+4} = g_{4k+3} + p_{4k+3}g_{i+2} + p_{4k+3}p_{4k+2}g_{4k+1} + p_{4k+3}p_{4k+2}p_{4k+1}g_{4k} + p_{4k+3}p_{4k+2}p_{4k+1}p_{4k}c_{4k}$$

para $k = 0, 1$ y 2

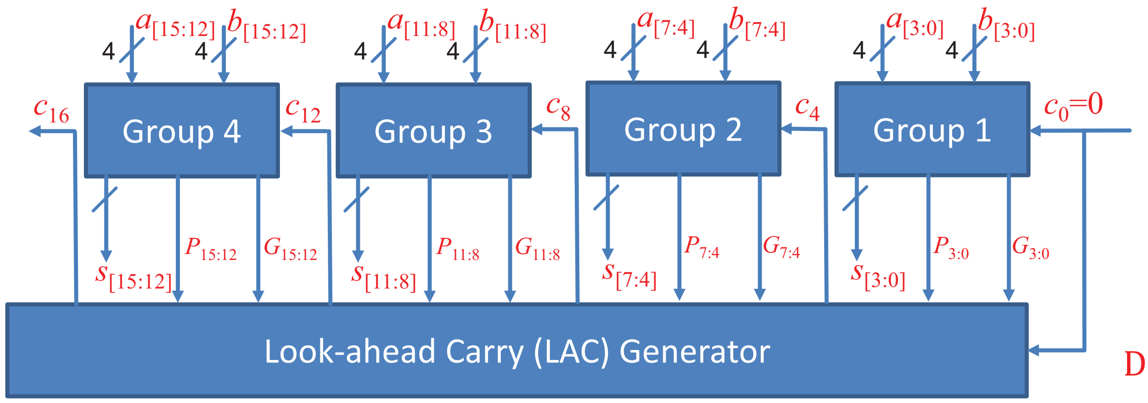
Sumador de acarreo anticipado (CLA, *Carry Look-Ahead*)

- La idea del acarreo anticipado se generaliza desde el concepto de g_i y p_i a nivel de bit al nivel de grupos de bits. Fijémonos en c_4:
- $c_4 = \underbrace{g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0}_{G_{3:0}} + \underbrace{p_3p_2p_1p_0}_{P_{3:0}}\,c_0 = G_{3:0} + P_{3:0}c_0$
  - ✓ El valor de c_4 se obtiene en dos niveles a partir de $G_{3:0}$ y $P_{3:0}$ y de c_0. $G_{3:0}$ y $P_{3:0}$ se realizan en dos niveles en el sumador.
  - ✓ En general,
    $G_{i+3:i} = g_{i+3} + p_{i+3}g_{i+2} + p_{i+3}p_{i+2}g_{i+1} + p_{i+3}p_{i+2}p_{i+1}g_i$
    $P_{i+3:i} = p_{i+3}p_{i+2}p_{i+1}p_i$
  - ✓ Un circuito LAC (*Look Ahead Carry*), que es una generalización del circuito en dos niveles a nivel de bit, proporciona los acarreos de salida de grupo.



- Los sumadores con acarreo anticipado se asocian: con acarreo serie de uno al siguiente usando un circuito LAC.

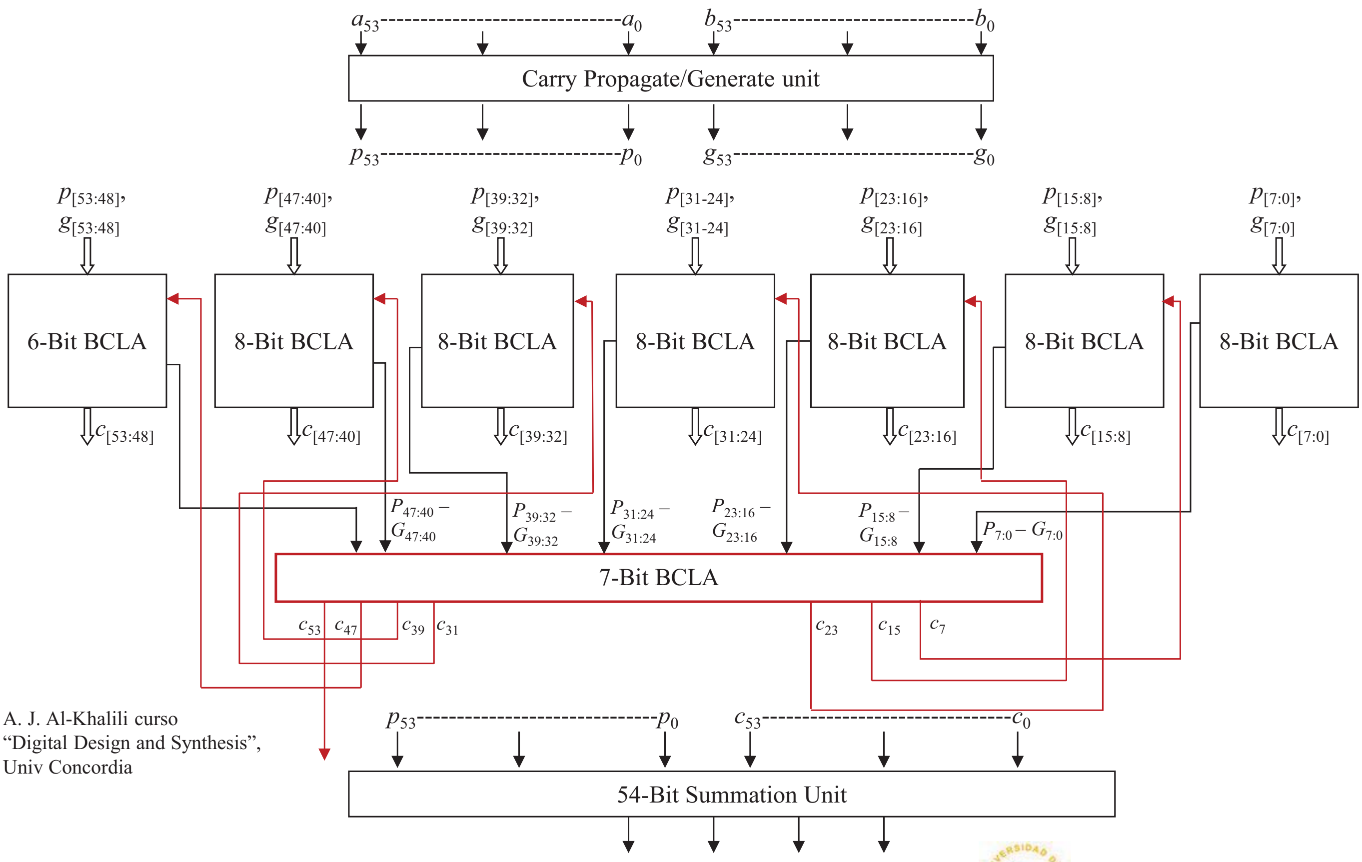
Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)



Delay $= 9\Delta_G$

- g y p individuales: Δ_G.
- G y P de grupo: a partir de los g y p individuales: $2\Delta_G$ ⇨ $\Delta_G + 2\Delta_G = 3\Delta_G$.
- c_4, c_8, c_{12}, c_{16}: A partir de los G y P de grupo: $2\Delta_G$ ⇨ $2\Delta_G + 3\Delta_G = 5\Delta_G$.
- los c_i individuales ($i \neq 4,8,12$) se obtienen a partir de los G y P de grupo que ya llevaban $5\Delta_G$ de retraso porque dependen de c_4, c_8, c_{12}: $2\Delta_G$ ⇨ $2\Delta_G + 5\Delta_G = 7\Delta_G$
- los s_i, a partir de los c_i individuales: $2\Delta_G$ ⇨ $2\Delta_G + 7\Delta_G = 9\Delta_G$.

Ejemplo de diseño de un CLA grande



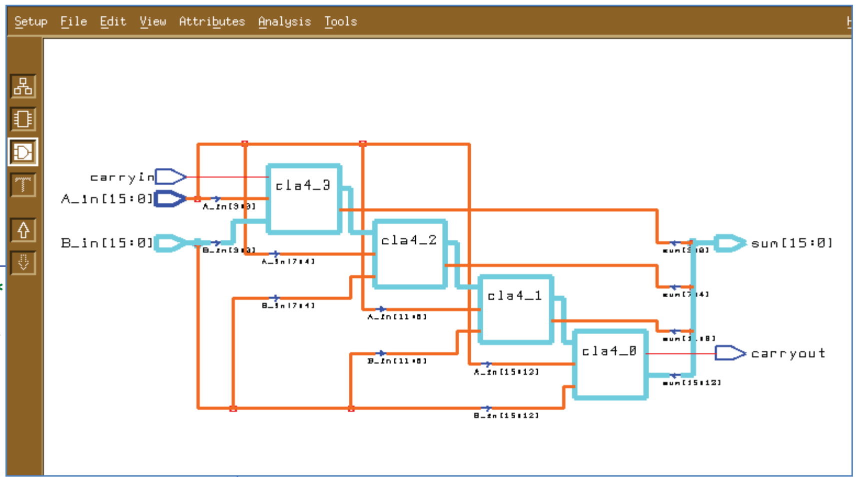
A. J. Al-Khalili curso “Digital Design and Synthesis”, Univ Concordia

Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)

```
//************ 4-bit carry look-ahead adder **********************
module cla4(s,cout,i1,i2,c0);
   output [3:0] s;            //summation
   output cout;               //carryout
   input [3:0] i1;            //input1
   input [3:0] i2;            //input2
   input c0;
   wire [3:0] s;
   wire cout;
   wire [3:0] g;
   wire [3:0] p;
   wire [3:1] c;
   assign  g[3:0]=i1[3:0] & i2[3:0];   //carry generation
   assign  p[3:0]=i1[3:0] ^ i2[3:0];   //carry propagation
   assign  c[1]=g[0] | (p[0] & c0);     //calculate each stage carryout
   assign  c[2]=g[1] | (g[0] & p[1]) | (p[0] & p[1] & c0);
   assign  c[3]=g[2] | (g[1] & p[2]) | (g[0] & p[1] & p[2]) | (p[0] & p[1] & p[2] & c0);
   assign  cout=g[3] | (g[2] & p[3]) | (g[1] & p[2] & p[3])
                     | (g[0] & p[1] & p[2] & p[3]) | (p[0] & p[1] & p[2] & p[3] & c0);
   assign  s[0]=p[0]^c0;                //calculate summation
   assign  s[3:1]=p[3:1]^c[3:1];
endmodule
```

Ming-Ping Kuo: Curso "Computer Arithmetic", Laboratory for Reliable Computing (LARC), National Tsing Hua University , Taiwan, 2003

Sumador de acarreo adelantado (CLA, *Carry Look-Ahead*)



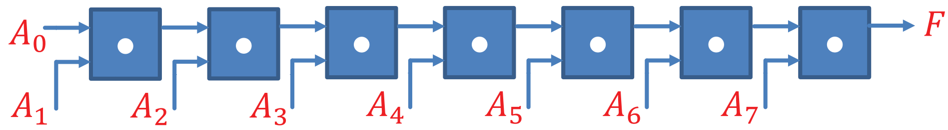
Camino critico de un sumador CLAde 16 bits desde `A_in` hasta `sum[15:0]`

```
//***** 16-bit carry look-ahead adder ********
module cla16(sum,carryout,A_in,B_in,carryin);
   output [15:0] sum;
   output carryout;
   input [15:0] A_in;
   input [15:0] B_in;
   input carryin;
   wire [15:0] sum;
   wire carryout;
   wire [2:0] carry;          // C4,C8,C12
   cla4 c1(sum[3:0],carry[0],A_in[3:0],B_in[3:0],carryin);
   cla4 c2(sum[7:4],carry[1],A_in[7:4],B_in[7:4],carry[0]);
   cla4 c3(sum[11:8],carry[2],A_in[11:8],B_in[11:8],carry[1]);
   cla4 c4(sum[15:12],carryout,A_in[15:12],B_in[15:12],carry[2]);
endmodule
```

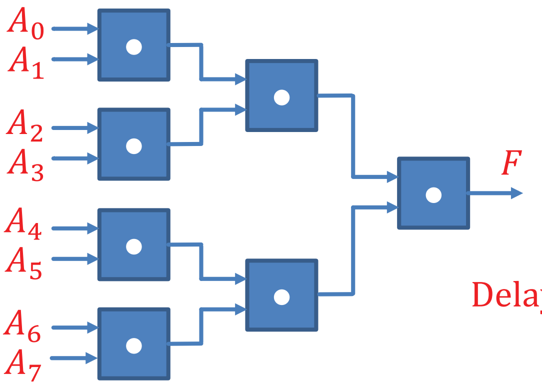
Ming-Ping Kuo: Curso "Computer Arithmetic", Laboratory for Reliable Computing (LARC), National Tsing Hua University , Taiwan, 2003

Sumador de Brent-Kung (Brent-Kung *adder*)

- ¿Es posible usar un árbol binario para mejorar el retraso?
  - Imaginemos la computación lineal de una función F



$\text{Delay} \cong N \cdot \Delta_G$



- La siguiente implementación usa el mismo número de bloques.

  ¿Qué deberíamos pedirle a la función realizada por el bloque para que esta implementación sea viable?

  ➔ Sólo que la función realizada sea asociativa

$\text{Delay} \cong (\log_2 N) \cdot \Delta_G$

Tema 5: Aritmética en punto fijo

Sumador de Brent-Kung (Brent-Kung *adder*)

- Consideremos ahora un bloque que realiza la siguiente operación:

$$(g_{left}, p_{left}) \bullet (g_{right}, p_{right}) = (g, p)$$

donde $g = g_{left} + p_{left} \cdot g_{right}$

y $p = p_{left} \cdot p_{right}$

➔ Esta operación es asociativa

- Definimos ahora G_i y P_i:

Para $i = 0$,

$$(G_0, P_0) = (g_0, p_0)$$

Para $i > 0$, $\begin{cases} g_i = A_i \cdot B_i \\ p_i = A_i \oplus B_i \end{cases}$

$$(G_i, P_i) = (g_i, p_i) \bullet (G_{i-1}, P_{i-1}) = (g_i, p_i) \bullet (g_{i-1}, p_{i-1}) \bullet \cdots \bullet (g_1, p_1) \bullet (g_0, p_0)$$

Sumador de Brent-Kung (Brent-Kung *adder*)

- Con esto ya estamos en condiciones de abordar el sumador de Brent-Kung

$$(G_1,P_1)=(g_1,p_1)\bullet(G_0,P_0)=(g_1,p_1)\bullet(g_0,p_0)=(g_1+p_1g_0,\quad p_1p_0)$$

$$(G_2,P_2)=(g_2,p_2)\bullet(G_1,P_1)=(g_2,p_2)\bullet(g_1+p_1g_0,p_1p_0)=(g_2+p_2(g_1+p_1g_0),\quad p_2p_1p_0)$$

$$=(g_2+p_2g_1+p_2p_1g_0,\quad p_2p_1p_0)$$

$$(G_3,P_3)=(g_3,p_3)\bullet(G_2,P_2)=(g_3,p_3)\bullet(g_2+p_2g_1+p_2p_1g_0,p_2p_1p_0)$$

$$=(g_3+p_3(g_2+p_2g_1+p_2p_1g_0,p_2p_1p_0),\quad p_3p_2p_1p_0)=$$

$$=(g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0,\quad p_3p_2p_1p_0)$$

…

➪ Si comparamos lo que va saliendo con las expresiones que obtuvimos en CLA:

$$(G_3,P_3)=(\underbrace{g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0}_{G_{3:0}},\underbrace{p_3p_2p_1p_0}_{P_{3:0}})$$

y $c_4=G_{3:0}+P_{3:0}\cdot c_0$

En general, $c_i=G_{i-1:0}+P_{i-1:0}\cdot c_0$

Tema 5: Aritmética en punto fijo

Sumador de Brent-Kung (Brent-Kung *adder*)

- Con esto ya estamos en condiciones de abordar el sumador de Brent-Kung

$$(G_1,P_1)=(g_1,p_1)\bullet(G_0,P_0)=(g_1,p_1)\bullet(g_0,p_0)=(g_1+p_1g_0,\quad p_1p_0)$$

$$(G_2,P_2)=(g_2,p_2)\bullet(G_1,P_1)=(g_2,p_2)\bullet(g_1+p_1g_0,p_1p_0)=(g_2+p_2(g_1+p_1g_0),\quad p_2p_1p_0)$$

$$=(g_2+p_2g_1+p_2p_1g_0,\quad p_2p_1p_0)$$

$$(G_3,P_3)=(g_3,p_3)\bullet(G_2,P_2)=(g_3,p_3)\bullet(g_2+p_2g_1+p_2p_1g_0,p_2p_1p_0)$$

$$=(g_3+p_3(g_2+p_2g_1+p_2p_1g_0,p_2p_1p_0),\quad p_3p_2p_1p_0)=$$

$$=(g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0,\quad p_3p_2p_1p_0)$$

…

➪ Si comparamos lo que va saliendo con las expresiones que obtuvimos en CLA:

$$(G_0,P_0)=(\underbrace{g_0}_{G_{0:0}},\underbrace{p_0}_{P_{0:0}}) \qquad c_1=G_{0:0}+P_{0:0}\cdot c_0$$

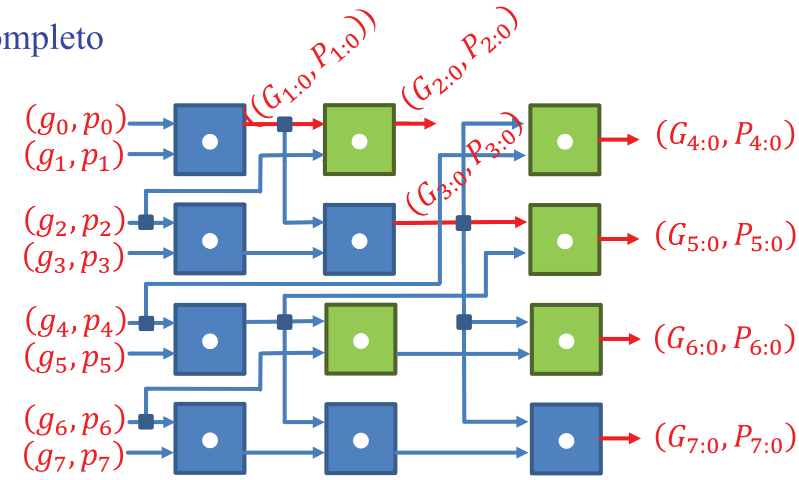
$$(G_1,P_1)=(\underbrace{g_1+p_1g_0}_{G_{1:0}},\underbrace{p_1p_0}_{P_{1:0}}) \qquad c_2=G_{1:0}+P_{1:0}\cdot c_0$$

$$(G_2,P_2)=(\underbrace{g_2+p_2g_1+p_2p_1g_0}_{G_{2:0}},\underbrace{p_2p_1p_0}_{P_{2:0}}) \qquad c_3=G_{2:0}+P_{2:0}\cdot c_0$$

$$(G_3,P_3)=(\underbrace{g_3+p_3g_2+p_3p_2g_1+p_3p_2p_1g_0}_{G_{3:0}},\underbrace{p_3p_2p_1p_0}_{P_{3:0}}) \qquad c_4=G_{3:0}+P_{3:0}\cdot c_0$$

Sumador de Brent-Kung (Brent-Kung *adder*)

- Y el árbol completo



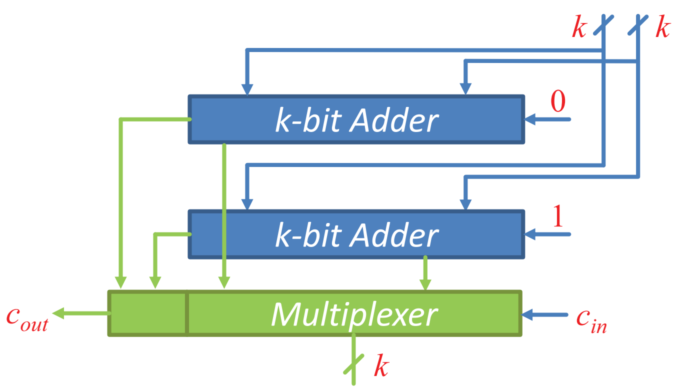
Delay $\cong (\log_2 N) \cdot \Delta_G$

- El área es, de media, casi el doble de un RCA.
- El *layout* es muy compacto
- El retraso crece en forma logarítmica.
- Obtenidas las señales de acarreo, los bits de suma requieren un tiempo adicional constante.
- Suelen usarse en sumadores "anchos"

Sumadores por suma condicional (*Conditional sum adders*)

- Forman otro grupo de sumadores rápidos que proporcionan retraso logarítmico.

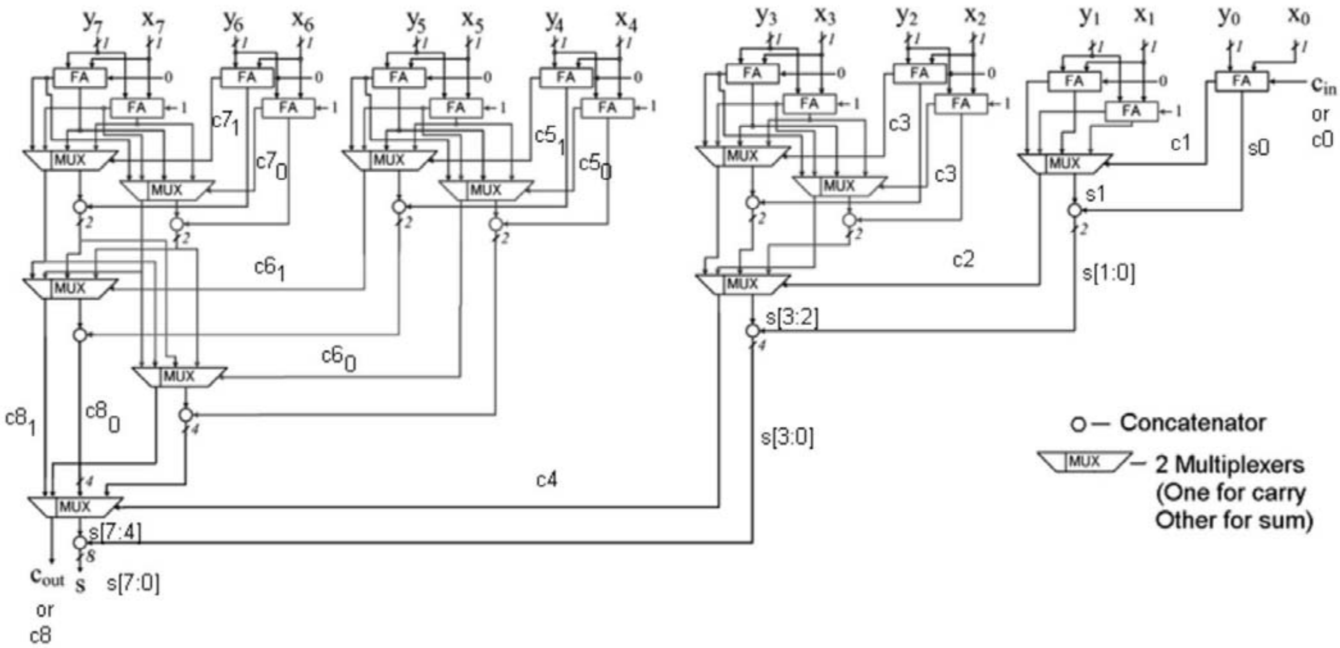
- Principio básico:
  - generación de dos grupos de salidas para un grupo concreto de bits de entrada.
  - en un grupo, su acarreo de entrada es 1 y en el otro, 0.
  - una vez que se conoce el acarreo de entrada real sólo se requiere un MUX para quedarnos con la suma correcta (sin esperar la propagación de ese acarreo).



- estos grupos pueden, a su vez, subdividirse para conseguir reducir el retraso

Sumadores por suma condicional (*Conditional sum adders*)

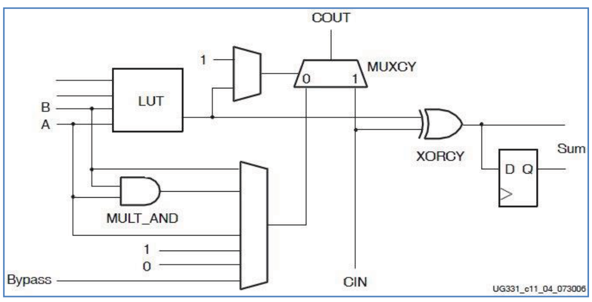
- 8-bit conditional sum adder.



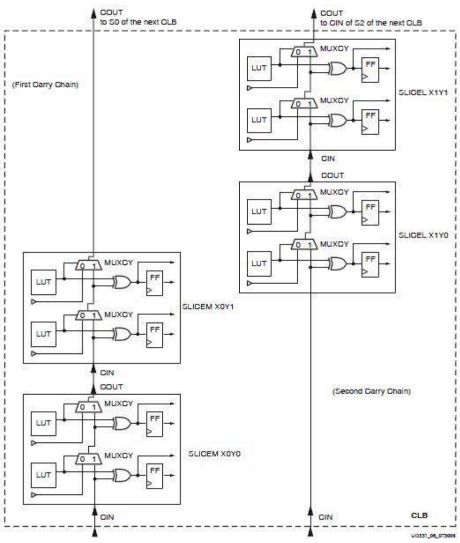
Recursos aritméticos en FPGA

Propagación de acarreo en Xilinx Spartan3

Lógica de propagación de acarreo y suma



Recursos de suma y acarreo en un CLB



Spartan-3 Generation FPGA User Guide (UG331)
https://docs.xilinx.com/v/u/en-US/ug331

Tema 5. ARITMÉTICA EN PUNTO FIJO

- Sistemas de numeración
- Circuitos básicos para aritmética binaria
- Arquitecturas de sumadores
- Ejercicios resueltos

Conversión decimal → binario

- Convierta los siguientes números en binario redondeando a 8 bits

Ejercicio 1: $0.534_{(10}$.

- Parte entera = $0_{(10}$ = $0_{(2}$;
- Parte decimal = 0.534

```
0.534×2=1.068; primer  digito 1
0.068×2=0.136; segundo dígito 0
0.136×2=0.272; tercer  dígito 0
0.272×2=0.544; cuarto  dígito 0
0.544×2=1.088; quinto  dígito 1
0.088×2=0.176; sexto   dígito 0
0.176×2=0.352; séptimo dígito 0
0.352×2=0.704; octavo  dígito 0
```

Por lo tanto, $0.534_{(10} = 0.10001000_{(2}$

Prueba: $2^{-1}+2^{-5} = 0.53125_{(10}$

(observe que aparece un error en la precisión)

Ejercicio 2: $3.84_{(10}$.

- Parte entera = $3_{(10}$ = $11_{(2}$;
- Parte decimal = 0.84

```
0.84×2=1.68; 1er digito 1
0.68×2=1.36; 2º  dígito 1
0.36×2=0.72; 3er dígito 0
0.72×2=1.44; 4º  dígito 1
0.44×2=0.88; 5º  dígito 0
0.88×2=1.76; 6º  dígito 1
```

Por lo tanto, $0.534_{(10} = 0.10001000_{(2}$

Prueba: $2^{-1}+2^{-5} = 0.53125_{(10}$

(observe que aparece un error en la precisión)

Sistemas de numeración no convencionales

➔ Ejercicio 3: Represente 5 y –3 en negabinario (base –2). Realice la suma de ambos números en negabinario justificando todos los pasos realizados para llegar al resultado.

- 1101 → $(-2)^3+(-2)^2+(-2)^0 = -8+4+1 = -3$;
  0101 → $(-2)^2+(-2)^0 = 4+1 = 5$

```
 1 1 1 1   0
… 1 1 1 1 1 1
        1 1 0 1
      + 0 1 0 1
 … 0 0 0 1 1 0
```

1+1 →	$2_{(-2}$ →	110
1 →	$1_{(-2}$ →	001
1+1+1 →	$3_{(-2}$ →	111

0110 → $(-2)^2+(-2)^1 = 4-2 = 2$

Representación de números negativos: ejemplos de operación

➔ Ejercicio 4: Dados los números binarios $X = 1010100_{(2}$ e $Y = 1000011_{(2}$, obtenga $X - Y$ e $Y - X$ usando complemento a 2.

a)

```
 X            =  1010100                         (-44)
Comp.2 de Y =  0111101
Suma:         = 10010001    (end carry = 1)
Descarte 2^7 =  0010001
X - Y         =  0010001                     (-44)-(-61)=17
```

b)

```
 Y            =  1000011                         (-61)
Comp.2 de X =  0101100                         (44)
Suma:         = 01101111    (end carry = 0)
Y - X         =  -(comp. a 2 de 1101111) = -0010001   (-17)
```

Representación de números negativos: ejemplos de operación

➔ Ejercicio 5: Dados los números binarios $X = 1010100_{(2}$ e $Y = 1000011_{(2}$, obtenga $X - Y$ e $Y - X$ usando complemento a 1.

a)

```
 X            =  1010100                                   (-43)
Comp.1 de Y  =  0111100
Suma:        = 10010000    (end carry = 1)
Descarte 2^7 =  0010000
Sumamos 1       0000001
X - Y        =  0010001                          (-43)-(-60)=17
```

b)

```
 Y            =  1000011                                   (-60)
Comp.1 de X  =  0101011
Suma:        =  1101110    (end carry = 0)
Y - X        =  -(comp. a 1 de 1101110) = -0010001         (-17)
```


Representación de números negativos: ejemplos de operación

❑ Sume en complemento a 2 usando 8 bits

➔ Ejercicio 6: 6_{10} y 13_{10}

6_{10} = $00000110_{(2}$
13_{10} = $00001101_{(2}$
Suma → $00010011_{(2}$ ⇨ 19_{10}

➔ Ejercicio 8: 6_{10} y $(-13)_{10}$

6_{10} = $00000110_{(2}$
-13_{10} = 11110011
Suma → 11111001 ⇨ $(-7)_{10}$

➔ Ejercicio 7: $(-6)_{10}$ y 13_{10}

-6_{10} = 11111010
13_{10} = 00001101
Suma → 100000111
Suma = 00000111 ⇨ 7_{10}

➔ Ejercicio 9: $(-6)_{10}$ y $(-13)_{10}$

-6_{10} = 11111010
-13_{10} = 11110011
Suma → 111101101
Suma = 11101101 ⇨ $(-19)_{10}$

⇨ Para obtener una respuesta correcta, debemos asegurar que el resultado tenga el número suficiente de bits para poder expresar el resultado.

Suma/resta: sin signo

➔ Ejercicio 10: Realice las operaciones siguientes usando números sin signo

⇨ 154_{10}	=	10011010			⇨ 154_{10}	=	10011010	
87_{10}	=	+01010111			87_{10}	=	−01010111	
		00011110					01000111	
Suma	→	11110001	⇨ 241_{10}		Resta	→	01000011	⇨ 67_{10}
⇨ 154_{10}	=	10011010			⇨ 154_{10}	=	10011010	
175_{10}	=	+10101111			175_{10}	=	−10101111	
		10111110					11101111	
Suma	→	01001001	⇨ 329_{10}		Resta	→	11101011	⇨ $(-21)_{10}$

ERROR de desbordamiento

$$\boxed{OV = c_n}$$

Suma/resta: con signo

➔ Ejercicio 11: Dados X = 102 e Y = 87, realice las operaciones siguientes usando números con signo: X+Y, X-Y, -X+Y, -X-Y.

⇨ 102_{10}	=	01100110			⇨ 102_{10}	=	01100110	
87_{10}	=	+01010111			-87_{10}	=	+10101001	
		01000110					11100000	
Suma	→	10111101	⇨ $(-15)_{10}$ Incorrecto		Suma	→	00001111	⇨ $(+15)_{10}$
⇨ -102_{10}	=	10011010			⇨ -102_{10}	=	10011010	
87_{10}	=	+01010111			-87_{10}	=	+10101001	
		00011110					10111000	
Suma	→	11110001	⇨ $(-15)_{10}$		Resta	→	01000011	⇨ $(+67)_{10}$ Incorrecto

ERROR de desbordamiento

$$\boxed{OV = c_n \oplus c_{n-1}}$$

No hay **ERROR** de desbordamiento