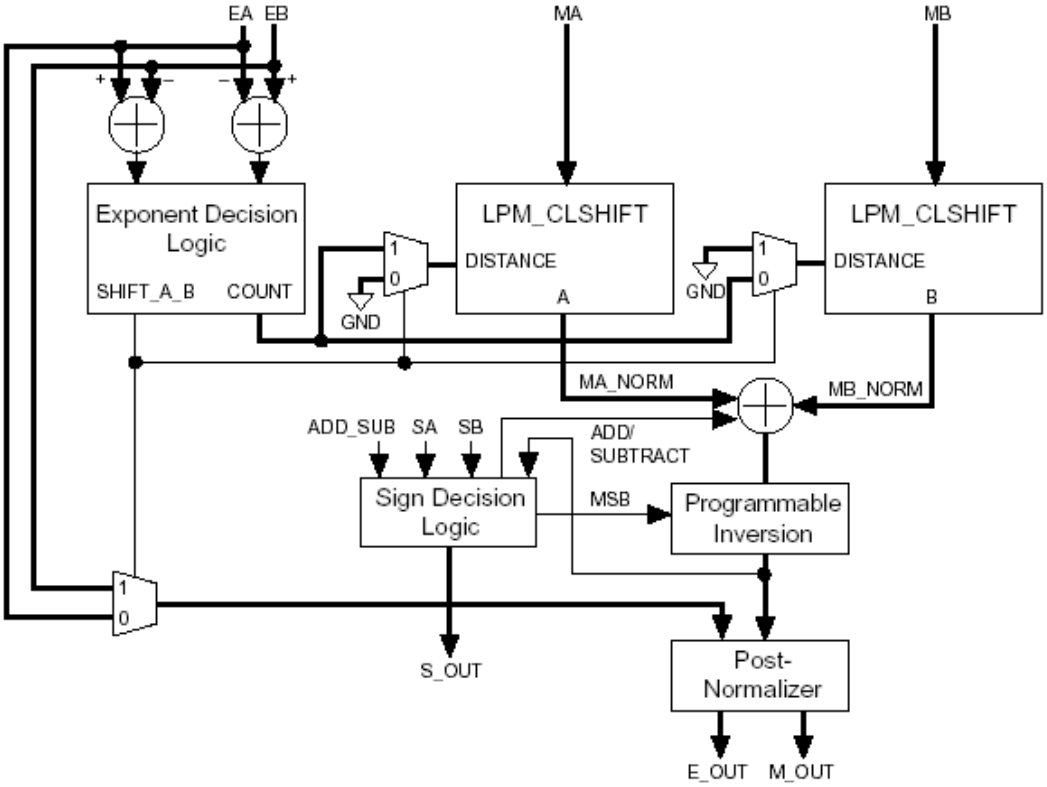
Departamento de Ingeniería Electrónica

# SISTEMAS DIGITALES DE INSTRUMENTACIÓN Y CONTROL

## *TEMA 3*

## *Diseño de Subsistemas Aritméticos*




Rafael Ramos Lara

Febrero 2007

# *TEMA 3*

## *Diseño de Subsistemas Aritméticos*



# Indice (I)

# Indice (II)





# Indice (III)

# Indice (IV)





## *3.1 Formatos de representación numérica*

# Formatos normales de datos numéricos

- Existen diversos formatos de representación numérica de uso común en los sistemas digitales
- El formato concreto a utilizar dependerá de las características (magnitud, signo y precisión) de los número que intervienen en el proceso

Los tipos de representaciones más habituales son:

- Números enteros sin signo
- Números negativos: - signo más magnitud absoluta
  - complemento a 1
  - complemento a 2
- BCD (números Decimales Codificados en Binario)
- Coma flotante



# Números enteros sin signo

Es la forma más sencilla de representar números

El rango de números que se puede representar depende de la longitud del dato:

- ***8 bits***: rango de representación: 0 <-> 255
- ***16 bits***: rango de representación: 0 <-> 65535

Ventajas:

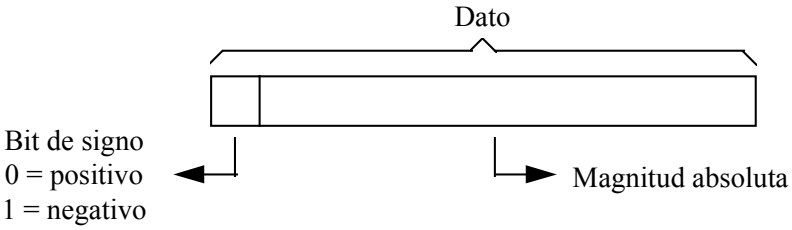
- Los circuitos lógicos que realizan operaciones con este formato son sencillos de diseñar y realizar

Inconvenientes:

- El intervalo de números está limitado a la longitud del dato
- No se puede representar números negativos o fraccionarios

# Números negativos: signo más magnitud absoluta

El bit de mayor peso es el bit de signo, y el resto de bits representa su magnitud absoluta




Ejemplo:
25: 0 0 0 1 1 0 0 1
-25: 1 0 0 1 1 0 0 1

Si, por ejemplo, el dato es de 8 bits, se pueden representar los siguientes números:

- positivos: del 0000 0000 al 0111 1111 (0 al 127)
- negativos: del 1000 0000 al 1111 1111 (0 al -127)



# Números negativos: complemento a 1

El valor negativo de un número en complemento a 1 se obtiene invirtiendo todos los dígitos ($d_{n-1}....d_0$) del número expresado en positivo:

Bit de signo

$$N = -\left(2^{n-1} - 1\right) d_{n-1} + \sum_{i=0}^{n-2} d_i \cdot 2^i$$

Resto de bits

Si, por ejemplo, el dato es de 8 bits, el rango de representación es:

- positivos: del 0000 0000 al 0111 1111 (0 al 127)
- negativos: del 1111 1111 al 1000 0000 (0 al -127)

# Números negativos: complemento a 2

Con este formato y con un dato de N bits, un número negativo -M se representa como: $2^N$ - M

Bit de signo

$$N = -2^{n-1} \cdot d_{n-1} + \sum_{i=0}^{n-2} d_i \cdot 2^i$$

Resto de bits

Si, por ejemplo, el dato es de 8 bits, se pueden representar los siguientes números:

- positivos: del 0000 0000 al 0111 1111 (0 al 127)
- negativos: del 1111 1111 al 1000 0000 (-1 al -128)



# Código BCD: Números Decimales Codificados en Binario

- Es un código binario utilizado para representar números decimales
- Cada dígito decimal se expresa con cuatro bits
- Permite representar de forma directa números decimales

| ***Dígito Decimal*** | ***BCD*** |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

- El rango de valores representable es pequeño
- Con 8 bits se puede representar: 0....99d

| | | |
|---|---|---|
| Decimal: | 7 | 5 |
| BCD: | 0111 | 0101 |
| Binario: | 101011 | |

# Coma Flotante

- El formato de números en coma flotante se define como: $\pm A \cdot 2^{\pm B}$
- A: ***mantisa,*** B: ***exponente***
- Permite representar números fraccionarios, enteros, positivos y negativos
- La precisión depende del número de bits de la mantisa

- Inconveniente: la complejidad de los circuitos que realizan operaciones con este formato



# *3.2 Arquitecturas serie, paralelo y pipelined*

# Arquitecturas para el procesado aritmético

El tipo de arquitectura adecuada para implementar un subsistema de procesado aritmético depende de:

- ***Velocidad de muestreo***: varia entre las velocidades muy bajas propias de aplicaciones de voz o comunicaciones, y velocidades muy altas propias de sistemas de radar, sonar, video y procesado de imagen
- ***Recursos hardware***: el número y tipo de componentes digitales que incorpora un PLD, así como la arquitectura interna (distribución de componentes y recursos de interconexión)

Tipos de arquitectura:

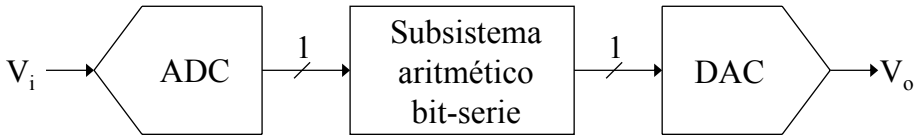
- ***Arquitectura serie***: procesan un bit del dato por cada ciclo de reloj
- ***Arquitectura paralelo***: procesan un dato por cada ciclo de reloj
- ***Arquitectura pipelined***: permite realizar en paralelo varias operaciones a la vez



# Arquitectura serie

- Procesa un bit del dato por cada ciclo de reloj
- Requiere menos recursos hardware que otras arquitecturas
- Necesita menos interconexiones y menos pines de entrada/salida
- Son idóneas en aplicaciones de baja frecuencia donde la velocidad de proceso no es un parámetro restrictivo
- Normalmente se utilizan junto con ADC y DAC serie

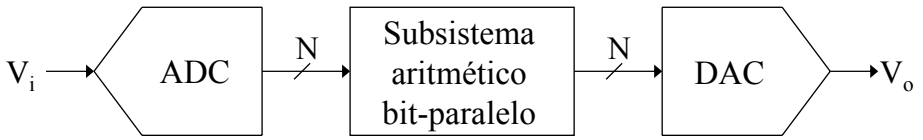
Esquema arquitectura serie



# Arquitectura paralelo

- Procesa un dato por cada ciclo de reloj
- Es la arquitectura que requiere mas recursos hardware
- Necesita mas interconexiones y pines de entrada/salida
- Son idóneas en aplicaciones de muy alta frecuencia
- Normalmente se utilizan junto con ADC y DAC paralelo con elevada velocidad de conversión (ADC Flahs)

Esquema arquitectura paralelo

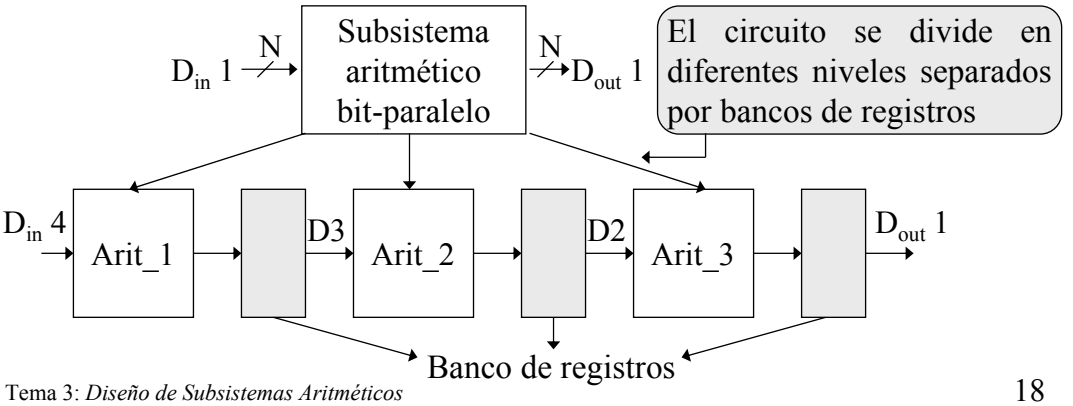





# Arquitectura Pipelined

- Se introducen bancos de registros entre los niveles de lógica
- Permite procesar varias muestras a la vez
- Aumenta la velocidad de cálculo en estructuras bit-paralelo

Esquema arquitectura pipelined



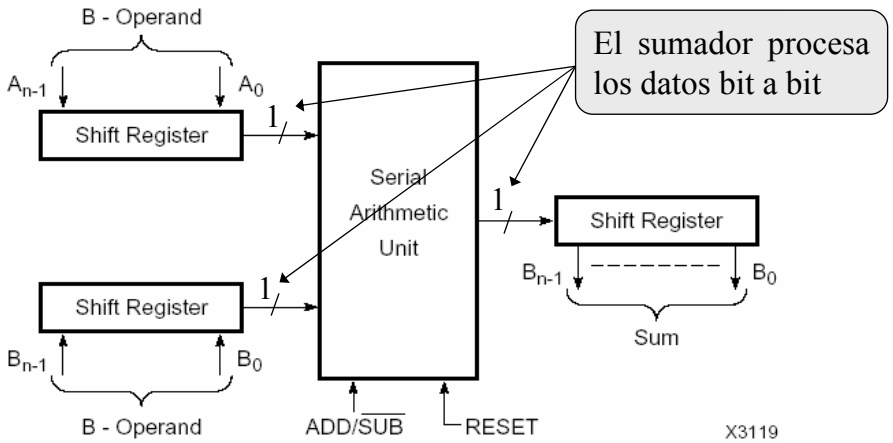
## *3.3 Sumadores/restadores*



## *3.3.1 Sumadores/restadores Serie-Serie*

# Esquema sumador/restador Serie-Serie

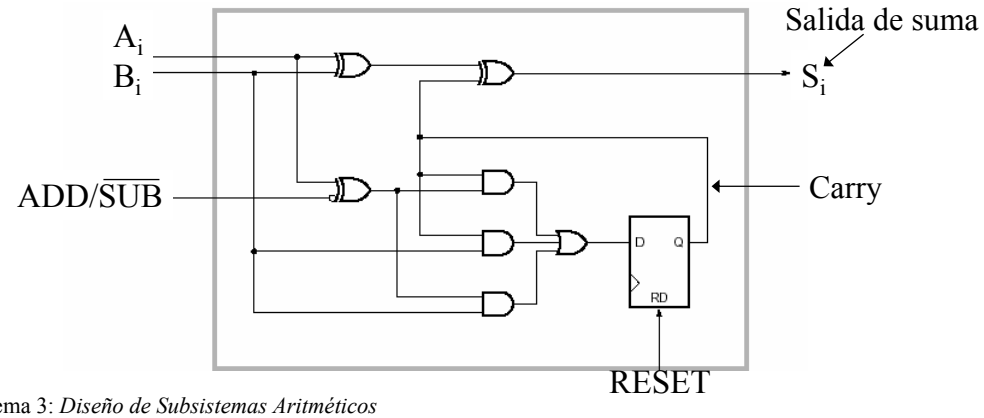
- Los dos operandos se introducen en registros de desplazamiento
- La unidad aritmética procesa los operandos bit a bit
- Se procesa primero el bit de menor peso





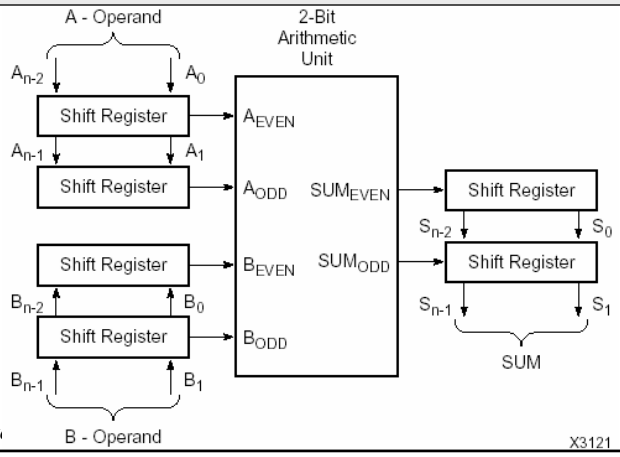

# Esquema de la célula básica sumador/restador serie

- Sumador/restador de 1 bit con salida de suma y bit de carry/borrow registrado
- Para completar la suma de dos datos de N bits se requieren N ciclos de reloj
- La unidad aritmética se puede implementar en un solo CLB



# Arquitectura 2-bit serie

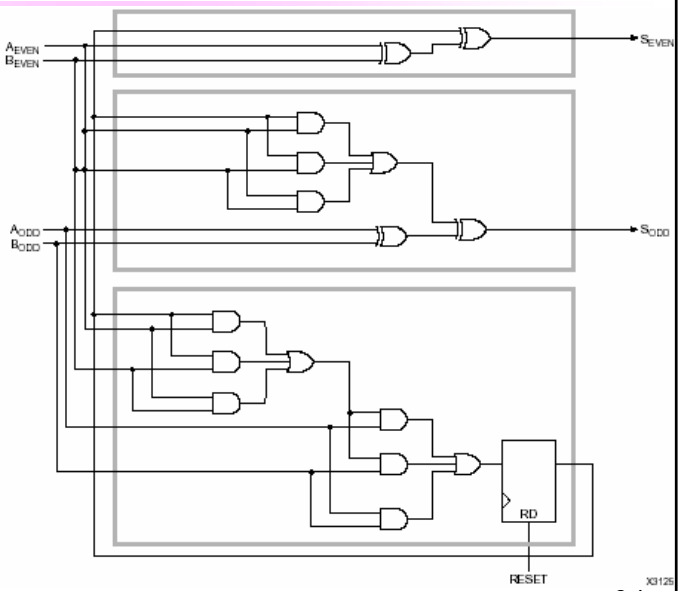
- Se puede aumentar la velocidad procesando dos bits simultáneamente
- Los bits pares e impares se cargan en registros de desplazamiento separados
- La unidad aritmética suma dos bits en cada ciclo de reloj
- El resultado se carga en dos registros de desplazamiento (bits par e impar)






# Esquema del sumador 2-bit serie

Para llevar a cabo la suma con esta estructura se requiere la mitad de ciclos de reloj que con el sumador de 1 bit serie

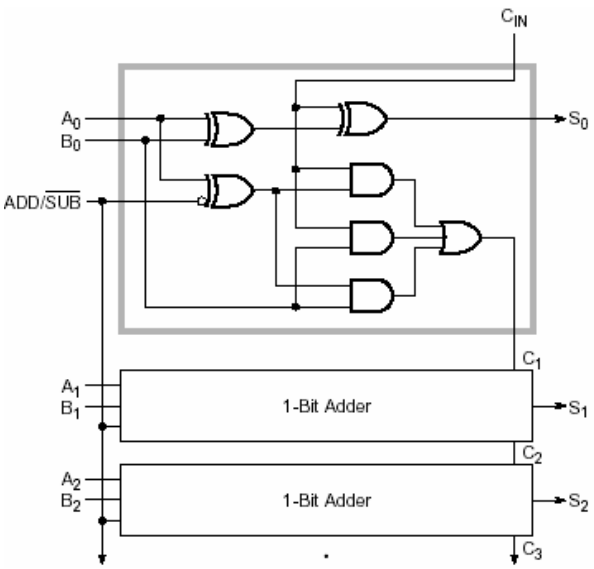


# 3.3.2 Sumadores/restadores Paralelo



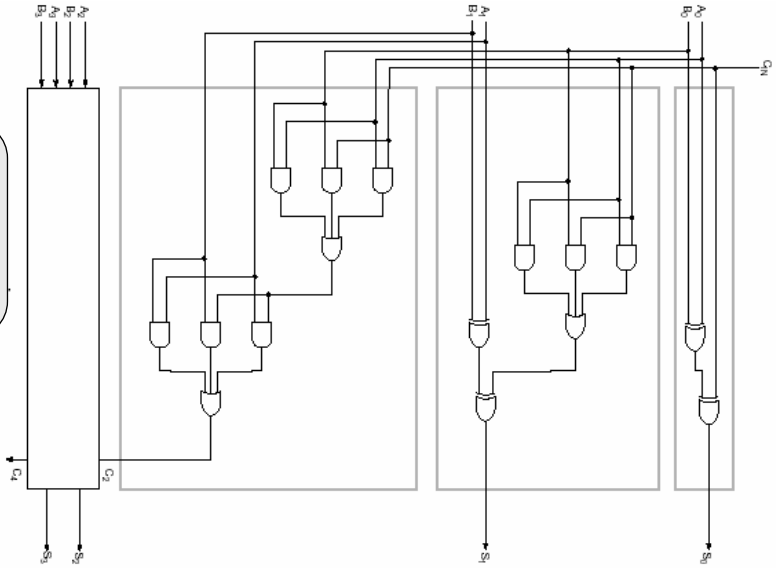
## Sumador paralelo con acarreo propagado

- Compuesto básicamente de N células sumadoras de 1 bit donde se ha suprimido la báscula que almacena el acarreo
- Utiliza un CLB por bit (menos que otros sumadores paralelo)
- El retardo es N veces el equivalente a un CLB
- Utilizando células sumadoras de dos bits se puede reducir el retardo



# Sumador paralelo con células sumadoras de 2 bits

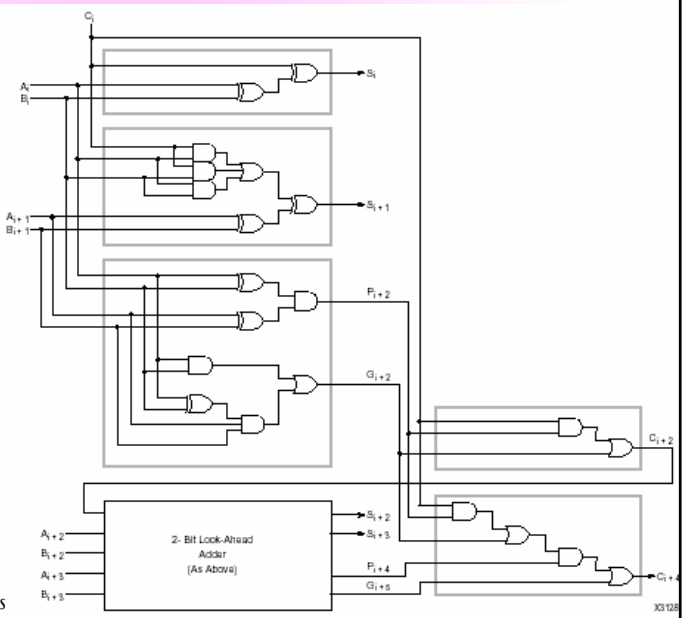
Utilizando células sumadoras de dos bits se reduce el retardo generado por la propagación del carry






# Sumador paralelo de acarreo anticipado

- Los sumadores/restadores de acarreo generado tienen elevado retardo para N↑
- La técnica de acarreo anticipado genera todos los bits de acarreo en paralelo
- Con acarreo anticipado se reduce el retardo
- Para implementar esta solución se requiere un número mayor de CLB's



# *3.4 Multiplicadores*



## Introducción

- El multiplicador es un componente fundamental en algoritmos *DSP*
- El producto de un número *A* de "*n*" bits por otro número *B* de "*m*" bits genera un producto *P* de "*m+n*" bits
- Tipos de multiplicadores:
  - Array (paralelo): los operandos de entrada y el resultado se procesan en paralelo
  - Serie-paralelo: un operando se carga en paralelo y el otro se introduce en serie, el resultado se obtiene en serie
  - Serie-Serie: los dos operandos entran en serie, el resultado aparece en serie
  - Multiplicadores pipelined: procesan varios productos a la vez
  - Multiplicadores de hardware reducido

## *3.4.1 Multiplicadores Array (Paralelo)*



## *3.4.1.1 Multiplicadores Array Unsigned*

# Multiplicación binaria *unsigned*

- El producto se obtiene multiplicando (*función AND*) bit a bit los operandos X e Y y sumando los resultados parciales (*función suma*)
- Para operandos de *"n"* y *"m"* bits, el resultado es de *"n+m"* bits

El producto de dos números enteros positivos viene dado por:

$$X = x_{n-1}\ x_{n-2}\ x_{n-3}\ .....\ x_0 = \sum_{i=0}^{n-1} x_i 2^i$$

$$Y = y_{n-1}\ y_{n-2}\ y_{n-3}\ .....\ y_0 = \sum_{j=0}^{n-1} y_j 2^j$$

$$XY = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 2^{i+j} x_i y_j \longleftarrow \text{Productos y sumas}$$



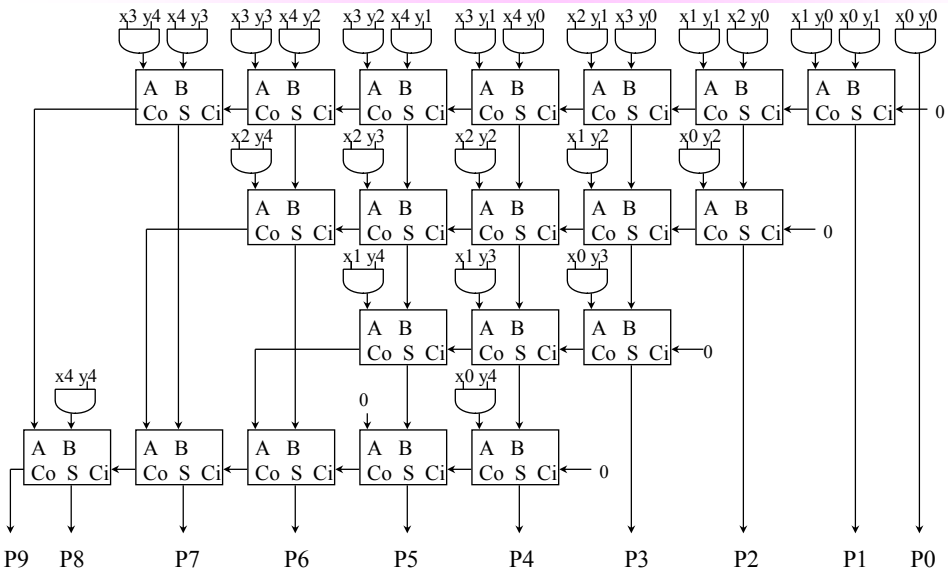
# Implementación multiplicador paralelo *unsigned*

- Se implementan directamente todos las puertas AND y sumadores que intervienen en la realización del producto
- Emplea más recursos hardware que otras soluciones pero el producto se obtiene sin señal de reloj

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Multiplicand | > | | | | | X3 | X2 | X1 | X0 |
| Multiplier | > | | | | x | Y3 | Y2 | Y1 | Y0 |
| 1st partial product | > | | AND | | | Y0X3 | Y0X2 | Y0X1 | Y0X0 |
| 2nd partial product | > | | | | Y1X3 | Y1X2 | Y1X1 | Y1X0 | |
| 3rd partial product | > | | | Y2X3 | Y2X2 | Y2X1 | Y2X0 | | |
| 4th partial product | > | + | Y3X3 | Y3X2 | Y3X1 | Y3X0 | | | |
| Final product | > | P7 | P6 | P5 | P4 | P3 | P2 | P1 | P0 |

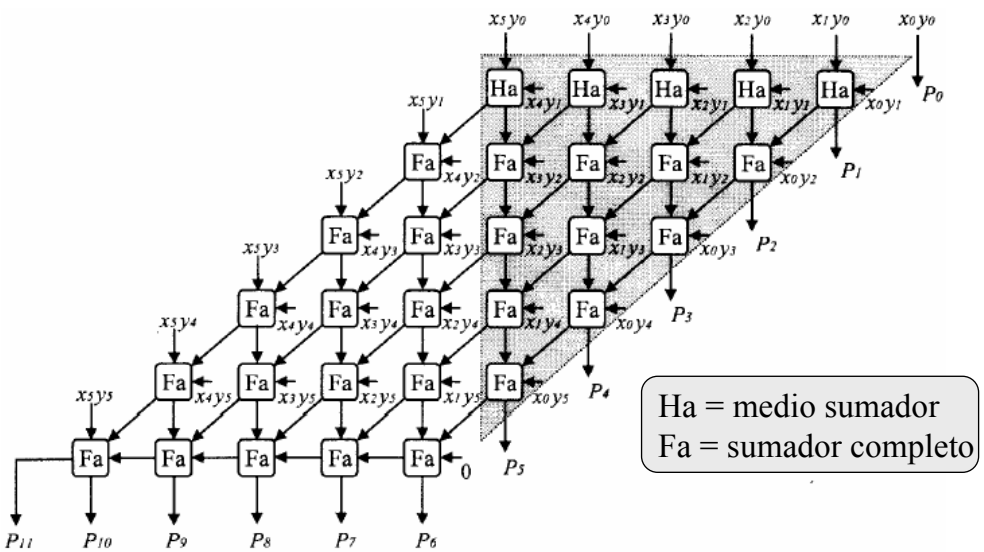
SUMA

# Multiplicador Paralelo 5x5



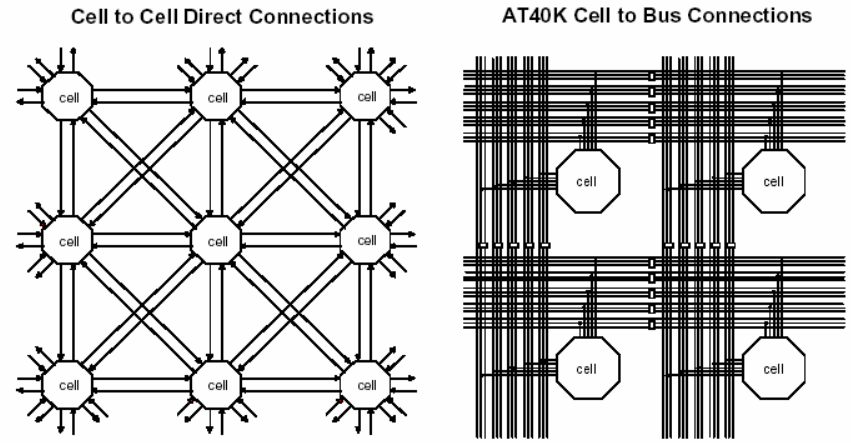



# Multiplicador Paralelo 6x6



# Implementación con FPGA *AT40K* de Atmel

Las conexiones diagonales entre células de la FPGA *AT40K* permite la fácil implementación de multiplicadores haciendo uso de menos puertas lógicas






# *3.4.1.2 Multiplicadores Array c.a.2*

# Multiplicación binaria en *c.a.2*

El producto de dos números expresados en complemento a dos viene dado por:

$$X = x_{n-1}\ x_{n-2}\ x_{n-3}\ .....\ x_0 = -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} x_i 2^i$$

$$Y = y_{n-1}\ y_{n-2}\ y_{n-3}\ .....\ y_0 = -2^{n-1} y_{n-1} + \sum_{j=0}^{n-2} y_j 2^j$$

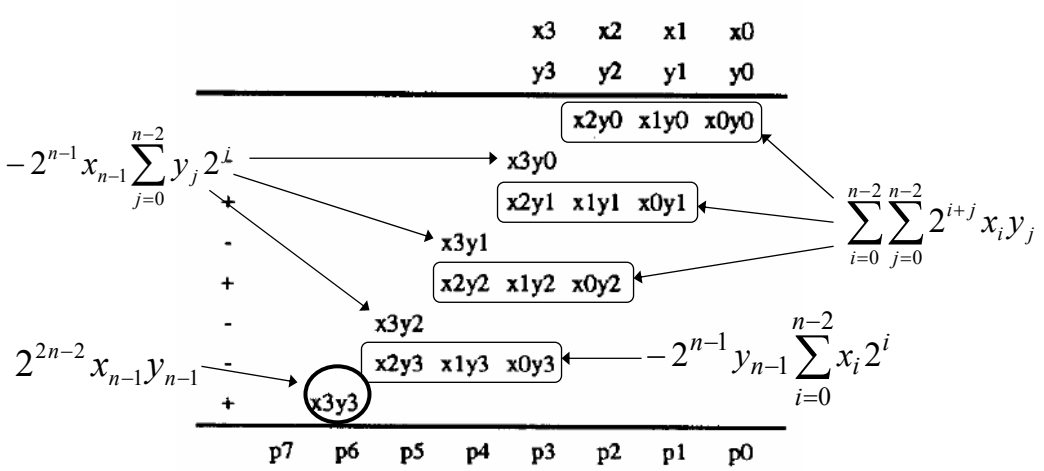
$$XY = \left( -2^{n-1} x_{n-1} + \sum_{i=0}^{n-2} x_i 2^i \right) \cdot \left( -2^{n-1} y_{n-1} + \sum_{j=0}^{n-2} y_j 2^j \right) =$$

$$\sum_{i=0}^{n-2} \sum_{j=0}^{n-2} 2^{i+j} x_i y_j + 2^{2n-2} x_{n-1} y_{n-1} - 2^{n-1} x_{n-1} \sum_{j=0}^{n-2} y_j 2^j - 2^{n-1} y_{n-1} \sum_{i=0}^{n-2} x_i 2^i$$



# Implementación multiplicador paralelo en *c.a.2*

En este caso se necesitan sumadores y restadores para realizar el producto



# Algoritmo Baugh-Wooley para multiplicar en *c.a.2* (I)

Permite realizar el producto utilizando sólo sumadores y puertas AND, NAND

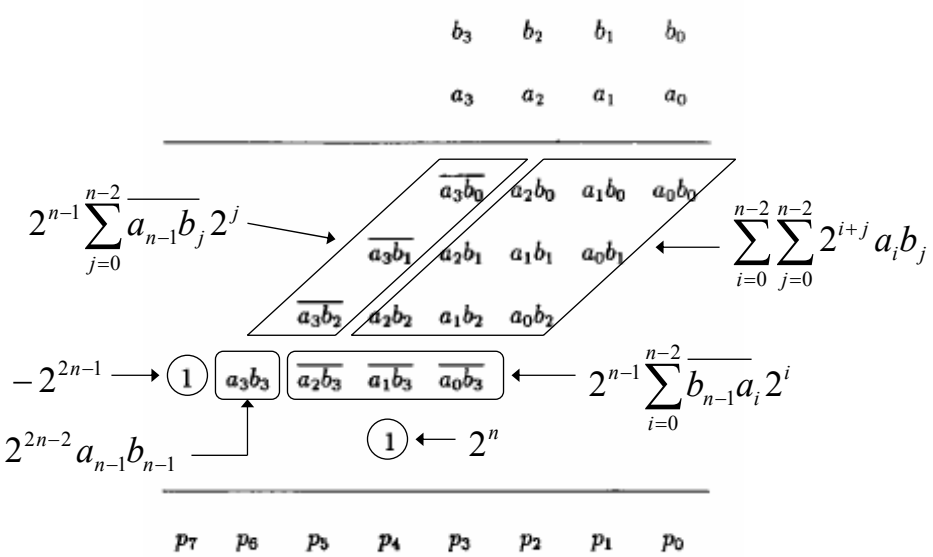
$$AB = \left(-2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right)\cdot\left(-2^{n-1}b_{n-1} + \sum_{j=0}^{n-2} b_j 2^j\right) =$$

$$\sum_{i=0}^{n-2}\sum_{j=0}^{n-2} 2^{i+j} a_i b_j + 2^{2n-2} a_{n-1} b_{n-1} - 2^{n-1} a_{n-1} \sum_{j=0}^{n-2} b_j 2^j - 2^{n-1} b_{n-1} \sum_{i=0}^{n-2} a_i 2^i =$$

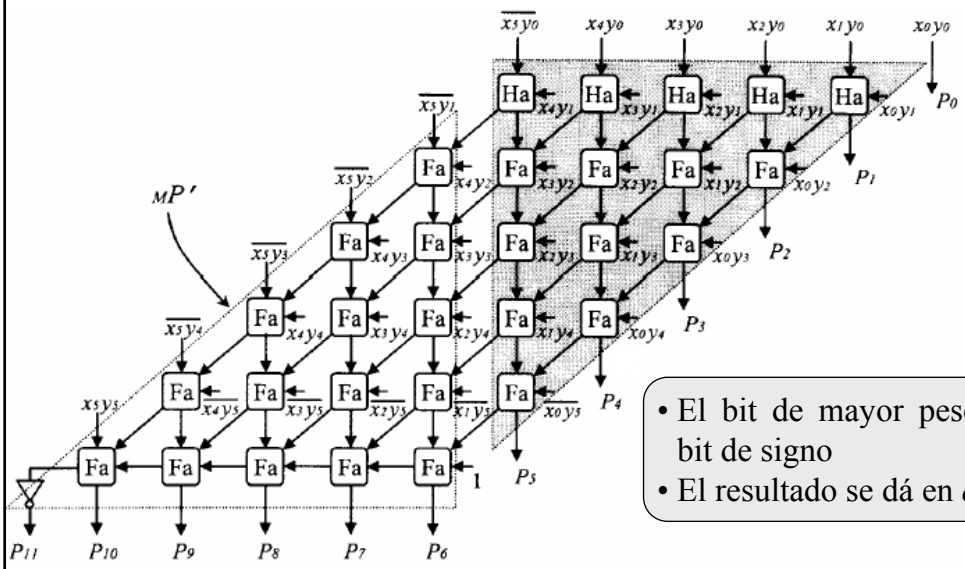
$$\sum_{i=0}^{n-2}\sum_{j=0}^{n-2} 2^{i+j} a_i b_j + 2^{2n-2} a_{n-1} b_{n-1} + 2^{n-1} \sum_{j=0}^{n-2} a_{n-1} b_j 2^j + 2^{n-1} \sum_{i=0}^{n-2} b_{n-1} a_i 2^i + 2^n - 2^{2n-1}$$



# Algoritmo Baugh-Wooley para multiplicar en *c.a.2* (II)



# Multiplicador 6x6 Paralelo en c. a. 2 con el algoritmo Baugh-Wooley



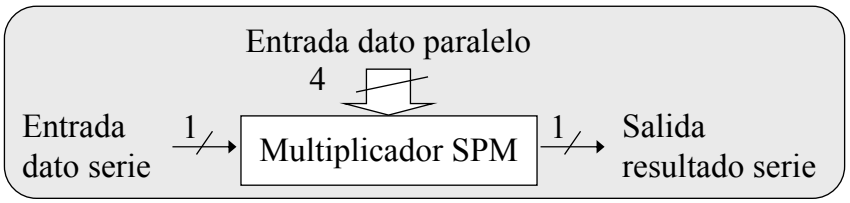

- El bit de mayor peso es el bit de signo
- El resultado se dá en *c. a. 2*



# *3.4.2 Multiplicadores Serie-Paralelo*

# Multiplicadores Serie-Paralelo (SPM)

- Dispone de una entrada de operando serie, otra entrada de operando paralelo y salida del resultado serie
- El hardware empleado en su diseño es más reducido que el multiplicador paralelo
- El producto se realiza secuencialmente bajo el control de una señal de reloj





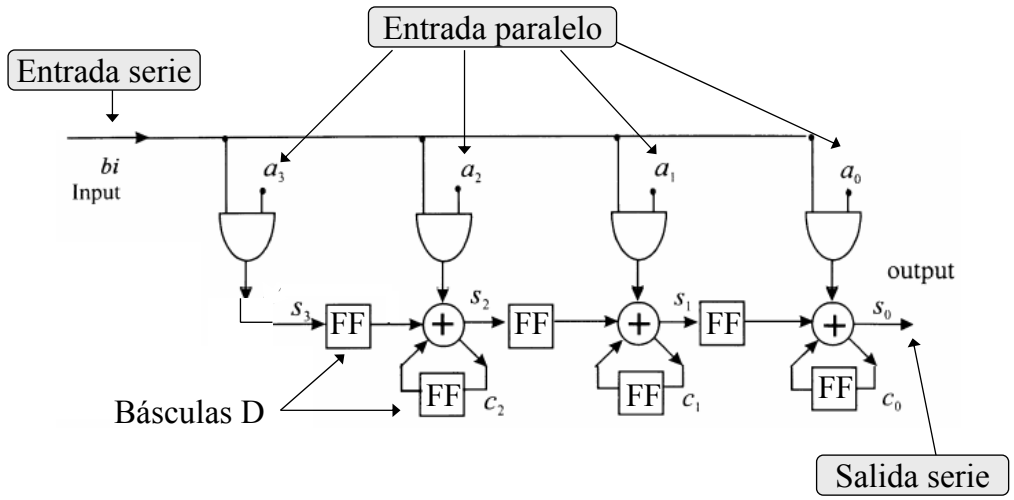

# *3.4.2.1 Multiplicadores Serie Paralelo Unsigned con algoritmo CSAS*

# SPM con algoritmo CSAS

- ***CSAS***: *Carry-Save Add-Shift*
- Su funcionamiento se basa en el algoritmo de suma-desplazamiento:
  - El dato paralelo se multiplica por cada uno de los bits del dato serie mediante puertas AND
  - El resultado de cada producto se desplaza una posición a la derecha
  - Cada nuevo producto parcial se suma con la acumulación desplazada de los anteriores productos
- Si $N$ es el número de bits del dato serie y $M$ del dato paralelo, el resultado de $N+M$ bits se obtiene después de $N+M$ ciclos de reloj
- Durante los primeros $N$ ciclos se introduce el dato serie empezando por el LSB
- Durante los $M$ ciclos siguientes se aplica un cero en la entrada serie y se propagan los bits de carry almacenados en los registros hacia la salida



# Multiplicador CSAS de 4 bits



# Ejemplo multiplicación Serie-Paralelo CSAS (I)

***Ejemplo:*** multiplicador SPM de 4 bits, producto de 9d x 5d

```
      1001  (9d) ←— Dato paralelo
    x 0101  (5d) ←— Dato serie
    ------
      1001
   +  0000
     1001
    0000
    --------
   00101101  (45d) ←
```

$P_7P_6P_5P_4P_3P_2P_1P_0$

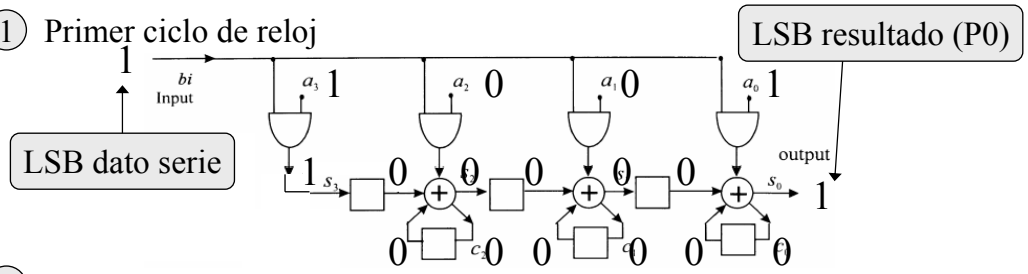
El resultado de 8 bits se obtiene después de 8 ciclos de reloj:

- 4 ciclos para hacer los 4 productos parciales
- 4 ciclos para propagar los bits de carry hacia la salida serie

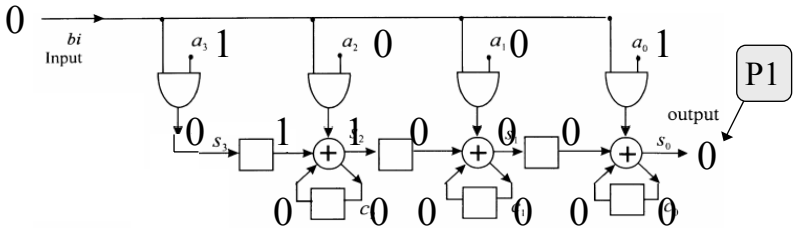


# Ejemplo multiplicación Serie-Paralelo CSAS (II)

1 Primer ciclo de reloj

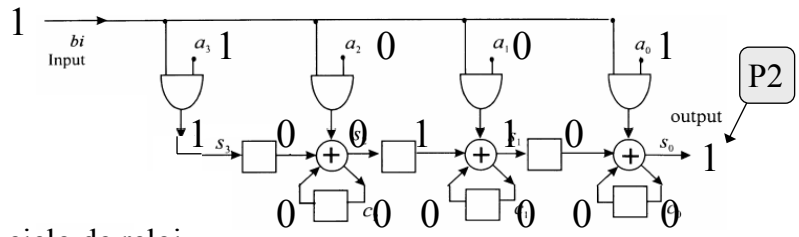



2 Segundo ciclo de reloj

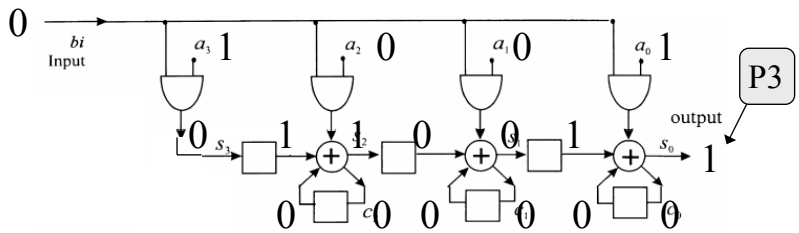


# Ejemplo multiplicación Serie-Paralelo CSAS (III)

3 Tercer ciclo de reloj



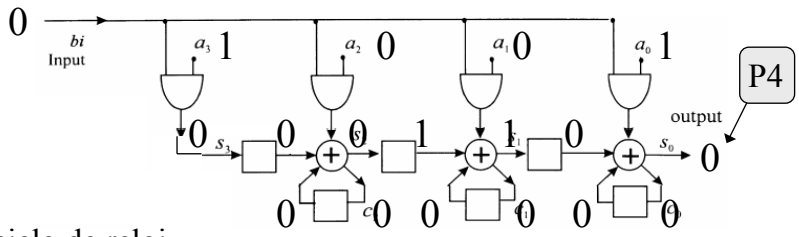

4 Cuarto ciclo de reloj



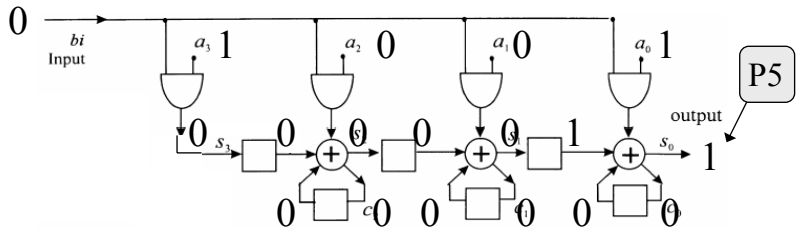



# Ejemplo multiplicación Serie-Paralelo CSAS (IV)

5 Quinto ciclo de reloj

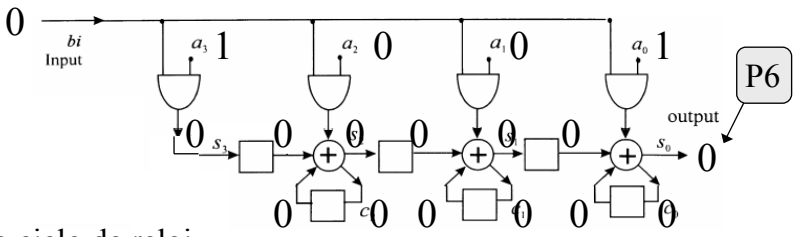



6 Sexto ciclo de reloj

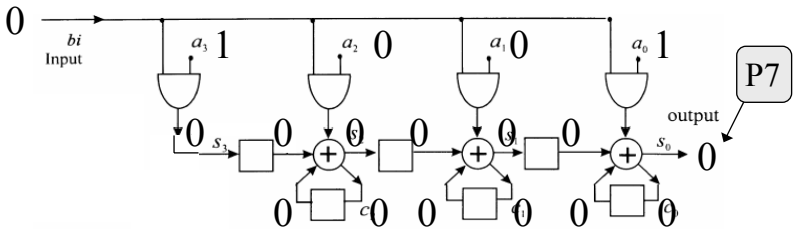


# Ejemplo multiplicación Serie-Paralelo CSAS (V)

7 Séptimo ciclo de reloj




8 Octavo ciclo de reloj





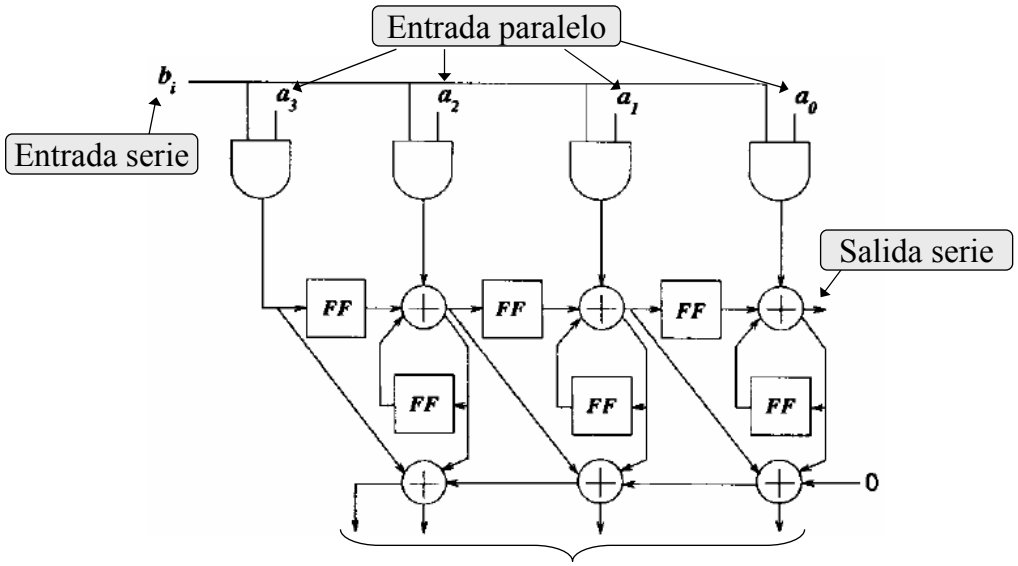

## *3.4.2.2 Multiplicadores Serie-Paralelo Unsigned con algoritmo FSP*

# SPM con algoritmo FSP

- ***FSP:*** *Fast Serial-Parallel*
- Realiza el producto en tan solo *N* ciclos de reloj (igual al número de bits del multiplicador u operando serie)
- Durante los primeros *N-1* ciclos funciona como una multiplicador *CSAS,* en el siguiente ciclo funciona como un sumador de *M-1* bits para extraer en paralelo los bits de carry remanentes



# Multiplicador FSP (Fast Serial-Parallel) de 4 bits



# Ejemplo multiplicación serie-paralelo FSP (I)

***Ejemplo:*** multiplicador SPM de 4 bits, producto de 9d x 5d

```
      1001  (9d) ←—— Dato paralelo
  x   0101  (5d) ←—— Dato serie
  ---------
      1001
    0000
+  1001
   0000
  --------
  00101101  (45d) ←
```

$P_7P_6P_5P_4P_3P_2P_1P_0$

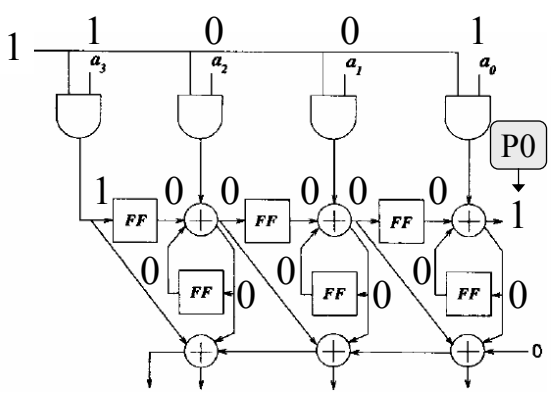
El resultado de 8 bits se obtiene después de 4 ciclos de reloj:

- 3 ciclos para hacer los 3 primeros productos parciales
- 1 ciclo para hacer el último producto parcial y extraer en paralelo los 4 bits de mayor peso

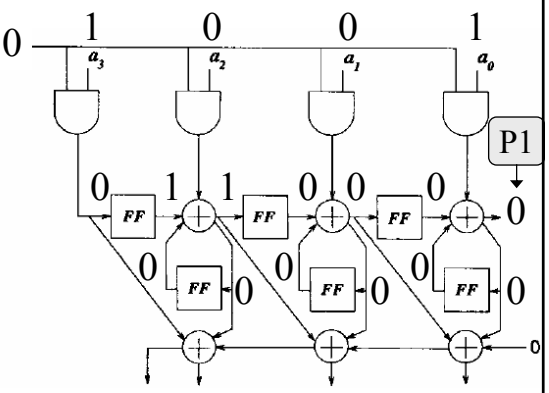


# Ejemplo multiplicación serie-paralelo FSP (II)

1 Primer ciclo de reloj

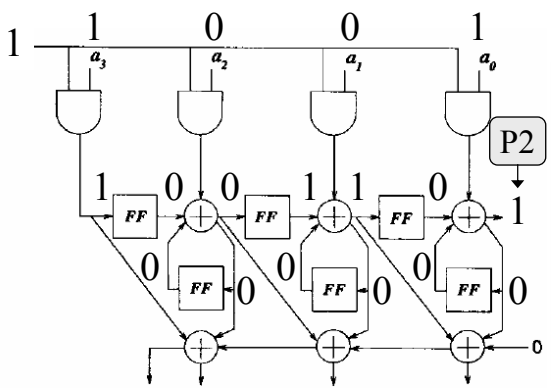



2 Segundo ciclo de reloj

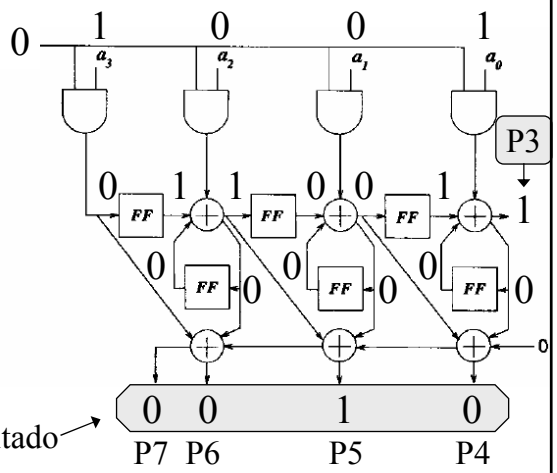


# Ejemplo multiplicación serie-paralelo FSP (III)

③ Tercer ciclo de reloj




④ Cuarto ciclo de reloj




Los cuatros bits de mayor peso del resultado



## *3.4.2.3 Multiplicadores Serie Paralelo c.a.2 con algoritmo CSAS*

# Multiplicador SPM c.a.2

Para realizar el producto serie-paralelo se utilizan dos técnicas:

- Extensión de signo: requiere más recursos hardware
- Algoritmo de Baugh-Wooley



# Algoritmo de extensión de signo

Se extiende el signo de ambos números *N+1* veces donde *N* es la longitud de los operandos que intervienen en el producto

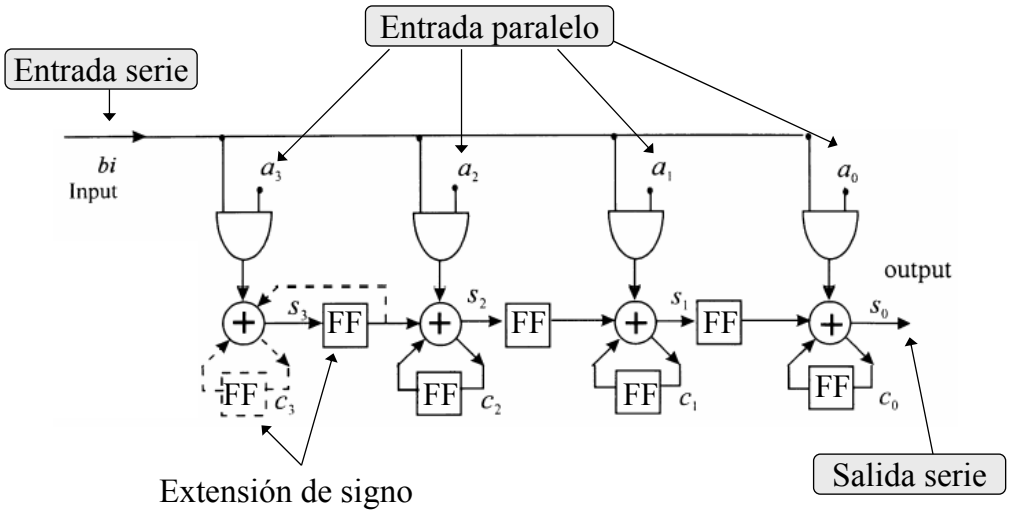
| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | x3 | x3 | x3 | x3 | x3 | x3 | x2 | x1 | x0 |
| | | | | | | | | | y3 | y3 | y3 | y3 | y3 | y3 | y2 | y1 | y0 |
| | | | | | | | | | x3y0 | x3y0 | x3y0 | x3y0 | x3y0 | x3y0 | x2y0 | x1y0 | x0y0 |
| + | | | | | | | | x3y1 | x3y1 | x3y1 | x3y1 | x3y1 | x3y1 | x2y1 | x1y1 | x0y1 | |
| + | | | | | | | x3y2 | x3y2 | x3y2 | x3y2 | x3y2 | x2y2 | x1y2 | x0y2 | | | |
| + | | | | | | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x2y3 | x1y3 | x0y3 | | | |
| + | | | | | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x2y3 | x1y3 | x0y3 | | | | |
| + | | | | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x2y3 | x1y3 | x0y3 | | | | | |
| + | | | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x2y3 | x1y3 | x0y3 | | | | | | |
| + | | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x2y3 | x1y3 | x0y3 | | | | | | | |
| + | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x3y3 | x2y3 | x1y3 | x0y3 | | | | | | | | |
| | | | | | | | | | p7 | p7 | p6 | p5 | p4 | p3 | p2 | p1 | p0 |

Términos que intervienen en el cálculo del resultado

Las operaciones a realizar son las mismas que en caso de números positivos

Resultado del producto

# Multiplicador Serie-Paralelo *c.a.2* CSAS de 4 bits con extensión de signo






# Ejemplo multiplicación Serie-Paralelo CSAS *c.a.2* con extensión de signo (I)

***Ejemplo:*** multiplicador SPM de 4 bits, producto de -5d x 3d

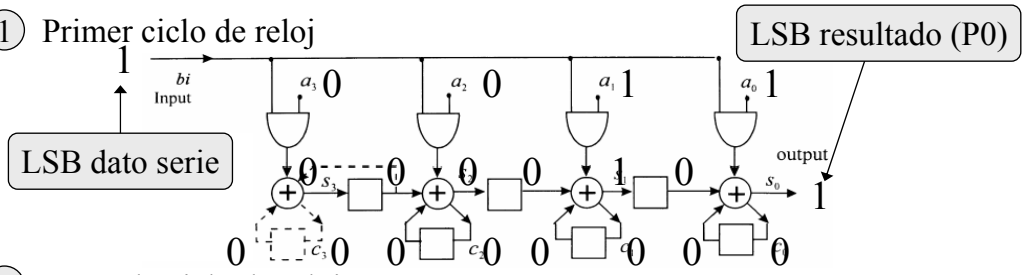
Extensión de signo

```
        000000011  (3d)  ← Dato paralelo
      x 11111 1011 (-5d) ← Dato serie
        000000011
       000000011
      000000000
     000000011
+/-  000000011
    000000011
   000000011
  00000001 1
 000000011
 111111111 11110001  (-15d)
```

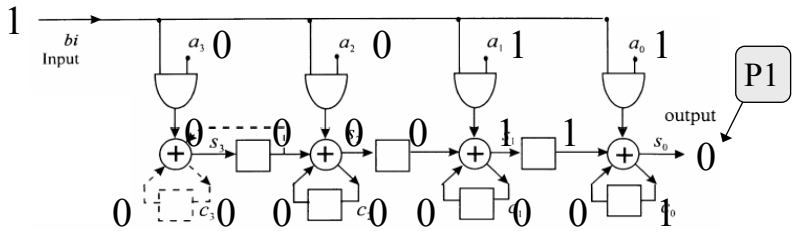
$P_7P_6P_5P_4P_3P_2P_1P_0$

# Ejemplo multiplicación Serie-Paralelo CSAS *c.a.2* con extensión de signo (II)

① Primer ciclo de reloj



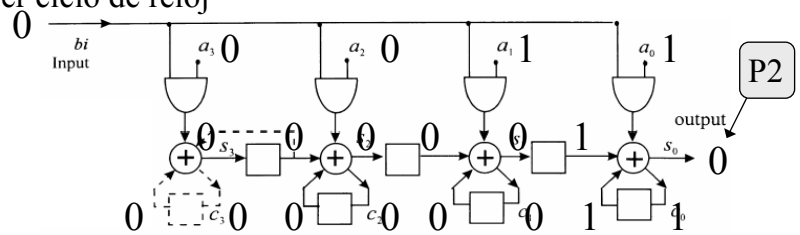

② Segundo ciclo de reloj



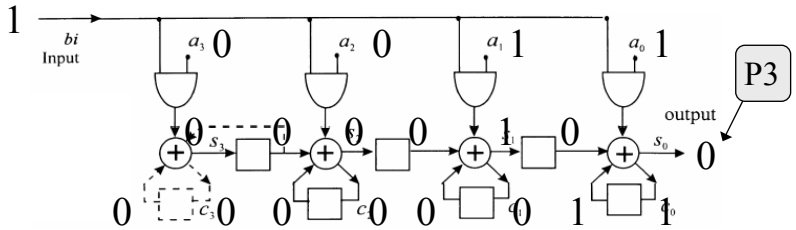



# Ejemplo multiplicación Serie-Paralelo CSAS c.a.2 con extensión de signo (III)

③ Tercer ciclo de reloj

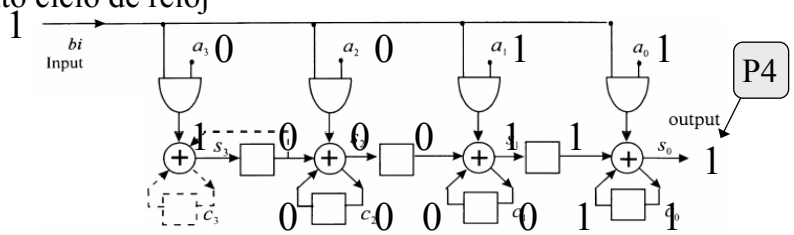



④ Cuarto ciclo de reloj

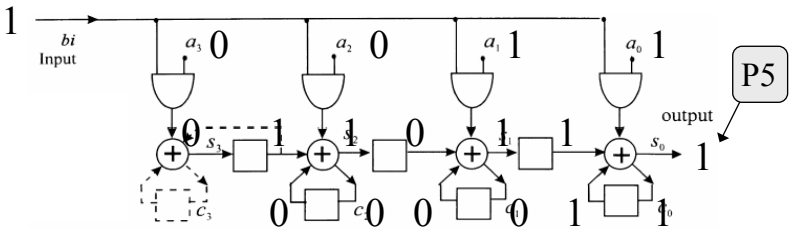


# Ejemplo multiplicación Serie-Paralelo CSAS c.a.2 con extensión de signo (IV)

5 Quinto ciclo de reloj



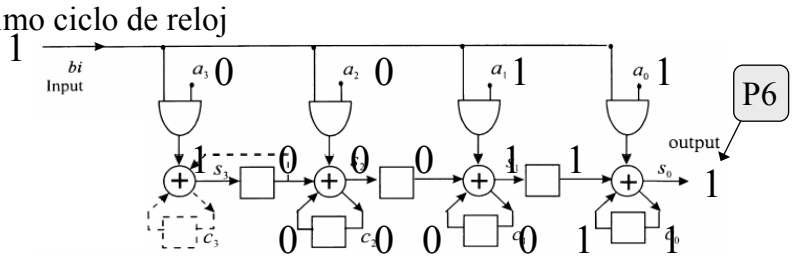

6 Sexto ciclo de reloj



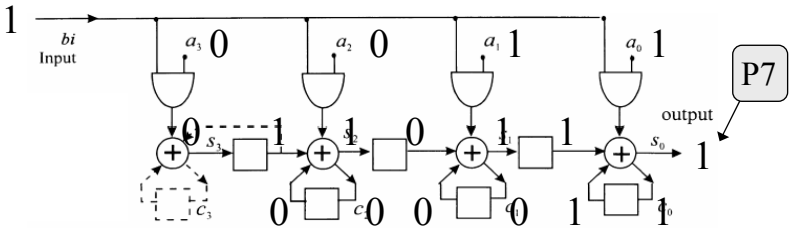



# Ejemplo multiplicación Serie-Paralelo CSAS c.a.2 con extensión de signo (V)

7 Séptimo ciclo de reloj




8 Octavo ciclo de reloj

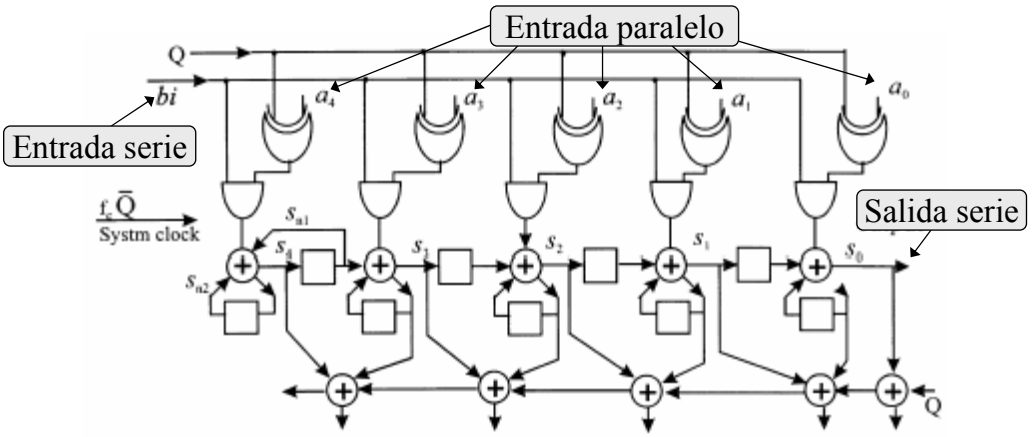


# 3.4.2.4 Multiplicadores Serie Paralelo c.a.2 con algoritmo FSP



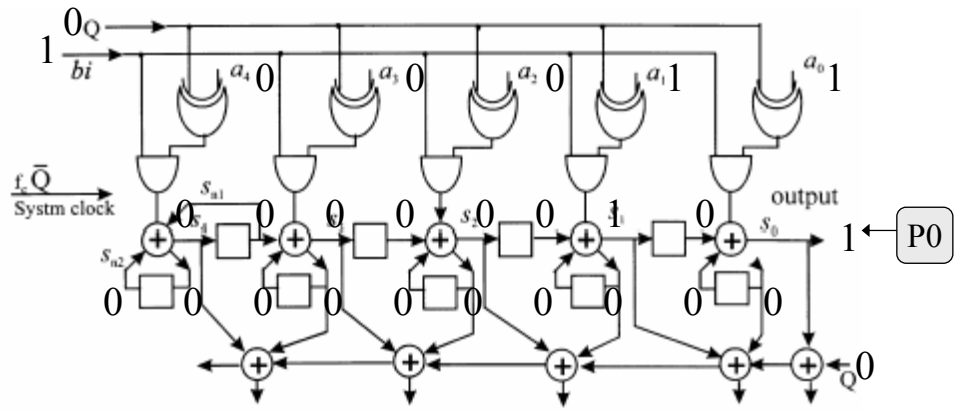
## Multiplicación Serie-Paralelo FSP *c.a.2* de 4 bits con extensión de signo

- Vale cero el resto durante los *N-1* primeros periodos de reloj y el circuito actúa como un CSAS
- La entrada Q vale uno en el ciclo *N* de reloj



# Ejemplo de multiplicación Serie-Paralelo FSP *c.a.2* de 4 bits con extensión de signo (I)

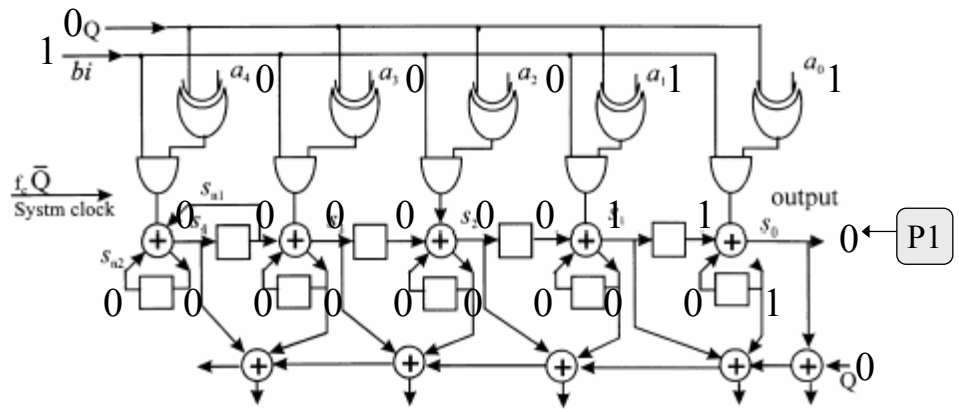
① Primer ciclo de reloj





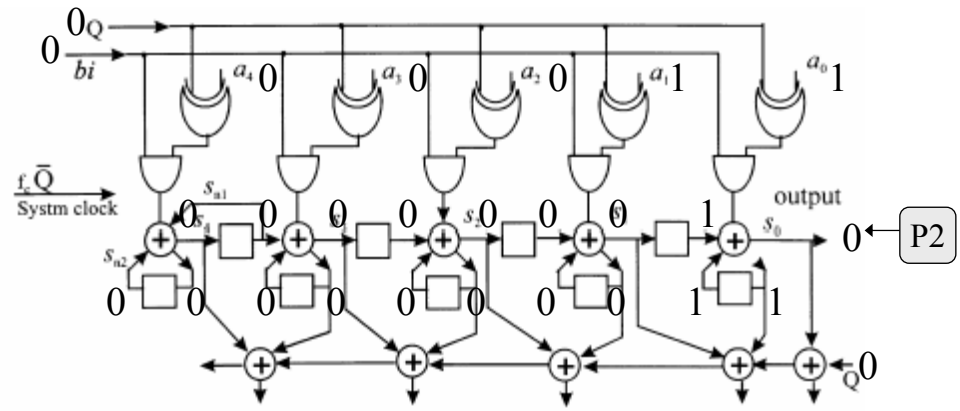

# Ejemplo de multiplicación Serie-Paralelo FSP *c.a.2* de 4 bits con extensión de signo (II)

② Segundo ciclo de reloj



# Ejemplo de multiplicación Serie-Paralelo FSP *c.a.2* de 4 bits con extensión de signo (III)

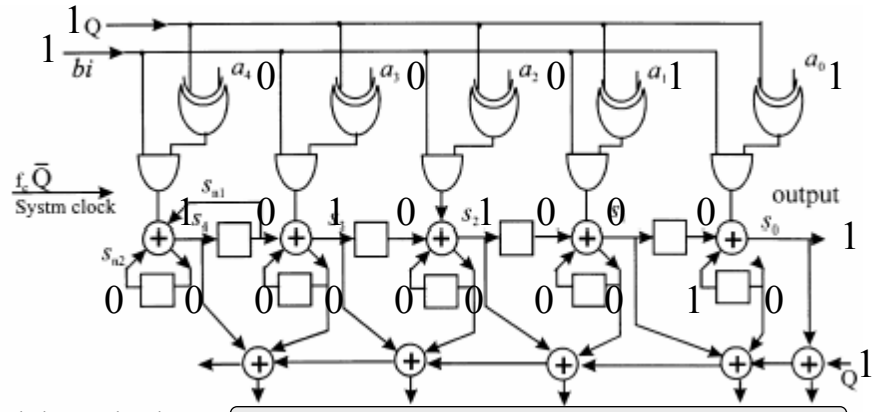
③ Tercero ciclo de reloj






# Ejemplo de multiplicación Serie-Paralelo FSP *c.a.2* de 4 bits con extensión de signo (IV)

④ Cuarto ciclo de reloj



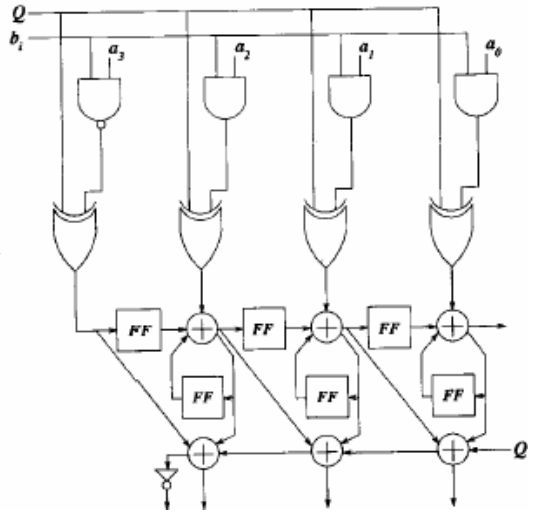

5 MSB del resultado →

| P7 | P6 | P5 | P4 | P3 |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 |

# Multiplicación Serie-Paralelo FSP
# *c.a.2* de 4 bits con algoritmo Baugh-Wooley (I)

El hardware es más reducido que en caso de FSP con signo extendido

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| | | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| | | | | $\overline{a_3b_0}$ | $a_2b_0$ | $a_1b_0$ | $a_0b_0$ |
| | | | $\overline{a_3b_1}$ | $a_2b_1$ | $a_1b_1$ | $a_0b_1$ | |
| | | $\overline{a_3b_2}$ | $a_2b_2$ | $a_1b_2$ | $a_0b_2$ | | |
| 1 | $a_3b_3$ | $\overline{a_2b_3}$ | $\overline{a_1b_3}$ | $\overline{a_0b_3}$ | | | |
| | | | 1 | | | | |
| $p_7$ | $p_6$ | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ |






# Multiplicación Serie-Paralelo FSP
# *c.a.2* de 4 bits con algoritmo Baugh-Wooley (II)

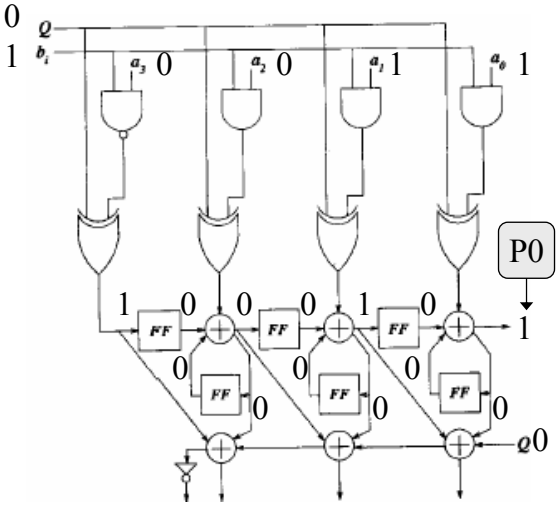
- Durante los *N-1* primeros productos parciales Q = 0, y se realizan una operación NAND y *N-1* AND
- En el último producto parcial, Q = 1 y se realiza una operación AND y *N-1* NAND

***Ejemplo:*** multiplicador SPM de 4 bits, producto de 3d x -5d

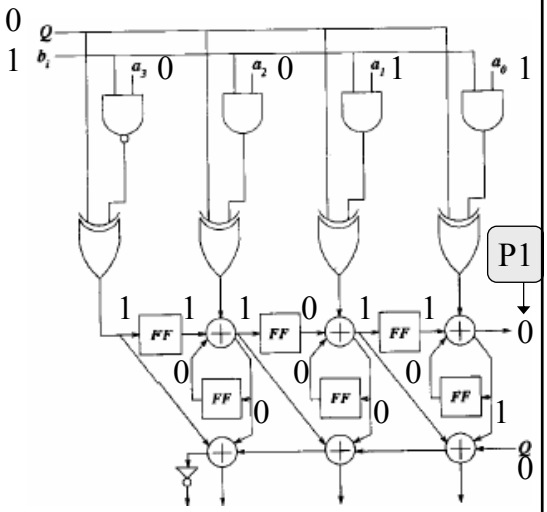
```
          0011  (3d) ←— Dato paralelo
        x 1011  (-5d) ←— Dato serie
        ------
          1011
         1011
    +   1000
       10100
          1
        --------
P7P6P5P4P3P2P1P0 → 11110001  (-15d)
```

# Ejemplo de multiplicación Serie-Paralelo FSP *c.a.2* de 4 bits con algoritmo Baugh-Wooley (I)

① Primer ciclo de reloj



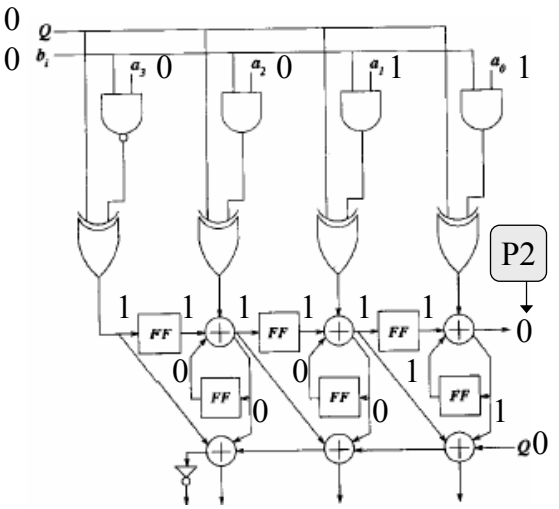

② Segundo ciclo de reloj



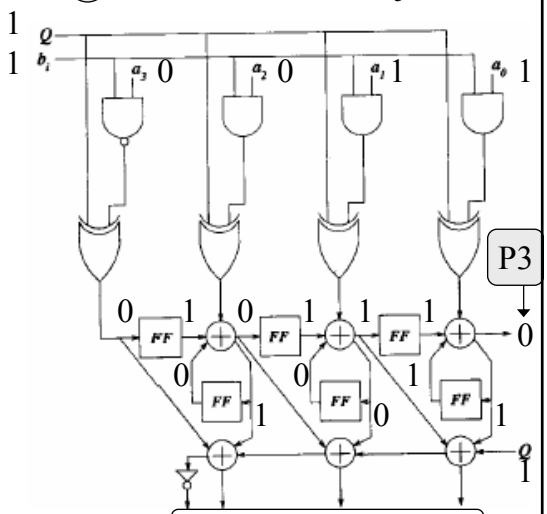



# Ejemplo de multiplicación Serie-Paralelo FSP c.a.2 de 4 bits con algoritmo Baugh-Wooley (II)

③ Tercer ciclo de reloj




④ Cuarto ciclo de reloj




P7 P6 P5 P4 → 1 1 1 1

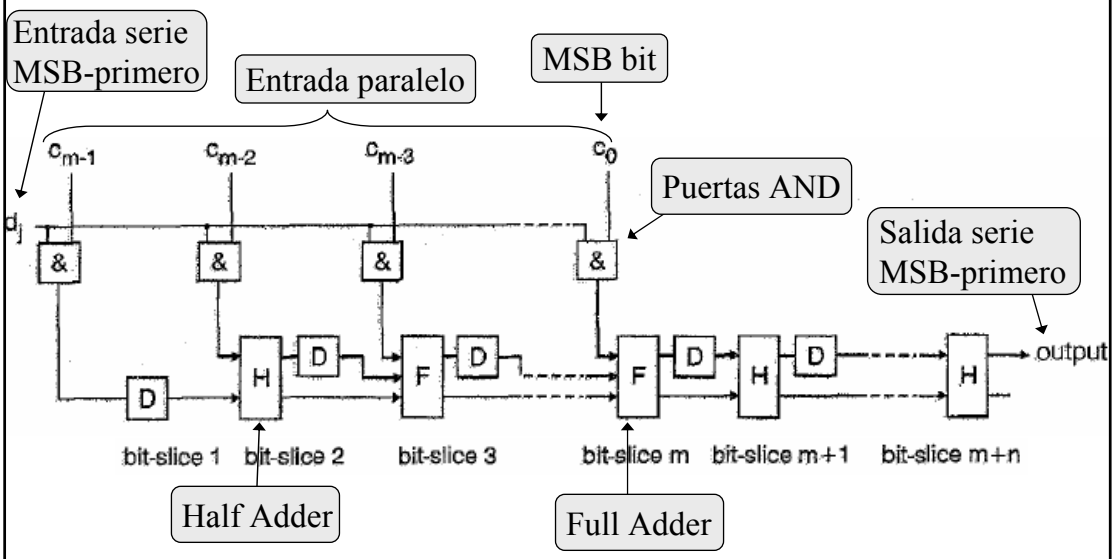
# *3.4.2.5 Multiplicadores Serie Paralelo MSB-primero*



## Introducción

- El en SPM MSB-primero, el operando serie se introduce en el multiplicador empezando por el bit de mayor peso y el resultado se obtiene empezando por el MSB
- Ventajas:
  - Los convertidores ADC de aproximaciones sucesivas y pipelined proporcionan el resultado de la conversión empezando por el MSB
  - Los algoritmos que calculan la raíz cuadrada y la división son del tipo MSB-primero
  - Con el multiplicador MSB-primero es más fácil truncar el resultado o eliminar los bits menos representativos
  - La ausencia de realimentación en la estructura de este tipo de multiplicadores permite incrementar la velocidad del reloj

# Multiplicador Serie-Paralelo MSB-primero *Unsigned*





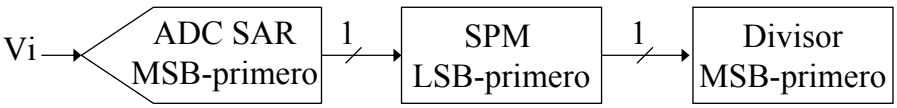

# Latencia en el SPM MSB-primero *Unsigned*

***Latencia***: nº de ciclos de reloj desde que entra el MSB del dato hasta que sale el MSB del resultado

- En el multiplicador MSB-primero *Unsigned* la latencia es igual al número de bits del operando serie
- En el multiplicador LSB-primero la latencia es de un ciclo de reloj

En determinados casos es conveniente utilizar el MSB-primero ya que la latencia global del diseño es menor

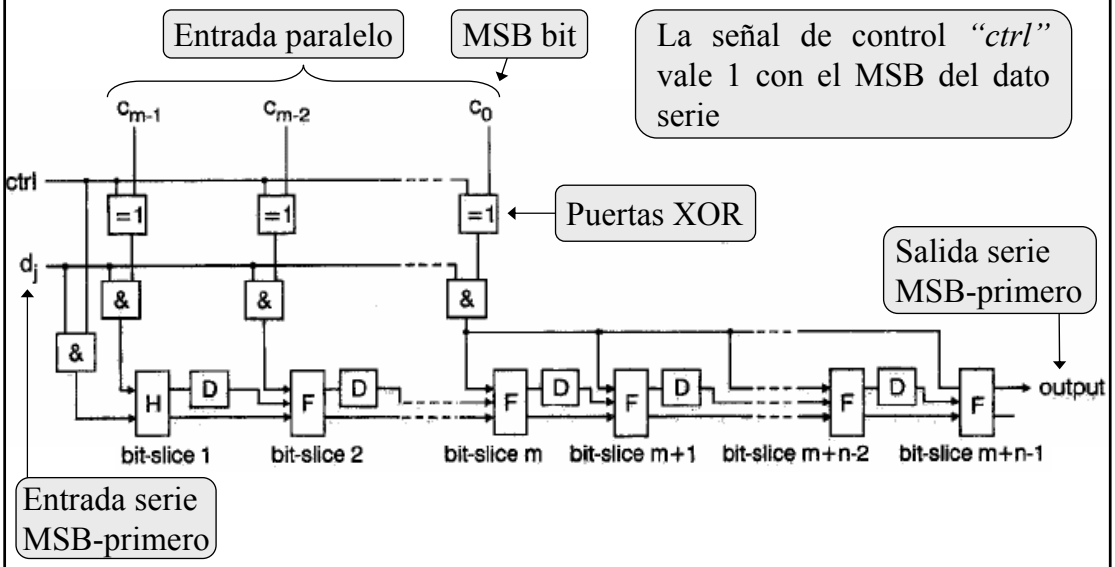
***Ejemplo:*** Cálculo de la latencia global de un sistema procesador con multiplicador de *MxN*




***Latencia***: N clocks + M+N clock = M+2N

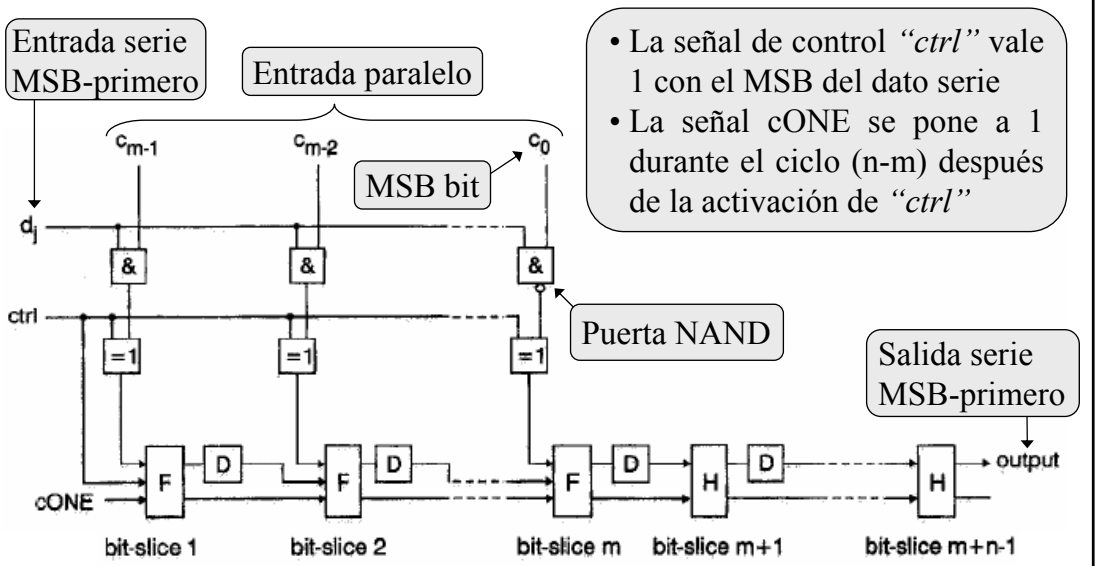
Con multiplicador MSB-primero la latencia es: 1 clock + N clock = N+1

# Multiplicador Serie-Paralelo MSB-primero *c.a.2* con signo extendido






# Multiplicador Serie-Paralelo MSB-primero *c.a.2* con algoritmo Baugh-Wooley



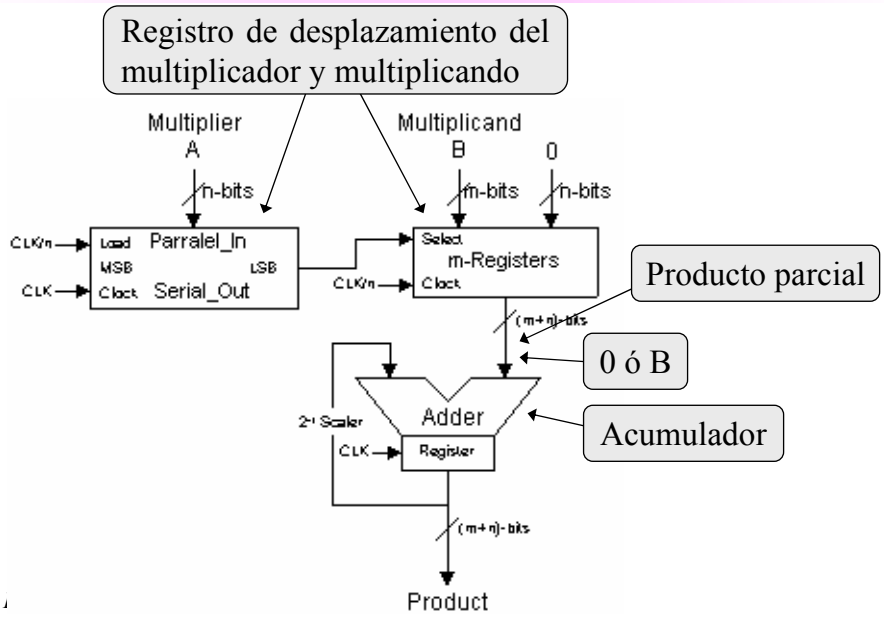
# 3.4.3 Multiplicadores Serie Secuencial



## Algoritmo multiplicador Serie Secuencial

- El algoritmo de funcionamiento es similar al multiplicador Serie-Paralelo
- La diferencia está en la utilización de un sumador paralelo en lugar de un sumador serie para sumar los productos parciales
- Algoritmo:
  - El multiplicando (*M* bits) y el multiplicador (*N* bits) se cargan en sendos registros de desplazamiento
  - Se realizan los productos parciales de los bits del multiplicador (bit a bit) con el multiplicando empezando por el LSB
  - Los sucesivos productos parciales se suman con el acumulador
  - El resultado definitivo se obtiene después de $N+M$ ciclos de reloj

# Arquitectura multiplicador serie-secuencial






# *3.4.4 Multiplicadores Serie-Serie*

# Algoritmo del multiplicador Serie-Serie

- Mediante un cálculo iterativo se puede extraer el valor del producto de dos números X e Y de *N* bits de longitud
- El algoritmo es válido para números positivos y para números en ***c.a.2***
- Si los dos números no son de igual longitud, se debe igualar su longitud:
  - ***Nº's Unsigned***: se rellena con ceros a la izquierda
  - ***Nº's Signed***: se extiende el signo
- El resultado se obtiene después de *2N* iteraciones

Algoritmo de cálculo de producto

$$\overline{Q}_i = \left[\frac{1}{2}\cdot\left(\overline{Q}_{i-1} + x_i \cdot Y_{i-1} + y_i \cdot X_{i-1} + 2^i \cdot x_i \cdot y_i\right)\right] \quad para \quad i < N$$

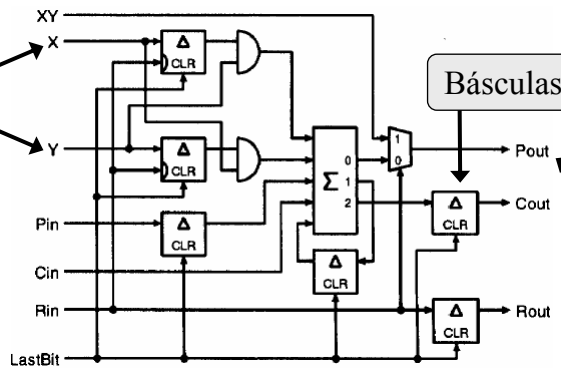
$$\overline{Q}_i = \left[\frac{1}{2}\cdot\left(\overline{Q}_{i-1} + x_{N-1} \cdot Y_{N-2} + y_{N-1} \cdot X_{N-2}\right)\right] \quad para \quad i \geq N$$

$$x_i : entrada \quad actual, \quad X_i : entrada \quad anterior$$



# Célula básica del multiplicador Serie-Serie

Entradas de datos serie, LSB primero

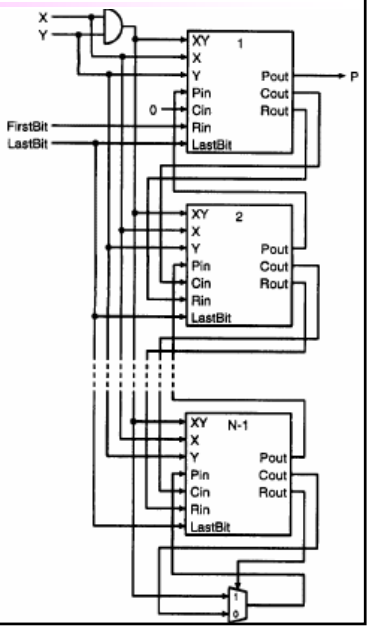



Básculas D

Salida serie del resultado (LSB primero)

# Multiplicador Serie-Serie

- Se necesitan *N-1* células básicas para hacer el producto de dos números de *N* bits
- El resultado se obtiene después de *2N* ciclos de reloj
- Durante los *N* primeros ciclos se introducen los 2 operandos (*LSB* primero) por las entradas serie
- Durante el resto de ciclos se introducen ceros (*nº positivos*) o se extiende el signo (*nº c.a.2*)
- Entradas de control del multiplicador:
  - ***FirsBit:*** Se pone a nivel alto durante el primer ciclo de señal de reloj
  - ***LastBit:*** Se pone a 1 durante el ciclo que precede a la siguiente multiplicación






# *3.4.5 Multiplicadores Pipelined*

# Multiplicación 8x8 (I)

- Los multiplicadores pipelined permiten aumentar la velocidad de cálculo realizando varios productos en paralelo
- El producto se puede realizar mediante con 4 multiplicadores y 2 etapas de sumadores

X[7..4] parte alta del dato X
X[3..0] parte baja del dato X
Y[7..4] parte alta del dato Y
Y[3..0] parte baja del dato Y

$$
\begin{array}{lll}
 & & X[7..4]\ X[3..0] \\
 & & \times\ Y[7..4]\ Y[3..0] \\
\hline
 & X[7..4] \times Y[3..0] & X[3..0] \times Y[3..0] \\
+\ X[7..4] \times Y[7..4] & X[3..0] \times Y[7..4] & \\
\hline
\end{array}
$$

$$X[7..4] \times Y[7..4] \times 16^2 + ((X[7..4] \times Y[3..0]) + (X[3..0] \times Y[7..4])) \times 16^1 + X[7..4] \times Y[3..0] \times 16^0$$



# Multiplicación 8x8 (II)

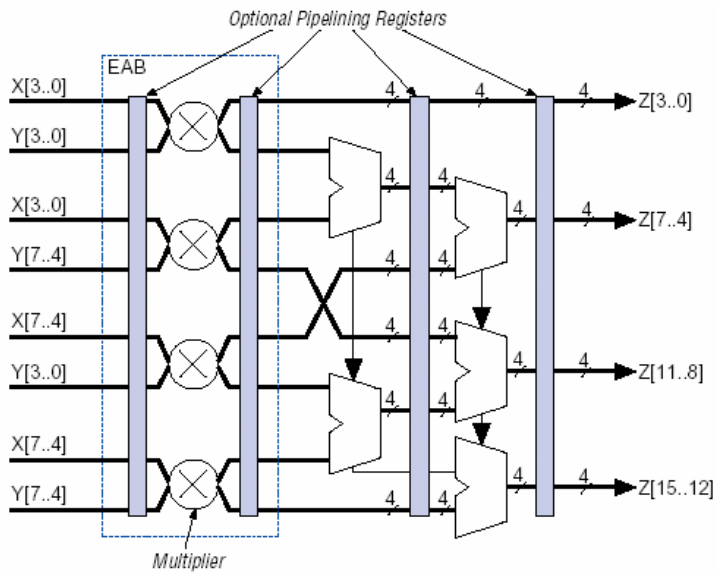
- Cada producto parcial se realiza con un multiplicador de 4x4 bits
- Los resultados parciales de los productos se suman con 2 etapas de sumadores de 4+4 bits

$$
\begin{array}{cccccccccccccccc}
 & & & & & & & & R_7 & R_6 & R_5 & R_4 & R_3 & R_2 & R_1 & R_0 \\
 & & & & S_7 & S_6 & S_5 & S_4 & S_3 & S_2 & S_1 & S_0 & & & & \\
 & & & & T_7 & T_6 & T_5 & T_4 & T_3 & T_2 & T_1 & T_0 & & & & \\
+ & U_7 & U_6 & U_5 & U_4 & U_3 & U_2 & U_1 & U_0 & & & & & & & \\
\hline
Q_{15} & Q_{14} & Q_{13} & Q_{12} & Q_{11} & Q_{10} & Q_9 & Q_8 & Q_7 & Q_6 & Q_5 & Q_4 & Q_3 & Q_2 & Q_1 & Q_0
\end{array}
$$

Where $R = X[3..0] \times Y[3..0]$
$S = X[3..0] \times Y[7..4]$
$T = X[7..4] \times Y[3..0]$
$U = X[7..4] \times Y[7..4]$

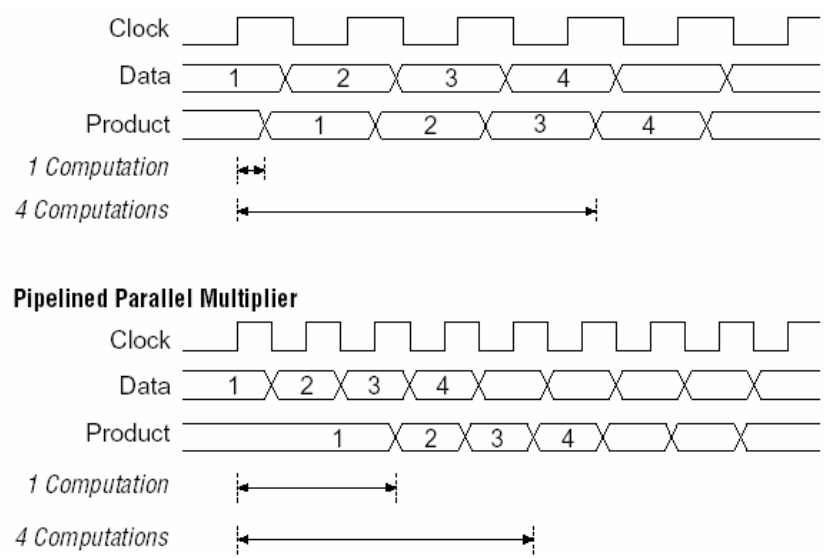
☐ Addition performed in the first stage
▒ Addition performed in the second stage

# Multiplicación 8x8 (III)






# Multiplicación 8x8. Pipelined vs. Non-Pipelined

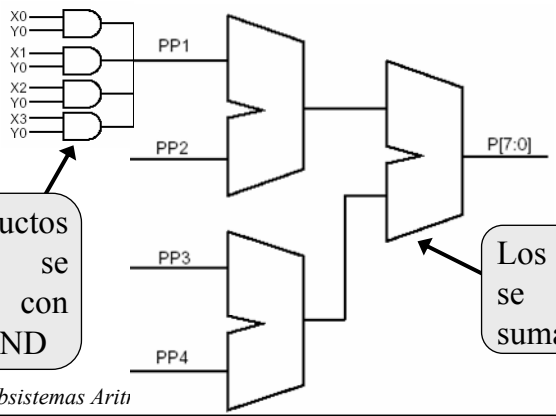


# 3.4.6 Algoritmo L-Booth



## Implementación clásica del multiplicador

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Multiplicand | > | | | | | X3 | X2 | X1 | X0 | |
| Multiplier | > | | | | x | Y3 | Y2 | Y1 | Y0 | |
| 1st partial product | > | | | | | Y0X3 | Y0X2 | Y0X1 | Y0X0 | → PP1 |
| 2nd partial product | > | | | | Y1X3 | Y1X2 | Y1X1 | Y1X0 | | → PP2 |
| 3rd partial product | > | | | Y2X3 | Y2X2 | Y2X1 | Y2X0 | | | → PP3 |
| 4th partial product | > | + | Y3X3 | Y3X2 | Y3X1 | Y3X0 | | | | → PP4 |
| Final product | > | P7 | P6 | P5 | P4 | P3 | P2 | P1 | P0 | |




Los productos parciales se obtienen con puertas AND

Los productos parciales se suman con tres sumadores binarios

# Producto 4x4 con algoritmo L-Booth

El producto se realiza en dos partes: X[3..0] x Y[1,0] y X[3..0] x Y[3,2]

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | X3 | X2 | X1 | X0 |
| Only 2 LSB used | > | | | | x | | | Y1 | Y0 |
| if Y1=0,Y0=0 | \| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| if Y1=0, Y0=1 | \| | 0 | 0 | 0 | 0 | X3 | X2 | X1 | X0 |
| if Y1=1, Y0=0 | \| | 0 | 0 | 0 | X3 | X2 | X1 | X0 | 0 |
| if Y1=1, Y0=1 | \| | 0 | 0 | 0 | X3 | X3+X2 | X2+X1 | X1+X0 | 0 |
| Multiplexer A result | > | 0 | 0 | PPA5 | PPA4 | PPA3 | PPA2 | PPA1 | PPA0 |
| | | | | | | X3 | X2 | X1 | X0 |
| Only 2 MSB used | > | | | | x | Y3 | Y2 | | |
| if Y3=0,Y2=0 | \| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| if Y3=0, Y2=1 | \| | 0 | 0 | X3 | X2 | X1 | X0 | 0 | 0 |
| if Y3=1, Y2=0 | \| | 0 | X3 | X2 | X1 | X0 | 0 | 0 | 0 |
| if Y2=1, Y2=1 | \| | 0 | X3 | X3+X2 | X2+X1 | X1+X0 | 0 | 0 | 0 |
| Multiplexer B result | > | PPB7 | PPB6 | PPB5 | PPB4 | PPB3 | PPB2 | 0 | 0 |
| | | 0 | 0 | PPA5 | PPA4 | PPA3 | PPA2 | PPA1 | PPA0 |
| | + | PPB7 | PPB6 | PPB5 | PPB4 | PPB3 | PPB2 | 0 | 0 |
| Final product | > | P7 | P6 | P5 | P4 | P3 | P2 | P1 | P0 |

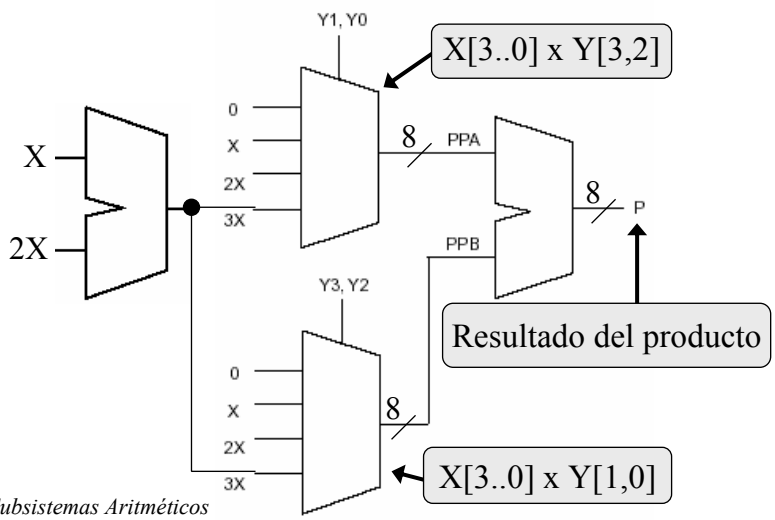
Productos parciales

El producto final es la suma de los dos productos parciales



# Implementación del multiplicador con el algoritmo L-Booth

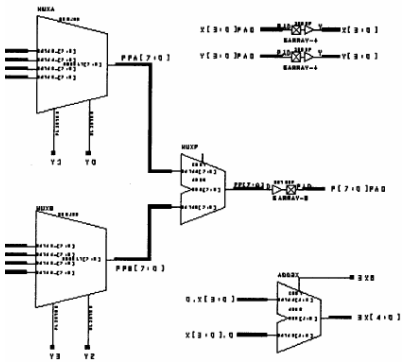
La implementación del algoritmo ***L-Booth*** está basada en multiplexores que es ideal para la arquitectura interna de las FPGAs de ***Actel***



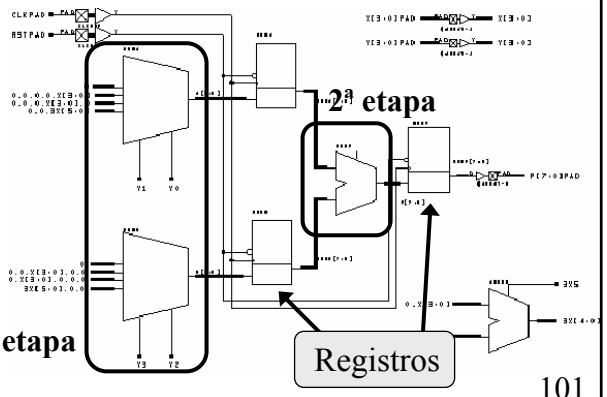
# Multiplicador L-Booth Pipelined

- Para aumentar la velocidad se introducen registros entre los niveles de lógica
- La distribución de registros es óptima cuando cada etapa tiene un retardo similar

Implementación combinacional



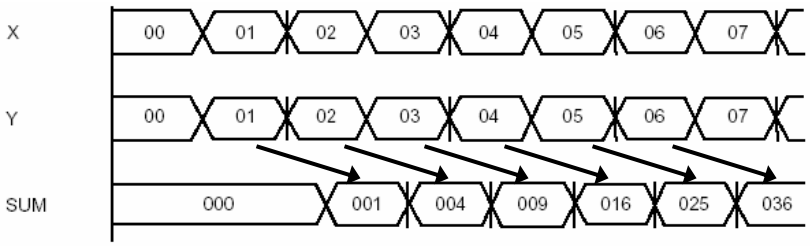
Implementación pipelined






# Latencia y velocidad de cálculo con implementación Pipelined

- Con la implementación pipelined el resultado se retarda tantos ciclos de reloj como etapas de registros se añaden.
- En este caso la latencia o retardo que existe entre las entradas de datos y la salida del sumador es de un ciclo de reloj



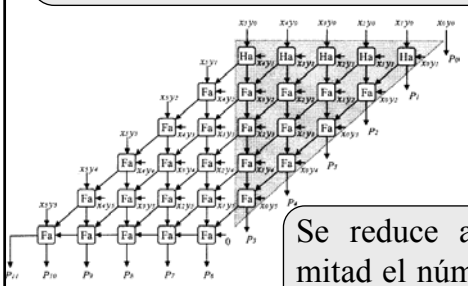

Velocidad de cálculo del multiplicador implementado con la FPGA de ***Actel*** *1225XL-1*: **combinacional -> 24 MHz**
**pipelined -> 57MHz**

# 3.4.7 Multiplicadores de hardware reducido

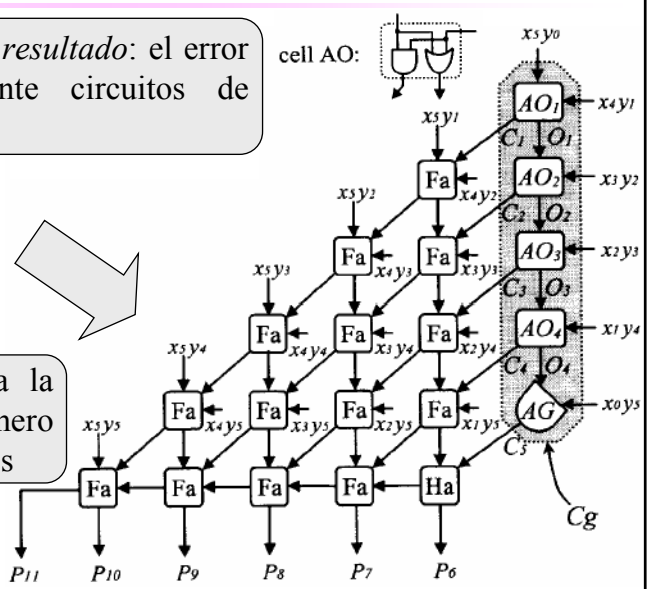


## Multiplicador paralelo binario con hardware reducido

*Truncar el número de bits del resultado*: el error cometido se reduce mediante circuitos de generación de carry

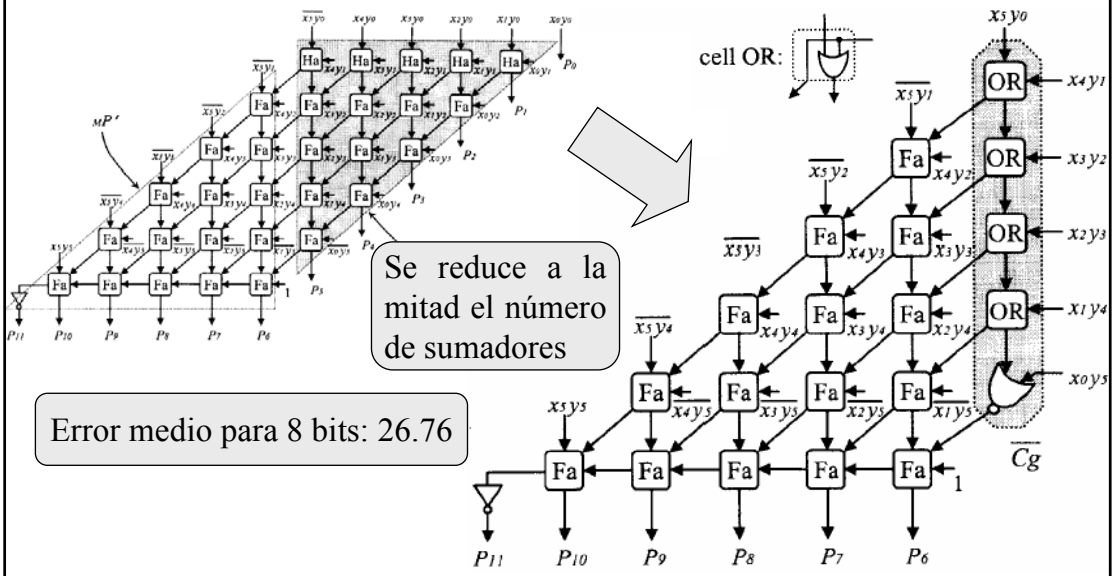


Se reduce a la mitad el número de sumadores




Error medio para 8 bits: 26.76

# Multiplicador paralelo *c.a.2* con hardware reducido




Error medio para 8 bits: 26.76

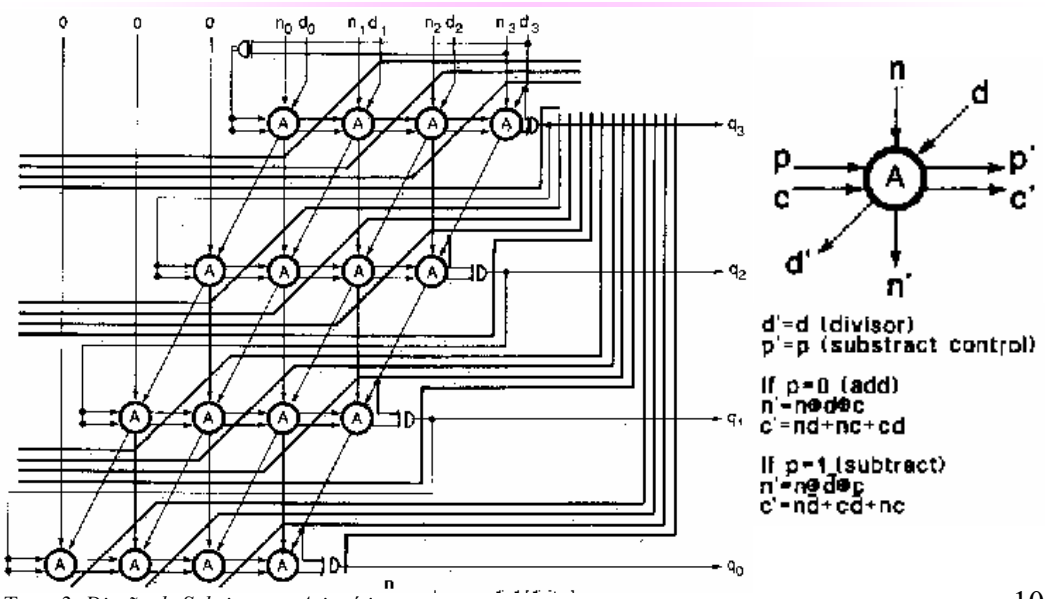


## *3.5 Divisores*

# 3.5.1 Divisor Paralelo-Paralelo



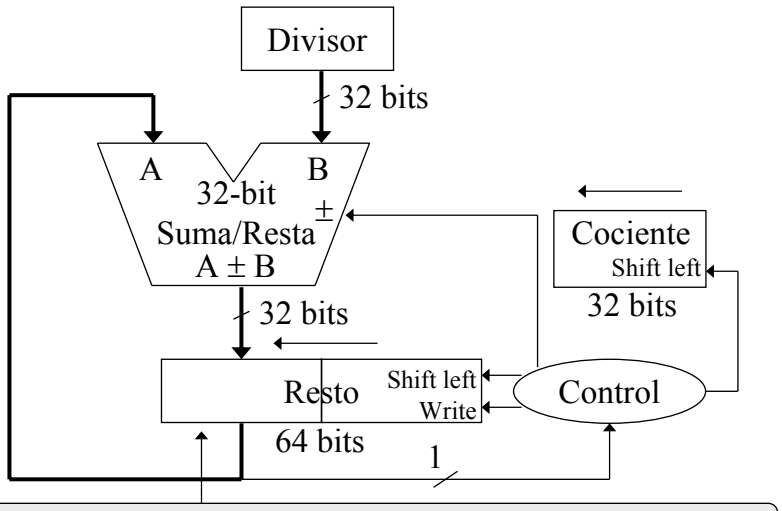
## Divisor Paralelo-Paralelo



# 3.5.2 Divisor secuencial

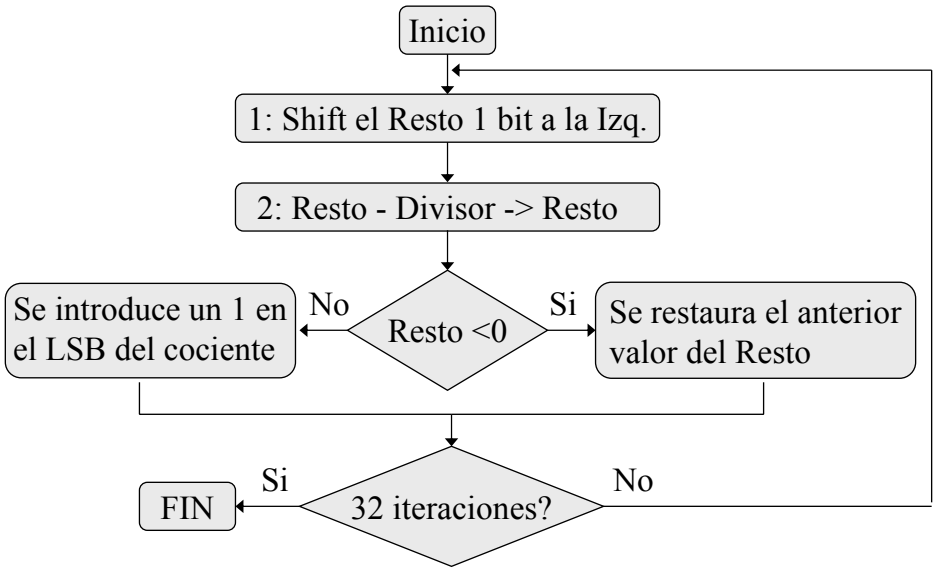


## Esquema divisor secuencial de 32 bits




Inicialmente se carga el registro del resto con el dividendo

# Algoritmo de la división secuencial






# Ejemplo de división con el algoritmo secuencial

División de 0000 0111 por 0010: cociente = 0011

| Iteración | Paso | Cociente | Divisor | Resto |
|---|---|---|---|---|
| 0 | Valor inicial | 0000 | 0010 | 0000 0111 |
| 1 | 1: Shift Resto Izq. | 0000 | 0010 | 0000 1110 |
| | 2: Resto = Resto-Divisor | 0000 | 0010 | 1110 1110 |
| | 3b: Resto<0 => +Div, sll Q, Q0=0 | 0000 | 0010 | 0000 1110 |
| 2 | 1: Shift Resto Izq. | 0000 | 0010 | 0001 1100 |
| | 2: Resto = Resto-Divisor | 0000 | 0010 | 1111 1100 |
| | 3b: Resto<0 => +Div, sll Q, Q0=0 | 0000 | 0010 | 0001 1100 |
| 3 | 1: Shift Resto Izq. | 0000 | 0010 | 0011 1000 |
| | 2: Resto = Resto-Divisor | 0000 | 0010 | 0001 1000 |
| | 3a: Resto≥0 => sll Q, Q0=1 | 0001 | 0010 | 0001 1000 |
| 4 | 1: Shift Resto Izq. | 0001 | 0010 | 0011 0000 |
| | 2: Resto = Resto-Divisor | 0001 | 0010 | 0001 0000 |
| | 3a: Resto≥0 => sll Q, Q0=1 | 0011 | 0010 | 0001 0000 |

***sll***: desplazamiento a la izquierda

## *3.6 Operaciones aritméticas en coma flotante*



## *3.6.1 Formatos de representación en coma flotante*

# Representación en coma flotante
## *Standard IEEE* 754 (I)

Existen diferentes formatos para representar números en coma flotante
Formato IEEE 754:

$$(-1)^S \times (1+\text{Mantisa}) \times 2^E$$

Simple precisión:

| 1 bit | 8 bits | 23 bits |
|---|---|---|
| Signo | Exponente | Mantisa |

Doble precisión:

| 1 bit | 11 bits | 20 bits |
|---|---|---|
| Signo | Exponente | Mantisa |

| 32 bits |
|---|
| Mantisa |



# Representación en coma flotante
## *Standard IEEE 754* (II)

**Signo** (S): 0 para números positivos; 1 para números negativos

**Mantisa**:
- Representa un número ***unsigned*** fraccionario entre 0 y 1
- El *MSB* de la mantisa tiene peso $2^{-1}$

**Exponente**:
- Se utiliza formato llamado ***biased notation***
- Este formato permite representar números positivos y negativos utilizando solo números positivos
- Al manejar solo números positivos se simplifica el cálculo del exponente en operaciones aritméticas

## Exponente en formato ***Bias Notation***

- Para simple precisión se debe sumar al exponente el valor 127 y para doble precisión el valor 1023

- Para simple precisión el margen de valores del exponente con formato *Bias Notation* y en valor real es:

  Formato *Bias* del exponente: 0 <-> 255
  Valor real del exponente: -127 <-> 128

- Para doble precisión el margen de valores del exponente con formato *Bias Notation* y en valor real es:

  Formato *Bias* del exponente: 0 <-> 2047
  Valor real del exponente: -1023 <-> 1024



## Ejemplos de formato IEEE 754 (I)

***Ejemplo 1***: El número **-0.75** decimal en binario sería **-1.1 x $2^{-1}$**

- La representación en coma flotante con simple precisión es:

$$(-1)^S \times (1 + \text{Mantisa}) \times 2^{(\text{exponente real}+127)}$$

lo que dá lugar:

$$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{(-1+127)}$$

Signo: 1
Mantisa: 1000 0000 0000 0000 0000 000
Exponente: 0111 1110

- Con doble precisión:

$(-1)^1 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000) \times 2^{(-1+1023)}$

Signo: 1
Mantisa: 1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
Exponente: 01111111110

# Ejemplos de formato IEEE 754 (II)

***Ejemplo 2***: Cúal es el valor del número siguiente en decimal

Signo: 1

Mantisa: 0100 0000 0000 0000 0000 000

Exponente: 1000 0001

$(-1)^1 \times (1 + .0100\ 0000\ 0000\ 0000\ 0000\ 000) \times 2^{(129-127)}$

$(-1)^1 \times (1 + 0.25) \times 2^{(129-127)} = -1 \times 1.25 \times 2^2 = -1.25 \times 4 = -5.0$



# Notación coma flotante de Altera

**Signo** (S): 1 para números positivos; 0 para números negativos

**Mantisa**:

- Representa un número ***unsigned*** fraccionario entre 0 y 1
- El *MSB* de la mantisa tiene peso $2^{-1}$
- El resultado de las operaciones se ajusta para que *MSB* = 1 y aprovechar al máximo la precisión que proporciona la mantisa

**Exponente**:

- Se utiliza el formato llamado ***offset*** $2^{(n-1)}$
- Este formato permite representar números positivos y negativos utilizando solo números positivos
- Al manejar solo números positivos se simplifica el cálculo del exponente en operaciones aritméticas

# Exponente en formato *offset* $2^{(n-1)}$

- Se debe sumar al exponente el valor $2^{(n-1)}$, donde $n$ es el número de bits utilizados para expresar el exponente

- Por ejemplo para $n = 7$, $2^{(n-1)} = 64$, y el margen de valores del exponente en formato *offset* $2^{(n-1)}$ y en valor real es:

| | |
|---|---|
| Formato *offset* del exponente: | 0 <-> 127 |
| Valor real del exponente: | -64 <-> 63 |

- Ejemplos:

| Exponente real | Formato *offset* $2^{(n-1)}$ |
|---|---|
| 10 | 74 |
| -10 | 54 |



# Ejemplos de coma flotante con formato de Altera (I)

***Mantisa de 8 bits y exponente de 7 bits***

***Ejemplo 1***: Mayor número positivo:

$+ 1111\ 1111b\ 1111111b$
$= +0.1111\ 1111b \times 2^{(1111111b-1000000b)}$
$= +0.1111\ 1111b \times 2^{0111111b}$
$= +0.1111\ 1111b \times 2^{63d}$
$= +1111\ 1111.0b \times 2^{55d}$
$= +255d \times 2^{55d} = +9.187343239836\ e+18$

***Ejemplo 2***: Mayor número negativo:

$-1111\ 1111b\ 1111111b$
$= -255d \times 2^{55d} = -9.187343239836\ e+18$

# Ejemplos de coma flotante con formato de Altera (II)

***Ejemplo 3***: Menor número (próximo a cero):

$\pm$ 1000 0000b 0000000b

$= \pm$ 0.1000 0000b x $2^{(0000000b-1000000b)}$

$= \pm$ 0.1000 0000b x $2^{(0d - 64d)}$

$= \pm$ 0.1000 0000b x $2^{-64d}$

$= \pm$ 1000 0000.0b x $2^{-72d}$

$= \pm$ 128d x $2^{-72d} = \pm$ 2.71050543121 e-20

***Ejemplo 2***: Valor típico:

-1100 0111b 1001001b

$=$ +0.1100 0111b x $2^{(1001001b-1000000b)}$

$=$ +0.1100 0111b x $2^{1101b}$

$=$ +0.1100 0111b x $2^{9d}$

$=$ +1100 0111.0b x $2^{1d}$

$=$ +199d x 2 $=$ +398



## *3.6.2 Suma/Resta en coma flotante*

# Suma/resta en coma flotante

El proceso de suma/resta en coma flotante involucra los siguientes pasos:

- ***Paso 1:*** se igualan los exponentes
  - Se comparan los exponentes para determinar el número mayor
  - Se desplaza a la derecha la mantisa del número menor tantas posiciones como resulte de la diferencia de exponentes (se pierde resolución en el número menor)
  - Si el número de desplazamientos es superior al número de bits de la mantisa se desprecia el operando menor (= 0)

- ***Paso 2:*** Se realiza la suma/resta de las mantisas con un restador binario convencional

- ***Paso 3:*** Si el resultado de la suma/resta de las mantisas es negativo se invierte para obtener la mantisa en positivo



# Ejemplos de suma/resta en coma flotante (I)

***Ejemplo 1***: suma de dos números positivos:

$0.3046875d \times 2^{45} + 0.34375 \times 2^{44}$
$= +0.01001110b \times 2^{45} + +0.01011000b \times 2^{44}$
$= +0.01001110b \times 2^{45} + +0.00101100b \times 2^{45}$
$= +0.01111010b \times 2^{45}$
$= +0.11110100b \times 2^{44}$
$= +0.953125d \times 2^{(44-64)} = +0.953125d \times 2^{-20} = +9.089708328\ e\text{-}7$

***Ejemplo 2***: número negativo más número positivo:

$-0.82421875d \times 2^{76} + +0.25390625 \times 2^{75}$
$= -0.11010011b \times 2^{76} + +0.01000001b \times 2^{75}$
$= -0.11010011b \times 2^{76} + +0.00100000b \times 2^{76}$
$= -0.10110011b \times 2^{76}$
$= -0.69921875d \times 2^{(76-64)} = -0.69921875d \times 2^{12} = -2.864\ e3$

El desplazamiento de la mantisa causa pérdida de resolución

# Ejemplos de suma/resta en coma flotante (II)

***Ejemplo 3***: número negativo más número positivo (insignificante):

$-0.5d \times 2^{89} + +0.5d \times 2^{68}$

$= -0.10000000b \times 2^{89} + +0.10000000b \times 2^{68}$

$= -0.10000000b \times 2^{89} + +0.00000000b \times 2^{89}$

$= -0.10000000b \times 2^{89}$

$= -0.5d \times 2^{89} = -0.5d \times 2^{(89 - 64)} = -1.6777216\ e7$

El operando positivo es insignificante comparado con el negativo

***Ejemplo 4***: número positivo más número negativo:

$+0.5 \times 2^{64} + -0.875 \times 2^{63}$

$= +0.10000000b \times 2^{64} + -0.11100000b \times 2^{63}$

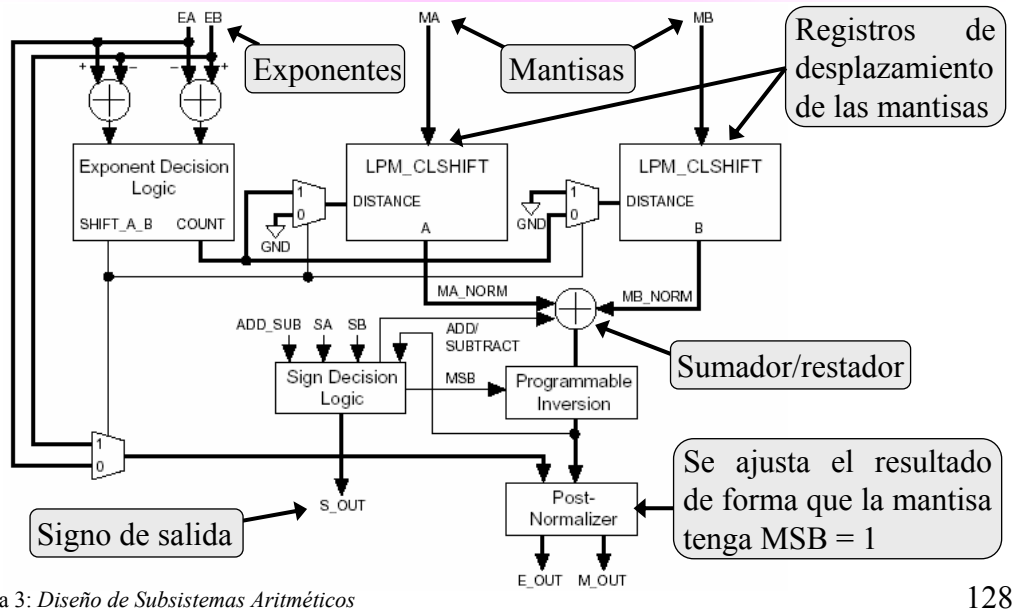
$= +0.10000000b \times 2^{64} + -0.01110000b \times 2^{64}$

$= +0.00010000b \times 2^{64}$

$= +0.10000000b \times 2^{61} = +0.5 \times 2^{(61 - 64)} = +0.5 \times 2^{-3} = 0.0625d$



# Sumador/Restador en coma flotante



# 3.6.3 Multiplicación en coma flotante



## Multiplicación en coma flotante

El proceso de multiplicación en coma flotante involucra los siguientes pasos:

- ***Paso 1:*** Se multiplican las mantisas con un multiplicador convencional

- ***Paso 2:*** Se suman los exponentes y se resta el offset del resultado:

  $(EA-2^{(n-1)}) + (EB-2^{(n-1)}) = EA+EB -2 \times 2^{(n-1)}$

  $(EA+EB -2 \times 2^{(n-1)}) - 2^{(n-1)} = EA+EB -2^{(n-1)}$

- ***Paso 3:*** El signo del resultado se obtiene con la operación XNOR entre el signo de número A (SA) y el signo del número B (SB)

- ***Paso 4:*** Si es necesario se ajusta el resultado para que la mantisa tenga MSB=1

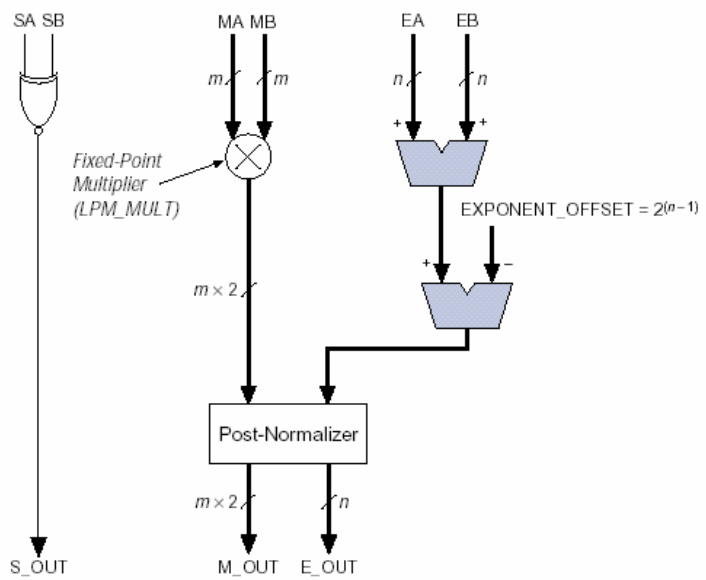
# Ejemplo de multiplicación en coma flotante

Multiplicación de 39936 y 13303808

| Decimal Equivalent (Exponent in Excess 0) | Binary (Exponent in Excess 64) |
|---|---|
| $(39 \times 2^{10d}) \times (203 \times 2^{6d})$ | $(00100111.0 \times 2^{74}) \times (11001011.0 \times 2^{70})$ |
| $(0.609375 \times 2^{16d}) \times (0.79296875 \times 2^{14d})$ | $(0.10011100 \times 2^{80}) \times (0.11001011 \times 2^{78})$ |
| $7917 \times 2^{16d}$ | $0.0001111011101101 \times 2^{80}$ |
| $63336 \times 2^{13d}$ | $0.1111011101101000 \times 2^{77}$ |
| 518,848,512 | — |



# Esquema de multiplicador en coma flotante



## Detección de errores en coma flotante

Errores habituales en operaciones en coma flotante:

- ***Overflow***: cuando el valor del exponente sobrepasa el número de bits disponibles.
  Por ejemplo, para 7 bits de exponente se tiene ***overflow*** cuando el exponente es mayor de 63.

- ***Underflow***: cuando el valor del exponente está por debajo del valor mínimo.
  Por ejemplo, para 7 bits de exponente se tiene ***underflow*** cuando el exponente es menor de -64.



# *3.7 Funciones especiales*

# 3.7.1 Multiplicación por una constante

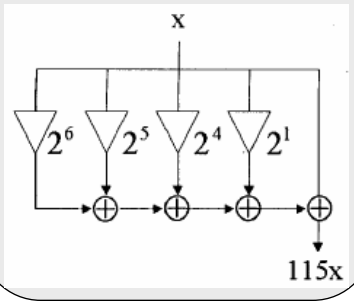


## Algoritmo Dempster-Macleod

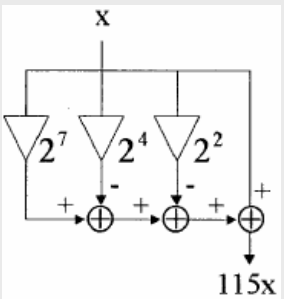
Permite diseñar un multiplicador mediante un número mínimo de sumadores/restadores

***Ejemplo***: multiplicación por 115

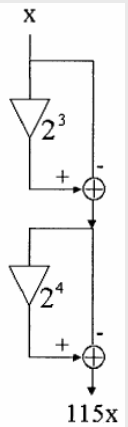
$115 = 2^6 + 2^5 + 2^4 + 2^1 + 2^0$




$115 = 2^7 - 2^4 - 2^2 + 2^0$



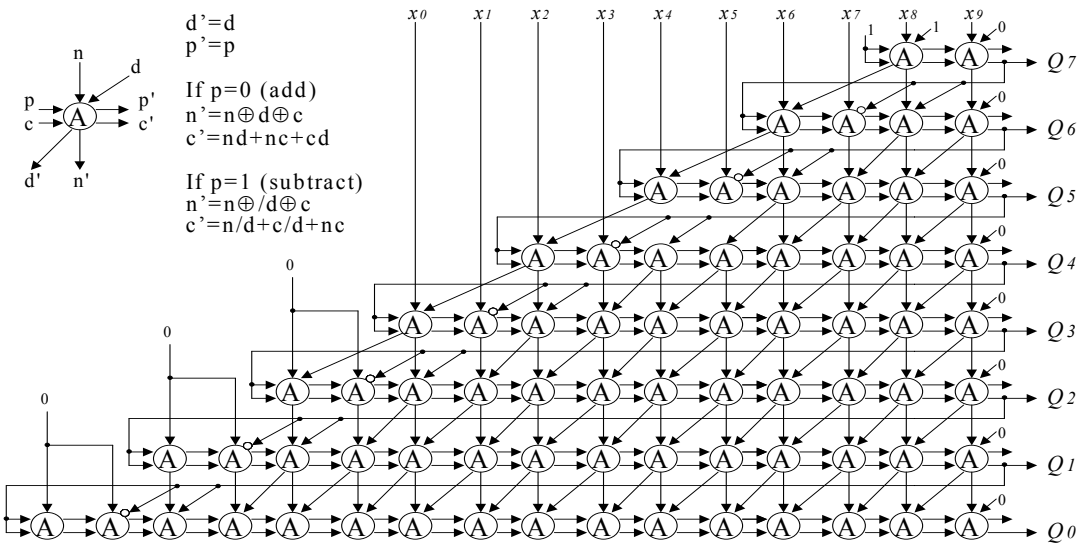

$115 = (7)x(15) = (2^3 - 2^0)x(2^4 - 2^0)$



# 3.7.2 Raíz cuadrada



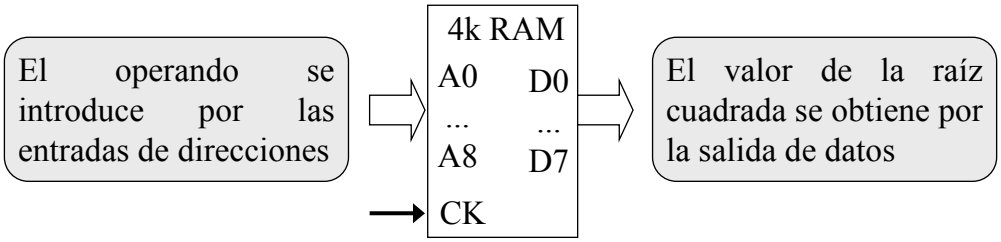
# Extractor de raíz cuadrada 8 bits con lógica combinacional



# Extractor de raíz cuadrada 8 bits basado en LUT

- Se almacena en una LUT implementada mediante memoria RAM los valores de la raíz cuadrada
- La velocidad de calculo se incrementa considerablemente ya que equivale al tiempo de acceso de la memoria RAM
- La memoria RAM de las FPGA's de Xilinx permiten implementar con LUT este tipo de funciones aritméticas

*Bloque memoria RAM*






# *3.7.3 Cuadrado de un número: $X^2$*

# Algoritmo de $X^2$ Serie-Serie

- Mediante un cálculo iterativo se puede extraer el valor de $X^2$
- El algoritmo es un caso particular del multiplicador Serie-Serie, pero utilizando un hardware simplificado
- El resultado correcto se obtiene después de *2N* iteraciones

Algoritmo de cálculo de $X^2$

$$\overline{Q}_i = \left[\frac{1}{2}\cdot\left(\overline{Q}_{i-1} + 2\cdot x_i \cdot X_{i-1} + 2^i \cdot x_i\right)\right] \quad para \quad i < N$$

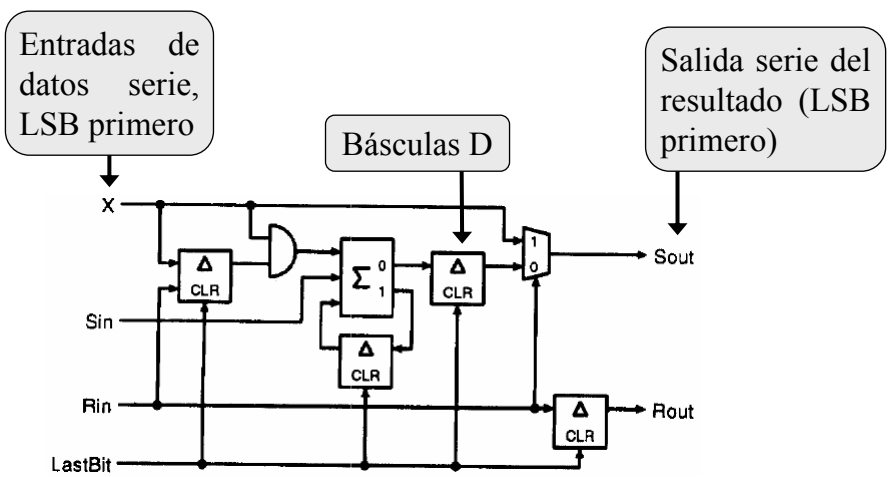
$$\overline{Q}_i = \left[\frac{1}{2}\cdot\left(\overline{Q}_{i-1} + 2\cdot x_{N-1} \cdot X_{N-2}\right)\right] \quad para \quad i \geq N$$

$x_i : entrada \quad actual$

$X_i : entrada \quad anterior$

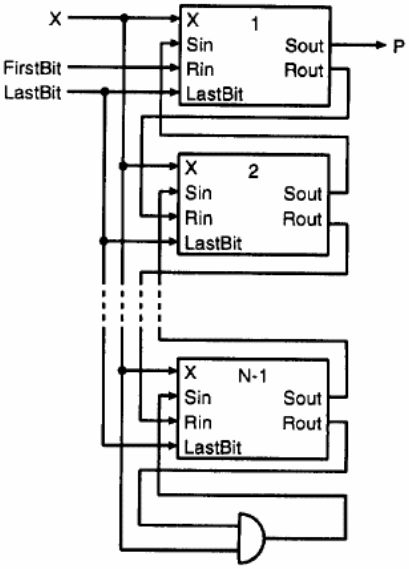


# Célula básica del algoritmo $X^2$



# Multiplicador $X^2$

- Se necesitan *N-1* células básicas para obtener el cuadrado de un número de *N* bits
- El resultado se obtiene después de *2N* ciclos de reloj
- Durante los *N* primeros ciclos se introduce el operando (*LSB* primero) por la entrada serie
- Durante el resto de ciclos se introducen ceros (*nº positivos*) o se extiende el signo (*nº c.a.2*)
- Entradas de control del elevador al cuadrado:
  - ***FirsBit:*** Se pone a nivel alto durante el primer ciclo de señal de reloj
  - ***LastBit:*** Se pone a 1 durante el ciclo que precede a la siguiente multiplicación



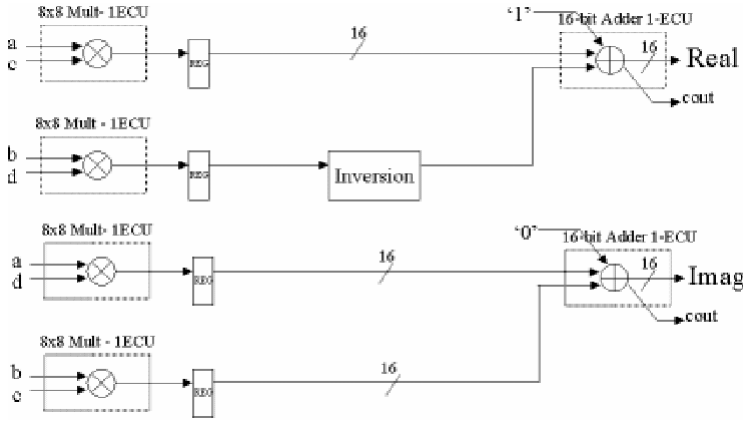



# *3.7.4 Multiplicación de complejos*

# Multiplicación de complejos

La multiplicación de números complejos viene dada por:

- (a + jb)(c + jd) = (ac-bd) + j(ad + bc)
- Parte Real = ac-bd
- Parte Imaginaria = ad+bc






# *3.8 Bloques aritméticos en FPGA's*

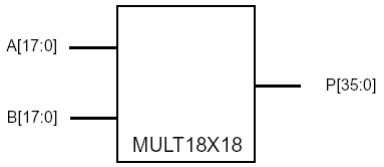
# 3.8.1 Multiplicadores de la familia Spartan-3



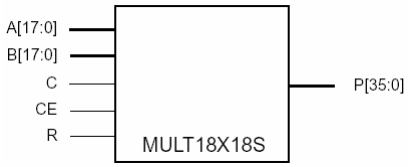
## Características generales

- La familia Spartan-3 dispone de multiplicadores 18x18 bits (c.a.2) integrados
- Prestaciones:
  - Realizan el producto con y sin signo (17x17 bits unsigned)
  - Se pueden conectar en cascada o con CLB's para realizar funciones complejas
  - Pueden realizar funciones adicionales: desplazamiento, generación de valor absoluto, generación del complemento a 2 de un número, etc...

Multiplicador Combinacional

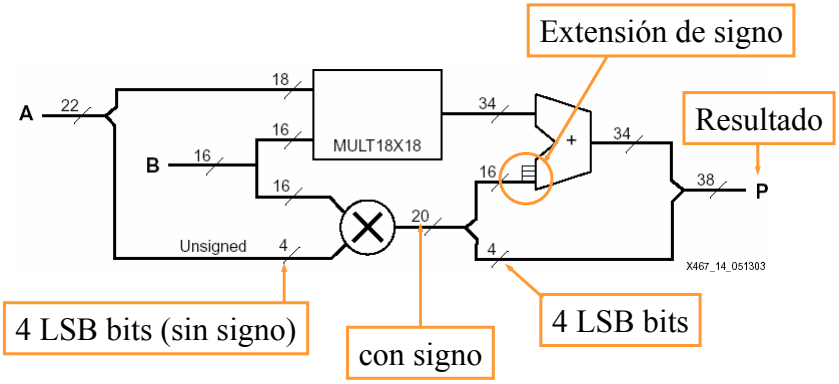



Multiplicador con registro



# Multiplicación expandida

- Es posible multiplicar números mayores de 18 bits descomponiendo el producto en procesos más simples
- Ejemplo: producto 22 x 16 bits con signo





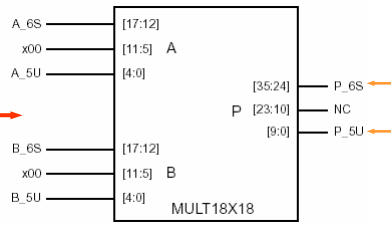

# Dos productos en un Multiplicador

- Es posible realizar dos multiplicaciones de dos números pequeños con un solo multiplicador siempre y cuando no se solapen los resultados

Tamaños permitidos para los operandos

| X * X | | Y * Y | |
|---|---|---|---|
| Signed Size | Unsigned Size | Signed Size | Unsigned Size |
| 7 X 7 | 6 X 6 | - | 4 X 4 |
| 6 X 6 | 5 X 5 | - | 5 X 5 |
| 5 X 5 | 4 X 4 | 3 X 3 | 6 X 6 |
| 4 X 4 | 3 X 3 | 3 X 3 | 7 X 7 |
| 3 X 3 | 2 X 2 | 4 X 4 | 8 X 8 |

***Ejemplo***: producto de 6 bits con signo (A_6S x B_6S) y 5 bits sin signo (A_5U x B_5U)




Resultado A_6S x B_6S

Resultado A_5U x B_5U

# Aplicaciones alternativas: desplazamiento

- Un multiplicador se puede utilizar para desplazar un operando de 0 a 16 posiciones multiplicando por $2^n$

- Dos tipos de desplazamiento:

  - Lógico → el bit de mayor peso del dato a desplazar es cero (positivo) ⇒ los bits de mayor peso del resultado son cero

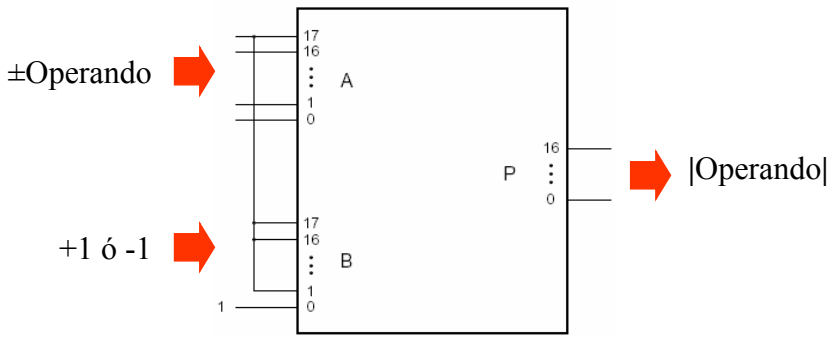
  - Aritmético → se extiende el bit de signo del dato a desplazar ⇒ los bits de mayor peso del resultado tienen el signo del dato desplazado



# Aplicaciones alternativas: valor absoluto

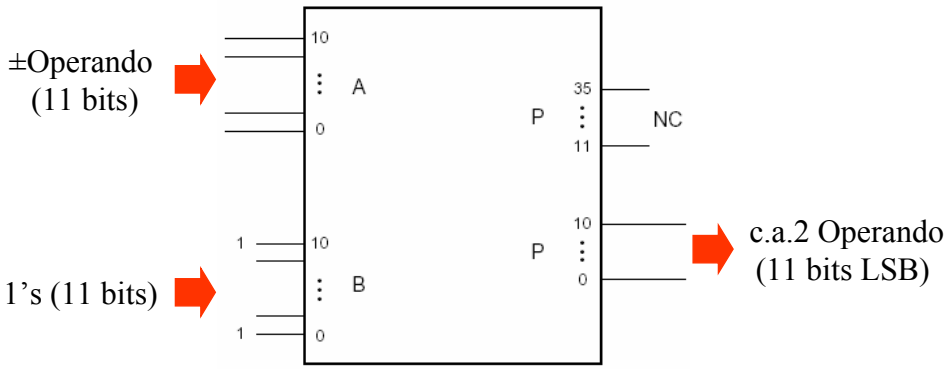
- Para calcular el valor absoluto de un número se multiplica por 1 si es positivo y por –1 si es negativo

- En c.a.2:
  - 1 positivo ⇒ 00 0000 0000 0000 0001
  - 1 negativo ⇒ 11 1111 1111 1111 1111



# Aplicaciones alternativas: generación c.a.2

- Para generar el c.a.2 de un operando se multiplica por un número de 1's de la misma longitud que el operando






# Aplicaciones alternativas: producto de complejos, producto en coma flotante

- Producto de dos números complejos: (a+ib)·(c+id) = ac-bd +i(ad+cb)
- Se puede solucionar con tres productos reales: ac, bd y (a+b)(c+d)
  - Parte real del resultado: ac-bd
  - Parte imaginaria del resultado: (a+b)(c+d)-ac-bd = ad+cb
- Se puede implementar un multiplicador en coma flotante de 32 bits con 4 multiplicadores y algunos CLB

# *3.8.2 XtremeDSP de la familia Virtex-4: DSP48*

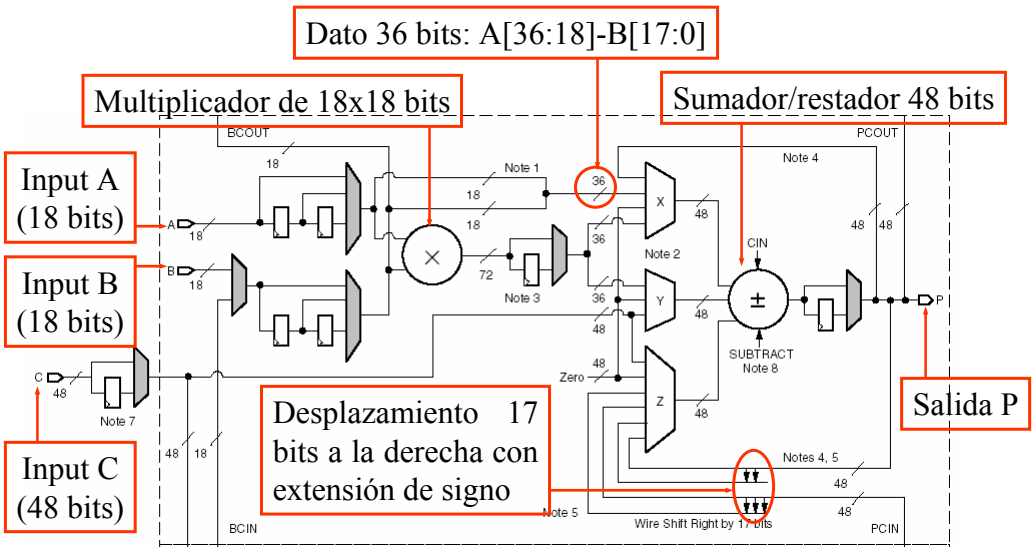


## Características generales

- La familia *Virtex-4* dispone de 32 a 192 elementos de procesado digital de señal denominados *XtremeDSP*

- Cada *XtremeDSP* incorpora 2 DSP48 *slices* que pueden realizar múltiples funciones:
  - Multiplicación
  - Multiplicación y acumulación (MACC)
  - Multiplicación y suma
  - Sumador de tres entradas
  - Comparador

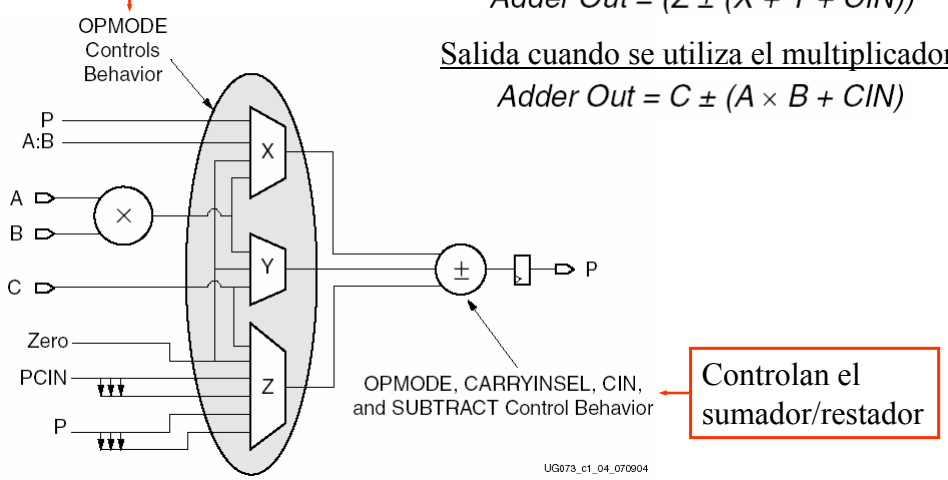
# Esquema general






# Modelo simplificado

Selecciona las entradas de los multiplexores X, Y y Z




Controlan el sumador/restador

Salida cuando no se utiliza el multiplicador:

$$Adder\ Out = (Z \pm (X + Y + CIN))$$

Salida cuando se utiliza el multiplicador:

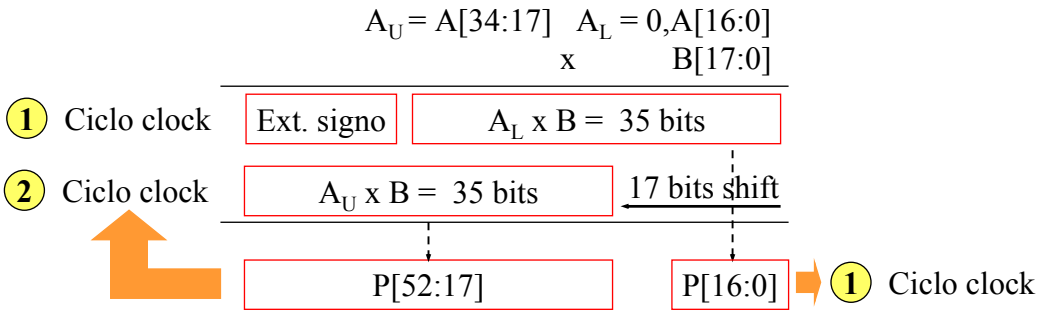
$$Adder\ Out = C \pm (A \times B + CIN)$$

# Aplicaciones: producto 35x18 bits c.a.2 (I)

Un *slice* puede realizar múltiples operaciones parciales durante sucesivos periodos de reloj que combinadas den lugar a operaciones complejas

Ejemplo: Producto 35x18 bits = A[34:0] x B[17:0] en dos ciclos de reloj

A se descompone en dos números: uno negativo $A_U$ = A[34:17] y otro positivo $A_L$ = 0,A[16:0]



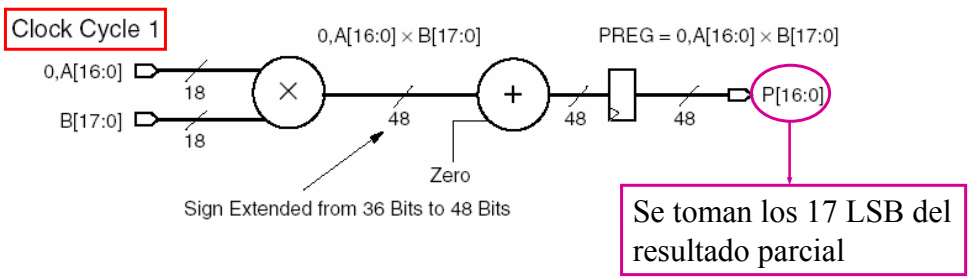



# Aplicaciones: producto 35x18 bits c.a.2 (II)

1 Ciclo clock

En el primer ciclo de reloj se realiza el producto parcial P = (0,A[16:0]) x B[17:0]

OPMODE: Multiplicación

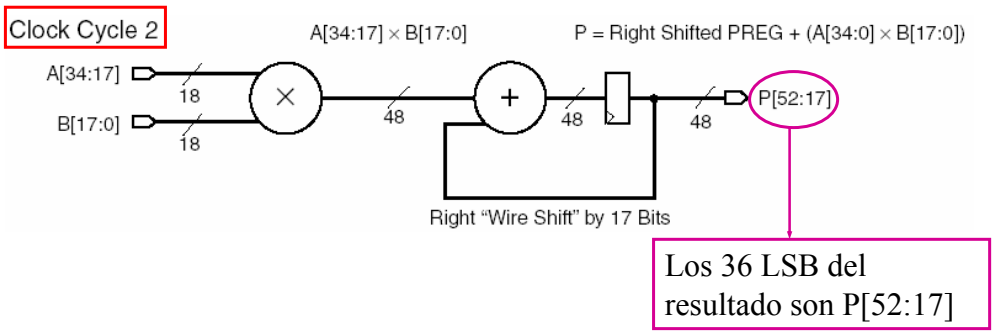


# Aplicaciones: producto 35x18 bits c.a.2 (III)

2 Ciclo clock

En el segundo ciclo de reloj se realiza el producto parcial P = A[34:17] x B[17:0]

OPMODE: 17-bits Shift P y Multiplicación con suma



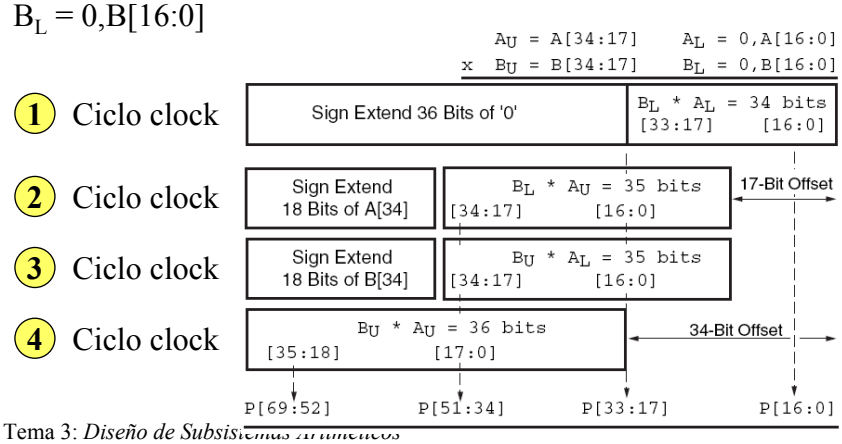

Los 36 LSB del resultado son P[52:17]



# Aplicaciones: producto 35x35 bits c.a.2 (I)

Ejemplo: 35x35 bits ⇒ A[34:0] x B[34:0]=P[69:0] en cuatro ciclos de reloj

- A se descompone en dos números: uno negativo $A_U$ = A[34:17] y otro positivo $A_L$ = 0,A[16:0]
- B se descompone en dos números: uno negativo $B_U$ = B[34:17] y otro positivo $B_L$ = 0,B[16:0]

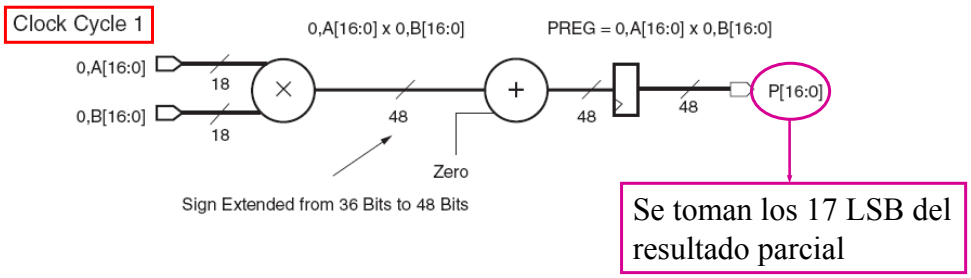


# Aplicaciones: producto 35x35 bits c.a.2 (II)

1 Ciclo clock

Se realiza el producto parcial P = 0,A[16:0] x 0,B[16:0]

OPMODE: Multiplicación



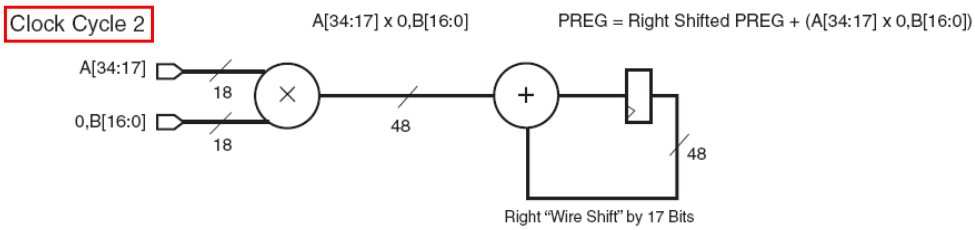



# Aplicaciones: producto 35x35 bits c.a.2 (III)

2 Ciclo clock

Se realiza el producto parcial P = A[34:17] x 0,B[16:0]

OPMODE: 17-bits Shift P y Multiplicación con suma

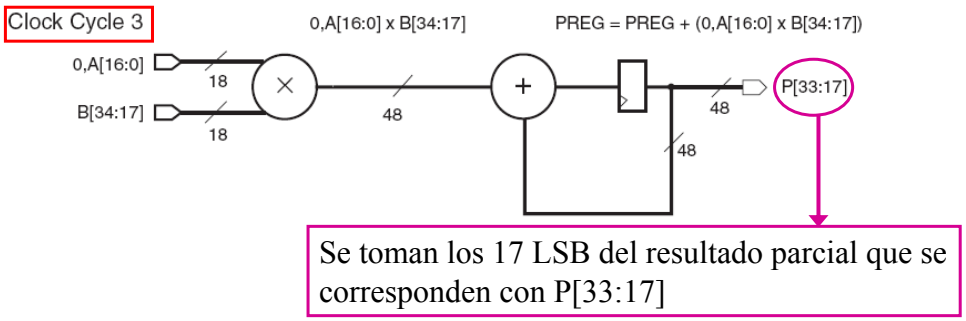


# Aplicaciones: producto 35x35 bits c.a.2 (IV)

3 Ciclo clock

Se realiza el producto parcial P = 0,A[16:0] x B[34:17]

OPMODE: Multiplicación con suma




Se toman los 17 LSB del resultado parcial que se corresponden con P[33:17]

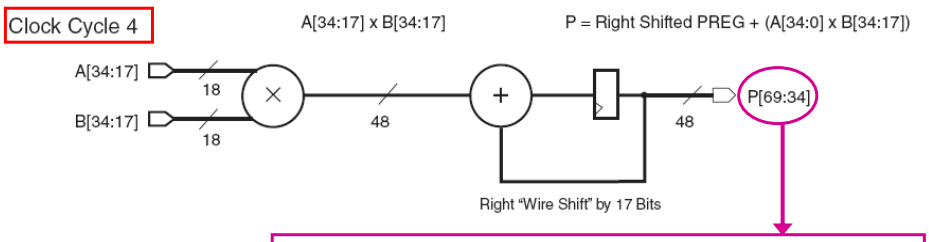


# Aplicaciones: producto 35x35 bits c.a.2 (V)

4 Ciclo clock

Se realiza el producto parcial P = A[34:17] x B[34:17]

OPMODE: 17-bits Shift P y Multiplicación con suma

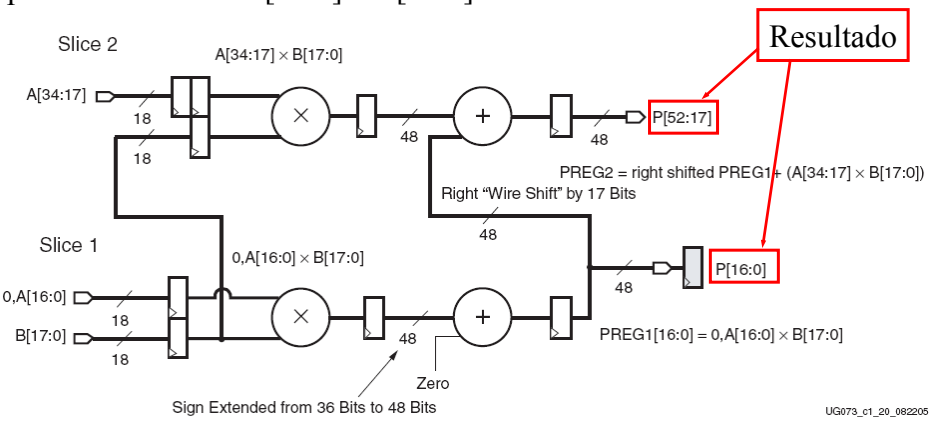



Se toman los 36 LSB del resultado parcial que se corresponden con P[69:34]

# Aplicaciones: producto 35x18 bits c.a.2 full pipelined

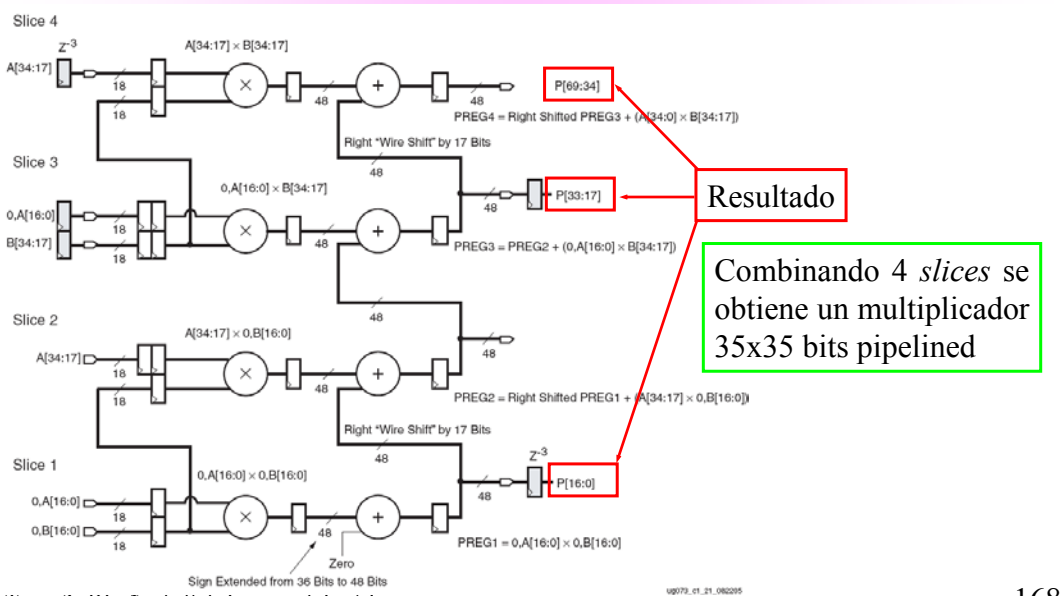
- Muchas aplicaciones DSP requieren alta velocidad de proceso
- Combinando varios DSP48 *slice* se puede realizar operaciones pipelined
- Pipelined ⇒ con cada periodo de reloj se obtiene un nuevo resultado

Ejemplo: 35x18 bits = A[34:0] x B[17:0]






# Aplicaciones: producto 35x35 bits c.a.2 full pipelined



## Otras funciones matemáticas básicas

Con un DSP48 *slice* se pueden realizar muchas funciones matemáticas básicas como:

- Sumar/restar
- Acumulación
- MAC: multiplicación mas suma
- Multiplexado
- Registro de desplazamiento
- Contador
- Multiplicador/divisor
- Raiz cuadrada



# *3.8.3 XtremeDSP de la familia Virtex-5: DSP48E*

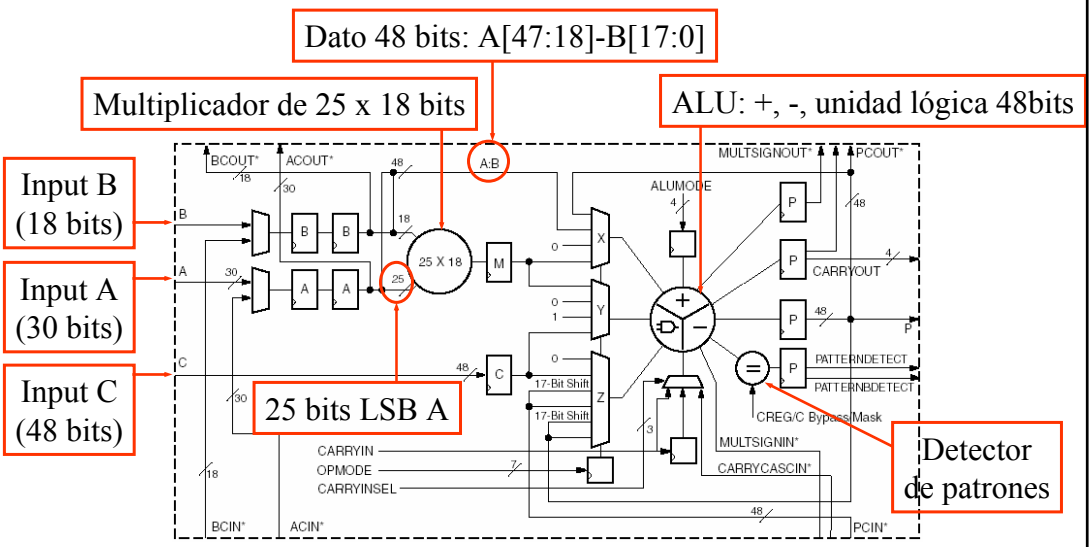
# Características generales

- La familia *Virtex-5* dispone de 32 a 192 elementos DSP denominados DSP48E que pueden realizar múltiples funciones:
  - Multiplicación
  - Multiplicación y acumulación (MACC)
  - Sumador de tres entradas
  - Comparador
  - Funciones lógicas
  - Detección de patrones

- Colocando en cascada múltiples DSP48E se pueden generar funciones matemáticas complejas, filtros digitales, etc sin tener que utilizar recursos generales de la FPGA



# Esquema general



# Bibliografía (I)


- Ernest Jamro, Kazimierz Wiatr, "***Constant Coefficient Convolution Implemented in FPGAs***", Proceedings of the Euromicro Symposium on Digital System Design (DSD'02)
- Keshab K. Parhi, "***A Systematic Approach for Design of Digit-Serial Signal Processing Architectures***", IEEE Transactions on Circuits and Systems, Vol. 38, no. 4, pp. 358-375, April 1991.
- Paolo Ienne and Marc A. Viredaz, "***Bit-Serial Multipliers and Squarers***", IEEE Transactions on Computers, Vol. 43, no. 12, pp. 1445-1450, Dec. 94.
- Richard Hartley and Peter Corbett, "***Digit-Serial Processing Techniques***", IEEE Transactions on Circuits and Systems, Vol. 37, no. 6, pp. 707-719, June 1990.
- Luigi Dadda, "***On Serial-Input Multipliers for Two's Complement Numbers***", IEEE Transactions on Computers, Vol. 38, no. 9, pp. 1341-1345, Sep. 89.
- Fuminori Kobayashi, Taro Tsujino, and Hirokazu Saitoh, "***Efficient FPGA Implementation of Multiplier-Adder, Quotient-Remainder Approach***", IEEE, pp. 227-230, 1998.




# Bibliografía (II)


- Javier Valls and Eduardo Boemo, "***Efficient FPGA-Implementation of Two's Complement Digit-Serial/Parallel Multipliers***", IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 50, no. 6, pp. 317-322, June 2003.
- Jer Min Jou, Shiann Rong Kuang, and Ren Der Chen, "***Design of Low-Error Fixed-Width Multipliers for DSP Applications***", IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Vol. 50, no. 6, pp. 836-842, June 1999.
- R. H. Turner, T. Courtney and R. Woods, "***Implementation of Fixed DSP Functions Using the Reduced Coefficient Multiplier***", IEEE, pp. 881-884, 2001.
- Ray Andraka, "A Survey of CORDIC Algorithms for FPGA Based Computers"
- J. C. Majithia, "***A cellular array for the nonrestoring extraction of square roots***", IEEE Transactions on Computers, Vol. C-20, pp. 1617-1618, Dec. 71.
- M. A. Ashour, H. I. Saleh, ***"An FPGA implementation guide for some different types of serial-parallel multiplier structures***", Microelectronics Journal 31, pp. 161-168, 2000.

# Bibliografía (III)


- S. Sunder, F. El-Guibaly, A. Antoniou, ***"Two's-complement fast serial-parallel multiplier***", IEE Proc.-Circuits Devices Syst., Vol. 142, no 1, pp. 41-44, Feb. 1995.
- K. Adaos, G. Alexiou, N. Kanopoulos, ***"Development of reusable serial FIR filters with reprogrammable coefficients designed for serial dataflow architectures",*** IEEE 2000.
- Leland B. Jackson, James F. Kaiser, Henry S. McDonald, ***"An Approach to the Implementation of Digital Filters",*** IEEE Trans. on Audio and Electroacoustics, Vol. AU-16, no 3, pp. 413-421, Sep. 1968.
- P. Larsson-Edefors, W. P. Marnane, ***"Most-significant-bit-first serial/parallel multipliers***", IEE Proc.-Circuits Devices Syst., Vol. 145, no 4, pp. 278-284, Aug. 1998.
- Kiamal Z. Pekmestzi, ***"Multiplexer-Based Array Multipliers",*** IEEE Trans. On Computers, Vol. 48, no 1, pp. 15-23, Jan. 1999.




# Bibliografía (IV)

www.actel.com


- ***Application Note***: Synchronous Dividers in Actel FPGAs
- ***Application Note***: Implementing Multipliers with Actel FPGAs
- ***Application Note***: Designing FIR Filters with Actel FPGAs


www.altera.com


- Altera Corporation, "***Using Soft Multipliers with Stratix and Stratix GX Devices***", Stratix Device Handbook, Volume 2, Cap 9, 2003.
- ***AN053***: Implementing Multipliers in Flex 10K Devices
- ***PIB21***: Implementing Logic with the Embedder Array in Flex 10K Devices
- ***FS02***: Floating-Point Adder-Subtractor
- ***FS04***: Floating-Point Multiplier

# Bibliografía (V)

www.xilinx.com

- ***XAPP022:*** Adders, Subtracters and Accumulators in XC3000
- ***DSPX5DEV***: Gregory R. Goslin, "Using Xilinx FPGA's to Design Custom Signal Processing Devices"
- ***XAPP054***: Constant Coefficient Multipliers for the XC4000E
- ***XAPP018:*** Estimating the Performance of XC4000E Adders and Counters
- ***DS099:*** Spartan-3 FPGA Family: Complete Data Sheet
- ***XAPP467:*** Using Embedded Multipliers in Spartan-3 FPGAs
- ***GS093:*** XtremeDSP for Virtex-4 FPGAs User Guide



# Bibliografía (VI)

www.atmel.com

- ***Application Note:*** FPGA-based FIR Filter Using Bir-Serial Digital Signal Processing
- ***Application Note:*** Implementing Bit-Serial Digital Filters in AT6000 FPGAs

www.latticesemiconductor.com

- ***AN8014:*** Adder and Subtractor Macros Using Lattice Design Tools

# Bibliografía (VII)


- ***APPNOTE53***: Wai-Leng Lim, “QuickDSP Complex Multiplier”