Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
- División
- Generadores de función
- Ejercicios resueltos

Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
  - Operación y técnicas básicas
  - Multiplicadores de altas prestaciones
  - Multiplicadores en FPGA
- División
- Generadores de función
- Ejercicios resueltos

Multiplicadores; multiplicación

$$P = A \times B$$

```
          1 0 1 0    A: multiplicando
        × 0 1 1 1    B: multiplicador
        ---------
          1 0 1 0    Primer producto parcial
      + 1 0 1 0      Segundo producto parcial
        ---------
        1 1 1 1 0
    + 0 0 0 0
      -----------
      0 1 1 1 1 0
    + 1 0 1 0        Tercer producto parcial
      -----------
      1 1 0 1 1 0
  + 0 0 1 0
    -------------
    0 1 0 0 1 1 0
  + 0 1 0
    -------------
    1 0 0 0 1 1 0    P: producto
```

acarreo

Multiplicación combinacional de números positivos

p_3:

b_1b_0 \ a_1a_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	0
10	0	0	0	0

p_2:

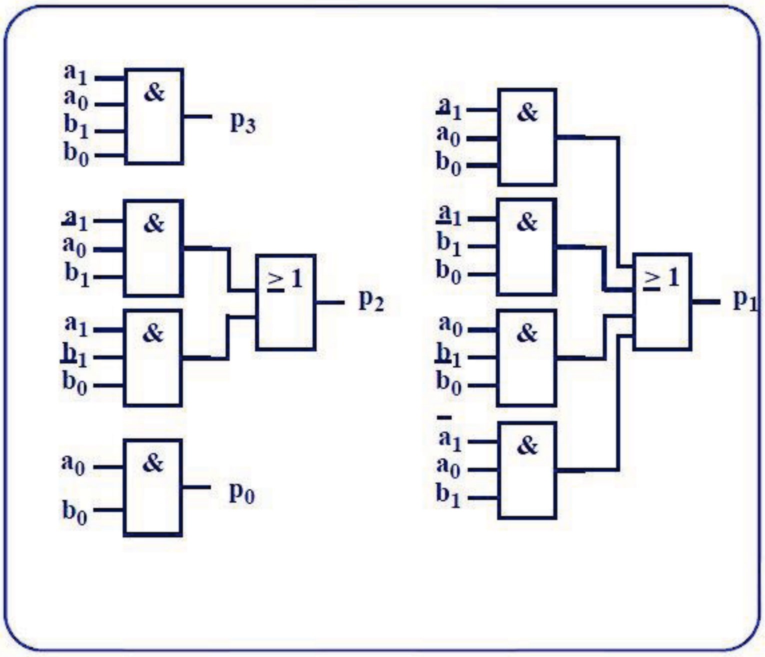
b_1b_0 \ a_1a_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	1
10	0	0	1	1

p_1:

b_1b_0 \ a_1a_0	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	0	1
10	0	1	1	0

p_0:

b_1b_0 \ a_1a_0	00	01	11	10
00	0	0	0	0
01	0	1	1	0
11	0	1	1	0
10	0	0	0	0



+ coste
+ complejidad en el diseño
– modularidad

+ velocidad

M. Valencia: "Curso de Técnicas Avanzadas de Diseño de Sistemas Digitales", Universidad de Sevilla, 2005

Multiplicación secuencial

- Sea el multiplicador $X = x_{n-1}x_{n-2}\cdots x_1x_0$ y el multiplicando $A = a_{n-1}a_{n-2}\cdots a_1a_0$, con x_{n-1} y a_{n-1} los bits de signo. Llamaremos U al producto
- En el paso j, se examina x_j, y se añade al producto parcial previamente acumulado, $P^{(j)}$, con $\left(P^{(0)} = 0\right)$: $\quad P^{(j+1)} = \left(P^{(j)} + x_j \cdot A\right) \cdot 2^{-1}; j = 0, 1, 2, \cdots, n-2.$
- Al multiplicar por 2^{-1} desplazamos una posición a la derecha ya que la alineación es necesaria dado que el peso de x_{j+1} es doble que el de x_j.
- Por repeticiones sucesivas →

$$P^{(1)} = \left(P^{(0)} + x_0A\right)2^{-1} = x_0A\,2^{-1}$$
$$P^{(2)} = \left(P^{(1)} + x_1A\right)2^{-1} = (x_0A\,2^{-1} + x_1A)2^{-1} = (x_02^{-2} + x_12^{-1})A$$
$$P^{(3)} = \left(P^{(2)} + x_2A\right)2^{-1} = (x_0A\,2^{-2} + x_12^{-1}A + x_2A)2^{-1} = (x_02^{-3} + x_12^{-2} + x_22^{-1})A$$
$$P^{(4)} = \left(P^{(3)} + x_3A\right)2^{-1} = \cdots = (x_02^{-4} + x_12^{-3} + x_22^{-2} + x_32^{-1})A$$
$$\ldots$$
$$P^{(n-1)} = \left(P^{(n-2)} + x_{n-2}A\right)2^{-1} = \cdots = \left(x_02^{-(n-1)} + x_12^{-(n-2)} + \cdots + x_{n-3}2^{-2} + x_{n-2}2^{-1}\right)A$$
$$P^{(n-1)} = A \cdot \sum_{j=0}^{n-2} x_j2^{-(n-1-j)} = 2^{-(n-1)} \cdot A \cdot \sum_{j=0}^{n-2} x_j2^j$$

Multiplicación secuencial: ejemplo

- Consideremos $X = 01011_{(2}$ y $A = 01101_{(2}$. Ambos números son positivos.

A				0 1 1 0 1		13
X			×	0 1 0 1 1		11
$P^{(0)} = 0$				0 0 0 0 0		
$x_0 = 1$	⇨ Add A		+	0 1 1 0 1		
				0 1 1 0 1		
Shift		$P^{(1)}$		0 1 1 0	1	
$x_1 = 1$	⇨ Add A		+	0 1 1 0 1		
				1 0 0 1 1	1	
Shift		$P^{(2)}$		1 0 0 1	1 1	
$x_2 = 0$	⇨ Shift	$P^{(3)}$		1 0 0	1 1 1	
$x_3 = 1$	⇨ Add A		+	0 1 1 0 1		
				1 0 0 0 1	1 1 1	
Shift		$P^{(4)}$		1 0 0 0	1 1 1 1	143

$$P^{(j+1)} = \left(P^{(j)} + x_j \cdot A\right) \cdot 2^{-1}$$

$$U = 2^{n-1}P^{(n-1)} = 2^4P^{(4)} = X \cdot A$$

Multiplicación secuencial

- Hemos obtenido:

$$P^{(n-1)} = A \cdot \sum_{j=0}^{n-2} x_j 2^{-(n-1-j)} = 2^{-(n-1)} \cdot A \cdot \sum_{j=0}^{n-2} x_j 2^j$$

- Si ambos números son positivos, $x_{n-1} = a_{n-1} = 0$, → $U = 2^{n-1} P^{(n-1)} = A \sum_{j=0}^{n-2} x_j 2^j = A \cdot X$

- Si el multiplicador es negativo, $x_{n-1} = 1$, → $X = -x_{n-1} 2^{n-1} + \tilde{X}$, con $\tilde{X} = \sum_{j=0}^{n-2} x_j 2^j$

  Ignorando ahora el bit de signo, el resultado final sería $\tilde{X} \cdot A$:

  $\tilde{X} \cdot A = (X + x_{n-1} 2^{n-1}) \cdot A = X \cdot A + A\, x_{n-1} 2^{n-1}$

  El resultado que buscamos es $X \cdot A$, por lo que deberíamos corregir $\tilde{X} \cdot A$ restándole $A\, x_{n-1} 2^{n-1}$ (es decir, sumando su complemento a 2)

  Dicho de otro modo, si $x_{n-1} = 1$, debemos restar el multiplicando A en la posición correspondiente.

Multiplicación secuencial

- Si los números se expresan con notación signo-magnitud, entonces se multiplican las magnitudes y se genera el signo en forma separada.

- Si los números se describen en notación de complemento, hay que distinguir entre
  → multiplicación con un multiplicando, A, negativo y
  → multiplicación con un multiplicador, X, negativo.

- Si sólo el multiplicando A es negativo, no hay que cambiar nada en el algoritmo anterior:

A		1 0 1 1		–5
X	×	0 0 1 1		3
$P^{(0)} = 0$		0 0 0 0		
$x_0 = 1$ ⇨ Add A	+	1 0 1 1		
		1 0 1 1		
Shift $P^{(1)}$		1 1 0 1	1	
$x_1 = 1$ ⇨ Add A	+	1 0 1 1		
		1 0 0 0	1	
Shift $P^{(2)}$		1 1 0 0	0 1	
$x_2 = 0$ ⇨ Shift $P^{(3)}$		1 1 1 0	0 0 1	–15

		1	0	1	1
	×	0	0	1	1
	1	1	0	1	1
	1	0	1	1	
1	1	0	0	0	1

Multiplicación secuencial

- Si el multiplicador X es negativo, el algoritmo cambia ya que el bit de signo no puede tratarse de la misma forma que los otros bits.

  → si $x_{n-1} = 1$, se resta el multiplicando

A			1 0 1 1		–5
X		×	1 1 0 1		–3
$P^{(0)} = 0$			0 0 0 0		
$x_0 = 1$	⇨ Add A	+	1 0 1 1		
			1 0 1 1		
Shift	$P^{(1)}$		1 1 0 1	1	
$x_1 = 0$	⇨ Shift	$P^{(2)}$	1 1 1 0	1 1	
$x_2 = 1$	⇨ Add A	+	1 0 1 1		
			1 0 0 1	1 1	
Shift	$P^{(3)}$		1 1 0 0	1 1 1	
$x_3 = 1$	⇨ Correct	+	0 1 0 1		5
			0 0 0 1	1 1 1	+15

			1	0	1	1
		×	1	1	0	1
	1	1	1	0	1	1
	1	0	1	1		
1	1	0	0	1	1	1
0	1	0	1			
0	0	0	1	1	1	1

$$U = 2^{n-1} P^{(n-1)} = 2^3 P^{(3)}$$

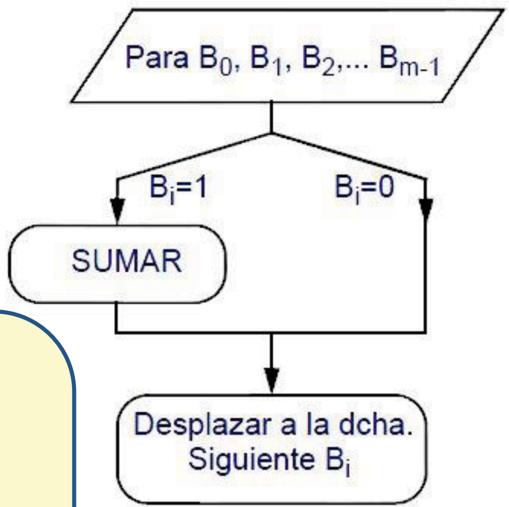
⇨ En el paso de corrección, restar el multiplicando ≡ sumar su complemento a 2.

Multiplicación secuencial de números positivos

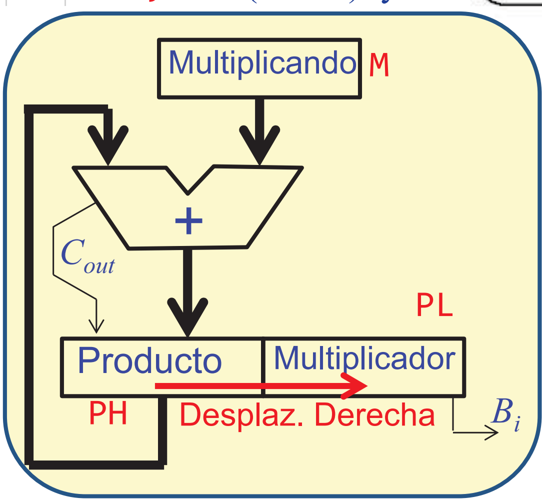
				1	1	0	1
			×	1	0	1	1
				1	1	0	1
			1	1	0	1	
		0	0	0	0		
	1	1	0	1			
1	0	0	0	1	1	1	1

→ A (Multiplicando)

→ B (Multiplicador)



Tres registros:
M, PH (inic. 0) y PL

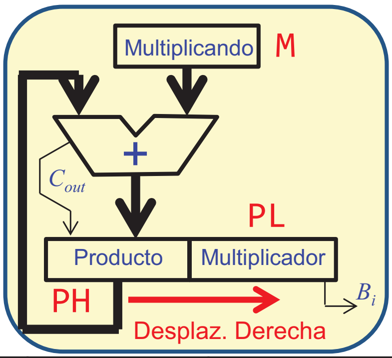


– coste

– velocidad

M. Valencia: "Curso de Técnicas Avanzadas de Diseño de Sistemas Digitales", Universidad de Sevilla, 2005

Multiplicación secuencial de números positivos



A (Multiplicando) →

B (Multiplicador) →

				1	1	0	1	
			×	1	0	1	1	PL
				1	1	0	1	
			1	1	0	1		
		0	0	0	0			
	1	1	0	1				
1	0	0	0	1	1	1	1	

c_{out}	PH				PL				T	
	0	0	0	0	1	0	1	1	0	PL_0 es 1, sumo 1101 con 0000
0	1	1	0	1	1	0	1	1		Resultado de la suma
	0	1	1	0	1	1	0	1	1	Desplazamiento; PL_0 es 1: sumo
	1	1	0	1						sumo 1101 con 0110 = 10011
1	0	0	1	1	1	1	0	1		Resultado de la suma
	1	0	0	1	1	1	1	0	2	Desplazamiento; PL_0 es 0: desplazo
		1	0	0	1	1	1	1	3	Desplazamiento; PL_0 es 1: sumo
	1	1	0	1						sumo 1101 con 0100 = 10001
1	0	0	0	1	1	1	1	1		Resultado de la suma
	1	0	0	0	1	1	1	1		Desplazamiento y resultado

Tema 6. Multiplicación, división y generación de funciones

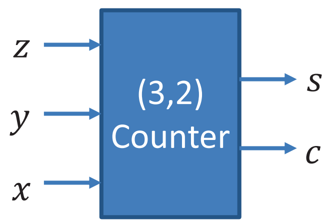
- **Multiplicación**
  - Operación y técnicas básicas
  - **Multiplicadores de altas prestaciones**
    - usando sumadores multioperando (*carry-save addition*)
    - reduciendo el número de productos parciales
    - usando multiplicadores más pequeños
    - multiplicación paralela
  - Multiplicadores en FPGA
- División
- Generadores de función
- Ejercicios resueltos

Sumadores con acarreo almacenado (CSA, Carry-Save Adder)

- En el caso de la multiplicación, se deben sumar simultáneamente tres o más operandos. Si usamos sumadores de dos operandos, se producen retrasos indeseables.
  - La operación de propagación de acarreos debe repetirse varias veces: para k operandos, → $(k-1)$ propagaciones
- Existen técnicas para la suma multioperando que intentan minimizar esta penalización.
  - ➔ La más conocida es la denominada suma con acarreo almacenado, **CSA** (*carry-save addition*).
    - El acarreo se propaga sólo en el último paso.
    - Los otros pasos generan sólo sumas parciales y secuencias de acarreo.
  - ➔ El **CSA básico** acepta **TRES** operandos de n-bits y genera **DOS** resultados de n-bits:
    - una suma parcial de n bits y
    - un acarreo de n bits.
  - El **segundo CSA** acepta los **DOS** resultados anteriores y **UN** operando de entrada y genera **DOS** nuevos resultados parciales de suma y acarreo.
  - El CSA reduce el número de operandos a sumar de **TRES** a **DOS**.

Sumadores con acarreo almacenado (CSA, Carry-Save Adder)

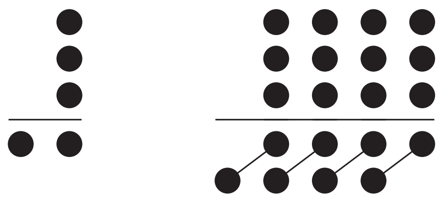
- La implementación más simple es mediante un sumador completo (**FA**, *full adder*), con tres entradas.
  - Las salidas no son más que la representación en binario del número de 1's en las entradas.
  - Al **FA** se le denomina contador (3,2).
  - Un **CSA** de n-bits se construye con n contadores (3,2) en paralelo, sin enlaces de acarreo.



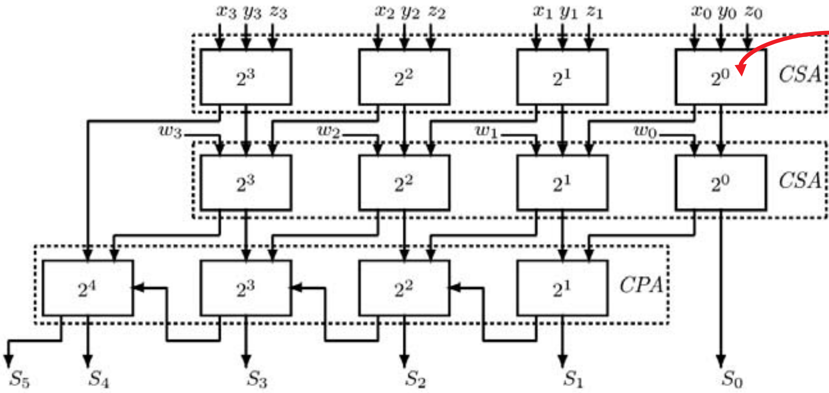
$$c = \frac{(x+y+z)-s}{2}$$

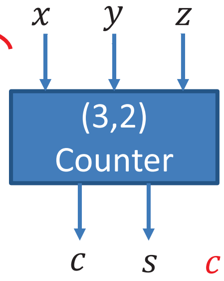
$$s = (x+y+z) \bmod 2$$

1-bit full-adder = (3; 2)-counter



CSA para cuatro operandos de 4 bits





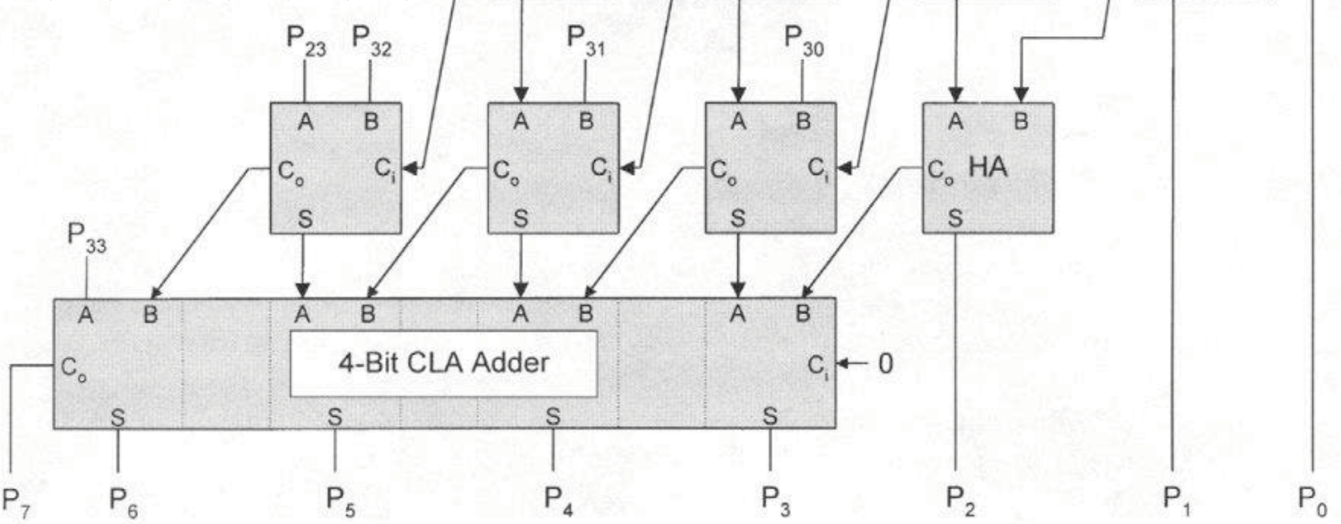
$$c = \frac{(x+y+z)-s}{2}$$

$$s = (x+y+z) \bmod 2$$

- Los dos niveles superiores son **CSAs** de 4-bits.
- El tercer nivel es un sumador de propagación de acarreo (**CPA**)
  - El **CPA** puede substituirse por un **CLA**
- Interconexión adecuada para garantizar sumas de igual peso.
- Si se suman k operandos $X_1, X_2, \ldots X_k$, se necesitan $(k-2)$ **CSAs** y un **CPA**, y el retraso en sumarlos es: $(k-2)\cdot T_{CSA} + T_{CPA}$.
- La suma de k operandos de n bits de longitud puede llegar a ser $k(2^n - 1)$.
- El resultado de la suma final puede ocupar $n + \lceil \log_2 k \rceil$ bits.

Multiplicador paralelo basado en CSA

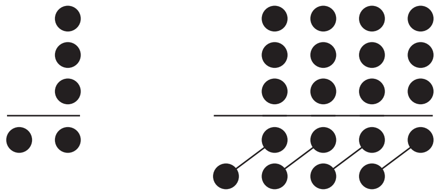
			P_{03}	P_{02}	P_{01}	P_{00}
		P_{13}	P_{12}	P_{11}	P_{10}	
	P_{23}	P_{22}	P_{21}	P_{20}		
P_{33}	P_{32}	P_{31}	P_{30}			



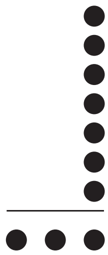
R.F. Tinder: "Engineering Digital Design", Academic Press, 2000

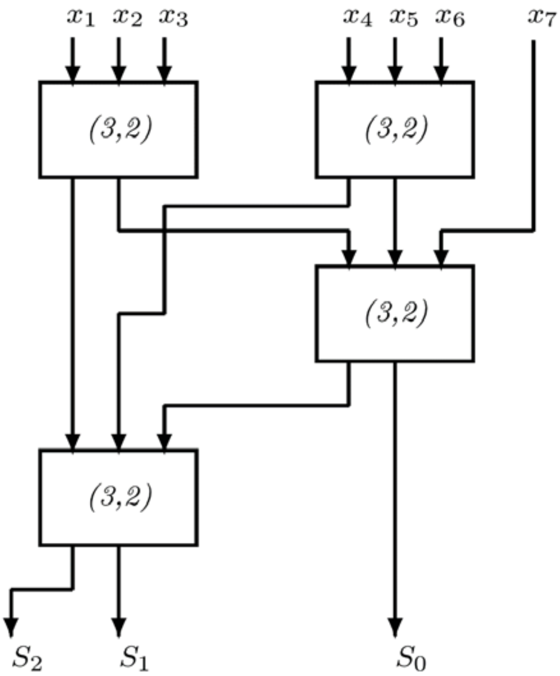
Multiplicación: uso de sumadores multioperando

1-bit full-adder ➪ (3, 2)-counter



Circuito que reduce 7 bits a su suma de 3-bits ➪ $(7, 3)$-counter





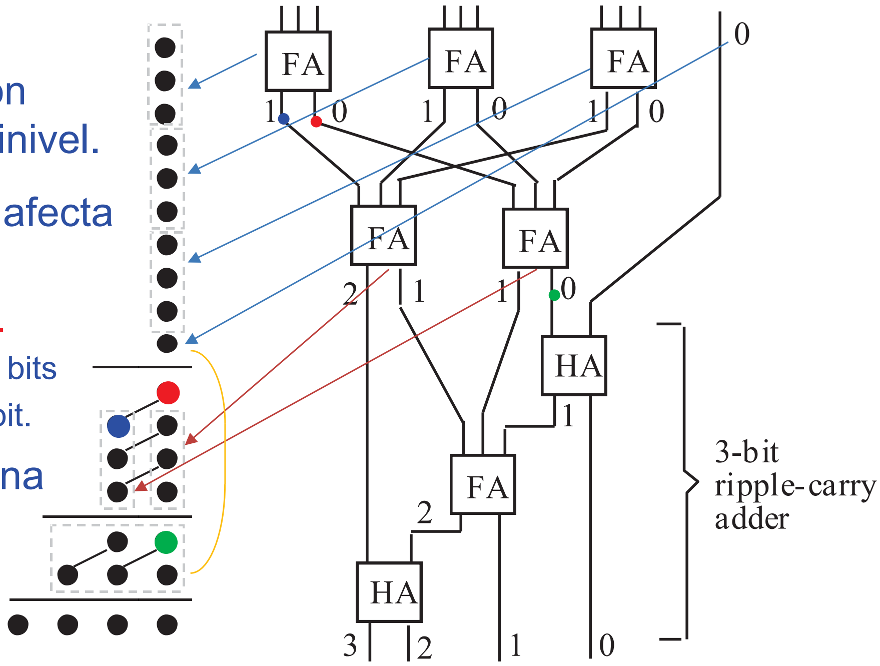
Circuito que reduce k bits a su suma de $m = \lceil \log_2(k+1) \rceil$-bit ➪ (k, m)-counter

Requiere 4 contadores (3,2) en tres niveles → no hay mejora en velocidad

Multiplicación: uso de sumadores multioperando

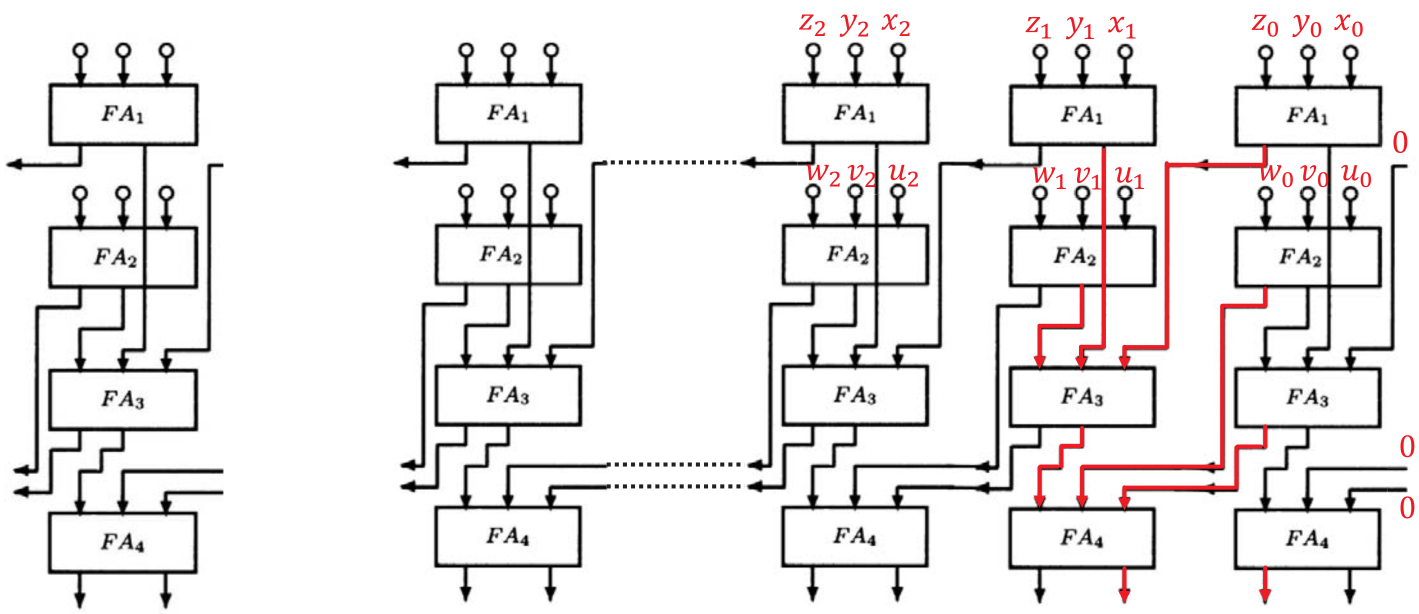
$(7, 3)$-counter

- Puede implementarse con técnicas de síntesis multinivel.
- El nº de interconexiones afecta al área de silicio:
  - Se prefiere el $(7, 3)$ al $(3, 2)$.
  - $(7, 3)$: 10 conex. y reduce 4 bits
  - $(3, 2)$: 5 conex. y reduce 1 bit.
- Se puede encontrar alguna mejora de velocidad al construir mediante **ROM** contadores (k, m) con k alto.



Contador paralelo de 10 entradas, conocido como un contador (10, 4)

Multiplicación: uso de sumadores multioperando



An implementation
of a 6-input bit-slice

Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
  - Operación y técnicas básicas
  - Multiplicadores de altas prestaciones
    - usando sumadores multioperando (carry save addition)
    - reduciendo el número de productos parciales
    - usando multiplicadores más pequeños
    - multiplicación paralela
  - Multiplicadores en FPGA
- División
- Generadores de función
- Ejercicios resueltos

Reduciendo el número de productos parciales

- Si se examinan simultáneamente dos o más bits del multiplicador, entonces podemos reducir el número de productos parciales.
- Un algoritmo que utiliza esta técnica es el conocido como algoritmo de Booth
  - ⇨ si hay una cadena unos repetidos, el valor se puede descomponer como una resta de dos números con un número menor de unos:

    `01111100 = 10000000 - 00000100`

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

$7A = (2^3 - 2^0)A = 2^3A - 2^0A = 2^3A + A_{(\mathrm{Ca2})}$ 0111 → 1000-0001

```
        0 1 1 0
      × 0 1 1 1
      ---------
        0 1 1 0
      0 1 1 0
    0 1 1 0
  0 0 0 0
---------------
0 0 1 0 1 0 1 0
```

```
        0 1 1 0
      × 0 1 1 1
1 1 1 1 1 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 0
0 0 1 1 0
---------------
0 0 1 0 1 0 1 0
```

×(-1) → -6

-0110 →
11111001+00000001
→ 11111010

```
        0 1 1 0
      × 0 1 1 1
1 1 1 1 1 0 1 0
0 0 1 1 0 0 0
---------------
0 0 1 0 1 0 1 0
```


Multiplicación de números con signo: algoritmo de Booth

⇨ Se examina el multiplicador en parejas de bits $x_i x_{i-1}$.

$x_i x_{i-1} =$ 00 → ninguna operación aritmética (en medio de una cadena de 0's)
01 → se suma el multiplicando (fin de la cadena de unos) en posición "i"
10 → se resta el multiplicando (comienzo de la cadena de unos) en posición "i"
En esta resta debe conservarse el signo (extensión de signo)
11 → ninguna operación aritmética (en medio de una cadena de 1's)

```
      0 1 1 0
    × 0 1 1 1 (0)
1 1 1 1 1 0 1 0
```

Se resta el multiplicando: complemento a 2 con extensión del signo.

```
        0 1 1 0
    x 0 1 1 1 (0)
1 1 1 1 1 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 0
```

En medio de una cadena de 0's

```
        0 1 1 0
      × 0 1 1 1 (0)
1 1 1 1 1 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 0
0 0 1 1 0
---------------
0 0 1 0 1 0 1 0
```

Se suma el multiplicando

Recodificación de Booth

$x_i x_{i-1}$	y_i
0 0	0
0 1	1
1 0	-1
1 1	0

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

x_{n-1}	x_{n-2}	y_{n-1}
1	0	-1
1	1	0

Multiplicación: algoritmo de Booth – ejemplo

➪ Se examina el multiplicador en parejas de bits $x_i x_{i-1}$.

$x_i x_{i-1} =$ `00` → ninguna operación aritmética (en medio de una cadena de `0`’s)
`01` → se suma el multiplicando (fin de la cadena de unos) en posición “i”
`10` → se resta el multiplicando (comienzo de la cadena de unos) en posición “i”
En esta resta debe conservarse el signo (extensión de signo)
`11` → ninguna operación aritmética (en medio de una cadena de `1`’s)

A			1 0 1 1		-5
X		×	1 1 0 1		-3
Y	Recodific.		0 $\bar{1}$ 1 $\bar{1}$		
	➪ Add $-A$		0 1 0 1		
	➪ Shift		0 0 1 0	1	
	➪ Add A	+	1 0 1 1		
			1 1 0 1	1	
	➪ Shift		1 1 1 0	1 1	
	➪ Add $-A$	+	0 1 0 1		
			0 0 1 1	1 1	
	➪ Shift		0 0 0 1	1 1 1	+15

Recodificación de Booth

x_i	x_{i-1}	y_i
0	0	0
0	1	1
1	0	$\bar{1}$
1	1	0

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

x_{n-1}	x_{n-2}	y_{n-1}
1	0	$\bar{1}$
1	1	0

Multiplicación: algoritmo de Booth

- La multiplicación comienza por el bit menos significativo.
- Se pueden manejar multiplicaciones en complemento a 2 de forma adecuada.
- Si se van a multiplicar números sin signo, entonces debemos añadir un cero a la izquierda del multiplicador ($x_n = 0$) para asegurar un resultado correcto.

Sin embargo,

- El número de operaciones de adición/substracción es variable y, por tanto, también lo es el de desplazamientos entre dos operaciones consecutivas de adición/substracción.
  - ➪ muy inconveniente al diseñar un multiplicador síncrono.
- El algoritmo se vuelve ineficiente cuando hay unos aislados
  - ➪ por ejemplo, `001010101(0)` se recodifica como $01\bar{1}1\bar{1}1\bar{1}1\bar{1}$, que requiere 8 operaciones, en vez de 4.

❖ Esta situación puede mejorarse examinando simultáneamente grupos de 3 bits de X.

Multiplicación: algoritmo de Booth modificado

- Se conoce como algoritmo de Booth modificado (Radix-4)
- Los bits x_i y x_{i-1} se recodifican en y_i e y_{i-1} , sirviendo x_{i-2} como bit de referencia.
  - → De forma separada, los bits x_{i-2} y x_{i-3} se recodifican en y_{i-2} e y_{i-3}, sirviendo x_{i-4} como bit de referencia.
- Así, se superponen grupos de 3 bits.

$$\bullet\ \bullet\ \bullet\quad x_7\ x_6\quad x_5\ x_4\quad x_3\ x_2\quad x_1\ x_0\quad (x_{-1})$$

$$y_7\,y_6\qquad y_5\,y_4\qquad y_3\,y_2\qquad y_1\,y_0$$

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

- $i = 1, 3, 5, \ldots$
- Los 0/1 aislados se manejan muy eficientemente.
- Si x_{i-1} es un 1 aislado, $x_{i-1} = 1$, por lo que sólo se precisa una operación.
- Si x_{i-1} es un 0 aislado, sólo se precisa una operación.

- Como regla para obtener la operación requerida: calcule $x_{i-1} + x_{i-2} - 2x_i$ y represente el resultado como un número $y_i y_{i-1}$ con codificación SD.

Multiplicación: algoritmo de Booth modificado

- El patrón `01|01|01|01|(0)` se recodifica como `01|01|01|01|`, con lo que el número de operaciones es el mismo que en la multiplicación original.
- El patrón `00|10|10|10|(0)` se recodifica como $01|0\bar{1}|0\bar{1}|\bar{1}0|$, que requiere una operación más que en la multiplicación original.
- Comparado con el algoritmo de Booth original, hay menos patrones para el que el número de productos parciales se incrementa.
- La multiplicación de números complemento a dos se maneja correctamente siempre que n sea par. En otro caso, se precisa una extensión del bit de signo.
  - ⇨ También se precisa añadir un cero a la izquierda del multiplicador si se van a multiplicar números sin signo y n es impar

Multiplicación: algoritmo de Booth modificado – ejemplo

- Ejemplo 1: Considere el producto de 17 por –9

A			01	00	01			17
X		×	11	01	11			–9
Y	Recodific.		$0\bar{1}$	10	$0\bar{1}$			
			$-A$	$+2A$	$-A$			
	➪ Add $-A$		10	11	11			
	➪ 2-bit Shift	1	11	10	11	11		
	➪ Add $2A$	+ 0	10	00	10			
			01	11	01	11		
	➪ 2-bit Shift		00	01	11	01	11	
	➪ Add $-A$	+	10	11	11			
			11	01	10	01	11	–153

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

- Se requieren 3 pasos
- Todas las operaciones de desplazamiento son de dos bits
- Se requiere un bit adicional para el almacenaje del signo correcto cuando se suman 2A.

Multiplicación: algoritmo de Booth modificado – ejemplo

- Ejemplo 2: Considere el producto de –9 por –9

A			11	01	11			–9
X		×	11	01	11			–9
Y	Recodific.		$0\bar{1}$	10	$0\bar{1}$			
			$-A$	$+2A$	$-A$			
	➪ Add $-A$		00	10	01			
	➪ 2-bit Shift		00	00	10	01		
	➪ Add $2A$	+ 1	10	11	10			
		1	11	00	00	01		
	➪ 2-bit Shift		11	11	00	00	01	
	➪ Add $-A$	+	00	10	01			
			00	01	01	00	01	81

x_i	x_{i-1}	x_{i-2}	y_i	y_{i-1}	operation	comments
0	0	0	0	0	$+0$	string of zeros
0	1	0	0	1	$+A$	a single 1
1	0	0	$\bar{1}$	0	$-2A$	beginning of 1's
1	1	0	0	$\bar{1}$	$-A$	beginning of 1's
0	0	1	0	1	$+A$	end of 1's
0	1	1	1	0	$+2A$	end of 1's
1	0	1	0	$\bar{1}$	$-A$	a single 0
1	1	1	0	0	$+0$	string of 1's

Tema 6. Multiplicación, división y generación de funciones

- **Multiplicación**
  - Operación y técnicas básicas
  - **Multiplicadores de altas prestaciones**
    - usando sumadores multioperando (carry save addition)
    - reduciendo el número de productos parciales
    - **usando multiplicadores más pequeños**
    - multiplicación paralela
  - Multiplicadores en FPGA
- División
- Generadores de función
- Ejercicios resueltos

Multiplicación usando multiplicadores más pequeños

- Si un multiplicador de $n \times n$ bits se construye como un circuito integrado, entonces podemos emplearlos para construir multiplicadores más complejos.
- Un multiplicador de $2n \times 2n$ bits puede construirse a partir de cuatro multiplicadores de $n \times n$ bits ya que:

$$A \cdot X = (A_H \cdot 2^n + A_L) \cdot (X_H \cdot 2^n + X_L) = A_H X_H 2^{2n} + (A_H X_L + A_L X_H)2^n + A_L X_L$$

donde el subíndice H o L corresponde a las mitades más o menos significativas.

→ 4 productos parciales de $2n$ bits correctamente alineados antes de la suma

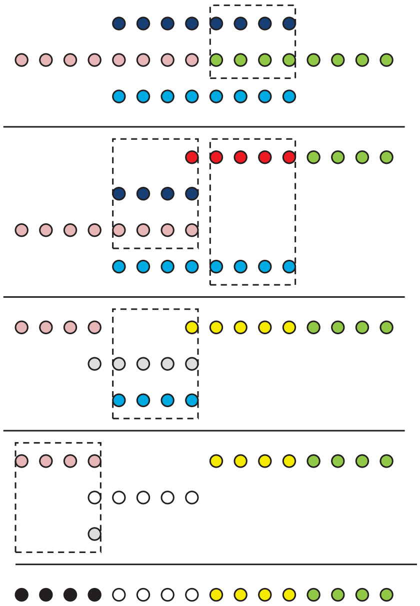
$$\begin{array}{rcc} & \boxed{A_H} & \boxed{A_L} \\ \times & \boxed{X_H} & \boxed{X_L} \\ \hline \end{array}$$

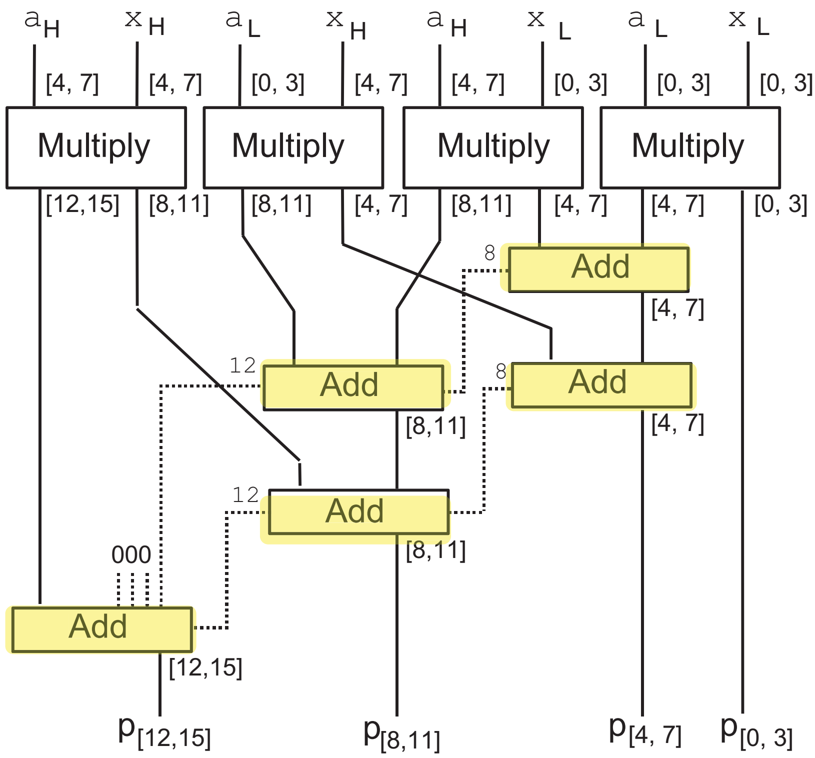
→ los n bits menos significativos son parte directa del resultado final.

→ Tal como está, no es la mejor disposición para obtener la suma.

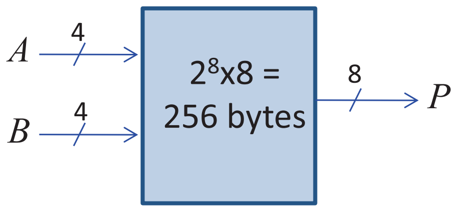
Multiplicación usando multiplicadores más pequeños: ejemplo

Using 4 × 4 multipliers and 4-bit adders to synthesize an 8 × 8 multiplier.



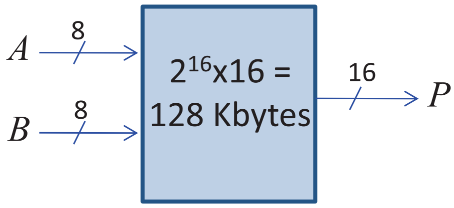


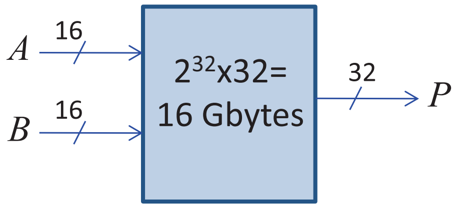
Multiplicador *look-up table* (LUT)



A	B	P
0000	0000	00000000
0000	0001	00000000
...	...	...
1010	1101	10000010
1010	1110	10001100
...	...	...
1111	1110	11010010
1111	1111	11100001

Contenido de la memoria





+ coste: crece exponencialmente

– velocidad

+ modularidad

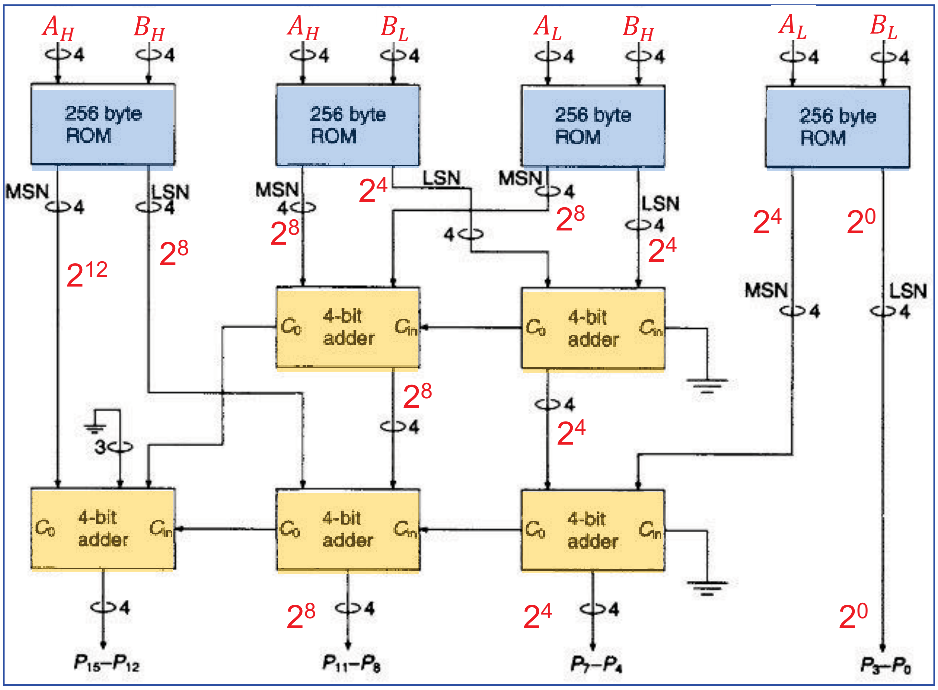
Multiplicador *look-up table* (LUT)

$A = \underbrace{A_7A_6A_5A_4}_{A_H}\underbrace{A_3A_2A_1A_0}_{A_L} = 2^4A_H + A_L$

$B = \underbrace{B_7B_6B_5B_4}_{B_H}\underbrace{B_3B_2B_1B_0}_{B_L} = 2^4B_H + B_L$

$P = A \times B =$
$(2^4A_H + A_L) \times (2^4B_H + B_L)$

$$P = 2^8A_HB_H + 2^4A_HB_L + 2^4A_LB_H + A_LB_L$$

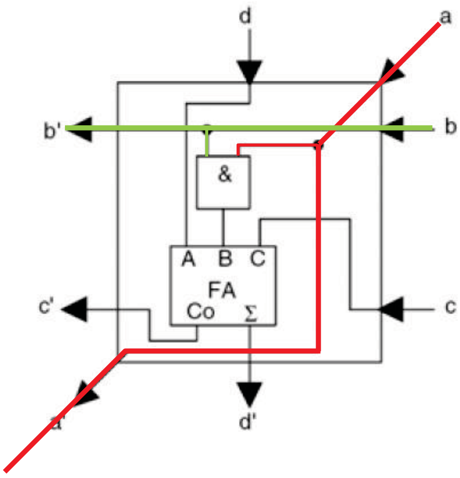


B. Holdsworth and R.C. Woods: "*Digital Logic Design*", Elsevier, 2003

Tema 6. Multiplicación, división y generación de funciones

- **Multiplicación**
  - Operación y técnicas básicas
  - **Multiplicadores de altas prestaciones**
    - usando sumadores multioperando (carry save addition)
    - reduciendo el número de productos parciales
    - usando multiplicadores más pequeños
    - **multiplicación paralela**
  - Multiplicadores en FPGA
- División
- Generadores de función
- Ejercicios resueltos

Multiplicador paralelo

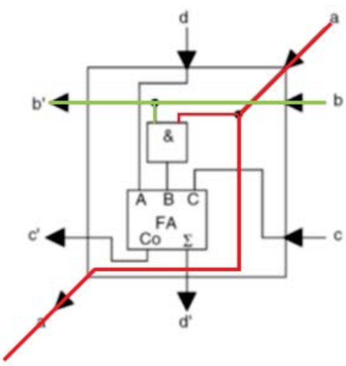


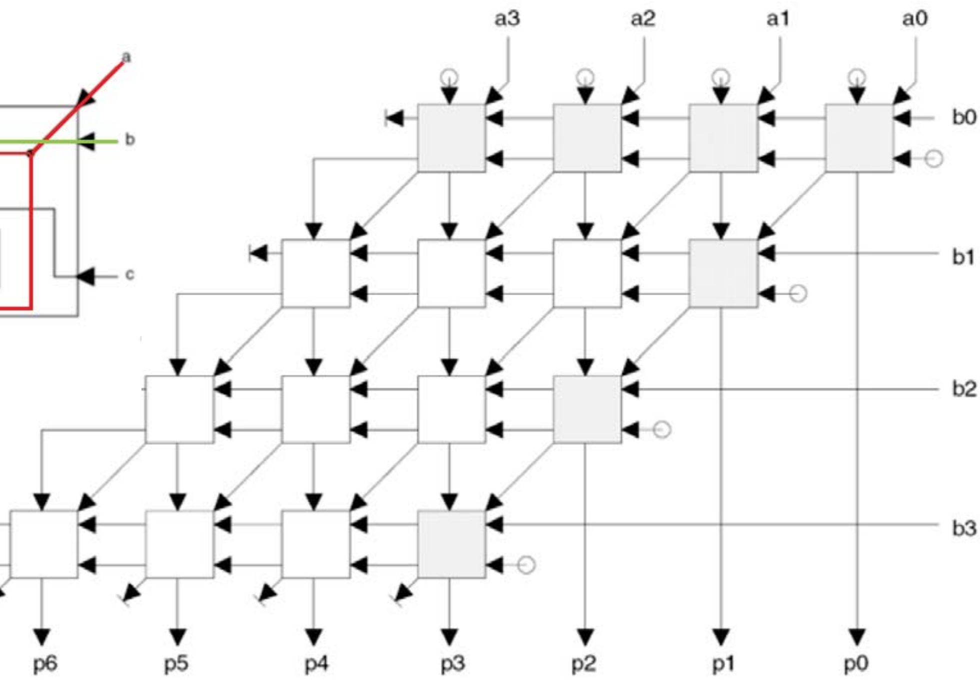
$$a' = a$$
$$b' = b$$
$$c' = a \cdot b \cdot d + a \cdot b \cdot c + c \cdot d$$
$$d' = (a \cdot b \oplus d) \oplus c$$

				a_3	a_2	a_1	a_0	= A
			X	b_3	b_2	b_1	b_0	= B
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	
			a_3b_1	a_2b_1	a_1b_1	a_0b_1		
		a_3b_2	a_2b_2	a_1b_2	a_0b_2			
+	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	P = A x B

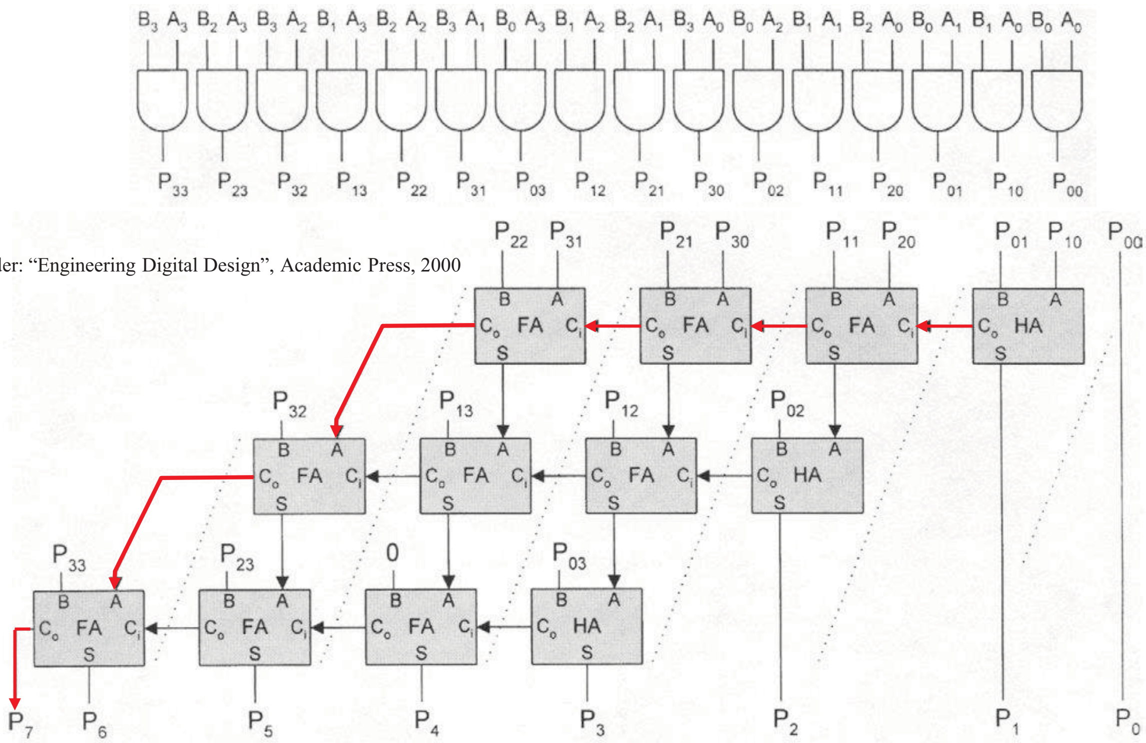
P.Y.K. Cheung, Course of Digital System Design, Imperial College of Science, Technology and Medicine, University of London, Spring 2009

Multiplicador paralelo





Multiplicador paralelo



R.F. Tinder: “Engineering Digital Design”, Academic Press, 2000

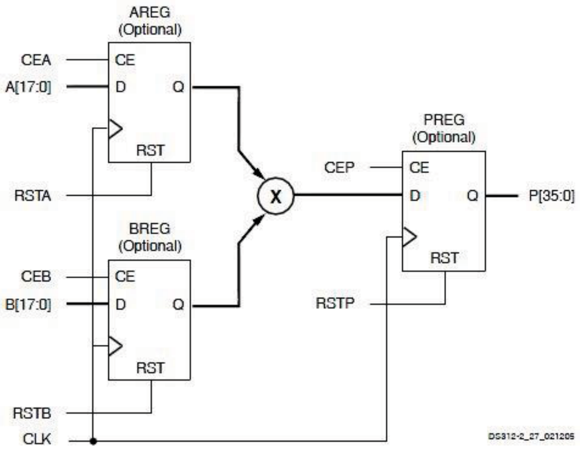
la velocidad depende del retraso en la propagación del acarreo.

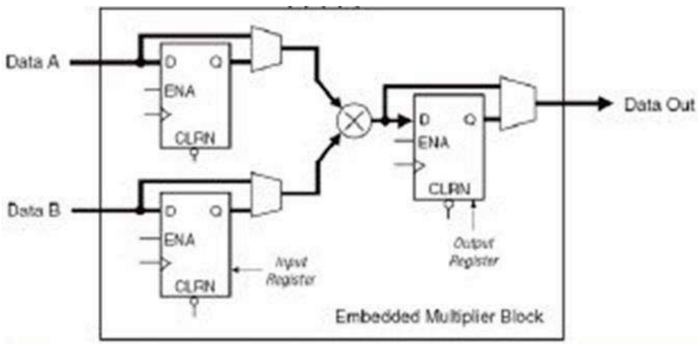
– velocidad

Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
  - Operación y técnicas básicas
  - Multiplicadores de altas prestaciones
  - Multiplicadores en FPGA
- División
- Generadores de función
- Ejercicios resueltos

Multiplicadores empotrados en FPGA



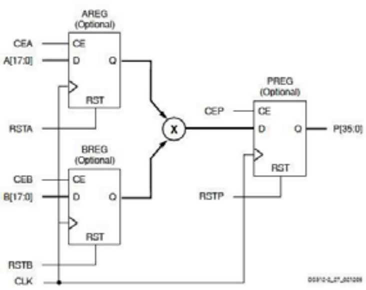


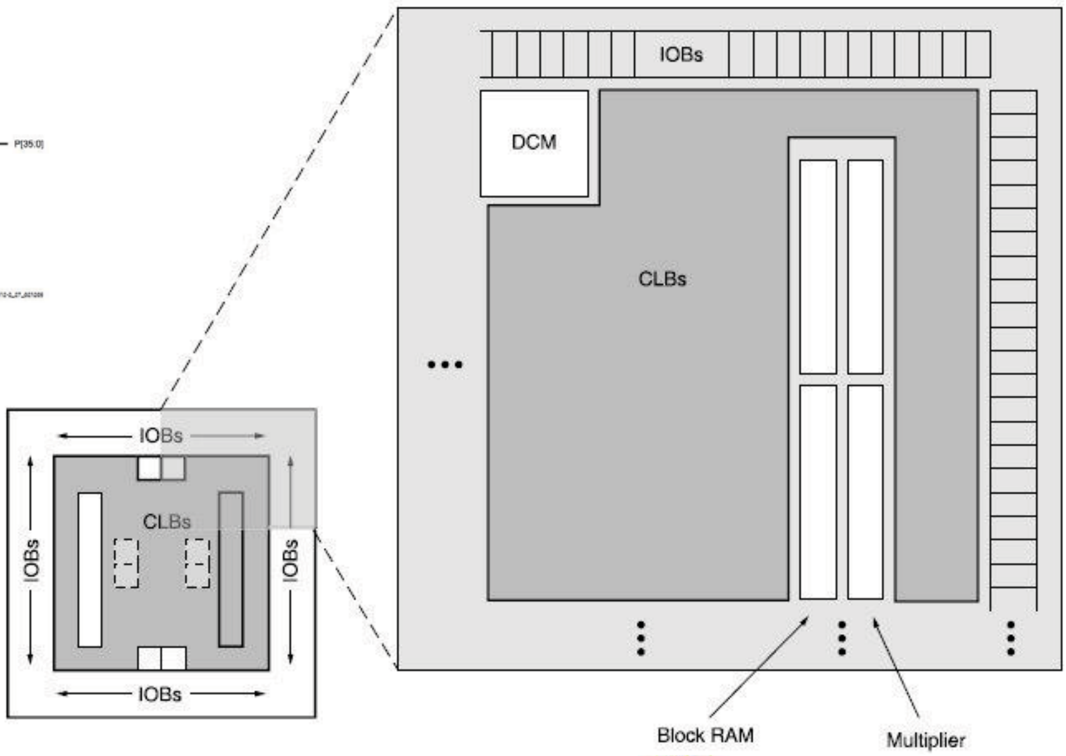
Xilinx Spartan3 MULT18x18SIO

Altera Ciclone II 18x18 multiplier

Multiplicadores empotrados en FPGA

Xilinx Spartan3E MULT18x18SIO

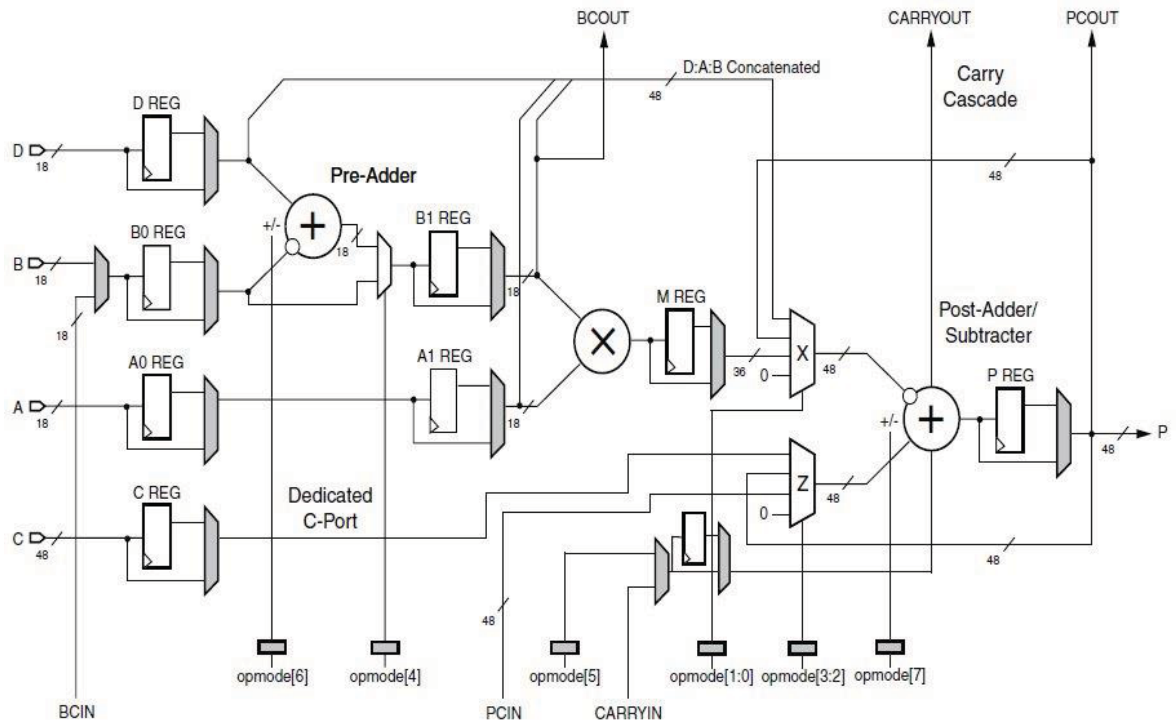




Xilinx: “Spartan-3E FPGA Family: Complete Data Sheet”, 2005

Multiplicadores empotrados en FPGA

Slice del DSP48A permite realizar multiplicaciones y sumas acumulando o no los resultados



M. Salazar Arcucci: "Estudio del Módulo DSP48A de la familia Spartan-3A DSP de Xilinx", Universidad de Alcalá, 2009

Multiplicadores empotrados en FPGA

- En función de señales de control el bloque DSP48A puede realizar las operaciones que se muestran en la tabla.
  Esto permite reducir el número de recursos utilizados en el FPGA (número de CLBs).
- Estos circuitos están basados en una arquitectura paralela que permite realizar las operaciones en un ciclo de reloj.

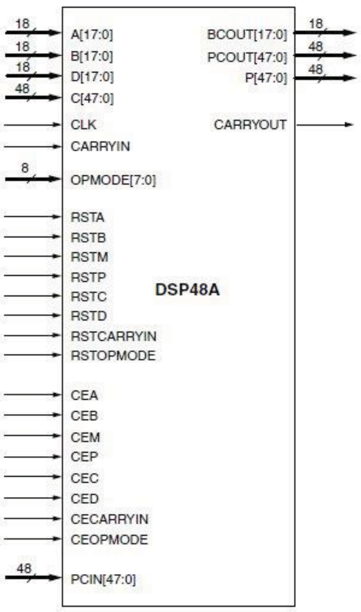
Modos de operación del DSP48A

$$P = C \pm (A \times (D \pm B) + CARRYIN)$$

Modo	Ecuación
Multiplicador	$\pm(A \times B + CARRYIN)$
Pre-sumador/Multiplicador	$\pm(A \times (D \pm B) + CARRYIN)$
Pre-sumador/Sumador en cascada	$PCIN \pm D : A : (D \pm B) + CARRYIN$
Pre-sumador/Multiplicador/Sumador en cascada	$PCIN \pm (A \times (D \pm B) + CARRYIN)$
Pre-sumador/Multiplicador/Sumador realimentado	$P \pm (A \times (D \pm B) + CARRYIN)$
Sumador de 48 bits con C	$C \pm D : A : B + CARRYIN$
Pre-sumador/Multiplicador/Sumador con C	$C \pm (A \times (D \pm B) + CARRYIN)$
Pre-sumador/Sumador de 48 bits	$C \pm D : A : (D \pm B) + CARRYIN$

M. Salazar Arcucci: "Estudio del Módulo DSP48A de la familia Spartan-3A DSP de Xilinx", Universidad de Alcalá, 2009

Multiplicadores empotrados en FPGA



Verilog Instantiation Template

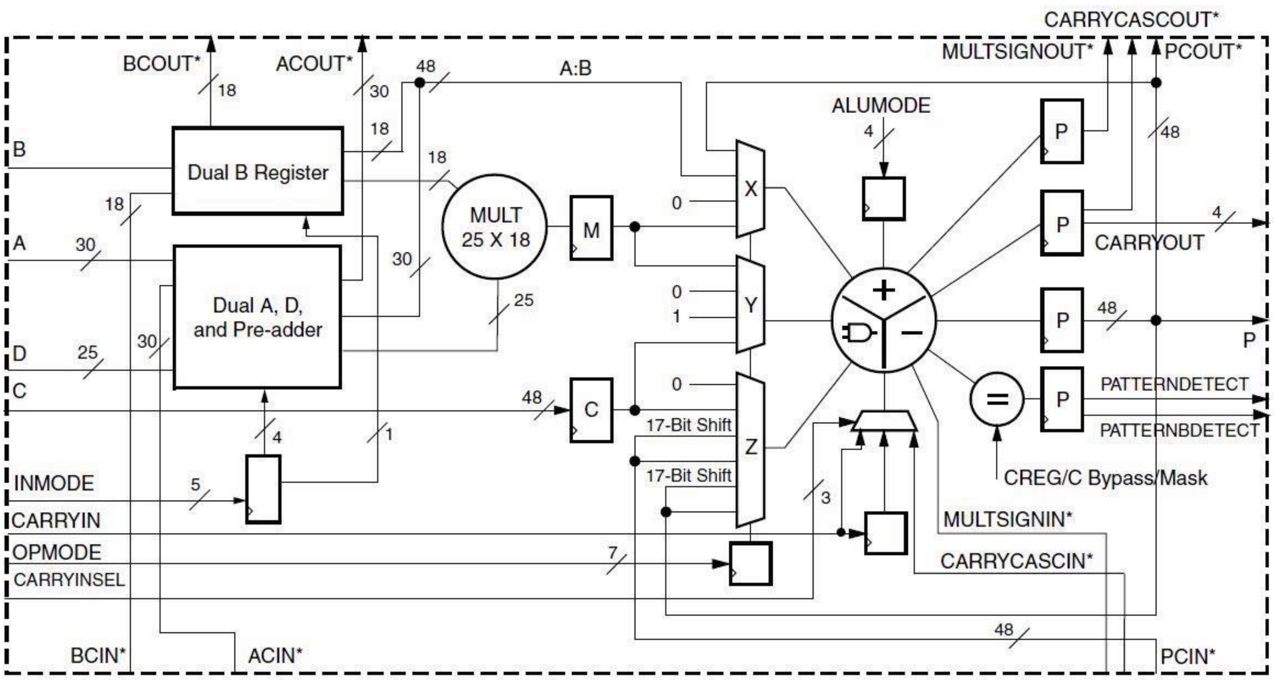
```
// DSP48A: DSP Function Block
//         Spartan-3A DSP
// Xilinx HDL Libraries Guide, version 12.4

DSP48A #(
    .A0REG(0), // Enable=1/disable=0 first stage A input pipeline register
    .A1REG(1), // Enable=1/disable=0 second stage A input pipeline register
    .B0REG(0), // Enable=1/disable=0 first stage B input pipeline register
    .B1REG(1), // Enable=1/disable=0 second stage B input pipeline register
    .CARRYINREG(1), // Enable=1/disable=0 CARRYIN input pipeline register
    .CARRYINSEL("CARRYIN"), // Specify carry-in source, "CARRYIN" or "OPMODE5"
    .CREG(1),   // Enable=1/disable=0 C input pipeline register
    .DREG(1),   // Enable=1/disable=0 D pre-adder input pipeline register
    .MREG(1),   // Enable=1/disable=0 M pipeline register
    .OPMODEREG(1), // Enable=1/disable=0 OPMODE input pipeline register
    .PREG(1),   // Enable=1/disable=0 P output pipeline register
    .RSTTYPE("SYNC") // Specify reset type, "SYNC" or "ASYNC"
) DSP48A_inst (
    .BCOUT(BCOUT), // 18-bit B port cascade output
    .CARRYOUT(CARRYOUT), // 1-bit carry output
    .P(P), // 48-bit output
    .PCOUT(PCOUT), // 48-bit cascade output
    .A(A), // 18-bit A data input
    .B(B), // 18-bit B data input (can be connected to fabric or BCOUT of adjacent DSP48A)
    .C(C), // 48-bit C data input
    .CARRYIN(CARRYIN), // 1-bit carry input signal
    .CEA(CEA), // 1-bit active high clock enable input for A input registers
    .CEB(CEB), // 1-bit active high clock enable input for B input registers
    .CEC(CEC), // 1-bit active high clock enable input for C input registers
    .CECARRYIN(CECARRYIN), // 1-bit active high clock enable input for CARRYIN registers
    .CED(CED), // 1-bit active high clock enable input for D input registers
    .CEM(CEM), // 1-bit active high clock enable input for multiplier registers
    .CEOPMODE(CEOPMODE), // 1-bit active high clock enable input for OPMODE registers
    .CEP(CEP), // 1-bit active high clock enable input for P output registers
    .CLK(CLK), // Clock input
    .D(D), // 18-bit B pre-adder data input
    .OPMODE(OPMODE), // 8-bit operation mode input
    .PCIN(PCIN), // 48-bit P cascade input
    .RSTA(RSTA), // 1-bit reset input for A input pipeline registers
    .RSTB(RSTB), // 1-bit reset input for B input pipeline registers
    .RSTC(RSTC), // 1-bit reset input for C input pipeline registers
    .RSTCARRYIN(RSTCARRYIN), // 1-bit reset input for CARRYIN input pipeline registers
    .RSTD(RSTD), // 1-bit reset input for D input pipeline registers
    .RSTM(RSTM), // 1-bit reset input for M pipeline registers
    .RSTOPMODE(RSTOPMODE), // 1-bit reset input for OPMODE input pipeline registers
    .RSTP(RSTP) // 1-bit reset input for P output pipeline registers
);

// End of DSP48A_inst instantiation
```


Multiplicadores empotrados en FPGA

Virtex6 Slice del DSP48E1 Multiplicador + ALU



*These signals are dedicated routing paths internal to the DSP48E1 column. They are not accessible via fabric routing resources.

Xilinx: "Virtex-6 FPGA DSP48E1 Slice. User Guide", 2011

Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
- División
  - Operación y técnicas básicas
  - División con restauración
  - División sin restauración
- Generadores de función
- Ejercicios resueltos

División

- La división es la operación más compleja y consumidora de tiempo de las cuatro operaciones aritméticas básicas.
- En general, dados un dividendo X y un divisor D, se genera un cociente Q y un resto R tal que: $X = Q \cdot D + R$ (con $R < D$), suponiendo X, D, Q y R positivos.
- En muchas unidades aritméticas de punto fijo se obtiene un producto de longitud doble después de una operación de multiplicación.
  - → X puede ocupar un registro de longitud doble ($2k$), $X_H X_L$, mientras que los otros operandos se almacenan en registros de longitud simple (k).
  - → Debemos estar seguros de que el cociente resultante Q sea menor o igual que el mayor número que puede almacenarse en un registro de longitud simple

    ⇨ $Q < 2^k$ y $R < D$ → $X = Q \cdot D + R < (2^k - 1)D + D = 2^k D$
  - ❖ Es decir, los k bits de la mitad superior de X, X_H, deben cumplir $X_H < D$.
  - ❖ Si esta condición no se cumple, se debe producir una indicación de *overflow*.
  - ❖ Otra condición que debe chequearse es que $D \neq 0$. Si no fuera el caso, se debería producir una indicación de división por cero (*divide by zero*).

División

- La figura muestra una división entre un número de 8 bits ($2k$ bits en general) y otro de 4 (k bits) en notación de punto (*dot notation*).

$$
\begin{array}{l|cccccccc|cccc}
 & & & & X & & & & & & D & & \\
r^{(0)} & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
r^{(1)} = 2r^{(0)} - q_3 D\, 2^4 & & \bullet & \bullet & \bullet & \bullet & & & & q_3 & q_2 & q_1 & q_0 \quad Q \\
r^{(2)} = 2r^{(1)} - q_2 D\, 2^4 & & & \bullet & \bullet & \bullet & \bullet & & & & & & \\
r^{(3)} = 2r^{(2)} - q_1 D\, 2^4 & & & & \bullet & \bullet & \bullet & \bullet & & & & & \\
r^{(4)} = 2r^{(3)} - q_0 D\, 2^4 & & & & & \bullet & \bullet & \bullet & \bullet & & & & \\
\hline
R & & & & & r_3 & r_2 & r_1 & r_0 & & & &
\end{array}
$$

Dividendo: $X = x_{2k-1}x_{2k-2} \dots x_k\ x_{k-1}x_{k-2} \dots x_1x_0$
Divisor: $D = d_{k-1}d_{k-2} \dots d_1d_0$
Cociente: $Q = q_{k-1}q_{k-2} \dots q_1q_0$
Resto: $R = r_{k-1}r_{k-2} \dots r_1r_0$

→ Las líneas de puntos sucesivas corresponden a la resta de la diferencia previa con producto del divisor y un bit del cociente.

→ Como $q_{k-j} \in \{0,1\}$, cada término $D \cdot q_{k-j}$ es 0 o D, → el problema de la división binaria se reduce a restar del dividendo X, los números 0 o una versión desplazada del divisor D.

→ Comparada con la multiplicación, la división añade la complejidad de elegir el bit del cociente.

División: ejemplo

- Ejemplo de división de $32_{(10}$ por $6_{(10}$.

¿Se satisface $X_H < D$? → SÍ, `100 < 110`

$$
\begin{array}{r|ccccccc|ccc}
 & & & & X & & & & & D & \\
r^{(0)} & & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
2r^{(0)} & 1 & 0 & 0 & 0 & 0 & 0 & & & & \\
\text{¿Puede ser } q_2 = 1\text{?} & - & 1 & 1 & 0 & & & & & & D2^3
\end{array}
$$

Dividendo: $X = x_{2k-1}x_{2k-2} \dots x_k\ x_{k-1}x_{k-2} \dots x_1x_0$
Divisor: $D = d_{k-1}d_{k-2} \dots d_1d_0$
Cociente: $Q = q_{k-1}q_{k-2} \dots q_1q_0$
Resto: $R = r_{k-1}r_{k-2} \dots r_1r_0$

División: ejemplo

- Ejemplo de división de $32_{(10}$ por $6_{(10}$.

	X	D
$r^{(0)}$	1 0 0 0 0 0	1 1 0
$2r^{(0)}$	1 0 0 0 0 0	1
SÍ	- 1 1 0	q_2
$r^{(1)} = 2r^{(0)} - q_2 D 2^3$	1 0 0 0	

Dividendo: $X = x_{2k-1}x_{2k-2} \dots x_k \; x_{k-1}x_{k-2} \dots x_1 x_0$
Divisor: $D = d_{k-1}d_{k-2} \dots d_1 d_0$
Cociente: $Q = q_{k-1}q_{k-2} \dots q_1 q_0$
Resto: $R = r_{k-1}r_{k-2} \dots r_1 r_0$

División: ejemplo

- Ejemplo de división de $32_{(10}$ por $6_{(10}$.

	X	D	
$r^{(0)}$	1 0 0 0 0 0	1 1 0	
$2r^{(0)}$	1 0 0 0 0 0	1	
SÍ	- 1 1 0	q_2	
$r^{(1)} = 2r^{(0)} - q_2 D 2^4$	1 0 0 0		
$2r^{(1)}$	1 0 0 0		
¿Puede ser $q_1 = 1$?	- 1 1 0		$D2^3$

Dividendo: $X = x_{2k-1}x_{2k-2} \dots x_k \; x_{k-1}x_{k-2} \dots x_1 x_0$
Divisor: $D = d_{k-1}d_{k-2} \dots d_1 d_0$
Cociente: $Q = q_{k-1}q_{k-2} \dots q_1 q_0$
Resto: $R = r_{k-1}r_{k-2} \dots r_1 r_0$

División: ejemplo

- Ejemplo de división de $32_{(10}$ por $6_{(10}$.

	X	D
$r^{(0)}$	1 0 0 0 0 0	1 1 0
$2r^{(0)}$	1 0 0 0 0 0	1 0
SÍ	- 1 1 0	q_2 q_1
$r^{(1)} = 2r^{(0)} - q_2 D 2^4$	1 0 0 0	
$2r^{(1)}$	1 0 0 0	
NO		
$r^{(2)} = 2r^{(1)}$	1 0 0 0	

Dividendo: $X = x_{2k-1}x_{2k-2} \dots x_k \; x_{k-1}x_{k-2} \dots x_1 x_0$
Divisor: $D = d_{k-1}d_{k-2} \dots d_1 d_0$
Cociente: $Q = q_{k-1}q_{k-2} \dots q_1 q_0$
Resto: $R = r_{k-1}r_{k-2} \dots r_1 r_0$

División: ejemplo

- Ejemplo de división de $32_{(10}$ por $6_{(10}$.

	X	D
$r^{(0)}$	1 0 0 0 0 0	1 1 0
$2r^{(0)}$	1 0 0 0 0 0	1 0
SÍ	- 1 1 0	q_2 q_1
$r^{(1)} = 2r^{(0)} - q_2 D 2^4$	1 0 0 0	
$2r^{(1)}$	1 0 0 0	
NO		
$r^{(2)} = 2r^{(1)}$	1 0 0 0	
¿Puede ser $q_0 = 1$?	- 1 1 0	$D2^3$

Dividendo: $X = x_{2k-1}x_{2k-2} \dots x_k \; x_{k-1}x_{k-2} \dots x_1 x_0$
Divisor: $D = d_{k-1}d_{k-2} \dots d_1 d_0$
Cociente: $Q = q_{k-1}q_{k-2} \dots q_1 q_0$
Resto: $R = r_{k-1}r_{k-2} \dots r_1 r_0$

División: ejemplo

- Ejemplo de división de $32_{(10}$ por $6_{(10}$.

  - Comprobación: $X = Q \cdot D + R = 101_2 \cdot 110_2 + 10_2 = 5 \cdot 6 + 2 = 32$

	X	D	
$r^{(0)}$	1 0 0 0 0 0	1 1 0	
$2r^{(0)}$	1 0 0 0 0 0	1 0 1	Q
SÍ	- 1 1 0	q_2 q_1 q_0	
$r^{(1)} = 2r^{(0)} - q_2 D 2^2$	1 0 0 0		
$2r^{(1)}$	1 0 0 0		
NO			
$r^{(2)} = 2r^{(1)}$	1 0 0 0		
SÍ	- 1 1 0		
	1 0		
	r_1 r_0		

Dividendo:	$X =$	$x_{2k-1}x_{2k-2} \dots x_k \; x_{k-1}x_{k-2} \dots x_1 x_0$
Divisor:	$D =$	$d_{k-1}d_{k-2} \dots d_1 d_0$
Cociente:	$Q =$	$q_{k-1}q_{k-2} \dots q_1 q_0$
Resto:	$R =$	$r_{k-1}r_{k-2} \dots r_1 r_0$

División

- En el paso j del proceso, el resto se compara con el divisor D. Si el resto es el mayor, entonces el bit q_j del cociente se fija a 1, si no, a 0.

  - Resto parcial paso j: $r^{(j)} = 2r^{(j-1)} - q_{k-j}(2^k D), j = 1, \dots, k-1; r^{(0)} = X; r^{(k)} = 2^k R$

    desp. a la izq.

    resta

  El factor 2^k por el que se premultiplica D asegura una alineación adecuada de los valores.

- Después de k iteraciones, la recurrencia precedente conduce a:

$$r^{(1)} = 2r^{(0)} - q_{k-1}(2^k D)$$
$$r^{(2)} = 2r^{(1)} - q_{k-2}(2^k D) = 2[2r^{(0)} - q_{k-1}(2^k D)] - q_{k-2}(2^k D) = 2^2 r^{(0)} - q_{k-1}(2^{k+1} D) - q_{k-2}(2^k D)$$
$$r^{(3)} = 2r^{(2)} - q_{k-3}(2^k D) = 2^3 r^{(0)} - q_{k-1}(2^{k+2} D) - q_{k-2}(2^{k+1} D) - q_{k-3}(2^k D)$$
$$r^{(4)} = 2r^{(3)} - q_{k-4}(2^k D) = 2^4 r^{(0)} - q_{k-1}(2^{k+3} D) - q_{k-2}(2^{k+2} D) - q_{k-3}(2^{k+1} D) - q_{k-4}(2^k D)$$
$$r^{(5)} = 2r^{(4)} - q_{k-5}(2^k D) = 2^5 r^{(0)} - q_{k-1}(2^{k+4} D) - q_{k-2}(2^{k+3} D) - q_{k-3}(2^{k+2} D) - q_{k-4}(2^{k+1} D) - q_{k-5}(2^k D)$$

………

$$r^{(k)} = 2r^{(k-1)} - q_0(2^k D) = 2^k r^{(0)} - 2^k D(q_0 2^0 + q_1 2^1 + q_2 2^2 + \dots + q_{k-1} 2^{k-1})$$
$$r^{(k)} = 2r^{(k-1)} - q_0(2^k D) = 2^k r^{(0)} - 2^k D \cdot Q \quad \Rightarrow \quad 2^{-k} r^{(k)} = X - Q \cdot D$$

División

- La división entera puede reformularse como división fraccionaria y viceversa.
  - ➪ $X = Q \cdot D + R \;\rightarrow\; 2^{-2k}X = 2^{-2k}Q \cdot D + 2^{-2k}R \;\rightarrow\; X_f = Q_f D_f + 2^{-k}R_f$

  Es decir, podemos dividir fracciones de la misma manera en que dividimos enteros, pero el resto debe desplazarse hacia la derecha k bits.
  - ❖ La condición de no *overflow* debe ser ahora $X_f < D_f$.
- La presentación del algoritmo de la división es más simple cuando dividendo, divisor, cociente y resto se interpretan como fracciones.
- Para obtener el cociente fraccionario $Q = 0.q_{-1}q_{-2}\dots q_{-(k-1)}$, realizamos la división como una secuencia de substracciones y desplazamientos.
- En el caso fraccionario:
  - ➪ Resto parcial paso j: $r_f^{(j)} = 2r_f^{(j-1)} - q_{-j}D_f;\; j = 1, 2, \dots, k-1;$

    $r_f^{(0)} = X_f; \qquad r_f^{(k)} = 2^k R_f$

División: algunos ejemplos – división entera

❖ Ejemplo: Supongamos $X = 117_{(10} =$ `01110101`$_{(2}$ y $D = 10_{(10} =$ `1010`$_{(2}$

X				0	1	1	1	0	1	0	1	
2^4D				1	0	1	0					
$r^{(0)}$				0	1	1	1	0	1	0	1	
$2r^{(0)}$			0	1	1	1	0	1	0	1		
$-q_3 2^4D$	¿?			1	0	1	0					$q_3 = 1$
$r^{(1)}$				0	1	0	0	1	0	1		
$2r^{(1)}$			0	1	0	0	1	0	1			
$-q_2 2^4D$	¿?			0	0	0	0					$q_2 = 0$
$r^{(2)}$			0	1	0	0	1	0	1			
$2r^{(2)}$		0	1	0	0	1	0	1				
$-q_1 2^4D$	¿?			1	0	1	0					$q_1 = 1$
$r^{(3)}$			0	1	0	0	0	1				
$2r^{(3)}$			1	0	0	0	1					
$-q_0 2^4D$	¿?			1	0	1	0					$q_0 = 1$
$r^{(4)}$				0	1	1	1					
R								0	1	1	1	$r^{(4)} = 2^4R$
Q								1	0	1	1	

- Dividendo: ocupa registro de longitud doble
- Se satisface $X_H < D$
- Cociente: $Q = 1011_{(2} = 11_{(10}$
- Resto: $r^{(4)}2^{-4} = 7$
- Comprobación: $X = Q \cdot D + R = 11 \cdot 10 + 7$

División: algunos ejemplos – división fraccionaria

❖ Ejemplo: Supongamos $X = 0.01110101_{(2} = \left(\frac{117}{256}\right)_{(10}$ y $D = 0.1010_{(2} = \left(\frac{10}{16}\right)_{(10}$

X_f		.0 1 1 1 0 1 0 1	
D_f		.1 0 1 0	
$r^{(0)}$		.0 1 1 1 0 1 0 1	
$2r^{(0)}$		0 .1 1 1 0 1 0 1	
$-q_{-1}D_f$	¿?	.1 0 1 0	$q_{-1} = 1$
$r^{(1)}$		.0 1 0 0 1 0 1	
$2r^{(1)}$		0 .1 0 0 1 0 1	
$-q_{-2}D_f$	¿?	0 0 0 0	$q_{-2} = 0$
$r^{(2)}$		0 .1 0 0 1 0 1	
$2r^{(2)}$		1 .0 0 1 0 1	
$-q_{-3}D_f$	¿?	.1 0 1 0	$q_{-3} = 0$
$r^{(3)}$		0 .1 0 0 0 1	
$2r^{(3)}$		1 .0 0 0 1	
$-q_{-4}D_f$	¿?	.1 0 1 0	$q_{-4} = 1$
$r^{(4)}$		.0 1 1 1	
R_f		.0 0 0 0 0 1 1 1	$r^{(4)} = 2^4 R_f$
Q_f		.1 0 1 1	

- Se satisface $X_f < D_f$
- Resultado final: $Q = (0.1011)_2 = \frac{11}{16}$
- Resto: $r^{(4)} 2^{-4} = (0.0111) \cdot 2^{-4} = \frac{7}{16} \cdot 2^{-4} = \frac{7}{256}$
- Comprobación:

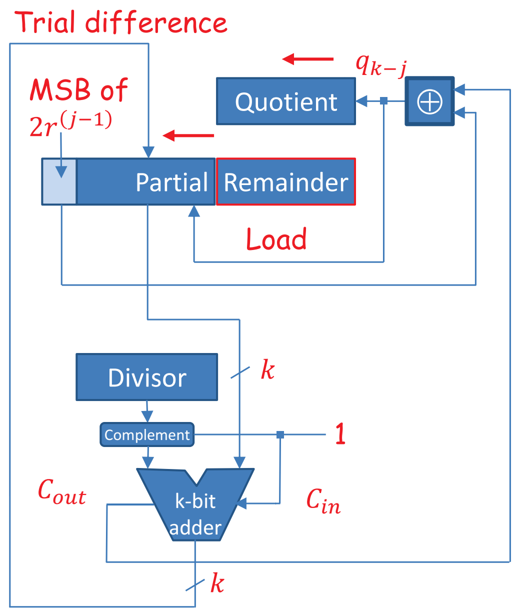
$$X_f = Q_f D_f + R_f = \frac{11}{16} \cdot \frac{10}{16} + \frac{7}{256} = \frac{117}{256}$$

Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
- **División**
  - Operación y técnicas básicas
  - **División con restauración**
  - División sin restauración
- Generadores de función
- Ejercicios resueltos

División: división con restauración

- La figura muestra una realización *hardware* del algoritmo de división secuencial que acabamos de ver para enteros sin signo.
  - → Al comienzo de cada ciclo, el resto intermedio $r^{(j-1)}$ se desplaza a la izquierda, y su MSB → registro especial
  - → Se evalúa la diferencia $r^{(j)} = 2r^{(j-1)} - q_{k-j}(2^k D)$. Con el factor 2^k, alineamos el divisor con los k bits superiores del resto intermedio (*partial remainder, PR*).
  - → El bit del cociente será 1 si el MSB de $2r^{(j-1)}$ es 1 o si la diferencia es positiva ($C_{out} = 1$). En este caso:
    - $q_{k-j} = 1$ se introduce en el registro del cociente, Q
    - $r^{(j)}$ → mitad superior de PR, formando el nuevo registro PR en el siguiente ciclo.
  - → En otro caso, $q_{k-j} = 0$ y el resto parcial no se carga.



- El esquema de división de la figura se conoce como división con restauración (*restoring división*).
  - → El resto se restaura a su valor correcto si el resultado de la resta indica que 1 no era la elección correcta para q_{k-j}.

División con restauración: ejemplo

- Ejemplo: Supongamos $X = 01110101_{(2} = 117_{(10}$ y $D = 1010_{(2} = 10_{(10}$

X		0 1 1 1	0 1 0 1	No overflow, since:	
$2^4 D$	0	1 0 1 0		$0111_{(2} < 1010_{(2}$	
$-2^4 D$	1	0 1 1 0			
$r^{(0)} = X$		0 1 1 1	0 1 0 1		117
$2r^{(0)}$	0	1 1 1 0	1 0 1		
⇨ Add $(-2^4 D)$	+ 1	0 1 1 0			
$r^{(1)} = 2r^{(0)} - 2^4 D$	0	0 1 0 0	1 0 1	Positive, so set $q_3 = 1$	74
$2r^{(1)}$	0	1 0 0 1	0 1		
$+(-2^4 D)$	+ 1	0 1 1 0			
$r^{(2)} = 2r^{(1)} - 2^4 D$	1	1 1 1 1	0 1	Negative, so set $q_2 = 0$	
$r^{(2)} = 2r^{(1)}$	0	1 0 0 1	0 1	and restore	128
$2r^{(2)}$	1	0 0 1 0	1		
$+(-2^4 D)$	+ 1	0 1 1 0			
$r^{(3)} = 2r^{(2)} - 2^4 D$	0	1 0 0 0	1	Positive, so set $q_1 = 1$	136
$2r^{(3)}$	1	0 0 0 1			
$+(-2^4 D)$	+ 1	0 1 1 0			
$r^{(4)}$	0	0 1 1 1		Positive, so set $q_0 = 1$	
R			0 1 1 1	$r^{(k)} = 2^k R$	
Q			1 0 1 1		

División con restauración: ejemplo

❖ Ejemplo: Supongamos $X = 0.100000_{(2} = \frac{1}{2}$ y $D = 0.110_{(2} = \frac{3}{4}$

X_f	`0. 1 0 0 │ 0 0 0`	No overflow, since:
D_f	`0. │ 1 1 0`	$0.1_{(2} < 0.11_{(2}$
$-D_f$	`1. │ 0 1 0`	
$r^{(0)} = X_f$	`0. 1 0 0 │ 0 0 0`	
$2r^{(0)}$	`1. │ 0 0 0 │ 0 0`	
⇨ Add $(-D_f)$	`+ 1. │ 0 1 0`	
$r^{(1)} = 2r^{(0)} - D_f$	`0. 0 1 0 │ 0 0`	Positive, so set $q_{-1} = 1$
$2r^{(1)}$	`0. 1 0 0 │ 0`	
$+(-D_f)$	`+ 1. 0 1 0`	
$r^{(2)} = 2r^{(1)} - D_f$	`1. 1 1 0 │ 0`	Negative, so set $q_{-2} = 0$
$r^{(2)} = 2r^{(1)}$	`0. 1 0 0 │ 0`	and restore
$2r^{(2)}$	`1. │ 0 0 0`	
$+(-D_f)$	`+ 1. │ 0 1 0`	
$r^{(3)}$	`0. 0 1 0`	Positive, so set $q_{-3} = 1$
R_f	`. 0 0 0 │ 0 1 0`	$r^{(k)} = 2^k R_f$
Q_f	`. 1 0 1`	

- Resto: $R_f = r^{(3)} 2^{-3} = \frac{1}{4} 2^{-3} = \frac{1}{32}$
- Comprobación:

$$X_f = Q_f \cdot D_f + R_f = \frac{5}{8} \cdot \frac{3}{4} + \frac{1}{32} = \frac{16}{32} = \frac{1}{2}$$

División con restauración

- Hasta ahora, hemos supuesto operandos sin signo.
  Para operandos con signo, la expresión básica

  $$X = Q \cdot D + R \quad \text{junto con} \quad \text{sign}(R) = \text{sign}(X) \quad \text{y} \quad |R| < |D|$$

  definen unívocamente tanto el cociente Q como el resto R.
- Ejemplos:
  - $X = 5$ $D = 3$ ⇨ $Q = 1$ $R = 2$
  - $X = 5$ $D = -3$ ⇨ $Q = -1$ $R = 2$ (no $Q = -2$, $R = -1$)
  - $X = -5$ $D = 3$ ⇨ $Q = -1$ $R = -2$
  - $X = -5$ $D = -3$ ⇨ $Q = 1$ $R = -2$
- A partir de estos ejemplos vemos que las magnitudes de Q y R no se ven alteradas por los signos de las entradas y que sus signos pueden derivarse a partir de los de X y D.

  ➔ De aquí que una manera de hacer división entre operandos con signo consiste en convertir los operandos en valores sin signo, calcular la división, y ajustar los signos.

  ➔ Este es el método preferente cuando se usa un algoritmo de división con restauración.

Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
- **División**
  - Operación y técnicas básicas
  - División con restauración
  - **División sin restauración**
- Generadores de función
- Ejercicios resueltos

División sin restauración

- La implementación de la división con restauración requiere prestar atención a la temporización de varios eventos:
  - → Cada uno de los k ciclos debe ser lo suficientemente largo para permitir:
    - Desplazamiento de los registros
    - Propagación de las señales a través del sumador
    - Almacenamiento del dígito del cociente
    - Almacenamiento de la diferencia tentativa, si es el caso.
- Un esquema alterativo es la división sin restauración (*non restoring division*).
- Como antes, suponemos $q_{k-j} = 1$ y realizamos la resta, almacenando la diferencia en el registro de restos parciales (registro *PR*).
  - ⇨ esto conduce a un resto parcial temporalmente incorrecto (→ sin restauración)

※ ¿Por qué es aceptable almacenar un valor incorrecto en el registro *PR*?

División sin restauración

- ¿Por qué es aceptable almacenar un valor incorrecto en el registro *PR*?
  - Supongamos que, al comienzo del ciclo, el registro *PR* tiene almacenado el valor u.
  - Si hemos restaurado el resto parcial $(u - 2^k D)$ a su valor correcto u, deberíamos continuar con el desplazamiento de u a $2u$ y la nueva resta $2u - 2^k D$.
  - Si en vez de hacer esto hubiéramos usado el resto parcial incorrecto, el desplazamiento y la resta posterior conduciría a $2(u - 2^k D) - 2^k D = 2u - 3 \cdot 2^k D$, que no es el resultado previsto, pero que puede repararse si añadimos $2^k D$ (en vez de restarlo):
    - $2(u - 2^k D) + 2^k D = 2u - 2^k D$, que es el valor obtenido después de restauración y resta

⇨ En la división sin restauración, si el resto parcial es negativo,
- se mantiene dicho resto incorrecto
- se corrige el dígito correspondiente
- en el ciclo siguiente se suma en vez de restar.

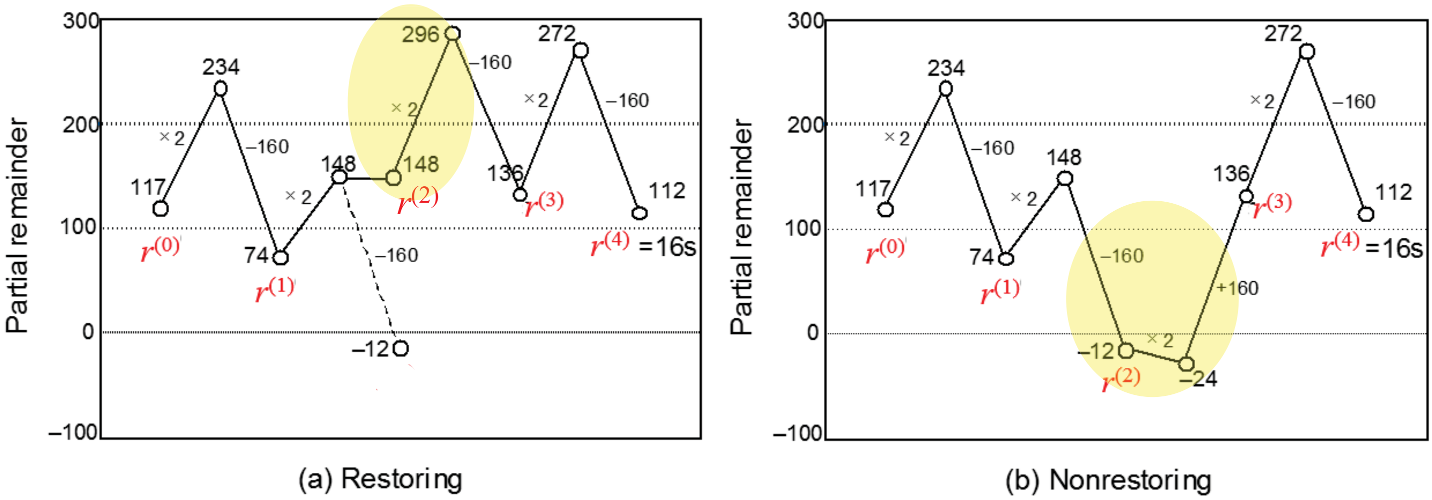
División sin restauración: ejemplo

❖ Ejemplo: Supongamos $X = 117_{(10} = 01110101_{(2}$ y $D = 10_{(10} = 1010_{(2}$

X			0 1 1 1	0 1 0 1	117	No overflow, since:
2^4D		0	1 0 1 0			$0111_{(2} < 1010_{(2}$
-2^4D		1	0 1 1 0			
$r^{(0)} = X$			0 1 1 1	0 1 0 1		
$2r^{(0)}$		0	1 1 1 0	1 0 1		Positive,
⇨ Add (-2^4D)	+	1	0 1 1 0			so substract
$r^{(1)} = 2r^{(0)} - 2^4D$		0	0 1 0 0	1 0 1		
$2r^{(1)}$		0	1 0 0 1	0 1		Positive, so set $q_3 = 1$
$+(-2^4D)$	+	1	0 1 1 0			and substract
$r^{(2)} = 2r^{(1)} - 2^4D$		1	1 1 1 1	0 1		
$2r^{(2)}$		1	1 1 1 0	1		Negative, so set $q_2 = 0$
$+(2^4D)$	+	0	1 0 1 0			and add
$r^{(3)} = 2r^{(2)} + 2^4D$		0	1 0 0 0	1		
$2r^{(3)}$	0	1	0 0 0 1			Positive, so set $q_1 = 1$
$+(-2^4D)$	+	1	0 1 1 0			and substract
$r^{(4)}$		0	0 1 1 1			Positive, so set $q_0 = 1$
R				0 1 1 1	7	$r^{(k)} = 2^k R$
Q				1 0 1 1	11	

División sin restauración: ejemplo

- Ejemplo: $X = \texttt{01110101}_{(2} = 117_{(10}$ y $D = \texttt{1010}_{(2} = 10_{(10}$

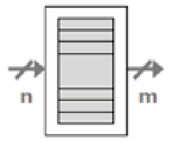


Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
- División
- Generadores de función
- Ejercicios resueltos

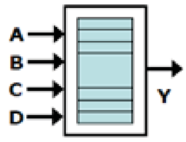
Generación de funciones: *LookUp Table* (LUT)

- Una LUT de n bits puede codificar cualquier función Booleana modelando dicha función con su tabla de verdad.



LUT genérica

- LUTs con 4-6 bits de entrada son componentes clave en las actuales FPGAs.



LUT en FPGAs

- Un ejemplo típico de reducción de tiempo de cómputo usando LUTs es el del cálculo trigonométrico. Este cálculo, sustancialmente lento, puede acelerarse notablemente cuando se almacenan valores ya pre-calculados de la función en una LUT. Cuando se tiene que calcular un valor de la función, se recupera de la LUT el valor más cercano.

Ejemplo: función XOR2

Tabla de verdad

A	B	C	D	Y
0	0	X	X	0
0	1	X	X	1
1	0	X	X	1
1	1	X	X	0

⇨

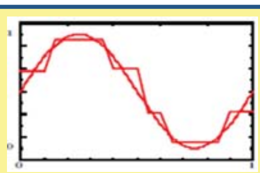
LUT

Input (ABCD)	Output (Y)
00XX	0
01XX	1
10XX	1
11XX	0

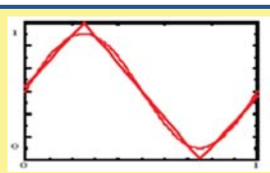
Generación de funciones: Interpolación

- Solución intermedia que combina memoria (LUTs) con cálculo.
- El pre-cálculo combinado con la interpolación permite computar con mayor precisión los valores que se encuentran entre valores pre-calculados.
- Esta técnica requiere un tiempo de cómputo ligeramente superior, pero con la ventaja de incrementar considerablemente la precisión.
- El pre-cálculo combinado con la interpolación también puede usarse para disminuir el tamaño de la LUT sin perder precisión.
- Una solución de amplio uso es la interpolación lineal (PWA, *PieceWise Affine*), que calcula una línea entre los dos valores precalculados de la tabla y sitúa en dicha línea la solución. Es rápido de calcular y mucho más preciso para generar funciones suaves.

Interpolando una sinusoide



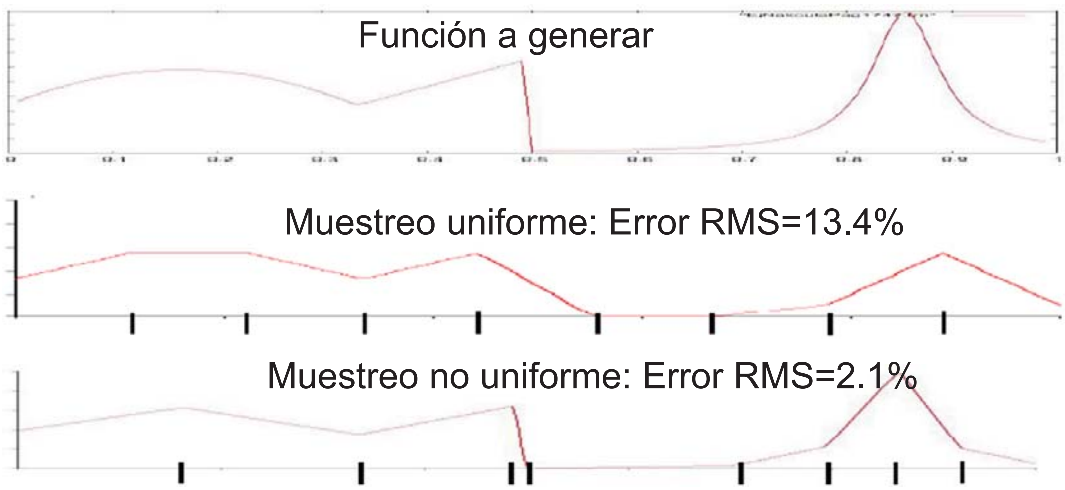
LUT sin interpolación



LUT con interpolación lineal (sumadores y multiplicadores)

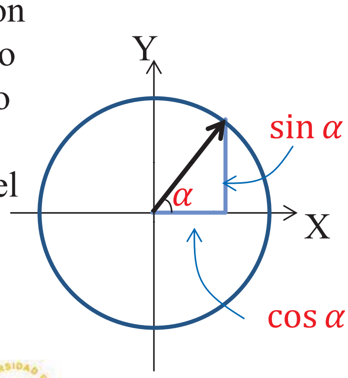
Generación de funciones: Interpolación

- Muchas veces conviene emplear muestreo no uniforme para interpolar:
  - → cuando la función se comporta de forma lineal se emplean pocos puntos de muestra,
  - → cuando la función cambia de valor bruscamente, se emplean más puntos para aproximarla.



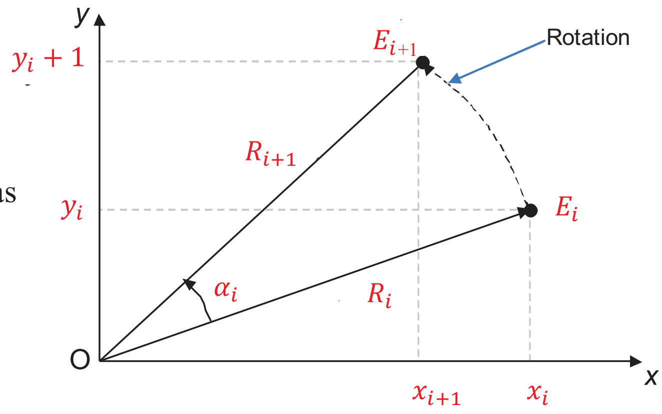
Generación de funciones: Algoritmo CORDIC

- El algoritmo CORDIC (*COordinate Rotation DIgital Computer*), conocido como método de dígito por dígito, o como algoritmo de Volder, fue descrito por primera vez en 1959 por Jack E. Volder para funciones trigonométricas.
- John Stephen Walther, en Hewlett-Packard, generalizó el algoritmo, permitiendo calcular funciones hiperbólicas, exponenciales, logaritmos, multiplicación, división, y la raíz cuadrada.
- Fue propuesto para el cálculo de funciones de una forma muy simple, rápida y precisa, sin requerir una unidad de multiplicación.
- En su forma más simple CORDIC está basado en la observación de que si se rota un ángulo α un vector de longitud unidad cuyo extremo se encuentra, inicialmente, en $(x, y) = (1,0)$, el nuevo extremo se encuentra en $(x, y) = (\cos\alpha, \sin\alpha)$, por lo que $\cos\alpha$ y $\sin\alpha$ podrían calcularse obteniendo las coordenadas del nuevo extremo después de la rotación α.



Generación de funciones: Algoritmo CORDIC

Rotaciones



- Consideremos el vector OE_i, de coordenadas (x_i, y_i).
- Si rotamos OE_i, un ángulo α_i, las coordenadas del nuevo extremo OE_{i+1}, serán:

$$\boxed{\begin{array}{l} \begin{cases} x_{i+1} = x_i \cdot \cos\alpha_i - y_i \cdot \sin\alpha_i \\ y_{i+1} = y_i \cdot \cos\alpha_i + x_i \cdot \sin\alpha_i \end{cases} \\ z_{i+1} = z_i - \alpha_i \end{array}}$$

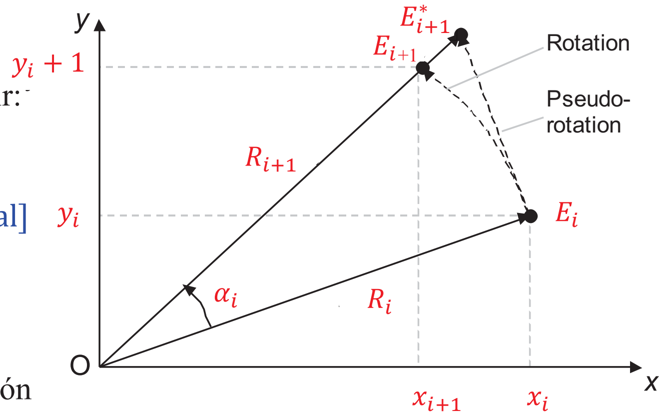
donde z permite guardar registro de la rotación total realizada en los diferentes pasos.

→ Si z_0 es la rotación objetivo y si los ángulos α_i se seleccionan en cada paso de manera que z_m tienda a 0, entonces el punto E_m, de coordenadas (x_m, y_m), será el extremo del vector después de haberlo rotado un ángulo z_0.

→ z_i puede verse como la rotación residual que aún debe hacerse y así, z_{i+1} es la versión actualizada de z_i después de que se haya hecho la rotación α_i.

Generación de funciones: Algoritmo CORDIC

Pseudo-rotaciones



- Las ecuaciones anteriores se pueden reescribir:

$$\begin{cases} x_{i+1} = \dfrac{x_i - y_i \cdot \tan\alpha_i}{(1+\tan^2\alpha_i)^{1/2}} \\ y_{i+1} = \dfrac{y_i + x_i \cdot \tan\alpha_i}{(1+\tan^2\alpha_i)^{1/2}} \end{cases} \quad \text{[Rotación real]}$$

$$z_{i+1} = z_i - \alpha_i$$

- En el algoritmo CORDIC, los pasos de rotación se substituyen por pseudo-rotaciones:

$$\begin{cases} x_{i+1} = x_i - y_i \cdot \tan\alpha_i \\ y_{i+1} = y_i + x_i \cdot \tan\alpha_i \end{cases} \quad \text{[Pseudo-rotación]}$$

$$z_{i+1} = z_i - \alpha_i$$

→ En tanto que una rotación real no cambia la longitud R_i del vector, un paso de pseudo-rotación lo incrementa a $R_{i+1} = R_i(1+\tan^2\alpha_i)^{1/2}$.

→ Las coordenadas del nuevo extremo, E^*_{i+1}, después de una pseudo-rotación se obtienen multiplicando las coordenadas de E_{i+1} por el factor $(1+\tan^2\alpha_i)^{1/2}$

Generación de funciones: Algoritmo CORDIC

Pseudo-rotaciones

- Suponiendo ahora $x_0 = x$, $y_0 = y$ y $z_0 = z$, después de m rotaciones reales con ángulos $\alpha_1, \alpha_2, \ldots, \alpha_m$, tendremos:

$$\begin{cases} x_m = \left(x \cdot \cos\left(\sum \alpha_i\right) - y \cdot \sin\left(\sum \alpha_i\right)\right) \prod (1 + \tan^2 \alpha_i)^{1/2} \\ y_m = \left(y \cdot \cos\left(\sum \alpha_i\right) + x \cdot \sin\left(\sum \alpha_i\right)\right) \prod (1 + \tan^2 \alpha_i)^{1/2} \end{cases}$$

$$z_m = z - \left(\sum \alpha_i\right)$$

- El factor de expansión $K = \prod (1 + \tan^2 \alpha_i)^{1/2}$ depende de los ángulos de rotación $\alpha_1, \alpha_2, \ldots, \alpha_m$. No obstante, si siempre rotamos los mismos ángulos, con signos positivos o negativos, la constante K puede evaluarse previamente.
- En este caso, el efecto de usar las pseudo-rotaciones, mucho más simples que las rotaciones, es sólo el escalado del vector de coordenadas y de la longitud por una constante conocida.

Generación de funciones: Algoritmo CORDIC

Pseudo-rotaciones

- ¿Qué pasaría si los ángulos en que puede rotar el vector se restringen de tal manera que $\tan \alpha_i = \pm 2^{-i}$?

$$\begin{cases} x_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i} \\ y_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i} \\ z_{i+1} = z_i - d_i \cdot \tan^{-1}\left(2^{-i}\right) \end{cases}$$ [Iteración CORDIC]

- La evaluación de x_{i+1} o de y_{i+1}, requiere un desplazamiento de i-bits a la derecha y una suma/resta.
- Si se evalúa previamente la función $e_i = \tan^{-1}\left(2^{-i}\right)$, para diferentes valores de i, y se almacena en una tabla, sólo se precisa una simple adición/substracción para evaluar z_{i+1}.
- Si se hacen las pseudo-rotaciones usando ángulos de un mismo conjunto de ángulos (sumando o restando), entonces K es una constante que se puede calcular previamente.
  - ⇨ Ejemplo de pseudo-rotación 30º:

    30.0 ≈ 45.0 – 26.6 + 14.0 – 7.1 + 3.6 + 1.8 – 0.9 + 0.4 – 0.2 – 0.1 = 29.9291

i	e_i
0	45.0000
1	26.5651
2	14.0362
3	7.1250
4	3.5763
5	1.7899
6	0.8951
7	0.4476
8	0.2238
9	0.1119

Generación de funciones: Algoritmo CORDIC

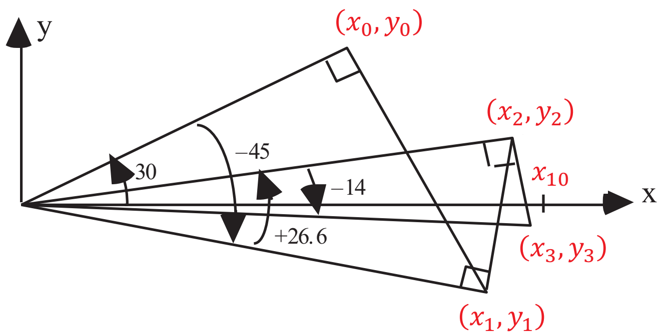
- Pseudo-rotaciones
  - Ejemplo de pseudo-rotación 30º:

    $30.0 \approx 45.0 - 26.6 + 14.0 - 7.1 + 3.6 + 1.8 - 0.9 + 0.4 - 0.2 - 0.1 = 29.9291$

    - En CORDIC, la rotación z se inicializa a 30º y luego, en cada paso, el signo del siguiente ángulo se selecciona intentando cambiar el signo de z.

      ⇨ Es decir, $d_i = \text{sign}(z_i)$, donde la función signo se define como -1 o 1 dependiendo de que el argumento sea negativo o no.

i	e_i
0	45.0000
1	26.5651
2	14.0362
3	7.1250
4	3.5763
5	1.7899
6	0.8951
7	0.4476
8	0.2238
9	0.1119



i	z_i	−	α_i	=	z_{i+1}
0	+30.0	−	45.0	=	−15.0
1	−15.0	+	26.5651	=	+11.5651
2	+11. 5651	−	14.0362	=	−2.4712
3	−2.4712	+	7.1250	=	+4.6538
4	+4.6538	−	3.5763	=	+1.0775
5	+1.0775	−	1.7899	=	−0.7124
6	−0.7124	+	0.8951	=	+0.1827
7	+0.1827	−	0.4476	=	−0.2648
8	−0.2648	+	0.2238	=	−0.0410
9	−0.0410		0.1119	=	+0. 0708

Generación de funciones: Algoritmo CORDIC

- En la terminología empleada en CORDIC, la regla de selección que acabamos de usar para d_i, que hace que z converja a 0, se conoce como **modo de rotación**.
- Podemos reescribir las iteraciones de la forma siguiente, donde $e_i = \tan^{-1}(2^{-i})$

$$\begin{cases} x_{i+1} = x_i - d_i(2^{-i} y_i) \\ y_{i+1} = y_i + d_i(2^{-i} x_i) \\ z_{i+1} = z_{i+1} - d_i e_i \end{cases}$$

- Después de m iteraciones en modo de rotación, cuando z_m está lo suficientemente próximo a 0, tenemos que $\sum \alpha_i = z$, y las ecuaciones para las pseudo-rotaciones:

$$\begin{cases} x_m = K(x \cos z - y \sin z) \\ y_m = K(y \cos z + x \sin z) \\ z_m = 0 \end{cases} \qquad \text{[Modo de rotación]}$$

con $d_i \in \{-1,1\}$, de forma que $z \to 0$ y K =1.64676.

Generación de funciones: Algoritmo CORDIC

Pseudo-rotaciones

- Para evaluar $\cos z$ y $\sin z$, se comienza con $z = 1/K = 0.607252$ e $y = 0$.
- Las iteraciones del algoritmo de CORDIC en modo rotación fuerzan que $z_m \to 0$, con lo que x_m e y_m convergen a $\cos z$ y $\sin z$, respectivamente.
- Una vez obtenidos el seno y el coseno, la tangente puede obtenerse mediante división.

i	xi	yi	zi	di
0	0,6073	0,0000	30,0000	1
1	0,6073	0,6073	-15,0000	-1
2	0,9109	0,3036	11,5651	1
3	0,8350	0,5313	-2,4711	-1
4	0,9014	0,4270	4,6539	1
5	0,8747	0,4833	1,0776	1
6	0,8596	0,5106	-0,7123	-1
7	0,8676	0,4972	0,1828	1
8	0,8637	0,5040	-0,2648	-1
9	0,8657	0,5006	-0,0410	-1
10	0,8666	0,4989	0,0709	
15	0,8660	0,5000	-0,0025	

$$\sin 30 = \frac{1}{2} = 0.5$$

$$\cos 30 = \frac{\sqrt{3}}{2} = 0.8660$$

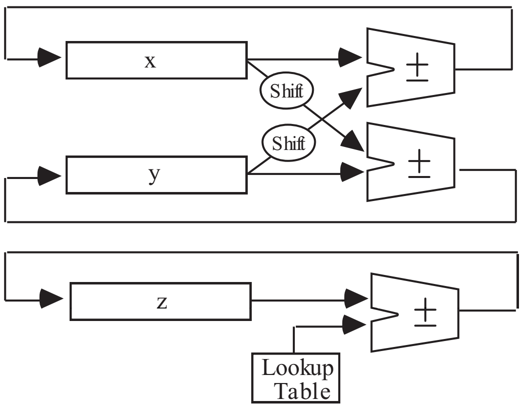
i	xi	yi	zi	di
0	0,6073	0,0000	60,0000	1
1	0,6073	0,6073	15,0000	1
2	0,3036	0,9109	-11,5651	-1
3	0,5313	0,8350	2,4711	1
4	0,4270	0,9014	-4,6539	-1
5	0,4833	0,8747	-1,0776	-1
6	0,5106	0,8596	0,7123	1
7	0,4972	0,8676	-0,1828	-1
8	0,5040	0,8637	0,2648	1
9	0,5006	0,8657	0,0410	1
10	0,4989	0,8666	-0,0709	
15	0,5000	0,8660	0,0025	

con 15 iteraciones:

Generación de funciones: Hardware CORDIC

Las iteraciones

$$\begin{cases} x_{i+1} = x_i - d_i(2^{-i}y_i) \\ y_{i+1} = y_i + d_i(2^{-i}x_i) \\ z_{i+1} = z_{i+1} - d_i e_i \end{cases}$$



Requiere.

- tres registros para x, y y z.
- una tabla *lookup* para almacenar los valores de $e_i = \tan^{-1} 2^{-i}$ y
- dos desplazadores que suministren los términos $2^{-i}x$ y $2^{-i}y$ a las unidades de adición/substracción

Tema 6. Multiplicación, división y generación de funciones

- Multiplicación
- División
- Generadores de función
- Ejercicios resueltos

Multiplicación

➔ Ejercicio 1: Realice la siguiente operación en binario `110001×101` usando seis bits.

Solución:

Dado que nos dicen que son seis los bits, eso significa que el multiplicando (*A*) es negativo, y no hay que tener especial cuidado.

Se trata del `110001` → `001110+000001 = 001111` → `15`. Por tanto, el primer número es `-15`.

El segundo número es `000101` → `5`. El producto a calcular debe ser `(-15)×5`.

			1	1	0	0	0	1
					×	1	0	1
	1	1	1	1	0	0	0	1
	1	1	0	0	0	1		
1	1	0	1	1	0	1	0	1

El resultado es `110110101`. Para comprobar que es correcto, invertimos y sumamos 1: `001001010+1` → `001001011` → $2^6+2^3+2^1+2^0$ = 64+8+2+1 = 75

Multiplicación

➔ Ejercicio 2: Realice la siguiente operación en binario `101×110001` usando seis bits.

Solución:

Dado que nos dicen que son seis los bits, eso significa que el multiplicando (*A*) es positivo y el multiplicador (*X*) es negativo. El producto a calcular debe ser `5×(-15)`.

El tratamiento es ligeramente diferente al caso anterior ya que el bit que indica el signo no puede usarse igual que el resto.

El complemento a 2 de un número satisface: $X = -x_{n-1}2^{n-1} + \underbrace{\sum_{j=0}^{n-2} x_j 2^j}_{\tilde{X}}$

En nuestro caso, $-2^5 + (2^4 + 2^0) = -32 + 17 = -15$

Si ignoramos el bit de signo, el resultado de la multiplicación sería

$\tilde{X} \cdot A = (X + x_{n-1}2^{n-1}) \cdot A = X \cdot A + A \cdot x_{n-1}2^{n-1}$

El resultado que buscamos es $X\ A$. Es decir, debemos corregir $\tilde{X}\ A$ restándole $A\ x_{n-1}2^{n-1}$ (sumando su Ca2)

							1	0	1
			×	1	1	0	0	0	1
		0	0	0	0	0	1	0	1
		0	1	0	1				
		0	1	0	1	0	1	0	1
Corrección	1	0	1	1					
	1	1	0	1	1	0	1	0	1

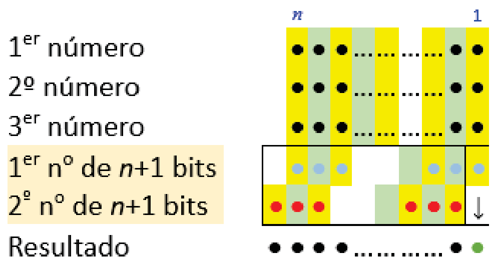
El resultado es `110110101`. Para comprobar que es correcto, invertimos y sumamos 1:
`001001010+1` → `001001011` → $2^6+2^3+2^1+2^0$ = 64+8+2+1 = 75

Suma con contadores

➔ Ejercicio 3: Se pretende sumar 3 números de *n* bits para lo que se dispone de hasta *n* contadores (3,2) cuyo retraso es Δ_1, y de un único sumador para (dos) números de *n* bits, cuyo retraso, Δ_S, es proporcional al logaritmo de la longitud, *n*, de los números que íbamos originalmente a sumar ($\Delta_S = K \ln n$).
Explique y dibuje un esquema de cómo podría construirse un sumador para resolver el problema usando los elementos de los que dispone. ¿Cuántos contadores de cada tipo se necesitan? ¿Cuál es el retraso de la solución propuesta?

Solución:

Usamos n contadores (3,2) que proporcionan 2 números de n+1 bits para sumar los 3 números de n bits, pero no es necesario tener en cuenta el primer bit del primer número. Así sólo se necesita un sumador para dos números de n bits.



Retraso para sumar los tres números de *n* bits: Δ_1, ya que los bits se suman en paralelo.

Retraso para sumar los dos números de n bits finales: Δ_S.

Retraso total: $\Delta_S + \Delta_1$.

Multiplicación

➔ Ejercicio 4: Realice la multiplicación $-4.875_{(10}\times 7.625_{(10}$ en binario aplicando el algoritmo de Booth.

Solución:

Expresamos los números en binario usando un número par de bits, 8 en este caso.

A es -4.875,

-A: 4.875 → $0100.1110_{(2}$ 1011.0001 A → 1011.0010

7.625 → $0111.1010_{(2}$ X → 0111.1010

Recod.: Y → $1000.\bar{1}1\bar{1}0$

Multiplicación:

A 10110010

X × 01111010

Y $1000\bar{1}1\bar{1}0$

101101011010100 es el producto sin comas decimales.

El producto real será: 1011010.11010100 que es un número negativo en complemento a 2. Por tanto,

-P = 0100101.00101011 + 0000000.00000001

-P = 0100101.00101100 = $2^5+2^2+2^0+2^{-3}+2^{-5}+2^{-6}$ = 37.171875

		00000000
$1000\bar{1}1\bar{1}0$	Shift	000000000
$1000\bar{1}1\bar{1}0$	-A	01001110
		010011100
	Shift	0010011100
$1000\bar{1}1\bar{1}0$	+A	10110010
		1101100100
	Shift	11101100100
$1000\bar{1}1\bar{1}0$	-A	01001110
		100111010100
	Shift	000111010100
$1000\bar{1}1\bar{1}0$	Shift	0000111010100
$1000\bar{1}1\bar{1}0$	Shift	00000111010100
$1000\bar{1}1\bar{1}0$	Shift	000000111010100
$1000\bar{1}1\bar{1}0$	+A	10110010
		101101011010100

Multiplicación

➔ Ejercicio 4 (cont.): Realice la multiplicación $-4.875_{(10}\times 7.625_{(10}$ en binario aplicando el algoritmo de Booth.

Solución:

-A: 4.875 → $0100.1110_{(2}$

										1	0	1	1	0	0	1	0
									×	1	0	0	0	1	1	1	0
$\bar{1}$: Sumo −A	-A		0	0	0	0	0	0	0	1	0	0	1	1	1	0	0
1: Sumo A	A		1	1	1	1	1	1	0	1	1	0	0	1	0		
$\bar{1}$: Sumo −A	-A		0	0	0	0	0	1	0	0	1	1	1	0			
1: Sumo A	A		1	0	1	1	0	0	1	0							
A es negativo		1	1	0	1	1	0	1	0	1	1	0	1	0	1	0	0

Multip. por el 1er cero

Resultado con 15 bits

Reescritura del número según su signo

Multiplicación

➔ Ejercicio 5: Realice la multiplicación $23.375_{(10}\times(-8.75)_{(10}$ en binario aplicando el algoritmo de Booth.

Solución:

Expresamos los números en binario usando un número par de bits, 10 en este caso.

A:23.75 → $0010111.011_{(2}$ -A → 1101000.101

8.75 → $00001000.11_{(2}$ 11110111.00 X → 11110111.01

Rec.: Y → $000\bar{1}100\bar{1}.1\bar{1}$

Multiplicación:

A 0010111011

X × 1111011101

Y $000\bar{1}100\bar{1}1\bar{1}$

11111110011001101111: producto sin comas decimales.

El producto real será: 111111100110011.01111 que es un número negativo en complemento a 2. Por tanto,

-P = 000000011001100.10000 + 000000000000000.00001

-P = 000000011001100.10001 = $2^7+2^6+2^3+2^2+2^{-1}+2^{-5}$ = 204.53125

$000\bar{1}100\bar{1}1\bar{1}$ -A 1101000101

1101000101

Shift 11101000101

$000\bar{1}100\bar{1}1\bar{1}$ +A 0010111011

100010111011

Shift 000010111011

$000\bar{1}100\bar{1}1\bar{1}$ -A 1101000101

110111001111

Shift 1110111001111

$000\bar{1}100\bar{1}1\bar{1}$ Shift 11110111001111

$000\bar{1}100\bar{1}1\bar{1}$ Shift 111110111001111

$000\bar{1}100\bar{1}1\bar{1}$ +A 0010111011

1001010100101111

Shift 0001010100101111

$000\bar{1}100\bar{1}1\bar{1}$ -A 1101000101

1110011001101111

Shift 11110011001101111

$000\bar{1}100\bar{1}1\bar{1}$ Shift 111110011001101111

$000\bar{1}100\bar{1}1\bar{1}$ Shift 1111110011001101111

$000\bar{1}100\bar{1}1\bar{1}$ Shift 11111110011001101111

Multiplicación

➔ Ejercicio 5 (cont.): Realice la multiplicación $23.375_{(10}\times(-8.75)_{(10}$ en binario aplicando el algoritmo de Booth.

Solución:

Multiplicación:

A 0010111011

Y × $000\bar{1}100\bar{1}1\bar{1}$

Reescritura del número según su signo

$\bar{1}$: Sumo $-A$

1: Sumo A

$\bar{1}$: Sumo $-A$

1: Sumo A

$\bar{1}$: Sumo $-A$

											0	0	1	0	1	1	1	0	1	1
										×	0	0	0	1	1	0	0	1	1	1
-A		1	1	1	1	1	1	1	1	1	1	1	0	1	0	0	0	1	0	1
A		0	0	0	0	0	0	0	0	0	0	1	0	1	1	1	0	1	1	
-A		1	1	1	1	1	1	1	1	1	0	1	0	0	0	1	0	1		
A		0	0	0	0	0	0	1	0	1	1	1	0	1	1					
-A		1	1	1	1	1	0	1	0	0	0	1	0	1						
		1	1	1	1	1	1	0	0	1	1	0	0	1	1	0	1	1	1	1

A es positivo

Debidos a los tres últimos ceros

Resultado con 15 bits

Multiplicación

➔ Ejercicio 6: Considere el producto de 17 por –9. (Ej. 1 pág. 26 – Booth modificado)

Solución:

Multiplicación:

A   01 00 01

Y × $0\bar{1}$ 10 $0\bar{1}$

Reescritura del número según su signo

						0	1	0	0	0	1
					×	0	$\bar{1}$	1	0	0	$\bar{1}$
$0\bar{1}$: Sumo $-A$	-A		1	1	1	1	1	0	1	1	1	1
10: Sumo $2A$	2A			0	1	0	0	0	1	X		
$0\bar{1}$: Sumo $-A$	-A		1	0	1	1	1	1				
A es positivo			1	1	0	1	1	0	0	1	1	1

- Como se multiplica por bloques de 2 bits, el desplazamiento es de dos casillas.
- La X marca el hecho de que al desplazamiento se le añade la multiplicación por 2.
- El último sumando entra en su posición

Resultado con 15 bits

División

➔ Ejercicio 7: Realice la división binaria de los números X = `0.100000`$_{(2}$ = ½ y D = `0.110`$_{(2}$ = ¾ .

Solución:

$r^{(0)} = X$	0 .1 0 0 | 0 0 0	
$2r^{(0)}$	0 1 .0 0 0 | 0 0	$q_{-1} = 1$
Sumo $-D$	+ 1 1 .0 1 0	
$r^{(1)} = 2r^{(0)} - D$	0 0 .0 1 0 | 0 0 0	
$2r^{(1)}$	0 0 .1 0 0 | 0 0	$q_{-2} = 0$
$r^{(2)} = 2r^{(1)}$	0 0 .1 0 0 | 0 0	
$2r^{(2)}$	0 1 .0 0 0 | 0	$q_{-3} = 1$
Sumo $-D$	+ 1 1 .0 1 0	
$r^{(3)} = 2r^{(2)} - D$	0 0 .0 1 0 | 0 0 0	

D = 0.110_2; $-D$ = 1.010_2

- El dividendo ocupa un registro de longitud doble.
- Se satisface que $X < D$
- $2r^{(0)}$ no debería proporcionar una indicación de *overflow* (es un número positivo), de ahí la necesidad de un bit extra.
- En todo momento comparamos $2r^{(i-1)}$ con D para determinar q_{-i}.
- El resultado final es $Q = (0.101)_2 = \frac{5}{8}$, y
$$R = r^{(m)}2^{-m} = r^{(3)}2^{-3} = \frac{1}{4}2^{-3} = \frac{1}{32}$$
- El cociente y el resto final satisfacen
$$X = Q \cdot D + R = \frac{5}{8} \times \frac{3}{4} + \frac{1}{32}$$

División

➔ Ejercicio 8: Realice la división binaria de $X = 5.40625_{(10}$ con $D = 3.25_{(10}$.

Solución: $X = 101.01101_{(2}$ $D = 11.01_{(2}$

$r^{(0)} = X$	0.1010|1101	
$2r^{(0)}$	01.0101|101	$q_{-1} = 1$
Sumo $-D$	+11.0011	
$r^{(1)} = 2r^{(0)} - D$	00.1000|101	
$2r^{(1)}$	01.0001|01	$q_{-2} = 1$
Sumo $-D$	+11.0011	
$r^{(2)} = 2r^{(1)} - D$	00.0100|01	
$2r^{(2)}$	00.1000|1	$q_{-3} = 1$
Sumo $-D$	+11.0011	
$r^{(3)} = 2r^{(2)} - D$	11.1011|1	< 0, luego $q_{-3} = 0$
Correción	+00.1101	
$r^{(3)} = 2r^{(2)}$	00.1000|1	
$2r^{(3)}$	01.0001	$q_{-4} = 1$
Sumo $-D$	+11.0011	
	00.0100	

- El dividendo ocupa un registro de longitud doble.
- Se satisface que $X < D$
- $2r^{(0)}$ no debería proporcionar una indicación de *overflow* (es un número positivo), de ahí la necesidad de un bit extra (sombreado).
- En todo momento comparamos $2r^{(i-1)}$ con D para determinar q_{-i}.

- El problema está hecho de forma que se divide 0.10101101 entre 0.1101. El resultado final cumple: $X^* = Q \cdot D^* + R \cdot 2^{-4}$.
  La operación es $X \cdot 2^{-3} = Q \cdot D \cdot 2^{-2} + R \cdot 2^{-4}$
  $X = Q \cdot D \cdot 2^{1} + R \cdot 2^{-1}$, y
  De aquí,
  el cociente real es: $2Q \Longrightarrow 1.101$
  El resto real es: $R/2 \Longrightarrow 0.001$
- El cociente y el resto final satisfacen
  $$X = 1.101 \cdot 11.01 + 0.001$$
  $$= \left(1 + \frac{5}{8}\right) \times \left(3 + \frac{1}{4}\right) + \frac{1}{8} = 5 + \frac{13}{32}$$