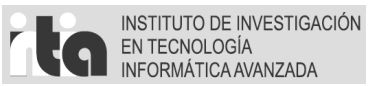
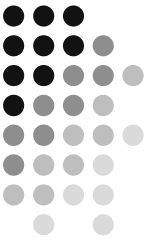
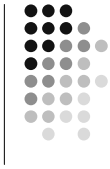
Multiplicación

Martín Vázquez
Arquitectura I - Curso 2013
UNICEN



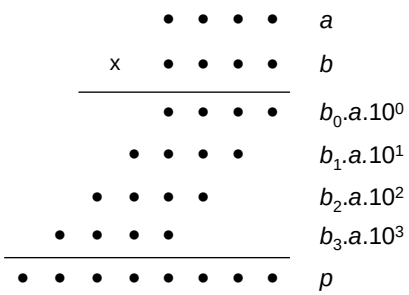


Multiplicación

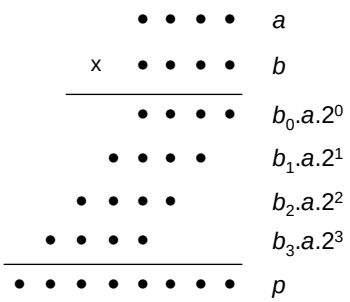


Notación *dot*

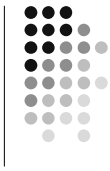
Multiplicación decimal (base 10)



Multiplicación binaria (base 2)



Multiplicación

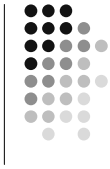


- Independientemente de la base (2, 10, u otra), es un algoritmo clásico de desplazamiento y suma (*shift and add*)
- Para base 2 y multiplicador de *m* bits:

$$p = \sum_{j=0}^{m-1} a.b_j.2^j$$

- Multiplicar un número por la base elevada a un exponente, es desplazarlo a la izquierda (*shift right*) tantos lugares como indica el exponente y completar con 0's
  - $13_{10}.10^4 = 130000_{10}$; $101_2.2^3 = 101000_2$

Multiplicación binaria



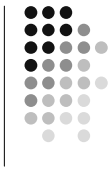
- Ejemplo

Para multiplicando *a* = 1010_2, y multiplicador *b* = 1101_2

$$p = a.b_3.2^3 + a.b_2.2^2 + a.b_1.2^1 + a.b_0$$

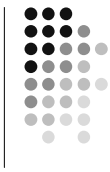
$$p = 1010000_2 + 101000_2 + 00000 + 1010 = 10000010_2 \ (130_{10})$$

Multiplicación binaria



- Dos estrategias principales para abordar la implementación hardware de los algoritmos de multiplicación *shift and add*
  - *Desplazamiento a la izquierda (shift left)*
  - *Desplazamiento a la derecha (shift right)*
- En cada paso se suma el operando multiplicando, al resultado parcial desplazado (a izquierda o derecha)

Multiplicación binaria



Algoritmo *shift left*

- Sea $p = a.b$, con b de 4 bits, entonces

$$p = a.b_3.2^3 + a.b_2.2^2 + a.b_1.2^1 + a.b_0$$

$$p = (((a.b_3).2 + a.b_2).2 + a.b_1).2 + a.b_0 \quad \text{(Expansión de Hörner)}$$

Multiplicación binaria

Algoritmo *shift left*

- Sea $p = a.b$, con b de 4 bits, entonces

$$p = a.b_3.2^3 + a.b_2.2^2 + a.b_1.2^1 + a.b_0$$

$$p = ((a.b_3).2 + a.b_2).2 + a.b_1).2 + a.b_0 \quad \text{(Expansión de Hörner)}$$

$$p^{(1)} = p^{(0)}.2 + a.b_3$$
$$p^{(2)} = p^{(1)}.2 + a.b_2$$
$$p^{(3)} = p^{(2)}.2 + a.b_1$$
$$p^{(4)} = p^{(3)}.2 + a.b_0$$

- El algoritmo consta de 5 pasos, $p^{(0)}$ se inicializa en 0

Multiplicación binaria

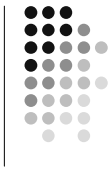
Algoritmo *shift left*

- Generalizando $p = a.b$, con a de n bits y b de m bits
- Si a es el multiplicando y b en es multiplicador. El algoritmo comprende m+1 pasos
- Cada paso computa

$$p^{(j+1)} = p^{(j)}.2 + a.b_{m\text{-}1\text{-}j}, \text{ con } j \text{ en } [0, m\text{-}1]$$

- En cada iteración se desplazan a la izquierda $p^{(j)}$ y b

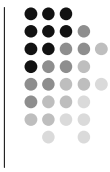
Multiplicación binaria



Algoritmo *shift left*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1110_2$

Multiplicación binaria



Algoritmo *shift left*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1110_2$

```
  1010
x 1110
------
  0000   p(0).2
  1010   a.b3
------
  1010   p(1)
```

Multiplicación binaria

Algoritmo *shift left*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1110_2$

1010			1010		
x 1110			x 1110		
0000	$p^{(0)}.2$		10100	$p^{(1)}.2$	
1010	$a.b_3$		1010	$a.b_2$	
1010	$p^{(1)}$		11110	$p^{(2)}$	

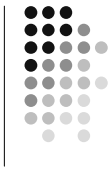
Multiplicación binaria

Algoritmo *shift left*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1110_2$

1010			1010			1010	
x 1110			x 1110			x 1110	
0000	$p^{(0)}.2$		10100	$p^{(1)}.2$		111100	$p^{(2)}.2$
1010	$a.b_3$		1010	$a.b_2$		1010	$a.b_1$
1010	$p^{(1)}$		11110	$p^{(2)}$		1000110	$p^{(3)}$

Multiplicación binaria



Algoritmo *shift left*

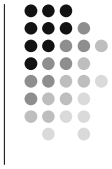
- Determinar $p = a.b$, con $a = 1010_2$ y $b = 1110_2$

1010		1010		1010		1010	
x 1110		x 1110		x 1110		x 1110	
0000	$p^{(0)}.2$	10100	$p^{(1)}.2$	111100	$p^{(2)}.2$	10001100	$p^{(3)}.2$
1010	$a.b_3$	1010	$a.b_2$	1010	$a.b_1$	0000	$a.b_0$
1010	$p^{(1)}$	11110	$p^{(2)}$	1000110	$p^{(3)}$	10001100	$p^{(4)}$

$p = p^{(4)} = 10001100_2$ (140_{10})

- Se requieren sumadores de 8 bits.

Multiplicación binaria



Implementación de multiplicador binario

Tamaño del resultado

$$a = \sum_{j=0}^{n-1} a_j.2^j \qquad b = \sum_{j=0}^{m-1} b_j.2^j$$

Multiplicación binaria

Implementación de multiplicador binario

Tamaño del resultado

$$a = \sum_{j=0}^{n-1} a_j.2^j \qquad b = \sum_{j=0}^{m-1} b_j.2^j$$

$p = a.b = (a_{n-1}.b_{m-1}).2^{m+n-2} + (a_{n-1}.b_{m-2}+a_{n-2}.b_{m-1}).2^{m-n-3} + \ldots + (a_0.b_1+a_1.b_0).2^1 + a_0.b_0$

Multiplicación binaria

Implementación de multiplicador binario

Tamaño del resultado

$$a = \sum_{j=0}^{n-1} a_j.2^j \qquad b = \sum_{j=0}^{m-1} b_j.2^j$$

$p = a.b = (a_{n-1}.b_{m-1}).2^{m+n-2} + (a_{n-1}.b_{m-2}+a_{n-2}.b_{m-1}).2^{m-n-3} + \ldots + (a_0.b_1+a_1.b_0).2^1 + a_0.b_0$

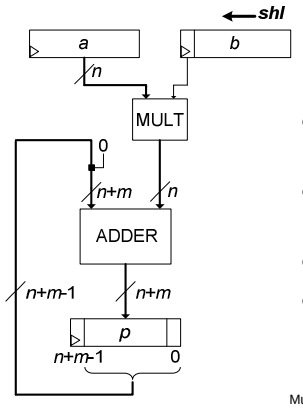
carry carry

Si la suma en el peso $m+n-2$ genera acarreo, entonces se produce un dígito en el peso $m+n-1$ (2^{m+n-1})

El tamaño del resultado, p, es de $m+n$ bits

Multiplicación binaria

Implementación de multiplicador binario *shift left* Versión Secuencial (serie)



- *MULT* se implementa con *n* compuertas *and* en paralelo
- necesita una unidad de control sencilla
- *m* ciclos de latencia
- sumador de *n+m* bits

Multiplicación binaria

Algoritmo *shift right*

- Sea $p = a.b$, con b de 4 bits, entonces

$$p = (a.b_3.2^3 + a.b_2.2^2 + a.b_1.2^1 + a.b_0).2^{-3}.2^3$$

$$p = ((((a.b_0).2^{-1} + a.b_1).2^{-1} + a.b_2).2^{-1} + a.b_3).2^3$$

Multiplicación binaria

Algoritmo *shift right*

- Sea $p = a.b$, con b de 4 bits, entonces

$$p = (a.b_3.2^3 + a.b_2.2^2 + a.b_1.2^1 + a.b_0).2^{-3}.2^3$$

$$p = ((((a.b_0).2^{-1} + a.b_1).2^{-1} + a.b_2).2^{-1} + a.b_0).2^3$$

$$p^{(1)} = p^{(0)}.2^{-1} + a.b_0.2^3$$

$$p^{(2)} = p^{(1)}.2^{-1} + a.b_1.2^3$$

$$p^{(3)} = p^{(2)}.2^{-1} + a.b_2.2^3$$

$$p^{(4)} = p^{(3)}.2^{-1} + a.b_3.2^3$$

- El algoritmo consta de 5 pasos, $p^{(0)}$ se inicializa en 0

Multiplicación binaria

Algoritmo *shift right*

- Generalizando $p = a.b$, con a de n bits y b de m bits
- Si a es el multiplicando y b en es multiplicador. El algoritmo comprende m+1 pasos
- Cada paso computa

$$p^{(j+1)} = p^{(j)}.2^{-1} + a.b_j.2^{m-1} \quad \text{con } j \text{ en } [0, m\text{-}1]$$

- En cada iteración se desplazan a la derecha $p^{(j)}$ y b

Multiplicación binaria

Algoritmo *shift right*

Análisis de la suma involucrada en en cálculo de $p^{(j+1)} = p^{(j)}.2^{-1} + a.b_j.2^{m-1}$

- Ejemplo, sumar dos operandos de 8 bits, *x* e *y*, donde *y* termina siempre con cuatro 0's, es decir $y = y'.2^4$

$$\begin{array}{rcccccccc} & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ + & y'_3 & y'_2 & y'_1 & y'_0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

Multiplicación binaria

Algoritmo *shift right*

Análisis de la suma involucrada en en cálculo de $p^{(j+1)} = p^{(j)} + a.b_j.2^{m-1}$

- Ejemplo, sumar dos operandos de 8 bits, *x* e *y*, donde *y* termina siempre con cuatro 0's, es decir $y = y'.2^4$

$$\begin{array}{rcccccccc} & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ + & y'_3 & y'_2 & y'_1 & y'_0 & 0 & 0 & 0 & 0 \\ \hline & & & & & x_3 & x_2 & x_1 & x_0 \end{array}$$

- Solo se necesita un sumador de 4 bits en lugar de uno de 8 bits.

Multiplicación binaria

Algoritmo *shift right*

Análisis de la suma involucrada en en cálculo de $p^{(j+1)} = p^{(j)} + a.b_j.2^{m-1}$

- Ejemplo, sumar dos operandos de 8 bits, *x* e *y*, donde *y* termina siempre con cuatro 0's, es decir $y = y'.2^4$

$$\begin{array}{rcccccccc} & x_7 & x_6 & x_5 & x_4 & x_3 & x_2 & x_1 & x_0 \\ + & y'_3 & y'_2 & y'_1 & y'_0 & 0 & 0 & 0 & 0 \\ \hline & & & & & x_3 & x_2 & x_1 & x_0 \end{array}$$

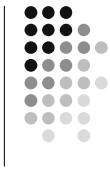
- Solo se necesita un sumador de 4 bits en lugar de uno de 8 bits.
- Para el caso de la suma en algoritmo *shift right*, se requiere un sumador de *n* bits en lugar de un sumador de *n+m* bits como en el caso de *shift left*.

Multiplicación binaria

Algoritmo *shift right*

- Determinar $p = a.b$, con $a = 1010_2$ y $b = 1111_2$

Multiplicación binaria

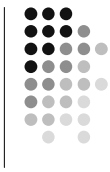


Algoritmo *shift right*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1111_2$

```
  1010
x 1111
 ------
  0000   p(0).2-1
  1010   a.b0
 ------
0 1010   p(1)
```


Multiplicación binaria



Algoritmo *shift right*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1111_2$

```
  1010                  1010
x 1111                x 1111
 ------                ------
  0000   p(0).2-1       01010   p(1).2-1
  1010   a.b0           1010    a.b1
 ------                ------
0 1010   p(1)         0 11110   p(2)
```

Multiplicación binaria

Algoritmo *shift right*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1111_2$

1010			1010			1010		
x 1111			x 1111			x 1111		
0000	$p^{(0)}.2^{-1}$		01010	$p^{(1)}.2^{-1}$		011110	$p^{(2)}.2^{-1}$	
1010	$a.b_0$		1010	$a.b_1$		1010	$a.b_2$	
0 1010	$p^{(1)}$		0 11110	$p^{(2)}$		1 000110	$p^{(3)}$	

Multiplicación binaria

Algoritmo *shift right*

- Determinar $p = a.b$, con $a = 1010_2$ y $b=1111_2$

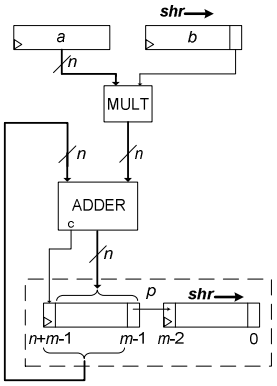
1010			1010			1010			1010		
x 1111			x 1111			x 1111			x 1111		
0000	$p^{(0)}.2^{-1}$		01010	$p^{(1)}.2^{-1}$		011110	$p^{(2)}.2^{-1}$		1000110	$p^{(3)}.2^{-1}$	
1010	$a.b_0$		1010	$a.b_1$		1010	$a.b_2$		1010	$a.b_3$	
0 1010	$p^{(1)}$		0 11110	$p^{(2)}$		1 000110	$p^{(3)}$		1 0010110	$p^{(4)}$	

$p = p^{(4)} = 10010110_2$ (150_{10})

- Se requiere sumador de 4 bits

Multiplicación binaria

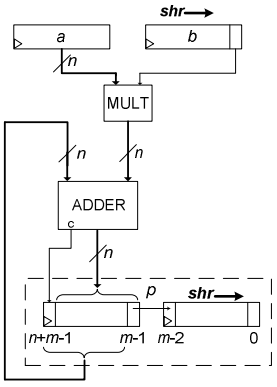
Implementación de multiplicador binario *shift right*
Versión Secuencial (serie)



- *MULT* se implementa con *n* compuertas *and* en paralelo
- necesita una unidad de control sencilla
- *m* ciclos de latencia
- sumador de *n* bits

Multiplicación binaria

Implementación de multiplicador binario *shift right*
Versión Secuencial (serie)



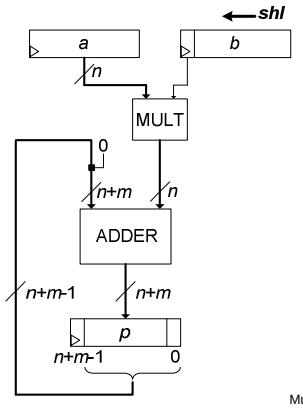
- *MULT* se implementa con *n* compuertas *and* en paralelo
- necesita una unidad de control sencilla
- *m* ciclos de latencia
- sumador de *n* bits

Comparado a *shift left*

- *Menor ocupación de área*
- *Menor tiempo de ciclo*

Multiplicación binaria

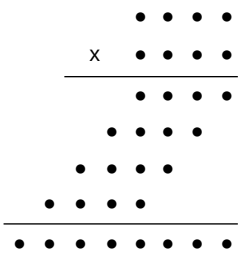
Implementación de multiplicador binario *shift left*
Versión Secuencial (serie)



- *MULT* se implementa con *n* compuertas *and* en paralelo
- necesita una unidad de control sencilla
- *m* ciclos de latencia
- sumador de *n*+*m* bits

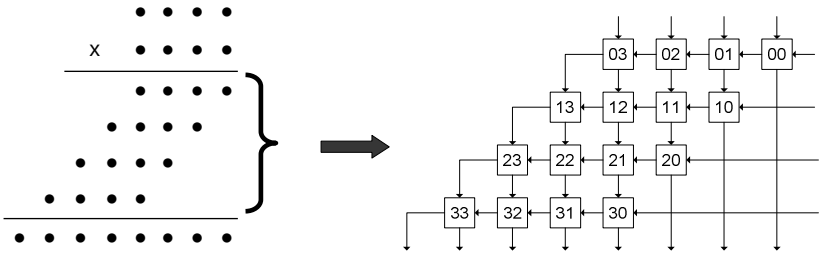
Multiplicadores celulares

$p = a.b$



Multiplicadores celulares

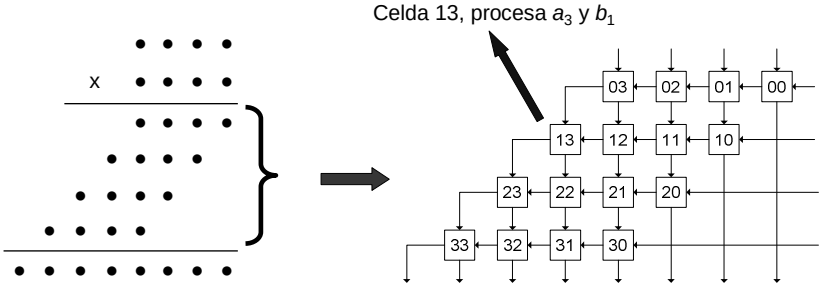
$p = a.b$



Multiplicadores celulares

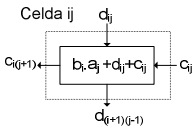
$p = a.b$

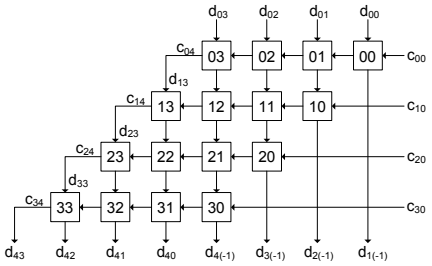
Celda 13, procesa a_3 y b_1



Multiplicador *Ripple-Carry*
con propagación de acarreo

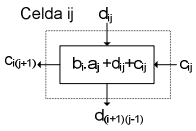
$p = a.b$

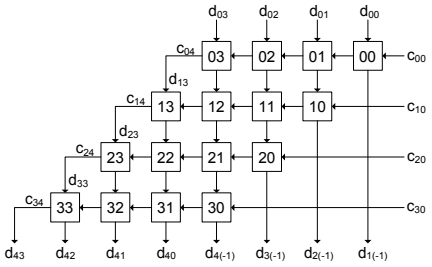




Multiplicador *Ripple-Carry*
con propagación de acarreo

$p = a.b$





Se puede realizar $p = a.b + k + l$,

seteando $d_{0j} = k_j$ y $c_{i0} = l_i$

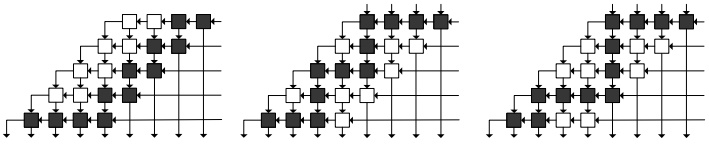
Multiplicador *Ripple-Carry*
con propagación de acarreo

Costos de implementar *p* = *a*.*b*, con *a* de *n* bits y *b* de *m* bits

Multiplicador *Ripple-Carry*
con propagación de acarreo

Costos de implementar *p* = *a*.*b*, con *a* de *n* bits y *b* de *m* bits

camino crítico, para *a* de 4 bits y *b* de 5 bits, pasa por 12 celdas

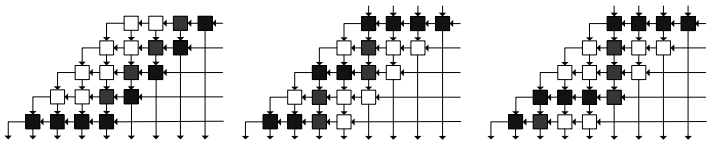


generalizando, en el camino crítico se transita *n*+2.*m*-2 celdas

Multiplicador *Ripple-Carry*
con propagación de acarreo

Costos de implementar *p* = *a*.*b*, con *a* de *n* bits y *b* de *m* bits

camino crítico, para *a* de 4 bits y *b* de 5 bits, pasa por 12 celdas



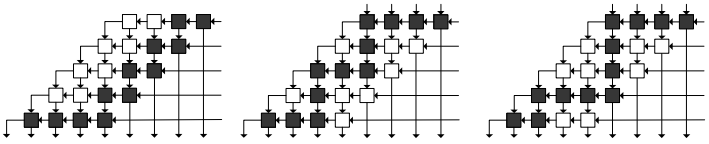
generalizando, en el camino crítico se transita $n+2.m-2$ celdas

$n+m-1$ celdas (cantidad de columnas), y
$m-1$ celdas (cantidad de filas menos uno)

Multiplicador *Ripple-Carry*
con propagación de acarreo

Costos de implementar *p* = *a*.*b*, con *a* de *n* bits y *b* de *m* bits

camino crítico, para *a* de 4 bits y *b* de 5 bits, pasa por 12 celdas



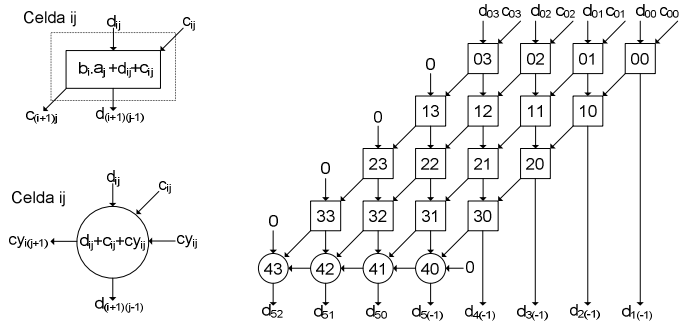
generalizando, en el camino crítico se transita $n+2.m-2$ celdas

Tiempo de cálculo ⟶ $T_{RCM} = (n+2.m-2).T_{celda}$

Costo en área ⟶ $C_{RCM} = n.m.C_{celda}$

Multiplicador celular *Carry-Save*

p = *a.b,* con *a* y *b* de 4 bits

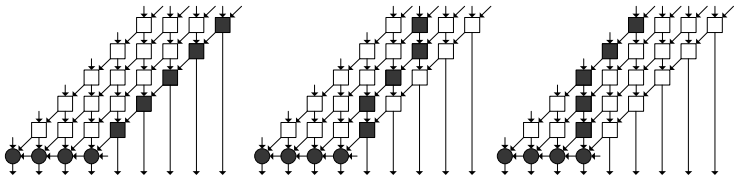


- "Salva" el acarreo entre dos celdas de una misma fila
- Posee una fila adicional, respecto a *Ripple-carry*, con suma de los acarreos "salvados"

Multiplicador celular *Carry-Save*

Costos de implementar *p* = *a.b,* con *a* de *n* bits y *b* de *m* bits

camino crítico, para *a* de 4 bits y *b* de 5 bits, pasa por 9 celdas

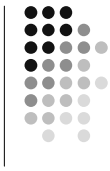


generalizando, en el camino crítico se transita *n*+*m* celdas

Tiempo de cálculo ⟶ $T_{CSM} = (n+m).T_{celda}$

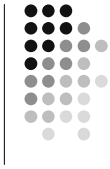
Costo en área ⟶ $C_{CSM} = (n.m+n).C_{celda}$

Multiplicadores celulares estudiados



- Basados en algoritmo clásico de *shift&add*
- Soluciones combinacionales o paralelos
- Ideales para segmentar (*pipelining*)
- Multiplicador *Carry-Save* vs. *Ripple-Carry* (multiplicando de *n* bits y multiplicador de *m* bits)
  - Aceleración = $1+\frac{m-2}{m+n}$
  - Penalización en área = $1+\frac{1}{m}$

Multiplicación con signo
(complemento a la base)



- Se considera el signo del multiplicando en la acumulación de los productos parciales
  - utiliza sumador/restador
  - consideración de *overflow* en el desplazamiento
- Se interpreta el bit más significativo del multiplicador con peso negativo
  - si el multiplicador es negativo, suma el opuesto del multiplicando en última iteración

Multiplicación con signo
(complemento a la base)

Ejemplo 1:

- Determinar $p = a.b$, con
  - a y b representaciones en complemento a la base
  - tamaño de a 5 bits, entonces a está en [-16,15]
  - tamaño de b 4 bits, entonces b está en [-8, 7]
  - tamaño del resultado, p, 9 bits, entonces está en [-256,255]

Multiplicación con signo
(complemento a la base)

Ejemplo 1:

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = 7_{10}$ (0111_2)

```
  10100
x  0111
-------
  00000   p(0).2-1
  10100   a.b0
-------
  10100   p(1)
```

no hay *overflow* y resultado negativo,

conserva signo
➡1

Multiplicación con signo
(complemento a la base)

Ejemplo 1:

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = 7_{10}$ (0111_2)

10100 x 0111		10100 x 0111	
00000	$p^{(0)}.2^{-1}$	110100	$p^{(1)}.2^{-1}$
10100	$a.b_0$	10100	$a.b_1$
10100	$p^{(1)}$	011100	$p^{(2)}$
no hay *overflow* y resultado negativo,		**hay *overflow* y resultado positivo,**	
conserva signo ➡1		**invierte signo** ➡1	

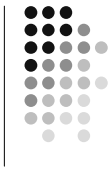
Multiplicación con signo
(complemento a la base)

Ejemplo 1:

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = 7_{10}$ (0111_2)

10100 x 0111		10100 x 0111		10100 x 0111	
00000	$p^{(0)}.2^{-1}$	110100	$p^{(1)}.2^{-1}$	1011100	$p^{(2)}.2^{-1}$
10100	$a.b_0$	10100	$a.b_1$	10100	$a.b_2$
10100	$p^{(1)}$	011100	$p^{(2)}$	0101100	$p^{(3)}$
no hay *overflow* y resultado negativo,		**hay *overflow* y resultado positivo,**		**hay *overflow* y resultado positivo,**	
conserva signo ➡1		**invierte signo** ➡1		**invierte signo** ➡1	

Multiplicación con signo
(complemento a la base)



Ejemplo 1:

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = 7_{10}$ (0111_2)

10100 x 0111			
00000	$p^{(0)}.2^{-1}$		
10100	$a.b_0$	10100 x 0111	
10100	$p^{(1)}$	110100	$p^{(1)}.2^{-1}$
		10100	$a.b_1$
		011100	$p^{(2)}$
			10100 x 0111
			1011100 $p^{(2)}.2^{-1}$
			10100 $a.b_2$
			0101100 $p^{(3)}$
no hay *overflow* y resultado negativo,		**hay *overflow* y resultado positivo,**	
conserva signo ➡1		**invierte signo** ➡1	

10100 x 0111		
10101100	$p^{(3)}.2^{-1}$	
00000	$a.b_3$	
10101100	$p^{(4)}$	
hay *overflow* y resultado positivo,		**no hay *overflow* y resultado negativo,**
invierte signo ➡1		**conserva signo** ➡1

$$p = p^{(4)} = 110101100_2\ (-84_{10})$$

- Se requiere sumador/restador de 5 bits

Multiplicación con signo
(complemento a la base)

Ejemplo 2:

- Determinar $p = a.b$, $a = 12_{10}$ (01100_2) y $b = -5_{10}$ (1011_2)

01100 x 1011	
00000	$p^{(0)}.2^{-1}$
01100	$a.b_0$
01100	$p^{(1)}$

no hay *overflow* y resultado positivo,

conserva signo ➡0

Multiplicación con signo
(complemento a la base)

Ejemplo 2:

- Determinar $p = a.b$, $a = 12_{10}$ (01100_2) y $b = -5_{10}$ (1011_2)

01100		01100	
x 1011		x 1011	
00000	$p^{(0)}.2^{-1}$	001100	$p^{(1)}.2^{-1}$
01100	$a.b_0$	01100	$a.b_1$
01100	$p^{(1)}$	100100	$p^{(2)}$

no hay *overflow* y resultado positivo,

conserva signo

➡0

hay *overflow* y resultado negativo,

invierte signo

➡0

Multiplicación con signo
(complemento a la base)

Ejemplo 2:

- Determinar $p = a.b$, $a = 12_{10}$ (01100_2) y $b = -5_{10}$ (1011_2)

01100		01100		01100	
x 1011		x 1011		x 1011	
00000	$p^{(0)}.2^{-1}$	001100	$p^{(1)}.2^{-1}$	0100100	$p^{(2)}.2^{-1}$
01100	$a.b_0$	01100	$a.b_1$	00000	$a.b_2$
01100	$p^{(1)}$	100100	$p^{(2)}$	0100100	$p^{(3)}$

no hay *overflow* y resultado positivo,

conserva signo

➡0

hay *overflow* y resultado negativo,

invierte signo

➡0

no hay *overflow* y resultado positivo,

conserva signo

➡0

Multiplicación con signo
(complemento a la base)

Ejemplo 2:

- Determinar $p = a.b$, $a = 12_{10}$ (01100_2) y $b = -5_{10}$ (1011_2)

01100 x 1011	01100 x 1011	01100 x 1011	01100 x 1011
00000 $p^{(0)}.2^{-1}$	001100 $p^{(1)}.2^{-1}$	0100100 $p^{(2)}.2^{-1}$	00100100 $p^{(3)}.2^{-1}$
01100 $a.b_0$	01100 $a.b_1$	00000 $a.b_2$	**10100** $-\boldsymbol{a}.b_3$
01100 $p^{(1)}$	100100 $p^{(2)}$	0100100 $p^{(3)}$	11000100 $p^{(4)}$
no hay *overflow* y resultado positivo,	**hay *overflow* y resultado negativo,**	**no hay *overflow* y resultado positivo,**	**no hay *overflow* y resultado negativo,**
conserva signo ➡0	**invierte signo** ➡0	**conserva signo** ➡0	**conserva signo** ➡1

$$p = p^{(4)} = 111000100_2\ (-60_{10})$$

- Se requiere sumador/restador de 5 bits

Multiplicación con signo
(complemento a la base)

Ejemplo 3:

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = -5_{10}$ (1011_2)

10100 x 1011	10100 x 1011	10100 x 1011	10100 x 1011
00000 $p^{(0)}.2^{-1}$	110100 $p^{(1)}.2^{-1}$	1011100 $p^{(2)}.2^{-1}$	11011100 $p^{(3)}.2^{-1}$
10100 $a.b_0$	10100 $a.b_1$	00000 $a.b_2$	**01100** $-\boldsymbol{a}.b_3$
10100 $p^{(1)}$	011100 $p^{(2)}$	1011100 $p^{(3)}$	00111100 $p^{(4)}$
no hay *overflow* y resultado negativo,	**hay *overflow* y resultado positivo,**	**no hay *overflow* y resultado negativo,**	**no hay *overflow* y resultado positivo,**
conserva signo ➡1	**invierte signo** ➡1	**conserva signo** ➡1	**conserva signo** ➡0

$$p = p^{(4)} = 000111100_2\ (60_{10})$$

- Se requiere sumador/restador de 5 bits

Multiplicación con signo
(complemento a la base)

Ejemplo 4:

- Determinar $p = a.b$, $a = 12_{10}$ (01100_2) y $b = 7_{10}$ (0111_2)

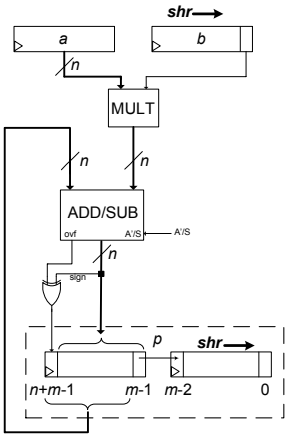
01100		01100		01100		01100	
x 0111		x 0111		x 0111		x 0111	
00000	$p^{(0)}.2^{-1}$	001100	$p^{(1)}.2^{-1}$	0100100	$p^{(2)}.2^{-1}$	01010100	$p^{(3)}.2^{-1}$
01100	$a.b_0$	01100	$a.b_1$	01100	$a.b_2$	00000	$a.b_3$
01100	$p^{(1)}$	100100	$p^{(2)}$	1010100	$p^{(3)}$	01010100	$p^{(4)}$
no hay *overflow* y resultado positivo,		**hay *overflow* y resultado negativo,**		**hay *overflow* y resultado negativo,**		**no hay *overflow* y resultado positivo,**	
conserva signo ➡0		**invierte signo** ➡0		**invierte signo** ➡0		**conserva signo** ➡0	

$$p = p^{(4)} = 001010100_2\ (84_{10})$$

- Se requiere sumador/restador de 5 bits

Multiplicación con signo
(complemento a la base)

Implementación de multiplicador binario con signo *shift right*
Versión Secuencial (serie)



- *MULT* se implementa con *n* compuertas *and* en paralelo
- necesita una unidad de control sencilla
- *m* ciclos de latencia
- sumador/restador *n* bits

Multiplicación
recodificación de Booth

- Una alternativa al complemento a la base, es representar el operando multiplicador usando recodificación de Booth
- Desarrollado para acelerar la multiplicación binaria en las computadoras antiguas
- Cuando se procesa un 0, se evita la suma y se efectúa solo el desplazamiento
- Demora variable, depende de la cantidad de 1's en el multiplicador

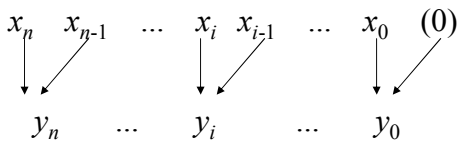
Multiplicación
recodificación de Booth

- Booth observó que una cadena de 1's, puede ser reemplazada por:

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

- Ejemplos
  - sea 011110_2 representación de 30_{10}= $2^4+2^3+2^2+2$, entonces aplicando recodificación de Booth es 1000-10, esto es $30_{10}=2^5-2^1$
  - sea 011101_2 representación de 29_{10}= $2^4+2^3+2^2+2^0$ entonces aplicando recodificación de Booth es 100-11-1, esto es $29_{10}=2^5-2^2+2^1-2^0$
  - sea 001111_2 representación de 15_{10}= $2^3+2^2+2^1+2^0$ entonces aplicando recodificación de Booth es 01000-1, esto es $15_{10}=2^4-2^0$

Multiplicación
recodificación de Booth



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

x_i	x_{i-1}	**Operación**	**Comentario**	y_i
0	0	sólo shift	cadena de 0´s	0
1	1	sólo shift	cadena de 1´s	0
1	0	resta y shift	inicio cadena 1´s	-1
0	1	suma y Swift	fin cadena de 1´s	1

Multiplicación
recodificación de Booth

- Ejemplos
  - sea 011110_2 representación en CB(2,6) de $30_{10} = 2^4+2^3+2^2+2$, entonces aplicando recodificación de Booth es 1000-10, esto es $30_{10}=2^5-2^1$
  - sea 111100_2 representación en CB(2,6) de $-4_{10} = -2^5+2^4+2^3+2^2$, entonces aplicando recodificación de Booth es 000-100, esto es $-4_{10}=-2^2$
  - sea 101111_2 representación en CB(2,6) de $-17_{10} = -2^5+2^3+2^2+2^1+2^0$ entonces aplicando recodificación de Booth es -11000-1, esto es $-17_{10}=-2^5+2^4-2^0$

Multiplicación
(recodificación de Booth)

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = -5_{10}$ (1011_2), por Booth $-5_{10} = -110\text{-}1$

```
  10100
x -110-1
-------
  00000   p(0).2-1
  01100   -a.b0
-------
  01100   p(1)
```

no hay *overflow* y resultado positivo,

conserva signo
➡ 0

Multiplicación
(recodificación de Booth)

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = -5_{10}$ (1011_2), por Booth $-5_{10} = -110\text{-}1$

```
  10100
x -110-1
-------
  00000   p(0).2-1
  01100   -a.b0
-------
  01100   p(1)
```

no hay *overflow* y resultado positivo,

conserva signo
➡ 0

```
  10100
x -110-1
-------
  001100  p(1).2-1
  00000   a.b1
-------
  001100  p(2)
```

no hay *overflow* y resultado positivo,

conserva signo
➡ 0

Multiplicación
(recodificación de Booth)

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = -5_{10}$ (1011_2), por Booth $-5_{10} = -110\text{-}1$

10100			10100			10100		
x -110-1			x -110-1			x -110-1		
00000	$p^{(0)}.2^{-1}$		001100	$p^{(1)}.2^{-1}$		0001100	$p^{(2)}.2^{-1}$	
01100	$-a.b_0$		00000	$a.b_1$		10100	$a.b_2$	
01100	$p^{(1)}$		001100	$p^{(2)}$		1011100	$p^{(3)}$	
no hay *overflow* y resultado positivo,			**no hay *overflow* y resultado positivo,**			**no hay *overflow* y resultado negativo,**		
conserva signo ➡ 0			**conserva signo** ➡ 0			**conserva signo** ➡ 1		

Multiplicación
(recodificación de Booth)

- Determinar $p = a.b$, $a = -12_{10}$ (10100_2) y $b = -5_{10}$ (1011_2), por Booth $-5_{10} = -110\text{-}1$

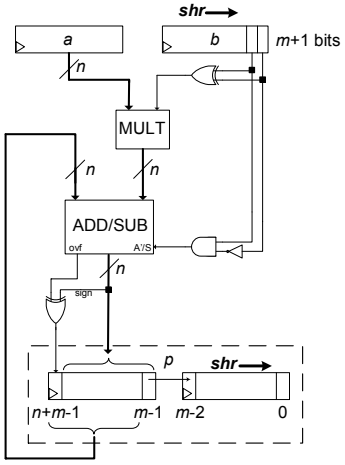
10100			10100			10100			10100		
x -110-1			x -110-1			x -110-1			x -110-1		
00000	$p^{(0)}.2^{-1}$		001100	$p^{(1)}.2^{-1}$		0001100	$p^{(2)}.2^{-1}$		11011100	$p^{(3)}.2^{-1}$	
01100	$-a.b_0$		00000	$a.b_1$		10100	$a.b_2$		01100	$-a.b_3$	
01100	$p^{(1)}$		001100	$p^{(2)}$		1011100	$p^{(3)}$		00111100	$p^{(4)}$	
no hay *overflow* y resultado positivo,			**no hay *overflow* y resultado positivo,**			**no hay *overflow* y resultado negativo,**			**no hay *overflow* y resultado positivo,**		
conserva signo ➡ 0			**conserva signo** ➡ 0			**conserva signo** ➡ 1			**conserva signo** ➡ 0		

$p = p^{(4)} = 000111100_2$ (60_{10})

Multiplicación con signo

(recodificación de Booth)

Implementación de multiplicador binario con signo *shift right*
Versión Secuencial (serie)

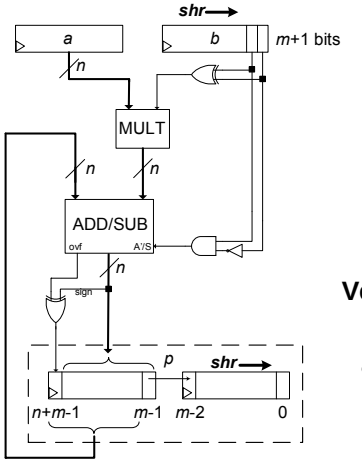


- *MULT* se implementa con *n* compuertas *and* en paralelo
- necesita una unidad de control sencilla
- *m* ciclos de latencia
- sumador/restador *n* bits

Multiplicación con signo

(recodificación de Booth)

Implementación de multiplicador binario con signo *shift right*
Versión Secuencial (serie)



- *MULT* se implementa con *n* compuertas *and* en paralelo
- necesita una unidad de control sencilla
- *m* ciclos de latencia
- sumador/restador *n* bits

Ventaja respecto a multiplicador *shift right* complemento a la base

- Probablemente menor consumo de energía