

Estructura de Computadores

Tema 3

Aritmética de enteros

DISCA

Aritmética de enteros

- Objetivos:

- ✓ Hacer la correspondencia entre los tipos de datos numéricos de alto nivel (p. ej, Java y C/C++) y los tipos nativos del procesador
- ✓ Traducir a ensamblador expresiones aritméticas (cálculos y guardas) expresadas en alto nivel
- ✓ Distinguir entre operadores combinacionales y secuenciales y calcular su tiempo de operación y la productividad en casos sencillos a partir de los retardos de los componentes
- ✓ Relacionar manipulación de bits con operaciones de alto nivel (p. ej, operar con los campos del formato de coma flotante)
- ✓ Distinguir las diferentes respuestas del computador ante las operaciones que no se pueden completar (por desbordamiento o por indeterminación)
- ✓ Entender el soporte que da el juego de instrucciones a las singularidades del cálculo (excepciones, indicadores de la norma IEEE)

Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Índice

Introducción

1. *Típos de datos en alto y bajo nivel*
2. *Operaciones y operadores*
 - *Los cálculos en los computadores*
 - *La unidad aritmética y lógica*
 - *Parámetros de una ALU*
 - *Prestaciones*
3. *Ejemplos: operaciones lógicas*
4. *La representación de los enteros*
 - *Notas sobre la representación de los enteros*
 - *Conversión entre tipos*
 - *El salto condicional en el MIPS*

I. Tipos en alto y en bajo nivel

- Tipos de datos básicos

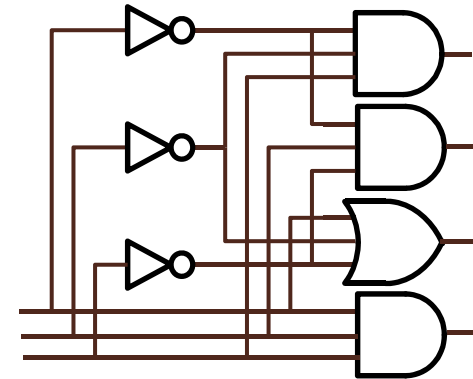
	bits	Java	C/C++ (32 bits)	MIPS	x64/IA-64
carácter	8	—	char	byte	byte
	16	char	wchar_t	halfword	word
entero sin signo	8	—	unsigned byte	byte	byte
	16	—	unsigned short	halfword	word
	32	—	unsigned int (long)	word	dword
entero con signo	8	byte	byte	byte	byte
	16	short	short	halfword	word
	32	int	int /long	word	dword
	64	long		—	qword
coma flotante	32	float	float	float	float
	64	double	double	double	double

2. Operaciones y operadores

- Los cálculos en los computadores
 - ✓ Las operaciones lógicas y aritméticas expresadas en alto nivel se traducen en datos y instrucciones de código máquina
 - ✓ Durante el ciclo de instrucción, el procesador aplica operadores para procesar los datos

```
int[] j;  
short a;  
float x,y;  
  
x = 5*j[a];  
if (a>x){  
    y=x*1.3e5;  
    j=(int)exp(x);  
}
```

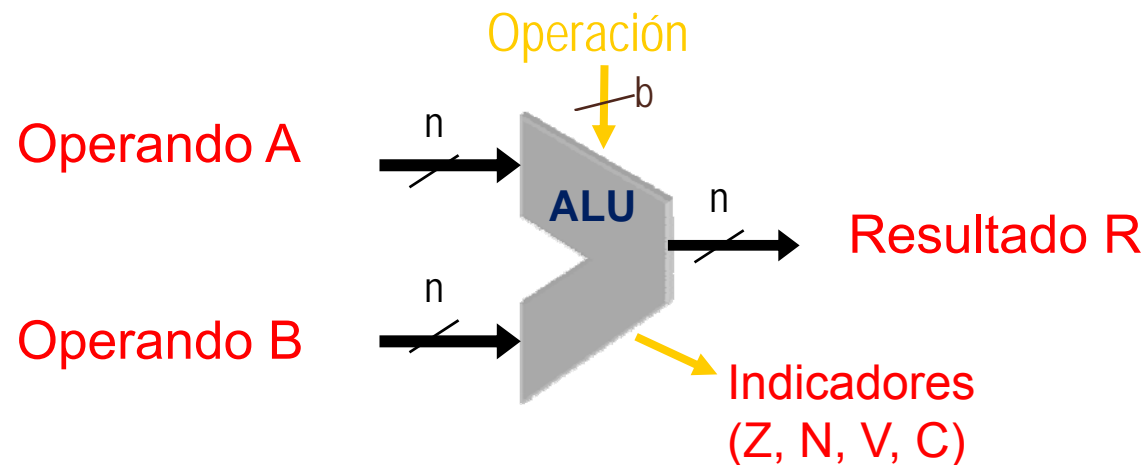
lbu \$t0,0xFF0(\$0)
lw \$t1,0x1020(\$2)
add \$t0,\$t0,\$t1
mtc1 \$t0,\$f1
...



2. Operaciones y operadores

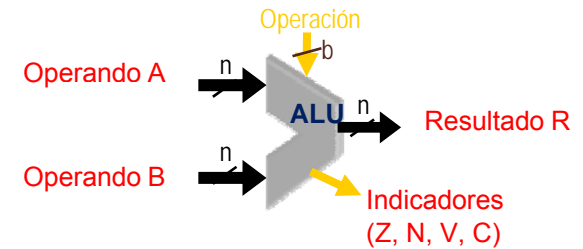
- La unidad aritmético-lógica

ALU (*Arithmetic Logic Unit*)



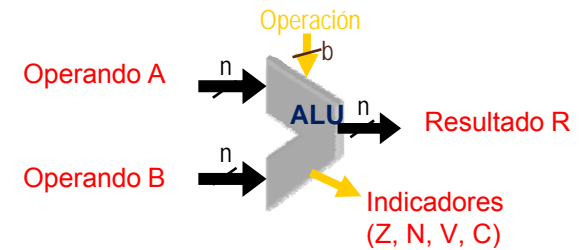
- ✓ es un elemento funcional del procesador formado por un conjunto de operadores digitales que realizan las operaciones codificadas en las instrucciones

2. Operaciones y operadores



- ✓ Cada operador implementa una o más operaciones que se aplican a los operandos y produce un resultado
- ✓ El control del procesador, dirigido por las instrucciones que se decodifican, selecciona los operadores y encamina operandos y resultados desde y a los registros implicados

2. Operaciones y operadores



- ✓ La ALU puede activar indicadores (*flags*) que dan información del resultado
 - $Z=1$ si $R=0$,
 - $N=1$ si $R<0$,
 - $V=1$ si desbordamiento en $Ca2$,
 - etc.

2. Operaciones y operadores

- Parámetros de una ALU

- ✓ Funcionales:

- Operaciones que puede realizar
 - Conversión entre tipos
 - Operaciones de bit (&, |) y desplazamientos
 - Cálculo elemental: suma, resta, multiplicación y división
 - Comparación (=, ≠, <, ≤, ≥, >)
 - Tipos de operandos que puede manejar

- ✓ Prestaciones (coste temporal):

- ¿A qué velocidad opera una ALU?
 - ¿Cuántas operaciones puede hacer por unidad de tiempo?

- ✓ Complejidad (coste espacial)

- ¿Cuántos recursos físicos hay que dedicar a los operadores?
 - ¿Qué espacio del chip hay que dedicar a los operadores?

2. Operaciones y operadores

- ¿Cómo expresaremos las prestaciones?
 - ✓ Tiempo de respuesta
 - Tiempo necesario para realizar un cálculo. Cuanto más corto, mejor
 - Se mide en unidades de tiempo (ns, tiempo de puerta...)
 - ✓ Productividad
 - Número de operaciones por unidad de tiempo. Cuanto más grande, mejor
 - Se puede medir con unidades genéricas OPS (operaciones por segundo), KOPS, MOPS, GOPS...
 - caso de coma flotante: FLOPS, MFLOPS, etc.
- Complejidad (coste espacial)
 - ✓ Número de puertas
 - ✓ Número de transistores
 - ✓ Superficie de xip. Unidades típicas: μm^2 , nm^2

3. Ejemplo: operaciones lógicas

- En el MIPS

formato R	formato I
OR	ORI
AND	ANDI
XOR	XORI
NOR	

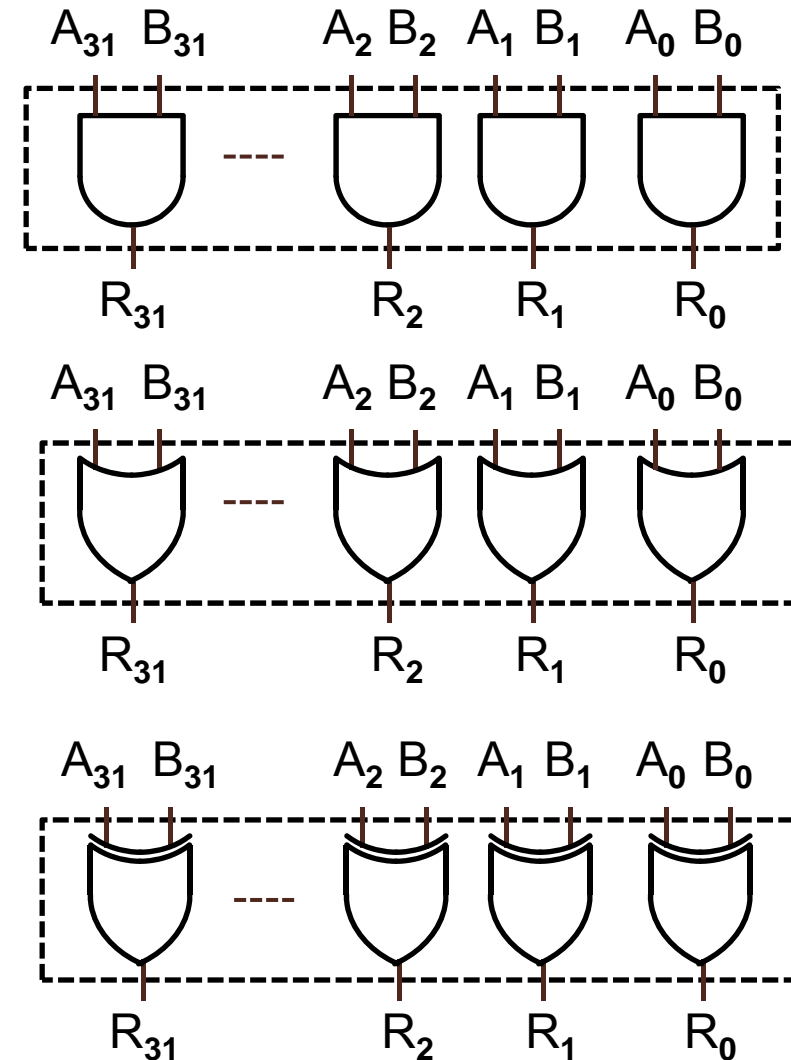
- ✓ Operación derivada: NOT (Cal)

```
nor $t0,$t0,$zero
```

alternativamente:

```
li $t1,0xFFFFFFFF
```

```
xor $t0,$t0,$t1
```



3. Ejemplo: operaciones lógicas

- En el MIPS

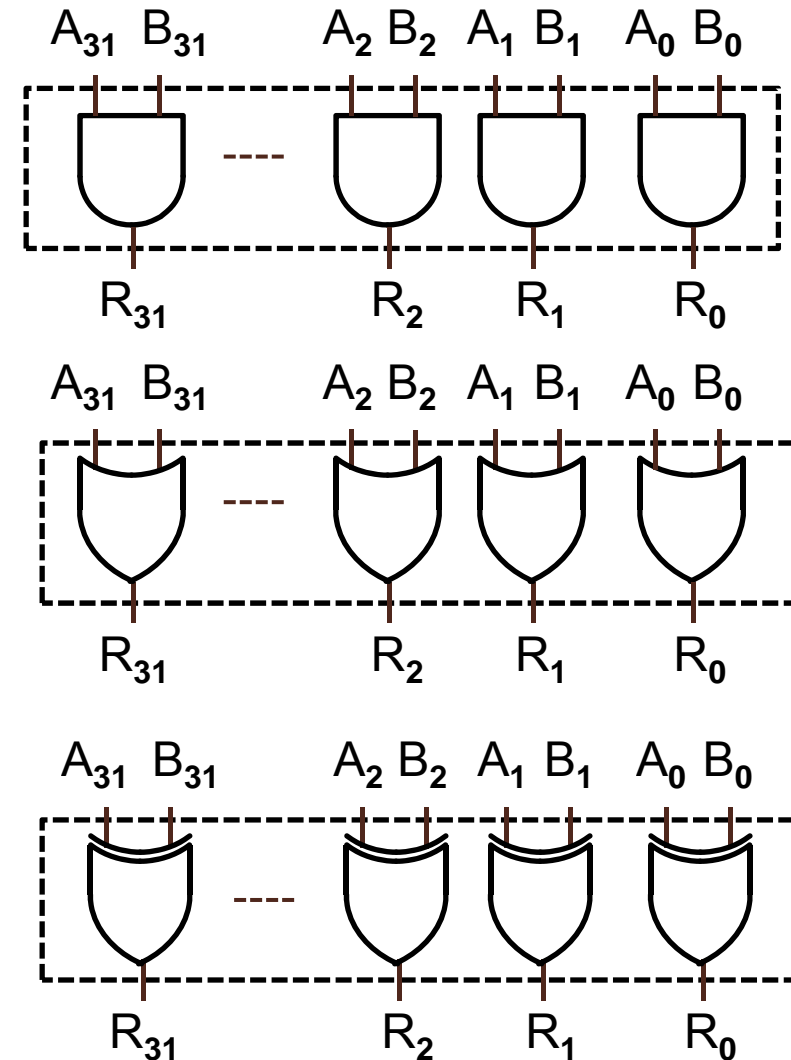
formato R	formato I
OR	ORI
AND	ANDI
XOR	XORI
NOR	

- ✓ Productividad de los operadores:

$$\chi = 1/t_{\text{Puerta}}$$

- Si $t_{\text{Puerta}} = 50 \text{ ps} = 50 \times 2^{-12} \text{ s}$

$$\chi = 1000 / 50 \text{ GOPS} = 20 \text{ GOPS}$$



3. Ejemplo: operaciones lógicas

- En Java o C

& (and) | (or) ^ (xor) ~ (not)

```
Int a = 0xA;
Int b=0x3;
Int c;
...
c= a & ~b;
...
System.out.println
(a+ " ^"+b+"=" +c);
```

lw \$s0,a
lw \$s1,b
li \$t0, 0xFFFFFFFF
xor \$s1,\$s1,\$t0
and \$s0,\$s0,\$s1
sw \$s0,c

$$10 \wedge \sim 3 = 8$$

4. Representación de los enteros

- Notas sobre la representación de enteros
 - ✓ Los conjuntos matemáticos \mathbb{N} y \mathbb{Z} son infinitos, pero los tipos de datos básicos de un computador tienen una capacidad de representación finita, exacta pero limitada.
 - Con n bits sólo se pueden representar 2^n palabras diferentes
 - Conjunto \mathbb{N} : números naturales, se codifican en código binario natural CBN.
 - Conjunto \mathbb{Z} : números enteros con signo, se codifican en complemento a dos:
 - Los positivos (se incluye el cero): en código binario natural
 - Los negativos : se hace al valor positivo la operación complemento a dos para representarlo.

4. Representación de los enteros

- Por ejemplo supongamos que $n = 3$

CBN	Sin Signo
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

$[0 \dots +7]$

$[0 \dots +2^n - 1]$

Ca2	Con Signo
3	011
2	010
1	001
0	000
-1	111
-2	110
-3	101
-4	100

$[-4 \dots +3]$

$[-2^{n-1} \dots +2^{n-1} - 1]$

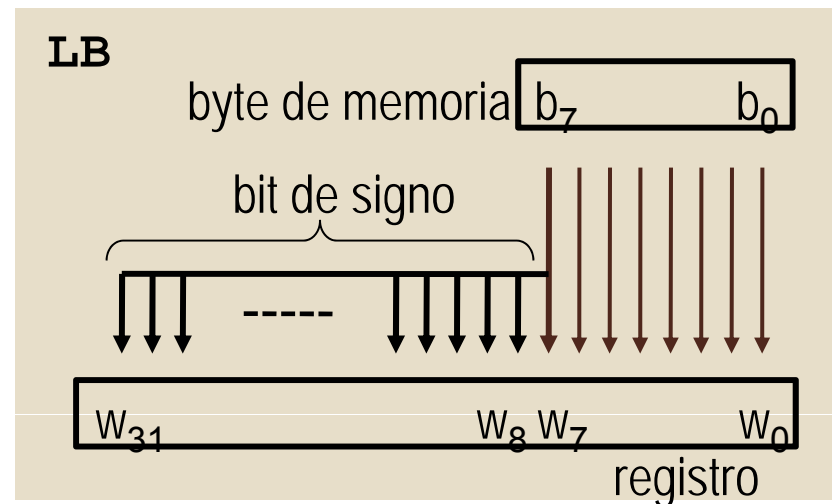
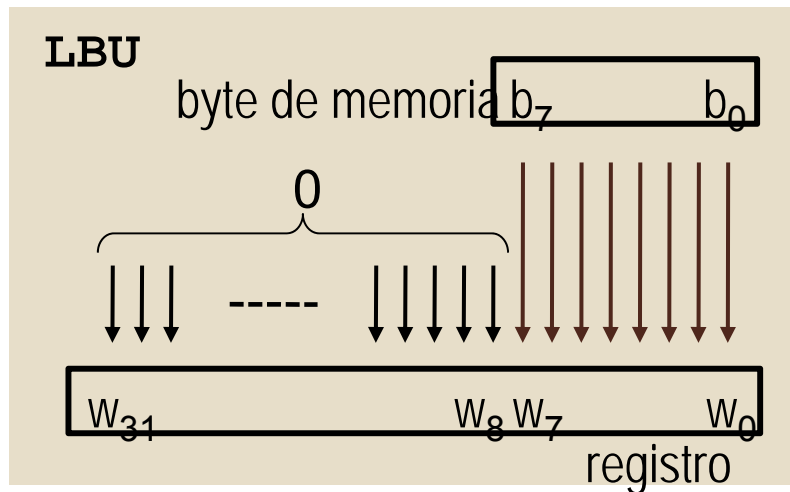
4. Representación de los enteros

- ✓ Rangos de representación con n bits
 - Para N: C.B.N. Rango $[0 \dots +2^n - 1]$
 - Para Z: codificación en complemento a 2: $[-2^{n-1} \dots +2^{n-1} - 1]$

n	Sin signo	Con signo
8	0 ... 255	-128 ... +127
16	0 ... 65.535	-32.768 ... +32.767
32	0 ... 4.294.967.295	-2.147.483.648 ... +2.147.483.647
64	0 ... $1.84 \cdot 10^{19}$ (aprox)	$-9.2 \cdot 10^{18}$... $+9.2 \cdot 10^{18}$ (aprox)

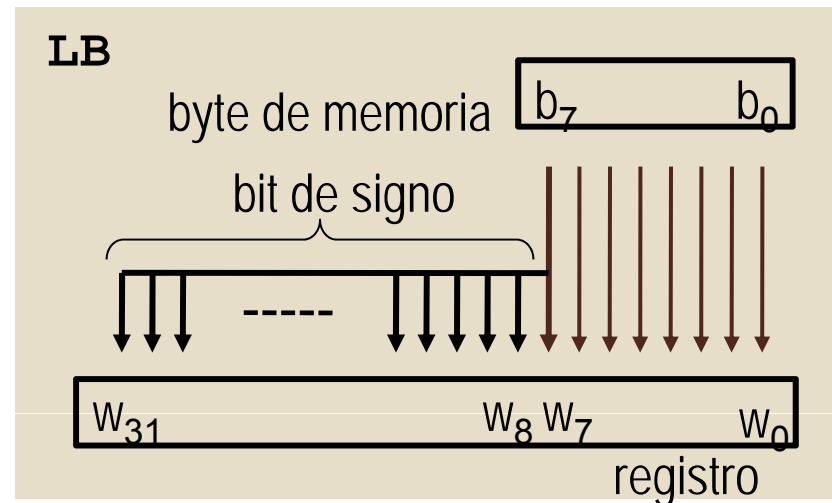
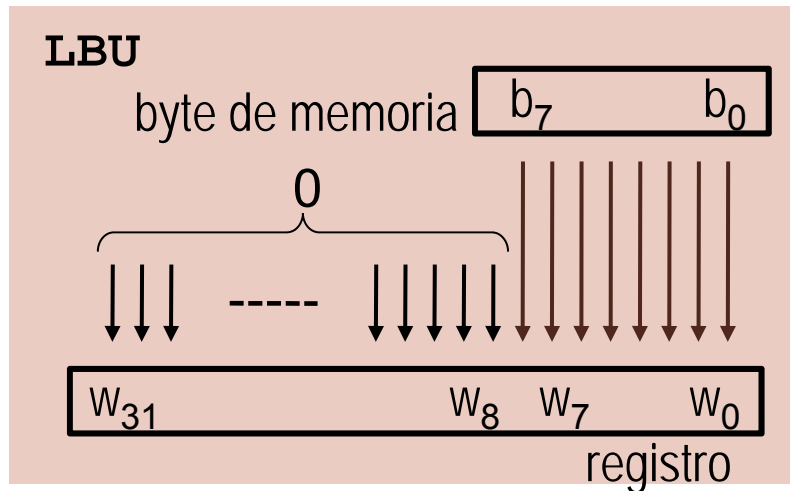
4. Representación de los enteros

- Conversión entre tipos enteros en el MIPS R2000
 - ✓ El tipo entero *nativo* es de 32 bits (ALU y registros)
 - ✓ Las instrucciones de acceso a la memoria hacen la conversión si hace falta
 - LBU y LHU añaden ceros por la izquierda (válido para CBN)
 - LB y LH hacen extensión de signo (válido para Ca2)
 - SB y SH eliminan bits por la izquierda



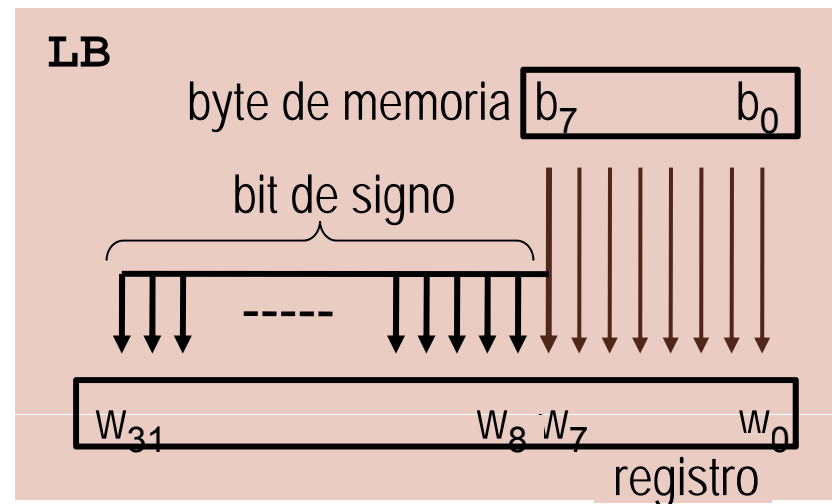
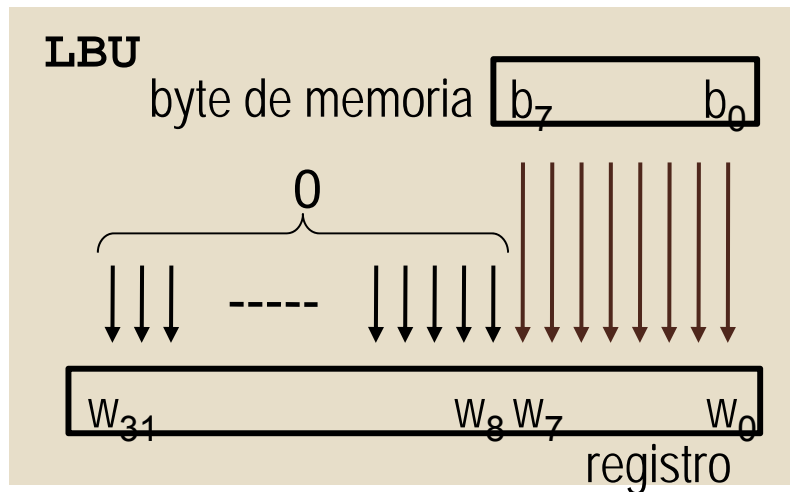
4. Representación de los enteros

- Conversión entre tipos enteros en el MIPS R2000
 - ✓ El tipo entero *nativo* es de 32 bits (ALU y registros)
 - ✓ Las instrucciones de acceso a la memoria hacen la conversión si hace falta
 - **LBU y LHU** añaden ceros por la izquierda (válido para CBN)
 - LB y LH hacen extensión de signo (válido para Ca2)
 - SB y SH eliminan bits por la izquierda



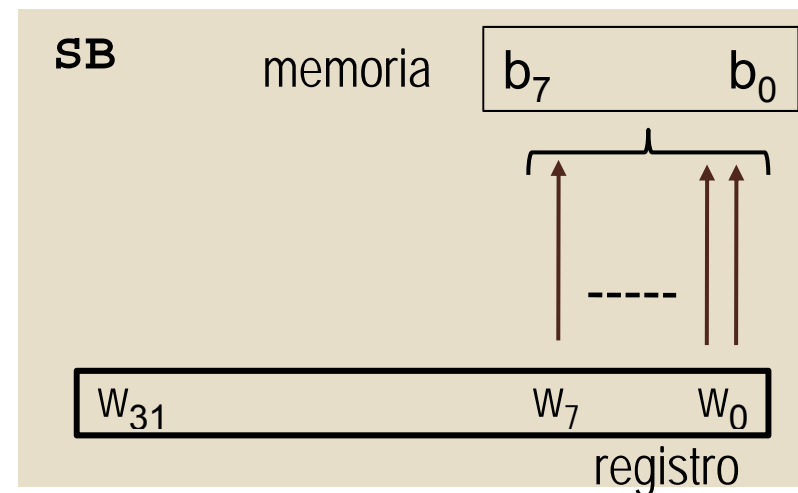
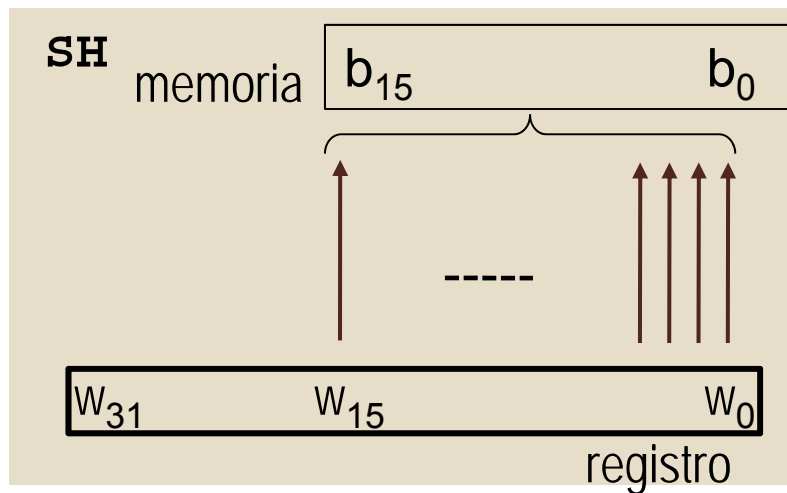
4. Representación de los enteros

- Conversión entre tipos enteros en el MIPS R2000
 - ✓ El tipo entero *nativo* es de 32 bits (ALU y registros)
 - ✓ Las instrucciones de acceso a la memoria hacen la conversión si hace falta
 - LBU y LHU añaden ceros por la izquierda (válido para CBN)
 - **LB y LH** hacen extensión de signo (válido para Ca2)
 - SB y SH eliminan bits por la izquierda



4. Representación de los enteros

- Conversión entre tipos enteros en el MIPS R2000
 - ✓ El tipo entero *nativo* es de 32 bits (ALU y registros)
 - ✓ Las instrucciones de acceso a la memoria hacen la conversión si hace falta
 - LBU y LHU añaden ceros por la izquierda (válido para CBN)
 - LB y LH hacen extensión de signo (válido para Ca2)
 - **SB y SH** eliminan bits por la izquierda, toman la parte de menor peso para escribir.



4. Representación de los enteros

El salto condicional en el MIPS

- ✓ Por comparación entre dos operandos:

(BEQ/BNE r_A, r_B, eti)

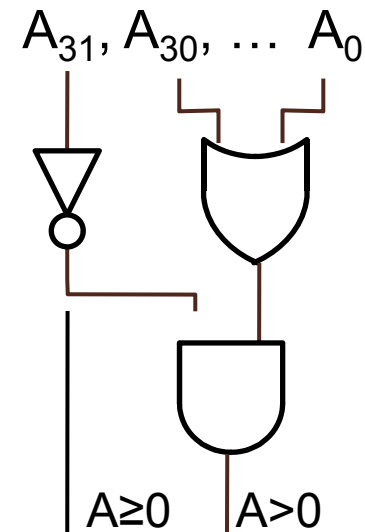
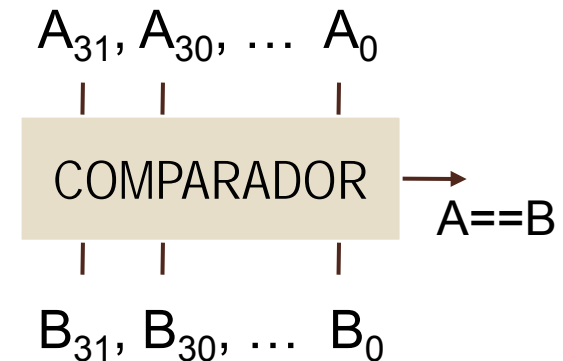
Instrucción	Condición
BEQ	$A = B$
BNE	$A \neq B$

- Las pseudoinstrucciones
BEQZ/BNEZ r, eti
derivan de BEQ y BNE

- ✓ MIPS: Basadas en el análisis de un operando:

Instrucción	Condición
BGEZ	$A_{31} = 0$
BGTZ	$A_{31} = 0$ y $A_{30} \dots A_0 \neq 0$
BLTZ	$A_{31} = 1$
BLEZ	$A_{31} = 1$ o $A_{31} \dots A_0 = 0$

Algunos operadores de cálculo de condición de salto



Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

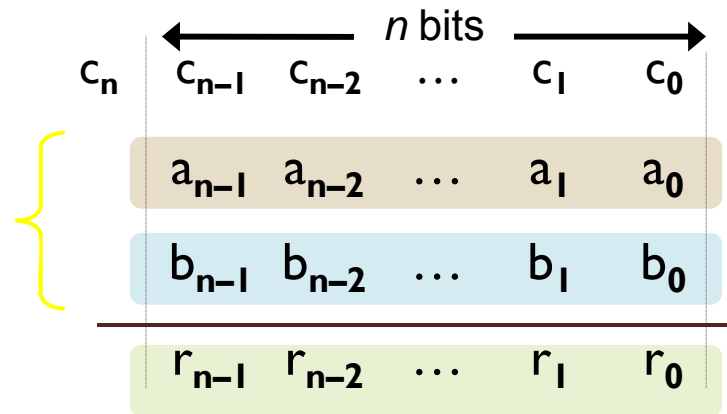
- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Fundamentos

• Anatomía de la suma

- ✓ El procedimiento general calcula $R = A + B + c_0$



Acarreos:

- c_0 (de entrada), habitualmente $c_0 = 0$
- $c_{n-1} \dots c_1$, forman parte del cálculo
- c_n (de salida) útil a veces

- El procedimiento de suma es el mismo para CBN y Ca2

	← 4 bits →	CBN	Ca2
	0 1 0 1	5	+5
+	1 0 0 1	+ 9	+ -7
	1 1 1 0	14	-2

Fundamentos

- El desbordamiento en la suma CBN $R = A + B$

✓ Detección:

- Desde la circuitería: El acarreo final $c_n = 1$
- Desde software: cuando $(R < A)$ o $(R < B)$

✓ Ejemplos con $n = 4$ bits Rango +0 hasta 15

CBN: $c_n \rightarrow$

1	1	1	0	0	
	0	1	1	0	6
+	1	1	1	0	14
<hr/>					
	0	1	0	0	4

$c_n \rightarrow$

1	0	0	0	0	
	1	0	0	0	8
+	1	0	0	0	8
<hr/>					
	0	0	0	0	0

Fundamentos

• El desbordamiento en la suma Ca2 $R = A + B$

✓ Detección:

- Desde software: sólo puede darse si los signos de los sumandos son iguales. Y se produce cuando es distinto del signo del resultado.

- Dos maneras de detectarlo desde la circuitería:

- el signo del resultado generado está invertido (como por software):

$$a_{n-1} = b_{n-1} \neq r_{n-1} \text{ o } (a_{n-1} \oplus b_{n-1}) * (b_{n-1} \oplus r_{n-1}) = 1$$

- los dos acarreos finales son distintos: $c_n \neq c_{n-1}$ o $c_n \oplus c_{n-1} = 1$

✓ Ejemplos con $n = 4$ bits Rango -8 hasta $+7$

Ca2:

$c_n \rightarrow$	0	1	0	0	0	
		0	1	1	0	+6
+		0	1	0	0	+4
<hr/>						
	1	0	1	0		-6

$c_n \rightarrow$	0	1	0	0	0	
		0	1	1	0	+6
+		0	1	0	1	+5
<hr/>						
\rightarrow	1	0	1	1		-5

Fundamentos

- Anatomía de la resta $R=A-B$ (en CBN y en Ca2)

puede hacerse como la suma $R = A + Ca2(B) = A + Cal(B) + 1$

$ \begin{array}{r} 1\ 1\ 1\ 0 \\ -\ 0\ 0\ 0\ 1 \\ \hline 1\ 1\ 0\ 1 \end{array} $	<div style="display: inline-block; text-align: center;"> <div style="margin-bottom: 5px;">CBN</div> <div style="margin-bottom: 5px;">14</div> <div style="margin-bottom: 5px;">- +1</div> <div style="border-top: 1px solid black; margin-top: 5px;">13</div> </div> <div style="font-size: 2em; color: #00AEEF; margin: 0 10px;">➡</div> <div style="display: inline-block; text-align: center;"> <div style="margin-bottom: 5px;">Ca2</div> <div style="margin-bottom: 5px;">-2</div> <div style="margin-bottom: 5px;">- +5</div> <div style="border-top: 1px solid black; margin-top: 5px;">-7</div> </div>	$ \begin{array}{r} 1\ 1\ 1\ 0 \\ -\ 0\ 1\ 0\ 1 \\ \hline 1\ 0\ 0\ 1 \end{array} $
$ \begin{array}{r} 1\ 1\ 1\ 0\ 1 \\ +\ 1\ 1\ 1\ 0 \\ \hline 1\ 1\ 0\ 1 \end{array} $		$ \begin{array}{r} 1\ 1\ 1\ 0\ 1 \\ +\ 1\ 0\ 1\ 0 \\ \hline 1\ 0\ 0\ 1 \end{array} $

Fundamentos

- El desbordamiento en la resta CBN $R = A - B$
 - ✓ Detección: (Los préstamos de la resta aparecen invertidos respecto de los acarreos de la suma)
 - Desde la circuitería: cuando $c_n = 0$
 - Desde software: cuando $(A < B) \circ (R > A)$

0	0	0	0	1	CBN		1	1	1	1	1	CBN
	1	1	1	0	14			0	1	0	1	5
+	0	0	0	0	-15		+	1	1	1	0	-1
1 1 1 1					-1	➡	0 1 0 0					+4

Fundamentos

- Desbordamiento en la resta en Ca2 $R = A - B$

✓ Detección:

- Por software: se detecta cuando los signos de los operandos son distintos entre sí, y el signo de A es distinto del signo del resultado R.
- Desde la circuitería:
 - Acarreos finales distintos (como suma) : $c_n \neq c_{n-1}$ o $c_n \oplus c_{n-1} = 1$
 - Comprobando lo mismo que por software:

$$a_{n-1} = -b_{n-1} \neq r_{n-1} \text{ o } (a_{n-1} \oplus b_{n-1}) * (a_{n-1} \oplus r_{n-1}) = 1$$

0	1	1	1	1	Ca2		1	1	0	1	Ca2
	0	1	1	0	6			1	1	1	-2
+	0	0	1	1	-4		+	1	0	1	+5
<hr/>					<hr/>		<hr/>				<hr/>
	1	0	1	0	-6			1	0	0	-7

Fundamentos

- La comparación de dos enteros CBN:
 - ✓ El resultado de una comparación es un bit
(1 = cierto, 0 = falso)
 - ✓ La comparación $A < B$ se puede hacer restando ($R = A - B$) y analizando los acarreos y el signo de R
 - el valor concreto de R es irrelevante

Comparación	CBN
$A == B$	$R = 0$
$A \geq B$	$c_n = 1$ (R es representable)
$A < B$	$c_n = 0$ (R no es representable)

Fundamentos

- La comparación de dos enteros en Ca2:
 - ✓ El resultado de una comparación es un bit
(1 = cierto, 0 = falso)
 - ✓ La comparación $A < B$ se puede hacer restando ($R = A - B$) y viendo si ha habido desbordamiento, bit V y analizando el signo del resultado

Comparación	Ca2	
$A == B$	$R = 0$	$R = 0$
$(V = 0)$ (R es representable)	R es positivo $A \geq B$	R es negativo $A < B$
$(V = 1)$ (R no es representable)	R es positivo $A < B$	R es negativo $A > B$

Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Suma y resta con el MIPS R2000

- Instrucciones

- ✓ Operandos de 32 bits

- ✓ Versiones registro-registro (formato R) o registro-inmediato (formato I)

- ✓ Instrucciones aditivas:

operación	formato	con signo	sin signo
suma	R	ADD	ADDU
suma	I	ADDI	ADDIU
resta	R	SUB	SUBU
comparación	R	SLT	SLTU
comparación	I	SLTI	SLTIU

- ✓ **ADD** y **ADDU** (etc...) hacen la misma operación, pero **ADD** puede producir excepciones

- ✓ Todas las instrucciones de formato I hacen extensión de signo de la constante

- ✓ No hay resta en formato I. Para restar la constante k se suma $-k$

Suma y resta con el MIPS R2000

- Instrucciones de suma y resta
 - ✓ **ADD *rdst*,*rfnt1*,*rfnt2***
ADDI *rdst*,*rfnt*,*imm*
 - Si hay desbordamiento (con signo), provocan una excepción y el registro ***rdst*** no se modifica
 - ✓ **ADDU *rdst*,*rfnt1*,*rfnt2***
ADDIU *rdst*,*rfnt*,*imm*
 - Nunca provocan ninguna excepción, ni detectan situaciones de desbordamiento
 - ✓ **SUB *rdst*,*rfnt1*,*rfnt2***
 - Si hay desbordamiento (con signo), provocan una excepción y el registro ***rdst*** no se modifica
 - ✓ **SUBU *rdst*,*rfnt1*,*rfnt2***
 - Nunca provocan ninguna excepción, ni detectan situaciones de desbordamiento

Suma y resta con el MIPS R2000

- Tratamiento del desbordamiento en alto nivel
 - ✓ En Java y en C la aritmética entera ignora el desbordamiento

Java

```
int a,b,c;
a = 2000000000; // 0x77359400
b = 1000000000; // 0x3B9ACA00
c = a + b;
System.out.println(a + " + " + b + " = " + c);
```



2 000 000 000 + 1 000 000 000 = -1 294 967 296

```
if ((a^b)>=0 && ((c^b)<0))
    throw new ArithmeticOverflowException; Java
```

- ✓ Al generar código para MIPS, el compilador escogerá **ADDU** en vez de **ADD**
- ✓ Para detectar el desbordamiento, habrá que añadir al programa:

Suma y resta amb el MIPS R2000

- Tratamiento del desbordamiento en bajo nivel
 - ✓ Detección de la desigualdad de los signos entre operandos

```
int a,b,c;  
c = a + b;
```

```
lw $t0,a  
lw $t1,b  
addu $t2,$t0,$t1  
xor $t3,$t0,$t1  
bltz $t3,OK  
xor $t3,$t0,$t2  
bltz $t3,AritOvException  
OK: sw $t2,c
```

Suma de a+b

Detección desbordamiento:
1. Signo a vs. signo b
2. Signo a (o b) vs. signo c

Salto para tratar el
desbordamiento

Escritura del resultado
si no ha habido desbordamiento

Suma y resta con el MIPS R2000

- Instrucciones de comparación (*Set on Less Than*)
 - ✓ **SLT *rdst*,*rfnt1*,*rfnt2***
SLTI *rdst*,*rfnt*,*inm*
 - Hace la comparación menor estricto ($rfnt1 < rfnt2$ o $rfnt < inm$) entre dos operandos, interpretándolos en **Ca2**
 - Si la condición se cumple hace $rdst=1$, en caso contrario $rdst=0$
 - Nunca generan ninguna excepción
 - ✓ **SLTU *rdst*,*rfnt1*,*rfnt2***
SLTIU *rdst*,*rfnt*,*inm*
 - Hacen la comparación ($rfnt1 < rfnt2$ o $rfnt < inm$) entre dos operandos, interpretándolos en **CBN**
 - Si la condición se cumple hace $rdst=1$, en caso contrario $rdst=0$
 - Nunca generan ninguna excepción

Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

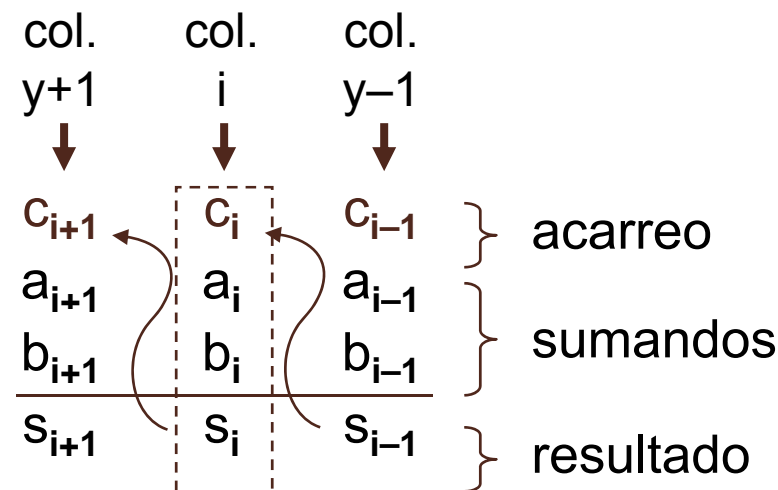
- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Operadores de suma

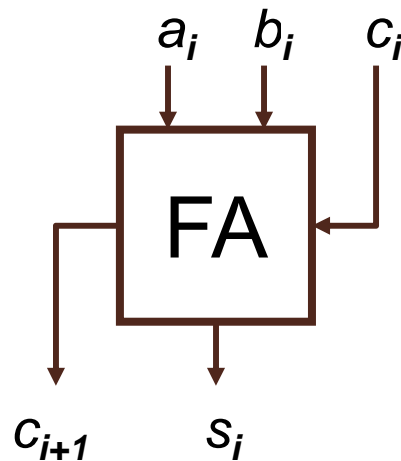
- La suma en serie

- ✓ La suma en serie reproduce el procedimiento humano de suma
- ✓ La suma se hace por orden, desde el LSB hacia el MSB
- ✓ En cada columna i , hay que sumar los bits de los sumandos a_i , b_i y el acarreo c_i que llega de la columna $i-1$ para obtener el bit del resultado s_i y generar el acarreo c_{i+1} hacia la columna $i+1$ siguiente
- ✓ Transporte de entrada $c_0 = 0$



Operadores de suma

- El sumador completo de un bit (*Full adder*):
 - ✓ Implementa los cálculos de una columna de la suma en serie
 - ✓ Admite 3 entradas de un bit y produce dos salidas:



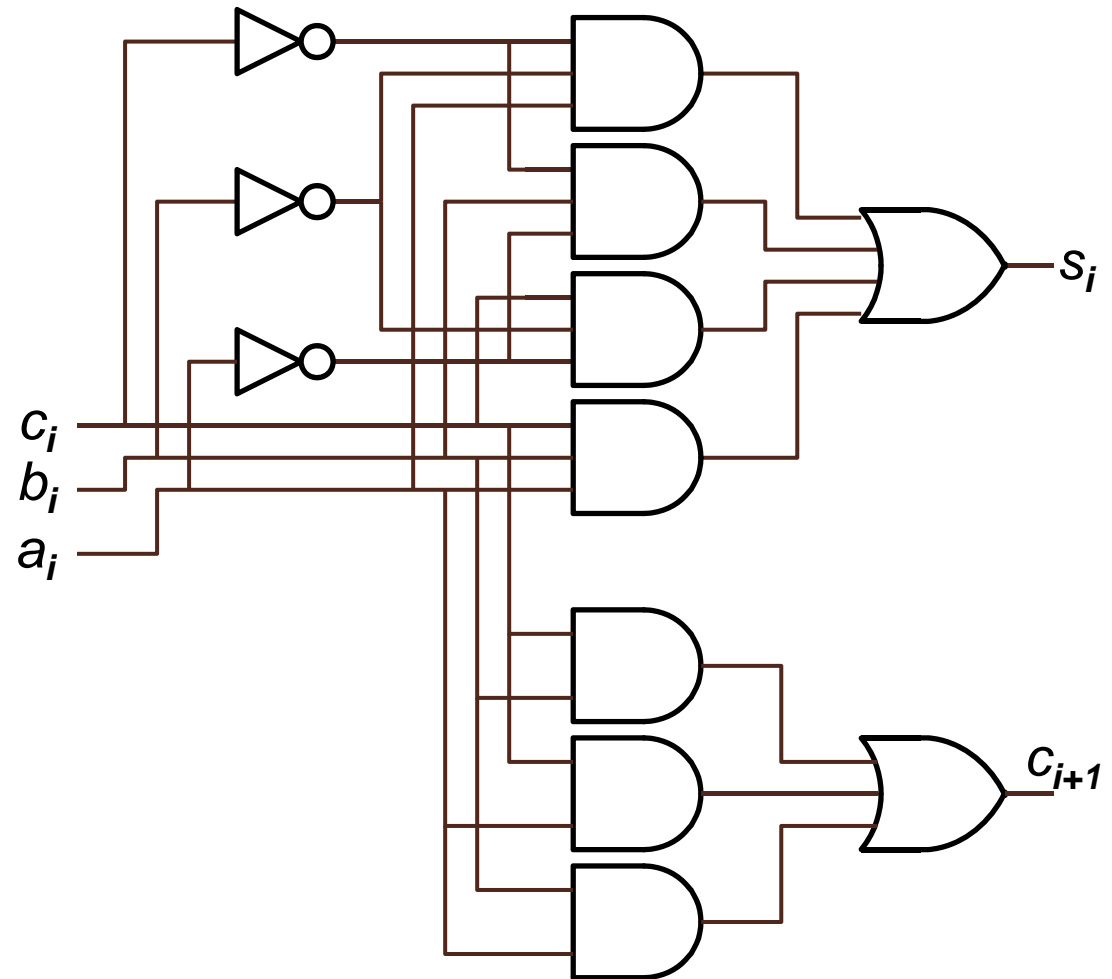
a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$s_i = \overline{a_i} \cdot \overline{b_i} \cdot c_i + \overline{a_i} \cdot b_i \cdot \overline{c_i} + a_i \cdot \overline{b_i} \cdot \overline{c_i} + a_i \cdot b_i \cdot c_i$$

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$

Operadores de suma

- El sumador completo de un bit: Implementación
 - ✓ Implementación a partir de las funciones lógicas



Operadores de suma

- El sumador completo de un bit: Prestaciones

✓ Tiempo de retardo:

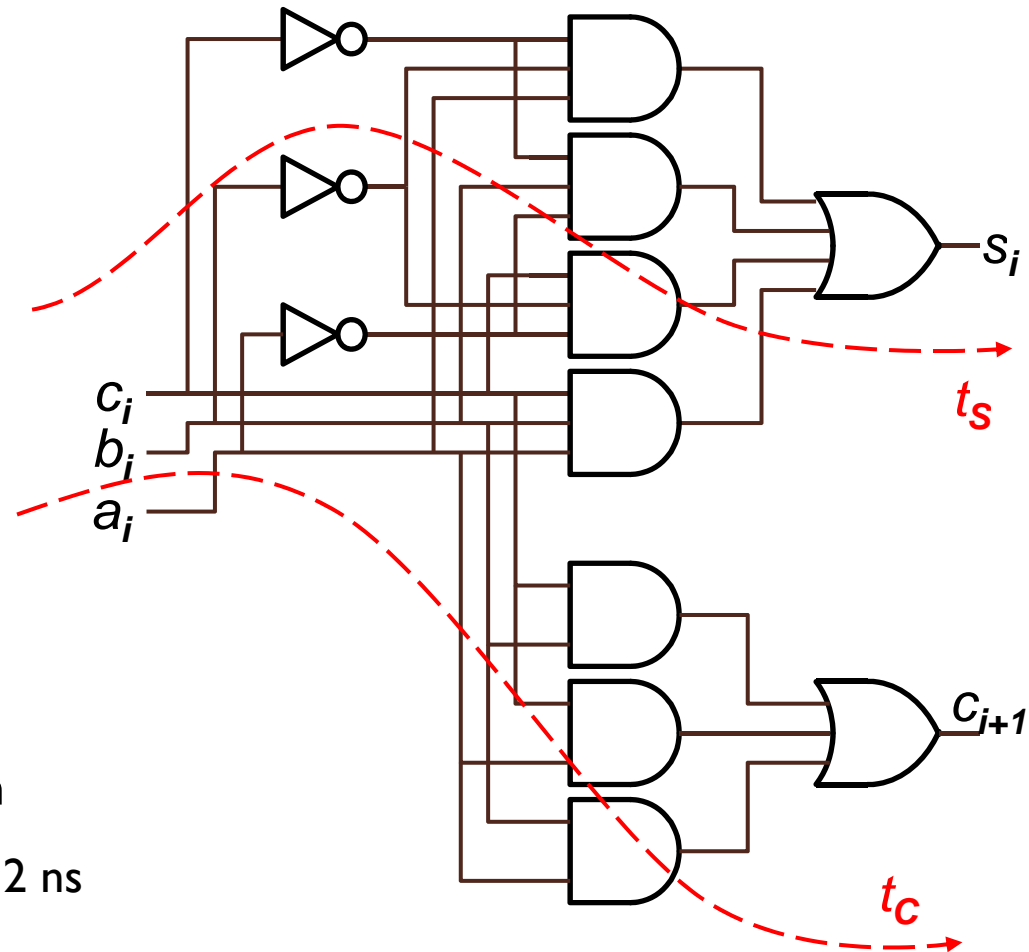
$$t_s = t_{NOT} + t_{AND} + t_{OR}$$

$$t_c = t_{AND} + t_{OR}$$

✓ Complejidad: 12 puertas

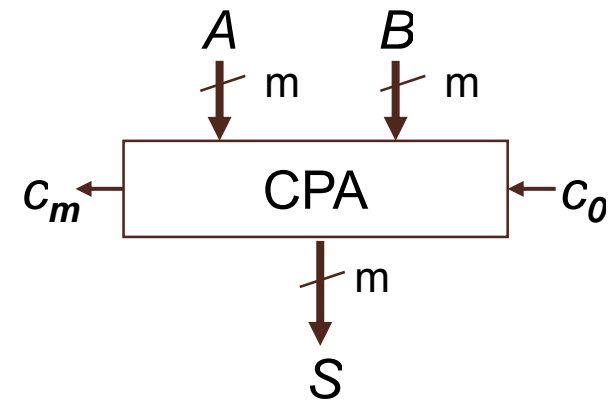
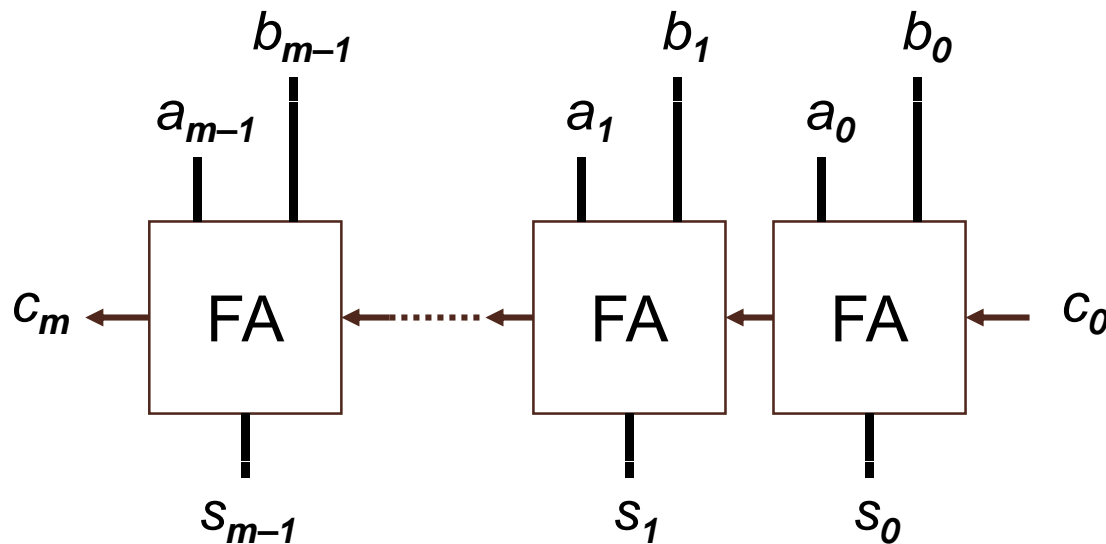
✓ Con tecnología CMOS 0.5 μm

- tiempo de respuesta: entre 1 y 2 ns
- superficie: 1000 μm^2



Operadores de suma

- El sumador serie de n bits
 - ✓ CPA (Carry Propagation Adder)
 - ✓ Par hacer un sumador de dos números de m bits, se conectan m sumadores completos en cascada
 - la salida de acarreo del sumador i -ésimo se conecta a la entrada de acarreo del sumador $i+1$ -ésimo
 - el operador resultante tiene una entrada y una salida de acarreo globales



Operadores de suma

- El sumador serie de n bits. Complejidad
 - ✓ El tiempo de cálculo de un sumador serie para operandos de n bits se puede expresar en términos de los retardos de un sumador completo:
$$t(n \text{ bits}) = (n-1) t_C + \text{máx}\{t_C, t_S\}$$
 - ✓ Asintóticamente el tiempo de cálculo es lineal, $t(n \text{ bits}) = O(n)$
 - ✓ El coste espacial también es lineal, $\text{coste}(n \text{ bits}) = O(n)$
 - ✓ La suma serie es poco eficiente para las aplicaciones propias de un procesador convencional, con $n=32$ o $n=64$ bits

Operadores de suma

- Mejoras en la suma

- ✓ Anticipación del acarreo: *CLA, Carry Lookahead Adder*

- Calcula los bits de acarreo c_i antes que los bits de suma s_i
- El retardo del cálculo de los acarreos es $O(\log(n))$

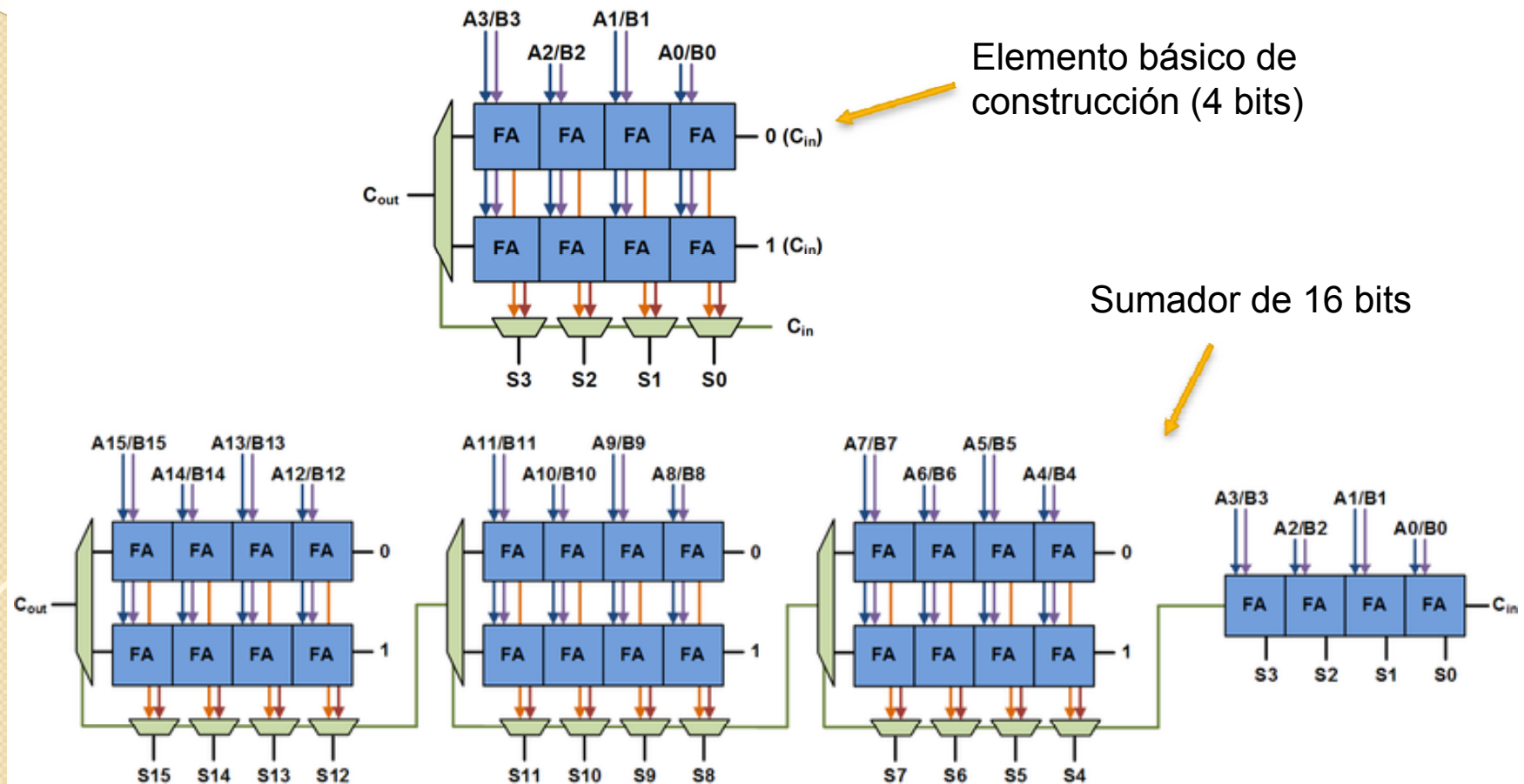
- ✓ Selección del acarreo: *CSA, Carry Select Adder*

- Divide la suma en dos partes y gestiona la parte alta con los posibles valores del acarreo proveniente de la parte baja

Operadores de suma

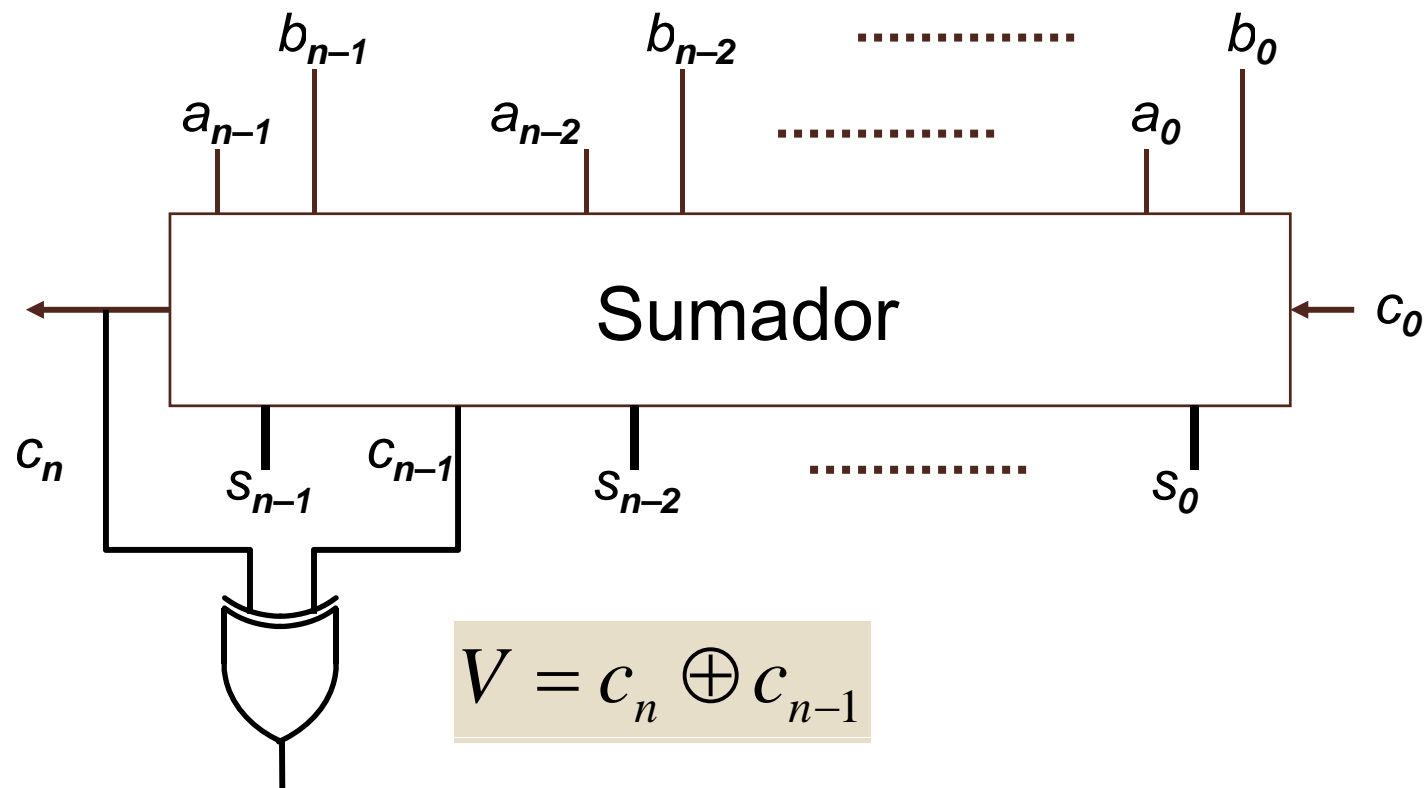
- CSA (Carry Select Adder)

- ✓ Acelera la operación de suma a base de invertir más circuitos



Operadores de suma

- Cálculo del desbordamiento en la aritmética en Ca2
 - ✓ Se detecta cuando los bits de acarreo de orden n y $n-1$ no son iguales
 - ✓ El signo del resultado s_{n-1} es incorrecto



Operadores de resta y comparación

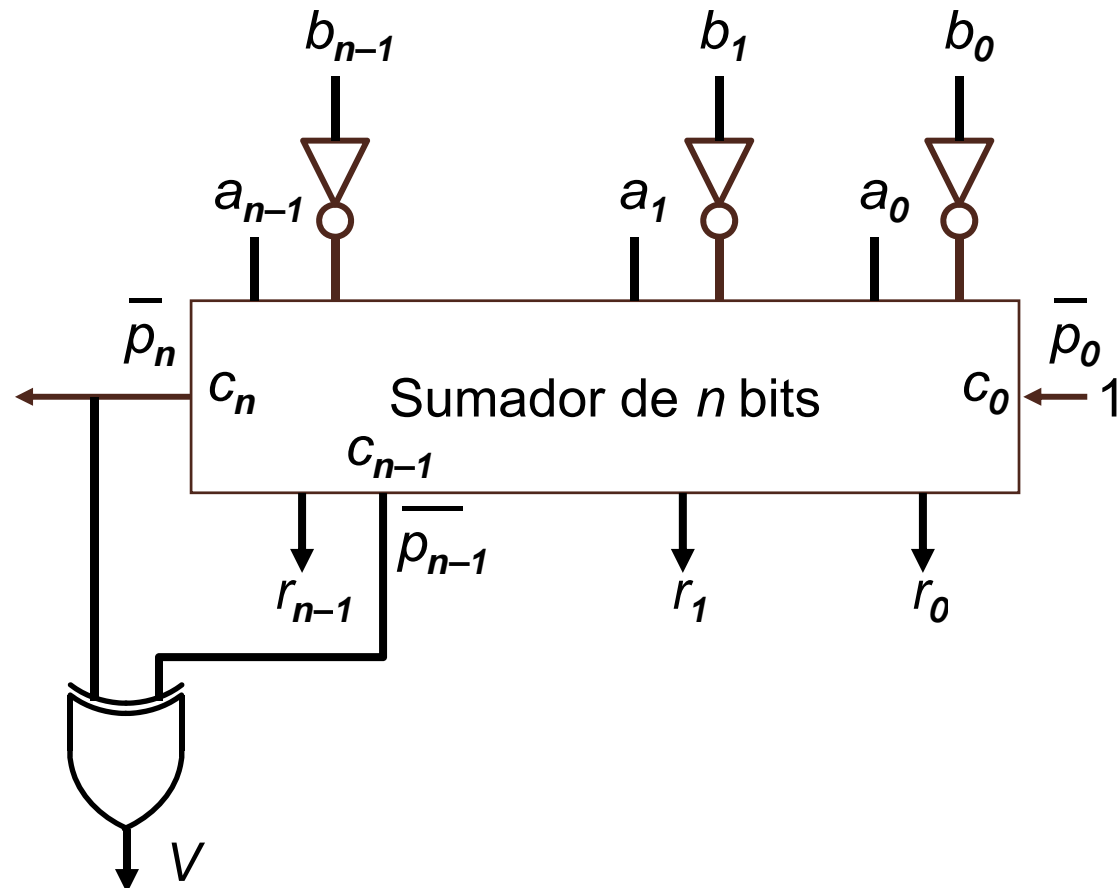
- Diseño del restador a partir del sumador

✓ $R = A - B = A + C_{a2}(B) = A + \text{not}(B) + 1$

✓ La detección de desbordamiento es igual que con la suma:

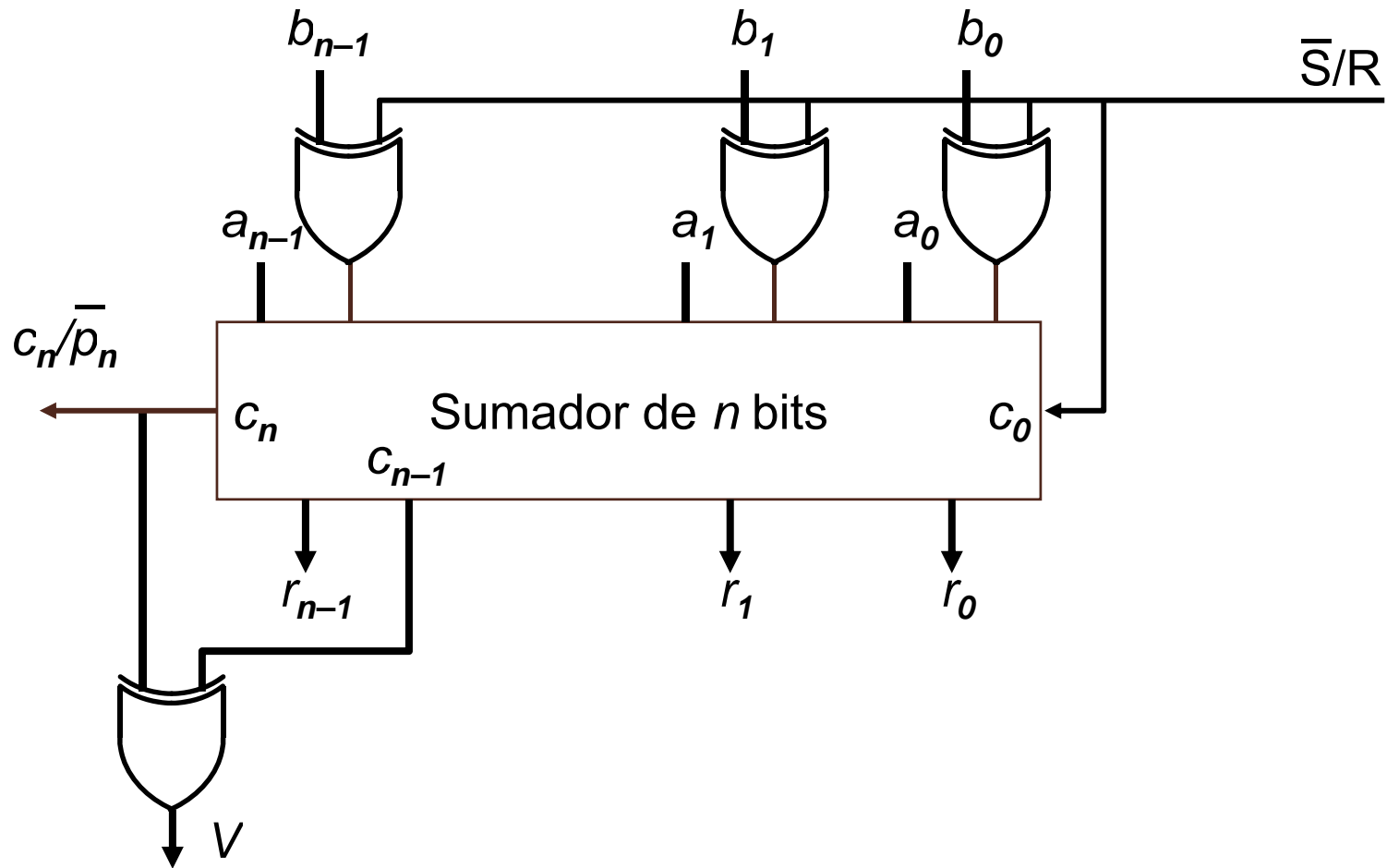
- En CBN: $p_n = 1$ ($c_n = 0$)

- En Ca2: $p_n \neq p_{n-1}$
($c_n \text{ xor } c_{n-1} = 1$)



Operadores de resta y comparación

Diseño clásico del sumador/restador



Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Fundamentos

• Desplazamientos y aritmética entera (I)

- ✓ Desplazar n bits hacia la izquierda es equivalente a multiplicar por 2^n
 - Entran n ceros por la derecha
 - Operación válida para enteros con y sin signo
 - Nombre de la operación: desplazamiento **lógico** hacia a la izquierda

$$\begin{array}{r} 0 \ 0 \ 0 \ 1 \ 1 \ 0 \quad 6 \\ \swarrow \ll 2 \searrow \\ 0 \ 1 \ 1 \ 0 \ 0 \ 0 \quad 24 \end{array} \quad \underline{\times 2^2}$$

$$\begin{array}{r} 1 \ 1 \ 1 \ 0 \ 1 \ 0 \quad -6 \\ \swarrow \ll 2 \searrow \\ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \quad -24 \end{array} \quad \underline{\times 2^2}$$

```
int a = -12;  
int b = a << 3;  
System.out.println(a + " * 8 = " + b);
```

Java

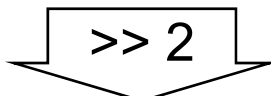
```
-12 * 8 = -96
```

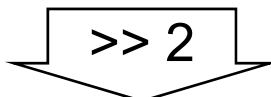
Fundamentos

• Desplazamientos y aritmética entera (II)

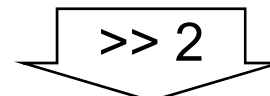
- ✓ Desplazar n bits hacia la derecha es equivalente a dividir por 2^n
 - Enteros sin signo: entran n ceros por la izquierda
 - Enteros con signo: el bit de signo se replica n veces

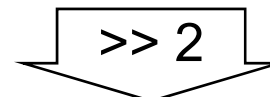
Sin signo:
desplazamiento **lógico**

0	1	1	0	0	1	25
						$/ 2^2$
0	0	0	1	1	0	6

1	0	1	0	0	0	40
						$/ 2^2$
0	0	1	0	1	0	10

Con signo:
desplazamiento **aritmético**

0	1	1	0	0	1	+25
						$/ 2^2$
0	0	0	1	1	0	6

1	0	1	0	0	0	-24
						$/ 2^2$
1	1	1	0	1	0	-6

Fundamentos

- Los compiladores evitan las operaciones de multiplicación siempre que es posible

```
int a,b,c,d;  
a = a*2;      //  $2=2^1$   
b = b*8;      //  $8=2^3$   
c = c*1024;   //  $1024=2^{10}$   
d = d*5       //  $5=2^2+1$ 
```

lw \$s0, a
lw \$s1, b
lw \$s2, c
lw \$s3, d
add \$s0, \$s0, \$s0
sll \$s1, \$s1, 3
sll \$s2, \$s2, 10
sll \$t0, \$s3, 2
add \$s3, \$s3, \$t0
sw \$s0, a
sw \$s1, b
...

Fundamentos

- Anatomía de la multiplicación sin signo

- ✓ En general, para representar el producto de dos números de n bits hacen falta $2n$ bits
- ✓ El procedimiento *humano* se basa en sumas y desplazamientos

				1	1	0	1	(13 ₁₀)
			×	1	0	0	1	(9 ₁₀)
				1	1	0	1	
				0	0	0	0	0
		0	0	0	0	0	0	
	1	1	0	1	0	0	0	
0	1	1	1	0	1	0	1	
2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰	
Pesos								

Notación

M = Multiplicando; m_i = bit i-ésimo

Q = Multiplicador; q_i = bit i-ésimo

P = Producto; p_i = bit i-ésimo

$$P = M \times Q = \sum_{i=0}^{n-1} Mq_i 2^i$$

$$(117_{10}) = 1101_2 \times (1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)$$

Fundamentos

- Problemática de la multiplicación con signo
 - ✓ Hay que extender el signo del multiplicando
 - ✓ El peso del bit de signo del multiplicador es de -2^{n-1} (y no 2^{n-1})
 - Conviene buscar codificaciones alternativas

1	1	1	1	1	1	0	1
			×	1	0	0	1
<hr/>							
1	1	1	1	1	1	0	1
			0	0	0	0	0
		0	0	0	0	0	0
	0	0	1	1	0	0	0
<hr/>							
0	0	0	1	0	1	0	1
<hr/>							
2 ⁷	2 ⁶	2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
Pesos							

(-3₁₀)

(-7₁₀)

$$P = M \times Q = \sum_{i=0}^{n-2} Mq_i 2^i - Mq_{n-1} 2^{n-1}$$

$$(+21_{10}) = 11111101_2 \times (-1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)$$

Fundamentos

La división

- ✓ Dados un dividendo y un divisor, produce dos resultados: el cociente y el resto (o módulo)
- ✓ Ejemplo sin signo: algoritmo humano de restas y desplazamientos
 - Si no cabe (✗): 0 al cociente
 - Si cabe (✓): restar y 1 al cociente

$$\begin{array}{r}
 1101 \\
 - 101 \\
 \hline
 \checkmark 0011 \\
 \times - 0101 \\
 \hline
 0011 \text{ Resto final}
 \end{array}
 \qquad
 \begin{array}{r}
 0101 \\
 \hline
 10
 \end{array}$$

- ✓ Con signo: por convención, el resto tiene el mismo signo que el divisor

Fundamentos

La multiplicación y la división en alto nivel

✓ El módulo

```
int x,y,z,t;
x = 13;
y = 5;
z = x/y;
t = x%y;
System.out.println(x + " = " + y + "*" + z + " + " + " + t);
```

Java

13 = 5*2 + 3

```
int x,y,z; Java,C
```

✓ |

```
x = 0;
y = 1;
z = y/x;
```

Exception in thread "main"
java.lang.ArithmeticException: / by
zero at ...

Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Multiplicación y división en el MIPS

- Instrucciones de desplazamiento
 - ✓ Son de la forma “operación Rr, Ri, Long”, donde Long puede ser una constante o un registro
 - ✓ El desplazamiento máximo es de 31 posiciones. Sólo cuentan los 5 bits de menor peso de Long

tipo	formato R	
izquierda	<code>sll rd,rt,inm</code>	<code>sllv rd,rs,rt</code>
derecha (lógico)	<code>srl rd,rt,inm</code>	<code>srlv rd,rs,rt</code>
derecha (aritmética)	<code>sra rd,rt,inm</code>	<code>srav rd,rs,rt</code>

Multiplicación y división en el MIPS

- Instrucciones de multiplicación y división generales

- ✓ Dos registros especiales de 32 bits: HI y LO

- Combinados forman un registro de 64 bits

- ✓ Operaciones

`mult $2, $3:` HI-LO \leftarrow $\$2 * \3 ; Operandos con signo

`multu $2, $3:` HI-LO \leftarrow $\$2 * \3 ; Operandos positivos sin signo

`div $2, $3:` LO \leftarrow $\$2 / \3 ; HI \leftarrow $\$2 \bmod \3 ; Con signo

`divu $2, $3:` LO \leftarrow $\$2 / \3 ; HI \leftarrow $\$2 \bmod \3 ; Sin signo

- ✓ Transferencia de resultados

- `mfhi $2:` $\$2 \leftarrow$ HI

- `mflo $2:` $\$2 \leftarrow$ LO

Multiplicación y división en el MIPS

- Instrucciones de multiplicación y división generales
 - ✓ Hay pseudoinstrucciones que permiten almacenar el resultado en un registro destinatario de propósito general y multiplicar por constantes
 - ✓ Ninguna de estas instrucciones comprueba desbordamientos o división por cero: hay que hacerlo por software

Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

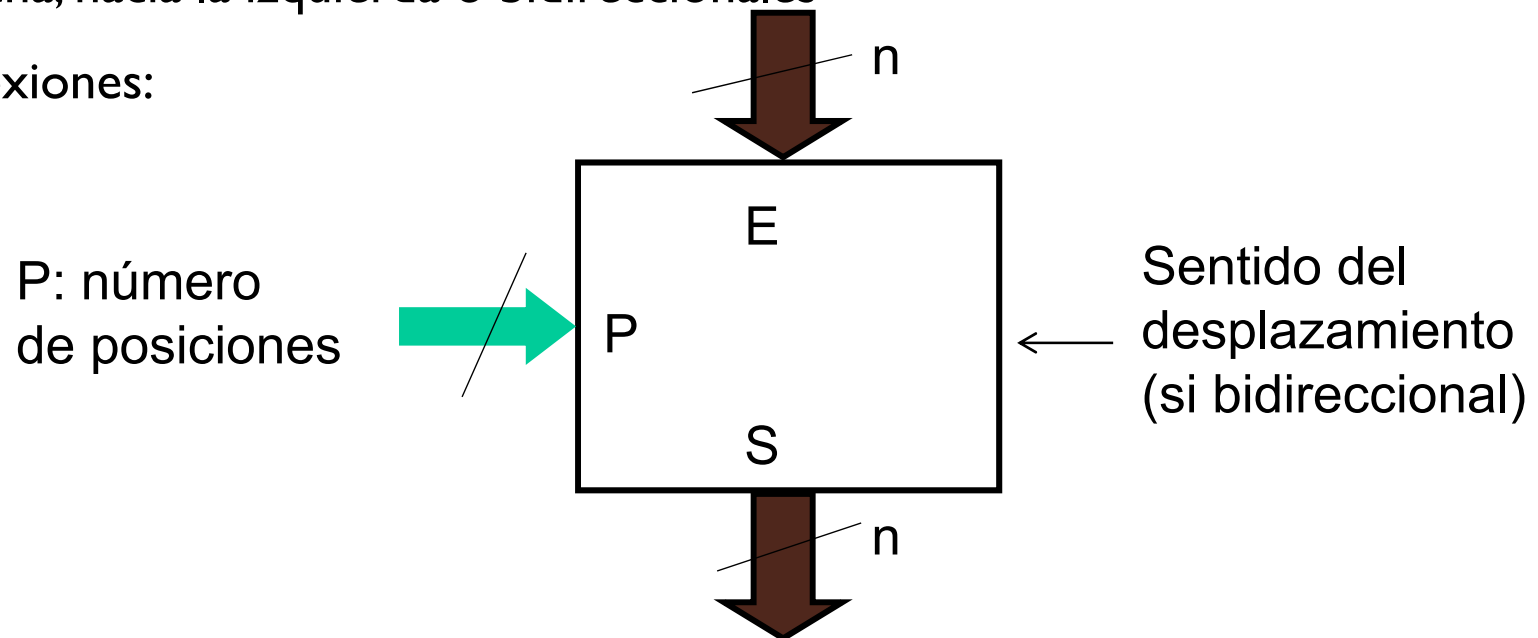
- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Operadores de desplazamiento

El Barrel Shifter

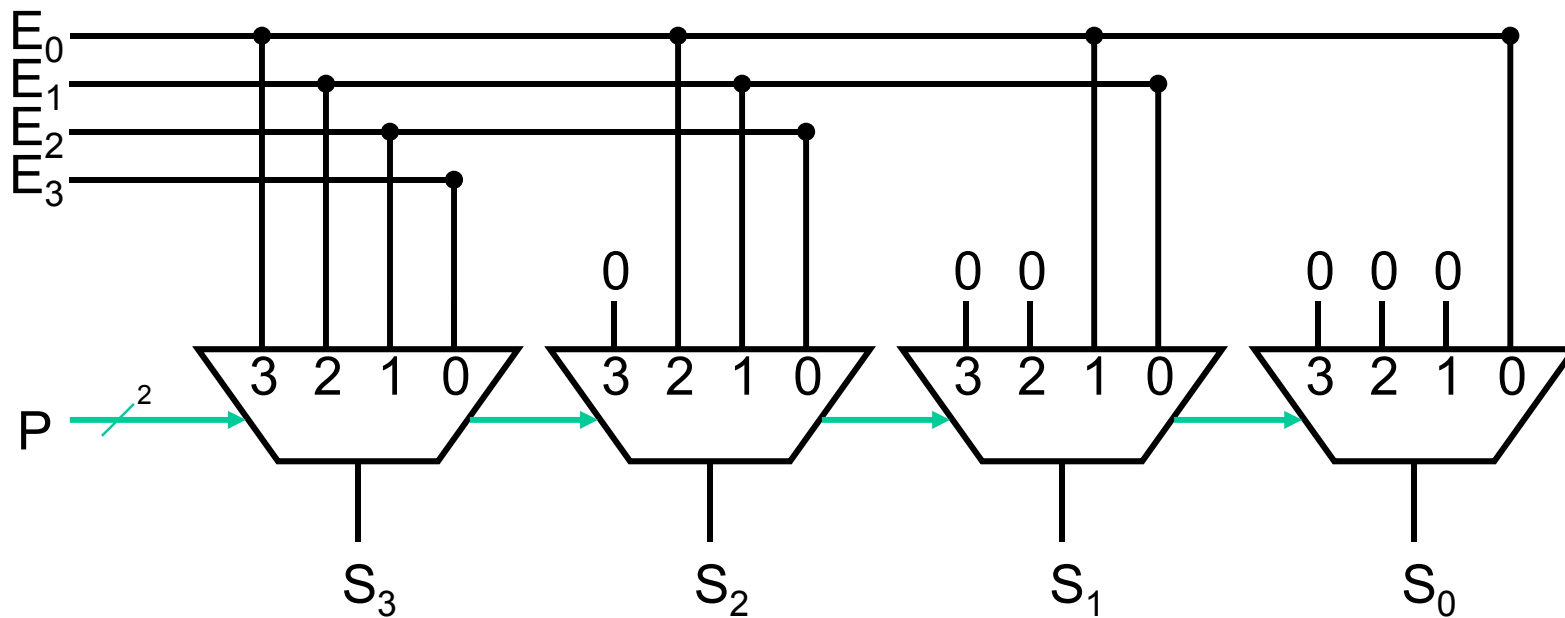
- ✓ Un Barrel Shifter es un circuito que permite realizar desplazamientos variables sobre datos de n bits
- ✓ Puede implementarse mediante multiplexores
- ✓ Dependiendo del diseño, hace desplazamientos lógicos o aritméticos hacia la derecha, hacia la izquierda o bidireccionales
- ✓ Conexiones:



Operadores de desplazamiento

Barrel shifter: ejemplo de diseño

- ✓ Implementación de un operador de desplazamiento lógico hacia la izquierda (sll) para datos de 4 bits



Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Operadores de multiplicación sin signo

- Operadores secuenciales aritméticos

- ✓ Son circuitos secuenciales síncronos que hacen una operación dada
- ✓ Necesitan un cierto número de ciclos de reloj para hacer la operación
- ✓ El ciclo de reloj se ajusta para que puedan actuar los circuitos
- ✓ Si un operador necesita n ciclos de t segundos para una operación,
 - el tiempo de operación será $T = n \times t$
 - la productividad será $P = f/n$, donde $f = 1/t$ es la frecuencia de trabajo del reloj

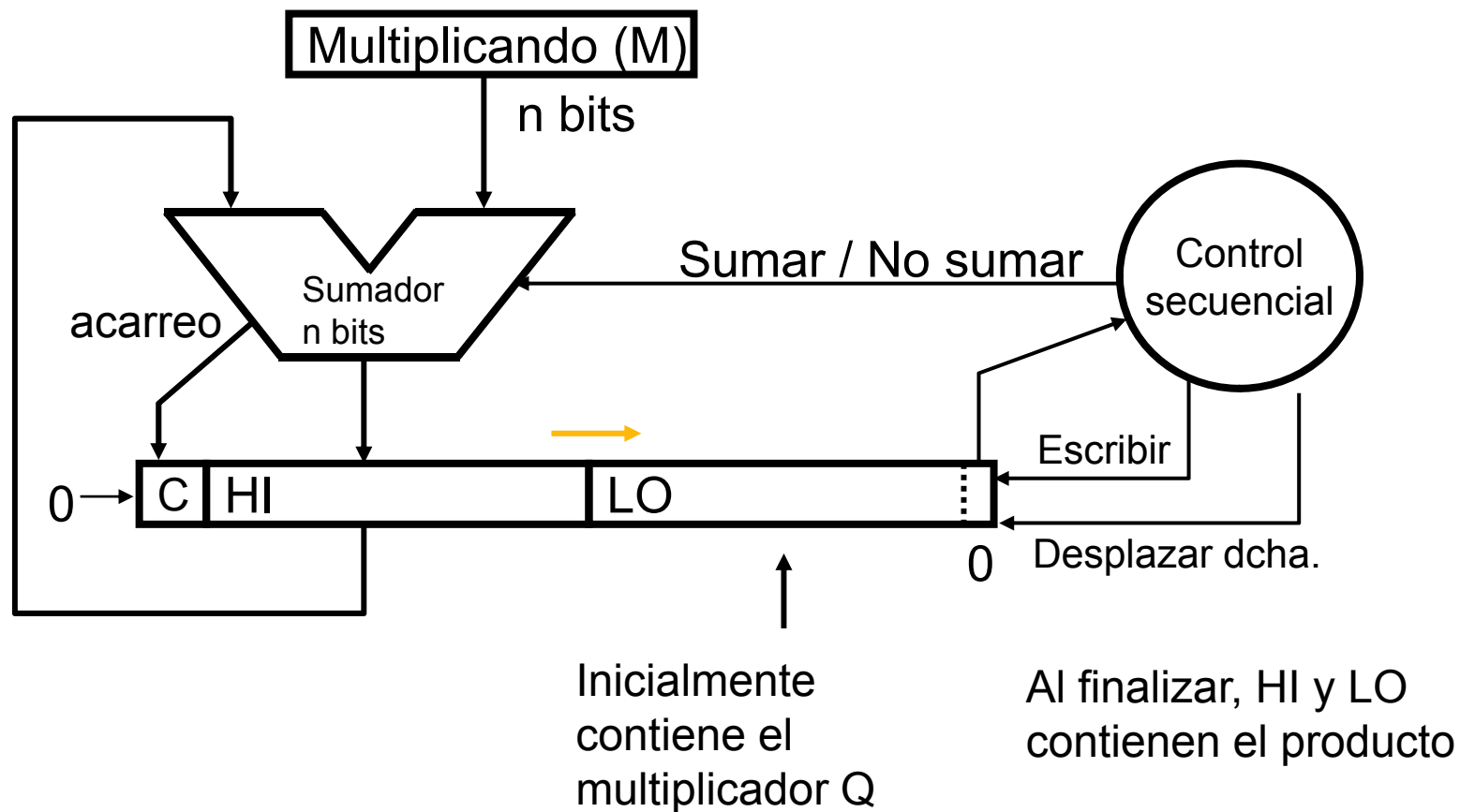
- Notación:

- ✓ M = Multiplicando; m_i = bit i -ésimo de M
- ✓ Q = Multiplicador; q_i = bit i -ésimo de Q
- ✓ P = Producto; p_i = bit i -ésimo del producto
- ✓ n = Número de dígitos de los operandos M y Q (de 0 a $n-1$)

Operadores de multiplicación sin signo

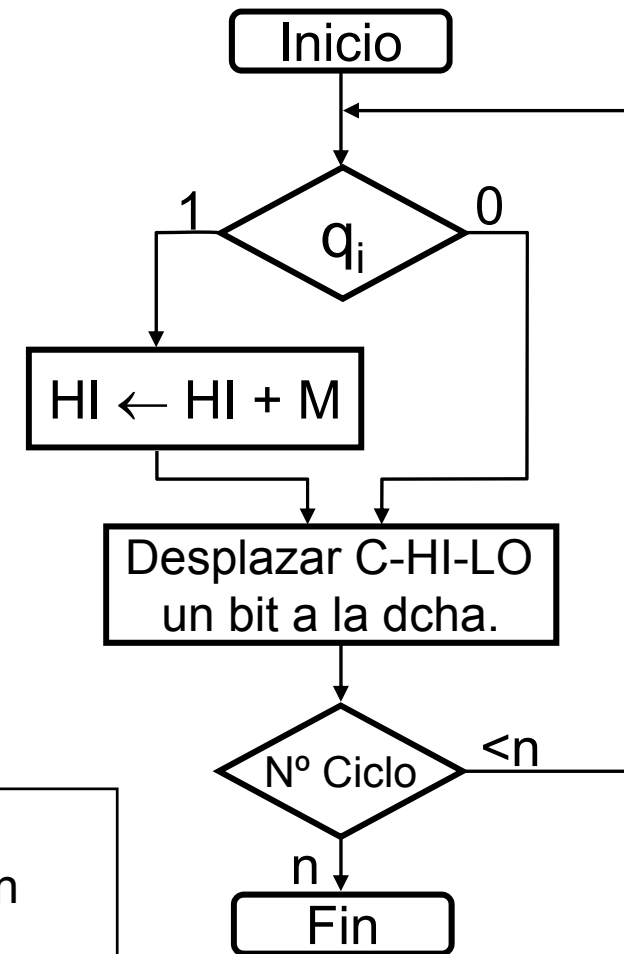
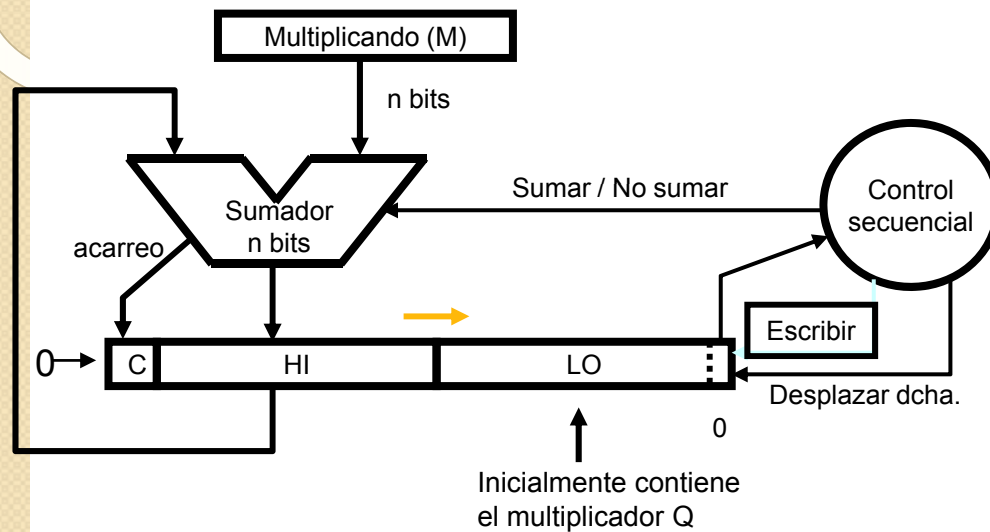
Operador para el algoritmo de sumas y desplazamientos

- ✓ M y Q de n bits; P de 2n bits



Operadores de multiplicación sin signo

- ## Algoritmo con que funciona el operador secuencial



- El algoritmo requiere n ciclos
- En cada ciclo hay que hacer hasta una suma y un desplazamiento

Operadores de multiplicación sin signo

Ejemplo: $n=4$; $M=1011_2$; $Q=0101_2$ ($11_{10} \times 5_{10} = 55_{10}$)

Ciclo	Acción	C-HI-LO
0	Valores iniciales	0 0000 <u>0101</u>
1	$HI \leftarrow HI + M$	0 1011 0101
	Desplazar C-HI-LO 1 bit a la derecha	0 0101 <u>1010</u>
2	No sumar	0 0101 1010
	Desplazar C-HI-LO 1 bit a la derecha	0 0010 <u>1101</u>
3	$HI \leftarrow HI + M$	0 1101 1101
	Desplazar C-HI-LO 1 bit a la derecha	0 0110 <u>1110</u>
4	No sumar	0 0110 1110
	Desplazar C-HI-LO 1 bit a la derecha	0 <u>0011</u> 0111

Operadores de multiplicación sin signo

- Ejercicio: $n=4$; $M=1101_2$; $Q=1011_2$ ($13_{10} \times 11_{10} = 143_{10}$)

Ciclo	Acción	C-HI-LO
0	Valores iniciales	0 0000 <u>1011</u>
1		
2		
3		
4		

Solución: 1000 1111

Índice

- *Introducción*

1. *Típos en alto y bajo nivel*
2. *Operaciones y operadores*
3. *Operaciones lógicas*
4. *La representación de los enteros*

- *Suma y resta de enteros*

1. *Suma y resta en el MIPS R2000*
2. *Operadores de suma*
3. *Operadores de resta*

- *Multiplicación de enteros*

1. *Fundamentos*
2. *Multiplicación y división en el MIPS*
3. *Operadores de desplazamiento*
4. *Operadores de multiplicación sin signo*
5. *Operadores de multiplicación con signo*

Operadores de multiplicación con signo

- Tratamiento del signo por separado
 - ✓ Se trata de multiplicar los valores absolutos y considerar el signo aparte. Considerando que $\text{Signo}(X)$ es el bit de signo de X :

```
Signo_Prod ← Signo(M) XOR Signo(Q);  
si M < 0 entonces M ← -M; fin si;  
si Q < 0 entonces Q ← -Q; fin si;  
P ← M × Q;  
si Signo_Prod = 1 entonces P ← -P; fin si;
```

Operadores de multiplicación con signo

- Tratamiento del signo por separado

```
Signo_Prod ← Signo(M) XOR Signo(Q);  
si M < 0 entonces M ← -M; fin si;  
si Q < 0 entonces Q ← -Q; fin si;  
P ← M × Q;  
si Signo_Prod = 1 entonces P ← -P; fin si;
```

- Inconvenientes
 - ✓ Requiere cierto hardware adicional para el caso particular de números con signo, a fin de complementar M, Q o P
 - ✓ Existen otros métodos para tratar uniformemente el producto de números con o sin signo (algoritmo de Booth, más adelante)

Operadores de multiplicación con signo

- Sumas y desplazamientos con extensión de signo
 - ✓ El algoritmo 2 puede funcionar con signo **sólo si Q es positivo**
 - ✓ Para ello, hay que extender el signo de los productos intermedios
 - ✓ Ejemplo: $n = 4$; $M = -3$; $Q = 6$; Representados en Ca2

$$\begin{array}{r}
 1\ 1\ 0\ 1\ (-3_{10}) \\
 \times 0\ 1\ 1\ 0\ (6_{10}) \\
 \hline
 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \\
 1\ 1\ 1\ 1\ 1\ 0\ 1 \\
 1\ 1\ 1\ 1\ 0\ 1 \\
 0\ 0\ 0\ 0\ 0 \\
 \hline
 1\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ (-18_{10})
 \end{array}$$

Para funcionar en cualquier caso, pueden procesarse los signos de M y Q de antemano:

```

si Q < 0 entonces
    Q ← -Q;
    M ← -M;
fin si
P ← M × Q;
  
```

Aunque es más sencillo que el anterior, también requiere procesar por separado los casos de multiplicación de números con o sin signo y complementar M y Q si $Q < 0$

Algoritmo de Booth

- ✓ Consiste en recodificar el multiplicador como una suma de potencias **positivas o negativas** de la base: usa dígitos 0, +1 y -1
- ✓ Por ejemplo:
 - El número 30 puede expresarse como $(32 - 2)$
 - $30_{10} = 0011110_2 = 0 + 1\ 0\ 0\ 0 - 1\ 0_{\text{Booth}} = -1 \cdot 2^1 + 1 \cdot 2^5$

							0	1	0	1	1	0	1	(45 ₁₀)
						×	0	+1	0	0	0	-1	0	(30 _{Booth})
<hr/>														
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	1	1	1	1	1	1	0	1	0	0	1	1		← Ca2 de M (restar M)
0	0	0	0	0	0	0	0	0	0	0	0			
0	0	0	0	0	0	0	0	0	0	0				
0	0	0	0	0	0	0	0	0	0					
0	0	0	1	0	1	1	0	1						← Copia de M (sumar M)
0	0	0	0	0	0	0	0							
<hr/>														
0	0	0	1	0	1	0	1	0	0	0	1	1	0	(1350 ₁₀)

Operadores de multiplicación con signo

• Algoritmo de Booth

- ✓ Funciona con números positivos o negativos:
 - Multiplicación sin signo: suponer un bit de signo de $M = 0$
 - Multiplicación con signo: extender el MSB de M como signo
- ✓ Ejemplo: Multiplicar con y sin signo los números 1101_2 y $0+10-1_{Booth}$

Sin signo

Signo positivo

implícito	→	0	1	1	0	1	(13 ₁₀)
		×	0	+1	0	-1	(3 _{Booth})
		<hr/>					
1	1	1	1	0	0	1	1
0	0	0	0	0	0	0	
0	0	1	1	0	1		
0	0	0	0	0			
		<hr/>					
0	0	1	0	0	1	1	1 (39 ₁₀)

Con signo

Signo

implícito →

1

1

1

0

1

(-3₁₀)

×

0

+1

0

-1

(3_{Booth})

0

0

0

0

0

0

1

1

0

0

0

0

0

0

0

1

1

1

1

0

1

0

0

0

0

0

1

1

1

1

0

1

1

1

(-9₁₀)

Operadores de multiplicación con signo

- Recodificación del multiplicador por el método de Booth
 - ✓ Deben considerarse parejas de bits correlativos, de dcha. a izda.
 - ✓ Debe suponerse un bit implícito = 0 a la derecha del LSB
 - ✓ Debe aplicarse la siguiente tabla de conversión:

q_i	q_{i-1}	Dígito Booth
0	0	0
0	1	+1
1	0	-1
1	1	0

Para recordar:
 $\text{Dígito Booth} = q_{i-1} - q_i$

Operadores de multiplicación con signo

- Recodificación del multiplicador por el método de Booth

✓ Ejemplo: Obténgase el código Booth de **1110 0111 0011** (-397_{10})

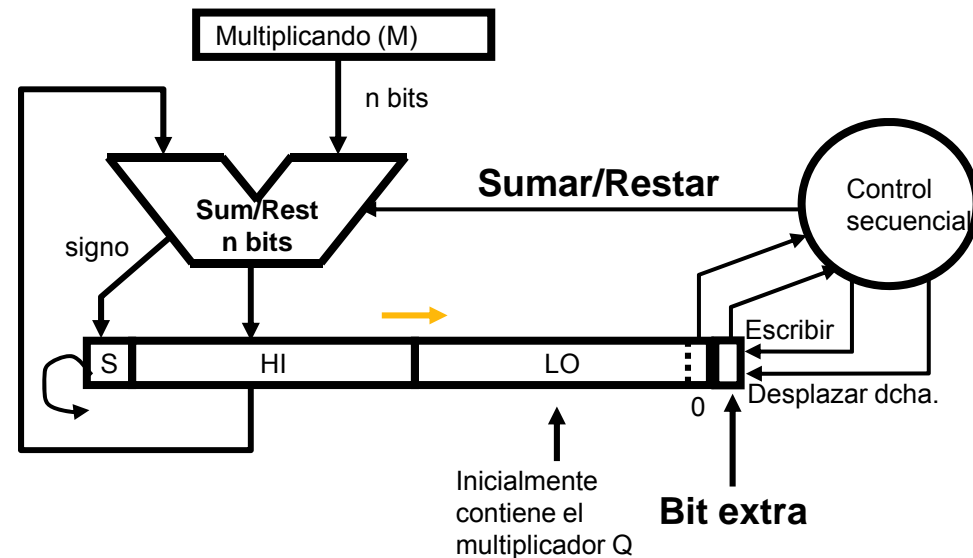
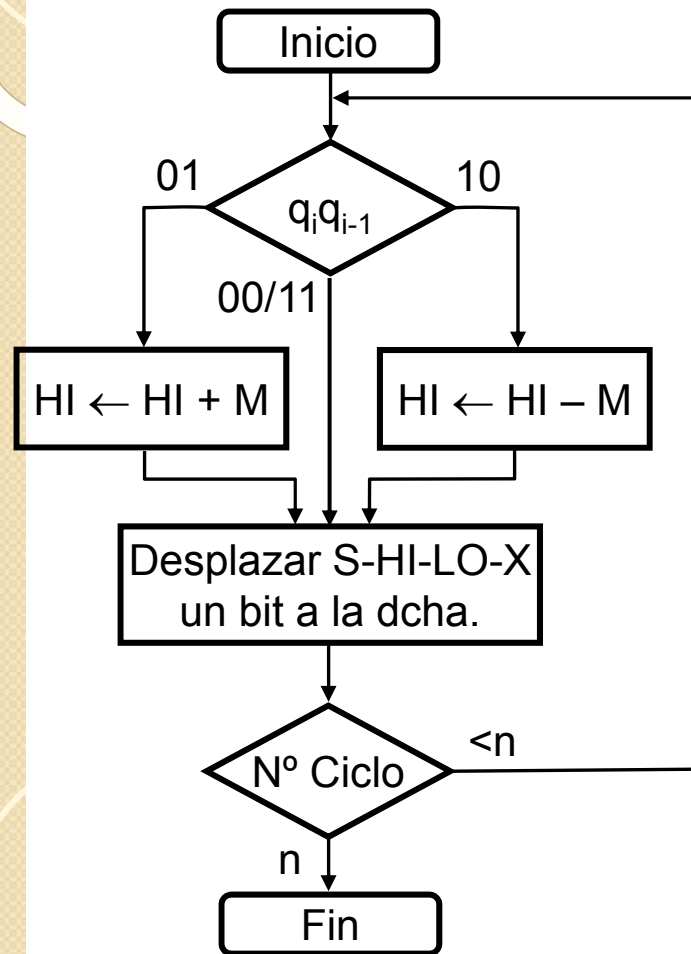
• Solución: 0 0 -1 0 +1 0 0 -1 0 +1 0 -1 =

$$-1 \times 2^0 + 1 \times 2^2 - 1 \times 2^4 + 1 \times 2^7 - 1 \times 2^9 =$$

$$-1 + 4 - 16 + 128 - 512 = -397$$

Operadores de multiplicación con signo

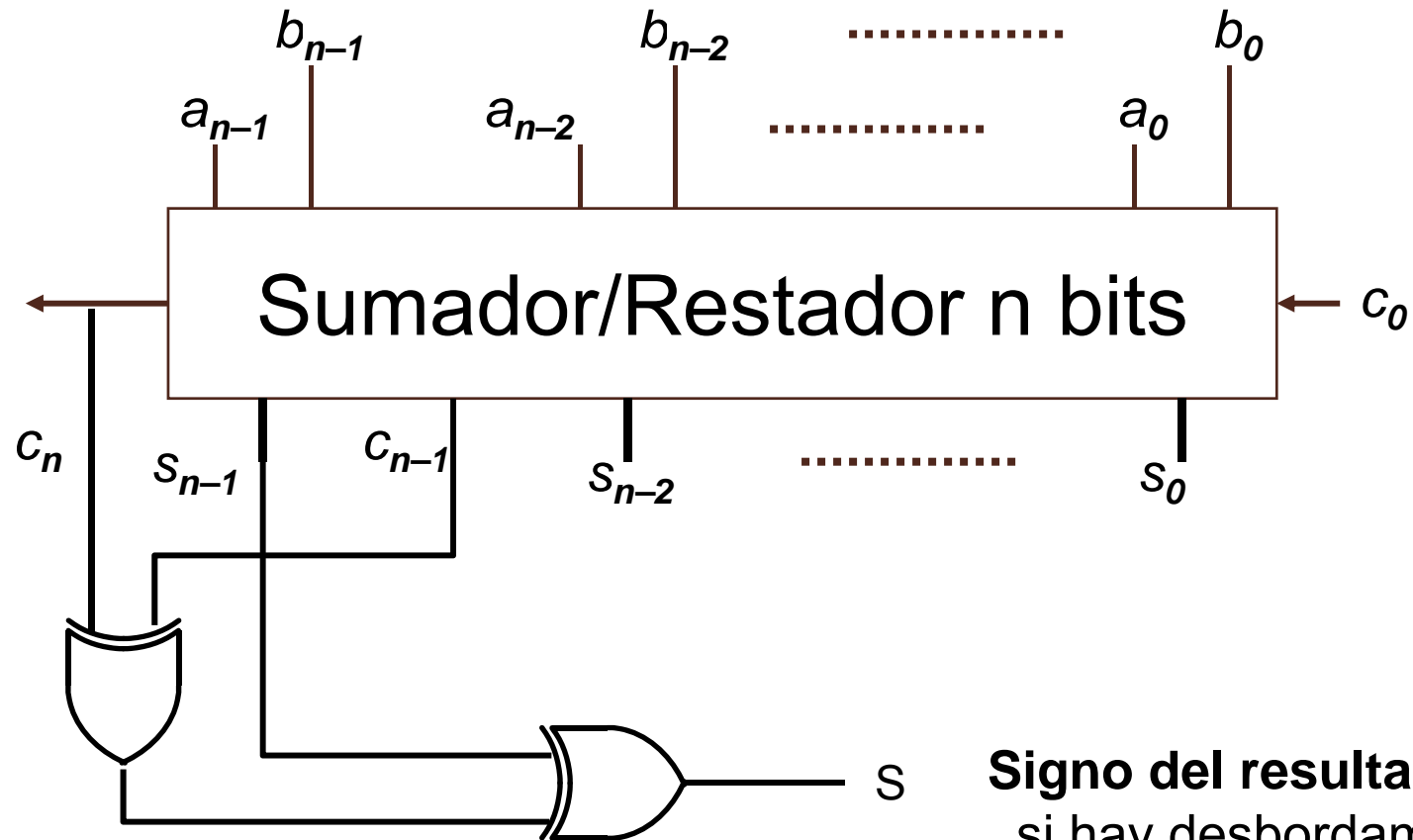
- **Modificación algoritmo 2 y operador secuencial para Booth**



- El algoritmo requiere n ciclos
- En cada ciclo hay que hacer una suma o una resta más un desplazamiento

Operadores de multiplicación con signo

- Detalle del cálculo del bit de signo adicional



Signo del resultado:
si hay desbordamiento,
invertir el MSB

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 1001 0
1		
2		
3		
4		

← Bit extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 1001 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 1001 0
2		
3		
4		

← Bit extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 <u>1001</u> 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 <u>1001</u> 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0 <u>100</u> 1
2		
3		
4		

← Bit extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 <u>1001</u> 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 1001 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0 <u>100</u> 1
2	Caso 01: $HI \leftarrow HI + M$	0 0001 0100 1
3		
4		

← Bit extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 <u>1001</u> 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 1001 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0 <u>100</u> 1
2	Caso 01: $HI \leftarrow HI + M$	0 0001 0100 1
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 10 <u>10</u> 0
3		
4		

← Bit extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 1001 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 1001 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0100 1
2	Caso 01: $HI \leftarrow HI + M$	0 0001 0100 1
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 1010 0
3	Caso 00: No hacer nada	0 0000 1010 0
4		

Bit
extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 <u>1001</u> 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 1001 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0 <u>100</u> 1
2	Caso 01: $HI \leftarrow HI + M$	0 0001 0100 1
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 10 <u>10</u> 0
3	Caso 00: No hacer nada	0 0000 1010 0
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 010 <u>1</u> 0
4		

← Bit extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 <u>1001</u> 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 1001 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0 <u>100</u> 1
2	Caso 01: $HI \leftarrow HI + M$	0 0001 0100 1
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 10 <u>10</u> 0
3	Caso 00: No hacer nada	0 0000 1010 0
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 010 <u>1</u> 0
4	Caso 10: $HI \leftarrow HI - M$	1 1110 0101 0

← Bit extra

Operadores de multiplicación con signo

- Ejemplo: algoritmo de Booth con el operador secuencial

✓ Con signo; $n = 4$; $M = 0010_2$; $Q = 1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 0000 1001 0
1	Caso 10: $HI \leftarrow HI - M$	1 1110 1001 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0100 1
2	Caso 01: $HI \leftarrow HI + M$	0 0001 0100 1
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 1010 0
3	Caso 00: No hacer nada	0 0000 1010 0
	Desplazar S-HI-LO 1 bit a la derecha	0 0000 0101 0
4	Caso 10: $HI \leftarrow HI - M$	1 1110 0101 0
	Desplazar S-HI-LO 1 bit a la derecha	1 1111 0010 1

Bit
extra

Operadores de multiplicación con signo

Ejercicio: $n=4$; $M=1101_2$; $Q=0110_2$ ($-3_{10} \times 6_{10} = -18_{10}$)

Cicle	Acció	S-HI-LO-X
0	Valores iniciales	0 0000 <u>0110</u> <u>0</u>
1		
2		
3		
4		

Solución: 1110 1110

Operadores de multiplicación con signo

- Recodificación por parejas de bits
 - ✓ Extensión de Booth que reduce a la mitad el número de dígitos de Q , reduciendo así el número de productos intermedios a sumar

			Booth		Parejas	Acción
q_{i+1}	q_i	q_{i-1}	q'_{i+1}	q'_i	q''_i	
0	0	0	0	0	0	Nada
0	0	1	0	1	1	Sumar M
0	1	0	1	-1	1	Sumar M
0	1	1	1	0	2	Sumar $2 \times M$
1	0	0	-1	0	-2	Restar $2 \times M$
1	0	1	-1	1	-1	Restar M
1	1	0	0	-1	-1	Restar M
1	1	1	0	0	0	Nada

En cada iteración se desplaza S-HI-LO dos posiciones a la derecha

Operadores de multiplicación con signo

- Ejemplo de multiplicación con recodificación por parejas

✓ $n = 5$; $M = 01101_2 (13_{10})$; $Q = 11010_2 (-6_{10})$

Extensión del signo \rightarrow 1 1 1 0 1 0 0 \leftarrow Bit implícito



Recodificación parejas \rightarrow

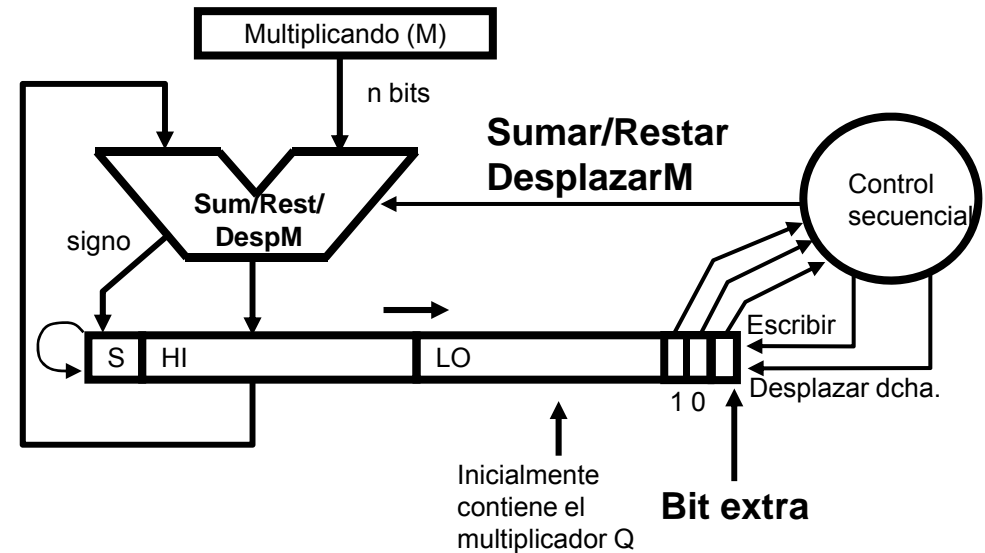
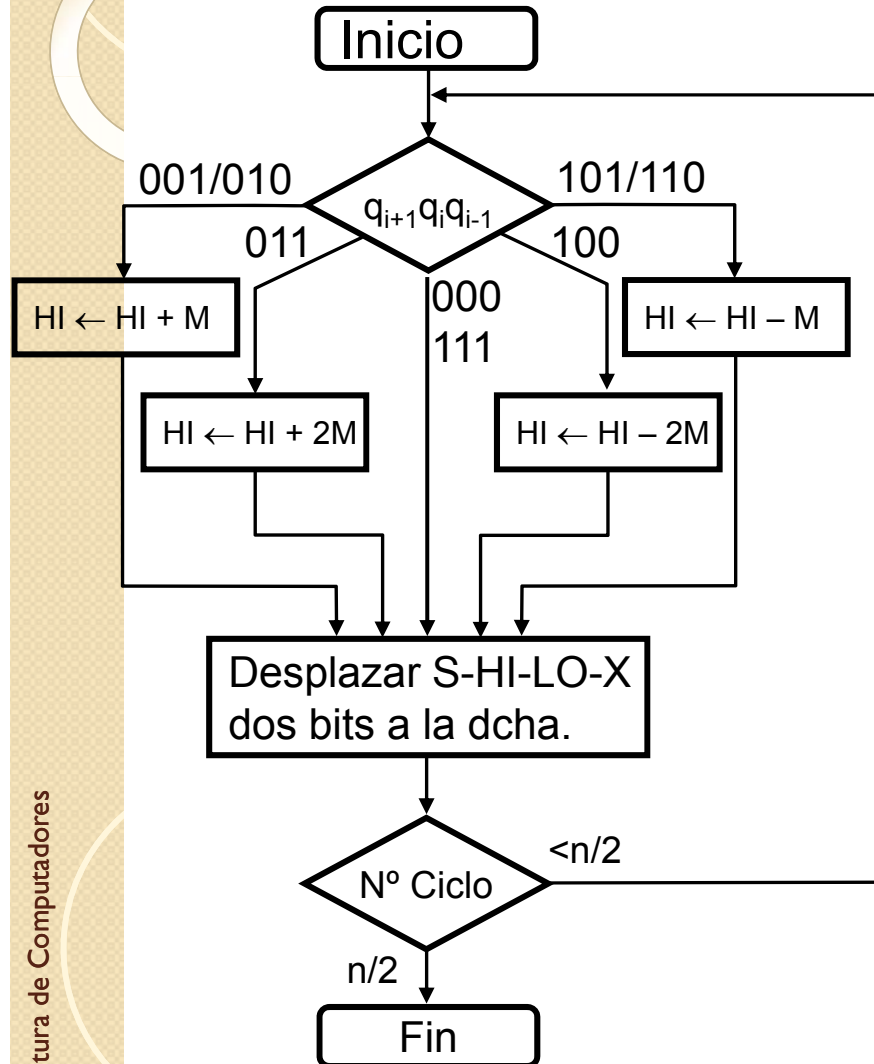
	0	0	-1	1	-1	0
	└──┘		└──┘		└──┘	
	↓		↓		↓	
	0		-1		-2	

						0	1	1	0	1
					×	0		-1		-2
						<hr/>				
Restar $2 \times M$	1	1	1	1	1	0	0	1	1	0
Restar M	1	1	1	1	0	0	1	1		
Nada	0	0	0	0	0	0				
	<hr/>									
	1	1	1	0	1	1	0	0	1	0

(-78_{10})

Operadores de multiplicación con signo

- Modificación algoritmo 2 y operador para parejas de bits



Los registros HI y LO deben tener un número par de bits

- El algoritmo requiere $n/2$ ciclos
- En cada ciclo hay que hacer hasta una suma o resta y dos desplazamientos

Operadores de multiplicación con signo

Ejemplo de multiplicación con recodificación por parejas

✓ $n=6$ (n ha de ser par); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 <u>111010</u> 0
1		
2		
3		

Operadores de multiplicación con signo

Ejemplo de multiplicación con recodificación por parejas

✓ $n=6$ (n ha de ser par); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 <u>111010</u> 0
1	Caso 100: $HI \leftarrow HI - 2M$	1 100110 111010 0
2		
3		

Operadores de multiplicación con signo

Ejemplo de multiplicación con recodificación por parejas

✓ $n=6$ (n ha de ser par); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 <u>111010</u> 0
1	Caso 100: $HI \leftarrow HI - 2M$	1 100110 <u>111010</u> 0
	Desplazar S-HI-LO 2 bits a la derecha	1 111001 <u>101110</u> 1
2		
3		

Operadores de multiplicación con signo

Ejemplo de multiplicación con recodificación por parejas

✓ $n=6$ (n ha de ser par); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 <u>111010</u> 0
1	Caso 100: $HI \leftarrow HI - 2M$	1 100110 <u>111010</u> 0
	Desplazar S-HI-LO 2 bits a la derecha	1 111001 <u>101110</u> 1
2	Caso 101: $HI \leftarrow HI - M$	1 101100 <u>101110</u> 1
3		

Operadores de multiplicación con signo

Ejemplo de multiplicación con recodificación por parejas

✓ $n=6$ (n ha de ser par); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 <u>111010</u> 0
1	Caso 100: $HI \leftarrow HI - 2M$	1 100110 <u>111010</u> 0
	Desplazar S-HI-LO 2 bits a la derecha	1 111001 <u>101110</u> 1
2	Caso 101: $HI \leftarrow HI - M$	1 101100 <u>101110</u> 1
	Desplazar S-HI-LO 2 bits a la derecha	1 111011 <u>001011</u> 1
3		

Operadores de multiplicación con signo

Ejemplo de multiplicación con recodificación por parejas

✓ $n=6$ (n ha de ser par); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 <u>111010</u> 0
1	Caso 100: $HI \leftarrow HI - 2M$	1 100110 <u>111010</u> 0
	Desplazar S-HI-LO 2 bits a la derecha	1 111001 <u>101110</u> 1
2	Caso 101: $HI \leftarrow HI - M$	1 101100 <u>101110</u> 1
	Desplazar S-HI-LO 2 bits a la derecha	1 111011 <u>001011</u> 1
3	Caso 111: No hacer nada	1 111011 <u>001011</u> 1

Operadores de multiplicación con signo

Ejemplo de multiplicación con recodificación por parejas

✓ $n=6$ (n ha de ser par); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 <u>111010</u> 0
1	Caso 100: $HI \leftarrow HI - 2M$	1 100110 <u>111010</u> 0
	Desplazar S-HI-LO 2 bits a la derecha	1 111001 <u>101110</u> 1
2	Caso 101: $HI \leftarrow HI - M$	1 101100 <u>101110</u> 1
	Desplazar S-HI-LO 2 bits a la derecha	1 111011 <u>001011</u> 1
3	Caso 111: No hacer nada	1 111011 <u>001011</u> 1
	Desplazar S-HI-LO 2 bits a la derecha	1 <u>111110</u> 110010 1

Operadores de multiplicación con signo

- Ejercicio: $n = 6$; $M = 101001_2 (-23_{10})$; $Q = 001001_2 (9_{10})$

$$(-23_{10} \times 9_{10} = -207_{10})$$

Ciclo	Acción	S-HI-LO-X
0	Valores iniciales	0 000000 001001 0
1		
2		
3		

Solución: 111100 110001

La multiplicación secuencial: resumen

- Implementación del operador:
 - ✓ La multiplicación puede implementarse mediante sumas y desplazamientos. En hardware sólo requiere:
 - registro de desplazamiento
 - sumador o sumador/restador, si es multiplicación con signo
 - circuito de control, si se utiliza un operador secuencial
- El método de Booth
 - ✓ Permite tratar de manera uniforme la multiplicación con o sin signo
- La recodificación del multiplicador
 - ✓ Permite reducir el número de dígitos del multiplicador y, por tanto, el número de iteraciones del operador
 - ✓ La recodificación por parejas de bits permite reducir a la mitad el número de ciclos requerido para una multiplicación secuencial
 - ✓ Otras recodificaciones permiten reducir aún más el número de iteraciones