

Estructura de Computadors

Tema 3

Aritmètica d'enters

DISCA



Aritmètica d'enters

■ Objectius:

- Fer la correspondència entre els tipus de dades numèrics d'alt nivell (p. ex, Java i C/C++) i els tipus nadius del processador
- Traduir a assembler expressions aritmètiques (càlculs i guardes) expresades en alt nivell
- Distingir entre operadors combinacionals i seqüencials i calcular-ne el temps d'operació i la productivitat en casos senzills a partir dels retards dels components
- Relacionar manipulació de bits amb operacions d'alt nivell (p. ex, operar amb els camps del format de coma flotant)
- Distingir les diferents respostes del computador davant de les operacions que no es poden completar (per desbordament o per indeterminació)
- Entendre el suport que dóna el joc d'instruccions a les singularitats del càlcul (excepcions, indicadors de la norma IEEE).

Índex

1. Introducció
 1. Tipus en alt i baix nivell
 2. Operacions i operadors
 3. Operacions lògiques
 4. La representació dels enters
2. Suma i resta d'enters
 1. Fonaments
 2. Suma i resta en el MIPS R2000
 3. Operadors de suma
 4. Operadors de resta
3. Multiplicació d'enters
 1. Fonaments
 2. Multiplicació i divisió en el MIPS
 3. Operadors de desplaçament
 4. Operadors de multiplicació sense signe
 5. Operadors de multiplicació amb signe

Bibliografia

D.L. Patterson, J. L. Hennessy: Estructura y diseño de computadores

- Ed Reverté, 2000: vol 1, cap 4
- Ed Reverté, 2011, traducció de la 4a edició en anglès: cap. 3

W. Stallings: Organización y Arquitectura de Computadores (7. ed)
Prentice Hall. Cap 8

David Goldberg: Computer Arithmetic

- Appendix H de J. L. Hennessy, D. L. Patterson: Computer Architecture, a Quantitative Approach, 3ª ed
- (PDF)
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.3375&rep=rep1&type=pdf>
- (PDF) <http://www.cs.clemson.edu/~mark/464/appH.pdf>

Aritmètica d'enters

Introducció

1. Tipus de dades en alt i baix nivell
2. Operacions i operadors
 - Els càlculs en els computadors
 - La unitat aritmètica i lògica
 - Paràmetres d'una ALU
 - Prestacions
3. Exemples: operacions lògiques
4. La representació dels enters
 - Notes sobre la representació dels enters
 - Conversió entre tipus
 - El salt condicional en el MIPS

1.1 Tipus en alt i en baix nivell

- Tipus de dades bàsiques

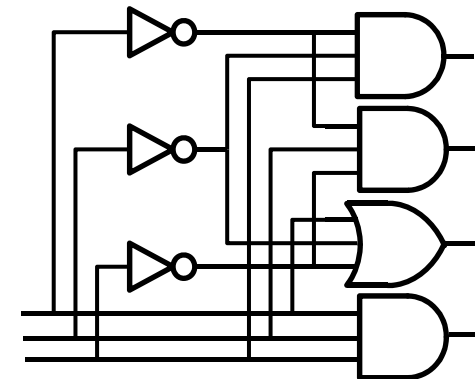
	bits	Java	C/C++ (32 bits)	MIPS	x64/IA-64
caràcter	8	—	char	byte	byte
	16	char	wchar_t	halfword	word
enter sense signe	8	—	unsigned byte	byte	byte
	16	—	unsigned short	halfword	word
	32	—	unsigned long	word	dword
enter amb signe	8	byte	byte	byte	byte
	16	short	short	halfword	word
	32	int	long	word	dword
	64	long	—	—	qword
coma flotant	32	float	float	float	float
	64	double	double	double	double

1.2 Operacions i operadors

- Els càlculs en els computadors
 - Les operacions lògiques i aritmètiques expressades en alt nivell es tradueixen en dades i instruccions de codi màquina
 - Durant el cicle d'instrucció, el processador aplica operadors per a processar les dades

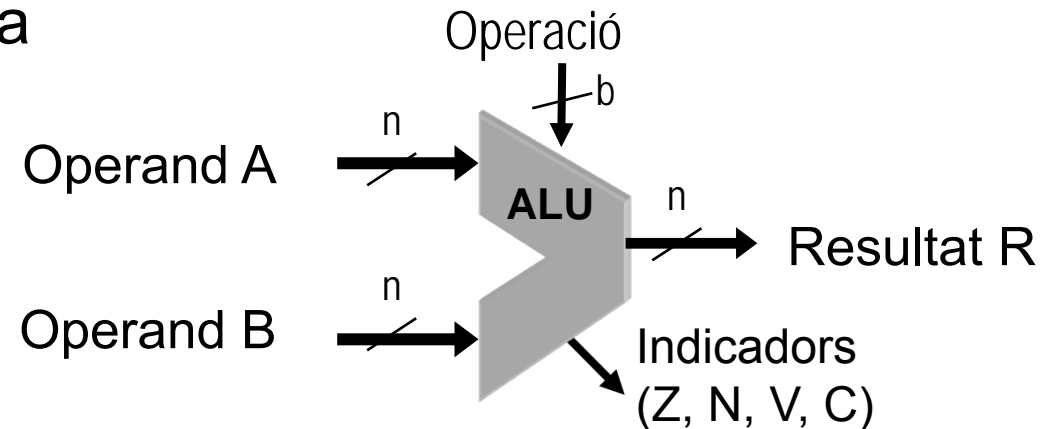
```
int[] j;  
short a;  
float x,y;  
  
x = 5*j[a];  
if (a>x){  
    y=x*1.3e5;  
    j=(int)exp(x);  
}
```

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



1.2 Operacions i operadors

- La unitat aritmético-lògica



- L'ALU (*Arithmetic Logic Unit*) és un element funcional del processador format per un conjunt d'operadors digitals que fan les operacions codificades en les instruccions
 - Cada operador implementa una o més operacions que s'apliquen als operands i produeix un resultat
- El control del processador, dirigit per les instruccions que es descodifiquen, selecciona els operadors i encamina operands i resultats des de i als registres implicats
- L'ALU pot activar indicadors (*flags*) que donen informació del resultat (Z=1 si R=0, N=1 si R<0, V=1 si desbordament en Ca2, etc.)

1.2 Operacions i operadors

- Paràmetres d'una ALU
 - Funcionals:
 - Operacions que pot realitzar
 - Conversió entre tipus
 - Operacions de bit (&, |) i desplaçaments
 - Càlcul elemental: suma, resta, multiplicació i divisió
 - Comparació (=, ≠, <, ≤, ≥, >)
 - Tipus d'operands que pot manejar
 - Prestacions (cost temporal):
 - A quina velocitat opera una ALU?
 - Quantes operacions pot fer per unitat de temps?
 - Complexitat (cost espacial)
 - Quants recursos físics cal dedicar a l'operador?
 - Quin espai del xip cal dedicar a l'operador?

1.2 Operacions i operadors

- Com expressarem les prestacions?
 - Temps de resposta
 - Temps que cal per a realitzar un càlcul. Com més curt, millor
 - Es mesura en unitats de temps (ns, temps de porta...)
 - Productivitat
 - Nombre d'operacions per unitat de temps. Com més gran, millor
 - Es pot mesurar amb unitats genèriques OPS (operacions per segon), KOPS, MOPS, GOPS...
 - cas de coma flotant: FLOPS, MFLOPS, etc.
- Complexitat (cost espacial)
 - Nombre de portes
 - Nombre de transistors
 - Superfície de xip
 - Unitats típiques: μm^2 , nm^2

1.3 Exemple: operacions lògiques

- 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 ^ ~3 = 8

1.3 Exemple: operacions lògiques

- En el MIPS

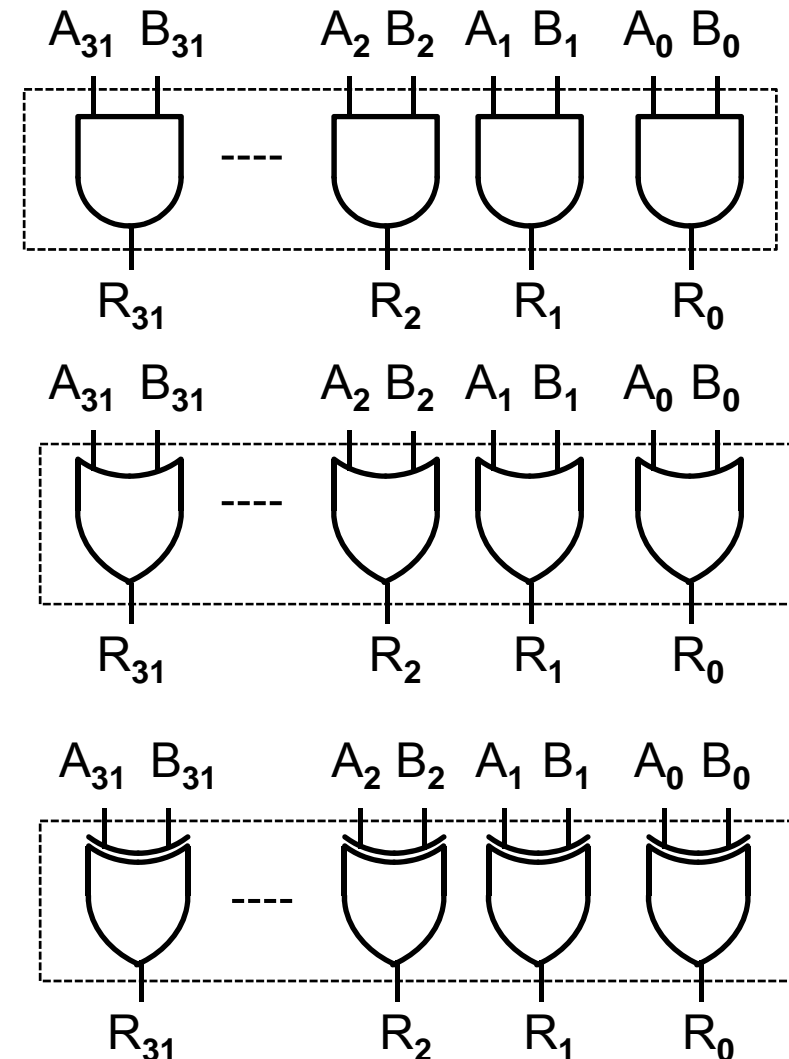
format R	format I
OR	ORI
AND	ANDI
XOR	XORI
NOR	

- Operació derivada: NOT (Ca1)
`nor $t0,$t0,$zero`

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

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

- Productivitat dels operadors: $1/t_{\text{Porta}}$
 - Si $t_{\text{Porta}} = 50 \text{ ps}$, productivitat
 $P = 20 \text{ GOPS}$



1.4 Representació dels enters

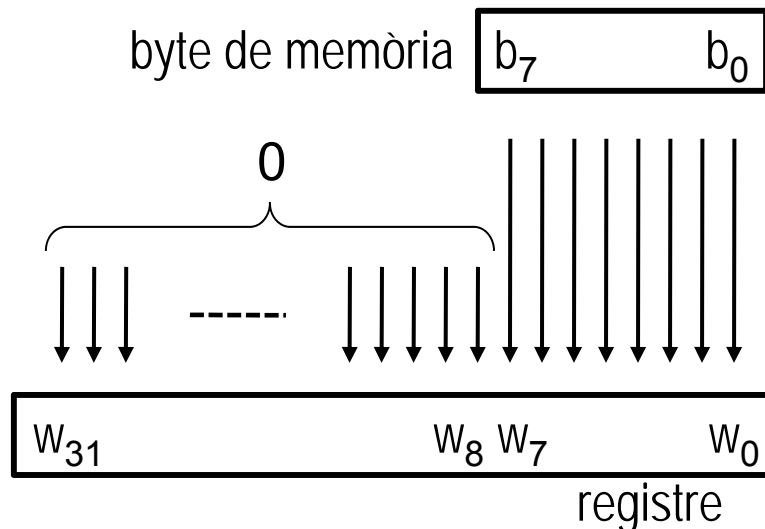
- Notes sobre la representació d'enters
 - Els conjunts matemàtics \mathbb{N} i \mathbb{Z} són infinits, però els tipus de dades bàsics d'un computador tenen una capacitat de representació finita
 - Amb n bits només es poden representar 2^n paraules diferents
 - La representació d'enters en el computador és exacta però limitada
 - Rangs de representació amb n bits
 - Per a \mathbb{N} : codificació binària natural: $[0 \dots +2^n-1]$
 - Per a \mathbb{Z} : codificació en complement a 2: $[-2^{n-1} \dots +2^{n-1}-1]$

n	sense signe	amb signe
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)

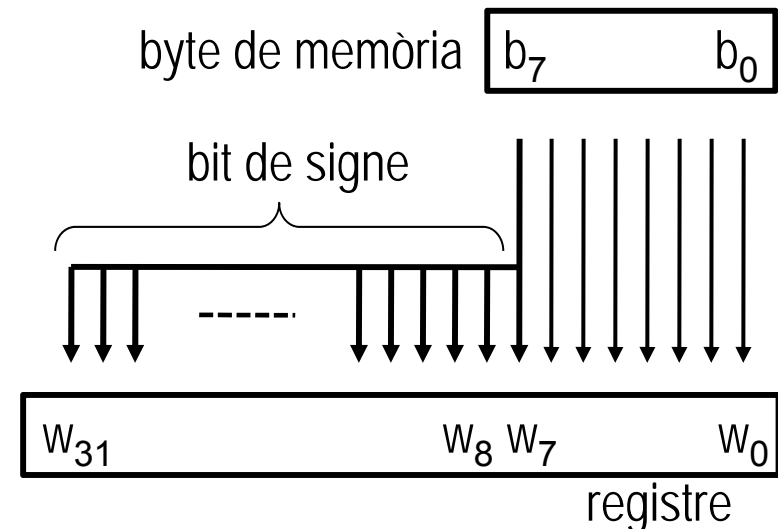
1.4 Representació dels enters

- Conversió entre tipus enters en el MIPS R2000
 - El tipus enter *natiu* és de 32 bits (ALU i registres)
 - Les instruccions d'accés a la memòria fan conversió si cal
 - LBU i LHU afegeixen zeros per l'esquerra (vàlid per a CBN)
 - LB i LH fan extensió de signe (vàlid per a Ca2)
 - SB i SH eliminen bits per l'esquerra

LBU



LB



1.4 Representació dels enters

- El salt condicional en el MIPS
 - Per comparació entre dos operands:
(BEQ/BNE r_A, r_B , eti)

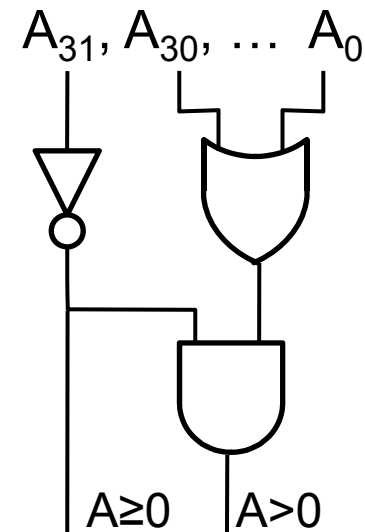
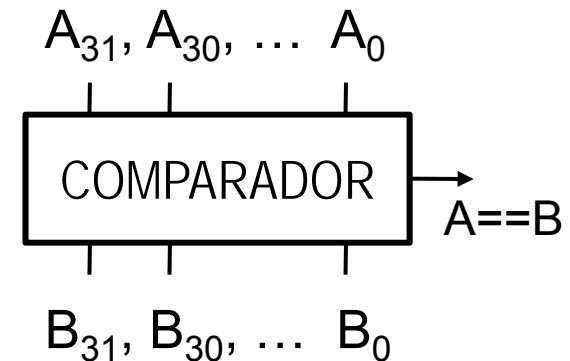
Instrucció	Condició
BEQ	$A = B$
BNE	$A \neq B$

- Les pseudoinstruccions
BEQZ/BNEZ r, eti
deriven de BEQ i BNE

- Basades en l'anàlisi d'un operand:

Instrucció	Condició
BGEZ	$A_{31} = 0$
BGTZ	$A_{31} = 0$ i $A_{30} \dots A_0 \neq 0$
BLTZ	$A_{31} = 1$
BLEZ	$A_{31} = 1$ o $A_{31} \dots A_0 = 0$

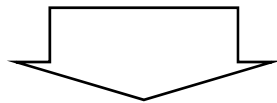
Alguns operadors de càlcul de condició de salt



1.4 Representació dels enters

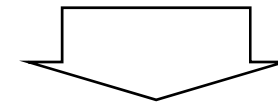
- El salt condicional en el MIPS
 - Exemples de condició

```
int i;  
...  
do {  
    ...  
} while (i>=0);
```



```
eti_do: ...  
    ...  
    lw $t0,i  
    bgez $t0,eti_do
```

```
int i,j;  
...  
if ((i==j) && (j>0))  
    ...  
else  
    ...
```



```
lw $t0,i  
lw $t1,j  
bne $t0,$t1,eti_else  
blez $t1,eti_else  
...  
eti_else:  
    ...
```


Índex

2. Suma i resta d'enters

2.1 Fonaments

- Anatomia de la suma
- La resta
- La comparació

2.2 Suma i resta en el R2000

- Instruccions de suma i resta
- Tractament del desbordament
- Instruccions de comparació

2.3 Operadors de suma

- El sumador d'un bit
- El sumador sèrie o CPA
- Detecció del desbordament

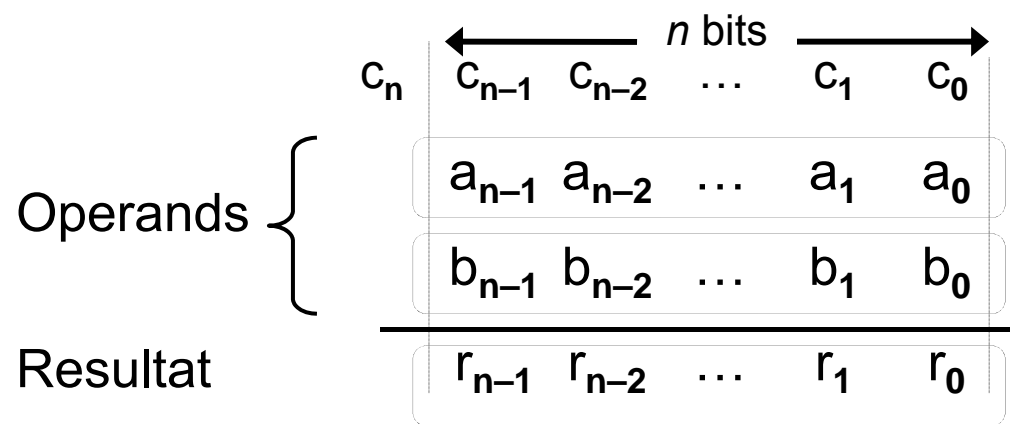
– 2.4 Operadors de resta

- Disseny a partir del sumador

2.1 Fonaments

■ Anatomia de la suma

- El procediment general calcula $R = A + B + c_0$



Transports:

- c_0 (de entrada), habitualment $c_0 = 0$
- $c_{n-1} \dots c_1$, formen part del càlcul
- c_n (d'eixida) útil de vegades

- El procediment de suma és el mateix per a CBN i Ca2

	← 4 bits →	CBN	Ca2
	0 1 0 1	5	+5
+	1 0 0 1	+ 9	+ -7
<hr/>			
	1 1 1 0	14	-2

2.1 Fonaments

▪ El desbordament en la suma

– Detecció:

- En aritmètica en CBN, un transport final $c_n=1$
- En aritmètica en Ca2, només pot donar-se si els signes dels sumands són iguals. Dues maneres de detectar-ho:

– el signe del resultat generat està invertit: $a_{n-1} = b_{n-1} \neq r_{n-1}$ o $(a_{n-1} \oplus b_{n-1}) \cdot (b_{n-1} \oplus r_{n-1}) = 1$

– els dos transports finals són diferents: $c_n \neq c_{n-1}$ o $c_n \oplus c_{n-1} = 1$

– Exemples amb $n = 4$ bits

CBN:

$$\begin{array}{r}
 c_n \rightarrow \textcircled{1} \quad 1 \quad 1 \quad 0 \\
 \quad \quad 0 \quad 1 \quad 1 \quad 0 \quad 6 \\
 + \quad 1 \quad 1 \quad 1 \quad 0 \quad 14 \\
 \hline
 \quad \quad 0 \quad 1 \quad 0 \quad 0 \quad \cancel{4}
 \end{array}$$

Ca2:

$$\begin{array}{r}
 \rightarrow \textcircled{0} \quad 1 \quad 0 \quad 0 \\
 \quad \quad 0 \quad 1 \quad 1 \quad 0 \quad +6 \\
 + \quad 0 \quad 1 \quad 0 \quad 1 \quad +5 \\
 \hline
 \rightarrow \textcircled{1} \quad 0 \quad 1 \quad 1 \quad \cancel{-5}
 \end{array}$$

2.1 Fonaments

▪ La resta

- La resta $R=A-B$ (en CBN i en Ca2) pot fer-se com la suma
 $R = A + Ca2(B) = A + Ca1(B) + 1$

0	0	0	1		CBN	Ca2		1	1	1	0	1
	1	1	1	0	14	-2			1	1	1	0
-	0	1	0	1	-5	- +5		+	1	0	1	0
<hr/>								<hr/>				
	1	0	0	1	9	-7			1	0	0	1

- Els préstecs de la resta apareixen invertits en els transports de la suma
- En CBN: el desbordament es detecta quan $c_n = 0$
- En Ca2: el desbordament es detecta quan $c_n \neq c_{n-1}$

2.1 Fonaments

- La comparació de dos enters:
 - El resultat d'una comparació és un bit (1 = cert, 0 = fals)
 - La comparació $A < B$ és pot fer restant ($R = A - B$) i analitzant els transports i el signe de R
 - el valor concret de R és irrellevant

Comparació	CBN	Ca2
$A == B$	$R == 0$	$R = 0$
$A \geq B$	$c_n = 1$ (R és representable)	R és positiu
$A < B$	$c_n = 0$ (R no és representable)	R és negatiu

2.2 Suma i resta amb el MIPS R2000

▪ Instruccions

- Operands de 32 bits
- Versions registre-registre (format R) o registre-immediat (format I)
- Instruccions additives:

operació	format	amb signe	sense signe
suma	R	ADD/ADDU	
suma	I	ADDI/ADDIU	
resta	R	SUB/SUBU	
comparació	R	SLT	SLTU
comparació	I	SLTI	SLTIU

- ADD i ADDU (etc...) fan la mateixa operació, però ADD pot produir excepcions
- Totes les instruccions de format I fan extensió de signe de la constant
- No hi ha resta en format I
 - Per a restar una constant k caldrà sumar $-k$

2.2 Suma i resta amb el MIPS R2000

- Instruccions de suma i resta

- `ADD rdst,rfnt1,rfnt2`

- `ADDI rdst,rfnt,imm`

- Si hi ha desbordament (amb signe), provoquen una excepció i el registre `rdst` no es modifica

- `ADDU rdst,rfnt1,rfnt2`

- `ADDIU rdst,rfnt,imm`

- Mai no provoquen cap excepció, ni detecten situacions de desbordament

- `SUB rdst,rfnt1,rfnt2`

- Si hi ha desbordament (amb signe), provoquen una excepció i el registre `rdst` no es modifica

- `SUBU rdst,rfnt1,rfnt2`

- Mai no provoquen cap excepció, ni detecten situacions de desbordament

2.2 Suma i resta amb el MIPS R2000

- Tractament del desbordament en alt nivell
 - En Java i en C l'aritmètica entera ignora el desbordament

```
int a,b,c;                                     Java
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

- En generar codi per a MIPS, el compilador triarà ADDU i no ADD
- Per a detectar el desbordament, caldrà afegir al programa:

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


2.2 Suma i resta amb el MIPS R2000

- Tractament del desbordament en baix nivell
 - Detecció de la desigualtat dels signes entre operands

```
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

Detecció del desbordament:
1. Signe a vs. signe b
2. Signe a (o b) vs. signe c

Salt per a tractar el
desbordament

Esriptura del resultat
si no hi ha hagut desbordament

2.2 Suma i resta amb el MIPS R2000

- Instruccions de comparació (*Set on LessThan*)

- **SLT** *rdst, rfnt1, rfnt2*

- SLTI** *rdst, rfnt, imm*

- Fan la comparació menor estricta ($rfnt1 < rfnt2$ o $rfnt < imm$) entre dos operands, tot interpretant-los en Ca2
 - Si la condició s'acompleix fa $rdst=1$, en cas contrari $rdst=0$
 - Mai no generen cap excepció

- **SLTU** *rdst, rfnt1, rfnt2*

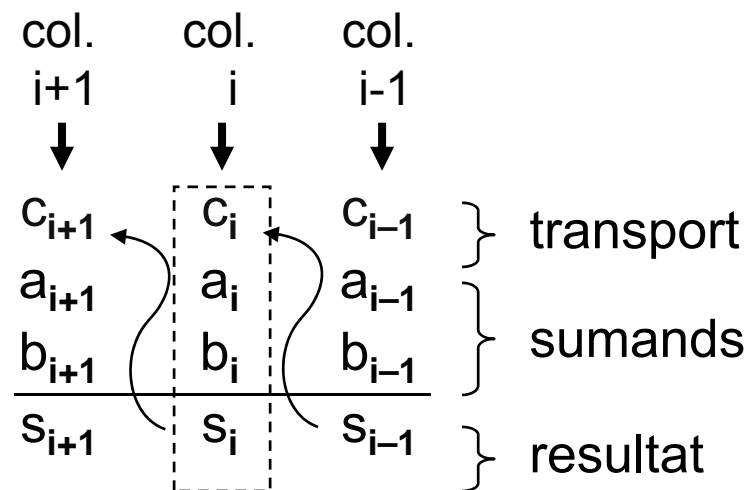
- SLTIU** *rdst, rfnt, imm*

- Fan la comparació ($rfnt1 < rfnt2$ o $rfnt < imm$) entre dos operands, tot interpretant-los en CBN
 - Si la condició s'acompleix fa $rdst=1$, en cas contrari $rdst=0$
 - Mai no generen cap excepció

2.3 Operadors de suma

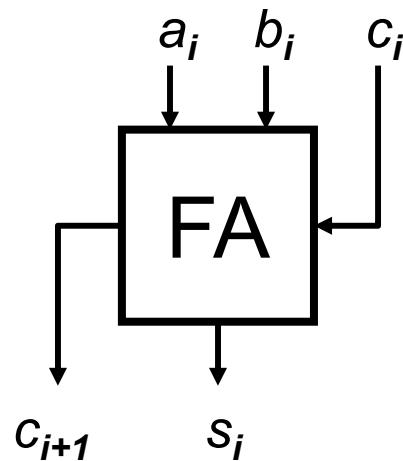
▪ La suma en sèrie

- La suma en sèrie reproduueix el procediment humà de suma
- La suma es fa per ordre, des del LSB cap al MSB
- En cada columna i , cal sumar els bits dels sumands a_i , b_i i el transport c_i provinent de la columna $i-1$ per a obtenir el bit del resultat s_i i generar el transport c_{i+1} cap a la columna $i+1$ següent
- Transport d'entrada $c_0 = 0$



2.3 Operadors de suma

- El sumador complet d'un bit (*Full adder*):
 - Implementa els càlculs d'una columna de la suma en sèrie
 - És un operador que admet 3 entrades d'un bit i produeix dues eixides:

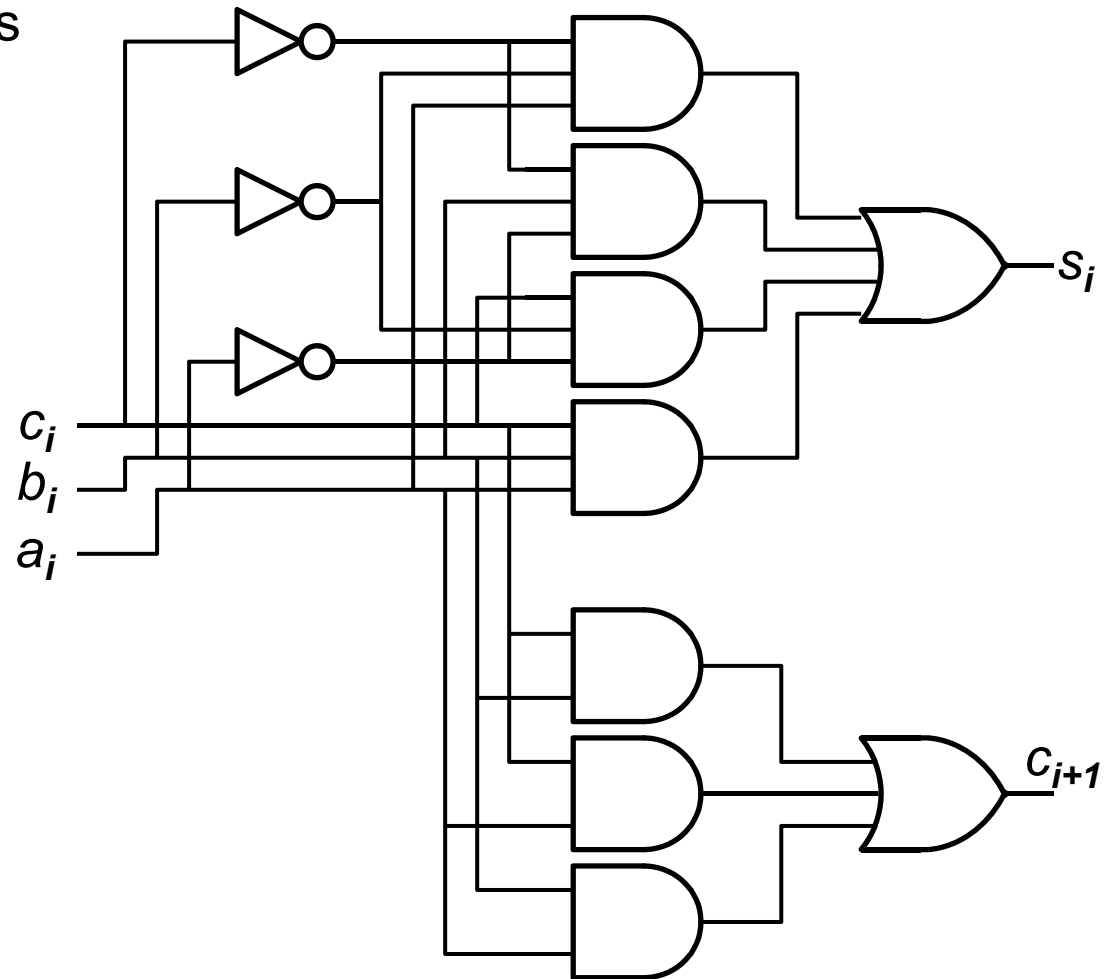


a_i	b_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

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

2.3 Operadors de suma

- El sumador complet d'un bit: Implementació
 - Implementació a partir de les funcions lògiques



2.3 Operadors de suma

■ El sumador complet d'un bit: Prestacions

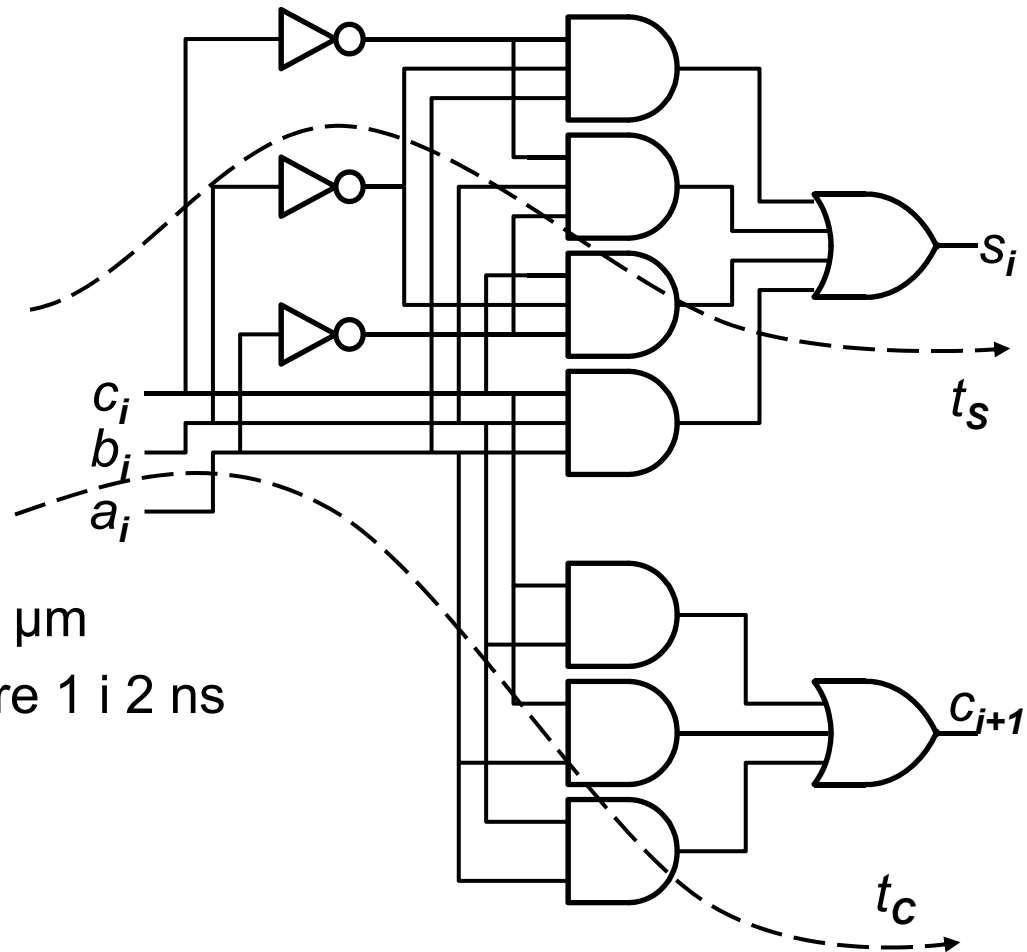
- Temps de retard:

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

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

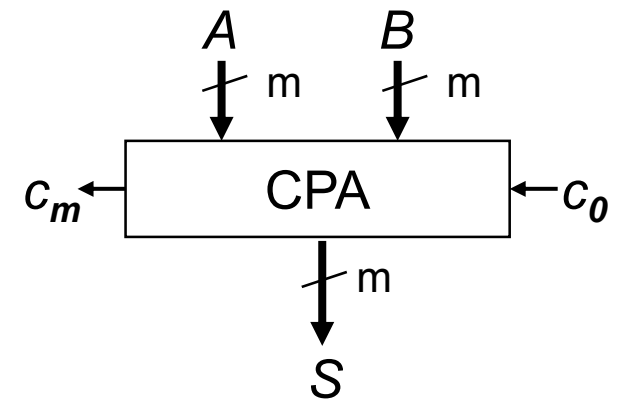
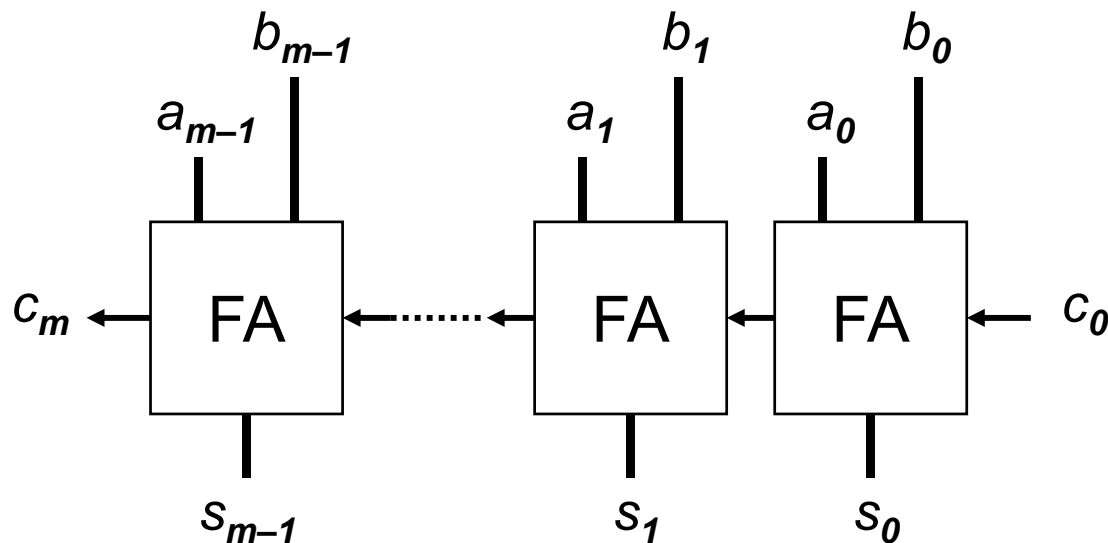
- Complexitat: 12 portes

- Amb tecnologia CMOS 0.5 μm
 - temps de resposta: entre 1 i 2 ns
 - superfície: 1000 μm^2



2.3 Operadors de suma

- El sumador sèrie de n bits
 - CPA (*Carry Propagation Adder*)
 - Per a fer un sumador de dos números de m bits, es connecten m sumadors complets en cascada
 - l'eixida de transport del sumador i -èssim es connecta a l'entrada de transport del sumador $i+1$ -èssim
 - l'operador resultant té una entrada i una eixida de transport globals

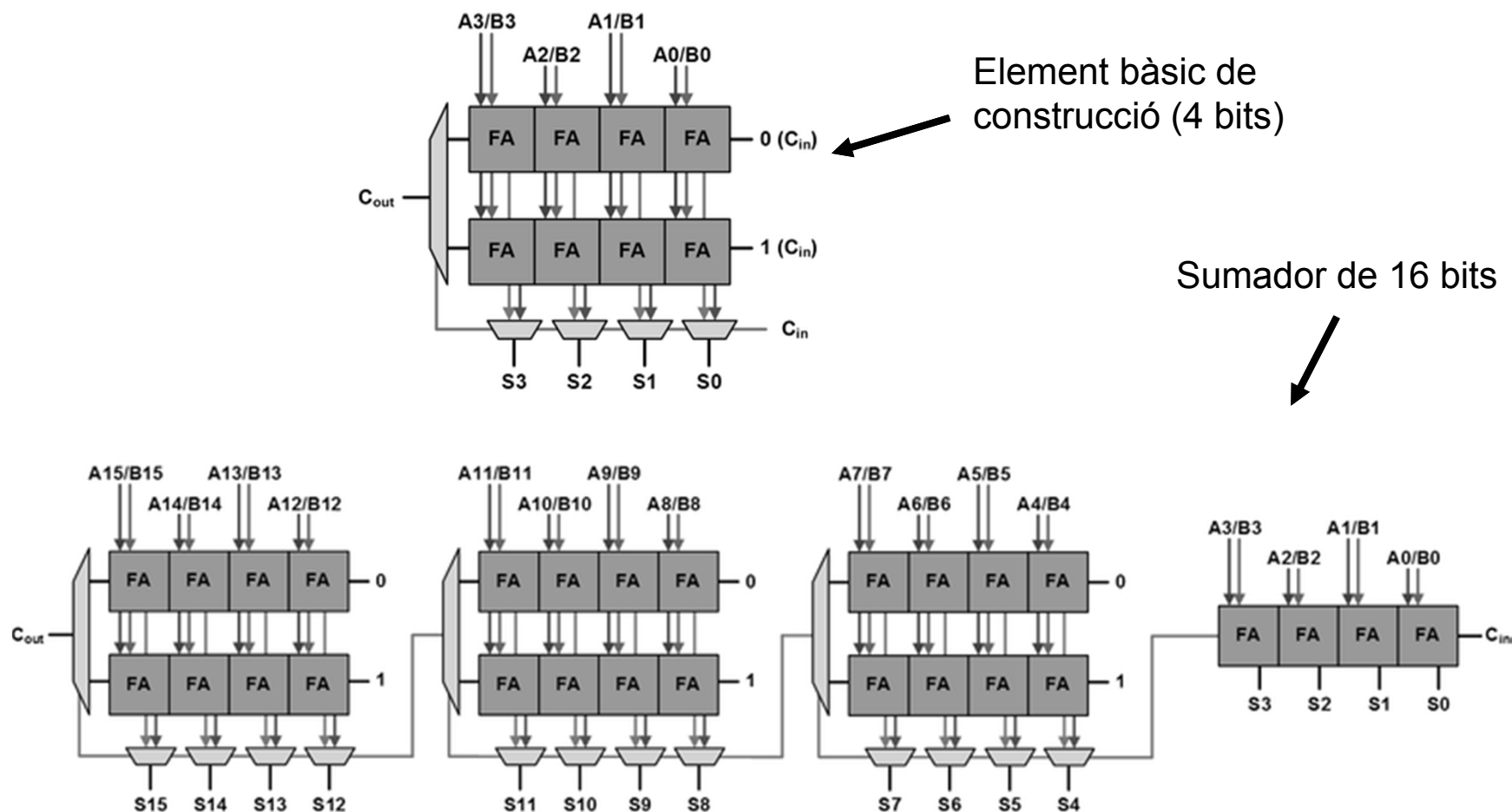


2.3 Operadors de suma

- El sumador sèrie de n bits. Complexitat
 - El temps de càlcul d'un sumador sèrie per a operands de n bits es pot expressar en termes dels retards d'un sumador complet:
$$t(n \text{ bits}) = (2n+1) \cdot T, \text{ sent } T = \text{retard d'una porta}$$
 - Asimptòticament el temps de càlcul és lineal, $t(n \text{ bits}) = O(n)$
 - El cost espacial també és lineal, $\text{cost}(n \text{ bits}) = O(n)$
 - La suma sèrie és poc eficient per a les aplicacions pròpies d'un processador convencional, amb $n=32$ o $n=64$ bits
- Millores en la suma
 - Anticipació del transport: CLA, *Carry Lookahead Adder*
 - Calcula els bits de transport c_i abans que els bits de suma s_i
 - El retard del càlcul dels transports és $O(\log(n))$
 - Selecció del transport: CSA, *Carry Select Adder*
 - Divideix la suma en dues parts i gestiona la part alta amb els possibles valors del transport provinent de la part baixa

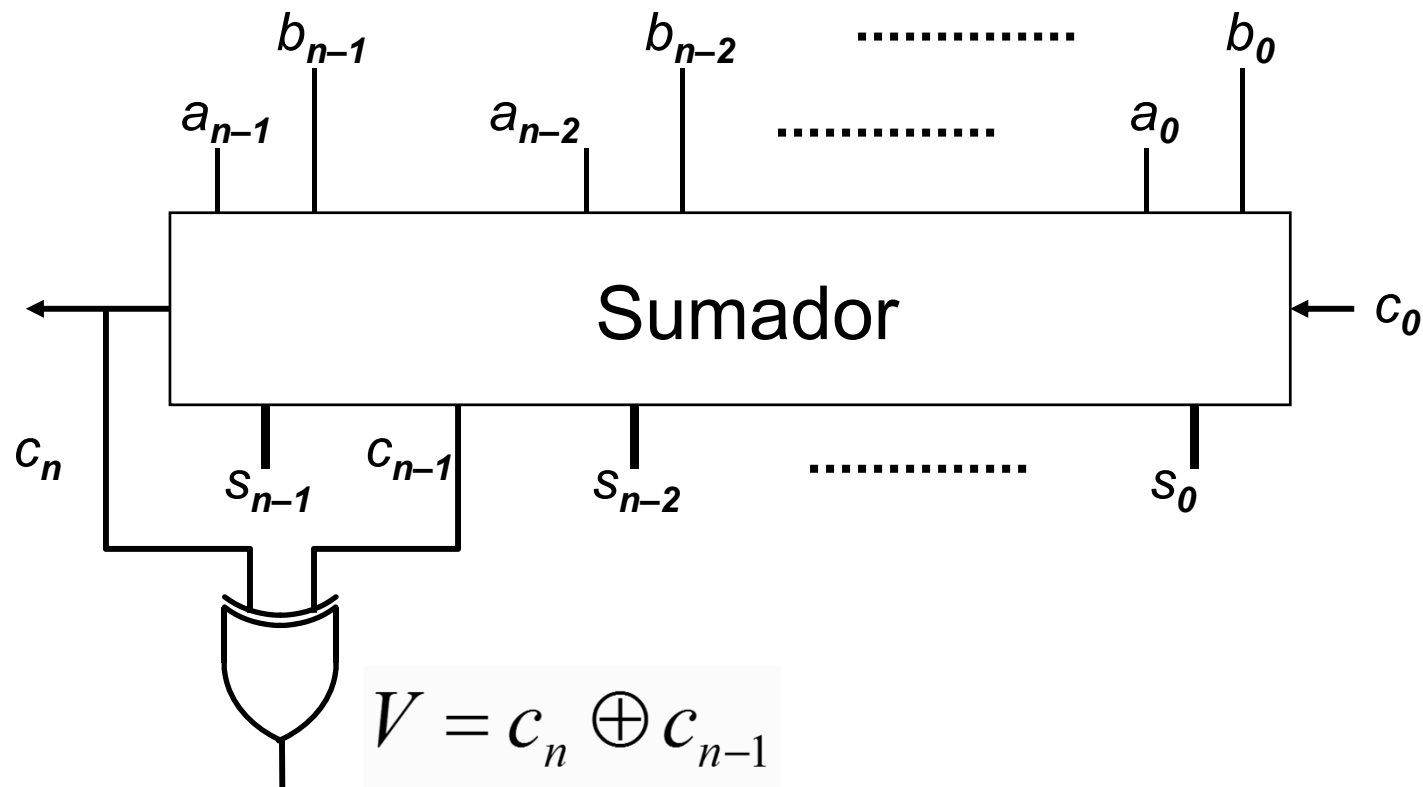
2.3 Operadors de suma

- CSA (Carry Select Adder)
 - Accelera l'operació de suma a base d'invertir-hi més circuits



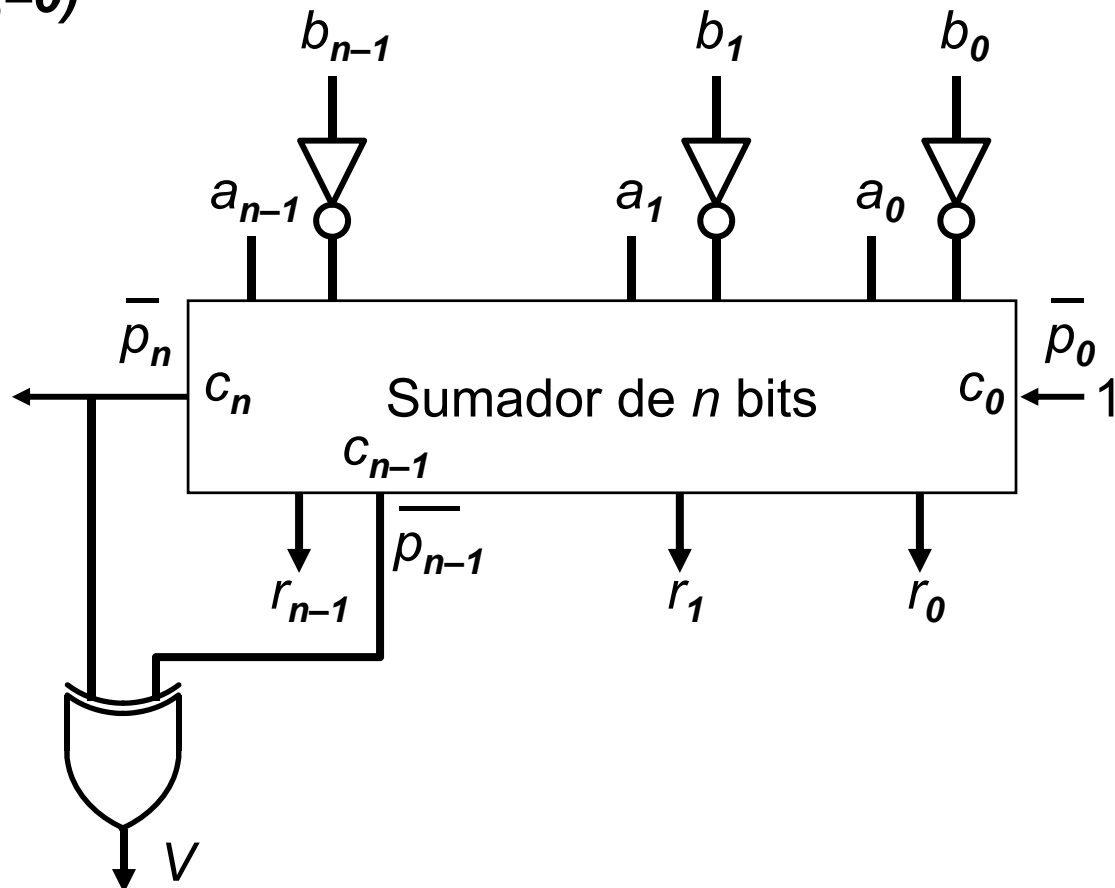
2.3 Operadors de suma

- Càlcul del desbordament en l'aritmètica en Ca2
 - Es detecta quan els bits de transport d'ordre n i $n-1$ no són iguals
 - El signe del resultat s_{n-1} és incorrecte



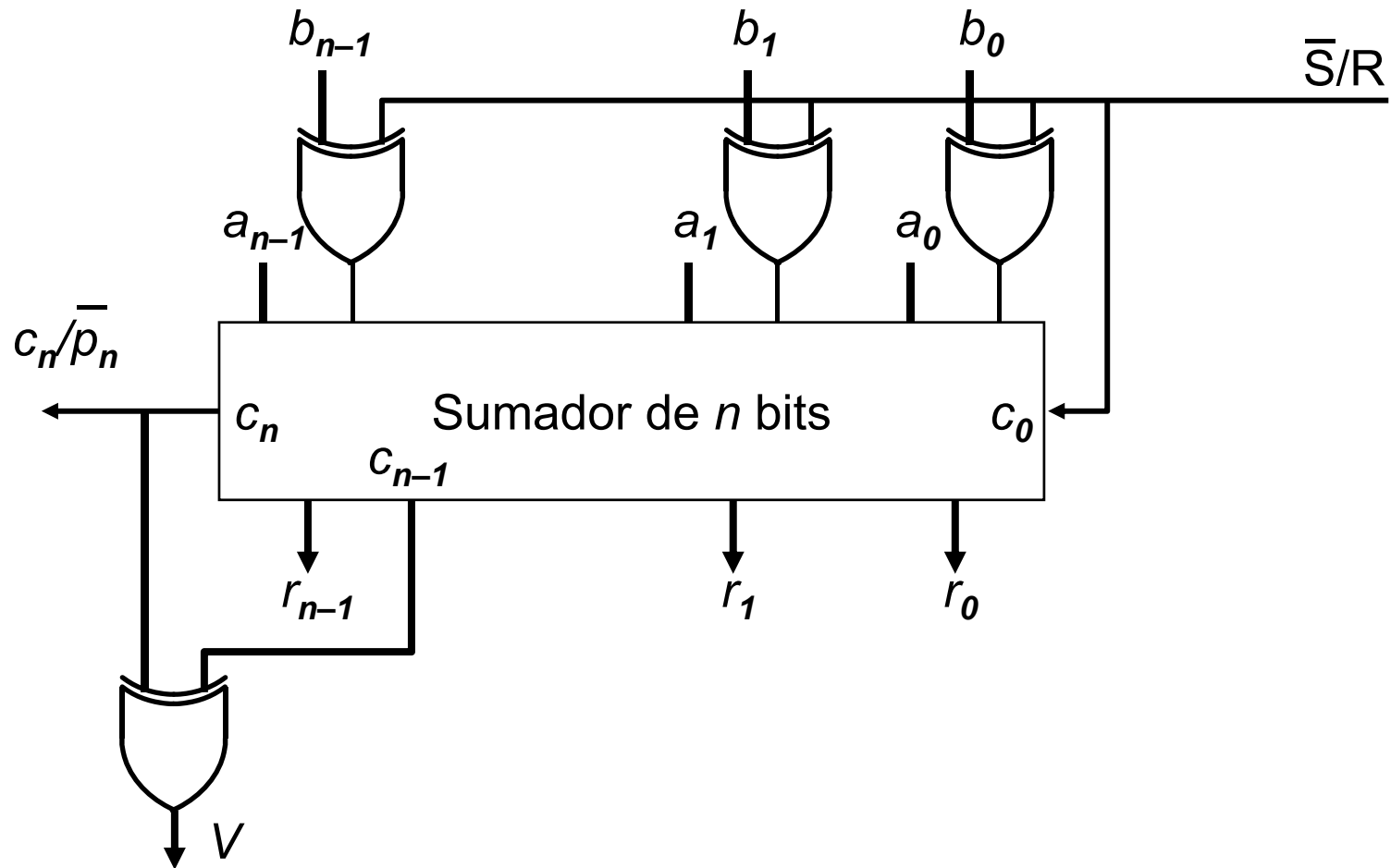
2.4 Operadors de resta

- Disseny del restador a partir del sumador
 - $R = A - B = A + \text{Ca2}(B) = A + \text{not}(B) + 1$
 - La detecció de desbordament és igual que amb la suma:
 - En CBN: $p_n=1$ ($c_n=0$)
 - En Ca2: $c_n \neq c_{n-1}$ ($c_n \text{ xor } c_{n-1} = 1$)



2.4 Operadors de resta

- Disseny clàssic del sumador/restador



Índex

3. Multiplicació i divisió d'enters

3.1 Fonaments

Desplaçaments i aritmètica

Anatomia de la multiplicació sense signe

Problemàtica de la multiplicació amb signe

La divisió

3.2 Multiplicació i divisió en el MIPS

Instruccions de desplaçament

Instruccions de multiplicació i divisió generals

3.3 Operadors de desplaçament

3.4 Operadors de multiplicació sense signe

Operadors seqüencials

Operador de multiplicació sense signe

3.5 Operadors de multiplicació amb signe

Algorismes 1 i 2

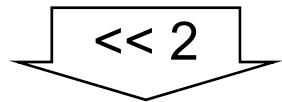
Algorisme de Booth

Algorisme de Booth amb recodificació per parelles

3.1 Fonaments

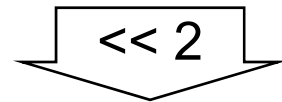
- Desplaçaments i aritmètica entera (I)
 - Desplaçar n bits cap a l'esquerra és equivalent a multiplicar per 2^n
 - Entren n zeros per la dreta
 - Operació vàlida per a enters amb i sense signe
 - Nom de l'operació: desplaçament lògic cap a l'esquerra

0 0 0 1 1 0 6

 $\times 2^2$

0 1 1 0 0 0 24

1 1 1 0 1 0 -6

 $\times 2^2$

1 0 1 0 0 0 -24

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

Java

-12 * 8 = -96

3.1 Fonaments

- Desplaçaments i aritmètica entera (II)
 - Desplaçar n bits cap a la dreta és equivalent a dividir per 2^n
 - Amb enters sense signe: entren n zeros per l'esquerra
 - Amb enters amb signe: el bit de signe es replica n vegades

Sense signe:
desplaçament **lògic**

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

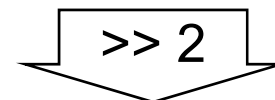
Amb signe:
desplaçament **aritmètic**

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

3.1 Fonaments

- Els compiladors eviten les operacions de multiplicació sempre que és possible

```
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
...

3.1 Fonaments

- Anatomia de la multiplicació sense signe
 - En general, per representar el producte de dos nombres de n bits cal $2n$ bits
 - El procediment *humà* demana sumes i desplaçaments

				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								

Notació

M = Multiplicand; m_i = bit i-èssim

Q = Multiplicador; q_i = bit i-èssim

P = Producte; p_i = bit i-èssim

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

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

3.1 Fonaments

- Problemàtica de la multiplicació amb signe
 - Cal estendre el signe del multiplicand
 - El pes del bit de signe del multiplicador és de -2^{n-1} (i no 2^{n-1})
 - Convé buscar codificacions alternatives

1	1	1	1	1	1	0	1	(-3_{10})
			×	1	0	0	1	(-7_{10})
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	
0	0	0	1	0	1	0	1	
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
Pesos								

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

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

3.1 Fonaments

- La divisió

- Donats un dividend i un divisor, produeix dos resultats: el quocient i el residu (o mòdul)
- Exemple sense signe: algoritme humà de restes i desplaçaments
 - Si no cap (X): 0 al quocient
 - Si cap (✓): restar i 1 al quocient

$$\begin{array}{r}
 \begin{array}{cccc|cccc}
 & & & & 1 & 1 & 0 & 1 \\
 X- & 0 & 1 & 0 & 1 & & & \\
 & & & & & & & \\
 X- & 0 & 1 & 0 & 1 & & & \\
 & & & & & & & \\
 \checkmark- & 0 & 1 & 0 & 1 & & & \\
 & & & & \hline
 & & & & 0 & 0 & 1 & 1 \\
 X- & & & & 0 & 1 & 0 & 1 \\
 & & & & 0 & 0 & 1 & 1
 \end{array}
 \end{array}
 \begin{array}{l}
 \hline
 0 \ 1 \ 0 \ 1 \\
 0 \ 0 \ 1 \ 0 \\
 \\
 \\
 \\
 \\
 \\
 \\
 \text{Residu final}
 \end{array}$$

- Amb signe: per convenció, el residu té el mateix signe que el dividend

3.1 Fonaments

- La multiplicació i la divisió en alt nivell
 - El mòdul

```
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

- La divisió per zero

```
int x,y,z; Java,C
x = 0;
y = 1;
z = y/x;
```

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

3.2 Multiplicació i divisió en el MIPS

- Instruccions de desplaçament
 - Són de la forma *operació Rr, Ri, Desp*, on *Desp* pot ser una constant o un registre
 - El desplaçament màxim és de 31 posicions. Només compten els 5 bits de menor pes de *Long*

tipus	format I	format R
esquerra	sll <i>rt,rs,shamt</i>	sllv <i>rd,rs,rt</i>
dreta (lògic)	srl <i>rt,rs,shamt</i>	srlv <i>rd,rs,rt</i>
dreta (aritmètic)	sra <i>rt,rs,shamt</i>	srav <i>rd,rs,rt</i>

Shamt: **shift amount**

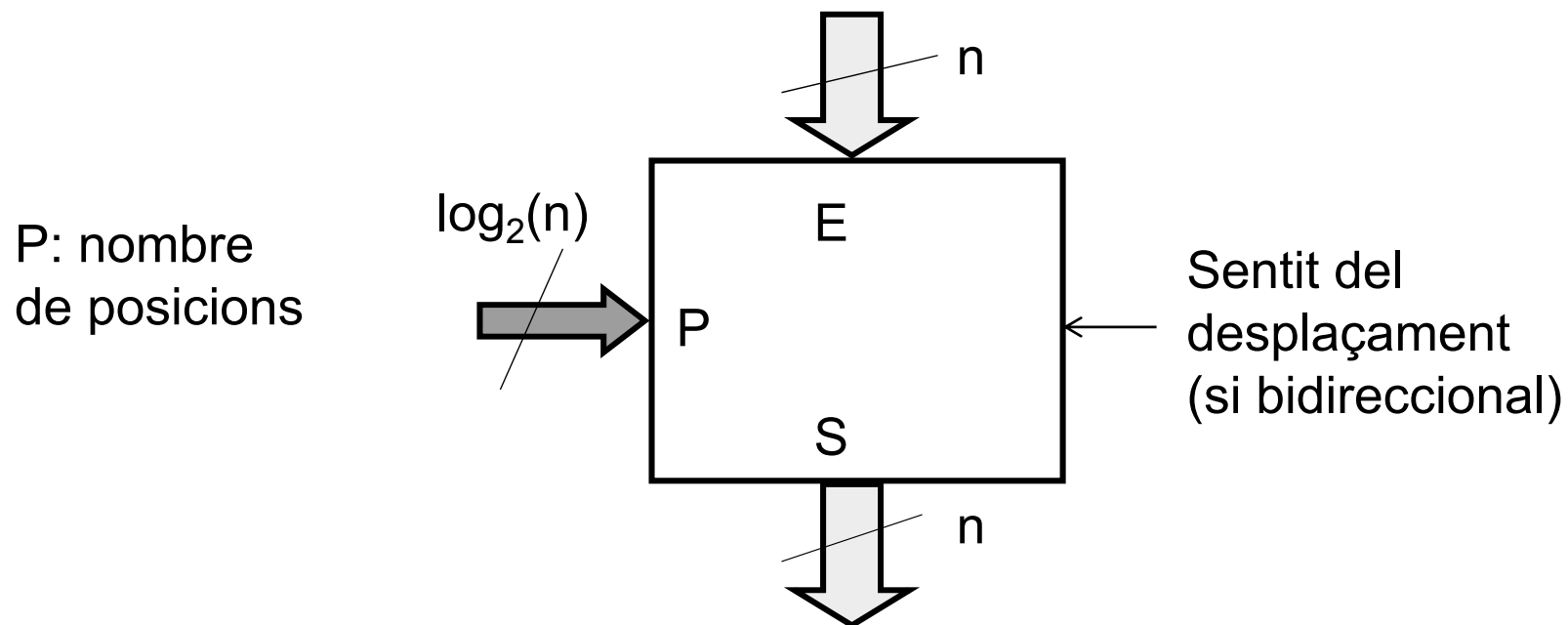
3.2 Multiplicació i divisió en el MIPS

- Instruccions de multiplicació i divisió generals
 - Dos registres especials de 32 bits: HI i LO
 - Combinats formen un registre de 64 bits
 - Operacions
 - `mult` \$2, \$3: HI-LO \leftarrow \$2*\$3; Operands amb signe
 - `multu` \$2, \$3: HI-LO \leftarrow \$2*\$3; Operands positius sense signe
 - `div` \$2, \$3: LO \leftarrow \$2/\$3; HI \leftarrow \$2 mod \$3; Amb signe
 - `divu` \$2, \$3: LO \leftarrow \$2/\$3; HI \leftarrow \$2 mod \$3; Sense signe
 - Transferència de resultats
 - `mfhi` \$2: \$2 \leftarrow HI
 - `mflo` \$2: \$2 \leftarrow LO
 - Hi ha pseudoinstruccions que permeten emmagatzemar el resultat en un registre destinatari de propòsit general i multiplicar per constants
 - Cap d'aquestes instruccions comprova desbordaments o divisió per zero: cal fer-ho per software

3.3 Operadors de desplaçament

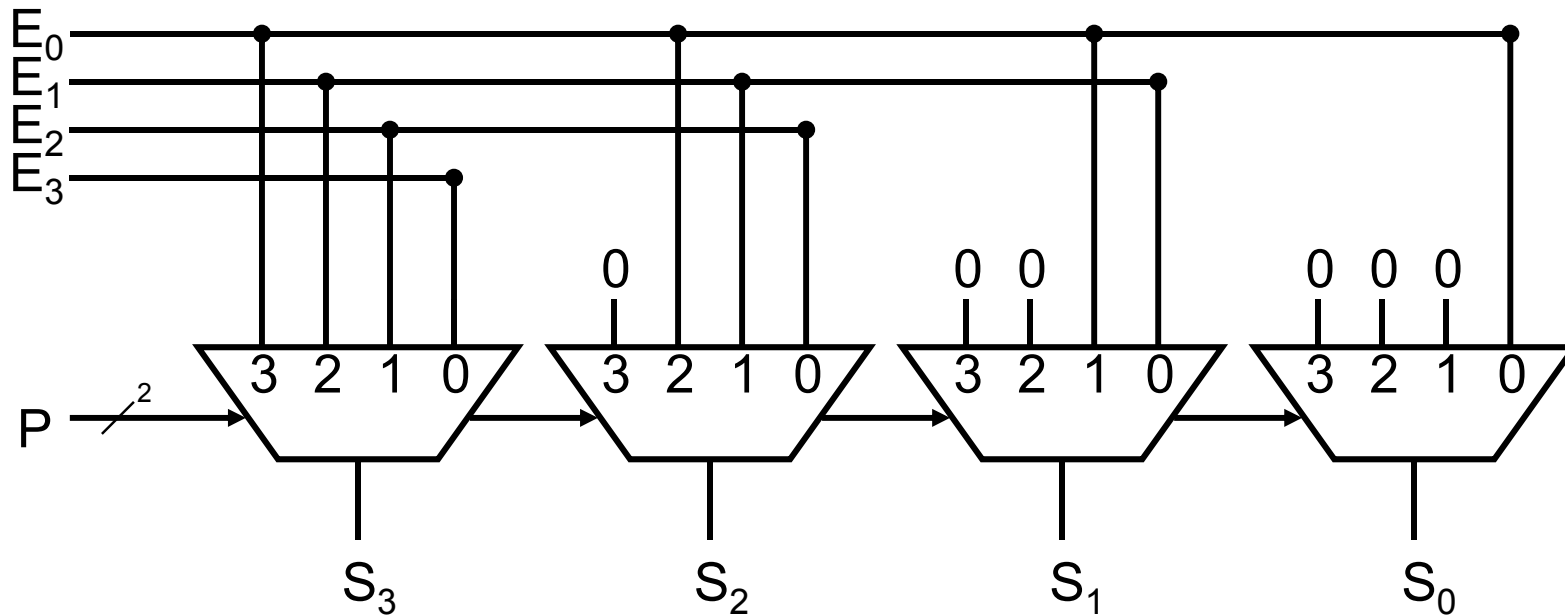
■ El Barrel Shifter

- Un Barrel Shifter és un circuit que permet realitzar desplaçaments variables sobre dades de n bits
- Pot implementar-se mitjançant multiplexors
- Depenent del disseny, fa desplaçaments lògics o aritmètics cap a la dreta, cap a l'esquerra o bidireccionals
- Connexions:



3.3 Operadors de desplaçament

- Barrel shifter: exemple de disseny
 - Implementació d'un operador de desplaçament lògic cap a l'esquerra (sll) per a dades de 4 bits

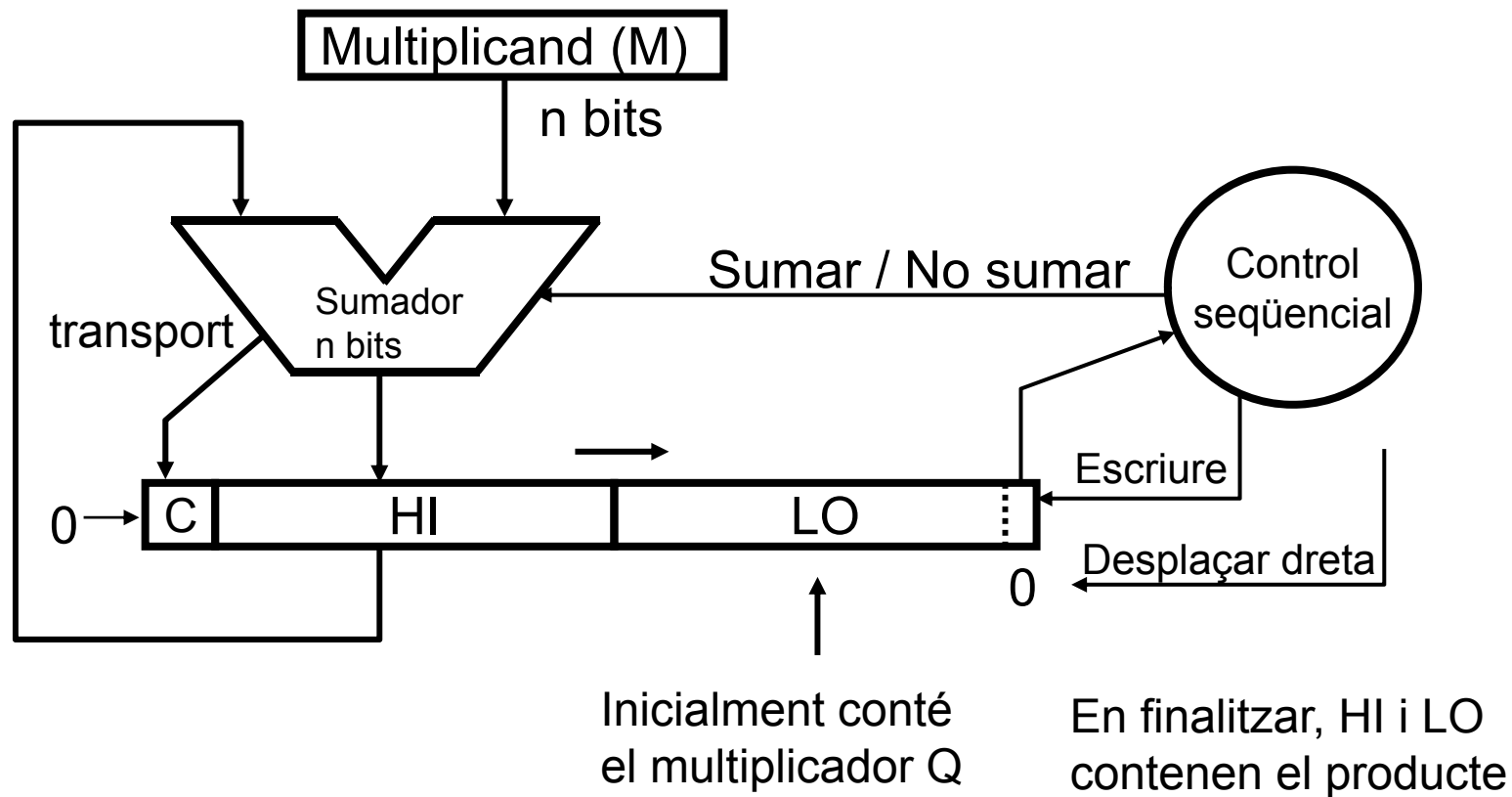


3.4 Operadors de multiplicació sense signe

- Operadors seqüencials aritmètics
 - Són circuits seqüencials síncrons que fan una operació donada
 - Necessiten un cert nombre de cicles de rellotge per a fer l'operació
 - El cicle de rellotge s'ajusta per a que puguin actuar els circuits
 - Si un operador necessita n cicles de t segons per a una operació,
 - el temps d'operació serà $T = n \times t$
 - la productivitat serà $P = f/n$, on $f = 1/t$ és la freqüència de treball del relloge
- Notació:
 - M = Multiplicand; m_i = bit i -èssim de M
 - Q = Multiplicador; q_i = bit i -èssim de Q
 - P = Producte; p_i = bit i -èssim del producte
 - n = Nombre de dígit dels operands M i Q (de 0 a $n-1$)

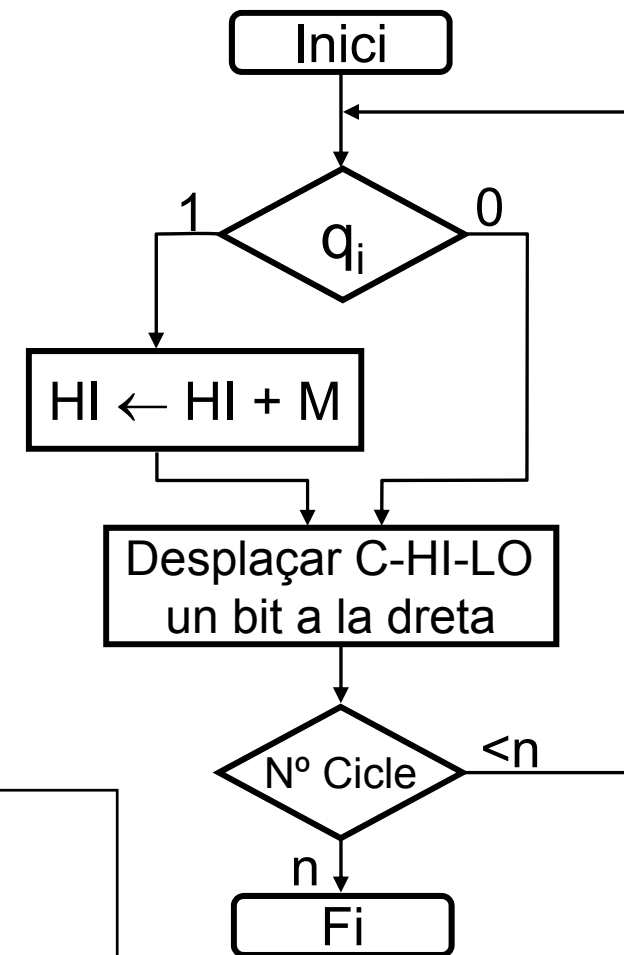
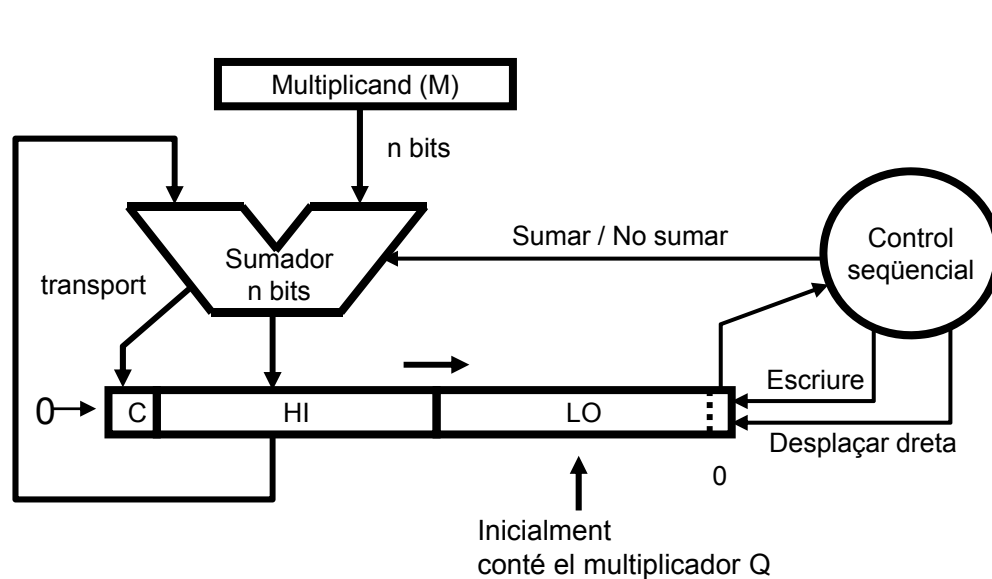
3.4 Operadors de multiplicació sense signe

- Operador per a l'algorisme de sumes i desplaçaments
 - M i Q de n bits; P de 2n bits



3.4 Operadors de multiplicació sense signe

- Algorisme amb què funciona l'operador seqüencial



- L'algorisme requereix n cicles
- En cada cicle cal fer fins a una suma i un desplaçament

3.4 Operadors de multiplicació sense signe

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

Cicle	Acció	C-HI-LO
0	Valors inicials	0 0000 010 <u>1</u>
1	$HI := HI + M$	0 1011 0101
	Desplaçar C-HI-LO 1 bit a la dreta	0 0101 101 <u>0</u>
2	No sumar	0 0101 1010
	Desplaçar C-HI-LO 1 bit a la dreta	0 0010 110 <u>1</u>
3	$HI := HI + M$	0 1101 1101
	Desplaçar C-HI-LO 1 bit a la dreta	0 0110 111 <u>0</u>
4	No sumar	0 0110 1110
	Desplaçar C-HI-LO 1 bit a la dreta	0 0011 0111

3.4 Operadors de multiplicació sense signe

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

Cicle	Acció	C-HI-LO
0	Valors inicials	0 0000 101 <u>1</u>
1		
2		
3		
4		

Solució: 1000 1111

3.5 Operadors de multiplicació amb signe

- Algorisme 1: Tractament del signe por separat
 - Es tracta de multiplicar els valors absoluts i considerar el signe a banda. Tot considerant-hi que $\text{Signe}(X)$ és el bit de signe de X :

```
Signe_Prod ← Signe(M) XOR Signe(Q) ;  
si M < 0 aleshores M ← -M; fi si;  
si Q < 0 aleshores Q ← -Q; fi si;  
P ← M × Q;  
si Signe_Prod = 1 aleshores P ← -P; fi si;
```

- Inconvenients
 - Cal cert hardware addicional per al cas particular de nombres amb signe, a fi de complementar M , Q o P
 - Hi ha d'altres mètodes per a tractar uniformement el producte de nombres amb o sense signe (algorisme de Booth, més endavant)

3.5 Operadors de multiplicació amb signe

- Algorisme 2: Sumes i desplaçaments amb extensió del signe
 - L'algorisme pot funcionar amb signe només si Q es positiu
 - Per a això, cal estendre el signe dels productes intermedis
 - Exemple: $n=4$; $M=-3$; $Q=6$; Representats en complement a 2

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

Per a funcionar en qualsevol cas, poden processar-se els signes de M i Q d'avantmà:

```
si Q < 0 aleshores
    Q ← -Q;
    M ← -M;
fi si
P ← M × Q;
```

Encara és més senzill que l'anterior; també demana processar per separat els casos de multiplicació de nombres amb o sense signe i complementar M o Q

3.5 Operadors de multiplicació amb signe

■ Algorisme de Booth

- Consisteix a recodificar el multiplicador com a una suma de potències positives o negatives de la base: usa els dígit 0, +1 i -1
- Per exemple:
 - El nombre 30 pot expressar-se com $(32 - 2)$
 - $30_{10} = 0011110_2 = 0 + 1\ 0\ 0\ 0 - 1\ 0_{\text{Booth}} = + 1 \cdot 2^5 - 1 \cdot 2^1$

							0	1	0	1	1	0	1	(45 ₁₀)
						×	0	+1	0	0	0	-1	0	(30 _{Booth})
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	0	0	1	0	1	1	0	1						← Còpia de M (sumar M)
0	0	0	0	0	0	0	0							
0	0	0	1	0	1	0	1	0	0	0	1	1	0	(1350 ₁₀)

3.5 Operadors de multiplicació amb signe

■ Algorisme de Booth

- Funciona amb nombres positius o negatius:
 - Multiplicació sense signe: suposar un bit de signe de $M = 0$
 - Multiplicació amb signe: estendre el MSB de M com a signe
- Exemple: Multiplicar amb i sense signe els nombres 1101_2 i $0+10-1_{\text{Booth}}$

Sense signe

Signe positiu

implícit	→	0	1	1	0	1		(13_{10})
		×	0	+1	0	-1		(3_{Booth})
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				
0	0	1	0	0	1	1	1	(39_{10})

Amb signe

Signe

implícit	→	1	1	1	0	1		(-3_{10})
		×	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	(-9_{10})

3.5 Operadors de multiplicació amb signe

- Recodificació del multiplicador pel mètode de Booth
 - S'han de considerar parelles de bits correlatius, de dreta a esquerra
 - Cal suposar un bit implícit = 0 a la dreta del LSB
 - Cal aplicar la taula de conversió següent :

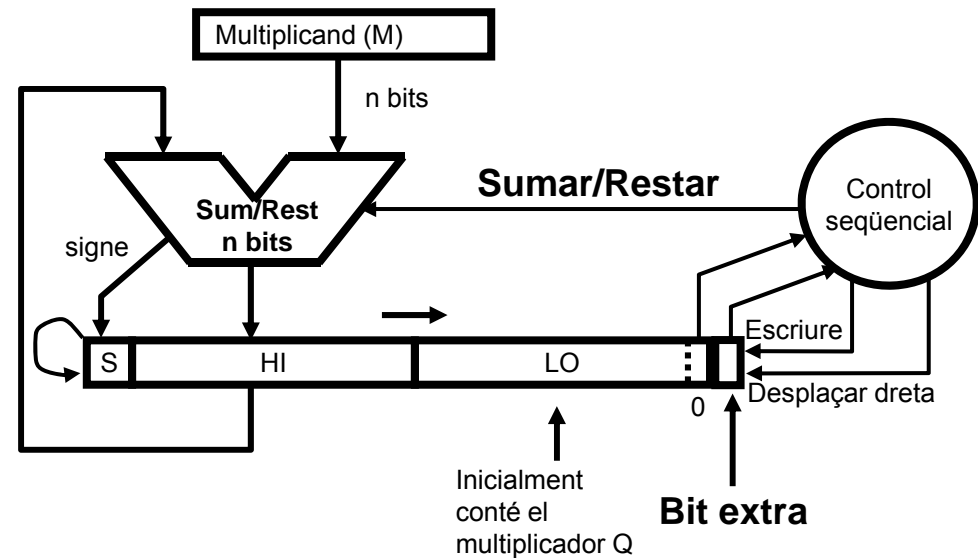
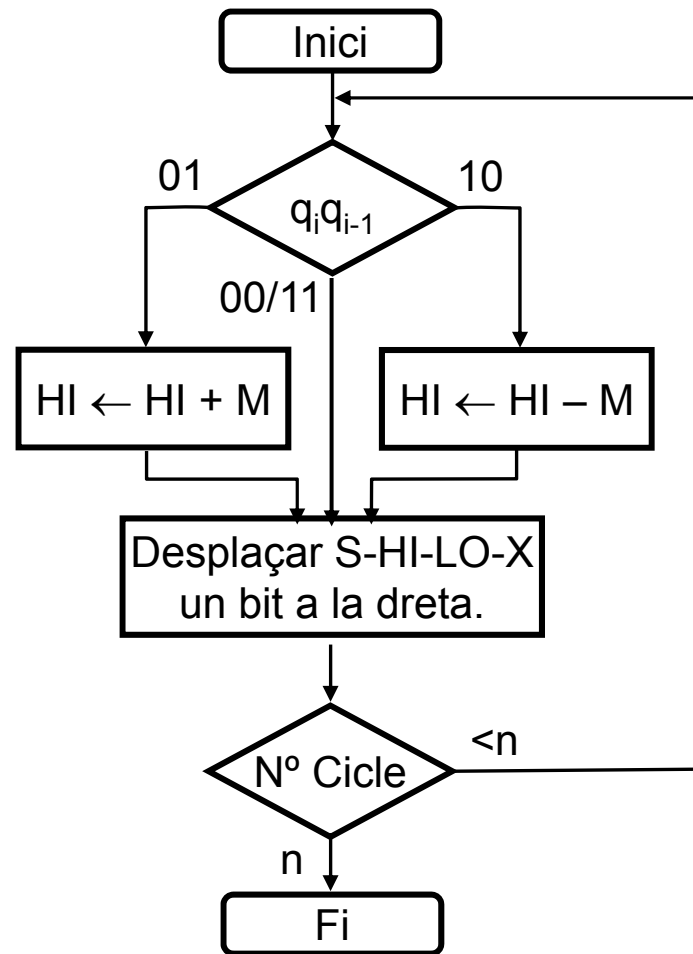
q_i	q_{i-1}	Dígit Booth
0	0	0
0	1	+1
1	0	-1
1	1	0

Per a recordar:
Dígit Booth = $q_{i-1} - q_i$

- Exemple: Obteniu el codi Booth de 1110 0111 0011 (-397_{10})
 - Solució: 0 0 -1 0 +1 0 0 -1 0 +1 0 -1 =
 $-1 \cdot 2^0 + 1 \cdot 2^2 - 1 \cdot 2^4 + 1 \cdot 2^7 - 1 \cdot 2^9 =$
 $-1 + 4 - 16 + 128 - 512 = -397$

3.5 Operadors de multiplicació amb signe

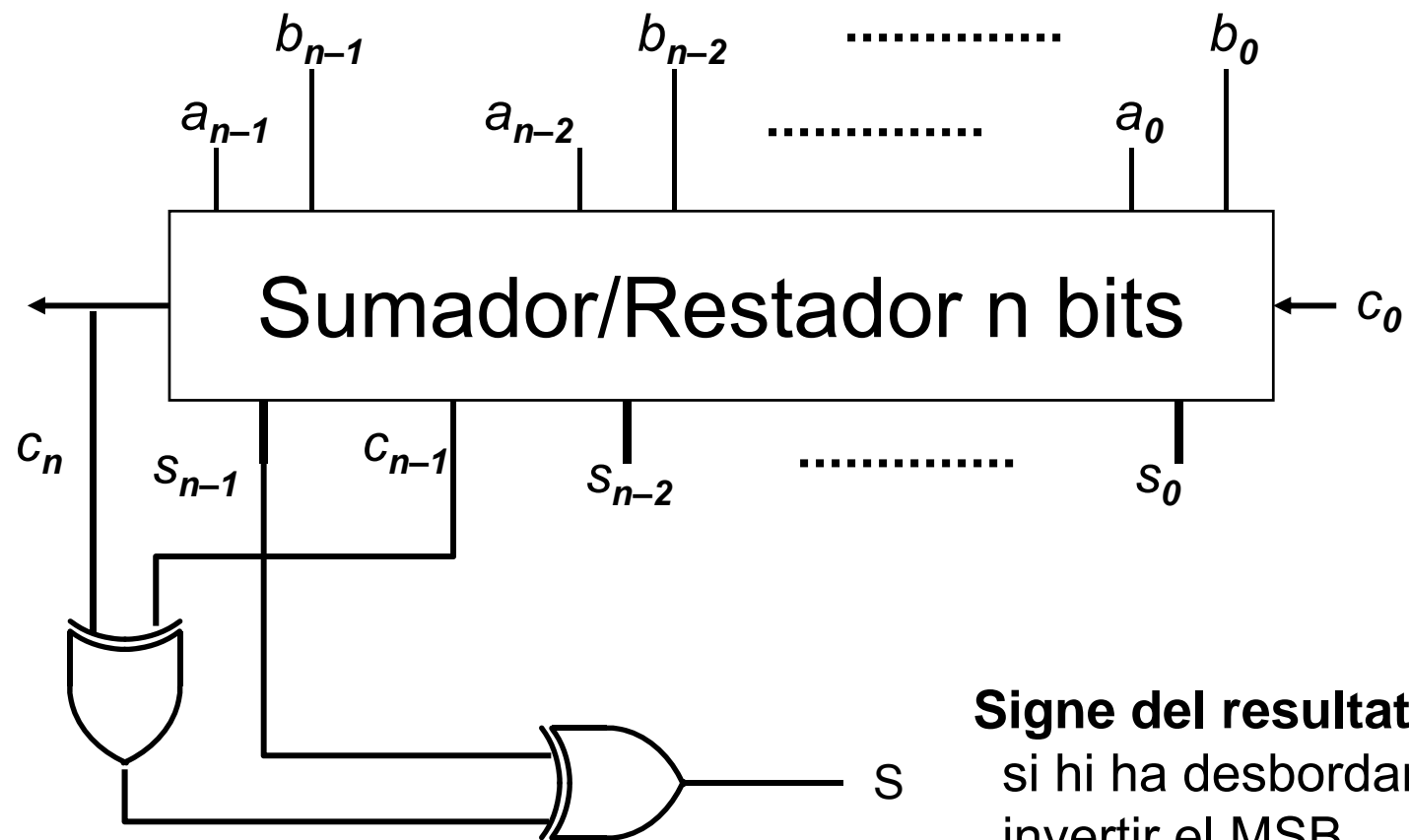
- Modificació d'algorisme 2 i operador seqüencial per a Booth



- L'algorisme demana n cicles
- En cada cicle cal fer una suma o una resta més un desplaçament

3.5 Operadors de multiplicació amb signe

- Detall del càlcul del bit de signe addicional



3.5 Operadors de multiplicació amb signe

- Exemple d'algorisme de Booth amb l'operador seqüencial
 - Amb signe; $n=4$; $M=0010_2$; $Q=1001_2$ $(2_{10} \times (-7_{10}) = -14_{10})$

Cicle	Acció	S-HI-LO-X	← Bit extra
0	Valors inicials	0 0000 1001 <u>0</u>	
1	Cas 10: $HI := HI - M$	1 1110 1001 0	
	Desplaçar S-HI-LO 1 bit a la dreta	1 1111 0100 <u>1</u>	
2	Cas 01: $HI := HI + M$	0 0001 0100 1	
	Desplaçar S-HI-LO 1 bit a la dreta	0 0000 1010 <u>0</u>	
3	Cas 00: No fer res	0 0000 1010 0	
	Desplaçar S-HI-LO 1 bit a la dreta	0 0000 0101 <u>0</u>	
4	Cas 10: $HI := HI - M$	1 1110 0101 0	
	Desplaçar S-HI-LO 1 bit a la dreta	1 1111 0010 1	

3.5 Operadors de multiplicació amb signe

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

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

Solució: 1110 1110

3.5 Operadors de multiplicació amb signe

- Recodificació de Q per parelles de bits
 - Extensió de Booth que redueix a la meitat el nombre de dígitos de Q, tot reduint així el nombre de productes intermedis que cal sumar

			Booth		Parelles	Acció
q_{i+1}	q_i	q_{i-1}	q'_{i+1}	q'_i	q''_i	
0	0	0	0	0	0	No res
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	No res

A cada iteració es desplaça S-HI-LO dues posicions a la dreta

3.5 Operadors de multiplicació amb signe

- Exemple de multiplicació amb recodificació per parelles
 - $n = 5$; $M = 01101_2 (13_{10})$; $Q = 11010_2 (-6_{10})$

Extensió del signe \rightarrow 1 1 1 0 1 0 0 \leftarrow Bit implícit



Recodificació per parelles \rightarrow

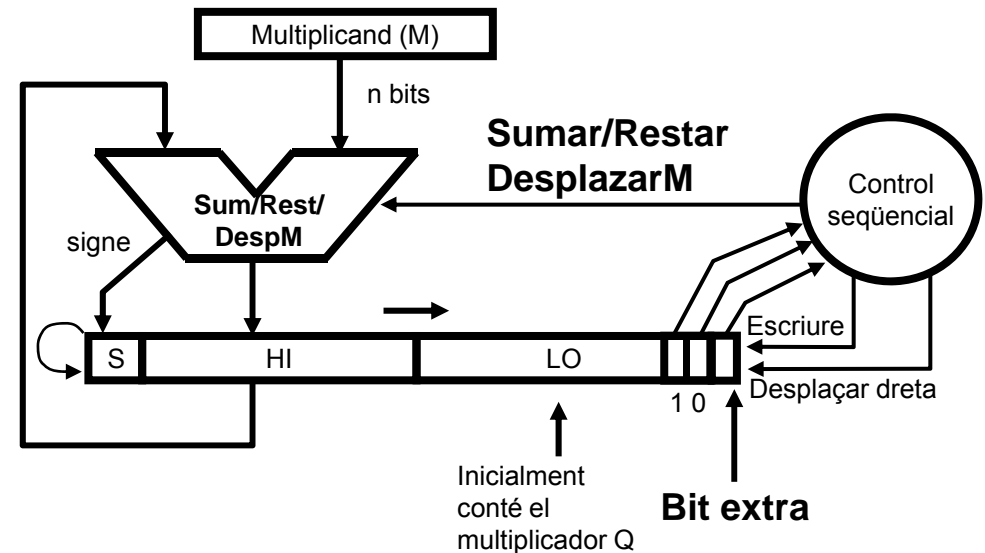
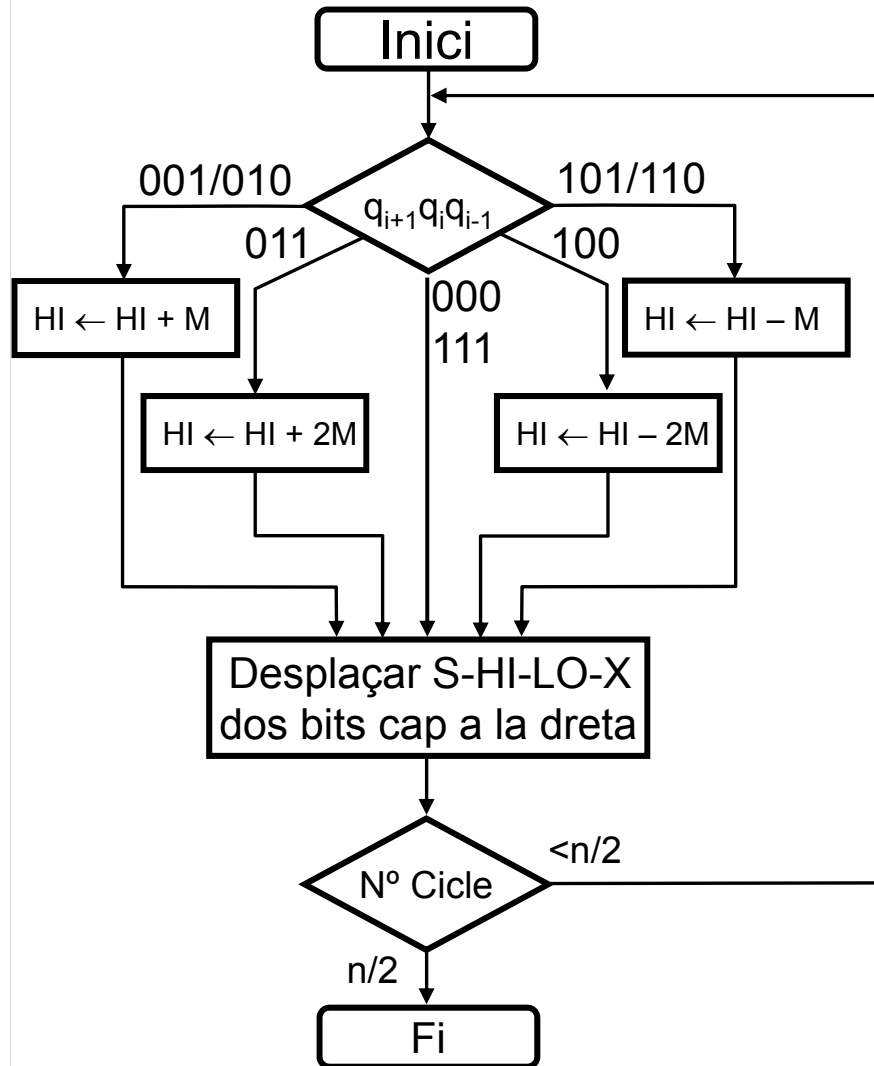
0	0	-1	1	-1	0
└──┘		└──┘		└──┘	
↓		↓		↓	
0		-1		-2	

						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		
No res	0	0	0	0	0	0				
	<hr/>									
	1	1	1	0	1	1	0	0	1	0

(-78_{10})

3.5 Operadors de multiplicació amb signe

- Modificació de l'algorisme 2 per a parelles de bits



Els registres HI i LO han de tindre un nombre parell de bits

- L'algorisme demana $n/2$ cicles
- En cada cicle cal fer fins a una suma o resta i dos desplaçaments

3.5 Operadors de multiplicació amb signe

- Exemple de multiplicació amb recodificació per parelles
 - $n=6$ (n ha de ser parell); $M=001101_2$ (13_{10}); $Q=111010_2$ (-6_{10})

Cicle	Acció	S-HI-LO-X
0	Valors inicials	0 000000 1110 <u>10</u> 0
1	Cas 100: $HI \leftarrow HI - 2M$	1 100110 111010 0
	Desplaçar S-HI-LO 2 bits a la dreta	1 111001 1011 <u>10</u> 1
2	Cas 101: $HI \leftarrow HI - M$	1 101100 101110 1
	Desplaçar S-HI-LO 2 bits a la dreta	1 111011 0010 <u>11</u> 1
3	Cas 111: No fer res	1 111011 001011 1
	Desplaçar S-HI-LO 2 bits a la dreta	1 111110 110010 1

Solució: 111100 110001

3.5 Operadors de multiplicació amb signe

- Exercici: $n = 6$; $M = 101001_2 (-23_{10})$; $Q = 001001_2 (9_{10})$
($-23_{10} \times 9_{10} = -207_{10}$)

Cicle	Acció	S-HI-LO-X
0	Valors inicials	0 000000 00100 <u>1</u> 0
1		
2		
3		

Solució: 111100 110001

La multiplicació seqüencial: resum

- Implementació de l'operador:
 - La multiplicació pot implementar-se mitjançant sumes i desplaçaments. En el hardware només cal:
 - registre de desplaçament
 - sumador o sumador/restador, en cas de multiplicació amb signe
 - circuit de control, si s'utilitza un operador seqüencial
- El mètode de Booth
 - Permet tractar de manera uniforme la multiplicació amb o sense signe
- La recodificació del multiplicador
 - Permet reduir el nombre de dígit del multiplicador i, per tant, el nombre d'iteracions de l'operador
 - La recodificació per parelles de bits permet reduir a la meitat el nombre de cicles requerit per a una multiplicació seqüencial
 - Altres recodificacions permeten reduir encara més el nombre d'iteracions