CIRCUITS ARITMÈTICS

Índex de conceptes

- **Sumador Complet**
- **Generació anticipada de carry**
- **Sumador modificat**
- **Restador**
- **ALUs**

Un tipus especial d'aplicacions dels circuits combinacionals són la realització de funcions aritmètiques: suma, resta, producte, divisió. D'algunes d'aquestes funcions existeixen circuits comercials estàndard.

Sumadors

Per realitzar la suma de 2 bits podem fer servir un semi-sumador, per sumar més de 2 ens caldrà tenir en compte l'arrossegament del bit anterior en ordre de significació.

Semi-sumador (S.S.)(*Half Adder*)

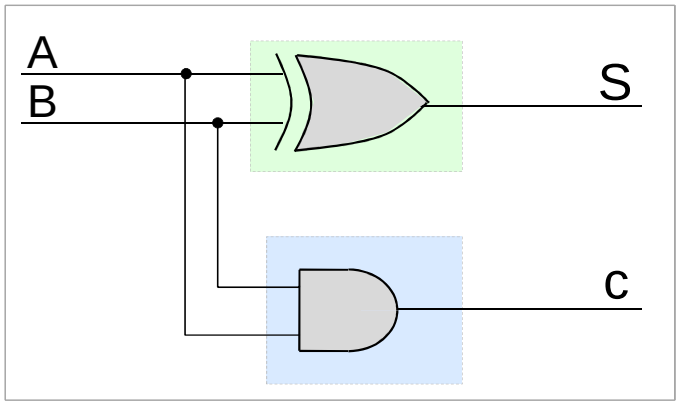
La suma de 2 bits ve donada a la taula següent:

A B	S c	Operació
0 0	0 0	0+0= 0
0 1	1 0	0+1= 1
1 0	1 0	1+0= 1
1 1	0 1	1+1=10(=2)

Si simplifiquem les dues funcions s (suma) i c (carry, arrossegament) per separat tenim:

$$S = \overline{A}\cdot B + A\cdot\overline{B} = A \oplus B$$
$$c = A\cdot B$$

Així, la funció suma de 2 bits serà la donada pel circuit semisumador (Half-adder, HA):

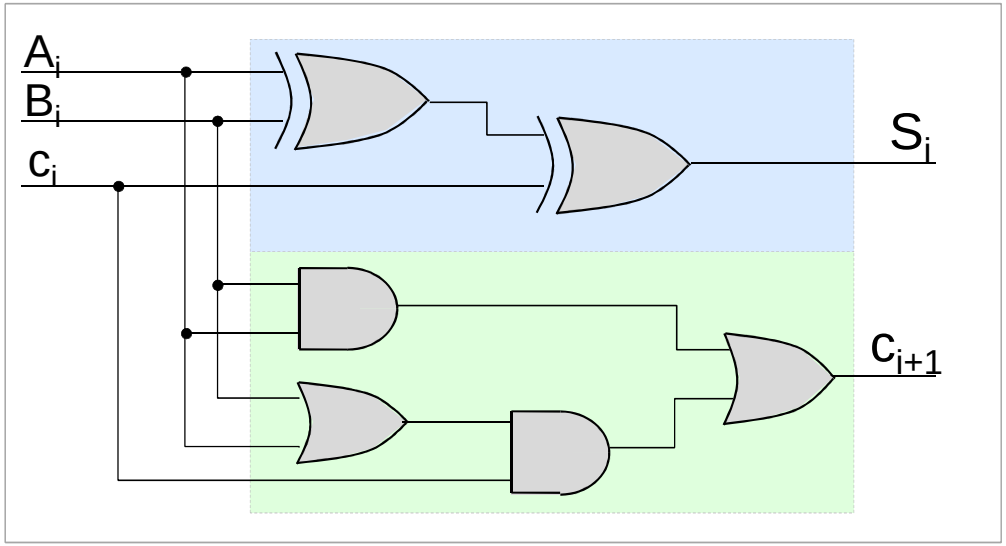


Sumador complet (S.C.)(*Full Adder*)

Per sumar números de 2 o més bits cal que tinguem present l'arrossegament (*carry*) que hem generat amb el bit anteriorment significatiu segons la taula següent:

c_i	A_i	B_i	S_i	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

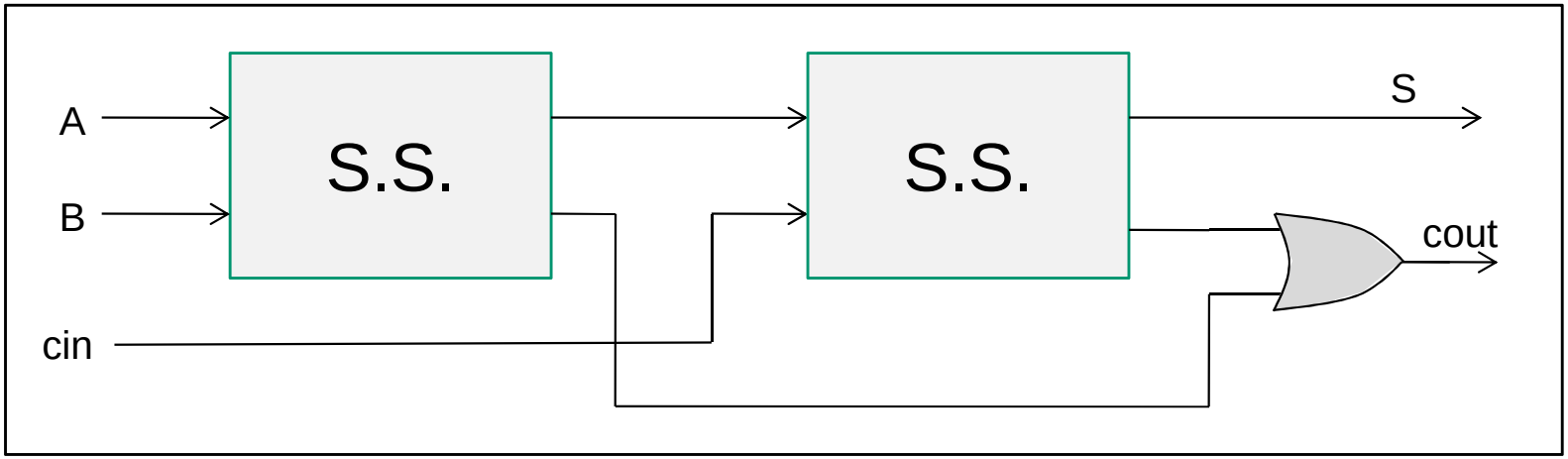
Fent la simplificació per a les dues funcions S_i i c_{i+1} per separat obtenim el circuit següent:



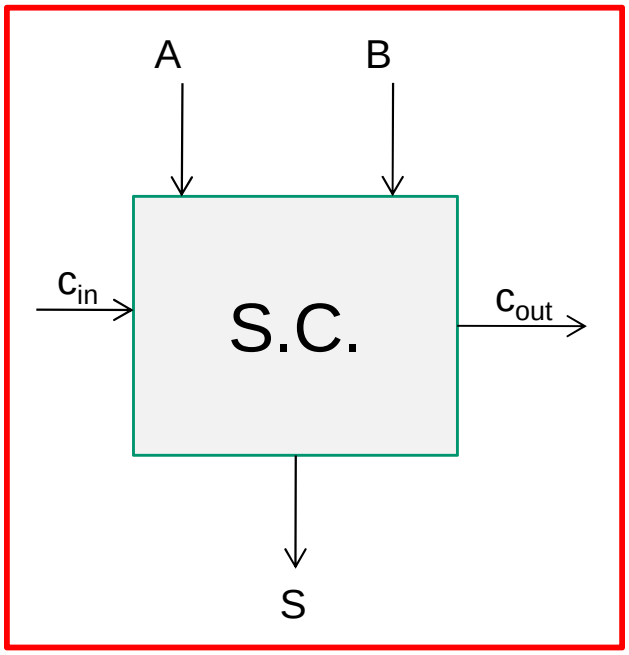
$$S_i = (A_i \oplus B_i) \oplus c_i$$

$$c_{i+1} = A_i \cdot B_i + c_i \cdot (A_i + B_i)$$

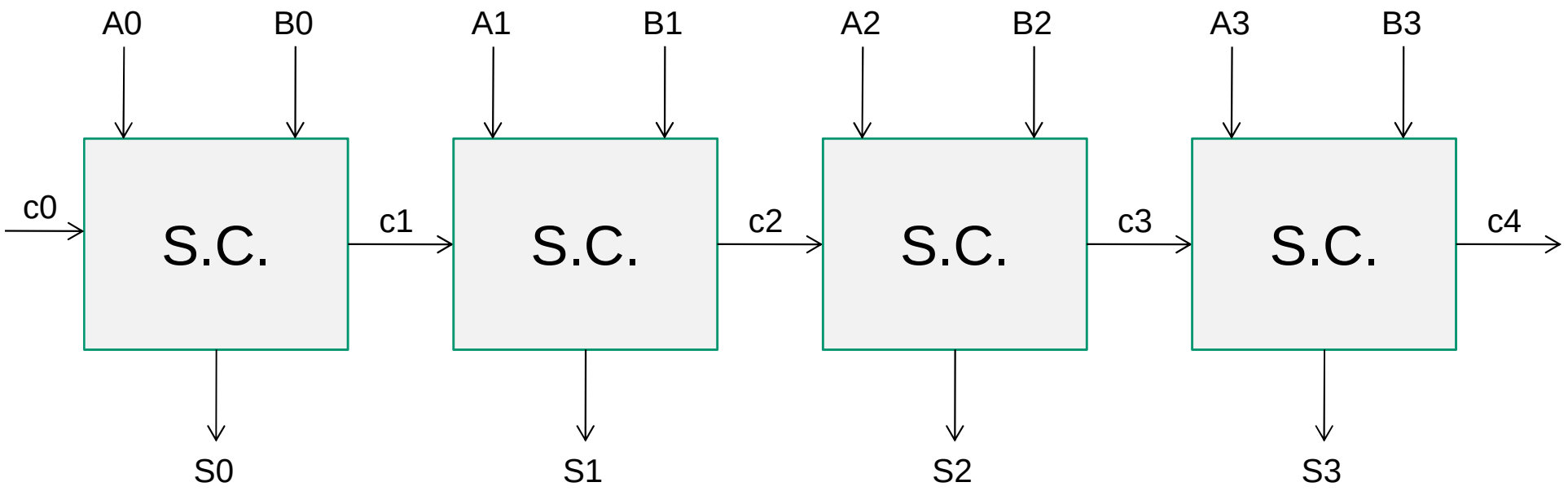
També podem realitzar el sumador complet a partir de dos semisumadors:



Tal com es dedueix de la taula de la veritat, les tres entrades de dades (A, B i c_{in}) són equivalents.



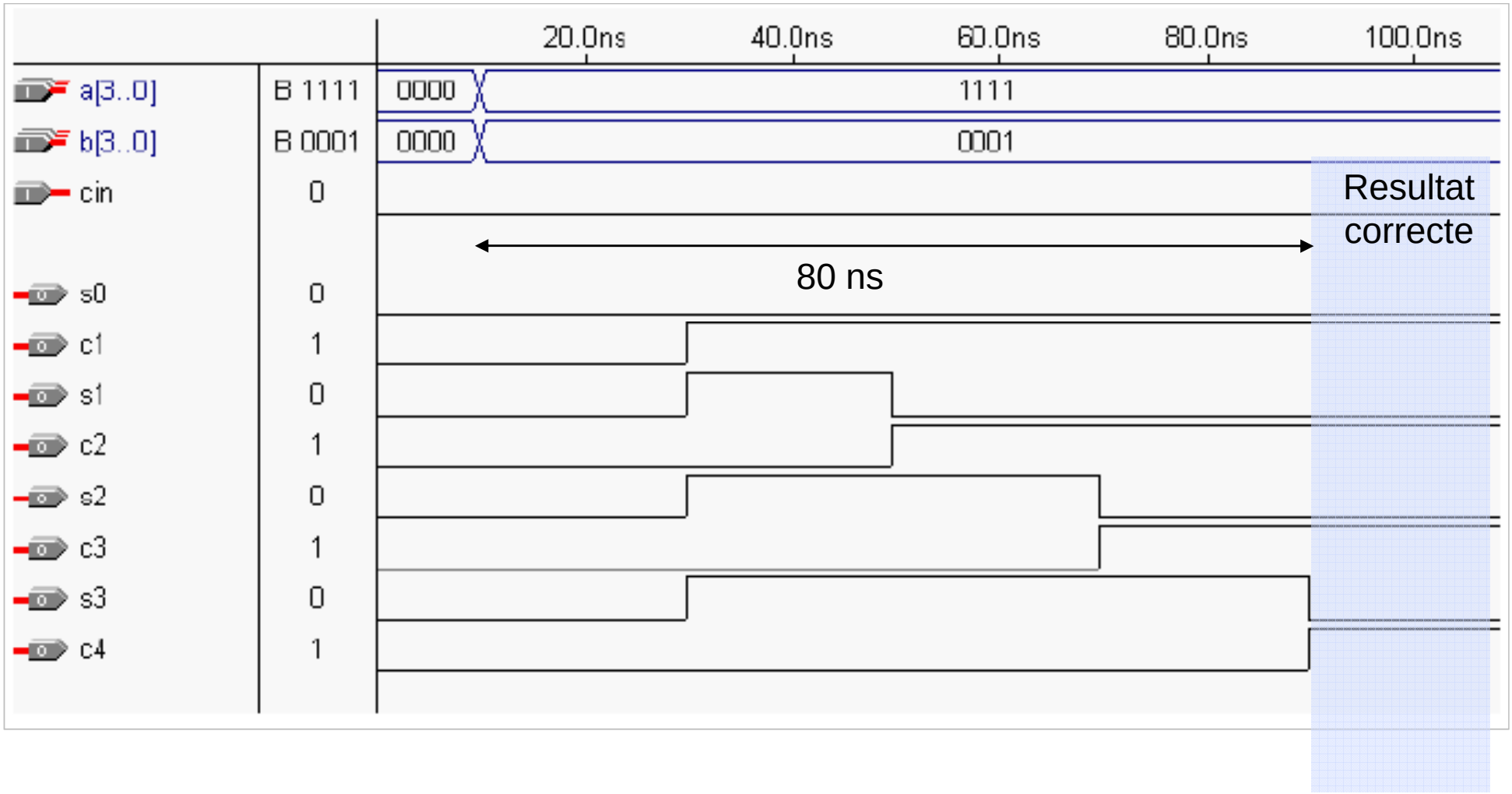
Ripple-carry adder (propagador del carry)



Consisteix en connectar el carry de sortida del sumador i-èssim al carry d'entrada del sumador i+1-èssim.

Problema: Augmentant el nombre de bits a sumar, les etapes augmentem, per tant es sistema es molt lent (retard proporcional al nombre de bits)

Aquest circuit té l'inconvenient que per fer la suma ha de generar l'arrossegament de cada bit a partir de la suma d'aquests. A l'exemple es necessiten 4 retards, 80 ns, per presentar el resultat final correcte.



Sumador amb generació anticipada del carry

Per tal de fer les sumes de forma més ràpida podem fixar-nos en com s'obté el terme d'arrossegament:

$$c_{i+1} = a_i \cdot b_i + (a_i + b_i) \cdot c_i \quad = \quad a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$$

En aquest cas podeu comprovar que la XOR és equivalent a la OR, el terme 11 que li falta a la XOR l'aporta la AND $a_i b_i$

Definim dos termes en aquest arrossegament, el terme generador (es genera al sumador d'1 bit) i el terme propagador (generat com a conseqüència de l'arrossegament d'entrada):

$$\textit{Terme generador} \quad G_i = a_i \cdot b_i$$
$$\textit{Terme propagador} \quad P_i = (a_i \oplus b_i)$$

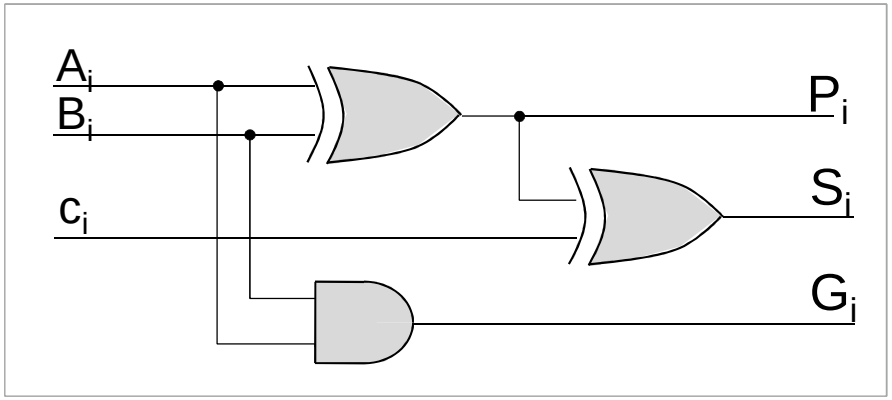
Per tant l'arrossegament l'obtenim a partir de:

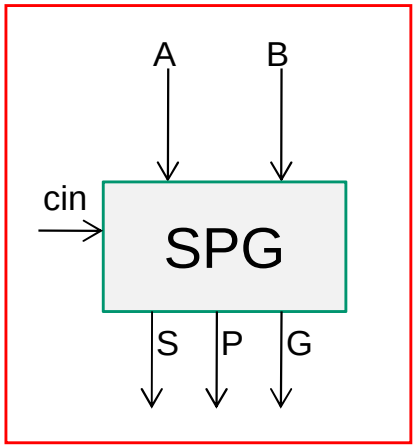
$$c_{i+1} = G_i + P_i \cdot c_i$$

Si apliquem aquesta relació de recurrència a la suma d'n bits, tenim:

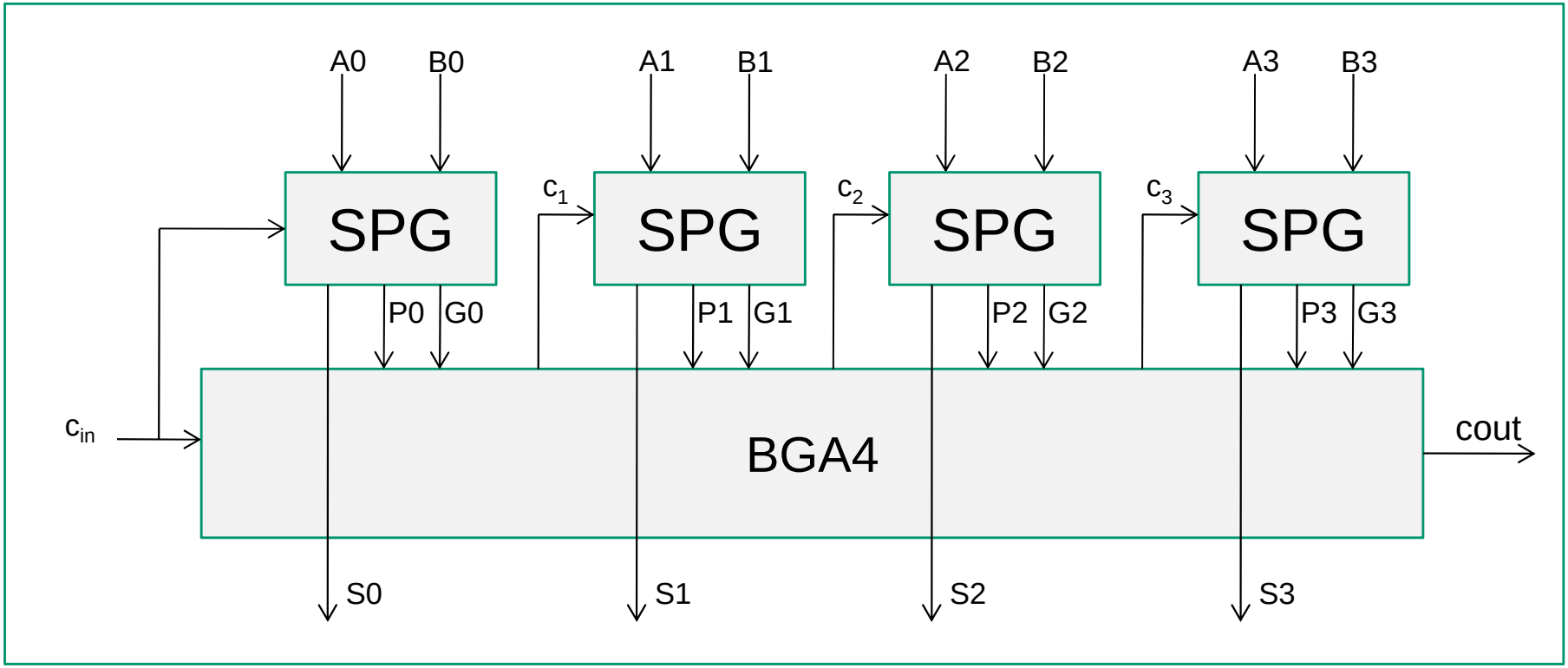
$$c_1 = G_0 + P_0 c_0$$
$$c_2 = G_1 + P_1 c_1 = G_1 + P_1 G_0 + P_1 P_0 c_0$$
$$c_3 = G_2 + P_2 C_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 c_0$$
$$c_4 = G_3 + P_3 C_3 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 c_0$$
$$\vdots$$
$$c_i = G_{i-1} + P_{i-1} c_{i-1} = G_{i-1} + P_{i-1} G_{i-2} + P_{i-1} P_{i-2} G_{i-3} + \ldots + P_i P_{i-1} \ldots P_1 c_0$$

Per tal de realitzar aquesta operació podem definir un sumador modificat que dóna els termes generador i propagador (**bloc SPG**):





Per tal de realitzar la suma de 4 bits fem servir 4 sumadors SPG i un bloc generador d'arrossegament (BGA4), que realitza les sumes i productes dels termes generador i propagador:



Restador

Tal com s'ha vist al tema 1, per fer resta de dos números és molt útil treballar amb la representació dels números negatius en complement a 2 (Ca2): *es complementa el substraend i se li suma al minuend*.

El **complement a 2** es realitza a partir del complement a 1 (inversió) i sumant-li 1 al resultat:

1. El **complement a 1** l'obtenim a partir de la funció **XOR**.
2. La **suma d'1** la realitzem utilitzant **l'arrossegament d'entrada**.

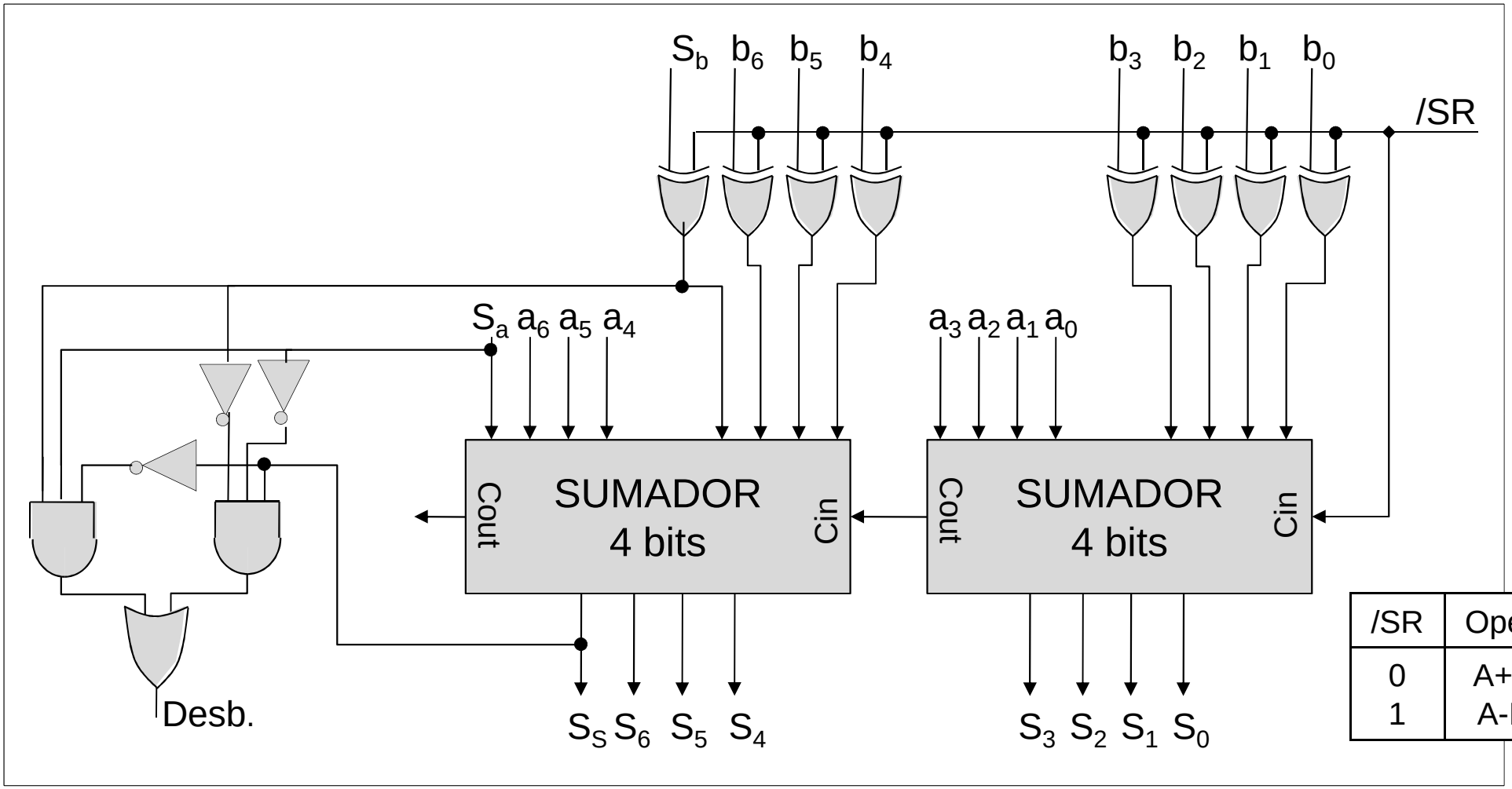
Exemple: per tal de realitzar un circuit que permeti sumar i restar de *dos números de 7 bits + bit de signe*, es poden utilitzar dos sumadors de 4 bits.

Aquí poden fer un circuit per detectar el desbordament. Es produirà quan:

(i) Si sumem dos nombres positius (Sa=0, Sb=0) i el bit de signe del resultat és (Ss=1)
(ii) Si sumem dos nombres negatius (Sa=1, Sb=1) i el bit de signe del resultat és (Ss=0)
(iii) Si restem un nombre negatiu d'un positiu (Sa=0, Sb=1) i el bit de signe del resultat és (Ss=1)
(iv) Si restem un nombre positiu d'un negatiu (Sa=1, Sb=0) i el bit de signe del resultat és (Ss=0)

SUMADOR/RESTADOR de dos nombres de 8 bits

Amb detecció de desbordament (*overflow*)



/SR	Oper.
0	A+B
1	A-B

Unitats aritmètico-lògiques (ALU)

És un circuit combinacional que permet de realitzar funcions aritmètiques o lògiques en funció d'unes variables de control.

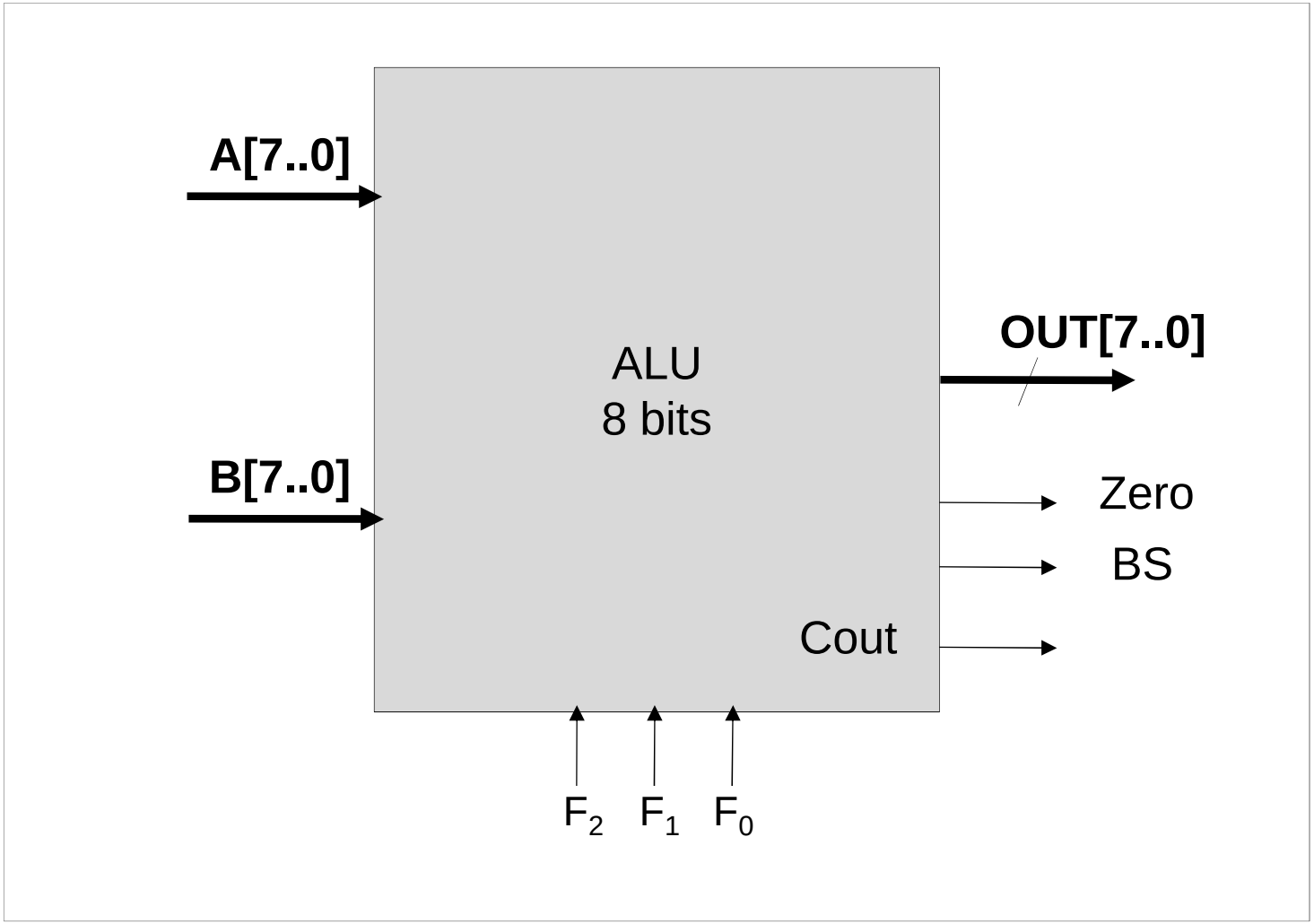
Són la unitat central dels microprocessadors i d'altres sistemes digitals.

Treballa amb paraules d'n bits i, en funció d'unes variables de control F, realitza funcions aritmètiques o lògiques.

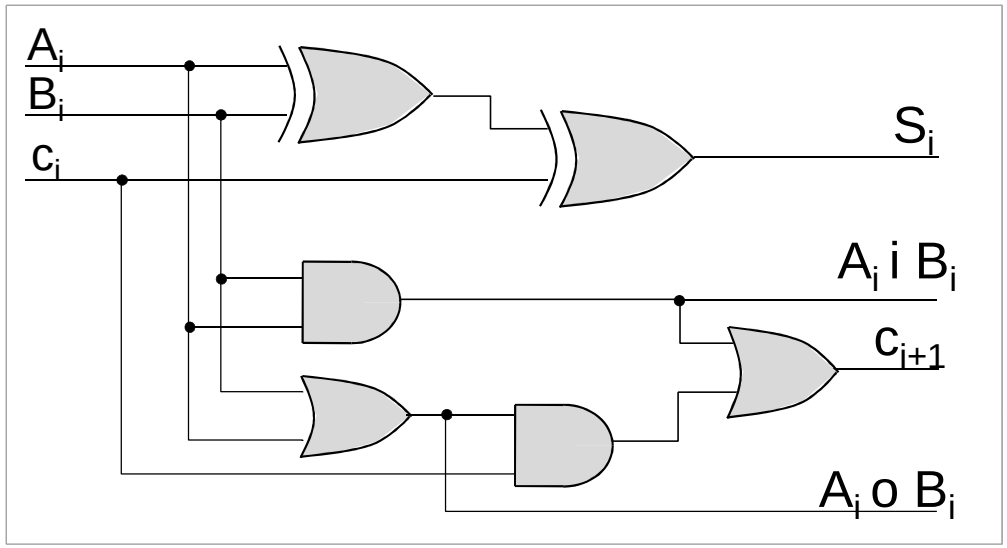
Com a exemple dissenyarem una ALU amb el següent llistat d'operacions:

Tipus	Variables de selecció F2 F1 F0	OPERACIO
Funció aritmètica	0 0 0	A + B
Funció aritmètica	0 0 1	A – B
Funció aritmètica	0 1 0	A – 1
Funció aritmètica	0 1 1	A + 1
Funció aritmètica	1 0 0	Ca2 B
Funció lògica	1 0 1	A or B
Funció lògica	1 1 0	A and B
Funció lògica	1 1 1	Transferir A

Dissenyarem una ALU de 8 bits amb 8 funcions Aritmètico/Lògiques en base al sumador modificat



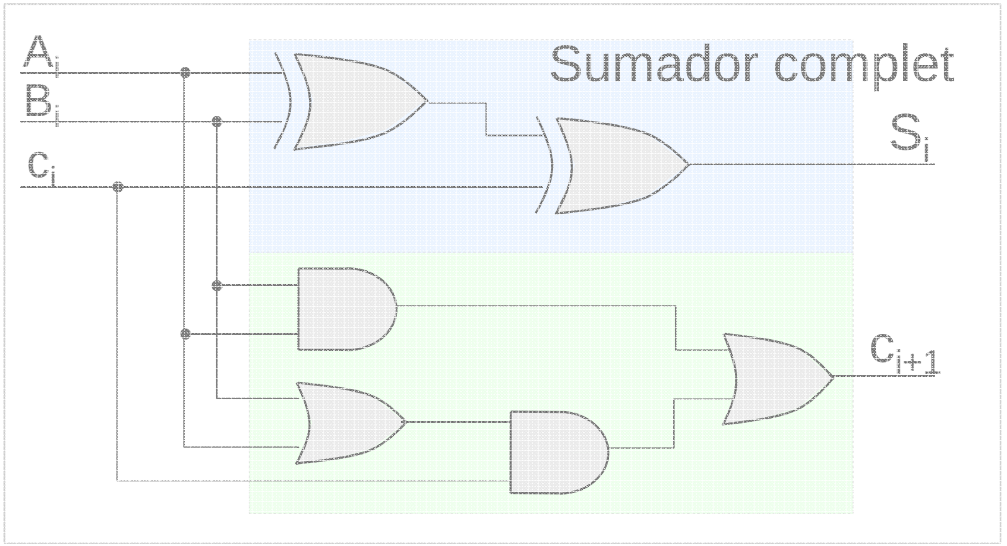
Per tal de fer aquestes funcions es pot utilitzar un sumador igual que el presentat anteriorment, però modificat, per tal de poder extreure altres funcions necessàries apart de la suma i el *carry* :



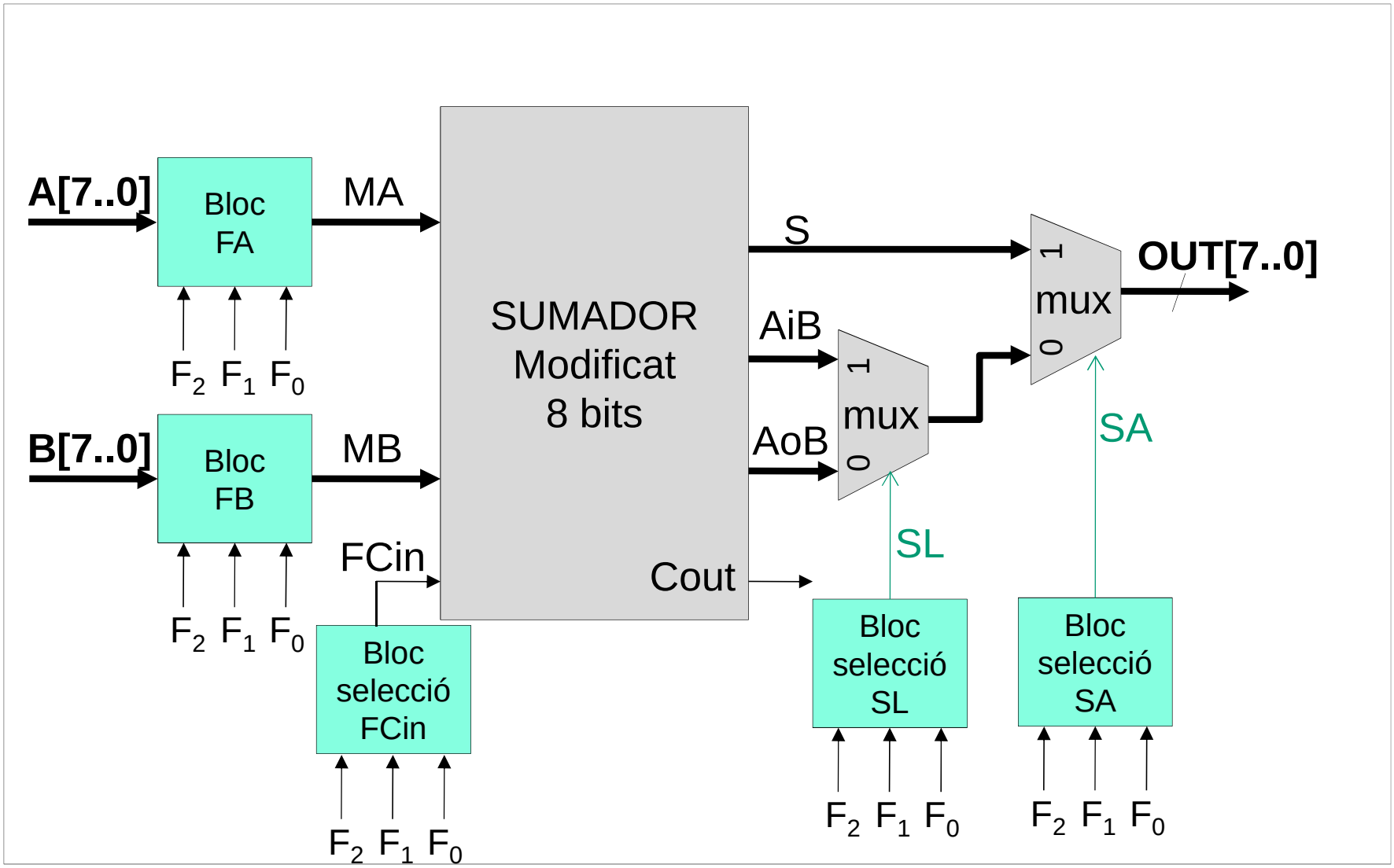
sumador complet modificat

Extraiem les funcions (AiB) i (AoB)

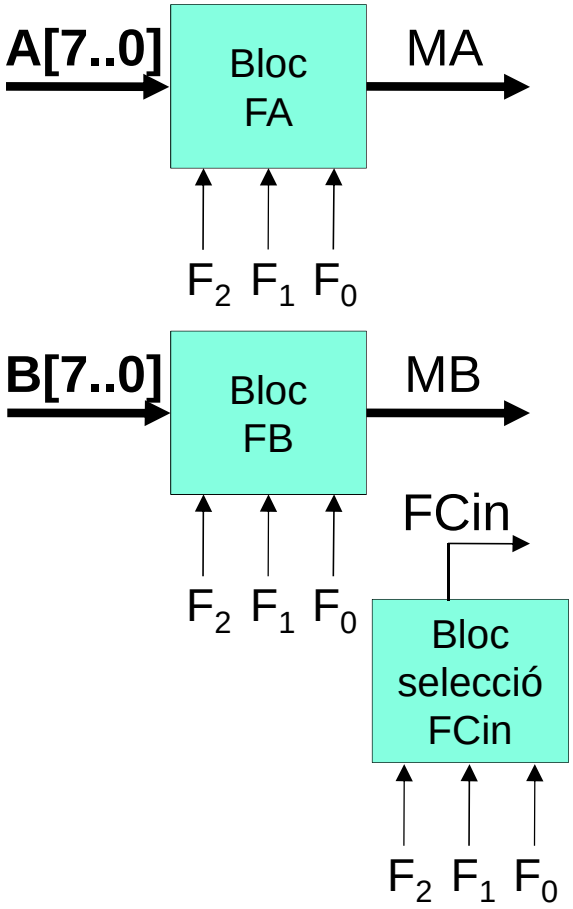
Podríem extreure també la A⊕B si fos necessari

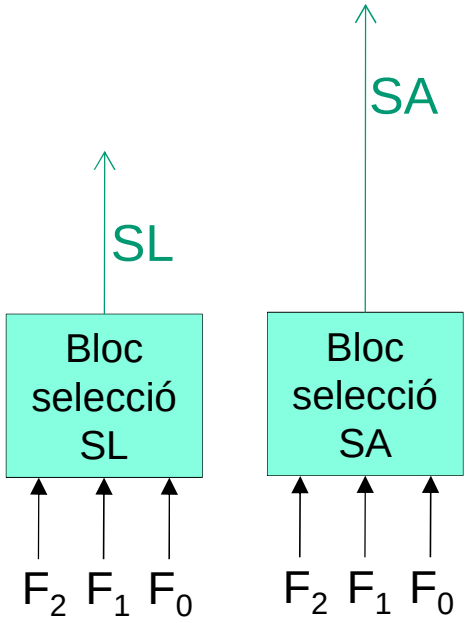


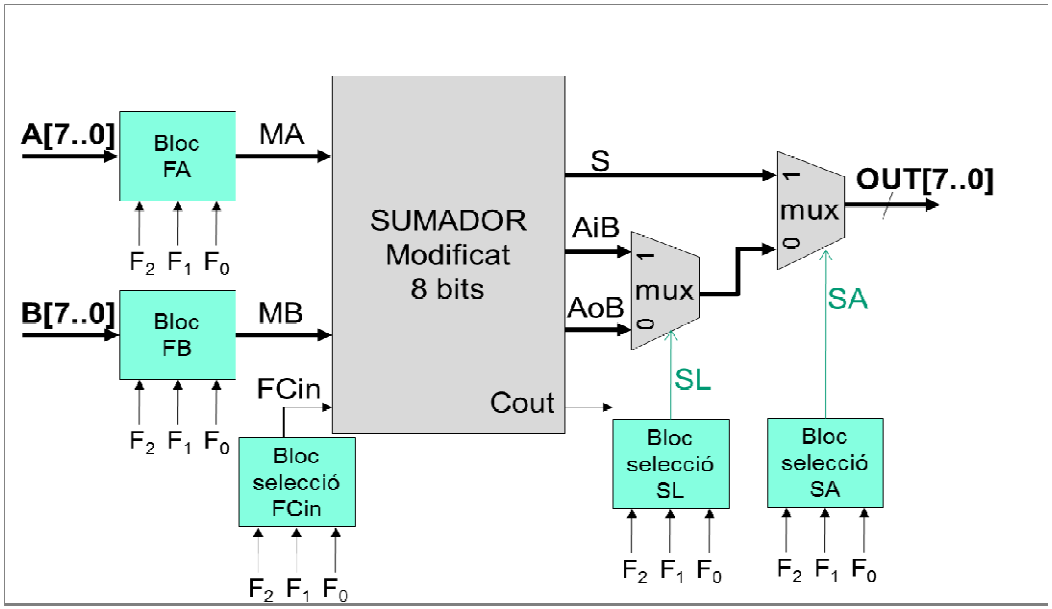
S'han definit unes variables de control SL i SA, així com uns blocs FA, FB i FCin que modifiquen les variables d'entrada del sumador modificat per tal de poder realitzar les operacions amb el mínim de blocs possibles.



Cal dissenyar les funcions lògiques dels blocs de selecció de sortida i d'acondicionament de les entrades



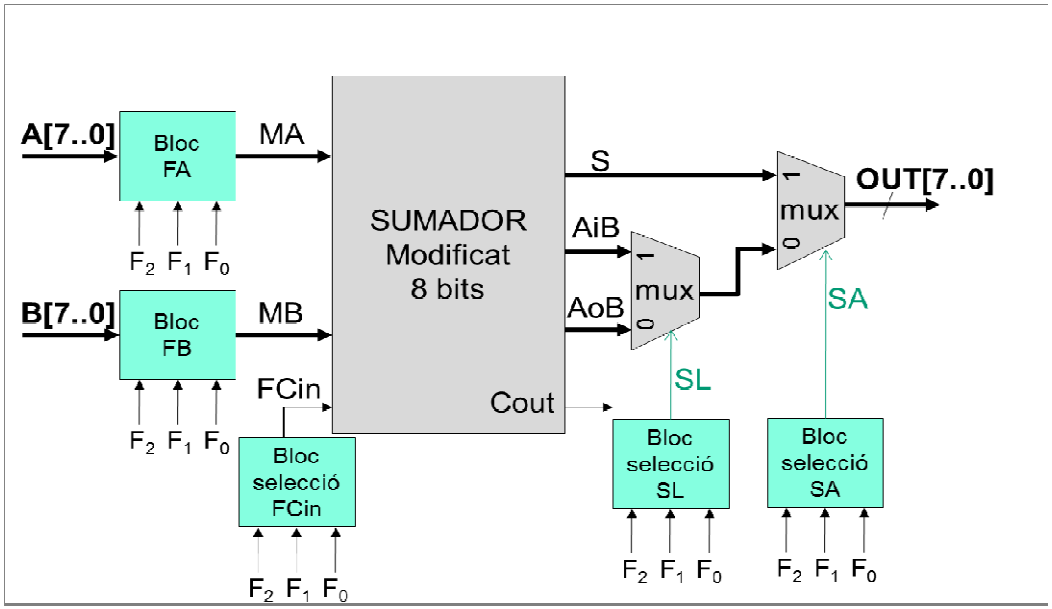




Tipus	Variables de selecció F2 F1 F0	OPERACIO
Funció aritmètica	0 0 0	A + B
Funció aritmètica	0 0 1	A – B
Funció aritmètica	0 1 0	A – 1
Funció aritmètica	0 1 1	A + 1
Funció aritmètica	1 0 0	Ca2 B
Funció lògica	1 0 1	A or B
Funció lògica	1 1 0	A and B
Funció lògica	1 1 1	Transferir A

En base a la taula de funcions construïm la taula de veritat dels blocs que hem de dissenyar

F2	F1	F0	Operació	MA[7..0]	MB[7..0]	FCin	Sortida	SA	SL
0	0	0	A + B	A[7..0]	B[7..0]	0	S	1	X
0	0	1	A – B	A[7..0]	/B[7..0]	1	S	1	X
0	1	0	A – 1	A[7..0]	11111111	0	S	1	X
0	1	1	A + 1	A[7..0]	00000000	1	S	1	X
1	0	0	Ca2 B	00000000	/B[7..0]	1	S	1	X
1	0	1	A or B	A[7..0]	B[7..0]	X	OR	0	0
1	1	0	A and B	A[7..0]	B[7..0]	X	AND	0	1
1	1	1	Transferir A	A[7..0]	11111111	X	AND	0	1



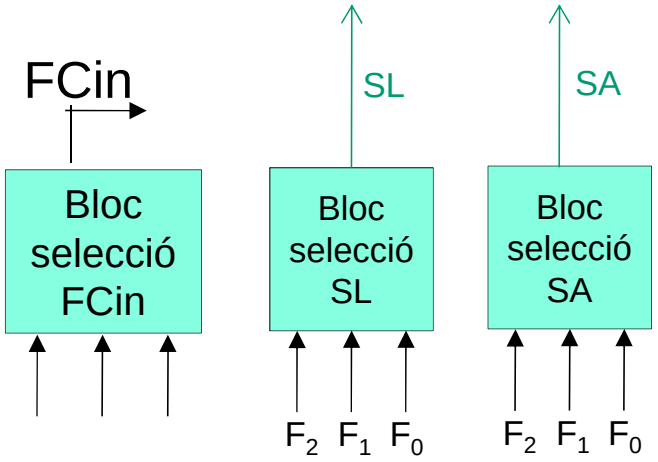
Per calcular MA i MB haurem de postular amb quines portes es podem trobar aquestes funcions de sortida de cada bloc.

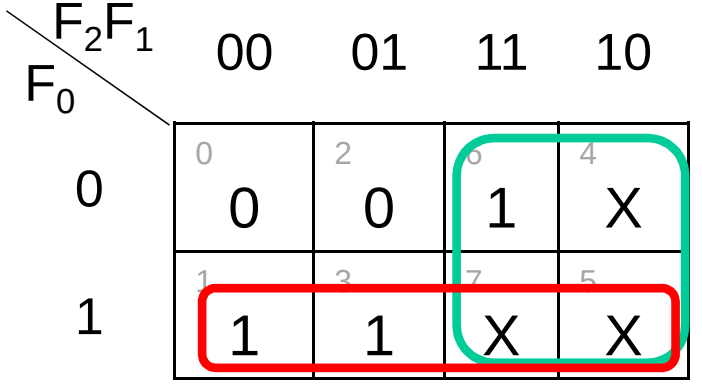
Ara ja podem calcular els mapes de Karnaugh de FCin, SA i SL (on hi ha 0's, 1's i X's) en funció de F2, F1 i F0.

F2	F1	F0	Operació	MA[7..0]	MB[7..0]	FCin	Sortida	SA	SL
0	0	0	A + B	A[7..0]	B[7..0]	0	S	1	X
0	0	1	A – B	A[7..0]	/B[7..0]	1	S	1	X
0	1	0	A – 1	A[7..0]	11111111	0	S	1	X
0	1	1	A + 1	A[7..0]	00000000	1	S	1	X
1	0	0	Ca2 B	00000000	/B[7..0]	1	S	1	X
1	0	1	A or B	A[7..0]	B[7..0]	X	OR	0	0
1	1	0	A and B	A[7..0]	B[7..0]	X	AND	0	1
1	1	1	Transferir A	A[7..0]	11111111	X	AND	0	1

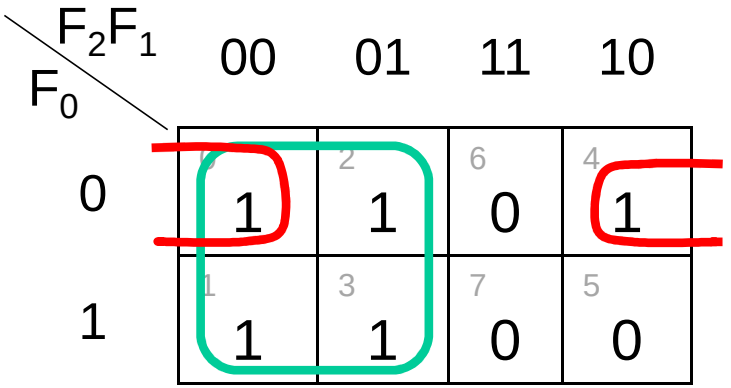
Disseny de les funcions de **control de sortida (SA i SL) i FCin**

F2	F1	F0	FCin	SA	SL
0	0	0	0	1	X
0	0	1	1	1	X
0	1	0	0	1	X
0	1	1	1	1	X
1	0	0	1	1	X
1	0	1	X	0	0
1	1	0	X	0	1
1	1	1	X	0	1

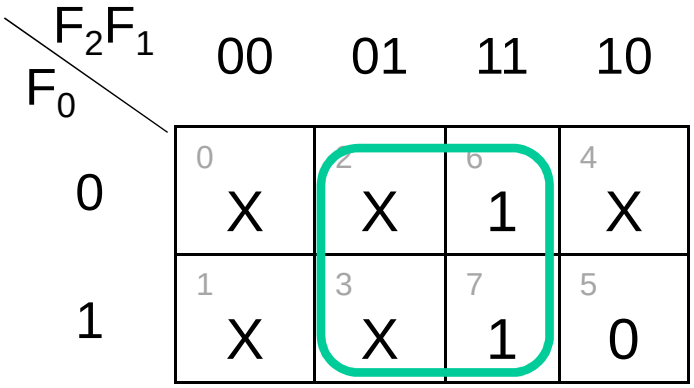




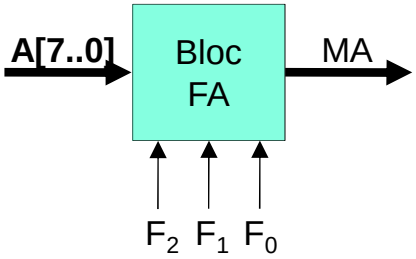
$FC_{in}=F_2+F_0$



$SA=/F_1/F_0+/F_2$



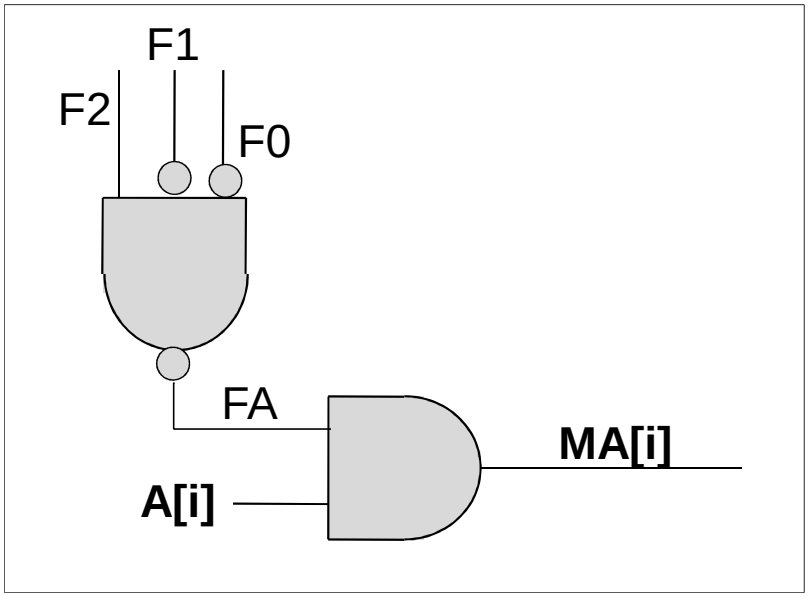
$SL=F_1$



F2	F1	F0	MA[7..0]	FA
0	0	0	A[7..0]	1
0	0	1	A[7..0]	1
0	1	0	A[7..0]	1
0	1	1	A[7..0]	1
1	0	0	00000000	0
1	0	1	A[7..0]	1
1	1	0	A[7..0]	1
1	1	1	A[7..0]	1

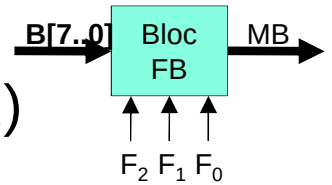
Per implementar el **bloc FA** cal que fem que MA sigui igual a A excepte en el cas 100, llavors la MA valdrà 00...0. Això es pot aconseguir amb una porta AND de 2 entrades a cadascun dels bits A_i,

Quan F2=1, F1=0 i F0=0, el senyal FA=0, per tant cadascun dels bits MA[i]=0



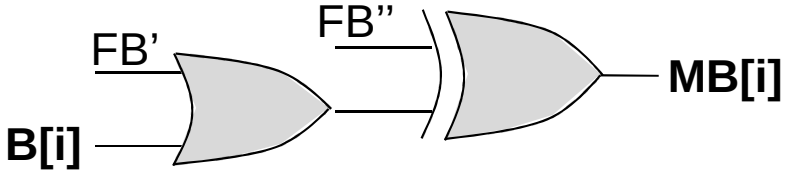
Les condicions pel **bloc FB** són més complexes, ja que ens cal:

- Poder complementar o deixar passar (realitzable amb portes XOR)
- Posar tot a 1 (realitzable amb portes OR)
- Posar tot a 0 (realitzable amb AND o NOR, o complementant l'anterior)

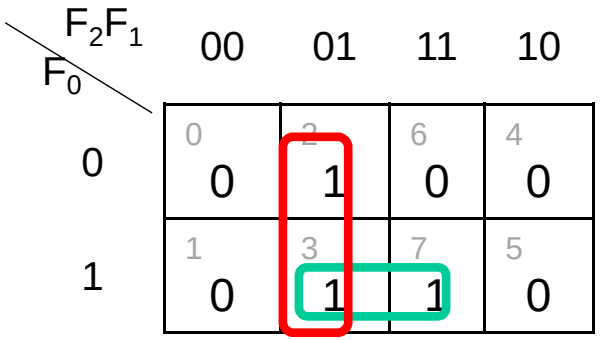


Definirem 2 blocs:

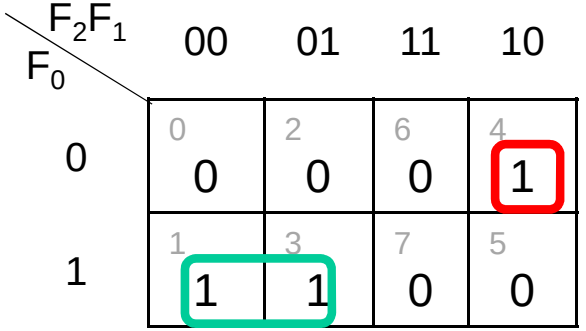
- FB' (control de porta OR) i
- FB'' (segona entrada de la porta XOR)



F2	F1	F0	MB[7..0]	FB'	FB''
0	0	0	B[7..0]	0	0
0	0	1	/B[7..0]	0	1
0	1	0	11111111	1	0
0	1	1	00000000	1	1
1	0	0	/B[7..0]	0	1
1	0	1	B[7..0]	0	0
1	1	0	B[7..0]	0	0
1	1	1	11111111	1	0



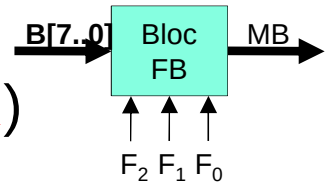
$FB' = /F_2F_1 + F_1F_0$



$FB'' = /F_2F_0 + F_2/F_1/F_0$

Les condicions pel **bloc FB** són més complexes, ja que ens cal:

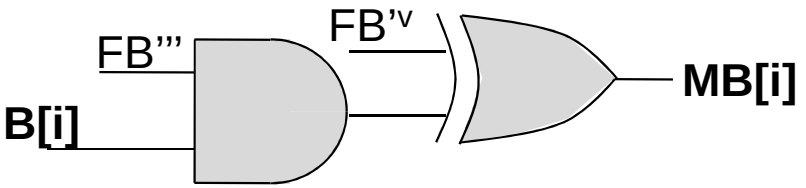
- Poder complementar o deixar passar (realitzable amb portes XOR)
- Posar tot a 1 (realitzable amb portes OR)
- Posar tot a 0 (realitzable amb AND o NOR, o complementant l'anterior)



Definirem 2 blocs:

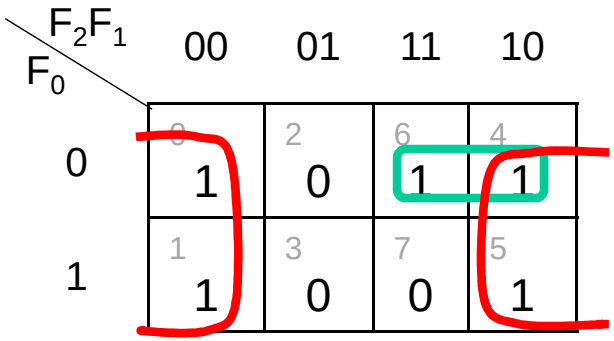
- FB''' (control de porta AND) i
- FB'^{v} (segona entrada de la porta XOR)

Si ho fem amb ANDs enlloc d'ORs

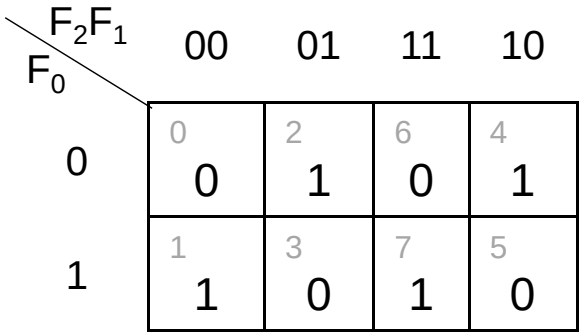


F2	F1	F0	MB[7..0]
0	0	0	B[7..0]
0	0	1	/B[7..0]
0	1	0	11111111
0	1	1	00000000
1	0	0	/B[7..0]
1	0	1	B[7..0]
1	1	0	B[7..0]
1	1	1	11111111

FB'''	FB'^{v}
1	0
1	1
0	1
0	0
1	1
1	0
1	0
0	1

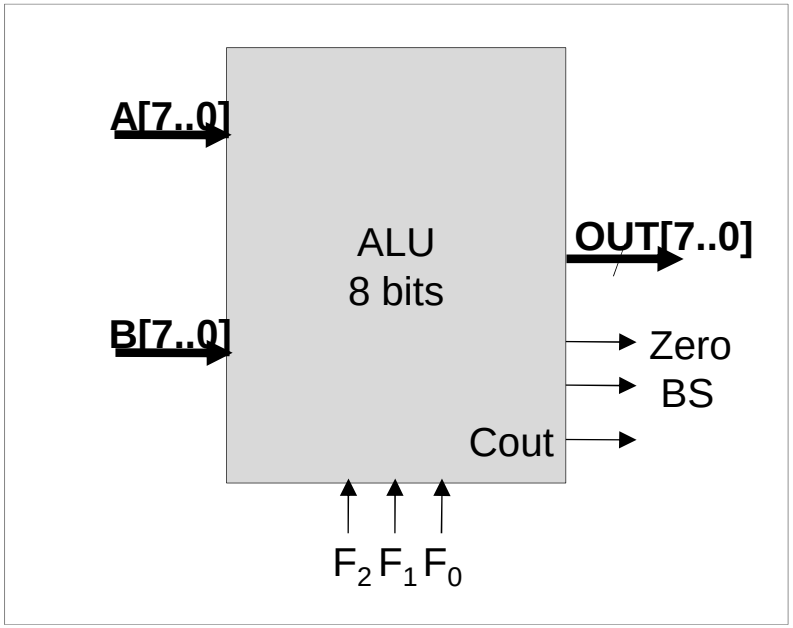


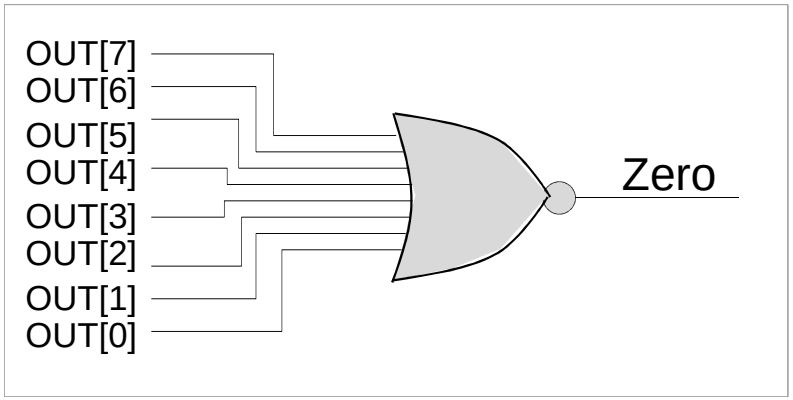
FB'''=$F_2/F_0+/F_1$



FB'^{v}=$F_2\oplus(F_1\oplus F_0)$

Sortides Zero i BS



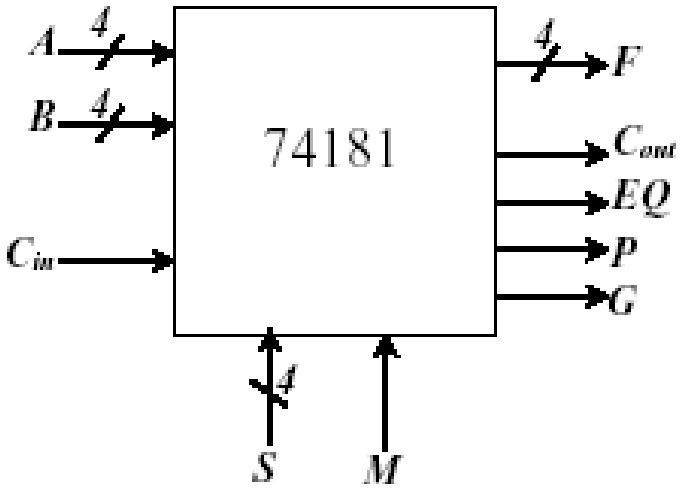


- La Sortida **Zero** és una sortida que dóna un 1 quan tots els bits del resultat (OUT[7..0]) són 0, per tant és una funció NOR dels 8 bits
- La **BS**, és directament el bit de signe del resultat
- Aquestes dues sortides són útils quan es vol implementar una **comparació** amb l'ALU. Les comparacions es fan amb una **resta** i la sortida Zero indica la **igualtat** i quan A i B són diferents el BS indica **quin és el més gran**

Exemple d'una ALU comercial

A.L.U. 74181

Pin-out



Taula de funcions de l'ALU 74181

$S_3S_2S_1S_0$	$M=1$	$M=0$
0000	$F=A'$	$w=v(A)+v(C_{in})$
0001	$F=(A+B)'$	$w=v(A+B)+v(C_{in})$
0010	$F=A'B$	$w=v(A+B')+v(C_{in})$
0011	$F=(0,0,0,0)$	$w=v(C_{in})$
0100	$F=(AB)'$	$w=v(A)+v(AB')+v(C_{in})$
0101	$F=B'$	$w=v(A+B)+v(AB')+v(C_{in})$
0110	$F=XOR(A,B)$	$w=v(A)-v(B)-v(C'_{in})$
0111	$F=AB'$	$w=v(AB')-v(C'_{in})$
1000	$F=A'+B$	$w=v(A)+v(AB)+v(C_{in})$
1001	$F=XNOR(A,B)$	$w=v(A)+v(B)+v(C_{in})$
1010	$F=B$	$w=v(A+B')+v(AB)+v(C_{in})$
1011	$F=AB$	$w=v(AB)-v(C'_{in})$
1100	$F=(1,1,1,1)$	$w=v(A)+v(A)+v(C'_{in})$
1101	$F=A+B'$	$w=v(A+B)+v(A)+v(C_{in})$
1110	$F=A+B$	$w=v(A+B')+v(A)+v(C_{in})$
1111	$F=A$	$w=v(A)-v(C'_{in})$

Variables de control

M mode d'operació: lògica (M=1) o aritmètica (M=0)

S selecció de funció (16 de lògiques i 16 d'aritmètiques)

Sortides aritmètiques

$EQ = F_3F_2F_1F_0$ detecció de sortida zero

$C_{out} = 0$ (sortida < 16) o 1 (sortida ≥ 16)

P,G termes propagat i generat de la suma aritmètica