Arhitectura sistemelor de calcul

- Prelegerea 8 -

ALU (Unitatea aritmetică şi logică)

Ruxandra F. Olimid

Facultatea de Matematică şi Informatică

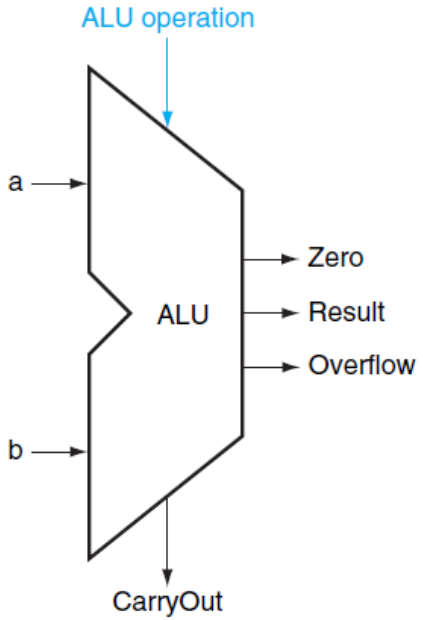
Universitatea din Bucureşti

Cuprins

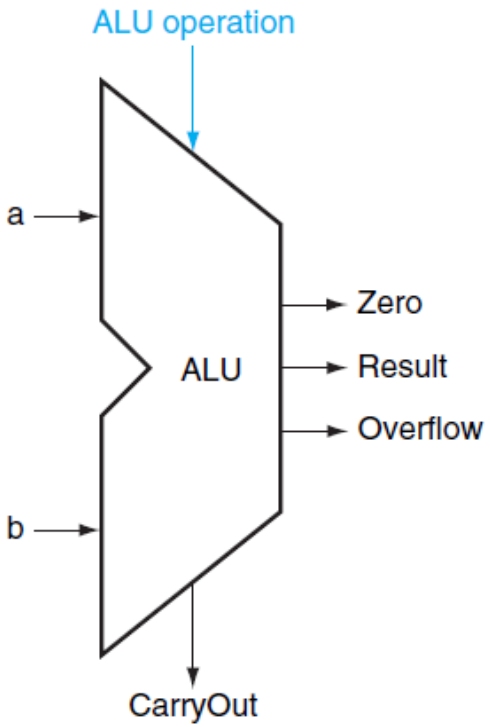
ALU (Unitatea aritmetică şi logică)

- *Scop:* realizarea calculelor aritmetice şi logice
- *Abreviere:* ALU (Arithmetic and Logic Unit)
- *Mod de funcţionare*: este un cicuit pur combinaţional (nu necesită cicluri) cu rolul de a efectua operaţii aritmetice (adunare, scădere,…) şi operaţiile logice (AND, OR,…)
- *Reprezentare schematică:*

[Considerăm ALU pe 32 biţi (MIPS32)]



ALU (Unitatea aritmetică şi logică)



[COD]

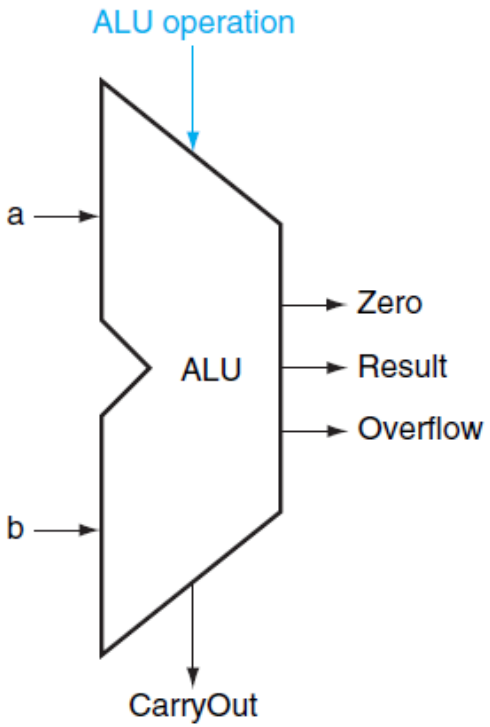
Intrare:

- *ALU operation:* indică operaţia care se efectuează (4 biţi)

ALU operation	Operaţie
0000	and
0001	or
0010	add
0110	substract
1100	nor
0111	set on less than

- a, b: operanzi (32 biţi)

ALU (Unitatea aritmetică şi logică)



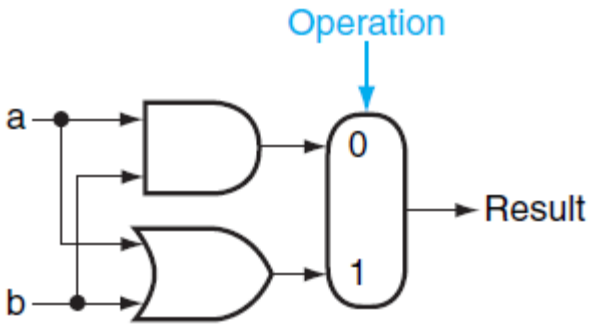
Ieşire:

- *Zero:* indică dacă se obţine 0 (1 bit)
- *Result*: rezultatul operaţiei (32 biţi)
- *Overflow*: indică depăşire (1 bit)
- *CarryOut*: bitul de transport (1 bit)

[COD]

ALU simplificat pe 1 bit

- Considerăm ALU pe 1 bit care efectuează doar **AND** şi **OR**
- Acesta se construieşte uşor cu ajutorul a 2 porţi şi un EMUX care selectează operaţia

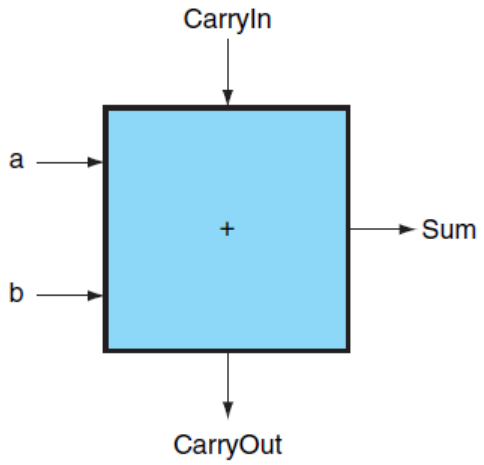


Operation	Operaţie
0	and
1	or

[COD]

ALU simplificat pe 1 bit

- Adăugăm operaţia de **adunare** pe 1 bit cu bit de transport [info]
- Acesteia îi corespunde următoarea reprezentare schematică şi următorul tabel de adevăr:

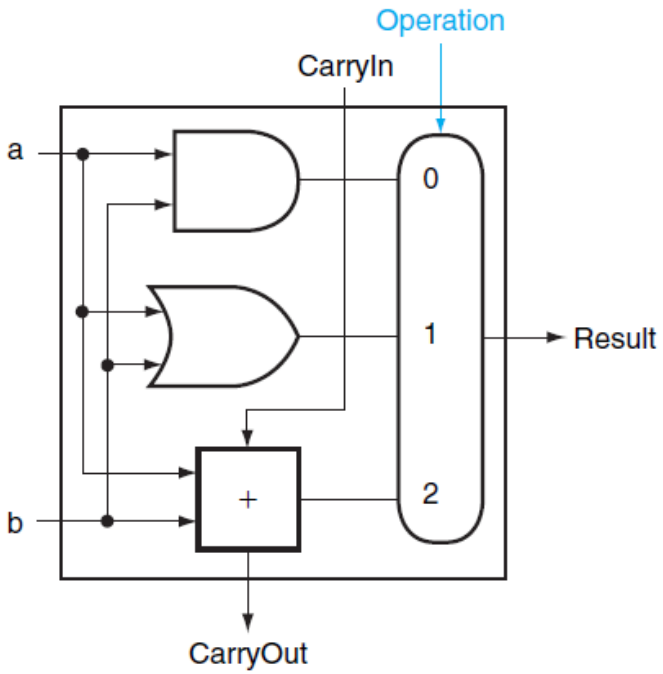


[COD]

a	b	CarryIn	Sum	CarryOut
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

ALU simplificat pe 1 bit

- Construcţia necesită MUX cu 2 intrări care selectează operaţia
- Atunci ALU pe 1 bit care efectuează **AND**, **OR** şi **ADD** devine:

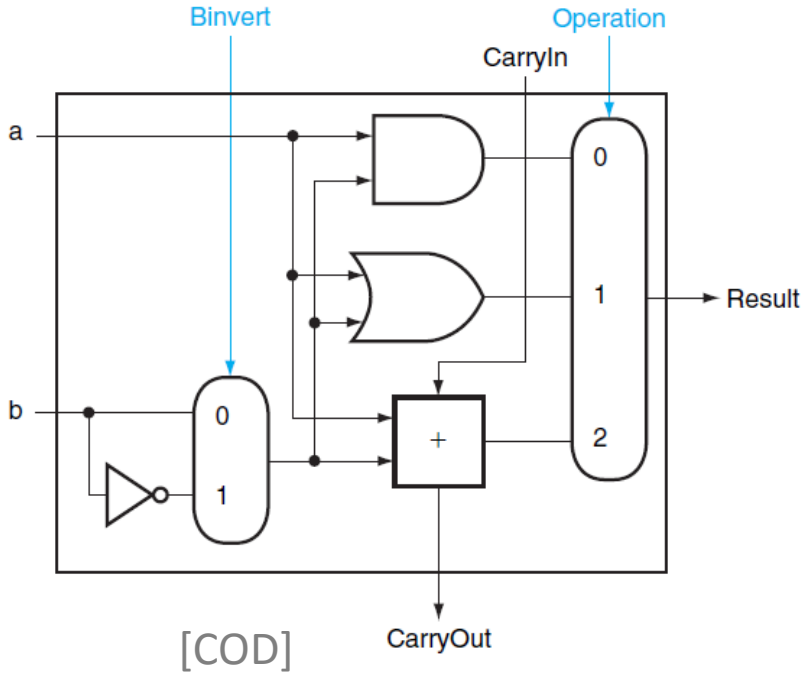


Operation	Operaţie
00	and
01	or
10	add

[COD]

ALU simplificat pe 1 bit

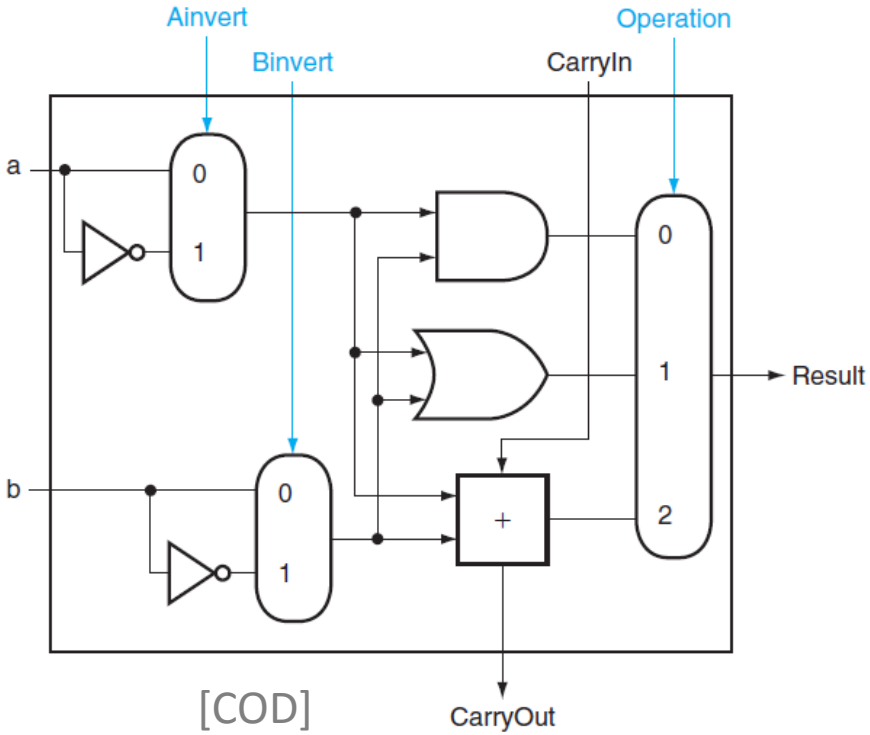
- Adăugăm operaţia de **scădere** (aceasta se reduce la adunare dacă scăzătorul este reprezentat în complement faţă de 2, i.e. negat şi adunat cu 1)
- Construcţia necesită în plus un EMUX al cărui bit de selecţie să fie setat la 1 pentru scădere (preia $\bar{b}$), iar *CarryIn* trebuie setat la 1
- Atunci ALU pe 1 bit care efectuează **AND**, **OR, ADD** şi **SUB** devine:



Binvert	Operation	Operaţie
0	00	and
0	01	or
0	10	add
1	10	substract

ALU simplificat pe 1 bit

- Adăugăm operaţia **NOR** ($a\ NOR\ b$ se reduce la $\bar{a}\ AND\ \bar{b}$ din legile lui DeMorgan) [info]
- Construcţia are deja o poartă AND şi negarea intrării b, deci se mai adaugă negarea lui a şi un bit de selecţie *Ainvert*
- Atunci ALU pe 1 bit care efectuează **AND**, **OR, ADD, SUB** şi **NOR** devine:



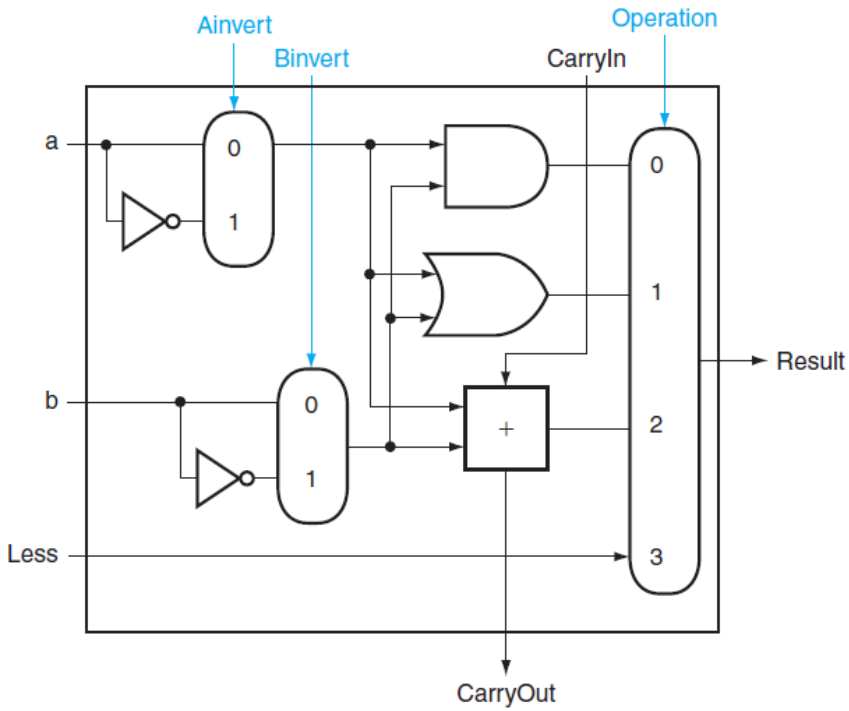
Ainvert	Binvert	Operation	Operaţie
0	0	00	and
0	0	01	or
0	0	10	add
0	1	10	substract
1	1	00	nor

ALU simplificat pe 1 bit

- Adăugăm operaţia **set on less than** ($a < b$ se reduce la $a - b < 0$)
- Pentru numere mai mari de 1 bit, **set on less than** va întoarce rezultatul 0 peste tot cu excepţia ultimului bit (*lsb*) care va fi :
  - ✓ 1 dacă $a < b$
  - ✓ 0 altfel
- Observaţi că *lsb* se reduce la bitul de semn al valorii $a - b$
  - ✓ 1 dacă $a - b$ este negativ, echivalent cu $a < b$
  - ✓ 0 altfel
- Construcţia necesită o nouă intrare în EMUX care va genera rezultatul comparaţiei; fie *Less* această intrare

ALU simplificat pe 1 bit

- Construcţia devine:

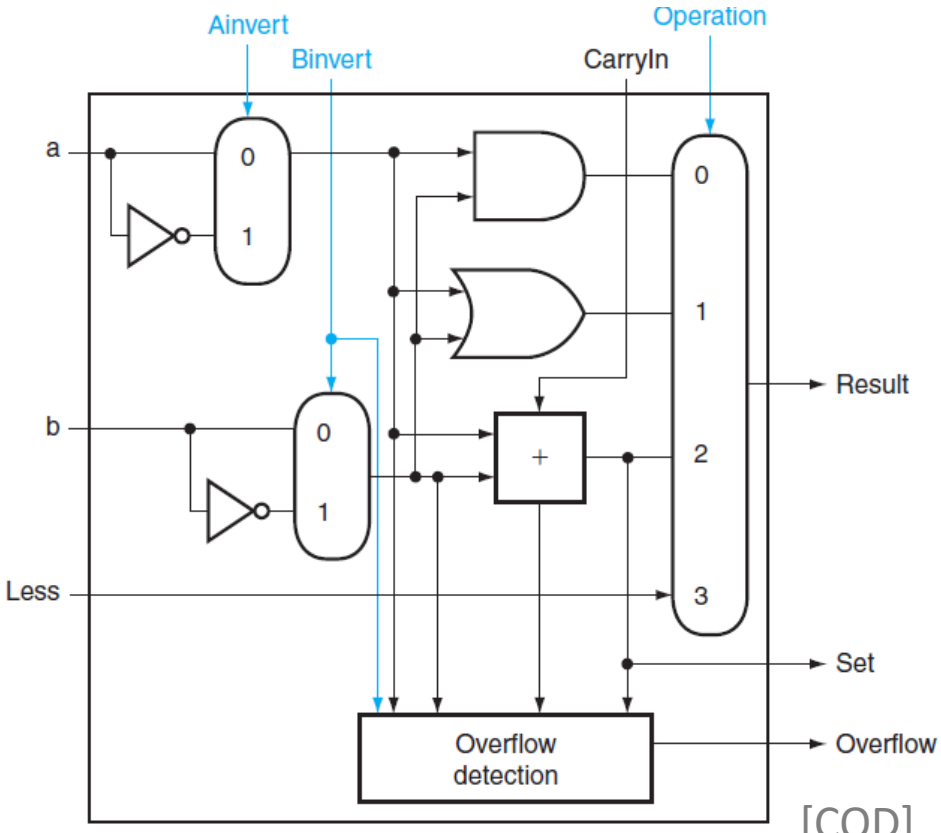


[COD]

Ainvert	Binvert	Operation	Operaţie
0	0	00	and
0	0	01	or
0	0	10	add
0	1	10	substract
1	1	00	nor
0	1	11	set on less than

ALU simplificat pe 1 bit

- ✓ Pentru *Less = 0* rezultă primii 31 de biţi ai comparaţiei, care sunt mereu 0
- ✓ Pentru *Less = bitul de semn* rezultă *lsb*, deci avem nevoie de o ieşire suplimentară *Set*: ieşirea din Full Adder



[COD]

Ainvert	Binvert	Operation	Operaţie
0	0	00	and
0	0	01	or
0	0	10	add
0	1	10	substract
1	1	00	nor
0	1	11	set on less than

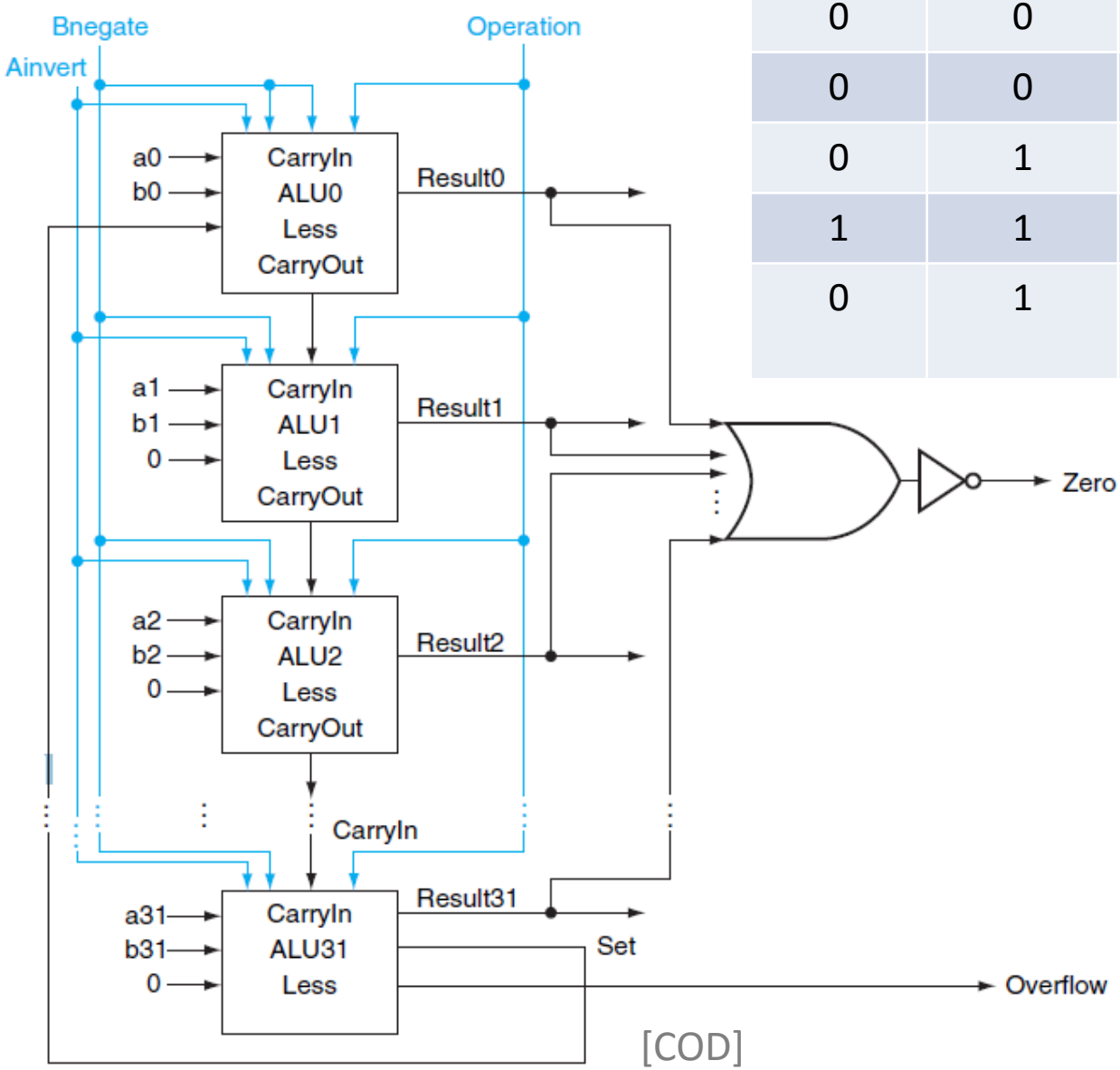
[Am introdus şi detecţie la depăşire (overflow), fără să explicităm]

ALU pe 32 biţi

- Extensia serială conduce la ALU pe 32 de biţi
- Notatăm $a_{31} \dots a_0$ reprezentarea binară a lui a, respectiv $b_{31} \dots b_0$ reprezentarea binară a lui b
- Observaţii:
  - ✓ Se propagă transportul: CarryIn pentru ALU i este CarryOut al ALU (i-1)
  - ✓ BInvert şi CarryIn au aceeaşi valoare (la scădere 1, altfel 0), deci semnalele se pot comasa într-unul singur, numit *Bnegate*
  - ✓ Set din ALU31 devine Less în ALU1 ca să se realizeze corect comparaţia
- Adăugăm în plus un flag *Zero*, care va fi activ când $a = b$ (necesar pentru instrucţiunile de branch; acesta se obţine imediat prin **OR** pe toate ieşirile $a - b$

ALU pe 32 biţi

Ainvert	Bnegate	Operation	Operaţie
0	0	00	and
0	0	01	or
0	0	10	add
0	1	10	substract
1	1	00	nor
0	1	11	set on less than



Carry Lookahead

- Dezavantajul construcţiei seriale este imediat: circuitul este lent (pentru o adunare, fiecare FA trebuie să aştepte rezultatul FA precedent)

- Pentru a înlătura acest neajuns se poate folosi *Carry Lookahead*, un circuit care calculează transportul mult mai rapid, nefiind pur serial

Carry Lookahead

- Din tabela de adevăr rezultă imediat că putem exprima:

$$CarryOut = ab \oplus (a \oplus b)CarryIn$$

- Cum *CarryOut* devine *CarryIn* pentru următorul sumator, se obţine:

$$c_{i+1} = a_i b_i \oplus (a_i \oplus b_i)c_i = G_i \oplus P_i c_i$$

- Notaţii:
  - ✓ c_i= transportul în runda i
  - ✓ $G_i = a_i b_i$= *generate*
  - ✓ $P_i = a_i \oplus b_i$= *propagate*

a	b	CarryIn	Sum	CarryOut
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

Carry Lookahead

- *Carry Lookahead* primeşte la intrare valorile binare a şi b şi generează biţii de transport

- Problema care apare este complexitatea circuitului:

$$c_1 = G_0 \oplus P_0 c_0$$

$$c_2 = G_1 \oplus P_1 c_1 = G_1 \oplus P_1 (G_0 \oplus P_0 c_0) = G_1 \oplus P_1 G_0 \oplus P_1 P_0 c_0$$

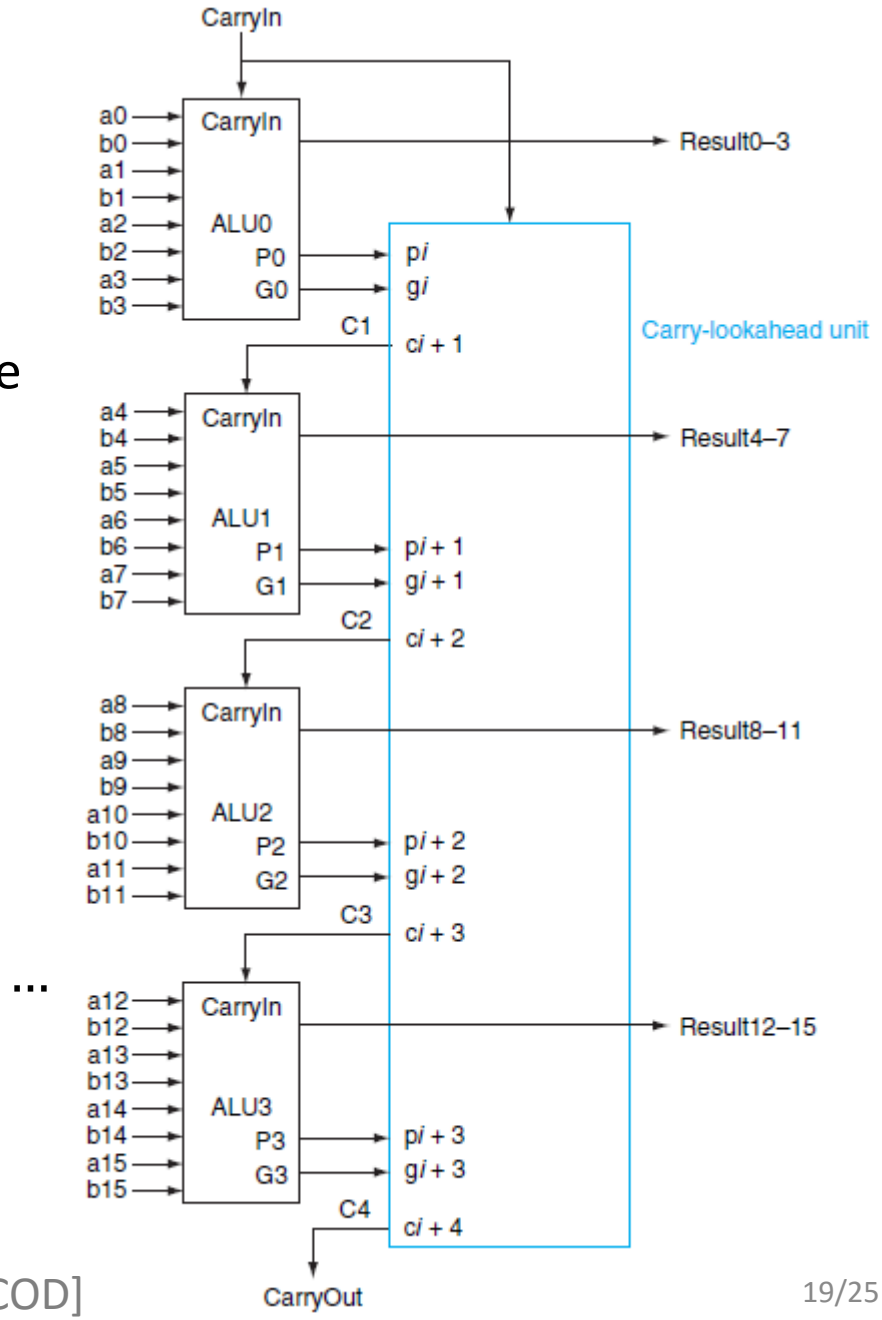
$$c_3 = G_2 \oplus P_2 c_2 = \cdots = G_2 \oplus P_2 G_1 \oplus P_2 P_1 G_0 \oplus P_2 P_1 P_0 c_0$$

...

- În consecinţă se alege un compromis între compunerea serială şi CarryLookahead pe un număr favorabil de biţi

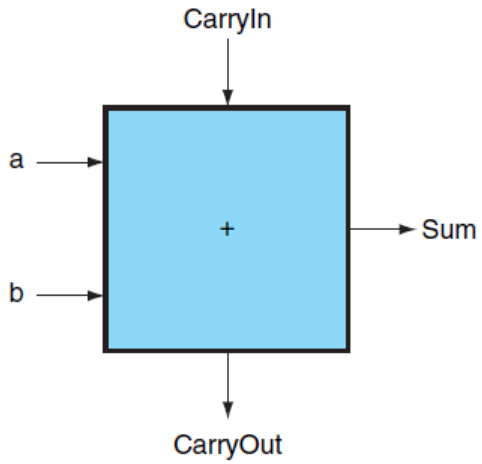
Carry Lookahead

- Un exemplu de astfel de sumator pe 16 biţi foloseşte care foloseşte valorile G_i , P_i calculate pe grupuri de câte 4 biţi:



Sumator pe 1 bit

- Plecăm de la tabelul de adevăr şi construim sumatorul complet (*Full Adder*)



[COD]

a	b	CarryIn	Sum	CarryOut
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

Sumator pe 1 bit

- Funcţiile canonice pentru cele 2 componente care formează ieşirea sunt:

$$Sum = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$$

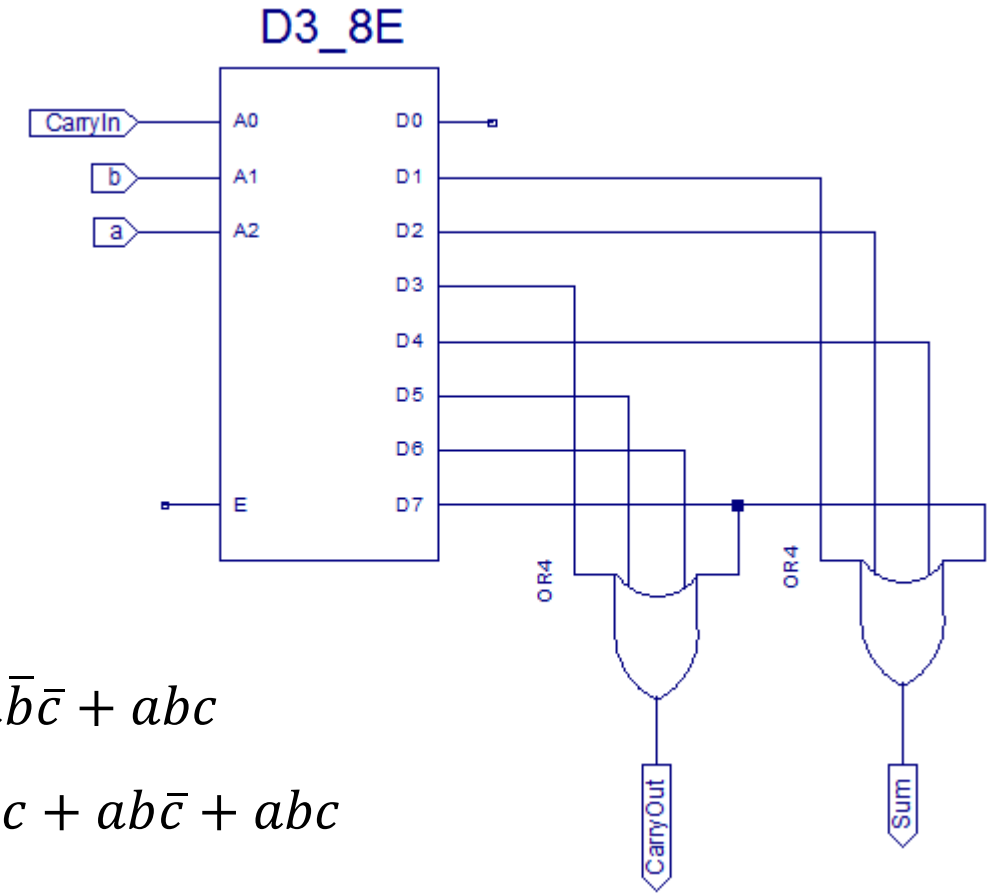
$$CarryOut = \bar{a}bc + a\bar{b}c + ab\bar{c} + abc$$

Notă: Pentru simplitate, am notat $CarryIn$ cu c

a	b	CarryIn	Sum	CarryOut
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

Sumator pe 1 bit

- Având formele canonice, se defineşte circuitul combinaţional:



$$Sum = \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}\bar{c} + abc$$

$$CarryOut = \bar{a}bc + a\bar{b}c + ab\bar{c} + abc$$

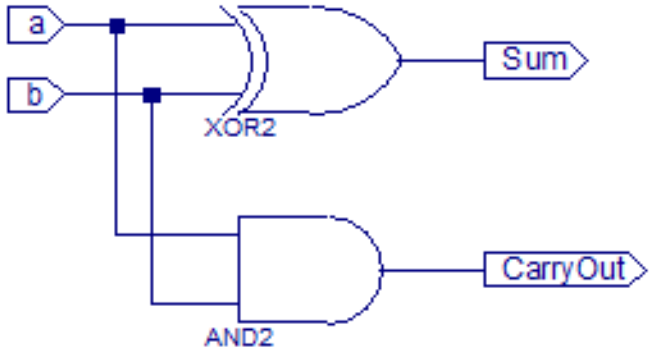
[Xilinx - ISE]

Sumator pe 1 bit

- Sumatorul complet (*FA=Full Adder*) accept acceptă transport la intrare, spre deosbire de un semi-sumator (*HA=Half Adder*)

- *Half Adder* poate fi realizat imediat (suma a doi biţi este realizată prin XOR şi transportul este 1 numai dacă ambii biţi sunt 1) astfel:

a	b	Sum	CarryOut
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



[Xilinx - ISE]

[inapoi]

Legile lui DeMorgan

- Permit exprimarea conjuncţiei şi a disjuncţiei una în funcţie de cealaltă prin negare

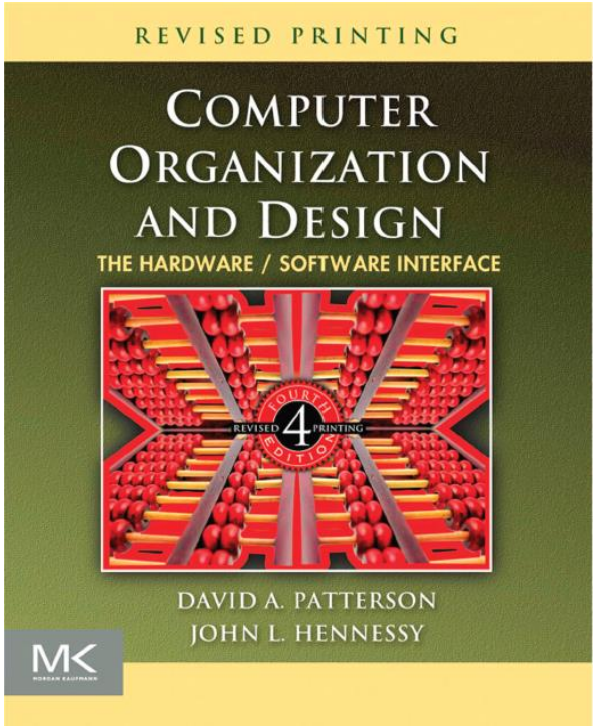
$$\overline{a+b} = \bar{a}\bar{b}$$

$$\overline{ab} = \bar{a} + \bar{b}$$

- Rezultă imediat că se poate renunţa la unul dintre operatorii AND sau OR pentru reprezentarea unei funcţii logice (în condiţiile în care se poate utiliza negarea).

[inapoi]

Referințe bibliografice



[AAT] A. Atanasiu, Arhitectura calculatorului

[COD] D. Patterson and J. Hennessy, Computer Organisation and Design

Schemele [Xilinx - ISE] au fost realizate folosind

http://www.xilinx.com/tools/projnav.htm