Computer Science & Engineering Department

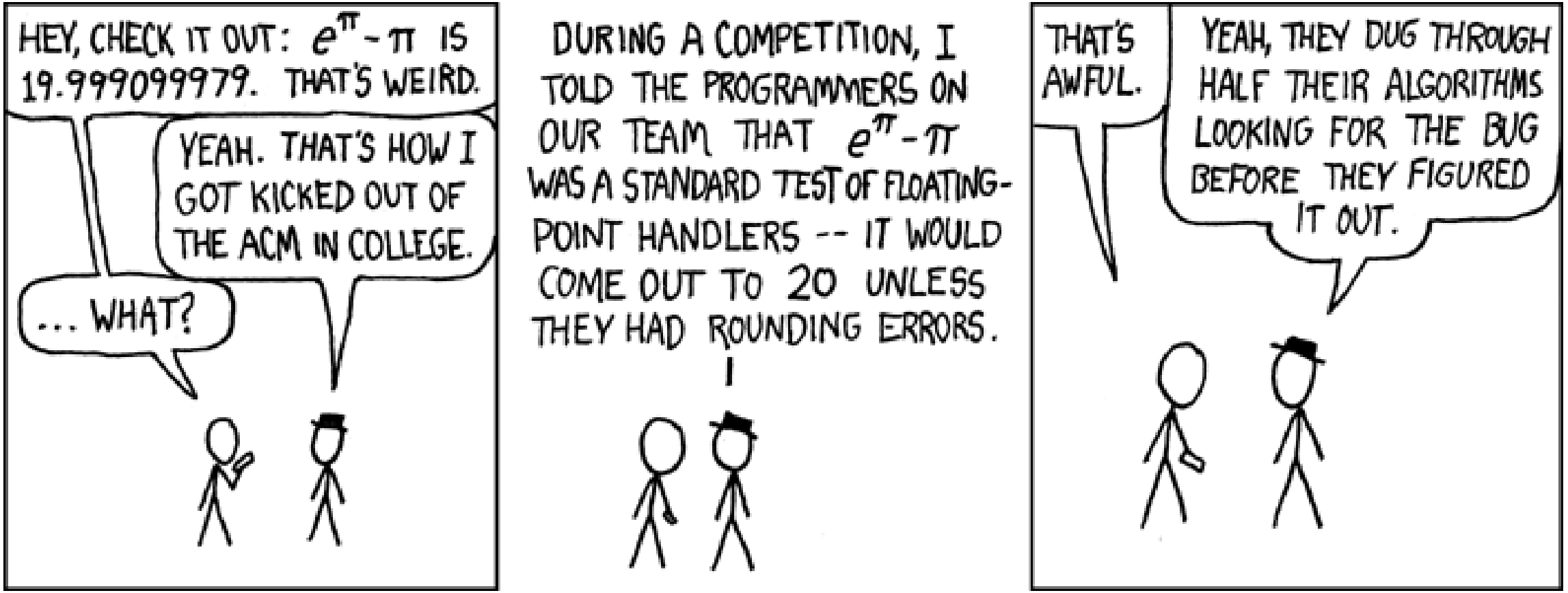
POLITEHNICA BUCUREȘTI 1818

Structura și Organizarea Calculatoarelor

– Cursul 3 –

Operații aritmetice

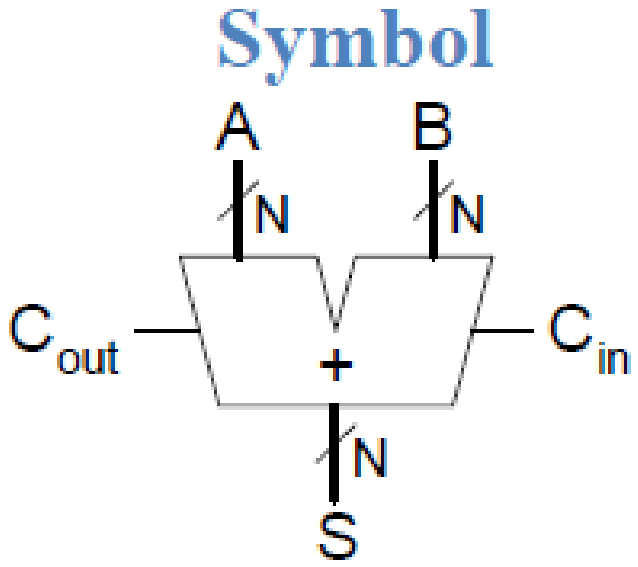
Facultatea de Automatică și Calculatoare
Universitatea Politehnica București



http://xkcd.com/217/

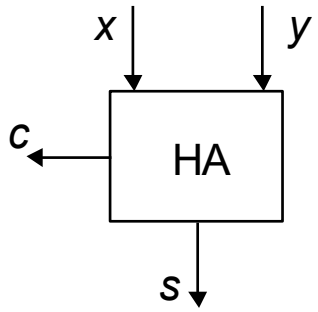
Sumatorul binar

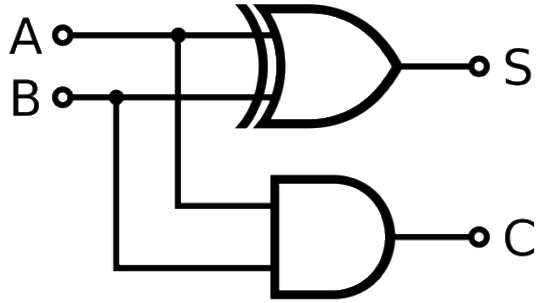
- Tipuri de sumatoare cu propagare de carry
  - Ripple-carry (lent)
  - Carry look-ahead (rapid)
  - Prefix (și mai rapid)
- Sumatoarele carry look-ahead și cu prefix sunt mai rapide pentru numere pe mai mulți biți dar necesită mai mult hardware



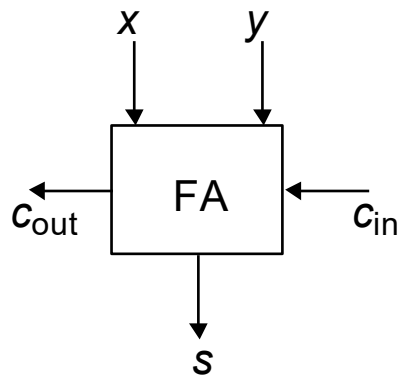
Binary half-adder (HA) & full-adder (FA)

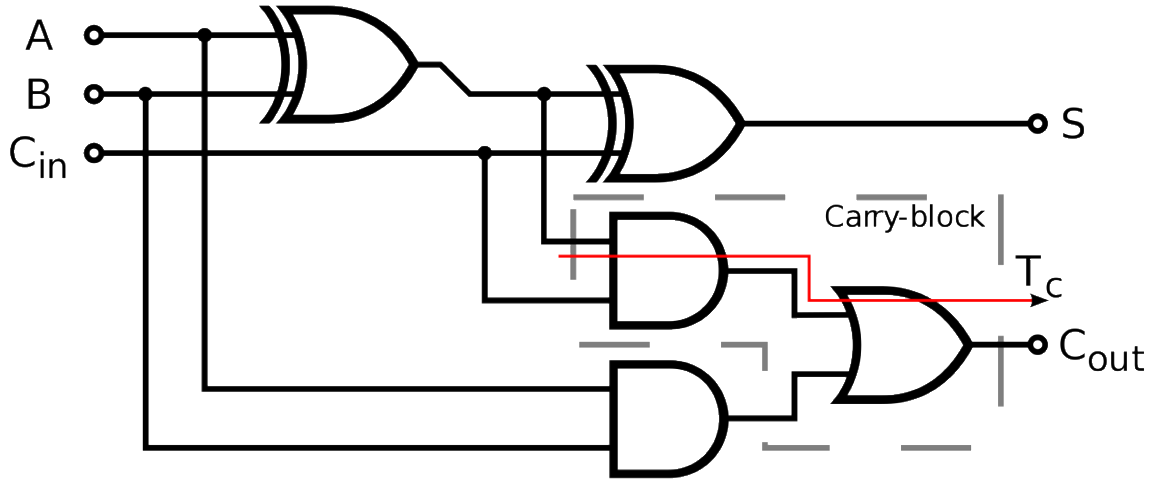
Inputs		Outputs	
x	*y*	*c*	*s*
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



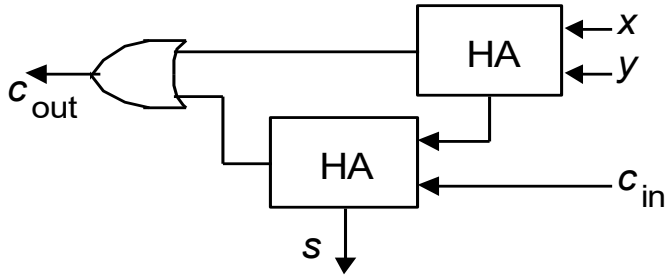


Inputs			Outputs	
x	*y*	c_{in}	c_{out}	*s*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

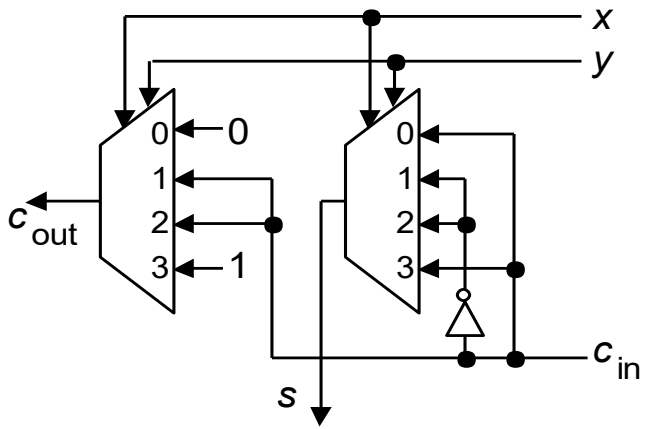




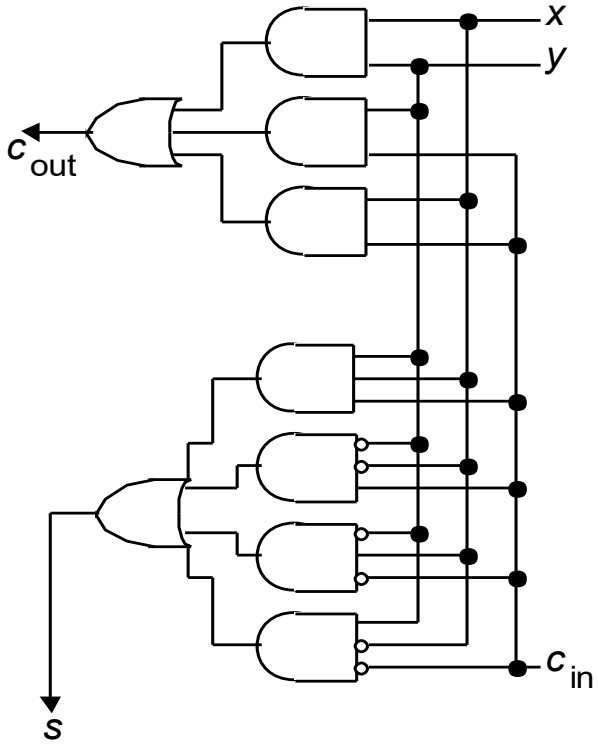
Implementarea unui sumator complet



(a) FA built of two HAs



(b) CMOS mux-based FA

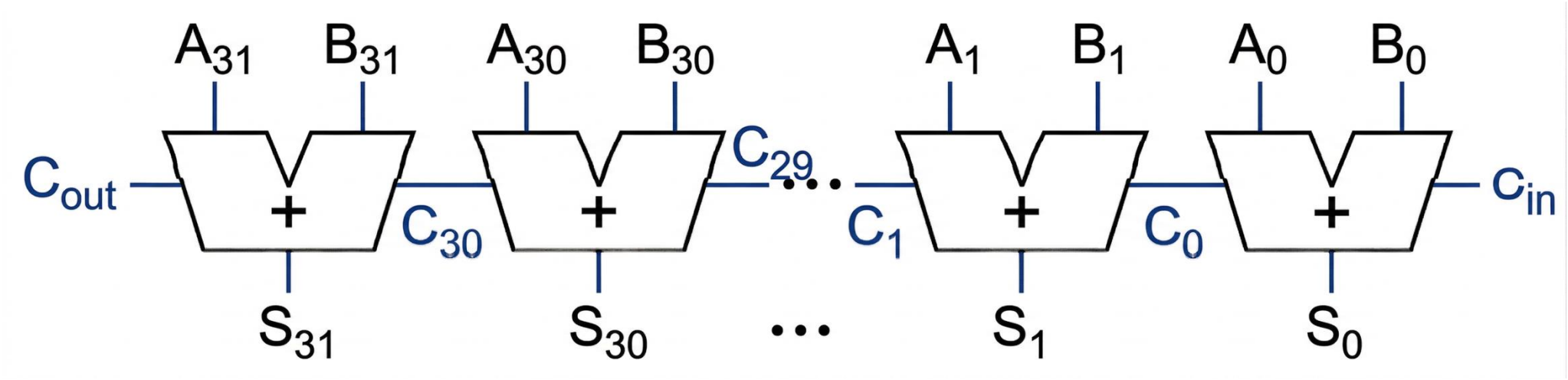


(c) Two-level AND-OR FA

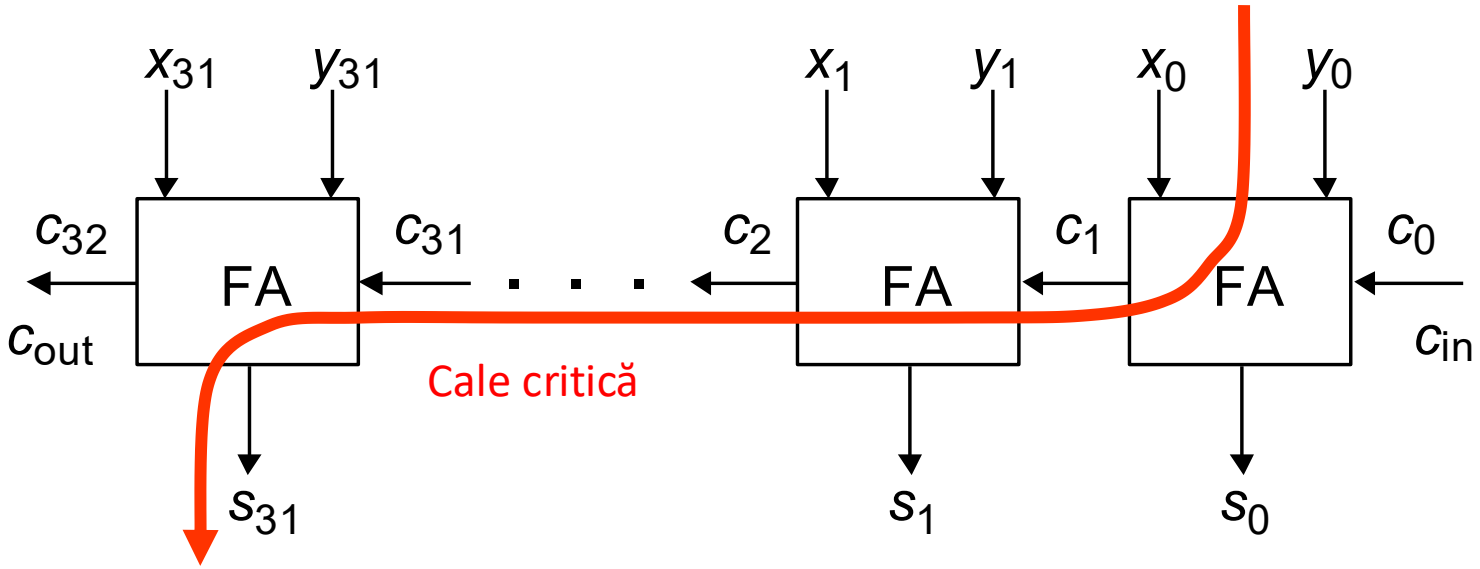
Sumator complet implementat folosind două sumatoare tip half-adder, cu ajutorul a două multiplexoare cu 4 intrări sau un circuit cu porți pe două niveluri.

Sumator Ripple-Carry

- Înlănțuim mai multe sumatoare de un bit
- Semnalul de carry se propagă prin întregul lanț
- Dezavantaj: **lent**



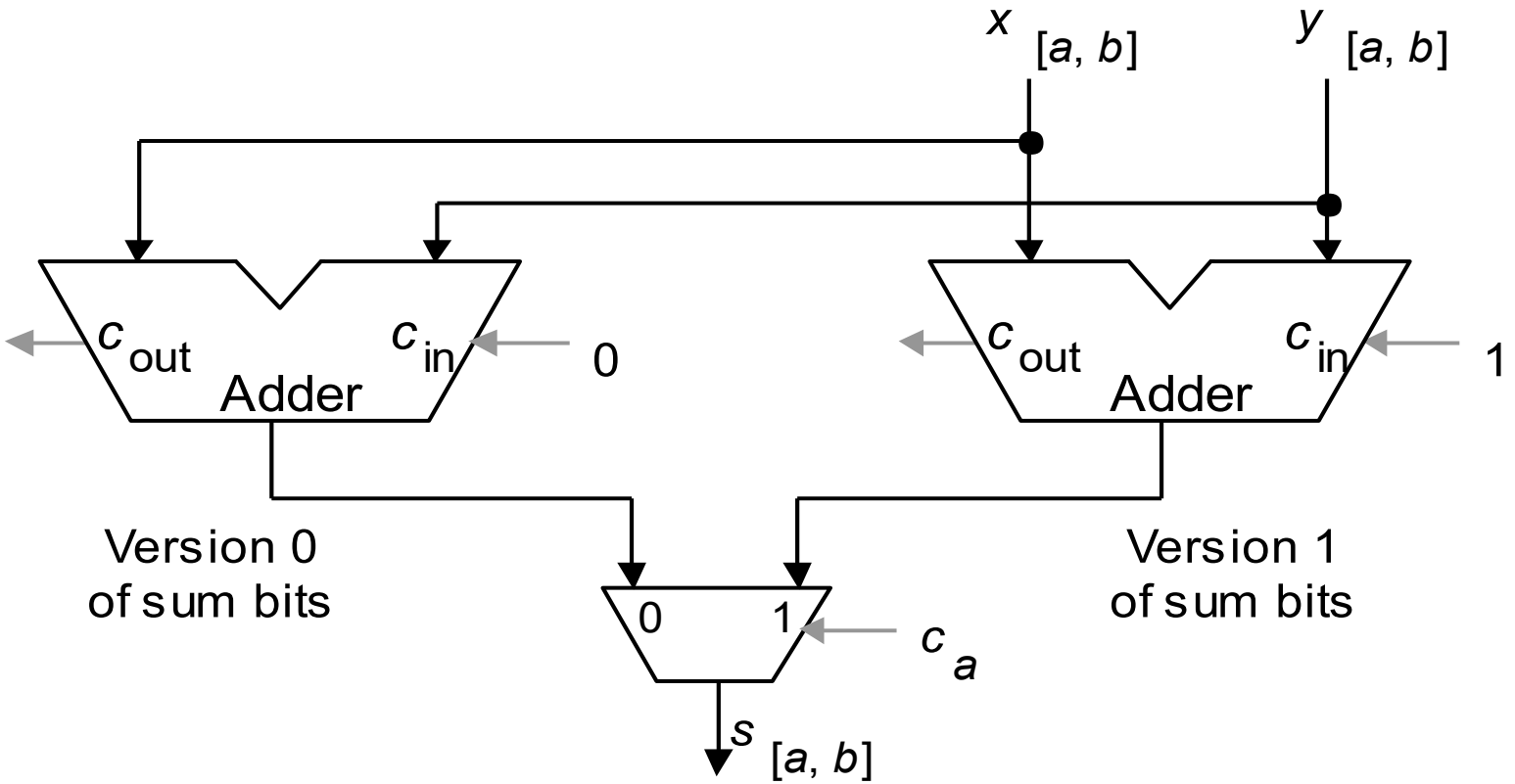
Sumator în riplu (ripple carry): foarte simplu dar lent



Sumator ripple-carry pentru numere de 32 de biți.

Sumator Carry Select

Presupune dublarea sumatorului inițial și introduce o întârziere suplimentară printr-un singur MUX



Principiul de funcționare al sumatorului Carry-select

Rangurile inferioare ale lui a, (0 la $a-1$) sunt sumate normal

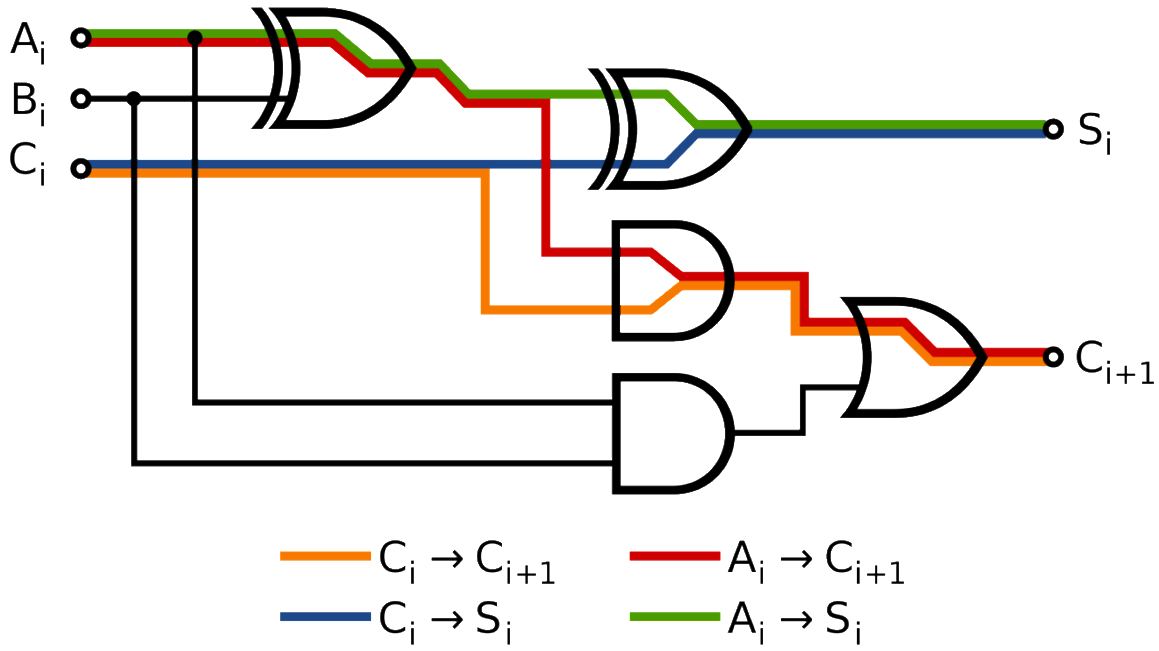
Întârzierea prin ripple-carry

Full-adder propagation delay:

- AND & OR au întârzieri similare (1-gate delay)
- XOR are întârziere dublă (2-gate delay) pentru că este facută din o combinație de porți AND și OR

Un sumator complet are următoarele întârzieri de propagare:

- De la A_i sau B_i la C_{i+1}: 4 gate-delays (XOR → AND → OR)
- De la A_i sau B_i to S_i: 4 gate-delays (XOR → XOR)
- De la C_i la C_{i+1}: 2 gate-delays (AND → OR)
- De la C_i la S_i: 2 gate-delays (XOR)

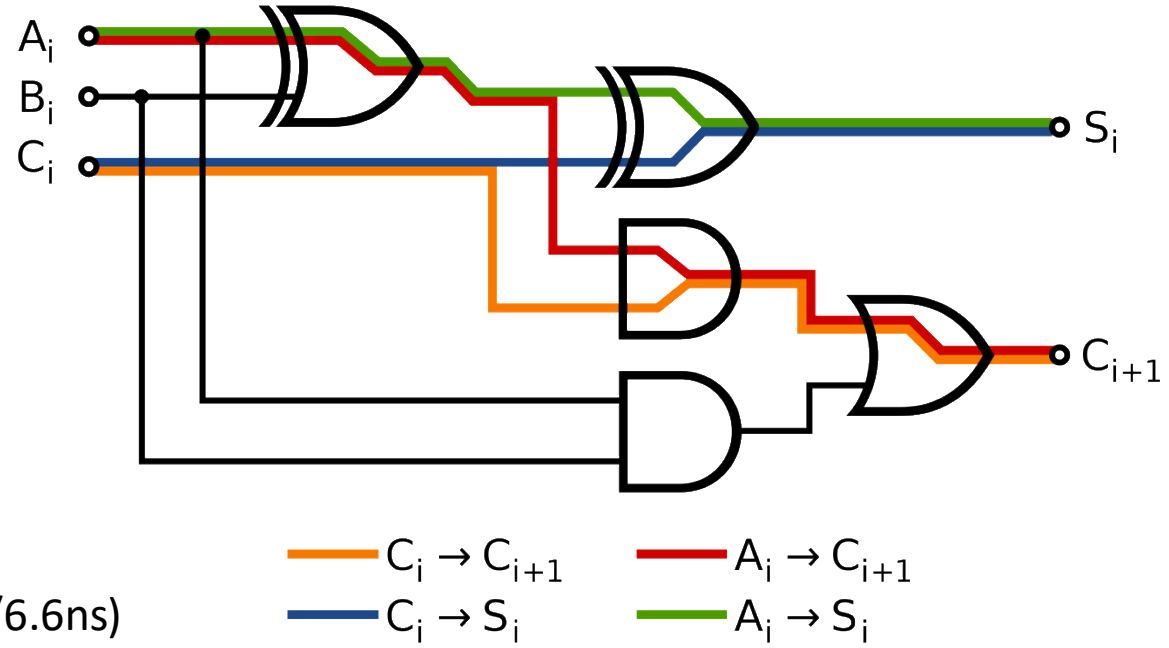


Întârzierea prin ripple-carry

Pentru că ieșirea de carry de la o etapă se leagă direct la intrarea de carry a următoarei etape, întârzierea totală pentru un sumator ripple-carry:

- 4 gate-delays din generarea primului semnal carry ($A_0/B_0 \rightarrow C_1$)
- 2 gate-delays pentru fiecare etapă intermediară ($C_i \rightarrow C_{i+1}$)
- 2 gate-delays la ultima etapă pentru producerea sumei și a ultimului carry-out ($C_{n-1} \rightarrow C_n$ și S_{n-1})

$$t_{ripple} = 4 + 2(n-2) + 2 = 2n + 2$$



Exemplu:

- Sumator pe 32 biți
- 2 gate-delay = 100ps
- Total delay = (2*32+2)*100ps = 6.6ns
- Dacă o adunare se execută într-un singur ciclu de ceas, frecvența procesorului trebuie limitată la maxim 151MHz (1/6.6ns)

Analiza întârzierii: Generare & Propagare

Decuplarea calculului transportului de rangul anterior:

- Generate (G_i): transport generat intern
  $G_i = A_i \cdot B_i$ (ambii biți sunt 1)
- Propagate (P_i): transportul trece mai departe
  $P_i = A_i + B_i$ (unul din cei doi biți este 1)
- Ecuația transportului:
  $C_{i+1} = A_i \cdot B_i + (A_i + B_i)C_i = G_i + P_i \cdot C_i$

A	B	Cin	Cout
0	0	0	0
0	0	1	0

Kill

0	1	0	0
1	0	0	0
0	1	1	1
1	0	1	1

Propagate

1	1	0	1
1	1	1	1

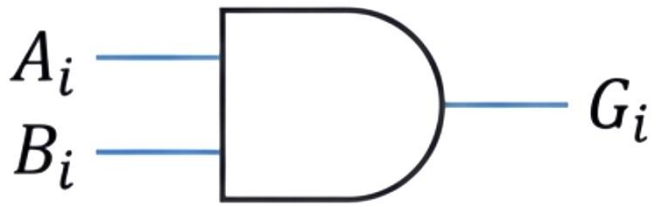
Generate

Generate & Propagate

Generare (G)

Funcția: Generează carry indiferent de C_{in}.

$$G_i = A_i \cdot B_i$$

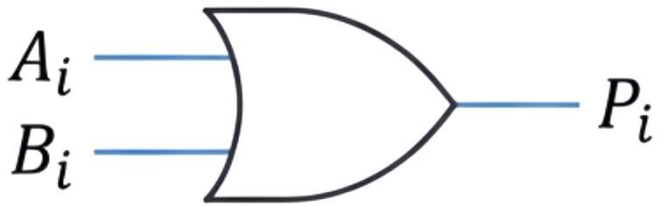


Activ când $A = 1$, $B = 1$.

Propagare (P)

Funcția: Transmite carry-ul de intrare la ieșire.

$$P_i = A_i + B_i$$



Activ când $A = 1$ sau $B = 1$.

Ecuația recursivă a transportului:

$$C_{i+1} = G_i + (P_i \cdot C_i)$$

A	B	P	G	State
0	0	0	0	Kill
0	1	1	0	Propagate
1	0	1	0	Propagate
1	1	1	1	Generate

Sumator Carry Look-Ahead

- Calculează carry out (C_{out}) pentru blocuri de k biți folosind semnalele de *generare* și *propagare*
- **Definiții:**

– Coloana *i* produce un carry out fie prin
  - *generarea* unui carry out
  - *propagarea* unui carry in la ieșirea de carry out

– Semnalele Generate (G_i) și Propagate (P_i) petru fiecare coloană:
  - Coloana *i* va genera un carry out dacă A_i *ȘI* B_i sunt ambele 1.

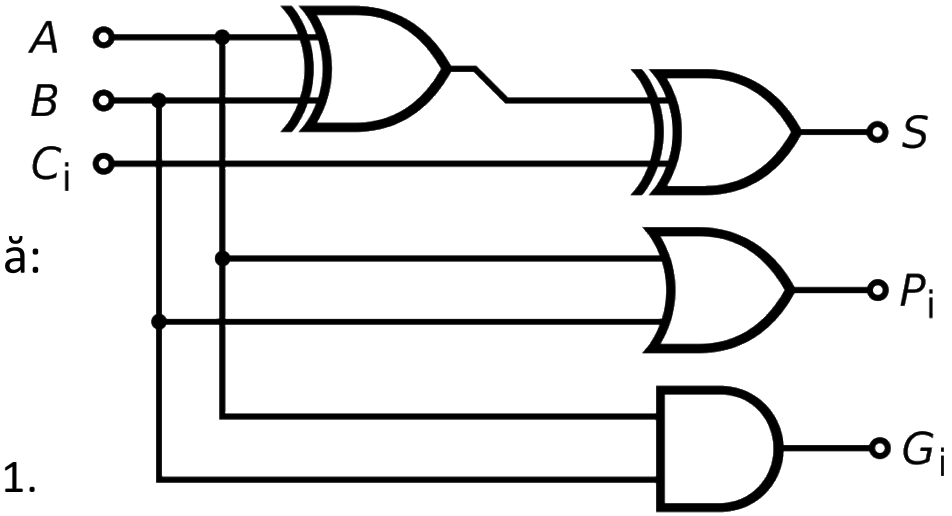
$$G_i = A_i B_i$$

  - Coloana *i* va propaga un carry in la carry out dacă A_i *SAU* B_i sunt 1.

$$P_i = A_i + B_i$$

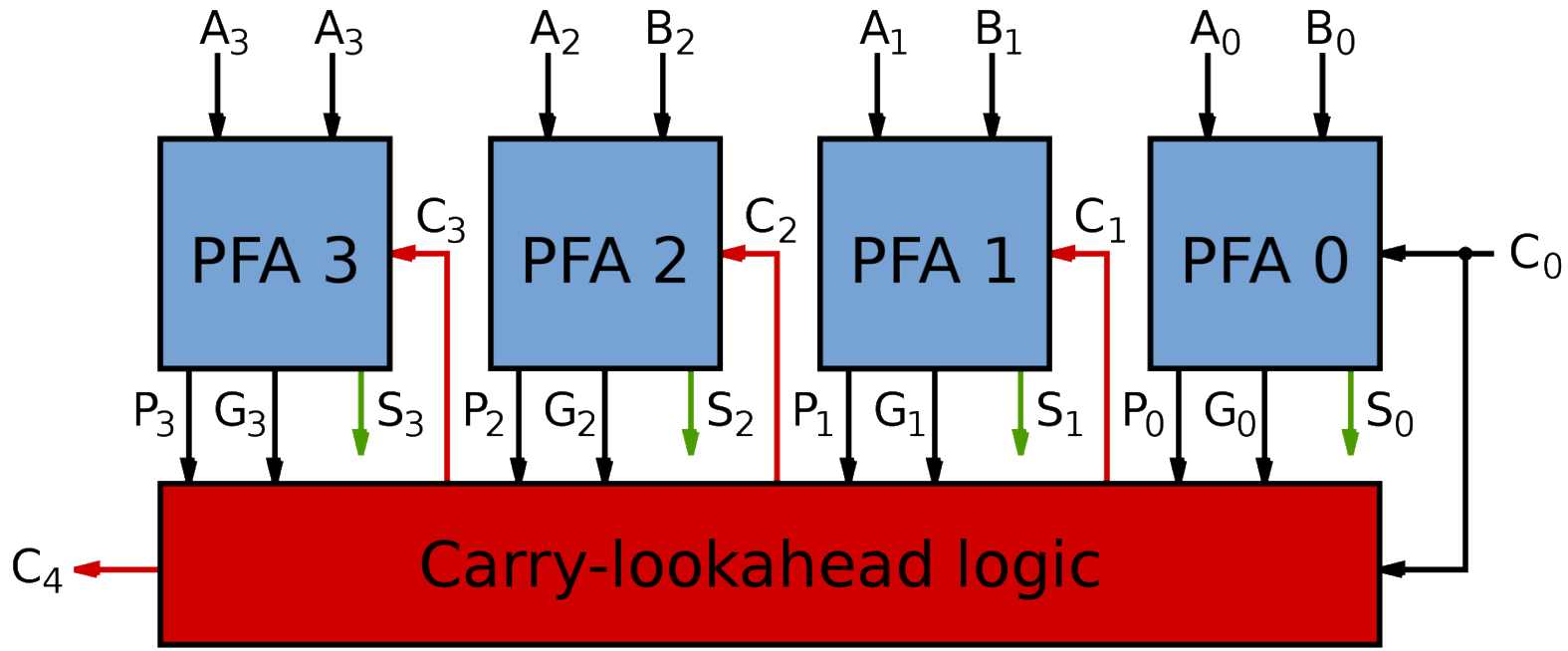
  - Semnalul carry out al coloanei *i* (C_i) este:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}$$



Adunarea Carry Look-Ahead

- **Pas 1:** Calculează G_i și P_i pentru toate coloanele
- **Pas 2:** Calculează G și P pentru blocuri de k-biți
- **Pas 3:** C_{in} se propagă prin fiecare bloc de k-biți de propagate/generate



Sumatorul Carry Look-Ahead

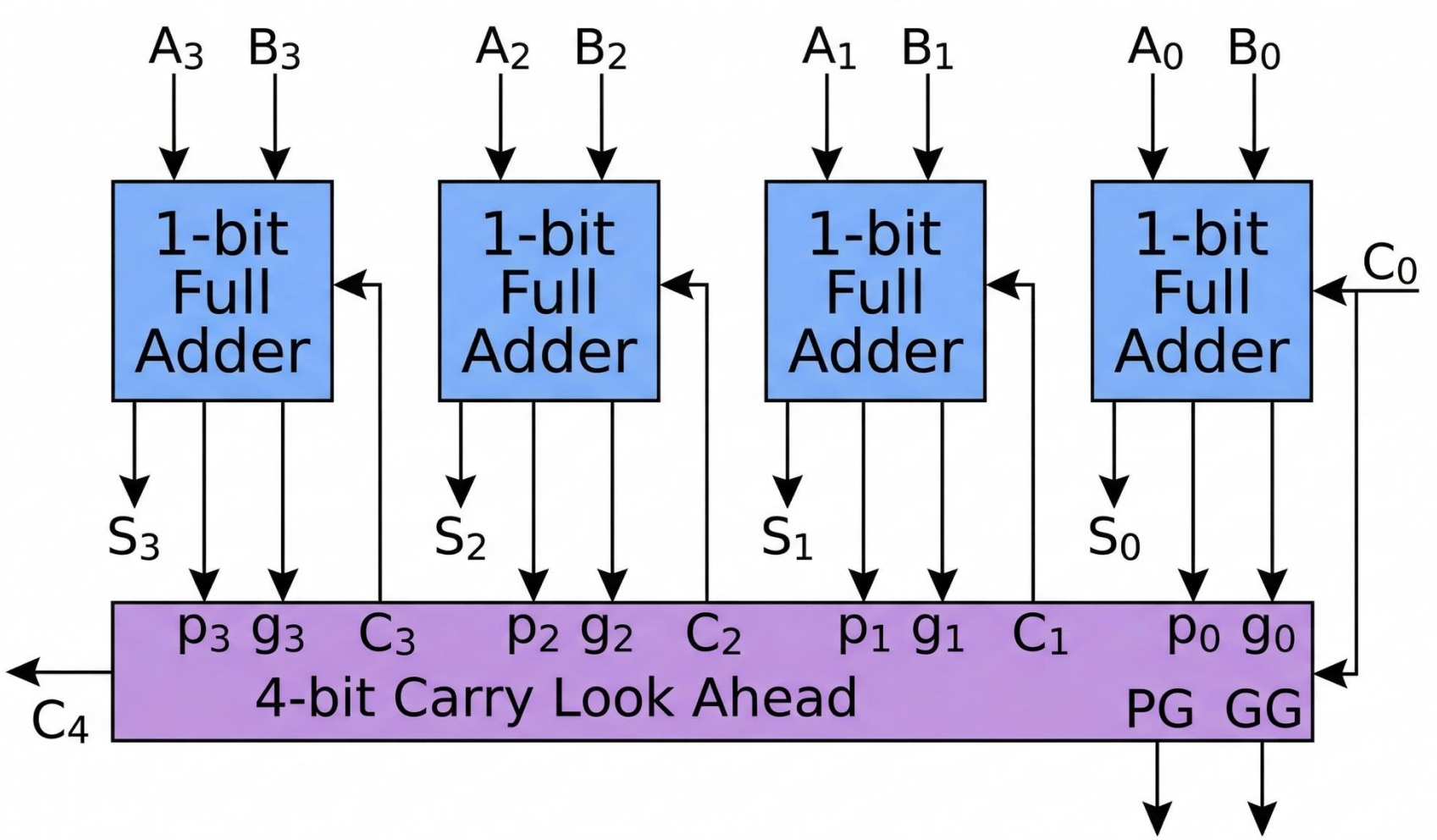
- **Exemplu:** blocuri de 4 biți ($G_{3:0}$ și $P_{3:0}$):

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)$$
$$P_{3:0} = P_3 P_2 P_1 P_0$$

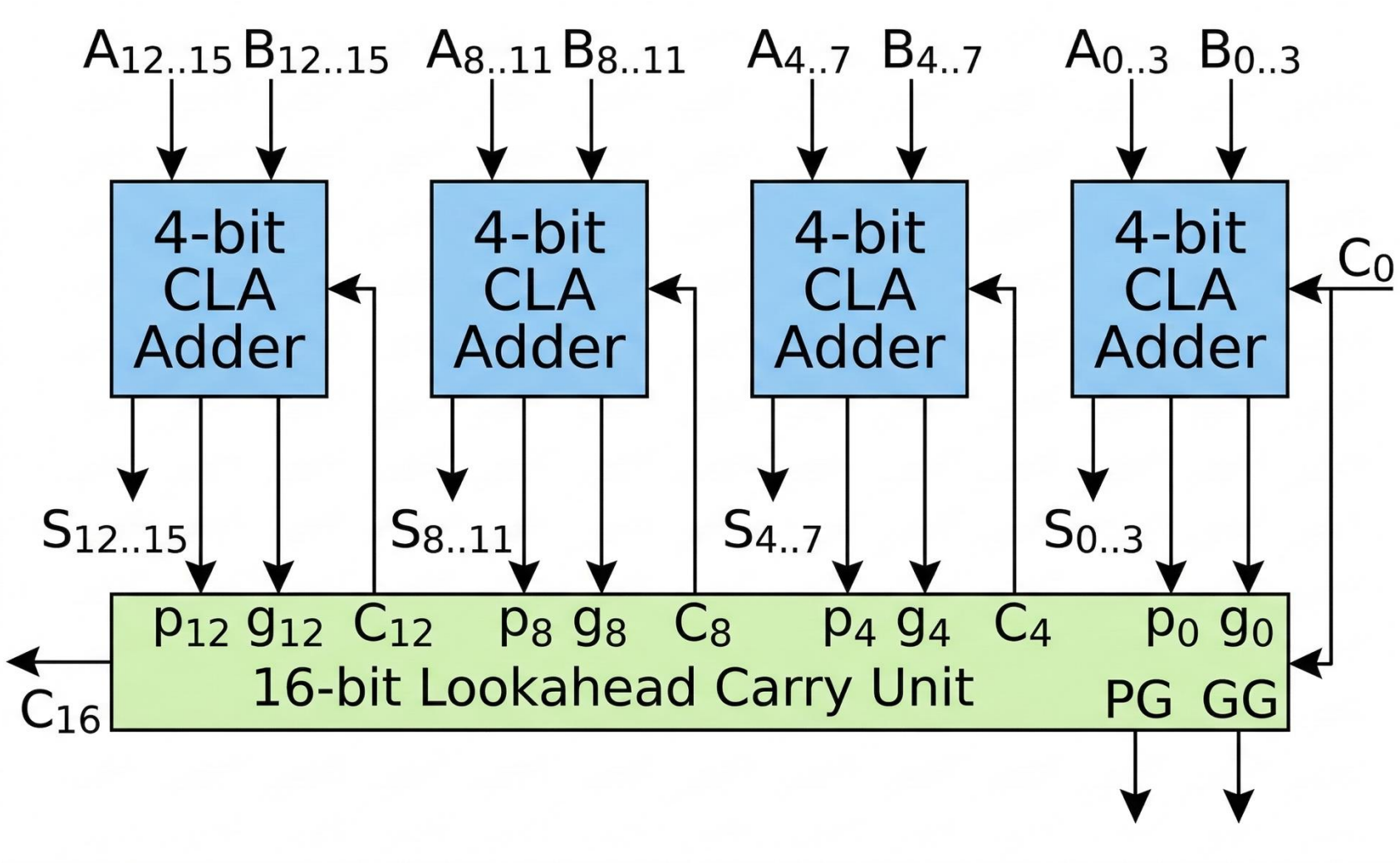
- **În general,**

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j)$$
$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$
$$C_i = G_{i:j} + P_{i:j} C_{i-1}$$

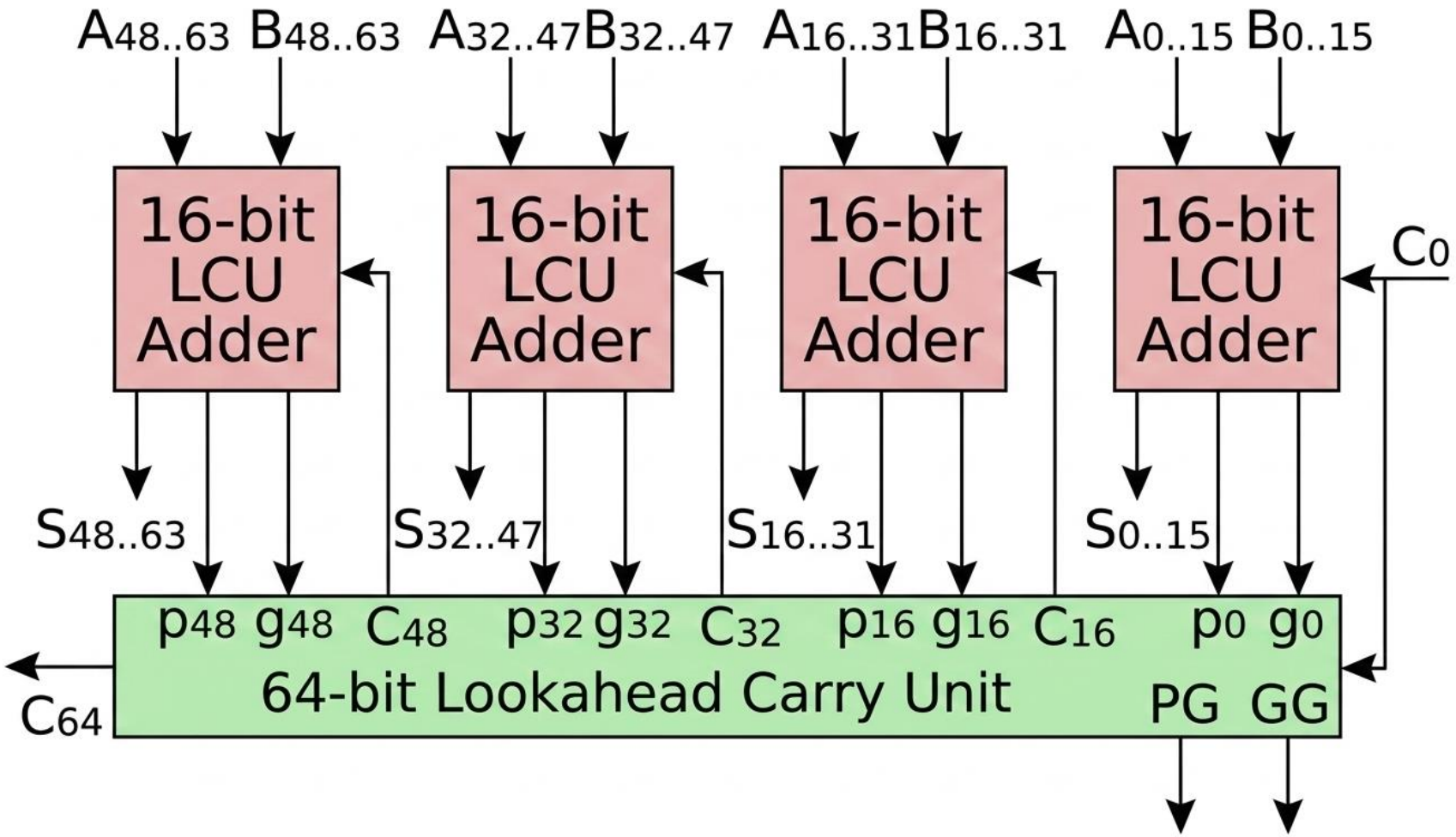
Carry Look-Ahead 4-biți



Carry Look-Ahead 16-biți

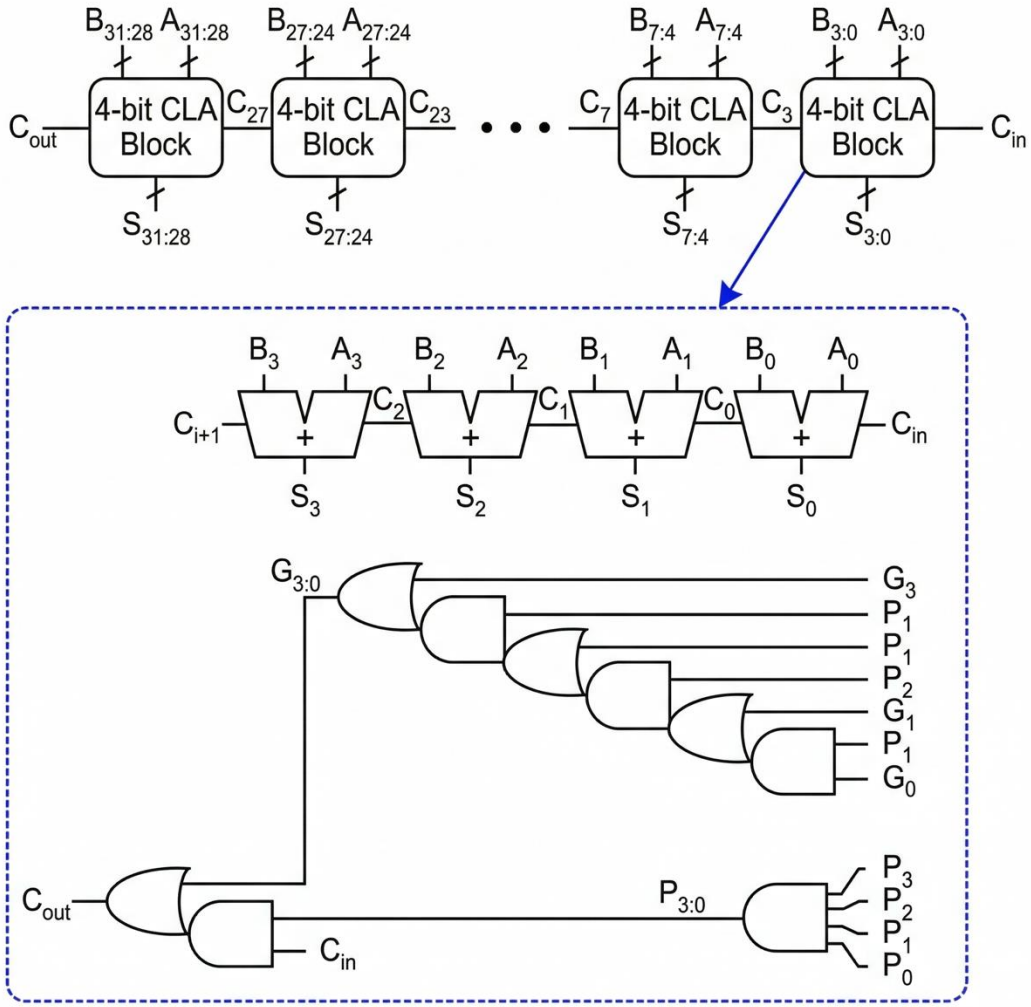


Carry Look-Ahead 64-biți



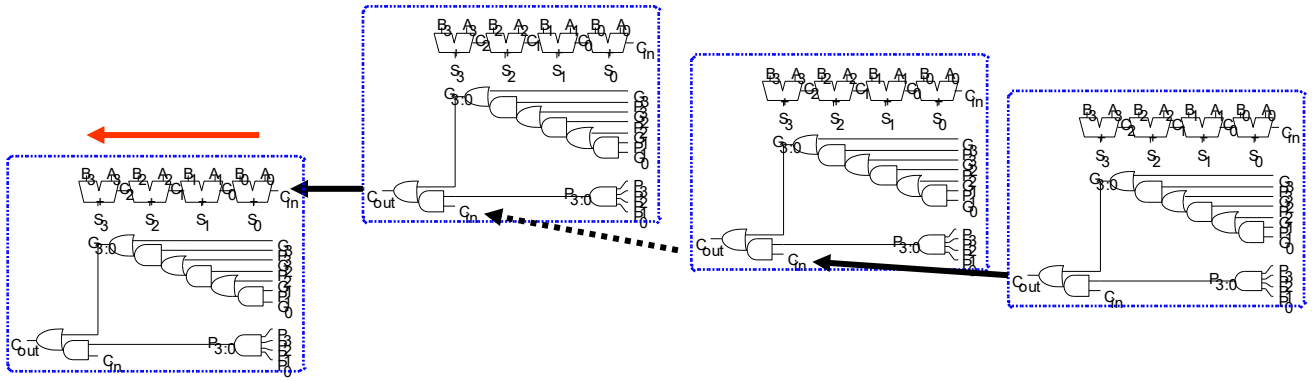
CLA pe 32-biți cu blocuri de 4-biți

- Grupăm rangurile binare în mai multe blocuri (blocuri de 4 ranguri binare în acest caz)
- Avantajul este că putem să calculăm carry-out mult mai repede decât dacă am trece carry-ul de la un rang la altul
- În esență, calculul carry este “scos” în afara sumatorului propriu-zis



Calculul sumei cu CLA

- **Pasul 1:** Calculează G_i și P_i pentru toate rangurile din un bloc
- **Pasul 2:** Calculează G și P pentru blocuri de k biți
- **Pasul 3:** C_{in} se propagă prin fiecare bloc logic de k biți propagate/generate (în același timp se calculează sumele)
- **Pasul 4:** Calculează suma pentru cel mai semnificativ bloc de k biți



Întârzierile printr-un sumator Carry Look-Ahead

Pentru un CLA pe N biți din blocuri de k biți:

$$t_{CLA} = t_{pg} + t_{pg\_block} + (N/k - 1)t_{AND\_OR} + kt_{FA}$$

– t_{pg} : delay pentru generarea P_i, G_i
– t_{pg_block} : delay pentru generarea tuturor $P_{i:j}$, $G_{i:j}$
– t_{AND_OR} : delay de la C_{in} la C_{out} a ultimei porți AND/OR din blocul de k-biți al CLA

Un sumator carry look-ahead de N biți este mult mai rapid decât un sumator ripple-carry dacă $N > 16$


KEEP
CALM AND
CARRY
LOOKAHEAD
imgflip.com

Sumator cu prefixe

- Calculează carry in (C_{i-1}) pentru fiecare coloană apoi calculează suma:

$$S_i = (A_i \text{ XOR } B_i) \text{ XOR } C_i$$

- Calculează *G* și *P* pentru blocuri de 1-, 2-, 4-, 8-biți, etc.

Până când toate G_i (carry in) sunt cunoscute

- $\log_2 N$ etape

Sumator cu prefixe

- Carry in este ori *generat* de o coloană sau *propagat* de la o coloană anterioară
- Coloana -1 ține C_{in}, așa că:

$$G_{-1} = C_{in},\ P_{-1} = X \text{ (nu ne pasă de propagate -1)}$$

- Carry in al coloanei *i* = carry out al coloanei *i-1*:

$$C_{i-1} = G_{i-1:-1}$$

$G_{i-1:-1}$: semnal generate de la coloana *i*-1 la -1

- Ecuația sumei:

$$S_i = (A_i \text{ XOR } B_i) \text{ XOR } G_{i-1:-1}$$

- **Scop: Calculează rapid** $G_{0:-1}$, $G_{1:-1}$, $G_{2:-1}$, $G_{3:-1}$, $G_{4:-1}$, $G_{5:-1}$, ...numite ***prefixe***

(*$G_{0:-1} = C_0$, $G_{1:-1} = C_1$, $G_{2:-1} = C_2$, $G_{3:-1} = C_3$* etc.)

Sumator cu prefixe

- Semnalele Generate și Propagate pentru un bloc de la biții *i*:*j*:

$$G_{i:j} = G_{i:k} + P_{i:k}\, G_{k-1:j}$$

$$P_{i:j} = P_{i:k} P_{k-1:j}$$

- În limba română:

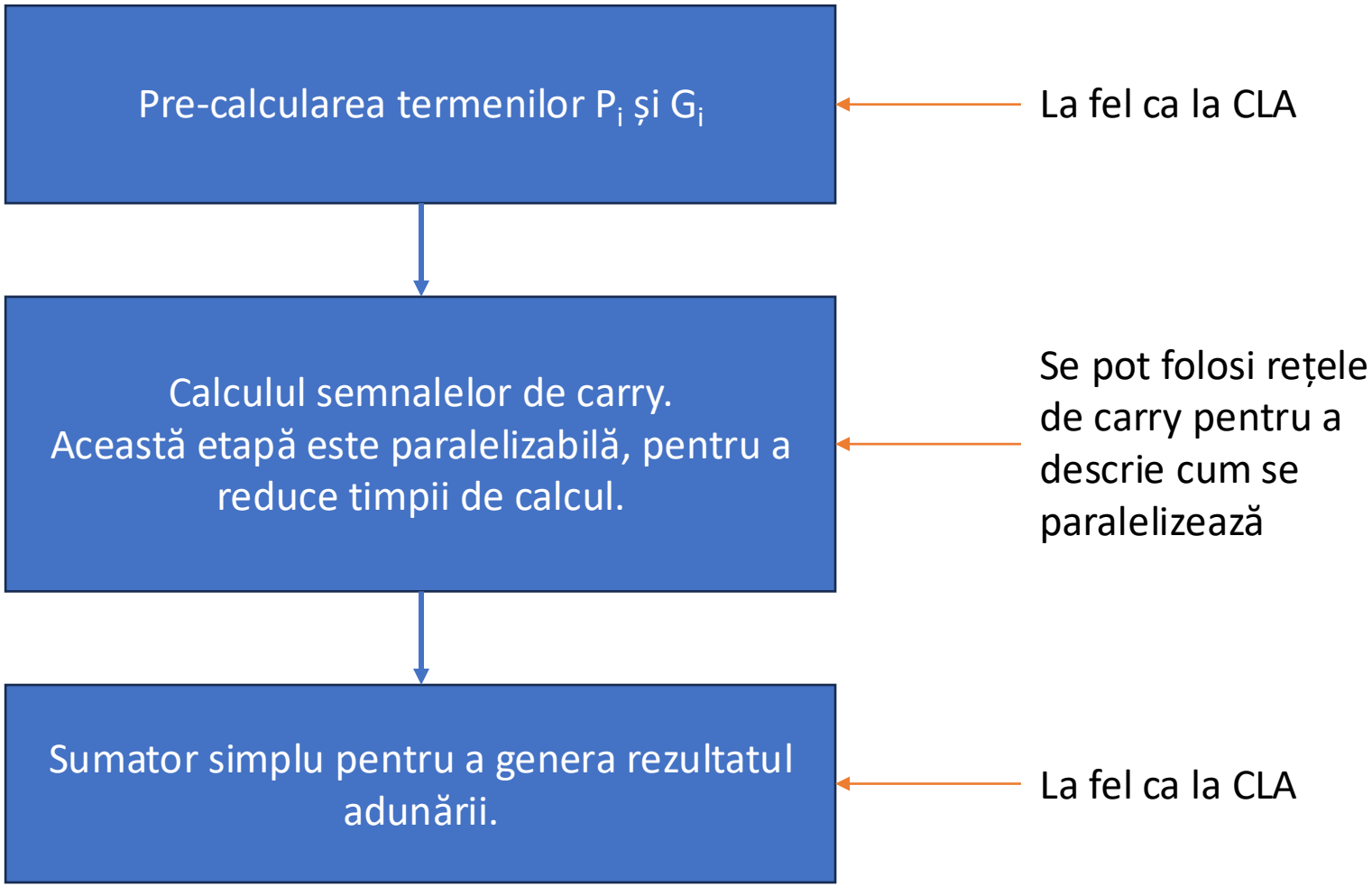
– **Generate:** blocul *i*:*j* va genera carry dacă:

- partea superioară (*i*:*k*) generează carry, sau
- partea superioară propagă un carry generat de partea inferioară (*k*-1:*j*)

– **Propagate:** blocul *i*:*j* va propaga carry dacă și blocul superior și cel inferior propagă carry-ul

Sumatoare cu prefixe

- Sumatorul folosește o structură similară cu cea a unui sumator carry look-ahead
- Îmbunătățirea este etapa de generare carry



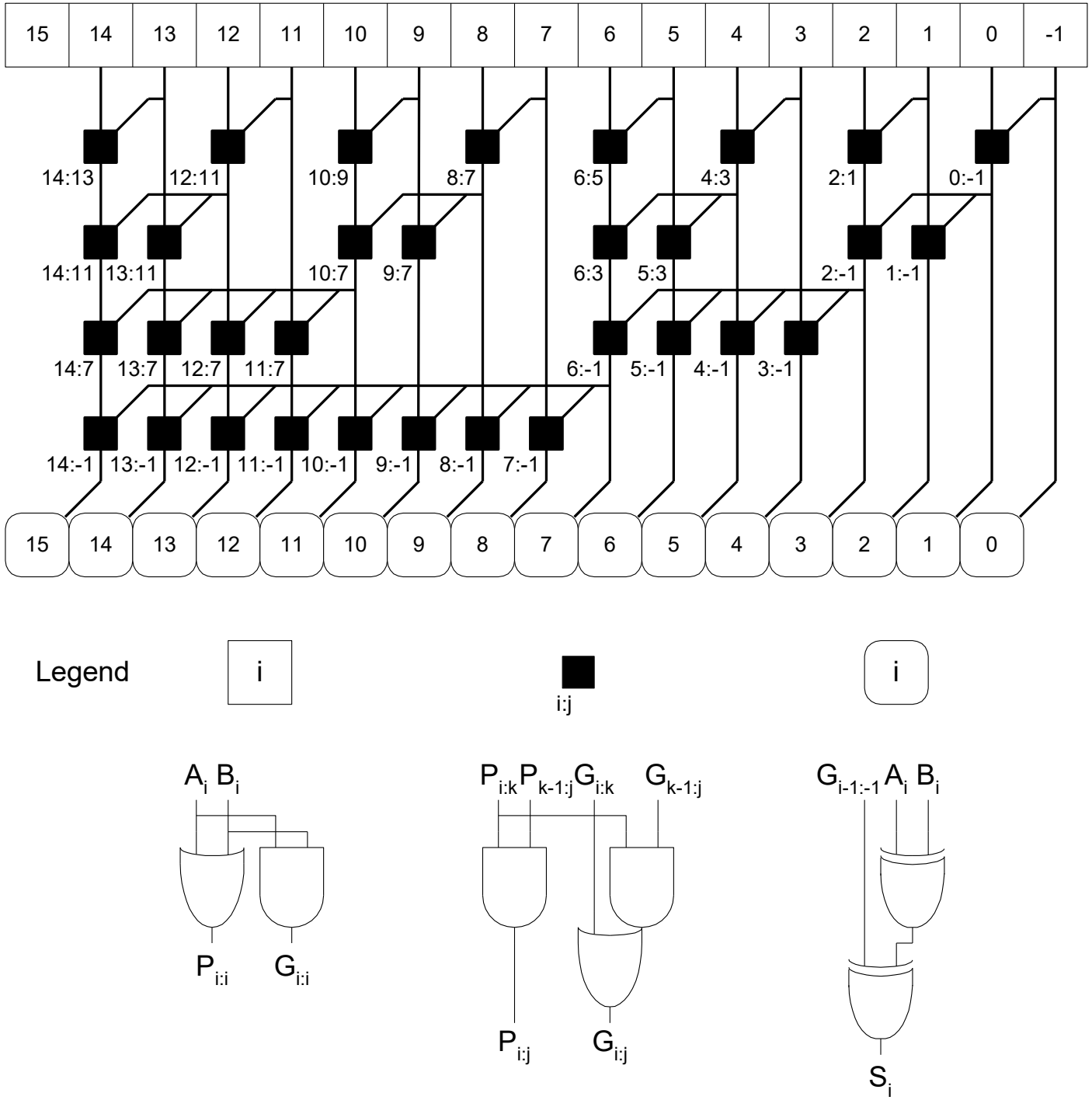
Prefix Adder

Structura generală are trei blocuri logice combinaționale distincte:

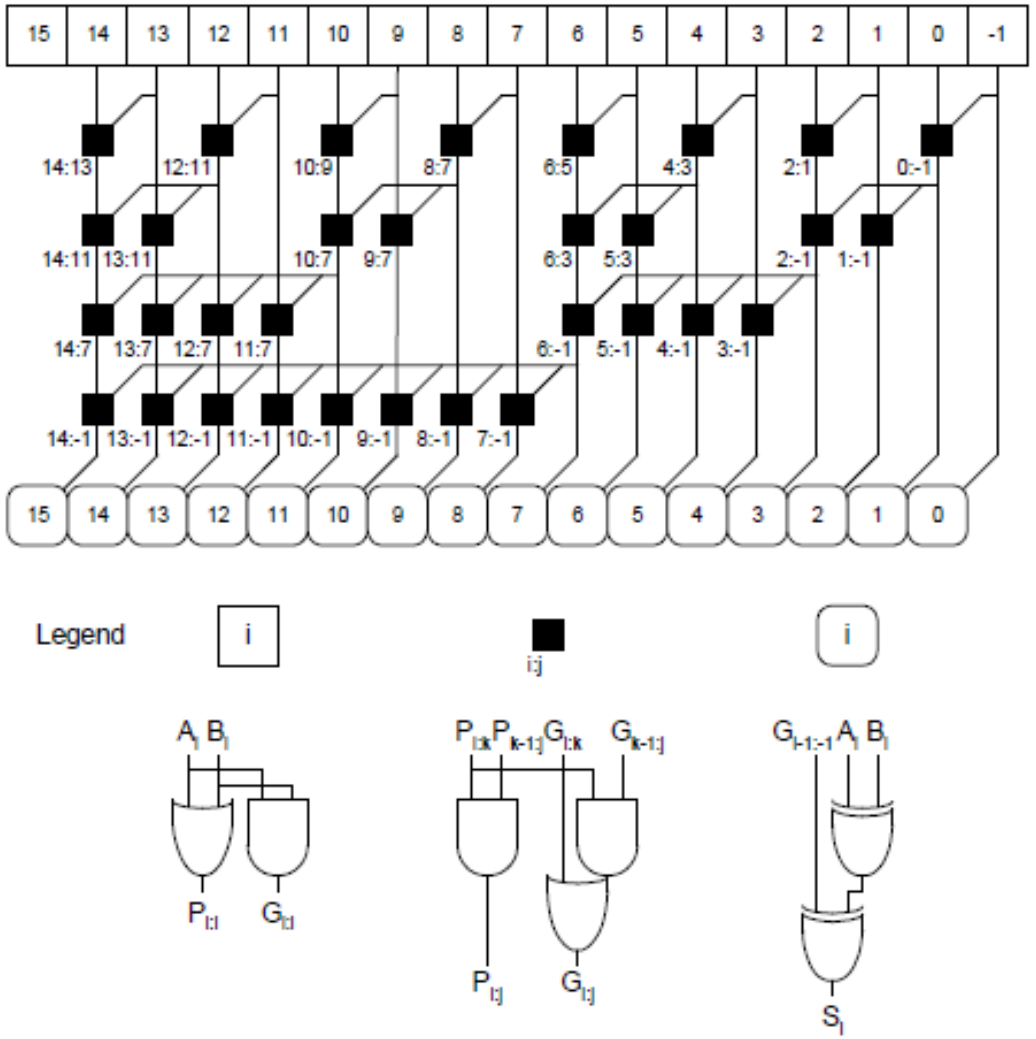
- Bloc pentru calculul P_i și G_i
- Bloc pentru calculul prefixelor $P_{i:j}$ și $G_{i:j}$
- Bloc pentru calculul

$$S_i = A_i \text{ XOR } B_i \text{ XOR } G_{i-1:-1}$$

Tot efortul este depus pentru a calcula cât mai rapid $G_{i-1:-1}$!



Schema unui sumator cu prefixe



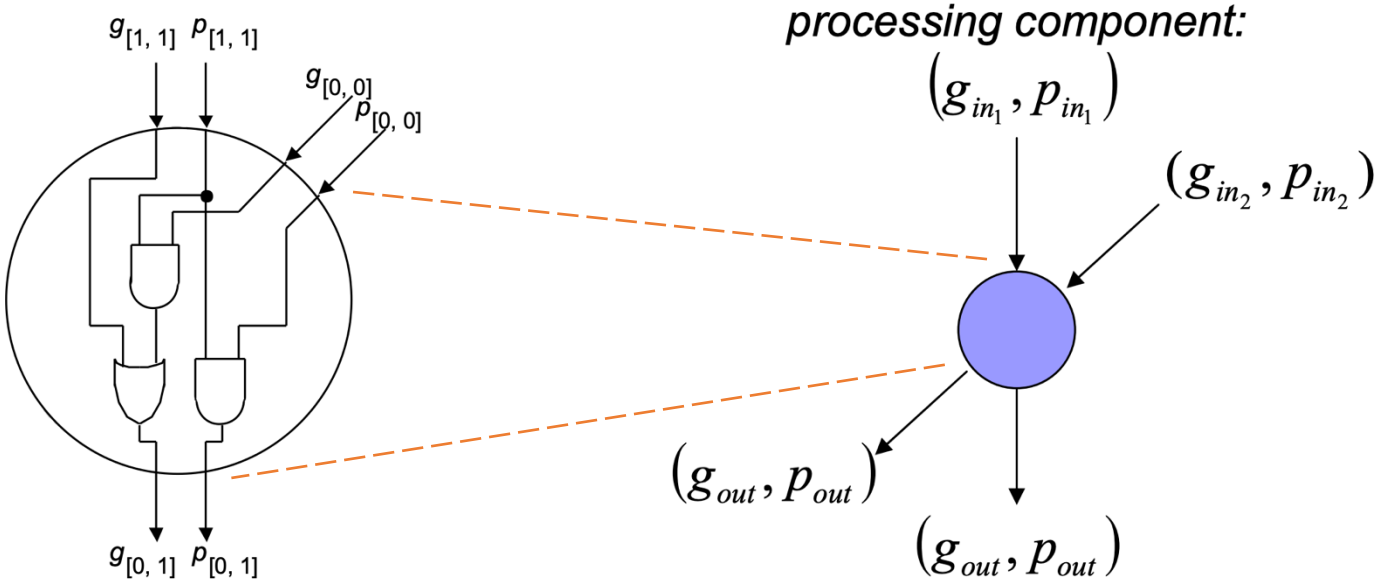
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_prefix}) + t_{XOR}$$

– t_{pg}: delay pentru a produce $P_i\,G_i$ (poartă AND sau OR)

– t_{pg_prefix}: delay introdus de celula (neagră) pentru calculul semnalelor combinate (i,j) (porți AND-OR)

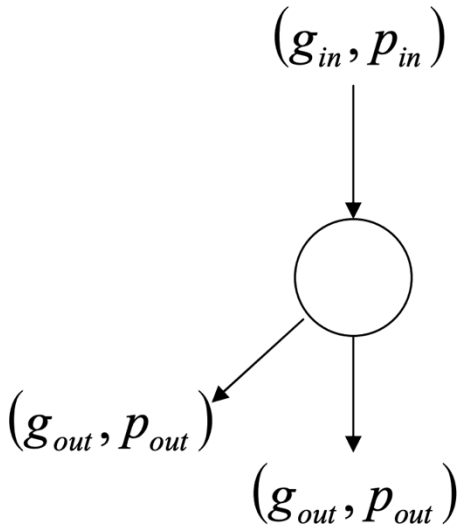
Grafuri prefix

Componentele dintr-un graf prefix sunt următoarele:



processing component:

buffer component:

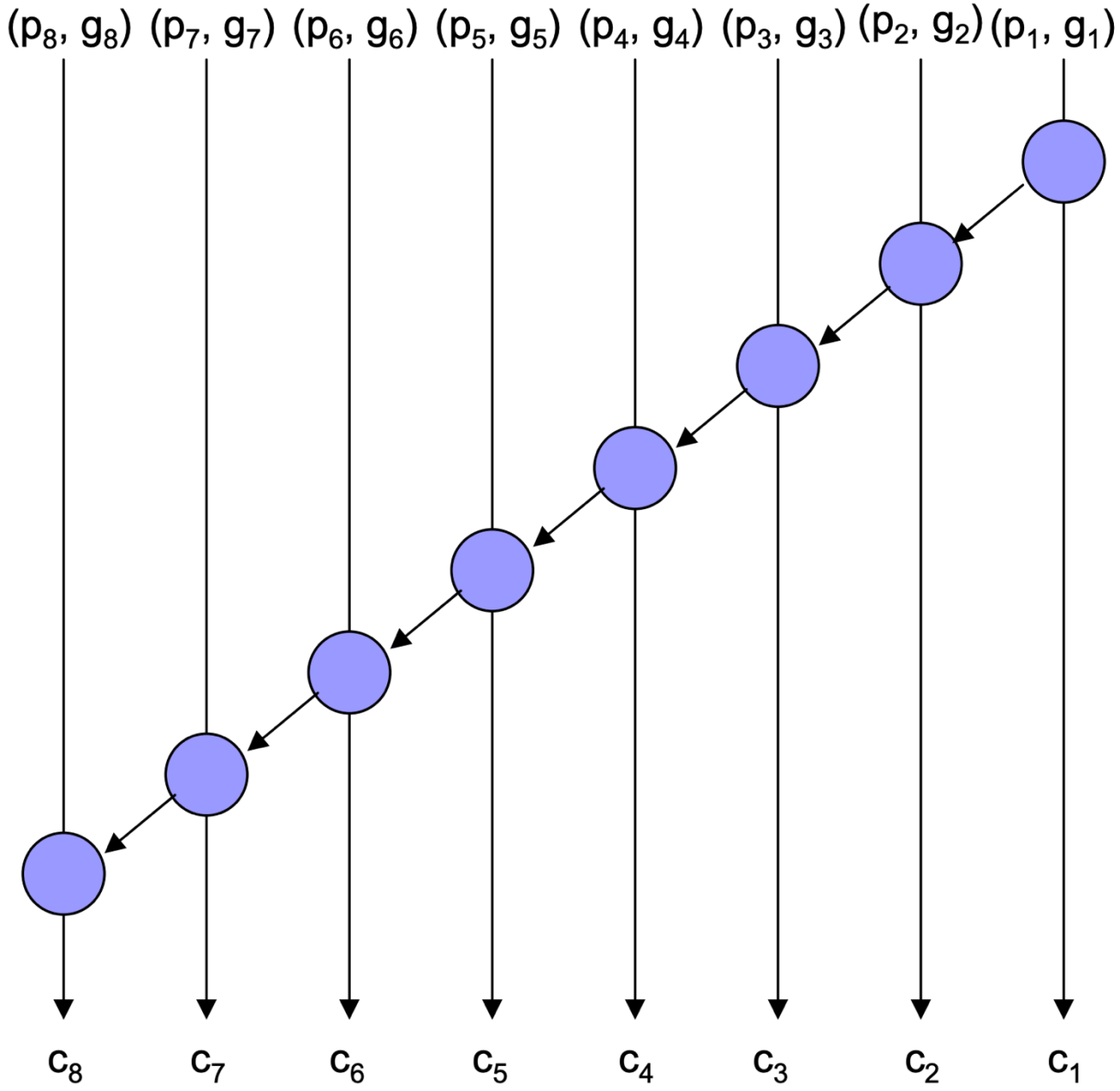


$$(g_{out}, p_{out}) = (g_{in_1} + p_{in_1} \cdot g_{in_2}, p_{in_1} \cdot p_{in_2})$$

$$(g_{out}, p_{out}) = (g_{in}, p_{in})$$

Exemplu simplu

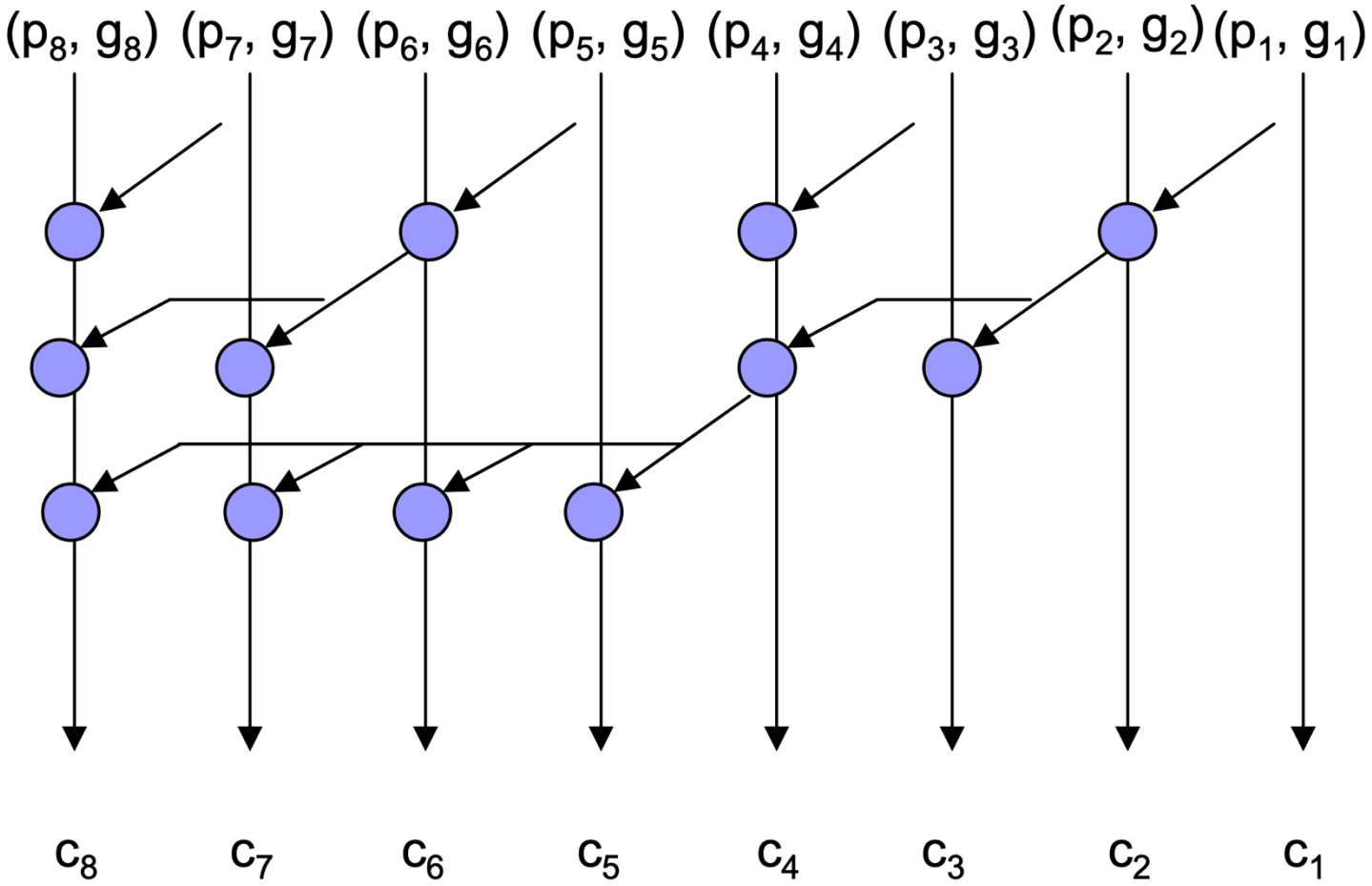
- Sumator secvențial
- Generarea semnalelor carry reprezentată prin graf



1960: J. Sklansky – sumator conditional

Sumatorul Sklansky:

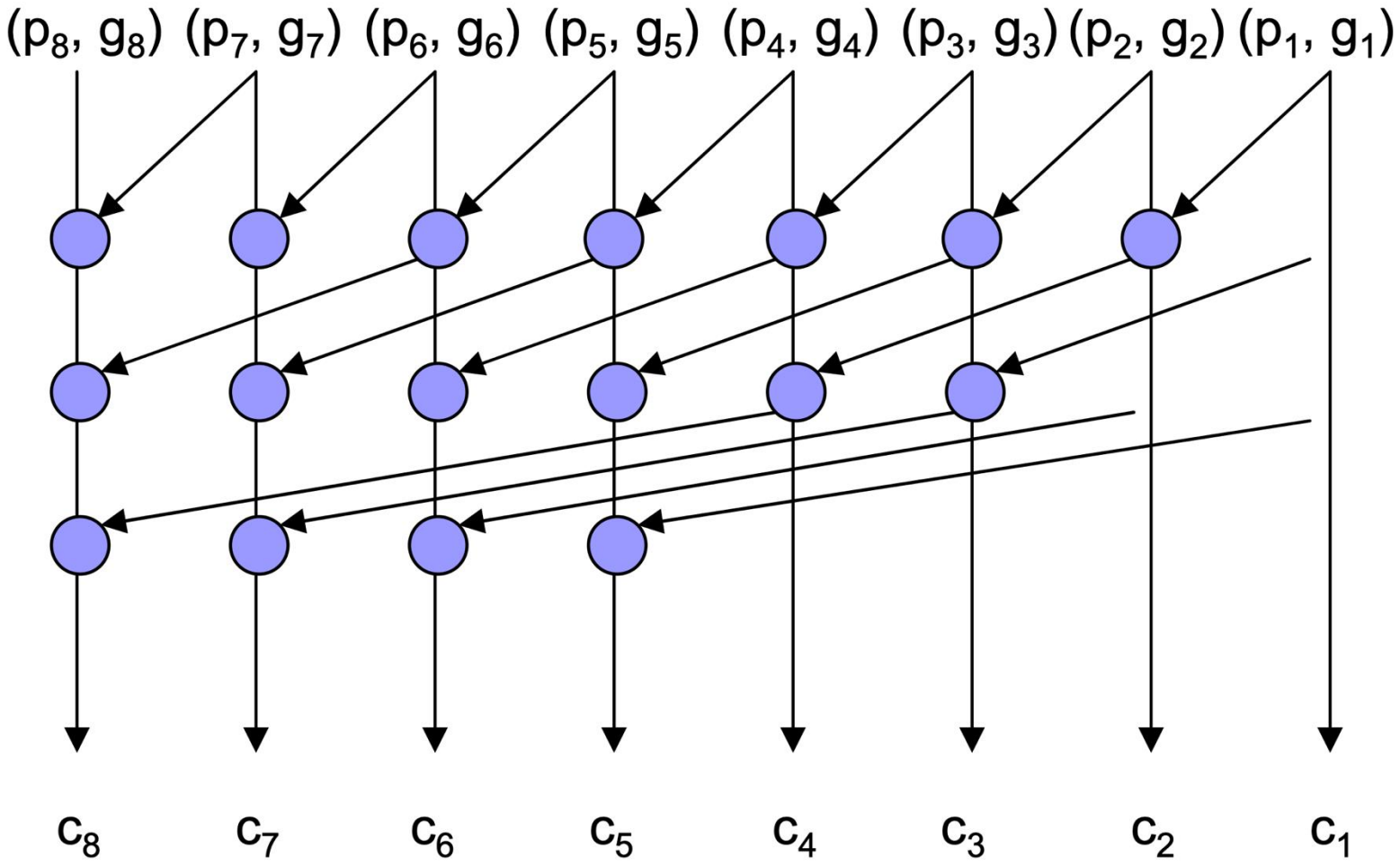
- Adâncime minimă
- Noduri cu fan-out mare



J. Sklansky, "Conditional-Sum Addition Logic", IRE transactions on computers, 1960

1973: Sumator Kogge-Stone

- Adâncime mică
- Multe noduri (suprafață mare)
- Fan-out minim (performanță mare)

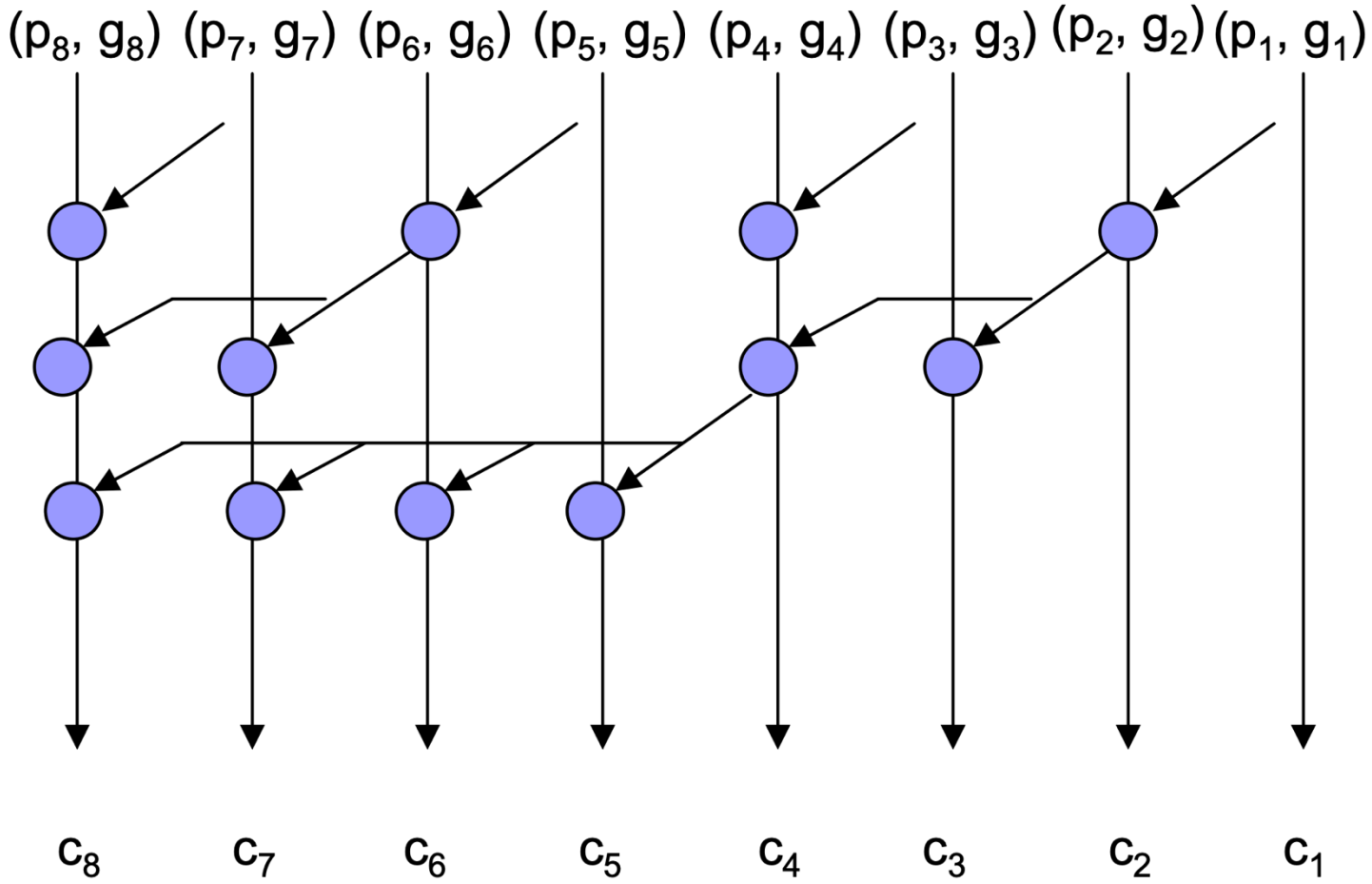


Kogge, Stone, "A Parallel Algorithm for the Efficient solution of a General Class of Recurrence equations", IEEE, 1973

1980: Sumator Ladner-Fischer

Structură similară cu Sklansky

- Adâncime minimă
- Noduri cu fan-out mare

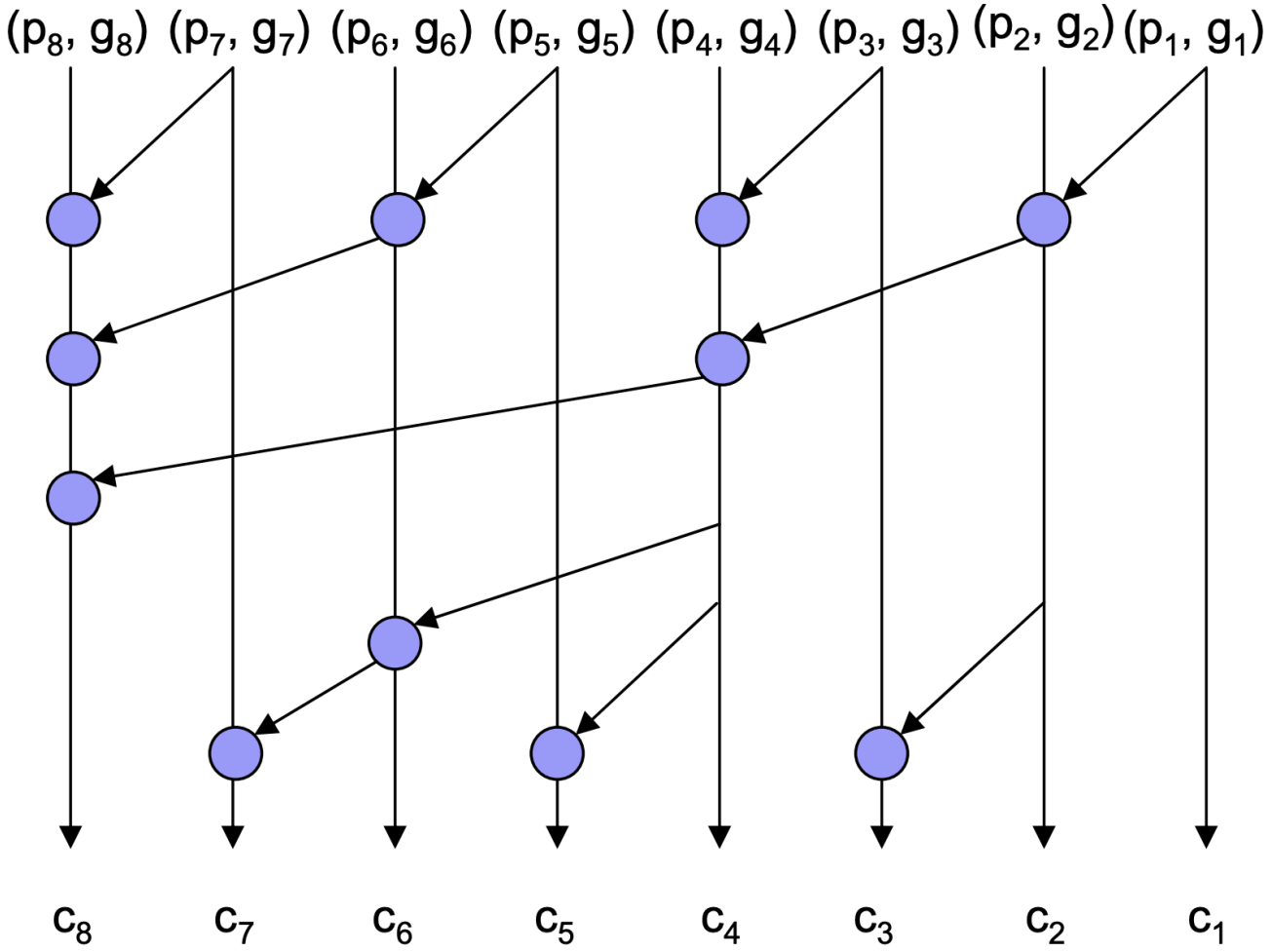


Ladner, Fischer, “Parallel Prefix Computation”, ACM, 1980

1982: Sumator Brent-Kung

Sumatorul Brent Kung este un caz limită

- Adâncime logică maximă (timp de calcul mai mare)
- Număr minim de noduri (suprafață minimă în siliciu)

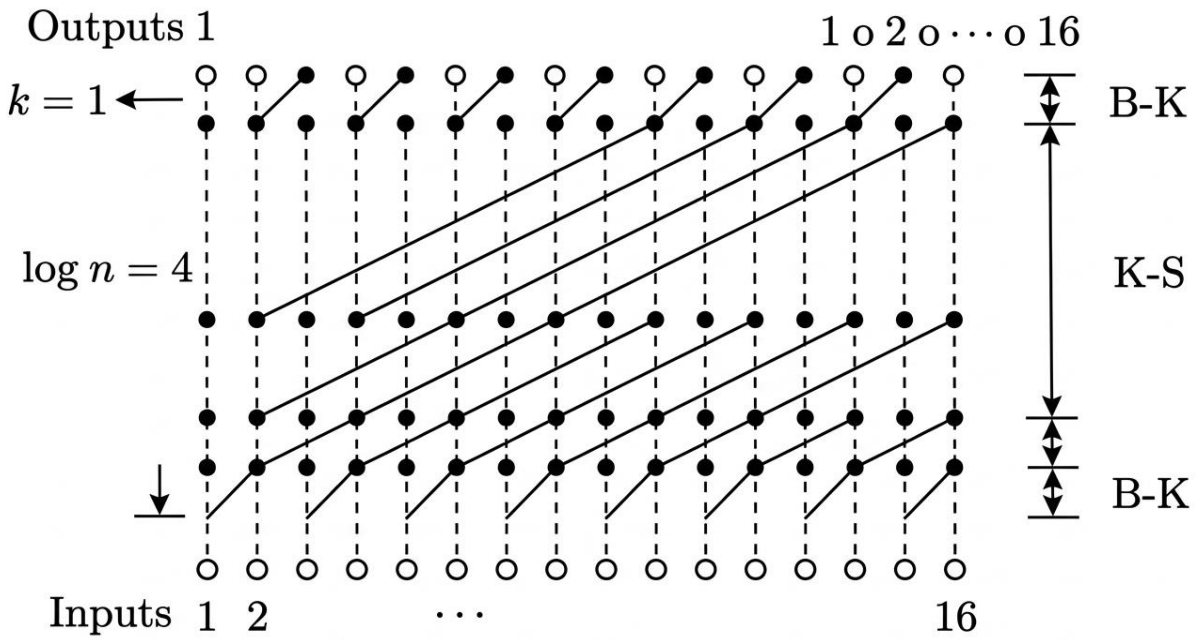


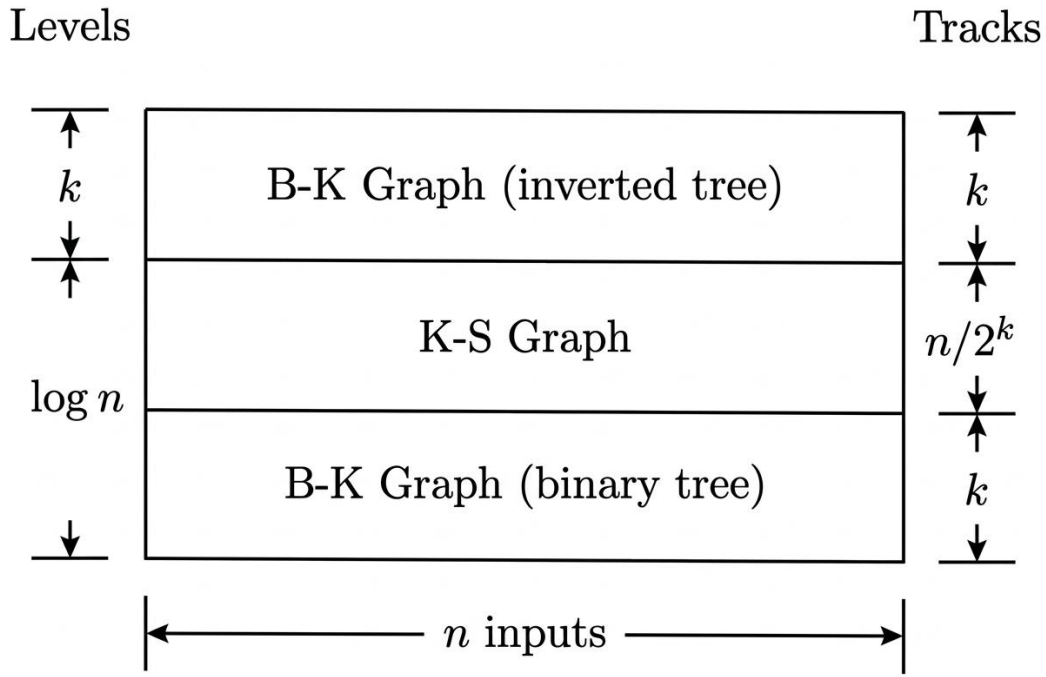
Brent, Kung, "A regular Layout for Parallel Adders", IEEE, 1982

1987: Sumator Han Carlson

Sumatorul Han-Carlson combină Brent-Kung și Kogge-Stone

- Structură hibridă
- Eficient
- Potrivit pentru implementare VLSI.





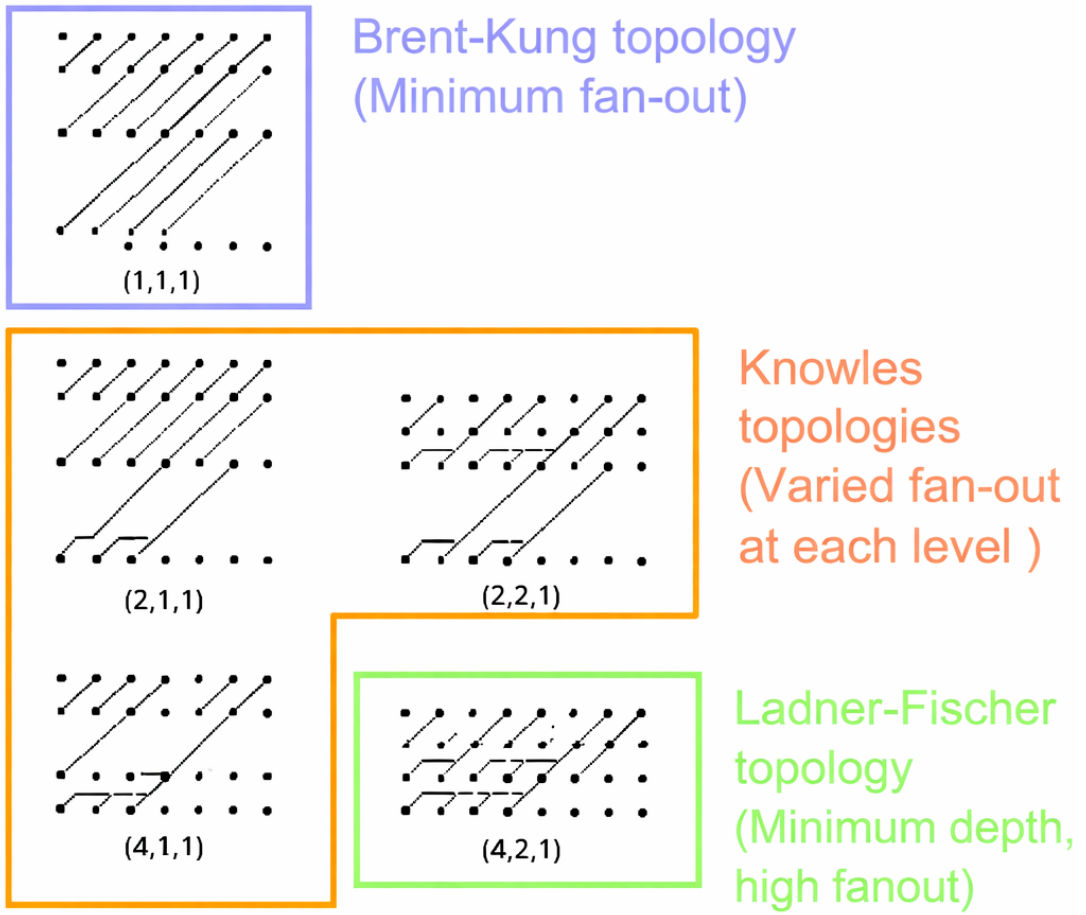
Han, Carlson, "Fast Area-Efficient VLSI Adders, IEEE, 1987

2001: S. Knowles

Knowles a propus sumatoare care realizează un compromis între:

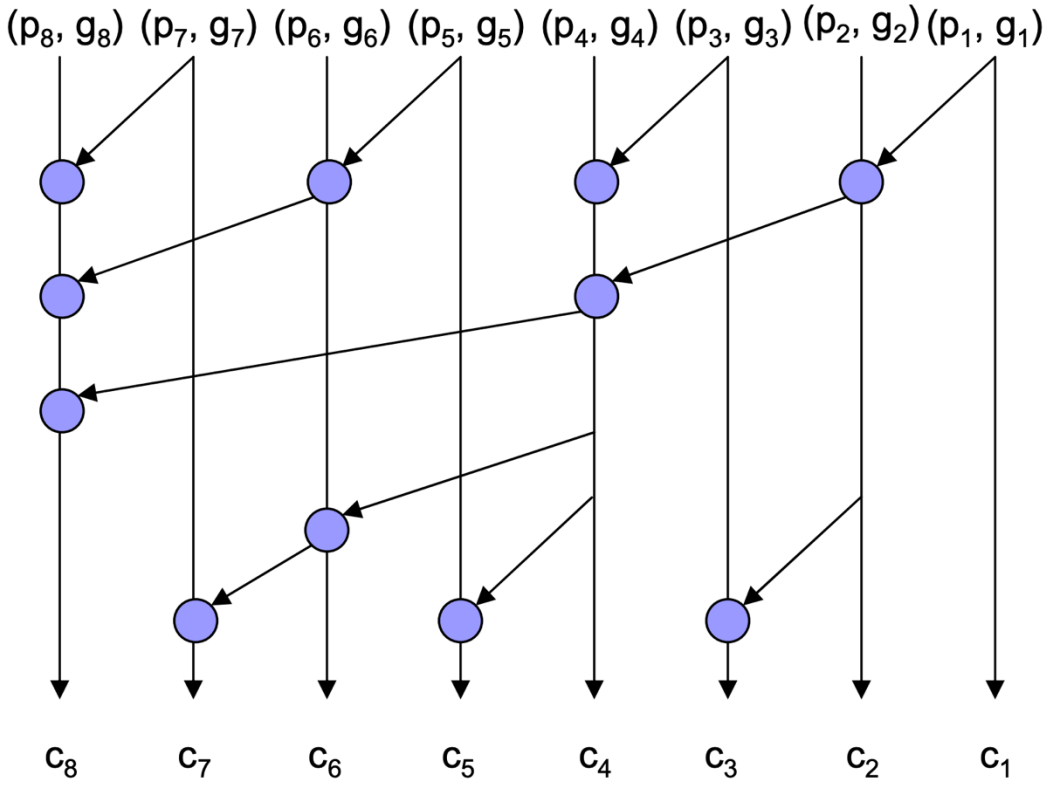
- Adâncime,
- Interconectare
- Suprafață

Aceste sumatoare sunt încadrate între topologiile Ladner-Fischer (adâncime minimă) și Brent-Kung (fanout minim).

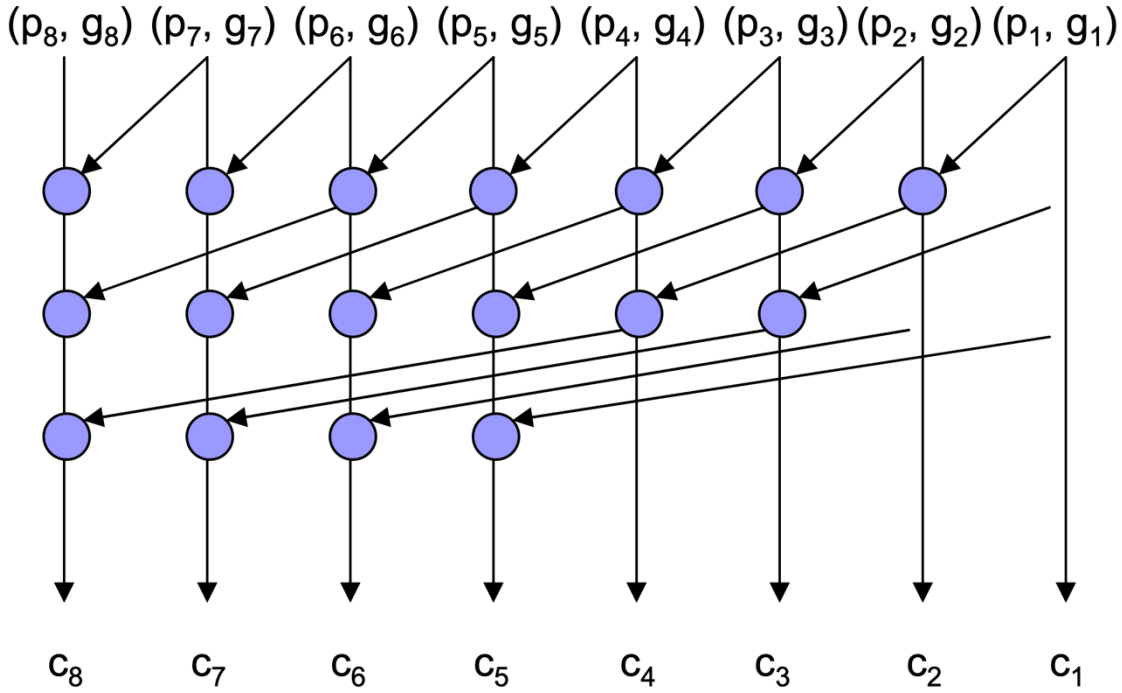


Simon Knowles, "A Family of adders", IEEE, 2001

Brent-Kung vs. Kogge-Stone Carry Network



11 operatori carry
4 niveluri



17 operatori carry
3 niveluri

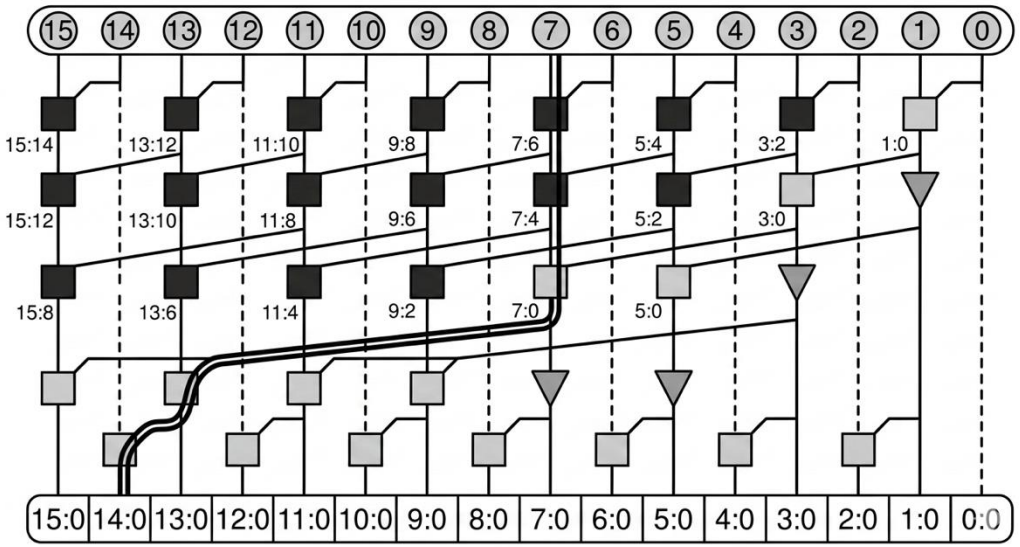
O taxonomie interesantă

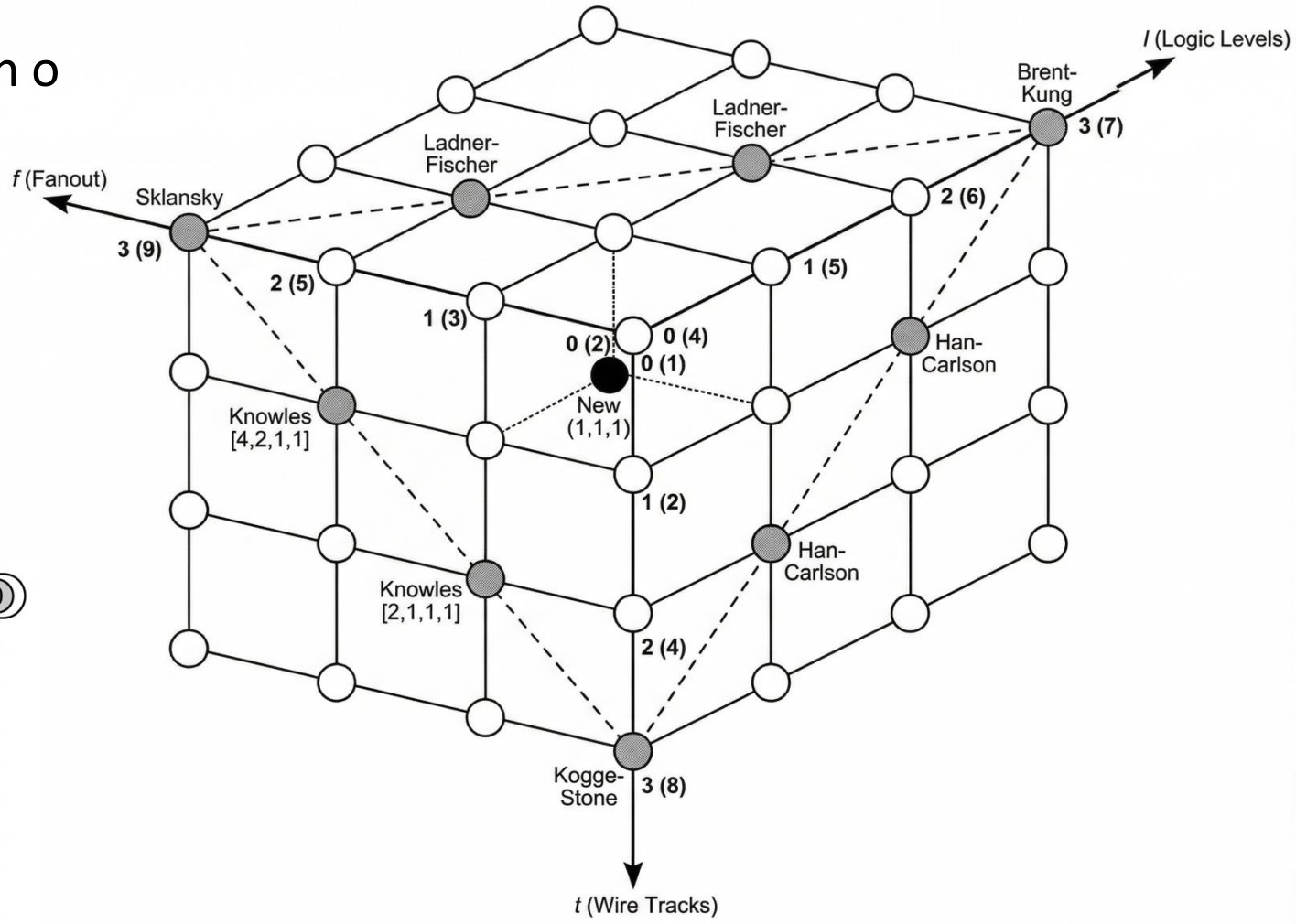
Harris[2003] a organizat principalele arhitecturi pentru sumatoare cu prefixe în o taxonomie 3D.

Fiecare axă reprezintă o caracteristică a sumatoarelor:

- Fanout
- Adâncime logică
- Complexitate conexiuni

Tot el a propus arhitectura de mai jos





D. Harris, "A Taxonomy of Parallel Prefix Networks", IEEE, 2003

Compararea performanțelor

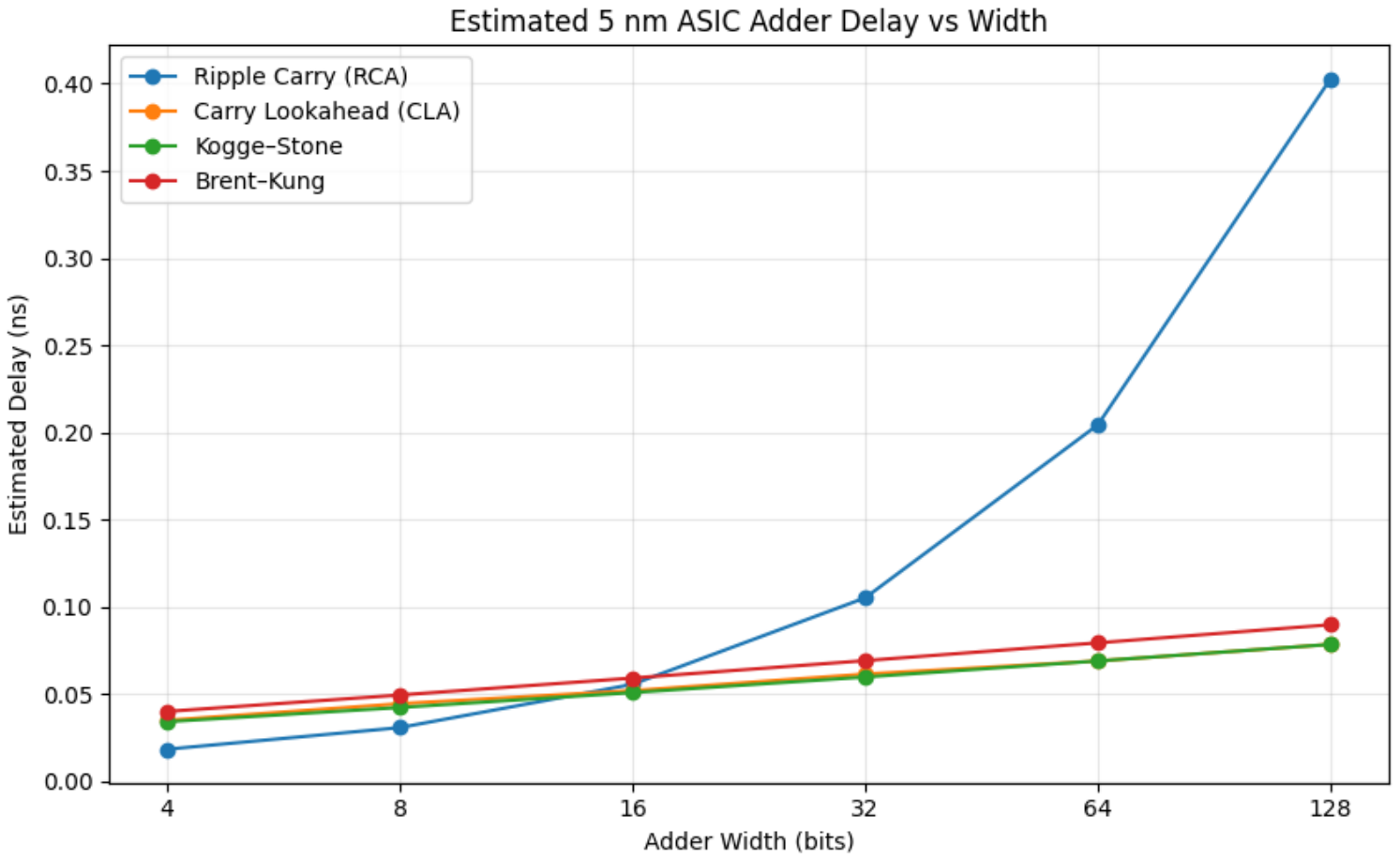
Comparăm întârzierile prin trei sumatoare de 32 de biți cu ripple-carry, carry-lookahead și prefixe

- CLA are blocuri de 4-biți
- 2-input gate delay = 100 ps; full adder delay = 300 ps

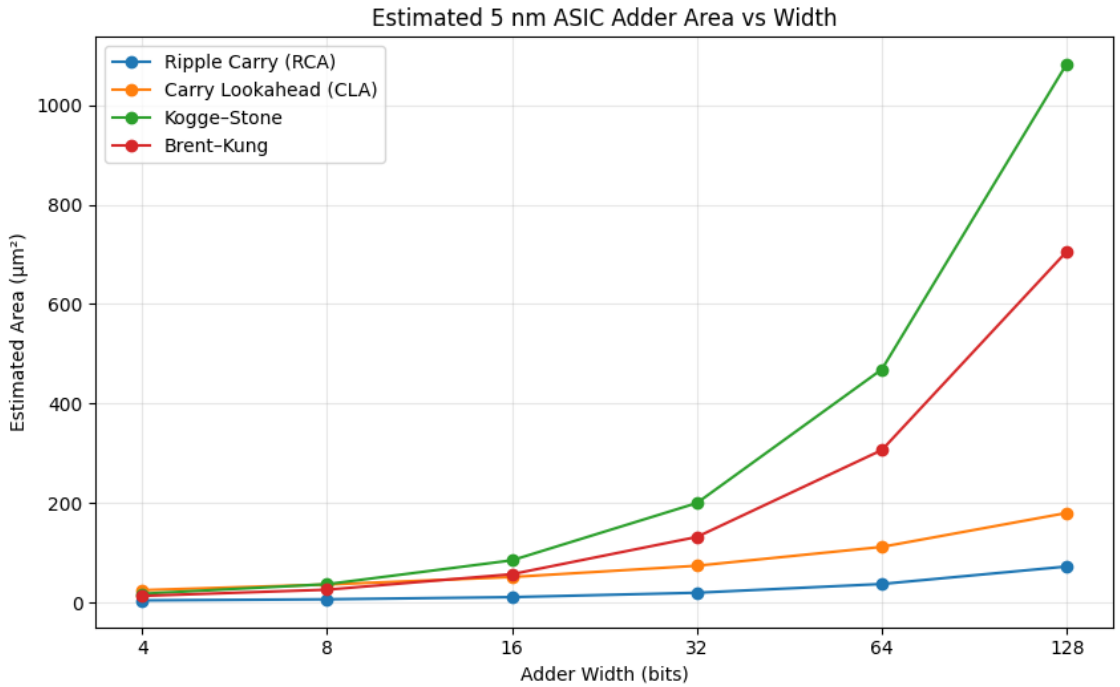
$$t_{\text{ripple}} = Nt_{FA} = 32(300 \text{ ps})$$
$$= \mathbf{9.6 \text{ ns}}$$
$$t_{CLA} = t_{pg} + t_{pg\_\text{block}} + (N/k - 1)t_{\text{AND\_OR}} + kt_{FA}$$
$$= [100 + 600 + (7)200 + 4(300)] \text{ ps}$$
$$= \mathbf{3.3 \text{ ns}}$$
$$t_{PA} = t_{pg} + \log_2 N(t_{pg\_\text{prefix}}) + t_{\text{XOR}}$$
$$= [100 + \log_2 32(200) + 100] \text{ ps}$$
$$= \mathbf{1.2 \text{ ns}}$$

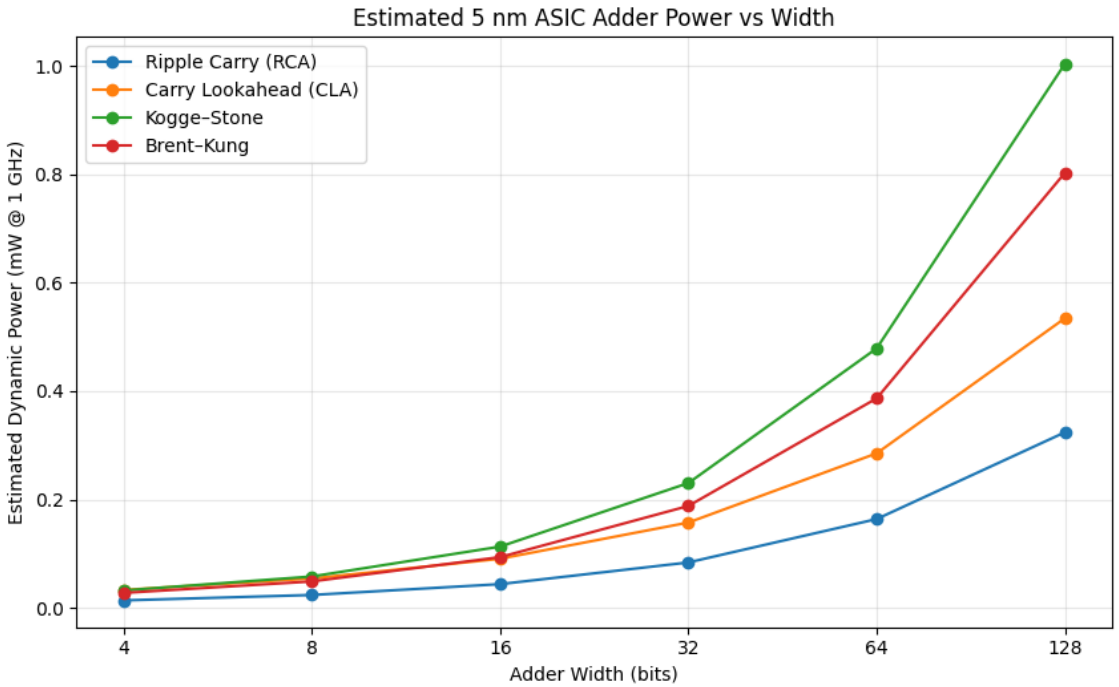
Performanța la 5nm (o estimare)

- **RCA** e cea mai bună opțiune când contează simplitatea din punct de vedere al ariei și consumului de putere (IoT & Embedded).
- **Kogge–Stone** este cea mai rapidă la lățimi mai mari.
- **Brent–Kung** e puțin mai lentă decât Kogge–Stone, dar este adesea preferată atunci când contează constrângerile legate de rutare și suprafață.
- **CLA** rămâne competitivă pentru lățimi moderate, dar poziționarea ei exactă față de Brent–Kung depinde foarte mult de stilul de implementare.



Performanța nu este totul!





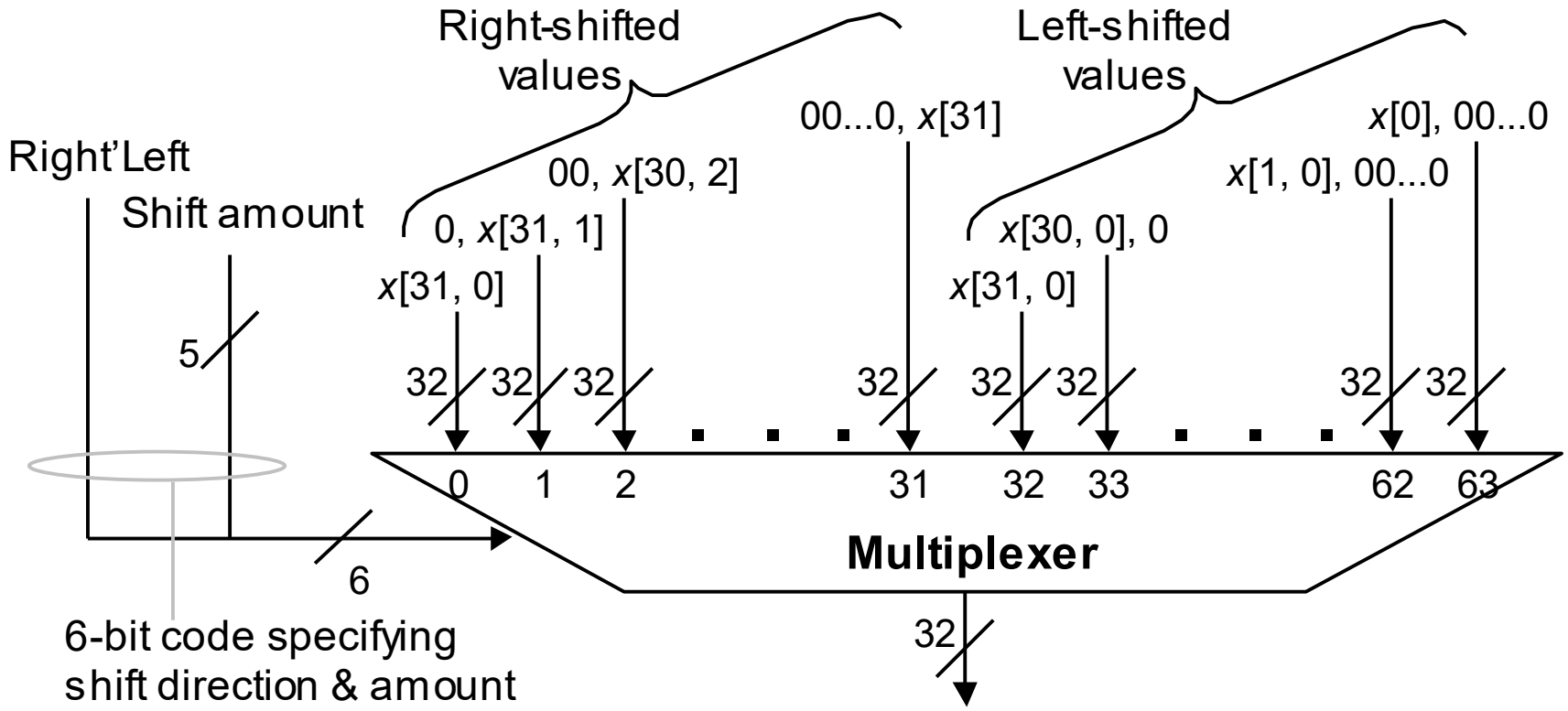
- **RCA** este cel mai mic ca suprafață și cel mai low-power
- **CLA** crește moderat
- **Brent–Kung** este mai costisitor decât CLA, mai eficient decât Kogge–Stone
- **Kogge–Stone** este cel mai optimizat pentru viteză, plătește asta prin cel mai mare cost suprafață/putere disipată

Performanța sistemului e definită de eficiența sumatorului

Arhitectură	Caracteristici	Utilizare tipică
RCA	Latență mare, arie mică, putere mică	IoT, microcontrollere low-power, circuite ne-critice
Carry skip/select	Compromis mediu	DSP, FPGA, aplicații multimedia
Prefix (Kogge-Stone)	Viteză mare, arie mare, putere mare	Procesoare desktop/server.

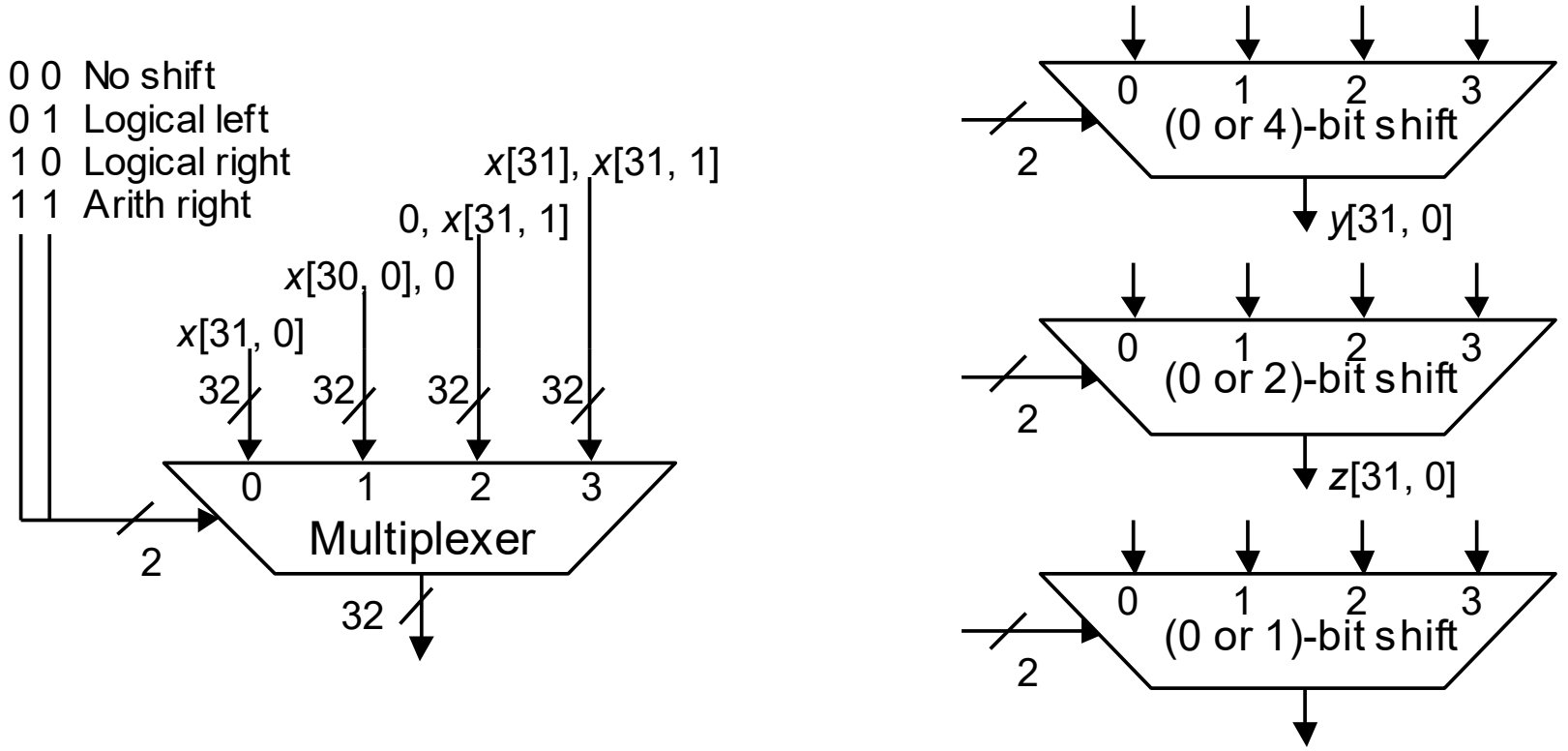
Operații logice și de shiftare

Conceptual, shiftările pot fi implementate pur combinațional prin multiplexoare



Unitate de shiftare logică folosind multiplexoare.

Implementarea shiftării pe mai multe niveluri



(a) Single-bit shifter

(b) Shifting by up to 7 bits

Shiftare pe mai multe niveluri într-un barrel shifter.

Scăzător

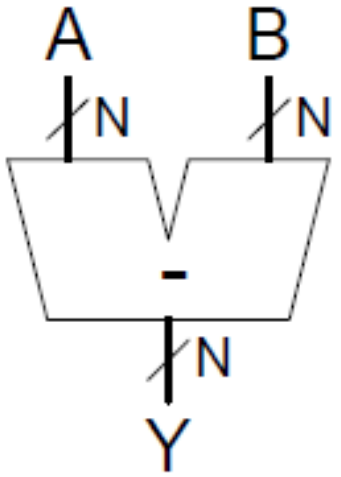
Un scăzător este un sumator în care al doilea operand este un număr negativ

- A - B = A + (-B)

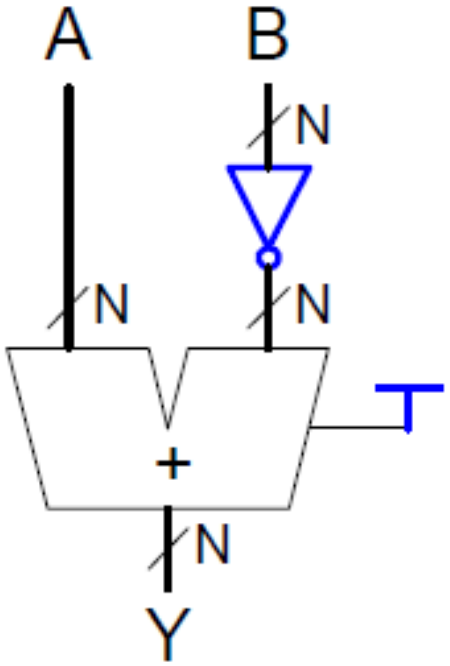
Numărul negativ este reprezentat în cod complement al lui 2

- -B = $\bar{B}$ + 1

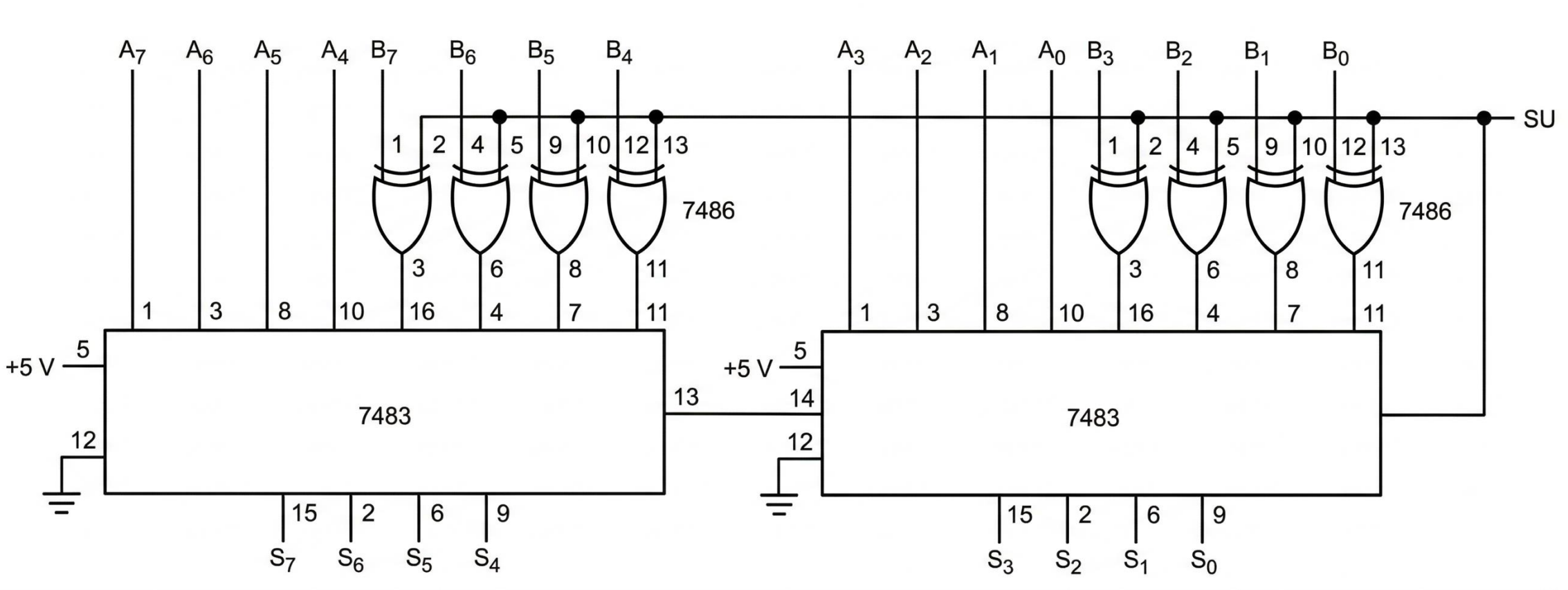
Simbol



Implementare



Sumator / Scăzător binar pe 8 biți



74LS83 – Sumator binary complet pe 4 biți cu logică de fast carry

74HC86 – Quad XOR 2-input gate

Comparator: Egalitate

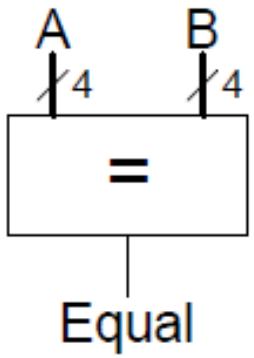
Un comparator trece toți biții operanzilor, rang cu rang, prin porți NXOR

Dacă biții coincid,
A NXOR B = 1

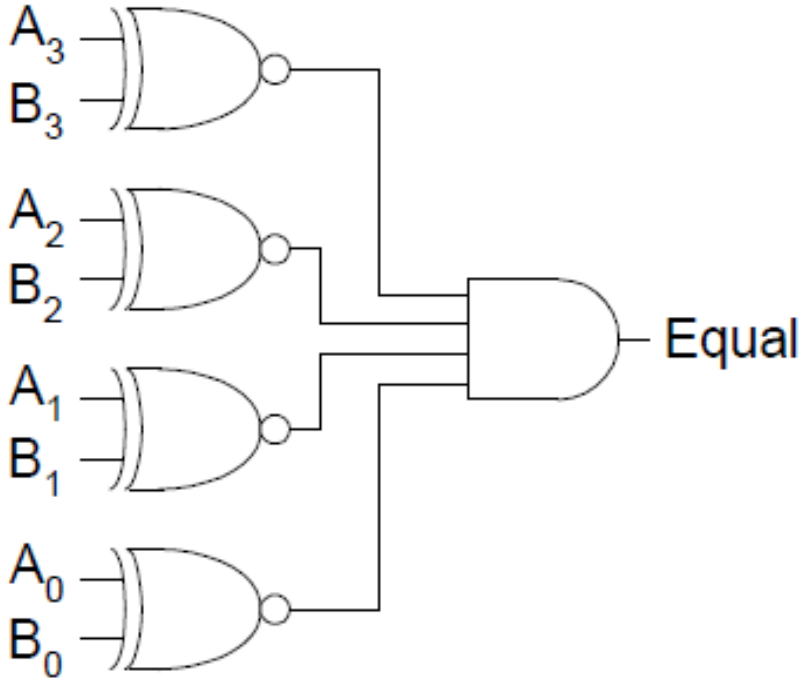
Paralelizare perfectă

- Foarte rapid!

Simbol

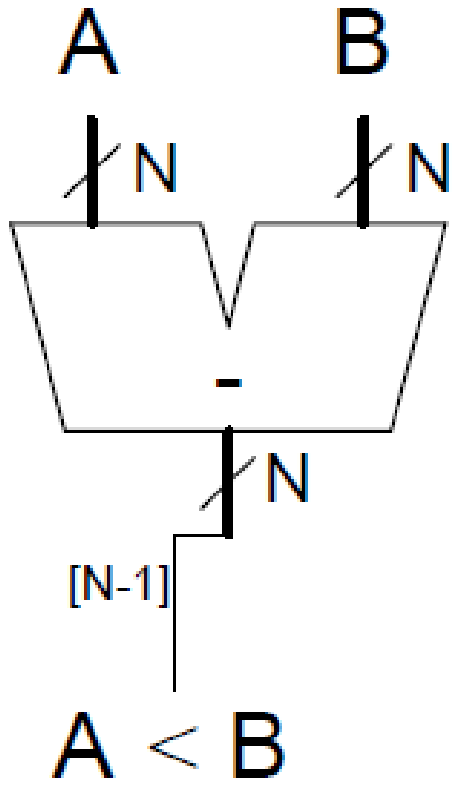


Implementare

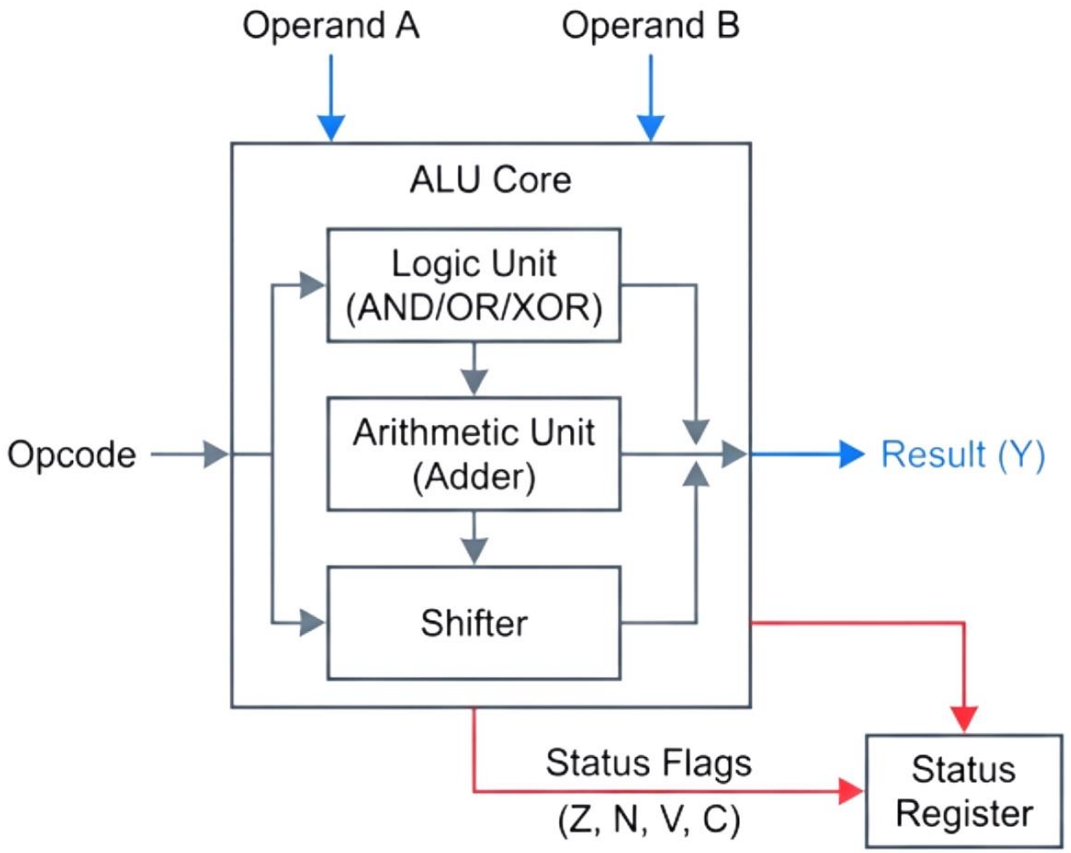


Comparator: “Mai mic decât”

- Operanzii pot fi numere positive sau negative (cod complement al lui 2)
- MSB-ul are rolul de **bit de semn** (0 pentru numere pozitive, 1 pentru numere negative).
- Dacă A este mai mic decât B, rezultatul operației A - B va fi negativ.
- Dacă rezultatul este negativ, bitul său de semn (adică bitul [N-1]) va fi egal cu 1.
- Simpla citire a acestui bit de semn ne dă răspunsul la întrebarea „este A < B?”.



Unitatea Aritmetică Logică (UAL)



Z (Zero): Rezultat = 0 (NOR logic pe ieșire).

N (Negative): MSB al rezultatului (S_{n-1}).

V (Overflow): Eroare de semn ($C_n \oplus C_{n-1}$).

C (Carry): Transport final (pentru numere fără semn).

Acknowledgements

- Aceste slide-uri conțin materiale aparținând:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Behrooz Parhami (UCSB)

- MIT material derived from course 6.823
- UCB material derived from course CS252