Computer Science & Engineering Department

POLITEHNICA BUCUREȘTI 1818

Structura și Organizarea Calculatoarelor

– Cursul 4 –

Multiplicatoare binare

Facultatea de Automatică și Calculatoare
Universitatea Politehnica București


THREE BEERS.

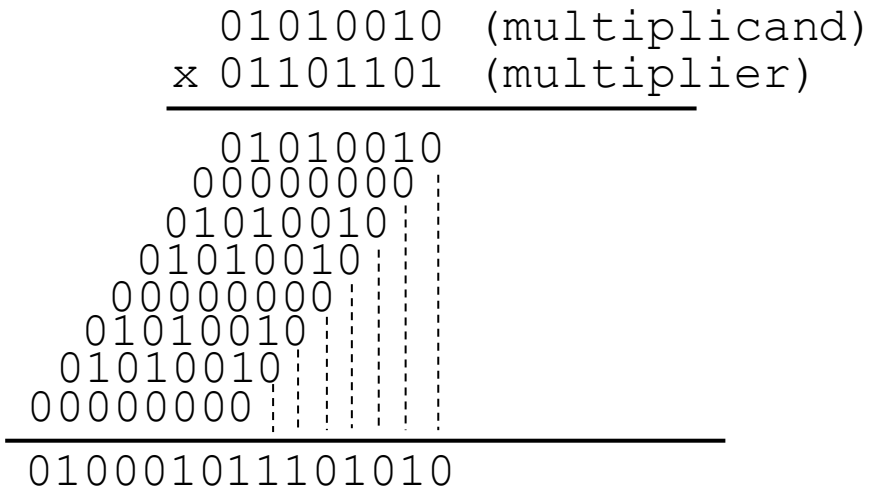
Înmulțirea întregilor

- Mai complicată decât adunarea
  - O implementare directă presupune deplasări și adunări
- Operațiile mai complexe pot duce la
  - Suprafață mai mare în siliciu
  - Timp mai lung de execuție (mai mulți cicli sau ciclu de ceas mai lung)
- Să începem cu metoda directă

Înmulțirea binară (numere întregi)

$$
\begin{array}{lccccccccccc}
 & & & & & & a_4 & a_3 & a_2 & a_1 & a_0 & \\
 & & & & & x & x_4 & x_3 & x_2 & x_1 & x_0 & \\
\hline
ax_0\ 2^0 & & & & & & a_4x_0 & a_3x_0 & a_2x_0 & a_1x_0 & a_0x_0 & \\
ax_1\ 2^1 & & & & & a_4x_1 & a_3x_1 & a_2x_1 & a_1x_1 & a_0x_1 & & \\
ax_2\ 2^2 & & & & a_4x_2 & a_3x_2 & a_2x_2 & a_1x_2 & a_0x_2 & & & \\
ax_3\ 2^3 & & & a_4x_3 & a_3x_3 & a_2x_3 & a_1x_3 & a_0x_3 & & & & \\
ax_4\ 2^4 & & a_4x_4 & a_3x_4 & a_2x_4 & a_1x_4 & a_0x_4 & & & & & \\
\hline
 & p_9 & p_8 & p_7 & p_6 & p_5 & p_4 & p_3 & p_2 & p_1 & p_0 &
\end{array}
$$

Algoritm simplu



Algoritm simplu

```
      01010010 (multiplicand)
    x 01101101 (multiplier)
-----------------------------
0000000000000000 (partial sum)
```

Algoritm simplu

```
        01010010  (multiplicand)
      x 01101101  (multiplier)
-----------------------------------
000000001010010  (partial sum)
```

Algoritm simplu

```
  01010010 (multiplicand)
x 01101101 (multiplier)
----------
000000001010010 (partial sum)
  00000000
```

Algoritm simplu

```
  01010010 (multiplicand)
x 01101101 (multiplier)
----------
000000001010010 (partial sum)
```

Algoritm simplu

```
    01010010 (multiplicand)
  x 01101101 (multiplier)
  ------------------------
000000001010010 (partial sum)
    01010010
```

Algoritm simplu

```
  01010010 (multiplicand)
x 01101101 (multiplier)
----------
000000110011010 (partial sum)
```

Algoritm simplu

```
  01010010 (multiplicand)
x 01101101 (multiplier)
----------
000000110011010 (partial sum)
   01010010
```

Algoritm simplu

```
  01010010 (multiplicand)
x 01101101 (multiplier)
----------
000010000101010 (partial sum)
```

Algoritm simplu

```
  01010010 (multiplicand)
    x 01101101 (multiplier)
  ----------
000010000101010 (partial sum)
   00000000
```

Algoritm simplu

```
  01010010 (multiplicand)
x 01101101 (multiplier)
----------
000010000101010 (partial sum)
```

Algoritm simplu

```
  01010010 (multiplicand)
     x 01101101 (multiplier)
  ------------------------
000010000101010 (partial sum)
  01010010
```

Algoritm simplu

```
  01010010 (multiplicand)
x 01101101 (multiplier)
----------
00011100 1101010 (partial sum)
```

Algoritm simplu

```
 01010010 (multiplicand)
      x 01101101 (multiplier)
-----------------------------
000111001101010 (partial sum)
  01010010
```

Algoritm simplu

```
01010010 (multiplicand)
     x 01101101 (multiplier)
-----------------
010001011101010 (partial sum)
```

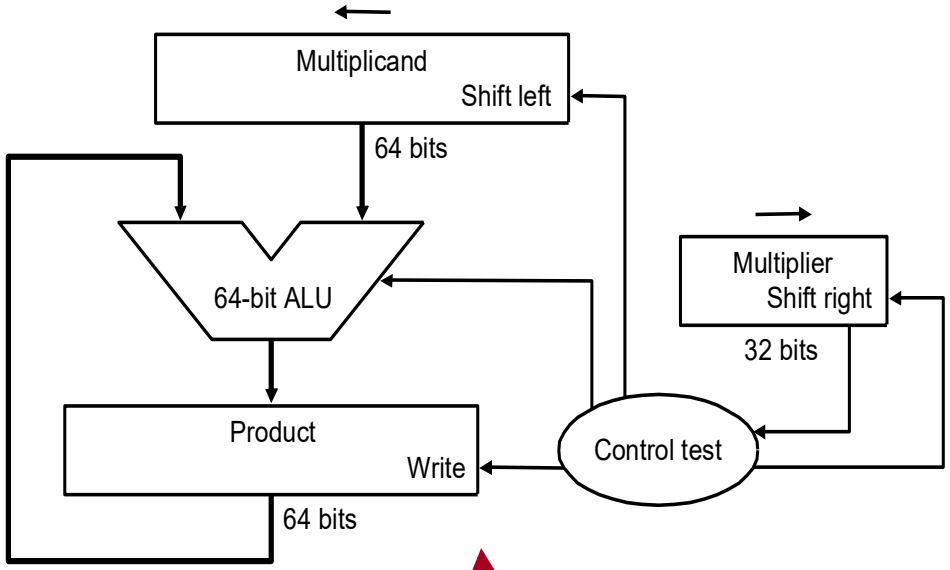
Algoritm simplu

```
01010010 (multiplicand)
      x 01101101 (multiplier)
-----------------------------
010001011101010 (partial sum)
00000000
```

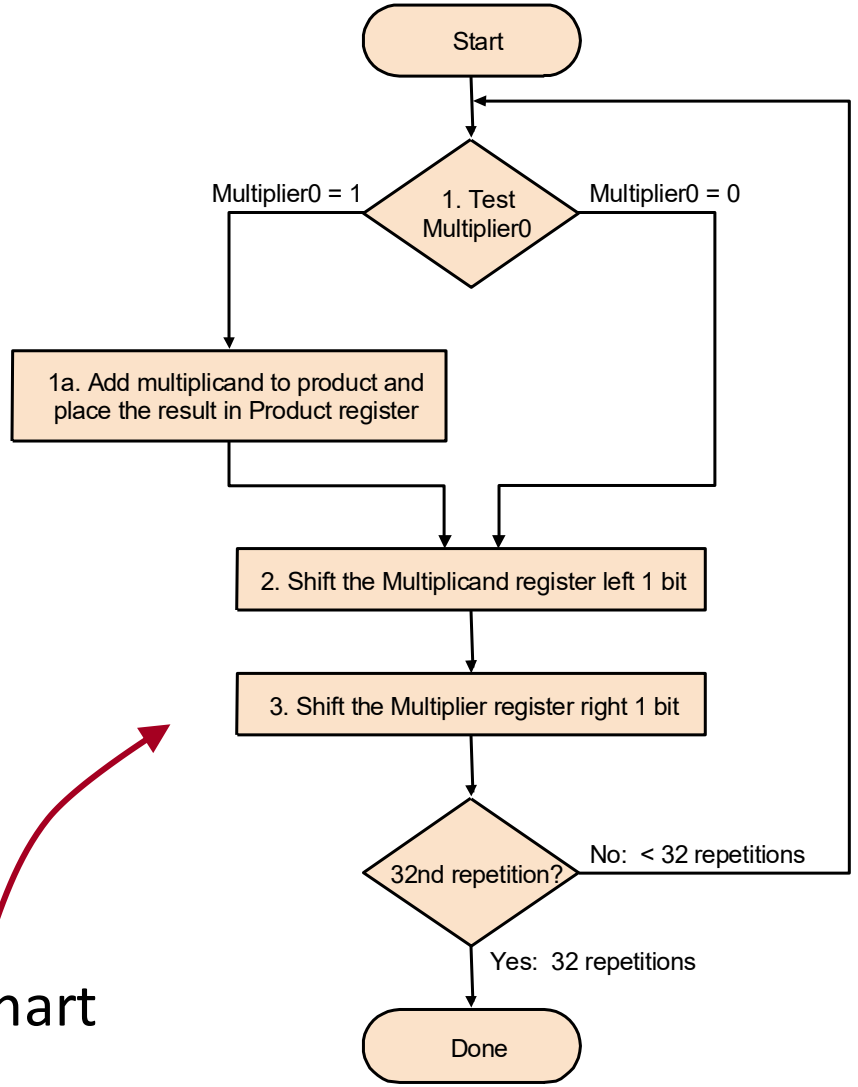
Algoritm simplu

```
01010010 (multiplicand)
      x 01101101 (multiplier)
--------------------------
010001011101010 (full sum)
```

Implementarea hardware



Circuitul pentru înmulțire



Flow Chart

Exemplu

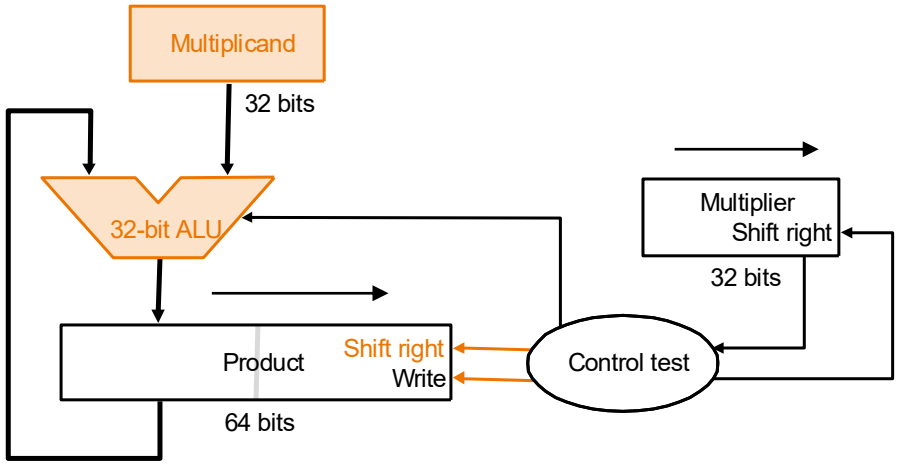
Să calculăm 0010 x 0110 (2 x 6), fără semn

Iterație	Implementarea 1			
	Pas	Multiplier	Multiplicand	Produs
0	initial values	0110	0000 0010	0000 0000
1	1: 0 -> no op	0110	0000 0010	0000 0000
	2: Multiplier shift right/ Multiplicand shift left	→ 011	0000 0100 ←	0000 0000
2	1: 1 -> product = product + multiplicand	011	0000 0100	0000 0100
	2: Multiplier shift right/ Multiplicand shift left	→ 01	0000 1000 ←	0000 0100
3	1: 1 -> product = product + multiplicand	01	0000 1000	0000 1100
	2: Multiplier shift right/ Multiplicand shift left	→ 0	0001 0000 ←	0000 1100
4	1: 0 -> no op	0	0001 0000	0000 1100
	2: Multiplier shift right/ Multiplicand shift left		0010 0000	

Deficiențele implementării

- UAL-ul este de două ori mai lat decât trebuie
- Registrul care conține deînmulțitul (multiplicand) este de două ori mai lat decât e nevoie
- Registrul ce conține produsul nu are nevoie de 2n biți decât la ultimul pas
- Registrul înmulțitorului (multiplier) este golit în acest proces

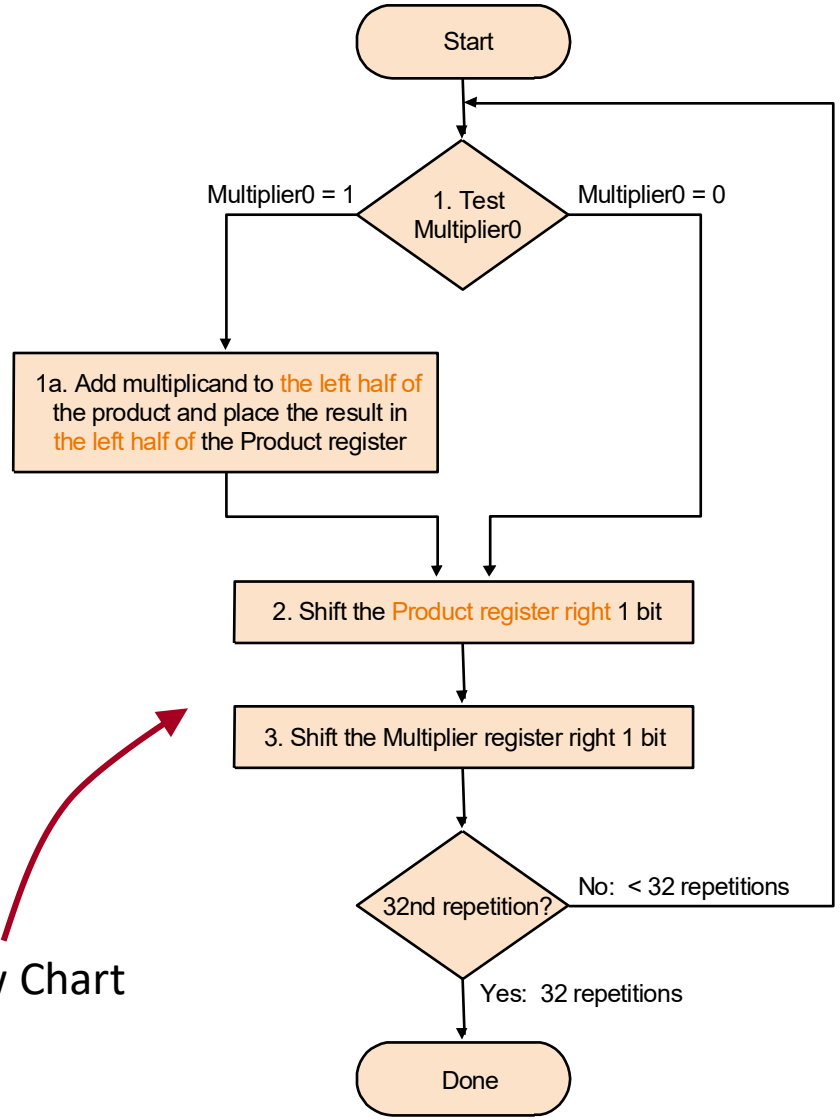
A doua implementare



Implementare hardware mai eficientă:

- **Multiplicand staționar**
- **Multiplier și PP shift dreapta**

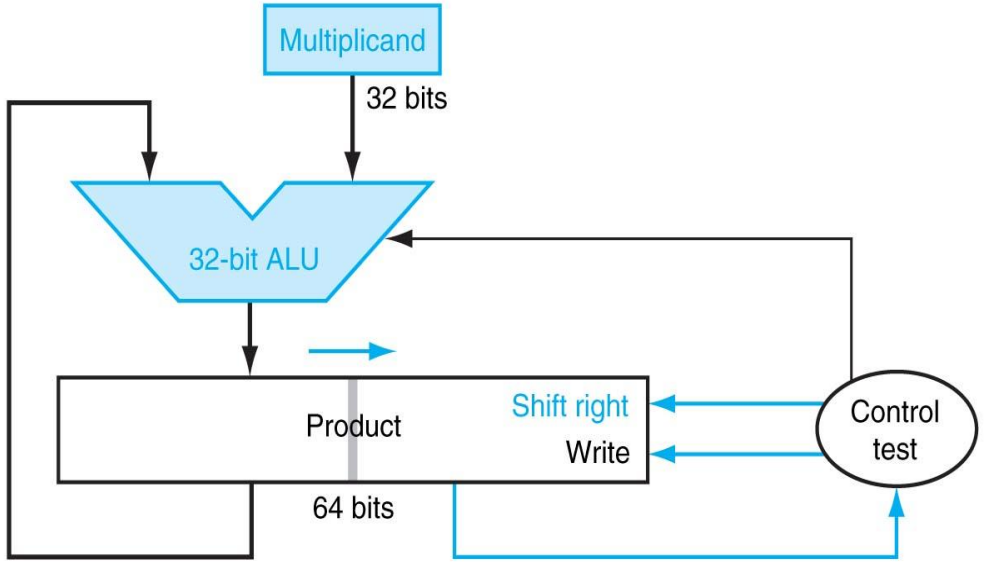
Flow Chart



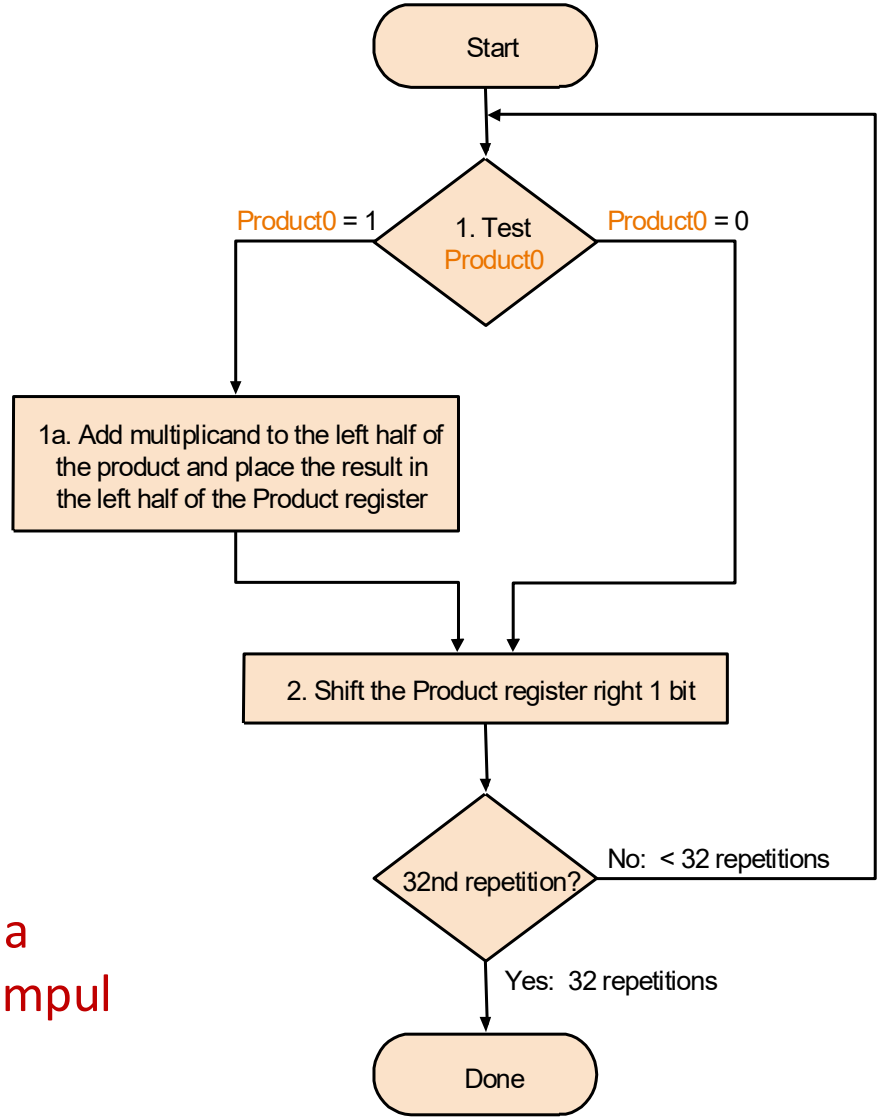
Exemplu

Iterație	Implementarea 2			
	Pas	Multiplier	Multiplicand	Produs
0	initial values	0110	0010	0000 $_{xxxx}$
1	1: 0 -> no op	0110	0010	0000 $_{xxxx}$
	2: Multiplier shift right/ Product shift right	$_{x}$011	0010	0000 0$_{xxx}$
2	1: 1 -> product = product + multiplicand	$_{x}$011	0010	0010 0$_{xxx}$
	2: Multiplier shift right/ Product shift right	$_{xx}$01	0010	0001 00$_{xx}$
3	1: 1 -> product = product + multiplicand	$_{xx}$01	0010	0011 00$_{xx}$
	2: Multiplier shift right/ Product shift right	$_{xxx}$0	0010	0001 100$_{x}$
4	1: 0 -> no op	$_{xxx}$0	0010	0001 100$_{x}$
	2: Multiplier shift right/ Product shift right	xxxx	0010	0000 1100

A treia implementare



O implementare hardware și mai eficientă



Avantajul este să folosim jumătatea de jos a produsului pentru a stoca înmulțitorul în timpul operației.

Exemplu

Iterația	Implementarea 3			
	Pasul	Multiplier	Multiplicand	Product|Multiplier
0	Valori inițiale	0110	0010	0000 0110
1	1: 0 -> no op	0110	0010	0000 0110
	2: Multiplier shift right/ Product shift right	×011	0010	0000 0011
2	1: 1 -> product = product + multiplicand	×011	0010	0010 0011
	2: Multiplier shift right/ Product shift right	××01	0010	0001 0001
3	1: 1 -> product = product + multiplicand	××01	0010	0011 0001
	2: Multiplier shift right/ Product shift right	××00	0010	0001 1000
4	1: 0 -> no op	×××0	0010	0001 1000
	2: Multiplier shift right/ Product shift right	××××	0010	0000 1100

Codificarea lui Booth

- Ne întoarcem la clasa a treia
  - 123 x 9 = 123 x (10 - 1) = 1230 - 123
    - Înmulțim cu 10 (shift la stânga o poziție)
    - Scădem o dată
    - Mai rapid decât să faci înmulțirea direct cu 9!
- Algoritmul lui Booth aplică același principiu în binar (deci, este și mai ușor!)
  - 01110 x n = n << 1 + n << 2 + n << 3
    - trei shiftări și două adunări
  - Sau: 01110 = 2^4-2, deci 01110 x n = n << 4 – n << 2
    - două shiftări și o scădere

Algoritmul lui Booth

- Caută o secvență de 1 în operand
  - De ex. ‘0110’ are doi de 1 în mijloc
  - Înmulțirea cu ‘0110’ (6 în zecimal) e echivalentă cu înmulțirea cu 8 și scăderea de două ori, 6 x m = (8 – 2) x m = 8m – 2m
- Algoritmul este simplu: iterează de la dreapta la stânga și:
  - Scade operandul din produs la primul bit de 1
  - Adună operandul la produs după ultimul bit de 1
  - Nu face nimic pentru biții de 1 din mijlocul secvenței

Algoritmul lui Booth

Bit curent	Bitul de la dreapta	Explicație	Exemplu	Operație
1	0	Început secvență ‘1’	00001111000	Scădere
1	1	Mijlocul secvenței ‘1’	00001111000	Nimic
0	1	Sfârșit secvență ‘1’	00001111000	Adunare
0	0	Mijlocul secvenței ‘0’	00001111000	Nimic

Algoritmul lui Booth

Să calculăm M x Q = 0010 x 1101 (2 x -3)

Inițializare algoritm:

- A = 0000 Q = 1101 Q-1 = 0 M = 0010 -M = 1110

Step	Q0 Q-1	Operation	A before shift	Q before shift	Q-1 before shift	A after shift	Q after shift	Q-1 after shift
0	_	Initialize	0000	1101	0	_	_	_
1	10	A = A − M	1110	1101	0	1111	0110	1
2	01	A = A + M	0001	0110	1	0000	1011	0
3	10	A = A − M	1110	1011	0	1111	0101	1
4	11	No operation	1111	0101	1	1111	1010	1

Rezultatul este registrul A concatenat cu registrul Q, adică 11111010 = -6

Înmulțire binară standard

Funcționează bine pentru numere fără semn, dar pentru înmulțirea numerelor cu semn în complement față de 2 este nevoie de tratarea suplimentară a semnului.

Configurare brută shift-and-add:

```
            0010
  ×         1101
  ------------------
            0010
           0000
          0010
         0010
  ------------------
         11010
```

Această așezare tratează 1101 ca pe un model obișnuit de biți. Dar 1101 este de fapt -3 în complement față de 2, deci produsele parțiale de tip fără semn nu oferă direct rezultatul corect pentru numere cu semn.

Rezultatul corect cu semn:

```
2 × (-3) = -6 = 11111010
```

Algoritmul Booth

Suportă direct înmulțirea numerelor cu semn în complement față de 2, analizând perechea (Q0, Q-1) și alegând adunare, scădere sau nicio operație.

Inițializare:

```
M   = 0010
-M  = 1110
A   = 0000
Q   = 1101
Q-1 = 0
```

Deciziile Booth:

Pas	Q0Q-1	Acțiune	După deplasare
1	10	A = A - M	A=1111 Q=0110
2	01	A = A + M	A=0000 Q=1011
3	10	A = A - M	A=1111 Q=0101
4	11	Nicio op.	A=1111 Q=1010

Rezultatul final:

```
AQ = 11111010 = -6
```

Concluzie: pentru numere cu semn precum 0010 × 1101, înmulțirea binară obișnuită are nevoie de corecții suplimentare pentru semn, în timp ce algoritmul Booth produce sistematic rezultatul corect în complement față de 2.

Avantajele algoritmului lui Booth

- Funcționează la fel pentru numere pozitive și negative
- Mai puține adunări/scăderi decât înmulțirea obișuintă
- Comprimă execuția pentru secvențe lungi de 1
- Mai eficient de implementat hardware
- Stă la baza unor versiuni îmbunătățite
  - Radix-4 Booth

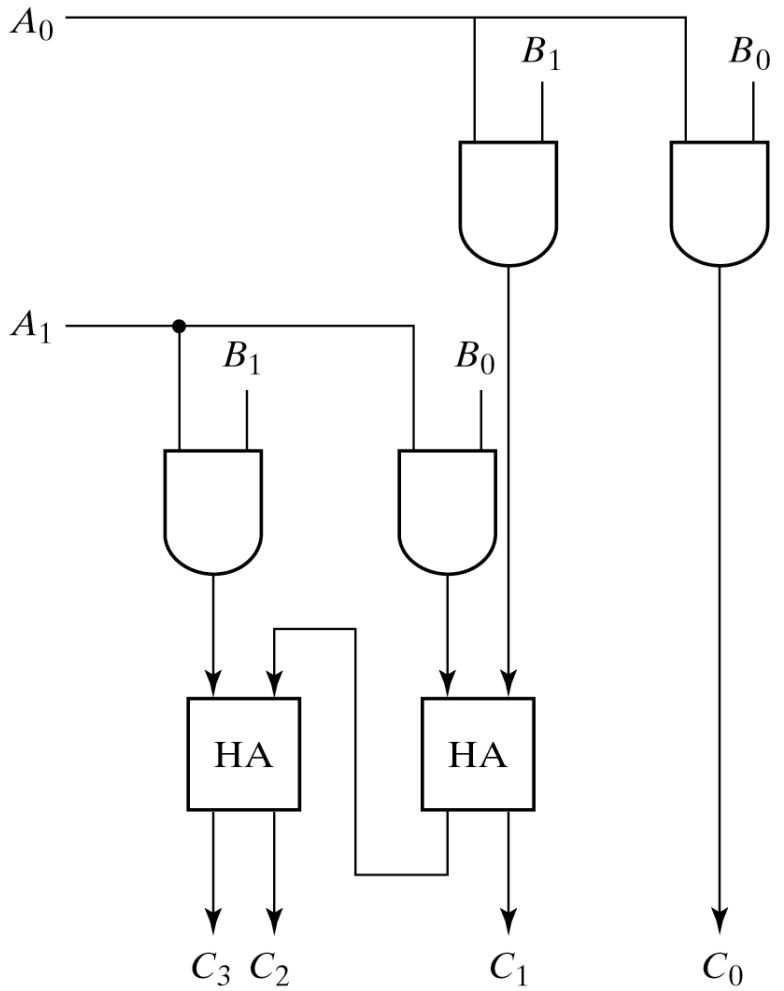
Putem face înmulțirea și mai rapid?

- În loc să iterăm în timp…
- Ce-ar fi să iterăm în spațiu?
  - Repetă circuitele în spațiu
    - Consumă mai multă suprafață
    - Dar face calcule mai rapid

Multiplicatorul combinațional

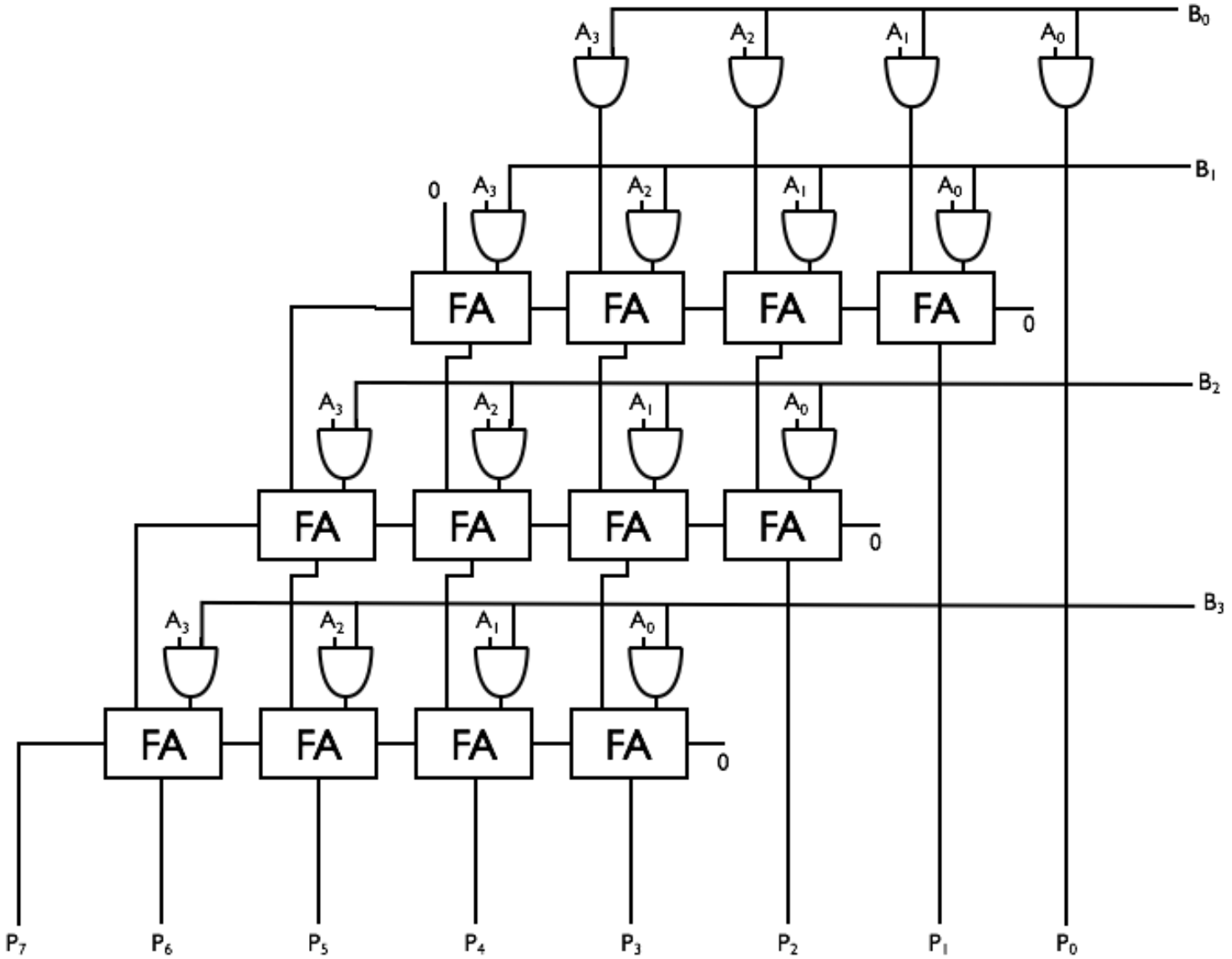
- Circuit pentru un multiplicator binar 2x2
- Implementare pur combinațională

$$
\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\hline
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}
$$



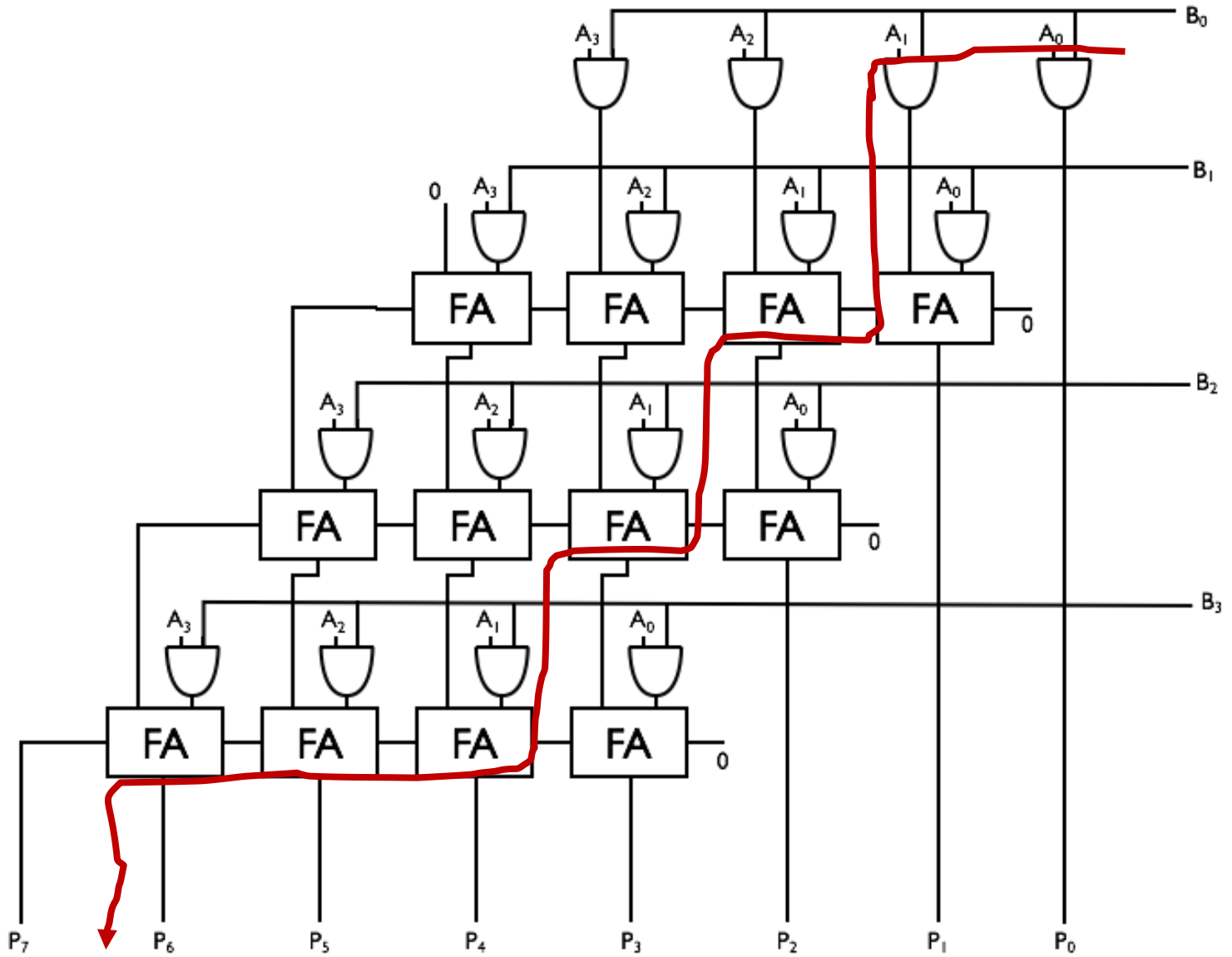
Multiplicatorul combinațional

- "Array Multiplier"
  - Repetăm în spațiu, în loc de timp
  - Componente folosite:
    - N*(N-1) sumatoare complete de 1 bit
    - N^2 porți AND



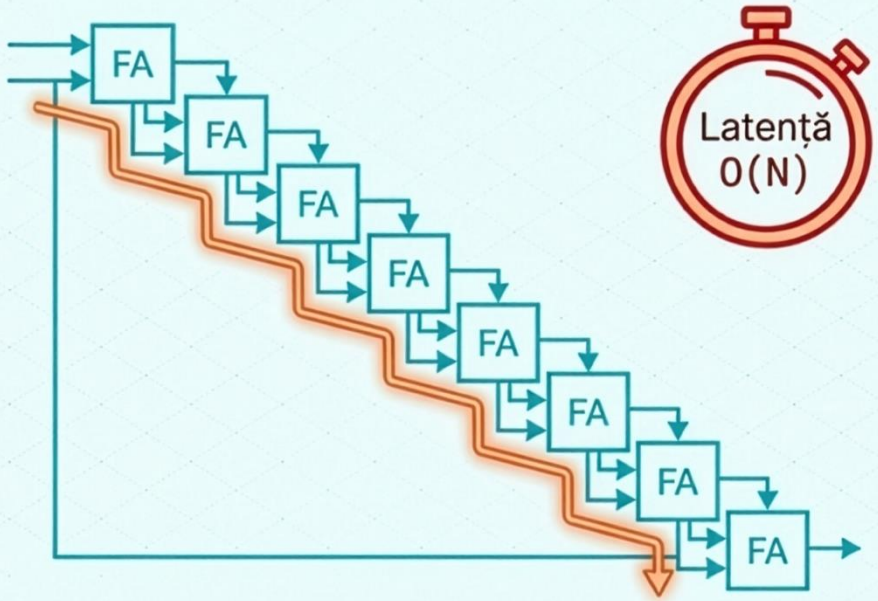
Multiplicatorul combinațional

- Calea critică
  - Întârziere proporțională cu N biți operanzi
  - ~ $3N*t_{pd,FA}$



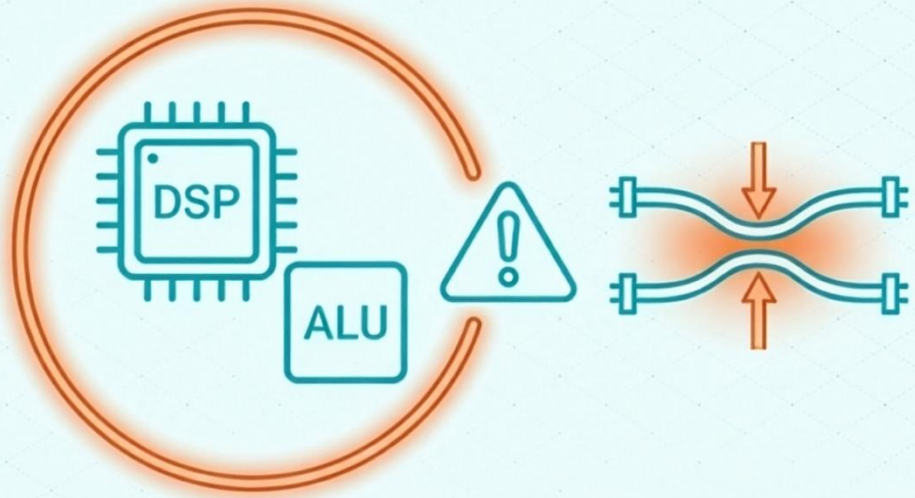
Problema arhitecturilor tradiționale de înmulțire

Problema: Înmulțirea Secvențială



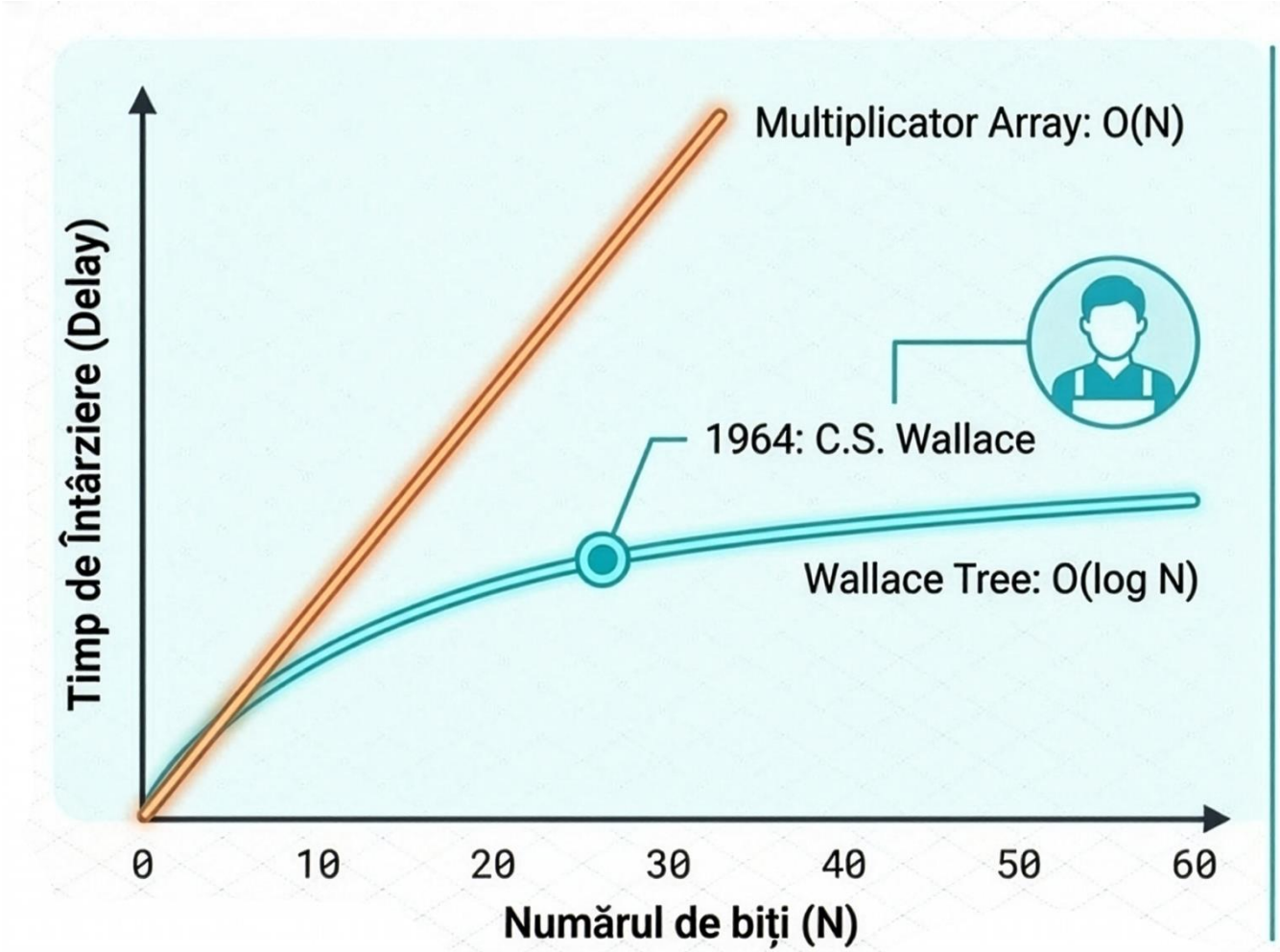
- **Crestere liniară a întârzierii:** Înmulțirea binară printr-o arhitectură Array propagă transportul rând cu rând.
- **Blocajul de performanță:** Frecvența maximă a procesorului este sever limitată de calea critică lungă.

Necesitatea: Calcul de Înaltă Performanță



- **Soluția necesară:** Sistemele moderne necesită o abordare care să paralelizeze suma produselor parțiale pentru a reduce drastic timpul de execuție.

Schimbarea de paradigmă



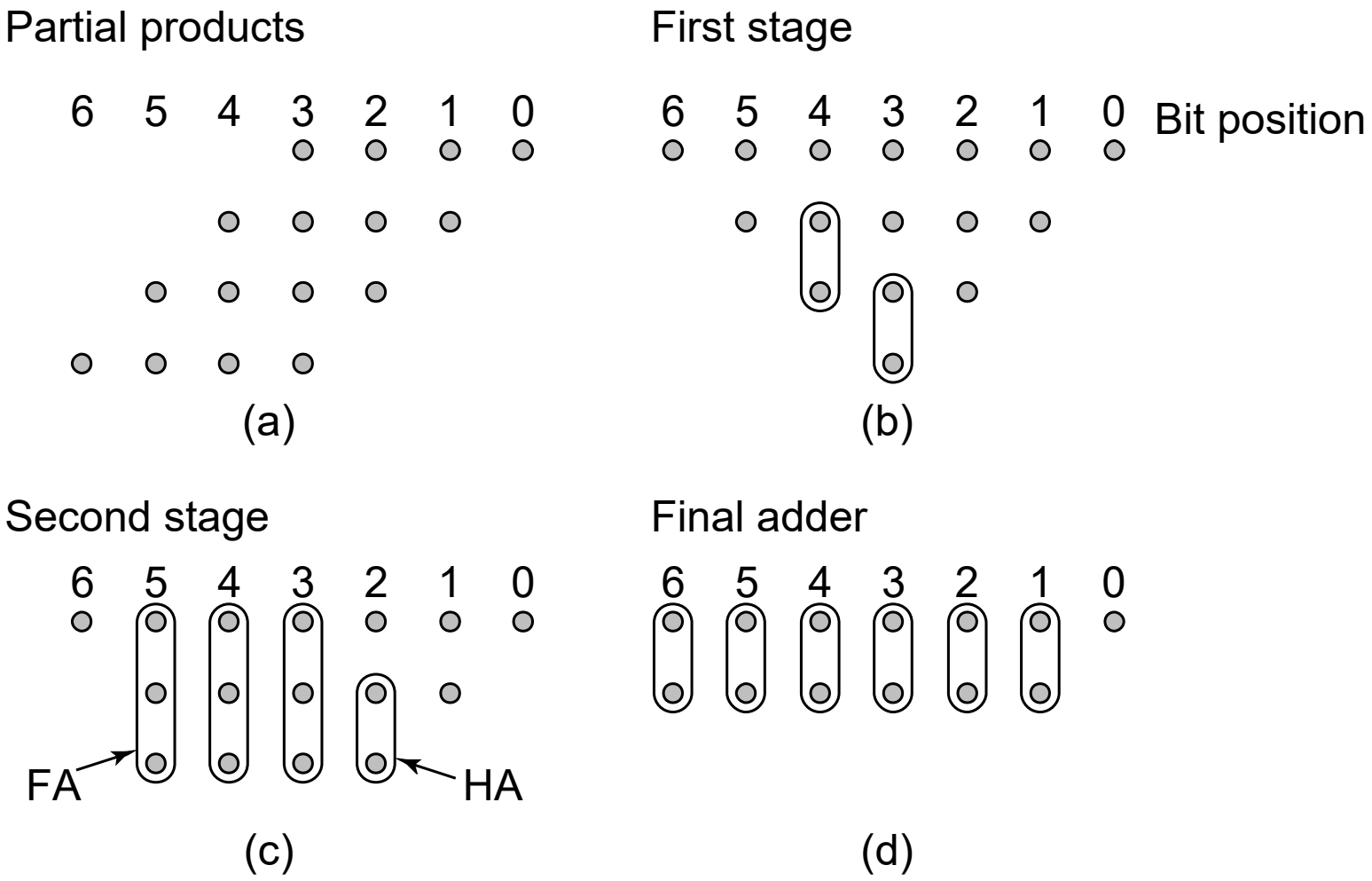
Inovația din 1964:
C.S. Wallace a revoluționat designul hardware abandonând complet adunarea secvențială tradițională.

Rețea de compresoare:
Utilizează o structură arborescentă de sumatoare (Carry-Save Adders) pentru a procesa toți biții în paralel, fără propagare orizontală.

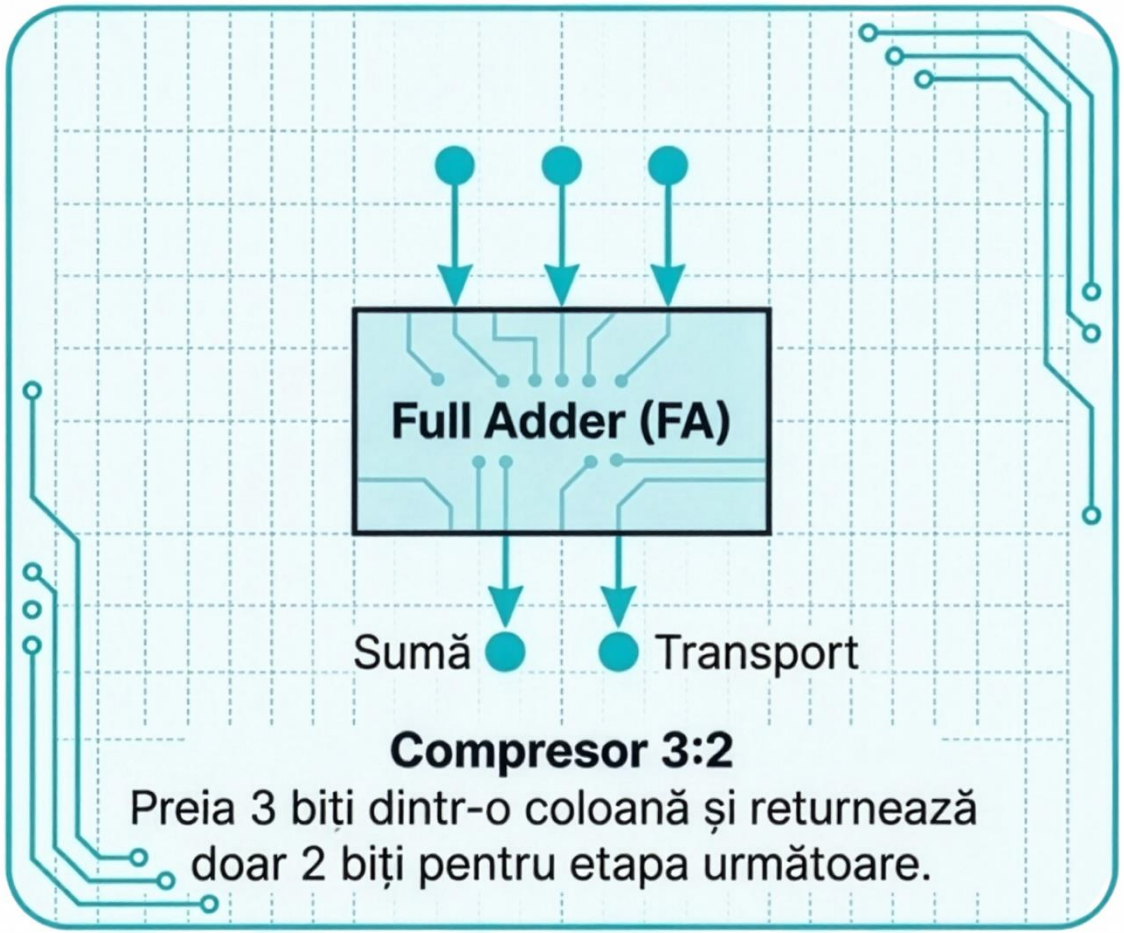
Performanță asimptotică:
Latența scade masiv, permițând multiplicări extrem de rapide pentru numere mari (ex: 32-bit, 64-bit).

C. S. Wallace, "A Suggestion for a Fast Multiplier," in *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 14-17, Feb. 1964

Multiplicator cu arbore Wallace

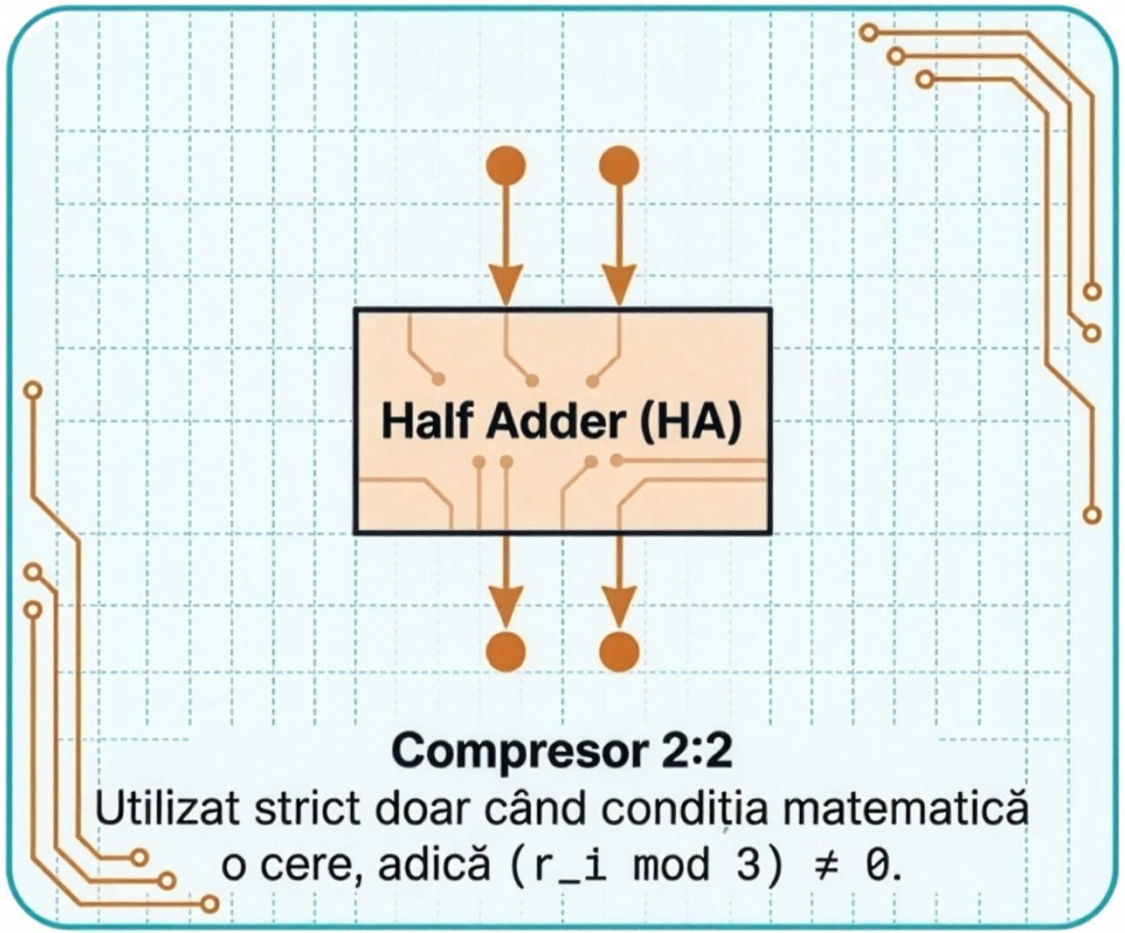


Motorul de compresie la nivel logic



Compresor 3:2

Preia 3 biți dintr-o coloană și returnează doar 2 biți pentru etapa următoare.



Compresor 2:2

Utilizat strict doar când condiția matematică o cere, adică $(r\_i \bmod 3) \neq 0$.

Multiplicatorul Wallace

Ideea-cheie: produsele parțiale nu sunt adunate pe rând, ci sunt reduse în paralel până rămân doar două rânduri, apoi se face o singură adunare finală.

- foarte rapid pentru înmulțiri hardware
- folosește sumatoare complete și pe jumătate
- reduce adâncimea logică față de un multiplicator de tip array
- este des combinat cu recodarea Booth și cu compresoare 4:2

Fluxul general

1. Generare produse parțiale

a_i AND b_j pentru fiecare pereche de biți

↓

2. Reducere Wallace

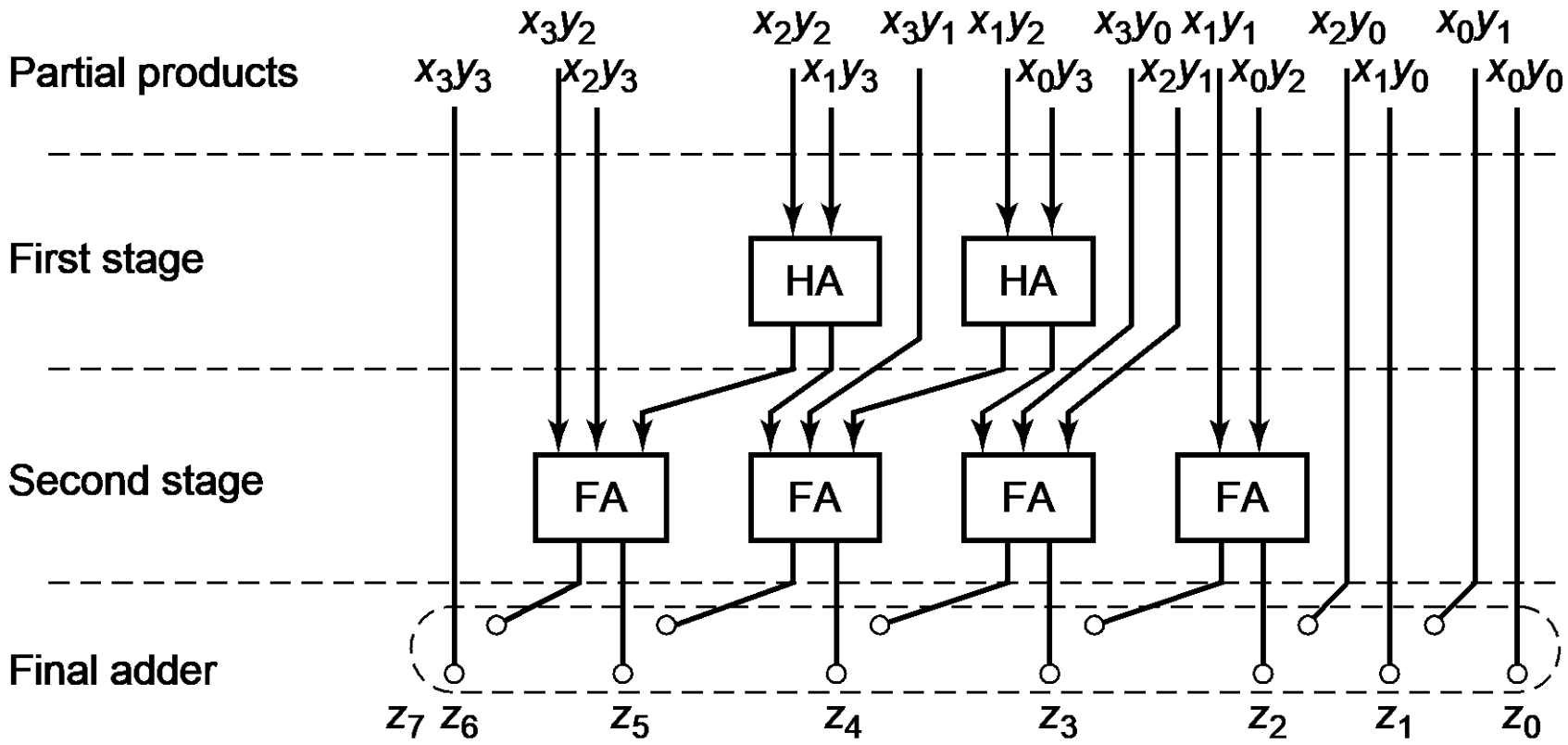
grupare pe coloane de aceeași pondere
și comprimare 3:2 / 2:2 în mai multe etape

↓

3. Adunare finală

cele două rânduri rămase → sumă finală

Multiplicator cu arbore Wallace



Full adder = (3,2) compressor

Structură generală

Generare produse parțiale

- n×n porți AND
- eventual recodare Booth înainte
- toți biții sunt disponibili în paralel

→

Arbore de reducere Wallace

- etape succesive de compresoare 3:2
- fiecare nivel micșorează înălțimea coloanelor
- nu este un „arbore” perfect; este o rețea de reducere

→

Sumator final

- carry-lookahead / carry-select / prefix
- primește ultimele două rânduri
- produce rezultatul pe 2n biți

Avantaj structural

multe operații se execută în paralel, deci întârzierea crește lent cu numărul de biți.

Provocare practică

rutarea este mai complicată; firele lungi și încărcarea capacitivă pot limita frecvența reală.

Optimizări comune

pipeline între niveluri, compresoare 3:2, combinare cu Booth pentru mai puține produse parțiale.

Performanță

Viteză: O(log N)

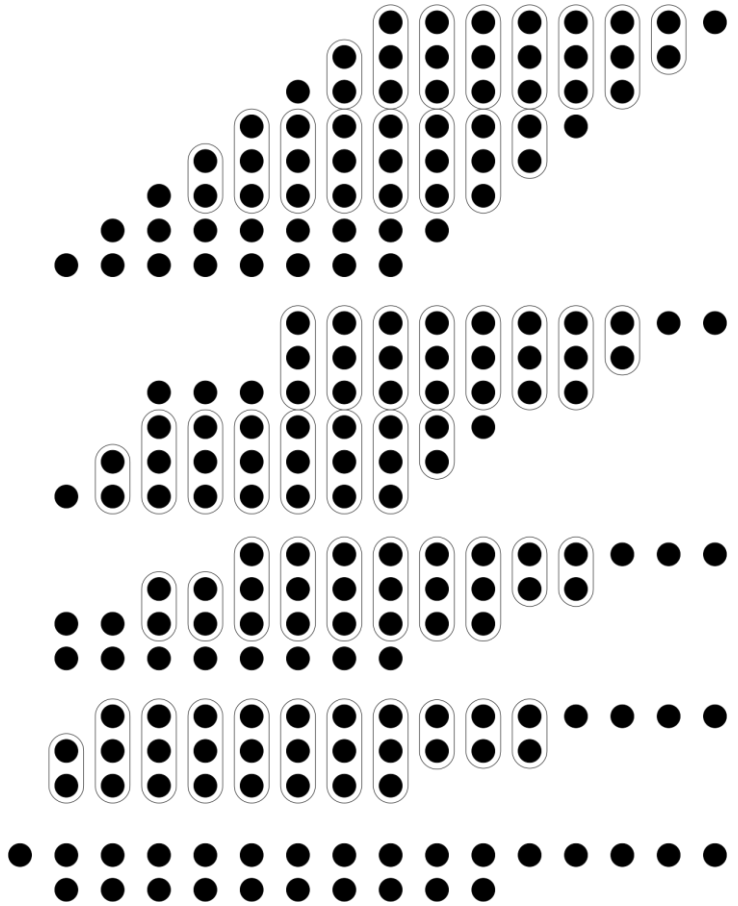
- Numărul de etaje crește lent odată cu creșterea mărimiipe biți a operanzilor

Spațiu: $O(N^2)$

- Complexitate mare, necesită un număr mare de tranzistoare

Rutare: Complexitate neregulată a conexiunilor

- Spre deosebire de array multiplier, Wallace tree multiplier e mai dificil de rutat



Reducere 4 straturi Wallace, pentru operanzi 8x8
64 porți AND, 15 Half-Adders, 38 Full-Adders

Wrap-up

Criteriu	Booth	Wallace Tree	Array Multiplier	Dadda Tree	Naiv secvențial
Delay / latență	~2.0–3.0 ns	~1.2–2.0 ns	~3.0–5.0 ns	~1.1–1.8 ns	8 ns (8 ciciuri la 1 GHz)
Cost / suprafață	mediu	ridicat	mediu-ridicat	mediu-ridicat	foarte redus
Consum de putere	mediu	ridicat	mediu-ridicat	mediu	redus
Throughput	1 rezultat / ciclu	1 rezultat / ciclu	1 rezultat / ciclu	1 rezultat / ciclu	125 Mmul/s
Frecvență maximă realistă	medie-ridicată	ridicată	moderată	ridicată	ridicată

Observație metodologică:

Nota: valoriie numerice sunt onentative pentru muitiplicatoare 8×8 si depind de tehnoiogia CMOS, bibiioteca standard cell, topologia adderuiul final, tensiunea de alimentare si strategia de impiementare

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