

OPERATII PE BIȚI

Adunare: $1 + 1 \rightarrow 0$ și carry 1

Scădere: $0 - 1 \rightarrow 1$ și carry -1

C:

$$\begin{array}{r} 0101110011110011 \\ + 0111000011110000 \\ \hline 1100110111100011 \end{array}$$

Înmulțire: Verific dacă se înmulțește cu o putere a lui 2 (2, 4, 8, 16).

Atunci shiftez la stânga (1, 2, 3, 4).

bit - binary digit

baza x - are cifre de la 0 la $x - 1$

Numere pozitive:

Numărul Y din baza X = $\sum_0^{n-1} b_i \cdot X^i$, n este numărul de biți din reprezentare

b_0 e LSB, b_{n-1} e MSB

Conversii:

- $B_{\text{old}} = 10$ are 10 cifre (0 - 9) $B \rightarrow B^p$ - Grupăm din B în câte P cifre, $p = \log_b B^P$
- $B_{\text{new}} = 100$ are 100 de cifre (0 - 99)

Numere negative (Pentru care, uneori, e nevoie de circuite speciale)

$/S/ ___$ - Se MSB, bit rezervat pentru semn, deci sunt disponibili $n - 1$ biți din reprezentare!

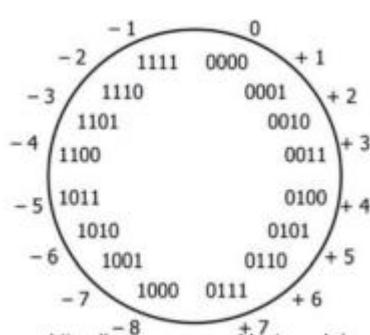
Numărul -Y din baza X = $-b_i \cdot 2^{n-1} + \sum_0^{n-2} b_i \cdot 2^i$

Complement față de doi:

- | | | | |
|-----------------------------|-----------------------------|-------------------|--------------|
| $(-x)_{b10} \rightarrow b2$ | 1. Scriem $ x $ în binar | 2. Inversăm biții | 3. Adăugăm 1 |
| $(-x)_{b2} \rightarrow b10$ | 1. Scriem numărul aşa cum e | 2. Inversăm biții | 3. Adăugăm 1 |

Folosim sistemul binar: Același algoritm de adunare, pot fi folosite aceleasi circuite pentru adunare naturală

- un exemplu explicit: $(4215)_{10} = (1000001110111)_2$



2 4215	— 1	← LSB
2 2107	— 1	
2 1053	— 1	
2 526	— 1	
2 263	— 0	
2 131	— 1	
2 65	— 1	
2 32	— 1	
2 16	— 0	
2 8	— 0	
2 4	— 0	
2 2	— 0	
2 1	— 0	
	0	← MSB

Cel mai mare număr care se poate reprezenta pe N biți: $2^N - 1$

Cel mai mare număr care se poate reprezenta (complement față de 2) pe N biți: $2^{N-1} - 1$

Cel mai mic: -2^{N-1}

MSB pe poziția: N – 1 (maximal în reprezentare)

X – Număr natural. Biți necesari pentru reprezentare: $\text{ceil}(\log_2 X)$

X – K cifre în HEX, de câți biți e nevoie pentru BIN: $K * \log_2 16 = 4$

X – K cifre în BIN, de câți biți e nevoie pentru HEX: $\text{ceil}(k / 4)$

X – K cifre în DEC, de câți biți e nevoie pentru BIN: $\text{ceil}(k * \log_2 10)$

HEX_{old} BIN_{new}

BIN_{old} HEX_{new}

DEC_{old} BIN_{new}

De ce funcționează complementul față de 2?

$$\begin{aligned}
 -\left(-2^N + \sum_{i=0}^{N-1} b_i 2^i\right) &= 2^N - \sum_{i=0}^{N-1} b_i 2^i \\
 2^{N+1} = \sum_{i=0}^N 2^i + 1 &= \sum_{i=0}^{N-1} 2^i + 1 - \sum_{i=0}^{N-1} b_i 2^i \\
 &= \sum_{i=0}^{N-1} (1 - b_i) 2^i + 1 \\
 &= (\text{inversam bitii}) + 1
 \end{aligned}$$

BINARY FIXED-POINT

...	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	...
-----	-------	-------	-------	-------	-------	-------	-------	-------	--	----------	----------	----------	----------	----------	----------	----------	-----

2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625
2^{-5}	0.03125
2^{-6}	0.015625
2^{-7}	0.0078125
2^{-8}	0.00390625
2^{-9}	0.001953125
2^{-10}	0.0009765625
2^{-11}	0.00048828125
2^{-12}	0.000244140625
2^{-13}	0.0001220703125

Ex: 101.101

$$\begin{cases} 101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 4 + 1 = 5 \\ 101 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.5 + 0.125 = 0.625 \end{cases}$$

$\Rightarrow 101.\underline{101} = 5.625$
 \Rightarrow

Ex: 3.75

$$\begin{cases} 3 = 1 \cdot 2^1 + 1 \cdot 2^0 \\ 0.75 = 1 \cdot 2^{-1} + 1 \cdot 2^{-2} \end{cases}$$

$$\Rightarrow \underline{3.75} = 11.11$$

LOGARITM ÎNTREG

9. Demonstrați că $\lfloor \log_2 x \rfloor = i_{\max}$ unde x este un număr dat pe N biți iar $i_{\max} = \max \{i \mid b_i = 1, \forall i = 0, \dots, N-1\}$ unde b_i reprezintă al i -lea bit din reprezentarea binară a numărului x . De exemplu: dacă $x = 00101110$ (46 zecimal) atunci $\lfloor \log_2 x \rfloor = 5$.

- **arătați că** $\lfloor \log_2 x \rfloor = i_{\max}$
- **pornim de la reprezentarea binară și aplicăm logaritmul**

$$\begin{aligned} x &= \sum_{i=0}^{N-1} b_i 2^i \\ \log_2 x &= \log_2 \left(\sum_{i=0}^{N-1} b_i 2^i \right) \\ &= \log_2 \left(2^{i_{\max}} \left(\sum_{i=0}^{N-1} b_i \frac{2^i}{2^{i_{\max}}} \right) \right) \\ &= \log_2 2^{i_{\max}} + \log_2 \left(\left(\sum_{i=0}^{N-1} b_i \frac{2^i}{2^{i_{\max}}} \right) \right) \\ &= i_{\max} + C, \quad C < 1 \end{aligned}$$

OVERFLOW-SAFE BINARY-SEARCH

```

int binarySearch1(int arr[], int start, int end, int x)
{
    if (end >= start)
    {
        int mid = start + (end - start) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch1(arr, start, mid - 1, x);

        return binarySearch1(arr, mid + 1, end, x);
    }

    return -1;
}

```

GENERAL / ISTORIC

ENIAC – 1945 – 1955 / USA – 1000 op/s (aka Electronic Numerical Integrator and Computer)

HPE CRAY – 2021 / USA – 1714 petaoperații/s

“compute” – Infrastructura hardware pe care rulează algoritmii

Putem scădea costul de calcul pentru ML prin: algoritmi eficienți, hardware dedicat

Varianta A – Cea mai rapidă
(Calculul a două matrice)

Complexitate: $O(n^3)$

Operații elementare: Adunare, înmulțire

Numărul operațiilor elementare: 2 pentru fiecare for (pe lângă asta, în fiecare for sunt câte n operații elementare de incrementare)

Număr de operații: $2 * n^3$

```

# varianta A
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
        for (int k = 0; k < n; ++k)
            C[i][j] += A[i][k] * B[k][j];

```

PERSONALITĂȚI

Richard Hamming - “The purpose of computing is insight, not numbers.”. Codul Hamming pentru integritatea mesajelor transmise la distanțe mari.

Blaise Pascal – Creează Pascaline (1642) – Calculator mecanic, capabil de +-, jucăria aristocrației | Limbajul Pascal e numit în onoarea lui

Gottfried Wilhelm Von Leibniz – Studiază sistemul binar, extinde mașina lui Pascal (adaugă * /)

George Boole – Scrie "The Laws of Thought" | Introduce logica booleană și analizează operațiile logice de bază: Negație / Conjunction / Disjunction / Disjunction exclusivă | Ele stau la baza teoriei informației

Charles Babbage – Proiectează Mașina Diferențială Nr. 2 – prima mașină de calcul mecanică programabilă – prototipuri de 13 tone – Tatăl calculatoarelor moderne

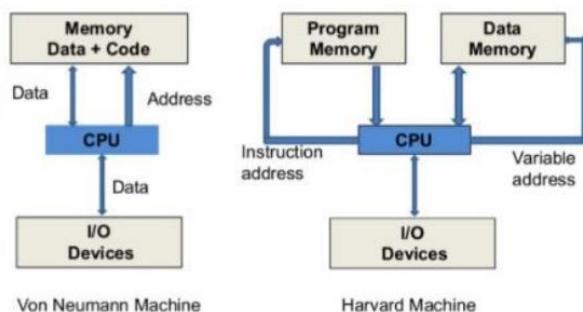
Ada Lovelace – Colaborează cu Babbage | Scrie un program care calculează numerele numere Bernoulli – Nu existau limbaje de programare, dar scrie pași pentru execuție – Primul programator

Konrad Zuse – Introduce o serie de calculatoare: Z1, Z2, Z3, Z4 | Folosește sistemul binar | Instrucțiuni stocate pe o bandă | Introduce reprezentarea și calculul în virgulă mobilă / floating point | Aproape totul în izolare, pe perioada celui de-Al II-lea Război Mondial.

Alan Turing – Decriptează rapid mesaje Enigma folosind mașina The Bomb - Brute-Force search | Introduce Mașina Turing pentru algoritmi Turing Complete – Sistem de analiză și recunoaștere a unui număr mare de date | Introduce Testul Turing – Pot mașinile să gândească?

John Von Neumann – Contribuie la crearea primului calculator electronic: ENIAC | Creează EDVAC – Sistem binar cu programe stocate. Introduce arhitectura von Neumann

Arhitectura von Neumann:



Claude Shannon: Părintele teoriei informației (o inventează literally) | Demonstrează că problemele de logică booleană se pot rezolva cu circuite electronice | Teorema de eșantionare Shannon – Nyquist: Analog <-> Digital fără a pierde date

Grace Hopper – Contribuitoare în dezvoltarea de limbaje high-level și scrie compilatorul COBOL

Margaret Hamilton - Scrie cod pentru software-ul de la bord pentru misiunea Apollo

Barbara Liskov – Design pattern: Prințipiu substituției Liskov + altele în distributed computing

Rivest Shamir Adleman – Creatorii primului sistem de criptare cu chei, public: RSA.
Foloeşte numere prime. Foarte util în tranzacţii bancare.

Diffie Hellman – Schimbul de cheie Diffie-Hellman – Soluţia în a trimite mesaje secrete într-un canal de comunicaţie nesecurizat.

Ritchie and Thompson – Dezvoltatorii limbajului C | Pun bazele sistemelor open-source, creează Unix OS.

Richard Stallman – Contributori la GNU Project.

Linus Tolvads – Creatorul Linux și GIT (pentru controlul versiunilor)

Steve Jobs – Cofondează Apple, Pixar

Bill Gates – Fondator Microsoft <3 | Acum CEO este Satya Nadella

Jeff Bezos – Fondator & fost CEO Amazon

Mark Zuckerberg – CEO Meta

Larry Page and Sergey Brin – Fondatorii Google

Idei mari:

- De la mecanic la electric
- De la o maşină care face un singur lucru la o maşină programabilă
- Design modular (pe module)
- Teorie despre ce este posibil pe aceste maşini
- Dorinţă de a face lucrurile optim, la limită şi fără risipă

După Al Doilea Război Mondial, dorinţa de cercetare în domeniul calculatoarelor creşte în ritm exponențial

Actori importanţi: Grupuri profesionale (IEEE, ACM, Bell Labs) și state (Statele Unite: DARPA)

TEORIA PROBABILITĂȚILOR

$P = \frac{\text{Numărul cazurilor favorabile}}{\text{Numărul cazurilor posibile}}$ (atunci când evenimentul nu este influențat de mediul exterior)

$P_1 \cdot P_2$ (probabilitatea ca două evenimente **INDEPENDENTE** să se întâpte, fără să fie influențate de mediul exterior) Este intersecția dintre evenimentul care se putea întâmpla primul și cele totale. Trebuie să ținem cont de cazurile care sunt favorabile pentru ambele evenimente.

TEORIA INFORMAȚIEI

Informația: Date care afectează (aproape mereu reduce) incertitudinea despre un fenomen.
Se poate acumula, este constantă (nu este creată/distrusă).

$I(x_i) = \log_2 \left(\frac{1}{p_i} \right)$ (Câtă informație ne oferă o valoare în funcție de probabilitatea ca ea să apară)

Calculăm cantitatea de informație folosind X (variabila aleatoare):

N – Numărul de valori distincte: x_1, x_2, \dots, x_n

P - Fiecare valoare apare cu probabilitatea p_1, p_2, \dots, p_n

p_i mai mic \rightarrow Obținem cantitate mai mare de informație

Putem privi formula astfel: $I(x_i) = \log_2 \left(\frac{1}{M \frac{1}{N}} \right) = \log_2 \left(\frac{N}{M} \right) =$ rezultat în biți!

N – Numărul total de evenimente

M – Numărul de evenimente favorabile

- **entropia**

- valoarea medie de informație primită despre o variabilă X

$$H(X) = E(I(X)) = \sum_{i=1}^N p_i \log_2 \frac{1}{p_i} = \sum_{i=1}^N -p_i \log_2 p_i$$

- $H(X)$ se numește entropia lui X
 - $I(X)$ este informația despre X
 - E este “expected value”, operația care calculează valoarea medie
 - exemplu: $X = \{A, B, C, D\}$ cu probabilități $\{1/3, 1/2, 1/12, 1/12\}$

$$H(X) = -\frac{1}{3} \log_2 \frac{1}{3} - \frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{12} \log_2 \frac{1}{12} - \frac{1}{12} \log_2 \frac{1}{12} = 1.626 \text{ biti}$$

I) 12 piese: 3 cuburi, 3 conuri, 3 sfere, 3 piramide | Fiecare formă are una dintre culorile: roșu, galben, albastru

A extrage o piesă roșie. Câtă informație primește copilul B, care vrea să afle tipul piesei?

Înțial: 12 variante, dar 4 piese sunt de culoare roșie, deci $p_i = 4 \cdot \frac{1}{12}$

$$I(x_i) = \log_2 \left(\frac{1}{4^{-\frac{1}{3}}} \right) = \log_2(3) \quad (\text{x}_i \text{ este o piesă roșie})$$

$$I(\text{bila albastra}) = \log_2 \left(\frac{1}{\frac{3}{8}} \right) = \log_2 \left(\frac{8}{3} \right) = 1.42 \text{ biti}$$

II) În urnă: 5 bile roșii, 3 bile albastre

Se extrage o bilă albastră.

Primim I(bilă albastră) cantitate de informație

$$H(\text{urna}) = \frac{5}{8} \log_2 \left(\frac{8}{5} \right) + \frac{3}{8} \log_2 \left(\frac{8}{3} \right) = 0.95 \text{ biti}$$

$$H(\text{urna dupa extragere}) = \frac{5}{7} \log_2 \left(\frac{7}{5} \right) + \frac{2}{7} \log_2 \left(\frac{7}{2} \right) = 0.86 \text{ biti}$$

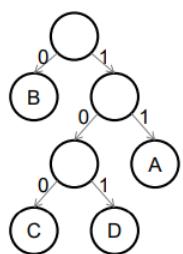
Entropia spune că: Nu se poate face o reprezentare pe biți mai optimă decât valoarea ei fără a pierde informație. (Ceva compresat perfect se numește zgomot)

Este limita de compresie posibilă!

Codarea eficientă și unică se face cu arbori binari:

Eruțele - Codurile

Stânga/Dreanta – Decis de 0/1



Implementare: Algoritmul lui Huffman

Implementare: Algoritmii lui Hammurabi
(Optim când considerăm un singur caracter pe rând)

Input: Probabilitatea fiecărui eveniment ex: {1/3 1/2 1/12 1/12}

Output: Codurile de pe un arbore binar

Cheia: O sedare mai scurtă pentru evenit, cu probabilități mari, una mai scurtă

//-- mici.

Dacă toate evenimentele sunt echiprobabile, nu putem face nimic.

00100111010011 0 0 100 11 101 0 0 11
 B B C A D B B A (Parcurgem arborele)

Detectarea erorilor: Ineficientă ca stocare.

Metoda I) Adăugăm biți de paritate simbolurilor, pentru a recupera informația.

Corecția erorilor: Distanță Hamming – O distanță mare poate duce și la corectarea erorilor
O distanță Hamming de $2N+1$ poate corecta E erori. Distanța hamming minimă pentru 4 biți schimbați este 4.

Orice comunicație / Stocare este redundantă!

Aproximarea lui Stirling: $\log_2(a!) \approx a \log_2(a) - a \log_2 e$

Probabilitățile pentru fiecare caz:

(În principiu : Cât știm / Cât vrem să știm)

a) x are exact două valori "1" în reprezentarea sa binară: $p = \frac{C_N^2}{2^N}$, pp. n par.

b) x are exact $N/2$ valori "1" în reprezentarea sa binară (N par): $p = \frac{C_N^{N/2}}{2^N}$, pp. n par.

c) x are o secvență continuă de $N/4$ biți de "1" (restul sunt "0", N divizibil la 4) :

$$p = \frac{(3N/4+1)}{2^N}, \text{ pp. } 4 \mid n$$

d) x are MSB (Most Significant Bit) setat la "1": $p = \frac{2^{N-1}}{2^N} = \frac{1}{2}$

e) x este impar: $p = \frac{2^{N-1}}{2^N} = \frac{1}{2}$

f) x este o putere a lui 2: $p = \frac{N}{2^N}$

g) x are primii $N/2$ biți din reprezentarea sa binară setați la "0" (N par): $p = \frac{2^{N/2}}{2^N}$

h) x este un număr prim (aproximativ estimat):

$$p \approx [2N/\ln(2N)]2N = 1\ln(2N) p \approx \frac{[2^N/\ln(2^N)]}{2^N} = \frac{1}{\ln(2^N)}$$

i) x are un număr par de biți setați la "1" (N par): $p = \frac{1+\sum_{i=1}^{N/2} C_N^{2i}}{2^N} = \frac{1}{2}$

j) $x = 42$: $p = \frac{1}{2^N}$

ASSEMBLY x86

Utilizări: Securitate informatică | Hacking, Reverse Engineering | Optimizare: Dezvoltare jocuri și ML / AI | Debugging | Dezvoltare software low-level: Sisteme embedded / Sisteme de operare

În Assembly x86, rezultatul înmulțirii poate fi stocat pe 64 de biți ($\%edx * 2^{32} + \%eax$)

Shift: $|S|XXX.XXXX >> 2 = 00|S|X.XXXX$ – Se pierde bitul de semn!
SAR/SAL – Shift aritmetic, ține cont de semn.

jmp *%eax – Sare la o adresă

aritate 1 – Operația se aplică unui singur număr

.asciz -> Lungimea sirului + 1 pentru \n

S1: "ASC", S2: "FMI" -> syscall de print pentru \$S1 cu lungimea 5 -> ASCFM

\neg NOT (are aritate 1) \wedge AND \vee OR Tsunotshi DX – eu irl | ORNOTAND

CMP – Compară două valori și setează flag-uri:

- Zero Flag (ZF) $op1 = op2$
- Sign Flag (SF) Rezultatul comparației este negativ
- Carry Flag (CF) $op1 < op2$

$$a(b, c, d) = a + b + c \cdot d$$

operand	Descriere	Semn (Da/Nu)	Flaguri setate	Condiție Flag
jc	jump dacă este carry setat	Nu	CF (Carry Flag)	CF = 1
jnc	jump dacă nu este carry setat	Nu	CF	CF = 0
jo	jump dacă este overflow setat	Nu	OF (Overflow Flag)	OF = 1
jno	jump dacă nu este overflow setat	Nu	OF	OF = 0
jz	jump dacă este zero setat	Nu	ZF (Zero Flag)	ZF = 1
jnz	jump dacă nu este zero setat	Nu	ZF	ZF = 0
js	jump dacă este sign setat	Da	SF (Sign Flag)	SF = 1
jns	jump dacă nu este sign setat	Da	SF	SF = 0
jb	jump if below (unsigned $op2 < op1$)	Nu	CF	CF = 1
jbe	jump if below or equal (unsigned $op2 \leq op1$)	Nu	CF, ZF	CF = 1 or ZF = 1
ja	jump if above (unsigned $op2 > op1$)	Nu	CF, ZF	CF = 0 and ZF = 0
jae	jump if above or equal (unsigned $op2 \geq op1$)	Nu	CF	CF = 0
jl	jump if less than (signed $op2 < op1$)	Da	SF, OF	SF \neq OF
jle	jump if less than or equal (signed $op2 \leq op1$)	Da	SF, OF, ZF	SF \neq OF or ZF = 1
jg	jump if greater than (signed $op2 > op1$)	Da	SF, OF, ZF	SF = OF and ZF = 0
jge	jump if greater than or equal (signed $op2 \geq op1$)	Da	SF, OF	SF = OF
je	jump if equal ($op1 = op2$)	Nu	ZF	ZF = 1
jne	jump if not equal ($op1 \neq op2$)	Nu	ZF	ZF = 0

TABELA 1. Operandi de salt condiționat, flagurile setate și condițiile lor

CIRCUITE

CIRCUIT DIGITAL COMBINAȚIONAL

La ieșire, este o combinație (funcție logică care combină) toate sau o parte a intrărilor

Eficiente în general. Problema majoră:

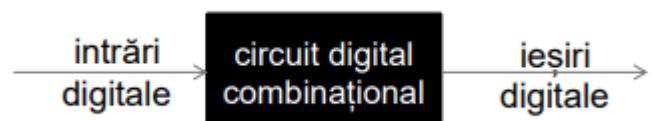
sunt one-shot

- Nu putem itera

- Nu permit niciun fel de logică internă / memorie internă (nu are stări interne)

- Sunt prea simple: Pui un semnal digital constant la intrare și ai un semnal digital constant la ieșire

- Logica combinațională este insuficientă pentru anumite implementări



Timp de propagare t_p : Timp maxim necesar pentru a produce la ieșire semnale digitale corecte și valide

Pentru fiecare intrare, trebuie să știm care e ieșirea

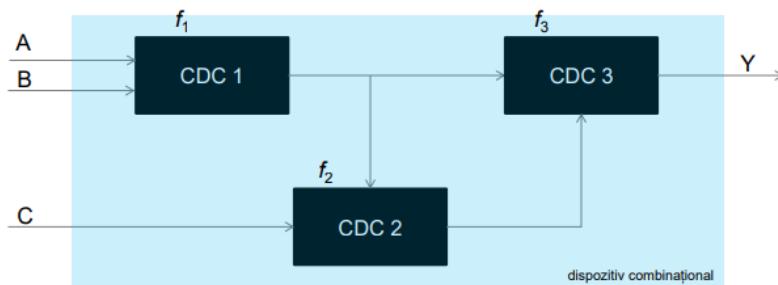
A	B	C	X	Y
0	0	0	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	0

Dispozitiv combinațional

Elementele sale: Circuite combinaționale

O intrare este conectată la exact o ieșire / la o constantă

Fără cicluri în graful direcțional al dispozitivului



Funcția dispozitivului: $Y = f_3(f_1(A, B), f_2(f_1(A, B), C))$ – Ne uităm ce intră în Y și cum.

Timpul total de propagare: $t_{p, total} = t_{p, 1} + t_{p, 2} + t_{p, 3}$ (cea mai lungă cale)

Timpul maxim după care avem o ieșire validă dacă avem intrări valide

Un computer care funcționează la 1GHz trimite comenzi o dată la 1ns

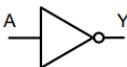
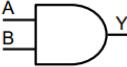
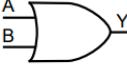
De ce folosim semnale digitale în loc de analogice:

- Într-un sistem analogic zgomotul se acumulează
- Într-un sistem digital, avem corecțiile de zgomot (avem margini)

De ce folosim sistemul binar?

- Număr redus de stări (2) (pentru HEX ar trebui 16, deci ar trebui distinse 16 nivele de voltaj)
- Pentru baza 4, am avea 4 nivele, deci am fi de 2 ori mai eficienți

Tot ce facem pe sistemul de calcul trebuie redus la circuite care sunt porti logice

Operația	Valori	Notăție															
NOT	 <table border="1" style="margin-left: 20px;"> <tr> <th>A</th> <th>Y</th> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </table>	A	Y	0	1	1	0	\bar{A} (A bar, A complement) sau $!A$									
A	Y																
0	1																
1	0																
AND	 <table border="1" style="margin-left: 20px;"> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	AB
A	B	Y															
0	0	0															
0	1	0															
1	0	0															
1	1	1															
OR	 <table border="1" style="margin-left: 20px;"> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1	$A + B$
A	B	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	1															
XOR	 <table border="1" style="margin-left: 20px;"> <tr> <th>A</th> <th>B</th> <th>Y</th> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </table> <p>$A \oplus B = \bar{A}B + A\bar{B}$</p>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0	$A \oplus B$
A	B	Y															
0	0	0															
0	1	1															
1	0	1															
1	1	0															

A — intrări → circuit digital combinational → ieșiri → x B — digitale → combinational → digitale → y		
A	B	C
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
X		
0		1
1		0
1		0
0		1
1		0
0		1
0		1
1		0
Y		
1		0
0		1
0		0
1		1
1		0
0		1
1		0

o expresie booleană care conține regulile din tabel?

- $X = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}\bar{C} + ABC$

poartă	întârziere (ps)	suprafață (μm^2)
AND-2	50	25
NAND-2	30	15
OR-2	55	26
NOR-2	35	16
AND-4	90	40
NAND-4	70	30
OR-4	100	42
NOR-4	80	32

N* - Mai rapide, mai mici ca suprafață față de *

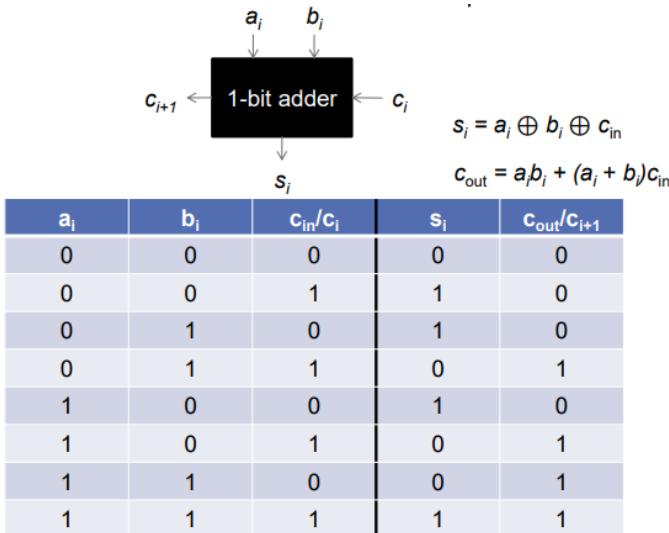
*-2 – Mai rapide, mai mici ca suprafață față de *-4

Tranzitoare CMOS: Circuitele analogice sunt mai eficiente pe logică negată (mai puține componente electrice)

CIRCUIT	INPUT	OUTPUT	NR INTRĂRI	NR IEȘIRI
ADUNARE	A (N biți) B (N biți)	S (N + 1 biți)	2N	N + 1

Cât de mare va fi circuitul? $(N + 1)2^{2N-1}$

Putem defini un circuit bloc fundamental pe care bazăm totul:



Folosim circuitul în cascadă

$$t_p = N \cdot t_{p,bit-adder}$$

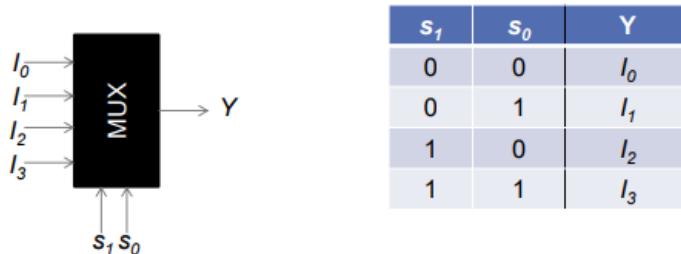
Trebuie așteptată calcularea biților de carry

Putem generaliza

- notăm $g_i = a_i b_i$ și $p_i = a_i + b_i$
 - dacă $g_i = 1$ atunci $c_{out} = 1$
 - dacă $g_i = 0$ și $p_i = 0$ atunci $c_{out} = 0$
 - dacă $g_i = 0$ și $p_i = 1$ atunci $c_{out} = c_{in}$

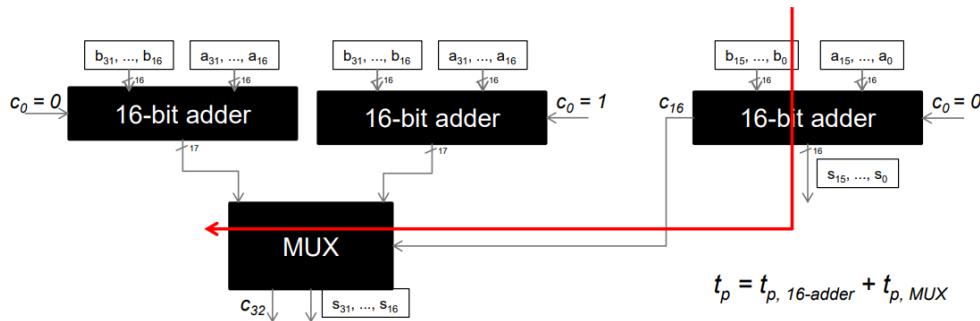
Multiplexor (MUX)

Circuitul selectează: un semnal digital de la intrare pe baza unui semnal de activare S
Util pentru implementare hardware pentru "if", "case", operații shift



Inputuri: $2^n \rightarrow n$ linii de selecție

Îmbunătățire: Adunare pe 32 de biți



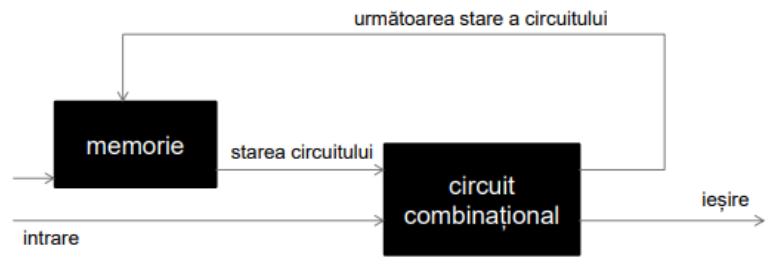
Putem aplica aceeași idee pentru circuitele de 16 biți de mai sus: $t_p = O(\log_2 N)$

CIRCUITE SECVENTIALE

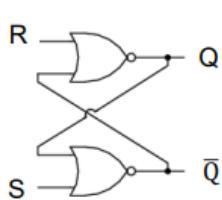
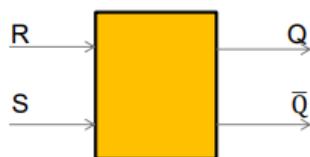
Permit elemente de tip "memorie" > putem adăuga o stare circuitului (există stare internă)

Au variabila de timp: Intrările / Ieșirile nu sunt fixe | Număr variabil de pași în rezolvare

Biții pe care îi reprezentăm - voltaj.
Energia electrică - dificil de stocat
(aproape fenomenul de scurgere / leakage)
Pentru a memora ceva > Refresh din când în când pentru actualizarea nivelului de energie electrică



SR Latch (Set-Reset Latch) | Memorează un bit de informație | Bun, dar are două intrări

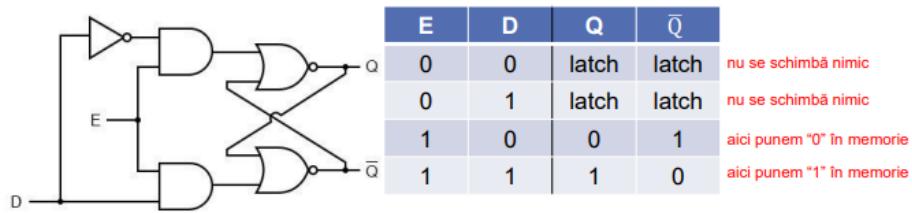
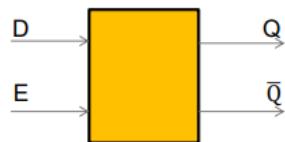


S	R	Q	\bar{Q}
0	0	latch	latch
0	1	0	1
1	0	1	0
1	1	0	0

nu se schimbă nimic
 aici punem "0" în memorie
 aici punem "1" în memorie
 stare invalidă

D Latch | Bun, are o intrare dar vrem să sincronizăm mai multe dispozitive

Se activează când E este activ. Vrem să se activeze când E crește.

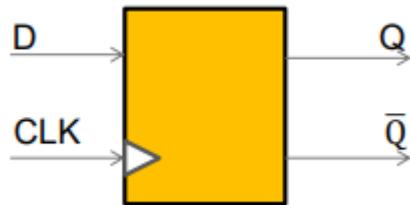


E de la Enable, adică activare
dacă E = 0 nu se întâmplă nimic

D Flip-Flop

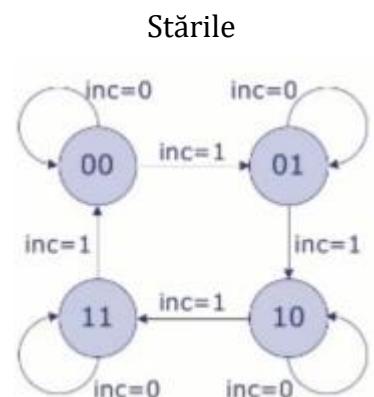
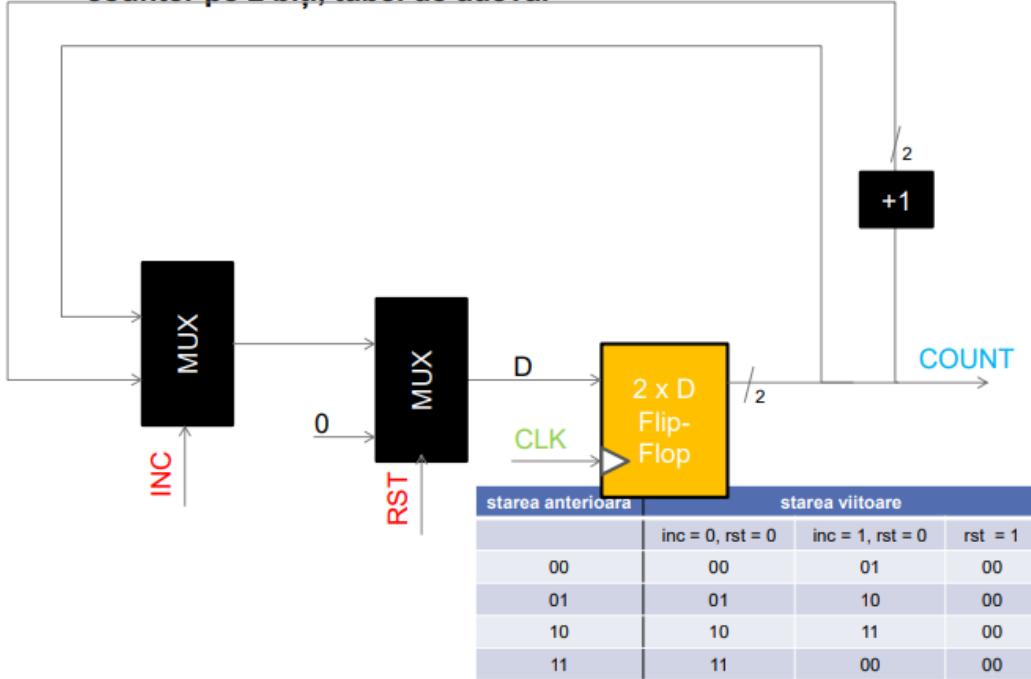
E devine clock (ceasul sistemului)

CLOCK: La un interval fix de timp (la un ciclu), sistemul face ceva



Registru = Un set de câteva D Flip Flops care au același CLK

- counter pe 2 biți, tabel de adevăr



LEGILE DE MORGAN

- $!(A+B) = A \cdot B$
- $!(A \cdot B) = A + B$
- $!(A+B+C) = A \cdot B \cdot C$
- $!(ABC) = A + B + C$
- $!(A+B) \cdot A \cdot B = A \cdot B$
- $!(AB)(A+B) = A + B$
- $!(A+B)(A+B) = A \cdot B$
- $A \cdot B \cdot (A+B) = A \cdot B$
- $C + !(C \cdot B) = 1$
- $!(AB)(A+B)(B+B) = A \cdot B$

SIMPLIFICĂRI

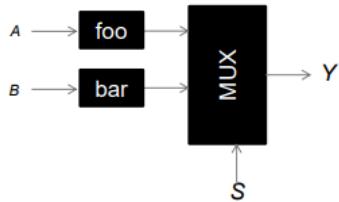
$$\begin{aligned}
 & (A + C)(AD + A\bar{D}) + AC + C \\
 & (A + C)A(D + \bar{D}) + AC + C \quad // \text{distribuim, invers} \\
 & (A + C)A + AC + C \quad // \text{suma variabila si complement} \\
 & A((A + C) + C) + C \quad // \text{distribuim, invers} \\
 & A(A + C) + C \quad // \text{asociem, idempotent} \\
 & AA + AC + C \quad // \text{distribuim} \\
 & A + (A + 1)C \quad // \text{idempotent, identitate, factor} \\
 & A + C \quad // \text{identitate de doua ori} \\
 \\
 & A+0 = A \qquad \qquad AB+AIB = A \\
 & !Ax0 = 0 \qquad \qquad !A+B!A = !A \\
 & A+A = 1 \qquad \qquad (D+!A+B+!C)B = B \\
 & A+A = A \qquad \qquad (A+IB)(A+B) = A \\
 & A+AB = A \qquad \qquad C(C+CD) = C \\
 & A+!AB = A+B \qquad \qquad A(A+AB) = A \\
 & A(!A+B) = AB \qquad \qquad !(A+!A) = A \\
 & AB+!AB = B \qquad \qquad !(A+!A) = 0 \\
 & (!AIB+!AB) = !A \qquad \qquad D+(D!CBA) = D \\
 & A(A+B+C+\dots) = A \qquad \qquad !D!(DBCA) = !D \\
 & \qquad \qquad \qquad AC+!AB+BC = AC+!AB \\
 & \qquad \qquad \qquad (A+C)(!A+B)(B+C) = AB+!AC \\
 & \qquad \qquad \qquad !A+!B+AB!C = !A+!B+!C \\
 & \qquad \qquad \qquad (A+B)^2+(A+B)^3+A+3!A+A^3 = 1 \\
 \\
 & A+A!A = A
 \end{aligned}$$

- implementați o poartă NOT cu un MUX: $Y = \text{NOT } A$
- $Y = I_0 \bar{s}_0 + I_1 s_0 = 1\bar{A} + 0A = \bar{A}$

- implementați o poartă AND cu un MUX: $Y = A \text{ AND } B$
- $Y = I_0 \bar{s}_0 + I_1 s_0 = 0\bar{A} + BA = AB$

- implementați o poartă OR cu un MUX: $Y = A \text{ OR } B$
- $Y = I_0 \bar{s}_0 + I_1 s_0 = B\bar{A} + 1A = A + B\bar{A} = A + B$ [vezi ex. 4 f)]

- $Y = S ? \text{foo}(A) : \text{bar}(B)$

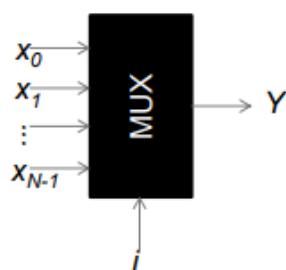


I_0 $\text{foo}(X)$	I_1 $\text{bar}(Y)$	S	Y
*	*	0	I_1
*	*	1	I_0

- care e diferența cu un limbaj de programare?

- indiferent de valoarea lui S, se execută $\text{foo}(A)$ și $\text{bar}(B)$
- doar că la ieșire vedem doar una dintre funcții (cea selectată de S)

Vrem să accesăm $x[i]$



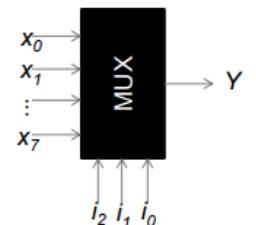
Intrări: Vectorul x

Semnal: Indexul i

Dimensiunea intrării: $N - 1 - 0 + 1 = N$

Dimensiunea lui S: $\text{ceil}(\log_2 N)$

$s_0(i)$	Y
000	$I_0(x_0)$
001	$I_1(x_1)$
010	$I_2(x_2)$
011	$I_3(x_3)$
100	$I_4(x_4)$
101	$I_5(x_5)$
110	$I_6(x_6)$
111	$I_7(x_7)$



Câte MUX cu 2 intrări pentru a simula MUX cu N intrări?

$$N - 1$$

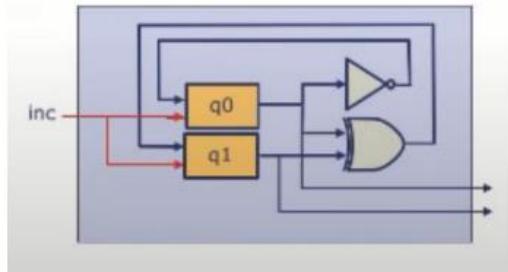
Circuit secvențial COUNTER pe 2 biți

Relația dintre starea anterioară și starea viitoare

$$\bullet q_0^{(t+1)} = !\text{INC} \times q_0^{(t)} + \text{INC} \times !q_0^{(t)}, q_1^{(t+1)} = !\text{INC} \times q_1^{(t)} + \text{INC} \times (q_1^{(t)} \otimes q_0^{(t)})$$

$q_1(t)$	$q_0(t)$	$q_1(t+1)$	$q_0(t+1)$
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

Desenul circuitului secvențial (stările sunt q_0 și q_1)



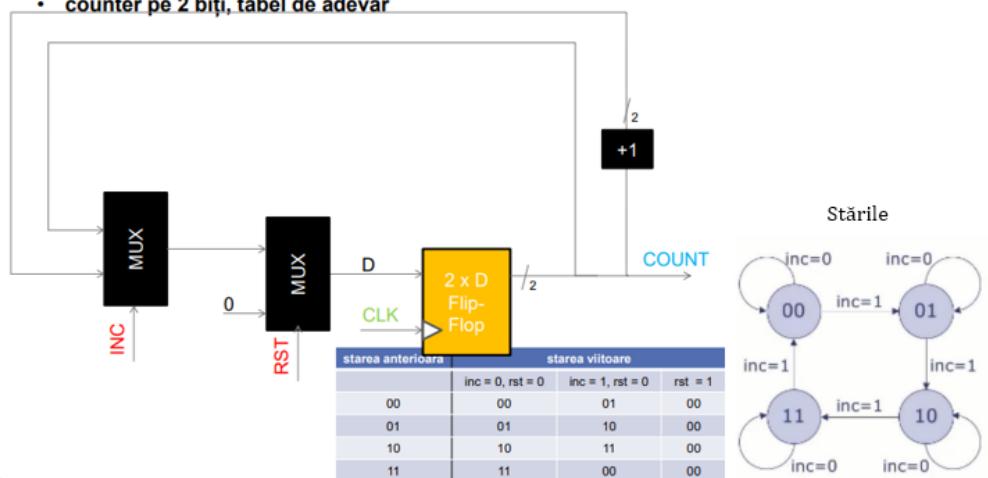
- aici INC e pe post de semnal Enable

c) Codul Gray – cod binar, proprietate: diferența de la un simbol la altul este un singur bit care se schimbă (Pentru 3 biți, codul Gray: 000, 001, 011, 010, 110, 111, 101, 100).

$q_2^{(t)}$	$q_1^{(t)}$	$q_0^{(t)}$	$q_2^{(t+1)}$	$q_1^{(t+1)}$	$q_0^{(t+1)}$
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	1	0
0	1	1	0	1	0
1	0	0	0	0	0
1	0	1	1	0	0
1	1	0	1	1	1
1	1	1	1	0	1

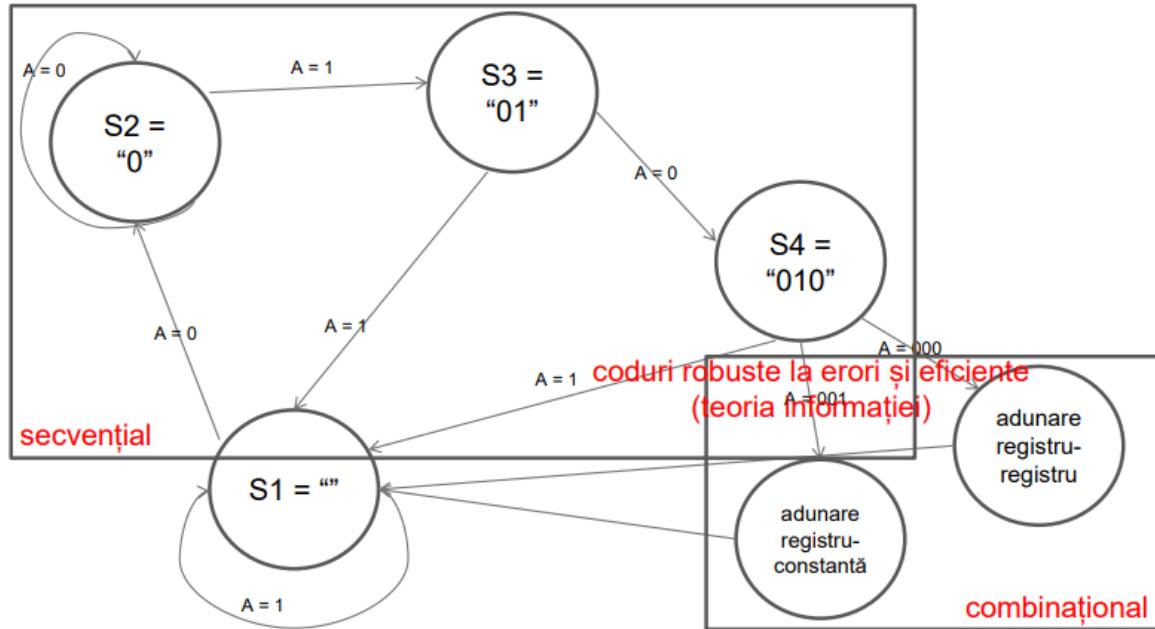
- $q_0^{(t+1)} = !q_2^{(t)}!q_1^{(t)}!q_0^{(t)} + !q_2^{(t)}!q_1^{(t)}q_0^{(t)} + q_2^{(t)}q_1^{(t)}!q_0^{(t)} + q_2^{(t)}q_1^{(t)}q_0^{(t)}$
 $= q_2^{(t)}q_1^{(t)} + !q_2^{(t)}!q_1^{(t)}$
- $q_1^{(t+1)} = \dots$
- $q_2^{(t+1)} = \dots$

• counter pe 2 biți, tabel de adevăr



Un semnal digital A valoarea 010 la un n
 Y este setată la 1, a

- desenați diagrame
- cum arată circ
- calculați tabelul
- scrieți expresii



cod mașină ...0100001011010001010000100011001011 ...

Circuit secvențial pentru CMMDC

```
def cmmdc(a, b): if a == b: return b elif a > b: return cmmdc(a-b, b) else: return cmmdc(b, a)
```

Avem variabilele: $a^{(t)}, b^{(t)}$

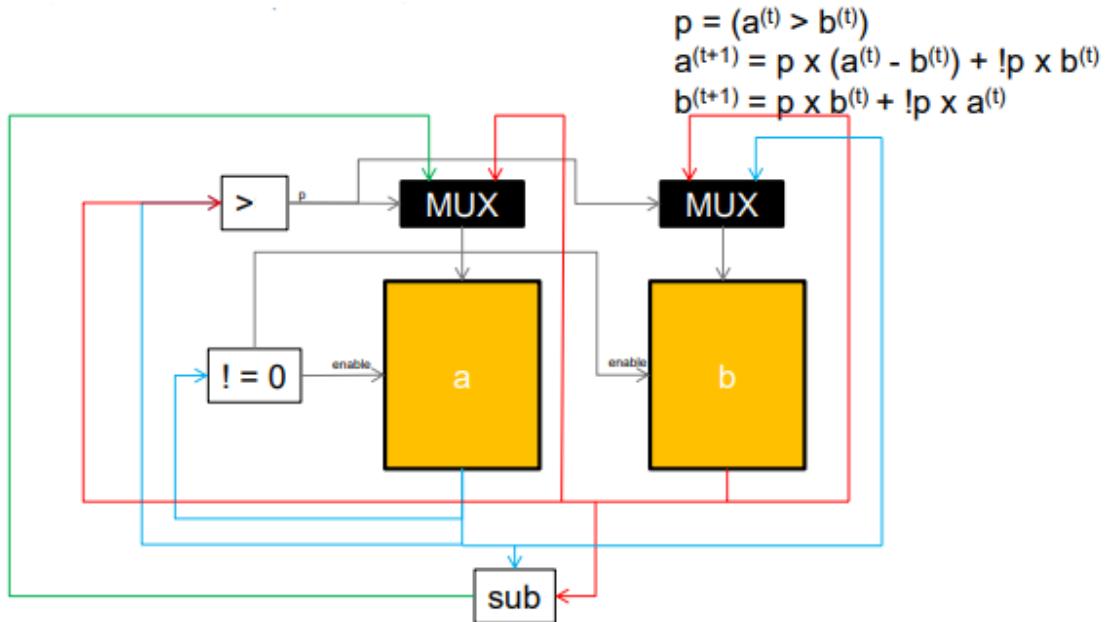
Ecuațiile de evoluție:

facem ceva doar dacă $a^{(t)} \neq 0$

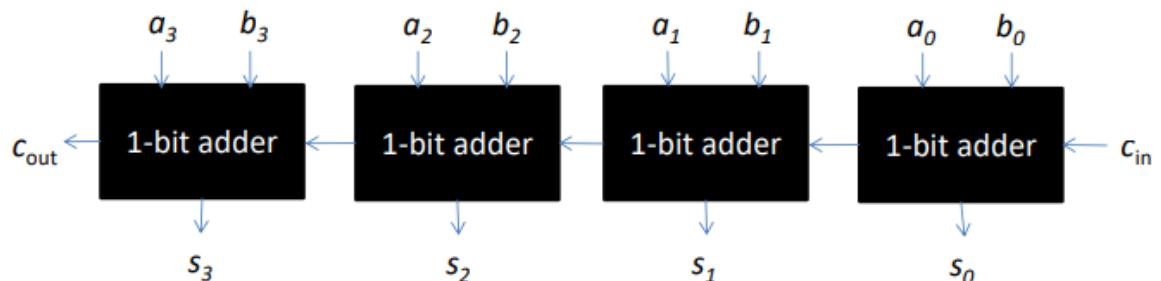
$$p = (a^{(t)} > b^{(t)})$$

$$a^{(t+1)} = p \times (a^{(t)} - b^{(t)}) + !p \times b^{(t)}$$

$$b^{(t+1)} = p \times b^{(t)} + !p \times a^{(t)}$$



Circuit de adunare pe 4 biți



Circuit de adunare pe 4 biți.

ÎMPĂRTIREA NUMERELOR ÎNTREGI

Tratăm împărțitorul ca număr natural în baza 10, fie el **n** în exemplu:

```
int Divide(NrMare x, int n)
//x = x /n, returneaza x%n
{
    int i,r=0;
    for(i=x[0];i>0;i--)
    {
        r=2*r+x[i];
        x[i]=r/n;
        r%=n;
    }
    for(;x[x[0]]==0 && x[0]>1;)
        x[0]--;
    return r;
}
```

- **s = a ÷ b**
 - ce se întâmplă dacă *a* sau *b* sunt variabile negative?
 - rezultatul este negativ dacă *a* și *b* au semne diferite (XOR logic)
 - În general
 - $a = s \times b + r$
 - semnul lui *r* este semnul lui *a*

ÎNMULȚIREA NUMERELOR ÎNTREGI

De la dreapta la stânga

Dacă suntem în B pe bit 1, copiem A

Dacă suntem în B pe bit 0, punem 0 peste tot

A și B naturale

<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	1	1	0	$a = 14$			
1	1	1	0					
<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	1	0	1	$b = 5$			
0	1	0	1					
<hr/>								
<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	1	1	0	$s = 70$			
1	1	1	0					
<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0	ce am făcut aici este corect, dar am presupus că am primit numere naturale. ce se întâmplă dacă <i>a</i> și <i>b</i> sunt în complement față de doi?			
0	0	0	0					
<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>0</td></tr></table>	1	1	1	0				
1	1	1	0					
<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0				
0	0	0	0					
<hr/>								
<table border="1"><tr><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	1	0	0	0	1	1	0	
1	0	0	0	1	1	0		

A și B întregi

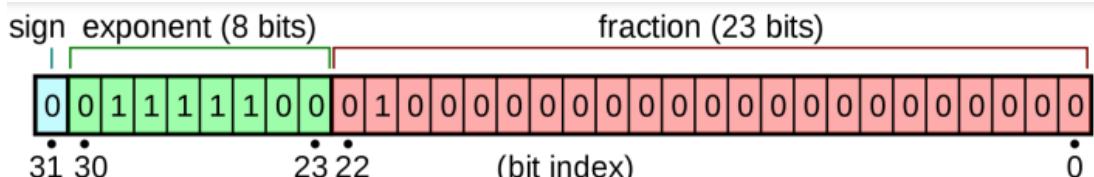
Extindem numerele:

$$A = -2 \quad A = 1110 \quad A = 1111.1110$$

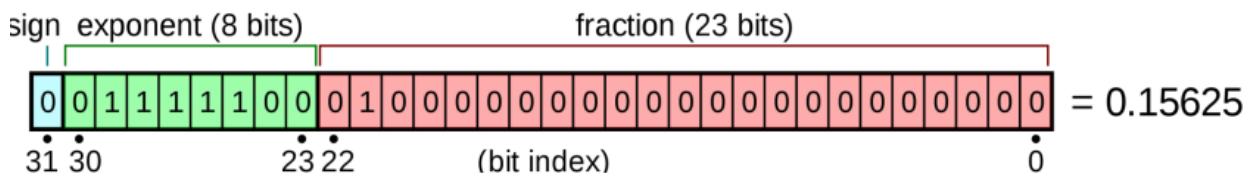
$$B = 5 \quad B = 0101 \quad B = 0000.0101$$

Se calculează la fel ca la numere naturale, dar **rezultatul este în complement față de doi**.

FLOATING POINT



- -1313.3125
 - partea întreagă este: 1313
 - partea fracționară: 0.3125
 - $0.3125 \times 2 = 0.625 \Rightarrow 0$
 - $0.625 \times 2 = 1.25 \Rightarrow 1$
 - $0.25 \times 2 = 0.5 \Rightarrow 0$
 - $0.5 \times 2 = 1.0 \Rightarrow 1$
 - deci, $1313.3125_{10} = 10100100001.0101_2$
 - normalizare: $10100100001.0101_2 = 1.01001000010101_2 \times 2^{10}$
 - mantisa este 010010000101010000000000
 - exponentul este $10 + 127 = 137 = 10001001_2$
 - semnul este 1



Schimbați semnul lui a: $a = a \wedge (1 << 31)$

Exponentul pe: 0x7F80.0000

Extragem exponentul: (a & 0x7F80.0000) >> 23

Pentru a împărti la 4

Exponent > 1 => Exponent = Exponent – 2 | Altfel: a = 0

$a = (a \& \sim MASK) | (exponent << 23)$

SEMN	$a >> 31$
EXPONENT	$(a >> 23) \& 0x000000FF$
MANTISĂ	$a \& 0x007FFFFF$
Nr. în baza 10 cu mantisă de M biți	$\lfloor \log_{10} 2 \times M \rfloor \approx M/3$
abs(a)	$a = a \& \sim(1 << 31)$
$a \times 2$	$a << 1$ sau $a + a$

$a \times 2$	$(a < 0) ? -((-a) \ll 1)) : a \ll 1$
$a \times 16$	$a \ll 4$
$a \times 3$	$a \ll 1 + a$
$a \times 7$	$a \ll 3 - a$
$a / 8$	$a \gg 3$
$a \% 16$	$a \& 0x00F$
$a \% m$	$a \& (m - 1)$
$a / 16$	$(a \& FFF0) \gg 4$
$a \times 72$	$a \ll 6 + a \ll 3$
$a / 16 + a \% 16$	$a \& FFF0 + a \& 000F$

d) $0xDEADBEEF = 0b110111010101101111011101111$

- $S = 1$
- $E = 10111101$
- $M = 0101101101111011101111$
- $(-1)^S \cdot M \cdot 2^{E-127} = (-1) \cdot 1.0101101101111011101111 \cdot 2^{189-127}$
 $= -1.0101101101111011101111 \cdot 2^{62}$
 $= -6259853398707798000$

- setați $s = 0, e = 0, f = 0$
- $a = (-1)^0 \times 1.00...00 \times 2^{-127} = 2^{-127} \neq 0$
- **0.2 + 0.3**
- **primul pas, trecem fiecare număr în format**
 - $0.2 = +1.10011001100110011001100 \times 2^{-3} = 0.19999998807907104$
 - $0.3 = +1.00110011001100110011001 \times 2^{-2} = 0.29999998211860657$
- **al doilea pas, alinierea**
 - $0.2 = +0.110011001100110011001100|000 \times 2^{-2}$
 - $0.3 = +1.00110011001100110011001|000 \times 2^{-2}$
- **al treilea pas, adunăm**
 - $0.2 + 0.3 = 1.111111111111111111111111111111|000 \times 2^{-2}$

- a / 19

$$a \times \frac{1}{19} \approx \frac{a \times \frac{2938661835}{2^{32}} + \frac{a-a \times \frac{2938661835}{2^{32}}}{2^1}}{2^4}$$

$$a \times \frac{1}{19} \approx (a \times 2938661835 \times 2^{-32} + (a - a \times 2938661835 \times 2^{-32}) \times 2^{-1}) \times 2^{-4}$$

$$a \times \frac{1}{19} \approx a \times \frac{7233629131}{137438953472}$$

- soluția generală

$$\frac{a}{D} \approx \frac{\frac{aC}{2^X} + \frac{a - \frac{aC}{2^X}}{2^Y}}{2^Z}$$

$$D \approx \frac{2^{X+Y+Z}}{C \times (2^Y - 1) + 2^X}$$

- calculați aproximarea binară
 - soluția: 0.099999904632568359375
- care este diferența dintre valoarea calculată și 0.1
 - soluția: 0.1 - 0.099999904632568359375
- care este eroarea (de timp) după 100 de ore de operare
 - soluția: $100 \times 60 \times 60 \times 10 \times (0.1 - 0.099999904632568359375) \approx 0.34$
- care este eroarea dacă reprezentăm 0.1 în formatul IEEE 754 FP?
 - soluția: $100 \times 60 \times 60 \times 10 \times (0.1 - 0.09999999403953552) \approx 0.021$
- dacă rachetele SCUD pot atinge o viteza MACH 5, care este distanța pe care racheta o poate parcurge în timpul eroare calculat?
 - soluția: $1715 \text{ m/s} \times 0.34 \text{ s} \approx 583 \text{ m}$, $1715 \text{ m/s} \times 0.021 \text{ s} \approx 36 \text{ m}$

FAST INVERSE SQUARE ROOT, Q III

```

552 float Q_rsqrt( float number )
553 {
554     long i;
555     float x2, y;
556     const float threehalves = 1.5F;
557
558     x2 = number * 0.5F;
559     y = number;
560     i = *( ( long * ) &y );                                // evil floating point bit level hacking
561     i = 0x5f3759df - ( i >> 1 );                      // what the duck?
562     y = *( ( float * ) &i );
563     y = y * ( threehalves - ( x2 * y * y ) );          // 1st iteration
564     y = y * ( threehalves - ( x2 * y * y ) );          // 2nd iteration, this can be removed

```

- prima instrucțiune interesantă e pe linia 560

- dacă avem în M mantisa și exponentul în E atunci

- numărul nostru în FP este $2^{23} \times E + M$
- iar valoarea numărului este $(1 + M / 2^{23}) \times 2^{E - 127}$
- observăm că:

$$\begin{aligned}\log_2 \left(\left(1 + \frac{M}{2^{23}} \right) \times 2^{E-127} \right) &= \log_2 \left(1 + \frac{M}{2^{23}} \right) + \log_2 (2^{E-127}) \\ &= \log_2 \left(1 + \frac{M}{2^{23}} \right) + E - 127 \\ &\approx \frac{M}{2^{23}} + E - 127 + \mu \quad (\text{am folosit } \log_2(1+x) \approx x, \mu \text{ este o corecție}) \\ &= \frac{1}{2^{23}} (2^{23} \times E + M) + \mu - 127 \\ &= \frac{1}{2^{23}} (\text{reprezentarea pe biti}) + \mu - 127\end{aligned}$$

- de ce calculăm $\log(y)$? defapt vrem $1/\sqrt{y}$. dar:

$$\log_2 \left(\frac{1}{\sqrt{y}} \right) = \log_2 (y^{-1/2}) = -\frac{1}{2} \log_2 (y) = -(i \gg 1)$$

- $\pi \approx 3.14159265 = (-1)^0 1.10010010000111111011010 2^{b10000000-127}$
- $+0 = (-1)^0 1.0000000000000000000000000000000 2^{00000000-127}$
- $-0 = (-1)^1 1.0000000000000000000000000000000 2^{00000000-127}$
- signaling NaN: 0x7F800001 sau 0x7FBFFFFF sau între 0xFF800001 și 0xFFBFFFFF
- quiet NaN: 0x7FC00000 sau 0xFFFFFFF sau între 0xFFC00000 și 0xFFFFFFF

CONSECINȚE FLOATING POINT

- $(0.1 + 0.2) == 0.3$ versus $(0.2 + 0.3) == 0.5$ (rotunjiri)
- $\text{math.sqrt}(3)*\text{math.sqrt}(3) == 3$ versus $\text{math.sqrt}(3*3) == 3$
- $(0.7 + 0.2) + 0.1$ versus $(0.7 + 0.1) + 0.2$ (nu avem asociativitatea)
- diferența cu numere întregi
 - dacă folosim tip de date întreg: $16777216 + 1 = 16777217$
 - dacă folosim tip de date FP: $16777216.0 + 1 = 16777216.0$
 - $\text{float}(123456789101112) + 1.0 = 123456789101113.0$
 - $\text{float}(1234567891011121) + 1.0 = 1234567891011122.0$
 - $\text{float}(12345678910111213) + 1.0 = 1.2345678910111212e+16$

ARHITECTURA CALCULATOARELOR MODERNE

Pornirea sistemului

- Buton de power ON/OFF > Realizează alimentarea cu electricitate a componentelor
- CPU este activat > Caută pornește BIOS
 - Testează componentele HW (RAM, I/O, HDD, etc.)
 - Încarcă BIOS (scris pe placă de bază) din ROM (read only memory) în RAM pentru execuție

BIOS: Știe cât e ceasul (CMOS Real-Time Clock) și HW, se accesează cu F2 la pornirea sistemului

CPU/BIOS: Pornesc Boot Code (caută sistemul de operare)

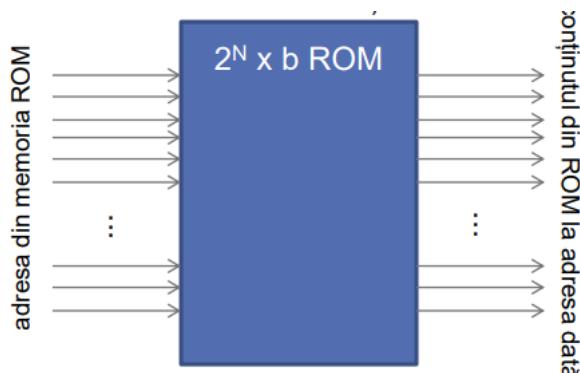
- În general, SO este pe HD (poate fi și pe CD, stick). Se încarcă în RAM pentru execuție

Componente

TOT BIOS: Scris în ROM, câteodată în Programmable ROM, Erasable Programmable ROM

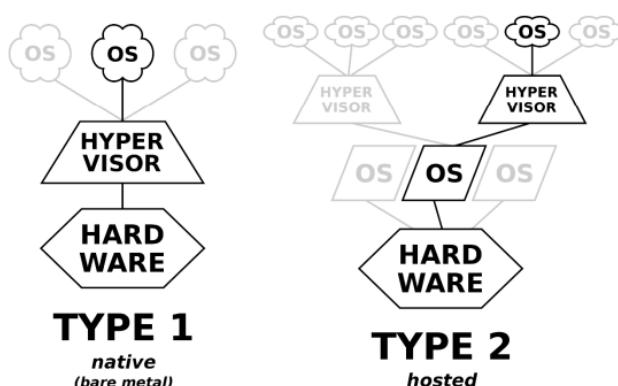
„Să scriem în ROM”: „burning” sau „flashing” the ROM

Este un circuit combinațional



OS preia controlul de la BIOS

- Doar OS are acces la periferice, prin drivere
- Virtualizare/Emulare/Containere(Dockere)



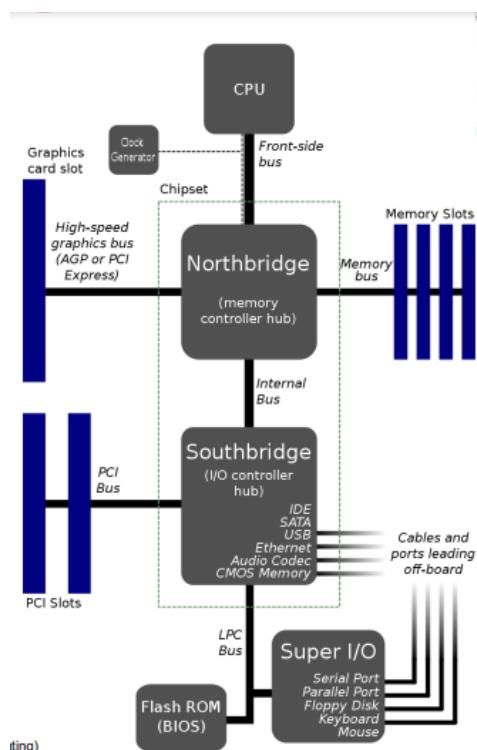
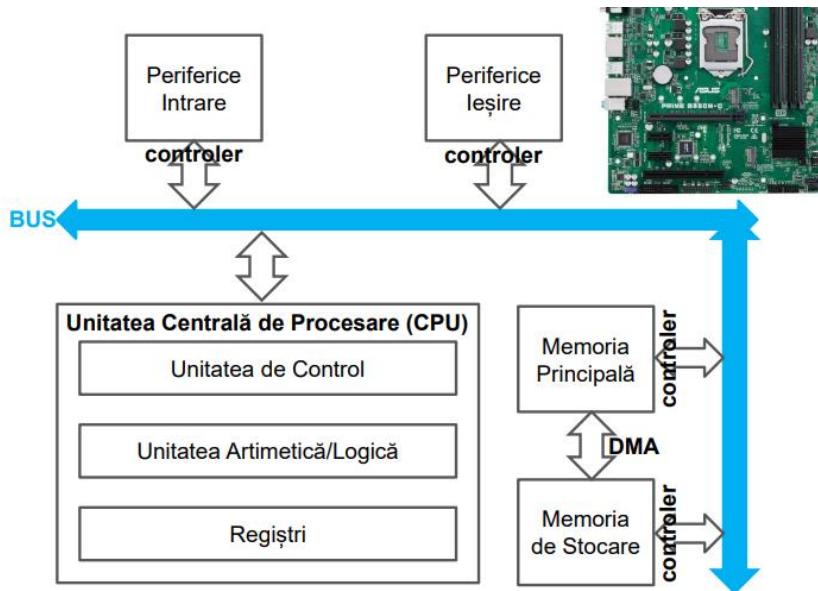
OS: Oferă imagine abstractizată a memoriei pentru fiecare proces pornit.

OS pornește > Sistemul de calcul intră în ciclul obișnuit de procesare

(Secvența boot se încheie)

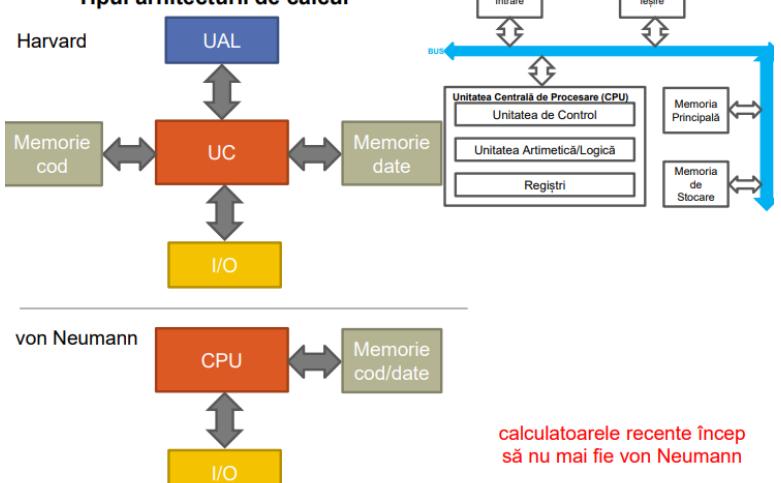
Un sistem de calcul trebuie să

- Calculeze > Să execute instrucțiuni
- Să comunice > Să transfere biți între componente electronice
- Să stocheze > Date - folosite de instrucțiuni | Instrucțiuni pentru execuție



• Tipuri arhitecturii de calcul

Harvard



calculatoarele recente încep să nu mai fie von Neumann

I. CPU (5 componente) – Creierul unității de calcul / Execută instrucțiuni

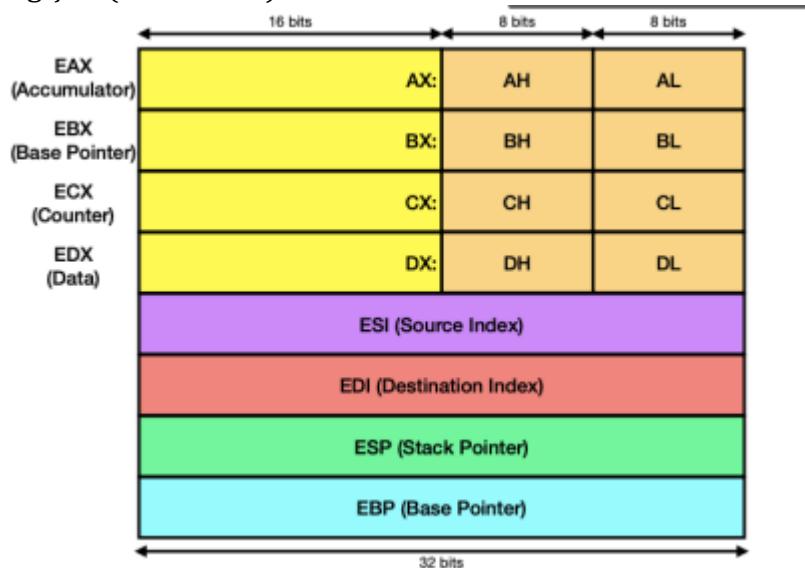
1. Clock

- Circuit special care generează „ceasul”

- Frecvența la care operează CPU (calcule + sincronizarea componentelor secvențiale)

- Frecvență mare -> Mai bine | Se măsoară în MHz sau GHz

2. Reșiștri („memoria”)



3. UAL („operații”) – Unitatea aritmetică-logică

- Operații logice

- Operații aritmetice cu int/float

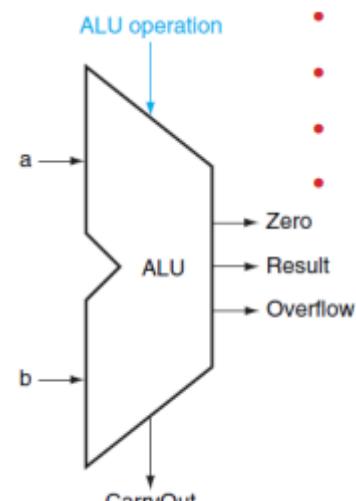
- Operații speciale: sqrt, exp, trig

4. BUS

- CPU are nevoie de șiruri de biți din memoria principală sau de stocare

- CPU are nevoie să scrie înapoi rezultatele

- CPU coordonează perifericele



5. UC („instrucțiunile”) – Unitatea de coordonare

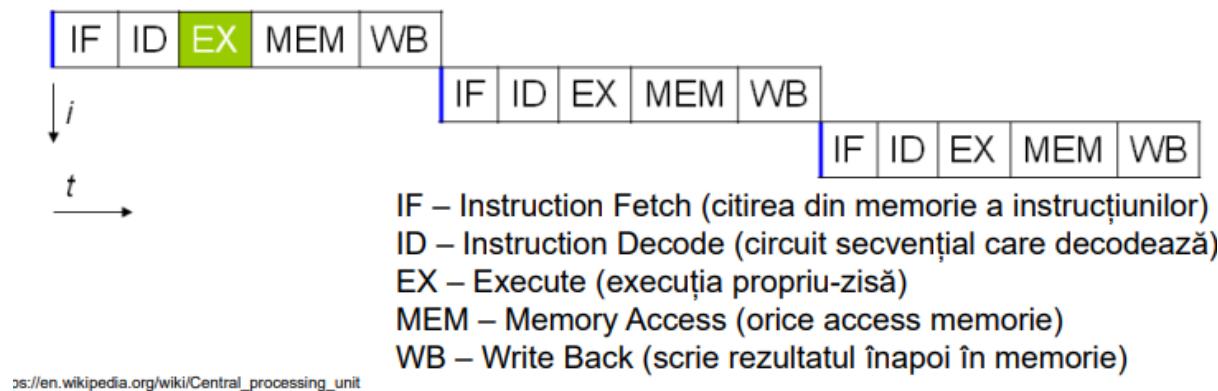
- **Fetch:** Citim din memorie codul care trebuie executat (dictat de Instruction Pointer / EIP)

- **Decode:** Circuitul „Instruction Decoder” analizează biții din memorie ca să înțeleagă ce să facă cu ei

- **Execute:** Execută instrucțiunea decodată, poate duce la schimbarea Instruction Pointer / transmiterea a ceva pe BUS către memorie

- Calculează următorul Instrucion Pointer

- fetch
 - IP = 10011 (locația în memorie de unde să citim biții)
 - după citire, IP este actualizat
- decode
 - s-a citit "1110011" care este decodat în
 - opcode = 111, operand1 = 00 operand2 = 11
 - de exemplu: 111 = "adună registrul A la registrul R", R = 00 este EAX, A = 11 este EDX (prin convenție)
- execute
 - trimite EAX ← EAX + EDX la UAL
 - citește rezultatul din UAL și pune-l în registrul EAX

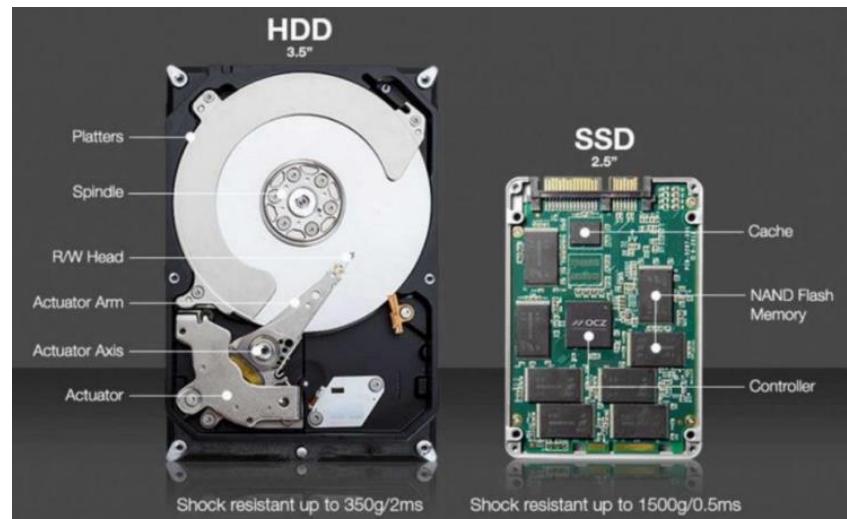


II. Memoria principală

- Conține cod și date
 - Este volatilă
- Tipuri
- Static RAM (SRAM)
 - Bazat pe flip-flops, rapid, scump, regiștri din CPU de același tip
 - Dynamic RAM (DRAM)
 - Fiecare bit reprezentat prin combinație de tranzistor + condensator (condensatoarele suferă de leakage – surgeri de tensiune)
 - DRAM trebuie actualizat o dată la fiecare câteva zeci de ms
 - Double Data Rate RAM (DDR RAM: DDR4/5/6)
 - Performanță definită de capacitate, dacă au sistem pentru ECC, timpi de acces (temp de refresh / în cât timp de la citire sunt disponibilizate datele)
 - Consumul de energie

III. Memorie de Stocare

- Conține cod și date, este nevolatilă
- SSD (Solid State Disk): Memorie flash, rapidă, scumpă, scriere mult mai lentă decât citirea
- HDD (Hard Disks): Mecanic



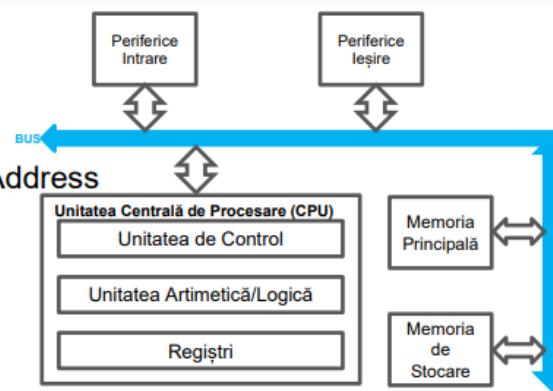
IV. BUS

- Conectează CPU/Memorie
- Proprietăți: Capacitatea (Lățimea de bandă / Bandwidth) + Viteza

PIPELINING

- CPU execută instrucțiuni:

- IF – Instruction Fetch
- ID – Instruction Decode
 - COA – Calculate Operand Address
 - FO – Fetch Operand
 - FOs – Fetch Operands
- EX – Execution
- MEM – Memory Access
- WB – Write Back



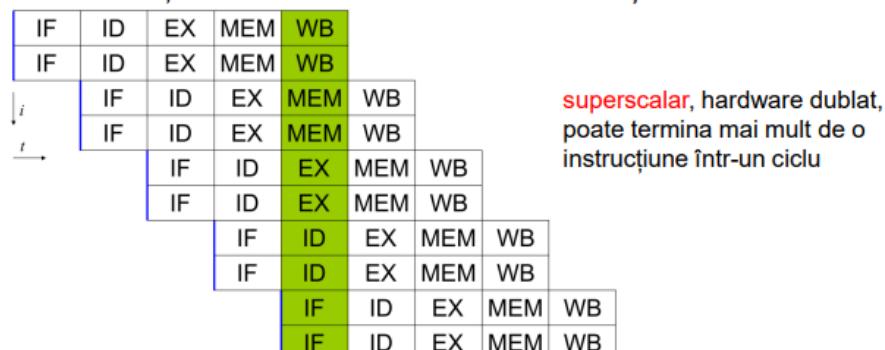
Pipeline stalls = Întârzieri în conducta de date

Aceste evenimente se numesc hazards (erori)

Structural Hazards – O unitate de calcul este deja utilizată

I. Structural Hazards

- două instrucțiuni încearcă să acceseze aceeași unitate

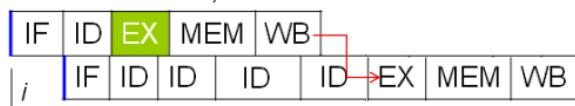


2. Data Hazards – Datele nu sunt pregătite pentru utilizare

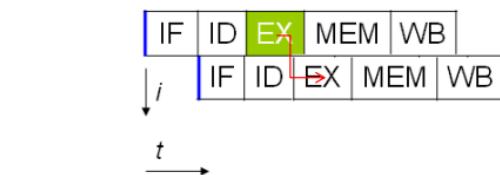
- o instrucțiune depinde de rezultatul unei instrucțiuni anterioare
- **True Dependence (Read After Write – RAW)**
 - add %ebx, %eax
 - sub %eax, %ecx
- **Anti-dependence (Write After Read – WAR)**
 - add %ebx, %eax
 - sub %ecx, %ebx
- **Output Dependence (Write After Write – WAW)**
 - mov \$0x10, %eax
 - mov \$0x01, %eax

- **True Dependence (Read After Write – RAW)**

- add %ebx, %eax
- sub %eax, %ecx



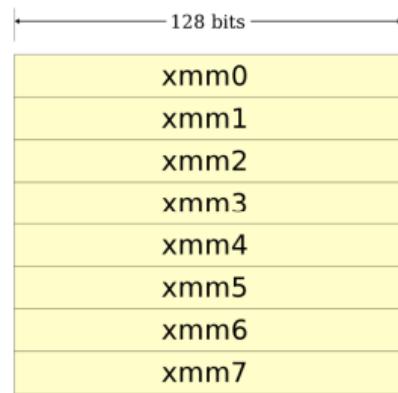
- posibilă soluție *bypassing*



- suplimentar, situația e complicată de faptul că instrucțiuni diferite au nevoie de un număr diferit de cicli de CPU

operația	instrucțiuni	# cicli
operații întregi/biți	add, sub, and, or, xor, sar, sal, lea, etc.	1
înmulțirea întregilor	mul, imul	3
împărțirea întregilor	div, idiv	depinde (20–80)
adunare floating point	addss, addsd	3
înmulțire floating point	mulss, mulsd	5
împărțire floating point	divss, divsd	depinde (20–80)
fused-multiply-add floating point	vfmass, vfmasd	5 avem registrii speciali

- pentru instrucțiuni “dificile” avem regiștri speciali
 - xmm?
 - xmm8-xmm15 există doar pe x64
 - original, regiștri suportau
 - 4 înmulțiri pe 32-bit FP
 - datorită SSE2, operații cu
 - două numere 64-bit FP
 - două numere întregi pe 64-biți
 - 4 numere pe 32 de biți
 - 8 numere pe 16 biți
 - 16 numere pe 8 biți
 - Streaming SIMD Extensions
 - exemplu: $\text{result.x} = \text{v1.x} + \text{v2.x}$, $\text{result.y} = \text{v1.y} + \text{v2.y}$, $\text{result.z} = \text{v1.z} + \text{v2.z}$, $\text{result.w} = \text{v1.w} + \text{v2.w}$
 - devine:
 - $\text{movaps v1, xmm0} \# \text{xmm0} = \text{v1.w} | \text{v1.z} | \text{v1.y} | \text{v1.x}$
 - $\text{movaps v2, xmm1} \# \text{xmm1} = \text{v2.w} | \text{v2.z} | \text{v2.y} | \text{v2.x}$
 - $\text{addps xmm1, xmm0} \# \text{xmm0} = \text{v1.w} + \text{v2.w} | \text{v1.z} + \text{v2.z} | \text{v1.y} + \text{v2.y} | \text{v1.x} + \text{v2.x}$
- redenumirea regiștrilor (*register renaming*)
- considerăm codul (unde folosim doar registrul R1)
 - $\text{R1} = \text{data}[1024]$
 - $\text{R1} = \text{R1} + 2$
 - $\text{data}[1032] = \text{R1}$
 - $\text{R1} = \text{data}[2048]$
 - $\text{R1} = \text{R1} + 4$
 - $\text{data}[2056] = \text{R1}$
- echivalent, dar mai bine pentru pipelining
 - $\text{R1} = \text{data}[1024]$
 - $\text{R1} = \text{R1} + 2$
 - $\text{data}[1032] = \text{R1}$
 - $\text{R2} = \text{data}[2048]$
 - $\text{R2} = \text{R2} + 4$
 - $\text{data}[2056] = \text{R2}$
- ultimele 3 instrucțiuni se pot executa acum în paralel cu primele 3



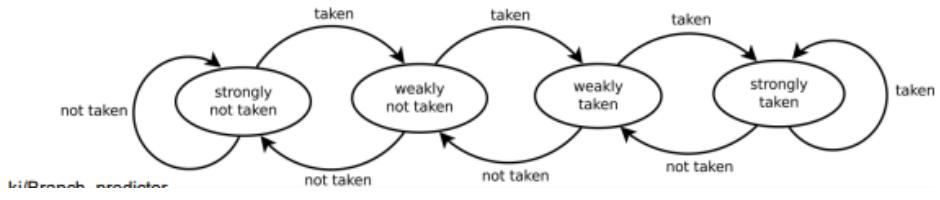
- ce se întâmplă în procesoarele moderne?
 - se citesc câteva sute de instrucțiuni la un moment dat
 - în hardware, se realizează un graf (*data-flow graph*) de dependențe între aceste instrucțiuni
 - exemplu: quad(a, b, c)
 - $t1 = a * c; t3 = b * b; t6 = -b; t9 = 2 * a;$
 - $t2 = 4 * t1;$
 - $t4 = t3 - t2;$
 - $t5 = \sqrt{t4};$
 - $t7 = t6 - t5; t8 = t6 + t5;$
 - $r1 = t7/t9; r2 = t8/t9;$
- 6 vs. 11 pași
-
- out of order execution

BRANCH PREDICTION

- o potențială soluție: branch prediction (predicția saltului)

```
f2:
    pushq %rbx
    xorl %ebx, %ebx
.L3:
    movl %ebx, %edi
    addl $1, %ebx
    call callfunc
    cmpl $10, %ebx
    jne .L3
    popq %rbx
    ret
```

- în general predicția este binară (dacă sări sau nu, *then* sau *else*)
- ce propuneți voi?
 - predicție fixă: mereu sare / mereu nu sare
 - predicția de la pasul anterior: ce a făcut instrucțiunea ultima dată
 - predicția cu istoric: ce face de obicei instrucțiunea
 - execuție speculativă (eager execution): calculează cu și fără salt



ce e asta?
counter 2 biți

Când complicăm hardware și software pot apărea probleme

În special, probleme de securitate: Meltdown, spectre -> Aceste două atacuri exploatează execuția speculativă și sistemul ierarhic al memoriei (Cache-ul)

Când complicăm hardware, e mai rău:

Soluție: Trebuie înlocuit hardware-ul

Soluție: Sistemul de operare trebuie să ia în considerare problema | Totul va fi mai lent

--

Adresa de memorie cea mai accesibilă $2^{32} = 4\text{GB}$ (4.294.967.296 bytes)

Adresa de memorie cea mai accesibilă pentru **jne etichetă cu OPCODE 0110** 2^{28}

Regiștri suportați de add R1 R2 (OPCODE: 0011) 2^{14}

Regiștri suportați de add R1 R2 R3 (OPCODE: 0100) 9.33 biți pentru fiecare reprezentare a unui registru: Două poziții suportă 2^9 | 0 poziție suportă 2^{10}

int i = 1; i++ // == 1 și i == 2

int i = 1; ++i // == 2 și i == 2 | Compilatorul nu mai are nevoie de variabilă temporară

<pre> main: ; initializare xor %eax, %eax » int sum = 0; xor %ecx, %ecx ; while loop et_loop: add %ecx, %eax inc %ecx » • totul în regiștri cmp \$10, %ecx • ca programul acesta să fie identic cu while jne et_loop • la sfârșit • mov %eax, sum • mov \$10, i ; afiseaza suma push %eax push \$formatPrint call printf pop %ebx pop %ebx ; flush push \$0 call fflush pop %ebx ; exit mov \$1, %eax mov \$0, %ebx int \$0x80 </pre>	<ul style="list-style-type: none"> • se poate cu mai puține instrucțiuni? <ul style="list-style-type: none"> • da, parcuregere inversă • se poate cu și mai puține instrucțiuni? <ul style="list-style-type: none"> • da: mov \$45, %eax <p style="text-align: center;">» LOOP UNROLLING</p> <pre> int sum = 0; int i = 0; for (i = 0; i < 10; i+=2) { sum += i; sum += i+1; } </pre> <ul style="list-style-type: none"> • de ce am vrea să facem aşa ceva? <ul style="list-style-type: none"> • mai puține salturi
---	--

- a) $\%eax \leftarrow \%ebx + \%ecx$
 $\%eax \leftarrow \%ebx + \%edx$ **WAW**
- b) $\%ebx \leftarrow \%ecx + \%eax$
 $\%eax \leftarrow \%edx + \%eax$ **WAR**
- c) $\%eax \leftarrow \%ebx + \%ecx$
 $\%edx \leftarrow \%eax + \%edx$ **RAW**
- d) $\%eax \leftarrow 6$
 $\%eax \leftarrow 3$ **WAW**
 $\%ebx \leftarrow \%eax + 7$ **RAW , rezultatul poate fi 10 sau 13**

ALGORITM DE INTERCLASARE / MERGE

- ce face algoritmul?
 - merge (interclasare)
- câte instrucțiuni de salt avem?
 - 4
- predicția pentru fiecare?
 - Salt 1: sare mereu
 - **Salt 2: în general, nu știm**
 - Salt 3: sare mereu
 - Salt 4: sare mereu
- cum eliminăm Saltul 2?
 - int cmp = (*A <= *B)
 - int min = *B ^ ((*B ^ *A) & (-cmp))
 - *C++ = min
 - A += cmp, na -= cmp
 - B += !cmp, nb -= !cmp

```

while (na > 0 && nb > 0)
{
    if (*A <= *B) {
        *C++ = *A++; --na;
    } else {
        *C++ = *B++; --nb;
    }
}

while (na > 0) {
    *C++ = *A++; --na;
}

while (nb > 0) {
    *C++ = *B++; --nb;
}

```

TOUPPER()

```

void toUpper(char *buff, int count) {
    for (int i = 0; i < count; ++i)
    {
        if (buff[i] >= 'a' && buff[i] <= 'z')
            buff[i] -= 32;
    }
}

```

- branchless? mai bine?

```

void toUpper(char *buff, int count) {
    for (int i = 0; i < count; ++i)
    {
        buff[i] = buff[i]*!((buff[i] >= 'a' && buff[i] <= 'z'))
                  + (buff[i] - 32)*(buff[i] >= 'a' && buff[i] <= 'z');
    }
}

void toUpper(char *buff, int count) {
    for (int i = 0; i < count; ++i)
    {
        buff[i] -= 32*(buff[i] >= 'a' && buff[i] <= 'z');
    }
}

```

VERIFICARE X² + Y² = Z² PENTRU X <= Y

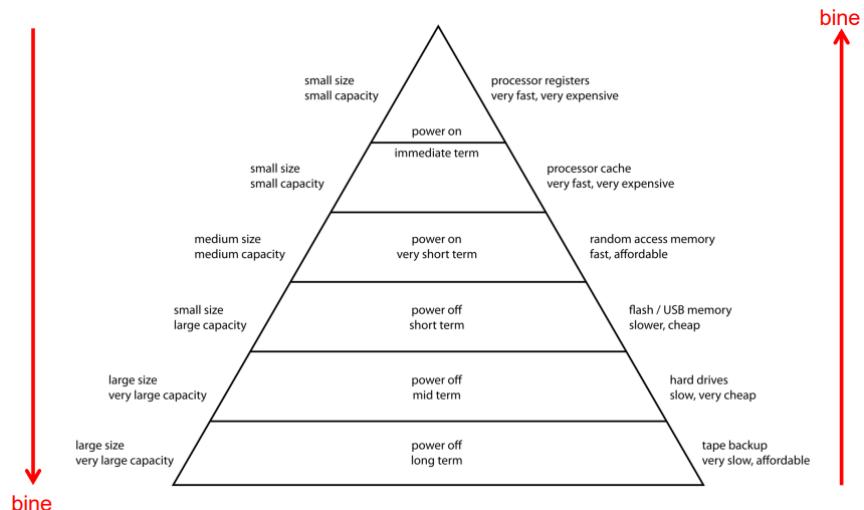
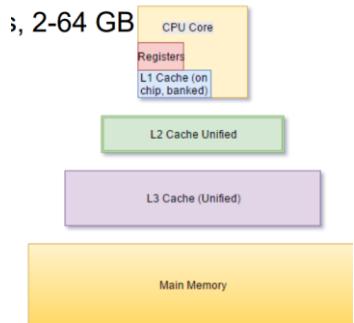
```
.globl f
f:
    movl $1, %r8d
    jmp .LBB0_1
.LBB0_6:
    incl %r8d
.LBB0_1:
    movl %r8d, %ecx
    imull %ecx, %ecx
    movl $1, %edx
.LBB0_2:
    movl %edx, %edi
    imull %edi, %edi
    movl $1, %esi
    .align 16, 0x90
.LBB0_3:
    movl %esi, %eax
    imull %eax, %eax
    addl %edi, %eax
    cmpl %ecx, %eax
    je .LBB0_7
    cmpl %edx, %esi
    leal 1(%rsi), %eax
    movl %eax, %esi
    jl .LBB0_3
    cmpl %r8d, %edx
    leal 1(%rdx), %eax
    movl %eax, %edx

    jl .LBB0_3
    cmpl %r8d, %edx
    leal 1(%rdx), %eax
    movl %eax, %edx
    jl .LBB0_2
    jmp .LBB0_6
.LBB0_7:
    pushq %rax
.Ltmp0:
    movl $.L.str, %edi
    xorl %eax, %eax
    callq printf
    movl $1, %eax
    popq %rcx
    retq

.L.str:
```

IERARHIA MEMORIEI

TIP DE MEMORIE	CARACTERISTICI	VITEZĂ	CAPACITATE
Regiștrii procesorului	Acces imediat		100-1000 bytes
Cache L0	Acces foarte rapid		5-20 KBytes
Cache L1	Cache instrucțiuni și date	700GB/s	100-500Kbytes
Cache L2		200 GB/s	500-1000 Kbytes
Cache L3		100GB/s	1-5MB
Memoria principală RAM		100-500MB/s	2-64GB
Disc HD/SDD		10-100MB/s	1TB



- **de ce e bine să avem cache?**
- **presupunem că timpii de acces sunt:**
 - în memoria principală: 50 ns
 - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
 - în L2: 5 ns cu 1% miss rate
 - în L3: 10 ns cu 0.2% miss rate
- **să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:**
 - verificăm în RAM: 50 ns
 - verificăm în L1: $1\text{ ns} + (0.1 \times 50\text{ ns}) = 6\text{ ns}$
 - verificăm în L2: $1\text{ ns} + (0.1 \times (5\text{ ns} + (0.01 \times 50\text{ ns}))) = 1.55\text{ ns}$
 - verificăm în L3: $1\text{ ns} + (0.1 \times (5\text{ ns} + (0.01 \times (10\text{ ns} + (0.002 \times 50\text{ ns})))))) = 1.5101\text{ ns}$

observați trade-off-ul între viteza de acces și probabilitatea de miss rate

cât este miss rate în RAM? 0% (în RAM sigur avem informația)
cu ce dimensiune are legatură miss rate? cu dimensiunea memoriei

- cu un miss rate de 10% merită să avem cache L1
- pentru ce probabilitate miss rate nu mai merită cache L1?
 - $p = 49 / 50 = 98\%$

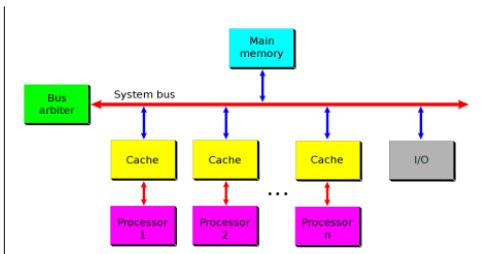
- **Exemplu:**

- memoria RAM este 1 GB
- memoria cache L1 este 128 kbytes
- memoria RAM este de aproximativ 8000 de ori mai multă decât memoria cache L1, deci cum putem avea miss rate 10%?
- ne bazăm pe **principiul de localizare**

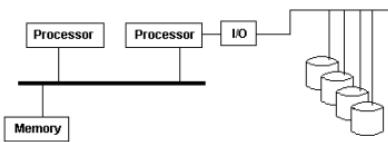
IERARHIZAREA MEMORIEI

- când programezi, nu vedeți această ierarhizare
- cine e responsabil de ce anume?
 - programatorul: transfer între HD/SSD și RAM (citire de pe disc)
 - logică hardware: din/in RAM în/din memoriile cache
 - compilatorul: generează cod care exploatează cache-ul
- cât timp performanța este acceptabilă totul e OK, apoi Assembly

SYMMETRIC MULTIPROCESS SYSTEMS



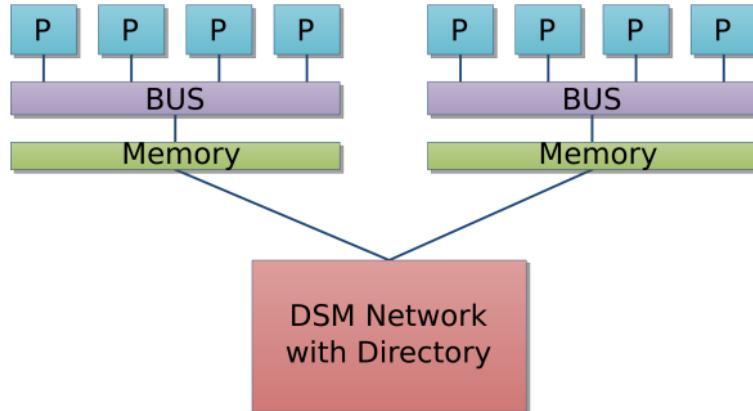
ASYMMETRIC MULTIPROCESS SYSTEMS



- UMA (Uniform Memory Access)
- procese diferite pe procesoare diferite, dar e nevoie de modificarea programelor ca acestea să ruleze paralel
- dezavantaj: cache coherence

- fiecare procesor are se ocupă de ceva diferit
 - unul execute programe
 - altul se ocupă de I/O

NON-UNIFORM MEMORY ACCESS



- rezolvă problema accesului la memoria de către procesoare multiple (fiecare procesor are memoria/cache-ul său)
- procesoarele așteaptă mai puțin să ajunge datele la ele

2 Exerciții

1. Presupunem că avem un sistem de calcul pe 32 de biți în care accesul la memoria RAM este $t_{RAM} = 50\text{ns}$, accesul la memorile cache sunt: $t_{L1} = 1\text{ns}$ cu miss rate $m_{L1} = 10\%$, $t_{L2} = 5\text{ns}$ cu $m_{L2} = 1\%$ și $t_{L3} = 10\text{ns}$ cu $m_{L3} = 0.2\%$. Răspundeți la următoarele întrebări scurte:
 - (a) calculați noua valoare m_{L1} pentru care timpul de acces la memorie este jumătate t_{RAM} ;
 - (b) calculați noua valoare t_{L2} pentru care timpul de acces la memorie este o zecime din t_{RAM} ;
 - (c) calculați noua valoare m_{L3} pentru care timpul de acces la memorie este același t_{RAM} ;
 - (d) avem posibilitatea de a îmbunătăți timpii de răspuns pentru memorile cache cu câte 10% pentru costurile $c_{L1} = 100\$$, $c_{L2} = 25\$$ și $c_{L3} = 5\$$ (costuri mai mari pentru cache mai rapid). Reduceti timpul de acces la memorie de o mie de ori față de t_{RAM} cu cost minim;
 - (e) presupunem că avem, în general, memorii cache Li cu $t_{Li} = \left(\frac{i}{i+1}\right)^2 \times t_{RAM}$ și $m_{Li} = \frac{1}{(i+1)^2}$, pentru $i = 1, \dots, n$. Sunt aceste valori consistente pentru $n \rightarrow \infty$? Care este timpul de acces la memorie în acest caz?

a) timpul total de acces memorie este (din curs)

$$1\text{ ns} + (0.1 \times (5\text{ ns} + (0.01 \times (10\text{ ns} + (0.002 \times 50\text{ ns }))))))$$

în cazul nostru

$$1\text{ ns} + (A \times (5\text{ ns} + (0.01 \times (10\text{ ns} + (0.002 \times 50\text{ ns })))))) = t_{RAM} / 2$$

b) $1\text{ ns} + (0.1 \times (A \text{ ns} + (0.01 \times (10\text{ ns} + (0.002 \times 50\text{ ns })))))) = t_{RAM} / 10$

c) $1\text{ ns} + (0.1 \times (5\text{ ns} + (0.01 \times (10\text{ ns} + (A \times 50\text{ ns })))))) = t_{RAM}$

d) pentru L1, să trecem de la 1ns la 0.9ns ne costă 100\$

pentru L2, să trecem de la 5ns la 4.5ns ne costă 25\$

pentru L3, să trecem de la 10ns la 9ns ne costă 5\$

$$1 \times 0.9^A + (0.1 \times (5 \times 0.9^B + (0.01 \times (10 \times 0.9^C + (0.002 \times 50)))))) = t_{RAM}/1000$$

vrem: minimize $100 \times A + 25 \times B + 5 \times C$

rezolvați pentru A, B și C

Criterii de performanță

Ne interesează să putem răspunde la cât mai multe cereri de date venite de pe Internet:

- Utilizare CPU (Eventual sisteme multi-core), media aritmetică

Vrem să ne asigurăm că cererile de date sunt finalizate în 10ms

- Wall-clock time, media aritmetică

Ne interesează să ocupe maxim 100MB în memorie

- Memoria RAM, maximum

Vrem să rulăm sistemul de calcul cu un cost cât mai mic

- Performanța per Watt, media aritmetică sau maximum

Vrem să știm că majoritate cererilor sunt servite în maximum 50ms

- Wall-clock time, 50/90/99th percentile mediana

Vrem să estimăm timpul total de răspuns al sistemului

- Wall-clock time speedup, media aritmetică

Este vreodată folositor să estimăm performanța folosind minimul/maximul?

- Minimul – Când zgromotul / erorile din sistem sunt minime (best case)
- Maximum – Când zgromotul / erorile din sistem sunt maxime (worst case)

De ce este important să măsurăm cât mai bine / exact performanța unui sistem de calcul?

- Dacă măsurăm cât mai bine și exact fiecare componentă, putem optimiza cât mai bine.

3. Presupunem că avem Program A și Program B pe care le rulăm pe același sistem de calcul. Programele rulează de 4 ori iar timpii de execuție sunt prezenți în tabelul de mai sus. Cerințe:

- (a) calculați media aritmetică pentru timpii de execuție pentru fiecare program în parte și pentru fiecare test în parte.
- (b) de câte ori se execută mai repede Program B față de program A?
- (c) de câte ori se execută mai repede Program A față de program B?
- (d) ce fel de medie putem să folosim când comparăm cele două programe?
- (e) care este cea mai bună metodă de a compara două programe între ele (head-to-head)?

Test	Program A	Program B	A/B	B/A
1	9	3	3	0.33
2	8	2	4	0.25
3	2	20	0.1	10
4	10	2	5	0.2
Media	7.25	6.75	3.025	2.7
Media	(a) 7.25	(a) 6.75	(g) 1.57	(g) 0.64

Concluzia: Program B este de 3 ori mai rapid decât Program A

Concluzia: Program A este de 2.7 ori mai rapid decât Program B

Nu luați media aritmetică a rapoartelor A/B sau B/A

Luați media geometrică a rapoartelor A/B sau B/A

- în acest caz, media rapoartelor este raportul medilor

Vrem să comparăm Program A vs. Program B: cine este mai rapid? A sau B?

- rulăm programele de mai multe ori
- comparăm linie cu linie în tabelul de mai sus
- pentru fiecare linie decidem cine câștigă (A sau B)
- apoi calculăm: care este probabilitatea ca A să fie mai rapid decât B dacă am observat că în n cazuri (din totalul de N) A este mai rapid decât B
- p-value

4. Se dau două numere complexe $x = a + bi$ și $y = c + di$. Răspundeți la următoarele întrebări:

- scrieți explicit formula pentru $z = x \times y$;
- câte adunări și înmulțiri se realizează pentru calculul lui z ?
- puteți să calculați z cu mai puține înmulțiri?
- de câte ori ar trebui să fie mai lentă o înmulțire față de o adunare pentru ca rezultatul de la punctul precedent să fie eficient?
- ideea de a înlocui o înmulțire cu mai multe adunări (în general, o operație dificilă cu o serie de operații simple) apare de mai multe ori în algoritmici (vedeți algoritmul lui Strassen).

a) $z = (a+bi)(c+di) = ac - bd + i(ad + bc)$

b) 2 adunări, 4 înmulțiri

c) calculăm $S1 = ac$, $S2 = bd$ și $S3 = (a+b)(c+d)$

$$z = S1 - S2 + i(S3 - S1 - S2)$$

5 adunări, 3 înmulțiri

d) $C1$ – costul unei adunări

$C2$ – costul unei înmulțiri

$$2C1 + 4C2 > 5C1 + 3C2$$

$$C2/C1 > 3$$

e) algoritmul lui Strassen

- vrem să calculăm $C = AB$ (unde A și B sunt matrice)

```

for (i = 0; i < row_length_A; i++)
{
    for (k = 0; k < column_length_B; k++)
    {
        sum = 0;
        for (j = 0; j < column_length_A; j++)
        {
            sum += A[i][j] * B[j][k];
        }
        C[i][k] = sum;
    }
}

```

- pe blocuri:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$$

$$C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$$

$$C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$$

$$C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

$$M_1 := (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})$$

$$M_2 := (A_{2,1} + A_{2,2})B_{1,1}$$

$$M_3 := A_{1,1}(B_{1,2} - B_{2,2})$$

$$M_4 := A_{2,2}(B_{2,1} - B_{1,1})$$

$$M_5 := (A_{1,1} + A_{1,2})B_{2,2}$$

$$M_6 := (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$

$$M_7 := (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})$$

O(n^3)

$$C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5$$

$$C_{2,1} = M_2 + M_4$$

$$C_{2,2} = M_1 - M_2 + M_3 + M_6$$

O($n^{2.8}$)

LUCRU CU VECTOR/MATRICE, EX. 5

- ### • produs scalar

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	...	x_9
-------	-------	-------	-------	-------	-------	-------	-------	-----	-------

y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	\dots	y_9
-------	-------	-------	-------	-------	-------	-------	-------	---------	-------

- cum calculăm eficient?

- $c += x_i y_i$
 - instrucțiune Fast Multiply-Add (fma)
 - aceeași operație pe date diferite (SIMD)
 - foarte ușor de paralelizat
 - atenție, rezultatul va fi diferit (adunarea nu mai e asociativă)
 - cu p procesoare ne așteptăm să fim de p ori mai rapizi
 - exploatează cache la maxim: datele sunt continue în memorie

LUCRU CU VECTOR/MATRICE, EX. 5

- **produs matrice-matrice**

$$\begin{array}{c|ccc}
 1 & 2 & 3 \\
 \hline
 4 & 5 & 6 \\
 \hline
 1 & 3 & 2
 \end{array}
 \quad
 \begin{array}{c|cc}
 10 & 11 \\
 \hline
 7 & 5 \\
 \hline
 2 & 4
 \end{array}
 \quad
 = \quad
 \begin{array}{c|cc}
 1*10+2*7+3*2 & 1*11+2*5+3*4 \\
 \hline
 4*10+5*7+6*2 & 4*11+5*6+4*4 \\
 \hline
 1*10+3*7+2*2 & 1*11+3*5+2*4
 \end{array}$$

- cum calculăm eficient?

- $c += x_i y_i$
 - n^2 produse scalare (se pot realiza în paralel)
 - cum exploatăm eficient cache-ul?
 - calcul pe blocuri, nu pe linii sau coloane

```

for (ii = 0; ii < SIZE; ii += BLOCK_SIZE)
  for (kk = 0; kk < SIZE; kk += BLOCK_SIZE)
    for (jj = 0; jj < SIZE; jj += BLOCK_SIZE)
      maxi = min(ii + BLOCK_SIZE, SIZE);
      maxk = min(kk + BLOCK_SIZE, SIZE);
      for (k = kk; k < maxk; k++)
        maxj = min(jj + BLOCK_SIZE, SIZE);
        for (j = jj; j < maxj; j++)
          C[i][j] = C[i][j] + A[i][k] * B[k][j];

```

8. Considerăm că rulăm același program pe două sisteme diferite. Pe sistemul X, programul execută 80 de instrucțiuni aritmetice/logice, 40 de operații citire/scriere memorie și 25 de instrucțiuni de branch/salt. Pe sistemul Y, programul execută 50 de instrucțiuni aritmetice/logice, 50 de operații citire/scriere memorie și 40 de instrucțiuni de branch/salt. Pe ambele sisteme, operațiile aritmetice/logice au 1 ciclu, operațiile de citire/scriere au 3 cicli iar operațiile de branch/salt au 5 cicli. Cerințe:
- (a) calculați ponderea tipurilor de operații pentru fiecare sistem;
 - (b) calculați numărul mediu de cicli de ceas pe instrucțiune pentru fiecare sistem;
 - (c) presupunem că frecvența sistemului Y este cu 20% mai ridicată decât cea a sistemului X. Care sistem execută programul mai repede?

PERFORMANȚA SISTEME, EX. 8

a) $p = 80/145, p = 40/145$ și $p = 25/145$ pentru sistemul X

$p = 50/140, p = 50/140$ și $p = 40/140$ pentru sistemul Y

b) 2.24 pentru sistemul X și 2.86 pentru sistemul Y

c) cpu time pe sistemul X = $145 \times 2.24 / f$

cpu time pe sistemul Y = $140 \times 2.86 / (1.2 \times f)$

accelerarea este raportul valorilor: $Y / X \approx 1.03$ (sistemul X este cu 3% mai rapid decât Y)

PUTERILE LUI 2

0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
13	8192
14	16384
15	32768
16	65536
17	131072
18	262144
19	524288
20	1048576
21	2097152
22	4194304
23	8388608
24	16777216
25	33554432
26	67108864
27	134217728
28	268435456
29	536870912
30	1073741824
31	2147483648
32	4294967296

	9	10 (A)	11 (B)	12 (C)	13 (D)	14 (E)	15 (F)	
Puteri a lui 16	1	2	3	4	5	6	7	8
4.	65536	131072	262144	524288	1048576	2097152	4194304	8388608
3.	4096	8192	16384	32768	65536	131072	262144	524288
2.	256	512	768	1024	1280	1536	1792	2048
1.	16	32	48	64	80	96	112	128
0.	1	2	3	4	5	6	7	8

	9	10	11	12	13	14	15
589824	655360	7208896	786432	851968	917504	983648	
36864	40960	45056	49152	53248	57344	61440	
2304	2560	2816	3072	3328	3584	3840	
144	160	176	192	208	224	240	
9	10	11	12	13	14	15	

Performanta sistemului

exemplu: înmulțirea a două numere aflate în memorie

$$\text{MEM_LOC2} = \text{MEM_LOC1} \times \text{MEM_LOC2}$$

- Complex Instruction Set Computers
 - **MUL** MEM_LOC_1, MEM_LOC_2
- Reduced Instruction Set Computers
 - **LOAD** MEM_LOC1, R1
 - **LOAD** MEM_LOC2, R2
 - **MUL** R1, R2
 - **STORE** R2, MEM_LOC2

un criteriu nou de performanță: cantitatea de energie consumată
extrem de important pentru dispozitive pe baterie:

- laptop-uri
- tablete
- telefoane inteligente
- sisteme embedded, sisteme Internet of Things (IoT)

toate aceste dispozitive își determină dinamic ciclul de ceas
pentru a balansa consum de energie, răcire și performanță

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

Performanta sistemului

DIMENSIUNEA FIŞIERELOR

- este un indicator destul de bun pentru numărul de instrucțiuni
- compilatorul/SO-ul mereu adaugă
 - header
 - informație de mediul de execuție
 - etc.
- pentru gcc, flagul de optimizare pentru dimensiune cod este -Os

```
#include <stdio.h>
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

18k

```
#include <unistd.h>
#include <sys/syscall.h>

void _start()
{
    const char msg [] = "Hello World!";
    syscall(SYS_write, 0, msg, sizeof(msg)-1);
    syscall(SYS_exit, 0);
}
```

14k

```
.data
helloWorld: .asciz "Hello World!\n"

.text
.globl _start
_start:
    mov $4, %eax
    mov $1, %ebx
    mov $helloWorld, %ecx
    mov $14, %edx
    int $0x80

    mov $1, %eax
    mov $0, %ebx
    int $0x80
```

8k

PERFORMANȚA VS. SECURITATE

- de cele mai multe ori există o balanță între performanță și elemente de securitate în software
- de ce?
 - verificări suplimentare de input de la user
 - verificări suplimentare la dimensiunea variabilelor/bufferelor
 - randomizarea adreselor de memorie este o metodă care sporește securitatea sistemului
 - criptare/decriptare date

PERFORMANȚA TOTALĂ

- este această cantitate una deterministă?
 - dacă rulez același program (executabil) pe același sistem de calcul (arhitectura completă) cu exact aceleași setări (de exemplu folosește SIMD, etc.) și stare inițială (memorie RAM, cache inițializată la fel), am același timp de rulare?
 - NU
 - de ce?
 - nu uitați de detectarea și corectarea erorilor
 - deci, sistemul vostru de calcul nu face exact aceleași operații niciodată (cu probabilitate mare)